Study Guide for the Beta Teradata Database Associate Exam – 2017 – Reference 3

Contents

Note: The numbering in this document is solely provided to separate contents and for ease of use. Please also note that internal Teradata linked information is also available in Reference documents 1 and 2 for all exam candidates.

1. Foreign Keys

Relational databases permit data values to associate across more than one entity. A **Foreign Key (FK)** value identifies table relationships.

On the next frame you will see that the employee table has three FK attributes, one of which models the relationship between employees and their departments. A second FK attributes models the relationship between employees and their jobs.

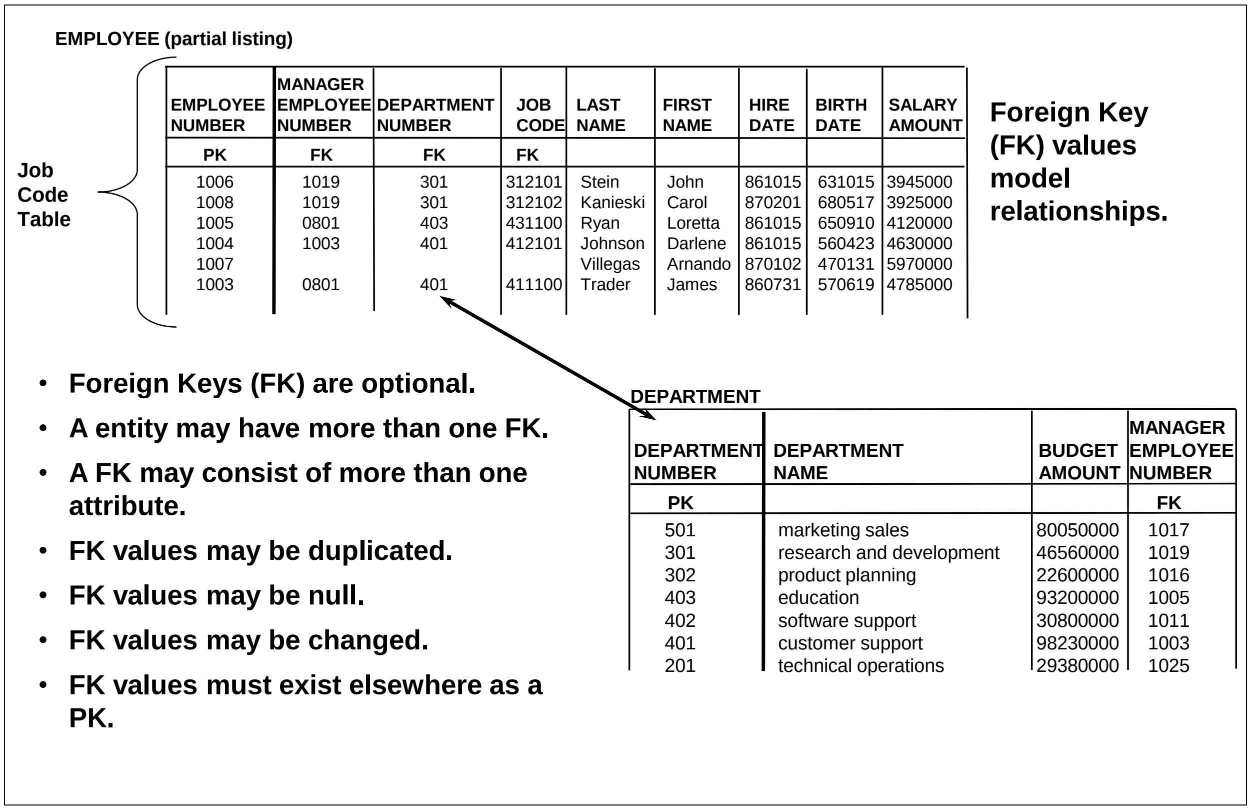
A third FK attributes is used to model the relationship between employees and each other. This is called a recursive relationship.

Foreign Key Rules

- Duplicate values are allowed in a FK attribute.
- NULLs are allowed in a FK attribute.
- Values may be changed in a FK attribute.
- Each FK must exist elsewhere as a Primary Key.

Note that Department_Number is the **Primary Key** for the DEPARTMENT entity.

Remember, these terms are not Teradata specific - they are just general relational concepts.



EMPLOYEE (partial listing)

Job Code Table

EMPLOYEE NUMBER	MANAGER EMPLOYEE NUMBER	DEPARTMENT NUMBER	JOB CODE	LAST NAME	FIRST NAME	HIRE DATE	BIRTH DATE	SALARY AMOUNT
PK	FK	FK	FK					
1006	1019	301	312101	Stein	John	861015	631015	3945000
1008	1019	301	312102	Kanieski	Carol	870201	680517	3925000
1005	0801	403	431100	Ryan	Loretta	861015	650910	4120000
1004	1003	401	412101	Johnson	Darlene	861015	560423	4630000
1007				Villegas	Arnando	870102	470131	5970000
1003	0801	401	411100	Trader	James	860731	570619	4785000

Foreign Key (FK) values model relationships.

- **Foreign Keys (FK) are optional.**
- **A entity may have more than one FK.**
- **A FK may consist of more than one attribute.**
- **FK values may be duplicated.**
- **FK values may be null.**
- **FK values may be changed.**
- **FK values must exist elsewhere as a PK.**

DEPARTMENT

DEPARTMENT NUMBER	DEPARTMENT NAME	BUDGET AMOUNT	MANAGER EMPLOYEE NUMBER
PK			FK
501	marketing sales	80050000	1017
301	research and development	46560000	1019
302	product planning	22600000	1016
403	education	93200000	1005
402	software support	30800000	1011
401	customer support	98230000	1003
201	technical operations	29380000	1025

2. What is a Database?

A **database** is a collection of permanently stored data that is used by an application or enterprise.

A database contains **logically related data**, which means that the database was created with a specific purpose in mind. A database supports **shared access** by many users. One characteristic of a database is that many people use it, often for many different purposes. A database is **protected** to control access and **managed** to retain its value and integrity.

One example of a database is payroll data that includes the names of the employees, their employee numbers, and their salary history. This database is logically related—it's all about payroll. It must have shared access, since it will be used by the payroll department to generate checks, and also by management to make decisions. This database must be protected; much of the information is confidential and must be managed to ensure the accuracy of the records.

The Teradata Database is a relational database. Relational databases are based on the relational model, which is founded on mathematical Set Theory. The relational model uses and extends many principles of **Set Theory** to provide a disciplined approach to data management. Users and applications access data in an RDBMS using industry-standard SQL statements. SQL is a set-oriented language for relational database management.

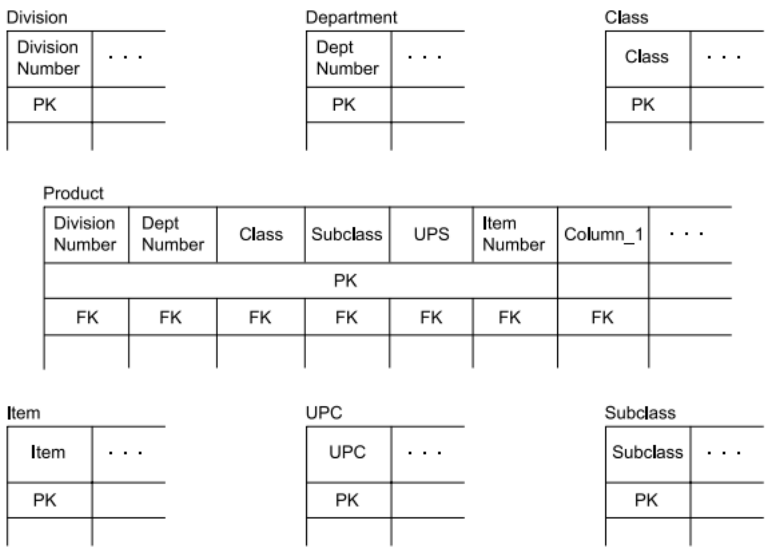
Later in this course we will provide another definition of database that is specific to the Teradata Database.

A database is a collection of permanently stored data that is:

- **Logically related** - **the data relates to other data (tables to tables).**
- **Shared** - **many users may access the data.**
- **Protected** - **access to data is controlled.**
- **Managed** - **the data has integrity and value.**

3. Dimensional Modeling, Star, and Snowflake Schemas Definition of Dimensional Modeling

According to Ralph Kimball, the creator of the dimensional modeling methodology, "DM is a logical design technique that seeks to present the data in a standard, intuitive framework that allows for high-performance access. It is inherently dimensional, and it adheres to a discipline that uses the relational model with some important restrictions. Every dimensional model is composed of one table with a multipart key, called the fact table, and a set of smaller tables called dimension tables. Each dimension table has a single-part primary key that corresponds exactly to one of the components of the multipart key in the fact table" (Kimball, 1997). The graphic indicates a simplified example of a fact table (Product) and its associated dimension tables (Division, Department, Class, Item, UPC, and Subclass).



Fact Tables and Dimension Tables The structure of a dimension model somewhat resembles that of a crude drawing of a star or snowflake (see the following graphics). In a dimensional model, fact tables always represent M:M relationships (see Many-to-Many Relationships). According to the model, a fact table should contain one or more numerical measures (the "facts" of the fact table) that occur for the combination of keys that define each tuple in the table. Dimension tables are satellites of the central fact table. They typically contain textual information that describes the attributes of the fact table.

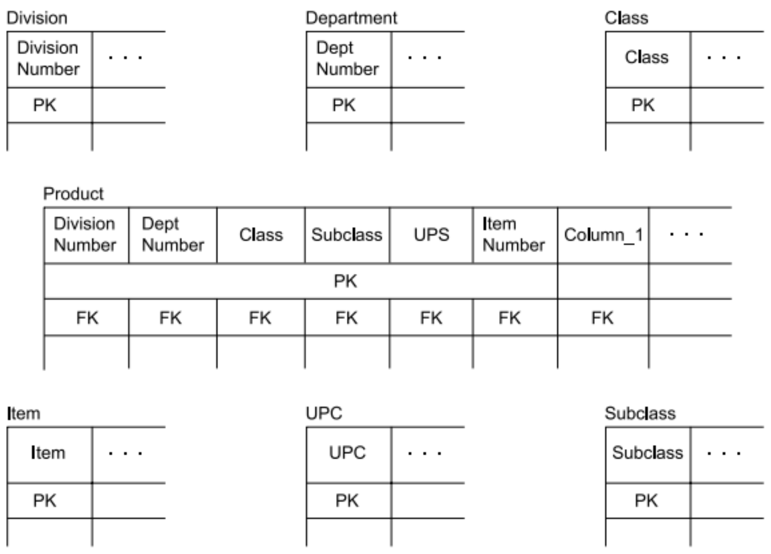
Star Schema The following graphic illustrates the classical star schema:

According to Ralph Kimball, the creator of the dimensional modeling methodology, “DM is a logical design technique that seeks to present the data in a standard, intuitive framework that allows for high-performance access. It is inherently dimensional, and it adheres to a discipline that uses the relational model with some important restrictions. Every dimensional model is composed of one table with a multipart key, called the fact table, and a set of smaller tables called dimension tables. Each dimension table has a single-part primary key that corresponds exactly to one of the components of the multipart key in the fact table” (Kimball, 1997). The graphic indicates a simplified example of a fact table (Product) and its associated dimension tables (Division, Department, Class, Item, UPC, and Subclass).

Fact Tables and Dimension Tables

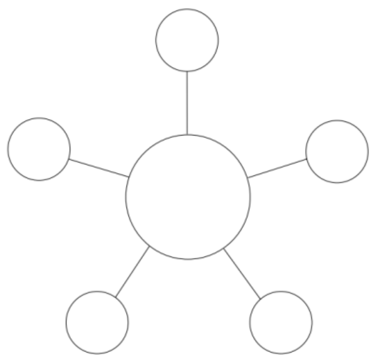
The structure of a dimension model somewhat resembles that of a crude drawing of a star or snowflake (see the graphics “Star Schema” on page 188 and “Snowflake Schema” on page 189). In a dimensional model, fact tables always represent M:M relationships (see “Many-to-Many Relationships” on page 70). According to the model, a fact table should contain one or more numerical measures (the “facts” of the fact table) that occur for the combination of keys that define each tuple in the table. Dimension tables are satellites of the central fact table. They typically contain textual information that describes the attributes of the fact table. Star Schema The following graphic illustrates the classical star schema:

According to Ralph Kimball, the creator of the dimensional modeling methodology, “DM is a logical design technique that seeks to present the data in a standard, intuitive framework that allows for high-performance access. It is inherently dimensional, and it adheres to a discipline that uses the relational model with some important restrictions. Every dimensional model is composed of one table with a multipart key, called the fact table, and a set of smaller tables called dimension tables. Each dimension table has a single-part primary key that corresponds exactly to one of the components of the multipart key in the fact table” (Kimball, 1997). The graphic indicates a simplified example of a fact table (Product) and its associated dimension tables (Division, Department, Class, Item, UPC, and Subclass).



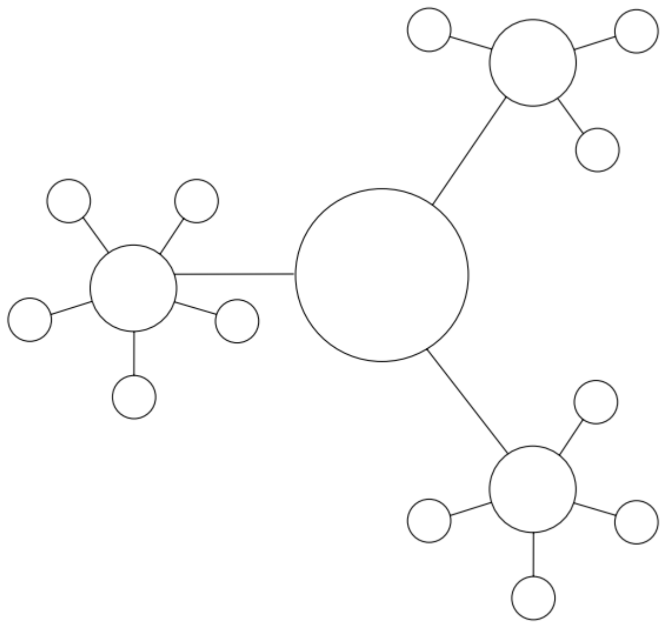
Fact Tables and Dimension Tables The structure of a dimension model somewhat resembles that of a crude drawing of a star or snowflake (see the following graphics). In a dimensional model, fact tables always represent M:M relationships (see Many-to-Many Relationships). According to the model, a fact table should contain one or more numerical measures (the “facts” of the fact table) that occur for the combination of keys that define each tuple in the table. Dimension tables are satellites of the central fact table. They typically contain textual information that describes the attributes of the fact table.

Star Schema The following graphic illustrates the classical star schema:



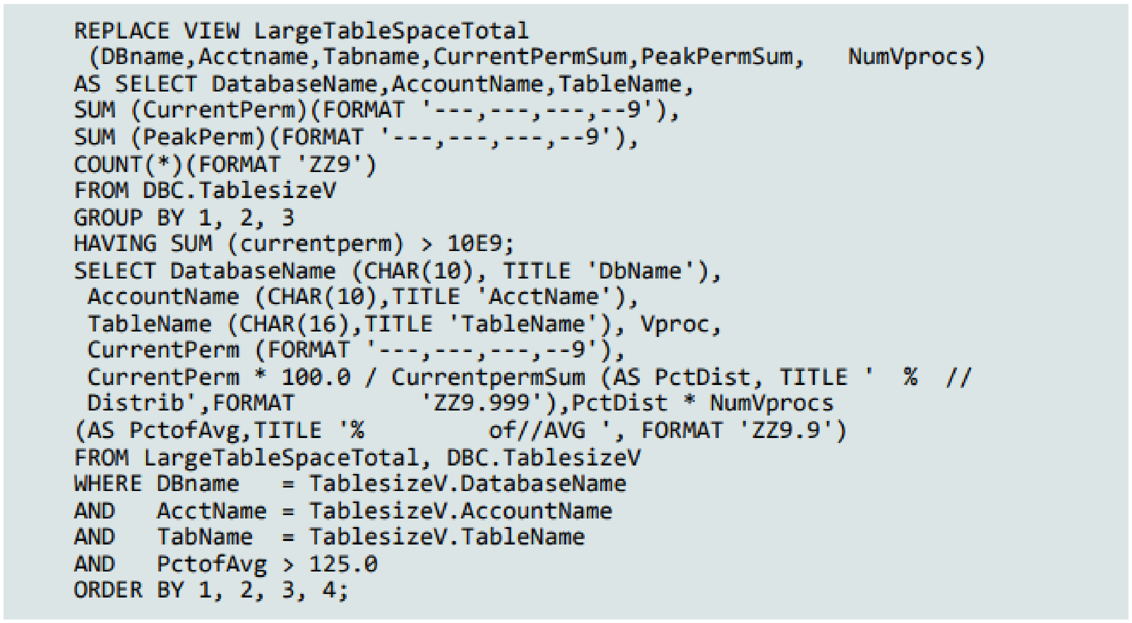
Snowflake Schema

The following graphic illustrates the classical snowflake schema



4. Prejoin with Aggregation

The following example creates a prejoin view with aggregation. Note that you can create a functionally identical object as a join index



```
REPLACE VIEW LargeTableSpaceTotal
 (DBname,Acctname,Tabname,CurrentPermSum,PeakPermSum,   NumVprocs)
AS SELECT DatabaseName,AccountName,TableName,
SUM (CurrentPerm)(FORMAT '---,---,---,--9'),
SUM (PeakPerm)(FORMAT '---,---,---,--9'),
COUNT(*)(FORMAT 'ZZ9')
FROM DBC.TablesizeV
GROUP BY 1, 2, 3
HAVING SUM (currentperm) > 10E9;
SELECT DatabaseName (CHAR(10), TITLE 'DbName'),
 AccountName (CHAR(10),TITLE 'AcctName'),
 TableName (CHAR(16),TITLE 'TableName'), Vproc,
 CurrentPerm (FORMAT '---,---,---,--9'),
 CurrentPerm * 100.0 / CurrentpermSum (AS PctDist, TITLE '  %  //
 Distrib',FORMAT         'ZZ9.999'),PctDist * NumVprocs
(AS PctofAvg,TITLE '%         of//AVG ', FORMAT 'ZZ9.9')
FROM LargeTableSpaceTotal, DBC.TablesizeV
WHERE DBname   = TablesizeV.DatabaseName
AND   AcctName = TablesizeV.AccountName
AND   TabName  = TablesizeV.TableName
AND   PctofAvg > 125.0
ORDER BY 1, 2, 3, 4;
```

Dimensional Views

A dimensional view is a virtual star or snowflake schema layered over detail data maintained in fullynormalized base tables. Not only does such a view provide high performance, but it does so without incurring the update anomalies caused by a physically denormalized database schema. The following illustration, adapted from a report by the Hurwitz Group (1999), graphs the capability of various data modeling approaches to solve ad hoc and data mining queries as a function of ease-of navigation of the database. As you can see, a dimensional view of a normalized database schema optimizes both the capability of the database to handle ad hoc queries and the navigational ease of use desired by many end users.

5. CHAPTER 6 Denormalizing the Physical Schema

Pages 99 – 114 – Database Design – TD 16.00

Overview

This chapter describes some of the common ways to denormalize the physical implementation of a fully normalized model. The chapter also briefly describing the popular technique of dimensional modeling and shows how the most useful attributes of a dimensional model can be emulated for a fully-normalized or partially-normalized physical database implementation through the careful use of dimensional views. The term denormalization describes any number of physical implementation techniques that enhance performance by reducing or eliminating the isomorphic mapping of the logical database design on the physical implementation of that design. The result of these operations is usually a violation of the design goal of making databases application-neutral. In other words, a "denormalized" database favors one or a few applications at the expense of all other possible applications. Strictly speaking, these operations are not denormalization at all. The concept of database schema normalization is logical, not physical. Logical denormalization should be avoided. Develop a fully normalized design and then, if necessary, adjust the semantic layer of your physical implementation to provide the desired performance enhancement. Finally, use views to tailor the external schema to the usability needs of users and to limit their direct access to base tables (see Denormalizing Through Views).

