Informix Product Family
Informix
Version 11.50

IBM Informix Data Warehouse Guide

SC27-3835-00

Informix Product Family
Informix
Version 11.50

IBM Informix Data Warehouse Guide

SC27-3835-00

Note

Before using this information and the product it supports, read the information in "Notices" on page B-1.

Edition

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Introduction

In this introduction

This introduction provides an overview of the information in this publication and describes the conventions that this publication uses.

About this publication

This publication provides reference material for IBM® Informix®. This publication contains comprehensive information about designing dimensional databases, using the Informix Warehouse Feature, and the IBM Informix Warehouse Accelerator, which is used to quickly process data warehouse queries.

This section discusses the intended audience for this publication.

Types of users

This publication is written for the following users:
- Database administrators
- System administrators
- Performance engineers
- Application developers

This publication is written with the assumption that you have the following background:
- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Some experience working with relational and dimensional databases or exposure to database concepts
- Some experience with database server administration, operating-system administration, network administration, or application development

You can access the Informix information centers, as well as other technical information such as technotes, white papers, and IBMRedbooks® publications online at http://www.ibm.com/software/data/sw-library/.

Software dependencies

This publication is written with the assumption that you are using IBM Informix or IBM Informix Dynamic Server with J/Foundation, Version 11.50, as your database server.

Assumptions about your locale

IBM Informix products can support many languages, cultures, and code sets. All the information related to character set, collation, and representation of numeric data, currency, date, and time is brought together in a single environment, called a Global Language Support (GLS) locale.

The examples in this publication are written with the assumption that you are using the default locale, **en_us.8859-1**. This locale supports U.S. English format conventions for date, time, and currency. In addition, this locale supports the ISO 8859-1 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use nondefault characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules of character data, you need to specify the appropriate nondefault locale.

For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the *IBM Informix GLS User's Guide*.

Demonstration databases

The DB-Access utility includes one or more demonstration databases that you can use to learn and test with. After you add, delete, or change the data and scripts that are in the database, you can re-initialize the database to its original condition.

The demonstration databases are:

- The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. Many examples in IBM Informix publications are based on the **stores_demo** database.
- The **sales_demo** database provides an example of a simple data-warehousing environment and works in conjunction with the stores_demo database. The scripts for the sales_demo database create new tables and add extra rows to the items and orders tables of stores_demo database.
- The **superstores_demo** database illustrates an object-relational schema. The **superstores_demo** database contains examples of extended data types, type and table inheritance, and user-defined routines.

For information about how to create and populate the demonstration databases, see the *IBM Informix DB-Access User's Guide*. For descriptions of the databases and their contents, see the *IBM Informix Guide to SQL: Reference*.

The scripts that you use to install the demonstration databases reside in the **$INFORMIXDIR/bin** directory on UNIX and in the **%INFORMIXDIR%\bin** directory on Windows.

What's new in Data Warehouse Guide for Informix, Version 11.50

This publication includes information about new features and changes in existing functionality.

The following changes and enhancements are relevant to this publication. For a comprehensive list of all new features for this release, see the *IBM Informix Getting Started Guide*.

The following table lists the new features for Version 11.50.

Table 1. What's New in IBM Informix Data Warehouse Guide for Version 11.50xC9.

Overview	Reference
New guide for data warehousing To coincide with IBM's increased focus on improving data warehousing, this new guide has been created. The dimensional database content that was in the *IBM Informix Database Design and Implementation Guide* has been moved into this guide and the topics have been updated.	
New editions and product names IBM Informix Dynamic Server editions were withdrawn and new Informix editions are available. Some products were also renamed. The publications in the Informix library pertain to the following products: • IBM Informix database server, formerly known as IBM Informix Dynamic Server (IDS) • IBM OpenAdmin Tool (OAT) for Informix, formerly known as OpenAdmin Tool for Informix Dynamic Server (IDS) • IBM Informix SQL Warehousing Tool, formerly known as Informix Warehouse Feature	For more information about the Informix product family, go to http://www.ibm.com/software/data/informix/.

Example code conventions

Examples of SQL code occur throughout this publication. Except as noted, the code is not specific to any single IBM Informix application development tool.

If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement. If you are using DB–Access, you must delimit multiple statements with semicolons.

Tip: Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the documentation for your product.

Additional documentation

Documentation about this release of IBM Informix products is available in various formats.

You can access or install the product documentation from the Quick Start CD that is shipped with Informix products. To get the most current information, see the Informix information centers at ibm.com®. You can access the information centers and other Informix technical information such as technotes, white papers, and IBM Redbooks publications online at http://www.ibm.com/software/data/sw-library/.

Compliance with industry standards

IBM Informix products are compliant with various standards.

IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.

The IBM Informix Geodetic DataBlade® Module supports a subset of the data types from the *Spatial Data Transfer Standard (SDTS)—Federal Information Processing Standard 173*, as referenced by the document *Content Standard for Geospatial Metadata*, Federal Geographic Data Committee, June 8, 1994 (FGDC Metadata Standard).

IBM Informix Dynamic Server (IDS) Enterprise Edition, Version 11.50 is certified under the Common Criteria. For more information, see *Common Criteria Certification: Requirements for IBM Informix Dynamic Server*, which is available at http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss?CTY=US&FNC=SRX&PBL=SC23-7690-00.

Chapter 1. Dimensional databases

A dimensional database is a relational database that uses a *dimensional data model* to organize data. This model uses fact tables and dimension tables in a star or snowflake schema.

A dimensional database is the optimal type of database for data warehousing.

The availability and reliability of the Informix database server includes a full active-active cluster solution for high availability and low cost scalability. You can use Informix to manage workload distribution across multiple read-only or full-transaction nodes. You can dynamically add different types of nodes into your cluster environment to scale out or increase availability in the most demanding environments.

Warehouse workloads have the flexibility to work on the same database with operational data, running real-time on a separate node in the cluster. Data can also be replicated in real-time using Enterprise Replication, or copied to a separate data warehouse server. With Informix, you have the flexibility to design the system to meet your needs and to make the most of your existing infrastructure.

Overview of data warehousing

Data warehouse databases provide a decision support system (DSS) environment in which you can evaluate the performance of an entire enterprise over time.

In the broadest sense, the term *data warehouse* is used to refer to a database that contains very large stores of historical data. The data is stored as a series of snapshots, in which each record represents data at a specific time. By analyzing these snapshots you can make comparisons between different time periods. You can then use these comparisons to help make important business decisions.

Data warehouse databases are optimized for data retrieval. The duplication or grouping of data, referred to as *database denormalization*, increases query performance and is a natural outcome of the dimensional design of the data warehouse. By contrast, traditional online transaction processing (OLTP) databases automate day-to-day transactional operations. OLTP databases are optimized for data storage and strive to eliminate data duplication. Databases that achieve this goal are referred to as *normalized* databases.

An enterprise data warehouse (EDW) is a data warehouse that services the entire enterprise. An *enterprise data warehousing environment* can consist of an EDW, an operational data store (ODS), and physical and virtual data marts.

A data warehouse can be implemented in several different ways. You can use a single data management system, such as Informix, for both transaction processing and business analytics. Or, depending on your system workload requirements, you can build a data warehousing environment that is separate from your transactional processing environment.

Informix uses the umbrella terms *data warehousing* and *data warehousing environment* to encompass any of the following forms that you might use to store your data:

Data warehouse

A database that is optimized for data retrieval to facilitate reporting and analysis. A data warehouse incorporates information about many subject areas, often the entire enterprise. Typically you use a dimensional data model to design a data warehouse. The data is organized into dimension tables and fact tables using star and snowflake schemas. The data is denormalized to improve query performance. The design of a data warehouse often starts from an analysis of what data already exists and how to collected in such a way that the data can later be used. Instead of loading transactional data directly into a warehouse, the data is often integrated and transformed before it is loaded into the warehouse.

The primary advantage of a data warehouse is that it provides easy access to and analysis of vast stores of information on many subject areas.

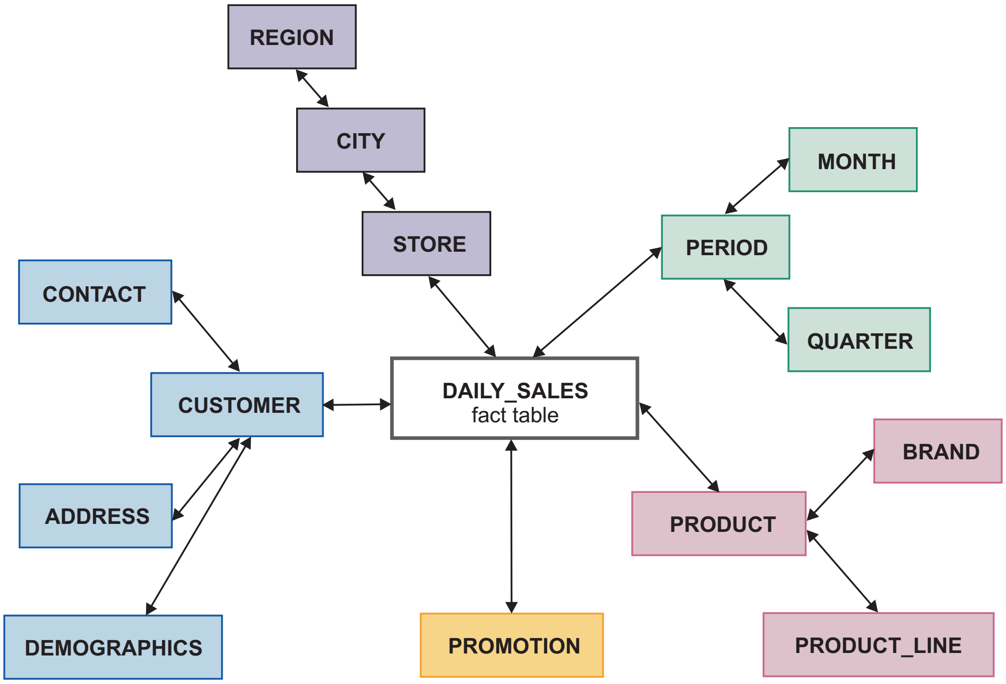

Figure 1-1. A sample snowflake schema which has the DAILY_SALES table as the fact table.

Data mart

A database that is oriented towards one or more specific subject areas of a business, such as tracking inventories or transactions, rather than an entire enterprise. A data mart is used by individual departments or groups. Like a data warehouse, you typically use a dimensional data model to build a data mart. For example the data mart might use a single star schema comprised of one fact table and several dimension tables. The design of a data mart often starts with an analysis of what data the user needs rather than focusing on the data that already exists.

