# New Variants of Hash-Division Algorithm for Tolerant and Stratified Division

Noussaiba Benadjmi(✉) and Khaled Walid Hidouci

Ecole Nationale Supérieure en Informatique(ESI),
BP 270, 160290 Oued-smar, Algiers, Algeria
{an_benadjmi,hidouci}@esi.dz

**Abstract.** Works done in the context of the relational division for DBMS led to several approaches. Among which, the Hash-Division algorithm proved its superiority compared to the other approaches in the most of the cases. Nowadays, current trends of division are been oriented towards flexible queries and those involving preferences. However, the emphasis was always on proposing new operators which provide more flexibility and tolerance than the classical division operator. The performance aspect has not been adequately addressed. The proposed approaches in the literature suffer from a lack of performance, especially in a large volume of data. In this paper, we attempt to address this problem. Our idea consists in exploiting the advantages offered by the classical Hash-Division algorithm to propose new variants tailored for the flexible context. We paid a special attention to the improvement of some extended tolerant operators. Furthermore, we introduce a parallel implementation of our proposed techniques. Experimental results show the efficiency of our proposition. We obtained a very satisfactory improvement in processing time (the gain exceeds a ratio of 20 in the majority of cases) in both sequential and parallel implementation.

**Keywords:** Relational division · Preferences · Tolerant division · Stratified division · Hash-division algorithm

## 1 Introduction

Relational division is an interesting type of queries. They are very useful to many applications, especially in business intelligence applications (on-line analytic processing OLAP, data mining ...), and in recommendation systems. In relational algebra, the division is the most complex operator. That's why several researchers focus on their implementation, algorithms and optimisation [1].

### 1.1 The Division Operator

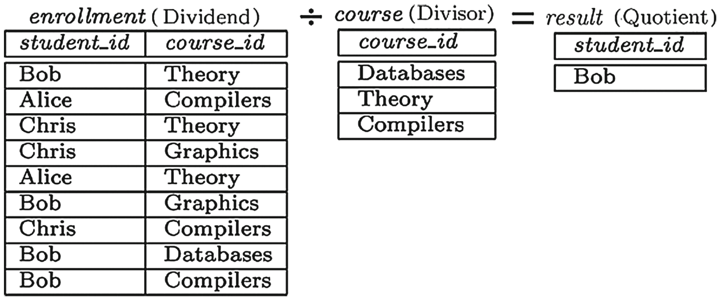
Relational division is used when an element that satisfies a whole set of requirements is sought for. In relational algebra, the division of relation *r(X, Y)*, called ***"dividend"***; by relation *s(Y)*, called ***"divisor"***; is a new relation q(X) that

DOI: 10.1007/978-3-319-59692-1_9

includes some parts of projection(r,X) satisfying the following condition: x is in q(X) iff x is in Project (r,X) and for **all** y in s(Y), r(X,Y) contain <x,y> [2]. More formally, the relational division is characterised by the formula (1):

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

To illustrate the division operator, we use the example sketched in Fig. 1, representing data from a department of a university [2]. In the figure above, Bob is the only resulting quotient because he is the unique student who is not missing any of the courses of the divisor.



*enrollment* (Dividend) ÷ *course* (Divisor) = *result* (Quotient)

| *student_id* | *course_id* |
|---|---|
| Bob | Theory |
| Alice | Compilers |
| Chris | Theory |
| Chris | Graphics |
| Alice | Theory |
| Bob | Graphics |
| Chris | Compilers |
| Bob | Databases |
| Bob | Compilers |

| *course_id* |
|---|
| Databases |
| Theory |
| Compilers |

| *student_id* |
|---|
| Bob |

**Fig. 1.** Division operation representing the query: ***"Which students have taken all Courses?".***

## 1.2 Related Works and Current Trends

Division operator has not a specific expression in SQL, because relational algebra does not directly support the quantifier "all" [1]. That's why, in literature, several studies have been focused on how to efficiently implement division [2], including those surveyed in [3] for the relational model, and [1] for the object-oriented model.

