Contents

Chapter 3

Data Warehouse and OLAP Technology: An Overview

Data warehouses generalize and consolidate data in multidimensional space. The construction of data warehouses, which involves data cleaning, data transformation, and data integration, can be viewed as an important preprocessing step for data mining. Moreover, data warehouses provide *on-line analytical processing (OLAP)* tools for the interactive analysis of multidimensional data of varied granularities, which facilitates effective data generalization and data mining. Many other data mining functions, such as classification, prediction, association, and clustering, can be integrated with OLAP operations to enhance interactive mining of knowledge at multiple levels of abstraction. Hence, the data warehouse has become an increasingly important platform for data analysis and on-line analytical processing and will provide an effective platform for data mining. Therefore, data warehousing and OLAP form an essential step in the knowledge discovery process. This chapter presents an overview of data warehouse and OLAP technology. Such an overview is essential for understanding the overall data mining and knowledge discovery process.

In this chapter, we study a well-accepted definition of the data warehouse and see why more and more organizations are building data warehouses for the analysis of their data. In particular, we study the *data cube*, a multidimensional data model for data warehouses and OLAP, as well as OLAP operations, such as roll-up, drill-down, slicing and dicing. We also look at data warehouse architecture, including steps on data warehouse design and construction. An overview of data warehouse implementation is presented, which examines general strategies for efficient data cube computation, OLAP data indexing, and OLAP query processing. Finally, we look at *online-analytical mining*, a powerful paradigm that integrates data warehouse and OLAP technology with that of data mining.

3.1 What Is a Data Warehouse?

Data warehousing provides architectures and tools for business executives to systematically organize, understand, and use their data to make strategic decisions. Data warehouse systems are valuable tools in today's competitive, fast-evolving world. In the last several years, many firms have spent millions of dollars in building enterprise-wide data warehouses. Many people feel that with competition mounting in every industry, data warehousing is the latest must-have marketing weapon—a way to keep customers by learning more about their needs.

"Then, what exactly is a data warehouse?" Data warehouses have been defined in many ways, making it difficult to formulate a rigorous definition. Loosely speaking, a data warehouse refers to a database that is maintained separately from an organization's operational databases. Data warehouse systems allow for the integration of a variety of application systems. They support information processing by providing a solid platform of consolidated historical data for analysis.

According to William H. Inmon, a leading architect in the construction of data warehouse systems, "A data warehouse is a subject-oriented, integrated, time-variant, and nonvolatile collection of data in support of management's decision making process" [Inm96]. This short, but comprehensive definition presents the major features of a data warehouse. The four keywords, *subject-oriented, integrated, time-variant*, and *nonvolatile*, distinguish data warehouses from other data repository systems, such as relational database systems, transaction processing systems, and file systems. Let's take a closer look at each of these key features.

- **Subject-oriented**: A data warehouse is organized around major subjects, such as customer, supplier, product, and sales. Rather than concentrating on the day-to-day operations and transaction processing of an organization, a data warehouse focuses on the modeling and analysis of data for decision makers. Hence, data warehouses typically provide a simple and concise view around particular subject issues by excluding data that are not useful in the decision support process.

- **Integrated**: A data warehouse is usually constructed by integrating multiple heterogeneous sources, such as relational databases, flat files, and on-line transaction records. Data cleaning and data integration techniques are applied to ensure consistency in naming conventions, encoding structures, attribute measures, and so on.

- **Time-variant**: Data are stored to provide information from a historical perspective (e.g., the past 5–10 years). Every key structure in the data warehouse contains, either implicitly or explicitly, an element of time.

- **Nonvolatile**: A data warehouse is always a physically separate store of data transformed from the application data found in the operational environment. Due to this separation, a data warehouse does not require transaction processing, recovery, and concurrency control mechanisms. It usually requires only two operations in data accessing: *initial loading of data* and *access of data*.

In sum, a data warehouse is a semantically consistent data store that serves as a physical implementation of a decision support data model and stores the information on which an enterprise needs to make strategic decisions. A data warehouse is also often viewed as an architecture, constructed by integrating data from multiple heterogeneous sources to support structured and/or ad hoc queries, analytical reporting, and decision making.

Based on the above, we view *data warehousing* as the *process of constructing and using data warehouses*. The construction of a data warehouse requires data cleaning, data integration, and data consolidation. The utilization of a data warehouse often necessitates a collection of *decision support* technologies. This allows "knowledge workers" (e.g., managers, analysts, and executives) to use the warehouse to quickly and conveniently obtain an overview of the data, and to make sound decisions based on information in the warehouse. Some authors use the term "data warehousing" to refer only to the process of data warehouse *construction*, while the term "warehouse DBMS" is used to refer to the *management and utilization* of data warehouses. We will not make this distinction here.

"*How are organizations using the information from data warehouses?*" Many organizations use this information to support business decision making activities, including (1) increasing customer focus, which includes the analysis of customer buying patterns (such as buying preference, buying time, budget cycles, and appetites for spending); (2) repositioning products and managing product portfolios by comparing the performance of sales by quarter, by year, and by geographic regions, in order to fine-tune production strategies; (3) analyzing operations and looking for sources of profit; and (4) managing the customer relationships, making environmental corrections, and managing the cost of corporate assets.

Data warehousing is also very useful from the point of view of *heterogeneous database integration*. Many organizations typically collect diverse kinds of data and maintain large databases from multiple, heterogeneous, autonomous, and distributed information sources. To integrate such data, and provide easy and efficient access to it, is highly desirable, yet challenging. Much effort has been spent in the database industry and research community towards achieving this goal.

The traditional database approach to heterogeneous database integration is to build **wrappers** and **integrators** (or **mediators**), on top of multiple, heterogeneous databases. When a query is posed to a client site, a metadata dictionary is used to translate the query into queries appropriate for the individual heterogeneous sites involved. These queries are then mapped and sent to local query processors. The results returned from the different sites

are integrated into a global answer set. This **query-driven approach** requires complex information filtering and integration processes, and competes for resources with processing at local sources. It is inefficient and potentially expensive for frequent queries, especially for queries requiring aggregations.

Data warehousing provides an interesting alternative to the traditional approach of heterogeneous database integration described above. Rather than using a query-driven approach, data warehousing employs an **update-driven** approach in which information from multiple, heterogeneous sources is integrated in advance and stored in a warehouse for direct querying and analysis. Unlike on-line transaction processing databases, data warehouses do not contain the most current information. However, a data warehouse brings high performance to the integrated heterogeneous database system since data are copied, preprocessed, integrated, annotated, summarized, and restructured into one semantic data store. Furthermore, query processing in data warehouses does not interfere with the processing at local sources. Moreover, data warehouses can store and integrate historical information and support complex multidimensional queries. As a result, data warehousing has become popular in industry.

3.1.1 Differences between Operational Database Systems and Data Warehouses

Since most people are familiar with commercial relational database systems, it is easy to understand what a data warehouse is by comparing these two kinds of systems.

The major task of on-line operational database systems is to perform on-line transaction and query processing. These systems are called **on-line transaction processing (OLTP)** systems. They cover most of the day-to-day operations of an organization, such as purchasing, inventory, manufacturing, banking, payroll, registration, and accounting. Data warehouse systems, on the other hand, serve users or knowledge workers in the role of data analysis and decision making. Such systems can organize and present data in various formats in order to accommodate the diverse needs of the different users. These systems are known as **on-line analytical processing (OLAP)** systems.

The major distinguishing features between OLTP and OLAP are summarized as follows.

- **Users and system orientation**: An OLTP system is *customer-oriented* and is used for transaction and query processing by clerks, clients, and information technology professionals. An OLAP system is *market-oriented* and is used for data analysis by knowledge workers, including managers, executives, and analysts.

- **Data contents**: An OLTP system manages current data that, typically, are too detailed to be easily used for decision making. An OLAP system manages large amounts of historical data, provides facilities for summarization and aggregation, and stores and manages information at different levels of granularity. These features make the data easier to use in informed decision making.

- **Database design**: An OLTP system usually adopts an entity-relationship (ER) data model and an application-oriented database design. An OLAP system typically adopts either a *star* or *snowflake* model (to be discussed in Section 3.2.2) and a subject-oriented database design.

- **View**: An OLTP system focuses mainly on the current data within an enterprise or department, without referring to historical data or data in different organizations. In contrast, an OLAP system often spans multiple versions of a database schema, due to the evolutionary process of an organization. OLAP systems also deal with information that originates from different organizations, integrating information from many data stores. Because of their huge volume, OLAP data are stored on multiple storage media.

- **Access patterns**: The access patterns of an OLTP system consist mainly of short, atomic transactions. Such a system requires concurrency control and recovery mechanisms. However, accesses to OLAP systems are mostly read-only operations (since most data warehouses store historical rather than up-to-date information), although many could be complex queries.

Other features that distinguish between OLTP and OLAP systems include database size, frequency of operations, and performance metrics. These are summarized in Table 3.1.

Table 3.1: Comparison between OLTP and OLAP systems.

Feature	OLTP	OLAP
Characteristic	operational processing	informational processing
Orientation	transaction	analysis
User	clerk, DBA, database professional	knowledge worker (e.g., manager, executive, analyst)
Function	day-to-day operations	long-term informational requirements, decision support
DB design	ER based, application-oriented	star/snowflake, subject-oriented
Data	current; guaranteed up-to-date	historical; accuracy maintained over time
Summarization	primitive, highly detailed	summarized, consolidated
View	detailed, flat relational	summarized, multidimensional
Unit of work	short, simple transaction	complex query
Access	read/write	mostly read
Focus	data in	information out
Operations	index/hash on primary key	lots of scans
Number of records accessed	tens	millions
Number of users	thousands	hundreds
DB size	100 MB to GB	100 GB to TB
Priority	high performance, high availability	high flexibility, end-user autonomy
Metric	transaction throughput	query throughput, response time

NOTE: Table is partially based on [CD97].

3.1.2   But, Why Have a Separate Data Warehouse?

Since operational databases store huge amounts of data, you may wonder, *"why not perform on-line analytical processing directly on such databases instead of spending additional time and resources to construct a separate data warehouse?"* A major reason for such a separation is to help promote the *high performance of both systems.* An operational database is designed and tuned from known tasks and workloads, such as indexing and hashing using primary keys, searching for particular records, and optimizing "canned" queries. On the other hand, data warehouse queries are often complex. They involve the computation of large groups of data at summarized levels, and may require the use of special data organization, access, and implementation methods based on multidimensional views. Processing OLAP queries in operational databases would substantially degrade the performance of operational tasks.

Moreover, an operational database supports the concurrent processing of multiple transactions. Concurrency control and recovery mechanisms, such as locking and logging, are required to ensure the consistency and robustness of transactions. An OLAP query often needs read-only access of data records for summarization and aggregation. Concurrency control and recovery mechanisms, if applied for such OLAP operations, may jeopardize the execution of concurrent transactions and thus substantially reduce the throughput of an OLTP system.

Finally, the separation of operational databases from data warehouses is based on the different structures, contents, and uses of the data in these two systems. Decision support requires historical data, whereas operational databases do not typically maintain historical data. In this context, the data in operational databases, though abundant, is usually far from complete for decision making. Decision support requires consolidation (such as aggregation and summarization) of data from heterogeneous sources, resulting in high-quality, clean, and integrated data. In contrast, operational databases contain only detailed raw data, such as transactions, which need to be consolidated before analysis. Since the two systems provide quite different functionalities and require different kinds of data, it is presently necessary to maintain separate databases. However, many vendors of operational relational database management systems are beginning to optimize such systems so as to support OLAP queries. As this trend continues, the separation between OLTP and OLAP systems is expected to decrease.

3.2 A Multidimensional Data Model

Data warehouses and OLAP tools are based on a **multidimensional data model**. This model views data in the form of a *data cube*. In this section, you will learn how data cubes model n-dimensional data. You will also learn about concept hierarchies and how they can be used in basic OLAP operations to allow interactive mining at multiple levels of abstraction.

3.2.1 From Tables and Spreadsheets to Data Cubes

"What is a data cube?" A **data cube** allows data to be modeled and viewed in multiple dimensions. It is defined by dimensions and facts.

In general terms, **dimensions** are the perspectives or entities with respect to which an organization wants to keep records. For example, *AllElectronics* may create a *sales* data warehouse in order to keep records of the store's sales with respect to the dimensions *time*, *item*, *branch*, and *location*. These dimensions allow the store to keep track of things like monthly sales of items, and the branches and locations at which the items were sold. Each dimension may have a table associated with it, called a **dimension table**, which further describes the dimension. For example, a dimension table for *item* may contain the attributes *item_name*, *brand*, and *type*. Dimension tables can be specified by users or experts, or automatically generated and adjusted based on data distributions.

A multidimensional data model is typically organized around a central theme, like *sales*, for instance. This theme is represented by a fact table. **Facts** are numerical measures. Think of them as the quantities by which we want to analyze relationships between dimensions. Examples of facts for a sales data warehouse include *dollars_sold* (sales amount in dollars), *units_sold* (number of units sold), and *amount_budgeted*. The **fact table** contains the names of the *facts*, or measures, as well as keys to each of the related dimension tables. You will soon get a clearer picture of how this works when we look at multidimensional schemas.

Table 3.2: A 2-D view of sales data for *AllElectronics* according to the dimensions *time* and *item*, where the sales are from branches located in the city of Vancouver. The measure displayed is *dollars_sold* (in thousands).

| | *location* = "Vancouver" | | | |
| *time* (quarter) | *item* (type) | | | |
	home entertainment	computer	phone	security
Q1	605	825	14	400
Q2	680	952	31	512
Q3	812	1023	30	501
Q4	927	1038	38	580

Although we usually think of cubes as 3-D geometric structures, in data warehousing the data cube is n-dimensional. To gain a better understanding of data cubes and the multidimensional data model, let's start by looking at a simple 2-D data cube that is, in fact, a table or spreadsheet for sales data from *AllElectronics*. In particular, we will look at the *AllElectronics* sales data for items sold per quarter in the city of Vancouver. These data are shown in Table 3.2. In this 2-D representation, the sales for Vancouver are shown with respect to the *time* dimension (organized in quarters) and the *item* dimension (organized according to the types of items sold). The fact or measure displayed is *dollars_sold* (in thousands).

Now, suppose that we would like to view the sales data with a third dimension. For instance, suppose we would like to view the data according to *time*, *item*, as well as *location* for the cities Chicago, New York, Toronto, and Vancouver. These 3-D data are shown in Table 3.3. The 3-D data of Table 3.3 are represented as a series of 2-D tables. Conceptually, we may also represent the same data in the form of a 3-D data cube, as in Figure 3.1.

Suppose that we would now like to view our sales data with an additional fourth dimension, such as *supplier*.

Table 3.3: A 3-D view of sales data for *AllElectronics*, according to the dimensions *time*, *item*, and *location*. The measure displayed is *dollars_sold* (in thousands).

| t i m e	location = "Chicago" item				location = "New York" item				location = "Toronto" item				location = "Vancouver" item			
	home ent.	comp.	phone	sec.	home ent.	comp.	phone	sec.	home ent.	comp.	phone	sec.	home ent.	comp.	phone	sec.
Q1	854	882	89	623	1087	968	38	872	818	746	43	591	605	825	14	400
Q2	943	890	64	698	1130	1024	41	925	894	769	52	682	680	952	31	512
Q3	1032	924	59	789	1034	1048	45	1002	940	795	58	728	812	1023	30	501
Q4	1129	992	63	870	1142	1091	54	984	978	864	59	784	927	1038	38	580
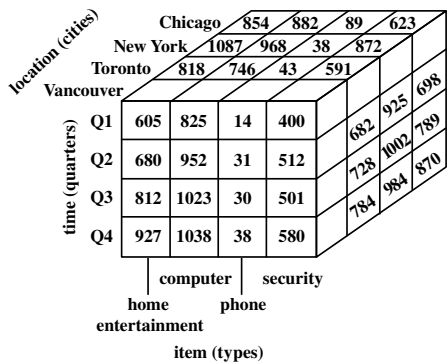
Figure 3.1: A 3-D data cube representation of the data in Table 3.3, according to the dimensions *time*, *item*, and *location*. The measure displayed is *dollars_sold* (in thousands). [TO EDITOR For consistency with text, please kindly show *time*, *item*, and *location* in italics since they are dimension names.]

