Data Processing Languages for Business Intelligence. SQL vs. R

Marin FOTACHE
Al.I. Cuza University of Iasi, Romania
fotache@uaic.ro

As data centric approach, Business Intelligence (BI) deals with the storage, integration, processing, exploration and analysis of information gathered from multiple sources in various formats and volumes. BI systems are generally synonymous to costly, complex platforms that require vast organizational resources. But there is also another face of BI, that of a pool of data sources, applications, services developed at different times using different technologies. This is “democratic” BI or, in some cases, “fragmented”, “patched” (or “chaotic”) BI. Fragmentation creates not only integration problems, but also supports BI agility as new modules can be quickly developed. Among various languages and tools that cover large extents of BI activities, SQL and R are instrumental for both BI platform developers and BI users. SQL and R address both monolithic and democratic BI. This paper compares essential data processing features of two languages, identifying similarities and differences among them and also their strengths and limits.

Keywords: Business Intelligence, Data Processing, SQL, R

1 Introduction

Often seen as a reincarnation of Decision Support Systems [1] and sometimes referred as Business Intelligence and Analytics [2], Business intelligence (BI) is a broad category of applications, technologies, and processes for gathering, storing, accessing, and analyzing data to help business users make better decisions [3]. Figure 1 displays a classical BI architecture [4].

Common business intelligence related tasks are:
- data storage
- data extraction-transformation-load from various sources in a different formats, more or less structured, to the storage layer
- data processing
- information integration
- visualization
- exploratory analysis
- data mining/data science etc.

Slightly outdated, the schema in figure 1 is still valid in suggesting the vast array of technologies, processes and tools gathered (or rebranded) within BI umbrella. Chen et al [Chen 2012] identified three generations of BI and
Analytics (BI&A) systems whose core technologies have been:
- data management and warehousing [5] [6]
- text and web analytics for unstructured web contents [7]
- mobile technologies [8].

Implementation of BI platforms requires vast quantity of organizational resources. Some of the most important current BI solutions are shown in figure 2 [9]. As with Enterprise Resource Planning applications, BI systems implementation requires extensive organizational changes and business expertise and sometimes it requires full vendor participation.

Apart from impressive costs, BI platforms have the drawback of keeping captive the customer. Every organizational change and also new or updated external data source and service must be negotiated with BI platform provider, which usually attracts new costs and also delays.

In this paper we scrutinize two languages, SQL and R, involved not only in BI application development but especially in the “democratization” of BI as they allow various types of data professionals and users to access and process vast quantity of data in an interactive, ad-hoc, way. Using two reliable sources, their role and popularity in current BI market will be outlined, taking into account job demand and a survey concerning BI tools and languages usage. Next the range of BI activities that can be supported by each SQL and R will be presented. The main section will compare SQL and R features syntax for the most common data processing/reporting problems, particularly important for BI users.

2 Languages and Tools for Business Intelligence

There is a vast array of tools, languages and technologies covering large extents of BI tasks. Some of them target regular users who are unable to write code and scripts in any programming language. Others are BI application developer’s toolbox. But there some technologies that serve both users and developers in data processing, integration, visualization and analysis. Comparison of BI tools and languages is also problematic because they can be available as programming languages, development environments, ecosystems or integrated platforms.

In evaluating the popularity of Business Intelligence languages and tools, we gather information from two reliable sources. Search engine www.indeed.com provides data about

DOI: 10.12948/issn14531305/20.1.2016.05
job trends. Figure 3 compares job demand in 2012-2016 interval for some of the most important data processing and analysis languages [10].

![Figure 3: Demand for main BI languages/tools](image)

SQL and R share most of the job postings. In 2012 SQL was by far the most demanded data language. Its share decreased slightly and seems to have stabilized since the end of 2014. R grew spectacularly in 2012-2014 interval, overpass SQL in 2014 for a brief period, and then fell back. Since 2014 it has fluctuated around 2% share. After SQL and R, the next popular is Python followed by SAS, SPSS, Stata and Julia. Currently there is still a visible lag between SQL-R group and the rest of the languages/tools, although Python seems to increase steadily and might catch up with the leading group. SAS is not only a language, but also an integrated platform covering large sections of BI applications, whereas low figures for SPSS and Stata suggest they are used mainly in academia/research. Julia is a newcomer and it is unsure if it will reach the critical mass adoption for being an important player.