However, the approach proposed and detailed in [3], called ***Hash-Division***, has proven to be an effective algorithm. Hash-Division is based especially, as indicated by its name, on the *hashing mechanism*. The experimental results illustrated in the same paper demonstrate that the hash-division, in most cases, is far better than the traditional algorithms in processing time.

On the other hand, the relational division often provides an empty answer. This is a widely studied problem in the last two decades [4]. Flexible division (tolerant division and division dealing with users preferences), is the most desirable technique to solve this problem and improve the DBMS answers quality, especially in the context of recommendation systems [5]. Flexible division consists in the weakening of the universal quantifier all used in the traditional operator. In the literature, many research works, including [6,7] have suggested new operators for the relational division, which are tailored for the flexible context.

Similarly, there are many papers on preferences over simple queries such as the selection operator, most of them are surveyed in [8]. However, there are only few works on preferences over the division, on crisp and fuzzy databases [9].

### 1.3 Motivations and Main Contributions in This Paper

Researches done for the tolerant division, in literature, has concentrated on improving the quality of systems answer, by introducing new operators that deal with flexible division. Hence, an important, but largely overlooked aspect in these researches, is the performance issue. In fact, all the proposed approaches suffer from a lack of performance. Indeed, these approaches are mainly based on the nested loop algorithm. Experimental results have shown that the response-time is far from being acceptable and deteriorates significantly as data size increases. Even for recent research, queries evaluations are performed with a reduced size of data (dividend and divisor). This does not fit reality, especially with the advent of the Big Data, and analysis treatments on extra-large databases.

In this paper, we attempt to address the previous problem. To the best of our knowledge, processing flexible division, using Hash-Division is not yet investigated. Thus, the main purpose of our work is to improve the performance of the flexible division operator. Hereafter we summarise our contributions:

- Investigate performance enhancement of the flexible division essentially for very large volumes of data.
- Extend the Hash-Division algorithm to the following approaches:
  - Exception-based tolerant division.
  - Division with ordinal layered preferences (Stratified division).
- Propose a new efficient processing to better discriminate and rank the results.
- Examinate the feasibility of the parallel implementation for the extended approaches.

### 1.4 Outline of the Paper

The remainder of this paper is organised as follows. In Sect. 2, we present the classical Hash-Division algorithm. Section 3 gives an overview of the flexible division, their categories and their semantics. In Sect. 4, our contribution is presented with analytics and discussion of the experimental results obtained. Finally, Sect. 5 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 [3] for further detail). It uses two hash tables, one for the divisor and the other for the quotient. Thanks to these two structures, both dividend and divisor tables are scanned exactly once, that makes the division operator faster. Hash-Division algorithm is proceeding in three stages:

***Stage 01- Building the hash-divisor table:*** Here, we insert all divisor tuples into hash buckets in the hash-divisor table. Each entry is stored together with an integer called **divisor number *"Num_div"***. "Num_div" is initialized to zero, and incremented whenever a new insertion in the hash-divisor table has occurred.

***Stage 02- Building the hash-quotient table:*** For each dividend row that corresponds to one of the divisor tuples, 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. Each bit set to 1, indicate that the candidate is associated with the divisor corresponding to the bit position.

***Stage 03- Finding result in the hash-quotient table:*** In this last stage, during the scan of the constructed hash-quotient table, we select all quotient candidates whose bitmaps contain only ones as valid quotients.

