# Contents

# Chapter 2

# Data Warehouse and OLAP Technology for Data Mining

The construction of data warehouses, which involves data cleaning 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 mining. Furthermore, 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, 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, prior to presenting a systematic coverage of data mining technology in the remainder of this book, we devote this chapter to an overview of data warehouse technology. Such an overview is essential for understanding data mining technology.

In this chapter, you will learn the basic concepts, general architectures, and major implementation techniques employed in data warehouse and OLAP technology, as well as their relationship with data mining.

## 2.1   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. A large number of organizations have found that 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.

*"So"*, you may ask, full of intrigue, *"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 W. 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." (Inmon 1992). 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, vendor, 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.

*"OK"*, you now ask, *"what, then, is data warehousing?"*

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 integration, data cleaning, 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 are using 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. A variety of data joiner and data blade products belong to this category. 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 very popular in industry.

**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.

1. **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.

2. **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 for use in informed decision making.

3. **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 2.2.2), and a subject-oriented database design.

4. **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.

5. **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 which distinguish between OLTP and OLAP systems include database size, frequency of operations, and performance metrics. These are summarized in Table 2.1.

**But, why have a separate data warehouse?**

*"Since operational databases store huge amounts of data"*, you observe, *"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 several 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.

| 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 | E-R 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 |
| # of records accessed | tens | millions |
| # 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 |

Table 2.1: Comparison between OLTP and OLAP systems.

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, cleansed 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 necessary to maintain separate databases.

## 2.2    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.

### 2.2.1    From tables 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 later look at multidimensional schemas.

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 which is, in fact, a table 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 2.2. 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*.

| **time** (quarter) | Sales for all **locations** in Vancouver | | | |
| | **item** (type) | | | |
| | home entertainment | computer | phone | security |
| Q1 | 605K | 825K | 14K | 400K |
| Q2 | 680K | 952K | 31K | 512K |
| Q3 | 812K | 1023K | 30K | 501K |
| Q4 | 927K | 1038K | 38K | 580K |

Table 2.2: 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*.

| t i m e | location = "Vancouver" | | | | location = "Montreal" | | | | location = "New York" | | | | location = "Chicago" | | | |
| | item | | | | item | | | | item | | | | item | | | |
| | home ent. | comp. | phone | sec. | home ent. | comp. | phone | sec. | home ent. | comp. | phone | sec. | home ent. | comp. | phone | sec. |
| Q1 | 605K | 825K | 14K | 400K | 818K | 746K | 43K | 591K | 1087K | 968K | 38K | 872K | 854K | 882K | 89K | 623K |
| Q2 | 680K | 952K | 31K | 512K | 894K | 769K | 52K | 682K | 1130K | 1024K | 41K | 925K | 943K | 890K | 64K | 698K |
| Q3 | 812K | 1023K | 30K | 501K | 940K | 795K | 58K | 728K | 1034K | 1048K | 45K | 1002K | 1032K | 924K | 59K | 789K |
| Q4 | 927K | 1038K | 38K | 580K | 978K | 864K | 59K | 784K | 1142K | 1091K | 54K | 984K | 1129K | 992K | 63K | 870K |

Table 2.3: A 3-D view of sales data for *AllElectronics*, according to the dimensions *time*, *item*, and *location*. The measure displayed is *dollars_sold*.

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*. These 3-D data are shown in Table 2.3. The 3-D data of Table 2.3 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 2.1.

Suppose that we would now like to view our sales data with an additional fourth dimension, such as *supplier*. 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 2.2. 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 referred to as a **cuboid**. Given a set of dimensions, we can construct a *lattice* of cuboids, each showing the data at a different level of summarization, or `group by` (i.e., summarized by a different subset of the dimensions). The lattice of cuboids is then referred to as a **data cube**. Figure 2.8 shows a lattice of cuboids forming a data cube for the dimensions *time*, *item*, *location*, and *supplier*.
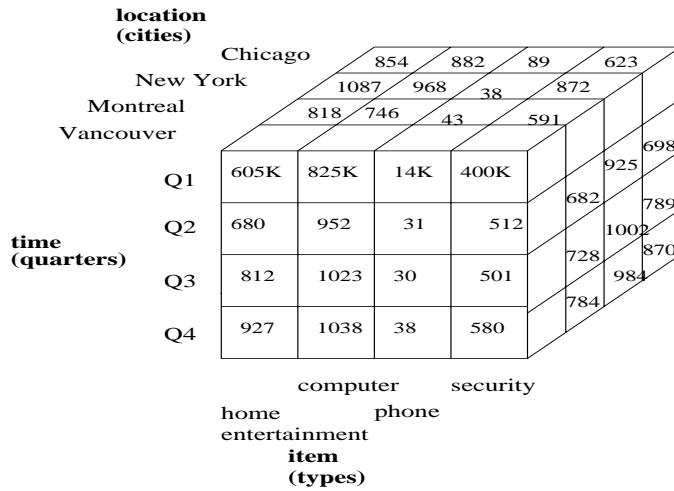
Figure 2.1: A 3-D data cube representation of the data in Table 2.3, according to the dimensions *time*, *item*, and *location*. The measure displayed is *dollars_sold*.
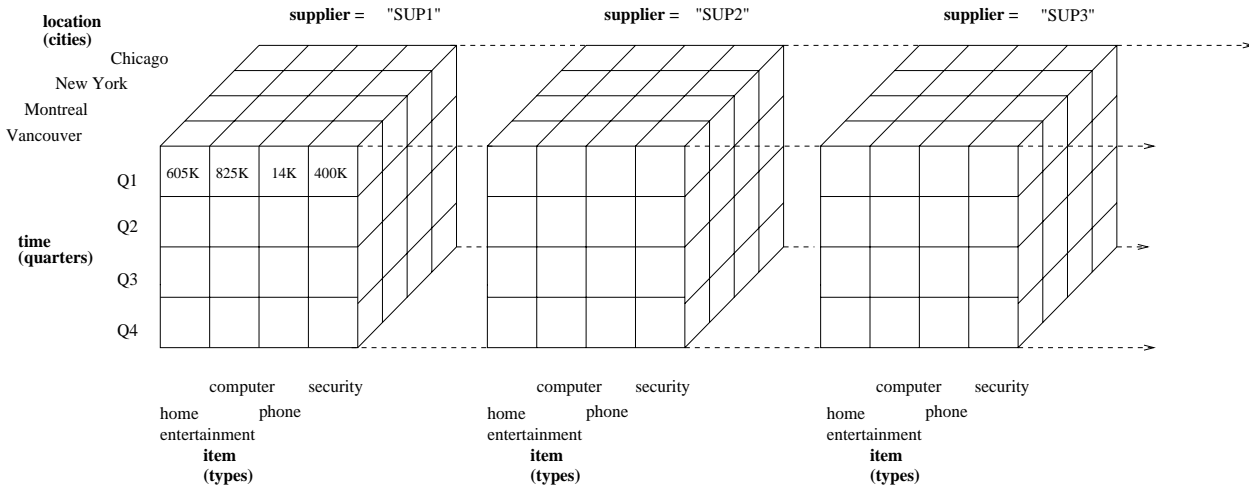


Figure 2.2: A 4-D data cube representation of sales data, according to the dimensions *time*, *item*, *location*, and *supplier*. The measure displayed is *dollars_sold*.

The cuboid which holds the lowest level of summarization is called the **base cuboid**. For example, the 4-D cuboid in Figure 2.2 is the base cuboid for the given *time*, *item*, *location*, and *supplier* dimensions. Figure 2.1 is a 3-D (non-base) 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 for all four dimensions. The apex cuboid is typically denoted by all.

## 2.2.2   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 or objects, and the relationships between them. Such a data model is appropriate for on-line transaction processing. Data warehouses, however, require a concise, subject-oriented schema which facilitates on-line data analysis.

The most popular data model for data warehouses is a **multidimensional model**. This 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.
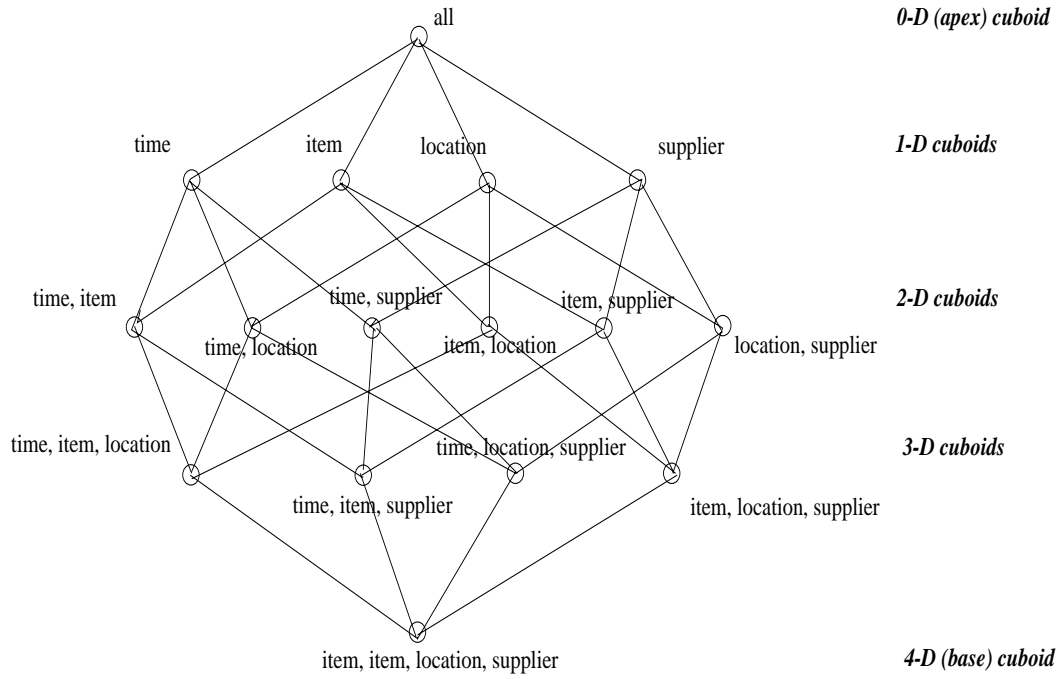
Figure 2.3: Lattice of cuboids, making up a 4-D data cube for the dimensions *time*, *item*, *location*, and *supplier*. Each cuboid represents a different degree of summarization.

- **Star schema**: The star schema is a modeling paradigm in which the data warehouse contains (1) a large central table (**fact table**), 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.
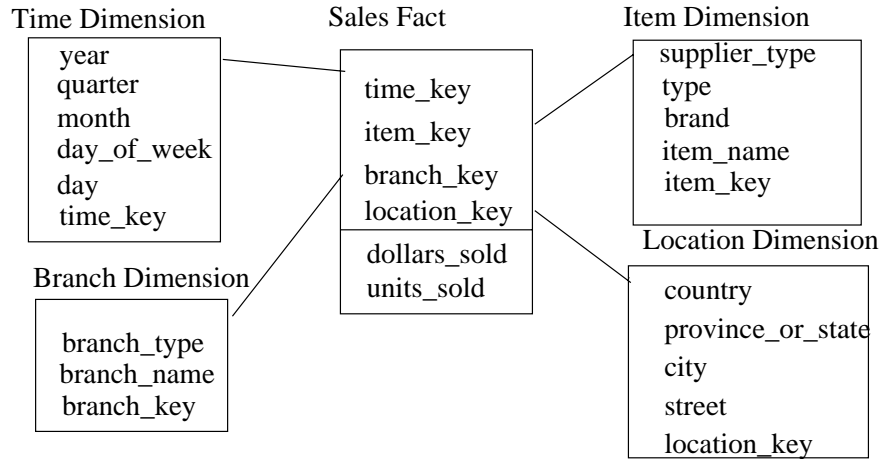


Figure 2.4: Star schema of a data warehouse for sales.

