IBM solidDB
Version 7.0

In-Memory Database User Guide

SC27-3845-05

Note
Before using this information and the product it supports, read the information in "Notices" on page 41.

First edition, fifth revision

This edition applies to V7.0 Fix Pack 8 of IBM solidDB (product number 5724-V17) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Tables

Summary of changes

Changes for revision 05

- Editorial corrections.

Changes for revision 04

- Editorial corrections.

Changes for revision 03

- Editorial corrections.

Changes for revision 02

- Information related to concurrency control updated in section Persistent in-memory tables.

Changes for revision 01

- Editorial corrections.

About this manual

IBM® solidDB® in-memory database allows you to choose the optimal balance of maximum performance and the ability to handle large volumes of data by providing a unique dual-engine Database Management System (DBMS) architecture. Inside the database server, there are two engines: a main memory engine (MME) for fastest possible access to performance-critical data and a traditional on-disk engine for efficiently handling virtually any volume of data.

The solidDB main memory engine is built on solidDB disk-based engine and solidDB capabilities, which means that solidDB main memory engine inherits all functionality of these products. The solidDB main memory engine can be used in embedded systems, requiring virtually no administration or maintenance. You can make solidDB main memory engine suitable for highly available systems by deploying solidDB as a High Availability configuration. You can also deploy the Advanced Replication component, which enables multiple solidDB main memory engine and solidDB disk-based engine servers to share and synchronize data with each other.

This guide introduces you to the features that allow you to optimize your database server's performance by using in-memory database technology.

This guide assumes the reader has general knowledge of relational database management systems and familiarity with SQL. This guide also assumes that the reader has basic familiarity with the solidDB product family. You should read *IBM solidDB Administrator Guide* prior to reading this guide. If you are not already familiar with relational databases, you should also read the *IBM solidDB Getting Started Guide* and *IBM solidDB SQL Guide* first.

Typographic conventions

solidDB documentation uses the following typographic conventions:

Table 1. Typographic conventions

Format	Used for
Database table	This font is used for all ordinary text.
NOT NULL	Uppercase letters on this font indicate SQL keywords and macro names.
`solid.ini`	These fonts indicate file names and path expressions.
`SET SYNC MASTER YES;` `COMMIT WORK;`	This font is used for program code and program output. Example SQL statements also use this font.
`run.sh`	This font is used for sample command lines.
TRIG_COUNT()	This font is used for function names.
java.sql.Connection	This font is used for interface names.

Table 1. Typographic conventions (continued)

Format	Used for
LockHashSize	This font is used for parameter names, function arguments, and Windows registry entries.
argument	Words emphasized like this indicate information that the user or the application must provide.
Administrator Guide	This style is used for references to other documents, or chapters in the same document. New terms and emphasized issues are also written like this.
File path presentation	Unless otherwise indicated, file paths are presented in the UNIX format. The slash (/) character represents the installation root directory.
Operating systems	If documentation contains differences between operating systems, the UNIX format is mentioned first. The Microsoft Windows format is mentioned in parentheses after the UNIX format. Other operating systems are separately mentioned. There may also be different chapters for different operating systems.

Syntax notation conventions

solidDB documentation uses the following syntax notation conventions:

Table 2. Syntax notation conventions

Format	Used for
`INSERT INTO table_name`	Syntax descriptions are on this font. Replaceable sections are on *this* font.
`solid.ini`	This font indicates file names and path expressions.
[ ]	Square brackets indicate optional items; if in bold text, brackets must be included in the syntax.
|	A vertical bar separates two mutually exclusive choices in a syntax line.
{ }	Curly brackets delimit a set of mutually exclusive choices in a syntax line; if in bold text, braces must be included in the syntax.
...	An ellipsis indicates that arguments can be repeated several times.
. . .	A column of three dots indicates continuation of previous lines of code.

1 Overview of solidDB in-memory features

The solidDB main memory engine combines the high performance of in-memory tables along with the nearly unlimited capacity of disk-based tables. Pure in-memory databases are fast, but strictly limited by the size of memory. Pure disk-based databases allow nearly unlimited amounts of storage, but their performance is dominated by disk access. Even if the computer has enough memory to store the entire database in memory buffers, database servers designed for disk-based tables can be slow because the data structures that are optimal for disk-based tables are far from being optimal for in-memory tables.

The solidDB solution is to provide a single database server that contains two optimized servers inside it: one server is optimized for disk-based access and the other is optimized for in-memory access. Both servers coexist inside the same process, and a single SQL statement may access data from both engines.

1.1 In-memory versus disk-based tables

If a table is an in-memory table (M-table), the entire contents of the table are stored in memory so that the data can be accessed as quickly as possible. If a table is disk-based (D-table), the data is stored primarily on disk, and usually the server copies only small pieces of data at a time into memory.

From application design perspective, in-memory tables and disk-based tables are the same in most respects.

- Both table types provide full persistence of data unless specified differently.
- You can run the same types of queries on each of them.
- You can combine disk-based and in-memory tables in the same SQL query or transaction.
- Both table types can be used with indexes, triggers, stored procedures, and other common database objects.
- Both table types allow constraints, including primary key and foreign key constraints, although there are some limitations on foreign key constraints with non-persistent in-memory tables.

The main difference between M-tables and D-tables is performance. M-tables provide better performance; they can provide the same durability and recoverability as D-tables. For example, read operations on M-tables do not wait for disk access, even when the system is engaged in activities such as checkpointing and transaction logging.

With solidDB, you can decide which tables are in-memory tables and which tables are disk-based tables. For example, you can put heavily used tables in main memory so that they can be accessed more quickly. If you have enough memory, you can put all of your tables in main memory.

1.2 Types of in-memory tables

There are two basic types of in-memory tables: persistent tables and non-persistent tables. Persistent tables provide recoverability of data; non-persistent tables provide fast access.

Disk-based tables are always persistent tables.

1.2.1 Persistent in-memory tables

Persistent in-memory tables persist indefinitely. Although client queries access the copy of the data in memory, the server stores the persistent in-memory tables on disk when it shuts down, and therefore the data is available each time that the server starts. Persistent in-memory tables also use transaction logging; if the server is shut down unexpectedly (for example, due to a power failure), the server has a record of the transactions that have occurred and can update the tables to ensure that they have all the data from all the committed transactions. As with disk-based tables, data in persistent in-memory tables is copied to the hard disk during checkpoints.

Persistent in-memory tables can also be used with the solidDB HotStandby component; data in in-memory tables is copied to the Secondary server from where it is available if the Primary server fails.

Differences between persistent in-memory tables and disk-based tables

In most regards, in-memory tables are indistinguishable from disk-based tables, except that in-memory tables are generally significantly faster. The following sections highlight the differences between in-memory tables and disk-based tables.

Concurrency control

In-memory tables always use pessimistic row-level concurrency control (locking). Disk-based tables use optimistic (versioning) concurrency control by default.

Depending on the type of table used, the error handling needs to take different error codes into account.

Checkpointing algorithm

The checkpointing of in-memory tables is entirely different from the algorithm used on disk-based tables. Checkpointing in-memory tables does not block the transactions' access to the tables in any way during the checkpoint. Thus, the predictability of response times is better with in-memory tables than with disk-based tables.

Secondary indexes

With in-memory tables, the secondary indexes are never written to the disk. Instead, they are maintained in-memory only and rebuilt when the server is started. The impact of secondary indexes on the write performance of in-memory tables is significantly smaller than with disk-based tables. Moreover, all indexes of in-memory tables are equally fast whereas on disk-based tables, the primary key is significantly faster than the other indexes.

1.2.2 Non-persistent in-memory tables

Non-persistent in-memory tables are not written to disk when the server shuts down. Therefore, any time that the server shuts down, whether normally or abnormally,

the data in non-persistent tables is lost. Their data is not logged or checkpointed. That makes them unrecoverable but remarkably faster than persistent tables.

There are two different types of non-persistent in-memory tables: *transient tables* and *temporary tables*. The main difference between temporary tables and transient tables is that the data of a temporary table is visible to a single connection whereas data of a transient table is visible to all users.

Non-persistent tables are useful as scratchpads. For example, you can copy data from a persistent table, do a series of intensive operations on the data while it is in the temporary table, and then store the results back in a persistent table. This allows you to maximize performance, yet still keep part or all of the data when you are done. If, for some reason, your work is interrupted, the original data is still safe in the persistent table, and you can restart the processing.

Because transactions for non-persistent tables are not logged, they cannot be used with HotStandby or solidDB Universal Cache.

Temporary tables

Data in temporary tables is visible only to the connection that inserted the data, and the data is retained only for the duration of the connection. Temporary tables are like private scratchpads that no one else can see. Temporary tables are even faster than transient tables because they do not use logging or any type of concurrency control mechanism (such as record locking).

Limited visibility

Data in temporary tables has limited visibility because only the session (connection) that inserted the data can see it.

