20.4 On-Line Analytic Processing

We shall now take up an important class of applications for integrated information systems, especially data warehouses. Companies and organizations create a warehouse with a copy of large amounts of their available data and assign analysts to query this warehouse for patterns or trends of importance to the organization. This activity, called *OLAP* (standing for *On-Line Analytic Processing* and pronounced "oh-lap"), generally involves highly complex queries that use one or more aggregations. These queries are often termed *OLAP queries* or *decision-support queries*. Some examples will be given in Section 20.4.1; a typical example is to search for products with increasing or decreasing overall sales.

Decision-support queries used in OLAP applications typically examine very large amounts of data, even if the query results are small. In contrast, common database operations, such as bank deposits or airline reservations, each touch only a tiny portion of the database; the latter type of operation is often referred to as *OLTP* (*On-Line Transaction Processing*, spoken "oh-ell-tee-pee").

Recently, new query-processing techniques have been developed that are especially good at executing OLAP queries effectively. Furthermore, because of the distinct nature of a certain class of OLAP queries, special forms of DBMS's have been developed and marketed to support OLAP applications. The same technology is beginning to migrate to standard SQL systems, as well. We shall discuss the architecture of these systems in Section 20.5.

20.4.1 OLAP Applications

A common OLAP application uses a warehouse of sales data. Major store chains will accumulate terabytes of information representing every sale of every item at every store. Queries that aggregate sales into groups and identify significant groups can be of great use to the company in predicting future problems and opportunities.

Example 20.27: Suppose the Aardvark Automobile Co. builds a data warehouse to analyze sales of its cars. The schema for the warehouse might be:

```
Sales(serialNo, date, dealer, price)
Autos(serialNo, model, color)
Dealers(name, city, state, phone)
```

A typical decision-support query might examine sales on or after April 1, 2001 to see how the recent average price per vehicle varies by state. Such a query is shown in Fig. 20.19.

Notice how the query of Fig. 20.19 touches much of the data of the database, as it classifies every recent `Sales` fact by the state of the dealer that sold it. In contrast, common OLTP queries, such as "find the price at which the auto with serial number 123 was sold," would touch only a single tuple of the data. □

Warehouses and OLAP

There are several reasons why data warehouses play an important role in OLAP applications. First, the warehouse may be necessary to organize and centralize corporate data in a way that supports OLAP queries; the data may initially be scattered across many different databases. But often more important is the fact that OLAP queries, being complex and touching much of the data, take too much time to be executed in a transaction-processing system with high throughput requirements. OLAP queries often can be considered "long transactions" in the sense of Section 19.7.

Long transactions locking the entire database would shut down the ordinary OLTP operations (e.g., recording new sales as they occur could not be permitted if there were a concurrent OLAP query computing average sales). A common solution is to make a copy of the raw data in a warehouse, run OLAP queries only at the warehouse, and run the OLTP queries and data modifications at the data sources. In a common scenario, the warehouse is only updated overnight, while the analysts work on a frozen copy during the day. The warehouse data thus gets out of date by as much as 24 hours, which limits the timeliness of its answers to OLAP queries, but the delay is tolerable in many decision-support applications.

For another OLAP example, consider a credit-card company trying to decide whether applicants for a card are likely to be credit-worthy. The company creates a warehouse of all its current customers and their payment history. OLAP queries search for factors, such as age, income, home-ownership, and zip-code, that might help predict whether customers will pay their bills on time. Similarly, hospitals may use a warehouse of patient data — their admissions, tests administered, outcomes, diagnoses, treatments, and so on — to analyze for risks and select the best modes of treatment.

```
SELECT state, AVG(price)
FROM Sales, Dealers
WHERE Sales.dealer = Dealers.name AND
      date >= '2001-01-04'
GROUP BY state;
```

Figure 20.19: Find average sales price by state

20.4.2 A Multidimensional View of OLAP Data

In typical OLAP applications there is a central relation or collection of data, called the *fact table*. A fact table represents events or objects of interest, such as sales in Example 20.27. Often, it helps to think of the objects in the fact table as arranged in a multidimensional space, or "cube." Figure 20.20 suggests three-dimensional data, represented by points within the cube; we have called the dimensions car, dealer, and date, to correspond to our earlier example of automobile sales. Thus, in Fig. 20.20 we could think of each point as a sale of a single automobile, while the dimensions represent properties of that sale.