Denormalization, Data Marts, and Data Warehouses

The following quotation is taken from the web site of Bill Inmon, who coined the term data warehousing. It supports the position argued here: the more general the analyses undertaken on the warehouse data store, the more important the requirement that the data be normalized. The audience size issue he raises is a reflection of the diversity of analysis anticipated and the need to support any and all potential explorations of the data. "The generic data model represents a logical structuring of data. Depending on whether the modeler is building the model for a data mart or a data warehouse the data modeler will wish to engage in some degree of denormalization. Denormalization of the logical data model serves the purpose of making the data more efficient to access. In the case of a data mart, a high degree of denormalization can be practiced. In the case of a data warehouse a low degree of denormalization is in order. "The degree of denormalization that is applicable is a function of how many people are being served. The smaller the audience being served, the greater the degree of denormalization. The larger the audience being served, the lower the degree of denormalization."

Denormalization Issues

The effects of denormalization on database performance are unpredictable: as many applications can be affected negatively by denormalization as are optimized. If you decide to denormalize your database, make sure you always complete your normalized logical model first. Document the pure logical model and keep your documentation of the physical model current as well. Denormalize the implementation of the logical model only after you have thoroughly analyzed the costs and benefits, and only after you have completed a normalized logical design. Consider the following list of effects of denormalization before you decide to undertake design changes:

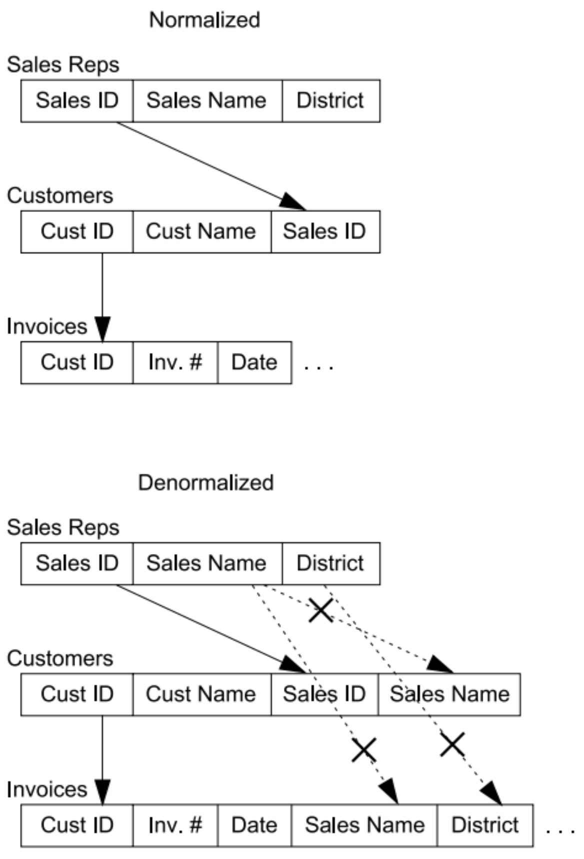
- A denormalized physical implementation can increase hardware costs.

The rows of a denormalized table are always wider than the rows of a fully normalized table. A row cannot span data blocks; therefore, there is a high probability that you will be forced to use a larger data block size for a denormalized table. The greater the degree of a table, the larger the impact on storage space. This impact can be severe in many cases.

Row width also affects the transfer rate for all I/O operations; not just for disk access, but also for transmission across the BYNET and to the requesting client.

- While denormalization benefits the applications it is specifically designed to enhance, it often decreases the performance of other applications, thus contravening the goal of maintaining application neutrality for the database.
- A corollary to this observation is the fact that a denormalized database makes it more difficult to implement new, high-performing applications unless the new applications rely on the same denormalized schema components as existing applications.
- Because of the previous two effects, denormalization often increases the cost and complexity of programming.
- Denormalization introduces update anomalies to the database. Remember that the original impetus behind normalization theory was to eliminate update anomalies.

The following graphic uses a simple database to illustrate some common problems encountered with denormalization:



Consider the denormalized schema. Notice that the name of the salesman has been duplicated in the Customers and Invoices tables in addition to being in the Sales Reps table, which is its sole location in the normalized form of the database.

This particular denormalization has all of the following impacts:

- When a sales person is reassigned to a different customer, then all accounts represented by that individual must be updated, either individually or by reloading the table with the new sales person added to the accounts in place of the former representative.

  Because the Customers, or account, table is relatively small (fewer than a million rows), either method of updating it is a minor cost in most cases.

- At the same time, because the ID and name for the sales person also appear in every Invoice transaction for the account, each transaction in the database must also be updated with the information for the new sales person. This update would probably touch many millions of rows in the Invoice table, and even a reload of the table could easily take several days to complete. This is a very costly operation from any perspective.
- Denormalized rows are always wider rows. The greater the degree of a table, the larger the impact on storage space. This impact can be severe in many cases.

  Row width also affects the transfer rate for all I/O operations; not just for disk access, but also for transmission across the BYNET and to the requesting client.

Evaluate all these factors carefully before you decide to denormalize large tables. Smaller tables can be denormalized with fewer penalties in those cases where the denormalization significantly improves the performance of frequently performed queries.

Commonly Performed Denormalizations

The following items are typical of the denormalizations that can sometimes be exploited to optimize performance:

- Repeating groups
- Prejoins
- Derived data (fields) and summary tables (column aggregations)

Alternatives to Denormalization

Teradata continues to introduce functions and facilities that permit you to achieve the performance benefits of denormalization while running under a direct physical implementation of your fully normalized logical model.

Among the available alternatives are the following:

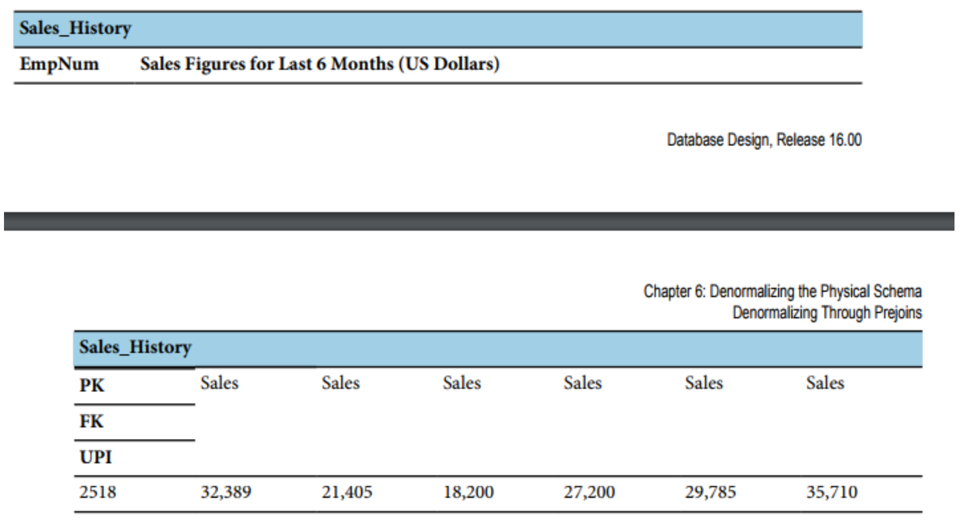
- Views
- Hash and join indexes
- Aggregate join indexes
- Global temporary and volatile tables

Denormalizing with Repeating Groups

Repeating groups are attributes of a non-1NF relation that would be converted to individual tuples in a normalized relation.

Example: Denormalizing with Repeating Groups

For example, this relation has six attributes of sales amounts, one for each of the past six months:



Sales_History	
EmpNum	Sales Figures for Last 6 Months (US Dollars)

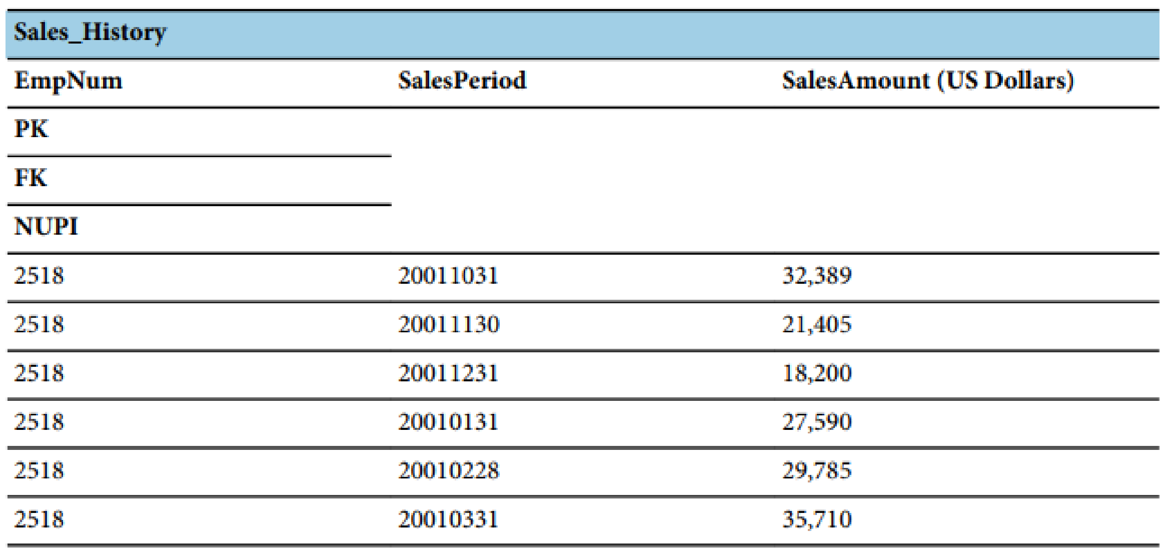
Database Design, Release 16.00

Chapter 6: Denormalizing the Physical Schema
Denormalizing Through Prejoins

Sales_History						
PK	Sales	Sales	Sales	Sales	Sales	Sales
FK						
UPI						
2518	32,389	21,405	18,200	27,200	29,785	35,710

When normalized, the Sales History relation has six tuples that correspond to the same six months of sales expressed by the denormalized relation:

Table 15: Sales_History



Sales_History		
EmpNum	SalesPeriod	SalesAmount (US Dollars)
PK		
FK		
NUPI		
2518	20011031	32,389
2518	20011130	21,405
2518	20011231	18,200
2518	20010131	27,590
2518	20010228	29,785
2518	20010331	35,710

Reasons to Denormalize With Repeating Groups

The following items are all possible reasons for denormalizing with repeating groups:

- Saves disk space
- Reduces query and load time
- Makes comparisons among values within the repeating group easier
- Many 3GLs and third party query tools work well with this structure

Reasons Not to Denormalize With Repeating Groups

The following items all mitigate the use of repeating groups:

- Makes it difficult to detect which month an attribute corresponds to

- Makes it impossible to compare periods other than months
- Changing the number of columns requires both DDL and application modifications

Denormalizing Through Prejoins

A prejoin moves frequently joined attributes to the same base relation in order to eliminate join processing. Some vendors refer to prejoins as materialized views.

Example: Denormalizing through Prejoins

The following example first indicates two normalized relations, Job and Employee, and then shows how the attributes of the minor relation Job can be carried to the parent relation Employee in order to enhance join processing:

Table 16: Job

JobCode	JobDesc
PK	**NN, ND**
UPI	
1015	Programmer
1023	Analyst

Table 17: Employee

EmpNum	EmpName	JobCode
PK, SA		**FK**
UPI		
22416	Jones	1023
30547	Smith	1015

This is the denormalized, prejoin form of the same data. This relation violates 2NF:

Employee			
EmpNum	**EmpName**	**JobCode**	**JobDesc**
PK, SA		**FK**	
UPI			
22416	Jones	1023	Analyst
30547	Smith	1015	Programmer

Reasons to Denormalize Using Prejoins

The following items are all possible reasons for denormalizing with prejoins:

- Performance can be enhanced significantly.
- The method is a good way to handle situations where there are tables having fewer rows than there are AMPs in the configuration.
- The minor entity is retained in the prejoin so anomalies are avoided and data consistency is maintained.

Reasons Not to Denormalize Using Prejoins

You can achieve the same results obtained with prejoins without denormalizing your database schema by using any of the following methods:

- Views with joins (see Denormalizing Through Views)

- Join indexes (see Denormalizing through Join Indexes)
- Global temporary tables (see Denormalizing Through Global Temporary and Volatile Tables)

Denormalizing through Join Indexes

Join indexes provide the performance benefits of prejoin tables without incurring update anomalies and without denormalizing your logical or physical database schemas.

Although join indexes create and manage prejoins and, optionally, aggregates, they do not denormalize the physical implementation of your normalized logical model because they are not a component of the fully normalized physical model.

Remember: normalization is a logical concept, not a physical concept.

Example: Denormalizing through Join Indexes

Consider the prejoin example in Denormalizing Through Prejoins. You can obtain the same performance benefits this denormalization offers without incurring any of its negative effects by creating a join index.

```
CREATE JOIN INDEX EmployeeJob
AS SELECT (JobCode, JobDescription), (EmployeeNumber, EmployeeName)
FROM Job JOIN Employee ON JobCode;
```

This join index not only eliminates the possibility for update anomalies, it also reduces storage by row compressing redundant Job table information.

Reasons to Denormalize Using Join Indexes

The following items are all reasons to use join indexes to “denormalize” your database by optimizing join and aggregate processing:

- Update anomalies are eliminated because the system handles all updates to the join index for you, ensuring the integrity of your database.
- Aggregates are also supported for join indexes and can be used to replace base summary tables. Related Information Join and Hash Indexes

Derived Data Attributes

Derived attributes are attributes that are not atomic. Their data can be derived from atomic attributes in the database. Because they are not atomic, they violate the rules of normalization.

Derived attributes fall into these basic types:

- Summary (aggregate) data
- Data that can be directly derived from other attributes

Approaches to Handling Standalone Derived Data

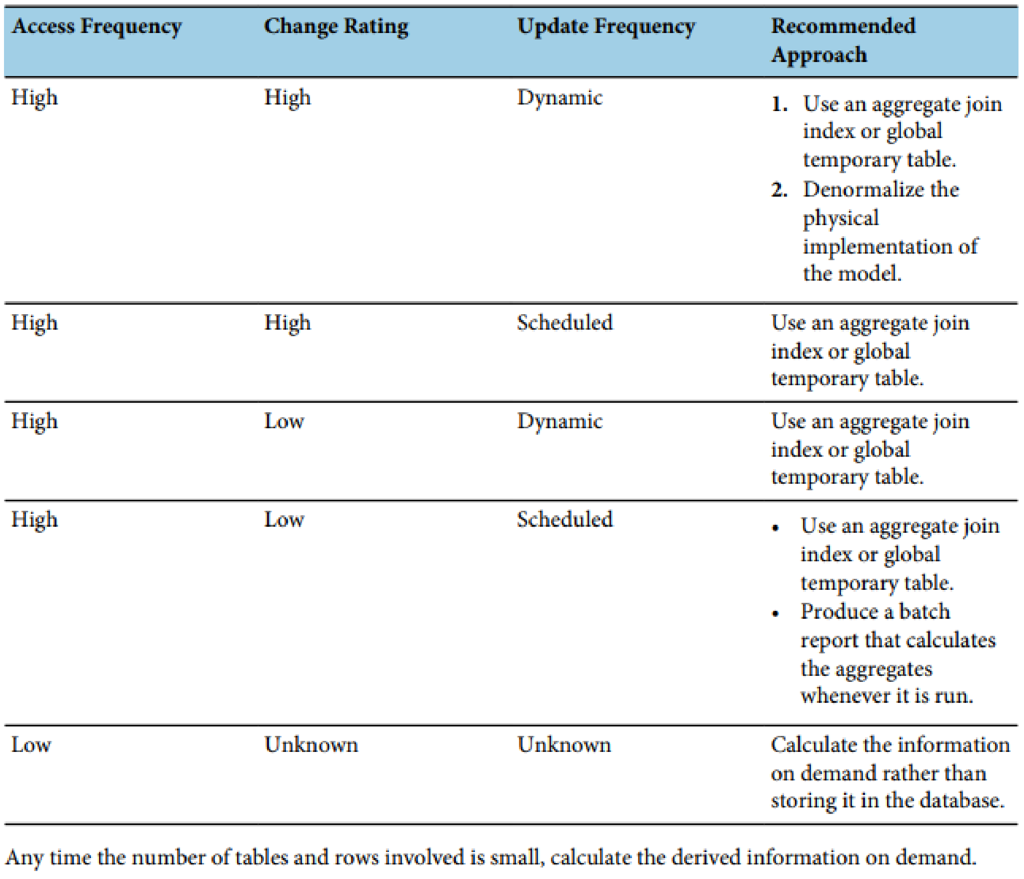
There are occasions when you might want to denormalize standalone calculations for performance reasons. Base the decision to denormalize on the following demographic information, all of which is derived through the ATM process.

- Number of tables and rows involved
- Access frequency
- Data volatility
- Data change schedule

Guidelines for Handling Standalone Derived Data

As a general rule, using an aggregate join index or a global temporary table is preferable to denormalizing the physical implementation of the fully normalized logical model.

The following table provides guidelines on handling standalone derived data attributes by denormalization. The decisions are all based on the demographics of the particular data. When more than one recommended approach is given, and one is preferable to the other, the entries are ranked in order of preference.



Access Frequency	Change Rating	Update Frequency	Recommended Approach
High	High	Dynamic	1. Use an aggregate join index or global temporary table. 2. Denormalize the physical implementation of the model.
High	High	Scheduled	Use an aggregate join index or global temporary table.
High	Low	Dynamic	Use an aggregate join index or global temporary table.
High	Low	Scheduled	• Use an aggregate join index or global temporary table. • Produce a batch report that calculates the aggregates whenever it is run.
Low	Unknown	Unknown	Calculate the information on demand rather than storing it in the database.

Any time the number of tables and rows involved is small, calculate the derived information on demand.

Any time the number of tables and rows involved is small, calculate the derived information on demand.

Reasons Not to Denormalize Using Derived Data

The following items deal with the issues of derived data without denormalizing user base data tables:

- Aggregate join index (see Aggregate Join Indexes)
- Global temporary table with derived column definitions
- View with derived column definitions

Denormalizing Through Global Temporary and Volatile Tables

Global temporary tables have a persistent stored definition just like any base table. The difference is that a global temporary table is materialized only when it is accessed by a DML request for the first time in a session and then remains materialized for the duration of the session unless explicitly dropped. At the close of the session, all rows in the table are dropped. Keep in mind that the containing database or user for a global temporary table must have a minimum of 512 bytes of PERM space per AMP in order to contain the table header. This means that the minimum amount of permanent space per global temporary table for the database is 512 bytes for each times the number of AMPs on your system.

Analogously, volatile tables can have a persistent stored definition if that definition is contained within a macro. When used in this manner, the properties of global temporary and volatile tables are largely identical in regard to persistence of the definition (see “CREATE TABLE” in SQL Data Definition Language Detailed Topics for other distinctions and differences).

Global temporary tables, like join and hash indexes, are not part of the logical model. Because of this, they can be denormalized to any degree desired, enhancing the performance of targeted applications without affecting the physically implemented normalization of the underlying database schema. The logical model is not affected, but all the benefits of physical schema denormalization are accrued.

It is important to remember that a materialized instance of a global temporary table and a volatile table are local to the session from which they are materialized or created, and only that session can access its materialized instance.

