# Spreadsheet As a Relational Database Engine

[Extended Abstract]

Jerzy Tyszkiewicz
Institute Informatics, University of Warsaw
Banacha 2, 02-097 Warsaw, Poland
jty@mimuw.edu.pl

## ABSTRACT

Spreadsheets are among the most commonly used applications for data management and analysis. Perhaps they are even among the most widely used computer applications of all kinds. However, the spreadsheet paradigm of computation still lacks sufficient analysis.

In this paper we demonstrate that a spreadsheet can play the role of a relational database engine, without any use of macros or built-in programming languages, merely by utilizing spreadsheet formulas. We achieve that by implementing all operators of relational algebra by means of spreadsheet functions.

Given a definition of a database in SQL, it is therefore possible to construct a spreadsheet workbook with empty worksheets for data tables and worksheets filled with formulas for queries. From then on, when the user enters, alters or deletes data in the data worksheets, the formulas in query worksheets automatically compute the actual results of the queries. Thus, the spreadsheet serves as data storage and executes SQL queries, and therefore acts as a relational database engine.

The paper is based on Microsoft Excel (TM), but our constructions work in other spreadsheet systems, too. We present a number of performance tests conducted in the beta version of *Excel* 2010. Their conclusion is that the performance is sufficient for a desktop database with a couple thousand rows.

## Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*relational databases*; H.4.1 [**Information Systems Applications**]: Office Automation—*spreadsheets*; K.8.1 [**Personal Computing**]: Application Packages—*spreadsheets*

## General Terms

Theory, Design, Performance



## Keywords

spreadsheets, relational databases, relational algebra, SQL, end-user computing

## 1. INTRODUCTION

Spreadsheets are the end-user computing counterpart of databases and OLAP in enterprise-scale computing. They serve basically the same purpose — data management and analysis, but at the opposite extremes of the data quantity scale.

At the same time spreadsheets are a great success, and are often described as the very first "killer app" for personal computers. Today they are used to manage home budgets, but also to create, manage and examine extremely sophisticated models and data arising in business and research. For example, *Excel* files are quite common as a form of supporting online material in *Science* journal.

It seems therefore surprising that relatively little research has been devoted to them and consequently they are still poorly understood. This is a source of many problems, e.g., *Science* provides an example of a scientific controversy [5, 3] which finally turned out to be related to the design of a spreadsheet used for data analysis. European Spreadsheet Risks Interest Group EuSpRIG `http://www.eusprig.org` with its annual conference are devoted to the problems with spreadsheets.

## 2. THE CONTRIBUTION

In this paper we aim at providing strong evidence, that spreadsheets are not only useful and successful, but also very interesting, and deserve more research. Specifically, we consider the relation of spreadsheets to database systems. It is a natural comparison, because spreadsheets indeed often play the role of small databases at the end-user level of computing.

We demonstrate that virtually any spreadsheet software present on the market is a relational database engine. We do so by implementing all operators of relational algebra using spreadsheet functions. For each query in SQL, we demonstrate how to construct a spreadsheet workbook with empty worksheets for data tables and worksheets filled with formulas for queries. As the user enters, alters or deletes data tuples in the data worksheets, the formulas in query worksheets automatically compute the actual results of the queries. Thus, the spreadsheet serves as data storage, and executes SQL queries. It is therefore a relational database engine. Consequently, any specification of a database, written in SQL in the form of table and view definitions, can be compiled into a spreadsheet workbook which has exactly the same functionality as if the database was implemented in a classical RDBMS. Crucially, this is achieved without any use of macros written in an external programming language, like Visual Basic or the like. One might consider our construction also as an implementation of a relational database on a completely new type of (virtual) hardware.

As a model of spreadsheet syntax and semantics we take *Microsoft Excel* (TM) (the general reference is [10]), but our constructions work in other similar systems, like *OpenOffice Calc*, *gnumeric* or *Google docs*, too.

We present number of performance tests, which indicate that our constructions are efficient enough to be practically used as a desktop database storing a few thousand rows of data.

## 2.1 Application scenarios

We believe that there is a market niche for a small SQL database implemented in a spreadsheet. It is a low-end solution, in terms of performance. However, there is market for low-end mobile phones, for low-end computers, and similarly there is also market for low-end database solutions.

E.g., Chrome OS does not permit installing new applications. For a user of a netbook with this system, a method to have a small SQL database is to use our solution with the *Google docs* spreadsheet. This might be facilitated by a public "query compiler" in WWW, where the user types in SQL code and gets a couple of spreadsheet formulas to copy and paste into his/her workbook, which implement the functionality he/she needs.

Next, many people and small businesses buy *MS Office* with *Access* to store just a few thousand tuples. With our solution they could buy much cheaper version without *Access* and have principally the same abilities. Of course, they could also use one of the free database solutions, like *PostgreSQL* or *MySQL*. But these RDBMS's do not integrate so easily with other elements of *MS Office*, and require technical skills necessary to install, configure and maintain them. In [9] the authors argue that spreadsheet is actually a very good interface for a database, allowing unexperienced users to create queries stepwise.

Moreover, the limit to a few thousands rows of data imposed by the low efficiency is not very severe for small businesses, which often process data from short periods only (typically 1 month), and after their tax reports are ready, the data can safely be archived. In the spreadsheet database it amounts to saving the finished one and creating a new empty instance for the next period. An alternative option is materialization of queries by *cut* and *paste special→values* which removes all formulas together with their need of recomputation, leaving the results.

## 2.2 Related work

To the best of our knowledge, the problem of expressing relational algebra and SQL in spreadsheets has not been considered before in the setting we adopt here. The the most similar to our work reported here are the following. The paper [9] proposes an extension of the set of spreadsheet functions by carefully designed database function, whereby the user can specify (and later execute) SQL queries in a spreadsheet-like style, one step at a time. These additional operators were executed by a classical database engine running in the background. Our contribution means that exactly the same functionality can be achieved by the spreadsheet itself. Two papers [14, 15] describe a project, later named *Query by Excel* to extend SQL by spreadsheet-inspired functionality, allowing the user to treat database tables as if they were located in a spreadsheet and define calculations over rows and columns by formulas resembling those found in spreadsheets. In the final paper [16] a spreadsheet interface is offered for specifying these calculations, which had to be specified in an SQL-like code in the earlier papers. The paper [8] describes a method to allow RDBMs to query data stored in spreadsheets.

There is also a number of papers which discuss various methods to support high-level design of spreadsheets, in particular [1, 2, 6, 7, 11, 12, 13, 17, 18]. Some of them consider spreadsheets from the functional programming perspective.

# 3. TECHNICALITIES

We assume the reader to be basically familiar with spreadsheets. The paper is written to make the solutions compatible with *Microsoft Excel* (TM) 2003 version. The newest (at the time of this writing) *Excel* 2007 and beta 2010 provide a number of new functions, which simplify some of the tasks, but are not present in other spreadsheet systems. Therefore we chose compatibility with the older version, but refer to and test the newer systems, as well.

## 3.1 `R1C1` notation

We use the *row-column* `R1C1`-style addressing of cells and ranges, supported by *Excel*. This notation is easier to handle in a formal description, although in everyday practice the equivalent `A1` notation is dominating. The key advantage of the `R1C1` notation is that the meaning of the formula is independent of the cell in which it is located.

In the `R1C1` notation, both rows and columns of worksheets are numbered by integers from 1 onward. For arbitrary nonzero integers $i$ and $j$ and nonzero natural numbers $m, n$ the following expressions are cell references in the `R1C1` notation: `RmCn`, `R[i]Cm`, `RmC[j]`, `R[i]C[j]`, `RCm`, `RC[i]`, `RmC`, `R[i]C`.

