

A view selection algorithm with performance guarantee ^{*}

Nicolas Hanusse
LaBRI
University of Bordeaux 1
CNRS. UMR5800
hanusse@labri.fr

Sofian Maabout
LaBRI
University of Bordeaux 1
CNRS. UMR5800
maabout@labri.fr

Radu Tofan
LaBRI
University of Bordeaux 1
INRIA Futurs
radu.tofan@labri.fr

ABSTRACT

A view selection algorithm takes as input a fact table and computes a set of views to store in order to speed up queries. The performance of view selection algorithm is usually measured by three criteria: (1) the amount of memory to store the selected views, (2) the query response time and (3) the time complexity of this algorithm. The two first measurements deal with the output of the algorithm. No existing solutions give good trade-off between amount of memory and queries cost with a small time complexity. We propose in this paper an algorithm guaranteeing a constant approximation factor of queries response time with respect to the optimal solution. Moreover, the time complexity for a D -dimensional fact table is $O(D * 2^D)$ corresponding to the fastest known algorithm. We provide an experimental comparison with two other well known algorithms showing that our approach also gives good performance in terms of memory.

1. INTRODUCTION

In On Line Analytical Processing applications the focus is to optimize query response time. To do so, we often resort to pre-computing or, equivalently, materializing query results. However, due to space or time limitations, we cannot store the result of all queries. So, one has to select the *best* set of queries to materialize. In the multidimensional model, more precisely when considering datacubes, relationships between the views can be used in order to define what is the best set of views. A view selection algorithm in the context of datacubes takes as input a fact table and returns a set of views to store in order to speed up queries. The performance of the view selection algorithms is usually measured by three criteria: (1) the amount of memory to store the selected views, (2) the query response time and (3) the time complexity of this algorithm. The two first measurements deal with the

^{*}This work has been partially supported by ANR ALLADIN and the DGE of French ministry of industry in RECORDS project framework.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the ACM. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires a fee and/or special permissions from the publisher, ACM. *EDBT 2009*, March 24–26, 2009, Saint Petersburg, Russia. Copyright 2009 ACM 978-1-60558-422-5/09/0003 ...\$5.00

output of the algorithm. Most of the works proposed in the literature consider the problem of finding the *best* data to store in order to optimize query evaluation time while the memory space needed by these data does not exceed a certain limit fixed by the user. There are some variants of this problem depending on (1) the nature of data that can be stored, e.g only datacube views or views and indexes, (2) the chosen cost model e.g minimize not only the query response time but also the maintenance time for the stored views or (3) the set of possible queries e.g the user may ask all possible queries or just a subset of them. In this last case, the problem may be refined by taking into account the frequency of queries. To the best of our knowledge, none of the existing solutions give good trade-off between memory amount and queries cost with a reasonable time complexity. Generally, the performance is stated by experiments results without theoretical proof. In this paper, we propose an algorithm whose output, the selected views to be stored, guarantees a constant approximation factor of queries response time with respect to the optimal solution. Moreover, the time complexity of this algorithm for a D -dimensional fact table is in $O(D * 2^D)$ corresponding to the fastest known algorithm given in [15]. We provide an experimental comparison with two other well known algorithms showing that our approach not only guarantees query response time but also provides good performance in terms of memory.

This paper is organized as follows: in the next section we introduce some notations, we define formally the notion of performance factor and we motivate its use. Then we review some of the related works proposed so far. In section 3, we first introduce some basic algorithms that return partial datacubes to be materialized. These are not very efficient. However they serve as an introduction to the solution we propose. Then we propose our algorithm and prove its performance. In section 4, we describe some of the experiments we have conducted and compare our performances to those obtained by two other algorithms proposed in the literature. We terminate by a conclusion summarizing our present work and where we propose some interesting future research directions.

2. PRELIMINARIES

2.1 Notation

A fact table T is a relation where the set of attributes is divided into two parts: the set of dimensions $Dim(T)$ and the set of measures $Measure(T)$. In general, $Dim(T)$ is a key of T . The datacube [6] built from T is obtained by aggregating

T and grouping its tuples in all possible ways i.e all *Group By* c where c is a subset of $Dim(T)$. Each c corresponds to a *cuboid*¹. The datacube defined from T is denoted $DC(T)$. For notation convenience $Dim(DC(T))$ will denote the set $Dim(T)$. Let $DC(T)$ be a datacube, $Dim(T)$ its dimensions and $|Dim(T)| = D$. The set of cuboids of $DC(T)$ is denoted by $\mathcal{C}(T)$. Clearly $|\mathcal{C}(T)| = 2^D$. The fact table T is a distinguished cuboid of $\mathcal{C}(T)$. It is called *base cuboid* and is denoted c_b . The size of a cuboid c is expressed by the number of its rows and is denoted $size(c)$. The size of a set of cuboids \mathcal{S} is denoted by $|\mathcal{S}|$. For notation convenience, from now on we will omit the parameter (T) since T will be clear from the context.

The datacube lattice is induced by a partial order relationship \preceq between cuboids defined as follows: $v \preceq w$ iff $Dim(v) \subseteq Dim(w)$. We say that w is an *ancestor* of v . Moreover, if $|Dim(w)| = |Dim(v)| + 1$ then w is a *parent* of v . Actually, if $v \preceq w$ then v can be computed from w .

2.2 Performance measures of a view selection algorithm

Assume the queries that are asked against DC are all and only those of the form **select * from c** or equivalently **select * from T group by c** where c is a cuboid from \mathcal{C} . There are two extremal situations that can be considered here. The first one is that where only the base cuboid c_b is stored (materialized). In this case, every query requires the use of this cuboid and hence has a time cost proportional to the size of c_b . The other situation is that where all cuboids are materialized. In this latter case, the evaluation of each query consists just in scanning the corresponding cuboid making its cost proportional to the actual size of the cuboid. Of course, this last situation is quite unrealistic since in practice, we often do not have enough memory (or time) to compute and store the whole datacube. What is often done is rather a partial materialization. In order to formally state the problem we try to solve, let us first introduce some notations.

Let $\mathcal{S} \subseteq \mathcal{C}$ be the set of materialized cuboids and v be a cuboid. Then, $\mathcal{S}_v = \{w \in \mathcal{S} | v \preceq w\}$ is the set of materialized cuboids from which v can be computed. We define the cost of evaluating a query v w.r.t a set \mathcal{S} as follows: if \mathcal{S} does not contain any ancestor of v then $cost(v, \mathcal{S}) = \infty$ otherwise $cost(v, \mathcal{S}) = \min_{w \in \mathcal{S}_v} size(w)$. That is, a query is evaluated by using one of its stored ancestors. The chosen ancestor is the one with fewer tuples. This is the measure usually used to estimate the time complexity (see e.g [9, 15, 16]). Note that when $v \in \mathcal{S}_v$ then $cost(v, \mathcal{S}) = size(v)$. This is the most advantageous situation for v . We also define the cost of a set \mathcal{S} as the cost of evaluating all queries w.r.t \mathcal{S} . More precisely, $cost(\mathcal{S}) = \sum_{c \in \mathcal{C}} cost(c, \mathcal{S})$. When $\mathcal{S} = \mathcal{C}$ i.e all cuboids are stored, we have $cost(\mathcal{S}) = \sum_{c \in \mathcal{C}} size(c)$. This is the minimal cost and will be denoted $MinCost$. If $\mathcal{S} = \{c_b\}$ then $cost(\mathcal{S}) = |\mathcal{C}| * M$ where M is the size of c_b . This is the maximal cost and will be denoted $MaxCost$. Thus for every \mathcal{S} , we have $\sum_{c \in \mathcal{C}} size(c) \leq cost(\mathcal{S}) \leq |\mathcal{C}| * M$. Note that since the set of possible queries includes c_b then \mathcal{S} should contain c_b , otherwise $cost(\mathcal{S}) = \infty$. Indeed, the base

¹Here after, we will use equivalently the terms *cuboid*, *view* and *query*.

cuboid can be computed only from the fact table. Thus, in the definition of $MaxCost$ we have considered only the sets \mathcal{S} that contain c_b .

