Unified-Processing of Flexible Division Dealing with Positive and Negative Preferences

Noussaiba Benadjimi, Walid Hidouci

Unified-Processing of Flexible Division dealing with Positive and Negative Preferences

Noussaiba BENADJIMI , Walid HIDOUCI

Laboratoire de la Communication dans les Systmes Informatiques
Ecole nationale Supérieure d'Informatique, BP 68M, 16309, Oued-Smar,
Alger, Algérie. http://www.esi.dz
an_benadjimi@esi.dz , hidouci@esi.dz

Abstract. Nowadays, current trends of universal quantification-based queries are been oriented towards flexible ones (tolerant queries and-or those involving preferences). In this paper, we are interested in universal quantification-like queries dealing with both positive or negative preferences (requirements or prohibitions), considered separately or simultaneously. We have emphasised the improvement of the proposed operator, by designing new variants of the classical Hash-Division algorithm, presented in [1], for dealing with our context. The parallel implementation is also presented, and the issue of answers ranking is dealt with. Computational experiments are carried out in both sequential and parallel versions. They shows the relevance of our approach and demonstrate that the new operator outperforms the conventional one with respect to performance (the gain exceeds a ratio of 40).

Keywords: Universal quantification-like queries, Relational division, Relational anti-division, Preferences, Tolerant division, Hash-division.

1 Introduction

Relational operators including universal quantification are an interesting type of queries. They are very useful for many applications, especially in business intelligence applications and in recommendation systems [2]. In relational algebra, universal quantification-like queries are the most complex operators. That is why a lot of research focuses on their implementation, algorithms and optimisation[3]. Universal quantification-like queries are, often, about **division** or **anti-division** operators. The division searches elements associated with all members of a set of requirements, while the anti-division aims to find all elements that are associated with none of the members of a set of prohibitions[4]. In this paper, we are concerned with some relevant issues related to the improvement of queries combining both of required and forbidden associations.

1.1 The division and anti-division operators

Relational division is used when an element that satisfies a whole set of requirements is sought for. Whereas, the anti-division operator is used to select elements

that exclude any association with a set of prohibitions[6].
In relational algebra, the division (resp. anti-division) of relation $r(X,Y)$, called ***'dividend'***; by relation $s(Y)$, called ***'divisor'***; is a new relation $q(X)$, called ***'quotient'*** that includes some parts of $Projection(r,X)$ satisfying the following condition: x is in $q(X)$ if and only if x is in $Project(r,X)$ and for **all** (resp. **none**) y in $s(Y)$, $r(X,Y)$ contains (resp. doesn't contain) $< x,y >$[6]. X and Y are two compatible sets of attributes. More formally, the relational division is characterised by Equation 1, and the anti-division by Equation 2 :

$$Div(r,s,X,Y) = \{x \in projection(r,X) \mid \forall y, (y \in s) \Rightarrow (\langle x,y \rangle \in r)\} \quad (1)$$

$$Anti-Div(r,s,X,Y) = \{x \in projection(r,X) \mid \forall y, (y \in s) \Rightarrow (\langle y,s \rangle \notin r)\} \quad (2)$$

Example 1: Consider a distribution company of some products. In its commercial activity, the company wants to select its most valued customers (buyers). Customers ranking is based on some categories of products. Let ***Customer_Order*** $(\#customer, \#product, \#order\_state)$, ***Critical_Product*** $(\#product, \#order\_state)$ and ***Golden_Product***$(\#product, \#order\_state)$ be three crisp relations as sketched in Figure 1.

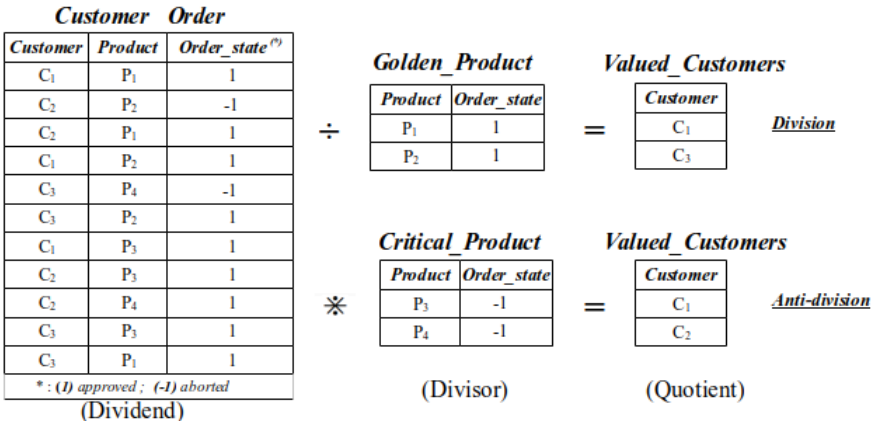

Fig. 1: Division query: ***"Which customers have made an approved order for each golden products?"****; Anti-division query:* ***"Which customers have not made an aborted order for any of the critical products?"***

