Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals

Jim Gray
Adam Bosworth
Andrew Layman
Hamid Pirahesh[1]

5 February 1995, Revised 18 October 1995

Technical Report
MSR-TR-95-22

Microsoft Research
Advanced Technology Division
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

[1] IBM Research, 500 Harry Road, San Jose, CA. 95120

Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals

Jim Gray	Microsoft	Gray@Microsoft.com
Adam Bosworth	Microsoft	AdamB@Microsoft.com
Andrew Layman	Microsoft	AndrewL@Microsoft.com
Hamid Pirahesh	IBM	Pirahesh@Almaden.IBM.com

Microsoft Technical report MSR-TR-95-22
5 February 1995, Revised 18 November 1995

Abstract: Data analysis applications typically aggregate data across many dimensions looking for unusual patterns. The SQL aggregate functions and the `GROUP BY` operator produce zero-dimensional or one-dimensional answers. Applications need the *N*-dimensional generalization of these operators. This paper defines that operator, called the *data cube* or simply *cube*. The cube operator generalizes the histogram, cross-tabulation, roll-up, drill-down, and sub-total constructs found in most report writers. The cube treats each of the *N* aggregation attributes as a dimension of *N*-space. The aggregate of a particular set of attribute values is a point in this space. The set of points forms an *N*-dimensional cube. Super-aggregates are computed by aggregating the *N*-cube to lower dimensional spaces. Aggregation points are represented by an "infinite value", `ALL`. For example, the point (`ALL`,`ALL`,`ALL`,...,`ALL`, sum(*)) would represent the global sum of all items. Each `ALL` value actually represents the set of values contributing to that aggregation.

1. Introduction

Data analysis applications look for unusual patterns in data. They summarize data values, extract statistical information, and then contrast one category with another. There are two steps to such data analysis:

- **extracting** the aggregated data from the database into a file or table, and
- **visualizing** the results in a graphical way.

Visualization tools use space, color, and time (motion) to display trends, clusters, and differences. The most exciting work in data analysis focuses on presenting new graphical metaphors that allow people to discover and quickly recognize data trends and anomalies. Many tools represent the dataset as an *N*-dimensional space. Two and three-dimensional sub-slabs of this space are rendered as 2D or 3D objects. Color and time add two more dimensions to the display giving the potential of a 5D display.

How do traditional relational databases fit into this picture? How can flat files (SQL tables) possibly model an *N*-dimensional problem? Relational systems model *N*-dimensional data as *N*-attribute domains. For example, 4-dimensional earth-temperature data is typically represented by a `Weather` table shown below. The first four columns represent the four dimensions: x, y, z, t. Additional columns represent measurements at the 4D points such as temperature, pressure, humidity, and wind velocity. Often these measured values are aggregates over time (the hour) or space (a measurement area).

Table 1: `Weather`

Time (UCT)	Latitude	Longitude	Altitude (m)	Temp (c)	Pres (mb)
27/11/94:1500	37:58:33N	122:45:28W	102	21	1009
27/11/94:1500	34:16:18N	27:05:55W	10	23	1024

The SQL standard provides five functions to aggregate the values in a table: `COUNT()`, `SUM()`, `MIN()`, `MAX()`, and `AVG()`. For example, the average of all measured temperatures is expressed as:

```
SELECT  AVG(Temp)
FROM    Weather;
```

In addition, SQL allows aggregation over distinct values. The following query counts the distinct number of reporting times in the `Weather` table:

```
SELECT  COUNT(DISTINCT Time)
FROM    Weather;
```

Many SQL systems add statistical functions (median, standard deviation, variance, etc.), physical functions (center of mass, angular momentum, etc.), financial analysis (volatility, Alpha, Beta, etc.), and other domain-specific functions.

Some systems allow users to add new aggregation functions. The Illustra system, for example, allows users to add aggregate functions by adding a program with the following three callbacks to the database system [Illustra]:

`Init` `(&handle)`: Allocates the handle and initializes the aggregate computation.

`Iter` `(&handle, value)`: Aggregates the next value into the current aggregate.

`value =` **`Final`**`(&handle)`: Computes and returns the resulting aggregate by using data saved in the handle. This invocation deallocates the handle.

Consider implementing the `Average()` function. The `handle` stores the `count` and the `sum` initialized to `zero`.