If your session creates a temporary table and inserts data into it, no other user session can see your data, even if you grant privileges on that table. Multiple sessions can use the same table simultaneously, but each session can see only its own data.

Since each session can see only its own data, you do not need to coordinate with other sessions to make sure that you insert unique values into the table, even if the table has a unique constraint. For example, if you create a temporary table that has a unique constraint on the ID column, you and another session might both insert records that have the ID set to the value 1. Since each session sees only its own data, operations such as UPDATE and DELETE affect only the data in the session.

Limited duration

Data in temporary tables has limited duration because as soon as you exit your current session (disconnect from the server), the data is discarded. If you connect again, you cannot see your data.

The word *temporary* in the term *temporary tables* refers to the data, not the table itself. The server stores the definition of the temporary table (but not the data) in the system tables and keeps that definition even after you disconnect. Thus, if you reconnect to the server later, you the table still exists, but it is empty. When you create the table, you do not need to create it again in future sessions. In fact, if you or another user try to create a temporary table with the same name as an existing

temporary table, you get an error message. The behavior can be unexpected if you think that a temporary table means that the table (not just the data) disappears as soon as you disconnect.

Because the tables persist (even though the data does not), use the DROP TABLE command to drop the table definition after you no longer need it. Also, because the table persists, if you export a database schema definition, the output includes the commands to re-create the temporary tables.

Because temporary tables are cleared when the user disconnects, the processor usage can seem high for some time after a session with a large amount of temporary table data.

Other characteristics

- If you use the HotStandby component, data in temporary tables is not replicated to the Secondary server. However, temporary table definitions themselves are replicated to the Secondary server. Thus, if you want to fail over to your Secondary, you do not need to re-create any temporary tables that are already created. However, you must re-create any data in them.
- In Universal Cache, if solidDB is as a source datastore, temporary tables are not supported and they cannot be part of a subscription. Temporary tables can be used in a subscription where solidDB is the target datastore.
- Temporary tables can be used only as replica tables in advanced replication systems, not as master tables.
- Temporary tables have restrictions on how they can be used with referential constraints. A temporary table can reference another temporary table, but it cannot reference transient or persistent tables. No other type of table can reference a temporary table.

With the exceptions of the limitations listed in this section, temporary tables behave like normal (persistent) in-memory tables. For example,

- Temporary tables can have indexes on them.
- Temporary tables can be used in Views.
- Temporary tables can have triggers on them.
- Temporary tables can contain BLOB columns (but the length of those columns is limited to a couple of kilobytes).
- Temporary tables exist in a specific catalog and schema.
- Privileges apply to temporary tables; in other words, the creator of the temporary table can grant and revoke privileges on the table. The DBA can also grant and revoke privileges on the table. However, when a session puts data into a temporary table, the data cannot be seen by any other session, even if that session is by a DBA or a user that has SELECT privilege on the temporary table. Therefore, granting privileges on a table merely grants the other user the right to use your table, not your data. Default privileges on temporary tables are the same as the default privileges on persistent tables.

Standards compliance

The solidDB implementation of temporary tables fully complies with the ANSI SQL:1999 standard for *global temporary tables*. All solidDB temporary tables are global. In the CREATE TABLE syntax, the keyword GLOBAL is supported for compatibility reasons. However, even if the keyword GLOBAL is not specified, all temporary tables are global.

The solidDB server does not support *local temporary tables* as defined by ANSI.

Transient tables

Transient tables last until the database server shuts down. Multiple users can use the same transient table, and each user sees the data of all other users.

In most regards, transient tables behave like standard (persistent) in-memory tables. For example:

- Data in transient tables has the same scope or visibility as data in persistent tables. The data that you insert into a transient table can be seen by other users' sessions, if those users have appropriate privileges.
- Transient tables can be used in views.
- Transient tables can have indexes on them.
- Transient tables can have triggers on them.
- Transient tables can contain BLOB columns. However, the length of BLOB columns is limited to few KB in all in-memory tables.
- Privileges apply to transient tables.
- Transient tables reside in a specific catalog and schema.
- You can import data into transient tables by using the solidDB Speed Loader (**solload**) utility.

If you export a database with a transient table, the data in the transient tables and the structure of the tables are exported.

The server stores the definition of the transient table (but not the data) in the system tables and keeps that definition even after the server is shut down. If you restart the server later, the table still exists, but the data does not. Thus, you need to create the table only once. In fact, if you or another user try to create a transient table with the same name as an existing transient table, you get an error message, even if the server has been shut down and restarted since the time that the table with that name was originally created. This behavior can be unexpected if you think that a transient table disappears as soon as you shut down the server.

Also, since a transient table persists (even though the data does not), you can use the DROP TABLE command to drop the table after you no longer need it.

Limitations

Transient tables have some limitations when compared to persistent in-memory tables.

- Transient table data is not replicated to the Secondary server when you use the HotStandby component. Transient tables themselves (but not their data) are replicated to the HotStandby Secondary server. Thus, for failover to Secondary, you do not have to re-create transient tables. However, you must re-create any data in them.
- In Universal Cache, if solidDB is as a source datastore, transient tables are not supported and they cannot be part of a subscription. Transient tables can be used in a subscription where solidDB is the target datastore.
- Transient tables have restrictions on how they can be used with referential constraints. Transient tables can reference other transient tables and persistent tables. They cannot reference temporary tables. Temporary tables and persistent tables cannot reference a transient table.

- Transient tables can be used only as replica tables in advanced replication systems, not as master tables.

Standards compliance

Transient tables are not defined by the ANSI standard for SQL. Transient tables are a solidDB extension to the SQL standard.

Differences between temporary and transient tables

The main differences between temporary tables and transient tables are:

- Transient tables allow all sessions (connections) in the system to see the same data. Temporary tables allow only the user who created a piece of data to see that data.
- Because users may access the same data, transient tables use concurrency control. Only pessimistic concurrency control (locking) is supported.
- Temporary tables are faster than transient tables because they do not use concurrency control.
- The data in transient tables lasts until the server is shut down, while data in temporary tables lasts only until the user logs out of the session. This means that if one session inserts data into a transient table, then other sessions may see that data even after the creator of the data disconnects.
- Data in transient tables is exportable using solexp tool. Data in temporary tables is not.
- The referential integrity rules for the two table types are different.

1.2.3 Table types and referential integrity

The persistent and non-persistent table differ in reference to referential integrity.

The following table shows which table types are allowed to refer to other types. For example, if a transient table is allowed to have a foreign key that references a persistent table, you will see "YES" in the cell at the intersection of the row "Transient Child" and the column "Persistent Parent". If the foreign key constraint is not allowed, you will see a dash (-).

Every type of table may reference itself. In addition, transient tables may reference persistent tables (but not vice-versa). All other combinations are invalid.

REFERENCED TABLE **REFERENCING TABLE**	**Persistent Disk-based Table**	**Persistent In-Memory Table**	**Transient Table**	**Temporary Table**
Persistent Disk-Based Table	YES	YES	-	-
Persistent In-Memory Table	YES	YES	-	-
Transient Table	YES	YES	YES	-

REFERENCED TABLE REFERENCING TABLE	Persistent Disk-based Table	Persistent In-Memory Table	Transient Table	Temporary Table
Temporary Table	-	-	-	Yes

1.3 Considerations for developing applications with in-memory tables

Before starting to develop applications with in-memory tables, review the following considerations on performance, memory usage, transaction isolation, and using M-tables with HotStandby or shared memory (SMA) and linked library (LLA) access methods.

1.3.1 Performance and in-memory tables

If data is stored in a disk-based table, it must be read into memory before it can be used, and it must be written back to the disk after it has been used. In-memory tables provide higher performance because all the data resides always in the main memory; the server may use more efficient techniques to provide the maximum performance for accessing and manipulating data.

Almost any database server will perform faster if it has more memory and can store a larger percentage of its data in the cache memory of the server. However, solidDB main memory engine's high-performance in-memory technology does much more than merely copy data into memory. The solidDB main memory engine also uses index structures that are optimized to work with data that is stored entirely in memory. The solidDB main memory engine also takes into account issues that arise with in-memory tables, such as memory "fragmentation" when tables grow or shrink.

Temporary and transient tables and performance

Temporary and transient tables provide higher performance than persistent tables for the following reasons:

- Data in temporary tables and transient tables is stored solely in memory; it is never written to disk. If you shut down and restart the server, or if the server terminates abnormally, the data is lost. In the case of temporary tables, the data is discarded at the end of the user session -- it does not even remain until the server is shut down.
- Temporary and transient tables do not log transaction data to disk. The data is not recoverable after an abnormal server termination.
- When the server does its periodic checkpoint operations, which write database data to the disk drive, the data in temporary tables and transient tables is not written to the disk.
- Temporary tables and transient tables use a more efficient data storage structure than regular in-memory tables use.
- Temporary tables have a further performance advantage over transient tables. Sessions (connections) do not see each other's records in a temporary table, and therefore they do not need sophisticated concurrency control – for example, there is no need to check for locking conflicts on records within the table.

