Taming Elephants, or How to Embed Parallelism into PostgreSQL*

Constantin S. Pan and Mikhail L. Zymbler

South Ural State University, Chelyabinsk, Russia

Abstract. The paper describes the design and the implementation of PargreSQL parallel database management system (DBMS) for cluster systems. PargreSQL is based on PostgreSQL open-source DBMS and exploits partitioned parallelism. Presented experimental results show that this scheme is worthy of further development.

1 Introduction

Currently open-source PostgreSQL DBMS is one of reliable alternatives for commercial DBMSes [9]. There are many practical database applications based upon PostgreSQL and research projects devoted to extension and improvement of PostgreSQL.

One of the research goals is to adapt PostgreSQL for parallel query processing. In this paper we describe the architecture and design of PargreSQL [8] parallel DBMS for analytical data processing on cluster systems. PargreSQL represents PostgreSQL with embedded partitioned parallelism.

The paper is organized as follows. Section 2 gives a description of the PargreSQL architecture in comparison with PostgreSQL's one. Section 3 introduces the implementationt principles of PargreSQL DBMS. The results of experiments on the current implementation are shown in section 4. Section 5 briefly discusses related work. Section 6 contains concluding remarks and directions for future work.

2 PargreSQL Design

PargreSQL utilizes the idea of partitioned parallelism [2] in cluster systems (see fig. 1). This form of parallelism supposes partitioning of relations and their distribution among the disks of the cluster.

The way the partitioning is done is defined by a *fragmentation function*, which for each tuple of the relation calculates the number of the processor node where this tuple should be placed. A query is executed in parallel on all processor nodes as a set of parallel *agents*. Each agent processes its own fragment and generates a partial query result. The partial results are merged into the result relation.

* The reported study was partially supported by the Russian Foundation for Basic Research, research projects No. 12-07-31217 and No. 12-07-00443.

H. Decker et al. (Eds.): DEXA 2013, Part I, LNCS 8055, pp. 153–164, 2013.
© Springer-Verlag Berlin Heidelberg 2013

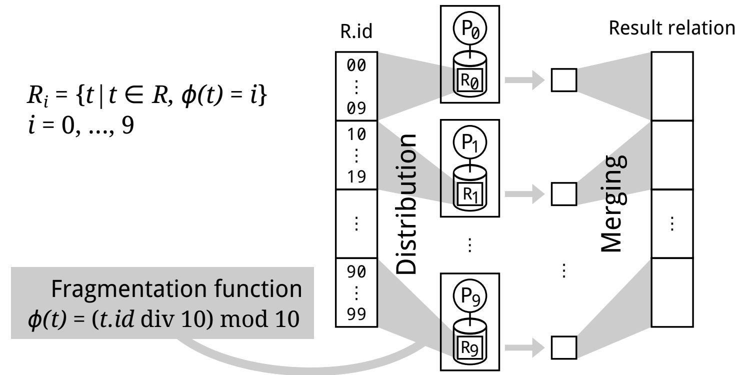
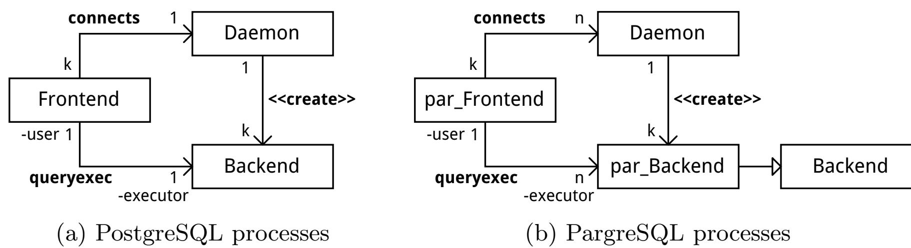


Fig. 1. Query processing with partitioned parallelism

2.1 Client-Server Model

PostgreSQL is based on the client-server model. A session involves three processes into interaction: a frontend, a backend and a daemon (see fig. 2a; here k is a number of clients).



