Query By Excel

Andrew Witkowski, Srikanth Bellamkonda, Tolga Bozkaya, Aman Naimat, Lei Sheng,
Sankar Subramanian, Allison Waingold.
Oracle, 500 Oracle Parkway, Redwood Shores CA 94065
Firstname.Lastname@oracle.com

Abstract

Spreadsheets, and MS Excel in particular, are established analysis tools. They offer an attractive user interface, provide an easy to use computational model, and offer substantial interactivity for what-if analysis. However, as opposed to RDBMS, spreadsheets do not provide a central repository hence they do not provide shareability of models built in Excel and lead to proliferation of multiple copies of the same spreadsheet. Furthermore, spreadsheets do not offer scalable computation, for example, they lack parallelization. To address the shareability, and scalability problems, we propose to automatically translate Excel computation into SQL. An analyst can import the data from a relational system, define computation over it using familiar Excel formulas and then translate and store it as a relational SQL view over the imported data. The Excel computation is then performed by the relational system. To edit the model, the analyst can bring the model back to Excel, modify it in Excel and store it back as an SQL View. We refer to this system as *Query by Excel*, QBX in short.

1 Introduction

Spreadsheets, MS Excel [5],[6] in particular, are established business and personal analysis tools. They offer an attractive user interface with graphs and customizable menus, provide an easy to use computational model, and offer very substantial interactivity for "what-if" analysis. Spreadsheets offer many financial, statistical, engineering and mathematical functions as well as data transformation services like pivot, aggregation, lookups, etc.

Spreadsheets as computational machines, however, have serious shortcomings. They lack a well defined algebra and their computation is cryptic. A scalability problem exists when the data set is large. Also, spreadsheets offer a fragmented, unconsolidated picture of a business with data residing in separate sources, like RDBMS, and formulas describing the business in a spreadsheet.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fe and/or permission for the Endowment

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

On the other hand, there are existing computational engines without scalability or fragmentation problems and with a well-defined computational algebra, for example, OLAP [7], [8], Statistical [12] and Relational engines. They are, however, at a disadvantage in interactivity, graphical presentation and popularity of the computational language.

This paper discusses a system that combines the presentational and interactive modeling power of Excel and the computational power and scalability of an established Relational Engine with Analytical Extensions. This system is called Query By Excel (QBX) and has these features:

- Analysts build and edit their model using familiar Excel formulas. The model is then automatically translated into SQL and stored as a set of publicly available relational views.
- Analysts designate areas in the spreadsheet that are relational sources, called *RTables*. The area contains an image of (a sample of) a relational table. An RTable can be transformed into another RTable using Excel operations corresponding to Outer Join, Selection, Projection and Aggregation. Hence, users can perform Relational operations with Excel without writing a SQL.
- The analyst writes Excel formulas on samples of tables that fit in an Excel spreadsheet. However, when operations are translated to SQL they operate on entire tables, hence applying the scalability of the RDBMS (in size and parallelism) to Excel models.
- To import relational data to Excel, users currently use Relational Query Builders [10], [11] that require some knowledge of SQL. QBX disposes of them as query building is done entirely with Excel formulas.
- Business Reporting tools [9] can access the Relational Views of translated Excel spreadsheets. Excel consolidation becomes as easy as combining these views using SQL operations like Join, Union, etc.

This paper was motivated by new SQL Analytic Extensions: SQL Model [1],[2] and SQL Pivot Operator [3] which provide language bindings to express Excel formulas in efficient SQL.

This paper is organized as follows. Section 2 briefly describes SQL extensions used for Excel translation. Section 3 describes the high level architecture and relational schemas. Section 4 discusses translation of Excel formulas to SQL and Section 5 presents its optimizations. Section 6 shows performance of our translated models. Section 7 concludes and suggests topics for further research.

2 SQL Extensions

This section briefly describes the SQL Model [1] [2] extension (available in Oracle 10g Release) used in our Excel to SQL Translation. Our examples are based on a star schema with three dimension tables: *time_dim*, *prod_dim*, *region_dim* and a fact table *f(t, r, p, s, c)*. *f* is dimensioned by time (*t*), region (*r*), and product (*p*) columns and has two measures: sales (*s*) and cost (*c*).

[1] introduced an SQL extension for analytical processing called SQL Spreadsheet Clause, later renamed SQL Model. This extension allows users to view a relation as a multidimensional array and specify multiple formulas over it. The SQL Model clause identifies partition, dimension and measure columns within the query result. The partition (PBY) columns divide the relation into disjoint subsets. The dimension (DBY) columns uniquely identify rows within each partition, called *cells*. The measure (MEA) columns are expressions computed by the model. A sequence of formulas describing computation on cells, follows the PBY, DBY and MEA definitions. The structure of the SQL Model is:

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

Cells are referenced using an array notation with dimension columns qualified by predicates, for example s[p='dvd', t=2002] or s['dvd', 2002] for short. A formula represents an assignment of expressions over measures to the target cells. For example:

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

partitions table *f* by region r and defines that within each region, sales of 'vcr' in 2002 will be the sum of sales in 2000 and 2001, and sales of 'tv' will be the average for years between 1992 and 2002. The left side of a formula can define a range of cells and the new function *cv()* carries the value of a dimension from the left side to the right side thus serving as a join between the right and left sides. For example:

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

states that sales of every product in the 'west' region for year > 2001 will be 20% higher than sales of the same product in the preceding year.

[1] also introduces read-only Reference Models which are n-dimensional arrays defined over other query blocks. They are used as lookup tables in the main SQL Model clause. For example, assume a budget table *budget(r,p)* containing predictions *p* for the 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
MODEL
  REFERENCE budget ON (SELECT r, p FROM budget)
  DBY(r) MEA(p)
DBY (r, t) MEA (sum(s) s)
(
  s['west',2002]=budget.p['west']*s['west',2001]
)
```

The evaluation of formulas can be done in the order of their dependencies or in the lexicographical order of their specification referred to as AUTOMATIC or SEQUENTIAL order, respectively. Consider the following SQL Model clause:

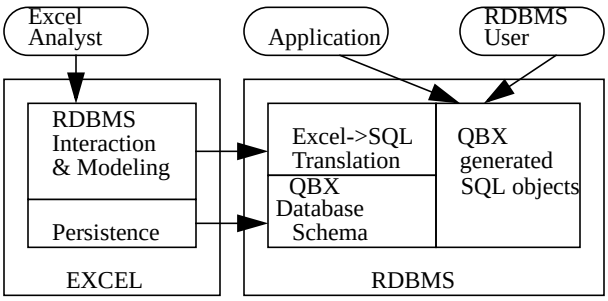
```
MODEL PBY(r) DBY (p,t) MEA (s) AUTOMATIC ORDER
( s['dvd',2002] = s['dvd',2000] + s['dvd',2001]
  s['dvd',2001] = 1000
)
```

Here, the first formula depends on the second and consequently we will evaluate the second one first. SQL Model also provides iterative execution of rules with termination conditions, and options to treat NULL values in numeric expressions as 0, etc., which are used in QBX.

3 Architecture of QBX

Our system, Query By Excel, aims at making Excel Spreadsheet a front end to Relational Databases. Figure 1 shows the system's components and their interactions. The QBX Schema stores a representation of a spreadsheet's data, formulas, layout, and RTables. The RDBMS interaction and modeling component allows the user to manage RTables. The Persistence component stores information about a spreadsheet to the QBX schema. The Excel to SQL Translation translates a spreadsheet into an SQL view. There are two categories of users in this system: the Excel analyst uses Excel to design models on the underlying transactional data; RDBMS users and applications consume the models created by the Excel analyst.

Figure 1 **Architecture of QBX**



The Database Schema Component stores Excel two-dimensional grid and relational objects represented in a set of relational tables:

Excels(eid, name, owner, ExcelBinary, SQLView) records accounting information about stored Excel spreadsheets including their name, internal Excel id (*eid*), owner, and the Excel xls file in a lob column (*ExcelBinary*). It also stores the name of the SQL View containing SQL translation of Excel in *SQLView* column. This view is available to public.

Cells(eid, sheet, row, col, x, f) stores cells of an Excel spreadsheet. It records a cell's coordinates (*sheet, row, col*),

and either its value (*x*), or its formula (*f*). Cells that are not populated are not stored. We store values as strings and rely on SQL for implicit type conversion.

RTables (eid, RTable, sheet, row, col, sample, RTableView, ..) stores, for each Excel, the name of each RTable imported into Excel (*RTable*), its location within Excel *(sheet, row, column)*, parameters (e.g. *sample* size), name of the SQL View representing the RTable (*RTableView*), and other accounting information like column types. The SQL Text representing RTable is stored in the public catalog.

To illustrate, consider the following Excel spreadsheet

Figure 2 **Example1**

	A	B	C	D	
1			sale	diff	
2			10.00		
3			12.00	=C3-C2	

When this Excel spreadsheet is persisted, we create a single entry in the *Excels* table, and then store five cells C1, C2, C3, D1 and D3 to the *Cells* table with their Excel <sheet, row, col> coordinates. Cells C1, C2, C3, D1 have constants (in the *Cells.x* column) and cell D3 has a formula (in the *Cells.f* column). This Excel spreadsheet does not contain RTables, hence RTables is empty. *Excels* also contains the name of an SQL view, which represents computations done by this spreadsheet.

QBX generated SQL Objects store views representing Excel computation in public dictionary. For the example above, we store the following view:

