



ระบบ IBM - iSeries

ฐานข้อมูล DB2 Universal Database for iSeries -
การโปรแกรม SQL

เวอร์ชัน 5 รีลีส 4





ระบบ IBM - iSeries

ฐานข้อมูล DB2 Universal Database for iSeries -
การโปรแกรม SQL

เวอร์ชัน 5 รีลีส 4

หมายเหตุ

ก่อนใช้ข้อมูลและผลิตภัณฑ์ที่ข้อมูลนี้สนับสนุน, โปรดอ่านข้อมูลใน “คำประกาศ”, ในหน้า 361.

ฉบับพิมพ์ครั้งที่เจ็ด (กุมภาพันธ์ 2006)

ข้อมูลในฉบับนี้ประยุกต์ใช้กับเวอร์ชัน 5, ซีรีส์ 4, โมดิฟิเคชัน 0 ของ IBM i5/OS (หมายเลขผลิตภัณฑ์ 5722-SS1) และกับซีรีส์ และ โมดิฟิเคชันที่ตามมาในภายหลังทั้งหมด จนกว่าจะถูกแจ้งไว้ในฉบับใหม่. เวอร์ชันนี้สามารถทำงานได้กับเครื่องที่ใช้ชุดคำสั่งแบบ reduced instruction set computer (RISC) เฉพาะในบางรุ่นและไม่สามารถใช้งานกับเครื่องรุ่นที่เป็น CISC.

© ลิขสิทธิ์ของ International Business Machines Corporation 1998, 2006. สงวนลิขสิทธิ์ทั้งหมด.

สารบัญ

การโปรแกรม SQL	1
มีอะไรใหม่ใน V5R4	2
PDF ที่สามารถพิมพ์ได้	2
บทนำสู่ DB2 UDB for iSeries Structured Query Language	3
แนวคิด SQL	3
SQL อ็อบเจกต์	9
อ็อบเจกต์แอ็พพลิเคชันโปรแกรม	13
Data definition language (DDL)	17
สร้างแบบแผน (schema)	17
สร้างตาราง	17
การสร้างตารางโดยใช้ LIKE	22
การสร้างตารางโดยใช้ AS	22
การสร้างและเปลี่ยนตาราง materialized query	23
การประกาศตารางชั่วคราวแบบโกลบอล	24
การสร้างและเปลี่ยน identity column	25
การใช้ ROWID	26
การสร้าง และการใช้ลำดับ	26
การสร้างเลเบลอธิบายโดยใช้คำสั่ง LABEL ON	29
การอธิบาย SQL อ็อบเจกต์โดยใช้ COMMENT ON	30
การเปลี่ยน definition ตาราง	31
การสร้างและการใช้งานชื่อ ALIAS	34
การสร้างและใช้มุมมอง	35
การเพิ่มตรรกษณ์	40
แคตตาล็อกในการออกแบบฐานข้อมูล	40
การลบอ็อบเจกต์ฐานข้อมูล	42
ภาษาสำหรับการดำเนินการข้อมูล	42
การดึงข้อมูลโดยใช้คำสั่ง SELECT	42
การแทรกแถวโดยใช้คำสั่ง INSERT	94
การเปลี่ยนข้อมูลในตารางโดยใช้คำสั่ง UPDATE	100
การลบแถวออกจากตารางโดยใช้คำสั่ง DELETE	106
การใช้เค็ยวรี่ย่อย	110
ลำดับการเรียง และ normalization ใน SQL	119
ลำดับการจัดเรียงที่ใช้กับ ORDER BY และการเลือก แถว	120
ลำดับการจัดเรียงและมุมมอง	123
ลำดับการจัดเรียงและคำสั่ง CREATE INDEX	124
ลำดับการจัดเรียงและข้อจำกัด	124
ลำดับการจัดเรียง ICU	124
Normalization	126
การปกป้องข้อมูล	126
การรักษาความปลอดภัยสำหรับ SQL อ็อบเจกต์	126

Data integrity	128
รูทีน	143
สตอร์โปรซีเจอร์	143
การใช้ user-defined functions (UDFs)	180
ทริกเกอร์	212
การดีบักรูทีน SQL	225
การปรับปรุงประสิทธิภาพของโปรซีเจอร์ และฟังก์ชัน การประมวลผลชนิดข้อมูลพิเศษ	226 229
การใช้ large objects (LOBs)	229
การใช้ user-defined distinct types (UDT)	241
ตัวอย่างการใช้ UDT, UDF, และ LOB	249
การใช้ DataLink	252
การใช้คำสั่ง SQL ในสภาพแวดล้อมที่ต่างกัน	256
การใช้เคอร์เซอร์	256
แอ็พพลิเคชัน Dynamic SQL	271
การใช้ SQL แบบ dynamic ผ่านไคลเอ็นต์อินเทอร์เน็ตเฟส การใช้ SQL แบบโต้ตอบ	289 291
การใช้ตัวประมวลผลคำสั่ง SQL	304
ฟังก์ชันของฐานข้อมูลเชิงสัมพันธ์แบบกระจายและ SQL DB2 UDB for iSeries การสนับสนุนฐานข้อมูลเชิง สัมพันธ์แบบกระจาย	309 310
DB2 UDB for iSeries โปรแกรมตัวอย่างฐานข้อมูลเชิง สัมพันธ์แบบกระจาย	310 312
การสนับสนุนการใช้งานแพ็คเกจของ SQL	312
ข้อควรพิจารณาเกี่ยวกับ CCSID สำหรับ SQL	316
การจัดการการเชื่อมต่อและ activation group	317
การสนับสนุนแบบกระจาย	322
หน่วยการทำงานแบบกระจาย	330
ไดเรกทอรีโปรแกรม application requester	333
การรับมือกับปัญหา	334
ข้อควรพิจารณาสำหรับโปรซีเจอร์ของ DRDA ที่บันทึก ไว้	334 335
ข้อมูลอ้างอิง	335
DB2 UDB for iSeries ตารางตัวอย่าง	335
DB2 UDB for iSeries คำอธิบายของคำสั่ง CL	359
ภาคผนวก. คำประกาศ	361
Programming Interface Information	363
เครื่องหมายการค้า	363
ข้อกำหนดและเงื่อนไข	364

การโปรแกรม SQL

หัวข้อเหล่านี้จะอธิบายถึงการนำ iSeries™ Structured Query Language (SQL) ของเซิร์ฟเวอร์ DB2® UDB for iSeries ไปปฏิบัติและไลเซนส์โปรแกรม DB2 UDB Query Manager and SQL Development Kit เวอร์ชัน 5.

ตัวอย่างคำสั่ง SQL ที่แสดงไว้ในหัวข้อนี้อิงจากตารางตัวอย่าง โดยถือว่ามีลักษณะดังนี้:

- ตัวอย่างดังกล่าวแสดงไว้ในสภาวะแวดล้อม SQL แบบโต้ตอบ หรือถูกบันทึกไว้ใน ILE C หรือใน COBOL. ใช้ EXEC SQL และ END-EXEC เพื่อคั่นคำสั่ง SQL ในโปรแกรม COBOL.
- SQL แต่ละตัวอย่างแสดงไว้บนบรรทัดต่างๆ, โดยคำสั่งแต่ละคำสั่งจะอยู่คนละบรรทัดกัน.
- คีย์เวิร์ด SQL จะถูกไฮไลต์ไว้.
- ชื่อของตารางตัวอย่างนี้ใช้แบบแผน CORPDATA. สำหรับชื่อตารางซึ่งไม่พบในตารางตัวอย่าง ควรใช้แบบแผนที่คุณสร้างขึ้น.
- คอลัมน์ที่มีการคำนวณจะอยู่ในวงเล็บ, (), และวงเล็บ, [].
- มีการใช้หลักการตั้งชื่อ SQL.
- มีการใช้อีอ็อปชันพีคอมไพเลอร์ APOST และ APOSTSQL แม้ว่าจะไม่ใช้อีอ็อปชันดีพอลต์ใน COBOL. literal ของสตริงอักขระภายในคำสั่ง SQL และคำสั่งภาษาโฮสต์ถูกคั่นด้วยเครื่องหมายอัญประกาศเดี่ยว (').
- มีการใช้การเรียงลำดับ *HEX, เว้นแต่จะมีการแจ้งเป็นอย่างอื่น.

เมื่อใดก็ตามที่ตัวอย่างแตกต่างไปจากสมมติฐานเหล่านี้, จะมีการแจ้งไว้.

เนื่องจากหัวข้อนี้มีไว้สำหรับแอสพลีเคชันโปรแกรมเมอร์, ตัวอย่างส่วนใหญ่จึงแสดงไว้ในลักษณะที่บันทึกไว้ในแอสพลีเคชันโปรแกรม. อย่างไรก็ตาม, ตัวอย่างจำนวนมาก สามารถทำการเปลี่ยนแปลงได้เล็กน้อยและรันแบบโต้ตอบด้วยการใช้ SQL แบบโต้ตอบ. ไวยากรณ์ของคำสั่ง SQL, เมื่อใช้ SQL แบบโต้ตอบ, จะแตกต่างเล็กน้อยจากรูปแบบคำสั่งเดียวกันเมื่อใส่อยู่ในโปรแกรม.

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

สิ่งอ้างอิงที่เกี่ยวข้อง

“DB2 UDB for iSeries ตารางตัวอย่าง” ในหน้า 335

หัวข้อนี้ประกอบด้วยตารางตัวอย่างที่ใช้อ้างอิง และใช้ในหัวข้อนี้ รวมทั้งหัวข้อ SQL Reference.

ข้อมูลที่เกี่ยวข้อง

Embedded SQL programming

การอ้างอิง SQL

มีอะไรใหม่ใน V5R4

หัวข้อนี้เน้นถึงการเปลี่ยนแปลงที่เกิดกับหัวข้อ V5R4.

มีการเพิ่มหัวข้อใหม่ดังนี้:


- “การใช้เคียวรีแบบเรียกซ้ำ” ในหน้า 75
- “ตัวอย่าง: คำสั่ง Select โดยใช้ SQL descriptor ที่ถูกจัดสรรแล้ว” ในหน้า 285
- “การใช้คำกำหนด OLAP” ในหน้า 63
- “ทริกเกอร์ INSTEAD OF SQL” ในหน้า 216

การเปลี่ยนแปลงและการอัปเดตเพียงเล็กน้อยมีเพิ่มเข้าไปในหัวข้อต่อไปนี้:

- ข้อมูลเกี่ยวกับ SQL descriptor ที่ถูกจัดสรรแล้วซึ่งเพิ่มเข้าไปในหัวข้อ “ตัวทำเครื่องหมายพารามิเตอร์” ในหน้า 288, “SQL descriptor area” ในหน้า 277, “คำสั่ง SELECT แบบ varying-list” ในหน้า 276 และ “การประมวลผลคำสั่ง SELECT และการใช้ descriptor” ในหน้า 275.
- เรจิสเตอร์พิเศษที่ถูกเพิ่มเข้าไปใน “เรจิสเตอร์พิเศษในคำสั่ง SQL” ในหน้า 55.
- การเปลี่ยนแปลงที่เกิดขึ้นใน “ประเภทของคำสั่ง SQL” ในหน้า 6.
- คำสั่งเพิ่มเติมที่สามารถใช้กับตัวประมวลผลคำสั่ง SQL ที่ถูกเพิ่มลงใน “การใช้ตัวประมวลผลคำสั่ง SQL” ในหน้า 304.

วิธีการดูหัวข้อมีอะไรใหม่ หรือที่เปลี่ยนแปลง

เพื่อช่วยให้คุณให้ดูที่ซึ่งมีการเปลี่ยนแปลงทางด้านเทคนิค, ข้อมูลนี้จะใช้:

- รูปภาพ  เพื่อทำเครื่องหมายว่า ที่ใดคือข้อมูลใหม่ หรือข้อมูลที่เปลี่ยนแปลง.
- รูปภาพ  เพื่อทำเครื่องหมายว่า ที่ใดคือจุดสิ้นสุดของข้อมูลใหม่ หรือข้อมูลที่เปลี่ยนแปลง.

หากต้องการค้นหาข้อมูลเกี่ยวกับหัวข้อมีอะไรใหม่ หรือที่เปลี่ยนแปลงในวิธีสั้น, โปรดดู บันทึกไปยังผู้ใช้.

PDF ที่สามารถพิมพ์ได้

ใช้เพื่อดู และพิมพ์ข้อมูลนี้ในรูปแบบของ PDF.


ถ้าต้องการดู หรือดาวน์โหลดเอกสารนี้ในเวอร์ชัน PDF, ให้เลือก SQL Programming (มีขนาดประมาณ 3554 KB).

การบันทึกไฟล์ PDF

ถ้าต้องการบันทึกไฟล์ PDF ลงที่เวิร์กสเตชันของคุณเพื่อนำมาอ่านหรือพิมพ์ภายหลัง:

1. กดปุ่มขวาของเมาส์ตรงไฟล์ PDF ในบราวเซอร์ของคุณ (กดปุ่มขวาที่ลิงก์ข้างบน).
2. คลิกที่อ็อปชันที่จะบันทึก PDF แบบโลคัล.
3. นำทางไปจนถึงไดเรกทอรีที่คุณต้องการบันทึกไฟล์ PDF.
4. เลือก Save.

การดาวน์โหลด Adobe Reader

คุณจำเป็นต้องติดตั้ง Adobe Reader บนระบบของคุณ เพื่อดู หรือพิมพ์ไฟล์ PDF เหล่านี้. คุณสามารถดาวน์โหลดโปรแกรมได้จาก เว็บไซต์ของ Adobe (www.adobe.com/products/acrobat/readstep.html) .

บทนำสู่ DB2 UDB for iSeries Structured Query Language

หัวข้อเหล่านี้อธิบายถึงการนำ Structured Query Language (SQL) ของเซิร์ฟเวอร์ iSeries ไปใช้โดยใช้ DB2 UDB for iSeries และไลเซนส์โปรแกรม DB2 UDB Query Manager and SQL Development Kit.

SQL จัดการข้อมูลซึ่งอยู่ในแบบจำลองข้อมูลเชิงสัมพันธ์. สามารถนำคำสั่ง SQL ฝังลงในภาษาชั้นสูง, หรือนำมาจัดเตรียมแบบไดนามิกก่อนรัน หรือนำมารันแบบโต้ตอบ. สำหรับข้อมูลเพิ่มเติมใน SQL, ให้ดู Embedded SQL.

SQL จะประกอบด้วยคำสั่งและข้อความ ที่อธิบายถึงสิ่งที่คุณต้องดำเนินการกับข้อมูลในฐานข้อมูลและเงื่อนไขต่างๆ ที่ทำให้คุณต้องการดำเนินการดังกล่าว.

SQL สามารถเรียกใช้ข้อมูลในฐานข้อมูลเชิงสัมพันธ์แบบรีโมต, โดยการใช้ IBM® Distributed Relational Database Architecture* (DRDA*).

สิ่งอ้างอิงที่เกี่ยวข้อง

“ฟังก์ชันของฐานข้อมูลเชิงสัมพันธ์แบบกระจายและ SQL” ในหน้า 309

ฐานข้อมูลเชิงสัมพันธ์แบบกระจาย ประกอบไปด้วยชุด SQL อ็อบเจกต์ที่กระจายอยู่บนระบบคอมพิวเตอร์ที่เชื่อมต่อกันและกัน.

โปรแกรมมิงฐานข้อมูลแบบกระจาย

แนวคิด SQL

DB2 UDB for iSeries SQL ประกอบด้วยส่วนหลักๆ ดังต่อไปนี้.

- ตัวสนับสนุนรันไทม์ SQL

รันไทม์ SQL จะวิเคราะห์และรันคำสั่ง SQL. การสนับสนุนนี้คือ ส่วนหนึ่งของไลเซนส์โปรแกรม i5/OS™, ที่อนุญาตให้แอปพลิเคชันที่มีคำสั่ง SQL สามารถรันได้บนระบบ โดยไม่ต้องติดตั้งไลเซนส์โปรแกรม DB2 UDB Query Manager and SQL Development Kit.

- SQL 프리คอมไพเลอร์

SQL 프리คอมไพเลอร์สนับสนุนคำสั่ง SQL ที่ฝังอยู่แบบ 프리คอมไพเลอร์ในภาษาไฮสท. ภาษาต่อไปนี้ เป็นภาษาที่ได้รับการสนับสนุน:

- ILE C
- ILE C++ สำหรับ iSeries
- ILE COBOL
- COBOL สำหรับ iSeries
- iSeries PL/I
- RPG III (ส่วนของ RPG สำหรับ iSeries)

– ILE RPG

SQL 프리คอมไพเลอร์ของภาษาโฮสต์จะเตรียมแ็พพลิเคชันโปรแกรมที่มีคำสั่ง SQL. คอมไพเลอร์ภาษาโฮสต์จึงคอมไพล์ซอร์สโปรแกรมโฮสต์แบบ 프리คอมไพล์. สำหรับข้อมูลเพิ่มเติมเกี่ยวกับ 프리คอมไพเลอร์, โปรดดูหัวข้อ การเตรียมและการรันโปรแกรมด้วยคำสั่ง SQL ใน Embedded SQL Programming. ตัวสนับสนุน 프리คอมไพเลอร์คือส่วนหนึ่งของไลเซนส์โปรแกรม DB2 UDB Query Manager and SQL Development Kit.

- อินเทอร์เน็ตแบบโต้ตอบของ SQL

อินเทอร์เน็ตแบบโต้ตอบของ SQL ทำให้คุณสามารถสร้างและรันคำสั่ง SQL. หากต้องการทราบข้อมูลเพิ่มเติมเกี่ยวกับ SQL แบบโต้ตอบ, โปรดดูที่ “การใช้ SQL แบบโต้ตอบ” ในหน้า 291. SQL แบบโต้ตอบคือส่วนหนึ่งของไลเซนส์โปรแกรม DB2 UDB Query Manager and SQL Development Kit.

- Run SQL Scripts

หน้าต่าง Run SQL Scripts ใน iSeries Navigator จะให้คุณสร้าง, แก้ไข, รัน, และแก้ปัญหาเบื้องต้นเกี่ยวกับสคริปต์คำสั่ง SQL. Run SQL Scripts เป็นส่วนหนึ่งของ iSeries Navigator.

- คำสั่ง Run SQL Statements CL

RUNSQLSTM จะช่วยในการรันชุดคำสั่ง SQL, ซึ่งถูกเก็บไว้ในไฟล์ต้นฉบับ. โปรดดูที่ “การใช้ตัวประมวลผลคำสั่ง SQL” ในหน้า 304 สำหรับข้อมูลเพิ่มเติมเกี่ยวกับคำสั่ง Run SQL Statements.

- DB2 Query Manager for iSeries

DB2 Query Manager for iSeries มีอินเทอร์เน็ตแบบโต้ตอบที่ทำงานโดยคำสั่ง ซึ่งอนุญาตให้คุณสร้างข้อมูล, เพิ่มข้อมูล, รักษาข้อมูล, และรันรายงานบนฐานข้อมูลได้. Query Manager คือส่วนหนึ่งของไลเซนส์โปรแกรม DB2 UDB Query Manager and SQL Development Kit. หากต้องการทราบข้อมูลเพิ่มเติม, โปรดดูที่หนังสือ Query Manager Use.

- SQL REXX Interface

SQL REXX interface จะทำให้คุณรันคำสั่ง SQL ในโปรแกรมรีเพล็กซ์ได้. สำหรับข้อมูลเพิ่มเติมเกี่ยวกับการใช้คำสั่ง SQL ในโปรแกรมรีเพล็กซ์, โปรดดูหัวข้อ Code SQL statements ในแ็พพลิเคชัน REXX ใน Embedded SQL Programming.

- SQL Call Level Interface

DB2 UDB for iSeries สนับสนุน SQL Call Level Interface. ซึ่งจะยอมให้ผู้ใช้ภาษา ILE ใดๆ สามารถเรียกใช้ฟังก์ชัน SQL โดยตรงผ่านการเรียกใช้ เซอร์วิสโปรแกรมที่จัดเตรียมโดยระบบ. เมื่อใช้ SQL Call Level Interface, ผู้ใช้สามารถใช้งานฟังก์ชัน SQL ทั้งหมดได้โดยไม่ต้องคอมไพล์. นี่คือชุดคำสั่งมาตรฐานใช้เรียกเพื่อเตรียมข้อความ SQL, รันข้อความ SQL, ดึงแถวของข้อมูลออกมา, และทำแม้กระทั่งฟังก์ชันระดับสูง เช่นการเรียกใช้แคตตาล็อก และการเชื่อมโยงตัวแปรโปรแกรมไปยังคอลัมน์ของเอาต์พุต.

สำหรับรายละเอียดที่สมบูรณ์ของฟังก์ชันทั้งหมดที่มีอยู่ และไวยากรณ์, โปรดดูหัวข้อ SQL call level interface (ODBC) ในส่วนของฐานข้อมูลของ iSeries Information Center.

- QSQRCE API

application program interface (API) นี้จะทำให้ SQL สามารถทำงานแบบ dynamic ได้. คำสั่ง SQL สามารถจัดเตรียมลงใน SQL แพ็กเกจและรันงานโดยการใช้ API นี้. คำสั่งที่ถูกจัดเตรียมเป็นแพ็กเกจด้วย API นี้จะยังคงอยู่จนกระทั่งแพ็กเกจหรือคำสั่งถูกลบออกไป. สำหรับข้อมูลเพิ่มเติมเกี่ยวกับ QSQRCE API, โปรดดูหัวข้อ QSQRCE. สำหรับข้อมูลทั่วไปเกี่ยวกับ API, โปรดดูหัวข้อ i5/OS API.

- QSQCHKS API

ไวยากรณ์ API จะตรวจสอบคำสั่ง SQL. สำหรับข้อมูลเพิ่มเติมเกี่ยวกับ QSQCHKS API, โปรดดูหัวข้อ QSQCHKS. สำหรับข้อมูลทั่วไปเกี่ยวกับ API, โปรดดูหัวข้อ i5/OS API.

- DB2 Multisystem

คุณลักษณะของระบบปฏิบัติการจะทำให้ข้อมูลของคุณกระจายอย่างทั่วถึงบนเซิร์ฟเวอร์หลายๆ ตัว. สำหรับข้อมูลเพิ่มเติมเกี่ยวกับ DB2 Multisystem, โปรดดูหัวข้อ DB2 Multisystem.

- DB2 UDB Symmetric Multiprocessing

คุณลักษณะนี้ของระบบปฏิบัติการจะมี query optimizer และวิธีการเพิ่มเติมในการเรียกข้อมูลด้วยการประมวลผลแบบขนาน. Symmetric multiprocessing (SMP) เป็นรูปแบบของการเข้าใช้งานแบบขนานบนระบบเดี่ยว ที่ซึ่งโพรเซสเซอร์หลายตัว (CPU และตัวประมวลผล I/O) ที่แบ่งใช้งานรีซอร์สของหน่วยความจำ และ ดิสก์ทำงานร่วมกัน เพื่อให้ได้มาซึ่งผลลัพธ์อย่างเร็วในตอนสุดท้าย. การประมวลผลแบบขนานหมายความว่าตัวจัดการฐานข้อมูลสามารถรองรับการสืบค้นจากตัวประมวลผลระบบได้มากกว่าหนึ่งตัว (หรือทั้งหมด) พร้อมๆ กัน. โปรดดูหัวข้อ การควบคุมการประมวลผลแบบขนาน สำหรับเคียวรีในหัวข้อ ประสิทธิภาพการทำงานของฐานข้อมูลและการใช้งานเคียวรีให้ได้ผลดีที่สุด สำหรับรายละเอียดเกี่ยวกับวิธีการควบคุมการประมวลผลแบบขนาน.

ฐานข้อมูลเชิงสัมพันธ์ SQL และศัพท์เฉพาะระบบ

ในแบบจำลองข้อมูลเชิงสัมพันธ์, จะถือว่าข้อมูลทั้งหมดมีอยู่ในตาราง. DB2 UDB for iSeries อ็อบเจ็กต์ถูกสร้างและรักษาไว้ในฐานะเป็นอ็อบเจ็กต์ระบบ.

ตารางต่อไปนี้จะแสดงความสัมพันธ์ระหว่างศัพท์เฉพาะของระบบ และศัพท์เฉพาะของฐานข้อมูลเชิงสัมพันธ์.

ตารางที่ 1. ความสัมพันธ์ของศัพท์เฉพาะระบบกับศัพท์เฉพาะ SQL

ศัพท์เฉพาะระบบ	ศัพท์เฉพาะ SQL
ไลบรารี. จัดกลุ่มอ็อบเจ็กต์ที่สัมพันธ์กันและให้คุณสามารถค้นหาอ็อบเจ็กต์ได้ด้วยชื่อของอ็อบเจ็กต์.	แบบแผน. ประกอบด้วยไลบรารี, เจอร์นัล, journal receiver, แค็ตตาล็อก SQL, และอาจรวมถึงพจนานุกรมข้อมูล. แบบแผนจะจัดกลุ่มอ็อบเจ็กต์ที่สัมพันธ์กันและให้คุณค้นหาอ็อบเจ็กต์ได้ด้วยชื่อ.
ไฟล์ฟิสิคัล. เซ็ตของเร็คคอร์ด. เร็คคอร์ด. เซ็ตของฟิลด์.	ตาราง. เซ็ตของคอลัมน์และแถว. แถว. ส่วนที่เป็นแนวนอนของตาราง ซึ่งเซตของคอลัมน์เรียงลำดับอยู่.
ฟิลด์. อักขระหนึ่งตัวขึ้นไปของข้อมูลที่เกี่ยวข้องในประเภทข้อมูลหนึ่งประเภท.	คอลัมน์. ส่วนที่เป็นแนวตั้งของตารางประเภทข้อมูลหนึ่งประเภท.
ไฟล์ลอจิคัล. กลุ่มย่อยของฟิลด์และเร็คคอร์ดของไฟล์ฟิสิคัลหนึ่งไฟล์ขึ้นไป.	มุมมอง. เซ็ตย่อยของคอลัมน์และแถวของตารางหนึ่งตารางขึ้นไป.
SQL แพ็กเกจ. ชนิดของอ็อบเจ็กต์ ที่มีการใช้ในการรันคำสั่ง SQL. โปรไฟล์ผู้ใช้	แพ็กเกจ. ชนิดของอ็อบเจ็กต์ ที่มีการใช้ในการรันคำสั่ง SQL. ชื่อหรือรหัสที่ได้รับอนุญาต

สิ่งอ้างอิงที่เกี่ยวข้อง

โปรแกรมมิ่งฐานข้อมูลแบบกระจาย

SQL และหลักการตั้งชื่อของระบบ

มีหลักการตั้งชื่ออยู่สองแบบที่สามารถนำมาใช้ในโปรแกรมมิ่ง DB2 UDB for iSeries: ระบบ (*SYS) และ SQL (*SQL).

หลักการตั้งชื่อที่ใช้จะมีผลต่อวิธีการคัดเลือกชื่อไฟล์และตารางและศัพท์เฉพาะที่ใช้บนหน้าจอส SQL แบบโต้ตอบ. หลักการตั้งชื่อที่ใช้จะถูกเลือกโดยพารามิเตอร์รันคำสั่ง SQL หรือ, ในกรณีที่เป็น REXX, เลือกผ่านทางคำสั่ง SET OPTION. โปรดดูรายละเอียดจากการคัดเลือกชื่ออ็อบเจ็กต์ที่ไม่ถูกต้องในคู่มืออ้างอิง SQL.

การตั้งชื่อระบบ (*SYS)

ในหลักการตั้งชื่อระบบ, ตาราง และ SQL อ็อบเจ็กต์อื่นๆ ในคำสั่ง SQL ที่ถูกต้องจะต้องอยู่ในรูปแบบ:
แบบแผน/ตาราง

การตั้งชื่อ SQL (*SQL)

ในหลักการตั้งชื่อ SQL, ตารางและ SQL อ็อบเจ็กต์อื่นๆ ในคำสั่ง SQL ที่ถูกต้องจะต้องอยู่ในรูปแบบ:
แบบแผน.ตาราง

ประเภทของคำสั่ง SQL

มีประเภทของคำสั่ง SQL พื้นฐานอยู่หลายประเภท. ซึ่งคำสั่งเหล่านั้นจะถูกแสดงไว้ที่นี้ตามฟังก์ชัน.

- | • คำสั่งแบบแผน SQL, ซึ่งรู้จักกันว่า คำสั่ง data definition language (DDL)
- | • ข้อมูล SQL และคำสั่งการเปลี่ยนข้อมูล, ซึ่งรู้จักกันว่า คำสั่ง data manipulation language (DML)
 - คำสั่ง SQL แบบ Dynamic
- | • คำสั่งภาษาโฮสต์ SQL แบบฝัง

| คำสั่งแบบแผน SQL

| ALTER SEQUENCE
| ALTER TABLE
| COMMENT ON
| CREATE ALIAS
| CREATE DISTINCT TYPE
| CREATE FUNCTION
| CREATE INDEX
| CREATE PROCEDURE
| CREATE SCHEMA
| CREATE SEQUENCE
| CREATE TABLE
| CREATE TRIGGER
| CREATE VIEW
| DROP ALIAS
| DROP DISTINCT TYPE
| DROP FUNCTION
| DROP INDEX
| DROP PACKAGE
| DROP PROCEDURE
| DROP SEQUENCE
| DROP SCHEMA
| DROP TABLE
| DROP TRIGGER
| DROP VIEW
| GRANT DISTINCT TYPE
| GRANT FUNCTION
| GRANT PACKAGE
| GRANT PROCEDURE
| GRANT SEQUENCE
| GRANT TABLE
| LABEL ON
| RENAME
| REVOKE DISTINCT TYPE
| REVOKE FUNCTION
| REVOKE PACKAGE
| REVOKE PROCEDURE
| REVOKE SEQUENCE
| REVOKE TABLE
|

| คำสั่งการเปลี่ยนข้อมูล SQL

| DELETE
| INSERT
| UPDATE
|
|

| คำสั่งข้อมูล SQL

CLOSE
DECLARE CURSOR
DELETE
FETCH
FREE LOCATOR
HOLD LOCATOR
INSERT
LOCK TABLE
OPEN
REFRESH TABLE
SELECT INTO
SET variable
UPDATE
VALUES INTO

| คำสั่งการเชื่อมต่อ SQL

CONNECT
DISCONNECT
RELEASE
SET CONNECTION

| คำสั่งรายการ SQL
 | COMMIT
 | RELEASE SAVEPOINT
 | ROLLBACK
 | SAVEPOINT
 | SET TRANSACTION
 |
 |

คำสั่งเซสชัน SQL
 DECLARE GLOBAL TEMPORARY TABLE
 SET CURRENT DEGREE
 SET ENCRYPTION PASSWORD
 SET PATH
 SET SCHEMA
 SET SESSION AUTHORIZATION

| คำสั่ง SQL แบบ Dynamic
 | ALLOCATE DESCRIPTOR
 | DEALLOCATE DESCRIPTOR
 | DESCRIBE
 | DESCRIBE INPUT
 | DESCRIBE TABLE
 | EXECUTE
 | EXECUTE IMMEDIATE
 | GET DESCRIPTOR
 | PREPARE
 | SET DESCRIPTOR
 |
 |

คำสั่งภาษาโฮสต์ SQL แบบฝัง
 BEGIN DECLARE SECTION
 DECLARE PROCEDURE
 DECLARE STATEMENT
 DECLARE VARIABLE
 END DECLARE SECTION
 GET DIAGNOSTICS
 INCLUDE
 SET OPTION
 SET RESULT SETS
 SIGNAL
 WHENEVER

| SQL control statement
 | CALL
 |

ข้อความ SQL สามารถปฏิบัติกรกับอ็อบเจ็กต์ที่ถูกสร้างโดย SQL เช่นเดียวกับ ฟิสิคัลไฟล์ชนิด externally described และ โลจิคัลไฟล์ชนิด single-format, ถึงแม้ว่า มันจะอยู่ในรูปแบบ SQL . คำสั่ง SQL จะไม่อ้างอิงถึงคำนิยามในพจนานุกรม IDDU สำหรับไฟล์ที่อธิบายด้วยโปรแกรม. ไฟล์ที่อธิบายด้วยโปรแกรมจะปรากฏเป็นตารางที่มีคอลัมน์เดี่ยว.

หลักการที่เกี่ยวข้อง

“Data definition language (DDL)” ในหน้า 17

Data definition language (DDL) คือส่วนของ SQL ที่อนุญาตให้คุณสร้าง, เปลี่ยน, และทำลายอ็อบเจ็กต์ฐานข้อมูล. อ็อบเจ็กต์ฐานข้อมูลเหล่านี้ประกอบด้วยแบบแผน, ตาราง, มุมมอง, ลำดับ, แคตตาล็อก, ตรรกษนี้, และ aliase.

“ภาษาสำหรับการดำเนินการข้อมูล” ในหน้า 42

ภาษาสำหรับการดำเนินการข้อมูล (DML) คือส่วนของ SQL ที่อนุญาตให้คุณดำเนินการ หรือควบคุมข้อมูลของคุณได้.

สิ่งอ้างอิงที่เกี่ยวข้อง

การอ้างอิง SQL

SQL communication area (SQLCA)

SQLCA คือชุดตัวแปรที่ถูกอัปเดตเมื่อสิ้นสุดการรันคำสั่ง SQL ทุกคำสั่ง.

หลักการที่เกี่ยวข้อง

SQL communication area (SQLCA)

สิ่งอ้างอิงที่เกี่ยวข้อง

การจัดการกับคำสั่งคืนของ SQL

พื้นที่วินิจฉัย SQL

ขอบเขตการวิเคราะห์ SQL คือชุดของข้อมูลที่ดูแลโดยตัวจัดการฐานข้อมูล ที่เกี่ยวกับคำสั่ง SQL ซึ่งมีการรันงานครั้งล่าสุด. มันสามารถถูกเรียกใช้ได้จาก โปรแกรมของคุณโดยการใช้คำสั่ง GET DIAGNOSTICS SQL.

หลักการที่เกี่ยวข้อง

GET DIAGNOSTICS statement

งานที่เกี่ยวข้อง

การใช้พื้นที่วินิจฉัย SQL

SQL อ็อบเจกต์

SQL *อ็อบเจกต์* คือแบบแผน, เจอร์นัล, แคตตาล็อก, ตาราง, alias, มุมมอง, ดรรชนี, ข้อจำกัด, ทรริกเกอร์, ลำดับ, โพรซีเจอร์ที่เก็บไว้, ฟังก์ชันแบบผู้ใช้กำหนดเอง, ประเภทแบบผู้ใช้กำหนดเอง, และ SQL แพ็กเกจ. SQL จะสร้างและรักษาอ็อบเจกต์เหล่านี้ให้เป็นอ็อบเจกต์ระบบ.

แบบแผน (Schemas)

แบบแผนจะมีการจัดกลุ่มของ SQL อ็อบเจกต์แบบโลจิคัล.

แบบแผน ประกอบด้วยไลบรารี, เจอร์นัล, journal receiver, แคตตาล็อก, และอาจมีพจนานุกรมข้อมูล. การสร้าง, ย้าย, หรือเก็บตาราง, มุมมอง และอ็อบเจกต์ระบบ (เช่นโปรแกรม) อาจไว้ในไลบรารี, ระบบใดๆ ก็ได้. ไฟล์ระบบทั้งหมดอาจถูกสร้างหรือย้ายไปยังแบบแผน SQL หากแบบแผน SQL ไม่มีพจนานุกรมข้อมูล. หากแบบแผน SQL มีพจนานุกรมข้อมูลแล้ว:

- source physical ไฟล์หรือ nonsource physical ไฟล์ที่มีต้นทางที่มีรายการย่อยเดียวสามารถถูกสร้าง, ย้าย, หรือเก็บไว้ในแบบแผน SQL ได้.
- เราไม่สามารถวางโลจิคัลไฟล์ในแบบแผน SQL ได้เนื่องจากไฟล์เหล่านั้นไม่สามารถอธิบายอยู่ใน พจนานุกรมข้อมูลได้.

คุณสามารถสร้างและเป็นเจ้าของแบบแผนได้หลายรายการ. ศัพท์คำว่า *collection* อาจใช้ในความหมายเดียวกับแบบแผน (*schema*) ได้.

เจอร์นัลและตัวรับเจอร์นัล

เจอร์นัล และ *journal receiver* จะใช้บันทึกการเปลี่ยนแปลงของตารางและมุมมองในฐานข้อมูล.

เจอร์นัลและ *journal receiver* จึงใช้ในการประมวลผลคำสั่ง SQL COMMIT, ROLLBACK, SAVEPOINT, และ RELEASE SAVEPOINT. เจอร์นัล และ *journal receiver* ยังสามารถใช้งานเป็นหลักฐานการตรวจสอบ หรือใช้สำหรับ forward recovery หรือ backward recovery.

หลักการที่เกี่ยวข้อง

การทำเจอร์นัล

Commitment control

แคตตาล็อก

แคตตาล็อก SQL ประกอบด้วยชุดตาราง และมุมมอง ซึ่งอธิบายตาราง, มุมมอง, ดรรชนี, แพ็กเกจ, โพรซีเจอร์, ฟังก์ชัน, ไฟล์, ลำดับ, ทรริกเกอร์, และข้อจำกัด.

ข้อมูลนี้อยู่ในชุดตาราง cross-reference ในไลบรารี QSYS และ QSYS2. ในแบบแผน SQL จะมีชุดมุมมองซึ่งถูกสร้างขึ้นจาก ตารางแคตตาล็อกซึ่งมีข้อมูลเกี่ยวกับตาราง, มุมมอง, ดรรชนี, แพ็กเกจ, ไฟล์, และข้อจำกัดในแบบแผน.

แคตตาล็อกจะถูกสร้างขึ้นโดยอัตโนมัติเมื่อคุณสร้างแบบแผน. คุณไม่สามารถลบหรือเปลี่ยนแคตตาล็อก.

หลักการที่เกี่ยวข้อง

แคตตาล็อก SQL

ตาราง, แถว, และคอลัมน์

ตาราง เป็นการจัดการข้อมูลแบบสองด้านประกอบด้วยแถว และ คอลัมน์.

แถวคือส่วนที่เป็นแนวนอนซึ่งประกอบด้วยคอลัมน์ตั้งแต่หนึ่งคอลัมน์ขึ้นไป. คอลัมน์คือส่วนที่เป็นแนวตั้งซึ่งประกอบด้วย แถวตั้งแต่หนึ่งแถวขึ้นไปในประเภทข้อมูลหนึ่งประเภท. ข้อมูลทั้งหมดในหนึ่งคอลัมน์ต้องเป็นข้อมูลประเภทเดียวกัน. ตาราง ใน SQL คือไฟล์คัลไฟล์ที่มีคีย์ หรือไม่มีคีย์ก็ได้.

materialized query table คือตารางที่ใช้ในการเก็บ ข้อมูลที่เป็นสื่อพิมพ์ที่ซึ่งแปลงมาจากตารางต้นฉบับหนึ่งหรือหลายตาราง ที่ระบุโดย select-statement.

partitioned table คือตารางที่มีข้อมูลที่เป็นส่วนประกอบใน โคล์พาร์ติชัน (เมมเบอร์) หนึ่งหรือมากกว่าหนึ่ง.

หลักการที่เกี่ยวข้อง

ชนิดข้อมูล

สิ่งอ้างอิงที่เกี่ยวข้อง

“การสร้างและเปลี่ยนตาราง materialized query” ในหน้า 23

ตาราง materialized query เป็นตารางที่มี definition อยู่บนพื้นฐาน ของผลลัพธ์ของ query. ดังเช่น, ตาราง materialized query ปกติจะประกอบด้วยผลลัพธ์ที่คำนวณไว้ล่วงหน้า ซึ่งขึ้นอยู่กับข้อมูลที่มีอยู่ในตารางก่อนแล้ว หรือตารางที่มีพื้นฐาน ตาม definition.

DB2 Multisystem

Alias

alias คืออีกชื่อหนึ่งของตารางหรือมุมมอง.

คุณสามารถใช้ *alias* เพื่ออ้างถึงตารางหรือมุมมองในกรณีที่ทำได้. นอกจากนี้, คุณยังสามารถใช้ *alias* เพื่อรวมตารางเข้าไว้ด้วยกัน.

ข้อมูลที่เกี่ยวข้อง

Alias

มุมมอง

มุมมอง จะเป็นเหมือนตารางสำหรับแอปพลิเคชันโปรแกรม; อย่างไรก็ตาม, มุมมองจะไม่มีข้อมูลใดๆ.

มุมมองถูกสร้างขึ้นบนตารางตั้งแต่หนึ่งตารางขึ้นไป. มุมมองสามารถรองรับคอลัมน์ทั้งหมด หรือชุดย่อยของคอลัมน์ที่กำหนด, และสามารถรองรับแถวทั้งหมดหรือชุดย่อยของตารางที่กำหนดได้. คอลัมน์ในมุมมองสามารถจัดวางให้ต่างไปจากที่เป็นอยู่ในตารางที่นำคอลัมน์นั้นมาได้. มุมมองใน SQL คือรูปแบบพิเศษของโลจิคัลไฟล์ที่ไม่มีคีย์.

ข้อมูลที่เกี่ยวข้อง

มุมมอง

ดรรชนี (Index)

ดรรชนี SQL คือ ชุดข้อมูลย่อยในคอลัมน์ของตารางที่จัดเรียงตามลำดับการเรียงจากมากไปหาน้อยหรือจากน้อยไปหามากตามความเหมาะสม.

ดรรชนีแต่ละตัวมีการจัดเรียงที่แยกจากกัน. การจัดเรียงเหล่านี้ได้แก่ การเรียงลำดับ (ORDER BY clause), การจัดกลุ่ม (GROUP BY clause), และการเชื่อมโยง. ดรรชนี SQL คือ โลจิคัลไฟล์แบบมีคีย์.

ระบบจะใช้ดรรชนีเพื่อให้ดึงข้อมูลออกมาได้รวดเร็วขึ้น. คุณสามารถเลือกได้ว่าจะสร้างหรือไม่สร้างดรรชนีก็ได้. คุณสามารถสร้างดรรชนีจำนวนเท่าใดก็ได้. นอกจากนี้ คุณอาจสร้างหรือลบดรรชนีได้ตลอดเวลา. ดรรชนีจะถูกรักษาไว้โดยระบบโดยอัตโนมัติ. อย่างไรก็ตาม, เนื่องจากดรรชนีจะถูกเก็บไว้บนระบบ, ดังนั้นหากมีดรรชนีจำนวนมากจะส่งผลต่อประสิทธิภาพการทำงานของแอปพลิเคชันที่เปลี่ยนตาราง.

ข้อมูลที่เกี่ยวข้อง

ยุทธวิธีในการสร้างดรรชนี

ข้อจำกัด

ข้อจำกัด คือกฎที่ตัวจัดการฐานข้อมูลใช้.

DB2 UDB for iSeries สนับสนุนข้อจำกัดดังต่อไปนี้:

- ข้อจำกัดแบบเฉพาะ (Unique constraints)

ข้อจำกัดแบบเฉพาะ คือ กฎที่บังคับว่าค่าของคีย์จะถูกต้องก็ต่อเมื่อกำหนดเป็นค่าเฉพาะ. ข้อจำกัดแบบเฉพาะสามารถสร้างได้โดยใช้คำสั่ง CREATE TABLE และ ALTER TABLE. แม้ว่า CREATE INDEX สามารถสร้างดรรชนีแบบเฉพาะซึ่งจะไม่ซ้ำกัน, แต่ดรรชนีดังกล่าวก็ไม่ถือเป็นข้อจำกัด.

ข้อจำกัดแบบเฉพาะจะถูกนำมาบังคับใช้เมื่อมีการรันคำสั่ง INSERT และ UPDATE. ข้อจำกัด PRIMARY KEY คือ รูปแบบข้อจำกัด UNIQUE. ความแตกต่างคือ PRIMARY KEY ต้องไม่มีคอลัมน์ที่เป็น null.

- ข้อจำกัดแบบอ้างอิง (Referential constraints)

ข้อจำกัดแบบอ้างอิง คือ กฎที่บังคับว่าค่าของคีย์ foreign จะถูกต้องก็ต่อเมื่อ:

- ค่าเหล่านั้นปรากฏเป็นค่าของคีย์หลัก (parent key), หรือ
- ส่วนประกอบบางตัวของคีย์ foreign เป็น null.

ข้อจำกัดแบบอ้างอิงจะถูกบังคับใช้เมื่อมีการรันคำสั่ง INSERT, UPDATE, และ DELETE.

- ข้อจำกัดการตรวจสอบ

ข้อจำกัดการตรวจสอบ คือ กฎที่จำกัดค่าที่ใช้ในคอลัมน์หรือกลุ่มคอลัมน์. ข้อจำกัดการตรวจสอบสามารถสร้างเพิ่มเติมได้โดยใช้คำสั่ง CREATE TABLE และ ALTER TABLE. ข้อจำกัดการตรวจสอบจะถูกบังคับใช้เมื่อมีการรันคำสั่ง INSERT และ UPDATE. หากต้องการปฏิบัติตามข้อจำกัด, แถวข้อมูลแต่ละแถวที่แทรกเข้าไปหรือถูกอัปเดตในตารางต้องทำให้เงื่อนไขที่ระบุไว้เป็นแบบ TRUE หรือ ไม่รู้จัก (เนื่องจากเป็น null).

สิ่งอ้างอิงที่เกี่ยวข้อง

“ข้อจำกัด” ในหน้า 139

DB2 UDB for iSeries สนับสนุนข้อจำกัดที่เป็นเอกลักษณ์, อ้างอิงได้, และเป็นตัวตรวจสอบข้อจำกัด.

ทริกเกอร์

ทริกเกอร์คือชุดของ action ที่รันโดยอัตโนมัติ เมื่อใดก็ตามที่มีเหตุการณ์ที่ระบุไว้เกิดขึ้นกับตารางพื้นฐาน หรือมุมมองที่ระบุไว้.

เหตุการณ์ดังกล่าวอาจเป็น การแทรก, การอัปเดต, การลบ, หรือการอ่าน. ทริกเกอร์สามารถรันก่อน หรือหลังเหตุการณ์เหล่านี้. DB2 UDB for iSeries จะสนับสนุนการแทรก SQL, อัปเดต, และทริกเกอร์ที่ใช้ลบ และทริกเกอร์ภายนอก.

ข้อมูลที่เกี่ยวข้อง

การทำทริกเกอร์เหตุการณ์แบบอัตโนมัติในฐานข้อมูลของคุณ

โพรซีเจอร์ที่เก็บไว้

โพรซีเจอร์ที่เก็บไว้คือ โปรแกรมที่อาจถูกเรียกโดยใช้คำสั่ง SQL CALL.

DB2 UDB for iSeries สนับสนุนโพรซีเจอร์ที่เก็บไว้แบบภายนอกและ SQL โพรซีเจอร์. โพรซีเจอร์ที่เก็บไว้แบบภายนอกอาจเป็นโปรแกรมระบบ, เซอร์วิสโปรแกรม, หรือโพรซีเจอร์ REXX ใดๆก็ได้. แต่ไม่สามารถเป็น โปรแกรม หรือ โพรซีเจอร์ System/36™ ได้. SQL โพรซีเจอร์จะถูกกำหนดไว้ทั้งหมดใน SQL และอาจมีคำสั่ง SQL รวมทั้ง SQL control statements.

หลักการที่เกี่ยวข้อง

“สตอร์โพรซีเจอร์” ในหน้า 143

โพรซีเจอร์ (ซึ่งมักเรียกว่า สตอร์โพรซีเจอร์) คือโปรแกรมที่สามารถเรียกขึ้นมาเพื่อปฏิบัติงาน ซึ่งมีทั้งข้อความภาษาโฮสต์ และคำสั่ง SQL. โพรซีเจอร์ใน SQL มีข้อดีเหมือนกับโพรซีเจอร์ในภาษาโฮสต์.

ลำดับ (Sequences)

sequence คืออ็อบเจกต์พื้นที่ข้อมูล ที่กำหนดให้วิธีการที่ง่ายและรวดเร็วในการสร้างหมายเลขเฉพาะ.

คุณสามารถใช้ลำดับในการแทนที่ คอลัมน์ IDENTITY หรือ คอลัมน์ตัวเลขที่ผู้ใช้สร้างขึ้นมาได้. ลำดับมีลักษณะการใช้งานคล้ายกับทางเลือกเหล่านี้.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การสร้าง และการใช้ลำดับ” ในหน้า 26

ลำดับ คืออ็อบเจกต์ชนิดหนึ่งที่อนุญาตให้คุณสร้างค่าได้อย่างรวดเร็วและง่าย.

ฟังก์ชันแบบผู้ใช้กำหนด (User-defined functions)

ฟังก์ชันแบบผู้ใช้กำหนดเอง คือโปรแกรมที่อาจถูกเรียกทำงานได้เหมือนกับฟังก์ชันในตัวอื่นๆ.

DB2 UDB for iSeries สนับสนุนฟังก์ชันแบบภายนอก, ฟังก์ชัน SQL, และฟังก์ชันแบบต้นทาง (sourced functions). ฟังก์ชันแบบภายนอกอาจเป็นโปรแกรม ILE ของระบบใดๆ หรือเซอร์วิสโปรแกรม. ฟังก์ชัน SQL ถูกกำหนดไว้ทั้งหมดใน SQL และอาจมีคำสั่ง SQL รวมทั้งคำสั่ง SQL control. ฟังก์ชันแบบต้นทางจะถูกสร้างขึ้นมาบนฟังก์ชันในตัวใด (built-in) หรือ บนฟังก์ชันแบบผู้ใช้กำหนดที่มีอยู่. คุณสามารถสร้างฟังก์ชัน scala หรือฟังก์ชันตารางให้เป็น SQL หรือฟังก์ชันแบบภายนอก.

หลักการที่เกี่ยวข้อง

“การใช้ user-defined functions (UDFs)” ในหน้า 180

ในการเขียนแอ็พพลิเคชัน SQL, คุณสามารถเลือกปฏิบัติหรือดำเนินการบางอย่างได้ในแบบ UDF หรือ แบบรูทีนย่อยในแอ็พพลิเคชันของคุณ. ถึงแม้ว่ามันอาจจะดูง่ายกว่าใน การเลือกการดำเนินการใหม่แบบรูทีนย่อยในแอ็พพลิเคชันของคุณ, คุณอาจต้องพิจารณา ถึงประโยชน์ของการใช้งาน UDF แทน.

ประเภทที่ผู้ใช้กำหนด (User-defined types)

ประเภทที่ผู้ใช้กำหนด คือ ประเภทข้อมูลเฉพาะที่ผู้ใช้สามารถกำหนดได้โดยไม่ขึ้นกับประเภทข้อมูลที่มีอยู่ในระบบจัดการฐานข้อมูล.

ชนิดข้อมูลเฉพาะแม้พบกับประเภทฐานข้อมูลที่มีอยู่แบบหนึ่งต่อหนึ่ง.

หลักการที่เกี่ยวข้อง

“การใช้ user-defined distinct types (UDT)” ในหน้า 241

user-defined distinct type คือกลไกที่ทำให้คุณขยายความสามารถของDB2 ให้มีชนิดข้อมูลมากไปกว่าที่มีอยู่.

SQL แพ็กเกจ

SQL แพ็กเกจคืออ็อบเจกต์ที่มีโครงสร้างควบคุมซึ่งเกิดขึ้นเมื่อมีการเชื่อมโยงคำสั่ง SQL ในแอปพลิเคชันโปรแกรมเข้ากับระบบจัดการฐานข้อมูลเชิงสัมพันธ์แบบรีโมต (DBMS).

DBMS จะใช้โครงสร้างควบคุมเพื่อประมวลผลคำสั่ง SQL ที่พบขณะรันแอปพลิเคชันโปรแกรม.

SQL แพ็กเกจจะถูกสร้างขึ้นเมื่อระบุชื่อฐานข้อมูลเชิงสัมพันธ์ (พารามิเตอร์ RDB) บนคำสั่ง Create SQL (CRTSQLxxx) และเมื่อสร้างอ็อบเจกต์โปรแกรม. แพ็กเกจสามารถสร้างขึ้นได้โดยใช้คำสั่ง CRTSQLPKG.

SQL แพ็กเกจสามารถสร้างขึ้นได้โดยใช้คำสั่ง QSQRCE API. การอ้างถึง SQL แพ็กเกจภายในหัวข้อนี้จะหมายถึงแพ็กเกจ Distributed Program SQL เท่านั้น. QSQRCE ใช้ SQL แพ็กเกจเพื่อให้การสนับสนุน Extended Dynamic SQL.

หมายเหตุ: xxx ในคำสั่งนี้หมายถึงตัวบ่งชี้ภาษาโฮสต์ซึ่งได้แก่: CI สำหรับภาษา ILE C, CPPI สำหรับ ILE C++ สำหรับภาษา iSeries, CBL สำหรับ COBOL สำหรับภาษา iSeries, CBLI สำหรับภาษา ILE COBOL, PLI สำหรับภาษา iSeries PL/I, RPG สำหรับภาษา RPG/400, และ RPGI สำหรับภาษา ILE RPG.

สิ่งอ้างอิงที่เกี่ยวข้อง

“ฟังก์ชันของฐานข้อมูลเชิงสัมพันธ์แบบกระจายและ SQL” ในหน้า 309

ฐานข้อมูลเชิงสัมพันธ์แบบกระจายประกอบไปด้วยชุด SQL อ็อบเจกต์ที่กระจายอยู่บนระบบคอมพิวเตอร์ที่เชื่อมต่อกันและกัน.

ข้อมูลที่เกี่ยวข้อง

QSQRCE

อ็อบเจกต์แอปพลิเคชันโปรแกรม

กระบวนการสร้างแอปพลิเคชันโปรแกรม DB2 UDB for iSeries อาจทำให้เกิดการสร้างอ็อบเจกต์หลายตัว. หัวข้อนี้จะอธิบายคร่าวๆ ถึงกระบวนการสร้างแอปพลิเคชัน DB2 UDB for iSeries.

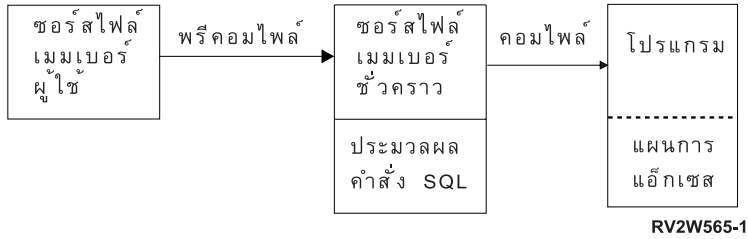
DB2 UDB for iSeries สนับสนุนพีริคอมไพเลอร์แบบไม่มี ILE และแบบ ILE. แอปพลิเคชันโปรแกรมอาจเป็นแบบกระจายหรือไม่กระจายก็ได้.

ด้วย DB2 UDB for iSeries คุณอาจต้องจัดการอ็อบเจกต์ต่อไปนี้:

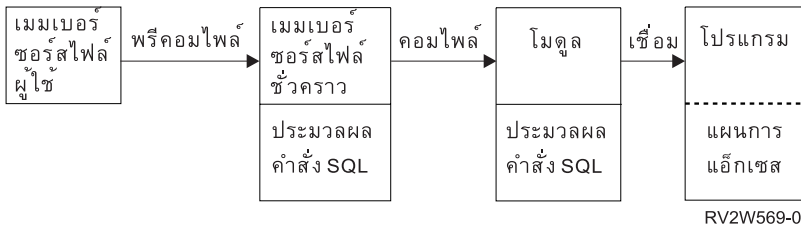
- ซอร์สต้นฉบับ
- หรือ, อ็อบเจกต์โมดูลสำหรับโปรแกรม ILE

- โปรแกรมหรือเซอวิวิโปรแกรม
- SQL แพ็กเกจสำหรับโปรแกรมแบบกระจาย

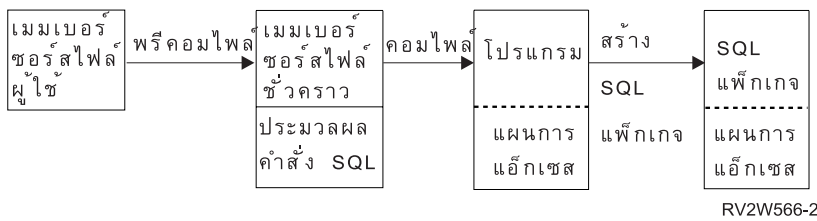
ด้วยโปรแกรม DB2 UDB for iSeries แบบไม่มี ILE และแบบไม่กระจาย, คุณต้องจัดการเฉพาะซอร์สต้นฉบับ และโปรแกรมผลลัพธ์. ข้อมูลต่อไปนี้จะแสดงอ็อบเจ็กต์ที่เกี่ยวข้องและขั้นตอนที่เกิดขึ้นระหว่างกระบวนการพรีคอมไพล์และคอมไพล์สำหรับโปรแกรม DB2 UDB for iSeries ที่ไม่มี ILE แบบไม่กระจาย:



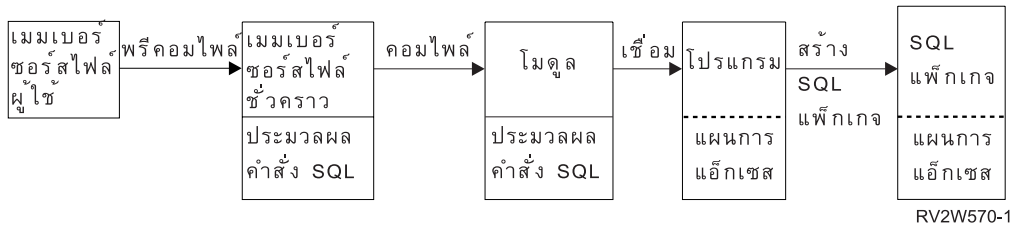
ด้วยโปรแกรม DB2 UDB for iSeries ที่มี ILE แบบไม่กระจาย, คุณอาจต้องจัดการซอร์สต้นฉบับ, โมดูล, และโปรแกรมผลลัพธ์ หรือเซอวิวิโปรแกรม. ข้อมูลต่อไปนี้จะแสดงอ็อบเจ็กต์ที่เกี่ยวข้องและขั้นตอนที่เกิดขึ้นระหว่างกระบวนการพรีคอมไพล์และคอมไพล์สำหรับโปรแกรม DB2 UDB for iSeries ที่มี ILE แบบไม่กระจาย เมื่อระบุ OBJTYPE(*PGM) ในคำสั่งพรีคอมไพล์:



ด้วยโปรแกรม DB2 UDB for iSeries ที่ไม่มี ILE แบบกระจาย, คุณต้องจัดการซอร์สต้นฉบับ, โปรแกรมผลลัพธ์, และแพ็กเกจผลลัพธ์. ข้อมูลต่อไปนี้จะแสดงอ็อบเจ็กต์ที่เกี่ยวข้องและขั้นตอนที่เกิดขึ้นระหว่างกระบวนการพรีคอมไพล์และคอมไพล์สำหรับโปรแกรม DB2 UDB for iSeries ที่ไม่มี ILE แบบกระจาย:



ด้วยโปรแกรม DB2 UDB for iSeries ที่มี ILE แบบกระจาย, คุณต้องจัดการซอร์สต้นฉบับ, อ็อบเจ็กต์โมดูล, โปรแกรมผลลัพธ์ หรือเซอวิวิโปรแกรม, และแพ็กเกจผลลัพธ์. SQL แพ็กเกจสามารถสร้างขึ้นสำหรับโมดูลแบบกระจายแต่ละโมดูลในโปรแกรมที่มี ILE แบบกระจายหรือเซอวิวิโปรแกรม. ข้อมูลต่อไปนี้จะแสดงอ็อบเจ็กต์ที่เกี่ยวข้องและขั้นตอนที่เกิดขึ้นระหว่างกระบวนการพรีคอมไพล์สำหรับโปรแกรม DB2 UDB for iSeries ที่มี ILE แบบกระจาย:



หมายเหตุ: แผนการเข้าใช้งานที่เชื่อมโยงกับอ็อบเจกต์โปรแกรม DB2 UDB for iSeries แบบกระจายจะไม่ถูกสร้างขึ้นจนกว่าจะรันโปรแกรมในระบบ.

ข้อมูลที่เกี่ยวข้อง

การเตรียมและรันโปรแกรมด้วยคำสั่ง SQL

รายการไฟล์ต้นฉบับย่อยของผู้ใช้ (User source file member)

รายการไฟล์ต้นฉบับย่อยจะมีภาษาแอสมบลีเคชันของโปรแกรมเมอร์ และคำสั่ง SQL. คุณสามารถสร้างและปรับปรุงรายการไฟล์ต้นฉบับย่อยโดยใช้ source entry utility (SEU), ซึ่งเป็นส่วนหนึ่งของไลเซนส์โปรแกรม IBM WebSphere® Development Studio for iSeries.

เอาต์พุตรายการไฟล์ต้นฉบับย่อย (Output source file member)

SQL 프리คอมไพล์จะสร้างเอาต์พุตรายการไฟล์ต้นฉบับย่อย.

ตามค่าดีฟอลต์, กระบวนการ 프리คอมไพล์จะสร้างไฟล์ต้นฉบับชั่วคราว QSQLTxxxxx ใน QTEMP, หรือคุณสามารถระบุเอาต์พุตไฟล์ต้นฉบับให้เป็นชื่อไฟล์ถาวรได้บนคำสั่ง 프리คอมไพล์. หากกระบวนการ 프리คอมไพล์ใช้งานไลบรารี QTEMP, ระบบจะลบไฟล์ออกอัตโนมัติเมื่องานเสร็จสมบูรณ์. รายการย่อยที่มีชื่อเดียวกันกับชื่อโปรแกรมจะถูกเพิ่มเข้าไปยัง เอาต์พุตไฟล์ต้นฉบับ. รายการย่อยนี้จะมีไอเท็มต่อไปนี้:

- การเรียกไปยังตัวสนับสนุนรันไทม์ SQL, ซึ่งได้แทนที่คำสั่ง SQL ที่ฝังอยู่
- คำสั่ง SQL ที่ได้รับการวิเคราะห์ค่าและตรวจสอบซินแทกซ์

ตามค่าดีฟอลต์, 프리คอมไพลเลอร์จะเรียกคอมไพลเลอร์ภาษาโฮสต์.

ข้อมูลที่เกี่ยวข้อง

การเตรียมและรันโปรแกรมด้วยคำสั่ง SQL

โปรแกรม

โปรแกรมคืออ็อบเจกต์ที่คุณสามารถรัน และถูกสร้างขึ้นจากกระบวนการคอมไพล์สำหรับการคอมไพล์แบบ non-ILE หรือกระบวนการเชื่อมโยงสำหรับคอมไพล์แบบ ILE.

แผนการเข้าใช้งานคือชุดโครงสร้างภายในและข้อมูลซึ่งบอก SQL ว่าจะรันคำสั่ง SQL แบบฝังอยู่ให้เกิดประสิทธิภาพสูงสุดได้อย่างไร. แผนการจะถูกสร้างขึ้นเมื่อสร้างโปรแกรมสำเร็จแล้ว. แผนการเข้าใช้งานจะไม่ถูกสร้างขึ้นระหว่างการสร้างโปรแกรมสำหรับคำสั่ง SQL หากคำสั่งมีลักษณะดังนี้:

- อ้างถึงตารางหรือมุมมองที่หาไม่พบ
- อ้างถึงตารางหรือมุมมองที่คุณไม่มีสิทธิ์ใช้งาน

แผนการเข้าใช้งานของคำสั่งดังกล่าวนั้นจะถูกสร้างขึ้นเมื่อรันโปรแกรม. หาก, ในเวลานั้น, ยังไม่พบตารางหรือมุมมองหรือคุณยังไม่มิลิทธิ์เข้าใช้งาน, SQLCODE ค่าลบจะถูกส่งคืนมา. แผนการเข้าใช้งานจะถูกเก็บไว้และรักษาไว้ในอ็อบเจกต์โปรแกรมสำหรับโปรแกรม SQL แบบไม่กระจาย และใน SQL แพ็กเกจสำหรับโปรแกรม SQL แบบกระจาย.

SQL แพ็กเกจ

SQL แพ็กเกจจะมีแผนการเข้าใช้งานสำหรับโปรแกรม SQL แบบกระจาย.

SQL แพ็กเกจคือ อ็อบเจกต์ที่ถูกสร้างขึ้นเมื่อ:

- โปรแกรม SQL แบบกระจายจะถูกสร้างขึ้นอย่างสมบูรณ์โดยใช้พารามิเตอร์ RDB บนคำสั่ง CRTSQLxxx .
- เมื่อรันคำสั่ง Create SQL Package (CRTSQLPKG) แล้ว.

เมื่อโปรแกรม SQL แบบกระจายถูกสร้างขึ้น, ชื่อของ SQL แพ็กเกจและโทเค็นตรวจสอบความสอดคล้องกันภายในจะถูกบันทึกไว้ในโปรแกรม. ชื่อแพ็กเกจและโทเค็นนี้จะถูกใช้งานในช่วงรันไทม์เพื่อค้นหา SQL แพ็กเกจและเพื่อตรวจสอบว่า SQL แพ็กเกจถูกต้องสำหรับโปรแกรมนี้อหรือไม่. เนื่องจากชื่อของ SQL แพ็กเกจสำคัญมากสำหรับการรันโปรแกรม SQL แบบกระจาย, SQL แพ็กเกจจะต้องไม่มีการ:

- ย้าย
- เปลี่ยนชื่อ
- ทำซ้ำ
- เรียกคืนไปไว้ยังไลบรารีอื่น

โมดูล

โมดูล คืออ็อบเจกต์ชนิด Integrated Language Environment® (ILE) ที่ถูกสร้างโดยการคอมไพล์ซอร์สโค้ดโดยใช้คำสั่ง CRTxxxMOD (หรือคำสั่งใดๆ ของ CRTBNDxxx โดยที่ xxx คือ C, CBL, CPP, หรือ RPG).

คุณสามารถรันโมดูลได้ก็ต่อเมื่อคุณใช้คำสั่ง Create Program (CRTPGM) เพื่อเชื่อมโยงโมดูลนั้นเข้ากับโปรแกรม. ตามปกติคุณสามารถเชื่อมโยงโมดูลหลายตัวเข้าด้วยกันได้, แต่คุณอาจเชื่อมโยงโมดูลเข้ากับตัวมันเองก็ได้ด้วยเช่นกัน. โมดูลจะมีข้อมูลเกี่ยวกับคำสั่ง SQL อย่างไรก็ตาม แผนการเข้าใช้งาน SQL จะไม่ถูกสร้างขึ้นจนกว่าจะเชื่อมโยงโมดูลเข้ากับโปรแกรม หรือเซอวิสโปรแกรม.

ข้อมูลที่เกี่ยวข้อง

คำสั่งการสร้างโปรแกรม (CRTPGM)

เซอวิสโปรแกรม

เซอวิสโปรแกรม คืออ็อบเจกต์ชนิด Integrated Language Environment (ILE) ซึ่งจัดเตรียมค่ากลุ่มของแพ็กเกจที่สนับสนุนคอลเลอรัทที่ภายนอก (ฟังก์ชัน หรือโพรซีเจอร์) ลงในอ็อบเจกต์แต่ละตัว.

โปรแกรมที่เกี่ยวข้องและเซอวิสโปรแกรมอื่นๆ สามารถเข้าใช้งานรูทีนเหล่านี้ได้โดยการ resolve รายการอิมพอร์ตของรูทีนไปยังรายการเอ็กชพอร์ตของเซอวิสโปรแกรม. การเชื่อมต่อเข้ากับเซอวิสเหล่านี้จะเกิดขึ้นเมื่อมีการสร้างโปรแกรมเรียกทำงาน. วิธีการนี้จะปรับปรุงประสิทธิภาพในการเรียกใช้งานรูทีนเหล่านี้โดยไม่ต้องใส่โค้ดไว้ในโปรแกรมเรียกทำงาน.

Data definition language (DDL)

Data definition language (DDL) คือส่วนของ SQL ที่อนุญาตให้คุณสร้าง, เปลี่ยน, และทำลายอ็อบเจกต์ฐานข้อมูล. อ็อบเจกต์ฐานข้อมูลเหล่านี้ประกอบด้วยแบบแผน, ตาราง, มุมมอง, ลำดับ, แคตาล็อก, ดรรชนี, และ alias.

หลักการที่เกี่ยวข้อง

“ประเภทของคำสั่ง SQL” ในหน้า 6

มีประเภทของคำสั่ง SQL พื้นฐานอยู่หลายประเภท. ซึ่งคำสั่งเหล่านั้นจะถูกแสดงไว้ที่นี้ตามฟังก์ชัน.

ข้อมูลที่เกี่ยวข้อง

เริ่มต้นด้วย SQL

สร้างแบบแผน (schema)

แบบแผนจะมีการจัดกลุ่มของ SQL อ็อบเจกต์แบบโลจิคัล.

แบบแผนประกอบด้วยไลบรารี, เจอร์นัล, journal receiver, แคตาล็อก, และอาจรวมพจนานุกรมข้อมูล. การสร้าง, ย้าย, หรือเก็บตาราง, มุมมอง และอ็อบเจกต์ระบบ (เช่นโปรแกรม) อาจไว้ในไลบรารี, ระบบใดๆ ก็ได้. ไฟล์ระบบทั้งหมดอาจถูกสร้างหรือย้ายไปยังแบบแผน SQL หากแบบแผน SQL ไม่มีพจนานุกรมข้อมูล. หากแบบแผน SQL มีพจนานุกรมข้อมูลแล้ว:

- source physical ไฟล์หรือ nonsource physical ไฟล์ที่มีต้นทางที่มีรายการย่อยเดียวสามารถถูกสร้าง, ย้าย, หรือเก็บไว้ในแบบแผน SQL ได้.
- เราไม่สามารถวางโลจิคัลไฟล์ในแบบแผน SQL ได้เนื่องจากไฟล์เหล่านั้นไม่สามารถอธิบายอยู่ใน พจนานุกรมข้อมูลได้.

คุณสามารถสร้างและเป็นเจ้าของแบบแผนได้หลายอัน.

แบบแผนถูกสร้างขึ้นโดยใช้ข้อความ CREATE SCHEMA. ตัวอย่างเช่น:

สร้างแบบแผนที่ชื่อ DBTEMP.

```
CREATE SCHEMA DBTEMP
```

ข้อมูลที่เกี่ยวข้อง

CREATE SCHEMA statement

สร้างตาราง

ตารางสามารถ visualize จัดเรียงข้อมูลเป็นสองมิติ ซึ่งประกอบด้วยแถวและคอลัมน์.

แถวคือส่วนที่เป็นแนวนอนซึ่งประกอบด้วยคอลัมน์ตั้งแต่หนึ่งคอลัมน์ขึ้นไป. คอลัมน์คือส่วนที่เป็นแนวตั้งซึ่งประกอบด้วยแถวตั้งแต่หนึ่งแถวขึ้นไปในประเภทข้อมูลหนึ่งประเภท. ข้อมูลทั้งหมดในหนึ่งคอลัมน์ต้องเป็นข้อมูลประเภทเดียวกัน. ตารางใน SQL คือฟิลิซัลไฟล์ที่มีคีย์หรือไม่มีคีย์ก็ได้.

ตารางถูกสร้างขึ้นโดยใช้ข้อความ CREATE TABLE. definition ต้องมีชื่อ definition และชื่อและแอตทริบิวต์ของคอลัมน์ด้วย. definition อาจรวมถึงแอตทริบิวต์อื่นๆ ของตารางเช่นคีย์หลัก.

ตัวอย่าง: สมมติว่าคุณมีสิทธิระดับผู้ดูแลระบบให้สร้างตารางชื่อ 'INVENTORY' โดยมีคอลัมน์ต่อไปนี้:

- หมายเลขชิ้นส่วน: จำนวนเต็มระหว่าง 1 ถึง 9999, และต้องไม่เป็นค่า null

- รายละเอียด: อักขระต้องมีความยาวตั้งแต่ 0 ถึง 24
- จำนวนที่มีอยู่: จำนวนเต็มตั้งแต่ 0 ถึง 100000

คีย์หลักคือ PARTNO.

```
CREATE TABLE INVENTORY
(PARTNO          SMALLINT      NOT NULL,
DESCR           VARCHAR(24 ),
QONHAND         INT,
PRIMARY KEY(PARTNO))
```

ข้อมูลที่เกี่ยวข้อง

ชนิดข้อมูล

การเพิ่ม และการลบข้อจำกัดในตาราง

สามารถเพิ่มข้อจำกัดเข้าในตาราง หรือตารางที่มีอยู่เดิมได้. คุณสามารถเพิ่มคีย์หลักแบบเฉพาะ, ข้อจำกัดแบบอ้างอิง, หรือข้อจำกัดเพื่อการตรวจสอบ, โดยใช้ *ADD constraint* clause บนข้อความ CREATE TABLE หรือข้อความ ALTER TABLE.

ตัวอย่างเช่น, เพิ่มคีย์หลักลงยังตารางใหม่หรือลงยังตารางที่มีอยู่เดิม. ตัวอย่างต่อไปนี้อธิบายการเพิ่มคีย์หลักลงในตารางที่มีอยู่เดิม โดยใช้คำสั่ง ALTER TABLE.

```
ALTER TABLE CORPDATA.DEPARTMENT
ADD PRIMARY KEY (DEPTNO)
```

หากต้องการให้คีย์นี้เป็นคีย์แบบเฉพาะ, ให้ใส่แทนที่คีย์เวิร์ด PRIMARY ด้วย UNIQUE.

คุณสามารถลบข้อจำกัดออกโดยใช้คำสั่ง ALTER TABLE แบบเดียวกัน:

```
ALTER TABLE CORPDATA.DEPARTMENT
DROP PRIMARY KEY (DEPTNO)
```

Referential integrity และตาราง

Referential integrity คือเงื่อนไขสำหรับชุดของตารางในฐานข้อมูล ซึ่งใช้ในการอ้างอิงทั้งหมดจากตารางหนึ่งไปยังอีกตารางหนึ่ง.

พิจารณาตัวอย่างนี้:

- CORPDATA.EMPLOYEE ใช้เป็นรายการหลักของพนักงานทั้งหมด.
- CORPDATA.DEPARTMENT ทำหน้าที่เป็นเสมือนรายการหลักของจำนวนแผนกทั้งหมดที่ถูกต้อง.
- CORPDATA.EMP_ACT ให้รายการหลักของกิจกรรมที่ดำเนินการในโครงการต่างๆ.

ตารางอื่นๆ จะอ้างอิงถึง entity เดียวกับที่อธิบายไว้ในตารางเหล่านี้. เมื่อตารางประกอบด้วยข้อมูลที่มีรายการหลัก, ข้อมูลดังกล่าวควรปรากฏในรายการหลักนั้น, ไม่เช่นนั้นแสดงว่าการอ้างอิงไม่ถูกต้อง. ตารางซึ่งประกอบด้วยรายการหลักคือ ตาราง *parent*, และตารางที่อ้างอิงถึงตาราง parent คือ ตาราง *dependent*. เมื่อการอ้างอิงจาก ตาราง dependent ไปยังตาราง parent ถูกต้อง, สภาวะของชุดตารางจะถูกเรียกว่า *ความสัมพันธ์ในการอ้างอิง*.

หรือจะกล่าวอีกนัยหนึ่ง, ความสัมพันธ์ในการอ้างอิงคือสภาวะของฐานข้อมูลโดยที่ค่าของ foreign key ทั้งหมดถูกต้อง. ค่าแต่ละค่าของ foreign key จะต้องอยู่ใน parent key เช่นกันหรือมีค่าเป็น null. Definition ของความสัมพันธ์ในการอ้างอิงนี้จะต้องมีความเข้าใจในคำศัพท์ต่อไปนี้:

- *unique key* คือคอลัมน์หรือชุดคอลัมน์ในตารางซึ่งระบุแถวเป็นการเฉพาะ. แม้ว่าในหนึ่งตารางสามารถมี *unique key* ได้เป็นจำนวนมาก, แต่แถวสองแถวในหนึ่งตารางจะต้องไม่มีค่า *unique key* ค่าเดียวกัน.
- *primary key* คือ *unique key* ซึ่งต้องไม่มีค่าเป็น null. ในหนึ่งตารางจะต้องมี *primary key* เดียวเท่านั้น.
- *parent key* คือ *unique key* หรือ *primary key* ซึ่งถูกอ้างอิงในข้อจำกัดการอ้างอิง.
- *foreign key* คือ คอลัมน์หรือชุดคอลัมน์ซึ่งค่าต้องตรงกับค่าของ *parent key*. หากค่าคอลัมน์ใดๆ ที่ใช้ในการสร้าง *foreign key* เท่ากับ null, กฎดังกล่าวก็จะใช้ไม่ได้.
- *parent table* คือ ตารางซึ่งประกอบด้วย *parent key*.
- *dependent table* คือ ตารางซึ่งประกอบด้วย *foreign key*.
- *descendent table* คือ ตาราง *dependent* หรือตารางที่อยู่ในลำดับถัดจากตาราง *dependent*.

การบังคับใช้ความสมบูรณ์ในการอ้างอิงเป็นการละเมิดกฎที่ระบุว่า ทุกๆ *foreign key* ที่ไม่ใช่ null ต้องมี *parent key* ที่ตรงกัน.

SQL สนับสนุนแนวคิดเรื่องความสมบูรณ์ในการอ้างอิงด้วยคำสั่ง CREATE TABLE และ ALTER TABLE.

สิ่งอ้างอิงที่เกี่ยวข้อง

“DB2 UDB for iSeries ตารางตัวอย่าง” ในหน้า 335

หัวข้อนี้ประกอบด้วยตารางตัวอย่างที่ใช้อ้างอิง และใช้ในหัวข้อนี้ รวมทั้งหัวข้อ SQL Reference.

ข้อมูลที่เกี่ยวข้อง

CREATE TABLE

ALTER TABLE

การเพิ่ม หรือการลบข้อจำกัดในการอ้างอิง:

ข้อจำกัด คือ กฎที่รับรองว่าการอ้างอิงจากตารางหนึ่ง, หรือตาราง *dependent*, ไปยังข้อมูลในอีกตารางหนึ่ง, หรือตาราง *parent*, ถูกต้อง. คุณใช้ข้อจำกัดในการอ้างอิงเพื่อตรวจสอบความสมบูรณ์ในการอ้างอิง.

ใช้คำสั่ง SQL CREATE TABLE และ ALTER TABLE เพื่อเพิ่ม หรือเปลี่ยนข้อจำกัดในการอ้างอิง.

ด้วยข้อจำกัดในการอ้างอิง, ค่าที่ไม่ใช่ null ของ *foreign key* จะใช้ได้ก็ต่อเมื่อค่าเหล่านั้น ปรากฏขึ้นเป็นเสมือนค่าของ *parent key*. เมื่อคุณกำหนดข้อจำกัดในการอ้างอิง, ให้คุณระบุ:

- *primary* หรือ *unique key*
- *foreign key*
- ลบและอัปเดตกฎที่ระบุการดำเนินงานที่เกิดขึ้นซึ่งเกี่ยวข้องกับแถว *dependent* เมื่อแถว *parent* ถูกลบออก หรือมีการอัปเดต.

หรือ, คุณสามารถระบุชื่อให้กับข้อจำกัดได้. หากไม่มีการระบุชื่อ, ระบบจะสร้างชื่อให้โดยอัตโนมัติ.

หลังจากมีการกำหนดข้อจำกัดในการอ้างอิงแล้ว, ระบบจะบังคับให้ใช้ข้อจำกัดดังกล่าวกับทุกๆ การปฏิบัติคำสั่ง INSERT, DELETE, และ UPDATE โดยกระทำผ่าน SQL หรืออินเทอร์เฟซอื่นๆ ซึ่งรวมถึง iSeries Navigator, คำสั่ง CL, ยูทิลิตี้, หรือข้อความภาษาชั้นสูง.

ข้อมูลที่เกี่ยวข้อง

CREATE TABLE

ALTER TABLE

การเพิ่มข้อจำกัดในการอ้างอิง:

กฎที่ว่า ทุกๆ หมายเลขแผนกที่แสดงไว้ในตารางตัวอย่างพนักงานที่ปรากฏในตารางแผนกนั้น คือ ข้อจำกัดในการอ้างอิง.

ข้อจำกัดนี้เป็นการรับรองว่าพนักงานทุกคนอยู่ในแผนกที่มีอยู่. คำสั่ง SQL ต่อไปนี้เป็นการสร้างตาราง CORPDATA.DEPARTMENT และตาราง CORPDATA.EMPLOYEE ซึ่งมีการกำหนดความสัมพันธ์ของข้อจำกัดเหล่านั้น.

```
CREATE TABLE CORPDATA.DEPARTMENT
  (DEPTNO    CHAR(3)    NOT NULL PRIMARY KEY,
   DEPTNAME  VARCHAR(29) NOT NULL,
   MGRNO     CHAR(6),
   ADMRDEPT  CHAR(3)    NOT NULL
   CONSTRAINT REPORTS_TO_EXISTS
   REFERENCES CORPDATA.DEPARTMENT (DEPTNO)
   ON DELETE CASCADE)
```

```
CREATE TABLE CORPDATA.EMPLOYEE
  (EMPNO     CHAR(6)    NOT NULL PRIMARY KEY,
   FIRSTNME  VARCHAR(12) NOT NULL,
   MIDINIT   CHAR(1)    NOT NULL,
   LASTNAME  VARCHAR(15) NOT NULL,
   WORKDEPT  CHAR(3)    CONSTRAINT WORKDEPT_EXISTS
   REFERENCES CORPDATA.DEPARTMENT (DEPTNO)
   ON DELETE SET NULL ON UPDATE RESTRICT,

   PHONENO   CHAR(4),
   HIREDATE  DATE,
   JOB       CHAR(8),
   EDLEVEL   SMALLINT  NOT NULL,
   SEX       CHAR(1),
   BIRTHDATE DATE,
   SALARY    DECIMAL(9,2),
   BONUS     DECIMAL(9,2),
   COMM      DECIMAL(9,2),
   CONSTRAINT UNIQUE_LNAME_IN_DEPT UNIQUE (WORKDEPT, LASTNAME))
```

ในกรณีนี้, ตาราง DEPARTMENT มีคอลัมน์ของจำนวนแผนกเฉพาะ (DEPTNO) ซึ่งฟังก์ชันเป็น primary key, และเป็นตาราง parent ในความสัมพันธ์สองข้อของข้อจำกัด:

REPORTS_TO_EXISTS

คือ ข้อจำกัดในการอ้างอิงด้วยตนเองโดยที่ตาราง DEPARTMENT เป็นทั้ง parent และ dependent ในความสัมพันธ์เดียวกัน. ทุกๆ ค่าของ ADMRDEPT ที่ไม่เป็นค่า null ต้องตรงกับค่าของ DEPTNO. แผนกต้องรายงานไปยังแผนกที่มีอยู่ในฐานข้อมูล. กฎ DELETE CASCADE แสดงว่าหากแถวที่มีค่า DEPTNO n ถูกลบออก, ทุกๆ แถวในตารางที่ ADMRDEPT เท่ากับ n ต้องถูกลบออกเช่นกัน.

WORKDEPT_EXISTS

สร้างตาราง EMPLOYEE เป็นตาราง dependent, และคอลัมน์การกำหนดแผนกพนักงาน (WORKDEPT) เป็น foreign key. ดังนั้น, ทุกๆ ค่าของ WORKDEPT ต้องตรงกับค่าของ DEPTNO. The กฎ DELETE SET NULL กล่าวว่า หากแถวถูกลบออกจาก DEPARTMENT โดยที่ค่าของ DEPTNO เท่ากับ n , ค่าของ WORKDEPT ใน

EMPLOYEE จะถูกตั้งค่าให้เป็น null ในทุกๆ แถวที่มีค่าเป็น n. กฎ UPDATE RESTRICT กล่าวว่า ค่าของ DEPTNO ใน DEPARTMENT ไม่สามารถอัปเดตได้ หากมีค่าของ WORKDEPT ใน EMPLOYEE ที่ตรงกับค่าปัจจุบันของ DEPTNO.

ข้อจำกัด UNIQUE_LNAME_IN_DEPT ในตาราง EMPLOYEE เป็นสาเหตุทำให้ LASTNAME ที่อยู่ภายใน department เป็น unique หรือต้องไม่ซ้ำกัน. ขณะที่ข้อจำกัดนี้ไม่เป็นเช่นนั้น, ข้อจำกัดนั้นจะอธิบายวิธีการที่ข้อจำกัดซึ่งสร้างคอลัมน์จำนวนมาก ถูกกำหนดที่ระดับตาราง.

ลบข้อจำกัดออก

ตัวอย่างนี้เป็นการลบคีย์หลักจากคอลัมน์ DEPTNO ในตาราง DEPARTMENT.

ข้อจำกัด REPORTS_TO_EXISTS, ที่กำหนดไว้ในตาราง DEPARTMENT, และข้อจำกัด WORKDEPT_EXISTS, ที่กำหนดไว้ในตาราง EMPLOYEE, จะถูกเอาออกเช่นกัน, เนื่องจาก การนำ primary key ออกก็คือการนำ parent key ในข้อจำกัดนั้นที่มีความสัมพันธ์กันออกนั่นเอง.

```
ALTER TABLE CORPDATA.EMPLOYEE DROP PRIMARY KEY
```

คุณยังสามารถลบข้อจำกัดตามชื่อ, ดังตัวอย่างนี้:

```
ALTER TABLE CORPDATA.DEPARTMENT  
DROP CONSTRAINT UNIQUE_LNAME_IN_DEPT
```

การระงับการตรวจสอบ

ข้อจำกัดในการอ้างอิงและข้อจำกัดในการตรวจสอบสามารถอยู่ในสถานะที่เรียกว่าการระงับการตรวจสอบ, โดยที่มีความเป็นไปได้ที่จะมีการละเมิดข้อจำกัดอยู่.

ในส่วนข้อจำกัดในการอ้างอิง, การละเมิดเกิดขึ้นได้เมื่อมีความไม่ตรงกันที่อาจเกิดขึ้นระหว่าง parent key และ foreign key. ในส่วนข้อจำกัดในการตรวจสอบ, การละเมิดเกิดขึ้นได้เมื่อค่าที่อาจเกิดขึ้นอยู่ในคอลัมน์ซึ่งถูกจำกัดโดยข้อจำกัดในการตรวจสอบ. เมื่อระบบตัดสินใจแล้วว่าอาจมีการละเมิดข้อจำกัด (อย่างเช่นหลังการดำเนินการกู้คืน), ข้อจำกัดนั้นจะถูกทำเครื่องหมายว่าเป็นระงับการตรวจสอบ. เมื่อเกิดกรณีเช่นนี้ขึ้น, จะมีการใช้ข้อบังคับในการใช้ตารางที่เกี่ยวข้องกับข้อจำกัดดังกล่าว. ในส่วนของข้อจำกัดในการอ้างอิง, มีการใช้ข้อบังคับต่อไปนี้:

- อนุญาตให้อินพุตหรือเอาต์พุตไฟล์ dependent.
- อนุญาตเฉพาะการอ่านและแทรกบนไฟล์ parent.

เมื่อข้อจำกัดในการตรวจสอบอยู่ในสถานะการระงับการตรวจสอบ, จะใช้ข้อบังคับต่อไปนี้:

- อนุญาตให้อ่านไฟล์.
- อนุญาตให้มีการแทรกและอัปเดตและบังคับใช้ข้อจำกัด.

เพื่อลบข้อจำกัดออกจากการระงับการตรวจสอบ, คุณต้อง:

1. ยกเลิกความสัมพันธ์ด้วยคำสั่ง CL Change Physical File Constraint (CHGPFCSST).
2. แก้ไขข้อมูลคีย์ (foreign, parent, หรือทั้งสอง) ของข้อจำกัดในการอ้างอิงหรือข้อมูลคอลัมน์สำหรับข้อจำกัดในการตรวจสอบ.
3. ใช้งานข้อจำกัดอีกครั้งด้วยคำสั่ง CL CHGPFCSST.

คุณสามารถระบุแถวที่ละเมิดข้อจำกัดด้วยคำสั่ง CL Display Check Pending Constraint (DSPCPCSST).

ข้อมูลที่เกี่ยวข้อง

ทำงานกับข้อจำกัดที่อยู่ในสถานะรอการตรวจสอบ
สถานะรอการตรวจสอบในข้อจำกัดในการอ้างอิง

การสร้างตารางโดยใช้ LIKE

คุณสามารถสร้างตารางที่เหมือนตารางอื่น. นั่นคือ, คุณสามารถสร้างตารางที่แทรก definition ของคอลัมน์ทั้งหมดจากตารางที่มีอยู่.

definition ที่ถูกคัดลอกคือ:

- ชื่อคอลัมน์ (และชื่อคอลัมน์ระบบ)
- ประเภทข้อมูล, ความแม่นยำ, ความยาว, และมาตราส่วน
- CCSID
- ข้อความของคอลัมน์ (LABEL ON)
- หัวคอลัมน์ (LABEL ON)

หาก LIKE clause อยู่ตามหลังชื่อตารางในทันทีและไม่ได้ปิดท้ายด้วยวงเล็บ, แอ็ททริบิวต์ต่อไปนี้จะถูกแทรกเข้าไป:

- ค่าดีฟอลต์
- ความเป็นศูนย์

ถ้าตารางหรือมุมมองที่กำหนดไว้ประกอบด้วย identity column, คุณต้อง ระบุ INCLUDING IDENTITY บนคำสั่ง CREATE TABLE ถ้าหากคุณต้องการให้มี identity column เกิดขึ้นในตารางใหม่. การทำงานซึ่งเป็นค่าดีฟอลต์ของ CREATE TABLE คือ EXCLUDING IDENTITY. หากตารางที่ระบุหรือมุมมองคือไฟล์แบบฟิลิคัลที่สร้างขึ้นแบบไม่มี SQL หรือไฟล์แบบลोजิคัล, แอ็ททริบิวต์แบบไม่มี SQL จะถูกลบออก.

สร้างตาราง EMPLOYEE2 ที่รวมคอลัมน์ทั้งหมดไว้ใน EMPLOYEE.

```
CREATE TABLE EMPLOYEE2 LIKE EMPLOYEE
```

ข้อมูลที่เกี่ยวข้อง

CREATE TABLE

การสร้างตารางโดยใช้ AS

คำสั่ง CREATE TABLE AS จะสร้างตารางจากผลลัพธ์ของคำสั่ง SELECT.

สามารถใช้งานนิพจน์ทั้งหมดซึ่งสามารถใช้ในคำสั่ง SELECT ในคำสั่ง CREATE TABLE AS ได้. คุณสามารถแทรกข้อมูลทั้งหมดจากตารางหรือตารางที่คุณเลือกจากได้.

ตัวอย่างเช่น, สร้างตารางที่ชื่อ EMPLOYEE3 ซึ่งรวมเอา definition คอลัมน์ทั้งหมดจาก EMPLOYEE ที่ซึ่ง DEPTNO = D11.

```
CREATE TABLE EMPLOYEE3 AS
  (SELECT PROJNO, PROJNAME, DEPTNO
   FROM EMPLOYEE
   WHERE DEPTNO = 'D11') WITH NO DATA
```

ถ้าตารางหรือมุมมองที่กำหนดไว้ประกอบด้วย identity column, คุณต้อง ระบุ INCLUDING IDENTITY บนคำสั่ง CREATE TABLE ถ้าหากคุณต้องการให้มี identity column เกิดขึ้นในตารางใหม่. การทำงานซึ่งเป็นค่าดีฟอลต์ของ CREATE TABLE คือ EXCLUDING IDENTITY. ประโยค WITH NO DATA ชี้ให้เห็นว่า definition ของคอลัมน์ได้ถูกก๊อปปี้ไปโดยไม่มีข้อมูล. ถ้าคุณต้องการใส่เพิ่มข้อมูลลงในตารางใหม่, EMPLOYEE3, ให้ใส่ประโยค WITH DATA. หากเคียเวิร์ที่ระบุไว้มีไฟล์คัลไฟล์ที่สร้างขึ้นแบบไม่มี SQL หรือโลจิคัลไฟล์, แอ็ททริบิวต์ที่เป็นผลลัพธ์ซึ่งไม่มี SQL จะถูกลบออก.

หลักการที่เกี่ยวข้อง

“การดึงข้อมูลโดยใช้คำสั่ง SELECT” ในหน้า 42

ศึกษาถึงวิธีต่างๆ ที่ทำให้เคียเวิร์ของคุณเก็บข้อมูลโดยใช้คำสั่ง SELECT. วิธีหนึ่งในการทำเช่นนี้คือการใช้คำสั่ง SELECT ในโปรแกรมเพื่อดึงค่าเฉพาะ (ตัวอย่างเช่น, แถวข้อมูลพนักงาน). นอกจากนี้, คุณสามารถใช้ clause เพื่อรวบรวมข้อมูลในแบบที่ต้องการได้.

ข้อมูลที่เกี่ยวข้อง

CREATE TABLE

การสร้างและเปลี่ยนตาราง materialized query

ตาราง materialized query เป็นตารางที่มี definition อยู่บนพื้นฐานของผลลัพธ์ของ query. ดังเช่น, ตาราง materialized query ปกติจะประกอบด้วยผลลัพธ์ที่คำนวณไว้ล่วงหน้า ซึ่งขึ้นอยู่กับข้อมูลที่มีอยู่ในตารางก่อนแล้ว หรือตารางที่มีพื้นฐานตาม definition.

optimizer จะมองหาตาราง materialized query และตัดสินใจว่าจะรันเคียเวิร์ให้มีประสิทธิภาพ เมื่อเทียบกับตาราง materialized query มากกว่าตารางพื้นฐาน หรือตารางทั้งหลาย. ถ้าจะให้มันทำงานเร็วกว่าเดิม, แล้ว query จะต้องรันเทียบกับ ตาราง materialized query. คุณสามารถทำเคียเวิร์ได้โดยตรงกับตาราง materialized query. สำหรับข้อมูลเพิ่มเติมเกี่ยวกับวิธีที่ optimizer ใช้ตาราง materialized query, โปรดดูหัวข้อ ประสิทธิภาพในการทำงานของฐานข้อมูลและการ optimization เคียเวิร์.

สมมติว่ามีตาราง transaction ขนาดใหญ่มากชื่อ TRANS ประกอบด้วย transaction ในแต่ละแถว ที่ประมวลผลต่อหนึ่งบริษัท. ตารางถูกกำหนดให้มีหลายคอลัมน์. ให้สร้างตาราง materialized query สำหรับตาราง TRANS ที่ประกอบด้วยข้อมูลสรุปรายวันสำหรับวันที่ และจำนวนของ transaction โดยใช้คำสั่งต่อไปนี้:

```
CREATE TABLE STRANS
AS (SELECT YEAR AS SYEAR, MONTH AS SMONTH, DAY AS SDAY, SUM(AMOUNT) AS SSUM
FROM TRANS
GROUP BY YEAR, MONTH, DAY )
DATA INITIALLY DEFERRED
REFRESH DEFERRED
MAINTAINED BY USER
```

ตาราง materialized query นี้ระบุว่าตารางนี้ไม่ได้มีอยู่ ณ เวลาที่มันถูกสร้างขึ้นมาโดย โดยการใช้ประโยค DATA INITIALLY DEFERRED. REFRESH DEFERRED ชี้ให้เห็นว่าการเปลี่ยนแปลงที่เกิดขึ้นกับ TRANS ไม่มีผลกระทบต่อใน STRANS. นอกจากนี้, ตารางนี้ได้รับการดูแลจากผู้ใช้, ทำให้ผู้ใช้สามารถใช้ ALTER, INSERT, DELETE, และ UPDATE.

เพื่อที่จะให้ตาราง materialized query คงอยู่ หรือ รีเฟรชตารางนั้นหลังจาก มันได้เกิดขึ้นแล้ว, ให้ใช้ข้อความ REFRESH TABLE. สิ่งนี้จะป็นสาเหตุให้ query ที่เชื่อมโยงกับตาราง materialized query ทำงาน และ ทำให้เกิดผลลัพธ์ของ query บรรจุอยู่ในตาราง. เพื่อให้ตาราง STRANS คงสภาพอยู่ตลอด, ให้รันคำสั่งต่อไปนี้:

```
REFRESH TABLE STRANS
```

คุณสามารถสร้างตาราง materialized query จากตารางฐานที่สร้างขึ้นแล้ว ตรงเท่าที่ผลลัพธ์ของ select-statement ได้เตรียมกลุ่มของคอลัมน์ที่ตรงกับคอลัมน์ในตารางที่สร้างขึ้นก่อนแล้ว (จำนวนคอลัมน์เท่ากัน และ definitions ของคอลัมน์เข้ากันได้). ตัวอย่างเช่น, ให้สร้างตาราง TRANSCOUNT. แล้ว, เปลี่ยน ตารางฐาน TRANSCOUNT ไปเป็นตาราง materialized query:

การสร้างตาราง:

```
CREATE TABLE TRANSCOUNT
  (ACCTID SMALLINT NOT NULL,
   LOCID SMALLINT,
   YEAR DATE
   CNT INTEGER)
```

คุณสามารถเปลี่ยนตารางนี้ไปเป็นตาราง materialized query:

```
ALTER TABLE TRANSCOUNT
  ADD MATERIALIZED QUERY
  (SELECT ACCTID, LOCID, YEAR, COUNT(*) AS CNT
   FROM TRANS
   GROUP BY ACCTID, LOCID, YEAR )
  DATA INITIALLY DEFERRED
  REFRESH DEFERRED
  MAINTAINED BY USER
```

ท้ายที่สุด, คุณก็ยังสามารถเปลี่ยนตาราง materialized query กลับไปเป็นตารางฐานเหมือนเดิมได้. ตัวอย่างเช่น:

```
ALTER TABLE TRANSCOUNT
  DROP MATERIALIZED QUERY
```

ในตัวอย่างนี้, ตาราง TRANSCOUNT ไม่ได้ถูกเอาออกไป, แต่มันไม่ได้เป็นตาราง materialized query อีกต่อไป.

หลักการที่เกี่ยวข้อง

“ตาราง, แถว, และคอลัมน์” ในหน้า 10

ตาราง เป็นการจัดการข้อมูลแบบสองด้านประกอบด้วยแถว และ คอลัมน์.

การประกาศตารางชั่วคราวแบบโกลบอล

คุณสามารถสร้างตารางชั่วคราวเพื่อใช้งานกับเซสชันปัจจุบันได้โดยการใช้คำสั่ง DECLARE GLOBAL TEMPORARY TABLE.

ตารางชั่วคราวจะไม่ปรากฏขึ้นในแค็ตตาล็อกระบบและไม่สามารถใช้งานร่วมกับเซสชันอื่นๆ ได้. เมื่อคุณสิ้นสุดเซสชัน, แถวของตารางจะถูกลบทิ้ง และตารางจะเลื่อนลงมา.

ไวยากรณ์ของข้อความนี้คล้ายกับ CREATE TABLE, รวมถึง LIKE และ AS clause.

ตัวอย่างเช่น, สร้างตารางชั่วคราว ORDERS:

```
DECLARE GLOBAL TEMPORARY TABLE ORDERS
  (PARTNO SMALLINT NOT NULL,
   DESCR VARCHAR(24),
   QONHAND INT)
  ON COMMIT DELETE ROWS
```

ตารางนี้จะถูกสร้างขึ้นใน QTEMP. หากต้องการอ้างอิงถึงตารางที่ใช้ชื่อแบบแผน, ให้ใช้ SESSION หรือ QTEMP. คุณสามารถใช้คำสั่ง SELECT, INSERT, UPDATE, และ DELETE กับตารางนี้, เช่นเดียวกับตารางอื่นๆทั่วไป. คุณสามารถเลื่อนตารางนี้ได้โดยใช้คำสั่ง DROP TABLE:

```
DROP TABLE ORDERS
```

ข้อมูลที่เกี่ยวข้อง

```
DECLARE GLOBAL TEMPORARY TABLE
```

การสร้างและเปลี่ยน identity column

ทุกครั้งที่เพิ่มแถวใหม่เข้าไปยังตารางด้วย identity column, ค่าของ identity column ในแถวใหม่นี้จะเพิ่มขึ้น (หรือลดลง) เพราะระบบ.

เฉพาะคอลัมน์ของประเภท SMALLINT, INTEGER, BIGINT, DECIMAL, หรือ NUMERIC ที่สามารถถูกสร้างเป็น identity column ได้. คุณมีสิทธิ์สร้าง identity column ได้หนึ่งคอลัมน์ต่อตาราง. เมื่อคุณเปลี่ยน definition ของตาราง, สามารถระบุเฉพาะคอลัมน์ที่คุณจะเพิ่มเป็น identity column ได้; ไม่สามารถระบุคอลัมน์ที่มีอยู่เดิมได้.

เมื่อคุณสร้างตาราง, คุณสามารถกำหนดคอลัมน์ในตารางให้เป็น identity column ได้. ยกตัวอย่างเช่น, ให้สร้างตาราง ORDERS โดยมีสามคอลัมน์ที่มีชื่อว่า ORDERNO, SHIPPED_TO, และ ORDER_DATE. กำหนด ORDERNO ให้เป็น identity column.

```
CREATE TABLE ORDERS
  (ORDERNO SMALLINT NOT NULL
   GENERATED ALWAYS AS IDENTITY
   (START WITH 500
    INCREMENT BY 1
    CYCLE),
  SHIPPED_TO VARCHAR (36) ,
  ORDER_DATE DATE)
```

คอลัมน์นี้จะถูกกำหนดด้วยค่าเริ่มต้นของ 500, เพิ่มขึ้นทีละหนึ่ง 1 เมื่อแทรกแถวใหม่, และจะหมุนเวียนกลับมาใช้ใหม่เมื่อถึงค่าสูงสุด. ในตัวอย่างนี้, ค่าสูงสุดสำหรับ identity column คือค่าสูงสุดสำหรับประเภทข้อมูล. เพราะประเภทข้อมูลถูกกำหนดไว้เป็น SMALLINT, ช่วงของค่าที่สามารถกำหนดให้กับ ORDERNO ได้จึงอยู่ระหว่าง 500 ถึง 32767. เมื่อค่าของคอลัมน์มีถึง 32767, ค่ามันจะกลับมาเริ่มต้นใหม่ที่ 500 อีกครั้ง. หาก 500 ยังคงถูกกำหนดให้กับคอลัมน์, และคีย์แบบเฉพาะบน identity column, จะมีข้อผิดพลาดเกิดขึ้น. การแทรกครั้งต่อไปจะพยายามใช้ 501. หากคุณไม่มีคีย์แบบเฉพาะที่ระบุไว้สำหรับ identity column, 500 จะถูกนำมาใช้อีกครั้ง, โดยไม่สนใจว่าค่ามันจะปรากฏกี่ครั้งในตาราง.

สำหรับช่วงค่าที่กว้างกว่า, ให้ระบุคอลัมน์ที่จะเป็น INTEGER หรือแม้แต่ BIGINT. ถ้าคุณต้องการให้ค่าของคอลัมน์ identity ลดลง, ให้ระบุค่าที่เป็นลบ สำหรับตัวเลือก INCREMENT. เป็นไปได้ที่ระบุช่วงจำนวนที่ถูกต้องโดยการใช้นิพจน์ MINVALUE และ MAXVALUE.

คุณสามารถดัดแปลงแอตทริบิวต์ของ identity column ที่มีอยู่เดิมโดยใช้ข้อความ ALTER TABLE. ตัวอย่างเช่น, หากคุณต้องการรีเซ็ตค่า identity column ด้วยค่าใหม่:

```
ALTER TABLE ORDER
  ALTER COLUMN ORDERNO
  RESTART WITH 1
```

คุณสามารถเลื่อน identity attribute จากคอลัมน์ได้:

```
ALTER TABLE ORDER
ALTER COLUMN ORDERNO
DROP IDENTITY
```

คอลัมน์ ORDERNO ยังคงเป็นคอลัมน์ SMALLINT, แต่ identity attribute จะถูกลบออกไป. ระบบจะไม่สร้างค่าสำหรับคอลัมน์นี้อีกแล้ว.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การเปรียบเทียบคอลัมน์ identity และลำดับ” ในหน้า 28

ขณะที่คอลัมน์ IDENTITY และ ลำดับมีลักษณะเหมือนกันในหลายๆทาง, แต่ก็ยังมีที่แตกต่างกันบ้าง.

“การแทรกค่าเข้าในคอลัมน์ identity” ในหน้า 99

คุณสามารถแทรกค่าเข้าในคอลัมน์ identity หรืออนุญาตให้ระบบแทรกค่าให้คุณ.

“การอัปเดต identity column” ในหน้า 103

คุณสามารถอัปเดตค่าใน identity column ให้เป็นค่าที่ระบุ หรือให้ระบบสร้างค่าใหม่.

การใช้ ROWID

การใช้ ROWID เป็นอีกวิธีหนึ่งที่ทำให้ระบบกำหนดค่าเฉพาะให้กับคอลัมน์ในตาราง. ROWID เหมือนกับคอลัมน์ identity, แต่แทนที่จะเป็นแอตทริบิวต์ของคอลัมน์ตัวเลข, มันเป็นชนิดข้อมูลที่แยกต่างหาก.

วิธีการสร้างตารางที่คล้ายกับตัวอย่างคอลัมน์ identity:

```
CREATE TABLE ORDERS
  (ORDERNO ROWID
  GENERATED ALWAYS,
  SHIPPED_TO VARCHAR (36) ,
  ORDER_DATE DATE)
```

การสร้าง และการใช้ลำดับ

ลำดับ คืออ็อบเจกต์ชนิดหนึ่งที่อนุญาตให้คุณสร้างค่าได้อย่างรวดเร็วและง่าย.

ลำดับจะเหมือนกับคอลัมน์ identity ในเรื่องที่ว่าทั้งคู่จะสามารถสร้างค่าที่เป็น unique. อย่างไรก็ตาม, ลำดับจะเป็นอ็อบเจกต์อิสระจากตาราง. อย่างเช่น, มันจะไม่ผูกติดกับคอลัมน์ และสามารถเรียกใช้งานแยกต่างหากได้. นอกจากนี้, มันจะไม่ถูกควบคุมให้เป็นส่วนหนึ่งของ transaction ของงานของหน่วยนั้นๆ.

คุณสามารถสร้างลำดับโดยใช้ข้อความ CREATE SEQUENCE. สำหรับตัวอย่างจะคล้ายกับตัวอย่างของคอลัมน์ identity, การสร้างลำดับ ORDER_SEQ:

```
CREATE SEQUENCE ORDER_SEQ
START WITH 500
INCREMENT BY 1
MAXVALUE 1000
CYCLE
CACHE 24
```

ลำดับนี้ถูกกำหนดให้เริ่มต้นค่าที่ 500, และเพิ่มขึ้นทีละ 1 ทุกๆครั้งที่ใช้งาน, และจะ will รีไซเคิลเมื่อถึงค่าสูงสุด. ในตัวอย่างนี้, ค่าสูงสุดสำหรับลำดับคือ 1000. เมื่อค่านี้ถึง 1000, มันจะกลับมาเริ่มต้นใหม่ที่ 500 อีกครั้ง.

ดังนั้นเมื่อมีการสร้างลำดับขึ้น, คุณสามารถแทรกค่าลงในคอลัมน์โดยใช้ลำดับ. ตัวอย่างเช่น, แทรกค่าถัดไปของลำดับ ORDER_SEQ ลงในตาราง ORDERS โดยมีคอลัมน์ ORDERNO และ CUSTNO.

ก่อนอื่น, ให้สร้างตาราง ORDERS:

```
CREATE TABLE ORDERS  
(ORDERNO SMALLINT NOT NULL,  
CUSTNO SMALLINT);
```

แล้ว, แทรกค่าลำดับ:

```
INSERT INTO ORDERS (ORDERNO, CUSTNO)  
VALUES (NEXT VALUE FOR ORDER_SEQ, 12)
```

การรันคำสั่งต่อไปนี้, จะส่งกลับค่าลงในคอลัมน์:

```
SELECT *  
FROM ORDERS
```

ตารางที่ 2. ผลลัพธ์สำหรับ SELECT จากตาราง ORDERS

ORDERNO	CUSTNO
500	12

ในตัวอย่างนี้, ค่าถัดมาสำหรับลำดับ ORDER ได้ถูกแทรก ลงไปในคอลัมน์ ORDERNO. ให้เรียกข้อความ INSERT อีกครั้ง. แล้วรัน SELECT.

ตารางที่ 3. ผลลัพธ์สำหรับ SELECT จากตาราง ORDERS

ORDERNO	CUSTNO
500	12
501	12

คุณยังสามารถแทรกค่าก่อนหน้าสำหรับลำดับ ORDER โดยการใช้งาน นิพจน์ PREVIOUS VALUE. คุณสามารถใช้ NEXT VALUE และ PREVIOUS VALUE ในนิพจน์ ดังต่อไปนี้:

- ภายใน *select-clause* ของข้อความ SELECT หรือข้อความ SELECT INTO ตราบเท่าที่ ข้อความไม่ได้ประกอบด้วยคีย์เวิร์ด DISTINCT, ประโยค GROUP BY, ประโยค ORDER BY, คีย์เวิร์ด UNION, คีย์เวิร์ด INTERSECT, หรือคีย์เวิร์ด EXCEPT
- ภายในประโยค VALUES ของข้อความ INSERT
- ภายใน *select-clause* ของ fullselect ของข้อความ INSERT
- ภายในประโยค SET ของการค้นหา หรือ ตำแหน่งข้อความ UPDATE, ถึงแม้ว่า NEXT VALUE ไม่สามารถระบุลงใน *select-clause* ของ subselect ของนิพจน์ในประโยค SET

คุณสามารถเปลี่ยนแปลงลำดับโดยใช้ข้อความ ALTER SEQUENCE. ลำดับ สามารถเปลี่ยนแปลงได้โดยแนวทางต่อไปนี้:

- การเริ่มทำ ลำดับ ต่อ
- เปลี่ยนส่วนเพิ่มระหว่างค่าลำดับที่จะเกิดขึ้นข้างหน้า

- การตั้ง หรือ การปรับค่า ต่ำสุด หรือ สูงสุด
- การเปลี่ยนจำนวนเลขแคชของลำดับ
- การเปลี่ยนแอตทริบิวต์ที่กำหนดว่า ลำดับ จะเป็นวัฏจักร หรือไม่
- การเปลี่ยนว่า เลขลำดับต้องถูกสร้างขึ้นตามลำดับที่ร้องขอหรือไม่

ตัวอย่างเช่น, เปลี่ยนส่วนเพิ่มของค่าของลำดับ ORDER จาก 1 ถึง 5:

```
ALTER SEQUENCE ORDER_SEQ
INCREMENT BY 5
```

หลังจากการเปลี่ยนแปลงเสร็จสิ้น, ให้รันข้อความ INSERT อีกครั้ง, แล้วใช้ SELECT. ดังนั้นตารางก็จะประกอบด้วยคอลัมน์ดังต่อไปนี้:

ตารางที่ 4. ผลลัพธ์สำหรับ SELECT จากตาราง ORDERS

ORDERNO	CUSTNO
500	12
501	12
528	12

โปรดสังเกตว่าค่าถัดไปที่ ลำดับใช้คือ 528. ในตอนแรก, หมายเลขนี้จะปรากฏไม่ถูกต้อง. อย่างไรก็ตาม, เมื่อมองตามเหตุการณ์ที่นำไปสู่การมอบหมายนี้. ชั้นแรก, เมื่อ ลำดับถูกสร้างขึ้นตามปกติ, ค่าของแคชจะถูกกำหนดเป็น 24. ระบบจะกำหนดค่า 24 ตัวแรกสำหรับ แคชนี้. ถัดมา, ลำดับจะมีการเปลี่ยนแปลง. เมื่อมีการใช้ข้อความ ALTER SEQUENCE, ระบบจะยกเลิกค่าที่กำหนดไว้ และเริ่มงานใหม่อีกครั้งด้วย ค่าที่มีอยู่ต่อไป; ในกรณีนี้ ค่าเริ่มต้นของแคชที่ 24, จะบวกเพิ่มขึ้น ไปอีก, 5. ถ้าเดิมข้อความ CREATE SEQUENCE ไม่มีประโยค CACHE, ระบบจะกำหนดค่าแคชดีฟอลต์เป็น 20 โดยอัตโนมัติ. ถ้าลำดับนั้นมีการเปลี่ยนแปลง, แล้วค่าที่ใช้ได้จะเป็น 25.

หลักการที่เกี่ยวข้อง

“ลำดับ (Sequences)” ในหน้า 12

sequence คืออ็อบเจกต์พื้นที่ข้อมูล ที่กำหนดให้วิธีการที่ง่ายและรวดเร็วในการสร้างหมายเลขเฉพาะ.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การเปรียบเทียบคอลัมน์ identity และลำดับ”

ขณะที่คอลัมน์ IDENTITY และ ลำดับมีลักษณะเหมือนกันในหลายๆทาง, แต่ก็ยังมีที่ แตกต่างกันบ้าง.

การเปรียบเทียบคอลัมน์ identity และลำดับ

ขณะที่คอลัมน์ IDENTITY และ ลำดับมีลักษณะเหมือนกันในหลายๆทาง, แต่ก็ยังมีที่ แตกต่างกันบ้าง.

ให้พิจารณาความแตกต่างเหล่านี้ ก่อนที่คุณจะตัดสินใจเลือกใช้.

คอลัมน์ identity มีลักษณะเฉพาะตัวดังต่อไปนี้:

- เราสามารถกำหนดคอลัมน์ identity เป็นเพียงส่วนหนึ่งของตารางได้เมื่อมีการสร้าง ตารางขึ้น. ครั้นเมื่อตารางถูกสร้างขึ้น, คุณไม่สามารถเปลี่ยนให้มันเพิ่มคอลัมน์ identity ได้. (อย่างไรก็ตาม, ลักษณะเฉพาะตัวของคอลัมน์ identity ที่เกิดขึ้นแล้วนั้นอาจเปลี่ยนแปลงได้.)

- คอลัมน์ identity จะสร้างค่าสำหรับตารางเดี่ยวโดยอัตโนมัติ.
- เมื่อคอลัมน์ identity ถูกกำหนดเป็น GENERATED ALWAYS, ค่าที่นำไปใช้ จะถูกสร้างโดยตัวจัดการฐานข้อมูลเสมอ. แอปพลิเคชัน จะถูกจำกัดให้ไม่สามารถ ใช้ค่าของตัวเองได้ ระหว่างการแก้ไขเนื้อหาของ ตาราง.
- เราสามารถใช้ฟังก์ชัน IDENTITY_VAL_LOCAL เพื่อดูค่าที่ถูกกำหนดล่าสุดสำหรับ คอลัมน์ identity.

ลำดับ มีลักษณะเฉพาะตัวดังนี้:

- ลำดับ เป็นอ็อบเจกต์ระบบชนิด *DTAARA ที่ไม่ผูกติดกับตาราง.
- ลำดับ จะเป็นตัวสร้างค่าเรียงลำดับที่สามารถนำไปใช้ในคำสั่ง SQL ใดๆ.
- มีนิพจน์อยู่สองแบบที่ใช้สำหรับเรียกค่าถัดไปใน ลำดับ ออกมา และ ใช้มองหาค่าก่อนหน้าที่ถูกกำหนดไว้สำหรับ ลำดับ. นิพจน์ PREVIOUS VALUE จะส่งกลับค่าที่ถูกกำหนดล่าสุดสำหรับลำดับที่ระบุ สำหรับข้อความก่อนหน้าในเซสชัน ปัจจุบัน. นิพจน์ NEXT VALUE จะส่งกลับค่าถัดไปสำหรับ ลำดับที่ระบุ. การใช้นิพจน์เหล่านี้ จะอนุญาตให้ค่าที่เหมือนกัน ถูกนำไปใช้ข้ามข้อความ SQL ได้หลายๆข้อความ ในหลายๆตาราง.

ขณะที่สิ่งเหล่านี้ไม่ใช่ลักษณะเฉพาะตัวทั้งหมดของทั้งสองรายการนี้, ลักษณะเฉพาะตัวเหล่านี้จะช่วยคุณในการตัดสินใจว่า จะใช้อะไร ขึ้นอยู่กับการออกแบบฐานข้อมูลของคุณ และแอปพลิเคชันที่ใช้ฐานข้อมูลนั้น.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การสร้างและเปลี่ยน identity column” ในหน้า 25

ทุกครั้งที่เพิ่มแถวใหม่เข้าไปยังตารางด้วย identity column, ค่าของ identity column ในแถวใหม่นี้จะเพิ่มขึ้น (หรือลดลง) เพราะระบบ.

“การสร้าง และการใช้ลำดับ” ในหน้า 26

ลำดับ คืออ็อบเจกต์ชนิดหนึ่ง ที่อนุญาตให้คุณสร้างค่าได้อย่างรวดเร็วและง่าย.

การสร้างเลเบลอธิบายโดยใช้คำสั่ง LABEL ON

บางครั้ง ชื่อตาราง, ชื่อคอลัมน์, ชื่อมุมมอง, ชื่อดรกรณี, ชื่อลำดับ, ชื่อ alias, หรือชื่อ SQL แพ็กเกจไม่ได้กำหนดข้อมูลของตารางที่แสดงอยู่บนจอแสดงผลแบบโต้ตอบของตารางไว้อย่างชัดเจน. คุณสามารถสร้างเลเบลคำอธิบายได้มากกว่านี้สำหรับชื่อเหล่านี้โดยใช้คำสั่ง LABEL ON.

สามารถเห็นเลเบลเหล่านี้ในแคตตาล็อก SQL ในคอลัมน์ LABEL .

คำสั่ง LABEL ON จะเป็นดังนี้:

```
LABEL ON
TABLE CORPDATA.DEPARTMENT IS 'Department Structure Table'
```

```
LABEL ON
COLUMN CORPDATA.DEPARTMENT.ADMRDEPT IS 'Reports to Dept.'
```

หลังจากคำสั่งเหล่านี้ถูกรันแล้ว, ตารางที่ชื่อ DEPARTMENT จะแสดงผลรายละเอียดข้อความ เป็น *Department Structure Table* และคอลัมน์ที่ชื่อ ADMRDEPT จะแสดงผลส่วนหัว *Reports to Dept.* เลเบลสำหรับตาราง, มุมมอง, ดรกรณี, ลำดับ, SQL แพ็กเกจ, และข้อความคอลัมน์ไม่สามารถมีความยาวเกิน 50 อักขระ และเลเบลสำหรับส่วนหัวของคอลัมน์ไม่สามารถมีความยาวเกิน 60 อักขระ (รวมถึงช่องว่าง). ต่อไปนี้คือตัวอย่างของคำสั่ง LABEL ON สำหรับส่วนหัวของคอลัมน์:

คำสั่ง LABEL ON มีส่วนหัวคอลัมน์ 1 และส่วนหัวคอลัมน์ 2.

```
*...+...1...+...2...+...3...+...4...+...5...+...6..*
LABEL ON COLUMN CORPDATA.EMPLOYEE.EMPNO IS
      'หมายเลขประจำตัว      พนักงาน'
```

คำสั่ง LABEL ON มีส่วนหัวคอลัมน์ 3 ระดับสำหรับคอลัมน์ SALARY.

```
*...+...1...+...2...+...3...+...4...+...5...+...6..*
LABEL ON COLUMN CORPDATA.EMPLOYEE.SALARY IS
      'เงินเดือน      ประจำปี      (เป็นดอลลาร์)'
```

คำสั่ง LABEL ON นี้จะลบส่วนหัวคอลัมน์ของ SALARY.

```
*...+...1...+...2...+...3...+...4...+...5...+...6..*
LABEL ON COLUMN CORPDATA.EMPLOYEE.SALARY IS ''
```

ตัวอย่างของส่วนหัวคอลัมน์ DBCS และระดับสองระดับที่ระบุไว้.

```
*...+...1...+...2...+...3...+...4...+...5...+...6..*
LABEL ON COLUMN CORPDATA.EMPLOYEE.SALARY IS
      '<AABBCCDD>      <EEFFGG>'
```

คำสั่ง LABEL ON จะมีข้อความคอลัมน์สำหรับคอลัมน์ EDLEVEL.

```
*...+...1...+...2...+...3...+...4...+...5...+...6..*
LABEL ON COLUMN CORPDATA.EMPLOYEE.EDLEVEL TEXT IS
      'จำนวนปีการศึกษาภาคบังคับ'
```

ข้อมูลที่เกี่ยวข้อง

LABEL ON

การอธิบาย SQL อ็อบเจ็กต์โดยใช้ COMMENT ON

หลังจากที่คุณสร้าง SQL อ็อบเจ็กต์แล้ว เช่น ตาราง, มุมมอง, ดรรชนี, แפקเกจ, โพรซีเจอร์, พารามิเตอร์, ชนิดที่ผู้ใช้กำหนดเอง, ฟังก์ชัน, ทรริกเกอร์, หรือลำดับ, คุณสามารถจัดหาข้อมูลเกี่ยวกับสิ่งเหล่านี้เพื่อไว้อ้างอิงในอนาคต. คุณสามารถเพิ่มข้อมูลได้โดยใช้คำสั่ง COMMENT ON.

ข้อมูลอาจเป็นวัตถุประสงค์ของอ็อบเจ็กต์, ที่ใช้, และเป็นสิ่งอื่นๆ ที่ผิดปกติ หรือเป็นสิ่งพิเศษ. คุณสามารถแทรกข้อมูลที่คล้ายคลึงกันเกี่ยวกับแต่ละคอลัมน์ของตารางหรือมุมมองได้. ข้อสังเกตจะมีประโยชน์มากหากชื่อของคุณไม่ได้ระบุเนื้อหาของคอลัมน์หรืออ็อบเจ็กต์ไว้อย่างชัดเจน. ในกรณีนั้น, ใช้หมายเหตุเพื่ออธิบายเนื้อหาของคอลัมน์หรืออ็อบเจ็กต์. โดยปกติ, หมายเหตุของคุณต้องมีอักขระไม่เกิน 2000 อักขระ, แต่สำหรับลำดับ จะมีความยาวมากที่สุด 500 อักขระ.

ตัวอย่างการใช้ COMMENT ON มีดังนี้:

```
COMMENT ON TABLE CORPDATA.EMPLOYEE IS
      'ตารางพนักงาน. แต่ละแถวในตารางนี้จะแสดง
      พนักงานหนึ่งคนของบริษัท.'
```

รับหมายเหตุหลังจากรันคำสั่ง COMMENT ON

หลังจากรันคำสั่ง COMMENT ON ของตาราง, หมายเหตุของคุณจะถูกเก็บไว้ในคอลัมน์ LONG_COMMENT ของ SYSTABLES. ข้อสังเกตสำหรับอ็อบเจ็กต์อื่นๆ จะถูกเก็บไว้ในคอลัมน์ LONG_COMMENT ของตารางเค็ตตาลีอ็อกที่เหมาะสม. หากแถวที่ระบุมีข้อสังเกตอยู่แล้ว, ข้อสังเกตเดิมจะถูกแทนที่ด้วยข้อสังเกตใหม่. ตัวอย่างต่อไปนี้รับข้อสังเกตที่เพิ่มโดยข้อความ COMMENT ON ในตัวอย่างก่อนหน้านี้:

```
SELECT LONG_COMMENT
FROM CORPDATA.SYSTABLES
WHERE NAME = 'EMPLOYEE'
```

ข้อมูลที่เกี่ยวข้อง

COMMENT ON

การเปลี่ยน definition ตาราง

การเปลี่ยน definition ของตารางจะให้คุณสามารถเพิ่มคอลัมน์ใหม่, เปลี่ยน definition ของคอลัมน์ที่มีอยู่ (เปลี่ยนความยาวของคอลัมน์, ค่าดีฟอลต์, และอื่นๆ), ลบคอลัมน์ที่มีอยู่เดิม, และเพิ่ม และลบข้อจำกัด.

definition ตารางจะถูกเปลี่ยนโดยใช้คำสั่ง SQL ALTER TABLE.

คุณสามารถเพิ่ม, เปลี่ยน, หรือลบคอลัมน์และเพิ่มหรือลบข้อจำกัดทั้งหมดด้วยคำสั่ง ALTER TABLE. อย่างไรก็ตาม, สามารถอ้างอิงถึงคอลัมน์เดี่ยวเพียงหนึ่งครั้งใน ADD COLUMN, ALTER COLUMN, และ DROP COLUMN clause. นั่นคือ, คุณไม่สามารถเพิ่มคอลัมน์และเปลี่ยนคอลัมน์นั้นในคำสั่ง ALTER TABLE เดียวกัน.

ข้อมูลที่เกี่ยวข้อง

ALTER TABLE

การเพิ่มคอลัมน์

เมื่อคุณเพิ่มคอลัมน์ใหม่เข้ายังตาราง, คอลัมน์จะถูก initialize ด้วยค่าดีฟอลต์สำหรับแถวทั้งหมดที่มีอยู่เดิม. หากไม่ได้ระบุค่าดีฟอลต์, จะต้องระบุค่าดีฟอลต์ด้วย.

คุณสามารถเพิ่มคอลัมน์ไปยังตารางโดยใช้ ADD COLUMN clause ของคำสั่ง SQL ALTER TABLE.

ตารางที่เปลี่ยนไปอาจประกอบด้วยคอลัมน์ไม่เกิน 8000. จำนวนการนับไบต์ของคอลัมน์ต้องไม่เกิน 32766 หรือ, หากมีการระบุคอลัมน์ VARCHAR หรือ VARGRAPHIC, 32740. หากมีการระบุคอลัมน์ LOB, จำนวนของการนับไบต์ของเร็กคอร์ดข้อมูลของคอลัมน์ต้องมีขนาดไม่เกิน 15 728 640.

ข้อมูลที่เกี่ยวข้อง

ALTER TABLE

การเปลี่ยนคอลัมน์

คุณสามารถเปลี่ยน definition คอลัมน์ในตารางได้โดยใช้ ALTER COLUMN clause ของคำสั่ง ALTER TABLE.

เมื่อคุณเปลี่ยนประเภทข้อมูลของคอลัมน์ที่มีอยู่, แอ็ททริบิวต์เดิมและใหม่ต้องทำงานร่วมกันได้. คุณสามารถเปลี่ยนอักขระ, กราฟิก, หรือ คอลัมน์ไบนารีได้เสมอจาก ความยาวคงที่ เป็นความยาวไม่คงที่ หรือ LOB; หรือ จากความยาวไม่คงที่ หรือ LOB เป็น ความยาวคงที่.

เมื่อคุณแปลงไปเป็นประเภทข้อมูลโดยมีความยาวเพิ่มขึ้น, ข้อมูลจะถูกเติมเต็มด้วยแพ็คอักขระที่เหมาะสม. เมื่อคุณแปลงไปเป็นประเภทข้อมูลซึ่งมีความยาวน้อยกว่า, ข้อมูลอาจหายเพราะเกิดการตัดปลาย. ข้อความสอบถามจะถามให้คุณยืนยันการร้องขอ.

หากคุณมีคอลัมน์ที่ไม่อนุญาตให้มีค่าเป็นศูนย์และคุณต้องการเปลี่ยนให้เป็นคอลัมน์ที่อนุญาตให้มีค่าเป็นศูนย์, ให้ใช้ DROP NOT NULL clause. หากคุณมีคอลัมน์ที่อนุญาตให้มีค่าศูนย์และคุณต้องการป้องกันการใส่ค่าศูนย์, ให้ใช้ SET NOT NULL clause. หากค่าใดค่าหนึ่งของค่าที่มีอยู่ในคอลัมน์นั้นเป็นค่าศูนย์, ALTER TABLE จะไม่ถูกเรียกทำงานและจะเกิด SQLCODE of -190.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การแปลงที่ได้รับอนุญาต”

เมื่อคุณเปลี่ยนประเภทข้อมูลของคอลัมน์ที่มีอยู่, แอ็ททริบิวต์เดิมและใหม่ต้องทำงานร่วมกันได้.

ข้อมูลที่เกี่ยวข้อง

ALTER TABLE

การแปลงที่ได้รับอนุญาต

เมื่อคุณเปลี่ยนประเภทข้อมูลของคอลัมน์ที่มีอยู่, แอ็ททริบิวต์เดิมและใหม่ต้องทำงานร่วมกันได้.

ตารางที่ 5. การแปลงที่ได้รับอนุญาต

FROM data type	TO data type
Decimal	Numeric
Decimal	Bigint, Integer, Smallint
Decimal	Float
Numeric	Decimal
Numeric	Bigint, Integer, Smallint
Numeric	Float
Bigint, Integer, Smallint	Decimal
Bigint, Integer, Smallint	Numeric
Bigint, Integer, Smallint	Float
Float	Numeric
Float	Bigint, Integer, Smallint
Character	DBCS-open
Character	UCS-2 or UTF-16 graphic
DBCS-open	Character
DBCS-open	UCS-2 or UTF-16 graphic
DBCS-either	Character
DBCS-either	DBCS-open
DBCS-either	UCS-2 or UTF-16 graphic
DBCS-only	DBCS-open

ตารางที่ 5. การแปลงที่ได้รับอนุญาต (ต่อ)

FROM data type	TO data type
DBCS-only	DBCS graphic
DBCS-only	UCS-2 or UTF-16 graphic
DBCS graphic	UCS-2 or UTF-16 graphic
UCS-2 or UTF-16 graphic	Character
UCS-2 or UTF-16 graphic	DBCS-open
UCS-2 or UTF-16 graphic	DBCS graphic
distinct type	source type
source type	distinct type

เมื่อตัดแปลงคอลัมน์ที่มีอยู่, เฉพาะแอ็ททริบิวต์ที่คุณระบุไว้เท่านั้นที่จะเปลี่ยนไป. แอ็ททริบิวต์อื่นๆ ทั้งหมดจะไม่ถูกเปลี่ยนแปลง. ตัวอย่างเช่น, สมมติว่า definition ตารางต่อไปนี้คือ:

```
CREATE TABLE EX1 (COL1 CHAR(10) DEFAULT 'COL1',
                  COL2 VARCHAR(20) ALLOCATE(10) CCSID 937,
                  COL3 VARGRAPHIC(20) ALLOCATE(10)
                  NOT NULL WITH DEFAULT)
```

หลังจากรันข้อความ ALTER TABLE ต่อไปนี้:

```
ALTER TABLE EX1 ALTER COLUMN COL2 SET DATA TYPE VARCHAR(30)
ALTER COLUMN COL3 DROP NOT NULL
```

COL2 จะยังคงมีความยาวที่ถูกจัดสรรเท่ากับ 10 และ CCSID 937, และ COL3 ยังคงมีความยาวที่ถูกจัดสรรเป็น 10.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การเปลี่ยนคอลัมน์” ในหน้า 31

คุณสามารถเปลี่ยน definition คอลัมน์ในตารางได้โดยใช้ ALTER COLUMN clause ของคำสั่ง ALTER TABLE.

การลบคอลัมน์

คุณสามารถลบคอลัมน์โดยใช้ DROP COLUMN clause ของคำสั่ง ALTER TABLE.

คุณสามารถลบคอลัมน์โดยใช้ DROP COLUMN clause ของคำสั่ง ALTER TABLE.

การลบคอลัมน์จะลบคอลัมน์นั้นออกจาก definition ตาราง. หากมีการระบุ CASCADE, มุมมองใดๆ, ดรรชนีใดๆ, และข้อจำกัดใดๆ ที่ขึ้นอยู่กับคอลัมน์นั้นจะถูกลบออกไปเช่นกัน. หากมีการระบุ RESTRICT, และมุมมองใดๆ, ดรรชนีใดๆ, หรือข้อจำกัดที่ขึ้นอยู่กับคอลัมน์, คอลัมน์จะไม่ถูกลบออกไป และจะมีการออกคำสั่ง SQLCODE เป็น -196.

```
ALTER TABLE DEPT
DROP COLUMN NUMDEPT
```

ข้อมูลที่เกี่ยวข้อง

```
ALTER TABLE
```

ลำดับการดำเนินการของคำสั่ง ALTER TABLE

คำสั่ง ALTER TABLE จะถูกใช้งานเป็นเซตของขั้นตอนที่แสดง.

1. ลบข้อจำกัด
2. ลบตาราง materialized query
3. ลบข้อมูลพาร์ติชัน
4. ลบคอลัมน์ที่มีการระบุอ็อปชัน RESTRICT
5. เปลี่ยน definition คอลัมน์ (หมายรวมถึงการเพิ่มคอลัมน์และการลบคอลัมน์ที่มีการระบุอ็อปชัน CASCADE)
6. ใส่เพิ่ม หรือเปลี่ยนตาราง materialized query
7. ใส่เพิ่มพาร์ติชันในตาราง
8. เพิ่มข้อจำกัด

ภายในแต่ละขั้นตอน, ลำดับที่คุณระบุ clause คือลำดับที่คุณดำเนินการ, โดยมี exception หนึ่งข้อ. หากคอลัมน์ใดคอลัมน์หนึ่งถูกลบออก, การดำเนินการนั้นจะเสร็จสิ้นแบบโลจิคัลก่อนที่ definition ของคอลัมน์ใดๆ จะถูกเพิ่มหรือเปลี่ยนไป, ในกรณีที่ความยาวเรกคอร์ดเพิ่มขึ้นเนื่องมาจากคำสั่ง ALTER TABLE.

การสร้างและการใช้งานชื่อ ALIAS

เมื่อคุณอ้างถึงตารางหรือมุมมองที่มีอยู่เดิม, หรือถึงไฟล์ฟิสิคัลที่ประกอบด้วยเมมเบอร์จำนวนมาก, คุณสามารถเลี่ยงการใช้การบันทึกทับไฟล์โดยการสร้าง alias. คุณสามารถใช้คำสั่ง SQL CREATE ALIAS เพื่อทำเช่นนี้.

คุณสามารถสร้าง alias สำหรับ:

- ตารางหรือมุมมอง
- เมมเบอร์ของตาราง

alias ของตารางจะกำหนดชื่อสำหรับไฟล์, รวมถึงชื่อเมมเบอร์เฉพาะ. คุณสามารถใช้ชื่อ alias นี้ในคำสั่ง SQL โดยวิธีเดียวกับที่ใช้ในชื่อของตาราง. ต่างจากการบันทึกทับค่าเดิม, ชื่อ alias คืออ็อบเจกต์ที่มีอยู่จนกว่าจะถูกลบทิ้ง.

ตัวอย่างเช่น, หากมีไฟล์เมมเบอร์จำนวนมาก MYLIB.MYFILE พร้อมด้วยเมมเบอร์ MBR1 และ MBR2, สามารถสร้าง alias ไว้สำหรับเมมเบอร์ที่สอง เพื่อที่ว่า SQL จะสะดวกในการอ้างถึงเมมเบอร์ที่สองนั้นได้.

```
CREATE ALIAS MYLIB.MYMBR2_ALIAS FOR MYLIB.MYFILE (MBR2)
```

เมื่อมีการระบุ alias MYLIB.MYMBR2_ALIAS ไว้บนข้อความการแทรกต่อไปนี่, ค่าจะถูกแทรกลงยังเมมเบอร์ MBR2 ใน MYLIB.MYFILE.

```
INSERT INTO MYLIB.MYMBR2_ALIAS VALUES('ABC', 6)
```

สามารถระบุชื่อ alias ไว้บนคำสั่ง DDL. สมมติว่า alias MYLIB.MYALIAS มีอยู่และเป็น alias สำหรับตาราง MYLIB.MYTABLE. ตัวอย่างคำสั่ง DROP จะลบตาราง MYLIB.MYTABLE.

```
DROP TABLE MYLIB.MYALIAS
```

หากคุณต้องการลบชื่อ alias แทน, โปรดระบุคีย์เวิร์ด ALIAS ไว้บนคำสั่ง:

```
DROP ALIAS MYLIB.MYALIAS
```


ข้อมูลที่เกี่ยวข้อง

CREATE ALIAS

การสร้างและใช้มุมมอง

สามารถใช้มุมมองเพื่อเข้าใช้งานข้อมูลในตารางหนึ่งตารางหรือมากกว่าหรือมุมมองหนึ่งมุมมองหรือมากกว่าได้. คุณสามารถสร้างมุมมองได้โดยใช้คำสั่ง SELECT.

ตัวอย่างเช่น, หากต้องการสร้างมุมมองที่เลือกเฉพาะนามสกุลและแผนกของผู้จัดการทั้งหมด, โปรดระบุ:

```
CREATE VIEW CORPDATA.EMP_MANAGERS AS
  SELECT LASTNAME, WORKDEPT FROM CORPDATA.EMPLOYEE
  WHERE JOB = 'MANAGER'
```

เมื่อคุณสร้างมุมมองแล้ว, คุณสามารถใช้งานมุมมองนั้นในคำสั่ง SQL เหมือนกับชื่อตาราง. คุณสามารถเปลี่ยนข้อมูลในตารางฐาน. คำสั่ง SELECT ต่อไปนี้จะแสดงเนื้อหาของ EMP_MANAGERS:

```
SELECT *
  FROM CORPDATA.EMP_MANAGERS
```

ผลที่ได้คือ:

LASTNAME	WORKDEPT
THOMPSON	B01
KWAN	C01
GEYER	E01
STERN	D11
PULASKI	D21
HENDERSON	E11
SPENSER	E21

หากรายการให้เลือกมีส่วนประกอบนอกเหนือจากคอลัมน์เช่นนิพจน์, ฟังก์ชัน, ค่าคงที่, หรือการลงทะเบียเป็นพิเศษ, และ AS clause ไม่ได้ถูกใช้งานเพื่อตั้งชื่อคอลัมน์, ต้องระบุรายการคอลัมน์สำหรับมุมมอง. ในตัวอย่างต่อไปนี้, คอลัมน์ของมุมมองคือ LASTNAME และ YEARSOFSERVICE.

```
CREATE VIEW CORPDATA.EMP_YEARSOFSERVICE
  (LASTNAME, YEARSOFSERVICE) AS
  SELECT LASTNAME, YEAR (CURRENT DATE - HIREDATE)
  FROM CORPDATA.EMPLOYEE
```

เนื่องจากผลลัพธ์ของการสอบถามมุมมองนี้เปลี่ยนตามการเปลี่ยนปีปัจจุบัน, ซึ่งไม่ได้รวมอยู่ในที่นี่.

มุมมองก่อนหน้าสามารถถูกกำหนดได้โดยการใช้ AS clause ในรายการให้เลือกเพื่อตั้งชื่อคอลัมน์ในมุมมอง. ตัวอย่างเช่น:

```
CREATE VIEW CORPDATA.EMP_YEARSOFSERVICE AS
  SELECT LASTNAME,
         YEARS (CURRENT_DATE - HIREDATE) AS YEARSOFSERVICE
  FROM CORPDATA.EMPLOYEE
```

การใช้คีย์เวิร์ด UNION, คุณสามารถรวมการเลือกย่อยสองรายการหรือมากกว่าเพื่อสร้างมุมมองเดียว. ตัวอย่างเช่น:

```
CREATE VIEW D11_EMPS_PROJECTS AS
  (SELECT EMPNO
   FROM CORPDATA.EMPLOYEE
   WHERE WORKDEPT = 'D11'
  UNION
  SELECT EMPNO
   FROM CORPDATA.EMPPROJACT
   WHERE PROJNO = 'MA2112' OR
        PROJNO = 'MA2113' OR
        PROJNO = 'AD3111')
```

ผลในมุมมองและข้อมูลต่อไปนี้:

ตารางที่ 6. การสร้างมุมมองให้เป็นผลลัพธ์ UNION

EMPNO

000060

000150

000160

000170

000180

000190

000200

000210

000220

000230

000240

200170

200220

สามารถสร้างมุมมองด้วยการเรียงลำดับที่มีทำงานก็ต่อเมื่อคำสั่ง CREATE VIEW ถูกรัน. ลำดับการจัดเรียงประยุกต์ใช้กับอักขระทั้งหมด, หรือ UCS-2 หรือกราฟิก UTF-16 เปรียบเทียบในการเลือกย่อยของคำสั่ง CREATE VIEW.

มุมมองสามารถสร้างได้โดยใช้ WITH CHECK OPTION เพื่อระบุระดับของการตรวจสอบที่ควรดำเนินการ เมื่อแทรกข้อมูลหรืออัปเดตข้อมูลผ่านมุมมอง.

หลักการที่เกี่ยวข้อง

“การดึงข้อมูลโดยใช้คำสั่ง SELECT” ในหน้า 42

ศึกษาถึงวิธีต่างๆ ที่ทำให้เคียวรีของคุณเก็บข้อมูลโดยใช้คำสั่ง SELECT. วิธีหนึ่งในการทำเช่นนี้คือการใช้คำสั่ง SELECT ในโปรแกรมเพื่อดึงค่าเฉพาะ (ตัวอย่างเช่น, แถวข้อมูลพนักงาน). นอกจากนี้, คุณสามารถใช้ clause เพื่อรวบรวมข้อมูลในแบบที่ต้องการได้.

“ลำดับการเรียง และ normalization ใน SQL” ในหน้า 119

ลำดับการจัดเรียงกำหนดความสัมพันธ์ของอักขระในชุดอักขระ เมื่อมีการเปรียบเทียบหรือจัดลำดับ. Normalization อนุญาตให้คุณเปรียบเทียบสตริงที่มีอักขระแบบผสม.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การใช้คีย์เวิร์ด UNION เพื่อรวมการเลือกย่อย” ในหน้า 82

การใช้คีย์เวิร์ด UNION, คุณสามารถรวมการเลือกย่อยมากกว่าสองได้ เพื่อสร้าง fullselect.

ข้อมูลที่เกี่ยวข้อง

CREATE VIEW

WITH CHECK OPTION บนมุมมอง

WITH CHECK OPTION คือข้อความตัวเลือกบนคำสั่ง CREATE VIEW ที่ระบุระดับการตรวจสอบที่ต้องดำเนินการ เมื่อมีการแทรกหรืออัปเดตคำสั่งผ่านมุมมอง. หากมีการระบุข้อผิดพลาด, ทุกๆ แถวที่ถูกแทรกหรืออัปเดตผ่านมุมมอง จะต้องตรงตาม definition ของมุมมองนั้น.

ไม่สามารถระบุ WITH CHECK OPTION ได้หากมุมมองเป็นแบบอ่านอย่างเดียว. definition ของมุมมองจะต้องไม่รวมการสืบค้นย่อย.

หากมุมมองถูกสร้างขึ้นโดยไม่มี WITH CHECK OPTION clause, การแทรกและการอัปเดตที่กระทำบนมุมมอง จะไม่ถูกตรวจสอบว่าตรงตาม definition ของมุมมองหรือไม่. แต่อาจมีการตรวจสอบบางอย่างอยู่หากมุมมองขึ้นโดยตรงหรือโดยอ้อมกับมุมมองอื่นซึ่งประกอบด้วย WITH CHECK OPTION. เนื่องจากไม่ได้ใช้ definition ของมุมมอง, จึงอาจมีการแทรกหรืออัปเดตแถวผ่านมุมมองที่ไม่ตรงกับ definition ของมุมมอง. นั่นหมายความว่า แถวไม่สามารถถูกเลือกได้อีกครั้งเมื่อมีการใช้มุมมอง.

ข้อมูลที่เกี่ยวข้อง

CREATE VIEW

WITH CASCADED CHECK OPTION:

WITH CASCADED CHECK OPTION ระบุว่า ทุกๆ แถวที่ถูกแทรก หรืออัปเดตผ่านมุมมองจะต้องตรงตาม definition ของมุมมอง.

นอกจากนี้, เงื่อนไขการค้นหามุมมอง dependent ทั้งหมดจะถูกตรวจสอบเมื่อมีการแทรกหรืออัปเดตแถว. หากแถวไม่ตรงตาม definition ของมุมมอง, จะไม่สามารถเรียกแถวดังกล่าวออกมาได้ด้วยการใช้มุมมอง.

สำหรับตัวอย่าง, ให้พิจารณามุมมองที่แก้ไขได้ดังต่อไปนี้:

```
CREATE VIEW V1 AS SELECT COL1  
FROM T1 WHERE COL1 > 10
```

เนื่องจากไม่มีการระบุ WITH CHECK OPTION, คำสั่ง INSERT ต่อไปนี้จึงใช้ได้ แม้ว่าค่าที่ถูกแทรกจะไม่เป็นไปตามเงื่อนไขการค้นหาของมุมมอง.

```
INSERT INTO V1 VALUES (5)
```

สร้างอีกมุมมองหนึ่งที่ V1, ระบุ WITH CASCADED CHECK OPTION:

```
CREATE VIEW V2 AS SELECT COL1  
FROM V1 WITH CASCADED CHECK OPTION
```

ข้อความ INSERT ดังต่อไปนี้ล้มเหลวเนื่องจากการสร้างแถวซึ่งไม่เป็นไปตาม คำจำกัดความของ V2:

```
INSERT INTO V2 VALUES (5)
```

พิจารณาหนึ่งมุมมองหรือมากกว่าที่สร้างขึ้นที่ V2:

```
CREATE VIEW V3 AS SELECT COL1  
FROM V2 WHERE COL1 < 100
```

คำสั่ง INSERT ต่อไปนี้ใช้ไม่ได้เนื่องจาก V3 ต้องอิงกับ V2, และ V2 มี WITH CASCADED CHECK OPTION.

```
INSERT INTO V3 VALUES (5)
```

อย่างไรก็ตาม, คำสั่ง INSERT ต่อไปนี้ใช้ได้เนื่องจากตรงตาม definition ของ V2. เนื่องจาก V3 ไม่มี WITH CASCADED CHECK OPTION, จึงไม่ใช่เรื่องสำคัญที่ว่าคำสั่งดังกล่าว จะไม่ตรงตาม definition ของ V3.

```
INSERT INTO V3 VALUES (200)
```

WITH LOCAL CHECK OPTION:

WITH LOCAL CHECK OPTION clause เหมือนกับ WITH CASCADED CHECK OPTION clause เว้นแต่คุณสามารถอัปเดตแถวได้ซึ่งทำให้ไม่สามารถเรียกแถวออกมาผ่านทางมุมมองอีกต่อไป. กรณีนี้จะเกิดขึ้นได้ต่อเมื่อมุมมองนั้นต้องอิงโดยตรงหรือโดยอ้อมกับมุมมองที่ถูกกำหนดโดยไม่มี WITH CHECK OPTION clause.

ตัวอย่างเช่น, ให้พิจารณามุมมองที่อัปเดตได้เหมือนกัน ซึ่งใช้อยู่ในตัวอย่างก่อนหน้านี้:

```
CREATE VIEW V1 AS SELECT COL1  
FROM T1 WHERE COL1 > 10
```

สร้างมุมมองที่สองบน V1, โดยครั้งนี้ให้ระบุ WITH LOCAL CHECK OPTION:

```
CREATE VIEW V2 AS SELECT COL1  
FROM V1 WITH LOCAL CHECK OPTION
```

คำสั่ง INSERT เดียวกันที่ใช้ไม่ได้ในตัวอย่าง CASCADED CHECK OPTION ก่อนหน้านี้ จะสามารถใช้ได้ในครั้งนี้ เนื่องจาก V2 ไม่มีเงื่อนไขการค้นหาใดๆ, และเงื่อนไขการค้นหาของ V1 ไม่จำเป็นต้องถูกตรวจสอบ เนื่องจาก V1 ไม่ได้ระบุอ็อปชันการตรวจสอบ.

```
INSERT INTO V2 VALUES (5)
```

ให้พิจารณาหนึ่งมุมมอง หรือมากกว่าที่สร้างขึ้นบน V2:

```
CREATE VIEW V3 AS SELECT COL1  
FROM V2 WHERE COL1 < 100
```

INSERT ต่อไปนี้จะใช้ได้อีกครั้งเพราะเงื่อนไขการค้นหาบน V1 ไม่ได้ถูกตรวจสอบเนื่องจากมี WITH LOCAL CHECK OPTION บน V2, เทียบกับ WITH CASCADED CHECK OPTION ในตัวอย่างก่อนหน้านี้.

```
INSERT INTO V3 VALUES (5)
```

ความแตกต่างระหว่าง LOCAL และ CASCADED CHECK OPTION อยู่ที่จำนวนครั้งในการตรวจสอบเงื่อนไขการค้นหาของมุมมอง dependent เมื่อมีการแทรกหรืออัปเดตแถว.

- WITH LOCAL CHECK OPTION ระบุว่าเงื่อนไขการตรวจสอบเฉพาะมุมมอง dependent ที่มี WITH LOCAL CHECK OPTION หรือ WITH CASCADED CHECK OPTION จะถูกตรวจสอบเมื่อมีการแทรกหรืออัปเดตแถว.
- WITH CASCADED CHECK OPTION ระบุว่าเงื่อนไขการตรวจสอบของ dependent view ทั้งหมดจะถูกตรวจสอบ เมื่อมีการแทรกหรืออัปเดตแถว.

ข้อพิจารณาการตรวจสอบแบบต่อเรียง:

ตัวอย่างนี้อธิบายถึงการใช้ข้อพิจารณาการตรวจสอบแบบต่อเรียง.

ใช้ตารางและมุมมองต่อไปนี้:

```
CREATE TABLE T1 (COL1 CHAR(10))

CREATE VIEW V1 AS SELECT COL1
FROM T1 WHERE COL1 LIKE 'A%'

CREATE VIEW V2 AS SELECT COL1
FROM V1 WHERE COL1 LIKE '%Z'
WITH LOCAL CHECK OPTION

CREATE VIEW V3 AS SELECT COL1
FROM V2 WHERE COL1 LIKE 'AB%'

CREATE VIEW V4 AS SELECT COL1
FROM V3 WHERE COL1 LIKE '%YZ'
WITH CASCADED CHECK OPTION

CREATE VIEW V5 AS SELECT COL1
FROM V4 WHERE COL1 LIKE 'ABC%'
```

ระบบจะตรวจสอบเงื่อนไขการค้นหาต่างๆ โดยขึ้นอยู่กับว่ามุมมองใดที่ทำงานอยู่โดยใช้ INSERT หรือ UPDATE.

- หาก V1 ทำงานอยู่, จะไม่มีการตรวจสอบเงื่อนไขใดๆ เนื่องจาก V1 ไม่ได้ระบุ WITH CHECK OPTION.
- หาก V2 ทำงานอยู่,
 - COL1 ต้องสิ้นสุดด้วยตัวอักษร Z, แต่ไม่จำเป็นต้องเริ่มต้นด้วยตัวอักษร A. ทั้งนี้เนื่องจากตัวเลือกการตรวจสอบเป็น LOCAL, และมุมมอง V1 ไม่ได้มีการระบุตัวเลือกการตรวจสอบ.
- หาก V3 ทำงานอยู่,
 - COL1 จะต้องจบด้วยตัวอักษร Z, แต่ไม่จำเป็นต้องขึ้นต้นด้วยตัวอักษร A, V3 ไม่ได้ระบุข้อพิจารณาการตรวจสอบไว้, ดังนั้นจึงไม่จำเป็นต้องเป็นไปตามเงื่อนไขการค้นหาของตนเอง. อย่างไรก็ตาม, ต้องมีการตรวจสอบเงื่อนไขการค้นหาสำหรับ V2 เนื่องจาก V3 ถูกกำหนดไว้บน V2, และ V2 มีข้อพิจารณาการตรวจสอบ.
- หาก V4 ทำงานอยู่,

- COL1 ต้องขึ้นต้นด้วย 'AB', และจบด้วย 'YZ'. เนื่องจาก V4 มีการระบุ WITH CASCADED CHECK OPTION, ทุกๆ เงื่อนไขการค้นหาสำหรับทุกมุมมองที่ V4 ต้องยึดตามจึงต้องถูกตรวจสอบ.
- หาก V5 ทำงานอยู่,
 - COL1 ต้องขึ้นต้นด้วย 'AB', ไม่จำเป็นต้องเป็น 'ABC'. ที่เป็นเช่นนี้เพราะ V5 ไม่ได้ระบุอ็อปชันการตรวจสอบ, ดังนั้น จึงไม่จำเป็นต้องตรวจสอบเงื่อนไขการค้นหาของตนเอง. อย่างไรก็ตาม, เนื่องจาก V5 ถูกระบุไว้บน V4, และ V4 มีอ็อปชันการตรวจสอบแบบต่อเรียง, ดังนั้นจึงต้องมีการตรวจสอบเงื่อนไขการค้นหาทั้งหมดของ V4, V3, V2, และ V1. กล่าวคือ, COL1 ต้องขึ้นต้นด้วย 'AB' และจบด้วย 'YZ'.

หาก V5 ถูกสร้างขึ้นด้วย WITH LOCAL CHECK OPTION, การทำงานบน V5 ย่อมหมายถึงว่า COL1 ต้องขึ้นต้นด้วย 'ABC' และจบด้วย 'YZ'. LOCAL CHECK OPTION ได้ใส่ข้อกำหนดเพิ่มเติมว่าอักขระตัวที่สามต้องเป็น 'C'.

การเพิ่มดรรชนี

คุณสามารถใช้ดรรชนีเพื่อเรียงลำดับ และเลือกข้อมูล. นอกจากนี้, ดรรชนียังช่วย ระบบให้เรียกข้อมูลออกมาได้เร็วขึ้น เพื่อประสิทธิภาพของการสอบถามที่ดีกว่าเดิม.

การใช้ข้อความ CREATE INDEX เพื่อสร้างดรรชนี. ตัวอย่างต่อไปนี้จะสร้างดรรชนีของคอลัมน์ *LASTNAME* ในตาราง *CORPDATA.EMPLOYEE*:

```
CREATE INDEX CORPDATA.INX1 ON CORPDATA.EMPLOYEE (LASTNAME)
```

คุณสามารถสร้างดรรชนีจำนวนเท่าใดก็ได้. อย่างไรก็ตาม, เนื่องจากระบบมีการปรับปรุง ดรรชนี, ดรรชนีที่มีขนาดใหญ่สามารถส่งผลกระทบต่อประสิทธิภาพการทำงานได้. ดรรชนีประเภทหนึ่ง, ซึ่งก็คือดรรชนี vector แบบเข้ารหัส (EVI), ทำให้สแกนได้อย่างรวดเร็วซึ่งทำให้ประมวลผลแบบขนานได้ง่าย.

หากดรรชนีที่ถูกสร้างมีแอตทริบิวต์เดียวกันกับดรรชนีที่มีอยู่เดิม, ดรรชนีใหม่จะใช้งาน binary tree ของดรรชนีที่มีอยู่เดิมร่วมกัน. มิฉะนั้น, binary tree อื่นจะถูกสร้างขึ้นมา. หากแอตทริบิวต์ของดรรชนีใหม่เป็นอันเดียวกันกับอีกดรรชนีหนึ่ง, เว้นเสียแต่ว่าดรรชนีใหม่มีคอลัมน์น้อยลง, binary tree จะยังคงถูกสร้างขึ้นมา. มันยังคงถูกสร้างเนื่องจากคอลัมน์พิเศษ จะป้องกันดรรชนีจากการใช้งานโดยเคอร์เซอร์หรือข้อความ UPDATE ที่อัปเดต คอลัมน์พิเศษเหล่านั้น.

ดรรชนีจะถูกสร้างขึ้นด้วยการเรียงลำดับที่ทำงานอยู่ขณะที่ข้อความ CREATE INDEX ถูกรัน. การเรียงลำดับจะใช้งานกับฟิลด์แบบอักขระ SBCS ทั้งหมด, หรือ UCS-2 หรือ ฟิลด์กราฟิก UTF-16 ของดรรชนี.

หลักการที่เกี่ยวข้อง

“ลำดับการเรียง และ normalization ใน SQL” ในหน้า 119

ลำดับการจัดเรียงกำหนดความสัมพันธ์ของอักขระในชุดอักขระ เมื่อมีการเปรียบเทียบหรือจัดลำดับ. Normalization อนุญาตให้คุณเปรียบเทียบสตริงที่มีอักขระแบบผสม.

ข้อมูลที่เกี่ยวข้อง

CREATE INDEX

ยุทธวิธีในการสร้างดรรชนี

แคตตาล็อกในการออกแบบฐานข้อมูล

แคตตาล็อกจะถูกสร้างขึ้นโดยอัตโนมัติเมื่อคุณสร้างแบบแผน. มีแคตตาล็อกที่ใช้งานระบบซึ่งอยู่ในไลบรารี QSYS2 เสมอ.

เมื่อ SQL อ็อบเจกต์ถูกสร้างขึ้นในแบบแผน, ข้อมูลจะถูกเพิ่มเข้าไปยังทั้งตารางแคตตาล็อกระบบและตารางแคตตาล็อกแบบแผน. เมื่อ SQL อ็อบเจกต์ถูกสร้างขึ้นในไลบรารี, จะมีเฉพาะแคตตาล็อก QSYS2 ที่ถูกอัปเดต. ตารางที่สร้างด้วย DECLARE GLOBAL TEMPORARY TABLE จะไม่ถูกเพิ่มเข้ายังแคตตาล็อก.

เมื่อตัวอย่างต่อไปนี้แสดงให้เห็น, คุณสามารถแสดงผลข้อมูลแคตตาล็อกได้. คุณไม่สามารถ INSERT, DELETE, หรือ UPDATE ข้อมูลแคตตาล็อกได้. คุณต้องมี privilege SELECT ในมุมมองแคตตาล็อกเพื่อรันตัวอย่างต่อไปนี้.

ข้อมูลที่เกี่ยวข้อง

มุมมองแคตตาล็อก

การรับข้อมูลแคตตาล็อกเกี่ยวกับตาราง

มุมมอง SYSTABLES จะมีแถวสำหรับทุกตาราง และทุกมุมมองในแบบแผน SQL. ซึ่งจะบอกคุณว่าอ็อบเจกต์เป็นตาราง หรือ มุมมอง, ชื่ออ็อบเจกต์, เจ้าของอ็อบเจกต์, อยู่ในแบบแผน SQL ใด, และอื่นๆ.

คำสั่งตัวอย่างต่อไปนี้จะแสดงข้อมูลในตาราง CORPDATA.DEPARTMENT:

```
SELECT *
FROM CORPDATA.SYSTABLES
WHERE TABLE_NAME = 'DEPARTMENT'
```

การรับข้อมูลแคตตาล็อกเกี่ยวกับคอลัมน์

มุมมอง SYSCOLUMNS จะมีแถวสำหรับแต่ละคอลัมน์ของทุกตาราง และมุมมองในแบบแผน.

ข้อความตัวอย่างต่อไปนี้จะแสดงชื่อคอลัมน์ในตาราง CORPDATA.DEPARTMENT:

```
SELECT *
FROM CORPDATA.SYSCOLUMNS
WHERE TABLE_NAME = 'DEPARTMENT'
```

ผลของข้อความตัวอย่างก่อนหน้าคือแถวของข้อมูลสำหรับแต่ละคอลัมน์ในตาราง. ข้อมูลบางอย่างไม่สามารถมองเห็นได้ เพราะความกว้างของข้อมูลกว้างกว่าจอแสดงผล.

สำหรับข้อมูลเพิ่มเติมเกี่ยวกับแต่ละคอลัมน์, โปรดระบุข้อความเพื่อเลือกเช่น:

```
SELECT COLUMN_NAME, TABLE_NAME, DATA_TYPE, LENGTH, HAS_DEFAULT
FROM CORPDATA.SYSCOLUMNS
WHERE TABLE_NAME = 'DEPARTMENT'
```

นอกเหนือจากชื่อคอลัมน์ของแต่ละคอลัมน์, ข้อความเพื่อเลือกจะแสดง:

- ชื่อของตารางที่มีคอลัมน์อยู่
- ประเภทข้อมูลของคอลัมน์
- แอ็ททริบิวต์ความยาวของคอลัมน์
- หากคอลัมน์อนุญาตให้มีค่าดีฟอลต์

ผลลัพธ์จะมีลักษณะเช่นนี้:

COLUMN_NAME	TABLE_NAME	DATA_TYPE	LENGTH	HAS_DEFAULT
DEPTNO	DEPARTMENT	CHAR	3	N
DEPTNAME	DEPARTMENT	VARCHAR	29	N
MGRNO	DEPARTMENT	CHAR	6	Y
ADMRDEPT	DEPARTMENT	CHAR	3	N

การลบอ็อบเจ็กต์ฐานข้อมูล

คำสั่ง DROP จะลบอ็อบเจ็กต์ออก. ขึ้นอยู่กับ action ที่ร้องขอ, อ็อบเจ็กต์ใดๆ ที่ขึ้นอยู่กับอ็อบเจ็กต์นั้นทั้งทางตรงและทางอ้อม อาจถูกลบทิ้ง หรืออาจถูกป้องกันไม่ให้ถูกลบ.

ตัวอย่างเช่น, หากคุณลบตาราง, alias ใดๆ, ข้อจำกัด, ทรริกเกอร์, มุมมอง, หรือตรรกษที่เชื่อมโยงกับตารางนั้นจะถูกลบออกด้วย. เมื่อใดก็ตามที่อ็อบเจ็กต์ถูกลบทิ้ง, รายละเอียดของอ็อบเจ็กต์นั้นจะถูกลบออกจากแคตตาล็อก.

ตัวอย่างเช่น, หากต้องการลบตาราง EMPLOYEE, ให้ใช้คำสั่งต่อไปนี้:

```
DROP TABLE EMPLOYEE RESTRICT
```

ข้อมูลที่เกี่ยวข้อง

DROP statement

ภาษาสำหรับการดำเนินการข้อมูล

ภาษาสำหรับการดำเนินการข้อมูล (DML) คือส่วนของ SQL ที่อนุญาตให้คุณดำเนินการ หรือควบคุมข้อมูลของคุณได้.

หลักการที่เกี่ยวข้อง

“ประเภทของคำสั่ง SQL” ในหน้า 6

มีประเภทของคำสั่ง SQL พื้นฐานอยู่หลายประเภท. ซึ่งคำสั่งเหล่านั้นจะถูกแสดงไว้ที่นี่ตามฟังก์ชัน.

การดึงข้อมูลโดยใช้คำสั่ง SELECT

ศึกษาถึงวิธีต่างๆ ที่ทำให้เคียวรีของคุณเก็บข้อมูลโดยใช้คำสั่ง SELECT. วิธีหนึ่งในการทำเช่นนี้คือการใช้คำสั่ง SELECT ในโปรแกรมเพื่อดึงค่าเฉพาะ (ตัวอย่างเช่น, แถวข้อมูลพนักงาน). นอกจากนี้, คุณสามารถใช้ clause เพื่อรวบรวมข้อมูลในรูปแบบที่ต้องการได้.

ถ้า SQL ไม่สามารถหาแถวที่ตรงกับเงื่อนไขการค้นหา, SQLCODE ที่มีค่าเป็น +100 จะถูกคืนค่ากลับมา.

ถ้า SQL เจอข้อผิดพลาดขณะรันคำสั่ง Select, SQLCODE ที่มีค่าเป็นลบจะถูกคืนค่ากลับมา. ถ้า SQL เจอตัวแปรโฮสต์ที่มากกว่าผลลัพธ์, ค่า +326 จะถูกส่งคืนกลับมา.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การสร้างตารางโดยใช้ AS” ในหน้า 22

คำสั่ง CREATE TABLE AS จะสร้างตารางจากผลลัพธ์ของคำสั่ง SELECT.

“การสร้างและใช้มุมมอง” ในหน้า 35

สามารถใช้มุมมองเพื่อเข้าใช้งานข้อมูลในตารางหนึ่งตารางหรือมากกว่าหรือมุมมองหนึ่งมุมมองหรือมากกว่าได้. คุณสามารถสร้างมุมมองได้โดยใช้คำสั่ง SELECT.

ข้อความ SELECT ระดับต้น

รูปแบบและซินแทกซ์ดังแสดงในที่นี้จะเป็นอย่างง่าย. คำสั่ง SELECT อาจแตกต่างจากตัวอย่างที่แสดงในหัวข้อนี้.

คุณสามารถเขียนคำสั่ง SQL ได้หลายบรรทัด. สำหรับคำสั่ง SQL ในโปรแกรมพีคอมไฟล์, กฎสำหรับการต่อบรรทัดที่ใช้จะเป็นกฎเดียวกับของภาษาโฮสต์ (ภาษาที่ใช้เขียนโปรแกรม). นอกจากนี้คุณสามารถเรียกใช้คำสั่ง SELECT จากเคอร์เซอร์ในโปรแกรมได้. และสุดท้าย, คำสั่ง SELECT สามารถสร้างในแอ็พพลิเคชันแบบ dynamic ได้.