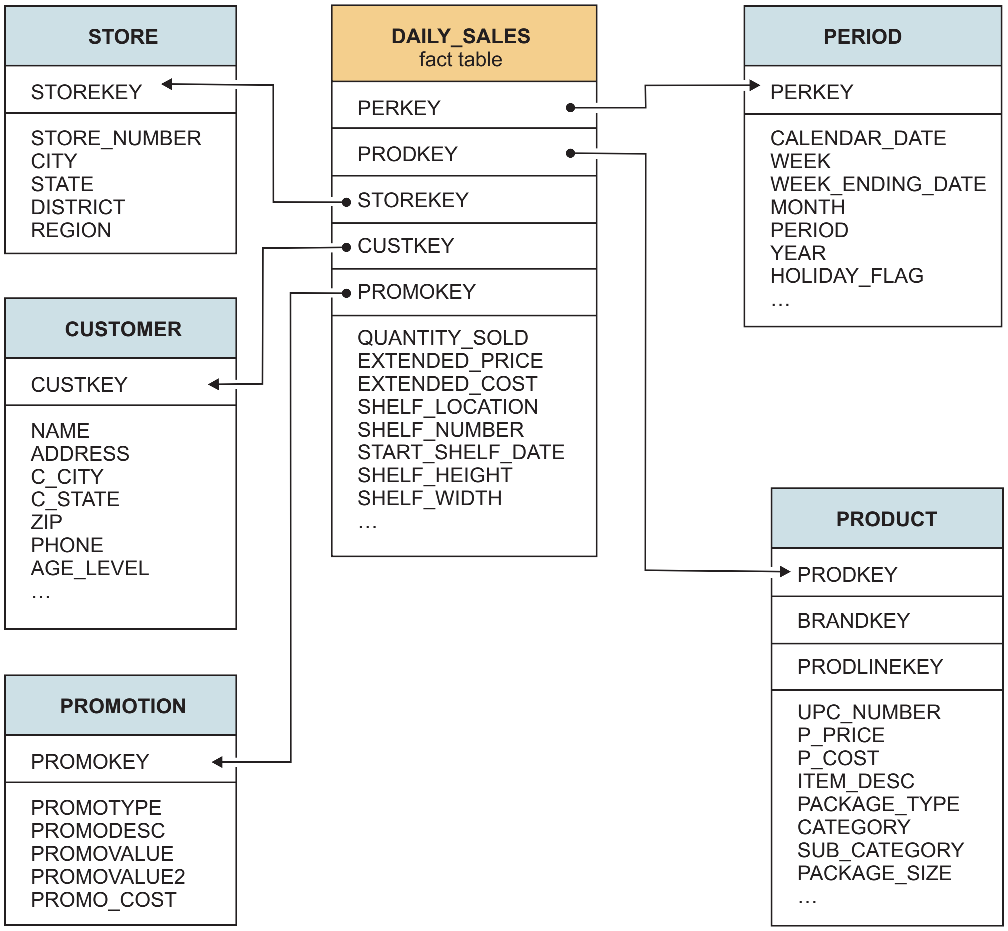

Figure 1-2. A data mart with the DAILY_SALES fact table

Operational data store

A subject-oriented system that is optimized for looking up one or two records at a time for decision making. An operational data store (ODS) is a hybrid form of data warehouse that contains timely, current, integrated information. Including the ODS in the data warehousing environment enables access to more current data more quickly, particularly if the data warehouse is updated by one or more batch processes rather than updated continuously. The data typically is of a higher level granularity than the transaction. You can use an ODS for clerical, day-to-day decision making. This data can serve as the common source of data for data warehouses.

Why build a dimensional database?

In a data warehousing environment, the relational databases need to be optimized for data retrieval and tuned to support the analysis of business trends and projections.

This type of informational processing is known as online analytical processing (OLAP) or decision support system (DSS) processing. OLAP is also the term that database designers use to describe a dimensional approach to informational processing.

A dimensional database needs to be designed to support queries that retrieve a large number of records and that summarize data in different ways. A dimensional database tends to be subject oriented and aims to answer questions such as, “What products are selling well?” “At what time of year do certain products sell best?” “In what regions are sales weakest?”

In a dimensional data model, the data is represented as either facts or dimensions. A *fact* is typically numeric piece of data about a transaction, such as the number of items ordered. A *dimension* is the reference information about the numeric facts, such as the name of the customer. Any new data that you load into the dimensional database is usually updated in a batch, often from multiple sources.

Relational databases are optimized for online transaction processing (OLTP) are designed to meet the day-to-day operational needs of the business. OLTP systems tend to organize data around specific processes, such as order entry. The database performance is tuned for those operational needs by using a *normalized data model* which stores data by using database normalization rules. Consequently, the database can retrieve a small number of records very quickly.

Some of the advantages of the dimensional data model are that data retrieval tends to be very quick and the organization of the data warehouse is easier for users to understand and use.

If you attempt to use a database that is designed for OLTP as your data warehouse, query performance will be very slow and it will be difficult to perform analysis on the data.

The following table summarizes the key differences between OLTP and OLAP databases:

Normalized database (OLTP)	Dimensional database (OLAP)
Data is atomized	Data is summarized
Data is current	Data is historical
Processes one record at a time	Processes many records at a time
Process oriented	Subject oriented
Designed for highly structured repetitive processing	Designed for highly unstructured analytical processing

Many of the problems that businesses attempt to solve are multidimensional in nature. For example, SQL queries that create summaries of product sales by region, region sales by product, and so on, might require hours of processing on an OLTP database. However, a dimensional database could process the same queries in a fraction of the time.

Besides the characteristic schema design differences between OLTP and OLAP databases, the query optimizer typically should be tuned differently for these two types of tasks. For example, in OLTP operations, the OPTCOMPIND setting (as specified by the environment variable or by the configuration parameter of that name) should typically be set to zero, to favor nested-loop joins. OLAP operations, in contrast, tend to be more efficient with an OPTCOMPIND setting of 2 to favor hash-join query plans. For more information, see the **OPTCOMPIND** environment variable and the OPTCOMPIND configuration parameter. See the *IBM Informix Performance Guide* for additional information about OPTCOMPIND, join methods, and the query optimizer.

IBM Informix also supports the SET ENVIRONMENT OPTCOMPIND statement to change OPTCOMPIND setting dynamically during sessions in which both OLTP and OLAP operations are required. See the *IBM Informix Guide to SQL: Syntax* for more information about the SET ENVIRONMENT statement of SQL.

Informix is designed to help businesses better leverage their existing information assets as they move into an on-demand business environment. In this type of environment, mission-critical database management applications typically require combination systems. The applications need both online transaction processing (OLTP), and batch and decision support systems (DSS), including online analytical processing (OLAP).

Related reference

OPTCOMPIND environment variable (SQL Reference)

OPTCOMPIND Configuration Parameter (Administrator's Reference)

What is dimensional data?

Traditional relational databases, such as OLTP databases, are organized around a list of records. Each record contains related information that is organized into attributes (fields). The **customer** table of the **stores_demo** demonstration database, which includes fields for name, company, address, phone, and so forth, is a typical example. While this table has several fields of information, each row in the table pertains to only one customer. If you wanted to create a two-dimensional matrix with customer name and any other field, for example, phone number), you would realize that there is only a one-to-one correspondence. The following table is an example of a database table with fields that have only a one-to-one correspondence.

Table 1-1. A table with a one-to-one correspondences between fields

Customer	Phone number --->		
Ludwig Pauli	408-789-8075	----------------	----------------
Carole Sadler	----------------	415-822-1289	----------------
Philip Currie	----------------	----------------	414-328-4543

You could put any combination of fields from the preceding **customer** table in this matrix, but you would always end up with a one-to-one correspondence, which shows that this table is not multidimensional and would not be well suited for a dimensional database.

However, consider a relational table that contains more than a one-to-one correspondence between the fields of the table. Suppose you create a table that contains sales data for products sold in each region of the country. For simplicity, the company has three products that are sold in three regions. The following table shows how you might store this data in a table, using a normalized data model. This table lends itself to multidimensional representation because it has more than one product per region and more than one region per product.

Table 1-2. A simple table with a many-to-many correspondence

Product	Region	Unit Sales
Football	East	2300
Football	West	4000

Table 1-2. A simple table with a many-to-many correspondence (continued)

Product	Region	Unit Sales
Football	Central	5600
Tennis racket	East	5500
Tennis racket	West	8000
Tennis racket	Central	2300
Baseball	East	10000
Baseball	West	22000
Baseball	Central	34000

Although this data can be forced into the three-field relational table, the data fits more naturally into the two-dimensional matrix in the following table. This matrix better represents the many-to-many relationship of product and region data listed above.

Table 1-3. A simple two-dimensional example

	Region	Central	East	West
Product	Football	5600	2300	4000
	Tennis Racket	2300	5500	8000
	Baseball	34000	10000	22000

The performance advantages of the dimensional model over the normalized model can be great. A dimensional approach simplifies access to the data that you want to summarize or compare. For example, using the dimensional model to query the number of products sold in the West, the database server finds the **West** column and calculates the total for all row values in that column. To perform the same query on the normalized table, the database server has to search and retrieve each row where the **Region** column equals 'West' and then aggregate the data. In queries of this kind, the dimensional table can total all values of the **West** column in a fraction of the time it takes the relational table to find all the 'West' records.

Related concepts

The stores_demo Database (SQL Reference)

Chapter 2. Design a dimensional data model

To build a dimensional database, you start by designing a dimensional data model for your business.

You will learn how a dimensional model differs from a transactional model, what fact tables and dimension tables are and how to design them effectively. You will learn how to analyze the business processes in your organization where data is gathered and use that analysis to design a model for your dimensional data.

IBM Informix includes several demonstration databases that are the basis for many examples in Informix publications, including examples in the *IBM Informix Data Warehouse Guide*. The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. You will use SQL and the data in the **stores_demo** database to populate a new dimensional database. The dimensional database is based on the simple dimensional data model that you learned about.

To understand the concepts of dimensional data modeling, you should have a basic understanding of SQL and relational database theory. This section provides only a summary of data warehousing concepts and describes a simple dimensional data model.

Related concepts

The stores_demo Database (SQL Reference)

Chapter 4, “Performance tuning dimensional databases,” on page 4-1

Related reference

Chapter 3, “Implement a dimensional database,” on page 3-1

Concepts of dimensional data modeling

To build a dimensional database, you start with a dimensional data model. The dimensional data model provides a method for making databases simple and understandable. You can conceive of a dimensional database as a database *cube* of three or four dimensions where users can access a slice of the database along any of its dimensions. To create a dimensional database, you need a model that lets you visualize the data.

Suppose your business sells products in different markets and you want to evaluate the performance over time. It is easy to conceive of this business process as a cube of data, which contains dimensions for time, products, and markets. The following figure shows this dimensional model. The various intersections along the lines of the cube would contain the *measures* of the business. The measures correspond to a particular combination: product, market, and time data.

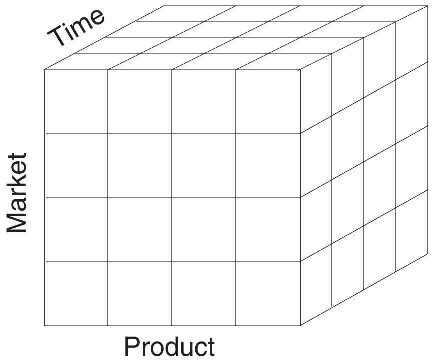

Figure 2-1. A dimensional model of a business that has time, product, and market dimensions

Another name for the dimensional model is the *star schema*. The database designers use this name because the diagram for this model looks like a star with one central table around which a set of other tables are displayed. The central table is the only table in the schema with multiple joins connecting it to all the other tables. This central table is called the *fact table* and the other tables are called *dimension tables*. The dimension tables all have only a single join that attaches them to the fact table, regardless of the query. The following figure shows a simple dimensional model of a business that sells products in different markets and evaluates business performance over time.

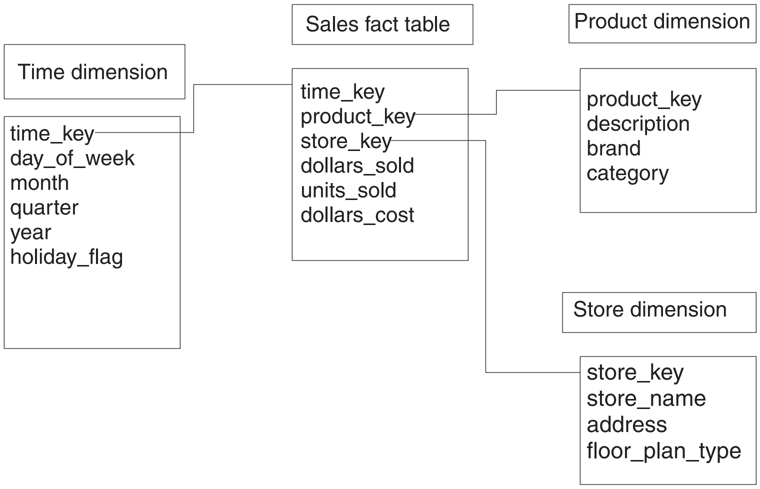

Figure 2-2. A typical dimensional model

The fact table

The fact table stores the measures of the business and points to the key value at the lowest level of each dimension table. The *measures* are quantitative or factual data about the subject.

The measures are generally numeric and correspond to the "how much" or "how many" aspects of a question. Examples of measures are price, product sales, product inventory, revenue, and so forth. A measure can be based on a column in a table or it can be calculated.

The following table shows a fact table whose measures are sums of the units sold, the revenue, and the profit for the sales of that product to that account on that day.