```
Q1
CREATE VIEW Example1 AS
SELECT sheet, row, col, x FROM cells
MODEL DBY (sheet,row,col) MEA (x) AUTOMATIC ORDER
(
  x[1,3,4] = x[1,3,3] - x[1,2,3]  -- D3 = C3-C2
)
```

The RDBMS Interaction and Modeling component is an Excel add-on written in VBA. This component exposes a menu-driven interface; a main menu called QBX allows us to manage RTables and to store, translate and load Excel to and from the database. It has two major sub-menus:

1. QBX->RTables menu manages RTables. It allows us to load a relational table (menu item *LoadTable*), add and drop columns from it (*Add Column/Drop Column*), save a transformed RTable as a relational view (*SaveTable*), and save regions of Excel spreadsheets as Relational Views (*SaveRegion*).
2. QBX->Spreadsheet menu. *Store* translates an Excel spreadsheet into SQL and stores it in the database together with the original Excel spreadsheet. *Load* loads a previously stored Excel spreadsheet for editing.

This component invokes the Translation and Persistence components for the bulk of its work.

The Persistence Component persists information about RTables, their locations in Excel, SQL views representing them, populated cell values and formulas in the RDBMS. This component is implemented as a VBA Excel add-on, using Objects for OLE. These steps are followed to persist an Excel spreadsheet:

1. A new Excel ID (eid) is created and the Excel xls file is stored in the *Excels* table together with its name, etc.
2. The location, and the formula or the value of each populated cell that does not belong to an RTable is stored in the *Cells* table.
3. The location of each RTable is stored. The SQL view representing the RTable is also stored.
4. The Excel to SQL translator (described in Section 4) is invoked and the resulting relational view, representing the Excel computation, is stored in the RDBMS public catalog. Its name is recorded in *Excels* table.

The Translation Component performs two types of translations: it translates Excel formulas into an SQL view over the Cells table; and over views representing RTables. The component runs in the RDBMS as a Java stored procedure. The translation is described in detail in Section 4.

4 Excel to SQL Translation

In QBX we try to minimize extensions to Excel. We used advanced Excel capabilities, like pivot functions, advanced filters and operations on named ranges, to simulate relational operations like aggregation, selections, and set functions, respectively.

We recognized that Excel supports functions not yet represented in SQL, like financial functions, and decided not to translate Excel spreadsheets containing them. Our goal was to translate a subset of Excel functions which have correspondents in SQL. Many Excel functions could be implemented as simple SQL row functions using PL/SQL. Others, like SUMPRODUCT that takes a variable number of ranges, could be translated to SQL using aggregates and collections, and these are left for future extensions.

We distinguish three types of Excel to SQL translations. Fixed frame translation (Section 4.1) handles self-contained Excel computation, i.e., formulas operating on embedded (i.e., non-imported) Excel data. The Table Translation (Section 4.2) uses Excel operations on imported relational tables to simulate parts of relational algebra. The Unified translation allows Excel computations on RTables (Section 4.3 and Section 4.4).

4.1 Fix Frame Translation.

The *fix frame translation* is based on the observation that Excel formulas operate on a fixed grid dimensioned by sheet, row, and col(umn). Within this grid, the formulas compute cells as functions of other cells or cell regions using Excel expressions and functions. The fix frame computation translates well into the SQL Model as it can define SQL formulas on an array dimensioned by Excel grid: (sheet, row, col). A few exceptions are noted below.

Each cell in Excel is stored in the *Cells* table, which keeps the location, the value, and the formula of every populated cell. The Excel spreadsheet is then represented by an SQL view over the *Cells* table that computes the value of the cells. The view contains an SQL Model dimensioned by Excel coordinates: (sheet, row, col) with measure x representing the value of the cells. Excel coordinates are expressed as

array indexes, e.g, Sheet1!A1 has (1,1,1) coordinates. For cells containing formulas, SQL Model's update rule computes their values.

For example, consider the following Excel:

	A	B	C
1	1	=A1+A2	
2	2	=A3+1	

When the Excel spreadsheet is persisted, four cells A1, B2, B1 & B2 are stored in the *Cells* table. Constants A1 and A2 do not generate update rules; B1 and B2 are represented with two update rules:

```
SELECT sheet, row, col FROM cells
MODEL DBY (sheet,row,col) MEA (x) AUTOMATIC ORDER
(    x[1,1,2] = x[1,1,1] + x[1,2,1],-- B1=A1+A2
     x[1,2,2] = x[1,3,1] + 1        -- B2=A3+1
)
```

Observe that SQL Model uses the AUTOMATIC ORDER option, which guarantees that formulas are processed in their dependency order, similar to Excel. We do not show this option and the sheet coordinate in the following examples for brevity.

Basic Fix Frame Translation. Figure 3 lists the rewrite rules for our translation. A recursive rewrite function *R* takes an Excel formula and converts it to an SQL Model formula. We use these rewrite rules:

- *cref* - translates a cell reference e.g. R(Sheet1!A1) = x[1, 1, 1].
- *crange* and *crange_in_list* - translates a cell range argument of an Excel function. For an Excel aggregate, we translate the Excel range into a relational range. For example, R(sum(A1:A10)) = sum(x)[1, 1<=row<=10, 1]. In some cases, the range is expanded into a list of cells (e.g. "A1:A3" expands to "A1,A2,A3").

There are three rewrite rules for Excel functions:

- *faggregate*: an Excel aggregate is translated to an SQL aggregate over a set of rows.
- *fscalar*: an Excel scalar function is translated to its SQL correspondent.
- *foperator*: some Excel functions are translated to infix SQL operators, for instance, PRODUCT() translates to the infix *.

Unfortunately, Excel has functions that are not available in SQL and these terminate translation with a failure.

The following helper functions are used in rewrite rules:

- *sqlFcn(excelFcn)* - translates Excel to SQL function.
- *op(excelFcn)* - translates Excel function to corresponding SQL infix operator.
- *sheet(c)*, *row(c)*, *col(c)* - translates Excel coordinates of a cell *c* to 1-based offsets.

Figure 3 **Rewrite Rules for Fix Frame Translation**

