Distrib Parallel Databases (2009) 25: 97–123
DOI 10.1007/s10619-009-7037-8

Parallel OLAP query processing in database clusters with data replication

Alexandre A.B. Lima · Camille Furtado · Patrick Valduriez · Marta Mattoso

Published online: 25 February 2009

Abstract We consider the problem of improving the performance of OLAP applications in a database cluster (DBC), which is a low cost and effective parallel solution for query processing. Current DBC solutions for OLAP query processing provide for intra-query parallelism only, at the cost of full replication of the database. In this paper, we propose more efficient distributed database design alternatives which combine physical/virtual partitioning with partial replication. We also propose a new load balancing strategy that takes advantage of an adaptive virtual partitioning to redistribute the load to the replicas. Our experimental validation is based on the implementation of our solution on the **SmaQSS** DBC middleware prototype. Our experimental results using the TPC-H benchmark and a 32-node cluster show very good speedup.

Keywords Parallel databases · Database clusters · OLAP query processing · Partial replication · Virtual partitioning · Dynamic load balancing

Communicated by Ladjel Bellatreche.

C. Furtado · M. Mattoso
COPPE, Federal University of Rio de Janeiro, Rio de Janeiro, Brazil

C. Furtado
e-mail: camillef@cos.ufrj.br

M. Mattoso
e-mail: marta@cos.ufrj.br

P. Valduriez
INRIA, Nantes, France
e-mail: Patrick.Valduriez@inria.fr

A.A.B. Lima (✉)
School of Science and Technology, Unigranrio University, Rio de Janeiro, Brazil
e-mail: abento@unigranrio.com.br

1 Introduction

OLAP applications demand high-performance from their underlying database systems in order to achieve low response time, which is crucial for decision making. They typically access huge amounts of data through high-cost read-intensive *ad-hoc* queries. For instance, the TPC-H Benchmark [1], which models OLAP applications, specifies 24 queries, including 22 complex high-cost read-intensive queries and only 2 update queries.

High-performance processing in database systems has traditionally been achieved through parallel database systems [2] running on multiprocessor servers. Data are typically partitioned and replicated between multiprocessor nodes for each of them to process queries in parallel over different data subsets. This approach is very efficient, but expensive in terms of hardware and software. Besides, the database management system must have full control over the database fragments, which makes it very expensive to migrate from a non-parallel environment.

Database Clusters (DBC) are a very efficient low-cost alternative to tightly-coupled multiprocessor database systems. A DBC [3] is defined as a cluster of PC each of them running an off-the-shelf sequential DBMS. These DBMS are orchestrated by a middleware that implements parallel query processing techniques. Figure 1 shows a DBC running the PostgreSQL DBMS at each node and our **SmaQSS** DBC middleware. **SmaQSS** implements parallel query processing techniques in a non-intrusive way. It is DBMS-independent, only requiring that the DBMS processes SQL3 queries and accepts connections through a JDBC driver. **SmaQSS** can make database migration from centralized environments to DBC quite easy, depending on the features one intends to explore. In its simplest form (with full database replication), the migration requires just the creation of clustered indexes for the largest tables, as described in the following. More complex features require the implementation of other database design techniques (e.g., partial replication).

There are basically two kinds of parallelism employed during query processing: inter-query and intra-query parallelism. Inter-query parallelism is obtained when different queries are simultaneously processed by different nodes. Intra-query parallelism consists of having multiple nodes processing, at the same time, different operations of the same query. This kind of parallelism is essential for high-performance OLAP application support as it can reduce the cost of heavy queries. Distributed database design is crucial for intra-query parallelism efficiency. It is executed in two

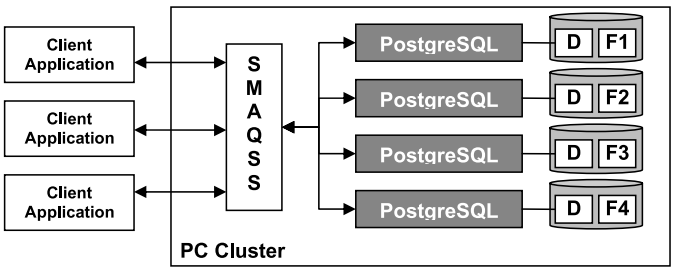

Fig. 1 Database cluster with **SmaQSS** middleware

steps: data fragmentation and allocation [4]. During fragmentation, data partitions are defined and, during allocation, decisions about their replication and placement on the DBC nodes are taken.

Not many DBC systems implement intra-query parallelism. To the best of our knowledge, only PowerDB [5], Apuama [6] and ParGRES [7] do it. However, all these solutions employ full database replication during data allocation and obtain intra-query parallelism by using virtual partitioning [3] during query processing. While in physical data partitioning a table is statically partitioned and physical subsets are generated accordingly, in virtual partitioning dynamic table subsets are defined during query processing. This allows for more flexibility as the partitions may be defined according to each query.

Entirely replicating the database in all cluster nodes demands high disk space, as each node must be able to hold the entire database. Furthermore, it makes data updates very expensive.

In this paper, we propose a distributed database design strategy for DBC based on physical and virtual data partitioning combined with replication techniques. To address the limitation of static physical partitioning we take advantage of replicas and propose a dynamic query load balancing strategy.

We present a query load balancing technique that, combined with an adaptive virtual partitioning technique [8], makes it possible to dynamically redistribute tasks from busy DBC nodes to idle nodes that contain replicas of their data fragments. Thus, our solution makes it possible to obtain intra-query parallelism during heavy query execution with dynamic load balancing while avoiding the overhead of full replication. To the best of our knowledge, this solution is unique in the context of DBC.

In order to evaluate our solution, we implemented a DBC middleware prototype called **SmaQSS** (**Sma**shing **Q**ueries while **S**hrinking disk **S**pace). We performed extensive experiments by running representative TPC-H queries on top of a 32-node cluster from the Grid'5000 experimental platform [9] running the PostgreSQL DBMS [10]. The results show that **SmaQSS** achieves excellent performance in scenarios with and without load unbalancing between nodes, thus showing the effectiveness of our techniques. They also show that good performance can be obtained while avoiding full replication.

This paper is organized as follows. Section 2 shows related work. Section 3 defines Virtual Partitioning. Section 4 presents our distributed database design strategy. In Sect. 5, we present our query processing algorithm along with the proposed load balancing technique. Section 6 shows experimental results and Sect. 7 concludes the paper.

2 Related work

Parallel processing techniques have been successfully employed to achieve high performance in DBC [3, 5–8, 11–15]. Inter-query parallelism aims at achieving high throughput during transaction processing and is very effective for On-Line Transaction Processing (OLTP) applications, characterized by large numbers of small simultaneous transactions. This is the only kind of parallelism explored in [11–13].

For OLAP applications, intra-query parallelism is more appropriate as it reduces the execution time of individual queries [3, 5–8, 14].

For intra-query parallelism efficiency, distributed database design is very important. It is a two-step process [4]: in the first step (data partitioning), data partitions are defined; in the second step (partition allocation), decisions about their replication and placement on the DBC nodes are taken.

There are two basic alternatives to data partitioning in DBC [14]: physical and virtual data partitioning. Each one has already been employed in isolation for intra-query parallelism implementation during OLAP query processing. Physical data partitioning consists of statically defining a table partitioning schema and physically generating table subsets according it. Virtual partitioning consists of dynamically defining table subsets. This approach allows for more flexibility as the partitions may be defined according to each query characteristics and the nodes available for processing.

The distributed data design proposed by Röhm et al. in [5] for OLAP applications has been successfully employed for implementing physical partitioning. For the data partitioning step it horizontally partitions large tables that are most accessed (typically the fact tables), and do not partition small tables (typically the dimension tables). For the allocation step, it replicates all dimension tables at all nodes and distributes the horizontal partitions of all fact tables among the DBC nodes. Such distributed data design is illustrated in Fig. 1, where D represents replicated dimensions and each Fi represents a partition i of a fact table F, where typically the number of partitions equals to the number of the DBC nodes. The partitioning of F is based on its key attributes.

This partitioning design for the fact and dimension tables can be used in both physical and virtual partitioning. However, the allocation design is quite different in each case. When virtual partitioning is employed, some database replication is required. Physical partitioning requires more complex allocation strategies, where full or partial or no replication may be adopted.

Many variations of virtual partitioning with full replication were implemented with very good results [3, 6–8, 14], showing this is a quite efficient alternative to accelerate OLAP query processing. However, full replication is a major issue due to its high cost concerning disk space utilization and data updates.

Experimental results obtained with physical partitioning without any replication also present very good results when the load between nodes was balanced during query processing [5, 15]. In this case, intra-query parallelism can be implemented with great reduction of disk space utilization when compared to full replication. However, in most real scenarios the load is not balanced and query performance results in the presence of skew are very poor [8].

In addition, physical partitioning without replication presents severe limitations. One of them is low availability as, if one table partition becomes unavailable, queries that require access to that table cannot be processed. Another major issue (which we address in this paper) is that dynamic load balancing during query processing cannot be employed without data transfers between nodes, which may also lead to poor performance. The implementation of dynamic load balancing in DBC is not trivial as, due to its DBMS independent nature, it is not possible to interrupt a sequential DBMS to make it to stop processing a running query to redistribute its load using

non-intrusive techniques. Then, high performance in this case strongly relies on a "balanced" static physical database design.