This also means that multiple sessions can simultaneously materialize instances of a global temporary table definition (or volatile tables) that are private to those sessions.

Using Global Temporary Tables and Volatile Tables to Avoid Denormalization

You can use global temporary and volatile tables to avoid the following denormalizations you might otherwise consider:

- Prejoins
- Summary tables and other derived data

This final point is important as an alternative for applications that do not require persistent storage of summary results as offered, for example, by aggregate join indexes.

Using Global Temporary and Volatile Tables to Enhance Performance

You can use global temporary tables to enhance performance in the following ways:

- Simplify application code
- Reduce spool usage
- Eliminate large numbers of joins

This final point is important as an alternative for applications that do not require persistent storage of prejoin results as offered, for example, by join indexes.

Example: Simple Denormalization for Batch Processing

The following global temporary table serves 500 different transactions that create the output it defines. These transactions collectively run over one million times per year, but 95% of them run only on a monthly batch schedule.

With the following table definition stored in the dictionary, the table itself, which violates 2NF, is materialized only when one of those batch transactions accesses it for the first time in a session:

Table 18: TemporaryBatchOutput

DeptNum	EmpNum	DeptName	LastName	FirstName
PK				
FK	FK			
…	…	…	…	…

Example: Aggregate Summary Table
The following global temporary table definition, if used infrequently and is not shared, might be an alternative to using an aggregate join index to define the equivalent summary table:

Table 19: DepartmentAggregations

DeptNum	Period	SumSalary	AvgSalary	EmpCount
PK				
FK				
NUPI				
…	…	…	…	…

Example: Prejoin
Prejoins are a form of derived relationship among tables. The following table definition, if used infrequently, might be an alternative to using a join index to define the equivalent prejoin table.

This particular table saves the cost of having to join the Order, Location, and Customer tables:

Table 20: OrderCustomer

OrdNum	CustNum	OrdCost
PK	FK	
FK	NUPI	
…	…	…

Denormalizing Through Views
You cannot denormalize a physical database using views, though views can be used to provide the appearance of denormalizing base relations without actually implementing the apparent denormalizations they simulate.

Denormalized views can be a particularly useful solution to the conflicting goals of dimensional and normalized models because it is possible to maintain a fully-normalized physical database while at the same time presenting a virtual multidimensional database to users through the use of a semantic layer based on dimensional views (see "Dimensional Views" below and Denormalized Physical Schemas and Ambiguity).

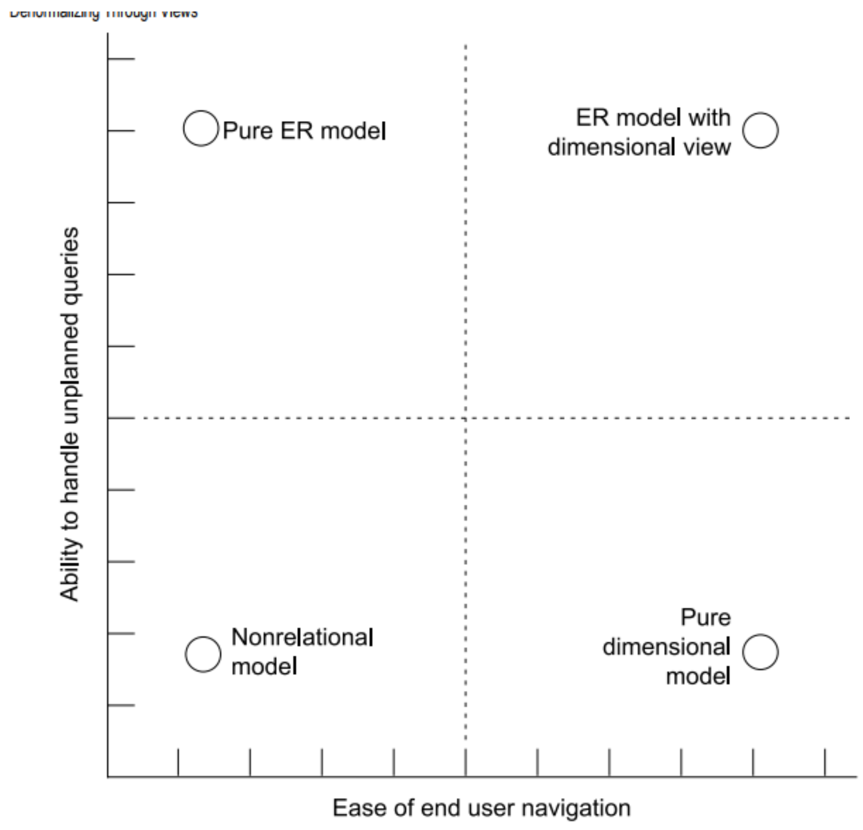
Prejoin with Aggregation
The following example creates a prejoin view with aggregation. Note that you can create a functionally identical object as a join index.

```
REPLACE VIEW LargeTableSpaceTotal
 (DBname,Acctname,Tabname,CurrentPermSum,PeakPermSum,   NumVprocs)
AS SELECT DatabaseName,AccountName,TableName,
SUM (CurrentPerm)(FORMAT '---,---,---,--9'),
SUM (PeakPerm)(FORMAT '---,---,---,--9'),
COUNT(*)(FORMAT 'ZZ9')
FROM DBC.TablesizeV
GROUP BY 1, 2, 3
HAVING SUM (currentperm) > 10E9;
SELECT DatabaseName (CHAR(10), TITLE 'DbName'),
 AccountName (CHAR(10),TITLE 'AcctName'),
 TableName (CHAR(16),TITLE 'TableName'), Vproc,
 CurrentPerm (FORMAT '---,---,---,--9'),
 CurrentPerm * 100.0 / CurrentpermSum (AS PctDist, TITLE '  %  //
 Distrib',FORMAT         'ZZ9.999'),PctDist * NumVprocs
(AS PctofAvg,TITLE '%         of//AVG ', FORMAT 'ZZ9.9')
FROM LargeTableSpaceTotal, DBC.TablesizeV
WHERE DBname   = TablesizeV.DatabaseName
AND   AcctName = TablesizeV.AccountName
AND   TabName  = TablesizeV.TableName
AND   PctofAvg > 125.0
ORDER BY 1, 2, 3, 4;
```

Dimensional Views

A dimensional view is a virtual star or snowflake schema layered over detail data maintained in fullynormalized base tables. Not only does such a view provide high performance, but it does so without incurring the update anomalies caused by a physically denormalized database schema.

The following illustration, adapted from a report by the Hurwitz Group (1999), graphs the capability of various data modeling approaches to solve ad hoc and data mining queries as a function of ease-of navigation of the database. As you can see, a dimensional view of a normalized database schema optimizes both the capability of the database to handle ad hoc queries and the navigational ease of use desired by many end users.



Many third party reporting and query tools are designed to access data that has been configured in a star schema (see Dimensional Modeling, Star, and Snowflake Schemas). Dimensional views combine the strengths of the E-R and dimensional models by providing the interface for which these reporting and query tools are optimized.

Access to the data through standard applications, or by unsophisticated end users, can also be accomplished by means of dimensional views. More sophisticated applications, such as ad hoc tactical and strategic queries and data mining explorations can analyze the normalized data either directly or by means of views on the normalized database.

The following procedure outlines a hybrid methodology for developing dimensional views in the context of traditional database design techniques:

1. Develop parallel logical database models. It makes no difference which model is developed first, nor does it make a difference if the two models are developed in parallel. The order of steps in the following procedure is arbitrary:
   - Develop an enterprise E-R model.
   - Develop an enterprise DM model.
2. Develop an enterprise physical model based on the E-R model developed in step 1.
3. Implement the physical model designed in step 2.
4. Implement dimensional views to emulate the enterprise DM model developed in step 1 as desired

Several Teradata customers use this hybrid methodology to provide a high-performing, flexible design that benefits data manipulation while simultaneously being user- and third-party-tool friendly.

Martyn (2004) examines dimensional views from a research-oriented perspective and concludes that dimensional views are an optimal means for overcoming the objections to normalized databases visa-a-vis DM models.

Dimensional Modeling, Star, and Snowflake Schemas

Definition of Dimensional Modeling

According to Ralph Kimball, the creator of the dimensional modeling methodology, “DM is a logical design technique that seeks to present the data in a standard, intuitive framework that allows for high-performance access. It is inherently dimensional, and it adheres to a discipline that uses the relational model with some important restrictions. Every dimensional model is composed of one table with a multipart key, called the fact table, and a set of smaller tables called dimension tables. Each dimension table has a single-part primary key that corresponds exactly to one of the components of the multipart key in the fact table” (Kimball, 1997).

The graphic indicates a simplified example of a fact table (Product) and its associated dimension tables (Division, Department, Class, Item, UPC, and Subclass).

Division

Division Number	. . .
PK	

Department

Dept Number	. . .
PK	

Class

Class	. . .
PK	

Product

Division Number	Dept Number	Class	Subclass	UPS	Item Number	Column_1	. . .
PK							
FK	FK	FK	FK	FK	FK	FK	

Item

Item	. . .
PK	

UPC

UPC	. . .
PK	

Subclass

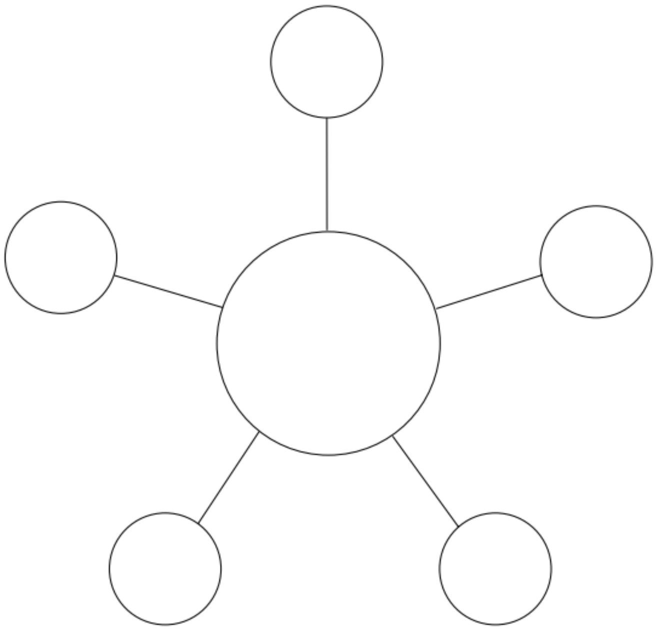
Subclass	. . .
PK	

Fact Tables and Dimension Tables

The structure of a dimension model somewhat resembles that of a crude drawing of a star or snowflake (see the following graphics).

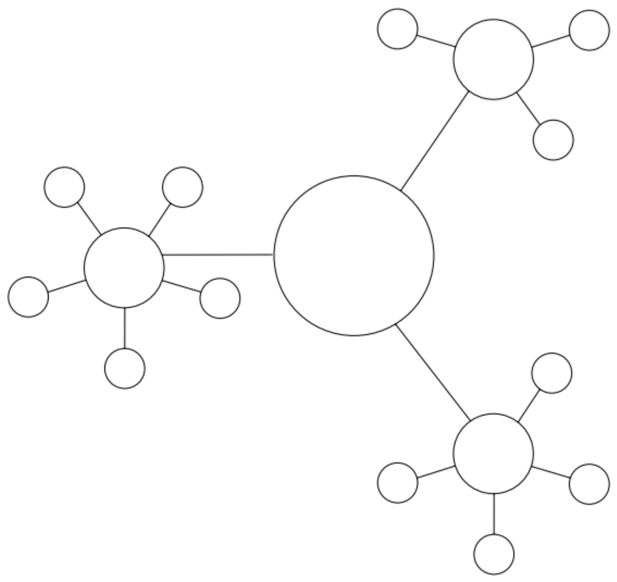
In a dimensional model, fact tables always represent M:M relationships (see Many-to-Many Relationships). According to the model, a fact table should contain one or more numerical measures (the “facts” of the fact table) that occur for the combination of keys that define each tuple in the table.

Dimension tables are satellites of the central fact table. They typically contain textual information that describes the attributes of the fact table.



Snowflake Schema

The following graphic illustrates the classical snowflake schema:



The E-R Model Versus the DM Model

While a table in a normalized E-R-derived database represents an entity and its relevant atomic descriptors, tables in a DM-derived database represent dimensions of the business rules of the enterprise. The meaning of business rule here is somewhat different from that used by writers in the business rules community, where the term applies to the declarative domain, range, uniqueness, referential, and other constraints you can specify in the database.

While advocates of implementing a normalized physical schema emphasize the flexibility of the model for answering previously undefined questions, DM advocates emphasize its usability because the tables in a DM database are configured in a structure more akin to their business use.

The E-R model for an enterprise is always more complex than a DM model for the same enterprise. While the E-R model might have hundreds of individual relations, the comparable DM model typically has dozens of star join schemas. The dimension tables of the typical DM-derived database are often shared to some extent among the various fact tables in the database.

6. NoPI Tables, Column-Partitioned NoPI Tables, and Column-Partitioned NoPI Join Indexes

A NoPI object is a table or join index that does not have a primary index or a primary AMP index and always has a table kind of MULTISET.

The basic types of NoPI objects are:

- Nonpartitioned NoPI tables
- Column-partitioned NoPI tables and NoPI join indexes (these may also have row partitioning)

The chief purpose of nonpartitioned NoPI tables is as staging tables. FastLoad can efficiently load data into empty nonpartitioned NoPI staging tables because NoPI tables do not have the overhead of row distribution among the AMPs and sorting the rows on the AMPs by rowhash.

Nonpartitioned NoPI tables are also critical to support Extended MultiLoad Protocol (MLOADX). A nonpartitioned NoPI staging table is used for each MLOADX job.

The optimal method of loading rows into any type of column-partitioned table from an external client is to use FastLoad to insert the rows into a staging table, then use an INSERT … SELECT request to load the rows from the source staging table into the column-partitioned target table.

You can also use Teradata Parallel Data Pump array INSERT operations to load rows into a column-partitioned table.

Global temporary trace tables are, strictly speaking, also a type of NoPI table because they do not have a primary index, though they are generally not treated as NoPI tables.

Because there is no primary index or a primary AMP index for the rows of a NoPI table, its rows are not hashed to an AMP based on their primary index or a primary AMP index value. Instead, Teradata Database either hashes on the Query ID for a row, or it uses a different algorithm to assign the row to its home AMP (see Hash-Based Table Partitioning to AMPs).

Teradata Database then generates a RowID for each row in a NoPI table by using a hash bucket that an AMP owns (see Indexes and Partitioning and Hash-Based Table Partitioning to AMPs). This strategy makes fallback and index maintenance very similar to their maintenance on a PI table.

Global temporary tables and volatile tables can be defined as nonpartitioned NoPI tables but not as partitioned NoPI tables. Column-partitioned tables and column-partitioned join indexes can also be defined without a primary index but can have a primary index or a primary AMP index. See Column-Partitioned NoPI Tables and Join Indexes for details about column partitioning and NoPI tables and join indexes.

INSERT… SELECT into NoPI and Column-Partitioned NoPI Tables

When the target table of an INSERT … SELECT request is a NoPI table, Teradata Database inserts the data from the source table locally into the target table, whether it comes directly from the source table or from an intermediate spool. This is very efficient because it avoids a redistribution and sort. However, if the source table or the resulting spool is skewed, the target table can also be skewed. In this case, you can specify a HASH BY clause to redistribute the data from the source before Teradata Database executes the local copy operation.

Consider using hash expressions that provide good distribution and, if appropriate, improve the effectiveness of autocompression for the insertion of rows into the target table. Alternatively, you can specify HASH BY RANDOM to achieve good distribution if there is not a clear choice for the expressions to hash on.

When inserting into a column-partitioned NoPI table, also consider specifying a LOCAL ORDER BY clause with the INSERT … SELECT request to improve the effectiveness of autocompression.

Uses for Nonpartitioned NoPI Tables

Nonpartitioned NoPI tables are particularly useful as staging tables for bulk data loads. When a table has no primary index or a primary AMP index, its rows can be dispatched to any given AMP arbitrarily and the rows do not need to be sorted, so the system can load data into a staging table faster and more efficiently using FastLoad or Teradata Parallel Data Pump array INSERT operations. You can only use FastLoad to load rows into a NoPI table when it is unpopulated, not partitioned, and there are no USIs.

You must use Teradata Parallel Data Pump array INSERT operations to load rows into NoPI tables that are already populated. If a NoPI table is defined with a USI, Teradata Database checks for an already existing row with the same value for the USI column (to prevent duplicate rows) when you use Teradata Parallel Data Pump array INSERT operations to insert rows into it.

By storing bulk loaded rows on any arbitrary AMP, the performance impact for both CPU and I/O is reduced significantly. After having been received by Teradata Database all of the rows can be appended to anonpartitioned or column-partitioned NoPI table without needing to be redistributed to their hash-owning AMPs.

Because there is no requirement for such tables to maintain their rows in any particular order, the system need not sort them. The performance advantage realized from NoPI tables is achieved optimally for applications that load data into a staging table, which must first undergo a conversion to some other form, and then be redistributed before they are stored in a secondary staging table or the target table.

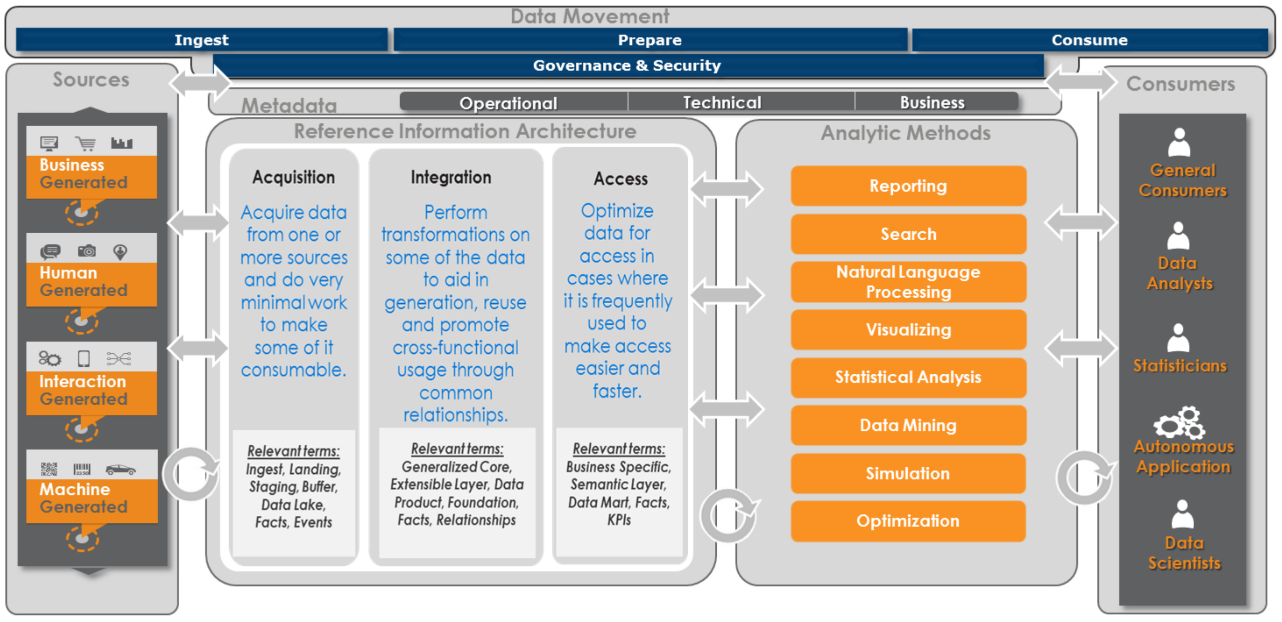
Using a nonpartitioned NoPI table as a staging table for such applications avoids the row redistribution and sorting required for primary-indexed staging tables. Another advantage of nonpartitioned NoPI tables is that you can quickly load data into them and be finished with the acquisition phase of the utility operation, which frees client resources for other applications.