## 3 Review of Flexible Division

Tolerant (or flexible) division was essentially proposed in order to avoid the empty result, which may occur mostly whenever we use **"for all"** quantifier [4,7]. There are a plethora of suggestions, in literature, showing that original relational division can be extended to different types of flexible queries. In this paper, we are interested in the flexible division over crisp databases exclusively. So, the approaches of flexible division being focused in our work are the following:

1. Exception-based tolerant division.
2. Flexible division over a stratified divisor (involving layered preferences).

### 3.1 Exception-Based Tolerant Division

In this category, the principle is to weak the universal quantifier all to the fuzzy quantifier almost all. Hence, for this **Gradual Division**, a maximum number of exceptions is allowed to be ignored (some elements, in the divisor set, are allowed to be not associated with the quotient) [9,10]. Satisfaction-level *(SL)* of a quotient is measured by the formula (2). A threshold is required for accepted quotients.

$$SL = \frac{Number\ of\ divisors\ associated\ with\ the\ quotient\ candidate}{total\ number\ of\ divisors} \tag{2}$$

### 3.2 Division Involving Layered Preferences

In the previous category, neither discrimination nor order is involved between the elements inside the divisor. However, in this category, depending on the users preferences, the divisor set is subdivided into layers $S_i$; $i = 1..n$. Each layer has a degree of importance, thereby an order between layers is guaranteed $S_1 > S_2 > ... > S_n$. Elements into the same layer have equal importances. The layers are linked with the connectors **"and if possible"**, **"or else"**, or **"and-or"**. Three queries (lets be Q1, Q2 and Q3) are defined (see [11,12] for further detail):

– ***Conjunctive queries (Q1):*** we search to find the best elements associated with $S_1$ **and if possible** $S_2$ **and if possible ... and if possible** $S_n$. Hence, a quotient $x$ must be associated with all values in $S_1$, and it is more preferred as it is connected with all values from $S_1$ to $S_p$ and $p$ is high (ideally $n$).
– ***Disjunctive queries (Q2):*** the purpose is to find the best elements associated with all values in $S_1$ **or else** $S_2$ **or else ... or else** $S_n$. $S_1$ is no longer mandatory. An element is more preferred as it is connected with all values of $S_k$ and $k$ is small (ideally 1).
– ***Full discrimination-based queries (Q3):*** we aim to find the best elements associated with all values in $S_1$ **and-or** $S_2$ **and-or ... and-or** $S_n$. The idea is to consider all layers for which a fully satisfaction occurs. An element is more preferred as it is associated with multiple layers highly important.

## 4 Our Proposed Approaches

Different contributions have been achieved for the flexible division. Nevertheless, the concerns were always on how to propose an efficient operator that supports flexible context, and to theoretically demonstrate it. So, the performance issue was not dealt with, whereas it is clearly a critical metric in information systems today, especially with the advent of big data and analysis processing on extra-large databases.

In the next section, we will address the tolerant and the stratified division, mentioned in Sect. 3, from a performance point of view. More specifically, we will present several proposing methods to improve the processing time of the flexible division relying on classic Hash-division like algorithm. Moreover, we propose new techniques to better discriminate quotients with no additional cost. Hence our approaches have the merit to be an efficient processing of the flexible division, in both processing time and answers quality.

### 4.1 Extend the Hash-Division for Gradual Tolerant Division (G-H-D)

In order to apply the classic hash-division approach on the gradual tolerant division, we have made some adaptations for the basic algorithm (described in Sect. 2). These adaptations are made in the second and the third stage. The first stage remains unchanged. In the second stage, we kept with each quotient-candidate a counter of ones ($bit$ = 1) in its bitmap. We called this counter ***Nb_ones***. The latter is incremented at each bit switching (0 $to$ 1) in the bitmap. Hereafter is a pseudo-code of this stage with our proposed adaptations:

**Algorithm 1.** Building of the hash-quotient table for G-H-D.

```
for each tuple t in the dividend table do
    Compute the hash bucket Hdiv on the divisor attributes of the tuple $t$;
    if the divisor value is contained in the bucket $Hdiv$ of the hash-divisor table then
        rank ← $num\_div$ of the matching divisor tuple;
        Compute the hash bucket Hqot on the quotient attributes of the tuple $t$;
        if the candidate (quotient value) is contained in the hash-quotient table at the hash bucket $Hqot$ then
            if the $rank^{th}$ bit in the bitmap of the candidate is set to 0 then
                Switch this bit to 1;
                Nb_ones← Nb_ones +1; /*Increment the number of ones*/
            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 except the $num\_div^{th}$ bit;
            Nb_ones ← 1; /*Initialize the number of ones to 1*/
        end if
    end if
end for
```