Viewing things in 4-D becomes tricky. However, we can think of a 4-D cube as being a series of 3-D cubes, as shown in Figure 3.2. If we continue in this way, we may display any n-D data as a series of $(n-1)$-D "cubes." The data cube is a metaphor for multidimensional data storage. The actual physical storage of such data may differ from its logical representation. The important thing to remember is that data cubes are n-dimensional and do not confine data to 3-D.

The above tables show the data at different degrees of summarization. In the data warehousing research literature, a data cube such as each of the above is often referred to as a **cuboid**. Given a set of dimensions, we can generate a cuboid for each of the possible subsets of the given dimensions. The result would form a *lattice* of cuboids, each showing the data at a different level of summarization, or group by. The lattice of cuboids is then referred to as a data cube. Figure 3.3 shows a lattice of cuboids forming a data cube for the dimensions *time*, *item*, *location*, and *supplier*.

The cuboid that holds the lowest level of summarization is called the **base cuboid**. For example, the 4-D cuboid in Figure 3.2 is the base cuboid for the given *time*, *item*, *location*, and *supplier* dimensions. Figure 3.1 is a 3-D (nonbase) cuboid for *time*, *item*, and *location*, summarized for all suppliers. The 0-D cuboid, which holds the highest level of summarization, is called the **apex cuboid**. In our example, this is the total sales, or *dollars_sold*, summarized over all four dimensions. The apex cuboid is typically denoted by all.
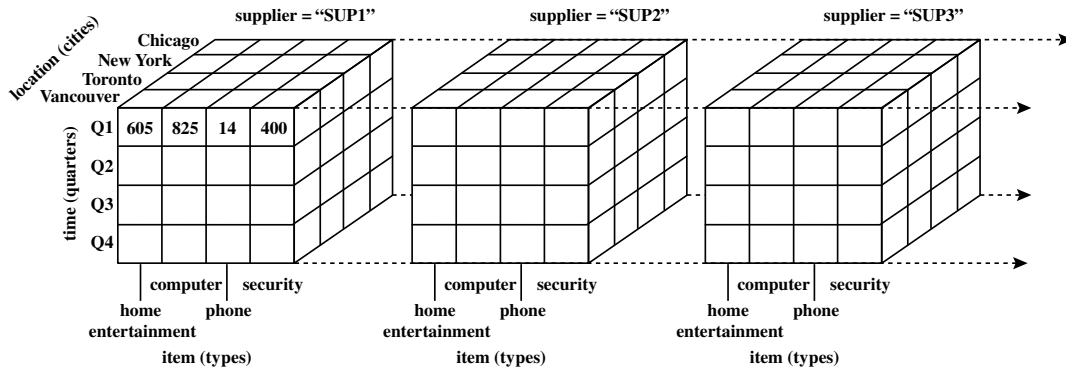
Figure 3.2: A 4-D data cube representation of sales data, according to the dimensions *time*, *item*, *location*, and *supplier*. The measure displayed is *dollars_sold* (in thousands). For improved readability, only some of the cube values are shown. [TO EDITOR For consistency, please show *time*, *item*, *location* and *supplier* in italics since they are dimension names.]

3.2.2 Stars, Snowflakes, and Fact Constellations: Schemas for Multidimensional Databases

The entity-relationship data model is commonly used in the design of relational databases, where a database schema consists of a set of entities and the relationships between them. Such a data model is appropriate for on-line transaction processing. A data warehouse, however, requires a concise, subject-oriented schema that facilitates on-line data analysis.

The most popular data model for a data warehouse is a **multidimensional model**. Such a model can exist in the form of a **star schema**, a **snowflake schema**, or a **fact constellation schema**. Let's have a look at each of these schema types.

Star schema: The most common modeling paradigm is the star schema, in which the data warehouse contains (1) a large central table (**fact table**) containing the bulk of the data, with no redundancy, and (2) a set of smaller attendant tables (**dimension tables**), one for each dimension. The schema graph resembles a starburst, with the dimension tables displayed in a radial pattern around the central fact table.

Example 3.1 Star schema. A star schema for *AllElectronics* sales is shown in Figure 3.4. Sales are considered along four dimensions, namely, *time, item, branch*, and *location*. The schema contains a central fact table for *sales* that contains keys to each of the four dimensions, along with two measures: *dollars_sold* and *units_sold*. To minimize the size of the fact table, dimension identifiers (such as *time_key* and *item_key*) are system-generated identifiers. ∎

Notice that in the star schema, each dimension is represented by only one table, and each table contains a set of attributes. For example, the *location* dimension table contains the attribute set {*location_key, street, city, province_or_state, country*}. This constraint may introduce some redundancy. For example, *"Vancouver"* and *"Victoria"* are both cities in the Canadian province of British Columbia. Entries for such cities in the *location* dimension table will create redundancy among the attributes *province_or_state* and *country*, that is, (..., *Vancouver, British Columbia, Canada*) and (..., *Victoria, British Columbia, Canada*). Moreover, the attributes within a dimension table may form either a hierarchy (total order) or a lattice (partial order).

Snowflake schema: The snowflake schema is a variant of the star schema model, where some dimension tables are *normalized*, thereby further splitting the data into additional tables. The resulting schema graph forms a shape similar to a snowflake.
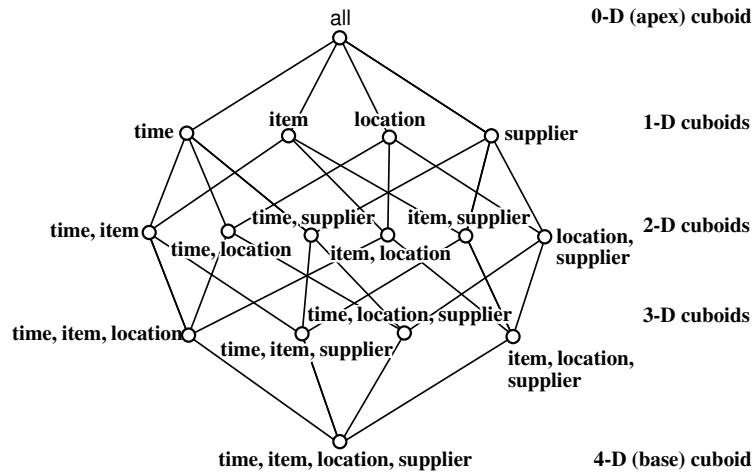
Figure 3.3: Lattice of cuboids, making up a 4-D data cube for the dimensions *time*, *item*, *location*, and *supplier*. Each cuboid represents a different degree of summarization. [TO EDITOR For consistency, please show *time*, *item*, *location* and *supplier* in italics since they are dimension names.]

The major difference between the snowflake and star schema models is that the dimension tables of the snowflake model may be kept in normalized form to reduce redundancies. Such a table is easy to maintain and saves storage space. However, this saving of space is negligible in comparison to the typical magnitude of the fact table. Furthermore, the snowflake structure can reduce the effectiveness of browsing since more joins will be needed to execute a query. Consequently, the system performance may be adversely impacted. Hence, although the snowflake schema reduces redundancy, it is not as popular as the star schema in data warehouse design.

Example 3.2 Snowflake schema. A snowflake schema for *AllElectronics* sales is given in Figure 3.5. Here, the *sales* fact table is identical to that of the star schema in Figure 3.4. The main difference between the two schemas is in the definition of dimension tables. The single dimension table for *item* in the star schema is normalized in the snowflake schema, resulting in new *item* and *supplier* tables. For example, the *item* dimension table now contains the attributes *item_key, item_name, brand, type*, and *supplier_key*, where *supplier_key* is linked to the *supplier* dimension table, containing *supplier_key* and *supplier_type* information. Similarly, the single dimension table for *location* in the star schema can be normalized into two new tables: *location* and *city*. The *city_key* in the new *location* table links to the *city* dimension. Notice that further normalization can be performed on *province_or_state* and *country* in the snowflake schema shown in Figure 3.5, when desirable. ∎

Fact constellation: Sophisticated applications may require multiple fact tables to *share* dimension tables. This kind of schema can be viewed as a collection of stars, and hence is called a **galaxy schema** or a **fact constellation**.

Example 3.3 Fact constellation. A fact constellation schema is shown in Figure 3.6. This schema specifies two fact tables, *sales* and *shipping*. The *sales* table definition is identical to that of the star schema (Figure 3.4). The *shipping* table has five dimensions, or keys: *item_key, time_key, shipper_key, from_location*, and *to_location*, and two measures: *dollars_cost* and *units_shipped*. A fact constellation schema allows dimension tables to be shared between fact tables. For example, the dimensions tables for *time, item*, and *location* are shared between both the *sales* and *shipping* fact tables. ∎

In data warehousing, there is a distinction between a data warehouse and a data mart. A data warehouse collects information about subjects that span the *entire organization*, such as *customers, items, sales, assets*, and
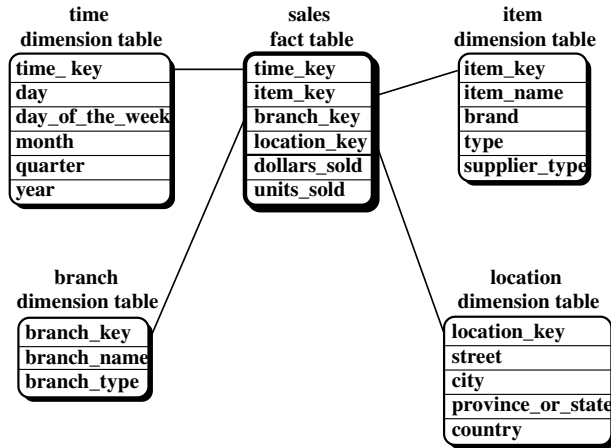
Figure 3.4: Star schema of a data warehouse for sales. [TO EDITOR For consistency, please italicize *time, item, branch, location* and *sales.*]

personnel, and thus its scope is *enterprise-wide*. For data warehouses, the fact constellation schema is commonly used since it can model multiple, interrelated subjects. A **data mart**, on the other hand, is a department subset of the data warehouse that focuses on selected subjects, and thus its scope is *department-wide*. For data marts, the *star* or *snowflake* schema are commonly used since both are geared towards modeling single subjects, although the star schema is more popular and efficient.

3.2.3 Examples for Defining Star, Snowflake, and Fact Constellation Schemas

"How can I define a multidimensional schema for my data?" Just as relational query languages like SQL can be used to specify relational queries, a **data mining query language** can be used to specify data mining tasks. In particular, we examine how to define data warehouses and data marts in our SQL-based data mining query language, **DMQL**.

Data warehouses and data marts can be defined using two language primitives, one for *cube definition* and one for *dimension definition*. The *cube definition* statement has the following syntax.

define cube ⟨cube_name⟩ [⟨dimension_list⟩] : ⟨measure_list⟩

The *dimension definition* statement has the following syntax.

define dimension ⟨dimension_name⟩ as (⟨attribute_or_dimension_list⟩)

Let's look at examples of how to define the star, snowflake, and fact constellation schemas of Examples 3.1 to 3.3 using DMQL. DMQL keywords are displayed in sans serif font.

Example 3.4 Star schema definition. The star schema of Example 3.1 and Figure 3.4 is defined in DMQL as follows.

define cube sales_star [time, item, branch, location]:
                dollars_sold = sum(sales_in_dollars), units_sold = count(*)
define dimension time as (time_key, day, day_of_week, month, quarter, year)
define dimension item as (item_key, item_name, brand, type, supplier_type)
define dimension branch as (branch_key, branch_name, branch_type)
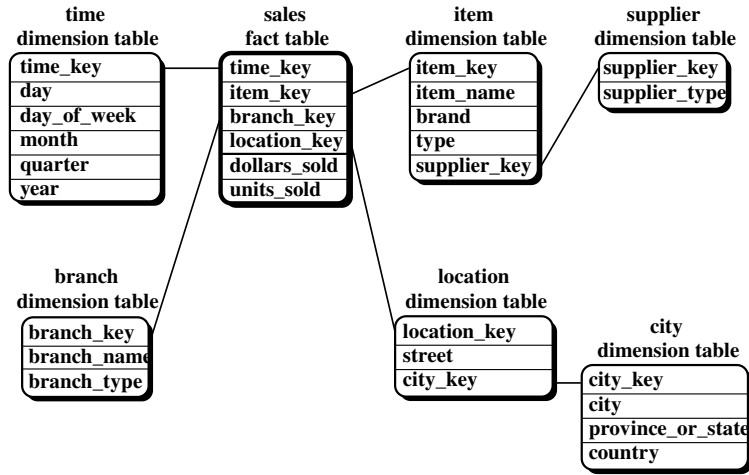define dimension location as (location_key, street, city, province_or_state, country)

Figure 3.5: Snowflake schema of a data warehouse for sales. [TO EDITOR For consistency, please italicize *time, item, supplier, branch, location, sales*, and *city*.]

The **define cube** statement defines a data cube called *sales_star*, which corresponds to the central *sales* fact table of Example 3.1. This command specifies the dimensions and the two measures, *dollars_sold* and *units_sold*. The data cube has four dimensions, namely, *time, item, branch*, and *location*. A **define dimension** statement is used to define each of the dimensions. ∎

Example 3.5 Snowflake schema definition. The snowflake schema of Example 3.2 and Figure 3.5 is defined in DMQL as follows.

> **define cube** sales_snowflake [time, item, branch, location]:
> > dollars_sold = **sum**(sales_in_dollars), units_sold = **count**(*)
> **define dimension** time **as** (time_key, day, day_of_week, month, quarter, year)
> **define dimension** item **as** (item_key, item_name, brand, type, supplier (supplier_key, supplier_type))
> **define dimension** branch **as** (branch_key, branch_name, branch_type)
> **define dimension** location **as** (location_key, street, city (city_key, city, province_or_state, country))

This definition is similar to that of *sales_star* (Example 3.4), except that, here, the *item* and *location* dimension tables are normalized. For instance, the *item* dimension of the *sales_star* data cube has been normalized in the *sales_snowflake* cube into two dimension tables, *item* and *supplier*. Note that the dimension definition for *supplier* is specified within the definition for *item*. Defining *supplier* in this way implicitly creates a *supplier_key* in the *item* dimension table definition. Similarly, the *location* dimension of the *sales_star* data cube has been normalized in the *sales_snowflake* cube into two dimension tables, *location* and *city*. The dimension definition for *city* is specified within the definition for *location*. In this way, a *city_key* is implicitly created in the *location* dimension table definition. ∎

Finally, a fact constellation schema can be defined as a set of interconnected cubes. Below is an example.

Example 3.6 Fact constellation schema definition. The fact constellation schema of Example 3.3 and Figure 3.6 is defined in DMQL as follows.

