Matrex Internal SQL (ISQL)

Table of Contents

Introduction

I always described Matrex as equivalent to a spreadsheet application (Excel, Calc...).

With the Internal SQL template Matrex goes a step forward.

Imagine that you have written an Excel sheet organized in **columns** (like in a table of a database). To organize a sheet in columns is normal when working with big amount of data, for example if you got your data from a database or from a feed.

What can you do on this data? It is a good assumption that you will search on them with a lookup function, add new columns with calculated values, summarize them or sort them.

And you will do this applying several formulas on cells, columns, rows.

Think if instead you can use **SQL** on these columns. Instead of writing thousand of formulas (because there are thousand of cells), so that your sheet will be unreadable to anyone else than you, you can apply a single SQL query on them and obtain exactly what you want.

Less work time, simpler sheet. And if you want to make some changes, there is a good probability that you need to change only the SQL query.

I don't know of anything like that in Excel. Excel works typically by cells, not by vectors.

But Matrex has it. It is called *Internal SQL*.

First look

ISQL (which means *Internal SQL*) is a Matrex function template.

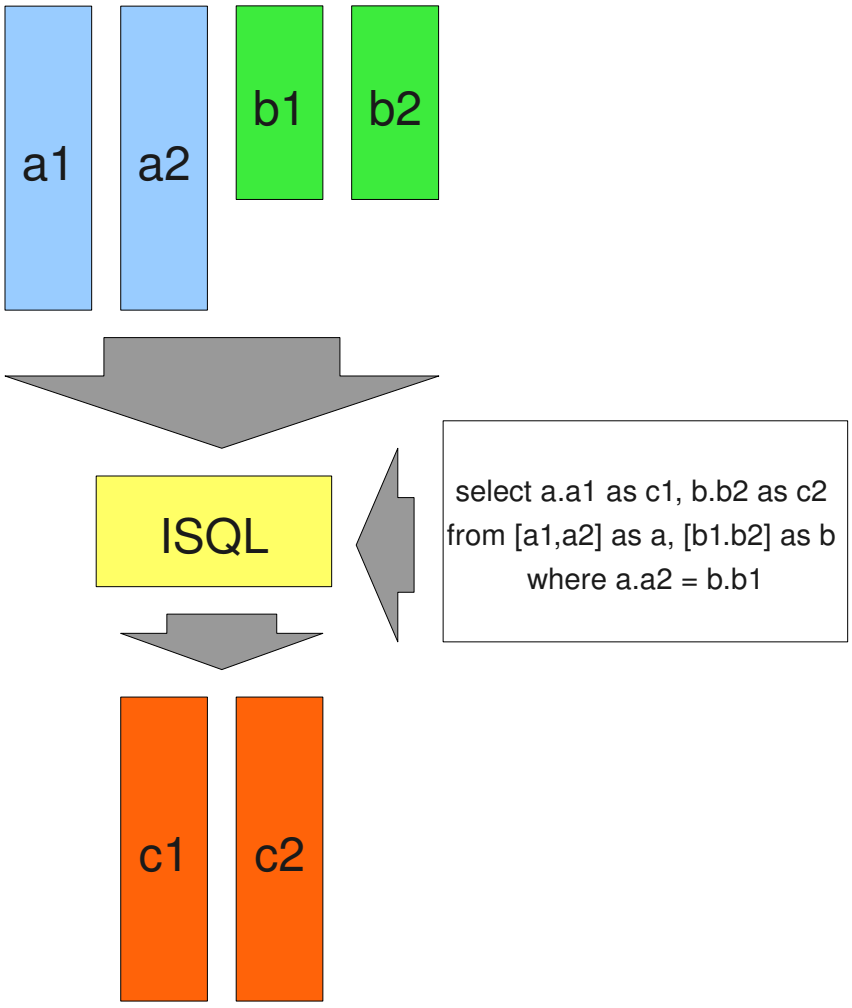
You can find it in the templates tree with path **sys.sql** and name **isql**.

The **input** arguments are:

- the SQL query, in form of a text parameter.
- the matrices that will be handled as table columns by the SQL query. They need to have only one column, in practice to be vectors.

The **output** are the matrices (also vectors) that are the resulting values of the SQL query.

The following picture gives an example of function built with this template:



In the figure:

- The name of the input and output tables are the name of the matrices in Matrex, excluded the path. This means that a matrix called *financial.calc.rent* is just considered *rent* by the template.
- The SQL is similar to the one we use to query a database, but the **from clause** is different. Since in Matrex there are no tables (it is not a database), we build our tables in the SQL query itself. *[a1, a2] as a* in the from clause means that the two input matrices a1 and a2 become the columns of the table a. In the rest of the SQL query a1 and a2 are always mentioned as a.a1 and a.a2 (it is not possible to use the column name without the table).
- The output matrices c1 and c2 are the results of the query. They are the aliases (as ...) in the select clause of the query.

Suppose we made a function from the ISQL template applying the parameters in the figure. When we call the function:

1. The SQL query is **parsed**, building an internal structure. If it contain errors the call is stopped and the errors are returned.
2. The **input** matrices are **applied** to the query internal structure.
3. The **output** matrices are **filled** with the return values of the query.