Indexes

If a table is stored in memory, all indexes on that table are also stored in memory. This improves performance but also consumes memory space. In general, in-memory indexes can be extremely fast, and you should use them to ensure fast access to the data of the tables. However, if you do not have enough memory to store all your tables and indexes in memory, adding a particular index might not help in all cases. This is because even though it will speed up some queries, it will slow up other queries by using memory that otherwise could be used to put other tables in memory.

1.3.2 Physical memory and virtual memory

The total size of the in-memory database tables cannot exceed the amount of virtual memory available.

Important:

Since virtual memory is swapped to disk frequently, using virtual memory negates part of the advantage of in-memory tables. You should limit your in-memory tables to less than the size of the available physical memory, not the size of the available virtual memory.

When calculating the amount of space required for tables, do not forget BLOB data. Generally, BLOB data should be kept on disk-based tables as the maximum size of a BLOB column is significantly reduced on main-memory tables.

The amount of space required to store a table includes the space not only for the data that is in the table, but also for any indexes on that table, including any indexes created in support of primary key and foreign key constraints. Also, tables occupy significantly more space in memory than on disk.

If the server runs out of virtual memory when it tries to allocate memory (for example, to expand a table during an INSERT or ALTER TABLE operation), you will get an error message.

1.3.3 Transaction isolation limitations with in-memory tables

The SERIALIZABLE isolation level is not supported with M-tables.
You cannot use in-memory tables in transactions where the transaction isolation level is SERIALIZABLE. The levels of transaction isolation that are supported for in-memory tables are REPEATABLE READ and READ COMMITTED.

For in-memory tables in the HotStandby secondary server, the transaction isolation level is always READ COMMITTED.
If you are using HotStandby and you connected to the HotStandby secondary server, when you read data from in-memory tables, the transaction isolation level is automatically set to READ COMMITTED, even if you specified REPEATABLE READ.

REPEATABLE READ related differences between M-tables and D-tables.
If you use in-memory tables and have set the transaction isolation level to REPEATABLE READ (default is READ COMMITTED), read operations block write operations for the duration of the read transaction. Moreover, with READ REPEATABLE isolation level, deadlocks are possible with

in-memory tables whereas they cannot occur on versioning disk-based tables. On the other hand, concurrency conflicts can occur when optimistic concurrency control is in use.

1.3.4 Shared memory access and linked library access

Shared memory access (SMA) and linked library access (LLA) provide the solidDB server in the form of a linkable library. You can link you application directly to the SMA or LLA library and access it via function calls without going through a network communications protocol.

SMA and LLA are compatible with in-memory tables.

For more information, see Overview of shared memory access and linked library access.

For more information about SMA and LLA, see the *IBM solidDB Shared Memory Access and Linked Library Access User Guide*.

1.3.5 HotStandby and in-memory tables

In-memory tables can be used with solidDB High Availability, given the following considerations:

- Persistent in-memory tables are replicated from the HotStandby Primary server to the Secondary server.
- Temporary tables and transient tables are not replicated to the Secondary server.

2 Working with in-memory tables

When creating tables, you specify the table type you want to use as part of the CREATE TABLE statement. By default, new tables are created as persistent in-memory tables. You can also change the table type of tables that have no data.

2.1 How to decide which tables to designate as in-memory tables

Ideally your computer would have enough memory to store all of your tables in memory and thus would provide the best possible performance for database transactions. However, in practice most users will have to choose a subset of tables to store in memory, while the remaining tables will be disk-based.

If you cannot fit all tables in memory, try to put the most-frequently-used data in memory. In principle, small frequently-used tables should go into memory, and large rarely-used tables can be left on disk. With other possible combinations, such as large tables that are heavily used, or small tables that are not heavily used, the table type should depend on the "density" of access to a table. In-memory tables work best the higher the number of accesses is per megabyte per second.

Once you have decided to store a table in memory, you must choose whether to store the data in a persistent table, a transient table, or a temporary table. The basic guidelines are as shown below.

You can decide on the most appropriate type of table by asking the questions below until you reach the first question for which you answer "Yes".

1. Do you need the data to be available again the next time that the server starts? If yes, use a persistent table.
2. Do you need the data to be copied to the Secondary HotStandby server? If yes, use a persistent table.
3. Do you need the data only during the current server session, but the data must be available to multiple users (or multiple connections from the same user)? If yes, use a transient table.

   The term *server session* refers to a single run of the server, from the time that it starts until the time that it is either deliberately shut down or it goes down for an unexpected reason (such as a power failure). A *connection* lasts from the time that a single user connects to the server until the time that user disconnects the same connection. A user may establish multiple connections, but each of these is independent.
4. If none of the above rules applied, use a temporary table.

Note: Temporary and transient tables have restrictions that might affect your decision. For example, a temporary table can reference another temporary table, but it cannot reference transient or persistent tables. No other type of table can reference a temporary table.

Related information:

Appendix A, “Algorithm for choosing which tables to store in memory,” on page 27

"Temporary tables" on page 3
Data in temporary tables is visible only to the connection that inserted the data, and the data is retained only for the duration of the connection. Temporary tables are like private scratchpads that no one else can see. Temporary tables are even faster than transient tables because they do not use logging or any type of concurrency control mechanism (such as record locking).

"Transient tables" on page 5
Transient tables last until the database server shuts down. Multiple users can use the same transient table, and each user sees the data of all other users.

2.2 Creating in-memory and disk-based tables

There are two ways to explicitly specify whether tables are to be located in-memory or on disk.

1. Use the STORE MEMORY or STORE DISK clause of the CREATE TABLE or ALTER TABLE command.

   ```
   CREATE TABLE employees (name CHAR(20)) STORE MEMORY;
   CREATE TABLE ... STORE DISK;
   ALTER TABLE network_addresses SET STORE MEMORY;
   ```

   For more information about the syntax of the CREATE TABLE and ALTER TABLE statement, see the *IBM solidDB SQL Guide*.

2. Specify the default with the **General.DefaultStoreIsMemory** parameter.

   For example:

   ```
   [General]
   DefaultStoreIsMemory=yes
   ```

   When **General.DefaultStoreIsMemory** is set to 'yes', new tables are created as in-memory tables unless specified otherwise in the CREATE TABLE statement. If this parameter is set to 'no', new tables are created as disk-based tables unless specified otherwise in the CREATE TABLE statement.

Note: These instructions apply to persistent tables only. Tables that are declared to be temporary tables or transient tables are automatically stored in memory, even if you do not use the STORE MEMORY clause.

2.3 Creating temporary and transient tables

By default, when you create an in-memory table, the table is persistent. To create temporary or transient tables, use the keyword TEMPORARY or TRANSIENT.

Creating temporary tables

To create a temporary table, use the following command:

```
CREATE [GLOBAL] TEMPORARY TABLE <...>;
```

where

`GLOBAL` is supported for compatibility reasons. However, even if the keyword GLOBAL is not specified, all temporary tables are global.

`<...>` denotes syntax that is the same as for any other type of table.

Temporary tables are always in-memory tables. If you use the STORE DISK clause, the server will give you an error. If you use STORE MEMORY, or if you omit the

STORE clause altogether, the server will create the temporary table as an in-memory table.

Creating transient tables

To create a transient table, use the following command:

```
CREATE TRANSIENT TABLE <...>;
```

where

`<...>` denotes syntax that is the same as for any other type of table.

Transient tables are always in-memory tables. If you use the STORE DISK clause, the server will give you an error. If you use STORE MEMORY, or if you omit the STORE clause altogether, the server will create the transient table as an in-memory table.

Related information:

1.2, “Types of in-memory tables,” on page 2
There are two basic types of in-memory tables: persistent tables and non-persistent tables. Persistent tables provide recoverability of data; non-persistent tables provide fast access.

2.4 Changing a table from in-memory to disk-based or vice-versa

If the table is empty, you can alter the type of a table from in-memory table to disk-based table or vice versa. To do this, use the following command:

```
ALTER TABLE table_name SET STORE MEMORY | DISK
```

If the table contains data, you need to create a new table with different name to which you copy the data. After copying the data to the new table, you can drop the old table and rename the new table with the same name as the original table.

3 Configuring in-memory database

You can configure solidDB to create new tables as in-memory tables (M-tables) by default by setting the **General.DefaultStoreIsMemory** parameter to yes. Most other in-memory features are configured using the parameters in the [MME] section of the solid.ini file.

You need to pay special attention to controlling memory consumption; if the in-memory database or the server process uses up all of the available virtual memory in the system, you will be unable to add or update data. If the server uses up all of the physical memory and starts to use virtual memory, the server will continue to operate, but performance will be greatly reduced.

3.1 Configuration parameters

Most parameters related to the solidDB in-memory database are stored into the [MME] section of the solid.ini configuration file.