**Example 2.1** An example of a star schema for *AllElectronics* sales is shown in Figure 2.4. Sales are considered along four dimensions, namely *time, item, branch*, and *location*. The schema contains a central fact table for *sales* which contains keys to each of the four dimensions, along with two measures: *dollars_sold* and *units_sold*. □

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*, i.e., (.., 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.

  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. Such a table is easy to maintain and also saves storage space because a large dimension table can be extremely large when the dimensional structure is included as columns. Since much of this space is redundant data, creating a normalized structure will reduce the overall space requirement. However, 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. Performance benchmarking can be used to determine what is best for your design.
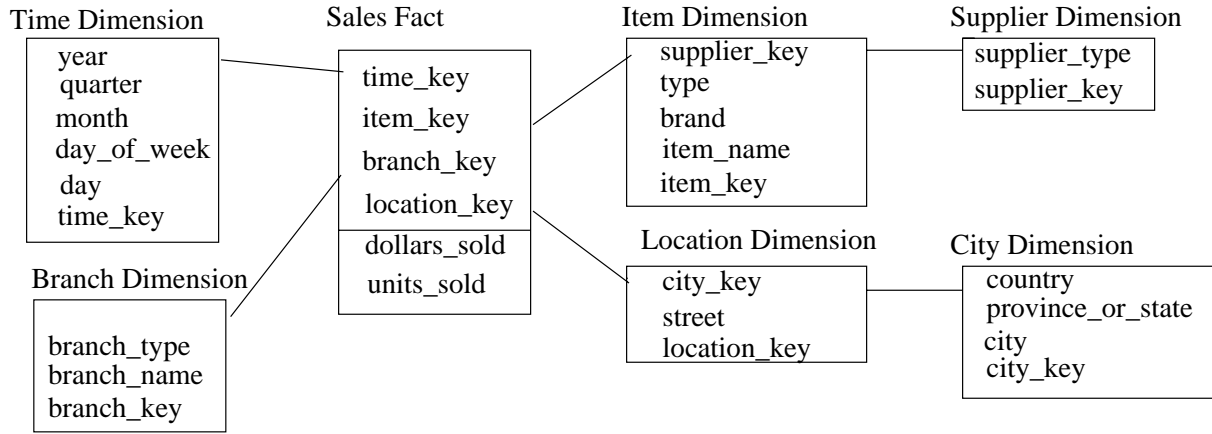


Figure 2.5: Snowflake schema of a data warehouse for sales.

**Example 2.2** An example of a snowflake schema for *AllElectronics* sales is given in Figure 2.5. Here, the *sales* fact table is identical to that of the star schema in Figure 2.4. 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 *supplier_key, type, brand, item_name*, and *item_key*, the latter of which is linked to the *supplier* dimension table, containing *supplier_type* and *supplier_key* information. Similarly, the single dimension table for *location* in the star schema can be normalized into two tables: new *location* and *city*. The *location_key* of the new *location* table now 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 2.5, when desirable. □

A compromise between the star schema and the snowflake schema is to adopt a **mixed schema** where only the very large dimension tables are normalized. Normalizing large dimension tables saves storage space, while keeping small dimension tables unnormalized may reduce the cost and performance degradation due to joins on multiple dimension tables. Doing both may lead to an overall performance gain. However, careful performance tuning could be required to determine which dimension tables should be normalized and split into multiple tables.

- **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**.
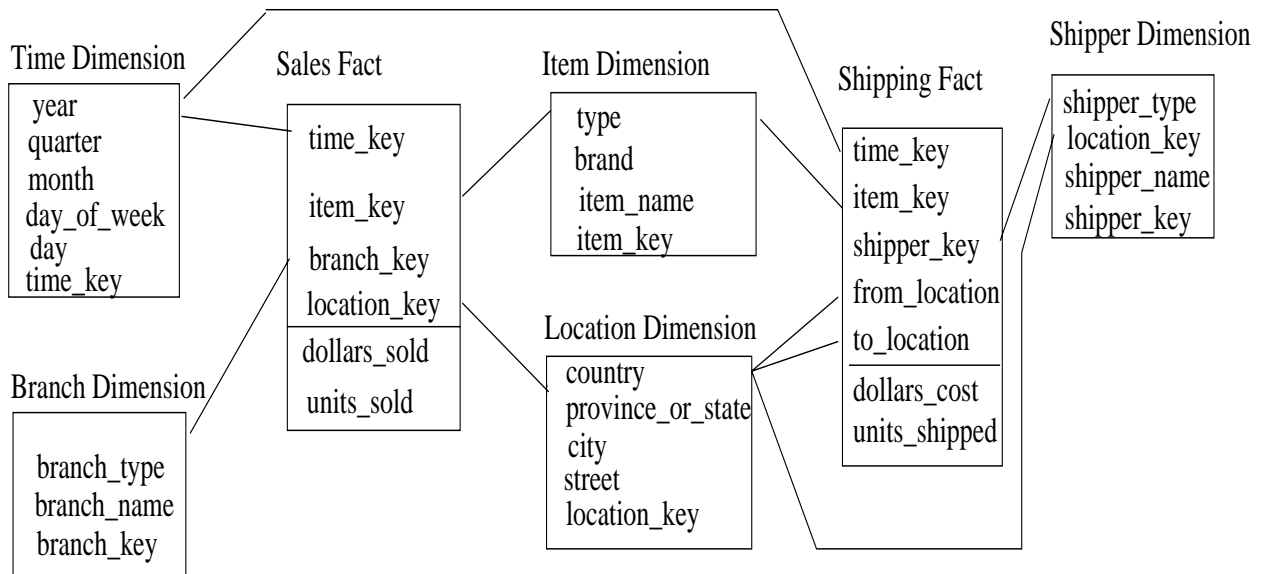
Figure 2.6: Fact constellation schema of a data warehouse for sales and shipping.

**Example 2.3** An example of a fact constellation schema is shown in Figure 2.6. This schema specifies two fact tables, *sales* and *shipping*. The *sales* table definition is identical to that of the star schema (Figure 2.4). The *shipping* table has five dimensions, or keys: *time_key, item_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 *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 popular since each are geared towards modeling single subjects.

### 2.2.3    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 an SQL-based data mining query language called **DMQL** which contains language primitives for defining data warehouses and data marts. Language primitives for specifying other data mining tasks, such as the mining of concept/class descriptions, associations, classifications, and so on, will be introduced in Chapter 4.

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_subdimension_list⟩)

Let's look at examples of how to define the star, snowflake and constellations schemas of Examples 2.1 to 2.3 using DMQL. DMQL keywords are displayed in sans serif font.

**Example 2.4** The star schema of Example 2.1 and Figure 2.4 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)

The **define cube** statement defines a data cube called *sales_star*, which corresponds to the central *sales* fact table of Example 2.1. This command specifies the keys to the dimension tables, 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 2.5** The snowflake schema of Example 2.2 and Figure 2.5 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 2.4), except that, here, the *item* and *location* dimensions 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 2.6** The fact constellation schema of Example 2.3 and Figure 2.6 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)
> define dimension item **as** (item_key, item_name, brand, 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]:
> > dollars_cost = **sum**(cost_in_dollars), units_shipped = **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 2.3. 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.                                           □

Instead of having users or experts explicitly define data cube dimensions, dimensions can be automatically generated or adjusted based on the examination of data distributions. DMQL primitives for specifying such automatic generation or adjustments are discussed in the following chapter.

## 2.2.4 Measures: their categorization and computation

*"How are measures computed?"*

To answer this question, we will first look at how measures can be categorized. Note that multidimensional points in the data cube space are defined by dimension-value pairs. For example, the dimension-value pairs in ⟨time= "Q1", location= "Vancouver", item= "computer"⟩ define a point in data cube space. 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 is partitioned into $n$ sets. The computation of the function on each partition derives one aggregate value. If the result derived by applying the function to the $n$ aggregate values is the same as that derived by applying the function on all the data without partitioning, the function can be computed in a 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.

- **algebraic**: An aggregate function is *algebraic* if it can be computed by an algebraic function with $M$ arguments (where $M$ is a bounded 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()`, `max_N()`, 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()` (i.e., the most frequently occurring item(s)), 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 can be difficult to compute holistic measures efficiently. Efficient techniques to *approximate* the computation of some holistic measures, however, do exist. For example, instead of computing the exact `median()`, there are techniques which can estimate the approximate median value for a large data set with satisfactory results. In many cases, such techniques are sufficient to overcome the difficulties of efficient computation of holistic measures.

**Example 2.7** Many measures of a data cube can be computed by relational aggregation operations. In Figure 2.4, we saw a star schema for *AllElectronics* sales which contains two measures, namely *dollars_sold* and *units_sold*. In Example 2.4, 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)
> 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 2.4 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 may 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 2.2.1. 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. Techniques for this are discussed in Chapter 9.

### 2.2.5   Introducing concept hierarchies
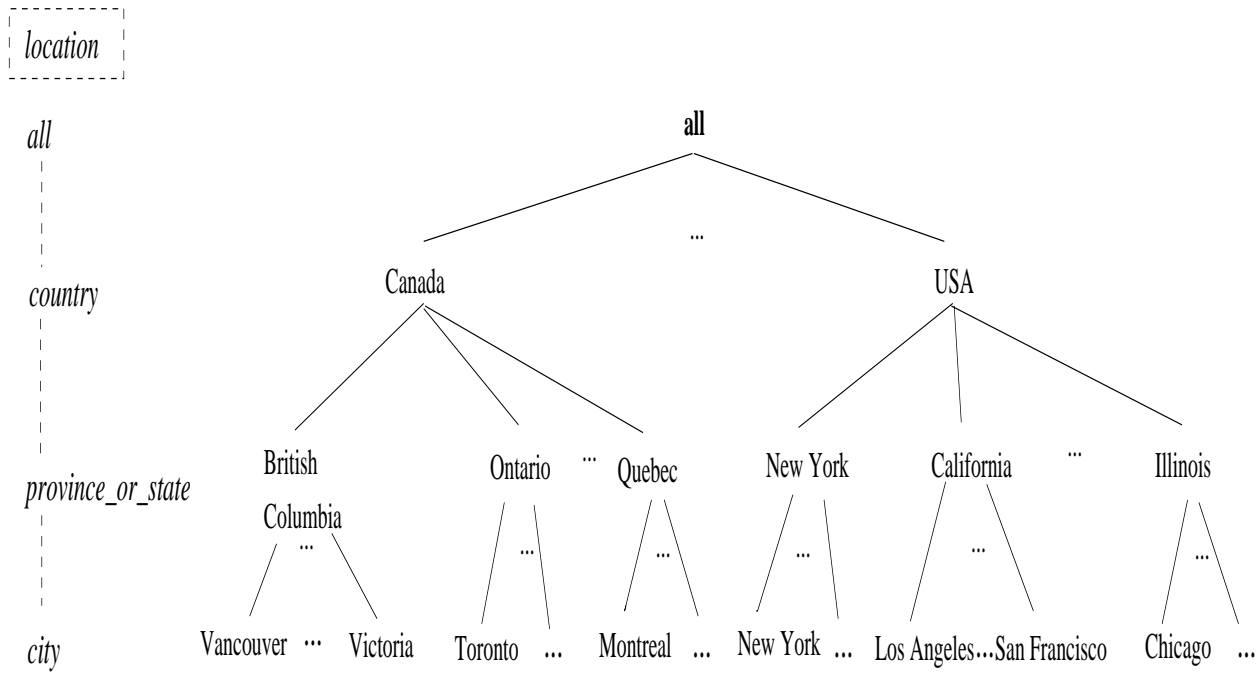
*"What is a concept hierarchy?"*

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, Montreal, 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 2.7.

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 2.8a). 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*" [1]. This lattice structure is shown in Figure 2.8b). 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 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, one may like to define a fiscal year starting on April 1, or an academic year starting on September 1.

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 2.9 for the dimension *price*.

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*.

