Chapter 4

Data Warehousing and Online Analytical Processing

Data Analytics, also often known as *business intelligence*, is the strategies and technologies that enable enterprises to gain deep and actionable insights into business data. Data mining plays the core role in data analytics and business intelligence. Fundamentally, *data warehouses* generalize and consolidate data in multidimensional space. The construction of data warehouses involves data cleaning, data integration and data transformation, and can be viewed as an important preparation step for data mining. Moreover, data warehouses provide *online analytical processing (OLAP)* tools for interactive analysis of multidimensional data of varied granularities, which facilitates effective data generalization and data mining. Many other data mining functions, such as association, classification, prediction and clustering, can be integrated with OLAP operations to enhance interactive mining of knowledge at multiple levels of abstraction. OLAP tools typically use *data cube*, a multidimensional data model, to provide flexible access to summarized data. Data lakes as enterprise information infrastructure collect extensive data in enterprises and integrate meta-data so that data exploration can be conducted effectively. Hence, data warehouses, OLAP, data cubes and data lakes have become essential data and information backbone for enterprises. This chapter presents an in-depth and comprehensive introduction to data warehouse, OLAP, data cube and data lake technology. This overview is essential for understanding the overall data mining and knowledge discovery process and practical applications. In addition, it can serve as a well-informed introduction to data analytics and business intelligence.

[This paragraph needs rewritten.] In this chapter, we study a well-accepted definition of the data warehouse and see why more and more organizations are building data warehouses for analyzing their data (Section 4.1). In particular, we study *data cube*, a multidimensional data model for data warehouses and OLAP, as well as OLAP operations such as roll-up, drill-down, slicing, and dicing (Section 4.2). We also look at data warehouse design and usage (Section **??**). In addition, we discuss *multidimensional data mining*, a powerful paradigm that integrates data warehouse and OLAP technology with that of data mining. An overview of data warehouse implementation examines general strategies for efficient data cube computation, OLAP data indexing, and OLAP query processing (Section 4.4). Finally, we study data generalization by attribute-oriented induction (Section **??**). This method uses concept hierarchies to generalize data to multiple levels of abstraction.

4.1 Data Warehouses

[Need to update] This section introduces data warehouses. We begin with a definition of data warehouses, and explain how data warehouses can serve as the foundation of business intelligence (Section 4.1.1). Next, we discuss data warehouse architecture (Section 4.1.2). Last, we study three data warehouse models—an enterprise model, a data mart and a virtual warehouse (Section 4.1.2).

4.1.1 Data Warehouses: What and Why?

More often than not, data in organizations are recorded at the operational level. For example, for the sake of business efficiency, an e-commerce company often records the details of customer transactions in a table, the information about customers in another table, and the particulars about product suppliers in a third table. Operational data are mainly concerned about individual business functionings, such as a purchase transaction, registration of a new customer and the shipment of a batch of products to a store. The major advantage is that business operations, such as a customer purchasing a product, can be conducted efficiently by inserting, deleting or modifying only one or several records in one or a small number of tables, and thus many business operations can be conducted concurrently.

At the same time, business analysts and executives often focus on historical, current and predictive views of business operations instead of individual transaction details. For example, a business analyst in an e-commerce company may want to investigate the categories of customers, such as their demographical groups, who spend the most last month, and the major categories of products they purchase. Computing answers to such analytic questions is often time and resource consuming, since it has to join multiple data tables and conduct a large number of group-by aggregation operations, and thus needs exclusive access to the data. Many analysis tasks may be periodic and some may be ad hoc, and thus may severely affect business operations, which are expected to be online, frequent and concurrent.

To address the gap between business operations and analysis, data warehousing provides architectures and tools for business analysts and executives to systematically organize, understand, and use their data to make strategic decisions. Data warehouse systems are valuable tools in today's competitive, fast-evolving world. In the last two decades, many firms have spent billions of dollars in building enterprise-wide data warehouses. It is well recognized that, with competition mounting in every industry, data warehousing is the must-have business infrastructure—a way to retain customers by learning more about their demands and behavior.

"Then, what exactly is a data warehouse?" In general, a data warehouse refers to a data repository that is specific for analysis and is maintained separately from an organization's operational databases. Data warehouse systems support information processing by providing a solid platform of consolidated historic data for analysis.

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

- **Subject-oriented**: A data warehouse is organized around major subjects that are often identified enterprise or department wise, such as customer, supplier, product, and sales. Rather than concentrating on the day-to-day operations and transaction processing of an organization, a data warehouse focuses on modeling and analyzing data for decision makers. Hence, data warehouses typically provide a simple and concise view of 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 online 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 is stored to provide information from a historic perspective (e.g., the past 5–10 years). Every key structure in a data warehouse contains, either implicitly or explicitly, a time element.

- **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 strong transaction processing, recovery, and concurrency control mechanisms, and thus has no interference with the operational systems. 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. It stores the information that 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. Correspondingly, **data warehousing** is the process of constructing and using data warehouses. The construction of a data warehouse requires data cleaning, data integration and data consolidation.

"*How do organizations use information from data warehouses?*" Many organizations use this information to support business decision-making activities. For example, by identifying the groups of most active customers an e-commerce company can design promotion campaigns to retain those customers firmly. By analyzing the sales patterns of products in different seasons, a company may design supply chain strategies to reduce the stocking cost of seasonal products. Analytic results from data warehouses are often presented to analysts and decision makers through periodic or ad hoc reports, such as daily, weekly and monthly sales analysis reports analyzing sales patterns on customer groups, regions, products and promotions.

"*What are the major differences between operational database systems and data warehouses?*" The major task of traditional operational database systems is to perform **online transaction processing (OLTP)**. These OLTP systems cover most of the day-to-day operations of an organization, such as purchasing, inventory, manufacturing, banking, payroll, registration and accounting. Data warehouse systems serve business analysts and executives (in general, also known as *knowledge workers*) in the role of obtaining business insights and making decisions by organizing and presenting data in various perspectives in order to accommodate the diverse needs from different users. These systems are known as **online analytical processing (OLAP)** systems.

The major distinguishing features of OLTP and OLAP are as follows:

- **Users and system orientation**: An OLTP system is *transaction-oriented* and is used for operation execution by clerks and clients. An OLAP system is *business insight-oriented* and is used for data summarization and analysis by knowledge workers, including managers, executives, and analysts.

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

- **Database design**: An OLTP system usually adopts an entity-relationship (ER) data model and an application-oriented database design. An OLAP system typically adopts either a *star* model or a *snowflake* model (see Section 4.2.2) and a subject-oriented database design.

- **View**: An OLTP system focuses mainly on the current data within an enterprise or department, without referring to historic 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.

- **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 (because most data warehouses store historic rather than up-to-date information), although many may be complex queries.

"Why not perform OLAP directly on operational databases instead of constructing a separate data warehouse?" A major reason for a separation is to ensure the *high performance of both systems*. An operational database is designed and tuned from known tasks and workloads like indexing and hashing using primary keys, searching for particular records, and optimizing "canned" queries, which are pre-programmed and frequently used queries in business. OLAP queries, however, are often complex. They involve the computation of large data groups at summarized levels, and may require the use of special data organization, access, and implementation methods based on multidimensional views. Processing OLAP queries directly in operational databases may substantially jeopardize the performance of operational tasks. An operational database supports the concurrent processing of multiple transactions. Concurrency control and recovery mechanisms (e.g., locking and logging) are required to ensure the consistency and robustness of transactions. An OLAP query often needs read-only access of massive data records for summarization and aggregation. Concurrency control and recovery mechanisms, if applied for such OLAP operations, may seriously delay the execution of concurrent transactions and thus substantially reduce the throughput of an OLTP system.

Finally, the separation of operational databases from data warehouses is based on the different structures, contents and uses of the data in these two kinds of systems. Decision support requires historic data, whereas operational databases do not typically maintain historic data. In this context, the data in operational databases are usually far from complete for decision making. Decision support requires consolidation (e.g., aggregation and summarization) of data from heterogeneous sources, resulting in high-quality, clean and integrated data. In contrast, operational databases contain only detailed raw data, such as transactions, which need to be consolidated before analysis. Because the two systems provide quite different functionalities and require different kinds of data, it is presently necessary to maintain separate databases.

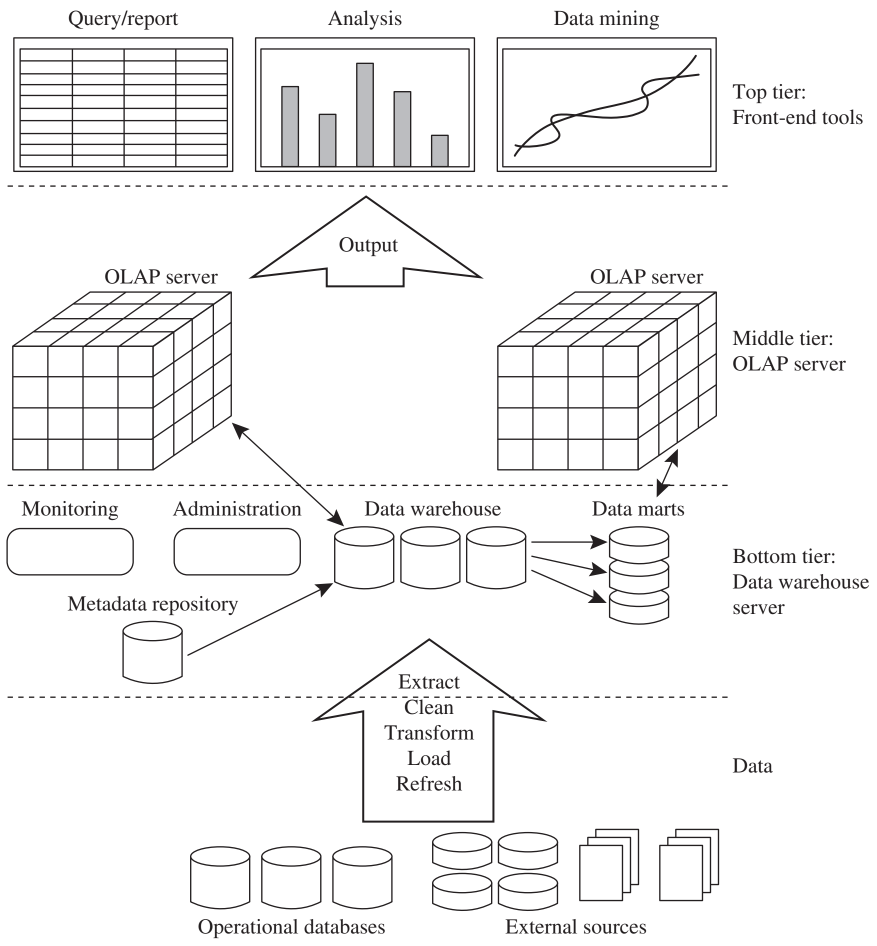


Figure 4.1: A three-tier data warehousing architecture.

4.1.2 Architecture of Data Warehouses: Enterprise Data Warehouses and Data Marts

"What does the architecture of a data warehouse look like?" To answer this question, we first introduce the general three-tier architecture of data warehouses, and then discuss two major data warehouse models, the enterprise warehouse and the data mart.

The Three-tier Architecture

Data warehouses often adopt a three-tier architecture, as presented in Figure 4.1.

The bottom level is a **warehouse database server** that is typically a mainstream database system, such as a relational database or a key-value store.

Back-end tools and utilities are used to feed data into the bottom tier from operational databases or other external sources (e.g., customer profile information provided by external partners). These tools and utilities perform data extraction, cleaning, and transformation, as well as load and refresh functions to update the data warehouse. This tier also contains a metadata repository, which stores information about the data warehouse and its contents.

The middle tier is an **OLAP server** that is typically implemented using either a **relational OLAP (ROLAP)** model (i.e., an extended relational DBMS that maps operations on multidimensional data to standard relational operations) or a **multidimensional OLAP (MOLAP)** model (i.e., a special-purpose server that directly implements multidimensional data and operations). We will discuss OLAP servers in detail soon.

The top tier is a **front-end client layer**, which contains tools for querying, reporting, visualization, analysis, and/or data mining, such as trend analysis and prediction.

"*What is meta data in a warehouse database server?*" **Metadata** is data about data. When used in a data warehouse, metadata is the data that defines warehouse objects. Metadata is created for the data names and definitions of the given warehouse. Additional metadata may be 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.

In addition, a metadata repository may contain a description of the *data warehouse structure* (e.g., the schema, view, dimensions, derived data definition, etc.), *operational metadata* (e.g., data transformation lineage, freshness of data), definitions of data summarization, mapping from operational data to the data warehouse, system information and related business information.

Metadata plays a very different role than other data warehouse data and is important for many reasons. For example, metadata are used as a directory to help analysts locate the contents of a data warehouse, and as a guide to data mapping when data are transformed from the operational environment to the data warehouse environment. Metadata also serves 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).

Data warehouse systems use back-end tools and utilities to populate and refresh their data (Figure 4.1). These tools and utilities include the functions of **data extraction** (gathering data from multiple, heterogeneous and external sources), **data cleaning** (detecting errors in the data and rectifying them when possible), **data transformation** (converting data from legacy or host format to warehouse format), **loading** (sorting, summarizing, consolidating, computing views, checking integrity and building indices and partitions) and **refresh** (propagating the updates from the data sources to the warehouse). In addition, data warehouse systems usually provide a good set of data warehouse management tools.

Enterprise Data Warehouse and Data Mart

From the architecture point of view, there are two major data warehouse models, namely the *enterprise warehouse* and the *data mart*.

Enterprise warehouse: An enterprise warehouse collects all 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 hundreds of gigabytes to terabytes or beyond. It requires extensive business modeling at the enterprise level 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, such as those within a business department. The scope is confined to specific selected subjects. For example, a marketing data mart may confine its subjects to customer, item, marketing channel and sales. A risk control data mart may focus on customer credit, risk, and different types of frauds. The data contained in data marts tend to be summarized. 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 are not enterprise-wide.

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

Some circumstances also employ a *virtual warehouse*, which 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 put excess overhead on operational database servers.

"It is often said that enterprises use artificial intelligence (AI for short) more and more in business and data is a foundation of AI. What is the relationship between data warehouses and AI?" In general, data warehouses can support deployment of AI and machine learning functionalities. At the same time, artificial intelligence and machine learning tools can be used on top of data warehouses to take the best advantage of data warehouses.

Artificial intelligence refers to various computer systems that can conduct tasks that normally need human intelligence, such as playing board games, automatic driving and dialog with humans. Machine learning, one of the core technologies in AI, is to build computer systems that can learn without being explicitly programmed the specific instructions. Many machine learning techniques are used by data mining, such as classification and clustering, which will be discussed in detail later in this book.

Artificial intelligence and machine learning tools need to consume considerable amounts of data to build various models for sophisticated tasks. Data warehouses organize and summarize data at proper levels and thus can support deployment of AI and machine learning functionalities. For example, an e-commerce company may want to build an AI model to categorize customers into different groups for better customer-relation management. This sophisticated task can be substantially benefited from a data mart of customer information, which can provide cleaned, integrated and summarized data about customers.

At the same time, AI and machine learning techniques are widely used in various steps of data warehousing. For example, machine learning techniques can be used in constructing data warehouses, such as filling in missing values and identifying entities in data cleaning ([Check against Chapter 3] see Chapter 3). Moreover, the output from AI models may be included in a data warehouse. For example, a data mart of customer information may likely include customer profiles, where customers and customer groups are often labeled based on their behavior, such as age groups, income levels, and consumption preferences. Those labels are often predicted by machine learning models trained from customer data. Third, AI and machine learning techniques can be used to optimize data warehouse performance. For example, machine learning techniques can be used to tune up the performance of data indexing and task execution in data warehouses distributed in large data centers, and also can help to lower down power consumption substantially. Last but not least, AI and machine learning techniques are essential for knowledge workers to explore and understand data in data warehouses and make well informed decisions. For example, an analyst can build machine learning models to explore the relation between business growth rates in different regions and marketing cost associated. More examples will be given in the later part of this book.

4.1.3 OLAP Server Architectures: ROLAP versus MOLAP versus HOLAP

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

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

ROLAP technology tends to have greater scalability than MOLAP technology. The DSS server of Microstrategy, for example, adopts the ROLAP approach.

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

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

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

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

"*How are data actually stored in ROLAP and MOLAP architectures?*" Let's first look at ROLAP. As its name implies, ROLAP uses relational tables to store data for online 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

Table 4.1: Single Table for Base and Summary Facts

RID	*item*	...	*day*	*month*	*quarter*	*year*	*dollars_sold*
1001	TV	...	15	10	Q4	2010	250.60
1002	TV	...	23	10	Q4	2010	175.00
...	...	...	...	...	...	...	...
5001	TV	...	`all`	10	Q4	2010	45,786.08
...	...	...	...	...	...	...	...

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

Example 4.1 **A ROLAP data store.** Table 4.1 shows a summary fact table that contains both base fact data and aggregated data. The schema is "⟨*record_identifier (RID), item, . . . , day, month, quarter, year, dollars_sold*⟩," where *day, month, quarter*, and *year* define the sales date, and *dollars_sold* is the sales amount. Consider the tuples with an *RID* of 1001 and 1002, respectively. The data of these tuples are at the base fact level, where the sales dates are October 15, 2010, and October 23, 2010, respectively. Consider the tuple with an *RID* of 5001. This tuple is at a more general level of abstraction than the tuples 1001 and 1002. The *day* value has been generalized to `all`, so that the corresponding *time* value is October 2010. That is, the *dollars_sold* amount shown is an aggregation representing the entire month of October 2010, rather than just October 15 or 23, 2010. The special value `all` is used to represent subtotals in summarized data.

