Apuama: Combining Intra-query and Inter-query Parallelism in a Database Cluster†

Bernardo Miranda[1], Alexandre A. B. Lima[1,3],
Patrick Valduriez[2], and Marta Mattoso[1]
[1]Computer Science Department, COPPE, Federal University of Rio de Janeiro
[2]Atlas Group, INRIA and LINA, University of Nantes – France
[3]School of Engineering and Computer Science, University of Grande Rio – Brazil
e-mail: [bmiranda, assis, marta]@cos.ufrj.br, Patrick.Valduriez@inria.fr

Abstract. Database clusters provide a cost-effective solution for high performance query processing. By using either inter- or intra-query parallelism on replicated data, they can accelerate individual queries and increase throughput. However, there is no database cluster that combines inter- and intra-query parallelism while supporting intensive update transactions. C-JDBC is a successful database cluster that offers inter-query parallelism and controls database replica consistency but cannot accelerate individual heavy-weight queries, typical of OLAP. In this paper, we propose the Apuama Engine, which adds intra-query parallelism to C-JDBC. The result is an open-source package that supports both OLTP and OLAP applications. We validated Apuama on a 32-node cluster running OLAP queries of the TPC-H benchmark on top of PostgreSQL. Our tests show that the Apuama Engine yields super-linear speedup and scale-up in read-only environments. Furthermore, it yields excellent performance under data update operations.

1 Introduction

Competitive organizations typically optimize their business processes using decision support systems (DSS) [9]. A DSS includes On-Line Analytical Processing (OLAP) tools and a data warehouse (DW) capable of efficiently handling large amounts of data [3]. Due to the important role played by DSS, much research has been devoted to provide high performance for OLAP queries.

High performance query processing on data warehouses can be achieved using a relational database management system (DBMS) running on top of a PC cluster. PC clusters can scale to very large configurations [8]. Examples of cluster-aware DBMSs are Oracle RAC 10g [5] and DB2 ICE [6]. However, software licensing, hardware specific requirements or database migration costs may prevent their use by many applications. An alternative approach for high-performance data warehousing using PC clusters is a database cluster [1, 2, 14, 18]. A database cluster (DBC) consists of a set of independent DBMSs (not cluster-aware) distributed over a set of cluster nodes,

† Work partially funded by CNPq, Finep, Capes, Cofecub and ACI "Massive Data" in France

and orchestrated by a middleware, responsible for offering a single external view of the whole system, like a virtual DBMS. Applications need not be modified when database servers are replaced by their cluster counterparts. Their queries are sent to the middleware which provides data distribution transparency. Previous work as PowerDB [1], Leg@net [4], C-JDBC [2] and SmaQ [13] have shown the effectiveness of the DBC approach.

Two kinds of parallelism can be exploited in a DBC for query processing: inter-query parallelism and intra-query parallelism. Inter-query parallelism consists of executing many queries at the same time, each at a different node. Inter-query works fine for On-Line Transactional Processing (OLTP) application support, where queries are usually light-weight. However, OLAP applications typically have heavy-weight queries, i.e., queries that access large amounts of data and perform complex operations, thus taking a long time to be processed. Using only inter-query parallelism is not appropriate for heavy-weight query processing as it does not reduce the processing time of individual queries. In such case, intra-query parallelism is the most adequate solution as shown in [14].

Intra-query parallelism consists of using many nodes to process each single query. In this case, each node addresses only a subset of query data and/or query operations. The main goal is to reduce the execution time of individual heavy-weight queries while improving the overall throughput.

Inter- and intra-query parallelisms can be combined in a DBC implementation. Moreover, a DBC with both kinds of parallelism and support for concurrent data updates can be used in both OLAP and OLTP applications. However, current DBC solutions [1], [4], [2] and [14], exclusively support inter-query for OLTP or intra-query for OLAP applications. First, because current DBC solutions for OLAP applications usually consider that database refresh operations are not controlled by them and takes place on a specific predefined time which the DSS is offline. The second reason is that combining inter- with intra-query parallelism can be conflicting. Intra-query parallelism requires the presence of data subsets which are typically produced by physical database design. When the data is physically partitioned among cluster nodes, inter-query parallel processing becomes very limited, since most queries need to scan all partitions in parallel. Depending on the data partitioning design, a simple OLTP query must be processed by intra-query parallelism and becomes very inefficient. On the other hand, OLAP queries without data partitioning cannot be performed efficiently.

