ISSN 0361-7688, Programming and Computer Software, 2010, Vol. 36, No. 4, pp. 205–215. © Pleiades Publishing, Ltd., 2010.
Original Russian Text © A.V. Lepikhov, L.B. Sokolinsky, 2010, published in Programmirovanie, 2010, Vol. 36, No. 4.

Query Processing in a DBMS for Cluster Systems

A. V. Lepikhov and L. B. Sokolinsky

South Ural State University, pr. im. V.I. Lenina 46, Chelyabinsk, 454080 Russia
e-mail: lepihov@gmail.com; sokolinsky@acm.org

Received June 20, 2009

Abstract—The paper is devoted to the problem of effective query execution in cluster-based systems. An original approach to data placement and replication on the nodes of a cluster system is presented. Based on this approach, a load balancing method for parallel query processing is developed. A method for parallel query execution in cluster systems based on the load balancing method is suggested. Results of computational experiments are presented, and analysis of efficiency of the proposed approaches is performed.

DOI: 10.1134/S0361768810040031

1. INTRODUCTION

Currently, there exist a number of database systems providing parallel query processing. Database systems designed for processing OLAP queries are used for controlling petabyte data arrays. For example, the Greenplum DBMS based on the MapReduce technology [1] performs deep analysis of 6.5 PB data on 96-node cluster in the eBay company. The DBMS Hadoop processes 2.5 PB data on a cluster consisting of 610 nodes for the popular web service facebook. There are several commercial parallel DBMSs for parallel processing of OLTP queries, the most famous among which are Teradata, Oracle Exadata, and DB2 Parallel Edition.

Currently, studies in this field are carried out in the direction of the DBMS self-tuning [2], load balancing and related problem of data placement [3], optimization of parallel queries [4], and efficient use of modern many-core processors [5, 6].

One of the most important tasks in parallel DBMSs is load balancing. In the classical work [7], it was shown that skews arising in the execution of queries in parallel database systems without resource sharing can result in almost complete degradation of system performance.

A solution of the load balancing problem for systems without resource sharing based on replication was suggested in [8]. This solution reduces overheads of the data transmission through the net in the course of load balancing. However, this approach is applicable in a quite narrow context of spatial databases in a specific segment of range queries. In [3], load balancing problem is solved by way of partial redistribution of data before query execution. This approach reduces the total number of data transfers between the computational nodes in the course of query execution; however, it imposes serious requirements on the rate of interprocessor communications.

In this work, we suggest a new method for parallel query processing, which is based on the original approach to database placement called partial mirroring. This method solves tasks of efficient query processing and load balancing in cluster-based systems.

The paper is organized as follows. In Section 2, a method for parallel query processing in DBMSs for cluster-based systems is described. In Section 3, a strategy of data placement on cluster systems and a load balancing algorithm are suggested. Section 4 presents results of computational experiments showing practical significance of the methods and algorithms proposed in this work. The last section gives summary of the basic results obtained and conclusions, as well as discusses directions of future research.

2. ORGANIZATION OF PARALLEL QUERY PROCESSING

Parallel query processing in relational database systems is based on partitioned parallelism (Fig. 1). This form of parallelism suggests partitioning of the relation that is an argument of a relational operation among the disks of the multiprocessor system. The way the partitioning is done is determined by a fragmentation function ϕ, which, for each tuple of the relation, calculates the number of the processor node on which this tuple is to be placed. The query is executed in parallel in all processor nodes as a set of parallel agents [9]. Each agent processes a separate fragment of the relation and generates partial query result. The results obtained by the agents are merged into the resulting relation. Although each parallel agent in the course of query execution independently processes its own fragment of the relation, transfers of tuples are required in order to obtain a correct result. To organize such

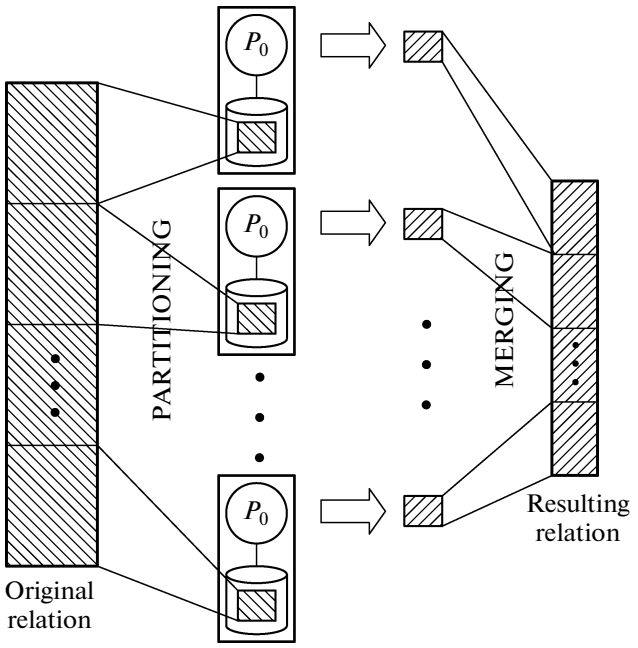

Fig. 1. Partitioned parallelism.

transfers, the operator **exchange** is inserted in the corresponding places of the query plan tree [10].

The **exchange** operator is identified in the plan tree by its number and distribution function ψ, which, for each input tuple, calculates the number of the computational node where this tuple is to be processed. Operator **exchange** performs transfers of tuples between parallel agents using communication channels, which are identified by pairs consisting of the node and port numbers. For the node number, the number of the parallel agent is used, and for the port number, the number of the **exchange** operator.

Let us describe the general scheme of the organization of parallel query processing in a parallel DBMS for cluster systems. We assume that the computing system is a cluster consisting of N computational nodes (Fig. 2). We also assume that each relation of the database used in the query processing is partitioned over all nodes of the computing system. In accordance with this scheme, the processing of an SQL query consists of three stages.

