item	IBM home computer

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

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

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

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

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

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

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

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

2.5.2 Complex aggregation at multiple granularities: Multifeature cubes

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

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

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

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

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

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

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

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

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

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

Let's look at another example.

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

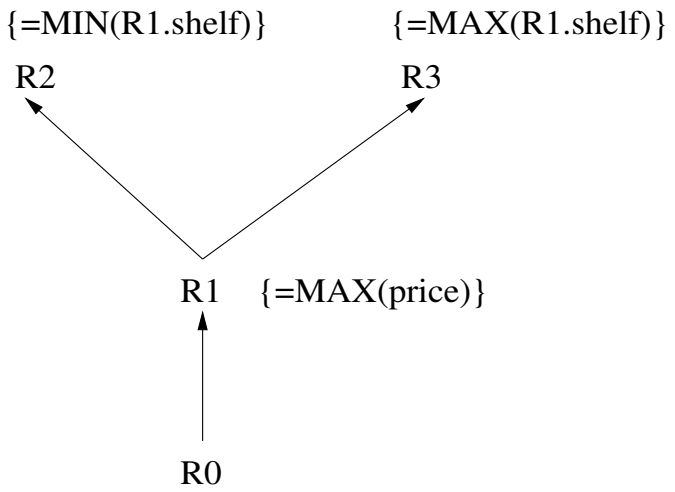

Figure 2.20: A multifeature cube graph for Query 3.

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

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

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

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

□

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

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

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

2.6 From data warehousing to data mining

2.6.1 Data warehouse usage

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

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

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

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

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

- **Analytical processing** supports basic OLAP operations, including slice-and-dice, drill-down, roll-up, and pivoting. It generally operates on historical data in both summarized and detailed forms. The major strength of on-line analytical processing over information processing is the multidimensional data analysis of data warehouse data.

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