![Figure 4: Top Analytics Tools and Trends in 2015](image)
The second source is annual KDnuggets Software Poll [11]. In the most recent edition, the 16th (2015), nearly 3000 voters choose from 93 different analytics and data mining tools. Results are displayed in figure 4. Survey revealed that R is the most popular overall tool among data miners, and Python is gathering traction steadily. RapidMiner continues to be most popular suite for data mining/data science. Hadoop/Big Data increased to 29%, up from 17% in 2014, and the fastest growth is for Spark whose usage share grew over 3-fold. KDnuggets Software Poll and indeed.com job trends confirm the centrality of both SQL and R as tools for BI users and developers.

3 SQL vs. R. The extent of analysis

As a data centric approach, BI heavily relies on various advanced data collection, extraction, and analysis technologies, from Data Warehouse, extract – transform – load (ETL) tools, analytical processing (OLAP), advanced reporting to advanced knowledge discovery tools and techniques [2][12].

SQL and R languages are pivotal in BI data processing, as will be detailed in section 4. SQL and R are not only contenders, but also partners, especially when accessing and processing huge volumes of data stored in databases, data warehouses and Hadoop ecosystems. Feature comparison of table 1 is not intended to rank the first of two languages, but to outline the main areas SQL and R can be analyzed and compared, and also some areas where they do not really match, so the comparison is fruitless.

<table>
<thead>
<tr>
<th>Table 1. Main similarities and differences between SQL and R</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Content</strong></td>
</tr>
<tr>
<td>Main target</td>
</tr>
<tr>
<td>Standardization</td>
</tr>
<tr>
<td>Implementation</td>
</tr>
<tr>
<td>Dialects</td>
</tr>
<tr>
<td>Modularization</td>
</tr>
<tr>
<td>Data Storage</td>
</tr>
<tr>
<td>Massiveness of data to be processed</td>
</tr>
</tbody>
</table>

DOI: 10.12948/issn14531305/20.1.2016.05
Data Sources

| Data Sources | Not related to the language, but dependent of DBMs. Usually limited to databases created with a couple of SQL/relational DBMSs | Almost every data source (.csv, .xlsx, xml/html tables, relational databases, NoSQL data stores, Hadoop etc.) |

Table 1 (continued). Main similarities and differences between SQL and R

<table>
<thead>
<tr>
<th>Content</th>
<th>SQL</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedularity</td>
<td>Implemented through section <em>Persistent Stored Modules</em> of the SQL standards; big differences among procedural extensions (PL/SQL, T-SQL, SQL PL, etc.) [17]</td>
<td>Native</td>
</tr>
<tr>
<td>Data Processing</td>
<td>Best known language for processing data</td>
<td>Various features included in base R and especially a large number of packages. SQL queries can be run with package <em>sqldf</em> [26]. Two workhorses – packages <em>dplyr</em> [27] and <em>tidyr</em> [28]</td>
</tr>
<tr>
<td>OLAP Functions</td>
<td>Implemented since SQL:2009 standard [16] [17] [29]</td>
<td>Implemented in <em>dplyr</em> package</td>
</tr>
<tr>
<td>Pivoting</td>
<td>“Manual” [16], with recursive queries, or with <em>pivot</em> or <em>model</em> clauses [29] [30]</td>
<td>Package <em>tidyr</em></td>
</tr>
<tr>
<td>Data Visualization</td>
<td>Simple (text-only) histograms [29]</td>
<td>Excellent features and packages [31]: <em>ggplot2</em> [32], <em>lattice</em> [33], <em>ggvis</em> [34], <em>googleVis</em> [35]</td>
</tr>
<tr>
<td>Data Analysis</td>
<td>Initially only basic descriptive statistics [16] Limited to some ANOVA, t-tests and basic non-parametric tests in some dialects (e.g. Oracle) [6]</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Dynamics</td>
<td>Mature. Slow pace of implementing new features because of multitude of institutions and companies involved in the standardization process</td>
<td>(Still) young. Accelerate evolution, dozens of new packages every week. Worries of becoming out of control</td>
</tr>
<tr>
<td>Learning curve</td>
<td>Easy to learn</td>
<td>Steep</td>
</tr>
</tbody>
</table>

Due to its processing power and accessibility, SQL has become the lingua franca of data processing. Its leading position seemed many times in jeopardy by the launch of languages like OQL and, more recently, NoSQL data stores and Hadoop systems [19]. But currently there is visible trend for adapting SQL even for NoSQL and Hadoop ecosystems [6] [19] [20]. That will reinforce SQL ubiquity since it can process data stored on every data platform.