In [16], Furtado proposed a distributed database design strategy for OLAP databases called Node Partitioned Data Warehouses (NPDW). NPDW adopts the approach of replicating dimensions and physically partitioning fact tables. It also presents some strategies for replicating the physical partitions between nodes. The techniques described in NPDW were designed for OLAP databases distributed over low-cost computer nodes, thus it could be easily implemented in a DBC. One major difference between our work and NPDW is that, in that work, replication is applied and analyzed in the terms of system availability, while in our work replication is applied to make it possible to implement dynamic load balancing with no intrusive adaptive virtual partitioning techniques. NPDW does not take advantage of replication to implement dynamic load balancing. In [17], improvements are made to NPDW such as the physical partitioning of large dimensions and the proposal of techniques for query processing. The improvements proposed to NPDW database design strategy take into account a previously known query workload to be processed by the system. Our work focuses on the typical *ad-hoc* nature of OLAP queries, thus no previous assumptions about such queries can be made. Furthermore, the query processing techniques proposed by Furtado in [17] require data repartitioning, which implies data transfers between nodes, which is not the case for our solution. Finally, techniques in [17] assume a load balanced scenario. So, the performance obtained relies on a good initial database design and on the efficiency of data repartitioning techniques.

Works like [18, 19] employ physical partitioning of the dimension tables. Small grain physical partitions have been successfully applied in [18]. However, it requires careful distributed database design to generate physical partitions. Thus, it is more sensitive to skew conditions. Simulation results presented in [18] lack an analysis of the complexity in managing a schema with very large number of partitions. Finally, performance results in the presence of data skew cannot be found in related work for database clusters. In [19], dimension partitions guide the fact table partitioning process and a genetic algorithm is employed to choose the best schema. However, in both works, previous knowledge about OLAP queries submitted to the database system is required. So, *ad-hoc* queries are not considered.

Our approach takes advantage of physical and virtual partitioning with partial replication, which allows for dynamic task redistribution between nodes that keep replicas of the same partition. Such redistribution is made possible through a combination of physical and adaptive virtual data partitioning. We do not use any previous knowledge about query profiles. Our solution is aimed at *ad-hoc* OLAP queries.

3 Query processing with virtual partitioning

In this section, we explain in more detail the concept of virtual partitioning (VP) proposed in [3], and show how queries can be processed by using it. Section 3.1 conceptually defines VP and explains its principles. Section 3.2 describes query rewriting and final result composition for VP considering queries containing aggregate functions. Section 3.3 discuss requirements for obtaining high performance during query

processing when using VP. Section 3.4 shows typical queries that can be parallelized through VP.

3.1 Virtual partitioning definition

We start by defining the concept of virtual partition:

Definition 1 (Virtual Partition) Given a relation R and a predicate P, a virtual partition $R_{VP} = \sigma_P(R)$, where P is of the form $A \in [a_1, a_2)$, A is the partitioning attribute and a_i are values from A's domain, with $a_2 > a_1$.

A virtual partition differs from a physical partition as it is dynamically obtained from a base relation through a relational algebra operation. The same base relation can be virtually partitioned in many different ways without requiring any physical reorganization of its data. This is not true for physical partitioning: different physical partitioning schemes require physical reorganization of the tuples on disk.

We call *virtual partitioning* the operation of producing virtual partitions of a relation. According to Definition 1, it seems that relations can only be horizontally partitioned. However, this is not true. Virtual partitioning can produce horizontal, vertical or hybrid relation partitions [4] from a relation. It suffices to replace the relational select (σ) operation that appears at Definition 1 by the appropriate operation, e.g., the relational project (π) operation (vertical virtual partitioning) or a select operation followed by a project operation (hybrid partitioning). In this work, we only deal with horizontal partitions. For the best of our knowledge, no work tried to use different kinds of virtual partitions, but this is not conceptually impossible.

With virtual partitioning, a query can be processed as follows. Let $Q(R, S)$ be a query submitted to the database cluster, where R is a relation chosen for virtual partitioning and S is a set of other relations accessed by Q. Let n be the chosen number of virtual partitions, n range predicates are computed to produce a virtual partitioning of R: $R_{VP1}, R_{VP2}, \ldots, R_{VPn}$. Then, Q is replaced by a set of sub-queries, each one over a different virtual partition of R: $Q_1(R_{VP1}, S), Q_2(_{RVP2}, S), \ldots, Q_n(R_{VPn}, S)$. In order to avoid wrong results, the virtual partitioning of R is required to be disjoint and complete. The virtual partitioning of a relation R is said to be *disjoint* if, and only if, $\forall i, j$ $(i \neq j, 1 \leq i, j \leq n)$, $R_{VPi} \cap R_{VPj} = \varnothing$. It is said to be *complete* if, and only if, $\forall r \in R, \exists R_{VPi}, 1 \leq i \leq n$, where $r \in R_{VPi}$. We use here the same terminology found in the literature for physical partitioning (or fragmentation) [4].

Obtaining the final result of a query processed by using virtual partitioning requires more than just executing its sub-queries. Let $Res(Q)$ be the result of query Q, $Res(Q)$ must be obtained by the composition of the results of each sub-query Q_i, $i = 1, \ldots, n$. In other words, $Res(Q) = \nabla Res(Q_i)$, $i = 1, \ldots, n$, where the operation ∇ varies from a simple relational union ($\cup$) operation to a set of complex operations, according to Q.

Let us give an example using SQL. Consider a database with a relation that has the following schema: T (*tpk*, *tprice*, *tqty*, *tname*). Suppose that this database is fully replicated over four nodes in a database cluster. Also, suppose the following *ad-hoc* query Q is submitted to the cluster:

Q: select max(tprice) **from** T;

We want to process Q by using virtual partitioning. Then, T must be virtually partitioned and we will choose *tpk* as its partitioning attribute. According to Definition 1, T should be replaced in each sub-query Q_i by the virtual partition $T_{VP} = \sigma_{(tpk>=:v1) \text{ and } (tpk<:v2)}(T)$, the values of :v1 and :v2 varying according to the sub-query. In SQL, it means that each Q_i will be written as follows:

Q$_i$: **select max**(tprice) **from** T
where *tpk* >= :v1 **and** *tpk* < :v2;

The parameters :v1 and :v2 that appear in Qi will receive different values for each virtual partition. Suppose, for example, that *tpk* values range from 1 to 10,000. We could produce four disjoint virtual partitions of T through four sub-queries and give the following values for :v1 and :v2 in each of them: Q1 (:v1=1; :v2=2,501), Q2 (:v1=2,501; :v2=5,001), Q3 (:v1=5,001; :v2=7,501) and Q4 (:v1=7,501; :v2=10,001). Here, we simply divided the value range of the partitioning attribute by the number of desired virtual partitions, producing equal-sized virtual partitions (considering interval sizes). As our hypothetical cluster has four nodes, we could simply give one sub-query to each one and pick the highest result as the response to Q. The number of virtual partitions from a table does not have to be the same of query processing nodes. This is done in PowerDB [3] and Apuama [6]. In such database clusters, the number of virtual partitions produced for a query is always equal to the number of cluster nodes available to execute them. If n nodes are available, n virtual partitions, and consequently n sub-queries are produced. Then, each node processes only one sub-query. We call this approach *Simple Virtual Partitioning* (SVP). It has some drawbacks, as we explain later. For this reason, we adopt in **SmaQSS** another VP approach that produces a number of virtual partitions greater than the number of nodes, which is an improved version of *Adaptive Virtual Partitioning* (AVP), we originally proposed in [8]. The AVP algorithm was slightly modified to achieve better performance. It is described on Sect. 5.

This explains the basic principles of VP. However, there are some details that must be considered when more complex queries have to be processed. In the next section, we discuss them by showing how to use VP to process queries that involve aggregate function, commonly used in OLAP applications.

3.2 Using VP to process queries with aggregate functions

Processing a query Q by using Virtual Partitioning involves replacing Q by a set of sub-queries Q_i. The sub-queries are very similar to each other, the only difference being the interval that determines the virtual partitions. However, in many cases, the sub-queries cannot be similar to the original query or wrong results would be produced. So, a rewriting process must be executed by the database cluster query processor in order to obtain correct sub-queries. In such cases, this rewriting can be more complex than just adding the range predicate required for virtual partitioning. Also, additional operations may have to be performed for the correct query result to be produced.

