# Spreadsheets in RDBMS for OLAP

Andrew Witkowski, Srikanth Bellamkonda, Tolga Bozkaya, Gregory Dorman, Nathan Folkert, Abhinav Gupta, Lei Shen, Sankar Subramanian

Some algorithms in this paper have US patent pending

{andrew.witkowski, srikanth.bellamkonda, tolga.bozkaya, gregory.dorman, nathan.folkert, abhinav.gupta, lei.shen, sankar.subramanian}@oracle.com

## Abstract

One of the critical deficiencies of SQL is lack of support for n-dimensional array-based computations which are frequent in OLAP environments. Relational OLAP (ROLAP) applications have to emulate them using joins, recently introduced SQL Window Functions [18] and complex and inefficient CASE expressions. The designated place in SQL for specifying calculations is the SELECT clause, which is extremely limiting and forces the user to generate queries using nested views, subqueries and complex joins. Furthermore, SQL-query optimizer is pre-occupied with determining efficient join orders and choosing optimal access methods and largely disregards optimization of complex numerical formulas. Execution methods concentrated on efficient computation of a cube [11], [16] rather than on random access structures for inter-row calculations. This has created a gap that has 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 paper presents SQL extensions involving array based calculations for complex modeling. In addition, we present optimizations, access structures and execution models for processing them efficiently.

## 1 Introduction

One of the most successful analytical tools for business data is a spreadsheet. A user can enter business data, define formulas over it using two-dimensional array abstractions, construct simultaneous equations with recursive models, pivot data and compute aggregates for selected cells, apply a rich set of business functions, etc. They also provide flexible user interfaces like graphs and reports.

Unfortunately, analytical usefulness of the RDBMS has not measured up to that of spreadsheets or specialized MOLAP tools [2]. It is cumbersome and in most cases inefficient to perform array-like calculations in SQL -- a fundamental problem resulting from lack of language constructs to treat relations as arrays and to define formulas over them and lack of efficient random access methods for array accesses.



SIGMOD 2003, June 9-12, 2003, San Diego, CA



Spreadsheets provide a terrific user interface but, on the other hand, have their own problems. They offer two dimensional "row-column" addressing, i.e. physical addressing using row and column offsets. Hence, it is hard to build a symbolic model where formulas reference actual data values. 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 as 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. There is no standard metadata or a unified abstraction inter-relating them akin to RDBMS dictionary tables and RDBMS relations.

This paper proposes spreadsheet-like computations in RDBMS through extensions to SQL, leaving the user interface aspects to be handled by OLAP tools. Here is a glimpse of our proposal:

- Relations can be viewed as n-dimensional arrays, and formulas can be defined over their cells. Cell addressing is symbolic, using dimensional columns.
- The formulas can automatically be ordered based on the dependencies between the cells.
- Recursive references and convergence conditions are supported providing for a recursive execution model.
- OLAP applications frequently fill gaps in sparse data, an operation called densification which is difficult in ANSI SQL, but natural in the proposed SQL-spreadsheet.
- Formulas are encapsulated in a new SQL query clause evaluated after the existing query clauses. Since this is an extension to the query block, the result is a relation and can be further used in joins, subqueries, etc.
- The new clause 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 paper also describes optimizations and execution strategies possible with the proposed extensions. For instance:

- The partitioning of data provides an obvious way to parallelize the computation of spreadsheet and provide scalability. If the partitioning is not explicitly specified, our optimizer can automatically infer the partitioning in some cases.
- Efficient hash based access structures on relations can be used for symbolic array addressing, enabling fast computation of formulas.
- The formulas whose results are not referenced in outer blocks can be removed from spreadsheet, thus removing unnecessary computations.
- The predicates from other query blocks can be moved inside query blocks with spreadsheets, thus considerably reducing the amount of data to be processed. Conditions for validity of this transformation are given.

This paper is organized as follows. Section 2 provides SQL language extensions for spreadsheet. Section 3 provides motivating examples and comparisons to equivalent processing in SQL. Section 4 describes analysis of the spreadsheet clause and query optimizations with spreadsheets. Section 5 discusses our execution models. Section 6 reports results from experiments on spreadsheet queries. Section 7 concludes and suggests topics for further research.

## 2 SQL Extensions For Spreadsheets

**Notation**. In the following examples we will use a fact table f(t, r, p, s, c) representing a data-warehouse of electronic products with three dimensions: time (t), region (r), and product (p), and two measures: sales (s) and cost (c).

**Spreadsheet clause**. OLAP applications divide relational attributes into dimensions and measures. To model that, we introduce a new SQL query clause, called the *spreadsheet clau*se, which identifies, within the query result, PARTITION, DIMENSION and MEASURES 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 index to the measure columns. The MEASURES (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, window function and, final projection, but before 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 called a *range reference* e.g. s[p='dvd', t<2002] where dimensions are qualified by predicates.

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

```
SELECT r, p, t, s
FROM f
SPREADSHEET PBY(r) DBY (p, t) MEA (s)
(
  s[p='dvd',t=2002] = s[p='dvd',t=2001]*1.6,
  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 within each region, sales of 'dvd' in 2002 will be 60% higher than in 2001, sales 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 defines calculations which can span a range of cells. A new function *currentv()* (referred to as *cv()* in short) carries the value of a dimension from the left side to the right side thus effectively serving as a join between right and left side. The * operator denotes all values in the dimension. For example:

```
SPREADSHEET DBY (r, p, t) MEA (s)
(
  s['west',*,t>2001]=
    1.2*s[cv(r),cv(p),t=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. Observe that region and product dimensions on the right side reference function *cv()* to carry dimension values from left to the right side.

Formulas may specify a range of cells to be updated. A formula referring to multiple cells on the left side is called an *existential formula*. For existential formulas, the result may be order dependent. For example the intention of

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

is that the sales of 'vcr' for all years before 2002 is an average of two preceding years. Processing rows in ascending and descending order w.r.t dimension t produces different results as we are both updating and referencing measure s. Such cases are detected by the compiler [Section 4 on page 4] and executed using cyclic algorithm [Section 5 on page 7]. To avoid ambiguity, the user can specify an order in which the rule should be evaluated :

```
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)]
)
```

An innovative feature of SQL spreadsheet is creation of new rows in the result set. Any formula, with a single cell reference on 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 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. An existential formula defines a range of dimension values on left side. Since these dimensions may belong to a continuous domain and it is not always possible to find out the individual set of values, we do not allow upsert mode with an existential formula.