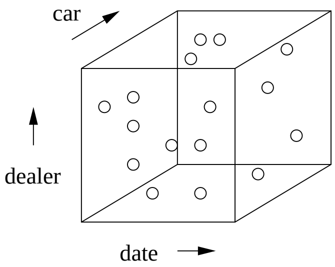

Figure 20.20: Data organized in a multidimensional space

A data space such as Fig. 20.20 will be referred to informally as a "data cube," or more precisely as a *raw-data cube* when we want to distinguish it from the more complex "data cube" of Section 20.5. The latter, which we shall refer to as a *formal* data cube when a distinction from the raw-data cube is needed, differs from the raw-data cube in two ways:

1. It includes aggregations of the data in all subsets of dimensions, as well as the data itself.

2. Points in the formal data cube may represent an initial aggregation of points in the raw-data cube. For instance, instead of the "car" dimension representing each individual car (as we suggested for the raw-data cube), that dimension might be aggregated by model only, and a point of a formal data cube could represent the total sales of all cars of a given model by a given dealer on a given day.

The distinctions between the raw-data cube and the formal data cube are reflected in the two broad directions that have been taken by specialized systems that support cube-structured data for OLAP:

1. *ROLAP*, or *Relational OLAP*. In this approach, data may be stored in relations with a specialized structure called a "star schema," described in Section 20.4.3. One of these relations is the "fact table," which contains the *raw*, or unaggregated, data, and corresponds to what we called the raw-data cube. Other relations give information about the values along

each dimension. The query language and other capabilities of the system may be tailored to the assumption that data is organized this way.

2. *MOLAP*, or *Multidimensional OLAP*. Here, a specialized structure, the formal "data cube" mentioned above, is used to hold the data, including its aggregates. Nonrelational operators may be implemented by the system to support OLAP queries on data in this structure.

20.4.3 Star Schemas

A *star schema* consists of the schema for the fact table, which links to several other relations, called "dimension tables." The fact table is at the center of the "star," whose points are the dimension tables. A fact table normally has several attributes that represent *dimensions*, and one or more *dependent* attributes that represent properties of interest for the point as a whole. For instance, dimensions for sales data might include the date of the sale, the place (store) of the sale, the type of item sold, the method of payment (e.g., cash or a credit card), and so on. The dependent attribute(s) might be the sales price, the cost of the item, or the tax, for instance.

Example 20.28: The `Sales` relation from Example 20.27

```
Sales(serialNo, date, dealer, price)
```

is a fact table. The dimensions are:

1. `serialNo`, representing the automobile sold, i.e., the position of the point in the space of possible automobiles.

2. `date`, representing the day of the sale, i.e., the position of the event in the time dimension.

3. `dealer`, representing the position of the event in the space of possible dealers.

The one dependent attribute is `price`, which is what OLAP queries to this database will typically request in an aggregation. However, queries asking for a count, rather than sum or average price would also make sense, e.g., "list the total number of sales for each dealer in the month of May, 2001." □

Supplementing the fact table are *dimension tables* describing the values along each dimension. Typically, each dimension attribute of the fact table is a foreign key, referencing the key of the corresponding dimension table, as suggested by Fig. 20.21. The attributes of the dimension tables also describe the possible groupings that would make sense in an SQL `GROUP BY` query. An example should make the ideas clearer.