หมายเหตุ:

1. คำสั่ง SQL ที่อธิบายในส่วนนี้สามารถใช้งานได้กับตาราง วิว, และไฟล์ฐานข้อมูลทั้งแบบฟิสิกัลและโลจิคัล.
2. สตริงอักขระที่ถูกระบุในคำสั่ง SQL (เช่น ค่าที่ใช้ด้วย WHERE หรือ VALUES clause) จะตรงตามตัวอักษรพิมพ์ใหญ่หรือพิมพ์เล็ก; นั่นคือ, ตัวอักษรพิมพ์ใหญ่ต้องถูกป้อนในแบบตัวพิมพ์ใหญ่และตัวอักษรพิมพ์เล็กต้องถูกป้อนด้วยตัวพิมพ์เล็ก.

```
WHERE ADMRDEPT='a00'      (จะไม่คืนค่าผลลัพธ์)
```

```
WHERE ADMRDEPT='A00'      (จะคืนค่าหมายเลขแผนกที่ถูกต้อง)
```

การดำเนินการเปรียบเทียบอาจไม่เป็นแบบตรงตามตัวอักษรพิมพ์ใหญ่พิมพ์เล็กถ้าใช้การเรียงแบบ shared-weight ซึ่งจะถือว่าตัวอักษรพิมพ์ใหญ่และตัวอักษรพิมพ์เล็กเป็นตัวอักขระเดียวกัน.

คำสั่ง SELECT อาจประกอบด้วยสิ่งต่อไปนี้:

1. ชื่อของแต่ละคอลัมน์ที่คุณต้องการรวมในผลลัพธ์.
2. ชื่อของตารางหรือมุมมองที่มีข้อมูลอยู่.
3. เงื่อนไขการค้นหาที่ระบุแถวซึ่งมีข้อมูลที่คุณต้องการ.
4. ชื่อของแต่ละคอลัมน์ที่ใช้เพื่อจัดกลุ่มข้อมูลของคุณ.
5. เงื่อนไขการค้นหาที่ระบุกลุ่มที่มีข้อมูลที่คุณต้องการ.
6. ลำดับการเรียงผลลัพธ์เพื่อให้ส่งคืนแถวที่ต้องการซึ่งอยู่ท่ามกลางข้อมูลที่ซ้ำกัน.

คำสั่ง SELECT จะเป็นดังนี้:

```
SELECT ชื่อคอลัมน์
FROM table or view name
WHERE search condition
GROUP BY column names
HAVING search condition
ORDER BY column-name
```

SELECT และ FROM clause จะต้องถูกระบุ. clause อื่นจะเป็นตัวเลือกว่าจะระบุหรือไม่ก็ได้.

การใช้ SELECT clause, จะทำให้คุณระบุชื่อของแต่ละคอลัมน์ที่คุณต้องการดึงค่าได้. ตัวอย่างเช่น:

```
SELECT EMPNO, LASTNAME, WORKDEPT
```

คุณสามารถระบุให้ดึงข้อมูลแค่คอลัมน์เดียวเท่านั้น, หรือได้มากที่สุดถึง 8000 คอลัมน์. ค่าของแต่ละคอลัมน์ที่คุณเลือกจะถูกดึงข้อมูลในลำดับที่ระบุใน SELECT clause.

ถ้าคุณต้องการดึงค่าคอลัมน์ทั้งหมด (ในลำดับเดียวกับที่ปรากฏใน definition ของตาราง), ให้ใช้เครื่องหมายดอกจัน (*) แทนการใช้ชื่อคอลัมน์:

```
SELECT *
```

FROM clause จะระบุตารางที่คุณต้องการเลือกข้อมูลจาก. คุณสามารถเลือกคอลัมน์จากตารางมากกว่าหนึ่งตารางได้. เมื่อเรียกใช้คำสั่ง SELECT, คุณต้องระบุ FROM clause ด้วย. ใช้คำสั่งต่อไปนี้:

```
SELECT *  
FROM EMPLOYEE
```

ผลลัพธ์คือ คอลัมน์และแถวทั้งหมดจากตาราง EMPLOYEE.

รายการ SELECT อาจมีนิพจน์, ซึ่งรวมถึงค่าคงที่, เรจิสเตอร์พิเศษ, และ scalar fullselect. AS clause ยังสามารถใช้เพื่อตั้งชื่อคอลัมน์ผลลัพธ์. ตัวอย่างเช่น, ใช้คำสั่งต่อไปนี้:

```
SELECT LASTNAME, SALARY * .05 AS RAISE  
FROM EMPLOYEE  
WHERE EMPNO = '200140'
```

ผลลัพธ์ของคำสั่งนี้คือ:

ตารางที่ 7. ผลลัพธ์สำหรับการสืบค้น

LASTNAME	RAISE
NATZ	1421

การระบุเงื่อนไขการค้นหาโดยใช้ WHERE clause

WHERE clause จะระบุเงื่อนไขการค้นหาที่บ่งชี้ถึงแถวหรือแถวที่คุณต้องการดึงค่า, อัปเดต, หรือลบ.

จำนวนแถวที่คุณประมวลผลด้วยคำสั่ง SQL จะขึ้นอยู่กับจำนวนแถวที่ตรงกับเงื่อนไขการค้นหาของ WHERE clause. เงื่อนไขการค้นหาประกอบด้วยเพรดิเคตหนึ่งตัวขึ้นไป. เพรดิเคตจะระบุการทดสอบที่คุณต้องการให้ SQL ใช้กับแถวในตารางที่ระบุ.

ในตัวอย่างต่อไปนี้, WORKDEPT = 'C01' คือเพรดิเคต, WORKDEPT และ 'C01' คือนิพจน์, และเครื่องหมายเท่ากับ (=) คือ comparison operator. โปรดสังเกตว่าค่าตัวอักษรจะถูกล้อมด้วย apostrophe (''); ส่วนค่าตัวเลขจะไม่ถูกล้อมด้วยคำนี้. วิธีการนี้จะใช้กับค่าคงที่ทั้งหมดที่ถูกโค้ดภายในคำสั่ง SQL. ตัวอย่างเช่น, เมื่อต้องการระบุว่า คุณสนใจแถวที่มีหมายเลขแผนกคือ C01, ให้ใช้คำสั่งต่อไปนี้:

```
... WHERE WORKDEPT = 'C01'
```

ในกรณีนี้, เงื่อนไขการค้นหาประกอบด้วยเพรดิเคตหนึ่งตัว คือ: WORKDEPT = 'C01'.

เพื่อแสดงการใช้ WHERE ต่อไป, ให้พิจารณาเมื่อใช้กับคำสั่ง SELECT. สมมติว่าแต่ละแผนกที่อยู่ในตาราง CORPDATA. DEPARTMENT มีหมายเลขแผนกไม่ซ้ำกัน. คุณต้องการดึงค่าชื่อแผนกและหมายเลขผู้จัดการจากตาราง CORPDATA. DEPARTMENT สำหรับแผนก C01. ใช้คำสั่งต่อไปนี้:

```
SELECT DEPTNAME, MGRNO
      FROM CORPDATA.DEPARTMENT
      WHERE DEPTNO = 'C01'
```

เมื่อรันคำสั่งนี้, ผลลัพธ์คือหนึ่งแถว:

ตารางที่ 8. ตารางผลลัพธ์

DEPTNAME	MGRNO
INFORMATION CENTER	000030

ถ้าเงื่อนไขการค้นหามีอักขระ, หรือเพรดิเคตคอลัมน์ที่เป็นกราฟิกแบบ UCS-2 หรือ UTF-16, ลำดับการเรียงของผลลัพธ์ การทำเคียวรีจะส่งผลต่อค่าเพรดิเคตด้วย. ถ้าไม่มีการใช้ลำดับการเรียง, ค่าคงที่ที่เป็นตัวอักษรจะต้องระบุในแบบตัวพิมพ์ใหญ่หรือตัวพิมพ์เล็กเพื่อให้ตรงกับคอลัมน์หรือประโยคที่ค่าเหล่านั้นกำลังเปรียบเทียบอยู่.

หลักการที่เกี่ยวข้อง

“ลำดับการเรียง และ normalization ใน SQL” ในหน้า 119

ลำดับการจัดเรียงกำหนดความสัมพันธ์ของอักขระในชุดอักขระ เมื่อมีการเปรียบเทียบหรือจัดลำดับ. Normalization อนุญาตให้คุณเปรียบเทียบสตริงที่มีอักขระแบบผสม.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การนิยามเงื่อนไขการค้นหาที่ซับซ้อน” ในหน้า 59

นอกจากเพรดิเคตการเปรียบเทียบพื้นฐานแล้ว (=, >, <, และอื่นๆ), เงื่อนไขการค้นหาสามารถมีเพรดิเคต BETWEEN, IN, EXISTS, IS NULL, และ LIKE.

“หลายเงื่อนไขการค้นหาภายใน WHERE clause” ในหน้า 62

คุณสามารถทำการร้องขอเพิ่มเติม โดยโค้ดเงื่อนไขการค้นหาที่ประกอบด้วยหลายเพรดิเคต.

นิพจน์ใน WHERE clause:

นิพจน์ใน WHERE clause ใช้เพื่อระบุสิ่งที่คุณต้องการเปรียบเทียบกับสิ่งอื่น.

นิพจน์ที่คุณระบุสามารถเป็น:

- ชื่อคอลัมน์คือการระบุคอลัมน์. ตัวอย่างเช่น:

```
... WHERE EMPNO = '000200'
```

EMPNO ระบุคอลัมน์ที่ถูกระบุด้วยค่าอักขระชนิด 6-ไบต์.

- นิพจน์ ระบุค่าสองค่าที่จะถูกเพิ่ม (+), ลบ (-), คูณ (*), หาร (/), ยกกำลัง (**), หรือเชื่อมต่อ (CONCAT หรือ ||) กับผลลัพธ์ในค่า. operand พื้นฐานของนิพจน์คือ:

- ค่าคงที่
- คอลัมน์
- ตัวแปรโฮสต์
- ฟังก์ชัน
- register พิเศษ
- scalar fullselect

- นิพจน์อื่น

ตัวอย่างเช่น:

```
... WHERE INTEGER(PRENDATE - PRSTDATE) > 100
```

เมื่อลำดับของการประเมินผลไม่ได้ระบุโดยเครื่องหมายวงเล็บแล้ว, นิพจน์จะถูกประเมินผลในลำดับดังต่อไปนี้:

1. โอเปอเรเตอร์ prefix
2. การยกกำลัง
3. การคูณ, การหาร, และการเชื่อมต่อ
4. การบวกและการลบ

โอเปอเรเตอร์ที่อยู่ระดับเดียวกันจะทำงานจากซ้ายมาขวา.

- ค่าคงที่จะระบุค่า literal สำหรับนิพจน์. ตัวอย่างเช่น:

```
... WHERE 40000 < SALARY
```

SALARY ระบุคอลัมน์ที่ถูกระบุเป็นค่าทศนิยมแบบ 9 หลัก (DECIMAL(9,2)). และจะถูกเปรียบเทียบกับค่าคงที่ 40000.

- ตัวแปรโฮสต์ระบุตัวแปรในแอ็พพลิเคชันโปรแกรม. ตัวอย่างเช่น:

```
... WHERE EMPNO = :EMP
```

- register พิเศษระบุค่าพิเศษที่นิยามโดยผู้จัดการฐานข้อมูล. ตัวอย่างเช่น:

```
... WHERE LASTNAME = USER
```

- ค่า NULL ระบุค่าที่ไม่ทราบค่า.

```
... WHERE DUE_DATE IS NULL
```

- scalar fullselect.

เงื่อนไขการค้นหาสามารถระบุเพรดิเคตหลายตัวที่แยกโดย AND และ OR. โดยไม่ว่าเงื่อนไขการค้นหาค่าจะซับซ้อนอย่างไร, มันจะให้ค่า TRUE หรือ FALSE เมื่อทำการประเมินผลกับแถว. และยังมีค่าความจริงเป็น *Unknown*, ซึ่งมีผลเป็น False. นั่นคือ, ถ้าค่าของแถวเป็น null แล้ว, ค่า null นี้จะไม่ถูกส่งคืนค่าเป็นผลลัพธ์ของการค้นหาเพราะว่าค่านี้ไม่ใช่ค่าที่น้อยกว่า, เท่ากับ, หรือมากกว่าค่าที่ระบุในเงื่อนไขการค้นหา.

เพื่อให้เข้าใจการใช้ WHERE clause, คุณจำเป็นต้องรู้ลำดับที่ SQL ทำการประเมินผลเงื่อนไขการค้นหาและเพรดิเคต, และวิธีที่ SQL เปรียบเทียบค่าของนิพจน์. หัวข้อนี้จะอธิบายไว้ในหัวข้อ การอ้างอิง SQL.

หลักการที่เกี่ยวข้อง

“การใช้เคียวรี้อยู่” ในหน้า 110

คุณสามารถใช้เคียวรี้อยู่ในเงื่อนไขการค้นหาเพื่อเป็นอีกทางหนึ่งในการเลือกข้อมูล. เคียวรี้อยู่สามารถใช้ได้ทุกที่ที่นิพจน์สามารถใช้งานได้.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การนิยามเงื่อนไขการค้นหาที่ซับซ้อน” ในหน้า 59

นอกจากเพรดิเคตการเปรียบเทียบพื้นฐานแล้ว (=, >, <, และอื่นๆ), เงื่อนไขการค้นหาสามารถมีเพรดิเคต BETWEEN, IN, EXISTS, IS NULL, และ LIKE.

นิพจน์

ตัวดำเนินการเปรียบเทียบ:

SQL สนับสนุน comparison operator เหล่านี้.

=	Equal to
<> or \neq or !=	Not equal to
<	Less than
>	Greater than
<= or \geq or !>	Less than or equal to (or not greater than)
>= or \leq or !<	Greater than or equal to (or not less than)

NOT keyword:

คุณสามารถเพิ่มหน้าเพรดิเคตด้วยคีย์เวิร์ด NOT เพื่อระบุว่า คุณต้องการค่าตรงกันข้ามกับค่าของเพรดิเคต (นั่นคือ, TRUE ถ้าเพรดิเคตคือ FALSE, หรือในทางกลับกัน).

NOT จะใช้กับเพรดิเคตที่อยู่ต่อจากมันเท่านั้น, ไม่ได้ใช้กับเพรดิเคตทั้งหมดใน WHERE clause. ตัวอย่างเช่น, เมื่อต้องการระบุว่าคุณสนใจในพนักงานทั้งหมดยกเว้นพนักงานที่ทำงานในแผนก C01, คุณอาจจะใช้คำสั่ง:

```
... WHERE NOT WORKDEPT = 'C01'
```

ซึ่งจะเท่ากับ:

```
... WHERE WORKDEPT <> 'C01'
```

GROUP BY clause

GROUP BY clause อนุญาตให้คุณค้นหาคุณลักษณะของกลุ่มของแถวมากกว่าที่จะค้นหาแถวเดียว.

เมื่อคุณระบุ GROUP BY clause แล้ว, SQL จะแบ่งแถวที่เลือกให้เป็นกลุ่ม ซึ่งแถวของแต่ละกลุ่มจะมีค่าที่ตรงกับค่าในคอลัมน์หรือนิพจน์. ต่อมา, SQL จะดำเนินการกับแต่ละกลุ่มเพื่อสร้างผลลัพธ์แบบแถวเดียวให้กลุ่ม. คุณสามารถระบุคอลัมน์หรือนิพจน์ได้มากกว่าหนึ่งใน GROUP BY clause เพื่อจัดกลุ่มแถว. รายการที่คุณระบุในคำสั่ง SELECT จะเป็นคุณสมบัติของแต่ละกลุ่มของแถว, ไม่ใช่คุณสมบัติของแถวเดี่ยวในตารางหรือมุมมอง.

เมื่อไม่มี GROUP BY clause, แอ็พพลิเคชันของฟังก์ชันโดยรวมของ SQL จะคืนค่าหนึ่งแถว. เมื่อ GROUP BY ถูกใช้, ฟังก์ชันจะถูกใช้กับแต่ละกลุ่ม, ดังนั้นจะคืนค่าแถวมากเท่ากับจำนวนกลุ่มที่มีอยู่.

ตัวอย่างเช่น, ตาราง CORPDATA.EMPLOYEE มีแถวอยู่หลายชุด, และแต่ละชุดจะมีแถวที่อธิบายข้อมูลสมาชิกของแต่ละแผนก. เมื่อต้องการหาค่าเฉลี่ยเงินเดือนของพนักงานในแต่ละแผนก, คุณสามารถใช้คำสั่ง:

```
SELECT WORKDEPT, DECIMAL (AVG(SALARY),5,0)
      FROM CORPDATA.EMPLOYEE
      GROUP BY WORKDEPT
```

ผลลัพธ์จะมีหลายแถว, หนึ่งแถวสำหรับแต่ละแผนก.

WORKDEPT	AVG-SALARY
A00	40850
B01	41250

WORKDEPT	AVG-SALARY
C01	29722
D11	25147
D21	25668
E01	40175
E11	21020
E21	24086

หมายเหตุ:

1. การจัดกลุ่มแถวไม่ได้หมายความว่าเรียงลำดับแถวเหล่านั้นด้วย. การจัดกลุ่มจะดึงแถวที่เลือกเข้ามา, เพื่อให้ SQL จัดการประมวลผลหาค่าความเป็นลักษณะเฉพาะออกมา. การเรียงลำดับแถวจะใส่แถวทั้งหมดเข้าไปในตารางผลลัพธ์ด้วยลำดับการเรียงแบบจากน้อยไปมากหรือจากมากไปน้อย. โดยขึ้นอยู่กับวิธีปฏิบัติที่ผู้จัดการฐานข้อมูลเลือก, กลุ่มของผลลัพธ์อาจจะเรียงลำดับก็ได้.
2. ถ้ามีค่า null ในคอลัมน์ที่คุณระบุใน GROUP BY clause, ผลลัพธ์แบบแถวเดียวจะถูกสร้างเป็นข้อมูลในแถวด้วยค่า null.
3. ถ้ามีการจัดกลุ่มกับอักขระ, หรือคอลัมน์กราฟิกประเภท USC-2 หรือ UTF-16, การเรียงลำดับที่เกิดจากการรันเดียวจะมีผลต่อการทำกลุ่มด้วย.

เมื่อคุณใช้ GROUP BY, ระบบจะแสดงรายชื่อคอลัมน์หรือนิพจน์ที่คุณต้องการให้ SQL ใช้ในการจัดกลุ่มแถว. ตัวอย่างเช่น, สมมติว่าคุณต้องการรายการจำนวนคนที่ทำงานในแต่ละโครงการสำคัญซึ่งอธิบายในตาราง CORPDATA.PROJECT. คุณสามารถใช้คำสั่ง:

```
SELECT SUM(PRSTAFF), MAJPROJ
       FROM CORPDATA.PROJECT
       GROUP BY MAJPROJ
```

ผลลัพธ์คือรายการโครงการสำคัญปัจจุบันของบริษัท และจำนวนคนที่ทำงานในแต่ละโครงการ:

SUM(PRSTAFF)	MAJPROJ
6	AD3100
5	AD3110
10	MA2100
8	MA2110
5	OP1000
4	OP2000
3	OP2010
32.5	?

คุณยังสามารถระบุว่าคุณต้องการแถวที่จัดกลุ่มโดยคอลัมน์หรือนิพจน์มากกว่าหนึ่งค่าได้. ตัวอย่างเช่น, คุณอาจใช้คำสั่ง Select เพื่อหาค่าเฉลี่ยเงินเดือนสำหรับผู้ชายและผู้หญิงในแต่ละแผนก, โดยใช้ตาราง CORPDATA.EMPLOYEE. เมื่อต้องการทำเช่นนี้, ให้คุณระบุ:

```
SELECT WORKDEPT, SEX, DECIMAL(AVG(SALARY),5,0) AS AVG_WAGES
FROM CORPDATA.EMPLOYEE
GROUP BY WORKDEPT, SEX
```

ผลลัพธ์จะเป็น:

WORKDEPT	SEX	AVG_WAGES
A00	F	49625
A00	M	35000
B01	M	41250
C01	F	29722
D11	F	25817
D11	M	24764
D21	F	26933
D21	M	24720
E01	M	40175
E11	F	22810
E11	M	16545
E21	F	25370
E21	M	23830

เนื่องจากคุณไม่ได้รวม WHERE clause เข้าไปในตัวอย่างนี้, SQL จึงตรวจสอบและดำเนินการกับทุกแถวในตาราง CORPDATA.EMPLOYEE. แถวจะถูกจัดกลุ่มโดยหมายเลขแผนกก่อนแล้วต่อด้วยเพศ (ภายในแต่ละแผนก) ก่อนที่ SQL จะหาค่า SALARY เฉลี่ยสำหรับแต่ละกลุ่ม.

หลักการที่เกี่ยวข้อง

“ลำดับการเรียง และ normalization ใน SQL” ในหน้า 119

ลำดับการจัดเรียงกำหนดความสัมพันธ์ของอักขระในชุดอักขระ เมื่อมีการเปรียบเทียบหรือจัดลำดับ. Normalization อนุญาตให้คุณเปรียบเทียบสตริงที่มีอักขระแบบผสม.

สิ่งอ้างอิงที่เกี่ยวข้อง

“ORDER BY clause” ในหน้า 51

ORDER BY ระบุลำดับเฉพาะที่คุณต้องการส่งคืนแถวที่เลือกไว้. ลำดับจะถูกเรียงลำดับการเรียงของค่าคอลัมน์หรือนิพจน์จากน้อยไปมาก หรือจากมากไปน้อย.

HAVING clause

HAVING clause ระบุเงื่อนไขการค้นหาสำหรับกลุ่มที่เลือกได้โดยใช้ GROUP BY clause.

HAVING clause เป็นการบอกว่าคุณต้องการเฉพาะกลุ่มที่เป็นไปตามเงื่อนไขใน clause นั้น. ดังนั้นแล้ว, เงื่อนไขการค้นหาที่คุณระบุใน HAVING clause จะต้องทดสอบคุณสมบัติของแต่ละกลุ่มมากกว่าที่จะทดสอบคุณสมบัติของแถวเดี่ยวในกลุ่ม.

HAVING clause จะอยู่ต่อจาก GROUP BY clause และสามารถประกอบด้วยประเภทของเงื่อนไขการค้นหาเดียวกับที่คุณสามารถระบุใน WHERE clause. นอกเหนือจากนั้น, คุณสามารถระบุฟังก์ชันการรวมใน HAVING clause. ตัวอย่างเช่น, สมมติว่าคุณต้องการดึงค่าเงินเดือนเฉลี่ยของผู้หญิงในแต่ละแผนก. หากต้องการทำสิ่งนี้, ให้ใช้ฟังก์ชันการรวม AVG และจัดกลุ่มแถวที่ได้ด้วย WORKDEPT และระบุ WHERE clause โดยที่กำหนด SEX = 'F'.

เมื่อต้องการระบุว่าคุณต้องการข้อมูลนี้เฉพาะเมื่อพนักงานผู้หญิงทั้งหมดในแผนกที่ถูกเลือกมีระดับการศึกษาเท่ากับหรือมากกว่า 16 (การศึกษาระดับวิทยาลัย), ให้ใช้ HAVING clause. HAVING clause จะทดสอบคุณสมบัติของกลุ่ม. ในกรณีนี้, การทดสอบคือ MIN(EDLEVEL), ซึ่งก็คือคุณสมบัติของกลุ่ม:

```
SELECT WORKDEPT, DECIMAL(AVG(SALARY),5,0) AS AVG_WAGES, MIN(EDLEVEL) AS MIN_EDUC
FROM CORPDATA.EMPLOYEE
WHERE SEX='F'
GROUP BY WORKDEPT
HAVING MIN(EDLEVEL)>=16
```

ผลลัพธ์จะเป็น:

WORKDEPT	AVG_WAGES	MIN_EDUC
A00	49625	18
C01	29722	16
D11	25817	17

คุณสามารถใช้เพรดิเคตหลายตัวใน HAVING clause ได้โดยเชื่อมโยงค่าเหล่านั้นด้วย AND และ OR, และคุณสามารถใช้ NOT สำหรับเพรดิเคตใดๆ ของเงื่อนไขการค้นหา.

หมายเหตุ: ถ้าคุณตั้งใจอัปเดตคอลัมน์หรือลบแถว, คุณไม่สามารถรวม GROUP BY clause หรือ HAVING clause เข้าไปในคำสั่ง SELECT ภายในคำสั่ง DECLARE CURSOR ได้. clause เหล่านี้ทำให้เคอร์เซอร์เป็นแบบอ่านได้อย่างเดียว.

เพรดิเคตที่มีอากิวเมนต์ที่ไม่ใช่ฟังก์ชันรวมสามารถโค้ดใน WHERE หรือ HAVING clause. ปกติการโค้ดเงื่อนไขการเลือกไว้ใน WHERE clause จะทำให้การค้นหามีประสิทธิภาพมากขึ้นเนื่องจากเงื่อนไขจะถูกจัดการก่อนในกระบวนการเดียว. การเลือก HAVING จะถูกทำในขั้นตอนหลังประมวลผลตารางผลลัพธ์.

ถ้าเงื่อนไขการค้นหาใช้อักขระ, หรือเพรดิเคตคอลัมน์ที่เป็นกราฟิกแบบ UCS-2 หรือ UTF-16, ลำดับการเรียงของผลลัพธ์การทำเคียวรีจะส่งผลต่อค่าเพรดิเคตด้วย.

หลักการที่เกี่ยวข้อง

“ลำดับการเรียง และ normalization ใน SQL” ในหน้า 119

ลำดับการจัดเรียงกำหนดความสัมพันธ์ของอักขระในชุดอักขระ เมื่อมีการเปรียบเทียบหรือจัดลำดับ. Normalization อนุญาตให้คุณเปรียบเทียบสตริงที่มีอักขระแบบผสม.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การใช้เคอร์เซอร์” ในหน้า 256

เมื่อ SQL รันข้อความที่เลือก, แถวผลลัพธ์จะประกอบขึ้นจากตารางผลลัพธ์. เคอร์เซอร์จะแสดงวิธีการเข้าถึงตารางผลลัพธ์.

ORDER BY clause

ORDER BY ระบุลำดับเฉพาะที่คุณต้องการส่งคืนแถวที่เลือกไว้. ลำดับจะถูกเรียงลำดับการเรียงของค่าคอลัมน์ หรือค่านิพจน์จากน้อยไปมาก หรือจากมากไปน้อย.

ตัวอย่างเช่น, เมื่อต้องการดึงชื่อและหมายเลขแผนกของพนักงานหญิงโดยให้หมายเลขแผนกเรียงตามลำดับตัวอักษร, คุณควรใช้คำสั่ง Select ดังนี้:

```
SELECT LASTNAME,WORKDEPT
      FROM CORPDATA.EMPLOYEE
      WHERE SEX='F'
      ORDER BY WORKDEPT
```

ผลลัพธ์จะเป็น:

LASTNAME	WORKDEPT
HAAS	A00
HEMMINGER	A00
KWAN	C01
QUINTANA	C01
NICHOLLS	C01
NATZ	C01
PIANKA	D11
SCOUTTEN	D11
LUTZ	D11
JOHN	D11
PULASKI	D21
JOHNSON	D21
PEREZ	D21
HENDERSON	E11
SCHNEIDER	E11

LASTNAME	WORKDEPT
SETRIGHT	D11
SCHWARTZ	E11
SPRINGER	E11
WONG	E21

หมายเหตุ: ค่า null จะถูกเรียงลำดับเป็นค่าที่สูงที่สุด.

คอลัมน์ที่ถูกระบุใน ORDER BY clause ไม่จำเป็นต้องถูกรวมเข้าไปใน SELECT clause. ตัวอย่างเช่น, คำสั่งต่อไปนี้จะคืนค่าพนักงานหญิงทั้งหมด ซึ่งเรียงลำดับด้วยเงินเดือนที่มากที่สุดตามลำดับ:

```
SELECT LASTNAME, FIRSTNAME FROM CORPDATA.EMPLOYEE
WHERE SEX='F'
ORDER BY SALARY DESC
```

ถ้า AS clause ถูกระบุเพื่อตั้งชื่อคอลัมน์ผลลัพธ์ในรายการที่เลือก, ชื่อนี้สามารถถูกระบุใน ORDER BY clause ได้. ชื่อที่ระบุใน AS clause จะต้องไม่ซ้ำกันในรายการที่เลือก. ตัวอย่าง, ถ้าต้องการตั้งชื่อเต็มของพนักงานให้เรียงลำดับตามตัวอักษร, คุณสามารถใช้คำสั่ง Select ต่อไปนี้:

```
SELECT LASTNAME CONCAT FIRSTNAME AS FULLNAME FROM CORPDATA.EMPLOYEE
ORDER BY FULLNAME
```

คำสั่ง select นี้อาจเขียนได้เป็น:

```
SELECT LASTNAME CONCAT FIRSTNAME
FROM CORPDATA.EMPLOYEE
ORDER BY LASTNAME CONCAT FIRSTNAME
```

แทนที่จะตั้งชื่อคอลัมน์เพื่อเรียงลำดับผลลัพธ์, คุณสามารถใช้หมายเลขได้. ตัวอย่างเช่น, ORDER BY 3 จะระบุว่า คุณต้องการผลลัพธ์ที่ถูกเรียงลำดับโดยคอลัมน์ที่สามของตารางผลลัพธ์, ที่ระบุโดยรายการที่เลือก. ใช้ตัวเลขเพื่อเรียงลำดับแถวของตารางผลลัพธ์เมื่อค่าที่เรียงเป็นคอลัมน์ที่ไม่มีชื่อ.

คุณยังสามารถระบุว่าคุณต้องการให้ SQL เรียงลำดับแถวในลำดับจากน้อยไปมาก (ASC) หรือจากมากไปน้อย (DESC). การเรียงลำดับจากน้อยไปมากจะเป็นค่าดีฟอลต์. ในคำสั่ง Select ก่อนหน้านี้, SQL จะคืนค่าแถวด้วยนิพจน์ *FULLNAME* ที่ต่ำที่สุดก่อน (ตามตัวอักษรและตัวเลข), ตามด้วยแถวที่มีค่ามากกว่า. เมื่อต้องการเรียงลำดับแถวในลำดับการเรียงจากมากไปน้อยโดยยึดจากชื่อนี้, ให้ระบุ:

```
... ORDER BY FULLNAME DESC
```

คุณสามารถระบุลำดับการเรียงสำรอง (หรือลำดับการเรียงอีกหลายระดับ) และลำดับการเรียงแรกได้. ในตัวอย่างก่อนหน้านี้, คุณอาจต้องการให้แถวเรียงลำดับโดยหมายเลขแผนกก่อน, และภายในแต่ละแผนก, ให้เรียงลำดับโดยชื่อพนักงาน. เมื่อต้องการทำเช่นนั้น, ให้ระบุ:

```
... ORDER BY WORKDEPT, FULLNAME
```

ถ้าคอลัมน์แบบตัวอักษร, หรือคอลัมน์แบบกราฟิก UCS-2 ถูกใช้ใน ORDER BY แล้ว, การเรียงลำดับของคอลัมน์เหล่านี้จะขึ้นกับการจัดลำดับที่กำหนดในการรันเคียวรี.

หลักการที่เกี่ยวข้อง

“ลำดับการเรียง และ normalization ใน SQL” ในหน้า 119

ลำดับการจัดเรียงกำหนดความสัมพันธ์ของอักขระในชุดอักขระ เมื่อมีการเปรียบเทียบหรือจัดลำดับ. Normalization อนุญาตให้คุณเปรียบเทียบสตริงที่มีอักขระแบบผสม.

สิ่งอ้างอิงที่เกี่ยวข้อง

“GROUP BY clause” ในหน้า 47

GROUP BY clause อนุญาตให้คุณค้นหาคุณลักษณะของกลุ่มของแถวมากกว่าที่จะค้นหาแถวเดียว.

คำสั่ง SELECT แบบ Static

สำหรับคำสั่ง SELECT แบบ static (ซึ่งฝังตัวอยู่ในโปรแกรม SQL), INTO clause จะต้องระบุไว้หน้า FROM clause.

INTO clause จะตั้งชื่อตัวแปรโฮสต์ (ตัวแปรในโปรแกรมของคุณซึ่งถูกใช้เพื่อเก็บค่าคอลัมน์ที่ดึงค่ามา). ค่าของคอลัมน์ผลลัพธ์แรกที่ระบุใน SELECT clause จะถูกใส่ค่าเข้าไปในตัวแปรโฮสต์ตัวแรกที่มีชื่อในค INTO clause; ค่าที่สองจะถูกใส่ค่าเข้าไปในตัวแปรโฮสต์ตัวที่สอง, เช่นนี้ไปเรื่อยๆ.

ตารางผลลัพธ์สำหรับ SELECT INTO ควรจะมีแค่หนึ่งแถวเท่านั้น. ตัวอย่างเช่น, แต่ละแถวในตาราง CORPDATA. EMPLOYEE จะมีคอลัมน์ EMPNO (หมายเลขพนักงาน) ที่ไม่ซ้ำกัน. ผลลัพธ์ของคำสั่ง SELECT INTO สำหรับตารางนี้ถ้า WHERE clause มีการเปรียบเทียบแบบเท่ากับกับคอลัมน์ EMPNO, ควรจะมีแถวเดียวเท่านั้น (หรือไม่มีแถวเลย). หากผลการค้นหาว่ามีมากกว่าหนึ่งแถวแสดงว่ามีข้อผิดพลาด, แต่ว่าจะมีแถวหนึ่งที่ถูกคืนค่ามา. คุณสามารถควบคุมว่าแถวไหนที่จะถูกคืนค่ามาในสถานะที่ผิดพลาดเช่นนี้โดยการใช้ ORDER BY clause. ถ้าคุณใช้ ORDER BY clause, แถวแรกในตารางผลลัพธ์จะถูกคืนค่ากลับมา.

ถ้าคุณต้องการผลลัพธ์ของคำสั่ง SELECT INTO มากกว่าหนึ่งแถว, ให้ใช้คำสั่ง DECLARE CURSOR เพื่อเลือกแถว, แล้วจึงตามด้วยคำสั่ง FETCH เพื่อย้ายค่าคอลัมน์ไปไว้ในตัวแปรโฮสต์ครั้งละหนึ่งแถวขึ้นไป.

เมื่อใช้คำสั่ง Select ในแอปพลิเคชันโปรแกรม, ให้ทำรายชื่อคอลัมน์เพื่อทำให้โปรแกรมของคุณมีข้อมูลที่เป็นอิสระมากขึ้น. การทำเช่นนี้มีเหตุผล 2 ประการด้วยกัน:

1. เมื่อคุณดูคำสั่งในซอร์สโค้ด, คุณจะเห็นชื่อคอลัมน์ใน SELECT clause และตัวแปรโฮสต์ที่มีชื่อใน INTO clause จะตรงกันแบบรายการต่อรายการ.
2. ถ้าคอลัมน์ถูกเพิ่มเข้าไปในตารางหรือมุมมองที่คุณเข้าถึง และคุณใช้ “SELECT * ...,” และคุณสร้างโปรแกรมอีกครั้งจากต้นฉบับนี้, INTO clause จะไม่มีชื่อตัวแปรโฮสต์ที่ตรงกับคอลัมน์ใหม่. คอลัมน์พิเศษนี้จะเป็เหตุให้คุณได้รับการเตือน (ไม่ใช่ข้อผิดพลาด) ใน SQLCA (SQLWARN3 จะมีค่า “W”). เมื่อใช้คำสั่ง GET DIAGNOSTICS, RETURNED_SQLSTATE จะมีค่าเท่ากับ '01503'.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การใช้เคอร์เซอร์” ในหน้า 256

เมื่อ SQL รันข้อความที่เลือก, แถวผลลัพธ์จะประกอบขึ้นจากตารางผลลัพธ์. เคอร์เซอร์จะแสดงวิธีการเข้าถึงตารางผลลัพธ์.

การจัดการค่า null

ค่า NULL จะระบุว่าไม่มีค่าคอลัมน์ในแถว.

ค่า null ไม่ใช่ค่าเดียวกับค่าศูนย์หรือค่าว่างเปล่า. null หมายถึงไม่ทราบค่า. เราสามารถใช้ค่า null เป็นเงื่อนไขใน WHERE และ HAVING clause ได้. ตัวอย่างเช่น, WHERE clause สามารถระบุคอลัมน์ที่, เก็บค่า null อยู่, ในบางแถว. การเปรียบเทียบพบดิเคตขึ้นพื้นฐานโดยใช้คอลัมน์ที่เก็บค่า null จะไม่เลือกแถวที่คอลัมน์มีค่า null. ที่เป็นเช่นนั้นเนื่องจากค่า null จะไม่น้อยกว่า, เท่ากับ, หรือมากกว่าค่าที่ระบุในเงื่อนไข. เพรดิเคต IS NULL จะถูกใช้เพื่อตรวจสอบค่า null. เมื่อต้องการเลือกค่าสำหรับทุกแถวที่มีค่าหมายเลขผู้จัดการเป็น null, คุณอาจจะระบุ:

```
SELECT DEPTNO, DEPTNAME, ADMRDEPT
FROM CORPDATA.DEPARTMENT
WHERE MGRNO IS NULL
```

ผลที่ได้คือ:

DEPTNO	DEPTNAME	ADMRDEPT
D01	DEVELOPMENT CENTER	A00
F22	BRANCH OFFICE F2	E01
G22	BRANCH OFFICE G2	E01
H22	BRANCH OFFICE H2	E01
I22	BRANCH OFFICE I2	E01
J22	BRANCH OFFICE J2	E01

เมื่อต้องการหาแถวที่หมายเลขของผู้จัดการไม่เป็น null, คุณควรเปลี่ยน WHERE clause ให้เป็นดังนี้:

```
WHERE MGRNO IS NOT NULL
```

DISTINCT เป็นเพรดิเคตที่มีประโยชน์สามารถในการเปรียบเทียบค่าที่เป็น null ได้. การเปรียบเทียบสองคอลัมน์ด้วยเครื่องหมายเท่ากับ (COL1 = COL2) จะให้ค่าเป็น true ถ้าทั้งสองคอลัมน์มีค่าเท่ากันและไม่เป็น null. แต่ถ้าทั้งสองคอลัมน์มีค่าเป็น null, ผลที่ได้จะเป็น false เพราะค่า null จะไม่เท่ากับค่าใดทั้งนั้น, แม้กระทั่ง null ด้วยกัน. สำหรับเพรดิเคต DISTINCT, ค่า null จะถือว่าเท่ากัน. ดังนั้นประโยค (COL1 is NOT DISTINCT from COL2) จะให้ผลเป็น true เมื่อทั้งสองคอลัมน์มีค่าเท่ากันทั้งในกรณีที่ทั้งคู่ไม่ใช่ null หรือมีค่าเป็น null.

ตัวอย่างเช่น, สมมติว่า คุณต้องการเลือกข้อมูลจาก 2 ตารางที่มีค่า null. ตารางแรก (T1) มีคอลัมน์ (C1) ที่มีค่าดังนี้:

C1
2
1
null

ตารางที่สอง (T2) มีคอลัมน์ (C2) ที่มีค่าดังนี้:

C2
2

C2

null

รันคำสั่ง SELECT ต่อไปนี้:

```
SELECT *  
  FROM T1, T2  
  WHERE C1 IS DISTINCT FROM C2
```

ผลที่ได้คือ:

C1	C2
1	2
1	-
2	-
-	2

สำหรับข้อมูลเพิ่มเติมเกี่ยวกับการใช้ค่า null, โปรดดูหัวข้อ การอ้างอิง SQL.

เรจิสเตอร์พิเศษในคำสั่ง SQL

คุณสามารถระบุเรจิสเตอร์พิเศษในคำสั่ง SQL ได้.

สำหรับคำสั่ง SQL ที่รันแบบ *โลคัล*, เรจิสเตอร์พิเศษ และเนื้อหาจะแสดงอยู่ในตารางต่อไปนี้:

เรจิสเตอร์พิเศษ	เนื้อหา
CURRENT DATE CURRENT_DATE	วันที่ปัจจุบัน.
I CURRENT DEGREE	จำนวนของงานที่ตัวจัดการฐานข้อมูลควรรันแบบขนาน.
CURRENT PATH CURRENT_PATH CURRENT FUNCTION PATH	พาท SQL ที่ใช้ resolve ชื่อของชนิดข้อมูล, ชื่อโพรซีเจอร์, และชื่อฟังก์ชันที่ไม่ครบตามเกณฑ์ในคำสั่ง SQL ที่เตรียมไว้แบบ dynamic.
CURRENT SCHEMA	ชื่อแบบแผนที่ใช้เพื่อทำให้อ้างอิงอ็อบเจ็กต์ฐานข้อมูลครบตามเกณฑ์ซึ่งสามารถใช้ได้ในคำสั่ง SQL ที่เตรียมไว้แบบ dynamic.
CURRENT SERVER CURRENT_SERVER	ชื่อของฐานข้อมูลเชิงสัมพันธ์ที่ถูกใช้ขณะนี้.

เรจิสเตอร์พิเศษ	เนื้อหา
CURRENT TIME CURRENT_TIME	เวลาปัจจุบัน.
CURRENT TIMESTAMP CURRENT_TIMESTAMP	วันที่และเวลาปัจจุบันที่อยู่ในรูปของ timestamp.
CURRENT TIMEZONE CURRENT_TIMEZONE	ช่วงระยะเวลาที่เชื่อมโยงเวลาที่ท้องถิ่นเข้ากับ Universal Time Coordinated (UTC) โดยใช้สูตร: เวลาท้องถิ่น - CURRENT TIMEZONE = UTC ซึ่งจะถูกนำมาจากค่ากำหนดของระบบ QUTCOFFSET.
SESSION_USER USER	ตัวบ่งชี้การให้สิทธิ์ (โปรไฟล์ผู้ใช้) ขณะรันไทม์ของงาน.
SYSTEM_USER	ตัวบ่งชี้การให้สิทธิ์ (โปรไฟล์ผู้ใช้) ของผู้ใช้ที่เชื่อมต่อกับฐานข้อมูล.

ถ้าคำสั่งเดียวมีการอ้างอิงไปยังเรจิสเตอร์พิเศษ CURRENT DATE, CURRENT TIME, หรือ CURRENT TIMESTAMP, หรือฟังก์ชันแบบสเกลาร์ CURDATE, CURTIME, หรือ NOW มากกว่าหนึ่งการอ้างอิงแล้ว, คำทั้งหมดจะอยู่บนพื้นฐานของการอ่านข้อมูลจากนาฬิกาครั้งเดียว.

สำหรับคำสั่ง SQL ที่ทำงานแบบรีโมต, register พิเศษและเนื้อหาจะแสดงในตารางต่อไปนี้:

เรจิสเตอร์พิเศษ	เนื้อหา
CURRENT DATE CURRENT_DATE CURRENT TIME CURRENT_TIME CURRENT TIMESTAMP CURRENT_TIMESTAMP	วันที่และเวลาปัจจุบันของระบบรีโมต, ไม่ใช่ระบบโลคัล.
CURRENT DEGREE 	จำนวนของงานที่ตัวจัดการฐานข้อมูลควรรันแบบขนานบนระบบรีโมต.
CURRENT TIMEZONE CURRENT_TIMEZONE	ระยะเวลาที่เชื่อมโยงระบบรีโมตเข้ากับ UTC.
CURRENT SERVER CURRENT_SERVER	ชื่อของฐานข้อมูลเชิงสัมพันธ์ที่ถูกใช้ขณะนี้.
CURRENT SCHEMA	ค่าแบบแผนปัจจุบันที่ระบบรีโมต.

เรจิสเตอร์พิเศษ	เนื้อหา
CURRENT_PATH CURRENT_PATH CURRENT FUNCTION PATH	ค่าพารามิเตอร์ที่ระบบรีโมต.
SESSION_USER USER	ตัวบ่งชี้การให้สิทธิ์ (โปรไฟล์ผู้ใช้) ของงานบนระบบรีโมต.
SYSTEM_USER	ตัวบ่งชี้การให้สิทธิ์ (โปรไฟล์ผู้ใช้) ของผู้ใช้ที่เชื่อมต่อกับฐานข้อมูลบนระบบรีโมต.

เมื่อทำการสืบค้นกับตารางแบบกระจายที่อ้างอิงถึง register พิเศษ, เนื้อหาของ register พิเศษบนระบบที่ร้องขอการสืบค้นจะถูกใช้. สำหรับข้อมูลเพิ่มเติมเกี่ยวกับตารางแบบกระจาย, โปรดดูหัวข้อ DB2 Multisystem.

การแปลงชนิดข้อมูล

ในบางครั้ง คุณอาจเจอสถานการณ์ที่ประเภทของนิพจน์จำเป็นต้องถูกแปลงค่า, หรือเปลี่ยนค่า, ไปเป็นประเภทข้อมูลอื่น หรือชนิดข้อมูลเดิมที่มีความยาว, ความแม่นยำ, หรือมาตราส่วนที่ต่างออกไป.

ตัวอย่างเช่น, ถ้าคุณต้องการเปรียบเทียบสองคอลัมน์ที่ต่างประเภทกัน, เช่น ประเภทที่ผู้ใช้กำหนดเองที่เป็นแบบอักขระ และแบบจำนวนเต็ม, คุณสามารถเปลี่ยนอักขระให้เป็นจำนวนเต็ม หรือจำนวนเต็มให้เป็นอักขระได้ เพื่อให้เปรียบเทียบกันได้. ประเภทข้อมูลที่สามารถถูกเปลี่ยนไปเป็นประเภทอื่นคือสามารถแปลงประเภทจากประเภทข้อมูลต้นฉบับไปเป็นประเภทข้อมูลปลายทาง.

คุณสามารถใช้ฟังก์ชันการแปลง หรือค่ากำหนด CAST เพื่อทำการแปลงชนิดข้อมูลไปเป็นชนิดข้อมูลแบบอื่นได้โดยตรง. ตัวอย่างเช่น, ถ้าคุณมีคอลัมน์วันที่ (BIRTHDATE) ที่นิยามเป็น DATE และต้องการแปลงชนิดข้อมูลไปเป็น CHARACTER โดยมีความยาวคงที่คือ 10, ให้ป้อนค่าดังนี้:

```
SELECT CHAR (BIRTHDATE,USA)
FROM CORPDATA.EMPLOYEE
```

คุณสามารถใช้ค่ากำหนด CAST เพื่อแปลงชนิดข้อมูลได้โดยตรง.

```
SELECT CAST(BIRTHDATE AS CHAR(10))
FROM CORPDATA.EMPLOYEE
```

ข้อมูลที่เกี่ยวข้อง

การแปลงชนิดข้อมูล

ประเภทข้อมูลวันที่, เวลา, และ timestamp

วันที่, เวลา, และ timestamp คือ ประเภทข้อมูลที่มีการแทนค่าในรูปแบบภายในซึ่งผู้ใช้ SQL ไม่เห็น.

วันที่, เวลา, และ timestamp สามารถถูกแทนค่าโดยค่าสตริงอักขระ และสามารถถูกกำหนดค่าให้กับตัวแปรสตริงอักขระได้. ผู้จัดการฐานข้อมูลจะยอมรับสิ่งต่อไปนี้เป็นค่าวันที่, เวลา, และ Timestamp:

- ค่าที่คืนค่าโดยฟังก์ชันแบบสเกลาร์ คือ DATE, TIME, หรือ TIMESTAMP.
- ค่าที่คืนค่าโดย register พิเศษ คือ CURRENT DATE, CURRENT TIME, หรือ CURRENT TIMESTAMP.

- ค่าของสตริงอักขระในรูปแบบวันที่, เวลา, หรือ timestamp มาตรฐาน ANSI/ISO, ตัวอย่างเช่น, DATE '1950-01-01'.
- สตริงอักขระเมื่อใช้เป็น Operand ของนิพจน์ทางคณิตศาสตร์หรือการเปรียบเทียบ และ Operand อื่นเป็น Date, Time, หรือ Timestamp. ตัวอย่างเช่น, ในเพรดิเคต:

```
... WHERE HIREDATE < '1950-01-01'
```

ถ้า HIREDATE คือคอลัมน์ข้อมูล, สตริงอักขระ '1950-01-01' จะถูกตีความให้เป็นวันที่.

- ตัวแปรหรือค่าคงที่แบบสตริงอักขระที่ถูกใช้เพื่อตั้งค่าคอลัมน์วันที่, เวลา, หรือ Timestamp ใน SET clause ของคำสั่ง UPDATE, หรือใน VALUES clause ของคำสั่ง INSERT.

ข้อมูลที่เกี่ยวข้อง

ชนิดข้อมูล

การระบุค่าวันที่และเวลาปัจจุบัน:

คุณสามารถระบุวันที่, เวลา, หรือ timestamp ในนิพจน์โดยระบุเรจิสเตอร์พิเศษหนึ่งในสามตัวนี้: CURRENT DATE, CURRENT TIME, หรือ CURRENT TIMESTAMP.

ค่าของแต่ละตัวจะอยู่บนพื้นฐานของการอ่านนาฬิกาบอกเวลาที่ได้รับขณะกำลังรันคำสั่ง. การอ้างอิงหลายตัวที่อ้างไปยัง CURRENT DATE, CURRENT TIME, หรือ CURRENT TIMESTAMP ภายในคำสั่ง SQL เดียวกันจะใช้ค่าเดียวกัน. คำสั่งต่อไปนี้จะคืนค่าอายุ (เป็นปี) ของพนักงานแต่ละคนในตาราง EMPLOYEE เมื่อรันคำสั่ง:

```
SELECT YEAR(CURRENT DATE - BIRTHDATE)
FROM CORPDATA.EMPLOYEE
```

Register พิเศษ CURRENT TIMEZONE อนุญาตให้เวลาท้องถิ่นถูกแปลงไปเป็น Universal Time Coordinated (UTC) ได้. ตัวอย่างเช่น, ถ้าคุณมีตารางชื่อ DATETIME, ที่เก็บประเภทคอลัมน์เวลาในชื่อ STARTT, และคุณต้องการแปลง STARTT ไปเป็น UTC, คุณสามารถใช้คำสั่งต่อไปนี้:

```
SELECT STARTT - CURRENT TIMEZONE
FROM DATETIME
```

การคำนวณวันที่/เวลา:

การบวกและการลบ คือเครื่องหมายคำนวณที่สามารถใช้กับค่าวันที่, เวลา, และ timestamp เท่านั้น.

คุณสามารถเพิ่มค่าและลดค่าวันที่, เวลา, หรือ timestamp เป็นช่วงระยะเวลาได้; หรือลบวันที่จากวันที่, ลบเวลาออกจากเวลา, หรือ ลบ timestamp จาก timestamp.

ข้อมูลที่เกี่ยวข้อง

การคำนวณ Datetime

การจัดการกับแถวซ้ำ

เมื่อ SQL ประเมินผลคำสั่ง select, หลายแถวอาจจะมีคุณสมบัติพอกที่จะอยู่ในตารางผลลัพธ์, ขึ้นอยู่กับจำนวนของแถวที่ตรงกับเงื่อนไขการค้นหาของคำสั่ง select. บางแถวในตารางผลลัพธ์อาจซ้ำกันได้.

คุณสามารถระบุว่าคุณไม่ต้องการข้อมูลซ้ำกันโดยใช้คีย์เวิร์ด DISTINCT, ตามด้วยรายชื่อนิพจน์:

```
SELECT DISTINCT JOB, SEX
...
```


DISTINCT หมายความว่า คุณต้องการเลือกเฉพาะแถวที่ไม่ซ้ำเท่านั้น. ถ้าแถวที่ถูกเลือกมีค่าซ้ำกับแถวอื่นในตารางผลลัพธ์, แถวที่ซ้ำจะถูกข้ามไป (ไม่นำมาใส่ในตารางผลลัพธ์). ตัวอย่างเช่น, สมมติว่าคุณต้องการรายการรหัสนางงานของพนักงาน. คุณไม่จำเป็นต้องรู้ว่าพนักงานคนไหนมีรหัสนางงานอะไร. เนื่องจากบางทีอาจมีหลายคนในแผนกที่มีรหัสนางงานเดียวกัน, ดังนั้นคุณสามารถใช้ DISTINCT เพื่อให้มั่นใจว่าตารางผลลัพธ์จะมีเฉพาะค่าที่ไม่ซ้ำเท่านั้น.

ตัวอย่างต่อไปนี้แสดงวิธีการดังที่กล่าวมา:

```
SELECT DISTINCT JOB
FROM CORPDATA.EMPLOYEE
WHERE WORKDEPT = 'D11'
```

ผลลัพธ์คือสองแถว:

JOB

DESIGNER

MANAGER

ถ้าคุณไม่รวม DISTINCT ไว้ใน SELECT clause, คุณอาจพบกับแถวที่ซ้ำกันในผลลัพธ์ของคุณ, เนื่องจาก SQL ส่งคืนค่าคอลัมน์ JOB สำหรับแต่ละแถวที่ตรงกับเงื่อนไขการค้นหา. ค่า null จะถูกตีความเป็นแถวที่ซ้ำกันสำหรับ DISTINCT.

ถ้าคุณรวม DISTINCT ไว้ใน SELECT และคุณยังรวมลำดับการเรียงแบบ shared-weight, อาจมีค่าส่งคืนเพียงเล็กน้อย. ลำดับการเรียงเป็นสาเหตุให้ค่าที่เก็บตัวอักษรเดียวกันมีน้ำหนักเท่ากัน. ถ้า 'MGR', 'Mgr', และ 'mgr' อยู่ในตารางเดียวกันทั้งหมด, เฉพาะค่าใดค่าหนึ่งในนั้นจะถูกส่งคืน.

หลักการที่เกี่ยวข้อง

“ลำดับการเรียง และ normalization ใน SQL” ในหน้า 119

ลำดับการเรียงกำหนดความสัมพันธ์ของอักขระในชุดอักขระ เมื่อมีการเปรียบเทียบหรือจัดลำดับ. Normalization อนุญาตให้คุณเปรียบเทียบสตริงที่มีอักขระแบบผสม.

การนิยามเงื่อนไขการค้นหาที่ซับซ้อน

นอกจากเพรดิเคตการเปรียบเทียบพื้นฐานแล้ว (=, >, <, และอื่นๆ), เงื่อนไขการค้นหาสามารถมีเพรดิเคต BETWEEN, IN, EXISTS, IS NULL, และ LIKE.

เงื่อนไขการค้นหาสามารถมี scalar fullselect.

สำหรับอักขระ, หรือเพรดิเคตคอลัมน์ที่เป็นกราฟิกแบบ UCS-2 หรือ UTF-16, จะมีการเรียงลำดับในส่วนของ Operand ก่อนที่จะไปทำกับส่วนที่เป็นเพรดิเคต BETWEEN, IN, EXISTS, และ LIKE clauses.

คุณยังสามารถทำเงื่อนไขการค้นหาได้หลายครั้ง.

- **BETWEEN ... AND ...** ถูกใช้เพื่อระบุเงื่อนไขการค้นหา ซึ่งเงื่อนไขจะถูกก็ต่อเมื่อกำหนดอยู่ระหว่างสองค่านี้. ตัวอย่างเช่น, เมื่อต้องการค้นหาพนักงานทั้งหมดที่รับเข้ามาในปี 1987, คุณสามารถเขียนคำสั่งได้ดังนี้:

```
... WHERE HIREDATE BETWEEN '1987-01-01' AND '1987-12-31'
```

คีย์เวิร์ด BETWEEN จะถูกรวมเข้าไปด้วย. เงื่อนไขการค้นหาที่ซับซ้อนขึ้น, แต่ตรงตัว, ซึ่งสร้างผลลัพธ์เดียวกันคือ:

```
... WHERE HIREDATE >= '1987-01-01' AND HIREDATE <= '1987-12-31'
```

- **IN** จะบอกว่าคุณสนใจแถวที่มีค่านิพจน์ที่คุณต้องการอยู่ระหว่างค่าที่คุณทำรายการไว้. ตัวอย่างเช่น, เมื่อต้องการค้นหาชื่อของพนักงานทั้งหมดในแผนก A00, C01, and E21, คุณสามารถระบุว่า:

```
... WHERE WORKDEPT IN ('A00', 'C01', 'E21')
```

- **EXISTS** จะบอกว่าคุณสนใจที่จะทดสอบแถวบางแถวมีอยู่หรือไม่. ตัวอย่างเช่น, เมื่อต้องการค้นหาว่ามีพนักงานคนใดที่มีเงินเดือนมากกว่า 60000, คุณอาจจะระบุ:

```
EXISTS (SELECT * FROM EMPLOYEE WHERE SALARY > 60000)
```

- **IS NULL** จะบอกว่าคุณสนใจที่จะทดสอบเพื่อหาค่า null. ตัวอย่างเช่น, เมื่อต้องการค้นหาว่ามีพนักงานคนใดที่ไม่มีเบอร์โทรศัพท์, คุณอาจจะระบุว่า:

```
... WHERE EMPLOYEE.PHONE IS NULL
```

- **LIKE** จะบอกว่าคุณสนใจแถวที่มีค่านิพจน์เหมือนกับค่าที่คุณจัดหาได้. เมื่อคุณใช้ LIKE, แล้ว SQL จะค้นหาสตริงอักขระที่เหมือนกับค่าที่คุณระบุ. ระดับของความเหมือนจะถูกพิจารณาโดยอักขระพิเศษสองตัวที่ใช้ในสตริงที่คุณรวมเข้าไปในเงื่อนไขการค้นหา:

– ตัวอักขระขีดเส้นใต้แทนตัวอักขระเดี่ยวใดๆ.

% เครื่องหมายเปอร์เซ็นต์แทนสตริงอักขระ 0 หรือมากกว่าที่ไม่รู้ค่า. ถ้าเครื่องหมายเปอร์เซ็นต์อยู่ตอนต้นของสตริงที่ใช้ค้นหา, แล้ว SQL จะอนุญาตให้ตัวอักขระ 0 ตัวหรือมากกว่านั้นมาอยู่หน้าค่าที่ตรงกันในคอลัมน์. มิฉะนั้นแล้ว, สตริงที่ใช้ค้นหาต้องเริ่มต้นที่ตำแหน่งแรกของคอลัมน์.

หมายเหตุ: ถ้าคุณดำเนินการกับข้อมูลแบบ MIXED, ลักษณะพิเศษดังต่อไปนี้จะใช้ได้: ตัวอักขระที่ขีดเส้นใต้ SBCS จะอ้างอิงไปยังอักขระ SBCS หนึ่งตัว. ข้อจำกัดนี้ใช้ไม่ได้กับเครื่องหมายเปอร์เซ็นต์; นั่นคือ, เครื่องหมายเปอร์เซ็นต์จะอ้างอิงถึงตัวอักขระ SBCS หรือ DBCS ก็ตัวก็ได้. โปรดดู การอ้างอิง SQL ใน iSeries Information Center สำหรับข้อมูลเพิ่มเติมเกี่ยวกับเพรดิเคต LIKE และข้อมูล MIXED.

ใช้อักขระขีดเส้นใต้ หรือเครื่องหมายเปอร์เซ็นต์ เมื่อคุณไม่รู้ หรือไม่สนใจเกี่ยวกับอักขระทั้งหมดของค่าคอลัมน์. ตัวอย่างเช่น, เมื่อต้องการค้นหาพนักงานที่อาศัยอยู่ใน Minneapolis, คุณอาจจะระบุว่า:

```
... WHERE ADDRESS LIKE '%MINNEAPOLIS%'
```

SQL จะคืนค่าแถวใดๆ ที่มีสตริง MINNEAPOLIS ในคอลัมน์ ADDRESS, โดยไม่สนใจตำแหน่งของสตริง.

ตัวอย่างถัดมา, เมื่อต้องการแสดงรายชื่อเมืองที่ขึ้นต้นด้วย 'SAN', คุณอาจจะระบุ:

```
... WHERE TOWN LIKE 'SAN%'
```

ถ้าคุณต้องการค้นหาที่อยู่ใดๆ ที่ชื่อถนนไม่ได้อยู่ในรายการชื่อถนนหลักของคุณ, คุณสามารถใช้นิพจน์ LIKE. ในตัวอย่างนี้, คอลัมน์ STREET ในตารางจะถูกสมมติว่าเป็นตัวพิมพ์ใหญ่.

```
... WHERE UCASE (:address_variable) NOT LIKE '%||STREET||%'
```

ถ้าคุณต้องการค้นหาสตริงอักขระที่มีตัวอักขระขีดเส้นใต้หรือตัวอักขระเครื่องหมายเปอร์เซ็นต์อย่างใดอย่างหนึ่งแล้ว, ให้ใช้ ESCAPE clause เพื่อระบุตัวอักขระที่ต้องการหลีกเลี่ยง. ตัวอย่างเช่น, เมื่อต้องการดูธุรกิจทั้งหมดที่มีเปอร์เซ็นต์อยู่ในชื่อ, คุณอาจจะระบุ:

```
... WHERE BUSINESS_NAME LIKE '%@%' ESCAPE '@'
```

อักขระเปอร์เซ็นต์ตัวแรกและตัวสุดท้ายในสตริง LIKE จะถูกตีความเป็นอักขระเปอร์เซ็นต์ LIKE ตามปกติ. การใช้ '@%' ปนเข้าไปด้วยจะถูกพิจารณาว่าเป็นอักขระเปอร์เซ็นต์.

หลักการที่เกี่ยวข้อง

“การใช้เคียวรีย่อย” ในหน้า 110

คุณสามารถใช้เคียวรีย่อยในเงื่อนไขการค้นหาเพื่อเป็นอีกทางหนึ่งในการเลือกข้อมูล. เคียวรีย่อยสามารถใช้ได้ทุกที่ที่นิพจน์สามารถใช้งานได้.

“ลำดับการเรียง และ normalization ใน SQL” ในหน้า 119

ลำดับการจัดเรียงกำหนดความสัมพันธ์ของอักขระในชุดอักขระ เมื่อมีการเปรียบเทียบหรือจัดลำดับ. Normalization อนุญาตให้คุณเปรียบเทียบสตริงที่มีอักขระแบบผสม.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การระบุเงื่อนไขการค้นหาโดยใช้ WHERE clause” ในหน้า 44

WHERE clause จะระบุเงื่อนไขการค้นหาที่บ่งชี้ถึงแถว หรือแถวที่คุณต้องการดึงค่า, อัปเดต, หรือลบ.

“นิพจน์ใน WHERE clause” ในหน้า 45

นิพจน์ใน WHERE clause ใช้เพื่อระบุสิ่งที่คุณต้องการเปรียบเทียบกับสิ่งอื่น.

“หลายเงื่อนไขการค้นหาภายใน WHERE clause” ในหน้า 62

คุณสามารถทำการร้องขอเพิ่มเติม โดยโค้ดเงื่อนไขการค้นหาที่ประกอบด้วยหลายเพรดิเคต.

“ข้อพิจารณาพิเศษสำหรับ LIKE”

นี่คือข้อพิจารณาพิเศษบางข้อสำหรับการใช้ LIKE.

ข้อมูลที่เกี่ยวข้อง

เพรดิเคต

ข้อพิจารณาพิเศษสำหรับ LIKE:

นี่คือข้อพิจารณาพิเศษบางข้อสำหรับการใช้ LIKE.

- เมื่อตัวแปรโฮสต์ถูกใช้แทนค่าคงที่สตริงในรูปแบบการค้นหาแล้ว, คุณควรพิจารณาการใช้ตัวแปรโฮสต์ที่มีความยาวต่างๆ กัน. ซึ่งทำให้คุณสามารถ:
 - กำหนดค่าคงที่สตริงที่ถูกใช้ก่อนหน้านี้ให้กับตัวแปรโฮสต์โดยไม่ต้องเปลี่ยนอะไร.
 - รับค่าเงื่อนไขการเลือกและผลลัพธ์เดียวกันเหมือนกับว่าค่าคงที่สตริงถูกใช้.
- เมื่อตัวแปรโฮสต์ที่มีความยาวคงที่ที่ถูกใช้แทนค่าคงที่สตริงในรูปแบบการค้นหา, คุณควรตรวจสอบให้แน่ใจว่าค่าที่ถูกระบุในตัวแปรโฮสต์มีค่าตรงกับรูปแบบที่ค่าคงที่สตริงใช้ในครั้งก่อน. ตัวอักขระทั้งหมดในตัวแปรโฮสต์ที่ไม่ได้ถูกกำหนดค่าจะถูกกำหนดค่าเริ่มต้นด้วยช่องว่าง.

ตัวอย่างเช่น, ถ้าคุณค้นหาโดยใช้รูปแบบสตริง 'ABC%' ในตัวแปรโฮสต์แบบ *ความยาวแปรผัน* ตัวอย่างค่าที่ได้กลับมามีดังนี้:

```
'ABCD'      'ABCDE'      'ABCxxx'      'ABC'
```

อย่างไรก็ตาม, ถ้าคุณค้นหาโดยใช้รูปแบบการค้นหา 'ABC%' ที่อยู่ในตัวแปรโฮสต์ที่มี *ความยาวคงที่* เท่ากับ 10, นี่คือนางค่าที่อาจได้กลับมา ถ้าสมมติว่าคอลัมน์มีความยาวเท่ากับ 12:

```
'ABCDE'      'ABCD'      'ABCxxx'      'ABC'
```

โปรดสังเกตว่าค่าที่คืนกลับมาทั้งหมดจะเริ่มต้นด้วย 'ABC' และสิ้นสุดด้วยช่องว่างอย่างน้อย 6 ตัว. ที่เป็นเช่นนี้เนื่องจากอักขระ 6 ตัวสุดท้ายในตัวแปรโฮสต์จะไม่ถูกกำหนดด้วยค่าที่เฉพาะ ดังนั้นช่องว่างจะถูกใช้แทน.

ถ้าคุณต้องการค้นหาโดยใช้ตัวแปรโฮสต์ที่มีความยาวคงที่โดยมีอักขระ 7 ตัวสุดท้ายสามารถเป็นค่าใดก็ได้, ให้ค้นหาเป็นลักษณะ 'ABC%?????'. ค่าที่ได้คืนค่ากลับมาจะเป็น:

```
'ABCDEFGH'IJ'      'ABCXXXXXXX'      'ABCDE'      'ABCDD'
```

สิ่งอ้างอิงที่เกี่ยวข้อง

“การนิยามเงื่อนไขการค้นหาที่ซับซ้อน” ในหน้า 59

นอกจากเพรดิเคตการเปรียบเทียบพื้นฐานแล้ว (=, >, <, และอื่นๆ), เงื่อนไขการค้นหาสามารถมีเพรดิเคต BETWEEN, IN, EXISTS, IS NULL, และ LIKE.

หลายเงื่อนไขการค้นหาภายใน WHERE clause:

คุณสามารถทำการร้องขอเพิ่มเติม โดยโค้ดเงื่อนไขการค้นหาที่ประกอบด้วยหลายเพรดิเคต.

เงื่อนไขการค้นหาที่คุณระบุสามารถมี comparison operators หรือเพรดิเคต BETWEEN, DISTINCT, IN, LIKE, EXISTS, IS NULL, และ IS NOT NULL.

คุณสามารถรวมสองเพรดิเคตใดๆ เข้ากันด้วยตัวเชื่อม AND และ OR. นอกจากนี้, คุณสามารถใช้คีย์เวิร์ด NOT เพื่อระบุเงื่อนไขการค้นหาที่ต้องการคือค่าตรงข้ามกับเงื่อนไขการค้นหาที่ระบุ. WHERE clause สามารถมีได้หลายเพรดิเคตตามที่คุณต้องการ.

- AND จะบอกว่า, เพื่อให้แถวข้อมูลถูกต้อง, แถวจะต้องตรงกับเพรดิเคตทั้งคู่ของเงื่อนไขการค้นหา. ตัวอย่างเช่น, เมื่อต้องการค้นหาพนักงานในแผนก D21 ที่รับเข้ามาทำงานหลังจาก 31 ธันวาคม, 1987, คุณอาจจะระบุ:

```
...  
WHERE WORKDEPT = 'D21' AND HIREDATE > '1987-12-31'
```

- OR จะบอกว่า, เพื่อให้แถวข้อมูลถูกต้อง, แถวต้องตรงกับเงื่อนไขที่ตั้งค่าโดยเพรดิเคตของเงื่อนไขการค้นหา เงื่อนไขใดเงื่อนไขหนึ่งหรือทั้งคู่. ตัวอย่างเช่น, เมื่อต้องการค้นหาพนักงานที่อยู่ในแผนก C01 หรือแผนก D11, คุณอาจจะระบุ:

```
...  
WHERE WORKDEPT = 'C01' OR WORKDEPT = 'D11'
```

หมายเหตุ: คุณยังสามารถใช้ IN เพื่อระบุในการร้องขอ: WHERE WORKDEPT IN ('C01', 'D11').

- NOT จะบอกว่า, เพื่อให้ถูกต้องตามเกณฑ์, แถวต้องไม่ตรงกับเกณฑ์ที่ตั้งขึ้นโดยเงื่อนไขการค้นหาหรือเพรดิเคตที่ตามหลัง NOT. ตัวอย่างเช่น, เมื่อต้องการค้นหาพนักงานในแผนก E11 ยกเว้นพวกที่มีรหัสงานเท่ากับ "analyst", คุณอาจจะระบุ:

```
...  
WHERE WORKDEPT = 'E11' AND NOT JOB = 'ANALYST'
```

เมื่อ SQL ประเมินผลเงื่อนไขการค้นหาที่มีตัวเชื่อมเหล่านี้, SQL จะทำในลำดับเฉพาะ. และจะประเมินผล NOT clause ก่อน, ต่อมาจึงประเมินผล AND clause, ตามด้วย OR clause.

คุณสามารถเปลี่ยนแปลงลำดับของการประเมินผลโดยการใส่วงเล็บ. เงื่อนไขการค้นหาที่อยู่ในวงเล็บจะถูกประเมินผลก่อน. ตัวอย่างเช่น, เมื่อต้องการค้นหาพนักงานในแผนก E11 และ E21 ที่มีการศึกษาสูงกว่าระดับ 12, คุณสามารถระบุ:

```
...  
WHERE EDLEVEL > 12 AND  
(WORKDEPT = 'E11' OR WORKDEPT = 'E21')
```

วงเล็บจะกำหนดความหมายของเงื่อนไขการค้นหา. ในตัวอย่างนี้, คุณต้องการแถวทั้งหมดที่มี:

- ค่า WORKDEPT เป็น E11 หรือ E21, และ
- ค่า EDLEVEL ที่มากกว่า 12

ถ้าคุณไม่ได้ใส่วงเล็บ:

```
...
WHERE EDLEVEL > 12 AND WORKDEPT = 'E11'
      OR WORKDEPT = 'E21'
```

ผลลัพธ์ของคุณจะแตกต่างออกไป. แถวที่ถูกเลือกคือแถวที่มี:

- WORKDEPT = E11 and EDLEVEL > 12, or
- WORKDEPT = E21, โดยไม่สนใจค่าของ EDLEVEL

| ถ้าคุณกำลังรวมการเปรียบเทียบการเท่ากันหลายๆ ครั้ง, คุณสามารถเขียนเพรดิเคตด้วย AND ตามที่แสดงไว้ในตัวอย่างต่อไปนี้:

```
| ...
| WHERE WORKDEPT = 'E11' AND EDLEVEL = 12 AND JOB = 'CLERK'
```

| คุณสามารถเปรียบเทียบรายการสองรายการ, ตัวอย่างเช่น:

```
| ...
| WHERE (WORKDEPT, EDLEVEL, JOB) = ('E11', 12, 'CLERK')
```

| เมื่อรายการสองรายการถูกใช้, ไอเท็มแรกในรายการแรกจะถูกเปรียบเทียบกับไอเท็มแรกในรายการที่สอง, และเป็นเช่นนี้ตลอดจนรายการทั้งสอง. ดังนั้น, แต่ละรายการต้องมีจำนวนของ entry ตรงกัน. การใช้รายการเฉพาะเพื่อเขียนเคียวรีด้วย AND. รายการอาจถูกใช้ด้วย comparison operator เท่ากับ หรือไม่เท่ากับเท่านั้น.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การนิยามเงื่อนไขการค้นหาที่ซับซ้อน” ในหน้า 59

นอกจากเพรดิเคตการเปรียบเทียบพื้นฐานแล้ว (=, >, <, และอื่นๆ), เงื่อนไขการค้นหาสามารถมีเพรดิเคต BETWEEN, IN, EXISTS, IS NULL, และ LIKE.

“การระบุเงื่อนไขการค้นหาโดยใช้ WHERE clause” ในหน้า 44

WHERE clause จะระบุเงื่อนไขการค้นหาที่บ่งชี้ถึงแถว หรือแถวที่คุณต้องการดึงค่า, อัปเดต, หรือลบ.

| การใช้ค่ากำหนด OLAP

| ค่ากำหนด Online analytical processing (OLAP) จะถูกใช้เพื่อส่งคืนลำดับหมายเลข และหมายเลขแถวที่เป็นผลลัพธ์แถวของเคียวรี. คุณสามารถระบุ RANK, DENSE_RANK, และ ROW_NUMBER ได้.

| ตัวอย่าง: การจัดลำดับและการกำหนดหมายเลขแถว

| สมมติว่า คุณต้องการรายการของเงินเดือน 10 อันดับแรกตามลำดับ. เคียวรีต่อไปนี้จะจัดลำดับหมายเลขให้คุณ.

```
| SELECT EMPNO, SALARY
|         RANK() OVER(ORDER BY SALARY DESC),
|         DENSE_RANK() OVER(ORDER BY SALARY DESC),
|         ROW_NUMBER() OVER(ORDER BY SALARY DESC)
| FROM EMPLOYEE
| FETCH FIRST 10 ROWS ONLY
```

เคียวรีนี่ส่งคืนข้อมูลต่อไปนี้.

ตารางที่ 9. ผลลัพธ์ของเคียวรีก่อนหน้า

EMPNO	SALARY	RANK	DENSE_RANK	ROW_NUMBER
000010	52,750.00	1	1	1
000110	46,500.00	2	2	2
200010	46,500.00	2	2	3
000020	41,250.00	4	3	4
000050	40,175.00	5	4	5
000030	38,250.00	6	5	6
000070	36,170.00	7	6	7
000060	32,250.00	8	7	8
000220	29,840.00	9	8	9
200220	29,840.00	9	8	10

ในตัวอย่างนี้, SALARY จะเรียงจากมากไปน้อยด้วย 10 แรกที่ถูกส่งคืนค่า. คอลัมน์ RANK จะแสดงการจัดลำดับของแต่ละเงินเดือน. สังเกตว่า มีอยู่สองแถวที่มีเงินเดือนเท่ากันที่ตำแหน่ง 2. แต่ละแถวจะถูกกำหนดค่าลำดับที่เหมือนกัน. แถวต่อไปนี้จะถูกกำหนดค่าเป็น 4. RANK จะส่งคืนค่าของแถวที่มากกว่าจำนวนของแถวทั้งหมดที่อยู่ก่อนหน้าแถวนั้น. ซึ่งจะมีช่องว่างอยู่ระหว่างลำดับหมายเลขไม่ว่าลำดับหมายเลขเหล่านั้นจะซ้ำกันหรือไม่ก็ตาม.

ในทางตรงกันข้าม, คอลัมน์ DENSE_RANK จะแสดงค่า 3 สำหรับแถวโดยตรง หลังจากแถวที่ซ้ำกัน. DENSE_RANK ส่งคืนค่าสำหรับแถวซึ่งเป็นค่าที่มากกว่าจำนวนของค่าแถวที่ต่างกันซึ่งอยู่ก่อนหน้าแถวนั้น. และจะไม่มีช่องว่างในลำดับหมายเลข.

ROW_NUMBER ส่งคืนหมายเลขเฉพาะสำหรับแต่ละแถว. สำหรับแต่ละแถวที่มีค่าซ้ำกันตามลำดับที่ระบุ, การกำหนดหมายเลขแถวจะไม่มีเกณฑ์; หมายเลขแถวสามารถถูกกำหนดด้วยลำดับที่แตกต่างกันสำหรับแถวที่ซ้ำกัน เมื่อรันเคียวรีอีกครั้ง.

ตัวอย่าง: กลุ่มการจัดลำดับ

ในตัวอย่างนี้, สมมติว่า คุณต้องการค้นหาแผนกที่มีเงินเดือนโดยเฉลี่ยสูงที่สุด. เคียวรีต่อไปนี้จะจัดกลุ่มข้อมูลตามแผนก, คำนวณเงินเดือนโดยเฉลี่ยสำหรับแต่ละแผนก, แล้วจึงจัดลำดับค่าเฉลี่ยที่ได้.

```
SELECT WORKDEPT, INT(AVG(SALARY)) AS AVERAGE,  
       RANK() OVER(ORDER BY AVG(SALARY) DESC) AS AVG_SALARY  
FROM EMPLOYEE  
GROUP BY WORKDEPT
```

เคียวรีนี่ส่งคืนข้อมูลต่อไปนี้.

ตารางที่ 10. ผลลัพธ์ของเคียวรีก่อนหน้า

WORKDEPT	AVERAGE	AVG_SALARY
B01	41,250	1

| ตารางที่ 10. ผลลัพธ์ของเคียวรีก่อนหน้า (ต่อ)

WORKDEPT	AVERAGE	AVG_SALARY
A00	40,850	2
E01	40,175	3
C01	29,722	4
D21	25,668	5
D11	25,147	6
E21	24,086	7
E11	21,020	8

| ตัวอย่าง: การจัดลำดับภายในแผนก

| สมมติว่า คุณต้องการรายการของพนักงานพร้อมกับการจัดลำดับโบนัสของพนักงานเหล่านั้นภายในแผนก. การใช้ PARTITION BY clause, คุณสามารถระบุกลุ่มโดยแยกหมายเลขต่างหาก.

```

| SELECT LASTNAME, WORKDEPT, BONUS,
|         DENSE_RANK() OVER(PARTITION BY WORKDEPT ORDER BY BONUS DESC)
|         AS BONUS_RANK_IN_DEPT
| FROM EMPLOYEE
| WHERE WORKDEPT LIKE 'E%'

```

| เคียวรีนี้ส่งคืนข้อมูลต่อไปนี้.

| ตารางที่ 11. ผลลัพธ์ของเคียวรีก่อนหน้า

LASTNAME	WORKDEPT	BONUS	BONUS_RANK_in_DEPT
GEYER	E01	800.00	1
HENDERSON	E11	600.00	1
SCHNEIDER	E11	500.00	2
SCHWARTZ	E11	500.00	2
SMITH	E11	400.00	3
PARKER	E11	300.00	4
SETRIGHT	E11	300.00	4
SPRINGER	E11	300.00	4
SPENSER	E21	500.00	1
LEE	E21	500.00	1
GOUNOT	E21	500.00	1

| ตารางที่ 11. ผลลัพธ์ของเคิวรี่ก่อนหน้า (ต่อ)

LASTNAME	WORKDEPT	BONUS	BONUS_RANK_in_DEPT
WONG	E21	500.00	1
ALONZO	E21	500.00	1
MENTA	E21	400.00	2

| ตัวอย่าง: การจัดลำดับและการเรียงลำดับด้วยผลลัพธ์ของนิพจน์ตาราง

| สมมติว่า คุณต้องการค้นหาพนักงานห้าคนแรกที่มีเงินเดือนสูงที่สุดพร้อมด้วยชื่อแผนก. ชื่อแผนกอยู่ในตารางแผนก, ดังนั้นจึงจำเป็นต้องใช้การรวม. เนื่องจากการเรียงลำดับถูกทำในนิพจน์ตารางที่ซ้อนกัน, การเรียงลำดับจึงสามารถใช้เพื่อกำหนดค่าของ ROW_NUMBER. ORDER BY ORDER OF table clause จะถูกใช้ในกรณีนี้.

```
| SELECT ROW_NUMBER() OVER(ORDER BY ORDER OF EMP),
|         EMPNO, SALARY, DEPTNO, DEPTNAME
|   FROM (SELECT EMPNO, WORKDEPT, SALARY
|         FROM EMPLOYEE
|         ORDER BY SALARY DESC
|         FETCH FIRST 5 ROWS ONLY) EMP,
|         DEPARTMENT
|   WHERE DEPTNO = WORKDEPT
```

| เคิวรี่นี้ส่งคืนข้อมูลต่อไปนี้.

| ตารางที่ 12. ผลลัพธ์ของเคิวรี่ก่อนหน้า

ROW_NUMBER	EMPNO	SALARY	DEPTNO	DEPTNAME
1	000010	52,750.00	A00	SPIFFY COMPUTER SERVICE DIV.
2	000110	46,500.00	A00	SPIFFY COMPUTER SERVICE DIV.
3	200010	46,500.00	A00	SPIFFY COMPUTER SERVICE DIV.
4	000020	41,250.00	B01	PLANNING
5	000050	40,175.00	E01	SUPPORT SERVICES

| การรวมข้อมูลจากตารางมากกว่าหนึ่งตาราง

บางครั้งข้อมูลที่คุณต้องการดูไม่ได้อยู่ในตารางเดียว. เมื่อต้องการสร้างแถวของตารางผลลัพธ์, คุณอาจต้องการดึงค่าบางคอลัมน์จากตารางหนึ่งและบางคอลัมน์จากตารางอื่น. คุณสามารถดึงค่าคอลัมน์และรวมค่าคอลัมน์จากสองตารางหรือมากกว่าเข้าไปอยู่ในแถวเดียวได้.

การรวมหลายประเภทได้รับการรองรับจาก DB2 UDB for iSeries: inner join, left outer join, right outer join, left exception join, right exception join, และ cross join.

การใช้หมายเหตุบนการดำเนินการรวม

เมื่อคุณรวมตารางตั้งแต่สองตารางขึ้นไป, ให้พิจารณารายการต่อไปนี้:

- ถ้ามีชื่อคอลัมน์ร่วมกัน, คุณจะต้องระบุแต่ละชื่อที่ตรงกันด้วยชื่อของตาราง (หรือชื่อที่สัมพันธ์กัน). ชื่อคอลัมน์ที่มีชื่อไม่ซ้ำไม่จำเป็นต้องระบุชื่อตาราง. อย่างไรก็ตาม, สามารถใช้ USING clause ในการรวม เพื่ออนุญาตให้คุณสามารถแยกแยะคอลัมน์ที่ซ้ำกันในตารางทั้งสองโดยไม่ต้องระบุชื่อของตาราง.
- ถ้าคุณไม่ได้ทำรายการชื่อคอลัมน์ที่คุณต้องการ, แต่ใช้ SELECT * แทน, SQL จะคืนค่าแถวที่ประกอบด้วยทุกคอลัมน์ของตารางแรก, ตามด้วยทุกคอลัมน์ของตารางที่สอง, เช่นนี้ไปเรื่อยๆ.
- คุณจะต้องมีสิทธิในการเลือกแถวจากตารางหรือมุมมองที่ระบุใน FROM clause.
- ลำดับการเรียงจะถูกใช้กับคอลัมน์แบบอักษร, หรือคอลัมน์กราฟิก UCS-2 หรือ UTF-16 ทั้งหมดที่ถูกรวม.

Inner join:

inner join จะคืนค่าเฉพาะแถวจากแต่ละตารางที่มีค่าที่จับคู่กันในคอลัมน์ร่วม. แถวใดๆ ที่ไม่มีค่าที่ตรงกันระหว่างตารางจะไม่ปรากฏในตารางผลลัพธ์.

โดยการใช้ Inner Join, ค่าคอลัมน์จากหนึ่งแถวของตารางจะถูกรวมกับค่าคอลัมน์จากอีกแถวหนึ่งของตารางอื่น (หรือตารางเดียวกัน) เพื่อสร้างแถวข้อมูลเดียว. SQL จะพิจารณาตารางทั้งคู่ที่ถูกระบุสำหรับการรวม เพื่อดึงค่าจากทุกแถวที่ตรงกับเงื่อนไขการค้นหาสำหรับการรวม. จะมีอยู่สองวิธีในการระบุ Inner Join: โดยการใช้ซินแทกซ์ JOIN, และโดยการใช้ WHERE clause.

สมมติว่าคุณต้องการดึงค่าหมายเลขพนักงาน, ชื่อ, และหมายเลขโครงการสำหรับพนักงานทั้งหมดที่ต้องรับผิดชอบโครงการ. หรือพูดอีกอย่างหนึ่งว่า, คุณต้องการคอลัมน์ EMPNO และ LASTNAME จากตาราง CORPDATA.EMPLOYEE และคอลัมน์ PROJNO จากตาราง CORPDATA.PROJECT. เฉพาะพนักงานที่มีนามสกุลเริ่มต้นด้วย 'S' หรือต่อมาเท่านั้นที่จะถูกพิจารณา. เมื่อต้องการค้นหาข้อมูลนี้, คุณจำเป็นต้องรวมสองตารางเข้าด้วยกัน.

Inner join โดยใช้ไวยากรณ์ JOIN:

โดยการใช้ไวยากรณ์ inner join, ตารางทั้งสองที่คุณกำลังรวมกันจะถูกแสดงใน FROM clause, พร้อมด้วยเงื่อนไขการรวมที่ใช้กับตารางเหล่านั้น.

เงื่อนไขการรวมจะถูกระบุหลังคีย์เวิร์ด ON และจะใช้ในการพิจารณาว่าตารางทั้งสองจะเปรียบเทียบกันเพื่อสร้างผลลัพธ์การรวมได้อย่างไร. เงื่อนไขอาจเป็นตัวดำเนินการเปรียบเทียบใดๆ ก็ได้; ไม่จำเป็นต้องเป็นตัวดำเนินการเท่ากับเท่านั้น. เงื่อนไขการรวมหลายตัวสามารถระบุใน ON clause ได้โดยแยกกันด้วยคีย์เวิร์ด AND. เงื่อนไขเพิ่มเติมใดๆ ที่ไม่เกี่ยวกับการรวมจะถูกระบุใน WHERE clause หรือระบุเป็นส่วนของการรวมใน ON clause.

```
SELECT EMPNO, LASTNAME, PROJNO
FROM CORPDATA.EMPLOYEE INNER JOIN CORPDATA.PROJECT
ON EMPNO = RESPEMP
WHERE LASTNAME > 'S'
```

ในตัวอย่างนี้, การรวมจะทำกับตารางสองตารางโดยใช้คอลัมน์ EMPNO และ RESPEMP จากตาราง. เนื่องจากเฉพาะพนักงานที่มีนามสกุลขึ้นต้นด้วย 'S' อย่างน้อยหนึ่งตัวเท่านั้นที่จะถูกคืนค่ากลับมา, เงื่อนไขเพิ่มเติมนี้จะถูกจัดเตรียมใน WHERE clause.

เคียวรีนี้จะคือค่าผลลัพธ์ดังนี้:

EMPNO	LASTNAME	PROJNO
000250	SMITH	AD3112
000060	STERN	MA2110
000100	SPENSER	OP2010
000020	THOMPSON	PL2100

Inner join โดยใช้ WHERE clause:

การใช้ WHERE clause เพื่อให้เกิดการรวมแบบเดียวกันใน Inner join โดยใช้ไวยากรณ์ JOIN ทำได้โดยใส่เงื่อนไขการรวม และเงื่อนไขการเลือกเพิ่มเติมใน WHERE clause.

ตารางที่จะถูกรวมจะถูกแสดงรายชื่อใน FROM clause, แยกกันด้วยเครื่องหมายจุลภาค.

```
SELECT EMPNO, LASTNAME, PROJNO
FROM CORPDATA.EMPLOYEE, CORPDATA.PROJECT
WHERE EMPNO = RESPEMP
AND LASTNAME > 'S'
```

เคียวรีนี้จะคืนค่าผลลัพธ์เดียวกับตัวอย่างที่แล้ว.

การรวมข้อมูลด้วย USING clause:

คุณสามารถกำหนดเงื่อนไขการรวมแบบย่อโดยใช้ USING clause. USING clause เหมือนกับเงื่อนไขการรวม ซึ่งแต่ละคอลัมน์จากตารางด้านซ้ายจะถูกเปรียบเทียบกับคอลัมน์ที่มีชื่อเดียวกันที่อยู่ในตารางด้านขวา.

ตัวอย่าง, การใช้ USING clause :

```
SELECT EMPNO, ACSTDATE
FROM CORPDATA.PROJECT INNER JOIN CORPDATA.EMPPROJECT
USING (PROJNO, ACTNO)
WHERE ACSTDATE > '1982-12-31';
```

ข้อความด้านบนมีไวยากรณ์ที่ถูกต้อง และจะให้ผลเหมือนกับเงื่อนไขการรวมในประโยคด้านล่างนี้:

```
SELECT EMPNO, ACSTDATE
FROM CORPDATA.PROJECT INNER JOIN CORPDATA.EMPPROJECT
ON CORPDATA.PROJECT.PROJNO = CORPDATA.EMPPROJECT.PROJNO AND
CORPDATA.PROJECT.ACTNO = CORPDATA.EMPPROJECT.ACTNO
WHERE ACSTDATE > '1982-12-31';
```

Left outer join:

left outer join จะคืนค่าทุกแถวที่ inner join คืนค่ามา บวกกับอีกหนึ่งแถวสำหรับแถวอื่นแต่ละแถวในตารางแรกที่ไม่มีค่าที่จับคู่กับตารางที่สอง.

สมมติว่าคุณต้องการค้นหาพนักงานทั้งหมดและโครงการที่พนักงานคนนั้นกำลังรับผิดชอบอยู่ในปัจจุบัน. คุณต้องการดูพนักงานที่ไม่ได้ทำโครงการโดยอยู่ด้วยเหมือนกัน. เคียวยี่ต่อไปนี้จะคืนรายชื่อของพนักงานทั้งหมดที่มีชื่อมากกว่า 'S', พร้อมกับโครงการที่ได้รับมอบหมาย.

```
SELECT EMPNO, LASTNAME, PROJNO
FROM CORPDATA.EMPLOYEE LEFT OUTER JOIN CORPDATA.PROJECT
ON EMPNO = RESPEMP
WHERE LASTNAME > 'S'
```

ผลลัพธ์ของการสืบค้นนี้มีข้อมูลพนักงานบางคนที่ไม่มีหมายเลขโครงการอยู่. เขาจะถูกแสดงชื่ออยู่ในการสืบค้น, แต่จะมีค่า null ที่คืนค่ากลับมาสำหรับหมายเลขโครงการของเขา.

EMPNO	LASTNAME	PROJNO
000020	THOMPSON	PL2100
000060	STERN	MA2110
000100	SPENSER	OP2010
000170	YOSHIMURA	-
000180	SCOUTTEN	-
000190	WALKER	-
000250	SMITH	AD3112
000280	SCHNEIDER	-
000300	SMITH	-
000310	SETRIGHT	-
200170	YAMAMOTO	-
200280	SCHWARTZ	-
200310	SPRINGER	-
200330	WONG	-

หมายเหตุ: การใช้ฟังก์ชันแบบสเกลาร์ชื่อ RRN เพื่อคืนค่าหมายเลขเร็กคอร์ดเชิงสัมพันธ์สำหรับคอลัมน์ในตารางด้านขวามือของ Left Outer Join หรือ Exception Join จะคืนค่าเป็น 0 สำหรับแถวที่ไม่มีค่าที่จับคู่กัน.

Right outer join:

right outer join จะคืนค่าทุกแถวที่ inner join คืนค่ามา บวกกับหนึ่งแถวสำหรับแถวอื่นแต่ละแถวในตารางที่สองที่ไม่ค่าที่จับคู่กับตารางแรก. ลักษณะนี้จะเหมือนกับการทำ Left Outer Join ด้วยตารางที่ระบุในลำดับที่ตรงกันข้ามกัน.

เคียวยี่ที่ใช้เป็นตัวอย่างของ Left Outer Join สามารถนำมาเขียนใหม่ด้วย Right Outer Join ได้ดังนี้:

```

SELECT EMPNO, LASTNAME, PROJNO
  FROM CORPDATA.PROJECT RIGHT OUTER JOIN CORPDATA.EMPLOYEE
    ON EMPNO = RESPEMP
 WHERE LASTNAME > 'S'

```

ผลลัพธ์ของเคียวรีนี้จะเหมือนกันทุกประการกับผลลัพธ์จากเคียวรีแบบ Left Outer Join.

Exception join:

left exception join จะคืนค่าเฉพาะแถวจากตารางแรกที่ไม่ม่มีค่าที่จับคู่กับตารางที่สอง.

โดยการใ้ตารางเดียวกันเหมือนก่อนหน้าี้, จะคืนค่าพนักงานที่ไม่ได้รับผิดชอบโครงการใดๆอยู่.

```

SELECT EMPNO, LASTNAME, PROJNO
  FROM CORPDATA.EMPLOYEE EXCEPTION JOIN CORPDATA.PROJECT
    ON EMPNO = RESPEMP
 WHERE LASTNAME > 'S'

```

การรวมนี้จะคืนค่าผลลัพธ์:

EMPNO	LASTNAME	PROJNO
000170	YOSHIMURA	-
000180	SCOUTTEN	-
000190	WALKER	-
000280	SCHNEIDER	-
000300	SMITH	-
000310	SETRIGHT	-
200170	YAMAMOTO	-
200280	SCHWARTZ	-
200310	SPRINGER	-
200330	WONG	-

Exception Join ยังสามารถเขียนให้เป็นการสืบค้นย่อยโดยใช้เพรดิเคต NOT EXISTS ได้. เคียวรีที่แล้วสามารถเขียนใหม่ได้ดังนี้:

```

SELECT EMPNO, LASTNAME
  FROM CORPDATA.EMPLOYEE
 WHERE LASTNAME > 'S'
    AND NOT EXISTS
      (SELECT * FROM CORPDATA.PROJECT
       WHERE EMPNO = RESPEMP)

```

ข้อแตกต่างเดียวเท่านั้นในการสืบค้นนี้ก็คือจะไม่มีการคืนค่าจากตาราง PROJECT.

นั่นคือ Right Exception Join, ด้วย, ที่ทำงานคล้าย Left Exception Join แต่ทำงานด้วยตารางที่กลับด้านกัน.

Cross join:

Cross join (หรือ ผลรวมคาร์ทีเซียน) คำนวณค่าตารางผลลัพธ์ที่เกิดจากการรวมค่าแต่ละแถวจากตารางแรกกับแต่ละแถวจากตารางที่สอง.

จำนวนของแถวในตารางผลลัพธ์จะเป็นผลคูณของจำนวนแถวในแต่ละตาราง. ถ้าตารางมีขนาดใหญ่, การรวมนี้อาจใช้เวลานานมาก.

Cross Join สามารถระบุได้ด้วยสองวิธี: โดยการใช้ซินแทกซ์ JOIN หรือโดยการแสดงรายชื่อตารางใน FROM clause ที่แบ่งโดยเครื่องหมายจุลภาคโดยไม่ใช้ WHERE clause เพื่อเรียกเงื่อนไขการรวม.

สมมติมีตารางดังต่อไปนี้.

ตารางที่ 13. ตาราง A

ACOL1	ACOL2
A1	AA1
A2	AA2
A3	AA3

ตารางที่ 14. ตาราง B

BCOL1	BCOL2
B1	BB1
B2	BB2

คำสั่ง Select ดังต่อไปนี้ จะสร้างผลลัพธ์ที่เท่ากันทุกประการ.

```
SELECT * FROM A CROSS JOIN B
```

```
SELECT * FROM A, B
```

ตารางผลลัพธ์ของคำสั่ง select อย่างไม่อย่างหนึ่ง จะมีลักษณะดังนี้:

ACOL1	ACOL2	BCOL1	BCOL2
A1	AA1	B1	BB1
A1	AA1	B2	BB2
A2	AA2	B1	BB1
A2	AA2	B2	BB2
A3	AA3	B1	BB1
A3	AA3	B2	BB2

การจำลอง full outer join:

Full Outer Join จะคืนค่าแถวที่จับคู่กันจากตารางทั้งคู่, เหมือนกับ Left และ Right Outer Join. อย่างไรก็ตาม, full outer join ยังได้คืนค่าแถวที่ไม่จับคู่กันจากตารางทั้งคู่ด้วย; นั่นคือ ตารางทางซ้ายและตารางทางขวา.

เนื่องจาก DB2 UDB สำหรับ iSeries ไม่สนับสนุนซินแทกซ์แบบ Full Outer Join , แต่คุณก็สามารถจำลอง Full Outer Join โดยใช้ Left Outer Join กับ Right Exception Join ได้. สมมติว่าคุณต้องการค้นหาพนักงานทั้งหมดและโครงการทั้งหมด และคุณต้องการดูพนักงานที่ไม่ได้ทำโครงการใดอยู่ด้วยเหมือนกัน. เคียวยวิธีต่อไปนี้จะคืนรายชื่อของพนักงานทั้งหมดที่มีชื่อมากกว่า 'S', พร้อมกับโครงการที่ได้รับมอบหมาย.

```
SELECT EMPNO, LASTNAME, PROJNO
  FROM CORPDATA.EMPLOYEE LEFT OUTER JOIN CORPDATA.PROJECT
    ON EMPNO = RESPEMP
 WHERE LASTNAME > 'S'
 UNION
 (SELECT EMPNO, LASTNAME, PROJNO
  FROM CORPDATA.PROJECT EXCEPTION JOIN CORPDATA.EMPLOYEE
    ON EMPNO = RESPEMP
 WHERE LASTNAME > 'S');
```

ประเภทการรวมหลายประเภทในหนึ่งคำสั่ง:

มีบางครั้งที่จะต้องรวมตารางมากกว่าสองตาราง เพื่อให้เกิดผลตามต้องการ.

ถ้าคุณต้องการรายชื่อพนักงาน, ชื่อแผนกของพนักงานนั้น, และโครงการที่พนักงานนั้นรับผิดชอบ, จึงจำเป็นต้องรวม, ตาราง EMPLOYEE, ตาราง DEPARTMENT, และตาราง PROJECT เพื่อให้ได้ข้อมูลตามต้องการ. ตัวอย่างต่อไปนี้จะแสดงการสืบค้นและผลลัพธ์.

```
SELECT EMPNO, LASTNAME, DEPTNAME, PROJNO
  FROM CORPDATA.EMPLOYEE INNER JOIN CORPDATA.DEPARTMENT
    ON WORKDEPT = DEPTNO
 LEFT OUTER JOIN CORPDATA.PROJECT
    ON EMPNO = RESPEMP
 WHERE LASTNAME > 'S'
```

ผลลัพธ์ของการสืบค้นนี้คือ:

EMPNO	LASTNAME	DEPTNAME	PROJNO
000020	THOMPSON	PLANNING	PL2100
000060	STERN	MANUFACTURING SYSTEMS	MA2110
000100	SPENSER	SOFTWARE SUPPORT	OP2010
000170	YOSHIMURA	MANUFACTURING SYSTEMS	-
000180	SCOUTTEN	MANUFACTURING SYSTEMS	-
000190	WALKER	MANUFACTURING SYSTEMS	-

EMPNO	LASTNAME	DEPTNAME	PROJNO
000250	SMITH	ADMINISTRATION SYSTEMS	AD3112
000280	SCHNEIDER	OPERATIONS	-
000300	SMITH	OPERATIONS	-
000310	SETRIGHT	OPERATIONS	-

การใช้นิพจน์ตาราง

คุณสามารถใช้นิพจน์ตารางเพื่อระบุตารางผลลัพธ์ชั่วคราวได้.

นิพจน์ตารางสามารถถูกใช้แทนมุมมองเพื่อหลีกเลี่ยงการสร้างมุมมอง เมื่อไม่จำเป็นต้องใช้มุมมอง. นิพจน์ตารางประกอบด้วยนิพจน์ตารางที่ซ้อนกัน (ซึ่งเรียกว่าตารางลูก) และนิพจน์ตารางกลาง.

นิพจน์ตารางที่ซ้อนกันจะถูกระบุภายในวงเล็บใน FROM clause. ตัวอย่างเช่น, สมมติว่าคุณต้องการตารางผลลัพธ์ที่แสดงหมายเลขผู้จัดการ, หมายเลขแผนก, และเงินเดือนสูงสุดสำหรับแต่ละแผนก. หมายเลขผู้จัดการอยู่ในตาราง DEPARTMENT, หมายเลขแผนกอยู่ในทั้งตาราง DEPARTMENT และ EMPLOYEE, และเงินเดือนอยู่ในตาราง EMPLOYEE. คุณสามารถใช้นิพจน์ตารางใน FROM clause เพื่อเลือกเงินเดือนสูงสุดสำหรับแต่ละแผนก. คุณยังสามารถเพิ่มชื่อที่สัมพันธ์กัน, T2, ตามด้วยนิพจน์ที่ซ้อนกันเพื่อตั้งชื่อตารางลูก. คำสั่ง Select ด้านนอกจะใช้ T2 เพื่อระบุคอลัมน์ที่ถูกเลือกจากตารางลูก, ในกรณีนี้คือ MAXSAL และ WORKDEPT. โปรดสังเกตว่าคอลัมน์ MAX(SALARY) ที่ถูกเลือกในนิพจน์ตารางที่ซ้อนกันจะต้องถูกตั้งชื่อเพื่อสามารถถูกอ้างถึงจากคำสั่ง Select ด้านนอกได้. ซึ่งทำได้โดยใช้ AS clause.

```
SELECT MGRNO, T1.DEPTNO, MAXSAL
FROM CORPDATA.DEPARTMENT T1,
     (SELECT MAX(SALARY) AS MAXSAL, WORKDEPT
      FROM CORPDATA.EMPLOYEE E1
      GROUP BY WORKDEPT) T2
WHERE T1.DEPTNO = T2.WORKDEPT
ORDER BY DEPTNO
```

ผลลัพธ์ของเคียวรีนี้คือ:

MGRNO	DEPTNO	MAXSAL
000010	A00	52750.00
000020	B01	41250.00
000030	C01	38250.00
000060	D11	32250.00
000070	D21	36170.00
000050	E01	40175.00
000090	E11	29750.00
000100	E21	26150.00

นิพจน์ตารางกลางสามารถระบุก่อน Full-Select ในคำสั่ง SELECT, คำสั่ง INSERT statement, หรือคำสั่ง CREATE VIEW . และนิพจน์แบบนี้สามารถใช้เมื่อตารางผลลัพธ์เดียวกันถูกใช้ร่วมกันใน Full-Select. นิพจน์ตารางกลางจะนำหน้าด้วยคีย์เวิร์ด WITH.

ตัวอย่างเช่น, สมมติว่าคุณต้องการตารางที่แสดงค่าต่ำสุดและค่าสูงสุดของเงินเดือนเฉลี่ยในกลุ่มแผนกที่ต้องการ. ตัวอักษรตัวแรกของหมายเลขแผนกจะมีความหมายบางอย่าง และคุณต้องการค่าต่ำสุดและสูงสุดสำหรับแผนกที่ขึ้นต้นด้วยตัวอักษร 'D' และแผนกที่ขึ้นต้นด้วยตัวอักษร 'E'. คุณสามารถใช้นิพจน์ตารางกลางเพื่อเลือกเงินเดือนเฉลี่ยสำหรับแต่ละแผนก. คุณต้องตั้งชื่อตารางลูกด้วย; ในกรณีนี้, ชื่อคือ DT, อีกครั้งหนึ่ง. คุณสามารถระบุคำสั่ง SELECT โดยใช้ WHERE clause เพื่อจำกัดการเลือกให้เฉพาะแผนกที่เริ่มต้นด้วยตัวอักษรบางค่าได้. ระบุค่าต่ำสุดและค่าสูงสุดของคอลัมน์ AVGSAL จากตารางลูก DT. ระบุ UNION เพื่อให้ได้ผลลัพธ์สำหรับตัวอักษร 'E' และผลลัพธ์สำหรับตัวอักษร 'D'.

```
WITH DT AS (SELECT E.WORKDEPT AS DEPTNO, AVG(SALARY) AS AVGSAL
            FROM CORPDATA.DEPARTMENT D , CORPDATA.EMPLOYEE E
            WHERE D.DEPTNO = E.WORKDEPT
            GROUP BY E.WORKDEPT)
SELECT 'E', MAX(AVGSAL), MIN(AVGSAL) FROM DT
WHERE DEPTNO LIKE 'E%'
UNION
SELECT 'D', MAX(AVGSAL), MIN(AVGSAL) FROM DT
WHERE DEPTNO LIKE 'D%'
```

ผลลัพธ์ของเคียวรีนี่คือ:

	MAX(AVGSAL)	MIN(AVGSAL)
E	40175.00	21020.00
D	25668.57	25147.27

สมมติว่าคุณต้องการเขียนเคียวรีที่ทำงานกับฐานข้อมูลรายการสั่งซื้อของคุณ ซึ่งจะคืนค่ารายการสูงสุด 5 รายการ (ในจำนวนรายการสั่งซื้อทั้งหมด) ภายในรายการสั่งซื้อ 1000 รายการล่าสุดจากลูกค้าที่สั่งซื้อรายการ 'XXX' ด้วย.

```
WITH X AS (SELECT ORDER_ID, CUST_ID
            FROM ORDERS
            ORDER BY ORD_DATE DESC
            FETCH FIRST 1000 ROWS ONLY),
Y AS (SELECT CUST_ID, LINE_ID, ORDER_QTY
       FROM X, ORDERLINE
       WHERE X.ORDER_ID = ORDERLINE.ORDER_ID)
SELECT LINE_ID
       FROM (SELECT LINE_ID
              FROM Y
              WHERE Y.CUST_ID IN (SELECT DISTINCT CUST_ID
                                   FROM Y
                                   WHERE LINE.ID = 'XXX' )
              GROUP BY LINE_ID
              ORDER BY SUM(ORDER_QTY) DESC)
       FETCH FIRST 5 ROWS ONLY
```

นิพจน์ตารางกลาง (X) จะคืนค่าหมายเลขรายการสั่งซื้อ 1000 รายการล่าสุด. ผลลัพธ์จะถูกเรียงลำดับโดยวันที่ในลำดับจากมากไปน้อย แล้วเฉพาะ 1000 แถวแรกเท่านั้นที่จะถูกคืนค่าเป็นตารางผลลัพธ์.

นิพจน์ตารางกลางตัวที่สอง (Y) จะทำการรวมรายการสั่งซื้อ 1000 รายการล่าสุดเข้ากับตารางบรรทัดรายการ และคืนค่าลูกค้า (สำหรับแต่ละรายการสั่งซื้อทั้ง 1000 รายการ), บรรทัดรายการ, และจำนวนของบรรทัดรายการสำหรับรายการสั่งซื้อนั้น.

ตารางลูกในคำสั่ง Select หลักจะคืนค่าบรรทัดรายการสำหรับลูกค้าที่อยู่ในรายการสั่งซื้อ 1000 อันดับแรกที่สั่งซื้อรายการ XXX. ผลลัพธ์สำหรับลูกค้าทั้งหมดที่สั่งซื้อ XXX จะถูกจัดกลุ่มโดยบรรทัดรายการ และกลุ่มจะเรียงลำดับโดยจำนวนทั้งหมดของบรรทัดรายการ.

ท้ายสุด, คำสั่ง Select ภายนอกจะเลือกเฉพาะ 5 แถวแรกจากรายการที่ถูกเรียงลำดับแล้วซึ่งตารางลูกคืนค่ากลับมา.

I การใช้เคียวรีแบบเรียกซ้ำ

I หัวข้อนี้แนะนำถึงการใช้นิพจน์ตารางแบบเรียกซ้ำ และมุมมองแบบเรียกซ้ำ.

I บางแอปพลิเคชันทำงานกับข้อมูลที่เรียกซ้ำโดยหน้าที่ของแอปพลิเคชันเอง. แอปพลิเคชัน Bill of Materials (BOM), สำหรับตัวอย่าง, จะทำงานกับการขยายของชิ้นส่วนและส่วนประกอบ. ตัวอย่างเช่น, แก้วอาจประกอบด้วยส่วนที่หนึ่งของแก้วอีก และชิ้นส่วนของขา. ส่วนที่หนึ่งของแก้วอาจประกอบด้วยที่นั่งและพนักวางแขนสองข้าง. แต่ละชิ้นส่วนเหล่านี้สามารถแตกลงเป็นชิ้นส่วนย่อยๆ จนกระทั่งได้รายการของชิ้นส่วนทั้งหมดที่ใช้ในการสร้างแก้ว. เคียวรีประเภทนี้สามารถจัดการได้โดยใช้นิพจน์ตารางแบบเรียกซ้ำ หรือมุมมองแบบเรียกซ้ำ.

I ในตัวอย่างการเดินทางโดยเครื่องบิน, เที่ยวบินและการเชื่อมต่อกับรถไฟจะถูกใช้เพื่อค้นหาเส้นทางการเดินทางระหว่างเมือง. definition ตารางและข้อมูลต่อไปนี้จะถูกใช้ในตัวอย่างนี้.

```
I CREATE TABLE FLIGHTS (DEPARTURE CHAR(20),
I                     ARRIVAL CHAR(20),
I                     CARRIER CHAR(15),
I                     FLIGHT_NUMBER CHAR(5),
I                     PRICE INT)
I
I
I INSERT INTO FLIGHTS VALUES('New York', 'Paris', 'Atlantic', '234', 400)
I INSERT INTO FLIGHTS VALUES('Chicago', 'Miami', 'NA Air', '2334', 300)
I INSERT INTO FLIGHTS VALUES('New York', 'London', 'Atlantic', '5473', 350)
I INSERT INTO FLIGHTS VALUES('London', 'Athens' , 'Mediterranean', '247', 340)
I INSERT INTO FLIGHTS VALUES('Athens', 'Nicosia' , 'Mediterranean', '2356', 280)
I INSERT INTO FLIGHTS VALUES('Paris', 'Madrid', 'Euro Air', '3256', 380)
I INSERT INTO FLIGHTS VALUES('Paris', 'Cairo' , 'Euro Air', '63', 480)
I INSERT INTO FLIGHTS VALUES('Chicago', 'Frankfurt', 'Atlantic', '37', 480)
I INSERT INTO FLIGHTS VALUES('Frankfurt', 'Moscow', 'Asia Air', '2337', 580)
I INSERT INTO FLIGHTS VALUES('Frankfurt', 'Beijing', 'Asia Air', '77', 480)
I INSERT INTO FLIGHTS VALUES('Moscow', 'Tokyo', 'Asia Air', '437', 680)
I INSERT INTO FLIGHTS VALUES('Frankfurt', 'Vienna', 'Euro Air', '59', 200)
I INSERT INTO FLIGHTS VALUES('Paris', 'Rome', 'Euro Air', '534', 340)
I INSERT INTO FLIGHTS VALUES('Miami', 'Lima', 'SA Air', '5234', 530)
I INSERT INTO FLIGHTS VALUES('New York', 'Los Angeles', 'NA Air', '84', 330)
I INSERT INTO FLIGHTS VALUES('Los Angeles', 'Tokyo', 'Pacific Air', '824', 530)
I INSERT INTO FLIGHTS VALUES('Tokyo', 'Hong Kong', 'Asia Air', '94', 330)
I INSERT INTO FLIGHTS VALUES('Washington', 'Toronto', 'NA Air', '104', 250)
I
I CREATE TABLE TRAINS(DEPARTURE CHAR(20),
I                     ARRIVAL CHAR(20),
I                     RAILLINE CHAR(15),
I                     TRAIN CHAR(5),
```

```

|           PRICE INT)
|
| INSERT INTO TRAINS VALUES('Chicago', 'Washington', 'UsTrack', '323', 90)
| INSERT INTO TRAINS VALUES('Madrid', 'Barcelona', 'EuroTrack', '5234', 60)
| INSERT INTO TRAINS VALUES('Washington', 'Boston', 'UsTrack', '232', 50)

```

ถึงตอนนี้ ตารางจะถูกสร้าง, ข้อมูลจะถูกเคียวรีเพื่อค้นหาข้อมูลเกี่ยวกับเน็ตเวิร์กสายการบิน. สมมติว่า คุณต้องการค้นหาเมืองที่คุณต้องการบินไป หากคุณเริ่มต้นที่ Chicago, และจำนวนเที่ยวบินที่จะไปเมืองนั้น. เคียวรีต่อไปนี้จะแสดงข้อมูลนี้ให้คุณ.

```

| WITH destinations (origin, departure, arrival, flight_count) AS
|   (SELECT a.departure, a.departure, a.arrival, 1
|     FROM flights a
|     WHERE a.departure = 'Chicago'
|     UNION ALL
|     SELECT r.origin, b.departure, b.arrival, r.flight_count + 1
|     FROM destinations r, flights b
|     WHERE r.arrival = b.departure)
| SELECT origin, departure, arrival, flight_count
|   FROM destinations

```

เคียวรีนี้จะส่งคืนข้อมูลต่อไปนี้:

ตารางที่ 15. ผลลัพธ์ของเคียวรีก่อนหน้านี้

ORIGIN	DEPARTURE	ARRIVAL	FLIGHT_COUNT
Chicago	Chicago	Miami	1
Chicago	Chicago	Frankfurt	1
Chicago	Miami	Lima	2
Chicago	Frankfurt	Moscow	2
Chicago	Frankfurt	Beijing	2
Chicago	Frankfurt	Vienna	2
Chicago	Moscow	Tokyo	3
Chicago	Tokyo	Hong Kong	4

เคียวรีแบบเรียกซ้ำนี้จะถูกเขียนออกเป็นสองส่วน. ส่วนแรกของนิพจน์ตารางจะเรียกว่า *intialization fullselect*. ซึ่งจะเลือกแถวแรกสำหรับชุดผลลัพธ์ของนิพจน์ตาราง. ในตัวอย่างนี้, เคียวรีจะเลือกแถวสองแถวในตารางที่ *เที่ยวบิน* ซึ่งจะบอกคุณถึงสถานที่อื่นจากชิคาโกได้โดยตรง. และยังเตรียมข้อมูลเบื้องต้นเกี่ยวกับจำนวนของเที่ยวบินสำหรับแต่ละแถวที่เลือก.

ส่วนที่สองของเคียวรีแบบเรียกซ้ำจะรวมแถวจากชุดผลลัพธ์ปัจจุบันของนิพจน์ตารางด้วยแถวอื่นจากตารางเดิม. ซึ่งเรียกว่า *iterative fullselect*. ทั้งหมดนี้คือการเรียกซ้ำ. สังเกตว่า แถวที่ถูกเลือกเป็นชุดผลลัพธ์จะถูกอ้างอิงโดยใช้ชื่อของนิพจน์ตารางเป็นชื่อตาราง และชื่อคอลัมน์ผลลัพธ์นิพจน์ตารางเป็นชื่อคอลัมน์.

ในส่วนของเคียวรีแบบเรียกซ้ำ, แถวใดๆ จากตารางต้นฉบับที่คุณสามารถรับค่าเมืองที่จะไปถึงที่เลือกไว้ก่อนหน้านี้ถูกเลือกไว้. เมืองที่จะไปถึงภายในแถวที่เลือกจะกลายเป็นเมืองที่จะออกเดินทางใหม่. แต่ละแถวจากการเลือกแบบเรียกซ้ำนี้จะเพิ่ม การนับเที่ยวบินไปยังปลายทางด้วยเที่ยวบินอื่น. แถวใหม่เหล่านี้จะถูกเพิ่มไปยังชุดผลลัพธ์นิพจน์ตาราง, ซึ่งจะส่งไปยัง iterative

| fullselect เพื่อสร้างแถวของชุดผลลัพธ์เพิ่มเติม. ในข้อมูลที่ได้จากผลลัพธ์สุดท้าย, คุณจะเห็นว่าจำนวนของเที่ยวบินทั้งหมด
| คือจำนวนของการรวมแบบเรียกซ้ำทั้งหมด (บวก 1) เพื่อที่จะได้เมืองที่จะไปถึง.

| มุมมองแบบเรียกซ้ำคล้ายกับนิพจน์ตารางแบบเรียกซ้ำ. คุณสามารถเขียนนิพจน์ตารางก่อนหน้านี้เป็นมุมมองแบบเรียกซ้ำ
| ได้ดังนี้:

```
| CREATE VIEW destinations (origin, departure, arrival, flight_count) AS  
|     SELECT departure, departure, arrival, 1  
|         FROM flights  
|         WHERE departure = 'Chicago'  
|     UNION ALL  
|     SELECT r.origin, b.departure, b.arrival, r.flight_count + 1  
|         FROM destinations r, flights b  
|         WHERE r.arrival = b.departure)
```

| ส่วนของ definition มุมมองแบบ interactive fullselect นี้อ้างอิงถึงตัวของมุมมองเอง. การเลือกจากมุมมองนี้จะส่งคืนแถวที่
| เหมือนกันกับที่คุณได้มาจากนิพจน์ตารางแบบเรียกซ้ำก่อนหน้านี้.

| ตัวอย่าง: จุดเริ่มต้นของเมืองสองเมือง

| ถึงตอนนี้, เพื่อให้เคียวรีซับซ้อนยิ่งขึ้น, สมมติว่า คุณกำลังบินจาก Chicago หรือ New York, และคุณต้องการรู้สถานที่ที่คุณจะ
| ไป และค่าใช้จ่ายที่เกิดขึ้น.

```
| WITH destinations (departure, arrival, connections, cost) AS  
|     (SELECT a.departure, a.arrival, 0, price  
|        FROM flights a  
|        WHERE a.departure = 'Chicago' OR  
|              a.departure = 'New York'  
|     UNION ALL  
|     SELECT r.departure, b.arrival, r.connections + 1,  
|           r.cost + b.price  
|        FROM destinations r, flights b  
|        WHERE r.arrival = b.departure)  
| SELECT departure, arrival, connections, cost  
|     FROM destinations
```

| เคียวรีนี้จะส่งคืนข้อมูลต่อไปนี้:

| ตารางที่ 16.

DEPARTURE	ARRIVAL	CONNECTIONS	COST
Chicago	Miami	0	300
Chicago	Frankfurt	0	480
New York	Paris	0	400
New York	London	0	350
New York	Los Angeles	0	330
Chicago	Lima	1	830
Chicago	Moscow	1	1,060

| ตารางที่ 16. (ต่อ)

DEPARTURE	ARRIVAL	CONNECTIONS	COST
Chicago	Beijing	1	960
Chicago	Vienna	1	680
New York	Madrid	1	780
New York	Cairo	1	880
New York	Rome	1	740
New York	Athens	1	690
New York	Tokyo	1	860
Chicago	Tokyo	2	1,740
New York	Nicosia	2	970
New York	Hong Kong	2	1,190
Chicago	Hong Kong	3	2,070

| สำหรับแต่ละแถวที่ถูกส่งคืนค่า, ผลลัพธ์จะแสดงจุดเริ่มต้นของเมืองต้นทาง และเมืองปลายทางสุดท้าย. ซึ่งจะนับจำนวนของ
 | การเชื่อมต่อที่จำเป็นแทนที่จะเป็นจำนวนของเที่ยวบินทั้งหมด และเพิ่มข้อมูลค่าใช้จ่ายทั้งหมดสำหรับแต่ละเที่ยวบิน.

| ตัวอย่าง: สองตารางที่ใช้สำหรับการเรียกซ้ำ

| ถึงตอนนี้, สมมติว่า คุณเริ่มต้นจาก Chicago แต่ต้องเดินทางโดยรถไฟ แล้วเดินทางต่อด้วยที่

| เคียวรี่ต่อไปนี้จะส่งคืนข้อมูลนี้:

```

| WITH destinations (departure, arrival, connections, flights, trains, cost) AS
| (SELECT f.departure, f.arrival, 0, 1, 0, price
|   FROM flights f
|   WHERE f.departure = 'Chicago'
| UNION ALL
| SELECT t.departure, t.arrival, 0, 0, 1, price
|   FROM trains t
|   WHERE t.departure = 'Chicago'
| UNION ALL
| SELECT r.departure, b.arrival, r.connections + 1 , r.flights + 1, r.trains,
|   r.cost + b.price
|   FROM destinations r, flights b
|   WHERE r.arrival = b.departure
| UNION ALL
| SELECT r.departure, c.arrival, r.connections + 1 ,
|   r.flights, r.trains + 1, r.cost + c.price
|   FROM destinations r, trains c
|   WHERE r.arrival = c.departure)
| SELECT departure, arrival, connections, flights, trains, cost
|   FROM destinations
  
```

| เคียวนี้จะส่งคืนข้อมูลต่อไปนี้:

| ตารางที่ 17. ผลลัพธ์ของเคียวก่อนหน้า

DEPARTURE	ARRIVAL	CONNECTIONS	FLIGHTS	TRAINS	COST
Chicago	Miami	0	1	0	300
Chicago	Frankfurt	0	1	0	480
Chicago	Washington	0	0	1	90
Chicago	Lima	1	2	0	830
Chicago	Moscow	1	2	0	1,060
Chicago	Beijing	1	2	0	960
Chicago	Vienna	1	2	0	680
Chicago	Toronto	1	1	1	340
Chicago	Boston	1	0	2	140
Chicago	Tokyo	2	3	0	1,740
Chicago	Hong Kong	3	4	0	2,070

| ในตัวอย่างนี้, มีสองส่วนของนิพจน์ตารางที่กำหนดค่าเริ่มต้นให้กับเคียวรี: ส่วนแรกสำหรับเที่ยวบิน และอีกหนึ่งส่วนสำหรับ
| รถไฟ. สำหรับแต่ละแถวของผลลัพธ์, มีสองการอ้างอิงแบบเรียกซ้ำเพื่อดึงข้อมูลจากตำแหน่งที่ไปถึงก่อนหน้านี้ให้เป็นสถาน
| ที่ปลายทางถัดไปที่เป็นไปได้: การอ้างอิงแรกสำหรับการเดินทางต่อด้วยเครื่องบิน, การอ้างอิงถัดมาสำหรับการเดินทางต่อ
| ด้วยรถไฟ. ในผลลัพธ์สุดท้าย, คุณจะเห็นจำนวนของการเชื่อมต่อที่จำเป็น และจำนวนของสายการบินหรือเที่ยวรถไฟที่
| สามารถไปได้..

| ตัวอย่าง: อีพซัน DEPTH FIRST และ BREADTH FIRST

| ตัวอย่างสองตัวอย่างนี้ จะแสดงลำดับแถวของชุดผลลัพธ์ที่แตกต่างกัน ขึ้นอยู่กับการเรียกซ้ำที่ถูกประมวลผลด้วย depth first
| หรือ breadth first.

| **หมายเหตุ:** search clause ไม่สนับสนุนสำหรับมุมมองแบบเรียกซ้ำ. คุณสามารถกำหนดมุมมองที่มีนิพจน์ตารางแบบเรียกซ้ำ
| เพื่อเรียกใช้ฟังก์ชันนี้.

| อีพซันที่จะเป็นตัวกำหนดผลลัพธ์โดยใช้ breadth first หรือ depth first คือการเรียงลำดับความสัมพันธ์แบบเรียกซ้ำ ขึ้นอยู่กับ
| คอลัมน์การรวมแบบเรียกซ้ำที่ระบุสำหรับ SEARCH BY clause. เมื่อการเรียกซ้ำถูกจัดการแบบ breadth first, ลูกทั้งหมดจะ
| ถูกประมวลผลครั้งแรก, แล้วตามด้วยหลาน, แล้วตามด้วยเหลน. เมื่อการเรียกซ้ำถูกจัดการแบบ depth first, กลุ่มของเรียก
| คอร์ตที่สัมพันธ์กันของเด็กหนึ่งคนแบบเรียกซ้ำแบบเต็มจะถูกประมวลผลก่อนที่จะไปยังเด็กคนต่อไป.

| ในทั้งสองกรณี, คุณระบุชื่อคอลัมน์เพิ่มเติม ซึ่งจะถูกใช้โดยกระบวนการเรียกซ้ำ เพื่อเก็บแทร็กของการเรียงลำดับแบบ depth
| first หรือ breadth first. คอลัมน์นี้ต้องใช้ใน ORDER BY clause ของเคียวรีแบบ outer เพื่อดึงแถวกลับเข้าในลำดับที่ระบุ. หาก
| คอลัมน์นี้ไม่ถูกใช้ใน ORDER BY, อีพซันการประมวลผลแบบ DEPTH FIRST หรือ BREADTH FIRST จะถูกละเลย.

| การเลือกว่า คอลัมน์ใดจะใช้สำหรับคอลัมน์ SEARCH BY เป็นสิ่งที่สำคัญ. เพื่อให้ผลลัพธ์มีความหมาย, จึงต้องเป็นคอลัมน์ที่ถูกใช้ใน iterative fullselect เพื่อรวมจาก initialization fullselect. ในตัวอย่างนี้, ARRIVAL คือคอลัมน์ที่ใช้.

| เคียวรีต่อไปนี้จะส่งคืนข้อมูลนี้:

```
| WITH destinations (departure, arrival, connections, cost) AS
|   (SELECT f.departure, f.arrival, 0, price
|     FROM flights f
|     WHERE f.departure = 'Chicago'
|   UNION ALL
|     SELECT r.departure, b.arrival, r.connections + 1,
|         r.cost + b.price
|     FROM destinations r, flights b
|     WHERE r.arrival = b.departure)
| SEARCH DEPTH FIRST BY arrival SET ordcol
| SELECT *
|   FROM destinations
|   ORDER BY ordcol
```

| เคียวรีนี้จะส่งคืนข้อมูลต่อไปนี้:

| ตารางที่ 18. ผลลัพธ์ของเคียวรีก่อนหน้านี้

DEPARTURE	ARRIVAL	CONNECTIONS	COST
Chicago	Miami	0	300
Chicago	Lima	1	830
Chicago	Frankfurt	0	480
Chicago	Moscow	1	1,060
Chicago	Tokyo	2	1,740
Chicago	Hong Kong	3	2,070
Chicago	Beijing	1	960
Chicago	Vienna	1	680

| ในข้อมูลผลลัพธ์นี้, คุณจะเห็นสถานที่ปลายทางทั้งหมดที่สร้างจากแถว Chicago-to-Miami ซึ่งถูกแสดงก่อนสถานที่ปลายทางจากแถว Chicago-to-Frankfort.

| ถัดไป, คุณสามารถรันเคียวรีเดียวกันนี้แต่ร้องขอให้มีการเรียงลำดับผลลัพธ์แบบ breadth first.

```
| WITH destinations (departure, arrival, connections, cost) AS
|   (SELECT f.departure, f.arrival, 0, price
|     FROM flights f
|     WHERE f.departure='Chicago'
|   UNION ALL
|     SELECT r.departure, b.arrival, r.connections + 1,
|         r.cost + b.price
|     FROM destinations r, flights b
|     WHERE r.arrival = b.departure)
```

```

| SEARCH BREADTH FIRST BY arrival SET ordcol
| SELECT *
| FROM destinations
| ORDER BY ordcol

```

| เคียวรีนี้จะส่งคืนข้อมูลต่อไปนี้:

| ตารางที่ 19. ผลลัพธ์ของเคียวรีก่อนหน้า

DEPARTURE	ARRIVAL	CONNECTIONS	COST
Chicago	Miami	0	300
Chicago	Frankfurt	0	480
Chicago	Lima	1	830
Chicago	Moscow	1	1,060
Chicago	Beijing	1	960
Chicago	Vienna	1	680
Chicago	Tokyo	2	1,740
Chicago	Hong Kong	3	2,070

| ในข้อมูลผลลัพธ์นี้, คุณจะเห็นการเชื่อมต่อโดยตรงทั้งหมดจากชิคาโกที่ถูกแสดงไว้ก่อนการเชื่อมต่อเที่ยวบิน. ข้อมูลจะ
| เหมือนกับกับผลลัพธ์ที่ได้จากเคียวรีก่อนหน้า, แต่อยู่ในลำดับการเรียงแบบ breadth first.

| ตัวอย่าง: การวน

| ดีสำหรับการประมวลผลแบบเรียกซ้ำ, ไม่ว่าจะเป็น algorithm โปรแกรมมิ่งแบบเรียกซ้ำ หรือการเคียวรีข้อมูลแบบเรียกซ้ำ,
| คือการเรียกซ้ำต้องถูกจำกัด. ถ้าไม่, คุณจะไม่จบจากการวนซ้ำ. อ็อพชัน CYCLE จะอนุญาตให้คุณป้องกันการวนซ้ำข้อมูล.
| ไม่เพียงแต่การยกเลิกการวนซ้ำ แต่จะอนุญาตให้คุณออกเอาต์พุตตัวบ่งชี้การมาร์กให้วนรอบ ซึ่งอาจนำคุณไปยังข้อมูลที่วน
| ซ้ำ.

| **หมายเหตุ:** cycle clause ไม่สนับสนุนสำหรับมุมมองแบบเรียกซ้ำ. คุณสามารถกำหนดมุมมองที่มีนิพจน์ตารางแบบเรียกซ้ำ
| เพื่อเรียกใช้ฟังก์ชันนี้.

| สำหรับตัวอย่างสุดท้าย, สมมติว่า เรามีข้อมูลแบบวัฏจักร. โดยการเพิ่มแถวเข้าไปหนึ่งแถวในตาราง, ซึ่งจะเป็นเที่ยวบินจาก
| Cairo ไปยัง Paris และจาก Paris ไปยัง Cairo. ไม่มีเหตุผลสำหรับการมีข้อมูลวัฏจักรแบบนี้, ซึ่งง่ายต่อการเคียวรีโดยที่จะไปยัง
| การประมวลผลการวนรอบแบบไม่รู้จบ.

| เคียวรีต่อไปนี้จะส่งคืนข้อมูลนี้:

```

| INSERT INTO FLIGHTS VALUES('Cairo', 'Paris', 'Euro Air', '1134', 440)
|
|
| WITH destinations (departure, arrival, connections, cost, itinerary) AS
| (SELECT f.departure, f.arrival, 1, price,
| CAST(f.departure CONCAT f.arrival AS VARCHAR(2000))
| FROM flights f

```

```

|         WHERE f.departure = 'New York'
|     UNION ALL
|     SELECT r.departure, b.arrival, r.connections + 1 ,
|           r.cost + b.price, CAST(r.itinerary CONCAT b.arrival AS VARCHAR(2000))
|         FROM destinations r, flights b
|         WHERE r.arrival = b.departure)
|     CYCLE arrival SET cyclic_data TO '1' DEFAULT '0'
| SELECT departure, arrival, itinerary, cyclic_data
|     FROM destinations
|     ORDER BY cyclic_data

```

เคียวรีนี้จะส่งคืนข้อมูลต่อไปนี้:

ตารางที่ 20. ผลลัพธ์ของเคียวรีก่อนหน้า

DEPARTURE	ARRIVAL	ITINERARY	CYCLIC_DATA
New York	Paris	New York Paris	0
New York	London	New York London	0
New York	Los Angeles	New York Los Angeles	0
New York	Madrid	New York Paris Madrid	0
New York	Cairo	New York Paris Cairo	0
New York	Rome	New York Paris Rome	0
New York	Athens	New York London Athens	0
New York	Tokyo	New York Los Angeles Tokyo	0
New York	Paris	New York Paris Cairo Paris	1
New York	Nicosia	New York London Athens Nicosia	0
New York	Hong Kong	New York Los Angeles Tokyo Hong Kong	0

ในตัวอย่างนี้, คอลัมน์ ARRIVAL จะถูกกำหนดใน CYCLE clause เป็นคอลัมน์ที่ใช้สำหรับตรวจการวนของข้อมูล. เมื่อพบการวนของข้อมูล, คอลัมน์พิเศษ, ในกรณีนี้คือ CYCLIC_DATA, จะถูกตั้งค่าอีกขระเป็น '1' สำหรับแถวที่วนในชุดผลลัพธ์. แถวอื่นๆ ทั้งหมดจะยังคงมีค่าดีฟอลต์เป็น '0'. เมื่อพบการวนบนคอลัมน์ ARRIVAL, การประมวลผลจะไม่ดำเนินการใดๆ ต่อ ดังนั้นการวนรอบแบบไม่รู้จบจะไม่เกิดขึ้น. เมื่อต้องการดูหากข้อมูลมีการอ้างอิงแบบวนรอบ, คอลัมน์ CYCLIC_DATA จะถูกอ้างอิงในเคียวรีแบบ outer.

การใช้เคียวรี UNION เพื่อรวมการเลือกย่อย

การใช้เคียวรี UNION, คุณสามารถรวมการเลือกย่อยมากกว่าสองได้ เพื่อสร้าง fullselect.

เมื่อ SQL เจอเคียวรี UNION, SQL จะดำเนินการกับการเลือกย่อยแต่ละตัวเพื่อสร้างตารางผลลัพธ์ชั่วคราว, จากนั้นจึงรวมตารางผลลัพธ์ชั่วคราวของการเลือกย่อยแต่ละครั้ง และลบแถวที่ซ้ำกันเพื่อสร้างตารางผลลัพธ์ที่ถูกรวมแล้ว. คุณสามารถใช้ clause และเทคนิคได้หลากหลาย เมื่อเขียนโค้ดคำสั่ง select.

คุณสามารถใช้ UNION เพื่อลบรายการที่ซ้ำกันเมื่อรวมรายการของค่าที่รับมาจากหลายตาราง. ตัวอย่างเช่น, คุณสามารถรับรายการรวมของหมายเลขพนักงานที่มี:

- คนในแผนก D11
- คนที่ได้รับมอบงานในโครงการ MA2112, MA2113, และ AD3111

รายการรวมจะรับค่ามาจากตารางสองตารางและเก็บค่าที่ไม่ซ้ำกัน. เมื่อต้องการทำสิ่งนี้, ให้ระบุ:

```
SELECT EMPNO
  FROM CORPDATA.EMPLOYEE
 WHERE WORKDEPT = 'D11'
 UNION
 SELECT EMPNO
  FROM CORPDATA.EMPPROJACT
 WHERE PROJNO = 'MA2112' OR
        PROJNO = 'MA2113' OR
        PROJNO = 'AD3111'
 ORDER BY EMPNO
```

เมื่อต้องการเข้าใจผลลัพธ์จากคำสั่ง SQL ยิ่งขึ้น, ลองจินตนาการว่า SQL จะทำงานผ่านกระบวนการดังต่อไปนี้:

ขั้นตอน 1. SQL จะดำเนินการกับคำสั่ง SELECT แรก:

```
SELECT EMPNO
  FROM CORPDATA.EMPLOYEE
 WHERE WORKDEPT = 'D11'
```

ซึ่งผลลัพธ์ในตารางผลลัพธ์ชั่วคราวคือ:

EMPNO from CORPDATA.EMPLOYEE

000060

000150

000160

000170

000180

000190

000200

000210

000220

200170

200220

ขั้นตอน 2. SQL จะดำเนินการกับคำสั่ง SELECT ที่สอง:

```

SELECT EMPNO
      FROM CORPDATA.EMPPROJACT
      WHERE PROJNO='MA2112' OR
             PROJNO= 'MA2113' OR
             PROJNO= 'AD3111'

```

ซึ่งผลลัพธ์ในตารางผลลัพธ์ข้างครวคือ:

EMPNO from CORPDATA.EMPPROJACT

000230

000230

000240

000230

000230

000240

000230

000150

000170

000190

000170

000190

000150

000160

000180

000170

000210

000210

ขั้นตอน 3. SQL จะรวมตารางผลลัพธ์ข้างครวทั้งสองเข้าด้วยกัน, เอาแถวที่ซ้ำกันออก, และเรียงลำดับผลลัพธ์:

```

SELECT EMPNO
      FROM CORPDATA.EMPLOYEE
      WHERE WORKDEPT = 'D11'
UNION
SELECT EMPNO
      FROM CORPDATA.EMPPROJACT
      WHERE PROJNO='MA2112' OR
             PROJNO= 'MA2113' OR
             PROJNO= 'AD3111'
ORDER BY EMPNO

```

ซึ่งผลลัพธ์ในตารางผลลัพธ์รวมจะมีค่าในลำดับจากน้อยไปมาก:

EMPNO

000060

000150

000160

000170

000180

000190

000200

000210

000220

000230

000240

200170

200220

เมื่อคุณใช้ UNION:

- ทุก ORDER BY clause ต้องปรากฏอยู่หลังการเลือกย่อยสุดท้ายที่เป็นส่วนหนึ่งของ Union. ในตัวอย่างนี้, ผลลัพธ์จะเรียงลำดับตามคอลัมน์ที่ถูกเลือกตัวแรก, *EMPNO*. ORDER BY clause จะระบุว่าตารางผลลัพธ์รวมจะอยู่ในรูปแบบที่มีลำดับการเรียง. ORDER BY จะไม่อนุญาตให้ใช้ในวิว.
- ชื่อสามารถถูกระบุในอนุประโยค ORDER BY ได้ถ้าคอลัมน์ผลลัพธ์ถูกตั้งชื่อ. คอลัมน์ผลลัพธ์จะถูกตั้งชื่อถ้าคอลัมน์ที่สัมพันธ์กันในแต่ละคำสั่ง Select ที่ถูกรวมมีชื่อเดียวกัน. AS clause สามารถใช้เพื่อกำหนดชื่อให้กับคอลัมน์ในรายการที่เลือกได้.

```
SELECT A + B AS X ...  
UNION  
SELECT X ... ORDER BY X
```

ถ้าคอลัมน์ผลลัพธ์ไม่ได้ถูกตั้งชื่อ, ให้ใช้จำนวนเต็มบวกเพื่อเรียงลำดับผลลัพธ์. หมายเลขจะอ้างอิงไปยังตำแหน่งของนิพจน์ในรายการของนิพจน์ที่คุณรวมเข้าไปในการเลือกย่อยของคุณ.

```
SELECT A + B ...  
UNION  
SELECT X ... ORDER BY 1
```

เมื่อต้องการระบุว่าการเลือกย่อยแต่ละแถวมาจากไหน, คุณสามารถรวมค่าคงที่เข้าไปที่ท้ายสุดของรายการเลือกสำหรับแต่ละการเลือกย่อยใน Union. เมื่อ SQL คืนค่าผลลัพธ์ของคุณกลับมา, คอลัมน์สุดท้ายจะเก็บค่าคงที่สำหรับการเลือกย่อยที่เป็นต้นฉบับของแถวนั้น. ตัวอย่างเช่น, คุณสามารถระบุ:

```
SELECT A, B, 'A1' ...
UNION
SELECT X, Y, 'B2'...
```

เมื่อแถวถูกส่งคืนค่ากลับมา, ค่าดังกล่าวจะรวมค่า (A1 หรือ B2) เพื่อบ่งชี้ถึงตารางที่มีต้นฉบับของค่าแถว. ถ้าชื่อคอลัมน์ใน Union ต่างกัน, SQL จะใช้ชุดของชื่อคอลัมน์ที่ระบุในการเลือกย่อยเมื่อ SQL แบบโต้ตอบแสดงหรือพิมพ์ผลลัพธ์, หรือใน SQLDA ที่เป็นผลลัพธ์จากการดำเนินการคำสั่ง DESCRIBE ของ SQL.

หมายเหตุ: การเรียงลำดับจะถูกใช้หลังจากฟิลเตอร์ระหว่าง UNION ถูกทำให้เข้ากันได้แล้ว. การเรียงลำดับถูกใช้สำหรับการดำเนินการเฉพาะที่เกิดขึ้นโดยตรงขณะดำเนินการ UNION.

หลักการที่เกี่ยวข้อง

“ลำดับการเรียง และ normalization ใน SQL” ในหน้า 119

ลำดับการจัดเรียงกำหนดความสัมพันธ์ของอักขระในชุดอักขระ เมื่อมีการเปรียบเทียบหรือจัดลำดับ. Normalization อนุญาตให้คุณเปรียบเทียบสตริงที่มีอักขระแบบผสม.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การสร้างและใช้มุมมอง” ในหน้า 35

สามารถใช้มุมมองเพื่อเข้าใช้งานข้อมูลในตารางหนึ่งตารางหรือมากกว่าหรือมุมมองหนึ่งมุมมองหรือมากกว่าได้. คุณสามารถสร้างมุมมองได้โดยใช้คำสั่ง SELECT.

การระบุ UNION ALL:

ถ้าคุณต้องการเก็บค่าที่ซ้ำกันในผลลัพธ์ของ UNION, ให้ระบุ UNION ALL แทนที่จะระบุแค่ UNION.

โดยใช้ขั้นตอนและตัวอย่างเดียวกับ UNION:

ขั้นตอน 3. SQL จะรวมตารางผลลัพธ์ชั่วคราวทั้งสอง:

```
SELECT EMPNO
  FROM CORPDATA.EMPLOYEE
  WHERE WORKDEPT = 'D11'
UNION ALL
SELECT EMPNO
  FROM CORPDATA.EMPPROJECT
  WHERE PROJNO='MA2112' OR
         PROJNO= 'MA2113' OR
         PROJNO= 'AD3111'
  ORDER BY EMPNO
```

ผลลัพธ์ในตารางผลลัพธ์ที่มีการเรียงลำดับซึ่งรวมข้อมูลที่ซ้ำด้วย:

EMPNO

000060

000150

000150

000150

EMPNO

000160

000160

000170

000170

000170

000170

000180

000180

000190

000190

000190

000200

000210

000210

000210

000220

000230

000230

000230

000230

000230

000240

000240

200170

200220

การดำเนินการ UNION ALL ทำให้เกิดการเชื่อมโยง, ตัวอย่างเช่น:

```
(SELECT PROJNO FROM CORPDATA.PROJECT  
UNION ALL  
SELECT PROJNO FROM CORPDATA.PROJECT)  
UNION ALL  
SELECT PROJNO FROM CORPDATA.EMPPROJECT
```

คำสั่งนี้ยังสามารถเขียนได้เป็น:

```
SELECT PROJNO FROM CORPDATA.PROJECT
UNION ALL
(SELECT PROJNO FROM CORPDATA.PROJECT
UNION ALL
SELECT PROJNO FROM CORPDATA.EMPPROJECT)
```

เมื่อคุณรวม UNION ALL เข้าไปในคำสั่ง SQL เดียวกับตัวดำเนินการ UNION, อย่างไรก็ตาม, ผลลัพธ์ของการดำเนินการจะขึ้นอยู่กับลำดับของการประเมินผล. เมื่อไม่มีวงเล็บ, การประเมินผลจะทำจากซ้ายไปขวา. หากมีวงเล็บ, การเลือกย่อยที่อยู่ในวงเล็บจะถูกประเมินผลก่อน, ตามด้วย, จากซ้ายไปขวา, โดยส่วนอื่นของคำสั่ง.

การใช้คีย์เวิร์ด EXCEPT

คีย์เวิร์ด EXCEPT จะส่งค่ากลับมาโดยนำผลที่ได้จากการเลือกย่อยชุดแรกลบด้วยแถวที่ซ้ำกันในผลของการเลือกย่อยชุดที่สอง.

สมมติว่าคุณต้องการรายการหมายเลขพนักงานที่ประกอบด้วย:

- คนในแผนก D11
- ยกเว้น ผู้ที่ได้รับมอบหมายงานในโครงการ MA2112, MA2113, และ AD3111

ผลจากเคียวรี่จะได้เป็นพนักงานทั้งหมดที่อยู่ในแผนก D11 ผู้ที่ *ไม่ได้* ทำงานในโครงการ MA2112, MA2113, และ AD3111.

เมื่อต้องการทำเช่นนี้, ให้ระบุ:

```
SELECT EMPNO
  FROM CORPDATA.EMPLOYEE
 WHERE WORKDEPT = 'D11'
EXCEPT
SELECT EMPNO
  FROM CORPDATA.EMPPROJECT
 WHERE PROJNO = 'MA2112' OR
        PROJNO = 'MA2113' OR
        PROJNO = 'AD3111'
 ORDER BY EMPNO
```

เมื่อต้องการเข้าใจผลลัพธ์จากคำสั่ง SQL ยิ่งขึ้น, ลองจินตนาการว่า SQL จะทำงานผ่านกระบวนการดังต่อไปนี้:

ขั้นตอน 1. SQL จะดำเนินการกับคำสั่ง SELECT แรก:

```
SELECT EMPNO
  FROM CORPDATA.EMPLOYEE
 WHERE WORKDEPT = 'D11'
```

ซึ่งผลลัพธ์ในตารางผลลัพธ์ชั่วคราวคือ:

EMPNO from CORPDATA.EMPLOYEE

000060

000150

EMPNO from CORPDATA.EMPLOYEE

000160

000170

000180

000190

000200

000210

000220

200170

200220

ขั้นตอน 2. SQL จะดำเนินการกับคำสั่ง SELECT ที่สอง:

```
SELECT EMPNO
      FROM CORPDATA.EMPPROJECT
      WHERE PROJNO='MA2112' OR
             PROJNO= 'MA2113' OR
             PROJNO= 'AD3111'
```

ซึ่งผลลัพธ์ในตารางผลลัพธ์ชั่วคราวคือ:

EMPNO from CORPDATA.EMPPROJECT

000230

000230

000240

000230

000230

000240

000230

000150

000170

000190

000170

000190

000150

EMPNO from CORPDATA.EMPPROJACT

000160

000180

000170

000210

000210

ขั้นตอน 3. SQL จะนำตารางผลลัพธ์ชั่วคราวตารางแรก, ลบแถวทั้งหมดที่มีเหมือนกับตารางผลลัพธ์ชั่วคราวที่สอง, ลบแถวที่ซ้ำกันออก, และเรียงลำดับผลลัพธ์:

```
SELECT EMPNO
      FROM CORPDATA.EMPLOYEE
      WHERE WORKDEPT = 'D11'
EXCEPT
SELECT EMPNO
      FROM CORPDATA.EMPPROJACT
      WHERE PROJNO='MA2112' OR
             PROJNO= 'MA2113' OR
             PROJNO= 'AD3111'
ORDER BY EMPNO
```

ซึ่งผลลัพธ์ในตารางผลลัพธ์รวมจะมีค่าในลำดับจากน้อยไปมาก:

EMPNO

000060

000200

000220

200170

200220

การใช้คีย์เวิร์ด INTERSECT

คีย์เวิร์ด INTERSECT จะส่งค่าผลลัพธ์รวมที่ประกอบด้วยแถวที่ปรากฏอยู่ในผลลัพธ์ทั้งสองชุด.

สมมติว่าคุณต้องการรายการหมายเลขพนักงานที่ประกอบด้วย:

- คนในแผนก D11
- คนที่ได้รับมอบงานในโครงการ MA2112, MA2113, และ AD3111

INTERSECT จะให้ผลเป็นรายการหมายเลขพนักงานทั้งหมดที่ปรากฏอยู่ในผลลัพธ์ทั้งสองอัน. หรือพูดอีกอย่างก็คือ, ผลจากเคียวรี่จะได้เป็นพนักงานทั้งหมดที่อยู่ในแผนก D11 ที่ทำงานอยู่ในโครงการ MA2112, MA2113, และ AD3111 ด้วย.

เมื่อต้องการทำสิ่งนี้, ให้ระบุ:


```

SELECT EMPNO
  FROM CORPDATA.EMPLOYEE
 WHERE WORKDEPT = 'D11'
INTERSECT
SELECT EMPNO
  FROM CORPDATA.EMPPROJACT
 WHERE PROJNO = 'MA2112' OR
        PROJNO = 'MA2113' OR
        PROJNO = 'AD3111'
 ORDER BY EMPNO

```

เมื่อต้องการเข้าใจผลลัพธ์จากคำสั่ง SQL ยิ่งขึ้น, ลองจินตนาการว่า SQL จะทำงานผ่านกระบวนการดังต่อไปนี้:

ขั้นตอน 1. SQL จะดำเนินการกับคำสั่ง SELECT แรก:

```

SELECT EMPNO
  FROM CORPDATA.EMPLOYEE
 WHERE WORKDEPT = 'D11'

```

ซึ่งผลลัพธ์ในตารางผลลัพธ์ชั่วคราวคือ:

EMPNO from CORPDATA.EMPLOYEE

000060

000150

000160

000170

000180

000190

000200

000210

000220

200170

200220

ขั้นตอน 2. SQL จะดำเนินการกับคำสั่ง SELECT ที่สอง:

```

SELECT EMPNO
  FROM CORPDATA.EMPPROJACT
 WHERE PROJNO='MA2112' OR
        PROJNO= 'MA2113' OR
        PROJNO= 'AD3111'

```

ซึ่งผลลัพธ์ในตารางผลลัพธ์ชั่วคราวคือ:

EMPNO from CORPDATA.EMPPROJACT

000230

000230

000240

000230

000230

000240

000230

000150

000170

000190

000170

000190

000150

000160

000180

000170

000210

000210

ขั้นตอน 3. SQL จะนำตารางผลลัพธ์ชั่วคราวตารางแรก, เปรียบเทียบกับตารางผลลัพธ์ชั่วคราวตารางที่สอง, และส่งคืนแถวที่ปรากฏในตารางทั้งสองยกเว้นแถวที่ซ้ำกัน, และเรียงลำดับผลลัพธ์.

```
SELECT EMPNO
  FROM CORPDATA.EMPLOYEE
 WHERE WORKDEPT = 'D11'
INTERSECT
SELECT EMPNO
  FROM CORPDATA.EMPPROJACT
 WHERE PROJNO='MA2112' OR
        PROJNO= 'MA2113' OR
        PROJNO= 'AD3111'
ORDER BY EMPNO
```

ซึ่งผลลัพธ์ในตารางผลลัพธ์รวมจะมีค่าในลำดับจากน้อยไปมาก:

EMPNO

000150

000160

000170

000180

000190

000210

ข้อผิดพลาดในการดึงข้อมูล

บางครั้ง เมื่อคุณใช้คำสั่ง, คุณอาจพบกับข้อผิดพลาด.

ถ้า SQL พบว่า คอลัมน์อักขระ หรือคอลัมน์กราฟิกที่มีความยาวเกินกว่าที่จะใส่เข้าไปในตัวแปรโฮสต์แล้ว, SQL จะทำสิ่งต่อไปนี้:

- ตัดข้อมูลขณะที่กำหนดค่าให้กับตัวแปรโฮสต์.
- ตั้งค่า SQLWARN0 และ SQLWARN1 ใน SQLCA ให้เป็นค่า 'W' หรือตั้งค่า RETURNED_SQLSTATE ให้เป็น '01004' ในพื้นที่การวินิจฉัยของ SQL.
- ตั้งค่าตัวแปร Indicator, ถ้าถูกเติมไว้, ให้เป็นความยาวของค่าก่อนที่จะถูกตัด.

ถ้า SQL เจอข้อผิดพลาดในการจับคู่ข้อมูลในขณะที่รันคำสั่งอยู่, SQL จะดำเนินการอย่างหนึ่งอย่างใดต่อไปนี้:

- ถ้าข้อผิดพลาดเกิดขึ้นในนิพจน์รายการ SELECT และตัวแปร Indicator ถูกเตรียมไว้สำหรับนิพจน์ที่มีข้อผิดพลาด:
 - SQL จะคืนค่า -2 สำหรับตัวแปรตัวบ่งชี้ที่สัมพันธ์กับนิพจน์ที่มีข้อผิดพลาด.
 - SQL คืนค่าข้อมูลที่ถูกต้องทั้งหมดสำหรับแถวนั้น.
 - SQL คืนค่า SQLCODE เป็นบวก.
- ถ้าไม่มีตัวแปร Indicator มาให้, SQL จะคืนค่า SQLCODE ที่เป็นค่าลบ.

ข้อผิดพลาดในการจับคู่ข้อมูลประกอบด้วย:

- +138 - อากิวเมนต์ของฟังก์ชันที่ใช้ตัดสตริงมีค่าไม่ถูกต้อง.
- +180 - ซินแทกซ์สำหรับสตริงที่แทนวัน, เวลา, หรือ timestamp มีค่าไม่ถูกต้อง.
- +181 - สตริงที่เป็นตัวแทนวัน, เวลา, หรือ timestamp ไม่ใช่ค่าที่ถูกต้อง.
- +183 - ผลลัพธ์ที่ไม่ถูกต้องจากนิพจน์วัน/เวลา, วันหรือ timestamp ที่ได้ไม่ได้อยู่ในช่วงวันหรือ timestamp ที่ถูกต้อง.
- +191 - รูปแบบข้อมูล MIXED ไม่ถูกต้อง.
- +304 - ข้อผิดพลาดในการแปลงตัวเลข (ตัวอย่างเช่น, โอเวอร์โฟลว์, อันเดอร์โฟลว์, หรือหารด้วยศูนย์).
- +331 - ตัวอักขระไม่สามารถถูกแปลงได้.
- +420 - ตัวอักขระในอากิวเมนต์ของ CAST มีค่าไม่ถูกต้อง.
- +802 - การแปลงข้อมูลหรือการจับคู่ข้อมูลมีข้อผิดพลาด.

สำหรับการแปลงข้อมูลมีข้อผิดพลาด, SQLCA จะรายงานเฉพาะข้อผิดพลาดสุดท้ายที่ตรวจเจอ. ตัวแปรตัวบ่งชี้ที่สัมพันธ์กับคอลัมน์ผลลัพธ์แต่ละคอลัมน์ที่มีข้อผิดพลาดจะถูกตั้งค่าเป็น -2.

สำหรับข้อผิดพลาดจากการแก้ไขข้อมูลในการดึงข้อมูลแบบหลายแถว, ข้อผิดพลาดการแก้ไขที่รายงานเป็นการเตือนแบบ SQLSTATE จะมีพื้นที่เงื่อนไขแยกออกไปในส่วนพื้นที่วินิจฉัยของ SQL. โปรดสังเกตว่า SQL จะหยุดทำงานเมื่อเกิดข้อผิดพลาดครั้งแรก, ดังนั้นข้อผิดพลาดการแก้ไขที่ถูกรายงานเป็นข้อผิดพลาด SQLSTATE เพียงอันเดียวจะถูกส่งไปในพื้นที่การวินิจฉัยของ SQL.

สำหรับข้อความ SQL อื่นๆ, เฉพาะคำเตือน SQLSTATE อันสุดท้ายจะถูกรายงานในพื้นที่การวินิจฉัยของ SQL.

ถ้า Full-Select มี DISTINCT ในรายการการเลือก และคอลัมน์ในรายการที่เลือกมีข้อมูลตัวเลขที่มีค่าไม่ถูกต้องแล้ว, ข้อมูลจะถูกพิจารณาให้เท่ากับค่า null ถ้าการสืบค้นมีการเรียงลำดับ. ถ้าตรรกะที่มีอยู่ถูกใช้แล้ว, ข้อมูลจะไม่ถูกพิจารณาให้มีค่าเท่า null.

ผลกระทบของข้อผิดพลาดในการจับคู่ข้อมูลใน ORDER BY clause จะขึ้นอยู่กับสถานการณ์:

- ถ้าข้อผิดพลาดในการจับคู่ข้อมูลเกิดขึ้นในขณะที่ข้อมูลถูกกำหนดค่าให้กับตัวแปรโฮสต์ในคำสั่ง SELECT INTO หรือ FETCH, และนิพจน์เดียวกันถูกใช้ใน ORDER BY clause, เร็กคอร์ดผลลัพธ์จะถูกเรียงลำดับอยู่บนพื้นฐานของค่านิพจน์. โดยไม่จัดลำดับเหมือนมีค่าเป็น null (มีค่ามากกว่าค่าอื่นทั้งหมด). ที่เป็นเช่นนั้นเนื่องจากนิพจน์ถูกประเมินผลก่อนที่จะทำการกำหนดค่าให้กับตัวแปรโฮสต์.
- ถ้าข้อผิดพลาดการจับคู่ของข้อมูลเกิดขึ้นในขณะที่นิพจน์ในรายการที่เลือกกำลังถูกประเมินผล และนิพจน์เดียวกันนี้ถูกใช้ใน ORDER BY clause, คอลัมน์ผลลัพธ์จะถูกเรียงลำดับตามปกติเหมือนมีค่าเป็น null (มีค่ามากกว่าค่าอื่นทั้งหมด). ถ้า ORDER BY clause ถูกดำเนินการโดยใช้การเรียงลำดับ, คอลัมน์ผลลัพธ์จะถูกเรียงลำดับเหมือนมีค่าเป็น null. ถ้า ORDER BY clause ถูกปฏิบัติโดยใช้ตรรกะที่มีอยู่, ในกรณีดังต่อไปนี้, คอลัมน์ผลลัพธ์จะถูกเรียงลำดับอยู่บนพื้นฐานของค่าจริงของนิพจน์ในตรรกะนี้:
 - นิพจน์เป็นคอลัมน์วันที่ที่มีรูปแบบวันที่เป็น *MDY, *DMY, *YMD, หรือ *JUL, และข้อผิดพลาดการแปลงวันที่เกิดขึ้นเพราะวันที่ไม่ได้อยู่ในช่วงวันที่ที่ถูกต้อง.
 - นิพจน์เป็นคอลัมน์ตัวอักษร และตัวอักษรไม่สามารถแปลงได้.
 - นิพจน์เป็นคอลัมน์เลขทศนิยมและค่าตัวเลขที่ไม่ถูกต้องถูกตรวจพบ.

การแทรกแถวโดยใช้คำสั่ง INSERT

หัวข้อนี้จะแสดงคำสั่ง SQL พื้นฐาน และ clause ที่แทรกข้อมูลเข้าไประบบตาราง และมุมมอง. มีตัวอย่างการใช้คำสั่ง SQL เหล่านี้เพื่อช่วยเหลือในการพัฒนาแอปพลิเคชัน SQL ของคุณ.

คุณสามารถใช้คำสั่ง INSERT เพื่อเพิ่มแถวใหม่เข้าในตารางหรือมุมมอง ด้วยวิธีใดวิธีหนึ่งต่อไปนี้:

- ให้เพิ่มการระบุค่าสำหรับคอลัมน์ในคำสั่ง INSERT.
- การรวมคำสั่ง select ในคำสั่ง INSERT เพื่อแจ้งแก่ SQL ว่า มีข้อมูลสำหรับแถวใหม่ข้อมูลใดอยู่ในตารางหรือมุมมองอื่น.
- การระบุลักษณะบ่งชี้ของคำสั่ง INSERT เพื่อเพิ่มหลายแถว.

สำหรับทุกแถวที่คุณแทรก, คุณต้องให้ค่าสำหรับแต่ละคอลัมน์ที่กำหนด โดยมีแอตทริบิวต์ NOT NULL หากคอลัมน์นั้นไม่มีค่าดีฟอลต์. คำสั่ง INSERT สำหรับเพิ่มแถวเข้าในตารางหรือมุมมองอาจมีลักษณะเช่นนี้:

```
INSERT INTO table-name
    (column1, column2, ... )
VALUES (value-for-column1, value-for-column2, ... )
```

clause INTO จะให้ชื่อคอลัมน์ที่คุณระบุค่า. clause VALUES จะระบุค่าสำหรับแต่ละคอลัมน์ที่มีชื่อใน clause INTO. ค่าที่คุณระบุอาจเป็น:

- ค่าคงที่. แทรกค่าที่มีให้ใน clause VALUES .
- ค่า null. แทรกค่า null, โดยใช้ คีย์เวิร์ด NULL. คอลัมน์จะต้องถูกกำหนดเป็นแบบมีค่า null ได้ มิฉะนั้นจะเกิดข้อผิดพลาด.
- ตัวแปรโฮสต์. แทรกเนื้อหาของ ตัวแปรโฮสต์.
- เรจิสเตอร์พิเศษ. แทรกค่าเรจิสเตอร์พิเศษ; ตัวอย่างเช่น, USER.
- นิพจน์. แทรกค่าที่เป็นผลลัพธ์จาก นิพจน์.
- scalar fullselect แทรกค่าที่เป็นผลลัพธ์ของการรันคำสั่ง select.
- คีย์เวิร์ด DEFAULT. แทรกค่าดีฟอลต์ของ คอลัมน์. คอลัมน์ต้องมีค่าดีฟอลต์ที่กำหนดไว้ให้ หรือยอมให้มีค่า NULL, มิฉะนั้นจะเกิดข้อผิดพลาด.

คุณต้องให้ค่าใน VALUES clause สำหรับแต่ละคอลัมน์ที่มีชื่อในรายการคอลัมน์ของคำสั่ง INSERT. รายการชื่อคอลัมน์อาจข้ามได้ หากคอลัมน์ทั้งหมดในตารางมีค่าที่กำหนดให้ใน clause VALUES. หาก คอลัมน์มีค่าดีฟอลต์, คีย์เวิร์ด DEFAULT อาจใช้เป็นค่าใน VALUES clause. ซึ่งทำให้ค่าดีฟอลต์สำหรับคอลัมน์ถูกใส่ในคอลัมน์.

ควรระวังใส่ชื่อคอลัมน์ทั้งหมดลงในคอลัมน์ที่คุณจะแทรกค่า เนื่องจาก:

- คำสั่ง INSERT ของคุณอธิบายได้ดีกว่า.
- คุณสามารถตรวจสอบได้ว่า คุณจะให้ค่าในลำดับที่เหมาะสม โดยยึดตามชื่อคอลัมน์.
- คุณมีข้อมูลที่เป็นแบบไม่ต้องพึ่งพา (independence) มากขึ้น. ลำดับที่กำหนดคอลัมน์ในตารางไม่มีผลต่อคำสั่ง INSERT ของคุณ.

หากคอลัมน์ถูกกำหนดให้ยอมให้มีค่า null หรือมีค่าดีฟอลต์, คุณ ไม่จำเป็นต้องใส่ชื่อในรายการชื่อคอลัมน์หรือระบุค่าให้. มันจะมีค่าเป็นดีฟอลต์. หากคอลัมน์ถูกกำหนดให้มีค่าดีฟอลต์, ค่าดีฟอลต์จะถูกใส่ในคอลัมน์. หากมีการระบุ DEFAULT สำหรับ column definition โดยไม่มีค่าดีฟอลต์ที่ชัดเจน, SQL จะใส่ค่าดีฟอลต์สำหรับประเภทข้อมูลนั้นในคอลัมน์. หากคอลัมน์ไม่มีค่าดีฟอลต์ที่กำหนดไว้ให้, แต่ยอมให้มีค่า null (ไม่มีการระบุ NOT NULL ในคอลัมน์ definition), SQL จะใส่ค่า null ไว้ในคอลัมน์.

- สำหรับคอลัมน์ตัวเลข, ค่าดีฟอลต์คือ 0.
- สำหรับคอลัมน์อักขระความยาวคงที่หรือกราฟิก, ค่าดีฟอลต์คือว่าง.
- สำหรับคอลัมน์ไบนารีความยาวคงที่, ค่าดีฟอลต์คือเลขฐานสิบหกที่เป็นศูนย์.
- สำหรับคอลัมน์อักขระความยาวผันแปร, กราฟิก, หรือไบนารี และคอลัมน์ LOB, ค่าดีฟอลต์คือสตริงความยาวศูนย์.
- สำหรับคอลัมน์วันที่, เวลา, และ timestamp, ค่าดีฟอลต์คือ วันที่, เวลา, หรือ timestamp ปัจจุบัน. เมื่อแทรกกลุ่มเร็กคอร์ด, ค่าวันที่/เวลาที่ เป็นดีฟอลต์ จะถูกตัดออกจากระบบเมื่อมีการเขียนบล็อก. ซึ่งหมายความว่า คอลัมน์จะถูกกำหนดค่าเดียวกันสำหรับแต่ละแถวในบล็อก.
- สำหรับคอลัมน์ DataLink, ค่าดีฟอลต์จะตรงกับ DLVALUE('','URL','').
- สำหรับคอลัมน์แยกประเภท (distinct-type), ค่าดีฟอลต์จะเป็นค่าดีฟอลต์ของ ประเภทซอร์สที่ตรงกัน.
- สำหรับคอลัมน์ ROWID หรือคอลัมน์ที่กำหนด AS IDENTITY, ตัวจัดการฐานข้อมูลจะสร้างค่าดีฟอลต์.

เมื่อโปรแกรมของคุณพยายามแทรกแถวที่ซ้ำกับแถวอื่น ซึ่งมีอยู่ในตารางแล้ว, อาจเกิดข้อผิดพลาดขึ้น. ค่า null หลายค่าอาจถือเป็นค่าซ้ำกันหรือไม่ก็ได้, ทั้งนี้ขึ้นอยู่กับอ็อปชันที่ใช้ เมื่อสร้างตารางนี้.