---

[1] Since a *week* usually crosses the boundary of two consecutive months, it is usually not treated as a lower abstraction of *month*. Instead, it is often treated as a lower abstraction of *year*, since a year contains approximately 52 weeks.

Figure 2.7: A concept hierarchy for the dimension *location*.

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 3. Concept hierarchies are further discussed in Chapter 4.

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

## 2.2.6   OLAP operations in the multidimensional data model

*"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



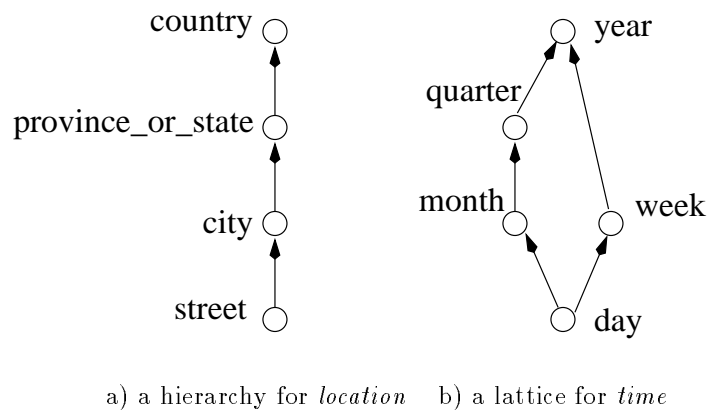a) a hierarchy for *location*     b) a lattice for *time*

Figure 2.8: Hierarchical and lattice structures of attributes in warehouse dimensions.
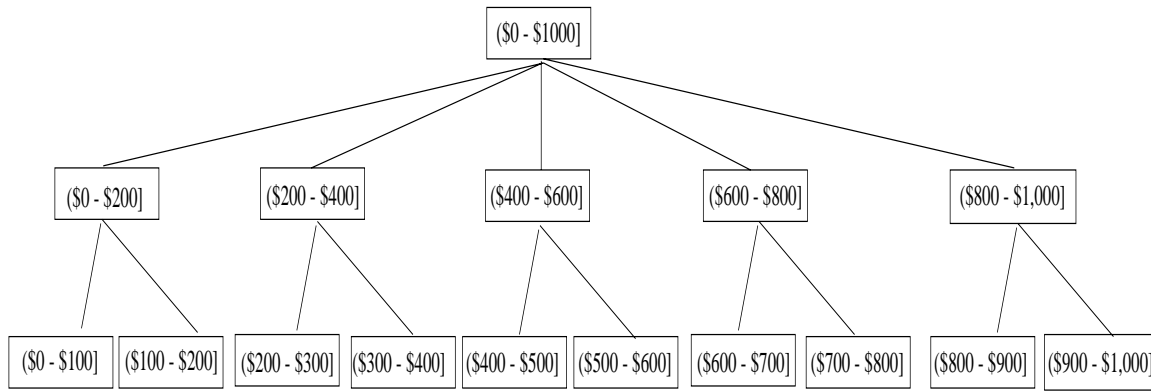
Figure 2.9: A concept hierarchy for the attribute *price*.

for interactive data analysis.

**Example 2.8** Let's have a look at some typical OLAP operations for multidimensional data. Each of the operations described below is illustrated in Figure 2.10. 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 data examined are for the cities Vancouver, Montreal, New York, and Chicago.

1. **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 2.10 shows the result of a roll-up operation performed on the central cube by climbing up the concept hierarchy for *location* given in Figure 2.7. 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.

2. **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 2.10 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 2.10 can occur by introducing an additional dimension, such as *customer_type*.

3. **slice and dice**: The *slice* operation performs a selection on one dimension of the given cube, resulting in a subcube. Figure 2.10 shows a slice operation where the sales data are selected from the central cube for the dimension *time* using the criteria *time="Q2"*. The *dice* operation defines a subcube by performing a selection on two or more dimensions. Figure 2.10 shows a dice operation on the central cube based on the following selection criteria which involves three dimensions: (*location="Montreal"* or *"Vancouver"*) and (*time="Q1"* or *"Q2"*) and (*item="home entertainment"* or *"computer"*).

4. **pivot (rotate)**: *Pivot* (also called *"rotate"*) is a visualization operation which rotates the data axes in view in order to provide an alternative presentation of the data. Figure 2.10 shows a pivot operation where the
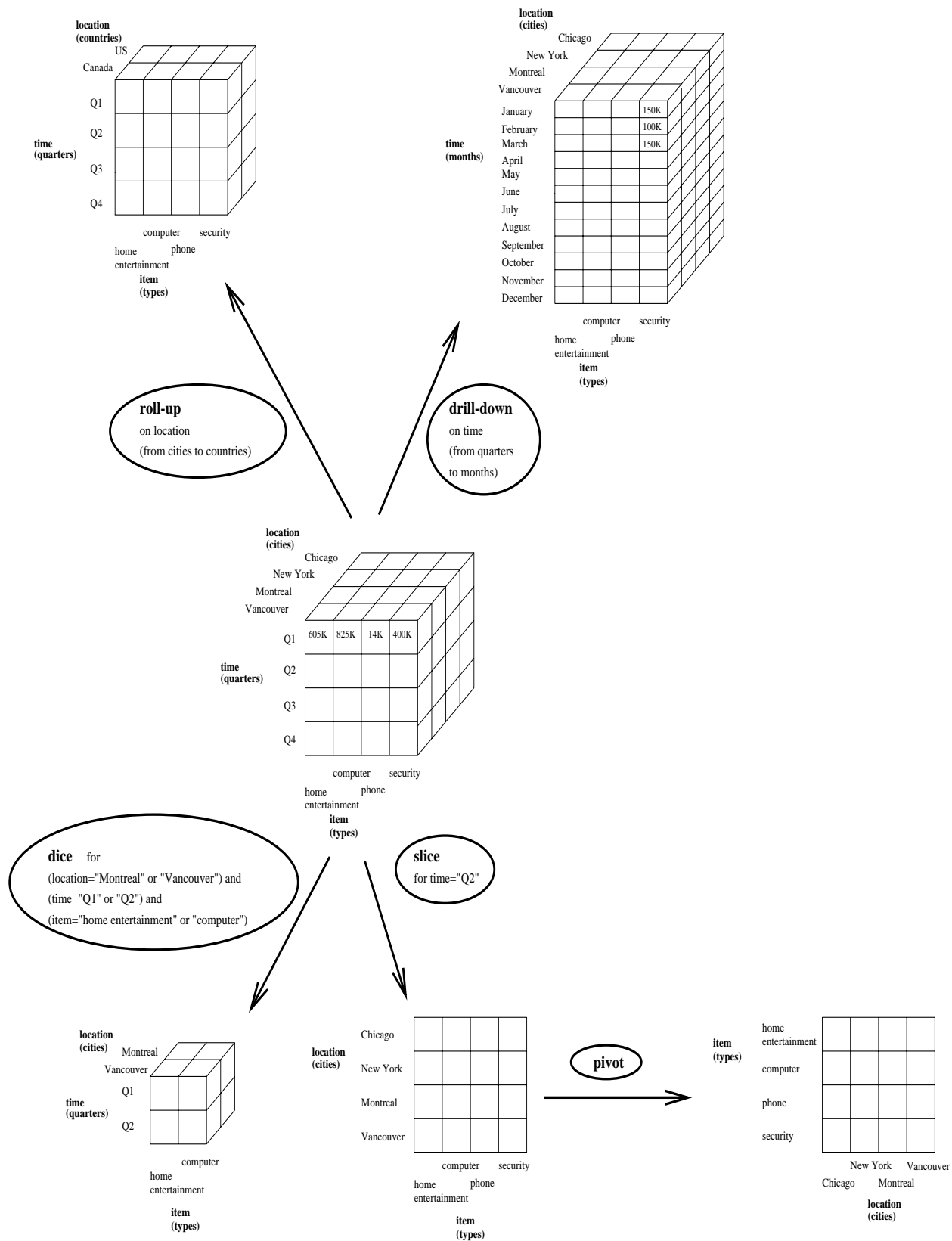
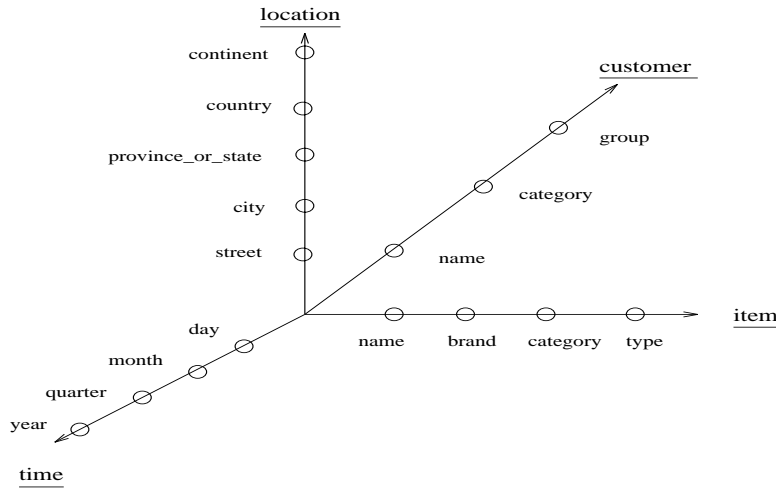Figure 2.10: Examples of typical OLAP operations on multidimensional data.

Figure 2.11: Modeling business queries: A starnet model.

*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.

5. **other OLAP operations**: Some OLAP systems offer additional drilling operations. For example, **drill-across** executes queries involving (i.e., acrosss) 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, etc., 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.

### 2.2.7   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.

**Example 2.9** A starnet query model for the *AllElectronics* data warehouse is shown in Figure 2.11. 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*, one 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.                                                                              □

## 2.3   Data warehouse architecture

### 2.3.1   Steps for the design and construction of data warehouses

**The design of a data warehouse: A business analysis framework**

*"What does the data warehouse provide for business analysts?"*

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 which accurately describes the organization. Third, a data warehouse facilitates *customer relationship marketing* 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 one needs 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 which 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 view point 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** which 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**

*"How can I design a data warehouse?"*

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 turn-around 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, e.g., orders, invoices, shipments, inventory, account administration, sales, and 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, e.g., individual transactions, individual daily snapshots, etc.

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 which 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 will include 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 will include 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.

### 2.3.2   A three-tier data warehouse architecture

*"What is data warehouse architecture like?"*

Data warehouses often adopt a three-tier architecture, as presented in Figure 2.12. The bottom tier is a **warehouse database server** which is almost always a relational database system. The middle tier is an **OLAP server** which is typically implemented using either (1) a **Relational OLAP (ROLAP)** model, i.e., an extended relational DBMS that maps operations on multidimensional data to standard relational operations; or (2) a **Multidimensional OLAP (MOLAP)** model, i.e., a special purpose server that directly implements multidimensional data and operations. The top tier is a **client**, which contains query and reporting tools, analysis tools, and/or data mining tools (e.g., trend analysis, prediction, and so on).
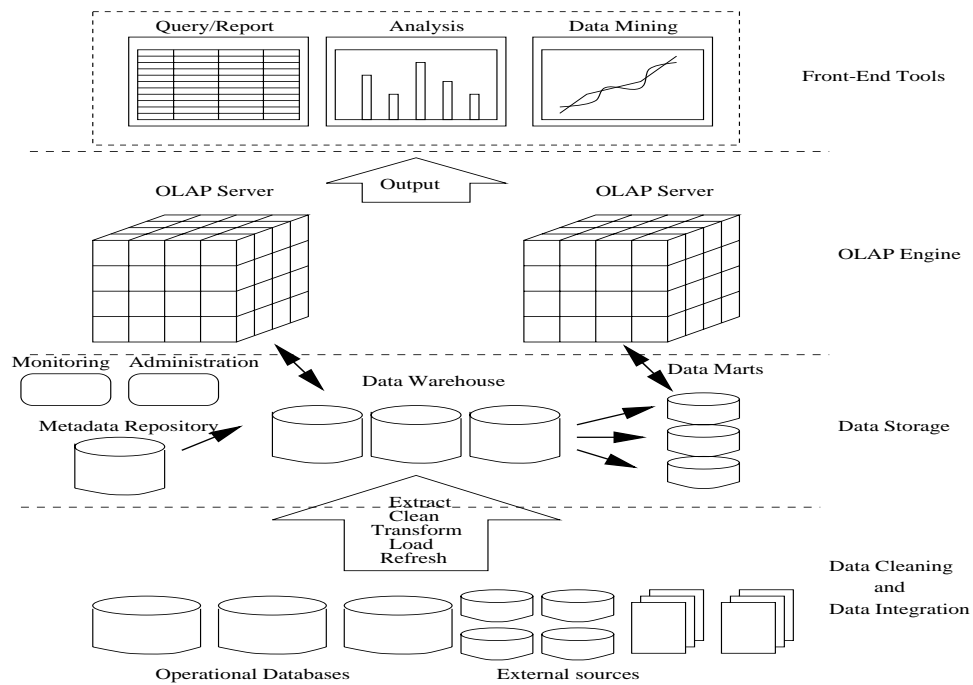
Figure 2.12: A three-tier data warehousing architecture.