> **define cube** sales [time, item, branch, location]:
> > dollars_sold = **sum**(sales_in_dollars), units_sold = **count**(*)
> **define dimension** time **as** (time_key, day, day_of_week, month, quarter, year)
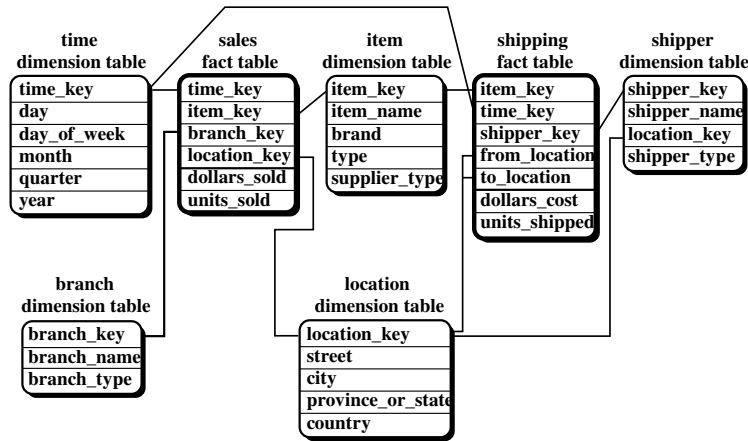
Figure 3.6: Fact constellation schema of a data warehouse for sales and shipping. [TO EDITOR For consistency, please italicize *time*, *item*, *shipping*, *shipper*, *branch*, *location*, and *sales*.]

define dimension item as (item_key, item_name, brand, type, supplier_type)
define dimension branch as (branch_key, branch_name, branch_type)
define dimension location as (location_key, street, city, province_or_state, country)

define cube shipping [time, item, shipper, from_location, to_location]:
$$\text{dollars\_cost} = \text{sum(cost\_in\_dollars)}, \text{units\_shipped} = \text{count}(*)$$
define dimension time as time in cube sales
define dimension item as item in cube sales
define dimension shipper as (shipper_key, shipper_name, location as location in cube sales, shipper_type)
define dimension from_location as location in cube sales
define dimension to_location as location in cube sales

A define cube statement is used to define data cubes for *sales* and *shipping*, corresponding to the two fact tables of the schema of Example 3.3. Note that the *time, item*, and *location* dimensions of the *sales* cube are shared with the *shipping* cube. This is indicated for the *time* dimension, for example, as follows. Under the define cube statement for *shipping*, the statement "define dimension *time* as *time* in cube *sales*" is specified. ∎

3.2.4 Measures: Their Categorization and Computation

"How are measures computed?" To answer this question, we first study how measures can be categorized. [1]. Note that a *multidimensional point* in the data cube space can be defined by a set of dimension-value pairs, for example, ⟨*time = "Q1", location = "Vancouver", item = "computer"*⟩. A data cube **measure** is a numerical function that can be evaluated at each point in the data cube space. A measure value is computed for a given point by aggregating the data corresponding to the respective dimension-value pairs defining the given point. We will look at concrete examples of this shortly.

Measures can be organized into three categories, based on the kind of aggregate functions used.

Distributive: An aggregate function is *distributive* if it can be computed in a distributed manner as follows. Suppose the data are partitioned into n sets. We apply the function to each partition, resulting in n aggregate values. If the result derived by applying the function to the n aggregate values is the same as that derived by applying the function to the entire data set (without partitioning), the function can be computed in a

---
[1]This categorization was briefly introduced in Chapter 2 with regards to the computation of measures for descriptive data summaries. We re-examine it here in the context of data cube measures.

distributed manner. For example, count() can be computed for a data cube by first partitioning the cube into a set of subcubes, computing count() for each subcube, and then summing up the counts obtained for each subcube. Hence, count() is a distributive aggregate function. For the same reason, sum(), min(), and max() are distributive aggregate functions. A measure is *distributive* if it is obtained by applying a distributive aggregate function. Distributive measures can be computed efficiently since they can be computed in a distributive manner.

Algebraic: An aggregate function is *algebraic* if it can be computed by an algebraic function with M arguments (where M is a bounded positive integer), each of which is obtained by applying a distributive aggregate function. For example, avg() (average) can be computed by sum()/count() where both sum() and count() are distributive aggregate functions. Similarly, it can be shown that min_N() and max_N() (which find the N minimum and N maximum values, respectively, in a given set), and standard_deviation() are algebraic aggregate functions. A measure is *algebraic* if it is obtained by applying an algebraic aggregate function.

Holistic: An aggregate function is *holistic* if there is no constant bound on the storage size needed to describe a subaggregate. That is, there does not exist an algebraic function with M arguments (where M is a constant) that characterizes the computation. Common examples of holistic functions include median(), mode(), and rank(). A measure is *holistic* if it is obtained by applying a holistic aggregate function.

Most large data cube applications require efficient computation of distributive and algebraic measures. Many efficient techniques for this exist. In contrast, it is difficult to compute holistic measures efficiently. Efficient techniques to *approximate* the computation of some holistic measures, however, do exist. For example, rather than computing the exact median(), Equation (2.3) of Chapter 2 can be used to estimate the approximate median value for a large data set. In many cases, such techniques are sufficient to overcome the difficulties of efficient computation of holistic measures.

Example 3.7 Interpreting measures for data cubes. Many measures of a data cube can be computed by relational aggregation operations. In Figure 3.4, we saw a star schema for *AllElectronics* sales that contains two measures, namely, *dollars_sold* and *units_sold*. In Example 3.4, the *sales_star* data cube corresponding to the schema was defined using DMQL commands. *"But, how are these commands interpreted in order to generate the specified data cube?"*

Suppose that the relational database schema of *AllElectronics* is the following:

time(time_key, day, day_of_week, month, quarter, year)
item(item_key, item_name, brand, type, supplier_type)
branch(branch_key, branch_name, branch_type)
location(location_key, street, city, province_or_state, country)
sales(time_key, item_key, branch_key, location_key, number_of_units_sold, price)

The DMQL specification of Example 3.4 is translated into the following SQL query, which generates the required *sales_star* cube. Here, the sum aggregate function, is used to compute both *dollars_sold* and *units_sold*.

```
select  s.time_key, s.item_key, s.branch_key, s.location_key,
             sum(s.number_of_units_sold * s.price), sum(s.number_of_units_sold)
from time t, item i, branch b, location l, sales s,
where s.time_key = t.time_key and s.item_key = i.item_key
             and s.branch_key = b.branch_key and s.location_key = l.location_key
group by s.time_key, s.item_key, s.branch_key, s.location_key
```

The cube created in the above query is the base cuboid of the *sales_star* data cube. It contains all of the dimensions specified in the data cube definition, where the granularity of each dimension is at the **join key** level. A join key is a key that links a fact table and a dimension table. The fact table associated with a base cuboid is sometimes referred to as the **base fact table**.

By changing the group by clauses, we can generate other cuboids for the *sales_star* data cube. For example, instead of grouping by *s.time_key*, we can group by *t.month*, which will sum up the measures of each group by month. Also, removing "group by *s.branch_key*" will generate a higher-level cuboid (where sales are summed for all branches, rather than broken down per branch). Suppose we modify the above SQL query by removing *all* of the group by clauses. This will result in obtaining the total sum of *dollars_sold* and the total count of *units_sold* for the given data. This zero-dimensional cuboid is the apex cuboid of the *sales_star* data cube. In addition, other cuboids can be generated by applying selection and/or projection operations on the base cuboid, resulting in a lattice of cuboids as described in Section 3.2.1. Each cuboid corresponds to a different degree of summarization of the given data. ∎

Most of the current data cube technology confines the measures of multidimensional databases to *numerical data*. However, measures can also be applied to other kinds of data, such as spatial, multimedia, or text data. This will be discussed in future chapters.

3.2.5 Concept Hierarchies

A **concept hierarchy** defines a sequence of mappings from a set of low-level concepts to higher-level, more general concepts. Consider a concept hierarchy for the dimension *location*. City values for *location* include Vancouver, Toronto, New York, and Chicago. Each city, however, can be mapped to the province or state to which it belongs. For example, Vancouver can be mapped to British Columbia, and Chicago to Illinois. The provinces and states can in turn be mapped to the country to which they belong, such as Canada or the USA. These mappings form a concept hierarchy for the dimension *location*, mapping a set of low-level concepts (i.e., cities) to higher-level, more general concepts (i.e., countries). The concept hierarchy described above is illustrated in Figure 3.7.
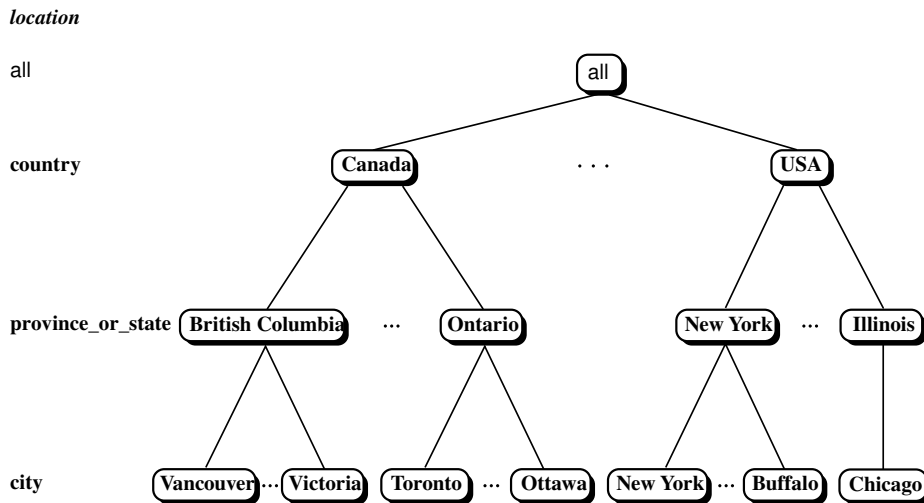
Figure 3.7: A concept hierarchy for the dimension *location*. Due to space limitations, not all of the nodes of the hierarchy are shown (as indicated by the use of "ellipsis" between nodes). [TO EDITOR Please add ellipsis and a node for the city Urbana under the Illinois node. Thanks!]

Many concept hierarchies are implicit within the database schema. For example, suppose that the dimension *location* is described by the attributes *number, street, city, province_or_state, zipcode*, and *country*. These attributes are related by a total order, forming a concept hierarchy such as "*street < city < province_or_state < country*". This hierarchy is shown in Figure 3.8(a). Alternatively, the attributes of a dimension may be organized in a partial order, forming a lattice. An example of a partial order for the *time* dimension based on the attributes *day, week, month, quarter*, and *year* is "*day < {month <quarter; week} < year*".[2] This lattice structure is shown

---

[2]Since a *week* often crosses the boundary of two consecutive months, it is usually not treated as a lower abstraction of *month*.

in Figure 3.8(b). A concept hierarchy that is a total or partial order among attributes in a database schema is called a **schema hierarchy**. Concept hierarchies that are common to many applications may be predefined in the data mining system, such as the concept hierarchy for *time*. Data mining systems should provide users with the flexibility to tailor predefined hierarchies according to their particular needs. For example, users may like to define a fiscal year starting on April 1, or an academic year starting on September 1.

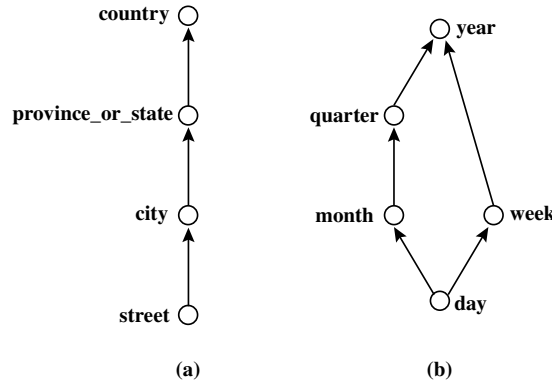**(a)**   **(b)**

Figure 3.8: Hierarchical and lattice structures of attributes in warehouse dimensions: (a) a hierarchy for *location*; (b) a lattice for *time*.
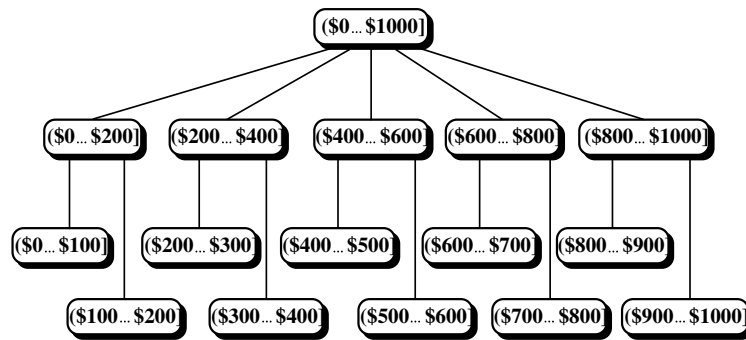
Figure 3.9: A concept hierarchy for the attribute *price*. [TO EDITOR Please change so that all of the categories 0-100, 100-200, 200-300, etc. appear on the same level! (This would be better, conceptually, for the reader! Thanks.)]

Concept hierarchies may also be defined by discretizing or grouping values for a given dimension or attribute, resulting in a **set-grouping hierarchy**. A total or partial order can be defined among groups of values. An example of a set-grouping hierarchy is shown in Figure 3.9 for the dimension *price*, where an interval ($X \ldots $Y] denotes the range from $X (exclusive) to $Y (inclusive).

There may be more than one concept hierarchy for a given attribute or dimension, based on different user viewpoints. For instance, a user may prefer to organize *price* by defining ranges for *inexpensive, moderately_priced*, and *expensive*.

Concept hierarchies may be provided manually by system users, domain experts, knowledge engineers, or automatically generated based on statistical analysis of the data distribution. The automatic generation of concept hierarchies is discussed in Chapter 2 as a preprocessing step in preparation for data mining.

Concept hierarchies allow data to be handled at varying levels of abstraction, as we shall see in the following

---

Instead, it is often treated as a lower abstraction of *year*, since a year contains approximately 52 weeks.

subsection.

