Contents

Chapter 5

Data Cube Technology

Data warehouse systems provide OLAP tools for interactive analysis of multidimensional data at varied levels of granularity. OLAP tools typically use the *data cube* and a multidimensional data model to provide flexible access to summarized data. For example, a data cube can store precomputed measures (like `count` and `total_sales`) for multiple combinations of data dimensions (like `item`, `region`, and `customer`). Users can pose OLAP queries on the data. They can also interactively explore the data in a multidimensional way through OLAP operations like *drill-down* (to see more specialized data, such as total sales per city) or *roll-up* (to see the data at a more generalized level, such as total sales per country).

Although the data cube concept was originally intended for OLAP, it is also useful for data mining. **Multidimensional data mining** is an approach to data mining that integrates OLAP-based data analysis with knowledge discovery techniques. It is also known as *exploratory multidimensional data mining* and *online analytical mining* (*OLAM*). It searches for interesting patterns by exploring the data in multidimensional space. This gives users the freedom to dynamically focus on any subset of interesting dimensions. Users can interactively drill down or roll up to varying levels of abstraction to find classification models, clusters, predictive rules, and outliers.

This chapter focusses on data cube technology. In particular, we study methods for data cube computation and methods for multidimensional data analysis. Precomputing a data cube (or parts of a data cube) allows for fast accessing of summarized data. Given the high dimensionality of most data, multidimensional analysis can run into performance bottlenecks. Therefore, it is important to study data cube computation techniques. Luckily, data cube technology provides many effective and scalable methods for cube computation. Studying such methods will also help in our understanding and the development of scalable methods for other data mining tasks, such as the discovery of frequent patterns (Chapters 6 and 7).

We begin in Section 5.1 with preliminary concepts for cube computation. These summarize the notion of the data cube as a lattice of cuboids, and describe the basic forms of cube materialization. General strategies for cube computation are given. Section 5.2 follows with an in-depth look at specific methods for data cube computation. We study both *full materialization* (that is, where all of the cuboids representing a data cube are precomputed and thereby ready for use) and *partial cuboid materialization* (where, say, only the more "useful" parts of the data cube are precomputed). The *Multiway Array Aggregation* method is detailed for full cube computation. Methods for partial cube computation, including *BUC*, *Star-Cubing*, and the use of *cube shell fragments*, are discussed.

In Section 5.3, we study cube-based query processing. The techniques described build upon the standard methods of cube computation presented in Section 5.2. You will learn about *sampling cubes* for OLAP query-answering on sampling data (such as survey data, which represent a sample or subset of a target data population of interest). In addition, you will learn how to compute *ranking cubes* for efficient top-k (ranking) query processing in large relational datasets.

In Section 5.4, we describe various ways to perform multidimensional data analysis using data cubes. *Prediction cubes* are introduced, which facilitate predictive modeling in multidimensional space. We discuss *multifeature cubes*, which compute complex queries involving multiple dependent aggregates at multiple granularities. You will also learn about the *exception-based discovery-driven exploration* of cube space, where visual cues are displayed to indicate discovered data exceptions at all levels of aggregation, thereby guiding the user in the data analysis process.

5.1 Data Cube Computation: Preliminary Concepts

Data cubes facilitate the on-line analytical processing of multidimensional data. *"But how can we compute data cubes in advance, so that they are handy and readily available for query processing?"* This section contrasts full cube materialization (i.e., precomputation) versus various strategies for partial cube materialization. For completeness, we begin with a review of the basic terminology involving data cubes. We also introduce a cube cell notation that is useful for describing data cube computation methods.

5.1.1 Cube Materialization: Full Cube, Iceberg Cube, Closed Cube, and Cube Shell

Figure 5.1 shows a 3-D data cube for the dimensions A, B, and C, and an aggregate measure, M. Commonly used measures include `count, sum, min, max`, and `total_sales`. A data cube is a lattice of cuboids. Each cuboid represents a group-by. ABC is the base cuboid, containing all three of the dimensions. Here, the aggregate measure, M, is computed for each possible combination of the three dimensions. The base cuboid is the least generalized of all of the cuboids in the data cube. The most generalized cuboid is the apex cuboid, commonly represented as all. It contains one value—it aggregates measure M for all of the tuples stored in the base cuboid. To drill down in the data cube, we move from the apex cuboid, downward in the lattice. To roll up, we move from the base cuboid, upward. For the purposes of our discussion in this chapter, we will always use the term data cube to refer to a lattice of cuboids rather than an individual cuboid.

A cell in the base cuboid is a **base cell**. A cell from a nonbase cuboid is an **aggregate cell**. An aggregate cell aggregates over one or more dimensions, where each aggregated dimension is indicated by a "$*$" in the cell notation. Suppose we have an n-dimensional data cube. Let $a = (a_1, a_2, \ldots, a_n, measures)$ be a cell from one of the cuboids making up the data cube. We say that a is an **m-dimensional cell** (that is, from an m-dimensional cuboid) if exactly m ($m \leq n$) values among $\{a_1, a_2, \ldots, a_n\}$ are *not* "$*$". If $m = n$, then a is a base cell; otherwise, it is an aggregate cell (i.e., where $m < n$).

Example 5.1 **Base and aggregate cells.** Consider a data cube with the dimensions *month, city*, and *customer_group*, and the measure *sales.* (Jan, $*$, $*$, 2800) and ($*$, *Chicago*, $*$, 1200) are 1-D cells, (Jan, $*$, *Business*, 150) is a 2-D cell, and (Jan, *Chicago*, *Business*, 45) is a 3-D cell. Here, all base cells are 3-D, whereas 1-D and 2-D cells are aggregate cells.

An ancestor-descendant relationship may exist between cells. In an n-dimensional data cube, an i-D cell $a = (a_1, a_2, \ldots, a_n, measures_a)$ is an **ancestor** of a j-D cell $b = (b_1, b_2, \ldots, b_n, measures_b)$, and b is a **descendant** of a, if and only if (1) $i < j$, and (2) for $1 \leq k \leq n$, $a_k = b_k$ whenever $a_k \neq$ "$*$". In particular, cell a is called a **parent** of cell b, and b is a **child** of a, if and only if $j = i + 1$.

Example 5.2 **Ancestor and descendant cells.** Referring to our previous example, 1-D cell $a = (Jan, *, *, 2800)$ and 2-D cell $b = (Jan, *, Business, 150)$ are *ancestors* of 3-D cell $c = (Jan, Chicago, Business, 45)$; c is a *descendant* of both a and

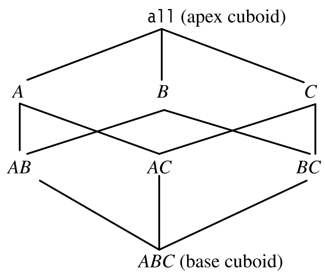


Figure 5.1: Lattice of cuboids, making up a 3-D data cube with the dimensions A, B, and C for some aggregate measure, M.

b; b is a *parent* of c, and c is a *child* of b.

In order to ensure fast on-line analytical processing, it is sometimes desirable to precompute the **full cube** (i.e., all the cells of all of the cuboids for a given data cube). A method of full cube computation is given in Section 4.4. Full cube computation, however, is exponential to the number of dimensions. That is, a data cube of n dimensions contains 2^n cuboids. There are even more cuboids if we consider concept hierarchies for each dimension.[1] In addition, the size of each cuboid depends on the cardinality of its dimensions. Thus, precomputation of the full cube can require huge and often excessive amounts of memory.

Nonetheless, full cube computation algorithms are important. **Individual** cuboids may be stored on secondary storage and accessed when necessary. Alternatively, we can use such algorithms to compute smaller cubes, consisting of a subset of the given set of dimensions, or a smaller range of possible values for some of the dimensions. In such cases, the smaller cube is a full cube for the given subset of dimensions and/or dimension values. A thorough understanding of full cube computation methods will help us develop efficient methods for computing partial cubes. Hence, it is important to explore scalable methods for computing all of the cuboids making up a data cube, that is, for full materialization. These methods must take into consideration the limited amount of main memory available for cuboid computation, the total size of the computed data cube, as well as the time required for such computation.

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

Many cells in a cuboid may actually be of little or no interest to the data analyst. Recall that each cell in a full cube records an aggregate value, such as `count` or `sum`. For many cells in a cuboid, the measure value will be zero. When the product of the cardinalities for the dimensions in a cuboid is large relative to the number of nonzero-valued tuples that are stored in the cuboid, then we say that the cuboid is **sparse**. If a cube contains many sparse cuboids, we say that the cube is **sparse**.

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

[1]Equation (4.1) of Section 4.4.1 gives the total number of cuboids in a data cube where each dimension has an associated concept hierarchy.

processing time and disk space, but also leads to a more focused analysis. The cells that cannot pass the threshold are likely to be too trivial to warrant further analysis. Such partially materialized cubes are known as **iceberg cubes**. The minimum threshold is called the **minimum support threshold**, or *minimum support* (*min_sup*), for short. By materializing only a fraction of the cells in a data cube, the result is seen as the "tip of the iceberg," where the "iceberg" is the potential full cube including all cells. An iceberg cube can be specified with an SQL query, as shown in the following example.

Example 5.3 **Iceberg cube.**

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

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

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

A naïve approach to computing an iceberg cube would be to first compute the full cube and then prune the cells that do not satisfy the iceberg condition. However, this is still prohibitively expensive. An efficient approach is to compute only the iceberg cube directly without computing the full cube. Sections 5.2.2 to 5.2.3 discuss methods for efficient iceberg cube computation.

Introducing iceberg cubes will lessen the burden of computing trivial aggregate cells in a data cube. However, we could still end up with a large number of uninteresting cells to compute. For example, suppose that there are 2 base cells for a database of 100 dimensions, denoted as $\{(a_1, a_2, a_3, \ldots, a_{100}) : 10, (a_1, a_2, b_3, \ldots, b_{100}) : 10\}$, where each has a cell count of 10. If the minimum support is set to 10, there will still be an impermissible number of cells to compute and store, although most of them are not interesting. For example, there are $2^{101} - 6$ distinct aggregate cells,[2] like $\{(a_1, a_2, a_3, a_4, \ldots, a_{99}, *) :$

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

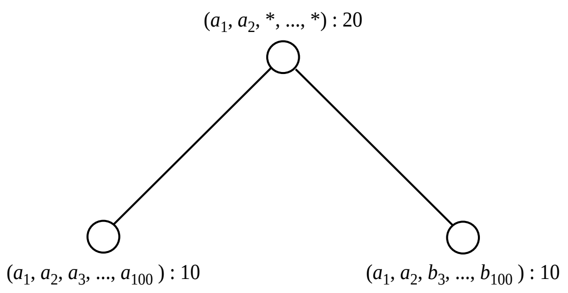

$(a_1, a_2, *, ..., *) : 20$
$(a_1, a_2, a_3, ..., a_{100}) : 10$
$(a_1, a_2, b_3, ..., b_{100}) : 10$

Figure 5.2: Three closed cells forming the lattice of a closed cube.

$10, \ldots, (a_1, a_2, *, a_4, \ldots, a_{99}, a_{100}) : 10, \ldots, (a_1, a_2, a_3, *, \ldots, *, *) : 10\}$, but most of them do not contain much new information. If we ignore all of the aggregate cells that can be obtained by replacing some constants by $*$'s while keeping the same measure value, there are only three distinct cells left: $\{(a_1, a_2, a_3, \ldots, a_{100}) : 10, (a_1, a_2, b_3, \ldots, b_{100}) : 10, (a_1, a_2, *, \ldots, *) : 20\}$. That is, out of $2^{101} - 4$ distinct base and aggregate cells, only three really offer valuable information.

To systematically compress a data cube, we need to introduce the concept of *closed coverage*. A cell, c, is a *closed cell* if there exists no cell, d, such that d is a specialization (descendant) of cell c (that is, where d is obtained by replacing a $*$ in c with a non-$*$ value), and d has the same measure value as c. A **closed cube** is a data cube consisting of only closed cells. For example, the three cells derived above are the three closed cells of the data cube for the data set: $\{(a_1, a_2, a_3, \ldots, a_{100}) : 10, (a_1, a_2, b_3, \ldots, b_{100}) : 10\}$. They form the lattice of a closed cube as shown in Figure 5.2. Other nonclosed cells can be derived from their corresponding closed cells in this lattice. For example, "$(a_1, *, *, \ldots, *) : 20$" can be derived from "$(a_1, a_2, *, \ldots, *) : 20$" because the former is a generalized nonclosed cell of the latter. Similarly, we have "$(a_1, a_2, b_3, *, \ldots, *) : 10$".

Another strategy for partial materialization is to precompute only the cuboids involving a small number of dimensions, such as 3 to 5. These cuboids form a **cube shell** for the corresponding data cube. Queries on additional combinations of the dimensions will have to be computed on the fly. For example, we could compute all cuboids with 3 dimensions or less in an n-dimensional data cube, resulting in a cube shell of size 3. This, however, can still result in a large number of cuboids to compute, particularly when n is large. Alternatively, we can choose to precompute only portions or *fragments* of the cube shell, based on cuboids of interest. Section 5.2.4 discusses a method for computing such **shell fragments** and explores how they can be used for efficient OLAP query processing.

5.1.2 General Strategies for Data Cube Computation

There are several methods for efficient data cube computation, based on the various kinds of cubes described above. In general, there are two basic data structures used for storing cuboids. The implementation of relational OLAP (ROLAP) uses relational tables, whereas multidimensional arrays are used in multidimensional OLAP (MOLAP). Although ROLAP and MOLAP may each explore different cube computation techniques, some optimization "tricks" can be shared among the different data representations. The following are general optimization techniques for efficient computation of data cubes.

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

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

For example, to compute total sales by *branch*, *day*, and *item*, it can be more efficient to sort tuples or cells by *branch*, and then by *day*, and then group them according to the *item* name. Efficient implementations of such operations in large data sets have been extensively studied in the database research community. Such implementations can be extended to data cube computation.

This technique can also be further extended to perform **shared-sorts** (i.e., sharing sorting costs across multiple cuboids when sort-based methods are used), or to perform **shared-partitions** (i.e., sharing the partitioning cost across multiple cuboids when hash-based algorithms are used).

Optimization Technique 2: Simultaneous aggregation and caching intermediate results. In cube computation, it is efficient to compute higher-level aggregates from previously computed lower-level aggregates, rather than from the base fact table. Moreover, simultaneous aggregation from cached intermediate computation results may lead to the reduction of expensive disk I/O operations.

For example, to compute sales by *branch*, we can use the intermediate results derived from the computation of a lower-level cuboid, such as sales by *branch* and *day*. This technique can be further extended to perform **amortized scans** (i.e., computing as many cuboids as possible at the same time to amortize disk reads).

Optimization Technique 3: Aggregation from the smallest child, when there exist multiple child cuboids. When there exist multiple child cuboids, it is usually more efficient to compute the desired parent (i.e., more generalized) cuboid from the smallest, previously computed child cuboid.

For example, to compute a sales cuboid, C_{branch}, when there exist two previously computed cuboids, $C_{\{branch,year\}}$ and $C_{\{branch,item\}}$, it is obviously more efficient to compute C_{branch} from the former than from the latter if there are many more distinct items than distinct years.

Many other optimization techniques may further improve the computational efficiency. For example, *string dimension attributes can be mapped to integers with values ranging from zero to the cardinality of the attribute.*

In iceberg cube computation the following optimization technique plays a particularly important role.

Optimization Technique 4: The Apriori pruning method can be explored to compute iceberg cubes efficiently. The **Apriori property**,[3] in the context of data cubes, states as follows: *If a given cell does not satisfy minimum support, then no descendant of the cell (i.e., more specialized cell) will satisfy minimum support either.* This property can be used to substantially reduce the computation of iceberg cubes.