When passed a new non-null value, `Iter()`increments the `count` by one and the `sum` by the value. The `Final()` call deallocates the `handle` and returns the `sum` divided by the `count`.

Aggregate functions return a single value. Using the `GROUP BY` construct, SQL can also create a table of many aggregate values indexed by a set of attributes. For example, The following query reports the average temperature for each reporting time and altitude:

```
SELECT   Time, Altitude, AVG(Temp)
FROM     Weather
GROUP BY Time, Altitude;
```

`GROUP BY` is an unusual relational operator: It partitions the relation into disjoint tuple sets and then aggregates over each set as illustrated in Figure 1.

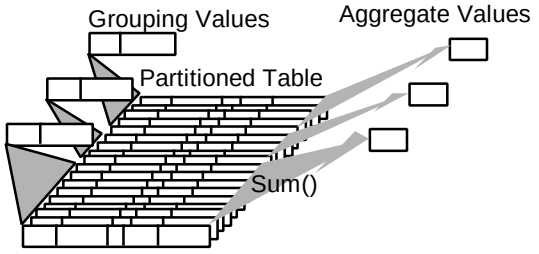

Figure 1: The `GROUP BY` relational operator partitions a table into groups. Each group is then aggregated by a function. The aggregation function summarizes some column of groups returning a value for each group.

Red Brick systems added some interesting aggregate functions that enhance the `GROUP BY` mechanism [Red Brick]:

`Rank``(expression)`: returns the expression's rank in the set of all values of this domain of the table. If there are *N* values in the column, and this is the highest value, the rank is *N*, if it is the lowest value the rank is 1.

`N_tile``(expression, n)`: The range of the expression (over all the input values of the table) is computed and divided into `n` value ranges of approximately equal population. The function returns the number of the range holding the value of the expression. The if your bank account was among the largest 10% then `rank(account.balance,10)` would return the value 10. In fact, Red Brick provides just `N_tile(expression,3)`.

`Ratio_To_Total``(expression)`: Sums all the expressions and then divides the expression by the total sum.

To give an example:

```
SELECT  Percentile,MIN(Temp),MAX(Temp)
FROM    Weather
GROUP BY N_tile(Temp,10) as Percentile
HAVING  Percentile = 5;
```

returns one row giving the minimum and maximum temperatures of the middle 10% of all temperatures. As mentioned later, allowing function values in the GROUP BY is not yet allowed by the SQL standard.

Red Brick also offers three **cumulative aggregates** that operate on ordered tables.

`Cumulative``(expression)`: Sums all values so far in an ordered list.

`Running_Sum``(expression,n)`: Sums the most recent `n` values in an ordered list. The initial `n-1` values are `NULL`.

`Running_Average``(expression,n)`: Averages the most recent `n` values in an ordered list. The initial `n-1` values are `NULL`.

Syntax is provided to optionally reset these aggregate functions each time an a grouping value changes in an ordered selection.

2. Problems With `GROUP BY`:

SQL's aggregation functions are widely used. In the spirit of aggregating data, the following table shows how frequently the database and transaction processing benchmarks use aggregation and `GROUP BY`. Surprisingly, aggregates also appear in the online-transaction processing TPC-C query set. Paradoxically, the TPC-A and TPC-B benchmark transactions spend most of their energies maintaining aggregates dynamically: they maintain the summary bank account balance, teller cash-drawer balance, and branch balance. All these can be computed as aggregates from the history table [TPC].

Table 2: SQL Aggregates in Standard Benchmarks

Benchmark	Queries	Aggregates	GROUP BYs
TPC-A, B	1	0	0
TPC-C	18	4	0
TPC-D	16	27	15
Wisconsin	18	3	2
AS^{3}AP	23	20	2
SetQuery	7	5	1

The TPC-D query set has one 6D `GROUP BY` and three 3D `GROUP BY`s. 1D and 2D `GROUP BY`s are the most common.