3.2.6 OLAP Operations in the Multidimensional Data Model
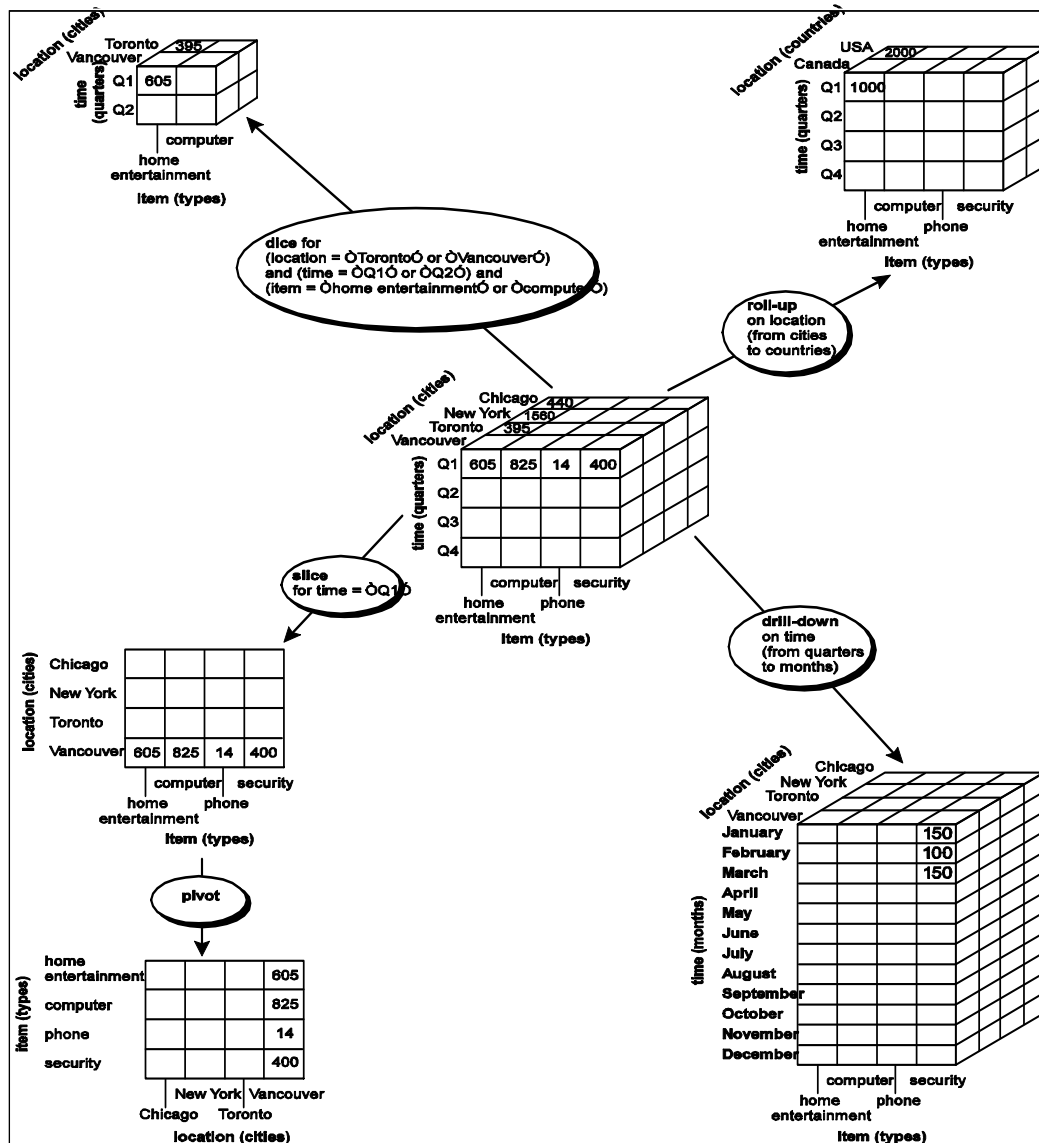
Figure 3.10: Examples of typical OLAP operations on multidimensional data. [TO EDITOR For consistency, please italicize *time*, *item*, and *location*.]

"How are concept hierarchies useful in OLAP?" In the multidimensional model, data are organized into multiple dimensions, and each dimension contains multiple levels of abstraction defined by concept hierarchies. This organization provides users with the flexibility to view data from different perspectives. A number of OLAP data cube operations exist to materialize these different views, allowing interactive querying and analysis of the data at hand. Hence, OLAP provides a user-friendly environment for interactive data analysis.

Example 3.8 OLAP operations. Let's have a look at some typical OLAP operations for multidimensional

data. Each of the operations described below is illustrated in Figure 3.10. At the center of the figure is a data cube for *AllElectronics* sales. The cube contains the dimensions *location, time*, and *item*, where *location* is aggregated with respect to city values, *time* is aggregated with respect to quarters, and *item* is aggregated with respect to item types. To aid in our explanation, we refer to this cube as the central cube. The measure displayed is *dollars_sold* (in thousands). (For improved readability, only some of the cubes' cell values are shown.) The data examined are for the cities Chicago, New York, Toronto, and Vancouver.

Roll-up: The roll-up operation (also called the *drill-up* operation by some vendors) performs aggregation on a data cube, either by *climbing up a concept hierarchy* for a dimension or by *dimension reduction*. Figure 3.10 shows the result of a roll-up operation performed on the central cube by climbing up the concept hierarchy for *location* given in Figure 3.7. This hierarchy was defined as the total order "*street < city < province_or_state < country*". The roll-up operation shown aggregates the data by ascending the *location* hierarchy from the level of *city* to the level of *country*. In other words, rather than grouping the data by city, the resulting cube groups the data by country.

When roll-up is performed by dimension reduction, one or more dimensions are removed from the given cube. For example, consider a sales data cube containing only the two dimensions *location* and *time*. Roll-up may be performed by removing, say, the *time* dimension, resulting in an aggregation of the total sales by location, rather than by location and by time.

Drill-down: Drill-down is the reverse of roll-up. It navigates from less detailed data to more detailed data. Drill-down can be realized by either *stepping down a concept hierarchy* for a dimension or *introducing additional dimensions*. Figure 3.10 shows the result of a drill-down operation performed on the central cube by stepping down a concept hierarchy for *time* defined as "*day < month < quarter < year*". Drill-down occurs by descending the *time* hierarchy from the level of *quarter* to the more detailed level of *month*. The resulting data cube details the total sales per month rather than summarized by quarter.

Since a drill-down adds more detail to the given data, it can also be performed by adding new dimensions to a cube. For example, a drill-down on the central cube of Figure 3.10 can occur by introducing an additional dimension, such as *customer_group*.

Slice and dice: The *slice* operation performs a selection on one dimension of the given cube, resulting in a subcube. Figure 3.10 shows a slice operation where the sales data are selected from the central cube for the dimension *time* using the criterion *time = "Q1"*. The *dice* operation defines a subcube by performing a selection on two or more dimensions. Figure 3.10 shows a dice operation on the central cube based on the following selection criteria that involve three dimensions: (*location = "Toronto"* or *"Vancouver"*) and (*time = "Q1"* or *"Q2"*) and (*item = "home entertainment"* or *"computer"*).

Pivot (rotate): *Pivot* (also called *rotate*) is a visualization operation that rotates the data axes in view in order to provide an alternative presentation of the data. Figure 3.10 shows a pivot operation where the *item* and *location* axes in a 2-D slice are rotated. Other examples include rotating the axes in a 3-D cube, or transforming a 3-D cube into a series of 2-D planes.

Other OLAP operations: Some OLAP systems offer additional drilling operations. For example, **drill-across** executes queries involving (i.e., across) more than one fact table. The **drill-through** operation makes use of relational SQL facilities to drill through the bottom level of a data cube down to its back-end relational tables.

Other OLAP operations may include ranking the top N or bottom N items in lists, as well as computing moving averages, growth rates, interests, internal rates of return, depreciation, currency conversions, and statistical functions.   ∎

OLAP offers analytical modeling capabilities, including a calculation engine for deriving ratios, variance, and so on, and for computing measures across multiple dimensions. It can generate summarizations, aggregations, and hierarchies at each granularity level and at every dimension intersection. OLAP also supports functional models for forecasting, trend analysis, and statistical analysis. In this context, an OLAP engine is a powerful data analysis tool.

OLAP Systems versus Statistical Databases

Many of the characteristics of OLAP systems, such as the use of a multidimensional data model and concept hierarchies, the association of measures with dimensions, and the notions of roll-up and drill-down, also exist in earlier work on statistical databases (SDBs). A **statistical database** is a database system that is designed to support statistical applications. Similarities between the two types of systems are rarely discussed, mainly due to differences in terminology and application domains.

OLAP and SDB systems, however, have distinguishing differences. While SDBs tend to focus on socioeconomic applications, OLAP has been targeted for business applications. Privacy issues regarding concept hierarchies are a major concern for SDBs. For example, given summarized socioeconomic data, it is controversial to allow users to view the corresponding low-level data. Finally, unlike SDBs, OLAP systems are designed for handling huge amounts of data efficiently.

3.2.7   A Starnet Query Model for Querying Multidimensional Databases

The querying of multidimensional databases can be based on a **starnet model**. A starnet model consists of radial lines emanating from a central point, where each line represents a concept hierarchy for a dimension. Each abstraction level in the hierarchy is called a **footprint**. These represent the granularities available for use by OLAP operations such as drill-down and roll-up.
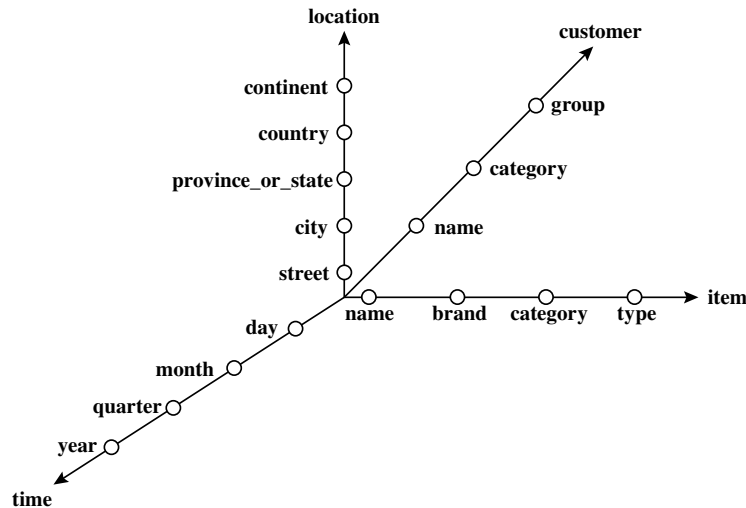
Figure 3.11: Modeling business queries: a starnet model. [TO EDITOR For consistency, please italicize *time*, *item*, *customer*, and *location*.]

Example 3.9 Starnet. A starnet query model for the *AllElectronics* data warehouse is shown in Figure 3.11. This starnet consists of four radial lines, representing concept hierarchies for the dimensions *location, customer, item*, and *time*, respectively. Each line consists of footprints representing abstraction levels of the dimension. For example, the *time* line has four footprints: "day," "month," "quarter," and "year." A concept hierarchy may involve a single attribute (like *date* for the *time* hierarchy), or several attributes (e.g., the concept hierarchy for *location* involves the attributes *street, city, province_or_state*, and *country*). In order to examine the item sales at *AllElectronics*, users can roll up along the *time* dimension from *month* to *quarter*, or, say, drill down along the *location* dimension from *country* to *city*. Concept hierarchies can be used to **generalize** data by replacing low-level values (such as "day" for the *time* dimension) by higher-level abstractions (such as "year"), or to **specialize** data by replacing higher-level abstractions with lower-level values. ■

3.3   Data Warehouse Architecture

In this section, we discuss issues regarding data warehouse architecture. Section 3.3.1 gives a general account of how to design and construct a data warehouse. Section 3.3.2 describes a three-tier data warehouse architecture. Section 3.3.3 describes back-end tools and utilities for data warehouses. Section 3.3.4 describes the metadata repository. Section 3.3.5 presents various types of warehouse servers for OLAP processing.

3.3.1   Steps for the Design and Construction of Data Warehouses

This subsection presents a business analysis framework for data warehouse design. The basic steps involved in the design process are also described.

The Design of a Data Warehouse: A Business Analysis Framework

"*What can business analysts gain from having a data warehouse?*" First, having a data warehouse may provide a *competitive advantage* by presenting relevant information from which to measure performance and make critical adjustments in order to help win over competitors. Second, a data warehouse can enhance business *productivity* since it is able to quickly and efficiently gather information that accurately describes the organization. Third, a data warehouse facilitates *customer relationship management* since it provides a consistent view of customers and items across all lines of business, all departments, and all markets. Finally, a data warehouse may bring about *cost reduction* by tracking trends, patterns, and exceptions over long periods of time in a consistent and reliable manner.

To design an effective data warehouse we need to understand and analyze business needs and construct a *business analysis framework*. The construction of a large and complex information system can be viewed as the construction of a large and complex building, for which the owner, architect, and builder have different views. These views are combined to form a complex framework that represents the top-down, business-driven, or owner's perspective, as well as the bottom-up, builder-driven, or implementor's view of the information system.

Four different views regarding the design of a data warehouse must be considered: the *top-down view*, the *data source view*, the *data warehouse view*, and the *business query view*.

- The **top-down view** allows the selection of the relevant information necessary for the data warehouse. This information matches the current and coming business needs.

- The **data source view** exposes the information being captured, stored, and managed by operational systems. This information may be documented at various levels of detail and accuracy, from individual data source tables to integrated data source tables. Data sources are often modeled by traditional data modeling techniques, such as the entity-relationship model or CASE (computer-aided software engineering) tools.

- The **data warehouse view** includes fact tables and dimension tables. It represents the information that is stored inside the data warehouse, including precalculated totals and counts, as well as information regarding the source, date, and time of origin, added to provide historical context.

- Finally, the **business query view** is the perspective of data in the data warehouse from the viewpoint of the end user.

Building and using a data warehouse is a complex task since it requires *business skills, technology skills*, and *program management skills*. Regarding *business skills*, building a data warehouse involves understanding how such systems store and manage their data, how to build **extractors** that transfer data from the operational system to the data warehouse, and how to build **warehouse refresh software** that keeps the data warehouse reasonably up-to-date with the operational system's data. Using a data warehouse involves understanding the significance of the data it contains, as well as understanding and translating the business requirements into queries that can be satisfied by the data warehouse. Regarding *technology skills*, data analysts are required to understand how to make

assessments from quantitative information and derive facts based on conclusions from historical information in the data warehouse. These skills include the ability to discover patterns and trends, to extrapolate trends based on history and look for anomalies or paradigm shifts, and to present coherent managerial recommendations based on such analysis. Finally, *program management skills* involve the need to interface with many technologies, vendors, and end users in order to deliver results in a timely and cost-effective manner.

The Process of Data Warehouse Design

A data warehouse can be built using a *top-down approach*, a *bottom-up approach*, or a *combination of both*. The **top-down approach** starts with the overall design and planning. It is useful in cases where the technology is mature and well known, and where the business problems that must be solved are clear and well understood. The **bottom-up approach** starts with experiments and prototypes. This is useful in the early stage of business modeling and technology development. It allows an organization to move forward at considerably less expense and to evaluate the benefits of the technology before making significant commitments. In the **combined approach**, an organization can exploit the planned and strategic nature of the top-down approach while retaining the rapid implementation and opportunistic application of the bottom-up approach.

From the software engineering point of view, the design and construction of a data warehouse may consist of the following steps: *planning, requirements study, problem analysis, warehouse design, data integration and testing*, and finally *deployment of the data warehouse*. Large software systems can be developed using two methodologies: the *waterfall method* or the *spiral method*. The **waterfall method** performs a structured and systematic analysis at each step before proceeding to the next, which is like a waterfall, falling from one step to the next. The **spiral method** involves the rapid generation of increasingly functional systems, with short intervals between successive releases. This is considered a good choice for data warehouse development, especially for data marts, because the turnaround time is short, modifications can be done quickly, and new designs and technologies can be adapted in a timely manner.

In general, the warehouse design process consists of the following steps:

1. Choose a *business process* to model, for example, orders, invoices, shipments, inventory, account administration, sales, or the general ledger. If the business process is organizational and involves multiple complex object collections, a data warehouse model should be followed. However, if the process is departmental and focuses on the analysis of one kind of business process, a data mart model should be chosen.

2. Choose the *grain* of the business process. The grain is the fundamental, atomic level of data to be represented in the fact table for this process, for example, individual transactions, individual daily snapshots, and so on.

3. Choose the *dimensions* that will apply to each fact table record. Typical dimensions are time, item, customer, supplier, warehouse, transaction type, and status.

4. Choose the *measures* that will populate each fact table record. Typical measures are numeric additive quantities like *dollars_sold* and *units_sold*.

Since data warehouse construction is a difficult and long-term task, its implementation scope should be clearly defined. The goals of an initial data warehouse implementation should be *specific, achievable*, and *measurable*. This involves determining the time and budget allocations, the subset of the organization that is to be modeled, the number of data sources selected, and the number and types of departments to be served.

Once a data warehouse is designed and constructed, the initial deployment of the warehouse includes initial installation, rollout planning, training, and orientation. Platform upgrades and maintenance must also be considered. Data warehouse administration includes data refreshment, data source synchronization, planning for disaster recovery, managing access control and security, managing data growth, managing database performance, and data warehouse enhancement and extension. Scope management includes controlling the number and range of queries, dimensions, and reports; limiting the size of the data warehouse; or limiting the schedule, budget, or resources.