**Reference Spreadsheets.** OLAP applications frequently deal, in a single 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 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,p) containing predictions *p* for sales increase for each region *r*. The following query predicts sales in 2002 in region 'west' scaling them using prediction *p* from the budget table.

```
SELECT r, t, s
FROM f GROUP by r, t
SPREADSHEET
  REFERENCE budget ON (SELECT r, p FROM budget)
                      DBY(r) MEA(p)
DBY (r, t) MEA (s)
(
  s['west',2002]= p['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 (e.g., hash table - see Section 5), 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. However, there are scenarios in which lexicographical ordering of evaluation is desired. We provide an explicit processing option, SEQUENTIAL ORDER, for that as in:

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

**Cycles and Recursive Models.** Similarly to existing spreadsheets 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 upto a maximum of "n" iterations as specified by ITERATE(n). The <condition> can reference cells before and after the iteration facilitating 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).

**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 options as 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.

The new predicate *<cell> IS PRESENT* indicates if the row indicated by the <cell> existed before the execution of the spreadsheet clause and is convenient for determining upserted values.

## 3 Motivating Example of Spreadsheet Usage

Here is an example demonstrating the expressive power of the SQL spreadsheet and its potential for efficient computation as compared to the alternative available in ANSI SQL.

An analyst predicts sales for the year 2002. Based on business trends, sales of 'tv' in 2002 is their sales in 2001 scaled by the average increase between 1992 and 2001. Sales of 'vcr' is the sum of their sales in 2000 and 2001. Sales of 'dvd' is the average of the three previous years. Finally, the analyst wants to introduce in every region a new dimension member 'video' for the year 2002, defined as sales of 'tv' plus sales of 'vcr'. Assuming that rows for 'tv', 'dvd', 'vcr' for year 2002 already exist, we can express the analyst's query as:

```
SELECT r, p, t, s
FROM f
SPREADSHEET PBY(r) DBY (p, t) MEA (s)
(
F1: UPDATE s['tv',2002] =
      slope(s,t)['tv',1992<=t<=2001]*s['tv',2001]
      + s['tv',2001]
F2: UPDATE s['vcr', 2002] =
      s['vcr', 2000] + s['vcr', 2001],
F3: UPDATE s['dvd',2002] =
   (s['dvd',1999]+s['dvd',2000]+s['dvd',2001])/3,
F4: UPSERT s['video', 2002] =
      s['tv',2002] + s['vcr',2002]
)
```

The *slope()* aggregate is a recent addition to ANSI SQL [18] and denotes linear regression slope. To express the above query in ANSI SQL, formula F1 would require an aggregate subquery plus a

join to the fact table f, formula F2 a double self-join of the fact table, formula F3 a triple self join of the fact table, and formula F4 a union operation. Such a query would not only be difficult to generate but would also result in an inefficient execution as compared to the query with spreadsheet. For the latter we need to scan the data and generate a point addressable access structure like a hash table or an index for all formulas only once. If we can deduce from database constraints that t is from an integer domain, then formula F1 is first transformed into

```
F1: UPDATE s['tv',2002] =
slope(s,t)['tv',t in (1992,.,2001)]*s['tv',2001]
+ s['tv',2001]
```

This way, the access structure can be used for random, multiple accesses along the time dimension as opposed to a scan to find out the rows satisfying the predicate. Formulas F2, F3, and F4 can use the structure directly. The structure is then used multiple times giving a performance advantage over multiple joins required by equivalent ANSI SQL. In real applications, we expect hundreds of formulas and consequently building a single point access structure in place of hundreds of joins provides a significant performance advantage.

As another example consider the "densification" of a dimension *d* - a process which assures that all *d* values are present in the output for every combination of other dimensions. This operation is frequently used in time-series where all time values must be present in the output. This 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[1])
(
  UPSERT x[FOR t IN (SELECT t FROM time_dt)]= 0
)
```

This partitions the query by (*r, p*) and within each partition upserts all values from the time dimension. 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)
```

## 4 Spreadsheet Analysis And Optimization

The spreadsheet analysis determines the order of evaluation of formulas, prunes formulas whose results are fully filtered out by outer queries, restricts the formulas whose results are partially filtered, migrates predicates from outer queries into inner WHERE clause to limit the data processed by the spreadsheet, and generates a filter condition to identify the cells that are required throughout the evaluation of the spreadsheet formulas.

The analysis also determines one of two types of execution methods: one for acyclic and one for (potentially) cyclic formulas. Because of complex predicates in formulas, analysis cannot always ascertain acyclicity of formulas in the spreadsheet. Hence, we sometimes end up using the cylic execution method for acyclic spreadsheets which is expensive compared to the acyclic method.

**Formula dependencies and Execution Order**: The order of evaluation of formulas is determined from their dependency graph. Formula $F_1$ depends on $F_2$ (written $F_2$ -> $F_1$) if a cell evaluated by $F_2$ is used by $F_1$. For example in:

```
F1:s['video',2000]=s['tv', 2000]+s['vcr', 2000]
F2:s['vcr',  2000]=s['vcr',1998]+s['vcr', 1999]
```

$F_2$ -> $F_1$ as $F_1$ requires a cell s['vcr',2000] computed by $F_2$. To form the -> relation, for each formula *F* we determine cells that are referenced on its right side R(F) and cells that are modified on its left side, L(F). Obviously, $F_2$ -> $F_1$ iff R($F_1$) intersects L($F_2$). In the presence of complex cell references, like s['tv', $t^2+t^3+t^4< t^5$], it is hard to determine the intersection of predicates. In this case, we assume that the formula references all cells. This may result in over-estimation of the -> relation leading to spurious cycles in the dependency graph.

The -> relation results in a graph with formulas as nodes and their dependency relationship as directed edges. The graph is then analyzed for (partial) ordering.

A spreadsheet formula can access a range of cells (e.g., an aggregate - avg(s)['tv',*] or left side of an existential formula - s[*, *] = 10) and thus require a scan of data. If two formulas are independent (unrelated in the partial order derived from the graph), they can be evaluated concurrently using a single scan. To enable concurrent evaluation, formulas are grouped into enumerated levels such that each level contains independent formulas, and no formula in the level may depend on a formula in a higher level.

The path through the partial order with the maximum number of scans represents the minimum number of total scans possible, since they are all related by the partial order. If we have an acyclic graph (i.e., a partial order), then we can minimize the number of levels containing scans to this value. The following algorithm generates the levels such that number of scans is minimized - for a proof of a minimality please refer to the extended version of the paper.

Let G(F, E) be the graph of the -> relation where *F* are formulas and *E* the -> edges. We will call a formula with no incoming edges a *source* and formulas with only single cell references *single_refs* :

```
GenLevels(G) {
    LEVEL <- 1
    WHILE (F is not empty) {
      Find the set FS of all the SOURCES in F
      If (cycle is detected) {
        break the cycle /* see below */
      } else if (FS contains single_refs) {
        assign single_refs in FS to level LEVEL;
        F = F - single_refs in FS
      } else (FS contains only scans) {
        assign formulas in FS to level LEVEL;
```

(1)This initializes the measure x to "0" before the execution of spreadsheet and is similar to naming constants in ANSI views

```
        F = F - FS
      }
      LEVEL <- LEVEL + 1,
  }
}
```

For example,

```
SELECT * FROM f
GROUP BY p, t
SPREADSHEET DBY(p,t) MEA(sum(s) s)
(
  F1: s['tv', 2000] = sum(s)['tv',  1990<t<2000],
  F2: s['vcr',2000] = sum(s)['vcr', 1995<t<2000],
  F3: s['vcr',1999]=s['vcr',1997]+s['vcr',1998]
)
```

Here, the spreadsheet graph has one edge: F3 -> F2. The algorithm will assign the point reference F3 to level 1 and the scan F2 to level 2, but will delay assigning the scan F1 until level 2 so that F1 and F2 can share a single scan.

The *GenLevels* algorithm presented simplifies the cyclic case. Before generating the levels, the graph is analyzed for strongly connected components using algorithms in [17]. We can then isolate cyclic subgraphs from acyclic parts of the graph and from other cyclic subgraphs. This is important because the computational complexity of cyclic evaluation is proportional to the total number of rows updated or upserted in a cycle (see *auto-acyclic* algorithm in Section 5). After assigning levels to formulas a cyclic subgraph is dependent on, the cyclic subgraph can be broken by removing formulas from the subgraph and assigning them to individual levels in the same order until the subgraph is exhausted.

For spreadsheets with sequential order of evaluation, the dependency edges created always point from the earlier formula to the latter formula. A spreadsheet graph can therefore never be cyclic. We still generate levels in order to group the independent formulas together and hence, minimize the number of scans that are required for computation of aggregates and existential rules in the spreadsheet.

**Pruning Formulas**: We expect that, to encapsulate common computations, applications will generate views containing spreadsheets with thousands of formulas. Users querying these views will likely require only a subset of the result, putting predicates over the views. This gives us an opportunity to prune formulas that compute cells discarded by these predicates. For example:

```
SELECT * FROM
(SELECT r, p, t, s FROM f
 SPREADSHEET PBY(r) DBY (p, t) MEA (s) UPDATE
 (
 F1: s['dvd',2000]=s['dvd', 1999]*1.2,
 F2: s['vcr',2000]=s['vcr',1998]+s['vcr',1999],
 F3: s['tv', 2000]=avg(s)['tv', 1990<t<2000]
 ) v
)
WHERE p in ('dvd', 'vcr', 'video');
```

The evaluation of the formula F3 is unnecessary as the outer query filters out the cell that F3 evaluates. The above formulas are independent, which makes the pruning process simple. If, however, we had a formula that depends on F3, for example,

```
F4: s['video',2000]=s['vcr',2000]+s['tv',2000]
```

then F3 cannot be pruned as it is referenced by F4.

The evaluation of a formula becomes unnecessary when the following conditions are satisfied:

- The cells it updates are not used in evaluation of any other formula in the spreadsheet.
- The cells updated by the formula are filtered out in the outer query block or the measure updated by the formula is never referenced in the outer query block.

Identification of formulas that can be pruned is done by the following algorithm based on the dependency graph G. Let *sink* be a formula with no outgoing edge, i.e., one no other formula depends on.

```
PruneFormulas(G)
{
   Find a set FS of all SINKS
   WHILE (FS is not empty) {
     Pick a formula Fi from FS,
     FS = FS - {Fi}   /* remove Fi from FS */

     If (all the cells referenced on the left
        side of Fi are filtered out in the outer
        query block
             OR
       the measure updated by the left side
       of Fi is not referenced in the outer
      query block)
     {
       F = F - {Fi} /* delete Fi from list F */
       E = E - {all incoming edges into Fi},

       If deletion of F generates new 'sink'
       nodes, insert them into the set FS
     }
   }
}
```

**Rewriting Formulas.** Pruning formulas alone is not sufficient to avoid unnecessary computations during spreadsheet evaluation. In some cases, the results computed by a formula may be partially filtered out in the outer query block. Consider the following query which predicts the sale of all products in 2002 to be twice the cost of the same product in 2002, and then selects the sale and cost values for 'dvd' and 'vcr' for years > = 2000.

```
SELECT * FROM
(SELECT r, p, t, s FROM f
 SPREADSHEET PBY(r) DBY (p, t) MEA (s,c) UPDATE
 (
   F1: s[*,2002]=c[cv(p), 2002]*2,
 ) v
)
WHERE p in ('dvd','vcr') and t >= 2000;
```

The formula F1 cannot be pruned away as part of its result is needed in the outer query block. Still, we do not need to compute the s values for all products in 2002 as the outer query filters out all the rows except for products 'dvd' and 'vcr'. Hence we rewrite the left side of formula F1 as follows to avoid unnecessary computation:

```
F1': s[p in ('dvd','vcr'),2002]= c[cv(p), 2002]*2
```

The rewriting of formulas is done with a small extension of the

algorithm *PruneFormulas*. In the new *PruneFormulas*, we try to rewrite the formulas in all sink nodes that we cannot prune. Note that similar to pruning of a formula, the rewrite of a formula may also change the dependency graph (some incoming edges of the formula might be deleted) possibly leading to generation of new sink nodes, so it is only natural that both rewrite and pruning of formulas are handled in the same process.

Rewriting of formulas that are not sink nodes is also possible but the rewrite in that case is complicated as it is not only based on outer predicates, but also on the reference predicates of the other formulas that depend on the formula being rewritten.

**Pushing predicates through spreadsheet clauses**. Pushing predicates into an inner query block [15], [13] and its generalization 'predicate move-around' [12] is an important optimization and has been incorporated into queries with spreadsheets. We perform three types of pushing optimization: pushing on PBY and independent DBY dimensions, pushing based on bounding rectangle analysis and pushing through reference spreadsheets.

Pushing predicates through the PBY expressions in or out of the query block is always correct as they filter entire partitions. For example, in:

```
SELECT * FROM
(SELECT r, p, t, s FROM f
 SPREADSHEET PBY(r) DBY (p, t) MEA (s) UPDATE
  (
    F1:s['dvd',2000]=s['dvd',1999]+s['dvd',1997],
    F2:s['vcr',2000]=s['vcr',1998]+s['vcr',1999]
  )
) v
WHERE r = 'east' and t = 2000 and p = 'dvd';
```

we push the predicate r = 'east' through the spreadsheet clause into the WHERE clause of the inner query.

Pushing can be extended to independent dimensions. A dimension *d* is called an *independent dimension* if, for every formula, the value of *d* referenced on the the right side is the same as the value of *d* on the left side. For example, in the above spreadsheet, the left side of F1 refers to the same values of *p* as the right side. The same is true for formula F2 as well, thereby making *p* an independent dimension. *t*, however is not an independent dimension. Observe that in the absence of UPSERT rules, independent dimensions are functionally equivalent to the partitioning dimensions and can moved from the DBY to the PBY clause. For example, in the above spreadsheet, we could replace PBY/DBY clauses with

```
SPREADSHEET PBY(r, p) DBY (t) MEA (s) UPDATE
```

Consequently, we can push predicate p = 'dvd' into the inner query.

We also pull predicates on PBY and independent DBY expressions out of the query to effect predicate move around of [12].

The outer predicates on the DBY expressions which are not independent can be also pushed in but we need to extend them so they do not filter out the cells referenced by the right sides of the formulas. For each formula we construct a predicate defining the rectangle bounding the referenced cells. For example for F2 these predicates are *p='vcr'* and *t in (1998, 1999)* and for F1 *p='dvd'* and *t in (1997, 1999*)**.** Then a bounding rectangle for the entire spreadsheet is obtained using methods described in [8], [3] which is a union of bounding rectangles for each formula. This in our case is *p in ('vcr', 'dvd')* and *t in (1997, 1998, 1999)*. Then the predicates on DBY columns from the outer query are extended with the corresponding predicates from the spreadsheet bounding rectangle, and these are pushed into the query. In our example we extend the outer predicate *t = 2000* with *t in (1997, 1998, 1999)* which results in pushing *t in (1997, 1998, 1999, 2000)*. The predicates on DBY expressions in the outer query block are kept in place unless the pushdown filter is the same as the outer filter and there are no upsert formulas in the spreadsheet.

A challenging scenario arises when the bounding rectangle for a formula cannot be determined at optimization time since it may depend on a subquery *S* whose bounds are known only after S's execution. This is common in OLAP queries which frequently inquire about the relationship of a measure at a child level to that of its parent (e.g., sales of a state as a percentage of sales of a country), or inquire about a prior value of a measure (e.g., sales in March-2002 vs. sales the same month a year ago or a quarter ago). These relationships are obtained by querying dimension tables. For example, assume that the primary key of time dimension *time_dt* is month *m* and the table *time_dt* stores the corresponding month a year ago as *m_yago*, and the corresponding month a quarter ago as *m_qago*. Note that quarter ago means the same month in the previous quarter, so quarter ago of 1999-01 is 1998-10 *(*see Table 1)

**Table 1: Mapping between m and y_ago/m_qago**

| m | m_yago | m_qago |
|---|---|---|
| 1999-01 | 1998-01 | 1998-10 |
| 1999-02 | 1998-02 | 1998-11 |
| 1999-03 | 1998-03 | 1998-12 |

An analyst wants to compute for a product 'dvd' and months (1999-01, 199-03) the ratio of that month's sales to the sales in corresponding months a year and quarter ago respectively (*r_yago* and *r_qago).* Using SQL spreadsheet, this query is:

```
S1
SELECT p, m, s, r_yago, r_qago  FROM
 (SELECT p, m, s FROM f GROUP BY p, m
  SPREADSHEET
    REFERENCE prior ON
     (SELECT m, m_yago, m_qago FROM time_dt)
      DBY(m) MEA(m_yago, m_qago)
   PBY(p) DBY (m) MEA (sum(s) s,r_yago,r_qago)
  (
  F1: r_yago[*] = s[cv(m)] / s[m_yago[cv(m)]],
  F2: r_qago[*] = s[cv(m)] / s[m_qago[cv(m)]]
  ) v
)
WHERE p = 'dvd' and m IN (1999-01, 1999-03);
```

The reference spreadsheet serves as a one-dimensional look-up table translating month *m* into the corresponding month a year ago *m_yago* and a quarter ago *m_qago*. An alternative formulation of the query using ANSI SQL requires the joins *f* >< *time_dt* >< *f* >< *f*, where the first join gives the month values a year and a quarter ago for each row in fact table and the other two joins give the sales values in the same month, a quarter ago and an year ago

respectively. Thus, using reference spreadsheet, the number of joins is reduced to one.

The predicate p = 'dvd' on the PBY column can be pushed into the inner block. However, *m* is neither an independent dimension nor can bounding rectangles be determined for it as the values *m_yago* and *m_qago* are unknown. Consequently, a restriction on *m* cannot be pushed-in resulting in all time periods pumped to the spreadsheet out of which all except 1999-01 and 1999-03 are subsequently discarded in the outer query. Let's call a dimension *d* a *functionally independent dimension* if for every formula, the value of *d* referenced on the right side is either the same as the value of *d* on the left side or a function of the value of *d* on the left side via a reference spreadsheet. In query S1, *m* is a functionally independent dimension, as the right side uses *m* directly or uses a function of the value of *m* on the left side: *m_yago[cv(m)]* and *m_qago[cv(m)]*.

We experimented with three transformations to push predicates through functionally independent dimensions. In the first, called *ref-subquery pushing*, we add into the inner block a subquery predicate which selects all values needed by the spreadsheet and the outer query. The transform is similar to the magic set transformation [14] which pushes a query derived from outer predicates into the inner block. In the above case, the outer query needs *m* IN (1999-01, 1999-03), and the spreadsheet needs these values plus their corresponding *m_yago* and *m_qago* values from the reference spreadsheet. These values can be obtained by constructing a subquery over the reference spreadsheet:

```
S2
WITH ref_subquery AS
  (SELECT m, m_yago, m_qago FROM time_dt
   WHERE m IN (1999-01, 1999-03))
SELECT m AS m_value FROM ref_subquery
UNION
SELECT m_yago AS m_value FROM ref_subquery
UNION
SELECT m_qago AS m_value FROM ref_subquery
```

and then pushing it into the inner block of the query:

```
SELECT p, m, s, r_yago, r_qago  FROM
 (SELECT p, m, s FROM f
  WHERE m IN (SELECT m_value FROM S2)
  GROUP BY p, m
  SPREADSHEET
  <.. as above in query S1 .. >
)
WHERE p = 'dvd' and m IN (1999-01, 1999-03);
```

In the second transformation, called *extended pushing*, we construct the pushed-in predicates by executing the reference spreadsheet query, obtaining the referenced values and building predicates on the dimension, and finally disjuncting them with the outer predicates. In the above case we execute

```
SELECT DISTINCT m_yago, m_qago FROM time_dt
WHERE m IN (1999-01, 1999-03)
```

to obtain the values for *m_yago* and *m_qago* corresponding to *m* IN (1999-01, 1999-03). Let's assume that the corresponding *m_yago* is (1998-01, 1998-03) and *m_qago* is (1998-10, 1998-12) i.e., the first and third month of the previous quarter. Finally we push this predicate into the inner query:

```
SELECT p, m, s, r_yago, r_qago  FROM
 (SELECT p, m, s FROM f
  WHERE m IN (1999-01, 1999-03, /* outer preds */
              1998-01, 1998-03, /* prev year */
              1998-10, 1998-12) /* prev quart */
  GROUP BY p, m
  SPREADSHEET
  <.. as above in query S1 .. >
)
WHERE p = 'dvd' and m IN (1999-01, 1999-03);
```

In the third transformation, called *formula unfolding*, we transform the formulas by replacing the reference spreadsheet with its values. Similarly to the second transformation, we execute reference spreadsheet and obtain its measure for each of the dimension values requested by the outer query. These values are then used to unfold the formulas. For example, for *m* = 1999-01, value of *m_yago* = 1998-01, and *m_qago*= 1998-10, and for *m* = 1999-03, value of *m_yago* = 1998-03, and *m_qago*= 1998-12. Thus formulas are unfolded as:

```
SELECT p, m, s, r_yago, r_qago  FROM
 (SELECT p, m, s FROM f GROUP BY p, m
  SPREADSHEET
    REFERENCE prior ON
     (SELECT m, m_yago, m_qago FROM time_dt)
      DBY(m) MEA(m_yago, m_qago)
   PBY(p) DBY (m) MEA (sum(s) s,r_yago,r_qago)
  (
  F1:  r_yago[1999-01] = s[1999-01] / s[1998-01],
  F1': r_yago[1999-03] = s[1999-03] / s[1998-03],
  F2:  r_qago[1999-01] = s[1999-01] / s[1998-10],
  F2': r_qago[1999-03] = s[1999-01] / s[1998-12]
  ) v
)
WHERE p = 'dvd' and m IN (1999-01, 1999-03);
```

Following formula flattening we perform analysis of the bounding rectangles described above and push the resulting bounding predicate into the inner query.

In our experiments, see Section 6, the *extended pushing* and *formula unfolding* transformations, resulted in similar performance as in most cases they push-in the same predicates. In comparison, the *ref-subquery push* transform had inferior performance. The use of *ref-subquery* gives the optimizer a choice of picking either index nested loop join, hash join, or sort join between the subquery and the main query block. The optimizer is more likely to make mistakes with subquery predicates than with simple qualified index predicates. And, in the ref-subquery case, the optimizer sometimes picks up a wrong join method, thereby slowing down the query (see experimental results in Section 6 on page 10).

## 5 SQL Spreadsheet Execution

**Access structures**. For efficient access to single cells (like *s[p='dvd', t=2000]*), we build a two-level hash access structure. In the first level, data is hash partitioned on the PBY columns, and in the second level, a hash table is built on the PBY and DBY columns within each first level partition. The formulas are evaluated for one spreadsheet partition at a time. A spreadsheet partition contains all rows with the same PBY column values. Hence, one spreadsheet partition lies completely within one first level hash partition of the access structure. Therefore, if the second level hash tables of each

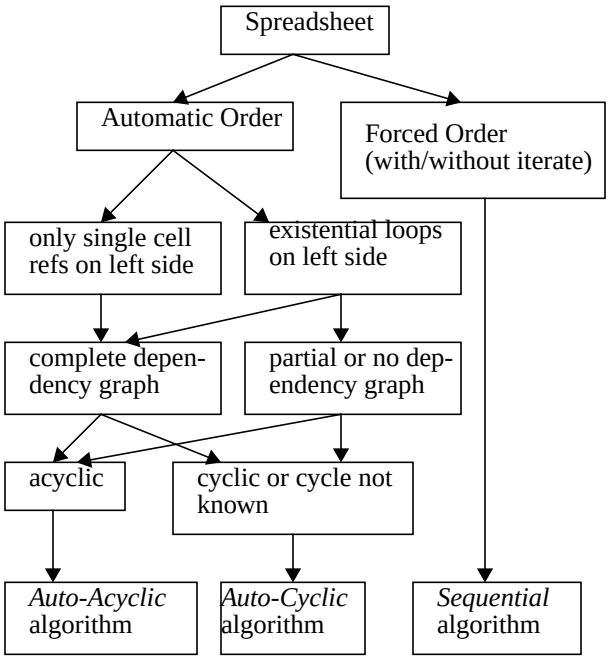
of the first level partitions fit in memory, we altogether avoid spilling to disk for evaluating the formulas. To further minimize size and build time of hash tables, we build this access structure only on rows required by the formulas as defined by the spreadsheet bounding rectangle (see Section 4).

Hash access structure supports operations like probe, update, upsert, insert and scan of all records within a spreadsheet partition. The hash access structure maintains records within a hash bucket clustered on PBY and DBY column values, thereby making the scan and probe on a spreadsheet partition efficient.

The number of first level partitions is chosen based on estimated size of data to be inserted into the access structure and the amount of available memory. The goal is to fit second level hash tables for each first level partition in memory. However, the spreadsheet partitions may be really big and we may not have enough memory to fit a partition. In such cases, we build a disk based hash table employing a weighted LRU scheme for block replacement,and pointer swizzling to make references lightweight.

**Execution.** Formulas in SQL spreadsheet operate in automatic order or sequential order. Figure 1 classifies the spreadsheet based on the evaluation order and dependency analysis and identifies the execution algorithm. There are three algorithms: *Auto-Acyclic*, *Auto-Cyclic* & *Sequential*.

Figure 1 **Classification of Spreadsheet**




**Automatic Order**. The order of evaluation of formulas in an automatic order spreadsheet is given by their dependencies (see Section 4 on page 4). We have two methods of its execution.

The ***Auto-Acyclic*** algorithm is taken when a complete and accurate dependency graph can be built and no cycles are detected in the dependency graph.

```
Auto-Acyclic()
{
  for each partition P in the spreadsheet
  {
    for level Li from L1 to Ln
    {
     /* LSi = set of formulas in Li with
      *       single cell refs on left side
      * LEi = set of formulas in Li with exist-
      *       ential conditions on left side
      *
      * First, evaluate all aggregates in set
      * LSi, then all formulas in that set
      */
      for each record r in P            - (I)
        for each aggregate A in LSi
          apply r to A;
      for each formula F in LSi
        evaluate F;

      /* Evaluate all formulas in LEi */
      for each record r in P            - (II)
      {
        find all formulas EF in LEi to be
          evaluated for r
        for each record r' in P         - (III)
          for each aggregate A in EF
            apply r' to A;
        for each formula EF
          evaluate EF;
      }
    }
  }
}
```

Notice that all the aggregates at any level are computed before evaluation of formulas at that level so they are available for the formulas. This requires a scan of records in the partition for each level. In the absence of existential formulas, and presence of only those aggregate functions for which an inverse is defined (for example, SUM, COUNT etc.), the aggregates for all the levels are computed in a single scan. And, with each formula we store a list of aggregates dependent on the cell being upserted (or updated) by it. It is possible to determine such a list because there are only single cell references on the left side. So, if a formula changes the value of a measure, the corresponding dependent aggregates are updated by applying the current value and inverse of the old value of the measure. In the above algorithm, we can also combine the scan (I) with the scan (II) or scan (III).

An example of an acyclic spreadsheet:

```
SELECT r, p, t, s
FROM f
SPREADSHEET PBY(r) DBY (p, t) MEA (s)
(
  s['tv', 2002] =s['tv', 2001] * 1.1,
  s['vcr',2002] =s['vcr', 1998] + s['vcr', 1999],
  s['dvd',2002] =(s['dvd',1997]+s['dvd',1998])/2,
  s[*,    2003] =s[cv(p), 2002] * 1.2
)
```

The above query makes sales forecasts for years 2002 and 2003. The formulas are split into 2 levels. The first level consists of the first 3 formulas, projecting sales for 2002, and the second level, dependent on the first level, consists of the last formula, projecting sales for 2003. The algorithm *Auto-Acyclic* evaluates formulas in the first level before evaluating formulas in the second level.

*Auto-Cyclic* Algorithm. There are also automatic order spreadsheets which are either cyclic, or have complex predicates that make the existence of cycles indeterminate. In such cases (Section 4 on page 4), the dependency analysis approximately groups the formulas into levels by finding sets of formulas comprising strongly connected components (SCCs -- the largest union of intersecting cycles), and assigning the formulas in an SCC to consecutive levels. The *Auto-Cyclic* algorithm evaluates formulas that are not contained in SCCs as in the acyclic case, but when formulas in SCCs are encountered, it iterates over the consecutive SCC formulas until a fixed point is reached, but only upto a maximum of 'N' iterations where N = number of cells upserted (or updated) in the first iteration. If the spreadsheet was actually acyclic, the formulas will converge after at most 'N' iterations. In the worst case, if the formulas were evaluated in exactly the opposite order of (real) dependency, each iteration will propagate one correct value to another formula, hence requiring 'N' iterations. Therefore, to evaluate all acyclic spreadsheets which could not be classified as acyclic and limit the number of iterations for cyclic spreadsheets, the maximum number of iterations for evaluation of formulas is fixed at 'N'. If the spreadsheet does not converge in 'N' iterations, an error is returned to the user. To determine if the spreadsheet has converged after an iteration, a flag is stored with the measure. This flag can be set whenever the measure is referenced while evaluating a formula. Later, update of a measure, which the flag set, to a different value indicates that additional iterations are required to reach a fixed point. Similarily an insert of a new cell (by an UPSERT formula) signals additional iterations. This technique will require resetting flags for each measure after each iteration - an expensive proposition. Hence, instead of a single flag, two flags are stored, each one being used in alternate iterations - as one of the flags is set, the other one can be cleared.

**Sequential Order**. In a sequential order spreadsheet, formulas are evaluated in the order they appear in the spreadsheet clause. The dependency analysis still groups the formulas into levels consisting of independent formulas so that the number of scans required for computation of aggregate functions is minimized. The algorithm is similar to Auto-Acyclic, but there may be multiple iterations as specified in the spreadsheet processing option - 'ITERATE'.

**Parallel Execution of SQL Spreadsheet.** We execute the spreadsheet scalably by evaluating the formulas over partitions in parallel. The data is distributed to Processing Elements (PE) based on the PBY columns so that each PE can work on partitions independent of other PEs. The distribution of partitions to PEs can be hash or range based on PBY columns. The work of PEs is coordinated by a single process called the *query coordinator*. For example, the following spreadsheet query can be evaluated by hash partitioning the data on r:

```
SPREADSHEET PBY(r) DBY(p, t) MEA(s)
(
  s['dvd', 2000] = 1.2*s['dvd', 1999]
)
```

In some cases, data partitioning on just the PBY columns limits the degree of parallelization and hence the scalability. That is the case when there are no PBY columns, or PBY columns have very small cardinality as in PBY(gender) where only two partitions ('male' and 'female') exist.

In such cases, we can include, in addition to PBY columns, some DBY columns for data partitioning. For example, the following query:

S3
```
SPREADSHEET PBY(r) DBY(p, t) MEA(s) UPDATE
(
  F1: s[*,2002]=
       avg(s)[cv(p), t in (1998,2000)],
  F2: s[*,2003]=
       avg(s)[cv(p), t in (1999,2001)]
)
```
is changed to
```
SPREADSHEET PBY(r, p) DBY(t) MEA(s) UPDATE
(
    s[2002] = avg(s)[t in (1998, 2000)],
    s[2003] = avg(s)[t in (1999, 2001)]
)
```
as formulas are independent of the dimension column *p*. This can be done for any independent dimension - see Section 4.

Higher granularity of partitioning results in better load balancing and processor utilization as parallelization takes place on both *r* and *p*. All PEs execute the same set of formulas but with different data sets. If the column corresponding to an independent dimension on the left side doesn't qualify all values of that dimension, (i.e., in our notation is not a "*"), it will not be possible to promote the independent dimension to PBY. This is because promoting to PBY would incorrectly update all values for that dimension. Instead, we will duplicate the independent dimension in the PBY clause as well to get better parallelization. For example:

S4
```
SPREADSHEET PBY(r) DBY(p, t) MEA(s) UPDATE
(
  s[p != 'bike',2002]= avg(s)[cv(p),t<2001]
)
```
is rewritten by optimizer for parallelization to:
```
SPREADSHEET PBY(r, p) DBY(p, t) MEA(s) UPDATE
(
  s[p != 'bike',2002]= avg(s)[cv(p),t<2001]
)
```
Complex scenarios exist where different PEs need to execute different sets of formulas. One such scenario is the presence of UPSERT option as in:
```
SPREADSHEET PBY(r) DBY(p, t) MEA(s) UPSERT
(
  F1:  s['dvd',2002] = sum(s)['dvd', t<1999]
             + avg(s)['dvd',1999<=t<= 2001],
  F2:  s['vcr',2002] = avg(s)['vcr', t <= 2001],
  F3:  s[*, 2003] = 1.2*s[cv(p), 2002]
)
```
In this spreadsheet, *p* is again an independent dimension. But because of the UPSERT option, using *p* as a partitioning column with all PEs evaluating the same set of formulas can lead to an incorrect result. Assume that products 'dvd' and 'vcr' get assigned to different PEs: PE1 and PE2 respectively. If the same formulas are executed by both PEs, the result would have spurious rows - for example, PE1 working on product 'dvd' would introduce a row for 'vcr' while this row might already exist in the data set passed to

PE2. In such cases, spreadsheet formulas need to be grouped and assigned to PEs based on data distribution so that the formula F is assigned to PE iff PE is processing data which F touches. For example, PE1 might evaluate formulas F1 and F3 while PE2 evaluates formulas F2 and F3.

This process of grouping formulas cannot be done at compile time as it is data dependent. Instead this is done by passing an extra condition to each PE, indicating the data set for which the formulas should be evaluated. In this case, if data is distributed to PEs by HASH partitioning, the extra condition is of the form:

```
WHERE HASH(p) = hash_value_of_P_for_this_PE.
```

The value for *hash_value_of_P_for_this_PE* is passed to each PE by the query coordinator. Then each PE, before evaluating a non-existential formula, i.e., one which explicitly qualifies all dimensions would find the value for *p* and verify that the triggering condition holds. If so, the formula is executed, otherwise it is skipped. The HASH(p) is the hash function used for data partitioning. For existential formulas we do not need to evaluate the condition as they never generate new rows operating only in the update mode. So in the above spreadsheet assume that *HASH*('dvd') = 1 and *HASH*('vcr') = 2, and PE1 and PE2 operate in hash partitions 1 and 2 correspondingly. Then PE1 evaluates F1 and F3 while PE2 evaluates formulas F2 and F3.

## 6 Experimental Results

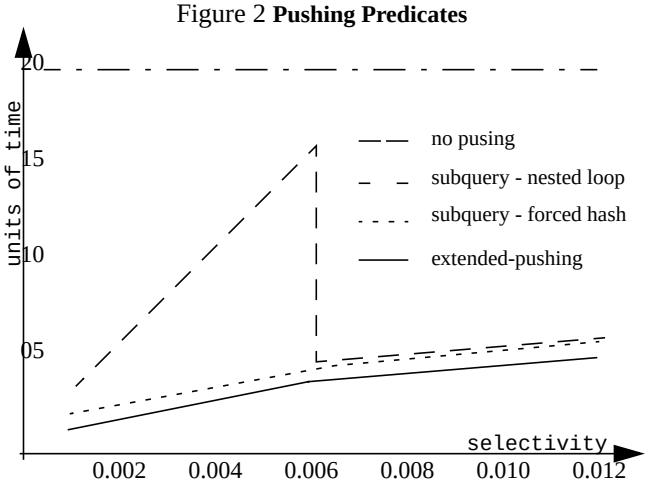
We conducted experiments on the APB benchmark [1] populated with 0.1 density data. The APB schema has a fact table with 4 hierarchical dimensions: channel with 2 levels, time with 3 levels, customer with 3 levels and product with 7 levels. We constructed a cube over the fact table and materialized it in *apb_cube* table. Similar to the fact table, the cube has 4 dimensions - *t*(ime), *p*(roduct), *c*(ustomer), *h*(channel), each represented as a single column with all hierarchical levels encoded into a single value. The cube had bitmap indexes on the dimensions and had 22,721,998 rows. The experiments were conducted on a 12 CPU, 336 Mhz, shared memory machine with a total of 12 GB of memory. Per commercial product, only relative units of time are reported.

**Pushing predicates experiment**. We used a spreadsheet query which calculates ratio of sales for every product level to its 1st, 2nd and 3rd parent in the product hierarchy. APB product hierarchy has 7 levels: *prod, class, group, family, line division*, and *top*. Thus for a product in the *prod* level, we calculate the share of its sales relative to its corresponding *class, group* and *family* levels. Assuming that the parent information of a product is stored in a dimension table *product_dt* with columns *p*, *parent1*, *parent2*, *parent3* (product, its parent, grand parent and great-grand parent respectively), the query has the form:

```
S5
  SELECT
    s, share_1, share_2, share_3, p, c, h, t,
  FROM
    apb_cube
  SPREADSHEET
    REFERENCE ON
       (SELECT p, parent1, parent2, parent3
        FROM product_dt)
       DBY (p) MEA (parent1, parent2, parent3)
    PBY (c,h,t)  DBY (p)
    MEA (s, 0 share_1, 0 share_2, 0 share_3)
  RULES UPDATE
  (
  F1:  share_1[*] = s[cv(p)] / s[parent1[cv(p)]]
  F2:  share_2[*] = s[cv(p)] / s[parent2[cv(p)]]
  F3:  share_3[*] = s[cv(p)] / s[parent3[cv(p)]]
  )
```

The analyst indicates products of interest via a predicate on *p* in the outer query. We studied three algorithms (namely the *subquery, extended-pushing* and *formula-unfolding*) for pushing predicates by changing the selectivity (fraction of rows selected) of the predicate.

Figure 2 **Pushing Predicates**




As shown in Figure 2, we observed 5 to 20 times improvement in the query response time (serial execution) by pushing predicates as compared to not pushing them at all. In general, the improvement can be arbitrarily large. The *extended-pushing* and *formula-unfolding* algorithms performed almost identically as expected and their response times were predictable (Figure 2 shows only the former). The *subquery pushing* algorithm offered a surprise as the response time curve was not smooth. For low selectivity of the predicates (up to 0.006) the optimizer chose nested loop join between the subquery and the *apb_cube (*see the *subquery-nested loop* curve*)*. This was not the optimal choice and caused linear degradation in performance up to 3 times over the extended-pushing method. Beyond the 0.006 selectivity, the optimizer chose more optimal hash join. However, the response time was still 20% worse than the response time for the extended-pushing method. When we forced the optimizer to always chose hash join between the subquery and *apb_cube (*see *subquery-forced hash* graph*)*, the response time for the subquery method was about 20% worse than extended-pushing for the entire range of investigated selectivities.

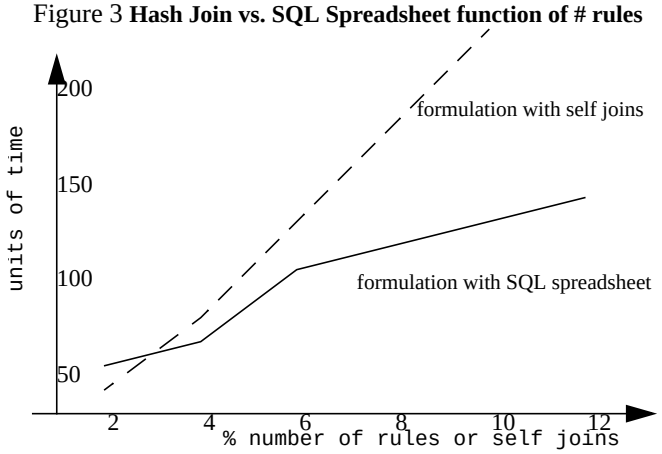
**Hash-Join vs. SQL Spreadsheet experiment**. Many SQL Spreadsheet operations can be expressed with standard ANSI SQL using joins and UNIONs. For example, query S5 can be expressed using joins three self joins of *abp_cube* and a join to *product_dt*:

```
SELECT
   s, a1.s/a2.s AS share_1, a1.s/a3.s AS share_2,
   a1.s/a4.s AS share_3, p, c, h, t,
FROM
   apb_cube a1, apb_cube a1, apb_cube a3,
```

```
   apb_cube a4,  product_dt p
WHERE
 a1.p=p.p &
 a2.p=p.parent1 & a2.c=a1.c & a2.h=a1.h & a3.t=a1.t
 a3.p=p.parent2 & a3.c=a1.c & a3.h=a1.h & a3.t=a1.t
 a4.p=p.parent3 & a4.c=a1.c & a4.h=a1.h & a4.t=a1.t
```

The number of self joins is equal to the number formulas (say *N*) and all joins to the original *apb_cube* (*a1*) are right outer joins. For hash joins this requires construction of *N* hash tables while our SQL Spreadsheet needs only one hash access structure per spreadsheet. Consequently there is a break even point $N_i$, when the cost of the spreadsheet access structure is amortized, and spreadsheet outperforms ANSI hash-join formulation as shown in Figure 4. In the above query, $N_i$, is 3 (i.e, 3 rules). Above 14 rules, spreadsheet execution is twice as fast as that using joins. In the experiment joins and spreadsheet were processed serially and the access structures for both fit in memory.

Figure 3 **Hash Join vs. SQL Spreadsheet function of # rules**




**Access Method - Hash Table.** We tested the scalability of our execution methods as a function of the number of formulas, and memory available for the hash structure.

Figure 4 **Scalability with number of formulas**

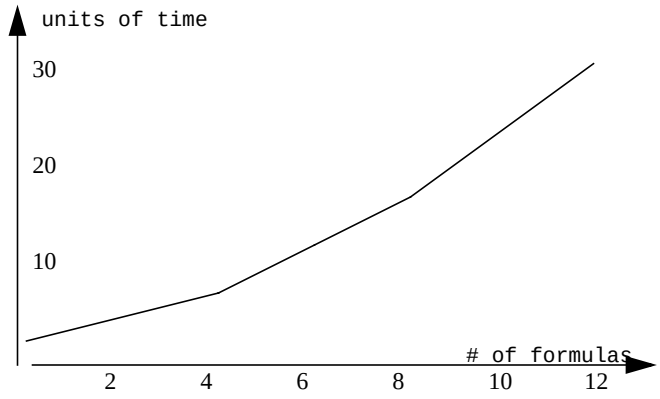



Figure 4 shows an almost linear scalability between the response time of a spreadsheet and the number of formulas. Each formula came from query S5, and simulated a double join *apb_cube* >< *product_td* >< *apb_cube*. In the experiment, the physical memory was large enough to accommodate every individual partition of the *apb_cube* which in our case was a maximum of 15MB - about 20% of the cube. The formulas were processed in parallell (12 processors) with close to linear (about 80%) parallel efficiency.

Figure 5 **Scalability with size of physical memory**

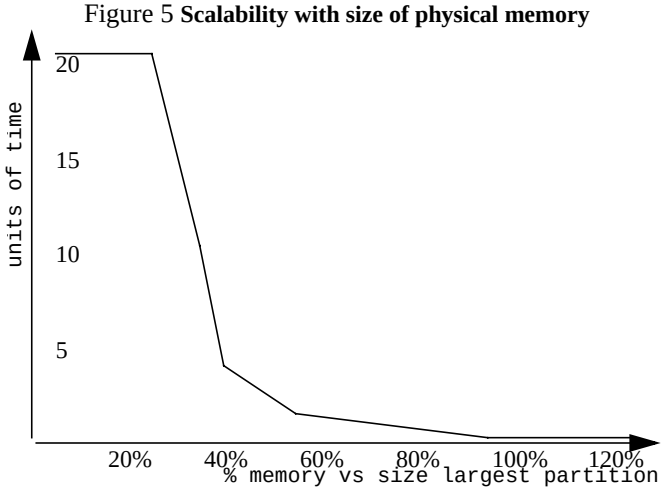



Figure 5 shows the performance of our access structure as a function of available memory. The memory size is expressed as a percentage of the size required to fit the largest partition of data in the hash access structure in physical memory. Recall from Section 5, that we first partition the data on the PBY columns, and process one partition at a time to execute the formulas. In the experiment we executed a single formula, F1, from query S5:

```
F1:  share_1[*] = s[cv(p)] / s[parent1[cv(p)]]
```

The formula accesses, within each PBY (`c,h,t`) partition, sales for a product and its parent. If a partition does not fit in memory we incur an I/O if a referenced cell is not cached. In a severe case of memory shortage, each reference may be a cache miss, reducing our access method to an un-cached, nested loop join. In the case of formula F1 which references a product and its parent, this occurs when the available memory is less than 30% of the largest partition - see Figure 5. Thus our method works very well and outperforms equivalent simulations of formulas with joins (for hash, sort and nested loop join methods) when the PBY partitions fit in memory as in those cases, we reduce the number of required joins. Note that the equivalent simulations must preform *abp_cube* >< *product_dt* >< *abp_cube*, while with spreadsheet we effectively build access structure for only one join *abp_cube* >< *product_dt.* For extreme cases of memory shortage, we degrade to the equivalent performance of simulation with nested loop joins. Observe that in these cases, hash join simulations would not perform better as they would have to spill to disk all of its data.

## 7 Conclusions and future research

This paper extends SQL with a computational clause which allows us to treat a relation as a multi-dimensional array and specify a set of formulas over it. The formulas replace multiple joins and UNION operations which must be performed for equivalent computation with current ANSI SQL. This not only allows for ease of programming, but also offers the RDBMS an

opportunity to perform better optimizations as there are fewer complex query blocks to optimize - an Achilles heel of many RDBMSs. We also create a single run time access structure which replaces multiple hash or sort structures needed for equivalent joins and UNIONs. Our intent is an eventual migration of computations from classical spreadsheets into the RDBMS. Such migration would offer an unprecedented integration of business models which are currently distributed among thousands of incompatible and incomparable spreadsheets. In our model, the result of an SQL Spreadsheet is a relation with well defined semantics and can easily be compared to other SQL spreadsheets via joins, unions, and other relational operations. The SQL Spreadsheet can be stored in a relational view and hence, become known to tools through the RDBMS catalog, thereby enhancing their cooperation.

There are several topics we are now investigating:

**Materialized Views.** A SQL Spreadsheet can be stored in a materialized view [4], [5], [7] and providing incremental refresh on this view would offer an automatic what-if analysis. Modification to detail data would be incrementally propagated through the formulas allowing us to observe the change. A rollback operation would remove it. A significant performance improvement could be achieved if a query with an SQL Spreadsheet were rewritten with an MV containing another spreadsheet. In general this is an undecidable problem; however, there are practical restrictions on the formulas which make the problem solvable.

**Parametric Models.** ANSI SQL doesn't provide a good separation between data and computation. ANSI SQL views, which could store SQL Spreadsheet, do not allow us to pass data to the formulas during view invocation. We are working on extending the SQL View model, such that tables or subqueries could be passed as parameters to the SQL Spreadsheet formulas. Conversely, SQL Spreadsheet could be passed to views as subqueries with an important advantage of performing dynamic optimizations (see Section 4).

**Automatic Migration of Classic Spreadsheet to the RDBMS.** We would like to support all the classical spreadsheet functions in the RDBMS so that it is easy to migrate classic spreadsheets into the RDBMS. Another problem is that Classic Spreadsheets are very unstructured with data intermixed with formulas and the latter expressed in unreadable row-column references, which makes user assisted translation necessary.

**Access Models.** Our initial implementation of the access method was based on a B-tree supported by our RDBMS. This proved more expensive than the current hash table mostly due to code path length for the B-Tree. We are investigating a very light B-Tree structure that could be useful as an alternate access method for formulas which need ordering - see the ORDER BY formula clause in Section 2. We are also investigating methods of reducing IO when a hash partition cannot fit in the available memory, and the formulas resemble these in query S5, i.e., scenarios in Figure 5. In many of these cases there is a correlation between dimension values on the left and right side, so re-clustering or ordering the data using existing hash or sort merge join methods would reduce physical IO.

## 8 References


[1] APB Benchmark Specifications. http://www.olapcouncil.org/research/APB1R2_spec.pdf

[2] A. Balmin, T. Papadimitriou, Y. Papakonstantinou. "Hypothetical Queries in an OLAP Environment," In *Proceedings of the 26th VLDB Conference*, Cairo, Egypt, 2000.

[3] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger, "The R*-tree: An efficient and Robust Access Method for Points and Rectangles," In *Proc. 1990 ACM SIGMOD Conf.*

[4] R. G. Bello, et al, "Materialized Views In Oracle", *Proceedings of VLDB'98, New York, USA, 1998*

[5] J. A. Blakeley, P. Larson, and F. W. Tompa. "Efficiently Updating Materialized Views," *Proceedings of ACM SIGMOD 1986*

[6] D.Chatziantoniou and K.Ross, "Querying Multiple Features of Groups in Relational Databases, " *Proceedings of VLDB'96*.

[7] A. Gupta, I.S. Mumick, and V. S. Subrahmanian. "Maintaining views incrementally". *Proceedings of ACM SIGMOD 1993 International Conference on Management of Data*, Washington DC, 1993.

[8] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. 1984 ACM SIGMOD Conf.*

[9] Joseph M. Hellerstein. "Practical predicate placement," *Proceedings of ACM SIGMOD 1994 International Conference on Management of Data*, 1994.

[10] J. M. Hellerstein and M. Stonebraker. "Predicate migration: Optimizing queries with expensive predicates," *Proceedings of ACM SIGMOD 1993, Washington DC, 1993*.

[11] L. Lakshamanan, J. Pei, Y. Zhao, "QC-Trees. Efficient Summary Structure for Semantic OLAP", *Proceedings of ACM SIGMOD 2003, San Diego, CA 2003*.

[12] A. Y. Levy, I. S. Mumick, and Y. Sagiv. "Query optimization by predicate move-around," *Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994.*

[13] K. Ross, D. Srivastava, P. Stuckey, and S. Sudarshan. "Foundations of aggregation constraints," *Alan Borning, editor, Principles and Practice of Constraint Programming, 1994.*

[14] I. S. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. "Magic is relevant," *Proceedings of the ACM SIGMOD 1990*, Atlantic City, New Jersey, 1990

[15] D. Srivastava and R. Ramakrishnan. "Pushing Constraint Selections," *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, San Diego, CA, 1992.

[16] Y. Sismanis, N. Roussopoulos, A. Deligiannakis, Y. Kotidis, "Dwarf: Shrinking the Petacube", *Proceedings of ACM SIGMOD* 2002, Madison, WI,2002.

[17] R. Tarjan. "Dept-first search and linear graph algorithms," *SIAM J. Computing, 1997.*

[18] F. Zemke. "Rank, Moving and reporting functions for OLAP," *99/01/22 proposal for ANSI-NCTS.*