Certain forms of data analysis are difficult if not impossible with the SQL constructs. As explained here, three common problems are:
(1) Histograms
(2) Roll-up Totals and Sub-Totals for drill-downs
(3) Cross Tabulations
The SQL standard `GROUP BY` operator does not allow a direct construction of **histograms** (aggregation over computed categories.) For example, for queries based on the `Weather` table, it would be nice to be able to group times into days, weeks, or months, and to group locations into areas (e.g., US, Canada, Europe,...). This would be easy if function values were allowed in the `GROUP BY` list. If that were allowed, the following query would give the daily maximum reported temperature.

```
SELECT    day, nation, MAX(Temp)
FROM      Weather
GROUP BY  Day(Time) AS day,
          Country(Latitude,Longitude)
                    AS nation;
```

Some SQL systems support histograms but the standard does not. Rather, one must construct a table-valued expression and then aggregate over the resulting table. The following statement demonstrates this SQL92 construct.

```
SELECT day, nation, MAX(Temp)
FROM (
 SELECT Day(Time) AS day,
Country(Latitude, Longitude)
                  AS nation,
        Temp
  FROM  Weather
     ) AS foo
GROUP BY day, nation;
```

A second problem relates to roll-ups using totals and subtotals for drill-down reports. Reports commonly aggregate data at a coarse level, and then at successively finer levels. The following report of car sales shows the idea. Data is aggregated by Model, then by Year, then by Color. The report shows data aggregated at three levels. Going up the levels is called **rolling-up** the data. Going down is called **drilling-down** into the data.

Table 3: Sales Roll Up by Model by Year by Color

Model	Year	Color	Sales by Model by Year by Color	Sales by Model by Year	Sales by Model
Chevy	1994	black	50		
		white	40		
				90	
	1995	black	85		
		white	115		
				200	
					290

Table 3 is not relational –null values in the primary key are not allowed. It is also not convenient -- the number of columns grows as the power set of the number of aggregated attributes. Table 4 is a relational and more convenient representation:

Table 4: Sales Summary

Model	Year	Color	Units
Chevy	1994	black	50
Chevy	1994	white	40
Chevy	1994	ALL	90
Chevy	1995	black	85
Chevy	1995	white	115
Chevy	1995	ALL	200
Chevy	ALL	ALL	290

where the dummy value "`ALL`" has been added to fill in the super-aggregation items.

The SQL statement to build this `SalesSummary` table from the raw `Sales` data is:

```
SELECT Model, ALL, ALL, SUM(Sales)
   FROM      Sales
   WHERE     Model = 'Chevy'
   GROUP BY  Model
UNION
SELECT Model, Year, ALL, SUM(Sales)
   FROM      Sales
   WHERE     Model = 'Chevy'
   GROUP BY  Model, Year
UNION
SELECT Model, Year, Color, SUM(Sales)
   FROM      Sales
   WHERE     Model = 'Chevy'
   GROUP BY  Model, Year, Color;
```

This is a simple 3-dimensional roll-up. Aggregating over *N* dimensions requires *N* such unions.

Roll-up is asymmetric – notice that the table above does not aggregate the sales by year. It lacks the rows aggregating sales by color rather than by year. These rows are:

Model	Year	Color	Units
Chevy	ALL	black	135
Chevy	ALL	white	155

These additional rows could be captured by adding the following clause to the SQL statement above:

```
UNION
SELECT Model, ALL, Color, SUM(Sales)
   FROM     Sales
   WHERE    Model = 'Chevy'
   GROUP BY Model, Color;
```

The symmetric aggregation result is a table called the **cross-tabulation,** or **cross tab** for short[2]. Cross tab data is routinely displayed in the more compact format of Table 5.

Table 5: Chevy Sales Cross Tab			
Chevy	**1994**	**1995**	**total** (ALL)
black	50	85	135
white	40	115	155
total (ALL)	90	200	290

This cross tab is a two-dimensional aggregation. If other automobile models are added, it becomes a 3D aggregation. For example, data for Ford products adds an additional cross tab plane.

Table 5a: Ford Sales Cross Tab			
Ford	**1994**	**1995**	**total** (ALL)
black	50	85	135
white	10	75	85
total (ALL)	60	160	220

The cross tab array representation is equivalent to the relational representation using the ALL value. Both generalize to an *N*-dimensional cross tab.

The representation of Table 4 and the use of unioned GROUP BYs "solves" the representation problem – it represents aggregate data in a relational data model. The problem remains that expressing histogram, roll-up, drill-down, and cross-tab queries with conventional SQL is daunting. A 6D cross-tab requires a 64-way union of 64 different GROUP BY operators to build the underlying representation. Incidentally, on most SQL systems this will result in 64 scans of the data, 64 sorts or hashes, and a long wait.

