Dynamic Workload-Based Partitioning Algorithms for Continuously Growing Databases

Miguel Liroz-Gistau, Reza Akbarinia, Esther Pacitti, Fabio Porto, Patrick Valduriez

Dynamic Workload-Based Partitioning Algorithms for Continuously Growing Databases*

Miguel Liroz-Gistau[1], Reza Akbarinia[1], Esther Pacitti[2], Fabio Porto[3], and Patrick Valduriez[1]

[1] INRIA & LIRMM, Montpellier, France
{Miguel.Liroz_Gistau, Reza.Akbarinia, Patrick.Valduriez}@inria.fr
[2] University Montpellier 2, INRIA & LIRMM, Montpellier, France
Esther.Pacitti@lirmm.fr
[3] LNCC, Petropolis, Brazil
fporto@lncc.br

Abstract. Applications with very large databases, where data items are continuously appended, are becoming more and more common. Thus, the development of efficient data partitioning is one of the main requirements to yield good performance. In the case of applications that have complex access patterns, e.g. scientific applications, workload-based partitioning could be exploited. However, existing workload-based approaches, which work in a static way, cannot be applied to very large databases. In this paper, we propose *DynPart* and *DynPartGroup*, two dynamic partitioning algorithms for continuously growing databases. These algorithms efficiently adapt the data partitioning to the arrival of new data elements by taking into account the affinity of new data with queries and fragments. In contrast to existing static approaches, our approach offers constant execution time, no matter the size of the database, while obtaining very good partitioning efficiency. We validated our solution through experimentation over real-world data; the results show its effectiveness.

1 Introduction

We are witnessing the proliferation of applications that have to deal with huge amounts of data. The major software companies, such as Google, Amazon, Microsoft or Facebook have adapted their architectures in order to support the enormous quantity of information that they have to manage. Scientific applications are also struggling with those kinds of scenarios and significant research efforts are directed to deal with it [4]. An example of these applications is the management of astronomical catalogs; for instance those generated by the Dark Energy Survey (DES) [1] project with which we are collaborating. In this project, huge tables with billions of tuples and hundreds of attributes (corresponding to dimensions, mainly double precision real numbers) store the collected sky data.

* Work partially funded by the CNPq-INRIA HOSCAR project.

Data are appended to the catalog database as new observations are performed and the resulting database size is estimated to reach 100TB very soon. Scientists around the globe can access the database with queries that may contain a considerable number of attributes.

The volume of data that such applications hold poses important challenges for data management. In particular, efficient solutions are needed to partition and distribute the data in multiple servers, e.g., in a cluster. An efficient partitioning scheme would try to minimize the number of fragments that are accessed in the execution of a query, thus minimizing the overhead of the distributed execution. Vertical partitioning solutions, such as column-oriented databases [18], may be useful for physical design on each node, but fail to provide an efficient distributed partitioning, in particular for applications with high dimensional queries, where joins would have to be executed by transferring data between nodes. Traditional horizontal partitioning approaches, such as hashing or range-based partitioning, are unable to capture the complex access patterns present in scientific computing applications, especially because these applications usually make use of complicated relations, including mathematical operations, over a big set of columns, and are difficult to be predefined a priori.

One solution is to use partitioning techniques based on the workload. Graph-based partitioning is an effective approach for that purpose [8]. A graph (or hypergraph) that represents the relations between queries and data elements is built and the problem is reduced to that of minimum k-way cut problem, for which several libraries are available. However, this method requires to process the entire graph in order to obtain the partitioning. This strategy works well for static applications, but scenarios where new data are inserted to the database continuously, which is the most common case for scientific computing, introduce an important problem. Each time a new set of data is appended, the partitioning should be redone from scratch, and as the size of the database grows, the execution time of such operation may become prohibitive.

In this paper, we are interested in dynamic partitioning of large databases that grow continuously. After modeling the problem of data partitioning in dynamic datasets, we propose two dynamic workload-based algorithms, called *DynPart* and *DynPartGroup*, that efficiently adapt the partitioning to the arrival of new data elements. Our algorithms are designed based on a heuristic that we developed by taking into account the affinity of new data with queries and fragments. In contrast to the static workload-based algorithms, the execution time of our algorithms do not depend on the total size of the database, but only on that of the new data and this makes them appropriate for continuously growing databases.

We validated our solutions through experimentation over real-world data sets. The results show that they obtain high performance gains in terms of partitioning execution time compared to one of the most efficient static partitioning algorithms. We also compared both algorithms and concluded that the grouping strategy of *DynPartGroup* obtains better partitioning efficiencies and performs

better, specially in scenarios with high correlation between new data items and strict imbalance constraints.

This paper is a major extension of [12], which only presented the *DynPart* algorithm. Here, we propose a variation, *DynPartGroup*, which groups data items before calculating fragment affinities. This strategy adapts better for the situations where there is high correlation on the new data items and the imbalance constraints (maximum allowed imbalance) are strict, and offers an improved performance. We also extend the imbalance constraint by adding the possibility of considering the load imbalance between fragments in addition to the size imbalance. Moreover, we deal with data deletions and updates in addition to insertions. Finally, we include an extended set of experimental results for the new contributions.

The remainder of this paper is organized as follows. In Section 2, we describe our assumptions and define formally the problem we address. In Section 3, we propose our basic solution for dynamic data partitioning, that we extend in Section 4 by grouping similar data items. Section 5 reports on the results of our experimental validation. Section 6 discusses related work, and Section 7 concludes.

2 Problem Definition

In this section, we state the problem we are addressing and specify our assumptions. We start by defining the problem of static partitioning, and then extend it for a dynamic situation where the database can evolve over time.

2.1 Static Partitioning

The static partitioning is done over a set of *data items* and for a *workload*. Let $D = \{d_1, ..., d_n\}$ be the set of data items. The workload consists of a set of queries $W = \{q_1, ..., q_m\}$. We use $q(D) \subseteq D$ to denote the set of data items that a query q accesses when applied to the data set D. Given a data item $d \in D$, we say that it is *compatible* with a query q, denoted as $comp(q, d)$, if $d \in q(D)$. Queries are associated with a relative frequency $f : W \to [0, 1]$, such that $\sum_{q \in W} f(q) = 1$.

Partitioning of a data set is defined as follows.

Definition 1. *Partitioning of a data set D consists of dividing the data of D into a set of fragments, $\pi(D) = \{F_1, ..., F_p\}$, such that there is no intersection between the fragments, $\forall i \neq j : F_i \cap F_j = \emptyset$, and the union of all fragments is equal to D, i.e., $\bigcup_{i=1}^{p} F_i = D$.*

Let $q(F)$ denote the set of data items in fragment F that are compatible with q. Given a partitioning $\pi(D)$, the set of *relevant fragments* of a query q, denoted as $rel(q, \pi(D))$, is the set of fragments that contain some data accessed by q, i.e., $rel(q, \pi(D)) = \{F \in \pi(D) : q(F) \neq \emptyset\}$.

To avoid a high imbalance on the size of the fragments, we use an *imbalance factor*, denoted by ϵ_s. The size of the fragments at each time should satisfy the following condition: $|F| \leq \left\lceil \frac{|D|}{|\pi(D)|}(1 + \epsilon_s) \right\rceil$.

In this paper, we are interested in minimizing the number of query accesses to fragments. Note that the minimum number of relevant fragments of a query q is $minfr(q, \pi(D)) = \left\lceil \frac{|q(D)|}{(|D| \,/\, |\pi(D)|)(1+\epsilon_s)} \right\rceil$. We define the *efficiency of a partitioning* for a workload based on its efficiency for queries. Intuitively, the *efficiency of a partitioning for a query* represents the ratio between the minimum number of relevant fragments of q and the number of fragments that are actually accessed under the given partitioning:

Definition 2. *Given a query q, then the efficiency of a partitioning $\pi(D)$ for q, denoted as $eff(q, \pi(D))$ is computed as:*

$$eff(q, \pi(D)) = \frac{minfr(q, \pi(D))}{|rel(q, \pi(D))|} \tag{1}$$

When the number of accessed fragments is equal to the minimum possible, i.e., $minfr(q, \pi(D))$, the efficiency is 1.

