IBM PureData for Operational Analytics

Best practices
Transforming IBM Industry Models into a production data warehouse

Austin Clifford
DB2 Warehouse QA specialist
IBM Dublin Lab

David Murphy
DB2 Development lead
IBM Dublin Lab

Garrett Fitzsimons
Best practices specialist for warehouse & appliances
IBM Dublin Lab

Pat Meehan
Senior IT Specialist
IBM Dublin Lab

Rónán O'Suilleabhain
Integration & deployment specialist, Industry Models
IBM Dublin Lab

Sami Abed
DB2 Kernel developer
IBM Dublin Lab

Issued: October 2012

Executive summary

Implementing an industry model can help accelerate projects in a wide variety of industry sectors by reducing the effort required to create a database design optimized for data warehousing and business intelligence.

IBM Industry Models cover a range of industries which include banking, healthcare, retail, and telecommunications. The example that is chosen in this document is from a subset of the IBM Insurance Information Warehouse model pertaining to Solvency II (SII) regulations.

Many of the most important partitioned database design decisions are dependent on the queries that are generated by reporting and analytical applications. This paper explains how to translate reporting needs into database design decisions.

This paper guides you through the following recommended process for transforming a logical data model for dimensional warehousing into a physical database design for production use in your environment. The key phases of this process are:

- Create a subset of the data model subset from the supplied logical data model
- Prepare the physical data model for deployment as a partitioned DB2 database
- Refine the physical data model to reflect your reporting and analytics needs
- Optimize the database architecture and design for a production environment

Implement a database architecture that is aligned with best practices for warehousing before tuning your database design to reflect performance needs for reports and queries. When you have created and populated the test database can you further optimize the database design to reflect the anticipated query, ingest and maintenance workload.

A poorly designed database and architecture can lead to poor query performance and a need for outages to accommodate maintenance operations. Using the recommendations in this paper can help you transform an IBM Industry Model dimensional warehouse solution into a partitioned database that is ready for production use.

Introduction

IBM Industry Models provide you with an extensive and extensible data model for your industry sector. Use the logical data model as provided by IBM to build a physical model that is customized for your reporting requirements then deploy and populate a best-practice partitioned-database production environment.

This paper does not discuss data modeling concepts but instead focuses on what you must do to transform a non-vendor-specific logical data model into a best-practice production DB2 partitioned database.

This paper is targeted at people involved in transforming the dimensional data warehouse logical data model into a production partitioned database that is based on DB2® Database for Linux®, UNIX®, and Windows® software v10.1.

The goal of the IBM Insurance Information Warehouse model in addressing Solvency II is to facilitate reporting in line with European Union directives and internal business requirements. A subset of tables from the dimensional layer, together with a sample SII-based Cognos report, is referenced throughout the paper. The test environment used is described in appendix A.

The first section of this paper looks at the deployment pattern and the main design challenges that you must address when you implement an industry model. The process of identifying and manipulating the components of the industry model that are relevant to your business is outlined.

The second section of the paper looks at transforming the logical data model into a physical data model that is aligned with best practices for a partitioned DB2 database.

The third section of the paper shows how to translate reporting requirements into database design choices that help shape the physical data model.

The fourth section of the paper describes how to optimize the database design to reflect the specific needs of your production environment. The temporal feature and continuous data ingest utility, both introduced in DB2 Version 10.1, are described in the context of how they might influence your database design decisions.

The IBM PureData for Operational Analytics System implements best practices for data warehouse architecture that uses DB2 software. The shared-nothing architecture of the IBM PureData for Operational Analytics System provides a platform that emphasizes performance, scalability, and balance. The paper "Best Practices: Physical database design for data warehouse environments", referenced in the *Further reading* section, covers the recommendations for data warehouse database design in detail.

Using industry models

An industry model is a comprehensive set of predesigned models that form the basis of a business and software solution. An industry models solution consists of a set of industry-specific integrated models that are optimized for business challenges in a particular sector. Domain areas include data warehousing, business intelligence, business process management, service-oriented architecture, business terminology, and business glossary templates.

Solvency II (SII), a European Union directive on insurance regulation, and the associated Pillar 3 reporting requirements are extensively covered by the IBM Insurance Information Warehouse (IIW) data models. This paper uses the SII coverage within the IIW model to show how best practices in data warehousing for DB2 can be applied to an industry model.

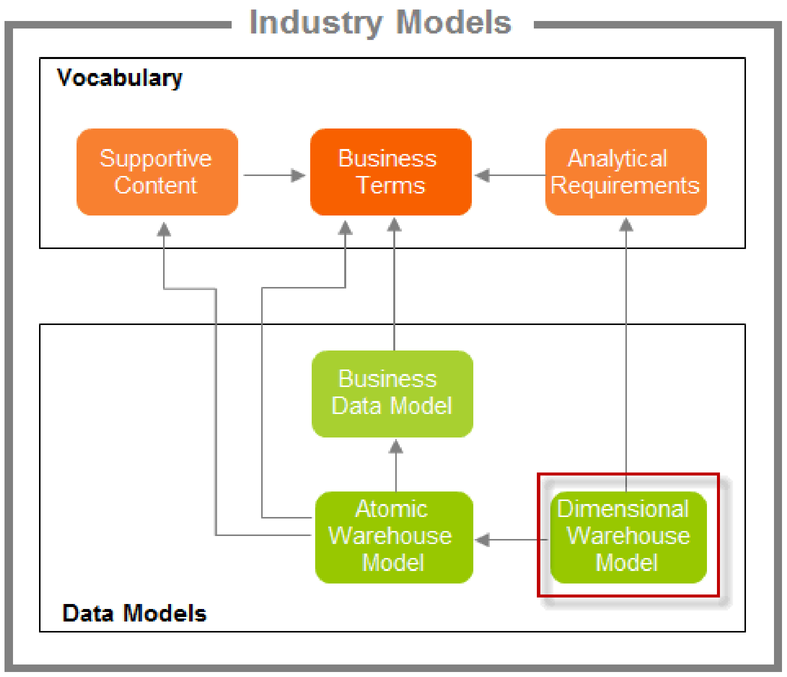


Figure 1 The dimensional warehouse model within the context of industry models

To choose the entities or scope of the logical data model that are relevant to your business, first determine your specific reporting requirements.

Implementing an industry model as a physical database

Implementing a logical data model as a physical database presents technical challenges. There are several steps through which you create and optimize your physical database for production use.

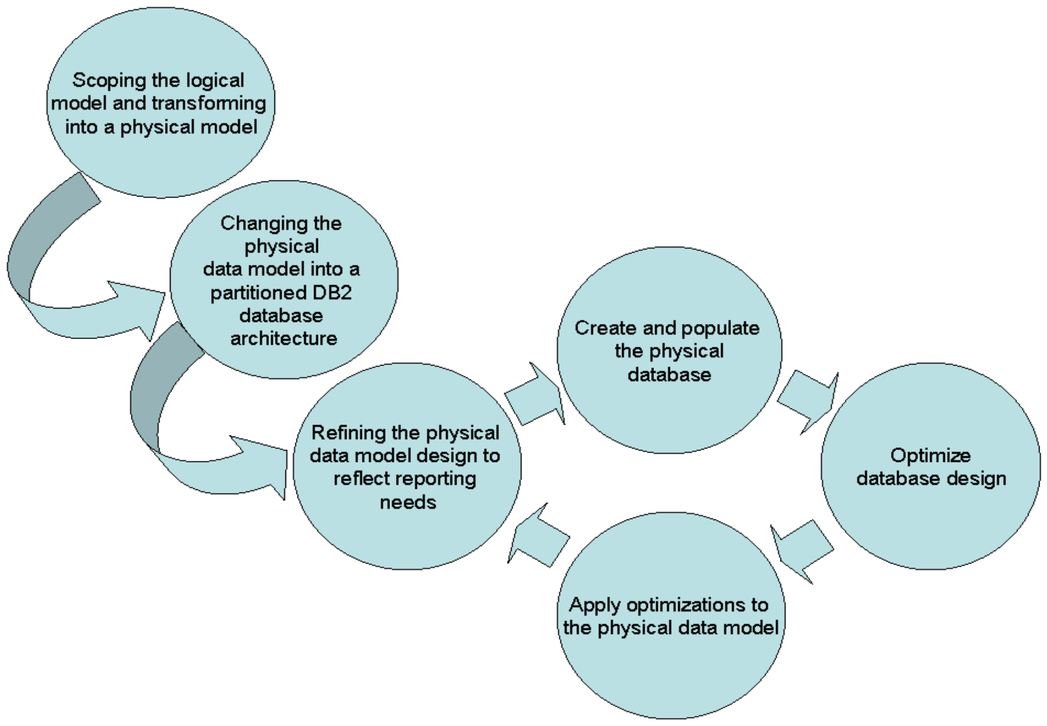

Scoping the logical model and transforming into a physical model
Changing the physical data model into a partitioned DB2 database architecture
Refining the physical data model design to reflect reporting needs
Create and populate the physical database
Optimize database design
Apply optimizations to the physical data model

Figure 2 Typical deployment patterns for creating a physical database

The phases that are involved in this process include:

- Scoping the logical model and transforming it into a physical model

  Map reporting requirements to the logical data model to determine the scope of the model. Include only those entities and attributes for which you have a reporting requirement.

- Preparing the physical data model for deployment as a partitioned DB2 database

  Update the physical data model to be compatible with the target partitioned database architecture.

- Refining the physical data model to reflect your individual reporting needs

  Use the details of individual report specifications to help you further define the physical database model.

- Optimizing the database architecture and design for a production environment

  Create and populate the physical database to determine final database design optimizations that are based on actual query and maintenance workloads that must be applied back into the physical data model.

Understanding database design challenges

The logical data model as provided contains no vendor specific database features. You must implement the features of your database software in the physical data model.

Focus your data warehouse design decisions on the following elements:

- **Query performance** – efficient query performance minimizes resource usage.

  Use database partitioning, materialized query tables (MQT), multidimensional clustering (MDC), and table range partitioning (RP) to help maximize query performance across all database partitions.

- **Intelligent table space design**– the ability to manage, move, and archive data as it ages.

  Intelligent table space design facilitates data archiving, support for multi-temperature storage, flexibility in backup and recovery strategies.

- **Data ingest** - ingest data with minimal effect on data availability.

  Implement an architecture in which data ingest and data backup can operate concurrently.

- **Online maintenance operations** – architect and design for concurrent database operations.

  Reduce the number of operations that are needed to maintain data availability and query performance and enable online database operations.

Focus on performance, scalability, and balance when you design your database environment.

The following information must be available or determined before you can begin to optimize your database design:

- The volume of data predicted for each fact and dimension table initially and over the lifecycle of the database for each table.

  Know which fact and dimension tables are largest to finalize your distribution keys.

- Data lifecycle and multi-temperature requirements.

  The length of time data must be retained within the database. Maintaining data for queries and for regulatory compliance influences how you might partition your tables or configure storage groups.

- Sample queries from reports or analytics applications that reflect anticipated or real queries to be submitted in production.

  This method helps to determine MDC, MQT, indexing, and distribution key selection requirements.

- Stated objectives for backup and restore operations.

  Aggressive backup and restore time objectives can determine that certain tables are placed in separate table spaces.

Scoping the logical model and transforming into a physical data model

Scoping the industry model is the process of selecting the business objects that you need from the logical data model to build a valid physical data model. The physical data model must reflect your analytical and reporting needs.

As a pre-requisite to the scoping phase, you must model your business requirements and map these to analytical requirements. The quality and availability of your data sources must also be understood and assessed. However, these tasks are outside the scope of this document which focuses on the implementation of a production physical database.

When you complete the process of scoping, you can transform your logical data model into a physical data model by selecting a menu option in IBM InfoSphere® Data Architect.