The number after '`R`' refers to the row number and the number after '`C`' to the column number. If that number is missing, it means "same row (column)" as the cell in which this expression is used. A number written in square brackets is a relative reference and the cell to which this expression points should be determined by adding that number to the row (column) number of the present cell. Number without brackets is an absolute reference to a cell whose row (column) number is equal to that number. For example, `R[−1]C7` denotes a cell which is in the row directly above the present one in column 7, while `RC[3]` denotes a cell in the same row as the present one and 3 columns to the right. If `R` or `C` is itself omitted, the expression denotes the whole column or row (respectively), e.g., `C7` is column number 7. For referencing cells in other worksheets, `RC` may also be used, and references the cell whose row and column numbers are equal to the address of the cell in which this expression is located.

## 3.2 `IF` function

`IF` is a conditional function in spreadsheets. The syntax is `IF(condition,true_branch,false_branch)`. Its evaluation is *lazy*, i.e., after the `condition` is evaluated and yields

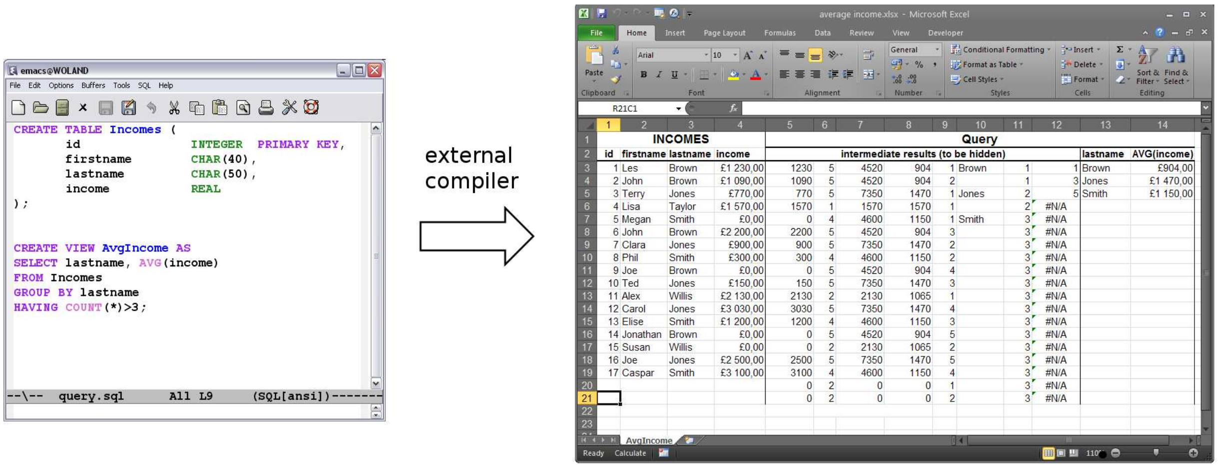


**Figure 1: The idea of a database implementation in a spreadsheet. (Errors appearing in the worksheet are intended.)**

either `TRUE` or `FALSE`, only one of the `branch`es is evaluated. It can be therefore used to protect functions from being applied to arguments of wrong types, trap errors, and, last but not least, to speed up execution of queries by avoiding computing certain branches.

## 3.3 `SUMPRODUCT` function

We will often use a special function called `SUMPRODUCT`. Its uses will be generally modifications of the following two examples.

*Example 1.*
`=SUMPRODUCT((R1C1:R5C1=R1C3)*(R1C2:R5C2=R1C4))` is calculated as follows:
(i) each cell in the range `R1C1:R5C1` is compared with `R1C3`, and this yields a sequence of five booleans;
(ii) each cell in the range `R1C2:R5C2` is compared with `R1C4`, and this yields another sequence of five booleans;
(iii) the two sequences from previous items are multiplied coordinate-wise, which results in automatic data type conversion from booleans to integers, and then normal multiplication;
(iv) the five numbers are summed to give a single number as a result.

Consequently, the final result is the count of rows, in which the columns `C1` and `C2` contain the same pair of numbers as in `R1C3:R1C4`.

*Example 2.*
`=SUMPRODUCT((R1C1:R5C1=R1C3)*(R1C2:R5C2=R1C4)* R1C5:R5C5)` is calculated as follows:
(i-iii) the first three steps of evaluation are the same as before;
(iv) the sequence of 0s and 1s from the previous item and the range `R1C5:R5C5` are multiplied again coordinate-wise, which results in a sequence of five numbers;
(v) again the sum of the above five numbers is returned.

The result is the sum of values in `C5`, calculated over those rows, in which the columns `C1` and `C2` contain the same pair of numbers as in `R1C3:R1C4`.

These two examples generalize to sum-multiplication of more than two or three arrays.

## 3.4 `MATCH` and `INDEX` functions

We use `MATCH` with syntax `MATCH(cell,range,0)`. It returns the relative position of the first value in `range` which is equal to the value in `cell`, and error if that value does not appear there.

In one case we use `MATCH(range,cell,1)`, in which case for `range` sorted into ascending order `MATCH` returns the relative position of the largest value in `range` that is less than or equal to the value in `cell`. We do not apply this form to unsorted ranges.

`INDEX` is used in the following syntax: `INDEX(range,cell)`, and returns the value from `range` whose relative position is given by the value from `cell`.

## 3.5 Syntax variants

We will be using the `SUMPRODUCT` function with whole-column ranges, e.g., `SUMPRODUCT((C1=R1C3)*(C2=R1C4)`, which is generally equivalent to the formula from Example 1, if no other values are present in columns `C1` and `C2`. However, this syntax is *not valid* in *Excel* versions prior to 2007. In early *Excel*'s one would have to use ranges like `R1C5:R1Cmax` in `SUMPRODUCT`, where `max` is the maximal row number of the range. *OpenOffice Calc* generally disallows `R1C1` syntax and whole columns in formulas. *Gnumeric* and *Google docs* in turn permit both.

Our formulas are designed to work in all above systems, but in some of them they require syntactical modifications, mainly switching to `A1` and eliminating whole-column ranges.

In *Excel* 2007 and 2010 there are new functions `COUNTIFS` and `SUMIFS`, which can substitute `SUMPRODUCT` in our applications, and are much more efficient. E.g., the first of our example formulas can be replaced by
`=COUNTIFS(R1C1:R5C1,R1C3,R1C2:R5C2,R1C4)`,
and the second by
`=SUMIFS(R1C5:R5C5,R1C1:R5C1,R1C3,R1C2:R5C2,R1C4)`.

# 4. ARCHITECTURE OF A DATABASE IMPLEMENTED IN A SPREADSHEET

In this paper, we disregard a number of minor issues arising in practical implementation of database operations in a spreadsheet. First of all, there is the obvious limitation of size on then number and sizes of relations, views and their intermediate results, imposed by the maximal available number of worksheets, columns and rows in the spreadsheet system at hand. Next, the size of the data values (integers, strings, etc.) is also limited. The variety of data types in spreadsheets is also restricted when compared to database systems.

The overall architecture of a relational database implemented in a spreadsheet is as follows.

Given specification of the database, an implementation of a database is created by an external program (which plays the role of query compiler), in the form of an `.xls`, `.xlsx`, `.odc`, etc., file.

The whole resulting database is a workbook, consisting of one worksheet per data table and one worksheet per view in the database.

The data table worksheets are where the data is entered, updated and deleted. In the case of the (more theoretical in flavor) implementation of the relational algebra, the data table sheets do not contain any formulas and are simply the place to enter tuples into relations. In the case of SQL implementation, the cells are equipped with data validation formulas, which perform data type verification, enforce `PRIMARY KEY`, `FOREIGN KEY` and other integrity constraints included in the `CREATE TABLE` statements.