Table 2-1. A fact table with sample records

Product Code	Account code	Day code	Units sold	Revenue	Profit
1	5	32104	1	82.12	27.12
3	17	33111	2	171.12	66.00
1	13	32567	1	82.12	27.12

Before you design a fact table, you must determine the *granularity* of the fact table. The granularity corresponds to how you define an individual low-level record in that fact table. The granularity might be the individual transaction, a daily snapshot, or a monthly snapshot. The fact table above contains one row for every product sold to each account each day. Thus, the granularity of the fact table is expressed as *product by account by day*.

Dimensions of the data model

A *dimension* represents a single set of objects or events in the real world. Each dimension that you identify for the data model gets implemented as a dimension table. Dimensions are the qualifiers that make the measures of the fact table meaningful, because they answer the what, when, and where aspects of a question. For example, consider the following business questions, for which the dimensions are italicized:

- What *accounts* produced the highest revenue last *year*?
- What was our profit by *vendor*?
- How many units were sold for each *product*?

In the preceding set of questions, revenue, profit, and units sold are measures (not dimensions), as each represents quantitative or factual data.

Dimension elements

A dimension can define multiple *dimension elements* for different levels of summation.

For example, all the elements that relate to the structure of a sales organization might comprise one dimension. The following figure shows the dimension elements that the **Accounts** dimension defines.

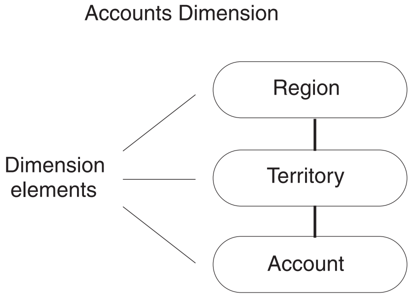

Figure 2-3. Dimension elements in the accounts dimension

Dimensions are made up of hierarchies of related elements. Because of the hierarchical aspect of dimensions, users are able to construct queries that access data at a higher level (*roll up*) or lower level (*drill down*) than the previous level of detail. The figure shows the hierarchical relationships of the dimension elements:

- The account elements roll up to the territory elements

- The territory elements roll up to the region elements

Users can query at different levels of the dimension, depending on the data they want to retrieve. For example, users might perform a query against all regions and then drill down to the territory or account level for detailed information.

Dimension elements are usually stored in the database as numeric codes or short character strings to facilitate joins to other tables.

Each dimension element can define multiple dimension attributes, in the same way dimensions can define multiple dimension elements.

Dimension attributes

A *dimension attribute* is a column in a dimension table. Each attribute describes a level of summary within a dimension hierarchy.

The dimension elements define the hierarchical relationships within a dimension table. The dimension attributes describe the dimension elements in terms that are familiar to users. The following figure shows the dimension elements and corresponding attributes of the **Account** dimension.

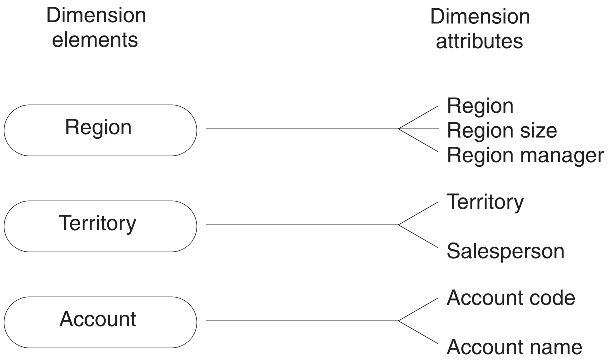

Figure 2-4. Attributes that correspond to the dimension elements

Because dimension attributes describe the items in a dimension, they are most useful when they are text.

Tip: Sometimes during the design process, it is unclear whether a numeric data field from a production data source is a measured fact or an attribute. Generally, if the numeric data field is a measurement that changes each time you sample it, the field is a fact. If field is a discretely valued description of something that is more or less constant, it is a dimension attribute.

Dimension tables

A *dimension table* is a table that stores the textual descriptions of the dimensions of the business. A dimension table contains an element and an attribute, if appropriate, for each level in the hierarchy.

The lowest level of detail that is required for data analysis determines the lowest level in the hierarchy. Levels higher than this base level store redundant data. This denormalized table reduces the number of joins that are required for a query and makes it easier for users to query at higher levels and then drill down to lower levels of detail. The term *drilling down* means to add row headers from the dimension tables to your query. The following table shows an example of a dimension table that is based on the **Account** dimension.

Table 2-2. An example of a dimension table

Acct code	Account name	Territory	Salesman	Region	Region size	Region manager
1	Javier's Mfg.	101	B. Gupta	Asia-Pacific	Over 50	T. Sent
2	TBD Sales	101	B. Gupta	Asia-Pacific	Over 50	T. Sent
3	Tariq's Wares	101	B. Gupta	Asia-Pacific	Over 50	T. Sent
4	The Golf Co.	201	S. Chiba	Asia-Pacific	Over 50	T. Sent

Building a dimensional data model

To build a dimensional data model, you need a methodology that outlines the decisions you need to make to complete the database design. This methodology uses a top-down approach because it first identifies the major processes in your organization where data is collected. An important task of the database designer is to start with the existing sources of data that your organization uses. After the processes are identified, one or more fact tables are built from each business process. The following steps describe the methodology you use to build the data model.

A dimensional database can be based on multiple business processes and can contain many fact tables. However, to focus on the concepts, the data model that this section describes is based on a single business process and has one fact table.

To build a dimensional database:

1. Choose the business processes that you want to use to analyze the subject area to be modeled.
2. Determine the granularity of the fact tables.
3. Identify dimensions and hierarchies for each fact table.
4. Identify measures for the fact tables.
5. Determine the attributes for each dimension table.
6. Get users to verify the data model.

A business process

A *business process* is an important operation in your organization that some legacy system supports. You collect data from this system to use in your dimensional database.

The business process identifies what end users are doing with their data, where the data comes from, and how to transform that data to make it meaningful. The information can come from many sources, including finance, sales analysis, market analysis, customer profiles. The following list shows different business processes you might use to determine what data to include in your dimensional database:

- Sales
- Shipments
- Inventory
- Orders
- Invoices

Summary of a business process

Suppose your organization wants to analyze customer buying trends by product line and region so that you can develop more effective marketing strategies. In this scenario, the subject area for your data model is **sales**.

After many interviews and thorough analysis of your sales business process, your organization collects the following information:

- Customer-base information has changed.

  Previously, sales districts were divided by city. Now the customer base corresponds to two regions: Region 1 for California and Region 2 for all other states.
- The following reports are most critical to marketing:
  - Monthly revenue, cost, net profit by product line from each vendor
  - Revenue and units sold by product, by region, and by month
  - Monthly customer revenue
  - Quarterly revenue from each vendor
- Most sales analysis is based on monthly results, but you can choose to analyze sales by week or accounting period (at a later date).
- A data-entry system exists in a relational database.

  To develop a working data model, you can assume that the relational database of sales information has the following properties:
  - The **stores_demo** database provides much of the revenue data that the marketing department uses.
  - The product code that analysts use is stored in the **catalog** table by the catalog number.
  - The product line code is stored in the **stock** table by the stock number. The product line name is stored as description.
  - The product hierarchies are somewhat complicated. Each product line has many products, and each manufacturer has many products.
- All the cost data for each product is stored in a flat file named **costs.lst** on a different purchasing system.
- Customer data is stored in the **stores_demo** database.

  The region information has not yet been added to the database.

An important characteristic of the dimensional model is that it uses business labels familiar to end users rather than internal tables or column names. After the analysis of the business process is completed, you should have all the information you need to create the measures, dimensions, and relationships for the dimensional data model. This dimensional data model is used to implement the **sales_demo** database that the section Chapter 3, “Implement a dimensional database,” on page 3-1 describes.

The **stores_demo** demonstration database is the primary data source for the dimensional data model that this section builds. For detailed information about the data sources that are used to populate the tables of the **sales_demo** database, see “Mapping data from data sources to the database” on page 3-3.

Determine the granularity of the fact table

After you gather all the relevant information about the subject area, the next step in the design process is to determine the granularity of the fact table.

To do this you must decide what an individual low-level record in the fact table should contain. The components that make up the granularity of the fact table correspond directly with the dimensions of the data model. Therefore, when you define the granularity of the fact table, you identify the dimensions of the data model.

How granularity affects the size of the database

The granularity of the fact table also determines how much storage space the database requires.

For example, consider the following possible granularities for a fact table:

- Product by day by region
- Product by month by region

The size of a database that has a granularity of product by day by region would be much greater than a database with a granularity of product by month by region. The database contains records for every transaction made each day as opposed to a monthly summation of the transactions. You must carefully determine the granularity of your fact table because too fine a granularity could result in an astronomically large database. Conversely, too coarse a granularity could mean the data is not detailed enough for users to perform meaningful queries against the database.

Use the business process to determine the granularity

A careful review of the information gathered from the business process should provide what you need to determine the granularity of the fact table. To summarize, your organization wants to analyze customer-buying trends by product line and region so that you can develop more effective marketing strategies.

Customer by product level granularity:

The granularity of the fact table should always represent the lowest level for each corresponding dimension.

When you review the information from the business process, the granularity for customer and product dimensions of the fact table are apparent. Customer and product cannot be reasonably reduced any further. These dimensions already express the lowest level of an individual record for the fact table. In some cases, product might be further reduced to the level of product component because a product could be made up of multiple components.

Customer by product by district level granularity:

Because the customer buying trends that your organization wants to analyze include a geographical component, you still need to decide the lowest level for the region information.

The business process indicates that in the past, sales districts were divided by city, but now your organization distinguishes between two regions for the customer base:

- Region 1 for California
- Region 2 for all other states

Nonetheless, at the lowest level, your organization still includes sales district data. The district represents the lowest level for geographical information and provides a third component to further define the granularity of the fact table.

Customer by product by district by day level granularity:

Customer-buying trends always occur over time, so the granularity of the fact table must include a time component.

Suppose your organization decides to create reports by week, accounting period, month, quarter, or year. At the lowest level, you probably want to choose a base granularity of day. This granularity allows your business to compare sales on Tuesdays with sales on Fridays, compare sales for the first day of each month, and so forth. The granularity of the fact table is now complete.

The decision to choose a granularity of day means that each record in the **time** dimension table represents a day. In terms of the storage requirements, even 10 years of daily data is only about 3,650 records, which is a relatively small dimension table.

Identify the dimensions and hierarchies

After you determine the granularity of the fact table, it is easy to identify the primary dimensions for the data model because each component that defines the granularity corresponds to a dimension.

The following figure shows the relationship between the granularity of the fact table and the dimensions of the data model.

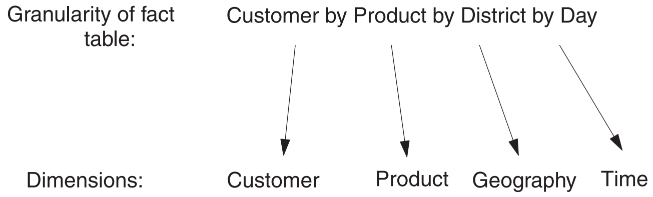
Granularity of fact table:
Customer by Product by District by Day
Dimensions:
Customer
Product
Geography
Time

Figure 2-5. The granularity of the fact table corresponds to the dimensions of the data model

With the dimensions (customer, product, geography, time) for the data model in place, the schema diagram begins to take shape.

Tip: At this point, you can add additional dimensions to the primary granularity of the fact table, where the new dimensions take on only a single value under each combination of the primary dimensions. If you see that an additional dimension violates the granularity because it causes additional records to be generated, then you must revise the granularity of the fact table to accommodate the additional dimension. For this data model, no additional dimensions need to be added.

You can now map out dimension elements and hierarchies for each dimension. The following figure shows the relationships among dimensions, dimension elements, and the inherent hierarchies.