- หากตารางมีคีย์หลัก, คีย์เฉพาะ, หรือดรอปคีย์เฉพาะ, จะไม่มีการแทรกแถว. แต่, SQL จะส่งคืนค่า SQLCODE เป็น -803.
- หากตารางไม่มีคีย์หลัก, คีย์เฉพาะ, หรือดรอปคีย์เฉพาะ, จะสามารถแทรกแถวได้โดยไม่เกิดข้อผิดพลาด.

หาก SQL พบข้อผิดพลาดขณะรันคำสั่ง INSERT, จะหยุดการแทรกข้อมูล. หากคุณระบุ COMMIT(*ALL), COMMIT(*CS), COMMIT(*CHG), หรือ COMMIT(*RR), จะไม่มีการแทรกแถวใดๆ. แถวที่ถูกแทรกโดยคำสั่งนี้แล้ว, ในกรณีที่เป็น INSERT ซึ่งมีคำสั่ง select หรือการแทรกแบบบล็อก, จะถูกลบออก. หากคุณระบุ COMMIT(*NONE), แถวใดๆ ที่ถูกแทรกแล้วจะ *ไม่*ถูกลบออก.

ตารางที่สร้างโดย SQL จะถูกสร้างด้วยพารามิเตอร์ Reuse Deleted Records ซึ่งเป็น *YES. ซึ่งทำให้ตัวจัดการฐานข้อมูลสามารถใช้ซ้ำแถวใดๆ ในตารางที่ถูกทำเครื่องหมายว่าลบออก. คำสั่ง CHGPF สามารถใช้เปลี่ยนแอตทริบิวต์เป็น *NO. ซึ่งทำให้ INSERT เพิ่มแถวที่ส่วนท้ายของตารางเสมอ.

ลำดับของแถวที่แทรกจะไม่รับรองว่าเป็นลำดับที่จะถูกเรียกออกมา.

หากมีการแทรกแถวโดยไม่เกิดข้อผิดพลาด, ฟิลด์ SQLERRD(3) ของ SQLCA จะมีค่าเป็น 1.

หมายเหตุ: สำหรับ INSERT ที่ถูกล็อกหรือสำหรับ INSERT ที่มีคำสั่ง select, จะสามารถแทรกได้มากกว่าหนึ่งแถว. จำนวนแถวที่แทรกจะแสดงอยู่ใน SQLERRD(3) ใน SQLCA. มันยังมีปรากฏจากรายการวินิจฉัย ROW_COUNT ในคำสั่ง GET DIAGNOSTICS.

ข้อมูลที่เกี่ยวข้อง

INSERT statement

การแทรกแถวโดยใช้คีย์เวิร์ด VALUES

ท่านสามารถใช้คีย์เวิร์ด VALUES เพื่อแทรกแถวเดี่ยว หรือ หลายแถวลงใน ตาราง.

ตัวอย่างในที่นี้เป็นการเพิ่มแทรกแถวใหม่ลงในตาราง DEPARTMENT. คอลัมน์สำหรับแถวใหม่แสดงดังต่อไปนี้:

- หมายเลข Department (DEPTNO) คือ 'E31'
- ชื่อ Department (DEPTNAME) คือ 'ARCHITECTURE'
- หมายเลข Manager (MGRNO) คือ '00390'
- รายงานสำหรับ (ADMRDEPT) ฝ่าย 'E01'

ข้อความ INSERT สำหรับแถวใหม่นี้แสดงดังต่อไปนี้:

```
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT)
VALUES ('E31', 'ARCHITECTURE', '00390', 'E01')
```

ท่านยังสามารถเพิ่มแทรกแถวหลายแถวลงในตารางโดยใช้ VALUES clause. ตัวอย่างต่อไปนี้ แสดงการแทรกแถวสองแถวลงในตาราง PROJECT. คำสำหรับ หมายเลข Project (PROJNO), ชื่อ Project (PROJNAME), หมายเลข Department (DEPTNO), และ พนักงานที่รับผิดชอบ (RESPEMP) ถูกกำหนดอยู่ในรายการแสดงค่า. คำสำหรับวันเริ่มต้น Project (PRSTDATE) ใช้วันที่ปัจจุบัน. คอลัมน์ที่เหลืออยู่ในตาราง ซึ่งไม่ได้แสดงไว้ในรายการคอลัมน์ได้ถูกกำหนดเป็นค่าดีฟอลต์.

```
INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDATE)
VALUES('HG0023', 'NEW NETWORK', 'E11', '200280', CURRENT DATE),
('HG0024', 'NETWORK PGM', 'E11', '200310', CURRENT DATE)
```

การแทรกแถวลงในตารางโดยใช้คำสั่ง select

คุณสามารถใช้คำสั่ง select ภายในคำสั่ง INSERT เพื่อแทรกเพิ่มศูนย์แถว, หนึ่งแถว, หรือ มากกว่า ลงในตารางจากตารางผลลัพธ์ของคำสั่ง select.

ประโยชน์ประการหนึ่งสำหรับคำสั่ง INSERT ประเภทนี้คือเพื่อย้ายข้อมูลเข้าในตาราง ที่คุณสร้างสำหรับข้อมูลสรุป. ตัวอย่างเช่น, สมมติว่า คุณต้องการตารางที่แสดงระยะเวลาการทำงานในโครงการของพนักงานแต่ละคน. สร้างตารางชื่อ EMPTIME โดยมีคอลัมน์ EMPNUMBER, PROJNUMBER, STARTDATE, และ ENDDATE และ แล้วใช้คำสั่ง INSERT ต่อไปนี้เพื่อเติมลงในตาราง:

```
INSERT INTO CORPDATA.EMPTIME
(EMPNUMBER, PROJNUMBER, STARTDATE, ENDDATE)
SELECT EMPNO, PROJNO, EMSTDATE, EMENDATE
FROM CORPDATA.EMPPROJACT
```

คำสั่ง select ที่อยู่ในคำสั่ง INSERT ไม่ต่างจาก คำสั่ง select ที่คุณใช้เพื่อเรียกข้อมูล. ยกเว้นกรณีที่มี clause ของ FOR READ ONLY, FOR UPDATE, หรือ OPTIMIZE, คุณสามารถใช้คีย์เวิร์ด, คอลัมน์ ฟังก์ชัน, และเทคนิคทั้งหมดที่ใช้เพื่อเรียกข้อมูล. SQL จะแทรกแถวทั้งหมดที่ตรงตามเงื่อนไขการค้นหาเข้าในตารางที่คุณระบุ. การแทรกแถว จากตารางหนึ่งไปยังอีกตารางหนึ่งไม่มีผลต่อแถวที่มีอยู่ใน ตารางต้นทางหรือตารางเป้าหมาย.

คุณควรพิจารณาสิ่งต่อไปนี้เมื่อแทรกหลายแถวเข้าในตาราง:

หมายเหตุ:

1. จำนวนคอลัมน์ที่แสดงโดยแฝงหรือชัดเจนในคำสั่ง INSERT จะต้องเท่ากับจำนวนคอลัมน์ที่แสดงในคำสั่ง select.
2. ข้อมูลในคอลัมน์ที่คุณเลือกต้องเข้ากันได้กับคอลัมน์ที่คุณจะแทรกคำสั่ง เมื่อใช้ INSERT กับคำสั่ง select.
3. ในกรณีที่คำสั่ง select ซึ่งผนวกรวมใน INSERT ไม่ให้แถวใดๆ, จะมี SQLCODE เป็น 100 เพื่อเตือนคุณว่า ไม่มีการแทรกแถว. หาก คุณแทรกแถวได้สำเร็จ, ฟิวด์ SQLERRD(3) ของ SQLCA จะมีจำนวนเต็ม ที่แทนจำนวนแถวซึ่ง SQL แทรกตามจริง. คำนี้ยังมีปรากฏจากรายการวิเคราะห์ ROW_COUNT ในคำสั่ง GET DIAGNOSTICS .
4. หาก SQL พบข้อผิดพลาดขณะรันข้อความ INSERT, SQL จะหยุด การดำเนินการ. หากคุณระบุ COMMIT (*CHG), COMMIT(*CS), COMMIT (*ALL), หรือ COMMIT(*RR), จะไม่มีการแทรกสิ่งใดเข้าในตารางและจะได้รับ SQLCODE ค่าลบ. หาก คุณระบุ COMMIT(*NONE), แถวใดๆ ที่แทรกก่อนเกิดข้อผิดพลาดจะยังคงอยู่ใน ตาราง.

การแทรกหลายแถวเข้าในตารางด้วยคำสั่ง INSERT ที่ถูกล็อก

คำสั่ง INSERT ที่ถูกล็อกสามารถใช้เพื่อแทรกหลายแถวเข้าในตารางที่มีคำสั่งเดียว.

ข้อความ INSERT แบบบล็อกจะสนับสนุนใน ทุกภาษายกเว้น REXX. ข้อมูลที่แทรกเข้าในตารางจะต้องอยู่ใน host structure array. หากใช้ตัวแปรตัวบ่งชี้กับ INSERT ที่ถูกล็อก, ตัวแปรเหล่านี้ ต้องอยู่ใน host structure array ด้วย.

ตัวอย่างเช่น, ในการเพิ่มพนักงานสิบคนเข้าในตาราง CORPDATA.EMPLOYEE:

```

INSERT INTO CORPDATA.EMPLOYEE
              (EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT)
10 ROWS VALUES(:DSTRUCT:ISTRUCT)

```

DSTRUCT เป็น host structure array ที่มีองค์ประกอบทำส่วนซึ่งแสดงใน โปรแกรม. องค์ประกอบทำส่วนตรงกับ EMPNO, FIRSTNME, MIDINIT, LASTNAME, และ WORKDEPT. DSTRUCT มีขนาดอย่างน้อยสิบเพื่อบรรจุ แถวที่แทรกสิบแถว. ISTRUCT เป็น host structure array ที่แสดงในโปรแกรม. ISTRUCT มีขนาดอย่างน้อยสิบฟิลด์จำนวนเต็มขนาดเล็กสำหรับตัวบ่งชี้.

ข้อความ INSERT จะสนับสนุนสำหรับแอ็พพลิเคชัน SQL แบบไม่กระจาย และสำหรับแอ็พพลิเคชันแบบกระจายซึ่งทั้งแอ็พพลิเคชันเซิร์ฟเวอร์ และ application requester เป็น iSeries ระบบ.

ข้อมูลที่เกี่ยวข้อง

Embedded SQL programming

การแทรกลงในตารางด้วยข้อจำกัดในการอ้างอิง

มีข้อควรจำที่สำคัญบางประการ เมื่อจะแทรกข้อมูลลงในตารางโดยใช้ข้อจำกัดในการอ้างอิง.

หากคุณแทรกข้อมูลลงใน parent table ด้วย parent key, SQL จะไม่อนุญาต:

- ให้ทำสำเนาค่า parent key
- หาก parent key คือ primary key, ค่าในคอลัมน์ใดๆ ของ primary key จะเป็นค่า null.

หากคุณแทรกข้อมูลลงในตาราง dependent ด้วย foreign key:

- แต่ละค่าที่ไม่ใช่ค่า null ที่คุณแทรกลงในคอลัมน์ foreign key จะต้องเท่ากับค่าบางค่าใน parent key ที่ตรงกันในตาราง parent.
- หากคอลัมน์ใดๆ ใน foreign key เป็น null, foreign key ทั้งหมดก็จะถูกพิจารณาว่าเป็น null เช่นกัน. หาก foreign key ทั้งหมดที่ประกอบด้วยคอลัมน์เป็น null, คำสั่ง INSERT จะทำงานสำเร็จ (ตราบใดที่ไม่มีการละเมิดตรรกะนี้เฉพาะ).

เปลี่ยนตารางโครงการแอ็พพลิเคชันตัวอย่าง (PROJECT) เพื่อกำหนด สอง foreign key:

- foreign key บนหมายเลขแผนก (DEPTNO) ซึ่งอ้างอิงถึงตารางแผนก
- foreign key บนหมายเลขพนักงาน (RESPEMP) ซึ่งอ้างอิงถึงตารางพนักงาน.

```

ALTER TABLE CORPDATA.PROJECT ADD CONSTRAINT RESP_DEPT_EXISTS
      FOREIGN KEY (DEPTNO)
      REFERENCES CORPDATA.DEPARTMENT
      ON DELETE RESTRICT

```

```

ALTER TABLE CORPDATA.PROJECT ADD CONSTRAINT RESP_EMP_EXISTS
      FOREIGN KEY (RESPEMP)
      REFERENCES CORPDATA.EMPLOYEE
      ON DELETE RESTRICT

```

โปรดสังเกตว่าคอลัมน์ตาราง parent ไม่ได้ถูกระบุไว้ใน REFERENCES clause. ไม่จำเป็นต้องระบุคอลัมน์ดังกล่าว ตราบที่ตารางซึ่งอ้างอิงถึงมี primary key หรือ unique key ซึ่งสามารถใช้เป็น parent key ได้.

ทุกแถวที่ถูกแทรกลงในตาราง PROJECT ต้องมีค่า DEPTNO ที่เท่ากับค่าบางค่าของ DEPTNO ในตารางแผนก. (ยกเว้นค่า null เนื่องจาก DEPTNO ในตารางโปรเจกต์ถูกระบุให้เป็น NOT NULL.) แถวนั้นต้องมีค่า RESPEMP ซึ่งเท่ากับค่าบางค่าของ EMPNO ในตารางพนักงานหรือเป็นค่า null.

คำสั่ง INSERT ต่อไปนี้ใช้ไม่ได้เนื่องจากไม่มีค่า DEPTNO ที่ตรงกัน ('A01') ในตาราง DEPARTMENT.

```
INSERT INTO CORPDATA.PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP)
VALUES ('AD3120', 'BENEFITS ADMIN', 'A01', '000010')
```

เช่นเดียวกัน, คำสั่ง INSERT ต่อไปนี้จะใช้ไม่ได้ถ้าไม่มีค่า EMPNO เป็น '000011' ในตาราง EMPLOYEE.

```
INSERT INTO CORPDATA.PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP)
VALUES ('AD3130', 'BILLING', 'D21', '000011')
```

คำสั่ง INSERT ต่อไปนี้สมบูรณ์อย่างครบถ้วนเนื่องจากมีค่า DEPTNO ของ 'E01' ที่ตรงกันในตาราง DEPARTMENT และมีค่า EMPNO ของ '000010' ที่ตรงกันในตาราง EMPLOYEE.

```
INSERT INTO CORPDATA.PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP)
VALUES ('AD3120', 'BENEFITS ADMIN', 'E01', '000010')
```

การแทรกค่าเข้าในคอลัมน์ identity

คุณสามารถแทรกค่าเข้าในคอลัมน์ identity หรืออนุญาตให้ระบบแทรกค่าให้คุณ.

ตัวอย่างเช่น, ตารางมีคอลัมน์ที่ชื่อ ORDERNO (คอลัมน์ identity), SHIPPED_TO (varchar(36)), และ ORDER_DATE (date). คุณสามารถแทรกแถวเข้าในตารางนี้ได้โดยใช้ข้อความต่อไปนี้:

```
INSERT INTO ORDERS (SHIPPED_TO, ORDER_DATE)
VALUES ('BME TOOL', 2002-02-04)
```

ในกรณีนี้, ระบบจะสร้างค่าสำหรับ identity column โดยอัตโนมัติ. คุณยังสามารถเขียนข้อความนี้ได้โดยใช้คีย์เวิร์ด DEFAULT:

```
INSERT INTO ORDERS (SHIPPED_TO, ORDER_DATE, ORDERNO)
VALUES ('BME TOOL', 2002-02-04, DEFAULT)
```

หลังจากแทรก, คุณสามารถใช้ฟังก์ชัน IDENTITY_VAL_LOCAL เพื่อกำหนดค่าที่ระบบกำหนดให้กับคอลัมน์.

บางครั้ง ผู้ใช้จะระบุค่าสำหรับคอลัมน์ identity, เช่น ในคำสั่ง INSERT โดยใช้ SELECT:

```
INSERT INTO ORDERS OVERRIDING USER VALUE
(SELECT * FROM TODAYS_ORDER)
```

ในกรณีนี้, OVERRIDING USER VALUE จะสั่งให้ระบบไม่สนใจค่าที่มีสำหรับ identity column จาก SELECT และให้สร้างค่าใหม่สำหรับ identity column. ต้องใช้ OVERRIDING USER VALUE หาก identity column สร้างด้วย clause GENERATED ALWAYS; แต่ไม่จำเป็นถ้าใช้ clause GENERATED BY DEFAULT. หากไม่ได้ระบุ OVERRIDING USER VALUE สำหรับ identity column ที่ระบุเป็น GENERATED BY DEFAULT, ค่าที่กำหนดใน SELECT จะนำมาแทรกลงใน identity column.

คุณสามารถบังคับให้ระบบใช้ค่าจาก SELECT ใน identity column แบบ GENERATED ALWAYS ได้โดยระบุ OVERRIDING SYSTEM VALUE. ตัวอย่างเช่น, ใช้คำสั่งต่อไปนี้:

```
INSERT INTO ORDERS OVERRIDING SYSTEM VALUE
(SELECT * FROM TODAYS_ORDER)
```

คำสั่ง INSERT นี้ใช้ค่าจาก SELECT; ซึ่งจะไม่สร้างค่าใหม่สำหรับ identity column. คุณไม่สามารถให้ค่าสำหรับ identity column ที่สร้างโดยใช้ GENERATED ALWAYS โดยไม่ใช้ clause OVERRIDING SYSTEM VALUE.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การสร้างและเปลี่ยน identity column” ในหน้า 25

ทุกครั้งที่เพิ่มแถวใหม่เข้าไปยังตารางด้วย identity column, ค่าของ identity column ในแถวใหม่จะเพิ่มขึ้น (หรือลดลง) เพราะระบบ.

ข้อมูลที่เกี่ยวข้อง

IDENTITY_VAL_LOCAL

การเปลี่ยนข้อมูลในตารางโดยใช้คำสั่ง UPDATE

หัวข้อนี้แสดงคำสั่ง SQL พื้นฐาน และ clause ที่อัปเดตข้อมูลเข้าในตารางและมุมมอง. ในการเปลี่ยนแปลงข้อมูลในตาราง, ให้ใช้คำสั่ง UPDATE.

ด้วยคำสั่ง UPDATE, คุณสามารถเปลี่ยนค่าของคอลัมน์ได้มากกว่าหนึ่งค่าในแต่ละแถว ซึ่งตรงตามเงื่อนไขการค้นหาของ WHERE clause. ผลลัพธ์ของคำสั่ง UPDATE จะมีค่าคอลัมน์ที่เปลี่ยนแปลงมากกว่าหนึ่งคอลัมน์ในจำนวนศูนย์หรือมากกว่าของตาราง (ขึ้นอยู่กับจำนวนแถวที่ตรงกับเงื่อนไขการค้นหาที่ระบุใน WHERE clause). คำสั่ง UPDATE จะเป็นดังนี้:

```
UPDATE table-name
  SET column-1 = value-1,
      column-2 = value-2, ...
  WHERE search-condition ...
```

ตัวอย่างเช่น, สมมติว่ามีการย้ายพนักงาน. ในการอัปเดตรายการข้อมูลพนักงานหลายรายการในตาราง CORPDATA.EMPLOYEE เพื่อให้เห็นถึงการย้าย, คุณสามารถระบุ:

```
UPDATE CORPDATA.EMPLOYEE
  SET JOB = :PGM-CODE,
      PHONENO = :PGM-PHONE
  WHERE EMPNO = :PGM-SERIAL
```

ใช้ clause SET เพื่อระบุค่าใหม่สำหรับแต่ละคอลัมน์ที่คุณต้องการอัปเดต. clause SET จะแสดงชื่อคอลัมน์ที่คุณต้องการให้อัปเดตและให้ค่า ที่คุณต้องการให้เปลี่ยนเป็นค่านั้น. ค่าที่คุณระบุอาจเป็น:

- ชื่อคอลัมน์. แทนที่ค่าปัจจุบันของคอลัมน์ด้วยเนื้อหาของอีกคอลัมน์ในแถวเดียวกัน.
- ข้อจำกัด. แทนที่ค่าปัจจุบันของคอลัมน์ด้วยค่าที่มีให้ใน SET clause.
- ค่า null. แทนที่ค่าปัจจุบันของคอลัมน์ด้วยค่า null, โดยใช้คีย์เวิร์ด NULL. คอลัมน์จะต้องถูกกำหนดเป็น แบบมีค่า null ได้เมื่อมีการสร้างตาราง, มิฉะนั้นจะเกิดข้อผิดพลาด.
- ตัวแปรโฮสต์. แทนที่ค่าปัจจุบันของคอลัมน์ด้วยเนื้อหาของตัวแปรโฮสต์.
- เรจิสเตอร์พิเศษ. แทนที่ค่าปัจจุบันของคอลัมน์ด้วยค่าเรจิสเตอร์พิเศษ; ตัวอย่างเช่น, USER.
- นิพจน์. แทนที่ค่าปัจจุบันของคอลัมน์ด้วยค่าที่เป็นผลลัพธ์จากนิพจน์.
- scalar fullselect. แทนที่ค่าปัจจุบันของคอลัมน์ ด้วยค่าที่ subquery ให้มา.
- คีย์เวิร์ด DEFAULT. แทนที่ค่าปัจจุบันของคอลัมน์ ด้วยค่าดีฟอลต์ของคอลัมน์. คอลัมน์ต้องมีค่าดีฟอลต์ ที่กำหนดไว้ให้ หรือยอมให้มีค่า NULL, มิฉะนั้นจะเกิดข้อผิดพลาด.

ต่อไปนี้เป็นตัวอย่างของคำสั่งที่ใช้ค่าต่างกันหลายค่า:

```

UPDATE WORKTABLE
  SET COL1 = 'ASC',
      COL2 = NULL,
          COL3 = :FIELD3,
      COL4 = CURRENT TIME,
          COL5 = AMT - 6.00,
          COL6 = COL7
  WHERE EMPNO = :PGM-SERIAL

```

ในการระบุแถวที่จะอัปเดต, ให้ใช้ clause WHERE:

- ในการอัปเดตแถวเดียว, ให้ใช้ WHERE clause ที่เลือกเพียงแถวเดียว.
- ในการอัปเดตหลายแถว, ให้ใช้ clause WHERE ที่เลือกเฉพาะแถว ที่คุณต้องการอัปเดต.

คุณสามารถละเว้น clause WHERE. หากคุณเว้น, SQL จะอัปเดตแต่ละแถวในตาราง หรือมุมมองซึ่งมีค่าที่คุณให้.

หากตัวจัดการฐานข้อมูลพบข้อผิดพลาดขณะรันข้อความ UPDATE ของคุณ, ตัวจัดการจะหยุดและให้ SQLCODE ที่เป็นค่าลบ. หากคุณระบุ COMMIT(*ALL), COMMIT(*CS), COMMIT(*CHG), หรือ COMMIT(*RR), จะไม่มีการเปลี่ยนแถวใดในตาราง (แถวที่ถูกเปลี่ยนโดยข้อความนี้แล้ว, หากมี, จะถูกเรียกคืนเป็นค่า ก่อนหน้านี้). หากระบุ COMMIT(*NONE), แถวใดๆ ที่ถูกเปลี่ยนแล้วจะ *ไม่* ถูกเรียกคืนเป็นค่าก่อนหน้านี้.

หากตัวจัดการฐานข้อมูลไม่พบแถวใดที่ตรงตามเงื่อนไขการค้นหา, จะได้ SQLCODE เป็น +100.

หมายเหตุ: ข้อความ UPDATE อาจได้อัปเดต มากกว่าหนึ่งแถว. จำนวนของแถวที่ถูกอัปเดตถูกแสดงเป็น SQLERRD(3) ของ SQLCA. ค่านี้ยังมีปรากฏจากรายการวิเคราะห์ ROW_COUNT ในข้อความ GET DIAGNOSTICS.

clause SET ของข้อความ UPDATE สามารถใช้ได้หลายวิธีเพื่อกำหนด ค่าแท้จริงที่จะกำหนดในแต่ละแถวที่กำลังอัปเดต. ตัวอย่างต่อไปนี้แสดงแต่ละคอลัมน์และค่าของมัน:

```

UPDATE EMPLOYEE
  SET WORKDEPT = 'D11',
      PHONENO = '7213',
      JOB = 'DESIGNER'
  WHERE EMPNO = '000270'

```

ตัวอย่างการอัปเดตก่อนหน้านี้ยังสามารถเขียนได้โดยระบุคอลัมน์ทั้งหมดแล้วระบุค่าทั้งหมด:

```

UPDATE EMPLOYEE
  SET (WORKDEPT, PHONENO, JOB)
    = ('D11', '7213', 'DESIGNER')
  WHERE EMPNO = '000270'

```

ข้อมูลที่เกี่ยวข้อง

UPDATE statement

การอัปเดตตารางโดยใช้ scalar-subselect

อีกวิธีหนึ่งในการเลือกค่าหนึ่ง (หรือหลายค่า) สำหรับการอัปเดต คือใช้ scalar-subselect. scalar-subselect อนุญาตให้คุณอัปเดตคอลัมน์หนึ่งคอลัมน์ หรือมากกว่าได้โดยกำหนดเป็นค่าหนึ่งหรือมากกว่า ซึ่งเลือกได้จากอีกตารางหนึ่ง.

ในตัวอย่างต่อไปนี้, พนักงานย้ายไปแผนกอื่นแต่ยังทำงานโปรเจกต์เดิมต่อไป. ตารางพนักงานถูกอัปเดต เพื่อให้มีหมายเลขแผนกใหม่. ขณะนี้จำเป็นต้องอัปเดตตารางโปรเจกต์ เพื่อแสดงหมายเลขแผนกใหม่ของพนักงานคนนี้ (หมายเลขพนักงานคือ '000030').

```
UPDATE PROJECT
  SET DEPTNO =
      (SELECT WORKDEPT FROM EMPLOYEE
       WHERE PROJECT.RESPEMP = EMPLOYEE.EMPNO)
 WHERE RESPEMP='000030'
```

เทคนิคเดียวกันนี้ สามารถใช้เพื่ออัปเดตรายการของคอลัมน์ที่มีค่าหลายค่าซึ่งได้มาจากการเลือก ครั้งเดียว.

การอัปเดตตารางที่มีแถวจากตารางอื่น

นอกจากนี้ยังสามารถอัปเดตแถวทั้งหมดในตารางหนึ่งที่มีค่าจากแถวหนึ่งในตารางอื่น.

สมมติว่ามีตารางกำหนดการคลาสหลักซึ่งจำเป็นต้องอัปเดตความเปลี่ยนแปลงที่เกิดขึ้นในสำเนาของตาราง. สำเนานั้นมีการเปลี่ยนแปลงและผนวกรวมเข้าในตารางหลัก ทุกคืน. ตารางสองตารางมีคอลัมน์ต่างๆ ที่เหมือนกัน และคอลัมน์ CLASS_CODE เป็นคีย์คอลัมน์เฉพาะ.

```
UPDATE CL_SCHED
  SET ROW =
      (SELECT * FROM MYCOPY
       WHERE CL_SCHED.CLASS_CODE = MYCOPY.CLASS_CODE)
```

การอัปเดตนี้จะอัปเดตแถวทั้งหมดใน CL_SCHED ด้วยค่าจาก MYCOPY.

การอัปเดตตารางด้วยข้อจำกัดในการอ้างอิง

หากคุณอัปเดตตาราง *parent*, คุณไม่สามารถแก้ไข primary key ที่มีแถว dependent อยู่ได้.

การเปลี่ยนคีย์เป็นการละเมิดข้อจำกัดในการอ้างอิงของตาราง dependent และทำให้แถวบางแถวไม่มี parent. นอกจากนี้, คุณไม่สามารถระบุค่า null ในส่วนใดๆ ของ primary key.

กฎการอัปเดต

การดำเนินการที่เกิดขึ้นบนตาราง dependent เมื่อมีการใช้ UPDATE บนตาราง parent ขึ้นอยู่กับกฎการอัปเดต ที่ระบุให้กับข้อจำกัดในการอ้างอิง. หากไม่มีการระบุกฎการอัปเดตสำหรับข้อจำกัดในการอ้างอิง, กฎ UPDATE NO ACTION ก็จะถูกนำมาใช้.

UPDATE NO ACTION

ระบุว่าแถวในตาราง parent สามารถอัปเดตได้หากไม่มีแถวอื่นที่ต้องพึ่งพิงแถวนั้น. หากมีแถว dependent อยู่ในความสัมพันธ์, UPDATE ก็จะใช้ไม่ได้. ระบบจะตรวจสอบแถว dependent เมื่อรันคำสั่งเสร็จ.

UPDATE RESTRICT

ระบุว่าแถวในตาราง parent สามารถอัปเดตได้หากไม่มีแถวอื่นที่ต้องพึ่งพิงแถวนั้น. หากมีแถว dependent อยู่ในความสัมพันธ์, UPDATE ก็จะใช้ไม่ได้. ระบบจะตรวจสอบแถว dependent ทันที.

ความแตกต่างอย่างชัดเจนระหว่างกฎ RESTRICT และกฎ NO ACTION สามารถเห็นได้อย่างง่ายดาย เมื่อดูจากปฏิกิริยาโต้ตอบของทริกเกอร์และข้อจำกัดในการอ้างอิง. คุณสามารถระบุทริกเกอร์ให้ทำงานก่อนหรือหลังการปฏิบัติการ (ซึ่งก็คือคำสั่ง UPDATE, ในกรณีนี้). คำ *ก่อนทริกเกอร์* จะสั่งงานก่อน UPDATE จะทำงานและก่อนการตรวจสอบข้อจำกัดใดๆ. คำ *หลังทริก*

ก่อนถูกสั่งงานหลังจากที่ UPDATE ทำงานแล้ว, และหลังกฎข้อจำกัด RESTRICT (โดยที่มีการตรวจสอบทันที), แต่สั่งงานก่อนกฎข้อจำกัด NO ACTION (โดยที่มีการตรวจสอบเมื่อสิ้นสุดคำสั่ง). ทริกเกอร์และกฎจะเกิดขึ้นตามลำดับต่อไปนี้:

1. ค่าก่อนทริกเกอร์จะถูกสั่งงานก่อน UPDATE และก่อนกฎข้อจำกัด RESTRICT หรือ NO ACTION.
2. ค่าหลังทริกเกอร์จะถูกสั่งงานหลังกฎข้อจำกัด RESTRICT, แต่สั่งงานก่อนกฎ NO ACTION.

หากคุณอัปเดตตาราง *dependent*, ค่า foreign key ใดๆ ที่ไม่ใช่ null ที่คุณเปลี่ยนต้องตรงกับ primary key สำหรับแต่ละความสัมพันธ์โดยที่ตารางเป็นแบบ dependent. ตัวอย่างเช่น, หมายเลขแผนกในตารางพนักงานขึ้นอยู่กับหมายเลขแผนกในตารางแผนก. คุณสามารถกำหนดค่าพนักงานให้เป็นไม่มีแผนกได้ (ค่า null), แต่กำหนดค่าพนักงานให้แผนกที่ไม่มีอยู่จริงไม่ได้.

หาก UPDATE ตารางโดยใช้ข้อจำกัดในการอ้างอิงล้มเหลว, การเปลี่ยนแปลงทั้งหมดที่เกิดขึ้นระหว่างการอัปเดตจะไม่สามารถสมบูรณ์.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การทำเจอร์นัล” ในหน้า 130

เจอร์นัลสนับสนุนของ DB2 UDB for iSeries จะช่วยเป็นหลักฐานการตรวจสอบและการการกู้คืนทั้งแบบ forward และ backward.

“Commitment control” ในหน้า 131

การสนับสนุน DB2 UDB for iSeries commitment control จะรวบรวมวิธีการประมวลผลกลุ่มของการเปลี่ยนแปลงของฐานข้อมูล เช่น อัปเดต, แทรก, DDL, หรือลบออก ในลักษณะของหน่วยการทำงานเดี่ยว (transaction).

ตัวอย่าง: กฎ UPDATE:

ตัวอย่างเหล่านี้อธิบายถึงกฎ UPDATE.

ตัวอย่างเช่น, คุณไม่สามารถอัปเดตหมายเลขแผนกจากตารางแผนกได้หากตารางดังกล่าวยังมีชื่อบางโครงการอยู่, ซึ่งอธิบายด้วยแถว dependent ในตารางโปรเจกต์.

UPDATE ใช้ไม่ได้เนื่องจากตารางโปรเจกต์มีแถวซึ่งต้องอิงกับ DEPARTMENT.DEPTNO ที่ประกอบด้วยค่า 'D01' แถวดังกล่าวถูกวางเป้าหมายด้วยคำสั่ง WHERE). ถ้าคำสั่ง UPDATE ใช้ได้, ข้อจำกัดที่อ้างอิงระหว่าง ตาราง PROJECT และตาราง DEPARTMENT จะยกเลิกไป.

```
UPDATE CORPDATA.DEPARTMENT
  SET DEPTNO = 'D99'
  WHERE DEPTNAME = 'DEVELOPMENT CENTER'
```

คำสั่งต่อไปนี้ใช้ไม่ได้เนื่องจากเป็นการละเมิดข้อจำกัดในการอ้างอิงที่มีอยู่ระหว่าง primary key DEPTNO ใน DEPARTMENT และ foreign key DEPTNO ใน PROJECT:

```
UPDATE CORPDATA.PROJECT
  SET DEPTNO = 'D00'
  WHERE DEPTNO = 'D01';
```

คำสั่งพยายามที่จะเปลี่ยนหมายเลขแผนก D01 ทั้งหมดเป็นหมายเลขแผนก D00. เนื่องจาก D00 ไม่ใช่ค่า primary key DEPTNO ใน DEPARTMENT, คำสั่งจึงใช้ไม่ได้.

การอัปเดต identity column

คุณสามารถอัปเดตค่าใน identity column ให้เป็นค่าที่ระบุ หรือให้ระบบสร้างค่าใหม่.

ตัวอย่างเช่น, เมื่อใช้ตารางที่มีคอลัมน์ชื่อ ORDERNO (identity column), SHIPPED_TO (varchar(36)), และ ORDER_DATE (date), คุณสามารถอัปเดตค่าใน identity column ได้โดยใช้คำสั่งต่อไปนี้:

```
UPDATE ORDERS
  SET (ORDERNO, ORDER_DATE)=
      (DEFAULT, 2002-02-05)
  WHERE SHIPPED_TO = 'BME TOOL'
```

ระบบจะสร้างค่า สำหรับ identity column โดยอัตโนมัติ. คุณสามารถแทนที่ค่าที่ระบบสร้างให้โดยใช้ clause OVERRIDING SYSTEM VALUE:

```
UPDATE ORDERS OVERRIDING SYSTEM VALUE
  SET (ORDERNO, ORDER_DATE)=
      (553, '2002-02-05')
  WHERE SHIPPED_TO = 'BME TOOL'
```

สิ่งอ้างอิงที่เกี่ยวข้อง

“การสร้างและเปลี่ยน identity column” ในหน้า 25

ทุกครั้งที่คุณเพิ่มแถวใหม่เข้าไปยังตารางด้วย identity column, ค่าของ identity column ในแถวใหม่นี้จะเพิ่มขึ้น (หรือลดลง) เพราะระบบ.

การอัปเดตข้อมูลตามที่ดึงมาจากตาราง

คุณสามารถอัปเดตแถวของข้อมูลตามที่คุณได้ดึงออกมาโดยใช้เคอร์เซอร์.

ในข้อความเลือก, ให้ใช้ FOR UPDATE OF ตามด้วยรายการคอลัมน์ที่อัปเดตได้. แล้ว ใช้ข้อความ UPDATE ที่ควบคุมด้วย เคอร์เซอร์. clause WHERE CURRENT OF จะระบุชื่อ เคอร์เซอร์ซึ่งชี้ไปยังแถวที่คุณต้องการอัปเดต. หากไม่ได้รับ clause FOR UPDATE OF, ORDER BY, FOR READ ONLY, หรือ SCROLL แบบที่ไม่มี clause DYNAMIC ไว้, คอลัมน์ทั้งหมด สามารถอัปเดตได้.

หากมีการระบุและรันข้อความ FETCH แบบหลายแถว, เคอร์เซอร์ จะอยู่ที่แถวสุดท้ายของบล็อก. ดังนั้น, หากมีการระบุ clause WHERE CURRENT OF ในข้อความ UPDATE, แถวสุดท้ายในบล็อก จะถูกอัปเดต. หากต้องอัปเดตแถวภายในบล็อก, ก่อนอื่นโปรแกรมต้อง วางเคอร์เซอร์ไว้ที่แถวนั้น. แล้วจึงระบุ UPDATE WHERE CURRENT OF ได้. พิจารณาตัวอย่างนี้:

ตารางที่ 21. การอัปเดตตาราง

Scrollable Cursor SQL Statement	Comments
<pre>EXEC SQL DECLARE THISEMP DYNAMIC SCROLL CURSOR FOR SELECT EMPNO, WORKDEPT, BONUS FROM CORPDATA.EMPLOYEE WHERE WORKDEPT = 'D11' FOR UPDATE OF BONUS END-EXEC.</pre>	
<pre>EXEC SQL OPEN THISEMP END-EXEC.</pre>	

ตารางที่ 21. การอัปเดตตาราง (ต่อ)

Scrollable Cursor SQL Statement	Comments
<pre>EXEC SQL WHENEVER NOT FOUND GO TO CLOSE-THISEMP END-EXEC.</pre>	
<pre>EXEC SQL FETCH NEXT FROM THISEMP FOR 5 ROWS INTO :DEPTINFO :IND-ARRAY END-EXEC.</pre>	<p>DEPTINFO และ IND-ARRAY ถูกประกาศในโปรแกรมเป็น host structure array และ indicator array.</p>
<p>... ตรวจสอบว่ามีพนักงานในแผนก D11 รับโบนัสน้อยกว่า 500.00 ดอลลาร์หรือไม่. หากเป็นเช่นนั้น, ให้อัปเดตเรกคอร์ดนั้นใหม่ให้ค่าน้อยที่สุดเป็น 500.00 ดอลลาร์.</p>	
<pre>EXEC SQL FETCH RELATIVE :NUMBACK FROM THISEMP END-EXEC.</pre>	<p>... ระบุตำแหน่งเรกคอร์ดในบล็อกเพื่ออัปเดตโดยดึงข้อมูลออกในลำดับกลับกัน.</p>
<pre>EXEC SQL UPDATE CORPDATA.EMPLOYEE SET BONUS = 500 WHERE CURRENT OF THISEMP END-EXEC.</pre>	<p>... อัปเดตโบนัสสำหรับพนักงานในแผนก D11 ผู้ได้น้อยกว่าอัตราใหม่ที่น้อยที่สุด 500.00 ดอลลาร์.</p>
<pre>EXEC SQL FETCH RELATIVE :NUMBACK FROM THISEMP FOR 5 ROWS INTO :DEPTINFO :IND-ARRAY END-EXEC.</pre>	<p>... วางตำแหน่งที่ส่วนเริ่มต้นของ บล็อกเดียวกันที่ถูกดึงข้อมูลออกมาแล้ว และดึงข้อมูลออกจากบล็อกอีกครั้ง. (NUMBACK - (5 - NUMBACK - 1))</p>
<p>... แบนช์ (branch) กลับเพื่อพิจารณาว่ามีพนักงานอื่นในกลุ่มเรกคอร์ดมีโบนัสน้อยกว่า 500.00 ดอลลาร์อีกหรือไม่.</p>	
<p>... แบนช์กลับ เพื่อดึงข้อมูลออกและดำเนินการกับบล็อกต่อไปของแถว.</p>	
<pre>CLOSE-THISEMP. EXEC SQL CLOSE THISEMP END-EXEC.</pre>	

สิ่งอ้างอิงที่เกี่ยวข้อง

“การใช้เคอร์เซอร์” ในหน้า 256

เมื่อ SQL รันข้อความที่เลือก, แถวผลลัพธ์จะประกอบขึ้นจากรายการผลลัพธ์. เคอร์เซอร์จะแสดงวิธีการเข้าถึงตารางผลลัพธ์.

การลบแถวออกจากตารางโดยใช้คำสั่ง DELETE

ในการลบแถวออกจากตาราง, ให้ใช้คำสั่ง DELETE.

เมื่อคุณ DELETE แถว, คุณจะลบออกทั้งแถว. DELETE จะไม่ลบเพียงบางคอลัมน์ออกจากแถว. ผลของคำสั่ง DELETE คือการลบแถวจำนวนศูนย์แถวหรือมากกว่าออกจากตาราง (ทั้งนี้ขึ้นอยู่กับว่ามีแถวที่ตรงตามเงื่อนไขการค้นหาที่ระบุใน clause WHERE เท่าใด). หากคุณละเว้น clause WHERE จากคำสั่ง DELETE, SQL จะลบแถวทั้งหมดของตาราง. คำสั่ง DELETE มีลักษณะ เช่นนี้:

```
DELETE FROM table-name
WHERE search-condition ...
```

ตัวอย่างเช่น, สมมติว่าแผนก D11 ถูกย้ายไปที่อื่น. คุณต้องการ ลบแต่ละแถวในตาราง CORPDATA.EMPLOYEE ที่มีค่า WORKDEPT ของ D11 ดังต่อไปนี้:

```
DELETE FROM CORPDATA.EMPLOYEE
WHERE WORKDEPT = 'D11'
```

clause WHERE จะบอก SQL ว่าแถวใดที่คุณต้องการลบออกจากตาราง. SQL จะลบแถวทั้งหมดที่ตรงตามเงื่อนไขการค้นหาจากตารางฐาน. การลบแถวจากมุมมองจะเป็นการลบแถวออกจากตารางฐาน. คุณสามารถละเว้น clause WHERE, แต่ไม่ควรจะรวมไว้, เนื่องจากคำสั่ง DELETE ที่ไม่มี clause WHERE จะลบแถวทั้งหมดจากตารางหรือมุมมอง. ในการลบ definition ตาราง และเนื้อหาของตาราง, ให้ใช้คำสั่ง DROP.

หาก SQL พบข้อผิดพลาดขณะรันคำสั่ง DELETE ของคุณ, จะหยุดการลบข้อมูลและให้ SQLCODE ที่เป็นลบ. หากคุณระบุ COMMIT(*ALL), COMMIT(*CS), COMMIT(*CHG), หรือ COMMIT(*RR), จะไม่มีการลบแถวใดในตาราง (แถวที่ถูกลบด้วยคำสั่งนี้แล้ว, หากมี, จะถูกเรียกคืนเป็นค่าก่อนหน้านั้น). หากไม่มีการระบุ COMMIT(*NONE), แถวใดๆ ที่ถูกลบแล้วจะ *ไม่* ถูกเรียกคืนเป็นค่าก่อนหน้านั้น.

หาก SQL ไม่พบแถวใดที่ตรงตามเงื่อนไขการค้นหา, จะได้ SQLCODE เป็น +100.

หมายเหตุ: คำสั่ง DELETE อาจใช้ลบแถวออกมากกว่าหนึ่งแถว. จำนวนของแถวที่ถูกลบจะแสดงอยู่ใน SQLERRD(3) ของ SQLCA. ค่านี้ยังมีปรากฏจากรายการวิเคราะห์ ROW_COUNT ในคำสั่ง GET DIAGNOSTICS .

ข้อมูลที่เกี่ยวข้อง

DROP statement

DELETE statement

การลบจากตารางด้วยข้อจำกัดในการอ้างอิง

ถ้าตารางมี primary key แต่ไม่มี dependent, คำสั่ง DELETE จะทำงานตามปกติโดยไม่มีข้อจำกัดในการอ้างอิง. หลักการนี้เป็นจริงหากตารางมีเฉพาะ foreign key, แต่ไม่มี primary key. ถ้าตารางมี primary key และเป็นตาราง dependent, DELETE จะลบหรืออัปเดตแถวตามกฎการลบที่ระบุ.

การดำเนินการจะต้องเป็นไปตามกฎการลบทั้งหมดของความสัมพันธ์ที่ได้รับผลกระทบเพื่อให้การลบสำเร็จ. หากมีการละเมิดข้อจำกัดในการอ้างอิง, DELETE จะใช้ไม่ได้.

สิ่งที่จะดำเนินการบนตาราง dependent เมื่อ DELETE ทำงานบนตาราง parent ขึ้นอยู่กับกฎการลบที่ระบุสำหรับข้อจำกัดในการอ้างอิง. หากไม่มีการกำหนดกฎการลบ, กฎ DELETE NO ACTION จะถูกนำมาใช้.

DELETE NO ACTION

ให้ระบุว่าแถวในตารางหลักสามารถลบออกได้ ถ้าไม่มีแถวอื่นๆ อ้างอิงถึงมัน. หากมีแถว dependent อยู่ในความสัมพันธ์, DELETE จะใช้ไม่ได้. ระบบจะตรวจสอบแถว dependent เมื่อรันคำสั่งเสร็จ.

DELETE RESTRICT

ให้ระบุว่าแถวในตารางหลักสามารถลบออกได้ ถ้าไม่มีแถวอื่นๆ อ้างอิงถึงมัน. หากมีแถว dependent อยู่ในความสัมพันธ์, DELETE จะใช้ไม่ได้. ระบบจะตรวจสอบแถว dependent ทันที.

ตัวอย่างเช่น, คุณไม่สามารถลบแผนกจากตารางแผนกได้ หากตารางดังกล่าวยังมีชื่อบางโครงการซึ่งอธิบายด้วยแถว dependent ในตารางโครงการ.

DELETE CASCADE

ระบุว่าแถวที่ถูกกำหนดในตาราง parent จะถูกลบออกเป็นอันดับแรก. จากนั้น, แถว dependent จะถูกลบออก.

ตัวอย่างเช่น, คุณสามารถลบแผนกได้ด้วยการลบแถวของแผนกในตารางแผนกออก. การลบแถวออกจากตารางแผนกยังเป็นการลบ:

- แถวของทุกแผนกที่รายงานมายังตารางแผนก
- ทุกแผนกที่รายงานมายังแผนกเหล่านั้นและต่อจากนั้น.

DELETE SET NULL

ระบุว่าแต่ละคอลัมน์ที่มีค่าเป็น null ได้ใน foreign key ของแต่ละแถว dependent ถูกตั้งให้เป็นค่าดีฟอลต์. หมายความว่าคอลัมน์จะถูกตั้งเฉพาะให้เป็นค่าดีฟอลต์หากคอลัมน์นั้นเป็นเมมเบอร์ของคีย์แปลกปลอมที่อ้างอิงถึงแถวที่ถูกลบออก. เฉพาะแถว dependent ที่อยู่ในชั้นถัดมาเท่านั้นที่ได้รับผลกระทบ.

DELETE SET DEFAULT

ระบุว่าแต่ละคอลัมน์ของ foreign key ในแต่ละแถว dependent ถูกตั้งให้เป็นค่าดีฟอลต์. หมายความว่าคอลัมน์จะถูกตั้งเฉพาะให้เป็นค่าดีฟอลต์หากคอลัมน์นั้นเป็นเมมเบอร์ของคีย์แปลกปลอมที่อ้างอิงถึงแถวที่ถูกลบออก. เฉพาะแถว dependent ที่เป็นอยู่ในชั้นถัดมาเท่านั้นที่ได้รับผลกระทบ.

ตัวอย่างเช่น, คุณสามารถลบพนักงานออกจากตารางพนักงาน (EMPLOYEE) ได้แม้ว่าพนักงานนั้นจะบริหารบางแผนกก็ตาม. ในกรณีนี้, ค่า MGRNO สำหรับพนักงานแต่ละคนซึ่งรายงานไปยังผู้จัดการแผนกคนนั้นจะถูกตั้งเป็นค่าเปล่าในตารางแผนก (DEPARTMENT). หากมีการระบุค่าดีฟอลต์อื่นๆ บางค่าในการสร้างตาราง, ค่าเหล่านั้นจะถูกนำไปใช้.

เนื่องจากการระบุข้อจำกัด REPORTS_TO_EXISTS สำหรับตารางแผนกไว้.

หากตารางในลำดับชั้นถัดไปมีกฎการลบ RESTRICT หรือ NO ACTION และพบแถวที่ที่ไม่สามารถลบแถวในลำดับชั้นถัดมาได้, DELETE ทั้งหมดจะใช้ไม่ได้.

เมื่อรันคำสั่งนี้ด้วยโปรแกรม, จำนวนแถวที่ถูกลบออกจะถูกส่งคืนมาใน SQLERRD(3) ใน SQLCA. จำนวนนี้มีเฉพาะจำนวนแถวที่ถูกลบออกในตารางซึ่งระบุในคำสั่ง DELETE. แต่ไม่รวมถึงแถวที่ถูกลบออกตามกฎ CASCADE. SQLERRD(5) ใน

SQLCA ประกอบด้วยแถวที่ได้รับผลกระทบโดยข้อจำกัดในการอ้างอิงในตารางทั้งหมด. ค่า SQLERRD(3) ยังมีปรากฏจากรายการ ROW_COUNT ในข้อความ GET DIAGNOSTICS . ค่า SQLERRD(5) ยังมีปรากฏจากรายการ DB2_ROW_COUNT_SECONDARY .

ความแตกต่างอย่างชัดเจนระหว่างกฎ RESTRICT และ NO ACTION สามารถเห็นได้อย่างง่ายดาย เมื่อดูที่ปฏิบัติการโต้ตอบของทริกเกอร์และข้อจำกัดในการอ้างอิง. คุณสามารถระบุทริกเกอร์ให้ทำงานก่อนหรือหลัง การปฏิบัติการ (ซึ่งก็คือคำสั่ง DELETE, ในกรณีนี้). คำ *ก่อนทริกเกอร์* จะทำงานก่อน DELETE จะทำงานและก่อนการตรวจสอบข้อจำกัดใดๆ. คำ *หลังทริกเกอร์* ถูกสั่งงานหลังจากที่ DELETE ทำงาน, และหลังกฎข้อจำกัด RESTRICT (โดยที่มีการตรวจสอบทันที), แต่ก่อนกฎข้อจำกัด NO ACTION (โดยที่มีการตรวจสอบเมื่อสิ้นสุดคำสั่ง). ทริกเกอร์และกฎจะเกิดขึ้นตามลำดับต่อไปนี้:

1. คำ *ก่อนทริกเกอร์* จะถูกสั่งงานก่อน DELETE และก่อนกฎข้อจำกัด RESTRICT หรือ NO ACTION.
2. คำ *หลังทริกเกอร์* จะถูกสั่งงานหลังกฎข้อจำกัด RESTRICT, แต่สั่งงานก่อนกฎ NO ACTION.

ตัวอย่าง: กฎ DELETE แบบต่อเรียง:

การลบแผนกออกจากตาราง DEPARTMENT เป็นการตั้งค่า WORKDEPT (ในตาราง EMPLOYEE) ให้เป็น null สำหรับพนักงานทุกคนที่ถูกมอบหมายให้กับแผนกนั้น. พิจารณาคำสั่ง DELETE ต่อไปนี้:

```
DELETE FROM CORPDATA.DEPARTMENT
WHERE DEPTNO = 'E11'
```

ดูจากตาราง และข้อมูลใน “DB2 UDB for iSeries ตารางตัวอย่าง” ในหน้า 335, มีแถวหนึ่งแถวที่ถูกลบออกจากตาราง DEPARTMENT, และตาราง EMPLOYEE ได้รับการอัปเดตเพื่อให้ตั้งค่า WORKDEPT เป็นค่าเริ่มต้นโดยที่ค่าเท่ากับ 'E11'. เครื่องหมายคำถาม (?) ในข้อมูลตัวอย่างข้างล่างแสดงถึงค่า null. ผลลัพธ์ปรากฏขึ้นดังต่อไปนี้:

ตารางที่ 22. ตาราง DEPARTMENT. เนื้อหาของตารางหลังจากที่คำสั่ง DELETE เสร็จสมบูรณ์.

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT
A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00
B01	PLANNING	000020	A00
C01	INFORMATION CENTER	000030	A00
D01	DEVELOPMENT CENTER	?	A00
D11	MANUFACTURING SYSTEMS	000060	D01
D21	ADMINISTRATION SYSTEMS	000070	D01
E01	SUPPORT SERVICES	000050	A00
E21	SOFTWARE SUPPORT	000100	E01
F22	BRANCH OFFICE F2	?	E01
G22	BRANCH OFFICE G2	?	E01
H22	BRANCH OFFICE H2	?	E01
I22	BRANCH OFFICE I2	?	E01
J22	BRANCH OFFICE J2	?	E01

โปรดสังเกตว่า ไม่มีการลบแบบต่อเรียงในตาราง DEPARTMENT เนื่องจากไม่มีแผนกใดที่รายงานไปยังแผนก 'E11'.

ต่อไปนี้เป็นข้อมูลเก็บจากหน่วยความจำของส่วนหนึ่งของตาราง EMPLOYEE ที่ได้รับผลกระทบก่อนและหลังจากที่ คำสั่ง DELETE จะเสร็จสมบูรณ์.

ตารางที่ 23. ตาราง EMPLOYEE บางส่วน. เนื้อหาบางส่วนก่อนหน้าคำสั่ง DELETE.

EMPNO	FIRSTNME	MI	LASTNAME	WORKDEPT	PHONENO	HIREDATE
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05
000250	DANIEL	S	SMITH	D21	0961	1960-10-30
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11
000270	MARIA	L	PEREZ	D21	9001	1980-09-30
000280	ETHEL	R	SCHNEIDER	E11	0997	1967-03-24
000290	JOHN	R	PARKER	E11	4502	1980-05-30
000300	PHILIP	X	SMITH	E11	2095	1972-06-19
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07
000330	WING		LEE	E21	2103	1976-02-23
000340	JASON	R	GOUNOT	E21	5696	1947-05-05

ตารางที่ 24. ตาราง EMPLOYEE บางส่วน. เนื้อหาบางส่วนหลังคำสั่ง DELETE.

EMPNO	FIRSTNME	MI	LASTNAME	WORKDEPT	PHONENO	HIREDATE
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05
000250	DANIEL	S	SMITH	D21	0961	1960-10-30
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11
000270	MARIA	L	PEREZ	D21	9001	1980-09-30
000280	ETHEL	R	SCHNEIDER	?	0997	1967-03-24
000290	JOHN	R	PARKER	?	4502	1980-05-30
000300	PHILIP	X	SMITH	?	2095	1972-06-19
000310	MAUDE	F	SETRIGHT	?	3332	1964-09-12
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07
000330	WING		LEE	E21	2103	1976-02-23

ตารางที่ 24. ตาราง EMPLOYEE บางส่วน (ต่อ). เนื้อหาบางส่วนหลังคำสั่ง DELETE.

EMPNO	FIRSTNAME	MI	LASTNAME	WORKDEPT	PHONENO	HIREDATE
000340	JASON	R	GOUNOT	E21	5696	1947-05-05

สิ่งอ้างอิงที่เกี่ยวข้อง

“DB2 UDB for iSeries ตารางตัวอย่าง” ในหน้า 335

หัวข้อนี้ประกอบด้วยตารางตัวอย่างที่ใช้อ้างอิง และใช้ในหัวข้อนี้ รวมทั้งหัวข้อ SQL Reference.

การใช้เคียวรี้อย

คุณสามารถใช้เคียวรี้อยในเงื่อนไขการค้นหาเพื่อเป็นอีกทางหนึ่งในการเลือกข้อมูล. เคียวรี้อยสามารถใช้ได้ทุกที่ที่นิพจน์สามารถใช้งานได้.

ตามหลักการแล้ว, จะมีการประเมินผลเคียวรี้อยเมื่อใดก็ตามที่แถวหรือกลุ่มแถวใหม่ถูกประมวลผล. ที่จริงแล้ว, หากเคียวรี้อยของทุกแถวหรือทุกกลุ่มเป็นแบบเดียวกัน, เคียวรี้อยนั้นก็จะถูกประเมินผลเพียงครั้งเดียว. เคียวรี้อยแบบนี้เรียกว่า ภาวะที่ไม่สัมพันธ์กัน.

เคียวรี้อยบางอย่างจะส่งคืนค่าที่แตกต่างกันจากแถวสู่แถวหรือจากกลุ่มสู่กลุ่ม. กลไกข้างต้นนี้เรียกว่า ภาวะที่สัมพันธ์กัน, และเคียวรี้อยดังกล่าวเรียกว่า เคียวรี้อยที่สัมพันธ์กัน.

สิ่งอ้างอิงที่เกี่ยวข้อง

“นิพจน์ใน WHERE clause” ในหน้า 45

นิพจน์ใน WHERE clause ใช้เพื่อระบุสิ่งที่ต้องการเปรียบเทียบกับสิ่งอื่น.

“การนิยามเงื่อนไขการค้นหาที่ซับซ้อน” ในหน้า 59

นอกจากเพรดิเคตการเปรียบเทียบพื้นฐานแล้ว (=, >, <, และอื่นๆ), เงื่อนไขการค้นหาสามารถมีเพรดิเคต BETWEEN, IN, EXISTS, IS NULL, และ LIKE.

เคียวรี้อยในคำสั่ง SELECT

เคียวรี้อยสามารถช่วยคุณกรองเงื่อนไขการค้นหาได้เพิ่มเติม.

ใน WHERE และ HAVING clause แบบง่ายๆ, คุณสามารถระบุเงื่อนไขการค้นหาด้วยการใช้ค่า literal, ชื่อคอลัมน์, นิพจน์, หรือ register พิเศษ. ในเงื่อนไขการค้นหาเหล่านั้น, คุณทราบว่าคุณกำลังค้นหาเฉพาะ. แต่ในบางครั้ง, คุณไม่สามารถป้อนค่านั้นได้จนกว่าจะได้เรียกข้อมูลอื่นๆ ออกมาจากตารางก่อน. ตัวอย่างเช่น, สมมติว่าคุณต้องการรายการหมายเลขพนักงาน, ชื่อ, และไค้ตงานของพนักงานทั้งหมดที่ทำงานในแต่ละโครงการ, เช่นหมายเลขโครงการ MA2100. คุณสามารถเขียนส่วนแรกของคำสั่งได้อย่างง่ายดายดังนี้:

```
SELECT EMPNO, LASTNAME, JOB
FROM CORPDATA.EMPLOYEE
WHERE EMPNO ...
```

แต่คุณไม่สามารถดำเนินการต่อไปได้เนื่องจากตาราง CORPDATA.EMPLOYEE ไม่ได้รวมข้อมูลหมายเลขโครงการไว้ด้วย. คุณไม่ทราบว่าพนักงานคนใดทำงานโครงการ MA2100 อยู่ หากไม่ได้ใช้คำสั่ง SELECT อีกคำสั่งหนึ่งให้กับตาราง CORPDATA.EMP_ACT.

ด้วยการใช้ SQL, คุณสามารถซ้อนคำสั่ง SELECT หนึ่งคำสั่งภายในอีกหนึ่งคำสั่งเพื่อแก้ปัญหานี้. คำสั่ง SELECT ภายในเรียกว่า การสืบค้นย่อย. คำสั่ง SELECT ที่อยู่นอกการสืบค้นย่อยเรียกว่า **outer-level SELECT**. เมื่อใช้การสืบค้นย่อย, คุณสามารถใช้คำสั่ง SQL เพียงคำสั่งเดียวในการดึงข้อมูลหมายเลขพนักงาน, ชื่อ, และไค้งานของพนักงานซึ่งทำงานโครงการ MA2100 ได้พร้อมกัน :

```
SELECT EMPNO, LASTNAME, JOB
FROM CORPDATA.EMPLOYEE
WHERE EMPNO IN
  (SELECT EMPNO
   FROM CORPDATA.EMPPROJECT
   WHERE PROJNO = 'MA2100')
```

เพื่อให้เกิดความเข้าใจยิ่งขึ้นว่าจะเกิดผลลัพธ์อะไรขึ้นจากคำสั่ง SQL, ให้จินตนาการว่า SQL เป็นไปตามกระบวนการต่อไปนี้:

ขั้นที่ 1: SQL จะประเมินผลเค็ยวรีย่อยเพื่อรับรายการค่า EMPNO:

```
(SELECT EMPNO
 FROM CORPDATA.EMPPROJECT
 WHERE PROJNO= 'MA2100')
```

ซึ่งจะทำให้ได้ผลลัพธ์เป็นตารางชั่วคราว:

EMPNO from CORPDATA.EMPPROJECT

000010

000110

ขั้นที่ 2: จากนั้นตารางผลลัพธ์ชั่วคราวจะทำหน้าที่เป็นเสมือนรายการในเงื่อนไขการค้นหาของ outer-level SELECT. ซึ่งมีความสำคัญ, เนื่องจากคำสั่งนี้เป็นคำสั่งที่รันอยู่.

```
SELECT EMPNO, LASTNAME, JOB
FROM CORPDATA.EMPLOYEE
WHERE EMPNO IN
  ('000010', '000110')
```

ตารางผลลัพธ์ขั้นสุดท้ายจะเป็นดังนี้:

EMPNO	LASTNAME	JOB
000010	HAAS	PRES
000110	LUCCHESI	SALESREP

เค็ยวรีย่อยและเงื่อนไขการค้นหา:

เค็ยวรีย่อยอาจเป็นส่วนหนึ่งของเงื่อนไขการค้นหา.

ซึ่งจะอยู่ในรูป *operand operator operand*. ทั้งนี้ operand อาจเป็นเค็ยวรีย่อยก็ได้. ดังตัวอย่างต่อไปนี้, **operand** แรกคือ EMPNO และ **operator** คือ IN. เงื่อนไขการค้นหาอาจเป็นส่วนหนึ่งของ WHERE หรือ HAVING clause. ในแต่ละ clause อาจ

ประกอบด้วยเงื่อนไขการค้นหาที่มีเคียวรี้อย่อยมากกว่าหนึ่งเงื่อนไข. เงื่อนไขการค้นหาที่มีเคียวรี้อย่อยอยู่, นั้นอาจอยู่ระหว่างวงเล็บ, นำหน้าด้วยคีย์เวิร์ด NOT, หรือเชื่อมโยงไปยังเงื่อนไข การค้นหาอื่นๆ โดยใช้คีย์เวิร์ด AND และ OR, เช่นเดียวกันกับเงื่อนไขการค้นหาอื่นๆ. ตัวอย่าง, WHERE clause ของเคียวรี้อาจมีลักษณะดังนี้:

```
WHERE (subquery1) = X AND (Y > SOME (subquery2) OR Z = 100)
```

เคียวรี้อย่อยนั้นอาจปรากฏอยู่ในเงื่อนไขการค้นหาของเคียวรี้อย่อยอื่นๆ. ลักษณะดังกล่าวเรียกว่าเคียวรี้อย่อยแบบซ้อนภายในที่อาจเกิดขึ้นในการซ้อนภายในบางระดับ. ตัวอย่างเช่น, เคียวรี้อย่อยซึ่งอยู่ภายในเคียวรี้อย่อยใน outer-level SELECT จะถูกซ้อนอยู่ในระดับที่สอง. คำสั่ง SQL ยอมให้มีการซ้อนภายในได้ทั้งสิ้น 32 ระดับ.

หมายเหตุการใช้บนเคียวรี้อย่อย:

เมื่อใช้เคียวรี้อย่อย, คุณควรทราบถึงหมายเหตุการใช้เหล่านี้.

1. เมื่อทำการซ้อนภายในคำสั่ง SELECT, คุณสามารถใช้เคียวรี้อย่อยได้หลายๆ ครั้งตามต้องการ (1 ถึง 255 เคียวรี้อย่อย), แม้การเพิ่มเคียวรี้อย่อยขึ้นมาแต่ละครั้งจะทำให้ระบบทำงานช้าลง.
2. สำหรับเพรดิเคตที่ใช้คีย์เวิร์ด ALL, ANY, SOME, หรือ EXISTS, จำนวนแถวที่ถูกส่งคืนจากเคียวรี้อย่อยสามารถเป็นได้ตั้งแต่ค่าศูนย์จนถึงค่าหลายๆ ค่า. สำหรับเคียวรี้อย่อยอื่นๆ ทั้งหมด, จำนวนแถวที่ถูกส่งคืนจะต้องเป็นศูนย์หรือหนึ่ง.
3. สำหรับเพรดิเคตต่อไปนี้, การเลือกแถวแบบเต็มสามารถใช้กับเคียวรี้อย่อยได้. นี่หมายความว่า เคียวรี้อย่อยสามารถส่งคืนค่าได้มากกว่าหนึ่งค่าสำหรับแถว.
 - เพรดิเคตพื้นฐานกับการเปรียบเทียบที่เท่ากันและไม่เท่ากัน
 - เพรดิเคตที่เป็นจำนวนโดยใช้ =ANY, =ALL, และ =SOME
 - เพรดิเคต IN และ NOT INถ้าใช้การเลือกแถวแบบเต็ม:
 - รายการที่เลือกต้องไม่มี SELECT *. ซึ่งต้องระบุค่าที่แน่นอน.
 - การเลือกแถวแบบเต็มต้องถูกเปรียบเทียบกับนิพจน์แถว. นิพจน์แถว คือรายการของค่าที่อยู่ในวงเล็บ. ซึ่งต้องเป็นจำนวนของค่าเดียวกันกับค่าที่ส่งคืนจากเคียวรี้อย่อยซึ่งอยู่ในนิพจน์แถว.
 - นิพจน์แถวสำหรับเพรดิเคต IN หรือ NOT IN ไม่สามารถมีตัวทำเครื่องหมายพารามิเตอร์ที่ไม่ถูกพิมพ์. ใช้ CAST เพื่อจัดหาชนิดข้อมูลผลลัพธ์สำหรับตัวทำเครื่องหมายพารามิเตอร์เหล่านี้.
 - เคียวรี้อย่อยไม่สามารถมี UNION, EXCEPT, หรือ INTERSECT หรือการอ้างอิงที่เกี่ยวข้องกัน.
4. เคียวรี้อย่อยไม่สามารถมี ORDER BY, FOR READ ONLY, FETCH FIRST *n* ROWS, UPDATE, หรือ OPTIMIZE clauses.

การรวมเคียวรี้อย่อยเข้าไปใน WHERE หรือ HAVING:

มีหลายวิธีที่คุณสามารถรวมเคียวรี้อย่อยเข้าไปใน WHERE หรือ HAVING.

- การดำเนินการเปรียบเทียบพื้นฐาน
- การเปรียบเทียบเชิงปริมาณ (ALL, ANY, และ SOME)
- คีย์เวิร์ด IN
- คีย์เวิร์ด EXISTS

การดำเนินการเปรียบเทียบพื้นฐาน

- | คุณสามารถใช้เคียวรี้อยู่ก่อนหรือหลัง comparison operator ใดๆ ก็ได้. เคียวรี้อยู่สามารถส่งคืนค่าได้เพียงหนึ่งแถวเท่านั้น.
- | และสามารถส่งคืนค่าแถวได้หลายๆ ค่า หากใช้โอเปอเรเตอร์เท่ากับและไม่เท่ากับ. SQL จะเปรียบเทียบค่าแต่ละค่าจากแถว
- | ของเคียวรี้อยู่ด้วยค่าที่สอดคล้องกันกับค่าที่อยู่อีกด้านหนึ่งของ comparison operator. ตัวอย่างเช่น, เมื่อคุณต้องการค้นหา
- | หมายเลขพนักงาน, ชื่อ, และเงินเดือนสำหรับพนักงานที่มีระดับการศึกษาสูงกว่าระดับการศึกษาโดยเฉลี่ยทั่วไปในบริษัท.

```
| SELECT EMPNO, LASTNAME, SALARY
| FROM CORPDATA.EMPLOYEE
| WHERE EDLEVEL >
| (SELECT AVG(EDLEVEL)
| FROM CORPDATA.EMPLOYEE)
```

- | SQL จะประเมินผลเคียวรี้อยู่ก่อน จากนั้นจะแทนที่ผลลัพธ์ใน WHERE clause ของคำสั่ง SELECT. ดังตัวอย่าง, ผลลัพธ์ที่ได้
- | คือระดับการศึกษาโดยเฉลี่ย ของทั้งบริษัท. นอกจากการส่งคืนค่าเพียงแถวเดียวแล้ว, เคียวรี้อยู่อาจไม่ส่งคืนค่าเลยก็ได้.
- | หากเป็นเช่นนั้น, จะไม่ทราบผลลัพธ์ของการเปรียบเทียบ.

การเปรียบเทียบเชิงปริมาณ (ALL, ANY, และ SOME)

คุณสามารถใช้เคียวรี้อยู่ที่อยู่หลัง comparison operator ต่อด้วยคีย์เวิร์ด ALL, ANY, หรือ SOME. ด้วยวิธีนี้, เคียวรี้อยู่อาจไม่ส่งคืนค่าแถว, ส่งคืนหนึ่งแถว, หรือส่งคืนมากกว่าหนึ่งแถว, ซึ่งรวมถึงค่า null ด้วย. คุณสามารถใช้ ALL, ANY, และ SOME ดังวิธีต่อไปนี้:

- ใช้ ALL เพื่อแสดงว่า ค่าที่คุณป้อนต้องถูกนำไปเปรียบเทียบตามวิธีการข้างต้นกับแถวที่ได้รับจากเคียวรี้อยู่ที่เรียกว่า ALL. ตัวอย่างเช่น, สมมติว่าคุณใช้ comparison operator มากกว่า พร้อมกับ ALL:

```
... WHERE expression > ALL (subquery)
```

เพื่อให้เป็นไปตาม WHERE clause, ค่าของนิพจน์จะต้องมากกว่าผลลัพธ์สำหรับแต่ละแถว (กล่าวคือ, ต้องมากกว่าค่าสูงสุด) ที่ส่งคืนค่าโดยเคียวรี้อยู่. หากเคียวรี้อยู่ส่งคืนชุดที่ว่างเปล่า (กล่าวคือ, ไม่มีทางเลือกแถว), ก็จะถือว่าเป็นไปตามเงื่อนไขเช่นกัน.

- ใช้ ANY หรือ SOME เพื่อแสดงว่า ค่าที่คุณป้อนต้องนำไปเปรียบเทียบตามวิธีการข้างต้นกับแถวอย่างน้อยหนึ่งแถวที่เคียวรี้อยู่ส่งคืน. ตัวอย่างเช่น, ถ้าคุณใช้ comparison operator มากกว่า พร้อมกับ ANY:

```
... WHERE expression > ANY (subquery)
```

เพื่อให้เป็นไปตาม WHERE, ค่าในนิพจน์ต้องมากกว่าค่าที่ถูกส่งคืนจากเคียวรี้อยู่อย่างน้อยหนึ่งแถว (นั่นคือ, ต้องมากกว่าค่าที่ต่ำสุด). หากค่าที่เคียวรี้อยู่ส่งคืนคือชุดค่าที่ว่างเปล่า, ก็ถือว่าไม่เป็นไปตามเงื่อนไข.

หมายเหตุ: ผลลัพธ์ที่ได้เมื่อเคียวรี้อยู่ส่งคืนค่าศูนย์หนึ่งค่า หรือมากกว่า อาจทำให้คุณประหลาดใจ, เว้นแต่ว่า คุณคุ้นเคยกับตรรกะตามรูปแบบอยู่แล้ว.

คีย์เวิร์ด IN

คุณสามารถใช้ IN เพื่อแสดงว่า ค่าในนิพจน์ต้องอยู่ระหว่างแถวที่ส่งคืนโดยเคียวรี้อยู่. การใช้ IN เท่ากับการใช้ =ANY หรือ =SOME. ซึ่งได้อธิบายไว้ก่อนหน้านี้แล้ว. คุณยังสามารถใช้คีย์เวิร์ด IN กับคีย์เวิร์ด NOT เพื่อใช้เลือกแถว เมื่อค่าไม่มีอยู่ในแถวที่ส่งคืนโดยเคียวรี้อยู่. ตัวอย่างเช่น, คุณสามารถระบุ:

```
... WHERE WORKDEPT NOT IN (SELECT ...)
```

คีย์เวิร์ด EXISTS

ในเคียวรีย่อยที่นำเสนอมาตั้งแต่ต้น, SQL จะประเมินผลเคียวรีย่อยและใช้ผลลัพธ์เป็นส่วนหนึ่งของ WHERE clause ของ outer-level SELECT. ในทางตรงกันข้าม, เมื่อคุณใช้คีย์เวิร์ด EXISTS, SQL จะตรวจสอบว่าเคียวรีย่อยส่งคืนค่าตั้งแต่หนึ่งแถวขึ้นไปหรือไม่. หากเป็นเช่นนั้น, แสดงว่าเป็นไปตามเงื่อนไข. หากไม่ส่งคืนแถวเลย, แสดงว่าไม่เป็นไปตามเงื่อนไข. ตัวอย่างเช่น:

```
SELECT EMPNO, LASTNAME
FROM CORPDATA.EMPLOYEE
WHERE EXISTS
  (SELECT *
   FROM CORPDATA.PROJECT
   WHERE PRSTDATE > '1982-01-01');
```

ในตัวอย่าง, เงื่อนไขการค้นหาจะเป็นจริงหากโครงการใดๆ ที่นำเสนอในตาราง CORPDATA.PROJECT มีวันที่เริ่มต้นโดยประมาณซึ่งเป็นหลังวันที่ 1 มกราคม, 1982. โปรดสังเกตว่าตัวอย่างนี้ไม่ได้แสดงประสิทธิภาพที่สมบูรณ์แบบของ EXIST, เนื่องจากผลลัพธ์ที่ได้จะเป็นแบบเดียวกันเสมอสำหรับทุกแถวถ้าตรวจสอบด้วยคำสั่ง outer-level SELECT. ดังนั้น, ผลลัพธ์ที่ได้อาจประกอบข้อมูลทุกแถว, หรือไม่มีเลยแม้แต่แถวเดียว. สำหรับตัวอย่างที่ตึกกว่า, ตัวเคียวรีย่อยเองควรสัมพันธ์กัน, และเปลี่ยนจากแถวไปยังแถว.

ดังที่แสดงไว้ในตัวอย่าง, คุณไม่จำเป็นต้องระบุชื่อคอลัมน์ในรายการการเลือกของเคียวรีย่อยสำหรับ EXISTS clause. แต่, คุณควรใส่ SELECT *.

คุณยังสามารถใช้คีย์เวิร์ด EXISTS คู่กับคีย์เวิร์ด NOT เพื่อเลือกแถวเมื่อไม่มีข้อมูลหรือเงื่อนไขที่คุณระบุอยู่. คุณสามารถใช้คำสั่งต่อไปนี้:

```
... WHERE NOT EXISTS (SELECT ...)
```

เคียวรีย่อยที่สัมพันธ์กัน

คุณสามารถเขียนเคียวรีย่อยที่ SQL จำเป็นต้องใช้ประเมินผลขณะที่ตรวจสอบแถวใหม่แต่ละแถว (WHERE clause) หรือกลุ่มของแถว (HAVING clause) ใน outer-level SELECT. การทำเช่นนี้เรียกว่า *เคียวรีย่อยที่สัมพันธ์กัน*.

ชื่อและการอ้างอิงที่สัมพันธ์กัน:

การอ้างอิงที่สัมพันธ์กันอาจปรากฏในเงื่อนไขการค้นหาในการสืบค้นย่อย. การอ้างอิงจะอยู่ในรูปของ X.C เสมอ, โดยที่ X คือชื่อที่มีความหมายเหมือนกัน และ C คือชื่อของคอลัมน์ในตารางที่ X อ้างถึง.

คุณสามารถกำหนดชื่อที่มีความหมายเหมือนกันสำหรับตารางใดๆ ที่ปรากฏอยู่ใน FROM clause. ชื่อการสืบค้นที่สัมพันธ์กันจะถือเป็นชื่อเฉพาะของตารางในเคียวรี. ชื่อตารางเดียวกันสามารถใช้ได้หลายครั้งภายในเคียวรีและการเลือกย่อยแบบซ้อนภายในของตาราง. การระบุชื่อต่างๆ ที่มีความหมายเหมือนกันสำหรับการอ้างอิงตารางแต่ละชื่อ อาจเป็นการกำหนดตารางเฉพาะที่คอลัมน์อ้างอิงถึงได้.

ชื่อที่มีความหมายเหมือนกันจะถูกกำหนดใน FROM clause ของเคียวรี. เคียวรีนี้อาจเป็น outer-level SELECT, หรือเป็นเคียวรีย่อยใดๆ ที่มีเคียวรีที่อ้างอิง. ตัวอย่าง, สมมติว่า, เคียวรีประกอบด้วยเคียวรีย่อย A, B, และ C, ซึ่ง A ประกอบด้วย B และ B ประกอบด้วย C. ชื่อที่มีความหมายเหมือนกันที่ใช้ใน C ก็ควรถูกกำหนดใน B, A, หรือ outer-level SELECT เช่นกัน. หากต้องการกำหนดชื่อที่มีความหมายเหมือนกัน, ให้ใส่ชื่อที่มีความหมายเหมือนกันไว้หลังชื่อตาราง T. เว้นช่องว่างไว้หนึ่งช่องหรือมากกว่าระหว่างชื่อตารางและชื่อที่มีความหมายเหมือนกันของตาราง, และใส่เครื่องหมายจุลภาคไว้หลังชื่อที่มีความ

หมายเหตุเหมือนกัน หากชื่อนั้นตามด้วยชื่อตารางอีกชื่อหนึ่ง. FROM clause ต่อไปนี้เป็นการกำหนดชื่อที่สัมพันธ์กันคือ TA และ TB สำหรับตาราง TABLEA และ TABLEB, และไม่มีชื่อที่สัมพันธ์กันสำหรับตาราง TABLEC.

```
FROM TABLEA TA, TABLEC, TABLEB TB
```

เคียวรีย่อยอาจประกอบด้วยจำนวนการอ้างอิงที่สัมพันธ์กัน. ตัวอย่างเช่น, ชื่อที่สัมพันธ์กันในเงื่อนไขการค้นหาสามารถกำหนดใน outer-level SELECT, ขณะที่อีกชื่อหนึ่งสามารถกำหนดในเคียวรีย่อย.

ก่อนที่จะใช้งานเคียวรีย่อย, ค่าจากคอลัมน์ที่อ้างอิงจะถูกแทนที่สำหรับการอ้างอิงที่สัมพันธ์กันเสมอ.

ตัวอย่าง: เคียวรีย่อยที่สัมพันธ์กันใน WHERE clause:

สมมติว่าคุณต้องการรายชื่อพนักงานทั้งหมดที่มีระดับการศึกษาสูงกว่าระดับการศึกษาโดยเฉลี่ยในแต่ละแผนก. ถ้าต้องการได้ข้อมูลนี้, SQL ต้องค้นหาตาราง CORPDATA.EMPLOYEE.

ในส่วนของพนักงานแต่ละคนในตาราง, SQL ต้องเปรียบเทียบระดับการศึกษาของพนักงานกับระดับการศึกษาโดยเฉลี่ยสำหรับแผนกของพนักงาน. ในการเคียวรีย่อย, คุณแจ้งให้ SQL ค้นหาระดับการศึกษาโดยเฉลี่ยสำหรับหมายเลขแผนกในแถวปัจจุบัน. ตัวอย่างเช่น:

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
FROM CORPDATA.EMPLOYEE X
WHERE EDLEVEL >
  (SELECT AVG(EDLEVEL)
   FROM CORPDATA.EMPLOYEE
   WHERE WORKDEPT = X.WORKDEPT)
```

เคียวรีย่อยที่สัมพันธ์กันจะคล้ายกับเคียวรีย่อยที่ไม่สัมพันธ์กัน, ยกเว้นการปรากฏขึ้นของการอ้างอิงที่สัมพันธ์กันหนึ่งครั้งหรือมากกว่า. ในตัวอย่าง, การอ้างอิงเดียวที่สัมพันธ์กันคือ X.WORKDEPT ใน FROM clause ของการเลือกย่อย. ในที่นี้, qualifier X คือชื่อที่มีความหมายเหมือนกันที่ถูกกำหนดไว้ใน FROM clause ของคำสั่ง outer SELECT. ใน clause ดังกล่าว, X จะเป็นชื่อที่สัมพันธ์กันของตาราง CORPDATA.EMPLOYEE.

ตอนนี้, ให้พิจารณาว่าจะเกิดอะไรขึ้นเมื่อเรียกใช้งานเคียวรีย่อยสำหรับแถว CORPDATA.EMPLOYEE ที่ให้มา. ก่อนที่จะเรียกใช้งาน, X.WORKDEPT จะถูกแทนที่ด้วยค่าของคอลัมน์ WORKDEPT สำหรับแถวนั้น. ตัวอย่าง, สมมติว่า, แถวดังกล่าวคือแถวสำหรับ CHRISTINE I HAAS. แผนกงานของเธอคือ A00, ซึ่งเท่ากับค่า WORKDEPT สำหรับแถวนั้น. เคียวรีย่อยที่ถูกเรียกใช้งานสำหรับแถวนั้นคือ:

```
(SELECT AVG(EDLEVEL)
 FROM CORPDATA.EMPLOYEE
 WHERE WORKDEPT = 'A00')
```

ดังนั้น, สำหรับแถวที่ถูกพิจารณา, เคียวรีย่อยจะแสดงระดับการศึกษาโดยเฉลี่ยของแผนกของ Christine. จากนั้น คำนี้นจะถูกนำไปเปรียบเทียบในคำสั่ง outer กับระดับการศึกษาของ Christine. สำหรับแถวอื่นบางแถวที่ WORKDEPT มีค่าอื่น, คำนี้นจะถูกปรากฏขึ้นในเคียวรีย่อยแทนที่ A00. ตัวอย่างเช่น, ในแถวสำหรับ MICHAEL L THOMPSON, คำนี้นคือ B01, และเคียวรีย่อยสำหรับแถวของเขาจะแสดงระดับการศึกษาโดยเฉลี่ยสำหรับแผนก B01.

ตารางผลลัพธ์จากเคียวรีมีค่าดังต่อไปนี้:

ตารางที่ 25. ชุดผลลัพธ์สำหรับเคียวรีก่อนหน้า

EMPNO	LASTNAME	WORKDEPT	EDLEVEL
000010	HAAS	A00	18
000030	KWAN	C01	20
000070	PULASKI	D21	16
000090	HENDERSON	E11	16
000110	LUCCHESI	A00	19
000160	PIANKA	D11	17
000180	SCOUTTEN	D11	17
000210	JONES	D11	17
000220	LUTZ	D11	18
000240	MARINO	D21	17
000260	JOHNSON	D21	16
000280	SCHNEIDER	E11	17
000320	MEHTA	E21	16
000340	GOUNOT	E21	16
200010	HEMMINGER	A00	18
200220	JOHN	D11	18
200240	MONTEVERDE	D21	17
200280	SCHWARTZ	E11	17
200340	ALONZO	E21	16

ตัวอย่าง: เคียวรีย่อยที่สัมพันธ์กันใน **HAVING** clause:

สมมติว่าคุณต้องการรายการแผนกทั้งหมดซึ่งมีเงินเดือนโดยเฉลี่ยสูงกว่าเงินเดือนโดยเฉลี่ยของหน่วยงาน ในส่วนนี้ (แผนกทั้งหมดที่ **WORKDEPT** เริ่มต้นด้วยตัวอักษรเดียวกันจะอยู่ในส่วนเดียวกัน). ถ้าต้องการได้ข้อมูลนี้, **SQL** ต้องค้นหาตาราง **CORPDATA.EMPLOYEE**.

สำหรับแต่ละแผนกในตาราง, **SQL** จะเปรียบเทียบเงินเดือนโดยเฉลี่ยของแผนกกับเงินเดือนโดยเฉลี่ยของหน่วยงาน. ในเคียวรีย่อย, **SQL** จะคำนวณเงินเดือนโดยเฉลี่ยสำหรับหน่วยงานของแผนกในกลุ่มปัจจุบัน. ตัวอย่างเช่น:

```
SELECT WORKDEPT, DECIMAL(AVG(SALARY),8,2)
FROM CORPDATA.EMPLOYEE X
GROUP BY WORKDEPT
HAVING AVG(SALARY) >
```

```
(SELECT AVG(SALARY)
FROM CORPDATA.EMPLOYEE
WHERE SUBSTR(X.WORKDEPT,1,1) = SUBSTR(WORKDEPT,1,1))
```

ให้พิจารณาว่าจะเกิดอะไรขึ้นเมื่อรันเคิวรี่ย่อยของแผนก CORPDATA.EMPLOYEE. ก่อนที่จะถูกรัน, X.WORKDEPT จะถูกแทนที่ด้วยค่าของคอลัมน์ WORKDEPT สำหรับกลุ่มดังกล่าว. สมมติว่า, ตัวอย่างเช่น, กลุ่มแรกที่ถูกเลือกค่า WORKDEPT เป็น A00. เคิวรี่ย่อยที่ถูกเรียกใช้งานสำหรับกลุ่มนี้คือ:

```
(SELECT AVG(SALARY)
FROM CORPDATA.EMPLOYEE
WHERE SUBSTR('A00',1,1) = SUBSTR(WORKDEPT,1,1))
```

ดังนั้น, สำหรับกลุ่มที่ถูกพิจารณา, เคิวรี่ย่อยจะแสดงเงินเดือนโดยเฉลี่ยสำหรับหน่วยงาน. จากนั้น ค่านี้จะถูกนำไปเปรียบเทียบในคำสั่ง outer กับเงินเดือนโดยเฉลี่ยของแผนก 'A00'. สำหรับกลุ่มอื่นที่ WORKDEPT เท่ากับ 'B01', เคิวรี่ย่อยจะส่งค่าเงินเดือนเฉลี่ยสำหรับ หน่วยงานที่มีแผนก B01 อยู่ด้วย.

ตารางผลลัพธ์จากเคิวรี่มีค่าดังต่อไปนี้:

WORKDEPT	AVG SALARY
D21	25668.57
E01	40175.00
E21	24086.66

ตัวอย่าง: การสืบค้นย่อยที่สัมพันธ์กันในรายการสำหรับเลือก:

สมมติว่า คุณต้องการรายการของแผนกทั้งหมด, รวมถึงชื่อแผนก, หมายเลข, และชื่อผู้จัดการแผนก.

ชื่อและหมายเลขแผนกหาได้จากตาราง CORPDATA.DEPARTMENT. อย่างไรก็ตาม, DEPARTMENT มีเฉพาะหมายเลขผู้จัดการแผนก แต่ไม่มีชื่อผู้จัดการแผนก. เพื่อค้นหาชื่อผู้จัดการแต่ละแผนก, คุณต้องค้นหาหมายเลขพนักงานจากตาราง EMPLOYEE ที่ตรงกับหมายเลขผู้จัดการในตาราง DEPARTMENT และส่งคืนแถวที่ตรงกัน. โดยจะส่งคืนเฉพาะแผนกที่มีผู้จัดการประจำอยู่เท่านั้นในปัจจุบัน. รันคำสั่งต่อไปนี้:

```
SELECT DEPTNO, DEPTNAME,
       (SELECT FIRSTNAME CONCAT ' ' CONCAT
        MIDINIT CONCAT ' ' CONCAT LASTNAME
        FROM EMPLOYEE X
        WHERE X.EMPNO = Y.MGRNO) AS MANAGER_NAME
FROM DEPARTMENT Y
WHERE MGRNO IS NOT NULL
```

แต่ละแถวที่ถูกส่งคืนสำหรับ DEPTNO และ DEPTNAME, ระบบจะค้นหา EMPNO = MGRNO และส่งคืนชื่อผู้จัดการ. ตารางผลลัพธ์จากการสืบค้นมีค่าดังต่อไปนี้:

ตารางที่ 26. ชุดผลลัพธ์สำหรับการสืบค้นก่อนหน้า

DEPTNO	DEPTNAME	MANAGER_NAME
A00	SPIFFY COMPUTER SERVICE DIV.	CHRISTINE I HAAS

ตารางที่ 26. ชุดผลลัพธ์สำหรับการสืบค้นก่อนหน้า (ต่อ)

DEPTNO	DEPTNAME	MANAGER_NAME
B01	PLANNING	MICHAEL L THOMPSON
C01	INFORMATION CENTER	SALLY A KWAN
D11	MANUFACTURING SYSTEMS	IRVING F STERN
D21	ADMINISTRATION SYSTEMS	EVA D PULASKI
E01	SUPPORT SERVICES	JOHN B GEYER
E11	OPERATIONS	EILEEN W HENDERSON
E21	SOFTWARE SUPPORT	THEODORE Q SPENSER

ตัวอย่าง: เคียวย่อยที่สัมพันธ์กันในคำสั่ง UPDATE:

เมื่อคุณใช้เคียวย่อยที่สัมพันธ์กันในคำสั่ง UPDATE, ชื่อที่มีความหมายเหมือนกันจะหมายถึงแถว ซึ่งคุณสนใจที่จะอัปเดต.

ตัวอย่างเช่น, หากกิจกรรมทั้งหมดของโครงการต้องทำให้เสร็จสมบูรณ์ก่อนเดือนกันยายน 1983, แผนกของคุณจะพิจารณาว่าโครงการนั้นคือโครงการที่มีความสำคัญในอันดับต้นๆ. คุณสามารถใช้คำสั่ง SQL ข้างล่างนี้เพื่อประเมินผลโครงการในตาราง CORPDATA.PROJECT, และใส่ค่า 1 (แฟล็กแสดงถึงระดับความสำคัญ) ในคอลัมน์ PRIORITY (คอลัมน์ที่คุณเพิ่มให้กับ CORPDATA.PROJECT สำหรับวัตถุประสงค์นี้) สำหรับแต่ละโครงการที่มีความสำคัญ.

```
UPDATE CORPDATA.PROJECT X
SET PRIORITY = 1
WHERE '1983-09-01' >
      (SELECT MAX(EMENDATE)
       FROM CORPDATA.EMPPROJECT
       WHERE PROJNO = X.PROJNO)
```

เมื่อ SQL ตรวจสอบแต่ละแถวในตาราง CORPDATA.EMPPROJECT, จะกำหนดวันที่สิ้นสุดกิจกรรมเป็นอย่างมากที่สุด (EMENDATE) สำหรับกิจกรรมทั้งหมดของโครงการ (จากตาราง CORPDATA.PROJECT). หากวันที่สิ้นสุดของแต่ละกิจกรรมที่เกี่ยวข้องกับโครงการ คือ วันก่อนเดือนกันยายน 1983, แถวปัจจุบันในตาราง CORPDATA.PROJECT จะเป็นไปตามเกณฑ์และถูกอัปเดต.

อัปเดตตารางลำดับหลักเมื่อเกิดการเปลี่ยนแปลงใดๆ ขึ้นกับปริมาณตามลำดับ. หากไม่ได้เซตปริมาณในตารางลำดับ (ค่า NULL), ให้เก็บค่าที่อยู่ในตารางลำดับหลักเอาไว้.

```
UPDATE MASTER_ORDERS X
SET QTY=(SELECT COALESCE (Y.QTY, X.QTY)
         FROM ORDERS Y
         WHERE X.ORDER_NUM = Y.ORDER_NUM)
WHERE X.ORDER_NUM IN (SELECT ORDER_NUM
                      FROM ORDERS)
```

ในตัวอย่างนี้, แต่ละแถวของตาราง MASTER_ORDERS จะถูกตรวจสอบเพื่อดูว่ามีแถวที่ตรงกันในตาราง ORDERS หรือไม่. หากมีแถวที่ตรงกันในตาราง ORDERS, ฟังก์ชัน COALESCE จะถูกนำมาใช้เพื่อส่งคืนค่าสำหรับคอลัมน์ QTY. หาก QTY

ในตาราง ORDERS มีค่าที่ไม่ใช่ null, ค่านั้นจะถูกนำไปใช้เพื่ออัปเดตคอลัมน์ QTY ในตาราง MASTER_ORDERS. หากค่า QTY ในตาราง ORDERS เท่ากับ NULL, คอลัมน์ MASTER_ORDERS QTY จะถูกอัปเดตด้วยค่าของมันเอง.

ตัวอย่าง: การสืบค้นย่อยที่สัมพันธ์กันในคำสั่ง DELETE:

เมื่อคุณใช้การสืบค้นย่อยที่สัมพันธ์กันในคำสั่ง DELETE, ชื่อที่มีความหมายเหมือนกันจะหมายถึงแถวที่คุณลบออกไป. SQL จะประเมินการสืบค้นย่อยที่สัมพันธ์กันหนึ่งครั้งสำหรับแต่ละแถวในตารางที่ปรากฏชื่อในคำสั่ง DELETE เพื่อตัดสินใจว่าจะลบแถวนั้นออกหรือไม่.

สมมติว่าแถวในตาราง CORPDATA.PROJECT ถูกลบออก. แถวที่เกี่ยวข้องกับโครงการที่ถูกลบออกในตาราง CORPDATA.EMPPROJECT จะต้องถูกลบออกด้วย. การดำเนินการข้างต้น, คุณสามารถใช้:

```
DELETE FROM CORPDATA.EMPPROJECT X
WHERE NOT EXISTS
  (SELECT *
   FROM CORPDATA.PROJECT
   WHERE PROJNO = X.PROJNO)
```

SQL จะตัดสินใจว่า, แต่ละแถวในตาราง CORPDATA.EMP_ACT, แถวที่มีหมายเลขโครงการเดียวกันมีอยู่แล้วในตาราง CORPDATA.PROJECT หรือไม่. หากไม่มีอยู่, แถว CORPDATA.EMP_ACT จะถูกลบทิ้ง.

ลำดับการเรียง และ normalization ใน SQL

ลำดับการจัดเรียงกำหนดความสัมพันธ์ของอักขระในชุดอักขระ เมื่อมีการเปรียบเทียบหรือจัดลำดับ. Normalization อนุญาตให้คุณเปรียบเทียบสตริงที่มีอักขระแบบผสม.

ลำดับการจัดเรียงใช้สำหรับอักขระทั้งหมดและการเปรียบเทียบกราฟิก UCS-2 และ UTF-16 ในคำสั่ง SQL. มีตารางลำดับการจัดเรียงสำหรับข้อมูลอักขระทั้งแบบไบต์เดียว และไบต์คู่. ตารางลำดับการจัดเรียงแบบไบต์เดียวแต่ละตารางจะมีตารางลำดับการจัดเรียงแบบไบต์คู่ที่สัมพันธ์กันเสมอ. การแปลงค่าระหว่างสองตารางจะเริ่มขึ้นเมื่อจำเป็นต้องดำเนินการสืบค้น. นอกจากนี้, คำสั่ง CREATE INDEX มีลำดับการจัดเรียง (มีผลเมื่อมีการรันคำสั่ง) ซึ่งใช้กับคอลัมน์อักขระที่อ้างอิงถึงในดรรชนี.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การสร้างและใช้มุมมอง” ในหน้า 35

สามารถใช้มุมมองเพื่อเข้าใช้งานข้อมูลในตารางหนึ่งตารางหรือมากกว่าหรือมุมมองหนึ่งมุมมองหรือมากกว่าได้. คุณสามารถสร้างมุมมองได้โดยใช้คำสั่ง SELECT.

“การเพิ่มดรรชนี” ในหน้า 40

คุณสามารถใช้ดรรชนี เพื่อเรียงลำดับ และเลือกข้อมูล. นอกจากนี้, ดรรชนียังช่วย ระบบให้เรียกข้อมูลออกมาได้เร็วขึ้นเพื่อประสิทธิภาพของการสอบถามที่ดีกว่าเดิม.

“การระบุเงื่อนไขการค้นหาโดยใช้ WHERE clause” ในหน้า 44

WHERE clause จะระบุเงื่อนไขการค้นหาที่บ่งชี้ถึงแถว หรือแถวที่คุณต้องการดึงค่า, อัปเดต, หรือลบ.

“GROUP BY clause” ในหน้า 47

GROUP BY clause อนุญาตให้คุณค้นหาคุณลักษณะของกลุ่มของแถวมากกว่าที่จะค้นหาแถวเดียว.

“HAVING clause” ในหน้า 50

HAVING clause ระบุเงื่อนไขการค้นหาสำหรับกลุ่มที่เลือกได้โดยใช้ GROUP BY clause.

“ORDER BY clause” ในหน้า 51

ORDER BY ระบุลำดับเฉพาะที่คุณต้องการส่งคืนแถวที่เลือกไว้ ลำดับจะถูกเรียงลำดับการเรียงของค่าคอลัมน์หรือค่านิพจน์จากน้อยไปมาก หรือจากมากไปน้อย.

“การจัดการกับแถวซ้ำ” ในหน้า 58

เมื่อ SQL ประเมินผลคำสั่ง select, หลายแถวอาจจะมีคุณสมบัติพอที่จะอยู่ในตารางผลลัพธ์, ขึ้นอยู่กับจำนวนของแถวที่ตรงกับเงื่อนไขการค้นหาของคำสั่ง select. บางแถวในตารางผลลัพธ์อาจซ้ำกันได้.

“การนิยามเงื่อนไขการค้นหาที่ซับซ้อน” ในหน้า 59

นอกจากเพรดิเคตการเปรียบเทียบพื้นฐานแล้ว (=, >, <, และอื่นๆ), เงื่อนไขการค้นหาสามารถมีเพรดิเคต BETWEEN, IN, EXISTS, IS NULL, และ LIKE.

“การใช้คีย์เวิร์ด UNION เพื่อรวมการเลือกย่อย” ในหน้า 82

การใช้คีย์เวิร์ด UNION, คุณสามารถรวมการเลือกย่อยมากกว่าสองได้ เพื่อสร้าง fullselect.

ข้อมูลที่เกี่ยวข้อง

เรียงลำดับ

ลำดับการจัดเรียงที่ใช้กับ ORDER BY และการเลือกแถว

หากต้องการดูวิธีการใช้ลำดับการจัดเรียง, ให้รันตัวอย่างในส่วนนี้กับตาราง STAFF ที่แสดงในหัวข้อนี้.

โปรดสังเกตว่าค่าใน คอลัมน์ JOB จะมีทั้งตัวพิมพ์ใหญ่และเล็ก. คุณสามารถดูค่า 'Mgr', 'MGR', และ 'mgr'.

ตารางที่ 27. ตาราง STAFF

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	0
20	Pernal	20	Sales	8	18171.25	612.45
30	Merenghi	38	MGR	5	17506.75	0
40	OBrien	38	Sales	6	18006.00	846.55
50	Hanes	15	Mgr	10	20659.80	0
60	Quigley	38	SALES	0	16808.30	650.25
70	Rothman	15	Sales	7	16502.83	1152.00
80	James	20	Clerk	0	13504.60	128.20
90	Koonitz	42	sales	6	18001.75	1386.70
100	Plotz	42	mgr	6	18352.80	0

ในตัวอย่างต่อไปนี้, ผลลัพธ์จะแสดงสำหรับคำสั่งแต่ละคำสั่งโดยใช้:

- ลำดับการจัดเรียง *HEX
- ลำดับการจัดเรียงแบบเฉลี่ยน้าหนักโดยใช้ language identifier ENU
- ลำดับการจัดเรียงแบบน้ำหนักเฉพาะโดยใช้ language identifier ENU

หมายเหตุ: ENU ถูกเลือกเป็น language identifier โดยระบุ SRTSEQ(*LANGIDUNQ), หรือ SRTSEQ(*LANGIDSHR) และ LANGID(ENU), บนคำสั่ง CRTSQLxxx, STRSQL, หรือ RUNSQLSTM, หรือโดยใช้คำสั่ง SET OPTION.

ลำดับการจัดเรียงและ ORDER BY

ดูลำดับการจัดเรียงที่ทำงานด้วย ORDER BY.

คำสั่ง SQL ต่อไปนี้ทำให้ตารางผลลัพธ์ถูกจัดเรียงโดยใช้ค่าในคอลัมน์ JOB:

```
SELECT * FROM STAFF ORDER BY JOB
```

ตารางต่อไปนี้จะแสดงผลลัพธ์โดยใช้ลำดับการจัดเรียง *HEX. แถวต่างๆ ถูกจัดเรียงโดยยึดตามค่า EBCDIC ในคอลัมน์ JOB. ในกรณีนี้, อักษรตัวพิมพ์เล็กทั้งหมดจะจัดเรียงก่อนอักษรตัวพิมพ์ใหญ่.

ตารางที่ 28. ผลลัพธ์ของการใช้ลำดับการเรียง *HEX

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
100	Plotz	42	mgr	6	18352.80	0
90	Koonitz	42	sales	6	18001.75	1386.70
80	James	20	Clerk	0	13504.60	128.20
10	Sanders	20	Mgr	7	18357.50	0
50	Hanes	15	Mgr	10	20659.80	0
30	Merenghi	38	MGR	5	17506.75	0
20	Pernal	20	Sales	8	18171.25	612.45
40	OBrien	38	Sales	6	18006.00	846.55
70	Rothman	15	Sales	7	16502.83	1152.00
60	Quigley	38	SALES	0	16808.30	650.25

ตารางต่อไปนี้จะแสดงวิธีการจัดเรียงสำหรับลำดับการจัดเรียงแบบน้ำหนักเฉพาะ. หลังจากใช้ลำดับการจัดเรียงกับค่าในคอลัมน์ JOB, แถวต่างๆ จะถูกจัดเรียง. โปรดสังเกตว่า หลังจากการจัดเรียง, อักษรตัวพิมพ์เล็กจะมาก่อนตัวอักษรตัวเดียวกันที่เป็นอักษรตัวพิมพ์ใหญ่, และค่า 'mgr', 'Mgr', และ 'MGR' จะอยู่ติดกัน.

ตารางที่ 29. ผลลัพธ์ของการใช้ลำดับการจัดเรียงแบบน้ำหนักเฉพาะสำหรับ ENU language identifier

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
80	James	20	Clerk	0	13504.60	128.20
100	Plotz	42	mgr	6	18352.80	0
10	Sanders	20	Mgr	7	18357.50	0
50	Hanes	15	Mgr	10	20659.80	0
30	Merenghi	38	MGR	5	17506.75	0

ตารางที่ 29. ผลลัพธ์ของการใช้ลำดับการจัดเรียงแบบน้ำหนักเฉพาะสำหรับ ENU language identifier (ต่อ)

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
90	Koonitz	42	sales	6	18001.75	1386.70
20	Pernal	20	Sales	8	18171.25	612.45
40	OBrien	38	Sales	6	18006.00	846.55
70	Rothman	15	Sales	7	16502.83	1152.00
60	Quigley	38	SALES	0	16808.30	650.25

ตารางต่อไปนี้จะแสดงวิธีการจัดเรียงสำหรับลำดับการจัดเรียงแบบน้ำหนักเฉลี่ย. หลังจากใช้ลำดับการจัดเรียงกับค่าในคอลัมน์ JOB, แถวต่างๆ จะถูกจัดเรียง. สำหรับการเปรียบเทียบการจัดเรียง, อักษรตัวพิมพ์เล็กแต่ละตัว จะถือว่าเหมือนกับอักษรตัวพิมพ์ใหญ่ที่ตรงกัน. ในตารางนี้, โปรดสังเกตว่า ค่า 'MGR', 'mgr' และ 'Mgr' ทั้งหมดถูกรวมไว้ด้วยกัน.

ตารางที่ 30. ผลลัพธ์ของการใช้ลำดับการจัดเรียงแบบเฉลี่ยสำหรับ ENU language identifier

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
80	James	20	Clerk	0	13504.60	128.20
10	Sanders	20	Mgr	7	18357.50	0
30	Merenghi	38	MGR	5	17506.75	0
50	Hanes	15	Mgr	10	20659.80	0
100	Plotz	42	mgr	6	18352.80	0
20	Pernal	20	Sales	8	18171.25	612.45
40	OBrien	38	Sales	6	18006.00	846.55
60	Quigley	38	SALES	0	16808.30	650.25
70	Rothman	15	Sales	7	16502.83	1152.00
90	Koonitz	42	sales	6	18001.75	1386.70

การเลือกแถว

โปรดดูลำดับการจัดเรียงที่ทำงานกับการเลือกแถว.

คำสั่ง SQL ต่อไปนี้จะเลือกแถวที่มีค่า 'MGR' ในคอลัมน์ JOB:

```
SELECT * FROM STAFF WHERE JOB='MGR'
```

ตารางแรกแสดงวิธีการเลือกแถวที่มีลำดับการจัดเรียงแบบ *HEX. แถวที่ตรงตามเกณฑ์การเลือกแถวสำหรับคอลัมน์ JOB จะถูกเลือกตามที่ระบุไว้ในคำสั่ง select. เฉพาะ 'MGR' ที่เป็นตัวพิมพ์ใหญ่เท่านั้นที่ถูกเลือก.

ตารางที่ 31. ผลลัพธ์ของการใช้ลำดับการเรียง *HEX

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
30	Merenghi	38	MGR	5	17506.75	0

ตาราง 2 แสดงวิธีการเลือกแถวที่มีลำดับการจัดเรียงแบบน้ำหนักเฉพาะ. ตัวอักษรตัวพิมพ์เล็กและตัวพิมพ์ใหญ่จะมีลักษณะเฉพาะ. 'mgr' ที่เป็นตัวพิมพ์เล็กจะถือว่าไม่เหมือนกับ 'MGR' ที่เป็นตัวพิมพ์ใหญ่. ดังนั้น, 'mgr' ที่เป็นตัวพิมพ์เล็กจะไม่ถูกเลือก.

ตารางที่ 32. ผลลัพธ์ของการใช้ลำดับการจัดเรียงแบบน้ำหนักเฉพาะสำหรับ ENU language identifier

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
30	Merenghi	38	MGR	5	17506.75	0

ตารางต่อไปนี้จะแสดงวิธีการเลือกแถวที่มีลำดับการจัดเรียงแบบน้ำหนักเฉลี่ย. แถวที่ตรงตามเกณฑ์การเลือกสำหรับคอลัมน์ 'JOB' จะถูกเลือกโดยถือว่าตัวอักษรตัวพิมพ์ใหญ่เหมือนกับตัวอักษรตัวพิมพ์เล็ก. โปรดสังเกตว่า ค่า 'mgr', 'Mgr' และ 'MGR' ทั้งหมดจะถูกเลือก.

ตารางที่ 33. ผลลัพธ์ของการใช้ลำดับการจัดเรียงแบบเฉลี่ยสำหรับ ENU language identifier

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	0
30	Merenghi	38	MGR	5	17506.75	0
50	Hanes	15	Mgr	10	20659.80	0
100	Plotz	42	mgr	6	18352.80	0

ลำดับการจัดเรียงและมุมมอง

เมื่อมีการรันคำสั่ง CREATE VIEW มุมมองจะถูกสร้างขึ้นด้วยลำดับการจัดเรียงที่ระบุไว้.

เมื่อมีการอ้างถึงมุมมองใน FROM clause, ลำดับการจัดเรียงนั้นจะถูกใช้สำหรับการเปรียบเทียบอักขระใดๆ ในการเลือกย่อยของ CREATE VIEW. ในขณะนั้น, ตารางผลลัพธ์ระดับกลางจะถูกสร้างจาก การเลือกย่อยของมุมมอง. จากนั้นลำดับการจัดเรียงซึ่งมีผลเมื่อรันเคียวรี จะถูกใช้กับอักขระและการเปรียบเทียบกราฟิก UCS-2 ทั้งหมด (รวมทั้งการเปรียบเทียบที่เกี่ยวข้องกับการแปลงแบบ implicit เป็นอักขระ, หรือกราฟิก UCS-2 หรือกราฟิก UTF-16) ที่ระบุในเคียวรี.

คำสั่ง SQL และตารางต่อไปนี้จะแสดงวิธีการทำงานของมุมมองและลำดับการจัดเรียง. มุมมอง V1, ซึ่งใช้ในตัวอย่างต่อไปนี้, ถูกสร้างด้วยลำดับการจัดเรียงแบบเฉลี่ยน้ำหนักของ SRTSEQ(*LANGIDSHR) และ LANGID(ENU). คำสั่ง CREATE VIEW จะเป็นดังนี้:

```
CREATE VIEW V1 AS SELECT *
FROM STAFF
WHERE JOB = 'MGR' AND ID < 100
```

ตารางที่ 34. "SELECT * FROM V1"

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	0
30	Merenghi	38	MGR	5	17506.75	0
50	Hanes	15	Mgr	10	20659.80	0

เคียวรีใดๆ ที่รันกับมุมมอง V1 จะรันกับตารางผลลัพธ์ที่แสดงไว้ด้านบน. เคียวรีที่แสดงด้านล่างรันกับลำดับการจัดเรียงของ SRTSEQ(*LANGIDUNQ) และ LANGID(ENU).

ตารางที่ 35. "SELECT * FROM V1 WHERE JOB = 'MGR'" โดยใช้ลำดับการจัดเรียงแบบนำหน้าเฉพาะสำหรับ ENU language identifier

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
30	Merenghi	38	MGR	5	17506.75	0

ลำดับการจัดเรียงและคำสั่ง CREATE INDEX

เมื่อมีการรันคำสั่ง CREATE INDEX ดรรชนีจะถูกสร้างขึ้นโดยใช้ลำดับการจัดเรียงที่ระบุไว้.

รายการจะเพิ่มเข้าในดรรชนีทุกครั้งที่มีการแทรก เข้าในตารางที่มีการกำหนดดรรชนีไว้. รายการของดรรชนีมีค่าถ่วงน้ำหนักสำหรับคอลัมน์คีย์อักขระ, และคอลัมน์คีย์กราฟิกแบบ UCS-2 และ UTF-16. ระบบได้รับค่าถ่วงน้ำหนักโดยการแปลงค่าคีย์ซึ่งยึดตาม ลำดับการจัดเรียงของดรรชนี.

เมื่อมีการเลือกโดยใช้ลำดับการเลือกนั้นและดรรชนีนั้น, ไม่จำเป็นต้องแปลงคีย์อักขระ, หรือคีย์กราฟิกแบบ UCS-2 หรือ UTF-16 ก่อนเปรียบเทียบ. ซึ่งจะทำให้ประสิทธิภาพการสืบค้นให้ดีขึ้น.

ข้อมูลที่เกี่ยวข้อง

การใช้ดรรชนีที่มีลำดับการจัดเรียง

ลำดับการจัดเรียงและข้อจำกัด

ข้อจำกัดเฉพาะจะดำเนินการพร้อมด้วยดรรชนี. หากตารางที่มีการเพิ่มข้อจำกัดเฉพาะถูกกำหนดโดยใช้ลำดับการจัดเรียง, ดรรชนีจะถูกสร้างขึ้นด้วยลำดับการจัดเรียงเดียวกัน.

หากระบุข้อจำกัดอ้างอิง, ลำดับการจัดเรียงระหว่างตาราง parent และ dependent ต้องตรงกัน.

ลำดับการจัดเรียงที่ใช้ขณะกำหนดข้อจำกัดการตรวจสอบ จะเป็น ลำดับการจัดเรียงเดียวกันกับที่ระบบใช้เพื่อตรวจสอบความถูกต้องของการปฏิบัติตาม ข้อจำกัดในขณะ INSERT หรือ UPDATE.

ลำดับการจัดเรียง ICU

เมื่อตารางลำดับการจัดเรียง ICU (International Components for Unicode) ถูกใช้, ฐานข้อมูลจะให้การสนับสนุน ICU ของระบบ (อ็อปชัน 39) เพื่อกำหนดการถ่วงน้ำหนักของข้อมูลตามกฎทางภาษาที่เฉพาะเจาะจงตามตารางของโลแคล.

ตารางลำดับการจัดเรียง ICU มีชื่อว่า en_us (United States locale) สามารถเรียงลำดับข้อมูลต่างไปจากตาราง ICU อื่นที่ชื่อว่า fr_FR (French locale) ตัวอย่างเช่น.

การสนับสนุน ICU ของระบบสามารถจัดการข้อมูลที่ไม่ถูกทำให้เป็นมาตรฐานได้อย่างเหมาะสม, ทำให้เกิดผลลัพธ์ที่เหมือนกันกับข้อมูลที่ได้รับการทำให้เป็นมาตรฐานแล้ว. ตารางลำดับการจัดเรียง ICU ของระบบสามารถเรียงลำดับตัวอักษรทั้งหมด, กราฟิก, และข้อมูลยูนิโคด (UTF-8, UTF-16 และ UCS-2).

ตัวอย่างเช่น, คอลัมน์ของอักขระ UTF-8 ที่ชื่อว่า NAME จะมีชื่อสามชื่อต่อไปนี้ (พร้อมค่าตัวเลขฐานสิบหกของคอลัมน์):

NAME	HEX (NAME)
Gómez	47C3B36D657A
Gomer	476F6D6572
Gumby	47756D6279

ลำดับการจัดเรียง *HEX จะจัดเรียงค่าของ NAME ดังนี้:

NAME
Gomer
Gumby
Gómez

ตารางลำดับการจัดเรียง ICU ที่ชื่อ en_us จะทำให้การจัดลำดับค่า NAME เป็นไปอย่างถูกต้อง.

NAME
Gomer
Gómez
Gumby

เมื่อตารางลำดับการจัดเรียง ICU ถูกระบุ, ประสิทธิภาพการทำงานของคำสั่ง SQL ที่ใช้ตารางนั้นอาจช้าลงมากกว่าการใช้ตารางลำดับการจัดเรียงที่ไม่ใช่ ICU หรือ ลำดับการจัดเรียงแบบ *HEX. ประสิทธิภาพการทำงานที่ช้าลงเป็นผลมาจากการเรียกใช้การสนับสนุน ICU ของระบบเพื่อที่จะได้ค่าการถ่วงน้ำหนักของข้อมูลแต่ละชิ้นที่จะถูกจัดเรียง. ตารางลำดับการจัดเรียง ICU ช่วยให้ฟังก์ชันในการเรียงลำดับเพิ่มมากขึ้นแต่จะทำให้การรันคำสั่ง SQL ช้าลง. อย่างไรก็ตาม, ดรรชนีที่ถูกสร้างด้วยตารางลำดับการจัดเรียง ICU สามารถถูกสร้างบนคอลัมน์เพื่อช่วยลดความต้องการในการเรียกไปยังการสนับสนุน ICU ของระบบ. ในกรณีนี้คีย์ดรรชนีควรจะมีค่าการถ่วงน้ำหนัก ICU อยู่แล้วซึ่งจะทำให้ไม่ต้องมีการเรียกใช้การสนับสนุน ICU ของระบบ.

ข้อมูลที่เกี่ยวข้อง

International Components for Unicode

Normalization

Normalization อนุญาตให้คุณเปรียบเทียบสตริงที่มีอักขระแบบผสม.

ข้อมูลที่มีป้าย UTF-8 หรือ UTF-16 CCSID สามารถเก็บอักขระแบบผสมได้. อักขระแบบผสมอนุญาตให้อักขระที่เป็นผลลัพธ์สามารถประกอบขึ้นจากอักขระมากกว่าหนึ่งอักขระ. ในข้อมูลสตริง หลังจากอักขระตัวแรกของอักขระแบบเชิงซ้อน, สามารถตามได้ด้วยอักขระชนิดที่ไม่ใช่เว้นวรรค ตัวอย่างเช่น เครื่องหมายแสดงสำเนียง. ถ้าอักขระผลลัพธ์เป็นหนึ่งในกลุ่มอักขระที่ถูกกำหนดไว้แล้ว, การ normalization ของสตริงจะเป็นผลทำให้ตัวอักขระแบบผสมหลายตัวถูกแทนที่ด้วยค่าของอักขระที่ถูกกำหนดไว้แล้ว. ตัวอย่างเช่น, ถ้าสตริงของคุณประกอบด้วยตัวอักษร 'a' ตามด้วย '..', สตริงจะถูกทำให้เป็นมาตรฐาน ซึ่งประกอบด้วยอักขระ 'ä' ตัวเดียว.

การ Normalization ทำให้เกิดความเป็นไปได้ในการเปรียบเทียบสตริงอย่างแม่นยำ. ถ้าข้อมูลไม่ถูกทำให้เป็นมาตรฐาน, สตริงสองชุดที่ดูเหมือนกันบนหน้าจออาจจะเปรียบเทียบได้ไม่เท่ากันเพราะการแทนค่าที่ถูกเก็บไว้อาจจะแตกต่างกัน. เมื่อข้อมูลสตริง UTF-8 และ UTF-16 ไม่ถูกทำให้เป็นมาตรฐาน, เป็นไปได้ว่า คอลัมน์ในตารางสามารถมีแถวหนึ่งแถวที่มีตัวอักษร 'a' ตามด้วยอักขระเน้นเสียง และอีกแถวที่เป็นอักขระผสม 'ä'. สองค่านี้มีค่าไม่เท่ากันเมื่อเปรียบเทียบเพรดิเคต: WHERE C1 = 'ä'. ด้วยเหตุผลนี้, จึงเป็นการสนับสนุนว่าคอลัมน์สตริงทั้งหมดในตารางควรจะถูกเก็บในรูปแบบที่ทำให้เป็นมาตรฐานแล้ว.

คุณสามารถทำข้อมูลของคุณให้เป็นมาตรฐานก่อนที่จะทำการแทรกหรือทำการอัปเดต, หรือคุณสามารถกำหนดคอลัมน์ในตารางของฐานข้อมูลให้เป็นมาตรฐานแบบอัตโนมัติ. เพื่อให้ฐานข้อมูลทำการ normalization, ระบุ NORMALIZED เป็นส่วนของการนิยามคอลัมน์. อ็อปชันนี้อนุญาตให้เฉพาะคอลัมน์ที่มีป้าย CCSID เป็น 1208 (UTF-8) หรือ 1200 (UTF-16). ฐานข้อมูลทำเสมือนว่าทุกคอลัมน์ในตารางถูกทำให้เป็นมาตรฐานแล้ว.

NORMALIZED clause สามารถระบุไว้สำหรับพารามิเตอร์ของฟังก์ชันและโพรซีเจอร์. ถ้าระบุสำหรับอินพุตพารามิเตอร์, การ normalization จะถูกกระทำโดยฐานข้อมูลสำหรับค่าของพารามิเตอร์ก่อนที่จะเรียกฟังก์ชันหรือโพรซีเจอร์. ถ้าระบุสำหรับเอาต์พุตพารามิเตอร์, clause จะไม่ถูกบังคับ; จะเป็นเสมือนว่าค่าที่ได้รับคืนจากรูทีนของผู้ใช้นั้นถูกทำให้เป็นมาตรฐานแล้ว.

อ็อปชัน NORMALIZE_DATA ในไฟล์ QAQQINI ถูกใช้เพื่อชี้ว่าระบบควรจะทำการ normalization หรือไม่เมื่อทำงานกับข้อมูลแบบ UTF-8 และ UTF-16. อ็อปชันนี้ควบคุมว่าเมื่อไรควรจะทำให้เป็นมาตรฐานก่อนการใช้ใน SQL สำหรับตัวอักษร, ตัวแปรโฮสต์, มาร์คเกอร์พารามิเตอร์, และนิพจน์ที่ประกอบด้วยสตริง. อ็อปชันถูกกำหนดในตอนเริ่มแรกว่าไม่ต้องทำการ normalization. ซึ่งเป็นค่าที่ถูกต้องสำหรับคุณถ้าข้อมูลในตารางของคุณ และค่าตัวอักษรใดๆในแอ็พพลิเคชันของคุณถูกทำให้เป็นมาตรฐานอยู่แล้วโดยกลไกอื่น หรือไม่มีอักขระที่จะต้องถูกทำให้เป็นมาตรฐาน. ถ้าเป็นกรณีนี้คุณจะต้องการหลีกเลี่ยงภาระของการทำ normalization ของระบบในการเคียวรี่ของคุณ. ถ้าข้อมูลของคุณไม่ถูกทำให้เป็นมาตรฐาน, คุณจะต้องการเปลี่ยนค่าของอ็อปชันนี้เพื่อให้ระบบทำการ normalization ให้กับคุณ.

ข้อมูลที่เกี่ยวข้อง

การเปลี่ยนแอ็ตทริบิวต์ของเคียวรี่ด้วยคำสั่ง Change Query Attributes (CHGQRYA)

การปกป้องข้อมูล

หัวข้อนี้จะกล่าวถึงแผนรักษาความปลอดภัยสำหรับการปกป้องข้อมูล SQL จากผู้ใช้ที่ไม่มีสิทธิในการทำงาน และวิธีการในการตรวจสอบ data integrity.

การรักษาความปลอดภัยสำหรับ SQL อ็อบเจกต์

อ็อบเจกต์ทุกตัวในเซิร์ฟเวอร์, รวมถึง SQL อ็อบเจกต์, จะถูกจัดการโดยฟังก์ชันความปลอดภัยของระบบ.

ผู้ใช้สามารถให้สิทธิในการใช้งาน SQL อ็อบเจกต์ได้โดยผ่านทางคำสั่ง SQL GRANT และ REVOKE หรือผ่านทางคำสั่ง CL อันได้แก่ Edit Object Authority (EDTOBJAUT), Grant Object Authority (GRTOBJAUT), และ Revoke Object Authority (RVKOBJAUT).

คำสั่ง SQL GRANT และ REVOKE จะปฏิบัติการบนฟังก์ชัน SQL, SQL แพ็กเกจ, SQL โพรซีเจอร์, ชนิดของการจำแนก, ลำดับ, ตาราง, มุมมอง, และในแต่ละคอลัมน์ของตารางและมุมมอง. นอกเหนือไปจากนั้น, คำสั่ง SQL GRANT และ REVOKE จะยอมรับแต่เพียงสิทธิในการใช้งานแบบ private และ public เท่านั้น. ในบางกรณี, มีความจำเป็นที่จะต้องใช้ EDTOBJAUT, GRTOBJAUT, และ RVKOBJAUT ในการมอบสิทธิในการใช้งาน อ็อบเจกต์อื่นๆให้กับผู้ใช้, ยกตัวอย่างเช่น คำสั่ง และ โปรแกรมต่างๆ.

สิทธิในการใช้งานคำสั่ง SQL จะได้รับการตรวจสอบหรือไม่ ขึ้นอยู่กับว่าในขณะนั้นคำสั่งอยู่ในสถานะ static, dynamic, หรือ กำลังรันแบบโต้ตอบอยู่.

สำหรับคำสั่ง SQL แบบ static:

- ถ้าค่าของ USRPRF เป็น *USER, สิทธิในการใช้งานเพื่อที่จะรันคำสั่ง SQL ในระบบจะได้รับการตรวจสอบโดยใช้โปรไฟล์ผู้ใช้ของผู้ที่กำลังรันโปรแกรมอยู่ในขณะนั้น. สิทธิในการใช้งานเพื่อที่จะรันคำสั่ง SQL แบบรีโมตนั้นจะได้รับการตรวจสอบโดยใช้โปรไฟล์ผู้ใช้ที่อยู่ในแอ็พพลิเคชันเซิร์ฟเวอร์. *USER เป็นค่าโดยปกติในการตั้งชื่อให้ระบบ (*SYS).
- ถ้าค่าของ USRPRF เป็น *OWNER, สิทธิในการใช้งานเพื่อที่จะรันคำสั่ง SQL ในระบบจะได้รับการตรวจสอบโดยใช้โปรไฟล์ผู้ใช้ของผู้ที่กำลังรันโปรแกรมอยู่ในขณะนั้น และโปรไฟล์ผู้ใช้ของเจ้าของโปรแกรม. สิทธิในการใช้งานเพื่อที่จะรันคำสั่ง SQL แบบรีโมตจะได้รับการตรวจสอบโดยใช้โปรไฟล์ผู้ใช้ของงานในแอ็พพลิเคชันเซิร์ฟเวอร์และโปรไฟล์ผู้ใช้ของเจ้าของแพ็คเกจ SQL นั้น. สิทธิที่มีขั้นสูงกว่าคือสิทธิในการใช้งานที่ถูกนำมาใช้. *OWNER เป็นค่าปกติสำหรับการตั้งชื่อให้ SQL (*SQL).

สำหรับคำสั่ง SQL แบบ dynamic:

- ถ้าค่าของ USRPRF เป็น *USER, สิทธิในการใช้งานเพื่อที่จะรันคำสั่ง SQL ในระบบ จะได้รับการตรวจสอบโดยใช้โปรไฟล์ผู้ใช้ของผู้ที่กำลังรันโปรแกรมอยู่ในขณะนั้น. สิทธิในการใช้งานเพื่อรันคำสั่ง SQL แบบรีโมตจะได้รับการตรวจสอบโดยใช้โปรไฟล์ผู้ใช้ของงานในแอ็พพลิเคชันเซิร์ฟเวอร์.
- ถ้าค่าของ USRPRF เป็น *OWNER และ DYNUSRPRF เป็น *USER, สิทธิในการใช้งานเพื่อที่จะรันคำสั่ง SQL ในระบบ จะได้รับการตรวจสอบโดยใช้โปรไฟล์ผู้ใช้ของผู้ที่กำลังรันโปรแกรมอยู่ในขณะนั้น. สิทธิในการใช้งานเพื่อรันคำสั่ง SQL แบบรีโมตจะได้รับการตรวจสอบโดยใช้โปรไฟล์ผู้ใช้ของงานในแอ็พพลิเคชันเซิร์ฟเวอร์.
- ถ้าค่าของ USRPRF เป็น *OWNER และ DYNUSRPRF เป็น *OWNER, สิทธิในการใช้งานเพื่อที่จะรันคำสั่ง SQL ในระบบ จะได้รับการตรวจสอบโดยใช้โปรไฟล์ผู้ใช้ของผู้ที่กำลังรันโปรแกรมอยู่ในขณะนั้น และโปรไฟล์ผู้ใช้ของเจ้าของโปรแกรม. สิทธิในการใช้งานเพื่อที่จะรันคำสั่ง SQL แบบรีโมตจะได้รับการตรวจสอบโดยใช้โปรไฟล์ผู้ใช้ของงานในแอ็พพลิเคชันเซิร์ฟเวอร์และโปรไฟล์ผู้ใช้ของเจ้าของแพ็คเกจ SQL นั้น. สิทธิที่มีขั้นสูงสุดคือสิทธิในการใช้งานที่ถูกนำมาใช้. เพื่อคำนึงถึงความปลอดภัย, จึงควรที่จะใช้ค่าของพารามิเตอร์ *OWNER สำหรับ DYNUSRPRF อย่างระมัดระวัง. ตัวเลือกนี้จะให้สิทธิในการใช้งานของเจ้าของโปรแกรมหรือ package ให้กับผู้ที่รันโปรแกรม.

สำหรับคำสั่ง SQL แบบโต้ตอบ, สิทธิในการใช้งานจะได้รับการตรวจสอบในลักษณะตรงกันข้ามกันกับ สิทธิของผู้ที่กำลังทำการประมวลผลคำสั่งในขณะนั้น. สิทธิที่รับมาจะไม่สามารถใช้ได้กับคำสั่ง SQL แบบโต้ตอบ.

ข้อมูลที่เกี่ยวข้อง

iSeries Security Reference PDF

GRANT (Table หรือ View Privileges)

REVOKE (Table หรือ View Privileges)

รหัสแสดงสิทธิการใช้งาน

รหัสแสดงสิทธิการใช้งานระบุผู้ใช้เพียงหนึ่งเดียวและเป็นอ็อบเจกต์โปรไฟล์ผู้ใช้บนเซิร์ฟเวอร์. Authorization ID สามารถสร้างได้โดยใช้คำสั่ง Create User Profile (CRTUSRPRF).

มุมมอง

มุมมองสามารถป้องกันผู้ใช้ที่ไม่มีสิทธิจากการเข้าถึงข้อมูลที่สำคัญ.

แอ็พพลิเคชันโปรแกรมสามารถเข้าถึงข้อมูลที่ต้องการได้จากตาราง, แต่จะไม่สามารถเข้าถึงข้อมูลที่สำคัญ หรือข้อมูลที่ถูกจำกัดเอาไว้ในตาราง. มุมมองสามารถจำกัดการเข้าถึงข้อมูลในคอลัมน์จำเพาะได้โดยการไม่ระบุคอลัมน์เหล่านั้นลงในรายการ SELECT (ตัวอย่างเช่น, เงินเดือนพนักงาน). มุมมองยังสามารถจำกัดการเข้าถึงแถวจำเพาะในตารางโดยการระบุ WHERE clause (ตัวอย่างเช่น, การอนุญาตให้เข้าถึงข้อมูลเฉพาะแถวที่เชื่อมโยงกับหมายเลขแผนกจำเพาะเท่านั้น).

การตรวจสอบ

DB2 UDB for iSeries ได้ถูกออกแบบมาให้ใช้ได้กับระดับการรักษาความปลอดภัย C2 ของรัฐบาลสหรัฐอเมริกา. คุณสมบัติหลักของระดับนี้ก็คือความสามารถในการตรวจสอบการทำงานของระบบ.

DB2 UDB for iSeries โดยใช้เครื่องมือช่วยในการตรวจสอบซึ่งควบคุมโดยฟังก์ชันความปลอดภัยของระบบ. การตรวจสอบสามารถกระทำได้ในระดับของอ็อบเจกต์, ผู้ใช้, หรือระดับของระบบ. ค่ากำหนดของระบบที่เป็นค่า QAUDCTL จะเป็นตัวควบคุมให้การตรวจสอบถูกกระทำในระดับของ อ็อบเจกต์หรือผู้ใช้. คำสั่ง Change User Audit (CHGUSRAUD) และ คำสั่ง Change Object Audit (CHGOBJAUD) ระบุว่าผู้ใช้หรืออ็อบเจกต์ใดถูกตรวจสอบ. ค่ากำหนดของระบบ QAUDLVL ควบคุมประเภทของการทำงานที่ถูกตรวจสอบ (ตัวอย่างเช่น, ความล้มเหลวในการให้สิทธิ, การสร้าง, การลบออก, การยอมรับ, การเรียกคืน, และอื่นๆ.)

DB2 UDB for iSeries ยังสามารถตรวจสอบการเปลี่ยนแปลงของแถวได้โดยใช้เจอร์นัลสแน็บซันของ DB2 UDB for iSeries.

ในบางกรณี, รายการในบันทึกการตรวจสอบจะไม่เป็นไปตามลำดับตามที่เกิดขึ้นจริง. ตัวอย่างเช่น, งานที่รันภายใต้ commitment control ที่ทำการลบตาราง, สร้างตารางใหม่โดยใช้ชื่อเดียวกับตารางที่ลบไป, แล้วทำการ commit. สิ่งเหล่านี้จะถูกบันทึกลงในเจอร์นัลตรวจสอบในลักษณะของการสร้างแล้วตามด้วยการลบออก. เนื่องจาก อ็อบเจกต์ที่ถูกสร้างขึ้นจะถูกบันทึกทันทีทันใด. อ็อบเจกต์ที่ถูกลบออกภายใต้ commitment control จะถูกซ่อนเอาไว้และยังไม่ถูกลบจริงจนกระทั่งทำการ commit เรียบร้อยแล้ว. ทันทีที่ทำการ commit เสร็จเรียบร้อย, การทำงานนั้นจะถูกบันทึกเอาไว้.

ข้อมูลที่เกี่ยวข้อง

iSeries Security Reference PDF

Data integrity

Data integrity ป้องกันข้อมูลจากการถูกทำลาย หรือเปลี่ยนแปลงโดยผู้ที่ไม่มีสิทธิในการใช้งาน, การดำเนินการของระบบ หรือความขัดข้องของฮาร์ดแวร์ (เช่น ความเสียหายที่เกิดกับดิสก์ในดาวน์ฟิลล์), ข้อผิดพลาดในการเขียนโปรแกรม, การขัดจังหวะก่อนที่งานจะเสร็จสมบูรณ์ (เช่น ความล้มเหลวในการจ่ายกระแสไฟฟ้า), หรือสอดแทรกจากการรันแอ็พพลิเคชันอื่นในเวลาเดียวกัน (เช่น ปัญหาที่เกิดต่อเนื่องกันไป).

Concurrency

Concurrency เป็นความสามารถในการรองรับผู้ใช้หลายคนในการเข้าถึงและเปลี่ยนแปลงข้อมูลในตารางเดียวกันหรือ มุมมองเดียวกันในเวลาเดียวกันโดยปราศจากความเสี่ยงในการสูญเสีย data integrity.

ความสามารถนี้จะถูกจัดไว้ให้โดยอัตโนมัติโดยตัวจัดการฐานข้อมูล DB2 UDB for iSeries. ล็อกเป็นสิ่งที่จำเป็นสำหรับตารางและแถว ซึ่งมีไว้เพื่อปกป้องไม่ให้ผู้ใช้หลายคนทำการเปลี่ยนแปลงข้อมูลชุดเดียวกันในเวลาตรงกันพอดี.

โดยปกติ, DB2 UDB for iSeries จำเป็นที่จะต้องมียกเว้นแถวเพื่อยืนยัน integrity. อย่างไรก็ตาม, ในบางสถานการณ์ต้องการให้ DB2 UDB for iSeries มีการล็อกในระดับตารางแทนที่จะเป็นการล็อกแถว.

ตัวอย่างเช่น, ล็อกอัปเดต (แบบเฉพาะ) ของแถวซึ่งถูกระงับการทำงานเคอร์เซอร์ตัวหนึ่งสามารถถูกเรียกใช้โดยเคอร์เซอร์อีกตัวหนึ่งในโปรแกรมเดียวกัน (หรือใน DELETE หรือ UPDATE คำสั่งที่ไม่เกี่ยวข้องกับเคอร์เซอร์นั้น). การทำแบบนี้จะป้องกันคำสั่ง UPDATE หรือ DELETE ที่ถูกกำหนดไว้ซึ่งเป็นตัวอ้างถึงเคอร์เซอร์ตัวแรก ไปจนกระทั่ง มีการ FETCH ครั้งต่อไปเกิดขึ้น. ล็อกแบบอ่าน (ไม่มีการอัปเดตร่วมกัน) ของแถวซึ่งถูกระงับการทำงาน โดยเคอร์เซอร์ตัวหนึ่ง จะไม่มีการป้องกันเคอร์เซอร์ตัวอื่นจากโปรแกรมเดียวกัน (หรือคำสั่ง DELETE หรือ UPDATE) จากการเรียกใช้ล็อกในแถวเดียวกัน.

ค่าปกติ และ user-specifiable lock-wait time-out จะถูกกำหนดให้. DB2 UDB for iSeries สร้างตาราง, มุมมอง, และดรรชนีด้วยค่าดีฟอลต์ของ record wait time (60 วินาที) และค่าปกติของ file wait time (*IMMED). wait time ของล็อกนี้จะถูกนำมาใช้ใน DML statement. สามารถทำการเปลี่ยนแปลงค่าเหล่านี้โดยใช้คำสั่ง CL นั่นคือ Change Physical File (CHGPF), Change Logical File (CHGLF), และ Override Database File (OVRDBF).

Wait time ของล็อกที่ใช้ในคำสั่ง DDL ทั้งหมดและคำสั่ง LOCK TABLE, จะเป็นค่า wait time ปกติของงาน (DFTWAIT). คุณสามารถเปลี่ยนค่านี้ได้โดยใช้คำสั่ง CL ที่เป็น Change Job (CHGJOB) หรือ Change Class (CHGCLS).

ในกรณีที่มีการระบุค่า record wait time ไว้มากๆ, จะมีการเตรียมการตรวจหา deadlock เอาไว้ด้วย. ตัวอย่างเช่น, สมมติว่างานชิ้นหนึ่งมีล็อกเฉพาะอยู่บนแถวที่ 1 และงานอีกชิ้นหนึ่งมีล็อกเฉพาะอยู่บนแถวที่ 2. ถ้างานชิ้นแรกพยายามที่จะล็อกแถวที่ 2, จะต้องรอเนื่องจากงานชิ้นที่ 2 มีล็อกอยู่บนแถวที่ 2 นั้น. ถ้างานชิ้นที่สองพยายามที่จะล็อกแถวที่ 1, DB2 UDB for iSeries จะตรวจพบว่า มีงานสองงานอยู่ใน deadlock และข้อผิดพลาดจะถูกส่งไปยังงานชิ้นที่ 2.

คุณสามารถป้องกันผู้ใช้คนอื่นๆ จากการใช้ตารางในเวลาเดียวกันโดยใช้คำสั่ง SQL LOCK TABLE. การใช้ COMMIT(*RR) จะช่วยป้องกันผู้ใช้คนอื่นๆ จากการใช้ตารางร่วมกันในระหว่างการทำงานแต่ละครั้ง.

หากต้องการปรับปรุงประสิทธิภาพการทำงาน, DB2 UDB for iSeries จะปล่อยให้ open data path (ODP) เปิดอยู่เสมอ. คุณสมบัตินี้ในการดำเนินการนี้จะปล่อยล็อกไว้บนตารางที่อ้างอิงโดย ODP, แต่จะไม่ทิ้งล็อกใดๆ ไว้บนแถว. ล็อกที่ถูกทิ้งไว้บนตารางจะป้องกันไม่ให้งานอื่นดำเนินการบนตารางนั้น. โดยส่วนใหญ่, อย่างไรก็ตาม, DB2 UDB for iSeries จะตรวจพบว่ามีงานอื่นๆ มีล็อกอยู่และจะมีการส่งสัญญาณเกี่ยวกับเหตุการณ์ต่างๆ ไปยังงานเหล่านั้น. เหตุการณ์นี้ทำให้ DB2 UDB for iSeries ปิด ODP ใดๆ (และทำการปล่อยล็อกในตาราง) ที่เชื่อมโยงกับตารางนั้น และขณะนี้จะเป็นเฉพาะในกรณีที่มีเหตุผลเกี่ยวกับการดำเนินงานเท่านั้น. สังเกตว่า wait time out ของล็อกจะต้องมีค่ามากพอสำหรับการส่งสัญญาณต่อเหตุการณ์และงานอื่นๆ เพื่อที่จะปิด ODP หรือส่งข้อผิดพลาดกลับมา.

นอกจากจะใช้คำสั่ง LOCK TABLE ในการเรียกใช้ล็อกของตาราง, หรือใช้ COMMIT(*ALL) หรือ COMMIT(*RR), ข้อมูลที่ถูกอ่านโดยงานหนึ่งอาจถูกเปลี่ยนแปลงได้โดยงานอีกงานหนึ่ง. โดยปกติ, ข้อมูลที่อ่านในขณะที่ คำสั่ง SQL ทำงานจะเป็นข้อมูลที่ป็นปัจจุบันมาก (ตัวอย่างเช่น, ระหว่างใช้คำสั่ง FETCH). อย่างไรก็ตาม, ในกรณีตัวอย่างนี้, ข้อมูลถูกอ่านขึ้นมาก่อนการทำงานของคำสั่ง SQL ซึ่งเป็นผลให้ได้ข้อมูลที่ไม่น่าเป็นปัจจุบันได้ (ตัวอย่าง, ระหว่างคำสั่ง OPEN).

- ถ้าระบุค่า ALWCPYDTA(*OPTIMIZE) optimizer จะกำหนดว่าการทำสำเนาข้อมูลจะทำงานได้ดีกว่าการไม่ทำสำเนา.
- การสืบค้นบางอย่างต้องอาศัยตัวจัดการฐานข้อมูลในการสร้างตารางผลลัพธ์ชั่วคราว. ข้อมูลที่อยู่ในตารางผลลัพธ์ชั่วคราวนี้จะไม่แสดงการเปลี่ยนแปลงที่เกิดขึ้นหลังจากที่เคอร์เซอร์ถูกเปิด. ตารางผลลัพธ์ชั่วคราวจะจำเป็นก็ต่อเมื่อ:
 - ข้อมูลที่เก็บในคอลัมน์ที่ถูกระบุไว้ใน ORDER BY clause มีค่าเกิน 2000 ไบต์.
 - ORDER BY และ GROUP BY clause ระบุคอลัมน์ที่ต่างกันหรือคอลัมน์ในลำดับที่ต่างกัน.
 - UNION หรือ DISTINCT clause ถูกระบุไว้.
 - ORDER BY หรือ GROUP BY clause ระบุคอลัมน์ที่ไม่ได้มาจากตารางเดียวกันทั้งหมด.
 - การเชื่อมโลจิคัลไฟล์ที่กำหนดโดยคีย์เวิร์ด JOIN DFT data definition specification (DDS) เข้ากับอีกไฟล์หนึ่ง.
 - การเชื่อมหรือการระบุ GROUP BY บนโลจิคัลไฟล์โดยยึดหลักรายการไฟล์ฐานข้อมูลย่อยหลายๆ ไฟล์.
 - การสืบค้นที่มี join in อยู่ซึ่งจะมีอย่างน้อยหนึ่งไฟล์ในจำนวนหลายๆ ไฟล์เป็น มุมมองที่มี GROUP BY clause อยู่.
 - การสืบค้นที่มี GROUP BY clause อยู่ซึ่งจะเป็นการอ้างอิงถึง มุมมองที่มี GROUP BY clause อยู่.
- การสืบค้นย่อยระดับต้นจะถูกประเมินผลเมื่อการสืบค้นนั้นถูกเปิด.

ข้อมูลที่เกี่ยวข้อง

LOCK TABLE statement

การทำเจอร์นัล

เจอร์นัลสนับสนุนของ DB2 UDB for iSeries จะช่วยเป็นหลักฐานการตรวจสอบและการกู้คืนทั้งแบบ forward และ backward.

Forward recovery หรือ การกู้คืนแบบส่งต่อ คือการนำตารางเวอร์ชันที่เก่ากว่ามาทำการเปลี่ยนแปลงข้อมูลตามบันทึกของเจอร์นัล. Backward recovery หรือการกู้คืนแบบเรียกคืน คือการยกเลิกการเปลี่ยนแปลงตามที่ได้บันทึกไว้ในเจอร์นัล.

เมื่อแบบแผน SQL ถูกสร้างขึ้น, เจอร์นัลและ receiver จะถูกสร้างขึ้นในแบบแผน. เมื่อ SQL สร้างเจอร์นัลและ เจอร์นัลreceiver ขึ้น, ทั้งหมดจะถูกสร้างขึ้นในส่วนของหน่วยความจำสำรอง (ASP) ของผู้ใช้ ถ้ามีการระบุ ASP clause ในคำสั่ง CREATE SCHEMA. อย่างไรก็ตาม, เนื่องจากการกำหนด journal receiver ในส่วนของ ASP ทำให้ประสิทธิภาพการทำงานดีขึ้น, ผู้ที่จัดการเกี่ยวกับเจอร์นัลนั้นอาจจะต้องการที่จะสร้าง journal receiver ที่จะมีในอนาคตในส่วนของ ASP ที่แยกออกไปต่างหาก .

เมื่อตารางถูกสร้างขึ้นในแบบแผน, จะมีการบันทึกเกิดขึ้นโดยอัตโนมัติในเจอร์นัล DB2 UDB for iSeries ที่ถูกสร้างขึ้นในแบบแผน (QSQRN). ตารางที่สร้างขึ้นใน non-schema จะมีการบันทึกเริ่มขึ้นถ้ามีเจอร์นัลชื่อ QSQRN ปรากฏในไลบรารีนั้น. หลังจากจุดนี้ไป, จะเป็นความรับผิดชอบของคุณที่จะใช้ฟังก์ชันของบันทึกในการจัดการเจอร์นัล, journal receiver, และการทำเจอร์นัลตารางลงในเจอร์นัล. ตัวอย่างเช่น, ถ้าตารางถูกย้ายไปอยู่ในแบบแผน, ระบบจะไม่ทำการเปลี่ยนแปลงสถานะการทำเจอร์นัลโดยอัตโนมัติ. ถ้าตารางถูกเรียกคืนมา, กฎของเจอร์นัลปกติจะถูกนำมาใช้. นั่นคือ, ถ้าตารางถูกทำเจอร์นัลในเวลาที่เป็นบันทึกข้อมูล, ตารางจะถูกบันทึกลงในเจอร์นัลเดียวกันกับเวลาที่เรียกคืนข้อมูล. ถ้าตารางไม่ถูกทำเจอร์นัลในเวลาที่เป็นบันทึกข้อมูล, เจอร์นัลจะไม่ถูกบันทึกในเวลาที่เราเรียกคืน.

เจอร์นัลที่ถูกสร้างขึ้นในคอลเล็กชัน SQL โดยปกติจะเป็นเจอร์นัลที่ถูกใช้สำหรับบันทึกการเปลี่ยนแปลงทั้งหมดที่เกิดขึ้นกับตาราง SQL เหล่านั้น. อย่างไรก็ตาม, คุณสามารถ, ใช้ฟังก์ชันเจอร์นัลของระบบในการบันทึกตาราง SQL ลงในเจอร์นัลที่ต่างกัน.

ผู้ใช้สามารถหยุดการทำเจอร์นัลตารางใดๆ โดยใช้ฟังก์ชันเจอร์นัล, แต่การทำดังกล่าวจะป้องกันไม่ให้แอ็พพลิเคชันรันภายใต้ commitment control. ถ้าการทำเจอร์นัลถูกหยุดในตารางที่เป็น parent ของข้อจำกัดที่อ้างอิงถึง โดยกฎการลบของ NO

ACTION, CASCADE, SET NULL, หรือ SET DEFAULT, การดำเนินการอัปเดตและการลบทั้งหมดจะไม่เกิดขึ้น. มิฉะนั้น, แอ็พพลิเคชันจะยังสามารถทำงานได้ ถ้าคุณได้ระบุ COMMIT(*NONE); อย่างไรก็ตาม, การทำเช่นนี้จะไม่ทำให้เกิด integrity ระดับเดียวกันกับที่ได้จากการทำเจอร์นัล และ commitment control.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การอัปเดตตารางด้วยข้อจำกัดในการอ้างอิง” ในหน้า 102

หากคุณอัปเดตตาราง *parent*, คุณไม่สามารถแก้ไข primary key ที่มีแถว dependent อยู่ได้.

ข้อมูลที่เกี่ยวข้อง

การทำเจอร์นัล

Commitment control

การสนับสนุน DB2 UDB for iSeries commitment control จะรวบรวมวิธีการประมวลผลกลุ่มของการเปลี่ยนแปลงของฐานข้อมูล เช่น อัปเดต, แทรก, DDL, หรือลบออก ในลักษณะของหน่วยการทำงานเดี่ยว (transaction).

commit operation รับประกันว่ากลุ่มของการปฏิบัติการได้ถูกดำเนินการอย่างสมบูรณ์. การทำ rollback รับประกันว่ากลุ่มของการปฏิบัติการได้ถูกส่งกลับออกมา. Savepoint สามารถนำมาใช้ในการแบ่ง transaction ให้เป็นหน่วยที่เล็กลงทำให้สามารถทำการ roll back ได้. Commit operation สามารถรับคำสั่งผ่านทางอินเตอร์เฟซที่แตกต่างกันได้หลายทาง. ตัวอย่างเช่น,

- SQL COMMIT statement
- คำสั่ง CL COMMIT
- คำสั่ง language commit (เช่น คำสั่ง RPG COMMIT)

การทำ rollback สามารถรับคำสั่งผ่านทางอินเตอร์เฟซที่แตกต่างกันได้หลายทาง. ตัวอย่างเช่น,

- SQL ROLLBACK statement
- คำสั่ง CL ROLLBACK
- คำสั่ง language rollback (เช่น คำสั่ง RPG ROLBK)

มีคำสั่ง SQL เพียงชุดเดียวที่ไม่สามารถทำการ commit หรือ roll back ได้นั้นคือ:

- DROP SCHEMA
- GRANT หรือ REVOKE ถ้ามีผู้ถือสิทธิในการใช้งานปรากฏอยู่ในอ็อบเจกต์ที่ได้รับเอาไว้

ถ้า commitment control ยังไม่ได้เริ่มทำงาน เมื่อมีการเรียกใช้คำสั่ง SQL ด้วยระดับการแยกกันอื่นๆ ที่ไม่ใช่ COMMIT (*NONE) หรือ เมื่อเรียกใช้คำสั่ง RELEASE, DB2 UDB for iSeries จะตั้งค่าสถานะแวดล้อมของ commitment control โดยการเรียกคำสั่ง CL นั้นคือ Start Commitment Control (STRCMTCTL). DB2 UDB for iSeries ระบุพารามิเตอร์ NRYOBJ (*NONE) และ CMTSCOPE(*ACTGRP) พร้อมกับ LCKLVL บนคำสั่ง STRCMTCTL. LCKLVL เป็นระดับล็อกในพารามิเตอร์ COMMIT ในคำสั่ง CRTSQLxxx, STRSQL, หรือ RUNSQLSTM. ใน REXX, LCKLVL ที่ได้รับไว้เป็นระดับล็อกที่อยู่ในคำสั่ง SET OPTION. คุณสามารถใช้คำสั่ง STRCMTCTL ในการระบุ CMTSCOPE, NRYOBJ, หรือ LCKLVL. ถ้าคุณระบุ CMTSCOPE(*JOB) เพื่อที่เริ่มการทำงานของ job level commitment definition, DB2 UDB for iSeries จะใช้ job level commitment definition นั้นสำหรับโปรแกรมใน activation group นั้น.

หมายเหตุ:

1. เมื่อใช้ commitment control, ตารางที่ถูกอ้างอิงในแอ็พพลิเคชันโปรแกรมโดยคำสั่ง Data Manipulation Language คำสั่งจะต้องมีการทำบันทึกลงเจอร์นัลไว้.

- สังเกตว่า LCKLVL ที่ถูกระบุไว้เป็นเพียงแค่ค่าปกติของระดับของล็อก. หลังจาก commitment control เริ่มการทำงาน, คำสั่ง SET TRANSACTION SQL และระดับของล็อกที่ถูกระบุเอาไว้ในพารามิเตอร์ COMMIT ในคำสั่ง CRTSQLxxx, STRSQL, หรือ RUNSQLSTM จะแทนที่ค่าเดิมซึ่งเป็นค่าปกติของระดับล็อก.

สำหรับเคอร์เซอร์ที่ใช้ฟังก์ชันของคอลัมน์, GROUP BY, หรือ HAVING, และกำลังรันภายใต้ commitment control, ROLLBACK HOLD จะไม่มีผลใดๆ ต่อตำแหน่งเคอร์เซอร์. นอกจากนี้, สิ่งต่างๆ ต่อไปนี้จะเกิดขึ้นภายใต้ commitment control:

- ถ้า COMMIT(*CHG) และ (ALWBLK(*NO) หรือ (ALWBLK(*READ))) ถูกระบุสำหรับเคอร์เซอร์ใดต่อไปนี้, ข้อความ (CPI430B) จะถูกส่งออกไปเพื่อแจ้งว่า COMMIT(*CHG) ถูกร้องขอแต่ไม่ได้รับอนุญาต.
- ถ้า COMMIT(*ALL), COMMIT(*RR), หรือ COMMIT(*CS) พร้อมกับ KEEP LOCKS clause ถูกระบุเอาไว้สำหรับเคอร์เซอร์ใด อันใดอันหนึ่ง, DB2 UDB for iSeries จะล็อกตารางที่ถูกอ้างถึงทั้งหมดให้อยู่ในโหมดใช้ร่วมกัน (*SHRNUP). การล็อกจะป้องกันการประมวลผลพร้อมกันเนื่องจากการเรียกใช้ปฏิบัติการใดๆ ยกเว้นปฏิบัติการแบบ read-only ในตารางที่ระบุชื่อไว้. ข้อความ (SQL7902 หรือ CPI430A อย่างใดอย่างหนึ่ง) จะถูกส่งออกไปว่า COMMIT(*ALL), COMMIT(*RR), หรือ COMMIT(*CS) พร้อมกับ KEEP LOCKS clause ได้ถูกระบุไว้สำหรับเคอร์เซอร์ใดถูกร้องขอแต่ไม่ได้รับอนุญาต. ข้อความ SQL0595 อาจถูกส่งไปด้วยเช่นกัน.

สำหรับเคอร์เซอร์ที่มี COMMIT(*ALL), COMMIT(*RR), หรือ COMMIT(*CS) อย่างใดอย่างหนึ่งพร้อมกับ KEEP LOCKS clause ที่ถูกระบุเอาไว้และไฟล์แค็ตตาล็อกไฟล์ใดไฟล์หนึ่งถูกใช้ หรือจำเป็นต้องใช้ตารางผลลัพธ์ชั่วคราว, DB2 UDB for iSeries จะล็อกตารางที่ถูกอ้างถึงทั้งหมดให้อยู่ในโหมดใช้งานร่วมกัน (*SHRNUP). การทำเช่นนี้จะป้องกันการประมวลผลพร้อมกันเนื่องจากการเรียกใช้ปฏิบัติการใดๆ ยกเว้นปฏิบัติการแบบ read-only ในตาราง. ข้อความ (SQL7902 หรือ CPI430A อย่างใดอย่างหนึ่ง) จะถูกส่งออกมาว่า COMMIT(*ALL) ถูกร้องขอแต่ไม่ได้รับอนุญาต. ข้อความ SQL0595 อาจถูกส่งไปด้วยเช่นกัน.

ถ้ามีการระบุ ALWBLK(*ALLREAD) และ COMMIT(*CHG), ตอนที่โปรแกรมถูกฟรีคอมไพล์, เคอร์เซอร์ที่เป็นแบบ read-only ทั้งหมดจะอนุญาตให้มีการจัดเป็นกลุ่มบล็อกของแถว โดยที่ ROLLBACK HOLD จะไม่ย้อนตำแหน่งของเคอร์เซอร์กลับมา.

ถ้า COMMIT(*RR) ถูกร้องขอ, ตารางจะถูกล็อกจนกระทั่งปิดการสืบค้น. ถ้าเคอร์เซอร์เป็นแบบ read-only, ตารางจะถูกล็อก (*SHRNUP). ถ้าเคอร์เซอร์อยู่ในโหมดอัปเดต, ตารางจะถูกล็อก (*EXCLRD). เนื่องจากผู้ใช้คนอื่นๆ จะถูกล็อกอยู่นอกตาราง, การรันด้วยการอ่านแบบอ่านซ้ำได้จะช่วยป้องกันการเข้าถึงตารางพร้อมกันได้.

หากมีการระบุระดับ isolation อื่นๆ ที่ไม่ใช่ COMMIT(*NONE) และแอ็พพลิเคชันได้ทำการออกคำสั่ง ROLLBACK หรือ activation group สิ้นสุดการทำงานแบบไม่ปกติ (และ commitment definition ไม่ใช่ *JOB), ปฏิบัติการอัปเดต, การแทรก, การลบออก, และ DDL ทั้งหมดที่ถูกกระทำในช่วงหน่วยการทำงานจะถูกยับยั้งเอาไว้. ถ้าแอ็พพลิเคชันออกคำสั่ง COMMIT หรือ activation สิ้นสุดการทำงานแบบไม่ปกติ, ปฏิบัติการอัปเดต, การแทรก, การลบออก, และ DDL ทั้งหมดที่ถูกกระทำอยู่ในช่วงหน่วยการทำงานจะถูก commit เอาไว้.

DB2 UDB for iSeries ใช้ล็อกกับแถวเพื่อที่จะป้องกันงานอื่นๆ ไม่ให้เข้าถึงข้อมูลที่ถูกเปลี่ยนแปลงก่อนที่หน่วยการทำงานนั้นจะเสร็จสิ้น. ถ้า COMMIT(*ALL) ถูกระบุเอาไว้, ล็อกการอ่านในแถวที่ถูกเรียกข้อมูลจะถูกใช้ป้องกันงานอื่นๆ จากการเปลี่ยนแปลงข้อมูลที่ถูกอ่านก่อนที่หน่วยการทำงานจะเสร็จสิ้น. การทำเช่นนี้เป็นการป้องกันงานอื่นๆ จากการอ่านแถวที่ไม่มีการเปลี่ยนแปลง. ซึ่งทำให้มั่นใจว่า, ถ้าหน่วยการทำงานเดียวกันทำการอ่านแถวนั้นซ้ำ, จะทำให้ได้ค่าผลลัพธ์เช่นเดิม. ล็อกป้องกันการอ่านจะไม่สามารถป้องกันงานอื่นจากการดึงข้อมูลในแถวเดียวกันออกมาใช้.

Commitment control สามารถจัดการกับการเปลี่ยนแปลงข้อมูลในแถวได้ถึง 500 ล้านแถวในหนึ่งหน่วยการทำงาน. ถ้า COMMIT(*ALL) หรือ COMMIT(*RR) ถูกระบุไว้, แถวที่ถูกอ่านทั้งหมดจะถูกรวมอยู่ในชุดจำกัดด้วย. (ถ้าแถวข้อมูลแถวหนึ่งถูกเปลี่ยนแปลงหรือถูกอ่านมากกว่าหนึ่งครั้งในหนึ่งหน่วยการทำงาน, จะถูกนับเป็นครั้งเดียวจากที่จำกัดไว้.) การมีล็อกเป็นจำนวนมากจะมีผลต่อประสิทธิภาพการทำงานของระบบและจะทำให้ผู้ใช้หลายคนเข้ามาใช้งานแถวที่ถูกล็อกไว้ในหน่วยการทำงานเดียวกันไม่ได้จนกระทั่งหน่วยนั้นทำงานเสร็จ. เรื่องที่ต้องคำนึงถึงที่สุดก็คือ ทำอย่างไรให้มีจำนวนแถวที่ถูกประมวลผลในหนึ่งหน่วยการทำงานมีค่าน้อยๆ.

Commitment control จะอนุญาตให้ไฟล์จำนวนมากถึง 512 ไฟล์ในแต่ละเจอร์นัลถูกเปิดภายใต้ commitment control หรือปิดด้วยการเปลี่ยนแปลงที่รออยู่ในหนึ่งหน่วยการทำงาน.

COMMIT HOLD และ ROLLBACK HOLD อนุญาตให้เปิดเคอร์เซอร์ทิ้งไว้ได้และเริ่มการทำงานของหน่วยการทำงานอื่น โดยไม่จำเป็นต้องออกคำสั่ง OPEN อีกครั้งหนึ่ง. ค่าของ HOLD จะไม่ปรากฏในกรณีที่คุณเชื่อมต่อกับฐานข้อมูลแบบรีโมทที่ไม่ได้อยู่ในระบบ iSeries. อย่างไรก็ตาม, คุณสามารถใช้ตัวเลือก WITH HOLD ใน DECLARE CURSOR ในการทำให้เคอร์เซอร์เปิดหลังจากการ COMMIT. เคอร์เซอร์ชนิดนี้จะสนับสนุนการทำงานเมื่อเชื่อมต่อกับฐานข้อมูลแบบรีโมทที่ไม่ได้อยู่ในระบบ iSeries. เคอร์เซอร์นี้จะปิดในช่วงการ rollback.

ตารางที่ 36. ช่วงระยะเวลาในการล็อกของแถว

คำสั่ง SQL	พารามิเตอร์ COMMIT (ดูในหมายเหตุ 5)	ช่วงระยะเวลาของการล็อกของแถว	ชนิดของการล็อก
SELECT INTO SET variable VALUES INTO	*NONE *CHG *CS (ดูในหมายเหตุ 6) *ALL (ดูในหมายเหตุ 2 และ	ไม่มีล็อก ไม่มีล็อก แถวจะถูกล็อกเมื่อถูกอ่านแล้วจะปล่อยล็อก ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT	READ READ
FETCH (cursor แบบ read-only)	*NONE *CHG *CS (ดูในหมายเหตุ 6) *ALL (ดูในหมายเหตุ 2 และ	ไม่มีล็อก ไม่มีล็อก ตั้งแต่การอ่านจนถึงการ FETCH ครั้งต่อไป ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT	READ READ
FETCH (เคอร์เซอร์ที่สามารถอัปเดตหรือลบออกได้) (ดูในหมายเหตุ 1)	*NONE *CHG *CS *ALL	เมื่อแถวข้อมูลไม่ได้ถูกอัปเดตหรือลบออก ตั้งแต่การอ่านจนถึงการ FETCH ครั้งต่อไป เมื่อแถวข้อมูลถูกอัปเดตหรือลบออก ตั้งแต่การอ่านจนถึง UPDATE หรือ DELETE เมื่อแถวข้อมูลไม่ได้ถูกอัปเดตหรือลบออก ตั้งแต่การอ่านจนถึงการ FETCH ครั้งต่อไป เมื่อแถวข้อมูลถูกอัปเดตหรือลบออก ตั้งแต่การอ่านจนถึง COMMIT หรือ ROLLBACK เมื่อแถวข้อมูลไม่ได้ถูกอัปเดตหรือลบออก ตั้งแต่การอ่านจนถึงการ FETCH ครั้งต่อไป เมื่อแถวข้อมูลถูกอัปเดตหรือลบออก ตั้งแต่การอ่านจนถึง COMMIT หรือ ROLLBACK ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT	UPDATE UPDATE UPDATE UPDATE

ตารางที่ 36. ช่วงระยะเวลาในการล็อกของแถว (ต่อ)

คำสั่ง SQL	พารามิเตอร์ COMMIT (ดูในหมายเหตุ 5)	ช่วงระยะเวลาของการล็อกของแถว	ชนิดของการล็อก
INSERT (ตารางเป้าหมาย)	*NONE *CHG *CS *ALL	ไม่มีล็อก ตั้งแต่การแทรกจนถึง ROLLBACK หรือ COMMIT ตั้งแต่การแทรกจนถึง ROLLBACK หรือ COMMIT ตั้งแต่การแทรกจนถึง ROLLBACK หรือ COMMIT	UPDATE UPDATE UPDATE ³
INSERT (ตารางในการเลือกย่อย)	*NONE *CHG *CS *ALL	ไม่มีล็อก ไม่มีล็อก แต่ละแถวจะถูกล็อกในขณะที่ถูกอ่าน ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT	READ READ
UPDATE (non-cursor)	*NONE *CHG *CS *ALL	แต่ละแถวจะถูกล็อกในขณะที่ถูกอัปเดต ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT	UPDATE UPDATE UPDATE UPDATE
DELETE (ไม่ใช่เคอร์เซอร์)	*NONE *CHG *CS *ALL	แต่ละแถวจะถูกล็อกในขณะที่ถูกลบออก ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT	UPDATE UPDATE UPDATE UPDATE
UPDATE (ใช้เคอร์เซอร์)	*NONE *CHG *CS *ALL	ล็อกจะถูกปล่อยเมื่อแถวได้อัปเดตแล้ว ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT	UPDATE UPDATE UPDATE UPDATE
DELETE (ใช้เคอร์เซอร์)	*NONE *CHG *CS *ALL	ล็อกจะถูกปล่อยเมื่อแถวถูกลบออกไป ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT	UPDATE UPDATE UPDATE UPDATE
การสืบค้นย่อย (เคอร์เซอร์ที่สามารถอัปเดตหรือลบออกได้ หรือ UPDATE หรือ DELETE โดยไม่ใช่เคอร์เซอร์)	*NONE *CHG *CS *ALL (ดูในหมายเหตุ 2)	ตั้งแต่การอ่านจนถึงการ FETCH ครั้งต่อไป ตั้งแต่การอ่านจนถึงการ FETCH ครั้งต่อไป ตั้งแต่การอ่านจนถึงการ FETCH ครั้งต่อไป ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT	READ READ READ READ
เคียวรีย่อย (เคอร์เซอร์แบบ read-only หรือ SELECT INTO)	*NONE *CHG *CS *ALL	ไม่มีล็อก ไม่มีล็อก แต่ละแถวจะถูกล็อกในขณะที่ถูกอ่าน ตั้งแต่การอ่านจนกระทั่ง ROLLBACK หรือ COMMIT	READ READ

ตารางที่ 36. ช่วงระยะเวลาในการล็อกของแถว (ต่อ)

คำสั่ง SQL	พารามิเตอร์ COMMIT (ดูในหมายเหตุ 5)	ช่วงระยะเวลาของการล็อกของแถว	ชนิดของการล็อก
<p>หมายเหตุ:</p> <ol style="list-style-type: none"> เคอร์เซอร์จะเปิดไว้และอนุญาตให้ UPDATE หรือ DELETE ถ้าตารางผลลัพธ์ไม่ได้เป็นแบบ read-only และถ้าหนึ่งในสิ่งต่อไปนี้เป็นจริง: <ul style="list-style-type: none"> เคอร์เซอร์ถูกระบุด้วย FOR UPDATE clause. เคอร์เซอร์ไม่ถูกระบุด้วย FOR UPDATE, FOR READ ONLY, หรือ ORDER BY clause และโปรแกรมมีสิ่งต่างๆ ต่อไปอย่างน้อยหนึ่งอย่าง: <ul style="list-style-type: none"> Cursor UPDATE ที่อ้างอิงชื่อเดียวกัน Cursor DELETE ที่อ้างอิงชื่อเดียวกัน EXECUTE หรือ EXECUTE IMMEDIATE คำสั่งและ ALWBK(*READ) หรือ ALWBK(*NONE) ที่ถูกระบุในคำสั่ง CRTSQLxxx. ตารางหรือ มุมมองสามารถถูกล็อกได้ต่างหากเพื่อที่จะให้รองรับคำสั่ง COMMIT(*ALL). ถ้าการเลือกย่อยถูกประมวลผลและรวม UNION, หรือถ้าการประมวลผลของการสืบค้นนั้นต้องการใช้ตารางผลลัพธ์ชั่วคราว, จำเป็นที่จะต้องใช้ล็อกต่างหากเพื่อไม่ให้คุณเห็นการเปลี่ยนแปลงที่ไม่มีการ commit. ล็อกแบบ UPDATE ในแถวของตารางเป้าหมายและล็อกแบบ READ ในแถวของตารางการเลือกย่อย. ตารางหรือ มุมมองสามารถถูกล็อกไว้ต่างหากได้เพื่อที่จะรองรับการอ่านแบบซ้ำๆ. การล็อกแถวจะยังคงดำเนินต่อไปในขณะที่มีการอ่านแบบซ้ำๆ. ล็อกเป็นสิ่งจำเป็นและมีช่วงระยะเวลาการทำงานเท่ากับ *ALL. การล็อกแถวที่มีการอ่านแบบซ้ำๆ (*RR) จะเป็นเช่นเดียวกับล็อกที่ระบุไว้สำหรับ *ALL. ถ้า KEEP LOCKS clause ถูกระบุไว้ด้วย *CS, ล็อกในการอ่านใดๆ จะถูกพักไว้จนกระทั่งปิดเคอร์เซอร์หรือการทำ COMMIT หรือ ROLLBACK เสร็จสิ้นลง. ถ้าไม่มีเคอร์เซอร์ใดที่เกี่ยวข้องกับ isolation clause แล้ว, ล็อกจะถูกพักไว้จนกระทั่งคำสั่ง SQL สิ้นสุดการทำงาน. ถ้ามีการระบุ USE AND KEEP EXCLUSIVE LOCKS clause พร้อมด้วย *RS หรือระดับการแยก *RR, การล็อกของ UPDATE บนแถวจะถูกกระทำแทนการล็อกของ READ. 			

ข้อมูลที่เกี่ยวข้อง

DECLARE CURSOR statement

ระดับการแยกกัน

Commitment control

Savepoints

savepoint คือ entity ที่ถูกตั้งชื่อเพื่อแสดงสถานะของข้อมูล และแบบแผนที่จุดเวลาภายในหน่วยของงาน. คุณสามารถสร้าง savepoint ภายใน transaction ได้. ถ้า transaction ทำการ roll back, การเปลี่ยนแปลงต่างๆ จะไม่สำเร็จ และกลับไปยัง savepoint ที่ระบุไว้, แทนที่จะกลับไปจุดเริ่มต้นของ transaction.

Savepoint จะถูกตั้งค่าโดยใช้คำสั่ง SAVEPOINT SQL. ตัวอย่างเช่น, การสร้าง savepoint ที่ชื่อ STOP_HERE:

```
SAVEPOINT STOP_HERE
ON ROLLBACK RETAIN CURSORS
```

ตรรกะของโปรแกรมในแอพลิเคชันจะกำหนดว่าจะใช้ชื่อของ savepoint อีกครั้งในการระบุความก้าวหน้าของแอพลิเคชัน, หรือถ้าชื่อของ savepoint ระบุตำแหน่งการทำงาน เฉพาะในแอพลิเคชันก็ไม่ควรที่จะนำกลับมาใช้อีกครั้งหนึ่ง.

ถ้า savepoint นั้นแทนตำแหน่งของงานเพียงหนึ่งเดียวที่ไม่สมควรจะถูกย้ายไปด้วยคำสั่ง SAVEPOINT อื่นๆ, ให้ระบุคีย์เวิร์ด UNIQUE. การทำเช่นนี้เป็นการป้องกันการนำชื่อมาใช้ใหม่โดยไม่ได้ตั้งใจซึ่งสามารถเกิดขึ้นโดยการเรียก stored procedure

ซึ่งใช้ชื่อเดียวกับ savepoint นี้ในคำสั่ง SAVEPOINT. อย่างไรก็ตาม, ถ้าคำสั่ง SAVEPOINT ถูกนำไปใช้ใน loop, ก็ไม่ควรใช้คีย์เวิร์ด UNIQUE. คำสั่ง SQL ต่อไปนี้ตั้งชื่อ savepoint ที่ไม่ซ้ำกันที่ชื่อ START_OVER.

