ISSN 0361-7688, Programming and Computer Software, 2015, Vol. 41, No. 6, pp. 350–360.
Original Russian Text © C.S. Pan, M.L. Zymbler, 2015, published in Programmirovanie, 2015, Vol. 41, No. 6.

Encapsulation of Partitioned Parallelism into Open-Source Database Management Systems

C. S. Pan and M. L. Zymbler

South Ural State University, pr. Lenina 76, Chelyabinsk, 454080 Russia
e-mail: pan@susu.ru, mzym@susu.ru

Received May 15, 2014

Abstract—This paper presents an original approach to parallel processing of very large databases by means of encapsulation of partitioned parallelism into open-source database management systems (DBMSs). The architecture and methods for implementing a parallel DBMS through encapsulation of partitioned parallelism into PostgreSQL DBMS are described. Experimental results that confirm the effectiveness of the proposed approach are presented.

DOI: 10.1134/S0361768815060067

1. INTRODUCTION

Big Data are now being one of the main factors that considerably affect the field of data processing. In today's information society, there is a variety of applications (social networks, digital libraries, geographic information systems, etc.) that produce—at 1 TB per day—huge amounts of unstructured data. Cleaning and structuring Big Data result in *very large databases*, which require parallel processing.

Presently, *parallel database management systems* (DBMSs) [1], which are responsible for query processing on multiprocessor and multicore computing systems, are regarded by the scientific community as almost the only effective instrument for storing and processing very large databases. Parallel DBMSs are based on *partitioned parallelism* [2], which assumes fragmenting database relations into horizontal partitions, which, in turn, can be processed independently on different nodes of a cluster computing system.

Presently-available parallel DBMSs (for example, Teradata [3], Greenplum [4], and DB2 Parallel Edition [5]), however, are expensive and often designed for special-purpose hardware and software platforms.

This fact gave reasons for development of *cluster DBMSs* [6], which implement parallel processing of very large databases on computing systems with cluster architecture by means of middleware. The cluster DBMS oriented to online transaction processing (OLTP) processes a large number of short transactions and uses middleware to provide inter-transaction parallelism. Clients connecting to the system are distributed to be serviced by several instances of the DBMS, which increases the availability of the system for a great number of clients. When the cluster DBMS is oriented to online analytical processing (OLAP) and executes complex select queries from very large databases, middleware provides intra-query parallelism by receiving, transforming, and distributing user queries, as well as by merging partial results and transferring them to the user.

MySQL Cluster [7], which supports OLTP, is constructed by connecting a NDB module to MySQL DBMS, which enables data storage in the memory of many distributed computational nodes with allowance for partitioning and replication. The scalability of MySQL Cluster is limited to 48 nodes; databases more than 3 TB in size are not supported. Oracle Real Application Clusters (RAC) [8] stores up to three database (DB) replicas to ensure high data availability and load balancing among cluster nodes; the scalability of this DBMS, however, is limited to 100 computational nodes.

Another OLAP cluster DBMS is implemented in the framework of the ParGRES project [9]. Experiments show high scalability of this system; however, full replication of all DB tables on computational nodes can be regarded as its drawback. vParNDB [10] is a middleware that rewrites queries so that they can be executed in parallel with the use of the computational nodes on which MySQL Cluster is deployed. Experiments show a decent gain in speed with this approach, yet the solutions based on MySQL Cluster inherit the above-mentioned limitations of this DBMS.

Open-source DBMSs [11] are now being a reliable alternative to commercial DBMSs [12]. At the same time, there is a lack of open-source DBMSs that support partitioned parallelism. In [13], a prototype open-source parallel DBMS for cluster computing systems is described. HadoopDB DBMS [14] is an architectural hybrid between the MapReduce para-

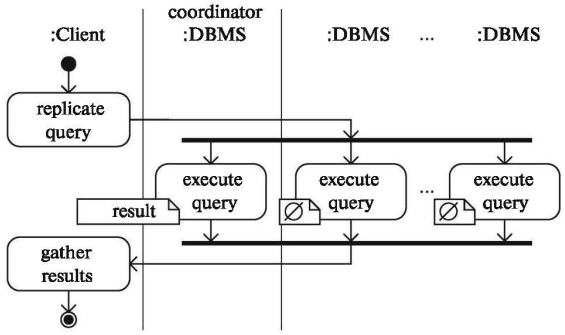

Fig. 1. Query replication.

digm [15] and the technology of relational DBMSs. In HadoopDB, the Hadoop framework [16] implements MapReduce computations and enables the communication infrastructure connecting the cluster nodes on which instances of PostgreSQL are deployed. SQL queries are translated into tasks for the MapReduce environment, which are then sent to the DBMS instances.

The lack of the open-source DBMSs exploiting partitioned parallelism is due to the fact that parallel DBMSs belong to the class of complex system software, while the development of such software is rather expensive and takes a lot of time.

Therefore, the idea of upgrading the original source code of an open-source serial DBMS to construct on its basis a parallel DBMS by encapsulation of partitioned parallelism seems promising. In this case, the upgrade of the source code implies no large-scale modifications of original subsystems, which otherwise would be similar to developing a parallel DBMS from scratch. Commercial parallel DBMSs designed for special-purpose hardware platforms are expected to be more effective then a parallel cluster DBMS constructed by modifying the source code of a serial DBMS. The latter, however, is potentially comparable with commercial DBMSs in terms of scalability, which is achieved by adding new computational nodes into the cluster, still offering a less expensive solution.

This paper presents an approach for parallel processing of very large databases, which is based on the idea of upgrading the original source code of an open-source serial DBMS to construct on its basis a parallel DBMS for cluster computing systems by encapsulation of partitioned parallelism.

