Join Execution Using Fragmented Columnar Indices on GPU and MIC

Elena V. Ivanova, Stepan O. Prikazchikov, and Leonid B. Sokolinsky

South Ural State University, Chelyabinsk, Russia
{Elena.Ivanova,prikazchikovso,Leonid.Sokolinsky}@susu.ru

Abstract. The paper describes an approach to the parallel natural join execution on computing clusters with GPU and MIC Coprocessors. This approach is based on a decomposition of natural join relational operator using the column indices and domain-interval fragmentation. This decomposition admits parallel executing the resource-intensive relational operators without data transfers. All column index fragments are stored in main memory. To process the join of two relations, each pair of index fragments corresponding to particular domain interval is joined on a separate processor core. Described approach allows efficient parallel query processing for very large databases on modern computing cluster systems with many-core accelerators. A prototype of the DBMS coprocessor system was implemented using this technique. The results of computational experiments for GPU and Xeon Phi are presented. These results confirm the efficiency of proposed approach.

Keywords: big data · parallel query processing · column indices · domain-interval fragmentation · natural join · GPU · MIC

1 Introduction

Nowadays, human scientific and practical activities create the new challenges that demand big data processing. According to IDC study [1], the amount of digital data is doubling in size every two years, and by 2020 the digital universe – the amount of digital data created and replicated – will reach 44 zettabytes, or 44 trillion gigabytes. One of the popular ways to process efficiently big data is using the parallel database system, which are able to process data in parallel on the high performance system with distributed memory [2–5]. The traditional approach for database storing is row-oriented representation. However, column-oriented database systems have been shown to perform more than an order of magnitude better than row-oriented database systems (”row-stores”) on analytical workloads such as those found in data warehouses, decision support, and business intelligence applications. The elevator pitch behind this performance difference is straightforward: column-stores are more I/O efficient for read-only queries since they only have to read from disk (or from memory) those attributes

accessed by a query [6]. Column-oriented databases are particularly well suited for compression because data of the same type is stored in consecutive sections. This makes it possible to use compression algorithms specifically tailored to patterns that are typical for the data type [7].

In recent years, more and more many-core processors are superseding sequential ones. Increasing parallelism, rather than increasing clock rate, has become the primary engine of processor performance growth, and this trend is likely to continue. Particularly, today's GPUs (Graphic Processing Units) and Intel's MIC (Many Integrated Cores), greatly outperforming traditional CPUs in arithmetic throughput and memory bandwidth, can use hundreds of parallel processor cores to execute tens of thousands of threads [8]. Recent trends in new hardware and architectures have gained considerable attention in the database community. Processing units such as GPU or MIC provide advanced capabilities for massively parallel computation. Database processing can take advantage of such units not only by exploiting this parallelism, e.g., in query operators (either as task or data parallelism), but also by offloading computation from the Central Processing Unit (CPU) to these coprocessors, saving CPU time for other tasks [9].

According to this, the problem of developing new efficient methods of parallel database processing on modern compute clusters with many-core accelerators using column-oriented representation and data compression is important. To meet this goal, we offer a special type of index structures called distributed column indices. Distributed column indices allow to perform a decomposition of relational operators, which admits the efficient parallel execution of them on computing cluster system, equipped with many-core accelerators. In this paper, we consider the decomposition of the natural join operator. We will use the notation from [10]. The symbol "$\circ$" will be used to denote the operation of concatenation of the tuples.

2 Column Index

Let $R(A, B_1, \ldots, B_u)$ be the R relation with *surrogate key* (surrogate) A and the following attributes: $B_1, \ldots, B_u$. Tuples of R have length of $u+1$ and form of $(a, b_1, \ldots, b_u)$, where $a \in \mathbb{Z}_{\geq 0}$ and $\forall j \in \{1, \ldots, u\} \left(b_j \in \mathfrak{D}_{B_j}\right)$. Here, $\mathfrak{D}_{B_j}$ is the domain of attribute B_j. Let $r.B_j$ denote a value of attribute B_j. Let $r.A$ denote a value of the surrogate key of tuple r: $r = (r.A, r.B_1, \ldots, r.B_u)$. The *surrogate key* of relation R has the property: $\forall r', r'' \in R\,(r' \neq r'' \Leftrightarrow r'.A \neq r''.A)$. Define *tuple address* as a surrogate key value of the tuple. To get the tuple by its address, we will use $\&_R$ *dereferencing function*: $\forall r \in R\,(\&_R(r.A) = r)$.