MOLAP uses multidimensional array structures to store data for online analytical processing.

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.

4.1.4 Data Lakes

"In some organizations, people mention 'data lakes'. What are data lakes and what are the relations and differences between data lakes and data warehouses?" In a big organization, there are often a massive number of complicated data sources with a wild variety in data types, formats, and quality, such as business data in relational databases, communication records between customers and the organization, regulations, market analysis and external market information. Many data exploration analyses are one-time and may have to use data from different corners. It may take a long time to design and develop a data warehouse, where data are integrated, transformed, structured and loaded according to defined usage. Moreover, many data-driven explorations have to be self-service business intelligence so that data scientists can analyze and explore data by themselves. To address the vast data usage demands in the organization, as an alternative, a data lake may be built.

Conceptually, a *data lake* is a single repository of all enterprise data in the natural format, such as relational data, semi-structured data (e.g., XML, CSV, JSON), unstructured data (e.g., emails, PDF files), and even binary data (e.g., images, audio, video). More often than not, a data lake takes the form of object blocks or files, and is hosted using a cloud-based or distributed data repository. A data lake often stores both raw data copies as well as transformed data. Many analytical tasks, such as reporting, visualization, analytics and data mining, can be conducted on data lakes.

"What are the essential differences between data warehouses and data lakes?" First, to build a data warehouse, one has to analyze the data sources, understand the business processes, and develop the corresponding data models. The subjects in data warehouses reflect the factors in the corresponding business analysis and decision making processes. In contrast, a data lake retains all data in an organization, including the current and historical data, as well as data being used now and not used at this time. The rationale is that the data lake as the complete repository can be used as the base of all data related tasks now and in the future.

Second, a data warehouse typically stores data extracted from transactional data, including quantitative metrics and attribute values, and does not cover much non-relational data, such as text, images and video. Data are loaded to data warehouses according to predefined schemas. In contrast, a data lake natively embraces all data types. Data are transformed when it is used.

Third, a data warehouse is designed for data analysts and executives. The queries on a data warehouse are typically supporting decision making. In contrast, since a data lake includes all data in the natural form, it can support all users in an organization, including operational users, analysts and executives.

Fourth, the well designed structures in a data warehouse provide high quality support to target analytical tasks. However, for new queries or business changes that are not covered by the data warehouse design, it takes time to upgrade the data warehouse to address the new demands, which is the major pain-point in data warehousing. In contrast, a data lake stores all data in the raw form and thus is always available for exploring any novel usages. Data scientists can directly work on data lakes to conduct data analysis. The analysis results may also become a part of the data lake.

Last, since building a data warehouse takes time and resource, a data warehouse typically cannot cover all business and analytic users in an organization. For those business and users not supported by a data warehouse, they still can use a data lake to obtain faster insights.

Data warehouses and data lakes represent two views on data analytics. Data warehouses are more top-down, structured and centralized. In contrast, data lakes are more bottom-up, quick prototyping and democratic. In enterprise practice, a combination is often exercised to harvest the best gain.

"As a data lake has to store all enterprise data, which are often huge in size and diverse in type and format, how are data in a data lake stored and organized?" Typically data lakes have a core storage layer, which stores raw and/or lightly processed data. There are several important considerations in designing and implementing the data lake storage. First, since data lakes are served as the centralized data repository for an entire enterprise, the data storage has to be exceptionally scalable. Second, as data lakes have to respond to a wide variety of queries and analytic tasks, data robustness is critical. Consequently, the data storage layer has to have high durability. In other words, the data stored in a data lake should be intact and pristine all the time. Third, to address the diversity of data in enterprises, data lake storage has to support different types of data in various format, including structured data, semi-structured data

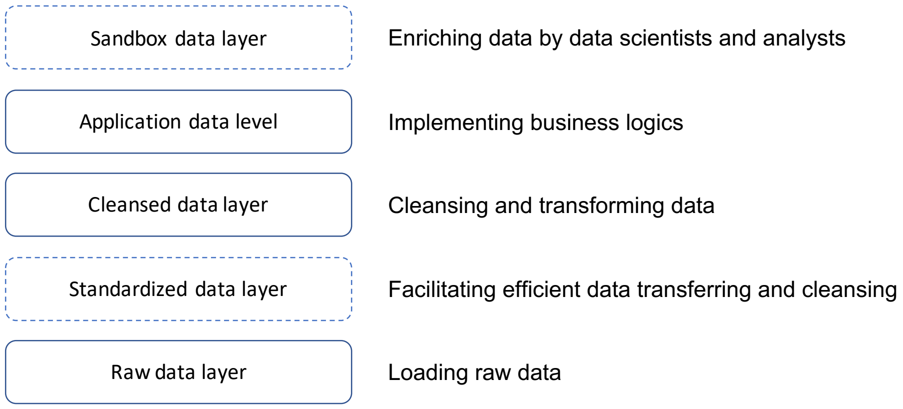

Sandbox data layer
Enriching data by data scientists and analysts
Application data level
Implementing business logics
Cleansed data layer
Cleansing and transforming data
Standardized data layer
Facilitating efficient data transferring and cleansing
Raw data layer
Loading raw data

Figure 4.2: The layers of data storage in data lakes.

and unstructured data. All such data have to stored and managed consistently and harmonically in the same repository. Fourth, as data lakes are use to support different kinds of queries, analyses and applications, the data storage should be able to support various data schemas, many of which may not be known or available when data lakes are designed. In other words, the data lake storage must be independent from any fixed schema. Last, in contrast to many applications where data and computation are corporate, the storage layer of data lakes should be decoupled with computation resources, so that various computation resources, ranging from legacy mainframe servers to clouds, can access data in data lakes. This separation can allow the maximum scalability in both data lakes and applications supported by data lakes.

Conceptually, a data lake has a storage layer as a single repository. In implementation, the data repository is still divided into multiple layers. Typically, the repository has three mandatory layers: raw data, cleansed data and application data. Optionally, a standardized data layer and a sandbox layer may be added. Let us explain the layers bottom up. Figure 4.2 summarizes the layers.

The raw data layer is the lowest layer, and is also known as the ingestion layer or the landing area. At this layer, raw data are loaded in the native format. No data processing is conducted, such as cleansing, duplicate removing or data transforming. Data are typically organized into folders by areas, data sources, objects and time of ingestion. The data at this level are not ready for use yet, and thus end users of data lakes should not be allowed to access to the raw data layer.

Optionally, a data lake may have a standardized data layer on top of the raw data layer. The main objective of the standardized data layer is to facilitate high performance in data transferring and cleansing. For example, in the raw data layer, data are stored in its native format. In the standardized data layer. data may be transformed into some formats that are best for cleansing. Moreover, data may be divided into structures of finer grain for more efficient access and

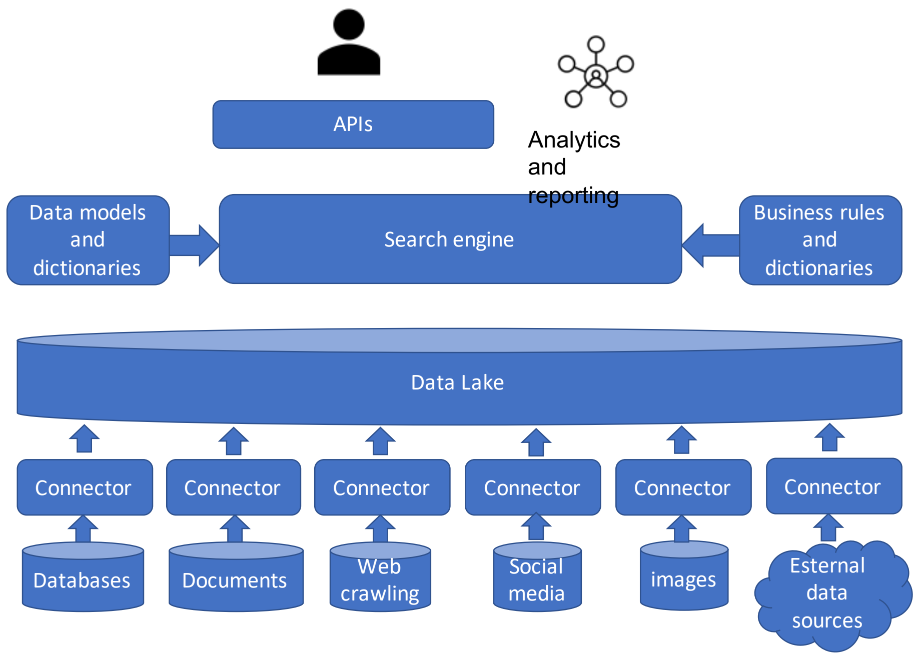


Figure 4.3: The conceptual architecture of data lakes.

processing.

The next layer is the cleansed data layer, also known as the curated layer or the conformed layer. At this layer, data are cleansed and transformed, such as being denormalized or consolidated. Moreover, data are organized into data sets and stored into tables or files. End users of data lakes are allowed to access data at this layer.

On top of the cleansed data layer is the application data layer, also known as the trusted layer, the secure layer or the production layer. Business logics are implemented at this layer. Therefore, many applications, including those data mining and machine learning applications, can be built based on this layer.

In some organizations, data scientists and analysts may conduct experiments and find patterns and correlations. Their projects may enrich the data substantially and thus create new data. Such data may be stored at the optional sandbox data layer.

"Centred by the data lake storage, what is the architecture of data lakes? What are the other important components in data lakes in addition to data storage?" Figure 4.3 shows the conceptual architecture of data lakes. A data lake takes data from a wide spectrum of data repositories in an enterprise or organization, such as the databases, the store of documents, data crawled from the web, social media, images (e.g., products) and possibly external data sources. Data from those data sources are loaded into the data lake through connectors

in a continuous manner. Once data are ingested into a data lake, the data go through the layers that we just discussed.

Data lakes serve as the centralized data repositories of enterprises and organizations. End users, such as analysts and data scientists, can access data sets in data lakes, at the cleansed data layer those upper layers. A major type of access is to discover the data sets that can be used to fulfill analytic tasks. These "data discovery" tasks are conducted through an enterprise search engine. For example, a data scientist designing a marketing campaign may want to find all data related to customers in industry section "electronics manufacturing". Through the search engine, the data scientist may find data sets like purchase transactions from the operational databases, communication documents with those related customers, product categories of those customers crawled from the web, product reviews from social media, product images and product availability data provided by those customers as external data. Clearly, without a data lake as a centralized data repository, the data scientist may have to spend a lot of time to find such data scattered in different departments of the enterprise and obtain access to those data sets. In order to facilitate better utility of data in data lakes, data models and dictionaries and business rules and dictionaries are employed as domain business knowledge bases for the enterprise search engine so that the search of data sets is business oriented instead of technical oriented. Last, many applications can be built on top of the data services provided by data lakes through the corresponding APIs. Regular analytics and reporting services can also be developed and maintained accordingly.

Data lakes as centralized data repositories in enterprises bring in huge efficiency and advantage in data-driven business operation and decision making. At the same time, data lakes also post grand challenges in management and administration. In addition to the data storage layer, data lakes also need to address a series of important aspects. Among others, security is a central piece. Access to data lakes should be properly defined and assigned to the right people for the right periods. Data stored in data lakes should be protected properly. Authentication, accountability, authorization and data protection should be held consistently and comprehensively. In order to ensure security and tune for high performance, data lakes should be under systematic governance. For example, monitoring, logging and lineage should be conducted regularly. Availability, usability, security and integrity of data lakes should be monitored and managed all the time. In addition, data quality, data auditing, archives and stewardship are some other important aspects in data lakes.

4.2 Data Warehouse Modeling: Schema and Measures

As discussed in the last section, a data warehouse integrates historical and current data in a subject-oriented and non-volatile manner. The data models used in data warehouses organize data according to subjects. Here, a subject, such as customers, is captured by dimensions, such as gender age group, and occupation, and measures, such as total purchase and average transaction amount. Naturally, data warehouses and OLAP tools are based on **multidimensional data models**, which view data in the form a *data cube.* In this section, you will learn how data cubes model n-dimensional data (Section 4.2.1). In Section 4.2.2, various multidimensional models are explained: star schema, snowflake schema, and fact constellation. You will also learn about different categories of measures and how they can be computed efficiently (Section 4.2.3).

4.2.1 Data Cube: A Multidimensional Data Model

"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.

A multidimensional data model is typically organized around a central theme, also known as a subject, such as sales. The information about a subject can be divided into two parts in analysis. The first part is the perspectives that the subject is to be analyzed. For example, for subject sales in a company, the possible perspectives include time, item, branch, and location. Those perspectives are modeled as **dimensions**. In the simplest multidimensional data model, a dimension table can be built for each dimension. For example, a dimension table for *item* may contain the attributes *item_name, brand* and *type.*

The second part is the measurements on a subject. Those measurements are called **facts**. For example, for subject sales in a company, the facts may be *dollars_sold* (sales amount in dollars), *units_sold* (number of units sold), and *amount_budgeted.* Facts are typically numerical, but still may take some other data types, such as categorical data or text.

In a data warehouse, a **fact table** stores the names of the *facts*, or measures, as well as keys to each of the related dimension tables.

In general, a data cube can have as many dimensions as the business needs, and thus is n-dimensional. To elaborate data cubes and the multidimensional data model, let us start by looking at a simple 2-D data cube that is, in fact, a table or spreadsheet for sales data for a company. In particular, we will look at the sales data for items sold per quarter in a city, say Vancouver. The data is shown in Table 4.2. In this 2-D representation, the sales for Vancouver are shown with respect to the *time* dimension (organized in quarters) and the *item* dimension (organized according to the types of items sold). The fact or measure displayed is *dollars_sold* (in thousands).

Now, suppose that we would like to view the sales data with a third dimension. For instance, suppose we would like to view the data according to *time* and *item*, as well as *location*, for the cities Chicago, New York, Toronto, and Vancouver. These 3-D data are shown in Table 4.3. The 3-D data in the table 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 4.4.

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

Table 4.2: 2-D View of Sales Data According to *time* and *item*
location = *"Vancouver"*

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

Note: The sales are from branches located in the city of Vancouver. The measure displayed is *dollars_sold* (in thousands).

Table 4.3: 3-D View of Sales Data According to *time*, *item*, and *location*

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

Note: The measure displayed is *dollars_sold* (in thousands).

Figure 4.5. If we continue in this way, we may display any n-dimensional data as a series of $(n-1)$-dimensional "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.

Tables 4.2 and 4.3 show the data at different degrees of summarization. In the data warehousing research literature, a data cube like those shown in Figures 4.4 and 4.5 is often referred to as a **cuboid**. Given a set of dimensions, we can generate a cuboid for each of the possible subsets of the given dimensions. The result would form a *lattice* of cuboids, each showing the data at a different level of summarization, or `group-by`. The lattice of cuboids is then referred to as a data cube. Figure 4.6 shows a lattice of cuboids forming a data cube for the dimensions *time*, *item*, *location*, and *supplier*.

The cuboid that holds the lowest level of summarization is called the **base cuboid**. For example, the 4-D cuboid in Figure 4.5 is the base cuboid for the given *time*, *item*, *location*, and *supplier* dimensions. Figure 4.4 is a 3-D

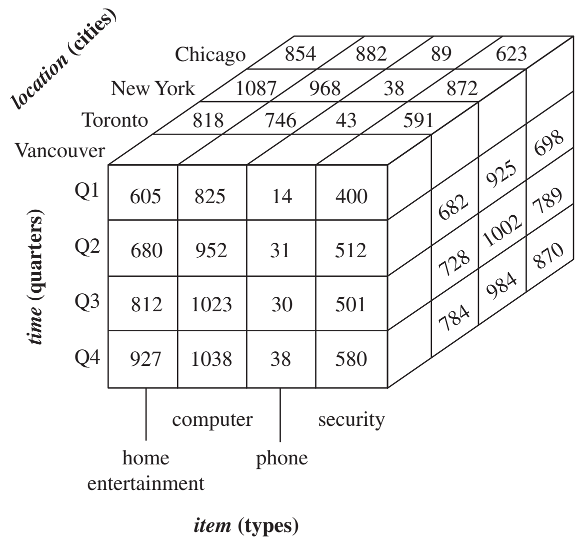


Figure 4.4: A 3-D data cube representation of the data in4.3 Table , according to *time*, *item*, and *location*. The measure displayed is *dollars_sold* (in thousands).

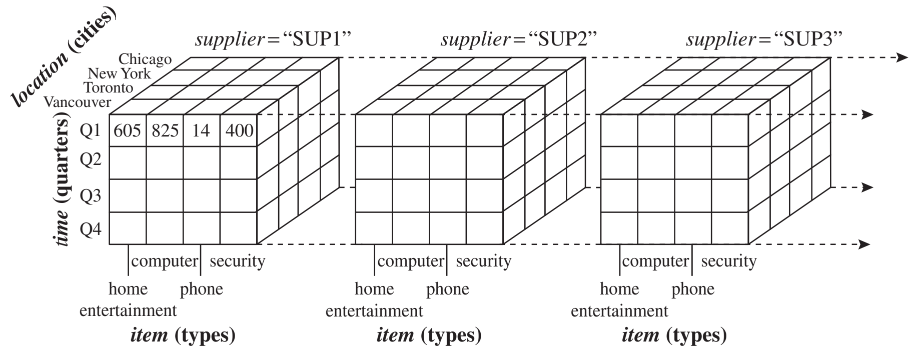


Figure 4.5: A 4-D data cube representation of sales data, according to *time*, *item*, *location*, and *supplier*. The measure displayed is *dollars_sold* (in thousands). For improved readability, only some of the cube values are shown.