```
SAVEPOINT START_OVER UNIQUE
ON ROLLBACK RETAIN CURSORS
```

ในการ rollback ไปยัง savepoint, ให้ใช้คำสั่ง ROLLBACK ตามด้วย TO SAVEPOINT clause. ตัวอย่างต่อไปนี้แสดงการใช้ SAVEPOINT และคำสั่ง ROLLBACK TO SAVEPOINT:

ตรรกะของแอปพลิเคชันนี้จะจองที่นั่งของสายการบินในวันที่ต้องการ, และจะทำการจองโรงแรม. ถ้าไม่มีที่พักรว่างในโรงแรม, จะ roll back การจองที่นั่งของสายการบินและทำการดำเนินการอีกครั้งสำหรับวันอื่นแทน. ระบบจะพยายามทำเช่นนี้ 3 ครั้ง.

```
got_reservations = 0;
EXEC SQL SAVEPOINT START_OVER UNIQUE ON ROLLBACK RETAIN CURSORS;

    if (SQLCODE != 0) return;

for (i=0; i<3 & got_reservations == 0; ++i)
{
    Book_Air(dates(i), ok);
    if (ok)
    {
        Book_Hotel(dates(i), ok);
        if (ok) got_reservations = 1;
        else
        {
            EXEC SQL ROLLBACK TO SAVEPOINT START_OVER;
            if (SQLCODE != 0) return;
        }
    }
}

EXEC SQL RELEASE SAVEPOINT START_OVER;
```

Savepoint จะถูกยกเลิกโดยคำสั่ง RELEASE SAVEPOINT. ถ้าไม่ใช้คำสั่ง RELEASE SAVEPOINT เพื่อยกเลิก savepoint อย่างชัดเจน, คำสั่งดังกล่าวจะถูกเรียกเมื่อสิ้นสุดระดับ savepoint ปัจจุบันหรือสิ้นสุด transaction. คำสั่งต่อไปนี้จะยกเลิก savepoint START_OVER.

```
RELEASE SAVEPOINT START_OVER
```

Savepoint จะถูกเลิกใช้เมื่อ transaction ถูก commit หรือถูก roll back. ทันทีที่ชื่อของ savepoint ถูกยกเลิก, คุณจะ rollback กลับไปยังชื่อของ savepoint นั้นไม่ได้อีก. คำสั่ง COMMIT หรือ ROLLBACK จะยกเลิกชื่อ savepoint ทั้งหมดที่กำหนดไว้ใน transaction. เนื่องจากชื่อของ savepoint ทั้งหมดจะถูกยกเลิกภายใน transaction, ชื่อเหล่านั้นจึงสามารถนำไปใช้อีกครั้งในการ commit หรือ rollback.

Savepoint จะถูกกำหนดขอบเขตไว้สำหรับการเชื่อมต่อแบบเดี่ยวเท่านั้น. ในทันทีที่ savepoint ถูกกำหนดขึ้น, savepoint จะไม่ถูกกระจายไปยังฐานข้อมูลแบบรีโมตทั้งหมดที่เชื่อมต่อกับแอปพลิเคชันนั้น. Savepoint จะใช้ได้กับฐานข้อมูลปัจจุบันที่แอปพลิเคชันนั้นเชื่อมต่ออยู่เมื่อ savepoint ถูกกำหนดขึ้นเท่านั้น.

คำสั่งเดี่ยวสามารถเรียกใช้ ฟังก์ชันที่ผู้ใช้กำหนด, ทรริกเกอร์, หรือกระบวนการที่บันทึกไว้ได้ทั้งโดยนัยหรือโดยชัดเจน. โดยรู้จักกันในชื่อของ nesting หรือ การซ้อนภายใน. ในบางกรณีเมื่อการซ้อนภายในระดับใหม่ถูกเริ่มขึ้น, ระดับของ savepoint ใหม่จะถูกเริ่มตามไปด้วย. ระดับของ savepoint ใหม่จะแยกการเรียกใช้แอ็พพลิเคชันจากการทำงานใดๆ ของ savepoint โดยใช้รัฐที่นหรือ ทรริกเกอร์ระดับที่ต่ำกว่า.

Savepoint สามารถถูกอ้างอิงได้เฉพาะในระดับของ savepoint เดียวกัน (หรือขอบเขตเดียวกัน) กับที่ระบุไว้ savepoint นั้นไว้. คุณไม่สามารถนำคำสั่ง ROLLBACK TO SAVEPOINT มาใช้ในการ rollback ไปยัง savepoint ที่กำหนดไว้ภายนอกระดับปัจจุบันของ savepoint นั้น. ในลักษณะเดียวกัน, คำสั่ง RELEASE SAVEPOINT ไม่สามารถถูกนำมาใช้ในการเรียกใช้ savepoint ที่กำหนดไว้ภายนอกระดับปัจจุบันของ savepoint นั้น. ตารางต่อไปนี้จะช่วยสรุปว่าเมื่อใดที่ระดับของ savepoint จะถูกเริ่มขึ้นและสิ้นสุดลง:

ระดับใหม่ของ savepoint จะเริ่มขึ้นเมื่อ:	ระดับ savepoint นั้นสิ้นสุดลงเมื่อ:
หน่วยการทำงานใหม่เริ่มทำงาน	มีการเรียกใช้ COMMIT หรือ ROLLBACK
มีการเรียกใช้ ทรริกเกอร์	ทรริกเกอร์เสร็จสิ้นลง
มีการเรียกใช้ฟังก์ชันที่ผู้ใช้ระบุ	มีการส่งฟังก์ชันที่ผู้ใช้ระบุกลับคืนไปยังผู้เรียกใช้งาน
กระบวนการที่ถูบันทึกไว้จะถูกเรียกมาใช้, และกระบวนการนั้นจะถูกสร้างขึ้นโดย NEW SAVEPOINT LEVEL clause	กระบวนการที่ถูบันทึกไว้จะถูกส่งกลับไปยังผู้เรียกใช้งาน
จะมี BEGIN สำหรับคำสั่ง ATOMIC compound SQL	จะมี END สำหรับคำสั่ง ATOMIC compound

savepoint ที่ถูกกำหนดขึ้นในระดับ savepoint จะยกเลิกโดยนัยเมื่อระดับ savepoint นั้นสิ้นสุดลง.

Atomic operations

ในการรันภายใต้ COMMIT(*CHG), COMMIT(*CS), or COMMIT(*ALL), ปฏิบัติการทั้งหมดถือว่าเป็นแบบ atomic.

นั่นคือ, ปฏิบัติการเหล่านั้นจะเสร็จสิ้นการทำงานหรือเสมือนว่าไม่มีการเริ่มการทำงาน. ลักษณะดังกล่าวจะเป็นจริงโดยไม่ต้องคำนึงถึงว่าฟังก์ชันถูกทำให้สิ้นสุดลงหรือถูกขัดจังหวะเมื่อไรหรืออย่างไร (ได้แก่ ไฟฟ้าขัดข้อง, การสิ้นสุดการทำงานแบบไม่ปกติ, หรือถูกยกเลิก).

ถ้า COMMIT (*NONE) ถูกระบุ, อย่างไรก็ตาม, ฐานข้อมูลที่เป็นรากฐานของฟังก์ชันของ data definition บางตัวจะไม่นับเป็น atomic. คำสั่ง SQL data definition ต่อไปนี้ถือว่าเป็นแบบ atomic:

- ALTER TABLE (ดูในหมายเหตุ 1)
- COMMENT ON (ดูในหมายเหตุ 2)
- LABEL ON (ดูในหมายเหตุ 2)
- GRANT (ดูในหมายเหตุ 3)
- REVOKE (ดูในหมายเหตุ 3)
- DROP TABLE (ดูในหมายเหตุ 4)
- DROP VIEW (ดูในหมายเหตุ 4)
- DROP INDEX
- DROP PACKAGE

- REFRESH TABLE

หมายเหตุ:

1. ถ้าจำเป็นต้องเพิ่มหรือลบข้อจำกัดใดๆ, รวมทั้งการเปลี่ยน definition ของคอลัมน์, ปฏิบัติการเหล่านั้นจะถูกประมวลผลครั้งละหนึ่งปฏิบัติการ, ดังนั้นคำสั่ง SQL ทั้งหมดไม่เป็นแบบ atomic. ลำดับของปฏิบัติการจะเป็นดังนี้:
 - ลบข้อจำกัดออก
 - ละคอลัมน์ที่ถูกระบุตัวเลือก RESTRICT เอาไว้
 - Definition ของคอลัมน์อื่นๆ มีการเปลี่ยนแปลง (DROP COLUMN CASCADE, ALTER COLUMN, ADD COLUMN)
 - เพิ่มข้อจำกัด
2. ถ้าคอลัมน์หลายๆ คอลัมน์ถูกระบุในคำสั่ง COMMENT ON หรือ LABEL ON, คอลัมน์เหล่านั้นจะถูกประมวลผลครั้งละหนึ่งคอลัมน์, ดังนั้นคำสั่ง SQL ทั้งหมดจึงไม่เป็น atomic, แต่ COMMENT ON หรือ LABEL ON ในแต่ละคอลัมน์หรืออ็อบเจกต์จะถือว่าเป็น atomic.
3. ถ้าตารางหลายตาราง, แพ็คเกจ SQL หลายแพ็คเกจ, หรือ ผู้ใช้หลายๆ คนถูกระบุไว้ในคำสั่ง GRANT หรือ REVOKE, ตารางจะถูกประมวลผลครั้งละหนึ่งตาราง, ดังนั้นคำสั่ง SQL ทั้งหมดจึงไม่เป็น atomic, แต่ GRANT หรือ REVOKE ของแต่ละตารางจะถือว่าเป็น atomic.
4. ถ้าต้องมีการละม่อมอง dependent ที่ไปในขณะที่ DROP TABLE หรือ DROP VIEW, แต่ละม่อมอง dependent จะถูกประมวลผลครั้งละหนึ่งม่อมอง, ดังนั้นคำสั่ง SQL ทั้งหมดจึงไม่เป็น atomic.

คำสั่ง data definition ต่อไปนี้ไม่เป็นแบบ atomic เนื่องจากคำสั่งเหล่านั้นเกี่ยวพันกับปฏิบัติการทางฐานข้อมูลมากกว่าหนึ่งฐานข้อมูล:

- ALTER PROCEDURE
- ALTER SEQUENCE
- CREATE ALIAS
- CREATE DISTINCT TYPE
- CREATE FUNCTION
- CREATE INDEX
- CREATE PROCEDURE
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- DROP ALIAS
- DROP DISTINCT TYPE
- DROP FUNCTION
- DROP PROCEDURE

- DROP SCHEMA
- DROP SEQUENCE
- DROP TRIGGER
- RENAME (ดูใน note 1)

หมายเหตุ:

1. RENAME เป็น atomic ก็ต่อเมื่อชื่อหรือชื่อของระบบถูกเปลี่ยนแปลง. เมื่อสองสิ่งนี้ถูกเปลี่ยนไป, RENAME ไม่ถือว่าเป็น atomic.

ตัวอย่างเช่น, CREATE TABLE สามารถถูกขัดจังหวะหลังจากที่ไฟล์ฟิสิคัลของ DB2 UDB for iSeries ถูกสร้างขึ้น, ก่อนที่รายการย่อยจะถูกเพิ่มเข้าไป. ดังนั้น, ในกรณีของคำสั่งที่เกี่ยวกับการสร้าง, ถ้าปฏิบัติการใดสิ้นสุดลงโดยไม่ปกติ, คุณจำเป็นต้องจะถูกละออบเจกต์ทิ้งไป และสร้างขึ้นใหม่อีกครั้ง. ในกรณีของคำสั่ง DROP SCHEMA, คุณจำเป็นต้องทำคำสั่งซ้ำหรือใช้คำสั่ง CL command Delete Library (DLTLIB) เพื่อลบส่วนที่ค้างอยู่ของแบบแผนออกไป.

ข้อจำกัด

DB2 UDB for iSeries สนับสนุนข้อจำกัดที่เป็นเอกลักษณ์, อ้างอิงได้, และเป็นตัวตรวจสอบข้อจำกัด.

ข้อจำกัดที่เป็นเอกลักษณ์นั้นเป็นกฎที่รับประกันว่าค่าของคีย์มีเพียงหนึ่งเดียว. ข้อจำกัดที่อ้างอิงได้นั้น คือ กฎที่ว่า foreign key ที่ไม่มีค่าเป็น null ทั้งหมดในตาราง dependent นั้น มี parent key ที่เกี่ยวข้องกันในตาราง parent. ข้อจำกัดการตรวจสอบคือ กฎที่จำกัดค่าที่ใช้ในคอลัมน์หรือกลุ่มคอลัมน์.

DB2 UDB for iSeries จะกำหนดความถูกต้องของข้อจำกัดในช่วงคำสั่ง DML ใดๆ (ภาษาที่ใช้ในการดำเนินการข้อมูล). อย่างไรก็ตาม, ปฏิบัติการบางอย่าง (เช่น การบันทึกตาราง dependent), ทำให้ความถูกต้องของข้อจำกัดนั้นไม่เป็นที่ทราบแน่ชัด. ในกรณีนี้, คำสั่ง DML จะถูกป้องกันจนกระทั่ง DB2 UDB for iSeries ได้รับการตรวจสอบความถูกต้องของข้อจำกัดนั้น.

- ข้อจำกัดเฉพาะจะดำเนินการพร้อมด้วยตรรกะนี้. ถ้าตรรกะหนึ่งที่ใช้กับข้อจำกัดที่เป็นเอกลักษณ์นั้นไม่ถูกต้อง, คำสั่ง Edit Rebuild of Access Paths (EDTRBDAP) จะถูกนำมาใช้ในการแสดงตรรกะนี้ใดๆ ที่จำเป็นต้องสร้างขึ้นใหม่.
- ถ้าในขณะนั้น DB2 UDB for iSeries ไม่ทราบว่า ข้อจำกัดที่อ้างอิงได้หรือข้อจำกัดที่ใช้ตรวจสอบได้มีค่าที่ถูกต้อง, ข้อจำกัดจะถูกนับว่าอยู่ในสถานะรอการตรวจสอบ. คำสั่ง Edit Check Pending Constraints (EDTCPCST) สามารถใช้ในการแสดงผลตรรกะนี้ใดๆ ที่จำเป็นต้องสร้างใหม่ในขณะนั้น.

หลักการที่เกี่ยวข้อง

“ข้อจำกัด” ในหน้า 11

ข้อจำกัด คือกฎที่ตัวจัดการฐานข้อมูลใช้.

การเพิ่ม และการใช้ข้อจำกัดในการตรวจสอบ:

ข้อจำกัดในการตรวจสอบ รับรองถึงความถูกต้องของข้อมูลระหว่างการแทรก และอัปเดตด้วยการจำกัดค่าที่ใช้ได้ในคอลัมน์หรือกลุ่มคอลัมน์.

ใช้คำสั่ง SQL CREATE TABLE และ ALTER TABLE เพื่อเพิ่ม หรือลบข้อจำกัดในการตรวจสอบ.

ในตัวอย่างนี้, คำสั่งต่อไปนี้เป็นการสร้างตารางที่มีสามคอลัมน์ และหนึ่งข้อจำกัดในการตรวจสอบบน COL2 ซึ่งจำกัดค่าที่ใช้ได้ในคอลัมน์นั้นให้เป็นจำนวนเต็มบวก:

```
CREATE TABLE T1 (COL1 INT, COL2 INT CHECK (COL2>0), COL3 INT)
```

จากตาราง, คำสั่งต่อไปนี้:

```
INSERT INTO T1 VALUES (-1, -1, -1)
```

ใช้ไม่ได้เนื่องจากค่าที่ต้องใส่ใน COL2 ไม่เป็นไปตามข้อจำกัดในการตรวจสอบ; กล่าวคือ, -1 ไม่ได้มากกว่า 0.

คำสั่งต่อไปนี้ใช้ได้:

```
INSERT INTO T1 VALUES (1, 1, 1)
```

แต่เมื่อจะแทรกแถว, คำสั่งต่อไปนี้จะใช้ไม่ได้:

```
ALTER TABLE T1 ADD CONSTRAINT C1 CHECK (COL1=1 AND COL1<COL2)
```

คำสั่ง ALTER TABLE พยายามที่จะเพิ่มข้อจำกัดในการตรวจสอบข้อที่สองซึ่งจำกัดค่าที่ใช้ได้ใน COL1 ไว้ที่ 1 และยังบังคับว่าค่าใน COL2 ต้องมากกว่า 1. ข้อจำกัดนี้ใช้ไม่ได้เนื่องจากส่วนที่สองของข้อจำกัด ไม่ตรงตามข้อมูลที่มีอยู่ (ค่า '1' ใน COL2 ไม่น้อยกว่าค่า '1' ใน COL1).

ข้อมูลที่เกี่ยวข้อง

ALTER TABLE statement

CREATE TABLE statement

การบันทึก/การเรียกคืน

ฟังก์ชันในการบันทึก/ การเรียกคืนของ i5/OS ถูกนำมาใช้ในการบันทึกตาราง, มุมมอง, ตรีรชนี, เจอร์นัล, journal receivers, ลำดับ, แพ็คเกจ SQL, SQL โพรซีเจอร์, SQL ทริกเกอร์, ฟังก์ชันที่ถูกระบุโดยผู้ใช้, ประเภทที่ถูกระบุโดยผู้ใช้, และ แบบแผนที่อยู่ในดิสก์ (ไฟล์ที่บันทึกไว้) หรือ ในสื่อบันทึกภายนอก (เทป หรือดิสเก็ต).

เวอร์ชันที่ถูกระบุสามารถถูกเรียกคืนมาไว้ในระบบ iSeries ใดๆ ในภายหลัง. ฟังก์ชันการบันทึก/เรียกคืนจะอนุญาตให้บันทึกคอลเล็กชันทั้งหมด, อ็อบเจกต์ที่ถูกเลือก, หรือ อ็อบเจกต์ที่ถูกเปลี่ยนแปลงตั้งแต่วันที่และเวลาที่กำหนดไว้ทั้งหมด. ข้อมูลทั้งหมดที่จำเป็นในการเรียกคืนของ อ็อบเจกต์หนึ่งๆ ไปยังสถานะก่อนหน้าจะถูกบันทึกไว้. ฟังก์ชันนี้สามารถใช้ในการกู้ข้อมูลคืนจากความเสียหายในตารางแต่ละส่วนโดยเรียกคืนข้อมูลของตารางเวอร์ชันก่อนหน้าหรือ คอลเล็กชันทั้งหมด.

โปรแกรมหรือเซอวิสโปรแกรมที่ถูกสร้างสำหรับ SQL โพรซีเจอร์, สำหรับฟังก์ชัน SQL, หรือสำหรับฟังก์ชันที่เป็นซอร์สที่ถูกเรียกคืนมา, จะถูกเพิ่มเข้าไปโดยอัตโนมัติในแค็ตตาล็อกของ SYSRoutines และ SYSPARMS, ถ้าไม่มีโพรซีเจอร์หรือฟังก์ชันที่มีลายมือหรือชื่อโปรแกรมเดียวกันนั้นอยู่ก่อน. โปรแกรม SQL ที่ถูกสร้างขึ้นใน QSYS จะไม่ถูกสร้างให้เป็นโพรซีเจอร์ของ SQL เมื่อถูกเรียกคืน. นอกจากนี้, โปรแกรมภายนอกหรือเซอวิสโปรแกรมที่ถูกอ้างอิงในคำสั่ง CREATE PROCEDURE หรือ CREATE FUNCTION จะเก็บข้อมูลที่จำเป็นในการลงทะเบียนรูทีนใน SYSRoutines. ถ้าข้อมูลปรากฏโดยมีลายมือชื่อเดียวกัน, ฟังก์ชันหรือโพรซีเจอร์จะถูกเพิ่มเข้าไปใน SYSRoutines และ SYSPARMS เมื่อถูกเรียกคืน.

เมื่อตาราง SQL ถูกเรียกคืนมา, definition ของ SQL ทริกเกอร์ที่ถูกกำหนดสำหรับตารางนั้นก็จะถูกเรียกคืนมาด้วย. definition ของ SQL ทริกเกอร์จะถูกเพิ่มเข้าไปโดยอัตโนมัติในแค็ตตาล็อกของ SYSTRIGGERS, SYSTRIGDEP, SYSTRIGCOL, และ SYSTRIGUPD. อ็อบเจกต์ของโปรแกรมที่ถูกสร้างขึ้นจากคำสั่ง SQL CREATE TRIGGER จะต้องถูกบันทึกและเรียกคืนเมื่อตาราง SQL ถูกบันทึกและเรียกคืน. การบันทึกและการเรียกคืนอ็อบเจกต์ของโปรแกรมไม่ได้ถูกทำให้เป็นอัตโนมัติโดยตัวจัดการฐานข้อมูล. ควรทบทวนข้อควรระวังสำหรับทริกเกอร์แบบอ้างอิงถึงตัวเองเมื่อมีการเรียกคืนตาราง SQL ไปยังไลบรารีใหม่.

เมื่ออ็อบเจกต์ *SQLUDT ถูกเรียกคืนสำหรับประเภทที่ผู้ใช้กำหนด, ประเภทดังกล่าวจะถูกเพิ่มเข้าโดยอัตโนมัติในแค็ตตาล็อก SYSTYPES. ฟังก์ชันที่เหมาะสมสำหรับการแปลงประเภท ที่ผู้ใช้กำหนดและประเภทต้นฉบับจำเป็นต้องถูกสร้างขึ้นด้วย, トラバิตที่ไม่ปรากฏชนิดและฟังก์ชันอยู่แล้ว.

เมื่อค่า *DTAARA สำหรับลำดับนั้นถูกกู้คืน, ลำดับนั้นจะถูกเพิ่มเข้าไปในแค็ตตาล็อก SYSSEQUENCES อย่างอัตโนมัติ. ถ้าการอัปเดตแค็ตตาล็อกไม่เป็นผลสำเร็จ, จะมีการตัดแปลง *DTAARA เพื่อไม่ให้นำมาใช้เป็นลำดับ และจะมีข้อความ SQL9020 ปรากฏอยู่ในบันทึกการใช้งาน.

โปรแกรม SQL แบบกระจายหรือแพ็คเกจ SQL แบบเชื่อมโยงสามารถบันทึกและเรียกคืนให้กับระบบที่ระบบก็ได้. ลักษณะนี้จะอนุญาตให้สำเนาโปรแกรมของโปรแกรม SQL เท่าไรก็ได้จากระบบต่างๆ กันสามารถเข้าถึงแพ็คเกจ SQL เดียวกันบนแอสเพล็กซ์เซิร์ฟเวอร์เดียวกัน. ทำให้โปรแกรม SQL แบบกระจาย สามารถเชื่อมต่อกับแอสเพล็กซ์เซิร์ฟเวอร์จำนวนเท่าใดก็ได้ที่เรียกคืนแพ็คเกจ SQL มาแล้ว (สามารถใช้ CRTSQLPKG ได้เช่นกัน). แพ็คเกจ SQL ไม่สามารถเรียกคืนไปยังไลบรารีที่ต่างกัน.

หมายเหตุ: การเรียกคืนแบบแผนหนึ่งไปยังไลบรารีที่มีอยู่หรือไปยังแบบแผนที่มีชื่อต่างกันจะไม่ทำการเรียกคืนเจอร์นัล, journal receiver, หรือ IDDU dictionary (ถ้ามีปรากฏอยู่). ถ้าแบบแผนถูกเรียกคืนมายังแบบแผนที่มีชื่อต่างกัน, มุมมองของแค็ตตาล็อกในแบบแผนนั้นจะแสดงให้เห็นเพียงอ็อบเจกต์ในแบบแผนเท่านั้น. อย่างไรก็ตาม, มุมมองของแค็ตตาล็อกใน QSYS2, จะแสดงอ็อบเจกต์ทั้งหมดอย่างเหมาะสม.

การต้านทานความเสียหาย

เซิร์ฟเวอร์จะมีวิธีลด หรือกำจัดความเสียหายที่เกิดจากความผิดพลาดของดิสก์ได้หลายวิธีด้วยกัน.

ตัวอย่างเช่น, การทำ mirror, checksum, และการทำ RAID ดิสก์เพื่อลดโอกาสเกิดปัญหาเกี่ยวกับดิสก์. ฟังก์ชัน DB2 UDB for iSeries มีค่าความต้านทานความเสียหายที่เกิดจากความผิดพลาดของดิสก์หรือความผิดพลาดของระบบได้ในระดับหนึ่ง.

ปฏิบัติการ DROP มักประสบความสำเร็จอยู่เสมอ, โดยไม่จำเป็นต้องคำนึงถึงความเสียหายใดๆ. นี่เป็นการตรวจสอบให้แน่ใจว่าความเสียหายที่เกิดขึ้น, อย่างน้อยตาราง, มุมมอง, แพ็คเกจ SQL, ดรรชนี, โพรซีเจอร์, ฟังก์ชัน, หรือ ประเภทที่แตกต่างกัน จะลบออกได้และเรียกกลับคืนมาหรือสร้างใหม่ได้อีกครั้ง.

ในกรณีที่ความผิดพลาดของดิสก์ทำความเสียหายให้กับส่วนเล็กๆ ของแถวข้อมูลในตาราง, ตัวจัดการฐานข้อมูลของ DB2 UDB for iSeries จะอนุญาตให้อ่านแถวเหล่านั้นได้.

Index recovery

DB2 UDB for iSeries สนับสนุนหลายๆ ฟังก์ชันที่ใช้ในการกู้คืนดรรชนี.

- ระบบจัดการการปกป้องดรรชนี

คำสั่ง CL ที่เป็น EDTRCYAP อนุญาตให้ผู้ใช้สามารถออกคำสั่งให้ DB2 UDB for iSeries รับประกันว่าเมื่อเกิดความผิดพลาดของระบบหรือไฟฟ้า, ระยะเวลาที่ใช้ในการกู้คืนดรรชนีทั้งหมดในระบบจะต้องต่ำกว่าเวลาที่ระบุไว้. ระบบจะทำการบันทึกข้อมูลที่เพียงพอลงในเจอร์นัลของระบบโดยอัตโนมัติเพื่อจำกัดเวลาในการกู้คืนให้อยู่ในจำนวนที่ระบุไว้.

- การทำเจอร์นัลดรรชนี

DB2 UDB for iSeries สนับสนุนฟังก์ชันการทำเจอร์นัลดรรชนีที่ทำให้คุณไม่ต้องสร้างดรรชนีทั้งหมดใหม่เนื่องจากกระแสไฟฟ้าขัดข้องหรือระบบขัดข้อง. ถ้าดรรชนีถูกบันทึก, การสนับสนุนฐานข้อมูลของระบบจะตรวจสอบให้แน่ใจว่าดรรชนีนั้น

เชื่อมโยงกับข้อมูลในตารางโดยไม่ต้องสร้างใหม่จากจุดเริ่มต้น (scratch). ตรรกะของ SQL จะ *ไม่*มีการทำเจอร์นัลโดยอัตโนมัติ. อย่างไรก็ตาม, คุณสามารถ, ใช้คำสั่ง CL ที่เป็น Start Journal Access Path (STRJRNAP) ในการทำเจอร์นัลตรรกะนี้ได้ๆ ที่ถูกสร้างขึ้นโดย DB2 UDB for iSeries.

- Index rebuild

ตรรกะทั้งหมดในระบบมีอ็อปชันในการดูแลรักษาซึ่งจะระบุว่าจะมีการรักษาตรรกะเมื่อใดบ้าง. ตรรกะของ SQL จะถูกสร้างด้วยแอ็ททริบิวต์ในการรักษา *IMMED.

ในกรณีที่ไฟฟ้าขัดข้องหรือระบบขัดข้องแบบไม่ปกติ, ถ้าตรรกะไม่ได้ถูกปกป้องไว้โดยเทคนิคใดที่อธิบายมาก่อนหน้านี้, ตรรกะเหล่านี้ในขั้นตอนของการเปลี่ยนแปลงอาจจำเป็นต้องถูกสร้างใหม่โดยตัวจัดการฐานข้อมูลเพื่อให้ถูกต้องตรงกับข้อมูลจริง. ตรรกะทั้งหมดในระบบจะมีอ็อปชันการกู้คืนซึ่งจะระบุว่าตรรกะจะถูกสร้างขึ้นใหม่ถ้าจำเป็น. ตรรกะ SQL ทั้งหมดที่มีแอ็ททริบิวต์ UNIQUE ถูกสร้างขึ้นด้วยแอ็ททริบิวต์การกู้คืนของ *IPL (หมายความว่า ตรรกะนี้เหล่านี้ถูกสร้างขึ้นใหม่ ก่อนที่ระบบปฏิบัติการ OS/400® จะเริ่มทำงาน). ตรรกะ SQL อื่นๆ ทั้งหมดถูกสร้างขึ้นอ็อปชันด้วยตัวเลือกการกู้คืน *AFTIPL (หมายความว่าหลังจากระบบปฏิบัติการเริ่มทำงาน, ตรรกะจะถูกสร้างขึ้นใหม่ในเวลาที่แตกต่างกัน). ในระหว่าง IPL, โอเปอเรเตอร์จะเห็นจอแสดงผลแสดงตรรกะที่จำเป็นต้องถูกสร้างใหม่และอ็อปชันการกู้คืน. โอเปอเรเตอร์สามารถทำการแทนที่ค่าเดิมของอ็อปชันการกู้คืนได้.

- การบันทึกและการเรียกคืนตรรกะ

ฟังก์ชันบันทึก/เรียกคืน อนุญาตให้บันทึกตรรกะได้เมื่อตารางถูกบันทึกโดยการใช้ ACCPTH(*YES) ใน Save Object (SAVOBJ) หรือ Save Library (SAVLIB) ที่เป็นคำสั่ง CL. ในกรณีที่มีการเรียกคืนเมื่อตรรกะถูกบันทึกไว้ด้วย, ไม่มีความจำเป็นที่จะต้องสร้างตรรกะเหล่านั้นขึ้นใหม่. ตรรกะใดๆ ที่ไม่ได้ถูกบันทึกไว้และเรียกคืนก่อนหน้านี้จะถูกสร้างขึ้นใหม่โดยอัตโนมัติในเวลาที่แตกต่างกันโดยตัวจัดการฐานข้อมูล.

ความสมบูรณ์ของแคตตาล็อก

แคตตาล็อกเก็บข้อมูลเกี่ยวกับตาราง, มุมมอง, SQL แพ็กเกจ, ลำดับ, ตรรกะ, โพรซีเจอร์, ฟังก์ชัน, ทรริกเกอร์, และพารามิเตอร์ในแบบแผน.

ตัวจัดการฐานข้อมูลจะตรวจสอบให้แน่ใจว่าข้อมูลในแคตตาล็อกถูกต้องอยู่ตลอดเวลา. ซึ่งสามารถกระทำได้โดยการป้องกันไม่ให้ผู้เปลี่ยนแปลงข้อมูลใดๆ ในแคตตาล็อก และรักษาข้อมูลใน แคตตาล็อกไว้เมื่อมีการเปลี่ยนแปลงเกิดขึ้นกับตาราง, มุมมอง, SQL แพ็กเกจ, ตรรกะ, ประเภท, โพรซีเจอร์, ฟังก์ชัน, ทรริกเกอร์, และพารามิเตอร์ที่อธิบายไว้ในแคตตาล็อก.

ความสมบูรณ์ของแคตตาล็อกจะถูกรักษาไว้ไม่ว่า อ็อบเจ็กต์ในแบบแผนจะถูกเปลี่ยนโดยคำสั่ง SQL, คำสั่ง i5/OS CL, คำสั่ง CL ในสภาวะแวดล้อม System/38™, ฟังก์ชันในสภาวะแวดล้อม System/36, หรือผลิตภัณฑ์อื่นๆ หรืออยู่ที่อื่น ๆ บนระบบ iSeries. ตัวอย่างเช่น, การลบตารางสามารถทำได้โดยการรันคำสั่ง SQL DROP, ออกคำสั่ง DLTF CL สำหรับ i5/OS, ออกคำสั่ง DLTF CL สำหรับ System/38 หรือการเข้าไปใช้อ็อปชัน 4 บนจอแสดงผล WRKF หรือ WRKOBJ. โดยไม่จำเป็นต้องคำนึงถึงอินเตอร์เฟซที่ใช้ในการลบตารางออก, ตัวจัดการฐานข้อมูลจะทิ้งรายละเอียดของตารางออกจากแคตตาล็อก เมื่อทำการลบออก. ต่อไปนี้เป็นรายการฟังก์ชันและผลกระทบที่เชื่อมโยงกันกับแคตตาล็อก:

ตารางที่ 37. ผลกระทบของฟังก์ชันต่างๆ กับแคตตาล็อก

ฟังก์ชัน	ผลกระทบกับแคตตาล็อก
เพิ่มข้อจำกัดเข้าไปในตาราง	ข้อมูลถูกเพิ่มเข้าไปในแคตตาล็อก
ลบข้อจำกัดออกจากตาราง	ข้อมูลที่เกี่ยวข้องจะถูกลบออกจากแคตตาล็อก
สร้างอ็อบเจ็กต์ขึ้นในแบบแผน	ข้อมูลถูกเพิ่มเข้าไปในแคตตาล็อก

ตารางที่ 37. ผลกระทบของฟังก์ชันต่างๆ กับแคตตาล็อก (ต่อ)

ฟังก์ชัน	ผลกระทบกับแคตตาล็อก
ลบอ็อบเจกต์ออกจากแบบแผน	ข้อมูลที่เกี่ยวข้องจะถูกลบออกจากแคตตาล็อก
เรียกคืนอ็อบเจกต์เข้ามาในแบบแผน	ข้อมูลถูกเพิ่มเข้าไปในแคตตาล็อก
การเปลี่ยนแปลงของข้อสังเกตแบบยาวของอ็อบเจกต์	ข้อสังเกตถูกอัปเดตในแคตตาล็อก
การเปลี่ยนแปลงเลเบล (ข้อความ) ของอ็อบเจกต์	เลเบลถูกอัปเดตในแคตตาล็อก
การเปลี่ยนแปลงเจ้าของอ็อบเจกต์	เจ้าของถูกอัปเดตในแคตตาล็อก
การลบอ็อบเจกต์ออกจากแบบแผน	ข้อมูลที่เกี่ยวข้องจะถูกลบออกจากแคตตาล็อก
การย้ายอ็อบเจกต์เข้าไปในแบบแผน	ข้อมูลถูกเพิ่มเข้าไปในแคตตาล็อก
การเปลี่ยนชื่ออ็อบเจกต์	ชื่ออ็อบเจกต์ถูกอัปเดตในแคตตาล็อก

ผู้ใช้ auxiliary storage pool (ASP)

แบบแผนสามารถสร้างขึ้นใน ASP ของผู้ใช้โดยการใช้ ASP clause บนคำสั่ง CREATE COLLECTION และ CREATE SCHEMA.

คำสั่ง CRTLIB สามารถใช้ในการสร้างไลบรารีใน ASP ของผู้ใช้ได้เช่นกัน. ไลบรารีดังกล่าวสามารถใช้ในการรับตาราง SQL, มุมมอง, และตรรกชนี้ได้.

ข้อมูลที่เกี่ยวข้อง

ไฟล์ PDF การสำรองข้อมูลและการกู้คืน

Independent auxiliary storage pool (IASP)

ดิสก์พูลแบบอิสระถูกนำมาใช้ในการติดตั้งฐานข้อมูลของผู้ใช้บนเซิร์ฟเวอร์ iSeries.

ดิสก์พูลอิสระแบ่งเป็น 3 ประเภทด้วยกันได้แก่ : primary, secondary, และ user-defined file system (UDFS). ฐานข้อมูลจะถูกติดตั้งโดยใช้ดิสก์พูลอิสระแบบ primary.

เมื่อใช้เซิร์ฟเวอร์ iSeries, คุณสามารถทำงานกับฐานข้อมูลหลายอัน. เซิร์ฟเวอร์ iSeries มีฐานข้อมูลของระบบ (มักเรียกว่า SYSBAS) และความสามารถในการทำงานกับฐานข้อมูลของผู้ใช้ได้หลายฐานข้อมูล. ฐานข้อมูลของผู้ใช้จะถูกจัดการด้วยเซอร์ฟเวอร์ iSeries ผ่านทางดิสก์พูลอิสระ, โดยถูกติดตั้งไว้ในฟังก์ชัน Disk Management ของ Navigator ใน iSeries. พื้นที่ที่ดิสก์พูลอิสระถูกติดตั้ง, มันจะปรากฏเป็นฐานข้อมูลอีกอันหนึ่งภายใต้ฟังก์ชัน Databases ของ Navigator ใน iSeries.

รูทีน

รูทีน คือโค้ดหรือโปรแกรมที่คุณสามารถเรียกใช้งานได้.

สตอร์โพรซีเดอ์

โพรซีเดอ์ (ซึ่งมักเรียกว่า สตอร์โพรซีเดอ์) คือโปรแกรมที่สามารถเรียกขึ้นมาเพื่อปฏิบัติงาน ซึ่งมีทั้งข้อความภาษาโฮสต์ และคำสั่ง SQL. โพรซีเดอ์ใน SQL มีข้อดีเหมือนกับโพรซีเดอ์ในภาษาโฮสต์.

DB2 SQL สำหรับ iSeries การสนับสนุนโพรซีเจอร์ที่เก็บไว้จะให้บริการสำหรับแอ็พพลิเคชัน SQL เพื่อกำหนดและเรียกโพรซีเจอร์ด้วยข้อความ SQL. โพรซีเจอร์ที่เก็บไว้สามารถนำมาใช้ทั้งใน DB2 SQL แบบกระจายและไม่กระจายสำหรับแอ็พพลิเคชัน iSeries. ข้อดีที่สุดข้อหนึ่งของการใช้โพรซีเจอร์ที่เก็บไว้คือสำหรับแอ็พพลิเคชันแบบกระจายแล้ว, การใช้ข้อความ CALL หนึ่งข้อความบน application requester, หรือไคลเอ็นต์, สามารถทำงานในปริมาณเท่าใดก็ตามบนแอ็พพลิเคชันเซิร์ฟเวอร์.

คุณอาจนิยามโพรซีเจอร์ว่าเป็น SQL โพรซีเจอร์หรือโพรซีเจอร์ภายนอก. โพรซีเจอร์ภายนอกสามารถเป็นโปรแกรมภาษาชั้นสูงที่สนับสนุนใดๆ ก็ตาม (ยกเว้นโปรแกรมและโพรซีเจอร์ System/36*) หรือโพรซีเจอร์ REXX. โพรซีเจอร์ดังกล่าวไม่จำเป็นต้องมีข้อความ SQL, แต่อาจมีข้อความ SQL ได้. SQL โพรซีเจอร์ถูกกำหนดไว้ทั้งหมดใน SQL, และสามารถมีข้อความ SQL ที่รวมเอา SQL control statement ไว้ได้.

การโค้ดโพรซีเจอร์ที่เก็บไว้ที่ผู้ใช้จำเป็นต้องเข้าใจดังนี้:

- บันทึก definition ของโพรซีเจอร์ด้วยข้อความ CREATE PROCEDURE
- บันทึกการเรียกโพรซีเจอร์ด้วยข้อความ CALL
- หลักการผ่านพารามิเตอร์
- วิธีการย้อนกลับสถานะที่สมบูรณ์ไปยังโปรแกรมที่เรียกโพรซีเจอร์.

คุณอาจนิยามโพรซีเจอร์ที่เก็บไว้ด้วยการใช้ข้อความ CREATE PROCEDURE. ข้อความ CREATE PROCEDURE จะเป็นการเพิ่ม definition ของโพรซีเจอร์และพารามิเตอร์ให้กับตารางแคตตาล็อก SYSROUTINES และ SYSPARMS. definition เหล่านี้สามารถเข้าไปได้โดยข้อความ SQL CALL ใดๆ บนระบบ.

ในการสร้างโพรซีเจอร์ภายนอก หรือ SQL โพรซีเจอร์, คุณสามารถใช้คำสั่ง SQL CREATE PROCEDURE.

ส่วนต่อไปนี้เป็นคำอธิบายคำสั่ง SQL ที่ใช้เพื่อกำหนดและเรียกโพรซีเจอร์ที่เก็บไว้, ข้อมูลเกี่ยวกับการผ่านพารามิเตอร์ไปยังโพรซีเจอร์ที่เก็บไว้, และตัวอย่างการใช้โพรซีเจอร์ที่เก็บไว้.

สำหรับข้อมูลเพิ่มเติมเกี่ยวกับโพรซีเจอร์ที่เก็บไว้, โปรดดูโพรซีเจอร์ที่เก็บไว้, ทรigger และฟังก์ชันแบบผู้ใช้กำหนดเองบน

DB2 Universal Database for iSeries PDF 

หลักการที่เกี่ยวข้อง

“โพรซีเจอร์ที่เก็บไว้” ในหน้า 12

โพรซีเจอร์ที่เก็บไว้คือ โปรแกรมที่อาจถูกเรียกโดยใช้คำสั่ง SQL CALL.

สิ่งอ้างอิงที่เกี่ยวข้อง

“ข้อควรพิจารณาสำหรับโพรซีเจอร์ของ DRDA ที่บันทึกไว้” ในหน้า 334

เซิร์ฟเวอร์ iSeries DRDA® สนับสนุนการส่งค่าชุดผลลัพธ์คืนมาจากโพรซีเจอร์ที่บันทึกไว้.

ข้อมูลที่เกี่ยวข้อง

CREATE PROCEDURE statement

Java SQL Routines

การกำหนดโพรซีเจอร์ภายนอก

คำสั่ง CREATE PROCEDURE สำหรับโพรซีเจอร์ภายนอกที่ตั้งชื่อโพรซีเจอร์, ให้กำหนดพารามิเตอร์และแอ็ททริบิวต์ของโพรซีเจอร์, และให้ข้อมูลอื่นๆ เกี่ยวกับโพรซีเจอร์ที่ระบบใช้ เมื่อเรียกโพรซีเจอร์นั้น.

พิจารณาตัวอย่างนี้:

```
CREATE PROCEDURE P1
  (INOUT PARM1 CHAR(10))
  EXTERNAL NAME MYLIB.PROC1
  LANGUAGE C
  GENERAL WITH NULLS
```

คำสั่ง CREATE PROCEDURE:

- ตั้งชื่อโพรซีเจอร์ P1
- กำหนดหนึ่งพารามิเตอร์ซึ่งถูกใช้เป็นตัวอินพุตพารามิเตอร์และเอาต์พุตพารามิเตอร์. พารามิเตอร์คือ ฟิลด์แบบอักขระที่มีความยาวสิบตัวอักษร. สามารถกำหนดพารามิเตอร์ให้เป็นประเภท IN, OUT, หรือ INOUT. จะกำหนดประเภทพารามิเตอร์เมื่อค่าสำหรับพารามิเตอร์ ผ่านไปยังและผ่านจากโพรซีเจอร์.
- กำหนดชื่อของโปรแกรมที่สอดคล้องกับโพรซีเจอร์, คือ PROC1 ใน MYLIB. MYLIB.PROC1 คือโปรแกรมซึ่งถูกเรียกเมื่อมีการเรียกโพรซีเจอร์ด้วยคำสั่ง CALL.
- แสดงว่าโพรซีเจอร์ P1 (โปรแกรม MYLIB.PROC1) ถูกบันทึกลงใน C. ภาษาเป็นสิ่งสำคัญเนื่องจากมีผลต่อประเภทพารามิเตอร์ที่สามารถผ่านไปได้. และยังส่งผลกระทบต่อวิธีการที่พารามิเตอร์ ถูกส่งไปยังโพรซีเจอร์(ตัวอย่างเช่น, สำหรับโพรซีเจอร์ ILE C, NUL-terminator จะถูกส่งด้วยอักขระ, กราฟิค, วันที่, เวลา, และพารามิเตอร์ timestamp).
- กำหนดประเภท CALL ให้เป็น GENERAL WITH NULLS. แสดงว่าพารามิเตอร์สำหรับโพรซีเจอร์อาจมีค่า NULL อยู่, ดังนั้นจึงต้องการให้มีอักขระเพิ่มเติมผ่านไปยังโพรซีเจอร์บนข้อความ CALL. อักขระเพิ่มเติมคืออะเรย์ของจำนวนเต็ม N ที่สั้น, โดยที่ N คือจำนวนพารามิเตอร์ ซึ่งประกาศในคำสั่ง CREATE PROCEDURE. ในตัวอย่างนี้, อะเรย์มีเพียงหนึ่งองค์ประกอบ เนื่องจากมีเพียงพารามิเตอร์เท่านั้น.

เป็นเรื่องสำคัญที่ต้องสังเกตว่าไม่จำเป็นต้องกำหนดโพรซีเจอร์เพื่อเรียกใช้งาน. อย่างไรก็ตาม, หากไม่พบ definition ของโพรซีเจอร์, จาก CREATE PROCEDURE ก่อนหน้านี้หรือจาก DECLARE PROCEDURE ในโปรแกรมนี้, แสดงว่ามีการตั้งข้อบังคับและสมมุติฐานบางอย่างเมื่อเรียกโพรซีเจอร์บนคำสั่ง CALL. ตัวอย่างเช่น, อักขระตัวบ่งชี้ NULL จะไม่สามารถผ่านไปได้.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การใช้คำสั่ง CALL แบบฝังโดยไม่มี definition ของโพรซีเจอร์อยู่” ในหน้า 152

คำสั่ง CALL แบบ static ที่ไม่มีคำสั่ง CREATE PROCEDURE ที่ตรงกันถูกประมวลผลด้วยกฎต่อไปนี้.

การกำหนด SQL โพรซีเจอร์

คำสั่ง CREATE PROCEDURE สำหรับ SQL โพรซีเจอร์ที่ตั้งชื่อโพรซีเจอร์, ให้กำหนดพารามิเตอร์และแอตทริบิวต์, และให้ข้อมูลอื่นๆ เกี่ยวกับโพรซีเจอร์ที่ใช้ เมื่อเรียกใช้โพรซีเจอร์, และกำหนดโครงโพรซีเจอร์. *โครงโพรซีเจอร์*คือส่วนที่สามารถเรียกทำงานได้ของโพรซีเจอร์และเป็นคำสั่ง SQL แบบเดี่ยว.

พิจารณาตัวอย่างง่ายๆ ต่อไปนี้ ซึ่งใช้อินพุตเป็นหมายเลขพนักงาน และอัปเดตเงินเดือนพนักงาน:

```
CREATE PROCEDURE UPDATE_SALARY_1
  (IN EMPLOYEE_NUMBER CHAR(10),
  IN RATE DECIMAL(6,2))
  LANGUAGE SQL MODIFIES SQL DATA
  UPDATE CORPDATA.EMPLOYEE
  SET SALARY = SALARY * RATE
  WHERE EMPNO = EMPLOYEE_NUMBER
```

ข้อความ CREATE PROCEDURE:

- ตั้งชื่อโพรซีเจอร์เป็น UPDATE_SALARY_1.
- กำหนดพารามิเตอร์ EMPLOYEE_NUMBER ซึ่งเป็นอินพุตพารามิเตอร์และเป็นประเภทข้อมูลอักขระ ความยาว 6 ตัวอักษรและพารามิเตอร์ RATE ซึ่งเป็นอินพุตพารามิเตอร์และจัดอยู่ในประเภทข้อมูลทศนิยม.
- แสดงว่าโพรซีเจอร์นี้คือ SQL โพรซีเจอร์ซึ่งแก้ไขข้อมูล SQL.
- กำหนดโครงสร้างโพรซีเจอร์เป็นข้อความ UPDATE แบบเดี่ยว. เมื่อมีการเรียกโพรซีเจอร์, ข้อความ UPDATE จะถูกเรียกขึ้นมาด้วยการใช้ค่าที่ส่งต่อสำหรับ EMPLOYEE_NUMBER และ RATE.

แทนที่จะใช้ข้อความ UPDATE แบบเดี่ยว, สามารถเพิ่มตรรกะนี้ให้กับ SQL โพรซีเจอร์ด้วยการใช้ SQL control statement. SQL control statement ประกอบด้วย:

- คำสั่งการกำหนด
- คำสั่ง CALL
- คำสั่ง CASE
- คำสั่งผสม
- คำสั่ง FOR
- คำสั่ง GET DIAGNOSTICS
- คำสั่ง GOTO
- คำสั่ง IF
- คำสั่ง ITERATE
- คำสั่ง LEAVE
- คำสั่ง LOOP
- คำสั่ง REPEAT
- คำสั่ง RESIGNAL
- คำสั่งข้อ RETURN
- คำสั่ง SIGNAL
- คำสั่ง WHILE

ใช้ตัวอย่างต่อไปนี้เป็นอินพุตของหมายเลขพนักงานและการจัดอันดับซึ่งได้รับการประเมิน ล่าสุด. โพรซีเจอร์ใช้คำสั่ง CASE เพื่อกำหนดการเพิ่มและโบนัสสำหรับอัปเดต.

```
CREATE PROCEDURE UPDATE_SALARY_2
  (IN EMPLOYEE_NUMBER CHAR(6),
  IN RATING INT)
LANGUAGE SQL MODIFIES SQL DATA
CASE RATING
  WHEN 1 THEN
    UPDATE CORPDATA.EMPLOYEE
      SET SALARY = SALARY * 1.10,
          BONUS = 1000
      WHERE EMPNO = EMPLOYEE_NUMBER;
  WHEN 2 THEN
    UPDATE CORPDATA.EMPLOYEE
      SET SALARY = SALARY * 1.05,
```



```

        BONUS = 500
    WHERE EMPNO = EMPLOYEE_NUMBER;
ELSE
    UPDATE CORPDATA.EMPLOYEE
        SET SALARY = SALARY * 1.03,
            BONUS = 0
    WHERE EMPNO = EMPLOYEE_NUMBER;
END CASE

```

คำสั่ง CREATE PROCEDURE:

- ตั้งชื่อโพรซีเจอร์เป็น UPDATE_SALARY_2.
- กำหนดพารามิเตอร์ EMPLOYEE_NUMBER ซึ่งเป็นอินพุตพารามิเตอร์และเป็นประเภทข้อมูลอักขระ ความยาว 6 ตัว อักขระและพารามิเตอร์ RATING ซึ่งเป็นอินพุตพารามิเตอร์และจัดอยู่ในประเภทข้อมูลจำนวนเต็ม.
- แสดงว่าโพรซีเจอร์นี้คือ SQL โพรซีเจอร์ซึ่งแก้ไขข้อมูล SQL.
- กำหนดโครงสร้างโพรซีเจอร์. เมื่อมีการเรียกโพรซีเจอร์นี้, อินพุตพารามิเตอร์ RATING จะถูกตรวจสอบและข้อความอัปเดตที่เหมาะสมจะถูกเรียกใช้งาน.

สามารถเพิ่มข้อความจำนวนมากให้กับโครงสร้างโพรซีเจอร์ด้วยการเพิ่มข้อความผสม. ภายในข้อความผสม, สามารถระบุจำนวนคำสั่ง SQL ใดๆ ก็ได้. นอกจากนี้, ยังสามารถประกาศตัวแปร SQL, เคอร์เซอร์, และ handler ได้.

ใช้ตัวอย่างต่อไปนี้เป็นอินพุตหมายเลขแผนก. ซึ่งจะรายงานเงินเดือนโดยรวมของพนักงานทั้งหมดในแผนกนั้น รวมทั้งจำนวนพนักงานในแผนกที่ได้โบนัส.

```

CREATE PROCEDURE RETURN_DEPT_SALARY
    (IN DEPT_NUMBER CHAR(3),
    OUT DEPT_SALARY DECIMAL(15,2),
    OUT DEPT_BONUS_CNT INT)
LANGUAGE SQL READS SQL DATA
P1: BEGIN
    DECLARE EMPLOYEE_SALARY DECIMAL(9,2);
    DECLARE EMPLOYEE_BONUS DECIMAL(9,2);
    DECLARE TOTAL_SALARY DECIMAL(15,2)DEFAULT 0;
    DECLARE BONUS_CNT INT DEFAULT 0;
    DECLARE END_TABLE INT DEFAULT 0;
    DECLARE C1 CURSOR FOR
        SELECT SALARY, BONUS FROM CORPDATA.EMPLOYEE
            WHERE WORKDEPT = DEPT_NUMBER;
    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET END_TABLE = 1;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        SET DEPT_SALARY = NULL;
    OPEN C1;
    FETCH C1 INTO EMPLOYEE_SALARY, EMPLOYEE_BONUS;
    WHILE END_TABLE = 0 DO
        SET TOTAL_SALARY = TOTAL_SALARY + EMPLOYEE_SALARY + EMPLOYEE_BONUS;
        IF EMPLOYEE_BONUS > 0 THEN
            SET BONUS_CNT = BONUS_CNT + 1;
        END IF;
        FETCH C1 INTO EMPLOYEE_SALARY, EMPLOYEE_BONUS;
    END WHILE;

```

```

CLOSE C1;
SET DEPT_SALARY = TOTAL_SALARY;
SET DEPT_BONUS_CNT = BONUS_CNT;
END P1

```

คำสั่ง CREATE PROCEDURE:

- ตั้งชื่อโพรซีเจอร์เป็น RETURN_DEPT_SALARY.
- กำหนดพารามิเตอร์ DEPT_NUMBER ซึ่งเป็นอินพุตพารามิเตอร์และเป็นประเภทข้อมูลอักขระ ความยาว 3 ตัวอักษร, พารามิเตอร์ DEPT_SALARY ซึ่งเป็นเอาต์พุตพารามิเตอร์และจัดอยู่ในประเภทข้อมูล ทศนิยม, และพารามิเตอร์ DEPT_BONUS_CNT ซึ่งเป็นเอาต์พุตพารามิเตอร์และเป็นประเภทข้อมูลจำนวนเต็ม.
- แสดงว่าโพรซีเจอร์นี้คือ SQL โพรซีเจอร์ซึ่งอ่านข้อมูล SQL
- กำหนดโครงโพรซีเจอร์.
 - ประกาศให้ SQL ผันแปรกับ EMPLOYEE_SALARY และ TOTAL_SALARY เป็นฟิลด์แบบทศนิยม.
 - ประกาศให้ SQL ผันแปรกับ BONUS_CNT และ END_TABLE ซึ่งเป็นจำนวนเต็มและถูก initialize เป็น 0.
 - ประกาศ C1 ซึ่งเลือกคอลัมน์จากตารางพนักงาน.
 - ประกาศ handler แบบต่อเนื่องสำหรับ NOT FOUND, ซึ่ง, เมื่อถูกเรียกจะกำหนดค่าผันแปร END_TABLE เป็น 1. handler นี้จะถูกเรียกเมื่อ FETCH ไม่มีแถวเหลืออยู่สำหรับส่งคืน. หาก handler นี้ถูกเรียกขึ้นมา, SQLCODE และ SQLSTATE จะถูก initialize ใหม่เป็น 0.
 - ประกาศ exit handler สำหรับ SQLEXCEPTION. หากถูกเรียก, DEPT_SALARY จะถูกตั้งให้เป็น NULL และการประมวลผลข้อความผสมถูกทำให้จบ. handler นี้จะถูกเรียกเมื่อมีข้อผิดพลาดเกิดขึ้น, กล่าวคือ, คลาส SQLSTATE ไม่ใช่ '00', '01' หรือ '02'. เนื่องจากตัวบ่งชี้จะถูกส่งไปยัง SQL โพรซีเจอร์เสมอ, ค่าตัวบ่งชี้สำหรับ DEPT_SALARY จึงเท่ากับ -1 เมื่อโพรซีเจอร์ส่งคืนค่า. หาก handler นี้ถูกเรียกขึ้นมา, SQLCODE และ SQLSTATE จะถูก initializ ใหม่เป็น 0.

หากไม่มีการระบุ handler สำหรับ SQLEXCEPTION และเกิดข้อผิดพลาดขึ้นซึ่งไม่ได้รับการจัดการในอีก handler หนึ่ง, การใช้ข้อความผสมจะถูกยกเลิกและข้อผิดพลาด จะถูกส่งคืนใน SQLCA. เช่นเดียวกับตัวบ่งชี้, SQLCA จะถูกส่งคืนจาก SQL โพรซีเจอร์เสมอ.

 - ประกอบด้วย OPEN, FETCH, และ CLOSE ของเคอร์เซอร์ C1. หากไม่มีการระบุ CLOSE ของเคอร์เซอร์, เคอร์เซอร์ นั้นจะถูกปิดในตอนท้ายของคำสั่งผสม เนื่องจากไม่มีการระบุ SET RESULT SETS ในคำสั่ง CREATE PROCEDURE.
 - ประกอบด้วยข้อความ WHILE ซึ่งจะวนซ้ำจนกว่าเรีกคอร์ดล่าสุดจะถูกดึงข้อมูลออก. สำหรับแต่ละแถวที่ถูกเรียกออกมา, TOTAL_SALARY จะเพิ่มขึ้นและ, หากโบนัสของพนักงานมากกว่า 0, BONUS_CNT ก็เพิ่มขึ้นเช่นกัน.
 - ส่งคืน DEPT_SALARY และ DEPT_BONUS_CNT เป็นเอาต์พุตพารามิเตอร์.

คำสั่งผสมสามารถทำเป็นแบบ atomic เพื่อที่หากเกิดข้อผิดพลาดแบบไม่คาดคิดขึ้น, คำสั่งที่อยู่ภายในคำสั่งแบบ atomic จะสามารถย้อนกลับได้. คำสั่งผสมแบบ atomic ถูกนำมาใช้ด้วยการใช้ SAVEPOINTS. หากคำสั่งผสมสำเร็จ, transaction จะถูก commit.

ใช้ตัวอย่างต่อไปนี้เป็นอินพุตหมายเลขแผนก. ซึ่งเป็นการรับประกันว่าตาราง EMPLOYEE_BONUS ยังคงมีอยู่, และใส่ชื่อของพนักงานทั้งหมดในแผนกที่ได้โบนัส. โพรซีเจอร์จะส่งคืนการนับโดยรวมของพนักงานทั้งหมดที่ได้โบนัส.

```

CREATE PROCEDURE CREATE_BONUS_TABLE
    (IN DEPT_NUMBER CHAR(3),
    INOUT CNT INT)

```

```

LANGUAGE SQL MODIFIES SQL DATA
CS1: BEGIN ATOMIC
DECLARE NAME VARCHAR(30) DEFAULT NULL;
DECLARE CONTINUE HANDLER FOR SQLSTATE '42710'
    SELECT COUNT(*) INTO CNT
    FROM DATALIB.EMPLOYEE_BONUS;
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'
    SET CNT = CNT - 1;
DECLARE UNDO HANDLER FOR SQLEXCEPTION
    SET CNT = NULL;
IF DEPT_NUMBER IS NOT NULL THEN
    CREATE TABLE DATALIB.EMPLOYEE_BONUS
        (FULLNAME VARCHAR(30),
         BONUS DECIMAL(10,2),
         PRIMARY KEY (FULLNAME));
FOR_1:FOR V1 AS C1 CURSOR FOR
    SELECT FIRSTNME, MIDINIT, LASTNAME, BONUS
    FROM CORPDATA.EMPLOYEE
    WHERE WORKDEPT = CREATE_BONUS_TABLE.DEPT_NUMBER
DO
    IF BONUS > 0 THEN
        SET NAME = FIRSTNME CONCAT ' ' CONCAT
            MIDINIT CONCAT ' ' CONCAT LASTNAME;
        INSERT INTO DATALIB.EMPLOYEE_BONUS
            VALUES(CS1.NAME, FOR_1.BONUS);
        SET CNT = CNT + 1;
    END IF;
END FOR FOR_1;
END IF;
END CS1

```

ข้อความ CREATE PROCEDURE:

- ตั้งชื่อโพรซีเจอร์เป็น CREATE_BONUS_TABLE.
- กำหนดพารามิเตอร์ DEPT_NUMBER ซึ่งเป็นอินพุตพารามิเตอร์และจัดอยู่ในประเภทข้อมูลอักขระ ความยาว 3 ตัวอักษร และพารามิเตอร์ CNT ซึ่งเป็นพารามิเตอร์อินพุต/เอาต์พุต และเป็นประเภทข้อมูลจำนวนเต็ม.
- แสดงว่าโพรซีเจอร์นี้คือ SQL โพรซีเจอร์ซึ่งแก้ไขข้อมูล SQL
- กำหนดโครงสร้างโพรซีเจอร์.
 - ประกาศ SQL variable NAME ให้เป็นอักขระที่เปลี่ยนแปลงได้.
 - ประกาศ handler แบบต่อเนื่องสำหรับ SQLSTATE 42710, มีตารางอยู่แล้ว. หากมีตาราง EMPLOYEE_BONUS อยู่แล้ว, handler จะถูกเรียกใช้งานและเรียกจำนวนเรกคอร์ดในตารางออกมา. SQLCODE และ SQLSTATE ถูกรีเซ็ต เป็น 0 และมีการประมวลผลอย่างต่อเนื่องด้วยข้อความ FOR.
 - ประกาศ handler แบบต่อเนื่องสำหรับ SQLSTATE 23505, ทำซ้ำคีย์. ถ้าโพรซีเจอร์พยายามใส่ชื่อซึ่งมีอยู่ในตารางอยู่ แล้ว, handler จะถูกเรียกใช้งานและไปลดส่วน CNT ลง. มีการประมวลผลอย่างต่อเนื่องบนข้อความ SET ซึ่งอยู่หลังข้อความ INSERT.
 - ประกาศ UNDO handler สำหรับ SQLEXCEPTION. ถ้าถูกเรียกใช้งาน, ข้อความก่อนหน้านี้จะย้อนกลับ, CNT จะถูกตั้งเป็น 0, และมีการประมวลผลต่อหลังข้อความผสม. ในกรณีนี้, เนื่องจากไม่มีข้อความตามหลังข้อความผสม, โพรซีเจอร์จึงส่งคืน.

- ใช้ข้อความ FOR เพื่อประกาศเคอร์เซอร์ C1 ให้อ่านเรกคอร์ดจากตาราง EMPLOYEE. ภายในข้อความ FOR, ชื่อคอลัมน์จากลิสต์ที่เลือกจะถูกใช้เป็นตัวแปร SQL ซึ่งประกอบด้วยข้อมูลจากแถวที่ถูกดึงข้อมูลออก. สำหรับแต่ละแถว, ข้อมูลจากคอลัมน์ FIRSTNAME, MIDINIT, และ LASTNAME จะถูกเชื่อมต่อเข้าด้วยกัน ด้วยที่ว่างในระหว่างและผลลัพธ์จะถูกใส่ไว้ใน SQL variable NAME. SQL variables NAME และ BONUS จะถูกใส่ไว้ในตาราง EMPLOYEE_BONUS. เนื่องจากต้องรู้ประเภทข้อมูลของไอเท็มลิสต์ที่เลือก เมื่อมีการสร้างโพรซีเจอร์, ตารางที่ระบุในคำสั่ง FOR จะต้องมียู่เมื่อสร้างโพรซีเจอร์.

ชื่อตัวแปร SQL สามารถทำให้ถูกกฎเกณฑ์ได้ด้วยการใช้ชื่อเลเบลของข้อความ FOR หรือข้อความผสมที่ชื่อตัวแปรนั้นระบุอยู่. ในตัวอย่าง, FOR_1.BONUS หมายถึงตัวแปร SQL ที่มีค่าของคอลัมน์ BONUS สำหรับแต่ละแถวที่เลือก. CS1.NAME คือตัวแปร NAME ซึ่งกำหนดไว้ในข้อความผสมขึ้นต้นด้วยเลเบล CS1. นอกจากนี้ชื่อพารามิเตอร์สามารถทำให้ถูกกฎเกณฑ์ได้ด้วยการใช้ชื่อโพรซีเจอร์. CREATE_BONUS_TABLE.DEPT_NUMBER คือพารามิเตอร์ DEPT_NUMBER สำหรับโพรซีเจอร์ CREATE_BONUS_TABLE. หากมีการใช้ชื่อตัวแปร SQL ที่ไม่ถูกกฎเกณฑ์ในข้อความ SQL โดยที่มีการอนุญาตใช้ชื่อคอลัมน์เช่นกัน, และชื่อตัวแปรเหมือนกับชื่อคอลัมน์, ชื่อนั้นก็จะถูกใช้เพื่ออ้างถึงคอลัมน์.

คุณสามารถใช้ SQL แบบไดนามิกใน SQL โพรซีเจอร์. ตัวอย่างต่อไปเป็นการสร้างตาราง ซึ่งประกอบด้วยพนักงานทั้งหมดในแผนกเฉพาะ. หมายเลขแผนกจะถูกส่งต่อเป็นอินพุตไปยังโพรซีเจอร์ และถูกเชื่อมต่อเข้าด้วยกันกับชื่อตาราง.

```
CREATE PROCEDURE CREATE_DEPT_TABLE (IN P_DEPT CHAR(3))
LANGUAGE SQL
BEGIN
  DECLARE STMT CHAR(1000);
  DECLARE MESSAGE CHAR(20);
  DECLARE TABLE_NAME CHAR(30);
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    SET MESSAGE = 'ok';
  SET TABLE_NAME = 'CORPDATA.DEPT_' CONCAT P_DEPT CONCAT '_T';
  SET STMT = 'DROP TABLE ' CONCAT TABLE_NAME;
  PREPARE S1 FROM STMT;
  EXECUTE S1;
  SET STMT = 'CREATE TABLE ' CONCAT TABLE_NAME CONCAT
    '( EMPNO CHAR(6) NOT NULL,
      FIRSTNAME VARCHAR(12) NOT NULL,
      MIDINIT CHAR(1) NOT NULL,
      LASTNAME CHAR(15) NOT NULL,
      SALARY DECIMAL(9,2))';
  PREPARE S2 FROM STMT;
  EXECUTE S2;
  SET STMT = 'INSERT INTO ' CONCAT TABLE_NAME CONCAT
    'SELECT EMPNO, FIRSTNAME, MIDINIT, LASTNAME, SALARY
    FROM CORPDATA.EMPLOYEE
    WHERE WORKDEPT = ?';
  PREPARE S3 FROM STMT;
  EXECUTE S3 USING P_DEPT;
END
```

ข้อความ CREATE PROCEDURE:

- ตั้งชื่อโพรซีเจอร์เป็น CREATE_DEPT_TABLE
- กำหนดพารามิเตอร์ P_DEPT ซึ่งเป็นอินพุตพารามิเตอร์และจัดอยู่ในประเภทข้อมูล อักขระความยาว 3 ตัวอักษร.
- แสดงว่าโพรซีเจอร์นี้คือ SQL โพรซีเจอร์.

- กำหนดโครงสร้างโปรแกรมเมอร์.
 - ประกาศ SQL variable STMT และ SQL variable TABLE_NAME เป็นอักขระ.
 - ประกาศ CONTINUE handler. โปรแกรมเมอร์พยายามที่จะ DROP ตารางในกรณีที่มีอยู่แล้ว. ถ้าไม่มีตารางอยู่, EXECUTE แรกจะล้มเหลว. ด้วยการใช้ handler, การประมวลผลจะดำเนินต่อไป.
 - ตั้งตัวแปร TABLE_NAME ให้เป็น 'DEPT_' ตามด้วยอักขระที่ถูกส่งผ่านมาในพารามิเตอร์ P_DEPT, ตามด้วย '_T'.
 - ตั้งตัวแปร STMT ให้เป็นข้อความ DROP, จากนั้นให้เตรียมและเรียกใช้งานข้อความ.
 - เซ็ตตัวแปร STMT ให้เป็นข้อความ CREATE, จากนั้นให้เตรียมและเรียกใช้งานข้อความ.
 - ตั้งตัวแปร STMT ให้เป็นข้อความ INSERT, จากนั้นให้เตรียมและเรียกใช้งานข้อความ. มีการระบุเครื่องหมายพารามิเตอร์ใน clause. เมื่อเรียกใช้งานคำสั่ง, ตัวแปร P_DEPT จะถูกส่งต่อไปบน USING clause.

หากมีการเรียกโปรแกรมเมอร์ผ่านค่า 'D21' สำหรับแผนก, ตาราง DEPT_D21_T ก็จะถูกสร้างขึ้นและตารางจะถูก initialize ด้วยพนักงานทั้งหมดในแผนก 'D21'.

การเรียกโปรแกรมเมอร์ที่เก็บไว้

คำสั่ง SQL CALL เรียกโปรแกรมเมอร์ที่เก็บไว้.

ที่ข้อความ CALL, จะมีการระบุชื่อของโปรแกรมเมอร์ที่เก็บไว้และอาร์กิวเมนต์ใดๆ. อาร์กิวเมนต์อาจเป็นจำนวนคงที่, เรจิสเตอร์พิเศษ, หรือตัวแปรโฮสต์. โปรแกรมเมอร์ที่เก็บไว้ภายนอกที่ระบุในข้อความ CALL ไม่จำเป็นต้องมีข้อความ CREATE PROCEDURE ที่ตรงกัน. โปรแกรมที่สร้างขึ้นโดย SQL โปรแกรมเมอร์จะสามารถเรียกได้ด้วยการเรียกใช้งาน ชื่อโปรแกรมเมอร์ที่ระบุบนข้อความ CREATE PROCEDURE เท่านั้น.

แม้ว่าโปรแกรมเมอร์จะเป็นอ็อบเจกต์โปรแกรมระบบ, การใช้คำสั่ง CALL ในชุดคำสั่ง CL เพื่อเรียกโปรแกรมเมอร์จะไม่สามารถใช้ได้. คำสั่ง CALL ในชุดคำสั่ง CL ไม่ได้ใช้ definition ของโปรแกรมเมอร์เพื่อแม็พพารามิเตอร์อินพุตและเอาต์พุต, และไม่ได้ส่งพารามิเตอร์ต่อไปยังโปรแกรมโดยใช้รูปแบบพารามิเตอร์ของโปรแกรมเมอร์.

มีข้อความ CALL อยู่สามประเภทที่จำเป็นต้องกล่าวถึงเนื่องจาก DB2 SQL สำหรับ iSeries มีกฎที่แตกต่างกันสำหรับแต่ละประเภท. ได้แก่:

- คำสั่ง CALL แบบ dynamic หรือที่ใส่อยู่ซึ่งมี definition ของโปรแกรมเมอร์อยู่
- คำสั่ง CALL ที่ใส่อยู่โดยที่ไม่มี definition ของโปรแกรมเมอร์อยู่
- คำสั่ง CALL แบบไดนามิกโดยที่ไม่มี CREATE PROCEDURE อยู่

หมายเหตุ:

Dynamic ในที่นี้หมายถึง:

- คำสั่ง CALL ที่ถูกเตรียม และเรียกใช้งานแบบ dynamic.
- คำสั่ง CALL ที่มีอยู่ในสถานะแวดล้อมแบบโต้ตอบ (ตัวอย่างเช่น, ผ่าน STRSQL หรือ Query Manager).
- คำสั่ง CALL ถูกเรียกใช้งานในคำสั่ง EXECUTE IMMEDIATE.

การใช้คำสั่ง CALL โดยที่มี definition ของโปรแกรมเมอร์อยู่:

คำสั่ง CALL ประเภทนี้จะอ่านข้อมูลทั้งหมดเกี่ยวกับโปรแกรมเมอร์ และแอ็ททริบิวต์อาร์กิวเมนต์จาก definition ของแคตตาล็อก CREATE PROCEDURE.

ตัวอย่าง PL/I ต่อไปนี้จะแสดงคำสั่ง CALL ซึ่งตรงกับคำสั่ง CREATE PROCEDURE ที่ได้แสดงไว้.

```
DCL HV1 CHAR(10);
DCL IND1 FIXED BIN(15);
:
EXEC SQL CREATE P1 PROCEDURE
      (INOUT PARM1 CHAR(10))
      EXTERNAL NAME MYLIB.PROC1
      LANGUAGE C
      GENERAL WITH NULLS;
:
EXEC SQL CALL P1 (:HV1 :IND1);
:
```

เมื่อมีการเรียกคำสั่ง CALL นี้ขึ้นมา, จะมีการเรียกไปยังโปรแกรม MYLIB/PROC1 และ ส่งต่อสองอาร์กิวเมนต์. เนื่องจากภาษาของโปรแกรมคือ ILE C, อาร์กิวเมนต์แรกจึงเป็นสตริงอักขระความยาว 11 ตัวอักษรที่จบด้วย C NUL ซึ่งมีเนื้อหาของตัวแปรโฮสต์ HV1 อยู่. โปรดสังเกตว่าในการเรียกไปยังโปรแกรม ILE C, DB2 SQL สำหรับ iSeries มีการเพิ่มอักขระหนึ่งตัวให้กับการประกาศพารามิเตอร์ถ้าพารามิเตอร์นั้นถูกประกาศให้เป็นอักขระ, กราฟิก, วันที่, เวลา, หรือตัวแปร timestamp. อาร์กิวเมนต์ที่สองคืออะเรย์ตัวบ่งชี้. ในกรณีนี้, พารามิเตอร์นี้เป็นจำนวนเต็มแบบสั้นหนึ่งจำนวน เนื่องจากมีเพียงหนึ่งพารามิเตอร์เท่านั้นในคำสั่ง CREATE PROCEDURE. อาร์กิวเมนต์นี้ประกอบด้วยเนื้อหาของตัวแปรตัวบ่งชี้ IND1 บน entry ไปยังโปรแกรม.

เนื่องจากพารามิเตอร์แรกถูกประกาศให้เป็น INOUT, SQL จึงอัปเดตตัวแปรโฮสต์ HV1 และตัวแปรตัวบ่งชี้ IND1 ด้วยค่าที่ส่งคืนมาจาก MYLIB.PROC1 ก่อนที่จะส่งคืนกลับไปยังโปรแกรมผู้ใช้.

หมายเหตุ:

1. ชื่อโปรแกรมที่ระบุในคำสั่ง CREATE PROCEDURE และ CALL จะต้องตรงกัน อย่างแท้จริงตามลำดับ เพื่อให้มีการลิงก์ระหว่างทั้งสองชื่อระหว่างที่ SQL คอมไพล์โปรแกรมล่วงหน้า.
2. สำหรับคำสั่ง CALL ที่ใส่อยู่โดยที่มีคำสั่งทั้ง CREATE PROCEDURE และ DECLARE PROCEDURE อยู่, คำสั่ง DECLARE PROCEDURE จะถูกนำมาใช้.

การใช้คำสั่ง CALL แบบฝังโดยไม่มี definition ของโปรแกรมอยู่:

คำสั่ง CALL แบบ static ที่ไม่มีคำสั่ง CREATE PROCEDURE ที่ตรงกันถูกประมวลผลด้วยกฎต่อไปนี้.

- อาร์กิวเมนต์ตัวแปรโฮสต์ทั้งหมดจะถูกปฏิบัติในฐานะพารามิเตอร์ประเภท INOUT.
- ประเภท CALL คือ GENERAL (ไม่มีการส่งต่ออาร์กิวเมนต์ตัวบ่งชี้).
- มีการกำหนดโปรแกรมที่จะเรียกใช้งานโดยอิงจากชื่อโปรแกรมที่ระบุใน CALL, และ, หากจำเป็น, ในห้องจากหลักการตั้งชื่อ.
- ภาษาของโปรแกรมที่จะเรียกใช้งานถูกกำหนดโดยอิงจากข้อมูลที่เรียกออกมาจาก ระบบเกี่ยวกับโปรแกรม.

ตัวอย่าง: คำสั่ง CALL แบบฝังที่ไม่มี definition ของโปรแกรมอยู่

ต่อไปนี้เป็นตัวอย่าง PL/I ของคำสั่ง CALL แบบฝังที่ไม่มี definition ของโปรแกรมอยู่:

```
DCL HV2 CHAR(10);
:
EXEC SQL CALL P2 (:HV2);
:
```

เมื่อมีการเรียกคำสั่ง CALL, DB2 SQL สำหรับ iSeries พยายามที่จะค้นหาโปรแกรมโดยยึดตามหลักการตั้งชื่อ SQL มาตรฐาน. สำหรับตัวอย่างข้างบน, ให้ตั้งสมมุติฐานว่าชื่อการตั้งชื่อ *SYS (การตั้งชื่อระบบ) จะถูกใช้และไม่มีการระบุพารามิเตอร์ DFTRDBCOL บนคำสั่ง CRTSQLPLI. ในกรณีนี้, รายชื่อไลบรารีจะค้นหาโปรแกรมที่ชื่อ P2. เนื่องจากประเภทการเรียกคือ GENERAL, จึงไม่มีการส่งอักขระเพิ่มเติมไปยังโปรแกรมสำหรับตัวแปรตัวบ่งชี้.

หมายเหตุ: หากมีการระบุตัวแปรตัวบ่งชี้บนคำสั่ง CALL และค่าตัวแปรดังกล่าวน้อยกว่าศูนย์เมื่อเรียกใช้งานคำสั่ง CALL, จะเกิดข้อผิดพลาดขึ้นเนื่องจากไม่มีทางที่จะส่งตัวบ่งชี้ผ่านไปยังโปรแกรมเมอร์ได้.

สมมติว่า พบโปรแกรม P2 ในรายชื่อไลบรารี, เนื้อหาของตัวแปรโฮสต์ HV2 จะถูกส่งต่อไปยังโปรแกรมบน CALL และอักขระที่ส่งคืนจาก P2 จะถูกแม็กลับไปยังตัวแปรโฮสต์ หลังจากที่ P2 ทำงานเสร็จสมบูรณ์แล้ว.

สำหรับจำนวนคงที่ที่เป็นตัวเลขที่ถูกส่งไปยังคำสั่ง CALL, จะมีการใช้กฎต่อไปนี้:

- จำนวนคงที่ที่เป็นจำนวนเต็มทั้งหมดจะถูกส่งผ่านเป็นจำนวนเต็มแบบ fullword binary.
- จำนวนคงที่ที่เป็นทศนิยมทั้งหมดจะถูกส่งผ่านเป็นค่า packed decimal. จำนวนเลขโดดและมาตราส่วน จะถูกกำหนดโดยอิงจากค่าคงที่. ตัวอย่างเช่น, ค่า 123.45 จะถูกส่งผ่านเป็น packed decimal(5,2). เช่นเดียวกัน, ค่า 001.01 ก็จะถูกส่งผ่านด้วยจำนวนเลขโดด และมาตราส่วน 5 และ 2 ตามลำดับ.
- จำนวนคงที่ที่ทศนิยมลอยตัวทั้งหมดจะถูกส่งผ่านเป็นทศนิยมลอยตัวที่มีความเที่ยงตรงสองเท่า.

เรจิสเตอร์พิเศษที่ระบุบนคำสั่ง CALL แบบ dynamic จะถูกส่งผ่านดังนี้:

CURRENT DATE

ส่งผ่านเป็นสตริงอักขระขนาด 10 ไบต์ในฟอร์แมต ISO.

CURRENT DEGREE

ส่งผ่านสตริงอักขระขนาด 5 ไบต์.

CURRENT TIME

ส่งผ่านเป็นสตริงอักขระขนาด 8 ไบต์ในฟอร์แมต ISO.

CURRENT TIMEZONE

ส่งผ่านตัวเลข packed decimal ด้วยจำนวนเลขโดด 6 และมาตราส่วน 0.

CURRENT TIMESTAMP

ส่งผ่านสตริงอักขระขนาด 26 ไบต์ในฟอร์แมต IBM SQL.

CURRENT SCHEMA

ส่งผ่านสตริงอักขระความยาวผันแปรขนาด 128 ไบต์.

CURRENT SERVER

ส่งผ่านเป็นสตริงอักขระความยาวผันแปรขนาด 18 ไบต์

USER ส่งผ่านเป็นสตริงอักขระความยาวผันแปรขนาด 18 ไบต์

CURRENT PATH

ส่งผ่านสตริงอักขระความยาวผันแปรขนาด 3483 ไบต์.

SESSION_USER

ส่งผ่านสตริงอักขระความยาวผันแปรขนาด 128 ไบต์.

SYSTEM_USER

ส่งผ่านสตริงอักขระความยาวผันแปรขนาด 128 ไบต์.

การใช้คำสั่ง CALL ที่ฝังด้วย SQLDA:

ใน CALL ประเภทฝัง (โดยที่ definition ของโพรซีเจอร์อาจจะมีหรือไม่มีอยู่), SQLDA อาจถูกส่งผ่านแทนที่ลิสต์ของพารามิเตอร์.

ตัวอย่าง C ต่อไปนี้อธิบายถึงสิ่งนี้. สมมติว่าโพรซีเจอร์ที่เก็บไว้คาดว่าจะมี 2 พารามิเตอร์, ประเภทแรกคือ SHORT INT และประเภทที่สองคือ CHAR ซึ่งมีความยาวอักขระ 4 ตัวอักษร.

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

```
#define SQLDA_HV_ENTRIES 2
#define SHORTINT 500
#define NUL_TERM_CHAR 460

exec sql include sqlca;
exec sql include sqlda;

...
typedef struct sqlda Sqlda;
typedef struct sqlda* Sqldap;
...
main()
{
    Sqldap dap;
    short col1;
    char col2[4];
    int bc;
    dap = (Sqldap) malloc(bc=SQLDASIZE(SQLDA_HV_ENTRIES));
    /* SQLDASIZE is a macro defined in the sqlda include */
    col1 = 431;
    strcpy(col2,"abc");
    strncpy(dap->sqldaid,"SQLDA ",8);
    dap->sqldabc = bc; /* bc set in the malloc statement above */
    dap->sqln = SQLDA_HV_ENTRIES;
    dap->sqld = SQLDA_HV_ENTRIES;
    dap->sqlvar[0].sqltype = SHORTINT;
    dap->sqlvar[0].sqlllen = 2;
    dap->sqlvar[0].sqldata = (char*) &col1;
    dap->sqlvar[0].sqlname.length = 0;
    dap->sqlvar[1].sqltype = NUL_TERM_CHAR;
    dap->sqlvar[1].sqlllen = 4;
    dap->sqlvar[1].sqldata = col2;
    ...
    EXEC SQL CALL P1 USING DESCRIPTOR :*dap;
    ...
}
```

ชื่อของโพรซีเจอร์ที่เก็บไว้อาจเก็บไว้ในตัวแปรโฮสต์ และตัวแปรโฮสต์ที่ใช้ในคำสั่ง CALL, แทนที่ชื่อโพรซีเจอร์แบบ hard-code. ตัวอย่างเช่น:


```

...
    main()
{
    char proc_name[15];
    ...
    strcpy (proc_name, "MYLIB.P3");
    ...
    EXEC SQL CALL :proc_name ...;
    ...
}

```

ในตัวอย่างข้างบน, หาก MYLIB.P3 คาดถึงพารามิเตอร์, ลิสต์พารามิเตอร์หรือ SQLDA ซึ่งถูกส่งผ่านด้วย USING DESCRIPTOR อาจถูกใช้งานอยู่, ตามที่แสดงไว้ในตัวอย่างก่อนหน้านี้.

เมื่อตัวแปรโฮสต์ที่มีชื่อโพรซีเจอร์ถูกใช้ในคำสั่ง CALL และมี CREATE PROCEDURE catalog definition อยู่, ตัวแปรโฮสต์นั้นจะถูกใช้. ชื่อโพรซีเจอร์ไม่สามารถระบุเป็นเครื่องหมายพารามิเตอร์ได้.

การใช้คำสั่ง CALL แบบ dynamic โดยที่ไม่มี CREATE PROCEDURE อยู่:

กฎต่อไปนี้จะเกี่ยวข้องกับการประมวลผลคำสั่ง CALL แบบ dynamic เมื่อไม่มี definition ของ CREATE PROCEDURE อยู่.

- อากิวเมนต์ทั้งหมดจะถูกจัดให้เป็นพารามิเตอร์ประเภท IN.
- ประเภท CALL คือ GENERAL (ไม่มีการส่งต่ออากิวเมนต์ตัวบ่งชี้).
- มีการกำหนดโปรแกรมที่จะเรียกขึ้นมาโดยอิงจากชื่อโพรซีเจอร์ที่ระบุใน CALL และหลักการตั้งชื่อ.
- ภาษาของโปรแกรมที่จะเรียกใช้งานถูกกำหนดโดยอิงจากข้อมูลที่เรียกออกมาจาก ระบบเกี่ยวกับโปรแกรม.

ตัวอย่าง: คำสั่ง CALL แบบ dynamic โดยที่ไม่มี CREATE PROCEDURE อยู่

ต่อไปนี้เป็นตัวอย่าง C ของคำสั่ง CALL แบบ dynamic:

```

char hv3[10],string[100];
:
strcpy(string,"CALL MYLIB.P3 ('P3 TEST')");
EXEC SQL EXECUTE IMMEDIATE :string;
:

```

ตัวอย่างนี้จะแสดงคำสั่ง CALL แบบ dynamic ที่ถูกเรียกใช้งานผ่านคำสั่ง EXECUTE IMMEDIATE. มีการเรียก ไปยังโปรแกรม MYLIB.P3 โดยมีการส่งต่อหนึ่งพารามิเตอร์ในฐานะที่เป็นตัวแปรอักขระซึ่งมี 'P3 TEST' อยู่.

เมื่อเรียกใช้งานคำสั่ง CALL และผ่านจำนวนคงที่แล้ว, ตั้งตัวอย่างก่อนหน้านี้, จะต้องจดจำความยาวของอากิวเมนต์ที่คาดไว้ในโปรแกรม. ถ้าโปรแกรม MYLIB.P3 คาดหวังอากิวเมนต์ที่มีความยาวอักขระเพียง 5 ตัวอักษร, อักขระ 2 ตัวสุดท้ายของค่าคงที่ที่ระบุไว้ในตัวอย่าง จะต้องเสียให้กับโปรแกรม.

หมายเหตุ: ด้วยสาเหตุนี้, จึงเป็นการปลอดภัยกว่าเสมอที่จะใช้ตัวแปรโฮสต์บนคำสั่ง CALL เพื่อที่แอตทริบิวต์ของโพรซีเจอร์จะได้ตรงกันแน่นอน และเพื่อที่อักขระจะไม่หายไป. สำหรับ SQL แบบ dynamic, สามารถระบุตัวแปรโฮสต์สำหรับอากิวเมนต์คำสั่ง CALL ได้หากคำสั่ง PREPARE และ EXECUTE ถูกนำมาใช้เพื่อประมวลผล SQL ดังกล่าว.

ตัวอย่างข้อความ CALL:

ตัวอย่างเหล่านี้แสดงถึงวิธีการที่อากิวเมนต์ของข้อความ CALL ถูกส่งผ่านไปยัง โพรซีเจอร์สำหรับหลายภาษา. และยังแสดงวิธีการรับอากิวเมนต์เข้าสู่ตัวแปรโลคัลในโพรซีเจอร์.

ตัวอย่างแรกแสดงการเรียกโปรแกรม ILE C ที่ใช้ CREATE PROCEDURE definition เพื่อเรียกโพรซีเจอร์ P1 และ P2. โพรซีเจอร์ P1 ถูกบันทึกลงใน C และมี 10 พารามิเตอร์. โพรซีเจอร์ P2 ถูกบันทึกใน PL/I และมี 10 พารามิเตอร์.

สมมติว่าทั้งสองโพรซีเจอร์ถูกกำหนดดังนี้:

```
EXEC SQL CREATE PROCEDURE P1 (INOUT PARM1 CHAR(10),
                              INOUT PARM2 INTEGER,
                              INOUT PARM3 SMALLINT,
                              INOUT PARM4 FLOAT(22),
                              INOUT PARM5 FLOAT(53),
                              INOUT PARM6 DECIMAL(10,5),
                              INOUT PARM7 VARCHAR(10),
                              INOUT PARM8 DATE,
                              INOUT PARM9 TIME,
                              INOUT PARM10 TIMESTAMP)
      EXTERNAL NAME TEST12.CALLPROC2
      LANGUAGE C GENERAL WITH NULLS
```

```
EXEC SQL CREATE PROCEDURE P2 (INOUT PARM1 CHAR(10),
                              INOUT PARM2 INTEGER,
                              INOUT PARM3 SMALLINT,
                              INOUT PARM4 FLOAT(22),
                              INOUT PARM5 FLOAT(53),
                              INOUT PARM6 DECIMAL(10,5),
                              INOUT PARM7 VARCHAR(10),
                              INOUT PARM8 DATE,
                              INOUT PARM9 TIME,
                              INOUT PARM10 TIMESTAMP)
      EXTERNAL NAME TEST12.CALLPROC
      LANGUAGE PLI GENERAL WITH NULLS
```

ตัวอย่าง 1: ILE C และโพรซีเจอร์ PL/I ที่ถูกเรียกจากแอฟพลิเคชัน ILE C:

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

ตัวอย่างของ CREATE PROCEDURE และ CALL

```
/******  
/****** START OF SQL C Application *****/  
  
#include <stdio.h>  
#include <string.h>  
#include <decimal.h>  
main()  
{  
    EXEC SQL INCLUDE SQLCA;  
    char PARM1[10];  
    signed long int PARM2;  
    signed short int PARM3;  
    float PARM4;
```

```

double PARM5;
decimal(10,5) PARM6;
struct { signed short int parm7l;
        char parm7c[10];
        } PARM7;
char PARM8[10];      /* FOR DATE */
char PARM9[8];      /* FOR TIME */
char PARM10[26];    /* FOR TIMESTAMP */
/*****
/* Initialize variables for the call to the procedures */
*****/
strcpy(PARM1,"PARI");
PARM2 = 7000;
PARM3 = -1;
PARM4 = 1.2;
PARM5 = 1.0;
PARM6 = 10.555;
PARM7.parm7l = 5;
strcpy(PARM7.parm7c,"PARI");
strncpy(PARM8,"1994-12-31",10);      /* FOR DATE      */
strncpy(PARM9,"12.00.00",8);        /* FOR TIME      */
strncpy(PARM10,"1994-12-31-12.00.00.000000",26);

/* FOR TIMESTAMP */

/*****
/* Call the C procedure      */
/*      */
/*      */
*****/
EXEC SQL CALL P1 (:PARM1, :PARM2, :PARM3,
                 :PARM4, :PARM5, :PARM6,
                 :PARM7, :PARM8, :PARM9,
                 :PARM10 );

if (strncmp(SQLSTATE,"00000",5))
{
    /* Handle error or warning returned on CALL statement */
}

/* Process return values from the CALL.      */
:

/*****
/* Call the PLI procedure      */
/*      */
/*      */
*****/
/* Reset the host variables before making the CALL      */
/*      */
:
EXEC SQL CALL P2 (:PARM1, :PARM2, :PARM3,
                 :PARM4, :PARM5, :PARM6,
                 :PARM7, :PARM8, :PARM9,
                 :PARM10 );

if (strncmp(SQLSTATE,"00000",5))
{

```

```

    /* Handle error or warning returned on CALL statement */
}
/* Process return values from the CALL.          */
:
}

/***** END OF C APPLICATION *****/
/*****

```

ตัวอย่างโปรแกรม P1

```

/***** START OF C PROCEDURE P1 *****/
/*      PROGRAM TEST12/CALLPROC2          */
/*****

#include <stdio.h>
#include <string.h>
#include <decimal.h>
main(argc,argv)
    int argc;
    char *argv[];
{
    char parm1[11];
    long int parm2;
    short int parm3,i,j,*ind,ind1,ind2,ind3,ind4,ind5,ind6,ind7,
        ind8,ind9,ind10;

    float parm4;
    double parm5;
    decimal(10,5) parm6;
    char parm7[11];
    char parm8[10];
    char parm9[8];
    char parm10[26];
    /* *****/
    /* Receive the parameters into the local variables - */
    /* Character, date, time, and timestamp are passed as */
    /* NUL terminated strings - cast the argument vector to */
    /* the proper data type for each variable. Note that */
    /* the argument vector can be used directly instead of */
    /* copying the parameters into local variables - the copy */
    /* is done here just to illustrate the method.          */
    /* *****/

    /* Copy 10 byte character string into local variable */
    strcpy(parm1,argv[1]);

    /* Copy 4 byte integer into local variable          */
    parm2 = *(int *) argv[2];

    /* Copy 2 byte integer into local variable          */
    parm3 = *(short int *) argv[3];

    /* Copy floating point number into local variable */
    parm4 = *(float *) argv[4];

    /* Copy double precision number into local variable */

```

```

    parm5 = *(double *) argv[5];

        /* Copy decimal number into local variable          */
    parm6 = *(decimal(10,5) *) argv[6];

/*****
/* Copy NUL terminated string into local variable.        */
/* Note that the parameter in the CREATE PROCEDURE was   */
/* declared as varying length character. For C, varying */
/* length are passed as NUL terminated strings unless    */
/* FOR BIT DATA is specified in the CREATE PROCEDURE    */
/*****
strcpy(parm7,argv[7]);

/*****
/* Copy date into local variable.                          */
/* Note that date and time variables are always passed in */
/* ISO format so that the lengths of the strings are      */
/* known. strcpy works here just as well.                */
/*****
strncpy(parm8,argv[8],10);

/* Copy time into local variable                            */
strncpy(parm9,argv[9],8);

/*****
/* Copy timestamp into local variable.                    */
/* IBM SQL timestamp format is always passed so the length*/
/* of the string is known.                                */
/*****
strncpy(parm10,argv[10],26);

/*****
/* The indicator array is passed as an array of short    */
/* integers. There is one entry for each parameter passed */
/* on the CREATE PROCEDURE (10 for this example).        */
/* Below is one way to set each indicator into separate   */
/* variables.                                             */
/*****
    ind = (short int *) argv[11];
        ind1 = *(ind++);
        ind2 = *(ind++);
        ind3 = *(ind++);
        ind4 = *(ind++);
        ind5 = *(ind++);
        ind6 = *(ind++);
        ind7 = *(ind++);
        ind8 = *(ind++);
        ind9 = *(ind++);
        ind10 = *(ind++);
:
/* Perform any additional processing here                */
:
    return;
}
/***** END OF C PROCEDURE P1 *****/

```

ตัวอย่างโพรซีเจอร์ P2

```
/****** START OF PL/I PROCEDURE P2 *****/
/****** PROGRAM TEST12/CALLPROC *****/
/******

CALLPROC :PROC( PARM1,PARM2,PARM3,PARM4,PARM5,PARM6,PARM7,
                PARM8,PARM9,PARM10,PARM11);

DCL SYSPRINT FILE STREAM OUTPUT EXTERNAL;
OPEN FILE(SYSPRINT);
DCL PARM1 CHAR(10);
DCL PARM2 FIXED BIN(31);
DCL PARM3 FIXED BIN(15);
DCL PARM4 BIN FLOAT(22);
DCL PARM5 BIN FLOAT(53);
DCL PARM6 FIXED DEC(10,5);
DCL PARM7 CHARACTER(10) VARYING;
DCL PARM8 CHAR(10);      /* FOR DATE */
DCL PARM9 CHAR(8);      /* FOR TIME */
DCL PARM10 CHAR(26);    /* FOR TIMESTAMP */
DCL PARM11(10) FIXED BIN(15); /* Indicators */

/* PERFORM LOGIC - Variables can be set to other values for */
/* return to the calling program.                               */

:

END CALLPROC;
```

ตัวอย่างถัดไปแสดงถึงโพรซีเจอร์ REXX ที่ถูกเรียกจากโปรแกรม ILE C.

สมมติว่าโพรซีเจอร์ถูกกำหนดดังนี้:

```
EXEC SQL CREATE PROCEDURE REXXPROC
      (IN PARM1 CHARACTER(20),
       IN PARM2 INTEGER,
       IN PARM3 DECIMAL(10,5),
       IN PARM4 DOUBLE PRECISION,
       IN PARM5 VARCHAR(10),
       IN PARM6 GRAPHIC(4),
       IN PARM7 VARGRAPHIC(10),
       IN PARM8 DATE,
       IN PARM9 TIME,
       IN PARM10 TIMESTAMP)
      EXTERNAL NAME 'TEST.CALLSRC(CALLREXX)'
      LANGUAGE REXX GENERAL WITH NULLS
```

ตัวอย่างที่ 2. โพรซีเจอร์ REXX ถูกเรียกจากแอฟพลิเคชัน C:

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

ตัวอย่างโพรซีเจอร์ REXX ถูกเรียกจากแอฟพลิเคชัน C

```

/*****
/***** START OF SQL C Application *****/

#include <decimal.h>
#include <stdio.h>
#include <string.h>
#include <wchar.h>
/*-----*/
    exec sql include sqlca;
    exec sql include sqlda;
/* *****/
    /* Declare host variable for the CALL statement          */
/* *****/
char parm1[20];
    signed long int parm2;
    decimal(10,5) parm3;
    double parm4;
    struct { short dlen;
            char dat[10];
            } parm5;
wchar_t parm6[4] = { 0xC1C1, 0xC2C2, 0xC3C3, 0x0000 };
    struct { short dlen;
            wchar_t dat[10];
            } parm7 = {0x0009, 0xE2E2,0xE3E3,0xE4E4, 0xE5E5, 0xE6E6,
                      0xE7E7, 0xE8E8, 0xE9E9, 0xC1C1, 0x0000 };

char parm8[10];
char parm9[8];
char parm10[26];
    main()
{
/* *****/
    /* Call the procedure - on return from the CALL statement the */
    /* SQLCODE should be 0. If the SQLCODE is non-zero,          */
/* the procedure detected an error.                               */
/* *****/
    strcpy(parm1,"TestingREXX");
    parm2 = 12345;
    parm3 = 5.5;
    parm4 = 3e3;
    parm5.dlen = 5;
    strcpy(parm5.dat,"parm6");
    strcpy(parm8,"1994-01-01");
    strcpy(parm9,"13.01.00");
    strcpy(parm10,"1994-01-01-13.01.00.000000");

    EXEC SQL CALL REXXPROC (:parm1, :parm2,
                           :parm3,:parm4,
                           :parm5, :parm6,
                           :parm7,
                           :parm8, :parm9,
                           :parm10);

    if (strncpy(SQLSTATE,"00000",5))
{

```



```

:
/*****
/* Indicate the call was successful by exiting with a      */
/* return code of 0                                       */
*****/
exit(0)

ar1tag:
say "ar1 did not match" ar1
exit(1)
ar2tag:
say "ar2 did not match" ar2
exit(1)

:
:

/***** END OF REXX MEMBER *****/

```

การส่งคืนเซตของผลลัพธ์จากโพรซีเจอร์ที่เก็บไว้

นอกจากการส่งกลับเอาต์พุตพารามิเตอร์แล้ว, โพรซีเจอร์ที่เก็บไว้ยังมีอีกคุณลักษณะหนึ่งซึ่งสามารถส่งกลับตารางพร้อมด้วยเคอร์เซอร์ที่ถูกเปิดไว้ในโพรซีเจอร์ที่เก็บไว้ (เรียกว่าเซตของผลลัพธ์) ไปยังแอฟพลิเคชันที่ส่งคำสั่ง CALL. จากนั้น แอฟพลิเคชันนั้นจะใช้การร้องขอถึงข้อมูลเพื่ออ่านแถวของเคอร์เซอร์ของเซตของผลลัพธ์.

เซตของผลลัพธ์ถูกส่งกลับตามแอตทริบิวต์ความสามารถในการส่งกลับของเคอร์เซอร์. แอตทริบิวต์ความสามารถในการส่งกลับของเคอร์เซอร์สามารถกำหนดได้ในคำสั่ง DECLARE CURSOR หรือเป็นค่าตีฟอลต์. คำสั่ง SET RESULT SETS ยังอนุญาตให้กำหนดว่า เซตของผลลัพธ์ควรถูกส่งกลับไปทีใด. โดยตีฟอลต์, เคอร์เซอร์ซึ่งถูกเปิดไว้ในโพรซีเจอร์ที่เก็บไว้ถูกกำหนดให้มีแอตทริบิวต์ของ RETURN TO CALLER. การส่งกลับเซตของผลลัพธ์พร้อมด้วยเคอร์เซอร์ไปยังแอฟพลิเคชันซึ่งอยู่ชั้นนอกสุดของสแต็กการเรียก, แอตทริบิวต์ความสามารถในการส่งกลับของ RETURN TO CLIENT ถูกกำหนดไว้ในข้อความ DECLARE CURSOR. ซึ่งจะอนุญาตให้โพรซีเจอร์ชั้นในสามารถส่งกลับเซตของผลลัพธ์เมื่อแอฟพลิเคชันเรียกโพรซีเจอร์แบบซ้ำซ้อน. สำหรับเคอร์เซอร์ของเซตของผลลัพธ์ที่ไม่เคยถูกส่งกลับไปยังผู้เรียกหรือไคลเอ็นต์, แอตทริบิวต์ความสามารถในการส่งกลับของ WITHOUT RETURN ถูกระบุไว้บนข้อความ DECLARE CURSOR.

มีหลายกรณีการเปิดเคอร์เซอร์ในโพรซีเจอร์ที่เก็บไว้และการรับเซตของผลลัพธ์กลับมีข้อดีหลายข้อเหนือการเปิดเคอร์เซอร์โดยตรงจากแอฟพลิเคชัน. ตัวอย่าง, ความปลอดภัยต่อตารางที่เคอร์เซอร์อ้างอิงสามารถรับประกันการใช้งานได้จากโพรซีเจอร์ที่เก็บไว้ ดังนั้นจึงไม่มีความจำเป็นต้องให้สิทธิ์การใช้งานตารางต่างๆกับผู้ใช้แอฟพลิเคชันโดยตรง. เพียงแค่, มีการให้สิทธิ์ในการใช้งานที่จะเรียกโพรซีเจอร์ที่เก็บไว้กับผู้ใช้, ซึ่งได้รับการคอมไพล์พร้อมกับสิทธิ์ในการใช้งานที่เพียงพอในการเข้าถึงตาราง. ข้อดีของการเปิดเคอร์เซอร์ในโพรซีเจอร์ที่เก็บไว้ก็อีกอันหนึ่งคือการที่การเรียกไปยังโพรซีเจอร์ที่เก็บไว้ครั้งหนึ่งสามารถรับค่าเซตผลลัพธ์กลับได้หลายชุด, ซึ่งทำให้มีประสิทธิภาพมากกว่าการเปิดเคอร์เซอร์แยกกันจากแอฟพลิเคชันที่เรียก. ยิ่งไปกว่านั้น, การเรียกไปที่โพรซีเจอร์ที่เก็บไว้อันเดียวกันในแต่ละครั้งอาจได้เซตผลลัพธ์ที่แตกต่างกัน, ทำให้เกิดความคล่องตัวของแอฟพลิเคชัน.

อินเตอร์เฟซที่สามารถใช้งานร่วมกับเซตผลลัพธ์ของโพรซีเจอร์ที่เก็บไว้ได้แก่ JDBC, CLI, และ ODBC. ตัวอย่างสำหรับวิธีการใช้อินเตอร์เฟซ API เหล่านี้สำหรับการทำงานกับเซตของผลลัพธ์ของโพรซีเจอร์ที่เก็บไว้ซึ่งอยู่ในตัวอย่างต่อไปนี้.

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

ตัวอย่าง 1: การเรียกโพรซีเจอร์ที่เก็บไว้ ซึ่งส่งกลับชุดผลลัพธ์หนึ่งชุด:

ตัวอย่างนี้แสดงการใช้ API ซึ่งแอฟพลิเคชัน ODBC ใช้ในการเรียกโพรซีเจอร์ที่เก็บไว้ที่ส่งกลับหนึ่งชุดของเซตผลลัพธ์.

สังเกตว่าไม่มีการระบุความสามารถในการส่งกลับอย่างชัดเจนในข้อความ DECLARE CURSOR. เมื่อมีเพียงหนึ่งโพรซีเจอร์ที่เก็บไว้ในสแต็กการเรียก, แอ็ททริบิวต์ความสามารถในการส่งกลับของ RETURN TO CALLER และของ RETURN TO CLIENT จะทำให้เซตผลลัพธ์พร้อมสำหรับผู้ที่ใช้แอฟพลิเคชัน. โปรดสังเกตด้วยว่าโพรซีเจอร์ที่เก็บไว้ถูกกำหนดด้วย DYNAMIC RESULT SETS clause. สำหรับโพรซีเจอร์ SQL, clause นี้จำเป็นต้องมีถ้าต้องการให้โพรซีเจอร์ที่เก็บไว้ส่งกลับเซตผลลัพธ์.

การนิยามโพรซีเจอร์ที่เก็บไว้:

```
PROCEDURE prod.reset
```

```
CREATE PROCEDURE prod.reset () LANGUAGE SQL
  DYNAMIC RESULT SETS 1
  BEGIN
  DECLARE C1 CURSOR FOR SELECT * FROM QIWS.QCUSTCDT;
  OPEN C1;
  RETURN;
  END
```

แอฟพลิเคชัน ODBC

หมายเหตุ: ตรวจจับบางตัวได้ถูกลบออก.

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

```
:
strcpy(stmt,"call prod.reset()");
rc = SQLExecDirect(hstmt,stmt,SQL_NTS);
    if (rc == SQL_SUCCESS)
{
    // CALL statement has executed successfully. Process the result set.
    // Get number of result columns for the result set.
    rc = SQLNumResultCols(hstmt, &wNum);
    if (rc == SQL_SUCCESS)
        // Get description of result columns in result set
    { rc = SQLDescribeCol(hstmt,a);
      if (rc == SQL_SUCCESS)
        :

    {
        // Bind result columns based on attributes returned
    //
    rc = SQLBindCol(hstmt,a);
    :
        // FETCH records until EOF is returned

        rc = SQLFetch(hstmt);
        while (rc == SQL_SUCCESS)
            { // process result returned on the SQLFetch
```

```

:
    rc = SQLFetch(hstmt);
}
:
}

// Close the result set cursor when done with it.
rc = SQLFreeStmt(hstmt,SQL_CLOSE);
:

```

ตัวอย่าง 2: การเรียกโพรซีเจอร์ที่เก็บไว้ซึ่งส่งกลับเซตผลลัพธ์หนึ่งชุดจากโพรซีเจอร์ที่ซ่อนอยู่:

ตัวอย่างนี้แสดงวิธีที่โพรซีเจอร์ที่เก็บไว้ที่ซ่อนอยู่สามารถเปิดและส่งกลับเซตผลลัพธ์ไปยังโพรซีเจอร์ที่อยู่นอกสุด.

การส่งกลับเซตผลลัพธ์ไปยังโพรซีเจอร์ที่อยู่นอกสุดในสภาพแวดล้อมที่มีโพรซีเจอร์ที่เก็บไว้ซ่อนอยู่, แอ็ดทริบิวต์ความสามารถในการส่งกลับ RETURN TO CLIENT ควรจะถูกใช้ในข้อความ DECLARE CURSOR หรือในข้อความ SET RESULT SETS เพื่อที่จะแสดงความต้องการที่จะส่งกลับเคอร์เซอร์ไปยังแอ็พพลิเคชันที่เรียกมาที่โพรซีเจอร์ชั้นนอกสุด. โปรดสังเกตว่าการเรียกโพรซีเจอร์ที่ซ่อนกันนี้จะส่งเซตผลลัพธ์สองชุดไปยังไคลเอ็นต์; ชุดแรก, ชุดผลลัพธ์แบบบอระเรย์, และชุดที่สองคือชุดผลลัพธ์แบบเคอร์เซอร์. ไคลเอ็นต์แอ็พพลิเคชันแบบ ODBC และแบบ JDBC พร้อมด้วยโพรซีเจอร์ที่เก็บไว้ถูกแสดงไว้ข้างล่างนี้.

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

การนิยามโพรซีเจอร์ที่เก็บไว้:

```

CREATE PROCEDURE prod.rtnnested () LANGUAGE CL DYNAMIC RESULT SET 2
    EXTERNAL NAME prod.rtnnested GENERAL

CREATE PROCEDURE prod.rtnclient () LANGUAGE RPGLE
    EXTERNAL NAME prod.rtnclient GENERAL

```

ซอร์ส CL สำหรับโพรซีเจอร์ที่เก็บไว้ prod.rtnnested

```

PGM
    CALL      PGM(PROD/RTNCLIENT)

```

ซอร์ส ILE RPG สำหรับโพรซีเจอร์ที่เก็บไว้ prod.rtnclient

```

DRESULT          DS          OCCURS(20)
D COL1           1         16A
C   1            DO         10          X          2 0
C   X            OCCUR     RESULT
C               EVAL      COL1='array result set'
C               ENDDO
C               EVAL      X=X-1
C/EXEC SQL DECLARE C2 CURSOR WITH RETURN TO CLIENT
C+ FOR SELECT LSTNAM FROM QIWS.QCUSTCDT FOR FETCH ONLY
C/END-EXEC
C/EXEC SQL
C+ OPEN C2
C/END-EXEC
C/EXEC SQL
C+ SET RESULT SETS FOR RETURN TO CLIENT ARRAY :RESULT FOR :X ROWS,

```

```

C+  CURSOR C2
    C/END-EXEC
C      SETON                               LR
    C      RETURN

```

แอปพลิเคชัน ODBC

```

//*****
//
// Module:
//   Examples.C
//
// Purpose:
//   Perform calls to stored procedures to get back result sets.
//
// *****

#include "common.h"
#include "stdio.h"

// *****
//
// Local function prototypes.
//
// *****

SWORD  FAR PASCAL RetClient(lpSERVERINFO lpSI);
BOOL   FAR PASCAL Bind_Params(HSTMT);
BOOL   FAR PASCAL Bind_First_RS(HSTMT);
BOOL   FAR PASCAL Bind_Second_RS(HSTMT);

// *****
//
// Constant strings definitions for SQL statements used in
// the auto test.
//
// *****
//
// Declarations of variables global to the auto test.
//
// *****
#define ARRAYCOL_LEN 16
#define LSTNAM_LEN 8
char  stmt[2048];
char  buf[2000];

UDWORD rowcnt;
char  arraycol[ARRAYCOL_LEN+1];
char  lstnam[LSTNAM_LEN+1];
SDWORD cbcol1,cbcol2;

lpSERVERINFO lpSI; /* Pointer to a SERVERINFO structure. */

```

```

// *****
//
// Define the auto test name and the number of test cases
// for the current auto test. These informations will
// be returned by AutoTestName().
//
// *****

LPSTR szAutoTestName = CREATE_NAME("Result Sets Examples");
UINT iNumOfTestCases = 1;

// *****
//
// Define the structure for test case names, descriptions,
// and function names for the current auto test.
// Test case names and descriptions will be returned by
// AutoTestDesc(). Functions will be run by
// AutoTestFunc() if the bits for the corresponding test cases
// are set in the rglMask member of the SERVERINFO
// structure.
//
// *****
struct TestCase TestCasesInfo[] =
{
    "Return to Client",
    "2 result sets ",
    RetClient
};

// *****
//
// Sample return to Client:
// Return to Client result sets. Call a CL program which in turn
// calls an RPG program which returns 2 result sets. The first
// result set is an array result set and the second is a cursor
// result set.
//
// *****
SWORD FAR PASCAL RetClient(lpSERVERINFO lpSI)
{
    SWORD sRC = SUCCESS;
    RETCODE returncode;
    HENV henv;
    HDBC hdbc;
    HSTMT hstmt;

    if (FullConnect(lpSI, &henv, &hdbc, &hstmt) == FALSE)

```

```

{
    SRC = FAIL;
    goto ExitNoDisconnect;
}
// *****
// Call CL program PROD.RTNNESTED, which in turn calls RPG
// program RTNCLIENT.
// *****
strcpy(stmt,"CALL PROD.RTNNESTED()");
// *****
// Call the CL program prod.rtnnested. This program will in turn
// call the RPG program proc.rtnclient, which will open 2 result
// sets for return to this ODBC application.
// *****
returncode = SQLExecDirect(hstmt,stmt,SQL_NTS);
if (returncode != SQL_SUCCESS)
{
    vWrite(lpSI, "CALL PROD.RTNNESTED is not Successful", TRUE);
}
else
{
    vWrite(lpSI, "CALL PROC.RTNNESTED was Successful", TRUE);
}
// *****
// Bind the array result set output column. Note that the result
// sets are returned to the application in the order that they
// are specified on the SET RESULT SETS statement.
// *****
if (Bind_First_RS(hstmt) == FALSE)
{
    myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS,
                returncode, "Bind_First_RS");

    SRC = FAIL;
    goto ErrorRet;
}
else
{
    vWrite(lpSI, "Bind_First_RS Complete...", TRUE);
}
// *****
// Fetch the rows from the array result set. After the last row
// is read, a returncode of SQL_NO_DATA_FOUND will be returned to
// the application on the SQLFetch request.
// *****
returncode = SQLFetch(hstmt);
while(returncode == SQL_SUCCESS)
{
    wsprintf(stmt,"array column = %s",arraycol);
    vWrite(lpSI,stmt,TRUE);
    returncode = SQLFetch(hstmt);
}
if (returncode == SQL_NO_DATA_FOUND) ;
    else {
        myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS_WITH_INFO,
                    returncode, "SQLFetch");
    }
}

```

```

        sRC = FAIL;
        goto ErrorRet;
    }
// *****
// Get any remaining result sets from the call. The next
// result set corresponds to cursor C2 opened in the RPG
// Program.
// *****
returncode = SQLMoreResults(hstmt);
if (returncode != SQL_SUCCESS)
{
    myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS, returncode, "SQLMoreResults");
    sRC = FAIL;
    goto ErrorRet;
}
// *****
// Bind the cursor result set output column. Note that the result
// sets are returned to the application in the order that they
// are specified on the SET RESULT SETS statement.
// *****

if (Bind_Second_RS(hstmt) == FALSE)
{
    myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS,
               returncode, "Bind_Second_RS");

    sRC = FAIL;
    goto ErrorRet;
}
else
{
    vWrite(lpSI, "Bind_Second_RS Complete...", TRUE);
}
// *****
// Fetch the rows from the cursor result set. After the last row
// is read, a returncode of SQL_NO_DATA_FOUND will be returned to
// the application on the SQLFetch request.
// *****
returncode = SQLFetch(hstmt);
while(returncode == SQL_SUCCESS)
{
    wsprintf(stmt,"lstnam = %s",lstnam);
    vWrite(lpSI,stmt,TRUE);
    returncode = SQLFetch(hstmt);
}
if (returncode == SQL_NO_DATA_FOUND) ;
else {
    myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS_WITH_INFO,
               returncode, "SQLFetch");

    sRC = FAIL;
    goto ErrorRet;
}

returncode = SQLFreeStmt(hstmt,SQL_CLOSE);
if (returncode != SQL_SUCCESS)
{

```

```

        myRETCHECK(lpSI, henv, hdbc, hstmt, SQL_SUCCESS,
                    returncode, "Close statement");
    sRC = FAIL;
    goto ErrorRet;
}
else
{
    vWrite(lpSI, "Close statement...", TRUE);
}

ErrorRet:
    FullDisconnect(lpSI, henv, hdbc, hstmt);
    if (sRC == FAIL)
    {
        // a failure in an ODBC function that prevents completion of the
        // test - for example, connect to the server
        vWrite(lpSI, "\t\t *** Unrecoverable RTNClient Test FAILURE ***", TRUE);
    } /* endif */

ExitNoDisconnect:

    return(sRC);
} // RetClient

```

```

BOOL FAR PASCAL Bind_First_RS(HSTMT hstmt)
{
    RETCODE rc = SQL_SUCCESS;
    rc = SQLBindCol(hstmt,1,SQL_C_CHAR,arraycol,ARRAYCOL_LEN+1, &cbcol1);
    if (rc != SQL_SUCCESS) return FALSE;
    return TRUE;
}

BOOL FAR PASCAL Bind_Second_RS(HSTMT hstmt)
{
    RETCODE rc = SQL_SUCCESS;
    rc = SQLBindCol(hstmt,1,SQL_C_CHAR,1stnam,LSTNAM_LEN+1,&dbcol2);
    if (rc != SQL_SUCCESS) return FALSE;
    return TRUE;
}

```

แอปพลิเคชัน JDBC

```

//-----
// Call Nested procedures which return result sets to the
// client, in this case a JDBC client.
//-----
import java.sql.*;
public class callNested
{
    public static void main (String argv[])           // Main entry point

```



```

{
    try {
        Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException e) {
        e.printStackTrace();
    }

    try {
        Connection jdbcCon =
DriverManager.getConnection("jdbc:db2:lp066ab","userid","xxxxxxx");
        jdbcCon.setAutoCommit(false);
        CallableStatement cs = jdbcCon.prepareCall("CALL PROD.RTNNESTED");
        cs.execute();
        ResultSet rs1 = cs.getResultSet();
        int r = 0;
while (rs1.next())
    {
        r++;
        String s1 = rs1.getString(1);
        System.out.print("Result set 1 Row: " + r + ": ");
        System.out.print(s1 + " ");
        System.out.println();
    }

        cs.getMoreResults();
        r = 0;
        ResultSet rs2 = cs.getResultSet();
        while (rs2.next())
        {
            r++;
            String s2 = rs2.getString(1);
            System.out.print("Result set 2 Row: " + r + ": ");
            System.out.print(s2 + " ");
            System.out.println();
        }
    }
    catch ( SQLException e ) {
        System.out.println( "SQLState: " + e.getSQLState() );
        System.out.println( "Message : " + e.getMessage() );
        e.printStackTrace();
    }
} // main
}

```

พารามิเตอร์ที่ผ่านหลักการสำหรับสำหรับโพรซีเจอร์ที่เก็บไว้และ UDFs

คำสั่ง CALL และการเรียกใช้ฟังก์ชันสามารถส่งผ่านอากิวเมนต์ไปยังโปรแกรมที่บันทึกในภาษาโฮสต์ และโพรซีเจอร์ REXX.

แต่ละภาษาจะสนับสนุนประเภทข้อมูลที่ต่างกันที่ถูกปรับแต่งสำหรับภาษานั้นๆ ดังแสดงในตารางต่อไปนี้. ประเภทข้อมูล SQL ถูกใส่ไว้ในคอลัมน์ด้านซ้ายสุดของตารางต่อไปนี้. คอลัมน์อื่นๆ ในแถวนั้นมีการบ่งชี้ว่ามีการสนับสนุนประเภทข้อมูลหรือไม่ในฐานที่เป็นประเภทพารามิเตอร์สำหรับภาษาเฉพาะ. หากคอลัมน์ว่างเปล่า, ชนิดข้อมูลจะไม่สนับสนุนในฐานเป็นประเภทพารามิเตอร์สำหรับภาษานั้น. การประกาศตัวแปรโฮสต์แสดงว่า DB2 SQL สำหรับ iSeries สนับสนุนประเภทข้อมูลนี้

ในฐานะที่เป็นพารามิเตอร์ในภาษานี้. การประกาศแสดงว่าตัวแปรโฮสต์จะต้องประกาศอย่างไรเพื่อให้ได้รับและตั้งอย่างถูกต้องด้วยโปรซีเจอร์หรือฟังก์ชัน. เมื่อเรียก SQL โปรซีเจอร์หรือฟังก์ชัน, ประเภทข้อมูล SQL ทั้งหมดจะถูกสับเปลี่ยน ดังนั้นจึงไม่มีคอลัมน์อยู่ในตาราง.

ตารางที่ 38. ชนิดข้อมูลของพารามิเตอร์

ชนิดข้อมูล SQL	C และ C++	CL	COBOL สำหรับ iSeries และ ILE COBOL สำหรับ iSeries
SMALLINT	short	N/A	PIC S9(4) BINARY
INTEGER	long	N/A	PIC S9(9) BINARY
BIGINT	long long	N/A	PIC S9(18) BINAR หมายเหตุ: จะสับเปลี่ยนเฉพาะ ILE COBOL สำหรับ iSeries เท่านั้น.
DECIMAL(p,s)	decimal(p,s)	TYPE(*DEC) LEN(p s)	PIC S9(p-s)V9(s) PACKED-DECIMAL หมายเหตุ: ความแม่นยำต้องไม่เกิน 18.
NUMERIC(p,s)	N/A	N/A	PIC S9(p-s)V9(s) DISPLAY SIGN LEADING SEPARATE หมายเหตุ: ความแม่นยำต้องไม่เกิน 18.
REAL หรือ FLOAT(p)	float	N/A	COMP-1 หมายเหตุ: จะสับเปลี่ยนเฉพาะ ILE COBOL สำหรับ iSeries เท่านั้น.
DOUBLE PRECISION หรือ FLOAT หรือ FLOAT(p)	double	N/A	COMP-2 หมายเหตุ: จะสับเปลี่ยนเฉพาะ ILE COBOL สำหรับ iSeries เท่านั้น.
CHARACTER(n)	char ... [n+1]	TYPE(*CHAR) LEN(n)	PIC X(n)
VARCHAR(n)	char ... [n+1]	N/A	สตริงอักขระความยาวแปรผัน
VARCHAR(n) FOR BIT DATA	รูปแบบโครงสร้าง VARCHAR	N/A	สตริงอักขระความยาวแปรผัน
CLOB	รูปแบบโครงสร้าง CLOB	N/A	รูปแบบโครงสร้าง CLOB หมายเหตุ: สับเปลี่ยนเฉพาะสำหรับ ILE COBOL สำหรับ iSeries.

ตารางที่ 38. ชนิดข้อมูลของพารามิเตอร์ (ต่อ)

ชนิดข้อมูล SQL	C และ C++	CL	COBOL สำหรับ iSeries และ ILE COBOL สำหรับ iSeries
GRAPHIC(n)	wchar_t ... [n+1]	N/A	PIC G(n) DISPLAY-1 or PIC N(n) หมายเหตุ: จะสนับสนุนเฉพาะ ILE COBOL สำหรับ iSeries เท่านั้น.
VARGRAPHIC(n)	รูปแบบโครงสร้าง VARGRAPHIC	N/A	กราฟิกสตริงความยาวแปรผัน หมายเหตุ: จะสนับสนุนเฉพาะ ILE COBOL สำหรับ iSeries เท่านั้น.
DBCLOB	รูปแบบโครงสร้าง DBCLOB	N/A	รูปแบบโครงสร้าง DBCLOB หมายเหตุ: สนับสนุนเฉพาะสำหรับ ILE COBOL สำหรับ iSeries.
BINARY	รูปแบบโครงสร้าง BINARY	N/A	รูปแบบโครงสร้าง BINARY
VARBINARY	รูปแบบโครงสร้าง VARBINARY	N/A	รูปแบบโครงสร้าง VARBINARY
BLOB	รูปแบบโครงสร้าง BLOB	N/A	รูปแบบโครงสร้าง BLOB หมายเหตุ: สนับสนุนเฉพาะสำหรับ ILE COBOL สำหรับ iSeries.
DATE	char ... [11]	TYPE(*CHAR) LEN(10)	PIC X(10) หมายเหตุ: สำหรับ ILE COBOL สำหรับ iSeries เท่านั้น, FORMAT DATE.
TIME	char ... [9]	TYPE(*CHAR) LEN(8)	PIC X(8) หมายเหตุ: สำหรับ ILE COBOL สำหรับ iSeries เท่านั้น, FORMAT TIME.
TIMESTAMP	char ... [27]	TYPE(*CHAR) LEN(26)	PIC X(26) หมายเหตุ: สำหรับ ILE COBOL สำหรับ iSeries เท่านั้น, FORMAT TIMESTAMP.
ROWID	รูปแบบโครงสร้าง ROWID	N/A	รูปแบบโครงสร้าง ROWID

ตารางที่ 38. ชนิดข้อมูลของพารามิเตอร์ (ต่อ)

ชนิดข้อมูล SQL	C และ C++	CL	COBOL สำหรับ iSeries และ ILE COBOL สำหรับ iSeries
DataLink	N/A	N/A	N/A
Indicator Variable	short	N/A	PIC S9(4) BINARY

ตารางที่ 39. ชนิดข้อมูลของพารามิเตอร์

ชนิดข้อมูล SQL	ลักษณะพารามิเตอร์ Java™ JAVA	ลักษณะพารามิเตอร์ Java DB2GENERAL	PL/I
SMALLINT	short	short	FIXED BIN(15)
INTEGER	int	int	FIXED BIN(31)
BIGINT	long	long	N/A
DECIMAL(p,s)	BigDecimal	BigDecimal	FIXED DEC(p,s)
NUMERIC(p,s)	BigDecimal	BigDecimal	N/A
REAL หรือ FLOAT(p)	float	float	FLOAT BIN(p)
DOUBLE PRECISION หรือ FLOAT หรือ FLOAT(p)	double	double	FLOAT BIN(p)
CHARACTER(n)	String	String	CHAR(n)
VARCHAR(n)	String	String	CHAR(n) VAR
VARCHAR(n) FOR BIT DATA	byte[]	com.ibm.db2.app.Blob	CHAR(n) VAR
CLOB	java.sql.Clob	com.ibm.db2.app.Clob	รูปแบบโครงสร้าง CLOB
GRAPHIC(n)	String	String	N/A
VARGRAPHIC(n)	String	String	N/A
DBCLOB	java.sql.Clob	com.ibm.db2.app.Clob	รูปแบบโครงสร้าง DBCLOB
BINARY	byte[]	com.ibm.db2.app.Blob	รูปแบบโครงสร้าง BINARY
VARBINARY	byte[]	com.ibm.db2.app.Blob	รูปแบบโครงสร้าง VARBINARY
BLOB	java.sql.Blob	com.ibm.db2.app.Blob	รูปแบบโครงสร้าง BLOB
DATE	Date	String	CHAR(10)
TIME	Time	String	CHAR(8)
TIMESTAMP	Timestamp	String	CHAR(26)
ROWID	byte[]	com.ibm.db2.app.Blob	รูปแบบโครงสร้าง ROWID

ตารางที่ 39. ชนิดข้อมูลของพารามิเตอร์ (ต่อ)

ชนิดข้อมูล SQL	ลักษณะพารามิเตอร์ Java™ JAVA	ลักษณะพารามิเตอร์ Java DB2GENERAL	PL/I
DataLink	N/A	N/A	N/A
Indicator Variable	N/A	N/A	FIXED BIN(15)

ตารางที่ 40. ชนิดข้อมูลของพารามิเตอร์

ชนิดข้อมูล SQL	REXX	RPG	ILE RPG
SMALLINT	N/A	โครงสร้างข้อมูลที่ประกอบด้วยฟิลด์ย่อยหนึ่งฟิลด์. B ในตำแหน่ง 43, ความยาวต้องเป็น 2, และ 0 ในตำแหน่ง 52 ของค่ากำหนดฟิลด์ย่อย.	ค่ากำหนดข้อมูล. B ในตำแหน่ง 40, ความยาวต้องเท่ากับ <=4, และ 00 ในตำแหน่ง 41-42 ของค่ากำหนดฟิลด์ย่อย. หรือ ค่ากำหนดข้อมูล. I ในตำแหน่ง 40, ความยาวต้องเท่ากับ 5, และ 00 ในตำแหน่ง 41-42 ของค่ากำหนดฟิลด์ย่อย.
INTEGER	สตริงตัวเลขที่ไม่มีทศนิยม (และเครื่องหมายนำหน้าเสริม)	โครงสร้างข้อมูลที่ประกอบด้วยฟิลด์ย่อยหนึ่งฟิลด์. B ในตำแหน่ง 43, ความยาวต้องเท่ากับ 4, และ 0 ในตำแหน่ง 52 ของค่ากำหนดฟิลด์ย่อย.	ค่ากำหนดข้อมูล. B ในตำแหน่ง 40, ความยาวต้องเป็น <=09 และ >=05, และ 00 ในตำแหน่ง 41-42 ของค่ากำหนดฟิลด์ย่อย. หรือ ค่ากำหนดข้อมูล. I ในตำแหน่ง 40, ความยาวต้องเท่ากับ 10, และ 00 ในตำแหน่ง 41-42 ของค่ากำหนดฟิลด์ย่อย.
BIGINT	N/A	N/A	ค่ากำหนดข้อมูล. I ในตำแหน่ง 40, ความยาวต้องเท่ากับ 20, และ 00 ในตำแหน่ง 41-42 ของค่ากำหนดฟิลด์ย่อย.
DECIMAL(p,s)	สตริงตัวเลขที่มีทศนิยม (และเครื่องหมายนำหน้าเสริม)	โครงสร้างข้อมูลที่ประกอบด้วยฟิลด์ย่อยหนึ่งฟิลด์. P ในตำแหน่ง 43 และ 0 ถึง 9 ในตำแหน่ง 52 ของค่ากำหนดฟิลด์ย่อย. หรือฟิลด์ใส่ข้อมูลตัวเลขหรือฟิลด์ผลการคำนวณ.	ค่ากำหนดข้อมูล. P ในตำแหน่ง 40 และ 00 ถึง 31 ในตำแหน่ง 41-42 ของค่ากำหนดฟิลด์ย่อย.
NUMERIC(p,s)	N/A	โครงสร้างข้อมูลที่ประกอบด้วยฟิลด์ย่อยหนึ่งฟิลด์. Blank ในตำแหน่ง 43 และ 0 ถึง 9 ในตำแหน่ง 52 ของค่ากำหนดฟิลด์ย่อย.	ค่ากำหนดข้อมูล. S ในตำแหน่ง 40, หรือ Blank ในตำแหน่ง 40 และ 00 ถึง 31 ในตำแหน่ง 41-42 ของค่ากำหนดฟิลด์ย่อย.

ตารางที่ 40. ชนิดข้อมูลของพารามิเตอร์ (ต่อ)

ชนิดข้อมูล SQL	REXX	RPG	ILE RPG
REAL หรือ FLOAT(p)	สตริงที่มีดีจิต, ตามด้วย E, (และตามด้วยเครื่องหมายเสริม), ตามด้วยดีจิต	N/A	ค่ากำหนดข้อมูล. <i>F</i> ในตำแหน่ง 40, ความยาวต้องเท่ากับ 4.
DOUBLE PRECISION หรือ FLOAT หรือ FLOAT(p)	สตริงที่มีดีจิต, ตามด้วย E, (และตามด้วยเครื่องหมายเสริม), ตามด้วยดีจิต	N/A	ค่ากำหนดข้อมูล. <i>F</i> ในตำแหน่ง 40, ความยาวต้องเท่ากับ 8.
CHARACTER(n)	สตริงที่มีอักขระ n ตัว ที่อยู่ในเครื่องหมายย่อสองตัว	ฟิลด์โครงสร้างข้อมูลที่ไม่มีฟิลด์ย่อยหรือโครงสร้างข้อมูลที่ประกอบด้วยฟิลด์ย่อยหนึ่งฟิลด์. <i>Blank</i> ในตำแหน่ง 43 และ 52 ของค่ากำหนดฟิลด์ย่อย. หรือฟิลด์ใส่ข้อมูลอักขระ A หรือฟิลด์ผลการคำนวณ.	ค่ากำหนดข้อมูล. <i>A</i> ในตำแหน่ง 40, หรือ <i>Blank</i> ในตำแหน่ง 40 และ 41-42 ของค่ากำหนดฟิลด์ย่อย.
VARCHAR(n)	สตริงที่มีอักขระ n ตัว ที่อยู่ในเครื่องหมายย่อสองตัว	N/A	ค่ากำหนดข้อมูล. <i>A</i> ในตำแหน่ง 40, หรือ <i>Blank</i> ในตำแหน่ง 40 และ 41-42 ของค่ากำหนดฟิลด์ย่อยและคีย์เวิร์ด VARYING ในตำแหน่ง 44-80.
VARCHAR(n) FOR BIT DATA	สตริงที่มีอักขระ n ตัว ที่อยู่ในเครื่องหมายย่อสองตัว	N/A	ค่ากำหนดข้อมูล. <i>A</i> ในตำแหน่ง 40, หรือ <i>Blank</i> ในตำแหน่ง 40 และ 41-42 ของค่ากำหนดฟิลด์ย่อยและคีย์เวิร์ด VARYING ในตำแหน่ง 44-80.
CLOB	N/A	N/A	รูปแบบโครงสร้าง CLOB
GRAPHIC(n)	สตริงขึ้นต้นด้วย G', ตามด้วยอักขระ n ที่มีไบต์สองเท่า, ตามด้วย '	N/A	ค่ากำหนดข้อมูล. <i>G</i> ในตำแหน่ง 40 ของค่ากำหนดฟิลด์ย่อย.
VARGRAPHIC(n)	สตริงขึ้นต้นด้วย G', ตามด้วยอักขระ n ที่มีไบต์สองเท่า, ตามด้วย '	N/A	ค่ากำหนดข้อมูล. <i>G</i> ในตำแหน่ง 40 ของค่ากำหนดฟิลด์ย่อยและคีย์เวิร์ด VARYING ในตำแหน่ง 44-80.
DBCLOB	N/A	N/A	รูปแบบโครงสร้าง DBCLOB
BINARY	N/A	N/A	รูปแบบโครงสร้าง BINARY
VARBINARY	N/A	N/A	รูปแบบโครงสร้าง VARBINARY
BLOB	N/A	N/A	รูปแบบโครงสร้าง BLOB

ตารางที่ 40. ชนิดข้อมูลของพารามิเตอร์ (ต่อ)

ชนิดข้อมูล SQL	REXX	RPG	ILE RPG
DATE	สตริงที่มีอักขระ 10 ตัว ที่อยู่ในเครื่องหมายย่อสองตัว	ฟิลด์โครงสร้างข้อมูลที่ไม่มีฟิลด์ย่อย หรือโครงสร้างข้อมูลที่ประกอบด้วยฟิลด์ย่อยหนึ่งฟิลด์. Blank ในตำแหน่ง 43 และ 52 ของค่า กำหนดฟิลด์ย่อย. ความยาวเท่ากับ 10. หรือฟิลด์ใส่ข้อมูลอักขระ A หรือฟิลด์ ผลการคำนวณ.	ค่ากำหนดข้อมูล. D ในตำแหน่ง 40 ของค่ากำหนดฟิลด์ย่อย. DATFMT (*ISO) ในตำแหน่ง 44-80.
TIME	สตริงที่มีอักขระ 8 ตัว ที่อยู่ในเครื่องหมายย่อสองตัว	ฟิลด์โครงสร้างข้อมูลที่ไม่มีฟิลด์ย่อย หรือโครงสร้างข้อมูลที่ประกอบด้วยฟิลด์ย่อยหนึ่งฟิลด์. Blank ในตำแหน่ง 43 และ 52 ของค่า กำหนดฟิลด์ย่อย. ความยาวเท่ากับ 8. หรือฟิลด์ใส่ข้อมูลอักขระ A หรือฟิลด์ ผลการคำนวณ.	ค่ากำหนดข้อมูล. T ในตำแหน่ง 40 ของค่ากำหนดฟิลด์ย่อย. TIMFMT (*ISO) ในตำแหน่ง 44-80.
TIMESTAMP	สตริงที่มีอักขระ 26 ตัว ที่อยู่ในเครื่องหมายย่อสองตัว	ฟิลด์โครงสร้างข้อมูลที่ไม่มีฟิลด์ย่อย หรือโครงสร้างข้อมูลที่ประกอบด้วยฟิลด์ย่อยหนึ่งฟิลด์. Blank ในตำแหน่ง 43 และ 52 ของค่า กำหนดฟิลด์ย่อย. ความยาวเท่ากับ 26. หรือฟิลด์ใส่ข้อมูลอักขระ A หรือฟิลด์ ผลการคำนวณ.	ค่ากำหนดข้อมูล. Z ในตำแหน่ง 40 ของค่ากำหนดฟิลด์ย่อย.
ROWID	N/A	N/A	รูปแบบโครงสร้าง ROWID
DataLink	N/A	N/A	N/A
Indicator Variable	สตริงตัวเลขที่ไม่มีทศนิยม (และเครื่องหมายนำหน้าเสริม).	โครงสร้างข้อมูลที่ประกอบด้วยฟิลด์ย่อยหนึ่งฟิลด์. B ในตำแหน่ง 43, ความยาวต้องเป็น 2, และ 0 ในตำแหน่ง 52 ของค่ากำหนดฟิลด์ย่อย.	ค่ากำหนดข้อมูล. B ใน ตำแหน่ง 40, ความยาวต้องเท่ากับ <=4, และ 00 ในตำแหน่ง 41-42 ของค่ากำหนดฟิลด์ย่อย.

ข้อมูลที่เกี่ยวข้อง

Embedded SQL programming

Java SQL routines

ตัวแปรตัวบ่งชี้และโพรซีเจอร์ที่เก็บไว้

สามารถใช้ตัวแปรตัวบ่งชี้ด้วยความ CALL, หากตัวแปรโฮสต์ถูกใช้สำหรับ พารามิเตอร์, เพื่อส่งผ่านข้อมูลเพิ่มเติมไปยัง และส่งจากโพรซีเจอร์.

ตัวแปรตัวบ่งชี้คือค่ากลาง มาตรฐาน SQL ของการบ่งชี้ว่าตัวแปรโฮสต์ที่เกี่ยวข้องควรถูกตีความว่า ประกอบด้วยค่าศูนย์, และ คือการใช้หลัก.

เพื่อแสดงให้เห็นว่าตัวแปรโฮสต์ที่เกี่ยวข้องประกอบด้วยค่าศูนย์, ตัวแปร ตัวบ่งชี้, ซึ่งเป็นจำนวนเต็มขนาดสองไบต์, จึงถูกตั้งให้เป็นค่าลบ. ข้อความ CALL ที่มีตัวแปรตัวบ่งชี้ถูกประมวลผลดังนี้:

- หากตัวแปรตัวบ่งชี้มีค่าเป็นลบ, จะหมายถึงค่าศูนย์. มีการส่งค่าดีฟอลต์สำหรับตัวแปรโฮสต์ที่เกี่ยวข้องบน CALL และตัวแปรตัวบ่งชี้จะถูกส่งผ่านเหมือนเดิม.
- หากตัวแปรตัวบ่งชี้ไม่ใช่ค่าลบ, แสดงว่าตัวแปรโฮสต์ประกอบด้วยค่าที่ไม่ใช่ศูนย์. ในกรณีนี้, ตัวแปรโฮสต์และตัวแปรตัวบ่งชี้จะถูกส่งผ่านเหมือนเดิม.

กฎการประมวลผลเหล่านี้เหมือนกับกฎสำหรับอินพุตพารามิเตอร์ที่ไปยังโพรซีเจอร์ และเอาต์พุตพารามิเตอร์ที่ส่งคืนจากโพรซีเจอร์. เมื่อตัวแปรตัวบ่งชี้ถูกใช้งานด้วย โพรซีเจอร์ที่เก็บไว้, วิธีการที่ถูกต้องในการโค้ดการจัดการคือการตรวจสอบค่าตัวแปรตัวบ่งชี้ก่อน ที่จะใช้ตัวแปรโฮสต์ที่เกี่ยวข้อง.

ตัวอย่างต่อไปนี้แสดงถึงการจัดการตัวแปรตัวบ่งชี้ในข้อความ CALL. โปรดสังเกตว่าตรรกะจะตรวจสอบค่าตัวแปรตัวบ่งชี้ก่อนที่จะใช้ตัวแปรที่เกี่ยวข้อง. และสังเกตเพิ่มเติมถึงวิธีการที่ตัวแปรตัวบ่งชี้ถูกส่งผ่านเข้าไปยังโพรซีเจอร์ PROC1 (ในฐานะที่เป็นอากิวเมนต์ที่สามซึ่งประกอบด้วยอะเรย์ของค่าขนาดสองไบต์).

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

สมมติว่าโพรซีเจอร์ถูกกำหนดดังนี้:

```
CREATE PROCEDURE PROC1
  (INOUT DECIMALOUT DECIMAL(7,2), INOUT DECOUT2 DECIMAL(7,2))
  EXTERNAL NAME LIB1.PROC1 LANGUAGE RPGLE
  GENERAL WITH NULLS)
```

การจัดการกับตัวแปรตัวบ่งชี้ในคำสั่ง CALL

```
+++++
โปรแกรม CRPG
+++++
      D INOUT1          S           7P 2
      D INOUT1IND       S           4B 0
      D INOUT2          S           7P 2
      D INOUT2IND       S           4B 0
      C                 EVAL       INOUT1 = 1
      C                 EVAL       INOUT1IND = 0
      C                 EVAL       INOUT2 = 1
      C                 EVAL       INOUT2IND = -2
      C/EXEC SQL CALL PROC1 (:INOUT1 :INOUT1IND , :INOUT2
      C+                 :INOUT2IND)
      C/END-EXEC
      C                 EVAL       INOUT1 = 1
      C                 EVAL       INOUT1IND = 0
      C                 EVAL       INOUT2 = 1
      C                 EVAL       INOUT2IND = -2
      C/EXEC SQL CALL PROC1 (:INOUT1 :INOUT1IND , :INOUT2
      C+                 :INOUT2IND)
      C/END-EXEC
      C   INOUT1IND      IFLT       0
      C*                :
      C*                HANDLE NULL INDICATOR
      C*                :
      C                 ELSE
      C*                :
```



```

C*          INOUT1 CONTAINS VALID DATA
C*          :
C          ENDIF
C*          :
C*          HANDLE ALL OTHER PARAMETERS
C*          IN A SIMILAR FASHION
C*          :
C          RETURN
+++++
สิ้นสุด PROGRAM CRPG
+++++
โปรแกรม PROC1
+++++
D INOUTP      S          7P 2
D INOUTP2     S          7P 2
D NULLARRAY   S          4B 0 DIM(2)
C  *ENTRY     PLIST
C             PARM          INOUTP
C             PARM          INOUTP2
C             PARM          NULLARRAY
C  NULLARRAY(1) IFLT      0
C*          :
C*          INOUTP DOES NOT CONTAIN MEANINGFUL DATA
C*
C          ELSE
C*          :
C*          INOUTP CONTAINS MEANINGFUL DATA
C*          :
C          ENDIF
C*          PROCESS ALL REMAINING VARIABLES
C*
C*          BEFORE RETURNING, SET OUTPUT VALUE FOR FIRST
C*          PARAMETER AND SET THE INDICATOR TO A NON-NEGATIV
C*          VALUE SO THAT THE DATA IS RETURNED TO THE CALLING
C*          PROGRAM
C*
C          EVAL      INOUTP2 = 20.5
C          EVAL      NULLARRAY(2) = 0
C*
C*          INDICATE THAT THE SECOND PARAMETER IS TO CONTAIN
C*          THE NULL VALUE UPON RETURN. THERE IS NO POINT
C*          IN SETTING THE VALUE IN INOUTP SINCE IT WON'T BE
C*          PASSED BACK TO THE CALLER.
C          EVAL      NULLARRAY(1) = -5
C          RETURN
+++++
สิ้นสุด PROGRAM PROC1
+++++

```

การย้อนกลับสถานะที่สมบูรณ์ไปยังโปรแกรมการเรียก

สำหรับ SQL โพรซีเจอร์, ข้อผิดพลาดใดๆ ที่ไม่ได้รับการจัดการในโพรซีเจอร์จะถูกส่งคืนมาที่ตัวเรียกใน SQLCA.

สามารถ SIGNAL และ RESIGNAL control statement เพื่อส่งข้อมูลข้อผิดพลาดได้เช่นกัน.

สำหรับโปรแกรมเมอร์ภายนอก, มีสองวิธีการในการส่งข้อมูลสถานะกลับ. วิธีการที่หนึ่งในการส่งคืนสถานะไปที่โปรแกรม SQL ที่ส่งข้อความ CALL คือให้โค้ดพารามิเตอร์ประเภท INOUT พิเศษและเซตไว้ก่อนที่จะกลับคืนจากโปรแกรมเมอร์ดังกล่าว. เมื่อโปรแกรมเมอร์ที่เรียกคือโปรแกรมที่มีอยู่แล้ว, วิธีการข้างต้นย่อมเป็นไปได้.

วิธีการที่สองในการส่งคืนสถานะไปที่โปรแกรม SQL ที่ส่งข้อความ CALL คือให้ส่ง escape message ไปยังโปรแกรมการเรียก (โปรแกรมระบบปฏิบัติการ QSQCALL) ที่เรียกโปรแกรมเมอร์. โปรแกรมการเรียกที่เรียกใช้งานโปรแกรมเมอร์คือ QSQCALL. แต่ภาษาไม่มีวิธีการสำหรับเงื่อนไขการส่งสัญญาณ และการส่งข้อความ. โปรดดูที่การอ้างอิงแต่ละภาษาเพื่อกำหนดวิธีการที่เหมาะสมในการส่งสัญญาณข้อความ. เมื่อมีการส่งสัญญาณข้อความ, QSQCALL จะแปลงข้อผิดพลาดเป็น SQLCODE/SQLSTATE -443/38501.

ข้อมูลที่เกี่ยวข้อง

SQL Control Statements

การใช้ user-defined functions (UDFs)

ในการเขียนแอพลิเคชัน SQL, คุณสามารถเลือกปฏิบัติการหรือดำเนินการบางอย่างได้ในแบบ UDF หรือ แบบรูทีนย่อยในแอพลิเคชันของคุณ: ถึงแม้ว่ามันอาจจะดูง่ายกว่าในการเลือกการดำเนินการใหม่แบบรูทีนย่อยในแอพลิเคชันของคุณ, คุณอาจต้องพิจารณาถึงประโยชน์ของการใช้งาน UDF แทน.

ตัวอย่างเช่น, ถ้าการดำเนินการใหม่เป็นสิ่งที่ผู้ใช้งานหรือโปรแกรมอื่นๆ สามารถได้รับประโยชน์, รูปแบบ UDF สามารถช่วยในการนำมาใช้งานได้อีก. นอกจากนี้, เราสามารถเรียกฟังก์ชันได้โดยตรงใน SQL ในที่ใดก็ตามที่สามารถใช้ฟังก์ชันได้. ฐานข้อมูลจะดูแลชนิดข้อมูลทั้งหลายของฟังก์ชันอักขระให้โดยอัตโนมัติ. ตัวอย่างเช่น, จาก DECIMAL ไปเป็น DOUBLE, ฐานข้อมูลอนุญาตให้ฟังก์ชันของคุณใช้ชนิดข้อมูลที่แตกต่างกันได้, แต่ต้องทำงานร่วมกันได้.

ในบางกรณี, การเรียก UDF โดยตรงจากเอ็นจินฐานข้อมูลแทนที่จะเรียกจากแอพลิเคชันของคุณสามารถทำให้ประสิทธิภาพดีขึ้นอย่างมาก. คุณอาจจะสังเกตเห็นประสิทธิภาพที่เพิ่มขึ้นได้ในกรณีที่ฟังก์ชันถูกใช้ในการตรวจสอบข้อมูลสำหรับการประมวลผลครั้งต่อไป. กรณีนี้จะเกิดขึ้นก็ต่อเมื่อฟังก์ชันถูกใช้ในกระบวนการเลือกแถว.

พิจารณาสถานการณ์ง่ายๆ เมื่อคุณต้องการดำเนินการกับบางข้อมูล. คุณอาจเจอเงื่อนไขการเลือกบางอย่างที่สามารถแสดงเป็นแบบฟังก์ชัน SELECTION_CRITERIA() ได้. แอพลิเคชันของคุณสามารถเรียกใช้คำสั่ง select ดังต่อไปนี้:

```
SELECT A, B, C FROM T
```

เมื่อได้รับข้อมูลแต่ละแถวแล้ว, จะเรียกใช้ฟังก์ชัน SELECTION_CRITERIA กับข้อมูลเหล่านั้นเพื่อตัดสินใจว่าข้อมูลนั้นเป็นที่สนใจในการประมวลผลข้อมูลต่อไปหรือไม่. นั่นคือ, ทุกแถวของตาราง T ต้องถูกส่งกลับไปยังแอพลิเคชัน. แต่, ถ้า SELECTION_CRITERIA() ถูกสร้างเป็นแบบ UDF แล้ว, แอพลิเคชันของคุณสามารถเรียกใช้คำสั่งดังต่อไปนี้ได้:

```
SELECT C FROM T WHERE SELECTION_CRITERIA(A,B)=1
```

ในกรณีนี้, มีเพียงแถวที่อยู่ในคอลัมน์เดียวที่สนใจเท่านั้นที่จะถูกส่งข้ามไปมาระหว่างอินเตอร์เฟซของแอพลิเคชันและฐานข้อมูล.

กรณีอื่นๆ ที่ UDF สามารถช่วยเพิ่มประสิทธิภาพก็คือเมื่อต้องทำงานกับ Large Objects (LOB). สมมุติว่าคุณมีฟังก์ชันที่ดึงข้อมูลจากค่าของหนึ่งในชนิด LOB. คุณสามารถทำการดึงข้อมูลนั้นบนเซิร์ฟเวอร์ฐานข้อมูลแล้วส่งผ่านเฉพาะข้อมูลที่ดึงออกมาไปยังแอพลิเคชันได้. วิธีนี้จะมีประสิทธิภาพมากกว่าการส่งผ่านค่า LOB ทั้งหมดกลับไปยังแอพลิเคชันแล้วค่อยทำการดึงข้อมูลออกมา. ค่าประสิทธิภาพของการจัดแพ็คเกจฟังก์ชันนี้เป็นแบบ UDF อาจจะมีค่าสูงมาก, ซึ่งจะขึ้นอยู่กับสถานการณ์เฉพาะ.

หลักการที่เกี่ยวข้อง

“ฟังก์ชันแบบผู้ใช้กำหนด (User-defined functions)” ในหน้า 12

ฟังก์ชันแบบผู้ใช้กำหนดเอง คือโปรแกรมที่อาจถูกเรียกทำงานได้เหมือนกับฟังก์ชันในตัวอื่นๆ.

แนวคิดของ UDF

หัวข้อนี้เป็นการพิจารณาถึงแนวคิดที่สำคัญที่คุณต้องทราบก่อนการโค้ด UDF.

ชนิดของฟังก์ชัน

มีชนิดของฟังก์ชันอยู่หลายชนิด:

- *ในตัว (Built-in)*. คือฟังก์ชันที่ถูกจัดเตรียมให้และมาพร้อมกับฐานข้อมูล. ตัวอย่างคือ SUBSTR().
- *ระบบสร้างให้ (System-generated)*. ฟังก์ชันนี้จะถูกสร้างโดยตรงโดยเอ็นจินฐานข้อมูลเมื่อ DISTINCT TYPE ถูกสร้างขึ้น. ฟังก์ชันนี้จัดเตรียมตัวดำเนินการเปลี่ยนชนิดข้อมูลระหว่าง DISTINCT TYPE และชนิดพื้นฐานของ DISTINCT TYPE นั้น.
- *ผู้ใช้กำหนดเอง (User-defined)*. ฟังก์ชันนี้ถูกสร้างโดยผู้ใช้แล้วจึงลงทะเบียนไปที่ฐานข้อมูล.

นอกเหนือจากนี้, แต่ละฟังก์ชันสามารถถูกจัดหมวดหมู่เป็นฟังก์ชันแบบ *Scalar*, ฟังก์ชันแบบ *Column*, หรือฟังก์ชันแบบ *Table*.

ฟังก์ชันแบบ Scalar จะคืนค่าผลลัพธ์เดียวในแต่ละครั้งที่เรียกใช้. ตัวอย่างเช่น, ฟังก์ชันในตัว SUBSTR() คือฟังก์ชันแบบ *Scalar*, ดังเช่นฟังก์ชันในตัวหลายๆฟังก์ชัน. ฟังก์ชันที่ระบบสร้างให้ (System-generated function) จะเป็นฟังก์ชันแบบ *Scalar* เสมอ. *Scalar UDFs* สามารถเป็นได้ทั้งส่วนภายนอก (โค้ดในภาษาโปรแกรมเช่น C), เขียนใน SQL, หรือ ในต้นฉบับ (การใช้ในการนำไปปฏิบัติของฟังก์ชันที่มีอยู่แล้ว).

ฟังก์ชันแบบ Column จะรับชุดของค่าที่คล้ายกัน (คอลัมน์ของข้อมูล) และคืนค่าผลลัพธ์เดียวจากชุดของค่าเหล่านั้น. ซึ่งยังถูกเรียกว่า *ฟังก์ชันรวม* ใน DB2. ฟังก์ชันในตัวบางฟังก์ชันก็เป็นฟังก์ชันแบบ *Column*. ตัวอย่างของฟังก์ชันแบบ *Column* คือ ฟังก์ชันในตัว AVG(). UDF ภายนอกไม่สามารถกำหนดให้เป็นฟังก์ชันแบบ *Column* ได้. อย่างไรก็ตาม, UDF ต้นฉบับจะถูกกำหนดให้เป็นฟังก์ชันแบบ *Column* ถ้าฟังก์ชันต้นฉบับเป็นฟังก์ชันแบบ *Column*. อย่างหลังสุดจะใช้ประโยชน์ได้มากสำหรับ *Distinct Types*. ตัวอย่างเช่น, ถ้ามี *Distinct Type* ชื่อ SHOESIZE อยู่และถูกกำหนดค่าพื้นฐานเป็น INTEGER, คุณสามารถกำหนด UDF, AVG(SHOESIZE), ให้เป็นฟังก์ชันแบบ *Column* ต้นฉบับบนฟังก์ชันแบบ *Column*, AVG(INTEGER) ได้.

ฟังก์ชันแบบ Table จะส่งคืนค่าตารางให้กับคำสั่ง SQL ที่อ้างอิงถึงฟังก์ชันนั้น. มันต้องถูกอ้างอิงในอนุประโยค FROM ของ SELECT. ฟังก์ชันตารางสามารถนำมาใช้เพิ่มความสามารถในการประมวลผลภาษา SQL กับข้อมูลที่ไม่ได้เป็นข้อมูลชนิด DB2, หรือเพื่อแปลงข้อมูลนั้นไปเป็นรูปแบบตาราง DB2. ซึ่งสามารถ, ยกตัวอย่างเช่น, ดึงไฟล์ และแปลงเป็นรูปแบบตาราง, ข้อมูลตัวอย่างจากใน World Wide Web และนำมาจัดเรียงเป็นตาราง, หรือเข้าถึงฐานข้อมูล Lotus Notes® และส่งกลับข้อมูลเกี่ยวกับข้อความอีเมล, เช่น วันที่, ผู้ส่ง, และเนื้อหาของข้อความนั้น. ข้อมูลเหล่านี้สามารถเชื่อมกับตารางอื่นๆในฐานข้อมูลได้. ฟังก์ชันแบบ *Table* สามารถถูกกำหนดให้เป็นแบบฟังก์ชันภายนอกหรือฟังก์ชัน SQL ได้; แต่ว่าจะไม่สามารถถูกกำหนดให้เป็นฟังก์ชันต้นฉบับได้.

ชื่อเต็มของฟังก์ชัน

ชื่อเต็มของฟังก์ชันที่ใช้การตั้งชื่อของ *SQL คือ <schema-name>.<function-name>.

ชื่อเต็มของฟังก์ชันในการตั้งชื่อของ *SYS คือ <schema-name>/<function-name>. ชื่อฟังก์ชันไม่สามารถครบตามเกณฑ์ถ้าใช้การตั้งชื่อโดย *SYS ในคำสั่ง DML.

คุณสามารถใช้ชื่อเต็มนี้ได้ในทุกที่ที่คุณอ้างอิงถึงฟังก์ชัน. ตัวอย่างเช่น:

```
QGPL.SNOWBLOWER_SIZE SMITH.FOO QSYS2.SUBSTR QSYS2.FLOOR
```

อย่างไรก็ตาม, คุณยังอาจจะเว้น <schema-name>., ในบางกรณี, DB2 ต้องพิจารณาว่า ฟังก์ชันใดที่คุณกำลังอ้างอิงอยู่. ตัวอย่างเช่น:

```
SNOWBLOWER_SIZE FOO SUBSTR FLOOR
```

พาร

แนวคิดเกี่ยวกับพาร จะมุ่งเน้นไปที่ความชัดเจนของ DB2 ในการอ้างอิงที่ *ไม่แน่นอน* ซึ่งเกิดขึ้นเมื่อไม่ได้มีการระบุชื่อแบบแผนที่ชัดเจน. พาร เป็นลำดับรายการของรูปแบบชื่อ ที่ใช้สำหรับการแก้ไขการอ้างอิงที่ไม่ชัดเจนกับ UDFs และ UDTs. ในกรณีที่มีการอ้างอิงฟังก์ชันไปตรงกับฟังก์ชันมากกว่าหนึ่งรูปแบบในพาร, ลำดับของรูปแบบในพารจะถูกใช้เพื่อแก้ปัญหาคงตรงกันนี้. พารถูกกำหนดขึ้นโดยตัวเลือก SQLPATH บนคำสั่งพรีคอมไพเลอร์สำหรับ static SQL. พารถูกตั้งค่าโดยคำสั่ง SET PATH สำหรับ dynamic SQL. เมื่อคำสั่ง SQL แรกที่ทำงานใน activation group ซึ่งทำงานด้วยการตั้งชื่อของ SQL, พารจะมีค่าดีฟอลต์ดังต่อไปนี้:

```
"QSYS", "QSYS2", "<ID>"
```

ค่านี้จะใช้ได้ทั้งใน SQL ทั้งแบบ static และ dynamic, โดยที่ <ID> คือ authorization ID ของคำสั่งปัจจุบัน.

เมื่อคำสั่ง SQL แรกที่ทำงานใน activation group ทำงานด้วยการตั้งชื่อของระบบ, ค่าดีฟอลต์คือ *LIBL.

ชื่อฟังก์ชันที่ถูก Overloaded

ชื่อฟังก์ชันสามารถ *Overloaded* ได้. การ Overloaded หมายความว่าหลายฟังก์ชัน, แม้ว่าจะอยู่ใน Schema เดียวกัน, สามารถใช้ชื่อเดียวกันได้. อย่างไรก็ตาม, สองฟังก์ชันไม่สามารถ, มี *signature* เหมือนกันได้. Signature ของฟังก์ชันสามารถกำหนดให้เป็นค่าชื่อฟังก์ชันที่ถูกตามเกณฑ์เชื่อมต่อกับชนิดข้อมูลของพารามิเตอร์ของฟังก์ชันทั้งหมดตามลำดับที่พารามิเตอร์เหล่านั้นถูกนิยาม.

การแก้ปัญหาของฟังก์ชัน

อัลกอริธึมการแก้ปัญหาฟังก์ชัน คือสิ่งที่นำมาใช้สำหรับการ Overloading และฟังก์ชันพาร เพื่อเลือกชื่อที่เหมาะสมที่สุดสำหรับทุกการอ้างอิงฟังก์ชัน, โดยไม่สนใจว่าการอ้างอิงครบตามเกณฑ์หรือไม่. ทุกฟังก์ชัน, รวมถึงฟังก์ชันในตัวด้วย, จะถูกดำเนินการด้วยอัลกอริธึมการเลือกฟังก์ชัน. อัลกอริธึมการแก้ปัญหาฟังก์ชันไม่ได้ถูกใช้ในการแก้ปัญหานิดของฟังก์ชัน. ดังนั้นฟังก์ชันตารางอาจถูก resolve ได้ราวกับเป็นฟังก์ชันที่ *เหมาะสมที่สุด*, แม้ว่า การใช้การอ้างอิงจะต้องการฟังก์ชัน scalar, หรือในทำนองกลับกัน.

ระยะเวลาที่ UDF รัน

เราเรียกใช้ UDFs จากการทำงานภายในข้อความ SQL, ซึ่งโดยปกติการปฏิบัติการ query ซึ่งมีศักยภาพในการทำงานกับจำนวนแถวนับพันแถวในตารางได้. ด้วยเหตุนี้, จึงจำเป็นต้องเรียกใช้ UDF จากฐานข้อมูลระดับต่ำ.

ผลของการถูกเรียกใช้งานจากระดับต่ำดังกล่าว, ทำให้รีซอร์สบางตัว (การล็อกและการยึด) ถูกเร่งการทำงานชั่วคราว ณ. ขณะที่มีการเรียกใช้ UDF และในระหว่างการทำงานของ UDF. รีซอร์สเหล่านี้คือตัวล็อกหลักบนตารางและตรรกษณ์ใดๆ ที่เกี่ยวข้องกับคำสั่ง SQL ซึ่งกำลังเรียก UDF ทำงาน. เนื่องจากรีซอร์สถูกเร่งการทำงาน, UDF จึงไม่ควรดำเนินการที่อาจใช้ระยะ

เวลาเกินไป (หลายนาทีหรือหลายชั่วโมง). เนื่องมาจากลักษณะที่สำคัญในการระงับการทำงานของรีซอร์สเป็นเวลานาน, ฐานข้อมูลจะคอยสักระยะเวลาหนึ่งเพื่อให้ UDF ทำงานเสร็จสิ้นก่อน. ถ้า UDF ทำงานไม่เสร็จสิ้นภายในเวลาที่กำหนดไว้, คำสั่ง SQL ที่กำลังเรียกใช้งาน UDF อยู่จะล้มเหลวลง.

ระยะเวลาที่ฐานข้อมูลรอ UDF ซึ่งเป็นค่าดีฟอลต์นั้นควรจะนานเกินกว่าเวลาที่ใช้จริงเพื่อให้ UDF แบบปกติรันให้เสร็จสิ้น. อย่างไรก็ตาม, ถ้าคุณมีการรัน UDF ที่ยาวนาน และต้องการเพิ่มเวลาในการรอ, คุณสามารถทำได้โดยการใช้ตัวเลือก UDF_TIME_OUT ในไฟล์สอบถาม INI. อย่างไรก็ตาม, โปรดจำว่า, ฐานข้อมูลจะใช้เวลาได้ไม่เกินข้อจำกัดสูงสุดที่กำหนดไว้, ไม่ว่าค่าที่ระบุไว้สำหรับ UDF_TIME_OUT จะเป็นเท่าไรก็ตาม.

เนื่องจากรีซอร์สถูกระงับการทำงานขณะที่รัน UDF, UDF จึงไม่ดำเนินการบนตารางหรือดรรชนีเดียวกันซึ่งถูกกำหนดให้กับคำสั่ง SQL ต้นฉบับหรือ, หากว่า UDF ดำเนินการไปแล้ว, UDF จะไม่ดำเนินการที่ขัดกับการดำเนินการที่กำลังปฏิบัติการอยู่ในคำสั่ง SQL. โดยเฉพาะอย่างยิ่ง, UDF จะไม่พยายามดำเนินการแทรก, อัปเดต, หรือลบการดำเนินการของแถวในตารางเหล่านั้น.

ข้อมูลที่เกี่ยวข้อง

Query Options File (QAQQINI)

การเขียน UDF เป็นฟังก์ชัน SQL

ฟังก์ชัน SQL คือ UDF ที่คุณได้กำหนดไว้, เขียนไว้, และลงทะเบียนโดยใช้คำสั่ง CREATE FUNCTION SQL.

เมื่อเป็นเช่นนั้น, ฟังก์ชันนั้นจะถูกเขียนขึ้นโดยใช้เฉพาะภาษา SQL และ definition จะมีอยู่ภายในคำสั่ง CREATE FUNCTION เดียว (อาจมีขนาดใหญ่). การสร้างฟังก์ชัน SQL จะทำให้ UDF ได้รับการลงทะเบียน, สร้างโค้ดที่ทำงานได้สำหรับฟังก์ชัน, และกำหนดรายละเอียดการส่งผ่านพารามิเตอร์ให้กับฐานข้อมูล.

ตัวอย่าง: SQL scalar UDF:

ในตัวอย่างนี้, ฟังก์ชันจะส่งกลับระดับความสำคัญที่ขึ้นกับวันที่.