Example 20.29: For the automobile data of Example 20.27, two of the three dimension tables are obvious:

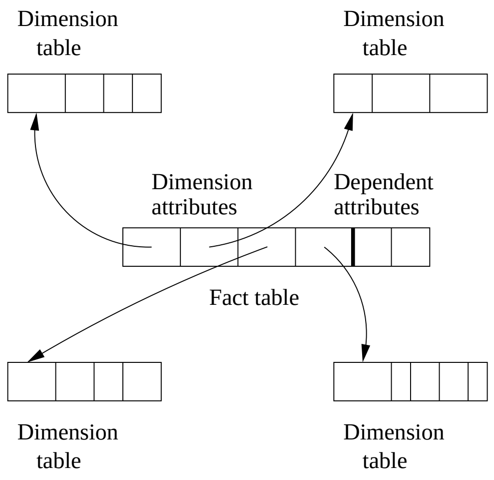

Figure 20.21: The dimension attributes in the fact table reference the keys of the dimension tables

```
Autos(serialNo, model, color)
Dealers(name, city, state, phone)
```

Attribute `serialNo` in the fact table

```
Sales(serialNo, date, dealer, price)
```

is a foreign key, referencing `serialNo` of dimension table `Autos`.[7] The attributes `Autos.model` and `Autos.color` give properties of a given auto. We could have added many more attributes in this relation, such as boolean attributes indicating whether the auto has an automatic transmission. If we join the fact table `Sales` with the dimension table `Autos`, then the attributes `model` and `color` may be used for grouping sales in interesting ways. For instance, we can ask for a breakdown of sales by color, or a breakdown of sales of the Gobi model by month and dealer.

Similarly, attribute `dealer` of `Sales` is a foreign key, referencing `name` of the dimension table `Dealers`. If `Sales` and `Dealers` are joined, then we have additional options for grouping our data; e.g., we can ask for a breakdown of sales by state or by city, as well as by dealer.

One might wonder where the dimension table for time (the `date` attribute of `Sales`) is. Since time is a physical property, it does not make sense to store facts about time in a database, since we cannot change the answer to questions such as "in what year does the day July 5, 2000 appear?" However, since grouping by various time units, such as weeks, months, quarters, and years, is frequently

[7] It happens that `serialNo` is also a key for the `Sales` relation, but there need not be an attribute that is both a key for the fact table and a foreign key for some dimension table.

desired by analysts, it helps to build into the database a notion of time, as if there were a time dimension table such as

```
Days(day, week, month, year)
```

A typical tuple of this "relation" would be

```
(5, 27, 7, 2000)
```

representing July 5, 2000. The interpretation is that this day is the fifth day of the seventh month of the year 2000; it also happens to fall in the 27th full week of the year 2000. There is a certain amount of redundancy, since the week is calculable from the other three attributes. However, weeks are not exactly commensurate with months, so we cannot obtain a grouping by months from a grouping by weeks, or vice versa. Thus, it makes sense to imagine that both weeks and months are represented in this "dimension table." □

20.4.4 Slicing and Dicing

We can think of the points of the raw-data cube as partitioned along each dimension at some level of granularity. For example, in the time dimension, we might partition ("group by" in SQL terms) according to days, weeks, months, years, or not partition at all. For the cars dimension, we might partition by model, by color, by both model and color, or not partition. For dealers, we can partition by dealer, by city, by state, or not partition.

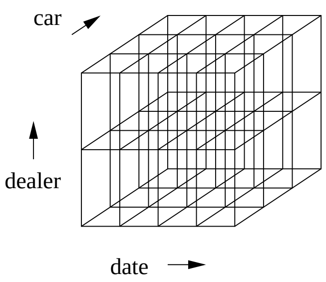

Figure 20.22: Dicing the cube by partitioning along each dimension

A choice of partition for each dimension "dices" the cube, as suggested by Fig. 20.22. The result is that the cube is divided into smaller cubes that represent groups of points whose statistics are aggregated by a query that performs the partitioning in its `GROUP BY` clause. Through the `WHERE` clause, a query also has the option of focusing on particular partitions along one or more dimensions (i.e., on a particular "slice" of the cube).

