ISSN 0361-7688, Programming and Computer Software, 2017, Vol. 43, No. 3, pp. 131–144. © Pleiades Publishing, Ltd., 2017.
Original Russian Text © E.V. Ivanova, L.B. Sokolinsky, 2017, published in Programmirovanie, 2017, Vol. 43, No. 3.

Parallel Processing of Very Large Databases Using Distributed Column Indexes

E.V. Ivanova* and L. B. Sokolinsky**

South Ural State University, Chelyabinsk, 454080 Russia

**e-mail: Elena.Ivanova@susu.ru*

***e-mail: Leonid.Sokolinsky@susu.ru*

Received April 20, 2016

Abstract—The development and investigation of efficient methods of parallel processing of very large databases using the columnar data representation designed for computer cluster is discussed. An approach that combines the advantages of relational and column-oriented DBMSs is proposed. A new type of distributed column indexes fragmented based on the domain-interval principle is introduced. The column indexes are auxiliary structures that are constantly stored in the distributed main memory of a computer cluster. To match the elements of a column index to the tuples of the original relation, surrogate keys are used. Resource hungry relational operations are performed on the corresponding column indexes rather than on the original relations of the database. As a result, a precomputation table is obtained. Using this table, the DBMS reconstructs the resulting relation. For basic relational operations on column indexes, methods for their parallel decomposition that do not require massive data exchanges between the processor nodes are proposed. This approach improves the class OLAP query performance by hundreds of times.

DOI: 10.1134/S0361768817030069

1. INTRODUCTION

Presently, very large data bases (VLBs) significantly affect the field of data processing technologies. By the predictions of the analytical company IDC, the amount of data doubles every two years and will reach 44 zettabytes by 2020 [1]. After cleansing and structuring, very large data are transformed into VLDBs and data warehouses. By Microsoft's estimates, the amount of data stored by 62% of large American companies exceeds 100 terabytes [2]. However, modern database technologies cannot process such large amounts of data. By the IDC estimates, in 2013 only 22% of the total amount of data were potentially useful and only 5% were analyzed. By 2020, the percentage of potentially useful data can reach 35% , mainly due to the use of embedded systems [1].

In the opinion of a leading expert in databases M. Stonebraker, very large amounts of data can be managed using DBMS technologies [3]. DBMSs provide many useful services not available in file systems, including schema (to manage the data semantics), query language (to organize the access to parts of the database), complex access control systems (data granulation), services for ensuring the data integrity (data integrity control and transaction mechanism), data compression (to reduce the database size), and indexing (to speed up query execution).

To process a large amount of data, high-performance computer systems are needed [4, 5]. Presently, this class of computer systems is dominated by systems with the cluster architecture, where the computer nodes are equipped with multicore accelerators. Computer clusters occupy 85% of the TOP500 list of the most powerful supercomputers in the world [6] (as of November 2015). More than 50% of computers in the first hundred in this list use multicore accelerators. The fastest supercomputer Tianhe-2 (The National Supercomputer Center in Guangzhou, China) also has a cluster architecture based on the Intel Xeon Phi processors. Its total performance is 33.9 PFLOP/S on the test LINPACK, and the total amount of memory is 1 petabyte. Recent research show that computer cluster can be effectively used to store and process very large databases [7–10]. By the present time, there are major advances in the development of parallel DBMSs [11, 12]. A review of the literature on this topic can be found in [5, 13]. However, a number of open problems remain in this field that are mainly related to managing large data [3].

The random access memory as the main storage of data becomes more and more attractive as the cost/size ratio decreases exponentially [14–16]. According to the Gartner report, the in-memory DBMSs will by widely used in 2–5 years [17].

An important class of applications related to processing very large databases is the set of applications

dealing with data warehouses [18–21], for which OLAP-type queries are typical. Investigations [22–24] showed that it is reasonable to use the column-oriented model of data representation, which provides the performance by an order of magnitude higher than the conventional DBMSs using the row-oriented representation of data. The difference in performance is explained by the fact that the column-oriented databases make a less number of data exchanges with disks when executing data selection queries because only the values of the attributes involved in the query are read from the disk (or from the main memory) [25]. An additional advantage of the column-oriented representation is the possibility to use efficient data compression algorithms because each column contains data of the same type. Compression can improve the performance by an order of magnitude because the I/O operations take less time [26]. A disadvantage of the column-oriented model of data representation is that it is more difficult to use the efficient SQL query optimization techniques, which proved to be useful in relational DBMSs. In addition, the column-oriented DBMSs are considerably inferior to the row DBMSs in terms of performance on the class of OLTP queries.

According to the aforesaid, it is important to develop new efficient methods for the parallel processing of very large databases stored in the main memory on computer clusters based on multicore accelerators that would be able to combine the advantages of the relational model with the column-oriented representation of data.

The paper is organized as follows. In Section 2, we review the studies on the column data storage model and analyze the publications that are closely related to the topic of the present paper. Section 3 is devoted to the formal description of the domain-column model of data representation. Based on this model, in Section 4 we propose a decomposition of the basic relational operations on distributed column indexes. In Section 5, we describe the architecture and implementation of a columnar coprocessor for relational DBMSs. In Section 6, we describe the computational experiments aimed at investigating the efficiency of the proposed models, methods, and algorithms applied to processing very large databases. In the Conclusions section, we summarize the main results obtained in the paper, and discuss the their distinctions from the results obtained by other authors, give recommendations on the practical application of column indexes, and discuss the directions of further research

2. THE COLUMN-ORIENTED MODEL OF DATA STORAGE

The column-oriented representation of data assumes that the database tables are stored by separate columns. The column data storage model has recently gained in popularity among the researchers in the field of on-line analytic data processing in data warehouses [27–30]. In distinction from the conventional row representation, the column-oriented representation is much more efficient in the execution of OLAP queries [25]. This is explained by the fact that, as a column-oriented DBMS executes a query, it reads from the disk only the attributes that are needed for the query execution, which reduces the amount of I/O operations and, consequently, decreases the query execution time. A disadvantage of the column-oriented representation is the low efficiency in executing row-oriented operations, such as insertion or deletion of tuples. As a result, the column-oriented DBMSs can be inferior to the row-oriented DBMSs in the execution of OLTP queries. Figure 1 illustrates the basic differences in the physical layout of column-stores compared to traditional row-oriented databases [29]. Here, three methods for the representation of the *Sales* relation containing five attributes are illustrated. In the column-oriented approach (Figs. 1a and 1b), each column is stored independently as a separate data object. Since data is typically read from the storage and written to the storage in blocks, the column-oriented approach means that each block that holds data for the sales table holds data for one column only. In this case, the query that calculates, e.g., the number of sales of a certain product in July must access only the columns prodid (the product identifier) and date (the sale date). Therefore, it is sufficient for the DBMS to load into the main memory only the blocks that contain the data for these columns. On the other hand, in the row-oriented approach (Fig. 1c), there is just a single data object containing all the data, i.e., each block in the storage that holds data for the *Sales* table contains data from all columns of the table. In this way, there is no way to read just the particular attributes needed for a particular query without also transferring the surrounding attributes. Taking into account the fact that the cost of data exchanges with the disk (or between the processor cash and the main memory) is a bottleneck of DBMSs and the database schemata become more and more complex and include wide tables containing hundreds of attributes, the column-oriented databases can significantly improve the performance in the execution of OLAP queries.

The simplest organization of a column-store is the addition a column of surrogate keys to each field [31]; the surrogate key is the identifier of the row in the corresponding relation. In this case, we obtain the so-called two-column representation with a surrogate key (Fig. 1b). In [32], a virtual key mechanism was proposed. In this case, the role of the surrogate key is played by the ordinal number of the value in the column Taking into account that all the data in each column have the same type (occupy the same number of bytes) and are stored in a continuous memory area, given the ordinal number of the value, one can determine the place in the memory occupied by this value. The ith tuple of the relation is reconstructed by select-

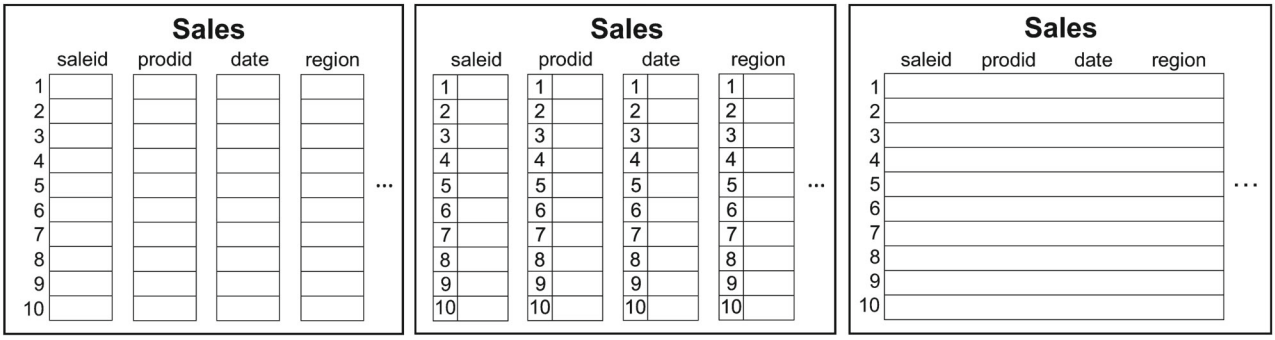

Fig. 1. Physical organization of column-store and row-store. (a) Column-store with virtual keys, (b) Column-store with urrogate keys, (c) Row-srore

ing and joining the ith values of the columns belonging to the relation. In this case, we obtain a single-column representation (Fig. 1a), which saves memory.