Using $eff(q, \pi(D))$, we define the efficiency of a partitioning $\pi(D)$ for a workload W as follows.

Definition 3. *The efficiency of a partitioning $\pi(D)$ for a workload W, denoted as $eff(W, \pi(D))$, is equal to the sum of the efficiencies of partitioning $\pi(D)$ for all queries in W multiplied by their relative frequencies. In other words,*

$$eff(W, \pi(D)) = \sum_{q \in W} f(q) \times eff(q, \pi(D)) \tag{2}$$

Given a set of data items D and a workload W, the goal of static partitioning is to find a partitioning $\pi(D)$ such that $eff(W, \pi(D))$ is maximized.

2.2 Dynamic Partitioning

Let us assume now that the data set D grows over time. For a given time t, we denote the set of data items of D at t as $D(t)$[4].

During the application execution, there are some events, namely *data insertions*, by which new data items are inserted into D. These events in the model correspond to the appending of the tuples corresponding to new observations in the DES catalog. No changes in the schema are involved. Let $T_{\text{ev}} = (t_1, \ldots, t_m)$ be the sequence of time points corresponding to those events. Note that between two consecutive time points t_i, t_{i+1}, D remains constant. In this paper, we assume that the workload is stable and neither the queries nor their frequencies change. However, the queries may access new data items as the data set grows.

Let us now define the problem of dynamic partitioning as follows. Let $T_{\text{ev}} = (t_1, \ldots, t_m)$ be the sequence of time points corresponding to data insertion events; $D(t_1), \ldots, D(t_m)$ be the set of data items at $t_1, \ldots, t_m$ respectively; and

[4] We confine this formulation to this subsection for the sake of simplicity, so that, in the next sections, when we use D we mean $D(t_i)$.

W be a given workload. Note that, as we only consider data insertions, if $t_i < t_j$ then $D(t_i) \subset D(t_j)\ \forall t_i, t_j \in T_{ev}$.

The goal is to find a set of partitionings $\pi(D(t_1)), ..., \pi(D(t_m))$ for data sets $D(t_1), \ldots, D(t_m)$ respectively, such that the sum of the efficiencies of the partitionings for W are maximized. In other words, our objective is as follows:

Objective: Maximize $\left(\sum_{q \in W} (f(q) \times \mathit{eff}(q, \pi(D(t))))\right) \forall t \in T_{ev}$

3 Affinity Based Dynamic Partitioning

In this section, we propose an algorithm, called *DynPart*, that deals with dynamic partitioning of data sets. It is based on a principle that we developed using the partitioning efficiency measure described in the previous section.

3.1 System Overview

In this paper, our proposal mainly focuses on how the data is partitioned in fragments. Here, we provide an overview of a system architecture taking advantage of our partitioning approach. The components of this architecture are as following (see Figure 1):

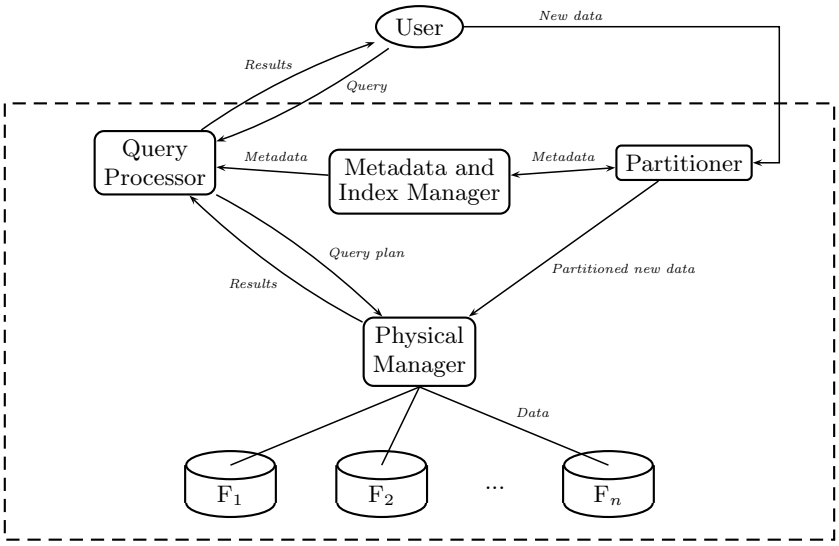


Fig. 1. System architecture

– **Query processor:** It parses the user queries, accesses the metadata and index manager, prepares an optimized execution plan and sends it to the physical manager to retrieve the data from fragments.

– **Metadata and Index Manager:** Stores metadata about the partitioning, and also indexes the location of the data items in the fragments.
– **Physical Manager:** It is in charge of storing/retrieving data to/from fragments.
– **Partitioner:** It holds the data items until a given number of items is inserted. Then, it obtains the necessary metadata and executes the partitioning algorithm. Finally, it transfers the data items to the corresponding fragments and informs the metadata and index manager about the modifications in the fragments. This component may also be contacted to include in the query results the corresponding data items in new added data.

We assume a shared nothing architecture composed of data nodes containing a physical data manager that stores one or several fragments at each node, and dedicated nodes for other components. We used a shared nothing architecture as it is the most common one since it is cheaper and can be scaled easily when required. The query processor and the metadata and index manager are preferred to be executed in the same node (nodes) to avoid communication overhead, as the query processor always has to access the index.

3.2 Principle

Let d be a new inserted data item. We can express the efficiency of the new dataset as:

$$\mathit{eff}(W, \pi(D \cup \{d\})) = \mathit{eff}(W, \pi(D)) + \Delta \tag{3}$$

Let assume that F is the fragment selected to insert d. The efficiency will remain the same for all queries but those which now have to access F in order to retrieve d but did not before. Hence, we can calculate Δ as[5]:

$$\Delta \approx \sum_{q:q(F)=\emptyset \wedge comp(q,d)} f(q)\left(\mathit{eff}(q, \pi(D \cup \{d\})) - \mathit{eff}(q, \pi(D))\right) \tag{4}$$

$$= \sum_{q:q(F)=\emptyset \wedge comp(q,d)} f(q)\left(\frac{\mathit{minfr}(q, \pi(D))}{|rel(q, \pi(D))| + 1} - \frac{\mathit{minfr}(q, \pi(D))}{|rel(q, \pi(D))|}\right) \tag{5}$$

$$= - \sum_{q:q(F)=\emptyset \wedge comp(q,d)} f(q)\frac{\mathit{minfr}(q, \pi(D))}{|rel(q, \pi(D))|\,(|rel(q, \pi(D))| + 1)} \tag{6}$$

where $q : q(F) = \emptyset \wedge comp(q, d)$ is the set of queries that will read d but no other data items in F.

Based on this idea, we define *the affinity between the data d and fragment F*:

$$\mathit{aff}(d, F) = - \sum_{q:q(F)=\emptyset \wedge comp(q,d)} f(q)\frac{\mathit{minfr}(q, \pi(D))}{|rel(q, \pi(D))|\,(|rel(q, \pi(D))| + 1)} \tag{7}$$

[5] Note that this approximation is an equality in all cases except when the increment in $|q(D)|$ makes $\mathit{minfr}(q, \pi(D))$ to be increased by 1, which happens very rarely.

Using (7), we develop a heuristic algorithm that places the new data items in the fragments based on the maximization of the affinity between the data items and the fragments.