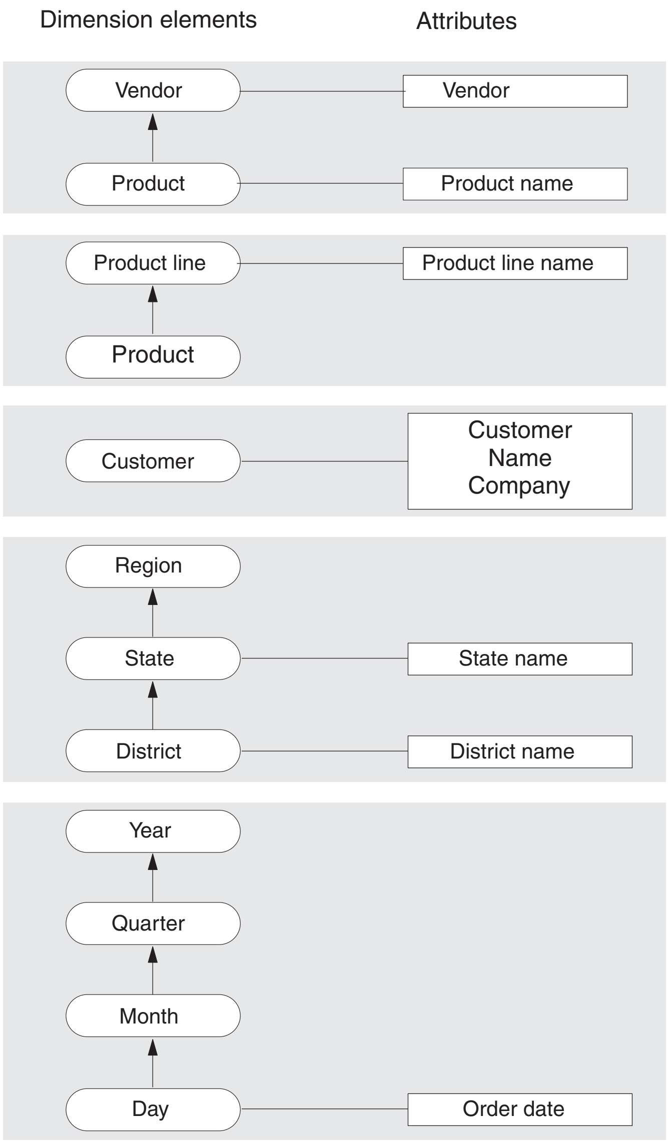

Figure 2-6. The relationships between dimensions, dimension elements, and the inherent hierarchies

In most cases, the dimension elements need to express the lowest possible granularity for each dimension, not because queries need to access individual low-level records, but because queries need to cut through the database in precise ways. In other words, even though the questions that a data warehousing environment poses are usually broad, these questions still depend on the lowest level of product detail.

Product dimension

The dimension elements for the **product** dimension are product, product line, and vendor:

- Product has a roll-up hierarchical relationship with product line and with vendor. Product has an attribute of product name.
- Product line has an attribute of product line name.
- Vendor has an attribute of vendor.

Customer dimension

The dimension element for the customer dimension is customer, which has attributes of customer, name, and company.

Geography dimension

The dimension elements for the geography dimension are district, state, and region:

- District has a roll-up hierarchical relationship with state, which has a roll-up hierarchical relationship with region.
- District has an attribute of district name.
- State has an attribute of state name.

Time dimension

The dimensional elements for the time dimension are day, month, quarter, and year.

- Day has a roll-up hierarchical relationship with month, which has a roll-up hierarchical relationship with quarter, which has a roll-up hierarchical relationship with year.
- Day has an attribute of order date.

Establish referential relationships

For the database server to support the dimensional data model, you must define logical dependencies between the fact tables and their dimension tables.

These logical dependencies should be reflected in the columns and indexes that you include in the schema of each table, and in the referential constraints that you define between each fact table and the associated dimension tables. For the large fragmented tables in typical data warehousing operations, these logical dependencies can be the basis for:

- Fragment-key expressions
- Join conditions
- Query predicates for fragment elimination

These query components can significantly improve the performance and throughput of the data warehouse.

A referential constraint enforces a one-to-one relationship between the values in referencing columns (of the foreign key) and the referenced columns (of the primary key or unique constraint). The relationship between the referenced table with the primary key constraint and the referencing table with the foreign key constraint is sometimes called a *parent-child relationship*. The corresponding columns of the parent and child tables can have the same identifiers, but having the same identifiers is not a requirement. There can also be a many-to-one relationship between the referencing table (with the foreign key) and the referenced table (with the primary key, or with the unique constraint).

In the dimensional model, a primary key constraint or a unique constraint in the fact table corresponds to a foreign key constraint in the dimension table. These constraints are specified in the CREATE TABLE or ALTER TABLE statements of SQL that defines the schema of the tables. Because the tables in the primary key and foreign key constraints must be in the same database, the database schema must include the dimension tables of each fact table.

The same data values can appear in the constrained columns of both tables. As a result, the index on which these referential constraints are defined can be used in queries as join predicates to join the fact table and the dimensional table.

For tables that are fragmented by expression, you can use the foreign key as the fragmentation key for the dimension tables. If you use the foreign key as the fragmentation key, you can use the equality operator or MATCHES operator with the primary key and foreign key values as the join predicate in queries and other data manipulation operations. The join predicate will be `TRUE` for only a subset of the fact table fragments. As a result, the query optimizer can use fragment elimination to process only the fact table partitions that contain qualifying rows.

Related concepts

Primary keys (Database Design and Implementation Guide)

Using the FOREIGN KEY Constraint (SQL Syntax)

How Indexes Affect Primary-Key, Unique, and Referential Constraints (SQL Syntax)

Restrictions on Referential Constraints (SQL Syntax)

Referential integrity (SQL Tutorial)

Adding a Primary-Key or Unique Constraint (SQL Syntax)

Restrictions on Referential Constraints (SQL Syntax)

Distribution schemes that eliminate fragments (Performance Guide)

Related reference

Star-Join Directives (SQL Syntax)

ENVIRONMENT Options (SQL Syntax)

Fragmentation expressions for fragment elimination (Performance Guide)

Choose the measures for the fact table

The measures for the data model include not only the data itself, but also new values that you calculate from the existing data. When you examine the measures, you might discover that you need to make adjustments either in the granularity of the fact table or the number of dimensions.

Another important decision you must make when you design the data model is whether to store the calculated results in the fact table or to derive these values at runtime.

The question to answer is “What measures are used to analyze the business?” Remember that the measures are the quantitative or factual data that tell *how much* or *how many*. The information that you gather from analysis of the sales business process results in the following list of measures:

- Revenue
- Cost
- Units sold
- Net profit

Use these measures to complete the fact table in the following figure.

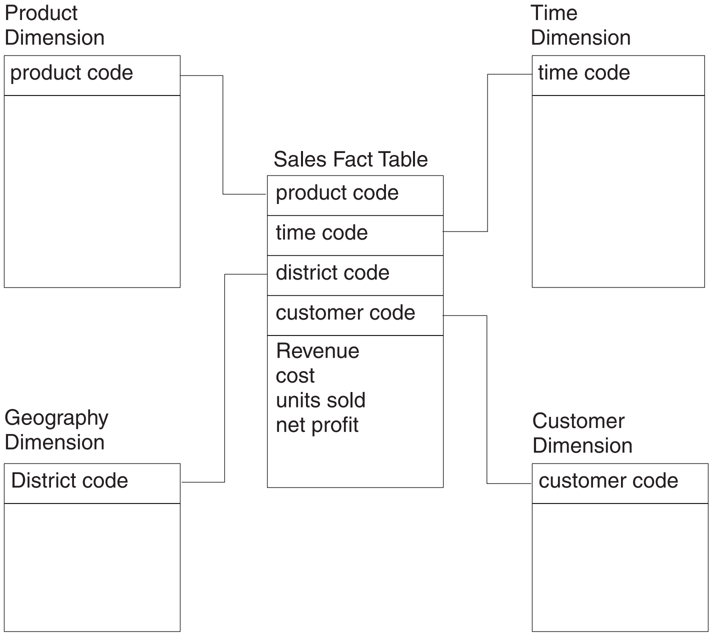

Figure 2-7. The Sales fact table references each dimension table

The elements of the Sales Fact table are: product code, time code, district code, customer code, revenue, cost, units sold, and net profit. Some of these elements join the Sales fact table to the dimension tables.

Product code element
Joins the Sales fact table to the Product dimension table. There are no other elements in the Product dimension table.

Time code element
Joins the Sales fact table to the Time dimension table. There are no other elements in the Time dimension table.

District code element
Joins the Sales fact table to the Geography dimension table. There are no other elements in the Geography dimension table.

Customer code element
Joins Sales fact table to Customer dimension table. There are no other elements in the Customer dimension.

In this model, additional space is left in the dimensional tables to add more elements. You will identify the other elements when you choose the attributes for each dimension table.

Related reference

"Choose the attributes for the dimension tables" on page 2-14

Keys to join the fact table with the dimension tables:

Each dimensional table needs to include a primary key that corresponds to a foreign key in the fact table. The fact table should have a primary (composite) key that is a combination of the foreign keys.

Assume that the following schema of shows both the logical and physical design of the database.

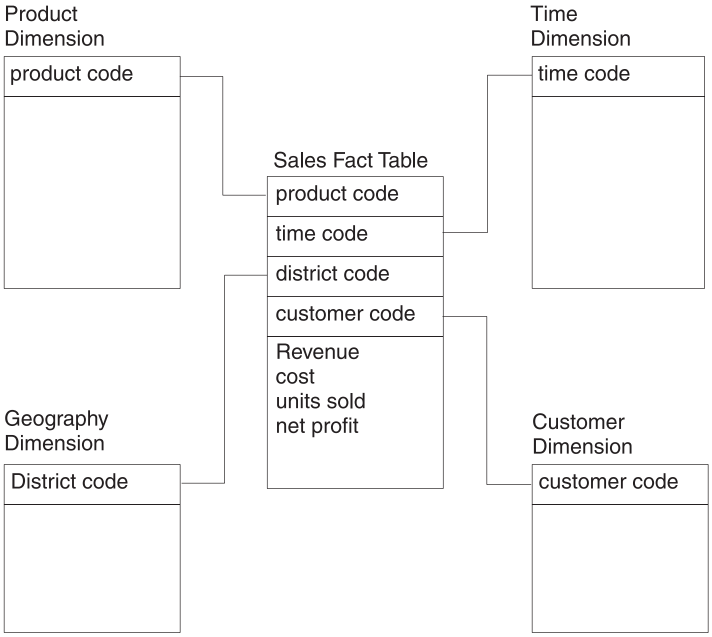

Figure 2-8. The Sales fact table references each dimension table

The database contains the following five tables:

- **Sales** fact table
- **Product** dimension table
- **Time** dimension table
- **Customer** dimension table
- **Geography** dimension table

Each of the dimensional tables includes a primary key (product, time_code, customer, district_code), and the corresponding columns in the fact table are foreign keys. The fact table also has a primary (composite) key that is a combination of these four foreign keys. As a rule, each foreign key of the fact table must have its counterpart in a dimension table.

Additionally, any table in a dimensional database that has a composite key must be a fact table. This means that every table in a dimensional database that expresses a many-to-many relationship is a fact table. Therefore a dimension table can also be a fact table for a separate star schema. This type of dimensional database model is referred to as a *snowflake schema*.

Tip: The primary key should be a short numeric data type (INT, SMALLINT, SERIAL) or a short character string (as used for codes). Do not use long character strings as primary keys.

Related concepts

"Use the snowflake schema for hierarchical dimension tables" on page 2-23

Resisting normalization

Efforts to normalize a dimensional database can actually prohibit an efficient dimensional design.

If the four foreign keys of the fact table are tightly administered consecutive integers, you could reserve as little as 16 bytes for all four keys (4 bytes each for time, product, customer, and geography) of the fact table. If the four measures in the fact table were each 4-byte integer columns, you would need to reserve only another 16 bytes. Thus, each record of the fact table would be only 32 bytes. Even a billion-row fact table would require only about 32 gigabytes of primary data space.

With its compact keys and data, such a storage-lean fact table is typical for dimensional databases. The fact table in a dimensional model is by nature highly normalized. You cannot further normalize the extremely complex many-to-many relationships among the four keys in the fact table because no correlation exists between the four dimension tables. Virtually every product is sold every day to all customers in every region.