Both NoPI and column-partitioned NoPI tables are also useful as so-called sandbox tables when an appropriate primary index has not yet been defined for the primary-indexed table they will eventually populate. This use of a NoPI table enables you to experiment with several different primary index possibilities before deciding on the most optimal choice for your particular application workloads.

7. Teradata Reference Information Architecture

Data Layers

The RIA can be further clarified via a series of data layers. The data layers within the Reference Information Architecture are depicted below. Each data layer has a specific purpose. There are many terms used for these layers, some of which are ambiguous. A few of the most common terms are provided.



Data Movement
Ingest
Prepare
Consume
Governance & Security
Sources
Metadata
Operational
Technical
Business
Consumers
Reference Information Architecture
Analytic Methods
Business Generated
Human Generated
Interaction Generated
Machine Generated
Acquisition
Acquire data from one or more sources and do very minimal work to make some of it consumable.
Relevant terms: Ingest, Landing, Staging, Buffer, Data Lake, Facts, Events
Integration
Perform transformations on some of the data to aid in generation, reuse and promote cross-functional usage through common relationships.
Relevant terms: Generalized Core, Extensible Layer, Data Product, Foundation, Facts, Relationships
Access
Optimize data for access in cases where it is frequently used to make access easier and faster.
Relevant terms: Business Specific, Semantic Layer, Data Mart, Facts, KPIs
Reporting
Search
Natural Language Processing
Visualizing
Statistical Analysis
Data Mining
Simulation
Optimization
General Consumers
Data Analysts
Statisticians
Autonomous Application
Data Scientists

Figure 1 – Analytic Ecosystem – Data Layers

The reference information architecture is broken down into 3 major data layers, i.e. acquisition, integration and access. These are logical layers and do not infer specific technologies or vendor products, simply the capabilities they enable. The acquisition layer focuses on acquiring raw data from source systems and performing basic standardizations.

The integration layer is primarily responsible for integrating from multiple systems both normalized and potential de-normalized. It also may create common metrics and summaries which are widely used within an organization.

The access layer's primary responsibility is to provide user-friendly data access which will perform according to service level agreements (SLAs). Each of the data layers are can be broken down further.

8. Teradata Reference Information Architecture

Data Tiers

Data layers are further broken down into data tiers. Data tiers provide a finer grain of how data progresses through the data layers.

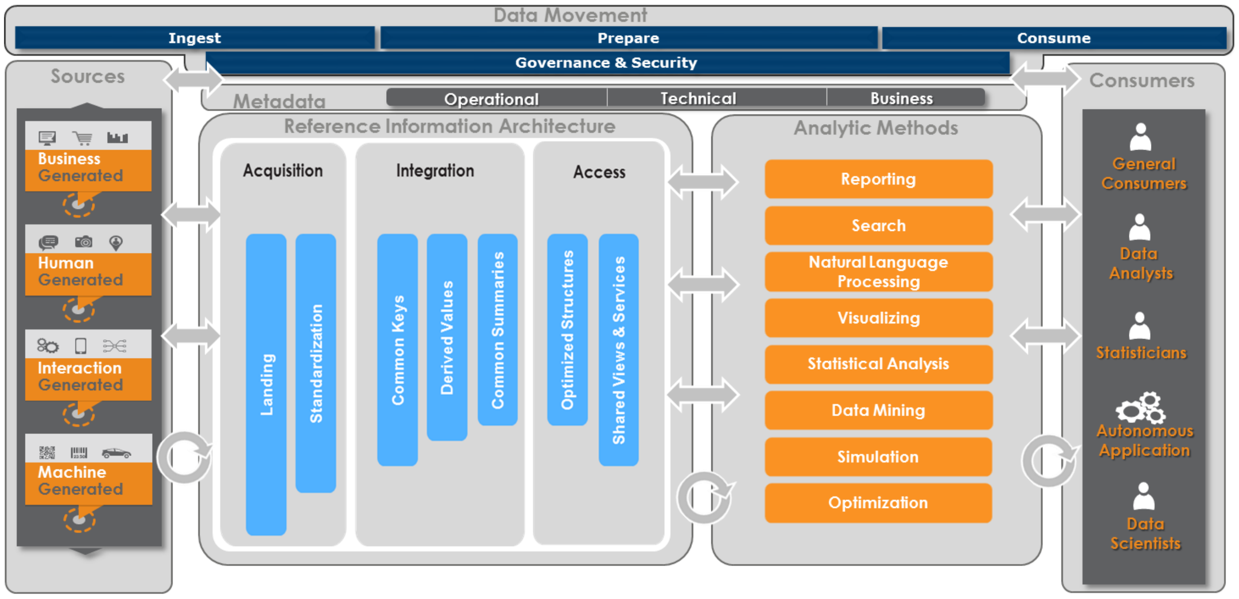


Figure 2 – Analytic Ecosystem – Data Tiers

Not all the data tiers are used for every feed coming in from a data source. For example there may not be a business need for derived values. Or if performance is not an issue, common summaries and/or optimized structures data tiers would not be needed. However, if these types of needs surface, these data tiers are specifically designed to address them. Below is a list of definitions for each of the data tiers within the data layers.

Layer	Data Tiers	Definition
Acquisition	Landing	This tier is the initial repository of data within data architecture. It is used to house data in its raw, unprocessed format at the lowest level of granularity enabling reconstitution of any view, aggregation or modification of data processed throughout the data architecture.
	Standardization	This tier processes data ingested into the Landing tier into a “consumable” format. The extensible, reusable format is necessary for further processing. Very light standardization occurs, such as values that are made to be consistent (e.g. gender codes, medical codes, etc. are standardized). Standardization may include optimization of the physical layout, e.g. indexing, (re-)partitioning, compression.

Integration	**Common Keys**	This tier standardizes heavily reused keys that are the basis for connecting subject areas. We use the term common to express a highest order of denomination – such as "customer_id", "product_id", "order_id", "session_id" and the like. This is not about defining every primary and foreign key for all data. In general, we'd expect < 20 common keys for a large enterprise. Point is, at this stage, invest in common keys to enable connecting the dots across subject areas, but don't go too far in standardizing every key such that it slows the overall deployment to a grinding halt.
	Derived Values	This tier is where enterprise, governed Key Performance Indicators (KPIs) are defined and automated.
	Common Summaries	This tier is where summarization occurs, not just for performance, but for consistency (e.g. Total Revenue may be a complex calculation involving SKU level roll ups and tax implications, less returns)
Consumption	**Optimized Structures**	This tier in the Access Layer is about performance, where a variety of schemes (such as indexing and partitioning) are used to optimize resource utilization and query speed. This is common use for autonomous applications.
	Shared Views & Services	This tier in the Access Layer is about ease of use, whereby techniques such as materialized views and metadata services are created to assist users in navigating and consuming the data.

Figure 3 – Data Tiers Definitions

Data tiers can be further broken down into sub-tiers. As processing patterns emerge, sub-tiers help to clarify repeatable steps and improve manageability. More details on sub-tiers are provided in the section "Bringing the RIA Concepts Together".

9. Increase Insights while Reducing Costs and Complexity with Teradata's Unified Data Architecture

http://assets.teradata.com/resourceCenter/downloads/Brochures/EB6732.pdf?processed=1,

10.Teradata IntelliCloud

http://www.teradata.com/products-and-services/intellicloud/

11. Hybrid Cloud Solutions – Analytics Should be Everywhere

http://www.teradata.com/Solutions-and-Industries/hybrid-cloud-solutions

12. What is a Hybrid Cloud?

http://www.teradata.com/Resources/Videos/What-is-a-Hybrid-Cloud/

13. Data Marts

Data Marts

A data mart is generally a relatively small application- or function-specific subset of the data warehouse database created to optimize application performance for a narrowly defined user population.

Data marts are often categorized into three different types:

•Independent data marts

Independent data marts are isolated entities, entirely separate from the enterprise data warehouse. Their data derives from independent sources and they should be viewed as data pirates in the context of the enterprise data warehouse because their independent inputs, which are entirely separate from the enterprise data warehouse, have a high likelihood of producing data that does not match that of the warehouse.
These independent data marts are sometimes referred to as , and Teradata strongly discourages their use.

•Dependent data marts

Dependent data marts are derived from the enterprise data warehouse. Depending on how a dependent data mart is configured, it might or might not be useful.
The recommended process uses only data that is derived from the enterprise data warehouse data store and also permits its users to have full access to the enterprise data store when the need to investigate more enterprise-wide issues arises.
The less useful forms of dependent data mart are sometimes referred to as .

•Logical data marts

The logical mart is a form of dependent data mart that is constructed virtually from the physical data warehouse. Data is presented to users of the mart using a series of SQL views that make it appear that a physical data mart underlies the data available for analysis.

Independent Data Marts

An independent data mart has neither a relationship with the enterprise data warehouse nor with any other data mart. Its data is input separately and its analyses are conducted autonomously. Because the data is not derived from the central warehouse, the likelihood that it does not match the enterprise data is high. Which version of reality is correct? How can a user know?

Teradata often discourages the use of independent data marts, sometimes referred to disparagingly as "data basements." Implementation of independent data marts is antithetical to the motivation for building a data warehouse in the first place: to have a consistent, centralized store of enterprise data that can be analyzed in a multiplicity of ways by multiple users with different interests seeking widely varying information.

A data basement is a collection of independent data marts. Suppose you have parts that you decide to store in your basement. There is no particular rhyme or reason to what part is stored or where it is stored other than convenience. Continuing the analogy, what is stored in the basement depends on what any family member decides needs to be stored there. If you need to locate a part that you think might have been stored in the basement, you ask everybody in the family if they have seen it recently and then you make your search based on their recollections. If you need to visit more than one basement to find your parts, it is unlikely they will be compatible even if you are able to find them.

This method of storing data is essentially the same as the mix of paper databases and mixed hierarchical and relational online databases spread among multiple departments that supports many businesses today. It is the sort of situation that businesses generally want to escape, not automate.

Dependent Data Marts

If you need to develop one or more physical data marts in the Teradata environment, you should strongly consider configuring them as dependent data marts. Dependent data marts can be built in one of two ways: either where a user can access both the data mart and the complete data warehouse, depending on need, or where access is limited exclusively to the data mart. The latter approach is not optimal and the type of data mart it produces is sometimes referred to as a data junkyard.

In the data junkyard, all data begins with a common source (in this analogy, "cars"), but they are scrapped, rearranged, and generally junked to get some common parts that the yard operator believes are useful to his customers. The parts collection in the junkyard relates more to what has been useful in the past: previous supply and demand determines what the user can access.

Continuing the analogy, you, as a user, visit the junkyard and search through the various wrecks you encounter in hopes of finding the part you need. To find your part (to answer your question), you will probably need to scavenge through several different junkyards.

The approach results in a decision support environment molded, and compromised, from a specific, well known set of questions and responses rather than around your ever-changing business needs.

Logical Data Marts

Perhaps the ideal approach to incorporating the data mart concept into your data warehouse is to construct one or more logical, or virtual, data marts. By using a system of carefully constructed views on the detail data of the warehouse, you can design multiple user- or department-specific virtual data marts that provide the same sort of highly tailored information a physical data mart would without the need for massive data loads, cleansing, and other necessary transformations.

http://info.teradata.com/htmlpubs/DB_TTU_16_00/index.html#page/Database_Management%2FB035-1094-160K%2Fwwv1472240583884.html%23

14. Overview of the Data Warehouse

Overview of the Data Warehouse

Initially, the data warehouse was a historical database, enterprise-wide and centralized, containing data derived from an operational database.

The data in the data warehouse was:

• Subject-oriented

• Integrated

• Usually identified by a timestamp

• Nonvolatile, that is, nothing was added or removed

Rows in the tables supporting the operational database were loaded into the data warehouse (the historical database) after they exceeded some well-defined date.

Data could be queried, but the responses returned only reflected historical information. In this sense, a data warehouse was initially static, and even if a historical data warehouse contained data that was being updated, it would still not be an active data warehouse.

http://info.teradata.com/htmlpubs/DB_TTU_16_00/index.html#page/General_Reference/B035-1091-160K/kxc1472241424495.html

15. Teradata Active Solutions

In an active data warehouse, Teradata provides both strategic intelligence and operational intelligence.

- Strategic intelligence entails delivering intelligence through tools and utilities and query mechanisms that support strategic decision-making.
  This includes, for example, providing users with simple as well as complex reports throughout the day which indicate the business trends that have occurred and that are occurring, which show why such trends occurred, and which predict if they will continue to occur.
- Operational intelligence entails delivering intelligence through tools and utilities and query mechanisms that support front-line or operational decision-making.
  This includes, for example, ensuring aggressive Service Level Goals (SLGs) with respect to high performance, data freshness, and system availability.

Active Load

Teradata is able to load data actively and in a non-disruptive manner and, at the same time, process other workloads.

Teradata delivers Active Load through methods that support continuous data loading. These include streaming from a queue, more frequent batch updates, and moving changed data from another database platform to Teradata.

These methods exercise such Teradata Database features as queue tables and triggers, and use FastLoad, MultiLoad, TPump, standalone utilities, and Teradata Parallel Transporter.

Teradata can effectively manage a complex workload environment on a “single version of the business.”

Active Access

Teradata is able to access analytical intelligence quickly and consistently in support of operational business processes.

But the benefit of Active Access entails more than just speeding up user and customer requests. Active Access provides intelligence for operational and customer interactions consistently.

Active Access queries, also referred to as tactical queries, support tactical decision-making at the front-line. Such queries can be informational, such as simply retrieving a customer record or transaction, or they may include complex analytics.

Active Events

Teradata is able to detect a business event automatically, apply business rules against current and historical data, and initiate operational actions when appropriate. This enables enterprises to reduce the latency between the identification of an event and taking action with respect to it. Active Events entails more than event detection.

When notified of something important, Teradata presents users with recommendations for appropriate action. The analysis done for users may prescribe the best course of action or give them alternatives from which to choose.

Active Workload Management

Teradata is able to manage mixed workloads dynamically and to optimize system resource utilization to meet business goals.

Teradata Active System Management (TASM) is a portfolio of products that enables real-time system management.

TASM assists the database administrator in analyzing and establishing workloads and resource allocation to meet business needs. TASM facilitates monitoring workload requests to ensure that resources are used efficiently and that dynamic workloads are prioritized automatically.

TASM also provides state-of-the-art techniques to visualize the current operational environment and to analyze long-term trends. TASM enables database administrators to set SLGs, to monitor adherence to them, and to take any necessary steps to reallocate resources to meet business objectives.

Active Enterprise Integration

Teradata is able to integrate itself into enterprise business and technical architectures, especially those that support business users, partners, and customers. This simplifies the task of coordinating enterprise applications and business processes.

For example, a Teradata event, generated from a database trigger, calls a stored procedure, which inserts a row into a queue table and publishes a message via the Teradata JMS Provider. The message is delivered to a JMS queue on a WebLogic, SAP NetWeaver, or other JMScompatible application server. SAP Customer Relationship Management receives the message, notifies the user, and takes an action.

Active Availability

Teradata is able to meet business objectives for its own availability. Moreover, it assists customers in identifying application-specific availability, recoverability, and performance requirements based on the impact of enterprise downtime. Teradata can also recommend strategies for achieving system availability goals.

16. Recovering a Specific AMP

When restoring a nonfallback table with after-image journaling to a specific AMP after a disk failure, use a ROLLFORWARD statement followed by a BUILD statement of the nonfallback table.

If the nonfallback table has unique indexes, rollforward time may be improved by using the PRIMARY DATA option. This option instructs the rollforward process to skip unique secondary index change images in the journal. These indexes would be invalid from the specific-AMP restore operation, therefore the PRIMARY DATA option might save a significant amount of I/O. Revalidate the indexes following the rollforward with the BUILD statement.

Copying Objects

Teradata ARC can copy (or restore) an object to a different Teradata Database environment. Use the COPY statement to:

- Replace an object in a target database
- Create an object in a target database
- Move an archived file to a different Teradata Database other than the one from which the archive was made
- Move an archived file to the same Teradata Database from which the archive was made

Note: The ability to copy all objects as individual objects is a feature of TTU 13.00.00 and later. Triggers cannot be copied. For a complete list of objects supported by Teradata ARC, see “Appendix A Database Objects” on page 271.

Copy vs. Restore

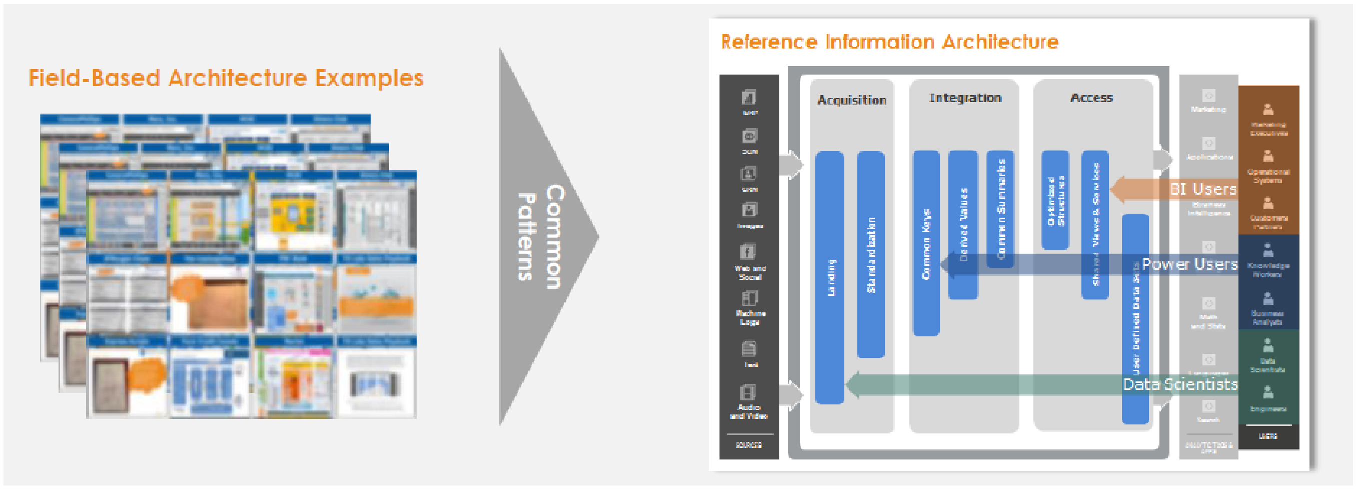
The difference between copy and restore depends on the kind of operation being performed:

- A restore operation moves data from archived files back to the same Teradata Database from which it was archived or moves data to a different Teradata Database so long as database DBC is already restored.
- A copy operation moves data from an archived file to any existing Teradata Database and creates a new object if one does not already exist on that target database. (The target object does not have to exist, however, the database must already exist.)