3.3 Algorithm

Our *DynPart* algorithm takes a set of new data items D' as input and selects the best fragments to place them. For each new data item $d \in D'$, it proceeds as follows (see the pseudo-code in Algorithm 1). First, it finds the set of queries that are compatible with the data item. This can be done by executing the queries of W on D' or by comparing their predicates with every new data item. Then, for each compatible query q, *DynPart* finds the relevant fragments of q, and increases the fragments affinity by using the expression in (7). Initially the affinity of fragments is set to zero.

Algorithm 1 Algorithm *DynPart*

```
procedure DynPart(D')
    for each d ∈ D' do
        for each q : comp(q, d) do
            for each F ∉ rel(q, π(D)) do
                if feasible(F) then
                    //aff(F) is initialized to 0
                    aff(F) ← aff(F) − f(q) minfr(q,π(D)) / (|rel(q,π(D))|(|rel(q,π(D))|+1))
                end if
            end for
        end for
        if ∃F ∈ π(D) : aff(F) > 0 then
            dests ← arg max_{F∈π(D)} aff(F)
        else
            dests ← {F ∈ π(D) : feasible(F)}
        end if
        F_dest ← select from arg min_{F∈dests} |F|
        move d to F_dest
        update metadata
    end for
end procedure
```

After computing the affinity of the relevant fragments, *DynPart* has to choose the best fragment for d. Not all of the fragments satisfy the imbalance constraints, thus we must only consider those that do meet the restrictions. We define the function $feasible(F)$ to determine whether a fragment can hold more data items or not. Accordingly, *DynPart* selects from the set of feasible fragments the one with the highest affinity. If there are multiple fragments that have the highest affinity, then the smallest fragment is selected, in order to keep the partitioning as balanced as possible.

DynPart works over a set of new data items D', instead of a single data item. This allows the system to perform bulk operations over a set of n data items instead of executing n times the same operations, which is in general more costly. Moreover, it gives the algorithm more flexibility in the application of the imbalance constraints and groups data insertions in each of the fragments.

Let $comp_{\text{avg}}$ be the average number of compatible queries per data item, and rel_{avg} be the average number of relevant fragments per query. Then, the average execution time of the algorithm is $O(comp_{\text{avg}} \times rel_{\text{avg}} \times |D'|)$, where $|D'|$ is the number of new data items to be appended to the fragments. The complexity can be $O(|W| \times |\pi(D)| \times |D'|)$ in the worst case, e.g. when all queries are compatible to all new data and the partitioning has not been done well. However, in practice, the averages are usually much smaller than the worst case values. The reason is that the queries usually access a small portion of the data (not the whole set), thus the average number of compatible queries per data item is low. In any case, in order to reduce the number of queries, we may use a threshold on the frequency, so that only queries above that threshold are considered. In addition, the partitioning efficiency of our approach is good (see experimental results in the next section), so the average number of relevant fragments per query is low.

3.4 Example

Figure 2 illustrates the execution of the *DynPart* algorithm. Before its execution, the system is partitioned into 4 fragments, whose sizes are shown in the figure. The workload consists of 5 queries, which are represented inside the fragments they access. There are 16 new data items, $d_1, ..., d_{16}$, that should be distributed over the fragments. The imbalance factor is $\epsilon_s = 0.05$, so resulting maximum size (taking into account new data items) is 42. We show the execution of the algorithm for some of the steps.

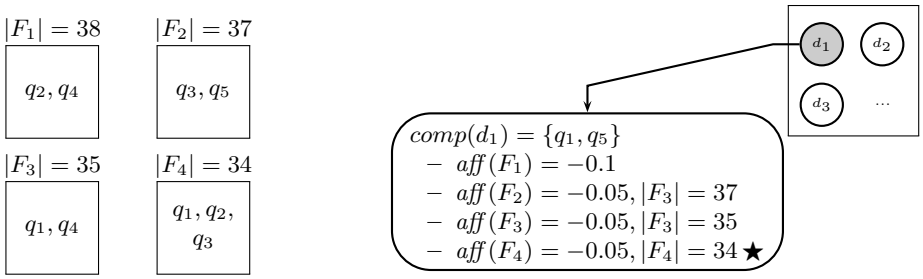
In Step 1 we show the insertion of data item d_1. The set of compatible queries is indicated in $comp(d_1)$. For each of these queries, the affinity of the relevant fragments is increased by the corresponding expression. As a consequence, F_1 has a total affinity of -0.1, resulting from the affinity expression applied to q_1 and q_5; and F_1, F_2 and F_3 have an affinity of -0.05, resulting from the expression applied to q_1 for F_2 and q_5 for F_3 and F_4. The three fragments have the highest affinity, but F_4 is selected since it is the smallest fragment.

In Step 2, the processing for data item d_2 is depicted. Note that the information has been updated as a consequence of last move: the size of F_4 has been incremented by 1 and the accessing queries now include q_5, provided that d_1 is accessed by it. In this case, the highest affinity is that of fragment F_4, so it is selected and d_2 is moved to it.

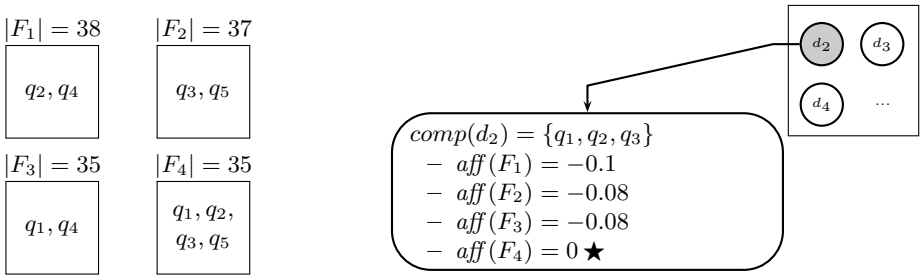
The algorithm continues to execute as before until Step 14. In that case, the fragment with the highest affinity is F_4, but it can not be selected, as it would violate the imbalance constraint. As a consequence, the next fragment in terms of affinity is selected and data item d_{14} is placed in fragment F_3.

$D = \{d_1, ..., d_{16}\}$, $\epsilon_s = 0.05$, $W = \{q_1, q_2, q_3, q_4, q_5\}$,

$f(q_1) = 0.3$, $q_1(D) = \{d_1, d_2, d_3, d_4, d_{11}, d_{12}, d_{13}, d_{14}, d_{15}, d_{16}\}$
$f(q_2) = 0.2$, $q_2(D) = \{d_2, d_9, d_{10}, d_{11}, d_{12}, d_{13}, d_{14}, d_{15}, d_{16}\}$
$f(q_3) = 0.3$, $q_3(D) = \{d_2, d_{11}, d_{12}, d_{13}, d_{14}, d_{15}, d_{16}\}$
$f(q_4) = 0.1$, $q_4(D) = \{d_9, d_{10}\}$
$f(q_5) = 0.1$, $q_5(D) = \{d_1, d_3, d_4\}$

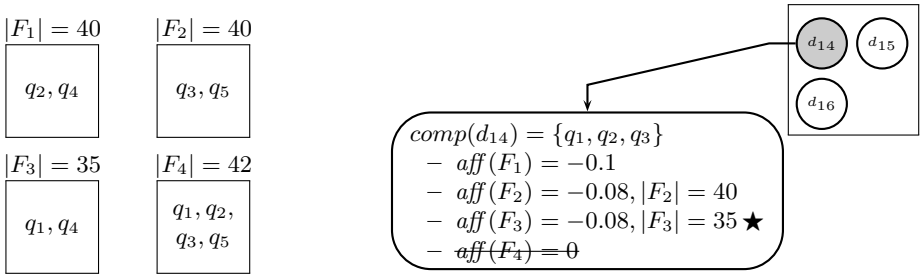


Step 1



Step 2

...