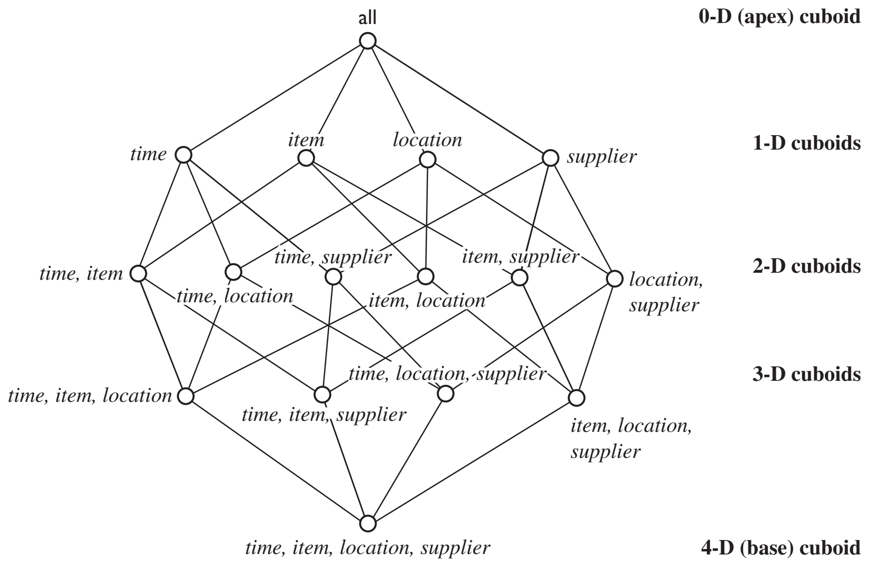


Figure 4.6: Lattice of cuboids, making up a 4-D data cube for *time*, *item*, *location*, and *supplier*. Each cuboid represents a different degree of summarization.

(nonbase) cuboid for *time*, *item*, and *location*, summarized for all suppliers. The 0-D cuboid, which holds the highest level of summarization, is called the **apex cuboid**. In our example, this is the total sales, or *dollars_sold*, summarized over all four dimensions. The apex cuboid is typically denoted by `all`.

4.2.2 Schemas for Multidimensional Data Models: Stars, Snowflakes, and Fact Constellations

The entity-relationship data model is commonly used in the design of relational databases, where a database schema consists of a set of entities and the relationships among them. Normalization is conducted to break a wide table into narrower tables so that many transactional operations only have to access very few records in one or a small number of tables, and thus concurrency of transactional operations can be maximized. Such a data model is appropriate for online transaction processing. An online data analysis often has to scan a lot of data. To support online data analysis, a data warehouse requires a concise, subject-oriented schema that facilitates scanning a large amount of data efficiently.

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

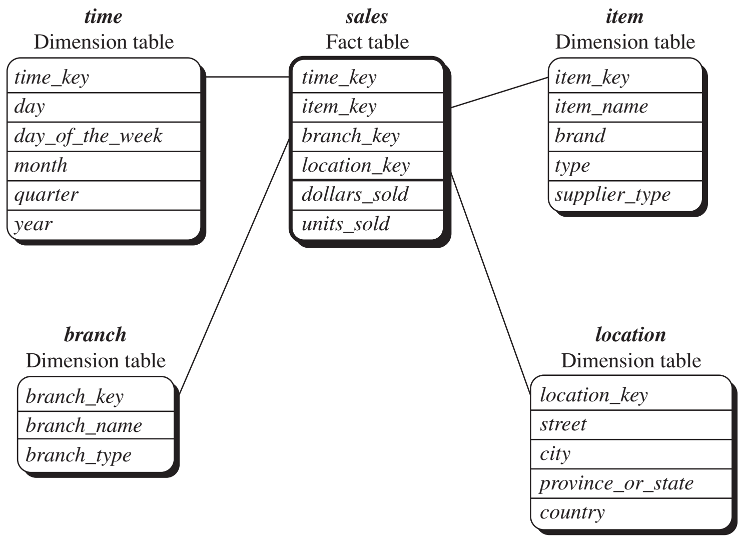


Figure 4.7: Star schema of *sales* data warehouse.

radial pattern around the central fact table.

Example 4.2 [**Star schema.**] A star schema for sales is shown in Figure 4.7. Sales are considered along four dimensions: *time, item, branch*, and *location*. The schema contains a central fact table for *sales* that contains the keys to each of the four dimensions, along with two measures: *dollars_sold* and *units_sold*. To minimize the size of the fact table, dimension identifiers (e.g., *time_key* and *item_key*) are system-generated identifiers.

Notice that in the star schema, each dimension is represented by only one table, and each table contains a set of attributes. For example, the *location* dimension table contains the attribute set {*location_key, street, city, province_or_state, country*}. This constraint may introduce some redundancy. For example, "Urbana" and "Chicago" are both cities in the state of Illinois, USA. Entries for such cities in the *location* dimension table will create redundancy among the attributes *province_or_state* and *country*; that is, (..., Urbana, IL, USA) and (..., Chicago, IL, USA).

Snowflake schema is a variant of star schema, 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 to reduce redundancies. Such a table is easy to maintain and saves storage space. However, this space savings is negligible in comparison to the typical magnitude of the fact table. Furthermore, the snowflake structure may reduce

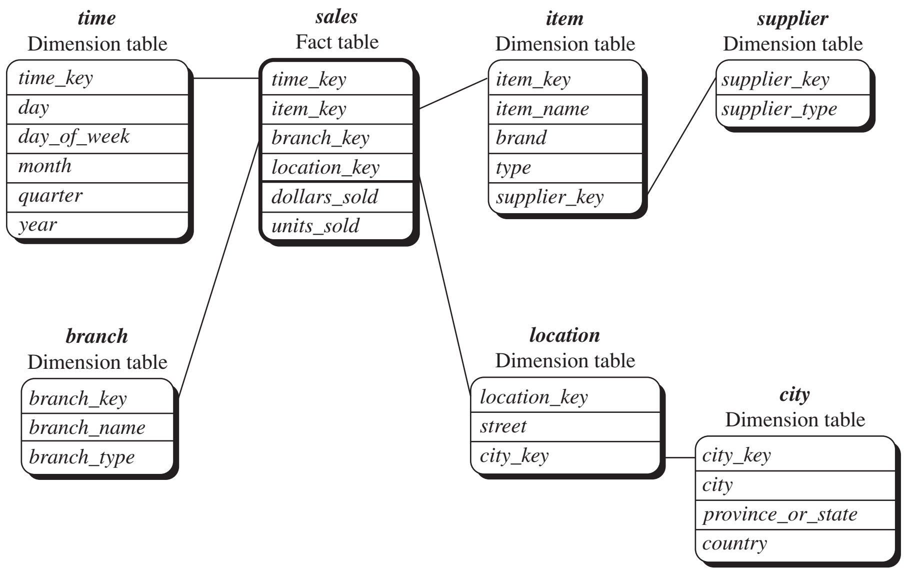


Figure 4.8: Snowflake schema of a *sales* data warehouse.

the effectiveness of browsing, since more joins will be needed to execute a query. Consequently, the system performance may be adversely impacted. Hence, although the snowflake schema reduces redundancy, it is not as popular as the star schema in data warehouse design.

Example 4.3 [**Snowflake schema.**] A snowflake schema for sales is given in Figure 4.8. Here, the *sales* fact table is identical to that of the star schema in Figure 4.7. The main difference between the two schemas is in the definition of dimension tables. The single dimension table for *item* in the star schema is normalized in the snowflake schema, resulting in new *item* and *supplier* tables. For example, the *item* dimension table now contains the attributes *item_key, item_name, brand, type*, and *supplier_key*, where *supplier_key* is linked to the *supplier* dimension table, containing *supplier_key* and *supplier_type* information. Similarly, the single dimension table for *location* in the star schema can be normalized into two new tables: *location* and *city.* The *city_key* in the new *location* table links to the *city* dimension. Notice that, when desirable, further normalization can be performed on *province_or_state* and *country* in the snowflake schema shown in Figure 4.8.

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

Example 4.4 [**Fact constellation.**] A fact constellation schema is shown in Figure 4.9. This schema specifies two fact tables, *sales* and *shipping.* The *sales* table definition

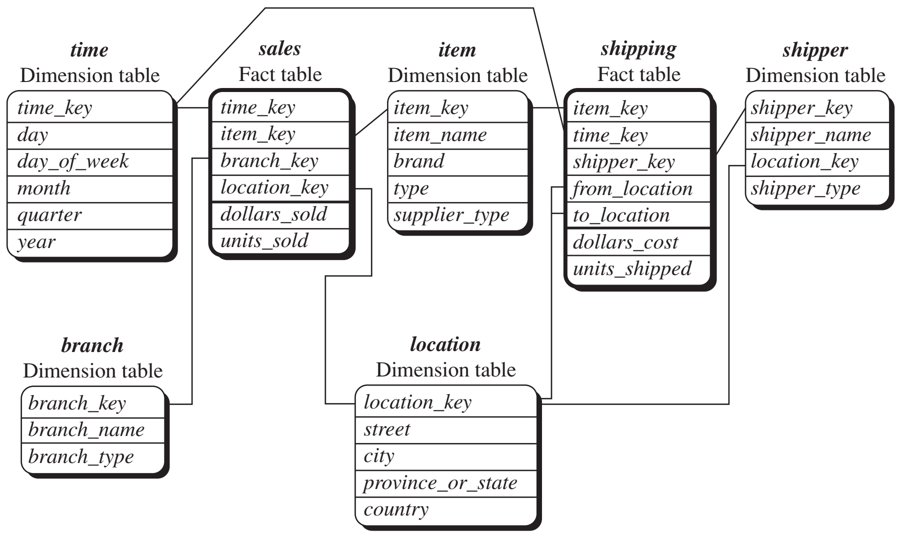


Figure 4.9: Fact constellation schema of a sales and shipping data warehouse.

is identical to that of the star schema (Figure 4.7). The *shipping* table has five dimensions, or keys—*item_key, time_key, shipper_key, from_location*, and *to_location*—and two measures—*dollars_cost* and *units_shipped*. A fact constellation schema allows dimension tables to be shared between fact tables. For example, the dimensions tables for *time, item*, and *location* are shared between the *sales* and *shipping* fact tables.

Dimensions define concept hierarchies. A **concept hierarchy** defines a sequence of mappings from a set of low-level concepts to higher-level, more general concepts. Consider a concept hierarchy for the dimension *location*. City values for *location* include Vancouver, Toronto, New York, and Chicago. Each city, however, can be mapped to the province or state to which it belongs. For example, Vancouver can be mapped to British Columbia, and Chicago to Illinois. The provinces and states can in turn be mapped to the country (e.g., Canada or the United States) to which they belong. 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). This concept hierarchy is illustrated in Figure 4.10.

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, zip_code*, 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 4.11(a). Alternatively, the attributes of a dimension may be organized in a partial order, forming a acyclic directed graph. An example of a partial order for the *time* dimension based on the attributes *day, week, month, quarter*, and *year* is "*day*

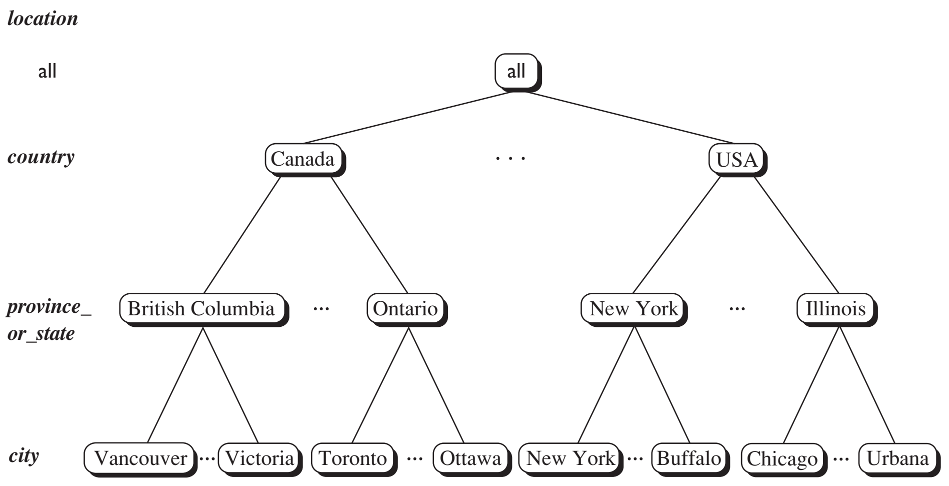


Figure 4.10: A concept hierarchy for *location*. Due to space limitations, not all of the hierarchy nodes are shown, indicated by ellipses between nodes.

$<$ {*month* $<$ *quarter; week*} $<$ *year*."[1] This partial order structure is shown in Figure 4.11(b). A concept hierarchy that is a total or partial order among attributes in a database schema is called a **schema hierarchy**. Concept hierarchies that are common to many applications (e.g., *for time*) may be predefined in the data mining system. Data mining systems should provide users with the flexibility to tailor predefined hierarchies according to their particular needs. For example, users may want 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 4.12 for the dimension *price*, where an interval $(\$X \ldots \$Y]$ denotes the range from $\$X$ (exclusive) to $\$Y$ (inclusive).

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

Concept hierarchies may be provided manually by system users, domain experts or knowledge engineers, or may be automatically generated based on statistical analysis of the data distribution. Concept hierarchies allow data to be handled at varying levels of abstraction, as we will see in Section 4.2.3.

[1]Since a *week* often 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.

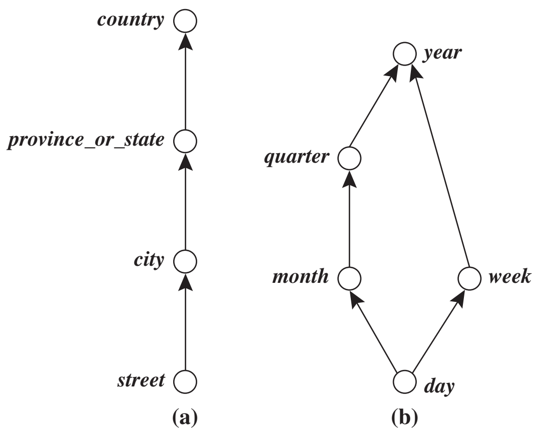


Figure 4.11: Hierarchical and lattice structures of attributes in warehouse dimensions: (a) a hierarchy for *location* and (b) a lattice for *time*.

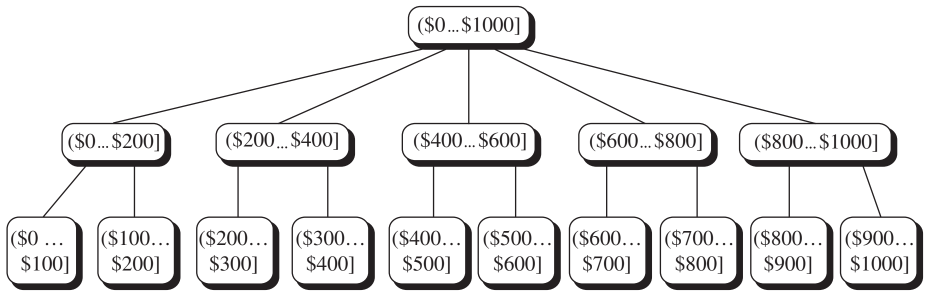


Figure 4.12: A concept hierarchy for *price*.

4.2.3 Measures: Categorization and Computation

"How are measures computed?" To answer this question, we first study how measures can be categorized. Note that a *multidimensional point* in the data cube space can be defined by a set of dimension–value pairs; for example, $\langle$*time* = "Q1", *location* = "Vancouver", *item* = "computer"$\rangle$. A data cube **measure** is a numeric 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—distributive, algebraic, and holistic—based on the kind of aggregate functions used.

Distributive: An aggregate function is *distributive* if it can be computed in a distributed manner as follows. Suppose the data are partitioned into n sets arbitrarily. We apply the function to each partition, resulting in n aggregate values. If the result derived by applying the function to the

n aggregate values is the same as that derived by applying the function to the entire data set (i.e., without partitioning), the function can be computed in a distributed manner.

For example, `sum()` can be computed for a data cube by first partitioning the cube into a set of subcubes, computing `sum()` for each subcube, and then summing up the counts obtained for each subcube. Hence, `sum()` is a distributive aggregate function. For the same reason, `count()`, `min()`, and `max()` are distributive aggregate functions. By treating the count value of each nonempty base cell as 1 by default, `count()` of any cell in a cube can be viewed as the sum of the count values of all of its corresponding child cells in its subcube. Thus, `count()` is distributive. A measure is *distributive* if it is obtained by applying a distributive aggregate function. Distributive measures can be computed efficiently because of the way the computation can be partitioned.