The paper is organized as follows. Section 2 presents an approach to developing a parallel DBMS through encapsulation of partitioned parallelism into an open-source serial DBMS. Section 3 describes architecture and methods for implementing a parallel DBMS constructed by applying the proposed approach to PostgreSQL. Results of computational experiments to estimate effectiveness of the proposed methods are given in Section 4. The basic results and directions of further investigations are discussed in the Conclusion section.

2. METHODS FOR ENCAPSULATION OF PARTITIONED PARALLELISM

This section describes a complex of methods for encapsulation of partitioned parallelism into an open-source DBMS.

2.1. Query Replication

Query replication assumes sending a query to a number of DBMS instances, with each instance processing its own DB partition (see Fig. 1).

One of the DBMS instances (for example, the one running on the cluster node with a zero number) is declared to be a *coordinator*. Query execution is organized so that all the instances—except the coordinator—return the empty result, having sent their partial results to the coordinator before execution is complete. The coordinator merges the partial results and sends them to the client. When one of the instances fails to execute the query, the coordinator returns the error as a final result.

2.2. Parallel Execution Plan and the Exchange Operation

Despite the fact that, in the process of query execution, each DBMS instance processes its own DB partition independently, tuple exchange is required to obtain a correct result. For example, when executing the natural join of two relations according to a common attribute, the tuples for which the join condition is fulfilled can be stored in different DB partitions. To handle such situations, a *parallel execution plan* is constructed, which is a serial plan with exchange operations inserted into its certain points.

The *exchange operation* [17] distributes tuples among DBMS instances deployed on different computational nodes of the cluster systems. This operation is implemented by analogy with other operations of physical algebra, which have the iterator interface. The exchange operation has two properties: port and distribution function ψ. The *port* property distinguishes exchange operations from one another in one execution plan: tuples from one point of the plan must fall within the same point of the plan on the other computational node. The *distribution function* $\psi(t)$ calculates the ID of the node on which the tuple t is to be processed. If the tuple t is required on the local node, then it is passed further along the plan; otherwise, it is sent to the node with the number $\psi(t)$.

Figure 2 depicts the structure of the exchange operation (the direction of tuple distribution is shown by arrows). The operations *split*, *scatter*, *gather*, and

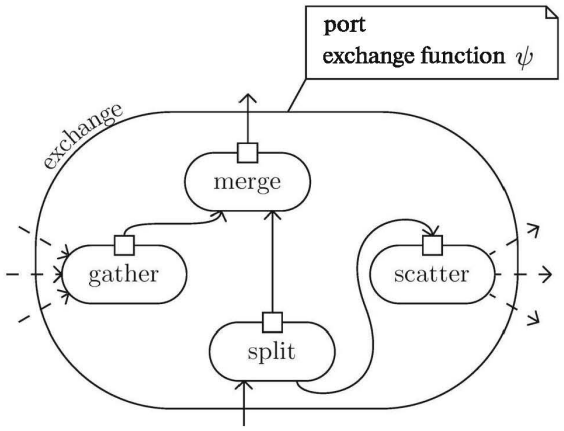

Fig. 2. Architecture of the exchange operation.

merge, which are part of the exchange operation, are also implemented based on the iterator model.

The *split* operation is a binary operation that classifies tuples arriving from the input stream either as "native" or as "alien." The "native" tuples must be processed on the current computational node and are sent to the output buffer of the *split* operation. The "alien" tuples must be processed on the computational nodes other than the current one; these tuples are placed by the *split* operation into the output buffer of the *scatter* operation.

The *scatter* operation is a 0-ary operation that, having extracted tuples from its output buffer, calculates the value of the distribution function for these tuples and sends them to the corresponding computational nodes according to the given number of the exchange port.

The *gather* operation is a 0-ary operation that reads into its output buffer tuples from the specified exchange port for all computational nodes other than the current one.

The *merge* operation is a binary operation that extracts tuples, one by one, from the output buffers of its sons and places them into its own output buffer.

The original query executor of a serial DBMS executes the exchange operation just like any other without any parallelism. Parallelism is achieved owing to the query parallelizer, which inserts *exchange* operations into certain points of the execution plan so that the logic of the query executor yields a correct result.

2.3. Adding Partition Metadata into the DBMS Dictionary

The procedure of relation partitioning depends on the partition function associated with the given relation. For each tuple of the relation, the *partition function* calculates the number of the computational node on which this tuple must be accommodated. To provide the DBMS instance with the information about table partitioning, the DB language should be extended with syntactic constructions, which allow one to define the partition function when executing the command `CREATE TABLE`, while the DBMS dictionary should be supplemented with metadata about relation partitioning.

2.4. Parallel Plan of Data Modification Queries

The above scheme of parallel plan construction is valid in the case of queries for data from the relations the partitions of which are distributed over computational nodes of the cluster system. This scheme, however, should be modified to ensure correct execution of queries for inserting and updating data (see Fig. 3). When executing the `INSERT` query, the tuple must be inserted only into one of the partitions, despite the fact that this query is replicated. When processing the `UPDATE` query, the updated tuples for which the partition function returns a value different from the ID of the current node must be transferred to the corresponding computational node.

2.5. Transparent Porting of Original DBMS Applications

The source code of the user applications written for the original open-source DBMS should undergo minimum modifications to make them capable of running in a parallel DBMS developed on the basis of the former. The transparent porting of the original DBMS applications to the parallel DBMS is implemented by developing an application programmer library, the interface of which is *identical* to that of the original library. The new library replicates queries through multiple invocations of functions from the original library and, while having the interface identical to that of the original library, enables transparent communication between the application and the parallel DBMS. Thus, when switching from the serial DBMS to the parallel one, only the name of the application programmer library is to be changed in the application code.