Various kinds of data warehouse design tools are available. **Data warehouse development tools** provide functions to define and edit metadata repository contents (such as schemas, scripts, or rules), answer queries, output reports, and ship metadata to and from relational database system catalogues. **Planning and analysis tools** study the impact of schema changes and of refresh performance when changing refresh rates or time windows.

3.3.2 A Three-Tier Data Warehouse Architecture

Data warehouses often adopt a three-tier architecture, as presented in Figure 3.12.
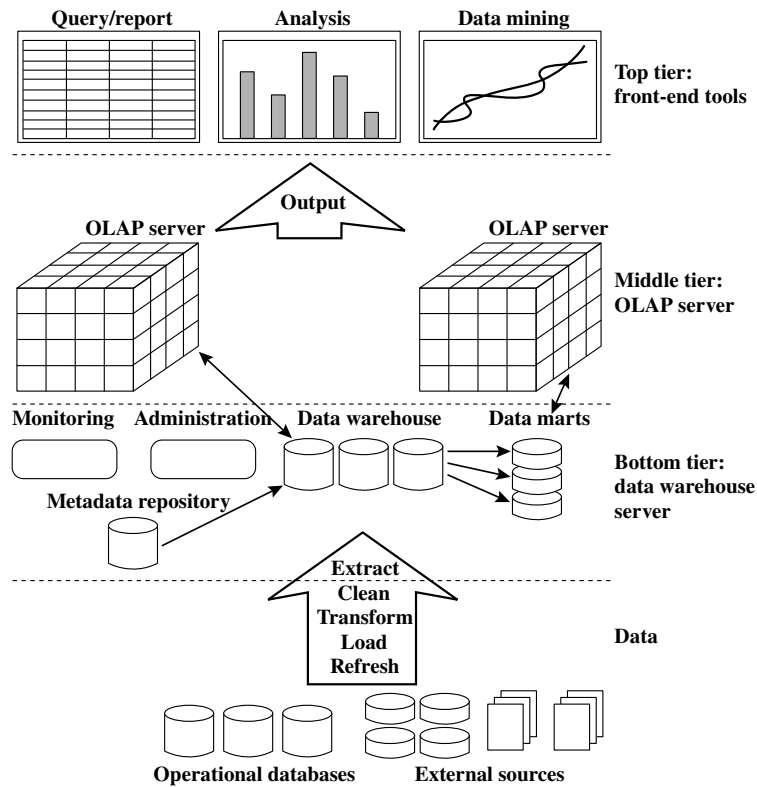
Figure 3.12: A three-tier data warehousing architecture.

1. The bottom tier is a **warehouse database server** that is almost always a relational database system. Back-end tools and utilities are used to feed data into the bottom tier from operational databases or other external sources (such as customer profile information provided by external consultants). These tools and utilities perform data extraction, cleaning, and transformation (e.g., to merge like data from different sources into a unified format), as well as load and refresh functions to update the data warehouse (Section 3.3.3). The data are extracted using application program interfaces known as **gateways**. A gateway is supported by the underlying DBMS and allows client programs to generate SQL code to be executed at a server. Examples of gateways include ODBC (Open Database Connection) and OLE-DB (Open Linking and Embedding for Databases) by Microsoft, and JDBC (Java Database Connection). This tier also contains a metadata repository, which stores information about the data warehouse and its contents. The metadata repository is further described in Section 3.3.4.

2. The middle tier is an **OLAP server** that is typically implemented using either (1) a **relational OLAP (ROLAP)** model, that is, an extended relational DBMS that maps operations on multidimensional data to

standard relational operations; or (2) a **multidimensional OLAP (MOLAP)** model, that is, a special-purpose server that directly implements multidimensional data and operations. OLAP servers are discussed in Section 3.3.5.

3. The top tier is a **front-end client layer**, which contains query and reporting tools, analysis tools, and/or data mining tools (e.g., trend analysis, prediction, and so on).

From the architecture point of view, there are three data warehouse models: the *enterprise warehouse*, the *data mart*, and the *virtual warehouse.*

Enterprise warehouse: An enterprise warehouse collects all of the information about subjects spanning the entire organization. It provides corporate-wide data integration, usually from one or more operational systems or external information providers, and is cross-functional in scope. It typically contains detailed data as well as summarized data, and can range in size from a few gigabytes to hundreds of gigabytes, terabytes, or beyond. An enterprise data warehouse may be implemented on traditional mainframes, computer superservers, or parallel architecture platforms. It requires extensive business modeling and may take years to design and build.

Data mart: A data mart contains a subset of corporate-wide data that is of value to a specific group of users. The scope is confined to specific selected subjects. For example, a marketing data mart may confine its subjects to customer, item, and sales. The data contained in data marts tend to be summarized.

Data marts are usually implemented on low-cost departmental servers that are UNIX/LINUX- or Windows-based. The implementation cycle of a data mart is more likely to be measured in weeks rather than months or years. However, it may involve complex integration in the long run if its design and planning were not enterprise-wide.

Depending on the source of data, data marts can be categorized as independent or dependent. *Independent* data marts are sourced from data captured from one or more operational systems or external information providers, or from data generated locally within a particular department or geographic area. *Dependent* data marts are sourced directly from enterprise data warehouses.

Virtual warehouse: A virtual warehouse is a set of views over operational databases. For efficient query processing, only some of the possible summary views may be materialized. A virtual warehouse is easy to build but requires excess capacity on operational database servers.

"*What are the pros and cons of the top-down and bottom-up approaches to data warehouse development?*" The top-down development of an enterprise warehouse serves as a systematic solution and minimizes integration problems. However, it is expensive, takes a long time to develop, and lacks flexibility due to the difficulty in achieving consistency and consensus for a common data model for the entire organization. The bottom-up approach to the design, development, and deployment of independent data marts provides flexibility, low cost, and rapid return of investment. It, however, can lead to problems when integrating various disparate data marts into a consistent enterprise data warehouse.

A recommended method for the development of data warehouse systems is to implement the warehouse in an incremental and evolutionary manner, as shown in Figure 3.13. First, a high-level corporate data model is defined within a reasonably short period of time (such as one or two months) that provides a corporate-wide, consistent, integrated view of data among different subjects and potential usages. This high-level model, although it will need to be refined in the further development of enterprise data warehouses and departmental data marts, will greatly reduce future integration problems. Second, independent data marts can be implemented in parallel with the enterprise warehouse based on the same corporate data model set as above. Third, distributed data marts can be constructed to integrate different data marts via hub servers. Finally, a **multitier data warehouse** is constructed where the enterprise warehouse is the sole custodian of all warehouse data, which is then distributed to the various dependent data marts.
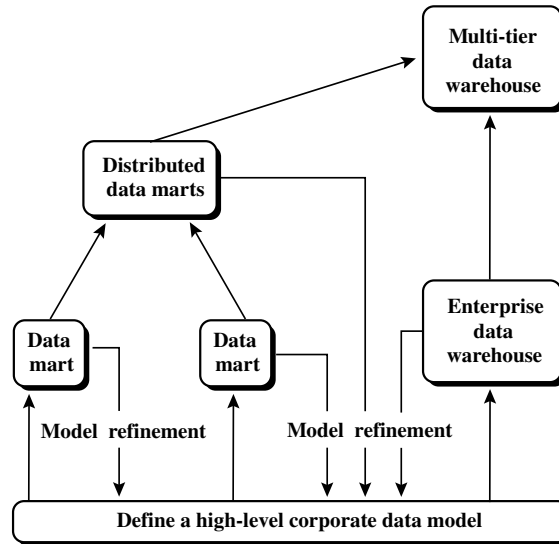
Figure 3.13: A recommended approach for data warehouse development. [TO EDITOR For consistency, please change 'multi-tier' to 'multitier'.]

3.3.3   Data Warehouse Back-End Tools and Utilities

Data warehouse systems use back-end tools and utilities to populate and refresh their data (Figure 3.12). These tools and utilities include the following functions:

- **Data extraction**, which typically gathers data from multiple, heterogeneous, and external sources

- **Data cleaning**, which detects errors in the data and rectifies them when possible

- **Data transformation**, which converts data from legacy or host format to warehouse format

- **Load**, which sorts, summarizes, consolidates, computes views, checks integrity, and builds indices and partitions

- **Refresh**, which propagates the updates from the data sources to the warehouse

Besides cleaning, loading, refreshing, and metadata definition tools, data warehouse systems usually provide a good set of data warehouse management tools.

Data cleaning and data transformation are important steps in improving the quality of the data and, subsequently, of the data mining results. They are described in Chapter 2 on Data Preprocessing. Since we are mostly interested in the aspects of data warehousing technology related to data mining, we will not get into the details of the remaining tools and recommend interested readers to consult books dedicated to data warehousing technology.

3.3.4   Metadata Repository

Metadata are data about data. When used in a data warehouse, metadata are the data that define warehouse objects. Figure 3.12 showed a metadata repository within the bottom tier of the data warehousing architecture. Metadata are created for the data names and definitions of the given warehouse. Additional metadata are created and captured for timestamping any extracted data, the source of the extracted data, and missing fields that have been added by data cleaning or integration processes.

A metadata repository should contain the following:

- A description of *the structure of the data warehouse*, which includes the warehouse schema, view, dimensions, hierarchies, and derived data definitions, as well as data mart locations and contents

- *Operational metadata*, which include data lineage (history of migrated data and the sequence of transformations applied to it), currency of data (active, archived, or purged), and monitoring information (warehouse usage statistics, error reports, and audit trails)

- *The algorithms used for summarization*, which include measure and dimension definition algorithms, data on granularity, partitions, subject areas, aggregation, summarization, and predefined queries and reports

- *The mapping from the operational environment to the data warehouse*, which includes source databases and their contents, gateway descriptions, data partitions, data extraction, cleaning, transformation rules and defaults, data refresh and purging rules, and security (user authorization and access control)

- *Data related to system performance*, which include indices and profiles that improve data access and retrieval performance, in addition to rules for the timing and scheduling of refresh, update, and replication cycles

- *Business metadata*, which include business terms and definitions, data ownership information, and charging policies

A data warehouse contains different levels of summarization, of which metadata is one type. Other types include current detailed data (which are almost always on disk), older detailed data (which are usually on tertiary storage), lightly summarized data and highly summarized data (which may or may not be physically housed).

Metadata play a very different role than other data warehouse data, and are important for many reasons. For example, metadata are used as a directory to help the decision support system analyst locate the contents of the data warehouse, as a guide to the mapping of data when the data are transformed from the operational environment to the data warehouse environment, and as a guide to the algorithms used for summarization between the current detailed data and the lightly summarized data, and between the lightly summarized data and the highly summarized data. Metadata should be stored and managed persistently (i.e., on disk).

3.3.5   Types of OLAP Servers: ROLAP versus MOLAP versus HOLAP

Logically, OLAP servers present business users with multidimensional data from data warehouses or data marts, without concerns regarding how or where the data are stored. However, the physical architecture and implementation of OLAP servers must consider data storage issues. Implementations of a warehouse server for OLAP processing include the following:

Relational OLAP (ROLAP) servers: These are the intermediate servers that stand in between a relational back-end server and client front-end tools. They use a *relational or extended-relational DBMS* to store and manage warehouse data, and OLAP middleware to support missing pieces. ROLAP servers include optimization for each DBMS back end, implementation of aggregation navigation logic, and additional tools and services. ROLAP technology tends to have greater scalability than MOLAP technology. The DSS server of Microstrategy, for example, adopts the ROLAP approach.

Multidimensional OLAP (MOLAP) servers: These servers support multidimensional views of data through *array-based multidimensional storage engines.* They map multidimensional views directly to data cube array structures. The advantage of using a data cube is that it allows fast indexing to precomputed summarized data. Notice that with multidimensional data stores, the storage utilization may be low if the data set is sparse. In such cases, sparse matrix compression techniques should be explored (Chapter 4).

Many MOLAP servers adopt a two-level storage representation to handle dense and sparse data sets: denser subcubes are identified and stored as array structures, while sparse subcubes employ compression technology for efficient storage utilization.

Hybrid OLAP (HOLAP) servers: The hybrid OLAP approach combines ROLAP and MOLAP technology, benefiting from the greater scalability of ROLAP and the faster computation of MOLAP. For example, a HOLAP server may allow large volumes of detail data to be stored in a relational database, while aggregations are kept in a separate MOLAP store. The Microsoft SQL Server 2000 supports a hybrid OLAP server.

Specialized SQL servers: To meet the growing demand of OLAP processing in relational databases, some database system vendors implement specialized SQL servers that provide advanced query language and query processing support for SQL queries over star and snowflake schemas in a read-only environment.

"*How are data actually stored in ROLAP and MOLAP architectures?*" Let's first have a look at ROLAP. As its name implies, ROLAP uses relational tables to store data for on-line analytical processing. Recall that the fact table associated with a base cuboid is referred to as a *base fact table*. The base fact table stores data at the abstraction level indicated by the join keys in the schema for the given data cube. Aggregated data can also be stored in fact tables, referred to as **summary fact tables**. Some summary fact tables store both base fact table data and aggregated data, as in Example 3.10. Alternatively, separate summary fact tables can be used for each level of abstraction, to store only aggregated data.

Example 3.10 A ROLAP data store. Table 3.4 shows a summary fact table that contains both base fact data and aggregated data. The schema of the table is "⟨*record_identifier (RID), item, ..., day, month, quarter, year, dollars_sold*⟩", where *day, month, quarter*, and *year* define the date of sales, and *dollars_sold* is the sales amount. Consider the tuples with an *RID* of 1001 and 1002, respectively. The data of these tuples are at the base fact level, where the date of sales is October 15, 2003, and October 23, 2003, respectively. Consider the tuple with an *RID* of 5001. This tuple is at a more general level of abstraction than the tuples 1001 and 1002. The *day* value has been generalized to all, so that the corresponding *time* value is October 2003. That is, the *dollars_sold* amount shown is an aggregation representing the entire month of October 2003, rather than just October 15 or 23, 2003. The special value all is used to represent subtotals in summarized data.   ■

Table 3.4: Single table for base and summary facts.

RID	item	...	day	month	quarter	year	dollars_sold
1001	TV	...	15	10	Q4	2003	250.60
1002	TV	...	23	10	Q4	2003	175.00
...	...	...	...	...	...	...	...
5001	TV	...	all	10	Q4	2003	45,786.08
...	...	...	...	...	...	...	...

MOLAP uses multidimensional array structures to store data for on-line analytical processing. This structure is discussed in the following section on data warehouse implementation and, in greater detail, in Chapter 4.

Most data warehouse systems adopt a client-server architecture. A relational data store always resides at the data warehouse/data mart server site. A multidimensional data store can reside at either the database server site or the client site.

3.4   Data Warehouse Implementation

Data warehouses contain huge volumes of data. OLAP servers demand that decision support queries be answered in the order of seconds. Therefore, it is crucial for data warehouse systems to support highly efficient cube computation techniques, access methods, and query processing techniques. In this section, we present an overview of methods for the efficient implementation of data warehouse systems.

3.4.1 Efficient Computation of Data Cubes

At the core of multidimensional data analysis is the efficient computation of aggregations across many sets of dimensions. In SQL terms, these aggregations are referred to as group-by's. Each group-by can be represented by a *cuboid*, where the set of group-by's forms a lattice of cuboids defining a data cube. In this section, we explore issues relating to the efficient computation of data cubes.

The compute cube Operator and the Curse of Dimensionality

One approach to cube computation extends SQL so as to include a compute cube operator. The compute cube operator computes aggregates over all subsets of the dimensions specified in the operation. This can require excessive storage space, especially for large numbers of dimensions. We start with an intuitive look at what is involved in the efficient computation of data cubes.