  When selected partitions are copied, the table must exist and be a table that was previously copied as a full-table copy

17. Analytic Architecture Modernization

Reference Information Architecture
A Familiar (But Modernized) Framework



Use the Reference Information Architecture to:

- **Help make sense of big data trends and concepts in the marketplace**
- **Architect modern day business solutions using big data sources**
- **Position new technologies as part of an analytic ecosystem**

18. Teradata Aster AppCenter

http://www.teradata.com/products-and-services/appcenter

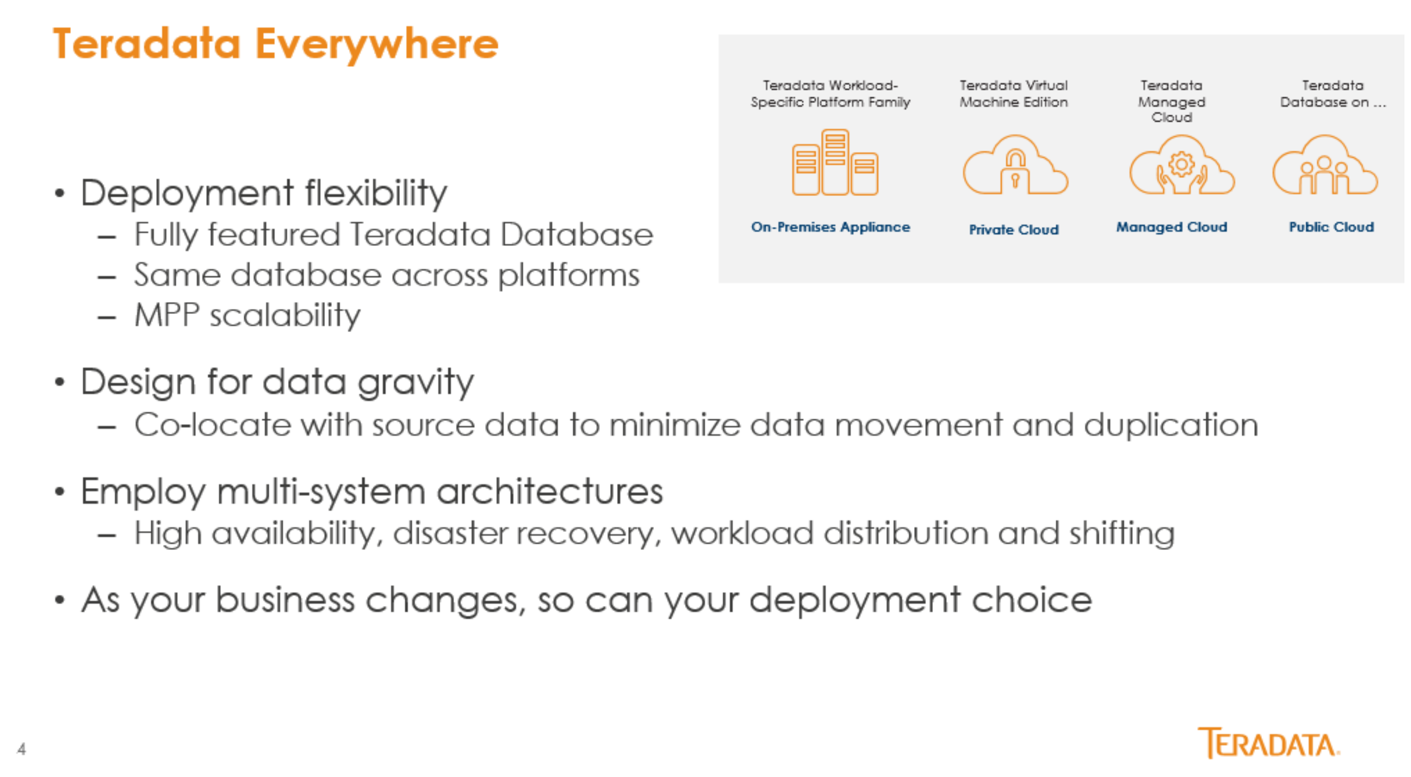
19. Reduce Big Data Complexity to Bring Better Visibility to Your Business

http://www.teradata.com/Solutions-and-Industries/unified-data-architecture

20. Teradata Everywhere Sales Introduction – 55751

Direct link to the course:
https://university.teradata.com/learning/user/deeplink_redirect.jsp?linkId=ITEM_DETAILS&componentID=55751&componentTypeID=RECORDED-WEBCAST&revisionDate=1472851800000

Slides 4 and 5, 02:13 to 05:04

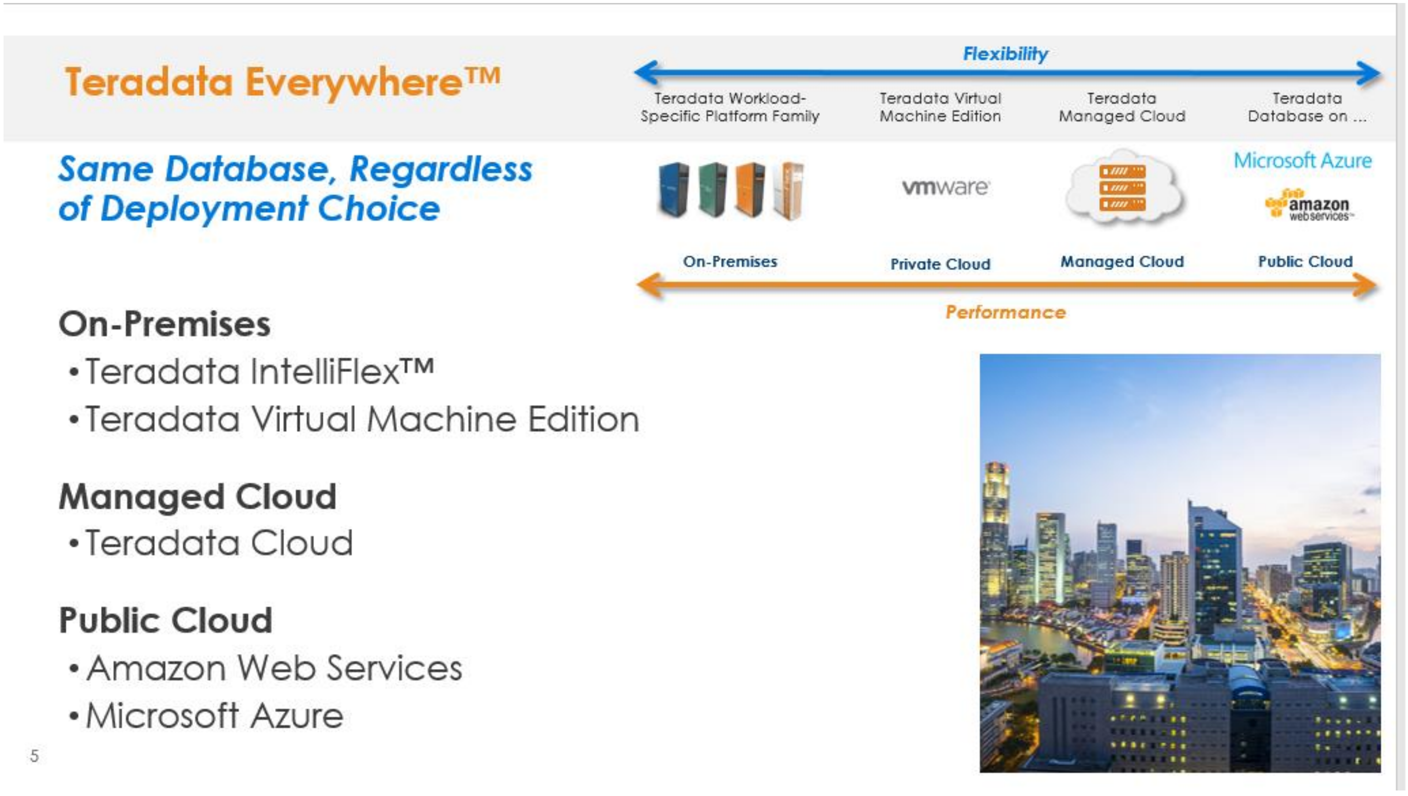


Teradata Everywhere

- Deployment flexibility
  - Fully featured Teradata Database
  - Same database across platforms
  - MPP scalability
- Design for data gravity
  - Co-locate with source data to minimize data movement and duplication
- Employ multi-system architectures
  - High availability, disaster recovery, workload distribution and shifting
- As your business changes, so can your deployment choice

4

TERADATA.



Teradata Everywhere™

Flexibility

Teradata Workload-Specific Platform Family | Teradata Virtual Machine Edition | Teradata Managed Cloud | Teradata Database on ...

vmware | Microsoft Azure | amazon web services

On-Premises | Private Cloud | Managed Cloud | Public Cloud

Performance

Same Database, Regardless of Deployment Choice

On-Premises
- Teradata IntelliFlex™
- Teradata Virtual Machine Edition

Managed Cloud
- Teradata Cloud

Public Cloud
- Amazon Web Services
- Microsoft Azure

5

21. Teradata QueryGrid Overview – course 52285

Direct link to the course:
https://university.teradata.com/learning/user/deeplink_redirect.jsp?linkId=ITEM_DETAILS&componentID=52285&componentTypeID=RECORDED-WEBCAST&revisionDate=1420477200000

Teradata QueryGrid Overview 52285 - Part 1: Introduction (9 minutes), 00:00 to 05:30

22. Teradata Listener

http://www.teradata.com/products-and-services/listener

23. SQL Data Definition Language Detailed Topics

ALTER TABLE, Fallback, and Block Level Compression of Fallback Tables

The effect on block level compression of altering a table defined with block level compression to have fallback falls into two general categories.

•The ALTER TABLE request alters the specified table in some way, but does not add fallback.

In this case, the table retains the block level compression it had before the request was submitted.

•The ALTER TABLE request adds fallback to the table. Whether the table definition is otherwise altered makes no difference.

In this case, the primary table retains the block level compression it had before the request was submitted, but the newly created fallback table does *not* inherit the block level compression from its primary table by default.

There are two possible actions for this case.

•You do nothing to override the system defaults for data block compression.

In this case, the block level compression assigned to the new fallback table depends on the system defaults that have been defined for your site using the compression fields of the DBS Control record (see *Utilities: Volume 1 (A-K)*) and whether you submit a SET QUERY_BAND … FOR SESSION request that overrides those defaults.

•You submit a SET QUERY_BAND request to override the system defaults for data block compression.

If you submit the ALTER TABLE request with a SET QUERY_BAND … FOR SESSION request that specifies BlockCompression=Y (see "Storage Management Query Bands"), Teradata Database creates the fallback table with block level compression regardless of the settings of the compression fields of the DBS Control record.

You must specify the query band FOR SESSION, not FOR TRANSACTION, because both SET QUERY_BAND and ALTER TABLE are DDL statements, and you cannot specify more than one DDL statement per transaction.

The following table explains how Teradata Database does or does not assign block level compression to a newly created fallback table for this case.

IF you submit an ALTER TABLE request that changes the table definition, adds fallback, and …	**THEN the FALLBACK table …**
also specifies the BlockCompression query band	ıses block level compression if the value for BlockCompression is set to Y. loes not use block level compression if the value for BlockCompression is set to N.
do not also specify the BlockCompression query band	defaults to the system-wide compression characteristics for your site as defined by the compression fields of the DBS Control record (see *Utilities: Volume 1 (A-K)*).

http://www.info.teradata.com/HTMLPubs/DB_TTU_13_10/index.html#page/SQL_Reference/B035_1184_109A/Alter_Function-Details.03.038.html

24. SQL Data Definition Language Syntax and Examples

Teradata Database, Tools and Utilities – Release 16.--

Table Kind

Table Kind option, CREATE TABLE SQL statement.

The kind of table determines duplicate row control. See SQL Data Definition Language - Detailed Topics, B035-1184 for details. The table can be created as a global temporary table or a volatile table. If you do not specify global temporary or volatile, then the table is defined as a persistent user data table, also referred to as base tables. Hash and join index tables are also considered base tables.

If you do not explicitly specify SET or MULTISET, the table kind assignment depends on the session mode:

Session Mode	Default
ANSI	MULTISET
Teradata	SET

The session mode default is in effect, except for when you:

- Copy a table definition using the non-subquery form of the CREATE TABLE … AS syntax. The default table kind is the table kind of the source table, regardless of the current session mode.
  Create a column-partitioned table. The default table kind is always MULTISET, regardless of the session mode or the setting of the DBS Control parameter PrimaryIndexDefault.

MULTISET

Duplicate rows are permitted, in compliance with the ANSI SQL:2011 standard. If there are uniqueness constraints on any column or set of columns in the table definition, then the table cannot have duplicate rows even if it is declared as MULTISET. Teradata Database creates NoPI and column-partitioned tables as MULTISET tables by default.

Some client utilities have restrictions regarding MULTISET tables. See the appropriate documentation:

- ◦Teradata FastLoad Reference
- ◦Teradata Archive/Recovery Utility Reference
- ◦Teradata Parallel Data Pump Reference

SET

Duplicate rows are not permitted. You cannot create the following kinds of tables as SET tables:

- ◦Temporal
- ◦Column-partitioned
- ◦NoPI

GLOBAL TEMPORARY

A temporary table definition is created and stored in the data dictionary for future materialization. You can create global temporary tables by copying a table WITH NO DATA, but not by copying a table WITH DATA.

You cannot create a column-partitioned global temporary table.

You cannot create a global temporary table with row-level security constraint columns.

VOLATILE

Create a volatile table. The definition is of a volatile table is retained in memory only for the duration of the session in which it is defined. Space usage is charged to the login user spool space. Because volatile tables are private to the session that creates them, the system does not check the creation, access, modification, and drop privileges. A single session can materialize up to 1,000 volatile tables.

The contents and the definition of a volatile table are dropped when a system reset occurs.

If you frequently reuse particular volatile table definitions, consider writing a macro that contains the CREATE TABLE text for those volatile tables.

You cannot create a column-partitioned volatile table or normalized volatile table.

You cannot create secondary, hash, or join indexes on a volatile table.

You cannot create a volatile table with row-level security constraint columns.

For further information about volatile tables, see SQL Data Definition Language - Detailed Topics, B035-1184

http://info.teradata.com/htmlpubs/DB_TTU_16_00/index.html#page/SQL_Reference/B035-1144-160K/qmh1472241477693.html

25. SQL Data Definition Language, Syntax and Examples

ALTER PROCEDURE (SQL Form)

Invocation Restrictions

Valid for SQL procedures only.

Not valid inside a procedure body.

Limitations

You cannot use ALTER PROCEDURE to change the DDL definition of an SQL procedure, that is, to REPLACE the procedure. To replace the definition on an SQL procedure, you must submit a REPLACE PROCEDURE request (see CREATE PROCEDURE and REPLACE PROCEDURE (SQL Form)).

Attributes Changed by ALTER PROCEDURE (SQL Form)

ALTER PROCEDURE can alter the following attributes of the recompiled SQL procedure.

- Platform.

  This is an implicit change and cannot be specified by an ALTER PROCEDURE request.

- TDSP version number.

  This is an implicit change and cannot be specified by an ALTER PROCEDURE request. For information about procedure version numbers, see HELP PROCEDURE.

- Creation time zone.

  This is an explicit change that you specify using the AT TIME ZONE option.

You can also change one or all of the following attributes:

- SPL to NO SPL.

  You cannot change NO SPL back to SPL.

- WARNING to NO WARNING and vice versa.

Attributes Not Changed by ALTER PROCEDURE (SQL Form)

The creator and the immediate owner of a procedure are not changed after recompilation.

ALTER PROCEDURE (SQL Form) also does not change the following attributes of the procedure being recompiled.

- Session mode
- Creator character set
- Creator character type
- Default database
- Privileges granted to the procedure

26. SQL Request and Transaction Processing

About Locking Levels

The hierarchy of locking levels for a database management system is a function of the available granularities of locking, with database-level locks having the coarsest granularity and rowkey-level locks having the finest granularity. Depending on the request being processed, the system places a certain default lock level on the object of the request, which can be one of the following database objects:

- Database
- Table

  See Proxy Locks for a description of a special category of table-level locking.

- View
- Partition
- RowHash
- RowKey (Partition and RowHash)

Lock Level …	What is Locked
Database	all rows of all tables in the specified database and their associated secondary index subtables.
Table	all rows in the specified base table and in any secondary index and fallback subtables associated with it.
View	all underlying tables accessed by the specified view.
Partition	the primary and fallback copy of rows in a partition for the specified table or single-table view. The table must be row partitioned. This lock permits other users to access the data in the table that are not in the same partition.
PartitionRange	the primary and fallback copy of rows in a range of partitions for the specified table or single-table view. The table must be row partitioned. This lock permits other users to access the data in the table that are outside the specified partition range.
RowHash	the specified primary or fallback copy of rows sharing the same row hash value for the specified table or single-table view. For a row-partitioned table, this lock level applies to the row hash value for all partitions. The rowhash-level lock permits other users to access the data in the table that do not have the same rowhash. The rowhash-level lock applies to a set of rows that shares the same hash code. It does not necessarily lock only a single row, since multiple rows may have the same rowhash. • A rowhash-level lock is applied whenever a non-row-partitioned table is accessed by using a unique primary index (UPI) or a nonunique primary index (NUPI). • For an update or delete that accesses a data row by using a unique secondary index (USI), the appropriate rowhash of the USI subtable is locked, as well as the indexed data rowhash or rowkey. • Rowhash locks on a table's nonunique secondary index (NUSI) subtables are usually not needed. First, a query or DML that uses a NUSI access path locks the whole table. Second, DML that does not lock the whole table

Lock Level …	What is Locked
	uses task locks rather than rowhash locks on any NUSI subtables that require index maintenance.
RowHash in a PartitionRange	the specified primary or fallback copy of rows sharing the same row hash value for the specified table or single-table view in a range of partitions. The table must be row partitioned. This lock permits other users to access other data in the table that do not have the same rowhash or are outside the specified partition range. The rowhash-level lock applies to a set of rows that shares the same hash code. It does not necessarily lock only one row since multiple rows may have the same rowhash in the same partition or in more than one partition in the partition range. This lock level is not used on rows in a USI or NUSI index subtable, as these subtables are never partitioned.
RowKey	the specified primary or fallback copy of rows sharing the same rowkey (partition and row hash value) for the specified table or single-table view. The table must be row partitioned. A rowkey-level lock permits other users to access other data in the table that do not have the same rowhash or partition value. A rowkey-level lock applies to a set of rows that shares the same partition and rowhash. It does not necessarily lock only one row since there could be multiple rows with the same rowhash in a partition. • A rowkey-level lock is applied whenever a row-partitioned table with a primary index (UPI or NUPI) is accessed by specifying the primary index and partitioning column values. • The rowkey-level lock is not used on rows in a USI or NUSI subtable, as these subtables are never partitioned.

The locking level determines whether other users can access the target object.

Locking severities and locking levels combine to exert various locking granularities. The less granular the combination, the greater the impact on concurrency and system performance, and the greater the delay in processing time.