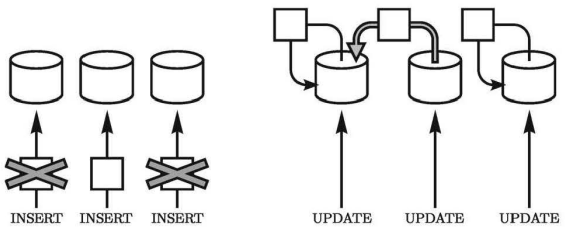

Fig. 3. Tuple insert and tuple update operations.

```
// origfile.c
#include "newfile.c"
typedef struct origstruct {
        ...
        newstruct ns;
} origstruct;

int origfunc() {
        ...
        newfunc();
        ...
}

// newfile.c
typedef struct newstruct {
        ...
} newstruct;

int newfunc() {
    ...
}
```

Fig. 4. Adding fields into the structure and the call statement into the function.

2.6. Soft Modification of the Original DBMS Source Code

A DBMS is a complex system software, the source code of which amounts to tens of thousands of lines. For such systems, the lack of technical discipline in the process of source code modification can destroy the whole project.

The proposed modification technique allows one to minimize changes in the source code by encapsulating the new code in separate subsystems. The modifications in data structures and algorithms are encapsulated in *new* source code files, which are linked to the source code files of the original DBMS.

Figure 4 illustrates the proposed technique. When adding new fields into the original data structure, the type `newstruct`, which contains the new fields, is described in a new file, and a new field of the `newstruct` data type is added into the original structure. When modifying original algorithms, the invocation of a new function `newfunc()`, which is defined in the source code file of a new subsystem, is added into the body of the original function.

3. ENCAPSULATION OF PARALLELISM INTO POSTGRESQL

This section describes the application of the proposed methods to PostgreSQL [18], which is now being one of the most popular open-source DBMSs. This choice is due to the fact that PostgreSQL has open and detailed internal specifications, as well as detailed programming guides. The source code developed by the authors of this paper amounts to about 5000 lines, which took about three person months. The developed parallel DBMS was called PargreSQL [19, 20].

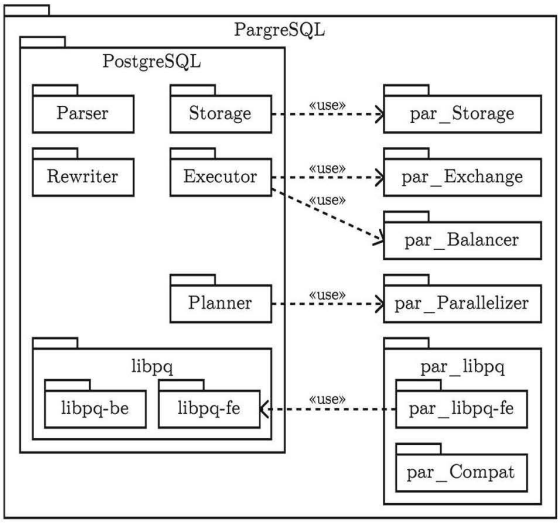

Fig. 5. Architecture of PargreSQL.

3.1. Architecture of PargreSQL

The architecture of PargreSQL parallel DBMS is shown in Fig. 5.

The original DBMS (PostgreSQL) is regarded as one of the subsystems of the parallel DBMS. Below, we briefly describe the structure of PostgreSQL.

The `Parser` subsystem analyzes the syntax of the query. The `Rewriter` subsystem transforms the query according to the rules specified by the administrator (for example, replacing names of representations by their definitions). The `Planner` subsystem constructs and optimizes the execution plan for this query. The `Executor` subsystem executes the plan. The `Storage` subsystem is responsible for low-level storage of data and metadata. The `libpq` library is an

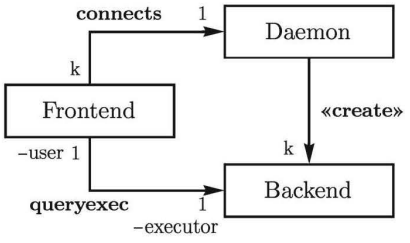

Fig. 6. Client-server model of PostgreSQL.

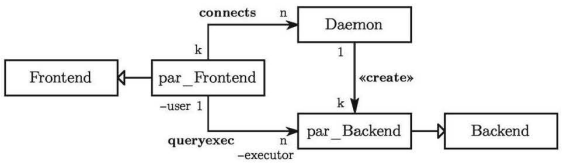

Fig. 7. PargreSQL processes.

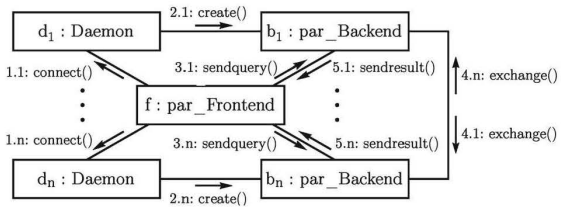

Fig. 8. Client-server interaction in PargreSQL.

API of PostgreSQL, which implements the interfacing protocol between the client (`libpq-fe`) and the server (`libpq-be`).

The session of PostgreSQL involves three interacting processes (see Fig. 6): *Frontend* (client application), *Daemon*, and *Backend*. The daemon handles incoming connections from clients and launches a separate backend for each individual client.

The other subsystems of PargreSQL implement the methods described in Section 2. The `par_libpq` subsystem implements query replication. The `par_Compat` subsystem is a set of macros, which enable transparent porting of applications to the new DBMS. The subsystems `par_Parallelizer` and `par_Exchange` construct the parallel execution plan and the exchange operation, respectively. The `par_Storage` subsystem stores metadata about table partitioning. The `par_Balancer` subsystem is responsible for load balancing in the process of query execution.

Figure 7 shows the model of client-server interactions for PargreSQL.

In contrast to PostgreSQL, the PargreSQL client can interact with two or more servers simultaneously.

The components `par_Backend` and `par_Frontend` are implemented based on the original components `Backend` and `Frontend` of PostgreSQL, respectively. The `Backend` component is extended to provide tuple exchange between DBMS instances, while the `Frontend` component is expanded with the query replication function.

3.2. Implementation of Query Replication

The interaction between the client application and PargreSQL is shown in Fig. 8.