Recall that the specification of iceberg cubes contains an iceberg condition, which is a constraint on the cells to be materialized. A common iceberg condition is that the cells must satisfy a *minimum support* threshold, such as a minimum count or sum. In this situation, the Apriori property can be used to prune away the exploration of the descendants of the cell. For example, if the count of a cell, c, in a cuboid is less than a minimum support threshold, v, then the count of any of c's descendant cells in the lower-level cuboids can never be greater than or equal to v, and thus can be pruned. In other words, if a condition (e.g., the iceberg condition specified in a having clause) is violated for some cell c, then every descendant of c will also violate that condition. Measures that obey this property are known as **antimonotonic**.[4] This form of pruning was made popular in frequent pattern mining, yet also aids in data cube computation by cutting processing time and disk space requirements. It can lead to a more focused analysis because cells that cannot pass the threshold are unlikely to be of interest.

In the following subsections, we introduce several popular methods for efficient cube computation that explore some or all of the above optimization strategies.

5.2 Data Cube Computation Methods

Data cube computation is an essential task in data warehouse implementation. The precomputation of all or part of a data cube can greatly reduce the response time and enhance the performance of on-line analytical processing. However, such computation is challenging because it may require substantial computational time and storage space. This section explores efficient methods for data cube computation. Section 5.2.1 describes the *multiway array aggregation* (MultiWay) method for computing full cubes. Section 5.2.2 describes a method known as BUC, which computes iceberg cubes from the apex cuboid, downward. Section 5.2.3 describes the Star-Cubing method, which integrates top-down and bottom-up computation. Finally, Section 5.2.4 describes a shell-fragment cubing approach that computes shell fragments for efficient high-dimensional OLAP.

[3] The Apriori property was proposed in the Apriori algorithm for association rule mining by R. Agrawal and R. Srikant [AS94]. Many algorithms in association rule mining have adopted this property. Association rule mining is the topic of Chapter 6.

[4] **Antimonotone** is based on *condition violation*. This differs from **monotone**, which is based on *condition satisfaction*.

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

5.2.1 Multiway Array Aggregation for Full Cube Computation

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

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

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

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

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

Example 5.4 **Multiway array cube computation.** Consider a 3-D data array containing the three dimensions A, B, and C. The 3-D array is partitioned into small, memory-based chunks. In this example, the array is partitioned into 64 chunks as shown in Figure 5.3. Dimension A is organized into four equal-sized partitions, a_0, a_1, a_2, and a_3. Dimensions B and C are similarly organized into four partitions each. Chunks 1, 2, ..., 64 correspond to the subcubes $a_0b_0c_0$,

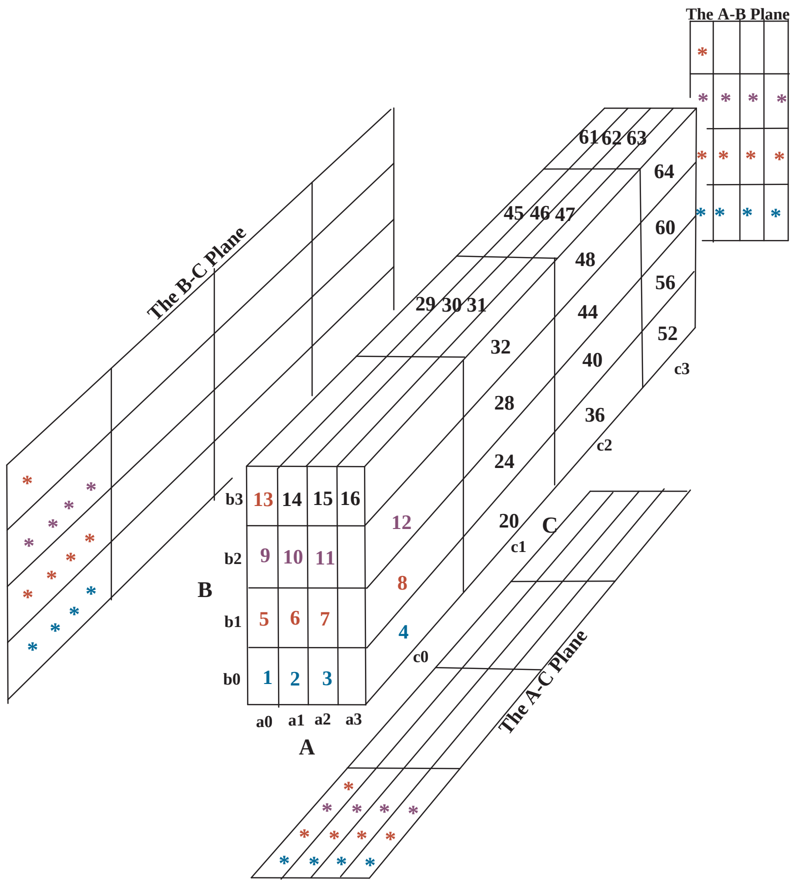


Figure 5.3: A 3-D array for the dimensions A, B, and C, organized into 64 *chunks*. Each chunk is small enough to fit into the memory available for cube computation. The $*$'s indicate the chunks from 1 to 13 that have been aggregated so far in the process.

$a_1b_0c_0$, ..., $a_3b_3c_3$, respectively. Suppose that the cardinality of the dimensions A, B, and C is 40, 400, and 4000, respectively. Thus, the size of the array for each dimension, A, B, and C, is also 40, 400, and 4000, respectively. The size of each partition in A, B, and C is therefore 10, 100, and 1000, respectively. Full materialization of the corresponding data cube involves the computation of all of the cuboids defining this cube. The resulting full cube consists of the following cuboids:

- The base cuboid, denoted by ABC (from which all of the other cuboids are directly or indirectly computed). This cube is already computed and corresponds to the given 3-D array.
- The 2-D cuboids, AB, AC, and BC, which respectively correspond to the

group-by's AB, AC, and BC. These cuboids must be computed.

- The 1-D cuboids, A, B, and C, which respectively correspond to the group-by's A, B, and C. These cuboids must be computed.
- The 0-D (apex) cuboid, denoted by `all`, which corresponds to the group-by (); that is, there is no group-by here. This cuboid must be computed. It consists of only one value. If, say, the data cube measure is `count`, then the value to be computed is simply the total count of all of the tuples in ABC.

Let's look at how the multiway array aggregation technique is used in this computation. There are many possible orderings with which chunks can be read into memory for use in cube computation. Consider the ordering labeled from 1 to 64, shown in Figure 5.3. Suppose we would like to compute the b_0c_0 chunk of the BC cuboid. We allocate space for this chunk in *chunk memory.* By scanning chunks 1 to 4 of ABC, the b_0c_0 chunk is computed. That is, the cells for b_0c_0 are aggregated over a_0 to a_3. The chunk memory can then be assigned to the next chunk, b_1c_0, which completes its aggregation after the scanning of the next four chunks of ABC: 5 to 8. Continuing in this way, the entire BC cuboid can be computed. Therefore, only *one* chunk of BC needs to be in memory, at a time, for the computation of all of the chunks of BC.

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

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

Suppose that the chunks are scanned in the order shown, from chunk 1 to 64. As mentioned above, b_0c_0 is fully aggregated after scanning the row containing chunks 1 to 4; b_1c_0 is fully aggregated after scanning chunks 5 to 8, and so on. Thus, we need to scan four chunks of the 3-D array in order to *fully* compute one chunk of the BC cuboid (where BC is the largest of the 2-D planes). In other words, by scanning in this order, one chunk of BC is fully computed for each row scanned. In comparison, the complete computation of one chunk of the second largest 2-D plane, AC, requires scanning 13 chunks, given the ordering from 1 to 64. That is, a_0c_0 is fully aggregated only after the scanning of chunks 1, 5, 9, and 13. Finally, the complete computation of one chunk of the smallest 2-D plane, AB, requires scanning 49 chunks. For example, a_0b_0 is fully aggregated after scanning chunks 1, 17, 33, and 49. Hence, AB requires the longest scan of chunks in order to complete its computation. To avoid bringing a 3-D chunk into memory more than once, the minimum memory requirement for holding all relevant 2-D planes in chunk memory, according to the chunk ordering of 1 to 64, is as follows: 40×400 (for the whole AB plane) $+ 40 \times 1000$ (for one column of the AC plane) $+ 100 \times 1000$ (for one chunk of the BC plane) $= 16,000 + 40,000 + 100,000 = 156,000$ memory units.

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

Similarly, we can work out the minimum memory requirements for the multiway computation of the 1-D and 0-D cuboids. Figure 5.4 shows the most efficient way to compute 1-D cuboids. Chunks for 1-D cuboids A and B are computed during the computation of the smallest 2-D cuboid, AB. The smallest 1-D cuboid, A, will have all of its chunks allocated in memory, whereas the larger 1-D cuboid, B, will have only one chunk allocated in memory at a time. Similarly, chunk C is computed during the computation of the second smallest 2-D cuboid, AC, requiring only one chunk in memory at a time. Based on this analysis, we see that the most efficient ordering in this array cube computation is the chunk ordering of 1 to 64, with the above stated memory allocation strategy.

Example 5.4 assumes that there is enough memory space for *one-pass* cube computation (i.e., to compute all of the cuboids from one scan of all of the chunks). If there is insufficient memory space, the computation will require more than one pass through the 3-D array. In such cases, however, the basic principle of ordered chunk computation remains the same. MultiWay is most effective when the product of the cardinalities of dimensions is moderate and the data are not too sparse. When the dimensionality is high or the data are very sparse, the in-memory arrays become too large to fit in memory, and this method becomes infeasible.

With the use of appropriate sparse array compression techniques and careful ordering of the computation of cuboids, it has been shown by experiments that MultiWay array cube computation is significantly faster than traditional ROLAP (relational record-based) computation. Unlike ROLAP, the array structure of MultiWay does not require saving space to store search keys. Furthermore, MultiWay uses direct array addressing, which is faster than the key-based addressing search strategy of ROLAP. For ROLAP cube computation, instead of cubing a table di-

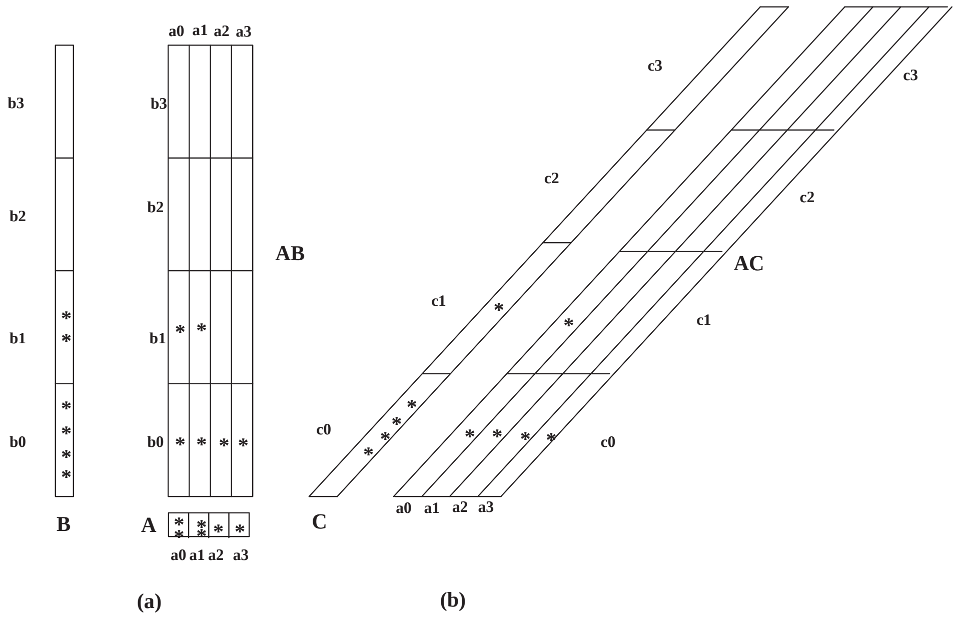


Figure 5.4: Memory allocation and order of computation for computing the 1-D cuboids of Example 5.4. (a) The 1-D cuboids, A and B, are aggregated during the computation of the smallest 2-D cuboid, AB (b) The 1-D cuboid, C, is aggregated during the computation of the second smallest 2-D cuboid, AC. The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

rectly, it can be faster to convert the table to an array, cube the array, and then convert the result back to a table. However, this observation works only for cubes with a relatively small number of dimensions because the number of cuboids to be computed is exponential to the number of dimensions.

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

5.2.2 BUC: Computing Iceberg Cubes from the Apex Cuboid Downward

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

Figure 5.5 shows a lattice of cuboids, making up a 3-D data cube with the dimensions A, B, and C. The apex (0-D) cuboid, representing the concept all (that is, $(*, *, *)$), is at the top of the lattice. This is the most aggregated or generalized level. The 3-D base cuboid, ABC, is at the bottom of the lattice. It is the least aggregated (most detailed or specialized) level. This representation of a lattice of cuboids, with the apex at the top and the base at the bottom, is commonly accepted in data warehousing. It consolidates the notions of *drill-down* (where we can move from a highly aggregated cell to lower, more detailed cells) and *roll-up* (where we can move from detailed, low-level cells to higher-level, more aggregated cells).

BUC stands for "Bottom-Up Construction." However, according to the lattice convention described above and used throughout this book, the order of processing of BUC is actually top-down! The authors of BUC view a lattice of cuboids in the reverse order, with the apex cuboid at the bottom and the base cuboid at the top. In that view, BUC does bottom-up construction. However, because we adopt the application worldview where *drill-down* refers to drilling from the apex cuboid down toward the base cuboid, the exploration process of BUC is regarded as top-down. BUC's exploration for the computation of a 3-D data cube is shown in Figure 5.5.

The BUC algorithm is shown in Figure 5.6. We first give an explanation of the algorithm and then follow up with an example. Initially, the algorithm is called with the input relation (set of tuples). BUC aggregates the entire input (line 1) and writes the resulting total (line 3). (Line 2 is an optimization feature that is discussed later in our example.) For each dimension d (line 4), the input

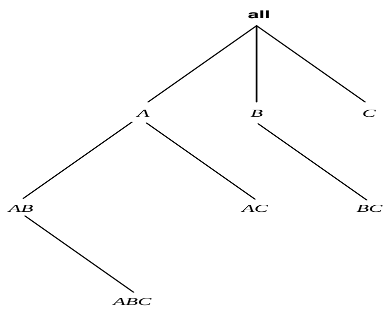


Figure 5.5: BUC's exploration for the computation of a 3-D data cube. Note that the computation starts from the apex cuboid.

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

Input:

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

Globals:

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

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

Method:

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

Figure 5.6: BUC algorithm for the computation of sparse or iceberg cubes [BR99].

is partitioned on d (line 6). On return from Partition(), *dataCount* contains the total number of tuples for each distinct value of dimension d. Each distinct value of *d forms its own partition.* Line 8 iterates through each partition. Line 10 tests the partition for minimum support. That is, if the number of tuples in the partition satisfies (i.e., is $\geq$) the minimum support, then the partition becomes the input relation for a recursive call made to BUC, which computes the iceberg cube on the partitions for dimensions $d + 1$ to *numDims* (line 12). Note that for a full cube (i.e., where minimum support in the having clause is 1), the minimum support condition is always satisfied. Thus, the recursive call descends one level deeper into the lattice. Upon return from the recursive call, we continue with the next partition for d. After all the partitions have been processed, the entire process is repeated for each of the remaining dimensions.

We explain how BUC works with the following example.

Example 5.5 **BUC construction of an iceberg cube.** Consider the iceberg cube expressed in SQL as follows:

compute cube iceberg_cube as
select A, B, C, D, count(*)
from R

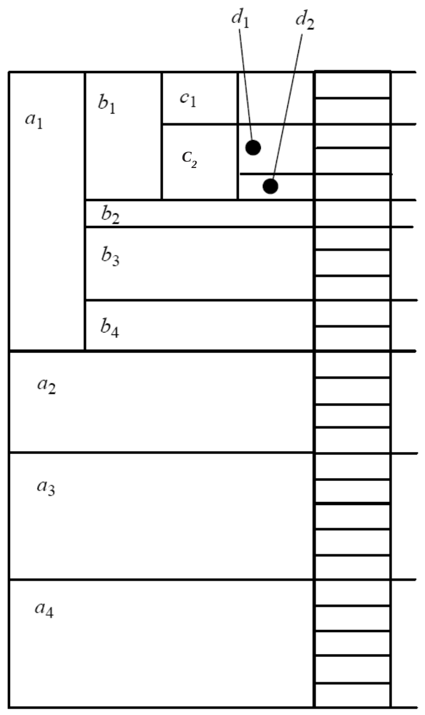


Figure 5.7: Snapshot of BUC partitioning given an example 4-D data set.

cube by A, B, C, D
having count(*) >= 3

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

Figure 5.7 illustrates how the input is partitioned first according to the different attribute values of dimension A, and then B, C, and D. To do so, BUC scans the input, aggregating the tuples to obtain a count for all, corresponding to the cell $(*, *, *, *)$. Dimension A is used to split the input into four partitions, one for each distinct value of A. The number of tuples (counts) for each distinct value of A is recorded in *dataCount*.