In the figure above, C_1 and C_3 are the resulting quotients of the division because they have made an approved order for all golden products. Whereas, for the anti-division, C_1 and C_2 are the valid quotients, since both of them have not made an aborted order of any critical product.

1.2 Current trends

Both relational division and anti-division often provide an empty answer. This is a widely studied problem in the last two decades[7]. **Flexible operators** (tol-

erant operators and operators dealing with user's preferences), is the most desirable technique to solve this problem and improve the DBMS answer quality[8], especially for recommender systems[9]. Flexible division (anti-division) consists in the weakening of the quantifier all (none) used in the classical operator[6,10].

1.3 Related work and motivation

Two main areas of research on division and anti-division can be identified. The first concerns the improvement of those operators, while the second area investigate them in a flexible context.

In literature, several studies have been focused on how to efficiently implement the division, including those surveyed in [1,2,3] in the relational model, and [5] in the object-oriented model. Indeed, the approach proposed in [1] and called **'Hash-Division'**, has proven through the experimental results to be better than the traditional algorithms in processing time in most cases. Further, there are only as far as we know, the work of *Bosc et al.* for the relational anti-division[6,11]. Nonetheless, their implementation is based upon the SQL query derivation and is far from being optimal.

In the flexible area, some authors have suggested new operators for relational division[10,12] and anti-division [6,11,13], which are tailored for the flexible context. However, the performance aspect has not been adequately dealt with. Besides, some extended variants of the hash-division algorithm have been discussed in our earlier work [14] to tailor with some forms of the flexible division and division with preferences. However, to the best of our knowledge, the only experimentations done for the anti-division are those presented in [6,11]. Although, their implementation is based on the nested loops algorithm which is far from being acceptable. Moreover, queries evaluation are performed with a reduced size of data (dividend and divisor). This does not fit reality, especially for analysis treatments on extra-large databases. In addition, authors in [15] have suggested a way for combining the division and the anti-division operators. However, neither implementation nor experimentations are presented in the paper.

1.4 Main contributions in this paper

This paper is carried out as a continuation of our previous work detailed in [14], which is proven to be an efficient processing of the flexible division. Hence, extended variant will be proposed in this work to cover additional forms of the universal-quantification based queries.

In fact, the main purpose of our work is to design a unified processing to handle queries involving requirements and prohibitions simultaneously, with a single operator. Such queries allow users to express several kinds of their preferences, which is very useful in information systems especially in artificial intelligence.

We also address the performance enhancement of the new operator drawing to the Hash-Division strategy as used in our previous work [14].

Example 2: Let's take relations in the previous example. Thanks to the mixing query, customers can be evaluated through the following query: "*Find customers who have made an* ***approved order for all golden products*** *and they haven't made* ***any aborted order for the critical products***?".

Here, C_3 is no longer a valid quotient because he has made an aborted order of one of the critical products (P_2). Idem for C_2, he hasn't made an approved order for all golden product. Thus, we can conclude that the customers can be better distinguished through the mixed query. In addition, a unified (single operator) and fast processing of such queries will improve them even more. This is the backdrop behind our work. Hereafter we summarise our contributions:

- Investigate performance enhancement of the flexible queries involving both of division and anti-division, essentially for very large volumes of data.
- Investigate the parallel implementation feasibility for the extended approach.

We consider in this work the flexible division and anti-division over crisp databases exclusively. Fuzzy relations will be studied in future work.

1.5 Outline of the paper

The remainder of this paper is organised as follows. In Section 2, we present the classical Hash-Division algorithm. Section 3 gives an overview of the flexible division and the flexible anti-division. In Section 4, our contribution is presented together with analytics and discussion of the experimental results obtained. Section 5 introduces a parallel implementation of the proposed operator. Finally, Section 6 concludes the paper and suggests directions for future work.

