Fast and Scalable Relational Division on Database Systems

André S. Gonzaga[1], **Robson L. F. Cordeiro**[1]

[1] Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo (USP) – São Carlos, SP – Brazil

`asgonzaga@usp.br, robson@icmc.usp.br`

Abstract. *The Relational Algebra is composed of several operators to assist queries and data manipulation on Relational Databases. The Relational Division operator, particularly, allows simple representations of several queries involving the concept of "for all", however, the SQL does not have an explicit implementation for it. In this paper, we compare the performance of the best implementation known for the division operator in SQL, considering different cases of use. We also present a new algorithm for the division, which we implemented through stored procedures. We performed a case study using the relational division to select genetic data. The results showed that our implementation for the relational division is potentially faster than the best implementation in SQL.*

1. Introduction

Data queries applying the concept of "for all" are widely used in real applications — e.g., to select candidates having all the skills for a given job, to select providers that produce all the models of a given type of product, to select the diseases that have all the given symptoms. The division ($\div$) operator, defined in the Relational Algebra, allows to represent these types of queries with a single operation, since it is the only direct algebraic correspondent to the Universal quantification ($\forall$) from the Relational Calculus [Codd 1972]. However, none of the Relational Database Management Systems (RDBMS) implement this operator, forcing users to write complex nested queries in order to express a simple concept of data relationship. In other words, there is no generic, intuitive and efficient implementation for the division operation.

In this work we investigate the performance of the best implementation known for the division operator in RDBMS on different cases of use. We also present a new implementation using stored procedures to perform the operation through a new algorithm potentially faster than the best implementation in SQL. We also applied the relational division on a practical context, selecting individuals that have all the given genetic conditions, considering a genetic dataset with Single Nucleotide Polymorphism (SNP). For that, the main contributions of this work are:

1. Evaluate the division implementations in RDBMS in different cases of use.
2. Investigate which aspects of the data affect the execution time of each implementation.
3. Propose a new algorithm to solve the relational division queries.
4. Perform a case study to select genetic data using the relational division.

2. Background

2.1. Genetic Data

The genome of an individual is the complete set of genetic information in its organism. It is associated to all phenotypic expressions of the individual, varying from its predisposition to develop diseases until the color of its eyes, for example [Zhang et al. 2004]. The genome is stored in molecules of DNA (deoxyribonucleic acid) called chromosomes. Each chromosome carries the genes responsible to develop the organism traits, formed from a chain of four nucleotide bases — A, T, C, and G. Population of the same species has almost all the DNA carrying the same nucleotides at a specific position in the DNA sequence, however, if more than one percent of the population have variations at a specific position, the position is therefore classified as a Single Nucleotide Polymorphism, or SNP (pronunced “snip”), as it is shown in Figure 1. When a SNP occurs within a gene, then it is described as having more than one allele, that is, a pair of genes that appear at a particular location on a particular chromosome and control the same characteristics. The datasets containing SNPs are represented by the position at chromosome and a pair of alleles, each one expressed by the numbers `1` or `2`.

```
                 SNP 1      SNP 2               SNP 3
Individual 1:  aat|t|ccgctga|a|gtgacgttggggcat|t|acaa
Individual 2:  aat|a|ccgctga|t|gtgacgttggggcat|c|acaa
Individual 3:  aat|c|ccgctga|g|gtgacgttggggcat|g|acaa
```

Figure 1: An example of SNPs along the chromosome of the individuals.

2.2. Relational Division

The Relational Division ($\div$)[Codd 1972] has an important role of expressing relational queries, since it is the only, directly, algebraic correspondent to the Universal Quantification ($\forall$) from the Relational Calculus, allowing simple and intuitive algebraic representations for queries involving the concept of “for all”. The division operation is a derived operator. That is, it can be expressed as a combination of other basic operators, as shown in the Equation 1. In this expression R_1 and R_2 are relations representing the dividend and the divisor of the operation, respectively. The division operates over the union-compatible lists of attributes A and B from relations R_1 and R_2. The result is a relation with the list of attributes $\overline{A}$, in which $\overline{A}$ contains all the attributes that are not in A.

$$R_1[A \div B]R_2 \Leftrightarrow (\pi_{\overline{A}}R_1) - \pi_{\overline{A}}((\pi_{\overline{A}}R_1) \times R_2) - R_1) \tag{1}$$

For instance, consider the Relational Division operation showed in Table 1 to answer the query “*Which individuals have **all** the given genetic conditions*”. In this example, the dataset is composed of SNP, representing the variations in a DNA sequence among individuals. In the example schema[1], the individual is represented by the attribute `ID`. Each individual have several tuples representing the SNPs — attributes `Allele 1` and `Allele 2` with its position along the chromosome — attribute `Position`. The relations involved in the division are Table 1a and 1b, representing the dividend R_1 and the divisor R_2, respectively. The lists of attributes A and B, from the division equation, represents the columns used to validate the genetic conditions desired for each individual, which in this example are given by the subset of attributes {`Position, Allele1, Allele2`}. The result relation is given by Table 1c, including all of the attributes of R_1 that are not included in the list of attributes A. In this example, just attribute `ID` is part of the result relation, representing the individuals who have a tuple with the same values for each tuple in the R_2, in this example, just the individual `1`.