On the first stage, the user passes the SQL query to a dedicated host machine, where it is translated into some sequential physical plan. On the second stage, the sequential physical plan is transformed into a parallel plan consisting of parallel agents. This is achieved by inserting the **exchange** operator into appropriate places of the query tree.

On the third stage, the parallel agents are spread from the host machine to the corresponding computational nodes, where they are interpreted by the query executor. The results of the agent execution are merged by the root **exchange** operator on the zero node, from which they are sent to the host machine. The role of the host machine may be played by any node of the computational cluster.

Let us illustrate the processing of a query in a parallel database system by the following example. Suppose that we need to calculate a natural join $Q = R \bowtie S$ of two relations R and S with respect to some common attribute Y. Let relation R be partitioned by the join attribute between two computational nodes CN_0 and CN_1 as two fragments R_0 and R_1,

$$R = R_0 \cup R_1, \quad R_0 \cap R_1 = \emptyset,$$
$$\pi_Y(R_0) \cap \pi_Y(R_1) = \emptyset,$$

where π is a projection operation.

Let ϕ: $R \longrightarrow \{0, 1\}$ be a fragmentation function for relation R. We have

$$\forall u, v \in R: u.Y = v.Y \Rightarrow \phi(y) = \phi(v).$$

Here, $u.Y$ and $v.Y$ denote the values of attribute Y in tuples u and v, respectively. Then, there exists a function ϕ_Y: $\pi_Y(R) \longrightarrow \{0, 1\}$ such that

$$\forall r \in R: \phi(r) = \phi_Y(r.Y).$$

Let relation S be partitioned with respect to some other attribute Z between the same computational nodes CN_0 and CN_1 as two fragments S_0 and S_1,

$$S = S_0 \cup S_1, \quad S_0 \cap S_1 = \emptyset,$$
$$\pi_Z(S_0) \cap \pi_Z(S_1) = \emptyset, \ Z \neq Y.$$

Then, the sequential physical plan of query Q and the corresponding parallel plan will have the form shown in Fig. 3. The parallel plan in this case includes two agents A_0 and A_1, which are executed on the computational nodes CN_0 and CN_1, respectively. In order that the natural join be executed correctly in the parallel plan, it is required to insert the **exchange** operator between the **join** and **scan** operators for relation S. In this case, the distribution function for e_1 will have the form

$$\psi_1(s) = \phi_Y(s.Y).$$

To collect tuples of the resulting relation on the node of agent A_0 after operator **join**, we add one more **exchange** operator e_2, the distribution function of which has the form

$$\psi_2(x) = 0.$$

3. DATA PLACEMENT AND LOAD BALANCING

3.1. Data Partitioning and Segmentation

The database distribution in a cluster computing system is specified as follows [11]. Each relation is par-

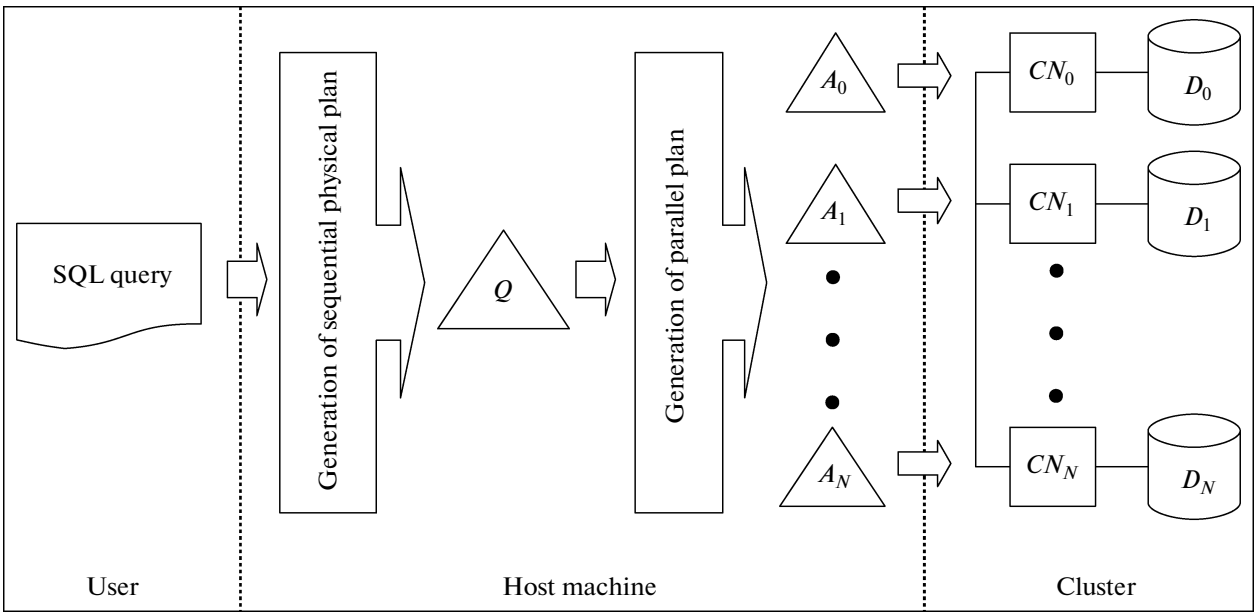

Fig. 2. Scheme of query execution in a parallel DBMS for cluster-based systems. Q is a sequential physical plan, A_i is a parallel agent, and CN_i is a computing node.

titioned into nonintersecting horizontal fragments, which are placed on different computational nodes. It is assumed that the tuples of a fragment are ordered in some way and that this order is fixed for each query and determines the order the tuples are read in the operation of scanning the fragment. This order is said to be *natural*. In practice, the natural order may be determined by the physical order of the tuples or the index.

On the logical level, each fragment is divided into a sequence of segments of fixed length. The segment length is measured in tuples and is an attribute of the fragment. The partitioning into segments is performed in accordance with the natural order and always begins with the first tuple. In accordance with this, the last segment may occur incomplete.