Building a cross-tabulation with SQL is even more daunting since the result is not a really a relational object – the bottom row and the right column are "unusual". Most report writers build in a cross-tabs feature, building the report up from the underlying tabular data such as Table 4 and its extension. See for example the TRANSFORM-PIVOT operator of Microsoft Access [Access].

3. The Data CUBE Operator

The generalization of these ideas seems obvious: Figure 2 shows the concept for aggregation up to 3-dimensions. The traditional GROUP can generate the core of the *N*-dimensional data cube. The *N-1* lower-dimensional aggregates appear as points, lines, planes, cubes, or hyper-cubes hanging off the core data cube.

The data cube operator builds a table containing all these aggregate values. The total aggregate is represented as the tuple:

ALL, ALL, ALL,..., ALL, f(*)

Points in higher dimensional planes or cubes have fewer ALL values. Figure 3 illustrates this idea with an example.

We extend SQL's SELECT-GROUP-BY-HAVING syntax to support histograms, decorations, and the CUBE operator.

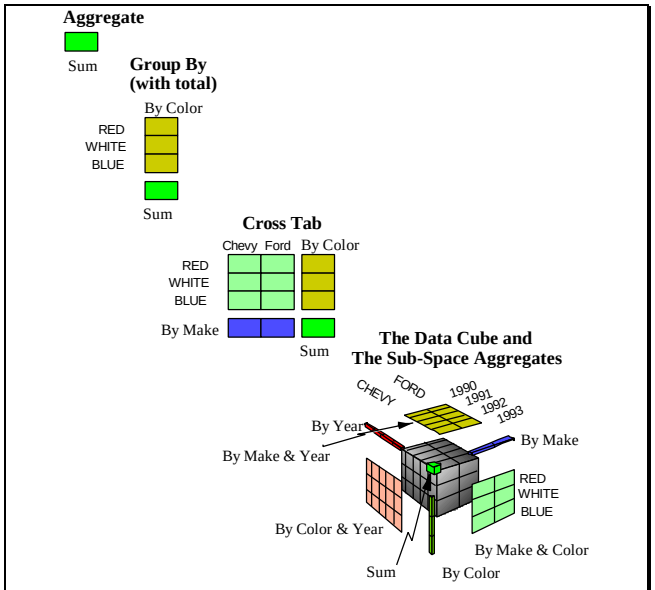

Figure 2: The CUBE operator is the *N*-dimensional generalization of simple aggregate functions. The 0D data cube is a point. The 1D data cube is a line with a point. The 2D data cube is a cross tabulation, a plane, two lines, and a point. The 3D data cube is a cube with three intersecting 2D cross tabs.

Currently the GROUP BY syntax is

[2] Spreadsheets call these *pivot-tables*.

```
GROUP BY
       {<column name> [collate clause]  ,...}
```

To support histograms, extend the syntax to:

```
GROUP BY
       { ( <column name> | <expression>)
             [   AS <correlation name> ]
             [ <collate clause>        ]
               ,...}
```

The next step is to allow *decorations*, columns that do not appear in the `GROUP BY` but that are functionally dependent on the grouping columns. Consider the example:

```
SELECT   department.name, sum(sales)
FROM     sales JOIN department
            USING (department_number)
GROUP BY sales.department_number;
```

The `department.name` column in the answer set is not allowed in current SQL, it is neither an aggregation column (appearing in the `GROUP BY` list) nor is it an aggregate. It is just there to decorate the answer set with the name of the department. We recommend the rule that *if a decoration* column (or column value) is functionally dependent on the aggregation columns, then it may be included in the `SELECT` answer list.

These extensions are independent of the `CUBE` operator. They remedy some pre-existing problems with `GROUP BY`. Some systems already allow these extensions, for example Microsoft Access allows function-valued `GROUP BY`s.