The usual performance measures of a view selection algorithm \mathcal{A} are:

- *the memory*: $Mem(\mathcal{S}) = \sum_{c \in \mathcal{S}} size(c)$, the amount of memory required to store \mathcal{S} ;
- *the query cost* or *cost*: $cost(\mathcal{S})$ is proportional to the time to answer the 2^D possible grouping/aggregate queries;
- *the time complexity*: of the view selection algorithm.

We also propose a new performance measure called the *performance factor*.

DEFINITION 1 (PERFORMANCE FACTOR). *Let \mathcal{S} be the set of materialized cuboids and c be a cuboid of \mathcal{C} . The performance factor of \mathcal{S} with respect to c is defined by $f(c, \mathcal{S}) = \frac{cost(c, \mathcal{S})}{size(c)}$. The average performance factor of \mathcal{S} with respect to $\mathcal{C}' \subseteq \mathcal{C}$ is defined by $\tilde{f}(\mathcal{C}', \mathcal{S}) = \frac{\sum_{c \in \mathcal{C}'} f(c, \mathcal{S})}{|\mathcal{C}'|}$*

Intuitively, the performance factor measures the query response time of a cuboid with a given materialized sample \mathcal{S} with respect to the query time whenever the whole datacube is stored. In other words, for a query c , we know that the minimal cost to evaluate it corresponds $size(c)$. This is reached when c itself is materialized. When c is not materialized, it is evaluated by using one of its ancestors present in \mathcal{S} . Thus, the performance factor for c measures how far is the time to answer c from the minimal time. The goal is to obtain the answer to a query with a time proportional to the size of the answer.

EXAMPLE 1. *Consider the graph of Figure 1. It represents the datacube lattice obtained from a fact table T whose dimensions are A, B, C, D and E . We will use this datacube as our running example throughout the paper. The measure attributes are omitted. Each node is a cuboid and is labeled with its dimensions together with its size. We will consider this datacube as our running example throughout this paper. There is an edge from c_1 to c_2 iff c_1 can be computed from c_2 , $c_1 \neq c_2$ and there is no c_3 such that $c_3 \neq c_1$, $c_3 \neq c_2$, c_3 can be computed from c_2 and c_1 can be computed from c_3 ². The top most cuboid is the base cuboid and corresponds to the fact table. The minimal cost for evaluating all queries corresponds to the case where each cuboid is precomputed and stored. Thus, $MinCost = \sum_{i=1}^{2^5} size(c_i) = 8928$. In contrast, the maximal cost corresponds to the situation where only the base cuboid is stored. In this case, every query is computed from $ABCDE$ and thus has a cost proportional to the base cuboid size. Hence $MaxCost = 2^5 * size(ABCDE) = 2000 * 32 = 64000$. Notice however that this is the minimal amount of memory we must use in order to be able to answer all queries.*

²This is the minimal cover of \preceq relationship.

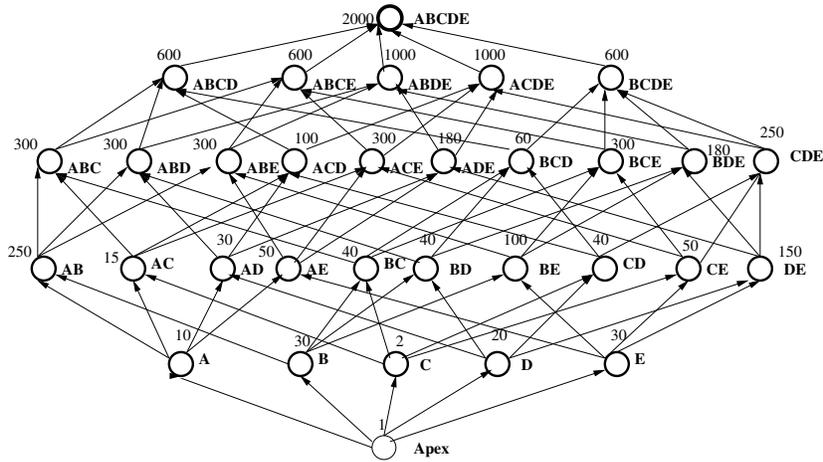


Figure 1: A datacube example

Assume now that $\mathcal{S} = \{ABCDE, BE\}$. The performance measures of \mathcal{S} are as follows: The memory required to store \mathcal{S} is $Mem(\mathcal{S}) = size(ABCDE) + size(BE) = 2000 + 100 = 2100$. The cost for evaluating all the 2^5 possible queries is calculated as follows. First, consider the stored cuboid BE . It can be used to compute the queries BE, B, E and $Apex^3$. All these queries can be computed from $ABCDE$ too. However this second alternative will require more time than the first one. Thus, the cost of \mathcal{S} corresponds to the sum of costs of evaluating BE, B, E and $Apex$ from the cuboid BE and all other queries (i.e $2^5 - 4$) from $ABCDE$. Hence, $Cost(\mathcal{S}) = 4 * size(BE) + 28 * size(ABCDE) = 56400$.

Let us now consider the cuboids BE and BC . Their respective performance factors w.r.t \mathcal{S} are $f(BE, \mathcal{S}) = \frac{cost(BE, \mathcal{S})}{size(BE)} = 100/100 = 1$ and $f(BC, \mathcal{S}) = \frac{cost(BC, \mathcal{S})}{size(BC)} = size(ABCDE)/40 = 2000/40 = 50$. This means that by storing $ABCDE$ and BE , the cost for evaluating the query BE is exactly the minimal cost, but for evaluating the query BC the cost is 50 times the minimal one.

2.3 Problem Statement

In this paper, we address the following problem:

Given a real number $f \geq 1$, find a set of cuboids \mathcal{S} of minimum size so that $cost(\mathcal{S}) \leq f * MinCost$ (a)

So we suppose that the user wants a set \mathcal{S} of cuboids which when materialized, the evaluation cost of queries does not exceed f times the minimal cost. Moreover \mathcal{S} should be of minimal size. Notice that the standard way in which the view selection problem is stated consists in fixing the maximal available memory space and selecting a set \mathcal{S} that respects this constraint and provides a good performance.

Given a memory space limit $space$, find a set of cuboids \mathcal{S} whose size is less than $space$ and which provides a minimal cost (b)

³Apex is the cuboid with no dimensions.

Even if these two ways of posing the problem are not equivalent, we will show in the next sections how to use our solution in order to solve problem (b).

The obvious solution to our problem consists simply in considering all subsets $\mathcal{S} \in 2^{\mathcal{C}}$, compute their respective costs, keep those \mathcal{S} satisfying $Cost(\mathcal{S}) \leq MinCost * f$ and then return \mathcal{S} whose size is the smallest. Of course this algorithm is unpractical because of its complexity. To the best of our knowledge, there is no solution proposed so far that provides this guarantee. This makes our essential contribution of the present work.