The number of segments of fragment F is denoted as $S(F)$ and can be calculated by the formula

$$S(F) = \left\lceil \frac{T(F)}{L(F)} \right\rceil.$$

Here, $T(F)$ denotes the number of tuples in fragment F, and $L(F)$ is segment length for fragment F.

3.2. Data Replication

Let a fragment F_0 be located on a disk $d_0 \in \mathfrak{D}$ of a cluster system. We assume that each disk $d_i \in \mathfrak{D}$ $(i > 0)$ contains a *partial replica* F_i, which includes some subset (possibly, empty) of tuples of fragment F_0.

The least unit of data replication is segment. The length of a replica segment always coincides with the segment length of the fragment being replicated:

$$L(F_i) = L(F_0), \quad \forall d_i \in \mathfrak{D}.$$

The length of replica F_i is given by the replication factor

$$\rho_i \in \mathbb{R}, \quad 0 \le \rho_i \le 1,$$

which is an attribute of replica F_i and is calculated by the formula

$$T(F_i) = T(F_0) - \lceil (1 - \rho_i) S(F_0) \rceil L(F_0).$$

The natural order of tuples of replica F_i is determined by the natural order of tuples of fragment F_0. The number N of the first tuple of replica F_i is given by

$$N(F_i) = T(F_0) - T(F_i) + 1.$$

For an empty replica F_i, we have $N(F_i) = T(F_0) + 1$, which corresponds to the "end of file" position.

The above-described mechanism of data replication allows us to use in cluster systems a simple and efficient method of load balancing, which is described in Section 3.3.

3.3. Load Balancing Method

3.3.1. Parallel agent operation scheme. Let a query Q with n input relations be given. Let $\mathfrak{Q}$ be a parallel plan of query Q. Each agent $Q \in \mathfrak{Q}$ has n input streams $s_1, \ldots, s_n$. Each stream s_i $(i = 1, \ldots, n)$ is determined by the following four parameters:

(1) f_i, pointer to the fragment of the relation;

(2) q_i, the number of segments in the interval to be processed;

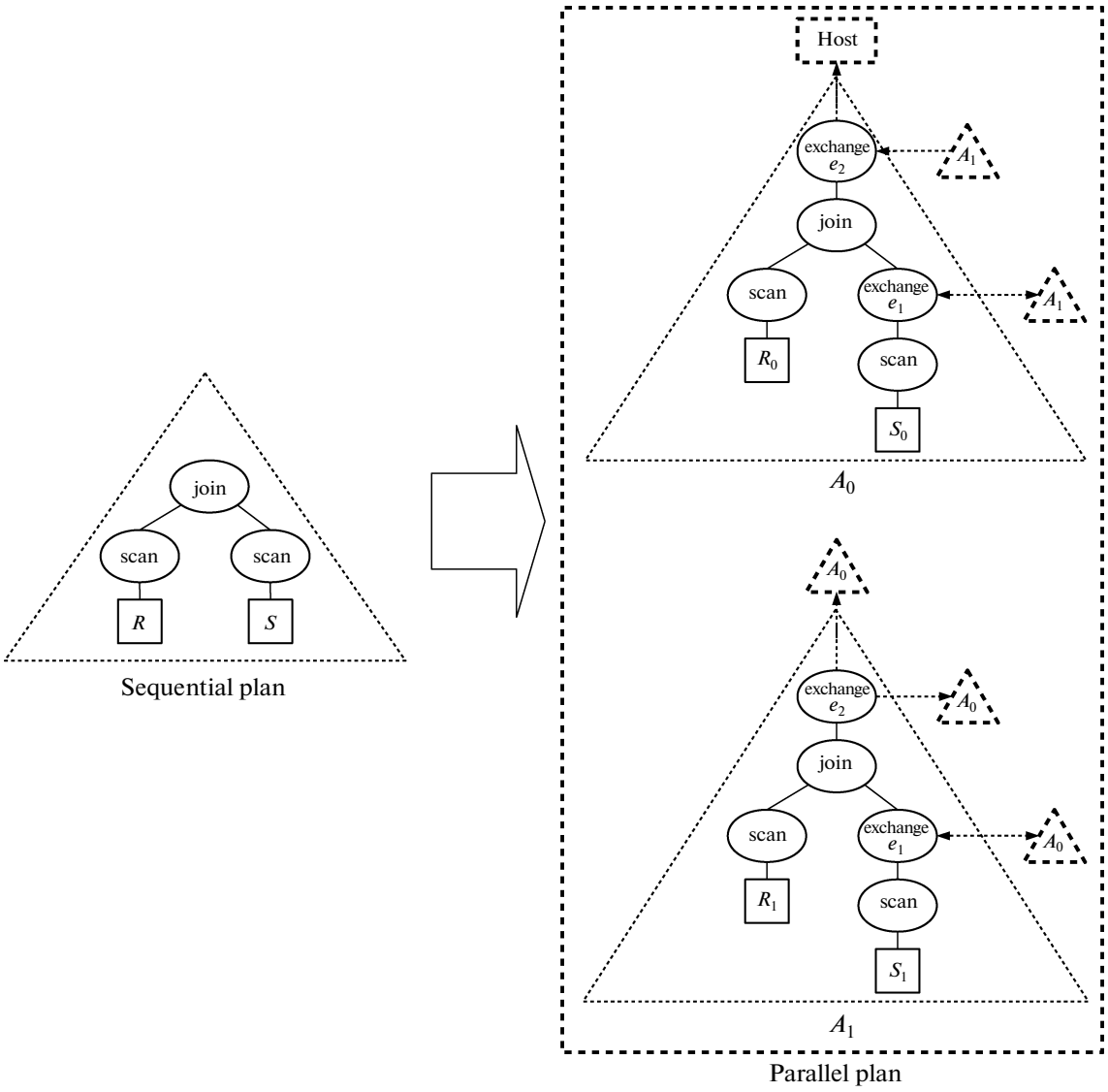

Fig. 3. Sequential and parallel plans for query $Q = R \bowtie S$.