```
CREATE FUNCTION PRIORITY(indate DATE) RETURNS CHAR(7)
LANGUAGE SQL
BEGIN
RETURN(
CASE WHEN indate>CURRENT DATE-3 DAYS THEN 'HIGH'
      WHEN indate>CURRENT DATE-7 DAYS THEN 'MEDIUM'
      ELSE 'LOW'
END
);
END
```

ฟังก์ชันจะถูกเรียกทำงานเป็น:

```
SELECT ORDERNBR, PRIORITY(ORDERDUEDATE) FROM ORDERS
```

ตัวอย่าง: ตาราง SQL UDF:

ตัวอย่างนี้อธิบายถึงฟังก์ชันตารางที่ส่งคืนข้อมูลโดยดูจากวันที่.

```
CREATE FUNCTION PROJFUNC(indate DATE) RETURNS TABLE (PROJNO CHAR(6), ACTNO SMALLINT, ACTSTAFF DECIMAL(5,2),
ACSTDATE DATE, ACENDATE DATE)
LANGUAGE SQL
```

```
BEGIN
RETURN SELECT * FROM PROJACT
WHERE ACSTDATE<=indate;
END
```

ฟังก์ชันจะถูกเรียกทำงานเป็น:

```
SELECT * FROM TABLE(PROJFUNC(:datehv)) X
```

ฟังก์ชันตาราง SQL จะต้องมีส่วน RETURN หนึ่งคำสั่งเท่านั้น.

การเขียน UDF ให้เป็นฟังก์ชันภายนอก

คุณสามารถเขียนโค้ดที่ทำงานได้ของ UDF ในภาษาอื่นที่นอกเหนือจาก SQL.

ขณะที่วิธีนี้จะยุ่งยากกว่าฟังก์ชันแบบ SQL , แต่วิธีนี้มีความยืดหยุ่นให้คุณได้ใช้ภาษาใดก็ได้ที่มีประสิทธิภาพที่สุดสำหรับคุณ. สามารถเก็บโค้ดที่ใช้งานได้โปรแกรม หรือเซอริสโปรแกรม.

ฟังก์ชันแบบภายนอกสามารถถูกเขียนเป็น จาวา.

ข้อมูลที่เกี่ยวข้อง

Java SQL Routines

เรจิสเตอร์ UDF:

UDF ต้องถูกลงทะเบียนในฐานะข้อมูลก่อนที่ฟังก์ชันจะรู้จัก และถูกใช้โดย SQL.

คำสั่งอนุญาตให้คุณระบุภาษาและชื่อของโปรแกรมได้, รวมทั้งตัวเลือกอย่างเช่น DETERMINISTIC, ALLOW PARALLEL, และ RETURNS NULL ON NULL INPUT. ตัวเลือกเหล่านี้จะช่วยให้ฐานข้อมูลระบุเป้าหมายของฟังก์ชันได้ตรงขึ้น และช่วยระบุว่า วิธีการเรียกไปยังฐานข้อมูลสามารถทำการปรับปรุงประสิทธิภาพได้อย่างไร.

คุณควรเรจิสเตอร์ UDF แบบภายนอก หลังจากที่คุณได้เขียนและทดสอบโค้ดจริงได้ เสร็จสมบูรณ์. และเป็นไปได้ที่จะกำหนด UDF ก่อนที่จะเขียนจริง. อย่างไรก็ตาม, เพื่อหลีกเลี่ยงปัญหาเกี่ยวกับการรัน UDF ของคุณ, คุณควรเขียนและทดสอบให้ครอบคลุมก่อนที่จะทำการลงทะเบียน.

ข้อมูลที่เกี่ยวข้อง

CREATE FUNCTION statement

ตัวอย่าง: การยกกำลัง:

ในตัวอย่างนี้, สมมติว่า คุณได้เขียน UDF ภายนอกเพื่อทำการยกกำลังค่าตัวเลขทศนิยม, และต้องการลงทะเบียนในรูปแบบแผน MATH.

```
CREATE FUNCTION MATH.EXPON (DOUBLE, DOUBLE)
RETURNS DOUBLE
EXTERNAL NAME 'MYLIB/MYPGM(MYENTRY)'
LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL
```

```
DETERMINISTIC
NO EXTERNAL ACTION
RETURNS NULL ON NULL INPUT
ALLOW PARALLEL
```

ในตัวอย่างนี้, มีการระบุ RETURNS NULL ON NULL INPUT เนื่องจากคุณต้องการ ผลลัพธ์เป็น NULL ถ้าค่าอักขรเมมต้อันใดอันหนึ่งเป็น NULL. และเนื่องจากไม่มีเหตุผลใดที่ EXPON จะไม่สามารถทำงานคู่ขนานได้, ดังนั้นค่า ALLOW PARALLEL จึงถูกระบุ.

ตัวอย่าง: การค้นหาสตริง:

สมมติว่า คุณได้เขียน UDF เพื่อมองหาการมีอยู่ของสตริงสั้นๆ, ที่ส่งผ่านเป็นอักขรเมมต, ภายในค่า CLOB ที่ถูกกำหนดไว้ ซึ่งส่งผ่านเป็นอักขรเมมตเหมือนกัน. UDF จะคืนค่าตำแหน่งของสตริงภายใน CLOB ถ้าเจอสตริงนั้น, หรือคืนค่าศูนย์ถ้าหาไม่เจอ.

มีการเขียนโปรแกรมภาษา C เพื่อให้ผลลัพธ์เป็น FLOAT กลับมา. สมมติว่าคุณรู้ว่าเมื่อ UDF นี้ถูกใช้ใน SQL, มันจะคืนค่า INTEGER เสมอ. คุณสามารถสร้างฟังก์ชันดังต่อไปนี้:

```
CREATE FUNCTION FINDSTRING (CLOB(500K), VARCHAR(200)) RETURNS INTEGER
CAST FROM FLOAT
SPECIFIC FINDSTRING
EXTERNAL NAME 'MYLIB/MYPGM(FINDSTR) '
LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL
DETERMINISTIC
NO EXTERNAL ACTION
RETURNS NULL ON NULL INPUT
```

โปรดสังเกตว่าประโยค CAST FROM ใช้ระบุว่าโปรแกรม UDF ได้ส่งกลับค่า FLOAT จริง, แต่คุณต้องการแปลงค่านี้ให้เป็น INTEGER ก่อนการส่งกลับค่า มาที่ข้อความ SQL ซึ่งเรียกใช้ UDF นั้น. ดังนั้น, คุณต้องเตรียมชื่อเฉพาะของคุณเอง สำหรับฟังก์ชัน. เนื่องจาก UDF ไม่ได้ถูกเขียนมาเพื่อจัดการค่า NULL ได้, คุณต้องใช้ RETURNS NULL ON NULL INPUT.

ตัวอย่าง: การค้นหาสตริง BLOB:

ในตัวอย่างนี้, คุณต้องการให้ฟังก์ชัน FINDSTRING ทำงานบน BLOB เช่นเดียวกับ CLOB. หากต้องการทำสิ่งนี้, คุณต้องกำหนด FINDSTRING อีกหนึ่งตัวให้นำ BLOB มาเป็นพารามิเตอร์แรก:

```
CREATE FUNCTION FINDSTRING (BLOB(500K), VARCHAR(200)) RETURNS INTEGER
CAST FROM FLOAT
SPECIFIC FINDSTRING_BLOB
EXTERNAL NAME 'MYLIB/MYPGM(FINDSTR) '
LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL
DETERMINISTIC
NO EXTERNAL ACTION
RETURNS NULL ON NULL INPUT
```

ตัวอย่างนี้แสดงให้เห็นการ Overloading ของชื่อ UDF และแสดงให้เห็นว่า UDF หลายตัวสามารถใช้เนื้อหาร่วมกันได้. โปรดสังเกตว่าถึงแม้ว่า BLOB ไม่สามารถถูกกำหนดค่าให้กับ CLOB ได้, แต่สามารถใช้ซอร์สโค้ดเดียวกันได้. ไม่มีปัญหาการโปรแกรมมิงในตัวอย่างข้างต้นเนื่องด้วย อินเทอร์เน็ตสำหรับ BLOB และ CLOB ระหว่าง DB2 และโปรแกรม UDF เหมือนกัน: คือความยาวตามด้วยข้อมูล.

ตัวอย่าง: การค้นหาสตริงบน UDT:

สมมติว่า คุณพอใจกับฟังก์ชัน FINDSTRING จากการค้นหาสตริง BLOB, แต่ตอนนี้คุณได้นิยาม distinct type ชื่อ BOAT ด้วยชนิดต้นฉบับคือ BLOB.

และคุณต้องการให้ FINDSTRING จัดการกับค่าที่มีชนิดข้อมูลเป็น BOAT ได้, ดังนั้นคุณจึงสร้างฟังก์ชัน FINDSTRING มาอีกหนึ่งฟังก์ชัน. ฟังก์ชันนี้จะใช้ต้นฉบับของ FINDSTRING ที่จัดการกับค่า BLOB. โปรดสังเกตว่า มีการ overload ของ FINDSTRING ในตัวอย่างนี้:

```
CREATE FUNCTION FINDSTRING (BOAT, VARCHAR(200))
  RETURNS INT
  SPECIFIC "slick_fboat"
  SOURCE SPECIFIC FINDSTRING_BLOB
```

โปรดสังเกตว่าฟังก์ชัน FINDSTRING นี้จะมี Signature ต่างจากฟังก์ชัน FINDSTRING ใน “ตัวอย่าง: การค้นหาสตริง BLOB” ในหน้า 185, ดังนั้นจึงไม่มีปัญหาในการ Overloading ของชื่อฟังก์ชัน. เนื่องจากคุณใช้อนุประโยค SOURCE, ดังนั้นคุณไม่สามารถใช้อนุประโยค EXTERNAL NAME หรือคีย์เวิร์ดอื่นที่ใช้ระบุฟังก์ชันแอตทริบิวต์ได้. แอตทริบิวต์เหล่านี้จะนำมาจากฟังก์ชันต้นฉบับ. สุดท้าย, สังเกตว่าในการระบุฟังก์ชันต้นฉบับนั้นก็คือคุณกำลังใช้ชื่อฟังก์ชันเฉพาะโดยตรง ซึ่งฟังก์ชันเหล่านี้จัดเตรียมไว้ใน “ตัวอย่าง: การค้นหาสตริง BLOB” ในหน้า 185. เพราะว่าเป็นการอ้างอิงที่ไม่ครบตามเกณฑ์, Schema ที่ฟังก์ชันต้นฉบับนี้เก็บอยู่จะต้องอยู่ในฟังก์ชันพาธ, มิฉะนั้นการอ้างอิงนี้จะหาไม่เจอ.

สิ่งอ้างอิงที่เกี่ยวข้อง

“ตัวอย่าง: การค้นหาสตริง BLOB” ในหน้า 185

ในตัวอย่างนี้, คุณต้องการให้ฟังก์ชัน FINDSTRING ทำงานบน BLOB เช่นเดียวกับ CLOB. หากต้องการทำสิ่งนี้, คุณต้องกำหนด FINDSTRING อีกหนึ่งตัวให้นำ BLOB มาเป็นพารามิเตอร์แรก:

ตัวอย่าง: AVG บน UDT:

ตัวอย่างนี้จะใช้ฟังก์ชันคอลัมน์ AVG บน Distinct Type ชนิด CANADIAN_DOLLAR.

การเข้มงวดเรื่องชนิดจะช่วยป้องกันคุณจากการใช้ฟังก์ชันในตัว AVG บน Distinct Type ได้. ซึ่งกลายเป็นว่า ชนิดต้นฉบับของ CANADIAN_DOLLAR ก็คือ DECIMAL, ดังนั้นคุณจึงสร้าง AVG โดยใช้ต้นฉบับของฟังก์ชันในตัว AVG(DECIMAL).

```
CREATE FUNCTION AVG (CANADIAN_DOLLAR)
  RETURNS CANADIAN_DOLLAR
  SOURCE "QSYS2".AVG(DECIMAL(9,2))
```

โปรดสังเกตว่า SOURCE clause ต้องใช้ชื่อฟังก์ชันที่ครบตามเกณฑ์, เพื่อในกรณีที่มีฟังก์ชัน AVG อื่นแฝงอยู่ใน SQL พาธของคุณ.

ตัวอย่าง: การนับ:

ฟังก์ชันนับจำนวนอย่างง่ายของคุณจะคืนค่า 1 ในครั้งแรกที่เรียกใช้และจะเพิ่มผลลัพธ์ทีละหนึ่งในแต่ละครั้งที่เรียกใช้. ฟังก์ชันนี้ไม่ได้รับ SQL อากิวเมนต์, และโดยนิยามแล้ว ฟังก์ชันนี้คือฟังก์ชันแบบ NOT DETERMINISTIC เนื่องจากผลลัพธ์จะเปลี่ยนไปในการเรียกใช้แต่ละครั้ง.

มันจะใช้ SCRATCHPAD เพื่อบันทึกค่าที่คืนค่ามาล่าสุด. แต่ละครั้งที่ถูกเรียก ฟังก์ชันจะเพิ่มค่านี้อีกและคืนค่านี้ไป.

```
CREATE FUNCTION COUNTER ()
  RETURNS INT
  EXTERNAL NAME 'MYLIB/MYFUNCS(CTR)'  
  LANGUAGE C  
  PARAMETER STYLE DB2SQL  
  NO SQL  
  NOT DETERMINISTIC  
  NOT FENCED  
  SCRATCHPAD 4  
  DISALLOW PARALLEL
```

โปรดสังเกตว่าไม่จำเป็นต้องกำหนดพารามิเตอร์, มีแค่วงเล็บว่าง. ฟังก์ชันด้านบนนี้ระบุ SCRATCHPAD และใช้ค่าดีฟอลต์ของ NO FINAL CALL. ในกรณีนี้, ขนาดของกระดาษทดจะถูกตั้งค่าให้เป็นขนาด 4 ไบต์เท่านั้น, ซึ่งก็เพียงพอแล้วสำหรับการนับ. เนื่องจากฟังก์ชัน COUNTER ต้องการใช้นั่งกระดาษทดเท่านั้น ในการทำงานให้ถูกต้อง, DISALLOW PARALLEL จึงถูกเพิ่มเข้าไปเพื่อป้องกัน DB2 จากการทำงานแบบขนาน.

ตัวอย่าง: ฟังก์ชันแบบ Table ที่คืนค่า document ID:

ในตัวอย่างนี้, คุณได้เขียนฟังก์ชันแบบ Table ที่คืนค่าแถวที่ประกอบด้วยคอลัมน์เดียวที่มี document identifier สำหรับแต่ละเอกสารในระบบการจัดการข้อความของคุณ โดยเอกสารนั้นจะตรงกับพื้นที่หัวข้อที่ให้มา (พารามิเตอร์แรก) และมีสตริงที่ให้มา (พารามิเตอร์ที่สอง).

UDF นี้จะใช้ฟังก์ชันของระบบจัดการเอกสารเพื่อระบุเอกสารได้อย่างรวดเร็ว:

```
CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
  RETURNS TABLE (DOC_ID CHAR(16))
  EXTERNAL NAME 'DOCFUNCS/UDFMATCH(udfmatch)'  
  LANGUAGE C  
  PARAMETER STYLE DB2SQL  
  NO SQL  
  DETERMINISTIC  
  NO EXTERNAL ACTION  
  NOT FENCED  
  SCRATCHPAD  
  NO FINAL CALL  
  DISALLOW PARALLEL  
  CARDINALITY 20
```

ภายในบริบทของเซสชันเดียวแล้ว ฟังก์ชันนี้จะคืนค่าตารางเดียวกันเสมอ, ดังนั้นมันจึงถูกกำหนดให้เป็นแบบ DETERMINISTIC. RETURNS clause กำหนดเอาต์พุตจาก DOCMATCH, รวมถึงชื่อคอลัมน์ DOC_ID. FINAL CALL ไม่จำเป็นต้องระบุสำหรับฟังก์ชัน Table นี้. คีย์เวิร์ด DISALLOW PARALLEL จำเป็นเนื่องจากฟังก์ชันตารางไม่สามารถทำงานแบบขนานได้. ถึงแม้ว่าขนาดของเอาต์พุตจาก DOCMATCH จะเป็นตารางขนาดใหญ่, แต่ค่า CARDINALITY 20 จะเป็นค่าแทนที่, และจะถูกระบุเพื่อช่วยให้ตัว optimizer ตัดสินใจได้ดีขึ้น.

โดยทั่วไป, ฟังก์ชันแบบ Table นี้อาจจะถูกใช้ในการเชื่อมโยงกับตารางที่เก็บข้อความเอกสาร, ดังด้านล่างนี้:

```

SELECT T.AUTHOR, T.DOCTEXT
FROM DOCS AS T, TABLE(DOCMATCH('MATHEMATICS', 'ZORN'S LEMMA')) AS F
WHERE T.DOCID = F.DOC_ID

```

โปรดสังเกตไวยากรณ์พิเศษ (คีย์เวิร์ด TABLE) สำหรับระบุฟังก์ชันแบบ Table ในอนุประโยค FROM. ในการพยายามนี้, ฟังก์ชันแบบ Table ที่ชื่อ DOCMATCH() จะคืนค่าแถวที่เก็บคอลัมน์ DOC_ID สำหรับแต่ละเอกสาร MATHEMATICS ที่อ้างอิงไปยัง ZORN'S LEMMA. ค่า DOC_ID จะถูกเชื่อมโยงกับตารางเอกสารหลัก, และใช้ตั้งชื่อผู้แต่งและข้อความเอกสาร.

การส่งผ่านอากิวเมนต์จาก DB2 ไปยังฟังก์ชันภายนอก:

DB2 ได้จัดเตรียมหน่วยเก็บสำหรับพารามิเตอร์ทุกตัวที่ส่งผ่านไปยัง UDF. ดังนั้น, พารามิเตอร์จะถูกส่งผ่านไปยังฟังก์ชันแบบภายนอกด้วยแอดเดรส.

นี่คือวิธีการส่งผ่านพารามิเตอร์แบบปกติสำหรับโปรแกรม. สำหรับเซอร์วิสโปรแกรม, โปรดตรวจสอบให้แน่ใจว่าพารามิเตอร์ถูกกำหนดไว้อย่างถูกต้องในฟังก์ชันโค้ด.

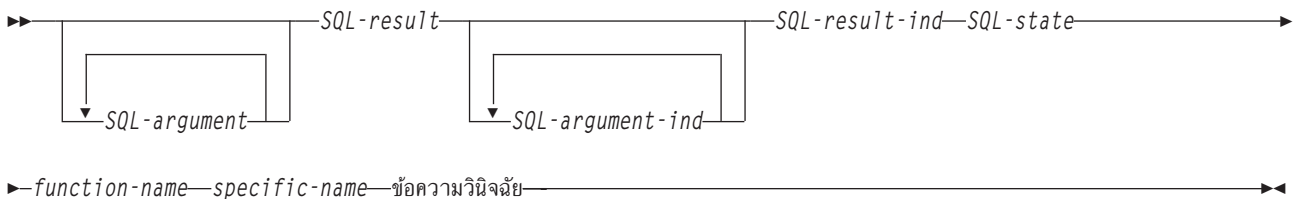
เมื่อกำหนดและใช้งานพารามิเตอร์ใน UDF แล้ว, ควรดูแลเพื่อให้แน่ใจว่าไม่มีการอ้างอิงถึงหน่วยเก็บสำหรับพารามิเตอร์ที่กำหนดให้มากกว่าที่ถูกกำหนดให้สำหรับพารามิเตอร์นั้น. พารามิเตอร์ถูกเก็บไว้ทั้งหมดในเนื้อที่เดียวกันและการใช้พื้นที่หน่วยเก็บของพารามิเตอร์เกินที่กำหนดให้จะบันทึกทับค่าของพารามิเตอร์อื่น. ในทางกลับกัน, วิธีนี้, สามารถทำให้ฟังก์ชันนี้ได้เห็นข้อมูลอินพุตที่ไม่ถูกต้องหรือทำให้ค่าถูกส่งคืนไปยังฐานข้อมูลที่ไม่ถูกต้อง.

พารามิเตอร์ที่ได้รับการสนับสนุนซึ่งใช้ได้กับ UDFs แบบภายนอกมีอยู่หลายลักษณะด้วยกัน. ส่วนใหญ่, ลักษณะที่ต่างกัน คือจำนวนพารามิเตอร์ที่ถูกส่งผ่านไปยังโปรแกรมภายนอกหรือเซอร์วิสโปรแกรม.

รูปแบบพารามิเตอร์ SQL:

ลักษณะพารามิเตอร์ SQL มีลักษณะตรงตามมาตรฐานอุตสาหกรรมของ Structured Query Language (SQL). ลักษณะพารามิเตอร์นี้สามารถใช้กับ scalar UDF ได้เท่านั้น.

ด้วยลักษณะพารามิเตอร์ SQL, พารามิเตอร์จะถูกส่งผ่านไปยังโปรแกรมภายนอกดังนี้ (ตามลำดับที่ระบุ):



SQL-argument

อากิวเมนต์นี้ ถูกกำหนดโดย DB2 ก่อนการเรียกใช้ UDF. ค่านี้จะทำซ้ำ n ครั้ง, โดยที่ค่า n เป็นจำนวนของ อากิวเมนต์ที่ระบุอยู่ในการอ้างอิงฟังก์ชัน. ค่าของแต่ละอากิวเมนต์เหล่านี้จะถูกนำมาจากนิพจน์ที่ระบุไว้ในการเรียกฟังก์ชันทำงาน. ค่าจะถูกแสดงออกในประเภทข้อมูลของพารามิเตอร์ที่กำหนดไว้ในคำสั่งฟังก์ชันการสร้าง. หมายเหตุ: พารามิเตอร์เหล่านี้จะนำไปใช้เป็นอินพุตเท่านั้น; การเปลี่ยนค่าพารามิเตอร์ใดๆ ที่กระทำโดย UDF จะละลายหายไปโดย DB2.

SQL-result

อักขระเมตริกนี้ถูกเซตค่าโดย UDF ก่อนการส่งกลับไปยัง DB2. ฐานข้อมูลมีหน่วยเก็บสำหรับค่าส่งคืน. เนื่องจากพารามิเตอร์ถูกส่งผ่านโดยแอดเดรส, แอดเดรสจึงเป็นหน่วยเก็บค่าส่งคืน. ฐานข้อมูลจะมีหน่วยเก็บมากเท่าที่จำเป็นสำหรับค่าส่งคืนตามที่กำหนดไว้บนคำสั่ง CREATE FUNCTION. ถ้าหากมีการใช้ประโยค CAST FROM ในข้อความ CREATE FUNCTION, DB2 ให้สันนิษฐานไว้ก่อนว่า UDF ได้ส่งกลับค่าที่ได้กำหนดไว้ในประโยค CAST FROM, หรือมิฉะนั้น DB2 ให้สันนิษฐานว่า UDF ส่งกลับค่าที่ได้กำหนดไว้ใน ประโยค RETURNS แทน.

SQL-argument-ind

อักขระเมตริกนี้ ถูกกำหนดโดย DB2 ก่อนการเรียกใช้ UDF. อักขระเมตริกสามารถถูกใช้งานโดย UDF เพื่อกำหนดว่า SQL-argument ที่ต้องการเป็น Null หรือไม่. The *nth SQL-argument-ind* corresponds to the *nth SQL-argument*, described previously. ตัวบ่งชี้แต่ละตัวถูกกำหนดเป็นจำนวนเต็มขนาดสองไบต์. ตัวบ่งชี้จะถูกกำหนดให้มีค่าใดค่าหนึ่งต่อไปนี้:

0 มีอักขระเมตริกอยู่และไม่เป็นศูนย์.

-1 อักขระเมตริกเป็นศูนย์.

หากฟังก์ชันถูกกำหนดด้วย RETURNS NULL ON NULL INPUT, UDF จะไม่ต้องตรวจสอบเพื่อหาค่าที่เป็น null. อย่างไรก็ตาม, หากฟังก์ชันถูกกำหนดด้วย CALLS ON NULL INPUT, อักขระเมตริกใดๆ สามารถเป็น NULL และ UDF ควรตรวจสอบเพื่อหาอินพุตที่เป็น null. หมายเหตุ: พารามิเตอร์เหล่านี้จะนำไปใช้เป็นอินพุตเท่านั้น; การเปลี่ยนค่าพารามิเตอร์ใดๆ ที่กระทำโดย UDF จะละลายเข้าไปโดย DB2.

SQL-result-ind

อักขระเมตริกนี้ถูกเซตค่าโดย UDF ก่อนการส่งกลับไปยัง DB2. ฐานข้อมูลมีหน่วยเก็บสำหรับค่าส่งคืน. อักขระเมตริกกำหนดไว้เป็นจำนวนเต็มขนาดสองไบต์. หากกำหนดให้เป็นค่าลบ, ฐานข้อมูลจะตีความผลของฟังก์ชันเป็นค่าศูนย์. หากกำหนดค่าให้เป็น null หรือค่าบวก, ฐานข้อมูลจะใช้งานค่าที่ถูกส่งคืนใน SQL-result. ฐานข้อมูล ได้จัดเตรียมหน่วยเก็บสำหรับตัวบ่งชี้ค่าส่งคืน. เนื่องจากพารามิเตอร์ถูกส่งผ่านโดยแอดเดรส, แอดเดรสจึงเป็นหน่วยเก็บค่าตัวบ่งชี้.

SQL-state

อักขระเมตริก คือค่า CHAR(5) ซึ่งแทนค่า SQLSTATE.

พารามิเตอร์นี้จะถูกส่งผ่านในรูปของฐานข้อมูลที่ตั้งค่าเป็น '00000' และสามารถตั้งค่าโดยฟังก์ชันให้เป็นสถานะผลลัพธ์ของฟังก์ชัน. ขณะที่ปกติแล้ว SQLSTATE ไม่ได้ถูกกำหนดค่าโดยฟังก์ชัน, แต่สามารถใช้งาน SQLSTATE เพื่อส่งสัญญาณข้อผิดพลาดหรือแจ้งเตือนไปยังฐานข้อมูลได้ดังต่อไปนี้:

01Hxx ฟังก์ชันได้ตรวจพบการแจ้งเตือน. การตรวจพบนี้ก่อให้เกิดการแจ้งเตือนแบบ SQL, ซึ่ง xx ในที่นี้อาจเป็นหนึ่งในหลายๆ สตริงที่อาจพบได้.

38xxx ฟังก์ชันได้ตรวจพบข้อผิดพลาด. การตรวจพบนี้ก่อให้เกิดข้อผิดพลาด SQL. ซึ่ง xxx ในที่นี้อาจเป็นหนึ่ง ในหลายๆ สตริงที่อาจพบได้.

function-name

อักขระเมตริกนี้ ถูกกำหนดโดย DB2 ก่อนการเรียกใช้ UDF. มันคือค่า VARCHAR(139) ที่ประกอบด้วยชื่อของฟังก์ชันที่ถูกเรียกใช้งานเสมือนเป็น ฟังก์ชันโค้ด.

รูปแบบของชื่อฟังก์ชันที่ถูกส่งผ่านคือ:

<schema-name>.<function-name>

พารามิเตอร์นี้มีประโยชน์เมื่อฟังก์ชันโค้ดถูกใช้งานโดย definition จำนวนมากของ UDF ดังนั้นโค้ดจะสามารถแยกแยะได้ว่า definition ไหนถูกเรียกใช้งาน. หมายเหตุ: พารามิเตอร์นี้จะถูกใช้เป็นเพียงอินพุตเท่านั้น; การเปลี่ยนค่าพารามิเตอร์ใดๆ ที่กระทำโดย UDF จะถูกละเลยเข้าไปโดย DB2.

specific-name

อักขรเมตต์นี้ ถูกกำหนดโดย DB2 ก่อนการเรียกใช้ UDF. มันคือค่า VARCHAR(128) ที่ประกอบด้วยชื่อของฟังก์ชันที่ถูกเรียกใช้งานเสมือนเป็น ฟังก์ชันโค้ด.

เช่นเดียวกับชื่อฟังก์ชัน, พารามิเตอร์นี้มีประโยชน์ เมื่อฟังก์ชันโค้ดถูกใช้งานโดย definition จำนวนมากของ UDF เพื่อที่โค้ดจะสามารถแยกแยะได้ว่า definition ไหนจะถูกเรียกใช้งาน. หมายเหตุ: พารามิเตอร์นี้จะถูกใช้เป็นเพียงอินพุตเท่านั้น; การเปลี่ยนค่าพารามิเตอร์ใดๆ ที่กระทำโดย UDF จะถูกละเลยเข้าไปโดย DB2.

diagnostic-message

อักขรเมตต์นี้ ถูกกำหนดโดย DB2 ก่อนการเรียกใช้ UDF. ค่าของอักขรเมตต์คือค่า VARCHAR(70) ซึ่งสามารถถูกใช้งานโดย UDF เพื่อส่งข้อความกลับเมื่อ UDF แจ้งการเตือนและข้อผิดพลาดของ SQLSTATE.

อักขรเมตต์จะถูก initialize โดยฐานข้อมูลบนอินพุตไปยัง UDF และอาจถูกกำหนดค่าโดย UDF โดยมีข้อมูลอธิบายรายละเอียด. ข้อความ Message จะถูกละเลยไปโดย DB2 เว้นแต่พารามิเตอร์ SQL-state ถูกกำหนดโดย UDF.

ข้อมูลที่เกี่ยวข้อง

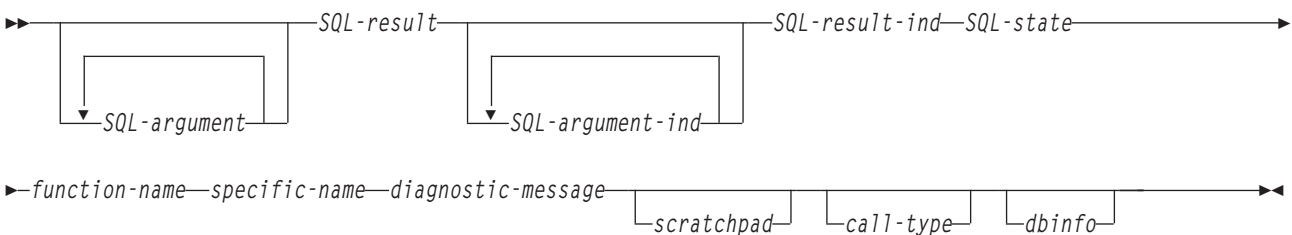
ข้อความและโค้ด SQL

รูปแบบพารามิเตอร์ DB2SQL:

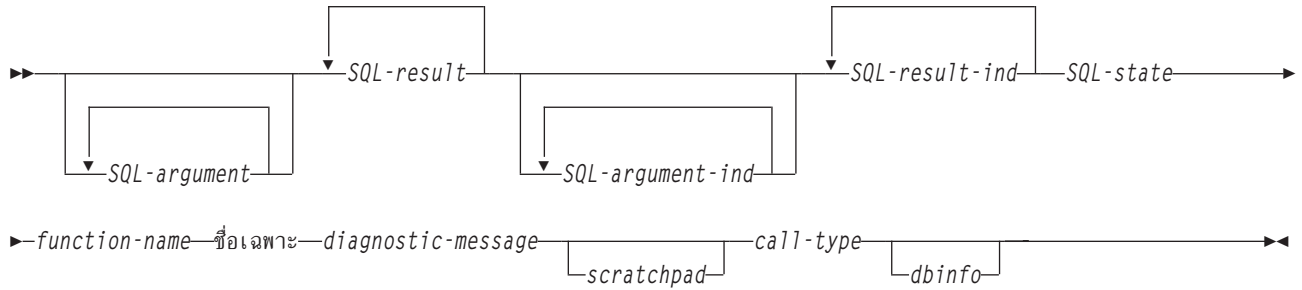
ด้วยลักษณะพารามิเตอร์ DB2SQL, พารามิเตอร์เดียวกันและลำดับพารามิเตอร์เดียวกันจะถูกส่งผ่านไปยังโปรแกรมภายนอกหรือเซอวิสโปรแกรมอย่างที่ถูกส่งผ่านไปสำหรับลักษณะพารามิเตอร์ SQL. อย่างไรก็ตาม, DB2SQL อนุญาตให้ส่งผ่านพารามิเตอร์ตัวเลือกเสริมได้เช่นกัน.

หากมีการระบุพารามิเตอร์ตัวเลือกเสริมมากกว่าหนึ่งใน definition UDF, พารามิเตอร์เหล่านั้นจะถูกส่งผ่านไปยัง UDF ตามลำดับที่กำหนดไว้ด้านล่าง. โปรดศึกษาจากลักษณะพารามิเตอร์ SQL เพื่อดูพารามิเตอร์ทั่วไป. ลักษณะพารามิเตอร์สามารถใช้งานได้กับ UDF แบบสเกลาร์และตาราง.

สำหรับฟังก์ชันสเกลาร์:



สำหรับฟังก์ชันตาราง:



scratchpad

อักขระเหล่านี้ ถูกกำหนดโดย DB2 ก่อนการเรียกใช้ UDF. อักขระเหล่านี้จะถูกแสดงก็ต่อเมื่อคำสั่ง CREATE FUNCTION สำหรับ UDF ระบุคีย์เวิร์ด SCRATCHPAD. อักขระเหล่านี้ คือโครงสร้างซึ่งมีส่วนประกอบต่อไปนี้:

- INTEGER แสดงความยาวของ scratchpad.
- scratchpad ที่แท้จริง, จะถูกเตรียมข้อมูลเบื้องต้นไปยังทุกไบทาร์รี่ 0 โดย DB2 ก่อนการเรียกใช้งาน UDF ในครั้งแรก.

UDF สามารถใช้งาน scratchpad ให้เป็นทั้งหน่วยเก็บใช้งาน หรือหน่วยเก็บถาวร, เนื่องจาก scratchpad จะถูกเก็บไว้เมื่อมีการเรียก UDF ทำงาน.

สำหรับฟังก์ชันตาราง, scratchpad จะถูก เตรียมข้อมูลเบื้องต้นตามที่กล่าวไว้ข้างต้นก่อนการเรียกทำงาน FIRST ไปยัง UDF หากมีการระบุ FINAL CALL ไว้บน CREATE FUNCTION. หลังจากการเรียกนี้แล้ว, เนื้อหา scratchpad จะอยู่ภายใต้การควบคุมของฟังก์ชันตารางทั้งหมด. DB2 ไม่ได้ทดสอบ หรือ เปลี่ยนเนื้อหาของ scratchpad หลังจากนั้น. scratchpad จะถูกส่งผ่านไปยังฟังก์ชันในการเรียกทำงานแต่ละครั้ง. ฟังก์ชันสามารถถูกป้อนกลับเข้าไปใหม่, และ DB2 จะเก็บเตรียมข้อมูลสถานะของตัวเองไว้ใน scratchpad.

หากมีการระบุ NO FINAL CALL หรือถูกกำหนดเป็นค่าดีฟอลต์สำหรับฟังก์ชันตาราง, scratchpad จะถูก initialize ตามที่กล่าวไว้ด้านบนสำหรับการเรียก OPEN แต่ละครั้ง, และเนื้อหา scratchpad จะอยู่ภายใต้การควบคุมของฟังก์ชันตารางทั้งหมดระหว่างการเรียก OPEN. นี่เป็นเรื่องสำคัญมากสำหรับฟังก์ชันตารางที่ใช้งานร่วมกันหรือในการสืบค้นย่อย. หากจำเป็นต้องรักษาเนื้อหาของ scratchpad ทุกครั้งที่มีการเรียก OPEN, คุณต้องระบุ FINAL CALL ในคำสั่ง CREATE FUNCTION ของคุณ. เมื่อระบุ FINAL CALL, เพิ่มเติมนอกเหนือจากการเรียก OPEN, FETCH, และ CLOSE, ฟังก์ชันตารางจะได้รับการเรียกแบบ FIRST และ FINAL เช่นกัน, เพื่อรักษา scratchpad และการปล่อยรีซอร์ส.

call-type

อักขระเหล่านี้ ถูกกำหนดโดย DB2 ก่อนการเรียกใช้ UDF. สำหรับฟังก์ชันแบบสเกลาร์, อักขระเหล่านี้จะถูกแสดงก็ต่อเมื่อคำสั่ง CREATE FUNCTION สำหรับ UDF ระบุคีย์เวิร์ด FINAL. อย่างไรก็ตาม, ฟังก์ชันตาราง อักขระเหล่านี้จะแสดงอยู่เสมอ. อักขระเหล่านี้จะอยู่ตามหลังอักขระ scratchpad; หรืออาร์กิวเมนต์ ข้อความวินิจฉัย หากอาร์กิวเมนต์ scratchpad ไม่ปรากฏขึ้นมา. อักขระเหล่านี้จะใช้รูปแบบของค่า INTEGER.

สำหรับฟังก์ชันแบบสเกลาร์:

- 1 นี่คือการเรียกครั้งแรกให้กับ UDF สำหรับคำสั่งนี้. การเรียกครั้งแรกคือการเรียกแบบปกติในค่าอักขระ มนต์ SQL ทั้งหมดที่ถูกส่งผ่าน.
- 0 นี่คือการเรียกแบบปกติ. (ค่าอักขระอื่นที่นอกเหนือจากปกติทั้งหมดจะถูกส่งผ่าน).

- 1 นี่คือการเรียกครั้งสุดท้าย. ไม่มีการส่งผ่านค่า *SQL-argument* หรือค่า *SQL-argument-ind*. UDF ไม่ควรส่งคืนคำตอบใดๆ โดยใช้ *SQL-result*, อากิวเมนต์ *SQL-result-ind*, *SQL-state*, หรือ ข้อความวินิจฉัย. ระบบจะละเลยข้ามอากิวเมนต์เหล่านี้ เมื่อมีการส่งคืนมาจาก UDF.

สำหรับฟังก์ชันตาราง:

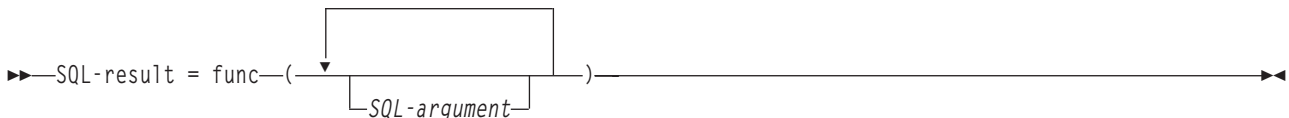
- 2 นี่คือการเรียกครั้งแรก ให้กับ UDF สำหรับคำสั่งนี้. การเรียกครั้งแรกคือการเรียกแบบปกติในค่าอากิวเมนต์ *SQL* ทั้งหมดที่ถูกส่งผ่าน.
- 1 นี่คือการเรียกแบบเปิด ไปยัง UDF สำหรับคำสั่งนี้. *scratchpad* จะถูก initialize หากไม่มีการระบุ *FINAL CALL*, แต่ไม่จำเป็นมากนัก. ค่าอากิวเมนต์ *SQL* ทั้งหมดจะถูกส่งผ่าน.
- 0 นี่คือการเรียกแบบดึงข้อมูลออก. DB2 คาดหวังว่าฟังก์ชันตารางจะส่งกลับ แถวซึ่งประกอบด้วยชุดของค่าส่งคืน, หรือ เงื่อนไขการสิ้นสุดตารางที่บ่งชี้โดย *SQLSTATE* ที่มีค่า '02000' อย่างไม่อย่างหนึ่ง.
- 1 นี่คือการเรียกแบบปิด. การเรียกแบบนี้จะสร้างสมดุลให้การเรียก *OPEN*, และสามารถถูกใช้ประมวลผล *CLOSE* แบบภายนอกและปล่อยรีซอร์ส.
- 2 นี่คือการเรียกครั้งสุดท้าย. ไม่มีการส่งผ่านค่า *SQL-argument* หรือค่า *SQL-argument-ind*. UDF ไม่ควรส่งคืนคำตอบใดๆ โดยใช้ *SQL-result*, อากิวเมนต์ *SQL-result-ind*, *SQL-state*, หรือ ข้อความวินิจฉัย. ระบบจะละเลยข้ามอากิวเมนต์เหล่านี้ เมื่อมีการส่งคืนมาจาก UDF.

dbinfo อากิวเมนต์นี้ ถูกกำหนดโดย DB2 ก่อนการเรียกใช้ UDF. อากิวเมนต์จะถูกแสดงก็ต่อเมื่อคำสั่ง *CREATE FUNCTION* สำหรับ UDF ระบุคีย์เวิร์ด *DBINFO*. อากิวเมนต์คือโครงสร้างที่มี *definition* อยู่ในการถอดแทรกคำสั่ง *sqludf*.

ลักษณะพารามิเตอร์ *GENERAL* (หรือ *SIMPLE CALL*):

ด้วยลักษณะพารามิเตอร์ *GENERAL*, พารามิเตอร์จะถูกส่งผ่านไปในเซอร์วิสโปรแกรมภายนอกเหมือนกันกับที่พารามิเตอร์เหล่านั้นถูกระบุไว้ในคำสั่ง *CREATE FUNCTION*. ลักษณะพารามิเตอร์นี้สามารถใช้กับ *scalar UDF* ได้เท่านั้น.

ฟอร์แมตคือ:



SQL-argument

อากิวเมนต์นี้ ถูกกำหนดโดย DB2 ก่อนการเรียกใช้ UDF. คำนี้จะทำซ้ำ *m* ครั้ง, โดยที่ค่า *m* เป็นจำนวนของ อากิวเมนต์ที่ระบุอยู่ในการอ้างอิงฟังก์ชัน. ค่าของแต่ละอากิวเมนต์เหล่านี้จะถูกนำมาจากนิพจน์ที่ระบุไว้ในการเรียกฟังก์ชันทำงาน. ค่าจะถูกแสดงออกในประเภทข้อมูลของพารามิเตอร์ที่กำหนดไว้ในคำสั่ง *CREATE FUNCTION*. หมายเหตุ: พารามิเตอร์เหล่านี้จะนำไปใช้เป็นอินพุตเท่านั้น; การเปลี่ยนค่าพารามิเตอร์ใดๆ ที่กระทำโดย UDF จะละเลยข้ามไปโดย DB2.

SQL-result

คำนี้จะถูกส่งกลับโดย UDF. DB2 ก็อปปีค่าลงในหน่วยเก็บฐานข้อมูล. เพื่อจะส่งคืนค่าอย่างถูกต้อง, ฟังก์ชันโค้ดต้องเป็นฟังก์ชันการส่งคืนค่า. ฐานข้อมูลจะคัดลอกเฉพาะค่าที่กำหนดไว้ให้มากที่สุดสำหรับคำสั่งคืนซึ่งระบุไว้บนคำสั่ง *CREATE FUNCTION*. ถ้าหากมีการใช้ประโยค *CAST FROM* ในข้อความ *CREATE FUNCTION*, DB2 ให้

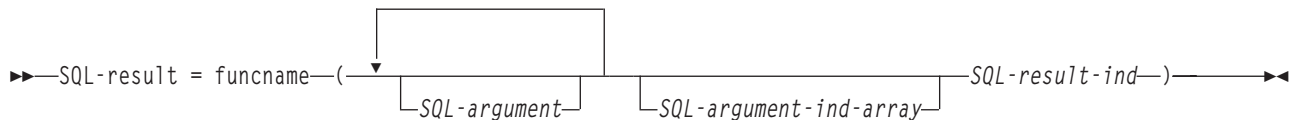
สันนิษฐานไว้ก่อนว่า UDF ได้ส่งกลับค่าที่ได้กำหนดไว้ในประโยค CAST FROM , หรือมีฉะนั้น DB2 ให้สันนิษฐานว่า UDF ส่งกลับค่าที่ได้กำหนดไว้ใน ประโยค RETURNS แทน.

เนื่องจากข้อกำหนดที่ว่าฟังก์ชันโค้ดต้องเป็นฟังก์ชันการส่งคืนค่า, คุณต้องสร้างฟังก์ชันโค้ดใดๆ ให้กับลักษณะพารามิเตอร์ GENERAL ลงในเซอร์วิสโปรแกรม.

รูปแบบพารามิเตอร์ GENERAL WITH NULLS:

รูปแบบพารามิเตอร์ GENERAL WITH NULLS จะใช้งานได้เฉพาะกับ UDF แบบสเกลาร์.

ด้วยลักษณะพารามิเตอร์, พารามิเตอร์จะถูกส่งผ่านไปยังเซอร์วิสโปรแกรมดังต่อไปนี้ (ตามลำดับที่ระบุไว้):



SQL-argument

อาร์กิวเมนต์นี้ ถูกกำหนดโดย DB2 ก่อนการเรียกใช้ UDF. ค่านี้จะทำซ้ำ n ครั้ง, โดยที่ค่า n เป็นจำนวนของ อาร์กิวเมนต์ที่ระบุอยู่ในการอ้างอิงฟังก์ชัน. ค่าของแต่ละอาร์กิวเมนต์เหล่านี้จะถูกนำมาจากนิพจน์ที่ระบุไว้ในการเรียกฟังก์ชันทำงาน. ค่าจะถูกแสดงออกในประเภทข้อมูลของพารามิเตอร์ที่กำหนดไว้ในคำสั่ง CREATE FUNCTION.หมายเหตุ: พารามิเตอร์เหล่านี้จะนำไปใช้เป็นอินพุตเท่านั้น; การเปลี่ยนค่าพารามิเตอร์ใดๆ ที่กระทำโดย UDF จะละลายข้ามไปโดย DB2.

SQL-argument-ind-array

อาร์กิวเมนต์นี้ ถูกกำหนดโดย DB2 ก่อนการเรียกใช้ UDF. อาร์กิวเมนต์สามารถถูกใช้งานโดย UDF เพื่อกำหนดว่า SQL-arguments ตั้งแต่หนึ่งรายการขึ้นไปมีค่าเป็น null หรือไม่. อาร์กิวเมนต์คือ array ของจำนวนเต็มขนาดสองไบต์ (ตัวบ่งชี้). อาร์กิวเมนต์ array ลำดับที่ n เชื่อมโยงกับ SQL-argument ลำดับที่ n . แต่ละ array entry ถูกตั้งค่าให้เป็นค่าใดค่าหนึ่งต่อไปนี้:

- 0 มีอาร์กิวเมนต์อยู่และไม่เป็นศูนย์.
- 1 อาร์กิวเมนต์เป็นศูนย์.

UDF ควรตรวจสอบอินพุตที่มีค่า null. หมายเหตุ: พารามิเตอร์นี้จะถูกใช้เป็นเพียงอินพุตเท่านั้น; การเปลี่ยนค่าพารามิเตอร์ที่กระทำโดย UDF จะถูกละเลยโดย DB2.

SQL-result-ind

อาร์กิวเมนต์นี้ถูกเซตค่าโดย UDF ก่อนการส่งกลับไปยัง DB2. ฐานข้อมูลมีหน่วยเก็บสำหรับค่าส่งคืน. อาร์กิวเมนต์ถูกกำหนดไว้เป็นจำนวนเต็มขนาดสองไบต์. หากกำหนดให้เป็นค่าลบ, ฐานข้อมูลจะตีความผลของฟังก์ชันเป็นค่าศูนย์. หากกำหนดค่าให้เป็น null หรือค่าบวก, ฐานข้อมูลจะใช้งานค่าที่ถูกส่งคืนใน SQL-result. ฐานข้อมูล ได้จัดเตรียมหน่วยเก็บสำหรับตัวบ่งชี้ค่าส่งคืน. เนื่องจากพารามิเตอร์ถูกส่งผ่านโดยแอดเดรส, แอดเดรสจึงเป็นหน่วยเก็บค่าตัวบ่งชี้.

SQL-result

ค่านี้จะถูกส่งกลับโดย UDF. DB2 ก็อปปีค่าลงในหน่วยเก็บฐานข้อมูล. เพื่อจะส่งคืนค่าอย่างถูกต้อง, ฟังก์ชันโค้ดต้องเป็นฟังก์ชันการส่งคืนค่า. ฐานข้อมูลจะคัดลอกเฉพาะค่าที่กำหนดไว้ให้มากที่สุดสำหรับค่าส่งคืนซึ่งระบุไว้บนค่า

สั่ง CREATE FUNCTION . ถ้าหากมีการใช้ CAST FROM clause ในคำสั่ง CREATE FUNCTION , DB2 ให้สันนิษฐานไว้ก่อนว่า UDF ได้ส่งกลับค่าที่ได้กำหนดไว้ใน CAST FROM clause, หรือมิฉะนั้น DB2 ให้สันนิษฐานว่า UDF ส่งกลับค่าที่ได้กำหนดไว้ใน RETURNS clause แทน.

เนื่องจากข้อกำหนดที่ว่า ฟังก์ชันโค้ด คือฟังก์ชันการส่งคืนค่า, ฟังก์ชันใดๆ ที่ใช้สำหรับรูปแบบพารามิเตอร์ GENERAL WITH NULLS ต้องถูกสร้างขึ้นภายในเซอร์วิสโปรแกรม.

หมายเหตุ:

1. คุณสามารถระบุชื่อภายนอกที่ระบุไว้บนคำสั่ง CREATE FUNCTION ได้ด้วยการใส่เครื่องหมายอัญประกาศ หรือไม่ใส่เครื่องหมายอัญประกาศ. หากชื่อไม่มีเครื่องหมายอัญประกาศ, ชื่อจะถูกทำเป็นอักษรตัวพิมพ์ใหญ่ก่อนที่จะถูกเก็บไว้; หากมีเครื่องหมายอัญประกาศ, ชื่อจะถูกเก็บไว้ตามที่ระบุ. เรื่องนี้สำคัญเมื่อตั้งชื่อโปรแกรม, เพราะฐานข้อมูลจะค้นหาโปรแกรมที่มีชื่อตรงกันกับชื่อที่เก็บไว้ด้วย definition ฟังก์ชัน. ตัวอย่างเช่น, หากฟังก์ชันถูกสร้างขึ้นเป็น:

```
CREATE FUNCTION X(INT) RETURNS INT
LANGUAGE C
EXTERNAL NAME 'MYLIB/MYPMG(MYENTRY)'
```

แลชอร์สสำหรับโปรแกรมคือ:

```
void myentry(
                int*in
                int*out,
                .
                .
                . .
```

ฐานข้อมูลจะไม่พบ entry เพราะ entry จะเป็นตัวอักษรตัวพิมพ์เล็ก *myentry* และฐานข้อมูลจะถูกสร้างขึ้นเพื่อดูตัวอักษรตัวพิมพ์ใหญ่ *MYENTRY*.

2. สำหรับเซอร์วิสโปรแกรมและโมดูล C++, โปรดแน่ใจว่าในซอร์สโค้ด C++ อยู่ก่อนหน้า definition ฟังก์ชันโปรแกรมโดยมี *extern "C"*. มิฉะนั้น, คอมไพเลอร์ C++ จะดำเนินการ 'name mangling' ของชื่อฟังก์ชันและฐานข้อมูลจะไม่พบชื่อนั้น.

รูปแบบพารามิเตอร์ DB2GENERAL:

พารามิเตอร์รูปแบบ DB2GENERAL ถูกใช้โดย Java UDFs.

ข้อมูลที่เกี่ยวข้อง

Java SQL Routines

รูปแบบพารามิเตอร์ Java:

รูปแบบพารามิเตอร์ Java เป็นรูปแบบที่กำหนดโดย SQLJ Part 1: มาตรฐาน SQL Routines.

ข้อมูลที่เกี่ยวข้อง

Java SQL Routines

ข้อควรพิจารณาฟังก์ชันแบบ Table:

ฟังก์ชันตารางภายนอก คือ UDF ที่ส่งตารางไปยัง SQL ที่ถูกอ้างถึง. การอ้างถึงฟังก์ชันตารางจะสามารถใช้งานได้เฉพาะใน FROM clause ของคำสั่ง SELECT.

เมื่อใช้งานฟังก์ชันตาราง, โปรดสังเกตดังต่อไปนี้:

- ถึงแม้ว่าฟังก์ชันตารางจะส่งค่าตาราง, อินเตอร์เฟซฟิลิเตอร์ ระหว่าง DB2 และ UDF จะเป็นแบบ ครั้งละหนึ่งแถว. การเรียกไปยังฟังก์ชันตารางมี 5 ประเภท: OPEN, FETCH, CLOSE, FIRST, และ FINAL. การเรียกแบบ FIRST และ FINAL ขึ้นอยู่กับวิธีที่คุณกำหนด UDF. ระบบ *ประเภทการเรียก* ที่สามารถใช้กับฟังก์ชันแบบสเกลาร์จะถูกใช้งานเพื่อแยกแยะการเรียกเหล่านี้.
- มาตรฐานอินเตอร์เฟซที่ใช้ระหว่าง DB2 และ ฟังก์ชัน scalar ที่กำหนดโดยผู้ใช้จะถูกขยายออก เพื่อรองรับฟังก์ชันตาราง. อากิวเมนต์ *SQL-result* จะทำงานซ้ำสำหรับฟังก์ชันตาราง; ซึ่งก็คือแต่ละ instance ที่เชื่อมโยงกับคอลัมน์ที่จะถูกส่งคืนตามที่กำหนดใน RETURNS TABLE ของคำสั่ง CREATE FUNCTION. อากิวเมนต์ *SQL-result-idx* จะทำซ้ำ, แต่ละ instance ที่เชื่อมโยงกับ instance *SQL-result* ที่เกี่ยวข้อง.
- คอลัมน์ผลลัพธ์บางคอลัมน์ที่กำหนดใน RETURNS clause ของคำสั่ง CREATE FUNCTION สำหรับฟังก์ชันตารางเท่านั้นที่จะถูกส่งคืน. คีย์เวิร์ด DBINFO ของ CREATE FUNCTION, และอากิวเมนต์ *dbinfo* ที่เกี่ยวข้องเปิดใช้งาน optimization ที่คอลัมน์เหล่านั้นต้องการสำหรับการอ้างอิงของฟังก์ชันตารางที่ต้องส่งคืน.
- ค่าคอลัมน์แต่ละคอลัมน์ที่ถูกส่งคืนสอดคล้องกับฟอร์แมตของค่าที่ถูกส่งคืนจากฟังก์ชันแบบสเกลาร์.
- คำสั่ง CREATE FUNCTION สำหรับฟังก์ชันตารางมีค่ากำหนด CARDINALITY *n*. ข้อกำหนดคุณสมบัตินี้จะเปิดทางให้ definer ได้แจ้งให้ DB2 optimizer ถึงขนาดโดยประมาณของผลลัพธ์ เพื่อว่า optimizer สามารถทำการตัดสินใจได้ดีขึ้นเมื่อมีการอ้างถึงฟังก์ชัน. โดยไม่ต้องคำนึงว่ามีการระบุอะไรเป็น CARDINALITY ของฟังก์ชันตาราง, โปรดปฏิบัติตามข้อควรระวังในการเขียนฟังก์ชันด้วย infinite cardinality; นั่นคือ, ฟังก์ชันที่มักจะส่งคืนแถวบนการเรียกแบบ FETCH. DB2 คาดหวังในเงื่อนไข *end-of-table*, เพื่อเป็นเสมือนตัวเร่งการทำงานภายใน query processing. ดังนั้นฟังก์ชันตารางที่ไม่เคยส่งคืนเงื่อนไข *end-of-table* (SQL-state value '02000') จะก่อให้เกิดการประมวลผลแบบวนซ้ำไม่สิ้นสุด.

การประมวลผลข้อผิดพลาดของ UDFs:

เมื่อเกิดข้อผิดพลาดขึ้น ในขณะที่ประมวลผล UDF, ระบบจะทำตามแบบจำลองที่ได้ระบุไว้.

การประมวลผลข้อผิดพลาดฟังก์ชันตาราง

การประมวลผลข้อผิดพลาดสำหรับการเรียกฟังก์ชันตารางมีดังต่อไปนี้:

- หากการเรียก FIRST ล้มเหลว, จะไม่มีการเรียกครั้งต่อไป.
- หากการเรียก FIRST สำเร็จ, จะมีการเรียก OPEN, FETCH, และ CLOSE แบบ nested, และจะมีการเรียกแบบ FINAL เสมอ.
- หากการเรียก OPEN ล้มเหลว, จะไม่มีการเรียก FETCH หรือ CLOSE.
- หากการเรียก OPEN ดำเนินต่อไป, จะมีการเรียกแบบ FETCH และ CLOSE.
- หากการเรียกแบบ FETCH ล้มเหลว, จะไม่มีการเรียกแบบ FETCH อีกต่อไป, แต่จะมีการเรียก CLOSE.

หมายเหตุ: แบบจำลองนี้จะอธิบายการประมวลผลข้อผิดพลาดแบบธรรมดาสำหรับตาราง UDF. ในกรณีที่ระบบล้มเหลวหรือมีปัญหาในการสื่อสาร, อาจไม่มีการเรียกที่ระบุโดยแบบจำลองการประมวลผลข้อผิดพลาด.

การประมวลผลข้อผิดพลาดฟังก์ชัน Scalar

แบบจำลองการประมวลผลข้อผิดพลาดสำหรับ UDF แบบ scalar, ซึ่งถูกกำหนดด้วยค่ากำหนด FINAL CALL, มีดังต่อไปนี้:

1. หากการเรียก FIRST ล้มเหลว, จะไม่มีการเรียกครั้งต่อไป.
2. หากการเรียก FIRST ดำเนินต่อไป, จะมีการเรียก NORMAL ต่อไปตามที่ได้รับประกันจากการประมวลผลคำสั่ง, และจะมีการเรียก FINAL เสมอ.
3. หากการเรียก NORMAL ล้มเหลว, จะไม่มีการเรียกแบบ NORMAL อีกต่อไป, แต่จะมีการเรียก FINAL (หากคุณระบุ FINAL CALL). ซึ่งหมายความว่าหากมีการส่งคืนข้อผิดพลาดบนการเรียก FIRST, UDF ต้องลบทิ้งก่อนการส่งคืน, เพราะไม่มีการเรียก FINAL.

หมายเหตุ: แบบจำลองนี้จะอธิบายการประมวลผลข้อผิดพลาดแบบธรรมดาสำหรับ UDF แบบสเกลาร์. ในกรณีที่ระบบล้มเหลวหรือมีปัญหาในการสื่อสาร, อาจไม่มีการเรียกที่ระบุโดยแบบจำลองการประมวลผลข้อผิดพลาด.

ข้อควรพิจารณา Threads:

UDF, ที่ถูกกำหนดเป็น FENCED, รันในงานเดียวกันเช่นเดียวกับคำสั่ง SQL ที่เรียก UDF ใช้งาน. อย่างไรก็ตาม, UDF รันใน thread ของระบบ, แยกจาก thread ที่กำลังรันคำสั่ง SQL.

เนื่องจาก UDF รันในงานเดียวกับคำสั่ง SQL, UDF จึงอยู่ในสภาพแวดล้อมเดียวกับคำสั่ง SQL. อย่างไรก็ตาม, เนื่องจาก UDF รันภายใต้ thread ที่แยกต่างหาก, จึงควรพิจารณาเรื่อง thread ต่อไปนี้:

- UDF จะขัดแย้งกับรีซอร์สระดับ thread ที่เกิดจาก thread ของคำสั่ง SQL. แรกเริ่ม, ข้อมูลเหล่านี้คือรีซอร์สตารางที่ถูกกล่าวถึงด้านบน.
- UDFs จะไม่รับช่วงสิทธิ์ที่รับมาจากโปรแกรมที่อาจเด้กที่ขณะที่คำสั่ง SQL ถูกเรียกทำงาน. สิทธิ UDF มาจากสิทธิในการใช้งานที่เชื่อมโยงกับตัวโปรแกรม UDF หรือมาจากสิทธิในการใช้งานของการรันคำสั่ง SQL ของผู้ใช้.
- UDF ไม่สามารถปฏิบัติการใดๆ ที่ถูกบล็อกไว้ไม่ให้รันใน thread รอง.
- โปรแกรม UDF ต้องถูกสร้างให้สามารถรันภายใต้ activation group ที่มีชื่อหรือใน activation group ของตัวเรียก (พารามิเตอร์ ACTGRP). โปรแกรมที่ระบุ ACTGRP(*NEW) จะไม่ได้รับอนุญาตให้รันเป็น UDFs.

สิ่งอ้างอิงที่เกี่ยวข้อง

“ข้อควรพิจารณาเรื่องเฟนซ์ (Fenced) หรือ อันเฟนซ์ (unfenced)” ในหน้า 197

เมื่อสร้างฟังก์ชันแบบผู้ใช้กำหนดเอง (UDF) โปรดพิจารณาว่า จะสร้าง UDF หรือ unfenced UDF.

ข้อมูลที่เกี่ยวข้อง

ข้อควรพิจารณาเกี่ยวกับฐานข้อมูลสำหรับโปรแกรมมิ่งแบบ multithreaded

การประมวลผลแบบขนาน:

คุณสามารถกำหนด UDF ให้ประมวลผลแบบขนานได้.

หมายความว่าโปรแกรม UDF เดียวกันสามารถรันในหลายๆ thread ในเวลาเดียวกันได้. ดังนั้น, หาก ALLOW PARALLEL ถูกระบุไว้สำหรับ UDF, โปรดตรวจสอบให้แน่ใจว่าการทำงานดังกล่าวไม่เป็นผลเสียต่อ thread.

ฟังก์ชันตารางแบบผู้ใช้กำหนดไม่สามารถรันแบบขนานได้; ดังนั้น, จึงต้องระบุ DISALLOW PARALLEL เมื่อสร้างฟังก์ชัน

ข้อมูลที่เกี่ยวข้อง

ข้อควรพิจารณาเกี่ยวกับฐานข้อมูลสำหรับโปรแกรมมิ่งแบบ multithreaded

ข้อควรพิจารณาเรื่องเฟนซ์ (Fenced) หรือ อันเฟนซ์ (unfenced):

เมื่อสร้างฟังก์ชันแบบผู้ใช้กำหนดเอง (UDF) โปรดพิจารณาว่าจะสร้าง UDF หรือ unfenced UDF.

ตามค่าดีฟอลต์, UDF จะถูกสร้างขึ้นเป็น fenced UDF. Fenced จะแสดงว่าฐานข้อมูลควรรัน UDF ใน thread ต่างหาก. สำหรับ UDF ที่ซับซ้อน, การแยกนี้มีความสำคัญเพราะช่วยเลี่ยงปัญหาที่อาจเกิดขึ้นได้ เช่น การสร้างชื่อเคอร์เซอร์ SQL แบบเฉพาะ. การไม่ต้องกังวลเรื่องความขัดแย้งของรีซอร์สคือเหตุผลหนึ่งที่ต้องใช้ค่าดีฟอลต์และสร้าง UDF เป็น UDF แบบ fenced. UDF ที่ถูกสร้างขึ้นด้วยอ็อปชัน 'NOT FENCED' จะระบุกับฐานข้อมูลว่าผู้ใช้กำลังร้องขอให้ UDF รันภายใน thread เดียวกับที่เริ่มต้นใช้งาน UDF. ขอแนะนำให้ใช้ Unfenced กับฐานข้อมูล, ซึ่งยังคงตัดสินใจให้รัน UDF ด้วยวิธีเดียวกันกับ fenced UDF.

```
CREATE FUNCTION QGPL.FENCED (parameter1 INTEGER)
RETURNS INTEGER LANGUAGE SQL
BEGIN
RETURN parameter1 * 3;
END;
```

```
CREATE FUNCTION QGPL.UNFENCED1 (parameter1 INTEGER)
RETURNS INTEGER LANGUAGE SQL NOT FENCED
-- สร้าง UDF เพื่อร้องขอการทำงานที่รวดเร็วยิ่งขึ้นผ่านทางอ็อปชัน NOT FENCED
BEGIN
RETURN parameter1 * 3;
END;
```

สิ่งอ้างอิงที่เกี่ยวข้อง

“ข้อควรพิจารณา Threads” ในหน้า 196

UDF, ที่ถูกกำหนดเป็น FENCED, รันในงานเดียวกันเช่นเดียวกับคำสั่ง SQL ที่เรียก UDF ใช้งาน. อย่างไรก็ตาม, UDF รันใน thread ของระบบ, แยกจาก thread ที่กำลังรันคำสั่ง SQL.

ข้อควรพิจารณาในการบันทึกและการกู้คืน:

เมื่อฟังก์ชันภายนอกที่เชื่อมโยงกับโปรแกรมภายนอก ILE หรือเซอร์วิสโปรแกรมถูกสร้างขึ้นมา, จะมีการพยายามบันทึกแอตทริบิวต์ของฟังก์ชันในโปรแกรม หรืออ็อบเจกต์เซอร์วิสโปรแกรมที่เชื่อมโยงอยู่.

ถ้าอ็อบเจกต์ *PGM หรือ *SRVPGM ถูกบันทึกแล้วถูกเรียกคืนให้กับระบบนี้หรือระบบอื่นแล้ว, แคตาล็อกจะอัปเดตแอตทริบิวต์เหล่านั้นให้โดยอัตโนมัติ. ถ้าแอตทริบิวต์ของฟังก์ชันไม่สามารถบันทึกได้, แล้วแคตาล็อกจะไม่อัปเดตให้โดยอัตโนมัติ และผู้ใช้ต้องสร้างฟังก์ชันภายนอกไว้บนระบบใหม่. แอตทริบิวต์สามารถบันทึกสำหรับฟังก์ชันภายนอกได้ภายใต้ข้อจำกัดดังต่อไปนี้:

- โปรแกรมไลบรารีภายนอกต้องไม่ใช่ QSYS หรือ QSYS2.
- โปรแกรมภายนอกต้องมีอยู่จริงเมื่อคำสั่ง CREATE FUNCTION ถูกเรียกใช้งาน.
- โปรแกรมภายนอกต้องเป็นอ็อบเจกต์ ILE *PGM หรือ *SRVPGM.
- โปรแกรมภายนอกหรือเซอร์วิสโปรแกรมต้องมีคำสั่ง SQL อย่างน้อยหนึ่งคำสั่ง.

ถ้าอ็อบเจกต์โปรแกรมไม่สามารถอัปเดตได้, ฟังก์ชันจะยังถูกสร้างอยู่.

ตัวอย่าง: โค้ด UDF

ตัวอย่างต่อไปนี้จะแสดงวิธีการใช้งานโค้ด UDF โดยใช้ฟังก์ชัน SQL และฟังก์ชันภายนอก.

ตัวอย่าง: Square ของ UDF หมายเลข:

ในตัวอย่างนี้, สมมติว่า คุณต้องการฟังก์ชันที่ส่งคืนค่ายกกำลังของตัวเลข.

คำสั่งเคียวรีคือ:

```
SELECT SQUARE(myint) FROM mytable
```

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

ตัวอย่างต่อไปนี้จะแสดงวิธีการกำหนด UDF หลากๆ วิธี.

การใช้ฟังก์ชัน SQL

คำสั่ง CREATE FUNCTION:

```
CREATE FUNCTION SQUARE( inval INT) RETURNS INT  
LANGUAGE SQL  
SET OPTION DBGVIEW=*SOURCE  
BEGIN  
RETURN(inval*inval);  
END
```

โปรแกรมนี้จะสร้างฟังก์ชัน SQL ที่คุณสามารถเรียกได้.

การใช้ฟังก์ชันภายนอก, ลักษณะของพารามิเตอร์ SQL

คำสั่ง CREATE FUNCTION:

```
CREATE FUNCTION SQUARE(INT) RETURNS INT CAST FROM FLOAT  
LANGUAGE C  
EXTERNAL NAME 'MYLIB/MATH(SQUARE)'  
DETERMINISTIC  
NO SQL  
NO EXTERNAL ACTION  
PARAMETER STYLE SQL  
ALLOW PARALLEL
```

โค้ด:

```
void SQUARE(int *inval,  
double *outval,  
short *inind,  
short *outind,  
char *sqlstate,  
char *funcname,  
char *specname,  
char *msgtext)  
{
```

```

if (*inind<0)
    *outind=-1;
else
    {
    *outval=*inval;
    *outval=(*outval)*(*outval);
    *outind=0;
    }
return;
}

```

วิธีการสร้างเซอริสโปรแกรมภายนอกเพื่อให้ต่อบักได้:

```

CRTCMOD MODULE(mylib/square) DBGVIEW(*SOURCE)
    CRTSRVPGM SRVPGM(mylib/math) MODULE(mylib/square)
    EXPORT(*ALL) ACTGRP(*CALLER)

```

การใช้ฟังก์ชันภายนอก, ลักษณะของพารามิเตอร์ GENERAL

คำสั่ง CREATE FUNCTION:

```

CREATE FUNCTION SQUARE(INT) RETURNS INT CAST FROM FLOAT
LANGUAGE C
EXTERNAL NAME 'MYLIB/MATH(SQUARE)'
DETERMINISTIC
NO SQL
NO EXTERNAL ACTION
PARAMETER STYLE GENERAL
ALLOW PARALLEL

```

โค้ด:

```

double SQUARE(int *inval)
{
    double outval;
    outval=*inval;
    outval=outval*outval;
    return(outval);
}

```

วิธีการสร้างเซอริสโปรแกรมภายนอกเพื่อให้ต่อบักได้:

```

CRTCMOD MODULE(mylib/square) DBGVIEW(*SOURCE)
    CRTSRVPGM SRVPGM(mylib/math) MODULE(mylib/square)
    EXPORT(*ALL) ACTGRP(*CALLER)

```

ตัวอย่าง: ตัวนับ:

สมมติว่าคุณต้องการกำหนดจำนวนแถวในคำสั่ง SELECT. ดังนั้น คุณจึงเขียน UDF ซึ่งเพิ่มขึ้น และส่งคืนตัวนับ.

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

ตัวอย่างนี้มีการใช้ ฟังก์ชันภายนอกกับรูปแบบพารามิเตอร์ DB2 SQL และ scratchpad.

```
CREATE FUNCTION COUNTER (
    RETURNS INT
    SCRATCHPAD
    NOT DETERMINISTIC
    NO SQL
    NO EXTERNAL ACTION
    LANGUAGE C
    PARAMETER STYLE DB2SQL
    EXTERNAL NAME 'MYLIB/MATH(ctr)'
    DISALLOW PARALLEL

/* structure scr defines the passed scratchpad for the function "ctr" */
struct scr {
    long len;
    long countr;
    char not_used[96];
};

void ctr (
    long *out,                /* output answer (counter) */
    short *outnull,          /* output NULL indicator */
    char *sqlstate,          /* SQL STATE */
    char *funcname,          /* function name */
    char *specname,          /* specific function name */
    char *mesgtext,          /* message text insert */
    struct scr *scratchptr) { /* scratch pad */

    *out = ++scratchptr->countr; /* increment counter & copy out */
    *outnull = 0;
    return;
}
/* end of UDF : ctr */
```

สำหรับ UDF นี้, โปรดสังเกตว่า:

- ไม่มีการระบุอักขระ SQL อินพุต, แต่มีการส่งคืนค่า.
- UDF จะอยู่ต่อท้ายอักขระอินพุต scratchpad หลังจากอักขระการติดตามมาตรฐาน 4 อักขระ, ซึ่งได้แก่ *SQL-state, function-name, specific-name, และ message-text*.
- UDF จะรวมถึง definition โครงสร้างเพื่อแม็พ scratchpad ที่ถูกส่งผ่าน.
- ไม่มีการกำหนดพารามิเตอร์อินพุต. การดำเนินการนี้สอดคล้องกับโค้ด.
- SCRATCHPAD ถูกโค้ด, ทำให้ DB2 มีการจัดสรร, initialize และส่งผ่าน อักขระอินพุต scratchpad ได้อย่างเหมาะสม.
- คุณต้องระบุ scratchpad ให้เป็น NOT DETERMINISTIC, เพราะ scratchpad ขึ้นอยู่กับอักขระอินพุต SQL มากกว่า, (ไม่ใช่ในกรณีนี้).
- คุณระบุ DISALLOW PARALLEL ไว้อย่างถูกต้องแล้ว, เพราะฟังก์ชันการแก้ไขของ UDF ขึ้นอยู่กับ single scratchpad.

ตัวอย่าง: ฟังก์ชันตารางอากาศ:

ต่อไปนี้คือฟังก์ชันตารางตัวอย่างที่รายงานข้อมูลอากาศของเมืองต่างๆ ในสหรัฐอเมริกา.

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

วันที่บันทึกสภาพอากาศสำหรับเมืองเหล่านี้จะถูกอ่านจากไฟล์ภายนอก, ตามที่ระบุไว้ในข้อสังเกตซึ่งอยู่ในโปรแกรมตัวอย่าง. ข้อมูลจะรวมถึงชื่อของเมืองตามด้วยข้อมูลอากาศของเมืองนั้น. โดยจะใช้รูปแบบเดียวกันนี้สำหรับเมืองอื่นๆ ด้วย.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqludf.h> /* for use in compiling User Defined Function */

#define SQL_NOTNULL 0 /* Nulls Allowed - Value is not Null */
#define SQL_ISNULL -1 /* Nulls Allowed - Value is Null */
#define SQL_TYP_VARCHAR 448
#define SQL_TYP_INTEGER 496
#define SQL_TYP_FLOAT 480

/* Short and long city name structure */
typedef struct {
    char * city_short ;
    char * city_long ;
} city_area ;

/* Scratchpad data */ (See note 1)
/* Preserve information from one function call to the next call */
typedef struct {
    /* FILE * file_ptr; if you use weather data text file */
    int file_pos ; /* if you use a weather data buffer */
} scratch_area ;

/* Field descriptor structure */
typedef struct {
    char fld_field[31] ; /* Field data */
    int fld_ind ; /* Field null indicator data */
    int fld_type ; /* Field type */
    int fld_length ; /* Field length in the weather data */
    int fld_offset ; /* Field offset in the weather data */
} fld_desc ;

/* Short and long city name data */
city_area cities[] = {
    { "alb", "Albany, NY" },
    { "atl", "Atlanta, GA" },
    .
    .
    .
    { "wbc", "Washington DC, DC" },
    /* You may want to add more cities here */

    /* Do not forget a null termination */
    { ( char * ) 0, ( char * ) 0 }
} ;

/* Field descriptor data */
```

```

fld_desc fields[] = {
    { "", SQL_ISNULL, SQL_TYP_VARCHAR, 30, 0 }, /* city      */
    { "", SQL_ISNULL, SQL_TYP_INTEGER, 3, 2 }, /* temp_in_f   */
    { "", SQL_ISNULL, SQL_TYP_INTEGER, 3, 7 }, /* humidity    */
    { "", SQL_ISNULL, SQL_TYP_VARCHAR, 5, 13 }, /* wind        */
    { "", SQL_ISNULL, SQL_TYP_INTEGER, 3, 19 }, /* wind_velocity */
    { "", SQL_ISNULL, SQL_TYP_FLOAT, 5, 24 }, /* barometer   */
    { "", SQL_ISNULL, SQL_TYP_VARCHAR, 25, 30 }, /* forecast    */
    /* You may want to add more fields here */

    /* Do not forget a null termination */
    { ( char ) 0, 0, 0, 0, 0 }
};

/* Following is the weather data buffer for this example. You */
/* may want to keep the weather data in a separate text file. */
/* Uncomment the following fopen() statement. Note that you */
/* need to specify the full path name for this file. */
char * weather_data[] = {
    "alb.forecast",
    " 34 28% wnw 3 30.53 clear",
    "atl.forecast",
    " 46 89% east 11 30.03 fog",
    .
    .
    .
    "wbc.forecast",
    " 38 96% ene 16 30.31 light rain",
    /* You may want to add more weather data here */

    /* Do not forget a null termination */
    ( char * ) 0
};

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Find a full city name using a short name */
int get_name( char * short_name, char * long_name ) {

    int name_pos = 0 ;

    while ( cities[name_pos].city_short != ( char * ) 0 ) {
        if ( strcmp(short_name, cities[name_pos].city_short) == 0 ) {
            strcpy( long_name, cities[name_pos].city_long );
            /* A full city name found */
            return( 0 );
        }
        name_pos++;
    }

    /* can not find such city in the city data */
    strcpy( long_name, "Unknown City" );
    return( -1 );
}

```



```

}

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Clean all field data and field null indicator data */
int clean_fields( int field_pos ) {

    while (fields[field_pos].fld_length !=0 ) {
        memset( fields[field_pos].fld_field, '\0', 31 );
        fields[field_pos].fld_ind = SQL_ISNULL ;
        field_pos++ ;
    }

    return( 0 ) ;

}

#ifdef __cplusplus
extern "C"
#endif
/* This is a subroutine. */
/* Fills all field data and field null indicator data ... */
/* ... from text weather data */
int get_value( char * value, int field_pos ) {

    fld_desc * field ;
    char field_buf[31] ;
    double * double_ptr ;
    int * int_ptr, buf_pos ;

    while ( fields[field_pos].fld_length != 0 ) {
        field = &fields[field_pos] ;
        memset( field_buf, '\0', 31 ) ;
        memcpy( field_buf,
            ( value + field->fld_offset ),
            field->fld_length ) ;
        buf_pos = field->fld_length ;
        while ( ( buf_pos > 0 ) &&
            ( field_buf[buf_pos] == ' ' ) )
            field_buf[buf_pos--] = '\0' ;
        buf_pos = 0 ;
        while ( ( buf_pos < field->fld_length ) &&
            ( field_buf[buf_pos] == ' ' ) )
            buf_pos++ ;
        if ( strlen( ( char * ) ( field_buf + buf_pos ) ) > 0 ||
            strcmp( ( char * ) ( field_buf + buf_pos ), "n/a") != 0 ) {
            field->fld_ind = SQL_NOTNULL ;

            /* Text to SQL type conversion */
            switch( field->fld_type ) {
                case SQL_TYP_VARCHAR:
                    strcpy( field->fld_field,
                        ( char * ) ( field_buf + buf_pos ) ) ;
                    break ;

```

```

        case SQL_TYP_INTEGER:
int_ptr = ( int * ) field->fld_field ;
            *int_ptr = atoi( ( char * ) ( field_buf + buf_pos ) ) ;
            break ;
        case SQL_TYP_FLOAT:
double_ptr = ( double * ) field->fld_field ;
            *double_ptr = atof( ( char * ) ( field_buf + buf_pos ) ) ;
            break ;
        /* You may want to add more text to SQL type conversion here */
    }
}
    field_pos++ ;
}
    return( 0 ) ;
}

#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN weather( /* Return row fields */
    SQLUDF_VARCHAR * city,
    SQLUDF_INTEGER * temp_in_f,
    SQLUDF_INTEGER * humidity,
    SQLUDF_VARCHAR * wind,
    SQLUDF_INTEGER * wind_velocity,
    SQLUDF_DOUBLE * barometer,
    SQLUDF_VARCHAR * forecast,
    /* You may want to add more fields here */

    /* Return row field null indicators */
    SQLUDF_NULLIND * city_ind,
    SQLUDF_NULLIND * temp_in_f_ind,
    SQLUDF_NULLIND * humidity_ind,
    SQLUDF_NULLIND * wind_ind,
    SQLUDF_NULLIND * wind_velocity_ind,
    SQLUDF_NULLIND * barometer_ind,
    SQLUDF_NULLIND * forecast_ind,
    /* You may want to add more field indicators here */

    /* UDF always-present (trailing) input arguments */
    SQLUDF_TRAIL_ARGS_ALL
) {

    scratch_area * save_area ;
    char line_buf[81] ;
    int line_buf_pos ;

    /* SQLUDF_SCRAT is part of SQLUDF_TRAIL_ARGS_ALL */
    /* Preserve information from one function call to the next call */
    save_area = ( scratch_area * ) ( SQLUDF_SCRAT->data ) ;

    /* SQLUDF_CALLT is part of SQLUDF_TRAIL_ARGS_ALL */
    switch( SQLUDF_CALLT ) {

```

```

/* First call UDF: Open table and fetch first row */
case SQL_TF_OPEN:
    /* If you use a weather data text file specify full path */
    /* save_area->file_ptr = fopen("tblsrv.dat","r"); */
    save_area->file_ptr = 0 ;
    break ;

/* Normal call UDF: Fetch next row */ (See note 2)
case SQL_TF_FETCH:
    /* If you use a weather data text file */
    /* memset(line_buf, '\0', 81); */
    /* if (fgets(line_buf, 80, save_area->file_ptr) == NULL) { */
    if ( weather_data[save_area->file_ptr] == ( char * ) 0 ) {

        /* SQLUDF_STATE is part of SQLUDF_TRAIL_ARGS_ALL */
        strcpy( SQLUDF_STATE, "02000" );

        break ;
    }

    memset( line_buf, '\0', 81 );
    strcpy( line_buf, weather_data[save_area->file_ptr] );
    line_buf[3] = '\0' ;

    /* Clean all field data and field null indicator data */
    clean_fields( 0 ) ;

    /* Fills city field null indicator data */
    fields[0].fld_ind = SQL_NOTNULL ;

    /* Find a full city name using a short name */
    /* Fills city field data */
    if ( get_name( line_buf, fields[0].fld_field ) == 0 ) {
save_area->file_ptr++ ;
        /* If you use a weather data text file */
        /* memset(line_buf, '\0', 81); */
        /* if (fgets(line_buf, 80, save_area->file_ptr) == NULL) { */
        if ( weather_data[save_area->file_ptr] == ( char * ) 0 ) {
            /* SQLUDF_STATE is part of SQLUDF_TRAIL_ARGS_ALL */
            strcpy( SQLUDF_STATE, "02000" );
            break ;
        }

        memset( line_buf, '\0', 81 );
        strcpy( line_buf, weather_data[save_area->file_ptr] );
        line_buf_pos = strlen( line_buf ) ;
        while ( line_buf_pos > 0 ) {
            if ( line_buf[line_buf_pos] >= ' ' )
                line_buf_pos = 0 ;
            else {
                line_buf[line_buf_pos] = '\0' ;
                line_buf_pos-- ;
            }
        }
    }
}

```

```

/* Fills field data and field null indicator data ... */
/* ... for selected city from text weather data */
get_value( line_buf, 1 ) ; /* Skips city field */

/* Builds return row fields */
strcpy( city, fields[0].fld_field ) ;
memcpy( (void *) temp_in_f,
        fields[1].fld_field,
        sizeof( SQLUDF_INTEGER ) ) ;
memcpy( (void *) humidity,
        fields[2].fld_field,
        sizeof( SQLUDF_INTEGER ) ) ;
strcpy( wind, fields[3].fld_field ) ;
memcpy( (void *) wind_velocity,
        fields[4].fld_field,
        sizeof( SQLUDF_INTEGER ) ) ;
memcpy( (void *) barometer,
        fields[5].fld_field,
        sizeof( SQLUDF_DOUBLE ) ) ;
strcpy( forecast, fields[6].fld_field ) ;

/* Builds return row field null indicators */
memcpy( (void *) city_ind,
        &(fields[0].fld_ind),
        sizeof( SQLUDF_NULLIND ) ) ;
memcpy( (void *) temp_in_f_ind,
        &(fields[1].fld_ind),
        sizeof( SQLUDF_NULLIND ) ) ;
memcpy( (void *) humidity_ind,
        &(fields[2].fld_ind),
        sizeof( SQLUDF_NULLIND ) ) ;
memcpy( (void *) wind_ind,
        &(fields[3].fld_ind),
        sizeof( SQLUDF_NULLIND ) ) ;
memcpy( (void *) wind_velocity_ind,
        &(fields[4].fld_ind),
        sizeof( SQLUDF_NULLIND ) ) ;
memcpy( (void *) barometer_ind,
        &(fields[5].fld_ind),
        sizeof( SQLUDF_NULLIND ) ) ;
memcpy( (void *) forecast_ind,
        &(fields[6].fld_ind),
        sizeof( SQLUDF_NULLIND ) ) ;

/* Next city weather data */
save_area->file_pos++ ;

        break ;

/* Special last call UDF for clean up (no real args!): Close table */ (See note 3)
case SQL_TF_CLOSE:
    /* If you use a weather data text file */
    /* fclose(save_area->file_ptr); */
    /* save_area->file_ptr = NULL; */
    save_area->file_pos = 0 ;

```

```

        break ;
    }
}

```

เมื่ออ้างถึงหมายเลขที่ถูกลบอยู่ในโค้ด UDF นี้, โปรดสังเกตว่า:

1. Scratchpad ถูกกำหนดนิยามไว้แล้ว. ตัวแปรแถว ถูก initialize ในการเรียกแบบ OPEN, และ iptr array และตัวแปร nbr_rows ถูกเติมเต็มด้วยฟังก์ชัน *mystery* ขณะเปิดใช้.
2. FETCH จะสำรวจ iptr array, โดยการใช้แถวเป็นดรรชนี, และย้ายค่าผลตอบแทนจากส่วนประกอบปัจจุบันของ iptr ไปยังตำแหน่งที่ถูกลบโดยตัวชี้ค่าผลลัพธ์ out_c1, out_c2, และ out_c3.
3. ในที่สุด, CLOSE จะลบข้อมูลในหน่วยเก็บของ OPEN และ anchor ไว้ใน scratchpad.

ต่อไปนี้เป็นคำสั่ง CREATE FUNCTION สำหรับ UDF นี้:

```

CREATE FUNCTION tfweather_u()
  RETURNS TABLE (CITY VARCHAR(25),
                 TEMP_IN_F INTEGER,
                 HUMIDITY INTEGER,
                 WIND VARCHAR(5),
                 WIND_VELOCITY INTEGER,
                 BAROMETER FLOAT,
                 FORECAST VARCHAR(25))

SPECIFIC tfweather_u
DISALLOW PARALLEL
  NOT FENCED
  DETERMINISTIC

NO SQL
NO EXTERNAL ACTION
  SCRATCHPAD
  NO FINAL CALL
  LANGUAGE C
  PARAMETER STYLE DB2SQL
EXTERNAL NAME 'LIB1/WEATHER(weather)';

```

การอ้างถึงหมายเลขหมายเหตุที่ฝังไว้, โปรดสังเกตว่า:

- คำสั่งนี้ไม่ได้ใช้งานอินพุตใดๆ, และส่งคืนคอลัมน์เอาต์พุต 7 คอลัมน์.
- มีการระบุ SCRATCHPAD, ดังนั้น DB2 ได้จัดสรร, เตรียมข้อมูลเบื้องต้น และส่งผ่านอักขระเมนต์ scratchpad อย่างเหมาะสม.
- มีการระบุ NO FINAL CALL.
- ฟังก์ชันถูกระบุเป็น NOT DETERMINISTIC, เพราะฟังก์ชันนั้นขึ้นอยู่กับสิ่งอื่นที่มากกว่าอักขระเมนต์อินพุต SQL. นั่นคือ, ฟังก์ชันนั้นขึ้นอยู่กับฟังก์ชัน *mystery* และเราเข้าใจว่าข้อมูลอาจแตกต่างกันไปในแต่ละการทำงาน.
- DISALLOW PARALLEL จำเป็นสำหรับฟังก์ชันตาราง.
- CARDINALITY 100 เป็นจำนวนแถวที่คาดว่าจะส่งกลับโดยประมาณ, ที่จัดเตรียมไว้สำหรับ DB2 optimizer.
- DBINFO ไม่ถูกใช้งาน, และไม่มีการ optimization เพื่อส่งคืนคอลัมน์ที่จำเป็นสำหรับคำสั่งเฉพาะที่อ้างอิงถึงฟังก์ชันนี้.
- มีการระบุ NOT NULL CALL, ดังนั้น จะไม่มีการเรียก UDF ถ้าอักขระเมนต์ SQL อินพุตเป็น NULL, และไม่จำเป็นต้องตรวจสอบสำหรับเงื่อนไขนี้.

หากต้องการเลือกแถวทั้งหมดที่สร้างขึ้นโดยฟังก์ชันตารางนี้, โปรดใช้การสืบค้นต่อไปนี้:

```
SELECT *  
FROM TABLE (tfweather_u())x
```

การใช้ UDF ในคำสั่ง SQL

สามารถเรียกใช้งาน UDF แบบ Scalar และแบบคอลัมน์ภายในคำสั่ง SQL เกือบทุกๆ ที่ที่ใช้งานนิพจน์ได้. เราสามารถเรียกใช้ตาราง UDF ใน FROM clause ของ SELECT. อย่างไรก็ตาม, มีข้อจำกัดบางอย่างสำหรับการใช้ UDF.

- UDF และฟังก์ชันที่ระบบสร้างขึ้นจะไม่สามารถถูกระบุในการตรวจสอบข้อจำกัดได้. การตรวจสอบข้อจำกัดยังคงไม่สามารถบรรจุการอ้างอิงถึงฟังก์ชันในตัวบางอย่าง ซึ่งปฏิบัติโดยระบบเช่นเดียวกับ UDFs.
- UDF ภายนอก, SQL UDFS และฟังก์ชันในตัว DLVALUE, DLURLPATH, DLURLPATHONLY, DLURLSCHEME, DLURLCOMPLETE, และ DLURLSERVER ไม่สามารถอ้างอิงได้จากอนุประโยค ORDER BY หรือ GROUP BY, นอกจากว่าคำสั่ง SQL จะอ่านได้อย่างเดียวเท่านั้นและมีการอนุญาตให้ประมวลผล (ALWCOPYDATA(*YES) หรือ (*OPTIMIZE)) ได้ชั่วคราว.

การใช้ตัวทำเครื่องหมายพารามิเตอร์ หรือค่า NULL ในอาทิวเมนต์ฟังก์ชัน:

ข้อจำกัดที่สำคัญที่มีผลต่อทั้งตัวทำเครื่องหมายพารามิเตอร์ และค่า NULL.

คุณไม่สามารถโค้ดสิ่งต่อไปนี้:

```
BLOOP(?)
```

หรือ

```
BLOOP(NULL)
```

เนื่องจากการแก้ปัญหาฟังก์ชันจะไม่ว่าชนิดข้อมูลของอาทิวเมนต์จะมีค่าเป็นอะไร, ดังนั้นจะไม่สามารถแก้ปัญหการอ้างอิงได้. คุณสามารถใช้คำกำหนด CAST เพื่อเตรียมชนิดข้อมูลสำหรับตัวทำเครื่องหมายพารามิเตอร์ หรือค่า NULL ที่ฟังก์ชันสามารถใช้ได้:

```
BLOOP(CAST(? AS INTEGER))
```

หรือ

```
BLOOP(CAST(NULL AS INTEGER))
```

การใช้การอ้างอิงฟังก์ชันที่ครบตามเกณฑ์:

ถ้าคุณใช้การอ้างอิงฟังก์ชันที่ครบตามเกณฑ์, คุณต้องจำกัดการค้นหาสำหรับฟังก์ชันที่ตรงกันกับแบบแผนนั้น.

ตัวอย่างเช่น, คุณมีคำสั่งต่อไปนี้:

```
SELECT PABLO.BLOOP(COLUMN1) FROM T
```

เฉพาะฟังก์ชัน BLOOP ใน Schema PABLO เท่านั้นที่ถูกพิจารณา. มันไม่สำคัญว่าผู้ใช้คือ SERGE เป็นผู้นิยามฟังก์ชัน BLOOP, หรือฟังก์ชันนั้นคือฟังก์ชันในตัว BLOOP หรือไม่. สมมติว่า ผู้ใช้คือ PABLO เป็นผู้นิยามฟังก์ชัน BLOOP สองฟังก์ชันในแบบแผนของเขา:

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS ...  
CREATE FUNCTION BLOOP (DOUBLE) RETURNS ...
```

ดังนั้น BLOOP จะมีการไหลตมมากเกินไปใน PABLO schema, และ algorithm ของการเลือก ฟังก์ชัน จะเลือก BLOOP ที่ดีที่สุด, โดยขึ้นอยู่กับชนิดข้อมูลของอาทิวเมนต์, COLUMN1. ในกรณีนี้, PABLO.BLOOP ทั้งสองฟังก์ชันรับอาทิวเมนต์ที่เป็นตัวเลข, และถ้า COLUMN1 ไม่ใช่ชนิดข้อมูลแบบตัวเลขแล้ว, คำสั่งนี้จะทำงานไม่สำเร็จ. ในทางตรงกันข้ามถ้า COLUMN1 เป็น SMALLINT หรือ INTEGER อย่างใดอย่างหนึ่งแล้ว, การเลือกฟังก์ชันจะเลือก resolve ตัว BLOOP ตัวแรก, ในขณะที่ถ้า COLUMN1 เป็น DECIMAL หรือ DOUBLE แล้ว, BLOOP ตัวที่สองจะถูกเลือก.

มีจุดที่น่าสนใจหลายอย่างเกี่ยวกับตัวอย่างนี้:

1. มันแสดงการเลื่อนระดับของอาทิวเมนต์. ฟังก์ชัน BLOOP ถูกนิยามด้วยพารามิเตอร์แบบ INTEGER, แต่คุณสามารถผ่านค่าเป็นอาทิวเมนต์แบบ SMALLINT ได้. algorithm การเลือกฟังก์ชันจะสนับสนุนการส่งเสริมระหว่างชนิดข้อมูลในตัว และ DB2 ที่ทำการแปลงค่าข้อมูลที่เหมาะสม.
2. ถ้าในบางเหตุผลที่คุณต้องการเรียกฟังก์ชัน BLOOP ตัวที่สองด้วยอาทิวเมนต์แบบ SMALLINT หรือ INTEGER แล้ว, คุณต้องกระทำโดยตรงในคำสั่งของคุณดังต่อไปนี้:

```
SELECT PABLO.BLOOP( DOUBLE(COLUMN1)) FROM T
```

3. ถ้าคุณต้องการเรียกใช้ BLOOP แรกด้วยอาทิวเมนต์แบบ DECIMAL หรือ DOUBLE, คุณมีตัวเลือกในการกระทำโดยตรง, โดยขึ้นอยู่กับจุดประสงค์ของคุณ:

```
SELECT PABLO.BLOOP(INTEGER(COLUMN1)) FROM T  
SELECT PABLO.BLOOP(FLOOR(COLUMN1)) FROM T
```

สิ่งอ้างอิงที่เกี่ยวข้อง

“การใช้การอ้างอิงฟังก์ชันที่ไม่ครบตามเกณฑ์”

คุณสามารถใช้การอ้างอิงฟังก์ชันที่ไม่ครบตามเกณฑ์แทนการอ้างอิงฟังก์ชันที่ครบตามเกณฑ์ได้. ในกรณีนี้, การค้นหาฟังก์ชันที่ตรงกันของ DB2 โดยปกติจะใช้ฟังก์ชันพารเพื่อให้เหมาะสมกับการอ้างอิง.

“การนิยาม UDT” ในหน้า 242

UDT จะถูกนิยามด้วยคำสั่ง CREATE DISTINCT TYPE.

การใช้การอ้างอิงฟังก์ชันที่ไม่ครบตามเกณฑ์:

คุณสามารถใช้การอ้างอิงฟังก์ชันที่ไม่ครบตามเกณฑ์แทนการอ้างอิงฟังก์ชันที่ครบตามเกณฑ์ได้. ในกรณีนี้, การค้นหาฟังก์ชันที่ตรงกันของ DB2 โดยปกติจะใช้ฟังก์ชันพารเพื่อให้เหมาะสมกับการอ้างอิง.

ในกรณีของฟังก์ชัน DROP FUNCTION หรือ COMMENT ON FUNCTION, การอ้างอิงจะถูกทำให้ครบตามเกณฑ์โดยใช้ Authorization ID ปัจจุบัน, ถ้าการอ้างนั้นไม่ครบตามเกณฑ์สำหรับการตั้งชื่อ *SQL, หรือการตั้งชื่อ *LIBL สำหรับการตั้งชื่อ *SYS. ดังนั้น, มันจึงสำคัญที่คุณจะรู้ว่าพารฟังก์ชันของคุณเป็นอะไร, และ ฟังก์ชันที่ขัดแย้งอะไร, ถ้ามีอยู่, ที่เกิดขึ้นในแบบแผนของพารฟังก์ชันในปัจจุบัน ของคุณ. ตัวอย่างเช่น, สมมติว่า คุณเป็น PABLO และคำสั่ง SQL แบบ static ของคุณเป็นดังนี้, โดยที่ COLUMN1 คือชนิดข้อมูลแบบ INTEGER:

```
SELECT BLOOP(COLUMN1) FROM T
```

คุณได้สร้างฟังก์ชัน BLOOP สองฟังก์ชันในส่วนของ การใช้การอ้างอิงฟังก์ชันตามเกณฑ์, และคุณต้องการ และคาดว่า หนึ่งในฟังก์ชันนั้นจะถูกเลือก. ถ้าดีฟอลต์ฟังก์ชันพารถูกใช้, BLOOP ตัวแรกจะถูกเลือก (เนื่องจาก COLUMN1 เป็น INTEGER), ถ้าไม่มี BLOOP ที่ขัดแย้งใน QSYS หรือ QSYS2:

```
"QSYS", "QSYS2", "PABLO"
```

อย่างไรก็ตาม, สมมุติว่าคุณลืมไปว่าคุณใช้สคริปต์สำหรับพรีคอมไพล์(precompile)และไบต์(bind)กับสิ่งที่เขียนไว้เพื่อจุดประสงค์อื่น. ในสคริปต์นี้, คุณเขียนพารามิเตอร์ SQLPATHของคุณโดยตรงเพื่อระบุฟังก์ชันพาราดต่อไปนี้ เพื่อใช้ในเหตุผลอื่นที่ไม่สามารถใช้กับงานปัจจุบันของคุณได้:

```
"KATHY", "QSYS", "QSYS2", "PABLO"
```

ถ้ามีฟังก์ชัน BLOOP ในแบบแผน KATHY, การเลือกฟังก์ชันสามารถ resolve ฟังก์ชันนั้นได้เป็นอย่างดี, และคำสั่งของคุณปฏิบัติงานโดยไม่มีข้อผิดพลาด. คุณจะไม่ได้รับการแจ้งบอก เนื่องจาก DB2 คิดว่าคุณรู้อยู่แล้วว่าคุณกำลังทำอะไร. เพราะมันเป็นความรับผิดชอบของคุณในการหาเอาต์พุตที่ผิดจากคำสั่งของคุณแล้วทำให้ถูกต้อง.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การใช้การอ้างอิงฟังก์ชันที่ครบตามเกณฑ์” ในหน้า 208

ถ้าคุณใช้การอ้างอิงฟังก์ชันที่ครบตามเกณฑ์, คุณต้องจำกัดการค้นหาสำหรับฟังก์ชันที่ตรงกันกับแบบแผนนั้น.

ข้อสรุปของการอ้างอิงฟังก์ชัน:

สำหรับการอ้างอิงฟังก์ชันทั้งแบบครบตามเกณฑ์และไม่ครบตามเกณฑ์, algorithm การเลือกฟังก์ชันจะมองหาฟังก์ชันที่ใช้ได้ทั้งหมด, ทั้งแบบในตัวและแบบผู้ใช้กำหนดเอง, ที่มีชื่อ, จำนวนของพารามิเตอร์ที่กำหนดเป็นอาร์กิวเมนต์เหมือนกัน, แต่ละพารามิเตอร์จะเหมือน หรือ สันนิษฐานชนิดของอาร์กิวเมนต์ที่ตรงกัน.

ฟังก์ชันประยุกต์ หมายถึงฟังก์ชันในแบบแผนที่ตั้งชื่อสำหรับการอ้างอิงที่ครบตามเกณฑ์, หรือฟังก์ชันในแบบแผนของฟังก์ชันพารสำหรับการอ้างอิงที่ไม่ครบตามเกณฑ์. algorithm จะมองหาฟังก์ชันที่ตรงที่สุด, หรือถ้าหาไม่เจอ, จึงค่อยหาฟังก์ชันที่ใกล้เคียงที่สุด. ฟังก์ชันพารปัจจุบันจะถูกใช้, ในกรณีของการอ้างอิงที่ไม่ครบตามเกณฑ์เท่านั้น, ซึ่งเป็นปัจจัยในการตัดสินใจ ถ้าพบฟังก์ชันที่เหมือนกันสองฟังก์ชันในแบบแผนที่แตกต่างกัน.

คุณลักษณะพิเศษที่น่าสนใจ เป็นข้อเท็จจริงว่า ฟังก์ชันการอ้างอิงสามารถซ้อนกันได้, ถึงแม้ว่า จะมีการอ้างอิงฟังก์ชันตัวเดียว. นี่เป็นความจริงสำหรับฟังก์ชันในตัว รวมถึง UDF ด้วย. อย่างไรก็ตาม, มีข้อจำกัดบางอย่าง เมื่อรวมถึงฟังก์ชันแบบคอลัมน์.

การกลั่นกรองตัวอย่างก่อนหน้านี้:

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...  
CREATE FUNCTION BLOOP (DOUBLE) RETURNS INTEGER ...
```

ดังนั้น ควรพิจารณาถึงคำสั่งต่อไปนี้:

```
SELECT BLOOP( BLOOP(COLUMN1)) FROM T
```

ถ้า COLUMN1 คือคอลัมน์ชนิด DECIMAL หรือ DOUBLE แล้ว, การอ้างอิง BLOOP ด้านในจะเรียกใช้ฟังก์ชัน BLOOP ตัวที่สองซึ่งถูกนิยามไว้ด้านบน. เนื่องจาก BLOOP นี้คืนค่าเป็น INTEGER, ดังนั้น BLOOP ด้านนอกจะเรียกใช้ BLOOP ตัวแรก.

หรืออีกทางหนึ่ง, ถ้า COLUMN1 คือคอลัมน์ชนิด SMALLINT หรือ INTEGER แล้ว, การอ้างอิง BLOOP ด้านในจะเรียกใช้ BLOOP ตัวแรกซึ่งถูกนิยามไว้ด้านบน. เนื่องจาก BLOOP นี้คืนค่าเป็น INTEGER, ดังนั้น BLOOP ด้านนอกก็เรียกใช้ BLOOP ตัวแรกด้วย. ในกรณีนี้, คุณจะมองเห็นว่ามีการอ้างอิงไปยังฟังก์ชันเดียวกันซ้อนกันอยู่.

จุดสำคัญเพิ่มเติมอีกเล็กน้อยสำหรับการอ้างอิงฟังก์ชันคือ:

- คุณสามารถนิยามฟังก์ชันด้วยชื่อของหนึ่งในตัวดำเนินการ SQL. ตัวอย่างเช่น, สมมุติว่าคุณสามารถแนบบางความหมายให้กับตัวดำเนินการ "+" สำหรับค่าที่มี Distinct Type ชนิด BOAT อยู่. คุณสามารถนิยาม UDF ดังต่อไปนี้:


```
CREATE FUNCTION "+" (BOAT, BOAT) RETURNS ...
```

Then you can write the following valid SQL statement:

```
SELECT "+"(BOAT_COL1, BOAT_COL2)
FROM BIG_BOATS
WHERE BOAT_OWNER = 'Nelson Mattos'
```

คุณไม่ได้รับสิทธิให้โหลดตัวดำเนินการที่มีเงื่อนไขในตัวมากเกินไป เช่น >, =, LIKE, IN, และอื่น, ในทำนองเดียวกันนี้.

- อัลกอริทึมการเลือกฟังก์ชันจะไม่พิจารณาบริบทของการอ้างอิงในการเลือกฟังก์ชัน. ดูที่ฟังก์ชัน BLOOP นี้, ซึ่งถูกดัดแปลงจากก่อนหน้าเล็กน้อย:

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS CHAR(10)...
```

แล้วสมมติว่าคุณเขียนคำสั่ง SELECT ดังต่อไปนี้:

```
SELECT 'ABCDEFG' CONCAT BLOOP(SMALLINT_COL) FROM T
```

เนื่องจากการจับคู่ที่เหมาะสมที่สุด, เลือกโดยใช้อาถิวเมนต์ SMALLINT, จะได้ BLOOP ตัวแรกที่นิยามไว้ด้านบน, ซึ่ง operand ตัวที่สองของ CONCAT จะแปลงเป็นชนิดข้อมูล INTEGER. คำสั่งอาจไม่ส่งกลับผลลัพธ์ที่คาดหวังไว้เนื่องจากจำนวนเต็มที่ส่งกลับจะถูกทำให้เป็น VARCHAR ก่อนจะปฏิบัติคำสั่ง CONCAT. ถ้า BLOOP ตัวแรกไม่ได้ปรากฏอยู่, BLOOP ตัวอื่นจะถูกเลือกและการเรียกใช้คำสั่งอาจจะทำได้สำเร็จ.

- UDFs สามารถนิยามให้มีพารามิเตอร์หรือมีผลลัพธ์ที่มีชนิด LOB: BLOB, CLOB, หรือ DBCLOB ได้. ระบบจะ materialize ค่า LOB ทั้งหมดใน หน่วยเก็บก่อนการเรียกใช้ฟังก์ชันใดๆ, ถึงแม้ว่าต้นทางของค่าจะเป็น ตัวแปรโฮสต์ LOB locator. ตัวอย่างเช่น, พิจารณาส່วนของแอ็พพลิเคชันภาษา C ดังต่อไปนี้:

```
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS CLOB(150K) clob150K ; /* LOB host var */
SQL TYPE IS CLOB_LOCATOR clob_locator1; /* LOB locator host var */
char string[40]; /* string host var */
EXEC SQL END DECLARE SECTION;
```

ทั้งตัวแปรโฮสต์ :clob150K หรือ :clob_locator1 จะเป็นอาถิวเมนต์ที่ต้องการสำหรับฟังก์ชันที่พารามิเตอร์ที่สอดคล้องกันถูกนิยามไว้เป็น CLOB(500K). การอ้างอิงถึง FINDSTRING ที่กำหนดไว้ใน “ตัวอย่าง: การค้นหาสตริง” ในหน้า 185 ทั้งสอง ตัวอย่างดังต่อไปนี้จะใช้ได้ในโปรแกรม:

```
... SELECT FINDSTRING (:clob150K, :string) FROM ...
... SELECT FINDSTRING (:clob_locator1, :string) FROM ...
```

- พารามิเตอร์แบบ UDF ภายนอกหรือผลลัพธ์ที่มีชนิด LOB อย่างน้อยหนึ่งชนิดสามารถ ถูกสร้างด้วย modifier ชนิด AS LOCATOR. ในกรณีนี้, ค่า LOB ทั้งหมดจะไม่ถูกดึงค่าไปเก็บก่อนที่จะเรียกใช้. แทนดังนั้น, LOB LOCATOR จะถูกส่งผ่านไปยัง UDF.

คุณยังสามารถใช้ความสามารถนั้นพารามิเตอร์ UDF หรือผลลัพธ์ที่มีชนิดเป็น Distinct Type ที่อยู่บนพื้นฐานของ LOB ได้. ความสามารถนี้จะจำกัดใช้ได้กับ UDFs แบบภายนอก. โปรดสังเกตว่าอาถิวเมนต์ของฟังก์ชันสามารถเป็นค่า LOB ของชนิดที่นิยามไว้ได้ทุกก็ได้; ไม่จำเป็นต้องเป็นตัวแปรโฮสต์ที่นิยามให้เป็นหนึ่งในชนิด LOCATOR. การใช้ Host variable Locators เป็นอาถิวเมนต์จะไม่เกี่ยวข้องใดๆกับการใช้ AS LOCATOR ในพารามิเตอร์ของ UDF และการนิยามผลลัพธ์.

- UDFs สามารถถูกนิยามโดยใช้ Distinct Types เป็นพารามิเตอร์หรือเป็นผลลัพธ์ได้. DB2 จะส่งผ่าน ค่าไปยัง UDF ในรูปแบบของชนิดข้อมูลต้นฉบับของ distinct type.

ค่า Distinct type ที่อยู่ในตัวแปรโฮสต์และที่ถูกใช้เป็นอาถิวเมนต์ของ UDF ที่มีพารามิเตอร์ที่สอดคล้องกันถูกนิยามเป็นชนิด distinct ต้องให้ผู้ใช้แปลงข้อมูลให้เป็น distinct type โดยตรง. จะไม่มีชนิดบนภาษาโฮสต์สำหรับ Distinct Type. การกำหนดชนิดที่ strong ของ DB2 มีความจำเป็นอย่างยิ่งในที่นี้. มิฉะนั้นแล้วผลลัพธ์ที่คุณได้อาจจะคลุมเคลือ. ดังนั้น, ให้

พิจารณาชนิด Distinct BOAT ที่ถูกกำหนดให้กับ BLOB ที่รับอ็อบเจกต์ของชนิด BOAT เป็นอาทิเวเมนต์. ในส่วนของแอ็พพลิเคชั่นภาษา C ดังต่อไปนี้, ตัวแปรโฮสต์ :ship จะเก็บค่า BLOB ที่จะถูกส่งผ่านไปยังฟังก์ชัน BOAT_COST:

```
EXEC SQL BEGIN DECLARE SECTION;  
SQL TYPE IS BLOB(150K) ship;  
EXEC SQL END DECLARE SECTION;
```

Both of the following statements correctly resolve to the BOAT_COST function, because both cast the :ship host variable to type BOAT:

```
... SELECT BOAT_COST (BOAT(:ship)) FROM ...  
... SELECT BOAT_COST (CAST(:ship AS BOAT)) FROM ...
```

ถ้ามี Distinct Type ชื่อ BOAT หลายตัวในฐานข้อมูล, หรือมีหลาย BOAT UDF ใน Schema อื่น, คุณต้องระมัดระวังฟังก์ชันพาราชของคุณให้ดี. มิฉะนั้นแล้วผลลัพธ์ที่คุณได้อาจจะคลุมเครือ.

ทริกเกอร์

ทริกเกอร์คือชุดของ action ที่รันโดยอัตโนมัติ เมื่อมีการเปลี่ยนแปลงที่ระบุบนตารางหรือมุมมองที่ระบุ. การเปลี่ยนแปลงดังกล่าวสามารถเป็นได้ทั้งคำสั่ง SQL INSERT, UPDATE, หรือ DELETE, หรือเป็นการแทรก, อัปเดต, หรือลบคำสั่งภาษาชั้นสูงในแอ็พพลิเคชั่นโปรแกรม. ทริกเกอร์เป็นประโยชน์สำหรับงานอย่างเช่น การบังคับใช้กฎธุรกิจ, การตรวจสอบความถูกต้องของข้อมูลอินพุต, และการเก็บหลักฐานการตรวจสอบ.

ทริกเกอร์สามารถกำหนดให้เป็น SQL หรือภายนอก.

สำหรับทริกเกอร์ใช้ภายนอก, จะมีการใช้คำสั่ง CRTPFTRG CL. โปรแกรมซึ่งประกอบด้วยชุดการทำงานของทริกเกอร์สามารถถูกกำหนดด้วยภาษาชั้นสูงใดที่สนับสนุนก็ได้. ทริกเกอร์ภายนอกสามารถแทรก, อัปเดต, ลบ, หรืออ่านทริกเกอร์.

สำหรับ SQL ทริกเกอร์, จะมีการใช้คำสั่ง CREATE TRIGGER. ทริกเกอร์โปรแกรมถูกกำหนดทั้งหมดโดยใช้ SQL. SQL ทริกเกอร์สามารถแทรก, อัปเดต, หรือลบทริกเกอร์.

| เมื่อทริกเกอร์ถูกเชื่อมโยงเข้ากับตารางหรือมุมมอง, ตัวสนับสนุนทริกเกอร์จะเรียกทริกเกอร์โปรแกรมขึ้นมาทุกครั้งที่มีการเปลี่ยนแปลงเกิดขึ้นกับตารางหรือมุมมอง, หรือโลจิคัลไฟล์หรือมุมมองใดๆ ที่สร้างขึ้นบนตารางหรือมุมมอง. คุณสามารถกำหนด SQL ทริกเกอร์และทริกเกอร์ใช้ภายนอกสำหรับตารางเดียวกันได้. เฉพาะ SQL ทริกเกอร์เท่านั้นที่สามารถกำหนดสำหรับมุมมองได้. ซึ่งสามารถกำหนดได้มากถึง 200 ทริกเกอร์สำหรับตารางหรือมุมมองเดียว.

| การเปลี่ยนแปลงตารางแต่ละครั้งสามารถเรียกทริกเกอร์ได้ก่อนหรือหลังจากที่เกิดการเปลี่ยนแปลง. นอกจากนี้, คุณสามารถเพิ่มทริกเกอร์ใช้ *อ่าน* ซึ่งจะถูกเรียกขึ้นมาทุกครั้งที่เข้าถึงตาราง. ดังนั้น, ตารางจึงสามารถเชื่อมโยงกับทริกเกอร์ได้หลายประเภท.

- ทริกเกอร์ก่อนลบ
- ทริกเกอร์ก่อนแทรก
- ทริกเกอร์ก่อนอัปเดต
- ทริกเกอร์หลังลบ
- ทริกเกอร์หลังแทรก
- ทริกเกอร์หลังอัปเดต
- ทริกเกอร์อ่านอย่างเดียว (ทริกเกอร์ใช้ภายนอกอย่างเดียว)

| การเปลี่ยนแปลงมุมมองแต่ละครั้งสามารถเรียกโปรแกรมอื่นแทนทริกเกอร์ ซึ่งจะทำงานของ action บางชุดแทนการแทรก, อัปเดต, หรือลบ. มุมมองสามารถเชื่อมโยงกับ:

- | • โปรแกรมแทนทริกเกอร์ใช้ลบ
- | • โปรแกรมแทนทริกเกอร์ใช้แทรก
- | • โปรแกรมแทนทริกเกอร์ใช้อัปเดต

ข้อมูลที่เกี่ยวข้อง

การทำทริกเกอร์เหตุการณ์แบบอัตโนมัติในฐานข้อมูลของคุณ

SQL ทริกเกอร์

| คำสั่ง SQL CREATE TRIGGER จะแสดงวิธีการสำหรับระบบจัดการฐานข้อมูล เพื่อให้ควบคุม, มอนิเตอร์, และจัดการกลุ่มตารางอย่างได้ผลทุกครั้งที่มีการแทรก, อัปเดต, หรือลบออก.

คำสั่งที่ระบุใน SQL ทริกเกอร์จะถูกเรียกใช้งานทุกครั้งที่มีการแทรก, อัปเดต, หรือลบ SQL. SQL ทริกเกอร์อาจเรียกโปรซีเจอร์ที่เก็บไว้หรือฟังก์ชันที่ผู้ใช้กำหนดให้ทำการประมวลผลเพิ่มเติมเมื่อเรียกใช้งานทริกเกอร์.

| ตรงกันข้ามกับโปรซีเจอร์ที่เก็บไว้, SQL ทริกเกอร์ไม่สามารถเรียกโดยตรงจากแอปพลิเคชันได้. แต่, SQL ทริกเกอร์จะถูกเรียกโดยระบบจัดการฐานข้อมูลเมื่อใช้การแทรก, อัปเดต, หรือลบทริกเกอร์. definition ของ SQL ทริกเกอร์จะถูกเก็บไว้ในระบบการจัดการฐานข้อมูล และถูกเรียกใช้งานโดยระบบการจัดการฐานข้อมูลเช่นกัน เมื่อตารางหรือมุมมอง SQL ที่มีการกำหนดทริกเกอร์ไว้, ถูกแก้ไข.

| คุณสามารถสร้าง SQL ทริกเกอร์ได้ด้วยการระบุคำสั่ง CREATE TRIGGER SQL. อ็อบเจกต์ทั้งหมดที่อ้างถึงคำสั่ง CREATE TRIGGER (เช่น ตาราง และฟังก์ชัน) ต้องมีอยู่; มิฉะนั้น, ทริกเกอร์จะไม่ถูกสร้าง. คำสั่งในส่วนรู่ทึนของ SQL ทริกเกอร์จะถูกแปลงโดย SQL ให้เป็นอ็อบเจกต์โปรแกรม (*PGM). โปรแกรมจะถูกสร้างในแบบแผนที่ระบุโดย qualifier ของชื่อทริกเกอร์. ทริกเกอร์ที่ระบุจะถูกลงทะเบียนในแคตตาล็อก SYSTRIGGERS, SYSTRIGDEP, SYSTRIGCOL, และ SYSTRIGUPD SQL.

หลักการที่เกี่ยวข้อง

“การดีบักรูทีน SQL” ในหน้า 225

ด้วยการระบุ SET OPTION DBGVIEW = *SOURCE ในข้อความ Create SQL Procedure, Create SQL Function, หรือ Create Trigger, คุณสามารถดีบั๊กโปรแกรมหรือโมดูลที่สร้างขึ้น ที่ระดับข้อความ SQL.

ข้อมูลที่เกี่ยวข้อง

SQL control statements

CREATE TRIGGER statement

ทริกเกอร์ BEFORE SQL:

ทริกเกอร์ BEFORE อาจไม่สามารถแก้ไขตารางได้, แต่สามารถใช้เพื่อตรวจสอบค่าอินพุตคอลัมน์, และเพื่อแก้ไขค่าคอลัมน์ที่ถูกแทรก หรืออัปเดตในตาราง.

ในตัวอย่างต่อไปนี้, ทริกเกอร์ถูกใช้เพื่อกำหนดปีการเงินรายไตรมาสสำหรับบริษัท ก่อนการแทรกแถวลงในตารางเป้าหมาย.

```
CREATE TABLE TransactionTable (DateOfTransaction DATE, FiscalQuarter SMALLINT)
```

```
CREATE TRIGGER TransactionBeforeTrigger BEFORE INSERT ON TransactionTable
```

```

REFERENCING NEW AS new_row
FOR EACH ROW MODE DB2ROW
BEGIN
  DECLARE newmonth SMALLINT;
SET newmonth = MONTH(new_row.DateOfTransaction);
  IF newmonth < 4 THEN
    SET new_row.FiscalQuarter=3;
  ELSEIF newmonth < 7 THEN
    SET new_row.FiscalQuarter=4;
  ELSEIF newmonth < 10 THEN
    SET new_row.FiscalQuarter=1;
  ELSE
    SET new_row.FiscalQuarter=2;
  END IF;
END

```

ในส่วนคำสั่งแทรก SQL ข้างล่างนี้, คอลัมน์ "FiscalQuarter" ควรถูกกำหนดเป็น 2, ถ้าวันที่ปัจจุบันคือ 14 พฤศจิกายน, 2000.

```

INSERT INTO TransactionTable(DateOfTransaction)
VALUES(CURRENT DATE)

```

SQL ทริกเกอร์ได้เข้าถึงและสามารถใช้ User-defined Distinct Types (UDTs) และโพรซีเจอร์ที่เก็บไว้. ในตัวอย่างต่อไปนี้, SQL ทริกเกอร์จะเรียกโพรซีเจอร์ที่เก็บไว้ขึ้นมา เพื่อเรียกใช้งานตารางธุรกิจบางข้อซึ่งได้ถูกกำหนดไว้ก่อน, ในกรณีนี้, เพื่อตั้งคอลัมน์ให้เป็นค่าซึ่งได้ถูกกำหนดไว้ก่อนสำหรับธุรกิจ.

```

CREATE DISTINCT TYPE enginesize AS DECIMAL(5,2) WITH COMPARISONS

```

```

CREATE DISTINCT TYPE engineclass AS VARCHAR(25) WITH COMPARISONS

```

```

CREATE PROCEDURE SetEngineClass(IN SizeInLiters enginesize,
OUT CLASS engineclass)

```

```

LANGUAGE SQL CONTAINS SQL

```

```

BEGIN

```

```

  IF SizeInLiters<2.0 THEN

```

```

    SET CLASS = 'Mouse';

```

```

  ELSEIF SizeInLiters<3.1 THEN

```

```

    SET CLASS = 'Economy Class';

```

```

  ELSEIF SizeInLiters<4.0 THEN

```

```

    SET CLASS = 'Most Common Class';

```

```

  ELSEIF SizeInLiters<4.6 THEN

```

```

    SET CLASS = 'Getting Expensive';

```

```

  ELSE

```

```

    SET CLASS = 'Stop Often for Fillups';

```

```

  END IF;

```

```

END

```

```

CREATE TABLE EngineRatings (VariousSizes enginesize, ClassRating engineclass)

```

```

CREATE TRIGGER SetEngineClassTrigger BEFORE INSERT ON EngineRatings

```

```

REFERENCING NEW AS new_row

```

```

FOR EACH ROW MODE DB2ROW

```

```

  CALL SetEngineClass(new_row.VariousSizes, new_row.ClassRating)

```

สำหรับคำสั่งแทรก SQL ข้างล่างนี้, คอลัมน์ "ClassRating" ถูกกำหนดเป็น "Economy Class", ถ้าคอลัมน์ "VariousSizes" มีค่าเท่ากับ 3.0.

```
INSERT INTO EngineRatings(VariousSizes) VALUES(3.0)
```

SQL กำหนดให้ต้องมีตาราง, ฟังก์ชันที่ผู้ใช้กำหนด, โพรซีเจอร์ และประเภทที่ผู้ใช้กำหนดทั้งหมดขึ้นมา ก่อนจะมีการสร้าง SQL ทริกเกอร์. ในตัวอย่างข้างบน, ตาราง, โพรซีเจอร์ที่เก็บไว้, และประเภทที่ผู้ใช้กำหนดทั้งหมดจะถูกกำหนดก่อนที่จะสร้างทริกเกอร์.

ทริกเกอร์ AFTER SQL:

คุณสามารถใช้เงื่อนไข WHEN ใน SQL ทริกเกอร์เพื่อระบุเงื่อนไขได้. หากเงื่อนไขประเมินผลว่าถูกต้อง, คำสั่ง SQL ในส่วนรูทีนของ SQL ทริกเกอร์จะถูกรัน. หากเงื่อนไขประเมินผลว่าผิด, คำสั่ง SQL ในส่วนของรูทีนของ SQL ทริกเกอร์จะไม่ถูกรัน, และการควบคุมจะถูกส่งกลับไปที่ระบบฐานข้อมูล. ชนิดของทริกเกอร์นี้เรียกว่า ทริกเกอร์ AFTER.

ในตัวอย่างต่อไปนี้, จะมีการประเมินผลเดียวเพื่อตัดสินว่า ควรมีการรันคำสั่งในส่วนรูทีนของทริกเกอร์หรือไม่ เมื่อทริกเกอร์ถูกเรียกทำงาน.

```
CREATE TABLE TodaysRecords(TodaysMaxBarometricPressure FLOAT,  
    TodaysMinBarometricPressure FLOAT)
```

```
CREATE TABLE OurCitysRecords(RecordMaxBarometricPressure FLOAT,  
    RecordMinBarometricPressure FLOAT)
```

```
CREATE TRIGGER UpdateMaxPressureTrigger  
AFTER UPDATE OF TodaysMaxBarometricPressure ON TodaysRecords  
REFERENCING NEW AS new_row  
FOR EACH ROW MODE DB2ROW  
WHEN (new_row.TodaysMaxBarometricPressure >  
    (SELECT MAX(RecordMaxBarometricPressure) FROM  
    OurCitysRecords))  
UPDATE OurCitysRecords  
    SET RecordMaxBarometricPressure =  
        new_row.TodaysMaxBarometricPressure
```

```
CREATE TRIGGER UpdateMinPressureTrigger  
AFTER UPDATE OF TodaysMinBarometricPressure  
ON TodaysRecords  
REFERENCING NEW AS new_row  
FOR EACH ROW MODE DB2ROW  
WHEN(new_row.TodaysMinBarometricPressure <  
    (SELECT MIN(RecordMinBarometricPressure) FROM  
    OurCitysRecords))  
UPDATE OurCitysRecords  
    SET RecordMinBarometricPressure =  
        new_row.TodaysMinBarometricPressure
```

ก่อนอื่นค่าปัจจุบันจะถูก initialize สำหรับตาราง.

```
INSERT INTO TodaysRecords VALUES(0.0,0.0)  
INSERT INTO OurCitysRecords VALUES(0.0,0.0)
```

ในส่วนคำสั่งอัปเดต SQL ข้างล่างนี้, RecordMaxBarometricPressure ใน OurCitysRecords จะถูกอัปเดตโดย UpdateMaxPressureTrigger.

```
UPDATE TodaysRecords SET TodaysMaxBarometricPressure = 29.95
```

แต่ในอนาคต, หาก Today'sMaxBarometricPressure เท่ากับ 29.91 เท่านั้น, RecordMaxBarometricPressure จะไม่ได้รับการอัปเดต.

```
UPDATE Today'sRecords SET Today'sMaxBarometricPressure = 29.91
```

SQL จะยอมรับ definition ของทริกเกอร์สำหรับการทำงานของทริกเกอร์แบบเดี่ยว. ในตัวอย่างก่อนหน้า, มีทริกเกอร์ AFTER UPDATE สองประเภทได้แก่: UpdateMaxPressureTrigger และ UpdateMinPressureTrigger. ทริกเกอร์ทั้งสองจะปฏิบัติงานก็ต่อเมื่อมีการอัปเดตคอลัมน์เฉพาะของตาราง Today'sRecords.

ทริกเกอร์ AFTER อาจแก้ไขตาราง. ในตัวอย่างข้างบน, การดำเนินการ UPDATE จะถูกนำมาใช้กับตารางที่สอง. โปรดสังเกตว่าควรหลีกเลี่ยงการแทรกและการอัปเดตแบบเรียกซ้ำ. ระบบจัดการฐานข้อมูลจะจบการทำงาน หากถึงระดับการซ้อนภายในสูงสุดของทริกเกอร์. คุณสามารถหลีกเลี่ยงการเรียกซ้ำได้ด้วยการเพิ่มตรรกะแบบมีเงื่อนไข เพื่อที่จะออกจากการแทรกหรืออัปเดตก่อนที่จะถึงระดับการซ้อนภายในสูงสุด. คุณควรหลีกเลี่ยงสถานการณ์เดียวกันนี้ในเครือข่ายทริกเกอร์ซึ่งมีการต่อเรียงแบบเรียกซ้ำผ่านเครือข่ายทริกเกอร์.

ทริกเกอร์ INSTEAD OF SQL:

ทริกเกอร์ INSTEAD OF คือ SQL ทริกเกอร์ที่ถูกประมวลผล “แทนที่” คำสั่ง SQL UPDATE, DELETE หรือ INSERT. ไม่เหมือนกับทริกเกอร์ SQL BEFORE และ AFTER, ทริกเกอร์ INSTEAD OF อาจถูกกำหนดไว้บนมุมมอง, ไม่ใช่ตาราง.

ทริกเกอร์ INSTEAD OF อนุญาตให้มุมมอง, ซึ่งไม่ใช่ความสามารถในการแทรก, ความสามารถในการอัปเดต, หรือความสามารถในการลบ, ถูกแทรก, อัปเดต, หรือลบทั้ง. โปรดดู CREATE VIEW สำหรับข้อมูลเพิ่มเติมเกี่ยวกับความสามารถในการลบ, ความสามารถในการอัปเดต, และความสามารถในการแทรกมุมมอง.

หลังจากที่ทริกเกอร์ SQL INSTEAD OF ถูกเพิ่มไปยังมุมมองแล้ว, มุมมองซึ่งก่อนหน้านี้สามารถอ่านได้อย่างเดียว ก็จะสามารถใช้เพื่อแทรก, อัปเดต, หรือลบการดำเนินการ. ทริกเกอร์ INSTEAD OF จะกำหนดการดำเนินงานที่จำเป็นต้องทำเพื่อคงไว้ซึ่งมุมมอง.

มุมมองสามารถใช้เพื่อควบคุมการเข้าถึงตาราง. ทริกเกอร์ INSTEAD OF สามารถดูแลรักษาการควบคุมการเข้าถึงตารางได้โดยง่าย.

การใช้ทริกเกอร์ INSTEAD OF

definition ของมุมมอง V1 ต่อไปนี้ คือ สามารถอัปเดตได้, สามารถลบได้, และสามารถแทรกได้:

```
CREATE TABLE T1 (C1 VARCHAR(10), C2 INT)
CREATE VIEW V1(X1) AS SELECT C1 FROM T1 WHERE C2 > 10
```

สำหรับคำสั่งการแทรก, C1 ลงในตาราง T1 จะถูกกำหนดค่าเป็น 'A'. C2 จะถูกกำหนดค่า NULL. ค่า NULL จะเป็นสาเหตุทำให้แถวใหม่ที่ไม่ตรงกับเงื่อนไขการเลือก C2 > 10 สำหรับมุมมอง V1.

```
INSERT INTO V1 VALUES('A')
```

การเพิ่มทริกเกอร์ INSTEAD OF IOT1 สามารถแสดงค่าที่แตกต่างกันสำหรับแถวที่จะถูกเลือกโดยมุมมอง. :

```
CREATE TRIGGER IOT1 INSTEAD OF INSERT ON V1
REFERENCING NEW AS NEW_ROW
FOR EACH ROW MODE DB2SQL
INSERT INTO T1 VALUES(NEW_ROW.X1, 15)
```

| การทำให้มุมมองสามารถลบได้

| definition ของมุมมองรวม V3 ไม่สามารถอัปเดตได้, ไม่สามารถลบได้, หรือไม่สามารถแทรกได้:

```
| CREATE TABLE A (A1 VARCHAR(10), A2 INT)
| CREATE VIEW V1(X1) AS SELECT A1 FROM A
|
| CREATE TABLE B (B1 VARCHAR(10), B2 INT)
| CREATE VIEW V2(Y1) AS SELECT B1 FROM B
|
| CREATE VIEW V3(Z1, Z2) AS SELECT V1.X1, V2.Y1 FROM V1, V2 WHERE V1.X1 = 'A' AND V2.Y1 > 'B'
```

| การเพิ่มทริกเกอร์ INSTEAD OF IOT2, จะทำให้มุมมอง V3 สามารถลบได้:

```
| CREATE TRIGGER IOT2 INSTEAD OF DELETE ON V3
| REFERENCING OLD AS OLD_ROW
| FOR EACH ROW MODE DB2SQL
| BEGIN
|   DELETE FROM A WHERE A1 = OLD_ROW.Z1;
|   DELETE FROM B WHERE B1 = OLD_ROW.Z2;
| END
```

| ด้วยทริกเกอร์นี้, คำสั่ง DELETE ต่อไปนี้จะสามารถใช้ได้. คำสั่งนี้จะลบแถวทั้งหมดออกจากตาราง A โดยที่ A1 มีค่าเป็น 'A', และ แถวทั้งหมดจากตาราง B โดยที่ B1 มีค่า 'X'.

```
| DELETE FROM V3 WHERE Z1 = 'A' AND Z2 = 'X'
```

| ทริกเกอร์ INSTEAD OF ที่มีมุมมองจะถูกกำหนดไว้บนมุมมอง

| definition ของมุมมอง V2 ต่อไปนี้ที่ถูกกำหนดไว้บน V1 จะไม่สามารถแทรกได้, ไม่สามารถอัปเดตได้, หรือไม่สามารถลบได้:

```
| CREATE TABLE T1 (C1 VARCHAR(10), C2 INT)
| CREATE TABLE T2 (D1 VARCHAR(10), D2 INT)
| CREATE VIEW V1(X1, X2) AS SELECT C1, C2 FROM T1
| UNION SELECT D1, D2 FROM T2
|
| CREATE VIEW V2(Y1, Y2) AS SELECT X1, X2 FROM V1
```

| การเพิ่มทริกเกอร์ INSTEAD OF IOT1 ลงใน V1, จะไม่ได้ทำให้ V2 สามารถอัปเดตได้:

```
| CREATE TRIGGER IOT1 INSTEAD OF UPDATE ON V1
| REFERENCING OLD AS OLD_ROW NEW AS NEW_ROW
| FOR EACH ROW MODE DB2SQL
| BEGIN
|   UPDATE T1 SET C1 = NEW_ROW.X1, C2 = NEW_ROW.X2 WHERE
|     C1 = OLD_ROW.X1 AND C2 = OLD_ROW.X2;
|   UPDATE T2 SET D1 = NEW_ROW.X1, D2 = NEW_ROW.D2 WHERE
|     D1 = OLD_ROW.X1 AND D2 = OLD_ROW.X2;
| END
```

| มุมมอง V2 ยังคงไม่สามารถอัปเดตได้ เนื่องจาก definition ของมุมมอง V2 แบบเดิมยังคงไม่สามารถอัปเดตได้.

| การใช้ทริกเกอร์ **INSTEAD OF** พร้อมด้วยทริกเกอร์ **BEFORE** และ **AFTER**

| การเพิ่มของทริกเกอร์ **INSTEAD OF** ในมุมมองไม่ได้เป็นสาเหตุทำให้เกิดความขัดแย้งใดๆ กับทริกเกอร์ **BEFORE** และ **AFTER** ที่ถูกกำหนดบนตารางหลัก:

```
| CREATE TABLE T1 (C1 VARCHAR(10), C2 DATE)
| CREATE TABLE T2 (D1 VARCHAR(10))
|
| CREATE TRIGGER AFTER1 AFTER DELETE ON T1
| REFERENCING OLD AS OLD_ROW
| FOR EACH ROW MODE DB2SQL
|   DELETE FROM T2 WHERE D1 = OLD_ROW.C1
|
| CREATE VIEW V1(X1, X2) AS SELECT SUBSTR(T1.C1, 1, 1), DAYOFWEEK_ISO(T1.C2) FROM T1
|
| CREATE TRIGGER IOT1 INSTEAD OF DELETE ON V1
| REFERENCING OLD AS OLD_ROW
| FOR EACH ROW MODE DB2SQL
|   DELETE FROM T1 WHERE C1 LIKE (OLD_ROW.X1 CONCAT '%')
```

| การดำเนินการลบใดๆ สำหรับมุมมอง V1 เป็นผลทำให้ทริกเกอร์ **AFTER DELETE AFTER1** ถูกเรียกใช้งานด้วย เนื่องจากทริกเกอร์ **IOT1** จะทำการลบบนตาราง T1. การลบตาราง T1 เป็นสาเหตุทำให้ทริกเกอร์ **AFTER1** ถูกเรียกใช้.

| มุมมองที่ถูกอ้างอิง และทริกเกอร์ **INSTEAD OF**

| เมื่อเพิ่มทริกเกอร์ **INSTEAD OF** ลงในมุมมอง, หาก definition ของมุมมองอ้างอิงกับมุมมองที่มีทริกเกอร์ **INSTEAD OF** กำหนดอยู่ด้วย, คุณควรจะกำหนดทริกเกอร์ **INSTEAD OF** สำหรับการดำเนินงานสามอย่างคือ, **UPDATE**, **DELETE**, และ **INSERT**, เพื่อหลีกเลี่ยงความสับสนว่า ความสามารถใดที่มุมมองถูกกำหนดให้มี กับความสามารถใดที่มุมมองที่ถูกอ้างอิงต้องมี.

Handlers ใน SQL ทริกเกอร์:

handler ใน SQL ทริกเกอร์เพิ่มความสามารถให้กับ SQL ทริกเกอร์ในการกู้คืนจากข้อผิดพลาด หรือข้อมูลบันทึกการทำงานเกี่ยวกับข้อผิดพลาดที่เกิดขึ้น ขณะที่ประมวลผลคำสั่ง SQL ในส่วนรูทีนของทริกเกอร์.

ในตัวอย่างต่อไปนี้, มีการกำหนด handler 2 อย่าง ได้แก่: handler แรกเพื่อจัดการกับสภาวะโอเวอร์โฟลว์ และ handler ที่สองเพื่อจัดการกับ SQL exception.

```
CREATE TABLE ExcessInventory(Description VARCHAR(50), ItemWeight SMALLINT)

CREATE TABLE YearToDateTotals(TotalWeight SMALLINT)

CREATE TABLE FailureLog(Item VARCHAR(50), ErrorMessage VARCHAR(50), ErrorCode INT)

CREATE TRIGGER InventoryDeleteTrigger
AFTER DELETE ON ExcessInventory
REFERENCING OLD AS old_row
FOR EACH ROW MODE DB2ROW
BEGIN
  DECLARE sqlcode INT;
  DECLARE invalid_number condition FOR '22003';
  DECLARE exit handler FOR invalid_number
```



```

INSERT INTO FailureLog VALUES(old_row.Description,
    'Overflow occurred in YearToDateTotals', sqlcode);
DECLARE exit handler FOR sqlexception
INSERT INTO FailureLog VALUES(old_row.Description,
    'SQL Error occurred in InventoryDeleteTrigger', sqlcode);
UPDATE YearToDateTotals SET TotalWeight=TotalWeight +
    old_row.itemWeight;
END

```

ก่อนอื่น, ค่าปัจจุบันสำหรับตารางจะถูก initialize.

```

INSERT INTO ExcessInventory VALUES('Desks',32500)
INSERT INTO ExcessInventory VALUES('Chairs',500)
INSERT INTO YearToDateTotals VALUES(0)

```

เมื่อคำสั่งการลบ SQL คำสั่งแรกข้างล่างนี้ถูกเรียกใช้งาน, ItemWeight สำหรับไอเท็ม "Desks" จะถูกเพิ่มเข้าไปในคอลัมน์ทั้งหมดสำหรับ TotalWeight ในตาราง YearToDateTotals. เมื่อคำสั่งการลบ SQL คำสั่งที่สองถูกเรียกใช้งาน, จะเกิดการโอเวอร์โฟลว์ขึ้นเมื่อ ItemWeight สำหรับไอเท็ม "Chairs" ถูกเพิ่มเข้าไปในคอลัมน์ทั้งหมดสำหรับ TotalWeight, เนื่องจากคอลัมน์จะจัดการเฉพาะค่าสูงสุดเท่ากับ 32767. เมื่อเกิดการโอเวอร์โฟลว์ขึ้น, exit handler ซึ่งมีหมายเลขที่ไม่ถูกต้องจะถูกเรียกใช้งาน และแถวจะถูกบันทึกลงในตาราง FailureLog. ตัวอย่างเช่น, sqlexception exit handler จะทำงาน, หากตาราง YearToDateTotals ถูกลบออกโดยบังเอิญ. ในตัวอย่างนี้, handler จะถูกใช้ในการบันทึกการทำงานเพื่อให้วินิจฉัยปัญหาได้ในภายหลัง.

```

DELETE FROM ExcessInventory WHERE Description='Desks'
DELETE FROM ExcessInventory WHERE Description='Chairs'

```

ตารางการถ่ายโอน SQL ทริกเกอร์:

SQL ทริกเกอร์อาจต้องอ้างอิงถึงแถวทั้งหมดที่ได้รับผลกระทบสำหรับการแทรก, อัปเดต, หรือลบ SQL. สิ่งนี้เป็นเรื่องที่ต้อง, ตัวอย่างเช่น, หากทริกเกอร์จำเป็นต้องใช้ฟังก์ชันแบบรวม, อย่างเช่น MIN หรือ MAX, กับคอลัมน์เฉพาะของแถวที่ได้รับผลกระทบ. คุณสามารถใช้ตารางการถ่ายโอน OLD_TABLE และ NEW_TABLE เพื่อการนี้ได้.

ในตัวอย่างต่อไปนี่, ทริกเกอร์จะใช้ฟังก์ชัน MAX แบบรวมกับแถวทั้งหมดของตาราง StudentProfiles ที่ได้รับผลกระทบ.

```

CREATE TABLE StudentProfiles(StudentsName VARCHAR(125),
    StudentsYearInSchool SMALLINT, StudentsGPA DECIMAL(5,2))

CREATE TABLE CollegeBoundStudentsProfile
    (YearInSchoolMin SMALLINT, YearInSchoolMax SMALLINT, StudentGPAMin
    DECIMAL(5,2), StudentGPAMax DECIMAL(5,2))

CREATE TRIGGER UpdateCollegeBoundStudentsProfileTrigger
AFTER UPDATE ON StudentProfiles
REFERENCING NEW_TABLE AS ntable
FOR EACH STATEMENT MODE DB2SQL
BEGIN
    DECLARE maxStudentYearInSchool SMALLINT;
    SET maxStudentYearInSchool =
        (SELECT MAX(StudentsYearInSchool) FROM ntable);
    IF maxStudentYearInSchool >
        (SELECT MAX (YearInSchoolMax) FROM
            CollegeBoundStudentsProfile) THEN

```

```

UPDATE CollegeBoundStudentsProfile SET YearInSchoolMax =
maxStudentYearInSchool;
END IF;
END

```

ในตัวอย่างก่อนหน้านี้, ทรริกเกอร์จะถูกเรียกใช้งานหนึ่งครั้ง หลังจากประมวลผลคำสั่งอัปเดตทรริกเกอร์ เนื่องจากถูกกำหนดให้เป็นทรริกเกอร์ FOR EACH STATEMENT. คุณจำเป็นต้องพิจารณาการประมวลผลทั่วไปที่กำหนดโดยระบบจัดการฐานข้อมูลสำหรับการบรรจุตารางการถ่ายโอน เมื่อคุณกำหนดทรริกเกอร์ซึ่งอ้างอิงตารางการถ่ายโอน.

ทรริกเกอร์ใช้ภายนอก

สำหรับทรริกเกอร์ภายนอก, คุณสามารถกำหนดโปรแกรมซึ่งมีชุด action ของทรริกเกอร์ในภาษาชั้นสูงที่สนับสนุน ซึ่งใช้สร้างอ็อบเจกต์ *PGM.

ทรริกเกอร์โปรแกรมสามารถมี SQL ที่ใส่อยู่ในทรริกเกอร์นั้น. เพื่อกำหนดทรริกเกอร์ภายนอก, คุณต้องสร้างทรริกเกอร์โปรแกรมและเพิ่มลงในตารางโดยใช้คำสั่ง CL ADDPFTRG CL หรือคุณสามารถเพิ่มลงในตารางโดยใช้ iSeries Navigator. การเพิ่มทรริกเกอร์ให้กับตาราง, คุณต้อง:

- จำแนกตาราง
- จำแนกประเภทการดำเนินการ
- จำแนกโปรแกรมซึ่งทำงานตามที่ต้องการ.

ข้อมูลที่เกี่ยวข้อง

การทำทรริกเกอร์เหตุการณ์แบบอัตโนมัติในฐานข้อมูลของคุณ

โปรแกรมตัวอย่างทรริกเกอร์ใช้ภายนอก:

หัวข้อนี้ประกอบด้วยตัวอย่างทรริกเกอร์โปรแกรมภายนอก, ซึ่งถูกบันทึกลงใน ILE C, ด้วย SQL ที่ฝังไว้.

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

ตัวอย่างทรริกเกอร์โปรแกรม

```

#include "string.h"
#include "stdlib.h"
#include "stdio.h"
#include <recio.h>
#include <xxcvt.h>
#include "qsysinc/h/trgbuf" /* Trigger input parameter */
#include "libl/csrm/msghand1" /* User defined message handler */
/*****
/* This is a trigger program which is called whenever there is an
/* update to the EMPLOYEE table. If the employee's commission is
/* greater than the maximum commission, this trigger program will
/* increase the employee's salary by 1.04 percent and insert into
/* the RAISE table.
/*
/* The EMPLOYEE record information is passed from the input parameter*
/* to this trigger program.
*****/

```

```

Qdb_Trigger_Buffer_t *hstruct;
char *datapt;

/*****
/* Structure of the EMPLOYEE record which is used to */
/* store the old or the new record that is passed to */
/* this trigger program. */
/* */
/* Note : You must ensure that all the numeric fields */
/* are aligned at 4 byte boundary in C. */
/* Used either Packed struct or filler to reach */
/* the byte boundary alignment. */
*****/

_Packed struct rec{
    char empn[6];
    _Packed struct { short fstlen ;
                    char fstnam[12];
                    } fstname;
    char minit[1];
    _Packed struct { short lstlen;
                    char lstnam[15];
                    } lstname;
    char dept[3];
    char phone[4];
    char hdate[10];
    char jobn[8];
    short edclvl;
    char sex1[1];
    char bdate[10];
    decimal(9,2) salary1;
    decimal(9,2) bonus1;
    decimal(9,2) comm1;
    } oldbuf, newbuf;
EXEC SQL INCLUDE SQLCA;

main(int argc, char **argv)
{
int i;
int obufoff; /* old buffer offset */
int nuloff; /* old null byte map offset */
int nbufoff; /* new buffer offset */
int nul2off; /* new null byte map offset */
short work_days = 253; /* work days during in one year */
decimal(9,2) commission = 2000.00; /* cutoff to qualify for */
decimal(9,2) percentage = 1.04; /* raised salary as percentage */
char raise_date[12] = "1982-06-01"; /* effective raise date */

struct {
    char empno[6];
    char name[30];
    decimal(9,2) salary;
    decimal(9,2) new_salary;
    } rpt1;

```

```

/*****/
/* Start to monitor any exception.          */
/*****/

    _FEEDBACK fc;
    _HDLR_ENTRY hdlr = main_handler;
        /*****/
        /* Make the exception handler active.  */
        /*****/
CEEHDLR(&hdlr, NULL, &fc);
        /*****/
                                /* Ensure exception handler OK          */
        /*****/

if (fc.MsgNo != CEE0000)
{
    printf("Failed to register exception handler.\n");
    exit(99);
};

/*****/
/* Move the data from the trigger buffer to the local */
/* structure for reference.                          */
/*****/

hstruct = (Qdb_Trigger_Buffer_t *)argv[1];
datapt = (char *) hstruct;

obufoff = hstruct ->Old_Record_Offset;    /* old buffer */
memcpy(&oldbuf,datapt+obufoff,; hstruct->Old_Record_Len);

nbufoff = hstruct ->New_Record_Offset;    /* new buffer */
memcpy(&newbuf,datapt+nbufoff,; hstruct->New_Record_Len);

EXEC SQL WHENEVER SQLERROR GO TO ERR_EXIT;

/*****/
/* Set the transaction isolation level to the same as */
/* the application based on the input parameter in the */
/* trigger buffer.                                    */
/*****/

if(strcmp(hstruct->Commit_Lock_Level,"0") == 0)
    EXEC SQL SET TRANSACTION ISOLATION LEVEL NONE;
else{
    if(strcmp(hstruct->Commit_Lock_Level,"1") == 0)
        EXEC SQL SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED, READ
            WRITE;
        else {
            if(strcmp(hstruct->Commit_Lock_Level,"2") == 0)
                EXEC SQL SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
            else
                if(strcmp(hstruct->Commit_Lock_Level,"3") == 0)
                    EXEC SQL SET TRANSACTION ISOLATION LEVEL ALL;
        }
}
}

```

```

/*****
/* If the employee's commission is greater than maximum */
/* commission, then increase the employee's salary      */
/* by 1.04 percent and insert into the RAISE table.      */
*****/

if (newbuf.comm1 >= commission)
{
    EXEC SQL SELECT EMPNO, EMPNAME, SALARY
                INTO :rpt1.empno, :rpt1.name, :rpt1.salary
                FROM TRGPERF/EMP_ACT
                WHERE EMP_ACT.EMPNO=:newbuf.empn ;

    if (sqlca.sqlcode == 0) then
    {
        rpt1.new_salary = salary * percentage;
        EXEC SQL INSERT INTO TRGPERF/RAISE VALUES(:rpt1);
    }
    goto finished;
}
err_exit:
    exit(1);

    /* All done */
finished:
    return;
} /* end of main line */

/*****
/* INCLUDE NAME : MSGHAND1                               */
/*                                                       */
/* DESCRIPTION : Message handler to signal an exception to */
/*               the application to inform that an        */
/*               error occurred in the trigger program.    */
/*                                                       */
/* NOTE : This message handler is a user defined routine. */
/*                                                       */
*****/
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#include <leawi.h>

#pragma linkage (QMHSNDPM, OS)
void QMHSNDPM(char *, /* Message identifier */
              void *, /* Qualified message file name */
              void *, /* Message data or text */
              int, /* Length of message data or text */
              char *, /* Message type */
              char *, /* Call message queue */
              int, /* Call stack counter */
              void *, /* Message key */
              void *, /* Error code */
              ...); /* Optionals:

length of call message queue
name

```

Call stack entry qualification
display external messages
screen wait time */

```

/*****
/***** This is the start of the exception handler function. */
/*****
void main_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc,
                 _FEEDBACK *new)
{
    /***/
    /** Initialize variables for call to */
    /** QMHSNDPM. */
    /** User must create a message file and */
    /** define a message ID to match the */
    /** following data. */
    /***/
char    message_id[7] = "TRG9999";
char    message_file[20] = "MSGF    LIB1    ";
char    message_data[50] = "Trigger error    " ;
    int    message_len = 30;
char    message_type[10] = "*ESCAPE ";
char    message_q[10] = "_C_pep ";
    int    pgm_stack_cnt = 1;
char    message_key[4];

    /***/
    /** Declare error code structure for */
    /** QMHSNDPM. */
    /***/
struct error_code {
    int bytes_provided;
    int bytes_available;
    char message_id[7];
} error_code;

error_code.bytes_provided = 15;
    /***/
    /** Set the error handler to resume and */
    /** mark the last escape message as */
    /** handled. */
    /***/
*rc = CEE_HDLR_RESUME;

    /***/
    /** Send my own *ESCAPE message. */
    /***/
QMHSNDPM(message_id,
          &message_file,
          &message_data,
          message_len,
          message_type,
          message_q,
          pgm_stack_cnt,
          &message_key,
          &error_code );
    /***/
    /** Check that the call to QMHSNDPM */
    /***/

```

```

/* finished correctly. */
/*****/
if (error_code.bytes_available != 0)
{
    printf("Error in QMHOVPM : %s\n", error_code.message_id);
}
}

```

การดีบักรูทีน SQL

ด้วยการระบุ SET OPTION DBGVIEW = *SOURCE ในข้อความ Create SQL Procedure, Create SQL Function, หรือ Create Trigger, คุณสามารถดีบั๊กโปรแกรมหรือโมดูลที่สร้างขึ้น ที่ระดับข้อความ SQL.

และคุณยังสามารถระบุ DBGVIEW(*SOURCE) เป็นพารามิเตอร์บนคำสั่ง RUNSQLSTM และจะนำมาใช้กับรูทีนทั้งหมดที่อยู่ภายใน RUNSQLSTM.

มุมมองซอร์สจะถูกสร้างขึ้นโดยระบบที่มาจากโครงสร้างเดิม โดยไว้อยู่ในซอร์สไฟล์ QSQDSRC ในไลบรารีของรูทีน. แต่ถ้าไม่สามารถกำหนดไลบรารีได้, QSQDSRC จะถูกสร้างอยู่ใน QTEMP. มุมมองซอร์สไม่ได้ถูกบันทึก ด้วยโปรแกรมหรือเซอวิวิโปรแกรม. มุมมองนั้นจะถูกแยกออกเป็นบรรทัดซึ่งตรงกับที่ที่คุณสามารถหยุดดีบั๊กได้. ข้อความ, รวมทั้งพารามิเตอร์และชื่อตัวแปร, จะถูกปิดด้วยตัวพิมพ์ใหญ่.

ตัวแปรและพารามิเตอร์ทั้งหมดถูกสร้างขึ้นให้เป็นส่วนหนึ่งของโครงสร้าง. ต้องใช้ชื่อโครงสร้าง เมื่อประเมินผลตัวแปรในดีบั๊ก. ตัวแปรจะเป็นไปตามเกณฑ์ด้วยการใช้ชื่อเลเบลปัจจุบัน. พารามิเตอร์จะเป็นไปตามเกณฑ์ด้วยการใช้โพสิเตอร์หรือชื่อฟังก์ชัน. ตัวแปรการส่งผ่านในทริกเกอร์ จะเป็นไปตามเกณฑ์ด้วยการใช้ข้อความสัมพันธ์ที่เหมาะสม. ขอแนะนำให้คุณระบุชื่อเลเบลสำหรับแต่ละข้อความผสมหรือข้อความ FOR. หากคุณไม่ได้ระบุชื่อ, ระบบจะสร้างชื่อให้คุณเอง. ซึ่งจะทำให้ใกล้เคียงต่อการประเมินผลตัวแปร. โปรดจำไว้ว่าพารามิเตอร์และตัวแปรทั้งหมดจะต้องถูกประเมินผลเป็นชื่อตัวพิมพ์ใหญ่. และคุณยังสามารถ ประเมินผลชื่อของโครงสร้างได้ด้วย. ซึ่งจะแสดงตัวแปรทั้งหมดภายในโครงสร้าง. หากตัวแปรหรือพารามิเตอร์เป็นศูนย์, ตัวบ่งชี้สำหรับตัวแปรหรือพารามิเตอร์นั้นก็จะตามหลังตัวแปรหรือพารามิเตอร์นั้นในโครงสร้างทันที.

เนื่องจากรูทีน SQL ถูกสร้างขึ้นใน C, จึงมีข้อจำกัดบางอย่างใน C ที่ส่งผลต่อการดีบั๊กซอร์ส SQL. ชื่อที่ถูกค้นซึ่งระบุในโครงสร้าง SQL ไม่สามารถระบุใน C ได้. ชื่อต่างๆ ถูกสร้างขึ้นสำหรับชื่อเหล่านี้, ซึ่งทำให้ยากต่อการดีบั๊กหรือประเมินผล. ในการประเมินผลเนื้อหาของ ตัวแปรอักขระใดๆ, ให้ระบุ * ก่อนชื่อตัวแปร.

เนื่องจากระบบจะสร้างตัวบ่งชี้สำหรับชื่อตัวแปรและพารามิเตอร์ส่วนใหญ่, จึงไม่มีทางที่จะตรวจสอบโดยตรงเพื่อดูว่าตัวแปร มีค่า SQL ที่เป็นศูนย์หรือไม่. การประเมินผลตัวแปรจะแสดงค่าเสมอ, แม้ว่าจะมีการตั้งตัวบ่งชี้ให้แสดงค่าศูนย์ก็ตาม.

ในการพิจารณาว่า handler ถูกเรียกขึ้นมาหรือไม่นั้น, ให้ใช้จุดพัก ที่ข้อความแรกภายใน handler. ตัวแปรที่ถูกประกาศในข้อความผสม หรือคำสั่ง FOR ภายใน handler สามารถนำมาประเมินผลได้.

หลักการที่เกี่ยวข้อง

“SQL ทริกเกอร์” ในหน้า 213

- | คำสั่ง SQL CREATE TRIGGER จะแสดงวิธีการสำหรับระบบจัดการฐานข้อมูล เพื่อให้ควบคุม, มอนิเตอร์, และจัดการ
- | กลุ่มตารางอย่างได้ผลทุกครั้งที่มีการแทรก, อัปเดต, หรือลบออก.

การปรับปรุงประสิทธิภาพของโพรซีเจอร์ และฟังก์ชัน

เมื่อมีการสร้างโพรซีเจอร์ที่เก็บไว้ และฟังก์ชันแบบผู้ใช้กำหนดเอง (UDF), ตัวประมวลผลภาษา SQL เชิงโพรซีเจอร์บน iSeries จะไม่สร้างโค้ดที่มีประสิทธิภาพสูงสุดได้. อย่างไรก็ตาม, คุณสามารถทำการเปลี่ยนแปลงบางอย่างเพื่อลดจำนวนการเรียกใช้เอ็นจินฐานข้อมูลที่จำเป็น และปรับปรุงประสิทธิภาพการทำงาน.

ซึ่งคุณสามารถแก้ไขได้ทั้งในส่วนการออกแบบรูทีน และในส่วนของการทำงานไปปฏิบัติ. ตัวอย่างเช่น, ช่วงเวลาระหว่างการจัดการตัวแปรโฮสต์ในคอมไพเลอร์ภาษา C กับช่วงที่ตัวประมวลผลโพรซีเจอร์ของ SQL ต้องรอการจัดการตัวแปรโฮสต์ สามารถทำให้เกิดการเรียกใช้ส่วนเอ็นจินฐานข้อมูลได้หลายครั้ง. การเรียกใช้ส่วนเอ็นจินฐานข้อมูลเหล่านี้ต้องใช้ทรัพยากรมาก, และเมื่อต้องทำหลายๆ ครั้ง, จะส่งผลทำให้ประสิทธิภาพของเครื่องลดลงอย่างมาก.

การปรับปรุงประสิทธิภาพการทำงานของโพรซีเจอร์และฟังก์ชัน

คำแนะนำเหล่านี้เป็นเทคนิคการเขียนโค้ดพื้นฐานที่สามารถช่วยลดเวลาในการประมวลผลของฟังก์ชันหรือโพรซีเจอร์ได้. คุณควรทำตามคำแนะนำในส่วนของฟังก์ชันอย่างเคร่งครัด, เพราะฟังก์ชันมีแนวโน้มว่าจะถูกเรียกใช้บ่อยครั้งจากหลายโพรซีเจอร์.

- ให้ใช้อ็อปชัน NOT FENCED เพื่อให้ UDF รันใน thread เดียวกันกับผู้ที่เรียกใช้
- ให้ใช้อ็อปชัน DETERMINISTIC กับโพรซีเจอร์และ UDF ที่ให้ค่าผลลัพธ์เหมือนเดิมทุกครั้งเมื่อใส่ค่าอินพุตเดียวกัน. การใช้อ็อปชันนี้จะทำให้ optimizer สามารถแคชค่าผลลัพธ์ของการเรียกฟังก์ชันหรือแคชลำดับการเรียกฟังก์ชันในช่วงกระแสที่โปรแกรมทำงานเอาไว้ได้เพื่อรันเวลารันใหม่.
- ใช้อ็อปชัน NO EXTERNAL ACTION กับ UDF ที่ไม่รับงานนอกขอบเขตของฟังก์ชัน ตัวอย่างของงานนอกขอบเขตได้แก่ ฟังก์ชันที่ต้องสร้างโพรเซสขึ้นมาใหม่เพื่อมารองรับ request การทำ transaction.

เทคนิคการเขียนโค้ดสำหรับส่วนรูทีนของ SQL จะได้ผลทางประสิทธิภาพช่วงรันใหม่อย่างมากเมื่อมีการสร้างเป็นโปรแกรมภาษา C ออกมา. ถ้าคุณหมั่นใช้ภาษา C ในการกำหนดค่าและการเปรียบเทียบในรูทีนของคุณ, คุณจะสามารถลดจำนวนประโยค SQL ที่ต้องไชลงได้. คำแนะนำเหล่านี้จะช่วยให้อูทีนของคุณสร้างโค้ดภาษา C มากขึ้น และลดจำนวนคำสั่ง SQL ลง.

- ควรประกาศตัวแปรโฮสต์เป็นแบบ NOT NULL. มันจะทำให้โค้ดที่ได้ไม่ต้องคอยไปตรวจสอบและเช็คค่าแฟล็กสำหรับ null. ไม่ควรเช็คค่าตัวแปรทั้งหมดเป็น NOT NULL โดยอัตโนมัติ. การที่คุณระบุเป็น NOT NULL, คุณต้องมีค่าดีฟอลต์เตรียมไว้ด้วย. ถ้าตัวแปรนั้นถูกเรียกใช้ในรูทีนเป็นประจำ, การใช้ค่าดีฟอลต์จะช่วยให้. แต่ถ้า, ตัวแปรนั้นไม่ได้ถูกเรียกใช้อย่างสม่ำเสมอ, การตั้งค่าเป็นดีฟอลต์จะทำให้เกิด overhead ที่ไม่จำเป็น. ค่าดีฟอลต์นั้นเหมาะกับค่าตัวเลขที่สุด, ทำให้ไม่จำเป็นต้องเรียกใช้ฐานข้อมูลเพื่อการโพรเซสการกำหนดค่าดีฟอลต์อีก.
- หลีกเลี่ยงการใช้กับข้อมูลแบบอักขระและแบบวันที่. ตัวอย่าง นี้คือการใช้ตัวแปรเป็นค่าแฟล็กโดยกำหนดค่าเป็น 0, 1, 2, หรือ 3. ถ้าตัวแปรนี้ถูกประกาศเป็นตัวแปรแบบอักขระเดี่ยวแทนที่จะเป็นจำนวนเต็ม, มันต้องมีการเรียกใช้เอ็นจินของฐานข้อมูลซึ่งควรหลีกเลี่ยง.
- ใช้จำนวนเต็ม อย่าใช้ทศนิยมที่มีสเกลเป็นศูนย์, โดยเฉพาะอย่างยิ่งเมื่อตัวแปรนั้นทำหน้าที่เป็นตัวนับ.
- อย่าใช้ตัวแปรชั่วคราว. พิจารณาตัวอย่างต่อไปนี้:

```
IF M_days<=30 THEN
  SET I = M_days-7;
  SET J = 23
  RETURN decimal(M_week_1 + ((M_month_1 - M_week_1)*I)/J,16,7);
END IF
```

ตัวอย่างนี้สามารถเขียนใหม่โดยไม่ต้องใช้ตัวแปรชั่วคราว:


```

IF M_days<=30 THEN
  Return decimal(M-week_1 + ((M_month_1 - M_week_1)* (M_days-7))/23,16,7);
END IF

```

- รวมคำสั่ง SET ที่มีลำดับซับซ้อนเป็นคำสั่งเดียว. ให้ใช้คำสั่งนี้กับคำสั่งที่ไม่สามารถสร้างเป็นภาษา C ได้เพราะมี CCSIDS หรือชนิดข้อมูลนั้นอยู่.