2.4 related works

Several solutions have been proposed in order to find the relevant subset of cuboids to store. Most of them suppose the the cuboids sizes known which is not realistic in general but this is not discussed in our paper. The real constraint is a bound on the available memory *space* and the main goal is to provide a subset of cuboids so that the cost is minimized. We note S^* the optimal solution of this problem, that is

$$cost(S^*) = \min_{\mathcal{S} \subseteq \mathcal{C} \text{ s.t. } Mem(\mathcal{S}) \leq space} cost(\mathcal{S})$$

where *space* denotes the available memory amount. In [9], the authors propose a greedy algorithm that returns a subset of views with a particular notion of guarantee. More precisely, they define the notion of *gain* as follows:

DEFINITION 2 (GAIN). Let $\mathcal{S} \subset \mathcal{C}$ be a set of cuboids. The gain of \mathcal{S} is defined by $cost(\{c_b\}) - cost(\mathcal{S})$.

One should have noticed that $cost(\{c_b\})$ is $MaxCost$. [9] shows that finding the optimal \mathcal{S} , i.e the one which maximizes the gain and respects the space constraint is an NP-complete problem. Thus they proposed an approximation algorithm⁴ whose performance guarantees that the gain of the returned solution cannot be less than 63% of that of the optimal solution S^* . In other words, $\frac{cost(c_b) - cost(\mathcal{S})}{cost(c_b) - cost(S^*)} \geq 0.63$.

⁴We will call it HRU algorithm.

Notice that this notion of performance is obtained by comparing the returned solution to the “worst” solution, while our performance factor is relative to the “best” solution. This remark has already been done in [10] where it is shown that maximizing gain does not mean necessary optimizing query response time. Indeed [9] provides no proof for the performance (as we have defined it) of their algorithm i.e \mathcal{S} is at most $f * Cost$ where f is a constant and $Cost$ is the cost of the optimal solution. Another weakness of [9] is its time complexity. Indeed, its complexity is $O(k * n^2)$ where k is the number of iterations (corresponds to the number of selected views) and n is the total number of cuboids. Since $n = 2^D$, so this algorithm is of little help when D is large. To overcome this problem, [15] proposed a simplification of the former algorithm and called it PBS (Pick By Size). This algorithm simply picks the cuboids in size ascending order until there is no enough memory left. They show that even its simplicity and its small complexity (linear), the solutions returned by PBS competes those of [9] in terms of gain. However, the query cost of its solutions suffers from the same problem as that of [9] i.e no performance guarantee. Some authors have considered a more general problem setting e.g [8, 16]. They consider the possibility to store both cuboids and indexes when the memory space is limited. [8] used an extension of [9] while [16] proposed to use integer programming techniques to solve the optimization problem. They propose both exact and approximate modeling of the optimization problem. The approximate modeling allows reducing the number of constraints. They tested their methods by using industrial constraint solvers (Ilog CPLEX). We have not considered indexes in the present work.

As a final remark, it is worthwhile to note that all these methods suppose a prior knowledge of cuboids sizes. This information is considered as part of their input and it is either computed or estimated. This represents a real limitation of these works since when the number of dimensions is large, the number of cuboids growing exponentially, they become intractable. This is in contrast to our proposal. Indeed, the algorithm PickBorders we propose does not need this knowledge even if actually computes the size of a part of the set of cuboids. This, in our opinion, is the most important contribution of our proposition. We also may cite [11] as another proposal for selecting materialized views. They consider the situation where the number of dimensions is so large, that none of the previous propositions can work. Their proposition consists simply in storing cuboids of 3 or 4 dimensions. The other cuboids can be answered by using a careful data storage technique. They justify their choice by the fact that usually people do not ask queries requiring too many dimensions. However, no performance study has been conducted along that work apart some experiments showing the feasibility of their method.

2.5 Results

Our results are twofold:

- in Section 3, we provide PickBorders an algorithm running in $O(D2^D)$ time with a performance factor less than f for any $f \geq 1$. We first start with two simple algorithms used as building blocks to design PickBorders;

- in Section 4, we compare PickBorders to HRU and PBS on different datasets computing the different measures of performance. In this section, we show that PickBorders performs almost as well as HRU in terms of cost and memory and it is much faster. PBS behaves poorly for these measures. PickBorders also gives the better performance factor.

3. PICKBORDERS

in this paper we address the following problem: Let $f \geq 1$ a real number given by the user. We want to find a set S that provides a cost no more than $MinCost * f$. Clearly, if $S = \mathcal{C}$ then this solves the problem. However, this solution could be unrealistic because the size of \mathcal{C} is huge. Thus, we add a constraint to the problem: we want the *smallest* S such that $Cost(S) \leq MinCost * f$. Here, *smallest* is intended in size terms. The obvious solution to this problem consists simply in considering all subsets $\mathcal{S} \in 2^{\mathcal{C}}$, compute their respective costs, keep those \mathcal{S} satisfying $Cost(\mathcal{S}) \leq MinCost * f$ and then return \mathcal{S} whose size is the smallest. Of course this solution is unrealistic because of its complexity. It turns out that this problem is NP-complete [10]. Thus, an approximation is needed. Although the solution we propose guarantees a bound in query evaluation time, it is not guaranteed to be optimal in terms of memory size.

In this section we present some techniques for selecting a subset of views to be stored. For each technique, we analyze its complexity and study its performance guarantees. We give a first solution to this problem. Even if it is rather not inefficient, it is a building block for our algorithm called PickBorders. So, its presentation makes PickBorders clearer. It consists simply in storing all cuboids whose size is less than the size of the base cuboid out of the factor f .

3.1 Algorithm PSC (Pick Small Cuboids)

Given f , a cuboid is called *small* if its size is less than M/f . Recall that $M = size(c_b)$. Keeping small cuboids for materialization guarantees a certain quality when querying the datacube.

LEMMA 1. *Let $f \geq 1$, $M = size(c_b)$ and $\mathcal{S} = \{c \in \mathcal{C} : size(c) \leq M/f\} \cup \{c_b\}$ then $cost(\mathcal{S}) \leq MinCost * f$.*

PROOF. Let $\mathcal{S}_1 = \{c \in \mathcal{C} : size(c) \leq M/f\}$ and $\mathcal{S}_2 = \mathcal{C} \setminus \mathcal{S}_1$. $MinCost = \sum_{c \in \mathcal{S}_1} size(c) + \sum_{c \in \mathcal{S}_2} size(c)$. On the other hand, $cost(\mathcal{S}) = cost(\mathcal{S}_1) + cost(\mathcal{S}_2) = \sum_{c \in \mathcal{S}_1} size(c) + \sum_{c \in \mathcal{S}_2} M$. Every $c \in \mathcal{S}_2$ will be computed from c_b with a cost equal to M . Since the size of each $c \in \mathcal{S}_2$ is greater than M/f then $\sum_{c \in \mathcal{S}_2} M \leq \sum_{c \in \mathcal{S}_2} size(c) * f$. Hence, $cost(\mathcal{S}) \leq MinCost * f$. \square

EXAMPLE 2. *Let us consider the datacube depicted in figure 3. It is the same datacube as that of Figure 1. We have just omitted the edges between nodes to make it more understandable. All the cuboids below the top most curve are the cuboids whose sizes are less than $size(ABCDE)/10$. So if the user wants the evaluation time of his/her queries not more than 10 times the minimal cost, then the algorithm will return these cuboids.*

We may consider two situations here, either we already know the size of cuboids or this information is unknown. In the former case, computing \mathcal{S} is quite simple. Indeed, it suffices to sort them and then pick those whose size is less than M/f . In the later case, the naive solution consists in computing the size of all cuboids, sort them and then keep those satisfying the condition. Notice however that the problem of computing \mathcal{S} in this context can be related to extracting frequent sets from transaction databases [13, 1]. Indeed, the condition $size(c) \leq M/f$ is anti-monotonic, that's $c \leq c'$ and $size(c) > M/f$ then $size(c') > M/f$. We thus can use level wise like algorithms such as A priori in order to prune cuboids whose size is greater than M/f . The algorithm is described below:

Algorithm PSC
Input: Parameter f , fact table T
Output: a partial datacube \mathcal{S}
$\mathcal{S} = \{\emptyset\}$ $\mathcal{C}_1 = \{c \in DC \text{ s.t } Dim(c) = 1\}$ $\mathcal{L}_1 = \{c \in \mathcal{C}_1 \text{ s.t } c \leq M/f\}$ $\mathcal{S} = \mathcal{S} \cup \mathcal{L}_1$ for ($i = 1; \mathcal{L}_i \neq \emptyset; i++$) do $\mathcal{C}_{i+1} = \{ \text{candidates generated from } \mathcal{L}_i \}$ for each $c \in \mathcal{C}_{i+1}$ do compute_size(c) $\mathcal{L}_{i+1} = \{c \in \mathcal{C}_{i+1} \text{ s.t } c \leq M/f\}$ $\mathcal{S} = \mathcal{S} \cup \mathcal{L}_{i+1}$ endfor Return \mathcal{S}

This is exactly A priori algorithm. The procedure *compute_size* could be implemented such that the actual size of the cuboid argument is calculated or it could use estimating size techniques such those discussed in [2]. This second solution may be preferred when we want to reduce computation time. The maximal complexity of this algorithm is 2^D . Indeed, if $f = 1$ then all cuboids have a size less than M/f thus all of them are computed. Of course, in practice $f > 1$ and the actual complexity is much less than 2^D .

Even if \mathcal{S} achieves a certain quality, it may be the case that its size is still too large and thus could not be stored entirely. Thus we want to reduce it. In the next section, we present another algorithm that selects a partial datacube. Just like the previous one, we will see that it does not solve our initial problem which consists in finding the smallest set of cuboids, in terms of memory space, to store while ensuring query evaluation time performance.

3.2 Algorithm PTB (Pick The Border)

We first distinguish between two sets of cuboids with respect to f , those for which we can reduce the maximal cost of computing them by a factor f and those for which we cannot. For example, for the base cuboid c_b , we cannot reduce its computation cost. It turns out that if we store only *maximal* cuboids w.r.t f than we reduce the maximal cost of cuboids of all those for which this reduction is possible. Let us first give some definitions.

DEFINITION 3 (f -REDUCIBLE CUBOID). *Let $c \in \mathcal{C}$, \mathcal{C}_c be the ancestors of c in \mathcal{C} , $M = size(c_b)$ and $f \geq 1$. c is*

*called f -reducible iff there exists $c' \in \mathcal{C}_c$ such that $size(c') \leq f * M/f$ and $c' \neq c$.*

In other words, c is *f-reducible* if it can be computed from a cuboid whose size is less than the maximal cuboid divided by the factor f .

EXAMPLE 3. *Let us continue with Figure 3. The cuboid $ABDE$ is not 10-reducible because none of its ancestors has a size less than $size(ABCDE)/10$. The cuboid B is 10-reducible because at least one of its ancestors has a size less than $size(ABCDE)/10$. Indeed, BC is an ancestor of B and its size is 40.*

Now we define the *f-reducible sets*.

DEFINITION 4 (f -REDUCIBLE SET OF CUBOIDS). *Let $f \geq 1$ and $\mathcal{S} \subseteq \mathcal{C}$. Then \mathcal{S} is an f -reducible set iff for all $c \in \mathcal{S}$, c is f -reducible.*

EXAMPLE 4. *Let us continue with Figure 3. The set $\{A, B, D, AB, BC, BD, CD, DE\}$ is 10-reducible since each of its elements has a least an ancestor whose size is less than $2000/10$.*

So for each $f \geq 1$ one can define the *maximal f-reducible set*. It is the maximal subset \mathcal{S} of \mathcal{C} such that \mathcal{S} is *f-reducible*. Now given $f \geq 1$, find \mathcal{S} such that for each *f-reducible* cuboid c , $cost(c, \mathcal{S}) \leq M/f$. We first show that the maximal *f-reducible set* fulfills this condition.

LEMMA 2. *Let $f \geq 1$. The maximal f -reducible set of 2^C is $\mathcal{S} = \{c \in \mathcal{C} : size(c) \leq M/f\}$.*

Let us now define the concept of maximal cuboid w.r.t to a set \mathcal{S} .

DEFINITION 5 (MAXIMAL CUBOID). *Let $f \geq 1$ and $\mathcal{S} \subseteq \mathcal{C}$. $c \in \mathcal{C}$ is maximal if there is no $c' \in \mathcal{S}$ such that $c \preceq c'$. Let us denote by $\mathcal{B}(\mathcal{S}) = \{c \in \mathcal{S} | c \text{ is maximal}\}$ ⁵.*

That is to say c is maximal in \mathcal{S} if it has no parent in \mathcal{S} . The border of \mathcal{S} is the set of its maximal cuboids. The following lemma shows that if we keep only the border of the *maximal f-reducible set* then the cost of the *f-reducible* cuboids is effectively reduced.

LEMMA 3. *Let $f \geq 1$ and \mathcal{S} be the maximal f -reducible set. Then for all $c \in \mathcal{S}$, $cost(c, \mathcal{B}(\mathcal{S})) \leq M/f$.*

PROOF. Each $c \in \mathcal{S}$ will be computed from an element $c' \in \mathcal{B}(\mathcal{S})$. Since $size(c') \leq M/f$ then $cost(c, \mathcal{S}) \leq M/f$. \square

⁵We use \mathcal{B} to denote what is called *positive border* in [12] and noted \mathcal{B}^+ there.

EXAMPLE 5. Let us continue with the datacube in figure 3. The border corresponding to $f = 10$ is the set $\mathcal{S} = \{ACD, ADE, BCD, BDE, CE\}$. This set together with the set of cuboids below this border is the maximal 10-reducible set.

Here below is a possible implementation of the algorithm PTB. It is an adaptation of the previous algorithm.

Algorithm PTB
Input: Parameter f , fact table T
Output: a partial datacube \mathcal{S}
$\mathcal{S} = \emptyset$ $\mathcal{C}_1 = \{c \in DC \text{ s.t. } Dim(c) = 1\}$ $\mathcal{L}_1 = \{c \in \mathcal{C}_1 \text{ s.t. } size(c) \leq M/f\}$ $\mathcal{S} = \mathcal{S} \cup \mathcal{L}_1$ for ($i = 1; \mathcal{L}_i \neq \emptyset; i++$) do $\mathcal{C}_{i+1} = \{ \text{candidates generated from } \mathcal{L}_i \}$ for each $c \in \mathcal{C}_{i+1}$ do compute_size(c) $\mathcal{L}_{i+1} = \{c \in \mathcal{C}_{i+1} \text{ s.t. } size(c) \leq M/f\}$ for each $c \in \mathcal{L}_i$ do Let $E = \{c' \in \mathcal{L}_{i+1} \text{ s.t. } c \leq c'\}$ If $E = \emptyset$ Then $\mathcal{S} = \mathcal{S} \cup \{c\}$ endfor Return \mathcal{S}

The complexity of this algorithm is $O(D * 2^D)$. Indeed, at most, the number of iterations is D . In this case, each of the 2^D cuboids is tested whether it is in the border or not. Since, the maximum number of parents of each cuboid is D , then the maximal number of tests is $D * 2^D$.

The implementation described above is probably not the best one. Indeed, since extracting borders from datacubes is equivalent to extracting maximal frequent sets from transaction databases, one can use, without much modifications, state of the art algorithms such as those of [5, 14, 12, 7, 3, 4].