(a) PostgreSQL processes (b) PargreSQL processes

Fig. 2. DBMS processes

The daemon handles incoming connections from frontends and creates a backend for each one. Each backend executes queries received from the related frontend. The architecture of PargreSQL, in contrast with PostgreSQL, assumes that a client connects to two or more servers (see fig. 2b; here n is a number of cluster nodes).

The interaction sequence is shown in fig. 3. As opposed to PostgreSQL there are many daemons running in PargreSQL. The frontend connects to each of them, sends the same query to many backends, and receives the result relation.

2.2 Deployment Scheme

The application library *libpq* implements the interaction protocol between the client and the server and consists of two parts: the frontend (*libpq-fe*) and the

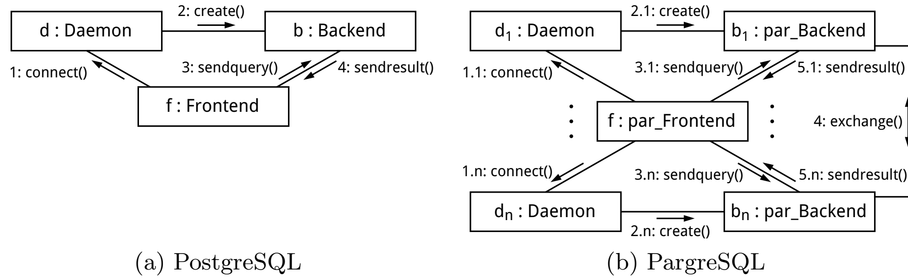


Fig. 3. Interaction of clients and servers

backend (*libpq-be*). The former is deployed on the client side and serves as an API for the end-user application. The latter is deployed on the server side and serves as an API for *libpq-fe*, as shown in fig. 4a.

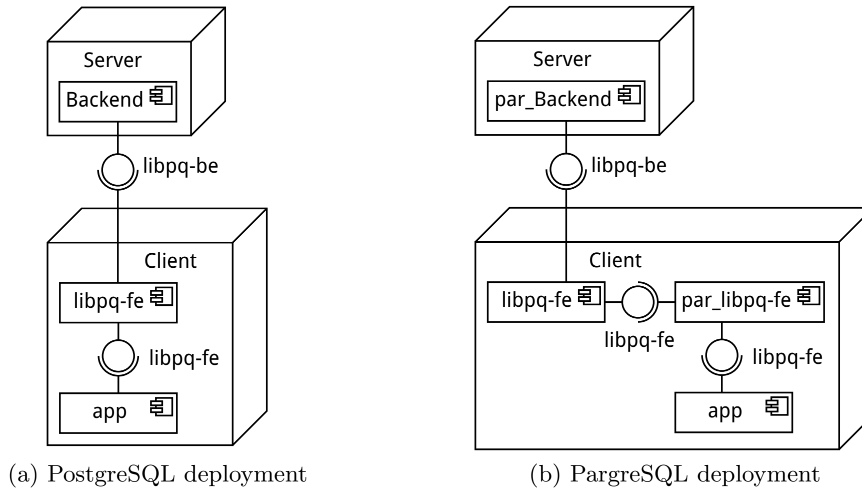


Fig. 4. DBMS deployment

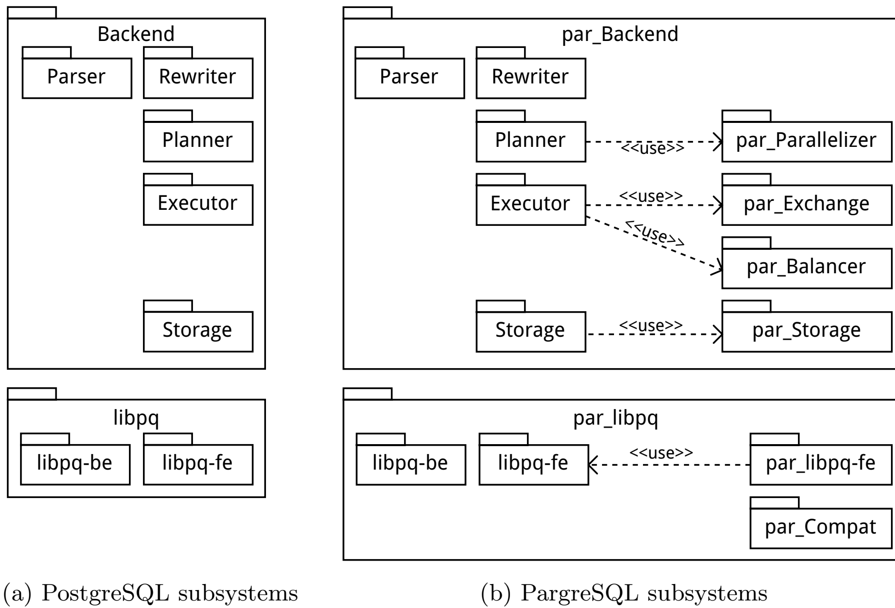
PargreSQL deployment scheme is depicted in fig. 4b. The only difference of deployment schemes (see fig. 4b) is that in case of PargreSQL there is one more component on the client side — the *libpq-fe* wrapper.