Our goal is to provide a high-performance and low-cost DBC solution that supports OLTP and OLAP workloads. To avoid the problems with physical database partitioning, we adopt dynamic virtual partitioning to a replicated database. We use C-JDBC, an industrial quality open-source DBC solution that offers support for inter-query parallelism and database replica consistency. C-JDBC provides excellent performance for OLTP applications [2] but does not support intra-query parallelism. Thus, we extend C-JDBC with a non-intrusive intra-query solution.

In this paper, we present the Apuama[1] Engine as an extension of C-JDBC. The main goal is to provide an environment to process OLAP queries using intra-query parallelism while keeping the effectiveness of C-JDBC to support OLTP transactions.

[1] *Apuama* means *fast* in Tupi-Guarani, a primitive language of South America.

No source code was changed in C-JDBC. Apuama acts as a connection proxy between C-JDBC and the DBMSs. It does not interfere with the C-JDBC query processing and is only used for OLAP query processing. Unlike all other DBC intra-query solutions, Apuama also provides for database replica consistency during intra-query processing.

To evaluate Apuama, we ran experiments based on the TPC-H benchmark [19] (specific for OLAP applications) on a 32-node cluster using PostgreSQL [16]. Query processing speedup and throughput scalability were measured on experiments with read-only queries and a mix of read-only queries and concurrent data updates. In most cases, Apuama achieves super-linear speedup and scale-up. Since there has been no change to C-JDBC inter-query processing, successful OLTP results are sustained.

The paper is organized as follows. Section 2 introduces the basic concepts for intra-query parallelism in DBC. Section 3 explains intra-query support in Apuama. Section 4 presents the architecture of Apuama as an extension to C-JDBC. Section 5 describes experimental results. Section 6 discusses related work. Section 7 concludes.

2 Intra-Query Parallel Processing in DBC

Intra-query parallelism consists of having each query being processed by many nodes in parallel. This can be achieved in different ways. The most frequent solutions are through data parallelism, where the same query is executed against different parts of a partitioned database in parallel. A DBMS that has parallel capabilities usually offers several data partitioning techniques that are used during physical database design. Such DBMS provides transparent access to the partitioned database and has full control over the parallel query execution plan. This is not the case for DBC.

In DBC, independent DBMSs are used by a middleware as “black-box” components. It is up to the middleware to implement and coordinate parallel execution. This means that query execution plans generated by such DBMSs are not parallel. Furthermore, as “black-boxes”, they cannot be modified to become aware of the other DBMS and generate parallel plans.

When the database is replicated at all nodes of the DBC, inter-query parallelism is almost straightforward. Any read query can be sent to any node and execute in parallel. On the other hand, to implement intra-query in a DBC, the application database must be partitioned and distributed among the DBC nodes. The use of a replicated database or a partitioned database can also impact on the way the update transactions are processed. For replicated databases, the DBC must send a notification to all replicas in order to complete an update transaction. Using a partitioned database, the update transaction processing is faster because the DBC has to notify a smaller number of nodes. The notification is sent just to the owners of updated tuples. However, physical data partitioning can be complex to design, hard to maintain, and can cause severe data skew. In addition, automatically generating a parallel query execution plan can be quite complex.

An interesting solution to combine inter- and intra-query parallelism is to keep the database replicated and design partitions using *virtual partitioning* (VP) [1]. VP is based on replication and dynamically designs partitions. The basic principle of VP is

to take one query, rewrite it as a set of sub-queries "forcing" the execution of each one over a different subset of the table. In the PowerDB DBC [1], this is implemented in a simple way called Simple Virtual Partitioning (SVP) that works as follows. First, the database is fully replicated over all cluster nodes. Then, when a query *Q* is submitted to a DBC with *n* nodes, a set of sub-queries $Q_{i=1..n}$ is produced. Each Q_i is formed by the addition of a different range predicate to *Q* at the where clause. The goal is to make each sub-query to run over a different subset of the data that must be accessed by *Q*. Then, each sub-query is sent to a different node, where it is executed by the local DBMS. After sub-query execution, the DBC produces the final result based on the partial results of each node. Let us take the following query *Q* to be executed in a DBC with four nodes:

```
Q: select sum(l_extendedprice) from lineitem        (1)
```