In this section, we discuss query rewriting and result composition for queries that perform aggregate functions, largely employed by decision support systems. Similar

Table 1 Obtaining the result of aggregate functions from sub-multisets

Function	Result composition from MS sub-multisets
MAX(MS)	MAX ({ MAX(MS_1), MAX(MS_2), ..., MAX(MS_m) })
MIN(MS)	MIN ({ MIN(MS_1), MIN(MS_2), ..., MIN(MS_m) })
SUM(MS)	SUM ({ SUM(MS_1), SUM(MS_2), ..., SUM(MS_m) })
COUNT(MS)	SUM ({ COUNT(MS_1), COUNT(MS_2), ..., COUNT(MS_m) })
AVG(MS)	$\frac{\text{SUM}(\{\text{SUM}(MS_1), \text{SUM}(MS_2), \ldots, \text{SUM}(MS_m)\})}{\text{SUM}(\{\text{COUNT}(MS_1), \text{COUNT}(MS_2), \ldots, \text{COUNT}(MS_m)\})}$

processes must also be employed for some queries that present no aggregate functions, but we leave them outside our discussion.

The main SQL aggregate functions are: MAX, MIN, SUM, AVG and COUNT, and we keep our focus onto them. All these functions are of the form Agg(MS), where MS is a multiset of values obtained from an expression based on attributes of the relation tuples that satisfy a query predicate. If MS is a multiset, we can split its elements in such a way that $\text{MS} = \text{MS}_1 \cup \text{MS}_2 \cup \cdots \cup \text{MS}_m$, where each MS_j, $j = 1, \ldots, m$, is also a multiset. If we have a query "**select max**(A) **from** T" and, in order to process it by using VP, we break it into sub-queries of the form "**select max**(A) **from** T **where** att >= :v1 and att < :v2", we are not directly applying the aggregate function MAX over the whole multiset of values associated to the attribute A of T. Indeed, we are applying the MAX function many times, one for each sub-multiset of the values associated to A, as each virtual partition contains only a subset of the tuples of T. The main issue is how to obtain the correct result for an aggregate function Agg(MS) from the sub-multisets $\text{MS}_1, \text{MS}_2, \ldots, \text{MS}_m$. Table 1 shows how it can be done for the aggregation functions considered here. In the table, the notation "{e1, e2, ..., e_m}" represents a multiset.

Table 1 shows that the result of an SQL aggregation function Agg(MS) can be obtained by applying one or more different functions (also from SQL) over each MS_i. Let us see how it can be used to process queries with virtual partitioning.

Let $\tilde{A}$ be the multiset of values associated to attribute A on a relation R. For a query $Q(R, S)$ to be processed by using virtual partitioning, a set of sub-queries $Q_i(R_{VPi}, S)$ will be produced. Each Q_i will access a sub-multiset of $\tilde{A}$ that we designate as $\tilde{A}_{VPi}$. Then, if Q includes an aggregation function $Agg(\tilde{A})$, the result of this function will have to be calculated from each $\tilde{A}_{VPi}$. Since $\tilde{A}$ is a multiset, we can apply the rules from Table 1 to obtain the correct results. This is shown by Table 2.

The aggregate functions query Q must be translated in order to produce sub-queries. The translation rules are given by Table 3.

To illustrate aggregate query rewriting, let us consider the following query:

QAgg: select max(A) as ca, **min**(B) as cb, **sum**(C) as cc,
count(D) as cd, **avg**(E) as ce
from R;

QAgg is to be processed by using VP. According the translation rules from Table 3, and considering PK the partitioning attribute of R, the sub-queries of QAgg will be written as follows:

Table 2 Result composition of aggregate functions using virtual partitions

Function	Result composition from $\tilde{A}$ sub-multisets
MAX($\tilde{A}$)	MAX ({ MAX($\tilde{A}_{VP1}$), MAX($\tilde{A}_{VP2}$), …, MAX($\tilde{A}_{VPm}$) })
MIN($\tilde{A}$)	MIN ({ MIN($\tilde{A}_{VP1}$), MIN($\tilde{A}_{VP2}$), …, MIN($\tilde{A}_{VPm}$) })
SUM($\tilde{A}$)	SUM ({ SUM($\tilde{A}_{VP1}$), SUM($\tilde{A}_{VP2}$), …, SUM($\tilde{A}_{VPm}$) })
COUNT($\tilde{A}$)	SUM ({ COUNT($\tilde{A}_{VP1}$), COUNT($\tilde{A}_{VP2}$), …, COUNT($\tilde{A}_{VPm}$) })
AVG($\tilde{A}$)	$\frac{\text{SUM}(\{\text{SUM}(\tilde{A}_{VP1}), \text{SUM}(\tilde{A}_{VP2}), \ldots, \text{SUM}(\tilde{A}_{VPm})\})}{\text{SUM}(\{\text{COUNT}(\tilde{A}_{VP1}), \text{COUNT}(\tilde{A}_{VP2}), \ldots, \text{COUNT}(\tilde{A}_{VPm})\})}$

Table 3 Translation rules for aggregate functions in VP

Aggregation function in Q	Corresponding aggregate function(s) in Q_i
MAX(A)	MAX(A)
MIN(A)	MIN(A)
SUM(A)	SUM(A)
COUNT(A)	COUNT(A)
AVG(A)	SUM(A), COUNT(A)

QAgg$_i$: select max(A) **as** ica, **min**(B) **as** icb, **sum**(C) **as** icc,
count(D) **as** icd, **sum**(E) **as** ice1, **count**(E) **as** ice2
from R
where PK >= :v1 **and** PK < :v2;

The final result will be produced according to the rules shown by Table 2, which can be easily implemented in SQL.

Usually, aggregate functions are used in conjunction with grouping operations (GROUP BY clause, in SQL). It does not affect the query rewriting process described so far. The grouping attributes that appear in Q just have to be repeated in each Q_i. However, for the final results to be externally composed, the database cluster middleware must have to be able to store groups and their corresponding partial results.

Other SQL operators like UNION, INTERSECTS, MINUS and ORDER BY must have to be externally implemented by the database cluster middleware but they are beyond the scope of this paper.

3.3 Virtual partitioning requirements for obtaining high-performance

Virtual Partitioning is a technique that can be employed to process a query even when only one processor is available: it suffices to make the same processor to execute all sub-queries. However, this is not very useful as modern sequential DBMS can do a better job by processing the entire query once. Virtual partitioning is an attractive approach in multi-processor systems, where it can be employed in order to obtain intra-query parallelism during query processing. However, even in parallel environments, simply using virtual partitioning does not guarantee high performance for database query processing. There are two requirements for this technique to achieve good results.

The first requirement is that each cluster node must physically access (or retrieve from the disk) only the tuples that belong to the virtual partition determined by its sub-query. Otherwise, the main goal of virtual partitioning may not be achieved and the performance can be severely hurt. This restriction cannot be guaranteed by the simple addition of a range predicate to the original query. The tuples of a virtual partition can be spread over many different disk pages that stores tuples of the virtually partitioned relation. They can even be mixed up with tuples from other virtual partitions. In the worst case, there can be tuples of a single virtual partition in all disk pages that store data from the virtually partitioned relation. In this case, the node responsible for processing such virtual partition would have to read the entire relation from the disk.

To solve this problem (or to diminish its effects), the tuples of the relation that is intended to be virtually partitioned must be physically ordered in the disk according to the partitioning attribute. This can be achieved by constructing a *clustered index* based on the partitioning attribute to the relation that is intended to be virtually partitioned, as indicated in [3]. If the tuples are clustered this way, a cluster node can use the index to find the first tuple of its virtual partition, and all the other tuples that must be accessed will be in logically contiguous disk pages. This way, the node almost exclusively reads from disk tuples that belong to its virtual partition. The exceptions are the first and the last disk pages that contain its tuples. If the node is processing the virtual partition R_{VPi} from relation R, the first page it accesses may contain tuples from $R_{VP(i-1)}$ and the last page may contain tuples from $R_{VP(i+1)}$.

The second requirement for virtual partitioning to achieve high performance is that the DBMS must effectively use the clustered index to access the virtually partitioned table when a cluster node is processing its sub-query. Many of the modern DBMS query optimizers estimate the number of tuples from a relation that must be retrieved by a query before deciding to use an index or not. If, for example, a query has a range predicate based on a attribute a of a relation R, and the DBMS query optimizer estimates that the number of tuples that must be retrieved is larger than a certain threshold (that varies according to the DBMS), it will decide to perform a full scan over the relation even if a clustered index based on a is associated to R. If one node performs a full relation scan, the virtual partitioning goal of restricting each node to its own partition is compromised because the cost of accessing disk pages is usually high. This happened when we re-executed the experiments done for testing SVP in PowerDB [3] using the DBMS PostgreSQL 7.3.4. In many cases, for different queries using different numbers of nodes, the DBMS performed full scans over the virtually partitioned relations, severely hurting the performance. It happened mostly when the number of nodes was low, in which case the size of each virtual partition is high. Apparently, it did not happen for PowerDB because the DBMS employed during the experiments always used the clustered index to access the virtually partitioned table. But we cannot guarantee that all DBMS will behave this way.

We conclude SVP solely depends on the underlying DBMS to achieve good performance. As our goal is to build a high performance database cluster middleware that is independent from the DBMS, SVP seems not to be a good approach to be adopted. For this reason, we adopt in **SmaQSS** the AVP technique, which we originally proposed in [8]. Briefly, it subdivides the virtual partitioning interval of a sub-query into smaller intervals, generating many sub-queries in each node. Such sub-queries can

be dynamically reallocated during query processing in order to achieve load redistribution. In **SmaQSS**, we take advantage of its characteristics in order to achieve dynamic load balancing during query processing. As a last remark, Apuama achieves good performance using SVP and PostgreSQL, but the use of the clustered index during query processing is forced by the middleware through hints that are sent to the DBMS query optimizer. These hints go against our goal of using the DBMS as a black-box component and require the middleware to be properly re-configured when used with a different DBMS.