```

SET var1 = function1(var2);
SET var2 = function2();

```

สามารถเขียนรวมเป็นคำสั่งเดียวได้คือ:

```

SET var1 = function1(var2), var2 = function2();

```

- ใช้รูปแบบเป็น IF () ELSE IF () ... ELSE ... แทนที่จะใช้ IF (x AND y) เพื่อเลี่ยงการเปรียบเทียบอันไม่จำเป็น.
- ทำให้ได้มากที่สุดในการคำสั่ง SELECT:

```

SELECT A INTO Y FROM B;
SET Y=Y||'X';

```

เขียนใหม่เป็น:

```

SELECT A || 'X' INTO Y FROM B

```

- หลีกเลี่ยงการเปรียบเทียบอักขระหรือวันที่ภายในรูป. บางครั้งการรวมรูปสามารถเขียนใหม่ให้การเปรียบเทียบไปอยู่นอกรูปโดยให้การเปรียบเทียบนั้นตั้งค่าตัวแปรจำนวนเต็มเพื่อนำมาใช้ในรูปต่อไป. วิธีนี้ทำให้มีการประเมินผลข้อความที่ซับซ้อนเพียงครั้งเดียว. การเปรียบเทียบค่าจำนวนเต็มภายในรูปจะมีประสิทธิภาพมากกว่าเพราะมันสามารถทำได้กับโค้ดภาษา C ที่สร้างออกมา.
- หลีกเลี่ยงการตั้งค่าตัวแปรที่ไม่ได้ใช้. ตัวอย่าง, ถ้ามีการเช็คค่าตัวแปรนอกประโยค IF, ต้องมั่นใจว่าตัวแปรนั้นถูกนำมาใช้จริงกับ instance ทั้งหมดของประโยค IF. มิฉะนั้น, ให้ตั้งค่าตัวแปรเฉพาะในส่วนของประโยค IF ที่ได้ใช้จริงๆ.
- ถ้าเป็นไปได้ แทนที่ส่วนของโค้ดด้วยประโยค SELECT ประโยคเดียว. พิจารณาตัวอย่างโค้ดต่อไปนี้:

```

SET vnb_decimal = 4;
cdecimal:
  FOR vdec AS cdec CURSOR FOR
  SELECT nb_decimal
  FROM K$FX_RULES
  WHERE first_currency=Pi_cur1 AND second_currency=P1_cur2
  DO
    SET vnb_decimal=SMALLINT(cdecimal.nb_decimal);
  END FOR cdecimal;

```

```

IF vnb_decimal IS NULL THEN
  SET vnb_decimal=4;
END IF;
SET vrate=ROUND(vrate1/vrate2,vnb_decimal);
RETURN vrate;

```

โค้ดตัวอย่างข้างต้นสามารถทำให้มีประสิทธิภาพดีขึ้นได้โดยเขียนเป็น:

```

RETURN( SELECT
  CASE
    WHEN MIN(nb_decimal) IS NULL THEN ROUND(Vrate1/Vrate2,4)
    ELSE ROUND(Vrate1/Vrate2,SMALLINT(MIN(nb_decimal)))
  )

```

```

END
FROM K$FX_RULES
WHERE first_currency=Pi_cur1 AND second_currency=Pi_cur2);

```

- โค้ดภาษา C สามารถใช้สำหรับการตั้งค่าและการเปรียบเทียบค่าของข้อมูลแบบอักขระได้ถ้า CCSID ของ operand ทั้งสองฝั่งนั้นเหมือนกัน, ถ้า CCSID ตัวหนึ่งเท่ากับ 65535, ถ้า CCSID ไม่ใช่ UTF8, และถ้าการ truncate ข้อมูลอักขระไม่สามารถทำได้. ถ้าไม่มีการระบุ CCSID ให้กับตัวแปร, จะไม่มีการกำหนดค่า CCSID จนกว่าจะมีการเรียกใช้โปรแกรมเมอร์. ในกรณีเช่นนี้, ต้องสร้างโค้ดให้ทำการกำหนดและเปรียบเทียบค่า CCSID ณ เวลารันใหม่. แต่ถ้ามีการกำหนดลำดับการเรียงทางเลือกไว้หรือกำหนดค่า *JOB RUN, คุณจะไม่สามารถสร้างโค้ดภาษา C เพื่อใช้เปรียบเทียบอักขระได้.
- สำหรับการกำหนดค่าตัวแปรแบบตัวเลขทั้งหมดให้ใช้ชนิดข้อมูลเดียวกัน, มีความยาวเท่ากันและสเกลเดียวกัน. การสร้างโค้ดภาษา C จะทำได้ก็ต่อเมื่อไม่สามารถทำการ truncate ได้.

```

DECLARE v1, v2 INT;
SET v1 = 100;
SET v1 = v2;

```

การออกแบบวิธีใหม่เพื่อเพิ่มประสิทธิภาพในการทำงาน

ถึงแม้จะทำตามคำแนะนำทุกอย่างแล้วก็ตาม, บางทีโปรแกรมเมอร์หรือฟังก์ชันนั้นก็ยังไม่ทำงานไม่ได้ตามต้องการ. ในกรณีเช่นนี้, คุณต้องย้อนกลับไปดูที่การออกแบบโปรแกรมเมอร์หรือ UDF แล้วพิจารณาว่าจะสามารถแก้ไขอะไรเพื่อเพิ่มประสิทธิภาพการทำงานได้บ้าง.

มีวิธีการปรับปรุงการออกแบบสองอย่างที่สมควรพิจารณา.

อย่างแรกคือการลดจำนวนการเรียกใช้ฐานข้อมูลหรือฟังก์ชันของโปรแกรมเมอร์, ซึ่งมีขั้นตอนคล้ายกับการค้นหาส่วนของโค้ดที่สามารถแปลงเป็นประโยค SQL. บ่อยครั้งที่คุณสามารถลดจำนวนการเรียกใช้ได้โดยการเพิ่มตรรกะเข้าไปในโค้ดของคุณ.

ส่วนอีกวิธีที่ยากกว่าก็คือการปรับโครงสร้างของฟังก์ชันทั้งหมดโดยยึดตามผลลัพธ์เดิมแต่เปลี่ยนวิธีการใหม่. ตัวอย่างเช่น, ฟังก์ชันของคุณใช้ประโยค SELECT เพื่อใช้หาหนทางที่ตรงตามเงื่อนไขที่กำหนด จากนั้นก็เรียกใช้งานประโยคค้นแบบไดนามิกส์. โดยพิจารณาจากวิธีที่ฟังก์ชันทำงาน, คุณอาจเปลี่ยนตรรกะการทำงานของฟังก์ชัน เพื่อให้ฟังก์ชันสามารถใช้เคียวรี SELECT แบบสแตติกในการค้นหาคำตอบ, ซึ่งส่งผลให้ได้ประสิทธิภาพที่ดีขึ้น.

คุณควรใช้ประโยค compound ในลักษณะที่ซ้อนกันเพื่อให้การ handle ของ exception และเคอร์เซอร์เป็นแบบ local. ถ้ามีการกำหนด handler เฉพาะเป็นจุดๆ, โค้ดที่ได้จะตรวจดูว่ามีข้อผิดพลาดหลังประโยคนั้นๆหรือไม่. โค้ดที่เกิดขึ้นจะสามารถปิดเคอร์เซอร์และเริ่มขั้นตอนของจุดช่วยเหลือถ้าเกิดข้อผิดพลาดในประโยค compound นั้น. สำหรับวิธีที่มีประโยค compound เดียวแต่มีหลาย handler และหลายเคอร์เซอร์, โค้ดที่ได้จะจัดการทุก handler และเคอร์เซอร์หลังจากทุกประโยค SQL. ถ้าคุณกำหนดขอบเขตของ handler และ เคอร์เซอร์กับในประโยค compound แบบซ้อน, handler และ เคอร์เซอร์จะถูกตรวจสอบภายในประโยค compound แบบซ้อนเท่านั้น.

ในรูทีนตัวอย่างนี้, โค้ดการตรวจสอบข้อผิดพลาด SQLSTATE '22H11' จะถูกสร้างขึ้นสำหรับประโยคที่อยู่ในประโยค compound ที่ชื่อ lab2 เท่านั้น. จะไม่มีการตรวจสอบข้อผิดพลาดเฉพาะนี้กับประโยคที่อยู่ในรูทีนนอกบล็อก lab2. จะมีการสร้างโค้ดการตรวจสอบข้อผิดพลาด SQLEXCEPTION สำหรับทุกประโยคที่อยู่ในบล็อก lab1 และ lab2. ในทำนองเดียวกัน, การจัดการข้อผิดพลาดสำหรับการปิดเคอร์เซอร์ c1 จะถูกจำกัดเฉพาะประโยคในบล็อก lab2 เท่านั้น.

```

Lab1: BEGIN
  DECLARE var1 INT;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
    RETURN -3;
  lab2: BEGIN

```

```
DECLARE EXIT HANDLER FOR SQLSTATE '22H11'  
    RETURN -1;  
DECLARE c1 CURSOR FOR SELECT col1 FROM table1;  
OPEN c1;  
CLOSE c1;  
END lab2;  
END Lab1
```

เนื่องจากการออกแบบรูทีนใหม่ทั้งหมดเป็นการใช้เวลาอย่างมาก, ให้ตรวจสอบเฉพาะรูทีนหลักๆที่จะทำให้เกิดปัญหาทางด้านประสิทธิภาพแทนที่จะดูแลพีลเคชันทั้งหมด. แต่สิ่งที่สำคัญกว่าการแก้ปัญหาด้วยการออกแบบใหม่ก็คือ การพยายามทบทวนถึงผลด้านประสิทธิภาพเป็นหลักตั้งแต่ช่วงที่ทำการออกแบบ. ถ้าคุณพยายามเน้นในส่วนของแอ็พพลิเคชั่นที่คาดว่าจะถูกใช้งานอย่างหนัก และมั่นใจว่าได้ออกแบบส่วนนั้นโดยคำนึงถึงประสิทธิภาพ คุณก็จะปลอดภัยจากการที่ต้องมาออกแบบส่วนดังกล่าวใหม่ภายหลัง.

การประมวลผลชนิดข้อมูลพิเศษ

ชนิดข้อมูลส่วนใหญ่, เช่น INTEGER และ CHARACTER ไม่ต้องการประมวลผลที่มีคุณลักษณะพิเศษ. อย่างไรก็ตาม, อาจมีชนิดข้อมูลบางประเภทที่ต้องใช้ฟังก์ชันพิเศษ หรือ locator ในการเรียกใช้ข้อมูลเหล่านั้น.

การใช้ large objects (LOBs)

ชนิดข้อมูล VARCHAR, VARCHARIC, และ VARBINARY จำกัดเนื้อที่การจัดเก็บได้เพียง 32 KB (โดยที่ KB มีค่าเท่ากับ 1024 ไบต์) ของหน่วยเก็บ. โดยที่ข้อจำกัดนี้อาจเพียงพอสำหรับข้อความที่มีขนาดเล็กถึงปานกลาง, แต่แอ็พพลิเคชั่นต้องการเก็บเอกสารที่มีข้อความขนาดใหญ่. ซึ่งแอ็พพลิเคชั่นเหล่านี้ต้องการเก็บชนิดข้อมูลอื่นๆ อีกหลายชนิดเพิ่มเติม เช่น เสียง, วิดีโอ, รูปภาพ, ข้อความผสมกับกราฟิก, และรูปภาพ. มีชนิดข้อมูลอยู่สามชนิดที่ไว้จัดเก็บอ็อบเจ็กต์ข้อมูลเหล่านี้ในลักษณะของสตริงที่มีขนาดสูงสุด 2 GB (โดยที่ GB มีค่าเท่ากับ 1 073 741 824 ไบต์).

ชนิดข้อมูลทั้งสามชนิดคือ: binary large objects (BLOBs), single-byte character large objects (CLOBs), และ double-byte character Large objects (DBCLOBs). แต่ละตารางอาจมีข้อมูล LOB เป็นจำนวนมาก. ถึงแม้ว่า แถวหนึ่งจะเก็บค่า LOB ได้ไม่เกิน 3.5 GB, แต่ตารางอาจมีข้อมูล LOB เกือบถึง 256 GB.

คุณสามารถอ้างอิงและดำเนินการกับ LOBs โดยใช้ตัวแปรโฮสต์ได้เหมือนกับที่คุณทำกับชนิดข้อมูลอื่นๆ. อย่างไรก็ตาม, ตัวแปรโฮสต์ใช้หน่วยความจำจากโปรแกรมซึ่งอาจจะมีความไม่ใหญ่พอที่จะเก็บค่า LOB. วิธีอื่นจึงจำเป็นสำหรับดำเนินการกับค่าขนาดใหญ่. *Locators* ใช้เพื่อระบุและดำเนินการกับอ็อบเจ็กต์ขนาดใหญ่ในเซิร์ฟเวอร์ฐานข้อมูลและใช้สำหรับดึงค่าของ LOB. *ตัวแปรที่อ้างอิงถึงไฟล์* ใช้เพื่อย้ายค่าอ็อบเจ็กต์ขนาดใหญ่ (หรือส่วนที่ใหญ่ของอ็อบเจ็กต์นั้น) ไปยังโคลเอนต์หรือย้ายค่านั้นมาจากโคลเอนต์.

การทำความเข้าใจกับชนิดข้อมูลอ็อบเจ็กต์ขนาดใหญ่ (BLOB, CLOB, DBCLOB)

ชนิดข้อมูลอ็อบเจ็กต์ขนาดใหญ่จะถูกกำหนดไว้ที่นี่.

- Binary large object (BLOB) — คือสตริงแบบไบนารีที่ถูกสร้างมาจากข้อมูลไบต์ที่ไม่มีการเชื่อมโยงกับโค้ดเพจ. ชนิดข้อมูลนี้สามารถเก็บข้อมูลไบนารีที่มีขนาดใหญ่กว่าชนิด VARBINARY (ถูกจำกัดไว้ที่ 32K). ชนิดข้อมูลนี้เหมาะสำหรับเก็บรูปภาพ, เสียง, กราฟิก, และข้อมูลเฉพาะทางธุรกิจหรือแอ็พพลิเคชั่นอื่นๆ.

- Character large object (CLOB) — คือสตริงอักขระที่ถูกสร้างมาจากอักขระแบบไบต์เดียวที่มีการเชื่อมโยงกับโค้ดเพจ. ชนิดข้อมูลนี้เหมาะสมสำหรับการเก็บข้อมูลลักษณะที่เป็นตัวอักษรโดยขนาดของข้อมูลอาจเพิ่มจนเกินขีดจำกัดของชนิดข้อมูล VARCHAR (จำนวนสูงสุดไม่เกิน 32 กิโลไบต์). ข้อมูลชนิดนี้รองรับการแปลงโค้ดเพจได้
- Double-byte character large object (DBCLOBs) — คือสตริงอักขระที่ถูกสร้างมาจากอักขระแบบสองไบต์ที่มีการเชื่อมโยงกับโค้ดเพจ. ชนิดข้อมูลนี้เหมาะสำหรับเก็บข้อมูลลักษณะที่เป็นตัวอักษรซึ่งใช้ชุดอักขระแบบสองไบต์. อีกครั้ง, ข้อมูลชนิดนี้รองรับการแปลงโค้ดเพจได้.

การทำความเข้าใจกับ large object locator

Large object (LOB) locator ใช้ค่าขนาดเล็ก, จัดการง่าย เพื่ออ้างอิงค่าที่ใหญ่กว่ามาก.

ถ้าชี้ชัดลงไป, LOB locator ก็คือค่าขนาด 4 ไบต์ที่เก็บอยู่ในตัวแปรโฮสต์ที่โปรแกรมใช้อ้างอิงไปสู่ค่า LOB ที่อยู่ในระบบฐานข้อมูล. โดยการใช้ LOB locator แล้ว, โปรแกรมสามารถจัดการกับค่า LOB เหมือนกับว่าค่า LOB เก็บอยู่ในตัวแปรโฮสต์ปกติ. เมื่อคุณใช้ LOB locator, จึงไม่จำเป็นต้องส่งค่า LOB จากเซิร์ฟเวอร์ไปยังแอปพลิเคชัน (และอาจจะส่งค่ากลับมาอีกครั้ง).

LOB locator จะเชื่อมโยงกับค่า LOB, ไม่ได้เชื่อมโยงกับแถวหรือตำแหน่งที่เก็บข้อมูลในฐานข้อมูล. ดังนั้น, หลังจากที่กำหนดค่า LOB ให้กับ locator แล้ว, คุณไม่สามารถทำอะไรกับแถวหรือตารางต้นฉบับที่จะมีผลกับค่าที่ถูกอ้างอิงจาก locator ได้. ค่าที่สัมพันธ์กับ locator จะยังถูกต้องจนกระทั่งหน่วยการทำงานสิ้นสุด, หรือเมื่อ locator ถูกปล่อยค่าโดยตรง, อยู่ที่ว่าเหตุการณ์ใดเกิดขึ้นก่อน. คำสั่ง FREE LOCATOR จะปลด locator จากค่าที่มันเชื่อมโยงอยู่. ในทำนองเดียวกัน, คำสั่ง commit หรือ rollback จะปลด LOB locators ที่ผูกกับ transaction ทั้งหมดออก.

LOB locators สามารถถูกผ่านค่าไปมากับ UDFs ได้. ใน UDF, ฟังก์ชันที่ใช้ข้อมูล LOB สามารถนำไปจัดการค่า LOB โดยใช้ LOB locators ได้.

เมื่อเลือกค่า LOB, คุณมี 3 ตัวเลือกคือ.

- เลือกค่า LOB ทั้งหมดไปที่ตัวแปรโฮสต์. ค่า LOB ทั้งหมดจะถูกทำสำเนาไปยังตัวแปรโฮสต์.
- เลือกค่า LOB ไปที่ LOB locator. ค่า LOB จะยังอยู่ที่เซิร์ฟเวอร์; ค่า LOB จะไม่ถูกทำสำเนาไปที่ตัวแปรโฮสต์.
- เลือกค่า LOB ทั้งหมดไปที่ตัวแปรที่อ้างอิงไปยังไฟล์. ค่า LOB จะถูกย้ายไปยังไฟล์ Integrated File System (IFS).

ลักษณะการใช้ค่า LOB ภายในโปรแกรมสามารถช่วยโปรแกรมเมอร์ตัดสินใจว่าวิธีการใดเหมาะสมที่สุด. ถ้าค่า LOB มีขนาดใหญ่มากและจำเป็นต้องใช้เป็นค่าอินพุตสำหรับคำสั่ง SQL ที่ตามมาเท่านั้น, ให้เก็บค่าไว้ใน locator.

ถ้าโปรแกรมจำเป็นต้องใช้ค่า LOB ทั้งหมดโดยไม่สนใจเรื่องขนาด, คงไม่มีทางเลือกอื่นนอกจากจะถ่ายโอน LOB เท่านั้น. แม้ในกรณีนี้, ก็ยังมีตัวเลือกสำหรับคุณ. คุณสามารถเลือกค่าทั้งหมดไปที่ตัวแปรโฮสต์ปกติหรือไปที่ตัวแปรโฮสต์ที่อ้างอิงไปยังไฟล์. คุณยังสามารถเลือกค่า LOB ให้กับ locator และอ่านค่ามันทีละส่วนจาก locator ไปยังตัวแปรโฮสต์ปกติได้.

สิ่งอ้างอิงที่เกี่ยวข้อง

“ตัวแปรที่อ้างอิงถึงไฟล์ LOB” ในหน้า 236

ตัวแปรที่อ้างอิงถึงไฟล์จะคล้ายกับตัวแปรโฮสต์ ยกเว้นว่า ตัวแปรนั้นถูกใช้เพื่อโอนย้ายข้อมูลไปมาระหว่างไฟล์ IFS (ไม่ใช่โอนย้ายไปมาระหว่างบัฟเฟอร์หน่วยความจำ).

“ตัวอย่าง: การใช้ locator เพื่อทำงานกับค่า CLOB” ในหน้า 231

ในตัวอย่างนี้, แอปพลิเคชันโปรแกรมจะดึงค่า locator สำหรับค่า LOB; ซึ่งจะใช้ locator เพื่อดึงข้อมูลจากค่า LOB ต่อไป.

ตัวอย่าง: การใช้ locator เพื่อทำงานกับค่า CLOB

ในตัวอย่างนี้, แอปพลิเคชันโปรแกรมจะดึงค่า locator สำหรับค่า LOB; ซึ่งจะใช้ locator เพื่อดึงข้อมูลจากค่า LOB ต่อไป.

โดยใช้วิธีนี้, โปรแกรมจัดสรรที่เก็บข้อมูลให้พอสำหรับข้อมูล LOB เพียงหนึ่งชิ้นเท่านั้น (ขนาดที่เก็บจะพิจารณาโดยโปรแกรม). นอกเหนือจากนี้, โปรแกรมสามารถออกคำสั่งดึงข้อมูลโดยใช้เคอร์เซอร์เพียงครั้งเดียวเท่านั้น.

โปรแกรมตัวอย่าง LOBLOC ทำงานอย่างไร

1. การประกาศตัวแปรโฮสต์. คำสั่ง BEGIN DECLARE SECTION และ END DECLARE SECTION เป็นส่วนที่ใช้สำหรับประกาศตัวแปรโฮสต์. ตัวแปรโฮสต์จะนำหน้าด้วยโคลอน (:) เมื่อถูกอ้างอิงในคำสั่ง SQL. ตัวแปรโฮสต์ CLOB LOCATOR จะถูกประกาศ.
2. การดึงค่า LOB ไปที่ตัวแปรโฮสต์ของ locator. รูทีน CURSOR และ FETCH ถูกใช้เพื่อรับค่าตำแหน่งของฟิลด์ LOB ในฐานข้อมูลไปไว้ที่ตัวแปรโฮสต์ locator.
3. ปล่อยค่า LOB LOCATORS. LOB LOCATORS ที่ถูกใช้ในตัวอย่างนี้จะถูกปล่อยค่า, เป็นการปล่อยค่า locator จากค่าที่ก่อนหน้านี้เชื่อมโยงอยู่.

แมโคร/ฟังก์ชัน CHECKERR คือยูทิลิตี้ที่ใช้ตรวจหาข้อผิดพลาดจากภายนอกโปรแกรม. ตำแหน่งของยูทิลิตี้ตรวจหาข้อผิดพลาดนี้จะขึ้นอยู่กับภาษาโปรแกรมที่ใช้. ในตัวอย่างนี้, เป็นภาษา C โดยค่า check_error ถูกกำหนดเป็น CHECKERR และนำไปไว้ที่ไฟล์ util.c.

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

หลักการที่เกี่ยวข้อง

“การทำความเข้าใจกับ large object locator” ในหน้า 230

Large object (LOB) locator ใช้ค่าขนาดเล็ก, จัดการง่าย เพื่ออ้างอิงค่าที่ใหญ่กว่ามาก.

ตัวอย่าง: LOBLOC.SQC ใน C:

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

#ifdef DB2MAC
    char * bufptr;
#endif

    EXEC SQL BEGIN DECLARE SECTION; 1
        char number[7];
```

```

        long deptInfoBeginLoc;
        long deptInfoEndLoc;
        SQL TYPE IS CLOB_LOCATOR resume;
        SQL TYPE IS CLOB_LOCATOR deptBuffer;
        short lobind;
char buffer[1000]="";
char userid[9];
char passwd[19];
EXEC SQL END DECLARE SECTION;

printf( "Sample C program: LOBLOC\n" );

if (argc == 1) {
    EXEC SQL CONNECT TO sample;
CHECKERR ("CONNECT TO SAMPLE");
}
else if (argc == 3) {
    strcpy (userid, argv[1]);
    strcpy (passwd, argv[2]);
    EXEC SQL CONNECT TO sample USER :userid USING :passwd;
    CHECKERR ("CONNECT TO SAMPLE");
}
    else {
printf ("\nUSAGE: lobloc [userid passwd]\n\n");
    return 1;
} /* endif */

/* พนักงาน A10030 ไม่ถูกรวมอยู่ในรายการที่เลือกต่อไปนี้, เพราะว่า
โปรแกรม lobeval จะดำเนินการกับเรกคอร์ดของ A10030 เพื่อที่ว่า
เรกคอร์ดนั้นจะทำงานร่วมกับ lobloc ได้ */

EXEC SQL DECLARE c1 CURSOR FOR
        SELECT empno, resume FROM emp_resume WHERE resume_format='ascii'
        AND empno <> 'A00130';

EXEC SQL OPEN c1;
CHECKERR ("OPEN CURSOR");

do {
EXEC SQL FETCH c1 INTO :number, :resume :lobind; 2
    if (SQLCODE != 0) break;
if (lobind < 0) {
    printf ("NULL LOB indicated\n");
} else {
    /* EVALUATE the LOB LOCATOR */
    /* Locate the beginning of "Department Information" section */
EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
        INTO :deptInfoBeginLoc;
    CHECKERR ("VALUES1");

    /* Locate the beginning of "Education" section (end of "Dept.Info" */
EXEC SQL VALUES (POSSTR(:resume, 'Education'))
        INTO :deptInfoEndLoc;
    CHECKERR ("VALUES2");
}
}

```

```

        /* Obtain ONLY the "Department Information" section by using SUBSTR */
        EXEC SQL VALUES(SUBSTR(:resume, :deptInfoBeginLoc,
            :deptInfoEndLoc - :deptInfoBeginLoc)) INTO :deptBuffer;
        CHECKERR ("VALUES3");

        /* Append the "Department Information" section to the :buffer var. */
        EXEC SQL VALUES(:buffer || :deptBuffer) INTO :buffer;
        CHECKERR ("VALUES4");
    } /* endif */
} while ( 1 );

#ifdef DB2MAC
    /* Need to convert the newline character for the Mac */
    bufptr = &(buffer[0]);
    while ( *bufptr != '\0' ) {
        if ( *bufptr == 0x0A ) *bufptr = 0x0D;
        bufptr++;
    }
#endif

    printf ("%s\n",buffer);

    EXEC SQL FREE LOCATOR :resume, :deptBuffer; 3
    CHECKERR ("FREE LOCATOR");

    EXEC SQL CLOSE c1;
    CHECKERR ("CLOSE CURSOR");

    EXEC SQL CONNECT RESET;
    CHECKERR ("CONNECT RESET");
    return 0;
}
/* end of program : LOBLOC.SQC */

```

ตัวอย่าง: LOBLOC.SQB ใน COBOL:

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

Identification Division.
 Program-ID. "lobloc".

Data Division.

Working-Storage Section.
 copy "sqlenv.cbl".
 copy "sql.cbl".
 copy "sqlca.cbl".

```

    EXEC SQL BEGIN DECLARE SECTION END-EXEC. 1
01 userid          pic x(8).
01 passwd.
   49 passwd-length pic s9(4) comp-5 value 0.
   49 passwd-name   pic x(18).
01 empnum         pic x(6).
01 di-begin-loc   pic s9(9) comp-5.

```

```

01 di-end-loc      pic s9(9) comp-5.
01 resume         USAGE IS SQL TYPE IS CLOB-LOCATOR.
01 di-buffer      USAGE IS SQL TYPE IS CLOB-LOCATOR.
01 lobind         pic s9(4) comp-5.
01 buffer         USAGE IS SQL TYPE IS CLOB(1K).
      EXEC SQL END DECLARE SECTION END-EXEC.

```

```
77 errloc         pic x(80).
```

Procedure Division.

Main Section.

```
      display "Sample COBOL program: LOBLOC".
```

* Get database connection information.

```
      display "Enter your user id (default none): "
          with no advancing.
      accept userid.
```

```
      if userid = spaces
          EXEC SQL CONNECT TO sample END-EXEC

```

```
      else
          display "Enter your password : " with no advancing
          accept passwd-name.
```

* Passwords in a CONNECT statement must be entered in a VARCHAR

* format with the length of the input string.

```
      inspect passwd-name tallying passwd-length for characters
          before initial " ".
```

```
      EXEC SQL CONNECT TO sample USER :userid USING :passwd
      END-EXEC.
```

```
      move "CONNECT TO" to errloc.
      call "checkerr" using SQLCA errloc.
```

* Employee A10030 is not included in the following select, because

* the lobeval program manipulates the record for A10030 so that it is

* not compatible with lobloc

```
      EXEC SQL DECLARE c1 CURSOR FOR
          SELECT empno, resume FROM emp_resume
          WHERE resume_format = 'ascii'
      AND empno <> 'A00130' END-EXEC.
```

```
      EXEC SQL OPEN c1 END-EXEC.
      move "OPEN CURSOR" to errloc.
      call "checkerr" using SQLCA errloc.
```

```
      Move 0 to buffer-length.
```

```
      perform Fetch-Loop thru End-Fetch-Loop
          until SQLCODE not equal 0.
```

* display contents of the buffer.

```
      display buffer-data(1:buffer-length).
```



```

EXEC SQL FREE LOCATOR :resume, :di-buffer END-EXEC. 3
  move "FREE LOCATOR" to errloc.
  call "checkerr" using SQLCA errloc.

EXEC SQL CLOSE c1 END-EXEC.
  move "CLOSE CURSOR" to errloc.
  call "checkerr" using SQLCA errloc.

EXEC SQL CONNECT RESET END-EXEC.
  move "CONNECT RESET" to errloc.
  call "checkerr" using SQLCA errloc.
End-Main.
  go to End-Prog.

Fetch-Loop Section.
EXEC SQL FETCH c1 INTO :empnum, :resume :lobind 2
END-EXEC.

  if SQLCODE not equal 0
    go to End-Fetch-Loop.

* check to see if the host variable indicator returns NULL.
  if lobind less than 0 go to NULL-lob-indicated.

* Value exists. Evaluate the LOB locator.
* Locate the beginning of "Department Information" section.
  EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
    INTO :di-begin-loc END-EXEC.
  move "VALUES1" to errloc.
  call "checkerr" using SQLCA errloc.

* Locate the beginning of "Education" section (end of Dept.Info)
  EXEC SQL VALUES (POSSTR(:resume, 'Education'))
    INTO :di-end-loc END-EXEC.
  move "VALUES2" to errloc.
  call "checkerr" using SQLCA errloc.

  subtract di-begin-loc from di-end-loc.

* Obtain ONLY the "Department Information" section by using SUBSTR
  EXEC SQL VALUES (SUBSTR(:resume, :di-begin-loc,
    :di-end-loc))
    INTO :di-buffer END-EXEC.
  move "VALUES3" to errloc.
  call "checkerr" using SQLCA errloc.

* Append the "Department Information" section to the :buffer var
  EXEC SQL VALUES (:buffer || :di-buffer) INTO :buffer
  END-EXEC.
  move "VALUES4" to errloc.
  call "checkerr" using SQLCA errloc.

  go to End-Fetch-Loop.

NULL-lob-indicated.

```

```
display "NULL LOB indicated".
```

```
End-Fetch-Loop. exit.
```

```
End-Prog.
```

```
stop run.
```

ตัวแปรตัวบ่งชี้ และ LOB locator

สำหรับตัวแปรโฮสต์ในแอฟพลิเคชันปรกติแล้ว, เมื่อเลือกค่า NULL ให้กับตัวแปรโฮสต์แล้ว, ค่าลบจะถูกกำหนดให้กับตัวแปรตัวบ่งชี้ เพื่อเป็นความหมายว่าค่าคือ NULL. อย่างไรก็ตาม, ในกรณีของ LOB locator, ความหมายของตัวแปรตัวบ่งชี้จะต่างไปเล็กน้อย.

เนื่องจากตัวแปรโฮสต์ไม่สามารถมีค่าเป็น NULL ได้, ดังนั้นตัวแปรตัวบ่งชี้ที่มีค่าเป็นลบจะทำให้รู้ว่าค่า LOB ที่อ้างถึงโดย LOB locator มีค่าเป็น NULL. ข้อมูล NULL จะถูกเก็บไว้ที่โคลเอ็นต์โดยใช้ค่าตัวแปรตัวบ่งชี้. เซิร์ฟเวอร์จะไม่แทรกค่า NULL ด้วย locator ที่ถูกต้อง.

ตัวแปรที่อ้างอิงถึงไฟล์ LOB

ตัวแปรที่อ้างอิงถึงไฟล์จะคล้ายกับตัวแปรโฮสต์ ยกเว้นว่า ตัวแปรนั้นถูกใช้เพื่อโอนย้ายข้อมูลไปมาระหว่างไฟล์ IFS (ไม่ใช่โอนย้ายไปมาระหว่างบัฟเฟอร์หน่วยความจำ).

ตัวแปรที่อ้างอิงถึงไฟล์จะแทนค่าไฟล์(มากกว่าที่จะเก็บไฟล์), คล้ายกับที่ LOB locator แทนค่า LOB(มากกว่าที่จะเก็บค่า LOB). เคียวรี, อัปเดต, และแทรกฐานข้อมูลอาจใช้ตัวแปรที่อ้างอิงถึงไฟล์ เพื่อเก็บ, หรือดึง, ค่า LOB หนึ่งค่า.

สำหรับอ็อบเจกต์ที่มีขนาดใหญ่มาก, ไฟล์คือที่เก็บตามปรกติ. มันเหมือนกับว่า LOBs ส่วนใหญ่เริ่มมาจากข้อมูลนั้นถูกเก็บไว้ในไฟล์บนโคลเอ็นต์ก่อนที่จะย้ายข้อมูลนั้นไปที่ฐานข้อมูลบนเซิร์ฟเวอร์. การใช้ตัวแปรที่อ้างอิงถึงไฟล์จะช่วยให้การย้ายข้อมูล LOB. โปรแกรมผู้ใช้ตัวแปรที่อ้างอิงถึงไฟล์เพื่อโอนถ่ายข้อมูล LOB จากไฟล์ IFS ไปยังเอ็นจินฐานข้อมูลโดยตรง. เมื่อต้องการย้ายข้อมูล LOB, แอฟพลิเคชันไม่จำเป็นต้องเขียนยูทิลิตี้รูทีน เพื่ออ่านและเขียนไฟล์โดยใช้ตัวแปรโฮสต์.

หมายเหตุ: ไฟล์ที่ถูกอ้างอิงถึงจะต้องสามารถเข้าถึงได้จาก(แต่ไม่จำเป็นต้องเก็บอยู่ใน)ระบบที่โปรแกรมนั้นทำงานอยู่. สำหรับโปรซีเดอร์ที่เก็บไว้, จะอยู่ที่เซิร์ฟเวอร์.

ตัวแปรที่อ้างอิงถึงไฟล์มีชนิดข้อมูลเป็น BLOB, CLOB, หรือ DBCLOB. และตัวแปรนี้ถูกใช้ให้เป็นแหล่งข้อมูล(อินพุต) หรือไม่กี่เป็นข้อมูลปลายทาง(เอาต์พุต). ตัวแปรที่อ้างอิงถึงไฟล์อาจจะเป็นชื่อไฟล์แบบอ้างอิง(relative file name) หรือชื่อไฟล์แบบสมบูรณ์(complete path name) ก็ได้ (แนะนำให้ใช้อย่างหลัง). ความยาวของชื่อไฟล์ถูกระบุโดยแอฟพลิเคชันโปรแกรม. ส่วนความยาวข้อมูลของตัวแปรที่อ้างอิงถึงไฟล์จะไม่ถูกใช้ขณะอินพุต. ขณะเอาต์พุต, ความยาวข้อมูลจะถูกตั้งค่าโดยโค้ดของ application requester ให้มีความยาวของข้อมูลใหม่เพื่อค่านี้อาจได้ถูกเขียนลงไปในไฟล์.

เมื่อใช้ตัวแปรที่อ้างอิงถึงไฟล์จะมีหลายตัวเลือกที่ต่างกันสำหรับทั้งอินพุตและเอาต์พุต. คุณต้องเลือกการกระทำสำหรับไฟล์ โดยการตั้งค่าฟิลด์ file_options ใน structure ของตัวแปรที่อ้างอิงถึงไฟล์. ตัวเลือกสำหรับการกำหนดค่าให้กับฟิลด์ ซึ่งครอบคลุมทั้งค่าอินพุตและค่าเอาต์พุตถูกแสดงไว้ด้านล่างนี้.

ค่า (แสดงสำหรับภาษา C) และอ็อบชัน เมื่อใช้ตัวแปรที่อ้างอิงถึงไฟล์มีค่าดังต่อไปนี้:

- **SQL_FILE_READ** (ไฟล์ปรกติ) — อ็อบชันนี้มีค่าเป็น 2. นี่คือไฟล์ที่สามารถเปิด, อ่าน, และปิดได้. DB2 กำหนดความยาวของข้อมูลในไฟล์(เป็นไบต์)ตอนเปิดไฟล์. แล้ว DB2 จึงผ่านค่าความยาวของข้อมูลไว้ที่ฟิลด์ data_length ของ structure ตัวแปรที่อ้างอิงถึงไฟล์. คำสำหรับ COBOL คือ SQL-FILE-READ.

ค่าและอ็อปชัน เมื่อใช้ตัวแปรที่อ้างอิงถึงไฟล์แบบเอาต์พุตมีค่าดังนี้:

- **SQL_FILE_CREATE** (สร้างไฟล์) – อ็อปชันนี้มีค่าเป็น 8. อ็อปชันนี้จะทำการสร้างไฟล์ใหม่. ถ้ามีไฟล์นี้อยู่แล้ว, จะส่งข้อความแสดงความผิดพลาดกลับมา. คำสำหรับ COBOL คือ SQL-FILE-CREATE.
- **SQL_FILE_OVERWRITE** (เขียนทับไฟล์) – อ็อปชันนี้มีค่าเป็น 16. ตัวเลือกนี้จะทำการสร้างไฟล์ขึ้นมาใหม่ถ้าไฟล์นั้นไม่เคยมีอยู่. แต่ถ้าไฟล์นั้นมีอยู่แล้ว, ข้อมูลใหม่จะเขียนทับข้อมูลเดิมในไฟล์นั้น. คำสำหรับ COBOL คือ SQL-FILE-OVERWRITE.
- **SQL_FILE_APPEND** (ต่อท้ายไฟล์) – อ็อปชันนี้มีค่าเป็น 32. ตัวเลือกนี้จะส่งผลลัพธ์ผนวกเข้าไปต่อท้ายไฟล์, ถ้าไฟล์นั้นมีอยู่. มิฉะนั้น, จะทำการสร้างไฟล์ขึ้นมาใหม่. คำสำหรับ COBOL คือ SQL-FILE-APPEND.

หมายเหตุ: ถ้าตัวแปรที่อ้างอิงถึงไฟล์ LOB ถูกใช้ในคำสั่ง OPEN, ห้ามลบไฟล์ที่เชื่อมโยงกับตัวแปรที่อ้างอิงถึงไฟล์ LOB จนกว่าเคอร์เซอร์จะถูกปิด.

หลักการที่เกี่ยวข้อง

“การทำความเข้าใจกับ large object locator” ในหน้า 230

Large object (LOB) locator ใช้ค่าขนาดเล็ก, จัดการง่าย เพื่ออ้างอิงค่าที่ใหญ่กว่ามาก.

ข้อมูลที่เกี่ยวข้อง

ระบบไฟล์รวม

ตัวอย่าง: การดึงเอกสารไปยังไฟล์

ตัวอย่างโปรแกรมนี้แสดงให้เห็นว่า ส่วนประกอบอักขระอ็อบเจ็กต์ขนาดใหญ่ (CLOB) จะถูกดึงค่าจากตารางไปเก็บไว้ในไฟล์ภายนอกได้อย่างไร.

โปรแกรมตัวอย่าง LOBFILE ทำงานอย่างไร

1. การประกาศตัวแปรโฮสต์. คำสั่ง BEGIN DECLARE SECTION และ END DECLARE SECTION เป็นส่วนที่ใช้สำหรับประกาศตัวแปรโฮสต์. ตัวแปรโฮสต์จะนำหน้าด้วยโคลอน (:) เมื่อถูกอ้างอิงในคำสั่ง SQL. ตัวแปรโฮสต์ CLOB FILE REFERENCE จะถูกประกาศ.
2. ตัวแปรโฮสต์ CLOB FILE REFERENCE จะถูกตั้งค่า. แอ็ตทริบิวต์ของ FILE REFERENCE จะถูกตั้งค่า. ชื่อไฟล์ที่ไม่ได้ประกาศพารามิเตอร์, โดยดีฟอลต์, จะถูกเก็บไว้ในไดเรกทอรีปัจจุบันของผู้ใช้. ถ้าชื่อพารามิเตอร์ไม่ได้ขึ้นต้นด้วยอักขระ forward slash (/), ชื่อนั้นจะใช้ไม่ได้.
3. เลือกไปที่ตัวแปรโฮสต์ CLOB FILE REFERENCE. ข้อมูลจากฟิลด์ resume ถูกกำหนดไปที่ชื่อไฟล์ที่ถูกอ้างอิงจากตัวแปรโฮสต์.

แมโคร/ฟังก์ชัน CHECKERR คือยูทิลิตี้ที่ใช้ตรวจหาข้อผิดพลาดในโปรแกรม. ตำแหน่งของยูทิลิตี้ตรวจหาข้อผิดพลาดนี้จะขึ้นอยู่กับภาษาโปรแกรมที่ใช้:

C check_error ถูกกำหนดใหม่เป็น CHECKERR และอยู่ในไฟล์ util.c.

COBOL

CHECKERR คือโปรแกรมภายนอกที่ชื่อ checkerr.cbl

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

ตัวอย่าง: LOBFILE.SQC ใน C:

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sql.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

#define CHECKERR(CE_STR) if (check_error (CE_STR, &sqlca) != 0) return 1;

int main(int argc, char *argv[]) {

    EXEC SQL BEGIN DECLARE SECTION; 1
        SQL TYPE IS CLOB_FILE resume;
        short lobind;
        char userid[9];
        char passwd[19];
    EXEC SQL END DECLARE SECTION;

    printf( "Sample C program: LOBFILE\n" );

    if (argc == 1) {
        EXEC SQL CONNECT TO sample;
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else if (argc == 3) {
        strcpy (userid, argv[1]);
        strcpy (passwd, argv[2]);
        EXEC SQL CONNECT TO sample USER :userid USING :passwd;
        CHECKERR ("CONNECT TO SAMPLE");
    }
    else {
        printf ("\nUSAGE: lobfile [userid passwd]\n\n");
        return 1;
    } /* endif */

    strcpy (resume.name, "RESUME.TXT"); 2
        resume.name_length = strlen("RESUME.TXT");
        resume.file_options = SQL_FILE_OVERWRITE;

    EXEC SQL SELECT resume INTO :resume :lobind FROM emp_resume 3
        WHERE resume_format='ascii' AND empno='000130';

    if (lobind < 0) {
        printf ("NULL LOB indicated \n");
    } else {
        printf ("Resume for EMPNO 000130 is in file : RESUME.TXT\n");
    } /* endif */

    EXEC SQL CONNECT RESET;
```

```

        CHECKERR ("CONNECT RESET");
        return 0;
    }
/* end of program : LOBFILE.SQC */

```

ตัวอย่าง: LOBFILE.SQB ใน COBOL:

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

```

Identification Division.
Program-ID. "lobfile".

Data Division.
Working-Storage Section.
    copy "sqlenv.cbl".
    copy "sql.cbl".
    copy "sqlca.cbl".

EXEC SQL BEGIN DECLARE SECTION END-EXEC. 1
01 userid          pic x(8).
01 passwd.
    49 passwd-length pic s9(4) comp-5 value 0.
    49 passwd-name   pic x(18).
01 resume          USAGE IS SQL TYPE IS CLOB-FILE.
01 lobind          pic s9(4) comp-5.
EXEC SQL END DECLARE SECTION END-EXEC.

77 errloc          pic x(80).

Procedure Division.
Main Section.
    display "Sample COBOL program: LOBFILE".

* Get database connection information.
    display "Enter your user id (default none): "
        with no advancing.
    accept userid.

    if userid = spaces
        EXEC SQL CONNECT TO sample END-EXEC
    else
        display "Enter your password : " with no advancing
        accept passwd-name.

* Passwords in a CONNECT statement must be entered in a VARCHAR
* format with the length of the input string.
    inspect passwd-name tallying passwd-length for characters
        before initial " ".

EXEC SQL CONNECT TO sample USER :userid USING :passwd
END-EXEC.
move "CONNECT TO" to errloc.
call "checkerr" using SQLCA errloc.

```

```

move "RESUME.TXT" to resume-NAME.                2
move 10 to resume-NAME-LENGTH.
move SQL-FILE-OVERWRITE to resume-FILE-OPTIONS.

EXEC SQL SELECT resume INTO :resume :lobind 3
      FROM emp_resume
      WHERE resume_format = 'ascii'
      AND empno = '000130' END-EXEC.
if lobind less than 0 go to NULL-LOB-indicated.

display "Resume for EMPNO 000130 is in file : RESUME.TXT".
go to End-Main.

NULL-LOB-indicated.
display "NULL LOB indicated".

End-Main.
EXEC SQL CONNECT RESET END-EXEC.
move "CONNECT RESET" to errloc.
call "checkerr" using SQLCA errloc.
End-Prog.
stop run.

```

ตัวอย่าง: การแทรกข้อมูลลงในคอลัมน์ CLOB

ตัวอย่างนี้แสดงวิธีการแทรกข้อมูลจากไฟล์ปรกติที่ถูกอ้างอิงโดย :hv_text_file ลงในคอลัมน์ CLOB.

ในส่วนคำจำกัดความของเช็กเมนต์โปรแกรมภาษา C ดังต่อไปนี้:

- userid เป็นตัวแทนไอดีเรียกทอรี่ของหนึ่งในผู้ใช้ของคุณ.
- dirname เป็นตัวแทนชื่อไอดีเรียกทอรี่ย่อยของ "userid".
- filnam.1 สามารถแปลงเป็นชื่อของเอกสารที่คุณต้องการแทรกเข้าไปในตาราง.
- clobtab คือชื่อของตารางที่มีชนิดข้อมูล CLOB.

```

strcpy(hv_text_file.name, "/home/userid/dirname/filnam.1");
hv_text_file.name_length = strlen("/home/userid/dirname/filnam.1");
hv_text_file.file_options = SQL_FILE_READ; /* this is a 'regular' file */

```

```

EXEC SQL INSERT INTO CLOBTAB
VALUES(:hv_text_file);

```

แสดงโครงร่างของคอลัมน์ LOB

เมื่อแถวของข้อมูลจากตารางที่เก็บคอลัมน์ LOB ถูกแสดงโดยใช้คำสั่ง CL เช่น Display Physical File Member (DSPPFM), ข้อมูล LOB ที่เก็บไว้ในแถวนั้นจะไม่ถูกแสดง. แทนที่, ฐานข้อมูลจะแสดงค่าพิเศษสำหรับคอลัมน์ LOB.

โครงร่างของค่าพิเศษนี้มีค่าดังต่อไปนี้:

- 13 ถึง 28 ไบต์จะเป็นค่าศูนย์ในเลขฐานสิบหก.
- 16 ไบต์จะขึ้นต้นด้วย *POINTER และตามด้วยช่องว่าง.

จำนวนของไบต์ในส่วนแรกของค่าถูกตั้งค่าด้วยจำนวนที่จำเป็นในการจัดตำแหน่งที่ละ 16 ไบต์กับส่วนที่สองของค่า.

ตัวอย่างเช่น, ถ้าคุณมีตารางที่เก็บ 3 คอลัมน์: ColumnOne Char(10), ColumnTwo CLOB(40K), และ ColumnThree BLOB(10M). ถ้าคุณใช้คำสั่ง DSPPFM กับตารางนี้, แต่ละแถวของข้อมูลจะมีค่าดังนี้.

- สำหรับ ColumnOne: 10 ไบต์จะถูกเติมค่าด้วยข้อมูลอักขระ.
- สำหรับ ColumnTwo: 22 ไบต์จะถูกเติมค่าด้วยศูนย์ในเลขฐานสิบหก และ 16 ไบต์จะถูกเติมค่าด้วย '*POINTER'.
- สำหรับ ColumnThree: 16 ไบต์จะถูกเติมค่าด้วยศูนย์ในเลขฐานสิบหก และ 16 ไบต์จะถูกเติมค่าด้วย '*POINTER'.

ชุดของคำสั่งทั้งหมดที่แสดงคอลัมน์ LOB ด้วยวิธีนี้คือ:

- Display Physical File Member (DSPPFM)
- Copy File (CPYF) เมื่อค่า *PRINT ถูกระบุสำหรับคีย์เวิร์ด TOFILE
- Display Journal (DSPJRN)
- Retrieve Journal Entry (RTVJRNE)
- Receive Journal Entry (RCVJRNE) เมื่อค่า *TYPE1, *TYPE2, *TYPE3 และ *TYPE4 ถูกระบุสำหรับคีย์เวิร์ด ENTFMT.

การแสดงผลโครงสร้าง Journal entry ของคอลัมน์ LOB

คำสั่งเหล่านี้จะส่งคืนบัพเฟอร์ที่ให้ผู้ใช้งานสามารถเข้าถึงข้อมูล LOB ที่ได้ผ่านการเจอร์นัลแล้วได้.

- Receive Journal Entry (RCVJRNE) CL command, เมื่อค่า *TYPEPTR ถูกระบุสำหรับคีย์เวิร์ด ENTFMT
- Retrieve Journal Entries (QjoRetrieveJournalEntries) API

โครงสร้างของคอลัมน์ LOB ใน entry นี้จะเป็นดังต่อไปนี้:

- 0 ถึง 15 ไบต์เป็นค่าศูนย์ในฐานสิบหก
- 1 ไบต์ของข้อมูลระบบตั้งค่าเป็น '00'x
- 4 ไบต์เก็บความยาวของข้อมูล LOB ที่จัดการโดยตัวชี้, ด้านล่าง
- 8 ไบต์ของค่าศูนย์ในฐานสิบหก
- 16 ไบต์เก็บค่าตัวชี้ไปยังข้อมูล LOB ที่เก็บไว้ใน Journal Entry.

ส่วนแรกของโครงสร้างนี้เจตนาให้จัดตำแหน่งที่ละ 16 ไบต์กับตัวชี้ไปยังข้อมูล LOB. จำนวนไบต์ในส่วนนี้จะขึ้นอยู่กับความยาวของคอลัมน์ที่ดำเนินการกับคอลัมน์ LOB. สำหรับตัวอย่างของวิธีการคำนวณความยาวของส่วนแรกนี้ให้อ้างอิงถึงส่วนด้านบนที่เกี่ยวกับการแสดงผลโครงสร้างของคอลัมน์ LOB .

การใช้ user-defined distinct types (UDT)

user-defined distinct type คือกลไกที่ทำให้คุณขยายความสามารถของ DB2 ให้มีชนิดข้อมูลมากไปกว่าที่มีอยู่.

User-defined distinct types ทำให้คุณสามารถกำหนดชนิดข้อมูลสำหรับ DB2 ขึ้นมาใหม่ ซึ่งให้ความสามารถที่มากขึ้นเนื่องจากคุณไม่จำเป็นต้องถูกจำกัดให้ใช้แค่ชนิดข้อมูลในตัวที่ระบบจัดเตรียมให้ในการจำลองแบบทางธุรกิจและโครงสร้างข้อมูลอีกต่อไป. ชนิดข้อมูลแบบ Distinct อนุญาตให้คุณจับคู่แบบหนึ่งต่อหนึ่งกับชนิดที่มีอยู่แล้วในฐานข้อมูลได้.

มีประโยชน์หลายอย่างที่เชื่อมโยงกับ UDT:

- ความสามารถในการต่อขยาย.

ด้วยการนิยามชนิดขึ้นมาใหม่, คุณสามารถเพิ่มชุดของชนิดใน DB2 เพื่อสนับสนุนแอ็พพลิเคชันของคุณได้อย่างไม่จำกัด.

- **ความยืดหยุ่น.**

คุณสามารถระบุความหมายและพฤติกรรมของชนิดใหม่ของคุณได้โดยใช้ User-defined Functions (UDFs) เพื่อเพิ่มความหลากหลายของชนิดที่ใช้ได้ในระบบ.

- **พฤติกรรมที่ไม่เปลี่ยนแปลง.**

ความเข้มงวดในเรื่องชนิดจะทำให้มั่นใจว่า UDTs ของคุณจะทำงานอย่างเหมาะสม. ซึ่งจะรับประกันว่าเฉพาะฟังก์ชันที่นิยามบน UDT ของคุณเท่านั้นที่จะถูกใช้กับ Instance ของ UDT.

- **การห่อหุ้ม (Encapsulation).**

พฤติกรรมของ UDT ของคุณจะถูกจำกัดโดยฟังก์ชันและตัวดำเนินการที่ใช้ได้กับ UDT ของคุณเท่านั้น. สิ่งนี้ทำให้เกิดความยืดหยุ่นในการนำไปปฏิบัติเนื่องจากการทำงานของแอปพลิเคชันไม่ได้ขึ้นอยู่กับการแทนค่าภายในที่คุณเลือกสำหรับชนิดข้อมูลของคุณ.

- **พฤติกรรมที่สามารถขยายได้.**

การนิยามของ User-defined Function บนชนิดสามารถเพิ่มหน้าที่ซึ่งจัดเตรียมไว้เพื่อดำเนินการกับ UDT ได้ตลอดเวลา.

- **พื้นฐานสำหรับส่วนขยาย object-oriented.**

UDTs คือส่วนขยายของคุณลักษณะที่สำคัญของ object-oriented. มันคือก้าวที่สำคัญสู่การเป็นส่วนขยายของ.

หลักการที่เกี่ยวข้อง

“ประเภทที่ผู้ใช้กำหนด (User-defined types)” ในหน้า 13

ประเภทที่ผู้ใช้กำหนด คือ ประเภทข้อมูลจำเพาะที่ผู้ใช้สามารถกำหนดได้โดยไม่ขึ้นกับประเภทข้อมูลที่มีอยู่ในระบบจัดการฐานข้อมูล.

การนิยาม UDT

UDT จะถูกนิยามด้วยคำสั่ง CREATE DISTINCT TYPE.

สำหรับคำสั่ง CREATE DISTINCT TYPE, โปรดสังเกตว่า:

1. ชื่อของ UDT ใหม่อาจเป็นชื่อที่ครบตามเกณฑ์หรือไม่ครบตามเกณฑ์ก็ได้.
2. ชนิดต้นฉบับของ UDT ถูกใช้โดยระบบเพื่อแทนค่า UDT ภายใน. ด้วยเหตุผลนี้, จึงจำเป็นต้องเป็นชนิดข้อมูลในตัว. UDT ที่นิยามขึ้นมาก่อนหน้าจะไม่สามารถใช้เป็นชนิดต้นฉบับของ UDT ได้.

เนื่องจากเป็นส่วนของการนิยาม UDT, ระบบจะสร้างฟังก์ชันที่แปลงชนิดข้อมูลให้เสมอเพื่อ:

- แปลงชนิดข้อมูลจาก UDT ไปเป็นชนิดต้นฉบับ, โดยใช้ชื่อมาตรฐานของชนิดต้นฉบับ. ตัวอย่างเช่น, ถ้าคุณสร้าง Distinct Type โดยอยู่บนพื้นฐานของ FLOAT, แล้วฟังก์ชันการแปลงชนิดข้อมูลที่ชื่อ DOUBLE จะถูกสร้างขึ้นมา.
- แปลงจากชนิดต้นฉบับไปเป็น UDT.

ฟังก์ชันนี้จะสำคัญมากสำหรับการดำเนินการของ UDT ในเคียวรี.

ฟังก์ชันพวกนี้ใช้เพื่อแก้ปัญหการอ้างอิงใดๆ ไปยังชื่อชนิดหรือฟังก์ชันที่ไม่ถูกต้อง, ยกเว้นแต่ชื่อชนิดหรือฟังก์ชันนั้นเป็นอ็อบเจกต์หลักของคำสั่ง CREATE, DROP, หรือ COMMENT ON.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การใช้การอ้างอิงฟังก์ชันที่ครบตามเกณฑ์” ในหน้า 208

ถ้าคุณใช้การอ้างอิงฟังก์ชันที่ครบตามเกณฑ์, คุณต้องจำกัดการค้นหาสำหรับฟังก์ชันที่ตรงกันกับแบบแผนนั้น.

ข้อมูลที่เกี่ยวข้อง

CREATE DISTINCT TYPE statement

ตัวอย่าง: เงินตรา:

ในตัวอย่างนี้, สมมติว่า คุณกำลังเขียนแอ็พพลิเคชันที่จำเป็นต้องจัดการกับระบบเงินตราหลายๆ ระบบ และต้องการตรวจสอบว่า DB2 ไม่อนุญาตให้ระบบเงินตราเหล่านี้ถูกเปรียบเทียบ หรือนำมาใช้ร่วมกันโดยตรงกับในเคียวรีอื่น.

โปรดจำไว้ว่าการแปลงค่าจะมีความจำเป็นถ้าคุณต้องการเปรียบเทียบค่าของระบบเงินตราที่ต่างกัน. ดังนั้นคุณนิยาม UDT ได้มากเท่าที่คุณต้องการ; หนึ่งในแต่ละระบบเงินตราที่คุณอาจจำเป็นต้องแทนค่าคือ:

```
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL (9,2)
CREATE DISTINCT TYPE CANADIAN_DOLLAR AS DECIMAL (9,2)
CREATE DISTINCT TYPE EURO AS DECIMAL (9,2)
```

ตัวอย่าง: ประวัติย่อของผู้สมัครงาน:

ในตัวอย่างนี้, สมมติว่า คุณต้องการเก็บแบบฟอร์มสมัครงานที่เติมข้อมูลโดยผู้สมัครงานกับบริษัทของคุณไว้ในตาราง และคุณกำลังจะใช้ฟังก์ชันเพื่อดึงข้อมูลจากแบบฟอร์มเหล่านี้.

เนื่องจากฟังก์ชันเหล่านี้ไม่สามารถใช้ได้กับสตริงอักขระทั่วไป (เพราะว่าไม่สามารถค้นหาข้อมูลที่ควรจะคืนค่ามาได้), คุณจึงนิยาม UDT เพื่อแทนค่าฟอร์มที่เติมข้อมูลแล้ว:

```
CREATE DISTINCT TYPE PERSONAL.APPLICATION_FORM AS CLOB(32K)
```

การนิยามตารางด้วย UDT

หลังจากที่คุณได้นิยาม UDT หลายชนิดแล้ว, คุณสามารถเริ่มนิยามตารางด้วยคอลัมน์ที่มีชนิดเป็น UDT ได้.

ต่อไปนี้เป็นตัวอย่างการใช้ CREATE TABLE.

ตัวอย่าง: การขาย:

สมมติว่า คุณต้องการนิยามตารางเพื่อเก็บยอดขายของบริษัทของคุณในประเทศต่างๆ.

คุณสร้างตารางได้ดังนี้:

```
CREATE TABLE US_SALES
  (PRODUCT_ITEM INTEGER,
   MONTH        INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR         INTEGER CHECK (YEAR > 1985),
   TOTAL        US_DOLLAR)

CREATE TABLE CANADIAN_SALES
  (PRODUCT_ITEM INTEGER,
   MONTH        INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR         INTEGER CHECK (YEAR > 1985),
   TOTAL        CANADIAN_DOLLAR)

CREATE TABLE GERMAN_SALES
  (PRODUCT_ITEM INTEGER,
   MONTH        INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR         INTEGER CHECK (YEAR > 1985),
   TOTAL        EURO)
```

UDT ในตัวอย่างด้านบนนี้ถูกสร้างโดยใช้คำสั่ง CREATE DISTINCT TYPE เดียวกันใน “ตัวอย่าง: เงินตรา” ในหน้า 243. โปรดสังเกตว่า ตัวอย่างด้านบนจะใช้ข้อจำกัดการตรวจสอบ.

ตัวอย่าง: แบบฟอร์มสมัครงาน:

สมมติว่า คุณต้องการนิยามตารางเพื่อเก็บฟอร์มที่กรอกโดยผู้สมัคร

สร้างตารางได้ดังนี้:

```
CREATE TABLE APPLICATIONS
  (ID          INTEGER,
   NAME       VARCHAR (30),
   APPLICATION_DATE DATE,
   FORM       PERSONAL.APPLICATION_FORM)
```

คุณต้องใช้ชื่อ UDT แบบครบถ้วนตามเกณฑ์เนื่องจาก qualifier ไม่ใช่ Authorization ID เดียวกับคุณ และคุณไม่ได้เปลี่ยนแปลงดีฟอลต์ฟังก์ชันพาร. โปรดจำไว้ว่าเมื่อใดก็ตามที่ชื่อชนิดหรือชื่อฟังก์ชันไม่ถูกต้องตามเกณฑ์แล้ว, DB2 จะค้นหาในรายชื่อ schemas ของฟังก์ชันพารปัจจุบันเพื่อหาชื่อชนิดหรือชื่อฟังก์ชันที่ใกล้เคียงกัน.

การดำเนินการ UDT

หนึ่งในแนวคิดที่สำคัญที่สุดที่เกี่ยวข้องกับ UDT คือ *ความเข้มงวดในเรื่องชนิด*. ความเข้มงวดในเรื่องชนิดจะรับประกันว่าเฉพาะฟังก์ชันและตัวดำเนินการที่ถูกลนิยามบน UDT เท่านั้นที่สามารถใช้ได้กับ instance ของมัน.

ความเข้มงวดในเรื่องชนิดจะสำคัญมากในการทำให้มั่นใจว่า instance ของ UDT ของคุณนั้นถูกต้อง. ตัวอย่างเช่น, ถ้าคุณได้นิยามฟังก์ชันเพื่อแปลงดอลลาร์สหรัฐไปเป็นดอลลาร์แคนาดาตามอัตราแลกเปลี่ยนปัจจุบัน, คุณไม่ต้องการให้ฟังก์ชันเดียวกันนี้ถูกใช้ในการแปลงค่าเงินยูโรไปเป็นแคนาดา เนื่องจากฟังก์ชันนี้จะคืนค่าที่ผิดอย่างแน่นอน.

ผลที่ตามมาของการเข้มงวดในเรื่องชนิด, DB2 จะไม่อนุญาตให้คุณเขียนเคียวรีที่ทำการเปรียบเทียบ, อย่างเช่น, ระหว่าง instance ของ UDT กับ instance ของ UDT ต้นฉบับ. ด้วยเหตุผลเดียวกัน, DB2 จะไม่อนุญาตให้คุณใช้ฟังก์ชันที่ถูกลนิยามบนชนิดอื่นกับ UDTs. ถ้าคุณต้องการเปรียบเทียบ instances ของ UDT กับ instance ชนิดอื่น, คุณจำเป็นต้องทำการแปลง instance ใด instance หนึ่ง. ในทำนองเดียวกัน, คุณจำเป็นต้องแปลง instance ของ UDT ให้เป็นชนิดของพารามิเตอร์ของฟังก์ชันที่ไม่ได้ถูกลนิยามบน UDT ถ้าคุณต้องการใช้ฟังก์ชันนี้.

ตัวอย่าง: การใช้ UDT

สิ่งนี้คือตัวอย่างของการใช้ UDT.

ตัวอย่าง: การเปรียบเทียบระหว่าง UDTs และค่าคงที่:

สมมติว่า คุณต้องการรู้ว่า สินค้าไหนที่ขายไปมากกว่า 100 000.00 ดอลลาร์สหรัฐในสหรัฐอเมริกาเมื่อเดือนกรกฎาคม, 1998 (7/98).

```
SELECT PRODUCT_ITEM
FROM   US_SALES
WHERE  TOTAL > US_DOLLAR (100000)
AND    month = 7
AND    year  = 1998
```

เนื่องจากคุณไม่สามารถเปรียบเทียบดอลลาร์สหรัฐกับ instance ต้นฉบับของดอลลาร์สหรัฐ (ซึ่งคือ, DECIMAL) ได้โดยตรง, คุณจึงใช้ฟังก์ชันการแปลงที่จัดเตรียมโดย DB2 เพื่อแปลงจาก DECIMAL ให้เป็นดอลลาร์สหรัฐ. คุณยังสามารถใช้ฟังก์ชันการแปลงอื่นที่จัดเตรียมโดย DB2 (ซึ่งคือ, ตัวที่ใช้แปลงจากดอลลาร์สหรัฐไปเป็น DECIMAL) และแปลงคอลัมน์ผลรวมไปเป็น DECIMAL. ไม่ว่าคุณจะใช้การแปลงอย่างไร, แปลงไปหรือแปลงกลับเป็น UDT, คุณสามารถใช้สัญลักษณ์ค่ากำหนดการแปลงเพื่อทำการแปลงได้, หรือใช้สัญลักษณ์หน้าที่. คุณสามารถเขียนเคียวรีข้างบนให้เป็นดังนี้:

```
SELECT PRODUCT_ITEM
FROM   US_SALES
WHERE  TOTAL > CAST (100000 AS us_dollar)
AND    MONTH = 7
AND    YEAR  = 1998
```

ตัวอย่าง: การแปลงระหว่าง UDT:

สมมติว่า คุณต้องการนิยาม UDF ที่ทำการแปลงดอลลาร์แคนาดาไปเป็นดอลลาร์สหรัฐ.

สมมติว่าคุณสามารถนำค่าอัตราแลกเปลี่ยนปัจจุบันมาจากไฟล์ที่อยู่ภายนอก DB2. คุณจึงนิยาม UDF ที่รับค่าในแบบดอลลาร์แคนาดา, เรียกดูไฟล์อัตราแลกเปลี่ยนเงินตราและคืนค่าเป็นจำนวนเงินในหน่วยดอลลาร์สหรัฐ.

ในครั้งแรกที่ดู, UDF นี้อาจรู้สึกว่ายาก. อย่างไรก็ตาม, คอมไพเลอร์ภาษา C ไม่ทุกตัวที่สนับสนุนค่า DECIMAL. UDTs ที่เป็นตัวแทนระบบเงินตราต่างๆอาจถูกนิยามให้เป็นแบบ DECIMAL. UDF ของคุณจึงอาจจำเป็นต้องรับและคืนค่าเป็นค่า DOUBLE, เนื่องจากค่านี้เป็นชนิดข้อมูลเดี่ยวเท่านั้นที่จัดเตรียมโดยภาษา C ซึ่งอนุญาตให้ใช้แทนค่า DECIMAL ได้โดยไม่สูญเสียความแม่นยำของทศนิยม. UDF ของคุณจึงควรนิยามดังนี้:

```
CREATE FUNCTION CDN_TO_US_DOUBLE(DOUBLE) RETURNS DOUBLE
EXTERNAL NAME 'MYLIB/CURRENCIES(C_CDN_US)'
LANGUAGE C
PARAMETER STYLE DB2SQL
NO SQL
NOT DETERMINISTIC
```

อัตราแลกเปลี่ยนเงินตราระหว่างดอลลาร์แคนาดาและดอลลาร์สหรัฐอาจมีการเปลี่ยนแปลงในระหว่างการเรียก UDF สองครั้ง, ดังนั้นคุณจึงประกาศให้เป็น NOT DETERMINISTIC.

คำถามคือ, คุณจะทำการส่งผ่านค่าดอลลาร์แคนาดาไปยัง UDF นี้และรับค่าดอลลาร์สหรัฐจาก UDF นี้ได้อย่างไร? ค่าดอลลาร์แคนาดาต้องถูกแปลงชนิดให้เป็นค่า DECIMAL. ค่า DECIMAL จะต้องถูกแปลงชนิดให้เป็น DOUBLE. และคุณยังจำเป็นต้องคืนค่า DOUBLE ที่ถูกแปลงชนิดให้เป็น DECIMAL และค่า DECIMAL ที่ถูกแปลงชนิดให้เป็นดอลลาร์สหรัฐ.

การแปลงจะทำให้อย่างอัตโนมัติโดย DB2 ทุกครั้งที่คุณนิยาม UDF ต้นฉบับ, โดยที่พารามิเตอร์และค่าคืนกลับมีชนิดไม่ตรงกับพารามิเตอร์และค่าที่คืนกลับของฟังก์ชันต้นฉบับ. ดังนั้น, คุณจึงจำเป็นต้องนิยาม UDF ต้นฉบับสองตัว. ตัวแรกจะนำค่า DOUBLE แล้วแทนค่าเป็น DECIMAL. ตัวที่สองจะนำค่า DECIMAL แล้วแทนค่าเป็น UDT. นิยามได้ดังต่อไปนี้:

```
CREATE FUNCTION CDN_TO_US_DEC (DECIMAL(9,2)) RETURNS DECIMAL(9,2)
SOURCE CDN_TO_US_DOUBLE (DOUBLE)
```

```
CREATE FUNCTION US_DOLLAR (CANADIAN_DOLLAR) RETURNS US_DOLLAR
SOURCE CDN_TO_US_DEC (DECIMAL())
```

โปรดสังเกตว่าการเรียกของฟังก์ชัน US_DOLLAR เป็นแบบ US_DOLLAR(C1), ซึ่ง C1 คือคอลัมน์ที่ชนิดคือดอลลาร์แคนาดา, จะมีผลเช่นเดียวกับการเรียก:

```
US_DOLLAR (DECIMAL(CDN_TO_US_DOUBLE (DOUBLE (DECIMAL (C1))))))
```

นั่นคือ, C1(ในดอลลาร์แคนาดา)จะถูกแปลงชนิดให้เป็น DECIMAL ซึ่งจะถูกละทิ้งให้เป็นค่า DOUBLE อีกทีหนึ่งก่อนที่จะผ่านค่าไปยังฟังก์ชัน CDN_TO_US_DOUBLE. ฟังก์ชันนี้จะใช้ไฟล์อัตราแลกเปลี่ยนเงินตราและคือค่า DOUBLE(ที่แทนจำนวนดอลลาร์สหรัฐ)ที่จะแปลงชนิดไปเป็น DECIMAL, แล้วจึงแปลงชนิดไปเป็นดอลลาร์สหรัฐอีกทีหนึ่ง.

ฟังก์ชันที่แปลงค่าเงินยูโรไปเป็นดอลลาร์สหรัฐจะคล้ายกับตัวอย่างข้างบน:

```
CREATE FUNCTION EURO_TO_US_DOUBLE(DOUBLE)
  RETURNS DOUBLE
  EXTERNAL NAME 'MYLIB/CURRENCIES(C_EURO_US)'
  LANGUAGE C
  PARAMETER STYLE DB2SQL
  NO SQL
  NOT DETERMINISTIC

CREATE FUNCTION EURO_TO_US_DEC (DECIMAL(9,2))
  RETURNS DECIMAL(9,2)
  SOURCE EURO_TO_US_DOUBLE(DOUBLE)

CREATE FUNCTION US_DOLLAR(EURO) RETURNS US_DOLLAR
  SOURCE EURO_TO_US_DEC (DECIMAL())
```

ตัวอย่าง: การเปรียบเทียบที่มี UDT รวมอยู่ด้วย:

สมมติว่า คุณต้องการรู้ว่า ผลิตภัณฑ์ไหนที่ขายในสหรัฐอเมริกามากกว่าในแคนาดา และเยอรมันสำหรับเดือนมีนาคม, 2003 (3/03).

ใช้คำสั่ง SELECT ต่อไปนี้:

```
SELECT US.PRODUCT_ITEM, US.TOTAL
  FROM US_SALES AS US, CANADIAN_SALES AS CDN, GERMAN_SALES AS GERMAN
 WHERE US.PRODUCT_ITEM = CDN.PRODUCT_ITEM
 AND US.PRODUCT_ITEM = GERMAN.PRODUCT_ITEM
 AND US.TOTAL > US_DOLLAR (CDN.TOTAL)
 AND US.TOTAL > US_DOLLAR (GERMAN.TOTAL)
 AND US.MONTH = 3
 AND US.YEAR = 2003
 AND CDN.MONTH = 3
 AND CDN.YEAR = 2003
 AND GERMAN.MONTH = 3
 AND GERMAN.YEAR = 2003
```

เนื่องจากคุณไม่สามารถทำการเปรียบเทียบดอลลาร์สหรัฐกับดอลลาร์แคนาดา หรือค่าเงินยูโร, คุณจึงใช้ UDF เพื่อแปลงจำนวนดอลลาร์แคนาดาให้เป็นดอลลาร์สหรัฐ, UDF เพื่อแปลงจำนวนในค่าเงินยูโรให้เป็นดอลลาร์สหรัฐ. คุณไม่สามารถแปลงค่าทั้งหมดให้เป็น DECIMAL แล้วเปรียบเทียบค่า DECIMAL ที่ถูกแปลงแล้วได้เนื่องจากจำนวนเงินนี้ไม่สามารถเปรียบเทียบกันได้เพราะว่าไม่ได้อยู่ในระบบเงินตราเดียวกัน.

ตัวอย่าง: UDF ต้นฉบับที่มี UDT รวมอยู่ด้วย:

สมมติว่า คุณได้นิยาม UDF บนฟังก์ชันในตัว SUM เพื่อสนับสนุน SUM ของค่าเงินยูโร.

คำสั่งฟังก์ชันจะเป็นดังนี้:

```
CREATE FUNCTION SUM (EURO)
RETURNS EURO
SOURCE SYSIBM.SUM (DECIMAL())
```

คุณต้องการรู้ยอดรวมของการขายในเยอรมันสำหรับแต่ละผลิตภัณฑ์ในปี 2004. และคุณต้องการยอดรวมการขายในสกุลเงินดอลลาร์สหรัฐ:

```
SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL)) FROM GERMAN_SALES
WHERE YEAR = 2004
GROUP BY PRODUCT_ITEM
```

คุณไม่สามารถใช้ SUM (US_DOLLAR (TOTAL)), ได้จนกว่าคุณจะกำหนดฟังก์ชัน SUM ของดอลลาร์สหรัฐในแบบเดียวกับด้านบน.

สิ่งอ้างอิงที่เกี่ยวข้อง

“ตัวอย่าง: การกำหนดค่าที่มี UDT ที่ต่างกันรวมอยู่ด้วย” ในหน้า 248

สมมติว่า คุณได้นิยาม UDF ต้นฉบับบนฟังก์ชันในตัว SUM ไว้สองตัว เพื่อสนับสนุน SUM ของดอลลาร์สหรัฐ และดอลลาร์แคนาดา, ที่คล้ายกับ UDF ต้นฉบับของค่าเงินยูโรในตัวอย่าง: UDF ต้นฉบับที่มี UDT รวมอยู่.

ตัวอย่าง: การกำหนดค่าที่มี UDT รวมอยู่ด้วย:

สมมติว่า คุณต้องการเก็บฟอร์มที่กรอกโดยผู้สมัครใหม่เข้าไปในฐานข้อมูล.

คุณได้นิยามตัวแปรโฮสต์ที่เก็บค่าสตริงอักขระที่ใช้เพื่อเป็นตัวแทนฟอร์มที่ถูกกรอกแล้ว:

```
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS CLOB(32K) hv_form;
EXEC SQL END DECLARE SECTION;

/* Code to fill hv_form */

INSERT INTO APPLICATIONS
VALUES (134523, 'Peter Holland', CURRENT DATE, :hv_form)
```

คุณไม่ได้เรียกฟังก์ชันการแปลงเพื่อแปลงสตริงอักขระไปเป็น UDT personal.application_form โดยตรง. นั่นก็เพราะ DB2 ยอมให้คุณกำหนดค่า instance ของซอร์สชนิด UDT ให้กับปลายทางที่มี UDT นั้นได้.

สิ่งอ้างอิงที่เกี่ยวข้อง

“ตัวอย่าง: การกำหนดค่าใน SQL แบบไดนามิก”

ถ้าคุณต้องการใช้คำสั่งเดียวกับในตัวอย่าง: การกำหนดค่าที่ใช้ UDT ใน SQL แบบ dynamic, คุณสามารถใช้ตัวทำเครื่องหมายพารามิเตอร์.

ตัวอย่าง: การกำหนดค่าใน SQL แบบไดนามิก:

ถ้าคุณต้องการใช้คำสั่งเดียวกับในตัวอย่าง: การกำหนดค่าที่ใช้ UDT ใน SQL แบบ dynamic, คุณสามารถใช้ตัวทำเครื่องหมายพารามิเตอร์.

คำสั่งเป็นดังนี้:

```

EXEC SQL BEGIN DECLARE SECTION;
    long id;
    char name[30];
    SQL TYPE IS CLOB(32K) form;
    char command[80];
EXEC SQL END DECLARE SECTION;

    /* Code to fill host variables */

strcpy(command,"INSERT INTO APPLICATIONS VALUES");
strcat(command,"(?, ?, CURRENT DATE, ?)");

EXEC SQL PREPARE APP_INSERT FROM :command;
EXEC SQL EXECUTE APP_INSERT USING :id, :name, :form;

```

คุณใช้ค่ากำหนดการแปลงของ DB2 เพื่อบอกกับ DB2 ว่าประเภทของตัวทำเครื่องหมายเป็นแบบ CLOB(32K), ซึ่งเป็นชนิดที่สามารถกำหนดให้กับคอลัมน์ UDT ได้. โปรดจำไว้ว่าคุณไม่สามารถประกาศตัวแปรโฮสต์ของชนิด UDT ได้, เนื่องจากภาษาโฮสต์ไม่ได้สนับสนุน UDT. ดังนั้น, คุณไม่สามารถระบุชนิดของตัวทำเครื่องหมายพารามิเตอร์ให้เป็น UDT ได้.

สิ่งอ้างอิงที่เกี่ยวข้อง

“ตัวอย่าง: การกำหนดค่าที่มี UDT รวมอยู่ด้วย” ในหน้า 247

สมมติว่า คุณต้องการเก็บฟอร์มที่กรอกโดยผู้สมัครใหม่เข้าไปในฐานข้อมูล.

ตัวอย่าง: การกำหนดค่าที่มี UDT ที่ต่างกันรวมอยู่ด้วย:

สมมติว่า คุณได้นิยาม UDF ต้นฉบับบนฟังก์ชันในตัว SUM ไว้สองตัว เพื่อสนับสนุน SUM ของดอลลาร์สหรัฐ และดอลลาร์แคนาดา, ที่คล้ายกับ UDF ต้นฉบับของค่าเงินยูโรในตัวอย่าง: UDF ต้นฉบับที่มี UDT รวมอยู่.

```

CREATE FUNCTION SUM (CANADIAN_DOLLAR)
    RETURNS CANADIAN_DOLLAR
    SOURCE SYSIBM.SUM (DECIMAL())

CREATE FUNCTION SUM (US_DOLLAR)
    RETURNS US_DOLLAR
    SOURCE SYSIBM.SUM (DECIMAL())

```

สมมติว่า หัวหน้าของคุณร้องขอให้คุณเก็บยอดรวมการขายทั้งปีในแบบดอลลาร์สหรัฐของแต่ละผลิตภัณฑ์ในแต่ละประเทศ, ในตารางที่แยกกัน:

```

CREATE TABLE US_SALES_04
    (PRODUCT_ITEM INTEGER,
     TOTAL          US_DOLLAR)

CREATE TABLE GERMAN_SALES_04
    (PRODUCT_ITEM INTEGER,
     TOTAL          US_DOLLAR)

CREATE TABLE CANADIAN_SALES_04
    (PRODUCT_ITEM INTEGER,
     TOTAL          US_DOLLAR)

INSERT INTO US_SALES_04

```

```
SELECT PRODUCT_ITEM, SUM (TOTAL)
FROM US_SALES
WHERE YEAR = 2004
GROUP BY PRODUCT_ITEM
```

```
INSERT INTO GERMAN_SALES_04
SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL)) FROM GERMAN_SALES
WHERE YEAR = 2004
GROUP BY PRODUCT_ITEM
```

```
INSERT INTO CANADIAN_SALES_04
SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL)) FROM CANADIAN_SALES
WHERE YEAR = 2004
GROUP BY PRODUCT_ITEM
```

คุณจึงทำการแปลงจำนวนเงินในแบบดอลลาร์แคนาดา และค่าเงินยูโรไปเป็นดอลลาร์สหรัฐ เนื่องจาก UDT ที่ต่างกันจะไม่สามารถกำหนดค่าให้ UDT ตัวอื่นได้. คุณไม่สามารถใช้ไวยากรณ์ค่ากำหนดการแปลงได้ เนื่องจาก UDT สามารถถูกแปลงให้เป็นชนิดต้นฉบับได้เท่านั้น.

สิ่งอ้างอิงที่เกี่ยวข้อง

“ตัวอย่าง: UDF ต้นฉบับที่มี UDT รวมอยู่ด้วย” ในหน้า 246

สมมติว่า คุณได้นิยาม UDF บนฟังก์ชันในตัว SUM เพื่อสนับสนุน SUM ของค่าเงินยูโร.

ตัวอย่าง: การใช้ UDT ในคำสั่ง UNION:

สมมติว่า คุณต้องการให้ผู้ใช้ในสหรัฐอเมริกาใช้เคียวรีที่แดงยอดขายของแต่ละผลิตภัณฑ์ของบริษัทของคุณ.

คำสั่ง SELECT จะเป็นดังนี้:

```
SELECT PRODUCT_ITEM, MONTH, YEAR, TOTAL
FROM US_SALES
UNION
SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR (TOTAL)
FROM CANADIAN_SALES
UNION
SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR (TOTAL)
FROM GERMAN_SALES
```

คุณสามารถแปลงดอลลาร์แคนาดาให้เป็นดอลลาร์สหรัฐ และแปลงค่าเงินยูโรให้เป็นดอลลาร์สหรัฐได้ เนื่องจาก UDT สามารถรวมเข้ากันได้กับ UDT เดียวกันเท่านั้น. คุณต้องใช้สัญลักษณ์เพื่อแปลงชนิดระหว่าง UDT เนื่องจากค่ากำหนดการแปลงจะอนุญาตให้คุณแปลงระหว่าง UDT และชนิดต้นฉบับของมันเท่านั้น.

ตัวอย่างการใช้ UDT, UDF, และ LOB

ตัวอย่างต่อไปนี้แสดงวิธีการที่คุณสามารถใช้ประเภทแบบผู้ใช้กำหนดเอง (UDT), ฟังก์ชันแบบผู้ใช้กำหนดเอง (UDF), และอ็อบเจ็กต์ขนาดใหญ่ (LOBs) พร้อมกันในแอ็พพลิเคชันที่ซับซ้อน.

ตัวอย่าง: การนิยาม UDT และ UDF

สมมติว่าคุณต้องการเก็บจดหมายอิเล็กทรอนิกส์ (อีเมล) ที่ส่งมายังบริษัทของคุณไว้ในตาราง.

โดยไม่สนใจเรื่องความเป็นส่วนตัวแล้ว, คุณวางแผนที่จะเขียนเคียวรีกับอีเมลเพื่อหาหัวข้อ, ความบ่อยของอีเมลเซอร์วิสที่ถูกใช้รับคำสั่งซื้อของลูกค้า, และอื่นๆ. อีเมลสามารถมีขนาดใหญ่ได้, และจะมีโครงสร้างภายในที่ซับซ้อน (ผู้ส่ง, ผู้รับ, เรื่อง, วันที่, และเนื้อหาอีเมล). ดังนั้น, คุณตัดสินใจเก็บอีเมลโดยใช้ UDT ที่ชนิดต้นฉบับคืออ็อบเจ็กต์ขนาดใหญ่. คุณนิยามชุดของ UDFs บนชนิดอีเมลของคุณ, เช่น ฟังก์ชันเพื่อดึงข้อมูลชื่อเรื่องของอีเมล, ชื่อผู้ส่ง, วันที่, และอื่นๆ. และคุณยังได้นิยามฟังก์ชันที่สามารถทำการค้นหาข้อความของอีเมลได้. คุณทำตั้งด้านบนโดยการใส่คำสั่ง CREATE ดังต่อไปนี้:

```
CREATE DISTINCT TYPE E_MAIL AS BLOB (1M)
```

```
CREATE FUNCTION SUBJECT (E_MAIL)
  RETURNS VARCHAR (200)
  EXTERNAL NAME 'LIB/PGM(SUBJECT)'
  LANGUAGE C
    PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
```

```
CREATE FUNCTION SENDER (E_MAIL)
  RETURNS VARCHAR (200)
  EXTERNAL NAME 'LIB/PGM(SENDER)'
  LANGUAGE C
    PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
```

```
CREATE FUNCTION RECEIVER (E_MAIL)
  RETURNS VARCHAR (200)
  EXTERNAL NAME 'LIB/PGM(RECEIVER)'
  LANGUAGE C
    PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
```

```
CREATE FUNCTION SENDING_DATE (E_MAIL)
  RETURNS DATE CAST FROM VARCHAR(10)
  EXTERNAL NAME 'LIB/PGM(SENDING_DATE)'
  LANGUAGE C
    PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
```

```
CREATE FUNCTION CONTENTS (E_MAIL)
  RETURNS BLOB (1M)
  EXTERNAL NAME 'LIB/PGM(CONTENTS)'
  LANGUAGE C
    PARAMETER STYLE DB2SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
```

```
CREATE FUNCTION CONTAINS (E_MAIL, VARCHAR (200))
```



```

RETURNS INTEGER
EXTERNAL NAME 'LIB/PGM(CONTAINS)'  

LANGUAGE C
    PARAMETER STYLE DB2SQL
NO SQL
    DETERMINISTIC
NO EXTERNAL ACTION

```

```

CREATE TABLE ELECTRONIC_MAIL  

    (ARRIVAL_TIMESTAMP TIMESTAMP,  

    MESSAGE E_MAIL)

```

ตัวอย่าง: การใช้ฟังก์ชัน LOB เพื่อใส่ค่าเข้าไปในฐานข้อมูล

สมมติว่า คุณใส่ค่าเข้าไปในตารางของคุณโดยการย้ายอีเมลของคุณที่เก็บไว้ในไฟล์เข้าไปใน DB2 UDB for iSeries.

รันคำสั่ง INSERT ต่อไปนี้หลายๆ ครั้ง โดยใช้ค่า HV_EMAIL_FILE ที่ต่างกันจนกว่าคุณจะจัดเก็บอีเมลได้ทั้งหมด:

```

EXEC SQL BEGIN DECLARE SECTION
    SQL TYPE IS BLOB_FILE HV_EMAIL_FILE;

EXEC SQL END DECLARE SECTION
    strcpy (HV_EMAIL_FILE.NAME, "/u/mail/email/mbox");
    HV_EMAIL_FILE.NAME_LENGTH = strlen(HV_EMAIL_FILE.NAME);
    HV_EMAIL_FILE.FILE_OPTIONS = 2;

EXEC SQL INSERT INTO ELECTRONIC_MAIL
    VALUES (CURRENT_TIMESTAMP, :hv_email_file);

```

ฟังก์ชันทั้งหมดที่มีใน DB2 ฟังก์ชันที่เกี่ยวกับ LOB สามารถใช้ได้กับ UDTs ที่มีชนิดต้นฉบับเป็น LOBs. ดังนั้น, คุณได้ใช้ตัวแปรอ้างอิงไฟล์ของ LOB เพื่อกำหนดค่าของไฟล์ให้กับคอลัมน์ของ UDT. คุณไม่ได้ใช้ฟังก์ชันการแปลงเพื่อแปลงค่าของชนิด BLOB ไปเป็นชนิดอีเมลของคุณ. นี่เป็นเพราะว่า DB2 อนุญาตให้คุณกำหนดค่าของชนิดต้นฉบับของ distinct type ให้กับปลายทางของ distinct type.

ตัวอย่าง: การใช้ UDF เพื่อเคียวรี instance ของ UDT

สมมติว่า คุณต้องการรู้จำนวนอีเมลที่ส่งไปให้ลูกค้าเกี่ยวกับคำสั่งซื้อของลูกค้า และคุณมีอีเมลแอดเดรสของลูกค้าอยู่ในตารางลูกค้า.

คำสั่งเป็นดังนี้:

```

SELECT COUNT (*)
    FROM ELECTRONIC_MAIL AS EMAIL, CUSTOMERS
    WHERE SUBJECT (EMAIL.MESSAGE) = 'customer order'
    AND CUSTOMERS.EMAIL_ADDRESS = SENDER (EMAIL.MESSAGE)
    AND CUSTOMERS.NAME = 'Customer X'

```

คุณได้ใช้ UDFs นิยามบน UDT ในเคียวรี SQL นี้เนื่องจากเป็นวิธีเดียวเท่านั้นที่ใช้ดำเนินการกับ UDT. ในกรณีนี้, อีเมลแบบ UDT ของคุณจะถูกห่อหุ้ม(encapsulated)อย่างสมบูรณ์. การแทนค่าและโครงสร้างภายในจะถูกซ่อนและสามารถถูกดำเนินการโดย UDFs ที่นิยามไว้เท่านั้น. UDF เหล่านี้จะวิธีตีความข้อมูลโดยไม่จำเป็นต้องเปิดเผยค่าที่แทนที่อยู่.

สมมติว่า คุณต้องการรู้รายละเอียดอีเมลทั้งหมดที่บริษัทของคุณได้รับในปี 1994 ที่ทำให้ต้องปรับปรุงประสิทธิภาพผลิตภัณฑ์ของคุณในตลาด.

```

SELECT SENDER (MESSAGE), SENDING_DATE (MESSAGE), SUBJECT (MESSAGE)
FROM ELECTRONIC_MAIL
WHERE CONTAINS (MESSAGE,
'performance" AND "products" AND "marketplace"') = 1

```

คุณสามารถใช้ contains UDF ที่สามารถวิเคราะห์เนื้อหาของการค้นหาข้อความสำหรับคีย์เวิร์ด หรือคำเหมือนที่ตรงประเด็น.

ตัวอย่าง: การใช้ LOB locator เพื่อจัดการกับ instance ของ UDT

สมมุติว่า คุณต้องการดึงข้อมูลของอีเมลที่กำหนด แต่ไม่ต้องการย้ายอีเมลทั้งหมดไปยังตัวแปรโฮสต์ในแอพลิเคชันโปรแกรมของคุณ.

โปรดจำไว้ว่า อีเมลสามารถมีขนาดใหญ่ได้. เนื่องจาก UDT ถูกนิยามเป็น LOB, คุณสามารถใช้ LOB locator สำหรับจุดประสงค์นั้นได้:

```

EXEC SQL BEGIN DECLARE SECTION
    long hv_len;
    char hv_subject[200];
    char hv_sender[200];
    char hv_buf[4096];
    char hv_current_time[26];
    SQL TYPE IS BLOB_LOCATOR hv_email_locator;
EXEC SQL END DECLARE SECTION

EXEC SQL SELECT MESSAGE
    INTO :hv_email_locator
    FROM ELECTRONIC_MAIL
    WHERE ARRIVAL_TIMESTAMP = :hv_current_time;

EXEC SQL VALUES (SUBJECT (E_MAIL(:hv_email_locator))
    INTO :hv_subject;
.... code that checks if the subject of the e_mail is relevant ....
.... if the e_mail is relevant, then.....

EXEC SQL VALUES (SENDER (CAST (:hv_email_locator AS E_MAIL)))
    INTO :hv_sender;

```

เนื่องจากตัวแปรโฮสต์ของคุณเป็นชนิด BLOB locator (ชนิดต้นฉบับของ UDT), คุณจึงทำการแปลง BLOB locator ไปเป็น UDT ของคุณโดยตรง, เมื่อใดก็ตามที่มันถูกใช้เป็นอาร์กิวเมนต์ของ UDF ที่ถูกนิยามบน UDT.

การใช้ DataLink

ชนิดข้อมูล DataLink เป็นหนึ่งในส่วนพื้นฐานของการขยายชนิดของข้อมูลที่สามารถเก็บไว้ในไฟล์ฐานข้อมูลได้. แนวคิดของ DataLink คือข้อมูลที่เก็บจริงในคอลัมน์ที่เป็นตัวชี้ไปยังอ็อบเจกต์.

อ็อบเจกต์ที่วันนี้สามารถเป็นอะไรก็ได้, ไฟล์รูปภาพ, เสียงที่บันทึกไว้, ไฟล์ข้อความ, และอื่นๆ. วิธีที่ใช้สำหรับอ้างอิงไปหาอ็อบเจกต์คือการเก็บ Uniform Resource Locator (URL). นี่ก็หมายความว่าแถวในตารางสามารถถูกใช้เก็บข้อมูลเกี่ยวกับอ็อบเจกต์ในชนิดข้อมูลแบบเดิม, และตัวอ็อบเจกต์เองก็สามารถถูกอ้างอิงถึงโดยใช้ชนิดข้อมูล DataLink ได้. ผู้ใช้สามารถใช้ฟังก์ชันแบบ Scalar ของ SQL เพื่อดึงค่าพารามิเตอร์ของอ็อบเจกต์และเซิร์ฟเวอร์ที่เก็บอ็อบเจกต์นั้น (ดูที่เรื่องฟังก์ชันในตัว ในส่วนอ้างอิงของ SQL). ด้วยชนิดข้อมูล DataLink, จะมีความสัมพันธ์แบบหลวมๆ ระหว่างแถวและอ็อบเจกต์. สำหรับ instance, การลบแถวจะตัดความสัมพันธ์ไปยังอ็อบเจกต์ที่อ้างอิงถึงโดย DataLink, แต่ตัวอ็อบเจกต์เองอาจจะไม่ถูกลบ.

ตารางที่ถูกสร้างโดยมีคอลัมน์ DataLink สามารถใช้เก็บข้อมูลเกี่ยวกับอ็อบเจกต์ได้, โดยไม่จำเป็นต้องเก็บอ็อบเจกต์นั้นจริง. แนวคิดนี้ให้ความยืดหยุ่นแก่ผู้ใช้ในเรื่องชนิดของข้อมูลที่สามารถจัดการได้โดยใช้ตาราง. ถ้า, ตัวอย่างเช่น, ผู้ใช้มีวิดีโอคลิปอยู่หลายพันที่เก็บไว้ในระบบไฟล์รวมของเซิร์ฟเวอร์ของพวกเขา, พวกเขาอาจจะต้องการใช้ตาราง SQL เพื่อเก็บข้อมูลเกี่ยวกับวิดีโอคลิปเหล่านั้น. เนื่องจากผู้ใช้มีอ็อบเจกต์ที่เก็บอยู่ในไดเรกทอรีอยู่แล้ว, พวกเขาจึงต้องการตาราง SQL เพื่อใช้อ้างอิงไปยังอ็อบเจกต์เหล่านั้น, ไม่ได้ต้องการไว้เป็นที่เก็บข้อมูลจริง. ทางออกที่ดีก็คือการใช้ DataLinks. ตาราง SQL อาจใช้ชนิดข้อมูล SQL แบบเดิมเพื่อเก็บข้อมูลของแต่ละคลิป, เช่น ชื่อเรื่อง, ความยาว, วันที่, เป็นต้น. แต่ตัวคลิปเองจะถูกอ้างอิงจากคอลัมน์ของ DataLink. แต่ละแถวในตารางจะเก็บค่า URL ของอ็อบเจกต์และหมายเหตุต่างๆ. ดังนั้นแอปพลิเคชันที่ทำงานกับคลิปสามารถดึงค่า URL โดยใช้อินเตอร์เฟซของ SQL ได้, แล้วจึงใช้บราวเซอร์หรือซอฟต์แวร์อื่นเพื่อทำงานกับ URL และแสดงผลวิดีโอคลิป.

มีประโยชน์หลายอย่างในการใช้เทคนิคนี้:

- ระบบไฟล์รวมสามารถเก็บ stream file ใดๆ ก็ได้.
- ระบบไฟล์รวมสามารถเก็บอ็อบเจกต์ที่มีขนาดใหญ่หลายๆ ได้, ซึ่งไม่พอถ้าเป็นคอลัมน์ชนิดตัวอักษร, หรือแม้กระทั่งคอลัมน์ชนิด LOB
- ธรรมชาติของความเป็นลำดับชั้นของระบบไฟล์รวมเหมาะสมดีกับการจัดระบบและการทำงานกับอ็อบเจกต์แบบ stream file.
- ด้วยการปล่อยให้อ็อบเจกต์จริงอยู่ภายนอกฐานข้อมูลและอยู่ในระบบไฟล์รวม, แอปพลิเคชันสามารถได้รับประสิทธิภาพที่ดีกว่าโดยการให้รันไทม์เอ็นจินของ SQL จัดการกับเคียวรีและรายงาน, และให้ระบบไฟล์จัดการกับวิดีโอ, การแสดงภาพ, ข้อความ, และอื่นๆ.

การใช้ DataLinks ยังให้การควบคุมบนอ็อบเจกต์ในขณะที่สถานะของอ็อบเจกต์คือ "linked" ได้. คอลัมน์ DataLink สามารถถูกสร้างให้อ็อบเจกต์ที่ถูกอ้างอิงถึงไม่สามารถถูกลบ, ถูกย้าย, หรือถูกเปลี่ยนชื่อในขณะที่แถวในตาราง SQL นั้นกำลังอ้างอิงถึงอ็อบเจกต์อยู่. จะถือว่าอ็อบเจกต์นี้ถูกเชื่อมโยงอยู่. เมื่อแถวที่มีการอ้างอิงอยู่ถูกลบออก, อ็อบเจกต์จะถูกยกเลิกการเชื่อมโยง. เมื่อต้องการทำความเข้าใจกับแนวคิดนี้ทั้งหมด, คุณควรเข้าใจระดับของตัวควบคุมที่สามารถถูกระบุเมื่อทำการสร้างคอลัมน์ DataLink ด้วย.

ข้อมูลที่เกี่ยวข้อง

ชนิดข้อมูล

ระดับของการควบคุมลิงก์ใน DataLinks

คุณสามารถสร้างคอลัมน์ DataLink ได้ด้วยการควบคุมลิงก์ที่แตกต่างกัน.

ระดับของการควบคุมที่แตกต่างกันเหล่านี้ได้แก่:

NO LINK CONTROL:

เมื่อคอลัมน์ถูกสร้างเป็นระดับ NO LINK CONTROL, จะไม่มีการเชื่อมโยงเกิดขึ้น เมื่อแถวถูกเพิ่มเข้าไปยังตาราง SQL. URL จะถูกตรวจสอบความถูกต้องของไวยากรณ์, แต่จะไม่มีการตรวจสอบเพื่อให้แน่ใจว่าเซิร์ฟเวอร์สามารถเข้าถึงได้หรือไม่, หรือไม่ตรวจสอบว่าไฟล์นั้นมีอยู่จริงหรือไม่.

FILE LINK CONTROL พร้อมด้วยสิทธิ์สำหรับ FS:

เมื่อคอลัมน์ DataLink ที่ถูกสร้างเป็นระดับ FILE LINK CONTROL พร้อมด้วยสิทธิ์สำหรับระบบไฟล์ (FS), ระบบจะตรวจสอบว่าค่า DataLink ใดๆ เป็น URL ที่ถูกต้องหรือไม่, ด้วยชื่อเซิร์ฟเวอร์ และชื่อไฟล์ที่ต้องการ.

ไฟล์ต้องมีอยู่จริงในเวลาที่แถวถูกแทรกเข้าไปในตาราง SQL. เมื่อเจออ็อบเจกต์นั้น, มันจะถูกทำเครื่องหมายให้เป็นถูกเชื่อมโยงอยู่. นั่นก็หมายความว่าอ็อบเจกต์ไม่สามารถถูกย้าย, ถูกลบ, หรือถูกเปลี่ยนชื่อในขณะที่อ็อบเจกต์นี้ถูกเชื่อมโยงอยู่. ด้วยเหมือนกัน, อ็อบเจกต์ไม่สามารถถูกเชื่อมโยงได้มากกว่าหนึ่งการเชื่อมโยง. ถ้าส่วนของชื่อเซิร์ฟเวอร์ของ URL ระบุถึงระบบทางไกล (remote system), ระบบนั้นต้องสามารถเข้าถึงได้. ถ้าแถวที่เก็บค่า DataLink ถูกลบ, อ็อบเจกต์จะถูกยกเลิกการเชื่อมโยง. ถ้าค่า DataLink ถูกอัปเดตให้เป็นค่าอื่น, อ็อบเจกต์เดิมจะถูกยกเลิกการเชื่อมโยง, และอ็อบเจกต์ใหม่จะถูกเชื่อมโยง.

ระบบไฟล์รวมยังคงรับภาระสำหรับการจัดการการอนุญาตสำหรับอ็อบเจกต์ที่ถูกเชื่อมโยง. การอนุญาตจะไม่ถูกเปลี่ยนแปลงขณะดำเนินการเชื่อมต่อหรือยกเลิกการเชื่อมต่อ. ตัวเลือกนี้ให้การควบคุมการมีอยู่ของอ็อบเจกต์สำหรับช่วงระยะเวลาที่อ็อบเจกต์ถูกเชื่อมโยงอยู่.

FILE LINK CONTROL พร้อมด้วยสิทธิสำหรับ DB:

เมื่อคอลัมน์ DataLink ถูกสร้างขึ้นเป็นแบบ FILE LINK CONTROL พร้อมด้วยสิทธิสำหรับฐานข้อมูล (DB), URL จะถูกตรวจสอบความถูกต้อง, และสิทธิที่มีอยู่ทั้งหมดของอ็อบเจกต์จะถูกลบออก.

ความเป็นเจ้าของของอ็อบเจกต์จะถูกเปลี่ยนให้กับโปรไฟล์ผู้ใช้ชนิดพิเศษที่ระบบจัดเตรียมไว้ให้. ช่วงระยะเวลาที่อ็อบเจกต์ถูกเชื่อมโยงอยู่, สามารถเข้าถึงอ็อบเจกต์ได้ทางเดียวเท่านั้นโดยการดึงค่า URL จากตาราง SQL ที่มีอ็อบเจกต์ที่ถูกเชื่อมโยงอยู่. การทำเช่นนี้จะถูกจัดการโดยใช้โทเค็นพิเศษที่ใช้ในการเข้าถึงซึ่งจะถูกผนวกต่อท้ายให้กับ URL ที่คืนค่าโดย SQL. ถ้าไม่มีโทเค็นที่ใช้ในการเข้าถึงแล้ว, การพยายามเข้าถึงอ็อบเจกต์ทั้งหมดจะไม่สำเร็จเนื่องจากจะเป็นการละเมิดสิทธิในการทำงาน. ถ้า URL ที่มีโทเค็นที่ใช้ในการเข้าถึงถูกดึงค่ามาจากตาราง SQL โดยวิธีปกติ (FETCH, SELECT INTO, อื่นๆ.) แล้วตัวกรองของระบบไฟล์จะตรวจสอบโทเค็นที่ใช้ในการเข้าถึงและจึงอนุญาตให้เข้าถึงอ็อบเจกต์ได้.

ตัวเลือกนี้ได้ในการควบคุมในการป้องกันการเปลี่ยนแปลงอ็อบเจกต์ที่ถูกเชื่อมโยงอยู่สำหรับผู้ใช้ที่พยายามเข้าถึงอ็อบเจกต์โดยตรง. เนื่องจากสามารถเข้าถึงอ็อบเจกต์ได้ทางเดียวโดยการดึงค่าโทเค็นที่ใช้ในการเข้าถึงมาจากการดำเนินการ SQL, ผู้ดูแลระบบสามารถควบคุมการเข้าถึงอ็อบเจกต์ที่ถูกเชื่อมโยงอยู่ได้อย่างมีประสิทธิภาพโดยใช้การอนุญาตของฐานข้อมูลกับตาราง SQL ที่มีคอลัมน์ DataLink.

คำสั่งที่ใช้สำหรับทำงานกับ DataLinks

การสนับสนุนสำหรับชนิดข้อมูล DataLink สามารถแบ่งย่อยได้เป็นส่วนประกอบสามส่วน.

1. สำหรับ DB2 มีชนิดข้อมูลที่เรียกว่า DATALINK. ชนิดข้อมูลนี้สามารถระบุในคำสั่ง SQL เช่น CREATE TABLE และ ALTER TABLE ได้. คอลัมน์จะไม่สามารถมีค่าดีฟอลต์อื่นนอกจาก NULL. การเข้าถึงข้อมูลต้องใช้อินเตอร์เฟสของ SQL เท่านั้น. ที่เป็นเช่นนี้เนื่องจากตัว DataLink เองจะเข้ากันไม่ได้กับชนิดข้อมูลโฮสต์ใดๆ ก็ตาม. ฟังก์ชันแบบ Scalar ของ SQL สามารถถูกใช้เพื่อดึงค่า DataLink ในรูปแบบอักขระได้. จะมีฟังก์ชันแบบ Scalar ชื่อ DLVALUE ที่ต้องใช้ใน SQL เพื่อ INSERT และ UPDATE ค่าในคอลัมน์.
2. DataLink File Manager (DLFM) คือส่วนประกอบที่ดูแลสถานะของลิงก์สำหรับไฟล์หรือเซิร์ฟเวอร์, ตรวจสอบการเปลี่ยนแปลงของ meta-data สำหรับแต่ละไฟล์. โค้ดนี้จะจัดการกับการเชื่อมโยง, การยกเลิกการเชื่อมโยง, และคำสั่งเกี่ยวกับ commitment control. ด้านที่สำคัญของ DataLinks คือ DLFM ไม่จำเป็นต้องอยู่บนระบบเดียวกันกับตาราง SQL ที่เก็บคอลัมน์ DataLink. ดังนั้นตาราง SQL สามารถเชื่อมโยงไปยังอ็อบเจกต์ที่อยู่ในระบบไฟล์รวมเดียวกัน, หรืออยู่ในระบบไฟล์รวมทางไกลก็ได้.
3. ตัวกรองของ DataLink จะต้องทำงานเมื่อระบบไฟล์พยายามดำเนินการกับไฟล์ที่อยู่ในไดเรกทอรีที่ใช้เก็บอ็อบเจกต์ที่ถูกเชื่อมโยง. ส่วนประกอบนี้จะพิจารณาว่าไฟล์ถูกเชื่อมโยงหรือไม่, และโดยทางเลือกแล้ว, จะพิจารณาว่าผู้ใช้มีสิทธิในการ

เข้าถึงไฟล์หรือไม่. ถ้าชื่อไฟล์รวมโทเค็นที่ใช้ในการเข้าถึงไว้ด้วย, โทเค็นนั้นจะถูกตรวจสอบด้วย. เนื่องจากกระบวนการ
กรองนี้จะใช้เวลาเป็นพิเศษ, จึงต้องทำงานเมื่ออ็อบเจ็กต์ที่ถูกใช้อยู่ในหนึ่งในโดเร็กทอรีภายใน DataLink prefix.
ให้ดูการอธิบายด้านล่างเกี่ยวกับ 'prefix'.

เมื่อทำงานกับ DataLinks, จะมีหลายขั้นตอนที่ต้องทำเพื่อตั้งค่าระบบให้เหมาะสม:

- TCP/IP จะต้องถูกตั้งค่าบนระบบใดๆ ที่จะถูกใช้เมื่อทำงานกับ DataLinks. ซึ่งรวมถึงระบบที่กำลังสร้างตาราง SQL ที่มี
คอลัมน์ DataLink, หรือระบบที่มีอ็อบเจ็กต์ที่จะถูกเชื่อม. ในกรณีส่วนใหญ่แล้ว, จะอยู่บนระบบเดียวกัน. เนื่องจาก
URL ที่ถูกใช้ในการอ้างอิงไปยังอ็อบเจ็กต์จะเก็บชื่อเซิร์ฟเวอร์แบบ TCP/IP, ดังนั้นชื่อนี้ต้องรู้จักโดยระบบที่กำลังจะเก็บ.
คำสั่ง CFGTCP สามารถถูกใช้เพื่อตั้งค่าชื่อ TCP/IP ได้, หรือใช้เพื่อลงทะเบียนชื่อเซิร์ฟเวอร์แบบ TCP/IP.
- ระบบที่เก็บตาราง SQL จะต้องมีการ Relational Database Directory ที่ถูกปรับปรุงเพื่อส่งผลกับระบบฐานข้อมูลโลคัล, หรือ
ระบบรีโมตทางเลือกใดๆ. คำสั่ง WRKRDBDIRE สามารถถูกใช้เพื่อเพิ่มหรือแก้ไขข้อมูลในโดเร็กทอรีนี้. สำหรับผลที่ตาม
มา, แนะนำว่าชื่อเซิร์ฟเวอร์แบบ TCP/IP และชื่อ Relational Database ควรจะใช้ชื่อเดียวกัน.
- เซิร์ฟเวอร์ DLFM จะต้องถูกเรียกทำงานบนระบบใดๆ ที่จะใช้เก็บอ็อบเจ็กต์ที่ถูกเชื่อมโยง. คำสั่ง STRTCPSVR *DLFM
สามารถถูกใช้เพื่อเริ่มทำงานเซิร์ฟเวอร์ DLFM ได้. เซิร์ฟเวอร์ DLFM สามารถถูกจบการทำงานได้โดยใช้คำสั่ง CL คือ
ENDTCPSVR *DLFM.

เมื่อ DLFM ถูกเรียกใช้งานแล้ว, มีบางขั้นตอนที่จำเป็นในการตั้งค่า DLFM. ฟังก์ชัน DLFM นี้จะใช้ได้โดยสคริปต์ที่สามารถทำ
งานได้ ซึ่งสคริปต์นั้นสามารถใส่ค่าได้จากอินเตอร์เฟซของ QShell. เมื่อต้องการเข้าถึงเซสชันอินเตอร์เฟซแบบโต้ตอบ, ให้ใช้คำ
สั่ง CL ชื่อ QSH. คำสั่งนี้จะแสดงหน้าจอป้อนคำสั่งขึ้นมาซึ่งทำให้คุณสามารถป้อนคำสั่งสคริปต์ DLFM ได้. คำสั่งสคริปต์
"dfmadmin -help" สามารถใช้เพื่อแสดงข้อความวิธีใช้และไดอะแกรมของไวยากรณ์. สำหรับฟังก์ชันที่ถูกใช้บ่อยแล้ว, คำสั่ง
CL ยังได้ถูกจัดเตรียมไว้ด้วย. โดยการใช้คำสั่ง CL แล้ว, การตั้งค่า DLFM ทั้งหมดหรือเกือบทั้งหมดสามารถทำให้สำเร็จได้
โดยไม่ต้องใช้สคริปต์อินเตอร์เฟซ. ขึ้นอยู่กับความชอบของคุณ, คุณสามารถเลือกได้ว่าจะใช้คำสั่งสคริปต์จากหน้าจอป้อนคำ
สั่งของ QSH หรือว่าเลือกใช้คำสั่ง CL จากหน้าจอป้อนคำสั่งของ CL.

เนื่องจากฟังก์ชันเหล่านี้จำเป็นสำหรับผู้ดูแลระบบหรือผู้ดูแลฐานข้อมูล, ดังนั้นจึงต้องการสิทธิในการใช้แบบ *IOSYSCFG.

การเพิ่ม prefix

prefix คือพารามิเตอร์หรือโดเร็กทอรีที่จะเก็บอ็อบเจ็กต์ที่ถูกเชื่อมโยง. เมื่อเริ่มติดตั้ง Data Links File Manager (DLFM) บนระบบ,
ผู้ดูแลระบบต้องเพิ่ม prefix ใดๆ ที่จะถูกใช้สำหรับ DataLink ด้วย. คำสั่งสคริปต์ "dfmadmin -add_prefix" จะถูกใช้เพื่อเพิ่ม
"prefix". คำสั่ง CL ที่ใช้ในการเพิ่ม prefix คือ คำสั่ง Add Prefix to DataLink File Manager (ADDPFXDLFM).

ตัวอย่างเช่น, บนเซิร์ฟเวอร์ TESTSYS1, มีโดเร็กทอรีชื่อ /mydir/datalinks/ ที่เก็บอ็อบเจ็กต์ที่จะถูกเชื่อมโยง. ผู้ดูแลระบบใช้
คำสั่ง ADDPFXDLFM PREFIX((' /mydir/datalinks/')) เพื่อเพิ่ม "prefix". ลิงก์ของ URLs ต่อไปนี้ถูกต้องเพราะมีพารามิเตอร์
ค่า prefix ถูกต้อง:

```
http://TESTSYS1/mydir/datalinks/videos/file1.mpg
```

หรือ

```
file://TESTSYS1/mydir/datalinks/text/story1.txt
```

และยังเป็นไปได้ที่จะเอา prefix ออกโดยใช้คำสั่งสคริปต์ "dfmadmin -del_prefix". แต่ว่าฟังก์ชันนี้ไม่ได้ถูกใช้บ่อย เนื่องจาก
ฟังก์ชันสามารถถูกเรียกใช้ได้อีกต่อเมื่อไม่มีอ็อบเจ็กต์ที่ถูกเชื่อมโยงเหลืออยู่ในโครงสร้างโดเร็กทอรีที่เก็บอยู่ภายในชื่อ prefix.

หมายเหตุ:

1. ไตเร็กทอรีต่อไปนี้, รวมถึงไตเร็กทอรีย่อย, ไม่ควรนำมาใช้เป็น prefix สำหรับ DataLink:
 - /QIBM
 - /QReclaim
 - /QSR
 - /QFPNWSSTG
2. นอกจากนี้แล้ว, ไม่ควรนำไตเร็กทอรีพื้นฐานเหล่านี้มาใช้เว้นแต่มี prefix เป็นไตเร็กทอรีย่อยอยู่ในไตเร็กทอรีเหล่านี้:
 - /home
 - /dev
 - /bin
 - /etc
 - /tmp
 - /usr
 - /lib

การเพิ่มฐานข้อมูลโฮสต์

ฐานข้อมูลโฮสต์ คือระบบฐานข้อมูลเชิงสัมพันธ์ ซึ่งการร้องขอลิงก์เริ่มมาจากที่นี่. ถ้า DLFM อยู่บนระบบเดียวกันกับตาราง SQL ที่จะเก็บ DataLinks แล้ว, เฉพาะชื่อฐานข้อมูลโลคัลเท่านั้นที่จำเป็นในการใส่เพิ่ม. ถ้า DLFM จะมีการร้องขอลิงก์ที่มาจากระบบรีโมตแล้ว, ชื่อของระบบทั้งหมดต้องถูกลงทะเบียนด้วย DLFM. คำสั่งสคริปต์ที่ใช้ในการเพิ่มฐานข้อมูลโฮสต์คือ `dfmadmin -add_db` และ คำสั่ง CL คือคำสั่ง Add Host Database to DataLink File Manager (ADDHDBDLFM). ฟังก์ชันนี้ยังต้องการให้ไลบรารีที่เก็บตาราง SQL ถูกลงทะเบียนด้วย.

ตัวอย่างเช่น, บนเซิร์ฟเวอร์ TESTSYS1, ที่คุณได้เพิ่ม "prefix" /mydir/datalinks/ เข้าไปแล้ว, คุณต้องการตาราง SQL บนระบบโลคัลในไลบรารี TESTDB หรือ PRODDB ให้ถูกอนุญาตเพื่อเชื่อมโยงอ็อบเจกต์บนเซิร์ฟเวอร์นี้ได้. ใช้คำสั่งต่อไปนี้:

```
ADDHDBDLFM HOSTDBLIB((TESTDB) (PRODDB)) HOSTDB(TESTSYS1)
```

เมื่อ DLFM ถูกเรียกทำงานแล้ว, และ "prefix" และชื่อฐานข้อมูลโฮสต์ได้ถูกลงทะเบียนแล้ว, คุณสามารถเริ่มเชื่อมโยงอ็อบเจกต์ในระบบไฟล์ได้.

การใช้คำสั่ง SQL ในสภาวะแวดล้อมที่ต่างกัน

คุณสามารถใช้ SQL ในสภาวะแวดล้อมต่างๆ ได้. ซึ่งบางสภาวะแวดล้อมจะถูกอธิบายไว้ที่นี่.

การใช้เคอร์เซอร์

เมื่อ SQL รันข้อความที่เลือก, แถวผลลัพธ์จะประกอบขึ้นจากตารางผลลัพธ์. เคอร์เซอร์จะแสดงวิธีการเข้าถึงตารางผลลัพธ์.

โดยจะถูกใช้ภายในโปรแกรม SQL เพื่อรักษาตำแหน่งในตารางผลลัพธ์. SQL จะใช้เคอร์เซอร์ให้ทำงานร่วมกับแถวในตารางผลลัพธ์และเพื่อให้แถวเหล่านั้นใช้ได้กับโปรแกรมของคุณ. โปรแกรมของคุณสามารถมีหลายเคอร์เซอร์, แม้ว่าแต่ละเคอร์เซอร์ต้องมีชื่อเฉพาะก็ตาม.

ข้อความที่เกี่ยวกับการใช้เคอร์เซอร์ประกอบด้วย:

- ข้อความ DECLARE CURSOR เพื่อกำหนดและตั้งชื่อเคอร์เซอร์และระบุแถวที่ต้องเรียกออกมาด้วยข้อความที่ใส่อยู่ที่เลือกไว้.
- ข้อความ OPEN และ CLOSE เพื่อเปิดและปิดเคอร์เซอร์สำหรับใช้ภายในโปรแกรม. ต้องเปิดเคอร์เซอร์ก่อนที่จะเรียกแถวใดๆ ออกมา.
- คำสั่ง FETCH เพื่อค้นหาแถวจากตารางผลลัพธ์ของเคอร์เซอร์ หรือเพื่อวางตำแหน่งเคอร์เซอร์บนอีกแถวหนึ่ง.
- ข้อความ UPDATE ... WHERE CURRENT OF เพื่ออัปเดตแถวปัจจุบันของเคอร์เซอร์.
- ข้อความ DELETE ... WHERE CURRENT OF เพื่อลบแถวปัจจุบันของเคอร์เซอร์.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การอัปเดตข้อมูลตามที่ดึงมาจากตาราง” ในหน้า 104

คุณสามารถอัปเดตแถวของข้อมูลที่คุณได้ดึงออกมาโดยใช้เคอร์เซอร์.

ข้อมูลที่เกี่ยวข้อง

DECLARE CURSOR statement

CLOSE statement

FETCH statement

DELETE statement

UPDATE statement

ประเภทเคอร์เซอร์

SQL สนับสนุนเคอร์เซอร์แบบอนุกรมและแบบเลื่อนขึ้นลง. ประเภทของเคอร์เซอร์จะกำหนดวิธีการวางตำแหน่ง ซึ่งสามารถนำมาใช้ร่วมกับเคอร์เซอร์ได้.

เคอร์เซอร์แบบอนุกรม

เคอร์เซอร์แบบอนุกรมคือเคอร์เซอร์ที่ถูกกำหนดโดยไม่มีคีย์เวิร์ด SCROLL.

สำหรับเคอร์เซอร์แบบอนุกรมนั้น, ตารางผลลัพธ์แต่ละแถวสามารถดึงข้อมูลออกมาได้เพียงหนึ่งครั้งต่อการเปิดเคอร์เซอร์. เมื่อเคอร์เซอร์ถูกเปิด, จะถูกวางตำแหน่งหน้าแถวแรกในตารางผลลัพธ์. เมื่อมีการใส่ข้อความ FETCH, เคอร์เซอร์จะถูกย้ายไปยังแถวถัดไปในตารางผลลัพธ์. แถวนั้นจะกลายเป็นแถวปัจจุบัน. หากมีการระบุตัวแปรโฮสต์ (ด้วย INTO clause บนคำสั่ง FETCH), SQL จะย้ายเนื้อหาของแถวปัจจุบันเข้าไปในตัวแปรโฮสต์ของโปรแกรม.

จะมีการทำซ้ำลำดับดังกล่าวทุกครั้งที่มีการใส่คำสั่ง FETCH จนกว่าจะสิ้นสุดข้อมูล (SQLCODE = 100). เมื่อสิ้นสุดข้อมูลแล้ว, ให้ปิดเคอร์เซอร์. คุณไม่สามารถเข้าถึงแถวใดๆ ในตารางผลลัพธ์หลังจากที่สิ้นสุดข้อมูลแล้ว. ในการใช้เคอร์เซอร์แบบอนุกรมซ้ำ, ก่อนอื่นคุณต้องปิดเคอร์เซอร์ จากนั้นจึงใส่คำสั่ง OPEN อีกครั้ง. คุณไม่สามารถสำรองข้อมูลได้ด้วยการใช้เคอร์เซอร์แบบอนุกรม.

เคอร์เซอร์แบบเลื่อนได้

สำหรับเคอร์เซอร์แบบเลื่อนขึ้นลงนั้น, สามารถดึงข้อมูลออกจากแถวของตารางผลลัพธ์ได้หลายครั้ง. เคอร์เซอร์จะเคลื่อนผ่านตารางผลลัพธ์โดยยึดตามออฟชันตำแหน่งที่ระบุบนคำสั่ง FETCH. เมื่อเคอร์เซอร์ถูกเปิด, จะถูกวางตำแหน่งหน้าแถวแรกในตารางผลลัพธ์. เมื่อมีการใส่คำสั่ง FETCH, เคอร์เซอร์จะถูกวางตำแหน่งที่แถวในตารางผลลัพธ์ซึ่งถูกระบุด้วยออฟชัน

ตำแหน่ง. แลวนั้นจะกลายเป็นแถวปัจจุบัน. หากมีการระบุตัวแปรโฮสต์ (ด้วย INTO clause บนคำสั่ง FETCH), SQL จะย้ายเนื้อหาของแถวปัจจุบันเข้าไปไว้ในตัวแปรโฮสต์ของโปรแกรม. ไม่สามารถระบุตัวแปรโฮสต์สำหรับอ็อปชันตำแหน่ง BEFORE และ AFTER ได้.

จะมีการทำซ้ำลำดับดังกล่าวทุกครั้งที่มีการใส่คำสั่ง FETCH. ไม่จำเป็นต้องปิดเคอร์เซอร์ เมื่อสิ้นสุดข้อมูลหรือเริ่มต้นข้อมูล. อ็อปชันตำแหน่งทำให้โปรแกรมสามารถดึงข้อมูลแถวออกมาจากรายได้อย่างต่อเนื่อง.

มีการใช้อ็อปชันการเลื่อนต่อไปนี่เพื่อวางตำแหน่งเคอร์เซอร์เมื่อใส่คำสั่ง FETCH. ตำแหน่งเหล่านี้จะสัมพันธ์กับตำแหน่งเคอร์เซอร์ปัจจุบันในตารางผลลัพธ์.

NEXT	วางตำแหน่งเคอร์เซอร์บนแถวถัดไป. นี่คือนำเริ่มต้นหากไม่มีการระบุตำแหน่ง.
PRIOR	วางตำแหน่งเคอร์เซอร์บนแถวก่อนหน้านี้.
FIRST	วางตำแหน่งเคอร์เซอร์บนแถวแรก.
LAST	วางตำแหน่งเคอร์เซอร์บนแถวสุดท้าย.
BEFORE	วางตำแหน่งเคอร์เซอร์หน้าแถวแรก.
AFTER	วางตำแหน่งเคอร์เซอร์หลังแถวสุดท้าย.
CURRENT	ไม่ได้เปลี่ยนตำแหน่งเคอร์เซอร์.
RELATIVE n	ประเมินตัวแปรโฮสต์ หรือจำนวนเต็ม n ที่สัมพันธ์กับตำแหน่งปัจจุบันของเคอร์เซอร์. ตัวอย่างเช่น, ถ้า n เท่ากับ -1 , เคอร์เซอร์จะถูกวางตำแหน่งบนแถวก่อนหน้านี้ของตารางผลลัพธ์. ถ้า n เท่ากับ $+3$, เคอร์เซอร์จะถูกวางตำแหน่งสามแถวหลังแถวปัจจุบัน.

สำหรับเคอร์เซอร์แบบเลื่อนขึ้นลงนั้น, สามารถกำหนดจุดสิ้นสุดของตารางได้โดยใช้:

```
FETCH AFTER FROM C1
```

เมื่อวางตำแหน่งเคอร์เซอร์ที่จุดสิ้นสุดของตารางแล้ว, โปรแกรมสามารถใช้อ็อปชันเลื่อน PRIOR หรือ RELATIVE เพื่อวางตำแหน่งและดึงข้อมูลออกมาโดยเริ่มต้นจากจุดสิ้นสุดของตาราง.

ตัวอย่าง: การใช้เคอร์เซอร์

สมมติว่าโปรแกรมของคุณกำลังตรวจสอบข้อมูลเกี่ยวกับพนักงานในแผนก D11. ตัวอย่างต่อไปนี้จะแสดงคำสั่ง SQL ที่คุณควรรวมอยู่ในโปรแกรม เพื่อกำหนดและใช้เคอร์เซอร์แบบอนุกรมและแบบเลื่อนขึ้นลง.

สามารถใช้เคอร์เซอร์เหล่านี้เพื่อรับข้อมูลเกี่ยวกับแผนกจากรายการ CORPDATA.EMPLOYEE.

สำหรับตัวอย่างเคอร์เซอร์แบบอนุกรม, โปรแกรมจะประมวลผลทุกแถวจากรายการ, อัพเดทงานสำหรับสมาชิกทั้งหมดของแผนก D11 และลบเร็กคอร์ดของพนักงานจากแผนกอื่นๆ.

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

คำสั่งเคอร์เซอร์ SQL แบบอนุกรม

อธิบายไว้ในส่วนนี้

```
EXEC SQL
DECLARE THISEMP CURSOR FOR
  SELECT EMPNO, LASTNAME,
         WORKDEPT, JOB
  FROM CORPDATA.EMPLOYEE
  FOR UPDATE OF JOB
END-EXEC.
```

“ขั้นที่ 1: กำหนดเคอร์เซอร์” ในหน้า 261.

```
EXEC SQL
OPEN THISEMP
END-EXEC.
```

“ขั้นที่ 2: เปิดเคอร์เซอร์” ในหน้า 262.

```
EXEC SQL
WHENEVER NOT FOUND
  GO TO CLOSE-THISEMP
END-EXEC.
```

“ขั้นที่ 3: ระบุสิ่งที่ต้องทำเมื่อถึงจุดสิ้นสุดของข้อมูล” ในหน้า 263.

```
EXEC SQL
FETCH THISEMP
  INTO :EMP-NUM, :NAME2,
        :DEPT, :JOB-CODE
END-EXEC.
```

“ขั้นที่ 4: ดึงค่าแถวโดยใช้เคอร์เซอร์” ในหน้า 263.

```
... for all employees
      in department D11, update
      the JOB value:

EXEC SQL
UPDATE CORPDATA.EMPLOYEE
  SET JOB = :NEW-CODE
  WHERE CURRENT OF THISEMP
END-EXEC.
```

“ขั้นที่ 5a: อัปเดตแถวปัจจุบัน” ในหน้า 264.

```
... then print the row.
```

ตารางที่ 41. ตัวอย่างเคอร์เซอร์แบบอนุกรม (ต่อ)

คำสั่งเคอร์เซอร์ SQL แบบอนุกรม

อธิบายไว้ในส่วนนี้

... for other employees,
delete the row:

“ชั้นที่ 5b: ลบแถวปัจจุบัน” ในหน้า 264.

```
EXEC SQL
DELETE FROM CORPDATA.EMPLOYEE
WHERE CURRENT OF THISEMP
END-EXEC.
```

Branch back to fetch and process the next row.

“ชั้นที่ 6: ปิดเคอร์เซอร์” ในหน้า 265.

```
CLOSE-THISEMP.
EXEC SQL
CLOSE THISEMP
END-EXEC.
```

สำหรับตัวอย่างเคอร์เซอร์แบบเลื่อนขึ้นลง, โปรแกรมจะใช้ออฟชั่นตำแหน่ง RELATIVE เพื่อรับตัวอย่างเงินเดือนจากแผนก D11.

ตารางที่ 42. ตัวอย่างเคอร์เซอร์แบบเลื่อนขึ้นลง

คำสั่งเคอร์เซอร์ SQL แบบเลื่อนขึ้นลง

อธิบายไว้ในส่วนนี้

```
EXEC SQL
DECLARE THISEMP DYNAMIC SCROLL CURSOR FOR
SELECT EMPNO, LASTNAME,
SALARY
FROM CORPDATA.EMPLOYEE
WHERE WORKDEPT = 'D11'
END-EXEC.
```

“ชั้นที่ 1: กำหนดเคอร์เซอร์” ในหน้า 261.

```
EXEC SQL
OPEN THISEMP
END-EXEC.
```

“ชั้นที่ 2: เปิดเคอร์เซอร์” ในหน้า 262.

```
EXEC SQL
WHENEVER NOT FOUND
GO TO CLOSE-THISEMP
END-EXEC.
```

“ชั้นที่ 3: ระบุสิ่งที่ต้องทำเมื่อถึงจุดสิ้นสุดของข้อมูล” ในหน้า 263.

ตารางที่ 42. ตัวอย่างเคอร์เซอร์แบบเลื่อนขึ้นลง (ต่อ)

คำสั่งเคอร์เซอร์ SQL แบบเลื่อนขึ้นลง

อธิบายไว้ในส่วนนี้

```
...initialize program summation
    salary variable
EXEC SQL
    FETCH RELATIVE 3 FROM THISEMP
    INTO :EMP-NUM, :NAME2,
        :JOB-CODE
END-EXEC.

...add the current salary to
    program summation salary
...branch back to fetch and
    process the next row.
```

“ขั้นที่ 4: ดึงค่าแถวโดยใช้เคอร์เซอร์” ในหน้า 263.

```
...calculate the average
    salary
```

```
CLOSE-THISEMP.
EXEC SQL
CLOSE THISEMP
END-EXEC.
```

“ขั้นที่ 6: ปิดเคอร์เซอร์” ในหน้า 265.

ขั้นที่ 1: กำหนดเคอร์เซอร์:

การกำหนดตารางผลลัพธ์เพื่อเข้าถึงด้วยเคอร์เซอร์, ให้ใช้ข้อความ DECLARE CURSOR.

ข้อความ DECLARE CURSOR จะตั้งชื่อเคอร์เซอร์และระบุข้อความที่เลือก. ข้อความที่เลือกจะกำหนดชุดแถวซึ่ง, ตามแนวคิดแล้ว, คือการสร้างตารางผลลัพธ์. สำหรับเคอร์เซอร์แบบอนุกรม, คำสั่งจะเป็นดังนี้ (FOR UPDATE OF clause สามารถเลือกได้):

```
EXEC SQL
DECLARE cursor-name CURSOR FOR
SELECT column-1, column-2 ,...
FROM table-name , ...
FOR UPDATE OF column-2 ,...
END-EXEC.
```

สำหรับเคอร์เซอร์แบบเลื่อนขึ้นลง, คำสั่งจะเป็นดังนี้ (WHERE clause สามารถเลือกได้):

```
EXEC SQL
DECLARE cursor-name SCROLL CURSOR FOR
SELECT column-1, column-2 ,...
FROM table-name ,...
WHERE column-1 = expression ...
END-EXEC.
```

คำสั่ง select ซึ่งแสดงไว้ในที่นี้ค่อนข้างง่าย ๆ. อย่างไรก็ตาม, คุณสามารถโค้ด clause ประเภทอื่นๆ ในข้อความที่เลือกไว้ภายในคำสั่ง DECLARE CURSOR สำหรับเคอร์เซอร์แบบอนุกรมและแบบเลื่อนขึ้นลง.

หากคุณประสงค์ที่จะอัปเดตคอลัมน์ใดๆ ในแถวใดๆ หรือทุกแถวของตารางที่ระบุ (ตารางจะถูกตั้งชื่อใน FROM clause), รวมทั้ง FOR UPDATE OF clause. ตารางจะตั้งชื่อแต่ละคอลัมน์ที่คุณต้องการอัปเดต. หากคุณไม่ได้ระบุชื่อคอลัมน์, และระบุ ORDER BY clause หรือ FOR READ ONLY clause, จะมีการส่งคืน SQLCODE แบบลบหากมีความพยายามในการอัปเดต. ถ้าคุณไม่ระบุ FOR UPDATE OF clause, FOR READ ONLY clause, ORDER BY clause, และตารางผลลัพธ์ที่ไม่ได้เป็นแบบอ่านอย่างเดียว และเคอร์เซอร์เป็นแบบที่เลื่อนไม่ได้, คุณสามารถอัปเดตคอลัมน์ใดๆ ก็ได้ในตารางที่ระบุ.

คุณสามารถอัปเดตคอลัมน์ตารางที่ระบุได้แม้ว่าคอลัมน์นั้นจะไม่ใช่ส่วนหนึ่งของตารางผลลัพธ์ก็ตาม. ในกรณีนี้, คุณไม่ต้องตั้งชื่อคอลัมน์ในข้อความ SELECT. เมื่อเคอร์เซอร์ค้นหาแถว (โดยใช้ FETCH) ซึ่งมีค่าคอลัมน์ที่คุณต้องการอัปเดต, คุณสามารถใช้ UPDATE ... WHERE CURRENT OF เพื่ออัปเดตแถว.

ตัวอย่างเช่น, สมมุติว่าตารางผลลัพธ์แต่ละแถวประกอบด้วยคอลัมน์ EMPNO, LASTNAME, และ WORKDEPT จากตาราง CORPDATA.EMPLOYEE. หากคุณต้องการอัปเดตคอลัมน์ JOB (หนึ่งในคอลัมน์ในแต่ละแถวของตาราง CORPDATA.EMPLOYEE), คำสั่ง DECLARE CURSOR ควรจะประกอบด้วย FOR UPDATE OF JOB ... แม้ว่า JOB จะถูกตัดออกจากคำสั่ง SELECT ก็ตาม.

ตารางผลลัพธ์และเคอร์เซอร์เป็นแบบ *อ่านอย่างเดียว* หากข้อใดข้อหนึ่งต่อไปนี้ถูกต้อง:

- FROM clause แรกจะระบุมากกว่าหนึ่งตารางหรือหนึ่งมุมมอง.
- FROM clause แรกจะระบุมุมมองอ่านอย่างเดียว.
- FROM clause แรกจะระบุฟังก์ชันตารางแบบผู้ใช้กำหนด.
- SELECT clause แรกจะระบุคีย์เวิร์ด DISTINCT.
- การเลือกย่อยด้านนอกประกอบด้วย GROUP BY clause.
- การเลือกย่อยด้านนอกประกอบด้วย HAVING clause.
- SELECT clause แรกประกอบด้วยฟังก์ชันคอลัมน์.
- คำสั่ง select ประกอบด้วยเคียวรี่ย่อยโดยที่อ็อบเจ็กต์ฐานของการเลือกย่อยด้านนอกและของเคียวรี่ย่อยคือตารางเดียวกัน.
- คำสั่ง select จะประกอบด้วยโอเปอเรเตอร์ UNION, UNION ALL, EXCEPT, หรือ INTERSECT.
- คำสั่ง select ประกอบด้วย ORDER BY clause, และไม่มีการระบุคีย์เวิร์ด SENSITIVE และ FOR UPDATE OF clause.
- คำสั่ง select ยังรวมถึง FOR READ ONLY clause.
- คีย์เวิร์ด SCROLL ถูกระบุ, FOR UPDATE OF clause ไม่ถูกระบุ, และคีย์เวิร์ด SENSITIVE ไม่ถูกระบุ.
- รายการที่เลือกประกอบด้วยคอลัมน์ DataLink และไม่ได้ระบุ FOR UPDATE OF clause.
- การเลือกย่อยครั้งแรกต้องมีตารางผลลัพธ์ชั่วคราว.
- คำสั่ง select รวมถึง FETCH FIRST *n* ROWS ONLY.

ขั้นที่ 2: เปิดเคอร์เซอร์:

เพื่อเริ่มการประมวลผลแถวของตารางผลลัพธ์, ให้ใส่คำสั่ง OPEN.

เมื่อโปรแกรมของคุณใส่คำสั่ง OPEN, SQL จะประมวลผลคำสั่งที่เลือกภายในคำสั่ง DECLARE CURSOR เพื่อระบุชุดแถว, เรียกตารางผลลัพธ์, โดยใช้ค่าปัจจุบันของตัวแปรโฮสต์ใดๆ ที่ระบุในคำสั่งที่เลือก. ตารางผลลัพธ์สามารถประกอบด้วยแถว ศูนย์, หนึ่ง, หรือหลายแถว, ขึ้นอยู่กับขอบเขตความต้องการของเงื่อนไขการค้นหา. คำสั่ง OPEN จะเป็นดังนี้:

```
EXEC SQL
OPEN cursor-name
END-EXEC.
```

ขั้นที่ 3: ระบุสิ่งที่ต้องทำเมื่อถึงจุดสิ้นสุดของข้อมูล:

เพื่อค้นหาว่า ตารางผลลัพธ์จะสิ้นสุดเมื่อใด, ให้ทดสอบฟิลด์ SQLCODE สำหรับค่า 100 หรือทดสอบฟิลด์ SQLSTATE สำหรับค่า '02000' (ซึ่งคือ, จุดสิ้นสุดของข้อมูล). เงื่อนไขนี้จะเกิดขึ้น เมื่อคำสั่ง FETCH ทำการค้นหาแถวสุดท้ายในตารางผลลัพธ์ และโปรแกรมของคุณใส่ FETCH ที่ตามมา.

ตัวอย่างเช่น:

```
...
IF SQLCODE =100 GO TO DATA-NOT-FOUND.
```

หรือ

```
IF SQLSTATE ='02000' GO TO DATA-NOT-FOUND.
```

อีกทางเลือกหนึ่งของเทคนิคนี้คือให้โค้ดคำสั่ง WHENEVER. การใช้ WHENEVER NOT FOUND ทำให้เกิดการแยกสาขาไปยังอีกส่วนหนึ่งของโปรแกรม, ซึ่งมีการใส่ข้อความ CLOSE. คำสั่ง WHENEVER จะเป็นดังนี้:

```
EXEC SQL
WHENEVER NOT FOUND GO TO symbolic-address
END-EXEC.
```

โปรแกรมของคุณควรคาดการณ์ถึงเงื่อนไขการสิ้นสุดของข้อมูลเมื่อใดก็ตามที่เคอร์เซอร์ถูกใช้เพื่อดึงข้อมูลแถวออกมา, และควรเตรียมการเพื่อจัดการเมื่อเกิดสถานการณ์เช่นนี้ขึ้น.

เมื่อคุณกำลังใช้เคอร์เซอร์แบบอนุกรม และถึงจุดสิ้นสุดข้อมูล, ทุกๆ คำสั่ง FETCH ที่ตามมาจะย้อนกลับไปเงื่อนไขการสิ้นสุดข้อมูล. คุณไม่สามารถกำหนดตำแหน่งเคอร์เซอร์บนแถวซึ่งถูกประมวลผลเรียบร้อยแล้ว. คำสั่ง CLOSE คือการดำเนินการเพียงอย่างเดียวที่สามารถทำงานบนเคอร์เซอร์ได้.

เมื่อคุณกำลังใช้เคอร์เซอร์แบบเลื่อนขึ้นลง และถึงจุดสิ้นสุดข้อมูล, ตารางผลลัพธ์ยังสามารถประมวลผลข้อมูลเพิ่มเติมได้อีก. คุณสามารถกำหนดตำแหน่งเคอร์เซอร์ได้ทุกที่ในตารางผลลัพธ์ด้วยการใช้การผสมของอ็อปชันตำแหน่ง. คุณไม่จำเป็นต้องปิดเคอร์เซอร์เมื่อสิ้นสุดข้อมูล.

ขั้นที่ 4: ดึงค่าแถวโดยใช้เคอร์เซอร์:

การย้ายเนื้อหาของแถวที่เลือกเข้าไปในตัวแปรโฮสต์ของโปรแกรม, ให้ใช้คำสั่ง FETCH. คำสั่ง SELECT ภายในคำสั่ง DECLARE CURSOR เป็นการจำแนกแถวที่ประกอบด้วยค่าคอลัมน์ที่โปรแกรมต้องการ. อย่างไรก็ตาม, SQL จะไม่นำข้อมูลใดๆ ออกมาสำหรับแอ็พพลิเคชันโปรแกรมจนกว่าจะมีการใส่คำสั่ง FETCH.

เมื่อโปรแกรมของคุณใส่คำสั่ง FETCH, SQL จะใช้ตำแหน่งเคอร์เซอร์ปัจจุบันเป็นจุดเริ่มต้นเพื่อหาตำแหน่งแถวที่ร้องขอในตารางผลลัพธ์. ซึ่งจะเป็นการเปลี่ยนแถวดังกล่าวให้เป็น แถวปัจจุบัน. หากมีการระบุ INTO clause, SQL จะย้ายเนื้อหาของแถวปัจจุบันเข้าไปในตัวแปรโฮสต์ของโปรแกรม. จะมีการซ้ำลำดับนี้ทุกครั้งที่มีการใส่คำสั่ง FETCH.

SQL จะคงไว้ซึ่งตำแหน่งของแถวปัจจุบัน (กล่าวคือ, เคอร์เซอร์จะชี้ไปที่แถวปัจจุบัน) จนกว่าจะมีการใส่คำสั่ง FETCH ถัดไป เพื่อให้มีการออกเคอร์เซอร์. คำสั่ง UPDATE ไม่ได้เป็นการเปลี่ยนตำแหน่งของแถวปัจจุบันภายในตารางผลลัพธ์, แม้ว่าคำสั่ง DELETE ก่อให้เกิดการเปลี่ยนตำแหน่งก็ตาม.

คำสั่ง FETCH สำหรับเคอร์เซอร์แบบอนุกรมจะเป็นดังนี้:

```
EXEC SQL
FETCH cursor-name
INTO :host variable-1[, :host variable-2] ...
END-EXEC.
```

คำสั่ง FETCH สำหรับเคอร์เซอร์แบบเลื่อนขึ้นลงจะเป็นดังนี้:

```
EXEC SQL
FETCH RELATIVE integer
FROM cursor-name
INTO :host variable-1[, :host variable-2] ...
END-EXEC.
```

ขั้นที่ 5a: อัปเดตแถวปัจจุบัน:

เมื่อโปรแกรมของคุณกำหนดตำแหน่งเคอร์เซอร์ที่อยู่บนแถว, คุณสามารถอัปเดตข้อมูลได้ด้วยการใช้ข้อความ UPDATE ที่มี WHERE CURRENT OF clause. WHERE CURRENT OF clause จะระบุเคอร์เซอร์ที่ชี้ไปยังแถวที่คุณต้องการอัปเดต.

คำสั่ง UPDATE ... WHERE CURRENT OF จะเป็นดังนี้:

```
EXEC SQL
UPDATE table-name
SET column-1 = value [, column-2 = value] ...
WHERE CURRENT OF cursor-name
END-EXEC.
```

เมื่อใช้ร่วมกับเคอร์เซอร์, คำสั่ง UPDATE:

- จะอัปเดตเฉพาะแถวเดียว—แถวปัจจุบัน
- จะระบุเคอร์เซอร์ซึ่งชี้ไปที่แถวที่จะอัปเดต
- ต้องการให้คอลัมน์ที่ถูกอัปเดตมีการตั้งชื่อก่อนหน้านั้นใน FOR UPDATE OF clause ของข้อความ DECLARE CURSOR, หากมีการระบุ ORDER BY ด้วยเช่นกัน

หลังจากอัปเดตแถวแล้ว, ตำแหน่งของเคอร์เซอร์จะยังคงอยู่บนแถวนั้น (กล่าวคือ, แถวปัจจุบันของเคอร์เซอร์ไม่ได้เปลี่ยนไป) จนกว่าคุณจะใช้คำสั่ง FETCH สำหรับแถวถัดไป.

ขั้นที่ 5b: ลบแถวปัจจุบัน:

เมื่อโปรแกรมของคุณเรียกแถวปัจจุบันออกมา, คุณสามารถลบแถวดังกล่าวได้ด้วยการใช้คำสั่ง DELETE. ในการปฏิบัติดังกล่าว, ให้คุณใส่คำสั่ง DELETE ที่ถูกออกแบบมาสำหรับการใช้ร่วมกับเคอร์เซอร์; WHERE CURRENT OF clause จะระบุเคอร์เซอร์ซึ่งชี้ไปที่แถวที่คุณต้องการลบออก.

คำสั่ง DELETE ... WHERE CURRENT OF จะเป็นดังนี้:

```
EXEC SQL
DELETE FROM table-name
WHERE CURRENT OF cursor-name
END-EXEC.
```

เมื่อใช้ร่วมกับเคอร์เซอร์, คำสั่ง DELETE:

- จะลบเฉพาะแถวเดียว—แถวปัจจุบัน
- จะใช้ WHERE CURRENT OF clause เพื่อระบุเคอร์เซอร์ซึ่งชี้ไปที่แถวที่ต้องถูกลบออก

หลังจากลบแถวแล้ว, คุณไม่สามารถอัปเดตหรือลบอีกแถวหนึ่งได้โดยใช้เคอร์เซอร์ดังกล่าวจนกว่าคุณจะใส่ข้อความ FETCH เพื่อกำหนดตำแหน่งเคอร์เซอร์.

คุณสามารถใช้คำสั่ง DELETE เพื่อลบแถวทั้งหมดที่ตรงตามเงื่อนไขการค้นหาเฉพาะ. คุณยังสามารถใช้ FETCH และ DELETE ได้. คำสั่ง WHERE CURRENT OF เมื่อคุณต้องการรับก๊อปปี้ของแถว, ตรวจสอบ, แล้วลบออก.

ขั้นที่ 6: ปิดเคอร์เซอร์:

หากคุณประมวลผลแถวของตารางผลลัพธ์สำหรับเคอร์เซอร์แบบอนุกรม, และต้องการใช้เคอร์เซอร์นั้นอีกครั้ง, ให้ใส่คำสั่ง CLOSE เพื่อปิดเคอร์เซอร์ก่อนที่จะเปิดใหม่.

คำสั่งจะเป็นดังนี้:

```
EXEC SQL
CLOSE cursor-name
END-EXEC.
```

หากคุณประมวลผลแถวของตารางผลลัพธ์ และไม่ต้องการใช้เคอร์เซอร์นั้นอีก, คุณสามารถปล่อยให้ระบบปิดเคอร์เซอร์เอง. ระบบจะปิดเคอร์เซอร์โดยอัตโนมัติเมื่อ:

- มีการใส่ COMMIT โดยไม่มีข้อความ HOLD และไม่ได้ประกาศเคอร์เซอร์ด้วยการใช้ WITH HOLD clause.
- มีการใส่ ROLLBACK โดยไม่มีข้อความ HOLD.
- สิ้นสุดงาน.
- สิ้นสุด activation group ends และมีการระบุ CLOSQLCSR(*ENDACTGRP) บนพีริคอมไฟล์.
- โปรแกรม SQL แรกใน call stack สิ้นสุดลงและไม่มีการระบุทั้ง CLOSQLCSR(*ENDJOB) หรือ CLOSQLCSR(*ENDACTGRP) เมื่อโปรแกรมถูกคอมไพล์ล่วงหน้า.
- การเชื่อมต่อไปยังแอ็พพลิเคชันเซิร์ฟเวอร์ถูกทำให้สิ้นสุดลงด้วยการใช้ข้อความ DISCONNECT.
- การเชื่อมต่อไปยังแอ็พพลิเคชันเซิร์ฟเวอร์ถูกรีลีสและเกิด COMMIT ที่สมบูรณ์.
- เกิด *RUW CONNECT.

เนื่องจากเคอร์เซอร์แบบเปิดยังคงรักษาการล็อกบนการอ้างอิงถึงตารางหรือบนมุมมอง, คุณจึงควรปิดเคอร์เซอร์แบบเปิดใดๆ อย่างชัดเจนทันทีที่ไม่จำเป็นต้องใช้งานแล้ว.

การใช้คำสั่ง FETCH แบบหลายแถว

คำสั่ง FETCH แบบหลายแถวสามารถนำมาใช้เพื่อเรียกแถวจำนวนมากออกมาจากตาราง หรือดูคำสั่ง FETCH เดียว. โปรแกรมจะควบคุมการบล็อกแถวตามจำนวนแถวที่ร้องขอบนคำสั่ง FETCH (OVRDBF ไม่มีผลกระทบ).

จำนวนแถวสูงสุดที่สามารถร้องขอบนการเรียกดึงข้อมูลออกแบบเดี่ยวคือ 32767. เมื่อนำข้อมูลออกมาแล้ว, เคอร์เซอร์จะถูกวางตำแหน่งบนแถวสุดท้ายที่ถูกเรียกออกมา.

มีสองวิธีการในการกำหนดหน่วยเก็บในซึ่งแถวที่ถูกดึงข้อมูลออกมาตั้งอยู่: ระเบียบโครงสร้างโฮสต์หรือพื้นที่หน่วยเก็บของแถวซึ่งมี descriptor ที่เกี่ยวข้อง. สามารถโค้ดได้ทั้งสองวิธีในทุกภาษาที่พีคอมไพล์ของ SQL สนับสนุนอยู่, ยกเว้นระเบียบโครงสร้างโฮสต์ใน REXX. รูปแบบทั้งสองของคำสั่ง FETCH แบบหลายแถวยอมให้แอ็พพลิเคชันโค้ดระเบียบตัวบ่งชี้ที่แยกต่างหาก. ระเบียบตัวบ่งชี้ควรมีตัวบ่งชี้หนึ่งตัว indicator สำหรับแต่ละตัวแปรโฮสต์ซึ่งเป็น null.

คำสั่ง FETCH แบบหลายแถวสามารถนำมาใช้ร่วมกับทั้งเคอร์เซอร์แบบอนุกรมและแบบเลื่อนขึ้นลง. การดำเนินการที่ถูกใช้เพื่อกำหนด, เปิด, และปิดเคอร์เซอร์สำหรับคำสั่ง FETCH แบบหลายแถวยังคงเหมือนเดิม. เฉพาะคำสั่ง FETCH เท่านั้นที่เปลี่ยนเพื่อระบุจำนวนแถวที่จะเรียกออกมาและจำนวนหน่วยเก็บในซึ่งมีแถวตั้งอยู่.

หลังคำสั่ง FETCH แบบหลายแถวแต่ละข้อความ, ข้อมูลจะถูกส่งคืนไปยังโปรแกรมผ่าน SQLCA. นอกเหนือจากฟิลด์ SQLCODE และ SQLSTATE แล้ว, SQLERRD จะให้ข้อมูลต่อไปนี้:

- SQLERRD3 ประกอบด้วยจำนวนแถวที่ถูกเรียกออกมาบนข้อความ FETCH แบบหลายแถว. หาก SQLERRD3 น้อยกว่าจำนวนแถวที่ร้องขอ, จะเกิดข้อผิดพลาดหรือเงื่อนไขการสิ้นสุดข้อมูลขึ้น.
- SQLERRD4 ประกอบด้วยความยาวของแต่ละแถวที่ถูกเรียกออกมา.
- SQLERRD5 มีการบ่งชี้ที่ว่าแถวสุดท้ายในตารางถูกดึงข้อมูลออกมา. ซึ่งสามารถนำไปใช้เพื่อตรวจพบเงื่อนไขการสิ้นสุดข้อมูลในตารางที่ถูกดึงข้อมูลออกมาเมื่อเคอร์เซอร์ไม่มีความสามารถในการรับรู้ทันทีเพื่ออัปเดต. เคอร์เซอร์ซึ่งมีความสามารถทันทีในการอัปเดตควรดึงข้อมูลออกอย่างต่อเนื่องจนกว่าจะได้รับ SQLCODE +100 เพื่อตรวจพบเงื่อนไขการสิ้นสุดข้อมูล.

ข้อมูลที่เกี่ยวข้อง

Embedded SQL programming

การดึงข้อมูลออก (FETCH) แบบหลายแถวโดยใช้ระเบียบโครงสร้างโฮสต์:

การใช้ FETCH แบบหลายแถวที่มีระเบียบโครงสร้างโฮสต์, แอ็พพลิเคชันต้องกำหนดระเบียบโครงสร้างโฮสต์ที่ SQL สามารถใช้ได้.

แต่ละภาษาจะมีระเบียบและกฎของตัวเองสำหรับการกำหนดระเบียบโครงสร้างโฮสต์. สามารถกำหนดระเบียบโครงสร้างโฮสต์ได้ด้วยการใช้การประกาศที่เปลี่ยนแปลงได้หรือใช้คำสั่งคอมไพเลอร์เพื่อเรียก External File Descriptions ออกมา (อย่างเช่นคำสั่ง COBOL COPY).

ระเบียบโครงสร้างโฮสต์ประกอบด้วยระเบียบของโครงสร้าง. แต่ละโครงสร้างจะตรงกับแถวหนึ่งแถวของตารางผลลัพธ์. โครงสร้างแรกในระเบียบจะตรงกับแถวแรก, โครงสร้างที่สองในระเบียบจะตรงกับแถวที่สอง, เป็นอาทิ. SQL จะพิจารณาแอ็ททริบิวต์ของไอเท็มขั้นต้นในระเบียบโครงสร้างโฮสต์โดยยึดตามการประกาศระเบียบโครงสร้างโฮสต์. เพื่อประสิทธิภาพการทำงานสูงสุด, แอ็ททริบิวต์ของไอเท็มซึ่งสร้างระเบียบโครงสร้างโฮสต์ควรตรงกับแอ็ททริบิวต์ของคอลัมน์ที่ถูกเรียกออกมา.

พิจารณาตัวอย่าง COBOL ต่อไปนี้:

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.


```
EXEC SQL INCLUDE SQLCA
END-EXEC.
```

...

```
01 TABLE-1.
  02 DEPT OCCURS 10 TIMES.
    05 EMPNO PIC X(6).
    05 LASTNAME.
      49 LASTNAME-LEN PIC S9(4) BINARY.
      49 LASTNAME-TEXT PIC X(15).
    05 WORKDEPT PIC X(3).
    05 JOB PIC X(8).
01 TABLE-2.
  02 IND-ARRAY OCCURS 10 TIMES.
    05 INDS PIC S9(4) BINARY OCCURS 4 TIMES.
```

...

```
EXEC SQL
DECLARE D11 CURSOR FOR
SELECT EMPNO, LASTNAME, WORKDEPT, JOB
FROM CORPDATA.EMPLOYEE
WHERE WORKDEPT = "D11"
END-EXEC.
```

...

```
EXEC SQL
OPEN D11
END-EXEC.
PERFORM FETCH-PARA UNTIL SQLCODE NOT EQUAL TO ZERO.
ALL-DONE.
EXEC SQL CLOSE D11 END-EXEC.
```

...

```
FETCH-PARA.
EXEC SQL WHENEVER NOT FOUND GO TO ALL-DONE END-EXEC.
EXEC SQL FETCH D11 FOR 10 ROWS INTO :DEPT :IND-ARRAY
END-EXEC.
```

...

ในตัวอย่างนี้, มีการกำหนดเคอร์เซอร์สำหรับตาราง CORPDATA.EMPLOYEE เพื่อเลือกแถวทั้งหมดที่คอลัมน์ WORKDEPT มีค่าเท่ากับ 'D11'. ตารางผลลัพธ์ประกอบด้วยแปดแถว. คำสั่ง DECLARE CURSOR และ OPEN ไม่มีซินแทกซ์พิเศษใดๆ เมื่อถูกใช้ร่วมกับคำสั่ง FETCH แบบหลายแถว. คำสั่ง FETCH อีกคำสั่งหนึ่งซึ่งส่งคืนแถวเดี่ยวมาให้กับเคอร์เซอร์เดียวกันสามารถโค้ดได้ที่จุดอื่นในโปรแกรม. คำสั่ง FETCH แบบหลายแถวถูกนำมาใช้เพื่อเรียกแถวทั้งหมดในตารางผลลัพธ์ออกมา. หลัง FETCH, ตำแหน่งเคอร์เซอร์ยังคงอยู่ที่แถวสุดท้ายซึ่งถูกเรียกออกมา.

มีการกำหนดอะเรย์โครงสร้างโฮสต์ DEPT และอะเรย์ตัวบ่งชี้ IND-ARRAY ที่เกี่ยวข้องในแอ็พพลิเคชัน. ทั้งสองอะเรย์มีสิบตำแหน่ง. อะเรย์ตัวบ่งชี้มี entry สำหรับแต่ละคอลัมน์ในตารางผลลัพธ์.

แอ็ททริบิวต์ของประเภทและความยาวของไอเท็มชั้นต้นของอะเรย์โครงสร้างโฮสต์ DEPT ตรงกับ คอลัมน์ซึ่งถูกเรียกออกมา.

เมื่อคำสั่ง FETCH แบบหลายแถวเสร็จสมบูรณ์, อะเรย์โครงสร้างโฮสต์ จะประกอบด้วยข้อมูลสำหรับแถวทั้งหมดแปดแถว. อะเรย์ตัวบ่งชี้, IND_ARRAY, ประกอบด้วยค่าศูนย์สำหรับทุกคอลัมน์ในแถวทุกแถวเนื่องจากไม่มีการส่งคืนค่าศูนย์.

SQLCA ซึ่งถูกส่งคืนมายังแอ็พพลิเคชั่นประกอบด้วยข้อมูลต่อไปนี้:

- SQLCODE ประกอบด้วย 0
- SQLSTATE ประกอบด้วย '00000'
- SQLERRD3 ประกอบด้วย 8, ซึ่งคือจำนวนแถวที่ถูกดึงข้อมูลออกมา
- SQLERRD4 ประกอบด้วย 34, ซึ่งคือความยาวของแต่ละแถว
- SQLERRD5 ประกอบด้วย +100, แสดงว่าแถวสุดท้ายในตารางผลลัพธ์อยู่ในบล็อก

ข้อมูลที่เกี่ยวข้อง

SQLCA (SQL communications area)

การดึงข้อมูลออก (FETCH) แบบหลายแถวโดยใช้พื้นที่หน่วยเก็บของแถว:

แอ็พพลิเคชั่นต้องกำหนดพื้นที่หน่วยเก็บของแถวและพื้นที่รายละเอียดที่เกี่ยวข้องก่อนที่แอ็พพลิเคชั่นจะสามารถใช้ FETCH แบบหลายแถวที่มีพื้นที่หน่วยเก็บของแถวได้. พื้นที่หน่วยเก็บของแถว คือตัวแปรโฮสต์ซึ่งถูกกำหนดในแอ็พพลิเคชั่นโปรแกรม.

พื้นที่หน่วยเก็บของแถวประกอบด้วยผลลัพธ์ของ FETCH แบบหลายแถว. พื้นที่หน่วยเก็บของแถวสามารถเป็นตัวแปรอักขระที่มีไบต์มากพอต่อการรักษาแถวที่ร้องขอทั้งหมดไว้บน FETCH แบบหลายแถว.

SQLDA ซึ่งประกอบด้วย SQLTYPE และ SQLLEN สำหรับแต่ละคอลัมน์ที่ถูกส่งคืนมาถูกกำหนดโดย descriptor ที่เกี่ยวข้อง ซึ่งถูกใช้ในรูปแบบพื้นที่หน่วยเก็บของแถวของ FETCH แบบหลายแถว. ข้อมูลที่ให้ไว้ใน descriptor จะกำหนดข้อมูลที่เริ่มจากฐานข้อมูลไปยังพื้นที่หน่วยเก็บของแถว. เพื่อประสิทธิภาพการทำงานสูงสุด, ข้อมูลแอ็ททริบิวต์ใน descriptor ควรตรงกับแอ็ททริบิวต์ของคอลัมน์ที่ถูกเรียกออกมา.

พิจารณาตัวอย่าง PL/I ต่อไปนี้:

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

```
*.....1.....2.....3.....4.....5.....6.....7...*
```

```
EXEC SQL INCLUDE SQLCA;  
EXEC SQL INCLUDE SQLDA;
```

...

```
DCL DEPTPTR PTR;  
DCL I DEPT(20) BASED(DEPTPTR),  
    3 EMPNO CHAR(6),  
    3 LASTNAME CHAR(15) VARYING,  
    3 WORKDEPT CHAR(3),  
    3 JOB CHAR(8);  
DCL I BIN(31) FIXED;  
DEC J BIN(31) FIXED;
```

```

DCL ROWAREA CHAR(2000);

...

    ALLOCATE SQLDA SET(SQLDAPTR);
EXEC SQL
DECLARE D11 CURSOR FOR
SELECT EMPNO, LASTNAME, WORKDEPT, JOB
FROM CORPDATA.EMPLOYEE
WHERE WORKDEPT = 'D11';

...

EXEC SQL
OPEN D11;
/* SET UP THE DESCRIPTOR FOR THE MULTIPLE-ROW FETCH */
/* 4 COLUMNS ARE BEING FETCHED */
SQLD = 4;
SQLN = 4;
SQLDABC = 366;
SQLTYPE(1) = 452; /* FIXED LENGTH CHARACTER - */
/* NOT NULLABLE */
SQLLEN(1) = 6;
SQLTYPE(2) = 456; /*VARYING LENGTH CHARACTER */
/* NOT NULLABLE */
SQLLEN(2) = 15;
SQLTYPE(3) = 452; /* FIXED LENGTH CHARACTER - */
SQLLEN(3) = 3;
SQLTYPE(4) = 452; /* FIXED LENGTH CHARACTER - */
/* NOT NULLABLE */
SQLLEN(4) = 8;
/*ISSUE THE MULTIPLE-ROW FETCH STATEMENT TO RETRIEVE*/
/*THE DATA INTO THE DEPT ROW STORAGE AREA */
/*USE A HOST VARIABLE TO CONTAIN THE COUNT OF */
/*ROWS TO BE RETURNED ON THE MULTIPLE-ROW FETCH */

J = 20; /*REQUESTS 20 ROWS ON THE FETCH */

...