In the tolerant division, quotients haven't the same satisfaction level. Hence, the final stage must discriminate these quotients. To this end, we have, radically, changed the third stage of the basic approach (Sect. 2). To sort the accepted quotients, according to their satisfaction levels, we propose an efficient technique which involves no additional cost. We use an index table to improve the sorting phase. During the scan of the final hash-quotient table, we proceed as follows:

– Each accepted candidate, whose satisfaction level is greater than the predefined threshold, is stored in a bucket of index $\mathbf{class}_d$ ($class_0 > class_1 > ... > class_{max\_exception}$), where $\boldsymbol{d}$ is the number of zeros (**Nb_zeros**) in the candidate bitmap, representing the number of the missing divisors (Nb_zeros = total number of divisors − Nb_ones); and ***max_exception*** is the maximum number of exceptions allowed. The latter, chosen by the user, represent the threshold.
– Candidates having zeros superior to *max_exception*, are rejected.

*K-top Answers.* Introducing tolerance into the queries mostly returns an overabundant answer, especially for the very large sized dividend. Therefore, ***k-top answer selection*** is paramount ($k$ is chosen by the user) [13]. However, in the classic approaches, *k-top* answers selection requires an additional sorting phase which can be expensive for a big number of results. Whereas, in our approach, no sort is needed. To select the *k-top* answers, we just have to browse through the index table, constructed in the third stage, from the highest satisfaction level quotients to the lowest ones; until $k$ quotients are found. Thus, no additional costs will be incurred.

*Implementation and Experimental Results.* Hereafter, we consider, in all our implementations, four sizes of the dividend relation: $\mathbf{3.10^4}$, $\mathbf{5.10^5}$, $\mathbf{3.10^6}$ and $\mathbf{5.10^8}$ tuples[1] with a divisor of different cardinality (resp. **10, 20, 50, 100** tuples). We run all experiments on a machine with an ***Intel i5 CPU*** and ***8 Gb RAM***.

Figure 2 shows the run-times of our approach *G-H-D*, comparing with the nested loop algorithm presented in [9,10]. We notice that our own performs much faster than the classic one for the four sizes. So, the run-time is improved by several orders of magnitude (a factor of **241** in the case of $5.10^8$ dividend tuples).

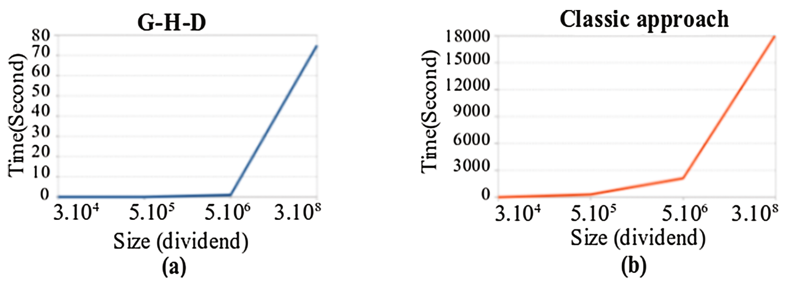


**Fig. 2.** Algorithm performance of G-H-D and Classic approach.

## 4.2 Extend Hash-Division for Stratified Division (S-H-D)

Here, for the stratified division involving layered preference, we have made some changes in both data structure and proceeding of our basic hash-division.