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, UNIX 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-, Windows/NT-, or OS/2-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 into the following two classes:

  - *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.

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
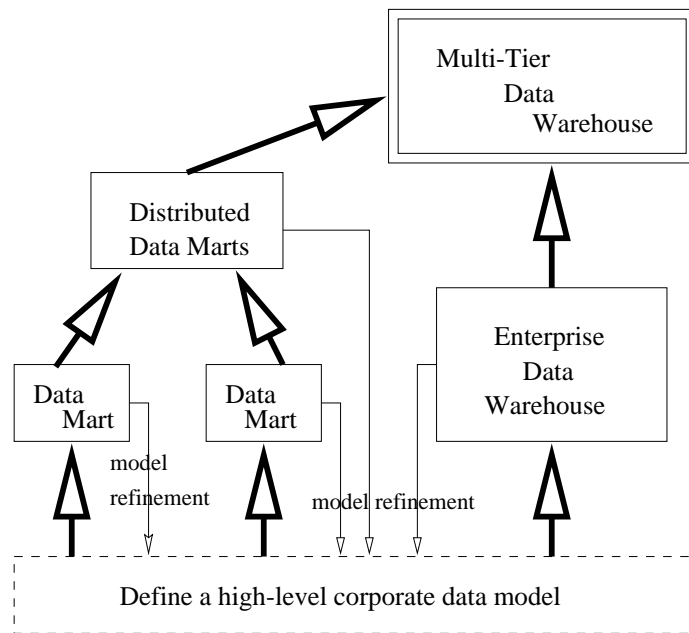
Figure 2.13: A recommended approach for data warehouse development.

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 2.13. 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 **multi-tier 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.

### 2.3.3   OLAP server architectures: ROLAP vs. MOLAP vs. HOLAP

*"What is OLAP server architecture like?"*

Logically, OLAP engines 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 engines must consider data storage issues. Implementations of a warehouse server engine for OLAP processing include:

- **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 and Metacube of Informix, for example, adopt the ROLAP approach[2].

- **Multidimensional OLAP (MOLAP) servers**: These servers support multidimensional views of data through array-based multidimensional storage engines. They map multidimensional views directly to data

---

[2] Information on these products can be found at www.informix.com and www.microstrategy.com, respectively.

cube array structures. For example, Essbase of Arbor is a MOLAP server. 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 (see Section 2.4) should be explored.

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

- **Hybrid OLAP (HOLAP) servers:** The hybrid OLAP approach combines ROLAP and MOLAP technology, benefitting 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 7.0 OLAP Services supports a hybrid OLAP server.

- **Specialized SQL servers**: To meet the growing demand of OLAP processing in relational databases, some relational and data warehousing firms (e.g., Redbrick) implement specialized SQL servers which provide advanced query language and query processing support for SQL queries over star and snowflake schemas in a read-only environment.

The OLAP functional architecture consists of three components: the *data store*, the *OLAP server*, and the *user presentation module*. The data store can be further classified as a *relational data store* or a *multidimensional data store*, depending on whether a ROLAP or MOLAP architecture is adopted.

*"So, how are data actually stored in ROLAP and MOLAP architectures?"*

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 2.10. Alternatively, separate summary fact tables can be used for each level of abstraction, to store only aggregated data.

**Example 2.10** Table 2.4 shows a summary fact table which contains both base fact data and aggregated data. The schema of the table is "⟨*record_identifier (RID), item, location, day, month, quarter, year, dollars_sold* (i.e., sales amount)⟩", where *day, month, quarter*, and *year* define the date of sales. Consider the tuple with an RID of 1001. The data of this tuple are at the base fact level. Here, the date of sales is October 15, 1997. Consider the tuple with an RID of 5001. This tuple is at a more general level of abstraction than the tuple having an RID of 1001. Here, the "*Main Street*" value for *location* has been generalized to "*Vancouver*". The *day* value has been generalized to all, so that the corresponding *time* value is October 1997. That is, the *dollars_sold* amount shown is an aggregation representing the entire month of October 1997, rather than just October 15, 1997. The special value all is used to represent subtotals in summarized data.

| RID | item | location | day | month | quarter | year | dollars_sold |
|------|------|-------------|-----|-------|---------|------|--------------|
| 1001 | TV | Main Street | 15 | 10 | Q4 | 1997 | 250.60 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 5001 | TV | Vancouver | all | 10 | Q4 | 1997 | 45,786.08 |
| ... | ... | ... | ... | ... | ... | ... | ... |

Table 2.4: Single table for base and summary facts.

□

MOLAP uses multidimensional array structures to store data for on-line analytical processing. For example, the data cube structure described and referred to throughout this chapter is such an array structure.

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. There are several alternative physical configuration options.

If a multidimensional data store resides at the client side, it results in a "fat client". In this case, the system response time may be quick since OLAP operations will not consume network traffic, and the network bottleneck happens only at the warehouse loading stage. However, loading a large data warehouse can be slow and the processing at the client side can be heavy, which may degrade the system performance. Moreover, data security could be a problem because data are distributed to multiple clients. A variation of this option is to partition the multidimensional data store and distribute selected subsets of the data to different clients.

Alternatively, a multidimensional data store can reside at the server site. One option is to set both the multidimensional data store and the OLAP server at the data mart site. This configuration is typical for data marts that are created by refining or re-engineering the data from an enterprise data warehouse.

A variation is to separate the multidimensional data store and OLAP server. That is, an OLAP server layer is added between the client and data mart. This configuration is used when the multidimensional data store is large, data sharing is needed, and the client is "thin" (i.e., does not require many resources).

### 2.3.4  SQL extensions to support OLAP operations

*"How can SQL be extended to support OLAP operations?"*

An OLAP server should support several data types including text, calendar, and numeric data, as well as data at different granularities (such as regarding the estimated and actual sales per item). An OLAP server should contain a calculation engine which includes domain-specific computations (such as for calendars) and a rich library of aggregate functions. Moreover, an OLAP server should include data load and refresh facilities so that *write* operations can update precomputed aggregates, and *write/load* operations are accompanied by data cleaning.

A multidimensional view of data is the foundation of OLAP. SQL extensions to support OLAP operations have been proposed and implemented in extended-relational servers. Some of these are enumerated as follows.

1. Extending the family of aggregate functions.

    Relational database systems have provided several useful aggregate functions, including `sum()`, `avg()`, `count()`, `min()`, and `max()` as SQL standards. OLAP query answering requires the extension of these standards to include other aggregate functions such as `rank()`, `N_tile()`, `median()`, and `mode()`. For example, a user may like to list the *top five most profitable items* (using `rank()`), list the *firms whose performance is in the bottom 10% in comparison to all other firms* (using `N_tile()`), or print *the most frequently sold items in March* (using `mode()`).

2. Adding reporting features.

    Many report writer softwares allow aggregate features to be evaluated on a time window. Examples include running totals, cumulative totals, moving averages, break points, etc. OLAP systems, to be truly useful for decision support, should introduce such facilities as well.

3. Implementing multiple group-by's.

    Given the multidimensional view point of data warehouses, it is important to introduce *group-by*'s for grouping sets of attributes. For example, one may want to list the total sales from 1996 to 1997 grouped by item, by region, and by quarter. Although this can be simulated by a set of SQL statements, it requires multiple scans of databases, and is thus a very inefficient solution. New operations, including `cube` and `roll-up`, have been introduced in some relational system products which explore efficient implementation methods.

## 2.4   Data warehouse implementation

Data warehouses contain huge volumes of data. OLAP engines 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. *"How can this be done?"*, you may wonder. In this section, we examine methods for the efficient implementation of data warehouse systems.
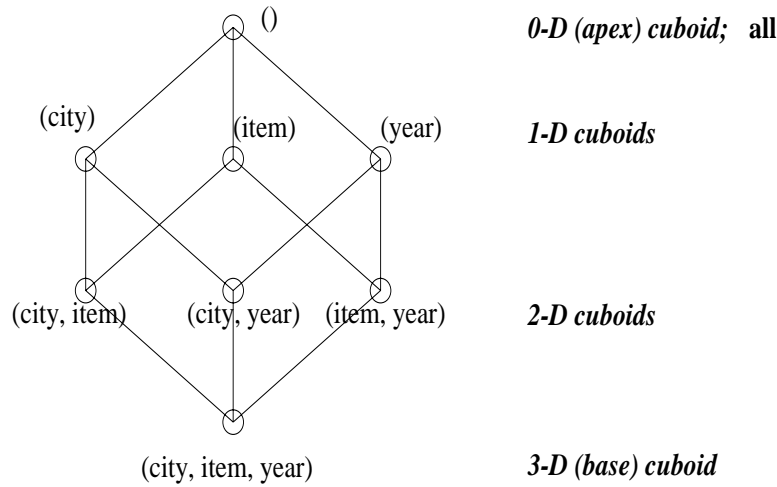
Figure 2.14: Lattice of cuboids, making up a 3-dimensional data cube. Each cuboid represents a different group-by. The base cuboid contains the three dimensions, *city, item*, and *year*.

## 2.4.1 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**.

### The `compute cube` operator and its implementation

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.

**Example 2.11** Suppose that you would like to create a data cube for *AllElectronics* sales which contains the following: *item, city, year*, and *sales_in_dollars*. You would like to be able to analyze the data, with queries such as the following:

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

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

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

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 three dimensions 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 2.14. 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. Consequently, it is represented by the special value all.

☐

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 2.2.3, the data cube in Example 2.11, can be defined as

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

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

    compute cube sales

explicitly instructs the system to compute the sales aggregate cuboids for all of the eight subsets of the set {*item, city, year*}, including the empty subset. A cube computation operator was first proposed and studied by Gray, et al. (1996).

On-line analytical processing may need to access different cuboids for different queries. Therefore, it does 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. Actually, 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 several dimensions associated with multiple level hierarchies.

"*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 just one level, such as *year*, but rather a hierarchy or a lattice, such as *day < week < 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:

$$T = \prod_{i=1}^{n}(L_i + 1),$$

where $L_i$ is the number of levels associated with dimension $i$ (excluding the *virtual* top level all since generalizing to all is equivalent to the removal of a dimension). This formula is based on the fact that at most one abstraction level in each dimension will appear in a cuboid. For example, if the cube has 10 dimensions and each dimension has 4 levels, the total number of cuboids that can be generated will be $5^{10} \approx 9.8 \times 10^6$.

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: (1) precompute only the base cuboid and none of the remaining "non-base" cuboids (**no materialization**), (2) precompute all of the cuboids (**full materialization**), and (3) selectively compute a proper subset of the whole set of possible cuboids (**partial materialization**). The first choice leads to computing expensive multidimensional aggregates on the fly, which could be slow. The second choice may require huge amounts of memory space in order to store all of the precomputed cuboids. The third choice presents an interesting trade-off between storage space and response time.