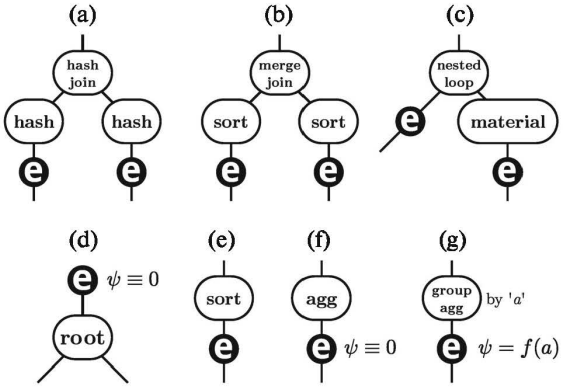

Fig. 9. Insertion of the exchange operation.

The client successively connects to all DBMS daemons, so `par_Backend` is launched on each node. Then, the client queries all of these components in parallel. Having received the query, each `par_Backend` instance executes it for its own DB partition while possibly exchanging data with other instances via the *exchange* operation. Once the query is processed, the client receives the results from the instances and aggregates them.

3.3. Construction of the Parallel Execution Plan

To construct the parallel execution plan, the following technique is used [17]. The post-order traversal of the serial plan tree is performed and an *exchange* operation is inserted under a join operation if the corresponding sub-operation results in a relation partitioned by the attribute that is not used in the join condition. In this case, the partitioning attribute is propagated over the tree from child operations to parent operations. Thus, in each point of the plan, the attribute, which is responsible for partitioning the result of the operation, is known. The cases that require inserting the exchange operation are shown in Fig. 9.

When constructing the execution plan in PostgreSQL, the following types of join operation are used: HashJoin [21], MergeJoin [22], and NestedLoop [23]. In all these cases, the insertion of exchange operations has its own peculiarities.

The *HashJoin* operation assumes creating a hash table for each relation being joined. The *HashJoin* operation has two child operations of the *Hash* type, each of which creates hash tables for their own subtrees. The exchange operation is inserted between the *Hash* operation and its subtree (see Fig. 9a), so the hash table is created upon receiving the tuples sent by other computational nodes, via the exchange operation, to the current computational node.

The *MergeJoin* operation implies presorting the relations being joined. The *MergeJoin* operation has

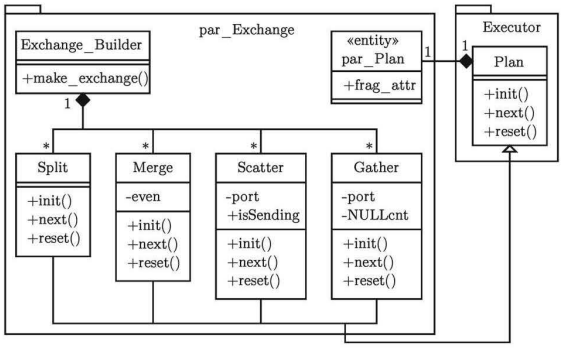

Fig. 10. Class diagram of the exchange operation.

two child operations of the *Sort* type, each of which sorts data of their own sons. The *exchange* operation is inserted between the *Sort* operation and its subtree (see Fig. 9b), so the tuples are sorted upon receiving them from other nodes.

The *NestedLoop* operation assumes that the right relation is fully loaded into memory for its multiple scanning in the inner loop of the join. The right son of the *NestedLoop* operation is the *Material* operation, which loads the results of its subtree into memory. The exchange operation is inserted between the *Material* operation and its subtree (see Fig. 9c), so the tuples are loaded into memory upon receiving them from other computational nodes. Inserting the exchange operation above the *Material* operation will enable sending the tuples of the right relation as many times as there are tuples in the left relation. This leads to a deadlock when the partitions of the left relation on different cluster nodes contain a different number of tuples.

Figure 9d shows the insertion of the exchange operation into the root of the execution plan. In this case, the exchange operation merges partial results, which are obtained by different computational nodes of the cluster, on the coordinator node. The exchange operation inserted into the root of the execution plan has a distribution function that is a constant value equal to the ID of the coordinator node.

To construct a correct parallel execution plan, in addition to inserting the exchange operation for the relation join operation, this operation should also be inserted when processing the operations that sort and aggregate tuples.

The *Sort* operation is used to arrange tuples arriving from the subtree; if the *exchange* operation is placed right above the *Sort* operation, the order of the tuples will be violated and the sorting will not have the expected effect. In such cases (see Fig. 9e), the *exchange* operation is shifted to a lower level, under the *Sort* operation. Thus, exchange precedes sorting, which is correct.

Table 1. Hardware platform used for the experiments

Characteristic	Value
Number of nodes/processors/cores	736/1472/8832
Processor type 3 TB	Intel Xeon X5680height RAM
Peak performance	117 TFlops
Performance LINPACK	100.4 TFlops

The *Agg* operation is used to evaluate aggregate functions without grouping in the queries of the form `select sum(a) from t`. Since this operation must process tuples that are located in all partitions of the relation, to obtain a correct result, the *exchange* operation with the exchange function identical to the ID of the coordinator node is inserted under the *Agg* operation (see Fig. 9f). This ensures sending all tuples to one node and, accordingly, correct evaluation of the aggregate function.

The *GroupAgg* operation is used to evaluate aggregate functions with grouping in the queries of the form `select a, sum(b) from t group by a`. In contrast to the previous case, for the correct execution of this operation, it is sufficient to process each individual group of tuples as a whole. Therefore, to obtain a correct result, the *exchange* operation with the exchange function that depends on the grouping attribute is inserted under the *Agg* operation (see Fig. 9g). This ensures sending all tuples of one group to one node and, accordingly, correct evaluation of the aggregate function for each group.

3.4. Implementation of the Exchange Operation

The *exchange* operation is implemented by introducing new functions and data types into PostgreSQL. Figure 10 shows the *par_Exchange* package, which contains new classes introduced into PostgreSQL.

