DynPart: Dynamic Partitioning for Large-Scale Databases

Miguel Liroz-Gistau [1] Reza Akbarinia [1] Esther Pacitti [2]
Fabio Porto [3] 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

Résumé

Il y a de plus en plus des applications scientifiques avec de très grandes bases de données distribuées où des nouvelles données sont ajoutées à la base de données en permanence. Pour offrir une bonne performance à la plupart de ces applications qui ont normalement des schémas d'accès complexes, on a besoin de développer des méthodes efficaces pour le partitionnement de données basé sur le workload. Toutefois, les approches existant basées sur le workload, qui sont exécutées d'une manière statique, ne peuvent pas être appliquée aux bases de données très volumineuses et dynamiques. Dans cet article, nous proposons DynPart, un algorithme dynamique de partitionnement pour les bases de données en croissance permanente. DynPart s'adapte efficacement à l'arrivée de nouvelles données en prenant en compte l'affinité de ces données avec les requêtes et les fragments. Contrairement aux approches antérieures, notre approche offre un temps d'exécution constant, peu importe la taille de la base de données, tout en obtenant une très bonne efficacité de partitionnement. Nous avons validé notre solution par l'expérimentation sur des données réelles ; les résultats montrent sa bonne performance.

Mots-clefs : Bases de données distribuées, partitionement dynamique, partitionnement basé sur workload

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 [1]. An example of these applications is the management of astronomical catalogs; for instance those generated by the Dark Energy Survey (DES) [7] project on 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. 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 query 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 several servers. An efficient partitioning scheme would try to minimize the number of fragments accessed in the execution of a query, thus reducing the overhead associated to handle the distributed execution. Vertical partitioning solutions, such as column-oriented databases [6], 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 require data transfers between nodes. Traditional horizontal approaches, such as hashing or range-based partitioning [2, 3], are unable to capture the complex access patterns of scientific applications, especially since they usually make use of complex relations over big sets of columns, and are hard to be predefined.

One solution is to use partitioning techniques based on the workload. Graph-based partitioning is an effective approach for that purpose [4]. 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 always requires to explore 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 a dynamic workload-based algorithm, called *DynPart*, that efficiently adapts the partitioning to the arrival of new data elements. Our algorithm is 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 algorithm does not depend on the total size of the database, but only on that of the new data and this makes it appropriate for continuously growing databases. We validated our solution through experimentation over real-world data sets. The results show that it obtains high performance gains in terms of partitioning execution time compared to one of the most efficient static partitioning algorithms.

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 solution for dynamic data partitioning. Section 4 reports on the results of our experimental validation and Section 5 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 $f : W \rightarrow [0, 1]$, 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 and the union of all fragments is equal to 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 fragments' load, we use an *imbalance factor*, denoted by ϵ. We define the load of a dataset D as $L(D) = \sum_{q \in W} f(q)|q(D)|$. Then, the load of each fragment must satisfy: $L(F) \leq \frac{L(D)}{|\pi(D)|}(1 + \epsilon)$.

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 $minfr(q, \pi(D)) = \left\lceil \frac{L(q(D))}{(L(D)\,/\,|\pi(D)|)(1+\epsilon)} \right\rceil$. We define the *efficiency of a partitioning* for a workload based on its efficiency for queries. Let us first define the *efficiency of a partitioning for a query* as follows:

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

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

In the equation above, when the number of accessed fragments is equal to the minimum possible, i.e. $minfr(q, \pi(D))$, the efficiency is 1. Using $\mathit{eff}(q, \pi(D))$, 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 $\mathit{eff}(W, \pi(D))$, is

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

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 $\mathit{eff}(W, \pi(D))$ 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 $D(t)$.