3.4 Query requirements for using virtual partitioning

Not all OLAP queries can be processed by using VP. In [3], a query classification is proposed, and we give a brief description of it here:

- **Class 1**: Queries that access a fact table and that do not present sub-queries
  - Queries that exclusively access the fact table;
  - Queries that access one fact table and references to other tables;
  - Queries that access more than one fact table, perform only equi-joins and have a cycle-free query graph.
- **Class 2**: Queries that present a sub-query and that are equivalent to Class 1 queries.
- **Class 3**: All other queries.

Queries from classes 1 and 2 can be parallelized by using VP. It does not happen with queries from class 3. A more detailed discussion about this can be found in [3].

4 Distributed database design for OLAP in database clusters

A DBC is a cluster of PC, each running an off-the-shelf sequential DBMS. These DBMS are orchestrated by a middleware that implements parallel query processing techniques. Sequential DBMS do not implement any parallel query processing technique. Furthermore, on a DBC, such DBMS are employed as "black-box" components, which means that it is assumed that their source codes are node available and, consequently, they cannot be altered to include parallel processing capabilities. Thus, DBC are DBMS independent, which eases application migration from sequential environments to clusters. However, all parallel query processing techniques must be implemented and coordinated by the middleware that runs on the DBC, as illustrated in Fig. 1.

Distributed database design for DBC has some special characteristics because partition management is performed by the middleware, not the DBMS. In this section, we first describe our solution to data partitioning design and then our approach to allocation design.

4.1 Partitioning design

Most OLAP queries access at least one fact table and are typically heavy-weight as they process huge amounts of data which may take a long time to execute. This

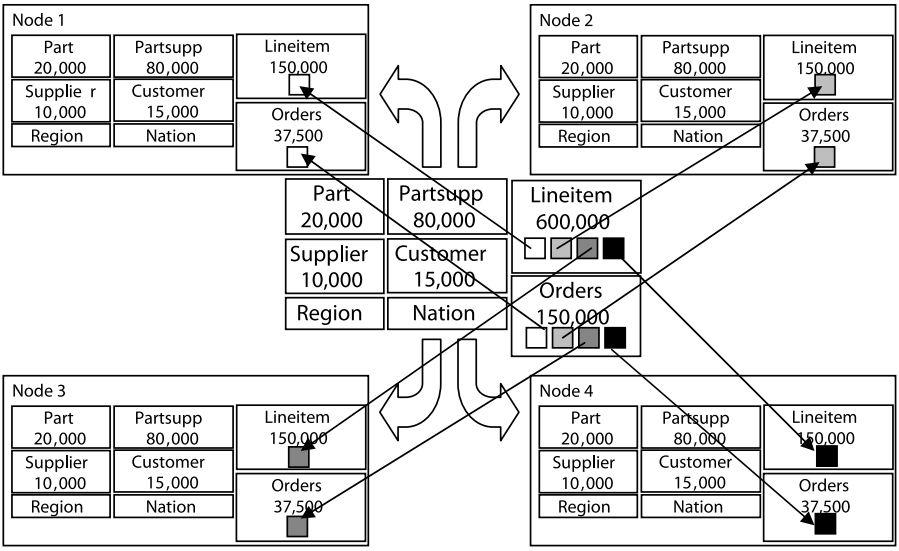

Fig. 2 Horizontally partitioning the Lineitem fact table

is why intra-query parallelism fits well for such queries. Considering the traditional characteristics of database schemas for OLAP applications, we propose the use of the distributed database design strategy proposed by [5]. Such a strategy horizontally partitions the fact tables, which are the larger ones. As in most parallel distributed design, the partitioning function is based on a primary key attribute and generates value ranges for the attribute. The number of ranges equals the number of DBC nodes. On the other hand, partitioning is not applied to dimensions, which are fully replicated on all DBC nodes. Figure 2 illustrates the application of such a design strategy over the database defined by the TPC-H benchmark [1] on a 4-node DBC.

TPC-H has a slightly different OLAP schema with two fact tables: Orders and Lineitem, which are linked by a foreign key. Primary horizontal partitioning is applied over the Orders fact table and its primary key (o_orderkey) is used as the partitioning attribute. The fact table Lineitem has a foreign key (l_orderkey) to Orders. This foreign key is also the first attribute of its primary key. Then, derived horizontal partitioning is applied over Lineitem, originating almost equal-sized partitions. Because of this partitioning schema, we can always physically allocate related partitions from Orders and Lineitem to the same DBC node, easing the join process between these tables. As in TPC-H, we consider dimensions are small when compared to fact tables. This is why we opt for fully replicating them, which eases the employment of virtual partitioning. Dimension partitioning is dealt with in works like [18, 19] but it is out of the scope of this paper. Figure 2 shows the partitioning schema and the allocation strategy we adopt.

4.2 Partition allocation through chained declustering

Our proposal is to offer an alternative to either full replication or no replication. We adopt partial replication of data partitions by adapting the technique called Chained

#Node	1	2	3	4	5	6	7	8
Primary Copy	$\mathbf{R_P(1)}$	$R_P(2)$	$R_P(3)$	$R_P(4)$	$R_P(5)$	$R_P(6)$	$R_P(7)$	$R_P(8)$
Backup Copies	$R_B(8)$	$\mathbf{R_B(1)}$	$R_B(2)$	$R_B(3)$	$R_B(4)$	$R_B(5)$	$R_B(6)$	$R_B(7)$
	$R_B(7)$	$R_B(8)$	$\mathbf{R_B(1)}$	$R_B(2)$	$R_B(3)$	$R_B(4)$	$R_B(5)$	$R_B(6)$
	$R_B(6)$	$R_B(7)$	$R_B(8)$	$\mathbf{R_B(1)}$	$R_B(2)$	$R_B(3)$	$R_B(4)$	$R_B(5)$

Fig. 3 Allocation of R partitions having 3 backup copies

Declustering [20]. This strategy was originally proposed to provide high availability to data stored on a set of disks. However, as there are data partition replicas, when a node is overloaded or fails, load redistribution can be performed by the reassignment of tasks that access some data partition to other nodes that contain replicas of it. We adapt chained declustering to our context in order to implement dynamic load balancing during query processing. However, fault tolerance is out of the scope of this paper.

Chained declustering distributes data partitions among DBC nodes using the following strategy. Let us take a DBC with M nodes, numbered from 1 to M, and a relation R, divided into R_i partitions, $i = 1$ to M. For each partition R_i, its primary copy is stored on node $\{[i - 1 + C(R)] \bmod M + 1\}$. Besides, a backup copy (replica) of R_i is stored on node $\{[i + C(R)] \bmod M + 1\}$, which guarantees that the primary copy and its backup are stored on different nodes. The $C(R)$ function defines the node on which the first R partition must be stored, thus allowing the first partition to be stored on any of the M nodes.

The expression "chained declustered" means that nodes are linked just as if they formed a chain. Figure 3 illustrates the allocation strategy of 8 partitions of relation R, each one having 3 replicas, on 8 nodes and $C(R) = 0$. $R_P(i)$ and $R_B(i)$ denote the primary and the backup copy of the i-th R partition, respectively. It is possible to have until $M - 1$ backups of each primary copy and store each of them on a different node. As each node has replicas of partitions that are stored in other nodes, if a node becomes overloaded, its tasks can be redistributed to others that have replicas of its partitions. This makes chained declustering a very attractive allocation strategy as it allows for dynamic load balancing during query processing.

The number of replicas depends on the user needs. As more replicas are used, the cost of updates and the disk space consumption increase. However, as seen in next section, the dynamic load balancing process becomes more efficient. In OLAP, updates are typically reduced to inserts and often decision can be made based on history rather than last minute changes. Thus, if data freshness may be relaxed, then update costs would decrease. This is a tradeoff that must be analyzed by the database administrator. We can suggest two possible approaches. In the first one, the administrator chooses the number of replicas based on his/her experience and knowledge of the data warehouse. In the second approach, experiments should be conducted, based on the available disk space and on the number of updates, in order to identify the best number of replicas.

5 Query processing with dynamic load balancing

Load unbalancing during query processing may be caused by many different reasons. One of the factors that determine the workload of nodes is the number of tuples it has to process. Even if table partitions are equal-sized between nodes, operations (e.g. selects and joins) performed during query processing can generate intermediate results with different sizes, which can lead to a kind of load unbalancing that is very hard to preview during distributed database design, mainly for *ad-hoc* queries. Most OLAP queries access at least one fact table. Let us take a scenario where fact tables are fully partitioned and allocated without any replication. If, during query processing, one node becomes overloaded, there is no way to redistribute its load without data transfer as there are no replicas from the data partitions it holds.