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

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

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

4.3 OLAP Operations

A data warehouse needs to support online multidimensional analytic queries. In this section, you will learn a series of typical OLAP operations on data warehouses (Section 4.3.1) and how to index data to support some OLAP queries

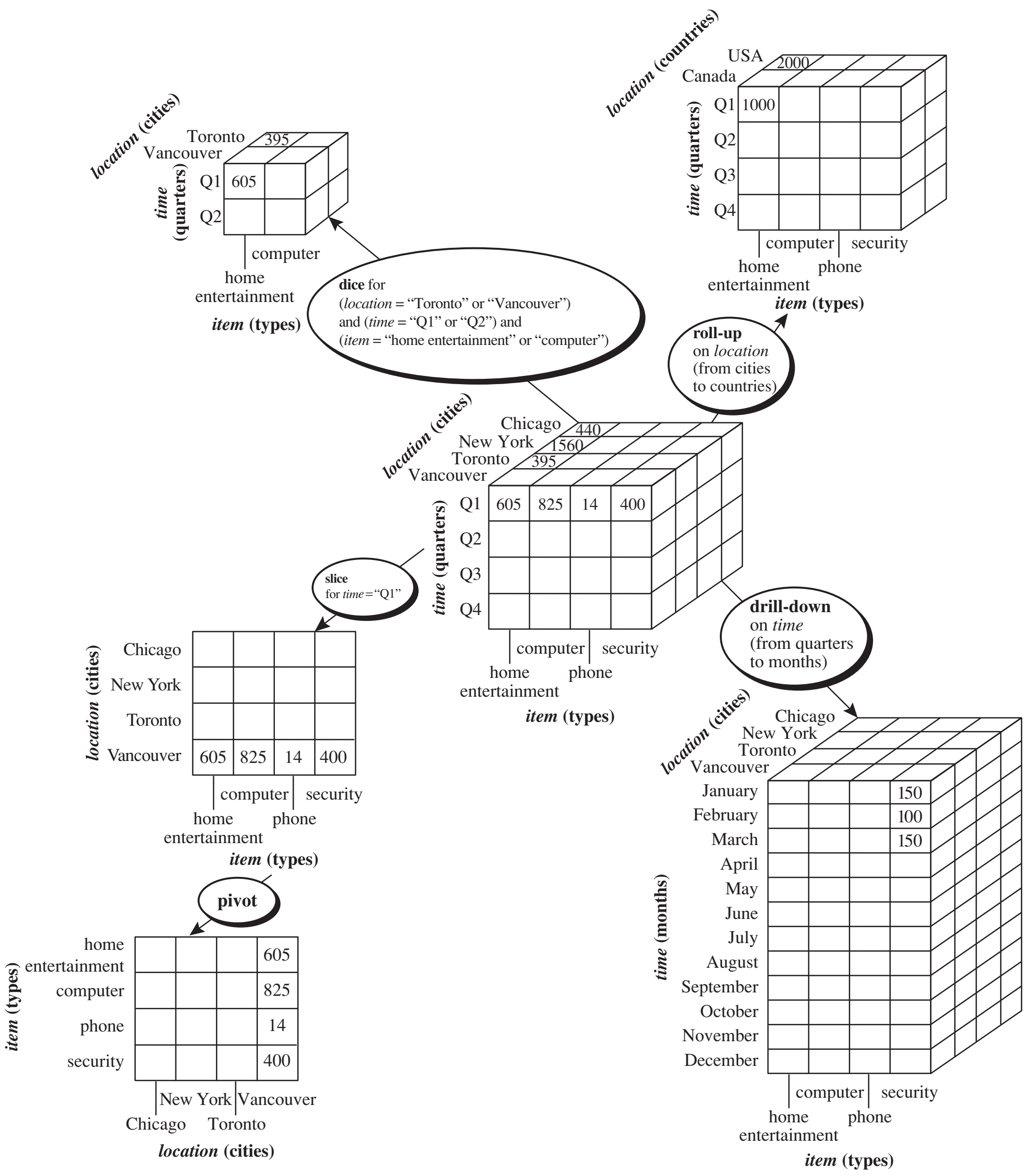


Figure 4.13: Examples of typical OLAP operations on multidimensional data.

(Section 4.3.2). An important problem is how data can be stored properly to support OLAP operations, which will be explained in Section 4.3.3.

4.3.1 Typical OLAP Operations

"How can multidimensional OLAP operations be used in data analysis?" In a multidimensional model, data is 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 empower interactive querying and analysis of the data at hand. Hence, OLAP provides a user-friendly environment for interactive data analysis.

Example 4.5 **OLAP operations.** Let us look at some typical OLAP operations for multidimensional data. Each of the following operations is illustrated in Figure 4.13. At the center of the figure is a data cube for sales in a company. The cube contains the dimensions *location, time*, and *item*, where *location* is aggregated with respect to city values, *time* is aggregated with respect to quarters, and *item* is aggregated with respect to item types. To aid in our explanation, we refer to this cube as the central cube. The measure displayed is *dollars_sold* (in thousands). (For the sake of readability, only some cell values in the cubes are shown.) The data examined are for the cities Chicago, New York, Toronto, and Vancouver.

Roll-up: The roll-up operation (also called the *drill-up* operation by some vendors) performs aggregation on a data cube, either by *climbing up a concept hierarchy* for a dimension or by *dimension reduction.* Figure 4.13 shows the result of a roll-up operation performed on the central cube by climbing up the concept hierarchy for *location* given in Figure 4.10. 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 *location* and *time* dimensions. Roll-up may be performed by removing, say, the *location* dimension, resulting in an aggregation of the total sales by time of the whole company, rather than by location and by time.

Drill-down: Drill-down is the reverse of roll-up. It navigates from less detailed data to more detailed data. Drill-down can be realized by either *stepping down a concept hierarchy* for a dimension or *introducing additional dimensions.* Figure 4.13 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 summarizing them by quarter.

Because 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 in Figure 4.13 can occur by introducing an additional dimension, such as *customer_group*.

Slice and dice: The *slice* operation performs a selection on one dimension of the given cube, resulting in a subcube. Figure 4.13 shows a slice operation where the sales data are selected from the central cube for the dimension *time* using the criterion *time* = "Q1." The *dice* operation defines a subcube by performing a selection on two or more dimensions. Figure 4.13 shows a dice operation on the central cube based on the following selection criteria that involve three dimensions: (*location* = "Toronto" `or` "Vancouver") `and` (*time* = "Q1" `or` "Q2") `and` (item = "home entertainment" `or` "computer").

Pivot (rotate): *Pivot* (also called *rotate*) is a visualization operation that rotates the data axes in view to provide an alternative data presentation. Figure 4.13 shows a pivot operation where the *item* and *location* axes in a 2-D slice are rotated. Other examples include rotating the axes in a 3-D cube, or transforming a 3-D cube into a series of 2-D planes.

Other OLAP operations: Some OLAP systems offer additional drilling operations. For example, **drill-across** executes queries involving (i.e., across) more than one fact table. The **drill-through** operation uses 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 return rates, depreciation, currency conversions, and statistical functions.

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

4.3.2 Indexing OLAP Data: Bitmap Index and Join Index

To facilitate efficient data accessing, most data warehouse systems support index structures and materialized views (using cuboids). We will discuss the general methods to select cuboids for materialization in Section 4.4. In this subsection, 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. A 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, $\boldsymbol{Bv}$, for each value v in the attribute's domain. If a given attribute's domain consists of n values, then n bits are needed for each entry in the bitmap index (i.e., there are n bit vectors). If the attribute has the value v for a given row in the data table, then the bit representing that value is set to 1 in the corresponding row of the bitmap index. All other bits for that row are set to 0.

Example 4.6 **Bitmap indexing.** Consider a customer information table shown in Figure 4.14, where there is an attribute `gender`. To keep our discussion simple, assume there are two possible values on attribute `gender`. We may use one character, that is 8 bits, for each record to represent the `gender` value, such as F for female and M for male. Bitmap index represents the `gender` value using one bit, such as 0 for female and 1 for male. This representation immediately brings in an eight-fold saving in storage.

More importantly, bitmap index can speed up many aggregate queries. For example, let us count the number of female customers in the customer information table. A straightforward method has to scan each record and count. For a table having $10,000$ records and each record taking 100 bytes, the total I/O cost is $10,000 \times 100 = 1,000,000$ bytes.

A bitmap index uses only 1 bit for each record. Those bits are packed into words in storage. For example, for the first 8 records in the table, the bitmap index values are packed into a byte 01010011. Scanning the whole bitmap index takes only $10,000$ bits in I/O, that is $1,250$ bytes. This is a 800 times less than scanning the whole table.

To calculate the number of 0s in a byte, we can simply use a pre-computed hash table that uses the byte values as the index and stores the corresponding numbers of 0s. For example, the hash table stores value 4 in the 83-th entry, since 83 is the decimal value of binary 01010011 and the binary string has 4 0's. Using the byte 01010011 to search the hash table, we immediately know that there are 4 female customers in the first 8 records. We can compute the number of 0s in the whole `gender` attribute byte by byte using the bitmap index, and sum up the byte-wise counts to derive the total number of female customers. In practice, one can use machine words instead of bytes to further speed up the counting process.

Bitmap indexing is advantageous compared to hash and tree indices in answering some types of OLAP queries. It is especially useful for low-cardinality

Customer information

Name	…	Gender	…
Ada	…	Female	…
Bob	…	Male	…
Cathy	…	Female	…
Dan	…	Male	…
Elsa	…	Female	…
Flora	…	Female	…
George	…	Male	…
Hogan	…	Male	…
…	…	…	…

Bitmap index

Gender
0
1
0
1
0
0
1
1
…

A table counting 0s in bytes

Byte	Number of 0s
00000000	8
00000001	7
00000010	7
00000011	6
…	…
01010011	4
…	…

Figure 4.14: Indexing OLAP data using bitmap indices.

A fact table

…	…	Amount
…	…	15.21
…	…	27.06
…	…	…

The bit-sliced index

Weights											
2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	0	1	1	1	1	1	0	0	0	1
1	0	1	0	1	0	0	1	0	0	1	0
…	…	…	…	…	…	…	…	…	…	…	…

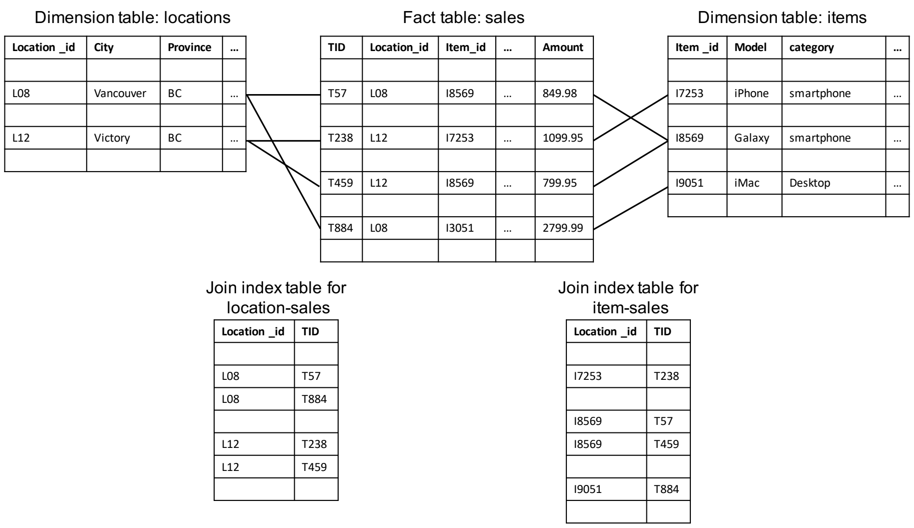
Figure 4.15: Indexing OLAP data using bitmap indices.

domains because comparison, join, and aggregation operations are then reduced to bit arithmetic, which substantially reduces the processing time. Bitmap indexing leads to significant reductions in space and input/output (I/O) since a string of characters can be represented by a single bit.

Bitmap indexing can be extended to bit-sliced indexing for numeric data. Let us illustrate the ideas using an example.

Example 4.7 **Bit-sliced indexing.** Suppose we want to compute the sum of the amount attribute in the fact table in Figure 4.15. We can write an amount into an integer number of pennies, and then represent it as a binary number of n bits. If we represent an amount using 32 bits, that is, 4 bytes, it is good for amounts up to $\$42,949,672.96$ and sufficient for many application scenarios.

After we represent all amount numbers in binary, we can build a bitmap index for every bit. To compute the sum of all amounts, we count for each bit the number of 1s. Denote by x_i ($i \geq 0$) the number of 1s in the i-th bits of the amounts from right to left, the rightmost being bit 0. Since a 1 at the i-th bit


Dimension table: locations
Fact table: sales
Dimension table: items
Join index table for location-sales
Join index table for item-sales

Dimension table: locations

Location _id	City	Province	...
L08	Vancouver	BC	...
L12	Victory	BC	...

Fact table: sales

TID	Location_id	Item_id	...	Amount
T57	L08	I8569	...	849.98
T238	L12	I7253	...	1099.95
T459	L12	I8569	...	799.95
T884	L08	I3051	...	2799.99

Dimension table: items

Item _id	Model	category	...
I7253	iPhone	smartphone	...
I8569	Galaxy	smartphone	...
I9051	iMac	Desktop	...

Join index table for location-sales

Location _id	TID
L08	T57
L08	T884
L12	T238
L12	T459

Join index table for item-sales

Location _id	TID
I7253	T238
I8569	T57
I8569	T459
I9051	T884

Figure 4.16: Join index.

carries a weight of 2^i pennies, the x_i 1s in the i-th bits of all amounts represent $x_i \cdot 2^i$ pennies in the sum of the amounts. Therefore, the sum of amounts is $\sum_{i\geq 0} x_i \cdot 2^i$ pennies or $\frac{\sum_{i\geq 0} x_i \cdot 2^i}{100}$ dollars.

In a data warehousing schema such as the star-schema, we often need to join the fact table and the dimension tables. Joining tables again and again for various queries is definitely costly. Therefore, **join indexing** is used to precompute and store the identifier pairs of the join results so that the join results can be accessed efficiently.

Example 4.8 **Join indexing.** In Example 4.1, we defined a star schema of the form "*sales_star* [*time, item, branch, location*]: *dollars_sold* = `sum` *(sales_in_dollars)*." An example of a join index relationship between the *sales* fact table and the *locations* and *items* dimension tables is shown in Figure 4.16. Consider the OLAP query "the total sales of smartphone and desktop in BC". If no index presents, then we have to join the fact table and the dimension tables *locations* and *items*, and select only those join results about "smartphone" and "desktop".

A join index table records the primary keys of the matching tuples in two tables. For example, in the join index table for location-sales, the pairs of *location_id* and *TID* of matching tuples in the dimension table *locations* and *sales* are recorded. From the join index table, we can quickly find out the TIDs of the tuples in the *sales* fact table belonging to "BC". Similarly, using the join index for item-sales, we can identify the tuples in the sales table about "smartphone" and "desktop". Using the identified TIDs as such, we can accurately access the tuples in the fact table that are needed to compute the OLAP aggregate and reduce the I/O cost. Typically, a data warehouse only contains a very small percentage of transactions about a selected area and product categories. For example, there may be only 0.1% of the transactions in the fact table that are smartphones and desktops sold in BC. Without using any index, we have to read the whole fact table into main memory in order to compute the aggregate. Using the join indexes, even each page contains 100 transaction records in the fact table, and all those transactions of smartphones and desktops sold in BC are evenly distributed, we only need to read 10% of the pages into main memory and thus save 90% of I/O.

4.3.3 Storage Implementation: Column-based Databases

"How to store data so that OLAP queries can be answered efficiently?" In many applications, a fact table may be wide and contain tens or even hundreds of attributes. More often than not, an OLAP query may compute the aggregate of all records or a large portion of records on a small number number of attributes. If the data are stored in a traditional relational table where records are stored row by row, then we have to scan all records in order to answer a query, but only a small segment in a record is used. This observation presents a significant opportunity to develop more efficient storage scheme for OLAP data.

To make the storage more efficient for answering OLAP queries, a column-based database stores a wide table that is often used for aggregate queries in a column by column style. Specifically, a column-based database stores the values of all records on a column in consecutive storage blocks. All records are listed in the same order across all columns.

Example 4.9 **Column-based Database.** Consider a fact table about customer information, which includes attributes and storage space in number of bytes `customer_id` (2), `last_name` (20), `first_name` (20), `gender` (1), `birthdate` (2), `address_street` (50), `address_city` (2), `address_province` (1), `address_country` (1), `email` (30), `registration_date` (2) textttfamily_income (2) Each record occupies 133 bytes, and the fact table contains 10 million customer records, then the total space is over 1 GB.

If the data are stored row by row and we want to answer the OLAP query of the average family income of female customers by province, then we have to scan the whole table, reading all records. The I/O cost is 1 GB. At the same time, for each record, we only need to use 4 bytes among the 133 bytes: the attributes `gender`, `address_province` and `family_income`. In other words, only $\frac{4}{133} = 3\%$ of the data read are useful to answer the query.

A column-based database stores the data attribute by attribute in column, as shown in Figure 4.17. To answer the above query, a column-based database only needs to read three columns, `gender`, `address_province` and `family_income`. It checks the values on `gender` and increments the total and count for `address_province` accordingly. Overall, the total amount of I/O incurred to a column-based database in this case is $4\times$ 10 million = 40 MB. A huge saving in I/O is achieved.

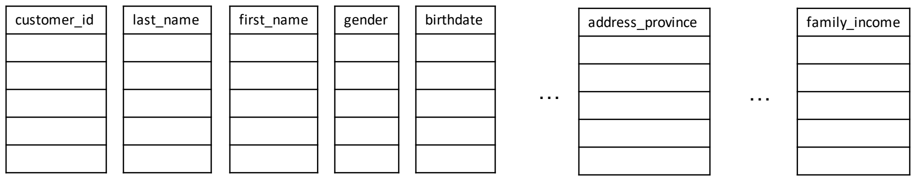


Figure 4.17: Column-based storage.

In implementation, preferably a column-based database processes a column at a time, and uses bitmaps to keep the intermediate results so that they can be passed to the next column. In this example, we can first process the column `gender` and use a bitmap to keep the list of female customers. That is, each customer is associated with a bit, female being 1 and male being 0. Next, we can process the column `address_province`, and form a bitmap for each province. If a customer lives in BC, for example, then the associated bit in the bitmap of BC is set to 1, otherwise, it is set to 0. Last, to calculate the average family income of customers in BC, we only need to conduct the bitwise AND operation between the bitmap for gender and the bitmap for province BC. The resulting bitmap is used to select the entries in column `family_income` to calculate the average.