The fact table is the largest table in a dimensional database. Because the dimension tables are usually much smaller than the fact table, you can ignore the dimension tables when you calculate the disk space for your database. Efforts to normalize any of the tables in a dimensional database solely to save disk space are pointless. Furthermore, normalized dimension tables undermine the ability of users to explore a single dimension table to set constraints and choose useful row headers.

Choose the attributes for the dimension tables

After you complete the fact table, you can decide the dimension attributes for each of the dimension tables. To illustrate how to choose the attributes, consider the **time** dimension. The data model for the sales business process defines a granularity of day that corresponds to the time dimension, so that each record in the **time** dimension table represents a day. Keep in mind that each field of the table is defined by the particular day the record represents.

The analysis of the sales business process also indicates that the marketing department needs monthly, quarterly, and annual reports, so the time dimension includes the elements: day, month, quarter, and year. Each element is assigned an attribute that describes the element and a code attribute, to avoid column values that contain long character strings. The following table shows the attributes for the **time** dimension table and sample values for each field of the table.

Table 2-3. Attributes for the time dimension

time code	order date	month code	month	quarter code	quarter	year
35276	07/31/2010	7	july	3	third q	2010
35277	08/01/2010	8	aug	3	third q	2010
35278	08/02/2010	8	aug	3	third q	2010

The previous table shows that the attribute names you assign should be familiar business terms that make it easy for end users to form queries on the database.

The following figure shows the completed data model for the sales business process with all the attributes defined for each dimension table. The elements of the Sales fact table are: product code, time code, district code, customer code, revenue, cost, units sold, and net profit. Some of these elements join the Sales fact table to the dimension tables. Additional elements for each dimension table have been identified.

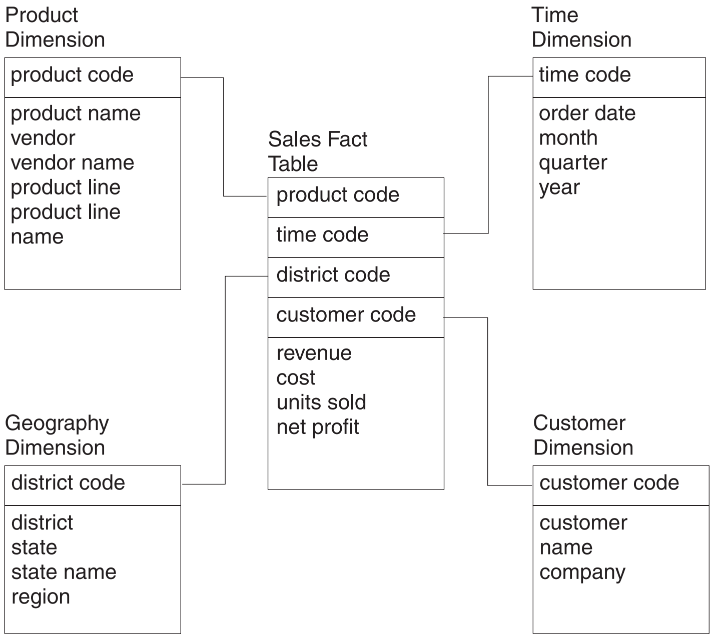

Figure 2-9. The completed dimensional data model for the sales business process

Product dimension table
The product code element joins the Sales fact table to the Product dimension table. The additional elements in the Product dimension table are: product name, vendor, vendor name, product line, and product line name.

Time dimension table
The time code element joins the Sales fact table to the Time dimension table. The additional elements in the Time dimension table are: order date, month, quarter, and year.

Geography dimension table
The district code element joins the Sales fact table to the Geography dimension table. The additional elements in the Geography dimension table are: district, state, state name, and region.

Customer dimension table
The customer code element joins Sales fact table to Customer dimension table. The additional elements in the Customer dimension table are: customer name and company.

Related concepts

"Choose the measures for the fact table" on page 2-11

Fragmentation: Storage distribution strategies

The performance of data warehousing applications can typically benefit from distributed storage allocation designs for partitioning a database table into two or more fragments. Each fragment has the same schema as the table, and stores a subset of the rows in the table (rather than a subset of its columns).

The fragments of a table can be stored in dbspaces on different devices, or in dbspaces on the same physical storage device. The fragments can also be stored in named partitions within a single dbspace.

A database can include both fragmented and nonfragmented tables. Index storage can also be fragmented, either in the same storage spaces as their table (called *attached* indexes) or in a different storage distribution scheme (*detached* indexes).

Potential performance and security advantages of distributed storage include these:

- For frequently-accessed tables, fragmentation can reduce the overhead of I/O contention for data that resides on a single storage device.
- The GRANT FRAGMENT and REVOKE FRAGMENT statements of SQL can specify the access privileges that users, roles, or the PUBLIC group hold on specified fragments of the table. With appropriate fragmentation strategies, these statements can selectively restrict user access to subsets of the records in a table.
- For databases that enables parallel-database queries (PDQ), multiple scan threads require less time to scan the fragments than to scan the same rows in a nonfragmented table.
- Input operations that distribute new rows across multiple fragments run more quickly (using multiple INSERT threads) that if a single table extent stores the same rows.
- For fragmentation strategies where the storage allocation of rows is correlated with data values, query execution plans can ignore fragments that are logically excluded by predicates in the query. Defining fragments to improve selectivity is called *fragment elimination*.
- In cluster environments, fragmentation can reduce the time required for recovery from hardware failure, because restoring only a subset of the fragments imposes a smaller data load than restoring the entire table.

Note: Do not confuse table fragmentation strategies, which can improve the efficiency and throughput of database operations, with the various pejorative meanings of *fragmentation* in reference to file systems that waste storage space or increase retrieval time through inefficient storage algorithms, or through insufficient use of defragmentation tools to store files in contiguous disk partitions.

Informix fragmentation options

Informix supports the following storage fragmentation strategies that can be applied to database tables:

By Round-robin

A specified number of fragments is defined for the table. Inserted rows are automatically distributed for storage in these fragments, without regard to data values in the row, in order to balance the number of rows in each fragment. Such fragments are called *round-robin fragments*.

By Expression

Each fragment is defined by a Boolean expression that can be evaluated for one or more columns of the table. Inserted rows are stored in a fragments for which the expression that defines the fragment is true for the data in that row. Rows that match the expression for more than one fragment are stored in the first matching fragment within the ordered list of fragments that the system catalog maintains for the table. Such fragments are called *expression fragments*.

Each user-defined permanent or temporary database table can either be *nonfragmented* or else can have exactly one fragmentation scheme. You cannot, for example, define a table in which some fragments use a round-robin strategy, and other fragments use an expression strategy.

You can use the ALTER FRAGMENT statement of SQL, however, to modify the fragmentation scheme of a table in various ways, including these:

- to change the fragmentation strategy of a fragmented table,
- to define a fragmentation strategy for a nonfragmented table,
- to change a fragmented table to a nonfragmented table,
- to add another fragment to an existing fragmented table,
- to combine two tables that have identical structures into a single fragmented table,
- to detach one fragment from a fragmented table and store the rows in a new nonfragmented table.

Storage fragmentation terms

The following terms are useful for understanding and using the various strategies available for the distributed storage of table and index fragments.

Fragment key

The column or a set of columns on which the table or index is fragmented. Depending on the chosen fragmentation strategy, the fragment key can be a column, or a single column expression, or a multi-column expression. For a row inserted into a table for which a fragment key is defined, the value of the column (or the set for values in the fragment key columns) determines which fragment stores the row. A synonym for fragment key is *partitioning key*. Tables partitioned by round-robin have no fragment key.

Fragment list

An ordered list of the fragments that the database server maintains for every fragmented table or index. By default, the ordinal positions of each fragment on this list reflects the sequence in which the fragments were created. The system catalog stores this integer value in the **sysfragments.evalpos** column of the row that describes the fragment. Queries that do not use fragment elimination read the fragments in ascending order of their **evalpos** values. The database server automatically updates **evalpos** values to reflect changes to the fragment list. Updates to the list are required, for example, when the ALTER FRAGMENT statement of SQL adds new fragments, or drops or modifies existing fragments.

Fragment expression

An expression that defines a specific fragment. For example, if the fragment key is **colA** of data type SMALLINT, a fragment could be defined by the expression `colA <=8 OR colA IN (9,10,21,22,23)` in an expression based fragmentation strategy. Tables partitioned by round-robin have no fragment expressions.

NULL fragment

A fragment that stores NULL values (either because its range fragment or list fragment expression is `IS NULL`, or because an expression-based fragment is defined with NULL as its fragment expression). For all fragmentation strategies except round-robin, the database server returns an exception if you insert a row whose fragment key value is missing, but no NULL fragment is defined (and for expression strategies, no REMAINDER fragment is defined). You do not need to define a NULL fragment if the fragment key column enforces a NOT NULL constraint.

REMAINDER fragment

A fragment that stores any row whose fragment key value does not match the fragment expression of any fragment. If you attempt to insert a row that does not match any fragment key value for a table or index that is fragmented by expression, and no REMAINDER fragment is defined, the database server issues an exception. You cannot define a REMAINDER fragment for tables fragmented by a round-robin strategy.

Related concepts

What is fragmentation? (Database Design and Implementation Guide)

Fragmentation guidelines (Performance Guide)

Distribution schemes (Performance Guide)

Table fragmentation and data storage (Administrator's Guide)

FRAGMENT BY Clause for Indexes (SQL Syntax)

Related reference

FRAGMENT BY Clause for Tables (SQL Syntax)

FRAGMENT BY Clause for Indexes (SQL Syntax)

Fragmentation by ROUND ROBIN

For a table that uses a round-robin distribution scheme, the rows that the database server stores in an insert or load operation are distributed cyclically among a user-defined number of fragments, so that the number of rows inserted into each fragment is approximately the same.

Round-robin distributions are also called *even* distributions, because the design goal of this strategy is for an evenly balanced distribution among the fragments.

The syntax for defining round-robin interval fragmentation requires that you specify at least two round-robin fragments in one of two forms. This form defines round-robin fragments and declare a name for each fragment:

```
FRAGMENT BY ROUND ROBIN
   PARTITION partition IN dbspace,
   . . .
   PARTITION partition IN dbspace
```

As in other fragmentation schemes, each PARTITION *partition* specification declares the name of a fragment, which must be unique among the names of fragments of the same table. The *dbspace* specification can be different for each fragment, or some fragments (or all of the fragments) can be stored in separate named partitions of the same dbspace. Each *partition* is the name of a round-robin fragment.

This alternative form defines round-robin fragments with no explicit name:

```
FRAGMENT BY ROUND ROBIN IN dbspace_list
```

Here the *dbspace_list* specification is a comma-separated list of at least 2 (but no more than 2048) dbspaces, each of which stores a single round-robin fragment. No *dbspace* can appear more than once in this list. (In the system catalog, the **sysfragments.partition** column stores the identifier of the fragment. For fragments defined without the PARTITION keyword, the **partition** value is the identifier of the dbspace where the fragment is stored. For this reason, a repeated *dbspace* in *dbspace_list* violates a uniqueness requirement for names of fragments of the same table.)

A round-robin distribution scheme must be defined by only one or the other of these two syntax forms.

A table that is fragmented by round-robin has no fragment key, no fragment expressions, and no REMAINDER fragment. (An alternative description is that every round-robin fragment resembles a remainder fragment, because no fragment expressions are defined to match a fragment key for the inserted rows. But the REMAINDER keyword is not valid in the SQL syntax to define a round-robin distribution strategy.)

Because no fragment expressions are evaluated when the database server loads new rows into round-robin fragments, this strategy provides the best performance for insert operations.

Only tables, not indexes, can be defined with round-robin fragmentation. For performance reasons, any indexes that you define on a table that is fragmented by round-robin should be nonfragmented indexes.

Because a round-robin distribution strategy has no fragment key and no fragment expressions, you cannot explicitly define a NULL round-robin fragment. When rows with missing data are loaded into a table by round-robin, the rows with NULL values are stored wherever the database server happens to insert them as it approximately equalizes the number of inserted rows for every fragment.