(3) b_i, the number of the first segment in the interval being processed; and

(4) a_i, the balancing indicator (1 if the balancing is permitted and 0 if the balancing is not permitted).

Figure 4 shows an example of a parallel agent with two input streams.

A parallel agent Q may occur in one of the two—*active* or *passive*—states. In the *active* state, Q successively reads and processes tuples from all input streams. In the course of the processing, values of parameters q_i and b_i are dynamically changed for all $i = 1, ..., n$. In the *passive* state, agent Q performs no actions. On the initial stage of query execution, the agent is initialized, which results in the determination of the parameters of all input streams. In each fragment, only those segments are processed that belong to the interval determined by the parameters of the stream associated with the given fragment. After all necessary segments in all input streams have been processed, the agent turns to the passive state.

3.3.2. Load balancing algorithm. After execution of the parallel plan of the query, some agents complete their operations and turn to the passive state, whereas others continue processing of the intervals assigned to them. In this way, a *skew* situation arises. We suggest the following load balancing algorithm based on the data replication [11].

Suppose that we have a situation where a parallel agent $\bar{Q} \in \mathfrak{Q}$ has finished processing of the segments assigned to it in all input streams and turned to the passive state, whereas an agent $\tilde{Q} \in \mathfrak{Q}$ is still continuing to process its data portion, so that load balancing is

required. The idle agent $\bar{Q}$ is called a *leader*, and the overloaded agent $\tilde{Q}$, an *outsider*. In this situation, a procedure of load balancing between the leader $\bar{Q}$ and outsider $\tilde{Q}$ is performed, which consists in transfer of a part of unprocessed segments from agent $\tilde{Q}$ to agent $\bar{Q}$. The scheme of the load balancing algorithm is depicted in Fig. 5 (in a C-like pseudocode). In load balancing, an external (with respect to this procedure) *balancing function* **Delta** is used. It calculates the number of segments of the corresponding input stream passed from the outsider $\tilde{Q}$ to the leader $\bar{Q}$.

The efficient use of the above-described load balancing algorithm requires solving the following two problems.

1. If there are idling agents–leaders, it is required to choose some outsider that will be an object of balancing. The way the outsider is selected is called a strategy of outsider selection.

2. It is necessary to determine how many unprocessed data segments are to be transferred from the outsider to a leader. The function that calculates this number is called the balancing function.

3.3.3. Outsider selection strategy. In this section, we propose an optimistic strategy of outsider selection. This strategy is based on the use of the replication mechanism described in Section 3.2.

Consider a multiprocessor computing system T. Let $\mathfrak{Q}$ be a parallel plan of query $\mathbb{Q}$ and Ψ be the set of nodes of the computing system T on which the parallel plan $\mathfrak{Q}$ is executed. Suppose that, in the process of query execution, an agent–leader $\bar{Q} \in \mathfrak{Q}$ located on a node $\bar{\psi} \in \Psi$ completed its operation at some time moment and turned to the passive state. Out of the set of agents of the parallel plan $\mathfrak{Q}$, it is required to choose some agent–outsider $\tilde{Q} \in \mathfrak{Q}$ ($\tilde{Q} \neq \bar{Q}$) to which the agent–leader $\bar{Q}$ will help. We assume that agent $\tilde{Q}$ is located on node $\tilde{\psi} \in \Psi$ and that $\tilde{\psi} \neq \bar{\psi}$. Let $\tilde{\rho}$ denote the replication factor, which determines the length of replica $\tilde{f}_i$ for fragment $\bar{f}_i$.

To select an agent–outsider, a rating mechanism is used. To each agent of the parallel plan, in the course of load balancing, a real number—rating—is assigned. The agent with the maximum positive rating is selected to be the outsider. If there are no such agents, the vacant agent $\bar{Q}$ simply completes its operation. If several agents have maximum positive rating, then one of them that was not subjected to load balancing for a longer time is selected to be the outsider.

To calculate the rating, the optimistic strategy uses rating function $\gamma: \mathfrak{Q} \longrightarrow \mathbb{R}$ of the form

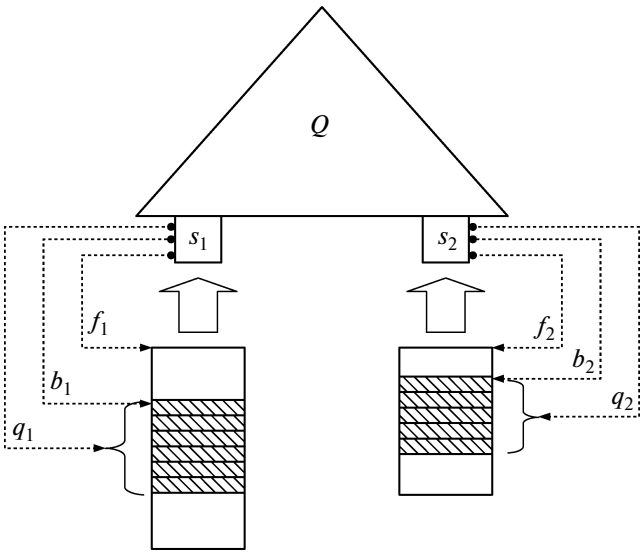

Fig. 4. Parallel agent with two input streams.

$$\gamma(\tilde{Q}) = \tilde{a}_i sgn(\max_{1 \le i \le n} \tilde{q}_i - B)\tilde{\rho}\vartheta\lambda.$$

Here, λ is a positive weight coefficient regulating the effect of the replication factor on the rating; B is a nonnegative integer specifying the lower bound of the number of segments to be transferred in load balancing; $\tilde{\rho}$ is the replication factor determining the length of replica of agent $\bar{Q}$ for fragments of agent $\tilde{Q}$; and ϑ is a static coefficient, which takes one of the following values:

• (-1) if the number of unprocessed data segments of the agent–outsider is less than the lower bound B;

• 0 if the agent–outsider did not take part in the load balancing; or

• a *positive integer* equal to the number of successful balancing operations in which this outsider took part.