Note the difference between the two former methods of how f is used. In the first case, it is intended to be the "lowest" factor by which we augment the minimal cost while in the second case, it is meant as the "largest" factor by which the maximal cost is divided. Even if this solution seems attractive since it is less space consuming than the first one, we have no guaranty respectively to the minimal cost and thus does not solve our initial problem. In the next section, we present our algorithm PickBorders whose returned solution, while guaranteeing the quality factor, has a reasonable memory size.

3.3 Algorithm PickBorders

Here, we present an algorithm that reduces the size of the solution returned by the first algorithm while guaranteeing the fact that the cost still be below $MinCost * f$. The idea consists simply in keeping the first border (i.e the solution of Algorithm PTB) together with the borders w.r.t $f^2, f^3, f^4 \dots$ i.e solutions of Algorithm PTB.

The obvious implementation of PickBorders consists in iterating the algorithm PTB for all the possible values of f .

Actually, the possible values of f are those f^i where i is an integer ranging from 0 to $\lfloor \log_f(M) \rfloor$. Thus a first implementation would be

$\mathcal{S} = \emptyset$ for ($i = 1; i \leq \lfloor \log_f(M) \rfloor; i++$) do $\mathcal{S} = \mathcal{S} \cup PTB(f)$ $f = f^i$ endfor

As discussed in the previous section, the complexity of $PTB(f)$ is in $O(D * 2^D)$. Since the loop **For** is executed $\log(M)$ times, then we conclude that the complexity of this algorithm is $O(\log(M) * D * 2^D)$.

Notice that by adopting this implementation, one can use whatever maximal frequent set algorithm among those proposed in the literature. Indeed, the procedure $PTB(f)$ is equivalent to those algorithms.

We propose another implementation. It consists in using PSC algorithm in order to label each visited cuboid c by a number k whenever $M/f^{k+1} \leq size(c) \leq M/f^k$. Now, it becomes simple to test whether a cuboid belongs to a border or not. It suffices to compare its label to the labels of its parents. If these are different, then c belongs to the set of borders. The concrete implementation could be described as follows.

Algorithm PickBorders
Input: Parameter f , fact table T
Output: a partial datacube \mathcal{S}
$\mathcal{S} = \emptyset$ $\mathcal{C}_1 = \{c \in DC \text{ s.t. } Dim(c) = 1\}$ $\mathcal{L}_1 = \{c \in \mathcal{C}_1 \text{ s.t. } size(c) \leq M/f\}$ for ($i = 1; \mathcal{L}_i \neq \emptyset; i++$) do $\mathcal{C}_{i+1} = \{ \text{candidates generated from } \mathcal{L}_i \}$ for each $c \in \mathcal{C}_{i+1}$ do compute_size(c) Let k such that $M/f^{k+1} \leq size(c) \leq M/f^k$ $Label(c) = k$ for each $c \in \mathcal{L}_i$ do Let $E = \{c' \in \mathcal{C}_{i+1} \text{ st } c \leq c'\}$ If for all $c' \in E$ $Label(c) \neq Label(c')$ Then $\mathcal{S} = \mathcal{S} \cup \{c\}$ $\mathcal{L}_{i+1} = \{c \in \mathcal{C}_{i+1} \text{ s.t. } size(c) \leq M/f\}$ endfor

PickBorders guarantees two nice properties. First, the cost of its output \mathcal{S} is bounded by the minimal cost times the factor f . This property may be qualified as a global property. The second one, is that the same output guarantees that if we take each cuboid individually, its cost with respect to \mathcal{S} is also bounded by the minimal cost times f . To the best of our knowledge, none of the solutions for partial datacube selection proposed so far can provide these guarantees. This result is stated in the following theorem.

THEOREM 1. Let $f \geq 1$ and $\mathcal{S}_i = \{c \in \mathcal{C} : |c| \leq M/f^i\}$ for $i = 1 \dots \lfloor \log_f(M) \rfloor$. Let $\mathcal{B}_i(\mathcal{S}) = \{c \in \mathcal{S}_i \text{ s.t. } c \text{ is maximal}\}$. Let $\mathcal{S} = \cup_{i=1 \dots \lfloor \log_f(M) \rfloor} \mathcal{B}_i$. Then

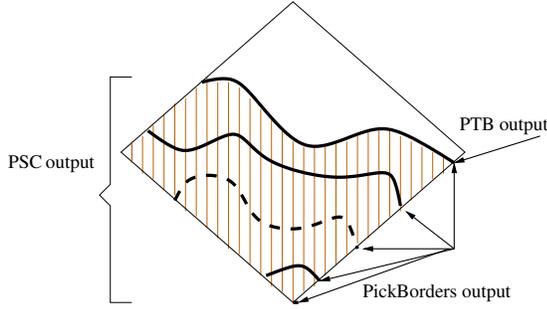


Figure 2: The three solutions.

1. $cost(\mathcal{S}) \leq MinCost * f$
2. For all $c \in \mathcal{C}$, $cost(c, \mathcal{S}) \leq size(c) * f$.
3. \mathcal{C} can be computed is $O(D * 2^D)$

PROOF. It is simple to note that for each cuboid c in \mathcal{C} , there exists an ancestor c' of c in \mathcal{S} such that $size(c') \leq f * size(c)$. So, for computing each cuboid, we need a cost at most equal to the size of the cuboid (i.e minimal cost) times the factor f . This proves item 2 of the theorem. The first item is a direct consequence of the second. Finally, the number of iterations of algorithm PickBorders is D . In this case, the number of cuboids for which we test whether they belong to a border or not is 2^D . The maximal number of parents of a cuboid is D . This gives the maximal number of tests which is bounded by $D * 2^D$. \square

An important corollary of this theorem is that the solution returned by PickBorders algorithm involves a cost less than the optimal solution times the factor f . Let us first define what is an optimal solution.

DEFINITION 6 (OPTIMAL SOLUTION). Let Mem denote a storage space amount. Let $\mathcal{S} \subseteq \mathcal{C}$. \mathcal{S} is a possible partial datacube iff (1) $size(\mathcal{S}) \leq Mem$ and (2) $cost(\mathcal{S}) \neq \infty$. The set of possible partial datacubes w.r.t Mem is denoted $Pos(Mem)$. \mathcal{S}^* is optimal w.r.t Mem iff (1) $\mathcal{S}^* \in Pos(Mem)$ and (2) $cost(\mathcal{S}^*) = \min_{\mathcal{S} \in Pos(Mem)} cost(\mathcal{S})$.

COROLLARY 1. Let $f > 1$. Let \mathcal{S} be the solution of PickBorders algorithm. Let $Mem = size(\mathcal{S})$. Let \mathcal{S}^* be the optimal partial datacube w.r.t Mem . Then $cost(\mathcal{S}) \leq f * cost(\mathcal{S}^*)$.

In other words, this says that if we consider among the sets of cuboids whose total size is less than the size of \mathcal{S} , the set \mathcal{S}^* that provides the lowest cost, then we guarantee that the cost of \mathcal{S} is at most f times the cost of \mathcal{S}^* .

Figure 2 summarizes the three algorithms. It shows the datacube lattice. The dashed area represents the solution returned by Algorithm PSC. The top most thick curve is

the border obtained by considering f , i.e the solution of algorithm PTB. The set of all curves represents the borders relative to $f^0, f^1, f^2, f^3 \dots \lfloor \log_f(M) \rfloor$ i.e the output of PickBorders.

Before illustrating our proposal, let us first make some remarks.

REMARK 1. Most proposals for selecting the views to be materialized aim at reducing the total cost of answering queries. They thus provide an optimal (on an approximation of) in this sens (see Definition 6.). It is worthwhile noting that this notion of optimality is a global property. Indeed, it does offer no guarantee about query time evaluation for individual cuboids, i.e it may be the case that for some cuboid c , $cost(c, \mathcal{S})/size(c)$ is very large. Moreover, the query response time may be not proportional to query result size. So even when it is possible to compute the optimal solution, and this can be achieved only when we have few dimensions, see e.g [16], the user may be surprised to find out that the time required for getting a response of n rows is more than that of getting m rows while $n \ll m$.