IBM InfoSphere Data Architect is a data modeling tool that you can used to scope, transform, and customize the data models that are supplied in the Industry Models solutions. The examples in this paper reference InfoSphere Data Architect and you can reference the *Further reading* section for more details about the product.

Scoping the logical model

The logical data model is designed to meet all aspects of reporting for an industry sector. Your enterprise might not need all of the objects that are provided. Scoping is the process, by using InfoSphere Data Architect or other modeling tools, of selecting those entities from the logical data model that align with your analytical requirements.

Refine your logical data model scope to address only your current data and reporting needs. This targets just those tables accessed by ETL, queries and maintenance operations.

When you scope the industry model to create your logical data model, use these steps with InfoSphere Data Architect:

- Create a diagram into which you can drag those entities that you need to address your warehousing and reporting needs.

  Creating a diagram for your logical data model avoids directly changing the base model. This method allows you to more easily accept future industry model upgrades.

- Navigate through the Aggregate Facts section and drag the aggregate facts that you need into your new diagram.

  Aggregate facts are related to the supporting fact tables which can be identified and included when you are completing the scoping process. Supporting entities can be identified under the heading “DWM Source”.

- Use InfoSphere Data Architect to identify and include all related entities (fact and dimensions tables) in the diagram.

  Avoid manually moving individual related entities because this can affect the integrity of the resulting database. Let InfoSphere Data Architect identify and automatically add all related entities; you can then prune those entitles that you do not need.

Transforming the logical data model into a physical data model

Since the logical data model applies to all databases, minimize the database architecture and design changes that you make to the logical data model. Instead, implement those changes in the physical data model. This strategy provides the following benefits:

- Easier upgrade strategy for future releases of industry models as only semantic differences will exist between your model and the industry model.
- Focus technical modeling effort on the physical data model and retain a logical data model that is suited for all databases.
- More easily control changes, in both the logical and physical data models, by assigning clear roles to each model. The logical model functions as a semantic master while the physical model is the technical master.

Entities can be added to the diagram at a later stage and merged into an existing physical data model by using the compare and merge functionality in InfoSphere Data Architect. Select this approach to build your physical data model incrementally over time.

Apply architecture and database design changes that are specific to DB2 databases to the physical database model rather than the logical data model to help accommodate future upgrades.

Transform your logical data model into a physical data model by selecting a blank area in the diagram and, from the InfoSphere Data Architect main menu, selecting **Data > Transform > Physical Data Model.**

When prompted for further details, select the DB2 database version that you require, for example DB2 V10.1. Use the default settings that are provided and complete the transformation process. Validate your DB2 installation and the physical data model by generating the DDL to create a test database.

Preparing the physical data model for deployment as a partitioned database

The physical data model is a representation of your logical data model that is compatible with DB2. However, the model does not yet reflect a partitioned database environment or specifically, a data warehouse architecture and design.

Several best practice recommendations for data warehousing can be applied to the physical data model before you generate DDL that is suitable for a partitioned database environment. These improvements include the following items:

- Introducing database partition groups
- Implementing a table space strategy
- Customizing data types
- Implementing surrogate keys

Refer to the *Further reading* section and the best practices paper called “*Physical database design for data warehouse environments*” for detailed explanations and examples.

Starting with database partition groups

In a partitioned database, database partition groups determine which table spaces and, by effect, which table and index objects are partitioned or nonpartitioned.

Minimize the number of database partition groups and avoid overlapping database partition groups on the same data host to avoid adding complexity to resource allocation and monitoring.

Adhere to the following defaults implemented in an IBM PureData for Operational Analytics System build when you are creating and using database partition groups:

- Create just two new database partition groups; one for tables you want to partition and one for tables you do not want to partition.

  There are no performance gains to be made from having multiple database partition groups. Collocated queries are not supported where tables are in different database partition groups.

- Avoid overlapping database partition groups.

  Create the nonpartitioned database partition group on the coordinator database partition and create the partitioned database partition group across each data host.

- Decide which tables you want to partition and which tables do not contain enough rows to be partitioned. Place these tables in the appropriate database partition group.

For example, a dimension table such as `Country Of Origin` would not contain enough rows to be considered for partitioning. The overhead of partitioning a small table would exceed any performance benefit.

Implementing a table space strategy

Intelligent table space design, with table partitioning, unlocks many features available in the DB2 software:

- Balanced table space size and growth enables more efficient backup performance and a more flexible backup and recovery strategy.

  For example, in a recovery scenario you can focus on recovering table data. You can opt to rebuild indexes, refresh MQTs, replicate tables, and restore older table data as separate operations.

- Table space maintenance operations can be targeted at active data rather than entire tables which include active and inactive data.

  For example, operations such as REORG can be targeted at just those table spaces that contain active data.

- Data lifecycle operations can take place online.

  For example, when data partitions are detached (rolled-out), the dedicated table space can be removed and the space can be reclaimed immediately.

- Multi-temperature database strategy is possible as individual table spaces can be moved from one storage layer to another as an online operation.

  For example, when table partitioning is implemented, inactive data can be moved as an online operation to less expensive storage, releasing storage capacity for more active data to be placed on.

Intelligent table space design facilitates concurrent database maintenance operations and helps avoid costly reorganization tasks in production

Correcting a poor table space design strategy post production can have a negative effect on resource usage and data availability:

- Significant resources are needed to physically move data from one table space to another post production.
- Having too few table spaces restricts your flexibility in performing data-specific backup and restore operations, performing maintenance operations on specific ranges of data or tables, and managing the data lifecycle.
- Having too many table spaces creates unnecessary overhead to the database manager when you activate the database and maintain recovery history. It can also require too many database operations to be issued in parallel to complete tasks.

Table 1 describes how to design a good table space design strategy that gives you the flexibility you need to meet your service level objectives for all workloads.

Table type	**Table space strategy**
Largest fact table, largest associated dimension table, mission critical tables	Create a separate table space for each table and for each data partition. Create a separate table space for indexes in line with each table and data partition. This process enables table level recovery from a backup.
Medium sized partitioned tables	Logically group medium sized dimension tables that are part of the same fact table star or snowflake schema; a group of five tables is adequate. Create a separate table space for this group of tables and a separate table space for the associated indexes.
Small sized partitioned tables	Group all small-sized dimension tables. Create a separate table space for the group and a separate table space for the indexes that are associated with these tables.
Partitioned materialized query tables	Create a separate table space for MQTs and a separate table space for indexes on MQTs. Use this method to determine a separate backup and recovery strategy for the aggregation layer
Replicated tables	Create a separate table space for replicated tables and a separate table space for indexes.
Temporal history tables	Create a separate table space to hold temporal history tables.
Nonpartitioned tables	Create a separate table space for data and for indexes in the nonpartitioned database partition group. This separates user data from catalog data on the catalog database partition.
Staging tables	Create a separate table space for staging tables and for other non-production tables.

Table 1 Guidelines for implementing a table space strategy

Assigning tables to table spaces

The process of placing tables into table spaces is made easier by adhering to the table space strategy in table 1. Assign each table by type to the appropriate table space strategy.

As a general guideline, replicate all non-collocated dimension tables but where non-collocated dimension tables are over 10m rows then consider partitioning these tables to avoid long refresh times.

Create partitioned tables as hash partitioned tables and assign them to the partitioned database partition group.

Place table data and index data in separate table spaces to provide more flexibility when you are designing your operational maintenance and data lifecycle strategy

Specific distribution keys can be assigned later during optimization but must always be identified and assigned before production use. Changing the distribution key requires you to drop and re-create the table.

Customizing keys and data types

The physical data model, when generated, uses default keys, constraints, and data type values that you need to modify based on your source data and your approach to data ingest. Consider the recommendations in the following areas:

Primary keys

The logical data model implements a composite primary key on fact tables. The primary key includes all dimension foreign keys that make the primary key unique.

Remove the primary key from the fact tables on the physical data model. The fact tables in the logical data model have many dimension keys and these keys can affect data ingest performance negatively. Replace these keys with MDC and the non-unique composite indexing strategy that is proposed in this paper.

Referential constraints

Since the logical data model is suitable for any database, referential constraints are created by default as enforced constraints. Enforced constraints can increase the effect on resources when ingesting data and this can result in slower ingest speeds and reduced query performance.

In a warehousing environment, change these constraints to informational constraints because informational constraints can be used by the DB2 optimizer when compiling access plans and this use helps improve query performance.

Use informational constraints instead of enforced constraints to minimize the effect of unique index maintenance when you are populating fact tables.

Identity keys and surrogate keys

The logical data model implements identity keys for the primary key, whose purpose is as a surrogate key, on each dimension table.

For dimension entities, the logical data model defines certain attributes as primary and surrogate keys.

Within the physical data model, use `GENERATED BY DEFAULT AS IDENTITY` for identity columns. This allows the ingest utility to supply its own values for the surrogate keys, if required, in the input data rather than letting DB2 generate them. If input values are not supplied, then DB2 generates them.

Supplying its own values for such columns can facilitate data ingest in creating parent-child relationships, since the key of the parent is known and already supplied.

The tools that are used to ingest data into the data warehouse generally influence where the surrogate keys get assigned during the ingest process.

- When you are using an engine-based ETL tool such as IBM DataStage®, you can generate surrogate key values within the ETL engine or by using sequences.
- When you are using the ingest utility with dimension tables, use identity keys to determine your surrogate key and omit the identity key column from the ingest column list.

Refine data types

The physical data model, when generated from your logical data model, uses default data-type settings for integer and character columns. Use the following guidelines to refine the default values but avoid overpruning column lengths. Modifying these values in production can incur the need to reorganize data which can be costly.

- Change the default `CHAR`(x) data type to `CHAR`(18) for those columns you anticipate to be no more than 18 characters long.
- Change the default `CHAR`(x) data type to `VARCHAR`(y) for those columns you anticipate to be greater than 18 characters long.
- Ensure that the modified data types of columns that are used in table joins match for optimal query access plans.
- Use the `BIGINT` data type where you expect the values in the columns to be greater than the capacity of the integer data type.
- Use the `DATE` data type for the primary key of the date and time dimension. This use enables you to implement a table partitioning strategy that is based on calendar date. In addition, the DB2 optimizer can eliminate entire table partitions (ranges of data within a table) based on a date predicate and this elimination can help improve query performance.

Use the `DATE` data type for the primary key on the date and time dimension table. This method enables data partitions to be date-based and facilitates query performance.

Using report specifications to influence your physical data model

The characteristics of your database design can be further improved by analyzing the physical data model from the perspective of your analytical and reporting requirements.

Review your report specifications with a view to identifying and contrasting what you see with the details of the underlying dimensional database. Consider how even distribution of data, collocated queries, and parallelism of the query workload across the partitioned database can be achieved.

- Contrast the granularity of the report with the granularity of data in your database.

  For example, if the granularity of the database fact table is 1 or more transactions per day, and the granularity of the report compares daily totals, then the data in the fact table is a candidate for aggregation before it is presented to the report.

- Look at what dimensions are used to aggregate data on your report.

  For example, data might be aggregated by country, or by credit rating. Examining these dimensions can help you identify suitable candidate columns to be used as distribution keys. Parameters that are used frequently in reports can affect the degree of parallelism in queries and are therefore unsuitable as distribution keys.

- Examine the report parameters and filters and the order in which they are used.

  Report parameters can be viewed as dimension filters. For example, a frequently used report that contains few parameters can help determine both your MDC columns for the fact tables and potential inclusion in any MQTs.

An excerpt from the Solvency II physical data model

Figure 3 below shows an excerpt from the SII data model. The center of the diagram shows the key fact tables and the rest of the diagram shows the dimension tables.