The efficiency of data compression in column-oriented databases was studied in [26, 33]. Since all the values in every column are of the same type, the most efficient compression method for each data type can be used. The greatest effect of compression is achieved in the case when the data is sorted and the column contains large portions of repeated values. To increase the speed of query execution, the lightweight data compression [33] is most appropriate; in this case, the processor load for the compression and decompression of data does not outweigh the benefit due to the reduced data transfer time [34, 35]. In [26, 36, 37], it was shown that the execution of operations on compressed data can increase the query execution performance by an order of magnitude.

A major advantage of row-stores is the availability of efficient query optimization techniques in row-oriented DBMSs developed based on the relational model. Another advantage of the row-oriented DBMSs is the speed of processing of OLTP queries. For this reason, a considerable effort in the database research community was aimed at the integration of the advantages of the column data storage model into the row-oriented DBMSs [25].

In [38], a new scheme for data mirroring was proposed, which is called the fractured mirror. This approach uses a hybrid scheme for data storage that includes both the row-oriented and column-oriented representation of data. The data modification operations are performed based on the row-oriented representation, and the data read and analysis operations are performed based on the column-oriented representation. The changes made in the row-oriented representation are copied to the column-oriented representation in a background process. The tables in the row-oriented representation are divided into fragments that are stored on different disks. Each row fragment is assigned its mirror copy in the column-oriented representation on the same disk using virtual keys. This scheme is well suited for the parallel execution of queries that do not require data exchanges between nodes. However, the execution of complex queries that require massive data exchanges between processor nodes involves large overhead due to message transmission. This approach assumes that a hybrid execution plan is created for each query that actually consists of a combination of two plans—one for the row-oriented representation and the other for the column-oriented representation. The optimization of such hybrid queries creates a large number of difficult problems.

In [31, 39], the Decomposition Storage Model (DSM), which uses the declusterization of each relation into columns using surrogate keys (see Fig. 1b), was proposed. In [25], an attempt to emulate the DSM using the relational DBMS System X was made. Each relation was decomposed into columns according to the number of attributes in the given relation. Each column was represented by a binary (two-column) table. The first column contained the surrogate key identifying the corresponding row in the original table, and the second column contained the attribute value. For the query execution, only the columns corresponding to the attributes involved in the query were loaded from the disk. They were joined by the surrogate key into "truncated" tables on which the query was then executed. By default, System X uses hash joins for this purpose, which turned out to be very costly. To overcome this drawback, an attempt to calculate the joins using clustered cluster indexes created for each column was made; however, due to the index access, the overheads turned out to be even higher.

The DSM has two disadvantages. First, each table must contain a column for the surrogate keys, which requires additional disk memory and data exchanges with the disk. Second, the majority of row-oriented DBMSs store a fairly large header along with each tuple, which also requires additional disk memory (in column-oriented databases, the headers are stored in special columns to avoid this overhead). To overcome these disadvantages, the paper [25] proposed and

investigated an approach in which the relations are stored in the conventional row form, and, for each column of every table, an index in the form of a B-tree is defined. For the execution of SQL queries, index-only execution plans were generated. For the execution of such plans, the actual tuples on the disk are never accessed. Even though the indexes explicitly store the identifiers of the stored records, each value of the corresponding column is stored only once, and the access to the column values typically requires less overhead because the indexes do not store the tuple headers. Experiments showed that this approach demonstrates a much lower performance than the column-oriented DBMs.

Another approach considered in [25] uses materialized views. In this approach, for each class of queries, the optimal set of materialized views is created that contain only the columns needed for the queries of this class. In these views, no joins of columns from different tables are made. The purpose of this strategy is to enable the DBMS to access only the data on the disk that are actually needed. This approach performs better than the approaches based on the DSM and index-only plans. However, its application requires the preliminary knowledge of the workload, which considerably restricts its practical application.

In [40], one more attempt to combine the advantages of the row-stores and column-stores in the framework of a relational DBMS was made. It was proposed to store data in a compressed form using a special table (called the *c*-table) for each column in the relational schema (similarly to the method of vertical decomposition of relations). The data were compressed using the Run-Length Encoding (RLE) [26]. On OLTP queries, this method showed the performance comparable with that of the column-oriented DBMSs. However, this approach requires the creation of a *c*-table for each attribute and the creation of a large number of indexes for each *c*-table, which requires a lot of disk memory. In addition, the dependences between tuples in the *c*-tables result in an unreasonable cost of insert operations even for application with relatively rare updates.

In [41], an alternative approach to using column query execution methods in row-oriented DBMSs is described. This method is based on the query execution plans that use only indexes and special operators *Index Merge, Index Merge Join,* and *Index Hash Join,* integrated into the DBMS kernel. The proposed methods are designed for use with solid state drives and multicore processors. The performance of a row-oriented DBMS modified in this way on OLAP queries can be comparable with or even higher than the performance of column-oriented DBMSs.

In [42], new auxiliary data structures—column-store indexes—used in Microsoft SQL Server 11 are described. The column store indexes form a purely column store because the data of different columns are stored on different disk pages, which considerably improves the performance of I/O operations. To improve the performance of the OLTP query execution, the user should create columnstore indexes for the corresponding tables. The decision about using the column store indexes is made by the DBMS, as is the case for conventional B-tree indexes. The advantages of the new indexes are illustrated using the TPC DS benchmark [43]. For the table *catalog_sales,* a columnstore index containing all 34 columns of this table is created. Therefore, along with the row storage, Microsoft SQL Server 11 creates a column storage in which the data of certain columns of certain tables is completely duplicated. The columnstore index is constructed as follows. The original table is divided into sequential groups of rows of the same length. The columns of attribute values in each group are encoded and compressed independently of each other. As a result, compressed column segments are obtained of which each is stored as a BLOB (Binary Large Object) [44]. In the encoding phase, the data are transformed into integers. Two strategies are supported: (1) encoding of values and (2) encoding by enumeration (as in enumerated types in universal programming languages). Next, the column is compressed using the RLE. To achieve the maximum compression ratio, the original groups of rows are sorted using the Vertipaq algorithm. Queries on the column-store in Microsoft SQL Server 11 are executed using special operators that support block data processing [45] (the query executor for the row-store uses the conventional tuple processing). Note that the block operators are not applicable to the data stored by rows. Thus, the Microsoft SQL Server 11 actually implements two independent query executors—one for the row-store and the other for the column-store. By analyzing the query, the DBMS decides which of the two query executors should be used. The DBMS also provides mechanisms for the synchronization of data in both stores.

The analysis of available solutions [25] shows that no advantage can be obtained by storing data by columns using a row-oriented DBMS with a vertically divided schema or by indexing all columns to ensure the independent access to them. The database systems with row storage have a considerably lower performance than the column-oriented databases on the Star Schema Benchmark (SSBM) [46–48]. The difference in performance shows that there are significant differences at the level of query execution between the two systems (in addition to the evident differences at the level of storage).

3. DOMAIN-COLUMN MODEL

In this section, we describe a new domain-column model of the data representation and define the concept of the column index.