You can change configuration parameters in either by manually editing the solid.ini configuration file or by entering the following command in solidDB SQL Editor:

```
ADMIN COMMAND 'parameter section_name.param_name=value'
```

For example:

```
ADMIN COMMAND 'parameter mme.imdbmemorylimit=1gb';
```

Note:

The server reads the configuration file only when it starts, and therefore changes to the configuration file do not take effect until the next time that the server starts.

3.1.1 General section

Table 3. MME related parameters in the [General] section

[General]	Description	Factory Value	Access Mode
DefaultStoreIsMemory	If set to yes, new tables are created as in-memory tables, unless they are created without an explicit STORE clause in the CREATE TABLE statement. If set to no, new tables are stored on disk by default. You can override the factory value by using the STORE clause in the CREATE TABLE statement. **Note:** System tables are stored on disk, even if this parameter is set to yes.	yes	RW

Table 3. MME related parameters in the [General] section (continued)

[General]	Description	Factory Value	Access Mode
MultiprocessingLevel	This parameter defines the number of processing units (processors, cores) available in the computer system. Typically, the concurrency of write operations in the database can be improved if the value matches the number of physical processors (cores) in your system. The factory value is read from the system as the number of logical processing units. The auto-detected value is output to `solmsg.out` at server startup. With some processor architectures, the number of logical processing units might not be the same as the number of physical cores. In such cases, the optimal value for this parameter typically varies between the number of the physical cores and the number of logical processing units. **Note:** the value of the **`MME.RestoreThreads`** parameter defaults to the value of this parameter, unless you set it to a different value explicitly.	Read from system	RW/Startup

3.1.2 MME section

Table 4. MME parameters

[MME]	Description	Factory Value	Access Mode

ImdbMemoryLimit	This sets an upper limit on the amount of memory (virtual memory) that the server will allocate for in-memory tables and indexes on in-memory tables. In-memory tables includes Temporary Tables and Transient Tables, as well as persistent in-memory tables.
	
	The limit may be specified in bytes, kilobytes (KB), megabytes (MB), or gigabytes (GB). For example:
	`ImdbMemoryLimit=1073741824`
	`ImdbMemoryLimit=1048576kb`
	`ImdbMemoryLimit=1024MB`
	`ImdbMemoryLimit=1GB`
	
	Value 0 means "no limit".
	
	As a general rule, for servers with 1 GB or less memory, the maximum amount that you should allocate to in-memory tables is usually 30% - 70% of the system's physical memory. The more memory the system has, the larger the percentage of it you may use for in-memory tables.
	Note: This parameter only applies only to solidDB main memory engine tables. It does not apply to disk-based tables.
	
	You can change this parameter with the command:
	`ADMIN COMMAND 'parameter MME.ImdbMemoryLimit=n[kb|mb|gb]';`
	
	where 'n' is a positive integer. You may only increase, not decrease, this value while the server is running. The command takes effect immediately. The new value is written back to the `solid.ini` file at shutdown.
	Important: Ensure that your in-memory tables will fit within the available physical memory. If you exceed the amount of physical memory available, performance will decrease significantly. If you use up all of the available virtual memory, the server will abruptly limit inserts, updates, and so on, and will return error codes.
		0
		
		Unit: 1 byte
		k=KB m=MB
		g=GB
			RW

ImdbMemoryLowPercentage	Once you have set **ImdbMemoryLimit**, you may set this additional parameter to give you advance warning before you use up all of memory. This **ImdbMemoryLowPercentage** parameter allows you to indicate what percentage of memory you may use before the server starts limiting your ability to insert rows into in-memory tables, and so on. For example, if **ImdbMemoryLimit** is 1000MB and **ImdbMemoryLowPercentage** is 90 (percent), then the server will stop accepting inserts when you've used up 900 megabytes of memory for your in-memory tables. Valid values are between 60 and 99 (percent). **Note:** This parameter only applies to solidDB main memory engine tables.	90	RW
ImdbMemoryWarningPercentage	This parameter sets a warning limit for the IMDB memory size. The warning limit is expressed as a percentage of the **ImdbMemoryLimit** parameter value. When the **ImdbMemoryWarningPercentage** limit is exceeded, a system event is given. The **ImdbMemoryWarningPercentage** parameter value is automatically checked for consistency. It must be lower than the **ImdbMemoryLimit** parameter value. **Note:** This parameter only applies to solidDB main memory engine tables. It does not apply to disk-based tables.	80	RW

Table 4. MME parameters (continued)

[MME]	Description	Factory Value	Access Mode
LockEscalationEnabled	Typically, when the server needs to use locks to prevent concurrency conflicts, the server locks individual rows. This means that each user affects only those other users who want to use the same row(s). However, the more rows are locked, the more time the server must spend checking for conflicting locks. In some cases, it is worthwhile to lock an entire table rather than a large number of the rows in that table. When this parameter is set to `yes`, the lock level is escalated from row-level to table-level after a specified number of rows (in the same table) have been locked within the current transaction. Lock escalation improves performance, but reduces concurrency, because it means that other users are temporarily unable to use the same table, even if they want to use different rows within that table. See also the parameter **`LockEscalationLimit`**. Possible values are `yes` and `no`. **Note:** This parameter applies to in-memory tables only.	no	RW/Startup
LockEscalationLimit	If **`LockEscalationEnabled`** is set to `yes`, this parameter indicates how many rows must be locked (within a single table) before the server will escalate lock level from row-level to table-level. See **`LockEscalationEnabled`** for more details. The value may be any number from 1 to 2,147,483,647 (2^32-1). **Note:** This parameter applies to in-memory tables only.	1000	RW/Startup
LockHashSize	The server uses a hash table (array) to store lock information. If the size of the array is remarkably underestimated the performance degrades. Too large hash table doesn't affect directly to the performance although it causes memory overhead. The **`LockHashSize`** determines the number of elements in hash table. This information is needed when the server is using pessimistic concurrency control (locking). The server uses separate arrays for in-memory tables and disk-based tables. This parameter applies to in-memory tables. In general, the more locks you need, the larger this array should be. However, it is difficult to calculate the number of locks that you need, so you may need to experiment to find the best value for your applications. The value that you enter is the number of hash table entries. Each table entry has a size of one pointer (4 bytes in 32-bit architectures). Thus, for example, if you choose a hash table size of 1,000,000, then the amount of memory required is 4,000,000 bytes (assuming 32-bit pointers).	1000000	RW/Startup
MaxBytesCachedInPrivateMemoryPool	This parameter defines the maximum bytes stored into the free list of MME's private memory pool (private memory pool is private for each main-memory index). If there is more free memory in the private pool, the extra memory is merged into global pools. Value 0 means immediate merge to global pool, usually degrades performance, but minimizes memory footprint. There is no maximum value; the default value of 100000 gives good performance with little memory overhead.	100000	RW/Startup
MaxCacheUsage	The value of **`MaxCacheUsage`** limits the amount of D-table cache used while checkpointing M-tables. The value is expected to be given in bytes. Regardless of the value of the **`MaxCacheUsage`** at most half of the D-table cache (**`IndexFile.CacheSize`**) is used for checkpointing M-tables. Value **`MaxCacheUsage=0`** sets the value unlimited, which means that the cache usage is **`IndexFile.CacheSize`**/2.	8MB	RW/Startup

Table 4. MME parameters (continued)

[MME]	Description	Factory Value	Access Mode
MaxTransactionSize	This parameter defines the maximum approximate size of a transaction in bytes. Some MME transactions (for example, DELETE FROM <table>) might cause solidDB to allocate a lot of memory for the operation. This can lead to an out-of-memory situation where solidDB cannot allocate any more memory from the operating system, and performs an emergency exit. To prevent this, use this parameter to define the maximum approximate size (in bytes) for each MME transaction; when the transaction size exceeds the value set with this parameter, the transaction fails with the error `SOLID Database error 16509: MME transaction maximum size exceeded`. Value 0 means unlimited.	0	RW

MemoryPoolScope	This parameter sets the memory pool scope. Possible values are `Global` and `Table`.
	
	When set to `Table`, only objects that belong to the same database table are allocated from a single memory segment. This ensures, for example, that dropping a whole table frees the memory segment back to operating system. Only unused memory segments can be returned back to system.
	
	When set to `Global`, memory pools are shared between all MME data.
	