During the application execution, there are some events, namely *data insertions*, by which new data items are inserted into D. Let $T_{\text{ev}} = (t_1, \ldots, t_m)$ be the sequence of time points corresponding to those events. 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 W be a given workload. Then, the goal is to find a set of partitionings $\pi(D(t_1)), ..., \pi(D(t_m))$ such that the sum of the efficiencies of the partitionings for W is maximized. In other words, our objective is as follows:

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

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 Principle

Let d be a new inserted data item. From the definition of partitioning efficiency, we infer that if we place d in a fragment F, then the total efficiency varies according to the following approximation [1]:

$$\mathit{eff}(W, \pi(D \cup \{d\})) \approx \mathit{eff}(W, \pi(D)) - \sum_{q:q(F)=\emptyset \wedge comp(q,d)} f(q) \frac{minfr(q, \pi(D))}{|rel(q, \pi(D))|\,(|rel(q, \pi(D))| + 1)}, \quad (3)$$

Thus, the partitioning efficiency is reduced whenever there are queries that did not access F but after the insertion of d to F have to access it, thereby increasing the number of relevant fragments. The lower the number of those queries, the less the resulting loss of efficiency. Based on this idea, we define *the affinity between the data d and fragment F*, which we denote as

$$\mathit{aff}(d, F) = \sum_{q:q(F)\neq\emptyset \wedge comp(q,d)} f(q) \frac{minfr(q, \pi(D))}{|rel(q, \pi(D))|\,(|rel(q, \pi(D))| + 1)} \quad (4)$$

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

Using Equation 4, we develop a heuristic algorithm that assigns each new data item to the fragment which maximizes the affinity measure.

3.2 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 Equation 4. Initially the affinity of fragments is set to zero.

Algorithm 1 *DynPart* algorithm

for each $d \in D'$ **do**
  for each $q : comp(q, d)$ **do**
    for each $F \in rel(q, \pi(D))$ **do**
      if $feasible(F)$ **then**
        //aff(F) *is initialized to 0*
        $aff(F) \leftarrow aff(F) + f(q)\frac{minfr(q,\pi(D))}{|rel(q,\pi(D))|(|rel(q,\pi(D))|+1)}$
  if $\exists F \in \pi(D) : aff(F) > 0$ **then** $dests \leftarrow \arg\max_{F \in \pi(D)} aff(F)$
  else $dests \leftarrow \{F \in \pi(D) : feasible(F)\}$
  $F_{dest} \leftarrow$ select from $\arg\min_{F \in dests} L(F)$
  move d to F_{dest} and update metadata

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 least loaded 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 particularly reduces the amortized cost of finding the set of queries that are compatible with each of the inserted items. This is why, in practice, we wait until

a given number of items have been inserted and then execute our algorithm for partitioning the new data.

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. 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, thus the average number of compatible queries 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, so the average number of relevant fragments per query is low.

4 Experimental Evaluation

To validate our dynamic partitioning algorithm, we conducted a thorough experimental evaluation over real-world data. In Section 4.1, we describe our experimental setup. In Section 4.2, we report on the execution time of our algorithm and compare it with a well known static workload-based algorithm. In Section 4.3, we study the effect our heuristic on the partitioning efficiency.

4.1 Set-up

For our experimental evaluation we used the data from the Sloan Digital Sky Survey catalog, Data Realease 8 (DR8) [5], as it is being used in LIneA in Brazil [2]. 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 workload consists of a total of 27000 queries.

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

2. Data from the DES project is still unavailable, so we have used data from SDSS, which is a similar, previous project

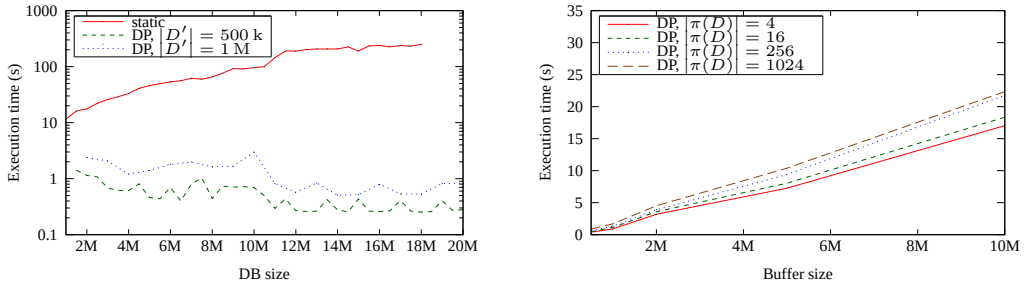

Figure 1: a) Execution times of the static (SP) and dynamic (DP) approaches as DB size increases ($|\pi(D)| = 16$), b) *DynPart* execution time vs. $|D'|$

following two parameters: 1) the number of tuples inserted to the database before each execution of our algorithm, $|D'|$; and 2) the number of fragments in which the database is partitioned, $|\pi(D)|$. On each of the experiments, the specific numbers are detailed. Throughout the experiments we chose an imbalance factor of 0.15. All experiments were executed in a 3.0 GHz Intel Core 2 Duo E8400, running Ubuntu 11.10 64 bits with 4GB of memory.