It is very complex to prevent and solve load balancing problems. One possible solution is to transfer data between nodes during query processing. However, such an operation tends to be very expensive, requiring extra I/O operations and rising communication costs. Thus, in order to avoid data transfers, our proposal is to perform query processing with dynamic load balancing over a distributed data set (fully or partially) replicated with chained declustering between cluster nodes. We call this approach **QueCh** (**Que**ry processing with **Ch**ained declustering replication). **QueCh** is implemented in **SmaQSS**.

QueCh query processing is a three-step process. First, the initial query that accesses the fact table is mapped to sub-queries that access its physical partitions. In the second step, each DBC node locally executes its sub-query over its partition. However, during this phase, the physical table partition is not entirely processed by only one sub-query. Instead, **QueCh** virtually re-partitions it and executes the sub-query once for each virtual partition. This way, it is possible to redistribute sub-queries for dynamic load balancing. The third step consists of sub-query reallocation, if a node becomes idle. Load balancing is performed by dynamically reallocating sub-queries from busy nodes to idle nodes that contain replicas of their partitions. The virtual partitioning technique makes it possible to dynamically reallocate sub-queries without requiring intrusive operations. In the following sections, we explain each step in more details.

5.1 Query splitting

The partitioning schema managed by **SmaQSS** is totally transparent to the user, whose queries are written according to the non-partitioned global database schema. After receiving the query, **SmaQSS** rewrites it considering the partitioning schema implemented. Such a schema is defined in Sect. 3 and consists of dividing the value range of the partitioning attribute in equal-sized ranges according to the number of DBC nodes that hold the fact table accessed.

As an example, let us consider the following query Q submitted to a 4-node DBC:

Q: select max(F.a), **min**(F.b)
from F, D
where F.dfk = D.dpk
and D.x = 1;

Q performs an equijoin between the fact table F and the dimension D. For 4 nodes, Q will be rewritten into 4 sub-queries $Q0$, $Q1$, $Q2$ and $Q3$, where each Qi replaces the fact table F by the primary copy of the partition $F_P(i)$ stored on node i. Suppose the partitioning attribute of F is *fpk* and its values range from 1 to 4,000,000. Let us also suppose that each partition $F_P(i)$ is equal-sized with respect to the range values of *fpk*. Then, for $F_P(0)$, *fpk* ranges from 1 to 1,000,000; for $F_P(1)$, it ranges from 1,000,001 to 2,000,000; and so on. This way, the 4 sub-queries will be written to correspond to the 4 $F_P(i)$ partitions as follows:

Q0: select max(F.a), **min**(F.b)
from F, D
where F.dfk = D.dpk
and D.x = 1
and *F.fpk* >= 1 ***and*** *F.fpk* < 1000001;

Q1: **select max**(F.a), **min**(F.b)
from F, D
where F.dfk = D.dpk
and D.x = 1
and *F.fpk* >= 1000001 ***and*** *F.fpk* < 2000001;

Q2: select max(F.a), **min**(F.b)
from F, D
where F.dfk = D.dpk
and D.x = 1
and *F.fpk* >= 2000001 ***and*** *F.fpk* < 3000001;

Q3: select max(F.a), **min**(F.b)
from F, D
where F.dfk = D.dpk
and D.x = 1
and *F.fpk* >= 3000001 ***and*** *F.fpk* < 4000001;

We can see this first phase is similar to what is done in SVP with respect to the number of virtual partitions produced.

5.2 Sub-query processing

Once received by its DBC node i, each sub-query is locally re-written to process smaller ranges of $F_P(i)$ according to AVP. AVP continuously divides the $F_P(i)$ range into smaller ones and executes one different sub-query for each of them. As an example, let us take sub-query *Q2* assigned to run on node 2 that has the corresponding primary copy of partition $F_P(2)$. Its *fpk* range [2,000,001; 3,000,000] is subdivided by AVP producing new sub-queries locally processed by node 2. The *fpk* sub-range covered by each sub-query is determined by AVP (more detail below). Then, if the first range size is 1024, for example, the first *Q2* sub-query is:

Q2$_1$: **select max**(F.a), **min**(F.b)
from F, D
where F.dfk = D.dpk
and D.x = 1
and *F.fpk* >= 2000001 ***and*** *F.fpk* < 2001025;

If AVP determines the second range size is 2048, then the next *Q2* sub-query is

Q2$_2$: **select max**(F.a), **min**(F.b)
from F, D
where F.dfk = D.dpk
and D.x = 1
and *F.fpk* >= 2001025 ***and*** *F.fpk* < 2003073;

The process continues until the entire range in $F_P(2)$ is processed.

The use of AVP makes it feasible to perform dynamic load balancing by using non-intrusive techniques, i.e., treating the DBMS as a "black-box" component. This way, when a node becomes idle, part of the non-processed range of a busy node can be assigned to the idle node, if it contains a replica of the busy node partition. Let us suppose node 3 becomes idle and that it contains $F_B(2)$, a backup replica of $F_P(2)$. Let us also suppose node 2 has a heavy join processing between $F_P(2)$ and D and still needs to process tuples with *fpk* ranging from 2,500,000 to 3,000,000. **SmaQSS** divides the remaining range into two equal parts and assigns the second one to node 3. This way, node 2 will continue processing tuples with *fpk* ranging from 2,500,000 to 2,750,000, and node 3 will start processing tuples with *fpk* ranging from 2,750,001 to 3,000,000 on its local $F_B(2)$. The busy node (node 2) keeps the first half of the remaining interval in order to take advantage of its database cache. If node 2 becomes idle before node 3, the process is done again, this way dividing the remaining interval in node 3 and assigning half of it to node 2. This is done without any data movement between nodes.

On our experiments, very often this reassignment of ranges had to be done to balance the load between nodes. In this example, we mentioned we assumed equal-sized partitions for the 4 initial ranges. However, this does not mean uniform node processing time. Nodes processing time may vary according to the selectivity of predicate "D.x = 1" on each partition. This becomes even worse when join selectivity between F and D varies according to each *dfk* join attribute value distribution. Indeed, the presence of skew in distributed and parallel query processing is almost bound to happen. Thus, solutions without load balancing or based on heavy data transfer are not as efficient.

Figure 4 shows how AVP divides the local value range of the partitioning attribute in a 4 node DBC. The initial query Q is divided into: $Q0$, $Q1$, $Q2$ and $Q3$. Then, the interval in each Qi, $0 <= i <= 3$, is divided again into smaller intervals I_{ij}. In this example, $1 <= j <= 6$, but this number can be different for each node. Each interval I_{ij} is sequentially processed by one sub-query Q_{ij} at each node. If some node finishes processing its j sub-queries and there are busy nodes with replicas of its local partitions, it can help them by taking part of their non-processed intervals, as explained above.

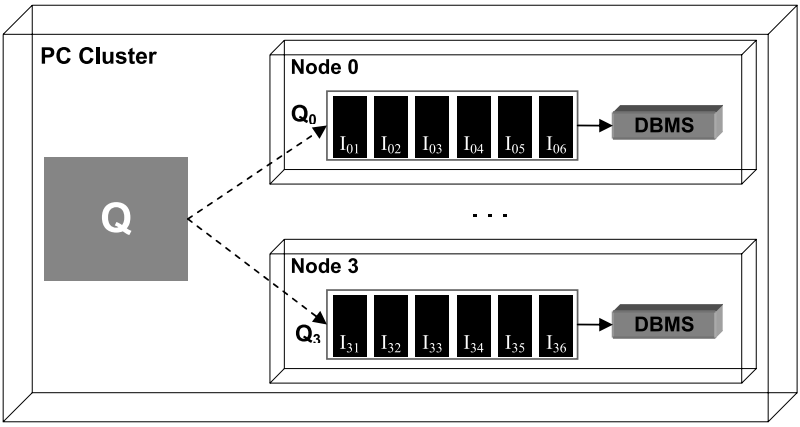

Fig. 4 Overview of AVP on a 4-node DBC

One characteristic of AVP is that the intervals I_{ij} are not equal-sized. We start trying the idea of working with small virtual partitions in [14]. We used pre-defined fixed-size small virtual partitions and obtained very good results. However, the main problem not addressed in [14] is determining the size for such partitions. Each query used during the experiments described in [14] employed a different partition size to obtain the best performance. Those sizes were empirically obtained from many tests we manually performed. So, using small virtual partitions turned out to be a good approach, but we needed a way to automatically find virtual partition sizes that fit to different kinds of queries and different node processing load.

The main goal of AVP is to automatically and dynamically adapt virtual partition sizes whatever the sub-query to be processed, avoiding full scans and reducing the time required to entirely process the initial workload. Different queries may benefit from different virtual partition sizes. The same happens for the sub-queries. For this reason, I_{ij} intervals may not be equal-sized and each node runs the AVP algorithm locally. As AVP does not "know" *a priori* the best size for each sub-query, it makes some tries. It keeps varying the size in order to obtain good performance. The algorithm is described in Fig. 5. Let us now show AVP basic principles and explain how the algorithm works.