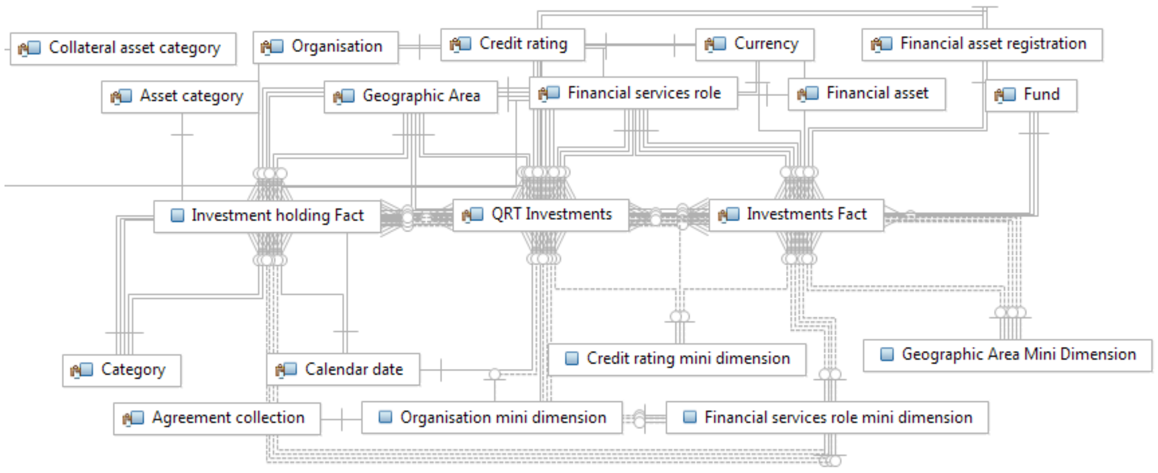


Figure 3 Excerpt from data model showing fact and dimension entities

The SII data model presents data at the granular level of month; there is a single row for an asset or investment for each given month. From a dimensional perspective, this means that the fact tables are periodic snapshot tables, representing a position at the end of the month.

Table 2 lists the fact tables and the largest dimension table, shown in figure 3, associated with each fact table.

Object type	**Description**
SLVC_INV_HLDG_FCT	Transaction fact table contains “investment holding fact” data at the granularity of one row per month.
AGRM_COLL	“Agreement collection” is the largest dimension table that is associated with the investment holding fact table.
SLVC_AST_VAL_FCT	Transaction fact table holds “asset valuation” at the granularity of one row per asset per month.
FNC_AST	“Financial asset” is the largest dimension table that is associated with the asset valuation fact table.
QRT_INV	Aggregate transaction table “QRT Investments” is a union of the two fact tables.
TM_DIMENSION	Time dimension.
SLVC_RPT_AST_CGY_ID	Solvency report “Asset Category” dimension table.
CR_RTG_ID	“Credit Rating” dimension table.
CNTRY_OF_CUSTODY_OF_AST_ID	“Country of Custody” dimension table.
ISSUR_OF_INV_ID	“Issuer of Investment” counterparty dimension table.

Table 2 List of Solvency II data model tables referenced in this paper

Interpreting Solvency II example reports as database design decisions

The example reports referenced in this paper present information about the holding of assets and investments for the month or series of months specified. The time periods that are compared could include the same assets, or have some assets that are removed, or have new assets that are introduced which triggers changes to asset prices, ratings, and so on.

The two example report requirements that are used to refine the physical data model and database design are:

- **Parent report:** Assets by region, by counterparty, by credit rating
- **Child report:** Asset drill through by dimension

Parent report: Assets by region by counterparty by credit rating report

The business question answered by this report is; “How exposed am I geographically by issuers of bonds in different parts of the world and what their credit rating is?”

The base query for this report is a `UNION` of both the Assets and the Investments fact tables. The report aggregates the data by region (country of origin,) by counterparty, by credit rating, and provides filters on these and other dimensions.

The following design points can be determined from this report:

- Distribution keys

  Use the report specification to help in choosing candidate distribution keys. Using the filters in this report could lead to queries that limit the number database partitions used in parallel to perform the query.

- Table partitioning

  Table partitioning by date is the most effective method for backup, restore, aging, and archiving. Since the data and report both use month, this is the most suitable option to use for partitioning the fact table.

- Materialized query tables (MQTs)

  This is a summary report and an ideal candidate for an MQT. The data model provides a table, `QRT_INV` (quantitative reporting templates) as a template for reports against these tables. It is recommended that you replace the provided `QRT_INV` table with two separate MQTs, one for each side of the `UNION` clause.

Child report: Asset drill-through by dimension report

This report is effectively a drill-through report from within the previous report. The business problem that is addressed by this report is; “From within the first report, I want to see the composition of the aggregated metrics for an entry on the first report”.

This report is more granular than the previous report and allows you to interrogate the data set right down to the granularity of the fact table and analyze data to the level of financial asset. The report has parameters that allow data to be filtered on individual region, counterparty, or credit rating. The following design points can be determined from this report:

- Distribution

  Since this report involves a query which joins the “Financial asset valuations” fact to the large “Financial Asset” dimension, collocation is an important design goal. The primary key for “Financial asset” and “Investment holding” dimensions could be an ideal distribution key to help collocate the fact with its largest dimension and help ensure an even distribution of data.

- Multidimensional clustering (MDC)

  The most commonly used dimension filter columns are good candidates for defining the MDC columns for the fact tables.

Optimizing the database architecture and design for your environment

Optimizing a database is an iterative process and you must ensure that sufficient test data that reflects the target production environment is available.

When you are optimizing your database design, it is critical that you qualify each optimization by ensuring that the intended change is performing as expected in isolation and in parallel with other expected workloads. The examples in this section include references to how you can use the explain plan tools to determine whether your optimizations succeed.

When you implement and successfully test changes to the database, apply the changes back into the physical data model.

This section looks at how to optimize the database that supports the reports described. The main design decisions are based on:

- Choosing distribution keys for partitioned tables
- Choosing MDC tables and MDC columns
- Choosing tables to be range partitioned
- Indexing for performance
- Identifying candidates for MQTs

Choosing distribution keys

Since the report does not filter by individual "Financial asset", the primary key of the "Financial assets" dimension table (`FNC_AST.FNC_AST_ID`) would be an ideal distribution key for the "Asset valuation" fact table to help ensure an even distribution of data across each database partition and promote parallelism in the query workload. The aggregation of data effectively removes the detail of individual assets on which the table is based.

For example, the column `FNC_AST_ID` makes sense as a distribution key for both the "Financial assets" (`FNC_AST`) and "Asset value" (`SLVC_AST_VAL_FCT`) because the following criteria are fulfilled:

- The distribution key is not used as a filter condition in the reports. This choice prevents a query from requesting data on a single database partition only and artificially limiting performance.
- Collocated queries for the target reports are achieved by partitioning both the fact and the chosen dimension on the primary key of the dimension table.
- Facilitate an even distribution of data by using the most granular dimension or the dimension that is closest to the granularity of the transaction table.

Design your database to support collocated queries within the constraint of having no more than 10% skew in the distribution of data across the partitioned database

In order to achieve an even distribution over each database partition, the distribution key must contain a relatively high number of distinct values (that is, cardinality) and the fact table must have an even distribution of data for the chosen distribution key. These choices help avoid an uneven distribution or skew which occurs when the database partition with the greatest number of rows has 10% more rows than the average row count across all database partitions.

Choosing MDC tables and columns

All fact tables are candidates for multidimensional clustering since the advantages gained in query performance and the reduction in maintenance operations are significant. Choosing columns that are not used in the filters of your most frequent queries can have a negative effect on query performance. Changing your MDC columns requires a rebuild of the table so choose and test your MDC strategy in line with report development before you introduce the strategy into production.

Use multidimensional clustering tables to organize data in all fact tables to reduce the need for regular indexes and associated maintenance operations

An MDC table physically groups data pages that are based on the values for one or more specified dimension columns. Effective use of MDC can significantly improve query performance because queries access only those pages that have rows with the correct dimension values.

Consider the following tasks when you are choosing MDC columns:

- Create MDC tables on the columns that have low cardinality in order to have enough rows to fill an entire cell.
- Create MDC tables that have an average of five cells per unique combination of values in the clustering keys.
- Use generated columns to “coarsify” or reduce the number of distinct values for MDC dimensions in tables that do not have column candidates with suitable low cardinality.

Avoid including all filter dimensions in the MDC `ORGANIZE BY` clause. Although this practice increases the flexibility of reporting, the number of cells can also be increased, resulting in sparsely populated MDC tables.

To determine whether the chosen columns are suitable as dimension columns in the fact table, use the following query to calculate the density of the cells columns that are based on an average unique cell count per unique value combination greater than 5. Table statistics must be current to return meaningful data.

```
SELECT CASE
WHEN (a.npages/extentsize)/
(SELECT COUNT(1) as num_distinct_values
FROM (SELECT 1 FROM QRT_DWM.SLVC_AST_VAL_FCT GROUP BY
DIM_SLVC_RPT_AST_CGY_ID, DIM_CR_RTG_ID)) > 5
THEN
'These columns are good candidates for dimension columns'
ELSE
'Try other columns for MDC' END
FROM syscat.tables a,syscat.datapartitions d,syscat.tablespaces b
WHERE a.TABNAME='SLVC_AST_VAL_FCT' AND a.tabschema='QRT_DWM'
AND a.tabschema = d.tabschema AND a.tabname = d.tabname
AND d.tbspaceid=b.tbspaceid
AND d.datapartitionid = 0
```

The two most commonly used columns in the sample fact tables are asset category (`SLVC_RPT_AST_CGY_ID`) and credit rating (`CR_RTG_ID`) for the "Asset valuation" report. These columns plus the time dimension (`TM_DIMENSION_ID`) were used as the MDC columns for the `SLVC_AST_VAL_FCT` fact table for the following reasons:

- These columns were the most frequently used columns in the report for filtering.
- The cardinality of the two columns was suitable for MDC cell population and could be coarsified.
- The time dimension was included as an MDC column to help reduce locking contention with data-ingest operations and to enable roll-in, roll-out capability.

Coarsification

Coarsification, within a DB2 database, is the process of creating a generated column to decrease the granularity of dimensions where your estimates show that the resulting MDC table would be sparsely populated.

A sparsely populated MDC table exists where the number of rows per cell (chosen MDC dimension columns) is less than the number of pages in a block (16 * 16K pages in an IBM PureData for Operational Analytics System). Since DB2 allocates 1 block per cell, it is important that each cell contains a healthy number of rows to avoid slower query performance through increased disk I/O.

Use generated columns where required to avoid creating sparsely populated MDC tables

For example, if using the earlier code sample does not identify column candidates with suitable low cardinality, then look to coarsify existing columns. For example, coarsifying the asset category and credit rating columns to reduce the cardinality in order to increase the number of rows per cell would create a more efficient MDC table.

Create a generated column in the fact table using the frequently used filter columns in the report. Use a divisible number that creates a suitable MDC candidate column as shown in the sample SQL statement above. The new column can then be added to the table as an MDC column, for example:

```
DIM_SLVC_RPT_AST_CGY_ID SMALLINT GENERATED ALWAYS AS
(SLVC_RPT_AST_CGY_ID/10),
DIM_CR_RTG_ID SMALLINT GENERATED ALWAYS AS (CR_RTG_ID/100)
```

Partitioning large tables for query performance and manageability

Use table partitioning to separate ranges of data within a table into data partitions to take advantage of DB2 features.

From a dimensional data warehouse perspective, a report or analytical query typically reads a large volume of rows in order to return a few rows. An efficient design must look to minimize the number of rows that are read to just those rows that are needed.

Build on an intelligent table space design strategy by partitioning your largest fact tables to facilitate query performance and enable flexibility when performing maintenance operations.

For example, by partitioning the fact table `SLVC_AST_VAL_FCT` by month, the following capabilities are enabled:

- Use the time dimension to determine ranges of data per month as the `DATE` column was used on the time dimension key.