	When **`MME.MemoryPoolScope`** is set to `Table`, you can use the DESCRIBE <table> statement to view the memory consumption for the table. For example:
	`DESCRIBE tmemlimit_tab;`
	`RESULT`
	`------`
	`Catalog: DBA`
	`  Schema: DBA`
	`    Table: TMEMLIMIT_TAB`
	`      Table type: in-memory`
	
	`      Memory usage: 7935 KB (total), 7925 KB (active),`
	`                    6192 KB (rows), 1733 KB (indexes).`
	
	`...`
	`1 rows fetched.`
		Global	RW/Startup

NumberOfMemoryPools	This parameter defines the number of global memory pools. Bigger values may give better performance on multicore systems with certain load scenarios but they also increase memory slack and hence server process size. Minimum value is 1. There is no maximum value; however, the number of cores in the system should not be exceeded.	1	RW/Startup
ReleaseMemoryAtShutdown	When set to `yes`, at shutdown, the server releases the memory used by M-tables explicitly, rather than relying on the operating system to clean up all memory associated with this process. Some operating systems may require you to set this to `yes` to ensure that all memory is released. The possible values are `yes` and `no`. The factory value is `no` because shutting down the server is faster that way.	no	RW/Startup

Table 4. MME parameters (continued)

[MME]	Description	Factory Value	Access Mode
RestoreThreads	This parameter defines the maximum number of threads used while restoring in-memory database during database startup. If you do not set this parameter explicitly, the value of this parameter is set to the same value as **General.MultiprocessingLevel**. Possible values are between 1 and 65536. Value 1 means that the load is executed in single thread. With invalid values, this parameter defaults to the value of **General.MultiprocessingLevel**. In-memory database restore assigns one thread per each table if the number of tables is smaller or equal to the number of the parameter value. Maximal concurrency is reached when the parameter value is smaller than the following two values: number of cores/processors, and the number of tables in the database.	Same as **General. Multiprocessing Level**	RW/Startup

3.2 Memory consumption

The in-memory database main memory usage differs from the standard solidDB. The in-memory database resides in its own memory pool.

solidDB main memory engine provides commands and configuration parameters to help you monitor and control memory consumption of the in-memory database and the server process. These commands and parameters focus on the server's in-memory database feature, not the server as a whole.

3.2.1 Monitoring memory consumption

There are several ADMIN COMMANDs available for monitoring memory consumption.

ADMIN COMMAND 'info imdbsize'

The **ADMIN COMMAND 'info imdbsize'** command returns the current amount of memory allocated to use by in-memory database tables and indexes. The value returned is a VARCHAR, and it indicates the number of kilobytes used by the server. The command returns the amount of virtual memory used, not the amount of physical memory used.

In time, the value of **imdbsize** can grow, because returning data back to operating system can only be done in allocation units which need to be completely unused before they can be returned back to the operating system.

Transient memory allocations (such as SQL execution graphs) are excluded from the **ADMIN COMMAND 'info imdbsize'** report.

ADMIN COMMAND 'info processsize'

The **ADMIN COMMAND 'info processsize'** command returns the virtual memory process size, that is, the full address space size of the database server that the in-memory database process uses. The value returned is a VARCHAR, and it indicates the number of kilobytes used by the process. This command returns the amount of virtual memory used, not the amount of physical memory used.

ADMIN COMMAND 'pmon mme'

There are also several performance counters available, which include the runtime information related to the in-memory database server.

The **ADMIN COMMAND 'pmon mme'** command produces the following list of current values of counters.

```
RC TEXT
-- ----
 0 Performance statistics:
 0 Time (sec)                              30    21    Total
 0 MME current number of locks       :     0     0        0
 0 MME maximum number of locks       :     0     0        0
 0 MME current number of lock chains :     0     0        0
 0 MME maximum number of lock chains :     0     0        0
 0 MME longest lock chain path       :     0     0        0
 0 MME memory used by tuples         :     0     0        0
 0 MME memory used by indexes        :     0     0        0
 0 MME memory used by page structures:     0     0        0
10 rows fetched.
```

In the performance statistics listing, the amount of memory used by tuples, indexes, and page structures is given in KB.

ADMIN COMMAND 'memory'

The **ADMIN COMMAND 'memory'** command reports the amount of dynamically allocated heap memory. In heap-based memory allocation, memory is allocated from a large pool of unused memory area called the heap. The size of the heap memory allocation can be determined at runtime. Transient memory allocations (such as SQL execution graphs) are included in the **ADMIN COMMAND 'memory'** report.

3.2.2 Controlling memory consumption

The in-memory database memory consumption is controlled by the following three configuration parameters in the [MME] section of the solid.ini file:

- **ImdbMemoryLimit**
- **ImdbMemoryLowPercentage**
- **ImdbMemoryWarningPercentage**

Additionally, the *process* memory consumption is controlled by the following four configuration parameters in the [SRV] section of the solid.ini file:

- **ProcessMemoryLimit**
- **ProcessMemoryLowPercentage**
- **ProcessMemoryWarningPercentage**
- **ProcessMemoryCheckInterval**

The violations of the in-memory database memory and process limits are logged in the solmsg.out log file. Every time the memory limit defined with the **ImdbMemoryLimit** and **ProcessMemoryLimit** parameters is crossed, a system event is posted. These system events are described in *IBM solidDB SQL Guide*.

Memory consumption

The in-memory database main memory usage differs from the standard solidDB. The in-memory database resides in its own memory pool.

solidDB main memory engine provides commands and configuration parameters to help you monitor and control memory consumption of the in-memory database and the server process. These commands and parameters focus on the server's in-memory database feature, not the server as a whole.

MME.ImdbMemoryLimit: The **MME.ImdbMemoryLimit** parameter specifies the maximum amount of virtual memory that can be allocated to in-memory tables (including temporary tables and transient tables) and the indexes on those in-memory tables.

The default value for **MME.ImdbMemoryLimit** is 0, which means "no limit". You should not use the default value; instead, set the parameter to a value that will ensure that the in-memory data will fit entirely within physical memory. Consider also the following factors:

- the amount of physical memory in the computer
- the amount of memory used by the operating system
- the amount of memory used by solidDB (the program itself)
- the amount of memory set aside for the solidDB server's cache (the **IndexFile.CacheSize** `solid.ini` configuration parameter)
- the amount of memory required by the connections, transactions and statements running concurrently in the server. The more concurrent connections and active statements there are in the server, the more working memory the server requires. Typically, you should allocate at least 0.5 MB of memory for each client connection in the server.
- the memory used by other processes (programs and data) that are running in the computer

When 100% of the memory specified by **MME.ImdbMemoryLimit** is reached, the server will prohibit UPDATE operations on in-memory tables. Before the limit is reached, the server will prohibit creation of new in-memory tables and INSERT operations on those tables. See “MME.ImdbMemoryLowPercentage” for more details.

Example:

```
[MME]
ImdbMemoryLimit=1000MB
```

MME.ImdbMemoryLowPercentage: The **MME.ImdbMemoryLowPercentage** parameter sets a "low water mark" for the amount of virtual memory that can be allocated to in-memory tables. The limit is expressed as a percentage of the **MME.ImdbMemoryLimit** parameter value.

When the server has consumed the percentage of memory specified with **MME.ImdbMemoryLowPercentage**, the server will start to limit activities in order to prevent memory consumption from continuing to grow. For example, if **MME.ImdbMemoryLimit** is 1000 megabytes and **MME.ImdbMemoryLowPercentage** is 90%, the server will start limiting activities if the memory allocated to the in-memory tables exceeds 900 megabytes. Specifically, the server will:

- Prohibit further creation of in-memory tables (including temporary tables and transient tables) and indexes on in-memory tables.
- Prohibit INSERTs into in-memory tables.

When the limit set with **MME.ImdbMemoryLimit** itself is reached, the server will also prohibit UPDATE operations on records in in-memory tables.

Valid values for **MME.ImdbMemoryLowPercentage** range between 60-99 (percent).

MME.ImdbMemoryWarningPercentage: The **MME.ImdbMemoryWarningPercentage** parameter sets a limit at which a system even is given to warn you that the maximum amount of virtual memory that can be allocated to in-memory tables is being reached.

The warning limit is expressed as a percentage of the **MME.ImdbMemoryLimit** parameter value. When the **MME.ImdbMemoryWarningPercentage** limit is exceeded, a system event is given.

Troubleshooting MME.ImdbMemoryLimit:

If you get an error message indicating that the limit set with **MME.ImdbMemoryLimit** has been reached, you need to take action immediately.

You must address both the immediate problems and the long term problems. The immediate problems are to prevent users from experiencing serious errors, and to free up some memory before shutting down the server so that your system is not out of memory when you restart the server. For long term, you need to ensure that you will not run out of memory in the future as tables expand.

Resolving the immediate problem

To address the immediate problem, you typically need do the following:

1. Notify users that they should disconnect from the server. This will accomplish two things: it will minimize the number of users who will be impacted if the situation deteriorates. Also, if any of the users who disconnect were using temporary tables, disconnecting will free up memory. You may wish to have a policy or error-checking code to ensure that users and/or programs will attempt to disconnect gracefully if they see this error.
2. If there were not enough temporary tables to free memory, drop some transient table indexes or transient tables if any exist.

If there were not enough temporary tables and transient tables to free enough memory, do the following:

1. Drop one or more indexes on in-memory tables.
2. Shut down the server.
3. If there was absolutely nothing in memory that you could discard (for example, you had only normal in-memory tables, none of which had indexes, and all of which had valuable data), increase the **MME.ImdbMemoryLimit** slightly before restarting the server. This may force the server to start paging virtual memory which will greatly reduce performance, but it will allow you to continue using the server and address the long-term problems. If you previously set the **ImdbMemoryLimit** a little bit lower than the maximum, you will be able to raise it slightly now without forcing the system to start paging virtual memory.
4. Restart the server.
5. Minimize the number of people using the system until you have had time to address the long-term problem. Ensure that users do not create temporary tables or transient tables until the long-term problem has been addressed.