The main drawback of SQL is a direct consequence of its maturity. SQL standard is hard to update. The process takes years and various dialects can implement, at least in the initial phase, quite differently the same features.

By contrast, one the most important strengths of R is its dynamics and the enthusiasm of R community. Until *dplyr* and *tidyr* packages, R
had been no match to the expressive power of SQL in terms of data processing options. Things are going to change, as seen in the next section. R is a vast endeavor. Data processed in R or within R but with SQL queries are ready to be explored, visualized, analyzed, and mined for patterns. In this respect, SQL cannot compete with R. Also R provides many other tools such as reporting (R Markdown), or even application development (Shiny) [36].

4 Main Data Processing Features in SQL

As in the current paper we did not test the performances of two systems, data were generated just for scale factor of 0.1 with the following number of records:
- 5 in table REGION
- 25 in table NATION
- 15000 in table CUSTOMER
- 1000 in table SUPPLIER
- 20000 in table PART
- 80000 in table PARTSUPP
- 150000 in table ORDERS
- 600572 in table LINEITEM

From PostgreSQL tables were imported as data frames using package RPostgreSQL [39]. Because for almost every problem both SQL and R provide a large number of solutions, we
selected the shortest or the most readable. As SQL is a mature technology, we will provide some details just for the R queries.

Base R options are no match for SQL in terms of querying power. For displaying basic information about the quantity and prices of items sold within orders of January 1996 (second simple query in section 4.1), base R solution (with a little help from lubridate package) is:

```r
library(lubridate)  # package needed for functions "year" and "month"
t1 <- merge(orders, lineitem, by.x = "o_orderkey", by.y = "l_orderkey")
t2 <- subset(t1, year(o_orderdate) == 1996 & month(o_orderdate) == 1,
            select = c(o_orderkey, l_linenumber, l_partkey, l_quantity, l_extendedprice, l_discount))
t2 <- transform(t2, line_amount = l_quantity * (l_extendedprice - l_discount))
t3 <- t2[order(t2$o_orderkey, t2$l_linenumber),]
```

The query is divided into a number of steps. The result of each step is a table that is processed by subsequent steps. The operating logic might be obvious, but the solution is cumbersome by any standard.

A series of packages, mainly dplyr and tidyr injected elegance and power into data processing in R. It is the main reason that on the subsequent examples, SQL syntax will be matched to syntax of (mainly) these two packages.

### 4.1 Basic Queries (Selection, Projection, Join)

First simple query requires a few basic operation: selection, projection, computed column, and sort - Display some basic information about the quantity and prices of items sold within orders 1284 and 1731.

Below is the syntax for both SQL and R:

**SQL:**

```sql
select l_orderkey, l_linenumber, l_partkey, l_quantity, l_extendedprice, l_discount
as line_amount from lineitem
where l_orderkey in (1284, 1731)
order by l_orderkey, l_linenumber
```

**R (dplyr):**

```r
library(lubridate)
orders %>%
inner_join(lineitem, by = c('o_orderkey' = 'l_orderkey')) %>%
filter(year(o_orderdate) == 1996 & month(o_orderdate) == 1) %>%
select(l_orderkey, l_linenumber, l_partkey, l_quantity, l_extendedprice, l_discount) %>%
mutate(line_amount = l_quantity * (l_extendedprice - l_discount)) %>%
arrange(l_orderkey, l_linenumber)
```

The “pipe” operator (`%>%`) passes current operation result to the next operation within the same query so there is no need to save intermediary results in separate data frames. Main predicate. Dplyr provides functions for each basic verb of data manipulation [27], from which in the query just

- `filter()` for selection of records
- `arrange()` for sorting records
- `select()` for (attributes) projection
- `mutate()` for adding computed columns.