The query (view) worksheets are not supposed to be edited by the user. They contain columns filled with formulas, which calculate the consecutive values of the result of the query. Besides the result columns of the query, the view worksheets can also contain a number of hidden columns, which calculate and store intermediate results emerging during query evaluation. It is important that the formulas are completely uniform in each column of the database workbook, and they do not depend on the data which will be stored in the application. Initially all formulas compute the empty string `""` value, representing unused space. When the user manually enters data into the tables, the automatic recomputation of the spreadsheet causes the results of queries to be computed and appear in the view worksheets.

# 5. THEORETICAL LEVEL: RELATIONAL ALGEBRA

We assume the semantics over a fixed domain of (the spreadsheet's implementation of) integers, so that a relation is a set or multiset of tuples over the integers that are implemented in the spreadsheet software.

## 5.1 Compositionality

We assume the unnamed syntax for the relational algebra: relations and queries have columns, which are numbered and do not have any names. Sometimes we consider the expressions `C1`, `C2`, etc., as the names of the worksheet columns, as well as the names of the columns in relations.

The representation of a relation $r$ of arity $n$ is a group of $n$ consecutive columns in a worksheet, whose rows contain the tuples in the relation. The rows in which there are no tuples of $r$ are assumed to be filled with the empty string formula `=""`, evaluating to the empty string value `""`, which the user can replace by the new tuples of the relation. The empty string is never a component of a tuple in a relation or query. Therefore either all cells in a row contain the empty string, or none does. The rows of tables and queries evaluating to empty strings are called *null rows* henceforth.

The assumption that `=""` formulas fill the empty rows of data tables is only for uniformity of presentation. A formula in a cell can not evaluate to "empty cell" (because the formula occupies that cell anyway), only to empty string. Therefore, if blank cells were used in empty rows, formulas expressing queries would have to be adapted to accept unused space in two different forms: empty cells in data tables, and empty strings in results of other queries.

The representation of a relational algebra query $Q$ of arity $m$ is a group of $l + m$ consecutive columns in a worksheet. Initial segments of all its rows are filled with formulas (identical in all cells of each column), which calculate the tuples in $Q$. We assume that the formulas in the last $m$ columns should return either (a component of) a tuple in the result of $Q$, or the empty string value `""`. The additional $l$ columns are also filled with identical formulas, which calculate intermediate results. A worksheet of this kind can be created by entering the formulas in the first row, and then *filling* them downward to span as many rows as necessary. This uniformity assumption means in particular, that the formulas are completely independent of the data they will work on. However, we would like to stress that there is no reason to reject nonuniform implementations, should they prove to be more effective or permit expressing queries inexpressible in a uniform way.

In the following we will consider both set and bag (multiset) semantics of the relational algebra. In the first case, duplicate rows are not permitted in the relations and queries; in the latter they are permitted. However, even in the set semantics a spreadsheet representation of a relation may contain many null rows.

Furthermore, the representation may be *loose* if null rows are interspersed with the tuples, or *standard* if all the tuples come first, followed by the null rows.

Consequently, we have loose-set, loose-bag, standard-set and standard-bag semantics. No matter which of the above semantics we have in mind, the result of the query appears exactly as if it were a table, and can be used as such. Now the only thing necessary to compose queries is to locate their implementations side by side in a single worksheet and change input column numbers in the formulas computing the outermost query, to agree with the column numbers of the outputs of the argument queries (and then the output columns of the argument queries become the intermediate results columns of the composition).

Therefore, queries represented in this way are compositional.

Now it suffices to demonstrate that the each of the following relational algebra operators from [4] can be implemented in a spreadsheet:

- Two operations specific to spreadsheets, absent in [4]: error trapping and standardization (converting to standard form).
- Sorting.
- Duplicate removal $\delta r$.
- Selection $\sigma_\theta r$.
- Projection $\pi_{i,j,\ldots} r$.

- Union $r \cup s$.
- Difference $r \setminus s$.
- Cartesian product $r \times s$.
- Grouping with aggregation $\gamma_L r$, where $L$ is one of: `SUM`, `COUNT`, `AVG`, `MAX` and `MIN`.

## 5.2 Notation

We present here implementations based on spreadsheet functions which are common to most of (or even all) spreadsheet systems. This way we believe to consider the *spreadsheet paradigm*, even if its definition is not yet formulated in the literature.

We use the following convention for presenting implementations:

`COLUMNS < =FORMULA`

means that `=FORMULA` is entered into each cell of the `COLUMNS`, which may be specified either to be a single column (e.g. `C5` ) or a range of a few columns (e.g. `C5:C7`), or a single cell (e.g. `R1C5`), and in each case belongs to the columns with intermediate values. The notation

`COLUMNS ≪ =FORMULA`

indicates that formulas located in `COLUMNS` calculate the output of the query.

Sometimes the output columns are not specified, and then it is always indicated, that the final output is computed by applying another, already defined operation to some of the columns with intermediate results. At two occasions we even write SQL queries as contents of spreadsheet columns, understanding that the spreadsheet formulas implementing them are written there.

Generally, we assume the arguments of the algebra operators to be two- or three-ary relations or queries, the generalization to higher arities is straightforward.

Except for the standardization and sorting, in all other cases we assume the input to be in the standard form, i.e., null rows at the bottom.

## 5.3 Error trapping and standardization

In this section we describe two special purpose operators, which perform common and useful tasks, specific to our spreadsheet environment.

### 5.3.1 *Error trapping*

If we replace a formula `=F`, which may produce an error, by the formula `=IF(ISERROR(F),"",F)`, any error produced by `=F` is replaced by the empty string, and otherwise the value is the same as the value of `=F`.

### 5.3.2 *Standardization*

This operation converts a relation from loose to standard form, moving null rows to the bottom. The relative order of non-null rows is preserved. We assume that columns `C1` and `C2` contain the source data.

`C3 < =SUMPRODUCT((R1C1:RC1<>"")*1)`

counts the non-null rows above the present row, including the present one. This number is the row number to which the present row should be relocated. Note that multiplication by 1 enforces boolean to integer conversion.

`C4 < =MATCH(ROW(),C3,0)`

returns the position of the first value in `C3` equal to the present row number. If no match is found (i.e., we are in a row whose number is higher than the total number of non-null rows), an error is returned.

`C5:C6 ≪ =IF(ISERROR(RC4),"",INDEX(C[-4],RC4))`

Errors are trapped, and when there is no error, `INDEX` returns the data from the suitable row of `C[-4]`. Thus the values from `C1:C2` get relocated to their positions calculated in `C3`.

## 5.4 Sorting

Now we describe an implementation of sorting, which is a generalization of standardization. We assume that columns `C1` and `C2` contain the source data and we sort in ascending order by the values in `C1`.

`C3 < =SUMPRODUCT((C1<=RC1)*1)`

This puts in `RiC3` the number of entries in column `C1` which are smaller than or equal to `RiC1`. `""` compared by `<=` is larger than any number, so null rows do not give any errors, and in the following are treated as the largest entries.

`C4 < =RC3-SUMPRODUCT((C1<=RC1)*1)+1`

Now in `RiC4` is the number of entries in column `C1` which are either smaller than `RiC1` or equal to it and located in the same row or above it. This is the number of the row into which `RiC1` should be relocated during sort.

`C5:C6 ≪ =INDEX(C[-4],MATCH(ROW(),C4,0))`

This part is very similar to the standardization solution, except that there are no errors to be trapped and we combine two formulas into one.

Sorting in descending order is done by reverting the signs of numbers by the formula `=IF(RC[-1]="";"";-RC[-1])` and sorting into ascending order, and then reverting the signs again. This leaves the null rows at the bottom. In particular, if sorting is necessary there is no need to standardize first.

An important property of this operation is that rows with empty string in the column on which the sort is performed, are moved to the bottom. Consequently, sorting brings any query or relation to standard form. Moreover, this form of sorting is stable.

## 5.5 Duplicate removal

Next we describe the implementation of duplicate removal, which, among other things, converts its input data from bag to set semantics. We assume the table to contain two columns `C1:C2`.

