Business Modeling Using SQL Spreadsheets

Andrew Witkowski, Srikanth Bellamkonda, Tolga Bozkaya, Nathan Folkert, Abhinav Gupta, Lei Sheng, Sankar Subramanian

Oracle Corporation
500 Oracle Parkway, Redwood Shores, CA 94065, U.S.A.
{andrew.witkowski, srikanth.bellamkonda, tolga.bozkaya, nathan.folkert, abhinav.gupta, lei.sheng, sankar.subramanian}@oracle.com

Abstract

One of the critical deficiencies of SQL is the lack of support for array and spreadsheet like calculations which are frequent in OLAP and Business Modeling applications. Applications relying on SQL have to emulate these calculations using joins, UNION operations, Window Functions and complex CASE expressions. The designated place in SQL for algebraic calculations is the SELECT clause, which is extremely limiting and forces applications to generate queries with nested views, subqueries and complex joins. This distributes Business Modeling computations across many query blocks, making applications coded in SQL hard to develop. The limitations of RDBMS have been filled by spreadsheets and specialized MOLAP engines which are good at formulas for mathematical modeling but lack the formalism of the relational model, are difficult to manage, and exhibit scalability problems. This demo presents a scalable, mathematically rigorous, and performant SQL extensions for Relational Business Modeling, called the SQL Spreadsheet. We present examples of typical Business Modeling computations with SQL spreadsheet and compare them with the ones using standard SQL showing performance advantages and ease of programming for the former. We will show a scalability example where data is processed in parallel and will present a new class of query optimizations applicable to SQL spreadsheet.

1 Introduction

Spreadsheets have been one of the most successful analytical tools. Data and formulas reside in one place which is convenient for rapid prototyping and formulas view data using a convenient two-dimensional array abstractions. Complex business models can be built with recursive and simultaneous equations and a rich set of business functions is provided for ease of use. Finally, a very flexible user interfaces with graphs and reports is provided.

Spreadsheets, however, have problems. There is no separation between data and formulas which results in unstructured, ill-defined models. The two dimensional “row-column” array abstraction is not well suited for building symbolic models or models of higher dimensionality. A significant scalability problem exists when either the data set is large (can one define a spreadsheet with terabytes of sales data?) or the number of formulas is significant (can one process tens of thousands of spreadsheet formulas in parallel?). In collaborative analysis with multiple spreadsheets, consolidation is difficult: it is nearly impossible to get a complete picture of the business by querying multiple spreadsheets each using its own layout and placement of data.

So far Business Modeling users who looked to the RDBMS for help with these problems have been disappointed as SQL analytical usefulness has not measured up to that of spreadsheets. It is cumbersome and inefficient to perform array-like calculations in SQL -- a fundamental problem resulting from lack of language constructs to treat relations as arrays and lack of efficient access and optimization methods for evaluating formulas over the arrays.

In [1] we have proposed SQL extensions, optimizations and an execution model, called the *SQL Spreadsheet,* which makes the RDBMS suitable for Relational Business Modeling. The salient features of SQL Spreadsheet are:

- Relations can be viewed as n-dimensional arrays.
- Formulas can be defined over the arrays and can automatically be ordered based on their dependencies.
- Recursive formulas and convergence conditions are supported thus supporting simultaneous equations.
- Evaluation can be order driven supporting sequenced computations like moving averages and cumulative sums.
- Formulas are encapsulated in a new SQL query clause that supports partitioning of the data. This allows evaluation of formulas independently for each partition providing a natural parallelization of execution.
- Formulas support UPSERT and UPDATE semantics as well as correlation between their left and right side. This allows us to simulate the effect of multiple joins and UNIONs using a single access structure.

This demo will present typical usage of SQL Spreadsheet for Relational Business Modeling on a real-life size (GB of data) data warehouse.

2 SQL Extensions For Spreadsheets

For completeness we summarize the language features of SQL Spreadsheet presented in [1].

Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003

Notation. Our examples are illustrated using an electronic data warehouse schema with fact table f(t, r, p, s, c) with three dimensions: time (t), region (r), and product (p), and two measures: sales (s) and cost (c).

Spreadsheet clause. ROLAP & Business Modeling applications divide relational attributes into dimensions and measures. To model that, we introduce a new SQL query clause, called the *spreadsheet claus*e, which identifies, within the query result, *partition*, *dimension* and *measure* columns. The partition (PBY) columns divide the relation into disjoint subsets. The dimension (DBY) columns uniquely identify a row within each partition, which we call a *cell*, and serve as array indexes to the measure columns. The measure (MEA) columns identify expressions computed by the spreadsheet. Following this, there is a sequence of formulas, each describing a computation on cells. Thus the structure of the spreadsheet clause is:

```
<existing parts of a query block>
SPREADSHEET PBY (cols) DBY (cols) MEA (cols)
<processing options>
(
      <formula>, <formula>,.., <formula>
)
```

It is evaluated after joins, aggregations, and window functions but before final projection and the ORDER BY clause.