Example 3.11 A data cube is a lattice of cuboids. Suppose that you would like to create a data cube for *AllElectronics* sales that contains the following: *city, item, year*, and *sales_in_dollars*. You would like to be able to analyze the data, with queries such as the following:

- "*Compute the sum of sales, grouping by city and item.*"

- "*Compute the sum of sales, grouping by city.*"

- "*Compute the sum of sales, grouping by item.*"

What is the total number of cuboids, or group-by's, that can be computed for this data cube? Taking the three attributes, *city, item*, and *year*, as the dimensions for the data cube, and *sales_in_dollars* as the measure, the total number of cuboids, or group-by's, that can be computed for this data cube is $2^3 = 8$. The possible group-by's are the following: $\{(city, item, year), (city, item), (city, year), (item, year), (city), (item), (year), ()\}$, where () means that the group-by is empty (i.e., the dimensions are not grouped). These group-by's form a lattice of cuboids for the data cube, as shown in Figure 3.14. The **base cuboid** contains all three dimensions, *city, item*, and *year*. It can return the total sales for any combination of the three dimensions. The **apex cuboid**, or 0-D cuboid, refers to the case where the group-by is empty. It contains the total sum of all sales. The base cuboid is the least generalized (most specific) of the cuboids. The apex cuboid is the most generalized (least specific) of the cuboids, and is often denoted as all. If we start at the apex cuboid and explore downwards in the lattice, this is equivalent to drilling down within the data cube. If we start at the base cuboid and explore upwards, this is akin to rolling up. ∎

An SQL query containing no group-by, such as "compute the sum of total sales," is a *zero-dimensional operation*. An SQL query containing one group-by, such as "compute the sum of sales, group by city," is a *one-dimensional operation*. A cube operator on n dimensions is equivalent to a collection of group by statements, one for each subset of the n dimensions. Therefore, the cube operator is the n-dimensional generalization of the group by operator.

Based on the syntax of DMQL introduced in Section 3.2.3, the data cube in Example 3.11 could be defined as

      define cube sales_cube [city, item, year]: sum(sales_in_dollars)

For a cube with n dimensions, there are a total of 2^n cuboids, including the base cuboid. A statement such as

      compute cube sales_cube

would explicitly instruct the system to compute the sales aggregate cuboids for all of the eight subsets of the set $\{city, item, year\}$, including the empty subset. A cube computation operator was first proposed and studied by Gray et al. [GCB+97].
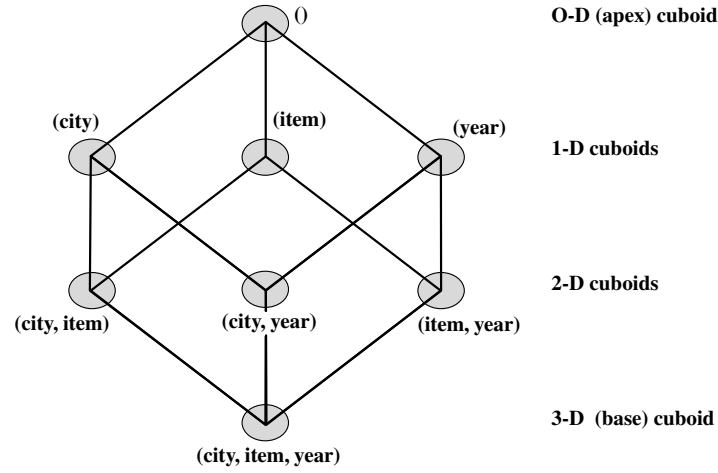
Figure 3.14: Lattice of cuboids, making up a 3-D data cube. Each cuboid represents a different group-by. The base cuboid contains the three dimensions *city, item*, and *year*. [TO EDITOR For consistency, please italicize *city, item*, and *year*.]

On-line analytical processing may need to access different cuboids for different queries. Therefore, it may seem like a good idea to compute all or at least some of the cuboids in a data cube in advance. Precomputation leads to fast response time and avoids some redundant computation. Most, if not all, OLAP products resort to some degree of precomputation of multidimensional aggregates.

A major challenge related to this precomputation, however, is that the required storage space may explode if all of the cuboids in a data cube are precomputed, especially when the cube has many dimensions. The storage requirements are even more excessive when many of the dimensions have associated concept hierarchies, each with multiple levels. This problem is referred to as the **curse of dimensionality**. The extent of the curse of dimensionality is illustrated below.

"*How many cuboids are there in an n-dimensional data cube?*" If there were no hierarchies associated with each dimension, then the total number of cuboids for an n-dimensional data cube, as we have seen above, is 2^n. However, in practice, many dimensions do have hierarchies. For example, the dimension *time* is usually not explored at only one conceptual level, such as *year*, but rather at multiple conceptual levels, such as in the hierarchy "*day < month < quarter < year*". For an n-dimensional data cube, the total number of cuboids that can be generated (including the cuboids generated by climbing up the hierarchies along each dimension) is

$$Total\ number\ of\ cuboids = \prod_{i=1}^{n}(L_i + 1), \tag{3.1}$$

where L_i is the number of levels associated with dimension i. One is added to L_i in Equation (3.1) to include the *virtual* top level, all. (Note that generalizing to all is equivalent to the removal of the dimension.) This formula is based on the fact that at most one abstraction level in each dimension will appear in a cuboid. For example, the time dimension as specified above has 4 conceptual levels, or 5 if we include the virtual level all. If the cube has 10 dimensions and each dimension has 5 levels (including all), the total number of cuboids that can be generated is $5^{10} \approx 9.8 \times 10^6$. The size of each cuboid also depends on the *cardinality* (i.e., number of distinct values) of each dimension. For example, if the *AllElectronics* branch in each city sold every item, there would be $|city| \times |item|$ tuples in the *city-item* group-by alone. As the number of dimensions, number of conceptual hierarchies, or cardinality increases, the storage space required for many of the group-by's will grossly exceed the (fixed) size of the input relation.

By now, you probably realize that it is unrealistic to precompute and materialize all of the cuboids that can possibly be generated for a data cube (or from a base cuboid). If there are many cuboids, and these cuboids are

large in size, a more reasonable option is *partial materialization*, that is, to materialize only *some* of the possible cuboids that can be generated.

Partial Materialization: Selected Computation of Cuboids

There are three choices for data cube materialization given a base cuboid:

1. **No materialization**: Do not precompute any of the "nonbase" cuboids. This leads to computing expensive multidimensional aggregates on the fly, which can be extremely slow.

2. **Full materialization**: Precompute all of the cuboids. The resulting lattice of computed cuboids is referred to as the *full cube*. This choice typically requires huge amounts of memory space in order to store all of the precomputed cuboids.

3. **Partial materialization**: Selectively compute a proper subset of the whole set of possible cuboids. Alternatively, we may compute a subset of the cube, which contains only those cells that satisfy some user-specified criterion, such as where the tuple count of each cell is above some threshold. We will use the term *subcube* to refer to the latter case, where only some of the cells may be precomputed for various cuboids. Partial materialization represents an interesting trade-off between storage space and response time.

The partial materialization of cuboids or subcubes should consider three factors: (1) identify the subset of cuboids or subcubes to materialize. (2) exploit the materialized cuboids or subcubes during query processing, and (3) efficiently update the materialized cuboids or subcubes during load and refresh.

The selection of the subset of cuboids or subcubes to materialize should take into account the queries in the workload, their frequencies, and their accessing costs. In addition, it should consider workload characteristics, the cost for incremental updates, and the total storage requirements. The selection must also consider the broad context of physical database design, such as the generation and selection of indices. Several OLAP products have adopted heuristic approaches for cuboid and subcube selection. A popular approach is to materialize the set of cuboids on which other frequently referenced cuboids are based. Alternatively, we can compute an *iceberg cube*, which is a data cube that stores only those cube cells whose aggregate value (e.g., count) is above some minimum support threshold. Another common strategy is to materialize a *shell cube*. This involves precomputing the cuboids for only a small number of dimensions (such as 3 to 5) of a data cube. Queries on additional combinations of the dimensions can be computed on the fly. Since our aim in this chapter is to provide a solid introduction and overview of data warehousing for data mining, we defer our detailed discussion of cuboid selection and computation to Chapter 4, which studies data warehouse and OLAP implementation in greater depth.

Once the selected cuboids have been materialized, it is important to take advantage of them during query processing. This involves several issues, such as how to determine the relevant cuboid(s) from among the candidate materialized cuboids, how to use available index structures on the materialized cuboids, and how to transform the OLAP operations onto the selected cuboid(s). These issues are discussed in Section 3.4.3 as well as in Chapter 4.

Finally, during load and refresh, the materialized cuboids should be updated efficiently. Parallelism and incremental update techniques for this should be explored.

3.4.2 Indexing OLAP Data

To facilitate efficient data accessing, most data warehouse systems support index structures and materialized views (using cuboids). General methods to select cuboids for materialization were discussed in the previous section. In this section, we examine how to index OLAP data by *bitmap indexing* and *join indexing*.

The **bitmap indexing** method is popular in OLAP products because it allows quick searching in data cubes. The bitmap index is an alternative representation of the *record_ID (RID)* list. In the bitmap index for a given

attribute, there is a distinct bit vector, Bv, for each value v in the domain of the attribute. If the domain of a given attribute consists of n values, then n bits are needed for each entry in the bitmap index (i.e., there are n bit vectors). If the attribute has the value v for a given row in the data table, then the bit representing that value is set to 1 in the corresponding row of the bitmap index. All other bits for that row are set to 0.

Example 3.12 Bitmap indexing. In the *AllElectronics* data warehouse, suppose the dimension *item* at the top level has four values (representing item types): *"home entertainment", "computer", "phone",* and *"security".* Each value (e.g., *"computer"*) is represented by a bit vector in the bitmap index table for *item.* Suppose that the cube is stored as a relation table with 100,000 rows. Since the domain of *item* consists of four values, the bitmap index table requires four bit vectors (or lists), each with 100,000 bits. Figure 3.15 shows a base (data) table containing the dimensions *item* and *city*, and its mapping to bitmap index tables for each of the dimensions. ∎

Base table				Item bitmap index table						City bitmap index table		
RID	**item**	**city**		**RID**	**H**	**C**	**P**	**S**		**RID**	**V**	**T**
R1	**H**	**V**		**R1**	1	0	0	0		**R1**	1	0
R2	**C**	**V**		**R2**	0	1	0	0		**R2**	1	0
R3	**P**	**V**		**R3**	0	0	1	0		**R3**	1	0
R4	**S**	**V**		**R4**	0	0	0	1		**R4**	1	0
R5	**H**	**T**		**R5**	1	0	0	0		**R5**	0	1
R6	**C**	**T**		**R6**	0	1	0	0		**R6**	0	1
R7	**P**	**T**		**R7**	0	0	1	0		**R7**	0	1
R8	**S**	**T**		**R8**	0	0	0	1		**R8**	0	1

Note: H for " home entertainment," C for " computer," P for " phone," S for " security,"
V for " Vancouver," T for " Toronto."

Figure 3.15: Indexing OLAP data using bitmap indices. [TO EDITOR For consistency with text, please italicize *item* and *city*.]

Bitmap indexing is advantageous compared to hash and tree indices. It is especially useful for low-cardinality domains because comparison, join, and aggregation operations are then reduced to bit arithmetic, which substantially reduces the processing time. Bitmap indexing leads to significant reductions in space and I/O since a string of characters can be represented by a single bit. For higher-cardinality domains, the method can be adapted using compression techniques.

The **join indexing** method gained popularity from its use in relational database query processing. Traditional indexing maps the value in a given column to a list of rows having that value. In contrast, join indexing registers the joinable rows of two relations from a relational database. For example, if two relations $R(RID, A)$ and $S(B, SID)$ join on the attributes A and B, then the join index record contains the pair (RID, SID), where RID and SID are record identifiers from the R and S relations, respectively. Hence, the join index records can identify joinable tuples without performing costly join operations. Join indexing is especially useful for maintaining the relationship between a foreign key[3] and its matching primary keys, from the joinable relation.

The star schema model of data warehouses makes join indexing attractive for cross table search because the linkage between a fact table and its corresponding dimension tables are the foreign key of the fact table and the primary key of the dimension table. Join indexing maintains relationships between attribute values of a dimension (e.g., within a dimension table) and the corresponding rows in the fact table. Join indices may span multiple dimensions to form **composite join indices**. We can use join indices to identify subcubes that are of interest.

Example 3.13 Join indexing. In Example 3.4, we defined a star schema for *AllElectronics* of the form *"sales_star [time, item, branch, location]: dollars_sold =* sum *(sales_in_dollars)".* An example of a join index relationship between the *sales* fact table and the dimension tables for *location* and *item* is shown in Figure 3.16. For example, the *"Main Street"* value in the *location* dimension table joins with tuples T57, T238, and T884 of the *sales* fact

---

[3]A set of attributes in a relation schema that forms a primary key for another relation schema is called a **foreign key**.
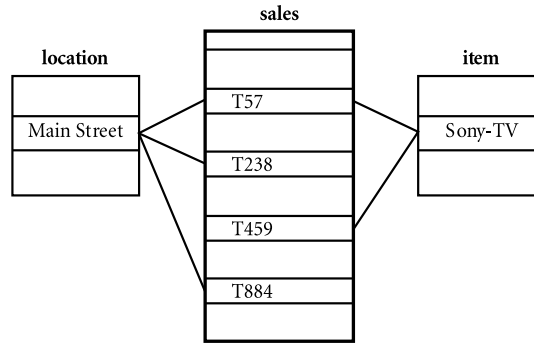
Figure 3.16: Linkages between a *sales* fact table and dimension tables for *location* and *item*. [TO EDITOR For consistency with text, please italicize *location*, *sales*, and *item*.]

table. Similarly, the "*Sony-TV*" value in the *item* dimension table joins with tuples T57 and T459 of the *sales* fact table. The corresponding join index tables are shown in Figure 3.17.

Join index table for location/sales

location	sales_key
...	...
Main Street	**T57**
Main Street	**T238**
Main Street	**T884**
...	...

Join index table for item/sales

item	sales_key
...	...
Sony-TV	**T57**
Sony-TV	**T459**
...	...

Join index table linking two dimensions location/item/sales

location	item	sales_key
...	...	...
Main Street	**Sony-TV**	**T57**
...	...	...

Figure 3.17: Join index tables based on the linkages between the *sales* fact table and dimension tables for *location* and *item* shown in Figure 3.16. [TO EDITOR For consistency with text, please italicize all instances of *location*, *sales*, *sales_key*, and *item*.]

Suppose that there are 360 time values, 100 items, 50 branches, 30 locations, and 10 million sales tuples in the *sales_star* data cube. If the *sales* fact table has recorded sales for only 30 items, the remaining 70 items will obviously not participate in joins. If join indices are not used, additional I/Os have to be performed to bring the joining portions of the fact table and dimension tables together. ∎

To further speed up query processing, the join indexing and bitmap indexing methods can be integrated to form **bitmapped join indices**.

3.4.3   Efficient Processing of OLAP Queries

The purpose of materializing cuboids and constructing OLAP index structures is to speed up query processing in data cubes. Given materialized views, query processing should proceed as follows:

1. **Determine which operations should be performed on the available cuboids:** This involves trans-

forming any selection, projection, roll-up (group-by), and drill-down operations specified in the query into corresponding SQL and/or OLAP operations. For example, slicing and dicing of a data cube may correspond to selection and/or projection operations on a materialized cuboid.

