23 DECISION SUPPORT

> Nothing is more difficult, and therefore more precious, than to be able to decide.
>
> —Napoleon Bonaparte

Database management systems are widely used by organizations for maintaining data that documents their everyday operations. In applications that update such *operational data*, transactions typically make small changes (for example, adding a reservation or depositing a check) and a large number of transactions must be reliably and efficiently processed. Such **online transaction processing (OLTP)** applications have driven the growth of the DBMS industry in the past three decades and will doubtless continue to be important. DBMSs have traditionally been optimized extensively to perform well in such applications.

Recently, however, organizations have increasingly emphasized applications in which current and historical data are comprehensively analyzed and explored, identifying useful trends and creating summaries of the data, in order to support high-level decision making. Such applications are referred to as **decision support**. Decision support has rapidly grown into a multibillion dollar industry, and further growth is expected. A number of vendors offer specialized database systems and analysis tools to facilitate decision support. Industry organizations are emerging to set standards and create consensus on issues like language and architecture design.

Mainstream relational DBMS vendors have recognized the importance of this market segment and are adding features to their products in order to support it. In particular, novel indexing and query optimization techniques are being added to support complex queries. Systems are also providing additional features for defining and using views. The use of views has gained rapidly in popularity because of their utility in applications involving complex data analysis. While queries on views can be answered by evaluating the view definition when the query is submitted, views also offer the option of precomputing the view definition and thereby making queries run much faster. This option becomes increasingly attractive as the view definition increases in complexity and the frequency of queries increases.

Carrying the motivation for precomputed views one step further, organizations can consolidate information from several databases into a *data warehouse* by copying tables

from many sources into one location or by materializing a view that is defined over tables from several sources. Data warehousing has become widespread, and many specialized products are now available to create and manage warehouses of data from multiple databases.

We begin this chapter with an overview of decision support in Section 23.1. We cover data warehousing in Section 23.2 and present on-line analytic processing, or OLAP, in Section 23.3. We discuss implementation techniques to support OLAP in Section 23.4. These new implementation techniques form the basis for specialized OLAP products, and are also being added to relational DBMS products to support complex decision support applications. We discuss the role of views in decision support applications and techniques for rapidly processing queries on views in Section 23.5. Finally, in Section 23.6, we discuss a recent trend toward quickly computing approximate answers or a desired subset of answers, rather than computing all answers.

23.1 INTRODUCTION TO DECISION SUPPORT

Organizational decision making requires a comprehensive view of all aspects of an enterprise, and many organizations have therefore created consolidated **data warehouses** that contain data drawn from several databases maintained by different business units, together with historical and summary information.

The trend toward data warehousing is complemented by an increased emphasis on powerful analysis tools. There are many characteristics of decision support queries that make traditional SQL systems inadequate:

- The conditions in the `WHERE` clause often contain many `AND` and `OR` conditions. As we saw in Section 12.3.3, `OR` conditions, in particular, are poorly handled in many relational DBMSs.
- Applications require extensive use of statistical functions such as standard deviation, which are not supported in SQL-92. Thus, SQL queries must frequently be embedded in a host language program.
- Many queries involve conditions over time or require aggregating over time periods. SQL-92 provides poor support for such time-series analysis.
- Users often need to pose several related queries. Since there is no convenient way to express these commonly occurring families of queries, users have to write them as a collection of independent queries, which can be tedious. Further, the DBMS has no way to recognize and exploit optimization opportunities arising from executing many related queries together.

Three broad classes of analysis tools are available. First, there are systems that support a class of stylized queries that typically involve group-by and aggregation operators

and provide excellent support for complex boolean conditions, statistical functions, and features for time-series analysis. Applications dominated by such queries are called **online analytic processing**, or **OLAP**. These systems support a querying style in which the data is best thought of as a multidimensional array, and are influenced by end user tools such as spreadsheets, in addition to database query languages.

Second, there are DBMSs that support traditional SQL-style queries but are designed to also support OLAP queries efficiently. Such systems can be regarded as relational DBMSs optimized for decision support applications. Many vendors of relational DBMSs are currently enhancing their products in this direction, and over time the distinction between specialized OLAP systems and relational DBMSs enhanced to support OLAP queries is likely to diminish.

The third class of analysis tools is motivated by the desire to find interesting or unexpected trends and patterns in large data sets, rather than by the complex query characteristics listed above. In **exploratory data analysis**, although an analyst can recognize an 'interesting pattern' when shown such a pattern, it is very difficult to formulate a query that captures the essence of an interesting pattern. For example, an analyst looking at credit-card usage histories may want to detect unusual activity indicating misuse of a lost or stolen card. A catalog merchant may want to look at customer records to identify promising customers for a new promotion; this identification would depend on income levels, buying patterns, demonstrated interest areas, and so on. The amount of data in many applications is too large to permit manual analysis or even traditional statistical analysis, and the goal of **data mining** is to support exploratory analysis over very large data sets. We discuss data mining further in Chapter 24.

Clearly, evaluating OLAP or data mining queries over globally distributed data is likely to be excruciatingly slow. Further, for such complex analysis, often statistical in nature, it is not essential that the most current version of the data be used. The natural solution is to create a centralized repository of all the data, i.e., a data warehouse. Thus, the availability of a warehouse facilitates the application of OLAP and data mining tools, and conversely, the desire to apply such analysis tools is a strong motivation for building a data warehouse.

23.2 DATA WAREHOUSING

Data warehouses contain consolidated data from many sources, augmented with summary information and covering a long time period. Warehouses are much larger than other kinds of databases; sizes ranging from several gigabytes to terabytes are common. Typical workloads involve ad hoc, fairly complex queries and fast response times are important. These characteristics differentiate warehouse applications from OLTP applications, and different DBMS design and implementation techniques must be used

to achieve satisfactory results. A distributed DBMS with good scalability and high availability (achieved by storing tables redundantly at more than one site) is required for very large warehouses.

A typical data warehousing architecture is illustrated in Figure 23.1. An organiza-

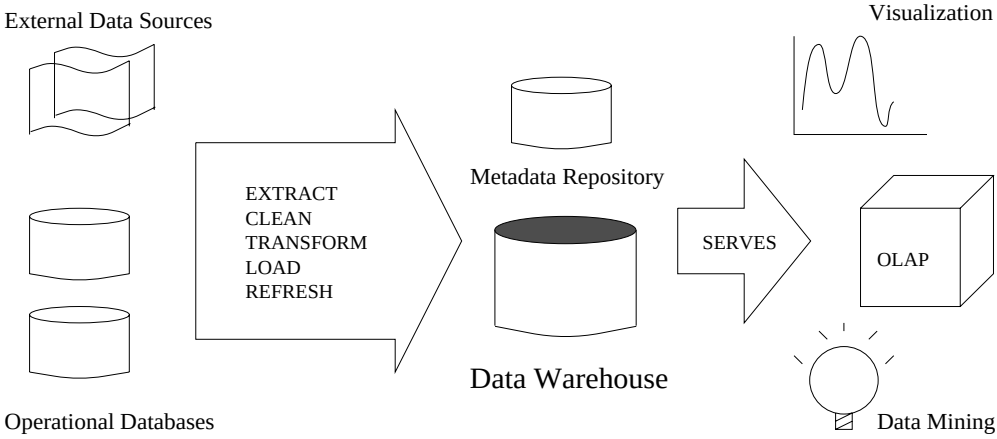


Figure 23.1 A Typical Data Warehousing Architecture

tion's daily operations access and modify **operational databases**. Data from these operational databases and other external sources (e.g., customer profiles supplied by external consultants) are **extracted** by using *gateways*, or standard external interfaces supported by the underlying DBMSs. A **gateway** is an application program interface that allows client programs to generate SQL statements to be executed at a server (see Section 5.10). Standards such as *Open Database Connectivity (ODBC)* and *Open Linking and Embedding for Databases (OLE-DB)* from Microsoft and *Java Database Connectivity (JDBC)* are emerging for gateways.