AVP starts with a small interval size. This way, it can avoid full relation scans at the very beginning of the process. Otherwise, the threshold after which the DBMS abandon clustered indexes and starts performing full relation scans should be precisely known. AVP sends to the DBMS a sub-query using such very small size and keep track of the execution time. After first sub-query completion, the partition size is increased. In our implementation, we double the size. A new sub-query is then executed and the execution time is measured. Not all query processing phases executed by a DBMS have their execution times determined by the amount of data to be processed. Query parsing and optimization, for example, depend on query statement complexity. So, if the number of sub-queries produced is huge, the amount of time spent on processing these phases can hurt the performance. By keeping track of sub-query execution times, AVP can roughly estimate after which point such query

```
avp( vpInterval : in )
vpInterval        {virtual partition interval}
local variables:
partSize,newSize  {virtual partition sizes used for
                   sub-querysub-query execution}
sizeIncrFactor    {partition size increase factor}
detExecCounter    {execution counter to detect
                   performance deterioration}
deter             {boolean variable to indicate
                   performance deterioration}
begin
  partSize ← initial partition size;
  sizeIncrFactor ← initial size increase factor;
  while (vpInterval not processed)
    { Beginning of Searching phase }
    execute sub-querysub-query with partSize
      keeping execution time;
    newSize ← partSize * (1.0 + sizeIncrFactor);
    execute sub-querysub-query with newSize
      keeping execution time;
    if (increase on exec time <= tolerated increase)
      partSize ← newSize; {keep increasing size}
    else {stop increasing size - end of Searching phase }
      { Beginning of Monitoring phase }
      deter ← FALSE;
      detExecCounter ← 0;
      while ((vpInterval not processed) and
             (not deter))
        execute sub-querysub-query with partSize keeping
                exec time;
        if (increase on exec time > tolerated)
          detExecCount ← detExecCounter + 1;
          if (detExecCount > limit for bad execs)
            deter ← TRUE; {deterioration!}
            decrease partSize;
            sizeIncrFactor ← factor for new size searching;
            { End of Monitoring phase }
          else
            detExecCounter ← 0;
end;
```

Fig. 5 AVP algorithm

processing phases do not significantly influence on total query execution time. If, for example, it doubles the partition size and gets an execution time that is almost twice the previous one, such point was found. Thus AVP stops increasing the size. The process of increasing virtual partition sizes trying to find a "good" one is called the *searching* phase of the algorithm.

The second phase of the algorithm is called *monitoring* phase. During this phase, AVP executes sub-queries using the partition size found during the *searching* phase and keeps track of their execution times. This is done because the partition size in use may not be ideal to process the whole virtual partition that must be processed by the node. System performance can deteriorate due to DBMS data cache misses, overall system load increase, and/or data skew, i.e., the number of tuples associated to different equal-sized intervals may vary (non-uniform data distribution). AVP determines that performance is deteriorating when consecutive sub-query executions with higher execution times are obtained. It may happen that the partition size obtained during the *searching* phase is too large and has benefited from previous data cache hits, i.e., the algorithm could have found a local optimum. Due to its non-intrusive nature, it is very hard to AVP to find a global optimum, i.e., a virtual partition size that can be used to process the whole original interval with the best performance. In this case, it may be better to shrink it. It gives AVP a chance to go back and inspect smaller partition sizes. On the other hand, if performance deterioration was due to a casual and temporary increase on system load or to data cache misses, keeping a small partition size can lead to poor performance. To avoid such situation, after shrinking the virtual partition size, AVP goes back to the searching phase and restarts increasing sizes. The size increase factor used in this new size increasing operation is not necessarily the same used on the beginning of the algorithm. We employ a smaller factor because we assume the first search has probably led the algorithm to an adequate partition size.

From the description above, we can notice that AVP has many parameters that must be set, like the initial virtual partition size, the size increasing factor used during the *searching* phase, and so on. We still did not develop a way for automatically

Table 4 AVP parameters

Parameter	Value
Initial partition size	1024
Initial size increase factor	100%
Tolerated increase in execution time during *searching* phase	25% * sizeIncrFactor
Tolerated increase in execution time during *monitoring* phase	10%
Number of consecutive bad executions during *monitoring* phase to conclude there is performance deterioration	3
Reduction on partition size when performance deterioration is detected	5%
Size increase factor after performance deterioration (new *searching* phase)	20%

setting them up. During AVP development, we tried many different values, refined them and chose the ones showed by Table 4, with which AVP showed the best performance we could obtain. Such values were employed during the experiments described in Sect. 6.

5.3 Dynamic load balancing

The dynamic load balancing technique implemented by **QueCh** is fully decentralized. Each node independently and asynchronously executes the technique. When a node finishes processing its sub-queries, it sends "help offering" messages to other nodes. This way, according to the taxonomy of [21], we can classify the technique as "event oriented", "receiving" and "based on transfer" technique.

The "help offer" is limited by the number of replicas defined by the allocation schema. When we define the set of nodes that can exchange messages, we are employing a neighborhood based technique. The neighborhood is determined by the partition replica chain defined by chained declustering, as shown in Fig. 6. Such limitation reduces the communication costs between nodes.

The load balancing algorithm is quite simple. The help offering node sequentially sends messages to all nodes in its neighborhood. It follows the replica chain defined by chained declustering, sending the first message to the node that contains the value range nearest to its. Thus, due to this proximity, this node is probably the one that will receive help.

If the receiving node is also idle, it simply ignores the message. Otherwise, it sends a help acceptance message back to the offering node. This is done for every help offering message received as there are no guarantees that the node will still be idle when the help acceptance message arrives. The help offering node remains idle until the first help acceptance message arrives.

According to the number of partition replicas it holds, a node can offer help to more than one neighbor. Consequently, it may receive help acceptance messages from more than one node. In such a case, those messages are queued according to some queuing policy, i.e., a different value is associated to each of them and they will be queued and processed according to the order determined by the adopted policy. As

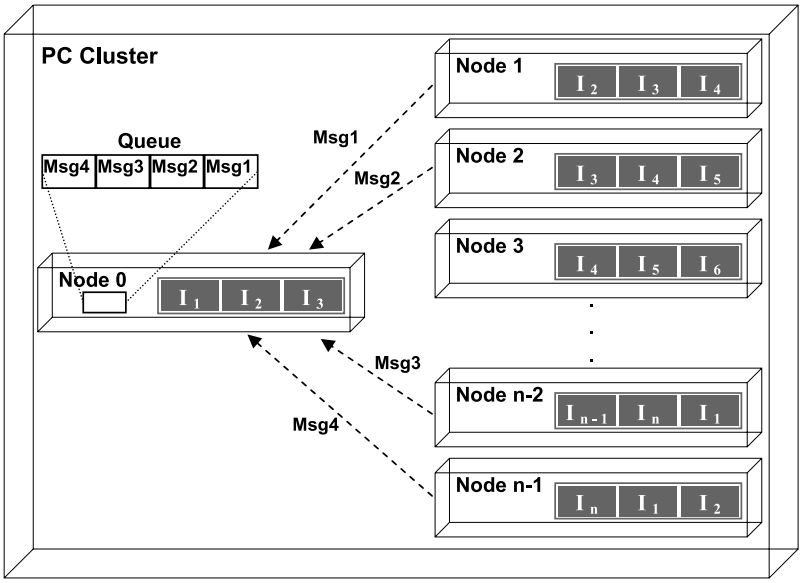

Fig. 6 Help offering messages during load balancing

our main goal in this work is not to make an exhaustive study about dynamic load balancing, we implemented a simple policy and performed our experiments using it. We employ the "Last In First Out" (LIFO) policy, which takes into account the order under which each message arrives at the help offering node. Messages are then decreasingly ordered and the first node to receive help is the one whose help acceptance message last arrived at the offering node. It is a simple technique that provided very good results. More detailed experiments with other policies (such as "First In First Out" (FIFO), "Most Workloaded First" (MWLF) and "Least Workloaded First" (LWLF)) should be done in the future.

The load balancing technique takes the first message from the queue and checks if the node which sent it still needs help. If not (which occurs if the message stands a long time in the queue), the message is discarded and the next one is taken. If, otherwise, the node still needs help, the offering node initiates the processes by asking the busy node an interval to be processed. After receiving it, the offering node starts processing the interval on its partition replica. During this process, new help acceptance messages may keep coming. In such a case, the offering node keeps queuing them. When the offering node finishes processing its new interval, it takes another message from the queue and the whole process restarts. If the queue becomes empty, the offering node sends help offering messages to its neighbors again. Each node stores its sub-query results. When all nodes are done and the whole process is finished, the results are sent to the coordinator for final result composition.

The existence of help acceptance messages from nodes that already finished their tasks causes a waste of communication and processing. To minimize this problem, **QueCh** has a parameter that indicates the maximum time during which a help acceptance message can stand in the queue. From time to time, the queue is scanned and

the "old" messages are removed. The whole process is continuously executed until all nodes have finished processing all their partitions. We did not perform exhaustive experiments in order to determine the best time for a message to stay in the queue. We intend to do that in the future.