EXAMPLE 6. To illustrate our proposition, let us continue with Figure 3. The curves represent the borders relative to different powers of the factor f when $f = 10$. The filled circles represent elements of at least one border. These cuboids are the only ones to be stored. The total size of the datacube is 8928 which also represents the minimal cost of computing all cuboids. The maximal cost is $32 * 2000 = 64000$. By keeping only the elements of the borders, this will occupy a memory whose size is 2607 and the total cost is 27554. One should notice here that even if we have fixed $f = 10$, the cost ratio between the cost of PickBorders solution and MinCost is $\frac{27554}{8927} = 3.09$. This means that in average, query response time w.r.t \mathcal{S} is 3 times the minimal cost. If we execute the HRU algorithm of [9] with a criterion of maximal available space equal to 2607, then ABCDE and BCDE are the only cuboids that are returned. Note that with this solution, the total cost is 41600. Thus, the cost ratio is $\frac{41600}{8927} = 4.67$. This is worse than the performance of PickBorders. In another hand, if we execute PBS of [15], 16 cuboids will be stored (the sixteen first cuboids ordered by their respective size). In this case, the total cost is 34518. The cost ratio now is $\frac{34518}{8927} = 3.87$.

4. EXPERIMENTS

An experimental validation is given in this section. We consider the following data sets:

- USData10: contains data with 2.5 millions of tuples with 10 attributes corresponding to the eleven first attributes (excluding the first one representing a rowid) of USData set. This dataset is US Census 1990 data available from <http://kdd.ics.uci.edu/>
- USData13: it is the same dataset as USData10 but with 13 attributes (adding the next three attributes to USData10).
- Objects: contains data with (only) 8000 tuples with 12 attributes dealing with objects found in archaeological

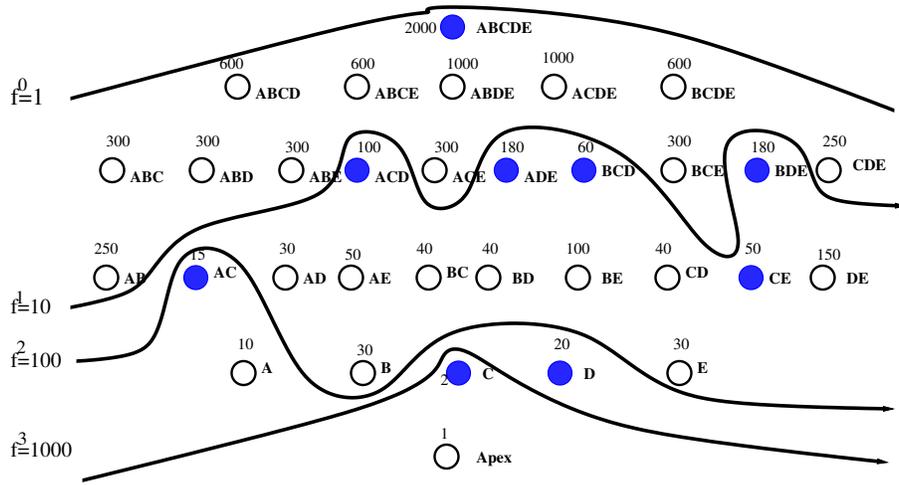


Figure 3: PickBorders: the borders w.r.t $f = 10$

mining. This example represents a case for which the number of attributes is relatively large with respect to the size of the dataset.

- Zipf10: this dataset is synthetic. It contains 10^6 rows and 10 dimensions. Many observations showed that the attributes values of real datasets do not follow - in general - a uniform distribution but often a power law distribution. That is, if we sort the values of a given attribute in the decreasing order, then the frequency of the value of rank i is proportional to $\frac{1}{i^\alpha}$, $\alpha > 0$. α belongs mostly to the range [2, 3]. In our experiments, we have considered a power law of parameter $\alpha = 2$.

Table 1 summarizes some characteristics of these data sets. It sums up MinCost (whenever the whole datacube is stored) and MaxCost (whenever only the base cuboid is stored).

Dataset	MinCost	MaxCost
USdata10	$4.37 * 10^6$	$5.35 * 10^7$
USdata13	$1.05 * 10^8$	$1.19 * 10^9$
Objects	$1.72 * 10^7$	$3.05 * 10^7$
ZIPF10	$4 * 10^7$	$3.93 * 10^8$

Table 1: MinCost and MaxCost of Datasets

4.1 Related algorithms

To the best of our knowledge, there is no algorithm proposed so far that solves the problem we have treated in this paper. Nevertheless, in order to validate experimentally our approach, we made a comparison with two well-known algorithms namely PBS (Pick By Size) [15] and HRU [9]. In fact, as mentioned earlier, these algorithms do not address exactly the same problem as us. Indeed, their constraint is space bound while ours is performance factor. So to make the comparison valid, in each experiment, we have first run PickBorders w.r.t a performance factor f then picked the size of the solution and considered it as the space constraint for the other two algorithms. Both HRU and PBS try to find the best subset \mathcal{S} of cuboids that maximizes the *gain* criterion (see Definition 2 in section 2.4 for the concept of

gain). In fact, they return an approximation. Let us now recall the principles of HRU and PBS here below.

PBS Algorithm

As described before in section 2.4, PBS simply keep the cuboids in their ascending size until there is no memory space left. So whenever the size of the cuboids is already known, the only difficulty of this algorithm is to sort the cuboids w.r.t their size.

Algorithm PBS

Input: space , the amount of available memory

Input: \mathcal{C} the datacube cuboids

```

 $\mathcal{S} = \{c_b\}$ 
 $\text{space} = \text{space} - \text{size}(b_c)$ 
While ( $\text{space} > 0$ ) DO
   $c = \text{smallest cuboid in } (\mathcal{C})$ 
  If ( $\text{space} - \text{size}(c) < 0$ ) THEN
     $\text{space} = \text{space} - \text{size}(c)$ 
     $\mathcal{S} = \mathcal{S} \cup \{c\}$ 
     $\mathcal{C} = \mathcal{C} \setminus \{c\}$ 
  Else
     $\text{space} = 0$ 
Return  $\mathcal{S}$ 

```

In fact, PBS and PSC Algorithm presented in section 3.1 are quite similar. The only difference is that in PSC, the input is a parameter f , implying a corresponding amount of memory.

HRU Algorithm

Roughly speaking, HRU chooses step by step the next cuboid c to store by keeping the one that maximizes the benefit whenever it is added to \mathcal{S} . This process is repeated until there is no more available storage space.

Algorithm HRU
Input: space, the amount of memory available
Input: \mathcal{C} the set of cuboids of the datacube
$\mathcal{S} = \{c_b\}$
space = space - size(b_c)
While (space > 0) DO
$c =$ cuboid not in \mathcal{S} such that $b(c, \mathcal{S})$ is maximized.
/* $b(c, \mathcal{S})$ is the benefit obtained by adding c to \mathcal{S} */
If (space - size(c) > 0) Then
space = space - size(c)
$\mathcal{S} = \mathcal{S} \cup \{c\}$
$\mathcal{C} = \mathcal{C} \setminus \{c\}$
Else
space = 0
Return \mathcal{S}

The main drawback of this algorithm is the time to compute the cuboid that maximizes the benefit. Indeed, at each iteration we need to compute the benefit of $\Theta(n^2)$ cuboids with $n = 2^D$. Clearly, this is infeasible when D is large. It is essentially to overcome this problem that PBS has been proposed.