We will use the notation for relational operations adopted in the well-known book [49]. By $\pi_{*\setminus A}(R)$, we mean the projection of the relation R onto all the attributes, except for the attribute A. The symbol $\circ$ denotes the concatenation of two tuples:

$$(x_1,\dots,x_u)\circ(y_1,\dots,y_v)=(x_1,\dots,x_u,y_1,\dots,y_v).$$

By $R(A,B_1,\dots,B_u)$, we denote the relation R with the *surrogate key* A (an integer identifier that uniquely determines the tuple) and the attributes $B_1,\dots,B_u$; this relation is the set of tuples of length $u+1$ of the form $(a,b_1,\dots,b_u)$, where $a\in\mathbb{Z}_{\geq 0}$ and $\forall j\in\{1,\dots,u\}$ $(b_j\in\mathfrak{D}_{B_j})$. Here $\mathfrak{D}_{B_j}$ is the domain of the attribute B_j. By $r.B_j$, we denote the value of the attribute B_j, and $r.A$ is the value of the surrogate key of the tuple r: $r=(r.A,r.B_1,\dots,r.B_u)$. The *surrogate key* of the relation R has the property $\forall r',r''\in R(r'\neq r''\Leftrightarrow r'.A\neq r''.A)$. The *address of the tuple r* is the value of this tuple's *surrogate key*. To obtain a tuple of the relation R given its address, we will use the *dereferencing function* $\&_R$: $\forall r\in R\big(\&_R(r.A)=r\big)$.

In what follows, we consider relations as sets rather than as multisets [49]. This means that, when an operation produces a relation with duplicate elements, the operation of removing duplicates is applied by default.

Let a relation $R(A,B,\dots)$ $(T(R)=n)$ be given, and let a total order be defined on the set $\mathfrak{D}_B$. A *column index* $I_{R.B}$ of the attribute B of the relation R is defined as an ordered relation $I_{R.B}(A,B)$ satisfying the following conditions:

$$T(I_{R.B})=n,\quad \pi_A(I_{R.B})=\pi_A(R); \tag{1}$$

$$\forall x_1,x_2\in I_{R.B}(x_1\leq x_2\Leftrightarrow x_1.B\leq x_2.B); \tag{2}$$

$$\forall r\in R(\forall x\in I_{R.B}(r.A=x.A\Rightarrow r.B=x.B)). \tag{3}$$

Condition (1) implies that the sets of values of the surrogate keys (addresses) of the index and of the relation being indexed are identical. Condition (2) implies that the elements of the index are arranged in increasing of the attribute B. Condition (3) implies that the attribute A of an index element contains the address of the tuple of the relation R that has the same value of the attribute B as this element of the column index.

Informally, the column index $I_{R.B}$ is a table consisting of two columns A and B. The number of rows in the column index coincides with the number of rows in the table being indexed. The column B of $I_{R.B}$ includes all values of the column B of the table R (with the repeated values taken into account) arranged in ascending order. Each row x of the index $I_{R.B}$ contains in its column A the surrogate key (address) of the row r of the table R that has the same value in the column B as x. Figure 1 shows examples of two different column indexes of the same relation.

Let a total order be defined on the set of values of the domain $\mathfrak{D}_B$. We divide the set $\mathfrak{D}_B$ into $k>0$ nonoverlapping intervals

$$\left.\begin{aligned}&V_0=[v_0;v_1],\quad V_1=(v_1;v_2],\dots,\\&V_{k-1}=(v_{k-1};v_k];\\&v_0<v_1<\dots<v_k;\\&\mathfrak{D}_B=\bigcup_{i=0}^{k-1}V_i.\end{aligned}\right\} \tag{4}$$

Note that, in the case $\mathfrak{D}_B=\mathbb{R}$, we have $v_0=-\infty$ and $v_0=+\infty$. the function $\varphi_{\mathfrak{D}_B}:\mathfrak{D}_B\to\{0,\dots,k-1\}$ is called the *domain fragmentation function* for $\mathfrak{D}_B$ if it satisfies the condition

$$\forall i\in\{0,\dots,k-1\}(\forall b\in\mathfrak{D}_B(\varphi_{\mathfrak{D}_B}(b)=i\Leftrightarrow b\in V_i)). \tag{5}$$

In other words, the domain fragmentation function assigns to the value b the number of the interval where this value belongs.

Let a column index $I_{R.B}$ for the relation $R(A,B,\dots)$ with the attribute B over the domain $\mathfrak{D}_B$ and the domain fragmentation function $\varphi_{\mathfrak{D}_B}$ be given. The function

$$\varphi_{I_{R.B}}:I_{R.B}\to\{0,\dots,k-1\} \tag{6}$$

defined by the rule

$$\forall x\in I_{R.B}\big(\varphi_{I_{R.B}}(x)=\varphi_{\mathfrak{D}_B}(x.B)\big) \tag{7}$$

is called the *domain-interval fragmentation function* for the index $I_{R.B}$. In other words, the fragmentation function $\varphi_{I_{R.B}}:I_{R.B}$ assigns to each tuple x in $I_{R.B}$ the number of the domain interval where the value $x.B$ belongs.

Define the ith fragment $(i=0,\dots,k-1)$ of the index $I_{R.B}$ as

$$I^i_{R.B}=\{x\,|\,x\in I_{R.B};\ \varphi_{I_{R.B}}(x)=i\}. \tag{8}$$

This means that the ith fragment contains the tuples whose value of the attribute B belongs to the ith domain interval. Such a fragmentation is said to be the *domain-interval fragmentation*. The number of fragments k is called the *fragmentation degree*.

The domain-interval fragmentation has the following fundamental properties, which are immediate consequences of its definition:

$$I_{R.B}=\bigcup_{i=0}^{k-1}I^i_{R.B}; \tag{9}$$

$$\forall i,j\in\{0,\dots,k-1\}(i\neq j\Rightarrow I^i_{R.B}\cap I^j_{R.B}=\varnothing). \tag{10}$$

Figure 3 schematically illustrates the fragmentation of the column index of degree $k=3$.

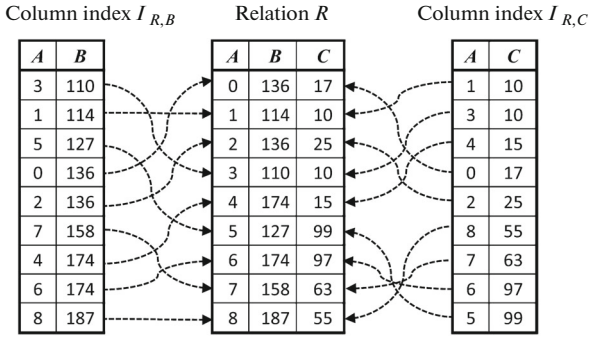

Fig. 2. A column index.

Let $I_{R.B}$ and $I_{R.C}$ be column indexes for the relation $R(A, B, C, ...)$. The fragmentation defined by the function $\ddot{\varphi}_{I_{R.C}} : I_{R.C} \to \{0, ..., k-1\}$ satisfying the condition $\forall x \in I_{R.C}$

$$\ddot{\varphi}_{I_{R.C}}(x) = \varphi_{I_{R.B}}(\sigma_{A=x.A}(I_{R.B})) \tag{11}$$

is called *a transitive fragmentation* of the index $I_{R.C}$ relative to the index $I_{R.B}$.

The transitive fragmentation makes it possible to place the elements of the column indexes corresponding to a tuple of the relation being indexed on the same node.

4. DECOMPOSITION OF RELATIONAL OPERATIONS

In this section, we describe the decomposition of the basic relational operations for distributed column indexes. This decomposition aims at partitioning the algorithm of the operation execution into subtasks that do not require the exchange of data.

Decomposition of the natural join. Consider the decomposition of the natural join operation $\pi_{*\setminus A}(R) \bowtie \pi_{*\setminus A}(S)$. Let $R(A, B_1, ..., B_u, C_1, ..., C_v)$ and $S(A, B_1, ..., B_u, D_1, ..., D_w)$ be two relations, and let there be two sets of column indexes on the attributes $B_1, ..., B_u$: $I_{R.B_1}, ..., I_{R.B_u}$, $I_{S.B_1}, ..., I_{S.B_u}$. Suppose that the domain-interval fragmentation

$$I_{R.B_j} = \bigcup_{i=0}^{k-1} I^i_{R.B_j}; \quad I_{S.B_j} = \bigcup_{i=0}^{k-1} I^i_{S.B_j}$$

of degree k be specified for all these indexes. Define

$$P^i_j = \pi_{I^i_{R.B_j}.A \to A_R,\ I^i_{S.B_j}.A \to A_S}\left(I^i_{R.B_j} \underset{I^i_{R.B_j}.B_j = I^i_{S.B_j}.B_j}{\bowtie} I^i_{S.B_j} \right) \tag{12}$$

for all $i = 0, ..., k-1$. Define

$$P = \bigcup_{i=0}^{k-1} P^i. \tag{13}$$

Let us construct the relation

$$Q(B_1, ..., B_u, C_1, ..., C_v, D_1, ..., D_w)$$

as follows:

$$\begin{aligned} Q = \{(&\&_R(p.A_R).B_1, ..., \&_R(p.A_R).B_u, \\ &\&_B(p.A_B).C_1, ..., \&_B(p.A_B).C_v, \\ &\&_S(p.A_S).D_1, ..., \&_S(p.A_S).D_w) \mid p \in P\}. \end{aligned} \tag{14}$$

Then, $Q = \pi_{*\setminus A}(R) \bowtie \pi_{*\setminus A}(S)$. A proof of this fact can be found in [50]. An example of the calculation of the natural join of two relations using distributed column indexes is discussed in [51].