Column-based databases have been extensively implemented in industry data warehousing and OLAP databases. Column-based databases have remarkable advantages for OLAP-like workloads, such as those aggregate queries searching a few columns of all records in a wide table. At the same time, column-based databases have to separate transactions into columns and compressed transactions as they are stored, which make column-based databases costly for OLTP workloads.

4.4 Data Cube

Data warehouses contain huge volumes of data. OLAP servers demand that decision support queries be answered in the order of seconds. Therefore, it is crucial for data warehouse systems to support highly efficient cube computation techniques, access methods, and query processing techniques. In this section, we present an overview of the ideas behind data cube. Section 4.4.1 introduces the intuition and basic concepts. Section 4.4.2 discusses various ideas in materializing fully or partially a data cube. Section 4.4.3 overviews the general strategies frequently used in data cube computation. The detailed algorithms for data cube computation will be introduced in Section 4.5.

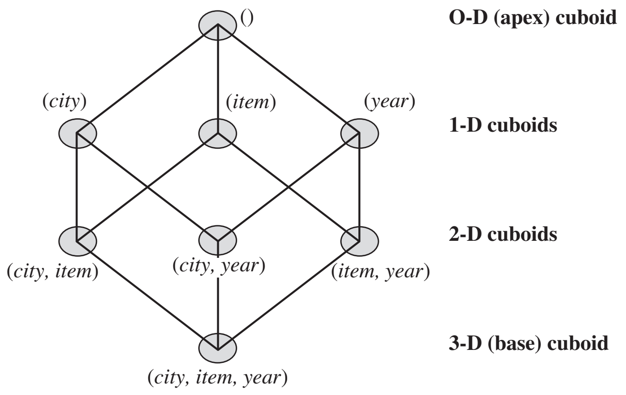


Figure 4.18: Lattice of cuboids, making up a 3-D data cube. Each cuboid represents a different group-by. The base cuboid contains *city, item*, and *year* dimensions.

4.4.1 Data Cube: the Concept

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

One approach to cube computation is to compute aggregates over all subsets of the dimensions specified by a user. This can require excessive storage space, especially for large numbers of dimensions. We start with an intuitive look at what is involved in the efficient computation of data cubes.

Example 4.10 **A data cube is a lattice of cuboids.** Suppose that you want to create a data cube on sales that contains the following: *city, item, year*, and *sales_in_dollars*. You want to be able to analyze the data, with queries such as the following:

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

Taking the three attributes, *city, item*, and *year*, as the dimensions for the data cube, and *sales_in_dollars* as the measure, the total number of cuboids, or group-by's, that can be computed for this data cube is $2^3 = 8$. The possible group-by's are the following: {(*city, item, year*), (*city, item*), (*city, year*), (*item, year*), (*city*), (*item*), (*year*), ()}, where () means that the group-by is empty (i.e., the dimensions are not grouped) and thus the sum of all sales is returned. These group-by's form a lattice of cuboids for the data cube, as shown in Figure 4.18.

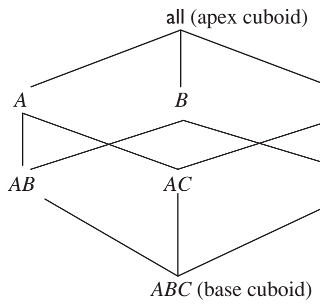


Figure 4.19: Lattice of cuboids making up a 3-D data cube with the dimensions A, B, and C for some aggregate measure, M.

The **base cuboid** contains all three dimensions, *city, item*, and *year*. It can return the total sales for any combination of the three dimensions. The **apex cuboid**, or 0-D cuboid, refers to the case where the group-by is empty. It contains the total sum of all sales. The base cuboid is the least generalized (most specific) of the cuboids. The apex cuboid is the most generalized (least specific) of the cuboids, and is often denoted as `all`. If we start at the apex cuboid and explore downward in the lattice, this is equivalent to drilling down within the data cube. If we start at the base cuboid and explore upward, this is akin to rolling up.

To discuss the details about data cube computation and analysis, we need some terminology. Figure 4.19 shows a 3-D data cube for the dimensions A, B, and C, and an aggregate measure, M. In this chapter, we will always use the term *data cube* to refer to a lattice of cuboids rather than an individual cuboid. A tuple in a cuboid is also called a **cell**. A cell in the base cuboid is a **base cell**. A cell from a nonbase cuboid is an **aggregate cell**. An aggregate cell aggregates over one or more dimensions, where each aggregated dimension is indicated by a $*$ in the cell notation. Suppose we have an n-dimensional data cube. Let $a = (a_1, a_2, \ldots, a_n, measures)$ be a cell from one of the cuboids making up the data cube. We say that a is an $\boldsymbol{m}$**-dimensional cell** (i.e., from an m-dimensional cuboid) if exactly m ($m \leq n$) values among $\{a_1, a_2, \ldots, a_n\}$ are *not* $*$. If $m = n$, then a is a base cell; otherwise, it is an aggregate cell (i.e., where $m < n$).

Example 4.11 **Base and aggregate cells.** Consider a data cube with the dimensions *month, city*, and *customer_group*, and the measure *sales*. $(Jan, *, *, 2800)$ and $(*, Chicago, *, 1200)$ are 1-D cells; $(Jan, *, Business, 150)$ is a 2-D cell; and $(Jan, Chicago, Business, 45)$ is a 3-D cell. Here, all base cells are 3-D, whereas 1-D and 2-D cells are aggregate cells.

An ancestor–descendant relationship may exist between cells. In an n-dimensional data cube, an i-D cell $a = (a_1, a_2, \ldots, a_n, measures_a)$ is an **an-**

cestor of a j-D cell $b = (b_1, b_2, \ldots, b_n, measures_b)$, and b is a **descendant** of a, if and only if (1) $i < j$, and (2) for $1 \leq k \leq n$, $a_k = b_k$ whenever $a_k \neq *$. In particular, cell a is called a **parent** of cell b, and b is a **child** of a, if and only if $j = i + 1$.

Example 4.12 **Ancestor and descendant cells.** Referring to Example 4.4.1, 1-D cell $a = (Jan,\ *\ ,\ *\ , 2800)$ and 2-D cell $b = (Jan,\ *\ , Business, 150)$ are *ancestors* of 3-D cell $c = (Jan,\ Chicago, Business, 45)$; c is a *descendant* of both a and b; b is a *parent* of c; and c is a *child* of b.

"*How many cuboids are there in an* n-*dimensional data cube?*" If there is no hierarchy associated with any dimension, then the total number of cuboids for an n-dimensional data cube, as we have seen, is $\binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n} = 2^n$. However, in practice, many dimensions do have hierarchies. For example, *time* is often explored at multiple conceptual levels such as in the hierarchy "*day* < *month* < *quarter* < *year*." On a dimension that is associated with L levels, cuboid has $L + 1$ possible choices, that is, one of the L levels or the virtual top level `all` meaning not including the dimension in the group-by. Thus, for an n-dimensional data cube, the total number of cuboids that can be generated (including the cuboids generated by climbing up the hierarchies along each dimension) is

$$Total\ number\ of\ cuboids = \prod_{i=1}^{n}(L_i + 1), \tag{4.1}$$

where L_i is the number of levels associated with dimension i.

For example, the time dimension as specified before has four conceptual levels, or five if we include the virtual level `all`. If the cube has 10 dimensions and each dimension has five levels (including `all`), the total number of cuboids that can be generated is $5^{10} \approx 9.8 \times 10^6$. The size of each cuboid also depends on the *cardinality* (i.e., number of distinct values) of each dimension. For example, if every item is sold in each city, there would be $|city| \times |item|$ tuples in the $(city, item)$ group-by alone. As the number of dimensions, number of conceptual hierarchies, or cardinality increases, the storage space required for many of the group-by's will grossly exceed the (limited) size of the input relation. Indeed, given a base table and a set of dimensions, how to fast calculate or estimate the number of tuples in the resulting data cube remains an unsolved challenge.

4.4.2 Data Cube Materialization: Ideas

By now, you probably realize that in large scale applications, it may not be desirable or realistic to precompute and materialize all cuboids that can possibly be generated for a data cube (i.e., 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.

There are three choices for data cube materialization.

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

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

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

Nonetheless, full cube computation algorithms are important. We can use such algorithms to compute smaller cubes, consisting of a subset of the given set of dimensions, or a smaller range of possible values for some of the dimensions. In these cases, the smaller cube is a full cube for the given subset of dimensions and/or dimension values. A thorough understanding of full cube computation methods will help us develop efficient methods for computing partial cubes. Hence, it is important to explore scalable methods for computing all 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, the total size of the computed data cube, as well as the time required for such computation.

Partial materialization of data cubes offers an interesting trade-off between storage space and response time for OLAP. Instead of computing the full cube, we can compute only a subset of the data cube's cuboids, or subcubes consisting of subsets of cells from the various cuboids.

Many cells in a cuboid may actually be of little or no interest to the data analyst. Recall that each cell in a full cube records an aggregate value such as `count` or `sum`. For many cells in a cuboid, the measure value will be zero. For example, if item "snow-tire" is not sold in city "Pheonix" in June at all, the corresponding aggregate cell will have measure value of 0 for `count` or `sum`. When the product of the cardinalities for the dimensions in a cuboid is large relative to the number of nonzero-valued tuples that are stored in the cuboid, then we say that the cuboid is **sparse**. If a cube contains many sparse cuboids, we say that the cube is **sparse**.

In many cases, a substantial amount of the cube's space could be taken up by a large number of cells with very low measure values. This is because the cube cells are often quite sparsely distributed within a multidimensional space. For example, a customer may only buy a few items in a store at a time. Such an event will generate only a few nonempty cells, leaving most other cube cells empty. In such situations, it is useful to materialize only those cells in a cuboid (group-by) with a measure value above some minimum threshold. In a data cube for sales, say, we may wish to materialize only those cells for which *count* $\geq$ *10* (i.e., where at least 10 tuples exist for the cell's given combination of dimensions), or only those cells representing *sales* $\geq$ *\$100*. This not only saves processing time and disk space, but also leads to a more focused analysis. The cells that cannot pass the threshold are likely to be too trivial to warrant further analysis.

Such partially materialized cubes are known as **iceberg cubes**. The minimum threshold is called the **minimum support threshold**, or *minimum support* (*min_sup*), for short. By materializing only a fraction of the cells in a data cube, the result is seen as the "tip of the iceberg," where the "iceberg" is the potential full cube including all cells. An iceberg cube can be specified with an SQL query, as shown in Example 4.4.2.

Example 4.13 **Iceberg cube.**

```
compute cube sales_iceberg as
select month, city, customer_group, count(*)
from salesInfo
cube by month, city, customer_group
having count(*) >= min_sup
```

The compute cube statement specifies the precomputation of the iceberg cube, *sales_iceberg*, with the dimensions *month*, *city*, and *customer_group*, and the aggregate measure count(). The input tuples are in the *salesInfo* relation. The cube by clause specifies that aggregates (group-by's) are to be formed for each of the possible subsets of the given dimensions. If we were computing the full cube, each group-by would correspond to a cuboid in the data cube lattice. The constraint specified in the having clause is known as the **iceberg condition**. Here, the iceberg measure is count(). Note that the iceberg cube computed here could be used to answer group-by queries on any combination of the specified dimensions of the form having count(*) $>= v$, where $v \geq$ *min_sup*. Instead of count(), the iceberg condition could specify more complex measures such as average().

If we were to omit the having clause, we would end up with the full cube. Let's call this cube *sales_cube*. The iceberg cube, *sales_iceberg*, excludes all the cells of *sales_cube* with a count that is less than *min_sup*. Obviously, if we were to set the minimum support to 1 in *sales_iceberg*, the resulting cube would be the full cube, *sales_cube*.

A naïve approach to computing an iceberg cube would be to first compute the full cube and then prune the cells that do not satisfy the iceberg condition. However, this is still prohibitively expensive. An efficient approach is to compute only the iceberg cube directly without computing the full cube. Sections sec:buc discusses methods for efficient iceberg cube computation.

Introducing iceberg cubes will lessen the burden of computing trivial aggregate cells in a data cube. However, we could still end up with a large number of uninteresting cells to compute. For example, suppose that there are 2 base cells for a database of 100 dimensions, denoted as $\{(a_1, a_2, a_3, \ldots, a_{100}) :$

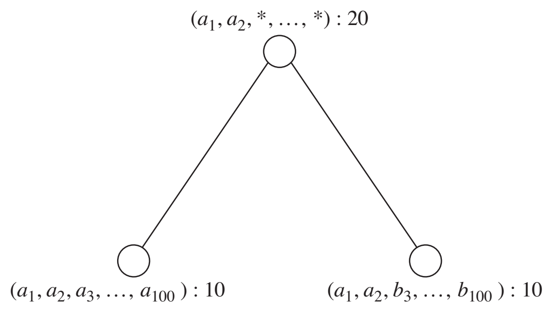

$(a_1, a_2, *, \ldots, *) : 20$
$(a_1, a_2, a_3, \ldots, a_{100}) : 10$
$(a_1, a_2, b_3, \ldots, b_{100}) : 10$

Figure 4.20: Three closed cells forming the lattice of a closed cube.

10, $(a_1, a_2, b_3, \ldots, b_{100}) : 10\}$, where each has a cell count of 10. If the minimum support is set to 10, there will still be an impermissible number of cells to compute and store, although most of them are not interesting. For example, there are $2^{101} - 6$ distinct aggregate cells,[2] like $\{(a_1, a_2, a_3, a_4, \ldots, a_{99}, *) : 10, \ldots, (a_1, a_2, *, a_4, \ldots, a_{99}, a_{100}) : 10, \ldots, (a_1, a_2, a_3, *, \ldots, *, *) : 10\}$, but most of them do not contain much new information. If we ignore all the aggregate cells that can be obtained by replacing some constants by $*$'s while keeping the same measure value, there are only three distinct cells left: $\{(a_1, a_2, a_3, \ldots, a_{100}) : 10, (a_1, a_2, b_3, \ldots, b_{100}) : 10, (a_1, a_2, *, \ldots, *) : 20\}$. That is, out of $2^{101} - 4$ distinct base and aggregate cells, only three really offer valuable information.

To systematically compress a data cube, we need to introduce the concept of *closed coverage*. A cell, c, is a *closed cell* if there exists no cell, d, such that d is a specialization (descendant) of cell c (i.e., where d is obtained by replacing $*$ in c with a non-$*$ value), and d has the same measure value as c. A **closed cube** is a data cube consisting of only closed cells. For example, the three cells derived in the preceding paragraph are the three closed cells of the data cube for the data set $\{(a_1, a_2, a_3, \ldots, a_{100}) : 10, (a_1, a_2, b_3, \ldots, b_{100}) : 10\}$. They form the lattice of a closed cube as shown in Figure 4.20. Other nonclosed cells can be derived from their corresponding closed cells in this lattice. For example, "$(a_1, *, *, \ldots, *) : 20$" can be derived from "$(a_1, a_2, *, \ldots, *) : 20$" because the former is a generalized nonclosed cell of the latter. Similarly, we have "$(a_1, a_2, b_3, *, \ldots, *) : 10$."

Another strategy for partial materialization is to precompute only the cuboids involving a small number of dimensions such as three to five. These cuboids form a **cube shell** for the corresponding data cube. Queries on additional combinations of the dimensions will have to be computed on-the-fly. For example, we could compute all cuboids with three dimensions or less in an n-dimensional data cube, resulting in a cube shell of size 3. This, however, can still result in a large number of cuboids to compute, particularly when n is large. Alternatively, we can choose to precompute only portions or *fragments* of the cube shell based on

[2]The proof is left as an exercise for the reader.

cuboids of interest. Section 4.5.3 discusses a method for computing **shell fragments** and explores how they can be used for efficient OLAP query processing.

4.4.3 General Strategies for Data Cube Computation

There are many methods for efficient data cube computation, based on the various kinds of cubes described earlier in this section. In general, there are two basic data structures used for storing cuboids. The implementation of relational OLAP (ROLAP) uses relational tables, whereas multidimensional arrays are used in multidimensional OLAP (MOLAP). Although ROLAP and MOLAP may each explore different cube computation techniques, some optimization techniques are popularly used.

Optimization Technique 1: Sorting, hashing, and grouping. Sorting, hashing, and grouping operations should be applied to the dimension attributes to reorder and cluster related tuples.

In cube computation, aggregation is performed on the tuples (or cells) that share the same set of dimension values. Thus, it is important to explore sorting, hashing, and grouping operations to access and group such data together to facilitate computation of such aggregates.

To compute total sales by *branch*, *day*, and *item*, for example, it can be more efficient to sort tuples or cells first by *branch*, then by *day*, and last by *item*. Using the sorted data, it is easy to group them according to the *item* name. Efficient implementations of such operations in large data sets have been extensively studied in the algorithm and database research communities, such as counting sort. Such implementations can be extended to data cube computation.

This technique can also be further extended to perform **shared-sorts** (i.e., sharing sorting costs across multiple cuboids when sort-based methods are used), or to perform **shared-partitions** (i.e., sharing the partitioning cost across multiple cuboids when hash-based algorithms are used). For example, using the data sorted first by *branch*, then by *day* and last by *item*, we can compute not only the cuboid (*branch*, *day*, *item*) but also the cuboids (*branch*, *day*, $*$), (*branch*, $*$, $*$) and ().