The second simple query joins tables (data frames) `orders` and `lineitem` - Display basic information about the quantity and prices of items sold within orders of January 1996:

**SQL:**

```sql
select o_orderkey, o_orderdate, l_linenumber, l_partkey, l_quantity, l_extendedprice, l_discount
as line_amount from orders inner join lineitem
on o_orderkey = l_orderkey
where extract(year from o_orderdate) = 1996 and extract(month from o_orderdate) = 1
order by o_orderkey, l_linenumber
```

**R (lubridate and dplyr):**

```r
library(lubridate)
orders %>%
inner_join(lineitem, by = c('o_orderkey' = 'l_orderkey')) %>%
filter(year(o_orderdate) == 1996 & month(o_orderdate) == 1) %>%
select(o_orderkey, l_linenumber, l_partkey, l_quantity, l_extendedprice, l_discount) %>%
mutate(line_amount = l_quantity * (l_extendedprice - l_discount)) %>%
arrange(o_orderkey, l_linenumber)
```
Syntax for the inner join is close enough to that of SQL. Similarly dplyr implements outer joins and semi joins, but also the anti-join, which is not available in too many SQL dialects.

4.2 Aggregate Queries

Similar to SQL, dplyr offers group_by() function. It breaks down a dataset into specified groups of rows so that when applying a verb to the resulting object, the verb will be performed for each group. Grouping affects the verbs as follows:

- grouping variables are always retained in the result
- grouped arrange() orders first by the grouping variables (which is not the case in SQL)
- mutate() and filter() are most useful in conjunction with window functions
- slice() extracts rows within each group.
- summarise() allows defining aggregate variables

First problem requiring basic aggregation with group filter is Display monthly sales for years 1996-2000 that are greater than one billion:

**SQL:**

```sql
select extract(year from o_orderdate) as year,
       extract(month from o_orderdate) as month,
       sum(l_quantity * (l_extendedprice - l_discount)) as monthly_sales
from orders
inner join lineitem
on o_orderkey = l_orderkey
where extract(year from o_orderdate) between 1996 and 2000
group by year, month
having sum(l_quantity * (l_extendedprice - l_discount)) > 1000000000
order by year, month
```

Here there is a slight advantage of dplyr due to the pipe operation, so that the expression for group filtering does not have to define repeatedly the expression for aggregation.

Second aggregate queries answers a problem with subtotals - Display monthly sales for years 1996-2000 with subtotals on each year:

**SQL:**

```sql
select extract(year from o_orderdate) as year,
       extract(month from o_orderdate) as month,
       sum(l_quantity * (l_extendedprice - l_discount)) as monthly_sales
from orders
inner join lineitem
on o_orderkey = l_orderkey
where extract(year from o_orderdate) between 1996 and 2000
group by year, month
union
select extract(year from o_orderdate) as year, null as month,
       sum(l_quantity * (l_extendedprice - l_discount))
from orders
inner join lineitem
on o_orderkey = l_orderkey
where extract(year from o_orderdate) between 1996 and 2000
group by year, month = NA
order by year, month
```

**R (lubridate and dplyr):**

```r
orders %>%
  inner_join(lineitem, by = c('o_orderkey' = 'l_orderkey')) %>%
  filter(year(o_orderdate) >= 1996 & year(o_orderdate) <= 2000) %>%
  group_by(year = year(o_orderdate), month = month(o_orderdate)) %>%
  summarise(monthly_sales = sum(l_quantity * (l_extendedprice - l_discount))) %>%
  filter(monthly_sales > 1000000000) %>%
  arrange(year, month)
```
Pipe operator was no useful in this case, so SQL query is shorter. Function bind_rows() in dplyr is the equivalent of UNION SQL operator.

4.3 Subqueries

There are numerous problems to be answered using what is SQL are called as subqueries. In SQL queries can be part of WHERE, HAVING, FROM, SELECT and WITH clauses. In dplyr there are no such differences.