Creating the `CUBE` requires generating the power set (set of all subsets) of the aggregation columns. We propose the following syntax to extend SQL's `GROUP BY` operator:

```
 GROUP BY  CUBE (
   { ( <column name> | <expression>)
      [ AS <correlation name> ]
      [ <collate clause>      ]
    ,...}
      )
```

Figure 3 has an example of this syntax. To give another, here follows a statement to aggregate the set of temperature observations:

```
SELECT    day, nation, MAX(Temp)
FROM      Weather
GROUP BY  CUBE ( Day(Time) AS day,
       Country(Latitude, Longitude)
                         AS nation
               );
```

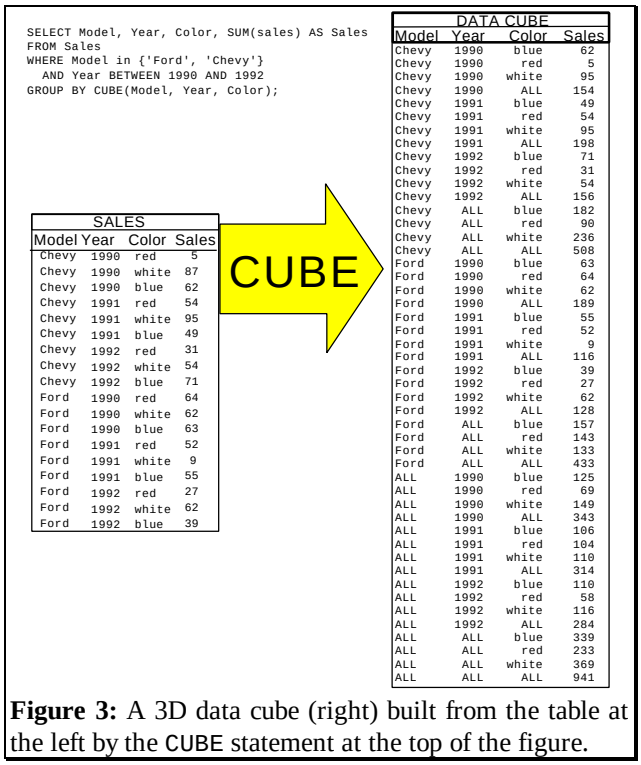

Figure 3: A 3D data cube (right) built from the table at the left by the `CUBE` statement at the top of the figure.

The semantics of the `CUBE()` operator are that it first aggregates over all the `<select list>` attributes as in a standard `GROUP BY`. Then, it `UNION`s in each super-aggregate of the global cube -- substituting `ALL` for the aggregation columns. If there are N attributes in the select list, there will be 2^N-1 super-aggregate values. If the cardinality of the N attributes are $C_1, C_2, ..., C_N$ then the cardinality of the resulting cube relation is ? $(C_i + 1)$. The extra value in each domain is `ALL`. For example, the SALES table has 2x3x3 = 18 rows, while the derived data cube has 3x4x4 = 48 rows.

Each `ALL` value really represents a set – the set over which the aggregate was computed. In the `SalesSummary` table the respective sets are:

```
Model.ALL = ALL(Model) = {Chevy, Ford }
Year.ALL  = ALL(Year)  = {1990,1991,1992}
Color.ALL = ALL(Color) = {red,white,blue}
```

Thinking of the `ALL` value as a token representing these sets defines the semantics of the relational operators (e.g., `equals` and `IN`). The ALL string is for display. A new `ALL()` function generates the set associated with this value as in the examples above. `ALL()` applied to any other value returns NULL. This design is eased by SQL3's support for set-valued variables and domains.

The ALL value appears to be essential, but creates substantial complexity. It is a non-value, like NULL. We do not add it lightly – adding it touches many aspects of the SQL language. To name a few:

- Treating each ALL value as the set of aggregates guides the meaning of the ALL value.
- ALL becomes a new keyword denoting the set value.
- ALL [NOT] ALLOWED is added to the column definition syntax and to the column attributes in the system catalogs.
- ALL, like NULL, does not participate in any aggregate except COUNT().
- The set interpretation guides the meaning of the relational operators {=, <, <=, =, >=, >, IN}.

There are more such rules, but this gives a hint of the added complexity. As an aside, to be consistent, if the ALL value is a set then the other values of that domain must be treated as singleton sets in order to have uniform operators on the domain.

Decoration's interact with aggregate values. If the aggregate tuple functionally defines the decoration column value, then the value appears in the resulting tuple. Otherwise the decoration field is NULL. For example:

```
SELECT  day,nation,MAX(Temp),
                   continent(nation)
FROM    Weather
GROUP BY  CUBE ( Day(Time) AS day,
        Country(Latitude, Longitude)
                          AS nation
                   );
```

would produce the sample tuples:

Table 6: Demonstrating decorations and ALL

day	nation	max(Temp)	continent
25/1/1995	USA	28	North America
ALL	USA	37	North America
25/1/1995	ALL	41	NULL
ALL	ALL	48	NULL

Unless nation is present, the continent is not functionally specified and so is NULL.

