Computation of OLAP Cubes

Amin A. Abdulghani
Quantiva, USA

INTRODUCTION

The focus of Online Analytical Processing (OLAP) is to provide a platform for analyzing data (e.g., sales data) with multiple dimensions (e.g., product, location, time) and multiple measures (e.g., total sales or total cost). OLAP operations then allow viewing of this data from a number of perspectives. For analysis, the object or data structure of primary interest in OLAP is a cube.

BACKGROUND

An *n-dimensional cube* is defined as a group of *k*-dimensional (k<=n) cuboids arranged by the dimensions of the data. A *cell* represents an association of a measure *m* (e.g., total sales) with a member of every dimension (e.g., product="toys", location="NJ", year="2003"). By definition, a cell in an *n*-dimensional cube can have fewer dimensions than *n*. The dimensions not present in the cell are aggregated over all possible members. For example, you can have a two-dimensional (2-D) cell, C1(product="toys", year="2003"). Here, the implicit value for the dimension location is '*', and the measure *m* (e.g., total sales) is aggregated over all locations. A *cuboid* is a group-by of a subset of dimensions, obtained by aggregating all tuples on these dimensions. In an *n*-dimensional cube, a cuboid is a base cuboid if it has exactly *n* dimensions. If the number of dimensions is fewer than *n,* then it is an aggregate cuboid. Any of the standard aggregate functions such as count, total, average, minimum, or maximum can be used for aggregating. Figure 1 shows an example of a three-dimensional (3-D) cube, and Figure 2 shows an aggregate 2-D cuboid.

In theory, no special operators or SQL extensions are required to take a set of records in the database and generate all the cells for the cube. Rather, the SQL group-by and union operators can be used in conjunction with *d* sorts of the dataset to produce all cuboids. However, such an approach would be very inefficient, given the obvious interrelationships between the various group-bys produced.

MAIN THRUST

I now describe the essence of the major methods for the computation of OLAP cubes.

Figure 1. A 3-D cube that consists of 1-D, 2-D, and 3-D cuboids

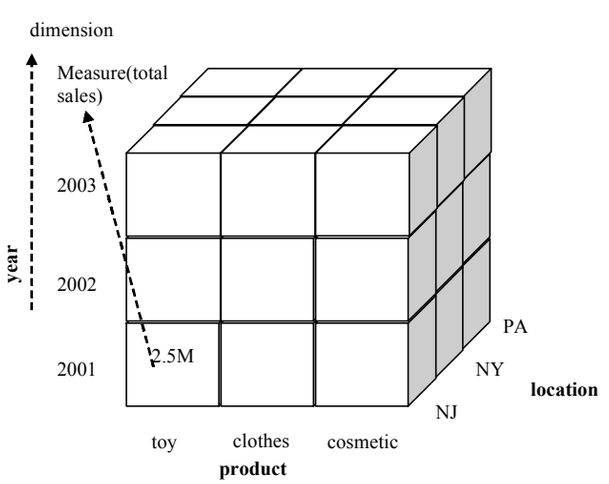
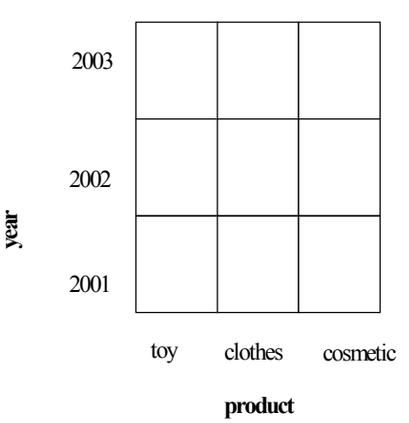


Figure 2. An example 2-D cuboid on (product, year) for the 3-D cube in Figure 1 (location=''); total sales needs to be aggregated (e.g., SUM)*



Top-Down Computation

In a seminal paper, Gray, Bosworth, Layman, and Pirahesh (1996) proposed the data cube operator as a means of simplifying the process of data cube construction. The algorithm presented forms the basis of the top-down approach. In the approach, the aggregation functions are categorized into three classes:

- **Distributive:** An aggregate function F is called distributive if there exists a function g such that the value of F for an *n*-dimensional cell can be computed by applying g to the value of F in an (*n* + 1)–dimensional cell. Examples of such functions include SUM and COUNT. For example, the COUNT() of an *n*-dimensional cell can be computed by applying SUM to the value of COUNT() in an (*n*+1)–dimensional cell.
- **Algebraic:** An aggregate function F is algebraic if F of an *n*-dimensional cell can be computed by using a constant number of aggregates of the (*n* + 1)–dimensional cell. An example is the AVERAGE() function. The AVERAGE() of an *n*-dimensional cell can be computed by taking the sum and count of the (*n*+1)–dimensional cell and then dividing the SUM by the COUNT to produce the global average.
- **Holistic:** An aggregate function F is called holistic if the value of F for an *n*-dimensional cell cannot be computed from a constant number of aggregates of the (*n*+1)–dimensional cell. Median and mode are examples of holistic functions.

The top-down cube computation works with distributive or algebraic functions. These functions have the property that more detailed aggregates (i.e., more dimensions) can be used to compute less detailed aggregates. This property induces a partial-ordering (i.e., a *lattice*) on all the group-bys of the cube. A group-by is called a child of some parent group-by if the parent can be used to compute the child (and no intermediate group-bys exist between the parent and child). Figure 3 depicts a sample lattice where A, B, C, and D are dimensions, nodes represent group-bys, and the edges show the parent-child relationship.

The basic idea for top-down cube construction is to start by computing the base cuboid (group-by for which no cube dimensions are aggregated). A single pass is made over the data, a record is examined, and the appropriate base cell is incremented. The remaining group-bys are computed by aggregating over already computed finer grade group-by. If a group-by can be computed from one or more possible parent group-bys, then the algorithm uses the parent smallest in size. For example, for computing the cube ABCD, the algorithm starts out by computing the cuboids for ABCD. Then, using ABCD, it computes the cuboids for ABC, ABD, and BCD. The algorithm then repeats itself by computing the 2-D cuboids, AB, BC, AD, and BD. Note that the 2-D–cuboids can be computed from multiple parents. For example, AB can be computed from ABC or ABD. The algorithm selects the smaller group-by (the group-by with the fewest number of cells). An example top-down cube computation is shown in Figure 4.

Variants of this approach optimize on additional costs. The best-known methods are the PipeSort and PipeHash (Agarwal, Agrawal, Deshpande, Gupta, Naughton, Ramakrishnan, et al., 1996). The basic idea of both algorithms is that a minimum spanning tree should be generated from the original lattice such that the cost of traversing edges will be minimized. The optimizations for the costs that these algorithms include are as follows:

- **Cache-results:** This optimization aims at ensuring that the result of a group-by is cached (in memory) so other group-bys can use it in the future.
- **Amortize-scans:** This optimization amortizes the cost of a disk read by computing the maximum possible number of group-bys together in memory.
- **Share-sorts:** For a sort-based algorithm, this aims at sharing sorting cost across multiple group-bys.

Figure 3. Cube lattice

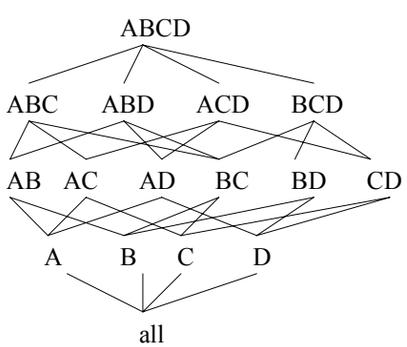


Figure 4. Top-down cube computation