2 Review of Hash-Division Algorithm

In this section, we give a brief description of the hash-division algorithm **(HD)** (see [1] for further details). It uses two hash tables, in order to avoid the exhaustive comparison, used in the traditional algorithms. The first table is for the divisor and the second for the quotient. Thanks to these two structures, both dividend and divisor relations are scanned exactly once, that makes the division operator faster. Hash-Division algorithm is proceeding in three stage:

Stage 01: Building the hash-divisor table : during the scan of the divisor table, we insert all divisor tuples into buckets in the hash-divisor table. Each entry in this table, is stored together with an integer called **divisor number *'Num_div'***. Num_div is initialized to 0 and it is incremented whenever a new insertion in the hash-divisor table occurs.

Stage 02: Building the hash-quotient table: during the scan of the dividend; for each row that corresponds to one of the divisors, stored in the hash-divisor table, we insert a quotient candidate into hash buckets in the hash-quotient table. Together with each inserted candidate, a bitmap is kept with one

bit for each divisor. All bits are initialized to 0, and updated to 1 whenever a match with the corresponding divisor occurred.

Stage 03 (end): Building the result: in this last stage, we select from the constructed hash-quotient table all quotient candidates whose bitmaps contain only ones as valid quotients.

3 Review of Flexible Division and Flexible anti-division

Flexible (or tolerant) division and anti-division were essentially proposed in order to avoid the empty result problem, which may occur mostly whenever we use **'for all'** or **'for none'** quantifiers [6,10]. There are a plethora of suggestions, in literature, showing that original relational division (anti-division) can be extended to different types of flexible queries. We are interested in this work on the following forms of flexible operators : *(i)* Exception-based tolerant division, *(ii)* Exception-based tolerant anti-division.

3.1 Principle

This category is based on exceptions into the requirements set for the division or the prohibitions set for the anti-division (divisor). The principle is to weak the quantifier *'all'* (resp. *'none'*) to the fuzzy quantifier *'almost all'* (resp. *'almost none'*) to express tolerant division (resp. anti-division)[6,10,12]. Thus, depending on the desired level of relaxation, some elements, in the divisor set, are allowed to be not associated (resp. associated) with the quotient in the dividend relation.

3.2 Modelling

In fact, a maximum number of exceptions is allowed to be ignored. Satisfaction-level SL of a quotient is measured by Equation 3 for the division and Equation 4 for the anti-division. A threshold is required for accepted quotients[10,13]. Valid quotients are sorted depending on their satisfaction levels.

$$SL_{Division} = \frac{Number\ of\ divisors\ associated\ with\ the\ candidate}{total\ number\ of\ divisors} \tag{3}$$

$$SL_{Anti-Division} = \frac{Number\ of\ divisors\ not\ associated\ with\ the\ candidate}{total\ number\ of\ divisors} \tag{4}$$

4 Our proposed approach for the mixed query

This section is devoted to a tolerant universal-quantification queries in which both division and anti-division are considered ***simultaneously***. We first give a novel way for combining those two types of associations: required and forbidden associations. Then, the performance of the proposed approach is highlighted.

In fact, we propose to improve the effectiveness of the mixed query by inspiring from the strategy of the hash-division algorithm. We have made various alterations to the structures and the procedures used in the classic algorithm, to deal with the unified mixed operator. Moreover, we describe an adequate technique to better discriminate final quotients, with no additional cost.
It should be noted that our work differs from *Bosc et al.*'s work presented in [15] in our formulating query. All preferences, requirements and prohibitions, are expressed thanks to a single operator. While the key *issue* with the approach presented in [15] is that is based on the decomposition of the mixed operator on several successive relational division and-or anti-division operations, depending on the number of layers, which is a very time-consuming process.

4.1 Strict and gradual Mixed Query

To deal with the mixed query, the divisor is subdivided into two sets, positive part (requirements) $\boldsymbol{P}$, and negative part (prohibitions) $\boldsymbol{N}$.

In the strict version, to be selected as a valid quotient, an element x must be associated with all values in P **and** must be not associated with any value in N. Thereby, P and N must be totally independents. In this strict version, all results are equally ranked. For the gradual mixed query, since some tolerances are allowed in both subsets P and N, results are discriminated depending on their satisfaction levels. Hence, for each accepted quotient we define two sub-level: $\mathbf{S}_p$ and $\mathbf{S}_n$ stand for the satisfaction level for the positive and the negative part respectively. S_p is computed as in Equation 3 with respect to the positive part, and S_n is computed as in Equation 4 regarding the negative part.