Decomposition of the grouping operation. Consider the decomposition of the grouping operation $\gamma_{B,C_1,...,C_u,\mathrm{agrf}(D_1,...,D_w)\to F}(R)$. Let the relation

$$R(A, B, C_1, ..., C_u, D_1, ..., D_w, ...)$$

with the surrogate key A be given. Let the aggregation function **agrf** for the attributes $D_1, ..., D_w$ be specified, and let there be the column index $I_{R.B}$. Furthermore, suppose that we have the column indexes $I_{R.C_1}, ..., I_{R.C_u}$; $I_{R.D_1}, ..., I_{R.D_w}$. Let the domain-interval fragmentation $I_{R.B} = \bigcup_{i=0}^{k-1} I^i_{R.B}$ of degree k be available for the index $I_{R.B}$. We also suppose that, for the indexes $I_{R.C_1}, ..., I_{R.C_u}$ and $I_{R.D_1}, ..., I_{R.D_w}$, there is the transitive (with respect to $I_{R.B}$) fragmentation

$$\forall j \in \{1, ..., u\} \left(I_{R.C_j} = \bigcup_{i=0}^{k-1} I^i_{R.C_j} \right);$$

$$\forall j \in \{1, ..., w\} \left(I_{R.D_j} = \bigcup_{i=0}^{k-1} I^i_{R.D_j} \right).$$

Define

$$\begin{aligned} P_i = \pi_{A,F}(&\gamma_{\min(A)\to A,B,C_1,...,C_u,\mathrm{agrf}(D_1,...,D_w)\to F}(I^i_{R.B} \\ &\bowtie I^i_{R.C_1} \bowtie ... \bowtie I^i_{R.C_u} \bowtie I^i_{R.D_1} \bowtie ... \bowtie I^i_{R.D_w})) \end{aligned} \tag{15}$$

for all $i = 0, ..., k-1$. Define $P = \bigcup_{i=0}^{k-1} P_i$, and construct the relation $Q(B, C_1, ..., C_u, F)$ as follows

$$\begin{aligned} Q = \{(&\&_R(p.A).B, \&_R(p.A).C_1, ..., \\ &\&_R(p.A).C_u, p.F) \mid p \in P\}. \end{aligned} \tag{16}$$

Then, $Q = \gamma_{B,C_1,...,C_u,\mathrm{agrf}(D_1,...,D_w)\to F}(R)$. A proof of the validity of this decomposition of the grouping operation can be found in [52].

Decomposition of the intersection operation. Consider the decomposition of the intersection operation $\pi_{B_1,...,B_u}(R) \cap \pi_{B_1,...,B_u}(S)$. Let $R(A, B_1, ..., B_u)$ and

$S(A, B_1, \ldots, B_u)$ be relations with an identical set of attributes. Let $I_{R.B_1}, \ldots, I_{R.B_u}$ and $I_{S.B_1}, \ldots, I_{S.B_u}$ be two sets of column indexes on the attributes $B_1, \ldots, B_u$. Suppose that a domain-interval fragmentation of degree k

$$I_{R.B_j} = \bigcup_{i=0}^{k-1} I^i_{R.B_j}; \quad I_{S.B_j} = \bigcup_{i=0}^{k-1} I^i_{S.B_j}$$

be given for these indexes. Define

$$P^i_j = \pi_{I^i_{R.B_j}.A \to A_R, I^i_{S.B_j}.A \to A_S} \left(I^i_{R.B_j} \underset{(I^i_{R.B_j}.B_j = I^i_{S.B_j}.B_j)}{\bowtie} I^i_{S.B_j} \right) \quad (17)$$

for all $i = 0, \ldots, k-1$ and $j = 1, \ldots, u$. Define $P_j = \bigcup_{i=0}^{k-1} P^i_j$, and set $P = \bigcap_{j=1}^{u} P_j$. Construct the relation $Q(A, B_1, \ldots, B_u)$ as follows:

$$Q = \{r \mid r \in R \wedge r.A \in \pi_{A_R}(P)\}. \quad (18)$$

Then, $\pi_{B_1,\ldots,B_u}(Q) = \pi_{B_1,\ldots,B_u}(R) \cap \pi_{B_1,\ldots,B_u}(S)$. A proof of the validity of this decomposition of the intersection operation can be found in [53].

Using the approach described above, we also decomposed the operations of projection, selection, removing duplicate rows, and union operations (see [54]).

5. THE COLUMNAR COPROCESSOR CCOP

Based on the domain-column model of data representation described in Section 3 and the decomposition of relational operations, we developed a software called Columnar COProcessor (CCOP) for computer clusters. In this section, we describe the architecture and implementation of the CCOP.

CCOP is a software designed for managing distributed column indexes placed in the main memory of the computer cluster. The purpose of the CCOP is to build the precomputation tables (PCTs) for resource hungry relational operations at the request of the DBMS. The schematic of the interaction between the DBMS and the CCOP is shown in Fig. 4. The CCOP includes the program *Coordinator* run on the node 0 of the cluster and the program *Executor* run on all other nodes assigned for the CCOP. A special *CCOP driver* is installed on the SQL server; this driver controls the interaction with the CCOP coordinator via the TCP/IP protocol. The CCOP works only with 32 and 64-byte integers. When column indexes for the attributes of other types are created, their values are encoded to integers or vector of integers. The CCOP supports the following basic operations, which can be used by the DBMS via the CCOP driver interface: CreateColumnIndex (creation of a distributed column index), Execute (building a PCT), Insert (insertion of a new tuple into the column index), TransitiveInsert (insertion of a new tuple into the column index by the transitive value), Delete (deletion of a tuple from the column index), and TransitiveDelete (deletion of a tuple from the column index by the transitive value).

To organize the interaction between the driver and the CCOP, a language CCOPQL (CCOP Query Language) based on the JSON data format was developed. Each column index is divided into fragments, which, in turn, are divided into segments. All segments of a fragment are stored in the main memory of one processor node in a compressed form. The fragments are compressed using the Zlib library [55, 56], which implements the DEFLATE compression method [57], which is a combination of the Huffman and the Lempel–Ziv compression methods.

We explain the operation of the CCOP using a simple example. Let a database contain two relations $R(A, B, D)$ and $S(A, B, C)$ stored in a SQL server (see Fig. 5). Suppose we want to execute the query

SELECT D, C
FROM R, S
WHERE R.B = S.B AND C < 13.

Let the CCOP have only two executor nodes, and let each node have three processor cores (the cores are labeled by $P_{11}, \ldots, P_{23}$ in Fig. 5). Let the attributes *R.B* and *S.B* be defined on the domains of integers in the interval [0; 120). The segment intervals for *R.B* and *S.B* are [0;20), [20;40), [40;60), [60;80), [80;100), and [100;120). As the fragment intervals for *R.B* and *S.B* we use [0; 59] and [60, 119]. Let the attribute *S.* be defined on the integer domain [0;25]. Initially, the database administrator creates the distributed column indexes $I_{R.B}$ and $I_{S.B}$ for the attributes *R.B* and *S.B* using the CCOP driver. Next, the distributed column index $I^B_{S.C}$ is created for the attribute *S.*, which is fragmented and segmented transitively with respect to the index $I_{S.B}$. The distributed column indexes $I_{R.B}$, $I_{S.B}$, and $I^B_{S.C}$ are stored in the main memory of the executor nodes. Thus, we obtain the distribution of data inside the CCOP illustrated in Fig. 5. Upon receiving the SQL query, it is transformed by the CCOP driver into the plan defined by the following relational expression:

$$\pi_{I_{R.B}.A \to A_R, I_{S.B}.A \to A_S} \left(I_{R.B} \underset{I_{R.B}.B = I_{S.B}.B}{\bowtie} (I_{S.B} \rtimes \sigma_{C<13}(I^B_{S.C})) \right).$$