According to SVP, *Q* would be rewritten as follows:

```
Qi: select sum(l_extendedprice) from lineitem       (2)
   where  l_orderkey >= :v1 and l_orderkey < :v2
```

The difference between *Q* and Q_i is the range predicate "l_orderkey > :*v1* and l_orderkey <= :*v2*". We call *virtual partitioning attribute* (VPA) the attribute chosen to virtually partition the table. The values used for parameters *v1* and *v2* vary from node to node and are computed according to the total range of the VPA values and the number of nodes. Assuming that the interval of values of l_orderkey is [1; 6,000,000] and we have 4 nodes, then, 4 sub-queries must be generated. The intervals covered by each sub-query are the following: Q_1: *v1*=1 and *v2*=1,500,001; Q_2: *v1*=1,500,001 and *v2*=3,000,001; and so on. Although all nodes have a replica of lineitem, VP forces each Q_i to process a different and disjoint subset of lineitem's tuples.

However, this approach does not guarantee that different physical parts of lineitem will be scanned. If the tuples of the added range are scattered along the table storage, all disk pages occupied by the table might be accessed. For SVP to be effective, the tuples of the virtual partition must be physically clustered according to the VPA and there must be an index associated to this attribute, i.e., there must exist a clustered ordered index on lineitem based on l_orderkey. Furthermore, the DBMS optimizer must choose the clustered index to be used in the execution plan.

Query re-writing is not trivial. We base our transformations on some typical query templates from OLAP queries, adopting some hints from [1]. To leverage this complexity we only apply VP on fact tables, which makes it easier to re-write queries with complex joins. Some SQL functions require a more complex query modification. For example, the avg() function of a query must be rewritten in the sub-queries as a sum() function followed by a count() function to address a global average. Still some queries, such as complex nested queries, cannot be transformed. In those cases, intra-query is not explored.

3 Query Processing with Apuama

Apuama is an extension of C-JDBC responsible for providing intra-query parallelism. It is implemented as an external component, without changing C-JDBC. OLTP transactions are processed by C-JDBC without any change. In this section we explain how OLAP queries are handled by Apuama through intra-query processing, result composition and update transactions.

Apuama implements intra-query parallelism based on SVP. Query speedup with SVP is DBMS-dependent since the partitioned table must be accessed through a clustered index associated to the partitioning attribute. If, for any reason, the DBMS optimizer chooses a full table scan to execute a sub-query, the virtual partition is ignored and the performance of SVP can be severely hurt. Even though a full table scan can be more efficient for an isolated query execution, in Apuama it is also important trying to keep most of the virtual partition data at the cache. Thus, in order to guarantee effective exclusive access to the virtual partition tuples, Apuama directly interferes in DBMS optimizer choices in order to force index usage. This is done by asking the DBMS to disable full table scans during heavy-weight intra-query processing. This can be done in many popular open-source DBMSs, e.g. MySQL [7] and PostgreSQL.

Apuama disables full scans only before starting to process a query using intra-query parallelism. When the query processing is finished, the original DBMS settings are reestablished. This strategy is not DBMS-independent because the command used to do that is not standard, although it is common knowledge for the most of DBAs. Thus, Apuama must detect which DBMS driver is being used to make DBMS-specific changes on the query execution plan. This information is part of Apuama's metadata and it is set during software installation.

Sub-queries produced by SVP in Apuama are independently processed by each node and their partial results must be combined in order to form the final query result. Apuama uses HSQLDB [10], a fast in-memory DBMS, to perform result composition. This method proved to be very efficient during our experiments. In many experiments, aggregations performed by HSQLDB took no more than one second to be processed even with large partial results involving several columns.

Updates in Apuama are propagated to all nodes in the same order to guarantee consistency among different replicas. The time needed to broadcast updates over all nodes increases according to the number of nodes in the cluster. With full replication, this can impact performance in update-intensive situations. Fortunately, this is not the case for most decision-making environments. Although C-JDBC does not require full replication, we adopted it to maximize speedup through intra-query parallelism. Solutions using the replica freshness techniques [15] are out of the scope of this work.