4.2 Hash-mixed query: an improvement of the mixed query

Here we will describe how we have improved the processing time of the mixed query relying on the Hash-Division like algorithm. Hence, the three altered phases of the hash-mixed query are described hereafter.

The first stage:
As in the classic algorithm, we store all divisor tuples in a hash table. Whereas for ours, each tuple is stored together with two integers:

- ***ind_lyr:*** index of the layer, 0 for P and 1 for N. This integer is used to indicate the offset of the divisor tuple inside the bitmap. Bits corresponding to divisors in P are located, in the bitmap, before those belonging in N.
- ***num_div_lyr:*** the divisor number (*rank*) of the tuple in its layer (P or N).

The data structure of a divisor tuple in the hash-divisor table is shown in the following figure.

Divisor value	ind_lyr	num_div_lyr

Fig. 2: Data structure of a hash-divisor tuple

Hence, for each layer, P and N, we keep its own divisors counter. These two counters are initialized to 0 and incremented whenever we insert a new divisor, of the corresponding layer, into the hash-divisor table. Pseudo-code of the hash-divisor table building for the mixed query is given hereafter:

Algorithm 1 Building of the hash-divisor table for the Hash-mixed query

```
num_divisors_P ← 0; num_divisors_N ← 0; /* initialize the two counters to zero */
for each tuple t in the divisor relation do
  Calculate its hash bucket (Hdiv);
  if t belongs in P then
    divisor.ind_lyr ← 0;
    divisor.num_div_lyr ← num_divisor_P; /*assign the offset to the current divisor*/
    num_divisor_P ++;
  end if
  if t belongs in N then
    divisor.ind_lyr ← 1;
    divisor.num_div_lyr ← num_divisor_N; /*assign the offset to the current divisor*/
    num_divisor_N ++;
  end if
  Insert the divisor tuple into the corresponding hash bucket (Hdiv);
end for
```

The second Stage:
In the second stage (*Construction of the Hash-quotient table*) of the hash-mixed query, we have made two major differences from the basic algorithm. The first is how to update the bitmap. Hence, if a divisor matching (P or N) with the quotient candidate occurs, we set the bit to 1 whose position, in the quotient bitmap[1], is equal to ***'offst_lyr+num_div_lyr'*** where:

- ***num_div_lyr***: the divisor number stored together with the matching divisor.
- ***offst_lyr***: is set to 0 if the matching divisor belongs to P, otherwise (belongs to N) it is set to $|P|$ (the cardinality of the positive subset).

Therefore, the data structure of the bitmap of candidates is as shown below:

.

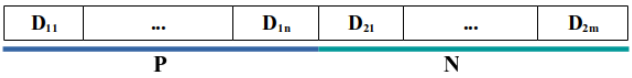

Fig. 3: Data structure of the bitmap for hash-mixed query.

The second difference is that we kept with each quotient candidate counters of ones ($bit = 1$), in its bitmap, for each layer. We called these counters $\boldsymbol{Nb\_ones}_1$ for the layer P and $\boldsymbol{Nb\_ones}_2$ for the layer N. These latter are incremented at each bit switching (0 to 1) in the corresponding layer of the quotient candidate bitmap. Hereafter is a pseudo-code of this stage:

[1] As in the basic version, the bitmap is initialized with 0 in all their bits

Algorithm 2 Building of the hash-quotient table for the hash-mixed query

for each tuple ***t*** in the dividend table **do**
  Calculate the hash bucket $Hdiv$ over the divisor attributes of the tuple t;
  if the divisor is contained in the hash-divisor table in the bucket Hdiv **then**
    layer ← ind_lyr of the matching divisor;
    rank ← num_div_lyr of the matching divisor;
    Calculate the hash bucket $Hqot$ over the quotient attributes of the tuple t;
    if the candidate (quotient value) is already contained in the hash-quotient table at the bucket Hqot **then**
      if $rank^{th}$ bit in the $layer^{th}$ part of the candidate bitmap is set to 0 **then**
        Set this bit to 1;
        Nb_ones $_{layer}$++; */*Increment the counter of ones in the corresponding layer*/*
      end if
    else {*/*quotient candidate does not yet exist*/*}
      Insert a new quotient candidate into the hash-quotient table at the bucket $Hqot$, with a bitmap where all bits are set to zero;
      Set the $rank^{th}$ bit in the $layer^{th}$ part of the bitmap to 1;
      Nb_ones $_{layer}$ ← 1; */*Initialize the counter of ones to 1*/*
      Set the other counter (Nb_ones $_{\overline{layer}}$) to 0;
    end if
  end if
end for