3.3.4. Balancing function. For each stream $\tilde{s}_i$ of an agent–outsider $\tilde{Q}$, the balancing function Δ determines the number of segments to be passed to the agent–leader $\bar{Q}$ for the processing. In the simplest case, we may set

$$\Delta(\tilde{s}_i) = \left\lceil \frac{\min(\tilde{q}_i, S(\tilde{f}_i)\tilde{\rho})}{N} \right\rceil,$$

where N depends on the number of parallel agents taking part in the query processing.

Function $S(\tilde{f}_i)$ introduced in Section 3.1 calculates the number of segments of fragment $\tilde{f}_i$. Thus, function Δ splits unprocessed segments of fragment $\tilde{f}_i$ into N

```
/* Procedure of load balancing between parallel agents Q̄ (leader) and Q̃ (outsider). */
ū = Node(Q̃); // pointer to the node of agent Q̄.
pause Q̃; // Turn Q̃ to the passive state.
for (i = 1; i<=n; i++) {

  if(Q̃.s[i].a == 1) {
    f_i = Q̃.s[i].f; // fragment processed by agent Q̃.
    r̃_i = Re(f_i, ū); // replica of fragment f̃_i on node ū.
    δ_i = Delta(Q̃.s[i]); // the number of transferred segments.
    Q̃.s[i].q− = δ_i;
    Q̄.s[i].f = r_i;
    Q̄.s[i].b = Q̃.s[i].b + Q̃.s[i].q;
    Q̄.s[i].q = δ_i;
  } else
      print(Balancing is not permitted);
};
activate Q̃ // Turn Q̃ to the active state.
activate Q̄ // Turn Q̄ to the active state.
```

Fig. 5. Load balancing algorithm for two parallel agents.

intervals and transfers one of them from $\tilde{Q}$ to $\bar{Q}$. Such a balancing function ensures uniform distribution of the load of agent $\tilde{Q}$ among newly appeared agents–leaders.

4. COMPUTATIONAL EXPERIMENTS

To study the proposed load balancing method, we carried out three series of computational experiments on the basis of the developed prototype of parallel DBMS "Omega" [12]. The experiments were aimed at achieving the following objectives:

- to obtain an estimate of optimal values of parameters for the proposed load balancing algorithm;
- to study effect of the load balancing algorithm on the DBMS scalability; and
- to perform analysis of efficiency of the load balancing algorithm for various values of skews.

4.1. Join Operation by Hashing in Operative Memory

To study the load balancing algorithm in the prototype of the parallel DBMS "Omega," we used the algorithm of θ-join by hashing in the operative memory. This algorithm is used in modern DBMSs for query processing in the cases where one of the relations joined can be completely placed to the operative memory.

Modern cluster systems have great total operative memory. For example, in the TOP50 rating [13], the average total operative memory of a cluster system exceeds one terabyte. In the framework of applications of database systems, this means that, when using partitioned parallelism, a relation of size less than one terabyte can be partitioned over processor nodes such that each fragment of this relation can completely be placed in the operative memory. In binary relational operations, one of the relations is often much greater than the other. As applied to the join relation, the lesser relation is called a *reference* relation, and the greater one, a *tested* relation. In real applications of the databases, it seldom happens that the size of a reference relation is greater than one terabyte. Therefore, join algorithms by hashing are of practical significance for cluster systems.

An algorithm performing a relational join operation by hashing in the operative memory can be divided into two stages. On the *construction stage*, the join operation is initialized, and a hash table of the binary relation R is constructed in the operative mem-

Parameters of the execution environment

Parameter	Value
Parameters of computer system	
The number of processor nodes	128
Processor type	Intel Xeon E5472 (4 cores 3.0 GHz each)
Operative memory	8 (Gb per node)
Disk memory	120 (Gb per node)
Type of communication net	InfinitiBand (20 Gb/s)
Operating system	SUSE Linux Enterprise Server 10
Database parameters	
Relation size R	60 million records
Relation size S	1.5 million records
Parameters of query	
Balancing indicator for relation R	0 (balancing is permitted)
Balancing indicator for relation S	1 (balancing is not permitted)

ory. In so doing, the following operations are performed by each processor node.

1. The related fragment of relation R is scanned tuple-by-tuple. For each tuple, the value of the distribution function ψ_R, which produces the number of the node–receiver of this tuple, is calculated. The tuple is transferred to the node–receiver.

2. A hash table is constructed in the operative memory by means of the hashing function $h(t)$, and the tuples transferred to this node on step 1.

On the *comparison stage*, the tested relation S and the join with the tuples of relation R are processed. In so doing, the following operations are performed by each processor node.

1. The related fragment of relation S is scanned tuple-by-tuple. For each tuple, the value of the distribution function ψ_S is calculated. The tuple is transferred to the node–receiver.

2. After receiving the tuple transferred to the node on step 1, the join with the tuples from the hash table constructed on step 2 of the construction stage is performed. The resulting tuple is formed.

Note that, for correct operation of the algorithm, the distribution functions ψ_R and ψ_S should satisfy the condition

$$\forall r \in R,\ \ \forall s \in S\colon \theta(r, s) \Rightarrow \psi_R(r) = \psi_S(s).$$

Figure 6 shows a general form of an agent of the query parallel plan that implements the **MHJ** join operation by hashing in the main memory. The **scan** operator performs scanning of a disk. The **exchange** operators are inserted as the left and right sons of the **MHJ** operator and, in the course of query processing, perform redistribution of tuples coming from the **scan** operators among the processor nodes.

4.2. Parameters of Computational Experiments

For the experiments, we used computing cluster "SKIF Ural" [14]. Basic parameters of the execution environment are presented in the table. Either of relations R and S consists of five attributes, which take nonnegative integer values in the range from 0 through 10 million. In the course of the experiments, natural join of relations R and S with respect to the second attribute was performed by hashing in the operative memory.