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

The partition process is facilitated by a linear sorting method, CountingSort. CountingSort is fast because it does not perform any key comparisons to find partition boundaries. In addition, the counts computed during the sort can be reused to compute the group-by's in BUC. Line 2 is an optimization for partitions having a count of 1, such as $(a_1, b_2, *, *)$ in our example. To save on partitioning costs, the count is written to each of the tuple's descendant group-by's. This is particularly useful since, in practice, many partitions have a single tuple.

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

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

5.2.3 Star-Cubing: Computing Iceberg Cubes Using a Dynamic Star-tree Structure

In this section, we describe the **Star-Cubing** algorithm for computing iceberg cubes. Star-Cubing combines the strengths of the other methods we have studied up to this point. It integrates top-down and bottom-up cube computation and explores both multidimensional aggregation (similar to MultiWay) and Apriori-like pruning (similar to BUC). It operates from a data structure called a star-tree, which performs lossless data compression, thereby reducing the computation time and memory requirements.

The Star-Cubing algorithm explores both the bottom-up and top-down computation models as follows: On the global computation order, it uses the bottom-up model. However, it has a sublayer underneath based on the top-down model, which explores the notion of *shared dimensions*, as we shall see below. This integration allows the algorithm to aggregate on multiple dimensions while still partitioning parent group-by's and pruning child group-by's that do not satisfy the iceberg condition.

Star-Cubing's approach is illustrated in Figure 5.8 for the computation of a 4-D data cube. If we were to follow only the bottom-up model (similar to Multiway), then the cuboids marked as pruned by Star-Cubing would still be explored. Star-Cubing is able to prune the indicated cuboids because it considers shared dimensions. ACD/A means cuboid ACD has shared dimension A, ABD/AB means cuboid ABD has shared dimension AB, ABC/ABC means cuboid ABC has shared dimension ABC, and so on. This comes from the generalization that all the cuboids in the subtree rooted at ACD include dimension A, all those rooted at ABD include dimensions AB, and all those rooted at ABC include dimensions ABC (even though there is only one such cuboid). We call these common dimensions the **shared dimensions** of those particular subtrees.

The introduction of shared dimensions facilitates shared computation. Because the shared dimensions are identified early on in the tree expansion, we can avoid recomputing them later. For example, cuboid AB extending from ABD in Figure 5.8 would actually be pruned because AB was already computed in ABD/AB. Similarly, cuboid A extending from AD would also be pruned because it was already computed in ACD/A.

Shared dimensions allow us to do Apriori-like pruning if the measure of an iceberg cube, such as *count*, is antimonotonic; that is, if the aggregate value on a shared dimension does not satisfy the iceberg condition, then *all of the cells descending from this shared dimension cannot satisfy the iceberg condition either*. Such cells and all of their descendants can be pruned, because these descendant cells are, by definition, more specialized (i.e., contain more dimensions) than those in the shared dimension(s). The number of tuples covered by the descendant cells will be less than or equal to the number of tuples covered by the shared dimensions. Therefore, if the aggregate value on a shared dimension fails the iceberg condition, the descendant cells cannot

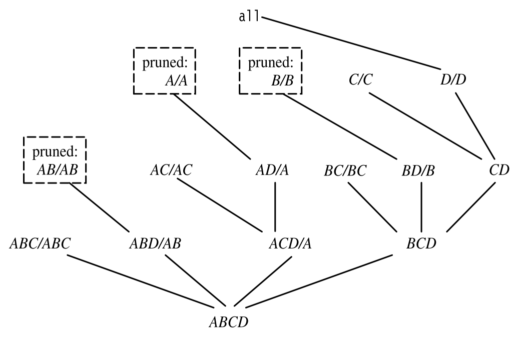


Figure 5.8: Star-Cubing: Bottom-up computation with top-down expansion of shared dimensions.

satisfy it either.

Example 5.6 **Pruning shared dimensions.** If the value in the shared dimension A is a_1 and it fails to satisfy the iceberg condition, then the whole subtree rooted at a_1CD/a_1 (including a_1C/a_1C, a_1D/a_1, a_1/a_1) can be pruned because they are all more specialized versions of a_1.

To explain how the Star-Cubing algorithm works, we need to explain a few more concepts, namely, *cuboid trees*, *star-nodes*, and *star-trees*.

We use trees to represent individual cuboids. Figure 5.9 shows a fragment of the **cuboid tree** of the base cuboid, $ABCD$. Each level in the tree represents a dimension, and each node represents an attribute value. Each node has four fields: the attribute value, aggregate value, pointer to possible first child, and pointer to possible first sibling. Tuples in the cuboid are inserted one by one into the tree. A path from the root to a leaf node represents a tuple. For example, node c_2 in the tree has an aggregate (count) value of 5, which indicates that there are five cells of value (a_1, b_1, c_2, $*$). This representation collapses the common prefixes to save memory usage and allows us to aggregate the values at internal nodes. With aggregate values at internal nodes, we can prune based on shared dimensions. For example, the cuboid tree of AB can be used to prune possible cells in ABD.

If the single dimensional aggregate on an attribute value p does not satisfy the iceberg condition, it is useless to distinguish such nodes in the iceberg cube computation. Thus the node p can be replaced by $*$ so that the cuboid tree can be further compressed. We say that the node p in an attribute A is a **star-node** if the single dimensional aggregate on p does not satisfy the iceberg condition; otherwise, p is a *non-star-node*. A cuboid tree that is compressed using star-nodes is called a **star-tree**.

The following is an example of star-tree construction.

Example 5.7 **Star-tree construction.** A base cuboid table is shown in Table 5.1. There

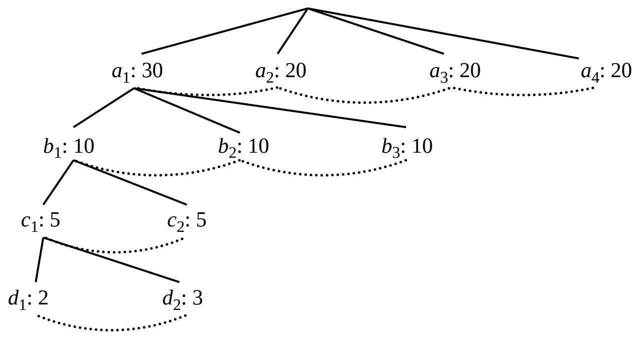


Figure 5.9: A fragment of the base cuboid tree.

Table 5.1: Base (Cuboid) Table: Before star reduction.

A	B	C	D	count
a_1	b_1	c_1	d_1	1
a_1	b_1	c_4	d_3	1
a_1	b_2	c_2	d_2	1
a_2	b_3	c_3	d_4	1
a_2	b_4	c_3	d_4	1

are 5 tuples and 4 dimensions. The cardinalities for dimensions A, B, C, D are 2, 4, 4, 4, respectively. The one-dimensional aggregates for all attributes are shown in Table 5.2. Suppose $min\_sup = 2$ in the iceberg condition. Clearly, only attribute values a_1, a_2, b_1, c_3, d_4 satisfy the condition. All the other values are below the threshold and thus become star-nodes. By collapsing star-nodes, the reduced base table is Table 5.3. Notice that the table contains two fewer rows and also fewer distinct values than Table 5.1.

Table 5.2: One-Dimensional Aggregates.

Dimension	count $= 1$	count ≥ 2
A	—	$a_1(3)$, $a_2(2)$
B	b_2, b_3, b_4	$b_1(2)$
C	c_1, c_2, c_4	$c_3(2)$
D	d_1, d_2, d_3	$d_4(2)$

We use the reduced base table to construct the cuboid tree because it is smaller. The resultant star-tree is shown in Figure 5.10.

Now, let's see how the Star-Cubing algorithm uses star-trees to compute an iceberg cube. The algorithm is given in Figure 5.13.

Example 5.8 **Star-Cubing.** Using the star-tree generated in Example 5.7 (Figure 5.10), we start the process of aggregation by traversing in a bottom-up fashion. Traversal is depth-first. The first stage (i.e., the processing of the first branch of the tree) is shown in Figure 5.11. The leftmost tree in the figure is the base star-

Table 5.3: Compressed Base Table: After star reduction.

A	B	C	D	count
a_1	b_1	$*$	$*$	2
a_1	$*$	$*$	$*$	1
a_2	$*$	c_3	d_4	2

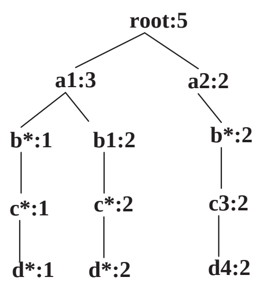


Figure 5.10: Star-tree of the compressed base table.

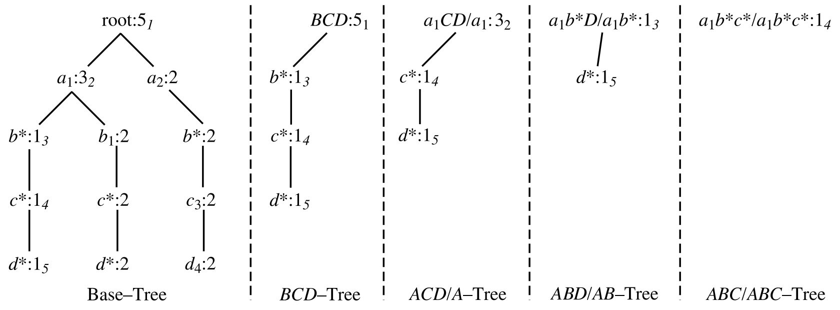


Figure 5.11: Aggregation Stage One: Processing of the left-most branch of BaseTree.

tree. Each attribute value is shown with its corresponding aggregate value. In addition, subscripts by the nodes in the tree show the order of traversal. The remaining four trees are BCD, ACD/A, ABD/AB, ABC/ABC. They are the child trees of the base star-tree, and correspond to the level of three-dimensional cuboids above the base cuboid in Figure 5.8. The subscripts in them correspond to the same subscripts in the base tree—they denote the step or order in which they are created during the tree traversal. For example, when the algorithm is at step 1, the BCD child tree root is created. At step 2, the ACD/A child tree root is created. At step 3, the ABD/AB tree root and the $b*$ node in BCD are created.

When the algorithm has reached step 5, the trees in memory are exactly as shown in Figure 5.11. Because the depth-first traversal has reached a leaf at this point, it starts backtracking. Before traversing back, the algorithm notices that all possible nodes in the base dimension (ABC) have been visited. This means the ABC/ABC tree is complete, so the count is output and the tree is destroyed. Similarly, upon moving back from $d*$ to $c*$ and seeing that $c*$ has no siblings, the count in ABD/AB is also output and the tree is destroyed.

When the algorithm is at $b*$ during the back-traversal, it notices that there exists a sibling in b_1. Therefore, it will keep ACD/A in memory and perform

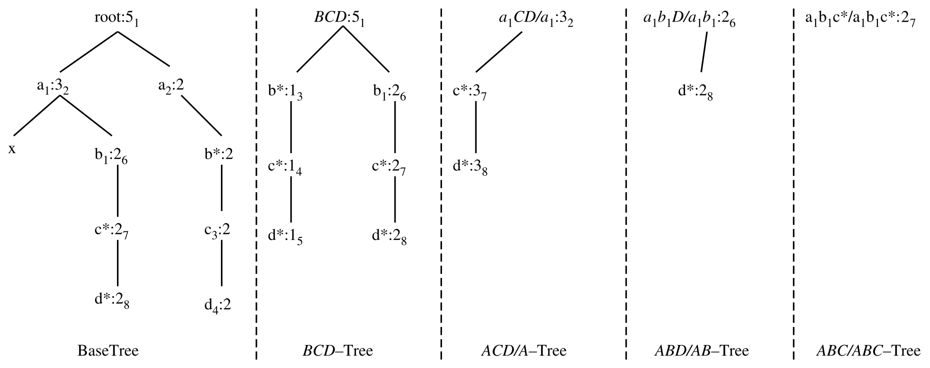


Figure 5.12: Aggregation Stage Two: Processing of the second branch of BaseTree.

a depth-first search on b_1 just as it did on $b*$. This traversal and the resultant trees are shown in Figure 5.12. The child trees ACD/A and ABD/AB are created again but now with the new values from the b_1 subtree. For example, notice that the aggregate count of $c*$ in the ACD/A tree has increased from 1 to 3. The trees that remained intact during the last traversal are reused and the new aggregate values are added on. For instance, another branch is added to the BCD tree.

Just like before, the algorithm will reach a leaf node at $d*$ and traverse back. This time, it will reach a_1 and notice that there exists a sibling in a_2. In this case, all child trees except BCD in Figure 5.12 are destroyed. Afterward, the algorithm will perform the same traversal on a_2. BCD continues to grow while the other subtrees start fresh with a_2 instead of a_1.

A node must satisfy two conditions in order to generate child trees: (1) the measure of the node must satisfy the iceberg condition; and (2) the tree to be generated must include at least one non-star (i.e., nontrivial) node. This is because if all the nodes were star-nodes, then none of them would satisfy *min_sup*. Therefore, it would be a complete waste to compute them. This pruning is observed in Figures 5.11 and 5.12. For example, the left subtree extending from node a_1 in the base-tree in Figure 5.11 does not include any non-star-nodes. Therefore, the a_1CD/a_1 subtree should not have been generated. It is shown, however, for illustration of the child tree generation process.

Star-Cubing is sensitive to the ordering of dimensions, as with other iceberg cube construction algorithms. For best performance, the dimensions are processed in order of decreasing cardinality. This leads to a better chance of early pruning, because the higher the cardinality, the smaller the partitions, and therefore the higher possibility that the partition will be pruned.

Star-Cubing can also be used for full cube computation. When computing the full cube for a dense data set, Star-Cubing's performance is comparable with MultiWay and is much faster than BUC. If the data set is sparse, Star-Cubing is significantly faster than MultiWay and faster than BUC, in most cases. For

Algorithm: Star-Cubing. Compute iceberg cubes by Star-Cubing.

Input:

- R: a relational table
- *min_support*: minimum support threshold for the iceberg condition (taking count as the measure).

Output: The computed iceberg cube.

Method: Each star-tree corresponds to one cuboid tree node, and vice versa.

```
BEGIN
        scan $R$ twice, create star-table $S$ and star-tree $T$;
        output count of T.root;
        call starcubing(T, T.root);
END

procedure starcubing(T, cnode)// cnode: current node
{
(1)     for each non-null child $C$ of $T$'s cuboid tree
(2)             insert or aggregate cnode to the corresponding
                    position or node in $C$'s star-tree;
(3)     if (cnode.count $\geq$ min_support) then {
(4)             if (cnode $\neq$ root) then
(5)                 output cnode.count;
(6)             if (cnode is a leaf) then
(7)                 output cnode.count;
(8)             else { // initiate a new cuboid tree
(9)                 create $C_C$ as a child of $T$'s cuboid tree;
(10)                let $T_C$ be $C_C$'s star-tree;
(11)                $T_C.root's\ count = cnode.count$;
(12)            }
(13)    }
(14)    if (cnode is not a leaf) then
(15)            starcubing(T, cnode.first_child);
(16)    if ($C_C$ is not null) then {
(17)            starcubing($T_C$, $T_C.root$);
(18)            remove $C_C$ from $T$'s cuboid tree; }
(19)    if (cnode has sibling) then
(20)            starcubing(T, cnode.sibling);
(21)    remove $T$;
}
```

Figure 5.13: The Star-Cubing algorithm.

iceberg cube computation, Star-Cubing is faster than BUC, where the data are skewed and the speedup factor increases as *min_sup* decreases.