If the application wants only a roll-up or drill-down report, the full cube is overkill. It is reasonable to offer the additional function ROLLUP() in addition to CUBE(). ROLLUP() produces just the super-aggregates:

```
(f1 ,f2 ,...,ALL),
        ...
(f1 ,ALL,...,ALL),
(ALL,ALL,...,ALL).
```

Cumulative aggregates , like running sum or running average, work especially well with ROLLUP() since the answer set is naturally sequential (linear) while the CUBE() is naturally non-linear (multi-dimensional). Both the ROLLUP() and CUBE() must be ordered for the cumulative operators to apply.

We investigated letting the programmer specify the exact list of super-aggregates but encountered complexities related to collation, correlation, and expressions. We believe ROLLUP() and CUBE() will serve the needs of most applications.

It is convenient to know when a column value is an aggregate. One way to test this is to apply the ALL() function to the value and test for a non-NULL value. This is so useful that we propose a Boolean function GROUPING() that, given a select list element, returns TRUE if the element is an ALL value, and FALSE otherwise.

Veteran SQL implementers will be terrified of the ALL value -- like NULL, it will create many special cases. If the goal is to help report writer and GUI visualization software, then it may be simpler to adopt the following approach.

?? Use the NULL value in place of the ALL value.
?? Do not implement the ALL() function.
?? Implement the GROUPING()function to discriminate between NULL and ALL .

In this minimalist design, tools and users can simulate the ALL value as by for example:

```
SELECT  Model,Year,Color,SUM(sales),
                   GROUPING(Model),
                    GROUPING(Year),
                    GROUPING(Color)
FROM Sales
GROUP BY CUBE(Model, Year, Color);
```

Wherever the ALL value appeared before, now the corresponding value will be NULL in the data field and TRUE in the corresponding grouping field. For example, the global sum of Table 2 will be the tuple:

```
(NULL,NULL,NULL,941,TRUE,TRUE,TRUE)
```

rather than the tuple one would get with the "real" cube operator:

```
( ALL,  ALL,  ALL,  941 ).
```

4. Addressing The Data Cube

Section 5 discusses how to compute the cube and how users can add new aggregate operators. This section considers extensions to SQL syntax to easily access the elements of the data cube -- making it recursive and allowing aggregates to reference sub-aggregates.

It is not clear where to draw the line between the reporting/visualization tool and the query tool. Ideally, application designers should be able to decide how to split the function between the query system and the visualization tool. Given that perspective, the SQL

tool. Given that perspective, the SQL system must be a Turing-complete programming environment.

SQL3 defines a Turing-complete procedural programming language. So, anything is possible. But, many things are not easy. Our task is to make simple and common things easy.

The most common request is for percent-of-total as an aggregate function. In SQL this is computed as two SQL statements.

```
SELECT Model,Year,Color,SUM(Sales),
       SUM(Sales)/  (SELECT SUM(Sales)
                     FROM Sales
       WHERE Model IN { 'Ford' , 'Chevy' }
          AND Year Between 1990 AND 1992
                    )
FROM   Sales
WHERE  Model IN { 'Ford' , 'Chevy' }
  AND  Year Between 1990 AND 1992
GROUP BY CUBE (Model, Year, Color);
```

It seems natural to allow the shorthand syntax to name the global aggregate:

```
SELECT Model, Year, Color
                  SUM(Sales) AS total,
     SUM(Sales) / total(ALL,ALL,ALL)
FROM Sales
WHERE Model IN { 'Ford' , 'Chevy' }
  AND Year Between 1990 AND 1992
GROUP BY CUBE(Model, Year, Color);
```

This leads into deeper water. The next step is a desire to compute the *index* of a value -- an indication of how far the value is from the expected value. In a set of *N* values, one expects each item to contribute one *N*th to the sum. So the 1D index of a set of values is:

$index(v_i) = v_i / (? _j v_j)$

If the value set is two dimensional, this commonly used financial function is a nightmare of indices. It is best described in a programming language. The current approach to selecting an field value from a 2D `cube` with fields `row` and `column` would read as:

```
SELECT v
FROM    cube
WHERE   row    = :i
 AND    column = :j
```

We recommend the simpler syntax:

```
cube.v(:i, :j)
```

as a shorthand for the above selection expression. With this notation added to the SQL programming language, it should be fairly easy to compute super-super-aggregates from the base cube.