Limitations

The SQL dialect to write the queries is not standard. These are the differences:

- The FROM clause is used to **build the tables** to query, using the input matrices (like in the example we saw before, [a1, a2] as a*)*.
- All input columns have to be written as **<tableName>.<columnName>** (as a.a1 and b.b2 in the figure), everywhere in the SQL query. To write columnName without tableName is an error.
- You need to set an **alias** (AS ...) for each output value in the SELECT clause. This is needed to connect the results with the output matrices.
- Only **aggregate functions** are allowed in the SELECT clause, in the WHERE clause no functions at all are allowed. Operators (+,-,*,/,%) are allowed.

- No **sub-queries** are allowed.
- Only SELECT, FROM, WHERE, ORDER BY, GROUP BY clauses are available.
- The LIKE operator works on regular expressions. Therefore don't write *LIKE 'inter%'* but *LIKE 'inter.*'* .

We have these limitations because:

- Queries are not run against a database, but vectors in memory.
- This is the first version of the template and reliability was preferred to completeness.

It is not excluded that in future versions some of this limitations will be removed.

Query Syntax

For who wants to understand precisely the SQL used for these queries, there is a grammar file, called **MatrexSelect.g**. This file is included in the Matrex distribution.

Here I give a more human readable explanation.

The query is in the following form:

> *SELECT <expression1> as <alias1>, ... <expressionN> as <aliasN>*
> *FROM [<column1_1>, ... <columnN_1>] as <table1>, ... [<column1_M>, .. <columnP_M>] as <tableM>*
> *WHERE <condition1> [AND|OR <condition2> ... AND|OR <conitionN>]*
> *ORDER BY <column1>, ... <columnN>*
> *GROUP BY <column1>, ... <columnN>*

and composed by the following parts:

SELECT clause

The SELECT clause contains the result values of the query. There must be an alias (AS) for each value field.
Each result value can be:

- a table column
- a constant
- an operation (+, -, *, /, %) between two expressions

- an aggregate function (SUM, AVERAGE, MAX, MIN) Other functions (exp, log, ...) are not implemented. If you need to apply a function to an input or output matrix you can do it in Matrex outside the ISQL template, with the other templates of Matrex.
- a CASE expression:

  CASE <column>
  WHEN <column> = (or another operator) <constant1> *THEN* <expression1>
  WHEN <column> = (or another operator) <constant2> *THEN* <expression2>
  ...
  ELSE expressionN
  END

  In each WHEN condition you compare the column (always the same column) with a constant (constant1, constant2...). When one of the WHEN condition is true, the result value is set to the THEN expression. If none of them is true, the result value is set to the ELSE expression.

FROM clause

The FROM clause contains the definitions of the tables.

Each definition is in the form [<column1>, <column2..>] AS <table>, like in the query in the figure.

All columns in a table need to be of the **same size**. If they have different size, you can distribute them in two or more tables and join them using the WHERE clause.

WHERE clause (optional)

The WHERE clause is, as in the standard SQL, a set of predicates connected by AND and OR.

Each predicate is in the form:

- a comparison between two expressions, using standard operators (=, <>, <, >) . We saw an example of it in the figure query.
- a comparison between an expression and a regular expression, using the LIKE operator. For example:

  WHERE a.a1 LIKE 'inter.*'

to search all the a.a1 values that start with 'inter'.

- an IN clause: <expression> IN (<constant1>, <constant2>..., <constantN>). For example:

  WHERE a.a1 IN (10, 12, 16 34)

  to accept only values of the column a.a1 that are in the given list

An expressions in the WHERE clause can be:

- a table column
- a constant
- an operation (+, -, *, /) between two expressions

ORDER BY clause (optional)

As in the standard SQL, the ORDER BY clause consists of the columns by which the query result has to be sorted.

It is in the form:

ORDER BY <column1>, <column2>, ... <columnN>

GROUP BY clause (optional)

Also the GROUP BY clause is the same as in the standard SQL.

When you have some aggregate function in the SELECT clause (SUM, AVERAGE...) all the columns in the SELECT clause that are not parameters of the average functions must be mentioned in the GROUP BY clause.

The GROUP BY clause is in the form:

GROUP BY <column1>, <column2>, ... <columnN>

Some additional information.

Constants can be of 4 types, like the matrices in Matrex:

- Texts. They need to be surrounded by single quotes ('example')
- Numbers.
- Dates. They need to be surrounded by sharp characters and written as ISO dates (#2007-10-11#).

- Booleans. They need to be written as *true* or *false*.

Examples

These are some query examples (used to test the system):

Query 1

This query uses a CASE expression to return different values depending by the price.fix value and orders the results by price.ticker.

```
select price.ticker as ticker,
case price.fix when price.fix > 20 then 2 when price.fix > 13 then 1 else 0 end as byfix,
price.ask * price.fix as askfix
from [ticker, fix, ask] as price
order by price.ticker
```

Query 2

This query uses a LIKE expression to select only the rows with ticker starting by *EZPM*.

```
select price.ticker as ticker,
price.fix as sumfix
from [ticker, fix] as price
where price.ticker like 'EZPM.*'
```

Query 3

This query joins the tables product and price in the WHERE clause.

```
select product.productticker as ticker,
product.delivery as delivery,
price.fix as fix,
price.ask as ask
from [productticker, delivery] as product, [ticker, fix, ask] as price
where product.productticker = price.ticker
```