2. **Determine to which materialized cuboid(s) the relevant operations should be applied:** This involves identifying all of the materialized cuboids that may potentially be used to answer the query, pruning the above set using knowledge of "dominance" relationships among the cuboids, estimating the costs of using the remaining materialized cuboids, and selecting the cuboid with the least cost.

Example 3.14 OLAP query processing. Suppose that we define a data cube for *AllElectronics* of the form "*sales_cube [time, item, location]*: sum*(sales_in_dollars)*". The dimension hierarchies used are "*day < month < quarter < year*" for *time*, "*item_name < brand < type*" for *item*, and "*street < city < province_or_state < country*" for *location*.

Suppose that the query to be processed is on {*brand, province_or_state*}, with the selection constant "*year = 2004*". Also, suppose that there are four materialized cuboids available, as follows:

- cuboid 1: {*year, item_name, city*}

- cuboid 2: {*year, brand, country*}

- cuboid 3: {*year, brand, province_or_state*}

- cuboid 4: {*item_name, province_or_state*} where *year = 2004*

"*Which of the above four cuboids should be selected to process the query?*" Finer-granularity data cannot be generated from coarser-granularity data. Therefore, cuboid 2 cannot be used since *country* is a more general concept than *province_or_state*. Cuboids 1, 3, and 4 can be used to process the query since (1) they have the same set or a superset of the dimensions in the query, (2) the selection clause in the query can imply the selection in the cuboid, and (3) the abstraction levels for the *item* and *location* dimensions in these cuboids are at a finer level than *brand* and *province_or_state*, respectively.

"*How would the costs of each cuboid compare if used to process the query?*" It is likely that using cuboid 1 would cost the most since both *item_name* and *city* are at a lower level than the *brand* and *province_or_state* concepts specified in the query. If there are not many *year* values associated with *items* in the cube, but there are several *item_names* for each *brand*, then cuboid 3 will be smaller than cuboid 4, and thus cuboid 3 should be chosen to process the query. However, if efficient indices are available for cuboid 4, then cuboid 4 may be a better choice. Therefore, some cost-based estimation is required in order to decide which set of cuboids should be selected for query processing.   ∎

Since the storage model of a MOLAP server is an n-dimensional array, the front-end multidimensional queries are mapped directly to server storage structures, which provide direct addressing capabilities. The straightforward array representation of the data cube has good indexing properties, but has poor storage utilization when the data are sparse. For efficient storage and processing, sparse matrix and data compression techniques should therefore be applied. The details of several such methods of cube computation are presented in Chapter 4.

The storage structures used by dense and sparse arrays may differ, making it advantageous to adopt a two-level approach to MOLAP query processing: use array structures for dense arrays, and sparse matrix structures for sparse arrays. The two-dimensional dense arrays can be indexed by B-trees.

To process a query in MOLAP, the dense one- and two-dimensional arrays must first be identified. Indices are then built to these arrays using traditional indexing structures. The two-level approach increases storage utilization without sacrificing direct addressing capabilities.

"*Are there any other strategies for answering queries quickly?*" Some strategies for answering queries quickly concentrate on providing *intermediate feedback* to the users. For example, in **on-line aggregation**, a data mining system can display "what it knows so far" instead of waiting until the query is fully processed. Such an approximate

answer to the given data mining query is periodically refreshed and refined as the computation process continues. Confidence intervals are associated with each estimate, providing the user with additional feedback regarding the reliability of the answer so far. This promotes interactivity with the system—the user gains insight as to whether or not she is probing in the "right" direction without having to wait until the end of the query. While on-line aggregation does not improve the total time to answer a query, the overall data mining process should be quicker due to the increased interactivity with the system.

Another approach is to employ **top N queries**. Suppose that you are interested in finding only the best-selling items among the millions of items sold at *AllElectronics*. Rather than waiting to obtain a list of all store items, sorted in decreasing order of sales, you would like to see only the top N. Using statistics, query processing can be optimized to return the top N items, rather than the whole sorted list. This results in faster response time while helping to promote user interactivity and reduce wasted resources.

The goal of this section was to provide an overview of data warehouse implementation. Chapter 4 presents a more advanced treatment of this topic. It examines the efficient computation of data cubes and processing of OLAP queries in greater depth, providing detailed algorithms.

3.5 From Data Warehousing to Data Mining

"How do data warehousing and OLAP relate to data mining?" In this section, we study the usage of data warehousing for information processing, analytical processing, and data mining. We also introduce on-line analytical mining (OLAM), a powerful paradigm that integrates OLAP with data mining technology.

3.5.1 Data Warehouse Usage

Data warehouses and data marts are used in a wide range of applications. Business executives use the data in data warehouses and data marts to perform data analysis and make strategic decisions. In many firms, data warehouses are used as an integral part of a *plan-execute-assess* "closed-loop" feedback system for enterprise management. Data warehouses are used extensively in banking and financial services, consumer goods and retail distribution sectors, and controlled manufacturing, such as demand-based production.

Typically, the longer a data warehouse has been in use, the more it will have evolved. This evolution takes place throughout a number of phases. Initially, the data warehouse is mainly used for generating reports and answering predefined queries. Progressively, it is used to analyze summarized and detailed data, where the results are presented in the form of reports and charts. Later, the data warehouse is used for strategic purposes, performing multidimensional analysis and sophisticated slice-and-dice operations. Finally, the data warehouse may be employed for knowledge discovery and strategic decision making using data mining tools. In this context, the tools for data warehousing can be categorized into *access and retrieval tools*, *database reporting tools*, *data analysis tools*, and *data mining tools*.

Business users need to have the means to know what exists in the data warehouse (through metadata), how to access the contents of the data warehouse, how to examine the contents using analysis tools, and how to present the results of such analysis.

There are three kinds of data warehouse applications: *information processing, analytical processing*, and *data mining*:

- **Information processing** supports querying, basic statistical analysis, and reporting using crosstabs, tables, charts, or graphs. A current trend in data warehouse information processing is to construct low-cost Web-based accessing tools that are then integrated with Web browsers.

- **Analytical processing** supports basic OLAP operations, including slice-and-dice, drill-down, roll-up, and pivoting. It generally operates on historical data in both summarized and detailed forms. The major

strength of on-line analytical processing over information processing is the multidimensional data analysis of data warehouse data.

- **Data mining** supports knowledge discovery by finding hidden patterns and associations, constructing analytical models, performing classification and prediction, and presenting the mining results using visualization tools.

"How does data mining relate to information processing and on-line analytical processing?" Information processing, based on queries, can find useful information. However, answers to such queries reflect the information directly stored in databases or computable by aggregate functions. They do not reflect sophisticated patterns or regularities buried in the database. Therefore, information processing is not data mining.

On-line analytical processing comes a step closer to data mining since it can derive information summarized at multiple granularities from user-specified subsets of a data warehouse. Such descriptions are equivalent to the class/concept descriptions discussed in Chapter 1. Since data mining systems can also mine generalized class/concept descriptions, this raises some interesting questions: *"Do OLAP systems perform data mining? Are OLAP systems actually data mining systems?"*

The functionalities of OLAP and data mining can be viewed as disjoint: OLAP is a data summarization/aggregation *tool* that helps simplify data analysis, while data mining allows the *automated discovery* of implicit patterns and interesting knowledge hidden in large amounts of data. OLAP tools are targeted toward simplifying and supporting interactive data analysis, whereas the goal of data mining tools is to automate as much of the process as possible, while still allowing users to guide the process. In this sense, data mining goes one step beyond traditional on-line analytical processing.

An alternative and broader view of data mining may be adopted in which data mining covers both data description and data modeling. Since OLAP systems can present general descriptions of data from data warehouses, OLAP functions are essentially for user-directed data summary and comparison (by drilling, pivoting, slicing, dicing, and other operations). These are, though limited, data mining functionalities. Yet according to this view, data mining covers a much broader spectrum than simple OLAP operations because it not only performs data summary and comparison, but also performs association, classification, prediction, clustering, time-series analysis, and other data analysis tasks.

Data mining is not confined to the analysis of data stored in data warehouses. It may analyze data existing at more detailed granularities than the summarized data provided in a data warehouse. It may also analyze transactional, spatial, textual, and multimedia data that are difficult to model with current multidimensional database technology. In this context, data mining covers a broader spectrum than OLAP with respect to data mining functionality and the complexity of the data handled.

Since data mining involves more automated and deeper analysis than OLAP, data mining is expected to have broader applications. Data mining can help business managers find and reach more suitable customers, as well as gain critical business insights that may help to drive market share and raise profits. In addition, data mining can help managers understand customer group characteristics and develop optimal pricing strategies accordingly, correct item bundling based not on intuition but on actual item groups derived from customer purchase patterns, reduce promotional spending, and at the same time increase the overall net effectiveness of promotions.

3.5.2   From On-Line Analytical Processing to On-Line Analytical Mining

In the field of data mining, substantial research has been performed for data mining at various platforms, including transaction databases, relational databases, spatial databases, text databases, time-series databases, flat files, data warehouses, and so on.

Among many different paradigms and architectures of data mining systems, **on-line analytical mining (OLAM)** (also called **OLAP mining**), which integrates on-line analytical processing (OLAP) with data mining and mining knowledge in multidimensional databases, is particularly important for the following reasons:

- **High quality of data in data warehouses:** Most data mining tools need to work on integrated, consistent, and cleaned data, which requires costly data cleaning, data transformation, and data integration as preprocessing steps. A data warehouse constructed by such preprocessing serves as a valuable source of high-quality data for OLAP as well as for data mining. Notice that data mining may also serve as a valuable tool for data cleaning and data integration as well.

- **Available information processing infrastructure surrounding data warehouses:** Comprehensive information processing and data analysis infrastructures have been or will be systematically constructed surrounding data warehouses, which include accessing, integration, consolidation, and transformation of multiple heterogeneous databases, ODBC/OLE DB connections, Web-accessing and service facilities, and reporting and OLAP analysis tools. It is prudent to make the best use of the available infrastructures rather than constructing everything from scratch.

- **OLAP-based exploratory data analysis:** Effective data mining needs exploratory data analysis. A user will often want to traverse through a database, select portions of relevant data, analyze them at different granularities, and present knowledge/results in different forms. On-line analytical mining provides facilities for data mining on different subsets of data and at different levels of abstraction, by drilling, pivoting, filtering, dicing and slicing on a data cube and on some intermediate data mining results. This, together with data/knowledge visualization tools, will greatly enhance the power and flexibility of exploratory data mining.

- **On-line selection of data mining functions:** Often a user may not know what kinds of knowledge she would like to mine. By integrating OLAP with multiple data mining functions, on-line analytical mining provides users with the flexibility to select desired data mining functions and swap data mining tasks dynamically.

Architecture for On-Line Analytical Mining

An OLAM server performs analytical mining in data cubes in a similar manner as an OLAP server performs on-line analytical processing. An integrated OLAM and OLAP architecture is shown in Figure 3.18, where the OLAM and OLAP servers both accept user on-line queries (or commands) via a graphical user interface API and work with the data cube in the data analysis via a cube API. A metadata directory is used to guide the access of the data cube. The data cube can be constructed by accessing and/or integrating multiple databases via an MDDB API and/or by filtering a data warehouse via a database API that may support OLE DB or ODBC connections. Since an OLAM server may perform multiple data mining tasks, such as concept description, association, classification, prediction, clustering, time-series analysis, and so on, it usually consists of multiple integrated data mining modules and is more sophisticated than an OLAP server.

Chapter 4 describes data warehouses on a finer level by exploring implementation issues such as data cube computation, OLAP query answering strategies, and methods of generalization. The chapters following it are devoted to the study of data mining techniques. As we have seen, the introduction to data warehousing and OLAP technology presented in this chapter is essential to our study of data mining. This is because data warehousing provides users with large amounts of clean, organized, and summarized data, which greatly facilitates data mining. For example, rather than storing the details of each sales transaction, a data warehouse may store a summary of the transactions per item type for each branch or, summarized to a higher level, for each country. The capability of OLAP to provide multiple and dynamic views of summarized data in a data warehouse sets a solid foundation for successful data mining.

Moreover, we also believe that data mining should be a human-centered process. Rather than asking a data mining system to generate patterns and knowledge automatically, a user will often need to interact with the system to perform exploratory data analysis. OLAP sets a good example for interactive data analysis and provides the necessary preparations for exploratory data mining. Consider the discovery of association patterns, for example. Instead of mining associations at a primitive (i.e., low) data level among transactions, users should be allowed to specify roll-up operations along any dimension. For example, a user may like to roll up on the *item* dimension to go from viewing the data for particular TV sets that were purchased to viewing the brands of these TVs, such as
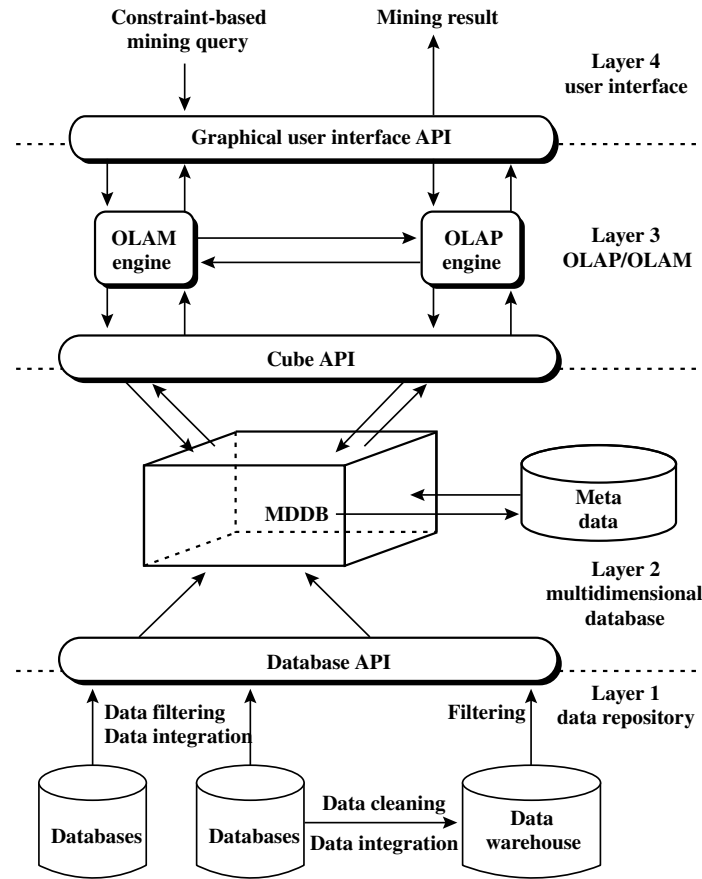
Figure 3.18: An integrated OLAM and OLAP architecture.

SONY or Panasonic. Users may also navigate from the transaction level to the customer level or customer-type level in the search for interesting associations. Such an OLAP-style of data mining is characteristic of OLAP mining. In our study of the principles of data mining in this book, we place particular emphasis on OLAP mining, that is, on the *integration of data mining and OLAP technology*.

3.6   Summary

- A **data warehouse** is a *subject-oriented, integrated, time-variant*, and *nonvolatile* collection of data organized in support of management decision making. Several factors distinguish data warehouses from operational databases. Since the two systems provide quite different functionalities and require different kinds of data, it is necessary to maintain data warehouses separately from operational databases.

- A **multidimensional data model** is typically used for the design of corporate *data warehouses* and *departmental data marts*. Such a model can adopt a *star schema, snowflake schema*, or *fact constellation schema*. The core of the *multidimensional model* is the **data cube**, which consists of a large set of *facts* (or *measures*) and a number of *dimensions*. Dimensions are the entities or perspectives with respect to which an organization wants to keep records and are hierarchical in nature.