In order to produce consistent results for heavy-weight queries, Apuama must guarantee that, before beginning to process a query using intra-query parallelism, all node replicas are consistent with each other. Distributed updates are performed by C-JDBC, which is not aware of the existence of sub-queries generated by SVP. With C-JDBC only, we can assure that updates are executed in the same order in all nodes but we cannot assure that updates and SVP sub-queries are executed in the same order. Different execution threads, executing update and read-only operations, may be scheduled in different orders by the operating systems of different nodes. Therefore,

Apuama provides a blocking mechanism to avoid performing updates along SVP sub-queries of the same query. Apuama has a transaction counter for each node. When a query must be processed with SVP, Apuama waits until a consistent state is reached by all nodes. This happens when all transaction counters are equal. If new update transactions arrive, they are blocked. Then, Apuama starts executing SVP, dispatching all sub-queries to their respective nodes. When all sub-queries are sent and started by the DBMSs, update transactions are unblocked and can be executed. The transaction isolation provided by the DBMS makes it possible to have the updates executed before each sub-query finishes, thereby improving throughput.

4 Apuama Architecture

This section describes the Apuama architecture and its integration within C-JDBC. Fig. 1(a) shows our architecture that contains only C-JDBC components relevant to our explanation and the Apuama Engine extension. The main purpose of C-JDBC is to offer transparent access to a cluster of databases without any modification on the client application. The unique requirement is to use a JDBC driver [11]. Instead of having the application directly connected to the DBMS, it is connected to C-JDBC controller using a C-JDBC JDBC driver.

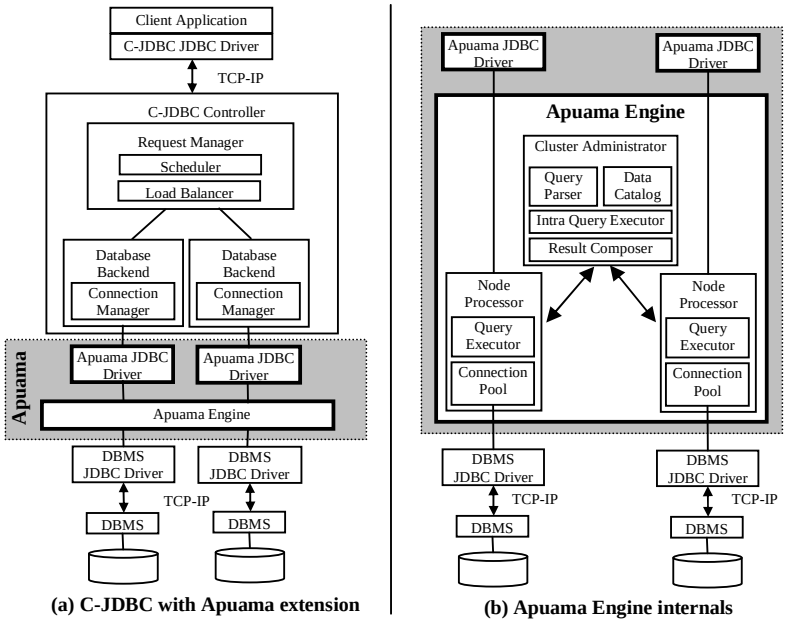

Fig. 1. Apuama architecture

The *C-JDBC controller* is a Java process that manages all database resources. It has a *Database Backend* component that manages a pool of connections to running

DBMSs. Each request received by C-JDBC is submitted to the *Scheduler* component that controls concurrent request executions and makes sure that update requests are executed in the same order by all DBMSs. The Scheduler can be configured to enforce different parallel levels of concurrency. In our experiments, it was set to concurrently execute read and write requests. After a request is scheduled to run, the *Load Balancer* component chooses which Database Backend will execute it. If it is a write request, the same query is executed in every Database Backend to maintain consistency. But if it is a read request, the Load Balancer chooses the best node to execute it. This choice is based on a previously established policy. We configured the Load Balancer to select the node with the least number of pending requests.

Apuama does not require any changes in C-JDBC source code. Fig. 1(a) shows that Apuama is a layer between C-JDBC and the DBMSs. C-JDBC no longer makes any direct connection to DBMSs. Each Database Backend connects to Apuama through a JDBC driver. It is Apuama that connects to the DBMSs.

Fig. 1(b) shows a detailed architecture of Apuama. It has two kinds of components: one that manages intra-query parallel executions, called *Cluster Administrator*, and a set of *Node Processor* components. For each connection established by C-JDBC using Apuama, a Node Processor is created and is responsible for mediating and monitoring requests sent to its corresponding DBMS. To be able to process multiple requests, the Node Processor creates a pool of connections.