2.3 PargreSQL Subsystems

There are following steps of query processing in PostgreSQL: *parse*, *rewrite*, *plan/optimize*, and *execute*.

Parallel query processing in PargreSQL adds two more steps: ***parallelize*** and ***balance***. During the query execution each agent processes its own part of the relation independently so, to obtain the correct result, transfers of tuples are required. On *parallelization* step creation of a parallel plan is performed by inserting special *exchange* operators into the corresponding places of the plan. *Balance* step provides load-balancing of the server nodes during the query execution process.

Comparison of PostgreSQL and PargreSQL architectures is depicted in fig. 5. PostgreSQL (see fig. 5a) is treated as one of the PargreSQL's subsystems (see fig. 5b). PargreSQL development involves changes in Storage, Executor and Planner subsystems of PostgreSQL.



(a) PostgreSQL subsystems

(b) PargreSQL subsystems

Fig. 5. DBMS subsystems

Parser checks the syntax of the query string and builds a parse tree. *Rewriter* processes the tree according to the rules specified by the user (e.g. view definitions). *Planner* creates an optimal execution plan for this query tree. *Executor* takes the execution plan and processes it recursively from the root. *Storage* provides functions to store and retrieve tuples and metadata.

The changes in the PostgeSQL's source code are needed to integrate it with the new subsystems. *par_Storage* is responsible for storing partitioning metadata of relations. *par_Exchange* encapsulates the implementation of the *exchange* operator. *Exchange* operator is meant to compute the exchange function ψ for each tuple of the relation, send "alien" tuples to the other nodes, and receive "native" tuples in response. In section 3.2 we will describe *exchange* operator in detail.

There are new subsystems which do not require any modifications to the PostgreSQL's source code: *par_libpq-fe* and *par_Compat*. *par_libpq-fe* is a wrapper around *libpq-fe*, it is needed to propagate queries from an application to many servers. *par_Compat* makes this propagation transparent to the application. Section 3.1 further describes implementation details of *par_libpq* subsystem.

3 PargreSQL Implementation

In this section we describe the implementation principles of some of the PargreSQL subsystems depicted in fig. 5b.

3.1 par_libpq

Since the frontend in PargreSQL has to initiate a connection to every of the database daemons, some modifications were introduced into the *libpq* application library. The modified version is called *par_libpq*. The purpose of this library is to serve as a replacement for the original *libpq* and to allow the applications to use PargreSQL without much effort.

par_libpq consists of *par_libpq-fe* library and a set of macros (*par_Compat*). *par_libpq-fe* is a library to be linked with frontend applications instead of original PostgreSQL's *libpq-fe*, around which it is a wrapper. Its implementation is illustrated with a class diagram in fig. 6. The idea is to use the original *libpq-fe* for connecting to many servers simultaneously.

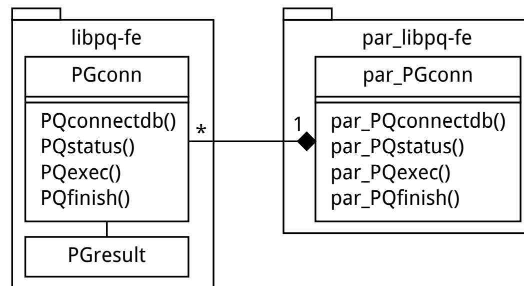


Fig. 6. PargreSQL libpq-fe wrapper

par_Compat is a set of C preprocessor definitions for transparent usage of *par_libpq-fe*. An example of these macros is given in fig. 7.