By design, the GRANT FRAGMENT and REVOKE FRAGMENT statements of SQL cannot reference round-robin fragments. Because each fragment stores a quasi-random subset of the rows, the DBA cannot predict which rows will be stored in a given round-robin fragment.

Because round-robin fragments are uncorrelated with data values, queries of tables that are fragmented by round-robin cannot benefit from fragment elimination. Round-robin distribution schemes are useful for balancing the rows in a set of table fragments across multiple devices, but other storage distribution schemes are typically used in data warehouse applications that query dimensional tables, because the performance advantages of round-robin in loading data are more than offset by slower data retrieval from round-robin fragments.

Related concepts

Round-robin distribution scheme (Database Design and Implementation Guide)

Fragmenting by ROUND ROBIN (SQL Syntax)

Fragmentation by EXPRESSION

For a table that uses an expression-based distribution scheme, the rows that the database server stores in an insert or load operation are distributed among a user-defined number of fragments, in which each fragment is defined by a Boolean expression for the fragment key.

The fragment expression must be a column expression. This can be the same column (or the same set of columns) for all of the fragments, or different fragments can be defined with different keys. The expression can only reference columns in the table that is being fragmented. Subqueries or calls to user-defined routines are not valid.

The syntax for defining an expression fragmentation strategy defines one or more expression fragments of this form:

```
FRAGMENT BY EXPRESSION
   PARTITION partition expression IN dbspace,
   . . .
```

```
PARTITION partition expression IN dbspace,
PARTITION partition VALUES (NULL) IN dbspace,
PARTITION partition REMAINDER IN dbspace
```

As in other fragmentation schemes, each `PARTITION` *`partition`* specification declares the unique name of a fragment. The *`expression`* specification defines the fragment expression, and the `IN` *`dbspace`* specification defines the storage location for the fragment. You can optionally define a NULL fragment by specifying `NULL` as the *`expression`*.

You also can optionally define a REMAINDER fragment for rows that match none of the specified fragment expressions. For some queries, the REMAINDER fragment might be difficult to eliminate, and for some tables, the REMAINDER fragment might become quite large, but the database server issues an exception if the fragment key value for an inserted row matches no fragment expression, and no REMAINDER fragment is defined.

You can optionally define a NULL fragment to stores rows in which the fragment key value is missing.

During an insert into a table that is fragmented by expression, the database server takes these actions:

1. The fragment key value for the row is evaluated.
2. The fragment expression for each fragment is evaluated and compared to the fragment key value for the row, beginning with the fragment whose **sysfragments.evalpos** value in the system catalog is lowest.
3. If there is no match, the previous step is repeated for the fragment with next highest **sysfragments.evalpos** value.
4. This continues until the first match is found between the fragment key value and a fragment expression, after which the row is stored in the matching fragment.
5. If no match is found in the entire list of fragments, the row is stored in the REMAINDER fragment. (In this case of a row with an unmatched fragment key, if no REMAINDER is defined, an exception is issued.)

For expression-based fragmentation schemes that define overlapping fragment expressions, the storage location of rows that match the fragment expression of more than one fragment is dependent on the **evalpos** value for that fragment. You can avoid this dependency by only defining non-overlapping fragment expressions.

The **evalpos** value of a fragment is determined by its position in the initial fragment list within the FRAGMENT BY EXPRESSION or PARTITION BY EXPRESSION clause that defined the storage distribution of the table. Any new fragments added by ALTER FRAGMENT operations are assigned, by default, the next higher **evalpos** value (and will therefore be evaluated last during INSERT operations) unless you explicitly specify a position with the BEFORE or AFTER keyword. In this case, the **evalpos** value for the new fragment will be the ordinal position where was inserted into the fragment list. For tables that are fragmented by expression into a large number of fragments, you can achieve greater efficiency in INSERT an LOAD operations when fragments that are more likely to match fragment key values have relatively low **evalpos** values within the fragment list.

Fragmentation by expressions that creates nonoverlapping fragments on a single column can be an effective strategy for supporting fragment elimination in queries. The database server can eliminate fragments, for example, for queries with range expressions as well as queries with equality expressions if the query predicates correspond to fragment expressions. Expressions with relational operators and logical operators (or with both) can similarly be used for fragment expressions that match query filters.

Related concepts

Using the REMAINDER Keyword (SQL Syntax)

Related reference

Expression Fragment Clause (SQL Syntax)

Handle common dimensional data-modeling problems

The dimensional model that the previous sections describe illustrates only the most basic concepts and techniques of dimensional data modeling. The data model you build to address the business needs of your enterprise typically involves additional problems and difficulties that you must resolve to achieve the best possible query performance from your database. This section describes various methods you can use to resolve some of the most common problems that arise when you build a dimensional data model.

Minimize the number of attributes in a dimension table

Dimension tables that contain customer or product information might easily have 50 to 100 attributes and many millions of rows. However, dimension tables with too many attributes can lead to excessively wide rows and poor performance. For this reason, you might want to separate out certain groups of attributes from a dimension table and put them in a separate table called a *minidimension* table. A minidimension table consists of a small group of attributes that are separated out from a larger dimension table. You might choose to create a minidimension table for attributes that have either of the following characteristics:

- The fields are rarely used as constraints in a query.
- The fields are frequently compared together.

The following figure shows a minidimension table for demographic information that is separated out from a **customer** table.

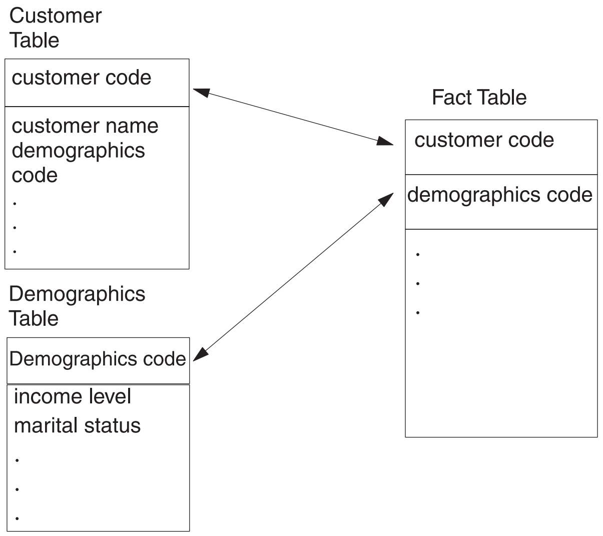

Figure 2-10. A Minidimension Table for Demographics Information

In the **demographics** table, you can store the demographics key as a foreign key in both the fact table and the **customer** table, which allows you to join the demographics table directly to the fact table. You can also use the demographics key directly with the **customer** table to browse demographic attributes.

Dimensions that occasionally change

In a dimensional database where updates are infrequent (as opposed to OLTP systems), most dimensions are relatively constant over time, because changes in sales districts or regions, or in company names and addresses, occur infrequently. However, to make historical comparisons, these changes must be handled when they do occur. The following figure shows an example of a dimension that has changed.

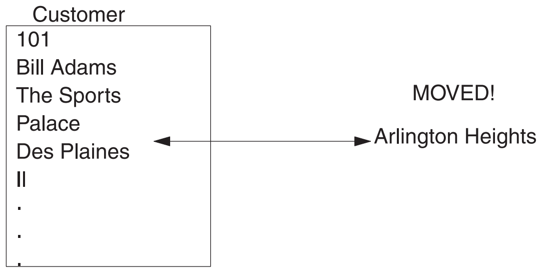

Figure 2-11. A dimension that changes

You can use three ways to handle changes that occur in a dimension:

Change the value stored in the dimension column
In the previous figure, the record for Bill Adams in the **customer** dimension table is updated to show the new address `Arlington Heights`. All of this customer's previous sales history is now associated with the district of Arlington Heights instead of Des Plaines.

Create a second dimension record with the new value and a generalized key
This approach effectively partitions history. The **customer** dimension table would now contain two records for Bill Adams. The old record with a key

of 101 remains, and records in the fact table are still associated with it. A new record is also added to the **customer** dimension table for Bill Adams, with a new key that might consist of the old key plus some version digits (101.01, for example). All subsequent records that are added to the fact table for Bill Adams are associated with this new key.

Add a new field in the customer dimension table for the affected attribute and rename the old attribute

This approach is rarely used unless you need to track old history in terms of the new value and vice-versa. The **customer** dimension table gets a new attribute named **current address**, and the old attribute is renamed **original address**. The record that contains information about Bill Adams includes values for both the original and current address.

Use the snowflake schema for hierarchical dimension tables

A *snowflake schema* is a variation on the star schema, in which very large dimension tables are normalized into multiple tables. Dimensions with hierarchies can be decomposed into a snowflake structure when you want to avoid joins to big dimension tables when you are using an aggregate of the fact table. For example, if you have brand information that you want to separate out from a **product** dimension table, you can create a brand snowflake that consists of a single row for each brand and that contains significantly fewer rows than the **product** dimension table. The following figure shows a snowflake structure for the brand and product line elements and the **brand_agg** aggregate table.

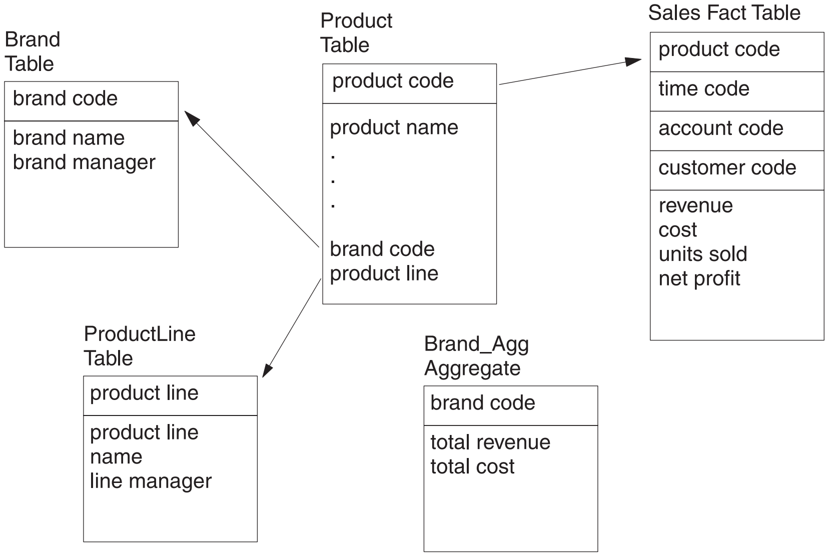

Figure 2-12. An example of a snowflake schema

If you create an aggregate table, **brand_agg**, that consists of the brand code and the total revenue per brand, you can use the snowflake schema to avoid the join to the much larger **sales** table. For example, you can use the following query on the **brand** and **brand_agg** tables:

```
SELECT brand.brand_name, brand_agg.total_revenue
FROM brand, brand_agg
   WHERE brand.brand_code = brand_agg.brand_code
   AND brand.brand_name = 'Anza'
```

Without a snowflaked dimension table, you use a SELECT UNIQUE or SELECT DISTINCT statement on the entire **product** table (potentially, a very large dimension table that includes all the brand and product-line attributes) to eliminate duplicate rows.

While snowflake schemas are unnecessary when the dimension tables are relatively small, a retail or mail-order business that has customer or product dimension tables that contain millions of rows can use snowflake schemas to significantly improve performance.

If an aggregate table is not available, any joins to a dimension element that was normalized with a snowflake schema must now be a three-way join, as the following query shows. A three-way join reduces some of the performance advantages of a dimensional database.

```
SELECT brand.brand_name, SUM(sales.revenue)
FROM product, brand, sales
   WHERE product.brand_code = brand.brand_code
   AND brand.brand_name = 'Alltemp'
GROUP BY brand_name
```

Related concepts

"Keys to join the fact table with the dimension tables" on page 2-12

Chapter 3. Implement a dimensional database

You will learn the SQL statements that you need to implement the dimensional data model

This section shows you the SQL statements required to implement the dimensional database that is described in the section Chapter 2, “Design a dimensional data model,” on page 2-1. Remember that this database serves only as an illustrative example of a data-warehousing environment. For the sake of the example, it is translated into SQL statements.

This section describes the **sales_demo** database.