In the query execution, R was a reference relation, and S was considered to be a tested relation. The size of relation R was selected such that any fragment of the relation could completely be placed to the operative memory of a computing node. The operation of forming the hash table did not require balancing in view of small size of the reference relation. Therefore, in performing the join operation by hashing in the main memory in the experiments, balancing was carried out only on the stage of processing the tested relation.

To form values of the partitioning attribute of relations R and S, a probabilistic model was used. In accordance with this model, the skew coefficient θ ($0 \le \theta \le 1$) specifies distribution in which, to each fragment, some weight coefficient p_i ($i = 1, \ldots, N$) is assigned by the formula

$$p_i = \frac{1}{i^{\theta} H_N^{(\theta)}}, \quad \sum_{i=1}^{N} p_i = 1,$$

where N is the number of fragments of the relation and $H_N^s = 1^{-s} + 2^{-s} + \ldots + N^{-s}$ is the Nth harmonic number of order s. For example, for $\theta = 0.5$, distribution of the weight coefficients obeys the rule "45–20," in accor-

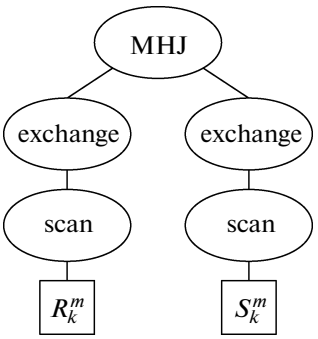

Fig. 6. Parallel plan of query $R \bowtie S$.

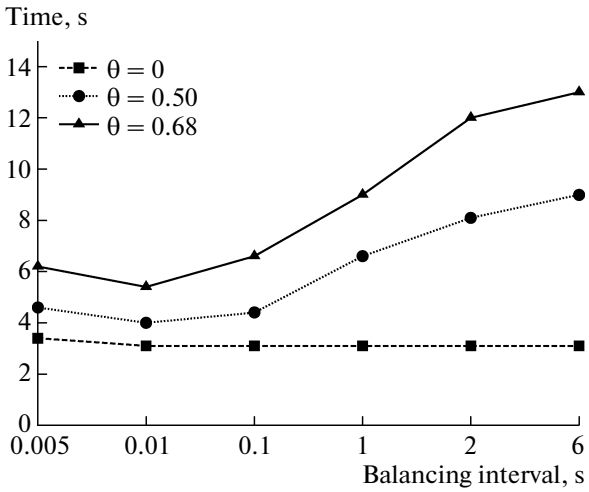

Fig. 7. Dependence of query execution time on the balancing interval ($n = 64$, $\mu = 50\%$).

dance with which 45% of relations of the tuple will be stored in 20% of fragments. In our study, the skew coefficient took values 0, 0.2, 0.5, and 0.68. The value 0.68 corresponds to the rule "80–20", and the value 0.2, to the rule "25–15." The zero skew coefficient corresponds to the uniform distribution.

To form values of the join attribute, "own-or-alien" classification of tuples based on a skew coefficient μ was used. The coefficient μ shows percent ratio of "own" tuples in the fragment. The "own" tuples are processed on the computing node where they are located. The "alien" tuples are transferred to other computing nodes. The distributions of values of the join attribute in each class obey the uniform law. For example, when $\mu = 50\%$, the relation is formed such that each parallel agent passes approximately 50% of tuples from its fragment to the counteragents. In so doing, all counteragents receive about the same number of tuples.

4.3. Study of Load Balancing Parameters

In the first series of experiments, we studied the following load balancing parameters: balancing interval and segment size. The study of the load balancing interval (Fig. 7) shows that the time of query execution almost does not change when the load balancing interval reduces from 0.1 s to 0.005 s. This means that, when the rate of the node balancing operation increases, the overheads have little effect on the total time of query execution. This is achieved owing to the implementation of the load balancing on the basis of asynchronous communication functions. On the other hand, the increase of the load balancing interval from 0.1 s to 6 s greatly reduces efficiency of the balancing procedure. This is explained by the fact that potential balancing capabilities are not completely used: an agent–leader idles for a long time waiting for an agent–outsider. Our experiments showed that the balancing interval equal to 0.01 s is a good option.

Studies of the effect of the segment size on the balancing efficiency (Fig. 8) showed that the use of segments of small size reduces efficiency, since the number of "small" balancing operations increases, which results in the growth of overheads spent on redistribution of tasks between parallel agents. The use of segments of large size is also inefficient, since the segment size in this case becomes comparable with the fragment size, which makes balancing impossible. Thus, we experimentally established that the most appropriate segment size is 20000 tuples, which is approximately 0.01 of the average fragment size.

4.4. Study of the Effect of Load Balancing on the Query Execution Time

In the second series of experiments, we studied effect of load balancing on the query execution time. The experiments were carried out with four values of skew with respect to the partitioning attribute on 64 processor nodes. In all tests of this series, coefficient μ was set equal to 50%.