5.2.4 Precomputing Shell Fragments for Fast High-Dimensional OLAP

Recall the reason that we are interested in precomputing data cubes: Data cubes facilitate fast on-line analytical processing (OLAP) in a multidimensional data space. However, a full data cube of high dimensionality needs massive storage space and unrealistic computation time. Iceberg cubes provide a more feasible alternative, as we have seen, wherein the iceberg condition is used to specify the computation of only a subset of the full cube's cells. However, although an iceberg cube is smaller and requires less computation time than its corresponding full cube, it is not an ultimate solution. For one, the computation and storage of the iceberg cube can still be costly. For example, if the base cuboid cell, $(a_1, a_2, \ldots, a_{60})$, passes minimum support (or the iceberg threshold), it will generate 2^{60} iceberg cube cells. Second, it is difficult to determine an appropriate iceberg threshold. Setting the threshold too low will result in a huge cube, whereas setting the threshold too high may invalidate many useful applications. Third, an iceberg cube cannot be incrementally updated. Once an aggregate cell falls below the iceberg threshold and is pruned, its measure value is lost. Any incremental update would require recomputing the cells from scratch. This is extremely undesirable for large real-life applications where incremental appending of new data is the norm.

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

Instead of computing a cube shell, we can compute only portions or fragments of it. This section discusses the *shell fragment* approach for OLAP query processing. It is based on the following key observation about OLAP in high-dimensional space. Although a data cube may contain many dimensions, *most OLAP operations are performed on only a small number of dimensions at a time.* In other words, an OLAP query is likely to ignore many dimensions (i.e., treating them as irrelevant), fix some dimensions (e.g., using query constants as instantiations), and leave only a few to be manipulated (for drilling, pivoting, etc.). This is because it is neither realistic nor fruitful for anyone to comprehend the changes of thousands of cells involving tens of dimensions simultaneously in a high-dimensional space at the same time. Instead, it is more natural to first locate some cuboids of interest and then drill along one or two dimensions to examine the changes of a few related dimensions. Most analysts will only need to examine, at any one moment, the combinations of a small number of dimensions. This implies that if multidimensional aggregates can be computed quickly on a *small number of dimensions inside a high-dimensional space*, we may still achieve fast OLAP without materializing the original high-dimensional data cube. Computing the full cube (or, often, even an iceberg cube or cube shell) can be excessive. Instead, a *semi-on-line computation model with certain preprocessing* may offer a more feasible solution. Given a base cuboid, some quick preparation computation can be done first (i.e., off-line). After that, a query can then be computed on-line using the preprocessed data.

The shell fragment approach follows such a semi-on-line computation strategy. It involves two algorithms: one for computing cube shell fragments and the other for query processing with the cube fragments. The shell fragment approach can handle databases of high dimensionality and can quickly compute small local cubes on-line. It explores the *inverted index* data structure, which is popular in information retrieval and Web-based information systems. The basic idea is as follows. Given a high-dimensional data set, we partition the dimensions into a set of disjoint dimension *fragments*, convert each fragment into its corresponding inverted index representation, and then construct *cube shell fragments* while keeping the inverted indices associated with the cube cells. Using the precomputed cubes shell fragments, we can dynamically assemble and compute cuboid cells of the required data cube on-line. This is made efficient by set intersection operations on the inverted indices.

To illustrate the shell fragment approach, we use the tiny database of Table 5.4 as a running example. Let the cube measure be `count()`. Other measures will be discussed later. We first look at how to construct the inverted index for the given database.

Example 5.9 **Construct the inverted index.** For each attribute value in each dimension, list the tuple identifiers (*TIDs*) of all the tuples that have that value. For example, attribute value a_2 appears in tuples 4 and 5. The TIDlist for a_2 then contains exactly two items, namely 4 and 5. The resulting inverted index table is shown in Table 5.5. It retains all of the information of the original database. If each table entry takes one unit of memory, Tables 5.4 and 5.5 each takes 25 units, i.e., the inverted index table uses the same amount of memory as the original database.

"How do we compute shell fragments of a data cube?" The shell fragment computation algorithm, **Frag-Shells**, is summarized in Figure 5.14. We first partition all the dimensions of the given data set into independent groups of dimensions, called *fragments* (line 1). We scan the base cuboid and construct an inverted index for each attribute (lines 2 to 6). Line 3 is for when the measure is other than the tuple `count()`, which will be described later. For each fragment, we compute the full *local* (i.e., fragment-based) data cube while retaining the inverted indices (lines 7 to 8). Consider a database of 60 dimensions, namely, $A_1, A_2, \ldots, A_{60}$. We can first partition the 60 dimensions into 20 fragments of size 3: (A_1, A_2, A_3), (A_4, A_5, A_6), …, (A_{58}, A_{59}, A_{60}). For each fragment, we compute its full data cube while recording the inverted indices. For example, in fragment (A_1, A_2, A_3), we would compute seven cuboids: A_1, A_2, A_3, A_1A_2, A_2A_3, A_1A_3, $A_1A_2A_3$. Furthermore, an inverted index is retained for each cell in the cuboids. That is, for each cell, its associated TIDlist

Table 5.4: The original database.

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

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

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

Output:

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

Method:

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

Figure 5.14: Algorithm for shell fragment computation.

is recorded.

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

Table 5.5: The inverted index.

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

Table 5.6: Cuboid AB.

Cell	*Intersection*	*Tuple ID List*	*List Size*
(a_1, b_1)	$\{1, 2, 3\} \cap \{1, 4, 5\}$	$\{1\}$	1
(a_1, b_2)	$\{1, 2, 3\} \cap \{2, 3\}$	$\{2, 3\}$	2
(a_2, b_1)	$\{4, 5\} \cap \{1, 4, 5\}$	$\{4, 5\}$	2
(a_2, b_2)	$\{4, 5\} \cap \{2, 3\}$	$\{\}$	0

Table 5.7: Cuboid DE.

Cell	*Intersection*	*Tuple ID List*	*List Size*
(d_1, e_1)	$\{1, 3, 4, 5\} \cap \{1, 2\}$	$\{1\}$	1
(d_1, e_2)	$\{1, 3, 4, 5\} \cap \{3, 4\}$	$\{3, 4\}$	2
(d_1, e_3)	$\{1, 3, 4, 5\} \cap \{5\}$	$\{5\}$	1
(d_2, e_1)	$\{2\} \cap \{1, 2\}$	$\{2\}$	1

experts or the past history of OLAP queries.

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

Example 5.10 **Compute shell fragments.** Suppose we are to compute the shell fragments of size 3. We first divide the five dimensions into two fragments, namely (A, B, C) and (D, E). For each fragment, we compute the full local data cube by intersecting the TIDlists in Table 5.5 in a top-down depth-first order in the cuboid lattice. For example, to compute the cell (a_1, b_2, *), we intersect the tuple ID lists of a_1 and b_2 to obtain a new list of $\{2, 3\}$. Cuboid AB is shown in Table 5.6.

After computing cuboid AB, we can then compute cuboid ABC by intersecting all pairwise combinations between Table 5.6 and the row c_1 in Table 5.5. Notice that because cell (a_2, b_2) is empty, it can be effectively discarded in subsequent computations, based on the Apriori property. The same process can be applied to compute fragment (D, E), which is completely independent from computing (A, B, C). Cuboid DE is shown in Table 5.7.

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

To illustrate the design of the *ID_measure* array, let's look at the following example.

Example 5.11 **Computing cubes with the average() measure.** Table 5.8 shows an example sales database where each tuple has two associated values, such as item_count and sum, where item_count is the count of items sold.

To compute a data cube for this database with the measure average(), we need to have a TIDlist for each cell: $\{TID_1, \ldots, TID_n\}$. Because each TID is uniquely associated with a particular set of measure values, all future computation just needs to fetch the measure values associated with the tuples in the list. In other words, by keeping an *ID_measure* array in memory for on-line processing, we can handle complex algebraic measures, such as average, variance, and standard deviation. Table 5.9 shows what exactly should be kept for our example, which is substantially smaller than the database itself.

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

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

In a **point query**, all of the *relevant* dimensions in the cube have been instantiated (that is, there are no *inquired* dimensions in the relevant set of dimensions). For example, in an n-dimensional data cube, $A_1A_2 \ldots A_n$, a point query could be in the form of $\langle A_1, A_5, A_9 : M? \rangle$, where $A_1 = \{a_{11}, a_{18}\}$, $A_5 = \{a_{52}, a_{55}, a_{59}\}$, $A_9 = a_{94}$, and M is the inquired measure for each corresponding cube cell. For a cube with a small number of dimensions, we can use "*" to represent a "don't care" position where the corresponding dimension is *ir-*

Table 5.8: A database with two measure values.

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

relevant, that is, neither inquired nor instantiated. For example, in the query $\langle a_2,\ b_1,\ c_1,\ d_1,\ * : \mathsf{count()}?\rangle$ for the database in Table 5.4, the first four dimension values are instantiated to $a_2,\ b_1,\ c_1$, and d_1, respectively, while the last dimension is irrelevant, and count() (which is the tuple count by context) is the inquired measure.

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

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

Let the point query be of the form $\langle \alpha_i, \alpha_j, \alpha_k, \alpha_p : M?\rangle$, where α_i represents a set of instantiated values of dimension A_i, and so on for α_j, α_k, and α_p. First, we check the shell fragment schema to determine which dimensions among A_i, A_j, A_k, and A_p are in the same fragment(s). Suppose A_i and A_j are in the same fragment, while A_k and A_p are in two other fragments. We fetch the corresponding TIDlists on the precomputed 2-D fragment for dimensions A_i and A_j using the instantiations α_i and α_j, and fetch the TIDlists on the 1-D fragments for dimensions A_k and A_p using the instantiations α_k and α_p, respectively. The obtained TIDlists are intersected to derive the TIDlist table. This table is then used to derive the specified measure (e.g., by taking the length of the TIDlists for tuple count(), or by fetching item_count() and sum() from the *ID_measure* array to compute average()) for the final set of cells.

Example 5.12 **Point query.** Suppose a user wants to compute the point query, $\langle a_2,\ b_1,$

Table 5.9: *ID_measure* array of Table 5.8.

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

$c_1,\ d_1,\ *$: count()?$\rangle$, for our database in Table 5.4 and that the shell fragments for the partitions (A, B, C) and (D, E) are precomputed as described in Example 4.10. The query is broken down into two subqueries based on the precomputed fragments: $\langle a_2,\ b_1,\ c_1,\ *,\ *\rangle$ and $\langle *,\ *,\ *,\ d_1,\ *\rangle$. The best fit precomputed shell fragments for the two subqueries are ABC and D. The fetch of the TIDlists for the two subqueries returns two lists: {4, 5} and {1, 3, 4, 5}. Their intersection is the list {4, 5}, which is of size 2. Thus the final answer is count() = 2.

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

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

Example 5.13 **Subcube query.** Suppose a user wants to compute the subcube query, $\langle a_2, b_1, ?, *, ? :$ count()?$\rangle$, for our database in Table 5.4, and that the shell fragments have been precomputed as described in Example 4.10. The query can be broken into three best-fit fragments according to the instantiated and inquired dimensions: AB, C, and E, where AB has the instantiation (a_2, b_1). The fetch of the TIDlists for these partitions returns: (a_2, b_1):{4, 5}, (c_1):{1, 2, 3, 4, 5}, and {(e_1:{1, 2}), (e_2:{3, 4}), (e_3:{5})}, respectively. The intersection of these corresponding TIDlists contains a cuboid with two tuples: $\{(c_1, e_2){:}\{4\}$[5], $(c_1, e_3){:}\{5\}\}$. This base cuboid can be used to compute the 2-D data cube, which is trivial.

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

5.3 Processing Advanced Kinds of Queries by Exploring Cube Technology

Data cubes are not confined to the simple multidimensional structure illustrated above for typical business data warehouse applications. The methods described in this section further develop data cube technology for effective processing of advanced kinds of queries. Section 5.3.1 explores *sampling cubes*. This extension of data cube technology can be used to answer queries on *sample data* (such as survey data, which represent a sample or subset of a target data population of interest). Section 5.3.2 explains how *ranking cubes* can be computed to answer top-k queries, such as “find the top 5 cars” according to some user-specified criteria.

The basic data cube structure has been further extended for various sophisticated types of data and new applications. Here we list some examples, such as *spatial data cubes* for the design and implementation of *geo-spatial data warehouses*, and *multimedia data cubes* for the multidimensional analysis of *multimedia data* (those containing images and videos). *RFID data cubes* handle the compression and multidimensional analysis of *RFID* (*i.e.*, radio-frequency identification) data. *Text cubes* and *topic cubes* were developed for the application of vector-space models and generative language models, respectively, in the analysis of *multidimensional text databases* (which contain both structure attributes and narrative text attributes). Data mining techniques with these cubes will be introduced in the second volume of this book.

5.3.1 Sampling Cubes: OLAP-based Mining on Sampling Data

When collecting data, we often collect only a *subset* of the data we would ideally like to gather. In statistics, this is know as collecting a **sample** of the data population. The resulting data are called **sample data**. Data are often

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

sampled to save on costs, manpower, time, and materials. In many applications, the collection of the entire data population of interest is unrealistic. In the study of TV ratings or pre-election polls, for example, it is impossible to gather the opinion of *everyone* in the population. Most published ratings or polls rely on a sample of the data for analysis. The results are extrapolated for the entire population, and associated with certain statistical measures, such as a *confidence interval*. The confidence interval tells us how reliable a result is. Statistical surveys based on sampling are a common tool in many fields, like politics, healthcare, market research, and social and natural sciences.

"How effective is OLAP on sample data?" OLAP traditionally has the full data population on hand, yet with sample data, we have only a small subset. If we try to apply traditional OLAP tools to sample data, we encounter three challenges. First, sample data are often sparse in the multidimensional sense. When a user drills down on the data, it is easy to reach a point with very few or no samples even when the overall sample size is large. Traditional OLAP simply uses whatever data are available to compute a query answer. To extrapolate such an answer for a population based on a small sample could be misleading: A single outlier or a slight bias in the sampling can distort the answer significantly. Second, with sample data, statistical methods are used to provide a measure of reliability (such as a confidence interval) to indicate the quality of the query answer as it pertains to the population. Traditional OLAP is not equipped with such tools. A *sampling cube* framework was introduced to tackle each of the above challenges.

The Sampling Cube Framework

The **sampling cube** is a data cube structure that stores the sample data and their multidimensional aggregates. It supports OLAP on sample data. It calculates confidence intervals as a quality measure, for any multidimensional query. Given a sample data relation (i.e., base cuboid) R, the sampling cube C_R typically computes the sample mean, sample standard deviation, and other task-specific measures. A confidence interval is associated with each computed measure.

In statistics, a *confidence interval* is used to indicate the reliability of an estimate. Suppose we want to estimate the mean age of all viewers of a given TV show. We have sample data (a subset) of this data population. Let's say our sample mean is 35 years. This becomes our estimate for the entire population of viewers as well, but how confident can we be that 35 is also the mean of the true population? It is unlikely that the sample mean will be exactly equal to the true population mean because of sampling error. Therefore, we need to qualify our estimate in some way to indicate the general magnitude of this error. This is typically done by computing a **confidence interval**, which is an *estimated range of values with a given high probability of covering the true population value.* A confidence interval for our example could be *"the actual mean will not vary by +/- two standard deviations 95% of the time"*. (Recall that the standard deviation is just a number, which can be computed as shown in Section 2.2.2.)

A confidence interval is always qualified by a particular *confidence level.* In our example, it is 95%.

The confidence interval is calculated as follows. Let x be a set of samples. The mean of the samples is denoted by $\bar{x}$, and the number of samples in x is denoted by l. Assuming that the standard deviation of the population is unknown, the *sample* standard deviation of x is denoted by s. Given a desired confidence level, the **confidence interval** for $\bar{x}$ is

$$\bar{x} \pm t_c \hat{\sigma}_{\bar{x}} \tag{5.1}$$

where t_c is the *critical t-value* associated with the confidence level and $\hat{\sigma}_{\bar{x}} = \frac{s}{\sqrt{l}}$ is the *estimated standard error of the mean.* To find the appropriate t_c, we specify the desired confidence level (*e.g.*, 95%) and also the *degree of freedom*, which is just $l-1$.