Related reference

Chapter 2, “Design a dimensional data model,” on page 2-1

Implement the sales_demo dimensional database

This section shows the SQL statements that you can use to create a dimensional database from the data model that you learned about in Chapter 2, “Design a dimensional data model,” on page 2-1. You can use interactive SQL to write the individual statements that create the database or you can run a script that automatically executes all the statements that you need to implement the database. The CREATE DATABASE and CREATE TABLE statements create the data model as tables in a database. After you create the database, you can use the LOAD and INSERT statements to populate the tables.

Create the dimensional database

You must create the dimensional database before you can create any of the tables or other objects that the database must contain.

When you use the IBM Informix database server to create a database, the server sets up records that show the existence of the database and its mode of logging. The database server manages disk space directly, so these records are not visible to operating-system commands.

The following statement shows the syntax that you use to create a database that is called **sales_demo**:

```
CREATE DATABASE sales_demo
```

The CREATE TABLE statement for the dimension and fact tables

This section includes the CREATE TABLE statements that you use to create the tables of the **sales_demo** dimensional database.

Referential integrity is, of course, an important requirement for dimensional databases. However, the following schema for the **sales_demo** database does not define the primary and foreign key relationships that exist between the fact table and its dimension tables. The schema does not define these primary and foreign key relationships because data-loading performance improves dramatically when the database server does not enforce constraint checking. Given that data warehousing environments often require that tens or hundreds of gigabytes of data are loaded within a specified time, data-load performance should be a factor when you decide how to implement a database in a warehousing environment. Assume that if the **sales_demo** database is implemented as a live data mart, some data extraction tool (rather than the database server) is used to enforce referential integrity between the fact table and dimension tables.

Tip: After you create and load a table, you can add primary key and foreign key constraints to the table with the ALTER TABLE statement to enforce referential integrity. This method is required only for express load mode. If the constraints and indexes are necessary and costly to drop before a load, then deluxe load mode is the best option.

The following statements create the **time**, **geography**, **product**, and **customer** tables. These tables are the dimensions for the **sales** fact table. A SERIAL field serves as the primary key for the **district_code** column of the **geography** table.

```
CREATE TABLE time
(
time_code     INT,
order_date    DATE,
month_code    SMALLINT,
month_name    CHAR(10),
quarter_code  SMALLINT,
quarter_name  CHAR(10),
year INTEGER
);

CREATE TABLE geography
(
district_code  SERIAL,
district_name  CHAR(15),
state_code     CHAR(2),
state_name     CHAR(18),
region         SMALLINT
);

CREATE TABLE product (
product_code   INTEGER,
product_name   CHAR(31),
vendor_code    CHAR(3),
vendor_name    CHAR(15),
product_line_code  SMALLINT,
product_line_name  CHAR(15)
);

CREATE TABLE customer (
customer_code  INTEGER,
customer_name  CHAR(31),
company_name   CHAR(20)
);
```

The **sales** fact table has pointers to each dimension table. For example, **customer_code** references the customer table, **district_code** references the geography table, and so forth. The **sales** table also contains the measures for the units sold, revenue, cost, and net profit.

```
CREATE TABLE sales
(
customer_code  INTEGER,
district_code  SMALLINT,
time_code      INTEGER,
product_code   INTEGER,
units_sold     SMALLINT,
revenue        MONEY(8,2),
cost           MONEY(8,2),
net_profit     MONEY(8,2)
);
```

Tip: The most useful measures (facts) are numeric and additive. Because of the great size of databases in data-warehousing environments, virtually every query against the fact table might require thousands or millions of records to construct a result set. The only useful way to compress these records is to aggregate them. In the **sales** table, each column for the measures is defined on a numeric data type, so you can easily build result sets from the **units_sold**, **revenue**, **cost**, and **net_profit** columns.

For your convenience, the file called `createdw.sql` contains all the preceding CREATE TABLE statements.

Mapping data from data sources to the database

The **stores_demo** demonstration database is the primary data source for the **sales_demo** database.

The following table shows the relationship between data warehousing business terms and the data sources. It also shows the data source for each column and table of the **sales_demo** database.

Table 3-1. The relationship between data warehousing business terms and data sources

Business Term	Data Source	Table.Column Name
Sales Fact Table:		
product code		sales.product_code
customer code		sales.customer_code
district code		sales.district_code
time code		sales.time_code
revenue	stores_demo:items.total_price	sales.revenue
units sold	stores_demo:items.quantity	sales.units_sold
cost	costs.lst (per unit)	sales.cost
net profit	calculated: revenue minus cost	sales.net_profit
Product Dimension Table:		
product	stores_demo:catalog.catalog_num	product.product_code
product name	stores_demo:stock.manu_code and stores_demo:stock.description	product.product_name
product line	stores_demo:orders.stock_num	product.product_line_code
product line name	stores_demo:stock.description	product.product_line_name
vendor	stores_demo:orders.manu_code	product.vendor_code
vendor name	stores_demo:manufact.manu_name	product.vendor_name
Customer Dimension Table:		
customer	stores_demo:orders.customer_num	customer.customer_code
customer name	stores_demo:customer.fname plus stores_demo:customer.lname	customer.customer_name
company	stores_demo:customer.company	customer.company_name
Geography Dimension Table:		
district code	generated	geography.district_code
district	stores_demo:customer.city	geography.district_name
state	stores_demo:customer.state	geography.state_code

Table 3-1. The relationship between data warehousing business terms and data sources (continued)

Business Term	Data Source	Table.Column Name
state name	stores_demo.state.sname	geography.state_name
region	derived: If state = "CA" THEN region = 1, ELSE region = 2	geography.region
Time Dimension Table:		
time code	generated	time.time_code
order date	stores_demo:orders.order_date	time.order_date
month	derived from order date generated	time.month_name time.month.code
quarter	derived from order date generated	time.quarter_name time.quarter_code
year	derived from order date	time.year

Several files with a `.unl` suffix contain the data that is loaded into the **sales_demo** database. The files that contain the SQL statements that create and load the database have a `.sql` suffix.

If your database server runs on UNIX, you can access the `*.sql` and `*.unl` files from the directory `$INFORMIXDIR/demo/dbaccess`.

If your database server runs on Windows, you can access the `*.sql` and `*.unl` files from the directory `%INFORMIXDIR%\demo\dbaccess`.

Load data into the dimensional database

An important step when you implement a dimensional database is to develop and document a load strategy. This section shows the LOAD and INSERT statements that you can use to populate the tables of the **sales_demo** database.

Tip: In a live data warehousing environment, you typically do not use the LOAD or INSERT statements to load large amounts of data to and from IBM Informix databases.

IBM Informix database servers provide different features for high-performance loading and unloading of data.

For information about high-performance loading, see your *IBM Informix Administrator's Guide* or *IBM Informix High-Performance Loader User's Guide*.

The following statement loads the **time** table with data first so that you can use it to determine the time code for each row that is loaded into the **sales** table:

```
LOAD FROM 'time.unl' INSERT INTO time
```

The following statement loads the **geography** table. After you load the **geography** table, you can use the district code data to load the **sales** table.

```
INSERT INTO geography(district_name, state_code, state_name)
SELECT DISTINCT c.city, s.code, s.sname
   FROM stores_demo:customer c, stores_demo:state s
      WHERE c.state = s.code
```

The following statements add the region code to the **geography** table:

```
UPDATE geography
   SET region = 1
   WHERE state_code = 'CA'

UPDATE geography
   SET region = 2
   WHERE state_code <> 'CA'
```

The following statement loads the **customer** table:

```
INSERT INTO customer (customer_code, customer_name, company_name)
SELECT c.customer_num, trim(c.fname) ||' '|| c.lname, c.company
FROM stores_demo:customer c
```

The following statement loads the **product** table:

```
INSERT INTO product (product_code, product_name, vendor_code,
   vendor_name,product_line_code, product_line_name)
SELECT a.catalog_num,
   trim(m.manu_name)||' '||s.description,
   m.manu_code, m.manu_name,
   s.stock_num, s.description
FROM stores_demo:catalog a, stores_demo:manufact m,
   stores_demo:stock s
   WHERE a.stock_num = s.stock_num
      AND a.manu_code = s.manu_code
      AND s.manu_code = m.manu_code;
```

The following statement loads the **sales** fact table with one row for each product, per customer, per day, per district. The cost from the **cost** table is used to calculate the total cost (cost * quantity).

```
INSERT INTO sales (customer_code, district_code, time_code,
   product_code, units_sold, cost, revenue, net_profit)
SELECT
   c.customer_num, g.district_code, t.time_code,
   p.product_code, SUM(i.quantity),
   SUM(i.quantity * x.cost), SUM(i.total_price),
   SUM(i.total_price) - SUM(i.quantity * x.cost)
FROM stores_demo:customer c, geography g, time t,
   product p,stores_demo:items i,
   stores_demo:orders o, cost x
WHERE c.customer_num = o.customer_num
   AND o.order_num = i.order_num
   AND p.product_line_code = i.stock_num
   AND p.vendor_code = i.manu_code
   AND t.order_date = o.order_date
   AND p.product_code = x.product_code
   AND c.city = g.district_name
GROUP BY 1,2,3,4;
```

Test the dimensional database

After you create the tables and load the data into the database, you should test the dimensional database.

You can create SQL queries to retrieve the data necessary for the standard reports listed in the business-process summary (see the “Summary of a business process” on page 2-6). Use the following ad hoc queries to test that the dimensional database was properly implemented.

The following statement returns the monthly revenue, cost, and net profit by product line for each vendor:

```
SELECT vendor_name, product_line_name, month_name,
   SUM(revenue) total_revenue, SUM(cost) total_cost,
   SUM(net_profit) total_profit
FROM product, time, sales
WHERE product.product_code = sales.product_code
   AND time.time_code = sales.time_code
GROUP BY vendor_name, product_line_name, month_name
ORDER BY vendor_name, product_line_name;
```

The following statement returns the revenue and units sold by product, by region, and by month:

```
SELECT product_name, region, month_name,
   SUM(revenue), SUM(units_sold)
FROM product, geography, time, sales
WHERE product.product_code = sales.product_code
   AND geography.district_code = sales.district_code
   AND time.time_code = sales.time_code
GROUP BY product_name, region, month_name
ORDER BY product_name, region;
```

The following statement returns the monthly customer revenue:

```
SELECT customer_name, company_name, month_name,
   SUM(revenue)
FROM customer, time, sales
WHERE customer.customer_code = sales.customer_code
   AND time.time_code = sales.time_code
GROUP BY customer_name, company_name, month_name
ORDER BY customer_name;
```

The following statement returns the quarterly revenue per vendor:

```
SELECT vendor_name, year, quarter_name, SUM(revenue)
FROM product, time, sales
WHERE product.product_code = sales.product_code
   AND time.time_code = sales.time_code
GROUP BY vendor_name, year, quarter_name
ORDER BY vendor_name, year
```

Implementing a dimensional data model and loading data with Informix Warehouse

You can use the Design Studio provided with Informix Warehouse to create a physical data model of your dimensional database based on your relational database. You can specify how to extract, transform, and load the data from your relational database to your dimensional database.

For specific instructions on how to install and use Design Studio and other Informix Warehouse tools, see the Data warehousing and analytics node in the Informix information center.

To implement a dimensional data model and load data using Design Studio:

1. Start Design Studio and create a new project.
2. Create connections to your existing relational database and the dimensional database that you will create.
3. Create a physical data model for the dimensional database. For example, you can:
   - Reverse engineer the model based on the relational database schema.
   - Manually create a model.

4. Design data flows that represent the movement of data from the source, through a series of transform operations, and into the target system.
5. Design control flows that define processing rules for the execution of a set of related data flows.

Moving data from relational tables into dimensional tables by using external tables

Use SQL statements to unload data from relational tables into external tables, which are data files that are in table format, and then load the data from the data files into the dimensional tables.

Before beginning, document a strategy for mapping data in the relational database to the dimensional database.