Optimization Technique 2: Simultaneous aggregation and caching of intermediate results. In cube computation, it is efficient to compute higher-level aggregates from previously computed lower-level aggregates, rather than from the base fact table, since the number of tuples of higher-level aggregates is far less than the number of tuples at the base fact table. For example, to computer the total sales amount of a year, it is more efficient to aggregate from the subtotals of different items of the year. Moreover, simultaneous aggregation from cached intermediate computation results may lead to the reduction of expensive disk input/output (I/O) operations. This technique can be further extended to perform **amortized scans** (i.e., computing as many cuboids as possible at the same time to amortize disk reads).

Optimization Technique 3: Aggregation from the smallest child when there exist multiple child cuboids. When there exist multiple child cuboids, it is usually more efficient to compute the desired parent (i.e., more generalized) cuboid from the smallest in size, previously computed child cuboid. For example, too compute a sales cuboid, C_{branch}, when there exist two previously computed cuboids, $C_{\{branch,year\}}$ and $C_{\{branch,item\}}$, for example, it is obviously more efficient to compute C_{branch} from the former than from the latter if there are many more distinct items than distinct years.

Optimization Technique 4: The downward anti-monotonicity can be used to prune search space in iceberg cube computation For many aggregate measures, the downward anti-monotonicity may hold. *If a given cell does not satisfy the iceberg condition, then no descendant of the cell (i.e., more specialized cell) can satisfy the iceberg condition.*

For example, consider the iceberg condition "`count(*)` >= 1000". If a cell (*, Bellingham, *): 800 fails the iceberg condition, then any descendant of the cell, such as (March, Bellingham, *) and (*, Bellingham, small-business), must also fail the condition, and thus cannot be entitled to be included in the iceberg cube.

The anti-monotonicity property can be used to substantially reduce the computation of iceberg cubes. A common iceberg condition is that the cells must satisfy a *minimum support* threshold such as a minimum count or sum. In this situation, the anti-monotonicity property can be used to prune away the exploration of the cell's descendants.

In the next section, we introduce several popular methods for efficient cube computation that explore these optimization strategies.

4.5 Data Cube Computation Methods

Data cube computation is an essential task in data warehouse implementation. The precomputation of all or part of a data cube can greatly reduce the response time and enhance the performance of online analytical processing. However, such computation is challenging because it may require substantial computational time and storage space. This section explores efficient methods for data cube computation. Section 4.5.1 describes the *multiway array aggregation* (MultiWay) method for computing full cubes. Section 4.5.2 describes a method known as BUC, which computes iceberg cubes from the apex cuboid downward. Section 4.5.3 describes a shell-fragment cubing approach that computes shell fragments for efficient high-dimensional OLAP. Last, Section 4.5.4 demonstrates how to answer OLAP queries using cuboids in data cubes.

To simplify our discussion, we exclude the cuboids that would be generated by climbing up any existing hierarchies for the dimensions. Those cube types can be computed by extension of the discussed methods. Methods for the efficient computation of closed cubes are left as an exercise for interested readers.

4.5.1 Multiway Array Aggregation for Full Cube Computation

The **multiway array aggregation** (or simply **MultiWay**) method computes a full data cube by using a multidimensional array as its basic data structure. It is a typical MOLAP approach that uses direct array addressing, where dimension values are accessed via the position or index of their corresponding array locations. Hence, MultiWay cannot perform any value-based reordering as an optimization technique. A different approach is developed for the array-based cube construction, as follows:

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.* A cell is *empty* if it does not contain any valid data (i.e., its cell count is 0). 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 at handling 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 idea is to exploit this ordering so that portions of the aggregate cells in multiple cuboids can be computed simultaneously, and any unnecessary revisiting of cells is avoided.

This chunking technique involves "overlapping" some of the aggregation computations; therefore, it is referred to as multiway array aggregation. It performs **simultaneous aggregation**, that is, it computes aggregations simultaneously on multiple dimensions.

We explain this approach to array-based cube construction by looking at a concrete example.

Example 4.14 **Multiway array cube computation.** 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 4.21. Dimension A is organized into four equal-sized partitions: a_0, a_1, a_2, and a_3. Dimensions B and C are similarly organized into four partitions each. Chunks 1, 2, ..., 64 correspond to the subcubes $a_0b_0c_0$, $a_1b_0c_0$, ..., $a_3b_3c_3$, respectively. Suppose that the cardinality of the dimensions A, B, and C is 40, 400, and 4000, respectively. Thus, the size of the array for each dimension, A, B, and C, is also 40, 400, and 4000, respectively. Since the number of partitions of each dimension is 4, the size of each partition in A, B, and C is therefore 10, 100, and 1000, respectively. Full materialization of the

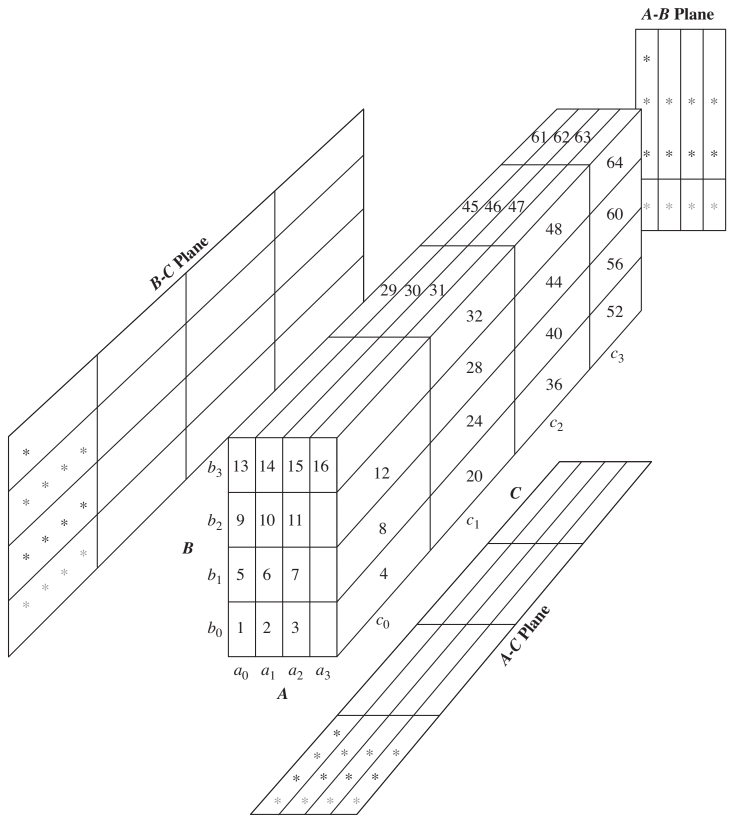


Figure 4.21: A 3-D array for the dimensions A, B, and C, organized into 64 *chunks*. Each chunk is small enough to fit into the memory available for cube computation. The $*$'s indicate the chunks from 1 to 13 that have been aggregated so far in the process.

corresponding data cube involves the computation of all the cuboids defining this cube. The resulting full cube consists of the following cuboids:

- The base cuboid, denoted by ABC (from which all 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 need to be computed.
- The 1-D cuboids, A, B, and C, which respectively correspond to the group-by's A, B, and C. These cuboids need to be computed.
- The 0-D (apex) cuboid, denoted by all, which corresponds to the group-by

(); that is, there is no group-by here. These cuboids need to be computed. It consists of only one value. If, say, the data cube measure is `count`, then the value to be computed is simply the total count of all the tuples in ABC.

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 4.21. Suppose we want to compute the b_0c_0 chunk of the BC cuboid. We allocate space for this chunk in *chunk memory.* By scanning ABC chunks 1 through 4, 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 four ABC chunks: 5 through 8. Continuing in this way, the entire BC cuboid can be computed. Therefore, only *one* BC chunk needs to be in memory at a time, for the computation of all the BC chunks.

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" or "simultaneous aggregation" 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 previously), 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 simultaneously aggregates to each of the 2-D planes while a 3-D chunk is in memory.

Now 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 $400 \times 4000 =$ 1,600,000). The second largest 2-D plane is AC (of size $40 \times 4000 =$ 160,000). AB is the smallest 2-D plane (of size $40 \times 400 = 16,000$).

Suppose that the chunks are scanned in the order shown, from chunks 1 to 64. As previously mentioned, b_0c_0 is fully aggregated after scanning the row containing chunks 1 through 4; b_1c_0 is fully aggregated after scanning chunks 5 through 8, and so on. Thus, we need to scan four chunks of the 3-D array to *fully* compute one chunk of the BC cuboid (where BC is the largest of the 2-D planes). In other words, by scanning in this order, one BC chunk is fully computed for each row scanned. 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. That is, a_0c_0 is fully aggregated only 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

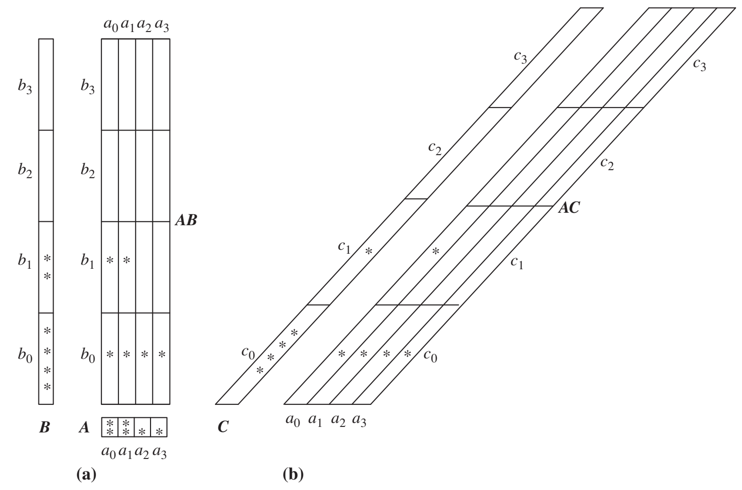


Figure 4.22: Memory allocation and computation order for computing Example 5.4's 1-D cuboids. (a) The 1-D cuboids, A and B, are aggregated during the computation of the smallest 2-D cuboid, AB. (b) The 1-D cuboid, C, is aggregated during the computation of the second smallest 2-D cuboid, AC. The $*$'s represent chunks that, so far, have been aggregated to.

chunks 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: 40×400 (for the whole AB plane) + 40×1000 (for one column of the AC plane) + 100×1000 (for one BC plane chunk) = 16,000 + 40,000 + 100,000 = 156,000 memory units.

Suppose, instead, that the chunks are scanned in the order 1, 17, 33, 49, 5, 21, 37, 53, and so on. That is, suppose the scan is in the order of first aggregating toward the AB plane, and then toward the AC plane, and lastly toward the BC plane. The minimum memory requirement for holding 2-D planes in chunk memory would be as follows: 400×4000 (for the whole BC plane) + 10×4000 (for one AC plane row) + 10×100 (for one AB plane chunk) = 1,600,000 + 40,000 + 1000 = 1,641,000 memory units. Notice that this is *more than 10 times* the memory requirement of the scan ordering of 1 to 64.

Similarly, we can work out the minimum memory requirements for the multiway computation of the 1-D and 0-D cuboids. Figure 4.22 shows the most efficient way to compute 1-D cuboids. Chunks for 1-D cuboids A and B are computed during the computation of the smallest 2-D cuboid, AB. The smallest 1-D cuboid, A, will have all of its chunks allocated in memory, whereas the larger 1-D cuboid, B, will have only one chunk allocated in memory at a time. Similarly, chunk C is computed during the computation of the second smallest 2-D cuboid, AC, requiring only one chunk in memory at a time. Based on this analysis, we see that the most efficient ordering in this array cube computation is the chunk ordering of 1 to 64, with the stated memory allocation strategy.

Example 4.5.1 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 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. MultiWay is most effective when the product of the cardinalities of dimensions is moderate and the data are not too sparse. When the dimensionality is high or the data are very sparse, the in-memory arrays become too large to fit in memory, and this method becomes impractical.

With the use of appropriate sparse array compression techniques and careful ordering of the computation of cuboids, it has been shown by experiments that MultiWay array cube computation is significantly faster than traditional ROLAP (relational record-based) computation. Unlike ROLAP, the array structure of MultiWay does not require saving space to store search keys. Furthermore, MultiWay uses direct array addressing, which is faster than ROLAP's key-based addressing search strategy. For ROLAP cube computation, instead of cubing a table directly, it can be faster to convert the table to an array, cube the array, and then convert the result back to a table. However, this observation works only for cubes with a relatively small number of dimensions, because the number of cuboids to be computed is exponential to the number of dimensions.

"What would happen if we tried to use MultiWay to compute iceberg cubes?" Remember that the downward anti-monotonicity property states that if a given cell does not satisfy the iceberg property, then neither will any of its descendants. Unfortunately, MultiWay's computation starts from the base cuboid and progresses upward toward more generalized, ancestor cuboids. It cannot take advantage of possible pruning using the anti-monotonicity, which requires a parent node to be computed before its child (i.e., more specific) nodes. For example, if the count of a cell c in, say, AB, does not satisfy the minimum support specified in the iceberg condition, we cannot prune away cell c, because the count of c's ancestors in the A or B cuboids may be greater than the minimum support, and their computation will need aggregation involving the count of c.

4.5.2 BUC: Computing Iceberg Cubes from the Apex Cuboid Downward

BUC is an algorithm for the computation of sparse and iceberg cubes. Unlike MultiWay, BUC constructs the cube from the apex cuboid toward the base cuboid. This allows BUC to share data partitioning costs. This processing order also allows BUC to prune during construction, using the downward anti-monotonicity property.

Figure 4.23 shows a lattice of cuboids, making up a 3-D data cube with the dimensions A, B, and C. The apex (0-D) cuboid, representing the concept all (i.e., $(*,\ *\ ,*)$), is at the top of the lattice. This is the most aggregated or

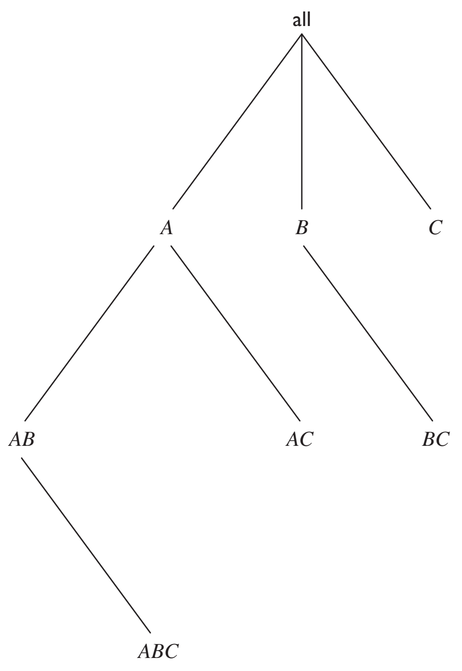


Figure 4.23: BUC's exploration for a 3-D data cube computation. Note that the computation starts from the apex cuboid.

generalized level. The 3-D base cuboid, ABC, is at the bottom of the lattice. It is the least aggregated (most detailed or specialized) level. This representation of a lattice of cuboids, with the apex at the top and the base at the bottom, is commonly accepted in data warehousing. It consolidates the notions of *drill-down* (where we can move from a highly aggregated cell to lower, more detailed cells) and *roll-up* (where we can move from detailed, low-level cells to higher-level, more aggregated cells).

BUC stands for "Bottom-Up Construction." However, according to the lattice convention described before and used throughout this book, the BUC processing order is actually top-down! The BUC authors view a lattice of cuboids in the reverse order, with the apex cuboid at the bottom and the base cuboid at the top. In that view, BUC does bottom-up construction. However, because we adopt the application worldview where *drill-down* refers to drilling from the apex cuboid down toward the base cuboid, the exploration process of BUC is regarded as top-down. BUC's exploration for the computation of a 3-D data cube is shown in Figure 4.23.

The BUC algorithm is shown in Figure 4.24. We first give an explanation of the algorithm and then follow up with an example. Initially, the algorithm is called with the input relation (set of tuples). BUC aggregates the entire input

Algorithm: BUC. Algorithm for the computation of sparse and iceberg cubes.

Input:

- *input*: the relation to aggregate;
- *dim*: the starting dimension for this iteration.

Globals:

- constant *numDims*: the total number of dimensions;
- constant *cardinality[numDims]*: the cardinality of each dimension;
- constant *min_sup*: the minimum number of tuples in a partition for it to be output;
- *outputRec*: the current output record;
- *dataCount[numDims]*: stores the size of each partition. *dataCount[i]* is a list of integers of size *cardinality[i]*.

Output: Recursively output the iceberg cube cells satisfying the minimum support.

Method:

```
(1)  Aggregate(input); // Scan input to compute measure, e.g., count. Place result in outputRec.
(2)  if input.count() == 1 then // Optimization
         WriteDescendants(input[0], dim); return;
     endif
(3)  write outputRec;
(4)  for ($d = dim$; $d < numDims$; $d++$) do //Partition each dimension
(5)      $C$ = cardinality[d];
(6)      Partition(input, d, C, dataCount[d]); //create $C$ partitions of data for dimension $d$
(7)      k = 0;
(8)      for ($i = 0; i < C; i++$) do // for each partition (each value of dimension $d$)
(9)          c = dataCount[d][i];
(10)         if $c >= min\_sup$ then // test the iceberg condition
(11)             outputRec.dim[d] = input[k].dim[d];
(12)             BUC(input[$k..k + c - 1$], $d + 1$); // aggregate on next dimension
(13)         endif
(14)         k +=c;
(15)     endfor
(16)     outputRec.dim[d] = all;
(17) endfor
```

Figure 4.24: BUC algorithm for sparse or iceberg cube computation. *Source:* Beyer and Ramakrishnan [BR99].