These macros change the original API calls into the new API calls, so by including them an application programmer can switch from PostgreSQL to PargreSQL without global changes in the application code.

```
#define PGconn par_PGconn
#define PQconnectdb(X) par_PQconnectdb()
#define PQfinish(X) par_PQfinish(X)
#define PQstatus(X) par_PQstatus(X)
#define PQexec(X,Y) par_PQexec(X,Y)
```

Fig. 7. PargreSQL compatibility macros

3.2 Exchange Operator

In order to compute the correct results the DBMS instances running in parallel have to send tuples to each other, because a tuple stored on one node could be processed on another node, e.g. in case of an aggregation with group-by on attribute A while the fragmentation attribute is B. To resolve such situations we should implement an operator that would move tuples from one point in the query plan to the same point on another node's plan.

Exchange operator [3,10] serves as an exchange point for transfering tuples between parallel agents. It is inserted into the query plans by *par_Parallelizer* subsystem (which will be discussed further). The operator's structure is presented in fig. 8.

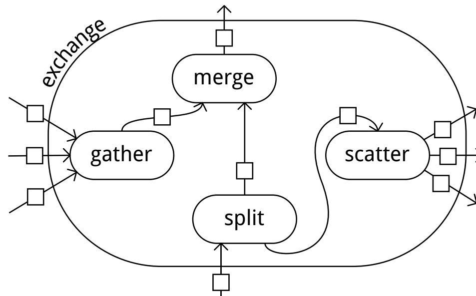


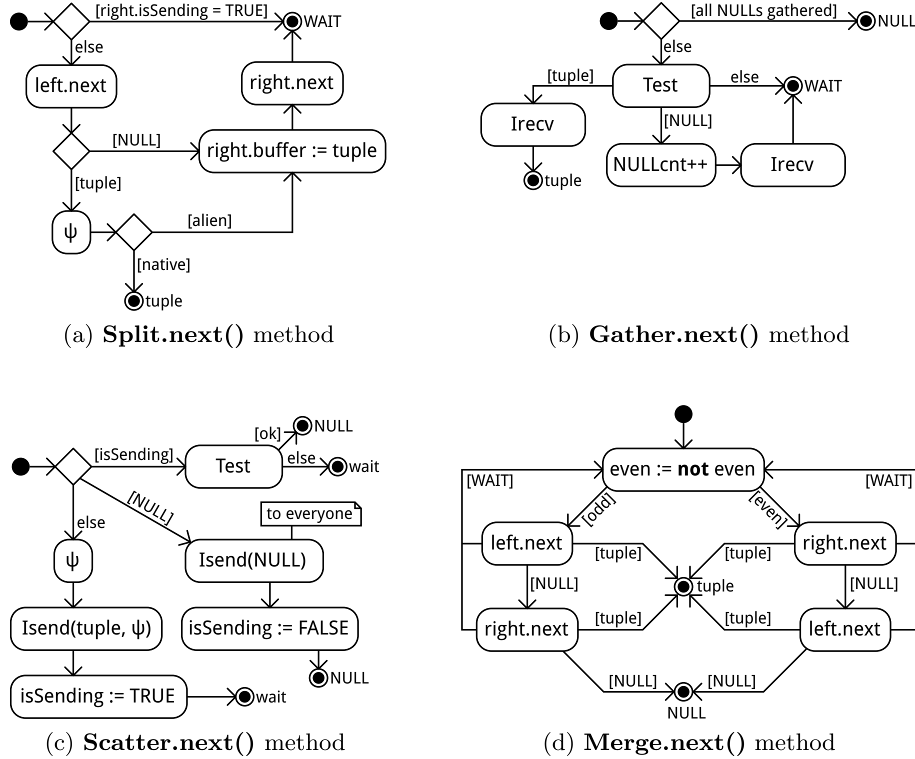
Fig. 8. *Exchange* operator structure

Fig. 9 shows the algorithms for **next()** method of *Exchange* suboperators.