6 Experimental validation

In order to evaluate **QueCh**, we implemented it into **SmaQSS** and run many query processing experiments on a 32-node cluster from Grid'5000 [9]. Previous work [15] showed excellent performance when physical partitioning of the fact tables with no replication was employed. Only dimensions were replicated. In such a scenario, parallel sub-queries benefit from the partitioning design. Furthermore, there was no load unbalancing between nodes during query processing due to the uniform data distribution verified in TPC-H database. This way, super-linear speedup was obtained with minimum disk space utilization. For 32 nodes setup, only 19% of disk space required for full replication was necessary for each node. However, a partitioning design that has no data replication presents serious limitations as it reduces fault tolerance and does not allow for dynamic load redistribution without data transfers.

In these experiments, our goal is to evaluate **QueCh** with the partitioning design already employed for TPC-H database on previous works but with a different partition allocation schema and varying the number of replicas. We first ran experiments using uniform partitions of the fact tables with no replication. Then, we generated data partitions with different sizes. This unbalanced scenario, called "data skew scenario", requires an allocation schema that involves data replication in order to make dynamic load balancing feasible. Furthermore, a data allocation schema that considers replication is more realistic as it increases fault tolerance.

This section is organized as follows. First, we describe the experimental setup. Then we describe our distributed database design. Then we show experimental results obtained when **QueCh** is employed in scenarios with load unbalancing between nodes and different numbers of fact table partition replicas. Finally, we discuss issues in distributed database design.

6.1 Experimental setup

We evaluate **QueCh** performance by running TPC-H queries against **SmaQSS** on a 32-node PC cluster from Grid'5000 [9] located at INRIA-Rennes, France. Each cluster node has two 2.4 GHz Intel Xeon processors, 1 GB RAM and 20 GB hard-disk. Nodes are interconnected by 1 GB Ethernet network. Each node runs an instance of the PostgreSQL 7.3.4 DBMS on Linux. **SmaQSS** is implemented in Java. It employs the multithreading capabilities of Java to take advantage of the 2 processors of each node. **SmaQSS** components communicate via Java RMI.

6.2 Distributed database design for TPC-H

During our experiments, we run different queries from the TPC-H benchmark. TPC-H was employed because it represents decision support applications that issue ad-hoc queries against the underlying database system [1]. The database was generated

according to the benchmark specifications with a scale factor of 5, requiring 11 GB of disk space.

The distributed design follows the specifications from Sect. 3. Dimension tables are fully replicated on all nodes. Tuples from partitioned fact tables (Orders and Lineitem) are physically ordered according to their partitioning attributes and clustered indexes based on these attributes were built. We also created indexes for foreign keys of all tables. As TPC-H queries are ad-hoc, no further optimizations were implemented. All the optimizations implemented are allowed by the benchmark. Due to the uniform value distribution verified for the partitioning attribute, we forced and accentuate unbalance during partition size definitions. The number of replicas was also not pre-determined. Our goal is to evaluate the effects of using different numbers of replicas on performance.

6.3 *QueCh* performance evaluation

We evaluate the speedup obtained during query processing through **QueCh**, our dynamic load balancing technique. The following TPC-H queries were employed: Q1, Q4, Q5, Q6, Q12, Q14, Q18 and Q21. We restrict our analysis to these queries because they have different characteristics and are quite representative of OLAP applications. Each query is executed three times for each kind of experiment, in order to minimize load fluctuations caused by the operating system tasks. The results shown by the tables in this section represent the arithmetic average of the execution times obtained during the three executions. To ease reading, Table 5 shows the average elapsed time for sequential executions of all queries, i.e., the elapsed time obtained when only one node is employed, with no parallelism.

Two cluster configurations were employed during our experiments: the first with 16 nodes and the second with 32 nodes. We chose such configurations because they are more sensitive to load unbalancing. For each configuration, queries were executed in two scenarios: with uniform data partitions and with data skew (non-uniform partitions). The scenario with uniform partitions is considered ideal because there is almost no load unbalancing between nodes during query processing and because it requires minimum disk space, as there is no data replication. The data skew scenario presents severe load unbalancing during query processing, thus requiring load redistribution between nodes. For 16 nodes, the largest Lineitem partition has 9,660,189 tuples, while the smallest has 296,617 tuples. Besides, 9 partitions have less than 900,000 tuples. For 32 nodes, the largest partition has 5,355,957 tuples and the smallest has 143,992. Besides, 23 partitions have less than 500,000 tuples. As we can see, in both cases, the non-uniformity in data distribution is quite magnified.

For data skew experiments, we run experiments by doubling the number of partition replicas from 1 (no replication) to 16 or 32 (full replication), according to the

Table 5 Sequential query processing times in seconds

Sequential execution	Queries							
	Q1	Q4	Q5	Q6	Q12	Q14	Q18	Q21
Elapsed time	802.8	358.8	360.8	200,4	361.6	387.6	300.1	532.1

configuration employed. This means that, for each execution, the number of replicas in each node is doubled and allocated in different consecutive nodes, just as determined by chained declustering. Our goal with such experiments is to evaluate the tradeoff between speedup and disk space utilization when compared to the ideal scenario (uniform data partitions).

Table 6 gives the results obtained for the 16-node configuration. It shows, for each query, the elapsed time obtained for the data skew scenario with different numbers of replicas and for the uniform scenario (ideal). The worst cases are in the second column, which represents experiments with data skew and no replication, which makes dynamic load balancing infeasible for **QueCh**.

Table 6 shows that the use of more replicas and dynamic load balancing accelerates query processing, obtaining elapsed times closer to the ideal. When full replication is adopted (16 replicas in 16 nodes) the elapsed time is even closer. The best cases were obtained for queries Q1 and Q21, which were 3% and 7% faster than in the ideal scenario, respectively. The worst cases were obtained for queries Q4 and Q14, which were 2.31 and 3.24 times slower than in the ideal scenario. For all other queries, the elapsed times were equal or less than 1.34 times the ideal time, showing a good performance for the load balancing strategy implemented.

It is important to notice that, with only 4 replicas, elapsed times not superior to 3.42 times the ideal time were obtained, except for query Q14. However, Q14 takes 6 minutes to run sequentially. Such time falls down to a little more than 1 minute when 16 nodes and 4 replicas are employed, thus significantly accelerating the decision making process.

Comparing Tables 5 and 6, we notice excellent performance improvement for all queries running with 4 replicas, thus requiring only 25% of the disk space employed with full replication. The elapsed execution times obtained with 16 nodes and 4 replicas are, on average, 1/15 of the sequential elapsed time, which is very close to linear speedup. And it is verified on a scenario with severe load unbalancing between nodes. In scenarios with less skew, the speedup obtained with our techniques tends to be greater and very close to the ideal scenario with the additional benefit of raising system availability, because of data replication.

Table 6 Elapsed time (in seconds) for parallel query processing on 16-node configuration with and without data skew

Query	Data skew with dynamic load balancing variation on the number of replicas					Uniform partitions
	1	2	4	8	16	
Q1	245	233	95	57	45	46
Q4	110	82	16	14	10	4
Q5	118	55	32	30	18	14
Q6	67	34	26	18	10	7
Q12	114	57	36	25	19	17
Q14	132	103	64	54	27	8
Q18	91	52	36	26	16	13
Q21	177	92	65	46	34	37

Table 7 Elapsed time (in seconds) for parallel query processing on 32-node configuration with and without data skew

Queries	Data skew with dynamic load balancing variation on the number of replicas						Uniform partitions
	1	2	4	8	16	32	
Q1	133	85	60	44	32	25	23
Q4	59	12	11	9	9	6	2
Q5	85	31	25	24	19	12	8
Q6	36	23	14	11	8	6	4
Q12	57	31	22	19	14	13	8
Q14	82	57	39	37	35	16	4
Q18	51	31	20	17	13	11	7
Q21	103	64	43	34	26	21	18

Similar results were obtained for experiments with the 32 node configuration. The elapsed times obtained are shown in Table 7, similar to Table 6, but with up to 32 replicas. It is very hard to obtain elapsed times lower than those presented by the ideal scenario as the data partitions are small enough to fit into the DBMS data cache, which benefits the consecutive executions of the queries which use them. Even in these cases, good results were obtained by dynamic load balancing.

For 4 replicas, the elapsed times for all queries were equal or less than 3.85 times the elapsed time for the ideal scenario, except for query Q14 (with elapsed time 8.15 times the ideal). It is worth remembering that all results obtained for the ideal scenario yield super-linear speedup when compared to the sequential executions. Thus, what is considered as ideal here is beyond the expectations of classic parallel processing. The best cases are obtained for queries Q1, Q12 and Q21, with elapsed times close to 2.5 times the ideal one, requiring only 25% of the disk space needed for full replication.

In these experiments, the dynamic load balancing technique combined with full replication yields performance that is equal or very close to the ideal for most of the queries considered here. Considering the speedup, it is very close to linear (super-linear for query Q14). Quantitatively, it represents elapsed times only 5 seconds superior to the ideal one, on average. The worst case is verified for query Q14, which required 12 seconds more than in the ideal scenario to be processed. For decision support scenarios, such differences are not critical.