Example 20.30: Figure 20.23 suggests a query in which we ask for a slice in one dimension (the date), and dice in two other dimensions (car and dealer).

The date is divided into four groups, perhaps the four years over which data has been accumulated. The shading in the diagram suggests that we are only interested in one of these years.

The cars are partitioned into three groups, perhaps sedans, SUV's, and convertibles, while the dealers are partitioned into two groups, perhaps the eastern and western regions. The result of the query is a table giving the total sales in six categories for the one year of interest. □

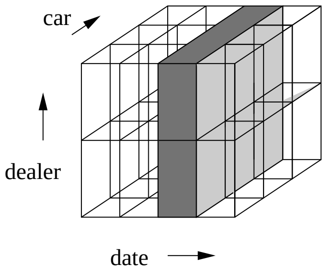

Figure 20.23: Selecting a slice of a diced cube

The general form of a so-called "slicing and dicing" query is thus:

```
SELECT grouping attributes and aggregations
FROM fact table joined with zero or more dimension tables
WHERE certain attributes are constant
GROUP BY grouping attributes;
```

Example 20.31: Let us continue with our automobile example, but include the conceptual `Days` dimension table for time discussed in Example 20.29. If the Gobi isn't selling as well as we thought it would, we might try to find out which colors are not doing well. This query uses only the `Autos` dimension table and can be written in SQL as:

```
SELECT color, SUM(price)
FROM Sales NATURAL JOIN Autos
WHERE model = 'Gobi'
GROUP BY color;
```

This query dices by color and then slices by model, focusing on a particular model, the Gobi, and ignoring other data.

Suppose the query doesn't tell us much; each color produces about the same revenue. Since the query does not partition on time, we only see the total over all time for each color. We might suppose that the recent trend is for one or more colors to have weak sales. We may thus issue a revised query that also partitions time by month. This query is:

```
SELECT color, month, SUM(price)
FROM (Sales NATURAL JOIN Autos) JOIN Days ON date = day
WHERE model = 'Gobi'
GROUP BY color, month;
```

It is important to remember that the `Days` relation is not a conventional stored relation, although we may treat it as if it had the schema

```
Days(day, week, month, year)
```

The ability to use such a "relation" is one way that a system specialized to OLAP queries could differ from a conventional DBMS.

We might discover that red Gobis have not sold well recently. The next question we might ask is whether this problem exists at all dealers, or whether only some dealers have had low sales of red Gobis. Thus, we further focus the query by looking at only red Gobis, and we partition along the dealer dimension as well. This query is:

```
SELECT dealer, month, SUM(price)
FROM (Sales NATURAL JOIN Autos) JOIN Days ON date = day
WHERE model = 'Gobi' AND color = 'red'
GROUP BY month, dealer;
```

At this point, we find that the sales per month for red Gobis are so small that we cannot observe any trends easily. Thus, we decide that it was a mistake to partition by month. A better idea would be to partition only by years, and look at only the last two years (2001 and 2002, in this hypothetical example). The final query is shown in Fig. 20.24. □

```
SELECT dealer, year, SUM(price)
FROM (Sales NATURAL JOIN Autos) JOIN Days ON date = day
WHERE model = 'Gobi' AND
      color = 'red' AND
      (year = 2001 OR year = 2002)
GROUP BY year, dealer;
```

Figure 20.24: Final slicing-and-dicing query about red Gobi sales

20.4.5 Exercises for Section 20.4

* **Exercise 20.4.1:** An on-line seller of computers wishes to maintain data about orders. Customers can order their PC with any of several processors, a selected amount of main memory, any of several disk units, and any of several CD or DVD readers. The fact table for such a database might be:

Drill-Down and Roll-Up

Example 20.31 illustrates two common patterns in sequences of queries that slice-and-dice the data cube.

1. *Drill-down* is the process of partitioning more finely and/or focusing on specific values in certain dimensions. Each of the steps except the last in Example 20.31 is an instance of drill-down.