```
R( e1 op e2 ) = R(e1) op R(e2)        -- operator

R( const ) = const                    -- constant

R( cell ) =
  s[sheet(cell), row(cell), col(cell)]  -- cref

R( cell1 : cell2 ) =
  s[ sheet(cell1),
     row(cell1)<=row<=row(cell2),
     col(cell1)<=col<=col(cell2) ]     -- crange

R( fcnAgg ( cell1 : cell2 ) ) =
  sqlFcn(fcnAgg) (s) [ sheet(cell1),
     row(cell1)<=row<=row(cell2),
     col(cell1)<=col<=col(cell2) ]-- faggregate

R( fcnOp ( e1 , e2 ,  ) ) =
  R( e1' ) op(fcnOp) R( e2' ) op(fcnOp)
  where R( e1, e2,  ) = (e1', e2', )-- foperator

R( fcn ( exprList ) ) =
    sqlFcn(fcn) ( R( exprList )      -- fscalar

R( e1, e2,.. ) = L(e1), L(e2),..    -- exprList

L( cell1 : cell2 ) =   -- L is a routine for lists
   s [ cell1 ],..,s [ cell2 ] -- crange_in_list

L( e ) = R( e ) , where e is not a cell range
                          -- other_expr_in_list
```

For example the following Excel spreadsheet

	A	B
1	1	=sum(A1:A4)
2	2	=product(A1,A2)
3	3	=ln(A1)

is translated as:

```
SELECT row,col, x FROM cells
MODEL DBY (row,col) MEA (x) AUTOMATIC ORDER
( x[1,2]= SUM(x)[1<=row<=4, 1<=col<=1],
                            -- B1=SUM(A1:A4)
  x[2,2]= x[1,1] * x[2,1],  -- B2=PRODUCT(A1,A2)
  x[3,2]= ln(x[1,1])        -- B3=LN(A1)
)
```

Note that the *Cells* table will contain 8 cells when persisted: constants in cells representing A1,A2,A3 and formulas in B1,B2,B3.

Vlookups and HLookups. These are frequently used Excel lookup functions. *Vlookup(key, range, col)* supports content-addressable lookup tables. A user specifies a *key* to match against, a *range* of cells, and which *column* to lookup within the range. The *key* is matched against *range's* first column, and the corresponding value in the lookup *column* is returned. There is no simple translation of *vlookup()* to a SQL function in the basic fix frame translation. However, *vlookup()* is very similar to SQL reference Model clause that implements lookup tables.

Informally, our translation of vlookups works as follows: For any formula containing a vlookup, we translate the formula to a reference model containing the key column as its dimension, and the lookup column as its measure. Thus, in the main MODEL clause, the reference model is used as a lookup table. More formally, assume that vlookup's range is defined by the left-upper corner $<r_s, c_s>$ and right lower corner $<r_e, c_e>$ Excel positions. The translation of a *Vlookup(key, (*$<r_s, c_s>$, $<r_e, c_e>$*), col)* results in the following SQL reference Model:

```
Q2
REFERENCE vlookup_ref ON
(  SELECT k.x key, v.x value
   FROM cells k, cells v
   WHERE k.col=cs & v.col=cs+col-1
       & k.row >= rs & k.row <= re & v.row=k.row )
DBY (key) MEA (value)
```

The *key* is in the first column of the range (c_s), the *value* is in the lookup column, the *key* and *value* are in the same row,

and that row is in the specified range (r_s to r_e). This reference model creates a mapping from *keys* (first column values) to *values* (lookup column values). A vlookup over the specified range, on the specified lookup column, with key value K is translated to:

```
vlookup_ref.value[K]
```

For example, vlookup(A1, A2:D4, 2) in cell A3 is translated as:

```
SELECT row, col, x FROM cells
MODEL
  REFERENCE vlookup_ref ON
   (SELECT k.x key,v.x value FROM cells k,cells v
    WHERE k.col = 1 & v.col = 2
        & k.row >= 0 & k.row <= 4 & v.row = k.row)
  DBY(key) MEA(value)
DBY (row, col) MEA (s)
(   s[3,1] = vlookup_ref.value[ s[1,1] ]   )
```

If an Excel spreadsheet has multiple vlookups referring to the same region, we construct a single SQL reference model for all of them. For example, vlookup(A1, A2:D4, 2) and vlookup(A1, A2:D4, 3) are satisfied with a single reference model.

We should mention that our vlookup() translation is based on finding an exact match of the specified key, whereas in Excel, vlookup() searches a column for a given key value and finds the row which has the largest value that is less than or equal to the specified value. However, equality matches are the most used and we implemented these.

Excel's *HLookup*() function is translated in a similar way.

This translation only works for vlookup's on constant data. In our formulation, the *key* and *value* are the values from the Excel cells and disregard the formulas stored there. To account for these, our SQL Reference clause would have to pull in the SQL rules corresponding to those formulas and any dependent cells. This is left for future work.

4.2 Table Translation.

The *table translation* creates named, rectangular, protected regions within Excel, called *RTables*, representing relational tables or views. A fundamental principle behind RTables and their Excel transformations is our ability to map them to SQL views. This mapping allows us to translate Excel operations over RTables to SQL to persist Excel computation in RDBMS.

An RTable has a header specifying the table and column names followed by the rows representing the data. Within Excel's RTable structures, we remember the associated metadata like the data types of the columns, PK and PK-FK constraints of the base tables, etc.

A region occupied by an RTable is visibly marked by an outline and is protected similar to the Excel Pivot Table, i.e., it cannot be split by adding new Excel columns or rows. We protect the shape of RTables to preserve and track their mapping to relational sources and other RTables.

An RTable can directly represent an RDBMS table or a view (*direct RTable*), or be derived from other RTables using a small set of Excel operations corresponding to the fundamental relational operators like join, aggregation, and inter-column computation (*derived RTable*).

A direct RTable is created from an RDBMS table or a view using our Add-On Excel menu, "QBX->RTable->LoadTable", similar to the "Data->Import External Data" menu available with Excel. Users have an option of importing the entire table, its schema only, or its sample. For the last case, the number N of imported rows must be given and we create a region with N rows even if the source has smaller cardinality. We provide random sampling (using ANSI SQL SAMPLE clause) or repeatable sampling for tables. The latter returns top N records ordered by the primary key of the table. A direct RTable by default inherits the name of its relational source and we require uniqueness of RTable names.

The data in RTables is always sorted for uniqueness of positioning within Excel. Sorting is done by either the PK of the source tables or by a user specified unique order. The latter is requested by standard Excel "Data->Sort" menu. The sort order is remembered by QBX. When an RTable is created on a relational table T, we will create a view T_o over T which in addition to T's columns has an integer column, rn, representing the ordering as a sequence of consecutive integers, 1,2... Column rn is calculated using the ANSI SQL function

```
row_number() over (order by <ordering cols>)
```

For example, Figure 4 shows RTables *fact* (A2:D10), *time_d (*I2:J4), *prod_d* (I6:J8), *region_d* (I10:J12) corresponding to our electronic warehouse. The user imported samples of the underlying relational tables.

Figure 4 **Electronic warehouse RTables in Excel**

	A	B	C	D	.....	I	J
1	**fact**					**time_d**	
2	**city**	**prod**	**month**	**sale**		**month**	**year**
3	LA	tv	m1.00	10.00		m1.00	y.00
4	LA	radio	m2.00	12.00		m2.00	y.00
5	LA	tv	m1.01	14.00		**prod_d**	
6	LA	radio	m2.01	16.00		**prod**	**categ**
7	Boston	tv	m1.00	20.00		tv	video
8	Boston	radio	m2.00	22.00		radio	audio
9	Boston	tv	m1.01	24.00		**region_d**	
10	Boston	radio	m2.01	26.00		**city**	**state**
11						LA	CA
12						Boston	ma

In the RDBMS, *fact* table will have a view $fact_o$ with the following column in addition to all columns of fact:

```
row_number() over (order by city, month, prod) rn
```

Similarly for the dimension tables, *time_d*, etc.

We provide a small set of operators on RTables (in menu and function form) to create derived RTables, emulating relational inter-column calculation, projection, join, aggregation and selection.

Inter-column calculations. Users add a new (calculated) column to an RTable with two steps.

1. The RTable is extended by a column and given a name via our "QBX->RTable->AddColumn" menu. This new column is initially populated with NULLs.

2. A calculation is added to the column. It is an Excel formula which references either constants or other columns from the **same** row in the RTable. The formula must be replicated in the entire column of the RTable.

Projection. Users can delete (i.e., project in relational terms) an RTable column via "QBX->RTable->DeleteColumn" menu. The menu acts similar to the standard "Edit-Cut" menu except that PK columns or the ordering columns of the RTable cannot be deleted, as they are required in order to preserve correspondence to relational sources and the uniqueness of their presentation.

Joining of RTables. Joining of two RTables R_1 and R_2 is non-trivial as Excel doesn't provide a natural correspondent of relational inner-join. The closest Excel operation, HLOOKUP or VLOOKUP, is similar to relational outer join, and this is what we implemented. The R_1 LEFT OUTER JOIN R_2 operation is similar to adding a calculated column and proceeds in two steps:

1. A new column is added to the left table of the outer join, R_1
2. The column is populated with VLOOKUP($R_1.col_1$, R_2, $R_2.col_2$), which left outer joins $R_1$to R_2 and projects the $R_2.col_2$.

Figure 5 **Join of fact to its dimension tables**

	A	B	C	D	E	F	G
1	**fact**						
2	**city**	**prod**	**month**	**state**	**categ**	**year**	**sale**
3	LA	tv	m1.00	ca	video[a]	y.00[b]	10.00
4	LA	radio	m2.00	ca	audio	y.00	12.00
5	LA	tv	m1.01	ca	video	y.01	14.00
6	LA	radio	m2.01	ca	audio	y.01	16.00
7	Boston	tv	m1.00	ma	video	y.00	20.00
8	Boston	radio	m2.00	ma	audio	y.00	22.00
9	Boston	tv	m1.01	ma	video	y.01	24.00
10	Boston	radio	m2.01	ma	audio	y.01	26.00

a. =vlookup(B3, I7:J8, 2)
b. =vlookup(C3, I3:J4, 2)

Figure 5 shows a join of the fact table with the dimension tables. We have extended the *fact* RTable with three columns (state, categ, year) and placed the Excel vlookup functions joining to the dimensions. For example, cell F3 contains vlookup(C3, I3:J4, 2) that looks up the year of 'm1.00' in the time_d RTable and returns 'y.00'. Region I3:J4 corresponds to the dimension table time_d from Figure 4. Cells E3:E10, representing year, have analogous vlookup functions. Cell E3 contains vlookup(B3, I7:J8, 2), which looks up the category of 'tv' in prod_d, resulting in 'video', etc.

The inner join is not supported by our translator, which is quite limiting. We plan to simulate it with a menu driven version of the join, similar to the Pivot Table and provide its evaluation support in Excel. However, since we remember the integrity constraints on RTables, if there is an enforced Foreign key constraint between join columns, outer join is converted to inner join.

Aggregating an RTable. Excel provides multiple ways to aggregate data, for example aggregation of named regions, the "Data->Subtotal" and the "Data->PivotTable" aggregations. We have selected the latter two as the most general and most similar to relational algebra. Both operations accept regions of data (hence the resemblance to relational algebra), which in our case are RTables, and both produce multiple aggregations. They are easily simulated with SQL GROUPING SETS functionality. The SQL query must have an ORDER BY to preserve the ordering generated by Excel aggregation, see examples in Q3 and Q4.

"Data->PivotTable" aggregation can also pivot the data using an additional SQL GROUP BY operator or with the proposed SQL pivot operator [3]. For example, consider applying a PivotTable operation to the RTable from Figure 5. We will pivot region D3:G10. It can be aggregated using a PivotTable without pivoting as shown in Figure 6, or with pivoting as shown in Figure 7. Aggregation in Figure 6 uses Excel PivotTable option "Grand Total For Rows"; in Figure 7 uses "Grand Total for Rows" and "Grand Total for Columns" The corresponding SQL expressions are given in Q3 and Q4, respectively.

Figure 6 **Aggregation using PivotTable without Pivoting Q3**

	L	M	N
1	**PivotQ3**		
2	**state**	**year**	**total**
3	ca	y.00	22.00
4		y.01	30.00
5	ca total		52.00
6	ma	y.00	42.00
7		y.01	50.00
8	ma total		92.00

Q3 - without pivoting and "Grand total for rows" option

```
SELECT state, year, sum(amt) amt,
  row_number() over (order by state,year) rn
FROM
 fact f outer join time_d t on f.month=t.month
        outer join prod_d p on f.prod = p.prod
        outer join geog_d g on f.city = g.city
GROUP BY GROUPING SETS ((state,year),(state))
ORDER BY state NULLS LAST, year NULLS LAST;
```

If there are Foreign key constraints between join columns, outer joins in Q3 and Q4 are replaced with inner joins.

Query Q4 can be expressed more elegantly using PIVOT [3] operation.

Figure 7 **Aggregation using PivotTable with Pivoting Q4**

	L	M	N	O
1	**PivotQ4**			
2	prod	audio		
3				
4	**state**	**y00**	**y01**	**total**
5	ca	12.00	16.00	28.00
6	ma	22.00	26.00	48.00
7	total	34.00	42.00	76.00

Q4 - with pivoting on time column

```
SELECT state,
  sum(case when year = 'y00' then amt end) y00,
  sum(case when year = 'y01' then amt end) y01,
  sum(case when year is null then amt end) total,
```

```
  row_number() over (order by state nulls last)rn
FROM
( SELECT state, year, sum(amt) amt
  FROM
    fact f outer join time_d t on f.month=t.month
           outer join prod_d p on f.prod = p.prod
           outer join geog_d g on f.city = g.city
  WHERE prod = 'audio'   -- <a predicate on prod>
  GROUP BY cube(state,year))
)
GROUP BY state ORDER BY state NULLS LAST;
```

Selecting data from an RTable. Excel provides an advanced filtering facility "Data->Filter->AdvancedFilter". It filters entire rows from Excel regions and hence is very suitable for filtering of RTables. This operation on RTables corresponds to the WHERE clause of an SQL query block and this is how our tool translates it to SQL.

Composition of RTable Transformations. QBX tracks the above operations on RTables. Any RTable can be a subject to Aggregation, Outer Join or Selection and this will create another RTable. We can track back the operation to the direct RTables (i.e., ones with direct relational tables) and re-create a corresponding SQL view over the relational tables. For example, aggregation on Figure 6, can be expressed in terms of relational tables as Q3.

Persisting of RTables in RDBMS. A user can persist a derived RTable in the RDBMS either as a view, or a materialized view (MV) using our menu "QBX->RTable->SaveTable". The view is expressed using a set of SQL transformations on the direct RTables. Note that when a user creates a direct RTable in Excel, he/she may request only a sample of the data from the underlying relational table. However, when we persist a derived RTable, we create the view on entire relational tables, not their samples. Hence the views corresponding to RTables operate on the entire underlying relational tables. For example, aggregation on Figure 6 is stored as a view on entire relational tables, see Q3, even though the RTables on Figure 4 contained only samples of data. Samples were imported into Excel just to provide example data which fits in a limited space.

In addition to storing an RTable as an RDBMS object (view or an MV), we record the position within Excel of the RTable so it can be later restored into the same place.

Refreshing a derived RTable from the RDBMS. Once a relational view representing an RTable has been created, the RTable can be restored back to Excel. This gives users the desirable capability to design a series of relational operations in Excel on a sample of the relational tables which fits in a limited Excel sheet, store this as a relational view, compute it in the RDBMS using scalable and parallel execution, and bring the final results back to Excel. It is very likely that the final result, due to aggregation, will be small enough to fit into Excel even though the original relational inputs would not. Refreshing is done using the same menu as loading, "QBX->RTable->LoadTable".

4.3 Translation of Fix Frame Operations on RTables

Frequently, an Excel application loads pre-processed data from external sources like relational tables, places them in Excel contiguous regions, and analyzes them further using Excel formulas. With RTables, we provide analysts a tool to perform the pre-processing of the relational data directly in Excel and represent it as a derived RTable. Ideally, we would like to represent the subsequent Excel computation over RTables as further SQL transformations, which is difficult to express using standard ANSI SQL as users can specify an arbitrary formula for every row of an RTable, each referencing arbitrary cells.

To support arbitrary Excel calculations on RTables, we linearize them as 2-D arrays with Excel (row, col) coordinates. Consider an RTable R(c1, c2, ..., cn) with M rows whose left-upper corner is rooted at Excel at $<r_s, c_s>$ position, and right-lower corner is at $<r_s+M, c_s+N>$. There is a relational view T_o corresponding to R with the same columns (c1,... cn), as well as an integer column, rn, representing unique ordering of rows for placement of T_o's data within Excel. Note that R represents all rows in T (i.e., R is built on the entire data set of T), otherwise linearization would depend on the size of the sample.

We experimented with two linearization techniques. First, called *Assignment Linearization*, represents T_o as a SQL Reference Model dimensioned by Excel coordinates (row, col). The transformation is based on stacking column c_i on the previous column c_{i-1} with a UNION ALL operator. For example, the first column, c1, of T_o is represented as