Cells are referenced using a familiar array notation. Cell references can designate a *single cell reference* when dimensions are uniquely qualified e.g., s[p='dvd', t=2002], or set of cells where dimensions are qualified by predicates e.g., s[p='dvd', t<2002].

Each formula represents an assignment and contains a left side that designate target cells and a right side that contains expressions involving cells within the partition. For example, the query:

```
SELECT r, p, t, s
FROM f
SPREADSHEET PBY(r) DBY (p, t) MEA (s)
(
  s[p='vcr',t=2002] = s[p='vcr',t=2000]
                    + s[p='vcr',t=2001],
  s[p='tv', t=2002] =avg(s)[p='tv',1992<t<2002]
)
```

partitions table *f* by region r and defines that sales, within each region, of 'vcr' in 2002 will be the sum of sales in 2000 and 2001, and sales of 'tv' will be the average of years between 1992 and 2002. As a shorthand, a positional notation exists, for example: s['dvd',2002] instead of s[p='dvd',t=2002].

The left side of a formula can define calculations which span a range of cells. A function *cv()* carries the current value of a dimension from the left side to the right side thus effectively serving as a join between right and left side. The ANY operator denotes all values in the dimension. For example:

```
SPREADSHEET DBY (r, p, t) MEA (s)
(
 s['west',ANY,t>2001]=1.2*s[cv(r),cv(p),cv(t)-1]
)
```

states that sales of every product in 'west' region for year > 2001 will be 20% higher than sales of the same product in the preceding year. Region, product and time dimensions on the right side reference function *cv()* to carry dimension values from left to the right side.

For formulas which update a range of cells, the result may depend on the order in which cells are processed, and for these cases we require explicit specification of the ordering. For example, in the following formula which specifies that sales of 'vcr' for all years before 2002 is an average of two preceding years, rows should be processed in ascending order of the time dimension, expressed as:

```
SPREADSHEET PBY(r) DBY (p, t) MEA (s)
(
  s['vcr', t<2002] ORDER BY t ASC =
      avg(s)[cv(p),cv(t)-2<=t<cv(t)]
)
```

SQL spreadsheet can create new rows in the result set thus effecting SQL UNION operation. A formula with a single cell reference on the left side can operate either in UPDATE or UPSERT (default) mode. The latter creates new cells within a partition if they do not exist, otherwise it updates them. UPDATE mode ignores nonexistent cells. For example,

```
SPREADSHEET PBY(r) DBY (p, t) MEA (s)
(
  UPSERT s['tv', 2000] =
      s['black-tv',2000] + s['white-tv',2000]
)
```

will create for each region a row with p='tv' and t=2000 if this cell is not present in the input stream.

Reference Spreadsheets. OLAP applications frequently deal, in a single business query, with objects of different dimensionality. For example, the sales table may have region(*r*), product(*p*), and time(*t*) dimensions, while the budget allocation table has only a region(*r*) dimension. To account for that, our query block can have, in addition to the main spreadsheet, multiple, read-only reference spreadsheets which are n-dimensional arrays defined over other query blocks. Reference spreadsheets, akin to main spreadsheets, have DBY and MEA clauses indicating their dimensions and measures respectively. For example, assume a budget table budget(r, pr) containing predictions *pr* for sales increase for each region *r*. The following query predicts sales in 2002 in region 'west' scaling them using prediction *pr* from the budget table.

```
SELECT r, t, s
FROM f GROUP by r, t
SPREADSHEET
  REFERENCE budget ON (SELECT r, pr FROM budget)
                     DBY(r) MEA(p)
DBY (r, t) MEA (sum(s) s)
(
  s['west',2002]= pr['west']*s['west',2001],
  s['east',2002]= s['east',2001]+s['east',2000]
)
```

The purpose of a reference spreadsheet is similar to a relational join, but it allows us to perform, within a spreadsheet clause, multiple joins using the same access structures (in our case hash or index structure), thus self-joins within spreadsheet can be cheaper than outside of it.