The Cluster Administrator has a *Query Parser* component capable of determining which tables are referenced by a query and a *Data Catalog* that contains information about tables that can be virtually partitioned. They are used to determine if a current OLAP query can be processed using intra-query parallelism or not. If not, the Node Processor simply redirects the request to the corresponding DBMS. Otherwise, the Cluster Administrator takes the query and processes it through the *Intra-Query Executor* (IQE) component. When all sub-queries are ready to be processed, they are sent in parallel to the *Query Executor* of each Node Processor. The Query Executor is responsible for sending the sub-query to its corresponding DBMS and waiting for results, which are sent to the IQE that forwards it to the *Result Composer*. It uses HSQLDB to store the partial results and perform final result composition. When all partial results are collected, the Result Composer produces the final result that is sent back to the client application.

5 Experiments

In this section, we evaluate the intra-query parallel processing capabilities added to C-JDBC by Apuama in different scenarios. We ran experiments based on the TPC-H benchmark. We stress Apuama in situations of high concurrency levels, even while executing database refresh operations.

All tables are replicated, but only the fact tables are virtually partitioned (orders and lineitem). Typically, fact tables have the highest cardinality in the database and they are frequently involved in OLAP queries, particularly in heavy joins. Thus, their reduced number of tuples can improve IO and CPU processing. Dimension tables are not virtually partitioned because they are small tables and represent 14% of total

database size. We employ virtual partitioning on orders, based on its primary key (o_orderkey). The first attribute of the primary key of lineitem (l_orderkey) is a foreign key to orders. So, by choosing l_orderkey we generate a derived partitioning on lineitem. Tuples of the fact tables are physically ordered according to their partitioning attributes and indexes were built over them. Also, indexes are built for all foreign keys of all tables. As TPC-H assumes *ad-hoc* queries, we perform no other optimization, as determined by the benchmark.

The TPC-H queries that involve a fact table can benefit from the virtual partitioning. Exceptionally, some queries that contain subqueries involving fact tables cannot be rewritten using virtual derived partitioning. We use a subset of 8 queries from TPC-H. As in the specification, we identify queries by their numbers: Q1, Q3, Q4, Q5, Q6, Q12, Q14 and Q21. We chose such queries because they represent OLAP queries of different complexities. Q1 accesses only the lineitem table and performs many aggregate operations. The "where" predicate of Q1 is not very selective since around 99% of tuples are retrieved. This is a very costly query. Q3 joins lineitem, orders and a dimension table. Differently from other queries, its result contains a large number of rows. Q4 contains a reference to lineitem table and a sub-query with another reference to lineitem. Q5 joins lineitem, orders and four dimension tables. It performs only one aggregate operation. As Q1, Q6 accesses just the lineitem table. The main differences between them are that Q6 has only one aggregate operation and its "where" predicate is much more selective, retrieving only 1.5% of tuples. Q12 joins lineitem and orders tables and has two aggregation operations. Q14 joins lineitem table and a dimension table. Q21 contains three references to lineitem table. Two of those references are part of two sub-queries, respectively.

There are three kinds of experiments: first, we analyze speedup obtained with Apuama when processing isolated individual queries. Then, we evaluate the system overall throughput with sequences of read-only queries. Finally, we evaluate throughput obtained while simultaneously processing read-only and update queries.

We ran experiments on top of a 32-node shared-nothing cluster system from the Paris Team at INRIA [17]. Each node has two 2.2 GHz Opteron processors with 2 GB RAM and 30 GB HD. The network is a Gigabit ethernet. An instance of PostgreSQL 8 was running at each node. The total database size on disk, including all tables and indexes, is about 11 GB, for a TPC-H database with a scale factor of 5. We use HSQLDB to compute the final results.

In the following, we present results that show query execution times and throughput rates for an increasing number of nodes (from 1 to 32). Every execution was repeated five times and the final metric is the mean value obtained in such runs, not considering the first one. In some cases, metrics were also normalized by dividing their value by the value obtained during the same kind of experiment with one node. In order to ease reading and analysis, values are presented in logarithmic scale to give a clear notion of linearity [14].