Step 14

Fig. 2. Example of operation of the *DynPart* algorithm

3.5 Data Structures

Our algorithm needs to maintain information about the relevant fragments of each query, so that we can compute the affinity efficiently. Queries are assigned a unique identifier and stored on a hash table for efficient access. For each of them, we store the set of relevant fragments as a list, as they are always accessed sequentially, i.e., no random access. Space complexity is $O(|W| \times |\pi(D)|)$ in the worst case, but, as we have pointed out, the average number of relevant fragments stays low even when the number of fragments increases. For example, in our experiments, for 1024 fragments, the average number of relevant fragments do not exceed 18 in any scenario. We also need to store the set of queries for each of the new data items. Again, as this set is accessed sequentially, we keep a list of query identifiers.

Our algorithm needs to create a data structure for each new data item to store the affinity of the possible destination fragments. For this, there are several alternatives. One option is to keep an array of size $|\pi(D)|$ initialized to zero. Note that, as the actual number of possible destinations is much lower than the total number of fragments, we would waste a lot of space with zero-affinity entries. Therefore, we keep a hash table of fragments and only compute those for which the affinity is non-zero. By using this method, access time will be maintained, while space requirements will be significantly reduced.

3.6 Dealing with Deletes and Updates

So far, we have only considered the case where data items are appended to the database. However, we could easily extend our approach to deal with deletions and updates. For a deletion, we only need to consider metadata maintenance. Whenever a data item d is deleted, the size of the fragment where it was placed should be reduced by one. We would also have to check for all queries compatible with d whether they still have to access that fragment or not, and update their set of relevant fragments if necessary. An efficient way to do this is to keep the number of data items accessed by each query on every of its relevant fragments, i.e., $|q(F)| \forall F \in rel(q, \pi(D))$. Then, whenever d is deleted from a fragment F, $|q(F)|$ would be reduced by 1. If the size reaches 0, then F should be deleted from the set of relevant fragments.

The case of updating a data item can be considered as a deletion followed by an insertion. However, we can benefit from previous information, and only recalculate the compatibility of queries that are affected by the changes.

4 Dealing with Imbalance

In the algorithm presented in the previous section, new data items are treated individually even if they are highly correlated. As a consequence, the destination chosen for them may differ if at a given point the selected fragment reaches the maximum size constrained by the imbalance factor. The problem might be

specially important when there are big groups of similar elements and/or the imbalance constraints are too restrictive. In this section we present a variation of the previous algorithm which tries to avoid such a situation by grouping similar elements together and taking a common decision for all the elements.

4.1 Algorithm

The extended version of our algorithm, which we call *DynPartGroup*, starts by dividing the buffer of new data items D' into a set of groups G such that all members of each group are accessed exactly by the same set of queries. Thus, the members of each group share exactly the same affinity for each given fragment. If they are allocated to different fragments, the partitioning efficiency of each of the incident queries is likely to decrease. The construction of the groups is included in Algorithm 2. A list of groups is built, where each group stores the set of composing tuples and the set of accessing queries. All items in a group are treated in the same way.

Algorithm 2 Function *CreateGroups*

```
function CreateGroups(D')
    G ← emptyList( )
    for each d ∈ D' do
        qs = {q : comp(q, d)}
        if ∃g ∈ G : g.qs = qs then
            g.ts ← g.ts ∪ {d}
        else
            g_new.ts ← {d}
            g_new.qs ← qs
            G ← insert(G, g_new)
        end if
    end for
    return G
end function
```

The algorithm (the pseudo-code is shown on Algorithm 3) first creates the groups and orders them by size in descending order, i.e., the biggest groups are considered before the smallest ones. The rationale is that, if we consider first the biggest groups, there is more free space on the fragments and the probability that all members of these groups fit on the same fragment is higher.

Once groups are ordered, an affinity value is calculated for each group, exactly in the same way it was done for individual data items in the basic algorithm. In this case, function $feasible(F, g)$ will return true if F plus the data items of the group g does not violate the imbalance factor, i,e:

$$feasible(F, g) = |F \cup g.ts| \leq \left\lceil \frac{|D|}{|\pi(D)|}(1 + \epsilon_s) \right\rceil \tag{8}$$

Algorithm 3 Algorithm *DynPartGroup*

procedure DynPartGroup(D')
  $G \leftarrow$ createGroups(D')
  order G by $|g.ts|$ in descending order
  while $G \neq \emptyset$ **do**
    $g \leftarrow$ first(G)
    $G \leftarrow G - \{g\}$
    for each $q \in g.qs$ **do**
      for each $F \notin rel(q, \pi(D))$ **do**
        if *feasible*(F) **then**
          //aff(F) *is initialized to 0*
          $aff(F) \leftarrow aff(F) - f(q)\frac{minfr(q,\pi(D))}{|rel(q,\pi(D))|(|rel(q,\pi(D))|+1)}$
        end if
      end for
    end for
    if $\exists F \in \pi(D) : aff(F) > 0$ **then**
      $dests \leftarrow \arg\max_{F \in \pi(D)} aff(F)$
    else
      $dests \leftarrow \{F \in \pi(D) : feasible(F)\}$
    end if
    if $dests \neq \emptyset$ **then**
      $F_{dest} \leftarrow$ select from $\arg\min_{F \in dests} |F|$
      move d to F_{dest}
      update metadata
    else
      split g into two equal sets g_1 and g_2
      insert g_1 and g_2 in G maintaining G's order
    end if
  end while
end procedure

If there is no feasible destination for F, the group is split into two equal halves and the resulting groups are inserted back in the list in the corresponding positions so that the order is maintained. At some point, those groups would be considered again but, in this case, individually. Note that other splitting strategies may be envisioned, e.g., assigning only the elements that fit in the fragment with the highest affinity and considering the rest as a new group. However, this will be in detriment of other big groups that might have to be subsequently split, and they will not offer any gain regarding the partitioning efficiency, as the group would be split anyway.

Let us now analyze the complexity of the algorithm. We divide the analysis in two parts; first we analyze the group creation and ordering part, and then the rest of the algorithm. Function CREATEGROUPS(D') has to go over all the elements in D'. Each of them has to be compared with existing groups to check if accessing queries match, which can be done by defining a hash function over the query sets. This function has a complexity of $O(|W|)$. As a result, the total complexity of group creation is $O(|D'| \times |W|)$. Let $|G|$ be the number of groups, then the complexity of group sorting is $O(|G| \times \log |G|)$. In the worst case, $|G| = |D'|$, but as we will see in the experimental section, the number of groups is usually much lower than that value.

The complexity of the rest of the algorithm is calculated in a similar way than in the basic algorithm. The main difference is the number of times the outer loop has to be executed. The worst case is the situation where there is a single group, the imbalance factor is near 0 and $|\pi(D)| \geq |D'|$. In that case, only one data item can be inserted on each fragment, and the group would have been split in $|D'|$ groups of size 1. This would cause $|D'| - 1$ splits and require $2 \times |D'| \in O(|D'|)$ executions of the outer loop, which would imply $O(|W| \times |\pi(D)| \times |D'|)$ affinity calculations, as in the basic algorithm.

The size of $|G|$ can vary throughout the execution, as each split increases its size by one. In the worst scenario explained above, its size will increase until reaching $|D'|$, point from which it will be consumed, as all groups would be of size 1. Assume that the ordered insertion on G is executed on $O(\log |G|)$. Then, all the sequence of insertions would need $O(\log 1) + O(\log 2) + ... + O(\log |D'|) = O(\log |D'|!) = O(|D'| \log |D'|)$. Hence, the worst case complexity is $O(|W| \times |\pi(D)| \times |D'| + |D'| \log |D'|)$

However, that worst case is very rare as usually there are a higher number of groups, and the splits are uncommon. Thus, we can say that in the average case execution complexity of this part of the algorithm is $O(comp_{\text{avg}} \times rel_{\text{avg}} \times |G|)$.