2. *Roll-up* is the process of partitioning more coarsely. The last step, where we grouped by years instead of months to eliminate the effect of randomness in the data, is an example of roll-up.

```
Orders(cust, date, proc, memory, hd, rd, quant, price)
```

We should understand attribute `cust` to be an ID that is the foreign key for a dimension table about customers, and understand attributes `proc`, `hd` (hard disk), and `rd` (removable disk: CD or DVD, typically) similarly. For example, an `hd` ID might be elaborated in a dimension table giving the manufacturer of the disk and several disk characteristics. The `memory` attribute is simply an integer: the number of megabytes of memory ordered. The `quant` attribute is the number of machines of this type ordered by this customer, and the `price` attribute is the total cost of each machine ordered.

a) Which are dimension attributes, and which are dependent attributes?

b) For some of the dimension attributes, a dimension table is likely to be needed. Suggest appropriate schemas for these dimension tables.

! **Exercise 20.4.2:** Suppose that we want to examine the data of Exercise 20.4.1 to find trends and thus predict which components the company should order more of. Describe a series of drill-down and roll-up queries that could lead to the conclusion that customers are beginning to prefer a DVD drive to a CD drive.

20.5 Data Cubes

In this section, we shall consider the "formal" data cube and special operations on data presented in this form. Recall from Section 20.4.2 that the formal data cube (just "data cube" in this section) precomputes all possible aggregates in a systematic way. Surprisingly, the amount of extra storage needed is often tolerable, and as long as the warehoused data does not change, there is no penalty incurred trying to keep all the aggregates up-to-date.

In the data cube, it is normal for there to be some aggregation of the raw data of the fact table before it is entered into the data-cube and its further aggregates computed. For instance, in our cars example, the dimension we thought of as a serial number in the star schema might be replaced by the model of the car. Then, each point of the data cube becomes a description of a model, a dealer and a date, together with the sum of the sales for that model, on that date, by that dealer. We shall continue to call the points of the (formal) data cube a "fact table," even though the interpretation of the points may be slightly different from fact tables in a star schema built from a raw-data cube.

20.5.1 The Cube Operator

Given a fact table F, we can define an augmented table CUBE(F) that adds an additional value, denoted `*`, to each dimension. The `*` has the intuitive meaning "any," and it represents aggregation along the dimension in which it appears. Figure 20.25 suggests the process of adding a border to the cube in each dimension, to represent the `*` value and the aggregated values that it implies. In this figure we see three dimensions, with the lightest shading representing aggregates in one dimension, darker shading for aggregates over two dimensions, and the darkest cube in the corner for aggregation over all three dimensions. Notice that if the number of values along each dimension is reasonably large, but not so large that most points in the cube are unoccupied, then the "border" represents only a small addition to the volume of the cube (i.e., the number of tuples in the fact table). In that case, the size of the stored data CUBE(F) is not much greater than the size of F itself.

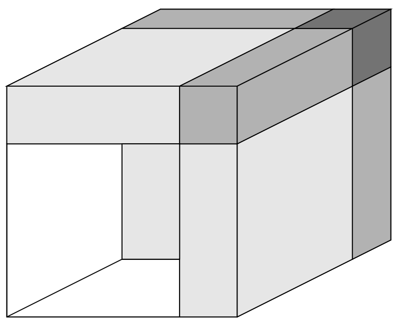

Figure 20.25: The cube operator augments a data cube with a border of aggregations in all combinations of dimensions

A tuple of the table CUBE(F) that has `*` in one or more dimensions will have for each dependent attribute the sum (or another aggregate function) of the values of that attribute in all the tuples that we can obtain by replacing the `*`'s by real values. In effect, we build into the data the result of aggregating along any set of dimensions. Notice, however, that the CUBE operator does not support aggregation at intermediate levels of granularity based on values in the dimension tables. For instance, we may either leave data broken down by day (or whatever the finest granularity for time is), or we may aggregate time completely, but we cannot, with the CUBE operator alone, aggregate by weeks, months, or years.