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

The selection of the subset of cuboids 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 selection. A popular approach is to materialize the set of cuboids having relatively simple structure. Even with this restriction, there are often still a large number of possible choices. Under a simplified assumption, a greedy algorithm has been proposed and has shown good performance.

Once the selected cuboids have been materialized, it is important to take advantage of them during query processing. This involves determining 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 on to the selected cuboid(s). These issues are discussed in Section 2.4.3 on query processing.

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

### Multiway array aggregation in the computation of data cubes

In order to ensure fast on-line analytical processing, however, we may need to precompute all of the cuboids for a given data cube. Cuboids may be stored on secondary storage, and accessed when necessary. Hence, it is important to explore efficient methods for computing all of the cuboids making up a data cube, that is, for full materialization. These methods must take into consideration the limited amount of main memory available for cuboid computation, as well as the time required for such computation. To simplify matters, we may exclude the cuboids generated by climbing up existing hierarchies along each dimension.

Since Relational OLAP (ROLAP) uses tuples and relational tables as its basic data structures, while the basic data structure used in multidimensional OLAP (MOLAP) is the multidimensional array, one would expect that ROLAP and MOLAP each explore very different cube computation techniques.

ROLAP cube computation uses the following major optimization techniques.

1. Sorting, hashing, and grouping operations are applied to the dimension attributes in order to reorder and cluster related tuples.

2. Grouping is performed on some subaggregates as a "partial grouping step". These "partial groupings" may be used to speed up the computation of other subaggregates.

3. Aggregates may be computed from previously computed aggregates, rather than from the base fact tables.

*"How do these optimization techniques apply to MOLAP?"* ROLAP uses value-based addressing, where dimension values are accessed by key-based addressing search strategies. In contrast, MOLAP uses direct array addressing, where dimension values are accessed via the position or index of their corresponding array locations. Hence, MOLAP cannot perform the value-based reordering of the first optimization technique listed above for ROLAP. Therefore, a different approach should be developed for the array-based cube construction of MOLAP, such as the following.

1. Partition the array into chunks. **A chunk** is a subcube that is small enough to fit into the memory available for cube computation. **Chunking** is a method for dividing an $n$-dimensional array into small $n$-dimensional chunks, where each chunk is stored as an object on disk. The chunks are compressed so as to remove wasted space resulting from empty array cells (i.e., cells that do not contain any valid data). For instance, "*chunkID + offset*" can be used as a cell addressing mechanism to **compress a sparse array structure** and when searching for cells within a chunk. Such a compression technique is powerful enough to handle sparse cubes, both on disk and in memory.

2. Compute aggregates by visiting (i.e., accessing the values at) cube cells. The order in which cells are visited can be optimized so as to *minimize the number of times that each cell must be revisited*, thereby reducing memory access and storage costs. The trick is to exploit this ordering so that partial aggregates can be computed simultaneously, and any unnecessary revisiting of cells is avoided.

   Since this chunking technique involves "overlapping" some of the aggregation computations, it is referred to as **multiway array aggregation** in data cube computation.

We explain this approach to MOLAP cube construction by looking at a concrete example.

**Example 2.12** Consider a 3-D data array containing the three dimensions, $A$, $B$, and $C$.

- The 3-D array is partitioned into small, memory-based chunks. In this example, the array is partitioned into 64 chunks as shown in Figure 2.15. Dimension $A$ is organized into 4 partitions, $a_0, a_1, a_2$, and $a_3$. Dimensions $B$ and $C$ are similarly organized into 4 partitions each. Chunks 1, 2, ..., 64 correspond to the subcubes $a_0b_0c_0$, $a_1b_0c_0$, ..., $a_3b_3c_3$, respectively. Suppose the size of the array for each dimension, $A$, $B$, and $C$ is 40, 400, 4000, respectively.

- Full materialization of the corresponding data cube involves the computation of all of the cuboids defining this cube. These cuboids consist of:
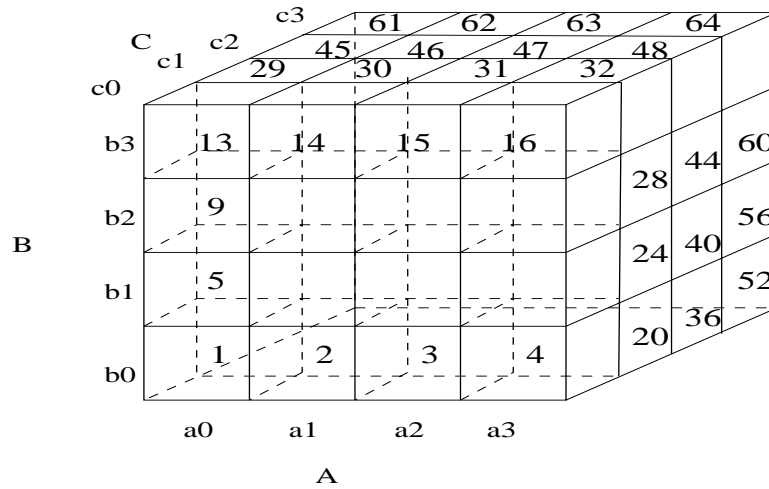
Figure 2.15: A 3-D array for the dimensions $A$, $B$, and $C$, organized into 64 *chunks*.

- The base cuboid, denoted by $ABC$ (from which all of the other cuboids are directly or indirectly computed). This cube is already computed and corresponds to the given 3-D array.

- The 2-D cuboids, $AB$, $AC$, and $BC$, which respectively correspond to the group-by's $AB$, $AC$, and $BC$. These cuboids must be computed.

- The 1-D cuboids, $A$, $B$, and $C$, which respectively correspond to the group-by's $A$, $B$, and $C$. These cuboids must be computed.

- The 0-D (apex) cuboid, denoted by all, which corresponds to the group-by (), i.e., there is no group-by here. This cuboid must be computed.

Let's look at how the multiway array aggregation technique is used in this computation.

- There are many possible orderings with which chunks can be read into memory for use in cube computation. Consider the ordering labeled from 1 to 64, shown in Figure 2.15. Suppose we would like to compute the $b_0c_0$ chunk of the $BC$ cuboid. We allocate space for this chunk in *"chunk memory"*. By scanning chunks 1 to 4 of $ABC$, the $b_0c_0$ chunk is computed. That is, the cells for $b_0c_0$ are aggregated over $a_0$ to $a_3$.

  The chunk memory can then be assigned to the next chunk, $b_1c_0$, which completes its aggregation after the scanning of the next 4 chunks of $ABC$: 5 to 8.

  Continuing in this way, the entire $BC$ cuboid can be computed. Therefore, only *one* chunk of $BC$ needs to be in memory, at a time, for the computation of all of the chunks of $BC$.

- In computing the $BC$ cuboid, we will have scanned each of the 64 chunks. *"Is there a way to avoid having to rescan all of these chunks for the computation of other cuboids, such as $AC$ and $AB$?"* The answer is, most definitely - *yes*. This is where the multiway computation idea comes in. For example, when chunk 1, i.e., $a_0b_0c_0$, is being scanned (say, for the computation of the 2-D chunk $b_0c_0$ of $BC$, as described above), all of the other 2-D chunks relating to $a_0b_0c_0$ can be simultaneously computed. That is, when $a_0b_0c_0$, is being scanned, each of the three chunks, $b_0c_0$, $a_0c_0$, and $a_0b_0$, on the three 2-D aggregation planes, $BC$, $AC$, and $AB$, should be computed then as well. In other words, multiway computation aggregates to each of the 3-D planes while a 3-D chunk is in memory.

Let's look at how different orderings of chunk scanning and of cuboid computation can affect the overall data cube computation efficiency. Recall that the size of the dimensions $A$, $B$, and $C$ is 40, 400, and 4000, respectively. Therefore, the largest 2-D plane is $BC$ (of size $400 \times 4,000 = 1,600,000$). The second largest 2-D plane is $AC$ (of size $40 \times 4,000 = 160,000$). $AB$ is the smallest 2-D plane (with a size of $40 \times 400 = 16,000$).

- Suppose that the chunks are scanned in the order shown, from chunk 1 to 64. By scanning in this order, one chunk of the largest 2-D plane, $BC$, is *fully* computed for each row scanned. That is, $b_0c_0$ is fully aggregated
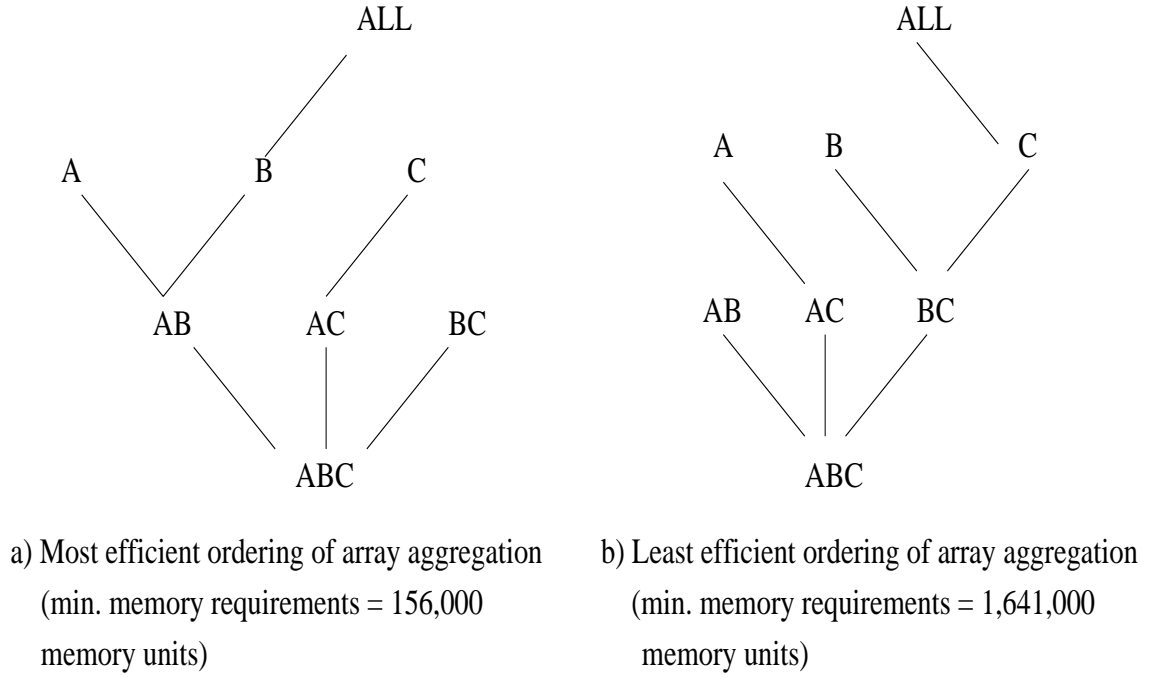
Figure 2.16: Two orderings of multiway array aggregation for computation of the 3-D cube of Example 2.12.

a) Most efficient ordering of array aggregation (min. memory requirements = 156,000 memory units)

b) Least efficient ordering of array aggregation (min. memory requirements = 1,641,000 memory units)

after scanning the row containing chunks 1 to 4; $b_1c_0$ is fully aggregated after scanning chunks 5 to 8, and so on. In comparison, the complete computation of one chunk of the second largest 2-D plane, $AC$, requires scanning 13 chunks (given the ordering from 1 to 64). For example, $a_0c_0$ is fully aggregated after the scanning of chunks 1, 5, 9, and 13. Finally, the complete computation of one chunk of the smallest 2-D plane, $AB$, requires scanning 49 chunks. For example, $a_0b_0$ is fully aggregated after scanning chunks 1, 17, 33, and 49. Hence, $AB$ requires the longest scan of chunks in order to complete its computation. To avoid bringing a 3-D chunk into memory more than once, the minimum memory requirement for holding all relevant 2-D planes in chunk memory, according to the chunk ordering of 1 to 64 is as follows: $40 \times 400$ (for the whole $AB$ plane) $+ 40 \times 1,000$ (for one row of the $AC$ plane) $+ 100 \times 1,000$ (for one chunk of the $BC$ plane) $= 16,000 + 40,000 + 100,000 = 156,000$.