3. Related Work

The computational complexity of the Relational division was studied in [Leinders and Van den Bussche 2005], in which it is proven that any implementation of the division using only operators from the Relational Algebra, e.g., Query 1, produces expressions with a minimum time complexity of O(n^2), where n is the number of tuples in the dividend. However, it is also shown that there are alternative approaches that uses counting and

[1]Usually, the datasets contains more than 10,000 SNP for each individual. Thus, it is unfeasible represent them through attribute columns.

Table 1: Example of the Relational Division used to select individuals that satisfy desired genetic conditions, in a dataset representing single nucleotide polymorphism alleles and their position along the chromosome of the individuals.

$\overline{A}$	A		
ID	**Position**	**Allele 1**	**Allele 2**
1	0.10	1	2
1	44.5	2	2
1	84.0	2	1
2	0.10	2	2
2	44.5	1	1
2	84.0	1	2
3	0.10	1	2
3	44.5	2	1
3	84.0	2	2

(a) R_1 — Genetic data of individuals.

$\div$

B		
Position	**Allele 1**	**Allele 2**
44.5	2	2
84.0	2	1

(b) R_2 — Desired genetic conditions.

$=$

$\overline{A}$
ID
1

(c) Selected individuals.

sorting strategies to optimize the operator implementation, allowing to achieve a complexity of O($n\ log\, n$), e.g., Query 2. In [Celko 2009], implementations of queries that use the Relational Division concept were investigated. Several cases of application were studied, and also covers alternative definitions of the operator. However, the work did not made any experiment to validate the performance of these custom queries. The work [Camps 2014] optimizes the division query through tuple reduction, i.e., filtering the data before executing the query.

Query 1: Division query based on the formal definition.

```
SELECT DISTINCT Ā FROM(
    (SELECT Ā FROM R1)
    MINUS (SELECT Ā FROM
        ((SELECT * FROM (SELECT Ā FROM R1) CROSS JOIN R2)
        MINUS (SELECT * FROM R1))));
```

Query 2: Division query based on counting tuples.

```
SELECT Ā FROM R1 NATURAL JOIN R2
GROUP BY Ā
HAVING COUNT(*) = (SELECT COUNT(*) FROM R2);
```

The work [Matos and Grasser 2001] compares possible implementations of the Relational Division in Structured Query Language (SQL). The study was made more than ten years ago and, although the small amount and the lack of diversity of the data used in the experiments, it shows that using the counting approach — Query 2, had the best performance and the lesser writing complexity among all the other implementations. On our previous work [Gonzaga 2014], we also investigate several different implementations, found on literature, of the Relational Division using SQL. However, differently from the [Matos and Grasser 2001], we performed experiments with millions of tuples and with a large diversity on data. The results, although, corroborate that the query using the counting approach — Query 2, is the most efficient overall.

Execution plan 1: Division query based on counting tuples.

```
FILTER
  HASH GROUP BY
    NESTED LOOPS
      TABLE ACESS (FULL)          R2
      INDEX RANGE SCAN            R1
  SORT AGGREGATE
    TABLE ACCESS (FULL)           R2
```

4. Proposed Algorithms

4.1. Index-Division

We developed a new algorithm for the division operation, shown in Algorithm 1, which has as premise the existence of index structures over the dividend attributes in list A. This approach starts with a set of valid individuals G_V filled with all possible individuals, that is, it assumes that all individuals satisfy the requirements at the beginning. Then, for each SNP requirement in R_2 it is performed a index query in relation R_1, using the requirement as the query object. It recovers then for each tuple retrieved by the query, to which individual it belongs. Then, the valid individuals are updated with the intersection of the currently valid groups and the groups that meet the last requirement, of the previous iteration. Therefore, at the end of the algorithm, just the individuals that meet all the SNPs requirements will be members of the set of valid individuals (G_V).