First problem answered in SQL with a subquery placed into WHERE clause is Retrieve orders issues in the same day as the order 3271:

SQL:  
```sql
select o_orderkey, o_custkey, o_orderstatus, o_orderdate, o_orderpriority
from orders
where o_orderdate in (select o_orderdate from orders where o_orderkey = 3271)
```

Here required attributes were consecutive. That shortened the projection clause (select) in the dplyr query. Also notice `in` operator, which is the counterpart of SQL IN.

The second problem for illustrating subqueries implies group filtering is Extract day (or days, if there are more than one) with the maximum number of orders. Three SQL solutions and two dplyr “counterparts” will be provided as follows:

SQL – solution based on a subquery in FROM clause

```sql
select dates_n_of_orders.*
from (select o_orderdate, count(*) as n_of_orders
from orders
group by o_orderdate
order by n_of_orders desc
limit 1) max_n_of_orders
inner join
(orders %>% group_by(o_orderdate) %>% summarize(n_of_orders = n()) %>%
inner_join(orders %>% group_by(o_orderdate) %>% summarize(n_of_orders = n()) %>%
inner_join(dates_n_of_orders %>% filter(n_of_orders >= max(n_of_orders))))
%>% top_n(1))
```

SQL – solution based on table expression

```sql
with dates_n_of_orders as (select o_orderdate, count(*) as n_of_orders
from orders
group by o_orderdate)
select *
from dates_n_of_orders
where n_of_orders = (select max(n_of_orders)
from dates_n_of_orders)
```
Now dplyr syntax looks more powerful and elegant. Second dplyr solution contains an OLAP (window) function – top_n().

The third problem involves group comparison - Extract dates with at least the number of orders issued on 1993-04-21:

SQL – solution R (dplyr):

```r
with dates_n_of_orders as 
(select o_orderdate, count(*) as n_of_orders 
from orders 
group by o_orderdate) 
select * 
from dates_n_of_orders 
where n_of_orders >= 
(select n_of_orders1 from dates_n_of_orders where o_orderdate = DATE'1993-04-21') 
orders %>% 
inner_join(lineitem, by = c('o_orderkey' = 'l_orderkey')) %>% 
filter(year(o_orderdate) >= 1991 & year(o_orderdate) <= 2000) %>% 
inner_join(part, by = c('l_partkey' = 'p_partkey')) %>% 
group_by(products = p_name, year = year(o_orderdate)) %>% 
summarize(sales = sum(l_quantity * (l_extendedprice - l_discount))) %>% 
spread(year, sales, fill=0)
```

Here a dplyr trick was needed in order the groups to be compared. After computing the number of orders for each date (n_of_orders1), we create a variable (n_of_orders2), which is zero for all of the dates except the date of reference (1993-04-21). Only for this date n_of_orders2 has the same value as n_of_orders1. The third new variable (n_of_orders3) stores the number of orders for the reference date and it was used for filtering the groups.

### 4.4 Pivoting

Pivot tables is one the key features in BI reporting and analysis. Many relational/SQL DBMSs have implemented PIVOT clause for answering this type of problems [29][30]. Unfortunately, currently PostgreSQL does not support dynamic pivoting. Function CROSS-TAB requires explicit definition of all column of the pivoting operation. This is acceptable from small number of columns and awkward for the rest of the cases. Consequently, for the problem Display product sales for sales years between 1991 and 2000 only dplyr solution will be presented:

```r
orders %>% 
inner_join(lineitem, by = c('o_orderkey' = 'l_orderkey')) %>% 
filter(year(o_orderdate) >= 1991 & year(o_orderdate) <= 2000) %>% 
inner_join(part, by = c('l_partkey' = 'p_partkey')) %>% 
group_by(products = p_name, year = year(o_orderdate)) %>% 
summarize(sales = sum(l_quantity * (l_extendedprice - l_discount))) %>% 
spread(year, sales, fill=0)
```

4.5 Recursivity

Introduced by ANSI SQL:1999 standard and implemented almost all major relational DBMSs, recursive queries (WITH RECURSIVE) are extremely powerful tools for database processing and remains largely underused by database professional. Recursivity is needed, for example, when the number of levels in tree (hierarchical structure) varies from one organization to another and there is a need to display, for each employee, the entire managerial path (from she/he until the top manager).