The first experiment (Fig. 2) evaluates the speedup obtained with Apuama when executing isolated queries in different cluster configurations. With 2 nodes, query execution time for all queries is reduced by almost 50%, when compared to the sequential execution. With 4 nodes, query execution time is decreased from 45% to 20% for all queries, except for Q4 and Q6 that were decreased to 1.2% and 6.8% of the original time, respectively. As Q4 and Q6 are highly selective queries, fragments

obtained by virtual partitioning are small enough to fit in main memory with just four nodes, resulting in super-linear speedup. For such configurations, we could observe that, after the first query execution, no page faults occur, thus avoiding disk accesses. However, we continue to see a linear speedup with 8, 16 and 32 nodes showing the effectiveness of virtual partitioning even for in-memory databases. Because Q1 and Q21 are CPU-bound queries, they do not benefit from IO improvement. Still their speed-up is always near linear.

In the next experiment, read-only query sequences are executed in parallel against the DBC. All sequences are composed by the same 8 queries, sorted in different ways, according to TPC-H specification. Each sequence submits the next query after the completion of the current query. This is how TPC-H simulates a decision-making user formulating new queries based on previous query results. Queries from different sequences are submitted in parallel.

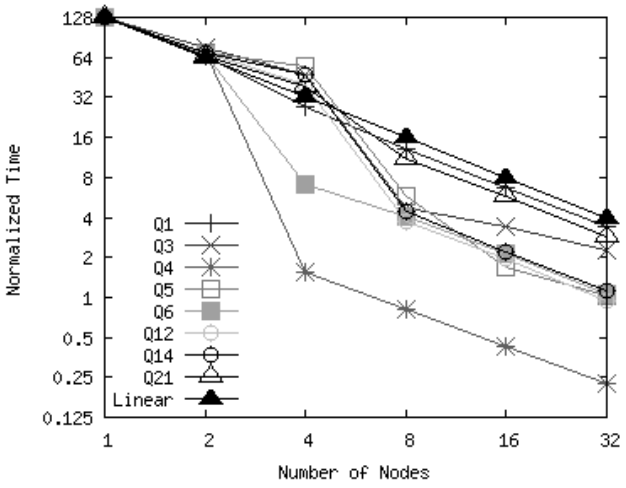

Fig. 2. Speedup experiments - normalized query execution times

Fig. 3(a) shows the throughput rate (in queries per minute) obtained during the execution of three concurrent query sequences in different cluster configurations. It also shows the throughput that would be achieved if linear scale-up throughput was obtained. The number of query sequences was defined according to TPC-H, which recommends this level of concurrency for OLAP databases with a scale factor of 5. For all configurations, the throughput rises super-linearly. With 2 nodes, it is near linear. With 4 nodes, the throughput is almost 2 times higher than if a linear gain was obtained. From 8 to 32 nodes, the throughput is constantly about 6 times higher than linear gain showing excellent performance.

Fig. 3(b) shows the scale-up throughput rate when Apuama is processing sequences of read-only queries. In this experiment, the number of concurrent query sequences is equal to the number of nodes being used. Therefore, the ideal situation is that the execution time would be the same for all cluster configurations, as the “Linear” curve shows. As in the previous experiment, the performance obtained with

2 nodes is better than expected. With 4 nodes, the performance is more than two times better than expected. From 8 to 32 nodes, the performance is always about 3 times better than expected, showing very good scalability with respect to query load. Therefore, Apuama can be used to successfully reduce OLAP individual query execution time and increase system throughput in a typical OLAP scenario and even in the presence of high-concurrency.

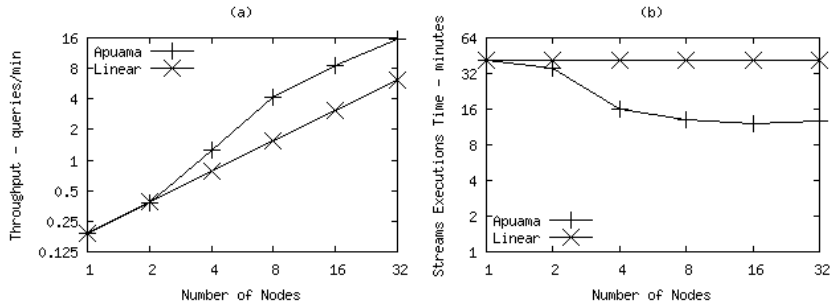

Fig. 3. Read-only query sequences experiment - (a) throughput with 3 concurrent sequences, (b) execution time with *n* concurrent sequences, where *n* is equal to the number of nodes

In the following experiment, we mix read-only query sequences with an update query sequence. The update operations consist of 52,500 transactions for all cluster configurations. First, the update queries insert an amount of data on the lineitem and orders tables. In a second step, the updates remove all inserted tuples from lineitem and orders tables.