Algorithm 1 Division based on index.

```
$Gv = \pi_{(\overline{A})}(R_1)$
**for each** $tuple\ t \in R_2$
    $R \leftarrow \pi_{(\overline{A})}(\sigma_{(t)}R_1)$
    **for each** $tuple\ r \in R$
        $Gv \leftarrow Gv\ \cap\ r$
**return** $G_V$
```

The proposed algorithm performs, for each tuple in relation R_2 an indexed search query in relation R_1. Assuming that an index structure performs search queries with a complexity of O($logn$), where n is the relation cardinality, thus, the theoretic complexity of the algorithm is O($m\, log\, n$), where n and m are the number of tuples in relations R_1 and R_2, respectively. On this work we implemented the proposed algorithm through stored procedures using PL/SQL on the Oracle database system.

4.2. Division Data Generator

To study the algorithms performance on different scenarios, we develop a data generator using parameters: (1) Cardinality, the number of tuples in the relations of dividend R_1 and of divisor R_2; (2) Number of individuals, the number of groups of tuples representing the individuals to be evaluated in the operation; (3) Correlation, the percentage of individuals, from the total, which satisfy all the requirements on R_2 thus being part of the result, e.g, 10% of correlation means that only 10% of all individuals are going to be selected on the final result; (4) Variability, the differences in size between individuals, adjusting the number of tuples on each group. High variability means that some individuals have many tuples, and others have just a few. Low variability means that almost all individuals have the same number of tuples;

All these parameters are linked and affect one another, thus, to perform the data generation, our algorithm is as follows: (1) set the cardinality of the divisor relation R_2; (2) set the number of individuals based on the correlation value; (3) create the group of tuples for each individual, taking the variability value into account; (4) distribute the values for each one of these tuples ensuring that only the individuals which are part of the result, setted by the correlation value, will satisfy all the tuples in the divisor.

5. Experiments

All the experiments were performed in a machine with an Intel i7 2.67GHz processor and 12Gb RAM, using the Oracle Database 11g Express running on a Ubuntu 14 operation system. All the queries use index structures to optimize their performance.

5.1. Synthetic Data

We perform experiments varying the parameters used on the data generator. Each plot in Figure 2 represents the variation of a single parameter while the others are fixed in: (1) dividend, one million of rows; divisor, ten rows; (2) one hundred of individuals; (3) 10% of correlation; (4) 50% of variability; The plot in Figures 2a and 2b shows that the cardinality of the relations of dividend and divisor, is the only parameter that have significant impact on the query running time. The plot of number of individuals, correlation and variability, showed in Figures 2c, 2d and 2e presents no significant variations on running time. The plot presented in Figure 2g, where we generate 40 test cases varying the parameters as:(1) dividend, from hundred to million of rows; divisor, from one to hundred of rows; (2) from two individuals to hundreds; (3) from 1% to 100% of correlation; and the variability fixed in 50%; shows that, overall, although the execution plan for the counting query — Execution Plan 1, is different from the Index Division algorithm, both approaches have almost the same performance. The query that translate the formal definition have the worst performance on all test cases. Theses results can be associated to the plot in Figure 2f, which shows the computer processing cost generated by the Oracle DBMS for each division query.