EXEC SQL
WHENEVER NOT FOUND
GOTO FINISHED;
EXEC SQL
WHENEVER SQLERROR
GOTO FINISHED;
EXEC SQL
FETCH D11 FOR :J ROWS
USING DESCRIPTOR :SQLDA INTO :ROWAREA;
/* ADDRESS THE ROWS RETURNED */
DEPTPTR = ADDR(ROWAREA);
/*PROCESS EACH ROW RETURNED IN THE ROW STORAGE */
/*AREA BASED ON THE COUNT OF RECORDS RETURNED */
/*IN SQLERRD3. */
DO I = 1 TO SQLERRD(3);
IF EMPNO(I) = '000170' THEN
DO;
:

```

```

END;
END;
  IF SQLERRD(5) = 100 THEN
DO;
      /* PROCESS END OF FILE */
END;
FINISHED:

```

ในตัวอย่างนี้, มีการกำหนดเคอร์เซอร์สำหรับตาราง CORPDATA.EMPLOYEE เพื่อเลือกแถวทั้งหมดที่คอลัมน์ WORKDEPT มีค่าเท่ากับ 'D11'. ตาราง EMPLOYEE ตัวอย่างใน Sample Tables แสดงตารางผลลัพธ์ที่มีหลายแถว. ข้อความ DECLARE CURSOR และ OPEN ไม่มีซินแทกซ์พิเศษเมื่อถูกใช้ร่วมกับข้อความ FETCH แบบหลายแถว. คำสั่ง FETCH อีกคำสั่งหนึ่งซึ่งส่งคืนแถวเดี่ยวมาให้กับเคอร์เซอร์เดียวกันสามารถโค้ดได้ที่จุดอื่นในโปรแกรม. คำสั่ง FETCH แบบหลายแถวถูกนำมาใช้เพื่อเรียกแถวทั้งหมดในตารางผลลัพธ์ออกมา. หลัง FETCH, ตำแหน่งเคอร์เซอร์ยังคงอยู่บนแถวสุดท้ายในบล็อก.

พื้นที่แถว, ROWAREA, ถูกกำหนดให้เป็นอะเรย์อักขระ. ข้อมูลจากตารางผลลัพธ์ถูกใส่ไว้ในตัวแปรโฮสต์. ในตัวอย่างนี้, ตัวแปรตัวชี้ (pointer) จะถูกกำหนดให้กับแอดเดรสของ ROWAREA. แต่ละไอเท็มในแถวซึ่งถูกส่งคืนจะถูกตรวจสอบและถูกใช้ด้วยโครงสร้าง DEPT พื้นฐาน.

แอ็ททริบิวต์ (ประเภทและความยาว) ของไอเท็มใน descriptor ตรงกับคอลัมน์ที่ถูกเรียกออกมา. ในกรณีนี้, ไม่มีการให้พื้นที่ตัวบ่งชี้ใดๆ.

หลังทำคำสั่ง FETCH เสร็จสิ้นแล้ว, ROWAREA จะประกอบด้วยแถวทั้งหมดที่มีค่าเท่ากับ 'D11', ในกรณีนี้จะมี 11 แถว. SQLCA ที่ถูกส่งคืนมายังแ็พพลิเคชั่นประกอบด้วยข้อมูลต่อไปนี้:

- SQLCODE ประกอบด้วย 0
- SQLSTATE ประกอบด้วย '00000'
- SQLERRD3 ประกอบด้วย 11, ซึ่งเป็นจำนวนแถวที่ถูกส่งคืน
- SQLERRD4 ประกอบด้วย 34, สำหรับความยาวของแถวที่ถูกดึงข้อมูลออกมา
- SQLERRD5 ประกอบด้วย +100, แสดงว่าแถวสุดท้ายในตารางผลลัพธ์ถูกดึงข้อมูลออกมา

ในตัวอย่างนี้, แ็พพลิเคชั่นได้ประโยชน์จากข้อเท็จจริงที่ว่า SQLERRD5 มีการบ่งชี้ว่าสิ้นสุดไฟล์แล้ว. ผลที่ได้คือ, แ็พพลิเคชั่นไม่ต้องเรียก SQL อีกครั้งเพื่อพยายามเรียกแถวออกมามากขึ้น. หากเคอร์เซอร์มีความสามารถในการรับรู้ทันทีต่อการแทรก, คุณควรเรียก SQL ในกรณีที่มีการใส่เพิ่มเร็กคอร์ดใดๆ. เคอร์เซอร์มีความสามารถในการรับรู้ทันทีเมื่อระดับ commitment control เป็นระดับอื่นที่ไม่ใช่ *RR.

สิ่งอ้างอิงที่เกี่ยวข้อง

“DB2 UDB for iSeries ตารางตัวอย่าง” ในหน้า 335

หัวข้อนี้ประกอบด้วยตารางตัวอย่างที่ใช้อ้างอิง และใช้ในหัวข้อนี้ รวมทั้งหัวข้อ SQL Reference.

ข้อมูลที่เกี่ยวข้อง

ภาคผนวก D. SQLDA (SQL Descriptor Area)

ยูนิตงานและเคอร์เซอร์ที่เปิดอยู่

เมื่อโปรแกรมของคุณทำยูนิตงานเสร็จสมบูรณ์แล้ว, โปรแกรมควร commit หรือ roll back การเปลี่ยนแปลงที่คุณได้ทำไว้.

เว้นแต่คุณจะระบุ HOLD บนคำสั่ง COMMIT หรือ ROLLBACK, เคอร์เซอร์ทั้งหมดที่เปิดอยู่จะถูกปิดอัตโนมัติโดย SQL. เคอร์เซอร์ที่ถูกประกาศด้วย WITH HOLD clause จะไม่ถูกปิดโดยอัตโนมัติบน COMMIT. แต่จะถูกปิดโดยอัตโนมัติบน ROLLBACK (WITH HOLD clause ซึ่งถูกระบุบนข้อความ DECLARE CURSOR จะถูกข้ามไป).

หากคุณต้องการประมวลผลต่อจากตำแหน่งเคอร์เซอร์ปัจจุบันหลัง COMMIT หรือ ROLLBACK, คุณต้องระบุ COMMIT HOLD หรือ ROLLBACK HOLD. เมื่อมีการระบุ HOLD, เคอร์เซอร์ใดๆ ที่เปิดอยู่จะถูกเปิดค้างไว้และรักษาตำแหน่งเคอร์เซอร์ไว้เพื่อที่จะเริ่มต้นการประมวลผลใหม่. บนข้อความ COMMIT, ตำแหน่งเคอร์เซอร์จะถูกคงไว้. บนคำสั่ง ROLLBACK, ตำแหน่งเคอร์เซอร์จะถูกเรียกคืนไปไว้ที่ข้างหลังของแถวสุดท้ายที่ถูกเรียกออกมาจากยูนิตงานก่อนหน้า. ล็อกของเร็กคอร์ดทั้งหมดยังคงถูกรีลีส.

หลังจากใส่คำสั่ง COMMIT หรือ ROLLBACK โดยไม่มี HOLD, ล็อกทั้งหมดจะถูกรีลีสและเคอร์เซอร์ทั้งหมดจะถูกปิด. คุณสามารถเปิดเคอร์เซอร์ได้อีกครั้ง, แต่คุณจะต้องเริ่มการประมวลผลที่แถวแรกของตารางผลลัพธ์.

หมายเหตุ: ค่ากำหนดของพารามิเตอร์ ALWBLK(*ALLREAD) ของคำสั่ง CRTSQLxxx สามารถเปลี่ยนกลับสู่สภาพเดิมของตำแหน่งเคอร์เซอร์สำหรับเคอร์เซอร์แบบอ่านอย่างเดียว. โปรดดู “แอ็พพลิเคชัน Dynamic SQL” สำหรับข้อมูลเกี่ยวกับการใช้พารามิเตอร์ ALWBLK และระดับการทำงานอื่นๆ ที่เกี่ยวข้องกับอ็อปชันบนคำสั่ง CRTSQLxxx.

หลักการที่เกี่ยวข้อง

“แอ็พพลิเคชัน Dynamic SQL”

Dynamic SQL อนุญาตให้แอ็พพลิเคชันกำหนด และ รัน SQL statement ที่เวลารันไทม์ของโปรแกรม. แอ็พพลิเคชัน ที่มีไว้เพื่อ dynamic SQL จะถือว่าเป็น อินพุต (หรือ build) ของ SQL statement ในรูปของสตริงอักขระ. แอ็พพลิเคชันไม่จำเป็นต้องทราบว่า คำสั่ง SQL ประเภทใดที่จะรัน.

ข้อมูลที่เกี่ยวข้อง

Commitment control

แอ็พพลิเคชัน Dynamic SQL

Dynamic SQL อนุญาตให้แอ็พพลิเคชันกำหนด และ รัน SQL statement ที่เวลารันไทม์ของโปรแกรม. แอ็พพลิเคชัน ที่มีไว้เพื่อ dynamic SQL จะถือว่าเป็น อินพุต (หรือ build) ของ SQL statement ในรูปของสตริงอักขระ. แอ็พพลิเคชันไม่จำเป็นต้องทราบว่า คำสั่ง SQL ประเภทใดที่จะรัน.

แอ็พพลิเคชัน:

- สร้าง หรือ รับเป็นอินพุตของคำสั่ง SQL
- เตรียมคำสั่ง SQL สำหรับการรัน
- ทำการรันคำสั่ง
- จัดการกับคำสั่งคืนของ SQL

SQL แบบโต้ตอบ คือตัวอย่างของโปรแกรม SQL แบบไดนามิก. คำสั่ง SQL จะถูกประมวลผลและทำงานไปอย่างต่อเนื่องโดย SQL แบบโต้ตอบ.

หมายเหตุ:

1. การประมวลผล SQL แบบ dynamic จะมีค่าใช้จ่ายสูงกว่า SQL แบบ static เนื่องจากคำสั่งอาจต้องใช้การประมวลผลแบบเต็ม ณ เวลารันไทม์. ในกรณีที่เลวร้ายที่สุด, คำสั่งต้องถูกเตรียม, ถูกผูก, และถูก optimize แบบเต็มทีโดยฐานข้อมูลก่อนที่จะรัน. ในกรณีอื่นๆ, ถ้ารันคำสั่งก่อน, ส่วนของการประมวลผลจะสามารถข้ามได้ เนื่องจาก algorithm ที่ถูกใช้ และแคชที่เก็บไว้โดยฐานข้อมูล. คุณลักษณะพิเศษเหล่านี้อนุญาตให้ DB2 สำหรับ iSeries เตรียมประสิทธิภาพในการทำงานที่ดีสำหรับคำสั่ง SQL แบบ dynamic. ถ้าประสิทธิภาพในการทำงานของแอ็พพลิเคชันแบบ dynamic ของคุณอยู่ในขั้นเลวร้าย, ให้พิจารณาถึงการเพิ่มความสามารถในการทำงานที่ถูกขยายแบบ dynamic โดยใช้ QSQRCEED API. คุณลักษณะพิเศษนี้อนุญาตให้แอ็พพลิเคชันคงไว้ซึ่งแคชของคำสั่ง SQL และลดค่าใช้จ่ายในช่วงรันไทม์ เมื่อรันแอ็พพลิเคชัน.
2. โปรแกรมที่มี EXECUTE หรือ EXECUTE IMMEDIATE statement อยู่และใช้ FOR READ ONLY clause ในการทำให้เคอร์เซอร์สำหรับอ่านอย่างเดียวมีการทำงานที่ดีขึ้น เนื่องจากมีการใช้การจับเป็นกลุ่มบล็อกเพื่อทำการเรียกแถวข้อมูลออกมาสำหรับเคอร์เซอร์.
ตัวเลือก ALWBLK(*ALLREAD) CRTSQLxxx จะแสดงความหมายโดยนัยของการประกาศ FOR READ ONLY สำหรับเคอร์เซอร์ทั้งหมดที่ไม่ได้แสดงโค้ดไว้อย่างชัดเจนว่า FOR UPDATE OF หรือมีการระบุการลบออกหรือการอัปเดตที่อ้างถึงในเคอร์เซอร์. เคอร์เซอร์ที่มีการประกาศโดยนัยว่า FOR READ ONLY จะได้รับผลประโยชน์จากรายการที่สองในรายชื่อนี้.

ในบาง dynamic SQL statement จำเป็นต้องใช้ตัวแปรแอดเดรส. โปรแกรม RPG/400 ต้องการความช่วยเหลือของ PL/I, COBOL, C, หรือโปรแกรม ILE RPG เพื่อจัดการกับตัวแปรแอดเดรส.

หลักการที่เกี่ยวข้อง

“การใช้ SQL แบบโต้ตอบ” ในหน้า 291

SQL แบบโต้ตอบอนุญาตให้โปรแกรมเมอร์ หรือผู้ดูแลฐานข้อมูลมีความรวดเร็ว และสะดวกในการกำหนด, อัปเดต, ลบ, หรือสำรวจข้อมูลเพื่อทำการทดสอบ, วิเคราะห์ปัญหา, และการดูแลฐานข้อมูล.

สิ่งอ้างอิงที่เกี่ยวข้อง

“ยูนิตงานและเคอร์เซอร์ที่เปิดอยู่” ในหน้า 270

เมื่อโปรแกรมของคุณทำยูนิตงานเสร็จสมบูรณ์แล้ว, โปรแกรมควร commit หรือ roll back การเปลี่ยนแปลงที่คุณได้ทำไว้.

“การใช้คำสั่ง PREPARE และ EXECUTE” ในหน้า 274

ถ้าคำสั่งที่ไม่ใช่ SELECT ไม่มีเครื่องหมายพารามิเตอร์อยู่เลย, จะสามารถถูกรันได้อย่างต่อเนื่องโดยใช้คำสั่ง EXECUTE IMMEDIATE. อย่างไรก็ตาม, ถ้าคำสั่งที่ไม่ใช่ SELECT มีตัวทำเครื่องหมายพารามิเตอร์, คำสั่งเหล่านี้จะต้องถูกรันโดยใช้ PREPARE และ EXECUTE.

“การประมวลผลคำสั่ง non-SELECT” ในหน้า 273

เมื่อต้องการสร้างคำสั่ง SQL non-SELECT แบบ dynamic, คุณจำเป็นต้องตรวจสอบว่า คำสั่ง SQL ที่คุณต้องการสร้างคือคำสั่งที่สามารถรันแบบ dynamic ได้ และสร้างคำสั่ง SQL ได้.

Process Extended Dynamic SQL (QSQRCEED) API

ข้อมูลที่เกี่ยวข้อง

Action ที่อนุญาตให้ใช้บนคำสั่ง SQL

การออกแบบและการรันแอ็พพลิเคชัน SQL

เพื่อที่จะกำหนด dynamic SQL statement, คุณจะต้องใช้ข้อความที่มี EXECUTE statement หรือ EXECUTE IMMEDIATE statement อย่างใดอย่างหนึ่ง, เนื่องจากไม่ได้จัดเตรียม dynamic SQL statement ที่เวลาคอมไพล์ล่วงหน้า และดังนั้นจึงต้องจัดเตรียมที่เวลารันไทม์. EXECUTE IMMEDIATE statement จัดเตรียม SQL statement และรันอย่างรวดเร็วในเวลารันไทม์ของโปรแกรม.

SQL statement สามารถแบ่งเป็นแบบพื้นฐานได้ 2 แบบด้วยกัน: SELECT statement และ non-SELECT statement. Non-SELECT statement จะประกอบด้วย statement อื่นได้แก่ DELETE, INSERT, และ UPDATE.

เซิร์ฟเวอร์แอ็พพลิเคชันโคลเอนต์ที่ใช้อินเตอร์เฟส เช่น ODBC โดยทั่วไปจะใช้ dynamic SQL ในการเข้าถึงฐานข้อมูล.

ข้อมูลที่เกี่ยวข้อง

โปรแกรมมิ่งสำหรับ iSeries Access Express

CCSID ของคำสั่ง SQL แบบ dynamic

โดยปกติแล้วคำสั่ง SQL จะเป็นตัวแปรโฮสต์. CCSID ของตัวแปรโฮสต์จะถูกใช้ในรูปของ CCSID ของ text statement. ใน PL/I, สามารถเป็นอักขรนิพจน์ได้ด้วย. ในกรณีนี้, ค่า CCSID ของงานจะถูกใช้ในรูปของ CCSID ของข้อความ.

Dynamic SQL statement จะถูกประมวลผลโดยใช้ CCSID ของ statement text. ซึ่งจะส่งผลต่อ variant character. ตัวอย่างเช่น, เครื่องหมาย not (-) จะถูกกำหนดให้อยู่ที่ 'BA'X ใน CCSID 500. ซึ่งหมายความว่า ถ้า CCSID ของข้อความคำสั่งของคุณคือ 500, SQL จะแสดงเครื่องหมาย not (-) ในค่า 'BA'X.

ถ้า CCSID ของข้อความคำสั่งเป็น 65535, SQL จะทำการประมวลผล variant character เหมือนกับว่ามีค่า CCSID เท่ากับ 37. ซึ่งหมายความว่า SQL จะทำการค้นหาเครื่องหมาย not (-) ที่ '5F'X.

การประมวลผลคำสั่ง non-SELECT

เมื่อต้องการสร้างคำสั่ง SQL non-SELECT แบบ dynamic, คุณจำเป็นต้องตรวจสอบว่า คำสั่ง SQL ที่คุณต้องการสร้าง คือคำสั่งที่สามารถรันแบบ dynamic ได้ และสร้างคำสั่ง SQL ได้.

การรัน SQL แบบไดนามิก คำสั่ง non-SELECT:

1. รันคำสั่ง SQL โดยใช้ EXECUTE IMMEDIATE, หรือทำการ PREPARE ให้กับคำสั่ง SQL, แล้วทำการ EXECUTE คำสั่งที่ถูกจัดเตรียมนั้น.
2. การจัดการกับคำสั่งคืนของ SQL ที่อาจเกิดขึ้น.

ตัวอย่างต่อไปนี้ เป็นตัวอย่างของแอ็พพลิเคชันที่รัน SQL แบบ dynamic คำสั่ง non-SELECT (stmtstrg):

```
EXEC SQL  
EXECUTE IMMEDIATE :stmtstrg;
```

หลักการที่เกี่ยวข้อง

“แอ็พพลิเคชัน Dynamic SQL” ในหน้า 271

Dynamic SQL อนุญาตให้แอ็พพลิเคชันกำหนด และ รัน SQL statement ที่เวลารันไทม์ของโปรแกรม. แอ็พพลิเคชัน ที่มีไว้เพื่อ dynamic SQL จะถือว่าเป็น อินพุต (หรือ build) ของ SQL statement ในรูปของสตริงอักขระ. แอ็พพลิเคชันไม่จำเป็นต้องทราบ ว่า คำสั่ง SQL ประเภทใดที่จะรัน.

“การใช้ SQL แบบโต้ตอบ” ในหน้า 291

SQL แบบโต้ตอบอนุญาตให้โปรแกรมเมอร์ หรือผู้ดูแลฐานข้อมูลมีความรวดเร็ว และสะดวกในการกำหนด, อัปเดต, ลบ, หรือสำรวจข้อมูลเพื่อทำการทดสอบ, วิเคราะห์ปัญหา, และการดูแลฐานข้อมูล.

การใช้คำสั่ง PREPARE และ EXECUTE:

ถ้าคำสั่งที่ไม่ใช่ SELECT ไม่มีเครื่องหมายพารามิเตอร์อยู่เลย, จะสามารถถูกรันได้อย่างต่อเนื่องโดยใช้คำสั่ง EXECUTE IMMEDIATE. อย่างไรก็ตาม, ถ้าคำสั่งที่ไม่ใช่ SELECT มีตัวทำเครื่องหมายพารามิเตอร์, คำสั่งเหล่านี้จะต้องถูกรันโดยใช้ PREPARE และ EXECUTE.

คำสั่ง PREPARE จะจัดเตรียมคำสั่งที่ไม่ใช่ SELECT (ตัวอย่างเช่น, คำสั่ง DELETE) และตั้งชื่อได้ตามความต้องการ. ถ้า DLYPRP (*YES) ถูกระบุไว้ในคำสั่ง CRTSQLxxx, การจัดเตรียมจะถูกหน่วงไว้จนกระทั่งมีการใช้คำสั่งเป็นครั้งแรกในคำสั่ง EXECUTE หรือ DESCRIBE, นอกจากนี้จะมีการระบุ USING clause ไว้ในคำสั่ง PREPARE. หลังจากที่จัดเตรียมข้อความแล้ว ได้ถูกจัดทำขึ้น, มันจะถูกรันได้หลายครั้งภายในโปรแกรมเดียวกัน, โดยใช้ค่าต่างกันสำหรับเครื่องหมายพารามิเตอร์. ตัวอย่างต่อไปนี้เป็นคำสั่งที่ถูกจัดเตรียมไว้เพื่อรันได้หลายๆ ครั้ง:

```
DSTRING = 'DELETE FROM CORPDATA.EMPLOYEE WHERE EMPNO = ?';
```

```
/*The ? is a parameter marker which denotes
```

```
ค่านี้เป็นตัวแปรโฮสต์ที่จะ
```

```
ถูกแทนค่าทุกครั้ง คำสั่งถูกรัน.*/
```

```
EXEC SQL PREPARE S1 FROM :DSTRING;
```

```
/*DSTRING เป็น delete statement ที่ PREPARE statement มีชื่อเป็น  
S1.*/
```

```
DO UNTIL (EMP =0);
```

```
/*แอฟพลิเคชันโปรแกรมอ่านค่าสำหรับ EMP จาก จอภาพ.*/
```

```
EXEC SQL
```

```
EXECUTE S1 USING :EMP;
```

```
END;
```

ลักษณะโดยทั่วไปที่คล้ายกันกับตัวอย่างที่กล่าวมาข้างต้น, คุณจะต้องทราบจำนวนของเครื่องหมายพารามิเตอร์ และชนิดข้อมูลของแต่ละตัว, เนื่องจากตัวแปรโฮสต์ที่ทำการจัดหาข้อมูลอินพุตนั้นจะถูกประกาศในช่วงที่โปรแกรมถูกเขียนขึ้น.

หมายเหตุ: คำสั่งที่ถูกจัดเตรียมทั้งหมดที่เชื่อมโยงกับแอฟพลิเคชันเซิร์ฟเวอร์จะถูกทำลายลงเมื่อการเชื่อมต่อกับแอฟพลิเคชันเซิร์ฟเวอร์สิ้นสุดลง. การเชื่อมต่อจะสิ้นสุดลงโดยการใช้คำสั่ง CONNECT (Type 1), คำสั่ง DISCONNECT, หรือ RELEASE ตามด้วย COMMIT.

หลักการที่เกี่ยวข้อง

“แอฟพลิเคชัน Dynamic SQL” ในหน้า 271

Dynamic SQL อนุญาตให้แอฟพลิเคชันกำหนด และ รัน SQL statement ที่เวลารันไทม์ของโปรแกรม. แอฟพลิเคชันที่มีไว้เพื่อ dynamic SQL จะถือว่าเป็น อินพุต (หรือ build) ของ SQL statement ในรูปของสตริงอักขระ. แอฟพลิเคชันไม่จำเป็นต้องทราบว่า คำสั่ง SQL ประเภทใดที่จะรัน.

การประมวลผลคำสั่ง SELECT และการใช้ descriptor

มีประเภทของคำสั่ง SELECT พื้นฐานอยู่สองประเภท คือ: **fixed list** และ **varying list**.

เมื่อต้องการประมวลผลคำสั่ง SELECT แบบ fixed-list, จึงไม่จำเป็นต้องใช้ SQL descriptor.

- | เมื่อต้องการประมวลผลคำสั่ง SELECT แบบ varying-list, คุณต้องประกาศโครงสร้าง SQL descriptor area (SQLDA) เป็น
- | อันดับแรก หรือ ALLOCATE SQLDA. ฟอรัมของ SQL descriptor ทั้งสองแบบสามารถใช้ส่งผ่านค่าอินพุตตัวแปรโฮสต์
- | จากแอฟพลิเคชันโปรแกรมไปยัง SQL และรับค่าเอาต์พุตจาก SQL. นอกจากนี้, ข้อมูลที่เกี่ยวกับนิพจน์ของรายการ SELECT
- | สามารถส่งคืนกลับมาในคำสั่ง PREPARE หรือ DESCRIBE.

คำสั่ง SELECT แบบรายการคงที่:

ใน dynamic SQL, fixed-list SELECT statement คือคำสั่งที่ถูกออกแบบเพื่อเรียกข้อมูลที่ทราบค่าและชนิดของข้อมูล. เมื่อใช้คำสั่งเหล่านี้, คุณสามารถคาดเดา และกำหนดตัวแปรโฮสต์ที่เหมาะสมกับข้อมูลที่ดึงออกมา, ดังนั้น SQL descriptor area (SQLDA) จึงไม่มีความจำเป็น.

ค่า FETCH ที่สำเร็จสมบูรณ์จะส่งคืนค่าตัวเลขที่เป็นค่าสุดท้ายกลับมาในแต่ละครั้ง, และค่าเหล่านี้จะมีรูปแบบเดียวกันกับค่าที่ส่งคืนมาสำหรับการ FETCH ในครั้งสุดท้าย. คุณสามารถระบุตัวแปรโฮสต์ได้เช่นเดียวกับกับแอฟพลิเคชัน SQL.

คุณสามารถใช้ fixed-list dynamic SELECT statement กับแอฟพลิเคชันโปรแกรมใดๆ ที่สนับสนุนการใช้งาน SQL.

การรัน fixed-list SELECT statement อย่างต่อเนื่อง, แอฟพลิเคชันของคุณจะต้อง:

1. ใส่อินพุตคำสั่ง SQL ลงในตัวแปรโฮสต์.
2. ออกคำสั่ง PREPARE เพื่อตรวจสอบความถูกต้องของคำสั่ง SQL แบบ dynamic และใส่ลงใน form ที่สามารถถูกรันได้. ถ้า DLYPRP (*YES) ถูกระบุไว้ในคำสั่ง CRTSQLxxx, การจัดเตรียมจะถูกหน่วงไว้จนกระทั่ง statement ถูกใช้เป็นครั้งแรกในคำสั่ง EXECUTE หรือ DESCRIBE, นอกเสียจากได้ระบุ USING clause ไว้ในคำสั่ง PREPARE.
3. ประกาศเคอร์เซอร์สำหรับชื่อของคำสั่ง.
4. เปิดเคอร์เซอร์.
5. FETCH แลวเข้าไปใส่ไว้ใน fixed list ของตัวแปร (แทนที่จะไว้ใน descriptor area, เช่นเดียวกับที่คุณใช้คำสั่ง varying-list SELECT).
6. เมื่อมีการสิ้นสุดของข้อมูลเกิดขึ้น, ปิดเคอร์เซอร์.
7. จัดการกับคำสั่งคืนของ SQL ใดๆที่เกิดขึ้น.

ตัวอย่างเช่น:

```
MOVE 'SELECT EMPNO, LASTNAME FROM CORPDATA.EMPLOYEE WHERE EMPNO>?'  
  TO DSTRING.  
EXEC SQL  
PREPARE S2 FROM :DSTRING END-EXEC.  
  
EXEC SQL  
DECLARE C2 CURSOR FOR S2 END-EXEC.  
  
EXEC SQL  
OPEN C2 USING :EMP END-EXEC.
```

```
PERFORM FETCH-ROW UNTIL SQLCODE NOT=0.
```

```
EXEC SQL  
CLOSE C2 END-EXEC.  
STOP-RUN.  
FETCH-ROW.  
EXEC SQL  
FETCH C2 INTO :EMP, :EMPNAME END-EXEC.
```

| **หมายเหตุ:** จำไว้ว่า เนื่องจากคำสั่ง SELECT, ในกรณีนี้, จะส่งคืนตัวเลข และชนิดของรายการข้อมูลเช่นเดียวกับที่รันคำสั่ง
| fixed-list SELECT, คุณไม่จำเป็นต้องใช้ SQL descriptor area.

คำสั่ง SELECT แบบ varying-list:

ใน dynamic SQL, คำสั่ง SELECT แบบ varying-list เป็นรูปแบบหนึ่งสำหรับจำนวนและรูปแบบของคอลัมน์ผลลัพธ์ที่จะถูกส่งคืนมาที่ไม่สามารถคาดเดาได้; นั่นคือ, คุณไม่สามารถทราบจำนวนตัวแปรที่คุณต้องการ, หรือชนิดของข้อมูล.

ดังนั้น, คุณจึงไม่สามารถกำหนดตัวแปรโฮสต์ล่วงหน้าได้ เพื่อที่จะให้เหมาะสมกับคอลัมน์ผลลัพธ์ที่จะถูกส่งคืนกลับมา.

| **หมายเหตุ:** ใน REXX, ขั้นตอนที่ 5.b, 6, และ 7 ไม่สามารถใช้ด้วยกันได้. REXX สนับสนุนเฉพาะ SQL descriptor ที่ถูก
| กำหนดโดยใช้โครงสร้าง SQLDA; และไม่สนับสนุน SQL descriptor ที่จัดสรรแล้ว.

ถ้าแอ็พพลิเคชันของคุณยอมรับคำสั่ง SELECT แบบ varying-list, โปรแกรมของคุณจะต้อง:

1. ใส่อินพุตคำสั่ง SQL ลงในตัวแปรโฮสต์.
2. ออกคำสั่ง PREPARE เพื่อตรวจสอบความถูกต้องของคำสั่ง SQL แบบ dynamic และใส่ลงไปใน form ที่สามารถถูกรันได้. ถ้า DLYPRP (*YES) ถูกระบุไว้ในคำสั่ง CRTSQLxxx, การจัดเตรียมจะถูกหน่วงไว้จนกระทั่งคำสั่งถูกใช้เป็นครั้งแรกในคำสั่ง EXECUTE หรือ DESCRIBE, นอกเสียจากได้ระบุ USING clause ไว้ในคำสั่ง PREPARE.
3. ประกาศเคอร์เซอร์สำหรับชื่อของข้อความ.
4. เปิดเคอร์เซอร์ (ที่ประกาศในขั้นตอนที่ 3) ที่มีชื่อของ dynamic SELECT statement อยู่.
5. สำหรับ SQL descriptor ที่จัดสรรแล้ว, ให้รันคำสั่ง ALLOCATE DESCRIPTOR เพื่อกำหนด descriptor ที่คุณต้องการใช้.
6. ออกคำสั่ง DESCRIBE เพื่อร้องขอข้อมูลจาก SQL เกี่ยวกับชนิดและขนาดของแต่ละคอลัมน์ในตาราง.

หมายเหตุ:

- a. คุณสามารถตัดคำสั่ง PREPARE โดยใช้ INTO clause เพื่อดำเนินการฟังก์ชันของ PREPARE และ DESCRIBE โดยใช้คำสั่งเดียว.
- b. ถ้า SQLDA และ SQLDA ไม่ใหญ่พอที่จะเก็บคอรายละเอียดคอลัมน์สำหรับคอลัมน์ที่ถูกเรียกออกมา, โปรแกรมต้องคำนวณว่า ต้องการเนื้อที่ว่างเท่าไร, และเตรียมที่เก็บให้มีขนาดเท่ากับที่ต้องการ, สร้าง SQLDA ตัวใหม่, และใช้คำสั่ง DESCRIBE.

| ถ้าการใช้ SQL descriptor ที่จัดสรรแล้วซึ่งไม่ใหญ่พอ, ให้ทำการจัดสรร descriptor ใหม่อีกครั้ง, จัด
| สรรด้วยจำนวน entry ขนาดใหญ่กว่า, และใช้คำสั่ง DESCRIBE ใหม่อีกครั้ง.

7. สำหรับ SQLDA descriptor, ให้จัดสรรจำนวนของหน่วยเก็บที่จำเป็น เพื่อเก็บแถวของข้อมูลที่ดึงออกมา.
8. สำหรับ SQLDA descriptor, ให้ใส่แอดเดรสหน่วยเก็บใน SQLDA เพื่อที่จะบอก SQL ให้ทราบว่าเก็บแต่ละไอเท็มของข้อมูลที่ดึงออกมาไว้ที่ไหน.

- I 9. FETCH แถวข้อมูล.
- 10. ประมวลผลข้อมูลที่ส่งคืนใน SQL descriptor.
- 11. การจัดการกับคำสั่งคืนของ SQL ที่อาจเกิดขึ้น.
- 12. เมื่อมีการสิ้นสุดของข้อมูลเกิดขึ้น, ปิดเคอร์เซอร์.
- I 13. สำหรับ SQL descriptor ที่จัดสรรแล้ว, ให้รันคำสั่ง DEALLOCATE DESCRIPTOR เพื่อลบ descriptor.

สิ่งอ้างอิงที่เกี่ยวข้อง

“ตัวอย่าง: คำสั่ง Select เพื่อจัดสรรหน่วยเก็บสำหรับ SQLDA” ในหน้า 280

สมมติว่า แอ็พพลิเคชันของคุณจำเป็นต้องมีความสามารถในการจัดการกับคำสั่ง SELECT แบบ dynamic, เมื่อค่าหนึ่งเปลี่ยนเป็นอีกค่าหนึ่งสำหรับใช้ต่อไป. คำสั่งนี้สามารถอ่านได้จากจอแสดงผล, ซึ่งถูกส่งผ่านจากแอ็พพลิเคชันอื่น, หรือถูกสร้างขึ้นจากแอ็พพลิเคชันของคุณในขณะปฏิบัติงาน.

SQL descriptor area:

SQL แบบ dynamic ใช้ SQL descriptor area (SQLDA) เพื่อส่งผ่านข้อมูลเกี่ยวกับคำสั่ง SQL ระหว่าง SQL และแอ็พพลิเคชันของคุณ. descriptor เป็นสิ่งจำเป็นสำหรับการรันคำสั่ง DESCRIBE, DESCRIBE INPUT และ DESCRIBE TABLE, และยังสามารถใช้บนคำสั่ง PREPARE, OPEN, FETCH, CALL, และ EXECUTE ได้.

ความหมายของข้อมูลใน SQLDA ขึ้นอยู่กับการใช้งาน. ใน PREPARE และ DESCRIBE, SQLDA จัดเตรียมข้อมูลให้กับแอ็พพลิเคชันโปรแกรมเกี่ยวกับข้อความที่ถูกจัดเตรียม. ใน DESCRIBE INPUT, SQL descriptor area จัดเตรียมข้อมูลเกี่ยวกับตัวทำเครื่องหมายพารามิเตอร์ในคำสั่งที่ได้เตรียมไว้ให้กับแอ็พพลิเคชันโปรแกรม. ใน DESCRIBE TABLE, SQLDA จัดเตรียมข้อมูลให้แอ็พพลิเคชันโปรแกรมเกี่ยวกับคอลัมน์ในตารางหรือภาพที่เห็น. ใน OPEN, EXECUTE, CALL, และ FETCH, SQLDA จัดเตรียมข้อมูลเกี่ยวกับตัวแปรโฮสต์. ตัวอย่างเช่น, คุณสามารถอ่านค่ามาใส่ใน SQLDA โดยใช้คำสั่ง DESCRIBE, เปลี่ยนค่าข้อมูลใน descriptor เพื่อใช้ตัวแปรโฮสต์, แล้วใช้ descriptor ตัวเดิมในคำสั่ง FETCH.

ถ้าแอ็พพลิเคชันของคุณอนุญาตให้คุณมีหลายเคอร์เซอร์ที่เปิดอยู่ในเวลาเดียวกัน, คุณสามารถโค้ดหลายๆ SQLDA ได้, หนึ่งโค้ดสำหรับแต่ละคำสั่ง SELECT แบบ dynamic.

- I มี SQLDA อยู่สองชนิด. ชนิดแรกถูกนิยามด้วยคำสั่ง ALLOCATE DESCRIPTOR. ชนิดที่สองถูกนิยามด้วยโครงสร้าง
- I SQLDA.

- I ALLOCATE DESCRIPTOR ไม่สนับสนุนใน REXX. SQLDA สามารถใช้ใน C, C++, COBOL, PL/I, REXX, และ RPG.
- I เนื่องจาก RPG/400 ไม่ได้จัดเตรียมวิธีการตั้งค่าตัวชี้, SQLDA จะต้องถูกตั้งค่าอยู่นอกโปรแกรม RPG/400 โดย
- I โปรแกรม PL/I, C, C++, COBOL, หรือ ILE RPG. โปรแกรมนี้ต้องเรียกโปรแกรม RPG/400.

ข้อมูลที่เกี่ยวข้อง

SQLCA (SQL communications area)

SQLDA (SQL descriptor area)

รูปแบบของ SQLDA:

SQLDA ประกอบด้วยตัวแปรสี่ตัวตามด้วยเลขเฉพาะของการเกิดขึ้นตามลำดับของกลุ่มตัวแปรหกตัวที่ชื่อ SQLVAR.

หมายเหตุ: SQLDA ใน REXX จะแตกต่างออกไป.

เมื่อ SQLDA ถูกใช้ใน OPEN, FETCH, CALL, และ EXECUTE, แต่ละครั้งของการเกิดขึ้นของ SQLVAR จะช่วยอธิบายตัวแปรโฮสต์.

ฟิลด์ของ SQLDA มีดังนี้:

SQLDAID

SQLDAID จะเป็นเช่นเดียวกับการใช้ "eyecatcher" สำหรับดัมพ์หน่วยเก็บ. เป็นชุดอักขระ 8 ตัวที่มีค่า 'SQLDA' หลังจาก SQLDA ถูกเรียกใช้ใน PREPARE หรือ DESCRIBE statement. ตัวแปรนี้ไม่ได้ใช้สำหรับ FETCH, OPEN, CALL, หรือ EXECUTE.

ไบต์ที่ 7 สามารถใช้ในการพิจารณาว่าในแต่ละคอลัมน์มีความจำเป็นต้องใช้ SQLVAR entry มากกว่าหนึ่งหรือไม่. SQLVAR entry หลายๆตัวอาจเป็นที่ต้องการหากมี LOB หรือชนิดที่ต่างกันของคอลัมน์เกิดขึ้น. แพล็กลักษณะนี้จะถูกตั้งให้ว่างเอาไว้ถ้าไม่มี LOB หรือความต่างชนิดเกิดขึ้น.

SQLDAID ไม่สามารถใช้ใน REXX ได้.

SQLDABC

SQLDABC ระบุความยาวของ SQLDA. มันจะเป็นจำนวนเต็มแบบ 4 ไบต์ที่มีค่า $SQLN * LENGTH(SQLVAR) + 16$ หลังจาก SQLDA ถูกเรียกใช้ใน PREPARE หรือ DESCRIBE statement. SQLDABC จะต้องมีค่าเท่ากับหรือมากกว่า $SQLN * LENGTH(SQLVAR) + 16$ ก่อนการเรียกใช้โดย FETCH, OPEN, CALL, หรือ EXECUTE.

SQLABC ไม่สามารถใช้ได้ใน REXX.

SQLN SQLN เป็นจำนวนเต็มแบบ 2 ไบต์ที่ระบุจำนวนที่เกิดขึ้นทั้งหมดของ SQLVAR. จะต้องมีการตั้งค่าก่อนที่จะถูกเรียกใช้โดย SQL statement ใดๆให้มีค่ามากกว่าหรือเท่ากับศูนย์.

SQLN ไม่สามารถใช้ได้ใน REXX.

SQLD SQLD เป็นจำนวนเต็มแบบ 2 ไบต์ที่ระบุจำนวนการเกิดขึ้นของ SQLVAR, เรียกได้อีกอย่างว่า, จำนวนของตัวแปรโฮสต์หรือ คอลัมน์ที่อธิบายโดย SQLDA. ฟิลด์จะถูกตั้งค่าโดย SQL ใน DESCRIBE หรือ PREPARE statement. ใน statement อื่นๆ, ฟิลด์นี้จะถูกตั้งค่าก่อนที่จะใช้ให้มีค่ามากกว่าหรือเท่ากับศูนย์และน้อยกว่าหรือเท่ากับ SQLN.

SQLVAR

ตัวแปรกลุ่มนี้จะถูกทวนซ้ำหนึ่งครั้งสำหรับแต่ละตัวแปรโฮสต์หรือ คอลัมน์. ตัวแปรเหล่านี้จะถูกตั้งค่าโดย SQL ใน DESCRIBE หรือ PREPARE statement. ใน statement อื่นๆ, จะต้องถูกตั้งค่าก่อนการใช้. ตัวแปรเหล่านี้จะถูกกำหนดดังต่อไปนี้:

SQLTYPE

SQLTYPE เป็นจำนวนเต็มแบบ 2 ไบต์ที่ระบุชนิดของข้อมูลของตัวแปรโฮสต์หรือคอลัมน์. โปรดดู SQLTYPE และ SQLLEN สำหรับตารางของค่าที่ถูกต้อง. จำนวนคี่ใน SQLTYPE แสดงให้เห็นว่า ตัวแปรโฮสต์มีตัวแปรชี้ที่เชื่อมโยงกันและจะถูกกำหนด address ให้โดย SQLIND.

SQLLEN

SQLLEN เป็นตัวแปรจำนวนเต็มแบบ 2 ไบต์ที่ระบุความยาวของตัวแปรโฮสต์หรือคอลัมน์.

SQLRES

SQLRES เป็นเนื้อที่ 12 ไบต์ที่สำรองไว้สำหรับจุดประสงค์ในการการจัดตำแหน่งที่มีขอบเขตต่อกัน. ให้สังเกตว่า, ใน i5/OS, ตัวชี้จะต้องอยู่ใน quad-word boundary.

SQLRES ไม่สามารถใช้ได้ใน REXX.

SQLDATA

SQLDATA เป็นตัวแปรซีแบบ 16 ไบต์ที่ระบุ address ของ ตัวแปรโฮสต์เมื่อ มีการใช้ SQLDA ใน OPEN, FETCH, CALL, และ EXECUTE.

เมื่อ SQLDA ถูกใช้ใน PREPARE และ DESCRIBE, พื้นที่นี้จะซ้อนกันด้วยข้อมูลต่อไปนี้:

CCSID ของฟิลด์ตัวอักษรหรือ กราฟฟิลด์ ที่บันทึกอยู่ในไบต์ที่สามและสี่ของ SQLDATA. สำหรับข้อมูล BIT, CCSID จะเป็น 65535. ใน REXX, CCSID จะถูกส่งคืนในรูปของตัวแปร SQLCCSID.

SQLIND

SQLIND เป็นตัวชี้แบบ 16 ไบต์ที่ระบุ address ของจำนวนเต็มจำนวนน้อยๆของตัวแปรโฮสต์ที่ใช้ในเป็นตัวระบุของ null หรือ notnull เมื่อ SQLDA ถูกใช้ใน OPEN, FETCH, CALL, และ EXECUTE. ค่าที่เป็นลบจะระบุ null และค่าที่ไม่เป็นลบก็จะระบุ not null. ตัวชี้นี้จะถูกใช้เมื่อ SQLTYPE มีค่าเป็นจำนวนคี่เท่านั้น .

เมื่อ SQLDA ถูกใช้ใน PREPARE และ DESCRIBE, พื้นที่นั้นจะถูกจองไว้สำหรับการใช้ในครั้งต่อไป.

SQLNAME

SQLNAME เป็นตัวแปรแบบอักขระที่มีค่าความยาวผันแปรได้โดยความยาวสูงสุดคือ 30 ตัวอักษร. เมื่อทำการ PREPARE หรือ DESCRIBE, ตัวแปรนั้นจะมีชื่อของคอลัมน์, เลเบล, หรือ คอลัมน์ของระบบที่เลือกไว้. ใน OPEN, FETCH, EXECUTE, หรือ CALL, ตัวแปรนี้สามารถนำมาใช้ในการส่งผ่านค่า CCSID ของสตริงอักขระ. CCSID จะถูกส่งผ่านสำหรับตัวแปรโฮสต์แบบอักขระ และแบบกราฟิก.

ฟิลด์ SQLNAME ใน SQLVAR array entry ของอินพุต SQLDA สามารถตั้งค่าให้ระบุค่าของ CCSID ได้. โปรดดูค่า CCSID ใน SQLDATA หรือ SQLNAME สำหรับฝั่งข้อมูล CCSID ในฟิลด์นี้.

หมายเหตุ: ต้องจำไว้ว่า ฟิลด์ SQLNAME นั้นมีไว้สำหรับแทนที่ค่าเดิมของ CCSID เท่านั้น. แอปพลิเคชันที่ใช้ค่าเดิมที่มีอยู่แล้วไม่จำเป็นต้องส่งผ่านข้อมูลของ CCSID. ถ้าค่าของ CCSID ไม่ถูกส่งผ่าน, จะใช้ค่าเดิมของ CCSID สำหรับงานนั้น.

ค่าเดิมของตัวแปรโฮสต์แบบกราฟิกเป็นค่าเชื่อมโยงของ CCSID แบบดับเบิลไบต์สำหรับ CCSID ของงานนั้น. ถ้าค่าเชื่อมโยงของ CCSID แบบดับเบิลไบต์ไม่ปรากฏ, จะใช้ค่า 65535 .

SQLVAR2

นี่เป็นโครงสร้างเพิ่มเติมของ SQLVAR ที่ประกอบด้วยฟิลด์ 3 ฟิลด์ด้วยกัน. Extended SQLVAR จำเป็นสำหรับคอลัมน์ทั้งหมดของผลลัพธ์ถ้าผลลัพธ์นั้นมีคอลัมน์ที่ต่างชนิดกันหรือ คอลัมน์ LOB อยู่. สำหรับชนิดที่ต่างกัน, จะมีชื่อเรียกที่ต่างกันด้วย. สำหรับ LOB, จะมีแอตทริบิวต์ความยาวของตัวแปรโฮสต์และตัวชี้ไปที่บีฟเฟอร์ที่มีขนาดแน่นอน. ถ้าตัวบอกตำแหน่งถูกใช้ในการแสดง LOB, entry เหล่านี้จะไม่มีความจำเป็นอีก. จำนวนของการเกิด SQLVAR ที่จำเป็นนั้นขึ้นกับข้อความที่ SQLDA ได้ถูกจัดเตรียมมา และ ชนิดของข้อมูลของคอลัมน์หรือพารามิเตอร์ที่กำลังอธิบาย. ไบต์ที่ 7 ของ SQLDAID จะถูกตั้งให้เป็นจำนวนชุดของ SQLVAR ที่จำเป็นเสมอ.

ถ้า SQLD ไม่ได้ถูกตั้งค่าให้เพียงพอกับจำนวน SQLVAR ที่เกิดขึ้น:

- SQLD จะถูกตั้งค่าเป็นจำนวนรวมทั้งหมดของการเกิด SQLVAR ที่จำเป็นสำหรับทุกชุด.
- สัญลักษณ์เตือน +237 จะถูกส่งคืนมาในฟิลด์ SQLCODE ของ SQLCA ถ้าอย่างน้อยถูกระบุไว้อย่างพอเพียงสำหรับ Base SQLVAR Entry. Base SQLVAR entry จะถูกส่งคืนมา แต่ไม่มี Extended SQLVAR ส่งคืนมา.
- สัญลักษณ์เตือน +239 จะถูกส่งคืนมาในฟิลด์ของ SQLCODE ของ SQLCA ถ้าไม่ได้ระบุ SQLVAR อย่างเพียงพอถึงแม้จะเป็น สำหรับ Base SQLVAR Entry. ไม่มี SQLVAR entry ถูกส่งกลับมา.

SQLLONGLEN

SQLLONGLEN เป็นตัวแปรแบบจำนวนเต็ม 4 ไบต์ที่ระบุความยาวของ LOB (BLOB, CLOB, หรือ DBCLOB) ตัวแปรโฮสต์หรือ คอลัมน์.

SQLDATALEN

SQLDATALEN เป็นตัวแปรแบบ 16 ไบต์ที่ระบุ address ของความยาวของตัวแปรโฮสต์. ตัวแปรนี้จะใช้สำหรับ LOB (BLOB, CLOB, และ DBCLOB) ตัวแปรโฮสต์เท่านั้น. ไม่ได้ใช้เพื่อ DESCRIBE หรือ PREPARE.

ถ้าฟิลด์นี้เป็น NULL, แล้วความยาวที่แน่นอนของข้อมูลจะถูกบันทึกใน 4 ไบต์ทันที ก่อนที่จะเป็นส่วนเริ่มของข้อมูล, และ SQLDATA จะชี้ไปยังไบต์แรกของความยาวของฟิลด์นั้น. ความยาวระบุจำนวนของไบต์สำหรับ BLOB หรือ CLOB, และจำนวนตัวอักษรสำหรับ DBCLOB.

ถ้าฟิลด์นี้ไม่ได้มีค่าเป็น NULL, จะมีการเก็บค่าของตัวชี้ใน long buffer แบบ 4 ไบต์ที่มีความยาวที่แน่นอนในหน่วยของไบต์(แม้จะเป็นสำหรับ DBCLOB) ของข้อมูลในบัพเฟอร์ที่ถูกชี้โดย ฟิลด์ SQLDATA ใน matching base SQLVAR.

SQLDATATYPE_NAME

SQLDATATYPE_NAME เป็นตัวแปรที่เป็นอักขระแบบความยาวผันแปรได้ด้วยความยาวสูงสุดเท่ากับ 30. ใช้สำหรับ DESCRIBE หรือ or PREPARE. ตัวแปรนี้จะถูกตั้งค่าให้เป็นค่าใดค่าหนึ่งต่อไปนี้:

- สำหรับคอลัมน์ต่างชนิดกัน, database manager ตั้งค่านี้ไว้เป็นชื่อที่แตกต่างกันอย่างสิ้นเชิง. ถ้าชื่อที่ตั้งไว้ยาวกว่า 30 ไบต์, ก็จะถูกตัดตอนปลายออกไป.
- สำหรับเลเบล, database manager ตั้งค่านี้ไว้ที่ 20 ไบต์แรกของเลเบล.
- สำหรับชื่อคอลัมน์, database manager จะตั้งค่านี้ไว้ที่ชื่อคอลัมน์.

สิ่งอ้างอิงที่เกี่ยวข้อง

“ตัวอย่าง: คำสั่ง Select เพื่อจัดสรรหน่วยเก็บสำหรับ SQLDA”

สมมติว่า แอ็พพลิเคชันของคุณจำเป็นต้องมีความสามารถในการจัดการกับคำสั่ง SELECT แบบ dynamic, เมื่อค่าหนึ่งเปลี่ยนเป็นอีกค่าหนึ่งสำหรับใช้ต่อไป. คำสั่งนี้สามารถอ่านได้จากจอแสดงผล, ซึ่งถูกส่งผ่านจากแอ็พพลิเคชันอื่น, หรือถูกสร้างขึ้นจากแอ็พพลิเคชันของคุณในขณะที่ปฏิบัติงาน.

ข้อมูลที่เกี่ยวข้อง

การโคัดคำสั่ง SQL ใน REXX Applications

ตัวอย่าง: คำสั่ง Select เพื่อจัดสรรหน่วยเก็บสำหรับ SQLDA:

สมมติว่า แอ็พพลิเคชันของคุณจำเป็นต้องมีความสามารถในการจัดการกับคำสั่ง SELECT แบบ dynamic, เมื่อค่าหนึ่งเปลี่ยนเป็นอีกค่าหนึ่งสำหรับใช้ต่อไป. คำสั่งนี้สามารถอ่านได้จากจอแสดงผล, ซึ่งถูกส่งผ่านจากแอ็พพลิเคชันอื่น, หรือถูกสร้างขึ้นจากแอ็พพลิเคชันของคุณในขณะที่ปฏิบัติงาน.

พูดได้อีกอย่างว่า, คุณไม่ทราบแน่ชัดว่าข้อความนี้จะส่งค่าอะไรคืนกลับมาในทุกครั้ง. แอ็พพลิเคชันจำเป็นต้องจัดการกับจำนวนที่แตกต่างกันออกไปของคอลัมน์ผลลัพธ์ที่ไม่ทราบชนิดข้อมูลที่แน่นอนก่อนล่วงหน้า

ยกตัวอย่างเช่น, ข้อความต่อไปนี้จำเป็นต้องถูกประมวลผล:

```
SELECT WORKDEPT, PHONENO
FROM CORPDATA.EMPLOYEE
WHERE LASTNAME = 'PARKER'
```

หมายเหตุ: SELECT statement นี้ไม่มี INTO clause. Dynamic SELECT statement จะต้องไม่มี INTO clause, ถึงแม้ว่าจะส่งค่าคืนมาเพียงแถวเดียว.

ข้อความจะถูกกำหนดค่าให้กับตัวแปรโฮสต์. ตัวแปรโฮสต์, ในกรณีนี้มีชื่อว่า DSTRING, จะถูกทำการประมวลผลโดยใช้คำสั่ง PREPARE ตามที่ได้แสดงไว้ดังนี้:

```
EXEC SQL  
PREPARE S1 FROM :DSTRING;
```

ขั้นถัดไป, คุณจำเป็นต้องหาค่าจำนวนของคอลัมน์ผลลัพธ์และชนิดของข้อมูล. ในการที่จะทำนั้น, ต้องอาศัย SQLDA.

ขั้นแรกในการกำหนด SQLDA, ก็คือจัดสรร (รีซอร์ส) เพื่อใช้งานให้. (การจัดสรร (รีซอร์ส) เพื่อใช้งานเป็นสิ่งที่ไม่จำเป็นใน REXX.) เทคนิคสำหรับการจองเนื้อที่ขึ้นกับภาษาที่ใช้. SQLDA จะต้องได้รับจัดสรร (รีซอร์ส) เพื่อใช้งานในขอบเขต 16 ไบต์. SQLDA ประกอบด้วยส่วนหัวที่ความยาวคงที่ซึ่งมีขนาดความยาว 16 ไบต์. ส่วนหัวจะต่อท้ายด้วยส่วนของ array ที่ความยาวแปรผัน (SQLVAR), แต่ละส่วนประกอบจะมีความยาว.

จำนวนของเนื้อที่ที่ต้องการในการจัดสรร (รีซอร์ส) เพื่อใช้งานขึ้นอยู่กับจำนวนองค์ประกอบที่ต้องการจะมีใน SQLVAR array. แต่ละคอลัมน์ที่เลือกจะต้องมีความสัมพันธ์กับองค์ประกอบของ SQLVAR array. ดังนั้น, จำนวนของคอลัมน์ที่แสดงใน SELECT statement จะเป็นตัวบอกจำนวนองค์ประกอบของ SQLVAR array ที่จะต้องทำการจัดสรร (รีซอร์ส) เพื่อใช้งาน. เนื่องจาก SELECT statement ถูกระบุที่เวลา รันไทม์, จึงเป็นไปได้ที่จะรู้แน่ชัดว่าจะมีการเข้าไปใช้คอลัมน์กี่คอลัมน์. ดังนั้น, คุณจึงควรที่จะประเมินจำนวนของคอลัมน์. สมมติว่า, ในตัวอย่างนี้, จะมีคอลัมน์ได้ไม่เกิน 20 คอลัมน์ที่จะถูกเรียกใช้โดย single SELECT statement. ในกรณีนี้, SQLVAR array ควรจะมีมิติเป็น 20, เพื่อให้แน่ใจว่าแต่ละรายการใน select-list มี entry ที่เกี่ยวเนื่องกันใน SQLVAR. จะทำให้ขนาดของ SQLDA เท่ากับ 20 x 80, หรือ 1600, บวก 16 สำหรับจำนวนไบต์ทั้งหมด 1616 ไบต์

การจัดสรร (รีซอร์ส) เพื่อใช้งานตามที่ประเมินให้เพียงพอสำหรับ SQLDA, จำเป็นที่จะต้องตั้งค่าในฟิลด์ SQLN ของ SQLDA ให้มีค่าเท่ากับจำนวนของ SQLVAR array element, ในที่นี้มีค่าเท่ากับ 20.

เมื่อทำการจัดสรรหน่วยเก็บ และเตรียมข้อมูลเบื้องต้นเกี่ยวกับขนาดแล้ว, คุณสามารถออกคำสั่ง DESCRIBE ได้.

```
EXEC SQL  
DESCRIBE S1 INTO :SQLDA;
```

เมื่อ DESCRIBE statement ถูกรัน, SQL จะใส่ค่าใน SQLDA เพื่อให้ข้อมูลเกี่ยวกับ select-list สำหรับ statement ของคุณ. ตารางต่อไปนี้แสดงเนื้อหาของ the SQLDA หลังจากการรัน DESCRIBE. จะแสดงเฉพาะ entry ส่วนที่มีความหมายใน context นี้เท่านั้น.

The SQLDA header ประกอบด้วย:

ตารางที่ 43. ส่วนหัวของ SQLDA

รายละเอียด	ค่า
SQLAID	'SQLDA'
SQLDABC	1616
SQLN	20
SQLD	2

SQLDAID เป็นฟิลด์ identifier ที่ initialize โดย SQL เมื่อ DESCRIBE ถูกรัน. SQLDABC เป็นไบต์ที่นับหรือบอกขนาดของ SQLDA. ส่วนหัวของ SQLDA จะต่อท้ายด้วย 2 occurrence ของโครงสร้าง SQLVAR, แต่ละ occurrence สำหรับแต่ละคอลัมน์ ในตารางผลลัพธ์ของคำสั่ง SELECT ซึ่งอธิบายได้ดังนี้:

ตารางที่ 44. SQLVAR องค์ประกอบ 1

รายละเอียด	ค่า
SQLTYPE	453
SQLLEN	3
SQLDATA (3:4)	37
SQLNAME	8 WORKDEPT

ตารางที่ 45. SQLVAR องค์ประกอบ 2

รายละเอียด	ค่า
SQLTYPE	453
SQLLEN	4
SQLDATA(3:4)	37
SQLNAME	7 PHONENO

โปรแกรมของคุณอาจจะต้องปรับเปลี่ยนค่าของ SQLN ถ้า SQLDA มีขนาดไม่ใหญ่พอที่จะทำการเก็บ SQLVAR elements ที่ได้ทำการประกาศไว้แล้ว. ตัวอย่างเช่น, สมมติว่าแทนที่ค่าสูงสุดจะเป็น 20 คอลัมน์, SELECT statement ได้ทำการส่งคืนกลับมาเป็น 27. SQL ไม่สามารถอธิบาย select-list นี้ได้เนื่องจาก SQLVAR ต้องการ element มากกว่าที่ระบุไว้ในเนื้อที่ที่กำหนด. แทนที่จะเป็นเช่นนั้น, SQL ตั้งค่าให้ SQLD เป็นตัวเลขที่แน่นอนของคอลัมน์ระบุโดย SELECT statement และส่วนที่เหลือของโครงสร้างก็จะถูกละเอาไว้. ดังนั้น, หลังจากการทำ DESCRIBE, ควรจะเปรียบเทียบค่าของ SQLN กับค่าของ SQLD. ถ้าค่าของ SQLD มากกว่าค่าของ SQLN, จะจัดสรร SQLDA ให้ใหญ่ขึ้นได้ยึดเอาค่า SQLD เป็นหลัก, ดังต่อไปนี้, และทำการ DESCRIBE อีกครั้ง:

```
EXEC SQL
  DESCRIBE S1 INTO :SQLDA; IF SQLN <= SQLD THEN
DO;
```

```
/*จัดสรร (รีซอร์ส) ให้ SQLDA ใหญ่ขึ้นโดยใช้ค่าของ SQLD.*/
/*ปรับค่าของ SQLN ให้เป็นค่าที่มากกว่า.*/
```

```
EXEC SQL
  DESCRIBE S1 INTO :SQLDA; END;
```

ถ้าใช้ DESCRIBE ใน non SELECT statement, SQL จะตั้งค่า SQLD ให้เป็น 0. ดังนั้น, ถ้าโปรแกรมของคุณถูกออกแบบมาให้ประมวลผลทั้ง SELECT และ non SELECT statement, สามารถอธิบายแต่ละข้อความหลังจากที่จัดเตรียมแล้วเพื่อที่จะตรวจสอบว่าเป็น SELECT statement หรือไม่. ตัวอย่างนี้ได้รับการออกแบบเพื่อให้ประมวลผลเฉพาะ SELECT statement; ค่าของ SQLD จะไม่ถูกตรวจสอบ.

โปรแกรมจะต้องวิเคราะห์ element ของ SQLVAR ที่ส่งกลับมาจากการ DESCRIBE ที่สมบูรณ์. รายการแรกในคือ WORKDEPT. ในฟิลด์ SQLTYPE, DESCRIBE จะส่งคืนค่าสำหรับชนิดข้อมูลของนิพจน์ และสามารถใช้ค่า null ได้หรือไม่.

ในตัวอย่างนี้, SQL จะตั้งค่า SQLTYPE เป็น 453 ใน SQLVAR องค์ประกอบ 1. ซึ่งระบุว่า WORKDEPT คือคอลัมน์ผลลัพธ์สตริงอักขระแบบความยาวคงที่ และค่า null จะถูกอนุญาตให้ใช้ในคอลัมน์.

SQL ตั้งค่า SQLLEN เป็นความยาวของคอลัมน์. เนื่องจากชนิดข้อมูลของ WORKDEPT เป็น CHAR, SQL จึงตั้งค่า SQLLEN ให้เท่ากับความยาวของคอลัมน์ตัวอักษร. สำหรับ WORKDEPT, ซึ่งมีความยาวเท่ากับ 3. ดังนั้น, เมื่อคำสั่ง SELECT ถูกรันในเวลาต่อมา, จึงจำเป็นที่จะต้องมีส่วนที่หน่วยเก็บที่เพียงพอในการเก็บชุดอักขระ CHAR(3) ได้.

เนื่องจากชนิดข้อมูลของ WORKDEPT เป็น CHAR FOR SBCS DATA, 4 ไบต์แรกของ SQLDATA จะถูกตั้งค่าให้เป็น CCSID ของคอลัมน์ตัวอักษร.

ฟิลด์สุดท้ายใน SQLVAR element จะเป็นสตริงอักขระแบบความยาวแปรผัน เรียกว่า SQLNAME. 2 ไบต์แรกของ SQLNAME จะเก็บค่าความยาวของข้อมูลตัวอักษรอยู่. ชื่อของข้อมูลตัวอักษรมักจะเป็นชื่อของคอลัมน์ที่ใช้ใน SELECT statement, ในกรณีนี้คือ WORKDEPT. exception ในกรณีนี้คือ รายการใน select-list ที่ไม่มีชื่อ, เช่น ฟังก์ชัน (ตัวอย่างเช่น, SUM(SALARY)), นิพจน์ (ตัวอย่างเช่น, A+B-C), และค่าคงที่. ในกรณีเหล่านี้, SQLNAME จะเป็นสตริงว่างเปล่า. SQLNAME สามารถเก็บค่าของเลเบลมากกว่าชื่อ. พารามิเตอร์ตัวหนึ่งที่เกี่ยวข้องกับ PREPARE และ DESCRIBE statement คือ USING clause. สามารถระบุได้ดังนี้:

```
EXEC SQL
  DESCRIBE S1 INTO:SQLDA
  USING LABELS;
```

ถ้าระบุว่า:

NAMES (หรือละพารามิเตอร์ USING ทั้งหมด)
จะใส่แต่ชื่อของคอลัมน์ในฟิลด์ SQLNAME.

SYSTEM NAMES
จะใส่แต่ชื่อของคอลัมน์ของระบบลงในฟิลด์ SQLNAME.

LABELS
จะใส่แต่เลเบลที่เชื่อมโยงกับคอลัมน์ที่แสดงอยู่ใน SQL statement.

ANY เลเบลจะถูกใส่ลงในฟิลด์ SQLNAME สำหรับคอลัมน์ที่มีเลเบล; มิฉะนั้นจะใส่ชื่อคอลัมน์แทน.

BOTH ชื่อและเลเบลจะถูกใส่ในฟิลด์ด้วยความยาวที่เท่ากัน. จำไว้ว่าให้เพิ่มขนาดของ SQLVAR array เนื่องจากมีการใช้จำนวนของ element นั้นเป็นสองเท่า.

ALL ชื่อคอลัมน์, เลเบล, และชื่อคอลัมน์ของระบบจะถูกใส่ในฟิลด์ด้วยความยาวที่เท่ากัน. จำไว้ว่าจะต้องเพิ่มขนาด SQLVAR array เป็นสามเท่า

ในตัวอย่างนี้, element ที่สองของ SQLVAR จะเก็บข้อมูลสำหรับคอลัมน์ที่สองที่ใช้ใน select: PHONENO. รหัส 453 ใน SQLTYPE ระบุว่า PHONENO เป็นคอลัมน์ CHAR. SQLLEN ถูกตั้งค่าให้เป็น 4.

ตอนนี้จำเป็นต้องติดตั้งเพื่อที่จะใช้ SQLDA ในการเรียกค่าออกมาเมื่อในขณะรัน SELECT statement.

หลังจากที่วิเคราะห์ผลลัพธ์ของ DESCRIBE, สามารถจัดสรร (รีซอร์ส) เพื่อใช้งานเนื้อที่สำหรับตัวแปรที่จะเก็บผลลัพธ์ของ SELECT statement. สำหรับ WORKDEPT, ฟิลด์ตัวอักษรที่มีความยาวเท่ากับ 3 จะต้องได้รับการจัดสรร (รีซอร์ส) เพื่อใช้งาน; สำหรับ PHONENO, จะต้องมีการจัดสรร (รีซอร์ส) เพื่อใช้งานฟิลด์ตัวอักษรที่มีความยาวเท่ากับ 4 . เนื่องจากผลลัพธ์ทั้งสองแบบนี้สามารถมีค่า NULL, ตัวแปรตัวบ่งชี้ต้องถูกจัดสรรเพื่อใช้งานสำหรับแต่ละฟิลด์ด้วยเช่นกัน.

หลังจากที่ได้จัดสรร (รีซอร์ส) เพื่อใช้งาน, จะต้องตั้งค่าให้ SQLDATA และ SQLIND ให้ชี้ไปยังพื้นที่ของการจัดสรร (รีซอร์ส) เพื่อใช้งาน. สำหรับแต่ละ element ของ SQLVAR array, SQLDATA จะชี้ไปยังที่ที่ผลลัพธ์จะถูกนำไปเก็บไว้. SQLIND ชี้ไปที่ที่ค่าของ null indicator จะถูกเก็บเอาไว้. ตารางดังต่อไปนี้จะแสดงลักษณะโครงสร้างในขณะนี้. เฉพาะ entry ที่มีความหมายจะถูกนำมาแสดงในบริบทนี้:

ตารางที่ 46. ส่วนหัวของ SQLDA

รายละเอียด	ค่า
SQLAID	'SQLDA'
SQLDABC	1616
SQLN	20
SQLD	2

ตารางที่ 47. SQLVAR องค์ประกอบ 1

รายละเอียด	ค่า
SQLTYPE	453
SQLLEN	3
SQLDATA	ตัวชี้ไปยังพื้นที่ที่เก็บผลลัพธ์แบบ CHAR(3)
SQLIND	ตัวชี้ไปยังตัวบ่งชี้จำนวนเต็มแบบ 2 ไบต์สำหรับคอลัมน์ผลลัพธ์

ตารางที่ 48. SQLVAR องค์ประกอบ 2

รายละเอียด	ค่า
SQLTYPE	453
SQLLEN	4
SQLDATA	ตัวชี้สำหรับพื้นที่ที่เก็บผลลัพธ์แบบ CHAR(4)
SQLIND	ตัวชี้ไปยังตัวบ่งชี้จำนวนเต็มแบบ 2 ไบต์สำหรับคอลัมน์ผลลัพธ์

ตอนนี้ก็พร้อมที่จะเรียกดูผลลัพธ์ของ SELECT statements . การระบุ SELECT statement แบบ dynamic จะต้องไม่มี INTO statement. ดังนั้น, SELECT statement ที่กำหนดแบบ dynamic ต้องใช้เคอร์เซอร์. รูปแบบพิเศษของ DECLARE, OPEN, และ FETCH ถูกใช้สำหรับการระบุคำสั่ง SELECT แบบ dynamic.

คำสั่ง DECLARE สำหรับตัวอย่างนี้คือ:

```
EXEC SQL DECLARE C1 CURSOR FOR S1;
```

ดังที่ให้เห็นแล้วว่า, ความแตกต่างเพียงประการเดียวคือชื่อของ prepared SELECT statement (S1) จะถูกใช้แทนชื่อของตัว SELECT statement เอง. การดึงข้อมูลออกมาสำหรับแถวที่เป็นผลลัพธ์จะทำได้ดังนี้:

```
EXEC SQL
  OPEN C1;
EXEC SQL
  FETCH C1 USING DESCRIPTOR :SQLDA;
DO WHILE (SQLCODE = 0);
/*Process the results pointed to by SQLDATA*/
EXEC SQL
  FETCH C1 USING DESCRIPTOR :SQLDA;
END;
EXEC SQL
  CLOSE C1;
```

เคอร์เซอร์ถูกเปิด. แถวที่เป็นผลลัพธ์จาก SELECT จะถูกส่งคืนมาครั้งละหนึ่งแถวโดยใช้ FETCH statement. ใน FETCH statement, จะไม่มีรายชื่อของตัวแปรโฮสต์อยู่. แทนที่จะเป็นเช่นนั้น, FETCH statement จะบอกให้ SQL ส่งคืนผลลัพธ์ไปในพื้นที่ที่ระบุโดย SQLDA. ผลลัพธ์จะถูกส่งคืนมาในพื้นที่จัดเก็บข้อมูลโดยฟิลด์ SQLDATA และ SQLIND ของ SQLVAR element. หลังจากที่ได้ทำการประมวลผล FETCH statement, ตัวชี้ SQLDATA สำหรับ WORKDEPT มีค่าอ้างอิงตั้งเป็น 'E11'. ซึ่งค่าของตัวบ่งชี้ที่เกี่ยวข้องเนื่องกันเป็น 0 เนื่องจากค่า non-null ถูกส่งคืนมา. ตัวชี้ SQLDATA สำหรับ PHONENO มีค่าอ้างอิงเป็น '4502'. และค่าของตัวบ่งชี้ที่เกี่ยวข้องเนื่องกันเป็น 0 เนื่องจากค่า non-null ถูกส่งคืนมา.

สิ่งอ้างอิงที่เกี่ยวข้อง

“คำสั่ง SELECT แบบ varying-list” ในหน้า 276

ใน dynamic SQL, คำสั่ง SELECT แบบ varying-list เป็นรูปแบบหนึ่งสำหรับจำนวนและรูปแบบของคอลัมน์ผลลัพธ์ที่จะถูกส่งคืนมาที่ไม่สามารถคาดเดาได้; นั่นคือ, คุณไม่สามารถทราบจำนวนตัวแปรที่คุณต้องการ, หรือชนิดของข้อมูล.

“รูปแบบของ SQLDA” ในหน้า 277

SQLDA ประกอบด้วยตัวแปรสี่ตัวตามด้วยเลขเฉพาะของการเกิดขึ้นตามลำดับของกลุ่มตัวแปรหกตัวที่ชื่อ SQLVAR.

I ตัวอย่าง: คำสั่ง Select โดยใช้ SQL descriptor ที่ถูกจัดสรรแล้ว:

I สมมติว่า แอ็พพลิเคชันของคุณจำเป็นต้องจัดการกับคำสั่ง SELECT แบบ dynamic; เมื่อค่าหนึ่งเปลี่ยนเป็นอีกค่าหนึ่งสำหรับ
I ใช้ต่อไป. คำสั่งนี้สามารถอ่านได้จากจอแสดงผล, ซึ่งถูกส่งผ่านจากแอ็พพลิเคชันอื่น, หรือถูกสร้างขึ้นจากแอ็พพลิเคชันของ
I คุณแบบ dynamic.

I พูดได้อีกอย่างว่า, คุณไม่ทราบแน่ชัดว่า คำสั่งนี้จะส่งค่าอะไรคืนกลับมาในทุกครั้ง. แอ็พพลิเคชันจำเป็นต้องจัดการกับจำนวน
I ที่แตกต่างกันออกไปของคอลัมน์ผลลัพธ์ที่ไม่ทราบชนิดข้อมูลที่แน่นอนก่อนล่วงหน้า

I ยกตัวอย่างเช่น, คำสั่งต่อไปนี้จำเป็นต้องถูกประมวลผล:

```
I   SELECT WORKDEPT, PHONENO
I     FROM CORPDATA.EMPLOYEE
I     WHERE LASTNAME = 'PARKER'
```

I **หมายเหตุ:** คำสั่ง SELECT นี้ไม่มี INTO clause. คำสั่ง SELECT แบบ dynamic จะต้องไม่มี INTO clause, ถึงแม้ว่าจะส่งค่า
I คืนมาเพียงแถวเดียว.

I คำสั่งจะถูกกำหนดค่าให้กับตัวแปรโฮสต์. ตัวแปรโฮสต์, ในกรณีนี้มีชื่อว่า DSTRING, จะถูกทำการประมวลผลโดยใช้คำสั่ง
I PREPARE ตามที่ได้แสดงไว้ดังนี้:

```
| EXEC SQL
| PREPARE S1 FROM :DSTRING;
```

| ขั้นถัดไป, คุณจำเป็นต้องหาค่าจำนวนของคอลัมน์ผลลัพธ์และชนิดของข้อมูล. หากต้องการทำสิ่งนี้, คุณจำเป็นต้องจัดสรรจำนวน entry ขนาดใหญ่ที่สุดสำหรับ SQL descriptor ที่คุณคิดว่าคุณต้องการ. สมมติว่า มีคอลัมน์ไม่เกิน 20 คอลัมน์ถูกเรียกใช้ โดยคำสั่ง SELECT เดียว.

```
| EXEC SQL
| ALLOCATE DESCRIPTOR 'mydescr' WITH MAX 20;
```

| ถึงตอนนี้ descriptor จะถูกจัดสรร, คำสั่ง DESCRIBE สามารถเรียกใช้เพื่อรับข้อมูลคอลัมน์.

```
| EXEC SQL
| DESCRIBE S1 USING DESCRIPTOR 'mydescr';
```

| เมื่อคำสั่ง DESCRIBE ถูกรัน, SQL จะใส่ค่าที่ได้เตรียมข้อมูลเกี่ยวกับรายการที่เลือกของคำสั่งเข้าไปใน SQL descriptor area ซึ่งถูกนิยามโดย 'mydescr'.

| ถ้า DESCRIBE กำหนดว่ามี entry ไม่เพียงพอที่จะถูกจัดสรรใน descriptor, SQLCODE +239 จะถูกเรียกใช้. ส่วนหนึ่งของการวินิจฉัยนี้, การแทนที่ค่าข้อความที่สองจะบ่งชี้ถึงจำนวนของ entry ที่ต้องการ. ตัวอย่างโค้ดต่อไปนี้จะแสดงวิธีที่เงื่อนไขสามารถตรวจพบ และแสดง descriptor ที่ถูกจัดสรรด้วยขนาดที่ใหญ่กว่า.

```
| /* Determine the returned SQLCODE from the DESCRIBE statement */
| EXEC SQL
| GET DIAGNOSTICS CONDITION 1: returned_sqlcode = DB2_RETURNED_SQLCODE;
|
| if returned_sqlcode = 239 then do;
|
| /* Get the second token for the SQLCODE that indicated
|    not enough entries were allocated */
|
| EXEC SQL
| GET DIAGNOSTICS CONDITION 1: token = DB2_ORDINAL_TOKEN_2;
| /* Move the token variable from a character host variable into an integer host variable */
| EXEC SQL
| SET :var1 = :token;
| /* Deallocate the descriptor that is too small */
| EXEC SQL
| DEALLOCATE DESCRIPTOR 'mydescr';
| /* Allocate the new descriptor to be the size indicated by the retrieved token */
| EXEC SQL
| ALLOCATE DESCRIPTOR 'mydescr' WITH MAX :var1;
| /* Perform the describe with the larger descriptor */
| EXEC SQL
| DESCRIBE s1 USING DESCRIPTOR 'mydescr';
| end;
```

| ถึงตอนนี้, descriptor จะมีข้อมูลเกี่ยวกับคำสั่ง select. และคุณพร้อมที่จะดึงผลลัพธ์ของคำสั่ง SELECT ออกมา. สำหรับ SQL แบบ dynamic, คำสั่ง SELECT INTO จะไม่อนุญาตให้ใช้. คุณต้องใช้เคอร์เซอร์.

```
| EXEC SQL
| DECLARE C1 CURSOR FOR S1;
```

| คุณจะสังเกตเห็นว่าชื่อคำสั่งที่ถูกจัดเตรียมจะถูกใช้ในการประกาศเคอร์เซอร์แทนการทำคำสั่ง SELECT ให้สมบูรณ์. ถึงตอนนี้
| คุณสามารถวนซ้ำแถวที่เลือก, ประมวลผลแถวเหล่านั้นตามที่อ่านได้. ตัวอย่างโค้ดต่อไปนี้จะแสดงถึงวิธีการทำสิ่งนี้.

```
| EXEC SQL
| OPEN C1;
|
| EXEC SQL
|   FETCH C1 USING SQL DESCRIPTOR 'mydescr';
| do while not at end of data;
|
|   /* process current data returned (see below for discussion of doing this) */
|
|   /* then read the next row */
|
|   EXEC SQL
|     FETCH C1 USING SQL DESCRIPTOR 'mydescr';
| end;
|
| EXEC SQL
| CLOSE C1;
```

| เคอร์เซอร์ถูกเปิด. แถวที่เป็นผลลัพธ์จากคำสั่ง SELECT จะถูกส่งคืนมาครั้งละหนึ่งแถวโดยใช้คำสั่ง FETCH. ในคำสั่ง
| FETCH, จะไม่มีรายชื่อของตัวแปรโฮสต์อยู่. แทนที่จะเป็นเช่นนั้น, คำสั่ง FETCH จะบอกให้ SQL ส่งคืนผลลัพธ์เข้าไปใน
| descriptor area.

| หลังจากประมวลผล FETCH แล้ว, คุณสามารถใช้คำสั่ง GET DESCRIPTOR เพื่ออ่านค่าเหล่านั้น. อันดับแรก, คุณต้องอ่านค่า
| ส่วนหัวที่บ่งชี้ถึงจำนวน descriptor entry ที่ถูกใช้.

```
| EXEC SQL
| GET DESCRIPTOR 'mydescr' :count = COUNT;
```

| หลังจากนั้น คุณสามารถอ่านข้อมูลเกี่ยวกับ descriptor entry แต่ละตัว. หลังจากที่คุณกำหนดชนิดข้อมูลของคอลัมน์ผลลัพธ์
| แล้ว, คุณสามารถทำให้ GET DESCRIPTOR อื่นส่งคืนค่าที่เป็นจริง. หากต้องการรับค่าของตัวบ่งชี้, ให้ระบุไอเท็ม
| INDICATOR. ถ้าค่าของไอเท็ม INDICATOR เป็นลบ, ค่าของไอเท็ม DATA จะไม่ถูกนิยาม. จนกว่า FETCH อื่นจะถูกทำ,
| ไอเท็ม descriptor จะยังคงรักษาค่าเหล่านั้นไว้.

```
| do i = 1 to count;
|   GET DESCRIPTOR 'mydescr' VALUE :i /* set entry number to get */
|                                     :type = TYPE, /* get the data type */
|                                     :length = LENGTH, /* length value */
|                                     :result_ind = INDICATOR;
|
|   if result_ind >= 0 then
|     if type = character
|       GET DESCRIPTOR 'mydescr' VALUE :i
|                                     :char_result = DATA; /* read data into character field */
|     else
|       if type = integer
|         GET DESCRIPTOR 'mydescr' VALUE :i
|                                     :int_result = DATA; /* read data into integer field */
|       else
|         /* continue checking and processing for all data types that might be returned */
|     end;
| end;
```

มีไอเท็ม descriptor อื่นๆ หลายไอเท็มที่คุณอาจต้องการตรวจสอบ เพื่อกำหนดวิธีการจัดการกับข้อมูลผลลัพธ์. PRECISION, SCALE, DB2_CCSD, และ DATETIME_INTERVAL_CODE อยู่ระหว่างกัน. ตัวแปรโฮสต์ที่มีการอ่านค่า DATA เข้าไปในตัวแปรต้องมีชนิดข้อมูลเดียวกัน และ CCSID ต้องเป็นข้อมูลที่อ่านได้. ถ้าชนิดข้อมูลมีความยาวผันแปร, ตัวแปรโฮสต์จะถูกประกาศความยาวได้ยาวกว่าข้อมูลจริง. สำหรับชนิดข้อมูลอื่นๆ ทั้งหมด, ความยาวต้องตรงกัน.

NAME, DB2_SYSTEM_COLUMN_NAME, และ DB2_LABEL จะถูกใช้เพื่อรับค่าชื่อที่สัมพันธ์กันสำหรับคอลัมน์ผลลัพธ์. โปรดดู GET DESCRIPTOR สำหรับข้อมูลเพิ่มเติมเกี่ยวกับไอเท็มที่ถูกส่งคืนสำหรับคำสั่ง GET DESCRIPTOR และสำหรับ definition ของค่า TYPE

ตัวทำเครื่องหมายพารามิเตอร์:

ในตัวอย่างที่เราใช้กันนั้น, คำสั่ง SELECT ที่ถูกรันแบบ dynamic มีค่าคงที่ใน WHERE clause.

ในตัวอย่าง, จะเป็น:

```
WHERE LASTNAME = 'PARKER'
```

ถ้าต้องการรันคำสั่ง SELECT ตัวเดียวกันหลายๆ ครั้ง, โดยใช้ค่า LASTNAME ที่แตกต่างกัน, คุณสามารถใช้คำสั่ง SQL ที่มีลักษณะดังนี้:

```
SELECT WORKDEPT, PHONENO
FROM CORPDATA.EMPLOYEE
WHERE LASTNAME = ?
```

เมื่อใช้ตัวทำเครื่องหมายพารามิเตอร์, แอ็พพลิเคชันของคุณไม่จำเป็นต้องตั้งค่าชนิดข้อมูล และค่าสำหรับพารามิเตอร์จนกว่ารันไทม์. ด้วยการระบุ descriptor บนคำสั่ง OPEN, คุณสามารถใช้ค่าแทนตัวทำเครื่องหมายพารามิเตอร์ในคำสั่ง SELECT ได้.

เมื่อต้องการโค้ดโปรแกรมแบบนี้, คุณจำเป็นต้องใช้คำสั่ง OPEN กับ descriptor clause. คำสั่ง SQL นี้ไม่เพียงแต่จะใช้ในการเปิดเคอร์เซอร์, แต่ยังใช้แทนตัวทำเครื่องหมายพารามิเตอร์แต่ละตัวด้วยค่าของ descriptor entry ที่เกี่ยวข้องกัน. ชื่อ descriptor ที่คุณระบุด้วยคำสั่งนี้ต้องระบุ descriptor ที่มีค่าของ definition ที่ถูกต้อง. descriptor นี้ไม่ได้ถูกใช้เพื่อส่งคืนข้อมูลเกี่ยวกับหน่วยข้อมูล ซึ่งเป็นส่วนหนึ่งของรายการ SELECT. และมีข้อมูลเกี่ยวกับค่าที่ถูกใช้เพื่อแทนตัวทำเครื่องหมายพารามิเตอร์ในคำสั่ง SELECT. ซึ่งจะได้รับข้อมูลมาจากแอ็พพลิเคชัน, ที่ต้องมีการออกแบบเพื่อแทนค่าที่เหมาะสมลงในฟิลด์ของ descriptor. descriptor จะพร้อมใช้งานโดย SQL เพื่อแทนตัวทำเครื่องหมายพารามิเตอร์ด้วยค่าที่แท้จริง.

เมื่อคุณใช้ SQLDA สำหรับอินพุตในคำสั่ง OPEN โดยใช้ USING DESCRIPTOR clause, คุณไม่จำเป็นต้องระบุค่าฟิลด์ทั้งหมด. โดยเฉพาะอย่างยิ่ง, SQLDAID, SQLRES, และ SQLNAME สามารถปล่อยว่างไว้ได้ (SQLNAME สามารถตั้งค่าได้ถ้าจำเป็นต้องใช้ค่าของ CCSID) ดังนั้น, เมื่อคุณใช้วิธีนี้เพื่อแทนตัวทำเครื่องหมายพารามิเตอร์ด้วยค่าเหล่านี้, คุณจำเป็นต้องกำหนด:

- จำนวนตัวทำเครื่องหมายที่ต้องมี
- ชนิดข้อมูลและแอ็ททริบิวต์ของตัวทำเครื่องหมายพารามิเตอร์เหล่านี้ (SQLTYPE, SQLLEN, และ SQLNAME)
- จำเป็นต้องมีตัวแปรตัวบ่งชี้หรือไม่

นอกจากนั้น, ถ้ารู้ที่จัดการทั้งคำสั่ง SELECT และ non-SELECT, คุณอาจต้องกำหนดประเภทของคำสั่งด้วย.

ถ้าแอ็พพลิเคชันของคุณใช้ตัวทำเครื่องหมายพารามิเตอร์, โปรแกรมของคุณจะต้องทำตามขั้นตอนต่อไปนี้. ซึ่งสามารถทำได้โดยใช้ SQLDA หรือ descriptor ที่จัดสรรแล้ว.

1. อ่านคำสั่งลงในตัวแปรโฮสต์ชื่อ DSTRING ซึ่งเป็นสตริงอักขระแบบความยาวแปรผัน .
2. หาค่าตัวเลขของเครื่องหมายพารามิเตอร์.
3. จัดสรรขนาดของ SQLDA หรือใช้ ALLOCATE DESCRIPTOR เพื่อจัดสรร descriptor ด้วยจำนวนของ entry. ไม่สามารถใช้ได้ใน REXX.
4. สำหรับ SQLDA, ให้ตั้งค่า SQLN และ SQLD เป็นจำนวนของตัวทำเครื่องหมายพารามิเตอร์. SQLN ไม่สามารถใช้ได้ใน REXX. สำหรับ descriptor ที่จัดสรรแล้ว, ให้ใช้ SET DESCRIPTOR เพื่อตั้งค่า COUNT entry เป็นจำนวนของตัวทำเครื่องหมายพารามิเตอร์.
5. สำหรับ SQLDA, ให้ตั้งค่า SQLDABC เท่ากับ $SQLN * LENGTH(SQLVAR) + 16$. ไม่สามารถใช้ได้ใน REXX.
6. สำหรับตัวทำเครื่องหมายพารามิเตอร์:
 - a. หาชนิดข้อมูล, ความยาว, และตัวบ่งชี้.
 - b. สำหรับ SQLDA, ให้ตั้งค่า SQLTYPE และ SQLLEN สำหรับตัวทำเครื่องหมายพารามิเตอร์. สำหรับ descriptor ที่จัดสรรแล้ว, ให้ใช้ SET DESCRIPTOR เพื่อตั้งค่า entry สำหรับ TYPE, LENGTH, PRECISION, และ SCALE สำหรับตัวทำเครื่องหมายพารามิเตอร์แต่ละตัว.
 - c. สำหรับ SQLDA, ให้จัดสรรหน่วยเก็บเพื่อจัดการกับค่าอินพุต.
 - d. สำหรับ SQLDA, ให้ตั้งค่าเหล่านี้ในหน่วยเก็บ.
 - e. สำหรับ SQLDA, ให้ตั้งค่า SQLDATA และ SQLIND (ถ้าสามารถใช้ได้) สำหรับตัวทำเครื่องหมายพารามิเตอร์แต่ละตัว. สำหรับ descriptor ที่จัดสรรแล้ว, ให้ใช้ SET DESCRIPTOR เพื่อตั้งค่า entry สำหรับ DATA และ INDICATOR (ถ้าสามารถใช้งานได้) สำหรับตัวทำเครื่องหมายพารามิเตอร์แต่ละตัว.
 - f. ถ้ามีการใช้ตัวแปรอักขระ และมีค่าของ CCSID ที่ไม่ใช่ดีฟอลต์งาน CCSID, หรือตัวแปรกราฟิกที่ถูกใช้และมี CCSID ที่ไม่ใช่ DBCS CCSID ที่ถูกเชื่อมโยงสำหรับงาน CCSID,
 - สำหรับ SQLDA, ให้ตั้งค่า SQLNAME (SQLCCSID in REXX) ตามลำดับ.
 - สำหรับ SQL descriptor ที่จัดสรรแล้ว, ให้ใช้ SET DESCRIPTOR เพื่อตั้งค่า DB2_CCSID.
 - g. ใช้คำสั่ง OPEN พร้อมด้วย USING DESCRIPTOR clause (สำหรับ SQLDA) หรือ USING SQL DESCRIPTOR clause (สำหรับ descriptor ที่จัดสรรแล้ว) เพื่อเปิดเคอร์เซอร์ และแทนค่าของตัวทำเครื่องหมายพารามิเตอร์แต่ละตัว.

จากนั้นจะนำข้อความมาประมวลผลได้ตามปกติ.

การใช้ SQL แบบ dynamic ผ่านไคลเอ็นต์อินเทอร์เน็ตเฟส

คุณสามารถเข้าถึงข้อมูล DB2 UDB for iSeries ผ่านไคลเอ็นต์อินเทอร์เน็ตเฟสบนเซิร์ฟเวอร์.

การเข้าถึงข้อมูลด้วย Java

คุณสามารถเข้าถึงข้อมูล DB2 UDB for iSeries ในโปรแกรม Java ของคุณด้วยไต่เวอร์ Developer Kit for Java Database Connectivity (JDBC).

ไต่เวอร์อนุญาตให้คุณทำงานต่อไปนี้.

- เข้าถึงไฟล์ฐานข้อมูล
- เข้าถึงฟังก์ชันฐานข้อมูลของ JDBC ด้วย Structured Query Language (SQL) ที่ฝังอยู่สำหรับ Java
- รับข้อความ SQL และประมวลผลผลลัพธ์.

ข้อมูลที่เกี่ยวข้อง

การตั้งค่าการใช้ IBM Developer Kit for Java (JDBC driver)

การเข้าถึงข้อมูลด้วย Domino

Domino® for iSeries คือผลิตภัณฑ์เซิร์ฟเวอร์ Domino ที่ให้คุณรวมข้อมูลจากฐานข้อมูล DB2 UDB for iSeries และฐานข้อมูล Domino ทั้งสองแนวทาง.

เพื่อให้ได้ประโยชน์จากการรวมครั้งนี้, คุณต้องทำความเข้าใจการควบคุมสิทธิการทำงานระหว่างฐานข้อมูลทั้งสองประเภท.

ข้อมูลที่เกี่ยวข้อง

Domino for iSeries

การเข้าถึงข้อมูลด้วย Open Database Connectivity (ODBC)

คุณใช้ไดรเวอร์ iSeries Access for Windows® ODBC เพื่อเรียกใช้งานไคลเอ็นต์แอปพลิเคชัน ODBC เพื่อแบ่งใช้ข้อมูลระหว่างกัน และกับเซิร์ฟเวอร์ได้อย่างมีประสิทธิภาพที่สุด.

ข้อมูลที่เกี่ยวข้อง

การดูแล ODBC

การเข้าถึงข้อมูลด้วย i5/OS Portable Application Solutions Environment (i5/OS PASE)

i5/OS PASE คือสภาวะแวดล้อมรันไทม์ที่รวบรวมไว้สำหรับแอปพลิเคชัน AIX® หรือแอปพลิเคชันอื่น เช่น UNIX ที่รันบนระบบ iSeries.

ข้อมูลที่เกี่ยวข้อง

i5OS PASE

การเข้าถึงข้อมูลด้วย iSeries Access for Windows OLE DB Provider

iSeries Access for Windows OLE DB Provider, พร้อมกับ Programmer's Toolkit, ช่วยให้การพัฒนาแอปพลิเคชัน iSeries ไคลเอ็นต์/เซิร์ฟเวอร์ เป็นไปได้ง่าย และรวดเร็วจากเครื่องไคลเอ็นต์ที่เป็น Windows.

iSeries Access for Windows OLE DB Provider ช่วยให้โปรแกรมเมอร์ iSeries สามารถเข้าถึงอินเตอร์เฟซระดับเรกคอร์ดของโลจิคัลและฟิสิคัลของ iSeries บนไฟล์ฐานข้อมูล DB2 Universal Database™ (UDB) for iSeries. นอกจากนี้ยังสนับสนุนการทำงานร่วมกับ SQL, ดาต้าคิว, โปรแกรม และ คำสั่ง. ถ้าคุณใช้ Visual Basic, ด้วย Visual Basic Wizards ช่วยให้ง่ายและสะดวกที่จะปรับแอปพลิเคชันให้เป็นไปตามความต้องการ.

ข้อมูลที่เกี่ยวข้อง

iSeries Access for Windows OLE DB Provider

การเข้าถึงข้อมูลด้วย Net.Data

Net.Data คือแอปพลิเคชันที่รันบนเซิร์ฟเวอร์และอนุญาตให้คุณสร้างเอกสารเว็บแบบ dynamic ที่เรียกว่าเว็บแมโครได้อย่างง่ายดาย. เว็บแมโครที่ถูกสร้างขึ้นสำหรับ Net.Data จะมีความเรียบง่ายของ HTML พร้อมด้วยฟังก์ชันของแอปพลิเคชัน CGI-BIN.

Net.Data ทำให้การเพิ่มข้อมูลปัจจุบันลงบนหน้าเว็บเป็นเรื่องง่าย. ข้อมูลปัจจุบันหมายความว่ารวมถึงข้อมูลที่เก็บอยู่ในฐานข้อมูล, ไฟล์, แอปพลิเคชัน, และการให้บริการของระบบ.

ข้อมูลที่เกี่ยวข้อง

Net.data programs for the HTTP Server

การเข้าถึงข้อมูลผ่านพาร์ติชัน ลินุกซ์

IBM และผู้จำหน่ายลินุกซ์® หลายๆ รายได้ร่วมมือกันรวบรวมระบบปฏิบัติการ ลินุกซ์ เข้ากับความเชื่อถือได้ของเซิร์ฟเวอร์ iSeries.

ลินุกซ์นำมาซึ่งแอพลิเคชันบนเว็บรุ่นใหม่ให้กับ iSeries. IBM ได้แก้ไขคอร์เนลของลินุกซ์ PowerPC® เพื่อให้ทำงานในโลจิคัลพาร์ติชันระดับรองได้ และได้แจกจ่ายคอร์เนลที่แก้ไขแล้วกลับไปกลุ่มผู้ใช้ลินุกซ์.

ข้อมูลที่เกี่ยวข้อง

ลินุกซ์และเซิร์ฟเวอร์ iSeries

การเข้าถึงข้อมูลโดยใช้ Distributed Relational Database (DRDA)

ฐานข้อมูลเชิงสัมพันธ์แบบกระจาย ประกอบไปด้วยชุด SQL อ็อบเจกต์ที่กระจายอยู่บนระบบคอมพิวเตอร์ที่เชื่อมต่อถึงกันและกัน. แต่ละฐานข้อมูลเชิงสัมพันธ์จะมีตัวจัดการฐานข้อมูลเชิงสัมพันธ์ เพื่อจัดการกับตารางในสภาวะแวดล้อมของมันเอง.

ตัวจัดการฐานข้อมูลทำการติดต่อและทำงานร่วมกับตัวจัดการฐานข้อมูลอื่นโดยที่จะอนุญาตให้ตัวจัดการข้อมูลที่มีอยู่สามารถรันคำสั่ง SQL ที่อยู่บนฐานข้อมูลเชิงสัมพันธ์ในระบบอื่นได้.

สิ่งอ้างอิงที่เกี่ยวข้อง

“ฟังก์ชันของฐานข้อมูลเชิงสัมพันธ์แบบกระจายและ SQL” ในหน้า 309

ฐานข้อมูลเชิงสัมพันธ์แบบกระจาย ประกอบไปด้วยชุด SQL อ็อบเจกต์ที่กระจายอยู่บนระบบคอมพิวเตอร์ที่เชื่อมต่อถึงกันและกัน.

การใช้ SQL แบบโต้ตอบ

SQL แบบโต้ตอบอนุญาตให้โปรแกรมเมอร์ หรือผู้ดูแลฐานข้อมูลมีความรวดเร็ว และสะดวกในการกำหนด, อัปเดต, ลบ, หรือสำรวจข้อมูลเพื่อทำการทดสอบ, วิเคราะห์ปัญหา, และการดูแลฐานข้อมูล.

โปรแกรมเมอร์, ที่ใช้ SQL แบบโต้ตอบ, สามารถแทรกแถวลงยังตารางและทดสอบข้อความ SQL ก่อนทำการรันข้อความเหล่านั้นในแอพลิเคชันโปรแกรม. ผู้บริหารระบบฐานข้อมูลสามารถใช้ SQL แบบโต้ตอบเพื่อให้ privilege หรือ เรียกคืน privilege, สร้างหรือลบแบบแผน, ตาราง, หรือมุมมอง, หรือเลือกข้อมูลจากตารางแค็ตตาล็อกระบบ.

หลังจากข้อความ SQL แบบโต้ตอบถูกรัน, ข้อความแสดงการเสร็จสิ้นหรือข้อความแสดงข้อผิดพลาดจะปรากฏ. นอกจากนี้, โดยปกติแล้ว ข้อความแสดงสถานะจะปรากฏขึ้นระหว่างข้อความที่รันเป็นเวลานาน.

คุณสามารถดูคำอธิบายจากข้อความโต้ตอบโดยเลื่อนเคอร์เซอร์ไปไว้บนข้อความและกด F1=Help.

ฟังก์ชันพื้นฐานของ SQL แบบโต้ตอบคือ:

- ฟังก์ชัน entry ข้อความ อนุญาตให้คุณ:
 - พิมพ์ข้อความ SQL แบบโต้ตอบและรันข้อความ.
 - เรียกข้อความออกมาและแก้ไขข้อความ.
 - พร้อมท์สำหรับข้อความ SQL.

- เลื่อนไปยังข้อความ(คำสั่ง) และ ข้อความ (แสดงผล) ก่อนหน้านี้.
- เรียกเซอวิสเซสชัน.
- เรียกฟังก์ชันการเลือกรายการ.
- ออกจาก SQL แบบโต้ตอบ.
- ฟังก์ชันพร้อมต์ อนุญาตให้คุณพิมพ์ข้อความ SQL หรือข้อความ SQL ส่วนหนึ่ง, กด F4=Prompt, และคุณจะถูกถามให้ใส่ไวยากรณ์ของข้อความ. นอกจากนี้คุณยังสามารถกด F4 เพื่อรับเมนูของข้อความ SQL ทั้งหมดได้ด้วย. จากเมนูนี้, คุณสามารถเลือกข้อความและจะถูกถามให้ใส่ไวยากรณ์ของข้อความ.
- ฟังก์ชันการเลือกรายการ อนุญาตให้คุณเลือกรูปร่างข้อมูลเชิงสัมพันธ์, แบบแผน, ตาราง, มุมมอง, คอลัมน์, ข้อจำกัด, หรือ SQL แพ็กเกจจากรายการตามที่คุณมีสิทธิ์.
รายการที่คุณเลือกจากรายการอาจนำมาแทรกลงในข้อความ SQL ที่ตำแหน่งของเคอร์เซอร์.
- ฟังก์ชัน session services อนุญาตให้คุณ:
 - เปลี่ยนแอ็ททริบิวต์เซสชัน.
 - พิมพ์เซสชันปัจจุบัน.
 - ย้าย entry ทั้งหมดออกจากเซสชันปัจจุบัน.
 - บันทึกเซสชันในซอร์สไฟล์.

หมายเหตุ:

1. คำว่าคอลเล็คชัน จะถูกใช้พ้องกับคำว่า แบบแผน.
2. ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า 360.

หลักการที่เกี่ยวข้อง

“แอ็พพลิเคชัน Dynamic SQL” ในหน้า 271

Dynamic SQL อนุญาตให้แอ็พพลิเคชันกำหนด และ รัน SQL statement ที่เวลารันไทม์ของโปรแกรม. แอ็พพลิเคชันที่มีไว้เพื่อ dynamic SQL จะถือว่าเป็น อินพุต (หรือ build) ของ SQL statement ในรูปของสตริงอักขระ. แอ็พพลิเคชันไม่จำเป็นต้องทราบ คำสั่ง SQL ประเภทใดที่จะรัน.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การประมวลผลคำสั่ง non-SELECT” ในหน้า 273

เมื่อต้องการสร้างคำสั่ง SQL non-SELECT แบบ dynamic, คุณจำเป็นต้องตรวจสอบว่า คำสั่ง SQL ที่คุณต้องการสร้างคือคำสั่งที่สามารถรันแบบ dynamic ได้ และสร้างคำสั่ง SQL ได้.

การเริ่มต้นใช้งาน SQL แบบโต้ตอบ

คุณสามารถเริ่มต้นใช้งาน SQL แบบโต้ตอบโดยการพิมพ์STRSQL บนบรรทัดรับคำสั่ง i5/OS .

จอแสดงผล Enter SQL Statements จะปรากฏขึ้นมา. นี่คือจอแสดงผลหลัก SQL แบบโต้ตอบ. จากจอแสดงผลนี้, คุณสามารถใส่คำสั่ง SQL และใช้:

- F4=prompt
- F13=Session services
- F16=Select collections
- F17=Select tables

เซสชันแบบโต้ตอบประกอบด้วย:

- คำพารามิเตอร์ที่คุณระบุไว้สำหรับคำสั่ง STRSQL .
- ข้อความ SQL ที่คุณป้อนลงในเซสชันพร้อมกับข้อความโต้ตอบข้อความ SQL นั้น
- ค่าของพารามิเตอร์ใดๆ ที่คุณเปลี่ยนโดยใช้ฟังก์ชันเซอริวิสเซสชัน
- รายการที่คุณเลือก

SQL แบบโต้ตอบมี session-ID แบบเฉพาะที่ประกอบด้วย user ID ของคุณและ workstation station ID ปัจจุบัน. session-ID นี้อนุญาตให้ผู้ใช้จำนวนมากที่มี user ID เดียวกันเข้าใช้งาน SQL แบบโต้ตอบจากเวิร์กสเตชันได้มากกว่าหนึ่งเวิร์กสเตชันในเวลาเดียวกัน. นอกจากนี้, สามารถรันเซสชัน SQL แบบโต้ตอบได้มากกว่าหนึ่งเซสชันได้จากเวิร์กสเตชันเดียวกันในเวลาเดียวกันจาก user ID เดียวกัน.

หากมีเซสชัน SQL และเซสชันนั้นถูกป้อนซ้ำ, พารามิเตอร์ใดๆ ที่ระบุไว้บนคำสั่ง STRSQL ถูกละเลย. พารามิเตอร์จากเซสชัน SQL ที่มีอยู่ถูกใช้งาน.

ข้อมูลที่เกี่ยวข้อง

Start SQL Interactive Session (STRSQL) statement

การใช้ฟังก์ชัน entry คำสั่ง

ฟังก์ชัน entry คำสั่ง คือฟังก์ชันที่คุณป้อนเข้าไปครั้งแรก เมื่อเลือก SQL แบบตอบโต้. คุณจะกลับไปยัง entry คำสั่งหลังจากประมวลผลคำสั่ง SQL แบบโต้ตอบแต่ละอัน.

ในฟังก์ชัน entry คำสั่ง, คุณจะพิมพ์หรือพร้อมท์สำหรับคำสั่ง SQL ทั้งหมดและส่งคำสั่งนั้นไปเพื่อการประมวลผลโดยกดคีย์ Enter.

คำสั่งที่คุณพิมพ์บนบรรทัดรับคำสั่งอาจยาวหนึ่งบรรทัดหรือมากกว่า. คุณสามารถพิมพ์หมายเหตุที่วงเล็บไว้ (/ ** /) ใน SQL แบบโต้ตอบ. อย่างไรก็ตาม, คุณไม่ควรใช้หมายเหตุธรรมดา (นั่นคือ, หมายเหตุที่เริ่มต้นด้วย --) ใน SQL แบบโต้ตอบ เนื่องจากหมายเหตุเหล่านี้จะประกอบด้วยส่วนที่เหลือของคำสั่ง SQL ภายในหมายเหตุนั้น. เมื่อคำสั่งถูกประมวลผล, คำสั่งและข้อความผลลัพธ์จะถูกย้ายขึ้นไปด้านบนบนจอแสดงผล. หลังจากนั้นคุณสามารถป้อนคำสั่งอื่นได้.

หาก SQL พบข้อผิดพลาดด้านไวยากรณ์ของคำสั่งที่ป้อนเข้ามา ข้อความผลลัพธ์ (ข้อผิดพลาดด้านไวยากรณ์) จะถูกย้ายไปด้านบนบนจอแสดงผล. ในส่วนอินพุต, จะมีสำเนาคำสั่งพร้อมด้วยเคอร์เซอร์ที่วางอยู่ที่ข้อผิดพลาดทางไวยากรณ์. คุณสามารถวางเคอร์เซอร์ไว้บนข้อความ และกด F1=Help เพื่อดูข้อมูลเพิ่มเติมเกี่ยวกับข้อผิดพลาดได้.

คุณสามารถเลื่อนหน้าไปยังข้อความ (คำสั่ง), คำสั่ง, และข้อความ (แสดงผล) ก่อนหน้านี้ได้. ถ้าคุณกด F9=Retrieve ในขณะที่เคอร์เซอร์อยู่บนบรรทัดใส่คำสั่ง, คำสั่งก่อนหน้านี้อาจจะถูกคัดลอกไว้ในพื้นที่อินพุต. กด F9 อีกครั้งจะทำให้เกิดการย้อนขึ้นไปอีกหนึ่งคำสั่ง และคัดลอกคำสั่งนั้นไว้ในพื้นที่อินพุต. การกด F9 ต่อไปทำให้คุณย้อนขึ้นไปเรื่อยๆ ที่ละคำสั่งก่อนหน้าจนกระทั่งคุณพบคำสั่งที่คุณต้องการ. หากคุณต้องการเนื้อที่เพิ่มสำหรับพิมพ์คำสั่ง SQL, ให้เลื่อนจอแสดงผลลง.

การออกคำสั่ง (Prompting)

ฟังก์ชันพร้อมท์ช่วยให้คุณหาข้อมูลที่จำเป็นสำหรับไวยากรณ์ของข้อความที่คุณต้องการใช้. ฟังก์ชันพร้อมท์อาจถูกใช้ในโหมดการประมวลผลคำสั่งเหล่านี้: *RUN, *VLD, และ *SYN.

คุณมีสองอ็อปชันเมื่อใช้งานตัวพร้อมท์:

- พิมพ์กิริยาของข้อความก่อนกด F4=Prompt.

คำสั่งจะถูวิเคราะห์และ clause ที่สมบูรณ์จะถูกกรอกในจอแสดงผลพร้อมด้.

หากคุณพิมพ์ SELECT และกด F4=Prompt, จอแสดงผลต่อไปนี้จะปรากฏ:

Specify SELECT Statement

Type SELECT statement information. Press F4 for a list.

FROM tables _____
SELECT columns _____
WHERE conditions _____
GROUP BY columns _____
HAVING conditions _____
ORDER BY columns _____
FOR UPDATE OF columns _____

Bottom

Type choices, press Enter.

DISTINCT rows in result table N Y=Yes, N=No
UNION with another SELECT N Y=Yes, N=No
Specify additional options N Y=Yes, N=No

F3=Exit F4=Prompt F5=Refresh F6=Insert line F9=Specify subquery
F10=Copy line F12=Cancel F14=Delete line F15=Split line F24=More keys

- กด F4=Prompt ก่อนพิมพ์ข้อมูลใดๆ ลงบนจอแสดงผล Enter SQL Statements. คุณจะเห็นรายการข้อความ. รายการข้อความจะต่างกันไปและขึ้นอยู่กับโหมดการประมวลผลข้อความ SQL แบบโต้ตอบในปัจจุบัน. สำหรับโหมดการตรวจสอบไวยากรณ์ด้วยภาษาอื่นนอกเหนือจาก *NONE, รายการจะรวมข้อความ SQL ทั้งหมดเข้าไว้ด้วย. สำหรับโหมดรันพร้อมตรวจสอบความถูกต้องนั้น, เฉพาะข้อความที่ถูกรันใน SQL แบบโต้ตอบเท่านั้นจะถูกแสดง. คุณสามารถเลือกจำนวนของข้อความที่คุณต้องการใช้งานได้. ระบบจะถามข้อความที่คุณเลือก.

หากคุณกด F4=Prompt โดยไม่พิมพ์อะไร, จอแสดงผลจะแสดง:

เลือกข้อความ SQL

เลือกรายการใดรายการหนึ่งต่อไปนี้:

1. ALTER TABLE
2. CALL
3. COMMENT ON
4. COMMIT
5. CONNECT
6. CREATE ALIAS
7. CREATE COLLECTION
8. CREATE INDEX
9. CREATE PROCEDURE
10. CREATE TABLE
11. CREATE VIEW
12. DELETE
13. DISCONNECT
14. DROP ALIAS

รายการที่เลือก

—

F3=Exit F12=Cancel

หากคุณกด F21=Display Statement บนจอแสดงผลพร้อมต์, ตัวพร้อมต์จะแสดงผลข้อความ SQL ที่ถูกฟอร์แมตเหมือนกับที่กรอกไว้ที่ตัวพร้อมต์.

เมื่อกด Enter ภายในการออกคำสั่ง, ข้อความซึ่งถูกสร้างด้วยหน้าจอพร้อมต์จะถูกแทรกลงยังเซสชัน. หากโหมดการประมวลผลข้อความคือ *RUN, ข้อความจะถูกรัน. ตัวพร้อมต์ยังคงอยู่ในการควบคุมหากพบข้อผิดพลาด.

การตรวจสอบไวยากรณ์:

ไวยากรณ์ของคำสั่ง SQL จะถูกตรวจสอบ เมื่อใส่ไวยากรณ์ในตัวพร้อมต์.

ตัวพร้อมต์ไม่รับคำสั่งที่ผิดพลาดทางไวยากรณ์. คุณต้องแก้ไขไวยากรณ์ หรือลบส่วนของคำสั่งที่ไม่ถูกต้อง มิฉะนั้นจะไม่อนุญาตให้ออกคำสั่ง.

โหมดการประมวลผลคำสั่ง:

สามารถเลือกโหมดการประมวลผลคำสั่งบนจอแสดงผล Change Session Attributes.

ในโหมด *RUN (run) หรือ *VLD (validate), เฉพาะคำสั่งที่อนุญาตให้รันใน SQL แบบโต้ตอบที่สามารถถามได้. ในโหมด *SYN (syntax check), ข้อความ SQL ทั้งหมดได้รับอนุญาต. ที่จริงแล้ว คำสั่งไม่ได้ถูกรันในโหมด *SYN หรือ *VLD; เฉพาะไวยากรณ์และอ็อบเจกต์จะถูกตรวจสอบ.

เคียวรีย่อย:

สามารถเลือกเคียวรีย่อยบนจอแสดงผลใดๆ ที่มี WHERE หรือ HAVING clause.

หากต้องการดูจอแสดงผลเคียวรีย่อย, โปรดกด F9=Specify subquery เมื่อเคอร์เซอร์อยู่ที่แถวอินพุต WHERE หรือ HAVING. จอแสดงผลจะแสดงผลให้คุณพิมพ์ข้อมูลการเลือกแบบย่อย. หากเคอร์เซอร์อยู่ภายในวงเล็บของเคียวรีย่อย เมื่อกด F9, ข้อมูลของเคียวรีย่อยจะถูกแสดงในจอผลถัดไป. หากเคอร์เซอร์อยู่นอกวงเล็บของเคียวรีย่อย, จอแสดงผลถัดไปจะว่างเปล่า.

การเรียกทำงาน CREATE TABLE:

เมื่อมีพร้อมต์สำหรับ CREATE TABLE, คุณสามารถป้อน definition คอลัมน์ทีละค่าได้.

วางเคอร์เซอร์ของคุณในส่วน definition คอลัมน์ของจอแสดงผล, และกด F4=Prompt. จอแสดงผลที่มีเนื้อที่สำหรับการป้อนข้อมูลทั้งหมดสำหรับ definition ของหนึ่งคอลัมน์จะปรากฏขึ้นมา.

หากต้องการป้อนชื่อคอลัมน์ที่มีความยาวมากกว่า 18 อักขระ, ให้กด F20=Display entire name. หน้าต่างที่มีเนื้อที่พอสำหรับชื่อขนาด 30 อักขระจะปรากฏขึ้นมา.

คีย์แก้ไข, F6=Insert line, F10=Copy line, และ F14=Delete line, สามารถใช้งานเพื่อเพิ่ม และลบ entry ในรายการ definition คอลัมน์.

การป้อนข้อมูล DBCS:

กฎสำหรับการประมวลผลข้อมูล DBCS บนหลายๆ แถว เป็นกฎเดียวกันกับที่อยู่บนจอแสดงผล Enter SQL Statements และในตัวพร้อมต์ SQL.

แต่ละแถวจะมีหมายเลขอักขระบนและล่างเดียวกัน. เมื่อทำการประมวลผลสตริงข้อมูล DBCS ซึ่งต้องการแถวข้อมูลมากกว่าหนึ่งแถวสำหรับป้อน, อักขระพิเศษบนและล่างจะถูกลบออกไป. หากคอลัมน์สุดท้ายบนแถวข้อมูลมีอักขระบนและคอลัมน์แรกของแถวข้อมูลถัดไปมีอักขระล่าง, อักขระบนและล่างจะถูกลบออกไปโดยตัวพร้อมท์เมื่อแถวข้อมูลทั้งสองถูกแปลภาษา. หากสองคอลัมน์สุดท้ายของแถวข้อมูลมีอักขระบนที่ตามด้วยช่องว่าง และคอลัมน์แรกของแถวข้อมูลถัดไปมีอักขระบน, อักขระล่าง, ช่องว่าง, การเรียงอักขระล่างถูกลบออกไปเมื่อแถวข้อมูลถูกแปลภาษา. การลบนี้อนุญาตให้ข้อมูล DBCS ถูกอ่านเป็นสตริงอักขระต่อเนื่องได้.

ในตัวอย่าง, สมมติว่าป้อนเงื่อนไข WHERE ลงไป. อักขระด้านบน จะปรากฏขึ้นในส่วนสตริงเริ่มต้นและสิ้นสุดบนแต่ละแถวข้อมูลสองแถว.

Specify SELECT Statement

Type SELECT statement information. Press F4 for a list.

FROM tables TABLE1_____

SELECT columns *_____

WHERE conditions COL1 = '<AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQ>
<RRSS>'_____

GROUP BY columns _____

HAVING conditions _____

ORDER BY columns _____

FOR UPDATE OF columns _____

เมื่อกด Enter, สตริงอักขระจะถูกดึงรวมกันไว้, โดยจะลบอักขระเสริมด้านบนออก. คำสั่งจะมีลักษณะเช่นนี้บนจอแสดงผล Enter SQL Statements:

```
SELECT * FROM TABLE1 WHERE COL1 = '<AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSS>'
```

การใช้ฟังก์ชันรายการที่เลือก

ฟังก์ชันรายการที่เลือกจะมีความพร้อมใช้งานด้วยการกด F4 บนจอแสดงผลพร้อมท์บางจอ, หรือ F16, F17, หรือ F18 บนจอแสดงผล Enter SQL Statements.

หลังจากกดคีย์ฟังก์ชัน, คุณจะได้รับรายการฐานข้อมูลเชิงสัมพันธ์ที่ได้รับสิทธิ์, แบบแผน, ตาราง, มุมมอง, alias, คอลัมน์, ข้อจำกัด, โพรซีเจอร์, พารามิเตอร์, หรือแฟกต์ที่ได้รับสิทธิ์ให้เลือก. หากคุณร้องขอรายการตาราง, แต่คุณไม่ได้เลือกแบบแผนไว้ก่อน, คุณจะถูกลำโพงให้เลือกแบบแผนก่อน.

ในรายการ, คุณสามารถเลือกไอเท็มได้หนึ่งไอเท็มหรือมากกว่า, โดยระบุลำดับที่คุณต้องการให้ปรากฏในข้อความด้วยตัวเลข. เมื่อฟังก์ชันรายการออกจากการทำงาน, รายการที่คุณเลือกจะถูกแทรกลงที่ตำแหน่งที่วางเคอร์เซอร์บนจอแสดงผลที่คุณออกมา.

โปรดเลือกรายการที่คุณสนใจเป็นหลักเสมอ. ตัวอย่างเช่น, หากคุณต้องการรายการคอลัมน์, แต่คุณเชื่อว่าคอลัมน์ที่คุณต้องการอยู่ในตารางที่ไม่ได้เลือกในปัจจุบัน, ให้กด F18=Select columns. แล้ว, จากรายการคอลัมน์, ให้กด F17 เพื่อเปลี่ยนตาราง. หากรายการตารางถูกเลือกเป็นอันดับแรก, ชื่อตารางจะถูกแทรกเข้าไปยังข้อความของคุณ. คุณจะไม่มีตัวเลือกสำหรับการเลือกคอลัมน์.

คุณสามารถร้องขอรายการเมื่อใดก็ได้ขณะพิมพ์ข้อความ SQL บนจอแสดงผล Enter SQL Statements. รายการที่คุณเลือกจากรายการจะถูกแทรกลงบนจอแสดงผล Enter SQL Statements. ข้อความจะถูกแทรกลงในที่ตำแหน่งที่เคอร์เซอร์วางอยู่ตามลำดับหมายเลขที่คุณระบุไว้บนจอแสดงผลรายการ. แม้ว่าข้อมูลของรายการที่คุณเลือกจะถูกเพิ่มเข้าไปแล้ว, คุณต้องพิมพ์คีย์เวิร์ดสำหรับข้อความด้วย.

ฟังก์ชันรายการพยายามจัดหาคุณสมบัติที่จำเป็นสำหรับคอลัมน์, ตาราง, และ SQL แพ็กเกจที่เลือก. อย่างไรก็ตาม, บางครั้งฟังก์ชันรายการจะไม่สามารถกำหนดจุดประสงค์ของข้อความ SQL ได้. คุณต้องตรวจสอบข้อความ SQL และตรวจสอบว่าคอลัมน์, ตาราง, และ SQL แพ็กเกจที่เลือกมีคุณสมบัติถูกต้อง.

ตัวอย่าง: การใช้ฟังก์ชันรายการที่เลือก:

ตัวอย่างต่อไปนี้จะแสดงวิธีการใช้ฟังก์ชันรายการเพื่อสร้างข้อความ SELECT ให้คุณ.

สมมติว่า คุณมี:

- เพียงแค่ป้อน SQL แบบโต้ตอบโดยการพิมพ์ STRSQL บนบรรทัดรับคำสั่ง i5/OS.
- ไม่ต้องเลือกรายการหรือป้อน entry.
- เลือก *SQL เพื่อดูหลักการตั้งชื่อ.

หมายเหตุ: ตัวอย่างจะแสดงรายการที่ไม่อยู่บนเซิร์ฟเวอร์ของคุณ. รายการเหล่านั้นจะถูกใช้เพื่อเป็นตัวอย่างเท่านั้น.

เริ่มต้นใช้ SQL statements:

1. พิมพ์ SELECT บนแถว entry แรกของข้อความ.
2. พิมพ์ FROM บนแถว entry ที่สองของข้อความ.
3. ปลดเคอร์เซอร์ไว้หลังคำว่า FROM.

```
Enter SQL Statements

Type SQL statement, press Enter.
==> SELECT
    FROM _
```

4. กด F17=Select tables เพื่อดูรายการตาราง, เพราะคุณต้องใส่ชื่อตารางตามหลัง FROM. แทนที่รายการตารางจะปรากฏตามที่คาดไว้, รายการคอลเล็คชันจะแสดง (จอแสดงผลคอลเล็คชัน Select and Sequence). คุณเพิ่มป้อนเซสชัน SQL และไม่ได้เลือกแบบแผนที่จะทำงานด้วย
5. พิมพ์ 1 ในคอลัมน์ Seq ถัดจากแบบแผน YOURCOLL2.

Select and Sequence Collections

พิมพ์หมายเลขลำดับ (1-999) เพื่อเลือกคอลเล็กชัน, กด Enter.

ลำดับ	คอลเล็กชัน	ประเภท	ข้อความ
	YOURCOLL1	SYS	ผลประโยชน์ของบริษัท
1	YOURCOLL2	SYS	ข้อมูลส่วนตัวพนักงาน
	YOURCOLL3	SYS	การแบ่งประเภทงาน/ข้อกำหนดของงาน
	YOURCOLL4	SYS	การประกันภัยบริษัท

6. กด Enter.

แสดงผล Select and Sequence Tables จะปรากฏขึ้นมา, โดยจะแสดงตารางที่อยู่ในแบบแผน YOURCOLL2.

7. พิมพ์ 1 ในคอลัมน์ Seq ถัดจากตาราง PEOPLE.

Select and Sequence Tables

พิมพ์หมายเลขลำดับ (1-999) เพื่อเลือกตาราง, กด Enter.

ลำดับ	ตาราง	คอลเล็กชัน	ประเภท	ข้อความ
	EMPLCO	YOURCOLL2	TAB	ข้อมูลบริษัทของพนักงาน
1	PEOPLE	YOURCOLL2	TAB	ข้อมูลส่วนบุคคลของพนักงาน
	EMPLEXP	YOURCOLL2	TAB	ประสบการณ์พนักงาน
	EMPLEVL	YOURCOLL2	TAB	รายงานการประเมินผลพนักงาน
	EMPLBEN	YOURCOLL2	TAB	ระเบียบข้อมูลผลประโยชน์พนักงาน
	EMPLMED	YOURCOLL2	TAB	ระเบียบข้อมูลทางการแพทย์ของพนักงาน
	EMPLINVST	YOURCOLL2	TAB	ระเบียบข้อมูลการลงทุนของพนักงาน

8. กด Enter.

แสดงผล Enter SQL Statements จะปรากฏขึ้นอีกครั้งพร้อมกับชื่อตาราง, YOURCOLL2.PEOPLE, จะถูกแทรกลงหลังจาก FROM. ชื่อตารางจะถูกคัดเลือกโดยชื่อแบบแผนในหลักการตั้งชื่อ *SQL.

Enter SQL Statements

Type SQL statement, press Enter.

```
===> SELECT
      FROM YOURCOLL2.PEOPLE _
```

9. วางเคอร์เซอร์หลังจาก SELECT.

10. กด F18=Select columns เพื่อดูรายการคอลัมน์, เพราะคุณต้องให้ชื่อคอลัมน์ตามหลัง SELECT.

แสดงผล Select and Sequence Columns จะปรากฏขึ้นมา, โดยจะแสดงตารางที่อยู่ในตาราง PEOPLE.

11. พิมพ์ 2 ในคอลัมน์ Seq ถัดจากคอลัมน์ NAME.

12. พิมพ์ 1 ในคอลัมน์ Seq ถัดจากคอลัมน์ SOCSEC.

Select and Sequence Columns

พิมพ์หมายเลขลำดับ (1-999) เพื่อเลือกคอลัมน์, กด Enter.

ลำดับ	คอลัมน์	ตาราง	ประเภท	ดิจิทัล	ความยาว
2	NAME	PEOPLE	CHARACTER		6
	EMPLNO	PEOPLE	CHARACTER		30
1	SOCSEC	PEOPLE	CHARACTER		11
	STRADDR	PEOPLE	CHARACTER		30
	CITY	PEOPLE	CHARACTER		20
	ZIP	PEOPLE	CHARACTER		9
	PHONE	PEOPLE	CHARACTER		20

13. กด Enter.

แสดงผล Enter SQL Statements จะปรากฏขึ้นมาพร้อมด้วย SOCSEC, NAME ซึ่งจะปรากฏหลังคำว่า SELECT.

Enter SQL Statements

```
Type SQL statement, press Enter.  
====> SELECT SOCSEC, NAME  
        FROM YOURCOLL2.PEOPLE
```

14. กด Enter.

ข้อความที่คุณสร้างจะถูกรันในขณะนี้.

เมื่อคุณใช้งานฟังก์ชันรายการ, ค่าที่คุณเลือกไว้จะยังใช้งานได้อยู่นกว่าคุณจะเปลี่ยนมันหรือจนกว่าคุณจะเปลี่ยนรายการของแบบแผนบนจอแสดงผล Change Session Attributes.

รายละเอียดเซอร์วิสเซชัน

แสดงผล SQL Session Services แบบโต้ตอบจะถูกร้องขอโดยการกด F13 บนจอแสดงผล Enter SQL Statements.

จากจอแสดงผลนี้ คุณสามารถเปลี่ยนแอตทริบิวต์เซชันและสั่งพิมพ์, ลบ, หรือบันทึกเซชันให้กับซอร์สไฟล์.

Option 1 (เปลี่ยนแอตทริบิวต์เซชัน) จะแสดงจอแสดงผล Change Session Attributes, ซึ่งอนุญาตให้คุณเลือกค่าปัจจุบันซึ่งยังใช้งานได้สำหรับเซชัน SQL แบบโต้ตอบ. อีพจน์ที่แสดงอยู่บนจอแสดงผลนี้จะเปลี่ยนไปตามอีพจน์การประมวลผลของข้อความที่เลือก.

แอตทริบิวต์เซชันต่อไปนี้ อาจถูกเปลี่ยนได้:

- Commitment control แอตทริบิวต์.
- การควบคุมการประมวลผลข้อความ.
- อุปกรณ์เอาต์พุต SELECT.
- รายการแบบแผน.
- ประเภทรายการสำหรับเลือกระบบของคุณ และ SQL อ็อบเจกต์ทั้งหมด, หรือเฉพาะ SQL อ็อบเจกต์.
- ข้อมูลจะรีเฟรชอีพจน์เมื่อแสดงผลข้อมูล.

- อีอพชันอนุญาตให้ทำสำเนาข้อมูล.
- อีอพชันการตั้งชื่อ.
- ภาษาโปรแกรม.
- รูปแบบวันที่.
- รูปแบบเวลา.
- ตัวแยกวันที่.
- ตัวแยกเวลา.
- การแทนค่าจุดทศนิยม.
- อักขระคั่นสตริง SQL .
- ลำดับการจัดเรียง.
- ตัวระบุภาษา (language identifier) .
- กฎ SQL.
- อีอพชันรหัสผ่าน CONNECT.

อีอพชัน 2 (Print current session) จะเข้าใช้งานจอแสดงผล Change Printer , ซึ่งจะให้คูปพิมพ์เซสชันปัจจุบันได้ทันทีและทำงานต่อไป. คุณจะถูกลถามให้ใส่ข้อมูลเครื่องพิมพ์. ข้อความ SQL ทั้งหมดที่คุณป้อนและข้อความที่แสดงผลจะถูกสั่งพิมพ์ออกมาเหมือนตอนที่ข้อความเหล่านั้นปรากฏบนจอแสดงผล Enter SQL Statements.

อีอพชัน 3 (Remove all entries from current session) จะให้คุณลบข้อความ SQL และข้อความจากจอแสดงผล Enter SQL Statements และประวัติเซสชัน. คุณจะถูกลถามเพื่อความแน่ใจว่าคุณต้องการลบข้อมูลนั้นจริงๆ.

อีอพชัน 4 (Save session in source file) จะเข้าใช้งานจอแสดงผล Change Source File , ซึ่งให้คุณบันทึกเซสชันในซอร์สไฟล์. คุณจะถูกลถามชื่อซอร์สไฟล์. ฟังก์ชันนี้จะให้คุณฝังซอร์สไฟล์ลงในโปรแกรมภาษาโฮสต์โดยการใช้ source entry utility (SEU).

หมายเหตุ: อีอพชัน 4 จะให้คุณฝังข้อความ SQL ที่เป็นต้นแบบในโปรแกรมภาษาชั้นสูง (HLL) ที่ใช้งาน SQL. ซอร์สไฟล์ที่ถูกรสร้างโดยอีอพชัน 4 อาจถูกแก้ไข และใช้เป็นซอร์สไฟล์อินพุตสำหรับคำสั่ง Run SQL Statements (RUNSQLSTM).

การออกจาก SQL แบบโต้ตอบ

การกด F3=Exit บนจอแสดงผล Enter SQL Statements จะอนุญาตให้คุณออกจากสภาวะแวดล้อม SQL แบบโต้ตอบ. คุณมีหลายอีอพชันสำหรับการออก.

- บันทึกและออกจากเซสชัน. ปลด SQL แบบโต้ตอบไว้. เซสชันปัจจุบันของคุณจะถูกบันทึกและใช้งานในครั้งต่อไปที่คุณเริ่มใช้งาน SQL แบบโต้ตอบ.
- ออกจากเซสชันโดยไม่ต้องบันทึกเซสชัน. ปลด SQL แบบโต้ตอบไว้โดยไม่ต้องบันทึกเซสชันของคุณ.
- เรียกเซสชันกลับสู่การทำงาน. ยังอยู่ใน SQL แบบโต้ตอบและกลับสู่จอแสดงผล Enter SQL Statements. พารามิเตอร์เซสชันปัจจุบันยังคงใช้งานได้อยู่.
- บันทึกเซสชันในซอร์สไฟล์. บันทึกเซสชันปัจจุบันในซอร์สไฟล์ จอแสดงผล Change Source File จะปรากฏขึ้นมาเพื่อให้คุณเลือกที่จะบันทึกเซสชัน. คุณไม่สามารถเรียกคืนและทำงานกับเซสชันนี้ได้อีกใน SQL แบบโต้ตอบ.

หมายเหตุ:

1. อีอ็อปชัน 4 จะให้คุณฝังข้อความ SQL ต้นแบบในโปรแกรมภาษาชั้นสูง (HLL) ที่ใช้งาน SQL. ใช้งาน source entry utility (SEU) เพื่อทำสำเนาข้อความลงยังโปรแกรมของคุณ. สามารถแก้ไขซอร์สไฟล์ และถูกใช้เป็นซอร์สไฟล์อินพุตสำหรับคำสั่ง Run SQL Statements (RUNSQLSTM).
2. หากมีการเปลี่ยนแปลงของแถวและมีการล็อกงานนี้และคุณพยายามจะออกจาก SQL แบบโต้ตอบ, จะมีข้อความแจ้งเตือนปรากฏขึ้นมา.

การใช้เซสชัน SQL ที่มีอยู่

หากคุณบันทึกเซสชัน SQL แบบโต้ตอบเซสชันโดยการใช้อีอ็อปชัน 1 (เซสชัน Save and exit) บนจอแสดงผล Exit Interactive SQL, คุณอาจต้องเรียกเซสชันนั้นกลับสู่การทำงานใหม่ที่เวิร์กสเตชันใดก็ได้.

อย่างไรก็ตาม, หากคุณใช้อีอ็อปชัน 1 เพื่อบันทึกเซสชันสองเซสชันหรือมากกว่าบนเวิร์กสเตชันที่ต่างกัน, SQL แบบโต้ตอบจะพยายามเรียกเซสชันที่ตรงกับเวิร์กสเตชันของคุณให้กลับมาเริ่มต้นทำงานใหม่ก่อน. หากไม่มีเซสชันใดตรงกัน, SQL แบบโต้ตอบจะเพิ่มขอบเขตของการค้นหาในครอบครัวถึงเซสชันทั้งหมดที่เป็นของ user ID คุณ. หากไม่มีเซสชันสำหรับ user ID ของคุณ, ระบบจะสร้างเซสชันใหม่สำหรับ user ID ของคุณและเวิร์กสเตชันปัจจุบัน.

ตัวอย่างเช่น, คุณได้บันทึกเซสชันไว้บนเวิร์กสเตชัน 1 และบันทึกอีกเซสชันหนึ่งไว้บนเวิร์กสเตชัน 2 และคุณยังคงทำงานอยู่บนเวิร์กสเตชัน 1. SQL แบบโต้ตอบจะพยายามเรียกเซสชันที่บันทึกไว้สำหรับเวิร์กสเตชัน 1 กลับมาทำงานใหม่. หากเซสชันนั้นกำลังอยู่ระหว่างใช้งาน, SQL แบบโต้ตอบจะพยายามเรียกเซสชันที่บันทึกไว้สำหรับเวิร์กสเตชัน 2. หากเซสชันนั้นอยู่ระหว่างใช้งานเช่นกัน, ระบบจะสร้างเซสชันที่สองสำหรับเวิร์กสเตชัน 1.

อย่างไรก็ตาม, สมมติว่าคุณกำลังทำงานที่เวิร์กสเตชัน 3 และต้องการใช้งานเซสชัน ISQL ที่เชื่อมโยงกับเวิร์กสเตชัน 2. คุณอาจต้องลบเซสชันจากเวิร์กสเตชัน 1 เป็นลำดับแรกโดยใช้อีอ็อปชัน 2 (Exit without saving session) บนจอแสดงผล Exit Interactive SQL.

การกู้คืนเซสชัน SQL

หากเซสชัน SQL ก่อนหน้านี้นหยุดทำงานแบบผิดปกติ, SQL แบบโต้ตอบจะปรากฏจอแสดงผล Recover SQL Session ตอนเริ่มต้นทำงานของเซสชันถัดไป (เมื่อป้อนคำสั่ง STRSQL ถัดไป).

จากจอแสดงผลนี้, คุณสามารถเลือกเพื่อทำหนึ่งในสิ่งต่อไปนี้.

- กู้คืนเซสชันเดิมด้วยการเลือกอีอ็อปชัน 1 (พยายามเรียกเซสชัน SQL ที่มีอยู่เดิมกลับสู่การทำงานใหม่).
- ลบเซสชันเดิมและเริ่มต้นเซสชันใหม่ด้วยการเลือกอีอ็อปชัน 2 (ลบเซสชัน SQL ที่มีอยู่เดิมและเรียกใช้งานเซสชันใหม่).

หากคุณเลือกที่จะลบเซสชันเดิมออกและทำงานกับเซสชันใหม่ต่อไป, พารามิเตอร์ที่คุณระบุไว้เมื่อคุณป้อน STRSQL จะถูกใช้งาน. หากคุณเลือกที่จะกู้คืนเซสชันเดิม, หรือกำลังป้อนเซสชันที่บันทึกไว้ก่อนหน้า, พารามิเตอร์ที่คุณระบุไว้เมื่อคุณป้อน STRSQL จะถูกละเลยและพารามิเตอร์จากเซสชันเดิมจะถูกใช้งาน. ข้อความจะถูกส่งกลับเพื่อระบุว่าพารามิเตอร์ใดถูกเปลี่ยนไปจากค่าที่ระบุไว้ของค่าเซสชันเดิม.

การเข้าใช้งานฐานข้อมูลแบบรีโมตด้วย SQL แบบโต้ตอบ

ใน SQL แบบโต้ตอบ, คุณสามารถสื่อสารกับฐานข้อมูลเชิงสัมพันธ์แบบรีโมตด้วยการใช้ข้อความ SQL CONNECT. SQL แบบโต้ตอบจะใช้ซีแมนทิกส์ CONNECT (Type 2) (หน่วยงานแบบกระจาย) สำหรับคำสั่ง CONNECT.

SQL แบบโต้ตอบเชื่อมต่อกับ RDB แบบโลคัลเมื่อเริ่มต้นใช้งานเซสชัน SQL. เมื่อคำสั่ง CONNECT เสร็จสมบูรณ์, ข้อความจะแสดงการเชื่อมต่อของฐานข้อมูลเชิงสัมพันธ์ซึ่งถูกสร้างขึ้น. หากไม่มีการระบุ COMMIT(*NONE) ในการเรียกใช้งานเซส

ชั้นใหม่, หรือหากการเรียกคืนเซสชันที่บันทึกไว้และระดับ commit ที่บันทึกไว้กับเซสชันที่ไม่ใช่ *NONE, การเชื่อมต่อจะถูกลง
ทะเบียนด้วย commitment control. การเชื่อมต่อโดยนัยและการลงทะเบียน commitment control ที่เป็นไปได้ อาจส่งผลต่อการ
เชื่อมต่อในลำดับต่อมาที่ฐานข้อมูลแบบรีโมต. แนะนำว่าก่อนการเชื่อมต่อกับระบบรีโมต:

- เมื่อทำการเชื่อมต่อกับแอปพลิเคชันเซิร์ฟเวอร์ที่ไม่รองรับหน่วยการทำงานแบบกระจาย, จะต้องมีการใช้คำสั่ง RELEASE ALL นำหน้า COMMIT เพื่อสิ้นสุดการเชื่อมต่อครั้งก่อนหน้า, รวมถึงการเชื่อมต่อโดยนัยกับโลคัล.
- เมื่อทำการเชื่อมต่อกับแอปพลิเคชันเซิร์ฟเวอร์แบบไม่มี DB2 UDB for iSeries, จะต้องมีการใช้คำสั่ง RELEASE ALL นำหน้า COMMIT เพื่อสิ้นสุดการเชื่อมต่อครั้งก่อนหน้า, รวมถึงการเชื่อมต่อโดยนัย, และเปลี่ยนระดับ commitment control ให้เป็น *CHG เป็นอย่างน้อย.

เมื่อคุณทำการเชื่อมต่อกับแอปพลิเคชันเซิร์ฟเวอร์แบบไม่มี DB2 UDB for iSeries, เซสชันแอ็ดทริบิวต์บางตัวถูกเปลี่ยนไป
เป็นแอ็ดทริบิวต์ ซึ่งถูกรองรับโดยแอปพลิเคชันเซิร์ฟเวอร์นั้น. ตารางต่อไปนี้จะแสดงแอ็ดทริบิวต์ที่เปลี่ยนไป.

ตารางที่ 49. ตารางค่า

เซสชันแอ็ดทริบิวต์	ค่าต้นฉบับ	ค่าใหม่
รูปแบบวันที่	*YMD *DMY *MDY *JUL	*ISO *EUR *USA *USA
รูปแบบเวลา	*HMS ด้วยตัวแยก : *HMS ด้วยตัวแยกอื่นๆ	*JIS *EUR
Commitment Control	*CHG, *NONE *ALL	*CS Repeatable Read
หลักการตั้งชื่อ	*SYS	*SQL
อนุญาตให้ทำสำเนาข้อมูล	*NO, *YES	*OPTIMIZE
รีเฟรชข้อมูล	*ALWAYS	*FORWARD
จุดทศนิยม	*SYSVAL	*PERIOD
เรียงลำดับ	ค่าใดๆ ที่นอกเหนือจาก *HEX	*HEX

หมายเหตุ:

1. หากมีการเชื่อมต่อกับเซิร์ฟเวอร์ที่กำลังรันรีลีสก่อนหน้า Version 2 Release 3, ค่าการเรียงลำดับจะเปลี่ยนไปเป็น *HEX.

2. เมื่อทำการเชื่อมต่อกับแอสพลีเคชันเซิร์ฟเวอร์ DB2/2 หรือ DB2/6000, วันที่และรูปแบบเวลาที่ระบุไว้ต้องเป็นรูปแบบเดียวกัน.

หลังจากเชื่อมต่อสมบูรณ์แล้ว, ข้อความจะถูกส่งไปเพื่อระบุว่าเซสชันแอ็ททริบิวต์ถูกเปลี่ยนไป. สามารถแสดงผลเซสชันแอ็ททริบิวต์ที่เปลี่ยนไปด้วยการใช้จอแสดงผลเซอริสเซสชัน. ผนวกรัน SQL แบบโต้ตอบ, จะไม่สามารถสร้างการเชื่อมต่ออื่นสำหรับ activation group ที่เป็นค่าดีฟอลต์.

เมื่อเชื่อมต่อกับระบบรีโมตด้วย SQL แบบโต้ตอบ, โหมดการประมวลผลข้อความเฉพาะไวยากรณ์จะตรวจสอบไวยากรณ์ของข้อความกับไวยากรณ์ที่ระบบโลคัลสนับสนุน ไม่ใช่ไวยากรณ์ที่ระบบรีโมตสนับสนุน. อย่างคล้ายคลึงกัน, ตัวพร้อมท์ SQL และตัวสนับสนุนรายการจะใช้ไวยากรณ์ข้อความและหลักการตั้งชื่อซึ่งสนับสนุนโดยระบบโลคัล. อย่างไรก็ตาม, ข้อความจะถูกรัน, บนระบบรีโมต. เนื่องจากมีความแตกต่างในระดับของการสนับสนุน SQL ระหว่างทั้งสองระบบ, จึงอาจพบข้อผิดพลาดทางไวยากรณ์ในข้อความบนระบบรีโมตขณะอยู่ในรันไทม์.

รายการแบบแผนและตารางจะมีอยู่เมื่อคุณถูกเชื่อมต่อกับฐานข้อมูลเชิงสัมพันธ์แบบโลคัล. รายการคอลัมน์มีอยู่เฉพาะเมื่อคุณถูกเชื่อมต่อกับตัวจัดการฐานข้อมูลเชิงสัมพันธ์ที่สนับสนุนข้อความ DESCRIBE TABLE.

เมื่อคุณออกจาก SQL แบบโต้ตอบด้วยการเชื่อมต่อที่มีการเปลี่ยนแปลงค้างอยู่ในการเชื่อมต่อหรือการเชื่อมต่อนั้นใช้การสนทนา (ระหว่างโปรแกรม) แบบป้องกัน, การเชื่อมต่อจะยังคงทำงานอยู่ต่อไป. หาก你不ทำงานอื่นๆ ขณะเชื่อมต่อ, การเชื่อมต่อจะสิ้นสุดในระหว่างการดำเนินงาน COMMIT หรือ ROLLBACK ถัดไป. คุณสามารถยุติการเชื่อมต่อได้โดยการใช้งาน RELEASE ALL และ COMMIT ก่อนออกจาก SQL แบบโต้ตอบ.

การใช้ SQL แบบโต้ตอบสำหรับการเข้าใช้งานแบบรีโมตกับแอสพลีเคชันเซิร์ฟเวอร์แบบไม่มี DB2 UDB for iSeries อาจต้องการการตั้งค่าบางอย่าง.

หมายเหตุ: ในเอาต์พุตของการติดตามการสื่อสาร, อาจมีการอ้างอิงถึงข้อความ 'CREATE TABLE XXX'. วิธีนี้ใช้เพื่อกำหนดการมีอยู่ของแพ็กเกจ;เป็นส่วนหนึ่งของการประมวลผลแบบปกติ, และอาจถูกละเลยได้.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การจำแนกประเภทของการเชื่อมต่อ” ในหน้า 323

เมื่อการเชื่อมต่อแบบรีโมตเกิดขึ้น ระบบจะทำการเชื่อมต่อระบบเครือข่ายแบบป้องกันหรือแบบไม่ได้ป้องกันอย่างใดอย่างหนึ่ง.

โปรแกรมมิ่งฐานข้อมูลแบบกระจาย

การใช้ตัวประมวลผลคำสั่ง SQL

ตัวประมวลผลคำสั่ง SQL ยอมให้คำสั่ง SQL ทำงานได้จากซอร์ส เมมเบอร์. คำสั่งในรายการต้นฉบับย่อสามารถรันซ้ำ, หรือเปลี่ยนได้, โดยไม่ต้องคอมไพล์ต้นฉบับ. ซึ่งทำให้การตั้งค่าสถานะแวดล้อมฐานข้อมูลง่ายตายขึ้น.

ตัวประมวลผลคำสั่ง SQL จะใช้งานได้โดยการใช้คำสั่ง Run SQL Statements (RUNSQLSTM).

คำสั่งที่ใช้ได้กับตัวประมวลผลคำสั่ง SQL ได้แก่:

- ALTER SEQUENCE
- ALTER TABLE
- CALL

- COMMENT ON
- COMMIT
- CREATE ALIAS
- CREATE DISTINCT TYPE
- CREATE FUNCTION
- CREATE INDEX
- CREATE PROCEDURE
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- DECLARE GLOBAL TEMPORARY TABLE
- DELETE
- DROP
- GRANT
- INSERT
- LABEL ON
- LOCK TABLE
- REFRESH TABLE
- RELEASE SAVEPOINT
- RENAME
- REVOKE
- ROLLBACK
- SAVEPOINT
- | • SET CURRENT DEGREE
- | • SET ENCRYPTION PASSWORD
- SET PATH
- SET SCHEMA
- SET TRANSACTION
- UPDATE

ในรายการต้นฉบับย่อ, คำสั่งลงท้ายด้วยเซมิโคลอนและไม่ขึ้นต้น ด้วย EXEC SQL. หากความยาวเรีกคอร์ดของรายการต้นฉบับย่อเกินกว่า 80, ระบบจะอ่านได้เฉพาะอักขระ 80 ตัวแรกเท่านั้น. หมายเหตุใน เมมเบอร์ต้นทางอาจเป็นหมายเหตุแบบแถวหรือหมายเหตุแบบบล็อก. หมายเหตุแบบแถวเริ่มต้นด้วยยัติภังค์คู่ (--) และสิ้นสุดที่ท้ายแถว. หมายเหตุแบบบล็อกเริ่ม

ต้นด้วย /* และมีต่อไปหลายแถวจนกว่าจะถึง */ ถัดไป. หมายเหตุแบบบล็อกสามารถซ่อนภายในได้. เฉพาะคำสั่ง SQL และ หมายเหตุเท่านั้นที่มีได้ในไฟล์ต้นฉบับ. การแสดงรายการเอาต์พุตและข้อความที่เป็นผลลัพธ์สำหรับคำสั่ง SQL จะถูกส่งไปยัง ไฟล์พิมพ์. ไฟล์พิมพ์ที่เป็นดีฟอลต์ คือ QSYSPRT.

ในการดำเนินการตรวจสอบซิงแทกซ์เฉพาะกับคำสั่งทั้งหมด ในรายการต้นฉบับย่อย, โพรเซสซอร์พารามิเตอร์ PROCESS (*SYN) ในคำสั่ง RUNSQLSTM.

ข้อมูลที่เกี่ยวข้อง

คำสั่ง Run SQL Statement (RUNSQLSTM)

การรันคำสั่งหลังเกิดข้อผิดพลาด

เมื่อคำสั่งเกิดข้อผิดพลาดที่มีค่าความรุนแรงสูงกว่าค่าที่ระบุในพารามิเตอร์ระดับความผิดพลาด (ERRLVL) ของคำสั่ง RUNSQLSTM, แสดงว่าคำสั่งล้มเหลว.

- | คำสั่งที่เหลือในต้นฉบับจะถูกวิเคราะห์ค่าเพื่อตรวจสอบข้อผิดพลาดของไวยากรณ์, และจะไม่ถูกรัน. ข้อผิดพลาดของ SQL
- | ส่วนใหญ่มีค่าความรุนแรงระดับ 30. หากคุณต้องการดำเนินการประมวลผลต่อ หลังจากคำสั่ง SQL ล้มเหลว, ให้ตั้งค่าพารามิเตอร์ ERRLVL ของคำสั่ง RUNSQLSTM เป็น 30 หรือสูงกว่า. คำสั่ง DROP จะออกข้อผิดพลาดของความรุนแรงระดับ 20
- | หากไม่พบข้อผิดพลาดที่จับ. การตั้งค่าพารามิเตอร์ ERRLVL ให้มีค่าเป็น 20 จะอนุญาตให้คุณข้ามข้อผิดพลาดของคำสั่ง
- | DROP ในขณะที่ไม่อนุญาตให้การประมวลผลดำเนินการต่อ หากเป็นข้อผิดพลาดที่มีความรุนแรงสูงกว่า.

commitment control ในตัวประมวลผลคำสั่ง SQL

ระดับ commitment-control ถูกระบุในคำสั่ง RUNSQLSTM.

หากมีการระบุ ระดับ commitment-control อื่นที่ไม่ใช่ *NONE, คำสั่ง SQL จะรันภายใต้ commitment control. หากคำสั่งทั้งหมดดำเนินการสำเร็จ, COMMIT จะดำเนินการเมื่อตัวประมวลผลคำสั่ง SQL เสร็จสิ้น. มิฉะนั้น, จะดำเนินการ ROLLBACK. ระบบจะถือว่าคำสั่งสำเร็จหากค่าความรุนแรงของโค้ดที่ได้ต่ำกว่าหรือเท่ากับค่าที่ระบุในพารามิเตอร์ ERRLVL ของคำสั่ง RUNSQLSTM.

คำสั่ง SET TRANSACTION สามารถใช้ภายในรายการต้นฉบับย่อยเพื่อแทนที่ระดับ commitment control ที่ระบุในคำสั่ง RUNSQLSTM.

หมายเหตุ: งานจะต้องอยู่ที่ยูนิคของขอบเขตงานเพื่อใช้ตัวประมวลผลคำสั่ง SQL กับ commitment control.

การแสดงรายการต้นฉบับย่อยสำหรับตัวประมวลผลคำสั่ง SQL

ตัวอย่างต่อไปนี้แสดงรายละเอียดของรายการต้นฉบับย่อยสำหรับตัวประมวลผลคำสั่ง SQL.

หมายเหตุ: ด้วยการใช้โค้ดตัวอย่าง, คุณตกลงในเงื่อนไขของ “ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ” ในหน้า


```

5722SS1 V5R4M0 060210      Run SQL Statements      SCHEMA      02/10/06 15:35:18  Page  1
Source file.....CORPDATA/SRC
Member.....SCHEMA
Commit.....*NONE
Naming.....*SYS
Generation level.....10
Date format.....*JOB
Date separator.....*JOB
Time format.....*HMS
Time separator .....*JOB
Default Collection.....*NONE
IBM SQL flagging.....*NOFLAG
ANS flagging.....*NONE
Decimal point.....*JOB
Sort Sequence.....*JOB
Language ID.....*JOB
Printer file.....*LIBL/QSYSPRT
Source file CCSID.....65535
Job CCSID.....0
Statement processing.....*RUN
Allow copy of data.....*OPTIMIZE
Allow blocking.....*READ
SQL rules.....*DB2
Decimal result options:
  Maximum precision.....31
  Maximum scale.....31
  Minimum divide scale....0
Source member changed on 04/01/98 11:54:10

```

รูปที่ 1. รายการ QSYSPRT listing สำหรับตัวประมวลผลคำสั่ง SQL

```

5722SS1 V5R4M0 060210      Run SQL Statements      SCHEMA      02/10/06 15:35:18  Page  2
Record *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8  SEQNBR Last change
1
2  DROP COLLECTION DEPT;
3  DROP COLLECTION MANAGER;
4
5  CREATE SCHEMA DEPT
6      CREATE TABLE EMP (EMPNAME CHAR(50), EMPNBR INT)
7      -- EMP will be created in collection DEPT
8      CREATE INDEX EMPIND ON EMP(EMPNBR)
9      -- EMPIND will be created in DEPT
10     GRANT SELECT ON EMP TO PUBLIC; -- grant authority
11
12     INSERT INTO DEPT/EMP VALUES('JOHN SMITH', 1234);
13     /* table must be qualified since no
14     longer in the schema */
15
16     CREATE SCHEMA AUTHORIZATION MANAGER
17     -- this schema will use MANAGER's
18     -- user profile
19     CREATE TABLE EMP_SALARY (EMPNBR INT, SALARY DECIMAL(7,2),
20     LEVEL CHAR(10))
21     CREATE VIEW LEVEL AS SELECT EMPNBR, LEVEL
22     FROM EMP_SALARY
23     CREATE INDEX SALARYIND ON EMP_SALARY(EMPNBR,SALARY)
24
25     GRANT ALL ON LEVEL TO JONES GRANT SELECT ON EMP_SALARY TO CLERK
26     -- Two statements can be on the same line
*** ** E N D O F S O U R C E * * * * *

```

```

5722SS1 V5R4M0 060210      Run SQL Statements      SCHEMA      02/10/06 15:35:18  Page  3
Record *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8  SEQNBR Last change
MSG ID SEV  RECORD  TEXT
SQL7953  0      1  Position 1 Drop of DEPT in QSYS complete.
SQL7953  0      3  Position 3 Drop of MANAGER in QSYS complete.
SQL7952  0      5  Position 3 Schema DEPT created.
SQL7950  0      6  Position 8 Table EMP created in DEPT.
SQL7954  0      8  Position 8 Index EMPIND created in DEPT on table EMP in
DEPT.
SQL7966  0     10  Position 8 GRANT of authority to EMP in DEPT completed.
SQL7956  0     10  Position 40 1 rows inserted in EMP in DEPT.
SQL7952  0     13  Position 28 Schema MANAGER created.
SQL7950  0     19  Position 9 Table EMP_SALARY created in collection
MANAGER.
SQL7951  0     21  Position 9 View LEVEL created in MANAGER.
SQL7954  0     23  Position 9 Index SALARYIND created in MANAGER on table
EMP_SALARY in MANAGER.
SQL7966  0     25  Position 9 GRANT of authority to LEVEL in MANAGER
completed.
SQL7966  0     25  Position 37 GRANT of authority to EMP_SALARY in MANAGER
completed.

```

```

Message Summary
Total  Info  Warning  Error  Severe  Terminal
13    13      0        0      0        0
00 level severity errors found in source
*** ** E N D O F L I S T I N G * * * * *

```

ฟังก์ชันของฐานข้อมูลเชิงสัมพันธ์แบบกระจายและ SQL

ฐานข้อมูลเชิงสัมพันธ์แบบกระจาย ประกอบไปด้วยชุด SQL อ็อบเจกต์ที่กระจายอยู่บนระบบคอมพิวเตอร์ที่เชื่อมต่อถึงกัน และกัน.

ฐานข้อมูลเชิงสัมพันธ์เหล่านี้อาจเป็นชนิดเดียวกัน (ตัวอย่างเช่น, DB2 UDB for iSeries) หรือต่างชนิดกัน (DB2 Universal Database สำหรับ z/OS®, DB2 สำหรับ VSE และ VM, DB2 Universal Database (UDB), หรือระบบการจัดการฐานข้อมูลแบบ non-IBM ที่สนับสนุน DRDA). แต่ละฐานข้อมูลเชิงสัมพันธ์นี้มีตัวจัดการฐานข้อมูลเชิงสัมพันธ์เพื่อจัดการกับตารางในสถานะแวดล้อมของตนเอง. ตัวจัดการฐานข้อมูลทำการติดต่อและทำงานร่วมกับตัวจัดการฐานข้อมูลอื่นโดยที่จะอนุญาตให้ตัวจัดการข้อมูลที่มีอยู่สามารถรันคำสั่ง SQL ที่อยู่บนฐานข้อมูลเชิงสัมพันธ์ในระบบอื่นได้.

Application requester จะสนับสนุนการเชื่อมต่อของแอฟพลิเคชัน. แอปพลิเคชันเซิร์ฟเวอร์จะเป็นฐานข้อมูลแบบโลคัลหรือแบบรีโมตให้กับ application requester ที่ถูกเชื่อมต่อ. DB2 UDB for iSeries สนับสนุน Distributed Relational Database Architecture™ (DRDA) เพื่อให้ application requester สามารถสื่อสารกับแอฟพลิเคชันเซิร์ฟเวอร์ได้. นอกจากนี้, DB2 UDB for iSeries ยังสามารถเรียกร้องให้โปรแกรมทางออกอนุมัติการเข้าไปใช้ข้อมูล ระบบจัดการฐานข้อมูลอื่นๆ ที่ไม่สนับสนุน DRDA. โปรแกรมทางออกเหล่านี้จะถูกเรียกว่าโปรแกรม application requester driver (ARD).

DB2 UDB for iSeries สนับสนุนฐานข้อมูลเชิงสัมพันธ์แบบกระจายซึ่งมีสองระดับคือ:

- Remote unit of work (RUW)

Remote unit of work คือการเตรียมการและการรันคำสั่ง SQL ที่เกิดขึ้นที่แอฟพลิเคชันเซิร์ฟเวอร์เดี่ยวเท่านั้นในระหว่างหน่วยการทำงาน. DB2 UDB for iSeries สนับสนุน RUW บน APPC หรือ TCP/IP.

- Distributed unit of work (DUW)

หน่วยการทำงานแบบกระจาย (Distributed unit of work) คือ สถานการณ์ที่การเตรียมการและการรันคำสั่ง SQL สามารถเกิดขึ้นได้ในหลายๆ แอฟพลิเคชันเซิร์ฟเวอร์ในช่วงของหนึ่งหน่วยการทำงาน. อย่างไรก็ตาม, คำสั่ง SQL แบบเดี่ยวสามารถอ้างอิงถึงอ็อบเจกต์ที่อยู่ในแอฟพลิเคชันเซิร์ฟเวอร์แบบเดี่ยวเท่านั้น. DB2 UDB for iSeries สนับสนุน DUW บน APPC และ, ในช่วงต้นๆ ของ V5R1, จะมีการแนะนำการสนับสนุนสำหรับ DUW บน TCP/IP.

หลักการที่เกี่ยวข้อง

“บทนำสู่ DB2 UDB for iSeries Structured Query Language” ในหน้า 3

หัวข้อเหล่านี้อธิบายถึงการนำ Structured Query Language (SQL) ของเซิร์ฟเวอร์ iSeries ไปใช้โดยใช้ DB2 UDB for iSeries และไลเซนส์โปรแกรม DB2 UDB Query Manager and SQL Development Kit.

“SQL แพ็กเกจ” ในหน้า 13

SQL แพ็กเกจ คืออ็อบเจกต์ที่มีโครงสร้างควบคุมซึ่งเกิดขึ้นเมื่อมีการเชื่อมโยงคำสั่ง SQL ในแอฟพลิเคชันโปรแกรมเข้ากับระบบจัดการฐานข้อมูลเชิงสัมพันธ์แบบรีโมต (DBMS).

สิ่งอ้างอิงที่เกี่ยวข้อง

“การเข้าถึงข้อมูลโดยใช้ Distributed Relational Database (DRDA)” ในหน้า 291

ฐานข้อมูลเชิงสัมพันธ์แบบกระจาย ประกอบไปด้วยชุด SQL อ็อบเจกต์ที่กระจายอยู่บนระบบคอมพิวเตอร์ที่เชื่อมต่อถึงกันและกัน. แต่ละฐานข้อมูลเชิงสัมพันธ์จะมีตัวจัดการฐานข้อมูลเชิงสัมพันธ์ เพื่อจัดการกับตารางในสถานะแวดล้อมของตนเอง.

โปรแกรมมีฐานข้อมูลแบบกระจาย

DB2 UDB for iSeries การสนับสนุนฐานข้อมูลเชิงสัมพันธ์แบบกระจาย

ไลเซนส์โปรแกรม DB2 UDB Query Manager and SQL Development Kit สนับสนุนการเข้าถึงแบบโต้ตอบของฐานข้อมูลแบบกระจายโดยใช้คำสั่ง SQL ต่อไปนี้.

- CONNECT
- SET CONNECTION
- DISCONNECT
- RELEASE
- DROP PACKAGE
- GRANT PACKAGE
- REVOKE PACKAGE

การสนับสนุนเพิ่มเติมจะมาจาก development kit ผ่านทางพารามิเตอร์ที่อยู่บนคำสั่งพีรีคอมไพเลอร์ของ SQL:

- คำสั่งสร้าง SQL ILE C Object (CRTSQLCI)
- คำสั่งสร้าง SQL ILE C++ Object (CRTSQLCPPI)
- คำสั่งสร้าง SQL COBOL Program (CRTSQLCBL)
- คำสั่งสร้าง SQL ILE COBOL Object (CRTSQLCBLI)
- คำสั่งสร้าง SQL PL/I Program (CRTSQLPLI)
- คำสั่งสร้าง SQL RPG Program (CRTSQLRPG)
- คำสั่งสร้าง SQL ILE RPG Object (CRTSQLRPGI)

สิ่งอ้างอิงที่เกี่ยวข้อง

“DB2 UDB for iSeries คำอธิบายของคำสั่ง CL” ในหน้า 359

DB2 UDB for iSeries ประกอบด้วยคำสั่ง CL สำหรับ SQL.

ข้อมูลที่เกี่ยวข้อง

การเตรียมและรันโปรแกรมด้วยคำสั่ง SQL

CONNECT statement

DISCONNECT statement

DROP statement

GRANT (Package) statement

REVOKE (Package) statement

RELEASE statement

SET CONNECTION statement

DB2 UDB for iSeries โปรแกรมตัวอย่างฐานข้อมูลเชิงสัมพันธ์แบบกระจาย

โปรแกรมตัวอย่างฐานข้อมูลเชิงสัมพันธ์ที่เป็นแบบ remote unit of work จะถูกส่งไปพร้อมกับผลิตภัณฑ์ของ SQL. มีไฟล์และรายการย่อยหลายตัวอยู่ในไลบรารี QSQL เพื่อช่วยในการจัดเตรียมสภาวะแวดล้อมในการรันโปรแกรมตัวอย่างของ DB2 UDB for iSeries แบบกระจาย.

ในการใช้ไฟล์และรายการย่อยเหล่านี้, จำเป็นที่จะต้องรันงานแบ็ตช์SETUPที่อยู่ในไฟล์ QSQL/QSQSAMP. งานแบ็ตช์SETUP อนุญาตให้คุณปรับแต่งตัวอย่างตามความต้องการเพื่อใช้ในการ:

- สร้างไลบรารี QSQSAMP ที่ตำแหน่ง local และ remote.
- ตั้งค่า directory entry ของฐานข้อมูลเชิงสัมพันธ์ที่ตำแหน่งโลคัลและรีโมต.
- สร้างแอ็พพลิเคชันพาเนลที่ตำแหน่ง local.
- ทำพีคอมไฟล์, คอมไฟล์, และการรันโปรแกรมเพื่อที่จะสร้างแอ็พพลิเคชันตัวอย่างของแบบแผน, ตาราง, ตรรกษนี้, และมุมมอง.
- โหลดข้อมูลเข้าตารางที่ตำแหน่ง local และ remote.
- โปรแกรมแบบ พีคอมไฟล์และ คอมไฟล์.
- สร้าง SQL แพ็กเกจที่ตำแหน่งรีโมตสำหรับแอ็พพลิเคชันโปรแกรม.
- ทำพีคอมไฟล์, คอมไฟล์, และรันโปรแกรมเพื่อที่จะอัปเดตคอลัมน์ตำแหน่งในตารางของแผนก.

ก่อนการรัน SETUP, อาจจำเป็นที่จะต้องแก้ไขรายการย่อยของ SETUP ที่อยู่ในไฟล์ QSQL/QSQSAMP. คำสั่งจะรวมอยู่ในรายการย่อยในรูปของความคิดเห็น. ในการรัน SETUP, ให้ระบุคำสั่งต่อไปนี้บนบรรทัดรับคำสั่ง:

```
=====> SBMDBJOB QSQL/QSQSAMP SETUP
```

รอให้งานแบ็ตช์เสร็จสิ้น.

การใช้โปรแกรมตัวอย่าง, ให้ระบุคำสั่งต่อไปนี้บนบรรทัดรับคำสั่ง:

```
=====> ADDLIB QSQSAMP
```

ในการเรียกจอบแสดงผลแรกที่อนุญาตให้คุณปรับแต่งโปรแกรมตัวอย่างได้ตามความต้องการ, ให้ระบุคำสั่งต่อไปนี้บนบรรทัดรับคำสั่ง.