The classes of this package—*Merge*, *Split*, *Scatter*, and *Gather*—implement sub-operations of the *exchange* operation with the same names. The *Exchange_Builder* class offers a method for constructing the operations mentioned above and for building a whole *exchange* operation from them.

To store the partitioning attribute, the *Plan* class of PostgreSQL, which is an operation in the execution plan, should be modified: an integer attribute `frag_attr` should be added into this class.

The algorithm for implementing the method `next` of the *Split* operation (see Fig. 11) is as follows. The *Split* operation calls the method `next` of the left subtree and applies the distribution function to the resultant tuple received from it. If the distribution function recognizes this tuple as a "native" one (the function value coincides with the ID of the current computational node), then the *Split* operation returns this tuple

Table 2

Number of clients	tpm-C ↓	Number of clients	tpm-C ↓	Number of clients	tpm-C ↓	Number of clients	tpm-C ↓
29	2202531	24	2165413	16	1882353	8	1156626
26	2107183	23	2156250	15	1747572	7	1150684
30	2195122	22	2146341	14	1647058	5	857142
32	2194285	20	2068965	13	1529411	6	847058
27	2189189	19	2054054	12	1358490	4	657534
31	2188235	18	2037735	11	1346938	3	444444
28	2181818	21	2016000	10	1290322	2	328767
25	2173913	17	1961538	9	1270588	1	150000

Table 3

	Cluster/DBMS	Number of nodes/clients		tpm-C
1	SPARC SuperCluster with T3-4 Servers/Oracle Database 11g R2 Enterprise Edition w/RAC w/Partitioning	108	81	30249688
2	IBM Power 780 Server Model 9179-MHB/IBM DB2 9.7	24	96	10366254
3	Sun SPAC Enterprise T5440 Server Cluster/Oracle Database 11g Enterprise Edition w/RAC w/Partitionin	48	24	7646486
	SKIF-Aurora SUSU/PargreSQL	12	29	2202531
4	HP Integrity rx5670 Cluster Itanium2/1.5 GHz-64p/Oracle Database 10g Enterprise Edition	64	80	1184893

as a result. Otherwise, the tuple is placed into the buffer of the right son (*Scatter* operation), the method `next` of the *Scatter* is called, and the *exchange* operation switches to the wait state.

Figure 12 shows the method `next` of the *Merge* operation. The *Merge* operation alternately calls the methods `next` of its left and right sons (operations *Gather* and *Split*). The calls are made while the *exchange* operation is in the wait state. If both sons return the `NULL` value, then the input stream of tuples is exhausted, and the *Merge* operation returns `NULL`. If at least one son returns a tuple, then the *Merge* operation returns that tuple as a result.

The algorithm implementing the method `next` of the *Scatter* operation is shown in Fig. 13. The *Scatter*

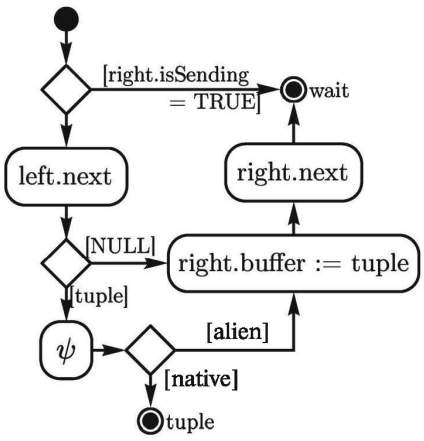

Fig. 11. Method `next` of the *Split* operation.

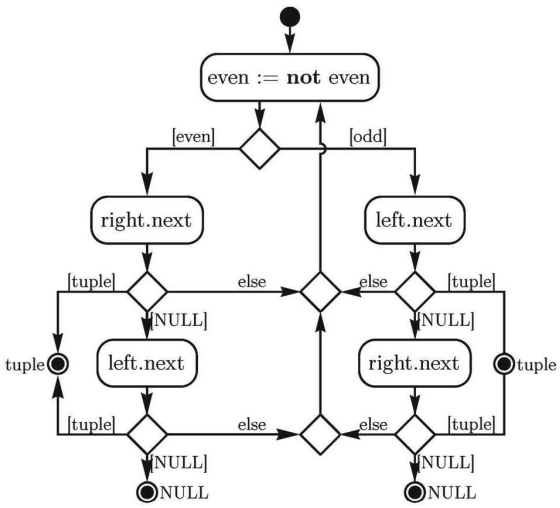

Fig. 12. Method `next` of the *Merge* operation.

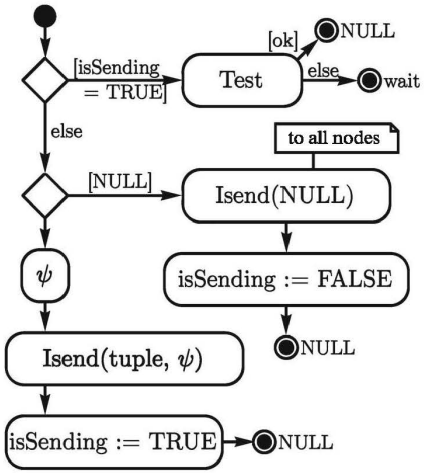

Fig. 13. Method `next` of the *Scatter* operation.

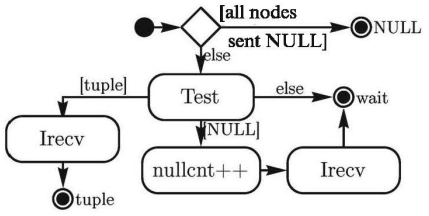

Fig. 14. Method `next` of the *Gather* operation.

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

Fig. 15. Creating a table in PargreSQL.

operation has no children, and calling its method `next` initiates sending the tuple, which is delivered by the parent operation (*Split*), to the computational node the ID of which is obtained by applying the exchange function to this tuple. If, when calling the method `next`, the tuple is not yet sent, then the value `WAIT` is returned.