27. SQL Request and Transaction Processing

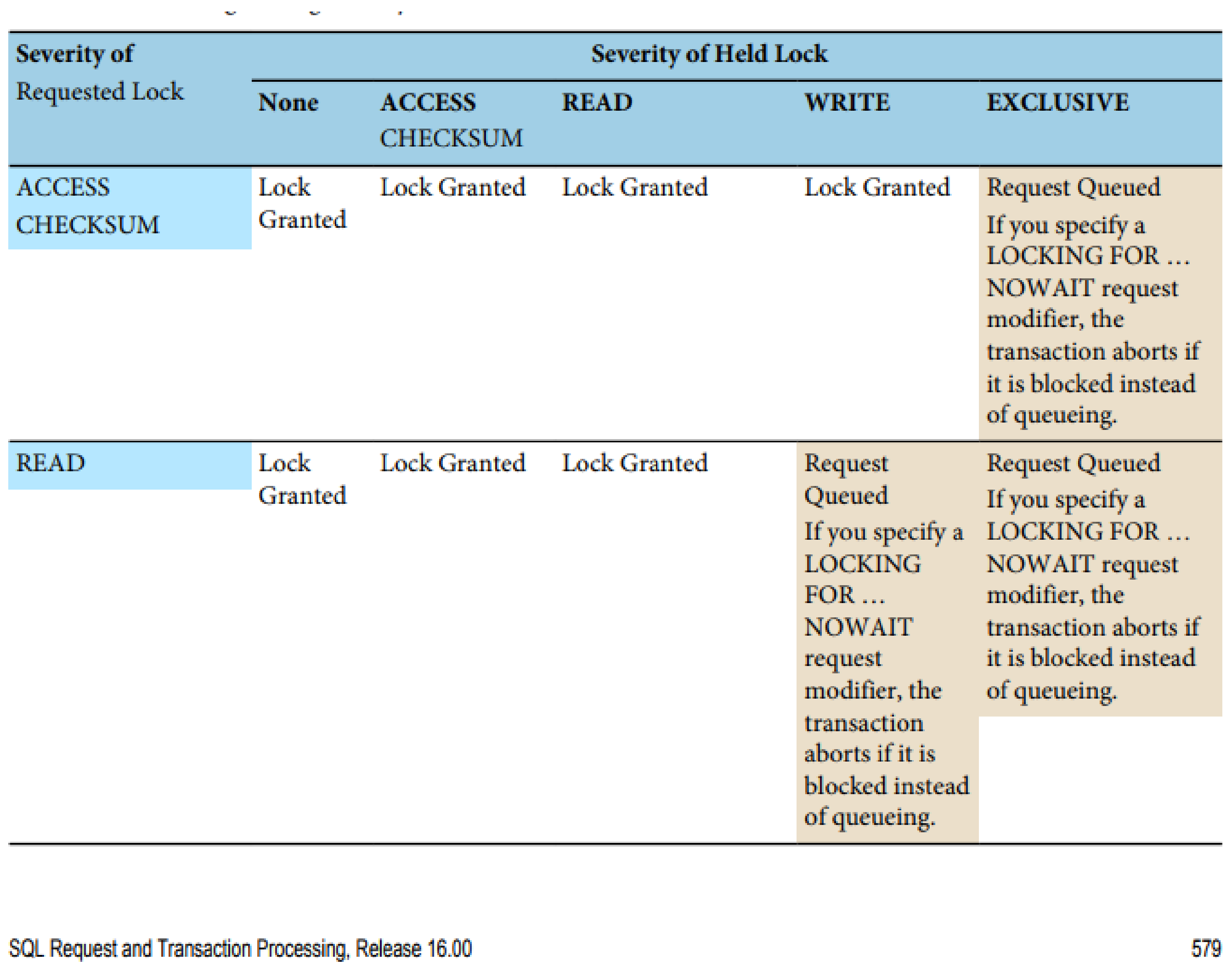
Notation	Definition
`lock(<object>,<lock requested>)`	A database object-locking severity requested pair.

Each database object has an associated locking queue *Q* and list of currently held locks *L*. All requests perform a locking operation before they access any database objects.

IF lock(<object>,<lock requested>) is …	THEN …
queued for any lock in *L*	the transaction is placed in *Q* and waits there as long as `lock(<object>,<lock requested>)` is queued. A request in this state is said to be blocked (see Blocked Requests).
granted	`lock(<object>,<lock requested>)` is added to *L* with `<lock requested>` and the transaction resumes processing.

After the transaction finishes with an object by either committing or rolling back, its lock is removed from *L*.

The table on the following page summarizes the action taken when a requested locking severity competes with an existing locking severity.



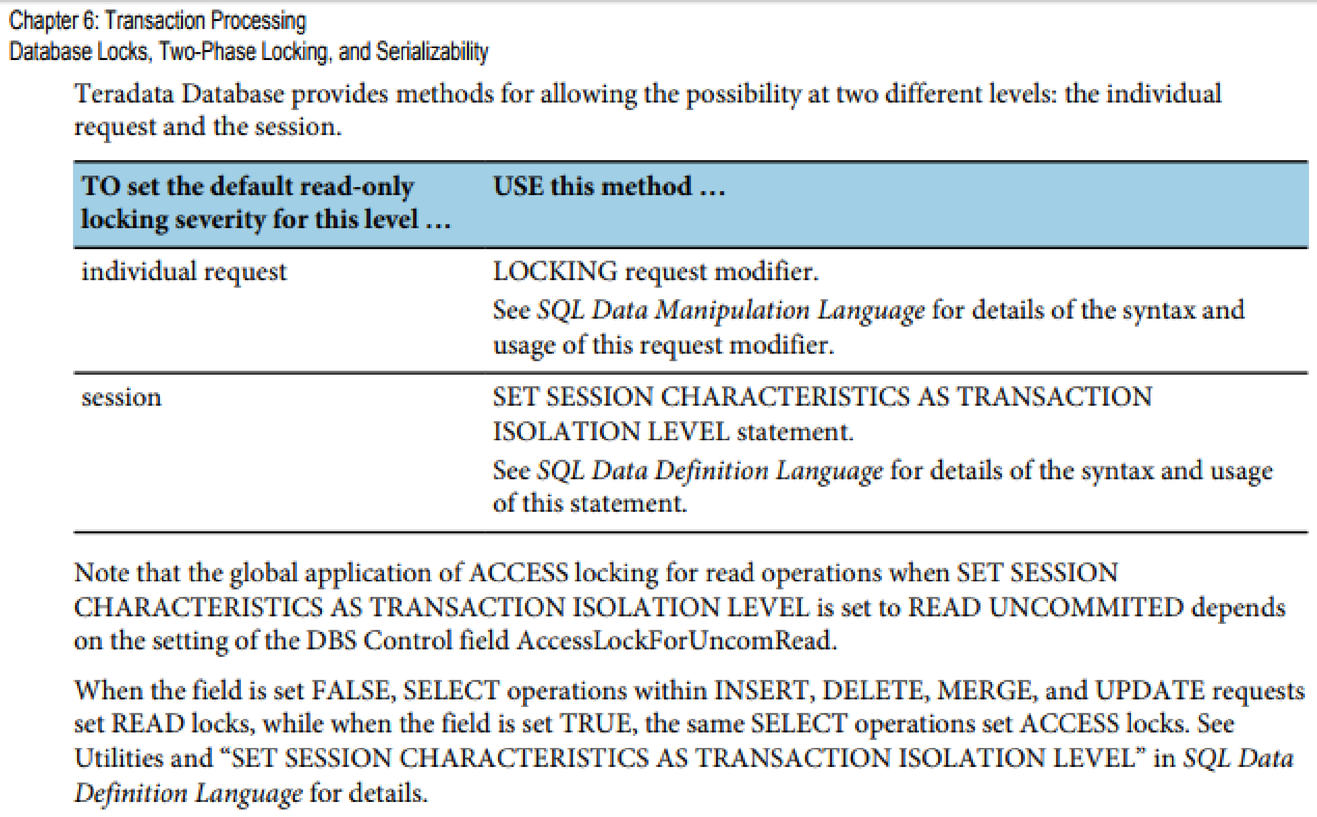
Severity of Requested Lock	Severity of Held Lock				
	None	**ACCESS** CHECKSUM	**READ**	**WRITE**	**EXCLUSIVE**
ACCESS CHECKSUM	Lock Granted	Lock Granted	Lock Granted	Lock Granted	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.
READ	Lock Granted	Lock Granted	Lock Granted	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.

Teradata Database Locking Levels and Severities

Severity of Requested Lock	Severity of Held Lock				
	None	ACCESS CHECKSUM	READ	WRITE	EXCLUSIVE
WRITE	Lock Granted	Lock Granted	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.
EXCLUSIVE	Lock Granted	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.

Because other client utilities such as BTEQ, FastExport, FastLoad, MultiLoad, Teradata Parallel Data Pump, and Teradata Parallel Transporter use standard database locks, the interactions of those locking severities with those of other database locks are identical. See the following manuals for details of the locks set by those utilities:

28. SQL Request and Transaction Processing



Teradata Database provides methods for allowing the possibility at two different levels: the individual request and the session.

TO set the default read-only locking severity for this level …	**USE this method …**
individual request	LOCKING request modifier. See *SQL Data Manipulation Language* for details of the syntax and usage of this request modifier.
session	SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL statement. See *SQL Data Definition Language* for details of the syntax and usage of this statement.

Note that the global application of ACCESS locking for read operations when SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL is set to READ UNCOMMITED depends on the setting of the DBS Control field AccessLockForUncomRead.

When the field is set FALSE, SELECT operations within INSERT, DELETE, MERGE, and UPDATE requests set READ locks, while when the field is set TRUE, the same SELECT operations set ACCESS locks. See Utilities and "SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL" in *SQL Data Definition Language* for details.

Retrievals With an ACCESS Lock

Because an ACCESS lock is compatible with all locking severities except EXCLUSIVE, a user requesting an ACCESS lock might be allowed to read an object on which a WRITE lock is being held, a situation that is referred to as a *dirty read*. This means that data could be retrieved by an application holding an ACCESS lock while that same data is simultaneously being modified by an application holding a WRITE lock. Therefore, any query that places an ACCESS lock can return incorrect or inconsistent results.

For example, assume that a SELECT request that uses a secondary index constraint is submitted with a LOCKING FOR ACCESS phrase.

If the ACCESS lock is granted on an object being held for WRITE by another user, the column value could change between the time the secondary index subtable key is located and the time the data row is retrieved (such a change is possible because a satisfied SELECT index constraint is not always double-checked against the base data row). This type of inconsistency might occur even if the data is changed only momentarily by a transaction that is later backed out.

The normal policy for ACCESS lock Read operations is that Teradata Database guarantees to return all rows that were not being updated at the time of the ACCESS lock Read operation to the requestor. For rows that are being updated, rows may be returned, possibly in an inconsistent state, or not returned.

This generalizes to ACCESS lock Read operations on a geospatial NUSI as follows: if while attempting to perform an ACCESS locked Read on a geospatial index, Teradata Database discovers that the current state of the Hilbert R-Tree for the index does not permit all rows not being updated to be returned to the requestor, it returns a retryable error to that requestor.

Considerations for Specifying LOCKING FOR ACCESS

A READ lock is normally placed on an object for a SELECT operation, which causes the request to be queued if the object is already locked for WRITE.

If an ad hoc query has no concern for data consistency, the LOCKING request modifier can be used to override the default READ lock with an ACCESS lock. For example:

```
LOCKING TABLE tablename FOR ACCESS
SELECT ...
FROM tablename ...;
```

Be aware that the effect of LOCKING FOR ACCESS is that of reading while writing, so dirty reads can occur with this lock. The best approach to specifying ACCESS locks is to use them only when you are interested in a broad, statistical snapshot of the data in question, not when you require precise results. On load-isolated tables, however, LOCKING FOR LOAD COMMITTED may be used to obtain committed data even while concurrent isolated writes occur simultaneously on the table.

ACCESS locking can result in incorrect or inconsistent data being returned to a requestor, as detailed in the following points:

- A SELECT with an ACCESS lock can retrieve data from the target object even when another request is modifying that same object.

  Therefore, results from a request that applies an ACCESS lock can be inconsistent.

  If this occurs while you are using an ACCESS lock to read data from a geospatial NUSI column and the current state of the Hilbert R-tree for that NUSI does not permit all of the unmodified rows to be returned to you, Teradata Database returns a retryable error to you.
- The possibility of an inconsistent return is especially high when the request applying the ACCESS lock uses a secondary index value in a conditional expression.

  If the ACCESS lock is granted on an object being held for WRITE, the constraint value could change between the time the secondary index subtable is located and the time the data row is retrieved.

  Such a change is possible because a satisfied SELECT index constraint is not always double-checked against the base data row.

The LOCKING ROW request modifier cannot be used to lock multiple row hashes. If LOCKING ROW FOR ACCESS is specified with multiple row hashes, the declaration implicitly converts to LOCKING TABLE FOR ACCESS.

Using the LOCKING Request Modifier: An Example

The possibility of an inconsistent return is especially high when an ACCESS request uses a secondary index value in a conditional expression, because satisfied index constraints are not always rechecked against the retrieved data row.

For example, assuming that *qualify_accnt* is defined as a secondary index, the following request could return the result that follows the request text:

```
    LOCKING TABLE accnt_rec FOR ACCESS
    SELECT accnt_no, qualify_accnt
    FROM accnt_rec
    WHERE qualify_accnt = 1587;
    Accnt_No  Qualify_Accnt
    --------  ------------
        1761          4214
```

In this case, the value 1587 was found in the secondary index subtable, and the corresponding data row was selected and returned. However, the data for account 1761 had been changed by the other user while this selection was in progress.

Returns such as this are possible even if the data is changed or deleted only momentarily by a transaction that is subsequently aborted.

This type of inconsistency can occur even if the data is changed only momentarily by a transaction that is later backed out. Note that for load isolated tables, you can avoid such inconsistency by using the LOCKING FOR LOAD COMMITTED modifier. Refer to Load Isolation and *SQL Data Manipulation Language*.

29. SQL Request and Transaction Processing

Lock Severity	Description
EXCLUSIVE	EXCLUSIVE locks are placed only on a database or table when the object is undergoing structural changes (for example, a column is being created or dropped). You can also place an EXCLUSIVE lock explicitly using the LOCKING request modifier (see “LOCKING Request Modifier” in *SQL Data Manipulation Language*).
WRITE	Placed in response to an INSERT, UPDATE, or DELETE request. A WRITE lock restricts access by other users (except for applications that are not concerned with data consistency and choose to override the automatically applied WRITE lock by specifying a less restrictive ACCESS lock). You can also place this lock explicitly using the LOCKING request modifier (see “LOCKING Request Modifier” in *SQL Data Manipulation Language*).
READ	A READ lock is placed in response to a SELECT request and restricts access by users who require EXCLUSIVE or WRITE locks. Several users can hold READ locks on a resource, during which the system permits no modification of that resource. READ locks ensure consistency during READ operations such as those that occur during a SELECT statement. You can also place the READ lock explicitly using the LOCKING request modifier (see “LOCKING Request Modifier” in *SQL Data Manipulation Language*).
The CHECKSUM and ACCESS locking severities are all at the same level in the restrictiveness hierarchy.	
CHECKSUM	Placed in response to a user-defined LOCKING FOR CHECKSUM modifier (see “LOCKING Request Modifier” in *SQL Data Manipulation Language*) when using cursors in embedded SQL. CHECKSUM locking is identical to ACCESS locking except that it adds checksums to the rows of a spool to allow a test of whether a row in the cursor has been modified by another user or session at the time an update is being made through the cursor. See also Cursor Locking Modes, *SQL Data Manipulation Language*, and *SQL Stored Procedures and Embedded SQL*.
ACCESS	Placed in response to a user-defined LOCKING FOR ACCESS modifier (see “LOCKING Request Modifier” in *SQL Data Manipulation Language*), or by setting the session default isolation level to READ UNCOMMITTED using the SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL statement (see *SQL Data Definition Language*). Permits a user to have a form of read access to an object that might already be locked for READ or WRITE. An ACCESS lock does not restrict access by another user except when an EXCLUSIVE lock is required; therefore it is sometimes referred to as a *dirty READ lock*. An ACCESS lock Read operation on a geospatial NUSI. If Teradata Database discovers while attempting to perform an ACCESS locked

Lock Severity	Description
	Read operation that the current state of the Hilbert R-Tree for the NUSI does not permit all non-updated rows to be returned to the requestor, it returns a retryable error to that requestor. See *SQL Geospatial Types*. The global application of ACCESS locking for read operations when SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL is set to READ UNCOMMITED depends on the setting of the DBS Control field AccessLockForUncomRead. When the parameter is set FALSE, SELECT operations within INSERT, DELETE, MERGE, and UPDATE requests set READ locks, while when the parameter is set TRUE, the same SELECT operations set ACCESS locks. A user requesting an ACCESS lock disregards all data consistency issues. Because ACCESS and WRITE locks are compatible, the data might be undergoing updates while the user who requested the access is reading it. Therefore, any query that requests an ACCESS lock might return incorrect or inconsistent results. An ACCESS lock is also placed in response to a user-defined LOAD COMMITTED locking modifier. See "LOCKING Request Modifier" in *SQL Data Manipulation Language*. For information about designating tables for ISOLATED LOADING, see SQL *Data Definition Language Syntax and Examples*. For a load-isolated table, this modifier allows users to read from committed rows in tables, even while the table is being loaded with data. If the table that is being read is not a load-isolated table, this severity results in an ACCESS lock. For information about load isolation, see Load Isolation.
Least Restrictive	

Compatibility Among Locking Severities

The Teradata Lock Manager controls the interaction of following types of lock, when placed at specific levels:

- ACCESS.
- CHECKSUM.
- READ.
- WRITE.
- EXCLUSIVE.

For information about HUT locks, see *Teradata Archive/Recovery Utility Reference*.

The following notation is used to describe the locking severity compatibilities:

Notation	Definition
Q	A locking queue associated with a database object.
L	A list of locks currently held.

Notation	Definition
`lock(<object>,<lock requested>)`	A database object-locking severity requested pair.

Each database object has an associated locking queue *Q* and list of currently held locks *L*. All requests perform a locking operation before they access any database objects.

IF lock(<object>,<lock requested>) is …	**THEN …**
queued for any lock in *L*	the transaction is placed in *Q* and waits there as long as `lock(<object>,<lock requested>)` is queued. A request in this state is said to be blocked (see Blocked Requests).
granted	`lock(<object>,<lock requested>)` is added to *L* with `<lock requested>` and the transaction resumes processing.

After the transaction finishes with an object by either committing or rolling back, its lock is removed from *L*.

The table on the following page summarizes the action taken when a requested locking severity competes with an existing locking severity.

Severity of Requested Lock	Severity of Held Lock				
	None	**ACCESS** CHECKSUM	**READ**	**WRITE**	**EXCLUSIVE**
ACCESS CHECKSUM	Lock Granted	Lock Granted	Lock Granted	Lock Granted	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.
READ	Lock Granted	Lock Granted	Lock Granted	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.

Severity of Requested Lock	**Severity of Held Lock**				
	None	**ACCESS CHECKSUM**	**READ**	**WRITE**	**EXCLUSIVE**
WRITE	Lock Granted	Lock Granted	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.
EXCLUSIVE	Lock Granted	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.	Request Queued If you specify a LOCKING FOR … NOWAIT request modifier, the transaction aborts if it is blocked instead of queueing.

Because other client utilities such as BTEQ, FastExport, FastLoad, MultiLoad, Teradata Parallel Data Pump, and Teradata Parallel Transporter use standard database locks, the interactions of those locking severities with those of other database locks are identical. See the following manuals for details of the locks set by those utilities:

- *Basic Teradata Query Reference*
- *Teradata FastExport Reference*
- *Teradata FastLoad Reference*
- *Teradata MultiLoad Reference*
- *Teradata Parallel Data Pump Reference*
- *Teradata Parallel Transporter Reference*

A queued request is in an I/O wait state and is said to be blocked (see Blocked Requests).

A WRITE or EXCLUSIVE lock on a database, table, or view restricts all requests or transactions except the one holding the lock from accessing data within the domain of that object.

Because a lock on an entire database can restrict access to a large quantity of data, the Parser ensures that default database locks are applied at the lowest possible level and severity required to secure the integrity of the database while simultaneously maximizing concurrency.

Table-level WRITE locks on dictionary tables prevent contending tasks from accessing the dictionary, so the Parser attempts to lock dictionary tables at the rowhash or rowkey (partition and rowhash) level whenever possible.

For information about how load isolation affects compatibility among locking severities, see Load Isolation.

Using the NOWAIT Option for the SQL LOCKING Request Modifier

When you specify the NOWAIT option for the SQL LOCKING request modifier, Teradata Database aborts a transaction that makes a lock request that cannot be fulfilled immediately. For details on how to use the NOWAIT option with the LOCKING request modifier, see "LOCKING Request Modifier" in *SQL Data Manipulation Language*.

Teradata Database uses a slight variation of this code internally to avoid blocking on DDL operations. Instead of aborting the request, the system instead downgrades rowhash lock severities from READ to ACCESS. See DDL and DCL Requests, Dictionary Access, and Locks.