`C3 < =SUMPRODUCT((C1=RC1)*(C2=RC2))`

This causes `RiC3` to contain the number of tuples from `C1:C2` which are equal to `RiC1:RiC2` and are located at the same level or above it. This number is 1 iff the row contains the first occurrence of this tuple.

`C4:C5 ≪ =IF(RC3=1,RC[-3],"")`

Now the first occurrences of tuples are copied into `C4:C5`, the other are replaced by null rows.

## 5.6 Selection

Assume that we are given a relation $r$ located in `C1:C2` and we want to compute $\sigma_\theta r$, where $\theta$ is a boolean combination of equalities and inequalities concerning the values of columns of $r$ and constants. Then we use a spreadsheet formula expressing $\theta$ to substitute "" for the rows which do not satisfy $\theta$. This is best explained on an example: if $\theta$ is $(\texttt{C1} \leq 100 \wedge \texttt{C2} > \texttt{C1}) \vee \texttt{C2} \neq 175$, then the selection is implemented by

```
C3:C4 ≪ =IF(OR(AND(RC1<=100,RC2>RC1),
        RC2<>175),RC[-2],"")
```

It leaves the result of the selection in a loose (set or bag, inherited from the input) form.

## 5.7 Projection

The case of projection is quite easy: it amounts to omitting some columns from the input relation/query.

## 5.8 Union

Assume that we are given two relations located in `C1:C2` and `C3:C4`, respectively.

Then use the following formulas to calculate their union in standard bag form.

```
R1C5 < =COUNT(C1)
```

This is the number of non-null rows in `C1`.

```
C6:C7 ≪ =IF(ROW()<=R1C5,RC[-5],
        INDEX(C[-3],ROW()-R1C5))
```

If the present row number is less than `R1C5` then we take the same row from `C1:C2`, otherwise we take rows from `C3:C4` whose numbers are suitably shifted. Note that this assumes the inputs to be in standard (set or bag) form.

## 5.9 Difference

Assume that we are given two relations located in `C1:C2` and `C3:C4`, respectively. Then use the following formulas to calculate their set difference.

```
C5 < =SUMPRODUCT((C3=RC1)*(C4=RC2))
```

This calculates in `RiC5` the number of times a tuple equal to `RiC1:RiC2` appears in `C3:C4`.

```
C6:C7 ≪ =IF(RC5=0,RC[-5],"")
```

Now if `RiC5` is 0, we copy the row `RiC1:RiC2` to the output, otherwise we replace it by a null row.

The set/bag form of the result is inherited from the inputs. However, this construction **does not** work for the bag format, since in this case we should count the copies of identical rows in both relations and put in the output a suitable number of such rows.

The more complicated construction which does work is as follows:

```
C5 < =SUMPRODUCT((C3=RC1)*(C4=RC2))
```

This, exactly as before, calculates in `RiC5` the number of times a tuple equal to `RiC1:RiC2` appears in `C3:C4`.

```
C6 < =SUMPRODUCT((R1C1:RC1=RC1)*(R1C2:RC2=RC2))
```

Now we calculate in `RiC6` the number of times a tuple equal to `RiC1:RiC2` appears in `C1:C2` in row $i$ or above it.

```
C7:C8 ≪ =IF(RC5>=RC6;"";RC[-6])
```

Now we replace by null rows the first `RiC5` occurrences of tuple `RiC1:RiC2`, and leave unaffected the remaining ones, which gives the desired bag difference. The resulting relation is loose.

## 5.10 Cartesian product

Assume that two relations are located in `C1:C2` and `C3:C4`, respectively.

Then use the following formulas to calculate their Cartesian product. The construction below works only for relations in standard form, otherwise standardization is necessary first.

```
R1C5 < =COUNT(C1)
R2C5 < =COUNT(C3)
```

We calculate the numbers of non-null rows in `C1:C2` and `C3:C4`.

```
C6:C7 ≪ =IF(ROW()<=R1C5*R2C5,INDEX(C[-5],
        INT((ROW()-1)/R2C5)+1),"")
```

creates `R1C5` blocks, the $i$-th block being `R2C5` copies of `RiC1:RiC2`.

```
C8:C9 ≪ =IF(ROW()<=R1C5*R2C5,INDEX(C[-5],
        MOD(ROW()-1,R2C5)+1),"")
```

repeats in circular fashion the consecutive rows of `C3:C4` a total of `R1C5` rounds.

Note that in this case, the set or bag form of the initial relations is inherited by their product.

## 5.11 Grouping with aggregation

In the following, we assume always the relation to be located in `C1:C3`, grouping done over `C1:C2` and aggregation over `C3`.

### *5.11.1* `GROUP BY` *with* `SUM`,`COUNT` *and* `AVG`

```
C4 < =SUMPRODUCT((C1=RC1)*(C2=RC2)*C3)
```

This array formula computes in `RiC4` the sum of all `RjC3` over all $j$ such that `RjC1:RjC2` is equal to `RiC1:RiC2`. Now we do duplicate elimination over `C1:C2` and `C4` and that is the desired result. Alternatively, with

```
C4 < =SUMPRODUCT((C1=RC1)*(C2=RC2))
```

we get the count instead of sum aggregation.

For average one has to compute `GROUP BY` with `SUM` and `COUNT` side-by-side and return the copy of `C1:C2` plus the sum column divided by the count column.

### *5.11.2* `GROUP BY` *with* `MAX` *and* `MIN`

Let us consider `MAX`, the other being handled symmetrically. First, the whole relation is sorted into descending order by `C3`. On the result, elimination of duplicates is performed, which considers two rows identical when they agree on `C1:C2`. Our implementation of this operation eliminates all occurrences of a tuple except the very first one. The one left is therefore accompanied by the maximal value of `C3`, as desired.

## 5.12 Additional operators

Although semijoin and join are not on the list of algebra operators, they are extremely inefficient if implemented as a selection of a Cartesian product. Therefore we present more effective implementations of equisemijoin and equijoin.

### *5.12.1 Semijoin*

Assume that we are given two relations located in `C1:C2` and `C3:C4`, respectively, and we wish to compute the equisemijoin $\texttt{C1}:\texttt{C2} \ltimes_{\texttt{C1}=\texttt{C3}} \texttt{C2}:\texttt{C3}$.

This is achieved in the following way. The formulas

```
C5 < =IF(ISERROR(MATCH(RC1,C3,0),"",RC1)
C6 < IF(RC5="","",RC2)
```

empty the rows from C1:C2 which do not belong to the semijoin. The resulting relation is loose.

### *5.12.2 Join*

Let two relations be located in C1:C2 and C3:C4, respectively, and the equijoin $\mathtt{C1:C2} \bowtie_{\mathtt{C1=C3}} \mathtt{C2:C3}$ should be computed.

We distinguish two cases here. First, that the tables C1:C2 and C3:C4 represent a many-to-many relationship. The second, that C3 is a foreign key for C1:C2, so the relation is many-to-one. The former is much more general, the latter is much more frequent in normalized databases.

*Many-to-many.*

Our implementation utilizes a kind of index on the data tables. It is assumed that the tables are sorted on the C1 and C3 columns, respectively. Then let us use the following formulas:

```
C5:C6 < =SELECT C1,COUNT(*) FROM C1:C2 GROUP BY C1
C7 < =IF(RC5="","",R[-1]C+RC6)
R1C7 < =IF(RC5="","",1)
```

The last two lines should be understood that we first fill the whole column C7 with formulas, and then change the formula in its first cell. The columns C5:C7 provide the index, where each key from C1 is associated with its number of occurrences and the location of its first occurrence in this column. An analogous index for C3:C4 is created in columns C8:C10.