The *Gather* operation (see Fig. 14) receives tuples from all computational nodes. When calling the method `next` of this operation, the status of the reception operations is checked: if the tuple is received from a certain node, then a new reception operation from this node is initiated, and the tuple received is returned as a result. If, instead of a tuple, the value `NULL` is received from all nodes, then the relation is exhausted, and the method returns `NULL` as a sign of the end of the relation.

To implement the operations *Scatter* and *Gather* in PargreSQL, a *message manager* is developed based on the message passing interface (MPI) [24]. MPI-based messaging is typical for distributed memory systems; however, in the case of PargreSQL, the direct use of the MPI is difficult, since the architecture of this DBMS implies dynamic generation of server processes.

The message manager consists of two—communicator and library—modules. The *communicator* is an MPI program, which runs as an independent daemon in one instance on each computational node. The *library* provides server processes with an interface for connecting to the communicator via shared memory and organizes message communication. The library of the message manager has the following main functions: "initiate data transmission," "initiate data reception," and "check status of transmission/reception"; the interface and semantics of these functions are similar to those of the asynchronous functions `MPI_Isend`, `MPI_Irecv`, and `MPI_Test`, respectively.

3.5. Implementation of Partition Metadata Storage

To implement data partitioning in PostgreSQL, a new attribute *fragattr* is introduced into table metadata. This attribute is of string type and defines the name of the column on which the partition function of the corresponding table depends. When creating a table, the value of this attribute must be set explicitly. The *fragattr* attribute is specified in the query `CREATE TABLE` by using the PostgreSQL construction `WITH` (see Fig. 15).

The attribute with the name specified in the table parameter *fragattr* is used in processing the `UPDATE` and `INSERT` queries to ensure partitioning with the function $\varphi(t) = \text{t.fragattr} \bmod N$, where N is the number of computational nodes in the cluster system and `mod` is the modulo operation.

3.6. Implementation of Data Modification Queries

When processing data insertion queries, it is mandatory to add a select operation with the condition $\psi(t) = i$ (where i is the number of the current node) into the root of the execution plan (see Fig. 16).

Such a condition will discard all the tuples that must be inserted into other computational nodes. Thus, each tuple inserted into the database will fall within only one DB partition.

To transfer modified tuples, the algorithm of the exchange operation should be changed. The new exchange operation (see Fig. 17) will detect the tuples with a modified partitioning attribute and create copies of such tuples.

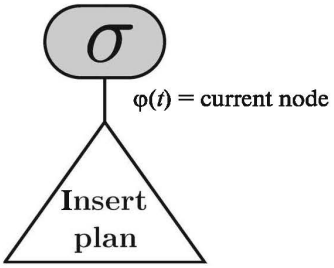

Fig. 16. Parallel query plan for the `INSERT` query.

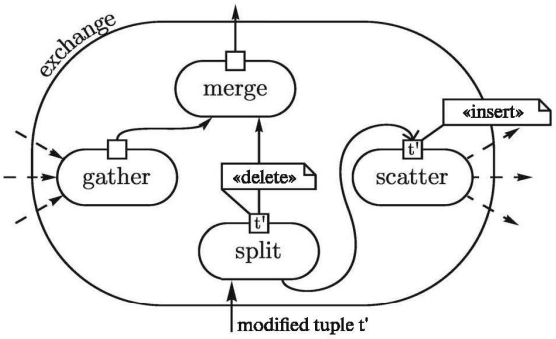

Fig. 17. Tuple stream in the exchange operation for the `UPDATE` query.

One copy labeled "delete me" is passed further along the plan, while the second copy labeled "insert me" is sent to the corresponding node. Thus, the new exchange algorithm makes it possible to relocate the tuples that became "alien" due to modification: if $\varphi(t') \neq \varphi(t)$, then the tuple t on the node $\varphi(t)$ is deleted and the tuple t' is inserted on the node $\varphi(t')$.

4. COMPUTATIONAL EXPERIMENTS

To estimate the effectiveness of the proposed methods and algorithms of PargreSQL, two series of computational experiments were carried out. The first series of experiments investigates the scalability of PargreSQL. In the second series of experiments, the efficiency of PargreSQL is compared with that of presently-available DBMSs with similar characteristics. A SKIF-Avrora YuUrGU supercomputer [25] is used as a hardware platform for the experiments; the characteristics of this supercomputer are presented in Table 1.

4.1. Scalability

Scalability is a measure of parallelization effectiveness for hardware platforms with different numbers of computational nodes. In the case of parallel DBMSs, the main qualitative characteristics of parallelization effectiveness are extendability and speedup, which characterize capabilities of the system to adapt to the increase in the number of cluster nodes and to the rise in the amount of data to be processed. These characteristics are defined as follows [26].

Let A and B be two different configurations of a parallel database machine with a fixed architecture, which differ in the number of processors and devices associated with them (all configurations assume the proportional increase in the number of memory modules and disks), and a test Q be defined. Then, the *speedup* a_{AB}, which is achieved when transferring from the configuration A to the configuration B, is defined as $a_{AB} = t_{QA}/t_{QB}$, where t_{QA} and t_{QB} characterize the time required for A and B to execute the test Q. The speedup parameter allows one to estimate the effectiveness of system expansion for comparable tasks.

Now let a set of tests Q_1, Q_2, ... be defined; these tests quantitatively surpass a certain fixed test Q by a factor of i, where i is the number of the corresponding test and configuration of the parallel database machine A_1, A_2, ..., the degree of parallelism (number of processors) of which exceeds that of a certain minimum configuration A by a factor of j (j is the number of the corresponding configuration). Then, the *extendability* e_{km}, which is achieved when transferring from the configuration A_k to the configuration A_m ($k < m$), is defined as $e_{km} = t_{Q_kA_k}/t_{Q_mA_m}$. The extendability parameter allows one to estimate the effectiveness of system expansion for more complex tasks.