The important thing to note is that the computation involved in computing a confidence interval is *algebraic.* Let's look at the three terms involved in Equation (5.1). The first is the mean of the sample set, $\bar{x}$, which is algebraic; the second is the critical t-value, which is calculated by a lookup, and with respect to x, it depends on l, a distributive measure; and the third is $\hat{\sigma}_{\bar{x}} = \frac{s}{\sqrt{l}}$, which also turns out to be algebraic if one records the linear sum ($\sum_{i=1}^{l} x_i$) and squared sum ($\sum_{i=1}^{l} x_i^2$). Because the terms involved are either algebraic or distributive, the confidence interval computation is algebraic. Actually, since both the mean and confidence interval are algebraic, at every cell, exactly three values are sufficient to calculate them—all of which are either distributive or algebraic. These are:

1. l
2. $sum = \sum_{i=1}^{l} x_i$
3. $squared\ sum = \sum_{i=1}^{l} x_i^2$

There are many efficient techniques for computing algebraic and distributive measures (Section 4.2.5). Therefore, any of the previously developed cubing algorithms can be used to efficiently construct a sampling cube.

Now that we have established that sampling cubes can be computed efficiently, our next step is to find a way of boosting the confidence of results obtained for queries on sample data.

Query Processing: Boosting Confidences for Small Samples

A query posed against a data cube can be either a *point query* or a *range query.* Without loss of generality, consider the case of a point query. Here, it corresponds to a cell in sampling cube C_R. The goal is to provide an accurate point estimate for the samples in that cell. Because the cube also reports the confidence interval associated with the sample mean, there is some measure of "reliability" to the returned answer. If the confidence interval is small, the reliability is deemed good; however, if the interval is large, the reliability is questionable.

"What can we do to boost the reliability of query answers?" Consider what affects the size of the confidence interval. There are two main factors – the variance of the sample data, and the sample size. First, a rather large variance in the cell may indicate that the chosen cube cell is poor for prediction. A better solution is probably to drill down on the query cell to a more specific one, *i.e.*, asking more specific queries. Second, a small sample size can cause a large confidence interval. When there are very few samples, the corresponding t_c is large because of the small degree of freedom. This in turn could cause a large confidence interval. Intuitively, this makes sense. Suppose one is trying to figure out the average income of people in the United States. Just asking two or three people does not give much confidence to the returned response.

The best way to solve this small sample size problem is to get more data. Fortunately, there is usually an abundance of additional data available in the cube. The data do not match the query cell exactly, however, we can consider data from cells that are "close by". There are two ways to incorporate such data to enhance the reliability of the query answer: (i) *intra-cuboid query expansion*, where we consider nearby cells *within* the same cuboid, and (ii) *inter-cuboid query expansion*, where we consider more general versions (from parent cuboids) of the query cell. Let's see how this works, starting with intra-cuboid query expansion.

Method 1. Intra-cuboid query expansion. Here, we expand the sample size by including nearby cells in the *same* cuboid as the queried cell, as shown in Figure 5.15(a). We just have to be careful that the new samples serve to increase the confidence in the answer without changing the semantics of the query.

So, the first question is "*Which dimensions should be expanded?*" The best candidates should be the dimensions that are *uncorrelated* or *weakly correlated* with the measure value (*i.e.*, the value to be predicted). Expanding within these dimensions will likely increase the sample size and not shift the answer of the query. Consider an example of a 2-D query specifying `Education` = "College" and `Birth_Month` = "July". Let the cube measure be `average Income`. Intuitively, education has a high correlation to income while birth month does not. It would be harmful to expand the `Education` dimension to include values such as "Graduate" or "High School." They are likely to alter the final result. However, expansion in the `Birth_Month` dimension to include other month values could be helpful, because it is unlikely to change the result but will increase sampling size.

To mathematically measure the correlation of a dimension to the cube value, the correlation between the dimension's values and their aggregated cube measures is computed. *Pearson's correlation coefficient* for numeric data and the χ^2 correlation test for nominal data are popularly used correlation measures, although many other measures, such as *covariance*, can be used. (These measures were presented in Section 3.3.2.) A dimension that is strongly correlated with

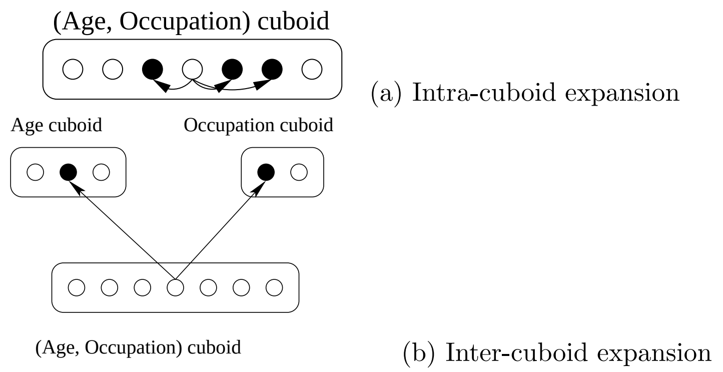


Figure 5.15: Query expansion within sampling cube: Given samll data samples, both methods use strategies to boost the reliability of query answers by considering additional data cell values. (a) Intra-cuboid expansion considers nearby cells in the same cuboid as the queried cell. (b) Inter-cuboid expansion considers more general cells from parent cuboids.

the value to be predicted should *not* be a candidate for expansion. Notice that since the correlation of a dimension with the cube measure is independent of a particular query, it should be precomputed and stored with the cube measure to facilitate efficient online analysis.

After selecting dimensions for expansion, the next question is "*Which values within these dimensions should the expansion use?*" This relies on the semantic knowledge of the dimensions in question. The goal should be to select semantically similar values in order to minimize the risk of altering the final result. Consider the `Age` dimension – similarity of values in this dimension is clear. There is a definite (numeric) order to the values. Dimensions with *numeric* or *ordinal* (ranked) data (like `Education`) have a definite ordering among data values. Therefore, we can select values that are close to the instantiated query value. For nominal data of a dimension that is organized in a multilevel hierarchy in a data cube (such as `Location`), we should select those values located in the same branch of the tree (such as in the same district or city).

By considering additional data during query expansion, we are aiming for a more accurate and reliable answer. As mentioned above, strongly correlated dimensions are precluded from expansion for this purpose. An additional strategy is to ensure that new samples share the "same" cube measure value (*e.g.*, mean income) as the existing samples in the query cell. The two sample **`t-test`** is a relatively simple statistical method that can be used to determine whether two samples have the same mean (or any other point estimate), where "same" means that they do not differ significantly. (It is described in greater detail in Section 8.5.5 on estimating confidence intervals.) The test determines whether two samples have the same mean (the null hypothesis) with the only assumption being that they are both normally distributed. The test fails if there is evidence that the two samples do not share the same mean. Furthermore, the test can be performed with a confidence level as an input. This allows the user to control how strict or loose the query expansion will be.

The next example shows how the strategies described above for intra-cuboid expansion can be used to answer a query on sample data.

Example 5.14 **Intra-cuboid query expansion to answer a query on sample data.** Consider a book retailer trying to learn more about its customers' annual income levels. In Table 5.10, a sample of the survey data collected is shown[6]. In the survey, customers are segmented by four attributes, namely `Gender`, `Age`, `Education`, and `Occupation`.

Gender	Age	Education	Occupation	Income
Female	23	College	Teacher	$85,000
Female	40	College	Programmer	$50,000
Female	31	College	Programmer	$52,000
Female	50	Graduate	Teacher	$90,000
Female	62	Graduate	CEO	$500,000
Male	25	Highschool	Programmer	$50,000
Male	28	Highschool	CEO	$250,000
Male	40	College	Teacher	$80,000
Male	50	College	Programmer	$45,000
Male	57	Graduate	Programmer	$80,000

Table 5.10: Sample customer survey data.

Let a query on customer income be "`Age` = 25", where the user specifies a 95% confidence level. Suppose this returns an `Income` value of $50,000 with a rather large confidence interval[7]. Suppose also, that this confidence interval is larger than a preset threshold and that the `Age` dimension was found to have little correlation with `Income` in this dataset. Therefore, intra-cuboid expansion starts within the `Age` dimension. The nearest cell is "`Age` = 23," which returns an `Income` of $85,000. The two sample t-test at the 95% confidence level passes so the query expands; it is now "`Age` = {23, 25}" with a smaller confidence interval than initially. However, it is still larger than the threshold so expansion continues to the next nearest cell: "`Age` = 28", which returns an `Income` of $250,000. The two sample t-test between this cell and the original query cell fails; as a result, it is ignored. Next, "`Age` = 31" is checked and it passes the test. The confidence interval of the three cells combined is now below the threshold and the expansion finishes at "`Age` = {23, 25, 31}." The mean of the `Income` values at these three cells is $\frac{85{,}000+50{,}000+52{,}000}{3} = \$62,333$, which is returned as the query answer. It has a smaller confidence interval and thus is

[6]For the sake of illustration, ignore the fact that the sample size is too small to be statistically significant.

[7]For the sake of the example, suppose this is true even though there is only one sample. In practice, more points are needed to calculate a legitimate value.

more reliable than the response of \$50,000, which would have been returned if intra-cuboid expansion had not been considered.

Method 2. Inter-cuboid query expansion. In this case, the expansion occurs by looking to a *more general cell*, as shown in Figure 5.15(b). For example, the cell in the 2-D cuboid `Age-Occupation` can use its parent in either of the 1-D cuboids, `Age` or `Occupation`. Think of inter-cuboid expansion as just an extreme case of intra-cuboid expansion, where *all* the cells within a dimension are used in the expansion. This essentially sets the dimension to $*$ and thus generalizes to a higher level cuboid.

A k-dimensional cell has k direct parents in the cuboid lattice, where each parent is $(k-1)$-dimensional. There are *many* more ancestor cells in the data cube (e.g., if multiple dimensions are rolled up simultaneously). However, we choose only one parent here to make the search space tractable and to limit the change in the semantics of the query. As with intra-cuboid query expansion, correlated dimensions are not allowed in inter-cuboid expansions. Within the uncorrelated dimensions, the two sample t-test can be performed to confirm that the parent and the query cell share the same sample mean. If multiple parent cells pass the test, the confidence level of the test can be adjusted progressively higher until only one passes. Alternatively, multiple parent cells can be used to boost the confidence simultaneously. The choice is application dependent.

Example 5.15 **Inter-cuboid expansion to answer a query on sample data.** Given the input relation in Table 5.10, let the query on `Income` be "`Occupation` = Teacher $\wedge$ `Gender` = Male." There is only one sample in Table 5.10 that matches the query, and it has an `Income` of \$80,000. Suppose the corresponding confidence interval is larger than a preset threshold. We use inter-cuboid expansion to find a more reliable answer. There are two parent cells in the data cube: "`Gender` = Male" and "`Occupation` = Teacher." By moving up to "`Gender` = Male" (and thus setting `Occupation` to $*$), the mean `Income` is \$101,000. A two sample t-test reveals that this parent's sample mean differs significantly from that of the original query cell, so it is ignored. Next, "`Occupation` = Teacher" is considered. It has a mean `Income` of \$85,000 and passes the two sample t-test. As a result, the query is expanded to "`Occupation` = Teacher" and an `Income` value of \$85,000 is returned with acceptable reliability.

"*How can we determine which method to choose – intra-cuboid expansion or inter-cuboid expansion?*" This is difficult to answer without knowing the data and the application. A strategy for choosing between the two is to consider what the tolerance is for change in the semantics of the query. This depends on the specific dimensions chosen in the query. For instance, the user might tolerate a bigger change in semantics for the `Age` dimension than `Education`. The difference in tolerance could be so large that the user is willing to set `Age` to $*$ (*i.e.*, inter-cuboid expansion) rather than letting `Education` change at all. Domain knowledge is helpful here.

So far, our discussion has only focused on full materialization of the sampling cube. In many real-world problems, this is often impossible, especially for high dimensional cases. Real-world survey data, for example, can easily contain over 50 variables (i.e., dimensions). The sampling cube size would grow exponentially with the number of dimensions. To handle high-dimensional data, a sampling cube method called *Sampling Cube Shell* was developed. It integrates the Frag-Shell method of Section 5.2.4 with the query expansion approach discussed above. The shell computes only a subset of the full sampling cube. The subset should consist of relatively low dimensional cuboids (that are commonly queried) and cuboids that offer the most benefit to the user. The details are left to interested readers as an exercise. The method was tested on both real and synthetic data and found to be efficient and effective in answering queries.

5.3.2 Ranking Cubes: Efficient Computation of Top-k Queries

The data cube helps not only online analytical processing of multidimensional queries but also search and data mining. In this section, we introduce a new cube structure called *Ranking Cube* and examine how it contributes to the efficient processing of *top-k queries*. Instead of returning a large set of indiscriminative answers to a query, a **top-k query** (or **ranking query**) returns only the best k results according to a user-specified preference. The results are returned in ranked order, so that the best is at the top. The user-specified preference generally consists of two components: a *selection condition* and a *ranking function.* Top-k queries are common in many applications like searching Web databases, k-nearest neighbor searches with approximate matches, and similarity queries in multimedia databases.

An example of a top-k query is as follows.

Example 5.16 **A top-k query.** Consider an online used car database, R, that maintains the following information for each car: maker (*e.g.*, Ford, Honda), model (*e.g.*, Taurus, Honda), type (*e.g.*, sedan, convertible), color (*e.g.*, red, silver), transmission (*e.g.*, auto, manual), price, mileage, and so on. A typical top-k query over this database is:

Q_1: `select` top 5 * `from` R
where maker = “Ford” `and` type = “sedan”
`order by` $(price - 10K)^2 + (mileage - 30K)^2$ `asc`

Within the dimensions (or attributes) for R, *maker* and *type* are used here as **selection dimensions**. The **ranking function** is given in the “`order by`” clause. It specifies the **ranking dimensions**, *price* and *mileage*. Q_1 searches for the top-5 sedans made by Ford. The entries found are ranked or sorted in ascending (“`asc`”) order, according to the ranking function. The ranking function is formulated so that entries whose price and mileage are closest to the user's specified values of \$$10K$ and $30K$, respectively, appear towards the top of the list.

The database may have many dimensions that could be used for selection, describing, for example, whether a car has power windows, air conditioning, or a sunroof. Users may pick any subset of dimensions and issue a top-k query using their preferred ranking function. There are many other similar application scenarios. For example, when searching for hotels, ranking functions are often constructed based on price and distance to an area of interest. Selection conditions can be imposed on, say, the hotel location district, the star rating, and whether the hotel offers complimentary treats or Internet access. The ranking functions may be linear, quadratic or any other form.

As shown in the above examples, individual users may not only propose *ad hoc* ranking functions, but also have different data subsets of interest. Users often want to thoroughly study the data via *multi-dimensional analysis* of the top-k query results. For example, if unsatisfied by the top-5 results returned by Q_1, the user may roll up on the maker dimension to check the top-5 results on all sedans. The dynamic nature of the problem imposes a great challenge to researchers. OLAP requires off-line pre-computation so that multi-dimensional analysis can be performed on the fly, yet the ad-hoc ranking functions prohibit full materialization. A natural compromise is to adopt a *semi off-line materialization* and *semi online computation* model.

Suppose a relation R has selection dimensions $(A_1, A_2, \ldots, A_S)$ and ranking dimensions $(N_1, N_2, \ldots, N_R)$. Values in each ranking dimension can be partitioned into multiple intervals according to the data and expected query distributions. Regarding the price of used cars, for example, we may have, say, these four partitions (or value ranges): $\leq 5K$, $[5 - 10K)$, $[10 - 15K)$, and $\geq 15K$. A ranking cube can be constructed by performing multidimensional aggregations on selection dimensions. We can store the count for each partition of each ranking dimension, thereby making the cube "rank-aware". The top-k queries can be answered by first accessing the cells in the more preferred value ranges before consulting the cells in the less preferred value ranges.