Now we create standardized semijoins

C11:C13 < $\mathtt{C5:C7} \ltimes_{\mathtt{C5=C8}} \mathtt{C8}$
C14:C16 < $\mathtt{C8:C10} \ltimes_{\mathtt{C8=C10}} \mathtt{C5}$

which eliminate the elements from the index which will not appear in the join. The next steps are

```
C17 < =IF(RC11="";"";RC12*RC15)
C18 < IF(ISERROR(R[-1]C+R[-1]C17);"";
      R[-1]C+R[-1]C17)
R1C18 < 0
R1C19 < =MAX(C18)
```

The values in C17 are the sizes of the fragments of the join, corresponding to the consecutive values of the columns on which the join is computed. C18 contains the numbers of rows at which they should begin minus 1. Finally, R1C19 contains the cardinality of the join. Finally,

```
C20 < =IF(ROW()>R1C19,"",MATCH(ROW()-1,C18,1))
C21 < =IF(RC20="","",IF(RC20<>R[-1]C20,1,1+R[-1]C))
R1C21 < =IF(RC20="","",0)
C22 ≪ =IF(RC20="","",INDEX(C11,RC20))
C23 ≪ =IF(RC20="","",INDEX(C2,INDEX(C13,RC20)+
       MOD(RC21,INDEX(C12,RC20))))
C24 ≪ =IF(RC20="","",INDEX(C4,INDEX(C16,RC20)+
       INT(RC21/INDEX(C12,RC20))))
```

compute the join doing Cartesian products of fragments of initial tables corresponding to the same entries in the indexes. The natural two-level addressing using nested INDEX functions is employed, but otherwise the formulas resemble those used for Cartesian product above.

In C20 function MATCH is used with the last parameter 1 to do inexact search for the fragment of the initial tables from which the tuple in join should originate (see section 3.4).

In C21 the consecutive numbers of rows with the same value of the key are calculated using dynamic programming rather than aggregation.

*Many-to-one.*

In the following solution MATCH finds the position of the foreign key and INDEX fetches the tuple from that row.

```
C5:C6 ≪ =IF(RC1="","",RC[-4])
C7:C8 ≪ =IF(RC1="","",INDEX(C[-3],MATCH(RC1,C3,0)))
```

## 6. PRACTICAL LEVEL: SQL

This part is devoted to the discussion of the implementation issues of SQL-92. It is less detailed than the previous section, and is more dependent on the particular properties of *Excel*.

Of the three parts of SQL: DDL, DML and DCL, that last one is irrelevant, since we construct a database for a single user.

### 6.1 NULL values

NULLs can be represented simply by the string NULL and handled as such. This is not difficult, but rather tedious, since all the formulas, whether implementing DDL or DML statements, must be adjusted to handle NULLs by introducing conditional IFs which test if the argument is a NULL and invoke either a special treatment of NULL or the standard formula for non-NULLs.

### 6.2 DDL

Let's discuss DDL, i.e., mainly CREATE TABLE statements. We adopt the option to distinguish the data table from its input area. So for each CREATE TABLE statement we create a separate data table and a separate input table. The latter is indeed a query table (see below for details), which filters tuples which do not satisfy integrity constraints included in the DDL statement and displays a warning message for the user. The former then fetches the rows which satisfy the integrity constraints (by merely looking if there is a warning message or not), and does standardization.

We assume the user enters data elements adding them at the bottom. If elements are removed (simply using the DEL key), no new elements are added at their positions. Updates are performed by removing the old version of the tuple and immediately adding a new one at the bottom.

Function TYPE allows one to distinguish text, booleans and numbers, which, together with length function LEN for strings and inequalities for numbers allow one to enforce data type declarations. There are a few limitations to this rule, e.g., the empty string "" plays a special role in our implementation of relational algebra operators, and so does the string "NULL", which imposes a (mild) restriction on what kind of strings can be used. For the DATE statement, however, one has to use formatting, instead, which enforces numbers to be interpreted and displayed as dates.

UNIQUE and PRIMARY KEY are enforced by the duplicate elimination operator described above, which rejects tuples which have already appeared before.

FOREIGN KEY statements are enforced by a semijoin query. It does not seem that there is an easy method to implement policies concerning behavior of the database when one deletes a foreign key for a tuple, except the CASCADE option. This one is completely automatic: when the foreign key disappears, the tuples which reference it become illegal and disappear from the data table (because the formulas which transfer them to the data table return "" in the absence of the foreign key), even though they remain in the input table (where a warning appears).

Concerning `INDEX`, we have already presented a kind of index in our implementation of equijoin.

*Example 3.* Let us consider the following DDL statement:

```
CREATE TABLE Ord(
 Id INT UNSIGNED NOT NULL PRIMARY KEY,
 ModelID INT NOT NULL REFERENCES Models (ModelID),
 Version SMALLINT,
 ModelDescrip VARCHAR(40));
```

We assume the following:

- the input worksheet for the above table is `OrdInput`, and the worksheet of the data table is `OrdData`;
- worksheet `ModelsData` keeps in column `C1` the primary key referenced to above;
- size limits for `INT` and `SMALLINT` are $M$ and $N$, respectively;
- the limit for the number of rows in data tables is `max`.

The following formula is placed in column `C5`:

```
=IF(AND(RC1="",RC2="",RC3="",RC4=""),"",
   IF(OR(RC1="",RC2="",RC3="",RC4=""),
   "Invalid data",
   IF(
    AND(
     IF(TYPE(RC1)=1,INT(RC1)=RC1,FALSE),
     RC1>=0,RC1<=M,
     COUNTIF(R1C1:RC1,RC1)=1,
     COUNTIF(ModelsData!C1,RC2)=1,
     IF(TYPE(RC3)=1,
      AND(INT(RC3)=RC3,ABS(RC3)<=N),
      RC3="NULL"),
     TYPE(RC4)=2,LEN(RC4)<=40),
   "","Invalid data")))
```

The formula is a big `IF`, which behaves as follows: it returns an empty string (numbers in parentheses refer to the lines in the formula above): when the row is a null row (1), and otherwise an error message if at least one (but not all) of its fields is `""` (2), and otherwise again `""` if all of the following conditions hold:

(5-6) The first column contains a number whose integer part is equal to itself, is nonnegative and does not exceed $M$ (note that we used `IF` – due to its lazy evaluation `INT` is never applied to non-numbers and does not give any error message).

(7) `RC1` appears for the first time in its column.

(8) `RC2` appears exactly once in the table `ModelsData` in the first column (assumed to contain the primary key of that relation) and in this branch of the initial `IF`s it is not `""`. Note that it is not necessary to verify that `RC2` is a nonnegative integer in the specified range, because it is enforced in the foreign key table, so the count takes care of it.

(9-11) If `RC3` is a number then it must be an integer whose absolute value does not exceed $N$, and if not number, then it is `"NULL"`.

(12) `RC4` is a text, whose length is in the specified range. Function `LEN` accepts numbers as inputs, so there is no need to protect it by `IF`.

Then the `OrdData` worksheet contains formulas

```
=IF(OrdInput!RC5="Invalid data","",OrdInput!RC)
```

in all its four columns. This gives a possibly loose data table, which can be standardized, if desired.

As it can be seen from the example, the actual translation of the `CREATE TABLE` statements can be quite complicated. The main reason is that now we have many data types and some of the functions must be prevented from being applied to arguments of wrong type, `NULL`s may show up, etc.

### 6.3 Transactions

A database expressed as a spreadsheet in general cannot be accessed concurrently, hence there is no need of transactions for concurrency control. Transactions for recovery are supported in the following form: `COMMIT` is executed by saving the current state of the workbook, while `ROLLBACK` is done by re-reading the spreadsheet from disk. "Undo" (Ctrl-Z) offers a limited but easy to use form of `ROLLBACK`.