(line 1) and writes the resulting total (line 3). (Line 2 is an optimization feature that is discussed later in our example.) For each dimension d (line 4), the input is partitioned on d (line 6). On return from `Partition()`, *dataCount* contains the total number of tuples for each distinct value of dimension d. Each distinct value of d *forms its own partition.* Line 8 iterates through each partition. Line

10 tests the partition for minimum support. That is, if the number of tuples in the partition satisfies (i.e., is $\geq$) the minimum support, then the partition becomes the input relation for a recursive call made to BUC, which computes the iceberg cube on the partitions for dimensions $d + 1$ to *numDims* (line 12).

Note that for a full cube (i.e., where minimum support in the having clause is 1), the minimum support condition is always satisfied. Thus, the recursive call descends one level deeper into the lattice. On return from the recursive call, we continue with the next partition for d. After all the partitions have been processed, the entire process is repeated for each of the remaining dimensions.

Example 4.13 **BUC construction of an iceberg cube.** Consider the iceberg cube expressed in SQL as follows:

```
compute cube iceberg_cube as
select A, B, C, D, count(*)
from R
cube by A, B, C, D
having count(*) >= 3
```

Let's see how BUC constructs the iceberg cube for the dimensions A, B, C, and D, where 3 is the minimum support count. Suppose that dimension A has four distinct values, a_1, a_2, a_3, a_4; B has four distinct values, b_1, b_2, b_3, b_4; C has two distinct values, c_1, c_2; and D has two distinct values, d_1, d_2. If we consider each group-by to be a *partition*, then we must compute every combination of the grouping attributes that satisfy the minimum support (i.e., that have three tuples).

Figure 4.25 illustrates how the input is partitioned first according to the different attribute values of dimension A, and then B, C, and D. To do so, BUC scans the input, aggregating the tuples to obtain a count for all, corresponding to the cell $(*, *, *, *)$. Dimension A is used to split the input into four partitions, one for each distinct value of A. The number of tuples (counts) for each distinct value of A is recorded in *dataCount*.

BUC uses the downward anti-monotonicity property to save time while searching for tuples that satisfy the iceberg condition. Starting with A dimension value, a_1, the a_1 partition is aggregated, creating one tuple for the A group-by, corresponding to the cell $(a_1, *, *, *)$. Suppose $(a_1, *, *, *)$ satisfies the minimum support, in which case a recursive call is made on the partition for a_1. BUC partitions a_1 on the dimension B. It checks the count of $(a_1, b_1, *, *)$ to see if it satisfies the minimum support. If it does, it outputs the aggregated tuple to the AB group-by and recurses on $(a_1, b_1, *, *)$ to partition on C, starting with c_1. Suppose the cell count for $(a_1, b_1, c_1, *)$ is 2, which does not satisfy the minimum support. According to the downward anti-monotonicity property, if a cell does not satisfy the minimum support, then neither can any of its descendants. Therefore, BUC prunes any further exploration of $(a_1, b_1, c_1, *)$. That is, it avoids partitioning this cell on dimension D. It backtracks to the a_1, b_1 partition and recurses on $(a_1, b_1, c_2, *)$, and so on. By checking the iceberg condition each time before performing a recursive

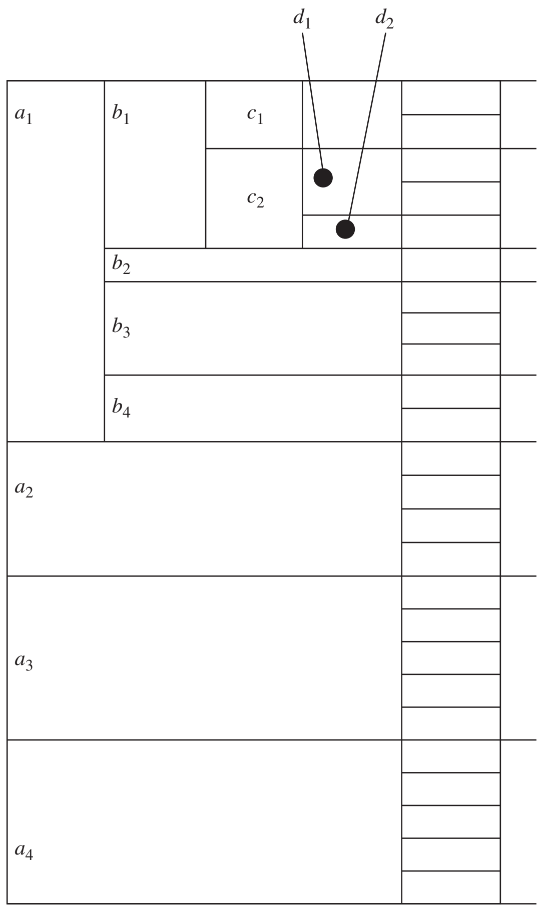


Figure 4.25: BUC partitioning snapshot given an example 4-D data set.

call, BUC saves a great deal of processing time whenever a cell's count does not satisfy the minimum support.

The partition process is facilitated by a linear sorting method, Counting-Sort. CountingSort is fast because it does not perform any key comparisons to find partition boundaries. For example, to sort 10000 tuples according to an attribute A whose value is an integer in the range between 1 and 100, we can set up 100 counters and scan the data once to count the number of 1's 2's, $\ldots$, 100's on attribute A. Suppose there are i_1 tuples having 1 on A, i_2 tuples having 2 on A, and so on. Then, in the next scan, we can move all the tuples having value 1 on attribute A to the first i_1 slots, the tuples have value 2 on attribute A to the slots $i_1 + 1, \ldots, i_1 + i_2$, and so on. After those two scans, the tuples are sorted according to A. In addition, the counts computed during the sort can be reused to compute the group-by's in BUC.

Line 2 is an optimization for partitions having a count of 1 such as (a_1, b_2, $*$, $*$) in our example. To save on partitioning costs, the count is written to each of the tuple's descendant group-by's. This is particularly useful since, in practice, many partitions may have a single tuple.

The BUC performance is sensitive to the order of the dimensions and to skew in the data. Ideally, the most discriminating dimensions should be processed first. Dimensions should be processed in the order of decreasing cardinality. The higher the cardinality, the smaller the partitions, and thus the more partitions there will be, thereby providing BUC with a greater opportunity for pruning. Similarly, the more uniform a dimension (i.e., having less skew), the better it is for pruning.

BUC's major contribution is the idea of sharing partitioning costs. However, unlike MultiWay, it does not share the computation of aggregates between parent and child group-by's. For example, the computation of cuboid AB does not help that of ABC. The latter needs to be computed essentially from scratch.

4.5.3 Precomputing Shell Fragments for Fast High-Dimensional OLAP

Recall the reason that we are interested in precomputing data cubes: Data cubes facilitate fast OLAP in a multidimensional data space. However, a full data cube of high dimensionality needs massive storage space and unrealistic computation time. Iceberg cubes provide a more feasible alternative, as we have seen, wherein the iceberg condition is used to specify the computation of only a subset of the full cube's cells. However, although an iceberg cube is smaller and requires less computation time than its corresponding full cube, it is not an ultimate solution.

For one, the computation and storage of the iceberg cube can still be costly. For example, if the base cuboid cell, $(a_1, a_2, \ldots, a_{60})$, passes minimum support (or the iceberg threshold), it will generate 2^{60} iceberg cube cells. Second, it is difficult to determine an appropriate iceberg threshold. Setting the threshold too low will result in a huge cube, whereas setting the threshold too high may invalidate many useful applications. Third, an iceberg cube cannot be incrementally updated. Once an aggregate cell falls below the iceberg threshold and is pruned, its measure value is lost. Any incremental update would require recomputing the cells from scratch. This is extremely undesirable for large real-life applications where incremental appending of new data is the norm.

One possible solution, which has been implemented in some commercial data warehouse systems, is to compute a thin **cube shell**. For example, we could compute all cuboids with three dimensions or less in a 60-dimensional data cube, resulting in a cube shell of size 3. The resulting cuboids set would require much less computation and storage than the full 60-dimensional data cube. However, there are two disadvantages to this approach. First, we would still need to compute $\binom{60}{3} + \binom{60}{2} + 60 = 36{,}050$ cuboids, each with many cells. Second, such a cube shell does not support high-dimensional OLAP because (1) it does not support OLAP on four or more dimensions, and (2) it cannot even support drilling along three dimensions, such as, say, (A_4, A_5, A_6), *on a subset of data* selected based on the constants provided in three *other* dimensions, such as (A_1, A_2, A_3), because this essentially requires the computation of the corresponding 6-D cuboid. (Notice that there is no cell in cuboid (A_4, A_5, A_6) computed for any particular constant set, such as (a_1, a_2, a_3), associated with dimensions (A_1, A_2, A_3).)

Instead of computing a cube shell, we can compute only portions or fragments of it. This section discusses the *shell fragment* approach for OLAP query processing. It is based on the following key observation about OLAP in high-dimensional space. Although a data cube may contain many dimensions, *most OLAP operations are performed on only a small number of dimensions at a time.* In other words, an OLAP query is likely to ignore many dimensions (i.e., treating them as irrelevant), fix some dimensions (e.g., using query constants as instantiations), and leave only a few to be manipulated (for drilling, pivoting, etc.). This is because it is neither realistic nor fruitful for anyone to comprehend the changes of thousands of cells involving tens of dimensions simultaneously in a high-dimensional space at the same time.

Instead, it is more natural to first locate some cuboids of interest and then drill along one or two dimensions to examine the changes of a few related dimensions. Most analysts will only need to examine, at any one moment, the combinations of a small number of dimensions. This implies that if multidimensional aggregates can be computed quickly on a *small number of dimensions inside a high-dimensional space*, we may still achieve fast OLAP without materializing the original high-dimensional data cube. Computing the full cube (or, often, even an iceberg cube or cube shell) can be excessive. Instead, a *semi-online computation model with certain preprocessing* may offer a more feasible solution. Given a base cuboid, some quick preparation computation can be done first (i.e., offline). After that, a query can then be computed online using the preprocessed data.

The shell fragment approach follows such a semi-online computation strategy. It involves two algorithms: one for computing cube shell fragments and the other for query processing with the cube fragments. The shell fragment approach can handle databases of high dimensionality and can quickly compute small local cubes online. It explores the *inverted index* data structure, which is popular in information retrieval and Web-based information systems.

The basic idea is as follows. Given a high-dimensional data set, we partition the dimensions into a set of disjoint dimension *fragments*, convert each fragment into its corresponding inverted index representation, and then construct *cube shell fragments* while keeping the inverted indices associated with the cube cells. Using the precomputed cubes' shell fragments, we can dynamically assemble and compute cuboid cells of the required data cube online. This is made efficient by set intersection operations on the inverted indices.

To illustrate the shell fragment approach, we use the tiny database of 4.4Table as a running example. Let the cube measure be `count()`. Other measures will be discussed later. We first look at how to construct the inverted index for the given database.

Example 4.16 **Construct the inverted index.** For each attribute value in each dimension,

Table 4.4: Original Database

TID	*A*	*B*	*C*	*D*	*E*
1	a_1	b_1	c_1	d_1	e_1
2	a_1	b_2	c_1	d_2	e_1
3	a_1	b_2	c_1	d_1	e_2
4	a_2	b_1	c_1	d_1	e_2
5	a_2	b_1	c_1	d_1	e_3

Table 4.5: Inverted Index

Attribute Value	*TID List*	*List Size*
a_1	{1, 2, 3}	3
a_2	{4, 5}	2
b_1	{1, 4, 5}	3
b_2	{2, 3}	2
c_1	{1, 2, 3, 4, 5}	5
d_1	{1, 3, 4, 5}	4
d_2	{2}	1
e_1	{1, 2}	2
e_2	{3, 4}	2
e_3	{5}	1

list the tuple identifiers (*TIDs*) of all the tuples that have that value. For example, attribute value a_2 appears in tuples 4 and 5. The TID list for a_2 then contains exactly two items, namely 4 and 5. The resulting inverted index table is shown in 4.5Table . It retains all the original database's information. If each table entry takes one unit of memory, 4.4Tables and 4.5 each takes 25 units, that is, the inverted index table uses the same amount of memory as the original database.

"How do we compute shell fragments of a data cube?" The shell fragment computation algorithm, **Frag-Shells**, is summarized in 4.26Figure . We first partition all the dimensions of the given data set into independent groups of dimensions, called *fragments* (line 1). We scan the base cuboid and construct an inverted index for each attribute (lines 2 to 6). Line 3 is for when the measure is other than the tuple `count()`, which will be described later. For each fragment, we compute the full *local* (i.e., fragment-based) data cube while retaining the inverted indices (lines 7 to 8). Consider a database of 60 dimensions, namely, $A_1, A_2, \ldots, A_{60}$. We can first partition the 60 dimensions into 20 fragments of size 3: (A_1, A_2, A_3), (A_4, A_5, A_6), $\ldots$, (A_{58}, A_{59}, A_{60}). For each fragment, we compute its full data cube while recording the inverted indices. For example, in fragment (A_1, A_2, A_3), we would compute seven cuboids: A_1, A_2, A_3, A_1A_2, A_2A_3, A_1A_3, $A_1A_2A_3$. Furthermore, an

Algorithm: Frag-Shells. Compute shell fragments on a given high-dimensional base table
(i.e., base cuboid).

Input: A base cuboid, B, of n dimensions, namely, $(A_1, \ldots, A_n)$.

Output:

- a set of fragment partitions, $\{P_1, \ldots, P_k\}$, and their corresponding (local) fragment
  cubes, $\{S_1, \ldots, S_k\}$, where P_i represents some set of dimension(s) and $P_1 \cup \ldots \cup P_k$
  make up all the n dimensions
- an *ID_measure* array if the measure is not the tuple count, count()

Method:

```
(1)  partition the set of dimensions ($A_1, \ldots, A_n$) into
         a set of $k$ fragments $P_1, \ldots, P_k$ (based on data & query distribution)
(2)  scan base cuboid, $B$, once and do the following {
(3)      insert each ⟨TID, measure⟩ into ID_measure array
(4)      for each attribute value $a_j$ of each dimension $A_i$
(5)          build an inverted index entry: ⟨$a_j$, TIDlist⟩
(6)  }
(7)  for each fragment partition $P_i$
(8)      build a local fragment cube, $S_i$, by intersecting their
         corresponding TIDlists and computing their measures
```

Figure 4.26: Shell fragment computation algorithm.

inverted index is retained for each cell in the cuboids. That is, for each cell, its associated TID list is recorded.

The benefit of computing local cubes of each shell fragment instead of computing the complete cube shell can be seen by a simple calculation. For a base cuboid of 60 dimensions, there are only $7 \times 20 = 140$ cuboids to be computed according to the preceding shell fragment partitioning. This is in contrast to the 36,050 cuboids computed for the cube shell of size 3 described earlier! Notice that the above fragment partitioning is based simply on the grouping of consecutive dimensions. A more desirable approach would be to partition based on popular dimension groupings. This information can be obtained from domain experts or the past history of OLAP queries.

Let's return to our running example to see how shell fragments are computed.

Example 4.17 **Compute shell fragments.** Suppose we are to compute the shell fragments of size 3. We first divide the five dimensions into two fragments, namely (A, B, C) and (D, E). For each fragment, we compute the full local data cube by intersecting the TID lists in 4.5Table in a top-down depth-first order in the cuboid lattice. For example, to compute the cell $(a_1, b_2, *)$, we intersect the TID lists of a_1 and b_2 to obtain a new list of {2, 3}. Cuboid AB is shown in 4.6Table .

After computing cuboid AB, we can then compute cuboid ABC by intersecting all pairwise combinations between 4.6Table and the row c_1 in 4.5Table . Notice that because cell (a_2, b_2) is empty, it can be effectively discarded in subsequent computations, based on the downward anti-monotonicity property. The same process can be applied to compute fragment (D, E), which is completely independent from computing (A, B, C). Cuboid DE is shown in 4.7Table .

If the measure in the iceberg condition is count() (as in tuple counting), there is no need to reference the original database for this because the *length* of the TID list is equivalent to the tuple count. "*Do we need to reference the original database if computing other measures such as average()?*" Actually, we can build and reference an *ID_measure* array instead, which stores what we need to compute other measures. For example, to compute average(), we let the *ID_measure* array hold three elements, namely, (*TID*, **item_count**, **sum**), for each cell (line 3 of the shell fragment computation algorithm in 4.26Figure). The average() measure for each aggregate cell can then be computed by accessing only this *ID_measure* array, using sum()/item_count(). Considering a database with 10^6 tuples, each taking 4 bytes each for *TID*, item_count, and sum, the *ID_measure* array requires 12 MB, whereas the corresponding database of 60 dimensions will require $(60 + 3) \times 4 \times 10^6 = 252$ MB (assuming each attribute value takes 4 bytes). Obviously, *ID_measure* array is a more compact data structure and is more likely to fit in memory than the corresponding high-dimensional database.

To illustrate the design of the *ID_measure* array, let's look at 4.5.3Example .

Example 4.18 **Computing cubes with the average() measure.** 4.8Table shows an exam-

Table 4.6: Cuboid AB

Cell	*Intersection*	*TID List*	*List Size*
(a_1, b_1)	{1, 2, 3} ∩ {1, 4, 5}	{1}	1
(a_1, b_2)	{1, 2, 3} ∩ {2, 3}	{2, 3}	2
(a_2, b_1)	{4, 5} ∩ {1, 4, 5}	{4, 5}	2
(a_2, b_2)	{4, 5} ∩ {2, 3}	{}	0

Table 4.7: Cuboid DE

Cell	*Intersection*	*TID List*	*List Size*
(d_1, e_1)	{1, 3, 4, 5} ∩ {1, 2}	{1}	1
(d_1, e_2)	{1, 3, 4, 5} ∩ {3, 4}	{3, 4}	2
(d_1, e_3)	{1, 3, 4, 5} ∩ {5}	{5}	1
(d_2, e_1)	{2} ∩ {1, 2}	{2}	1

ple sales database where each tuple has two associated values, such as item_count and sum, where item_count is the count of items sold.