- Suppose, instead, that the chunks are scanned in the order 1, 17, 33, 49, 5, 21, 37, 53, etc. That is, suppose the scan is in the order of first aggregating towards the $AB$ plane, and then towards the $AC$ plane and lastly towards the $BC$ plane. The minimum memory requirement for holding 2-D planes in chunk memory would be as follows: $400 \times 4,000$ (for the whole $BC$ plane) $+ 40 \times 1,000$ (for one row of the $AC$ plane) $+ 10 \times 100$ (for one chunk of the $AB$ plane) $= 1,600,000 + 40,000 + 1,000 = 1,641,000$. Notice that this is *more than 10 times* the memory requirement of the scan ordering of 1 to 64.

- Similarly, one can work out the minimum memory requirements for the multiway computation of the 1-D and 0-D cuboids. Figure 2.16 shows a) the most efficient ordering and b) the least efficient ordering, based on the minimum memory requirements for the data cube computation. The most efficient ordering is the chunk ordering of 1 to 64.

- In conclusion, this example shows that the planes should be sorted and computed according to their size in ascending order. Since $|AB| < |AC| < |BC|$, the $AB$ plane should be computed first, followed by the $AC$ and $BC$ planes. Similarly, for the 1-D planes, $|A| < |B| < |C|$ and therefore the $A$ plane should be computed before the $B$ plane, which should be computed before the $C$ plane.

□

Example 2.12 assumes that there is enough memory space for *one-pass* cube computation (i.e., to compute all of the cuboids from one scan of all of the chunks). If there is insufficient memory space, the computation will require

more than one pass through the 3-D array. In such cases, however, the basic principle of ordered chunk computation remains the same.

"*Which is faster — ROLAP or MOLAP cube computation?*" With the use of appropriate sparse array compression techniques and careful ordering of the computation of cuboids, it has been shown that MOLAP cube computation is significantly faster than ROLAP (relational record-based) computation. Unlike ROLAP, the array structure of MOLAP does not require saving space to store search keys. Furthermore, MOLAP uses direct array addressing, which is faster than the key-based addressing search strategy of ROLAP. In fact, for ROLAP cube computation, instead of cubing a table directly, it is even faster to convert the table to an array, cube the array, and then convert the result back to a table.

## 2.4.2  Indexing OLAP data

To facilitate efficient data accessing, most data warehouse systems support index structures and materialized views (using cuboids). 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.

Bitmap indexing is especially advantageous for low cardinality domains because comparison, join, and aggregation operations are then reduced to bit-arithmetic, which substantially reduces the processing time. Bitmap indexing also 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 indexing to identify subcubes that are of interest.

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

## 2.4.3  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 transforming 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

---

[3] A set of attributes in a relation schema that forms a primary key for another schema is called a **foreign key**.

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 2.13** Suppose that we define a data cube for *AllElectronics* of the form *"sales [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 = 1997"*. Also, suppose that there are four materialized cuboids available, as follows.

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

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

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

- cuboid 4: {*item_name, province_or_state*} where *year = 1997*.

*"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, and (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** sever 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 (Section 2.4.1) should therefore be applied.

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 arrays 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.

### 2.4.4 Metadata repository

*"What are metadata?"*

**Metadata** are data about data. When used in a data warehouse, metadata are the data that define warehouse objects. 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:

- a description of *the structure of the data warehouse*. This 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; and

- *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). Notice that the only type of summarization that is permanently stored in the data warehouse is that data which is frequently used.

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).

### 2.4.5    Data warehouse back-end tools and utilities

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

1. **data extraction**, which typically gathers data from multiple, heterogeneous, and external sources;

2. **data cleaning**, which detects errors in the data and rectifies them when possible;

3. **data transformation**, which converts data from legacy or host format to warehouse format;

4. **load**, which sorts, summarizes, consolidates, computes views, checks integrity, and builds indices and partitions; and

5. **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.

Since we are mostly interested in the aspects of data warehousing technology related to data mining, we will not get into the details of these tools and recommend interested readers to consult books dedicated to data warehousing technology.

## 2.5    Further development of data cube technology

In this section, you will study further developments in data cube technology. Section 2.5.1 describes data mining by *discovery-driven exploration of data cubes*, where anomalies in the data are automatically detected and marked for the user with visual cues. Section 2.5.2 describes *multifeature cubes* for complex data mining queries involving multiple dependent aggregates at multiple granularities.

## 2.5.1 Discovery-driven exploration of data cubes

As we have seen in this chapter, data can be summarized and stored in a multidimensional data cube of an OLAP system. A user or analyst can search for interesting patterns in the cube by specifying a number of OLAP operations, such as drill-down, roll-up, slice, and dice. While these tools are available to help the user explore the data, the discovery process is not automated. It is the user who, following her own intuition or hypotheses, tries to recognize exceptions or anomalies in the data. This **hypothesis-driven exploration** has a number of disadvantages. The search space can be very large, making manual inspection of the data a daunting and overwhelming task. High level aggregations may give no indication of anomalies at lower levels, making it easy to overlook interesting patterns. Even when looking at a subset of the cube, such as a slice, the user is typically faced with many data values to examine. The sheer volume of data values alone makes it easy for users to miss exceptions in the data if using hypothesis-driven exploration.

**Discovery-driven exploration** is an alternative approach in which precomputed measures indicating data exceptions are used to guide the user in the data analysis process, at all levels of aggregation. We hereafter refer to these measures as *exception indicators*. Intuitively, an **exception** is a data cube cell value that is significantly different from the value anticipated, based on a statistical model. The model considers variations and patterns in the measure value across *all of the dimensions* to which a cell belongs. For example, if the analysis of item-sales data reveals an increase in sales in December in comparison to all other months, this may seem like an exception in the time dimension. However, it is not an exception if the item dimension is considered, since there is a similar increase in sales for other items during December. The model considers exceptions hidden at all aggregated group-by's of a data cube. Visual cues such as background color are used to reflect the degree of exception of each cell, based on the precomputed exception indicators. Efficient algorithms have been proposed for cube construction, as discussed in Section 2.4.1. The computation of exception indicators can be overlapped with cube construction, so that the overall construction of data cubes for discovery-driven exploration is efficient.

Three measures are used as exception indicators to help identify data anomalies. These measures indicate the degree of surprise that the quantity in a cell holds, with respect to its expected value. The measures are computed and associated with every cell, for all levels of aggregation. They are:

1. **SelfExp**: This indicates the degree of surprise of the cell value, relative to other cells at the same level of aggregation.

2. **InExp**: This indicates the degree of surprise somewhere beneath the cell, if we were to drill down from it.

3. **PathExp**: This indicates the degree of surprise for each drill-down path from the cell.

The use of these measures for discovery-driven exploration of data cubes is illustrated in the following example.

**Example 2.14** Suppose that you would like to analyze the monthly sales at *AllElectronics* as a percentage difference from the previous month. The dimensions involved are *item*, *time*, and *region*. You begin by studying the data aggregated over all items and sales regions for each month, as shown in Figure 2.17.

To view the exception indicators, you would click on a button marked `highlight exceptions` on the screen. This translates the SelfExp and InExp values into visual cues, displayed with each cell. The background color of each cell is based on its SelfExp value. In addition, a box is drawn around each cell, where the thickness and color of the box are a function of its InExp value. Thick boxes indicate high InExp values. In both cases, the darker the color is, the greater the degree of exception is. For example, the dark thick boxes for sales during July, August, and September signal the user to explore the lower level aggregations of these cells by drilling down.

Drill downs can be executed along the aggregated *item* or *region* dimensions. Which path has more exceptions? To find this out, you select a cell of interest and trigger a `path exception` module that colors each dimension based on the PathExp value of the cell. This value reflects the degree of surprise of that path. Consider the PathExp indicators for *item* and *region* in the upper left-hand corner of Figure 2.17. We see that the path along *item* contains more exceptions, as indicated by the darker color.

A drill-down along *item* results in the cube slice of Figure 2.18, showing the sales over time for each item. At this point, you are presented with many different sales values to analyze. By clicking on the `highlight exceptions` button, the visual cues are displayed, bringing focus towards the exceptions. Consider the sales difference of 41% for "Sony b/w printers" in September. This cell has a dark background, indicating a high SelfExp value, meaning that the cell is an exception. Consider now the the sales difference of -15% for "Sony b/w printers" in November, and of -11% in

| item | all |
|------|-----|
| region | all |

| Sum of sales | month | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec |
| Total | | 1% | -1% | 0% | 1% | 3% | -1 | -9% | -1% | 2% | -4% | 3% |

Figure 2.17: Change in sales over time.

| Avg sales | month | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| item | Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec |
| Sony b/w printer | | 9% | -8% | 2% | -5% | 14% | -4% | 0% | 41% | -13% | -15% | -11% |
| Sony color printer | | 0% | 0% | 3% | 2% | 4% | -10% | -13% | 0% | 4% | -6% | 4% |
| HP b/w printer | | -2% | 1% | 2% | 3% | 8% | 0% | -12% | -9% | 3% | -3% | 6% |
| HP color printer | | 0% | 0% | -2% | 1% | 0% | -1% | -7% | -2% | 1% | -5% | 1% |
| IBM home computer | | 1% | -2% | -1% | -1% | 3% | 3% | -10% | 4% | 1% | -4% | -1% |
| IBM laptop computer | | 0% | 0% | -1% | 3% | 4% | 2% | -10% | -2% | 0% | -9% | 3% |
| Toshiba home computer | | -2% | -5% | 1% | 1% | -1% | 1% | 5% | -3% | -5% | -1% | -1% |
| Toshiba laptop computer | | 1% | 0% | 3% | 0% | -2% | -2% | -5% | 3% | 2% | -1% | 0% |
| Logitech mouse | | 3% | -2% | -1% | 0% | 4% | 6% | -11% | 2% | 1% | -4% | 0% |
| Ergo-way mouse | | 0% | 0% | 2% | 3% | 1% | -2% | -2% | -5% | 0% | -5% | 8% |

Figure 2.18: Change in sales for each item-time combination.

| item | IBM home computer | | | | | | | | | | | |
|------|------|------|------|------|------|------|------|------|------|------|------|------|
| Avg sales | month | | | | | | | | | | | |
| region | Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec |
| North |  | -1% | -3% | -1% | 0% | 3% | 4% | -7% | 1% | 0% | -3% | -3% |
| South |  | -1% | 1% | -9% | 6% | -1% | -39% | 9% | -34% | 4% | 1% | 7% |
| East |  | -1% | -2% | 2% | -3% | 1% | 18% | -2% | 11% | -3% | -2% | -1% |
| West |  | 4% | 0% | -1% | -3% | 5% | 1% | -18% | 8% | 5% | -8% | 1% |

Figure 2.19: Change in sales for the item *"IBM home computer"* per region.

December. The -11% value for December is marked as an exception, while the -15% value is not, even though -15% is a bigger deviation than -11%. This is because the exception indicators consider all of the dimensions that a cell is in. Notice that the December sales of most of the other items have a large positive value, while the November sales do not. Therefore, by considering the position of the cell in the cube, the sales difference for "Sony b/w printers" in December is exceptional, while the November sales difference of this item is not.