The third stage:

In the third stage, and for the strict version of the mixed query, quotient candidates whose bitmaps contain *only ones* in the positive part **and** *only zeros* in the negative part will be selected as valid quotients. Thereby, all final results will be equally ranked. Besides, for the tolerant version, we identify two manners to consider the satisfaction sub-levels (S_p and S_n) mentionned in subsection 4.1 in order to discriminate and rank the accepted quotients:

- **Strong symmetrical impact :** both of positive and negative part have the same impact on the result ranking. So the final satisfaction-level $\mathbf{S}_f$ is defined as '$\mathbf{S}_f$=$\mathbf{S}_p$+$\mathbf{S}_n$'. To sort accepted quotients, we propose to use a mechanism close to that used in our previous work, where we have used an indexed table for this sorting phase. Hence, a final quotient Q, whose satisfaction-level S_f is greater than the threshold chosen by the user, is stored in a bucket of index **'(|P|-Nb_ones$_1$)+ Nb_ones$_2$'** (see Figure 4 .a). Nb_ones$_1$ and Nb_ones$_2$ are the two counters stored with the bitmap of the quotient candidate.

- **Positive and negative part as hierarchical preferences:** here, the positive and the negative part haven't the same impact in results discrimination, one part is more important than the other. Indeed, an indexed table with two levels is used to rank valid quotients. The first level corresponds to the most important part. Each level is subdivided according to the number of exceptions allowed (seeFigure 4 .b). The positive bucket (level$_1$ or level$_2$) has the index '**|P|-Nb_ones$_1$**', while the negative one is equal to '**Nb_ones$_2$**'.

In such a way, final quotients are automatically sorted in decreasing order according to their satisfaction levels. The cell whose index is 0 points the best quotients (satisfying the whole set of requirements and dissatisfying all prohibitions). Hence, to select the $k - top$ answers, we just need to browse the indexed table from the top (from quotients with the highest satisfaction-level to the lowest ones); until k quotients are found. This sorting technique offers a better discrimination between accepted quotients, while no additional costs is needed.

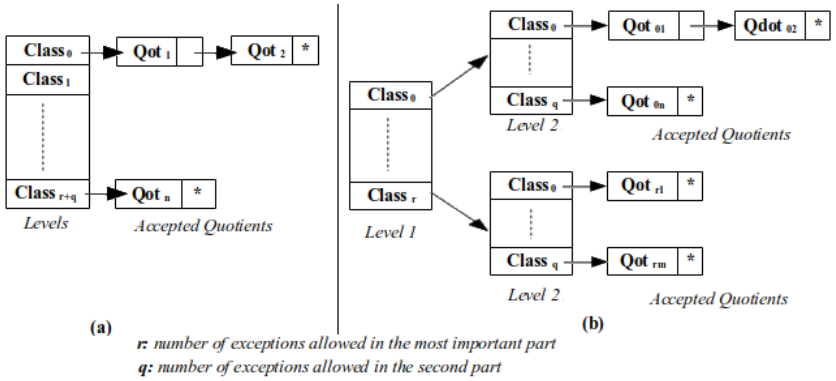

Fig. 4: Quotient Candidates Discrimination

In the light of the above, it can be said that we have been able to combine two types of associations (positive and negative) in a single operator. The conceived operator is not complex since it does not need to handle each operation (Division and anti-division) separately. Furthermore, it requires no iterations. Hence, thanks to th new unified operator, users can introduce simultaneously requirements and prohibitions in a constructively simple manner.