## 7. PERFORMANCE

Unfortunately, *Excel* and other spreadsheets have not been designed to serve as database engines, so we can not expect very good performance of our implementations. Formulas with ranges as arguments generally always do linear scans, and they are used in a linear number of cells. Recomputation of cells, whether invoked automatically or manually, always applies to all of them, so they produce quadratic algorithms. Of course, there are still possibilities to get some improvement (at least of the constants), by using dynamic algorithms, which compute the values in cells accessing only a few neighboring cells, or exploit the lazy evaluation of `IF` statements to prune the computation trees significantly. This area is largely unexplored, as the whole problem of optimization queries to be executed in a spreadsheet.

Apart from reducing the cost of operations, the other important possibility is to reduce recomputation. Namely, some of the systems (including *Excel* again) permit references not only to other worksheets, but to other workbooks (i.e., files), too. This gives the possibility to locate each query in a different file and open it only when it is necessary. It is then recomputed, but other queries are not. In particular, when working with data tables no queries need to be open. A similar solution is to work with tables and queries located in worksheets of one workbook, but with automatic recomputation turned off. Instead, as an act of computing a query, the user manually orders recomputation only of the currently active worksheet (Shift-F9 in *Excel*).

### 7.1 Tests

The tests reported below are not intended to give a complete account of the performance of our implementation of SQL in a spreadsheet, nor to suggest methods of optimization. Our main goal is to indicate that for the basic operators and a simple but already useful query it is possible to store and process a few thousand tuples reasonably fast, and thus that our solution has the potential of being useful in practice.

The experiments were conducted using beta version of *Excel* 2010 running on Intel(R) Core(TM)2 Duo CPU at 2.40GHz, on a laptop with 2 GB RAM and Windows XP Professional SP3. The testing procedures were programmed

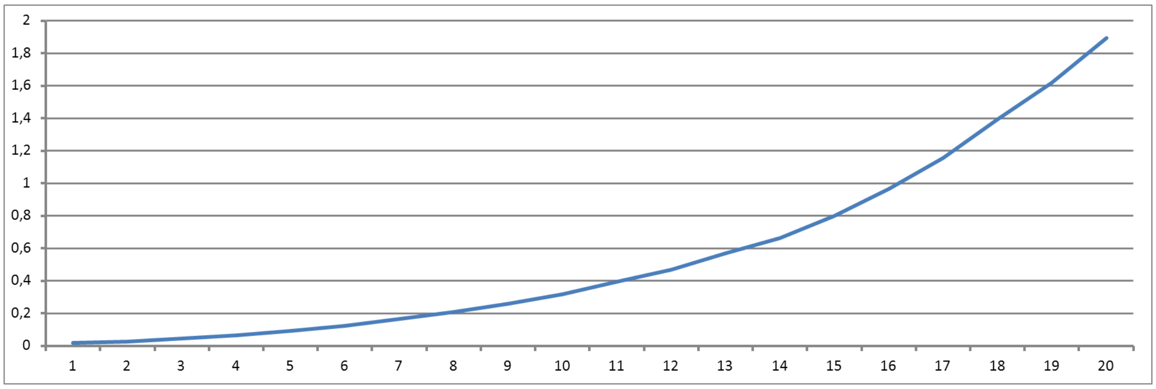


**Figure 2: Performance graph of standardization. Number of tuples in the table is given in thousands, the number of rows with formulas is always equal to the number of tuples.**

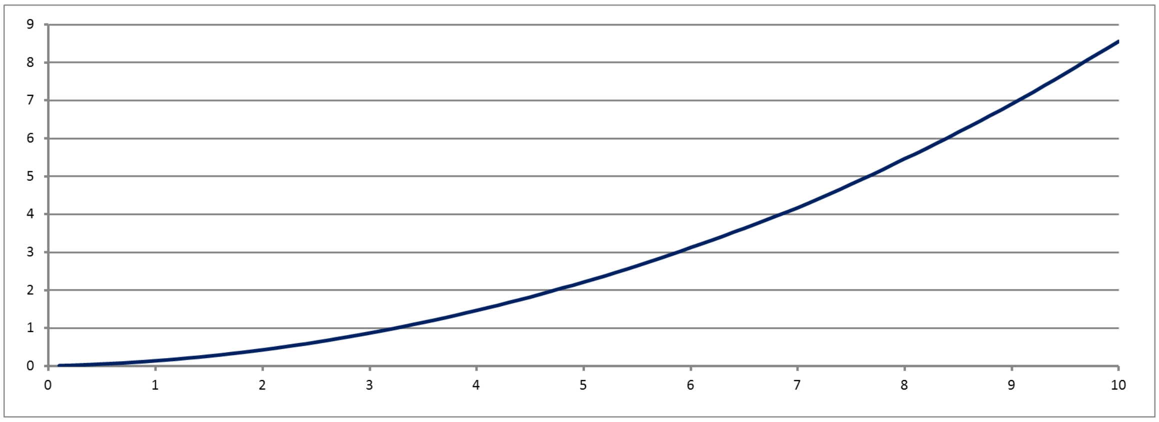


**Figure 3: Performance graph of sorting. Number of tuples given in thousands, number of rows with formulas is equal to the number of tuples.**

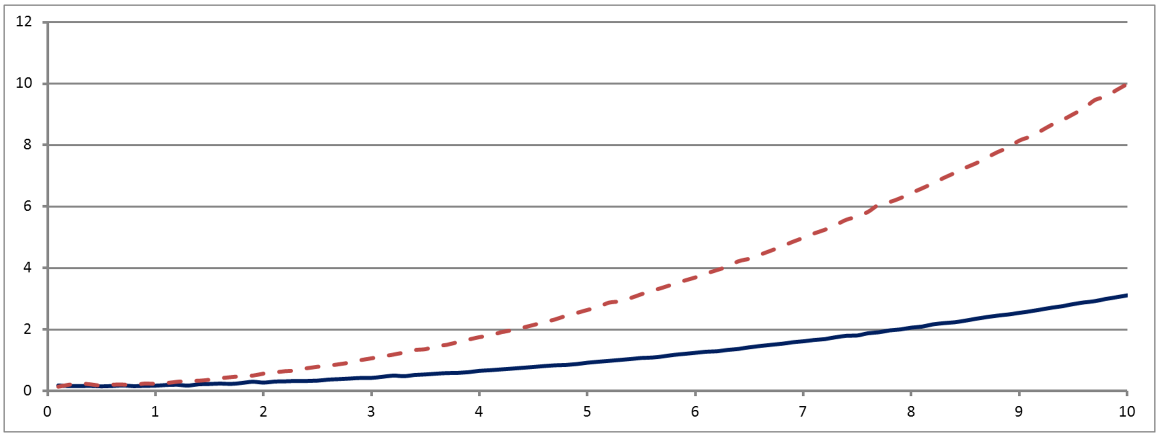


**Figure 4: Performance graph of duplicate elimination operator over a two-column table. There are 10000 rows of formulas in all cases, the number of tuples in the table is given in thousands. The graphs illustrate computation time for the solution with `SUMPRODUCT` (broken line) and for the solution with `SUMIFS` (solid line) in seconds.**