23.2.1 Creating and Maintaining a Warehouse

There are many challenges in creating and maintaining a large data warehouse. A good database schema must be designed to hold an integrated collection of data copied from diverse sources. For example, a company warehouse might include the inventory and personnel departments' databases, together with sales databases maintained by offices in different countries. Since the source databases are often created and maintained by different groups, there are a number of semantic mismatches across these databases, such as different currency units, different names for the same attribute, and differences in how tables are normalized or structured; these differences must be reconciled when data is brought into the warehouse. After the warehouse schema is designed, the

warehouse must be populated, and over time, it must be kept consistent with the source databases.

Data is **extracted** from operational databases and external sources, **cleaned** to minimize errors and fill in missing information when possible, and **transformed** to reconcile semantic mismatches. Transforming data is typically accomplished by defining a relational view over the tables in the data sources (the operational databases and other external sources). **Loading** data consists of materializing such views and storing them in the warehouse. Unlike a standard view in a relational DBMS, therefore, the view is stored in a database (the warehouse) that is different from the database(s) containing the tables it is defined over.

The cleaned and transformed data is finally **loaded** into the warehouse. Additional preprocessing such as sorting and generation of summary information is carried out at this stage. Data is partitioned and indexes are built for efficiency. Due to the large volume of data, loading is a slow process. Loading a terabyte of data sequentially can take weeks, and loading even a gigabyte can take hours. Parallelism is therefore important for loading warehouses.

After data is loaded into a warehouse, additional measures must be taken to ensure that the data in the warehouse is periodically **refreshed** to reflect updates to the data sources and to periodically **purge** data that is too old from the warehouse (perhaps onto archival media). Observe the connection between the problem of refreshing warehouse tables and asynchronously maintaining replicas of tables in a distributed DBMS. Maintaining replicas of source relations is an essential part of warehousing, and this application domain is an important factor in the popularity of asynchronous replication (Section 21.10.2), despite the fact that asynchronous replication violates the principle of distributed data independence. The problem of refreshing warehouse tables (which are materialized views over tables in the source databases) has also renewed interest in incremental maintenance of materialized views. (We discuss materialized views in Section 23.5.)

An important task in maintaining a warehouse is keeping track of the data currently stored in it; this bookkeeping is done by storing information about the warehouse data in the system catalogs. The system catalogs associated with a warehouse are very large and are often stored and managed in a separate database called a **metadata repository**. The size and complexity of the catalogs is in part due to the size and complexity of the warehouse itself and in part because a lot of administrative information must be maintained. For example, we must keep track of the source of each warehouse table and when it was last refreshed, in addition to describing its fields.

The value of a warehouse is ultimately in the analysis that it enables. The data in a warehouse is typically accessed and analyzed using a variety of tools, including OLAP

query engines, data mining algorithms, information visualization tools, statistical packages, and report generators.

23.3 OLAP

OLAP applications are dominated by ad hoc, complex queries. In SQL terms, these are queries that involve group-by and aggregation operators. The natural way to think about typical OLAP queries, however, is in terms of a multidimensional data model. We begin this section by presenting the multidimensional data model and comparing it with a relational representation of data. We describe OLAP queries in terms of the multidimensional data model and then consider some new implementation techniques designed to support such queries. Finally, we briefly contrast database design for OLAP applications with more traditional relational database design.

23.3.1 Multidimensional Data Model

In the multidimensional data model, the focus is on a collection of numeric **measures**. Each measure depends on a set of **dimensions**. We will use a running example based on sales data. The measure attribute in our example is *sales*. The dimensions are Product, Location, and Time. Given a product, a location, and a time, we have at most associated one sales value. If we identify a product by a unique identifier *pid*, and similarly identify location by *locid* and time by *timeid*, we can think of sales information as being arranged in a three-dimensional array Sales. This array is shown in Figure 23.2; for clarity, we show only the values for a single *locid* value, *locid*= 1, which can be thought of as a slice orthogonal to the *locid* axis.

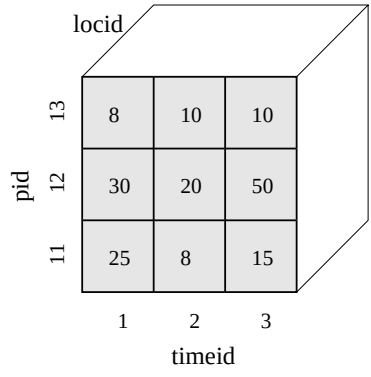


Figure 23.2 Sales: A Multidimensional Dataset

This view of data as a multidimensional array is readily generalized to more than three dimensions. In OLAP applications, the bulk of the data can be represented in such a multidimensional array. Indeed, some OLAP systems, for example, Essbase

from Arbor Software, actually store data in a multidimensional array (of course, implemented without the usual programming language assumption that the entire array fits in memory). OLAP systems that use arrays to store multidimensional datasets are called **multidimensional OLAP (MOLAP)** systems.

The data in a multidimensional array can also be represented as a relation, as illustrated in Figure 23.3, which shows the same data as in Figure 23.2; additional rows corresponding to the 'slice' *locid*= 2 are shown in addition to the data visible in Figure 23.3. This relation, which relates the dimensions to the measure of interest, is called the **fact table**.

locid	*city*	*state*	*country*
1	Ames	Iowa	USA
2	Chennai	TN	India
5	Tempe	Arizona	USA

Locations

pid	*pname*	*category*	*price*
11	Lee Jeans	Apparel	25
12	Zord	Toys	18
13	Biro Pen	Stationery	2

Products

pid	*timeid*	*locid*	*sales*
11	1	1	25
11	2	1	8
11	3	1	15
12	1	1	30
12	2	1	20
12	3	1	50
13	1	1	8
13	2	1	10
13	3	1	10
11	1	2	35
11	2	2	22
11	3	2	10
12	1	2	26
12	2	2	45
12	3	2	20
13	1	2	20
13	2	2	40
13	3	2	5

Sales

Figure 23.3 Locations, Products, and Sales Represented as Relations

Now let us turn to dimensions. Each dimension can have a set of associated attributes. For example, the Location dimension is identified by the *locid* attribute, which we used to identify a location in the Sales table. We will assume that it also has attributes

country, *state*, and *city*. We will also assume that the Product dimension has attributes *pname*, *category*, and *price*, in addition to the identifier *pid*. The *category* of a product indicates its general nature; for example, a product *pant* could have category value *apparel*. We will assume that the Time dimension has attributes *date, week, month, quarter, year,* and *holiday_flag*, in addition to the identifier *timeid*.

For each dimension, the set of associated values can be structured as a hierarchy. For example, cities belong to states, and states belong to countries. Dates belong to weeks and to months, both weeks and months are contained in quarters, and quarters are contained in years. (Note that a week could span a month; thus, weeks are not contained in months.) Some of the attributes of a dimension describe the position of a dimension value with respect to this underlying hierarchy of dimension values. The hierarchies for the Product, Location, and Time hierarchies in our example are shown at the attribute level in Figure 23.4.