**Stage 01:** In the three variants of stratified division (Q1, Q2 and Q3), the first stage (*Construction of the Hash-divisor table*) is the same. As in the classic approach, we store all divisor tuples in a hash table (see Sect. 2). Each divisor, belonging to the layer $S_i$, is stored into the hash-divisor table together with two integers:

- ***offst_strt:*** number of divisor tuples in all previous layers $S_j$ $j < i$. This number indicates the offset of the layer inside the bitmap. Bits corresponding to divisors in $S_i$ are located, in the bitmap, between *offset(*$S_i$*)* and *offset(*$S_{i+1}$*)*.
- ***num_div_strt:*** the divisor number (*rank*), of the tuple, in its layer.

The data structure of a divisor in the hash-divisor table is shown in Fig. 3.

| divisor_value | num_div_strt | offst_strt |
|---|---|---|

**Fig. 3.** Data structure of a hash-divisor tuple.

[1] In the literature, up to now, the largest dividend used never exceed 30000 tuples.

Hence, for each layer $S_i$, we first calculate its ***offst_strt***. Then, we keep for each one, its own divisors counter. All counter are initialized to 0. Whenever we insert a new divisor, of the layer $S_i$, into the hash-divisor table, the divisors counter of $S_i$ will be incremented. The pseudo-code used is as follows:

**Algorithm 2.** Calculation of the offset of the layers for G-H-AD.

```
offst_str = array [ 1: Nb_layers ] of integer;
offst_str_i ← 0;
for i ← 1 to Nb_layers do
    offst_str [ i ] ← offst_str_i;
    offst_str_i ← offst_str_i + $|S_i|$; /*$|S_i|$ is the cardinality of the layer $S_i$ */
end for
```

As well, the pseudo-code of the hash-divisor table building is given hereafter:

**Algorithm 3.** Building of the hash-divisor table for S-H-D.

```
num_divisor_str = array [1 : $Nb\_layers$] of integer;
initialize all cells of num_divisor_str to zero;
for each tuple in the divisor relation do
    compute its hash bucket ($Hdiv$) in the hash-divisor table;
    $S_i$ ← layer number of the current divisor;
    divisor.offst_strt ← offst_str [$S_i$]; /*assign the layer offset to the current divisor*/
    divisor.num_div_strt ← num_divisor_str [$S_i$];
    num_divisor_str[$S_i$] + +;
    insert the divisor tuple into the corresponding hash bucket;
end for
```

**Stage 02:** In the 2nd stage, we proceed as in the classic hash-division (Sect. 2). The only difference is how to update the bitmap. Hence, if a divisor is associated to a quotient candidate, we set the bit to 1 whose position, in the bitmap, is equal to ***"offst_strt+num_div"***, stored with the matching divisor. The data structure of the bitmap is shown in Fig. 4.

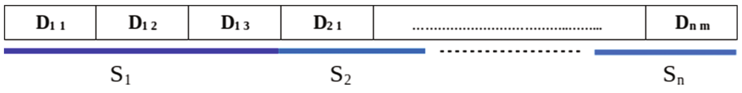


**Fig. 4.** Data structure for the bitmap for S-H-D.

**Stage 03:** Here, we have to compute the satisfaction level for each quotient candidate, in order to classify them thereafter. Hence, for the three queries (Q1, Q2 and Q3), we describe how to calculate these satisfaction levels. Moreover, we propose a fast mechanism to better discriminate the accepted candidates. Again, our sorting technique involves no additional cost and maintains the quality of answers.