Presented as above, both PBS and HRU takes as input a constraint on the available space of memory. In our experiments, we removed this constraint in order to compare them to PickBorders for any amount of memory. For a similar reason, PickBorders is not presented with a constraint of memory. However, it is easy to take it into account. Indeed, it suffices to run PickBorders with any small value of parameter f , say 1.2 for instance. If the size of the returned solution is too large to fit in the available memory then we rerun PickBorders with parameter f^2, f^3 until we get a partial datacube \mathcal{S} that can be stored w.r.t the available memory. In fact, it is not necessary to run several times PickBorders with different parameters. It suffices to note that PickBorders with parameter f^{i+1} has an output included in the PickBorders with parameter f^i . So, PickBorders(f^{i+1}) has already been computed in PickBorders(f^i).

The three algorithms (HRU, PBS and PickBorders) take as input the list of cuboids and their size. To pre-compute the size of the cuboids, we can use the code given in [2]. For PBS and HRU, the output is a list of cuboids. These lists represent the order by which the cuboids are chosen in order to be stored.

The curves of the next figures show the amount of memory and the corresponding cost whenever the k first cuboids of the list are materialized with k varying from 1 (only the base cuboid) to 2^D for a D -dimensional fact table.

Graphics of Section 4.2 give a general trend of the performance of the three algorithms. In Section 4.3, we take a closer look at the output of the three algorithms showing the behavior of the performance factor.

4.2 Cost and memory

In our experiments, since we make no assumption on the way the views are physically stored, the amount of memory is expressed as the number of rows of the materialized views set. For PickBorders, we run the algorithm taking

$f = 1.5, f^2 = 2.25, f^3 = 3.38, \dots$ for all datasets. Each execution leads to a pair Memory/Cost. We then used this Memory value as the space limit parameter for both HRU and PBS. Again, each time we obtain a pair Memory/Cost. This experiment is depicted in Figures 4, 5, 6 and 7. For instance, for USData10, when $f = 1.5$, PickBorders needs 2160000 units of memory for a cost of 4750000 whereas for $f = 3.38$, PickBorders takes 828000 units of memory and has a cost of 7100000.

In all experiments, PBS has the worst performance in terms of cost and memory. In general, HRU has the best performance but PickBorders is a very good challenger. We can also remark that PickBorders is very competitive whenever the amount of available memory is not too low (for $M > 10^5$ in USData10, for $M > 10^6$ in Objects and for $M > 15.10^6$ in ZIPF10).

Due to time required to run HRU, we stopped computation before adding all cuboids to \mathcal{S} as soon as the cost function is close to MinCost.

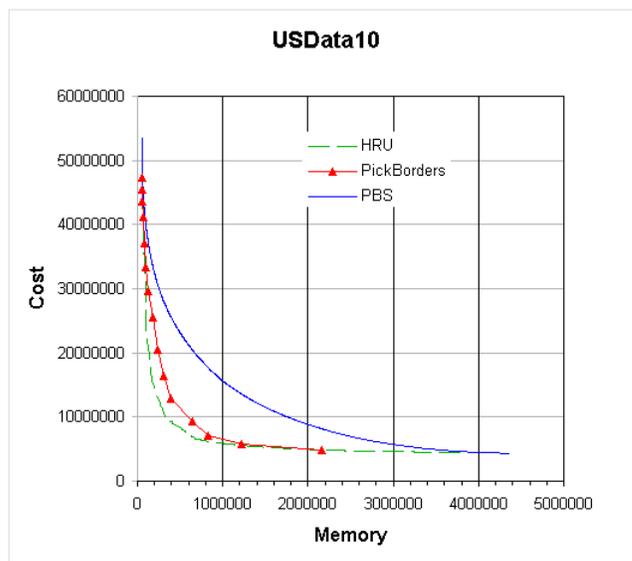


Figure 4: Cost/Memory for USData10.

4.3 Performance factor

Now we compare our approach to HRU and PBS with respect to their query evaluation performances. One of the main interest of PickBorders is the guarantee of the performance factor. Despite the fact that neither PBS nor HRU provides guarantee on this measure, it is interesting to measure the value of the performance factor in a practical setting.

For this experiment, we have executed PickBorders with some values of f . Each time we got a solution \mathcal{S} then we executed HRU. Figure 8 shows that PickBorders has a better average performance factor than PBS and HRU for USData10 (for $f = 3.38$ and $f = 11.39$).

A more careful look at the distribution of the performance factors in Figures 9 and 10 explains this fact by showing that

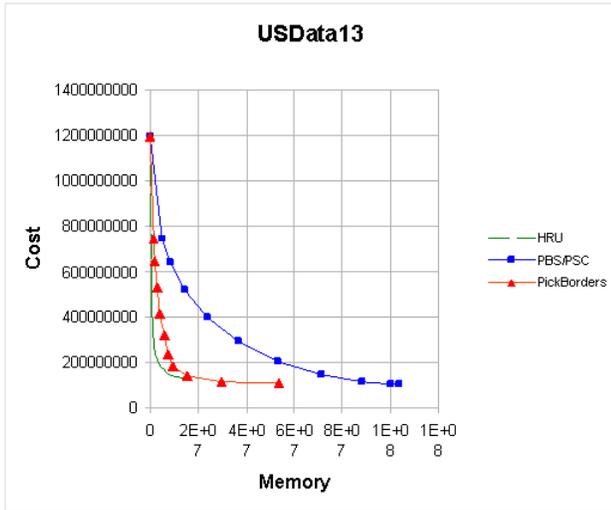


Figure 5: Cost/Memory for USData13.

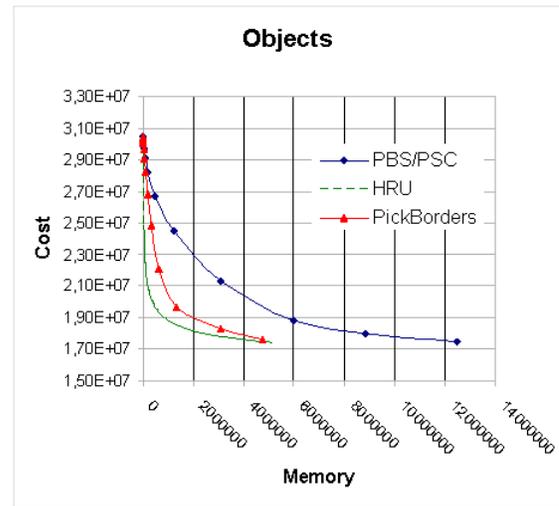


Figure 7: Cost/Memory for Objects.

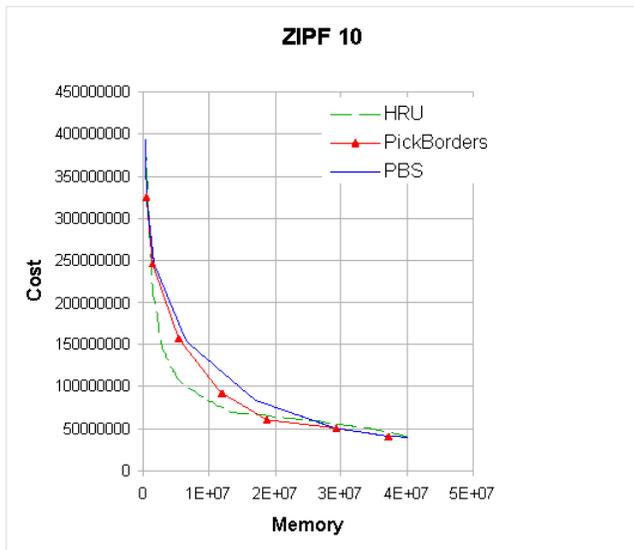


Figure 6: Cost/Memory for ZIPF10.