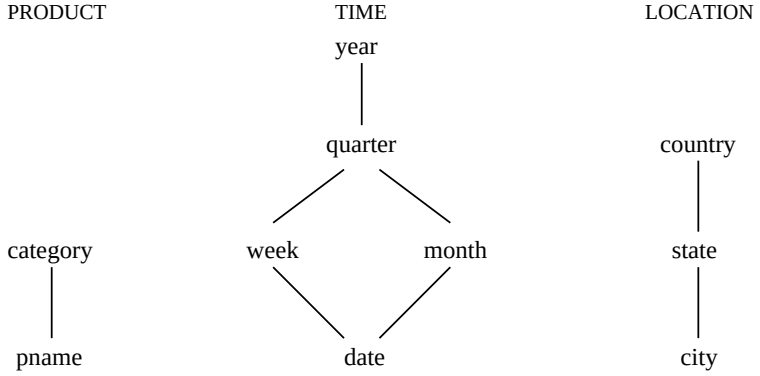


Figure 23.4 Dimension Hierarchies

Information about dimensions can also be represented as a collection of relations:

Locations(*locid:* `integer`, *city:* `string`, *state:* `string`, *country:* `string`)
Products(*pid:* `integer`, *pname:* `string`, *category:* `string`, *price:* `real`)
Times(*timeid:* `integer`, *date:* `string`, *week:* `integer`, *month:* `integer`, *quarter:* `integer`, *year:* `integer`, *holiday_flag:* `boolean`)

These relations are much smaller than the fact table in a typical OLAP application; they are called the **dimension tables**. OLAP systems that store all information, including fact tables, as relations are called **relational OLAP (ROLAP)** systems.

The Times table illustrates the attention paid to the Time dimension in typical OLAP applications. SQL's date and timestamp data types are not adequate; in order to support summarizations that reflect business operations, information such as fiscal quarters, holiday status, and so on is maintained for each time value.

23.3.2 OLAP Queries

Now that we have seen the multidimensional model of data, let us consider how such data can be queried and manipulated. The operations supported by this model are strongly influenced by end user tools such as spreadsheets. The goal is to give end users who are not SQL experts an intuitive and powerful interface for common business-oriented analysis tasks. Users are expected to pose ad hoc queries directly, without relying on database application programmers. In this section we assume that the user is working with a multidimensional dataset and that each operation returns either a different presentation or summarization of this underlying dataset; the underlying dataset is always available for the user to manipulate, regardless of the level of detail at which it is currently viewed.

A very common operation is aggregating a measure over one or more dimensions. The following queries are typical:

- Find the total sales.
- Find total sales for each city.
- Find total sales for each state.
- Find the top five products ranked by total sales.

The first three queries can be expressed as SQL queries over the fact and dimension tables, but the last query cannot be expressed in SQL (although we can approximate it if we return answers in sorted order by total sales, using `ORDER BY`).

When we aggregate a measure on one or more dimensions, the aggregated measure depends on fewer dimensions than the original measure. For example, when we compute the total sales by city, the aggregated measure is *total sales* and it depends only on the Location dimension, whereas the original *sales* measure depended on the Location, Time, and Product dimensions.

Another use of aggregation is to summarize at different levels of a dimension hierarchy. If we are given total sales per city, we can aggregate on the Location dimension to obtain sales per state. This operation is called **roll-up** in the OLAP literature. The inverse of roll-up is **drill-down**: given total sales by state, we can ask for a more detailed presentation by drilling down on Location. We can ask for sales by city or just sales by city for a selected state (with sales presented on a per-state basis for the remaining states, as before). We can also drill down on a dimension other than Location. For example, we can ask for total sales for each product for each state, drilling down on the Product dimension.

Another common operation is **pivoting**. Consider a tabular presentation of the Sales table. If we pivot it on the Location and Time dimensions, we obtain a table of total

sales for each location for each time value. This information can be presented as a two-dimensional chart in which the axes are labeled with location and time values, and the entries in the chart correspond to the total sales for that location and time. Thus, values that appear in columns of the original presentation become labels of axes in the result presentation. Of course, pivoting can be combined with aggregation; we can pivot to obtain yearly sales by state. The result of pivoting is called a **cross-tabulation** and is illustrated in Figure 23.5. Observe that in spreadsheet style, in addition to the total sales by year and state (taken together), we also have additional summaries of sales by year and sales by state.

	WI	CA	Total
1995	63	81	144
1996	38	107	145
1997	75	35	110
Total	176	223	399

Figure 23.5 Cross-Tabulation of Sales by Year and State

Pivoting can also be used to change the dimensions of the cross-tabulation; from a presentation of sales by year and state, we can obtain a presentation of sales by product and year.

The Time dimension is very important in OLAP. Typical queries include:

- Find total sales by month.
- Find total sales by month for each city.
- Find the percentage change in the total monthly sales for each product.
- Find the trailing n day moving average of sales. (For each day, we must compute the average daily sales over the preceding n days.)

The first two queries can be expressed as SQL queries over the fact and dimension tables. The third query can be expressed too, but it is quite complicated in SQL. The last query cannot be expressed in SQL if n is to be a parameter of the query.

Clearly, the OLAP framework makes it convenient to pose a broad class of queries. It also gives catchy names to some familiar operations: **slicing** a dataset amounts to an equality selection on one or more dimensions, possibly also with some dimensions projected out. **Dicing** a dataset amounts to a range selection. These terms come from visualizing the effect of these operations on a cube or cross-tabulated representation of the data.

Comparison with SQL Queries

Some OLAP queries cannot be easily expressed, or cannot be expressed at all, in SQL, as we saw in the above discussion. Notably, queries that rank results and queries that involve time-oriented operations fall into this category.

A large number of OLAP queries, however, can be expressed in SQL. Typically, they involve grouping and aggregation, and a single OLAP operation leads to several closely related SQL queries. For example, consider the cross-tabulation shown in Figure 23.5, which was obtained by pivoting the Sales table. To obtain the same information, we would issue the following queries:

```
SELECT   SUM (S.sales)
FROM     Sales S, Times T, Locations L
WHERE    S.timeid=T.timeid AND S.locid=L.locid
GROUP BY T.year, L.state
```

This query generates the entries in the body of the chart (outlined by the dark lines). The summary row at the bottom is generated by the query:

```
SELECT   SUM (S.sales)
FROM     Sales S, Times T
WHERE    S.timeid=T.timeid
GROUP BY T.year
```

The summary column on the right is generated by the query:

```
SELECT   SUM (S.sales)
FROM     Sales S, Locations L
WHERE    S.locid=L.locid
GROUP BY L.state
```

The example cross-tabulation can be thought of as roll-up on the Location dimension, on the Time dimension, and on the Location and Time dimensions together. Each roll-up corresponds to a single SQL query with grouping. In general, given a measure with k associated dimensions, we can roll up on any subset of these k dimensions, and so we have a total of 2^k such SQL queries.

Through high-level operations such as pivoting, users can generate many of these 2^k SQL queries. Recognizing the commonalities between these queries enables more efficient, coordinated computation of the set of queries. A proposed extension to SQL called the `CUBE` is equivalent to a collection of `GROUP BY` statements, with one `GROUP BY` statement for each subset of the k dimensions. We illustrate it using the Sales relation. Consider the following query:

```
CUBE pid, locid, timeid BY SUM Sales
```

This query will roll up the table Sales on all eight subsets of the set {pid, locid, timeid} (including the empty subset). It is equivalent to eight queries of the form:

```
SELECT    SUM (S.sales)
FROM      Sales S
GROUP BY  grouping-list
```

The queries differ only in the *grouping-list*, which is some subset of the set {pid, locid, timeid}. We can think of these eight queries as being arranged in a lattice, as shown in Figure 23.6. The result tuples at a node can be aggregated further to compute the result for any child of the node. This relationship between the queries arising in a CUBE can be exploited for efficient evaluation.