To unload data from the relational database into external tables and then load the data into the dimensional database:

1. Unload the data from a relational database to external tables. Repeat the following steps to create as many external tables as are required for the data that you want to move.
   a. Use the CREATE EXTERNAL TABLE statement to describe the location of the external table and the format of the data. The following sample CREATE EXTERNAL TABLE statement creates an external table called `emp_ext`, with data stored in a specified fixed format:

      ```
      CREATE EXTERNAL TABLE emp_ext
      ( name CHAR(18) EXTERNAL CHAR(18),
      hiredate DATE EXTERNAL CHAR(10),
      address VARCHAR(40) EXTERNAL CHAR(40),
      empno INTEGER EXTERNAL CHAR(6) )
      USING (
      FORMAT 'FIXED',
      DATAFILES
      ("DISK:/work2/mydir/employee.unl")
      );
      ```

   b. Use the INSERT...SELECT statement to map the relational database table to the external table. The following sample INSERT statement loads the employee database table into the external table called `emp_ext`:

      ```
      INSERT INTO emp_ext SELECT * FROM employee
      ```

      The data from the employee database table is stored in a data file called `employee.unl`.
2. If necessary, copy or move the data files to the system where the dimensional database is located.
3. Load the data from the data files to the dimensional database. Repeat the following steps to load all the data files that you created in the previous steps.
   a. Use the CREATE EXTERNAL TABLE statement to describe the location of the data file and the format of the data. The following code is a sample CREATE EXTERNAL TABLE statement:

      ```
      CREATE EXTERNAL TABLE emp_ext
      ( name CHAR(18) EXTERNAL CHAR(18),
      hiredate DATE EXTERNAL CHAR(10),
      address VARCHAR(40) EXTERNAL CHAR(40),
      empno INTEGER EXTERNAL CHAR(6) )
      USING (
      ```

```
FORMAT 'FIXED',
DATAFILES
("DISK:/work3/mydir/employee.unl")
);
```

b. Use the INSERT...SELECT statement to map the data from the data file to the table in the dimensional database. The following sample INSERT statement loads the employee data file into the employee database table:

```
INSERT INTO employee SELECT * FROM emp_ext
```

Chapter 4. Performance tuning dimensional databases

This section describes how to tune the performance of your queries and to understand data distribution statistics.

Related reference

Chapter 2, “Design a dimensional data model,” on page 2-1

Query execution plans

When a SELECT statement or other DML operation is submitted to the database server, the query execution optimizer designs a query execution plan. The query execution optimizer is often referenced as the *query optimizer*.

To design a query execution plan, and estimate the costs of candidate query plans, the query optimizer considers a wide range of information including:

- Specifications that identify the database objects, predicates, filters, joins, and other operations in the SQL syntax that defines the query operation
- System catalog information about indexes and constraints on the tables, views, and columns that are referenced or implied in the query
- Data distribution statistics for the tables and indexes, or for their fragments, that are referenced or implied in the query
- Optimizer directives that are specified inline or as external optimizer directives that favor or avoid subsets of the potential query plans
- Information in the database server environment or in the session environment that affects the query execution optimizer

Related concepts

Queries and the query optimizer (Performance Guide)

Enabling external directives (Performance Guide)

Related reference

Optimizer Directives (SQL Syntax)

Data distribution statistics

Data distribution statistics are stored in the system catalog for use by the query optimizer when it designs query execution plans. These statistics, together with other information, enable the optimizer to estimate the relative costs among the execution plans that the optimizer is considering for a specific query. Distribution statistics that the optimizer examines for tables that are referenced in queries can include column distribution statistics for the table and for its indexes, if the database server has gathered statistics for individual table or index fragments.

The following system catalog tables store data distribution information that is available to the query optimizer:

SYSDISTRIB
Stores data distribution information for tables and indexes.

The following system catalog tables store information pertaining to changes to rows since the most recent update to table, index, or fragment statistics.

SYSDISTRIB

Counts the number of rows changed by DML operations since table statistics were last updated, the date and time of that update, and the time required to build column distributions.

SYSFRAGMENTS

Counts the number of rows changed by DML operations since fragment-level statistics were last updated.

SYSINDICES

Counts the number of rows changed by DML operations since index statistics were last updated, the date and time of that update, and time required to build low level distributions for the lead column of the index.

The following configuration parameters can affect the database server behavior for the calculation, display, or other operations on data distribution statistics for tables or for fragments that can be used in query plans:

EXPLAIN_STAT

Enable or disable the inclusion of a Query Statistics section in the explain output file. This is enabled by default.

SYSSBSPACENAME

Specifies the name of the sbspace in which the database server stores data-distribution statistics (as smart large objects) that the UPDATE STATISTICS statement collects for certain user-defined data types. Because the data distributions for UDTs can be large, you have the option to store them in an sbspace instead of in the **sysdistrib** system catalog table where distribution statistics are stored by default.

Related concepts

System catalog tables (SQL Reference)

Updating Statistics for Tables (SQL Syntax)

Data-distribution configuration (Performance Guide)

Updating statistics on very large databases (Performance Guide)

Automatic management of data distribution statistics

The Informix database server supports several mechanisms for automating some of the tasks that are involved in gathering, dropping, and refreshing data distribution statistics for tables, indexes, table fragments, and index fragments.

Automatic statistics maintenance in DDL operations

The Informix database server automatically creates, updates, or drops data distribution statistics during certain operations that create, alter, or destroy database objects.

ALTER FRAGMENT ATTACH operations

If the automatic mode for detecting stale distribution statistics is enabled, and the table being attached to has fragmented distribution statistics, the database server calculates the distribution statistics of the new fragment. Stale distribution statistics of existing fragments are also recalculated at this point. This recalculation of fragment statistics runs in the background. After the database server has calculated the fragment statistics, it merges them to form table distribution statistics, and stores the results in the system catalog.

Distribution statistics are not recalculated, however, unless explicit or default value of the AUTO_STAT_MODE configuration parameter or the AUTO_STAT_MODE session environment setting has enabled the automatic mode for detecting stale data distribution statistics.

ALTER TABLE ADD CONSTRAINT operations

ALTER TABLE ADD CONSTRAINT statements that use the Single Column Constraint format to implicitly create an index on a non-opaque column also automatically calculate the distribution of the specified column. Similarly, if the Multiple-Column Constraint format specify a list of columns as the scope of the new constraint, the database server implicitly creates an index on the same non-opaque column or set of columns as the referential constraint, distribution statistics are automatically calculated on the specified column, or on the lead column of a multiple-column constraint.

These distribution statistics are available to the query optimizer when it designs query plans for the table on which the constraint is defined:

- For columns on which the new constraint is implemented as a B-tree index, the recalculated column distribution statistics are equivalent to distributions created by the UPDATE STATISTICS statement in HIGH mode.
- If the new constraint is not implemented as a B-tree index, the automatically recalculated statistics correspond to distributions created by the UPDATE STATISTICS statement in LOW mode.

These distribution statistics are available to the query optimizer when it designs query plans for the table on which the new constraint was created.

ALTER TABLE MODIFY operations

ALTER TABLE MODIFY statements that use the Single Column Constraint format or Multiple Column Constraint format to define constraints similarly cause the database server to calculate data distribution statistics for the indexes that are implicitly created to enforce the constraints. These distribution statistics have the same attributes as those that are automatically for an index on a non-opaque column, and that are also automatically calculated during ALTER TABLE ADD CONSTRAINT operations. These statistics are available to the query optimizer when it designs query plans for the table on which the constraints were define

CREATE INDEX operations

The database server automatically calculates index statistics, equivalent to the statistics gathered by UPDATE STATISTICS in LOW mode, when you create a B-tree index on a UDT column of an existing table, or if you create a functional index or a virtual index interface (VII) index on a column of an existing table. Statistics that are collected automatically by this feature are stored in the system catalog and are available to the query optimizer, without the need for running the UPDATE STATISTICS statement manually. When B-tree indexes are created, column statistics are collected on the first index column, equivalent to what UPDATE STATISTICS generates in HIGH mode, with a resolution is 1% for tables of fewer than a million rows, and 0.5% for larger tables. (Tables with more than 1 million rows have a better resolution, because they have more bins for statistics.)

Auto Update Statistics (AUS) maintenance system

This uses a combination of Scheduler sensors, tasks, thresholds, and tables to evaluate and update data distribution statistics. The system provides as built-in input criteria a set of configuration parameter values. The system administrator can modify these to reflect current requirements and workloads. The AUS system combines these criteria with information from the sysmaster database to automatically identify tables whose distributions are becoming stale, and generates the text of UPDATE STATISTICS statements to refresh the distribution statistics for those tables.

The list of generated UPDATE STATISTICS statements is run automatically each week at a designated period of low throughput, to update as many table distributions as can be recalculated during the designated maintenance period. Any UPDATE STATISTICS statements that do not complete are retained on the list for the next maintenance period.

The AUS maintenance system for data distribution statistics is also available in the IBM OpenAdmin Tool (OAT) for Informix. Refer to the OAT online help for detailed information on how to configure the AUS maintenance system to provide current table statistics automatically. OAT is available as an open source download from the **iiug.org** website and from IBM websites.

Related concepts

AUTO_STAT_MODE Environment Option (SQL Syntax)

STATCHANGE Environment Option (SQL Syntax)

Automated Table Statistics Maintenance (SQL Syntax)

Related reference

AUTO_STAT_MODE configuration parameter (Administrator's Reference)

Using the FORCE and AUTO keywords (SQL Syntax)

STATCHANGE configuration parameter (Administrator's Reference)

Chapter 5. Informix Warehouse Feature

Informix Warehouse Feature is a suite of products that combines the strength of Informix with a data warehousing infrastructure from IBM.

You can use Informix Warehouse Feature to build a complete data warehousing solution that includes a highly scalable relational database, data access capabilities, and front-end analysis tools.

More information about the Informix Warehouse Feature can be found in the Informix Information Center.

Related information

Informix Warehouse Feature Overview

Appendix. Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

Accessibility features for IBM Informix products

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in IBM Informix products. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- The attachment of alternative input and output devices.

Tip: The information center and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features by using the keyboard instead of the mouse.

Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

Related accessibility information

IBM is committed to making our documentation accessible to persons with disabilities. Our publications are available in HTML format so that they can be accessed with assistive technology such as screen reader software.

You can view the publications in Adobe Portable Document Format (PDF) by using the Adobe Acrobat Reader.

IBM and accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the IBM commitment to accessibility.

Dotted decimal syntax diagrams

The syntax diagrams in our publications are available in dotted decimal format, which is an accessible format that is available only if you are using a screen reader.

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), the elements can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, `3` or `3.1` or `3.1.1`. To hear these numbers correctly, make sure that your screen reader is set to read punctuation. All syntax elements that have the same dotted decimal number (for example, all syntax elements that have the number `3.1`) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, the word or symbol is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is read as 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol that provides information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, that element is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 refers to a separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

? Specifies an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element (for example, 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! Specifies a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used.

However, if you hear the lines `2? FILE`, `2.1`, `2.1.1! (KEEP)`, and `2.1.1 (DELETE)`, the default option KEEP only applies to the next higher dotted decimal number, `2.1` (which does not have an associated keyword), and does not apply to `2? FILE`. Nothing is used if the keyword `FILE` is omitted.

* Specifies a syntax element that can be repeated zero or more times. A dotted decimal number followed by the `*` symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line `5.1* data-area`, you know that you can include more than one data area or you can include none. If you hear the lines `3*`, `3 HOST`, and `3 STATE`, you know that you can include `HOST`, `STATE`, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write `HOST STATE`, but you cannot write `HOST HOST`.
3. The `*` symbol is equivalent to a loop-back line in a railroad syntax diagram.

+ Specifies a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times. For example, if you hear the line `6.1+ data-area`, you must include at least one data area. If you hear the lines `2+`, `2 HOST`, and `2 STATE`, you know that you must include `HOST`, `STATE`, or both. As for the * symbol, you can repeat a particular item if it is the only item with that dotted decimal number. The `+` symbol, like the `*` symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

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

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

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

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

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

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

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

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

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

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

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

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

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

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

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

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

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

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

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

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

COPYRIGHT LICENSE:

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

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

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

© Copyright IBM Corp. _enter the year or years_. All rights reserved.

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

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

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

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

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

Other company, product, or service names may be trademarks or service marks of others.

Index

Printed in USA

SC27-3835-00