When the driver performs the Execute operation, this query is passed to the CCOP coordinator in the form of the CCOPQL operator in JSON. The query is executed independently by the cores of the executor nodes over the corresponding groups of segments. Due to the domain fragmentation and segmentation, no data exchanges between the executor nodes and between the cores of one node are needed. Each core computes

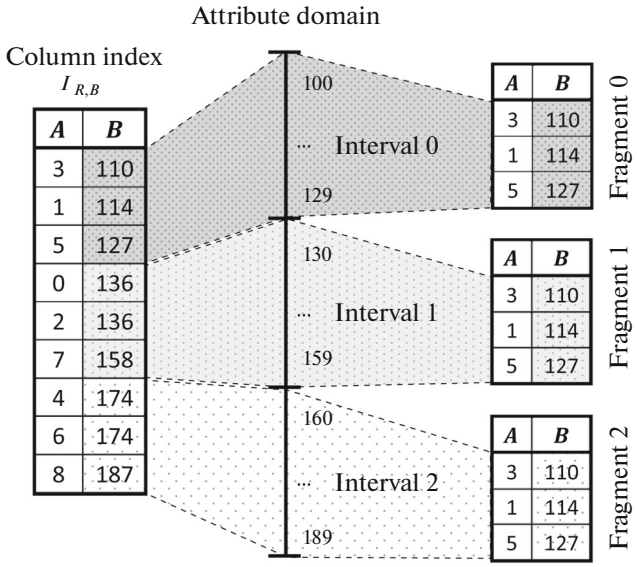

Fig. 3. Fragmentation of a column index.

its part of the PCT, which is sent to the coordinator node. the coordinator joins the PCT fragments into a unified table and sends it to the driver, which materializes this table as a relation in the database on the SQL server. Then, instead of the original query, the SQL server executes the query

```
SELECT D, C
FROM
  R INNER JOIN (
    ТПВ INNER JOIN S ON (S.A = ТПВ.AS)
  ) ON (R.A = ТПВ.AR).
```

The execution uses the conventional clustered B-tree indexes that were preliminary built for the attributes R.A and S.A.

The CCOP was implemented in C using hardware independent parallel programming technologies MPI and OpenMP. The size of the source code, which is available on the Internet at https://github.com/elena-ivanova/colomnindices/, is about 2500 lines.

6. COMPUTATIONAL EXPERIMENTS

In this section, we discuss the results of the computational experiments aimed at the investigation of efficiency of the proposed models, methods, and algorithms for processing very large databases using column indexes.

The experiments were performed on two computer clusters Tornado in South Ural State University and RSC PetaStream in the Joint Supercomputer Center (JSCC) of the Russian Academy of Sciences. Tornado [58] includes 384 processor nodes connected by the InfiniBand QDR and Gigabit Ethernet networks. Each processor node includes two six-core Intel Xeon X5680 CPUs, 24 GB of memory, and an Intel Xeon

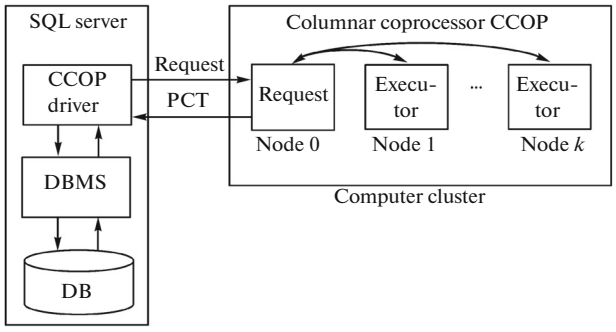

Fig. 4. Interaction between the SQL server and the CCOP.

Phi SE10X coprocessor (61 cores clocked at 1.1 GHz) connected by the PCI Express bus. RSC PetaStream [59] consists of eight modules consisting of eight Intel Xeon Phi 7120 coprocessors of which each has 61 cores and 16 GB of GDDR5 memory. The modules are connected by the InfiniBand FDR and Gigabit Ethernet networks. The Linux CentOS 7.0 is loaded on each coprocessor (on one core).

The CCOP was tested using a synthetic database built based on the benchmark TPC-H [60].

The test database consists of two tables ORDERS and CUSTOMER. The structure of these tables is described in [61]. To simulate skewed data the following distributions of the values of the attribute ORDERS.ID_CUSTOMER (the foreign key determining the customer who made the order): uniform, 45–20, 65–20, and 80–20 [62]. To vary the relation size, we used the selectivity coefficient *Sel* that takes values in the interval]0;1]. The coefficient *Sel* determines the size (in tuples) of the resulting relation relative to the size of the ORDERS relation. The database was scaled using the scale factor *SF* the value of which varied from 1 to 10. In our experiments, the size of the ORDERS relation was $SF \times 63000000$ tuples, and the size of the CUSTOMER relation was $SF \times 630000$ tuples.

In the first experiment, we investigated the balance of the Xeon Phi core workload under various skews in the distribution of the values of the foreign key ORDERS.ID_CUSTOMER. The results are illustrated in Fig. 6. It is seen from the plots in this figure that when the number of segments is small, a heavy disbalance in data results in a significant disbalance in the workload of the cores. When the number of segments coincides with the number of cores (60), the execution time of the operation in the case of the distribution 80–20 exceeds the execution time of the same operation for the uniform distribution by a factor four and more. However, when the number of segments increases, the effect of data disbalance is smoothed. For the distribution 45–20, the optimal

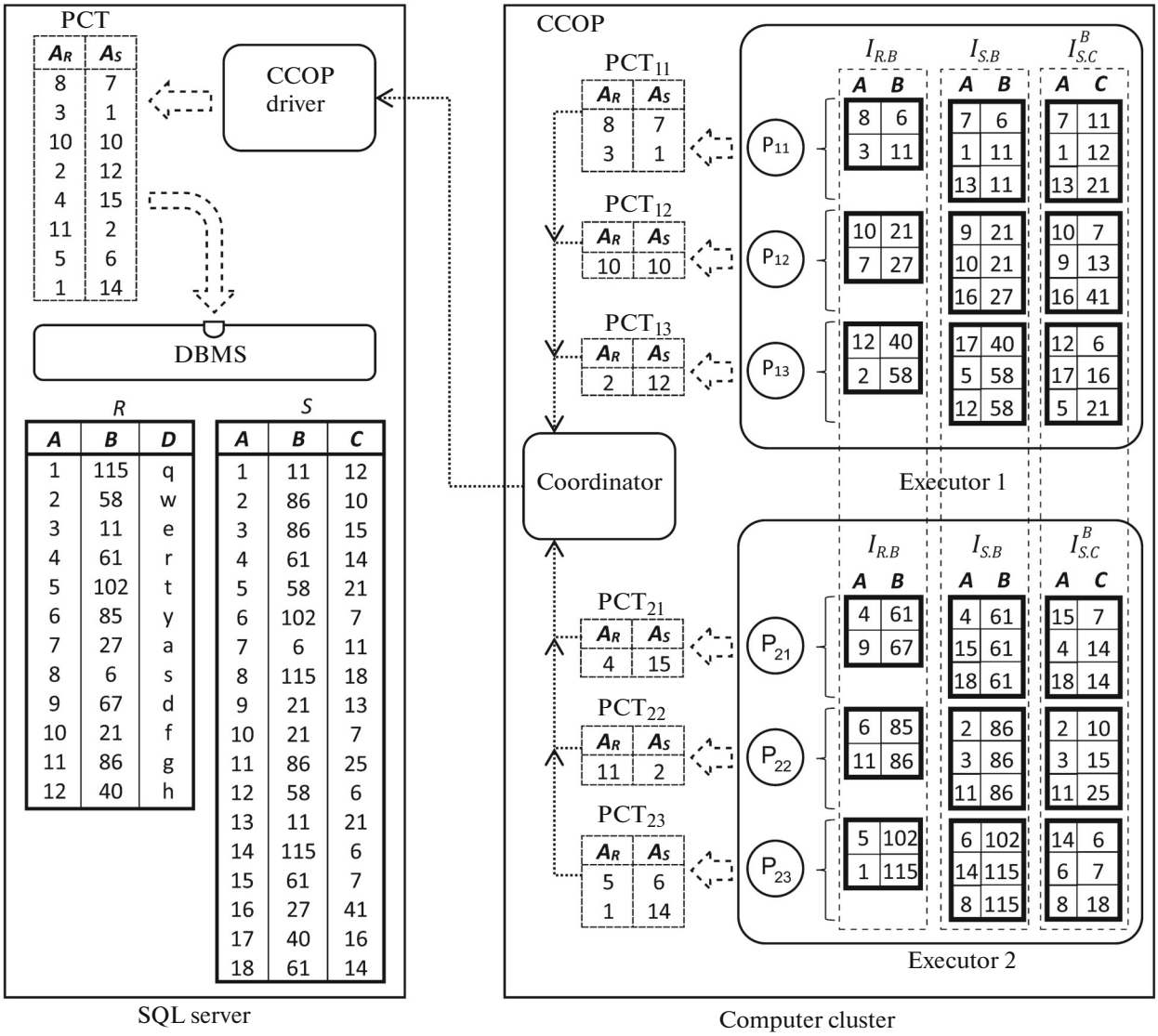

Fig. 5. Calculating the PCT using CCOP.

number of segments is 10000, for he distribution 65–20, it is 20000, and for the distribution 80–200000.