Example 5.17 **Using a ranking cube to answer a top-k query.** Suppose Table 5.11 shows C_{MT}, a materialized (i.e., precomputed) cuboid of a ranking cube for used car sales. The cuboid, C_{MT}, is for the selection dimensions *maker* and *type*. It shows the count and corresponding transaction ID numbers (*TIDs*) for various partitions of the ranking dimensions, *price* and *mileage*.

Maker	*Type*	*Price*	*Mileage*	*count*	*TIDs*
Ford	sedan	<5K	30-40K	7	$t_6, \ldots, t_{68}$
Ford	sedan	5-10K	30-40K	50	$t_{15}, \ldots, t_{152}$
Honda	sedan	10-15K	30-40K	20	$h_8, \ldots, h_{32}$
...	...	...	...	...	...

Table 5.11: A cuboid of a ranking cube for used car sales.

Query Q_1 can be answered by using a selection condition to select the appropriate selection dimension values (*i.e.*, maker = "Ford" and type = "sedan") in cuboid C_{MT}. In addition, the ranking function "$(price - 10K)^2 + (mileage - 30K)^2$" is used to find those tuples that most closely match the user's criteria. If there are not enough matching tuples found in the closest matching cells, the next closest matching cells will need to be accessed. We may even drill down to the corresponding lower level cells to see the count distributions of cells that match the ranking function and additional criteria regarding, say, model, maintenance situation, or other loaded features. Only users who really want to see more detailed information, such as interior photos, will need to access the physical records stored in the database.

Most real life top-k queries are likely to involve only a small subset of selection attributes. To support high-dimensional ranking cubes, we can carefully select the cuboids that need to be materialized. For example, we could choose to materialize only the 1-D cuboids that contain single selection dimensions. This will achieve low space overhead and still have high performance when the number of selection dimensions is large. In some cases, there may exist many ranking dimensions to support multiple users with rather different preferences. For example, buyers may search for houses by considering various factors like price, distance to school or shopping, number of years old, floor space, and tax. In this case, a possible solution is to create multiple data partitions, each of which consists of a subset of the ranking dimensions. The query processing may need to search over a joint space involving multiple data partitions.

In summary, the general philosophy of ranking cubes is to materialize such cubes on the set of selection dimensions. Use of the interval-based partitioning in ranking dimensions makes the ranking cube efficient and flexible at supporting *ad hoc* user queries. Various implementation techniques and query optimization methods have been developed for efficient computation and query processing based on this framework.

5.4 Multidimensional Data Analysis in Cube Space

Data cubes create a flexible and powerful means to group and aggregate subsets of data. They allow data to be explored in multiple dimensional combinations and at varying aggregate granularities. This capability greatly increases the analysis bandwidth and helps effective discovery of interesting patterns and knowledge from data. The use of cube space makes the data space both meaningful and tractable.

This section presents methods of multidimensional data analysis that make use of data cubes to organize data into intuitive regions of interest at varying granularities. Section 5.4.1 presents *prediction cubes*, a technique for multidimensional data mining that facilitates predictive modeling in multidimensional space. Section 5.4.2 describes how to construct *multifeature cubes.* These support complex analytical queries involving multiple dependent aggregates at multiple granularities. Finally, Section 5.4.3 describes an interactive method for users to systematically explore cube space. In such *exception-based discovery-*

driven exploration, interesting exceptions or anomalies in the data are automatically detected and marked for users with visual cues.

5.4.1 Prediction Cubes: Prediction Mining in Cube Space

Recently, researchers have turned their attention towards **multidimensional data mining** to uncover knowledge at varying dimensional combinations and granularities. Such mining is also known as *exploratory multidimensional data mining* and *online analytical data mining (OLAM).* Multidimensional data space is huge. In preparing the data, how can we identify the interesting subspaces for exploration? To what granularities should we aggregate the data? Multidimensional data mining in cube space organizes data of interest into intuitive regions at various granularities. It analyzes and mines the data by applying various data mining techniques systematically over these regions.

There are at least four ways in which OLAP-style analysis can be fused with data mining techniques:

1. *Use cube space to define the data space for mining.* Each region in cube space represents a subset of data over which we wish to find interesting patterns. Cube space is defined by a set of expert-designed, informative dimension hierarchies, not just a set of arbitrary subsets of data. Therefore, the use of cube space makes the data space both meaningful and tractable.

2. *Use OLAP queries to generate features and targets for mining.* The features and even the targets (that we wish to learn to predict) can sometimes be naturally defined as OLAP aggregate queries over regions in cube space.

3. *Use data-mining models as building blocks in a multi-step mining process.* Multidimensional data mining in cube space may consist of multiple steps, where data-mining models can be viewed as building blocks that are used to describe the behavior of interesting data sets, rather than the end results.

4. *Use data-cube computation techniques to speed up repeated model construction.* Multidimensional data mining in cube space may require building a model for each candidate data space, which is usually too expensive to be feasible. However, by carefully sharing computation across model-construction for different candidates based on data-cube computation techniques, efficient mining is achievable.

In this section we study *prediction cubes*, an example of multidimensional data mining where the cube space is explored for prediction tasks. A **prediction cube** is a cube structure that stores prediction models in multidimensional data space and supports prediction in an OLAP manner. Recall that in a data cube, each cell value is an aggregate number (e.g., `count`) computed over the subset of data in that cell. However, each cell value in a prediction cube is computed by evaluating a predictive model built on the subset of data in that cell, thereby representing the predictive behavior of that subset. Instead of seeing prediction models as the end result, prediction cubes use prediction models as building blocks to define the interestingness of subsets of data, that is, they identify subsets of data that indicate more accurate prediction.

This is best explained with an example.

Example 5.18 **Prediction cube for identification of interesting cube subspaces.** Suppose a company has a customer table with the attributes: `Time` (with two levels of granularity: `Month` and `Year`), `Location` (with two levels of granularity: `State` and `Country`), `Gender`, `Salary`, and one class-label attribute: `Valued_Customer`. A manager wants to analyze the decision process of whether a customer is highly-valued with respect to `Time` and `Location`. In particular, he is interested in the question: "*Are there times and locations in which the value of a customer depended greatly on the customers gender?*" Notice that he believes `Time` and `Location` play a role in predicting valued customers, but at what levels of granularity do they depend on `Gender` for this task? For example, is performing analysis using {`Month, Country`} better than {`Year, State`}?

Consider a data table **D** (e.g., the Customer table). Let **X** be the set of attributes for which no concept hierarchy has been defined (e.g., `Gender, Salary`). Let Y be the class-label attribute (e.g., `Valued_Customer`), and **Z** be the set of multilevel attributes, that is, attributes for which concept hierarchies have been defined (e.g., `Time, Location`). Let **V** be the set of attributes for which we would like to define their predictiveness. In our example, this set is {`Gender`}. The predictiveness of **V** on a subset of data can be quantified by the difference in accuracy between the model built on that subset using **X** to predict Y and the model built on that subset using $\mathbf{X} - \mathbf{V}$ (e.g., {`Salary`}) to predict Y. The intuition is that, if the difference is large, **V** must play an important role in the prediction of class label Y.

Given a set of attributes **V**, and a learning algorithm, the prediction cube at granularity $\langle l_1, \ldots, l_d \rangle$ (e.g., $\langle Year, State \rangle$) is a d-dimensional array, in which the value in each cell (e.g., [2010, Illinois]) is the predictiveness of **V** evaluated on the subset defined by the cell (e.g., the records in the Customer table with `Time` in 2010 and `Location` in Illinois).

Supporting OLAP roll-up and drill-down operations on a prediction cube is a computational challenge requiring the materialization of cell values at many different granularities. For simplicity, we can consider only full materialization. A naïve way to fully materialize a prediction cube is to exhaustively build models and evaluate them for each cell and for each granularity. This method is very expensive if the base dataset is large. An ensemble method called **Probability-Based Ensemble (PBE)** was developed as a more feasible alternative. It requires model construction for only the finest-grained cells. OLAP-style bottom-up aggregation is then used to generate the values of the coarser-grained cells.

The prediction of a predictive model can be seen as finding a class label that maximizes a scoring function. PBE was developed to approximately make the scoring function of any predictive model distributively decomposable. In our discussion of data cube measures in Section 4.5.2, we showed that distributive and algebraic measures can be computed efficiently. Therefore, if the scoring function used is distributively or algebraically decomposable, prediction cubes can also be computed with efficiency. In this way, the PBE method reduces prediction cube computation to data cube computation. For example, previous studies have shown that the naïve Bayes classifier has an algebraically decomposable scoring function, and the kernel density-based classifier has a distributively decomposable scoring function[8]. Therefore, either of these could be used to implement prediction cubes efficiently. The PBE method presents a novel approach to multidimensional data mining in cube space.

5.4.2 Multifeature Cubes: Complex Aggregation at Multiple Granularities

Data cubes facilitate the answering of analytical or mining-oriented queries as they allow the computation of aggregate data at multiple levels of granularity. Traditional data cubes are typically constructed on commonly-used dimensions (like `time`, `location`, and `product`) using *simple* measures (like `count`, `average`, and `sum`). In this section, you will learn a newer way to define data cubes called **multifeature cubes**. Multifeature cubes enable more in-depth analysis. They can compute more complex queries whose measures depend on groupings of multiple aggregates at varying levels of granularity. The queries posed can be much more elaborate and task-specific than traditional queries, as we shall illustrate in the next examples. Many complex data mining queries can be answered by multifeature cubes without significant increase in computational cost, in comparison to cube computation for simple queries with traditional data cubes.

To illustrate the idea of multifeature cubes, let's first look at an example of a query on a simple data cube.

Example 5.19 **A simple data cube query.** Let the query be "*find the total sales in 2010, broken down by item, region, and month, with subtotals for each dimension*". To answer this query, a traditional data cube is constructed that aggregates the total sales at the following eight different levels of granularity: {(*item, region, month*), (*item, region*), (*item, month*), (*month, region*), (*item*), (*month*), (*region*), ()}, where () represents `all`. This data cube is a simple in that it does not involve any dependent aggregates.

To illustrate what is mean by 'dependent aggregates', let's examine a more complex query, which can be computed with a multifeature cube.

[8]Naïve Bayes classifiers are detailed in Chapter 8. Kernel density-based classifiers, such as support vector machines, are described in Chapter 9.

Example 5.20 **A complex query involving dependent aggregates.** Suppose the query is "*grouping by all subsets of {item, region, month}, find the maximum price in 2010 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, it can be specified concisely using an extended SQL syntax as follows:

```
select      item, region, month, max(price), sum(R.sales)
from        Purchases
where       year = 2010
cube by     item, region, month: R
such that   R.price = max(price)
```

The tuples representing purchases in 2010 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 $g_1, \ldots, g_r$. 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 this query is dependent on the set of maximum price tuples for each group. In general, multifeature cubes give users the flexibility to define sophisticated, task-specific cubes on which multidimensional aggregation and OLAP-based mining can be performed.

"How can multifeature cubes be computed efficiently?" The computation of a multifeature cube depends on the types of aggregate functions used in the cube. In Chapter 4, we saw that aggregate functions can be categorized as either distributive, algebraic, or holistic. Multifeature cubes can be organized into the same categories and computed efficiently by minor extension of the cube computation methods of Section 5.2.

5.4.3 Exception-Based Discovery-Driven Exploration of Cube Space

As studied in previous sections, a data cube may have a large number of cuboids, and each cuboid may contain a large number of (aggregate) cells. With such an overwhelmingly large space, it becomes a burden for users to even just browse a cube, let alone think of exploring it thoroughly. Tools need to be developed to assist users in intelligently exploring the huge aggregated space of a data cube.

In this section, we describe a **discovery-driven approach** to exploring cube space. Precomputed measures indicating data exceptions are used to guide the user in the data analysis process, at all levels of aggregation. We hereafter refer to these measures as *exception indicators*. Intuitively, an **exception** is a data cube cell value that is significantly different from the value anticipated, based on a statistical model. The model considers variations and patterns in the measure value across *all of the dimensions* to which a cell belongs. For example, if the analysis of *item-sales* data reveals an increase in sales in December in comparison to all other months, this may seem like an exception in the time dimension. However, it is not an exception if the item dimension is considered, since there is a similar increase in sales for other items during December. The model considers exceptions hidden at all aggregated group-by's of a data cube. Visual cues such as background color are used to reflect the degree of exception of each cell, based on the precomputed exception indicators. Efficient algorithms have been proposed for cube construction, as discussed in Section 5.2. The computation of exception indicators can be overlapped with cube construction, so that the overall construction of data cubes for discovery-driven exploration is efficient.

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

- **SelfExp**: This indicates the degree of surprise of the cell value, relative to other cells at the same level of aggregation.
- **InExp**: This indicates the degree of surprise somewhere beneath the cell, if we were to drill down from it.
- **PathExp**: This indicates the degree of surprise for each drill-down path from the cell.

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

Example 5.21 **Discovery-driven exploration of a data cube.** Suppose that you would like to analyze the monthly sales at *AllElectronics* as a percentage difference from the previous month. The dimensions involved are *item*, *time*, and *region*. You begin by studying the data aggregated over all items and sales regions for each month, as shown in Figure 5.16.

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

Drill-downs can be executed along the aggregated *item* or *region* dimensions. *"Which path has more exceptions?"* you wonder. To find this out, you select a cell of interest and trigger a path exception module that colors each dimension based on the PathExp value of the cell. This value reflects the degree of surprise of that path. Suppose that the path along *item* contains more exceptions.

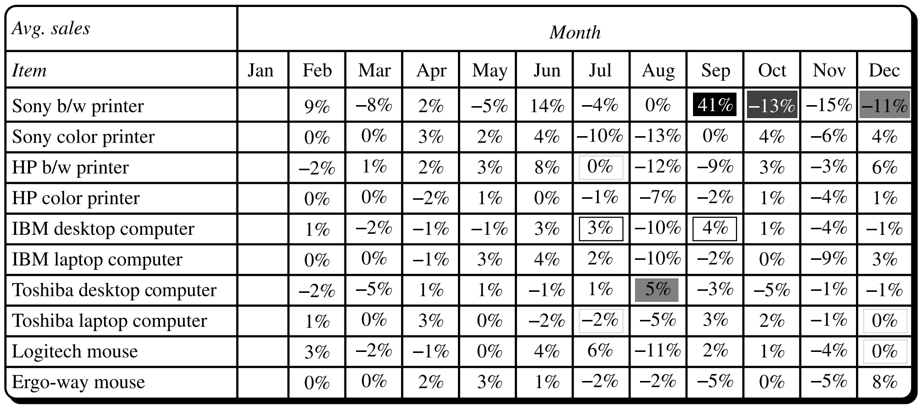
A drill-down along *item* results in the cube slice of Figure 5.17, showing the sales over time for each item. At this point, you are presented with many different sales values to analyze. By clicking on the highlight exceptions button, the visual cues are displayed, bringing focus toward the exceptions. Consider the sales difference of 41% for *"Sony b/w printers"* in September. This cell has a dark background, indicating a high SelfExp value, meaning that the cell is an exception. Consider now the sales difference of −15% for *"Sony b/w printers"* in November, and of −11% in 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 desktop 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 desktop computers"* by drilling down along *region*. The resulting sales difference by *region* is shown in Figure 5.18, 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 desktop 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 5.17. Thus, the InExp value is useful for searching for exceptions at lower-level cells of the cube.

"How are the exception values computed?" The SelfExp, InExp, and PathExp

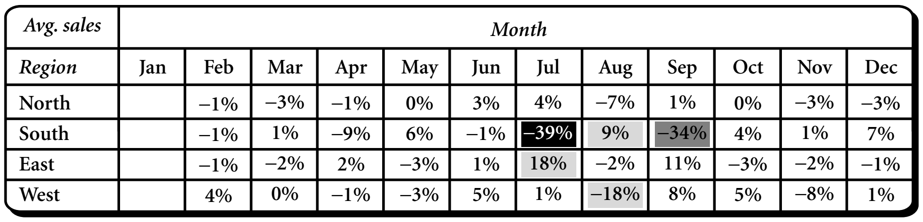
Sum of sales	***Month***											
	Jan	**Feb**	**Mar**	**Apr**	**May**	**Jun**	**Jul**	**Aug**	**Sep**	**Oct**	**Nov**	**Dec**
Total		1%	−1%	0%	1%	3%	−1%	−9%	−1%	2%	−4%	3%

Figure 5.16: Change in sales over time.