4.2 Example

Figure 3 compares the assignments performed by the basic version of the algorithm (*DynPart*), and the algorithm we described above (*DynPartGroup*), in the same scenario as in the previous section. Compatible queries for all data items are shown in previous example but can also be inferred from the groupings shown in the top of the figure, i.e., all the data items in a group have the corresponding set of compatible queries. In the basic algorithm, data items are assumed to

be processed in the order indicated in the subindex, i.e., first d_1, then d_2, etc. Finally, recall that an imbalance factor of 0.05 for a fragment of size 40 means that the maximum size of the fragment at the end of the execution is 42.

Figure 3(a) shows the final assignment performed by the extended algorithm. All the groups are assigned to a single fragment and the chosen fragments have always one of the highest affinities, so the allocations are optimal. In figure 3(b) the assignments resulting from the execution of the basic algorithm are depicted. Note that, in this case, groups g_1 and g_2 have to be split into different fragments. As a consequence, q_1, q_3 and q_5 increment the number of accessed fragments by 1 and q_2 by 2, thus decreasing partitioning efficiency. This is the consequence of fragment F_4 being at its maximum size in step 14, which prevents it to be selected in further phases of the algorithm.

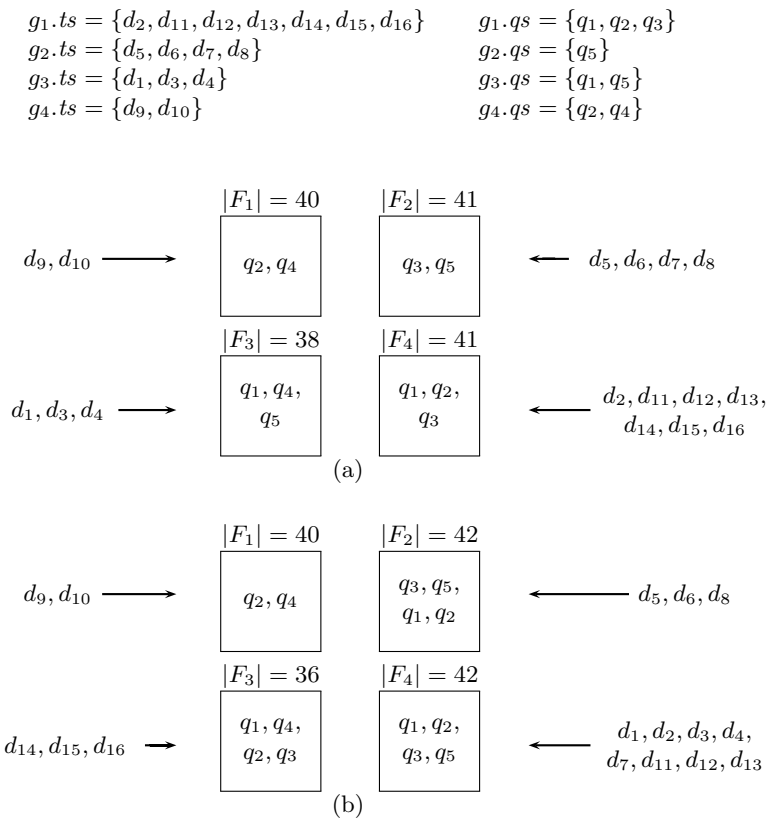


Fig. 3. Example of execution of the distribution algorithms: a) algorithm *DynPart-Group*, b) algorithm *DynPart*

4.3 Balancing Fragments Based on Load

In Section 2, we modeled the problem of data partitioning by using a size balancing constraint. Nonetheless, the problem may also be formalized if a load balancing constraint is required. Intuitively, with load we mean the number of accesses to the fragments.

Let us first define formally the load of a dataset as follows.

Definition 4. *The load of a data set D, denoted $L(D)$ is defined as the sum of the frequencies of the queries accessing its data items:*

$$L(D) = \sum_{q \in W} f(q) \times |q(D)| \tag{9}$$

Given this definition, we can reformulate the imbalance constraint in the following way: $L(F) \leq \frac{L(D)}{|\pi(D)|}(1 + \epsilon_l)$. As a result, the formula for the minimum number of fragments that should be accessed for a given query should be modified accordingly:

$$minfr(q, \pi(D)) = \left\lceil \frac{L(q(D))}{(L(D) \,/\, |\pi(D)|)(1 + \epsilon_l)} \right\rceil \tag{10}$$

Note that in the numerator we use $L(q(D))$ instead of $|q(D)|$ because we should take into account that items accessed by q are also accessed by other queries that we have to consider.

To use this new imbalance constraint, our algorithms only need some minor modifications as follows. In Algorithm 1, in case of ties in the affinity measure, the least loaded fragment should be selected instead of the smallest one. Moreover, in Algorithm 3, groups should be ordered by load instead of by size. Furthermore, function *feasible* should be redefined as follows:

$$feasible(F, g) = L(F \cup g.ts) \leq \left\lceil \frac{L(D)}{|\pi(D)|}(1 + \epsilon_l) \right\rceil \tag{11}$$

5 Experimental Evaluation

To validate our dynamic partitioning algorithms, we conducted a thorough experimental evaluation over real-world data. In Section 5.1, we describe our experimental setup. In Section 5.2, we report on the execution time of our algorithms and compare them with a well known static workload-based algorithm. In Section 5.3, we study the effect of the heuristic, which we used in our algorithms, on partitioning efficiency. Finally, Section 5.4 studies how the imbalance factor and the correlation of new data affect the partitioning efficiency.

5.1 Set-up

For our experimental evaluation we used the data from the Sloan Digital Sky Survey catalog, Data Release 8 (DR8) [2], as it is being used in LIneA in Brazil[6].

[6] Data from the DES project is still unavailable, so we have used data from SDSS, which is a similar, previous project

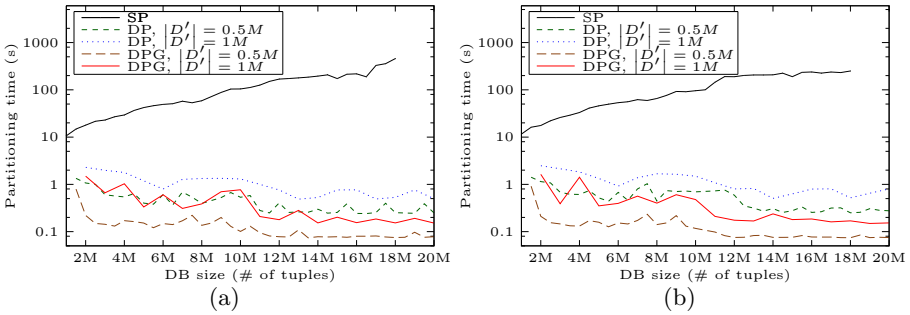


Fig. 4. Comparison of partitioning times of the dynamic and graph-based partitioning algorithms as the DB size increases ($|\pi(D)| = 16$) for a) data size balancing ($\epsilon_s = 0.15$) and b) load balancing ($\epsilon_l = 0.15$).

It consists of a relational database with several observations for both stars and galaxies. We obtained a workload sample from the SDSS SkyServer SQL query log data, which stores the information about the real accesses performed by users. In total, the database comprises almost 350 million tuples, that take 1.2 TB of space. The query log consists of a total of 27000 queries, some of which are similar in the SQL form but produce different results, as they use different parameters.