*Conjunctive Queries (Q1):* We search to find the best $k$ elements associated with $S_1$ **and if possible** $S_2$ **... and if possible** $S_n$ (see Subsect. 3.2). Therefore, we proceed as follows:

- Candidates having zeros in the first layer, in their bitmaps, are immediately rejected.
- We check, in ascending order of layers, if the layer $S_i$ contains only ones in the bitmap. If yes, we pass to the next layer. Otherwise, the candidate is labelled by $\{\boldsymbol{i}, \textbf{\textit{Nb\_ones\_i}}\}$ where:
  - **i,** $i \geq 2$ **:** index of the first layer not fully satisfied.
  - **Nb_ones_i,** $0 \leq Nb\_ones\_i \leq |S_i|$**:** number of ones in the layer of index $i$. This number is used to discriminate candidates having equal labels $i$. It is worth noticing that in the classic approach presented in [11,12] this discrimination does not occur. Thus, candidates having $|\boldsymbol{S}_i| - \boldsymbol{1}$ ones, in S$_i$, and candidates with **no ones**; will be ranked with a similar satisfaction-level. This we avoid in our technique, thanks to the label *Nb_ones_i*.
- If the layer $S_i$ is the last one, label $i$ takes the value of the $S_n$ index (last layer), and $Nb\_ones\_i$ takes the value of its cardinality ($|S_n|$).

To sort the accepted quotients, depending on their satisfaction levels, we use an index table, as in G-H-D. Its size is "***n (total number of divisors)*** $-$ $|\boldsymbol{S}_1|$ ***+ 1***". Candidate labelled with $\{i, Nb\_ones\_i\}$ are stored into the bucket whose index is equal to "***n*** $-$ ***(offst_strt (S***$_i$) + ***Nb_ones_i)***". Thereby, final quotients will be, automatically, sorted in decreasing order according to their satisfaction levels. The cell whose index is 0, points the most preferred quotients.

*Disjunctive Queries (Q2):* In the case of Q2, $S_1$ is no longer an obligation. The first check (whether $S_1$ is fully satisfied) is no longer present. Therefore, we proceed as follows:

- Starting by the first layer, we check if $S_i$ contains only ones. If not, we pass to the next layer. Otherwise, the candidate is labelled by $\{\boldsymbol{i}, \textbf{\textit{Nb\_zeros\_ip}}\}$:
  - **i,** $i \geq 1$**:** the first layer fully satisfied.
  - **Nb_zeros_ip,** $1 \leq Nb\_zeros\_ip \leq |S_{i-1}|$**:** number of zeros in the previous layer $S_{i-1}$ (the last layer that contains zeros). As for Q1, this number is used to better discriminate candidates having equal labels $i$.
- Whenever $S_i$ is the first layer, we assign to $i$ the value **1** and **0** to *Nb_zeros_ip*.

As in Q1, we store the accepted quotients in an index table whose size is "$\boldsymbol{n} - |S_n| + \boldsymbol{1}$". Candidate with label $\{i, Nb\_zeros\_ip\}$ are stored into the bucket of index equal to "***offst_strt (S***$_{i-1}$) + ***Nb_zeros_ip***". Idem, final quotients will be sorted in decreasing order according to their satisfaction levels.

*Full Discrimination-Based Queries (Q3):* Here we have to consider all layers that are fully satisfied. Candidates are labelled by a procedure identical to that used in Q2. Nevertheless, accepted quotients would not be totally sorted. Hence, for each bucket in the index table, depending on the *k-top* value, we sort the candidates depending on their satisfaction levels using a traditional sorting algorithm.

**Experimentations:** To examine the performance of our proposed approaches for stratified division, we make experimental comparisons between ours own and the classic approaches implemented in [11,12]. Queries Q1, Q2 and Q3 are evaluated over a dividend relation having the four sizes previously mentioned ($3.10^4$, $5.10^5$, $3.10^6$ and $5.10^8$ tuples). The divisor relation has the cardinalities: 10, 20, 50 and 100 uniformly distributed over five layers. Figure 5 illustrates the results obtained.