Results of these tests are presented in Fig. 9. They show that the optimal choice is $\rho = 0.8$, which made it possible to almost completely eliminate negative effect of skews with respect to the partitioning attribute. Complete replication does not yield additional advantages, since benefits from load balancing in the case of complete replication are outweighed by the increase of overheads associated with balancing and searching data segment in the replica. On the other hand, the efficiency of the load balancing method increases greatly when the skew coefficient θ grows. For skew "25–15" ($\theta = 0.2$), load balancing reduces query execution time by 30%. For skew "80–20" ($\theta = 0.68$), the use of load balancing reduces query execution time by 60%. The results of our experiments showed that the load balancing method can successfully be used in par-

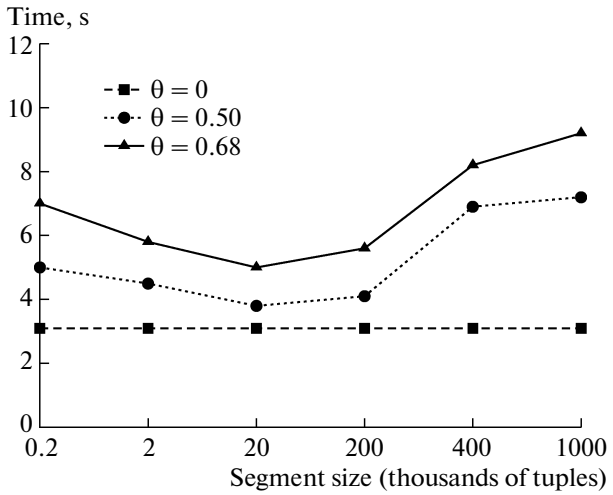

Fig. 8. Dependence of query execution time on the segment size ($n = 64$, $\mu = 50\%$).

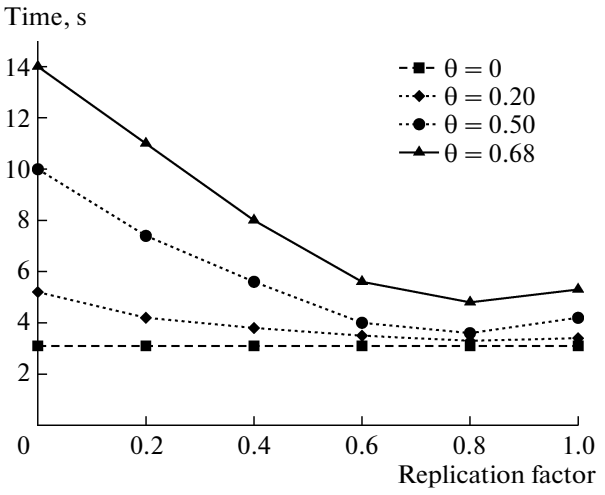

Fig. 9. Effect of replication factor ρ on query execution time ($n = 64$, $\mu = 50\%$).

allel database systems for eliminating load disbalance in the processing of "bad" queries.

4.5. Study of Scalability of the Load Balancing Algorithm

The last series of experiments was devoted to studying scalability of the proposed load balancing method. Results of these experiments are presented in Figs. 10, 11, and 12.

Scalability can be defined as a measure of efficiency of algorithm parallelization on multiprocessor configurations with different numbers of processor nodes. One of the most important qualitative characteristics of parallelization efficiency is *speedup*. In this work, under the speedup, we mean the following. Let two configurations A and B of a parallel database system with a given architecture be given. Let they differ in the number of the processors and devices associated with them. Let Q be some test. Speedup a_{AB} obtained upon transfer from configuration A to configuration B is defined by the following formula

$$a_{AB} = \frac{t_{QB}}{t_{QA}},$$

where t_{QA} is time spent by configuration A on the execution of test Q and t_{QB} is time spent by configuration B on the execution of test Q.

In the experiments presented in Fig. 10, the effect of the replication factor on the speedup was studied. In these experiments, we used skew $\mu = 50\%$, $\theta = 0.5$. The results demonstrate considerable increase of the speedup of query processing when the load balancing is applied. Increase of the replication factor results in "raising" the speedup plot and makes it "more linear." It should be noted that, in the case where the replication factor ρ is equal to 0.8, the speedup is approximately linear with coefficient 0.7.

In the experiments presented in Fig. 11, the effect of the skew with respect to the partitioning attribute on the speedup upon application of load balancing was studied. In spite of the fact that skews greatly reduce speedup, load balancing ensures significant speedup even for large values of the skew coefficient and makes it possible to avoid degradation of system performance even for skews of form "80–20" ($\theta = 0.68$).

In the experiments presented in Fig. 12, the effect of the skew coefficient with respect to the join attribute on the speedup was studied. Results of these experiments show that, even in the worst case (where 80% of

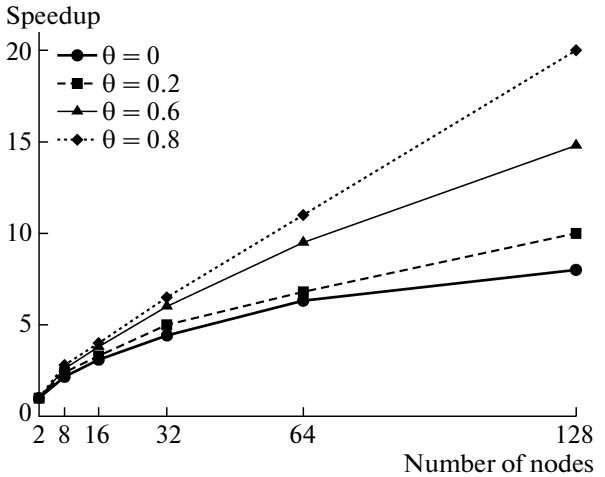

Fig. 10. Dependence of speedup on the replication factor ρ ($\theta = 0.5$, $\mu = 50\%$).

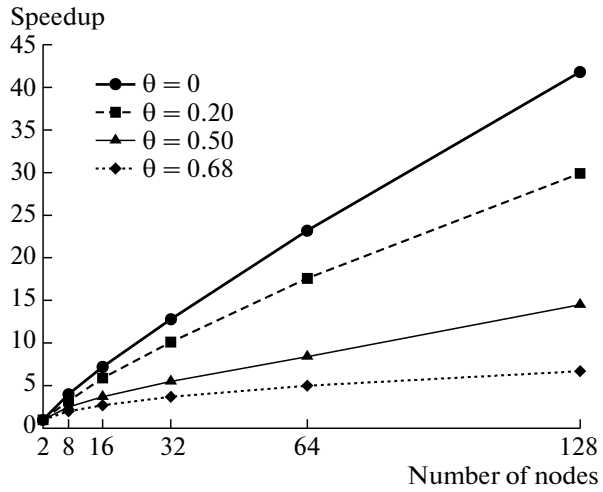

Fig. 11. Effect of skew with respect to data θ on speedup (ρ = 0.50, μ = 50%).