The InExp values can be used to indicate exceptions at lower levels that are not visible at the current level. Consider the cells for "IBM home computers" in July and September. These both have a dark thick box around them, indicating high InExp values. You may decide to further explore the sales of "IBM home computers" by drilling down along *region*. The resulting sales difference by region is shown in Figure 2.19, where the `highlight exceptions` option has been invoked. The visual cues displayed make it easy to instantly notice an exception for the sales of "IBM home computers" in the southern region, where such sales have decreased by -39% and -34% in July and September, respectively. These detailed exceptions were far from obvious when we were viewing the data as an item-time group-by, aggregated over region in Figure 2.18. Thus, the InExp value is useful for searching for exceptions at lower level cells of the cube. Since there are no other cells in Figure 2.19 having a high InExp value, you may roll up back to the data of Figure 2.18, and choose another cell from which to drill down. In this way, the exception indicators can be used to guide the discovery of interesting anomalies in the data. □

*"How are the exception values computed?"* The SelfExp, InExp, and PathExp measures are based on a statistical method for table analysis. They take into account all of the group-by (aggregations) in which a given cell value participates. A cell value is considered an exception based on how much it differs from its expected value, where its expected value is determined with a statistical model described below. The difference between a given cell value and its expected value is called a **residual**. Intuitively, the larger the residual, the more the given cell value is an exception. The comparison of residual values requires us to scale the values based on the expected standard deviation associated with the residuals. A cell value is therefore considered an exception if its scaled residual value exceeds a prespecified threshold. The SelfExp, InExp, and PathExp measures are based on this scaled residual.

The expected value of a given cell is a function of the higher level group-by's of the given cell. For example, given a cube with the three dimensions $A, B$, and $C$, the expected value for a cell at the $i$th position in $A$, the $j$th position in $B$, and the $k$th position in $C$ is a function of $\gamma, \gamma_i^A, \gamma_j^B, \gamma_k^C, \gamma_{ij}^{AB}, \gamma_{ik}^{AC}$, and $\gamma_{jk}^{BC}$, which are coefficients of the statistical model used. The coefficients reflect how different the values at more detailed levels are, based on generalized impressions formed by looking at higher level aggregations. In this way, the exception quality of a cell value is based on the exceptions of the values below it. Thus, when seeing an exception, it is natural for the user to further explore the exception by drilling down.

*"How can the data cube be efficiently constructed for discovery-driven exploration?"* This computation consists of three phases. The first step involves the computation of the aggregate values defining the cube, such as `sum` or `count`, over which exceptions will be found. There are several efficient techniques for cube computation, such as the multiway array aggregation technique discussed in Section 2.4.1. The second phase consists of model fitting, in

which the coefficients mentioned above are determined and used to compute the standardized residuals. This phase can be overlapped with the first phase since the computations involved are similar. The third phase computes the SelfExp, InExp, and PathExp values, based on the standardized residuals. This phase is computationally similar to phase 1. Therefore, the computation of data cubes for discovery-driven exploration can be done efficiently.

### 2.5.2   Complex aggregation at multiple granularities: Multifeature cubes

Data cubes facilitate the answering of data mining queries as they allow the computation of aggregate data at multiple levels of granularity. In this section, you will learn about *multifeature cubes* which compute complex queries involving multiple dependent aggregates at multiple granularities. These cubes are very useful in practice. Many complex data mining queries can be answered by multifeature cubes without any significant increase in computational cost, in comparison to cube computation for simple queries with standard data cubes.

All of the examples in this section are from the Purchases data of *AllElectronics*, where an *item* is purchased in a sales *region* on a business day (*year, month, day*). The shelf life in months of a given item is stored in *shelf*. The item price and sales (in dollars) at a given region are stored in *price* and *sales*, respectively. To aid in our study of multifeature cubes, let's first look at an example of a simple data cube.

**Example 2.15   Query 1: A simple data cube query:** Find the total sales in 1997, broken down by item, region, and month, with subtotals for each dimension.

To answer Query 1, a data cube is constructed which aggregates the total sales at the following 8 different levels of granularity: {(*item, region, month*), (*item, region*), (*item, month*), (*month, region*), (*item*), (*month*), (*region*), ()}, where () represents all. There are several techniques for computing such data cubes efficiently (Section 2.4.1).    □

Query 1 uses a data cube like that studied so far in this chapter. We call such a data cube a simple data cube since it does not involve any dependent aggregates.

"*What is meant by "dependent aggregates"?*" We answer this by studying the following example of a complex query.

**Example 2.16   Query 2: A complex query:** Grouping by all subsets of {item, region, month}, find the maximum price in 1997 for each group, and the total sales among all maximum price tuples.

The specification of such a query using standard SQL can be long, repetitive, and difficult to optimize and maintain. Alternatively, Query 2 can be specified concisely using an extended SQL syntax as follows:

```
select      item, region, month, MAX(price), SUM(R.sales)
from        Purchases
where       year = 1997
cube by     item, region, month: R
such that   R.price = MAX(price)
```

The tuples representing purchases in 1997 are first selected. The cube by clause computes aggregates (or group-by's) for all possible combinations of the attributes item, region, and month. It is an $n$-dimensional generalization of the group by clause. The attributes specified in the cube by clause are the **grouping attributes**. Tuples with the same value on all grouping attributes form one group. Let the groups be $g_1, .., g_r$. For each group of tuples $g_i$, the maximum price $max_{g_i}$ among the tuples forming the group is computed. The variable $R$ is a **grouping variable**, ranging over all tuples in group $g_i$ whose price is equal to $max_{g_i}$ (as specified in the such that clause). The sum of sales of the tuples in $g_i$ that $R$ ranges over is computed, and returned with the values of the grouping attributes of $g_i$. The resulting cube is a **multifeature cube** in that it supports complex data mining queries for which multiple dependent aggregates are computed at a variety of granularities. For example, the sum of sales returned in Query 2 is dependent on the set of maximum price tuples for each group.    □

Let's look at another example.

**Example 2.17   Query 3: An even more complex query:** Grouping by all subsets of {item, region, month}, find the maximum price in 1997 for each group. Among the maximum price tuples, find the minimum and maximum

{=MIN(R1.shelf)}          {=MAX(R1.shelf)}

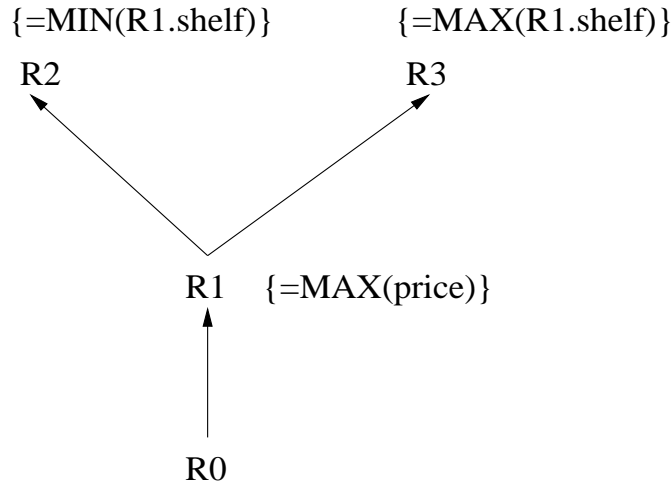R2                                  R3

R1    {=MAX(price)}

R0

Figure 2.20: A multifeature cube graph for Query 3.

item shelf lives. Also find the fraction of the total sales due to tuples that have minimum shelf life within the set of all maximum price tuples, and the fraction of the total sales due to tuples that have maximum shelf life within the set of all maximum price tuples.

The **multifeature cube graph** of Figure 2.20 helps illustrate the aggregate dependencies in the query. There is one node for each grouping variable, plus an additional initial node, R0. Starting from node R0, the set of maximum price tuples in 1997 is first computed (node R1). The graph indicates that grouping variables R2 and R3 are "dependent" on R1, since a directed line is drawn from R1 to each of R2 and R3. In a multifeature cube graph, a directed line from grouping variable $R_i$ to $R_j$ means that $R_j$ always ranges over a subset of the tuples that $R_i$ ranges for. When expressing the query in extended SQL, we write "$R_j$ in $R_i$" as shorthand to refer to this case. For example, the minimum shelf life tuples at R2 range over the maximum price tuples at R1, i.e., R2 in R1. Similarly, the maximum shelf life tuples at R3 range over the maximum price tuples at R1, i.e., R3 in R1.

From the graph, we can express Query 3 in extended SQL as follows:

```
select     item, region, month, MAX(price), MIN(R1.shelf), MAX(R1.shelf),
           SUM(R1.sales), SUM(R2.sales), SUM(R3.sales)
from       Purchases
where      year = 1997
cube by    item, region, month: R1, R2, R3
such that  R1.price = MAX(price) and
           R2 in R1 and R2.shelf = MIN(R1.shelf) and
           R3 in R1 and R3.shelf = MAX(R1.shelf)
```

□

*"How can multifeature cubes be computed efficiently?"* The computation of a multifeature cube depends on the types of aggregate functions used in the cube. Recall in Section 2.2.4, we saw that aggregate functions can be categorized as either *distributive* (such as `count()`, `sum()`, `min()`, and `max()`), *algebraic* (such as `avg()`, `min_N()`, `max_N()`), or *holistic* (such as `median()`, `mode()`, and `rank()`). Multifeature cubes can be organized into the same categories.

Intuitively, Query 2 is a distributive multifeature cube since we can distribute its computation by incrementally generating the output of the cube at a higher level granularity using only the output of the cube at a lower level granularity. Similarly, Query 3 is also distributive. Some multifeature cubes that are not distributive may be "converted" by adding aggregates to the `select` clause so that the resulting cube is distributive. For example, suppose that the `select` clause for a given multifeature cube has `AVG(sales)`, but neither `COUNT(sales)` nor `SUM(sales)`. By adding `SUM(sales)` to the `select` clause, the resulting data cube is distributive. The original cube is therefore algebraic. In the new distributive cube, the average sales at a higher level granularity can be computed from the average and total sales at lower level granularities. A cube that is neither distributive nor algebraic is holistic.

The type of multifeature cube determines the approach used in its computation. There are a number of methods for the efficient computation of data cubes (Section 2.4.1). The basic strategy of these algorithms is to exploit the lattice structure of the multiple granularities defining the cube, where higher level granularities are computed from lower level granularities. This approach suits distributive multifeature cubes, as described above. For example, in Query 2, the computation of `MAX(price)` at a higher granularity group can be done by taking the maximum of all of the `MAX(price)` values at the lower granularity groups. Similarly, `SUM(sales)` can be computed for a higher level group by summing all of the `SUM(sales)` values in its lower level groups. Some algorithms for efficient cube construction employ optimization techniques based on the estimated size of answers of groups within a data cube. Since the output size for each group in a multifeature cube is constant, the same estimation techniques can be used to estimate the size of intermediate results. Thus, the basic algorithms for efficient computation of simple data cubes can be used to compute distributive multifeature cubes for complex queries without any increase in I/O complexity. There may be a negligible increase in the CPU cost if the aggregate function of the multifeature cube is more complex than, say, a simple `SUM()`. Algebraic multifeature cubes must first be transformed into distributive multifeature cubes in order for these algorithms to apply. The computation of holistic multifeature cubes is sometimes significantly more expensive than the computation of distributive cubes, although the CPU cost involved is generally acceptable. Therefore, multifeature cubes can be used to answer complex queries with very little additional expense in comparison to simple data cube queries.

## 2.6  From data warehousing to data mining

### 2.6.1  Data warehouse usage

Data warehouses and data marts are used in a wide range of applications. Business executives in almost every industry use the data collected, integrated, preprocessed, and stored 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 which 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* which 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, but 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, textual, spatial, and multimedia data which 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 net effectiveness of promotions overall.

## 2.6.2   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, etc.

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.

1. **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.