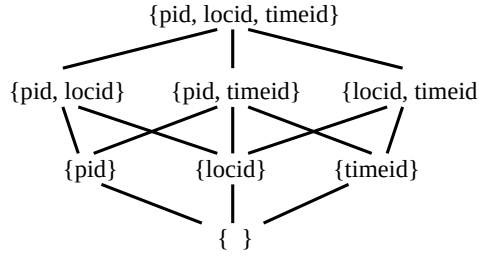


Figure 23.6 The Lattice of GROUP BY Queries in a CUBE Query

We conclude our discussion of the relationship between SQL and OLAP queries by noting that they complement each other, and both are important for decision support. The goal of OLAP is to enable end users to ask a broad class of business-oriented queries easily and with interactive response times over very large datasets. SQL, on the other hand, can be used to write complex queries that combine information from several relations. The data need not be schemas corresponding to the multidimensional data model, and the OLAP querying idioms are not always applicable. Such complex queries are written by application programmers, compiled, and made available to end users as 'canned' programs, often through a menu-driven graphical interface. The importance of such SQL applications is reflected in the increased attention being paid to optimizing complex SQL queries and the emergence of decision support oriented SQL benchmarks, such as TPC-D.

A Note on Statistical Databases

Many OLAP concepts are present in earlier work on **statistical databases (SDBs)**, which are database systems designed to support statistical applications, although this

connection has not been sufficiently recognized because of differences in application domains and terminology. The multidimensional data model, with the notions of a measure associated with dimensions, and classification hierarchies for dimension values, is also used in SDBs. OLAP operations such as roll-up and drill-down have counterparts in SDBs. Indeed, some implementation techniques developed for OLAP have also been applied to SDBs.

Nonetheless, there are some differences arising from the different domains that OLAP and SDBs were developed to support. For example, SDBs are used in socioeconomic applications, where classification hierarchies and privacy issues are very important. This is reflected in the fact that classification hierarchies in SDBs are more complex than in OLAP and have received more attention, along with issues such as potential breaches of privacy. (The privacy issue concerns whether a user with access to summarized data can reconstruct the original, unsummarized data.) In contrast, OLAP has been aimed at business applications with large volumes of data, and efficient handling of very large datasets has received more attention than in the SDB literature.

23.3.3 Database Design for OLAP

Figure 23.7 shows the tables in our running sales example.

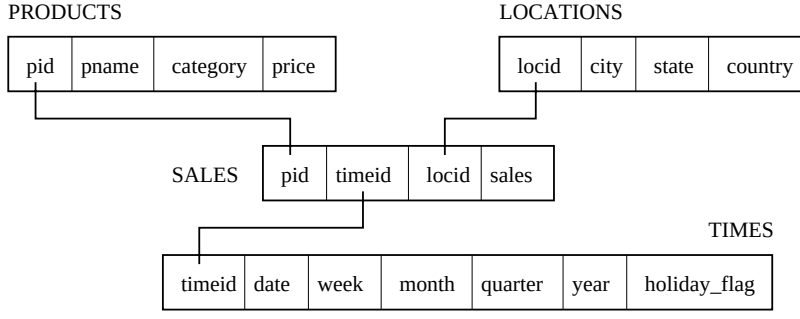


Figure 23.7 An Example of a Star Schema

It suggests a star, centered at the fact table Sales; such a combination of a fact table and dimension tables is called a **star schema**. This schema pattern is very common in databases designed for OLAP. The bulk of the data is typically in the fact table, which has no redundancy; it is usually in BCNF. In fact, to minimize the size of the fact table, dimension identifiers (such as *pid* and *timeid*) are system-generated identifiers.

Information about dimension values is maintained in the dimension tables. Dimension tables are usually not normalized. The rationale is that the dimension tables in a database used for OLAP are static and update, insertion, and deletion anomalies are not important. Further, because the size of the database is dominated by the fact table,

Beyond B+ trees: Complex queries have motivated the addition of powerful indexing techniques to DBMSs. In addition to B+ tree indexes, Oracle 8 supports bitmap and join indexes, and maintains these dynamically as the indexed relations are updated. Oracle 8 also supports indexes on expressions over attribute values, e.g., $10 * sal + bonus$. Microsoft SQL Server uses bitmap indexes. Sybase IQ supports several kinds of bitmap indexes, and may shortly add support for a linear hashing based index. Informix UDS supports R trees and Informix XPS supports bitmap indexes.

the space saved by normalizing dimension tables is negligible. Therefore, minimizing the computation time for combining facts in the fact table with dimension information is the main design criterion, which suggests that we avoid breaking a dimension table into smaller tables (which might lead to additional joins).

Small response times for interactive querying are important in OLAP, and most systems support the materialization of summary tables (typically generated through queries using grouping). Ad hoc queries posed by users are answered using the original tables along with precomputed summaries. A very important design issue is which summary tables should be materialized to achieve the best use of available memory and to answer commonly asked ad hoc queries with interactive response times. In current OLAP systems, deciding which summary tables to materialize may well be the most important design decision.

Finally, new storage structures and indexing techniques have been developed to support OLAP, and they present the database designer with additional physical design choices. We cover some of these implementation techniques briefly in the next section.

23.4 IMPLEMENTATION TECHNIQUES FOR OLAP

In this section we survey some implementation techniques motivated by the OLAP environment. The goal is to provide a feel for how OLAP systems differ from more traditional SQL systems; our discussion is far from comprehensive.

The mostly-read environment of OLAP systems makes the CPU overhead of maintaining indexes negligible, and the requirement of interactive response times for queries over very large datasets makes the availability of suitable indexes very important. This combination of factors has led to the development of new indexing techniques. We discuss several of these techniques. We then consider file organizations and other OLAP implementation issues briefly.

23.4.1 Bitmap Indexes

Consider a table that describes customers:

Customers(*custid:* `integer`, *name:* `string`, *gender:* `boolean`, *rating:* `integer`)

The *rating* value is an integer in the range 1 to 5, and only two values are recorded for *gender*. Columns with few possible values are called **sparse**. We can exploit sparsity to construct a new kind of index that greatly speeds up queries on these columns.

The idea is to record values for sparse columns as a sequence of bits, one for each possible value. For example, a *gender* value is either 10 or 01; a 1 in the first position denotes male, and 1 in the second position denotes female. Similarly, 10000 denotes the *rating* value 1, and 00001 denotes the *rating* value 5.

If we consider the *gender* values for all rows in the Customers table, we can treat this as a collection of two **bit vectors**, one of which has the associated value 'Male' and the other the associated value 'Female'. Each bit vector has one bit per row in the Customers table, indicating whether the value in that row is the value associated with the bit vector. The collection of bit vectors for a column is called a **bitmap index** for that column.

An example instance of the Customers table, together with the bitmap indexes for *gender* and *rating*, is shown in Figure 23.8.

M	*F*
1	0
1	0
0	1
1	0

custid	*name*	*gender*	*rating*
112	Joe	M	3
115	Ram	M	5
119	Sue	F	5
112	Woo	M	4

1	*2*	*3*	*4*	*5*
0	0	1	0	0
0	0	0	0	1
0	0	0	0	1
0	0	0	1	0

Figure 23.8 Bitmap Indexes on the Customers Relation

Bitmap indexes offer two important advantages over conventional hash and tree indexes. First, they allow the use of efficient bit operations to answer queries. For example, consider the query "How many male customers have a rating of 5?" We can take the first bit vector for *gender* and do a bit-wise `AND` with the fifth bit vector for *rating* to obtain a bit vector that has 1 for every male customer with rating 5. We can then count the number of 1s in this bit vector to answer the query. Second, bitmap indexes can be much more compact than a traditional B+ tree index and are very amenable to the use of compression techniques.