All queries were executed on the database and the tuple ids accessed by each of them were recorded. Only tuples accessed by at least one query were considered. We simulated the insertions on the database by selecting a subset of the tuples as the initial state and appending the rest of the tuples in groups. We varied the following parameters: 1) the number of tuples inserted to the database on each execution of our algorithm, $|D'|$; 2) the number of fragments in which the database is partitioned, $|\pi(D)|$; 3) the imbalance factors, ϵ_s and ϵ_l; and 4) the order of data items, so as to produce datasets with higher correlation between consecutive data items. On each of the experiments, the specific numbers are detailed.

All experiments were executed in a 3.0 GHz Intel Core 2 Duo E8400, running Ubuntu 11.10 64-bit with 4GB of memory.

5.2 Partitioning Time

In this section, we study the execution time of the dynamic algorithms *DynPart* (DP in the figure) and *DynPartGroup* (DPG) and compare them with a static graph partitioning algorithm (SP). For the later, we use PaToH[7], an hypergraph partitioner. Figure 4 shows the comparison of the partitioning time for 16 fragments and for data size balancing ($\epsilon_s = 0.15$) and load balancing ($\epsilon_l = 0.15$). We executed the dynamic algorithms with two values for $|D'|$: 500000 and 1

[7] `http://bmi.osu.edu/~umit/software.html`

million tuples. Similar results are obtained for different values of $|\pi(D)|$. As the difference between execution times of the static and the dynamic algorithms is significant, we use a logarithmic scale for the y-axis in order to show the results. The results are only depicted until a database size of 20 million tuples, as the memory requirements for the static partitioning are bigger than the memory of our servers. The dynamic algorithms, on the other hand, do not cause any problem as the memory footprint depends on $|D'|$, which is constant throughout the experiment.

As it can be seen, partitioning time increases for the graph partitioning algorithm as the size of the database increases, provided that the size of the graph increases accordingly. For the dynamic algorithms, on the other hand, the execution time stays at the same level, as it is always executed for the same number of data items. Some variation is observed since the features of the new items adapt differently to the partitioning. However the trend is constant.

In the figure, we can also observe that the execution times of the *DynPart-Group* algorithm are better that those of the basic algorithm. This is caused by the reduced number of affinity calculations, as we will show later.

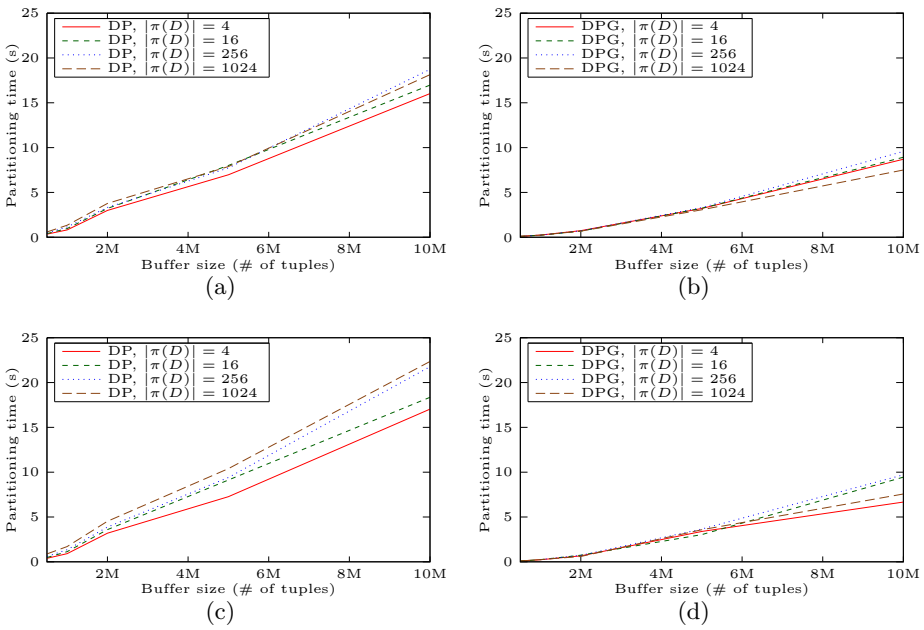


Fig. 5. Partitioning time vs. $|D'|$ for a) *DynPart* and data size balancing ($\epsilon_s = 0.15$), b) *DynPartGroup* and data size balancing ($\epsilon_s = 0.15$), c) *DynPart* and load balancing ($\epsilon_l = 0.15$) and d) *DynPartGroup* and load balancing ($\epsilon_l = 0.15$).

We compared the execution of our algorithms for different sizes of D'. Figure 5 shows the average execution time of the *DynPart* and the *DynPartGroup* algorithms as $|D'|$ increases for different number of fragments and for both balancing strategies. As expected, the execution time is linearly related to the buffer size. Also, the higher number of fragments, the higher the execution time. This increase is not linear since the number of relevant fragments does not increase at the same pace. In fact, the number of relevant fragments does not exceed 8 for $|\pi(D)| = 256$ and 16 for $|\pi(D)| = 1024$. The difference on the execution time between the *DynPart* and the *DynPartGroup* algorithms is also noticeable.

In Figure 6, we represent the average execution times for the different stages of the dynamic algorithms corresponding to the same scenario of Figure 4. Both algorithms contain the following stages: calculate affinities, select max affinity and update metadata. The extended algorithm also contains two additional stages, namely create groups and sort groups. Finally, another phase is depicted, which represents the rest of the operations executed during the distribution but not linked to a particular algorithm.

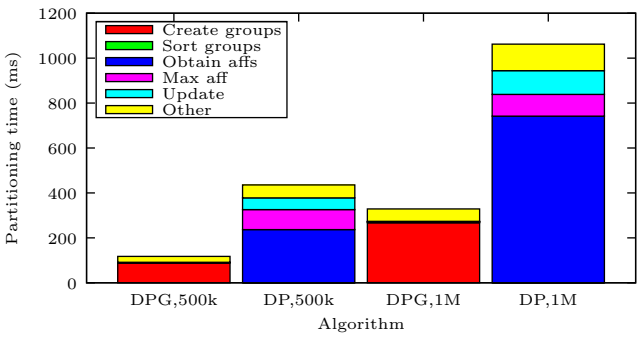


Fig. 6. Comparison of dynamic algorithms' execution times (data size balancing with $\epsilon_s = 0.15$)

As we can observe in the figure, the distribution of execution times is completely different for both algorithms. The *DynPart* algorithm spends most of the time in the calculation of the affinities, although the time spent in the rest of the phases is also significant. On the other hand, *DynPartGroup* spends almost all the time in the creation of the groups, whereas the time spent in the rest of the stages is negligible. This can be explained by considering the number of groups created in average, 664 for $|D'| = 500k$ and 1360 for $|D'| = 1M$, which represent around 0.13% of the number of tuples. As a consequence, with *DynPartGroup* the time for computing affinities, selecting the best fragment, and updating the corresponding metadata is significantly reduced.

5.3 Partitioning Efficiency

One of the important issues to consider for the dynamic algorithms is how they affect the partitioning efficiency.

We executed the algorithms as the database is fed with new data after an initial partitioning using the graph-based partitioning approach. With $|D'| = 1\,\mathrm{M}$, Figure 7 shows how the partitioning efficiency evolves as the database grows for different number of fragments, $|\pi(D)|$. Similar results were obtained for other configurations of $|D'|$. The efficiency decreases as the database grows, as expected, but this reduction is very small. For example, in the worst case, $|\pi(D)| = 1024$ and data size balancing, the partitioning efficiency decreases 2.82×10^{-3} in average for each 10 million new tuples. The difference between *DynPart* and *DynPartGroup* is very small for small values of $|\pi(D)|$, but increases for higher values. In any case, it is below 5% for the worst case.