Ordering The Evaluation Of Formulas. By default, the evaluation of formulas occurs in the order of their dependencies, and we refer to it as the AUTOMATIC ORDER. For example in

```
SPREADSHEET PBY(r) DBY (p, t) MEA (s)
(
  s['dvd',2002] = s['dvd',2000] + s['dvd',2001]
  s['dvd',2001] = 1000
)
```

the first formula depends on the second, and consequently we will evaluate the latter one first. For scenarios where lexicographical ordering of evaluation is desired we provide an explicit processing option, called SEQUENTIAL ORDER.

```
SPREADSHEET DBY(r,p,t) MEA(s) SEQUENTIAL ORDER
(. ..<formulas>....)
```

Cycles and Recursive Models. Similarly to existing spreadsheet, our computations may contain cycles, as in the formula:

s[1] = s[1]/2

Consequently we have processing options to specify the number of iterations or the convergence criteria for cycles and recursion. The ITERATE (n) option requests iteration of the formulas 'n' times. The optional UNTIL condition will stop the iteration when the <condition> has been met. The <condition> can reference cells before and after the iteration facilitating the definition of convergence conditions. A helper function *previous*(<cell>) returns the value of <cell> at the start of each iteration. For example,

```
SPREADSHEET DBY (x) MEA (s)
  ITERATE (10) UNTIL (PREVIOUS(s[1])-s[1] <= 1)
( s[1] = s[1]/2 )
```

will execute the formula s[1] = s[1]/2 until the convergence condition is met, up to a maximum of 10 iterations (in this case if initially s[1] is greater than or equal to 1024, evaluation of the formulas will stop after 10 iterations). For programming tasks, we export to the formulas current iteration number with a function ITERATION_NUMBER.

Spreadsheet Processing Options and Miscellaneous functions. There are other processing options for the SQL spreadsheet in addition to the ones for ordering of formulas and termination of cycles. For example, we can specify UPDATE/ UPSERT option as a default for the entire spreadsheet. The option IGNORE NAV allows us to treat NULL values in numeric operations as 0, which is convenient for newly inserted cells with the UPSERT option.

3 SQL Spreadsheet Functionality Examples

Here is are some examples showing the expressive power of the SQL spreadsheet and its potential for efficient computation compared to the alternative in ANSI SQL.

Computing Time-series and Parenthood ratios. In ROLAP databases, hierarchical dimensions and cubes are frequently expressed using the *Embedded Dimension* (*ED*) form which encodes in a single column all levels of a dimension. For example, for a time dimension time_dt(y,q,m) with year, quarter and month hierarchy this form is expressed as:

```
SELECT ed(y,q,m) AS ed_time
FROM time
GROUP BY ROLLUP (y,q,m)
```

where the function *ed* returns m (month) if grouping is by month level i.e., (y,q,m), q when grouping by quarter level, i.e, (y,q) , and y when grouping by (y). The ED form allows us to express such time-series entities as same-period-N-levels-ago. For example, sales of year-ago of the month level will be sale the same month year ago, of the quarter level the same quarter a year ago, etc.: Table 1 illustrates mapping of ED form of time t to same period year ago *yago*, same period quarter ago *qago* and some period month ago *mago*.

Table 1: time_ed table. Mapping between t and yago, qago, mago all expressed in ED form for time.

t	yago	qago	mago
1999-m01	1998-m01	1998-m10	1998-m12
1999-m02	1998-m02	1998-m11	1999-m01
1999-m03	1998-m03	1998-m12	1999-m02
..			
1999-q01	1998-q01	1998-q04	-
...			
1999-y	1998-y	-	-

Financial applications frequently compute ratios of current measures to measures same-period-N-levels-ago to discover patterns of change. In ANSI SQL this requires multiple self joins of the fact table. In SQL Spreadsheet, this has a very elegant and efficient representation. For example, the following query computes ratio of current sales to that of year ago (*r_yago*), quarter ago (*r_qago*) and month ago (*r_mago*).

```
SELECT p, ed(y,q,m) t, s
FROM f, time_dt
WHERE f.m = time_dt.m
GROUP BY p, rollup(y,q,m)
SPREADSHEET
  REFERENCE ON
   (SELECT t, yago, qago, mago FROM time_ed)
    DBY(t) MEA(yago, qago, mago)
PBY(p) DBY(t) MEA(sum(s)s, r_yago,r_qago,r_mago)
(
  F1: r_yago[ANY] = s[cv(t)] / s[yago[cv(t)]],
  F2: r_qago[ANY] = s[cv(t)] / s[qago[cv(t)]],
  F3: r_mago[ANY] = s[cv(t)] / s[mago[cv(t)]]
)
```

The reference spreadsheet serves as a one-dimensional look-up table translating, using *time_ed* table, time *t* into the corresponding period a year, quarter and month ago. Formula F1-F3 calculates the desired ratios. An alternative formulation of the query using ANSI SQL requires the joins *f* >< *time_dt* >- *f* >- *f* >- *f* where the first join provides lookups from the fact to the *time_ed* table and the three following outer joins (>-) provide measure values in year, quarter and month ago periods. SQL spreadsheet provides a significant performance advantage over the ANSI SQL formulation as it reduces number of joins. We build a single random access structure (hash table in our case) to satisfy all rules. In ANSI SQL we have to execute one outer join per rule, which in case of hash joins, multiplies the number of access structures needed.