To compute a data cube for this database with the measure average(), we need to have a TID list for each cell: $\{TID_1, \ldots, TID_n\}$. Because each TID is uniquely associated with a particular set of measure values, all future computation just needs to fetch the measure values associated with the tuples in the list. In other words, by keeping an *ID_measure* array in memory for online processing, we can handle complex algebraic measures, such as average, variance, and standard deviation. 4.9Table shows what exactly should be kept for our example, which is substantially smaller than the database itself.

Table 4.8: Database with Two Measure Values

TID	*A*	*B*	*C*	*D*	*E*	*item_count*	*sum*
1	a_1	b_1	c_1	d_1	e_1	5	70
2	a_1	b_2	c_1	d_2	e_1	3	10
3	a_1	b_2	c_1	d_1	e_2	8	20
4	a_2	b_1	c_1	d_1	e_2	5	40
5	a_2	b_1	c_1	d_1	e_3	2	30

Table 4.9: 4.8Table *ID_measure* Array

TID	*item_count*	*sum*
1	5	70
2	3	10
3	8	20
4	5	40
5	2	30

The shell fragments are negligible in both storage space and computation time in comparison with the full data cube. Note that we can also use the Frag-Shells algorithm to compute the full data cube by including all the dimensions as a single fragment. Because the order of computation with respect to the cuboid lattice is top-down and depth-first (similar to that of BUC), the algorithm can perform downward anti-monotonicity pruning if applied to the construction of iceberg cubes.

"Once we have computed the shell fragments, how can they be used to answer OLAP queries?" Given the precomputed shell fragments, we can view the cube space as a virtual cube and perform OLAP queries related to the cube online. In general, two types of queries are possible: (1) *point query* and (2) *subcube query.*

In a **point query**, all of the *relevant* dimensions in the cube have been instantiated (i.e., there are no *inquired* dimensions in the relevant dimensions set). For example, in an n-dimensional data cube, $A_1A_2 \dots A_n$, a point query could be in the form of $\langle A_1, A_5, A_9 : M?\rangle$, where $A_1 = \{a_{11}, a_{18}\}$, $A_5 = \{a_{52}, a_{55}, a_{59}\}$, $A_9 = a_{94}$, and M is the inquired measure for each corresponding cube cell. For a cube with a small number of dimensions, we can use $*$ to represent a "don't care" position where the corresponding dimension is *irrelevant*, that is, neither inquired nor instantiated. For example, in the query $\langle a_2, b_1, c_1, d_1, *$:`count()`$?\rangle$ for the database in 4.4Table , the first four dimension values are instantiated to a_2, b_1, c_1, and d_1, respectively, while the last dimension is irrelevant, and `count()` (which is the tuple count by context) is the inquired measure.

In a **subcube query**, at least one of the *relevant* dimensions in the cube is *inquired.* For example, in an n-dimensional data cube $A_1A_2 \dots A_n$, a subcube query could be in the form $\langle A_1, A_5?, A_9, A_{21}? : M?\rangle$, where $A_1 = \{a_{11}, a_{18}\}$ and $A_9 = a_{94}$, A_5 and A_{21} are the inquired dimensions, and M is the inquired measure. For a cube with a small number of dimensions, we can use $*$ for an irrelevant dimension and ? for an inquired one. For example, in the query $\langle a_2, ?, c_1, *, ? :$ `count()` $?\rangle$ we see that the first and third dimension values are instantiated to a_2 and c_1, respectively, while the fourth is irrelevant, and the second and the fifth are inquired. *A subcube query computes all possible value combinations of the inquired dimensions.* It essentially returns a local data cube consisting of the inquired dimensions.

"How can we use shell fragments to answer a point query?" Because a point query explicitly provides the instantiated variables set on the relevant dimensions set, we can make maximal use of the precomputed shell fragments by finding the *best fitting* (i.e., *dimension-wise completely matching*) fragments to fetch and intersect the associated TID lists.

Let the point query be of the form $\langle \alpha_i, \alpha_j, \alpha_k, \alpha_p : M?\rangle$, where α_i represents a set of instantiated values of dimension A_i, and so on for α_j, α_k, and α_p. First, we check the shell fragment schema to determine which dimensions among A_i, A_j, A_k, and A_p are in the same fragment(s). Suppose A_i and A_j are in the same fragment, while A_k and A_p are in two other fragments. We fetch the corresponding TID lists on the precomputed 2-D fragment for dimensions A_i

and A_j using the instantiations α_i and α_j, and fetch the TID lists on the 1-D fragments for dimensions A_k and A_p using the instantiations α_k and α_p, respectively. The obtained TID lists are intersected to derive the TID list table. This table is then used to derive the specified measure (e.g., by taking the length of the TID lists for tuple count(), or by fetching item_count() and sum() from the *ID_measure* array to compute average()) for the final set of cells.

Example 4.19 **Point query.** Suppose a user wants to compute the point query $\langle a_2, b_1, c_1, d_1, *$: count()?$\rangle$ for our database in 4.4Table and that the shell fragments for the partitions (A, B, C) and (D, E) are precomputed as described in 4.5.3Example . The query is broken down into two subqueries based on the precomputed fragments: $\langle a_2, b_1, c_1, * , *\rangle$ and $\langle *, * , * , d_1, *\rangle$. The best-fit precomputed shell fragments for the two subqueries are ABC and D. The fetch of the TID lists for the two subqueries returns two lists: $\{4, 5\}$ and $\{1, 3, 4, 5\}$. Their intersection is the list $\{4, 5\}$, which is of size 2. Thus, the final answer is count() = 2.

"How can we use shell fragments to answer a subcube query?" A subcube query returns a local data cube based on the instantiated and inquired dimensions. Such a data cube needs to be aggregated in a multidimensional way so that online analytical processing (drilling, dicing, pivoting, etc.) can be made available to users for flexible manipulation and analysis. Because instantiated dimensions usually provide highly selective constants that dramatically reduce the size of the valid TID lists, we should make maximal use of the precomputed shell fragments by finding the fragments that best fit the set of instantiated dimensions, and fetching and intersecting the associated TID lists to derive the reduced TID list. This list can then be used to intersect the best-fitting shell fragments consisting of the inquired dimensions. This will generate the relevant and inquired base cuboid, which can then be used to compute the relevant subcube on-the-fly using an efficient online cubing algorithm.

Let the subcube query be of the form $\langle \alpha_i, \alpha_j, A_k?, \alpha_p, A_q? : M?\rangle$, where α_i, α_j, and α_p represent a set of instantiated values of dimension A_i, A_j, and A_p, respectively, and A_k and A_q represent two inquired dimensions. First, we check the shell fragment schema to determine which dimensions among (1) A_i, A_j, and A_p, and (2) A_k and A_q are in the same fragment partition. Suppose A_i and A_j belong to the same fragment, as do A_k and A_q, but that A_p is in a different fragment. We fetch the corresponding TID lists in the precomputed 2-D fragment for A_i and A_j using the instantiations α_i and α_j, then fetch the TID list on the precomputed 1-D fragment for A_p using instantiation α_p, and then fetch the TID lists on the precomputed 2-D fragments for A_k and A_q, respectively, using no instantiations (i.e., all possible values). The obtained TID lists are intersected to derive the final TID lists, which are used to fetch the corresponding measures from the *ID_measure* array to derive the "base cuboid" of a 2-D subcube for two dimensions (A_k, A_q). A fast cube computation algorithm can be applied to compute this 2-D cube based on the derived base cuboid. The computed 2-D cube is then ready for OLAP operations.

Example 4.20 **Subcube query.** Suppose that a user wants to compute the subcube query, $\langle a_2, b_1, ?, *, ? : \texttt{count}()?\rangle$, for our database shown earlier in 4.4Table , and that the shell fragments have been precomputed as described in 4.5.3Example . The query can be broken into three best-fit fragments according to the instantiated and inquired dimensions: AB, C, and E, where AB has the instantiation (a_2, b_1). The fetch of the TID lists for these partitions returns $(a_2, b_1) : \{4, 5\}$, $(c_1) : \{1, 2, 3, 4, 5\}$ and $\{(e_1 : \{1, 2\}), (e_2 : \{3, 4\}), (e_3 : \{5\})\}$, respectively. The intersection of these corresponding TID lists contains a cuboid with two tuples: $\{(c_1, e_2) : \{4\}$,[3] $(c_1, e_3) : \{5\}\}$. This base cuboid can be used to compute the 2-D data cube, which is trivial.

For large data sets, a fragment size of 2 or 3 typically results in reasonable storage requirements for the shell fragments and for fast query response time. Querying with shell fragments is substantially faster than answering queries using precomputed data cubes that are stored on disk. In comparison to full cube computation, Frag-Shells is recommended if there are less than four inquired dimensions. Otherwise, more efficient algorithms, such as Star-Cubing, can be used for fast online cube computation. Frag-Shells can be easily extended to allow incremental updates, the details of which are left as an exercise.

4.5.4 Efficient Processing of OLAP Queries Using Cuboids

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 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 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 4.21 **OLAP query processing.** Suppose that we define a data cube for *AllElectronics* of the form "*sales_cube* [*time, item, location*]: sum*(sales_in_dollars)*."

[3]That is, the intersection of the TID lists for (a_2, b_1), (c_1), and (e_2) is $\{4\}$.

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 = 2010*." Also, suppose that there are four materialized cuboids available, as follows:

- cuboid 1: {*year, item_name, city*}
- cuboid 2: {*year, brand, country*}
- cuboid 3: {*year, brand, province_or_state*}
- cuboid 4: {*item_name, province_or_state*}, where *year* = 2010

"Which of these 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 because *country* is a more general concept than *province_or_state*. Cuboids 1, 3, and 4 can be used to process the query because (1) they have the same set or a superset of the dimensions in the query, (2) the selection clause in the query can imply the selection in the cuboid, and (3) the abstraction levels for the *item* and *location* dimensions in these cuboids are at a finer level than *brand* and *province_or_state*, respectively.

"How would the costs of each cuboid compare if used to process the query?" It is likely that using cuboid 1 would cost the most because 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 to decide which set of cuboids should be selected for query processing.

4.6 Summary

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

- 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* that contains query and reporting tools.

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

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

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

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

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

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

- 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 **multidimensional data mining** (also known as exploratory multidimensional data mining, online analytical mining, or OLAM). It emphasizes the interactive and exploratory nature of data mining.

- OLAP servers may adopt a **relational OLAP (ROLAP)**, a **multidimensional OLAP (MOLAP)**, or a **hybrid OLAP (HOLAP)** implementation. 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 historic data while maintaining frequently accessed data in a separate MOLAP store.

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

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

- **Data generalization** is a process that abstracts a large set of task-relevant data in a database from a relatively low conceptual level to higher conceptual levels. Data generalization approaches include data cube-based data aggregation and attribute-oriented induction. **Concept description** is the most basic form of descriptive data mining. It describes a given set of task-relevant data in a concise and summarative manner, presenting interesting general properties of the data. Concept (or class) description consists of **characterization** and **comparison** (or **discrimination**). The former summarizes and describes a data collection, called the **target class**, whereas the latter summarizes and distinguishes one data collection, called the **target class**, from other data collection(s), collectively called the **contrasting class(es)**.

- **Concept characterization** can be implemented using **data cube (OLAP-based) approaches** and the **attribute-oriented induction approach**. These are attribute- or dimension-based generalization approaches. The **attribute-oriented induction approach** consists of the following techniques: *data focusing*, *data generalization by attribute removal or attribute generalization*, *count and aggregate value accumulation*, *attribute generalization control*, and *generalization data visualization.*

- **Concept comparison** can be performed using the attribute-oriented induction or data cube approaches in a manner similar to concept characterization. Generalized tuples from the target and contrasting classes can be quantitatively compared and contrasted.

4.7 Exercises

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

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

   (a) Snowflake schema, fact constellation, starnet query model

   (b) Data cleaning, data transformation, refresh

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

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

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

   (c) Starting with the base cuboid $[day, doctor, patient]$, what specific *OLAP operations* should be performed in order to list the total fee collected by each doctor in 2010?

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

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

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

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

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

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

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

   (b) Starting with the base cuboid [$date, spectator, location, game$], what specific *OLAP operations* should you perform in order to list the total charge paid by student spectators at *GM_Place* in 2010?

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

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

7. Design a data warehouse for a regional weather bureau. The weather bureau has about 1000 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 is sent to the central station, which has collected such data for more than 10 years. Your design should facilitate efficient querying and online analytical processing, and derive general weather patterns in multidimensional space.

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

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

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

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

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

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

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

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

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

10. Suppose a company wants to design a data warehouse to facilitate the analysis of moving vehicles in an online analytical processing manner. The company registers huge amounts of auto movement data in the format of (*Auto_ID, location, speed, time*). Each *Auto_ID* represents a vehicle associated with information (e.g., *vehicle_category, driver_category*), and each location may be associated with a street in a city. Assume that a street map is available for the city.

    (a) Design such a data warehouse to facilitate effective online analytical processing in multidimensional space.

    (b) The movement data may contain noise. Discuss how you would develop a method to automatically discover data records that were likely erroneously registered in the data repository.

    (c) The movement data may be sparse. Discuss how you would develop a method that constructs a reliable data warehouse despite the sparsity of data.

    (d) If you want to drive from A to B starting at a particular time, discuss how a system may use the data in this warehouse to work out a fast route.

11. Radio-frequency identification is commonly used to trace object movement and perform inventory control. An RFID reader can successfully read an RFID tag from a limited distance at any scheduled time. Suppose a company wants to design a data warehouse to facilitate the analysis of objects with RFID tags in an online analytical processing manner. The company registers huge amounts of RFID data in the format of *(RFID, at_location, time*), and also has some information about the objects carrying the RFID tag, for example, *(RFID, product_name, product_category, producer, date_produced, price*).

    (a) Design a data warehouse to facilitate effective registration and online analytical processing of such data.

    (b) The RFID data may contain lots of redundant information. Discuss a method that maximally reduces redundancy during data registration in the RFID data warehouse.

    (c) The RFID data may contain lots of noise such as missing registration and misread IDs. Discuss a method that effectively cleans up the noisy data in the RFID data warehouse.

(d) You may want to perform online analytical processing to determine how many TV sets were shipped from the LA seaport to BestBuy in Champaign, IL, by *month*, *brand*, and *price_range*. Outline how this could be done efficiently if you were to store such RFID data in the warehouse.

(e) If a customer returns a jug of milk and complains that is has spoiled before its expiration date, discuss how you can investigate such a case in the warehouse to find out what the problem is, either in shipping or in storage.

12. In many applications, new data sets are incrementally added to the existing large data sets. Thus, an important consideration is whether a measure can be computed efficiently in an incremental manner. Use *count, standard deviation*, and *median* as examples to show that a distributive or algebraic measure facilitates efficient incremental computation, whereas a holistic measure does not.

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

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

    (a) Briefly describe each implementation technique.

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

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

    c Which implementation techniques do you prefer, and why?

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

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

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

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

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

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

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

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

17. 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).*

4.8 Bibliographic Notes

There are a good number of introductory-level textbooks on data warehousing and OLAP technology—for example, Kimball, Ross, Thornthwaite, et al. [**?**]; Imhoff, Galemmo, and Geiger [**?**]; and Inmon [**?**]. Chaudhuri and Dayal [**?**] provide an early overview of data warehousing and OLAP technology. A set of research papers on materialized views and data warehouse implementations were collected in *Materialized Views: Techniques, Implementations, and Applications* by Gupta and Mumick [**?**].

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

Gray et al. [**?**] proposed the data cube as a relational aggregation operator generalizing group-by, crosstabs, and subtotals. Harinarayan, Rajaraman, and Ullman [**?**] proposed a greedy algorithm for the partial materialization of cuboids in the computation of a data cube. Data cube computation methods have been investigated by numerous studies such as Sarawagi and Stonebraker [**?**]; Agarwal et al. [**?**]; Zhao, Deshpande, and Naughton [**?**]; Ross and Srivastava [**?**]; Beyer and Ramakrishnan [**?**]; Han, Pei, Dong, and Wang [**?**]; and Xin, Han, Li, and Wah [**?**]. These methods are discussed in depth in Chapter 5.

The concept of iceberg queries was first introduced in Fang, Shivakumar, Garcia-Molina et al. [**?**]. The use of join indices to speed up relational query processing was proposed by Valduriez [**?**]. O'Neil and Graefe [**?**] proposed a bitmapped join index method to speed up OLAP-based query processing.

A discussion of the performance of bitmapping and other nontraditional index techniques is given in O'Neil and Quass [?].

For work regarding the selection of materialized cuboids for efficient OLAP query processing, see, for example, Chaudhuri and Dayal [?]; Harinarayan, Rajaraman, and Ullman [?]; and Sristava et al. [?]. Methods for cube size estimation can be found in Deshpande et al. [?], Ross and Srivastava [?], and Beyer and Ramakrishnan [?]. Agrawal, Gupta, and Sarawagi [?] proposed operations for modeling multidimensional databases. Methods for answering queries quickly by online aggregation are described in Hellerstein, Haas, and Wang [?] and Hellerstein et al. [?]. Techniques for estimating the top N queries are proposed in Carey and Kossman [?] and Donjerkovic and Ramakrishnan [?]. Further studies on intelligent OLAP and discovery-driven exploration of data cubes are presented in the bibliographic notes in Chapter 5.