  The DB2 optimizer can eliminate a data partition where the date range of the query does not match the date range of the data partition. This can help significantly reduce the number of rows read for a query.

- Define each index on the fact table as a partitioned index (option in InfoSphere Data Architect) and include the range partitioning key.

  Index maintenance can take place at data partition level and be targeted at active data partitions.

- Define end dates for each data partition as `EXCLUSIVE` so that the first day in the subsequent month can be specified as the end of the period range.

  This method is clearer and less prone to error than manually determining the correct end of month date.

- Assign each table (range) partition to a separate table space; for example "`February 2012`" is assigned to the `PD_AST_VAL_FEB2012` table space.

  This increases visibility and flexibility in maintenance operations, backup and recovery, and data lifecycle management. Aged data can easily be detached from the database helping to maintain a balance between active and inactive data.

The following DDL excerpt was generated from the physical data model and shows the part of the `CREATE TABLE` statement that included the table partitioning syntax. Data partitions were created for January, February, March, and so on, for 2012 with a separate data partition for rows that exist for dates before and after the ranges specified.

```
PARTITION BY RANGE (TM_DIMENSION_ID)
(
PART PAST STARTING(MINVALUE)
ENDING('2012-01-01') EXCLUSIVE IN PD_AST_VAL_PAST,
PART PART_2012_JAN STARTING('2012-01-01')
ENDING('2012-02-01') EXCLUSIVE IN PD_AST_VAL_JAN2012,
PART PART_2012_FEB STARTING ('2012-02-01')
ENDING('2012-03-01') EXCLUSIVE IN PD_AST_VAL_FEB2012,
PART PART_2012_MAR STARTING ('2012-03-01')
ENDING('2012-04-01') EXCLUSIVE IN PD_AST_VAL_MAR2012,
-- Syntax shortened: Months April to December removed.
PART PART_2012_DEC STARTING ('2013-01-01')
ENDING(MAXVALUE) IN PD_AST_VAL_FUTURE);
```

The first and last ranges are defined as starting from `MINVALUE` and running to `MAXVALUE` to prevent boundary violations.

Using DB2 V10.1, range partitioning can be aligned with multi-temperature storage groups. Additionally, range partitioning facilitates filtering at the granularity of the partition (month) by using partition elimination. Refer to the *Further reading* section for a link to the paper "*DB2 V10.1 Multi-temperature data management recommendations*".

Confirming data partition elimination in SELECT statements

Use the `EXPLAIN PLAN` statement to determine that data (range) partitions are being referenced in your queries. For example, the following explain plan output from a report query indicates that the query identified a specific data partition (range) to satisfy the query before the I/O operations take place.

```
Range 1)
Start Predicate: (Q6.TM_DIMENSION_ID = '12/01/2011')
Stop Predicate: (Q6.TM_DIMENSION_ID = '12/01/2011')
```

Indexing for performance

The use of MDC on fact tables reduces your need to create multiple indexes on fact tables. Instead, use indexing to facilitate query performance when columns that are not used in the MDC are referenced. The DB2 optimizer can take advantage of singular or composite indexes on foreign keys.

MDC facilitates access to data in multiple dimensions by organizing the data in dimensional blocks, which also reduces the requirement to reorganize the tables.

Indexes are primarily used to enhance query performance, but also can be used to govern how data is organized on dimension tables or to enforce unique constraints. In a data warehouse environment, focus on query performance by using these techniques:

- Use indexes for query performance only; constraints should be checked and enforced by the ETL process and made aware to the DB2 optimizer using informational constraints.
- Use partitioned indexes over global indexes to minimize query cost and index maintenance.

Enhancements in DB2 V10.1 that allow the optimizer to recognize data warehouse queries and use of a zigzag join can help improve performance and require a specific design pattern. A zigzag join can occur where a fact table and two or more dimension tables in a star schema are joined.

Use composite indexes to include those foreign keys that are used in query joins, including MDC columns

To enable the optimizer to use a zigzag join:

- Create a primary key on each dimension table in the physical data model to enforce uniqueness and provide an index for the optimizer to use. Primary keys on the parent table are also a mandatory requirement for foreign key constraints (informational or enforced) with the fact table.
- Create a composite index on the fact table that includes all frequently used join columns, including MDC columns. Refine the columns that are used during the query optimization process; the explain plan output helps identify the columns needed.

Using the Explain facility to determine zigzag join usage

To facilitate the zigzag join access plans over dimension keys not included in the MDC columns, a non-unique composite index is created over all the foreign keys to the dimension tables that are referenced in the example reports. For example:

```
CREATE INDEX QRT_DWM.IDX_SLVC_AST_VAL_FCT ON
"QRT_DWM"."SLVC_AST_VAL_FCT" ("SLVC_RPT_AST_CGY_ID", "CR_RTG_ID",
"ISSUR_OF_INV_ID", "ISSUR_CNTRY_LGL_SEAT_ID",
"CNTRY_OF_CUSTODY_OF_AST_ID", "TM_DIMENSION_ID") PARTITIONED;
```

To ensure that zigzag join operator is being achieved, use the `EXPLAIN PLAN` statement and the `db2exfmt` command to examine the access plan for the query and look for the ZZJOIN operator in the access plan graph produced. For example:

```
EXPLAIN PLAN FOR <SELECT statement>

db2exfmt -d modeldb5 -t -g
```

The following is a snippet of the explain plan output that shows the ZZJOIN operator. The output also shows a BTQ (Broadcast Table Queue) that would be a possible candidate for table replication.

```
                           |
                       0.0633706
                         ZZJOIN
                         ( 12)
                        57.4938
                           24
      +---------------------+------+--------------+
           3.31022 1         .06757  0.0179323
             TBSCAN          TBSCAN    IXSCAN
             ( 13)         ( 18)       ( 23)
```

```
            46.1982     4.52372     6.77185
               22          1           1
              |           |            |
             3.31022     1.06757     50318
         TEMP        TEMP DP-INDEX|   QRT_DWM
          ( 14)       ( 19)       TEST_INDEX
           46.1969     4.52241           Q6
            22          1
            |           |
      3.31022           1.06757
      BTQ               BTQ
      ( 15)             ( 20)
      46.1853           4.51201
      22                1
      |                 |
      3.31022           1.06757
      FETCH             FETCH
      ( 16)             ( 21)
      46.1597           4.48722
      22                1
      /---+----\        /----+-----\
      1000 1000         122         122
IXSCAN TABLE:QRT_DWM    IXSCAN TABLE: QRT_DWM
( 17) GEO_AREA          ( 22) SLVC_RPT_AST_CGY
8.86251     Q2          0.00777168 Q4
      2                 0
      |                 |
      1000              122
INDEX: QRT_DWM          INDEX: QRT_DWM
GEO_AREA_PK             SLVC_RPT_AST_CGY_PK
      Q2                Q4
```

Using partitioned MQTs to enhance query performance

Use materialized query tables (MQTs) to enhance query performance by precomputing expected aggregation queries. When you are creating an aggregation layer:

- Remove columns from the MQTs that are not included in your report query to improve the performance of the MQT REFRESH operation and allow the MQT to be used in multiple contexts.

  Since columns are not removed from the underlying tables, columns can be added to the MQT at a future point in time if needed.

- Do not write queries directly against the MQTs.

  Designing and implementing MQT is an iterative process even after the data warehouse goes live particularly if self-service analysis queries are used.

- Where possible, implement partitioned MQTs by using the same distribution keys, MDC columns, compression setting, and range partitioning as the underlying fact table.

  This method helps ensure that the MQT is more favorable to the optimizer for query rewrite.

- Include RUNSTATS for MQTs into your overall statistics collection strategy and always refresh statistics for an MQT after you refresh the MQT.

Defining MQTs for the sample reports and data model

For the sample reports, a nested MQT strategy was chosen and applied to the physical data model. By nesting MQTs, the refresh of one MQT uses the contents of another MQT, helping to reduce the refresh time for MQTs.

- Two MQTs were created for each side of the select (union) statement that was used in the parent report. Each MQT aggregated fact table data only.

  By creating separate MQTs for each side of the union clause, other reports and other MQTs that reference the fact tables can take advantage of the MQTs.

- An MQT was created for the child report which aggregates the asset fact data by the dimensions in the report. This MQT include dimension columns for each report filter in addition to the aggregated fact data.

  The design goal for this MQT is to achieve 'nested MQTs'; the refresh operation for this MQT uses the fact table MQTs to help maximize performance and efficiency when it is maintaining MQTs.

In our test environment, the asset dimension that is aggregated in the MQT, and the MQT had roughly 500,000 rows, compared to the 10,000,000 rows in the assets fact table. These numbers represents a 20:1 consolidation ratio, which is in line with the data warehouse best practice recommendation of greater than 10:1 consolidation, to allow the optimizer to favor the MQT over the source fact table.

Using Explain Plan to determine MQT optimizations

Use the DB2 explain plan tools to help ensure that the MQTs you created are being used by the optimizer.

Confirm that the DB2 optimizer is identifying the changes that you made to the database before you apply the changes back into the physical model

Confirm that the optimizer is rewriting the query to use the MQT created by running the query with the `EXPLAIN PLAN` statement and then using the `db2exfmt` command to format the resulting access plan. For example:

```
EXPLAIN PLAN FOR <SELECT statement used in report>
```

Issue the `db2exfmt` command against the target database:

```
db2exfmt -d modeldb5 -t -g
```

Confirm that the MQT is being used by looking at the resulting access plan graph or by the comments in the "Extended Diagnostic Information" section of the formatted plan. In the following example MQTs were identified by the optimizer in preparing a plan for the query:

```
Extended Diagnostic Information:
Diagnostic Identifier: 1
Diagnostic Details:EXP0148W The following MQT or statistical view
was considered in query matching: "QRT_DWM"."QRT_INV_AST_VAL".
Diagnostic Identifier: 2
Diagnostic Details: EXP0148W The following MQT or statistical
view was considered in query matching: "QRT_DWM". "QRT_INV_HLDG".
Diagnostic Identifier: 3
Diagnostic Details: EXP0149W The following MQT was used (from
those considered) in query matching: "QRT_DWM".
"MQT_REPORT1_AST".
Diagnostic Identifier: 4
Diagnostic Details: EXP0149W The following MQT was used (from
those considered) in query matching: "QRT_DWM"."QRT_INV_HLDG".
```

The cost for the access preceding plan, which leverages MQTs, showed a significant improvement over the cost without the MQT, and results in a similar increase in query performance.

Enabling and evaluating compression

Enabling compression can help reduce storage needs and increase query performance through reduced I/O and dimensional databases are suited to enabling compression given data patterns generated by conformed dimensions.

- At design time, compress all tables and indexes.
- Evaluate compression ratios during testing by using representative data.

The following query against the SYSCAT.TABLES catalog view shows the average row compression ratio that is achieved for the SLV_AST_VAL_FACT table. It shows that 77% of the pages are saved through compression which represents an average compression ratio of 5.34.

```
SELECT substr(TABNAME,1,30), AVGROWCOMPRESSIONRATIO,
PCTPAGESSAVED FROM SYSCAT.TABLES WHERE TABNAME='SLVC_AST_VAL_FCT'
```

The following query against the SYSCAT.INDEXES catalog view shows the percentage of pages that are saved for each index. For example, the composite index SLVC_AST_VAL_FCT_IN1 saved 64% of the pages in this index.

```
SELECT substr(indname,1,30) as INDNAME, substr(TABNAME,1,30) AS
TABNAME, substr(colnames,1,50) AS COLNAMES, PCTPAGESSAVED FROM
syscat.indexes WHERE TABNAME = 'SLVC_AST_VAL_FCT'
```

Replicating nonpartitioned tables

Replicating nonpartitioned tables places a copy of the table on to each database partition. This method enables collocated queries with partitioned fact tables, avoiding unnecessary communication between data hosts which takes place as uncompressed data exchanges.