A parallel system is said to be highly scalable if its extendability and speedup are close to linear. *Linear speedup* implies that there is a constant $k > 0$ such that $a_{AB} = kd_B/d_A$ for any configurations A and B (d is the number of processors in the corresponding configuration). *Linear extendability* means that this parameter is equal to one for all configurations of a given system architecture.

In the speedup experiments, PargreSQL executes the query for natural join of two relations according to a common attribute. The sizes of the relations are 300 and 7.5 million tuples, respectively, with the tuples being uniformly distributed over cluster nodes.

Results of these experiments are presented in Fig. 18; it can be seen that speedup is close to linear.

In the extendability experiments, PargreSQL executes the query for natural join of two relations according to a common attribute. The tuples of these relations are uniformly distributed over the cluster nodes. The sizes of the relations are increased proportionally to the increase in the number of the cluster nodes, multiplied by 12 and 0.3 million tuples, respectively.

Results of these experiments are presented in Fig. 19: it can be seen that extendability is close to linear.

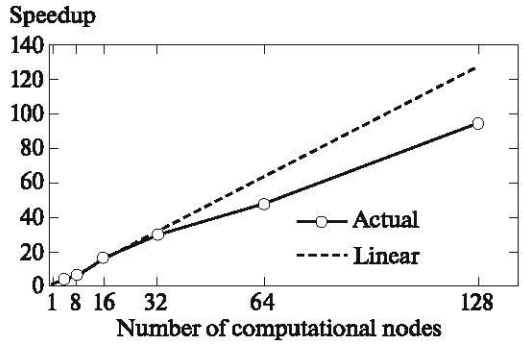

Fig. 18. Speedup.

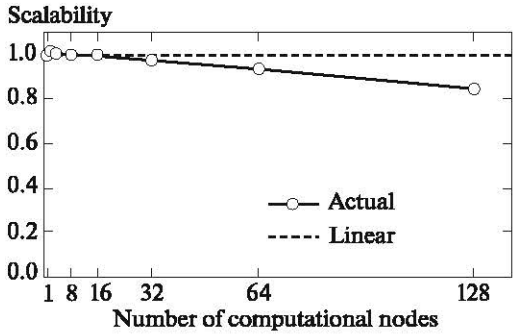

Fig. 19. Scalability of PargreSQL.

Thus, the experimental results show that the scalability of PargreSQL is close to linear.

4.2. TPC Benchmark

The TPC-C benchmark is developed by the Transaction Processing Council (TPC) [27] for measuring the performance of DBMSs in processing a mix of short transactions. This benchmark simulates the activity of a typical warehouse (booking, accounting management, product distribution, etc.). As a performance measure, the TPC-C uses the commercial throughput, which characterizes the number of orders that can be processed per minute. This performance measure is expressed by the maximum speed of transaction execution (tpm-C: transactions-per-minute-C).

In these experiments, from 1 to 30 concurrent clients query PargreSQL, which runs on 12 nodes of a cluster computing system. The DB size is 12 warehouses. Table 2 shows the PargreSQL performance on the TPC-C benchmark in the descending order of tpm-C.

This result lifted PargreSQL to the top five of the TPC-C rating for parallel cluster DBMSs as of September 2013 (see Table 3).

Thus, we can conclude that PargreSQL parallel DBMS is an effective and relatively inexpensive solution for storing and processing very large databases, which possesses high scalability.

CONCLUSIONS

In this paper, the problem of processing very large databases on computing systems with cluster architecture is considered. The approach to solving this problem is proposed, which implies modifying the original source code of an open-source DBMS to construct on its basis a parallel DBMS by encapsulation of partitioned parallelism. The modification of the source code involves as little changes as possible. The parallel DBMS constructed in this way has high scalability. The lag in performance as compared to commercial parallel DBMSs designed for special-purpose hardware and software platforms can be compensated by adding new computational nodes into the cluster while preserving the efficiency of the proposed solution. This approach can be used for parallelizing almost any open-source DBMS (PostgreSQL, MySQL, etc.).

The architecture and methods for implementing PargreSQL parallel DBMS, which is developed through encapsulation of partitioned parallelism into PostgreSQL, are described. The results of computational experiments show that the extendability and speedup characteristics of PargreSQL are close to linear; the experiments also show a rather high performance of PargreSQL on the TPC-C benchmark.

The following directions for further research seem promising.

1. For the open-source serial DBMS, the implementation of data replication based on both partial data mirroring [28] and estimating communication costs of partitioned relations processing [29]; for the parallel DBMS, the development of a load balancing subsystem.

2. For the parallel DBMS constructed by modifying the source code of a serial DBMS, the development of effective methods for controlling the buffer pool, which are oriented to parallel DB systems without resource sharing [30].

3. The adaptation—based on the DMM model [31]—of the proposed methods and algorithms to the cluster systems the nodes of which are equipped with multicore accelerators.

ACKNOWLEDGMENTS

This work was supported by the Ministry of Education and Science of the Russian Federation in the framework of the Federal Targeted Programme for Research and Development in Priority Areas of Development of the Russian Scientific and Technological Complex for 2014–2020 (project no. 14.574.21.0035).