Fig. 4(a) shows throughput (in queries per minute) obtained while concurrently processing 3 read-only query sequences and an update query sequence. Again, it shows the throughput that would be achieved if linear gain was obtained. From 2 to 8 nodes, performance of Apuama is near linear. For 16 and 32 nodes, the consistency protocol makes the update propagation delay hurt performance. There is almost no performance gain from 16 to 32 nodes. Fig. 4(b) shows scalability in Apuama with a concurrent updates. Here, the number of read-only sequences equals the number of nodes while there is always one update sequence. There is a performance gain up to 16 nodes. However, for 32 nodes, the performance is almost the same as with 4 nodes. This is due to the replica synchronization when using a large number of nodes.

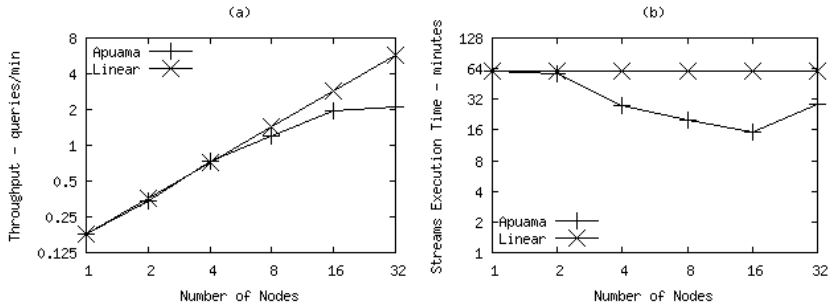

Fig. 4. Mixed workload experiment - (a) throughput with 3 read-only and 1 update sequence (b) execution time with 1 update and n read-only queries, where n is equal to the number of nodes

In summary, these experiments show that Apuama provides excellent performance in processing read-only query workloads. This is true for typical OLAP scenarios and for those with a high degree of concurrency. With mixed workloads (consisting of read-only and update queries), reasonable performance can still be obtained and the system does not need to be unavailable for end-users while data refresh operations are carried out. Thus, in typical OLAP scenarios, where such operations occur only from time to time, we can conclude that Apuama is a good solution for DSS.

6 Related Work

The main DBC projects that can be found in the literature are C-JDBC [2], PowerDB [1], PowerDB-FAS [18], SmaQ [13] and Leg@net [4]. C-JDBC, PowerDB-FAS and Leg@net only support inter-query parallelism. Apuama was developed as an extension to C-JDBC to support OLAP applications through intra-query parallelism.

PowerDB provides intra-query parallelism, but does not guarantee query speedup because it depends on a DBMS-specific query execution optimization. Thus their results are unstable. Furthermore, in contrast with our motivation to offer a low-cost solution, PowerDB software is not freely available.

SmaQ is also focused on OLAP. It uses a technique called “adaptive virtual partitioning” (AVP) [14] that reduces query execution time and allows for dynamic load balancing during query execution. Although SmaQ can support both inter- and intra-query parallelism, it does not support update transactions. The main difference between SmaQ and Apuama is the existence of a replica consistency management. Besides, experiments showed that execution of OLAP queries in environments with high levels of concurrency can lead to poor performance. Apuama uses a simpler virtual partition technique than AVP that allows for better concurrent queries support. Since AVP locally subdivides the local sub-query it increases the level of concurrency while inducing a bad memory cache use.

7 Conclusion

We proposed the Apuama Engine as an extension to C-JDBC, a successful open-source DBC. Apuama adds intra-query parallel processing capabilities. The result is a low-cost powerful and unique DBC that can simultaneously support OLTP and OLAP applications. We implemented intra-query parallelism using SVP, an efficient technique that can be used with different DBMSs requiring standard, non-intrusive techniques and almost “black-box” DBMS components.