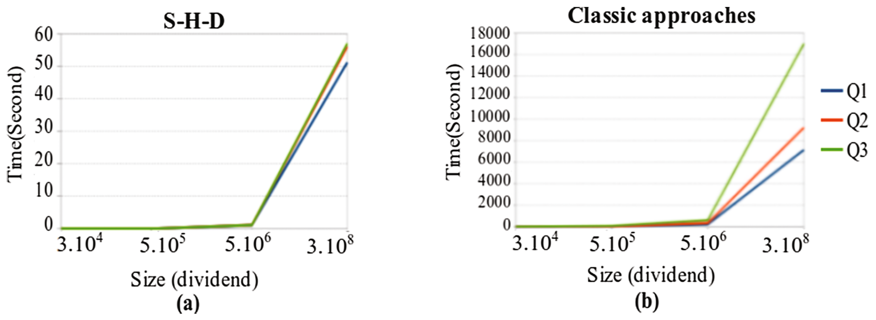


**Fig. 5.** Algorithm performance of S-H-D and the classic approach.

Analysis of the curves above leads to the following notices:

– Hash-division approach for the three queries (Q1, Q2 and Q3) proves its efficiency in run-time, compared to the classical one, especially for the overly large-sized dividend. The gain on time in this case was ***extremely high***: for the size $5.10^5$ in Q1, the improvement was from **12,5** to **0,085** seconds; and for the size $5.10^8$ in Q2, the improvement was from **9213** to **56** s.
– In the classic approach, the cost of queries Q1 and Q2 are relatively close. This is done thanks to the stop criteria used. However, Q3 is more expensive as it has to examine all layers. Whereas, for our own, all three queries involve, roughly, the same cost. Hence, it is totally independent on how to introduce preferences. This makes our approaches very beneficial.

## 4.3 Parallel Implementation

Our objective behind the parallel experiments is to demonstrate the parallelism feasibility of the proposed approaches (**G-H-D** and **S-H-D**). Parallel implementation is realized thanks to the **PVM** framework (**Parallel Virtual Machine**), on machines based on an ***Intel i5 CPU*** and ***8 Gb RAM***. Experimentations were performed over 2, 4 and 6 nodes. The parallelism strategy is as follows:

- The hash-divisor table is created only once on a single node called ***master***.
- The master sends the hash-divisor table created to the other nodes.
- The dividend table is uniformly partitioned (*horizontally*) between all nodes.
- Each node builds its own hash-quotient table.
- The master collects the sub-tables, of the hash-quotient, constructed in the nodes. Then, it merges all of them in one global table to select valid quotients. The pseudo-code of this last step, is given hereafter:

**Algorithm 4.** Parallel implementation of S-H-D.

```
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) over the quotient value of the candidate;
        if the candidate (quotient value) is contained in the hash-quotient table,
            constructed in the master, at the bucket Hqot then
            Update the candidate bitmap through performing a binary OR) operation
                between the two bitmaps (the bitmap in the master and that's in the
                node);
        else
            Insert a new candidate into the hash-quotient table of the master at the
                bucket Hqot, with a bitmap equal to that's present in the candidate
                received from the node;
        end if
    end for
end for
```

Hereafter, we restrict to describe the empirical results of Q1. Hence, Fig. 6 illustrates the speed-up behaviour of the parallel hash-division of Q1.

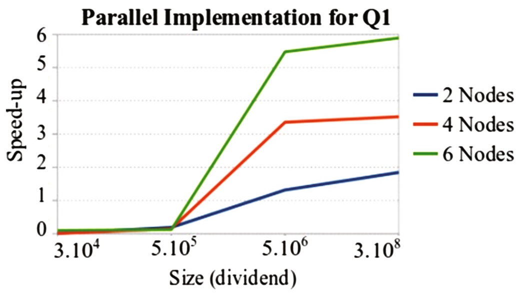


**Fig. 6.** Speed-up for parallel S-H-D algorithm of type Q1.

Although the parallelism of our approach involved an additional cost, but still negligible, for a relatively small size of the dividend ($3.10^4$ and $5.10^5$), it comes very close to linear speed-up in the case of large dividend ($3.10^6$ and $5.10^8$). Thus, experiments show that our approaches are more optimal when processing tolerant and stratified division over a parallel framework. Our approach is proved to be more efficient than some recent research, using highly-parallel new techniques, namely MapReduce framework [14].