4.3 Experimentations

We consider four sizes for the dividend relation: 3.10^4, 5.10^5, 3.10^6, and 5.10^8 tuples, randomly generated[2]. Sizes considered for the divisor relation are: 10, 20, 50, and 100 uniformly distributed over layers P and N. Obtained results are gathered in Table 1. Run-time is measured in seconds.
Table 1 shows the run-times of our variants of the mixed query, comparing with the classic one presented in [15] where several successive classical-divisions are involved. We can notice that our approaches complete performance much faster than the classic one for the four dividend sizes. Indeed, the run-time is improved by several orders of magnitude (the gain factor is greater than ***61*** in the case of 5.10^8 dividend tuples). In addition, implementation requires roughly the same run-time regardless of the investigated form of the mixed query. For the largest

[2] In the literature, up to now and as far as we know, the largest set used in the experimentations never exceed a cardinality of 3.10^4 tuples in the dividend relation.

dividend relation: run-time is approximately equal to $120s$ for the three variants (strict form, symmetrical impact form, and the hierarchical form).

Table 1: Experimental results for the hash-mixed query algorithms

Size		Classical Mixed Query	Strict Hash-Mixed Query	Gradual Hash-Mixed Query	
Dividend	Divisor:P-N			SSI [a]	PN-HP[b]
$3x10^4$	5-5	3.11	0.01	0.02	0.02
$5x10^5$	5-15	87	0.165	0.173	0.169
$3x10^6$	30-20	598	3.44	3.42	3.5
$5x10^8$	50-50	7645	120.3	121.01	123.65

[a] : Strong symmetrical impact.

[b] : Positive and negative part as hierarchical preferences.

5 Parallel implementation

Parallel implementation is realized thanks to the **PVM** framework (**Parallel Virtual Machine**), on machines based on an ***Intel i5 CPU*** and ***8 Go RAM***. Experimentations were performed over 2, 4 and 6 nodes. The parallelism strategy is as follows:

1. The hash-divisor table is created only once on a single node called ***master***.
2. The master sends the hash-divisor table created to all other nodes.
3. The dividend table is uniformly partitioned between all nodes.
4. Each node builds its own hash-quotient table. The hash function may be different between the nodes, depending on the memory space of each one.
5. When all sub-tables of the hash-quotient are completely constructed in all nodes, the master collects those sub-tables. Then, it merges all of them in one global hash-quotient table to select valid quotients.

The pseudo-code of the last step (point 5), in the master, is given hereafter:

Algorithm 3 Parallel implementation of the mixed query.

for each sub-hash-quotient table received from the slave nodes **do**
  for each quotient-candidate in the sub-hash-quotient table **do**
    Compute the hash bucket (***Hqot***) using the master hash function, over the quotient value of the candidate;
    if the candidate (quotient value) is already contained in the hash-quotient table, constructed in the master, at the bucket *Hqot* **then**
      Update the bitmap of the candidate by calculating the result of the **binary OR** operator between the bitmap in the master and that received from the node;
    else
      Insert a new quotient candidate into the hash-quotient table of the master at the bucket *Hqot*, with a bitmap equal to that received from the node;
    end if
  end for
end for

As well, Figure 5 illustrates the speed-up behaviour of the parallel algorithm of the hash-mixed query over 2, 4 and 6 nodes.

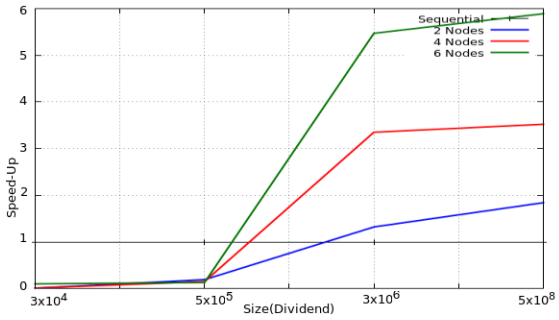

Fig. 5: Speed-up for parallel algorithm of the hash-mixed query.

Through the results obtained from the parallel implementation of the hash-mixed query and illustrated in the figure above, we observed a ***linear effect*** on speed-up in the case of large dividend ($\geq 3.10^6$). However, an additional cost[3], but still negligible, for a relatively small size of the dividend ($\leq 5.10^5$) occurs.

In summary, first results of the hash-mixed query presented in this paper are encouraging. The proposed approach has been successful in processing this complex forms of the universal-quantification based queries effectively. Although, there is still a need for multiple implementations in real SGBD, to firmly validate the hash mixed approach proposed.