Split (see fig. 9a) decides whether a given tuple is "native" and should be kept on the current node, or it is "alien" and should be sent to appropriate node. "Native" tuples are returned immediately whereas "alien" tuples and NULLs (meaning that scanning of tuples is over) are put into *Scatter*'s buffer for sending to appropriate nodes.

Gather (see fig. 9b) provides receiving of tuples from other processor nodes. Having received a tuple *Gather* starts a receive operation again. NULL value received means that the corresponding node has finished its work. As soon as a NULL is received from every node *Gather* finishes its work.

Scatter (see fig. 9c) sends tuples coming from *Split* to other processor nodes. Non-NULL tuple should be sent to a node with a number calculated by means



(a) **Split.next()** method

(b) **Gather.next()** method

(c) **Scatter.next()** method

(d) **Merge.next()** method

Fig. 9. Algorithms for *Exchange* suboperators

of fragmentation function. In case of NULL value *Scatter* sends NULLs to all the other nodes.

Merge (see fig. 9d) merges tuples from *Gather* and *Split* in an even-odd manner.

The asynchronous MPI methods Isend, Irecv are used by *Exchange* to transmit tuples and the Test method to check whether the appropriate transmission finished.

3.3 Parallelizer

par_Parallelizer subsystem prepares the query plan for parallel execution. The cases in which the *par_Parallelizer* inserts *Exchange* operators into the plan are shown in fig. 10.

The *Join* operation executed independently on multiple nodes will miss some tuples, unless we move the tuples matching the join qualifier to the same node. That is performed by the *Exchange* operators, which are inserted under the *Join* in cases **(a)**, **(b)**, and **(c)**.

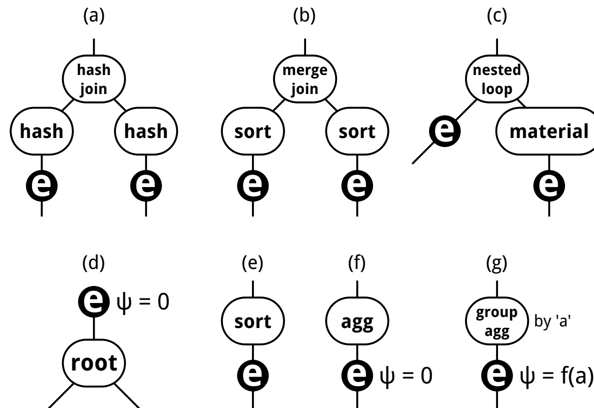


Fig. 10. Parallelizer cases

The root *Exchange* operator **(d)** will mix the order of the tuples, so there is no point in a plan where an *Exchange* is above a *Sort*. In this case **(e)** the Parallelizer inserts the *Exchange* operation a level deeper — below the Sort.

Another case where the tuples should get redistributed is an aggregation, which is performed by the *Agg* operation in PostgreSQL. There are two types of aggregation — simple and grouped. In case of simple aggregation **(f)** the Parallelizer inserts an *Exchange* that would accumulate all the tuples on one node (since they are all needed for some global aggregating function). However, the grouped aggregation **(g)** only needs to have all the tuples of the same group located together.

3.4 Data Manipulation Operations

The algorithms for the exchange subnodes shown above will only work for SELECT statements. In order to support data manipulation queries the execution process needs to become a bit trickier.

When PostgreSQL executes an UPDATE or DELETE query, the resulting tuples coming from the root of the plan have a special hidden attribute — the CTID. It is the address of this tuple inside the storage of PostgreSQL, with this CTID PostgreSQL tells which tuples are to be deleted or updated. The other attributes contain the updated values or, in case of a DELETE, there are no other attributes.

No changes are needed in order for DELETE to work in PargreSQL. But INSERT and UPDATE should have additional logic — since a tuple only needs to be inserted on one node and can move from one node to another during an UPDATE.

There are two places in the PargreSQL code that were changed in order to implement that behaviour for UPDATE queries — the *Split* operator, and the executor.