6.4 Issues in distributed database design

Our data partitioning design is based on primary and derived horizontal partitioning for fact tables and no partitioning for dimensions. Due to the non-deterministic nature of the techniques described here, we consider the results obtained very good. By raising the number of partition replicas, we also raise the number of nodes that can contribute to dynamic load balancing. However, the techniques cannot guarantee that each node will effectively be helped during the process.

We can observe that the use of partitioning provides for high performance and significantly improves data storage requirement and consistency maintenance. Using

partition replicas is critical for parallel query processing with load unbalancing. It is worth noticing that load unbalancing is almost unavoidable during parallel processing. Through a simple technique that redistributes tasks between nodes that hold the replicas of the same data partition, it is possible to obtain query processing times very close to the ideal (with almost no load unbalancing). We observed that applying chained declustering to distribute the replicas of the fragments does minimize hot spots and the overload of specific nodes.

On average, the use of 16 nodes with no replication on a scenario with severe load unbalancing obtained query elapsed times only 1/3 faster than the sequential times. With 4 replicas and the load balancing technique, such times are close to 1/15 of the sequential time, on average. This is a very significant speedup for parallel processing.

The results obtained show that the distributed design approach adopted and the dynamic load balancing implemented make it possible to obtain good performance with no full database replication. Previous works employ full replication or no replication at all. However, with our solution, the number of replicas employed depends on the parallel environment available for OLAP applications. The disk space required and the cost of the updates on the OLAP database must be taken into account. For TPC-H, as shown by our experiments, we consider the use of 4 replicas for each partition a good allocation strategy.

7 Conclusion

In this paper, we presented efficient distributed database design strategies for DBC which combine virtual and physical fragmentation with partial replication. We proposed a new load balancing strategy that takes advantage of the replicas to redistribute the load. The main idea behind this strategy is the use of dynamic adaptive virtual partitioning as opposed to defining very small physical partitions. Adaptive virtual partitioning gives much flexibility in defining the size of the partitions while keeping the number of physical partitions small. Thus, managing physical partitions and their replicas is simple. Another advantage of virtual partitioning is to take advantage of the DBMS cache content of a large physical partition, as evidenced in our experiments.

The flexibility of virtual partitioning is very important due to the different types of skew that can occur during query processing. Defining the size of a small physical partition that is efficient for all kinds of ad-hoc OLAP queries and non-uniform values is a complex issue. With adaptive virtual partitioning, the size of the partition can enlarge or shrink depending on the kind of skew as well as the load of the processor nodes.

The use of parallel query processing techniques (in particular, intra-query parallelism), physical data partitioning and an allocation strategy that uses only one replica of each primary partition is enough to obtain performance gains during query processing when compared to sequential environments. However, if disk space is not an issue, using more replicas yields better performance improvement, in particular when there is severe load unbalancing between cluster nodes during query processing.

We evaluated different partition replication strategies by considering scenarios with severe load unbalancing conditions between nodes, ranging from no replication to full replication. Significant performance gains are obtained when four replicas of each partition are employed. However, the appropriate number of replicas for each application environment depends on factors like disk space availability, degree of fault tolerance needed and costs related to data updates which are usually not critical for OLAP applications. Our results show that there is room to improve the load balancing strategy. However, the superlinear results obtained have directed us to keep the solution simple and effective, not necessarily with the highest performance that can be achieved.

The **QueCh** techniques proposed here are being implemented into ParGRES [7], our open source DBC middleware which supports data updates. ParGRES has an SQL3 parser that allows transparent query rewriting and result composition. The download version of ParGRES currently relies on full database replication to implement dynamic load balancing and could benefit from partial replication. We have started this effort with a preliminary implementation on ParGRES, where we evaluated the techniques presented in this paper with a real database from the Brazilian census. Unlike TPC-H, this dataset has a lot of skew within its attribute values and joins. By issuing typical user queries, we confirmed the efficiency of this approach by obtaining very good superlinear results while keeping low costs [22]. We also intend to explore **QueCh** techniques in the context of data grids in GParGRES [23].

Acknowledgements This work was partially supported by CNPq and INRIA. The experiments presented in this paper were carried out using the Grid'5000 experimental testbed. The authors are grateful to the Grid'5000 team.

References

1. TPC: TPC Benchmark^{TM} H—Revision 2.3.0. Transaction Processing Performance Council. http://www.tpc.org/tpch (2008). Accessed 20 April 2008
2. Valduriez, P.: Parallel database systems: open problems and new issues. Int. J. Distrib. Parallel Databases **1**(2), 137–165 (1993)
3. Akal, F., Böhm, K., Schek, H.-J.: OLAP query evaluation in a database cluster: a performance study on intra-query parallelism. In: Proceedings of the 6th East European Conference on Advances in Databases and Information Systems. LNCS, vol. 2435, pp. 218–231. Springer, Berlin (2002)
4. Özsu, T., Valduriez, P.: Principles of Distributed Database Systems, 2nd edn. Prentice Hall, Englewood Cliffs (1999)
5. Röhm, U., Böhm, K., Schek, H.-J.: OLAP query routing and physical design in a database cluster. In: Proceedings of the 7th International Conference on Extending Database Technology. LNCS, vol. 1777, pp. 254–268. Springer, Berlin (2000)
6. Miranda, B., Lima, A.A.B., Valduriez, P., Mattoso, M.: Apuama: combining intra-query and inter-query parallelism in a database cluster. In: Proceedings of the EDBT workshops. LNCS, vol. 4254, pp. 649–661. Springer, Berlin (2006)
7. Mattoso, M., Silva, G.Z., Lima, A.A.B., Baião, F.A., Braganholo, V.P., Aveleda, A., Miranda, B., Almentero, B.K., Costa, M.N.: ParGRES: middleware para processamento paralelo consultas OLAP em clusters de Banco de dados. In: Proceedings of the 21st Brazilian Symposium on Databases—2nd Demo Session, pp. 19–24, 2006
8. Lima, A.A.B., Mattoso, M., Valduriez, P.: Adaptive virtual partitioning for OLAP query processing in a database cluster. In: Proceedings of the 19th Brazilian Symposium on Databases, pp. 92–105, 2004

9. Cappello, F., Caron, E., Dayde, M., Desprez, F., Jegou, Y., Primet, P., Jeannot, E., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Quetier, B., Richard, O.: Grid5000: a large scale and highly reconfigurable grid experimental testbed. In: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, pp. 99–106, 2005
10. PostgreSQL, PostgreSQL DBMS. http://www.postgresql.org (2008). Accessed 11 April 2008
11. Cecchet, E., Marguerite, J., Zwaenepoel, W.: C-JDBC: flexible database clustering middleware. In: Proceedings of the annual conference on USENIX Annual Technical Conference, pp. 26–26, 2004
12. MySQL: A guide to high availability clustering—how MySQL supports 99,999% Availability. MySQL AB. http://www.mysql.com/why-mysql/white-papers/cluster.php (2004). Accessed 10 April 2008
13. PGCluster, PG-Cluster: the multi-master synchronous replication system for PostgreSQL. http://pgcluster.projects.postgresql.org/ (2005). Accessed 11 April 2008
14. Lima, A.A.B., Mattoso, M., Valduriez, P.: OLAP query processing in a database cluster. In: Proceedings of the 10th International Euro-Par Conference. LNCS, vol. 3149, pp. 355–362. Springer, Berlin (2004)
15. Furtado, C., Lima, A.A.B., Pacitti, E., Valduriez, P., Mattoso, M.: Physical and virtual partitioning in OLAP database clusters. In: Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing, pp. 143–150, 2005
16. Furtado, P.: Replication in node partitioned data warehouses. In: VLDB Workshop on Design, Implementation, and Deployment of Database Replication, 2005
17. Furtado, P.: Node partitioned data warehouses: experimental evidence and improvements. J. Database Manag. **17**(2), 42–60 (2006)
18. Stöhr, T., Märtens, H., Rahm, E.: Multi-dimensional database allocation for parallel data warehouses. In: Proceedings of the 26th International Conference on Very Large Databases, pp. 273–284, 2000
19. Bellatreche, L., Boukhalfa, K.: An evolutionary approach to schema partitioning selection in a data warehouse. In: Proceedings of the 7th International Conference DaWaK. LNCS, vol. 3589, pp. 115–125. Springer, Berlin (2005)
20. Hsiao, H., DeWitt, D.J.: Chained declustering: a new availability strategy for multiprocessor database machines. In: Proceedings of 6th International Data Engineering Conference, pp. 456–465, 1990
21. Plastino, A., Ribeiro, C.C., Rodriguez, N.: Developing SPMD applications with load balancing. Parallel Comput. **29**(6), 743–766 (2003)
22. Paes, M., Lima, A.A.B., Valduriez, P., Mattoso, M.: High performance query processing of a real-world OLAP database with ParGRES. In: Proceedings of the 8th International Conference VECPAR. LNCS, vol. 5336, pp. 188–200. Springer, Berlin (2008)
23. Kotowski, N., Lima, A.A.B., Pacitti, E., Valduriez, P., Mattoso, M.: Parallel query processing for OLAP in grids. Concurr. Comput. Pract. Exp. **20**(17), 2039–2048 (2008)