Bit vectors correspond closely to the rid-lists used to represent data entries in Alternative (3) for a traditional B+ tree index (see Section 8.3.1). In fact, we can think of a

bit vector for a given *age* value, say, as an alternative representation of the rid-list for that value. This leads to a possible way to combine bit vectors (and their advantages of bit-wise processing) with B+ tree indexes on columns that are not sparse; we can use Alternative (3) for data entries, using a bit vector representation of rid-lists. A caveat is that if an rid-list is very small, the bit vector representation may be much larger than a list of rid values, even if the bit vector is compressed. Further, the use of compression leads to decompression costs, offsetting some of the computational advantages of the bit vector representation.

23.4.2 Join Indexes

Computing joins with small response times is extremely hard for very large relations. One approach to this problem is to create an index that is designed to speed up specific join queries. Suppose that the Customers table is to be joined with a table called Purchases (recording purchases made by customers) on the *custid* field. We can create a collection of $\langle c, p \rangle$ pairs, where p is the rid of a Purchases record that joins with a Customers record with *custid* c.

This idea can be generalized to support joins over more than two relations. We will discuss the special case of a star schema, in which the fact table is likely to be joined with several dimension tables. Consider a join query that joins fact table F with dimension tables D1 and D2 and includes selection conditions on column C_1 of table D1 and column C_2 of table D2. We store a tuple $\langle r_1, r_2, r \rangle$ in the join index if r_1 is the rid of a tuple in table D1 with value c_1 in column C_1, r_2 is the rid of a tuple in table D2 with value c_2 in column C_2, and r is the rid of a tuple in the fact table F, and these three tuples join with each other.

The drawback of a join index is that the number of indexes can grow rapidly if several columns in each dimension table are involved in selections and joins with the fact table. An alternative kind of join index avoids this problem. Consider our example involving fact table F and dimension tables D1 and D2. Let C_1 be a column of D1 on which a selection is expressed in some query that joins D1 with F. Conceptually, we now join F with D1 to extend the fields of F with the fields of D1, and index F on the 'virtual field' C_1: If a tuple of D1 with value c_1 in column C_1 joins with a tuple of F with rid r, we add a tuple $\langle c_1, r \rangle$ to the join index. We create one such join index for each column of either D1 or D2 that involves a selection in some join with F; C_1 is an example of such a column.

The price paid with respect to the previous version of join indexes is that join indexes created in this way have to be combined (rid intersection) in order to deal with the join queries of interest to us. This can be done efficiently if we make the new indexes *bitmap* indexes; the result is called a **bitmapped join index**. The idea works especially well if columns such as C_1 are sparse, and therefore well suited to bitmap indexing.

Complex queries: The IBM DB2 optimizer recognizes star join queries and performs rid-based semijoins (using Bloom filters) to filter the fact table. Then fact table rows are rejoined to the dimension tables. Complex (multi-table) dimension queries (called 'snowflake queries') are supported. DB2 also supports CUBE using smart algorithms that minimize sorts. Microsoft SQL Server optimizes star join queries extensively. It considers taking the cross-product of small dimension tables before joining with the fact table, the use of join indexes, and rid-based semijoins. Oracle 8i also allows users to create dimensions to declare hierarchies and functional dependencies. It supports the CUBE operator and optimizes star join queries by eliminating joins when no column of a dimension table is part of the query result. There are also DBMS products developed specially for decision support applications, such as Sybase IQ and RedBrick (now part of Informix).

23.4.3 File Organizations

Since many OLAP queries involve just a few columns of a large relation, vertical partitioning becomes attractive. However, storing a relation column-wise can degrade performance for queries that involve several columns. An alternative in a mostly-read environment is to store the relation row-wise, but to also store each column separately.

A more radical file organization is to regard the fact table as a large multidimensional array, and to store it and index it as such. This approach is taken in MOLAP systems. Since the array is much larger than available main memory, it is broken up into contiguous chunks, as discussed in Section 25.7. In addition, traditional B+ tree indexes are created to enable quick retrieval of chunks that contain tuples with values in a given range for one or more dimensions.

23.4.4 Additional OLAP Implementation Issues

Our discussion of OLAP implementation techniques is far from complete. A number of other implementation issues must be considered for efficient OLAP query evaluation.

First, the use of compression is becoming widespread in database systems aimed at OLAP. The amount of information is so large that compression is obviously attractive. Further, the use of data structures like bitmap indexes, which are highly amenable to compression techniques, makes compression even more attractive.

Second, deciding which views to precompute and store in order to facilitate evaluation of ad hoc queries is a challenging problem. Especially for aggregate queries, the rich structure of operators such as `CUBE` offers many opportunities for a clever choice of views to precompute and store. Although the choice of views to precompute is made

by the database designer in current systems, ongoing research is aimed at automating this choice.

Third, many OLAP systems are enhancing query language and optimization features in novel ways. As an example of query language enhancement, Redbrick (recently acquired by Informix) supports a version of SQL that allows users to define new aggregation operators by writing code for *initialization*, *iteration*, and *termination*. For example, if tuples with fields *department*, *employee*, and *salary* are retrieved in sorted order by *department*, we can compute the standard deviation of salaries for each department; the initialization function would initialize the variables used to compute standard deviation, the iteration function would update the variables as each tuple is retrieved and processed, and the termination function would output the standard deviation for a department as soon as the first tuple for the next department is encountered. (Several ORDBMSs also support user-defined aggregate functions, and it is likely that this feature will be included in future versions of the SQL standard.) As an example of novel optimization features, some OLAP systems try to combine multiple scans, possibly part of different transactions, over a table. This seemingly simple optimization can be challenging: If the scans begin at different times, for example, we must keep track of the records seen by each scan to make sure that each scan sees every tuple exactly once; we must also deal with differences in the speeds at which the scan operations process tuples.

Finally, we note that the emphasis on query processing and decision support applications in OLAP systems is being complemented by a greater emphasis on evaluating complex SQL queries in traditional SQL systems. Traditional SQL systems are evolving to support OLAP-style queries more efficiently, incorporating techniques previously found only in specialized OLAP systems.

23.5 VIEWS AND DECISION SUPPORT

Views are widely used in decision support applications. Different groups of analysts within an organization are typically concerned with different aspects of the business, and it is convenient to define views that give each group insight into the business details that concern them. Once a view is defined, we can write queries or new view definitions that use it, as we saw in Section 3.6; in this respect a view is just like a base table. Evaluating queries posed against views is very important for decision support applications. In this section, we consider how such queries can be evaluated efficiently after placing views within the context of decision support applications.

23.5.1 Views, OLAP, and Warehousing

Views are closely related to OLAP and data warehousing.

Views and OLAP: OLAP queries are typically aggregate queries. Analysts want fast answers to these queries over very large datasets, and it is natural to consider precomputing views (see Sections 23.5.3 and 23.5.4). In particular, the `CUBE` operator—discussed in Section 23.3.2—gives rise to several aggregate queries that are closely related. The relationships that exist between the many aggregate queries that arise from a single `CUBE` operation can be exploited to develop very effective precomputation strategies. The idea is to choose a subset of the aggregate queries for materialization in such a way that typical `CUBE` queries can be quickly answered by using the materialized views and doing some additional computation. The choice of views to materialize is influenced by how many queries they can potentially speed up and by the amount of space required to store the materialized view (since we have to work with a given amount of storage space).

Views and Warehousing: A data warehouse is just a collection of asynchronously replicated tables and periodically maintained views. A warehouse is characterized by its size, the number of tables involved, and the fact that most of the underlying tables are from external, independently maintained databases. Nonetheless, the fundamental problem in warehouse maintenance is asynchronous maintenance of replicated tables and materialized views (see Section 23.5.4).