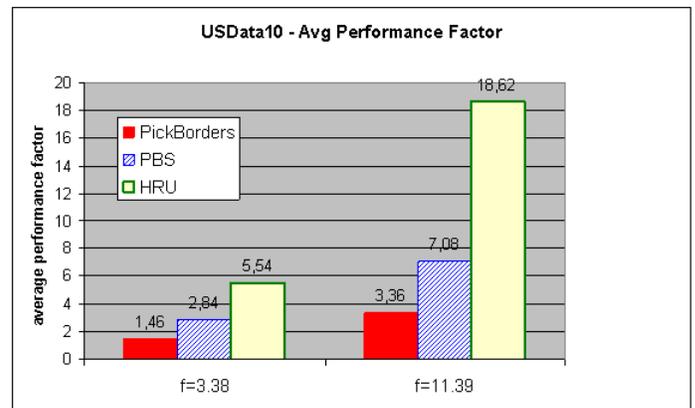


Figure 8: Performance factor with $f = 3.38$ and $f = 11.39$.

some views should be computed from materialized views whose size is very large. For instance, running HRU with 830000 units of memory, there are 7 (not materialized) cuboids whose least materialized ancestor is of size at least 100 times larger. We ran PBS and HRU with 309000 and 830000 units of memory limit corresponding respectively to the amount of memory used by PickBorders with $f = 3.38$ and $f = 11.39$. The first category represents the set of cuboids with performance factor 1 (materialized views and views whose least materialized ancestor have the same size). Second category counts the cuboids with a performance factor in [1, 2]. The third category counts the cuboids with a performance factor in [2, 3.5] and so on. Note that the spirit of each algorithm is sketched by the first category: PBS stores many small cuboids, HRU tends to materialize few large cuboids and PickBorders chooses a combination of cuboids of different sizes.

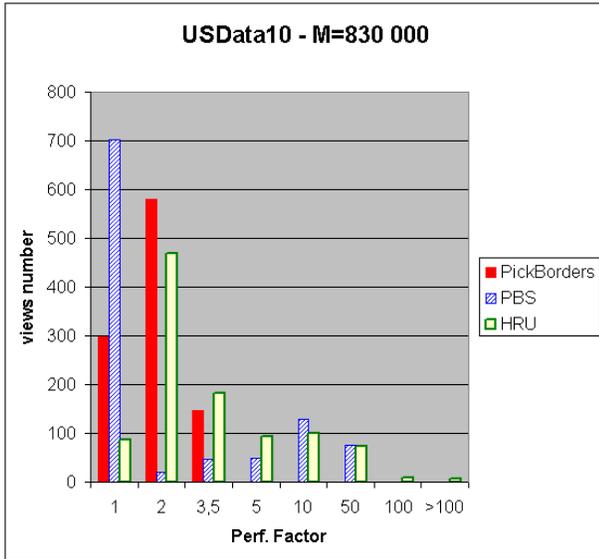


Figure 9: Performance factor distribution with $f = 3.38$.

We terminate this section by noting that the experiments we have presented in this paper aimed to show the quality of the solutions returned by PickBorders compared to other algorithms. We have not made comparisons in terms of execution time. However, we may say that in our experiments, PickBorders and PBS often took few seconds whereas HRU required hours. We leave the results of execution time analysis to an extended version of this work.

5. CONCLUSION

We have shown in this study that PickBorders is a view selection algorithm that is as quick as PBS and whose cost measure is close to the one of HRU with a comparable amount of memory to store the views. We also introduced the performance factor measure and showed that PickBorders outperforms the two other algorithms in terms of performance factor average, for different datasets.

In the experiments we have conducted, we assumed that views sizes are known. It is of course not very realistic since it takes a long time to compute or approximate this piece

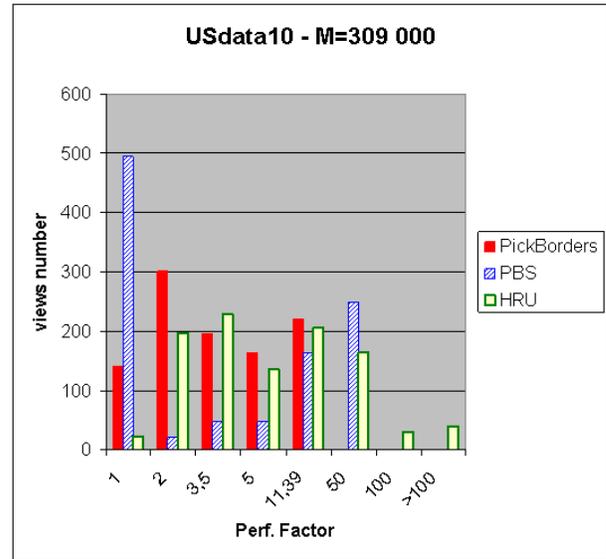


Figure 10: Performance factor distribution with $f = 11.39$.

of information. This is because the comparisons we have made are based just in the quality of the algorithms outputs. We have not compared them in terms of execution time complexity. For instance, using the code given in [2], it takes few seconds to compute a view size for a fact table of 2 millions of elements. As soon as the number of attributes approaches the value 10, it can take hours to pre-compute the views size before being able to run HRU. Moreover, if we have dozens of attributes, the amount of memory to store every view size becomes huge. It turns out that HRU cannot be used for d -dimensional fact table with d large. The next step consists in minimizing the number of calls to the computation of a view size.

Another direction for future research is an extension of PickBorders by taking into account a query workload (statistics on the frequency of queries of each cuboid) whenever it is known. Indeed, in the present work we assumed that the user may ask every possible query and all these queries are of the same probability. It may appear after a certain time of the datacube usage that only few queries are asked. Then it may be more interesting to try to optimize these interesting queries. Another direction consists in considering the possibility to store additional data with cuboids e.g indexes. In this latter case, the search space is enlarged. This what has been done in works like [16].

6. ACKNOWLEDGMENTS

We would like to thank the anonymous referees for their insightful comments.

7. REFERENCES

- [1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB Proceedings*, 1994.
- [2] K. Aouiche and D. Lemire. A comparison of five probabilistic view-size estimation techniques in olap.

- In *DOLAP Proceedings*, 2007.
- [3] R. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD Proceedings*, 1998.
 - [4] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE Proceedings*, 2001.
 - [5] K. Gouda and M. J. Zaki. GenMax: An efficient algorithm for mining maximal frequent itemsets. *Data Mining and Knowledge Discovery*, 11(3):223–242, 2005.
 - [6] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.
 - [7] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. S. Sharm. Discovering all most specific sentences. *ACM Transactions on Database Systems*, 28(2):140–174, 2003.
 - [8] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for olap. In *ICDE Proceedings*, 1997.
 - [9] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD Proceedings*, 1996.
 - [10] H. J. Karloff and M. Mihail. On the complexity of the view-selection problem. In *PODS Proceedings*, 1999.
 - [11] X. Li, J. Han, and H. Gonzalez. High-dimensional olap: A minimal cubing approach. In *VLDB Proceedings*, 2004.
 - [12] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.
 - [13] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *KDD Workshop*, 1994.
 - [14] K. Satoh and T. Uno. Enumerating maximal frequent sets using irredundant dualization. In *Discovery Science conference proceedings*, 2003.
 - [15] A. Shukla, P. M. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *VLDB Proceedings*, 1998.
 - [16] Z. A. Talebi, R. Chirkova, Y. Fathi, and M. Stallmann. Exact and inexact methods for selecting views and indexes for olap performance improvement. In *EDBT Proceedings*, 2008.