Because the fact table can be collocated through sharing the distribution key with a single "major" dimension table (in this case FNC_AST) only, replicate the other dimension tables that are involved in the report queries to achieve collocation across the entire query. For example, the GEO_AREA table is replicated by using the following DDL:

```
CREATE TABLE REPL_GEO_AREA AS (SELECT * FROM GEO_AREA)
DATA INITIALLY DEFERRED REFRESH IMMEDIATE ENABLE
QUERY OPTIMIZATION MAINTAINED BY SYSTEM DISTRIBUTE BY REPLICATION
IN PD_REPL_DIM_TBL;
```

Use the MAINTAINED BY SYSTEM and REFRESH IMMEDIATE options where the number of rows in the nonpartitioned table is relatively small and subject to relatively few changes. When a replicated table contains many rows and is subject to many changes consider employing the REFRESH DEFERRED option to retain control of resources used.

Always update statistics for an MQT after you issue a REFRESH or SET INTEGRITY command against the table. For example:

```
RUNSTATS ON TABLE <TABLENAME> FOR SAMPLED DETAILED INDEXES ALL
```

Ingesting data into the database

Tools including IBM DataStage, InfoSphere SQL Warehousing (SQW), and the DB2 load utility are documented and provide various features for ETL processing. Where data cleansing and transformations are needed, the choice exists to perform these tasks within the database layer or outside of the database layer.

A staging area within the database is commonly used to hold data for validation, cleansing, and transformation before transferring the data into the production tables. Create the staging layer in the physical data model and place the tables within the same database partition groups but in a separate schema and in a separate table space.

Staging tables are used under one or more of the following conditions:

- The data is not table ready – significant transformations or data cleansing is needed.
- The business is not ready – you want to avoid making the data available until the business asks for it or an event occurs. However, you want to have the data ready to insert into the production tables.
- Data referencing – data from different data sources must be cross referenced before inserted into the production tables.

In contrast to the DB2 load utility the DB2 ingest client utility, available in DB2 V10.1, can be used to populate the database with logged transactions and without the necessity to use staging tables and without off-lining target production tables.

Ingest provides a scalable solution for ingesting data into a partitioned database because data can be prepared on the client and directed to individual database partitions, avoiding any bottleneck through the coordinator database partition.

The ingest utility has a number of features that make it an effective way to get data into the production database:

- You need other applications to access or update the table while it is being populated.
- The input file contains fields that you want to transform or skip over.
- You need to use `MERGE` or `DELETE` functionality.
- You need to recover and continue on when the utility gets a recoverable error.

Use the ingest utility to populate the staging and production tables as a concurrent operation; ingest inserts data as a logged operation, is concurrent with backup, query, and other database operations and helps maintain a recoverable database.

For example, in populating the fact table `SLVC_AST_VAL_FCT` the following ingest command was used:

```
INGEST FROM source_SLVC_AST_VAL_FCT.del format DELIMITED
```

```
restart new  (
$f_CO_ID BIGINT external,
$f_CNTRY_OF_CUSTODY_OF_AST_ID BIGINT external)
merge into SLVC_AST_VAL_FCT
on (CO_ID = $f_CO_ID)
when matched and (CNTRY_OF_CUSTODY_OF_AST_ID =
 $f_CNTRY_OF_CUSTODY_OF_AST_ID) and (ESR_CCY_ID = 'USD')
then
  update set UNIT_PRC = UNIT_PRC * 0.0024
when matched and (UNIT_PRC > 20) then
  update set UNIT_PRC = 20
when matched and (CR_RTG_ID = 'BBB') then
 delete
```

The example that is shown illustrates:

- A conditional expression is used to merge data for specific rows. This demonstrates how the `INGEST` command can combine operations and use expressions to offload some of the analysis and data cleansing work that other processes perform in a staging table.
- The `RESTART NEW` parameter of the `INGEST` command determines a new instance of the ingest process. The `RESTART CONTINUE` parameter is used to restart an ingest process from the position where it stopped; the ingest utility maintains a restart position for each ingest process.

When you are using the ingest utility, choose the `COMMIT COUNT` or `COMMIT_PERIOD` options as a method for committing rows to the database. These options can be specified by:

- The number of rows that are ingested before a commit

```
INGEST SET COMMIT_COUNT 100
```

- The length of time in seconds between commits

```
INGEST SET COMMIT_PERIOD 90
```

The following example shows how an ingest process when interrupted can be restarted following an interruption:

```
INGEST FROM source_SLVC_AST_VAL_FCT.del format DELIMITED
restart new "update_SLVC_AST_VAL_FCT_001" (
$f_CO_ID BIGINT  external,
$f_CNTRY_OF_CUSTODY_OF_AST_ID BIGINT external,
$f_UNIT_PRC DECIMAL external)
update SLVC_AST_VAL_FCT
    set UNIT_PRC = $f_UNIT_PRC where CO_ID = $f_CO_ID and
CNTRY_OF_CUSTODY_OF_AST_ID <> $f_CNTRY_OF_CUSTODY_OF_AST_ID

<CTRL-C> Interrupt occurs

ingest from source_SLVC_AST_VAL_FCT.del format DELIMITED
restart continue "update_SLVC_AST_VAL_FCT_001" (
$f_CO_ID BIGINT external,
$f_CNTRY_OF_CUSTODY_OF_AST_ID BIGINT external,
$f_UNIT_PRC DECIMAL external)
update SLVC_AST_VAL_FCT
    set UNIT_PRC = $f_UNIT_PRC
    where CO_ID = $f_CO_ID and
      CNTRY_OF_CUSTODY_OF_AST_ID <>
$f_CNTRY_OF_CUSTODY_OF_AST_ID
```

where:

- "source_SLVC_AST_VAL_FCT.del" is the name of the source text file
- "update_SLVC_AST_VAL_FCT_001" is the named recoverable ingest job

The ingest job resumes from the point of last commit, according to settings specified by either the `COMMIT_COUNT` or `COMMIT_PERIOD`.

When used with the temporal feature, a full history of changes can be recorded in the temporal history table.

Use identity columns for the dimension table primary keys and omit that primary key column from the column specification list in the ingest control file.

The simple example below shows how the ingest utility can be used to populate the date dimension in the sample industry data model used.

```
INGEST FROM file source_date_dim.del format DELIMITED ($v_year
INTEGER EXTERNAL, $v_loopDate BIGINT external) restart off
INSERT INTO TM_DIMENSION (
CDR_DT, CDR_YR, CDR_QTR, DAY_OF_WK, CDR_MTH, CDR_MTH_NM,
WK_NUM_IN_CDR_YR, DAY_NUM_IN_CDR_YR, DAY_NUM_IN_CDR_MTH, WEEKDAY,
PBLC_HOL)
VALUES (
    DATE(cast($v_loopDate as INTEGER)),
    YEAR( date(cast($v_loopDate as INTEGER)) ),
    CASE month( date(cast($v_loopDate as INTEGER)) )
      when 1 then concat('Q1 ', cast(YEAR( date(cast($v_loopDate
as INTEGER))  ) as char(4)))
      when 2 then concat('Q1 ', cast(year( date(cast($v_loopDate
as INTEGER))  ) as char(4)))
      when 3 then concat('Q1 ', cast(year( date(cast($v_loopDate
as INTEGER))  ) as char(4)))
      when 4 then concat('Q2 ', cast(year( date(cast($v_loopDate
as INTEGER))  ) as char(4)))
      when 5 then concat('Q2 ', cast(year( date(cast($v_loopDate
as INTEGER))  ) as char(4)))
      when 6 then concat('Q2 ', cast(year( date(cast($v_loopDate
as INTEGER))  ) as char(4)))
      when 7 then concat('Q3 ', cast(year( date(cast($v_loopDate
as INTEGER))  ) as char(4)))
      when 8 then concat('Q3 ', cast(year( date(cast($v_loopDate
as INTEGER))  ) as char(4)))
```

```
      when 9 then concat('Q3 ', cast(year( date(cast($v_loopDate
as INTEGER))  ) as char(4)))
      when 10 then concat('Q4 ', cast(year( date(cast($v_loopDate
as INTEGER))  ) as char(4)))
      when 11 then concat('Q4 ', cast(year( date(cast($v_loopDate
as INTEGER))  ) as char(4)))
      when 12 then concat('Q4 ', cast(year( date(cast($v_loopDate
as INTEGER))  ) as char(4)))
      else null
      end,
    dayname( date(cast($v_loopDate as INTEGER)) ),
    month( date(cast($v_loopDate as INTEGER)) ),
    monthname( date(cast($v_loopDate as INTEGER)) ),
    week( date(cast($v_loopDate as INTEGER)) ),
    dayofyear( date(cast($v_loopDate as INTEGER)) ),
    day( date(cast($v_loopDate as INTEGER)) ),
    CASE dayofweek( date(cast($v_loopDate as INTEGER)) )
      when 1 then 'N'
      when 7 then 'N'
      else 'Y'
    END,
    'N')
```

Where:

- The file, source_date_dim.del, simply contains the numbers 1 to 365 to represent each day in the year.
- CDR_DT is the primary key with a data type of DATE

Using temporal tables to effectively manage change

Using temporal tables, the database can store and retrieve time-based data without more application logic. For example, a database can store the history of a table (deleted rows or the original values of rows that were updated) so you, or your auditors, can understand the history of a row in a table or retrieve data as of a specific point in time.

DB2 supports three types of temporal tables:

- System-period temporal tables (STTs).

  DB2 transparently keeps a history of updated and deleted rows over time.

- Application-period temporal tables (ATTs).

  New SQL constructs allow users to insert, query, update, and delete data in the past, present, or future. DB2 automatically applies temporal constraints and "row-splits" to correctly maintain the application-supplied business time, also known as valid time.

- Bitemporal tables (BTTs).

  This combination enables applications to manage the business validity of their data while DB2 keeps a full history of any updates and deletes. Every BTT is also an STT and an ATT.

Using temporal tables can help you track data changes over time and provide an efficient way to address Solvency II auditing and compliance requirements.

The temporal data feature is described in detail in the best practices paper titled "*Best practices: Temporal data management with DB2*" which is referenced in the *Further reading* section.

Implementing system-period temporal time for a dimension table

This example implements the STT feature to the agreement collection dimension (`AGRM_COLL`) to record all changes to the table at a system date level.

To enable STT for the agreement collection table the following `CREATE TABLE` statement would be used where the columns `VLF_FM_DT` (Valid From Date), `VLF_TO_DT` (Valid To Date), and `VLD_TX_START` (Valid Transaction Start) are added to the table and the keyword `PERIOD SYSTEM_TIME` determines the columns to be used:

```
CREATE TABLE "QRT_DWM"."AGRM_COLL" (
"AGRM_COLL_ID" BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY
(START WITH 1 INCREMENT BY 1 MINVALUE 1 MAXVALUE
9223372036854775807 NO CYCLE CACHE 20 NO ORDER ),
"ANCHOR_ID" BIGINT, "CGY_SCM_NUM" VARCHAR(20),
"DSC" VARCHAR(256), "EFF_FM_DT" TIMESTAMP, "EFF_TO_DT" TIMESTAMP,
"END_DT" DATE, "EXT_REFR" VARCHAR(20), "MX_SZ" INTEGER,
"NM" VARCHAR(20), "ONLY_RULE_DRVN" CHAR(1), "REFRESH_DT" DATE,
"STRT_DT" DATE, "TP" VARCHAR(20),
"VLD_FM_DT" TIMESTAMP(12) GENERATED ALWAYS AS ROW BEGIN NOT NULL,
"VLD_TO_DT" TIMESTAMP(12) GENERATED ALWAYS AS ROW END NOT NULL,
"VLD_TX_START" TIMESTAMP(12) GENERATED ALWAYS AS TRANSACTION
START ID IMPLICITLY HIDDEN, PERIOD SYSTEM_TIME(VLD_FM_DT,
VLD_TO_DT)
)
DISTRIBUTE BY HASH(AGRM_COLL_ID) DATA CAPTURE NONE
COMPRESS YES IN TS_PDPG_MED_DIMENSIONS;
```

The temporal feature is enabled by issuing the following command which specifies the history table to be used:

```
ALTER TABLE "QRT_DWM"."AGRM_COLL" ADD VERSIONING USE HISTORY
TABLE "QRT_DWM"."AGRM_COLL_HISTORY"
```

When creating the history table:

- Use the same distribution key as the parent dimension table
- Use the same compression state
- Use a separate table space to contain the history table

The temporal history table should use the same attributes and characteristics as the parent table for which you are enabling the temporal feature

Using temporal to determine system-period temporal state

Use the system-period temporal table to facilitate 'as of' reporting and to determine when data changed and what the before and after column values were.