23.5.2 Query Modification

Consider the view RegionalSales, defined below, which computes sales of products by category and state:

```
CREATE VIEW RegionalSales (category, sales, state)
        AS SELECT P.category, S.sales, L.state
           FROM   Products P, Sales S, Locations L
           WHERE  P.pid = S.pid AND S.locid = L.locid
```

The following query computes the total sales for each category by state:

```
SELECT   R.category, R.state, SUM (R.sales)
FROM     RegionalSales R
GROUP BY R.category, R.state
```

While the SQL-92 standard does not specify how to evaluate queries on views, it is useful to think in terms of a process called **query modification**. The idea is to replace the occurrence of RegionalSales in the query by the view definition. The result on the above query is:

```
SELECT   R.category, R.state, SUM (R.sales)
FROM     ( SELECT P.category, S.sales, L.state
```

```
            FROM    Products P, Sales S, Locations L
            WHERE   P.pid = S.pid AND S.locid = L.locid ) AS R
GROUP BY R.category, R.state
```

23.5.3 View Materialization versus Computing on Demand

We can answer a query on a view by evaluating the modified query constructed using the query modification technique described above. Often, however, queries against complex view definitions must be answered very fast because users engaged in decision support activities require interactive response times. Even with sophisticated optimization and evaluation techniques, there is a limit to how fast we can answer such queries.

A popular approach to dealing with this problem is to evaluate the view definition and store the result. When a query is now posed on the view, the (unmodified) query is executed directly on the precomputed result. This approach is called **view materialization** and is likely to be much faster than the query modification approach because the complex view need not be evaluated when the query is computed. The drawback, of course, is that we must maintain the consistency of the precomputed (or *materialized*) view whenever the underlying tables are updated.

Consider the RegionalSales view. It involves a join of Sales, Products, and Locations and is likely to be expensive to compute. On the other hand, if it is materialized and stored with a clustered B+ tree index on the composite search key ⟨category, state, sales⟩, we can answer the example query by an index-only scan.

Given the materialized view and this index, we can also answer queries of the following form efficiently:

```
SELECT    R.state, SUM (R.sales)
FROM      RegionalSales R
WHERE     R.category = 'Laptop'
GROUP BY R.state
```

To answer such a query, we can use the index on the materialized view to locate the first index leaf entry with *category* = 'Laptop' and then scan the leaf level until we come to the first entry with *category* not equal to 'Laptop.'

The given index is less effective on the following query, for which we are forced to scan the entire leaf level:

```
SELECT    R.state, SUM (R.sales)
FROM      RegionalSales R
```

```
WHERE     R.state = 'Wisconsin'
GROUP BY  R.category
```

This example indicates how the choice of views to materialize and the indexes to create are affected by the expected workload. This point is illustrated further by our next example.

Consider the following two queries:

```
SELECT    P.category, SUM (S.sales)
FROM      Products P, Sales S
WHERE     P.pid = S.pid
GROUP BY  P.category

SELECT    L.state, SUM (S.sales)
FROM      Locations L, Sales S
WHERE     L.locid = S.locid
GROUP BY  L.state
```

The above two queries require us to join the Sales table (which is likely to be very large) with another table and to then aggregate the result. How can we use materialization to speed these queries up? The straightforward approach is to precompute each of the joins involved (Products with Sales and Locations with Sales) or to precompute each query in its entirety. An alternative approach is to define the following view:

```
CREATE    VIEW TotalSales (pid, locid, total)
          AS SELECT    S.pid, S.locid, SUM (S.sales)
             FROM      Sales S
             GROUP BY  S.pid, S.locid
```

The view TotalSales can be materialized and used instead of Sales in our two example queries:

```
SELECT    P.category, SUM (T.total)
FROM      Products P, TotalSales T
WHERE     P.pid = T.pid
GROUP BY  P.category

SELECT    L.state, SUM (T.total)
FROM      Locations L, TotalSales T
WHERE     L.locid = T.locid
GROUP BY  L.state
```

23.5.4 Issues in View Materialization

There are three main questions to consider with regard to view materialization:

1. What views should we materialize and what indexes should we build on the materialized views?
2. Given a query on a view and a set of materialized views, can we exploit the materialized views to answer the query?
3. How frequently should we refresh materialized views in order to make them consistent with changes to the underlying tables?

As the example queries using TotalSales illustrated, the answers to the first two questions are related. The choice of views to materialize and index is governed by the expected workload, and the discussion of indexing in Chapter 16 is relevant to this question as well. The choice of views to materialize is more complex than just choosing indexes on a set of database tables, however, because the range of alternative views to materialize is wider. The goal is to materialize a small, carefully chosen set of views that can be utilized to quickly answer most of the important queries. Conversely, once we have chosen a set of views to materialize, we have to consider how they can be used to answer a given query.

A materialized view is said to be **refreshed** when we make it consistent with changes to its underlying tables. Ideally, algorithms for refreshing a view should be **incremental** in that the cost is proportional to the extent of the change, rather than the cost of recomputing the view from scratch. While it is usually possible to incrementally refresh views when new tuples are added to the underlying tables, incremental refreshing is harder when tuples are deleted from the underlying tables.

A **view maintenance policy** is a decision about when a view is refreshed and is independent of whether the refresh is incremental or not. A view can be refreshed within the same transaction that updates the underlying tables. This is called **immediate view maintenance**. The update transaction is slowed by the refresh step, and the impact of refresh increases with the number of materialized views that depend upon the updated table.

Alternatively, we can defer refreshing the view. Updates are captured in a log and applied subsequently to the materialized views. There are several **deferred view maintenance policies**:

1. **Lazy:** The materialized view V is refreshed at the time a query is evaluated using V, if V is not already consistent with its underlying base tables. This approach slows down queries rather than updates, in contrast to immediate view maintenance.

Views for decision support: DBMS vendors are enhancing their main relational products to support decision support queries. IBM DB2 supports materialized views with transaction-consistent or user-invoked maintenance. Microsoft SQL Server supports **partition views**, which are unions of (many) horizontal partitions of a table. These are aimed at a warehousing environment where each partition could be, for example, a monthly update. Queries on partition views are optimized so that only relevant partitions are accessed. Oracle 8i supports materialized views with transaction-consistent, user-invoked, or time-scheduled maintenance.

2. **Periodic:** The materialized view is refreshed periodically, e.g., once a day. The discussion of the Capture and Apply steps in asynchronous replication (see Section 21.10.2) should be reviewed at this point since it is very relevant to periodic view maintenance. In fact, many vendors are extending their asynchronous replication features to support materialized views. Materialized views that are refreshed periodically are also called **snapshots**.

3. **Forced:** The materialized view is refreshed after a certain number of changes have been made to the underlying tables.

In periodic and forced view maintenance, queries may see an instance of the materialized view that is not consistent with the current state of the underlying tables. That is, the queries would see a different set of tuples if the view definition was recomputed. This is the price paid for fast updates and queries, and the trade-off is similar to the trade-off made in using asynchronous replication.

23.6 FINDING ANSWERS QUICKLY

A recent trend, fueled in part by the popularity of the Internet, is an emphasis on queries for which a user wants only the first few, or the 'best' few, answers quickly. When users pose queries to a search engine such as AltaVista, they rarely look beyond the first or second page of results. If they do not find what they are looking for, they refine their query and resubmit it. The same phenomenon is being observed in decision support applications, and some DBMS products (e.g., DB2) already support extended SQL constructs to specify such queries. A related trend is that for complex queries, users would like to see an approximate answer quickly and then have it be continually refined, rather than wait until the exact answer is available. We now discuss these two trends briefly.