5. Computing the Data Cube

`CUBE` generalizes aggregates and `GROUP BY`, so all the technology for computing those results also applies to computing the core of the cube. The main techniques are:

- To minimize data movement and consequent processing cost, compute aggregates at the lowest possible system level.
- If possible, use arrays or hashing to organize the aggregation columns in memory, storing one aggregate value for each array or hash entry.
- If the aggregation values are large strings, it may be wise to keep a hashed symbol table that maps each string to an integer so that the aggregate values are small. When a new value appears, it is assigned a new integer. With this organization, the values become dense and the aggregates can be stored as an *N*-dimensional array.
- If the number of aggregates is too large to fit in memory, use sorting or hybrid hashing to organize the data by value and then aggregate with a sequential scan of the sorted data.
- If the source data spans many disks or nodes, use parallelism to aggregate each partition and then coalesce these aggregates.

Some innovation is needed to compute the "`ALL`" tuples of the cube from the `GROUP BY` core. The `ALL` value adds one extra value to each dimension in the `CUBE`. So, an *N*-dimensional cube of *N* attributes each with cardinality C_i, will have $? (C_i+1)$. If each C_i =4 then a 4D `CUBE` is 2.4 times larger than the base `GROUP BY`. We expect the C_i to be large (tens or hundreds) so that the `CUBE` will be only a little larger than the `GROUP BY`.

The cube operator allows many aggregate functions in the aggregation list of the `GROUP BY` clause. Assume in this discussion that there is a single aggregate function F() being computed on an *N*-dimensional cube. The extension to a computing a list of functions is a simple generalization.

The simplest algorithm to compute the cube is to allocate a handle for each cube cell. When a new tuple: (x_1, x_2,...., x_N, v) arrives, the `Iter(handle, v)` function is called 2^N times -- once for each handle of each cell of the cube matching this value. The 2^N comes from the fact that each coordinate can either be x_i or `ALL`. When all the input tuples have been computed, the system invokes the `final(&handle)` function for each of the $? (C_i+1)$ nodes in the cube. Call this the ***2^N-algorithm***.

If the base table has cardinality *T*, the *2^N-algorithm* invokes the `Iter()` function *T x* 2^N times. It is often faster to compute the super-aggregates from the core `GROUP BY`, reducing the number of calls by approximately a factor of *T* . It is often possible to compute the cube from the core or from intermediate results only *M* times larger than the core. The following trichotomy characterizes the options in computing super-aggregates.

Consider aggregating a two dimensional set of values $\{X_{ij} \mid i = 1,...,I;\ j=1,...,J\}$. Aggregate functions can be classified into three categories:

Distributive: Aggregate function *F()* is distributive if there is a function *G()* such that $F(\{X_{i,j}\}) = G(\{F(\{X_{i,j} \mid i=1,...,I\}) \mid j=1,...J\})$. `COUNT()`, `MIN()`, `MAX()`, `SUM()` are all distributive. In fact, *F = G* for all but `COUNT()`. G= `SUM()` for the `COUNT()` function. Once order is imposed, the cumulative aggregate functions also fit in the distributive class.

Algebraic: Aggregate function *F()* is algebraic if there is an *M*-tuple valued function *G()* and a function *H()* such that $F(\{X_{i,j}\}) = H(\{G(\{X_{i,j} \mid i=1,..., I\}) \mid j=1,..., J\})$. Average(), standard deviation, MaxN(), MinN(), center_of_mass() are all algebraic. For Average, the function *G()* records the sum and count of the subset. The *H()* function adds these two components and then divides to produce the global average. Similar techniques apply to finding the *N* largest values, the center of mass of group of objects, and other algebraic functions. The key to algebraic functions is that a fixed size result (an M-tuple) can summarize the sub-aggregation.

Holistic: Aggregate function *F()* is holistic if there is no constant bound on the size of the storage needed to describe a sub-aggregate. That is, there is no constant *M*, such that an *M*-tuple characterizes the computation $F(\{X_{i,j} \mid i=1,...,I\})$. Median(), MostFrequent() (also called the Mode()), and Rank() are common examples of holistic functions.