2. **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/OLEDB connections, Web-accessing and service facilities, reporting and OLAP
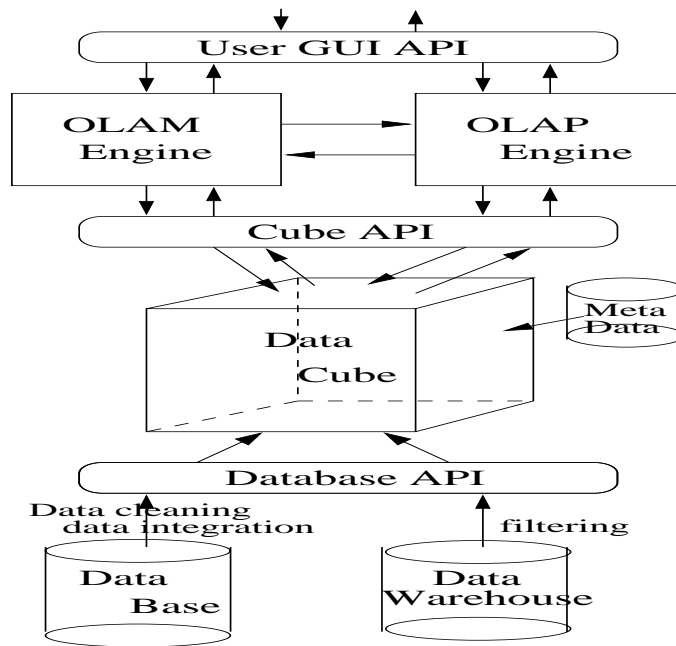
Figure 2.21: An integrated OLAM and OLAP architecture.

analysis tools. It is prudent to make the best use of the available infrastructures rather than constructing everything from scratch.

3. **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.

4. **On-line selection of data mining functions.** Often a user may not know what kinds of knowledge that she wants 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 engine performs analytical mining in data cubes in a similar manner as an OLAP engine performs on-line analytical processing. An integrated OLAM and OLAP architecture is shown in Figure 2.21, where the OLAM and OLAP engines both accept users' on-line queries (or commands) via a User_GUI_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 and/or by filtering a data warehouse via a Database_API which may support OLEDB or ODBC connections. Since an OLAM engine may perform multiple data mining tasks, such as concept description, association, classification, prediction, clustering, time-series analysis, etc., it usually consists of multiple, integrated data mining modules and is more sophisticated than an OLAP engine.

The following chapters of this book 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 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 the following chapters, we place particular emphasis on OLAP mining, that is, on the *integration of data mining and OLAP technology.*

## 2.7  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 either 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.

- **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, cross, through), slice-and-dice, pivot (rotate),* as well as statistical operations such as ranking, computing moving averages and growth rates, etc. 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.

- 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.

- A data cube consists of a **lattice of cuboids**, each corresponding to a different degree of summarization of the given multidimensional data. **Partial materialization** refers to the selective computation of a subset of the cuboids in the lattice. **Full materialization** refers to the computation of all of the cuboids in the lattice. If the cubes are implemented using MOLAP, then **multiway array aggregation** can be used. This technique "overlaps" some of the aggregation computation so that full materialization can be computed efficiently.

- OLAP query processing can be made more efficient with the use of indexing techniques. In **bitmap indexing**, each attribute has its own bimap vector. 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 methods, can be used to further speed up OLAP query processing.

- 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.

- 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.

- **Discovery-driven exploration** of data cubes uses precomputed measures and visual cues to indicate data exceptions, guiding the user in the data analysis process, at all levels of aggregation. **Multifeature cubes** compute complex queries involving multiple dependent aggregates at multiple granularities. The computation of cubes for discovery-driven exploration and of multifeature cubes can be achieved efficiently by taking advantage of efficient algorithms for standard data cube computation.

- 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.

## 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. 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.

3. What are the differences between the three typical methods for modeling multidimensional data: the *star* model, the *snowflake* model, and the *fact constellation* model? What is the difference between the *star* warehouse model and the *starnet* query model? Use an example to explain your point(s).

4. 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 which 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, and how to handle incremental data updates.

5. Data warehouse design:

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

   (b) Draw a schema diagram for a data warehouse which consists of three dimensions: *time, doctor*, and *patient*, and two measures: *count*, and *charge*, where *charge* is the fee that a doctor charges a patient for a visit.

   (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 VGH (Vancouver General Hospital) in 1997?

   (d) To obtain the same list, write an SQL query assuming the data is stored in a relational database with the schema.

   $$fee(day, month, year, doctor, hospital, patient, count, charge).$$

6. Computing 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 three dimensions: *time, location*, and *product*, 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: $\frac{1}{n}\sum_{i=1}^{n}(x_i)^2 - \bar{x_i}^2$, where $\bar{x_i}$ is the average of $x_i$'s.

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

7. Suppose that one needs 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.

8. In data warehouse technology, a multiple dimensional view can be implemented by a multidimensional database technique (MOLAP), or by a relational database technique (ROLAP), or 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?

9. Suppose that a data warehouse contains 20 dimensions each with about 5 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?

10. Data cube computation: Suppose a base cuboid has 3 dimensions, $(A, B, C)$, with the number of cells shown below: $|A| = 1,000,000$, $|B| = 100$, and $|C| = 1,000$. Suppose each dimension is partitioned evenly into 10 portions for chunking.

    (a) Assuming each dimension has only one level, draw the complete lattice of the cube.

    (b) If each cube cell stores one measure with 4 bytes, what is the total size of the computed cube if the cube is *dense*?

    (c) If the cube is very *sparse*, describe an effective multidimensional array structure to store the sparse cube.

    (d) State the order for computing the chunks in the cube which requires the least amount of space, and compute the total amount of main memory space required for computing the 2-D planes.

11. In both data warehousing and data mining, it is important to have some hierarchy information associated with each dimension. If such a hierarchy is not given, discuss how to generate such a hierarchy automatically for the following cases:

    (a) a dimension containing only numerical data.

    (b) a dimension containing only categorical data.

12. Suppose that a data cube has 2 dimensions, $(A, B)$, and each dimension can be generalized through 3 levels (with the top-most level being all). That is, starting with level $A_0$, $A$ can be generalized to $A_1$, then to $A_2$, and then to all. How many *different* cuboids (i.e., views) can be generated for this cube? Sketch a lattice of these cuboids to show how you derive your answer. Also, give a general formula for a data cube with $D$ dimensions, each starting at a base level and going up through $L$ levels, with the top-most level being all.

13. Consider the following *multifeature cube* query: Grouping by all subsets of {item, region, month}, find the minimum shelf life in 1997 for each group, and the fraction of the total sales due to tuples whose price is less than \$100, and whose shelf life is within 25% of the minimum shelf life, and within 50% of the minimum shelf life.

    (a) Draw the multifeature cube graph for the query.

    (b) Express the query in extended SQL.

    (c) Is this a *distributive* multifeature cube? Why or why not?

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)*.

## Bibliographic Notes

There are a good number of introductory level textbooks on data warehousing and OLAP technology, including Inmon [15], Kimball [16], Berson and Smith [4], and Thomsen [24]. Chaudhuri and Dayal [6] provide a general overview of data warehousing and OLAP technology.

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 [7] who coined the term *OLAP* for *on-line analytical processing*. The OLAP council was established in 1995. Widom [26] identified several research problems in data warehousing. Kimball [16] provides an overview of the deficiencies of SQL regarding the ability to support comparisons that are common in the business world.

The DMQL data mining query language was proposed by Han et al. [11] Data mining query languages are further discussed in Chapter 4. Other SQL-based languages for data mining are proposed in Imielinski, Virmani, and Abdulghani [14], Meo, Psaila, and Ceri [17], and Baralis and Psaila [3].

Gray et al. [9, 10] proposed the data cube as a relational aggregation operator generalizing group-by, crosstabs, and sub-totals. Harinarayan, Rajaraman, and Ullman [13] proposed a greedy algorithm for the partial materialization of cuboids in the computation of a data cube. Agarwal et al. [1] proposed several methods for the efficient computation of multidimensional aggregates for ROLAP servers. The chunk-based multiway array aggregation method described in Section 2.4.1 for data cube computation in MOLAP was proposed in Zhao, Deshpande, and Naughton [27]. Additional methods for the fast computation of data cubes can be found in Beyer and Ramakrishnan [5], and Ross and Srivastava [19]. Sarawagi and Stonebraker [22] developed a chunk-based computation technique for the efficient organization of large multidimensional arrays.

For work on the selection of materialized cuboids for efficient OLAP query processing, see Harinarayan, Rajaraman, and Ullman [13], and Sristava et al. [23]. Methods for cube size estimation can be found in Beyer and Ramakrishnan [5], Ross and Srivastava [19], and Deshpande et al. [8]. Agrawal, Gupta, and Sarawagi [2] proposed operations for modeling multidimensional databases.

The use of join indices to speed up relational query processing was proposed by Valduriez [25]. O'Neil and Graefe [18] proposed a bitmapped join index method to speed-up OLAP-based query processing.

There are some recent studies on the implementation of discovery-oriented data cubes for data mining. This includes the discovery-driven exploration of OLAP data cubes by Sarawagi, Agrawal, and Megiddo [21], and the construction of multifeature data cubes by Ross, Srivastava, and Chatziantoniou [20]. For a discussion of methodologies for OLAM (On-Line Analytical Mining), see Han et al. [12].

# Bibliography

[1] 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*, pages 506–521, Bombay, India, Sept. 1996.

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

[3] E. Baralis and G. Psaila. Designing templates for mining association rules. *Journal of Intelligent Information Systems*, 9:7–32, 1997.

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

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

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

[7] E. F Codd, S. B. Codd, and C. T. Salley. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. In *E. F. Codd & Associates available at http://www.arborsoft.com/OLAP.html*, 1993.

[8] 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.

[9] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational operator generalizing group-by, cross-tab and sub-totals. In *Proc. 1996 Int. Conf. Data Engineering*, pages 152–159, New Orleans, Louisiana, Feb. 1996.

[10] 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.

[11] J. Han, Y. Fu, W. Wang, J. Chiang, W. Gong, K. Koperski, D. Li, Y. Lu, A. Rajan, N. Stefanovic, B. Xia, and O. R. Zaïane. DBMiner: A system for mining knowledge in large relational databases. In *Proc. 1996 Int. Conf. Data Mining and Knowledge Discovery (KDD'96)*, pages 250–255, Portland, Oregon, August 1996.

[12] J. Han, Y. J. Tam, E. Kim, H. Zhu, and S. H. S. Chee. Methodologies for integration of data mining and on-line analytical processing in data warehouses. In *submitted to DAMI*, 1999.

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

[14] T. Imielinski, A. Virmani, and A. Abdulghani. DataMine – application programming interface and query language for KDD applications. In *Proc. 1996 Int. Conf. Data Mining and Knowledge Discovery (KDD'96)*, pages 256–261, Portland, Oregon, August 1996.

[15] W. H. Inmon. *Building the Data Warehouse*. QED Technical Publishing Group, Wellesley, Massachusetts, 1992.

[16] R. Kimball. *The Data Warehouse Toolkit*. John Wiley & Sons, New York, 1996.

[17] R. Meo, G. Psaila, and S. Ceri. A new SQL-like operator for mining association rules. In *Proc. 1996 Int. Conf. Very Large Data Bases*, pages 122–133, Bombay, India, Sept. 1996.

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

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

[20] K. A. Ross, D. Srivastava, and D. Chatziantoniou. Complex aggregation at multiple granularities. In *Proc. Int. Conf. of Extending Database Technology (EDBT'98)*, pages 263–277, Valencia, Spain, March 1998.

[21] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of OLAP data cubes. In *Proc. Int. Conf. of Extending Database Technology (EDBT'98)*, pages 168–182, Valencia, Spain, March 1998.

[22] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proc. 1994 Int. Conf. Data Engineering*, pages 328–336, Feb. 1994.

[23] 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*, pages 318–329, Bombay, India, September 1996.

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

[25] P. Valduriez. Join indices. In *ACM Trans. Database System*, volume 12, pages 218–246, 1987.

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

[27] 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*, pages 159–170, Tucson, Arizona, May 1997.