TPC-H schema does not contain a proper hierarchical structure. Nevertheless, the are types of problems that require recursivity. For example, Display, for each product, a string with the list of first 5 customers (in alphabetic order) that have purchased it:

SQL – solution based on recursive query built with a table expression

```sql
with recursive prod_customer_list (p_name, p_partkey, cust_no, cust_list) as 
(select pc.p_name, pc.p_partkey, pc.cust_no, cast(pc.c_name as varchar(100000)) as cust_list 
from pc 
inner_join(lineitem, by = c('p_partkey' = 'l_partkey')) 
inner_join(orders, by = c('l_orderkey' = 'o_orderkey')) 
inner_join(customer, by = c('o_custkey' = 'c_custkey'))) 
%>
```

```r
donkey %>% 
inner_join(lineitem, by = c('o_orderkey' = 'l_orderkey')) %>% 
inner_join(part, by = c('l_partkey' = 'p_partkey')) %>% 
group_by(products = p_name, year = year(o_orderdate)) %>% 
summarize(sales = sum(l_quantity * (l_extendedprice - l_discount))) %>% 
spread(year, sales, fill=0)
```
where cust_no = 1
    union all
select pc.p_name, pc.p_partkey, prod__customer_list.cust_no + 1, cast (prod__customer_list.cust_list || ' ' || pc.c_name AS varchar(100000))
from pc inner join prod__customer_list
    on pc.cust_no = prod__customer_list.cust_no + 1,
    pc as (select *
        from (select p_name, p_partkey, c_name, row_number() over (partition by p_name order by c_name) as cust_no
            from part inner join lineitem on p_partkey = l_partkey
            inner join orders on l_orderkey = o_orderkey
            inner join customer on o_custkey = c_custkey) x
        where cust_no <= 5)
select (p_name, c_name) %>%
group_by(p_name) %>
slice(1:5) %>%
group_by(p_name) %>
summarise(five_cust_list =
paste(c_name, collapse=" ", "))

From this example one might conclude that dplyr syntax is considerable more elegant in solving problems requiring recursive queries.

4.6 OLAP functions
In SQL OLAP functions, labeled also as analytic or window functions were added in SQL:1999 standard. In R dplyr provides a large number of window functions such as:
- lead - to copy with values shifted by 1
- lag - to copy with values lagged by 1
- dense_rank - to rank with no gaps
- min_rank – ranks; ties get min rank
- percent_rank - ranks rescaled to [0, 1]
- row_number - ranks; ties got to first value
- ntile - bin vector into n buckets
- cume_dist - cumulative distribution
- cumall - cumulative all.

An example of such analytic function was included in a previous query. Because the logic of these functions is quite similar to their SQL counterparts, no additional example will be provided.

5 Conclusions
For Business Intelligence SQL and R are core technologies. Addressing all the range of professionals between BI application developers and qualified users, both languages support BI decentralization, modularization, flexibility and, ultimately, the BI democratization.

SQL is the real Esperanto of the data processing languages. Since now, all the challengers (OQL, NoSQL and some Hadoop systems) ended in adopting a more SQL look or face extinction.

Various information sources show that both SQL and R have leading positions in professional BI professionals and job market demand.

Covering different segments of BI activities, both languages share data processing and reporting segment. This paper argues that, with new packages like dplyr and tidyr, R can compete with SQL in terms of processing power and syntax elegance.

References


DOI: 10.12948/issn14531305/20.1.2016.05


[33] Sarkar, Deepayan, Lattice: Multivariate Data Visualization with R. New York: Springer Verlag, 2009


DOI: 10.12948/issn14531305/20.1.2016.05
Marin FOTACHE has graduated (long time ago) the Faculty of Economics at Alexandru Ioan Cuza University of Iasi, Romania. He holds a PhD diploma in Business Information Systems (Business Informatics) from 2000 and he had gone through all didactic positions since 1990 when he joined the staff of Alexandru Ioan Cuza University, from teaching assistant in 1990, to full professor in 2002. Currently he is professor within the Department of Accounting, Business Informatics and Statistics in the Faculty of Economics and Business Administration at Alexandru Ioan Cuza University. He is the (co)author of books and journal articles in the fields of SQL, database design, NoSQL, big data, Information Systems curricula/market and knowledge management.