4.2 Execution time

In this section, we study the execution time of the *DynPart* algorithm (DP in the figure) and compare it with a static graph partitioning algorithm (SP). For the later, we use PaToH [3], an hyper-graph partitioner. Figure 1(a) shows the comparison of the execution time for 16 fragments. In the case of the dynamic algorithm, we executed the algorithm with two values for $|D'|$: 0.5 and 1 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 algorithm does not cause any problem as the memory footprint depends on $|D'|$, which is constant throughout the experiment.

As we can seen, execution time increases for the static algorithm as the size of the database increases, provided that the size of the graph increases accordingly.

3. `http://bmi.osu.edu/~umit/software.html`

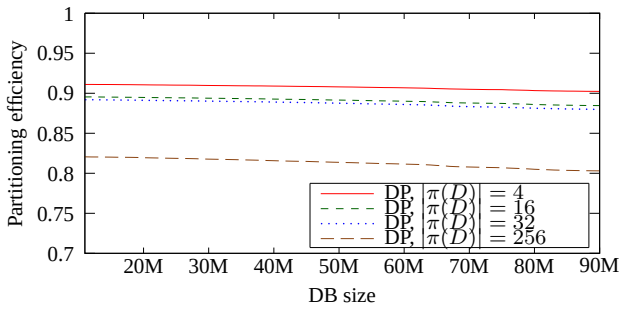

Figure 2: Comparison of partitioning efficiency as the size of the DB grows ($|D'| = 1M$)

For the *DynPart* algorithm, on the other hand, the execution time stays at the same level, as it is always executed for the same number of data items.

We executed our algorithm for different sizes of D'. Figure 1(b) shows the average execution time of the *DynPart* algorithm as $|D'|$ increases for different number of fragments. As expected, the execution time is linearly related to the number of tuples before execution. 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 $|\pi(D)| = 1024$.

4.3 Partitioning Efficiency

One of the important issues to consider for the *DynPart* algorithm is how our heuristic algorithm affects the partitioning efficiency. We executed the *DynPart* algorithm as the database is fed with new data after an initial partitioning using the static approach. With $|D'| = 1\,\mathrm{M}$, Fig. 2 shows how the partitioning efficiency evolves as the database grows for different number of fragments. 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, $|\pi(D)| = 256$, the partitioning efficiency decreases 2.19×10^{-3} in average for each 10 million new tuples.

5 Conclusions

In this paper, we proposed *DynPart*, a dynamic algorithm 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 its affinity with the different fragments in the application.

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 algorithm remains stable. They also demonstrate that although the *DynPart* algorithm is designed based on a heuristic approach, it does not degrade partitioning efficiency considerably.

Based on the results we can state that our dynamic partitioning strategy is able to efficiently deal with the data of our application. But, we believe that its use 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.

References

[1] A. Ailamaki, V. Kantere, and D. Dash. "Managing scientific data". In: *Communications of the ACM* 53.6 (2009), pp. 68–78.

[2] F. Chang et al. "Bigtable: a distributed storage system for structured data". In: *ACM Transactions on Computer Systems* 26.2 (2008), pp. 1–26.

[3] B. F. Cooper et al. "PNUTS: Yahoo!'s hosted data serving platform". In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1277–1288.

[4] C. Curino et al. "Schism: a workload-driven approach to database replication and partitioning". In: *Proceedings of the VLDB Endowment* 3.1 (2010), pp. 48–57.

[5] *Sloan Digital Sky Survey*. `http://www.sdss3.org`.

[6] M. Stonebraker et al. "C-store: a column-oriented DBMS". In: *Proceedings of the 31st international conference on Very Large Data Bases*. VLDB '05. 2005, pp. 553–564.

[7] *The Dark Energy Survey*. `http://www.darkenergysurvey.org/`.