The SQL statement below uses the system-period temporal table to return the values of a specific column, EXT_REFR (External Reference) for a specific time frame:

```
SELECT AGRM_COLL_ID, REFRESH_DT, EXT_REFR, VLD_FM_DT, VLD_TO_DT
FROM QRT_DWM.AGRM_COLL
FOR SYSTEM_TIME FROM '2010-01-01' TO '2012-10-01'
WHERE AGRM_COLL_ID = 9741
```

The output from the SQL statement shows that three versions of the row exist for the time-period with varying values for the column retrieved:

```
AGRM_COLL_ID REFRESH_DT EXT_REFR  VLD_FM_DT
VLD_TO_DT
------------ ---------- --------- --------------------------
        9741 04/07/2011 RQF-28/W  2010-01-01-
00.00.00.000000000000 2012-09-13-08.21.28.143129000000
        9741 04/07/2011 RQF-26/X  2012-09-13-
08.21.28.143129000000 2012-09-13-08.23.03.818945000000
```

```
        9741 04/07/2011 RQF-28/W  2012-09-13-
08.23.03.818945000000 9999-12-30-00.00.00.000000000000
```

The following SQL statement uses the system-period temporal table to return the values of a specific column, EXT_REFR (External Reference) as of a specific time:

```
SELECT AGRM_COLL_ID, REFRESH_DT, EXT_REFR, VLD_FM_DT, VLD_TO_DT
FROM QRT_DWM.AGRM_COLL
FOR SYSTEM_TIME AS OF '2012-10-01' WHERE AGRM_COLL_ID = 9741
```

The output from the SQL statement shows that three versions of the row exist for the time-period with varying values for the column retrieved:

```
AGRM_COLL_ID REFRESH_DT EXT_REFR VLD_FM_DT
VLD_TO_DT
------------ ---------- -------------------- --------------------
        9741 04/07/2011 RQF-28/W 2012-09-13-08.23.03.818945000000
9999-12-30-00.00.00.000000000000
        9741 04/07/2011 ABCDEFG  2012-09-13-08.21.28.143129000000
2012-09-13-08.23.03.818945000000
        9741 04/07/2011 RQF-28/W 2012-09-13-08.23.03.818945000000
9999-12-30-00.00.00.000000000000
```

Implementing business-period temporal time for a dimension table

Use business time and application-period temporal tables, if you need to describe when information is valid in the real world, outside of DB2.

When you create a temporal table to include business period temporal time, you are allowing DB2 software to create multiple rows for a dimension table where each row represents data for an effective business date range.

For example, to apply business period temporal time for the Agreement Collection dimension, using the existing columns EFF_FM_DT and EFF_TO_DT, the following create table statement would be generated:

```
CREATE TABLE "QRT_DWM"."AGRM_COLL" (
"AGRM_COLL_ID" BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY
(START WITH 1 INCREMENT BY 1 MINVALUE 1 MAXVALUE
9223372036854775807 NO CYCLE CACHE 20 NO ORDER ),
"ANCHOR_ID" BIGINT, "CGY_SCM_NUM" VARCHAR(20),
"DSC" VARCHAR(256), "EFF_FM_DT" DATE NOT NULL, "EFF_TO_DT" DATE
NOT NULL, "END_DT" DATE, "EXT_REFR" VARCHAR(20), "MX_SZ" INTEGER,
"NM" VARCHAR(20), "ONLY_RULE_DRVN" CHAR(1), "REFRESH_DT" DATE,
"STRT_DT" DATE, "TP" VARCHAR(20),
PERIOD BUSINESS_TIME(EFF_FM_DT, EFF_TO_DT)
)
DISTRIBUTE BY HASH(AGRM_COLL_ID) DATA CAPTURE NONE
COMPRESS YES IN TS_PDPG_MED_DIMENSIONS;
```

To accommodate multiple rows per dimension key, the business temporal columns are added to the primary key on the dimension table to make a composite primary key.

The effect of this is that the foreign key on the fact table that is aligned with the dimension table cannot be enforced as there can be more than one row in the dimension table for the row in the fact table.

When implementing business-period temporal time for a dimension table, remove the foreign key on the associated fact table.

Conclusion

Deploying an industry model solution in a partitioned DB2 database can be successfully achieved by following the recommendations in this paper.

When you implement a logical data model it is important that you scope only those entities that relate to your business requirements. Add other entities as your business needs change and grow. This helps to reduce your initial workload in getting the database into production and avoids implementing a database with many empty tables.

Before you make detailed database design decisions implement database architecture in line with best practice recommendations. This helps to avoid costly outages in your production environment when data movement or maintenance operations need to take place.

When you build a partitioned database for a data warehouse environment, incorporate features available in the DB2 software. These include multi-dimensional clustering, table partitioning, compression and partitioned indexes.

When you optimize your database design for production use ensure that you use a partitioned database environment populated with relevant data and use queries generated from specific reporting requirements and intended data ingest and database maintenance operations.

Good database architecture and design decisions applied to your physical data model result in a production data warehouse that can accommodate business needs for reporting, data availability, regulatory compliance and growth.

Best practices

- Apply architecture and database design changes specific to DB2 databases to the physical database model rather than the logical data model to help accommodate future upgrades
- Refine your logical data model scope to address only your current data and reporting needs. This targets just those tables accessed by ETL, queries and maintenance operations.
- Minimize the number of database partition groups and avoid overlapping database partition groups on the same data host to avoid adding complexity to resource allocation and monitoring
- Employ intelligent table space design to facilitate concurrent database maintenance operations and help avoid costly reorganization tasks in production
- Place table data and index data in separate table spaces to provide more flexibility for operational maintenance and data lifecycle strategy
- Use information constraints instead of enforced constraints to minimize the effect of unique index maintenance when you are populating fact tables
- Use the DATE data type for the primary key on the date/time dimension table. This method enables data partitions to be date-based and facilitates query performance.
- Use multi-dimensional clustering tables to organize data in all fact tables to reduce the need for regular indexes and associated maintenance operations
- Use composite indexes to include those foreign keys that are used in query joins, including MDC columns
- Confirm that the DB2 optimizer is identifying the changes that you made to the database before you apply the changes back into the physical model

Appendix A. Test environment

The test environment used in the research and development of this paper was an IBM PureData for Operational Analytics System which has a shared-nothing architecture. InfoSphere Warehouse V10.1 containing DB2 V10.1 was installed. The database was populated with over 1 TB of data to test the concepts and recommendations in this paper at scale.

Figure 4 illustrates the architecture of the IBM PureData for Operational Analytics System. The administration host contained five database partitions; one for the coordinator database partition, and four data database partitions. Two data hosts with eight database partitions on each data host completed the partitioned database.

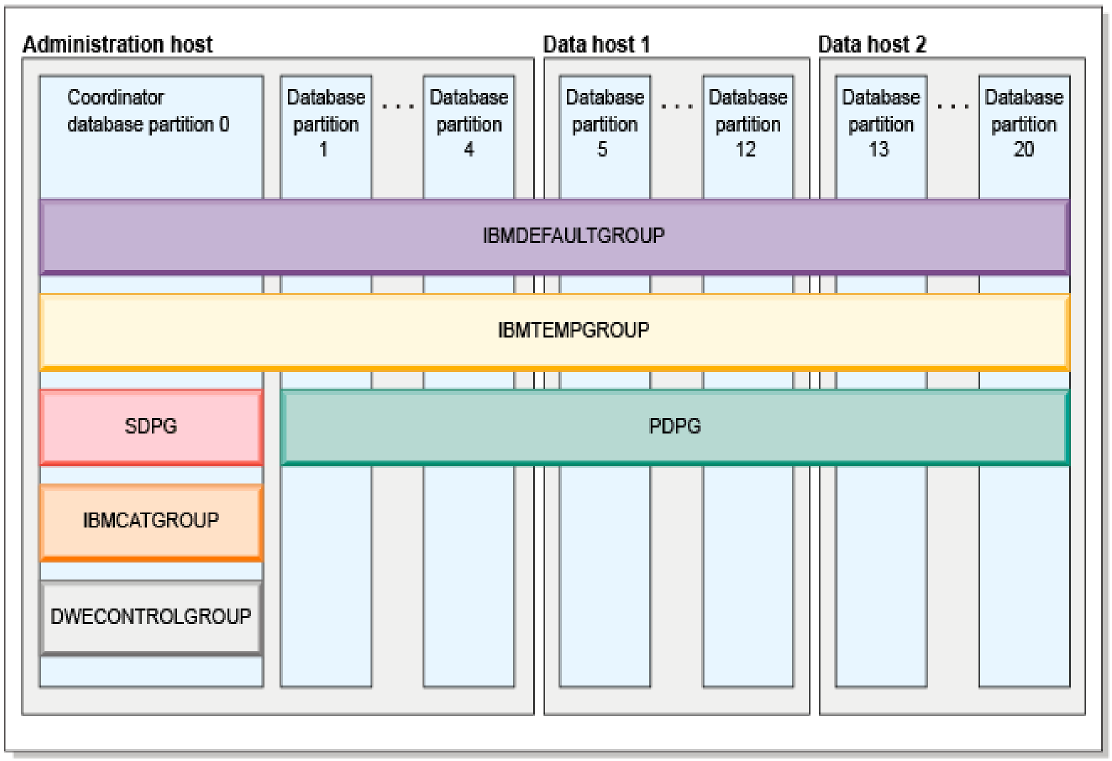


Figure 4 IBM PureData for Operational Analytics System database architecture

Appendix B Sample Queries

This section lists the main queries that were used by sample reports in this paper.