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

Figure 5.17: Change in sales for each *item-time* combination.



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 5.18: Change in sales for the item *IBM desktop computer* per region.

measures are based on a statistical method for table analysis. They take into account all of the group-by's (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 γ, γ_i^A, γ_j^B, γ_k^C, γ_{ij}^{AB}, γ_{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. 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 because 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.

5.5 Summary

- **Data cube computation and exploration** play an essential role in data warehousing and are important for flexible data mining in multidimensional space.

- A data cube consists of a **lattice of cuboids**. Each cuboid corresponds to a different degree of summarization of the given multidimensional data. **Full materialization** refers to the computation of all of the cuboids in a data cube lattice. **Partial materialization** refers to the selective computation of a subset of the cuboid cells in the lattice. Iceberg cubes and shell fragments are examples of partial materialization. An **iceberg cube** is a data cube that stores only those cube cells whose aggregate value (e.g., count) is above some minimum support threshold. For **shell fragments** of a data cube, only some cuboids involving a small number of dimensions are computed, and queries on additional combinations of the dimensions can be computed on the fly.

- There are several efficient **data cube computation methods**. In this chapter, we discussed four cube computation methods in detail: (1) **MultiWay** array aggregation for materializing full data cubes in sparse-array-based, bottom-up, shared computation; (2) **BUC** for computing iceberg cubes by exploring ordering and sorting for efficient top-down computation; (3) **Star-Cubing** for computing iceberg cubes by integrating top-down and bottom-up computation using a star-tree structure; and (4) **shell-fragment cubing**, which supports high-dimensional OLAP by precomputing only the partitioned cube shell fragments.

- **Multidimensional data mining in cube space** is the integration of knowledge discovery with multidimensional data cubes. It facilitates systematic and focussed knowledge discovery in large structured and semi-structured datasets. It will continue to endow analysts with tremendous flexibility and power at multidimensional and multigranularity exploratory analysis. This is a vast open area for researchers to build powerful and sophisticated data mining mechanisms.

- Techniques for processing advanced queries have been proposed that take advantage of cube technology. These include **sampling cubes** for multidimensional analysis on sampling data; and **ranking cubes** for efficient top-k (ranking) query processing in large relational datasets.

- This chapter highlighted three approaches to multidimensional data analysis with data cubes. **Prediction cubes** compute prediction models in multidimensional cube space. They help users identify interesting subsets of data at varying degrees of granularity for effective prediction. **Multifeature cubes** compute complex queries involving multiple dependent aggregates at multiple granularities. **Exception-based discovery-driven exploration** of cube space displays visual cues to indicate discovered data exceptions at all levels of aggregation, thereby guiding the user in the data analysis process.

5.6 Exercises

1. Assume a base cuboid of 10 dimensions contains only three base cells: (1) $(a_1, d_2, d_3, d_4, \ldots, d_9, d_{10})$, (2) $(d_1, b_2, d_3, d_4, \ldots, d_9, d_{10})$, and (3) $(d_1, d_2, c_3, d_4, \ldots, d_9, d_{10})$, where $a_1 \neq d_1$, $b_2 \neq d_2$, and $c_3 \neq d_3$. The measure of the cube is *count*.

   (a) How many *nonempty* cuboids will a full data cube contain?

   (b) How many *nonempty* aggregate (i.e., nonbase) cells will a full cube contain?

   (c) How many *nonempty* aggregate cells will an iceberg cube contain if the condition of the iceberg cube is "$count \geq 2$"?

   (d) A cell, c, is a *closed cell* if there exists no cell, d, such that d is a specialization of cell c (i.e., d is obtained by replacing a $*$ in c by a non-$*$ value) and d has the same measure value as c. A *closed cube* is a data cube consisting of only closed cells. How many closed cells are in the full cube?

2. There are several typical cube computation methods, such as *Multiway array computation* (MultiWay) [ZDN97], *BUC* (bottom-up computation) [BR99], and *Star-Cubing* [XHLW03].

   Briefly describe these three methods (i.e., use one or two lines to outline the key points), and compare their feasibility and performance under the following conditions:

   (a) Computing a dense full cube of low dimensionality (e.g., less than 8 dimensions)

   (b) Computing an iceberg cube of around 10 dimensions with a highly skewed data distribution

(c) Computing a sparse iceberg cube of high dimensionality (e.g., over 100 dimensions)

3. Suppose a data cube, C, has D dimensions, and the base cuboid contains k distinct tuples.

   (a) Present a formula to calculate the minimum number of cells that the cube, C, may contain.

   (b) Present a formula to calculate the maximum number of cells that C may contain.

   (c) Answer parts (a) and (b) above as if the count in each cube cell must be no less than a threshold, v.

   (d) Answer parts (a) and (b) above as if only closed cells are considered (with the minimum count threshold, v).

4. Suppose that a base cuboid has three dimensions, A, B, C, with the following number of cells: $|A| = 1,000,000$, $|B| = 100$, and $|C| = 1000$. Suppose that each dimension is evenly partitioned into 10 portions for *chunking*.

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

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

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

5. Often, the aggregate *count* value of many cells in a large data cuboid is zero, resulting in a huge, yet sparse, multidimensional matrix.

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

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

6. When computing a cube of high dimensionality, we encounter the inherent *curse of dimensionality* problem: there exists a huge number of subsets of combinations of dimensions.

   (a) Suppose that there are only two base cells, $\{(a_1, a_2, a_3, \ldots, a_{100}), (a_1, a_2, b_3, \ldots, b_{100})\}$, in a 100-dimensional base cuboid. Compute the number of nonempty aggregate cells. Comment on the storage space and time required to compute these cells.

(b) Suppose we are to compute an iceberg cube from the above. If the minimum support count in the iceberg condition is two, how many aggregate cells will there be in the iceberg cube? Show the cells.

(c) Introducing iceberg cubes will lessen the burden of computing trivial aggregate cells in a data cube. However, even with iceberg cubes, we could still end up having to compute a large number of trivial uninteresting cells (i.e., with small counts). Suppose that a database has 20 tuples that map to (or cover) the two following base cells in a 100-dimensional base cuboid, each with a cell count of 10: $\{(a_1, a_2, a_3, \ldots, a_{100}) : 10, (a_1, a_2, b_3, \ldots, b_{100}) : 10\}$.

i. Let the minimum support be 10. How many distinct aggregate cells will there be like the following: $\{(a_1, a_2, a_3, a_4, \ldots, a_{99}, *) : 10, \ldots, (a_1, a_2, *, a_4, \ldots, a_{99}, a_{100}) : 10, \ldots, (a_1, a_2, a_3, *, \ldots, *, *) : 10\}$?

ii. If we ignore all the aggregate cells that can be obtained by replacing some constants with $*$'s while keeping the same measure value, how many distinct cells are left? What are the cells?

7. Propose an algorithm that computes *closed iceberg cubes* efficiently.

8. Suppose that we would like to compute an iceberg cube for the dimensions, A, B, C, D, where we wish to materialize all cells that satisfy a minimum support count of at least v, and where *cardinality(A)* <*cardinality(B)* <*cardinality(C)* <*cardinality(D)*. Show the BUC processing tree (which shows the order in which the BUC algorithm explores the lattice of a data cube, starting from all) for the construction of the above iceberg cube.

9. Discuss how you might extend the *Star-Cubing* algorithm to compute iceberg cubes where the iceberg condition tests for an avg that is no bigger than some value, v.

10. A flight data warehouse for a travel agent consists of six dimensions: *traveler, departure (city), departure_time, arrival, arrival_time*, and *flight*; and two measures: *count*, and *avg_fare*, where *avg_fare* stores the concrete fare at the lowest level but average fare at other levels.

(a) Suppose the cube is fully materialized. Starting with the *base cuboid* [*traveller*, *departure*, *departure_time*, *arrival*, *arrival_time*, *flight*], what *specific OLAP operations* (e.g., roll-up *flight* to *airline*) should one perform in order to list the average fare per month for *each business traveler* who flies American Airlines (*AA*) from L.A. in the year 2009?

(b) Suppose we want to compute a data cube where the condition is that the minimum number of records is 10 and the average fare is over $500. Outline an efficient cube computation method (based on common sense about flight data distribution).

11. **(Implementation project)** There are four typical data cube computation methods: MultiWay [ZDN97], BUC [BR99], H-cubing [HPDW01], and Star-Cubing [XHLW03].

    (a) Implement any one of these cube computation algorithms and describe your implementation, experimentation, and performance. Find another student who has implemented a different algorithm on the same platform (e.g., C++ on Linux) and compare your algorithm performance with his/hers.

    Input:

    i. An n-dimensional base cuboid table (for $n < 20$), which is essentially a relational table with n attributes
    ii. An iceberg condition: *count* $(C) \geq k$ where k is a positive integer as a parameter

    Output:

    i. The set of computed cuboids that satisfy the iceberg condition, in the order of your output generation
    ii. Summary of the set of cuboids in the form of "*cuboid ID*: the number of nonempty cells", sorted in alphabetical order of cuboids, e.g., A:155, AB: 120, ABC: 22, $ABCD$: 4, $ABCE$: 6, ABD: 36, where the number after ":" represents the number of nonempty cells. (This is used to quickly check the correctness of your results.)

    (b) Based on your implementation, discuss the following:

    i. What challenging computation problems are encountered as the number of dimensions grows large?
    ii. How can iceberg cubing solve the problems of part (a) for some data sets (and characterize such data sets)?
    iii. Give one simple example to show that sometimes iceberg cubes cannot provide a good solution.

    (c) Instead of computing a data cube of high dimensionality, we may choose to materialize the cuboids that have only a small number of dimension combinations. For example, for a 30-dimensional data cube, we may only compute the 5-dimensional cuboids for every possible 5-dimensional combination. The resulting cuboids form a *shell cube*. Discuss how easy or hard it is to modify your cube computation algorithm to facilitate such computation.

12. The *sampling cube* was proposed for multidimensional analysis of sampling data (*e.g.*, survey data). In many real applications, sampling data can be of high dimensionality, *e.g.*, it is not unusual to have more than 50 dimensions in a survey dataset.

    (a) How can we construct an efficient and scalable high-dimensional sampling cube in large sampling datasets?

(b) Design an efficient incremental update algorithm for such a high-dimensional sampling cube, and

(c) Discuss how to support quality drill-down although some low-level cells may contain empty or too few data for reliable analysis.

13. The *ranking cube* was proposed for efficient computation of top-k (ranking) queries in relational databases. Recently, researchers have proposed another kind of queries, called *skyline queries*. A *skyline query* returns all the objects p_i such that p_i is not dominated by any other object p_j, where dominance is defined as follows. Let the value of p_i on dimension d be $v(p_i, d)$. We say p_i is dominated by p_j if and only if for each preference dimension d, $v(p_j, d) \leq v(p_i, d)$, and there is at least one d where the equality does not hold.

    (a) Can you design a ranking cube so that skyline queries can be processed efficiently?

    (b) Skyline is sometimes too strict to be desirable to some users. One may generalize the concept of skyline into *generalized skyline* as below: *Given a d-dimensional database and a query q, the* ***generalized skyline*** *is the set of the following objects: (1) the skyline objects, and (2) the non-skyline objects that are* ϵ*-neighbors of a skyline object, where r is an* ϵ*-neighbor of an object p if the distance between p and r is no more than* ϵ. Can you design a ranking cube to process generalized skyline queries efficiently?

14. The ranking cube was designed to support top-k (ranking) queries in relational database systems. However, ranking queries are also posed to data warehouses, where ranking is on multidimensional aggregates instead of on measures of base facts. For example, consider a product manager who is analyzing a sales database that stores the nationwide sales history, organized by location and time. In order to make investment decisions, the manager may pose the following queries: "*What are the top-10 (state, year) cells having the largest total product sales?*" and he may further drill-down and ask "*What are the top-10 (city, month) cells?*" Suppose the system can perform such partial materialization to derive two types of materialized cuboids: a *guiding cuboid* and a *supporting cuboid*, where the former contains a number of guiding cells that provide concise, high-level data statistics to guide the ranking query processing, whereas the latter provides inverted indices for efficient online aggregation.

    (a) Can you derive an efficient method for computing such aggregate ranking cubes?

    (b) Can you extend your framework to handle more advanced measures? One such example could be as follows. Consider an organization donation database, where donators are grouped by "*age*", "*income*", and other attributes. Interesting questions include: "*Which age and* *income groups have made the top-k average amount of donation (per-donor)?*" and "*Which income group of donators has the largest standard deviation in the donation amount?*"

15. *Prediction cube* is a good example of multidimensional data mining in cube space.

    (a) Propose an efficient algorithm that computes prediction cubes in a given multidimensional database; and

    (b) For what kind of classification models can your algorithm be applied. Explain.

16. *Multifeature cubes* allow us to construct interesting data cubes based on rather sophisticated query conditions. Can you construct the following multifeature cube by translating the following user requests into queries using the form introduced in this textbook?

    (a) Construct smart shopper cube where a shopper is smart if at least 10% of the goods she buys in each shopping are on sale; and

    (b) Construct a datacube for best deal products where the best deal products are those products whose price is the lowest for this product in the month.

17. *Discovery-driven cube exploration* is a desirable way to mark interesting points among a large number of cells in a data cube. Individual users may have different views on whether a point should be considered interesting enough to be marked. Suppose one would like to mark those objects whose absolute value of Z score is over 2 in every row and column in a d-dimensional plane.

    (a) Can you derive an efficient computation method to identify such points during the data cube computation?

    (b) Suppose a partially materialized cube has $(d-1)$-dimensional and $(d+1)$-dimensional cuboids materialized but not the d-dimensional one. Can you derive an efficient method to mark those $(d-1)$-dimensional cells whose d-dimensional children contain such marked point(s)?

5.7 Bibliographic Notes

Efficient computation of multidimensional aggregates in data cubes has been studied by many researchers. Gray, Chaudhuri, Bosworth, et al. [GCB$^+$97] proposed *cube-by* as a relational aggregation operator generalizing group-by, crosstabs, and subtotals, and categorized data cube measures into three categories: *distributive*, *algebraic* and *holistic*. Harinarayan, Rajaraman, and Ullman [HRU96] proposed a greedy algorithm for the partial materialization of cuboids in the computation of a data cube. Sarawagi and Stonebraker

[SS94] developed a chunk-based computation technique for the efficient organization of large multidimensional arrays. Agarwal, Agrawal, Deshpande, et al. [AAD$^+$96] proposed several guidelines for efficient computation of multidimensional aggregates for ROLAP servers. The chunk-based MultiWay array aggregation method for data cube computation in MOLAP was proposed in Zhao, Deshpande, and Naughton [ZDN97]. Ross and Srivastava [RS97] developed a method for computing sparse data cubes. Iceberg queries are first described in Fang, Shivakumar, Garcia-Molina, et al. [FSGM$^+$98]. BUC, a scalable method that computes iceberg cubes from the apex cuboid, downwards, was introduced by Beyer and Ramakrishnan [BR99]. Han, Pei, Dong, Wang [HPDW01] introduced an H-cubing method for computing iceberg cubes with complex measures using an H-tree structure. The Star-Cubing method for computing iceberg cubes with a dynamic star-tree structure was introduced by Xin, Han, Li, and Wah [XHLW03]. MMCubing, an efficient iceberg cube computation method that factorizes the lattice space was developed by Shao, Han, and Xin [SHX04]. The shell-fragment-based cubing approach for efficient high-dimensional OLAP was proposed by Li, Han, and Gonzalez [LHG04].

Aside from computing iceberg cubes, another way to reduce data cube computation is to materialize condensed, dwarf, or quotient cubes, which are variants of closed cubes. Wang, Feng, Lu, and Yu proposed computing a reduced data cube, called a *condensed cube* [WLFY02]. Sismanis, Deligiannakis, Roussopoulos, and Kotids proposed computing a compressed data cube, called a *dwarf cube*. Lakeshmanan, Pei, and Han proposed a *quotient cube* structure to summarize the semantics of a data cube [LPH02], which has been further extended to a *qc-tree structure* by Lakshmanan, Pei, and Zhao [LPZ03]. An *aggregation-based* approach, called C-cubing (i.e., *Closed-Cubing*), has been developed by Xin, Han, Shao, and Liu [XHSL06], which performs efficient closed cube computation by taking advantage of a new algebraic measure *closedness*.