AMP-Based Utilities and Logging

Following are the utilities related to logging:

- Lock Viewer Viewpoint portlet

  This portlet produces a report of miscellaneous database lock delay information that you can use to detect blocked transactions and global deadlocks. Teradata Database extracts the data reported by the Lock Viewer portlet from various DBQL transaction logs.
- Show Locks

  This utility produces a report on the various Host Utility (HUT) locks. For more information about HUT locks, see *Teradata Archive/Recovery Utility Reference*.

30. Database Design

There are 44 pages of material identified.

Teradata Database: 16.00, B035-1094-160K (see separate reference document)

Or, Information Products site that has the link to the referenced document – Teradata employees only:. http://www.info.teradata.com/doclist.cfm?Prod=1060&ProdName=Teradata%20Database

31. Teradata Database Security

Teradata Database security is based on the following concepts.

Security Element	Description
User	An individual or group of individuals represented by a single user identity.
Privileges	The database privileges explicitly or automatically granted to a user or database.
Logon	The process of submitting user credentials when requesting access to the database.
Authentication	The process by which the user identified in the logon is verified.
Authorization	The process that determines the database privileges available to the user.
Security Mechanism	A method that provides specific authentication, confidentiality, and integrity services for a database session.
Network Traffic Protection	The process for protecting message traffic between Teradata Database and mainframe-attached and workstation-attached clients against interception, theft, or other form of attack.
Message Integrity	Checks data sent across the network against what was received to ensure no data was lost or changed.
Access Logs	Logs that provide the history of users accessing the database and the database objects accessed.

For detailed information on these topics, see *Security Administration*.

Users

Users that access Teradata Database must be defined in the database or a supported directory.

Permanent Database Users

The CREATE USER statement defines permanent database users. Teradata recommends that the username represent an individual. Each username must be unique in the database.

Directory-based Users

Directory-based users that access the database must be defined in a supported directory. Creation of a matching database user may or may not be required, depending upon implementation. One or more configuration tasks must be completed before directory users can access the database.

Proxy Users

Proxy users are end users who access Teradata Database through a middle-tier application set up for trusted sessions. They are authenticated by the application rather than the database. The GRANT CONNECT THROUGH statement assigns role-based database privileges to proxy users. To establish privileges for a particular connection to the database, the application submits the SET QUERY_BAND statement when it connects the user. The SET QUERY_BAND statement and the rules for how it is applied to each proxy user must be coded into the application as part of setup for trusted sessions.

Database Privileges

Users can access only database objects on which they have privileges. The following table lists the types of database privileges and describes how they are acquired by a user.

Privilege	Description
Implicit (Ownership)	Privileges implicitly granted by the database to the owner of the space in which database objects are created.
Automatic	Privileges automatically provided by the system to: • The creator of a database, user, or other database object. • A newly created user or database.
Inherited	Privileges that are passed on indirectly to a user based on its relationship to another user or role to which the privileges were granted directly. • Directory users inherit the privileges of the database users to which they are mapped. Directory users who are members of groups also inherit the privileges defined in external roles to which the groups are mapped. • All users inherit the privileges of PUBLIC, the default database user, whether or not they have any other privileges.
Explicit (GRANT)	Privileges granted explicitly to a user or database in one of the following ways: • GRANT directly to a user or database. • GRANT to a role, then GRANT the role to one or more users.

Directly Granted Privileges

Privileges can be directly given to users with the GRANT statement. Administrators GRANTing a privilege must have been previously granted the privilege they are granting, as well as the WITH GRANT OPTION privilege on the privilege.

For additional information on how to use SQL statements to GRANT and REVOKE privileges, see *SQL Data Control Language*.

Roles

Roles can be used to define privileges on database objects for groups of users with similar needs, rather than granting the privileges to individual users. Roles also require less dictionary space than individually granted privileges. Use the CREATE ROLE statement to define each role, then use the GRANT statement to grant roles to users. The CREATE USER statement must also specify the default role for the user. The MODIFY USER statement can be used to assign additional user roles.

A member of a role may access all objects to which a role has privileges. Users can employ the SET ROLE statement to switch from the default to any alternate role of which the user is a member, or use SET ROLE ALL to access all roles.

For more information on use of roles, see *Database Administration*.

Roles for Proxy Users

Proxy users are users that access the database through a middle-tier application set up to offer trusted sessions. Proxy users are limited to privileges defined in roles that are assigned to them using the GRANT CONNECT THROUGH statement.

For details on using GRANT CONNECT THROUGH, see *Security Administration* and *SQL Data Control Language*.

External Roles

Use external roles to assign role privileges to directory users. External roles are created exactly like database roles; however, they cannot be granted to directory users because these users do not exist in the database. Instead, directory users must be members of groups that are mapped to external roles in the directory.

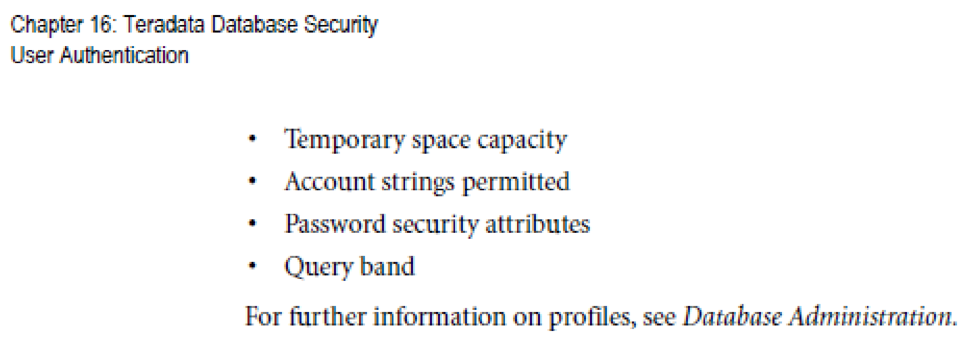
Directory users mapped to multiple external roles have access to all of them at logon to the database.

For information on mapping directory users to database objects, see *Security Administration*.

Profiles

To simplify user management, an administrator can define a profile and assign it to a group of users who share similar values for the following types of parameters:

- Default database assignment
- Spool space capacity



Chapter 16: Teradata Database Security
User Authentication

- Temporary space capacity
- Account strings permitted
- Password security attributes
- Query band

For further information on profiles, see *Database Administration.*

32. Security Administration

Using Teradata Wallet to Store and Retrieve Logon Elements

Users can optionally store usernames and passwords on a Teradata client computer or application server running Teradata Tools and Utilities 14.0 and up, using the included Teradata Wallet software, and then retrieve the needed data when logging on to any compatible Teradata Database system.

Benefits

Teradata Wallet storage is especially beneficial for easy retrieval of passwords on application servers or other shared computers that host multiple users and connect to multiple databases. Users wanting to use retrieved data in logon strings must have a personal Teradata Wallet instance on each computer through which they access Teradata Database.

Passwords and other data are securely stored in protected form. See Encryption.

Each user can store data only in their own wallet, which is not accessible by other users. The system retrieves data only from the wallet belonging to the logged on user.

Teradata Wallet substitution strings are accepted by both the .logon and .logdata statements in the logon string, and by the corresponding logon functions in ODBC (14.10 and up only) and CLI applications and scripts.

Use Cases

Users running scripted applications can embed password retrieval syntax into scripts instead of compromising security by including a password.

Users accessing multiple Teradata Database systems can automatically retrieve the correct username and password for a system (tdpid) instead of having to remember the information or look it up.

Restrictions

- On Windows, using the Credential Manager to modify Teradata Wallet string values is not supported because it corrupts the values and they must be deleted and re-added using the tdwallet command-line tool.
- When multiple users log on to the database from a single computer, each user must be uniquely identified on the computer so retrieval of wallet data is user-specific and private.
- Teradata Database authenticated users (TD2 mechanism) must reset passwords stored in Teradata Wallet to conform to any changes required by database password controls, for example, PasswordExpire. See Working with Password Controls.

Prerequisites

The topics that follow, showing how to use Teradata Wallet to store and retrieve logon string information, assume that Teradata Wallet is installed and configured on a Teradata client. For detailed information on Teradata Wallet installation and setup options, see the Teradata Tools and Utilities Installation Guide for the client operating system.

33. Security Administration

CHAPTER 11 Implementing Row Level Security

About Row Level Security

Access to Teradata Database objects is controlled primarily by object level user privileges. Object level privileges are discretionary, that is, object owners automatically have the right to grant access on any owned object to any other user.

In addition to object level privileges, you can use row level security (RLS) to control user access by table row and by SQL operation. RLS access rules are based on the comparison of the RLS access capabilities of each user and the RLS access requirements for each row.

Object owners do not have discretionary privileges to grant row access to other users. Only users with security constraint administrative privileges can manage row level access controls.

When multiple Teradata Database systems are managed by Unity, the same row level security constraints and access privileges should exist on all database systems.

Row Level Security Compared to View and Column Access Controls

Implementation of row level security can be complicated compared to standard discretionary access controls. Before you commit to using row level security, determine whether or not you can meet access control needs by more conventional means, for example:

- Grant user access to views that do not include columns with sensitive data, instead of granting user privileges on the entire base table.
- Grant or revoke access privileges only on selected columns in the base table.

When comparing access control methods, consider that view and column level access controls:

- Are usually adequate for controlling SELECT statements, but users cannot execute INSERT, UPDATE, and DELETE statements on columns they cannot see, and must revert to accessing the base tables for these operations.
- Are discretionary, that is, the object owner can grant access to any user.

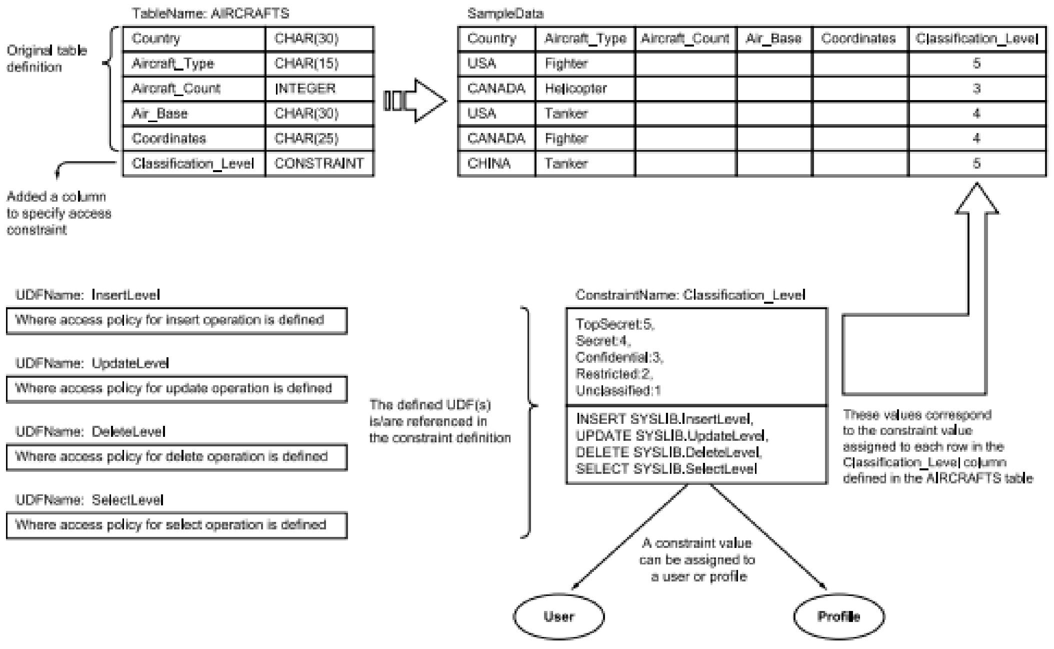
Related Information

For additional information on the use of view and column level privileges, see Other Options for Limiting Database Access.

Elements of Row Level Security

Element	Description
Security classification category	A set of labels (access levels or compartments) used to define user access capabilities and row access requirements.
Security CONSTRAINT	• A CONSTRAINT object named for a security classification system, which: ∘ Defines the range of valid label values for the classification system ∘ Specifies 1 to 4 security constraint UDFs ∘ Can be assigned to users to define their row access capabilities • A table column named for a CONSTRAINT object, in which the column value for each row determines the row access requirement
Security constraint user defined function (UDF)	Defines and enforces RLS policy on each incoming INSERT, SELECT, UPDATE, or DELETE statement.

The drawing shows the components required to define row-level security.



TableName: AIRCRAFTS

Country	CHAR(30)
Aircraft_Type	CHAR(15)
Aircraft_Count	INTEGER
Air_Base	CHAR(30)
Coordinates	CHAR(25)
Classification_Level	CONSTRAINT

SampleData

Country	Aircraft_Type	Aircraft_Count	Air_Base	Coordinates	Classification_Level
USA	Fighter				5
CANADA	Helicopter				3
USA	Tanker				4
CANADA	Fighter				4
CHINA	Tanker				5

Original table definition
Added a column to specify access constraint
UDFName: InsertLevel
Where access policy for insert operation is defined
UDFName: UpdateLevel
Where access policy for update operation is defined
UDFName: DeleteLevel
Where access policy for delete operation is defined
UDFName: SelectLevel
Where access policy for select operation is defined
The defined UDF(s) is/are referenced in the constraint definition
ConstraintName: Classification_Level
TopSecret:5, Secret:4, Confidential:3, Restricted:2, Unclassified:1
INSERT SYSLIB.InsertLevel, UPDATE SYSLIB.UpdateLevel, DELETE SYSLIB.DeleteLevel, SELECT SYSLIB.SelectLevel
These values correspond to the constraint value assigned to each row in the Classification_Level column defined in the AIRCRAFTS table
A constraint value can be assigned to a user or profile
User
Profile

Row Level Security Implementation Process

1. Create the security classifications that define security labels for users and data rows. See Defining Security Labels for Users and Rows.
2. Create user-defined functions to define and enforce row level security restrictions. Creating Row Level Security UDFs.
3. Grant the necessary administrator privileges for working with row level security constraints. See Granting Security Constraint Administrative Privileges.
4. Create security constraint objects. See Working with Security Constraints.
5. Assign security constraints and constraint values to database users. See Working with Constraint Assignments.
6. Create/Alter tables to define security constraint columns. See Working with Security Constraint Columns.
7. Assign constraint OVERRIDE privileges to users who need to bypass the enforcement of security constraints. See Working with Constraint OVERRIDE Privileges.

8. Evaluate database objects and processes that interface with RLS tables, and where necessary, rework them to ensure conformance with RLS requirements. See Working with Row Level Security Effects.
9. Learn how the system derives the session constraint values under various conditions, and how to set alternate values. See Determining the Session Constraint Values.
10. Enable logging of user attempts to access RLS tables, views, and administrative functions. Using Access Logging with Row Level Security.
11. Access system tables and views that contain security constraint information. See About Constraint Related System Tables and Views.
12. Examples are provided on an external website. See Examples - Row Level Security.

About Security Labels

You must set up a system of labels for each security classification category you want to use in defining user access levels and row access requirements.

A security classification system consists of:

- The name of the classification.
- The valid labels for use in classification, where each label is a name:value pair

The labels within a classification system may represent a value hierarchy, or they may be a series of compartments with no hierarchical relationship, for example:

- Security clearance (hierarchical): Top Secret, Secret, Classified, Unclassified
- User function (non-hierarchical): Administrator, Programmer, Batch User, End User
- Division/location (non-hierarchical): Canada, China, France, Germany, United States

Each classification system is the basis for:

- A security CONSTRAINT object, which defines a set of applicable access restrictions
- A security constraint column, which apply the restrictions defined in the corresponding CONSTRAINT object to each table in which the column appears

Defining Security Labels for Users and Rows

Before implementing row level security, you should define the security classification systems and associated labels required to support your site security policy.

1. Define each classification system and identify the labels in the system.

   Each system is the basis for a security CONSTRAINT object, which defines a set of access controls. Each user can be assigned up to 6 hierarchical and 2 non-hierarchical constraints.

2. For each table requiring RLS protection, determine which of the classification system (security constraints) should apply to the range of users who access the table.

   A table can contain up to 5 constraint columns.

3. Identify how security labels for each system should apply to table rows, and define the user access level required to perform each SQL operation (INSERT, SELECT, UPDATE, and DELETE).

You can use this analysis to help:

- Determine the level of protection required for each row
- Define the SQL access rules used in creating security constraint UDFs

- Determine which UDFs should be used in a security CONSTRAINT object

34. Implementing Teradata Secure Zones

Overview

Secure zones separate the access to data from the database administration duties in an exclusive database hierarchy inside a Teradata database system.

Teradata Secure Zones Overview

The Teradata Secure Zones feature allows you to create one or more exclusive database hierarchies, called zones, within a single Teradata database system. Access to the data in each zone and the database administration is handled separately from the Teradata database system and from other zones.

Secure zones are useful in situations where the access to data must be tightly controlled and restricted. You can also use secure zones to support some regulatory compliance requirements for the separation of data access from database administration duties.

For example, consider the following use of secure zones. Suppose you have a multinational company or conglomerate enterprise with many subsidiaries. You can create a separate zone for each of the subsidiaries. If your company has divisions in different countries, you can create separate zones for each country to restrict data access to the personnel that are citizens of that country. Your corporate personnel can manage and access data across multiple zones while the subsidiary personnel in each zone have no access to data or objects in the other zones. A system-level zone administrator can manage the subsidiary zones and object administration can be done by either corporate DBAs or zone DBAs, as required.

With Teradata Secure Zones, you can ensure the following:

- Users in one subsidiary have no access or visibility to objects in other subsidiaries.
- Corporate-level users may have access to objects across any or all subsidiaries.

Another typical scenario is the case of cloud companies that host multiple data customers as tenants. Companies that offer cloud-based database services can host multiple tenants within a single a Teradata Database system, using zones to isolate the tenants from each other as if they were running on physically segregated systems. Zone DBAs can administer the objects in their own zone as required. The tenant zones can be managed by a system-level zone administrator.

With Teradata Secure Zones, you can ensure the following:

- Users in a tenant zone have no access or visibility to objects within other zones.
- Users in a tenant zone cannot grant rights on any objects in the zone to any other users, databases, or roles of other zones within the system.

Secure Zone Objects

Zone objects are created, modified, and dropped in the same way as any other Teradata database object; an object exists only inside its own zone. Tables, triggers, and macros that are created inside a zone are zone objects. Objects such as roles and profiles, which are not qualified by database names, are only accessible inside the zone in which they are created. Security constraints are an exception. Security constraints that are created outside a zone can be assigned to zone users. Security constraints that are created inside a zone can be assigned to users who are outside the zone.

Secure Zone User Types

The following list describes the different types of users that are associated with a zone:

- zone creator

  Creates zones and assigns a user or a database as the zone root. Zone creators cannot access the objects or data in the zones that they create. Any user who has the ZONE rights with the WITH GRANT OPTION privilege can grant CREATE ZONE and DROP ZONE privileges.

  Only the zone's creator can add a root and primary DBA to a zone or drop a root and primary DBA from a zone.

  If the zone creator creates the zone with a user as root, then the zone creator must have DROP USER privilege on that user. Once the root is assigned to a zone, all privileges on the root user are revoked from the zone creator.

  If the zone creator creates the zone with a database as root, then the zone creator must have CREATE USER privilege on the database that becomes a root. Once the root is assigned to a zone, all privileges except CREATE USER privilege on the root database are revoked from the zone creator.

  A zone creator may grant zone access to users or roles that exist outside of the zone and is also responsible for revoking access to the zone.

  A zone creator must have CREATE ZONE and DROP ZONE privileges. A zone creator cannot be dropped until the zone itself is dropped.

- zone root

  The empty database or user on which the zone creator creates a zone.