Resolving the long term problem

After you have solved the immediate problem and have ensured that the server has at least some free memory, you are ready to address the long term problems.

For long term, reduce the amount of data stored in in-memory tables. The ways to do this are to reduce the number or size of in-memory tables (including temporary tables and transient tables), or reduce the number of indexes on in-memory tables.

- If the problem was caused solely by heavy usage of temporary or transient tables, ensure that not too many sessions create too many large temporary or transient tables at the same time.
- If the problem was caused by using too much memory for normal in-memory tables, and if you cannot increase the amount of memory available to the server, move one or more tables out of main memory and onto the disk.

To move a table from memory to disk, do the following:

1. Create an empty disk-based table with the same structure (but a different name) as one of the tables in memory.
2. Copy the information from the in-memory table to an intermediate disk-based table.

   If you try to copy records of a large table to another table using a single SQL statement (INSERT INTO ...VALUES SELECT FROM), keep in mind that the entire operation occurs in one transaction. Such an operation is efficient only if the entire amount of data fits in the cache memory of the server. If transaction size outgrows the cache size, the performance degrades significantly. Therefore, you should copy data of a large table to another table in smaller transactions (for example, few thousands of rows per transaction) using a simple stored procedure or application.

   Note: The intermediate table does not need indices. The indices should be re-created in the new table after the data has been successfully copied.
3. Drop the in-memory table.
4. Rename the disk-based table to have the original name of the dropped in-memory table.

Tip:

- You should set the **MME.ImdbMemoryLimit** to a slightly lower value than the maximum you really have available. If you run out of memory and have no unnecessary in-memory tables or indexes that you can get rid of, you can increase the **MME.ImdbMemoryLimit** slightly, restart the server with enough free memory that you can address the long-term need.
- Use the **MME.ImdbMemoryWarningPercentage** to warn you about increasing memory consumption.
- Not all situations require you to reduce the number of in-memory tables. In some cases, the most practical solution may be to simply install more memory in the computer.

Process memory consumption

Use the configuration parameters in the **Srv** section to control the maximum amount of virtual memory that can be allocated to the in-memory database process.

Srv.ProcessMemoryLimit:

The **Srv.ProcessMemoryLimit** parameter specifies the maximum amount of virtual memory that can be allocated to the in-memory database process.

The factory value for **Srv.ProcessMemoryLimit** is 0; there is no process memory limit. If you use the parameter, set it to a value that will ensure that the in-memory database process will fit entirely within physical memory. The following factors impact the amount of memory needed:

- the amount of physical memory in the computer
- the amount of memory used by the operating system
- the amount of memory used by in-memory tables (including temporary tables and transient tables) and the indexes on those in-memory tables
- the amount of memory set aside for the solidDB server's cache (the **IndexFile.CacheSize** parameter)
- the amount of memory required by the connections, transactions and statements running concurrently in the server. The more concurrent connections and active statements there are in the server, the more working memory the server requires. Typically, you should allocate at least 0.5 MB of memory for each client connection in the server.
- the memory used by other processes (programs and data) that are running in the computer

When the limit is reached, that is, when the in-memory database process uses up 100% of the memory specified by **Srv.ProcessMemoryLimit**, the server will accept ADMIN COMMANDs only. You can use the **Srv.ProcessMemoryWarningPercentage** and **Srv.ProcessMemoryLowPercentage** parameters to warn you about increasing process memory consumption.

Note:

- The **Srv.ProcessMemoryLimit** and **Srv.ProcessMemoryCheckInterval** parameters are interlinked; if the **ProcessMemoryCheckInterval** parameter is set to 0, the **ProcessMemoryLimit** parameter is not effective, that is, there is no process memory limit.
- You should not set the **Srv.ProcessMemoryLimit** parameter when using SMA. If you need to limit the memory the SMA server uses, use the **SharedMemoryAccess.MaxSharedMemorySize** parameter.

Srv.ProcessMemoryLowPercentage:

The **Srv.ProcessMemoryLowPercentage** parameter sets a warning limit for the total process size. The limit is expressed as percentage of the **Srv.ProcessMemoryLimit** parameter value.

Prior to exceeding the limit, you have exceeded the warning limit defined with the **ProcessMemoryWarningPercentage** parameter and received a warning in the `solmsg.out` log file. When the **Srv.ProcessMemoryLowPercentage** limit is exceeded, a system event is given.

The limit set with **Srv.ProcessMemoryLowPercentage** must be higher than the **Srv.ProcessMemoryWarningPercentage** limit. For example, if the **Srv.ProcessMemoryWarningPercentage** is set to 82, the **Srv.ProcessMemoryLowPercentage** value must be at least 83.

Srv.ProcessMemoryWarningPercentage:

The **Srv.ProcessMemoryWarningPercentage** parameter sets the first warning limit for the total process size. The warning limit is expressed as percentage of the **Srv.ProcessMemoryLimit** parameter value.

When the **Srv.ProcessMemoryWarningPercentage** limit is exceeded, a system event is given in the `solmsg.out` log file.

The limit set with **Srv.ProcessMemoryWarningPercentage** must be lower than the **Srv.ProcessMemoryLowPercentage** limit.

Srv.ProcessMemoryCheckInterval:

The **Srv.ProcessMemoryCheckInterval** parameter defines the interval for checking the process size limits. The interval is given in milliseconds.

The minimum non-zero value for **Srv.ProcessMemoryCheckInterval** is 1000 (ms). Only values 0, 1000, or above 1000 (1 second) are allowed. If the given value is above 0 but below 1000, an error message is given.

The factory value is 0, that is, the process size checking is disabled.

The **Srv.ProcessMemoryLimit** and **Srv.ProcessMemoryCheckInterval** parameters are interlinked; if the **ProcessMemoryCheckInterval** parameter is set to 0, the **ProcessMemoryLimit** parameter is not effective, that is, there is no process memory limit.

Appendix A. Algorithm for choosing which tables to store in memory

This section describes a strategy that will guide you in choosing which tables to put in memory.

The main principle is to consider the density of access to the table; the higher the frequency of access, the higher the access "density". Similarly, the larger the table, the lower the access density for a given number of accesses per second.

The access density is measured in units of accesses per megabyte per second, which is shown here as rows/MB/s. (For simplicity, one access per row is assumed.)

Example 1:

If you have a 1 megabyte table, and you access 300 rows in a 10-second period, the density is `30 rows/MB/s = 300 rows / 1 MB / 10 seconds`.

Example 2:

If you have a 500 KB table and you access 300 rows per second, the access density is `600 rows/MB/s = 300 rows / 0.5 MB / second`.

The table in the second example has a higher access density than the first one, and if you can only fit one of these tables into memory, you should put the second one into memory.

1. You may want to take into account the number of bytes accessed each time. This is typically the average row size, although it may be different if you are using binary large objects, or if the server can find all the information that it needs by reading just an index rather than the entire table.

   Because the server normally reads data from the disk in multiples of a "block" (where a block is typically 8 KB), the number of bytes per access or the number of bytes per row gives you only slightly more precise figures than the formula without these. Whether you read a 10-byte row or a 2000 byte row, the server does approximately the same amount of work.
2. When taking into account the size of the table, you must also take into account the size of any indexes on that table. Each time that you add an index, you add more data that is stored about that table. Furthermore, when you add a foreign key constraint to a table, the server will create an appropriate index (if one does not already exist) to speed up certain types of lookup operations on that table. When you calculate the size of your table in memory, you must take into account the table, all its indexes, and all its BLOBs.

Once you have calculated the access density of all your tables, you rank order those tables from highest to lowest. Starting with the table that has the highest density, work your way down the list, designating tables as in-memory tables until you use up all of the available physical memory.

This description is simplified as it assumes that you have perfect information and that you can change a table from disk-based to in-memory (or vice-versa) at any time. In fact, you may not know the total amount of free memory in your

computer. You might accidentally designate more in-memory tables than the computer has room in physical memory for. The result may be that tables are swapped to disk. This may substantially reduce performance. Also, you may not really know how frequently each table is accessed until that table has a substantial amount of data in it. Yet the solidDB server requires that you designate a table as in-memory or disk-based at the time that you create the table, before you have put any data into it. Thus your calculations are going to have to be based on estimates of the amount of usage each table gets, estimates of the size of each table, and estimates of the amount of free memory. It also assumes that the average access density does not change over time.

This approach also assumes that you are not planning to add still more tables in the future, and it assumes that your tables do not grow in size. In a typical situation, you should not use up all the memory that you have - you should leave enough space to take into account that your tables are likely to grow in size, and you should leave a little bit of a margin for error so that you do not run out of memory.