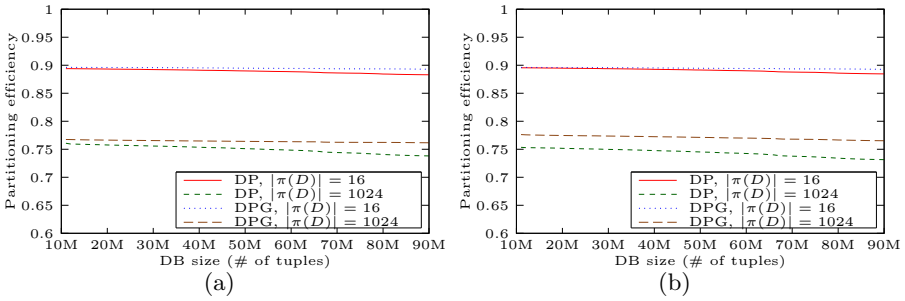


Fig. 7. Comparison of partitioning efficiency as the size of the DB grows ($|D'| = 1M$) for a) data imbalance and b) load imbalance

To evaluate the quality of our partitioning approach, in addition to the partitioning efficiency metrics, as in [8, 16] we studied the percentage of single-node queries, which means the percentage of the queries that can be executed by using the data of only one fragment. Figure 8 shows the results. As seen, when the number of fragments is small, the results are similar to what we reported for partitioning efficiency metrics. However, for higher number of nodes, the number of single-node queries is lower. The reason is that in these cases the partitions are smaller, so it is more difficult to confine all the results of a query in a single fragment

5.4 Effect of Imbalance Factor and Data Correlation

The imbalance factor (ϵ_s or ϵ_l) may affect the efficiency as it constraints the flexibility of the algorithm in allocating new data items. The lower the imbalance factor, the less flexibility, which may imply that some data items are not placed in the optimal fragments because they are full. Figures 9(a) and 9(c) show the

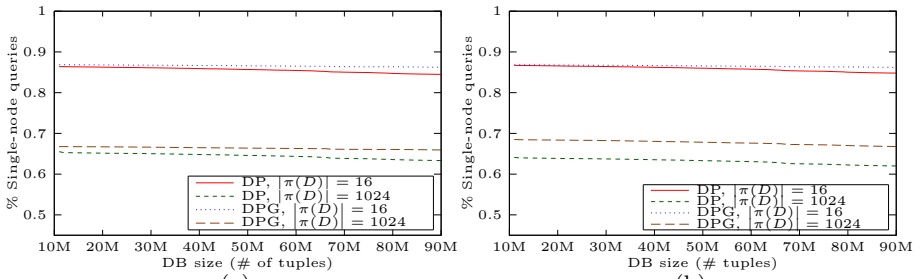


Fig. 8. Comparison of percentage of single-node queries as the size of the DB grows ($|D'| = 1M$) for a) data imbalance and b) load imbalance

average partitioning efficiency for different values of ϵ_s and ϵ_l, respectively. The efficiency decreases as the imbalance factor decreases, as expected, but it is much more noticeable for the *DynPart* algorithm.

To enrich our study, we have considered other scenarios by reordering the data so that correlated data items arrive together. In order to do that, we executed the *DynPart* algorithm over the initial data set and created the corresponding partitions. Then we reordered the data by placing on defined intervals data of only one of the fragments at a time. That way, we increase the correlation of new data (D') on each execution of the algorithm.

Figures 9(b) and 9(d) show the same configuration as before but with a new ordering created by producing 8 fragments on the original data and placing items of one of those fragment in intervals of $10M$[8]. As we see, in the case of correlated data, the impact of the imbalance factor is higher than in the previous scenario. Nevertheless, the *DynPartGroup* algorithm still shows good behavior for different values of ϵ_s and ϵ_l.

Finally, in Figure 10 we show the evolution of the partitioning efficiency as the database grows for imbalance factors of 0.001 and 0.5, which represent both extremes on the studied values of ϵ_s and ϵ_l. This confirms that higher correlations on the inserted data affect the resulting partitioning efficiency. At the beginning the efficiency is low, since all the inserted data is highly correlated and data items that should be allocated together have to be split because of imbalance constraints. However, as new data items with different affinities are included and the imbalance is more flexible, the efficiency increases.

By comparing the behavior of both dynamic algorithms we can state that the *DynPartGroup* algorithm obtains better partitioning efficiencies consistently. The *DynPart* algorithm approaches *DynPartGroup* when the imbalance factor is high, but degrades as the imbalance constraints are stricter. This difference between the partitioning efficiency of the two algorithms is even higher for configurations with more number of fragments.

[8] We have produced different reorderings and the experiments show similar results

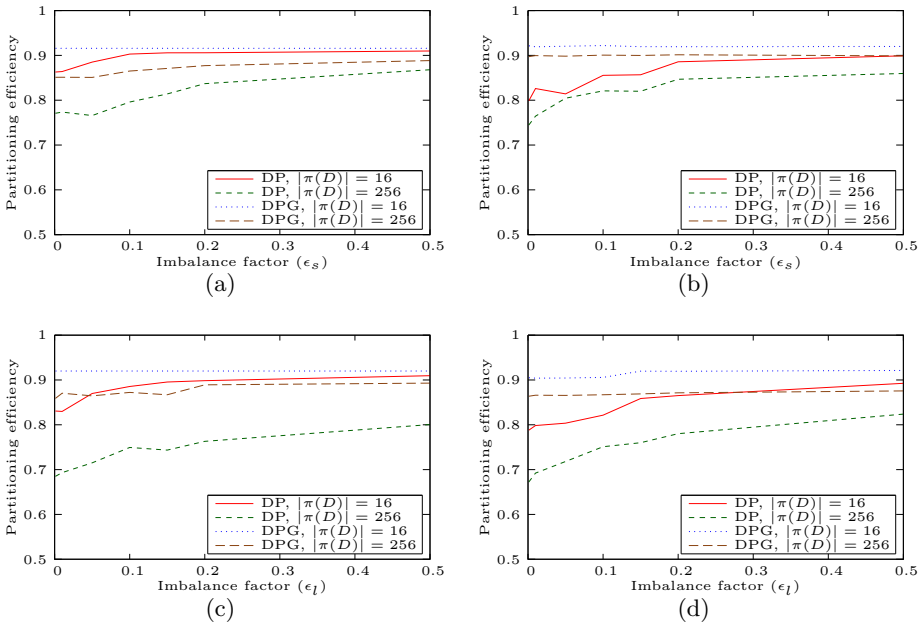


Fig. 9. Partitioning efficiency vs. imbalance factor for a) original data set and data size balancing, b) reordered data set and data size balancing, c) original data set and load balancing and d) reordered data set and load balancing

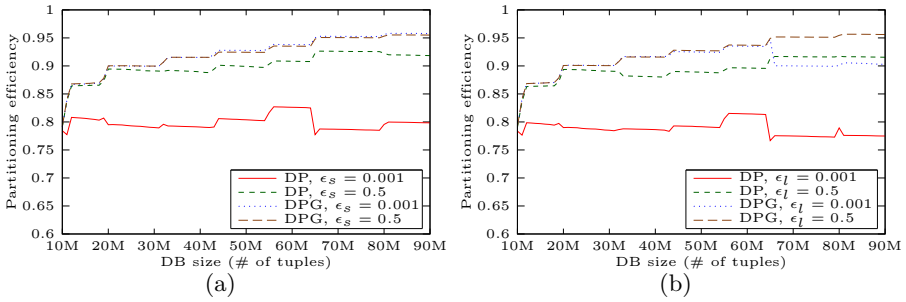


Fig. 10. Partitioning efficiency for the reordered data ($|\pi(D)| = 16$) for a) data size balancing and b) load balancing

6 Related Work

Partitioning has been used both for declustering (whose goal is to maximize parallelism) and clustering (to minimize the fragment accesses). In this paper, we are interested in the later, as we are trying to reduce the number of query accesses to the fragments.