The purpose of the second experiment was to determine how efficient is the application of hyperthreading for the CCOP operation. The CPU Intel Xeon X5680 has hardware support for two threads per core, and the coprocessor Intel Xeon Phi supports four threads per core. The experimental results are illustrated in Fig. 7. It is seen that, in the case of the CPU, the performance increases up to the use of the maximum number of hardware supported threads. However, if only one thread per core is used, the speedup is almost perfect, while in the case of two threads per core, the speedup is lower. As applied to MIC, the picture is different. When only one thread per core is used, the speedup is almost perfect. In the case of two threads per core, the speedup is lower. The use of the greater number of threads per core results in the degradation of performance.

In the third and fourth experiments, we investigated the scalability of CCOP on massively parallel computers. Figure 8 shows the speedup plots for the case of computation of PCTs by the CCOP on the Tornado cluster; Fig. 9 shows similar plots for RSC PetaStream. The plots in Fig. 8 show that the selectivity *Sel* of the query is a factor that restricts the scalability. For example, for $Sel = 0.0005$, the speedup curve is almost linear, and for $Sel = 0.005$, it is close to linear. However, when the selectivity coefficient is large ($Sel = 0.05$), the scalability is restricted to 150 processor nodes. The cause is that at $Sel = 0.05$, the time needed to transfer, decompress, and merge the PCT varies

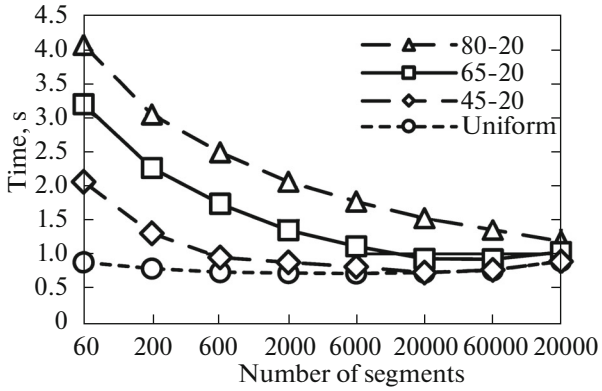

Fig. 6. Load balancing for the Xeon Phi coprocessor.

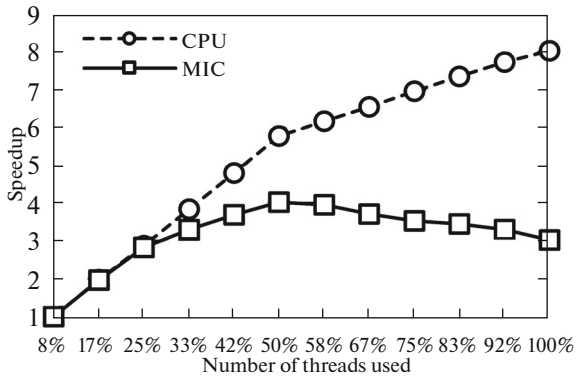

Fig. 7. Influence of hyper-threading on speedup.

insignificantly and considerably exceeds the time needed to compute the PCT; at $Sel = 0.0005$, the time needed to transfer, decompress, and merge the PCT is almost by an order of magnitude less than the time needed for its computation. Hence, we conclude that, when the query selectivity is small, CCOP demonstrates an almost linear speedup on large databases.

A similar picture was observed in testing the CCOP on the RSC PetaStream cluster (Fig. 9). In the next experiment, we investigated how much the use of the CCOP can speed up the execution of OLAP queries in a relational DBMS. In the experiment, we investigated three configurations:

1. PostgreSQL: the execution of a query without creating B-tree index files;

2. PostgreSQL & B-Trees: the execution of a query with the preliminary created B-tree index files for the join attributes;

3. PostgreSQL & CCOP: the execution of a query with the use of the PCT and the preliminary creation of B-tree index files for the surrogate keys.

In the last case, the query execution time was increased by the time needed for the CCOP to create the PCT. In each case, the time of the first and the repeated run of the query was measured. This is because PostgreSQL collects statistical data stored in the database dictionary and uses it to optimize the query execution plan. The experiments showed (see table) that, in the absence of B-tree indexes, the use of the columnar coprocessor speeds up the execution of queries by a factor of 100–150 for the selectivity coefficient $Sel = 0.0005$. However, if the selectivity coefficient is large, the efficiency of using CCOP is reduced and can even become negative (the speedup can become less than unity).

7. CONCLUSIONS

The development and investigation of efficient methods for the parallel processing of very large databases using the column-oriented data representation for computer clusters with multicore coprocessors that can be integrated into relational DBMSs was considered. Column indexes with surrogate keys are intro-

SQL query execution time and the speedup compared with PostgreSQL at $SF = 1$.

Configuration	Time in minutes					
	$Sel = 0.0005$		$Sel = 0.005$		$Sel = 0.05$	
	1st run	2nd run	1st run	2nd run	1st run	2nd run
PostgreSQL	7.3	1.21	7.6	1.29	7.6	1.57
PostgreSQL & B-Tree	2.62	2.34	2.83	2.51	2.83	2.63
PostgreSQL & CCOP	0.073	0.008	0.65	0.05	2.03	1.72
Speedup						
$\frac{t_{\text{PostgreSQL}}}{t_{\text{PostgreSQL\&CCOP}}}$	100	151	12	27	4	0.9
$\frac{t_{\text{PostgreSQL\&B-Trees}}}{t_{\text{PostgreSQL\&CCOP}}}$	36	293	4	50	1.4	1.53

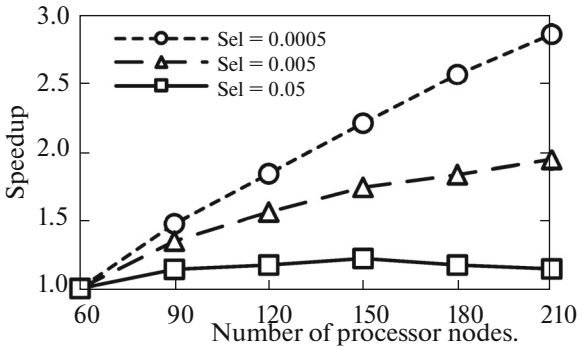

Fig. 8. Speedup of the PCT computation on the Tornado cluster at $SF = 10$.

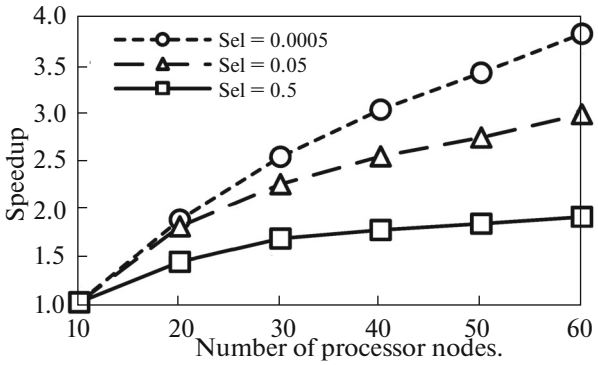

Fig. 9. Speedup of the PCT computation on the RSC PetaStream cluster at $SF = 1$.

duced. A domain–column model for the distribution of data over the nodes of the computer cluster is proposed. Based on this model, formal methods for the decomposition of relational operations into parts that can be executed independently (without data exchanges) on different processor nodes and cores were developed. The validity of all decomposition methods was rigorously proved. The proposed methods are implemented in the columnar coprocessor CCOP that can work on computer clusters including those that are equipped with multicore coprocessors; it can be used in combination with a relational DBMS for executing resource hungry operations. The DBMS interacts with the CCOP via a special driver running on the SQL server on which the DBMS runs. This interaction is organized via a special connector embedded into the DBMS code.