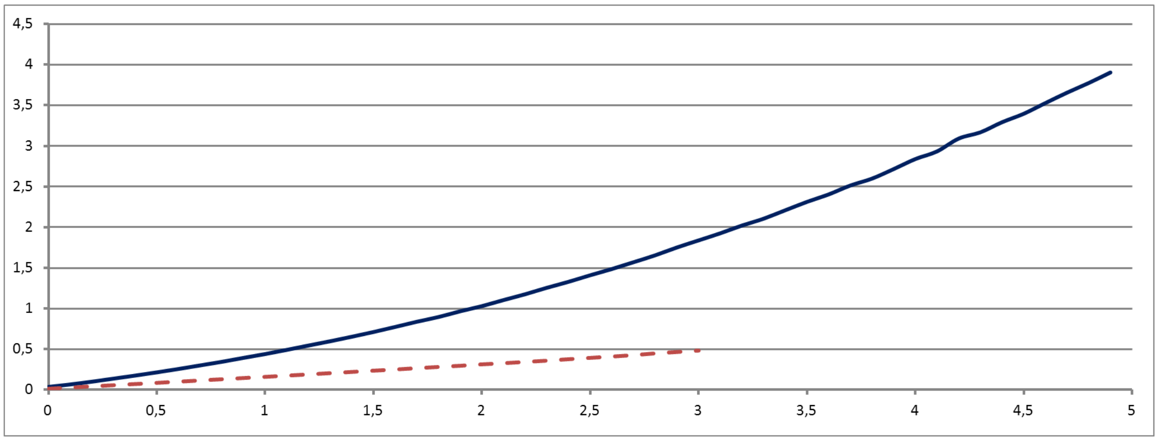


**Figure 5: Performance graph of bag difference operator for two tables of identical size. Number of tuples in the table is given in thousands, the number of rows with formulas is always equal to the number of tuples. Time for the solution with `SUMPRODUCT` (broken line) is given in *minutes*, time for the solution with `SUMIFS` (solid line) is given in seconds.**

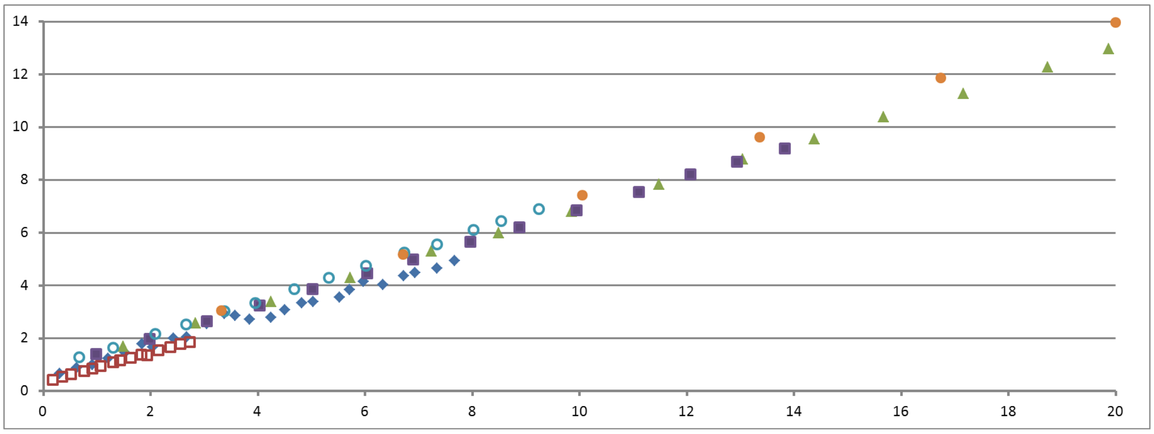


**Figure 6: Performance graph of a many-to-many equijoin. The horizontal axis represents cardinality of the join in thousands, the vertical one computation time in seconds. The data points come for six experiments with 10000 or 20000 rows of formulas in the query table, and sizes of joined relations ranging from 300 to 1000.**

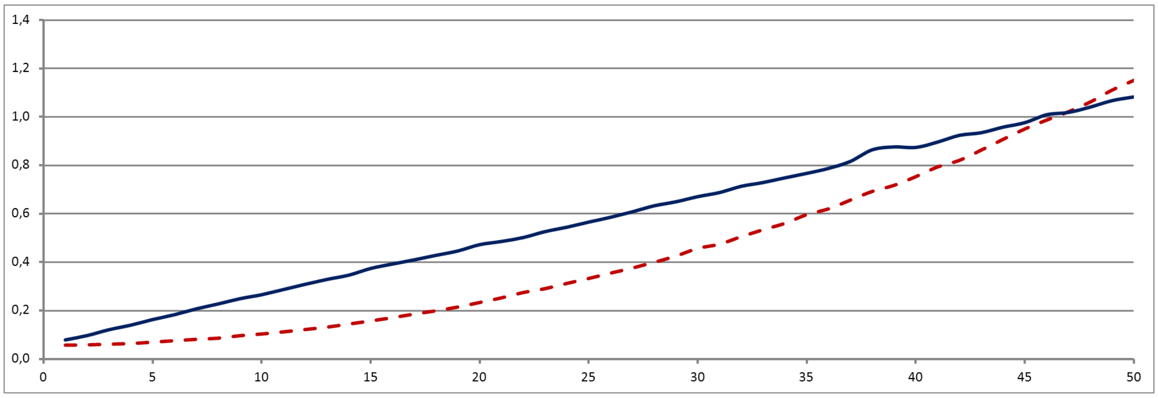


**Figure 7: Performance graph of a many-to-one equijoin. The worksheet consists of 50000 rows of formulas, number of tuples in the table is given in thousands, the size of the table with foreign key is 5000 tuples (solid line) or 10% of the number of tuples in the first table (broken line).**

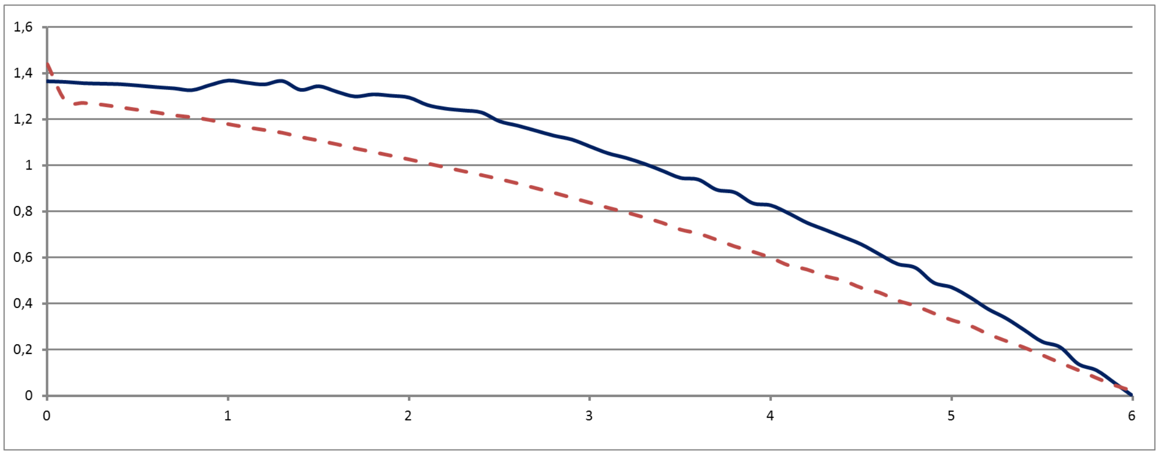


**Figure 8: Cost of a deletion or update under automatic recomputation, for table from Example 3 with 6000 rows (broken line) and 60000 rows (solid line), depending on the row in which this operation is performed (counted in thousands and tens of thousands, respectively). Time for the smaller table is given in seconds, for the larger one in *minutes*.**