Important: Since virtual memory may be swapped to disk frequently, using virtual memory negates the advantage of in-memory tables. Always make sure that the entire DBMS process fits into the physical memory of the computer.

Appendix B. Calculating maximum BLOB size

B.1 Purpose

One important difference between in-memory tables and disk-based tables is that column values in in-memory tables must fit into a single "page" (the page size is specified in the `solid.ini` configuration file, and its maximum is 32 KB Therefore, in-memory tables cannot store character or binary files larger than the page size. Smaller binary files, however, are supported.

This appendix shows how to calculate the maximum size of a character or binary column value that will fit in your in-memory tables.

B.2 Background

Many applications today use data that cannot be easily stored in the standard data types such as INT or CHAR. Instead, a long character or binary format may be better suitable. In these cases, the data may be stored as *CLOBs* and *BLOBs*, *Character* and *Binary Large OBjects*, respectively. A CLOB includes interpretable characters whose number may be up to 2 billion. A BLOB data type can hold virtually any data that can be stored as a series of binary numbers (8-bit bytes). Typically, BLOBs are used to store large, variable-length data that cannot be easily interpreted as numbers or characters. For example, BLOBs may hold digitized sound (for example, the music on a Compact Disc), multimedia files, or time-series data read from sensors.

In solidDB BLOBs are widely supported and there are several different data types to choose from: BINARY, VARBINARY and LONG VARBINARY, of which the latest is mapped to standard data type BLOB.

CLOB is implemented with six data types, CHAR, WCHAR, VARCHAR, WVARCHAR, LONG VARCHAR and LONG WVARCHAR. The two latest data types are mapped to standard data types CLOB and NCLOB. For detailed information about CLOB and BLOB data types see sections *Character Data Types* and *Binary Data Types* in the *Appendix A* in the *IBM solidDB SQL Guide*.

For disk-based tables, solidDB's implementation of BLOB storage balances speed of access with the need to be able to store large amounts of data. Regardless of the data type (VARCHAR, VARBINARY), short values are generally stored in the table, while longer values have part or all of their data stored in a separate area in the database storage tree. This is entirely transparent to the user; the user simply decides on the data type, and solidDB takes care of the rest. Your data will always be accessed the same way, and will appear to be stored in the table, regardless of the actual physical location of the data. In disk-based tables, the maximum length of a VARCHAR or VARBINARY field is 2 gigabytes.

For in-memory tables, BLOB data is stored entirely in the table itself, and the maximum length of a BLOB is limited by the "block size" (no row of an in-memory table may exceed the length of a page or "block"). In this appendix, we give you some information to help you estimate the largest size VARCHAR or VARBINARY data that you can store in an in-memory table.

B.3 Calculating

The algorithm for calculating the space available for BLOBs is approximate. Make a copy of the table below, then fill it in with the values appropriate for your table. Follow the steps to calculate the remaining space available for BLOB data.

Table 5. Calculating the space available for BLOB data

	VALUE	WHAT TO ENTER IN VALUE	WHAT THE VALUE MEANS
1		In the space to the left, enter either your block size or 32767 (whichever is smaller). The block size will be either the value that you set in the [IndexFile] BlockSize `solid.ini` configuration file, or the default documented in the *IBM solidDB Administrator Guide*.	The block size (page size) is the number of bytes in a "block", analogous to a disk block. Since each row must fit within a block, this represents the maximum size of a row.
2	17	Use the hardcoded value shown to the left.	This is the number of bytes of overhead per page.
3	10	Use the hardcoded value shown to the left.	This is the number of bytes of overhead per row. We'll assume that you have only 1 row per page if you have large BLOBs.
4		If you have declared an explicit primary key for your table, enter the value 10. Otherwise, enter 20.	This represents bytes used for columns that the server automatically adds to each table.
5		Enter the number of columns in your table, multiplied by 2.	This is the number of bytes of overhead for the columns.
6		Enter the sum of the sizes of the fixed-size columns of data in your table. (See table #2 below for the size of each fixed-size data type.)	This represents space taken up by fixed-size columns.
7		Enter the number of blob columns.	This is the number of bytes used to terminate BLOB values (1 byte per value).
8		Sum the values in rows 2 through 7.	This is the total space used by everything except the BLOB values.
9		Subtract row 8 from row 1.	This is the approximate number of bytes available for BLOB data. If you have a single BLOB column in your table, then this is the approximate maximum size of that BLOB value.

Note: The maximum block size is 64K; however, the maximum row size (and thus the maximum blob size) is only 32K (actually 32K-1, or 32767). If your block size is 64K or 32K, enter 32767 instead of the block size in row 1 of the table.

The table below indicates the number of bytes required to store a value of each fixed-size data type. For example, it takes 8 bytes to store a value of type SQL FLOAT.

Table 6. Number of bytes required to store values

Data Type	**Storage Size (in bytes)**
TINYINT	1
SMALLINT	2
INT	4
BIGINT	8
DATE/TIME/TIMESTAMP	11
FLOAT / DOUBLE PRECISION	8
REAL	4
NUMERIC / DECIMAL	11
CHAR / VARCHAR / LONG VARCHAR	char_length(column_value) + 1
WCHAR / WVARCHAR / LONG WVARCHAR	char_length(column_value) * 2 + 1
BINARY / VARBINARY / LONG VARBINARY	octet_length(column_value) + 1

Appendix C. Calculating storage requirements

This appendix gives you information that helps you estimate how much memory or disk space is required to store a table and its indexes in memory or on disk.

The formulas given here are not precise, for example, because of the following reasons:

- solidDB compresses some data.
- Variable-length data (for example, VARCHAR) requires different amounts of space, depending upon the actual lengths of the values stored.
- The in-memory data structures do not necessarily store the same number of pointers for every record.

In the formulas presented here, it is assumed that the data is not compressed, and there are the maximum number of pointers. Thus the results that you get by using these formulas are conservative - that is, the formulas typically overestimate the amount of space required.

In the formulas below, the notation `sum_of(x)` means to take the sum of the sizes of each `x`. For example:

- `sum_of(col_size)` means to take the sum of the sizes of each of the columns in the table or index.
- `sum_of(index_sizes)` means to take the sum of the sizes of all of the indexes on the table.

C.1 Calculating storage requirements for disk-based tables

The general formula for the space required for a disk-based table is:

```
chkpt_factor x (table_size + sum_of(index_sizes))
where
        chkpt_factor is between 1.0 and 3.0 (explained below), and
        table_size =
            1.4 x rows x (sum_of(col_size + 1) + 12)
where
        rows is the number of rows; and
        sum_of(col_size + 1) is the sum of the sizes of the columns
            plus one byte per column.
The column sizes are shown in a table later.
```

For each disk-based index, the *index_size* is

```
1.4 x rows x (pkey_size + idx_size)
```

where *pkey_size* is the sum of the sizes of the columns in the primary key, and *idx_size* is the sum of the sizes of the columns in the index.

The *chkpt_factor* is needed to take into account that "checkpoint" operations may briefly require up to three times the size of the database. During a checkpoint operation, a copy of each of the changed pages in the database is copied from memory to the disk. If every page in the database has been updated, then it is possible to copy as many pages from memory as there already are on disk. Furthermore, the most recent successful checkpoint is not deleted until the current checkpoint is successfully completed. Therefore, during a checkpoint the disk can simultaneously have up to 3 copies of each page (1 copy for the page in the

database, 1 copy in the most recent successful checkpoint, and 1 copy for the current checkpoint while it is executing). The checkpoint factor therefore can be between 1.0 and 3.0. Values approaching 3.0 are rare in most databases. A value of 1.5 is usually well sufficient even for small databases that have high levels of activity. The less frequent the checkpoint, the larger the *chkpt_factor* might need to be.

Note: In a disk-based index, if you do not explicitly define a primary key, the server uses a server-generated "row number" as the primary key. This forces the primary key index to store records in the same order that they were inserted.

Background information

On disk-based tables, data and indexes are stored in a B-tree. Each entry in the tree consumes space for the header and the data.

The space used by the actual data can be calculated using the column sizes per column type as shown in C.3, “Column sizes against column type,” on page 37.

In addition, in disk-based tables, the server requires 1 additional byte per column; this byte is used as part of the length indicator, which also serves as a null indicator.

The header for each row uses 12 bytes:

Table 7. Header bytes

Number of bytes	Used for...
3 bytes	Row header
3 bytes	Table id
6 bytes	Row version

If a disk-based table contains indexes other than the primary key, the size of the entries in those indexes must be estimated separately using the same guideline. An index entry contains the following components:

- Columns that are defined in the index
- Columns of the primary key of the table
- A row header (12 bytes)

Additionally, there is usually some empty space (for example, 20 - 40%) in the database pages. This is why the formulas include a multiplier of 1.4 for both tables and indexes.