To validate our solution, we implemented Apuama on a 32-node cluster system and ran experiments with typical queries of the TPC-H benchmark. By varying the number of nodes in our experiments, it was possible to examine the query processing performance in cases that the virtual partition size is larger or smaller than the amount

of available memory in a node. Although the performance improvement is better when the virtual partition fits in memory, the query super-linear speedup occurs in both cases. When processing isolated queries, super-linear speedup was obtained in most situations. When processing parallel sequences of read-only queries, the performance gain of Apuama is super-linear for all cluster configurations, even in scenarios with high levels of concurrency. With mixed workloads that combine parallel sequences of read-only queries and large amounts of data updates, the performance gain is also very good. The performance gain is near-linear for most experiments of speed-up test and super-linear for throughput scale-up test. Thus, Apuama is a good solution for high-performance DSS. In the presence of updates, Apuama presented performance deterioration when a large number of nodes were employed due to the replica consistency protocol. As a future work we plan to focus on this limitation using an alternative replication policy that relaxes consistency. The tradeoff between OLAP query result correctness and update transaction performance would be analyzed.

Apuama is released as an open source software protected under the terms of LGPL [12] license. It can be downloaded from http://www.cos.ufrj.br/~bmiranda/apuama.

References

1. Akal, F., Böhm, K., Schek, H.-J.: OLAP Query Evaluation in a Database Cluster: A Performance Study on Intra-Query Parallelism. Proceedings of the 6th East-European Conference on Advances in Databases and Information Systems (ADBIS), Bratislava, Slovakia (2002) 218-231
2. Cecchet, E.: C-JDBC: a Middleware Framework for Database Clustering. Proceedings of IEEE Data Engineering Bulletin Vol. 27 (2004) 19-26
3. Chaudhuri, S., Dayal, U.: An Overview of Data Warehousing and OLAP Technology. ACM SIGMOD Record Vol. 26 (1997) 65-74
4. Coulon, C., Pacitti, E., Valduriez, P.: Scaling Up the Preventive Replication of Autonomous Databases in Cluster Systems. Proceedings of 6th International Conference on High Performance Computing for Computational Science (VECPAR), Valencia, Spain (2004) 170-183
5. Cruanes, T., Dageville, B., Ghosh, B.: Parallel SQL Execution in Oracle 10g. Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France (2004) 850-854
6. DB2 ICE. Retrieved 11/09/2005, from http://ibm.com/software/data/db2/linux/ice.
7. MySQL 5.0 Documentation. Retrieved 11/09/2005, from http://mysql.com.
8. Gançarski, S., Naacke, H., Pacitti, E., Valduriez, P.: Parallel Processing with Autonomous Databases in a Cluster System. Proceedings of International Conference on Cooperative Information Systems (CoopIS), Los Angeles, USA (2002) 410-428
9. Gorla, N.: Features to Consider in a Data Warehousing System. Communications of the ACM Vol. 46 (2003) 111-115
10. HSQL Database Engine. Retrieved 11/09/2005, from http://hsqldb.org/.
11. JDBC. Retrieved 11/09/2005, from java.sun.com/products/jdbc.
12. LGPL. Retrieved 11/09/2005, from http://www.gnu.org/copyleft/lesser.html.
13. Lima, A.A.B.: Intra-Query Parallelism in Database Clusters. COPPE/UFRJ, D.Sc. Thesis, Rio de Janeiro (2004)

14. Lima, A.A.B., Mattoso, M., Valduriez, P.: Adaptive Virtual Partitioning for OLAP Query Processing in a Database Cluster. Proceedings of 19h Brazilian Symposium on Databases (SBBD), Brasilia, Brazil (2004) 92-105
15. Pape, C.L., Gançarski, S., Valduriez, P.: Refresco: Improving Query Performance Through Freshness Control in a Database Cluster. Proceedings of International Conference on Cooperative Information Systems (CoopIS), Agia Napa, Cyprus (2004) 174-193
16. PostgreSQL 8.0.1 Documentation. Retrieved 11/09/2005, from http://postgresql.org.
17. Paris Project. Retrieved 11/09/2005, from http://www.irisa.fr/paris/General/cluster.htm.
18. Röhm, U., Böhm, K., Schek, H.-J., Schuldt, H.: FAS - A Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components. Proceedings of the 28th International Conference on Very Large Data Bases (VLDB), Hong Kong, China (2002) 754-765
19. TPC-H Benchmark. Retrieved 11/09/2005, from http://tpc.org/tpch.