When *Split* meets an alien tuple, it creates a copy of the tuple and passes one instance to the *Merge* (with the “delete me” bit set inside the CTID) and the other — to the *Scatter* (with the “insert me” bit set inside the CTID). The schematic for tuple flow in *Exchange* is shown in fig. 11.

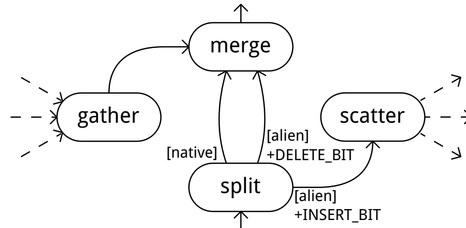


Fig. 11. Tuple flow during an exchange

The executor in its turn checks for these bits and reacts accordingly. So, if the “delete me” bit is set, it performs the delete routines, and if the “insert me” bit is set — the insert routines. If neither bit is set the tuple is considered native to the node, so the executor behaves as in PostgreSQL and updates the tuple in the local storage.

For plain INSERT queries the parallelizer appends an additional condition to the plan, that is equivalent to a `WHERE fragattr % nodes == this_node` clause. With this condition a tuple only gets inserted if it is native to the node. Complex INSERT queries like `INSERT INTO dest SELECT columns FROM src` do not need this additional condition.

3.5 Data Definition Operations

In order to provide data partitioning in PargreSQL we establish an additional storage parameter for PostgreSQL tables, named *fragattr* (fragmentation attribute). An application programmer is to specify an int-valued attribute of a table as *fragattr* on the table's creation. It is equivalent to defining the table's partitioning with $\psi(t) = t.fragattr \bmod n$ fragmentation function, where n is a number of nodes and **mod** denotes the *modulo* operation. The parameter is specified in the WITH clause of the CREATE TABLE query (see fig. 12).

3.6 Load Balancing (Future Work)

We are planning to implement a load balancing scheme proposed in [6]. The scheme is based on partial data replication. The last portion of tuples from each fragment are copied to several other nodes in case the native node would get delayed processing its fragment. When a node manages with its own fragment, and some other node has not, the idle node can start processing the corresponding copy, thus freeing the other node from some work. The “last” portion here

```
create table Person (
    id int,
    name varchar(30),
    gender char(1),
    birth date
) with (fragattr = id);
```

Fig. 12. Table creation in PargreSQL

means the tuples that get read last from the storage due to their physical order or an index.

In PargreSQL the scheme could be represented by a number of tables in a dedicated namespace of the database. The idle node would communicate to the busy one and ask where to start processing of the partial copy from. After that the two nodes would know where to start and to stop and would have rouhgly the same amount of tuples to process.

4 Experimental Evaluation

To evaluate our approach we performed a series of experiments on SKIF-Aurora SUSU supercomputer[1] based on Intel® Xeon 5680 processors and liquid cooling. We executed a query carrying out a natural join of two synthetical tables comprising of 60 mln. and 1.5 mln. records respectively and distributed uniformly among the computer nodes. To form values of tables' fragmentation attributes, a probabilistic model proposed in [6] was used.

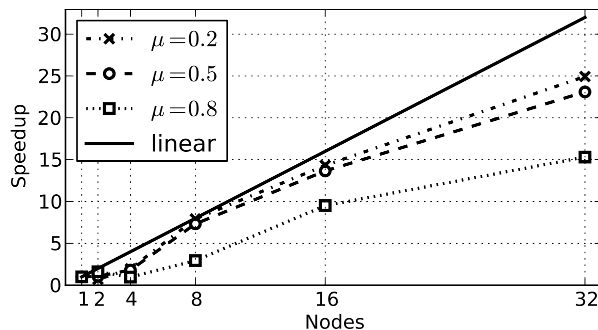


Fig. 13. PargreSQL speedup

Fig. 13 depicts the experimental results. Here μ is the portion of "alien" tuples at every fragment of the relations, e.g. $\mu = 0.75$ means that during the query

[1] http://www.hpcwire.com/hpcwire/2011-12-08/skif_aurora_susu_supercomputer_is_most_energy-efficient_hpc_system_in_russia.html

execution every agent was obliged to send 75% of the tuples stored at the agent's node to other nodes. As we can see PargreSQL demonstrates a quite acceptable speedup.