There are also various studies on the computation of compressed data cubes by approximation, such as *quasi-cubes* by Barbara and Sullivan [BS97], *wavelet cubes* by Vitter, Wang, and Iyer [VWI98], *compressed cubes* for query approximation on continuous dimensions by Shanmugasundaram, Fayyad, and Bradley [SFB99], using log-linear models to compress data cubes by Barbara and Wu [BW00], and OLAP over uncertain and imprecise data by Burdick, Deshpande, Jayram, et al. [BDJ$^+$05].

For works regarding the selection of materialized cuboids for efficient OLAP query processing, see Chaudhuri and Dayal [CD97], Harinarayan, Rajaraman, and Ullman [HRU96], and Sristava, Dar, Jagadish, and Levy [SDJL96], Gupta [Gup97], Baralis, Paraboschi, and Teniente [BPT97], and Shukla, Deshpande, and Naughton [SDN98]. Methods for cube size estimation can be found in Deshpande, Naughton, Ramasamy, et al. [DNR$^+$97], Ross and Srivastava [RS97], and Beyer and Ramakrishnan [BR99]. Agrawal, Gupta, and Sarawagi [AGS97] proposed operations for modeling multidimensional databases.

Data cube modeling and computation have been extended well beyond relational data. Computation of *stream cubes* for multidimensional stream data analysis has been studied by Chen, Dong, Han, et al. [CDH$^+$02]. Efficient computation of *spatial data cubes* was examined by Stefanovic, Han, and Koperski [SHK00], efficient OLAP in spatial data warehouses was studied by Papadias, Kalnis, Zhang, and Tao [PKZT01], and a map cube for visualizing spatial data warehouses was proposed by Shekhar, Lu, Tan, et al. [SLT$^+$01]. A multimedia data cube was constructed in MultiMediaMiner by Zaiane, Han, Li, et al. [ZHL$^+$98]. For analysis of multidimensional text databases, *TextCube*, based on the vector space model, was proposed by Lin, Ding, Han, et al. [LDH$^+$08], and *TopicCube*, based on a topic modeling approach, was proposed by Zhang, Zhai and Han [ZZH09]. *RFID cube* and *FlowCube* for analyzing RFID data were proposed by Gonzalez, Han, Li, et al. [GHLK06, GHL06]. *Sampling cube* was introduced for analyzing sampling data by Li, Han, Yin, et al. [LHY$^+$08]. *Ranking cube* was proposed by Xin, Han, Cheng, and Li [XHCL06] for efficient processing of ranking (top-k) queries in databases. This methodology has also been extended to supporting ranking aggregate queries in partially materialized data cubes, called *ARCube*, by Wu, Xin, and Han [WXH08], and for supporting promotion query analysis in multi-dimensional space, called *PromoCube*, by Wu, Xin, Mei, and Han [WXMH09].

The discovery-driven exploration of OLAP data cubes was proposed by Sarawagi, Agrawal, and Megiddo [SAM98]. Further studies on integration of OLAP with data mining capabilities for intelligent exploration of multidimensional OLAP data were done by Sarawagi and Sathe [SS01]. The construction of multifeature data cubes is described by Ross, Srivastava, and Chatziantoniou [RSC98]. Methods for answering queries quickly by on-line aggregation are described by Hellerstein, Haas, and Wang [HHW97] and Hellerstein, Avnur, Chou, et al. [HAC$^+$99]. A cube-gradient analysis problem, called *cubegrade*, was first proposed by Imielinski, Khachiyan, and Abdulghani [IKA02]. An efficient method for multidimensional constrained gradient analysis in data cubes was studied by Dong, Han, Lam, et al. [DHL$^+$01].

Mining cubespace, or integration of knowledge discovery and OLAP cubes, has been studied by many researchers. The concept of online analytical mining (OLAM), or OLAP mining, was introduced by Han [Han98]. Chen, Dong, Han, et al. developed a *regression cube* for regression-based multidimensional analysis of time-series data [CDH$^+$02, CDH$^+$06]. Fagin, Guha, R. Kumar et al. [FGK$^+$05] studied data mining in *multi-structured databases*. *Prediction cubes* that integrate prediction models with data cubes for discovery of data space to facilitate certain prediction was proposed by B.-C. Chen, L. Chen, Lin, and Ramakrishnan [CCLR05]. Using data-mining models as building blocks in a multi-step mining process and using cube space intuitively defines the space of interest for predicting global aggregates from local regions was studied by Chen, Ramakrishnan, Shavlik, and Tamma [CRST06]. Ramakrishnan and Chen [RC07] presented an organized picture on exploratory mining in cube space.

Bibliography

[AAD+96] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96)*, pages 506–521, Bombay, India, Sept. 1996.

[AGS97] R. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. In *Proc. 1997 Int. Conf. Data Engineering (ICDE'97)*, pages 232–243, Birmingham, England, April 1997.

[BDJ+05] D. Burdick, P. Deshpande, T. S. Jayram, R. Ramakrishnan, and S. Vaithyanathan. OLAP over uncertain and imprecise data. In *Proc. 2005 Int. Conf. Very Large Data Bases (VLDB'05)*, pages 970–981, Trondheim, Norway, Aug. 2005.

[BPT97] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multidimensional database. In *Proc. 1997 Int. Conf. Very Large Data Bases (VLDB'97)*, pages 98–12, Athens, Greece, Aug. 1997.

[BR99] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. 1999 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'99)*, pages 359–370, Philadelphia, PA, June 1999.

[BS97] D. Barbara and M. Sullivan. Quasi-cubes: Exploiting approximation in multidimensional databases. *SIGMOD Record*, 26:12–17, 1997.

[BW00] D. Barbará and X. Wu. Using loglinear models to compress datacube. In *Proc. 1st Int. Conf. Web-Age Information Management (WAIM'2000)*, pages 311–322, Shanghai, China, 2000.

[CCLR05] B.-C. Chen, L. Chen, Y. Lin, and R. Ramakrishnan. Prediction cubes. In *Proc. 2005 Int. Conf. Very Large Data Bases (VLDB'05)*, pages 982–993, Trondheim, Norway, Aug. 2005.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26:65–74, 1997.

[CDH$^+$02] Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang. Multi-dimensional regression analysis of time-series data streams. In *Proc. 2002 Int. Conf. Very Large Data Bases (VLDB'02)*, pages 323–334, Hong Kong, China, Aug. 2002.

[CDH$^+$06] Y. Chen, G. Dong, J. Han, J. Pei, B. W. Wah, and J. Wang. Regression cubes with lossless compression and aggregation. *IEEE Trans. Knowledge and Data Engineering*, 18:1585–1599, 2006.

[CRST06] B.-C. Chen, R. Ramakrishnan, J. W. Shavlik, and P. Tamma. Bellwether analysis: Predicting global aggregates from local regions. In *Proc. 2006 Int. Conf. Very Large Data Bases (VLDB'06)*, pages 655–666, Seoul, Korea, Sept. 2006.

[DHL$^+$01] G. Dong, J. Han, J. Lam, J. Pei, and K. Wang. Mining multi-dimensional constrained gradients in data cubes. In *Proc. 2001 Int. Conf. on Very Large Data Bases (VLDB'01)*, pages 321–330, Rome, Italy, Sept. 2001.

[DNR$^+$97] P. Deshpande, J. Naughton, K. Ramasamy, A. Shukla, K. Tufte, and Y. Zhao. Cubing algorithms, storage estimation, and storage and processing alternatives for OLAP. *Bull. Technical Committee on Data Engineering*, 20:3–11, 1997.

[FGK$^+$05] R. Fagin, R. V. Guha, R. Kumar, J. Novak, D. Sivakumar, and A. Tomkins. Multi-structural databases. In *Proc. 2005 ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS'05)*, pages 184–195, Baltimore, MD, June 2005.

[FSGM$^+$98] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 299–310, New York, NY, Aug. 1998.

[GCB$^+$97] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-totals. *Data Mining and Knowledge Discovery*, 1:29–54, 1997.

[GHL06] H. Gonzalez, J. Han, and X. Li. Flowcube: Constructuing RFID flowcubes for multi-dimensional analysis of commodity flows. In *Proc. 2006 Int. Conf. Very Large Data Bases (VLDB'06)*, pages 834–845, Seoul, Korea, Sept. 2006.

[GHLK06] H. Gonzalez, J. Han, X. Li, and D. Klabjan. Warehousing and analysis of massive RFID data sets. In *Proc. 2006 Int. Conf. Data Engineering (ICDE'06)*, page 83, Atlanta, Georgia, April 2006.

[Gup97] H. Gupta. Selection of views to materialize in a data warehouse. In *Proc. 7th Int. Conf. Database Theory (ICDT'97)*, pages 98–112, Delphi, Greece, Jan. 1997.

[HAC+99] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive data analysis: The control project. *IEEE Computer*, 32:51–59, July 1999.

[Han98] J. Han. Towards on-line analytical mining in large databases. *SIGMOD Record*, 27:97–107, 1998.

[HHW97] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'97)*, pages 171–182, Tucson, AZ, May 1997.

[HPDW01] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. In *Proc. 2001 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'01)*, pages 1–12, Santa Barbara, CA, May 2001.

[HRU96] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'96)*, pages 205–216, Montreal, Canada, June 1996.

[IKA02] T. Imielinski, L. Khachiyan, and A. Abdulghani. Cubegrades: Generalizing association rules. *Data Mining and Knowledge Discovery*, 6:219–258, 2002.

[LDH+08] C. X. Lin, B. Ding, J. Han, F. Zhu, and B. Zhao. Text Cube: Computing IR measures for multidimensional text database analysis. In *Proc. 2008 Int. Conf. on Data Mining (ICDM'08)*, Pisa, Italy, Dec. 2008.

[LHG04] X. Li, J. Han, and H. Gonzalez. High-dimensional OLAP: A minimal cubing approach. In *Proc. 2004 Int. Conf. Very Large Data Bases (VLDB'04)*, pages 528–539, Toronto, Canada, Aug. 2004.

[LHY+08] X. Li, J. Han, Z. Yin, J.-G. Lee, and Y. Sun. Sampling Cube: A framework for statistical OLAP over sampling data. In *Proc. 2008 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'08)*, Vancouver, BC, Canada, June 2008.

[LPH02] L. V. S. Lakshmanan, J. Pei, and J. Han. Quotient cube: How to summarize the semantics of a data cube. In *Proc. 2002 Int. Conf. on Very Large Data Bases (VLDB'02)*, pages 778–789, Hong Kong, China, Aug. 2002.

[LPZ03] L. V. S. Lakshmanan, J. Pei, and Y. Zhao. QC-Trees: An efficient summary structure for semantic OLAP. In *Proc. 2003 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'03)*, pages 64–75, San Diego, CA, June 2003.

[PKZT01] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *Proc. 2001 Int. Symp. Spatial and Temporal Databases (SSTD'01)*, pages 443–459, Redondo Beach, CA, July 2001.

[RC07] R. Ramakrishnan and B.-C. Chen. Exploratory mining in cube space. *Data Mining and Knowledge Discovery*, 15:29–54, 2007.

[RS97] K. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proc. 1997 Int. Conf. Very Large Data Bases (VLDB'97)*, pages 116–125, Athens, Greece, Aug. 1997.

[RSC98] K. A. Ross, D. Srivastava, and D. Chatziantoniou. Complex aggregation at multiple granularities. In *Proc. Int. Conf. of Extending Database Technology (EDBT'98)*, pages 263–277, Valencia, Spain, Mar. 1998.

[SAM98] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of OLAP data cubes. In *Proc. Int. Conf. of Extending Database Technology (EDBT'98)*, pages 168–182, Valencia, Spain, Mar. 1998.

[SDJL96] D. Sristava, S. Dar, H. V. Jagadish, and A. V. Levy. Answering queries with aggregation using views. In *Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96)*, pages 318–329, Bombay, India, Sept. 1996.

[SDN98] A. Shukla, P. M. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 488–499, New York, NY, Aug. 1998.

[SFB99] J. Shanmugasundaram, U. M. Fayyad, and P. S. Bradley. Compressed data cubes for OLAP aggregate query approximation on continuous dimensions. In *Proc. 1999 Int. Conf. Knowledge Discovery and Data Mining (KDD'99)*, pages 223–232, San Diego, CA, Aug. 1999.

[SHK00] N. Stefanovic, J. Han, and K. Koperski. Object-based selective materialization for efficient implementation of spatial data cubes. *IEEE Trans. Knowledge and Data Engineering*, 12:938–958, 2000.

[SHX04] Z. Shao, J. Han, and D. Xin. MM-Cubing: Computing iceberg cubes by factorizing the lattice space. In *Proc. 2004 Int. Conf.*

on Scientific and Statistical Database Management (SSDBM'04), pages 213–222, Santorini Island, Greece, June 2004.

[SLT+01] S. Shekhar, C.-T. Lu, X. Tan, S. Chawla, and R. R. Vatsavai. Map cube: A visualization tool for spatial data warehouses. In H. J. Miller and J. Han, editors, *eographic Data Mining and Knowledge Discovery*, pages 73–108. Taylor and Francis, 2001.

[SS94] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proc. 1994 Int. Conf. Data Engineering (ICDE'94)*, pages 328–336, Houston, TX, Feb. 1994.

[SS01] G. Sathe and S. Sarawagi. Intelligent rollups in multidimensional OLAP data. In *Proc. 2001 Int. Conf. Very Large Data Bases (VLDB'01)*, pages 531–540, Rome, Italy, Sept. 2001.

[VWI98] J. S. Vitter, M. Wang, and B. R. Iyer. Data cube approximation and histograms via wavelets. In *Proc. 1998 Int. Conf. Information and Knowledge Management (CIKM'98)*, pages 96–104, Washington, DC, Nov. 1998.

[WLFY02] W. Wang, H. Lu, J. Feng, and J. X. Yu. Condensed cube: An effective approach to reducing data cube size. In *Proc. 2002 Int. Conf. Data Engineering (ICDE'02)*, pages 155–165, San Fransisco, CA, April 2002.

[WXH08] T. Wu, D. Xin, and J. Han. ARCube: Supporting ranking aggregate queries in partially materialized data cubes. In *Proc. 2008 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'08)*, Vancouver, BC, Canada, June 2008.

[WXMH09] T. Wu, D. Xin, Q. Mei, and J. Han. Promotion analysis in multi-dimensional space. In *Proc. 2009 Int. Conf. on Very Large Data Bases (VLDB'09)*, Lyon, France, Aug. 2009.

[XHCL06] D. Xin, J. Han, H. Cheng, and X. Li. Answering top-k queries with multi-dimensional selections: The ranking cube approach. In *Proc. 2006 Int. Conf. on Very Large Data Bases (VLDB'06)*, Seoul, Korea, Sept. 2006.

[XHLW03] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *Proc. 2003 Int. Conf. Very Large Data Bases (VLDB'03)*, pages 476–487, Berlin, Germany, Sept. 2003.

[XHSL06] D. Xin, J. Han, Z. Shao, and H. Liu. C-cubing: Efficient computation of closed cubes by aggregation-based checking. In *Proc. 2006 Int. Conf. Data Engineering (ICDE'06)*, Atlanta, Georgia, April 2006.

[ZDN97] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'97)*, pages 159–170, Tucson, AZ, May 1997.

[ZHL$^+$98] O. R. Zaïane, J. Han, Z. N. Li, J. Y. Chiang, and S. Chee. MultiMedia-Miner: A system prototype for multimedia data mining. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98)*, pages 581–583, Seattle, WA, June 1998.

[ZZH09] D. Zhang, C. Zhai, and J. Han. Topic cube: Topic modeling for OLAP on multidimensional text databases. In *Proc. 2009 SIAM Int. Conf. on Data Mining (SDM'09)*, Sparks, NV, April 2009.