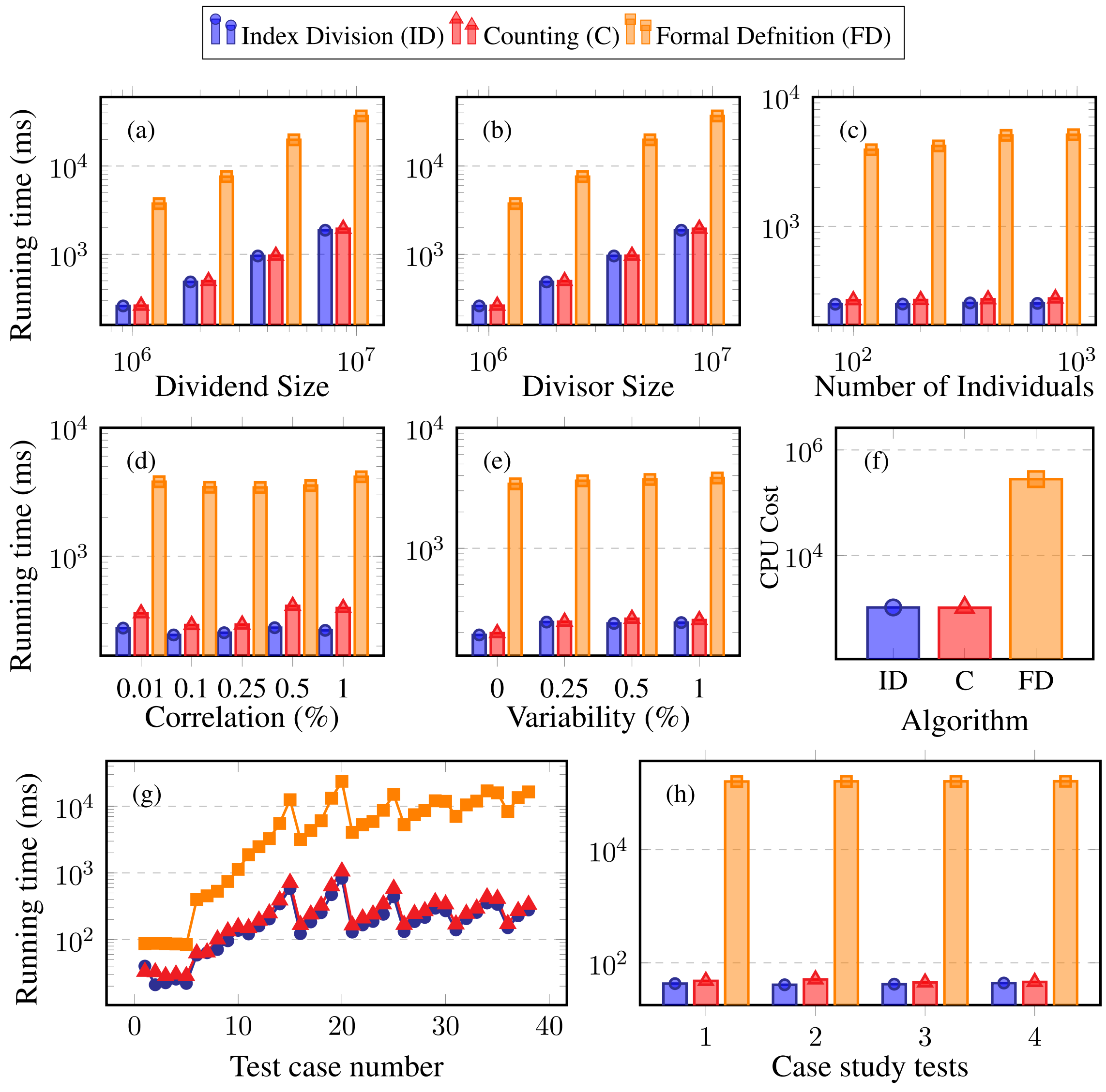

Figure 2: Relational Division queries performance on different use cases.

5.2. Case Study

We performed a case study where the division is used to select animals that have all the given genetic conditions, represented by SNPs. The dataset used was obtained through the XVI QTL-MAS Workshop[2] public data, where a cattle population has been simulated. It contains 10,000 SNP genotypes for 4,100 individuals, which gives a relation with more than 40 million tuples — considering the schema presented in Table 1. We performed four divison queries varying the genetic conditions from one to fifteen, and the correlation from 20% to 1%. The results are shown in Figure 2h and corroborate the results presented in the previous section, showing that the index division algorithm and the counting query have almost the same performance, while the formal definition is thousand times slower.

6. Conclusion

We evaluate the implementations of the relational division, and we showed that the cardinality of the relations involved in the division operation is the characteristic that most affects the execution time of the queries. Also, we present a case study using genetic data showing a practical application of the division operation over a high volume of data. The Index division algorithm showed to have almost the same performance against the best division query presented in the literature, being slighter faster overall. However, the Index Division was implemented through stored procedures using PL/SQL, which means that the algorithm have the bottleneck of switching contexts from traditional SQL and the procedural language. Thus, we consider that a possible implementation of the Index Division inside the core of the DBMS could achieve the best performance on relational division queries.

7. Acknowledgment

We thank the financial support of FAPESP, projects 2015/ 05607-6 and 2014/21483-2, CAPES and CNPq to this work.

References

Camps, D. (2014). High performance relational division in sql server. *Simple-Talk*. [Online; acessed April 26,2016].

Celko, J. (2009). Divided we stand: The sql of relational division. *Simple-Talk*. [Online; acessed April 26,2016].

Codd, E. F. (1972). Relational completeness of data base sublanguages. *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research RJ 987, San Jose, California*.

Gonzaga, A. (2014). Study aimed at simplification and optimization of relational division in database systems. Term Paper, University of São Paulo, São Carlos, Brazil.

Leinders, D. and Van den Bussche, J. (2005). On the complexity of division and set joins in the relational algebra. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 76–83. ACM.

Matos, V. M. and Grasser, R. (2001). Assessing performance of the relational division operator. *Data Base Management*, 22-20-30:1–11.

Zhang, K., Qin, Z. S., Liu, J. S., Chen, T., Waterman, M. S., and Sun, F. (2004). Haplotype block partitioning and tag snp selection using genotype data and their applications to association studies. *Genome Research*, 14(5):908–916.

[2]http://qtl-mas-2012.kassiopeagroup.com/