6 Conclusion and perspectives

We have presented in this paper a unified operator to deal with universal quantification based queries involving positive and negative preferences (desired and forbidden associations) simultaneously. Our new technique is then improved relying on the hash-division algorithm. Moreover, the issue of answers ranking is dealt with. We have conducted some experiments particularly for large-sized relations, and compare execution time with the original approaches (nested loop algorithms) proposed in the literature. As expected, the performance got is very interesting. We have been able to improve the response time of some queries by several orders of magnitude. We presented also a parallel version of the mixed query, where we have obtained a near-linear speed-up, especially for large tables. We are currently designed new forms of complex queries where more than two layers, several kinds of preferences, and several connectors come to play. Furthermore, there is still a need for multiple implementations in real SGBD, to firmly validate the hash mixed approach proposed. It will also be exciting to look at other parallelism strategies which take into account the data skew issue that causes deteriorations in performance.

[3] The additional cost comes from the fact that the communication time between nodes is more expensive than the run-time of algorithms in each node.

References

1. Graefe, Goetz. “Relational division: Four algorithms and their performance.” Data Engineering, 1989. Proceedings. Fifth International Conference on. IEEE, 1989.
2. Rantzau, Ralf, et al. “Universal quantification in relational databases: A classification of data and algorithms.” International Conference on Extending Database Technology. Springer Berlin Heidelberg, 2002.
3. VAVERKA, Ondrej et VYCHODIL, Vilem. Relational division in rank-aware databases. Information Sciences, 2016, vol. 366, p. 48-69.
4. BOSC, Patrick et PIVERT, Olivier. On some uses of a stratified divisor in an ordinal framework. In : Uncertainty Approaches for Spatial Data Modeling and Processing. Springer Berlin Heidelberg, 2010. p. 133-154.
5. MARIN, Nicols, MOLINA, Carlos, PONS, Olga, et al. Semantically-driven flexible division in fuzzy object oriented models. In: IFSA/EUSFLAT Conf.2009. p.1039-1044.
6. BOSC, Patrick, PIVERT, Olivier, et SOUFFLET, Olivier. Strict and tolerant antidivision queries with ordinal layered preferences. International Journal of Approximate Reasoning, 2011, vol. 52, no 1, p. 38-48.
7. BOSC, Patrick, HADJALI, Allel, et PIVERT, Olivier. Empty versus overabundant answers to flexible relational queries. Fuzzy sets and systems, 2008, vol. 159, no 12, p. 1450-1467.
8. ZADRONY, Sawomir et KACPRZYK, Janusz. Bipolarity in database querying: Various aspects and interpretations. In : Flexible Approaches in Data, Information and Knowledge Management. Springer International Publishing, 2014. p. 71-91.
9. PIGOZZI, Gabriella, TSOUKIAS, Alexis, et VIAPPIANI, Paolo. Preferences in artificial intelligence. Annals of Mathematics and Artificial Intelligence, 2016, vol. 77, no 3-4, p. 361-401.
10. BOSC, Patrick, PIVERT, Olivier, et SOUFFLET, Olivier. On three classes of division queries involving ordinal preferences. Journal of Intelligent Information Systems, 2011, vol. 37, no 3, p. 315-331.
11. BOSC, Patrick, PIVERT, Olivier, et SOUFFLET, Olivier. Anti-division queries with ordinal layered preferences. In : European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty. Springer Berlin Heidelberg, 2009. p. 769-780.
12. TAMANI, Nouredine, LIETARD, Ludovic, et ROCACHER, Daniel. Bipolarity and the relational division. In : The Joint 7th Conference of the European Society for Fuzzy Logic and Technology (EUSFLAT'11) and Rencontres Francophones sur la Logique Floue et ses Applications (LFA'11),. 2011. p. 424-430.
13. BOSC, Patrick et PIVERT, Olivier. A family of tolerant antidivision operators for database fuzzy querying. In : International Conference on Scalable Uncertainty Management. Springer Berlin Heidelberg, 2008. p. 92-105.
14. BENADJMI, Noussaiba et HIDOUCI, Khaled Walid. New Variants of Hash-Division Algorithm for Tolerant and Stratified Division. In : International Conference on Flexible Query Answering Systems. Springer, Cham, 2017. p. 99-111.
15. BOSC, Patrick et PIVERT, Olivier. Queries mixing positive and negative associations and their weakening. In : Fuzzy Information Processing Society (NAFIPS), 2010 Annual Meeting of the North American. IEEE, 2010. p. 1-6.