23.6.1 Top N Queries

An analyst often wants to identify the top-selling handful of products, for example. We can sort by sales for each product and return answers in this order. If we have a million products and the analyst is only interested in the top 10, this straightforward evaluation strategy is clearly wasteful. Thus, it is desirable for users to be able to explicitly indicate how many answers they want, making it possible for the DBMS to optimize execution. The example query below asks for the top 10 products ordered by sales in a given location and time:

```
SELECT    P.pid, P.pname, S.sales
FROM      Sales S, Products P
WHERE     S.pid=P.pid AND S.locid=1 AND S.timeid=3
ORDER BY  S.sales DESC
OPTIMIZE FOR 10 ROWS
```

The `OPTIMIZE FOR` N `ROWS` construct is not in SQL-92 (or even SQL:1999), but it is supported in IBM's DB2 product, and other products (e.g., Oracle 7) have similar constructs. In the absence of a cue such as `OPTIMIZE FOR 10 ROWS`, the DBMS computes sales for all products and returns them in descending order by sales. The application can close the result cursor (i.e., terminate the query execution) after consuming 10 rows, but considerable effort has already been expended in computing sales for all products and sorting them.

Now let us consider how a DBMS can make use of the `OPTIMIZE FOR` cue to execute the query efficiently. The key is to somehow compute sales only for products that are likely to be in the top 10 by sales. Suppose that we know the distribution of sales values because we maintain a histogram on the *sales* column of the Sales relation. We can then choose a value of *sales*, say c, such that only 10 products have a larger sales value. For those Sales tuples that meet this condition, we can apply the location and time conditions as well and sort the result. Evaluating the following query is equivalent to this approach:

```
SELECT    P.pid, P.pname, S.sales
FROM      Sales S, Products P
WHERE     S.pid=P.pid AND S.locid=1 AND S.timeid=3 AND S.sales > c
ORDER BY  S.sales DESC
```

This approach is, of course, much faster than the alternative of computing all product sales and sorting them, but there are some important problems to resolve:

1. *How do we choose the sales cutoff value c?* Histograms and other system statistics can be used for this purpose, but this can be a tricky issue. For one thing, the statistics maintained by a DBMS are only approximate. For another, even if we

choose the cutoff to reflect the top 10 sales values accurately, other conditions in the query may eliminate some of the selected tuples, leaving us with fewer than 10 tuples in the result.

2. *What if we have more than 10 tuples in the result?* Since the choice of the cutoff c is approximate, we could get more than the desired number of tuples in the result. This is easily handled by returning just the top 10 to the user. We have still saved considerably with respect to the approach of computing sales for all products, thanks to the conservative pruning of irrelevant sales information, using the cutoff c.

3. *What if we have fewer than 10 tuples in the result?* Even if we choose the sales cutoff c conservatively, there is the possibility that we compute fewer than 10 result tuples. In this case, we can re-execute the query with a smaller cutoff value c_2, or simply re-execute the original query with no cutoff.

The effectiveness of the approach depends on how well we can estimate the cutoff, and in particular, on minimizing the number of times we obtain fewer than the desired number of result tuples.

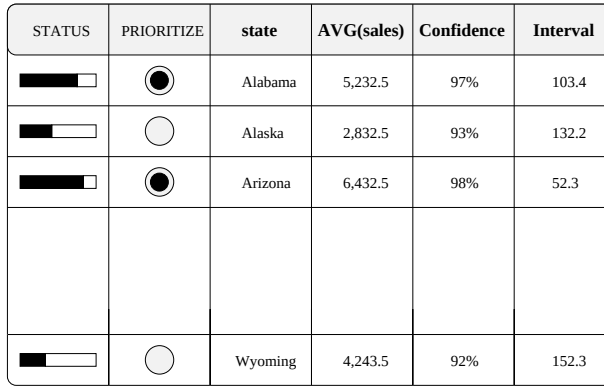
23.6.2 Online Aggregation

Consider the following query, which asks for the average sales amount by state:

```
SELECT    L.state, AVG (S.sales)
FROM      Sales S, Locations L
WHERE     S.locid=L.locid
GROUP BY L.state
```

This can be an expensive query if Sales and Locations are large relations, and we cannot achieve fast response times with the traditional approach of computing the anwer in its entirety when the query is presented. One alternative, as we have seen, is to use precomputation. Another alternative is to compute the answer to the query when the query is presented, but to return an approximate answer to the user as soon as possible. As the computation progresses, the answer quality is continually refined. This approach is called **online aggregation**. It is very attractive for queries involving aggregation, because efficient techniques for computing and refining approximate answers are available.

Online aggregation is illustrated in Figure 23.9: For each state—the grouping criterion for our example query—the current value for average sales is displayed, together with a confidence interval. The entry for Alaska tells us that the current estimate of average per-store sales in Alaska is $2,832.50, and that this is within the range $2,700.30 and $2,964.70 with 93 percent probability. The status bar in the first column indicates how



STATUS	PRIORITIZE	state	AVG(sales)	Confidence	Interval
		Alabama	5,232.5	97%	103.4
		Alaska	2,832.5	93%	132.2
		Arizona	6,432.5	98%	52.3
		Wyoming	4,243.5	92%	152.3

Figure 23.9 Online Aggregation

close we are to arriving at an exact value for the average sales, and the second column indicates whether calculating the average sales for this state is a priority. Estimating average sales for Alaska is not a priority, but estimating it for Arizona is a priority. As the figure indicates, the DBMS devotes more system resources to estimating the average sales for prioritized states; the estimate for Arizona is much tighter than that for Alaska, and holds with a higher probability. Users can set the priority for a state by clicking on the priority button at any time during the execution. This degree of interactivity, together with the continuous feedback provided by the visual display, makes online aggregation an attractive technique.

In order to implement online aggregation, a DBMS must incorporate statistical techniques to provide confidence intervals for approximate answers and use **nonblocking algorithms** for the relational operators. An algorithm is said to block if it does not produce output tuples until it has consumed all of its input tuples. For example, the sort-merge join algorithm blocks because sorting requires all input tuples before determining the first output tuple. Nested loops join and hash join are therefore preferable to sort-merge join for online aggregation. Similarly, hash-based aggregation is better than sort-based aggregation.

23.7 POINTS TO REVIEW

- A *data warehouse* contains consolidated data drawn from several different databases together with historical and summary information. *Online analytic processing (OLAP)* applications and *data mining* applications generate complex queries that make traditional SQL systems inadequate. Such applications support high-level decision making and are also called *decision support applications.* **(Section 23.1)**

- Information about daily operations of an organization is stored in *operational databases*. This data is *extracted* through *gateways*, then *cleaned* and *transformed* before *loading* it into the data warehouse. Data in the data warehouse is periodically *refreshed* to reflect updates, and it is periodically *purged* to delete outdated information. The system catalogs of the data warehouse can be very large and are managed in a separate database called the *metadata repository*. **(Section 23.2)**

- The multidimensional data model consists of *measures* and *dimensions*. The relation that relates the dimensions to the measures is called the *fact table*. OLAP systems that store multidimensional datasets as arrays are called *multidimensional OLAP (MOLAP)* systems. OLAP systems that store the data in relations are called *relational OLAP (ROLAP)* systems. Common OLAP operations have received special names: roll-up, drill-down, pivoting, slicing, and dicing. Databases designed for OLAP queries commonly arrange the fact and dimension tables in a *star schema*. **(Section 23.3)**

- Index structures that are especially suitable for OLAP systems include *bitmap indexes* and *join indexes*. **(Section 23.4)**

- Views are widely used in decision support applications. Since decision support systems require fast response times for interactive queries, queries involving views must be evaluated very efficiently. Views can either be materialized or computed on demand. We say that a materialized view is *refreshed* when we make it consistent with changes to the underlying tables. An algorithm for refreshing a view is *incremental* if the update cost is proportional to the amount of change at the base tables. A *view maintenance policy* determines when a view is refreshed. In *immediate view maintenance* the view is updated within the same transaction that modifies the underlying tables; otherwise the policy is said to be *deferred view maintenance*. Deferred view maintenance has three variants: In *lazy* maintenance we refresh the view at query time. In *periodic* maintenance we refresh the view periodically; such views are also called *snapshots*. In *forced* maintenance we refresh the view after a certain number of changes have been made to the base tables. **(Section 23.5)**

- New query paradigms include *top N queries* and *online aggregation*. In top N queries we only want to retrieve the first N rows of the query result. An online aggregation query returns an approximate answer to an aggregation query immediately and refines the answer progressively. **(Section 23.6)**

EXERCISES

Exercise 23.1 Briefly answer the following questions.