For example:

You first create a disk-based table as follows:

```
CREATE TABLE subscriber (
             id INTEGER NOT NULL PRIMARY KEY,
             name VARCHAR(50),
             salary FLOAT)
             STORE DISK;
```

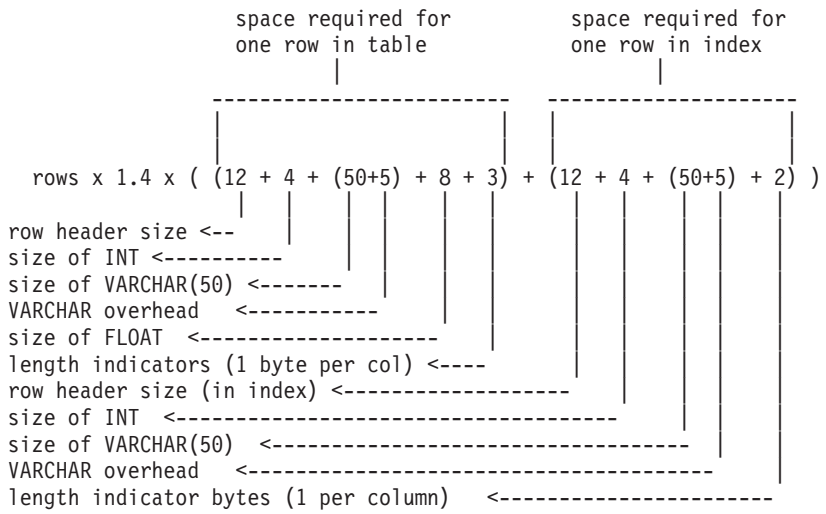
Then you create a secondary index as follows:

```
CREATE INDEX subscriber_idx_name ON subscriber (name);
```

The index entry contains the NAME column; it also contains the primary key column, which in this case is ID. The space required by that index should be estimated separately. The total size of the disk-based table, assuming the "empty space factor" is 1.4, can be calculated as follows:

```
rows x 1.4                  // 1.4 = the empty space estimate.
   x ( (12 + 4 + (50+5) + 8 + 3)   // size of the table entry,
       + (12 + 4 + (50+5) + 2) )   // size of the secondary index entry
```

Tip: You can present the above calculation also in the following way:



C.2 Calculating storage requirements for in-memory tables

The general formula for the space required for an in-memory table is:

table_size + sum_of(*index_sizes*)

table_size =

1.3 x rows x (sum_of(*col_sizes*) + (3 x *word_size*) + (2 * *num_cols*) + 2)

where:

- *rows* is the number of rows
- *word_size* is the machine word size (for example, 4 bytes for 32-bit OS and 8 bytes for 64-bit OS)
- *num_cols* is the number of columns
- sum_of(*col_sizes*) is the sum of the sizes of the columns

For each in-memory index, the index size is

1.3 x rows x ((*dist_factor* x sum_of(*col_sizes* + 1)) + (8 x *word_size*) + 4)

where *dist_factor* is a value between 1.0 and 2.0 that depends upon the distribution of the key values. If key values are highly dissimilar, use a value closer to 2.0. If key values are highly similar, use a value closer to 1.0.

Background information

When calculating the storage requirements of in-memory tables, the size of each entry is the combined size of the data of the table plus three memory pointers (4 bytes each in 32-bit operating systems or 8 bytes each in 64-bit operating systems)

of overhead per row. Additionally, there is overhead of two bytes per row and two bytes per each column of the row. You do not need to add 1 byte per column to take into account the length indicator; that is included in the 2 bytes per row.

In addition, the in-memory tables can have indexes, which are populated upon server startup. Each index entry contains the data of the columns defined in the index. Additionally, each index entry contains up to eight memory pointers. A copy of the primary key is NOT required for an in-memory index.

Furthermore, there is some other overhead that depends on the actual data values of the index. This is a percentage of the data size of the index. An exact value cannot be given exactly because it depends on the key value distribution, but the multiplier ranges between 1.0 and 2.0.

Additionally, the index structure itself needs an average of 4 bytes per index entry (that is, per row).

For the above example table and index

For example:

You first create an in-memory table as follows:

```
CREATE TABLE subscriber (
             id INTEGER NOT NULL PRIMARY KEY,
             name VARCHAR(50),
             salary FLOAT)
             STORE MEMORY;
```

Then you create a secondary index as follows:

```
CREATE INDEX subscriber_idx_name ON subscriber (name);
```

You can then estimate the memory consumption in a 32-bit operating system as follows:

```
rows * 1.3 x (
  ((3 x 4) + 2 + (4 + 2) + (50+5+2) + (8+2)) // Size of data in table
+ ((8 x 4) + 4 + 1.2 x 4)       // Size of the primary key index
+ ((8 x 4) + 4 + 1.2 x (50+5)))  // Size of the secondary index.
)


((3 x 4) + 2 + (4 + 2)+(50+5+2)+(8+2)) // Size of data in the table.
   |   |    |    |   |   | | |   | |
   |   |    |    |   |   | | |   |  -> 2 bytes per col
   |   |    |    |   |   | | |    ---> 8 bytes for FLOAT (Salary)
   |   |    |    |   |   | |  -------> 2 bytes per col
   |   |    |    |   |   |   ---------> 5 bytes overhead per VARCHAR
   |   |    |    |   |    ------------> 50 bytes for VARCHAR(50)
   |   |    |    |    ----------------> 2 bytes per col
   |   |    |     --------------------> 4 bytes for INT (ID)
   |   |     ------------------------> 2 bytes per row
   |    ----------------------------> pointer size (4 bytes on 32-bit OS)
    ---------------------------------> 3 pointers

((8 x 4) + 4 + 1.2 x 4)     // Size of the primary key index
   |   |    |    |   |
   |   |    |    |    ---------------> 4 bytes for INT
   |   |    |     --------------------> 1.2 index value distribution factor
   |   |     ------------------------> 4 bytes per index entry
   |    ----------------------------> pointer size (4 bytes on 32-bit OS)
    ---------------------------------> 8: up to 8 pointers
```

```
((8 x 4) + 4 + 1.2 x (50+5))  // Size of the secondary index.
  |   |    |    |     |  |
  |   |    |    |     |   -----------> 4 bytes for VARCHAR overhead
  |   |    |    |      --------------> 50 bytes for VARCHAR(50)
  |   |    |     --------------------> 1.2 index value distribution factor
  |   |     -------------------------> 4 bytes per index entry
  |    ----------------------------> pointer size (4 bytes on 32-bit OS)
   ---------------------------------> 8 up to 8 pointers
```

In a 64-bit operating system, use a memory pointer size of 8 bytes rather than 4 bytes.

The factor 1.2 in the above estimate is the "TRIE index value distribution factor" whose exact value depends on the actual values of the indexed column. Its value is typically between 1 and 2. With random value distribution, the value is closer to 2.0. With sequential value distribution, it is closer to 1.0. The 4 bytes is the data overhead needed by an index entry on average.

The factor of 1.3 is to take into account the internal overhead of the memory allocator.

Note: Indexes of main memory tables are created dynamically each time the server starts. The in-memory indexes are never written to disk and therefore they do not occupy any disk space. However, the tables themselves are written to disk during checkpoints and when the server shuts down, so the total amount of disk space that you have must be enough to store both the disk-based tables and the in-memory tables.

C.3 Column sizes against column type

Table 8. Column sizes against column type

Column type	Size
TINYINT	2 bytes
SMALLINT	2 bytes
INT	4 bytes
BIGINT	8 bytes
DATE/TIME/ TIMESTAMP	11 bytes
FLOAT / DOUBLE PRECISION	8 bytes
REAL	4 bytes
NUMERIC / DECIMAL	12 bytes
CHAR / VARCHAR / LONG VARCHAR	char_length(column_value) + 5
WCHAR / WVARCHAR / LONG WVARCHAR	char_length(column_value) * 2 + 5
BINARY / VARBINARY / LONG VARBINARY	octet_length(column_value) + 5

Note: The values in the above table are the maximum lengths. Variable-length data (VARCHAR) or compressible data might require fewer bytes.

C.4 Measuring memory consumption

After you have created your tables and indexes, you can measure the actual amount of memory consumed by using the command:

```
ADMIN COMMAND 'info imdbsize';
```

This command gives the total memory consumption of in-memory tables and indexes. The units are kilobytes.

Index

Special characters

A

B

C

D

E

H

I

L

M

N

P

R

S

T

Notices

This product is protected by U.S. patents 6144941, 7136912, 6970876, 7139775, 6978396, 7266702, 7406489, 7502796, and 7587429.

This product is assigned the U.S. Export Control Classification Number ECCN=5D992b.

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: 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. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Canada Limited
Office of the Lab Director
8200 Warden Avenue
Markham, Ontario
L6G 1C7
CANADA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, ibm.com®, Solid, solidDB, InfoSphere®, DB2®, Informix®, and WebSphere® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other product and service names might be trademarks of IBM or other companies.

SC27-3845-05