## 5 Conclusion and Perspectives

We have presented in this paper two new variants **(G-H-D and S-H-D)** of the basic Hash Division algorithm for computing some tolerant division operators (Quantitative tolerant division and Tolerant division involving layered preferences). We have conducted some experiments, particularly for large-sized relations, and compare execution time with the original approaches (nested loop algorithms) proposed for the tolerant division operators studied. We presented also a parallel approach of the new variants of hash-division algorithm for tolerant division. This parallel approach have a near-linear speed-up, especially for large tables. As expected, the performance obtained, both for sequential and parallel versions, are very interesting. We have been able to improve the response time of some queries by several orders of magnitude. This opens up many perspectives in some data analysis using universal quantification and handling preferences over very large volumes of both usual and fuzzy data.

## References

1. Habib, W.M., Mokhtar, H.M., El-Sharkawi, M.: Processing universal quantification queries using mapreduce. In: International Conference on Big Data and Smart Computing (BIGCOMP). IEEE (2014)
2. Rantzau, R., Shapiro, L., Mitschang, B., Wang, Q.: Universal quantification in relational databases: a classification of data and algorithms. In: Jensen, C.S., Šaltenis, S., Jeffery, K.G., Pokorny, J., Bertino, E., Böhn, K., Jarke, M. (eds.) EDBT 2002. LNCS, vol. 2287, pp. 445–463. Springer, Heidelberg (2002). doi:10.1007/3-540-45876-X_29
3. Graefe, G.: Relational division: four algorithms and their performance. In: Fifth International Conference on Data Engineering, Proceedings. IEEE (1989)
4. Bosc, P., Hadjali, A., Pivert, O.: Empty versus overabundant answers to flexible relational queries. Fuzzy Sets Syst. **159**(12), 1450–1467 (2008)
5. Pigozzi, G., Tsoukiàs, A., Viappiani, P.: Preferences in artificial intelligence. Ann. Math. Artif. Intell. **77**(3–4), 361 (2016)
6. Galindo, J., et al.: Relaxing the universal quantifier of the division in fuzzy relational databases. Int. J. Intell. Syst. **16**(6), 713–742 (2001)
7. Bosc, P., HadjAli, A., Pivert, O.: La notion de division tolèrante et son intèrêt pour remédier aux réponses vides. Ingénierie des systèmes d'information **13**(5), 131–154 (2008)
8. Kacprzyk, J., Zadrony, S., De Tre, G.: Fuzziness in database management systems: half a century of developments and future prospects. Fuzzy Sets Syst. **281**, 300–307 (2015)

9. Bosc, P., Pivert, O., Rocacher, D.: About quotient and division of crisp and fuzzy relations. J. Intell. Inform. Syst. **29**(2), 185–210 (2007)
10. Bosc, P., Pivert, O., Rocacher, D.: A propos de la sémantique de divers opérateurs de division de relations. In: INFORSID (2006)
11. Bosc, P., Pivert, O., Soufflet, O.: On three classes of division queries involving ordinal preferences. J. Intell. Inform. Syst. **37**(3), 315–331 (2011)
12. Bosc, P., Pivert, O.: On some uses of a stratified divisor in an ordinal framework. In: Kacprzyk, J., Petry, F.E., Yazici, A. (eds.) Uncertainty Approaches for Spatial Data Modeling and Processing. SCI, vol. 271, pp. 133–154. Springer, Heidelberg (2010)
13. Wu, Y., Liu, G., Liu, Y.: Multi-user preferences based top-k query processing algorithm. In: Tenth International Conference on Computational Intelligence and Security (CIS). IEEE (2014)
14. Habib, W.M., Mokhtar, H.M., El-Sharkawi, M.: A new approach for scholars matching using universal quantifier queries. In: IEEE World Congress on Services (SERVICES). IEEE (2015)