REFERENCES

1. Sokolinsky, L.B., Survey of architectures of parallel database systems, *Program. Comput. Software*, 2004, vol. 30, no. 6, pp. 337–346.
2. Lepikhov, A.V. and Sokolinsky, L.B., Query processing in a DBMS for cluster systems, *Program. Comput. Software*, 2010, vol. 36, no. 4, pp. 205–215.
3. Page, J., A study of a parallel database machine and its performance the NCR/Teradata DBC/1012, *Lect. Notes. Comput. Sci.*, 1992, vol. 618, pp. 115–137.
4. Waas, F.M., Beyond conventional data warehousing—Massively parallel data processing with greenplum database, *Proc. 2nd Int. Workshop on Business Intelligence for the Real-Time Enterprise (BIRTE) in conjunction with VLDB*, Auckland, 2008.
5. Baru, C.K., Fecteau, G., Goyal, A., et al., An overview of DB2 parallel edition, *Proc. ACM SIGMOD Int. Conf. Management of Data*, San Jose, 1995, pp. 460–462.
6. Akal, F., Bohm, K., and Schek, H.-J., OLAP query evaluation in a database cluster: A performance study on intra-query parallelism, *Lect. Notes. Comput. Sci.*, 2002, vol. 2435, pp. 218–231.
7. Ronström, M. and Oreland, J., Recovery principles in MySQL Cluster 5.1, *Proc. 31st Int. Conf. Very Large Data Bases*, Trondheim, 2005, pp. 1108–1115.
8. Pruscino, A., Oracle RAC: Architecture and performance, *Proc. ACM SIGMOD Int. Conf. Management of Data*, San Diego, 2003, p. 635.
9. Paes, M., Lima, A.A.B., Valduriez, P., and Mattoso, M., High-performance query processing of a real-world OLAP database with ParGRES, *Lect. Notes. Comput. Sci.*, 2008, vol. 5336, pp. 188–200.
10. Ngamsuriyaroj, S. and Pornpattana, R., Performance evaluation of TPC-H queries on MySQL Cluster, *Proc. 24th IEEE Int. Conf. Advanced Information Networking and Applications Workshops (WAINA)*, Perth, 2010, pp. 1035–1040.
11. Evdoridis, T. and Tzouramanis, T., A generalized comparison of open source and commercial database management systems, in *Database Technologies: Concepts, Methodologies, Tools, and Applications*, IGI Global, 2009, pp. 294–308.
12. Paulson, L.D., Open source databases move into the marketplace, *Computer*, 2004, vol. 37, no. 7, pp. 13–15.
13. Gavrish, E.V., Koltakov, A.V., Medvedev, A.A., and Sokolinsky, L.B., Open-source parallel DBMS for cluster computing systems, *Vestn. YuUrGU, Ser. Vychisl. Mat. Informatika*, 2013, vol. 2, no. 3, pp. 81–91.
14. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D.J., et al., HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads, *Proc. VLDB Endowment*, 2009, vol. 2, no. 1, pp. 922–933.
15. Dean, J. and Ghemawat, S., MapReduce: Simplified data processing on large clusters, *Commun. ACM*, 2008, vol. 51, no. 1, pp. 107–113.
16. White, T., *Hadoop: The Definitive Guide*, O'Reilly Media, 2009.
17. Sokolinsky, L.B., Organization of parallel query processing in multiprocessor database machines with hierarchical architecture, *Program. Comput. Software*, 2001, vol. 27, no. 6, pp. 297–308.
18. Stonebraker, M. and Kemnitz, G., The POSTGRES: Next-generation database management system, *Commun. ACM*, 1991, vol. 34, no. 10, pp. 78–92.
19. Pan, C.S., Development of a parallel DBMS on the basis of PostgreSQL, *Proc. 7th Spring Researchers Colloquium on Databases and Information Systems (SYRCoDIS)*, 2011, pp. 57–61.
20. Pan, C.S. and Zymbler, M.L., Taming elephants, or how to embed parallelism into PostgreSQL, *Lect. Notes. Comput. Sci.*, 2013, vol. 8055, pp. 153–164.
21. Zhou, J., Hash join, *Encyclopedia of Database Systems*, Liu, L. and Ozsu, M.T., Eds., Springer US, 2009, pp. 1288–1289.
22. Zhou, J., Nested loop join, *Encyclopedia of Database Systems*, Liu, L. and Ozsu, M.T., Eds., Springer US, 2009, p. 1895.
23. Zhou, J., Sort-merge join, *Encyclopedia of Database Systems*, Liu, L. and Ozsu, M.T., Eds., Springer US, 2009, pp. 2673–2674.
24. Gropp, W., MPI 3 and beyond: Why MPI is successful and what challenges it faces, *Lect. Notes. Comput. Sci.*, 2012, vol. 7490, pp. 1–9.
25. Moskovskii, A.A., Perminov, M.P., Sokolinsky, L.B., Cherepennikov, V.V., and Shamakina, A.V., Study of performance of the supercomputer family 'SKIF Aurora' on industrial problems, *Vestn. YuUrGU, Ser. Mat. Model. Program.*, 2010, vol. 211, no. 35, pp. 66–78.
26. Sokolinsky, L.B., *Parallel'nye sistemy baz dannykh* (Parallel Database Systems), Moscow: Mosk. Gos. Univ., 2013.
27. Nambiar, R.O., Poess, M., Masland, A., *et al.*, TPC benchmark roadmap 2012, *Lect. Notes. Comput. Sci.*, 2013, vol. 7755, pp. 1–20.
28. Kostenetskii, P.S., Lepikhov, A.V., and Sokolinsky, L.B., Technologies of parallel database systems for hierarchical multiprocessor environments, *Autom. Remote Control*, 2007, vol. 68, no. 5, pp. 847–859.
29. Gubin, M.V. and Sokolinsky, L.B., About communication cost estimation for processing of partitioned relation with uniform distribution, *Vestn. YuUrGU, Ser. Vychisl. Mat. Informatika*, 2013, vol. 2, no. 1, pp. 33–43.
30. Sokolinsky, L.B., Effective buffer management replacement algorithm for parallel shared-nothing database system, *Vychisl. Metody Program.*, 2002, vol. 3, no. 1, pp. 113–130.
31. Kostenetskii, P.S. and Sokolinsky, L.B., Simulation of hierarchical multiprocessor database systems, *Program. Comput. Software*, 2013, vol. 39, no. 1, pp. 10–24.

Translated by Yu. Kornienko