We know of no more efficient way of computing super-aggregates of holistic functions than the 2^N-algorithm using the standard `GROUP BY` techniques. We will not say more about cubes of holistic functions.

Cubes of distributive functions are relatively easy to compute. Given that the core is represented as an *N*-dimensional array in memory, each dimension having size C_i+1, the *N-1* dimensional slabs can be computed by projecting (aggregating) one dimension of the core. For example the following computation aggregates the first dimension.

`CUBE(ALL,` $x_2,...,x_N$) = F({`CUBE(i,` $x_2,...,x_N$) | i = $1,...C_1$}).

N such computations compute the *N-1* dimensional super-aggregates. The distributive nature of the function F() allows aggregates to be aggregated. The next step is to compute the next lower dimension -- an (...`ALL`,..., `ALL`...) case. Thinking in terms of the cross tab, one has a choice of computing the result by aggregating the lower row, or aggregating the right column (aggregate (`ALL`, *) or (*, `ALL`)). Either approach will give the same answer. The algorithm will be most efficient if it aggregates the smaller of the two (pick the * with the smallest C_i.) In this way, the super-aggregates can be computed dropping one dimension at a time.

Algebraic aggregates are more difficult to compute than distributive aggregates. Recall that an algebraic aggregate saves its computation in a handle and produces a result in the end - at the `Final()` call. `Average()` for example maintains the count and sum values in its handle. The super-aggregate needs these intermediate results rather than just the raw sub-aggregate. An algebraic aggregate must maintain a handle (*M*-tuple) for each element of the cube (this is a standard part of the group-by operation). When the core `GROUP BY` operation completes, the `CUBE` algorithm passes the set of handles to each *N-1* dimensional super-aggregate. When this is done the handles of these super-aggregates are passed to the super-super aggregates, and so on until the (`ALL`, `ALL`, ..., `ALL`) aggregate has been computed. This approach requires a new call for distributive aggregates:

```
Iter_super(&handle, &handle)
```

which folds the sub-aggregate on the right into the super aggregate on the left. The same ordering ideas (aggregate on the smallest list) applies.

If the data cube does not fit into memory, array techniques do not work. Rather one must either partition the cube with a hash function or sort it. These are standard techniques for computing the `GROUP BY`. The super-aggregates are likely to be orders of magnitude smaller than the core, so they are very likely to fit in memory.

It is possible that the core of the cube is sparse. In that case, only the non-null elements of the core and of the super-aggregates should be represented. This suggests a hashing or a B-tree be used as the indexing scheme for aggregation values [Essbase].

6. Summary:

The cube operator generalizes and unifies several common and popular concepts:

aggregates,
group by,
histograms,
roll-ups and drill-downs and,
cross tabs.

The cube is based on a relational representation of aggregate data using the ALL value to denote the set over which each aggregation is computed. In certain cases it makes sense to restrict the cube to just a roll-up aggregation for drill-down reports.

The cube is easy to compute for a wide class of functions (distributive and algebraic functions). SQL's basic set of five aggregate functions needs careful extension to include functions such as rank, N_tile, cumulative, and percent of total to ease typical data mining operations.

7. Acknowledgments

Joe Hellerstein suggested interpreting the ALL value as a set. Tanj Bennett, David Maier and Pat O'Neil made many helpful suggestions that improved the presentation.

8. References

[Access] *Microsoft Access Relational Database Management System for Windows, Language Reference -- Functions, Statements, Methods, Properties, and Actions*, DB26142, Microsoft, Redmond, WA, 1994.

[Essbase] *Method and apparatus for storing and retrieving multi-dimensional data in computer memory*, Inventor: Earle; Robert J., Assignee: Arbor Software Corporation, US Patent 05359724, October 1994,

[Illustra] *Illustra DataBlade Developer's Kit 1.1*., Illustra Information Technologies, Oakland, CA, 1994.

[Melton & Simon] Jim Melton and Alan Simon, *Understanding the New SQL*: A Complete Guide, Morgan Kaufmann, San Francisco, CA, 1993.

[Red Brick] *RISQL Reference Guide, Red Brick Warehouse VPT Version 3*, Part no: 401530, Red Brick Systems, Los Gatos, CA, 1994

[TPC] *The Benchmark Handbook for Database and Transaction Processing Systems - 2nd edition*, J. Gray (ed.), Morgan Kaufmann, San Francisco, CA, 1993.