1. How do warehousing, OLAP, and data mining complement each other?

2. What is the relationship between data warehousing and data replication? Which form of replication (synchronous or asynchronous) is better suited for data warehousing? Why?
3. What is the role of the metadata repository in a data warehouse? How does it differ from a catalog in a relational DBMS?
4. What are the considerations in designing a data warehouse?
5. Once a warehouse is designed and loaded, how is it kept current with respect to changes to the source databases?
6. One of the advantages of a warehouse is that we can use it to track how the contents of a relation change over time; in contrast, we have only the current snapshot of a relation in a regular DBMS. Discuss how you would maintain the history of a relation R, taking into account that 'old' information must somehow be purged to make space for new information.
7. Describe dimensions and measures in the multidimensional data model.
8. What is a fact table, and why is it so important from a performance standpoint?
9. What is the fundamental difference between MOLAP and ROLAP systems?
10. What is a star schema? Is it typically in BCNF? Why or why not?
11. How is data mining different from OLAP?

Exercise 23.2 Consider the instance of the Sales relation shown in Figure 23.3.

1. Show the result of pivoting the relation on *pid* and *timeid*.
2. Write a collection of SQL queries to obtain the same result as in the previous part.
3. Show the result of pivoting the relation on *pid* and *locid*.

Exercise 23.3 Consider the cross-tabulation of the Sales relation shown in Figure 23.5.

1. Show the result of roll-up on *locid* (i.e., state).
2. Write a collection of SQL queries to obtain the same result as in the previous part.
3. Show the result of roll-up on *locid* followed by drill-down on *pid*.
4. Write a collection of SQL queries to obtain the same result as in the previous part, starting with the cross-tabulation shown in Figure 23.5.

Exercise 23.4 Consider the Customers relation and the bitmap indexes shown in Figure 23.8.

1. For the same data, if the underlying set of rating values is assumed to range from 1 to 10, show how the bitmap indexes would change.
2. How would you use the bitmap indexes to answer the following queries? If the bitmap indexes are not useful, explain why.
   (a) How many customers with a rating less than 3 are male?
   (b) What percentage of customers are male?
   (c) How many customers are there?

(d) How many customers are named Woo?

(e) Find the rating value with the greatest number of customers and also find the number of customers with that rating value; if several rating values have the maximum number of customers, list the requested information for all of them. (Assume that very few rating values have the same number of customers.)

Exercise 23.5 In addition to the Customers table of Figure 23.8 with bitmap indexes on *gender* and *rating*, assume that you have a table called Prospects, with fields *rating* and *prospectid*. This table is used to identify potential customers.

1. Suppose that you also have a bitmap index on the *rating* field of Prospects. Discuss whether or not the bitmap indexes would help in computing the join of Customers and Prospects on *rating*.
2. Suppose that you do *not* have a bitmap index on the *rating* field of Prospects. Discuss whether or not the bitmap indexes on Customers would help in computing the join of Customers and Prospects on *rating*.
3. Describe the use of a join index to support the join of these two relations with the join condition *custid=prospectid*.

Exercise 23.6 Consider the instances of the Locations, Products, and Sales relations shown in Figure 23.3.

1. Consider the basic join indexes described in Section 23.4.2. Suppose you want to optimize for the following two kinds of queries: Query 1 finds sales in a given city, and Query 2 finds sales in a given state. Show the indexes that you would create on the example instances shown in Figure 23.3.
2. Consider the bitmapped join indexes described in Section 23.4.2. Suppose you want to optimize for the following two kinds of queries: Query 1 finds sales in a given city, and Query 2 finds sales in a given state. Show the indexes that you would create on the example instances shown in Figure 23.3.
3. Consider the basic join indexes described in Section 23.4.2. Suppose you want to optimize for the following two kinds of queries: Query 1 finds sales in a given city for a given product name, and Query 2 finds sales in a given state for a given product category. Show the indexes that you would create on the example instances shown in Figure 23.3.
4. Consider the bitmapped join indexes described in Section 23.4.2. Suppose you want to optimize for the following two kinds of queries: Query 1 finds sales in a given city for a given product name, and Query 2 finds sales in a given state for a given product category. Show the indexes that you would create on the example instances shown in Figure 23.3.

Exercise 23.7 Consider the view NumReservations defined below:

```
CREATE VIEW NumReservations (sid, sname, numres)
       AS SELECT S.sid, S.sname, COUNT (*)
          FROM     Sailors S, Reserves R
          WHERE    S.sid = R.sid
          GROUP BY S.sid, S.sname
```

1. How is the following query, which is intended to find the highest number of reservations made by some one sailor, rewritten using query modification?

```
SELECT   MAX (N.numres)
FROM     NumReservations N
```

2. Consider the alternatives of computing on demand and view materialization for the above query. Discuss the pros and cons of materialization.
3. Discuss the pros and cons of materialization for the following query:

```
SELECT   N.sname, MAX (N.numres)
FROM     NumReservations N
GROUP BY N.sname
```

BIBLIOGRAPHIC NOTES

A good survey of data warehousing and OLAP is presented in [137], which is the source of Figure 23.1. [597] provides an overview of OLAP and statistical database research, showing the strong parallels between concepts and research in these two areas. The book by Kimball [374], one of the pioneers in warehousing, and the collection of papers [51] offer a good practical introduction to the area. The term OLAP was popularized by Codd's paper [160]. For a recent discussion of the performance of algorithms utilizing bitmap and other nontraditional index structures, see [500].

[624] discusses how queries on views can be converted to queries on the underlying tables through query modification. [308] compares the performance of query modification versus immediate and deferred view maintenance. [618] presents an analytical model of materialized view maintenance algorithms. A number of papers discuss how materialized views can be incrementally maintained as the underlying relations are changed. This area has become very active recently, in part because of the interest in *data warehouses*, which can be thought of as collections of views over relations from various sources. An excellent overview of the state of the art can be found in [293], which contains a number of influential papers together with additional material that provides context and background. The following partial list should provide pointers for further reading: [87, 161, 162, 294, 312, 498, 524, 553, 577, 616, 700].

[285] introduced the CUBE operator, and optimization of CUBE queries and efficient maintenance of the result of a CUBE query have been addressed in several papers, including [9, 81, 182, 310, 320, 389, 552, 556, 598, 699]. Related algorithms for processing queries with aggregates and grouping are presented in [136, 139]. [538] addresses the implementation of queries involving generalized quantifiers such as *a majority of*. [619] describes an access method to support processing of aggregate queries.

[114, 115] discuss how to evaluate queries for which only the first few answers are desired.L [192] considers how a probabilistic approach to query optimization can be applied to this problem. [316, 40] discuss how to return approximate answers to aggregate queries and to refine them 'online.'