  A zone creator creates the zone and associates a database or a user as its root. The zone root database or user must be empty. It cannot have any objects, users, databases, roles, or profiles associated with it. It also cannot have privileges on any other user. Similarly, no user should have any privileges on root except for the zone creator, owner of the root, and creator of the root.

  If the zone root is a database, the zone creator must subsequently assign a primary DBA to the zone. If the zone root is a user, that user automatically becomes the primary DBA for the zone.

- primary zone DBA

  A primary zone DBA acts as the zone's database administrator.

  The zone creator creates the primary zone DBA. The primary zone DBA can create zone users, databases, objects, and zone-level objects such as roles and profiles.

- zone user

  A permanent database user with privileges in a zone. A zone user is a user that is created by another user in the zone, under the hierarchy of the zone root. Zone users are created using the existing CREATE USER syntax. A zone user cannot be a zone guest of another zone.

  Only zone users can grant privileges on database objects within the zone to zone guests.

- zone guest

A zone guest is a role or user that is located outside of the zone but is granted privileges to create and access objects in the zone where he is a guest. A zone can have many zone guests and a user or a role can be a guest of more than one zone.

Zone guests cannot grant privileges on zone objects to other users.

To make an external LDAP user a zone guest, the zone creator can use the GRANT ZONE syntax to grant zone access privilege to an external role. External users that log on with that role are able to access the zone objects that they have privileges on.

Only the zone users can grant privileges on database objects in a zone to zone guests. Zone users cannot grant privileges to zone guests with the WITH GRANT OPTION privilege.

Zone guests with the required privileges can create users, databases, and TVM objects inside the zone but they cannot add another guest to the zone.

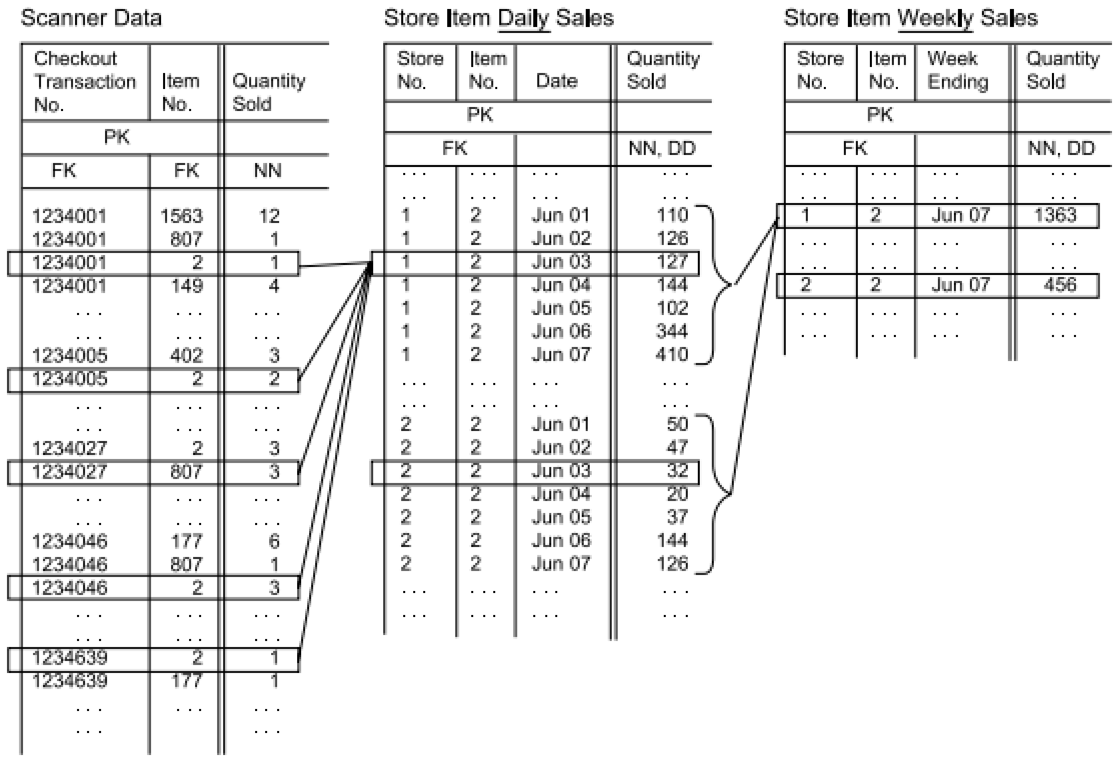
Zone guests can create views, triggers, and macros on the zone objects in their perm space

35. Usage Considerations: Summary Data and Detail Data

This topic examines the nature of the data you keep in your data warehouse and attempts to indicate why storing detail data is a better idea, particularly for ad hoc tactical and decision support queries and data mining explorations.

Observing the Effects of Summarization

Suppose we have an application that gathers check stand scanner data and stores it in relational tables. The raw detail data captured by the scanner includes a system-generated transaction number, codes for the individual items purchased as part of the transaction, and the number of each item purchased. The table that contains this detail data is named Scanner Data in the following graphic:


Scanner Data

Checkout Transaction No.	Item No.	Quantity Sold
PK		
FK	FK	NN
1234001	1563	12
1234001	807	1
1234001	2	1
1234001	149	4
...	...	...
...	...	...
1234005	402	3
1234005	2	2
...	...	...
...	...	...
1234027	2	3
1234027	807	3
...	...	...
...	...	...
1234046	177	6
1234046	807	1
1234046	2	3
...	...	...
...	...	...
1234639	2	1
1234639	177	1
...	...	...
...	...	...

Store Item Daily Sales

Store No.	Item No.	Date	Quantity Sold
PK			
FK			NN, DD
...	...	...	...
...	...	...	...
1	2	Jun 01	110
1	2	Jun 02	126
1	2	Jun 03	127
1	2	Jun 04	144
1	2	Jun 05	102
1	2	Jun 06	344
1	2	Jun 07	410
...	...	...	...
...	...	...	...
2	2	Jun 01	50
2	2	Jun 02	47
2	2	Jun 03	32
2	2	Jun 04	20
2	2	Jun 05	37
2	2	Jun 06	144
2	2	Jun 07	126
...	...	...	...
...	...	...	...

Store Item Weekly Sales

Store No.	Item No.	Week Ending	Quantity Sold
PK			
FK			NN, DD
...	...	...	...
...	...	...	...
1	2	Jun 07	1363
...	...	...	...
...	...	...	...
2	2	Jun 07	456
...	...	...	...
...	...	...	...

The middle table, store_item_daily_sales, illustrates a first level summary of the data in scanner_data. Notice that where we knew which items sold at which store at which time of day in scanner_data, now we only know the quantity of an item sold for an entire business day. The clarity of the detail data has been replaced by a more fuzzy snapshot. Potential insights have been lost.

The right most table, store_item_weekly_sales, illustrates a further abstraction of the detail data. Now all we have is the quantity of each item sold by each store for an entire week. Still fewer insights can be garnered from the data.

Of course, the data could be further abstracted. Summarization can occur at many levels. The important lesson to be learned from this study is that summaries hide valuable information. Worse still, it is not possible to reverse summarize the data in order to regain the original detail. The sharper granularity is lost forever.

Consider this simple, and highly logical, query that an analyst might ask of the sales data: How effective was the mid-week promotion we ran on sales for an item on Tuesday and Wednesday? If the only data available for analysis is a unit volume by week entity, then it is not possible to answer the question. The answer to the question is to be found in the detail, and the analyst has no way to determine the effectiveness of the promotion.

Other basic questions that cannot be answered by summary data include the following:

- What is the daily sales pattern for item 2 at any given store?
- When a customer purchases item 2, what other items are most frequently purchased with it?
- What is the profile of a customer who purchases item 2?

Information Value of Summary Data

The information value of summary data is extremely limited. As we have seen, summary data cannot answer questions about daily sales patterns by store, nor can it reveal what additional purchases were made in the same market basket, nor can it tell you anything at all about the individual customer who made the purchase.

What summary data can provide is summary answers and nothing more. This puts you in the position of always being reactive rather than proactive. Two classic retail dilemmas posed by this summary-only situation indicate that both extremes of a given problem can be caused by only having access to summary data:

- An out-of-stock situation has only been discovered after it is too late to remedy the problem.
- There is too much stock on hand, forcing an unplanned price reduction promotion to eliminate the unwanted inventories.

36. Teradata MultiLoad

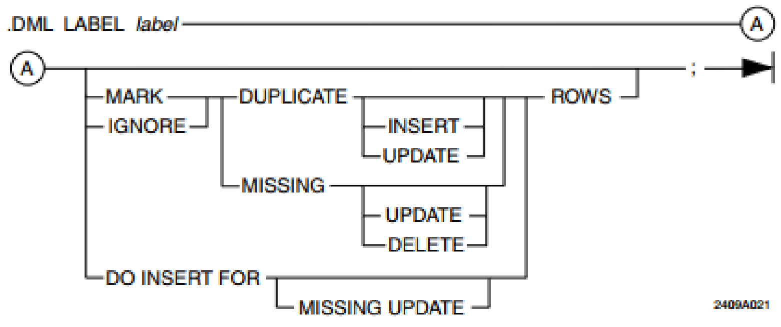
DML LABEL

Purpose

The DML LABEL command defines a label and error-treatment options for one or more immediately following INSERT, UPDATE, and DELETE statements.

Note: When using both UPDATE and INSERT statements, the resulting operation is referred to as an upsert.

Syntax



where:

Syntax Element	Description
DO INSERT FOR	An *upsert* may be implemented by subsequent UPDATE and INSERT statements for: • ROWS • MISSING UPDATE ROWS
label	The unique name of the label that is used for the immediately following set of one or more INSERT, UPDATE, or DELETE statements The *label* name must obey the same construction rules as Teradata SQL column names. The *label* name in the APPLY clause of an IMPORT command can be referenced.

Syntax Element	Description
MARK or IGNORE	Teradata Multiload either places (MARK) or does not place (IGNORE) rows in the *tname4* error table for the type of entry specified: • DUPLICATE • DUPLICATE INSERT • DUPLICATE UPDATE • MISSING • MISSING UPDATE • MISSING DELETE MARK/IGNORE DUPLICATE ROWS has no effect if the table is a multiset table (which allows duplicate rows). IGNORE DUPLICATE ROWS has no effect if the table has a unique primary index. Since a duplicate row implies a uniqueness violation in this case, the row is logged to the uniqueness violation table. In the case of an *upsert* operation, both the insert and update portions must fail for an error to be recorded. In this case, the mark rows for the missing update operations then have nulls for the target table columns. If either INSERT or UPDATE with DUPLICATE is specified, then the MARK or IGNORE specification applies to both insert and update operations. Similarly, if either UPDATE or DELETE with MISSING is not specified, then the MARK or IGNORE specification applies to both update and delete operations. **Note:** MARK is the default for all actions except MISSING UPDATE for an *upsert* operation.

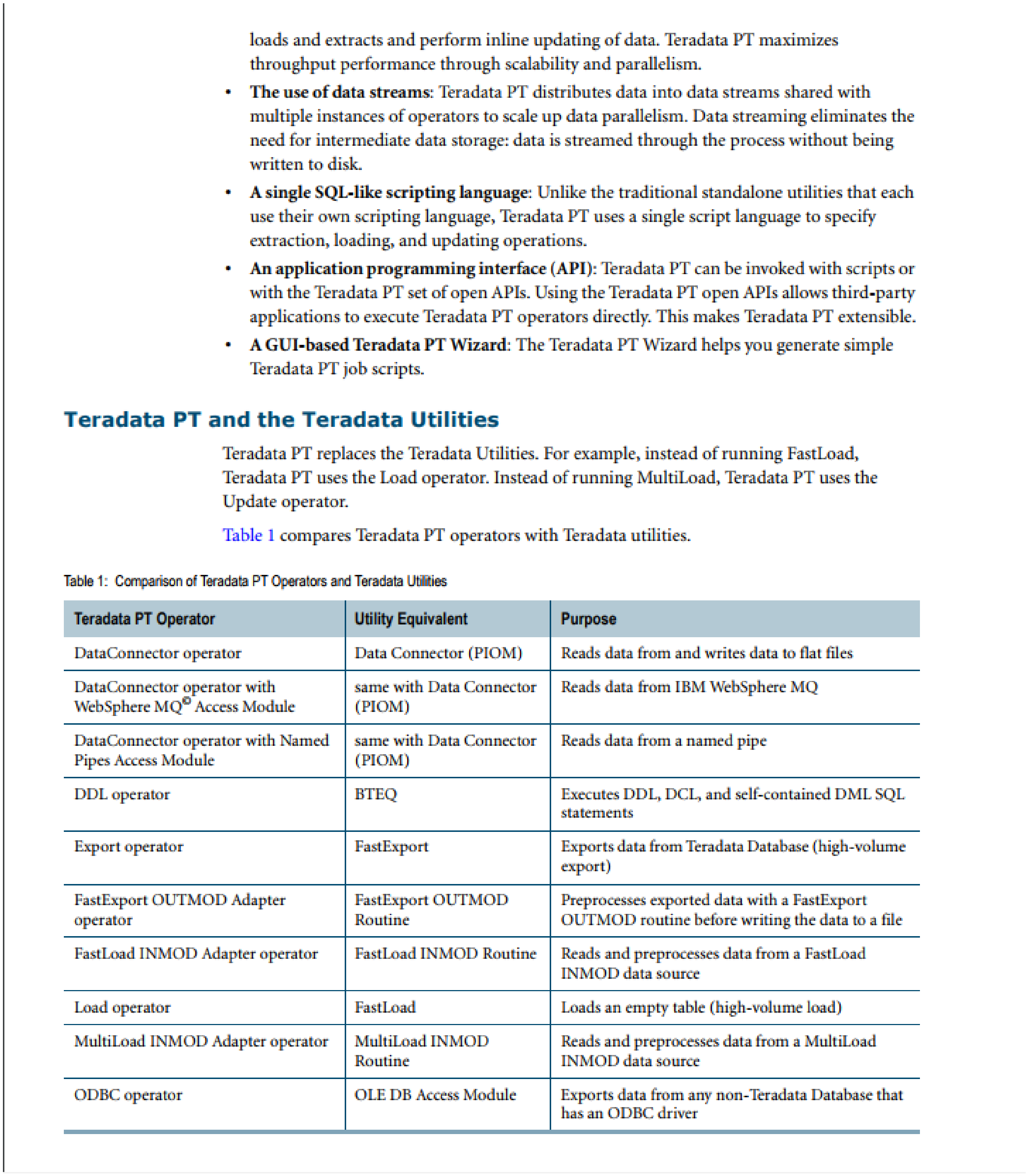
Usage Notes

Table 39 describes the things to consider when using the DML LABEL command.

Table 39: DML LABEL Considerations

Topic	Usage Notes
Bypassing the Duplicate Row Check	Duplicate row checking is not performed if the table is a multiset table (which allows duplicate rows) or if the table has a unique primary index (the uniqueness test takes the place of the duplicate row check).
DO INSERT FOR ROWS Option	By following the rules for upsert operations, a number of uses for the DO INSERT ROWS option can be found. With an upsert operation, Teradata Multiload needs only one pass of the data to both: • Update the rows that need to be updated. • Insert the rows that need to be inserted. The alternative would be to either: • Presort the data for the update and insert operations. • First use an UPDATE statement with all of the data, and then use an INSERT statement with the data that failed the update operation.

37. Teradata Parallel Transporter User Guide



loads and extracts and perform inline updating of data. Teradata PT maximizes throughput performance through scalability and parallelism.

- **The use of data streams**: Teradata PT distributes data into data streams shared with multiple instances of operators to scale up data parallelism. Data streaming eliminates the need for intermediate data storage: data is streamed through the process without being written to disk.
- **A single SQL-like scripting language**: Unlike the traditional standalone utilities that each use their own scripting language, Teradata PT uses a single script language to specify extraction, loading, and updating operations.
- **An application programming interface (API)**: Teradata PT can be invoked with scripts or with the Teradata PT set of open APIs. Using the Teradata PT open APIs allows third-party applications to execute Teradata PT operators directly. This makes Teradata PT extensible.
- **A GUI-based Teradata PT Wizard**: The Teradata PT Wizard helps you generate simple Teradata PT job scripts.

Teradata PT and the Teradata Utilities

Teradata PT replaces the Teradata Utilities. For example, instead of running FastLoad, Teradata PT uses the Load operator. Instead of running MultiLoad, Teradata PT uses the Update operator.

Table 1 compares Teradata PT operators with Teradata utilities.

Table 1: Comparison of Teradata PT Operators and Teradata Utilities

Teradata PT Operator	Utility Equivalent	Purpose
DataConnector operator	Data Connector (PIOM)	Reads data from and writes data to flat files
DataConnector operator with WebSphere MQ© Access Module	same with Data Connector (PIOM)	Reads data from IBM WebSphere MQ
DataConnector operator with Named Pipes Access Module	same with Data Connector (PIOM)	Reads data from a named pipe
DDL operator	BTEQ	Executes DDL, DCL, and self-contained DML SQL statements
Export operator	FastExport	Exports data from Teradata Database (high-volume export)
FastExport OUTMOD Adapter operator	FastExport OUTMOD Routine	Preprocesses exported data with a FastExport OUTMOD routine before writing the data to a file
FastLoad INMOD Adapter operator	FastLoad INMOD Routine	Reads and preprocesses data from a FastLoad INMOD data source
Load operator	FastLoad	Loads an empty table (high-volume load)
MultiLoad INMOD Adapter operator	MultiLoad INMOD Routine	Reads and preprocesses data from a MultiLoad INMOD data source
ODBC operator	OLE DB Access Module	Exports data from any non-Teradata Database that has an ODBC driver

Table 1: Comparison of Teradata PT Operators and Teradata Utilities (continued)

Teradata PT Operator	Utility Equivalent	Purpose
OS Command operator	Client host operating system	Executes host operating system commands
SQL Inserter operator	BTEQ	Inserts data into a Teradata table using SQL protocol
SQL Selector operator	BTEQ	Selects data from a Teradata table using SQL protocol
Stream operator	TPump	Continuously loads Teradata tables using SQL protocol
Update operator	MultiLoad	Updates, inserts, and deletes rows

Platforms

For a detailed list of supported platform environments for Teradata PT, as well as other Teradata Tools and Utilities *Teradata Tools and Utilities ##.# Supported Platforms and Product Versions*. For information about how to access this and other related publications "Supported Releases" on page 3.

Note: The 16.00 Teradata PT products are compiled on the AIX 6.1 using the xlC version 11 and must run on the AIX machine with the same level or higher C++ runtime library version and C runtime library version 6.1.

Compatibilities

Observe the following information about job script compatibility.

- Scripts written for the former Teradata Warehouse Builder work with Teradata PT *without* modification, but Teradata Warehouse Builder scripts cannot employ new Teradata PT features. Teradata recommends that all *new* scripts be written using the Teradata PT scripting language.
- Scripts written for Teradata standalone utilities are *incompatible* with Teradata PT. Teradata recommends that existing standalone utility scripts be reworked using Teradata PT scripting language. Contact Professional Services for help.

Other Vendors

ETL vendor products can be used with Teradata PT to generate scripts for load operations or to make API calls:

- **Extract, Transform, and Load (ETL)** vendors add value by performing:
  - Data extractions and transformations prior to loading Teradata Database. Teradata PT provides the ability to condition, condense, and filter data from multiple sources through the Teradata PT SELECT statement.
  - Data extractions and loading, but leaving all the complex SQL processing of data to occur inside the Teradata Database itself. Like ETL vendors, Teradata PT can condition, condense, and filter data from multiple sources into files.

- **The Teradata PT API** provides additional advantages for third-party ETL/ELT vendors. For more information, see the *Teradata Parallel Transporter Application Programming Interface Programmer Guide*.