The most popular approaches for database partitioning [10] are 1) round-robin, which consists on assigning each tuple to a different fragment; 2) hash-partitioning, which applies a hash function of a predefined set of attributes; and 3) range-based partitioning, which splits data on ranges on a given set of attributes. Recently, distributed key-value stores have been applying them. Dynamo [9] uses a modified version of hash-partitioning on the key and, as a consequence, only obtain single-site query executions when the query contains equality predicates on the key. In general, hash-based partitions are good for clustering only when the queries contain equality predicates on the partitioning attributes, which is not the case of our workload. BigTable [5] and PNUTS [7] use range-based partitioning on the keys; which still is too simple for our reference queries. In general, the complexity of scientific workloads makes it hard to design a good partitioning strategy manually, so automatic partitioning is preferred [15].

Automatic database partitioning have received significant attention from researchers and applied by some database vendors, notably Microsofts SQL Server AutoAdmin [6, 3] and IBMs DB2 Database Advisor [17, 19]. Many of these works have focused on partitioning (both vertical and horizontally) as an element of physical design for a single-node, along with indexing and materialized views. For instance, in [3] a set of physical design alternatives (that includes partitioning) is generated. Then, in order to limit the search space they prune the set of candidates. Similar procedures are used in other works, such as AutoPart [15], which is focused on scientific workloads. In this case only vertical and categorical partitioning are considered. After generating a set of fragment candidates from the predicates in the workload, composition of fragments is evaluated to reduce the overhead of joins. The resulting partitionings are also used for physical design in a single-node.

Some other proposals use analogous techniques to automatically generate partitions in distributed systems. The solution proposed in [17] uses a similar approach but with the goal of distributing the queries over all the nodes (data declustering). For the queries in the workload model a set of candidate partitions, which consist of applying a hash partitioning over a subset of columns, is generated. Then, they use the optimizer to estimate the costs under the new partitioning and eventually recommend some of the candidates. Automatic database partitioning for distributed databases has recently received further attention. In [14], data is partitioned automatically to optimize the execution of MPP systems. As a possible alternative they only consider hash-based partitioning over a single column. In [16], both hash and range-based partitioning on the most accessed attributes are considered for partitioning in OLTP systems. To find a near optimal solution, their approach explores a solution space by adapting the large-neighborhood search technique. However, this approach and most of the

approaches mentioned above are not well suited for our underlying scientific applications that are characterized by complex workload predicates involving many attributes; and this significantly degrades the efficiency of those approaches

Graph-based approaches have been used to capture more complex relations between the workload and the data both for partitioning with the objective of declustering [13, 11] and clustering [8]. They use two different models to represent data and queries: simple graph and hypergraph. In the hypergraph model [11], each query is modeled as a hyperedge (a set of vertices). In the simple graph model [13, 8], queries are modeled as cliques of simple edges. Schism [8] is a recent system that partitions the data by building a graph containing the relations between queries and tuples. Data items are retrieved using an index or by means of predicate-based explanations, depending on the scenario. However, like other existing graph-based approaches, it is static and needs to redo the partitioning from scratch when the data changes. As we showed in the paper, this approach does not perform well for growing databases, and a dynamic approach is hence required. Furthermore, as new produced partitionings are not aware of previous ones, large amounts of data transfers may have to take place in order to apply the new data placements.

7 Conclusions

In this paper, we proposed a pair of dynamic algorithms for partitioning continuously growing large databases. We modeled the partitioning problem for dynamic datasets and proposed a new heuristic to efficiently distribute new arriving data, based on the affinity it has with the different fragments in the application. We designed two alternatives, *DynPart*, the basic algorithm, and *DynPartGroup*, which deals better with strict imbalance constraints.

We validated our approach through implementation, and compared its execution time with that of a static graph-based partitioning approach. The results show that as the size of the database grows, the execution time of the static algorithm increases significantly, but that of our algorithms remains stable. They also show that, for the given dataset, our algorithms, although based on a heuristic approach, do not degrade partition efficiency considerably.

The results show that in the case of datasets in which there is a high correlation between new data items, the *DynPartGroup* algorithm maintains a very good behavior. The also show that this algorithm is not highly affected by the imbalance of fragments' sizes.

On the whole, our experiments show that our dynamic partitioning strategy is able to efficiently deal with the data of our astronomic application. But, we believe that its utilization is not limited to this application, and it can be used for data partitioning in many other applications in which the data items are appended continuously. We leave for a possible future work the scenarios with even higher data correlation where a simple eager approach, like ours, does not work and some form of data reorganization is needed.

References

1. The dark energy survey. http://www.darkenergysurvey.org/
2. Sloan digital sky survey. http://www.sdss3.org
3. Agrawal, S., Narasayya, V.R., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: Weikum, G., König, A.C., Deßloch, S. (eds.) SIGMOD Conference. pp. 359–370. ACM (2004)
4. Ailamaki, A., Kantere, V., Dash, D.: Managing scientific data. Communications of the ACM 53(6), 68–78 (2009)
5. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Transactions on Computer Systems 26(2), 1–26 (2008)
6. Chaudhuri, S., Narasayya, V.R.: Autoadmin 'what-if' index analysis utility. In: Haas, L.M., Tiwary, A. (eds.) SIGMOD Conference. pp. 367–378. ACM Press (1998)
7. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s hosted data serving platform. Proceedings of the VLDB Endowment 1(2), 1277–1288 (2008)
8. Curino, C., Jones, E., Zhang, Y., Madden, S.: Schism: a workload-driven approach to database replication and partitioning. Proceedings of the VLDB Endowment 3(1), 48–57 (2010)
9. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. SOSP '07, vol. 41, pp. 205–220. ACM (2007)
10. DeWitt, D.J., Gray, J.: Parallel database systems: the future of high performance database systems. Communications of the ACM 35(6), 85–98 (Jun 1992)
11. Koyutürk, M., Aykanat, C.: Iterative-improvement-based declustering heuristics for multi-disk databases. Information Systems 30, 47–70 (2005)
12. Liroz-Gistau, M., Akbarinia, R., Pacitti, E., Porto, F., Valduriez, P.: Dynamic workload based partitioning for large-scale databases. In: 23rd International Conference on Database and Expert Systems Applications. DEXA '12 (2012)
13. Liu, D.R., Shekhar, S.: Partitioning similarity graphs: a framework for declustering problems. Information Systems 21(6), 475–496 (1996)
14. Nehme, R.V., Bruno, N.: Automated partitioning design in parallel database systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 1137–1148 (2011)
15. Papadomanolakis, S., Ailamaki, A.: Autopart: Automating schema design for large scientific databases using data partitioning. In: SSDBM. pp. 383–392. IEEE Computer Society (2004)
16. Pavlo, A., Curino, C., Zdonik, S.B.: Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 61–72 (2012)
17. Rao, J., Zhang, C., Megiddo, N., Lohman, G.M.: Automating physical database design in a parallel database. In: Franklin, M.J., Moon, B., Ailamaki, A. (eds.) SIGMOD Conference. pp. 558–569. ACM (2002)
18. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-store: a column-oriented DBMS. In: Proceedings of the 31st international conference on Very Large Data Bases. pp. 553–564. VLDB '05 (2005)

19. Zilio, D.C., Rao, J., Lightstone, S., Lohman, G.M., Storm, A.J., Garcia-Arellano, C., Fadden, S.: Db2 design advisor: Integrated automatic physical database design. In: Nascimento, M.A., Özsu, M.T., Kossmann, D., Miller, R.J., Blakeley, J.A., Schiefer, K.B. (eds.) VLDB. pp. 1087–1097. Morgan Kaufmann (2004)