```
=====> CALL QSQ8HC3
```

ระบบจะแสดงหน้าจอต่อไปนี้. จากจอบแสดงผลนี้, คุณสามารถปรับแต่งโปรแกรมตัวอย่างฐานข้อมูลได้ตามความต้องการ.

DB2 OS/400 ORGANIZATION APPLICATION

ACTION.....: _ A (ADD) E (ERASE)
D (DISPLAY) U (UPDATE)

OBJECT.....: ___ DE (DEPARTMENT) EM (EMPLOYEE)
DS (DEPT STRUCTURE)

SEARCH CRITERIA...: ___ DI (DEPARTMENT ID) MN (MANAGER NAME)
DN (DEPARTMENT NAME) EI (EMPLOYEE ID)
MI (MANAGER ID) EN (EMPLOYEE NAME)

LOCATION.....: _____ (BLANK IMPLIES LOCAL LOCATION)

DATA.....: _____

Bottom

F3=Exit

(C) COPYRIGHT IBM CORP. 1982, 1991

การสนับสนุนการใช้งานแพ็คเกจของ SQL

ระบบปฏิบัติการสนับสนุนอ็อบเจกต์ที่เรียกว่า SQL แพ็คเกจ. ชนิดของอ็อบเจกต์คือ *SQLPKG.

แพ็คเกจ SQL ประกอบไปด้วยโครงสร้างควบคุมและแผนการเข้าไปใช้ข้อมูลที่จำเป็นในการประมวลผลคำสั่ง SQL บนแอฟพลิเคชั่นเซิร์ฟเวอร์ในขณะที่รันโปรแกรมแบบกระจาย. SQL แพ็คเกจสามารถสร้างได้เมื่อ:

- พารามิเตอร์ RDB ถูกระบุไว้ในคำสั่ง CRTSQLxxx และอ็อบเจกต์ของโปรแกรมได้ถูกกำหนดไว้เรียบร้อยแล้ว. SQL แพ็คเกจจะถูกสร้างขึ้นบนระบบที่ระบุไว้ในพารามิเตอร์ RDB.
ถ้าคอมไพล์ไม่ผ่านหรือเพียงแค่อ้างอิงอ็อบเจกต์โมดูลขึ้นมาเพียงอย่างเดียว, จะไม่มีการสร้าง SQL แพ็คเกจขึ้น.
- การใช้คำสั่ง CRTSQLPKG. CRTSQLPKG สามารถถูกนำมาใช้ในการสร้างแพ็คเกจเมื่อแพ็คเกจยังไม่ถูกสร้างขึ้นในช่วงฟรีคอมไพล์ หรือถ้ามีความจำเป็นที่จะต้องใช้แพ็คเกจที่ RDB แทนคำสั่งที่ระบุไว้ในคำสั่งฟรีคอมไพล์.

คำสั่ง Delete SQL Package (DLTSQLPKG) อนุญาตให้ลบ SQL แพ็คเกจที่อยู่บนระบบโลคัลได้.

SQL แพ็คเกจจะไม่ถูกสร้างขึ้นจนกว่าสิทธิพิเศษที่ถือครองโดย ID ที่ให้สิทธิไว้ซึ่งเกี่ยวข้องกับการสร้าง SQL แพ็คเกจรวมทั้งสิทธิที่เหมาะสมในการสร้างแพ็คเกจบนระบบรีโมต (แอฟพลิเคชั่นเซิร์ฟเวอร์). ในการรันโปรแกรม, ID ที่ให้สิทธิไว้จะต้องรวมสิทธิพิเศษในการ EXECUTE ไว้ใน SQL แพ็คเกจด้วย. บนระบบ iSeries, สิทธิพิเศษในการ EXECUTE จะรวมเอาสิทธิในการใช้งานของระบบอันได้แก่ *OBJOPR และ *EXECUTE เอาไว้ด้วย.

ข้อมูลที่เกี่ยวข้อง

Create SQL Package (CRTSQLPKG) command

คำสั่ง SQL ที่ถูกต้องในแพ็คเกจ SQL

โปรแกรมที่เชื่อมต่อกับเซิร์ฟเวอร์อื่นสามารถใช้คำสั่ง SQL ใดๆ ยกเว้นคำสั่ง SET TRANSACTION.

โปรแกรมที่ถูกคอมไพล์โดยใช้ DB2 UDB for iSeries ซึ่งอ้างอิงถึงระบบที่ไม่ใช่ DB2 UDB for iSeries สามารถใช้คำสั่ง SQL ที่เรียกใช้งานได้ซึ่งสนับสนุนโดยระบบริโมตนั่น. พีริคอมไฟเลอร์จะทำการส่งข้อความวินิจฉัยอย่างต่อเนื่องในกรณีมีคำสั่งที่ไม่เข้าใจ. คำสั่งเหล่านั้นจะถูกส่งไปยังระบบริโมตในระหว่างการสร้างแพ็คเกจ SQL. การสนับสนุนรันไทม์จะส่งคืนค่า SQLCODE เป็น -84 หรือ -525 เมื่อไม่สามารถรันคำสั่งบนแอ็พพลิเคชันเซิร์ฟเวอร์ปัจจุบัน. ตัวอย่างเช่น, การ FETCH แบบหลายแถว, การ INSERT แบบกลุ่มเร็กคอร์ด, และเคอร์เซอร์แบบเลื่อนจะใช้ได้เฉพาะในโปรแกรมแบบกระจายที่ทั้ง application requester และ แอ็พพลิเคชันเซิร์ฟเวอร์เป็น i5/OS เวอร์ชัน 2 รีลีส 2 หรือหลังจากนั้น, ด้วยข้อยกเว้นต่อไปนี้. application requester แบบ non-iSeries สามารถออกคำสั่งการดำเนินการในลักษณะ อ่านอย่างเดียว, เคอร์เซอร์แบบเลื่อนได้ที่ไม่ระบุ SENSITIVE บน V5R3 iSeries application server. ข้อจำกัดในการใช้ FETCH แบบหลายแถว, INSERT กลุ่มเร็กคอร์ด, และเคอร์เซอร์แบบเลื่อนได้คือ การไม่อนุญาตให้มีการส่งข้อมูลชนิด BLOB, CLOB และ DBCLOB เมื่อมีการใช้ฟังก์ชันเหล่านี้.

ข้อมูลที่เกี่ยวข้อง

ข้อควรพิจารณาสำหรับการใช้ฐานข้อมูลเชิงสัมพันธ์แบบกระจาย

ข้อควรพิจารณาในการสร้าง SQL แพ็คเกจ

มีข้อควรพิจารณาหลายข้อที่ควรคำนึงถึง เมื่อคุณสร้าง SQL แพ็คเกจ.

การให้สิทธิ์ CRTSQLPKG:

เมื่อคุณสร้าง SQL แพ็คเกจบนระบบ iSeries, authorization ID ที่ใช้จะต้องได้รับสิทธิ์ *USE สำหรับคำสั่ง CRTSQLPKG.

การสร้างแพ็คเกจบน non-DB2 UDB for iSeries:

เมื่อสร้างโปรแกรมและ SQL แพ็คเกจขึ้นสำหรับ non-DB2 UDB for iSeries, และพยายามที่จะใช้คำสั่ง SQL ที่ไม่ซ้ำสำหรับฐานข้อมูลเชิงสัมพันธ์นั้น, พารามิเตอร์ CRTSQLxxx GENLVL ควรจะถูกตั้งค่าอยู่ที่ 30.

โปรแกรมจะไม่ถูกสร้างขึ้นถ้าได้รับสัญญาณข้อความแสดงระดับค่าความรุนแรงมากกว่า 30. ถ้าสัญญาณข้อความถูกส่งออกมาด้วยระดับค่าความรุนแรงที่มีค่ามากกว่า 30, คำสั่ง อาจจะไม่สามารถใช้ได้กับฐานข้อมูลเชิงสัมพันธ์ใดๆ. ตัวอย่างเช่น, ตัวแปรโฮสต์ที่ไม่ได้ถูกระบุไว้หรือไม่สามารถใช้ได้หรือค่าคงที่ที่ไม่สามารถใช้ได้จะส่งสัญญาณข้อความแสดงค่าความรุนแรงมากกว่า 30.

รายการพีริคอมไฟเลอร์จะต้องถูกตรวจสอบหาสัญญาณข้อความผิดปกติเมื่อรันด้วย GENLVL ที่มีค่ามากกว่า 10. เมื่อจะทำการสร้างแพ็คเกจสำหรับฐานข้อมูล DB2 Universal Database, จะต้องตั้งค่าของพารามิเตอร์ GENLVL ให้มีค่าน้อยกว่า 20.

ถ้าพารามิเตอร์ RDB ระบุระบบที่ไม่ใช่ระบบ DB2 UDB for iSeries, อีอ็อปชันที่แสดงดังต่อไปนี้จะต้องไม่ใช้ใน คำสั่ง CRTSQLxxx :

- COMMIT(*NONE)
- OPTION(*SYS)
- DATFMT(*MDY)
- DATFMT(*DMY)
- DATFMT(*JUL)
- DATFMT(*YMD)
- DATFMT(*JOB)

- DYNUSRPRF(*OWNER)
- TIMFMT(*HMS) ถ้าระบุ TIMSEP(*BLANK) หรือ TIMSEP(','')
- SRTSEQ(*JOB RUN)
- SRTSEQ(*LANGIDUNQ)
- SRTSEQ(*LANGIDSHR)
- SRTSEQ(library-name/table-name)

หมายเหตุ: เมื่อทำการเชื่อมต่อกับเซิร์ฟเวอร์ของฐานข้อมูล DB2 Universal Database, จะต้องใช้กฎเพิ่มเติมดังต่อไปนี้:

- รูปแบบวันที่และเวลาที่ระบุไว้จะต้องเป็นรูปแบบเดียวกัน
- ค่าของ *BLANK ต้องถูกนำมาใช้ในพารามิเตอร์ TEXT
- ไม่ใช่แบบแผนที่เป็นค่าดีฟอลต์ (DFTRDBCOL)
- CCSID ของซอร์สโปรแกรมจากแพ็คเกจที่กำลังถูกสร้างจะต้องไม่เป็น 65535; ถ้า 65535 ถูกใช้, แพ็คเกจเปล่าจะถูกสร้างขึ้น.

พารามิเตอร์ Target release (TGTRLS):

ในขณะที่สร้างแพ็คเกจ, คำสั่ง SQL จะถูกตรวจสอบเพื่อหาว่า รีลีสใดที่สามารถสนับสนุนฟังก์ชันได้.

รีลีสนี้จะถูกตั้งค่าให้เป็นระดับเดิมของแพ็คเกจ. ตัวอย่างเช่น, ถ้าแพ็คเกจนั้นมีคำสั่ง CREATE TABLE ซึ่งเพิ่มข้อจำกัดของ FOREIGN KEY เข้าไปให้กับตาราง, จากนั้นค่าระดับที่เรียกคืนได้ของแพ็คเกจจะเป็น เวอร์ชัน 3 รีลีส 1, เนื่องจากไม่มีการสนับสนุนข้อจำกัดของ FOREIGN KEY ไว้ในรีลีสก่อนหน้านี้. สัญญาข้อความ TGTRLS จะถูกระบุไว้เมื่อพารามิเตอร์ TGTRLS มีค่าเป็น *CURRENT.

SQL statement size:

ฟังก์ชันในการสร้าง SQL แพ็คเกจอาจจะไม่สามารถจัดการคำสั่ง SQL ขนาดเดียวกันกับที่ฟรีคอมไพล์สามารถประมวลผลได้.

ในระหว่างการฟรีคอมไพล์ของโปรแกรม SQL, คำสั่ง SQL จะถูกระบุให้อยู่ในพื้นที่ที่เชื่อมโยงกันของโปรแกรม. เมื่อเหตุการณ์นี้เกิดขึ้น, แต่ละโทเค็นจะถูกแยกจากกันด้วยช่องว่าง. นอกจากนี้, เมื่อพารามิเตอร์ RDB ถูกระบุไว้, ตัวแปรโฮสต์ของคำสั่งต้นฉบับจะถูกแทนที่ด้วยตัวอักษร 'H'. ฟังก์ชันการสร้าง SQL แพ็คเกจจะส่งคำสั่งนี้ไปยังแอสเซมบลีเซิร์ฟเวอร์, พร้อมกับรายชื่อตัวแปรโฮสต์ของคำสั่งนั้น. การเติมช่องว่างระหว่างโทเค็นและการแทนที่ของตัวแปรโฮสต์อาจทำให้คำสั่งนั้นมีความยาวเกินความยาวสูงสุดของคำสั่ง SQL ได้ (SQL0101 เหตุผลที่ 5).

คำสั่งที่ไม่จำเป็นต้องใช้แพ็คเกจ:

ในบางกรณี, อาจมีความพยายามที่จะสร้าง SQL แพ็คเกจแต่ SQL แพ็คเกจนั้นจะไม่ถูกสร้างขึ้น และโปรแกรมจะยังรันอยู่. สถานการณ์นี้เกิดขึ้น เมื่อโปรแกรมมีแต่คำสั่ง SQL ที่ไม่จำเป็นต้องใช้ SQL แพ็คเกจในการรัน.

ตัวอย่างเช่น, โปรแกรมที่มีแต่คำสั่ง SQL ที่เป็น DESCRIBE TABLE สร้างข้อความ SQL5041 ในระหว่างการสร้าง SQL แพ็คเกจ. คำสั่ง SQL ที่ไม่จำเป็นต้องใช้ SQL แพ็คเกจได้แก่:

- COMMIT
- CONNECT
- DESCRIBE TABLE

- DISCONNECT
- RELEASE
- RELEASE SAVEPOINT
- ROLLBACK
- SAVEPOINT
- SET CONNECTION

ชนิดอ็อบเจกต์แพ็กเกจ:

แพ็คเกจ SQL จะถูกสร้างขึ้นให้เป็นอ็อบเจกต์แบบ non-ILE เสมอและจะรันอยู่ใน activation group ปกติเสมอ.

โปรแกรม ILE และโปรแกรมเซอริวิส:

โปรแกรม ILE และเซอริวิสโปรแกรมที่เชื่อมโมดูลหลายๆ โมดูลที่มีคำสั่ง SQL จะต้องมี SQL แพ็กเกจแยกออกจากกันสำหรับแต่ละโมดูล.

การเชื่อมต่อโดยสร้างแพ็กเกจ:

ชนิดของการเชื่อมต่อที่สร้างขึ้นสำหรับการสร้างแพ็คเกจจะขึ้นอยู่กับชนิดของการเชื่อมต่อที่ถูกร้องขอโดยใช้พารามิเตอร์ RDBCNMTH.

ถ้า RDBCNMTH(*DUW) ถูกระบุไว้, commitment control จะถูกนำมาใช้และการเชื่อมต่อจะเป็นแบบ read-only. ถ้าการเชื่อมต่อเป็นแบบ read-only, การสร้างแพ็คเกจจะล้มเหลว.

หน่วยของงาน:

เนื่องจากการสร้างแพ็คเกจจะเป็นการ commit หรือ rollback, commit definition จะต้องเป็นขอบเขตของหน่วยของการทำงานก่อนที่จะมีการสร้างแพ็คเกจ.

เงื่อนไขต่อไปนี้เป็นจริงทั้งหมด เพื่อให้ commit definition เป็นขอบเขตของหน่วยการทำงาน:

- SQL เป็นขอบเขตของหน่วยการทำงาน.
- ไม่มีไฟล์ล็อกหรือไฟล์ DDM ใดๆ ที่เปิดโดยใช้ commitment control และไม่มีการปิดไฟล์ล็อกหรือ DDM ใดขณะที่มีการเปลี่ยนแปลงค้างอยู่.
- ไม่มีรีจิสเตอร์ของ API ที่ถูก register ไว้.
- ไม่มีรีจิสเตอร์ของ LU 6.2 ที่ถูก register ไว้ที่ไม่เกี่ยวข้องกับ DRDA หรือ DDM.

การสร้างแพ็คเกจแบบโลคัล:

ชื่อที่ระบุบนพารามิเตอร์ RDB สามารถเป็นชื่อของระบบโลคัลได้.

ถ้าเป็นเช่นนั้น, SQL แพ็คเกจจะถูกสร้างขึ้นบนระบบโลคัล. SQL แพ็คเกจจะถูกบันทึก (คำสั่ง SAVOBJ) และเรียกคืน (คำสั่ง RSTOBJ) ไปยังอีกเซิร์ฟเวอร์หนึ่งได้. เมื่อทำการรันโปรแกรมด้วยการเชื่อมต่อไปยังระบบโลคัล, แพ็คเกจ SQL จะไม่ถูกนำมาใช้. ถ้าทำการระบุ *LOCAL สำหรับพารามิเตอร์ RDB ไว้, อ็อบเจกต์ *SQLPKG จะไม่ถูกสร้างขึ้น, แต่ข้อมูลของแพ็คเกจจะถูกบันทึกไว้ในอ็อบเจกต์ *PGM.

เลเบล (Label):

คุณสามารถใช้คำสั่ง LABEL ON ในการสร้างรายละเอียดสำหรับ SQL แพ็กเกจ.

โทเค็นที่สอดคล้องกัน:

โปรแกรมและ SQL แพ็กเกจที่เกี่ยวข้องจะมีโทเค็นที่มีความสอดคล้องกัน ซึ่งได้รับการตรวจสอบ เมื่อมีการเรียก SQL แพ็กเกจเกิดขึ้น.

โทเค็นที่มีความสอดคล้องกันนี้จะต้องมีค่าตรงกัน มิฉะนั้นจะไม่สามารถนำมาใช้ได้. อาจเป็นไปได้ที่โปรแกรมและแพ็คเกจ SQL จะไม่มีความสัมพันธ์กัน. สมมติให้โปรแกรมอยู่บนระบบ iSeries และแอปพลิเคชันเซิร์ฟเวอร์เป็นอีกระบบ iSeries หนึ่ง. โปรแกรมที่กำลังรันอยู่ในเซสชัน A และจะถูกสร้างขึ้นใหม่ในเซสชัน B (ซึ่ง SQL แพ็กเกจจะถูกสร้างขึ้นที่นี่ด้วย). การเรียกโปรแกรมในเซสชัน A ครั้งต่อไปอาจทำให้เกิดผลลัพธ์ที่ผิดพลาดของโทเค็นที่มีความสอดคล้องกัน. เพื่อหลีกเลี่ยงการระบุตำแหน่งของ SQL แพ็กเกจในการเรียกแต่ละครั้ง, SQL จะคงค่ารายการของแอดเดรสสำหรับ SQL แพ็กเกจที่ถูกใช้ในแต่ละเซสชันเอาไว้. เมื่อเซสชัน B ทำการสร้าง SQL แพ็กเกจขึ้นมาใหม่, SQL แพ็กเกจอันเก่าก็就会被ย้ายไปยังโลบรารี QRPLOBJ. แอดเดรสไปยัง SQL แพ็กเกจในเซสชัน A จะยังคงเป็นค่าที่ใช้ได้อยู่. (สถานการณ์เช่นนี้สามารถหลีกเลี่ยงได้โดยการสร้างโปรแกรมและ SQL แพ็กเกจจากเซสชันที่กำลังรันโปรแกรมอยู่, หรือโดยการส่งคำสั่งรีโมตไปลบ SQL แพ็กเกจอันเก่าก่อนจะสร้างโปรแกรมใหม่.)

ในการที่จะใช้ SQL แพ็กเกจอันใหม่, คุณควรจบการเชื่อมต่อเข้ากับระบบรีโมตเสียก่อน. คุณสามารถเลือกอย่างใดอย่างหนึ่งระหว่างการออกจากเซสชันก่อนแล้วจึงเข้ามาใหม่, หรือคุณสามารถใช้คำสั่ง SQL แบบโต้ตอบ (STRSQL) ในการออกคำสั่ง DISCONNECT สำหรับการเชื่อมต่อของเครือข่ายที่ไม่มีมอด็มหรือ คำสั่ง RELEASE ตามด้วย COMMIT สำหรับการเชื่อมต่อที่มีการมอด็ม. ดังนั้นจึงควรนำ RCLDDMCNV มาใช้ในการจบการเชื่อมต่อของเครือข่าย. จากนั้นจึงเรียกโปรแกรมอีกครั้งหนึ่ง.

SQL และการเรียกซ้ำ:

ถ้ามีการเรียก SQL จากโปรแกรมคีย์ Attention ในขณะที่กำลังทำการพีคอมไพล์อยู่แล้วนั้น, จะทำให้ได้รับผลลัพธ์ที่ไม่ปรารถนาได้.

คำสั่ง CRTSQLxxx, CRTSQLPKG, STRSQL และสถานะแวดล้อมของรันไทม์ของ SQL จะไม่สามารถถูกเรียกซ้ำได้. ทั้งนี้จะทำให้เกิดผลลัพธ์ที่ไม่ปรารถนาตามมา หากมีการพยายามทำการเรียกซ้ำ. การเรียกซ้ำจะเกิดขึ้นถ้าในขณะที่คำสั่งใดคำสั่งหนึ่งกำลังรันอยู่, (หรือในการรันโปรแกรมที่มีคำสั่ง SQL อยู่) งานนั้นเกิดขัดข้องก่อนที่คำสั่งจะทำงานเสร็จสมบูรณ์, และฟังก์ชัน SQL อีกอันหนึ่งจะถูกเริ่มการทำงานขึ้น.

ข้อควรพิจารณาเกี่ยวกับ CCSID สำหรับ SQL

ถ้ากำลังรันแอปพลิเคชันแบบกระจายอยู่และระบบใดระบบหนึ่งของระบบที่มีอยู่นั้นไม่ใช่ระบบ iSeries, ค่า CCSID ของงานที่อยู่บนเซิร์ฟเวอร์ iSeries นั้นจะไม่สามารถถูกตั้งค่าให้เป็น 65535 ได้.

ก่อนการร้องขอให้ระบบรีโมตสร้างแพ็คเกจ SQL, application requester จะทำการแปลงชื่อที่ระบุบนพารามิเตอร์ RDB, ชื่อของแพ็คเกจ SQL, ชื่อของโลบรารี, และเนื้อความที่อยู่ในแพ็คเกจจาก CCSID ไปเป็น CCSID 500 เสมอ. การกระทำเช่นนี้จะถูกเรียกกรองจาก DRDA. เมื่อฐานข้อมูลเชิงสัมพันธ์แบบรีโมตเป็นระบบ iSeries, ชื่อต่างๆ จะไม่ถูกแปลงจาก CCSID 500 ไปเป็น CCSID ของงานนั้นๆ.

คุณไม่ควรใช้ identifiers ที่ใช้สำหรับค้นกับชื่อของตาราง, มุมมอง, ตรรกษณ์, แบบแผน, ไลบรารี, หรือแพ็คเกจ SQL. การแปลงชื่อระหว่างระบบที่มี CCSID ต่างกันจะทำไม่ได้. พิจารณาตัวอย่างดังต่อไปนี้ซึ่งระบบ A กำลังรันด้วยค่า CCSID เป็น 37 และระบบ B กำลังรันด้วยค่า CCSID เป็น 500.

- สร้างโปรแกรมที่สร้างตารางด้วยชื่อ "a-blc" บนระบบ A.
- บันทึกโปรแกรม "a-blc" บนระบบ A, จากนั้นทำการเรียกคืนไปยังระบบ B.
- จุดของโค้ดสำหรับ - ใน CCSID 37 คือ x'5F' ขณะที่ใน CCSID 500 คือ x'BA'.
- บนระบบ B ชื่อจะแสดงผลเป็น "a[b]c". ถ้าคุณสามารถสร้างโปรแกรมที่อ้างอิงถึงตารางซึ่งมีชื่อว่า "a-blc.", โปรแกรมจะหาตารางไม่พบ.

อักขระ (@), เครื่องหมาย pound (#), และเครื่องหมายดอลลาร์ (\$) ไม่ควรนำมาใช้ในชื่อของ SQL อ็อบเจกต์. จำนวนจุดของโค้ดจะขึ้นอยู่กับค่าของ CCSID ที่ใช้. ถ้าใช้ชื่อที่มีตัวค้นหรือมีสาม national extender อยู่, อาจทำให้ฟังก์ชันการค้นหาชื่อที่รีสึทที่จะออกมาในขนาดล้มเหลวได้.

การจัดการการเชื่อมต่อและ activation group

การเชื่อมต่อ SQL จะถูกควบคุมจัดการที่ระดับของ activation group. แต่ละ activation group ภายในหนึ่งงานจะควบคุมจัดการการเชื่อมต่อของตัวเองโดยไม่มีการใช้ข้าม activation group กัน.

ก่อนที่จะใช้ TCP/IP โดย DRDA, คำว่า *การเชื่อมต่อ* เป็นคำที่มีความหมายชัดเจนมาก. คำนี้หมายถึงการเชื่อมต่อระหว่าง point of view ของ SQL. นั่นคือ, การเชื่อมต่อเริ่มต้นเมื่อมีการ CONNECT TO เข้ากับ RDB, และสิ้นสุดเมื่อมีการ DISCONNECT หรือมีการใช้คำสั่ง RELEASE ALL ตามด้วย COMMIT ที่รันเรียบร้อยแล้ว. การสนทนา (ระหว่างโปรแกรม) ของ APPC อาจจะถูกเก็บหรือไม่เก็บเอาไว้, ขึ้นอยู่กับค่าขององค์ประกอบ DDMCNV ของงาน, และขึ้นอยู่กับว่าการสนทนา (ระหว่างโปรแกรม) นั้นเกิดขึ้นกับระบบ iSeries หรือระบบอื่นๆ.

ศัพท์บัญญัติของ TCP/IP ไม่รวมถึงคำว่า *การสนทนา (ระหว่างโปรแกรม)*. อย่างไรก็ตาม, ได้มีการกล่าวถึงแนวคิดที่คล้ายกันได้. การสนับสนุน TCP/IP โดย DRDA, ทำให้การใช้งานของคำศัพท์ *การสนทนา (ระหว่างโปรแกรม)* ถูกแทนที่, ในหัวข้อนี้, ด้วยคำศัพท์ *การเชื่อมต่อ*, จนกว่าจะมีการกล่าวถึงการสนทนา (ระหว่างโปรแกรม) ของ APPC อย่างเฉพาะเจาะจง. ดังนั้น, จะมีการเชื่อมต่ออยู่สองประเภทที่ผู้อ่านต้องทราบ คือ: การเชื่อมต่อ SQL ของประเภทที่อธิบายมาแล้วข้างบน, และการเชื่อมต่อเน็ตเวิร์กที่ใช้แทนคำว่า *การสนทนา (ระหว่างโปรแกรม)*.

บางกรณีอาจเกิดความสับสนระหว่างการเชื่อมต่อสองประเภทนี้, ดังนั้น เราจะกล่าวถึงการเชื่อมต่อพวกนี้ด้วยคำว่า SQL หรือเน็ตเวิร์ก เพื่อให้ผู้อ่านเข้าใจชัดเจนขึ้นว่าหมายถึงเรื่องใด.

ต่อไปนี้เป็นตัวอย่างของแอ็พพลิเคชันที่รันอยู่ในหลายๆ activation group. ซึ่งจะแสดงการโต้ตอบระหว่าง activation group, การจัดการการเชื่อมต่อ, และ commitment control. รูปแบบนี้ไม่แนะนำในให้ใช้ในการเขียนโค้ด.

ซอร์สโค้ดสำหรับ PGM 1

นี่คือซอร์สโค้ดสำหรับ PGM1.

```

....
EXEC SQL
CONNECT TO SYSB
END-EXEC.
EXEC SQL
SELECT ....
END-EXEC.
CALL PGM2.
....

```

รูปที่ 2. ซอร์สโค้ดสำหรับ PGM1

คำสั่งในการสร้างโปรแกรม และ SQL แพ็กเกจสำหรับ PGM1:

```
CRTSQLCBL PGM(PGM1) COMMIT(*NONE) RDB(SYSB)
```

ซอร์สโค้ดสำหรับ PGM2

นี่คือซอร์สโค้ดสำหรับ PGM2.

```

...
EXEC SQL
CONNECT TO SYSC;
EXEC SQL
DECLARE C1 CURSOR FOR
    SELECT ....;
EXEC SQL
OPEN C1;
    do {
EXEC SQL
FETCH C1 INTO :st1;
EXEC SQL
UPDATE ...
    SET COL1 = COL1+10
    WHERE CURRENT OF C1;
        PGM3(st1);
    } while SQLCODE == 0;
EXEC SQL
CLOSE C1;
EXEC SQL COMMIT;
....

```

รูปที่ 3. ซอร์สโค้ดสำหรับ PGM2

คำสั่งที่ใช้ในการสร้างโปรแกรม และ SQL แพ็กเกจสำหรับ PGM2:

```
CRTSQLCI OBJ(PGM2) COMMIT(*CHG) RDB(SYSC) OBJTYPE(*PGM)
```

ซอร์สโค้ดสำหรับ PGM3

นี่คือซอร์สโค้ดสำหรับ PGM3.

```

...
EXEC SQL
  INSERT INTO TAB VALUES(:st1);
EXEC SQL COMMIT;
....

```

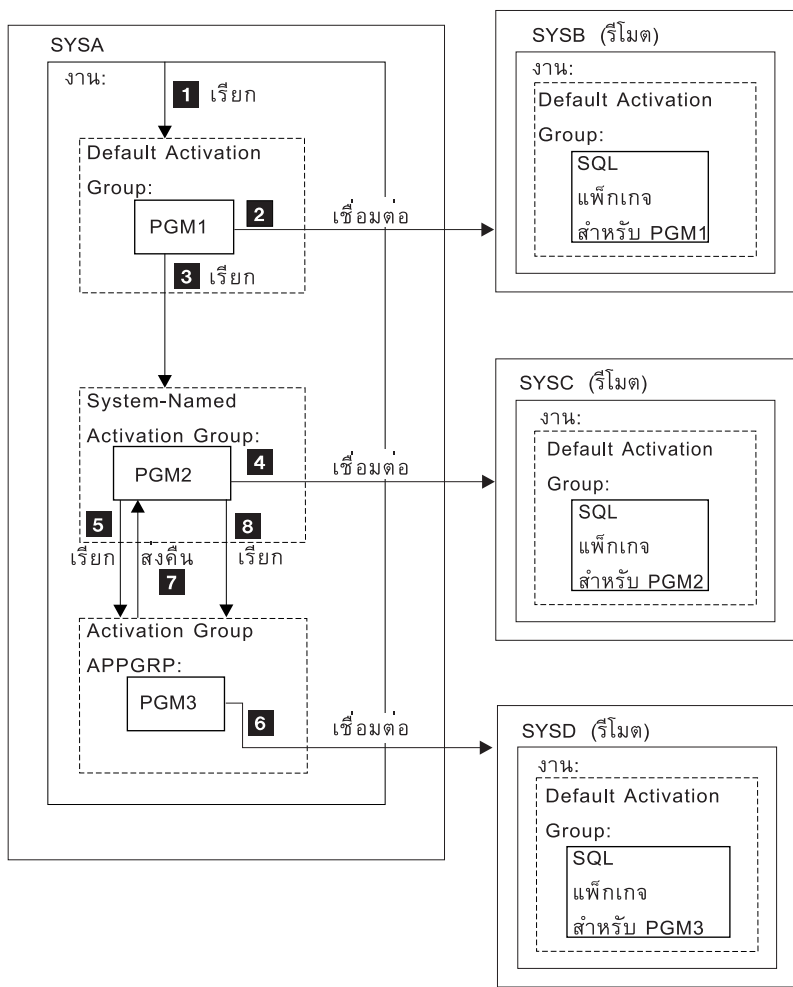
รูปที่ 4. ซอร์สโค้ดสำหรับ PGM3

คำสั่งในการสร้างโปรแกรม และ SQL แพ็กเกจสำหรับ PGM3:

```

CRTSQLCI OBJ(PGM3) COMMIT(*CHG) RDB(SYSD) OBJTYPE(*MODULE)
CRTPGM PGM(PGM3) ACTGRP(APPGRP)
CRTSQLPKG PGM(PGM3) RDB(SYSD)

```



RV2W577-3

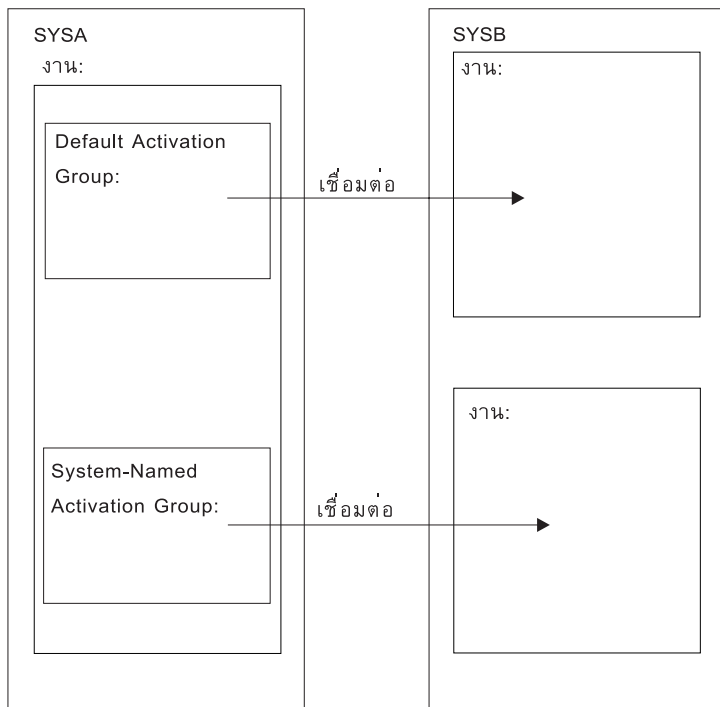
ในตัวอย่างนี้, PGM1 เป็นโปรแกรม non-ILE ที่ถูกสร้างขึ้นโดยใช้คำสั่ง CRTSQLCBL. โปรแกรมนี้จะรันใน activation group ดีฟอลต์. PGM2 ถูกสร้างขึ้นโดยใช้คำสั่ง CRTSQLCI, และจะรันใน activation group ที่ได้รับการตั้งชื่อโดยระบบ. PGM3 ก็ถูกสร้างโดยใช้คำสั่ง CRTSQLCI เช่นกัน, แต่จะรันใน activation group ชื่อ APPGRP. เนื่องจาก APPGRP ไม่ได้เป็นค่าดีฟอลต์สำหรับพารามิเตอร์ ACTGRP, คำสั่ง CRTPGM ก็จะถูกรันแยกต่างหาก. คำสั่ง CRTPGM จะตามด้วยคำสั่ง CRTSQLPKG ที่สร้างอ็อบเจกต์แพ็คเกจ SQL อยู่บนฐานข้อมูลเชิงสัมพันธ์ SYSD. ในตัวอย่างนี้, ผู้ใช้ไม่ได้ทำการเรียกโปรแกรมทำงานของ definition ของ job level commitment ไว้โดยชัดเจน. SQL จะเรียกโปรแกรมทำงานของ commitment control โดยนัย.

1. PGM1 ถูกเรียกและรันอยู่ใน activation group ดีฟอลต์.
2. PGM1 เชื่อมต่ออยู่กับฐานข้อมูลเชิงสัมพันธ์ SYSB และรันคำสั่ง SELECT.
3. PGM1 จะเรียก PGM2, ซึ่งรันใน activation group ที่ถูกตั้งชื่อโดยระบบ.
4. PGM2 ทำการเชื่อมต่อกับฐานข้อมูลเชิงสัมพันธ์ SYSC. เนื่องจาก PGM1 และ PGM2 จะอยู่ใน activation group ที่ต่างกัน, การเชื่อมต่อจาก PGM2 ใน activation group ที่ระบบตั้งชื่อให้ จะตัดการเชื่อมต่อจาก PGM1 ใน activation group ดีฟอลต์. การเชื่อมต่อทั้งสองนี้จะแอ็คทีฟทั้งคู่. PGM2 จะเปิดเคอร์เซอร์และดึงข้อมูลออกมา และจะอัปเดตแถวข้อมูล. โดยรันภายใต้ commitment control, ซึ่งอยู่ในระหว่างหน่วยการทำงาน, และไม่อยู่ที่สถานะที่จะเชื่อมต่อได้.
5. PGM2 เรียก PGM3, ซึ่งรันใน activation group APPGRP.
6. คำสั่ง INSERT เป็นคำสั่งแรกที่รันใน activation group APPGRP. คำสั่ง SQL จะทำให้เกิดการเชื่อมต่อไปยังฐานข้อมูลเชิงสัมพันธ์ SYSD โดยนัย. แถวข้อมูลจะถูกแทรกเข้าไปในตาราง TAB ที่อยู่ในฐานข้อมูลเชิงสัมพันธ์ SYSD. จากนั้นการแทรกก็จะถูก commit. การเปลี่ยนแปลงที่ค้างอยู่ใน activation group ที่ตั้งชื่อโดยระบบ จะไม่ถูก commit, เนื่องจาก commitment control ถูกเรียกทำงานโดย SQL ด้วยขอบเขตการ commit ของ activation group.
7. จากนั้น PGM3 จะจบการทำงาน และการควบคุมจะกลับไปเป็นของ PGM2. PGM2 ทำการดึงข้อมูลและอัปเดตแถวข้อมูลอีกแถวหนึ่ง.
8. PGM3 ถูกเรียกขึ้นมาอีกครั้งเพื่อแทรกแถว. การเชื่อมต่อโดยนัยจะเสร็จสิ้นตั้งแต่มีการเรียกไปยัง PGM3 ครั้งแรก. โดยไม่มีการเชื่อมต่อในการเรียกครั้งต่อๆ มาอีกเนื่องจาก activation group ไม่ได้จบลงในระหว่างการเรียกไปยัง PGM3. ท้ายที่สุด, แถวข้อมูลทั้งหมดจะถูกประมวลผลโดย PGM2 และหน่วยการทำงานที่เชื่อมโยงกับ activation group ที่ถูกตั้งชื่อโดยระบบก็จะถูก commit.

การเชื่อมต่อหลายครั้งไปยังฐานข้อมูลเชิงสัมพันธ์เดียวกัน

หากมีหลาย activation group เชื่อมต่อไปยังฐานข้อมูลเชิงสัมพันธ์เดียวกัน, การเชื่อมต่อ SQL แต่ละครั้งจะเชื่อมต่อเครือข่ายในตัวเอง และสร้างงานแอ็พพลิเคชันเซิร์ฟเวอร์ในตัวเองอีกด้วย.

ถ้า activation group ถูกรันด้วย commitment control, การเปลี่ยนแปลงที่ถูก commit ใน activation group ใดจะไม่ commit การเปลี่ยนแปลงใน activation group จนกว่า definition ของ job-level commitment ถูกนำมาใช้.



RV2W578-2

การจัดการเชื่อมต่อโดยนัยสำหรับ activation group ดีฟอลต์

application requester สามารถเชื่อมต่อโดยนัยกับแอ็พพลิเคชันเซิร์ฟเวอร์ได้.

การเชื่อมต่อ SQL โดยนัยเกิดขึ้นเมื่อ application requester ตรวจสอบว่า คำสั่ง SQL แรกกำลังถูกใช้โดยโปรแกรม SQL แรกที่แอ็คทีฟใน activation group ดีฟอลต์ และรายการต่อไปนี้เป็นจริง:

- คำสั่ง SQL ที่ถูกเรียกนั้นไม่ใช่คำสั่ง CONNECT ที่มีพารามิเตอร์.
- SQL ไม่แอ็คทีฟใน activation group ดีฟอลต์.

สำหรับโปรแกรมแบบกระจาย, การเชื่อมต่อโดยนัยของ SQL เป็นการเชื่อมต่อไปยังฐานข้อมูลเชิงสัมพันธ์ที่ระบุไว้บนพารามิเตอร์ RDB. สำหรับโปรแกรมที่เป็นแบบไม่กระจาย, การเชื่อมต่อโดยนัยของ SQL เป็นการเชื่อมต่อไปยังฐานข้อมูลเชิงสัมพันธ์แบบโลคัล.

SQL จะสิ้นสุดการเชื่อมต่อที่แอ็คทีฟใน activation group ดีฟอลต์เมื่อ SQL เปลี่ยนสภาพเป็นไม่แอ็คทีฟ. SQL จะไม่แอ็คทีฟเมื่อ:

- Application requester ตรวจสอบโปรแกรม SQL แรกที่แอ็คทีฟสำหรับกระบวนการนั้นได้สิ้นสุดลง และสิ่งต่อไปนี้ทั้งหมด:
 - ไม่มีการเปลี่ยนแปลง SQL ใดๆ ค้างอยู่
 - ไม่มีการเชื่อมต่อใดใช้การเชื่อมต่อที่ถูกป้องกันไว้
 - คำสั่ง SET TRANSACTION ไม่แอ็คทีฟ
 - ไม่มีการรันโปรแกรมใดๆ ที่ถูกพรีคอมไพล์ด้วย CLOSQLCSR(*ENDJOB).

ถ้ามีการเปลี่ยนแปลงที่ค้างไว้, การเชื่อมต่อถูกป้องกันเอาไว้, หรือมีคำสั่ง SET TRANSACTION แอ็คทีฟอยู่, SQL จะถูกระบุให้อยู่ในสถานะจบการทำงาน. ถ้ามีการรันโปรแกรมที่ถูกพรีคอมไพล์ด้วย CLOSQLCSR(*ENDJOB), SQL จะยังคงแอ็คทีฟสำหรับ activation group ดีฟอลต์จนกระทั่งงานนั้นสิ้นสุดลง.

- ในตอนท้ายของหน่วยของการทำงาน, ถ้า SQL อยู่ในสถานะจบการทำงาน. เหตุการณ์นี้จะเกิดขึ้นเมื่อทำการออกคำสั่ง COMMIT หรือ ROLLBACK จากนอกโปรแกรม SQL.
- ในตอนสิ้นสุดการทำงาน.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การสิ้นสุดการเชื่อมต่อ” ในหน้า 329

เนื่องจากการเชื่อมต่อแบบรีโมตมีการใช้ซอร์ส, การเชื่อมต่อที่จะไม่ถูกใช้อีก จึงควรจบการทำงานให้เร็วที่สุด. การเชื่อมต่อสามารถสิ้นสุดได้ไม่ว่าจะโดยนัย หรือโดยชัดเจน.

การจัดการการเชื่อมต่อโดยนัยสำหรับ activation group ที่ไม่ใช่ดีฟอลต์

Application requester สามารถเชื่อมต่อโดยนัยไปยังแอ็พพลิเคชันเซิร์ฟเวอร์ได้. ซึ่งจะเกิดขึ้นเมื่อ application requester ตรวจพบว่าคำสั่ง SQL แรกที่ส่งออกมาจาก activation group ไม่ใช่คำสั่ง CONNECT ที่มีพารามิเตอร์.

สำหรับโปรแกรมแบบกระจาย, การเชื่อมต่อ SQL โดยนัยจะถูกกระทำกับฐานข้อมูลเชิงสัมพันธ์ที่ระบุในพารามิเตอร์ RDB. สำหรับโปรแกรมที่เป็นแบบไม่กระจาย, การเชื่อมต่อ SQL โดยนัยจะถูกกระทำกับฐานข้อมูลเชิงสัมพันธ์แบบโลคัล.

การตัดการเชื่อมต่อโดยนัยสามารถเกิดขึ้นได้ในกระบวนการในช่วงเวลาต่อไปนี้:

- เมื่อ activation group สิ้นสุดการทำงาน, ถ้า commitment control ไม่ใช่แอ็คทีฟ, commitment control ในระดับของ activation group จะแอ็คทีฟ, หรือ commitment definition ของระดับงานจะอยู่ที่ขอบเขตของหน่วยการทำงาน.

ถ้า commitment definition ของระดับงานแอ็คทีฟ และไม่อยู่ที่ขอบเขตของหน่วยการทำงาน, SQL จะถูกระบุให้อยู่ในสถานะจบการทำงาน.

- ถ้า SQL อยู่ในสถานะจบการทำงาน, เมื่อ commitment definition ของระดับงานนั้นถูก commit หรือ roll back.
- ในตอนสิ้นสุดการทำงาน.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การสิ้นสุดการเชื่อมต่อ” ในหน้า 329

เนื่องจากการเชื่อมต่อแบบรีโมตมีการใช้ซอร์ส, การเชื่อมต่อที่จะไม่ถูกใช้อีก จึงควรจบการทำงานให้เร็วที่สุด. การเชื่อมต่อสามารถสิ้นสุดได้ไม่ว่าจะโดยนัย หรือโดยชัดเจน.

การสนับสนุนแบบกระจาย

DB2 UDB for iSeries สนับสนุนฐานข้อมูลเชิงสัมพันธ์แบบกระจายซึ่งมีสองระดับ.

- Remote unit of work (RUW)

Remote unit of work คือ สถานการณ์ที่การเตรียมการและการรันคำสั่ง SQL หลายๆ คำสั่ง เกิดขึ้นในหนึ่งแอ็พพลิเคชันเซิร์ฟเวอร์เท่านั้นในช่วงของหนึ่งหน่วยการทำงาน. activation group พร้อมด้วยกระบวนการของแอ็พพลิเคชันที่ application requester สามารถเชื่อมต่อไปยังแอ็พพลิเคชันเซิร์ฟเวอร์, และรันคำสั่ง SQL แบบ static หรือ dynamic ที่อ้างอิงถึงอ็อบเจกต์ที่อยู่บนแอ็พพลิเคชันเซิร์ฟเวอร์, ภายในหน่วยการทำงานตั้งแต่หนึ่งหน่วยขึ้นไป. Remote unit of work ก็ถูกอ้างอิงถึงเช่นเดียวกันว่าเป็น DRDA ระดับ 1.

- Distributed unit of work (DUW)

หน่วยการทำงานแบบกระจาย (Distributed unit of work) คือ สถานการณ์ที่การเตรียมการและการรันคำสั่ง SQL สามารถเกิดขึ้นได้ในหลายๆ แอ็พพลิเคชันเซิร์ฟเวอร์ในช่วงของหนึ่งหน่วยการทำงาน . อย่างไรก็ตาม, คำสั่ง SQL แบบเดี่ยวสามารถอ้างอิงถึงได้เฉพาะอ็อบเจกต์ที่อยู่ในแอ็พพลิเคชันเซิร์ฟเวอร์แบบเดี่ยวเท่านั้น. Distributed unit of work ก็ถูกอ้างอิงถึงเช่นเดียวกันว่าเป็น DRDA ระดับ 2.

หน่วยการทำงานแบบกระจายจะอนุญาตให้:

- อัปเดตการเข้าใช้งานในหลายๆ แอ็พพลิเคชันเซิร์ฟเวอร์ในหนึ่งหน่วยการทำงานแบบโลจิคัล หรือ
- อัปเดตการเข้าใช้งานในแอ็พพลิเคชันเซิร์ฟเวอร์แบบเดี่ยวโดยการเข้าไปอ่านข้อมูลในหลายแอ็พพลิเคชันเซิร์ฟเวอร์, ในหนึ่งหน่วยการทำงานแบบโลจิคัล.

แอ็พพลิเคชันเซิร์ฟเวอร์หลายเซิร์ฟเวอร์จะถูกอัปเดตในหนึ่งหน่วยการทำงานได้หรือไม่จะขึ้นอยู่กับว่ามี sync point manager ที่ application requester หรือไม่, รวมทั้ง sync point manager ที่แอ็พพลิเคชันเซิร์ฟเวอร์, และการสนับสนุน commit protocol แบบสองเฟสระหว่าง application requester กับ application server.

sync point manager เป็นส่วนประกอบของระบบที่ทำงานร่วมกับปฏิบัติการ commit และ rollback ท่ามกลางส่วนประกอบอื่นๆ ของ commit protocol แบบสองเฟส. เมื่อทำการรันการอัปเดตแบบกระจาย, sync point manager บนแต่ละระบบจะทำงานร่วมกันเพื่อตรวจสอบว่ามีรีซอร์สมากพอ. protocol และ flow ที่ถูกใช้โดย sync point manager จะถูกอ้างถึงว่าเป็น commit protocol แบบสองเฟสด้วยเช่นกัน. ถ้า commit protocol แบบสองเฟสจะถูกนำมาใช้, การเชื่อมต่อจะกลายเป็นรีซอร์สที่ถูกป้องกันไว้; มิฉะนั้น การเชื่อมต่อจะเป็นรีซอร์สที่ไม่ได้ป้องกันเอาไว้.

ชนิดของโปรโตคอลที่ใช้ในการส่งข้อมูลระหว่างระบบจะส่งผลกระทบต่อการเชื่อมต่อระบบว่าเป็นแบบป้องกันหรือไม่. ก่อน V5R1, จะไม่มีการป้องกันการเชื่อมต่อแบบ TCP/IP, ดังนั้นการเชื่อมต่อสามารถร่วมในหน่วยของงานในทางที่ค่อนข้างจำกัด. ใน V5R1, การสนับสนุนแบบเต็มสำหรับ DUW โดยการใช้การเชื่อมต่อ TCP/IP ถูกเพิ่มเติมเข้าไป. ตัวอย่างเช่น, ถ้าการเชื่อมต่อครั้งแรกที่เกิดขึ้นจากโปรแกรมเป็นการเชื่อมต่อไปยังเซิร์ฟเวอร์รุ่นก่อน V5R1 บน TCP/IP, การอัปเดตสามารถกระทำได้ในการเชื่อมต่อนี้, แต่การเชื่อมต่อที่ตามมา, แม้กระทั่งบน APPC, จะเป็นแบบอ่านอย่างเดียว.

จะสังเกตได้ว่าเมื่อมีการใช้ SQL แบบโต้ตอบ, การเชื่อมต่อ SQL ครั้งแรกจะเป็นการเชื่อมต่อไปยังระบบโลคัล. ดังนั้น, ในสภาวะแวดล้อมของระบบก่อนรุ่น V5R1, หากต้องการอัปเดตระบบรีโมตโดยการใช้ TCP/IP, จะต้องทำการ RELEASE ALL แล้วตามด้วย COMMIT ในการสิ้นสุดการเชื่อมต่อ SQL ทั้งหมด ก่อนที่จะทำการ CONNECT TO ระบบ remote-tcp-system.

การจำแนกประเภทของการเชื่อมต่อ

เมื่อการการเชื่อมต่อแบบรีโมตเกิดขึ้น ระบบจะใช้การเชื่อมต่อระบบเครือข่ายแบบป้องกันหรือแบบไม่ได้ป้องกันอย่างใดอย่างหนึ่ง.

สำหรับการอัปเดตที่สามารถ commit ได้ นั้น, การเชื่อมต่อของ SQL นี้จะเป็นแบบอ่านได้อย่างเดียว, อัปเดตได้, หรือไม่ทราบว่าการอัปเดตได้หรือไม่เมื่อมีการเชื่อมต่อเกิดขึ้น. การอัปเดตที่สามารถ commit ได้ นั้นจะเป็น การแทรก, การลบ, การอัปเดต, หรือคำสั่ง DDL ที่สามารถรันภายใต้ commitment control. ถ้าการเชื่อมต่อเป็นแบบอ่านเพียงอย่างเดียว, การเปลี่ยนแปลงที่ต้องใช้ COMMIT(*NONE) ยังรันได้. หลังจากการ CONNECT หรือ SET CONNECTION, SQLERRD(4) ของ SQLCA ได้ระบุชนิดของการเชื่อมต่อแล้ว.

DB2_CONNECTION_TYPE ค่าเฉพาะคือ:

1. การเชื่อมต่อไปยังฐานข้อมูลเชิงสัมพันธ์แบบโลคัลและการเชื่อมต่อได้รับการปกป้อง.
2. การเชื่อมต่อไปยังฐานข้อมูลเชิงสัมพันธ์แบบรีโมตและการเชื่อมต่อไม่ได้รับการปกป้อง.
3. การเชื่อมต่อไปยังฐานข้อมูลเชิงสัมพันธ์แบบรีโมตและการเชื่อมต่อได้รับการปกป้อง.

4. การเชื่อมต่อไปยังไดร์เวอร์โปรแกรม application requester และการเชื่อมต่อได้รับการปกป้อง.

SQLERRD(4) ค่าเฉพาะคือ:

1. การเชื่อมต่อไปยังฐานข้อมูลเชิงสัมพันธ์แบบรีโมตและการเชื่อมต่อไม่ได้รับการปกป้อง. การอัปเดตที่สามารถ commit ได้สามารถกระทำได้ในระหว่างการเชื่อมต่อ. ซึ่งจะเกิดขึ้นเมื่อข้อใดข้อหนึ่งต่อไปนี้เป็นจริง:
 - การเชื่อมต่อเกิดขึ้นโดยการใช้ remote unit of work (RUW).
 - ถ้าการเชื่อมต่อเกิดขึ้นโดยการใช้หน่วยการทำงานแบบกระจาย (DUW) แล้วสิ่งต่อไปนี้ทั้งหมดจะเป็นจริง:
 - การเชื่อมต่อไม่ได้เป็นแบบโลคัล.
 - แอ็พพลิเคชันเซิร์ฟเวอร์ไม่สนับสนุนหน่วยการทำงานแบบกระจาย. ตัวอย่างเช่น, DB2 UDB สำหรับแอ็พพลิเคชันเซิร์ฟเวอร์ iSeries ที่มี OS/400 V3R1 หรือรีลีสก่อนหน้านี.
 - ระดับของ commitment control ของโปรแกรมที่ทำการสร้างการเชื่อมต่อไม่ใช่ *NONE.
 - ไม่ว่าจะเป็นการที่ไม่มีการเชื่อมต่อใดๆ ไปยังแอ็พพลิเคชันเซิร์ฟเวอร์อื่นๆ (รวมทั้งโลคัล) ที่สามารถทำการอัปเดตที่ commit ได้หรือการเชื่อมต่อทั้งหมดเป็นการเชื่อมต่อแบบอ่านได้อย่างเดียวไปยังแอ็พพลิเคชันเซิร์ฟเวอร์ที่ไม่สนับสนุนหน่วยของการทำงานแบบกระจาย.
 - ไม่มีไฟล์โลคัลใดที่สามารถอัปเดตได้เปิดอยู่ภายใต้ commitment control สำหรับ commitment definition.
 - ไม่มีไฟล์ DDM ใดที่ทำการอัปเดตได้และเปิดอยู่โดยใช้การเชื่อมต่อที่แตกต่างกันภายใต้ commitment control สำหรับ commitment definition.
 - ไม่มี API commitment control รีซอร์สสำหรับ commitment definition.
 - ไม่มีการเชื่อมต่อแบบป้องกันไว้ถูกลงทะเบียนไว้สำหรับ commitment definition.

ถ้าทำการรันด้วย commitment control, SQL จะลงทะเบียนรีซอร์ส DRDA ที่สามารถอัปเดตได้แบบเฟสเดียวสำหรับการเชื่อมต่อแบบรีโมต หรือรีซอร์ส DRDA ที่สามารถอัปเดตได้แบบสองเฟสสำหรับการเชื่อมต่อแบบโลคัลและ ARD.
2. การเชื่อมต่อไปยังฐานข้อมูลเชิงสัมพันธ์แบบรีโมตและการเชื่อมต่อไม่ได้รับการปกป้อง. เนื่องจากการเชื่อมต่อเป็นแบบอ่านได้เพียงอย่างเดียว. ซึ่งจะเกิดขึ้นเมื่อสิ่งต่อไปนี้เป็นจริง:
 - การเชื่อมต่อไม่ได้เป็นแบบโลคัล.
 - แอ็พพลิเคชันเซิร์ฟเวอร์ไม่สนับสนุน หน่วยการทำงานแบบกระจาย
 - สิ่งต่างๆ ต่อไปนี้อย่างน้อยหนึ่งอย่างเป็นจริง:
 - ระดับของ commitment control ของโปรแกรมที่สั่งให้เชื่อมต่อเป็น *NONE.
 - การเชื่อมต่ออื่นปรากฏต่อแอ็พพลิเคชันเซิร์ฟเวอร์ที่ไม่สนับสนุน หน่วยการทำงานแบบกระจาย และแอ็พพลิเคชันเซิร์ฟเวอร์สามารถทำการอัปเดตที่ commit ได้
 - การเชื่อมต่ออื่นปรากฏต่อแอ็พพลิเคชันเซิร์ฟเวอร์ที่สนับสนุน หน่วยการทำงานแบบกระจาย (รวมไปจนถึง โลคัล).
 - มีไฟล์ที่สามารถอัปเดตได้เปิดอยู่ภายใต้ commitment control สำหรับ commitment definition.
 - มีไฟล์ DDM ใดที่ทำการอัปเดตได้เปิดอยู่โดยใช้การเชื่อมต่อที่แตกต่างกันภายใต้ commitment control สำหรับ commitment definition.
 - ไม่มี API commitment control รีซอร์สแบบเฟสเดียวสำหรับ commitment definition.
 - มีการเชื่อมต่อแบบป้องกันที่ลงทะเบียนไว้สำหรับ definition.

ถ้าทำการรันด้วย commitment control, SQL จะลงทะเบียนรีซอร์ส DRDA แบบอ่านได้อย่างเดียวแบบเฟสเดียว.

3. การเชื่อมต่อไปยังฐานข้อมูลเชิงสัมพันธ์แบบรีโมตและการเชื่อมต่อได้รับการปกป้อง. เราไม่สามารถทราบได้ว่าสามารถทำการอัปเดตแบบ commit ได้หรือไม่. เหตุการณ์นี้จะเกิดขึ้นเมื่อทั้งหมดนี้เป็นจริง:
 - การเชื่อมต่อนั้นไม่ได้เป็นแบบโลคัล.
 - ระดับของ commitment control ของโปรแกรมที่ทำการสร้างการเชื่อมต่อไม่ใช่ *NONE.
 - แอ็พพลิเคชันเซิร์ฟเวอร์สนับสนุนทั้งหน่วยการทำงานแบบกระจาย และ commit protocol แบบสองเฟส (การเชื่อมต่อแบบป้องกัน).

ถ้าทำการรันด้วย commitment control, SQL จะลงทะเบียนรีซอร์ส DRDA ที่ไม่ทราบค่าแบบสองเฟส.

4. การเชื่อมต่อไปยังฐานข้อมูลเชิงสัมพันธ์แบบรีโมตและการเชื่อมต่อไม่ได้รับการปกป้อง. เราไม่สามารถทราบได้ว่าสามารถทำการอัปเดตแบบ commit ได้หรือไม่. เหตุการณ์นี้จะเกิดขึ้นเมื่อทั้งหมดนี้เป็นจริง:
 - การเชื่อมต่อนั้นไม่ได้เป็นแบบโลคัล.
 - แอ็พพลิเคชันเซิร์ฟเวอร์สนับสนุนหน่วยการทำงานแบบกระจาย
 - อาจเป็นเพราะแอ็พพลิเคชันเซิร์ฟเวอร์ไม่สนับสนุน commit protocol แบบสองเฟส (การเชื่อมต่อแบบป้องกัน) หรือระดับของ commitment control ของโปรแกรมที่ทำการสร้างการเชื่อมต่อเป็น *NONE อย่างใดอย่างหนึ่ง.

ถ้าทำการรันด้วย commitment control, SQL จะลงทะเบียนรีซอร์ส DRDA ที่ไม่ทราบค่าแบบเฟสเดียว.

5. การเชื่อมต่อไปยังฐานข้อมูลเชิงสัมพันธ์แบบโลคัลหรือโปรแกรม application requester driver (ARD) และการเชื่อมต่อได้รับการปกป้อง. เราไม่สามารถทราบได้ว่าสามารถทำการอัปเดตแบบ commit ได้หรือไม่. ถ้าทำการรันด้วย commitment control, SQL จะลงทะเบียนรีซอร์ส DRDA ที่ไม่ทราบค่าแบบสองเฟส.

ตารางต่อไปนี้สรุปประเภทของการเชื่อมต่อที่เป็นผลลัพธ์ของการเชื่อมต่อหน่วยการทำงานรีโมตแบบกระจาย. SQLERRD (4) จะถูกตั้งค่าในคำสั่ง CONNECT และ SET CONNECTION.

ตารางที่ 50. ข้อสรุปประเภทของการเชื่อมต่อ

เชื่อมต่อภายใต้ Commitment Control	แอ็พพลิเคชันเซิร์ฟเวอร์ สนับสนุนการ commit แบบสองเฟส	แอ็พพลิเคชันเซิร์ฟเวอร์ สนับสนุนหน่วยการทำงาน แบบกระจาย	รีซอร์สแบบเฟสเดียว อื่นๆ ที่สามารถอัปเดต ได้ ซึ่งได้รับการลง ทะเบียนไว้	SQLERRD(4)
ไม่	ไม่	ไม่	ไม่	2
ไม่	ไม่	ไม่	ใช่	2
ไม่	ไม่	ใช่	ไม่	4
ไม่	ไม่	ใช่	ใช่	4
ไม่	ใช่	ไม่	ไม่	2
ไม่	ใช่	ไม่	ใช่	2
ไม่	ใช่	ใช่	ไม่	4
ไม่	ใช่	ใช่	ใช่	4
ใช่	ไม่	ไม่	ไม่	1

ตารางที่ 50. ข้อสรุปประเภทของการเชื่อมต่อ (ต่อ)

เชื่อมต่อภายใต้ Commitment Control	แอ็พพลิเคชันเซิร์ฟเวอร์ สนับสนุนการ commit แบบสองเฟส	แอ็พพลิเคชันเซิร์ฟเวอร์ สนับสนุนหน่วยการทำงาน แบบกระจาย	รีซอร์สแบบเฟสเดียว อื่นๆ ที่สามารถอัปเดต ได้ ซึ่งได้รับการลง ทะเบียนไว้	SQLERRD(4)
ใช่	ไม่	ไม่	ใช่	2
ใช่	ไม่	ใช่	ไม่	4
ใช่	ไม่	ใช่	ใช่	4
ใช่	ใช่	ไม่	ไม่	ไม่มี ¹
ใช่	ใช่	ไม่	ใช่	ไม่มี ¹
ใช่	ใช่	ใช่	ไม่	3
ใช่	ใช่	ใช่	ใช่	3

¹DRDA ไม่อนุญาตให้ใช้การเชื่อมต่อแบบป้องกันกับแอ็พพลิเคชันเซิร์ฟเวอร์ที่สนับสนุน remote unit of work (DRDA1). รวมถึงการเชื่อมต่อ TCP/IP ของ DB2 ทั้งหมดสำหรับ iSeries.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การเข้าใช้งานฐานข้อมูลแบบรีโมตด้วย SQL แบบโต้ตอบ” ในหน้า 302

ใน SQL แบบโต้ตอบ, คุณสามารถสื่อสารกับฐานข้อมูลเชิงสัมพันธ์แบบรีโมตด้วยการใช้ข้อความ SQL CONNECT. SQL แบบโต้ตอบจะใช้ซีแมนทิกส์ CONNECT (Type 2) (หน่วยงานแบบกระจาย) สำหรับคำสั่ง CONNECT.

ข้อมูลที่เกี่ยวข้อง

Commitment control

ข้อจำกัดของตัวควบคุมการเชื่อมต่อและตัวควบคุม Commitment

การเชื่อมต่อด้วย commitment control จะมีข้อจำกัดบางอย่าง. ข้อจำกัดเหล่านี้จะเกิดเมื่อมีการรันคำสั่งโดยใช้ commitment control ในขณะที่การเชื่อมต่อถูกสร้างขึ้นโดยใช้ COMMIT(*NONE).

ถ้ารีซอร์สที่ไม่ทราบค่าหรือสามารถอัปเดตได้แบบสองเฟสถูกลงทะเบียนเอาไว้หรือรีซอร์สที่สามารถอัปเดตได้แบบเฟสเดียวถูกลงทะเบียนเอาไว้แล้ว, รีซอร์สที่สามารถอัปเดตได้แบบเฟสเดียวอีกอันหนึ่งจะไม่สามารถถูกลงทะเบียนได้.

นอกจากนั้น, เมื่อการเชื่อมต่อกลายเป็น inactive และ attribute ของงาน DDMCNV เป็น *KEEP, การเชื่อมต่อที่ไม่ได้ใช้เหล่านี้จะทำให้คำสั่ง CONNECT ในโปรแกรมที่ถูกคอมไพล์ด้วยการจัดการการเชื่อมต่อแบบ RUW ทำงานล้มเหลว.

ถ้าทำการรันด้วยการจัดการการเชื่อมต่อแบบ RUW และใช้ definition ของ job-level commitment, ก็จะมีข้อจำกัดเพิ่มขึ้นมาด้วย.

- ถ้า definition ของ job-level commitment ถูกใช้โดย activation group มากกว่าหนึ่งกลุ่ม, การเชื่อมต่อแบบ RUW ทั้งหมดจะต้องไปยังฐานข้อมูลเชิงสัมพันธ์แบบโลคัล.
- ถ้าการเชื่อมต่อเป็นแบบรีโมต, จะมี activation group เพียงกลุ่มเดียวเท่านั้นที่สามารถใช้ definition ของ job-level commitment สำหรับการเชื่อมต่อแบบ RUW.

การจำแนกสถานะของการเชื่อมต่อ

คำสั่ง CONNECT ที่ปราศจากพารามิเตอร์สามารถใช้ค้นหาว่าการเชื่อมต่อปัจจุบันสามารถอัปเดตได้หรือเป็นแบบอ่านได้
อย่างเดียวยังสำหรับหน่วยการทำงานปัจจุบัน. ค่าของ 1 หรือ 2 จะถูกส่งคืนใน SQLERRD(3) ใน SQLCA หรือ
DB2_CONNECTION_STATUS ในพื้นที่วินิจฉัย SQL.

ผลลัพธ์จะปรากฏดังนี้:

1. การอัปเดตที่สามารถ commit ได้สามารถกระทำได้ในการเชื่อมต่อสำหรับหน่วยการทำงานนั้น.

ซึ่งจะเกิดขึ้นเมื่อข้อใดข้อหนึ่งต่อไปนี้เป็นจริง:

- การเชื่อมต่อเกิดขึ้นโดยการใช้ remote unit of work (RUW).
- ถ้าการเชื่อมต่อเกิดขึ้นโดยใช้หน่วยการทำงานแบบกระจาย (DUW) และสิ่งต่อไปนี้ทั้งหมดเป็นจริง:
 - ไม่มีการเชื่อมต่อใดเกิดขึ้นกับแอ็พพลิเคชันเซิร์ฟเวอร์ที่ไม่สนับสนุนหน่วยการทำงานแบบกระจายซึ่งอัปเดตแบบ commit ได้.
 - สิ่งใดต่อไปนี้ เป็นจริง:
 - การอัปเดตที่ commit ได้จะถูกกระทำการเชื่อมต่อแบบป้องกัน, ฐานข้อมูลโลคัล, หรือการเชื่อมต่อไปยังโปรแกรม ARD.
 - มีไฟล์โลคัลที่สามารถอัปเดตได้เปิดอยู่ภายใต้ commitment control.
 - มีไฟล์ DDM ที่สามารถอัปเดตได้เปิดอยู่และใช้การเชื่อมต่อแบบป้องกัน.
 - มีรีซอร์สของ API commitment control แบบสองเฟส.
 - ไม่ได้ทำการอัปเดตที่สามารถ commit ได้.
- ถ้าการเชื่อมต่อเกิดขึ้นโดยใช้หน่วยการทำงานแบบกระจาย (DUW) และสิ่งต่อไปนี้ทั้งหมดเป็นจริง:
 - ไม่มีการเชื่อมต่ออื่นเกิดขึ้นกับแอ็พพลิเคชันเซิร์ฟเวอร์ที่ไม่สนับสนุนหน่วยการทำงานแบบกระจายซึ่งอัปเดตแบบ commit ได้.
 - การอัปเดตแบบ commit ได้ครั้งแรกถูกกระทำการเชื่อมต่อนี้หรือไม่ได้ทำการอัปเดตแบบ commit ได้.
 - ไม่มีไฟล์ DDM ที่สามารถอัปเดตได้เปิดอยู่โดยใช้การเชื่อมต่อแบบป้องกัน.
 - ไม่มีมีไฟล์โลคัลที่สามารถอัปเดตได้เปิดอยู่ภายใต้ commitment control.
 - ไม่มีรีซอร์สใดๆ ของ API commitment control แบบสองเฟส.

2. ในการเชื่อมต่อสำหรับหน่วยการทำงานนี้ คุณจะไม่สามารถทำการอัปเดตที่ commit ได้.

ซึ่งจะเกิดขึ้นเมื่อข้อใดข้อหนึ่งต่อไปนี้ เป็นจริง:

- ถ้าการเชื่อมต่อเกิดขึ้นโดยการใช้หน่วยการทำงานแบบกระจาย (DUW) และสิ่งใดสิ่งหนึ่งต่อไปนี้ เป็นจริง:
 - การเชื่อมต่อเกิดขึ้นกับแอ็พพลิเคชันเซิร์ฟเวอร์แบบอัปเดตได้ที่สนับสนุนเพียงหน่วยการทำงานแบบรีโมต.
 - การอัปเดตแบบ commit ได้ครั้งแรกจะถูกกระทำการเชื่อมต่อแบบไม่ป้องกัน.
- ถ้าการเชื่อมต่อเกิดขึ้นโดยการใช้หน่วยการทำงานแบบกระจาย (DUW) และสิ่งใดสิ่งหนึ่งต่อไปนี้ เป็นจริง:
 - การเชื่อมต่อเกิดขึ้นกับแอ็พพลิเคชันเซิร์ฟเวอร์แบบอัปเดตได้ที่สนับสนุนเพียงหน่วยการทำงานแบบรีโมต.
 - การอัปเดตแบบ commit ได้ครั้งแรกไม่ได้ถูกกระทำการเชื่อมต่อนี้.
 - มีไฟล์ DDM ที่สามารถอัปเดตได้เปิดอยู่และใช้การเชื่อมต่อแบบป้องกัน.
 - มีไฟล์โลคัลที่สามารถอัปเดตได้เปิดอยู่ภายใต้ commitment control.
 - มีรีซอร์สของ API commitment control แบบสองเฟส.

ตารางต่อไปนี้จะสรุปว่าสถานะของการเชื่อมต่อถูกกำหนดค่าได้อย่างไรโดยยึดจากค่าของชนิดการเชื่อมต่อ, ถ้ามีการเชื่อมต่อแบบสามารถอัปเดตได้กับแอปพลิเคชันเซิร์ฟเวอร์ซึ่งสนับสนุนเพียงหน่วยการทำงานแบบรีโมตเท่านั้น, และที่ซึ่งการอัปเดตแบบ commit ได้เกิดขึ้นเป็นครั้งแรก.

ตารางที่ 51. ข้อสรุปการจำแนกค่าสถานะของการเชื่อมต่อ

วิธีการเชื่อมต่อ	การเชื่อมต่อเกิดขึ้นกับหน่วยการทำงานแบบรีโมตที่สามารถอัปเดตได้ของแอปพลิเคชันเซิร์ฟเวอร์	สถานการณ์ที่จะเกิดการอัปเดตที่สามารถ commit ได้ ¹	SQLERRD(3) หรือ DB2_CONNECTION_STATUS
RUW			1
DUW	ใช่		2
DUW	ไม่	ไม่มีการอัปเดต	1
DUW	ไม่	เฟสเดียว	2
DUW	ไม่	การเชื่อมต่อนี้	1
DUW	ไม่	สองเฟส	1

¹ คำศัพท์ในคอลัมน์นี้จะถูกระบุเป็น:

- *No updates* ระบุว่าไม่มีการอัปเดตที่ commit ได้, ไม่มีไฟล์ DDM ใดๆ เปิดไว้สำหรับการอัปเดตโดยใช้การเชื่อมต่อแบบป้องกัน, ไม่มีไฟล์ล็อกใดๆ เปิดไว้สำหรับการอัปเดต, และไม่มี commitment control API ใดๆ ถูกลงทะเบียนไว้.
- *One-phase* ระบุว่าการอัปเดตแรกที่สามารถ commit ได้ถูกกระทำโดยใช้การเชื่อมต่อแบบไม่ป้องกัน หรือไฟล์ DDM เปิดไว้สำหรับการอัปเดตโดยใช้การเชื่อมต่อแบบไม่ป้องกัน.
- *Two-phase* ระบุว่ามีการอัปเดตที่สามารถ commit ได้บนแอปพลิเคชันเซิร์ฟเวอร์แบบสองเฟสที่มีหน่วยการทำงานแบบกระจาย, ไฟล์ DDM จะเปิดไว้สำหรับการอัปเดตโดยใช้การเชื่อมต่อแบบป้องกัน, commitment control API ถูกลงทะเบียนไว้, หรือไฟล์ล็อกจะเปิดไว้สำหรับกัอัปเดตภายใต้ commitment control.

ถ้ามีการพยายามอัปเดตที่สามารถ commit ได้บนการเชื่อมต่อ, หน่วยการทำงานจะถูกระบุให้อยู่ในสถานะที่ต่อมีการ rollback. ถ้าเป็นดังนั้น, คำสั่งเดียวที่ใช้ได้คือ ROLLBACK ; ส่วนคำสั่งอื่นๆ ทั้งหมดจะส่งผลลัพธ์เป็น SQLCODE -918.

ข้อควรพิจารณาในการเชื่อมต่อหน่วยการทำงานแบบกระจาย

เมื่อทำการเชื่อมต่อในแอปพลิเคชันที่มีหน่วยการทำงานแบบกระจาย, มีข้อควรพิจารณาอยู่หลายข้อด้วยกัน.

- ถ้าหน่วยการทำงานจะกระทำการอัปเดตที่แอปพลิเคชันเซิร์ฟเวอร์มากกว่าหนึ่งเซิร์ฟเวอร์โดยมีการใช้ commitment control, การเชื่อมต่อทั้งหมดรวมทั้งการอัปเดตจะถูกกระทำโดยใช้ commitment control. ถ้าการเชื่อมต่อถูกทำขึ้นโดยไม่ใช้ commitment control และมีการอัปเดตแบบ commit ได้เกิดขึ้นในภายหลัง, ผลลัพธ์ที่ได้มาจะเป็นการเชื่อมต่อแบบอ่านได้อย่างเดียว.
- รีซอร์สที่เป็น non-SQL commit อื่นๆ, เช่น ไฟล์ล็อก, ไฟล์ DDM, และรีซอร์ส commitment control API, จะส่งผลกระทบต่อสถานะในการอัปเดตและการอ่านได้เพียงอย่างเดียวของการเชื่อมต่อนั้น.
- ถ้าการเชื่อมต่อโดยใช้ commitment control ไปยังแอปพลิเคชันเซิร์ฟเวอร์ที่ไม่สนับสนุนหน่วยการทำงานแบบกระจาย (ตัวอย่างเช่น, V4R5 iSeries ที่ใช้ TCP/IP), การเชื่อมต่อนั้นจะเป็นแบบสามารถอัปเดตได้ หรือแบบอ่านได้อย่างเดียวอย่างใดอย่างหนึ่ง. ถ้าการเชื่อมต่อนั้นสามารถอัปเดตได้ ก็จะเป็นเพียงการเชื่อมต่อที่สามารถอัปเดตได้เท่านั้น. ใน V5R3,

การอัปเดตที่เป็นผลมาจากทริกเกอร์หรือฟังก์ชันที่ใช้เป็นผู้กำหนดในระหว่างการทำเคอร์รี่ฐานข้อมูลจะ ถูกพิจารณาในระหว่างการดำเนินการ commit DRDA แบบสองเฟส.

การสิ้นสุดการเชื่อมต่อ

เนื่องจากการเชื่อมต่อแบบรีโมตมีการใช้รีซอร์ส, การเชื่อมต่อที่จะไม่ถูกใช้ก็ จึงควรจบการทำงานให้เร็วที่สุด. การเชื่อมต่อ สามารถสิ้นสุดได้ไม่ว่าจะโดยนัย หรือโดยชัดเจน.

ส่วนการสิ้นสุดการเชื่อมต่ออย่างชัดเจนทำได้โดยการใช้คำสั่ง DISCONNECT หรือ RELEASE อย่างใดอย่างหนึ่งและตาม ด้วยการ COMMIT ที่สำเร็จสมบูรณ์. คำสั่ง DISCONNECT ใช้ได้กับการเชื่อมต่อแบบป้องกันหรือการเชื่อมต่อโลคัล. คำสั่ง DISCONNECT จะทำให้การเชื่อมต่อสิ้นสุดเมื่อรับคำสั่งนี้. คำสั่ง RELEASE ใช้กับการเชื่อมต่อแบบป้องกันหรือไม่ก็ได้. เมื่อ รับคำสั่ง RELEASE, การเชื่อมต่อจะไม่สิ้นสุดลงแต่จะถูกระบุให้อยู่ในสถานะ release แทน. ซึ่งยังสามารถใช้การเชื่อมต่อ นั้น ได้อยู่. และการเชื่อมต่อนั้นจะไม่สิ้นสุดจนกว่าจะรับ COMMIT สำเร็จ. การ ROLLBACK หรือการ COMMIT ที่ไม่สำเร็จจะทำให้ การเชื่อมต่อที่อยู่ในสถานะ release ไม่สิ้นสุดลง.

เมื่อเกิดการเชื่อมต่อแบบรีโมตขึ้น, การเชื่อมต่อเครือข่ายของ DDM (การสนทนา (ระหว่างโปรแกรม) APPC หรือการเชื่อมต่อแบบ TCP/IP) จะถูกนำมาใช้. เมื่อการเชื่อมต่อของ SQL สิ้นสุดลง, การเชื่อมต่อเครือข่ายอาจจะอยู่ในสถานะที่ยังไม่ได้ใช้งานหรือหลุดไป. ไม่ว่าจะเป็นการเชื่อมต่อระบบเครือข่ายจะหลุดไปหรือถูกระบุให้อยู่ในสถานะที่ยังไม่ได้ใช้งานจะขึ้นอยู่กับแอททริบิวต์งานของ DDMCNV. ถ้าค่าของแอททริบิวต์งานเป็น is *KEEP และมีการเชื่อมต่อไปยังเซิร์ฟเวอร์ iSeries, การเชื่อมต่อนั้นจะอยู่ในสถานะไม่ใช้งาน. ถ้าค่าของแอททริบิวต์งานเป็น *DROP และมีการเชื่อมต่อไปยังอีกหนึ่งเซิร์ฟเวอร์ iSeries, การเชื่อมต่อจะหลุดไป. ถ้ามีการเชื่อมต่อไปยังเซิร์ฟเวอร์ non-iSeries, การเชื่อมต่อจะหลุดไปทุกครั้ง. สถานะ *DROP ควรจะเกิดขึ้นในสถานการณ์ดังต่อไปนี้:

- เมื่อการรักษาการเชื่อมต่อที่ไม่ได้ใช้นั้นทำให้เสียค่าใช้จ่ายมากและมีแนวโน้มว่าจะไม่ได้ใช้การเชื่อมต่อนั้นในระยะเวลาอันใกล้.
- เมื่อทำการรันโปรแกรมหลายๆ โปรแกรมด้วยกัน, บางโปรแกรมจะคอมไพล์ด้วยการจัดการการเชื่อมต่อแบบ RUW และ บางโปรแกรมจะ คอมไพล์ด้วยการจัดการการเชื่อมต่อแบบ DUW. การรันโปรแกรมที่ถูกคอมไพล์ด้วยการจัดการการเชื่อมต่อ RUW ไปยังตำแหน่งรีโมตจะล้มเหลวหากมีการเชื่อมต่อแบบป้องกันอยู่แล้ว.
- เมื่อทำการรันด้วยการเชื่อมต่อแบบป้องกันไว้โดยใช้ DDM หรือ DRDA อย่างใดอย่างหนึ่ง. จะทำให้เกิดค่าใช้จ่ายเพิ่มเติมในการ commit และ rollback สำหรับการเชื่อมต่อแบบป้องกันที่ไม่ได้ใช้งาน.

คำสั่งการเชื่อมต่อแบบ Reclaim DDM (RCLDDMCNV) สามารถใช้จบการเชื่อมต่อที่ไม่ใช้งาน, ถ้าอยู่ที่ขอบเขตของการ commit.

สิ่งอ้างอิงที่เกี่ยวข้อง

“การจัดการเชื่อมต่อโดยนัยสำหรับ activation group ดีฟอลต์” ในหน้า 321 application requester สามารถเชื่อมต่อโดยนัยกับแอ็พพลิเคชันเซิร์ฟเวอร์ได้.

“การจัดการการเชื่อมต่อโดยนัยสำหรับ activation group ที่ไม่ใช่ดีฟอลต์” ในหน้า 322 Application requester สามารถเชื่อมต่อโดยนัยไปยังแอ็พพลิเคชันเซิร์ฟเวอร์ได้. ซึ่งจะเกิดขึ้นเมื่อ application requester ตรวจพบว่าคำสั่ง SQL แรกที่ส่งออกมาจาก activation group ไม่ใช่คำสั่ง CONNECT ที่มีพารามิเตอร์.

หน่วยการทำงานแบบกระจาย

Distributed unit of work (DUW) อนุญาตให้มีการเข้าใช้งานแอปพลิเคชันเซิร์ฟเวอร์หลายเซิร์ฟเวอร์ภายในหน่วยการทำงานเดียวกัน. คำสั่ง SQL แต่ละคำสั่งสามารถเข้าใช้งานได้เพียงหนึ่งแอปพลิเคชันเซิร์ฟเวอร์เท่านั้น. ในขณะที่การใช้หน่วยการทำงานแบบกระจายจะอนุญาตให้ทำการเปลี่ยนแปลงที่หลายๆ แอปพลิเคชันเซิร์ฟเวอร์ในการ commit หรือ rollback ภายในหน่วยการทำงานเดียว.

การจัดการการเชื่อมต่อหน่วยการทำงานแบบกระจาย

คำสั่ง CONNECT, SET CONNECTION, DISCONNECT, และ RELEASE จะถูกใช้เพื่อจัดการการเชื่อมต่อในสถานะแวดล้อม DUW.

หน่วยการทำงานแบบกระจาย CONNECT จะถูกรันเมื่อโปรแกรมถูกคอมไพล์โดยใช้ RDBCNNMTH(*DUW), ซึ่งเป็นค่าดีฟอลต์. รูปแบบของคำสั่ง CONNECT นี้จะไม่ตัดการเชื่อมต่อที่มีอยู่แต่จะระบุการเชื่อมต่อที่มีอยู่ก่อนให้อยู่ในสถานะที่ถูกระงับไว้แทน. ฐานข้อมูลเชิงสัมพันธ์ที่ระบุบนคำสั่ง CONNECT จะกลายเป็นการเชื่อมต่อปัจจุบัน. คำสั่ง CONNECT สามารถใช้ในการเริ่มการเชื่อมต่อใหม่; ถ้าต้องการที่จะสลับระหว่างการเชื่อมต่อที่มีอยู่, คำสั่ง SET CONNECTION จะต้องถูกนำมาใช้. เนื่องจากการเชื่อมต่อใช้รีซอร์สของระบบ, การเชื่อมต่อเหล่านั้นจึงควรที่จะจบลงเมื่อไม่จำเป็นที่จะต้องใช้อีก. คำสั่ง RELEASE หรือ DISCONNECT สามารถนำมาใช้จบการเชื่อมต่อได้. คำสั่ง RELEASE จะต้องตามด้วยการ commit ที่สำเร็จเพื่อจบการเชื่อมต่อ.

ต่อไปนี้เป็นตัวอย่างของโปรแกรมภาษาซีที่รันอยู่ใน DUW environment ที่ใช้ commitment control.


```

....
EXEC SQL WHENEVER SQLERROR GO TO done;
EXEC SQL WHENEVER NOT FOUND GO TO done;
....
EXEC SQL
DECLARE C1 CURSOR WITH HOLD FOR
  SELECT PARTNO, PRICE
    FROM PARTS
     WHERE SITES_UPDATED = 'N'
     FOR UPDATE OF SITES_UPDATED;
/* Connect to the systems */
EXEC SQL CONNECT TO LOCALSYS;
EXEC SQL CONNECT TO SYSB;
EXEC SQL CONNECT TO SYSC;
/* Make the local system the current connection */
EXEC SQL SET CONNECTION LOCALSYS;
/* Open the cursor */
EXEC SQL OPEN C1;
while (SQLCODE==0)
{
  /* Fetch the first row */
  EXEC SQL FETCH C1 INTO :partnumber,:price;
  /* Update the row which indicates that the updates have been
    propagated to the other sites */
  EXEC SQL UPDATE PARTS SET SITES_UPDATED='Y'
    WHERE CURRENT OF C1;
  /* Check if the part data is on SYSB */
  if ((partnumber > 10) && (partnumber < 100))
  {
    /* Make SYSB the current connection and update the price */
    EXEC SQL SET CONNECTION SYSB;
    EXEC SQL UPDATE PARTS
      SET PRICE=:price
      WHERE PARTNO=:partnumber;
  }

  /* Check if the part data is on SYSC */
  if ((partnumber > 50) && (partnumber < 200))
  {
    /* Make SYSC the current connection and update the price */
    EXEC SQL SET CONNECTION SYSC;
    EXEC SQL UPDATE PARTS
      SET PRICE=:price
      WHERE PARTNO=:partnumber;
  }

  /* Commit the changes made at all 3 sites */
  EXEC SQL COMMIT;
  /* Set the current connection to local so the next row
    can be fetched */
  EXEC SQL SET CONNECTION LOCALSYS;
}
done:

EXEC SQL WHENEVER SQLERROR CONTINUE;
/* Release the connections that are no longer being used */
EXEC SQL RELEASE SYSB;
EXEC SQL RELEASE SYSC;
/* Close the cursor */
EXEC SQL CLOSE C1;
/* Do another commit which will end the released connections.
  The local connection is still active because it was not

```

ในโปรแกรมนี้, มีแอ็พพลิเคชันเซิร์ฟเวอร์สามตัวที่แอ็คทีฟ: LOCALSYS ซึ่งเป็นระบบโลคัล, และระบบรีโมตอีกสองตัว, SYSB และ SYSC. SYSB และ SYSC สนับสนุนหน่วยการทำงานแบบกระจาย และ two-phase commit.

เริ่มแรกการเชื่อมต่อทั้งหมดจะถูกทำให้แอ็คทีฟโดยการใช้อำนาจ CONNECT สำหรับแต่ละแอ็พพลิเคชันเซิร์ฟเวอร์ที่เกี่ยวข้องในการดำเนินงาน. เมื่อใช้ DUW, คำสั่ง CONNECT จะไม่ตัดการเชื่อมต่อที่มีอยู่ก่อนหน้านี้, แต่จะระบุการเชื่อมต่อที่มีอยู่ก่อนเป็นสถานะที่ถูกระงับไว้แทน. หลังจากที่แอ็พพลิเคชันเซิร์ฟเวอร์ทั้งหมดได้ถูกเชื่อมต่อแล้ว, การเชื่อมต่อแบบโลคัลจะถูกทำให้เป็นการเชื่อมต่อปัจจุบันโดยใช้อำนาจ SET CONNECTION. เคอร์เซอร์ก็จะถูกเปิดและแถวข้อมูลแรกก็จะถูกดึงออกมา. จากนั้นระบบจะตรวจสอบว่าต้องอัปเดตข้อมูลที่แอ็พพลิเคชันเซิร์ฟเวอร์ใด. ถ้า SYSB ต้องการการอัปเดต, SYSB จะถูกทำให้เป็นการเชื่อมต่อปัจจุบันโดยใช้อำนาจ SET CONNECTION และการอัปเดตจะถูกทำ. และจะมีการดำเนินการแบบเดียวกันนี้สำหรับ SYSC. การเปลี่ยนแปลงจะถูก commit หลังจากนั้น.

เนื่องจากการ commit แบบสองเฟสกำลังถูกใช้, การเปลี่ยนแปลงจะถูก commit ที่ระบบโลคัลและระบบรีโมตอีกสองระบบอย่างแน่นอน. เนื่องจากเคอร์เซอร์ถูกประกาศให้เป็น WITH HOLD, ทำให้ยังเปิดหลังจากที่ทำการ commit. จากนั้นการเชื่อมต่อปัจจุบันก็จะถูกเปลี่ยนเป็นระบบโลคัลทำให้แถวถัดไปของข้อมูลสามารถถูกดึงออกมาได้. การดึงข้อมูล, การอัปเดต, และการ commit แบบนี้จะถูกทำซ้ำไปเรื่อยๆ จนกระทั่งข้อมูลทั้งหมดถูกประมวลผล.

หลังจากข้อมูลทั้งหมดถูกดึงออกมา, การเชื่อมต่อสำหรับระบบรีโมตทั้งสองจะถูก release. โดยไม่จบการเชื่อมต่อเนื่องจากการเป็นการเชื่อมต่อแบบป้องกันอยู่. หลังจากที่มีการเชื่อมต่อถูก release, จะมีการ commit เพื่อที่จะสิ้นสุดการเชื่อมต่อทั้งหมด. ระบบโลคัลยังคงเชื่อมต่อและทำการประมวลผลไปเรื่อยๆ.

การตรวจสอบสถานะของการเชื่อมต่อ

ถ้ามีการรันในสถานะแวดล้อมที่ยอมให้มีการเชื่อมต่อแบบอ่านอย่างเดียวได้, สถานะของการเชื่อมต่อจะต้องถูกตรวจสอบก่อนจะทำการอัปเดตที่สามารถ commit ได้. การทำเช่นนี้จะช่วยป้องกันหน่วยการทำงานจากการเข้าสู่สถานะที่ต้องมีการ rollback.

ตัวอย่างภาษา COBOL ต่อไปนี้แสดงวิธีการตรวจสอบสถานะการเชื่อมต่อ.

```
...
EXEC SQL
SET CONNECTION SYS5
END-EXEC.
...
* Check if the connection is updatable.
EXEC SQL CONNECT END-EXEC.
* If connection is updatable, update sales information otherwise
* inform the user.
IF SQLERRD(3) = 1 THEN
EXEC SQL
INSERT INTO SALES_TABLE
VALUES (:SALES-DATA)
END-EXEC
ELSE
DISPLAY 'Unable to update sales information at this time'.
...
```

รูปที่ 6. ตัวอย่างของการตรวจสอบสถานะการเชื่อมต่อ

เคอร์เซอร์ และคำสั่งที่เตรียมไว้

เคอร์เซอร์และคำสั่งที่เตรียมไว้แล้วจะถูกระบุในหน่วยการคอมไพล์และการเชื่อมต่อด้วยเช่นกัน.

การกำหนดหน่วยการคอมไพล์หมายถึงการที่โปรแกรมซึ่งเรียกจากอีกโปรแกรมที่ถูกคอมไพล์แยกกันไม่สามารถใช้เคอร์เซอร์หรือคำสั่งที่เตรียมไว้ซึ่งถูกเปิดหรือเตรียมโดยการเรียกโปรแกรม. การระบุการเชื่อมต่อหมายถึงแต่ละการเชื่อมต่อภายในโปรแกรมสามารถมี instance ของเคอร์เซอร์หรือคำสั่งที่เตรียมไว้แยกจากกัน.

ตัวอย่างหน่วยการทำงานแบบกระจายต่อไปนี้แสดงวิธีเปิดเคอร์เซอร์ตัวเดียวกันสำหรับการเชื่อมต่อที่ต่างกัน, ทำให้เกิด instance ของเคอร์เซอร์ C1 สองอย่าง.

```
.....
EXEC SQL DECLARE C1 CURSOR FOR
        SELECT * FROM CORPDATA.EMPLOYEE;
        /* Connect to local and open C1 */
EXEC SQL CONNECT TO LOCALSYS;
EXEC SQL OPEN C1;
        /* Connect to the remote system and open C1 */
EXEC SQL CONNECT TO SYSA;
EXEC SQL OPEN C1;
        /* Keep processing until done */
        while (NOT_DONE) {
                /* Fetch a row of data from the local system */
EXEC SQL SET CONNECTION LOCALSYS;
EXEC SQL FETCH C1 INTO :local_emp_struct;
                /* Fetch a row of data from the remote system */
EXEC SQL SET CONNECTION SYSA;
EXEC SQL FETCH C1 INTO :rmt_emp_struct;
                /* Process the data */
        .....
}
        /* Close the cursor on the remote system */
EXEC SQL CLOSE C1;
        /* Close the cursor on the local system */
EXEC SQL SET CONNECTION LOCALSYS;
EXEC SQL CLOSE C1;
.....
```

รูปที่ 7. ตัวอย่างของเคอร์เซอร์ในโปรแกรม DUW

ไดรเวอร์โปรแกรม application requester

เพื่อให้เข้าถึงฐานข้อมูลซึ่งเตรียมไว้โดยผลิตภัณฑ์ที่ใช้ DRDA, DB2 UDB for iSeries นำเสนออินเตอร์เฟซสำหรับการเขียนโปรแกรมทางออกบน application requester ของ DB2 UDB for iSeries ในการประมวลผล SQL request. โปรแกรมทางออกดังกล่าวเรียกว่า *ไดรเวอร์ application requester*.

เซิร์ฟเวอร์ทำการเรียกโปรแกรม ARD ในขณะที่ปฏิบัติการต่อไปนี้:

- ในช่วงของการสร้างแพ็คเกจโดยใช้คำสั่ง CRTSQLPKG หรือ CRTSQLxxx, เมื่อพารามิเตอร์ฐานข้อมูลเชิงสัมพันธ์ (RDB) ตรงกับชื่อ RDB ที่สอดคล้องกับโปรแกรม ARD.
- การประมวลผลคำสั่ง SQL เมื่อมีการเชื่อมต่อปัจจุบันไปยังชื่อ RDB ที่สอดคล้องกับโปรแกรม ARD.

การเรียกเหล่านี้อนุญาตให้โปรแกรม ARD ส่งผ่านคำสั่ง SQL และข้อมูลเกี่ยวกับคำสั่งนั้นไปยังฐานข้อมูลเชิงสัมพันธ์แบบรีโมต และส่งผลลัพธ์กลับมายังระบบ. จากนั้นข้อมูลก็จะส่งค่ากลับมายังแอปพลิเคชันหรือผู้ใช้. การเข้าไปใช้ฐานข้อมูลเชิงสัมพันธ์โดยโปรแกรม ARD จะคล้ายกันกับการเข้าไปในแอปพลิเคชันเซิร์ฟเวอร์ DRDA ในสภาวะแวดล้อมที่ไม่เหมือนกัน. อย่างไรก็ตาม, สภาวะแวดล้อม ARD จะไม่สนับสนุนฟังก์ชัน DRDA. ตัวอย่างของฟังก์ชันที่ไม่ได้การสนับสนุน ได้แก่ อ็อบเจกต์ขนาดใหญ่ (LOBs) และรหัสผ่านที่มีขนาดยาว (passphrases).

การรับมือกับปัญหา

กลยุทธ์หลักสำหรับการดักจับและรายงานผลข้อมูลที่เกิดผิดพลาดสำหรับฟังก์ชันฐานข้อมูลแบบกระจายจะมีชื่อว่า *first failure data capture* – *FFDC*.

จุดประสงค์ของการสนับสนุน FFDC นี้ก็เพื่อเป็นการจัดเตรียมข้อมูลที่ต้องเกี่ยวกับข้อผิดพลาดที่ตรวจพบในส่วนประกอบ DDM ของระบบ i5/OS ที่ซึ่งมีการสร้าง APAR (Authorized Program Analysis Report). ด้วยการทำงานของฟังก์ชันนี้, โครงสร้างหลักและ data stream ของ DDM จะถูกดักจับโดยอัตโนมัติไปยังไฟล์สพูล. ข้อมูลที่ผิดพลาด 1024 ไบต์แรกจะถูกบันทึกไว้ในบันทึกข้อผิดพลาดของระบบ. การดักจับข้อมูลข้อผิดพลาดอัตโนมัติเกี่ยวกับข้อผิดพลาดครั้งแรกหมายถึงความล้มเหลวจะต้องไม่ถูกสร้างขึ้นอีกเพื่อให้ลูกค้าแจ้งเข้า. FFDC จะแอ็คทีฟทั้งในฟังก์ชัน application requester และแอปพลิเคชันเซิร์ฟเวอร์ในส่วนประกอบ DDM ของ i5/OS. อย่างไรก็ตาม, สำหรับข้อมูล FFDC ที่ต้องการบันทึกการทำงานไว้, ค่ากำหนดของระบบในส่วนประกอบ QSFWERRLOG จะต้องถูกตั้งให้เป็น *LOG.

หมายเหตุ: คำ SQLCODEs ที่เป็นลบจะถูกดักจับเป็นบางค่าเท่านั้น; ได้แก่ คำที่ใช้ในการผลิต APAR. สำหรับข้อมูลเพิ่มเติมเกี่ยวกับการรับมือกับปัญหาในปฏิบัติการของฐานข้อมูลเชิงสัมพันธ์แบบกระจาย, โปรดดู *แนวทางการแก้ปัญหาของฐานข้อมูลแบบกระจาย SC26-4782*

เมื่อข้อผิดพลาดของ SQL ถูกตรวจพบ, SQLCODE พร้อมด้วย SQLSTATE ที่เกี่ยวข้องจะถูกส่งคืนมาใน SQLCA.

ข้อมูลที่เกี่ยวข้อง

ข้อความและโค้ด SQL

ข้อควรพิจารณาสำหรับโปรแกรมเมอร์ของ DRDA ที่บันทึกไว้

เซิร์ฟเวอร์ iSeries DRDA สนับสนุนการส่งค่าชุดผลลัพธ์คืนมาจากโปรแกรมเมอร์ที่บันทึกไว้.

ชุดผลลัพธ์สามารถสร้างในโปรแกรมเมอร์ที่บันทึกเอาไว้โดยการเปิดเคอร์เซอร์ SQL จำนวนหนึ่งหรือมากกว่าที่เกี่ยวข้องกับคำสั่ง SQL SELECT. นอกเหนือจากนั้น, ค่าสูงสุดของอะเรย์ชุดผลลัพธ์สามารถถูกส่งค่าคืนมาได้เพียงหนึ่งชุด. ก่อนรุ่น V5R3, ณ เวลาใดเวลาหนึ่งโปรแกรมเมอร์ที่บันทึกไว้หนึ่งสามารถมีเปิดเพียงหนึ่ง instance ของเคียวรีเท่านั้น. เดียวนี้ สามารถเรียกโปรแกรมเมอร์ที่บันทึกไว้ได้หลายครั้งโดยไม่ต้องปิดชุดผลลัพธ์เคอร์เซอร์ ดังนั้น จึงสามารถเปิดเคียวรีได้มากกว่าหนึ่ง instance พร้อมกัน.

หลักการที่เกี่ยวข้อง

“สตอร์โปรแกรมเมอร์” ในหน้า 143

โปรแกรมเมอร์ (ซึ่งมักเรียกว่า สตอร์โปรแกรมเมอร์) คือโปรแกรมที่สามารถเรียกขึ้นมาเพื่อปฏิบัติงาน ซึ่งมีทั้งข้อความภาษาโฮสต์และคำสั่ง SQL. โปรแกรมเมอร์ใน SQL มีข้อดีเหมือนกับโปรแกรมเมอร์ในภาษาโฮสต์.

สิ่งอ้างอิงที่เกี่ยวข้อง

โปรแกรมมิงฐานข้อมูลแบบกระจาย

ข้อมูลที่เกี่ยวข้อง

SET RESULT SETS statement

CREATE PROCEDURE (SQL)

CREATE PROCEDURE (External)

ข้อมูลอ้างอิง

ข้อมูลอ้างอิงสำหรับโปรแกรมมิง SQL จะประกอบด้วยตัวอย่างตารางและคำสั่ง CL.

DB2 UDB for iSeries ตารางตัวอย่าง

หัวข้อนี้ประกอบด้วยตารางตัวอย่างที่ใช้อ้างอิง และใช้ในหัวข้อนี้ รวมทั้งหัวข้อ SQL Reference.

ที่ให้มาพร้อมกับตารางนี้คือ คำสั่ง SQL สำหรับสร้างตาราง.

ตารางนี้ประกอบด้วยข้อมูลซึ่งให้รายละเอียดของพนักงาน, แผนก, โครงการ, และการดำเนินการ, ในลักษณะเป็นกลุ่ม. ข้อมูลนี้มีโปรแกรมตัวอย่างซึ่งแสดงให้เห็น คุณลักษณะบางประการของไลเซนส์โปรแกรม DB2 UDB Query Manager and SQL Development Kit. ตัวอย่างทั้งหมดที่อยู่ในตาราง จะอยู่ในแบบแผนที่ชื่อว่า CORPDATA (สำหรับข้อมูลบริษัท).

นอกจากนี้ยังมีการส่งโปรซีเจอร์ซึ่งเป็นส่วนหนึ่งของระบบที่มีคำสั่ง DDL เพื่อสร้างตารางทั้งหมดนี้, และคำสั่ง INSERT เพื่อบรรจุตาราง. โดยโปรซีเจอร์จะสร้างแบบแผนที่ระบุในการเรียกไปยังโปรซีเจอร์. เนื่องจากเป็นโปรซีเจอร์นี้เป็นแบบ SQL ที่เก็บไว้ภายนอก, จึงสามารถเรียกได้จากอินเทอร์เฟซ SQL ใดๆ ก็ได้, รวมถึง SQL แบบโต้ตอบและ iSeries Navigator. การเรียกโปรซีเจอร์ที่คุณจะสร้างแบบแผน *SAMPLE*, ให้ใช้คำสั่งต่อไปนี้:

```
CALL QSYS.CREATE_SQL_SAMPLE ('SAMPLE')
```

ต้องใช้ตัวพิมพ์ใหญ่ระบุชื่อแบบแผน. และต้องไม่มีแบบแผนนั้นอยู่ก่อน.

หมายเหตุ: ในตารางตัวอย่างเหล่านี้, เครื่องหมายคำถาม (?) แสดงถึงค่าศูนย์.

หลักการที่เกี่ยวข้อง

“การโปรแกรม SQL”, ในหน้า 1

หัวข้อเหล่านี้จะอธิบายถึงการนำ iSeries Structured Query Language (SQL) ของเซิร์ฟเวอร์ DB2 UDB for iSeries ไปปฏิบัติและไลเซนส์โปรแกรม DB2 UDB Query Manager and SQL Development Kit เวอร์ชัน 5.

สิ่งอ้างอิงที่เกี่ยวข้อง

“Referential integrity และตาราง” ในหน้า 18

Referential integrity คือเงื่อนไขสำหรับชุดของตารางในฐานข้อมูล ซึ่งใช้ในการอ้างอิงทั้งหมดจากตารางหนึ่งไปยังอีกตารางหนึ่ง.

“ตัวอย่าง: กฎ DELETE แบบต่อเรียง” ในหน้า 108

“การดึงข้อมูลออก (FETCH) แบบหลายแถวโดยใช้พื้นที่หน่วยเก็บของแถว” ในหน้า 268
แอปพลิเคชันต้องกำหนดพื้นที่หน่วยเก็บของแถวและพื้นที่รายละเอียดที่เกี่ยวข้องก่อนที่แอปพลิเคชันจะสามารถใช้
FETCH แบบหลายแถวที่มีพื้นที่หน่วยเก็บของแถวได้. พื้นที่หน่วยเก็บของแถว คือตัวแปรโฮสต์ซึ่งถูกกำหนดในแอปพลิเคชันโปรแกรม.

ตารางแผนก (DEPARTMENT)

ตารางแผนกจะแสดงรายละเอียดของแต่ละแผนกในบริษัท และระบุชื่อผู้จัดการ และแผนกที่ต้องรายงาน.

สร้างตารางแผนกด้วย คำสั่ง CREATE TABLE และ ALTER TABLE ต่อไปนี้:

```
CREATE TABLE DEPARTMENT
  (DEPTNO   CHAR(3)           NOT NULL,
   DEPTNAME VARCHAR(36)       NOT NULL,
   MGRNO    CHAR(6)           ,
   ADMRDEPT CHAR(3)           NOT NULL,
   LOCATION CHAR(16),
   PRIMARY KEY (DEPTNO))
```

```
ALTER TABLE DEPARTMENT
  ADD FOREIGN KEY ROD (ADMRDEPT)
  REFERENCES DEPARTMENT
  ON DELETE CASCADE
```

foreign key ต่อไปนี้จะถูกเพิ่มในภายหลัง.

```
ALTER TABLE DEPARTMENT
  ADD FOREIGN KEY RDE (MGRNO)
  REFERENCES EMPLOYEE
  ON DELETE SET NULL
```

ดรรชนีต่อไปนี้จะถูกสร้าง.

```
CREATE UNIQUE INDEX XDEPT1
  ON DEPARTMENT (DEPTNO)
```

```
CREATE INDEX XDEPT2
  ON DEPARTMENT (MGRNO)
```

```
CREATE INDEX XDEPT3
  ON DEPARTMENT (ADMRDEPT)
```

alias ต่อไปนี้จะถูกสร้างให้กับตาราง.

```
CREATE ALIAS DEPT FOR DEPARTMENT
```

ตารางต่อไปนี้จะแสดงเนื้อหาของคอลัมน์.

ตารางที่ 52. คอลัมน์ของตารางแผนก

ชื่อคอลัมน์	รายละเอียด
DEPTNO	หมายเลขแผนกหรือ ID.
DEPTNAME	ชื่อที่อธิบายถึงกิจกรรมทั่วไปของแผนก.

ตารางที่ 52. คอลัมน์ของตารางแผนก (ต่อ)

ชื่อคอลัมน์	รายละเอียด
MGRNO	หมายเลขพนักงาน (EMPNO) ของผู้จัดการแผนก.
ADMRDEPT	แผนก (DEPTNO) ซึ่งเป็นหน่วยงานที่แผนกนี้รายงานตรงไปยัง; แผนกที่อยู่ในระดับสูงสุดจะรายงานตรงมายังแผนกของตนเอง.
LOCATION	ที่ตั้งของแผนก.

DEPARTMENT:

รายการของข้อมูลที่สมบูรณ์ในตาราง DEPARTMENT.

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00	?
B01	PLANNING	000020	A00	?
C01	INFORMATION CENTER	000030	A00	?
D01	DEVELOPMENT CENTER	?	A00	?
D11	MANUFACTURING SYSTEMS	000060	D01	?
D21	ADMINISTRATION SYSTEMS	000070	D01	?
E01	SUPPORT SERVICES	000050	A00	?
E11	OPERATIONS	000090	E01	?
E21	SOFTWARE SUPPORT	000100	E01	?
F22	BRANCH OFFICE F2	?	E01	?
G22	BRANCH OFFICE G2	?	E01	?
H22	BRANCH OFFICE H2	?	E01	?
I22	BRANCH OFFICE I2	?	E01	?
J22	BRANCH OFFICE J2	?	E01	?

ตารางพนักงาน (EMPLOYEE)

ตารางพนักงานจะแสดงข้อมูลพนักงานทั้งหมดตามหมายเลขพนักงาน และข้อมูลประจำตัวทั่วไป.

ตารางพนักงานสามารถสร้างด้วยคำสั่ง CREATE TABLE และ ALTER TABLE ต่อไปนี้:

```
CREATE TABLE EMPLOYEE
  (EMPNO      CHAR(6)          NOT NULL,
   FIRSTNME  VARCHAR(12)     NOT NULL,
   MIDINIT    CHAR(1)         NOT NULL,
   LASTNAME   VARCHAR(15)    NOT NULL,
   WORKDEPT   CHAR(3)        ,
```

```

PHONENO    CHAR(4)          ,
HIREDATE   DATE          ,
JOB        CHAR(8)       ,
EDLEVEL    SMALLINT     NOT NULL,
SEX        CHAR(1)       ,
BIRTHDATE  DATE          ,
SALARY     DECIMAL(9,2)  ,
BONUS      DECIMAL(9,2)  ,
COMM       DECIMAL(9,2)  ,
PRIMARY KEY (EMPNO))

```

```

ALTER TABLE EMPLOYEE
ADD FOREIGN KEY RED (WORKDEPT)
REFERENCES DEPARTMENT
ON DELETE SET NULL

```

```

ALTER TABLE EMPLOYEE
ADD CONSTRAINT NUMBER
CHECK (PHONENO >= '0000' AND PHONENO <= '9999')

```

มีการสร้างดรรชนีต่อไปนี้:

```

CREATE UNIQUE INDEX XEMP1
ON EMPLOYEE (EMPNO)

```

```

CREATE INDEX XEMP2
ON EMPLOYEE (WORKDEPT)

```

มีการสร้าง alias ต่อไปนี้ให้กับตาราง:

```

CREATE ALIAS EMP FOR EMPLOYEE

```

ตารางต่อไปนี้แสดงถึงรายละเอียดของคอลัมน์.

ชื่อคอลัมน์	รายละเอียด
EMPNO	หมายเลขพนักงาน
FIRSTNAME	ชื่อแรกของพนักงาน
MIDINIT	ตัวขึ้นต้นชื่อกลางของพนักงาน
LASTNAME	นามสกุลของพนักงาน
WORKDEPT	ID ของแผนกที่พนักงานทำงานอยู่
PHONENO	หมายเลขโทรศัพท์ของพนักงาน
HIREDATE	วันที่ว่าจ้าง
JOB	ตำแหน่งงานของพนักงาน
EDLEVEL	จำนวนปีการศึกษาตามที่บังคับ
SEX	เพศของพนักงาน (ช หรือ หญิง)
BIRTHDATE	วันเกิด

ชื่อคอลัมน์

รายละเอียด

SALARY

เงินเดือนต่อปีเป็นดอลลาร์

BONUS

โบนัสต่อปีเป็นดอลลาร์

COMM

คอมมิชชันต่อปีเป็นดอลลาร์

EMPLOYEE:

รายการของข้อมูลในตาราง EMPLOYEE.

EMP NO	FIRST NAME	MID INIT	LASTNAME	WORK DEPT	PHONE NO	HIRE DATE	JOB	ED LEVEL	SEX	BIRTH DATE	SAL-ARY	BONUS	COMM
000010	CHRISTINE	I	HAAS	A00	3978	1965-01-01	PRES	18	F	1933-08-24	52750	1000	4220
000020	MICHAEL	L	THOMPSON	B01	3476	1973-10-10	MANAGER	18	M	1948-02-02	41250	800	3300
000030	SALLY	A	KWAN	C01	4738	1975-04-05	MANAGER	20	F	1941-05-11	38250	800	3060
000050	JOHN	B	GEYER	E01	6789	1949-08-17	MANAGER	16	M	1925-09-15	40175	800	3214
000060	IRVING	F	STERN	D11	6423	1973-09-14	MANAGER	16	M	1945-07-07	32250	500	2580
000070	EVA	D	PULASKI	D21	7831	1980-09-30	MANAGER	16	F	1953-05-26	36170	700	2893
000090	EILEEN	W	HENDERSON	E11	5498	1970-08-15	MANAGER	16	F	1941-05-15	29750	600	2380
000100	THEODORE	Q	SPENSER	E21	0972	1980-06-19	MANAGER	14	M	1956-12-18	26150	500	2092
000110	VINCENZO	G	LUCCHESSI	A00	3490	1958-05-16	SALESREP	19	M	1929-11-05	46500	900	3720
000120	SEAN	O	CONNELL	A00	2167	1963-12-05	CLERK	14	M	1942-10-18	29250	600	2340
000130	DOLORES	M	QUINTANA	C01	4578	1971-07-28	ANALYST	16	F	1925-09-15	23800	500	1904
000140	HEATHER	A	NICHOLLS	C01	1793	1976-12-15	ANALYST	18	F	1946-01-19	28420	600	2274
000150	BRUCE		ADAMSON	D11	4510	1972-02-12	DESIGNER	16	M	1947-05-17	25280	500	2022
000160	ELIZABETH	R	PIANKA	D11	3782	1977-10-11	DESIGNER	17	F	1955-04-12	22250	400	1780
000170	MASATOSHI	J	YOSHIMURA	D11	2890	1978-09-15	DESIGNER	16	M	1951-01-05	24680	500	1974
000180	MARILYN	S	SCOUTTEN	D11	1682	1973-07-07	DESIGNER	17	F	1949-02-21	21340	500	1707
000190	JAMES	H	WALKER	D11	2986	1974-07-26	DESIGNER	16	M	1952-06-25	20450	400	1636
000200	DAVID		BROWN	D11	4501	1966-03-03	DESIGNER	16	M	1941-05-29	27740	600	2217
000210	WILLIAM	T	JONES	D11	0942	1979-04-11	DESIGNER	17	M	1953-02-23	18270	400	1462
000220	JENNIFER	K	LUTZ	D11	0672	1968-08-29	DESIGNER	18	F	1948-03-19	29840	600	2387
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21	CLERK	14	M	1935-05-30	22180	400	1774
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05	CLERK	17	M	1954-03-31	28760	600	2301
000250	DANIEL	S	SMITH	D21	0961	1969-10-30	CLERK	15	M	1939-11-12	19180	400	1534
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11	CLERK	16	F	1936-10-05	17250	300	1380
000270	MARIA	L	PEREZ	D21	9001	1980-09-30	CLERK	15	F	1953-05-26	27380	500	2190
000280	ETHEL	R	SCHNEIDER	E11	8997	1967-03-24	OPERATOR	17	F	1936-03-28	26250	500	2100
000290	JOHN	R	PARKER	E11	4502	1980-05-30	OPERATOR	12	M	1946-07-09	15340	300	1227
000300	PHILIP	X	SMITH	E11	2095	1972-06-19	OPERATOR	14	M	1936-10-27	17750	400	1420
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12	OPERATOR	12	F	1931-04-21	15900	300	1272
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07	FILEREP	16	M	1932-08-11	19950	400	1596
000330	WING		LEE	E21	2103	1976-02-23	FILEREP	14	M	1941-07-18	25370	500	2030
000340	JASON	R	GOUNOT	E21	5698	1947-05-05	FILEREP	16	M	1926-05-17	23840	500	1907
200010	DIAN	J	HEMMINGER	A00	3978	1965-01-01	SALESREP	18	F	1933-08-14	46500	1000	4220
200120	GREG		ORLANDO	A00	2167	1972-05-05	CLERK	14	M	1942-10-18	29250	600	2340
200140	KIM	N	NATZ	C01	1793	1976-12-15	ANALYST	18	F	1946-01-19	28420	600	2274
200170	KIYOSHI		YAMAMOTO	D11	2890	1978-09-15	DESIGNER	16	M	1951-01-05	24680	500	1974
200220	REBA	K	JOHN	D11	0672	1968-08-29	DESIGNER	18	F	1948-03-19	29840	600	2387
200240	ROBERT	M	MONTEVERDE	D21	3780	1979-12-05	CLERK	17	M	1954-03-31	28760	600	2301
200280	EILEEN	R	SCHWARTZ	E11	8997	1967-03-24	OPERATOR	17	F	1936-03-28	26250	500	2100
200310	MICHELLE	F	SPRINGER	E11	3332	1964-09-12	OPERATOR	12	F	1931-04-21	15900	300	1272
200330	HELENA		WONG	E21	2103	1976-02-23	FIELDREP	14	F	1941-07-18	25370	500	2030
200340	ROY	R	ALONZO	E21	5698	1947-05-05	FIELDREP	16	M	1926-05-17	23840	500	1907

ตารางภาพถ่ายพนักงาน (EMP_PHOTO)

ตารางภาพถ่ายพนักงานประกอบด้วยภาพถ่ายสำหรับพนักงาน ซึ่งเก็บไว้ตามหมายเลขพนักงาน.

สร้างตารางภาพถ่ายพนักงานด้วยคำสั่ง CREATE TABLE และ ALTER TABLE ต่อไปนี้:

```
CREATE TABLE EMP_PHOTO
  (EMPNO      CHAR(6)          NOT NULL,
   PHOTO_FORMAT VARCHAR(10) NOT NULL,
   PICTURE    BLOB(100K),
   EMP_ROWID  CHAR(40) NOT NULL DEFAULT '',
   PRIMARY KEY (EMPNO,PHOTO_FORMAT))
```

```
ALTER TABLE EMP_PHOTO
  ADD COLUMN DL_PICTURE DATALINK(1000)
  LINKTYPE URL NO LINK CONTROL
```

```
ALTER TABLE EMP_PHOTO
  ADD FOREIGN KEY (EMPNO)
  REFERENCES EMPLOYEE
  ON DELETE RESTRICT
```

ดรรชนีต่อไปนี้จะถูกสร้าง:

```
CREATE UNIQUE INDEX XEMP_PHOTO
  ON EMP_PHOTO (EMPNO,PHOTO_FORMAT)
```

ตารางต่อไปนี้จะแสดงถึงรายละเอียดของคอลัมน์.

ชื่อคอลัมน์	รายละเอียด
EMPNO	หมายเลขพนักงาน
PHOTO_FORMAT	รูปแบบสำหรับภาพที่เก็บไว้ใน PICTURE
PICTURE	ภาพถ่าย
EMP_ROWID	รหัสแถว, ไม่ได้ถูกใช้ในตอนนี้

EMP_PHOTO:

รายการของข้อมูลในตาราง EMP_PHOTO.

EMPNO	PHOTO_FORMAT	PICTURE	EMP_ROWID
000130	bitmap	?	
000130	gif	?	
000140	bitmap	?	
000140	gif	?	
000150	bitmap	?	
000150	gif	?	

EMPNO	PHOTO_FORMAT	PICTURE	EMP_ROWID
000190	bitmap	?	
000190	gif	?	

ตารางประวัติพนักงาน (EMP_RESUME)

ตารางภาพถ่ายพนักงานประกอบด้วยประวัติพนักงานซึ่งเก็บไว้ตามหมายเลขพนักงาน.

ตารางประวัติพนักงานถูกสร้างด้วยคำสั่ง CREATE TABLE และ ALTER TABLE ต่อไปนี้:

```
CREATE TABLE EMP_RESUME
  (EMPNO CHAR(6) NOT NULL,
   RESUME_FORMAT VARCHAR(10) NOT NULL,
   RESUME CLOB(5K),
   EMP_ROWID CHAR(40) NOT NULL DEFAULT '',
   PRIMARY KEY (EMPNO, RESUME_FORMAT))
```

```
ALTER TABLE EMP_RESUME
  ADD COLUMN DL_RESUME DATA LINK(1000)
  LINKTYPE URL NO LINK CONTROL
```

```
ALTER TABLE EMP_RESUME
  ADD FOREIGN KEY (EMPNO)
  REFERENCES EMPLOYEE
  ON DELETE RESTRICT
```

ดรรชนีต่อไปนี้จะถูกสร้าง:

```
CREATE UNIQUE INDEX XEMP_RESUME
  ON EMP_RESUME (EMPNO, RESUME_FORMAT)
```

ตารางต่อไปนี้แสดงถึงรายละเอียดของคอลัมน์.

ชื่อคอลัมน์	รายละเอียด
EMPNO	หมายเลขพนักงาน
RESUME_FORMAT	รูปแบบข้อความที่เก็บไว้ใน RESUME
RESUME	ประวัติ
EMP_ROWID	รหัสแถว, ไม่ได้ถูกใช้ในตอนนี้

EMP_RESUME:

รายการของข้อมูลในตาราง EMP_RESUME.

EMPNO	RESUME_FORMAT	RESUME	EMP_ROWID
000130	ascii	?	
000130	html	?	

EMPNO	RESUME_FORMAT	RESUME	EMP_ROWID
000140	ascii	?	
000140	html	?	
000150	ascii	?	
000150	html	?	
000190	ascii	?	
000190	html	?	

ตารางพนักงานต่อกิจกรรมโครงการ (EMPPROJECT)

ตารางพนักงานต่อกิจกรรมโครงการจะระบุถึงพนักงานที่ทำงานในแต่ละกิจกรรมสำหรับแต่ละโครงการ. ระดับความเกี่ยวข้องของพนักงาน (ประจำหรือพาร์ทไทม์) และกำหนดการสำหรับกิจกรรมยังอยู่ในตารางนี้เช่นกัน.

ตารางพนักงานต่อกิจกรรมโครงการสามารถสร้างขึ้นด้วยคำสั่ง CREATE TABLE และ ALTER TABLE ต่อไปนี้:

```
CREATE TABLE EMPPROJECT
  (EMPNO      CHAR(6)          NOT NULL,
   PROJNO     CHAR(6)          NOT NULL,
   ACTNO      SMALLINT        NOT NULL,
   EMPTIME    DECIMAL(5,2)    ,
   EMSTDATE   DATE            ,
   EMENDATE   DATE            )

ALTER TABLE EMPPROJECT
  ADD FOREIGN KEY REPAPA (PROJNO, ACTNO, EMSTDATE)
  REFERENCES PROJECT
  ON DELETE RESTRICT
```

alias ต่อไปนี้จะถูกสร้างขึ้นให้กับตาราง:

```
CREATE ALIAS EMPACT FOR EMPPROJECT

CREATE ALIAS EMP_ACT FOR EMPPROJECT
```

ตารางต่อไปนี้แสดงถึงรายละเอียดของคอลัมน์.

ตารางที่ 53. คอลัมน์ของพนักงานต่อกิจกรรมโครงการ

ชื่อคอลัมน์	รายละเอียด
EMPNO	รหัสพนักงาน
PROJNO	PROJNO ของโครงการที่มอบหมายให้พนักงาน
ACTNO	ID ของกิจกรรมภายในโครงการที่มอบหมายให้พนักงาน
EMPTIME	สัดส่วนเวลาเต็มของพนักงาน (ระหว่าง 0.00 ถึง 1.00) ที่ต้องใช้ในโครงการจาก EMSTDATE ถึง EMENDATE
EMSTDATE	วันเริ่มต้นกิจกรรม

ตารางที่ 53. คอลัมน์ของพนักงานต่อกิจกรรมโครงการ (ต่อ)

ชื่อคอลัมน์

รายละเอียด

EMENDATE

วันเสร็จสิ้นกิจกรรม

EMPPROJECT:

รายการของข้อมูลที่สมบูรณ์ในตาราง EMPPROJECT.

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000010	AD3100	10	.50	1982-01-01	1982-07-01
000070	AD3110	10	1.00	1982-01-01	1983-02-01
000230	AD3111	60	1.00	1982-01-01	1982-03-15
000230	AD3111	60	.50	1982-03-15	1982-04-15
000230	AD3111	70	.50	1982-03-15	1982-10-15
000230	AD3111	80	.50	1982-04-15	1982-10-15
000230	AD3111	180	.50	1982-10-15	1983-01-01
000240	AD3111	70	1.00	1982-02-15	1982-09-15
000240	AD3111	80	1.00	1982-09-15	1983-01-01
000250	AD3112	60	1.00	1982-01-01	1982-02-01
000250	AD3112	60	.50	1982-02-01	1982-03-15
000250	AD3112	60	1.00	1983-01-01	1983-02-01
000250	AD3112	70	.50	1982-02-01	1982-03-15
000250	AD3112	70	1.00	1982-03-15	1982-08-15
000250	AD3112	70	.25	1982-08-15	1982-10-15
000250	AD3112	80	.25	1982-08-15	1982-10-15
000250	AD3112	80	.50	1982-10-15	1982-12-01
000250	AD3112	180	.50	1982-08-15	1983-01-01
000260	AD3113	70	.50	1982-06-15	1982-07-01
000260	AD3113	70	1.00	1982-07-01	1983-02-01
000260	AD3113	80	1.00	1982-01-01	1982-03-01
000260	AD3113	80	.50	1982-03-01	1982-04-15
000260	AD3113	180	.50	1982-03-01	1982-04-15

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000260	AD3113	180	1.00	1982-04-15	1982-06-01
000260	AD3113	180	1.00	1982-06-01	1982-07-01
000270	AD3113	60	.50	1982-03-01	1982-04-01
000270	AD3113	60	1.00	1982-04-01	1982-09-01
000270	AD3113	60	.25	1982-09-01	1982-10-15
000270	AD3113	70	.75	1982-09-01	1982-10-15
000270	AD3113	70	1.00	1982-10-15	1983-02-01
000270	AD3113	80	1.00	1982-01-01	1982-03-01
000270	AD3113	80	.50	1982-03-01	1982-04-01
000030	IF1000	10	.50	1982-06-01	1983-01-01
000130	IF1000	90	1.00	1982-10-01	1983-01-01
000130	IF1000	100	.50	1982-10-01	1983-01-01
000140	IF1000	90	.50	1982-10-01	1983-01-01
000030	IF2000	10	.50	1982-01-01	1983-01-01
000140	IF2000	100	1.00	1982-01-01	1982-03-01
000140	IF2000	100	.50	1982-03-01	1982-07-01
000140	IF2000	110	.50	1982-03-01	1982-07-01
000140	IF2000	110	.50	1982-10-01	1983-01-01
000010	MA2100	10	.50	1982-01-01	1982-11-01
000110	MA2100	20	1.00	1982-01-01	1983-03-01
000010	MA2110	10	1.00	1982-01-01	1983-02-01
000200	MA2111	50	1.00	1982-01-01	1982-06-15
000200	MA2111	60	1.00	1982-06-15	1983-02-01
000220	MA2111	40	1.00	1982-01-01	1983-02-01
000150	MA2112	60	1.00	1982-01-01	1982-07-15
000150	MA2112	180	1.00	1982-07-15	1983-02-01
000170	MA2112	60	1.00	1982-01-01	1983-06-01
000170	MA2112	70	1.00	1982-06-01	1983-02-01
000190	MA2112	70	1.00	1982-01-01	1982-10-01

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000190	MA2112	80	1.00	1982-10-01	1983-10-01
000160	MA2113	60	1.00	1982-07-15	1983-02-01
000170	MA2113	80	1.00	1982-01-01	1983-02-01
000180	MA2113	70	1.00	1982-04-01	1982-06-15
000210	MA2113	80	.50	1982-10-01	1983-02-01
000210	MA2113	180	.50	1982-10-01	1983-02-01
000050	OP1000	10	.25	1982-01-01	1983-02-01
000090	OP1010	10	1.00	1982-01-01	1983-02-01
000280	OP1010	130	1.00	1982-01-01	1983-02-01
000290	OP1010	130	1.00	1982-01-01	1983-02-01
000300	OP1010	130	1.00	1982-01-01	1983-02-01
000310	OP1010	130	1.00	1982-01-01	1983-02-01
000050	OP2010	10	.75	1982-01-01	1983-02-01
000100	OP2010	10	1.00	1982-01-01	1983-02-01
000320	OP2011	140	.75	1982-01-01	1983-02-01
000320	OP2011	150	.25	1982-01-01	1983-02-01
000330	OP2012	140	.25	1982-01-01	1983-02-01
000330	OP2012	160	.75	1982-01-01	1983-02-01
000340	OP2013	140	.50	1982-01-01	1983-02-01
000340	OP2013	170	.50	1982-01-01	1983-02-01
000020	PL2100	30	1.00	1982-01-01	1982-09-15

ตารางโครงการ (PROJECT)

ตารางโครงการจะอธิบายถึงโครงการแต่ละอย่างที่ธุรกิจดำเนินการอยู่ในปัจจุบัน. ข้อมูลที่อยู่ในแต่ละแถวประกอบด้วยหมายเลขโครงการ, ชื่อ, บุคคลที่รับผิดชอบ, และวันที่ตามกำหนดการ.

ตารางโครงการถูกสร้างด้วยคำสั่ง CREATE TABLE และ ALTER TABLE ต่อไปนี้:

```
CREATE TABLE PROJECT
  (PROJNO CHAR(6) NOT NULL,
   PROJNAME VARCHAR(24) NOT NULL DEFAULT,
   DEPTNO CHAR(3) NOT NULL,
   RESPEMP CHAR(6) NOT NULL,
   PRSTAFF DECIMAL(5,2) ,
   PRSTDATE DATE ,
```

```

PRENDATE DATE          ,
MAJPROJ  CHAR(6)       ,
PRIMARY KEY (PROJNO))

```

```

ALTER TABLE PROJECT
  ADD FOREIGN KEY (DEPTNO)
  REFERENCES DEPARTMENT
  ON DELETE RESTRICT

```

```

ALTER TABLE PROJECT
  ADD FOREIGN KEY (RESPEMP)
  REFERENCES EMPLOYEE
  ON DELETE RESTRICT

```

```

ALTER TABLE PROJECT
  ADD FOREIGN KEY RPP (MAJPROJ)
  REFERENCES PROJECT
  ON DELETE CASCADE

```

มีการสร้างดรรชนีต่อไปนี้:

```

CREATE UNIQUE INDEX XPROJ1
  ON PROJECT (PROJNO)

```

```

CREATE INDEX XPROJ2
  ON PROJECT (RESPEMP)

```

มีการสร้าง alias ต่อไปนี้ให้กับตาราง:

```

CREATE ALIAS PROJ FOR PROJECT

```

ตารางต่อไปนี้แสดงถึงรายละเอียดของคอลัมน์:

ชื่อคอลัมน์	รายละเอียด
PROJNO	หมายเลขโครงการ
PROJNAME	ชื่อโครงการ
DEPTNO	หมายเลขแผนกของแผนกที่รับผิดชอบโครงการ
RESPEMP	หมายเลขพนักงานของพนักงานที่รับผิดชอบโครงการ
PRSTAFF	จำนวนพนักงานโดยประมาณ
PRSTDATE	วันที่เริ่มต้นโครงการโดยประมาณ
PRENDATE	วันที่สิ้นสุดโครงการโดยประมาณ
MAJPROJ	หมายเลขควบคุมโครงการสำหรับโครงการย่อย

PROJECT:

รายการของข้อมูลที่สมบูรณ์ในตาราง PROJECT.

PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
AD3100	ADMIN SERVICES	D01	000010	6.5	1982-01-01	1983-02-01	?
AD3110	GENERAL ADMIN SYSTEMS	D21	000070	6	1982-01-01	1983-02-01	AD3100
AD3111	PAYROLL PROGRAMMING	D21	000230	2	1982-01-01	1983-02-01	AD3110
AD3112	PERSONNEL PROGRAMMING	D21	000250	1	1982-01-01	1983-02-01	AD3110
AD3113	ACCOUNT PROGRAMMING	D21	000270	2	1982-01-01	1983-02-01	AD3110
IF1000	QUERY SERVICES	C01	000030	2	1982-01-01	1983-02-01	?
IF2000	USER EDUCATION	C01	000030	1	1982-01-01	1983-02-01	?
MA2100	WELDLINE AUTOMATION	D01	000010	12	1982-01-01	1983-02-01	?
MA2110	WL PROGRAMMING	D11	000060	9	1982-01-01	1983-02-01	MA2100
MA2111	WL PROGRAM DESIGN	D11	000220	2	1982-01-01	1982-12-01	MA2110
MA2112	WL ROBOT DESIGN	D11	000150	3	1982-01-01	1982-12-01	MA2110
MA2113	WL PRODCONT PROGS	D11	000160	3	1982-02-15	1982-12-01	MA2110
OP1000	OPERATION SUPPORT	E01	000050	6	1982-01-01	1983-02-01	?
OP1010	OPERATION	E11	000090	5	1982-01-01	1983-02-01	OP1000
OP2000	GEN SYSTEMS SERVICES	E01	000050	5	1982-01-01	1983-02-01	?
OP2010	SYSTEMS SUPPORT	E21	000100	4	1982-01-01	1983-02-01	OP2000
OP2011	SCPSYSTEMS SUPPORT	E21	000320	1	1982-01-01	1983-02-01	OP2010
OP2012	APPLICATIONS SUPPORT	E21	000330	1	1982-01-01	1983-02-01	OP2010
OP2013	DB/DC SUPPORT	E21	000340	1	1982-01-01	1983-02-01	OP2010
PL2100	WELDLINE PLANNING	B01	000020	1	1982-01-01	1982-09-15	MA2100

ตารางกิจกรรมโครงการ (PROJECT)

ตารางกิจกรรมโครงการจะอธิบายถึงโครงการแต่ละอย่างที่ถูกดำเนินการอยู่ในปัจจุบัน. ข้อมูลในแต่ละแถวประกอบด้วยหมายเลขโครงการ, หมายเลขกิจกรรม, และวันที่ตามกำหนดการ.

สร้างตารางกิจกรรมโครงการ ด้วยคำสั่ง CREATE TABLE และ ALTER TABLE ต่อไปนี้:

```
CREATE TABLE PROJECT
    (PROJNO CHAR(6) NOT NULL,
     ACTNO SMALLINT NOT NULL,
     ACSTAFF DECIMAL(5,2),
     ACSTDATE DATE NOT NULL,
     ACENDATE DATE ,
     PRIMARY KEY (PROJNO, ACTNO, ACSTDATE))
```

```
ALTER TABLE PROJECT
    ADD FOREIGN KEY RPAP (PROJNO)
    REFERENCES PROJECT
    ON DELETE RESTRICT
```

มีการเพิ่มคีย์ foreign ต่อไปนี้ในภายหลัง:

```
ALTER TABLE PROJECT
    ADD FOREIGN KEY RPAA (ACTNO)
    REFERENCES ACT
    ON DELETE RESTRICT
```

ดรรชนีต่อไปนี้จะถูกสร้าง:

```
CREATE UNIQUE INDEX XPROJAC1
    ON PROJECT (PROJNO, ACTNO, ACSTDATE)
```

ตารางต่อไปนี้แสดงถึงรายละเอียดของคอลัมน์:

ชื่อคอลัมน์	รายละเอียด
PROJNO	หมายเลขโครงการ
ACTNO	หมายเลขกิจกรรม
ACSTAFF	จำนวนพนักงานโดยประมาณ
ACSTDATE	วันที่เริ่มกิจกรรม
ACENDATE	วันที่สิ้นสุดกิจกรรม

PROJECT:

รายการของข้อมูลที่สมบูรณ์ในตาราง PROJECT.

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
AD3100	10	?	1982-01-01	?
AD3110	10	?	1982-01-01	?

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
AD3111	60	?	1982-01-01	?
AD3111	60	?	1982-03-15	?
AD3111	70	?	1982-03-15	?
AD3111	80	?	1982-04-15	?
AD3111	180	?	1982-10-15	?
AD3111	70	?	1982-02-15	?
AD3111	80	?	1982-09-15	?
AD3112	60	?	1982-01-01	?
AD3112	60	?	1982-02-01	?
AD3112	60	?	1983-01-01	?
AD3112	70	?	1982-02-01	?
AD3112	70	?	1982-03-15	?
AD3112	70	?	1982-08-15	?
AD3112	80	?	1982-08-15	?
AD3112	80	?	1982-10-15	?
AD3112	180	?	1982-08-15	?
AD3113	70	?	1982-06-15	?
AD3113	70	?	1982-07-01	?
AD3113	80	?	1982-01-01	?
AD3113	80	?	1982-03-01	?
AD3113	180	?	1982-03-01	?
AD3113	180	?	1982-04-15	?
AD3113	180	?	1982-06-01	?
AD3113	60	?	1982-03-01	?
AD3113	60	?	1982-04-01	?
AD3113	60	?	1982-09-01	?
AD3113	70	?	1982-09-01	?
AD3113	70	?	1982-10-15	?
IF1000	10	?	1982-06-01	?

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
IF1000	90	?	1982-10-01	?
IF1000	100	?	1982-10-01	?
IF2000	10	?	1982-01-01	?
IF2000	100	?	1982-01-01	?
IF2000	100	?	1982-03-01	?
IF2000	110	?	1982-03-01	?
IF2000	110	?	1982-10-01	?
MA2100	10	?	1982-01-01	?
MA2100	20	?	1982-01-01	?
MA2110	10	?	1982-01-01	?
MA2111	50	?	1982-01-01	?
MA2111	60	?	1982-06-15	?
MA2111	40	?	1982-01-01	?
MA2112	60	?	1982-01-01	?
MA2112	180	?	1982-07-15	?
MA2112	70	?	1982-06-01	?
MA2112	70	?	1982-01-01	?
MA2112	80	?	1982-10-01	?
MA2113	60	?	1982-07-15	?
MA2113	80	?	1982-01-01	?
MA2113	70	?	1982-04-01	?
MA2113	80	?	1982-10-01	?
MA2113	180	?	1982-10-01	?
OP1000	10	?	1982-01-01	?
OP1010	10	?	1982-01-01	?
OP1010	130	?	1982-01-01	?
OP2010	10	?	1982-01-01	?
OP2011	140	?	1982-01-01	?
OP2011	150	?	1982-01-01	?

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
OP2012	140	?	1982-01-01	?
OP2012	160	?	1982-01-01	?
OP2013	140	?	1982-01-01	?
OP2013	170	?	1982-01-01	?
PL2100	30	?	1982-01-01	?

ตารางกิจกรรม (ACT)

ตารางกิจกรรมจะอธิบายรายละเอียดของแต่ละกิจกรรม.

สามารถสร้างตารางกิจกรรมด้วยคำสั่ง CREATE TABLE ต่อไปนี้:

```
CREATE TABLE ACT
    (ACTNO SMALLINT NOT NULL,
    ACTKWD CHAR(6) NOT NULL,
    ACTDESC VARCHAR(20) NOT NULL,
    PRIMARY KEY (ACTNO))
```

มีการสร้างดรรชนีต่อไปนี้:

```
CREATE UNIQUE INDEX XACT1
    ON ACT (ACTNO)
```

```
CREATE UNIQUE INDEX XACT2
    ON ACT (ACTKWD)
```

ตารางต่อไปนี้แสดงถึงรายละเอียดของคอลัมน์.

ชื่อคอลัมน์	รายละเอียด
ACTNO	หมายเลขกิจกรรม
ACTKWD	คีย์เวิร์ดสำหรับกิจกรรม
ACTDESC	รายละเอียดของกิจกรรม

ACT:

รายการของข้อมูลที่สมบูรณ์ในตาราง ACT.

ACTNO	ACTKWD	ACTDESC
10	MANAGE	MANAGE/ADVISE
20	ECOST	ESTIMATE COST
30	DEFINE	DEFINE SPECS
40	LEADPR	LEAD PROGRAM/DESIGN

ACTNO	ACTKWD	ACTDESC
50	SPECS	WRITE SPECS
60	LOGIC	DESCRIBE LOGIC
70	CODE	CODE PROGRAMS
80	TEST	TEST PROGRAMS
90	ADMQS	ADM QUERY SYSTEM
100	TEACH	TEACH CLASSES
110	COURSE	DEVELOP COURSES
120	STAFF	PERS AND STAFFING
130	OPERAT	OPER COMPUTER SYS
140	MAINT	MAINT SOFTWARE SYS
150	ADMSYS	ADM OPERATING SYS
160	ADMDB	ADM DATA BASES
170	ADMDC	ADM DATA COMM
180	DOC	DOCUMENT

ตารางการกำหนดเวลาเรียน (CL_SCHED)

ตารางการกำหนดเวลาเรียนจะอธิบาย: แต่ละชั้นเรียน, เวลาเริ่มต้นของชั้นเรียน, เวลาสิ้นสุดของชั้นเรียน, และโค้ดของชั้นเรียน.

สามารถสร้างตารางกำหนดการเรียน ด้วยคำสั่ง CREATE TABLE ต่อไปนี้:

```
CREATE TABLE CL_SCHED
  (CLASS_CODE      CHAR(7),
   "DAY"           SMALLINT,
   STARTING        TIME,
   ENDING          TIME)
```

ตารางต่อไปนี้แสดงถึงรายละเอียดของคอลัมน์.

ชื่อคอลัมน์	รายละเอียด
CLASS_CODE	โค้ดของชั้นเรียน (ห้อง:อาจารย์)
DAY	หมายเลขวันของกำหนดการ 4 วัน
STARTING	เวลาเริ่มต้นของชั้นเรียน
ENDING	เวลาสิ้นสุดชั้นเรียน

CL_SCHED:

รายการของข้อมูลในตาราง CL_SCHED.

CLASS_CODE	DAY	STARTING	ENDING
042:BF	4	12:10:00	14:00:00
553:MJA	1	10:30:00	11:00:00
543:CWM	3	09:10:00	10:30:00
778:RES	2	12:10:00	14:00:00
044:HD	3	17:12:30	18:00:00

ตาราง In tray (IN_TRAY)

ตาราง in tray จะให้รายละเอียดของตะกร้ารับข้อมูลแบบอิเล็กทรอนิกส์ ซึ่งประกอบด้วย timestamp แสดงเวลาที่รับข้อความ, user ID ของบุคคลที่ส่งข้อความ, และตัวข้อความ.

สามารถสร้างตาราง in tray ด้วยคำสั่ง CREATE TABLE ต่อไปนี้:

```
CREATE TABLE IN_TRAY
  (RECEIVED          TIMESTAMP,
   SOURCE            CHAR(8),
   SUBJECT           CHAR(64),
   NOTE_TEXT        VARCHAR(3000))
```

ตารางต่อไปนี้แสดงถึงรายละเอียดของคอลัมน์.

ชื่อคอลัมน์	รายละเอียด
RECEIVED	วันและเวลาที่ได้รับ
SOURCE	user ID ของบุคคลที่ส่งหมายเหตุ
SUBJECT	รายละเอียดโดยย่อของหมายเหตุ
NOTE_TEXT	หมายเหตุ

IN_TRAY:

รายการของข้อมูลที่สมบูรณ์ในตาราง IN_TRAY.

RECEIVED	SOURCE	SUBJECT	NOTE_TEXT
1988-12-25-17.12. 30.000000	BADAMSON	FWD: Fanstastic year! 4th Quarter Bonus.	To: JWALKER Cc: QUINTANA, NICHOLLS Jim, Looks like our hard work has paid off . I have some good beer in the fridge if you want to come over to celebrate a bit. Delores and Heather, are you interested as well ? Bruce <Forwarding from ISTERN> Subject: FWD: Fantastic year! 4th Quarter Bonus. To: Dept_D11 Congratulations on a job well done. Enjoy this year's bonus. Irv <Forwarding from CHAAS> Subject: Fantastic year! 4th Quarter Bonus. To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas
1988-12-23-08.53. 58.000000	ISTERN	FWD: Fanstastic year! 4th Quarter Bonus.	To: Dept_D11 Congratulations on a job well done. Enjoy this year's bonus. Irv <Forwarding from CHAAS> Subject: Fantastic year! 4th Quarter Bonus. To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas
1988-12-22-14.07. 21.136421	CHAAS	Fantastic year! 4th Quarter Bonus.	To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas

ตารางโครงสร้าง (ORG)

ตารางโครงสร้างจะอธิบายถึงโครงสร้างของบริษัท.

สร้างตารางด้วยคำสั่ง CREATE TABLE ต่อไปนี้:

```
CREATE TABLE ORG
  (DEPTNUMB SMALLINT NOT NULL,
   DEPTNAME VARCHAR(14),
   MANAGER SMALLINT,
   DIVISION VARCHAR(10),
   LOCATION VARCHAR(13))
```


ตารางต่อไปนี้จะแสดงถึงรายละเอียดของคอลัมน์.

ชื่อคอลัมน์	รายละเอียด
DEPTNUMB	หมายเลขแผนก
DEPTNAME	ชื่อแผนก
MANAGER	หมายเลขผู้จัดการของแผนก
DIVISION	หน่วยงานของแผนก
LOCATION	ที่ตั้งของแผนก

ORG:

รายการของข้อมูลที่สมบูรณ์ในตาราง ORG.

DEPTNUMB	DEPTNAME	MANAGER	DIVISION	LOCATION
10	Head Office	160	Corporate	New York
15	New England	50	Eastern	Boston
20	Mid Atlantic	10	Eastern	Washington
38	South Atlantic	30	Eastern	Atlanta
42	Great Lakes	100	Midwest	Chicago
51	Plains	140	Midwest	Dallas
66	Pacific	270	Western	San Francisco
84	Mountain	290	Western	Denver

ตารางพนักงาน (STAFF)

ตารางพนักงานจะให้รายละเอียดเกี่ยวกับข้อมูลแบ็กกราวนด์ของพนักงาน.

สร้างตารางพนักงานด้วยคำสั่ง CREATE TABLE ต่อไปนี้:

```
CREATE TABLE STAFF
  (ID SMALLINT NOT NULL,
   NAME VARCHAR(9),
   DEPT SMALLINT,
   JOB CHAR(5),
   YEARS SMALLINT,
   SALARY DECIMAL(7,2),
   COMM DECIMAL(7,2))
```

ตารางต่อไปนี้แสดงถึงรายละเอียดของคอลัมน์.

ชื่อคอลัมน์	รายละเอียด
ID	หมายเลขพนักงาน
NAME	ชื่อพนักงาน
DEPT	หมายเลขแผนก
JOB	ตำแหน่งงาน
YEARS	จำนวนปีที่ทำงานกับบริษัท
SALARY	เงินเดือนต่อปีของพนักงาน
COMM	ค่านายหน้าของพนักงาน

STAFF:

รายการของข้อมูลที่สมบูรณ์ในตาราง STAFF.

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	?
20	Pernal	20	Sales	8	18171.25	612.45
30	Marenghi	38	Mgr	5	17506.75	?
40	O'Brien	38	Sales	6	18006.00	846.55
50	Hanes	15	Mgr	10	20659.80	?
60	Quigley	38	Sales	7	16508.30	650.25
70	Rothman	15	Sales	7	16502.83	1152.00
80	James	20	Clerk	?	13504.60	128.20
90	Koonitz	42	Sales	6	18001.75	1386.70
100	Plotz	42	Mgr	7	18352.80	?
110	Ngan	15	Clerk	5	12508.20	206.60
120	Naughton	38	Clerk	?	12954.75	180.00
130	Yamaguchi	42	Clerk	6	10505.90	75.60
140	Fraye	51	Mgr	6	21150.00	?
150	Williams	51	Sales	6	19456.50	637.65
160	Molinare	10	Mgr	7	22959.20	?
170	Kermisch	15	Clerk	4	12258.50	110.10

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
180	Abrahams	38	Clerk	3	12009.75	236.50
190	Sneider	20	Clerk	8	14252.75	126.50
200	Scoutten	42	Clerk	?	11508.60	84.20
210	Lu	10	Mgr	10	20010.00	?
220	Smith	51	Sales	7	17654.50	992.80
230	Lundquist	51	Clerk	3	13369.80	189.65
240	Daniels	10	Mgr	5	19260.25	?
250	Wheeler	51	Clerk	6	14460.00	513.30
260	Jones	10	Mgr	12	21234.00	?
270	Lea	66	Mgr	9	18555.50	?
280	Wilson	66	Sales	9	18674.50	811.50
290	Quill	84	Mgr	10	19818.00	?
300	Davis	84	Sales	5	15454.50	806.10
310	Graham	66	Sales	13	21000.00	200.30
320	Gonzales	66	Sales	4	16858.20	844.00
330	Burke	66	Clerk	1	10988.00	55.50
340	Edwards	84	Sales	7	17844.00	1285.00
3650	Gafney	84	Clerk	5	13030.50	188.00

ตารางยอดขาย (SALES)

ตารางยอดขายอธิบายถึงข้อมูลของการขายสำหรับพนักงานขายแต่ละคน.

สร้างตารางยอดขายด้วยคำสั่ง CREATE TABLE ต่อไปนี้:

```
CREATE TABLE SALES
(SALES_DATE DATE,
SALES_PERSON VARCHAR(15),
REGION VARCHAR(15),
SALES INTEGER)
```

ตารางต่อไปนี้แสดงถึงรายละเอียดของคอลัมน์.

ชื่อคอลัมน์	รายละเอียด
SALES_DATE	วันที่ขาย
SALES_PERSON	บุคคลที่ขาย

ชื่อคอลัมน์

รายละเอียด

REGION

ภูมิภาคที่มีการขาย

SALES

จำนวนยอดขาย

SALES:

รายการของข้อมูลที่สมบูรณ์ในตาราง SALES.

SALES_DATE	SALES_PERSON	REGION	SALES
12/31/1995	LUCCHESSI	Ontario-South	1
12/31/1995	LEE	Ontario-South	3
12/31/1995	LEE	Quebec	1
12/31/1995	LEE	Manitoba	2
12/31/1995	GOUNOT	Quebec	1
03/29/1996	LUCCHESSI	Ontario-South	3
03/29/1996	LUCCHESSI	Quebec	1
03/29/1996	LEE	Ontario-South	2
03/29/1996	LEE	Ontario-North	2
03/29/1996	LEE	Quebec	3
03/29/1996	LEE	Manitoba	5
03/29/1996	GOUNOT	Ontario-South	3
03/29/1996	GOUNOT	Quebec	1
03/29/1996	GOUNOT	Manitoba	7
03/30/1996	LUCCHESSI	Ontario-South	1
03/30/1996	LUCCHESSI	Quebec	2
03/30/1996	LUCCHESSI	Manitoba	1
03/30/1996	LEE	Ontario-South	7
03/30/1996	LEE	Ontario-North	3
03/30/1996	LEE	Quebec	7
03/30/1996	LEE	Manitoba	4
03/30/1996	GOUNOT	Ontario-South	2
03/30/1996	GOUNOT	Quebec	18

SALES_DATE	SALES_PERSON	REGION	SALES
03/30/1996	GOUNOT	Manitoba	1
03/31/1996	LUCCHESSI	Manitoba	1
03/31/1996	LEE	Ontario-South	14
03/31/1996	LEE	Ontario-North	3
03/31/1996	LEE	Quebec	7
03/31/1996	LEE	Manitoba	3
03/31/1996	GOUNOT	Ontario-South	2
03/31/1996	GOUNOT	Quebec	1
04/01/1996	LUCCHESSI	Ontario-South	3
04/01/1996	LUCCHESSI	Manitoba	1
04/01/1996	LEE	Ontario-South	8
04/01/1996	LEE	Ontario-North	?
04/01/1996	LEE	Quebec	8
04/01/1996	LEE	Manitoba	9
04/01/1996	GOUNOT	Ontario-South	3
04/01/1996	GOUNOT	Ontario-North	1
04/01/1996	GOUNOT	Quebec	3
04/01/1996	GOUNOT	Manitoba	7

DB2 UDB for iSeries คำอธิบายของคำสั่ง CL

DB2 UDB for iSeries ประกอบด้วยคำสั่ง CL สำหรับ SQL.

- คำสั่ง CRTSQLPKG (Create Structured Query Language Package)
- คำสั่ง DLTSQLPKG (Delete Structured Query Language Package)
- คำสั่ง PRSQLINF (Print Structured Query Language Information)
- คำสั่ง RUNSQLSTM (Run Structure Query Language Statement)
- คำสั่ง STRSQL (Start Structure Query Language)

สิ่งอ้างอิงที่เกี่ยวข้อง

“DB2 UDB for iSeries การสนับสนุนฐานข้อมูลเชิงสัมพันธ์แบบกระจาย” ในหน้า 310

ไลเซนส์โปรแกรม DB2 UDB Query Manager and SQL Development Kit สนับสนุนการเข้าถึงแบบโต้ตอบของฐานข้อมูลแบบกระจายโดยใช้คำสั่ง SQL ต่อไปนี้.

ไลเซนส์โค้ดและข้อความปฏิเสธความรับผิดชอบ

IBM มอบสิทธิการใช้โค้ดโปรแกรมมิ่งตัวอย่างทั้งหมดแบบไม่ผูกขาดให้กับคุณ และคุณสามารถสร้างฟังก์ชันที่คล้ายคลึงกันตามความต้องการเฉพาะจากโค้ดนี้ได้.

ภายใต้ข้อกำหนดการรับประกันตามกฎหมายซึ่งไม่สามารถละเว้นได้, IBM, ผู้พัฒนาโปรแกรม และผู้จัดจำหน่าย จะไม่ให้การรับประกันหรือมีเงื่อนไขใดๆ ทั้งโดยชัดแจ้งหรือโดยนัย, ซึ่งรวมถึงการรับประกันหรือเงื่อนไขโดยนัยเกี่ยวกับการนำไปจัดจำหน่าย, ความเหมาะสมสำหรับวัตถุประสงค์เฉพาะอย่าง, และการไม่ละเมิดสิทธิที่เกี่ยวข้องกับตัวโปรแกรมหรือการสนับสนุนทางเทคนิค (หากมี).

ไม่ว่ากรณีใดๆ ก็ตาม IBM, ผู้พัฒนาโปรแกรม หรือผู้จัดจำหน่ายไม่จำเป็นต้องรับผิดชอบต่อเหตุการณ์ต่างๆ ต่อไปนี้, แม้ว่าจะมีการแจ้งถึงความเป็นไปได้ของเหตุการณ์ดังกล่าวก็ตาม:

1. การสูญหาย, หรือความเสียหายของข้อมูล;
2. ความเสียหายโดยตรง, ความเสียหายพิเศษ, ความเสียหายโดยบังเอิญ, หรือความเสียหายทางอ้อม, หรือความเสียหายทางเศรษฐกิจที่ตามมา; หรือ
3. การสูญเสียของกำไร, ธุรกิจ, รายได้, ชื่อเสียง, หรือจำนวนเงินที่คาดว่าจะประหยัดได้.

อำนาจตามกฎหมายบางอย่างไม่อนุญาตให้ยกเว้น หรือการจำกัดความเสียหายโดยตรง, โดยบังเอิญ, หรือความเสียหายที่ตามมา, ดังนั้นข้อจำกัด หรือข้อยกเว้นข้างต้นไม่สามารถนำมาใช้กับคุณได้.

ภาคผนวก. คำประกาศ

ข้อมูลนี้ถูกพัฒนาขึ้นสำหรับผลิตภัณฑ์และบริการที่เสนอขายในประเทศสหรัฐอเมริกา.

IBM อาจไม่สามารถจัดเตรียมผลิตภัณฑ์, บริการ, หรือคุณลักษณะพิเศษที่กล่าวถึงในเอกสารนี้ในประเทศอื่นๆ ได้. ให้ปรึกษาตัวแทนจำหน่ายของ IBM สำหรับข้อมูลเกี่ยวกับผลิตภัณฑ์และบริการที่มีอยู่ในปัจจุบันในพื้นที่ของคุณ. การอ้างอิงใดๆ ถึงผลิตภัณฑ์ IBM, โปรแกรม, หรือบริการ ไม่ได้มีเจตนาในการระบุ หรือกล่าวถึงโดยนัยว่า ต้องใช้ผลิตภัณฑ์ IBM, โปรแกรม, หรือบริการดังกล่าวเท่านั้น. ผลิตภัณฑ์, โปรแกรม, หรือบริการใดๆ ที่สามารถทำงานได้เท่าเทียมกัน ที่ไม่ได้ละเมิดลิขสิทธิ์ทรัพย์สินทางปัญญาใดๆ ของ IBM อาจถูกนำมาใช้แทนได้. อย่างไรก็ตาม, เป็นความรับผิดชอบของผู้ใช้ที่จะประเมิน และตรวจสอบผลิตภัณฑ์, โปรแกรม, หรือบริการที่ไม่ใช่ของ IBM.

IBM อาจมีสิทธิบัตร หรือเอกสารซึ่งอยู่ระหว่างการดำเนินการขอสิทธิบัตรที่ครอบคลุมถึงประเด็นที่อธิบายไว้ในเอกสารนี้. การตกแต่งเอกสารใหม่ไม่ได้ทำให้คุณได้สิทธิของสิทธิบัตรเหล่านั้น. คุณสามารถสอบถามเกี่ยวกับไลเซนส์, โดยเขียนส่งไปที่:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

สำหรับการสอบถามไลเซนส์เกี่ยวกับข้อมูล double-byte (DBCS), ติดต่อแผนกทรัพย์สินทางปัญญาของ IBM ในประเทศของคุณ หรือส่งแบบสอบถามมาได้, โดยการเขียน, ไปยัง:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

ย่อหน้าต่อไปนี้ไม่ใช่กับประเทศสหราชอาณาจักร หรือประเทศอื่นที่สั่งจัดหาให้ไม่สอดคล้อง กับกฎหมายท้องถิ่น:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. บางรัฐไม่อนุญาตการสละความรับผิดชอบของการรับประกันโดยชัดแจ้ง หรือโดยนัยในธุรกรรมบางอย่าง, ดังนั้นข้อความข้างต้นนี้อาจไม่มีผลบังคับกับคุณ.

ข้อมูลนี้ได้รับความไม่ถูกต้องทางเทคนิคหรือความผิดพลาดทางการพิมพ์. การเปลี่ยนแปลงข้อมูลในนี้จะมีเป็นระยะๆ ซึ่งจะสอดคล้องกับการตีพิมพ์ในครั้งใหม่. IBM อาจทำการปรับปรุงและ/หรือ เปลี่ยนแปลงในผลิตภัณฑ์ และ/หรือ โปรแกรมที่ได้อธิบายไว้ในเอกสารนี้ได้ตลอดเวลา โดยไม่แจ้งให้ทราบล่วงหน้า.

การอ้างถึงเว็บไซต์ที่ไม่ใช่ของ IBM นั้นถูกจัดหามาเพื่อความสะดวกเท่านั้น และไม่ได้มีการรองรับเว็บไซต์เหล่านั้น. เนื้อหาในเว็บไซต์เหล่านั้นไม่ใช่ส่วนหนึ่งของเนื้อหาสำหรับผลิตภัณฑ์ IBM นี้ และการใช้งานเว็บไซต์เหล่านั้นถือเป็นความเสี่ยงของตัวเอง.

IBM อาจใช้งาน หรือเผยแพร่ส่วนใดๆ ของข้อมูลที่คุณให้ไว้ เมื่อเชื่อว่าเหมาะสมโดยไม่กำหนดภาระผูกพันใดๆ แก่คุณ.

สำหรับผู้ที่มิใช่เจ้าของโปรแกรมนี้ที่ต้องการมีข้อมูลเกี่ยวกับโปรแกรมสำหรับจุดประสงค์ให้ทำงานได้: (i) การแลกเปลี่ยนข้อมูลระหว่างโปรแกรมที่ถูกสร้างขึ้นอย่างเป็นอิสระและโปรแกรมอื่น (รวมทั้งโปรแกรมนี้) และ (ii) การใช้ข้อมูลร่วมกันที่ซึ่งมีการแลกเปลี่ยน ควรติดต่อ:

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

ข้อมูลเหล่านี้ อาจมีให้โดยขึ้นอยู่กับเงื่อนไขและสถานการณ์ที่เหมาะสม, ซึ่งรวมถึงบางกรณี, เช่น การจ่ายค่าธรรมเนียม.

- | โลเซนส์โปรแกรมที่อธิบายในข้อมูลนี้ และเนื้อหาที่มีโลเซนส์ได้จัดเตรียมไว้สำหรับโปรแกรมโดย IBM ซึ่งอยู่ภายใต้เงื่อนไข
- | ของ IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine
- | Code, หรือข้อตกลงที่เทียบเท่าระหว่างเรา.

ข้อมูลประสิทธิภาพใดๆ ที่มีอยู่ในนี้ถูกกำหนดอยู่ในสภาพแวดล้อมที่ถูกควบคุม. ดังนั้น, ผลที่ได้จากสภาพแวดล้อมของการปฏิบัติการอื่นอาจแตกต่างกันเป็นอย่างมาก. การวัดบางอย่างอาจถูกทำขึ้นบนระบบในระดับของการพัฒนา และไม่ได้มีการรับรองว่า การวัดเหล่านี้จะเหมือนกันบนระบบที่ใช้กันอยู่ทั่วไป. นอกเหนือจากนี้, การวัดบางอย่างอาจเป็นการประมาณผ่านการคาดการณ์. ซึ่งผลที่แท้จริงอาจแตกต่างกัน. ผู้ใช้เอกสารนี้ ควรทำการตรวจสอบข้อมูลที่ใช้ได้สำหรับสภาพแวดล้อมเฉพาะของพวกเขา.

ข้อมูลเกี่ยวกับผลิตภัณฑ์ที่ไม่ใช่ของ IBM ได้รับมาจากซัพพลายเออร์ของผลิตภัณฑ์เหล่านั้น, การประกาศทางสาธารณะชน หรือแหล่งที่เป็นของสาธารณะอื่นๆ. IBM ไม่ได้ทำการทดสอบผลิตภัณฑ์เหล่านั้น และไม่สามารถยืนยันความถูกต้องของประสิทธิภาพการทำงาน, การใช้แทนกันได้ หรือการเรียกร้องใดๆ ที่เกี่ยวข้องกับผลิตภัณฑ์ที่ไม่ได้เป็นของ IBM. คำถามเกี่ยวกับผลิตภัณฑ์ที่ไม่ใช่ของ IBM ควรถามไปที่ซัพพลายเออร์ของผลิตภัณฑ์เหล่านั้น.

ทุกประโยชน์ที่มีการเป็นเรื่องของทิศทางในอนาคตหรือความตั้งใจของ IBM อาจมีการเปลี่ยนแปลงหรือถอดถอนโดยไม่ต้องมีการแจ้งให้ทราบ, และเป็นการแสดงถึงจุดมุ่งหมายและวัตถุประสงค์เท่านั้น.

ราคาของ IBM ที่แสดงเป็นราคาขายปลีกที่แนะนำให้ขาย, เป็นราคาปัจจุบัน และสามารถเปลี่ยนแปลงราคาได้โดยไม่ต้องแจ้งให้ทราบล่วงหน้า. ราคาของดีลเลอร์อาจแตกต่างกัน.

ข้อมูลนี้ไว้สำหรับวัตถุประสงค์ของการวางแผนเท่านั้น. ข้อมูลในนี้อาจมีการเปลี่ยนแปลง ก่อนที่ผลิตภัณฑ์ที่อธิบายนั้นมีวางจำหน่าย.

ข้อมูลนี้มีตัวอย่างของข้อมูลและรายงานที่ใช้ในการปฏิบัติงานประจำวัน. เพื่อแสดงให้เห็นอย่างสมบูรณ์ที่สุดที่เป็นไปได้, ตัวอย่างเหล่านี้ประกอบด้วย ชื่อของแต่ละราย, ชื่อของบริษัท, ตราสินค้าและผลิตภัณฑ์. ชื่อทั้งหมดเหล่านี้ถูกทำขึ้น และคล้ายคลึงกับชื่อและที่อยู่ของหน่วยธุรกิจจริงๆ.

COPYRIGHT LICENSE:

ข้อมูลนี้ประกอบด้วยโปรแกรมแอสเซมบลีเคชันตัวอย่างในภาษาต้นฉบับ (source language), ซึ่งแสดงเทคนิคของโปรแกรมบนระบบปฏิบัติการที่หลากหลาย. คุณสามารถทำสำเนา, เปลี่ยนแปลง, และจำหน่ายโปรแกรมตัวอย่างเหล่านี้ในรูปแบบต่างๆ

โดยไม่จำเป็นต้องชำระเงินให้กับ IBM, สำหรับจุดประสงค์ในการพัฒนา, การใช้, การทำการตลาด หรือการจัดจำหน่ายแอปพลิเคชันโปรแกรมที่ใช้กับ แอปพลิเคชันโปรแกรมมิ่งอินเทอร์เน็ตเฟสสำหรับแพลตฟอร์มระบบปฏิบัติการที่โปรแกรมตัวอย่างได้ถูกพัฒนาขึ้น. ตัวอย่างเหล่านี้ไม่ได้ผ่านการทดสอบอย่างทั่วถึงภายใต้เงื่อนไขทั้งหมด. ดังนั้น, IBM, ไม่สามารถรับประกันหรือกล่าวเป็นนัยถึงความเชื่อถือ, การให้บริการ, หรือฟังก์ชันของโปรแกรมเหล่านี้.

แต่ละสำเนาหรือบางส่วนของโปรแกรมตัวอย่าง หรืองานใดๆ ที่มาจากโปรแกรมเหล่านี้ ต้องมีข้อความแสดงลิขสิทธิ์ ดังนี้:

© (ชื่อบริษัทของคุณ) (ปี). หลายส่วนของโค้ดนี้ถูกนำมาจาก IBM Corp. ตัวอย่างโปรแกรม. © Copyright IBM Corp. _ให้ป้อนปี_. สงวนลิขสิทธิ์.

ถ้าคุณกำลังดูสำเนาชั่วคราว (softcopy) ของข้อมูล, ภาพหรือสีที่แสดงอาจไม่ปรากฏ.

Programming Interface Information

เอกสารข้อมูล (ให้ใส่ชื่อหรืองานพิมพ์ที่นี่) เป็นเรื่องของโปรแกรมมิ่งอินเทอร์เน็ตเฟสที่ช่วยให้ลูกค้าสามารถเขียนโปรแกรมเพื่อให้ได้รับการบริการของ (ใส่ชื่อผลิตภัณฑ์ที่นี่).

เครื่องหมายการค้า

คำต่อไปนี้ เป็นเครื่องหมายการค้าของ International Business Machines Corporation ในประเทศสหรัฐอเมริกา, หรือในประเทศอื่น, หรือทั้งสองกรณี:

- | AIX
- | DB2
- | DB2 Universal Database
- | Distributed Relational Database Architecture
- | Domino
- | DRDA
- | e(logo)server
- | i5/OS
- | IBM
- | IBM (logo)
- | Integrated Language Environment
- | iSeries
- | Lotus Notes
- | Operating System/400
- | OS/390
- | OS/400
- | PowerPC
- | System/36

Windows เป็นเครื่องหมายการค้าของ Microsoft Corporation ในประเทศสหรัฐอเมริกา, หรือประเทศอื่น, หรือทั้งสองกรณี.

Java และเครื่องหมายการค้าที่เกี่ยวข้องกับ Java ทั้งหมดเป็นเครื่องหมายการค้าจดทะเบียนของ Sun Microsystems, Inc. ในประเทศสหรัฐอเมริกา, ประเทศอื่น, หรือทั้งสองกรณี.

I Linux เป็นเครื่องหมายการค้าของ Linus Torvalds ในประเทศสหรัฐอเมริกา, ประเทศอื่น, หรือทั้งสองกรณี.

ชื่ออื่นๆ ของบริษัท, ผลิตภัณฑ์, และการบริการ อาจเป็นเครื่องหมายการค้า หรือเครื่องหมายการบริการ ของผู้อื่น.

ข้อกำหนดและเงื่อนไข

คำอนุญาตในการใช้สิ่งพิมพ์เผยแพร่เหล่านี้เป็นไปตามข้อกำหนดและเงื่อนไขต่อไปนี้.

การใช้งานเป็นการส่วนตัว: คุณสามารถทำซ้ำสิ่งพิมพ์เผยแพร่เหล่านี้สำหรับใช้เป็นการส่วนตัว, มิใช่เพื่อการพาณิชย์โดยมีข้อแม้ว่าจะจะมีการคงค่าประกาศเกี่ยวกับกรรมสิทธิ์ทั้งหมด. คุณไม่สามารถแจกจ่าย, แสดง หรือสร้างงานที่สืบเนื่องจาก, หรือจากบางส่วนของสิ่งพิมพ์เผยแพร่เหล่านี้, โดยมีได้รับอนุญาตจาก IBM.

การใช้งานในเชิงพาณิชย์: คุณอาจทำซ้ำ แจกจ่าย และแสดงสิ่งพิมพ์เผยแพร่นี้ได้เฉพาะภายในองค์กรของคุณ โดยมีเงื่อนไขว่าจะต้องคงค่าประกาศเกี่ยวกับกรรมสิทธิ์. ไม่อนุญาตให้คุณผลิตงานที่มาจากสิ่งพิมพ์เผยแพร่, หรือทำซ้ำ, แจกจ่าย หรือแสดงสิ่งพิมพ์เผยแพร่หรือส่วนใดส่วนหนึ่งภายนอกองค์กรของคุณ, โดยไม่ได้รับความยินยอมอย่างชัดแจ้งจาก IBM.

นอกเหนือจากคำอนุญาตที่ได้แสดงไว้ในที่นี้, IBM ไม่ได้ให้อำนาจดำเนินการ, ไลเซนส์หรือสิทธิ์อื่นใด ทั้งโดยชัดแจ้งและโดยนัย กับสิ่งพิมพ์เผยแพร่หรือสารสนเทศ, ข้อมูล, ซอฟต์แวร์ หรือทรัพย์สินทางปัญญาอื่นๆ ที่อยู่ภายในที่นี้.

IBM ขอสงวนสิทธิ์ในการเพิกถอนคำอนุญาตที่ให้ไว้ในที่นี้, เมื่อใดก็ตามที่พิจารณาแล้วว่า, การใช้สิ่งพิมพ์เผยแพร่เหล่านี้ก่อให้เกิดความเสียหายต่อผลประโยชน์ของบริษัท, หรือเมื่อ IBM ได้พิจารณาแล้วว่า ไม่มีการปฏิบัติตามข้อกำหนดข้างต้นอย่างเหมาะสม.

คุณไม่สามารถดาวน์โหลด, ส่งออกหรือส่งออกข้อมูลนี้ซ้ำได้ ยกเว้นจะได้ปฏิบัติตามกฎหมายและข้อบังคับที่ผลบังคับใช้, รวมไปถึงกฎหมายและข้อบังคับในการส่งออกของสหรัฐอเมริกา.

IBM ไม่รับประกันเกี่ยวกับเนื้อหาของสิ่งพิมพ์เผยแพร่เหล่านี้. สิ่งพิมพ์เผยแพร่เหล่านี้ถูกจัดทำขึ้น "ตามสภาพที่เป็น" โดยไม่มีการรับประกันใดๆ ทั้งโดยชัดแจ้งหรือโดยนัย ตลอดจนไม่มีการรับประกันโดยนัยต่อการนำไปจัดจำหน่าย, การไม่ละเมิดสิทธิ, และความเหมาะสมสำหรับวัตถุประสงค์เฉพาะอย่างใดอย่างหนึ่ง.



พิมพีในสหรัฐอเมริกา