5 Related Work

The research on adaptation of PostgreSQL for parallel and distributed query processing includes the following.

In [5] authors introduce their work on extending PostgreSQL to support distributed query processing. Several limitations in PostgreSQL's query engine and corresponding query execution techniques to improve performance of distributed query processing are presented.

ParGRES [7] is an open-source database cluster middleware for high performance OLAP query processing. ParGRES exploits intra-query parallelism on PC clusters and uses adaptive virtual partitioning of the database.

GParGRES [4] exploits database replication and inter- and intra-query parallelism to support OLAP queries in a grid. The approach has two levels of query splitting: grid-level splitting, implemented by GParGRES, and node-level splitting, implemented by ParGRES.

In [1] building a hybrid between MapReduce and parallel database is explored. The authors created a prototype named HadoopDB on the basis of Hadoop (communication layer) and PostgreSQL (database layer), that is as efficient as parallel DBMS, but as scalable as MapReduce systems.

Our contribution is embedding partitioned parallelism into PostgreSQL on the basis of the methods for parallel query processing, proposed in [2,3,6,10]. We believe that our approach could be applied to other serial relational open-source DBMSes (e.g. MySQL) to implement their parallel versions.

6 Conclusion

In this paper we have described the design and implementation of PargreSQL parallel DBMS for cluster systems. PargreSQL is based upon PostgreSQL open-source DBMS and exploits partitioned parallelism. This approach is applicable to other open-source relational DBMSes. The results of preliminary experiments show that this scheme is worthy of further development.

As future work we plan to implement load-balancing based upon partial data replication, parallel execution of subqueries and stored procedures, and conduct advanced experiments to analyze PargreSQL performance on complex queries.

References

1. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D.J., Rasin, A., Silberschatz, A.: HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. PVLDB 2(1), 922–933 (2009)

2. DeWitt, D.J., Gray, J.: Parallel Database Systems: The Future of High Performance Database Systems. Commun. ACM 35(6), 85–98 (1992)
3. Graefe, G.: Encapsulation of parallelism in the volcano query processing system. In: Garcia-Molina, H., Jagadish, H.V. (eds.) SIGMOD Conference, pp. 102–111. ACM Press (1990)
4. Kotowski, N., Lima, A.A.B., Pacitti, E., Valduriez, P., Mattoso, M.: Parallel query processing for OLAP in grids. Concurrency and Computation: Practice and Experience 20(17), 2039–2048 (2008)
5. Lee, R., Zhou, M.: Extending PostgreSQL to Support Distributed/Heterogeneous Query Processing. In: Kotagiri, R., Radha Krishna, P., Mohania, M., Nantajeewarawat, E. (eds.) DASFAA 2007. LNCS, vol. 4443, pp. 1086–1097. Springer, Heidelberg (2007)
6. Lepikhov, A.V., Sokolinsky, L.B.: Query processing in a DBMS for cluster systems. Programming and Computer Software 36(4), 205–215 (2010)
7. Paes, M., Lima, A.A.B., Valduriez, P., Mattoso, M.: High-Performance Query Processing of a Real-World OLAP Database with ParGRES. In: Palma, J.M.L.M., Amestoy, P.R., Daydé, M., Mattoso, M., Lopes, J.C. (eds.) VECPAR 2008. LNCS, vol. 5336, pp. 188–200. Springer, Heidelberg (2008)
8. Pan, C.: Development of a parallel dbms on the basis of postgresql. In: Turdakov, D., Simanovsky, A. (eds.) SYRCoDIS. CEUR Workshop Proceedings, vol. 735, pp. 57–61. CEUR-WS.org (2011)
9. Paulson, L.D.: Open source databases move into the marketplace. IEEE Computer, 13–15 (2004)
10. Sokolinsky, L.B.: Organization of Parallel Query Processing in Multiprocessor Database Machines with Hierarchical Architecture. Programming and Computer Software 27(6), 297–308 (2001)