Example 20.32: Let us reconsider the Aardvark database from Example 20.27 in the light of what the CUBE operator can give us. Recall the fact table from that example is

```
Sales(serialNo, date, dealer, price)
```

However, the dimension represented by `serialNo` is not well suited for the cube, since the serial number is a key for `Sales`. Thus, summing the price over all dates, or over all dealers, but keeping the serial number fixed has no effect; we would still get the "sum" for the one auto with that serial number. A more useful data cube would replace the serial number by the two attributes — model and color — to which the serial number connects `Sales` via the dimension table `Autos`. Notice that if we replace `serialNo` by `model` and `color`, then the cube no longer has a key among its dimensions. Thus, an entry of the cube would have the total sales price for all automobiles of a given model, with a given color, by a given dealer, on a given date.

There is another change that is useful for the data-cube implementation of the `Sales` fact table. Since the CUBE operator normally sums dependent variables, and we might want to get average prices for sales in some category, we need both the sum of the prices for each category of automobiles (a given model of a given color sold on a given day by a given dealer) and the total number of sales in that category. Thus, the relation `Sales` to which we apply the CUBE operator is

```
Sales(model, color, date, dealer, val, cnt)
```

The attribute `val` is intended to be the total price of all automobiles for the given model, color, date, and dealer, while `cnt` is the total number of automobiles in that category. Notice that in this data cube, individual cars are not identified; they only affect the value and count for their category.

Now, let us consider the relation CUBE(`Sales`). A hypothetical tuple that would be in both `Sales` and CUBE(`Sales`), is

```
('Gobi', 'red', '2001-05-21', 'Friendly Fred', 45000, 2)
```

The interpretation is that on May 21, 2001, dealer Friendly Fred sold two red Gobis for a total of $45,000. The tuple

```
('Gobi', *, '2001-05-21', 'Friendly Fred', 152000, 7)
```

says that on May 21, 2001, Friendly Fred sold seven Gobis of all colors, for a total price of $152,000. Note that this tuple is in CUBE(`Sales`) but not in `Sales`.

Relation CUBE(`Sales`) also contains tuples that represent the aggregation over more than one attribute. For instance,

```
('Gobi', *, '2001-05-21', *, 2348000, 100)
```

says that on May 21, 2001, there were 100 Gobis sold by all the dealers, and the total price of those Gobis was $2,348,000.

```
('Gobi', *, *, *, 1339800000, 58000)
```

Says that over all time, dealers, and colors, 58,000 Gobis have been sold for a total price of $1,339,800,000. Lastly, the tuple

```
(*, *, *, *, 3521727000, 198000)
```

tells us that total sales of all Aardvark models in all colors, over all time at all dealers is 198,000 cars for a total price of $3,521,727,000. □

Consider how to answer a query in which we specify conditions on certain attributes of the `Sales` relation and group by some other attributes, while asking for the sum, count, or average price. In the relation CUBE(`Sales`), we look for those tuples t with the following properties:

1. If the query specifies a value v for attribute a, then tuple t has v in its component for a.
2. If the query groups by an attribute a, then t has any non-`*` value in its component for a.
3. If the query neither groups by attribute a nor specifies a value for a, then t has `*` in its component for a.

Each tuple t has the sum and count for one of the desired groups. If we want the average price, a division is performed on the sum and count components of each tuple t.

Example 20.33: The query

```
SELECT color, AVG(price)
FROM Sales
WHERE model = 'Gobi'
GROUP BY color;
```

is answered by looking for all tuples of CUBE(`Sales`) with the form

('Gobi', c, *, *, v, n)

where c is any specific color. In this tuple, v will be the sum of sales of Gobis in that color, while n will be the number of sales of Gobis in that color. The average price, although not an attribute of `Sales` or CUBE(`Sales`) directly, is v/n. The answer to the query is the set of $(c, v/n)$ pairs obtained from all (`'Gobi'`, c, `*`, `*`, v, n) tuples. □