- A data cube consists of a **lattice of cuboids**, each corresponding to a different degree of summarization of the given multidimensional data.

- **Concept hierarchies** organize the values of attributes or dimensions into gradual levels of abstraction. They are useful in mining at multiple levels of abstraction.

- **On-line analytical processing (OLAP)** can be performed in data warehouses/marts using the multidimensional data model. Typical OLAP operations include *roll-up*, *drill-(down, across, through)*, *slice-and-dice*, *pivot* (*rotate*), as well as statistical operations such as ranking, computing moving averages and growth rates, and so on. OLAP operations can be implemented efficiently using the data cube structure.

- Data warehouses often adopt a **three-tier architecture**. The bottom tier is a *warehouse database server*, which is typically a relational database system. The middle tier is an *OLAP server*, and the top tier is a *client*, containing query and reporting tools.

- A data warehouse contains **back-end tools and utilities** for populating and refreshing the warehouse. These cover data extraction, data cleaning, data transformation, loading, refreshing, and warehouse management.

- Data warehouse **metadata** are data defining the warehouse objects. A metadata repository provides details regarding the warehouse structure, data history, the algorithms used for summarization, mappings from the source data to warehouse form, system performance, and business terms and issues.

- OLAP servers may use **relational OLAP (ROLAP)**, or **multidimensional OLAP (MOLAP)**, or **hybrid OLAP (HOLAP)**. A ROLAP server uses an extended relational DBMS that maps OLAP operations on multidimensional data to standard relational operations. A MOLAP server maps multidimensional data views directly to array structures. A HOLAP server combines ROLAP and MOLAP. For example, it may use ROLAP for historical data while maintaining frequently accessed data in a separate MOLAP store.

- **Full materialization** refers to the computation of all of the cuboids in the lattice defining a data cube. It typically requires an excessive amount of storage space, particularly as the number of dimensions and size of associated concept hierarchies grow. This problem is known as the **curse of dimensionality**. Alternatively, **partial materialization** is the selective computation of a subset of the cuboids or subcubes in the lattice. For example, an **iceberg cube** is a data cube that stores only those cube cells whose aggregate value (e.g., count) is above some minimum support threshold.

- OLAP query processing can be made more efficient with the use of indexing techniques. In **bitmap indexing**, each attribute has its own bitmap index table. Bitmap indexing reduces join, aggregation, and comparison operations to bit arithmetic. **Join indexing** registers the joinable rows of two or more relations from a relational database, reducing the overall cost of OLAP join operations. **Bitmapped join indexing**, which combines the bitmap and join index methods, can be used to further speed up OLAP query processing.

- Data warehouses are used for *information processing* (querying and reporting), *analytical processing* (which allows users to navigate through summarized and detailed data by OLAP operations), and *data mining* (which supports knowledge discovery). OLAP-based data mining is referred to as **OLAP mining**, or on-line analytical mining (**OLAM**), which emphasizes the interactive and exploratory nature of OLAP mining.

3.7 Exercises

1. State why, for the integration of multiple heterogeneous information sources, many companies in industry prefer the *update-driven approach* (which constructs and uses data warehouses), rather than the *query-driven approach* (which applies wrappers and integrators). Describe situations where the query-driven approach is preferable over the update-driven approach.

2. Briefly compare the following concepts. You may use an example to explain your point(s).

   (a) Snowflake schema, fact constellation, starnet query model
   (b) Data cleaning, data transformation, refresh
   (c) Discovery-driven cube, multifeature cube, virtual warehouse

3. Suppose that a data warehouse consists of the three dimensions *time, doctor*, and *patient*, and the two measures *count* and *charge*, where *charge* is the fee that a doctor charges a patient for a visit.

   (a) Enumerate three classes of schemas that are popularly used for modeling data warehouses.

   (b) Draw a schema diagram for the above data warehouse using one of the schema classes listed in (a).

   (c) Starting with the base cuboid [*day, doctor, patient*], what specific *OLAP operations* should be performed in order to list the total fee collected by each doctor in 2004?

   (d) To obtain the same list, write an SQL query assuming the data is stored in a relational database with the schema *fee* (*day, month, year, doctor, hospital, patient, count, charge*).

4. Suppose that a data warehouse for *Big-University* consists of the following four dimensions: *student, course, semester*, and *instructor*, and two measures *count* and *avg_grade*. When at the lowest conceptual level (e.g., for a given student, course, semester, and instructor combination), the *avg_grade* measure stores the actual course grade of the student. At higher conceptual levels, *avg_grade* stores the average grade for the given combination.

   (a) Draw a *snowflake schema* diagram for the data warehouse.

   (b) Starting with the base cuboid [*student, course, semester, instructor*], what specific *OLAP operations* (e.g., roll-up from *semester* to *year*) should one perform in order to list the average grade of *CS* courses for each *Big-University* student.

   (c) If each dimension has five levels (including all), such as "*student* < *major* < *status* < *university* < all", how many cuboids will this cube contain (including the base and apex cuboids)?

5. Suppose that a data warehouse consists of the four dimensions, *date, spectator, location*, and *game*, and the two measures, *count* and *charge*, where *charge* is the fare that a spectator pays when watching a game on a given date. Spectators may be students, adults, or seniors, with each category having its own charge rate.

   (a) Draw a *star schema* diagram for the data warehouse.

   (b) Starting with the base cuboid [*date, spectator, location, game*], what specific *OLAP operations* should one perform in order to list the total charge paid by student spectators at GM_Place in 2004?

   (c) *Bitmap indexing* is useful in data warehousing. Taking this cube as an example, briefly discuss advantages and problems of using a bitmap index structure.

6. A data warehouse can be modeled by either a *star schema* or a *snowflake schema*. Briefly describe the similarities and the differences of the two models, and then analyze their advantages and disadvantages with regard to one another. Give your opinion of which might be more empirically useful and state the reasons behind your answer.

7. Design a data warehouse for a regional weather bureau. The weather bureau has about 1,000 probes, which are scattered throughout various land and ocean locations in the region to collect basic weather data, including air pressure, temperature, and precipitation at each hour. All data are sent to the central station, which has collected such data for over 10 years. Your design should facilitate efficient querying and on-line analytical processing, and derive general weather patterns in multidimensional space.

8. A popular data warehouse implementation is to construct a multidimensional database, known as a data cube. Unfortunately, this may often generate a huge, yet very sparse multidimensional matrix.

   (a) Present an example illustrating such a huge and sparse data cube.

   (b) Design an implementation method that can elegantly overcome this sparse matrix problem. Note that you need to explain your data structures in detail and discuss the space needed, as well as how to retrieve data from your structures.

   (c) Modify your design in (b) to handle *incremental data updates.* Give the reasoning behind your new design.

9. Regarding the *computation of measures* in a data cube:

    (a) Enumerate three categories of measures, based on the kind of aggregate functions used in computing a data cube.

    (b) For a data cube with the three dimensions *time, location*, and *item*, which category does the function *variance* belong to? Describe how to compute it if the cube is partitioned into many chunks.

    Hint: The formula for computing *variance* is $\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x}_i)^2$, where $\bar{x}_i$ is the average of x_is.

    (c) Suppose the function is "*top 10 sales*." Discuss how to efficiently compute this measure in a data cube.

10. Suppose that we need to record three measures in a data cube: min, average, and median. Design an efficient computation and storage method for each measure given that the cube allows data to be *deleted incrementally* (i.e., in small portions at a time) from the cube.

11. In data warehouse technology, a multiple dimensional view can be implemented by a relational database technique (*ROLAP*), or by a multidimensional database technique (*MOLAP*), or by a hybrid database technique (*HOLAP*).

    (a) Briefly describe each implementation technique.

    (b) For each technique, explain how each of the following functions may be implemented:

        i. The generation of a data warehouse (including aggregation)
        ii. Roll-up
        iii. Drill-down
        iv. Incremental updating

    Which implementation techniques do you prefer, and why?

12. Suppose that a data warehouse contains 20 dimensions, each with about five levels of granularity.

    (a) Users are mainly interested in four particular dimensions, each having three frequently accessed levels for rolling up and drilling down. How would you design a data cube structure to support this preference efficiently?

    (b) At times, a user may want to *drill through* the cube, down to the raw data for one or two particular dimensions. How would you support this feature?

13. A data cube, C, has n dimensions, and each dimension has exactly p distinct values in the base cuboid. Assume that there are no concept hierarchies associated with the dimensions.

    (a) What is the *maximum number of cells* possible in the base cuboid?

    (b) What is the *minimum number of cells* possible in the base cuboid?

    (c) What is the *maximum number of cells* possible (including both base cells and aggregate cells) in the data cube, C?

    (d) What is the *minimum number of cells* possible in the data cube, C?

14. What are the differences between the three main types of data warehouse usage: *information processing, analytical processing*, and *data mining?* Discuss the motivation behind *OLAP mining (OLAM)*.

3.8 Bibliographic Notes

There are a good number of introductory-level textbooks on data warehousing and OLAP technology, including Kimball and Ross [KR02], Imhoff, Galemmo and Geiger [IGG03], Inmon [Inm96], Berson and Smith [BS97], and Thomsen [Tho97]. Chaudhuri and Dayal [CD97] provide a general overview of data warehousing and OLAP technology. A set of research papers on materialized views and data warehouse implementations were collected in *Materialized Views: Techniques, Implementations, and Applications* by Gupta and Mumick [GM99].

The history of decision support systems can be traced back to the 1960s. However, the proposal of the construction of large data warehouses for multidimensional data analysis is credited to Codd [CCS93] who coined the term *OLAP* for *on-line analytical processing*. The OLAP council was established in 1995. Widom [Wid95] identified several research problems in data warehousing. Kimball and Ross [KR02] provide an overview of the deficiencies of SQL regarding the ability to support comparisons that are common in the business world and present a good set of application cases that require data warehousing and OLAP technology. For an overview of OLAP systems versus statistical databases, see Shoshani [Sho97].

Gray et al. [GCB$^+$97] proposed the data cube as a relational aggregation operator generalizing group-by, crosstabs, and subtotals. Harinarayan, Rajaraman, and Ullman [HRU96] proposed a greedy algorithm for the partial materialization of cuboids in the computation of a data cube. Sarawagi and Stonebraker [SS94] developed a chunk-based computation technique for the efficient organization of large multidimensional arrays. Agarwal et al. [AAD$^+$96] proposed several methods for the efficient computation of multidimensional aggregates for ROLAP servers. A chunk-based multiway array aggregation method for data cube computation in MOLAP was proposed in Zhao, Deshpande, and Naughton [ZDN97]. Ross and Srivastava [RS97] pointed out the problem of the curse of dimensionality in cube materialization and developed a method for computing sparse data cubes. Iceberg queries are first described in Fang, Shivakumar, Garcia-Molina, et al. [FSGM$^+$98]. BUC, an efficient bottom-up method for computing iceberg cubes was introduced by Beyer and Ramakrishnan [BR99]. References for the further development of cube computation methods are given in the Bibliographic Notes of Chapter 4. The use of join indices to speed up relational query processing was proposed by Valduriez [Val87]. O'Neil and Graefe [OG95] proposed a bitmapped join index method to speed up OLAP-based query processing. A discussion of the performance of bitmapping and other nontraditional index techniques is given in O'Neil and Quass [OQ97].

For work regarding the selection of materialized cuboids for efficient OLAP query processing, see Chaudhuri and Dayal [CD97], Harinarayan, Rajaraman, and Ullman [HRU96], Sristava et al. [SDJL96]. Methods for cube size estimation can be found in Deshpande et al. [DNR$^+$97], Ross and Srivastava [RS97], and Beyer and Ramakrishnan [BR99]. Agrawal, Gupta, and Sarawagi [AGS97] proposed operations for modeling multidimensional databases. Methods for answering queries quickly by on-line aggregation are described in Hellerstein, Haas, and Wang [HHW97] and Hellerstein et al. [HAC$^+$99]. Techniques for estimating the top N queries are proposed in Carey and Kossman [CK98] and Donjerkovic and Ramakrishnan [DR99]. Further studies on intelligent OLAP and discovery-driven exploration of data cubes are presented in the Bibliographic Notes of Chapter 4.

Bibliography

[AAD⁺96] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96)*, pages 506–521, Bombay, India, Sept. 1996.

[AGS97] R. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. In *Proc. 1997 Int. Conf. Data Engineering (ICDE'97)*, pages 232–243, Birmingham, England, April 1997.

[BR99] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. 1999 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'99)*, pages 359–370, Philadelphia, PA, June 1999.

[BS97] A. Berson and S. J. Smith. *Data Warehousing, Data Mining, and OLAP*. New York: McGraw-Hill, 1997.

[CCS93] E. F. Codd, S. B. Codd, and C. T. Salley. Beyond decision support. *Computer World*, 27, July 1993.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26:65–74, 1997.

[CK98] M. Carey and D. Kossman. Reducing the braking distance of an SQL query engine. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 158–169, New York, NY, Aug. 1998.

[DNR⁺97] P. Deshpande, J. Naughton, K. Ramasamy, A. Shukla, K. Tufte, and Y. Zhao. Cubing algorithms, storage estimation, and storage and processing alternatives for OLAP. *Data Engineering Bulletin*, 20:3–11, 1997.

[DR99] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top N queries. In *Proc. 1999 Int. Conf. Very Large Data Bases (VLDB'99)*, pages 411–422, Edinburgh, UK, Sept. 1999.

[FSGM⁺98] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 299–310, New York, NY, Aug. 1998.

[GCB⁺97] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-totals. *Data Mining and Knowledge Discovery*, 1:29–54, 1997.

[GM99] A. Gupta and I. S. Mumick. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999.

[HAC⁺99] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive data analysis: The control project. *IEEE Computer*, 32:51–59, July 1999.

[HHW97] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'97)*, pages 171–182, Tucson, Arizona, May 1997.

[HRU96]   V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'96)*, pages 205–216, Montreal, Canada, June 1996.

[IGG03]   C. Imhoff, N. Galemmo, and J. G. Geiger. *Mastering Data Warehouse Design : Relational and Dimensional Techniques.* John Wiley & Sons, New York, 2003.

[Inm96]   W. H. Inmon. *Building the Data Warehouse.* John Wiley & Sons, 1996.

[KR02]   R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling (2ed.).* John Wiley & Sons, New York, 2002.

[OG95]   P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24:8–11, Sept. 1995.

[OQ97]   P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'97)*, pages 38–49, Tucson, Arizona, May 1997.

[RS97]   K. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proc. 1997 Int. Conf. Very Large Data Bases (VLDB'97)*, pages 116–125, Athens, Greece, Aug. 1997.

[SDJL96] D. Sristava, S. Dar, H. V. Jagadish, and A. V. Levy. Answering queries with aggregation using views. In *Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96)*, pages 318–329, Bombay, India, Sept. 1996.

[Sho97]   A. Shoshani. OLAP and statistical databases: Similarities and differences. In *Proc. 16th ACM Symp. Principles of Database Systems*, pages 185–196, Tucson, Arizona, May 1997.

[SS94]   S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proc. 1994 Int. Conf. Data Engineering (ICDE'94)*, pages 328–336, Houston, TX, Feb. 1994.

[Tho97]   E. Thomsen. *OLAP Solutions: Building Multidimensional Information Systems.* John Wiley & Sons, 1997.

[Val87]   P. Valduriez. Join indices. *ACM Trans. Database Systems*, 12:218–246, 1987.

[Wid95]   J. Widom. Research problems in data warehousing. In *Proc. 4th Int. Conf. Information and Knowledge Management*, pages 25–30, Baltimore, Maryland, Nov. 1995.

[ZDN97]   Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'97)*, pages 159–170, Tucson, Arizona, May 1997.