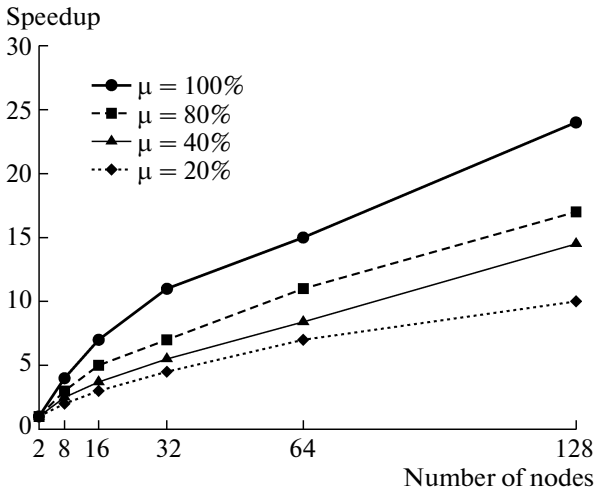

Fig. 12. Dependence of speedup on skew with respect the join attribute μ (θ = 0.5, ρ = 0.50).

tuples are "aliens"), load balancing ensures marked speedup.

6. CONCLUSION

In this paper, we have discussed parallel query execution in multiprocessor systems with cluster architecture. The load balancing problem and related data placement problem have been studied. An approach to placement fragments and replicas in cluster systems based on logical partitioning of a relation fragment into segments of equal length has been suggested. A load balancing algorithm based on the partial mirroring has been described. A strategy of selection of an outsider and a particular formula for calculating the rating function are suggested. The proposed method and algorithm are implemented in a prototype of the parallel DBMS "Omega." The debugged code of the system contained more than 10000 lines in the C language. On the basis of the prototype of the DBMS "Omega," large-scale computational experiments were carried out on the cluster "SKIF Ural." Results of the experiments substantiate efficiency of the proposed methods and algorithms.

In terms of further studies, of interest are the following problems:

(1) Encapsulation of parallelism and integration of the suggested load balancing algorithm in a parallel DBMS with open code PostgreSQL.

(2) Implementation and study of the above-described load balancing method in a DBMS for multiprocessor hierarchies including computing clusters on many-core processors connected in a grid environment.

ACKNOWLEDGMENTS

This work was supported by the Russian Foundation for Basic Research, project no. 09-07-00241-a.

REFERENCES

1. Dean, J., and Ghemawat, S., MapReduce: Simplified Data Processing on Large Clusters, *Commun. ACM*, 2008, vol. 51, no. 1, pp. 107–113.
2. Chaudhuri, S. and Narasayya, V., Self-tuning Database Systems: A Decade of Progress, *Proc. of the 33rd Int. Conf. on Very Large Data Bases*, Vienna, 2007, pp. 3–14.
3. Xu, Y., Kostamaa, P., Zhou, X., and Chen, L., Handling Data Skew in Parallel Joins in Shared-nothing Systems, *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Vancouver: ACM, 2008, pp. 1043–1052.
4. Han, W., Ng, J., Markl, V., Kache, H., and Kandil, M., Progressive Optimization in a Shared-nothing Parallel Database, *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Beijing, 2007, pp. 809–820.
5. Zhou, J., Cieslewicz, J., Ross, K.A., and Shah, M., Improving Database Performance on Simultaneous Multithreading Processors, *Proc. of the 31st Int. Conf. on Very Large Data Bases*, Trondheim, Norway, 2005, pp. 49–60.
6. Garcia, P. and Korth, H.F., Pipelined Hash-join on Multithreaded Architectures, *Proc. of the 3rd Int. Workshop on Data Management on New Hardware (DaMoN'07)* (Beijing, China, 2007), New York: ACM, pp. 1–8.
7. Lakshmi, M.S. and Yu, P.S., Effect of Skew on Join Performance in Parallel Architectures, *Proc. of the first Int. Symp. on Databases in Parallel and Distributed Systems*, Austin, Texas: IEEE Comput. Society, 1988, pp. 107–120.
8. Ferhatosmanoglu, H., Tosun, A.S., Canahuate, G., and Ramachandran, A., Efficient Parallel Processing of

Range Queries through Replicated Declustering, *Distributed Parallel Databases*, 2006, vol. 20, no. 2, pp. 117–147.

9. Kostenetskii, P.S., Lepikhov, A.V., and Sokolinskii, L.B., Technologies of Parallel Database Systems for Hierarchical Multiprocessor Environments, *Avtom. Telemekh.*, 2007, no. 5, pp. 112–125 [*Automation Remote Control* (Engl. Transl.), 2007, vol. 68, no. 5, pp. 847–859.
10. Sokolinskii, L.B., Organization of Parallel Query Processing in Multiprocessor Database Machines with Hierarchical Architecture, *Programmirovanie*, 2001, no. 6, pp. 13-29. [*Programming Comput. Software* (Engl. Transl.), 2001, vol. 27, no. 6, pp. 297–308].
11. Lepikhov, A.V. and Sokolinsky, L.B., Data Placement Strategy in Hierarchical Symmetrical Multiprocessor Systems, *Proc. of Spring Young Researchers Colloquium in Databases and Information Systems (SYRCoDIS'2006)*, Moscow: Moscow State University, 2006, pp. 31–36.
12. Parallel DBMS "Omega" for Multiprocessor Hierarchies. URL: http://fireforge.net/projects/omega/.
13. Rating TOP50: A list of 50 Most Powerful Computers in CIS. URL: http://supercomputers.ru/.
14. Computational Cluster "SKIF Ural". URL: http://supercomputer.susu.ru/computers/ckif_ural/.

SPELL: 1. petabyte