Densification of Data. Data stored in ROLUP databases is frequently sparse, i.e., only some combination of dimension values are present in fact tables. Densification on a dimension *d* assures that all *d* values are present in the output for every existing combination of other dimensions. Combinations not present in fact tables, will have null values in the measure columns. Densification is frequently needed in time-series where all time values must be present in the output and is used for moving averages, prior-period computation, calendar construction, etc. Assume that for each product(*p*) and region(*r*) we want to ensure that all years present in the dimension table, *time_dt*, are present in the output. The fact table f, is sparse, and may not have all time periods for every product-region pair. Using our spreadsheet this is expressed as:

```
SELECT r, p, t, s
FROM f
SPREADSHEET PBY(r, p) DBY (t) MEA (s, 0 as x)
(
  UPSERT x[FOR t IN (SELECT t FROM time_dt)]= 0
)
```

This partitions the data by (*p*,*r*) and within each partition upserts all values from the time dimension and assigns to measure *x* a default value 0. An equivalent formulation using ANSI SQL involves a cartesian product of *f* to *time_td* and a joinback to *f*, a series of operations much less efficient than these required for the above spreadsheet execution:

```
SELECT f.r, f.p, f.t, f.s
FROM f RIGHT OUTER JOIN
     ( (SELECT DISTINCT r, p FROM f)
        CROSS JOIN
        (SELECT t FROM time_dt)
       ) v
   ON (f.r = v.r and f.p = v.p and f.t = v.t)
```

Recursive Model Solving. Many financial applications solve a set of simultaneous equations and many of them can now be directly expressed in SQL. For example, an application for personal general ledger containing accounts and their balances specifies that:

1. Net pay is salary minus interest, minus tax (see F1)
2. Taxes are 38% of (salary-interest) and 28% of capital gains (F2)
3. We want 30% of net income as interest (F3)

This model can be expressed with 3 simultaneous formulas shown below. The formulas are recursive as formula F1 depends on F2, F2 depends on F3 and F3 depends on F1. We solve that model re-executing the formulas until the difference between net pay from previous and current iteration is greater than $1 for a maximum of 1000 iterations.

```
SELECT account, s
FROM ledger
SPREADSHEET IGNORE NAV  DBY(acct) MEA (sums s)
ITERATE 1000 UNTIL (PREVIOUS(s['net'])-s['net']>1
(
F1: s['net']=s['salary']-s['interest']-s['tax']
F2: s['tax']=(s['salary']-s['interest'])*0.38
                          +s['capital_gains']*0.28
F3: s['interest']=s['net']*0.30
 )
```

FIGURE 1.   Recursive Model Solving

account	sums
salary	10,000
capital_g	1,500
net	-
tax	-
interest	-

account	sums
salary	10,000
capital_g	1,500
net	5,227
tax	3,204
interest	1,568

The above figure shows the input ledger table where salary and capital gains are given and the output after SQL spreadsheet. It is difficult to state an equivalent ANSI SQL formulation for this problem.

4 Spreadsheets Optimization & Execution

As shown in [1], our implementation of SQL Spreadsheet contains novel optimizations which include:

- Parallelization of Formulas. The PBY clause provides a partitioning of the data, thus enabling formula computations in parallel. If the partitioning is not specified, we infer the partitioning and parallelization for typical business cases.
- Pruning of Formulas. The formulas whose results are not referenced in outer blocks can be removed from the spreadsheet, thus removing unnecessary computations.
- Predicate Pushing. The predicates from other query blocks can be moved inside query blocks with spreadsheets, thus considerably reducing the amount of data to be processed.

Execution methods include a novel algorithm for discovering the convergence of spreadsheet formulas and two structures for random-access, symbolic addressing in formulas: a multi-dimensional index and hash table (picked if the index is not available).

The optimizations, access methods, and run-time rule convergence algorithm will be demonstrated on a real-life size (several GBytes) data warehouse.

5 Demo Content

The demo will contain several GB data warehouse populated with synthetic data. The schema resembles that used for the APB [2] benchmark with a fact table and 4 dimensions. The dimensions will have 2, 3, 3, and 7 levels. We will present typical Business Modeling solutions using SQL Spreadsheet and compare their formulations and performance to that of standard SQL. We present time-series and parenthood computations and recursive financial models. In addition, we will show sparse matrix multiplication done directly with SQL. We will demonstrate, using Oracle's explain plan facility, optimizations mentioned in Section 4. We will also demonstrate an automatic conversion tool between Excel and SQL Spreadsheet. It stores the Excel data cells in relational tables and Excel formulas in relational views with SQL Spreadsheet clause.

6 References

[1] "Spreadsheet in RDBMS for OLAP", SIGMOD 2003, San Diego, USA.

[2] "APB Benchmark Specifications", http://www.olapcouncil.org/research/APB1R2_spec.pdf