The main results discussed in this paper are new—they are not covered in the publication of other authors that were reviewed in Section 2.

In [42], the columnstore indexes used in Microsoft SQL Server 11 are described. Essentially, this approach assumes the creation of two independent query executors in the DBMS —one for the row-store and the other for the column-store. The approach described in this paper assumes a software implementation of the column indexes—the columnar coprocessor CCOP. In addition, [42] provides no details on the fragmentation of the columnstore indexes and on the methods used to parallelize operations on the columnstore indexes. We also note that the columnstore indexes in the Microsoft SQL Server 11 are represented in BLOB fields stored on disks, while in the CCOP they are distributed and stored in the main memory.

The methods for embedding column-oriented query processing into a row-oriented DBMS that were proposed in [41] require a significant modification of the DBMS; they do not suit for running on computer clusters with distributed memory. In distinction, the CCOP requires a minimum modification of the CCOP by adding a special connector to it, and it shows almost linear scalability on computer clusters with hundreds of processor nodes and tens of thousands of cores. In [40], additional data structures called *c*-tables were introduced that resemble the CCOP column indexes, but the issues concerning data distribution and parallel processing are not discussed.

The approach to the emulation of the column-store in a row-oriented DBMS based on materialized views [25] precludes the use of columns from different tables in one view, which restricts the application of this approach for the parallel execution of joins in multiprocessor systems with distributed memory. By contrast, the CCOP can execute joins without exchanging data on computer clusters with a large number of processor nodes.

The fractured mirror approach proposed in [38] makes it possible to organize the efficient parallel processing of fragment-independent queries in the column-like style on computer clusters. However, if the query depends of the method of fragmentation, the system's performance is degraded due to the large number of data transfers between the nodes. The domain-interval fragmentation used in the CCOP allows one to avoid data exchange. The other approaches to emulation described in [25] are not considered here because they have lower performance compared with the conventional relational DBMSs. In distinction from them, the CCOP speeds up the execution of queries on the data store by a factor of several hundred compared with the relational DBMS PostgreSQL.

The results obtained in this paper can be used for the creation of scalable columnar coprocessors for the existing commercial and free DBMSs. This makes it possible to process very large data stores on computer clusters including the computer systems whose nodes include multicore GPU or MIC coprocessors.

The amount of memory needed for storing the column indexes can be estimated on the standard TPC-H benchmark [60] used for the simulation of processing of analytical databases. The number of rows in all

tables of the TPC-H is 8661245 with the scaling coefficient $SF = 1$. Assume that three column indexes are created for each table. Our experiments show that the column indexes are compressed with the factor three on average. Taking into account the fact that each record of the column index consists of two 8-byte fields, we conclude that the amount of memory needed to storing the column indexes at $SF = 1$ is $8661245 \times 3 \times 16/3 \approx 0.13$ GB. At $SF = 30000$, which corresponds to an average size database, the required amount of memory is 4.2 TB. At the maximum $SF = 100000$, which corresponds to a large database, the amount of memory needed for storing the column indexes is 14 TB. The total amount of memory available in the cluster Tornado [58] (384 nodes with the memory of 24 GB and 96 nodes with the memory of 48 GB) is 13.8 TB. This cluster occupies place 349 in the 46th edition of the TOP500 list of the most powerful computer systems (as of November 2015); thus it is a medium power cluster. The estimates above show that the main memory of such a computer is sufficient for storing medium size analytical databases. For very large databases, more powerful computers are needed. Note that number one computer in the 46th edition of the TOP500 list (as of November 2015) has the memory of one petabyte, which is quite enough for storing column indexes of a very large analytical database.

The further lines of research can be as follows.

• Development and investigation of methods for the integration of the CCOP into free relational DBMSs like PostgreSQL.

• Integration of lightweight compression methods into the CCOP that do not require decompression for performing operations on data.

• Extension of the proposed approaches and methods to multidimensional data.

ACKNOWLEDGMENTS

This work was supported by the Act 211 Government of the Russian Federation (project no. 02.A03.21.0011) and by the Ministry of education and science of Russian Federation (government order 2.7905.2017).

REFERENCES

1. Turner, V., Gantz, J.F., Reinsel, D., et al., The Digital Universe of Opportunities: Rich Data and the creasing Value of the Internet of Things: IDC white paper, 2014. http://www.idcdocserv.com/1678.
2. Big Data Insights. Microsoft, 2013. https://blogs.msdn.microsoft.com/microsoftenterpriseinsight/2013/04/12/big-data-insights/.
3. Stonebraker, M., Madden, S., and Dubey, P., Intel "big data" science and technology center vision and execution plan, *ACM SIGMOD Record,* 2013, vol. 42, no. 1, pp. 44–49.
4. Harizopoulos S., Abadi D., Madden S., and Stonebraker, M., OLTP through the looking glass, and what we found there, in *Proc. of the ACM SIGMOD Int. Conf. on Management of Data,* 2008, pp. 981–992.
5. Williams, M.H. and Zhou, S., Data placement in parallel database systems, *Parallel database techniques*, 1998, pp. 203–218.
6. TOP500: 500 most powerful computer systems in the world. http://top500.org.
7. Kostenetskii, P.S. and Sokolinsky, L.B., Simulation of hierarchical multiprocessor database systems, *Program. Comput. Software,* 2013, vol. 39, no. 1, pp. 10–24.
8. 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.
9. Lima, A.A., Furtado, C., Valduriez, P., and Mattoso, M., Parallel OLAP query processing in database clusters with data replication, *Distributed Parallel Databases,* 2009, vol. 25, no. 1–2, pp. 97–123.
10. Pukdesree, S., Lacharoj, V., and Sirisang, P., Performance evaluation of distributed database on PC cluster computers, *WSEAS Trans. Comput.,* 2011, vol. 10, no. 1, pp. 21–30.
11. Sokolinsky, L.B., *Parallel Database Systems.* Moscow: Mosk. Gos. Univ., 2013.
12. Taniar, D., Leung, C.H.C., Rahayu, W., and Goel, S., *High Performance Parallel Database Processing and Grid Databases*, Wiley, 2008.
13. Sokolinsky, L.B., Survey of architectures of parallel database systems, *Program. Comput. Software,* 2004, vol. 30, no. 6, pp. 337–346.
14. Deshmukh, P.A., Review on main memory database, *Int. J. Comput. Commun. Technol.,* 2011. vol. 2, no. 7, pp. 54–58.
15. Garcia-Molina, H. and Salem, K., Main memory database systems: An overview, *IEEE Trans. Knowl. Data Eng.,* 1992, vol. 4, no. 6, pp. 509–516.
16. Plattner, H. and Zeier, A., *In-Memory Data Management: An Inflection Point for Enterprise Applications,* Springer, 2011.
17. LeHong, H., Fenn, J., Hype Cycle for Emerging Technologies, Research Report, Gartner, 2013.
18. Chaudhuri, S. and Dayal, U., An overview of data warehousing and OLAP technology, *SIGMOD Record,* 1997, vol. 26, no. 1, pp. 65–74.
19. Furtado, P., A survey of parallel and distributed data warehouses, *Int. J. Data Warehousing Mining,* 2009, vol. 5, no. 5, pp. 57–77.
20. Golfarelli, M. and Rizzi, S., A survey on temporal data warehousing, *Int. J. Data Warehousing Mining,* 2009, vol. 5, no. 1, pp. 1–17.
21. Oueslati, W. and Akaichi, J., A survey on data warehouse evolution, *Int. J. Database Management Syst.,* 2010, vol. 2, no. 4, pp. 11–24.
22. Boncz, P.A. and Kersten, M.L., MIL primitives for querying a fragmented world, *VLDB J.,* 1999, vol. 8, no. 2, pp. 101–119.
23. Boncz, P.A., Zukowski, M., and Nes, N., MonetDB/X100: Hyper-pipelining query execution, in *Proc. of the Second Biennial Conference on Innovative Data Systems Research (CIDR),* 2005, pp. 225–237.

24. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S.R., O'Neil E.J., O'Neil, P.E., Rasin, A., Tran, N., and Zdonik, S.B., C-Store: A column-oriented DBMS in *Proc. of the 31st Int. Conf. on Very Large Data Bases (VLDB'05),* 2005, pp. 553–564.