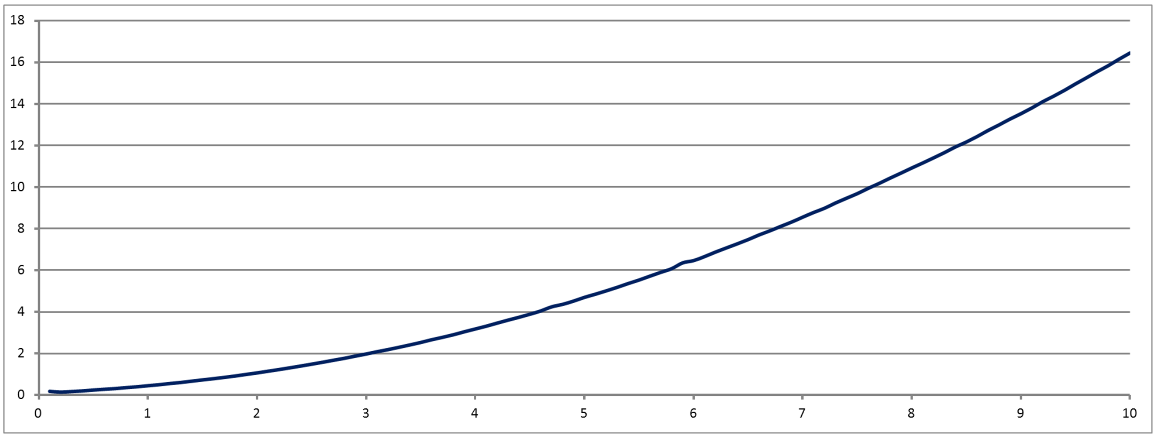


**Figure 9: Cost of computing query `SELECT lastname, AVG(income) FROM Incomes GROUP BY lastname HAVING COUNT(*)>3` from Figure 1, for table with 10000 rows of formulas. The number of tuples in the table is given in thousands.**

in VBA. In general, during computation of the queries we tested both cores of the processor were fully active, but it remains to be verified how to implement queries to achieve optimal scalability with the increasing number of cores and/or processors. The amount of available memory was never a problem.

Descriptions of the charts are generally informative enough, and the operations correspond to those already described above. Additional explanations are necessary in a few cases, which are treated below.

We do not claim the the findings of this section carry over to other spreadsheets systems.

### *7.1.1 Insertions, deletions, updates*

In automatic recomputation mode *Excel* recomputes only those formulas, which refer to the cells which have changed. We tested the implementation of the `CREATE TABLE` statement described in Section 6.2, whose formulas do not refer to cells below in the same table, only to those which are above. Hence insertion (always at the bottom of the table) evaluates a single formula only and, according to our measurements, takes about 20 to 30 milliseconds, almost irrespectively of the size of the table. However, the cost of a deletion or an update depends on the position at which it is performed, as it causes recomputing of all formulas which are located below, too. This is illustrated on the chart. The table with foreign keys had 1000 rows in all tests.

### *7.1.2 Join*

The unusual shape of the chart for many-to-many join has been chosen to illustrate the fact, that the computation time depends linearly on the size of the resulting join, and the other factors (sizes of the joined tables, size of the query table with formulas) are relatively insignificant.

## 7.2 `SUMIFS` vs. `SUMPRODUCT`

In two cases we tested the alternative implementations, using `SUMIFS` instead of `SUMPRODUCT` function. The results are clear: solutions with the former are reasonable, while formulas using the latter are extremely slow and generally impractical—on one of the charts, we had to use minutes instead of seconds to measure the length of the computation.

## 8. CONCLUSIONS, FURTHER RESEARCH

We have demonstrated that relational algebra can be naturally expressed in a spreadsheet, thus showing the power of the spreadsheet paradigm, which subsumes on the theoretical level the paradigm of relational databases. This can be understood as an implementation of a relational database on a completely new type of (virtual) hardware. Of course, in practice the effectiveness of this database is rather low, but our test demonstrate that it has the potential to be useful as an end-user desktop database with a few thousand rows of data.

The following problems questions seem worth exploring:

- Develop a methodology to optimize SQL queries executed in a spreadsheet.
- Can spreadsheets execute queries not expressible in SQL-92? In particular, can spreadsheets execute recursive queries, like those `WITH ...SELECT` in SQL-99, or those in Datalog?
- Our implementations of SQL queries use uniform spreadsheets, in which all rows of a query table are identical. Could nonuniformity help in improving performance, or even allow expressing more queries?
- Can spreadsheets naturally implement other models of databases, like semi-structured or object-relational ones?

## 9. ACKNOWLEDGMENTS

I would like to thank Jan Van den Bussche for many valuable discussions and advices, Krzysztof Stencel for encouragement, the anonymous referees for pointing out weak points of the previous version of the paper and suggesting improvements.

## 10. REFERENCES

[1] R. Abraham and M. Erwig. Type inference for spreadsheets. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 73–84, New York, NY, USA, 2006. ACM.

[2] M. M. Burnett, J. W. Atwood, R. W. Djang, J. Reichwein, H. J. Gottfried, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *J. Funct. Program.*, 11(2):155–206, 2001.

[3] E. J. Chesler, S. L. Rodriguez-Zas, J. S. Mogil, A. Darvasi, J. Usuka, A. Grupe, S. Germer, D. Aud, J. K. Belknap, R. F. Klein, M. K. Ahluwalia, R. Higuchi, and G. Peltz. In silico mapping of mouse quantitative trait loci. *Science*, 294(5551):2423, 2001. In Technical Comments.

[4] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation.* Prentice-Hall, 2000.

[5] A. Grupe, S. Germer, J. Usuka, D. Aud, J. K. Belknap, R. F. Klein, M. K. Ahluwalia, R. Higuchi, and G. Peltz. In silico mapping of complex disease-related traits in mice. *Science*, 292(5523):1915–1918, 2001.

[6] S. P. Jones, A. Blackwell, and M. Burnett. A user-centred approach to functions in Excel. In *ICFP '03: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 165–176, New York, NY, USA, 2003. ACM.

[7] M. Kassoff, L.-M. Zen, A. Garg, and M. Genesereth. PrediCalc: a logical spreadsheet management system. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 1247–1250. VLDB Endowment, 2005.

[8] L. Lakshmanan, S. Subramanian, N. Goyal, and R. Krishnamurthy. On querying spreadsheets. *ICDE '98: Proceedings of the 1998 IEEE International Conference on Data Engineering*, pages 134–141, Orlando, FL, USA, 1998. IEEE Computer Society.

[9] B. Liu and H. V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 417–428, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Microsoft Corporation. Excel Home Page - Microsoft Office Online. `http://office.microsoft.com/en-us/excel/default.aspx`, accessed 20/10/2009.

[11] R. Mittermeir and M. Clermont. Finding high-level structures in spreadsheet programs. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering*, page 221, Washington, DC, USA, 2002. IEEE Computer Society.

[12] B. Ronen, M. A. Palley, and J. Henry C. Lucas. Spreadsheet analysis and design. *Commun. ACM*, 32(1):84–93, 1989.

[13] D. Wakeling. Spreadsheet functional programming. *J. Funct. Program.*, 17(1):131–143, 2007.

[14] A. Witkowski, S. Bellamkonda, T. Bozkaya, G. Dorman, N. Folkert, A. Gupta, L. Shen, and S. Subramanian. Spreadsheets in RDBMS for OLAP. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 52–63, New York, NY, USA, 2003. ACM.

[15] A. Witkowski, S. Bellamkonda, T. Bozkaya, N. Folkert, A. Gupta, L. Sheng, and S. Subramanian. Business modeling using SQL spreadsheets. In *VLDB '2003: Proceedings of the 29th International Conference on Very large Data Bases*, pages 1117–1120. VLDB Endowment, 2003.

[16] A. Witkowski, S. Bellamkonda, T. Bozkaya, A. Naimat, L. Sheng, S. Subramanian, and A. Waingold. Query by Excel. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 1204–1215. VLDB Endowment, 2005.

[17] A. G. Yoder and D. L. Cohn. Architectural issues in spreadsheet languages. In *Proceedings of the International Conference on Programming Languages and System Architectures*, pages 245–258, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[18] A. G. Yoder and D. L. Cohn. Real spreadsheets for real programmers. In H. E. Bal, editor, *Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages, May 16-19, 1994, Toulouse, France*, pages 20–30, 1994.