```
SELECT rn+r_s AS row, c_s+1 AS col, c1 AS x FROM T_o,
```

the second column, c2, as

```
UNION ALL
SELECT rn+r_s AS row, c_s+2 AS col, c2 AS x FROM T_o
```

etc. In the Main SQL Model, it assigns the reference values to corresponding Excel cells in the ($<r_s, c_s>$, $<r_s+M, c_s+N>$) region. The SQL formulation is:

```
Q5
SELECT row,col, x FROM cells
MODEL
  REFERENCE ref_t ON
  (SELECT rn+r_s rn, c_s+1 c, c1 x FROM t_o UNION ALL
   SELECT rn+r_s rn, c_s+2 c, c2 x FROM t_o UNION ALL
      ..
   SELECT rn+r_s rn, c_s+M c, cn x FROM t_o)
  DBY (rn, c) MEA (x)
MAIN DBY (row, col) MEA (x)
( x[r_s<=row<=r_s+M, 1] = ref_t.x[cv(row)],
  x[r_s<=row<=r_s+M, 2] = ref_t.x[cv(row)],
  ..
  x[r_s<=row<=r_s+M, N] = ref_t.x[cv(row)],
  -- followed by translated Excel formulas
)
```

This linearization assumes that columns of different data types are first converted to a common type, such as the character type.

The second linearization, called *Reference Linearization*, directly translates references to cells occupied by RTable R to references to a reference model as shown below:

```
  REFERENCE ref_t ON
  (    SELECT rn, c1, c2, .., cn x FROM t_o  )
  DBY (rn) MEA ( c1, c2, ... cn)
```

In this case, since the RTable is rooted at $<r_s, c_s>$, and extends N columns to the right and M columns down,

references to a cell <r_s+x, c_s+y> in this region is expressed by SQL model reference

```
Q6      ref_t.c_y[x]
```

As an example, consider RTable from Figure 6, and the following fragment of Excel which calculates, per each state, ratio of sales in individual year to total sales in all years.

Figure 8 **Example-2. Ratio yearly sales to total sales per state**

	L	M	N	O
1	**PivotQ3**			
2	**state**	**year**	**total**	ratio
3	ca	y.00	22.00	=N3/N5
4		y.01	30.00	=N4/N5
5	ca total		52.00	=N5/N5
6	ma	y.00	42.00	=N6/N8
7		y.01	50.00	=N7/N8
8	ma total		92.00	=N8/N8

It is translated using the Assignment Linearization method as:

```
SELECT row, col, x FROM cells
MODEL
  REFERENCE ref_t ON
  ( SELECT rn+2, 1 c, state x FROM t_o UNION ALL
    SELECT rn+2, 2 c, time  x  FROM t_o UNION ALL
    SELECT rn+2, 3 c  total x  FROM t_o )
  DBY (rn, c) MEA (x)
MAIN DBY (row, col) MEA (x)
( x[3<=row<=8, 12] = ref_t.x[cv(row)],-- column L
  x[3<=row<=8, 13] = ref_t.x[cv(row)],-- column M
  x[3<=row<=8, 14] = ref_t.x[cv(row)],-- column N
  x[3, 15] = x[3, 14] / x[5, 14],  -- =N3/N5
  x[4, 15] = x[4, 14] / x[5, 14],  -- =N4/N5
  x[5, 15] = x[5, 14] / x[5, 14],  -- =N5/N5
  x[6, 15] = x[6, 14] / x[8, 14],  -- =N6/N5
  x[7, 15] = x[7, 14] / x[8, 14],  -- =N6/N5
  x[8, 15] = x[8, 14] / x[8, 14]   -- =N8/N5
)
```

The Reference Linearization is:

```
Q7
SELECT row, col, x FROM cells
MODEL
  REFERENCE r ON
  ( SELECT rn, state, time, total  FROM t_o  )
  DBY (rn) MEA (state, time, total)
MAIN DBY (row, col) MEA (x)
( x[3, 15] = r.total[1] / r.total[3],  -- =N3/N5
  x[4, 15] = r.total[2] / r.total[3],  -- =N4/N5
  x[5, 15] = r.total[3] / r.total[3],  -- =N5/N5
  x[6, 15] = r.total[4] / r.total[6],  -- =N6/N5
  x[7, 15] = r.total[5] / r.total[6],  -- =N6/N5
  x[8, 15] = r.total[6] / r.total[6]   -- =N8/N5
)
```

In the above translations, table t_o corresponds to the PivotQ3 table - see Q3.

Observe that with Assignment Linearization, translation of an Excel formula referencing the RTable region (<r_s, c_s>, <r_s+M, c_s+N>) can be done as the fix frame translation of Section 4.1, since the SQL Main Model populates the cells corresponding to the RTable before executing rules which reference the RTable. This applies to any Excel function except the lookup functions, like *vlookup()*, which reference portions of the RTable. These functions are translated directly using the SQL Reference Model of Q5.

The Reference Linearization avoids assignment to the Excel cells using SQL Model rules and uses direct references to the SQL Reference Model instead. This requires us to verify that a reference to a cell falls within an RTable region during formula translation (see Figure 3), and if so, use the translation of Q6.

For reference linearization, range arguments to Excel functions have to be translated to lists of cells. For example, consider a cell containing the formula SUM(N3:N4) in Figure 8. The formula must be translated first to SUM(N3,N4), then to the infix notation N3+N4, which finally can be translated to SQL as:

```
r.total[1]+r.total[2]
```

Rules of Figure 3 perform this translation for Excel functions except lookup-functions like *vlookup*(). A lookup function on an RTable R is translated by constructing a SQL Reference Model which is dimensioned by the column of R serving as a lookup key and whose measure is the looked up column. For example, consider *vlookup(key,M3:N4, 2)* in the spreadsheet of Figure 8. M3:N4 represents a region within the RTable with column *time* serving as the lookup key, and column *total* as the looked up column hence the formulation:

```
REFERENCE ref_lookup ON
(    SELECT time, total FROM t_o WHERE rn<=2 )
DBY (time) MEA (total)
```

vlookup(key,M3:N4, 2) is then translated as a reference to:

```
Q8      ref_lookup.total[key]
```

Observe that the SQL Reference Model contains a WHERE predicate (rn <= 2) restricting the underlying table to the rows present in RTable. Our translation of lookup functions on RTables assumes that the range of the function doesn't extend beyond the RTable.

The Reference Linearization avoids assignment to the Excel cells using SQL Model rules and multiple scans of the relational table corresponding to the RTable t_o in Q5. So it is likely to be more efficient than Assignment Linearization. Section 6 compares performance of the two methods.

In many cases, OLAP in particular, the RTable represents a relatively small cube or small aggregation where the number of rows in the corresponding relational table fits in an Excel sheet and does not change often. In such cases, it may be useful to store fixed frame computations on the RTable as a new column. For example, consider Figure 8. Assuming that PivotQ3 represents all data (i.e, RDBMS stores only CA and MA states and years y.00 and y.00), we could make column *ratio* (Excel column O), a calculated column of RTable. The new column *ratio* will be calculated using SQL Model rules. Our translator will construct the following query:

```
Q9
SELECT rn, state, year, total, ratio FROM PivotQ3
MODEL
  REFERENCE r ON
  ( SELECT rn, total  FROM PivotQ3  )
  DBY (rn) MEA (total)
MAIN DBY (rn) MEA (state, year, total, 0 ratio)
```

```
( ratio[1] = r.total[1] / r.total[3],  -- =N3/N5
  ratio[2] = r.total[2] / r.total[3],  -- =N4/N5
  ratio[3] = r.total[3] / r.total[3],  -- =N5/N5
  ratio[4] = r.total[4] / r.total[6],  -- =N6/N5
  ratio[5] = r.total[5] / r.total[6],  -- =N6/N5
  ratio[6] = r.total[6] / r.total[6]   -- =N8/N5
)
ORDER BY rn
```

Observe that we reference the PivotQ3 view twice in Q9. The first reference is in the main query block and the second in the Reference Model clause. Values of the *ratio* column are computed using fixed frame computation on the reference model "r". Our translator verifies that Excel formulas reference only cells within the RTable. We refer to this translation, where an RTable has a column that is calculated using translated Excel formulas, as *RTable Reference Linearization*. This also works when the calculated column references other RTables. These tables will be added to the list of reference models in the SQL query block calculating the column.

4.4 Relative referencing to RTables

Section 4.3 showed how to add a calculated column to an RTable based on fixed frame Excel computation - see for example Q9. The method, although computationally general as it translates any Excel formula, is limited to RTables that are based on a full data set and do not change after translation occurred. We need a method which is computationally rich and can operate on samples of RTables like the methods discussed in Section 4.2.

As noted before, we remember an RTable's primary key (PK) in its metadata. For example, the PK of RTable *PivotQ3* from Figure 6 is (*state*, *year*). To support referencing values from RTables, we have implemented a new Excel lookup function *rtlookup(rtregion*, *col*, *pkeys)* using VBA. Given a region of RTable (*rtregion*), its primary key values (*pkeys*) and its column number (*col*), it returns the value of that column at the primary key. For example, in Figure 9:

```
rtlookup(L2:N8, 3, 'ca', 'y.01')
```

retrieves the value of column #3, i.e., the *total* column, for key 'ca' and 'y.01', which is 30. Users can also use the symbolic name of the RTable (instead of its range) and the name of the referenced column (instead of its relative number) as shown below:

```
rtlookup(PivotQ3, total, 'ca', 'y.01')
```

The *rtlookup()* function can be used to populate a new column in an RTable. If the calculations in every row of the new column are the same except for relative references, our translator will convert *rtlookup* into a rule in SQL Model clause.

For example, consider Figure 9 which represents PivotQ3 RTable from Q3. We added a new column, *ratio* representing ratio of yearly sales to total sales per region. The formula in the column is of the form:

```
Ni/rtlookup(L2:N2, 3, Li, null)
```

rtlookup(L2:N2, 3, L_i, null) retrieves total sales of every state. Our translator translates it to SQL Model rule:

```
ratio[*,*]=total[CV(),CV()]/total[CV(state),null]
```

Note that equivalent rules have been collapsed in this translation, which is an optimization we discuss in Section 5.

Figure 9 **Usage of rtlookup**

	L	M	N	O	P	R
1	**PivotQ3**					
2	**state**	**year**	**total**	**ratio**		
3	ca	y.00	22.00	0.42[a]		
4		y.01	30.00	0.58[b]		
5	ca		52.00	1.00		
6	ma	y.00	44.00	0.48		
7		y.01	50.00	0.52		
8	ma		94.00	1.00		

a. =N3/rtlookup(L2:N2, 3, L3, null)
b. =N4/rtlookup(L2:N2, 3, L4, null)

The RTable PivotQ3 will be translated to SQL as a view:

Q10
```
CREATE VIEW PivotQ3 AS
SELECT state, year, ratio, sum(amt) amt,
  row_number() over () (order by..) rn
FROM
 fact f outer join time_d t on f.month=t.month
        outer join prod_d p on f.prod = p.prod
        outer join geog_d g on f.city = g.city
GROUP BY GROUPING SETS ((state,year),(state))
MODEL DBY (state, year) MEA (total, 0 ratio)
(   ratio[*,*]
      = total[CV(),CV()]/total[CV(state), null]
)
order by state nulls last, year nulls last;
```

This translation will work well on samples of data as it can be correctly applied to entire tables.

Furthermore, we found that usage of *rtlookup()* is quite general. *rtlookup()* could reference not only the RTable we are adding a column to, but other RTables in the spreadsheet. These RTables, which can be based on samples, are added to the reference model of the SQL view. For example, given an RTable *years_dim (year, prev_year)* that stores the previous year for a given year, we can calculate the year-to-year (*y_diff*) sales difference in Figure 10. This would add a new reference model and a new rule to the SQL Model of Q10

```
MODEL
  REFERENCE r ON
  ( SELECT year, prev_year  FROM year_dim  )
  DBY (year) MEA (prev_year)
DBY (state, year) MEA (total, 0 ratio)
(  ratio[*, *] =
       total[CV(),CV()]/total[CV(state), null],
   y_diff[*,*] = total[CV(),CV()] -
                 total[CV(),r.prev_year[CV(year)]]
)
```

Observe that this view would operate on entire tables.

Figure 10 **Usage of rtlookup**

	L	M	N	O	P	R	S	T
1	**PivotQ3**							
2	**state**	**year**	**total**	**ratio**	**y_diff**		**year_d**	
3	ca	y.00	22.00	0.42[a]	[b]		**year**	**prev**
4		y.01	30.00	0.58[c]	0.12[d]		y.00	y.99
5	ca		52.00	1.00			y.01	y.00

a. =N3/rtlookup(L2:N2, 3, L3, null)
b. =N3 - vlookup(rtlookup(S3:T5, 2, M3), M3:M4, 2)
c. =N4/rtlookup(L2:N2, 3, L4, null)
d. =N4 - vlookup(rtlookup(S3:T5, 2, M4), M3:M4, 2)

The relative translation results in some cases in parallelizable execution. SQL Model engine promotes independent dimensions into PBY clause hence creating partitions which can be executed in parallel. For example, Q10 is transformed as:

```
MODEL PBY (state) DBY (year) MEA (total, 0 ratio)
(    ratio[*] = total[CV()]/total[null]   )
```

and the rule can be executed in parallel for each partition.

5 Optimizations

In Section 4.3 we described two methods for translation of fixed frame computation on RTables. In addition to this, we consider two other optimizations to the translation.

Collapsing of Equivalent Rules. In many cases, Excel formulas in a row or in a column are identical to each other except for the relative differences. In this case, they can be translated as a single SQL Model rule. For example, consider the following Excel fragment:

Table 1: Example-2. Collapsing of Formulas

	A	B	C
1	1	4	A1+B1
2	2	5	A2+B2
3	3	6	A3+B3

This Excel spreadsheet would be translated into a SQL Model clause with three rules, one per formula. In fact the three formulas are equivalent given that the cell references are relative and they can be replaced by a single SQL Model rule instead:

```
Q11      x[FOR row FROM 1 to 3 INCREMENT 1,3]=
                s[CV(row),1] + s[CV(row),2]
```

Our translation uses this optimization to coalesce equivalent formulas in a contiguous range of cells (in a row or column). As another example, the SQL Model clause from Q7 will be simplified to:

```
MAIN DBY (row, col) MEA (x)
( x[FOR row 3 to 5 INCREMENT 1, 3] =
    total[CV(row) - 2] / total[3],
  x[FOR row 6 to 8 INCREMENT 1, 3] =
    total[CV(row) - 2] / total[6]
)
```

Fortunately, the Excel Algebra for formulas is a regular expression, hence finding equivalence amounts to their recursive traversal and can be done in $O(N^2)$ time where N is the number of Excel formulas.

The optimization significantly reduces SQL Model compilation time as the compiler deals with fewer rules when deciding their ordering. It also reduces execution time as SQL Model uses an efficient run time looping for its FOR loops or existential rules.

For Loops vs. Existential Form. Our initial fix frame translation expanded an Excel cell range $<r_1, c_1> : <r_2, c_2>$ reference to

```
x[ r1 <= row <= r2, c1 <= col <= c2 ]
```

See crange and faggregate transformations in Figure 3. SQL Model evaluates this rule as a scan of all data, so in some cases the looping construct is more efficient:

```
x[ FOR row FROM r1 TO r2 INCREMENT 1,
   FOR col FROM c1 TO c2 INCREMENT 1]
```

On the other hand, in the example of Q11, if the entire column of Excel or RTable is populated with the same formulas (except for relative differences), we could use an existential (term of [2]) rule instead of the FOR loop rule:

```
Q12      x[*, 3]= s[CV(row),1] + s[CV(row),2]
```

For rules operating on a large data set, this formulation is more efficient than the one of Q11 as it uses a scan rather than random access of cells. The decision of which translation to use (FOR loop or existential rule) should be cost based and integrated into the SQL Model engine. Our current implementation uses heuristics and applies the existential form to RTables only when a scan is more efficient. In the fix frame translation, we always generate the FOR loop rule.

6 Performance Of Excel Translation

Our experiments are based on an actual Excel spreadsheet computing two financial measures: increase of sales from prior period and ratios of sales between different levels of product dimension.

For experiments with RTables we used a synthetic star schema from Section 2. The *time_dim* contained 3 levels (*month, quart, year*) for a total of 10 years. *region_dim* contained 2 levels (*city, state*) for 1000 cities in the USA evenly distributed among states. *prod_dim* contained 3 levels (*prod, brand, categ*) with 10,000 products in 1000 brands, and 100 categories. The fact table had 10,000,000 rows. It was populated so that the aggregated cube starting from the second level of dimensions was dense, i.e., every element of (*quart* X *brand* X *state*) is present in cube.

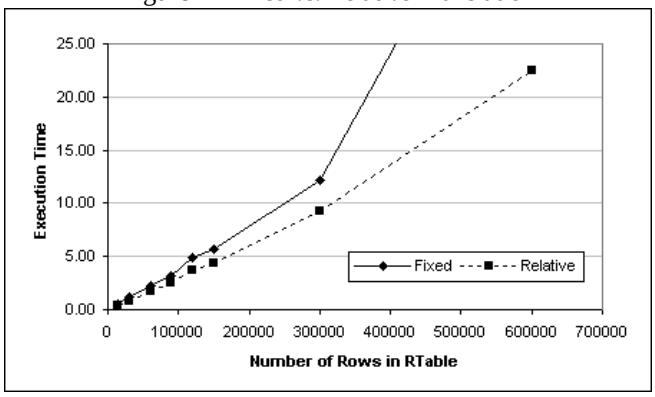
The experiments were conducted on a 12 CPU, 336 Mhz, shared memory machine with a total of 12 GB of memory.

Linearization Techniques. We compared fixed frame and relative RTable translation techniques. In particular, we chose to compare the fixed frame RTable Reference Linearization technique with relative reference technique of Section 4.4. We used the above mentioned schema and queries Q9 and Q10 for this experiment assuming Foreign key relationships between join columns (hence outer joins were inner joins). We varied the size of the RTable and measured the execution time of the queries.

The relative translation technique outperforms the fixed frame translation technique in all cases. This is expected as Q10 requires one scan over data whereas Q9 requires as many lookups as there are rows in RTable. SQL MODEL evaluation of Q10 requires us to scan the data and evaluate the rule for each record scanned. Rule evaluation requires a lookup operation for the right side and an assignment operation. However, for Q9, each rule evaluation requires three lookup operations. The fact that Q9 performance is reasonably close to that of Q10 shows that the MODEL

access structure (hash table) lookup operation is efficient.

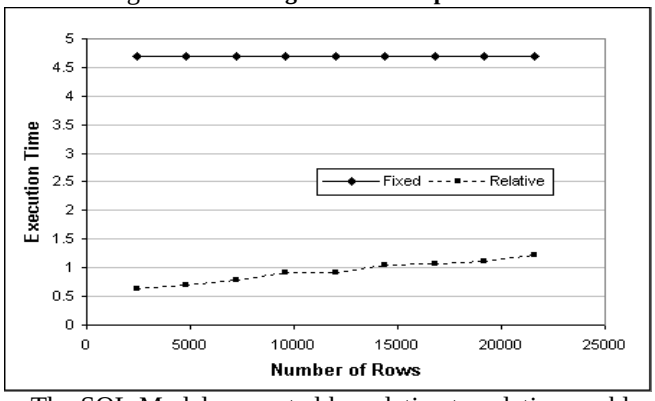
Figure 11 **Fixed vs. Relative Translation**



The performance of fixed frame translation degrades drastically (the time taken by Q9 when RTable had 600,000 rows is way off the graph) once data spills to disk. Q10 performs well as it processes data in two stages - first, it splits data into partitions based on *state* and then processes one partition at a time. It can do that because *independent dimension* [2] optimization of SQL Model identifies *state* as an independent dimension and promotes it as a PBY key. In Q9, the entire data has to be processed as one partition as there is no independent dimension. In this case, Q9 requires random disk accesses and hence performs poorly.

The SQL Model generated by relative RTable translation can be executed in parallel because of the independent dimension optimization. SQL MODEL computation of Q10 can be evaluated in parallel by distributing the data (either range or hash based) across processing elements based on *state.* The SQL Model generated by fixed translation of Q9 cannot be executed in parallel.

Figure 12 **Pushing Predicates Optimization**



The SQL Model generated by relative translation enables further optimization of pushing predicates. Predicates in the outer query block on columns that are PBY or independent dimensions of SQL Model can be pushed through the MODEL clause, SQL analytic functions, and Group-By into the WHERE clause of the inner query. We have added a predicate on *state* and measured the execution time of the queries Q9 and Q10 varying the selectivity of the predicate. Q9 always takes the same time (and is inferior to Q10) as the predicate was not pushed through the MODEL clause. The predicate on *state* gets pushed through the MODEL clause of Q10 as *state* is an independent dimension. Q10's performance is linear to the selectivity of the predicate. Figure 12 shows this result.

Our experiments show that fixed RTable Reference Linearization translation, though general, suffers from inefficient execution. We leave improvements of SQL Model queries generated by this technique as future work.

Rule Pruning. Our Fixed Frame translation stores Excel computation as an SQL View with schema: (*sheet, row, col, x*) over the *Cells* table, for example see Figure 2 and Q1. In many cases, users may be interested only in a small subset *S* of cells of the spreadsheet. This set may be derived from a larger set S_d which is still small in comparison to all populated cells. In this case, it is beneficial to evaluate only cells in S_d to return set *S*.

SQL Model [2] includes an optimization called rule pruning which eliminates rules whose computations are filtered out by the outer filters (e.g, WHERE clause of an outer query block). To test it we designed two experiments.

In the first experiment Excel formulas depend on a single cell only. Odd Excel rows contain data and even rows formulas. Each formula copies the value of the cell above as shown in Figure 13:

Figure 13 **Example3. Formulas depend on single cell.**

	A	B	C
1	1	2	3
2	=A1	=B1	=C1

In the second experiment, we constructed in Excel rectangles where formulas form a dependency chain as shown below:

Figure 14 **Example4. Formulas form dependency chains**

	A	B	C	D	E	F
1	1	=A1	=B1	1	=D1	=E1
2	=C1	=A2	=B2	=E1	=D2	=E2
3	=C2	=A3	=B3	=F2	=D3	=E3

In Figure 14 we partitioned the spreadsheet into rectangles with 9 cells. Within each rectangle, the second cell depends on the first, the third on the second, etc.

In both cases we issued queries against the Excel view. The queries retrieved a single cell. In the case of Figure 13, this was a cell from an even row. In the case of Figure 14, this was a cell from the right lower corner of a rectangle, as illustrated below:

```
Q13     SELECT sheet, row, col, x FROM ExcelView
         WHERE sheet=1 AND row=3 AND col=3;
```

We observed rule pruning using the RDBMS explain plan facility and internal traces. In case of Figure 13, the SQL Model engine performs optimal pruning. Rules are pruned to a single rule and we retrieve from the *Cells* table a minimal set of cells. For example, when the outer filter selects cell B2 the engine retrieves only B1 and B2 from the *Cells* table.

In the case of chain dependency of Figure 14, the SQL Model engine performs optimal pruning as well. Rules from unreferenced rectangles are pruned away. For example, in the case of Q13 we execute only rules from A1:C3 region and we select only cells from the A1:C3 region from the *Cells* table.

Effect of Optimizations of Section 5. The Collapsing of Equivalent Rules optimization replaces multiple rules with an equivalent single FOR loop rule or an Existential rule. Figure 15 shows its effect on the analysis time of SQL Model as a function of the number of rules. As expected, that time increases quadratically with increasing the number N of uncollapsed rules (according to [2] the analysis is $O(N^2)$). The analysis time becomes negligible when rules are collapsed into a single rule. This optimization also reduces the execution time (not shown) about 5% to 10% due to a more efficient execution mechanism.

Figure 15 **Collapsing of Rules.**

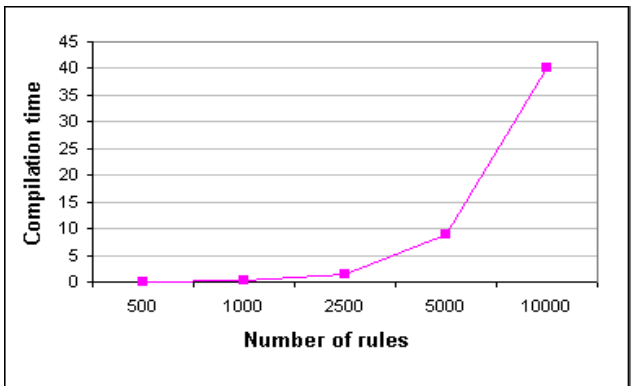
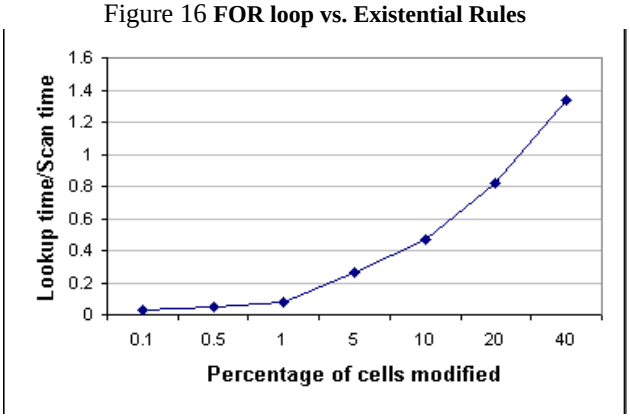


Figure 16 **FOR loop vs. Existential Rules**



As mentioned in Section 5, rules can be collapsed either into a FOR rule (see Q11) or an Existential Rule (see Q12). The former implies execution based on random access to cells via a SQL Model hash table and the latter an execution with a scan over the data. We compared performance of the two translations as a function of the percentage of data the rules access for the case when the SQL Model hash table fits into memory. Figure 16 depicts the ratio of execution time of an Existential rule to the equivalent rule with FOR loops. We had 100,000 cells in the SQL Model access structure when the rules were evaluated. Evaluation time for the existential rule does not change significantly with the increased number of cells that are accessed, although the execution time for the FOR loop method increases linearly. The graph depicts that FOR loops are more efficient when less then 30% of cells are modified (provided that the access structure fits in memory).

When the data spills to disk, both methods degrade very quickly as both may involve random access to the cells. In this case, we observed that a scan performs better as it pages more efficiently.

7 Conclusions

Our goal was to translate Excel computation to SQL and use natural extensions to Excel formulas and menus to perform Relational Operations on RDBMS tables. We found that computation can be specified on samples of data which fit in Excel and then applied to entire relations within RDBMS. We found that the type of translation is critical. The fix frame translation even though very general was not as efficient as more restrictive relative translations over RTables. The latter resulted in SQL formulations which are scalable in size and parallelization and, in many cases, easily optimizable by relational engines.

We found that existing Excel operations, in particular pivot and advanced filtering, are very suited to simulate Relational Aggregation and selection. We postulate that if Excel had a few more Relational friendly extensions, particularly those that simulate joins, allow partitioning of Excel regions ala SQL window functions [4] and perform relative computations, it could replace most of query building and reporting tools. Thousands of analysts with Excel expertise could construct scalable and efficient relational queries without writing SQL. Finally, we found many essential functions missing from SQL, in particular, financial ones.

8 References

[1] A. Witkowski et al, “Spreadsheets in RDBMS for OLAP”, *Proceedings of ACM SIGMOD 2003, San Diego, 2003.*

[2] A. Witkowski et al, “Advanced SQL Modeling in RDBMS”, *Transactions on Database Systems, vol 30, issue 1, March 2005.*

[3] C. Cunningham, C. Legaria, G. Graefe, “PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS”, *Proceedings of VLDB 2004, Toronto, Canada, 2004*.

[4] F. Zemke et al, “Introduction to OLAP functions”, ANSI NCITS H2-99-154r2, Change Proposal, May 1999.

[5] P. Blattner. Microsoft Excel Functions in Practice. QUE, 1999.

[6] J. Simon, Excel 2000 in a Nutshell. O’Reilly & Associates, Inc. 2000.

[7] OLAP Application Developer’s Guide, 2004. Oracle Database 10g Release 1 (10.1) Documentation. 2004.

[8] T., Peterson, J. Pinkelman, Microsoft OLAP Unleashed. SAMS Publishing, 2000.

[9] C. Howson. Business Objects; The Complete Reference. McGraw-Hill, 2002.

[10] Exploiting The Power of Oracle Using Microsoft Excel. http://www.oracle.com/technology/products/bi/pdf/BI_Spreadsheet_Addin_WP.pdf

[11] SAP BI Excel Add-in: http://www.sap.com/solutions/netweaver/businessintelligence/pdf/BWP_BI_Overview.pdf, page 7.

[12] SAS Technologies/Analytics: http://www.sas.com/technologies/analytics/statistics/index.html