25. Abadi, D.J., Madden, S.R., and Hachem, N., Column-stores vs. row-stores: How different are they really? in *Proc. of the 2008 ACM SIGMOD Int. Conf. on Management of Data,* 2008, pp. 967–980.

26. Abadi, D.J., Madden, S.R., and Ferreira, M., Integrating compression and execution in column-oriented database systems, in *Proc. of the 2006 ACM SIGMOD Int. Conf. on Management of Data,* 2006, pp. 671–682.

27. Chernyshev, G.A., Organization of the physical level of column-oriented DBMSs, *Tr. St. Petersburg Inst. Infor. Avtom. Ross. Akad. Nauk SPIIRAN,* 2013, no. 7 (30), pp. 204–222. http://www.proceedings.spiiras.nw.ru/ojs/index.php/sp/index.

28. Abadi, D.J., Boncz, P.A., and Harizopoulos, S., Column-oriented database Systems, in *Proc. of the VLDB Endowment,* 2009, vol. 2, no. 2, pp. 1664–1665.

29. Abadi, D.J., Boncz, P.A., Harizopoulos, S., Idreos, S., and Madden S., The design and implementation of modern column-oriented database systems, *Foundations Trends Databases,* 2013, vol. 5, no. 3, pp. 197–280.

30. Plattner, H., A common database approach for OLTP and OLAP using an in-memory column database, in *Proc. of the 2009 ACM SIGMOD Int. Conf. on Management of Data,* 2009, pp. 1–2.

31. Copeland, G.P. and Khoshafian, S. N., A decomposition storage model, in Proc. of the 1985 ACM SIGMOD *Int. Conf. on Management of Data,* 1985, pp. 268–279.

32. Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, S., and Kersten, M.L., MonetDB: Two decades of research in column-oriented database architectures, *IEEE Data Eng. Bull.,* 2012, vol. 35, no. 1, pp. 40–45.

33. Zukowski, M., Heman, S., Nes, N., and Boncz, P., Super-scalar RAM-CPU cache compression, *Proc. of the 22nd Int. Conf. on Data Engineering,* 2006, pp. 59–71.

34. Chen, Z., Gehrke, J., and Korn, F., Query optimization in compressed database systems, in *Proc. of the 2001 ACM SIGMOD International Conference on Management of Data,* 2001, pp. 271–282.

35. Westmann, T., Kossmann, D., Helmer, S., and Moerkotte, G., The implementation and performance of compressed databases, *ACM SIGMOD Record,* 2000. vol. 29, no. 3, pp. 55–67.

36. Aghav, S., Database compression techniques for performance optimization, in *Proc. of the 2010 2nd Int. Conf. on Computer Engineering and Technology (ICCET),* 2010, pp. 714–717.

37. Lemke, C., Sattler, K.-U., Faerber, F., Zeier, A., Speeding up queries in column stores: A case for compression, *Proc. of the 12th Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK'10),* 2010, pp. 117–129.

38. Ramamurthy, R., Dewitt, D., and Su, Q., A case for fractured mirrors, in *Proc. of the VLDB Endowment,* 2002, vol. 12, no. 2. pp. 89–101.

39. Khoshafian, S., Copeland, G., Jagodis, T., Boral, H., and Valduriez, P., A query processing strategy for the decomposed storage model, in *Proc. of the Third Int. Conf. on Data Engineering,* 1987, pp. 636–643.

40. Bruno, N., Teaching an old elephant new tricks, in *Online Proc. of the Fourth Biennial Conf. on Innovative Data Systems Research (CIDR 2009),* 2009. http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_2.pdf.

41. El-Helw, A., Ross, K.A., Bhattacharjee, B., Lang, C.A., and Mihaila, G.A., Column-oriented query processing for row stores, *Proc. of the ACM 14th Int. Workshop on Data Warehousing and OLAP (DOLAP '11),* 2011, pp. 67–74.

42. Larson, P.-A., Clinciu, C., Hanson, E.N., Oks, A., Price, S.L., Rangarajan, S., Surna, A., and Zhou, Q., SQL server column store indexes, in *Proc. of the 2011 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '11),* 2011, pp. 1177–1184.

43. TPC Benchmark DS – Standard Specification, Transaction Processing Performance Council, 2015. http://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-ds_v2.1.0.pdf.

44. Shapiro, M. and Miller, E., Managing databases with binary large objects, in *16th IEEE Symp. on Mass Storage Systems,* 1999, pp. 185–193.

45. Padmanabhan, S., Malkemus, T., Agarwal, R., and Jhingran A., Block oriented processing of relational database operations in modern computer architectures, in *Proc. of the 17th Int. Conf. on Data Engineering,* 2001, pp. 567–574.

46. O'Neil, P.E., Chen, X., and O'Neil, E.J., Adjoined dimension column index to improve star schema query performance, in *Proc. of the 24th Int. Conf. on Data Engineering (ICDE 2008),* 2008, pp. 1409–1411.

47. O'Neil, P.E., O'Neil, E.J., and Chen, X., The Star Schema Benchmark (SSB), Revision 3, June 5, 2009. http://www.cs.umb.edu/ poneil/StarSchemaB.PDF.

48. O'Neil, P.E., O'Neil, E.J., Chen, X., and Revilak, S., The star schema benchmark and augmented fact table indexing: performance evaluation and benchmarking, *in First TPC Technology Conference (TPCTC 2009),* 2009, pp. 237–252.

49. Garcia-Molina, H., Ullman, J.D., and Widom, J., *Database Systems: The Complete Book, Upper Saddle River,* NJ: Prentice Hall, 2002.

50. Ivanova, E. and Sokolinsky, L.B., Join decomposition based on fragmented column indices, *Lobachevskii J. Math.*, 2016, vol. 37, no. 3, pp. 255–260.

51. Ivanova, E.V. and Sokolinsky, Using Intel Xeon Phi Coprocessors for execution of natural join on compressed data, *Vychisl. Metody Program: Novye Vychisl. Tekhnol,* 2015, vol. 16, no. 4, pp. 534–542.

52. Ivanova, E.V. and Sokolinsky, L.B., Decomposition of the grouping operation based on distributed column indexes, Nauka YurGU: *Materialy 67 nauchnoi konferentsii professorsko-prepodavatel'skogo sostava, aspirantov i sotrudnikov, Sec. Estestvennykh nauk* (Proc. of the Conf. of the faculty and postgraduates of Yuzhno-Ural'sk State Unversity, Ser. Natural Sciences), 2015, pp. 15–22.

53. Ivanova, E.V. and Sokolinsky, L.B., Decomposition of intersection and join operations based on the domain-

interval fragmented column indexes, *Vestn. Yuzhno-Ural'sk. Gos. Univ., Ser. Vychisl. Mat. Inform.,* 2015, vol. 4, no. 1, pp. 44–56.

54. Ivanova, E.V. and Sokolinsky, L.B., Parallel decomposition of relational operations based on fragmented column indexes, *Vestn. Yuzhno-Ural'sk. Gos. Univ., Ser. Vychisl. Mat. Inform.,* 2015, vol. 4, no. 4, pp. 80–100.
55. Deutsch, P. and Gailly, J.-L., ZLIB Compressed Data Format Specification version 3.3. RFC Editor, 1996. https://www.ietf.org/rfc/rfc1950.txt.
56. Roelofs, G., Gailly, J., and Adler, M., Zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library. http://www.zlib.net/.
57. Deutsch, P., DEFLATE Compressed Data Format Specification version 1.3. RFC Editor, 1996. https://www.ietf.org/rfc/rfc1951.txt.
58. Kostenetskiy, P.S. and Safonov, A.Y., SUSU Supercomputer Resources, in *Proc. of the 10th Annual Int. Scientific Conf. on Parallel Computing Technologies (PCT 2016),* CEUR Workshop Proceedings, Vol. 1576, CEUR-WS 2015, pp. 561–573.
59. Massively Parallel Supercomputer RSC PetaStream. http://rscgroup.ru/ru/our-solutions/massivno-parallelnyy-superkompyuter-rsc-petastream.
60. TPC Benchmark H – Standard Specification. Transaction Processing Performance Council, 2014. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf.
61. Ivanova, E.V. and Sokolinsky, L.B., Columnar database coprocessor for computing cluster system, *Vestn. Yuzhno-Ural'sk. Gos. Univ., Ser. Vychisl. Mat. Inform.,* 2015, vol. 4, no. 4, pp. 5–31.
62. Gray, J., Sundaresan, P., Englert, S., Baclawski, K.,and Weinberger, P.J., Quickly generating billion-record synthetic databases in *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data,* 1994, pp. 243–252.

Translated by A. Klimontovich