Query for report example 1: Assets by region, by counterparty, by credit rating

This is the query similar to that used by the IBM Cognos report and submitted to DB2:

```
--QUERY for selecting Investment Holdings.
SELECT
SLVC_INV_HLDG_FCT.SLVC_RPT_AST_CGY_ID AS "CIC",
SLVC_RPT_AST_CGY.TP AS "ID Code Type",
SLVC_RPT_AST_CGY.NM AS "SII Category Name",
"Asset Custody Country".CNTRY AS "Country of Custody",
"Counterparty".AGRM_REFR AS "Counterparty Issuer Name",
CR_RTG.EXT_REFR AS "External rating",
CDR_DT.CDR_YR, CDR_DT.CDR_MTH,
SUM(SLVC_INV_HLDG_FCT.QTY) AS "Quantity",
SUM(SLVC_INV_HLDG_FCT.SLVC_II_VAL) AS "Unit SII price",
SUM(SLVC_INV_HLDG_FCT.ACQ_COST) AS "Acquisition cost",
SUM(SLVC_INV_HLDG_FCT.QTY * SLVC_INV_HLDG_FCT.SLVC_II_VAL) AS
"Total SII price",
SUM(SLVC_INV_HLDG_FCT.ACR_INT) AS "Accrued Interest"
FROM
SLVC_INV_HLDG_FCT, SLVC_RPT_AST_CGY, FNC_SERVICES_RL AS
"Counterparty",
GEO_AREA AS "Asset Custody Country", CR_RTG, CDR_DT
WHERE
SLVC_INV_HLDG_FCT.SLVC_RPT_AST_CGY_ID =
SLVC_RPT_AST_CGY.SLVC_RPT_AST_CGY_ID AND
SLVC_INV_HLDG_FCT.ISSUR_OF_INV_ID =
"Counterparty".FNC_SERVICES_RL_ID AND
SLVC_INV_HLDG_FCT.CNTRY_OF_CUSTODY_OF_AST_ID = "Asset Custody
Country".GEO_AREA_ID AND
SLVC_INV_HLDG_FCT.CR_RTG_ID = CR_RTG.CR_RTG_ID AND
SLVC_INV_HLDG_FCT.TM_DIMENSION_ID = CDR_DT.CDR_DT
GROUP BY
SLVC_INV_HLDG_FCT.SLVC_RPT_AST_CGY_ID,
SLVC_RPT_AST_CGY.TP, "Asset Custody Country".CNTRY,
"Counterparty".AGRM_REFR, SLVC_RPT_AST_CGY.NM,
CR_RTG.EXT_REFR, CDR_DT.CDR_YR, CDR_DT.CDR_MTH
UNION ALL
-- SELECT statement for selecting Asset Valuations looks like:
SELECT
--SLVC_AST_VAL_FCT.FNC_AST_ID,
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID AS "CIC",
SLVC_RPT_AST_CGY.TP AS "ID Code Type",
SLVC_RPT_AST_CGY.NM AS "SII Category Name",
"Asset Custody Country".CNTRY AS "Country of Custody",
"Counterparty".AGRM_REFR AS "Counterparty Issuer Name",
CR_RTG.EXT_REFR AS "External rating",
CDR_DT.CDR_YR, CDR_DT.CDR_MTH,
null AS "Quantity",
SUM(SLVC_AST_VAL_FCT.UNIT_PRC) AS "Unit SII price",
null AS "Acquisition cost",
SUM(SLVC_AST_VAL_FCT.UNIT_PRC) AS "Total SII price",
null AS "Accrued Interest"
FROM
SLVC_AST_VAL_FCT, SLVC_RPT_AST_CGY, FNC_SERVICES_RL AS
"Counterparty",
GEO_AREA AS "Asset Issuer Country", GEO_AREA AS "Asset Custody
Country", CR_RTG, CDR_DT
WHERE
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID =
SLVC_RPT_AST_CGY.SLVC_RPT_AST_CGY_ID AND
SLVC_AST_VAL_FCT.ISSUR_OF_INV_ID =
"Counterparty".FNC_SERVICES_RL_ID AND
SLVC_AST_VAL_FCT.CNTRY_OF_CUSTODY_OF_AST_ID = "Asset Custody
Country".GEO_AREA_ID AND
SLVC_AST_VAL_FCT.CR_RTG_ID = CR_RTG.CR_RTG_ID AND
SLVC_AST_VAL_FCT.TM_DIMENSION_ID = CDR_DT.CDR_DT
GROUP BY
--SLVC_AST_VAL_FCT.FNC_AST_ID,
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID,
SLVC_RPT_AST_CGY.TP, "Asset Custody Country".CNTRY,
"Counterparty".AGRM_REFR, SLVC_RPT_AST_CGY.NM,
CR_RTG.EXT_REFR, CDR_DT.CDR_YR, CDR_DT.CDR_MTH;
```

Materialized query table for asset valuations

This is the MQT created for the asset valuation side of the SQL statement that was submitted in the previous example. The DB2 optimizer rewrites the query plan for the SQL statement to take advantage of the MQT where a more efficient plan is estimated.

```
CREATE TABLE QRT_DWM.QRT_INV_AST_VAL AS (
SELECT
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.CR_RTG_ID,
SLVC_AST_VAL_FCT.ISSUR_OF_INV_ID,
SLVC_AST_VAL_FCT.ISSUR_CNTRY_LGL_SEAT_ID,
SLVC_AST_VAL_FCT.CNTRY_OF_CUSTODY_OF_AST_ID,
SLVC_AST_VAL_FCT.TM_DIMENSION_ID,
SLVC_AST_VAL_FCT.DIM_SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.DIM_CR_RTG_ID,
SUM(UNIT_PRC) as UNIT_PRC
FROM SLVC_AST_VAL_FCT
GROUP BY
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.CR_RTG_ID,
SLVC_AST_VAL_FCT.ISSUR_OF_INV_ID,
SLVC_AST_VAL_FCT.ISSUR_CNTRY_LGL_SEAT_ID,
SLVC_AST_VAL_FCT.CNTRY_OF_CUSTODY_OF_AST_ID,
SLVC_AST_VAL_FCT.TM_DIMENSION_ID,
SLVC_AST_VAL_FCT.DIM_SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.DIM_CR_RTG_ID
)
DATA INITIALLY DEFERRED REFRESH DEFERRED
ENABLE QUERY OPTIMIZATION MAINTAINED BY SYSTEM
```

```
COMPRESS YES
ORGANIZE BY DIMENSIONS ( (DIM_SLVC_RPT_AST_CGY_ID),
(DIM_CR_RTG_ID) )
DISTRIBUTE BY HASH(SLVC_RPT_AST_CGY_ID)
PARTITION BY RANGE (TM_DIMENSION_ID)
(
PART PAST STARTING(MINVALUE)
ENDING('2010-01-01') EXCLUSIVE IN TS_PD_MQT_QRT_INV,
PART PART_2010_JAN STARTING('2010-01-01')
ENDING('2010-02-01') EXCLUSIVE IN TS_PD_MQT_QRT_INV,
PART PART_2010_FEB STARTING ('2010-02-01')
ENDING('2010-03-01') EXCLUSIVE IN TS_PD_MQT_QRT_INV,
..
PART PART_2011_NOV STARTING ('2011-11-01')
ENDING('2011-12-01') EXCLUSIVE IN TS_PD_MQT_QRT_INV,
PART PART_2011_DEC STARTING ('2011-12-01')
ENDING('2012-01-01') EXCLUSIVE IN TS_PD_MQT_QRT_INV,
PART PART_FUTURE STARTING ('2012-01-01')
ENDING(MAXVALUE) IN TS_PD_MQT_QRT_INV);
```

To prepare the MQT for use after the initial take-on of historical data into the warehouse from operational sources, use the LOAD FROM CURSOR approach which is a fast method for populating the table; a logged operation is not needed here.

```
DECLARE C_CUR CURSOR FOR
(SELECT
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.CR_RTG_ID,
SLVC_AST_VAL_FCT.ISSUR_OF_INV_ID,
SLVC_AST_VAL_FCT.ISSUR_CNTRY_LGL_SEAT_ID,
SLVC_AST_VAL_FCT.CNTRY_OF_CUSTODY_OF_AST_ID,
SLVC_AST_VAL_FCT.TM_DIMENSION_ID,
SLVC_AST_VAL_FCT.DIM_SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.DIM_CR_RTG_ID,
SUM(UNIT_PRC) as UNIT_PRC
FROM SLVC_AST_VAL_FCT
GROUP BY
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.CR_RTG_ID,
SLVC_AST_VAL_FCT.ISSUR_OF_INV_ID,
SLVC_AST_VAL_FCT.ISSUR_CNTRY_LGL_SEAT_ID,
SLVC_AST_VAL_FCT.CNTRY_OF_CUSTODY_OF_AST_ID,
SLVC_AST_VAL_FCT.TM_DIMENSION_ID,
SLVC_AST_VAL_FCT.DIM_SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.DIM_CR_RTG_ID);

LOAD FROM C_CUR OF CURSOR REPLACE
INTO QRT_DWM.QRT_INV_AST_VAL NONRECOVERABLE;

SET INTEGRITY FOR QRT_DWM.QRT_INV_AST_VAL ALL IMMEDIATE
UNCHECKED;
```

After the initial data taken and the warehouse is in production, the following approach can use the range partitioned configuration of the MQT to refresh only the new data that was ingested for the current monthly reporting period:

1. Drop the staging table

```
DROP TABLE MQT_STAGE.QRT_INV_AST_VAL_STAGE;
```

2. Re-create the staging table

```
CREATE TABLE MQT_STAGE.QRT_INV_AST_VAL_STAGE
LIKE QRT_DWM.QRT_INV_AST_VAL
COMPRESS YES
DISTRIBUTE BY HASH(SLVC_RPT_AST_CGY_ID)
ORGANIZE BY DIMENSIONS ( (DIM_SLVC_RPT_AST_CGY_ID),
(DIM_CR_RTG_ID) )
IN TS_PD_MQT_QRT_INV;
```

3. Populate the staging table

```
INSERT INTO MQT_STAGE.QRT_INV_AST_VAL_STAGE
SELECT
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.CR_RTG_ID,
SLVC_AST_VAL_FCT.ISSUR_OF_INV_ID,
SLVC_AST_VAL_FCT.ISSUR_CNTRY_LGL_SEAT_ID,
SLVC_AST_VAL_FCT.CNTRY_OF_CUSTODY_OF_AST_ID,
SLVC_AST_VAL_FCT.TM_DIMENSION_ID,
SLVC_AST_VAL_FCT.DIM_SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.DIM_CR_RTG_ID,
SUM(UNIT_PRC) as UNIT_PRC
FROM SLVC_AST_VAL_FCT
WHERE TM_DIMENSION_ID = '2011-11-01'
GROUP BY
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.CR_RTG_ID,
SLVC_AST_VAL_FCT.ISSUR_OF_INV_ID,
SLVC_AST_VAL_FCT.ISSUR_CNTRY_LGL_SEAT_ID,
SLVC_AST_VAL_FCT.CNTRY_OF_CUSTODY_OF_AST_ID,
SLVC_AST_VAL_FCT.TM_DIMENSION_ID,
SLVC_AST_VAL_FCT.DIM_SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.DIM_CR_RTG_ID;
```

4. Remove the original MQT from the table

```
ALTER TABLE QRT_DWM.QRT_INV_AST_VAL DROP MATERIALIZED QUERY;
```

5. Attach the new MQT data partition

```
ALTER TABLE QRT_DWM.QRT_INV_AST_VAL ATTACH PARTITION
PART_2011_NOV
STARTING ('2011-11-01')
ENDING('2011-12-01') EXCLUSIVE
FROM MQT_STAGE.QRT_INV_AST_VAL_STAGE;
```

6. Set Integrity for the entire MQT

```
SET INTEGRITY FOR QRT_DWM.QRT_INV_AST_VAL
ALLOW WRITE ACCESS IMMEDIATE CHECKED INCREMENTAL;
```

7. Reintroduce the MQT to the target table

```
ALTER TABLE QRT_DWM.QRT_INV_AST_VAL ADD MATERIALIZED QUERY
SELECT
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.CR_RTG_ID,
SLVC_AST_VAL_FCT.ISSUR_OF_INV_ID,
SLVC_AST_VAL_FCT.ISSUR_CNTRY_LGL_SEAT_ID,
SLVC_AST_VAL_FCT.CNTRY_OF_CUSTODY_OF_AST_ID,
SLVC_AST_VAL_FCT.TM_DIMENSION_ID,
SLVC_AST_VAL_FCT.DIM_SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.DIM_CR_RTG_ID,
SUM(UNIT_PRC) as UNIT_PRC
FROM SLVC_AST_VAL_FCT
GROUP BY
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.CR_RTG_ID,
SLVC_AST_VAL_FCT.ISSUR_OF_INV_ID,
SLVC_AST_VAL_FCT.ISSUR_CNTRY_LGL_SEAT_ID,
SLVC_AST_VAL_FCT.CNTRY_OF_CUSTODY_OF_AST_ID,
SLVC_AST_VAL_FCT.TM_DIMENSION_ID,
SLVC_AST_VAL_FCT.DIM_SLVC_RPT_AST_CGY_ID,
SLVC_AST_VAL_FCT.DIM_CR_RTG_ID)
DATA INITIALLY DEFERRED REFRESH DEFERRED;
```

8. Set integrity and collect statistics for the target table

```
SET INTEGRITY FOR QRT_DWM.QRT_INV_AST_VAL ALL IMMEDIATE
UNCHECKED;

RUNSTATS ON TABLE QRT_DWM.QRT_INV_AST_VAL FOR SAMPLED DETAILED
INDEXES ALL
```

Create a materialized query table dimensional joins

An additional MQT is created to materialize the dimensional joins for report 1, which looks like the following on the assets side:

```
CREATE TABLE MQT_REPORT1_AST AS (
SELECT
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID,
SLVC_RPT_AST_CGY.TP, SLVC_RPT_AST_CGY.NM, "Asset Custody
Country".CNTRY, "Counterparty".AGRM_REFR, CR_RTG.EXT_REFR,
CDR_DT.CDR_YR, CDR_DT.CDR_MTH, SLVC_AST_VAL_FCT.TM_DIMENSION_ID,
SUM(SLVC_AST_VAL_FCT.UNIT_PRC) AS "Unit SII price",
SUM(SLVC_AST_VAL_FCT.UNIT_PRC) AS "Total SII price"
FROM
SLVC_AST_VAL_FCT, SLVC_RPT_AST_CGY, FNC_SERVICES_RL AS
"Counterparty",
GEO_AREA AS "Asset Custody Country", CR_RTG, CDR_DT
WHERE
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID =
SLVC_RPT_AST_CGY.SLVC_RPT_AST_CGY_ID AND
SLVC_AST_VAL_FCT.ISSUR_OF_INV_ID =
"Counterparty".FNC_SERVICES_RL_ID AND
SLVC_AST_VAL_FCT.CNTRY_OF_CUSTODY_OF_AST_ID = "Asset Custody
Country".GEO_AREA_ID AND
SLVC_AST_VAL_FCT.CR_RTG_ID = CR_RTG.CR_RTG_ID AND
SLVC_AST_VAL_FCT.TM_DIMENSION_ID = CDR_DT.CDR_DT
GROUP BY
--SLVC_AST_VAL_FCT.FNC_AST_ID,
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID,
SLVC_RPT_AST_CGY.TP, "Asset Custody Country".CNTRY,
"Counterparty".AGRM_REFR, SLVC_RPT_AST_CGY.NM,
CR_RTG.EXT_REFR, SLVC_AST_VAL_FCT.TM_DIMENSION_ID,
CDR_DT.CDR_YR, CDR_DT.CDR_MTH
)
DATA INITIALLY DEFERRED REFRESH DEFERRED
ENABLE QUERY OPTIMIZATION MAINTAINED BY SYSTEM
COMPRESS YES
DISTRIBUTE BY HASH(SLVC_RPT_AST_CGY_ID)
PARTITION BY RANGE (TM_DIMENSION_ID)
(
PART PAST STARTING(MINVALUE)
ENDING('2010-01-01') EXCLUSIVE IN TS_PD_MQT_QRT_INV,
PART PART_2010_JAN STARTING('2010-01-01')
ENDING('2010-02-01') EXCLUSIVE IN TS_PD_MQT_QRT_INV,
PART PART_2010_FEB STARTING ('2010-02-01')
ENDING('2010-03-01') EXCLUSIVE IN TS_PD_MQT_QRT_INV,
..
PART PART_2010_NOV STARTING ('2010-11-01')
ENDING('2010-12-01') EXCLUSIVE IN TS_PD_MQT_QRT_INV,
PART PART_2010_DEC STARTING ('2010-12-01')
ENDING('2012-01-01') EXCLUSIVE IN TS_PD_MQT_QRT_INV,
PART PART_FUTURE STARTING ('2012-01-01')
ENDING(MAXVALUE) IN TS_PD_MQT_QRT_INV);
```

To prepare MQT_REPORT1_AST for use, issue the REFRESH command and the RUNSTATS command for the table. The same REFRESH techniques can be used in the same way they were used for QRT_DWM.QRT_INV_AST_VAL.

```
REFRESH TABLE MQT_REPORT1_AST
RUNSTATS ON TABLE QRT_DWM.MQT_REPORT1_AST FOR SAMPLED DETAILED
INDEXES ALL
```

Importantly, however, the refresh operation on MQT_REPORT1_AST is rewritten by the optimizer to use the QRT_DWM.QRT_INV_AST_VAL aggregate MQT, which improves the performance of the refresh operation. In this way the REFRESH operations of these two MQTs are nested.

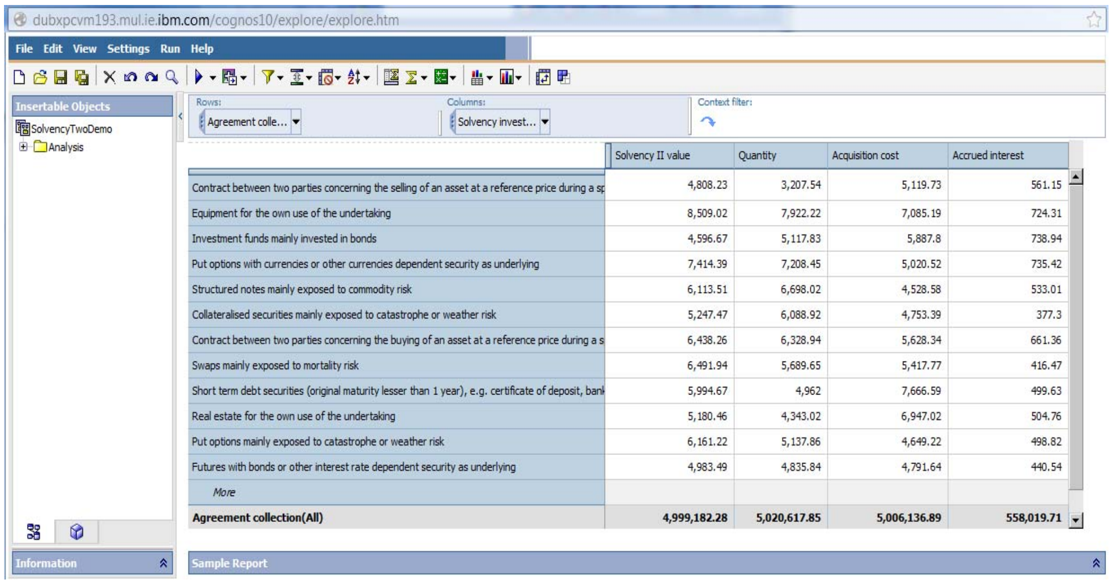
Query for report example 2: Asset valuations drill down by dimension filter

This is a query similar to that used by the second IBM Cognos report and submitted to DB2:

```
SELECT
FNC_AST.AST_NM as "Asset Name",
FNC_AST.DSC as "Asset Description",
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID AS "CIC",
SLVC_RPT_AST_CGY.TP AS "ID Code Type",
SLVC_RPT_AST_CGY.NM AS "SII Category Name",
"Asset Custody Country".CNTRY AS "Country of Custody",
"Counterparty".AGRM_REFR AS "Counterparty Issuer Name",
CR_RTG.EXT_REFR AS "External rating",
SLVC_AST_VAL_FCT.UNIT_PRC
FROM
SLVC_AST_VAL_FCT, FNC_AST, SLVC_RPT_AST_CGY, FNC_SERVICES_RL AS
"Counterparty", GEO_AREA AS "Asset Custody Country", CR_RTG ,
CDR_DT
WHERE
SLVC_AST_VAL_FCT.SLVC_RPT_AST_CGY_ID =
SLVC_RPT_AST_CGY.SLVC_RPT_AST_CGY_ID AND
SLVC_AST_VAL_FCT.ISSUR_OF_INV_ID =
"Counterparty".FNC_SERVICES_RL_ID AND
SLVC_AST_VAL_FCT.CNTRY_OF_CUSTODY_OF_AST_ID = "Asset Custody
Country".GEO_AREA_ID AND
SLVC_AST_VAL_FCT.CR_RTG_ID = CR_RTG.CR_RTG_ID AND
SLVC_AST_VAL_FCT.FNC_AST_ID = FNC_AST.FNC_AST_ID AND
SLVC_AST_VAL_FCT.TM_DIMENSION_ID = CDR_DT.CDR_DT AND
SLVC_RPT_AST_CGY.NM = 'Government bonds' AND
"Asset Custody Country".CNTRY = 'El Salvador' AND
CR_RTG.EXT_REFR = 'BB' AND
CDR_DT.CDR_YR = 2011 AND CDR_DT.CDR_MTH = 10;
```

Appendix C Sample Solvency II IBM Cognos report

The IBM Cognos report presented here simply showcases the report referenced in this paper for reference.



	Solvency II value	Quantity	Acquisition cost	Accrued interest
Contract between two parties concerning the selling of an asset at a reference price during a sp	4,808.23	3,207.54	5,119.73	561.15
Equipment for the own use of the undertaking	8,509.02	7,922.22	7,085.19	724.31
Investment funds mainly invested in bonds	4,596.67	5,117.83	5,887.8	738.94
Put options with currencies or other currencies dependent security as underlying	7,414.39	7,208.45	5,020.52	735.42
Structured notes mainly exposed to commodity risk	6,113.51	6,698.02	4,528.58	533.01
Collateralised securities mainly exposed to catastrophe or weather risk	5,247.47	6,088.92	4,753.39	377.3
Contract between two parties concerning the buying of an asset at a reference price during a s	6,438.26	6,328.94	5,628.34	661.36
Swaps mainly exposed to mortality risk	6,491.94	5,689.65	5,417.77	416.47
Short term debt securities (original maturity lesser than 1 year), e.g. certificate of deposit, bank	5,994.67	4,962	7,666.59	499.63
Real estate for the own use of the undertaking	5,180.46	4,343.02	6,947.02	504.76
Put options mainly exposed to catastrophe or weather risk	6,161.22	5,137.86	4,649.22	498.82
Futures with bonds or other interest rate dependent security as underlying	4,983.49	4,835.84	4,791.64	440.54
More				
Agreement collection(All)	**4,999,182.28**	**5,020,617.85**	**5,006,136.89**	**558,019.71**

Figure 5 Sample IBM Cognos report for sample parent report

Further reading

- Governing and managing enterprise models

  http://www.ibm.com/developerworks/rational/library/10/governingandmanagingenterprisemodels-series/index.html

- DB2 for Linux, UNIX, and Windows best practices: http://www.ibm.com/developerworks/data/bestpractices/db2luw/

- Best practices for DB2 data warehouse environments:

  http://www.ibm.com/developerworks/data/bestpractices/db2luw/#analytics

- Best practices: Temporal data management with DB2:

  http://www.ibm.com/developerworks/data/bestpractices/temporal/index.html

- Best practices: DB2 V10.1 multi-temperature data management:

  http://www.ibm.com/developerworks/data/library/long/dm-1205multitemp/index.html

- Best practices: Storage optimization with deep compression:

  http://www.ibm.com/developerworks/data/bestpractices/deepcompression/index.html

- Scoping the IBM Industry Model for banking using Enterprise Model Extender and InfoSphere Data Architect:

  http://www.ibm.com/developerworks/data/tutorials/dm-1003bankindustrymodel/

- IBM PureData System for Operational Analytics:

  http://www-01.ibm.com/software/data/puredata/analytics/operational/

Contributors

Pat G O'Sullivan

Senior Technical Staff Member, Industry Models Architecture, IBM Dublin Lab

Richard Lubell

Information Development for DB2 appliances and warehousing, IBM Dublin Lab

Paul O'Sullivan

Insurance data modeling architect, IBM Dublin Lab

Notices

Trademarks