Let $R(A, B, \ldots)$, $T(R) = n$ be given. Let a linear order be defined on set $\mathfrak{D}_B$. The *column index* $I_{R.B}$ for attribute B of relation R is an ordered relation, which satisfies the following requirements:

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

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

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

Condition (1) means that the sets of surrogate keys of column index and indexed relation are equal. Condition (2) means that index elements are sorted in ascending order of values of attribute B. Condition (3) means that attribute A of an index element contains the address of tuple of R, which has the same value of B attribute as the corresponding element of column index has.

From the intensional point of view, the column index $I_{R.B}$ is a table with two columns A and B (Fig. 1). The number of rows in the column index is equal to the number of rows in the indexed table. Column B of index $I_{R.B}$ contains all the values of column B in table R (including duplicates). These values are sorted in ascending order inside column index.

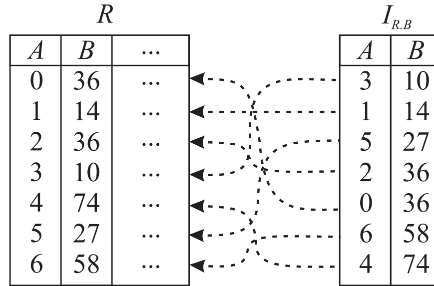

Fig. 1. Column index.

3 Domain-Interval Fragmentation

Let a total ordering relation be defined on domain $\mathfrak{D}_B$. Divide $\mathfrak{D}_B$ into $k > 0$ nonintersecting intervals:

$$\left.\begin{array}{l} V_0 = [v_0; v_1]\,, V_1 = (v_1; v_2]\,, \ldots, V_{k-1} = (v_{k-1}; v_k]\,; \\ v_0 < v_1 < \ldots < v_k; \\ \mathfrak{D}_B = \bigcup\limits_{i=0}^{k-1} V_i. \end{array}\right\} \tag{4}$$

Define *interval fragmentation function* on domain $\mathfrak{D}_B$ as $\varphi_{\mathfrak{D}_B} : \mathfrak{D}_B \to \{0, \ldots, k-1\}$. This function satisfies the following requirement:

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

Let column index $I_{R.B}$ be given for relation $R\,(A, B, \ldots)$ with attribute B on domain $\mathfrak{D}_B$. Let interval fragmented function $\varphi_{\mathfrak{D}_B}$ be defined on domain $\mathfrak{D}_B$. The function

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

is called *domain-interval fragmentation function* for index $I_{R.B}$, if it satisfies the following requirement:

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

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

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

It means that the ith fragment contains tuples, which have values of attribute B from the ith domain interval. This fragmentation is called the *domain-interval fragmentation*. The number of fragments is the *degree of fragmentation*.

The domain-interval fragmentation has the following fundamental properties, which follow directly from its definition:

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

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

4 Decomposition of the Natural Join Operator

Let two relations be given:

$$R\left(A, B_1, \ldots, B_u, C_1, \ldots, C_v\right) \tag{11}$$

and

$$S\left(A, B_1, \ldots, B_u, D_1, \ldots, D_w\right). \tag{12}$$

Let two sets of column indices be given for attributes $B_1, \ldots, B_u$:

$$I_{R.B_1}, \ldots, I_{R.B_u}; \tag{13}$$

$$I_{S.B_1}, \ldots, I_{S.B_u}. \tag{14}$$

Let domain-interval fragmentation of degree k be defined for these indices:

$$I_{R.B_j} = \bigcup_{i=0}^{k-1} I^i_{R.B_j}; \tag{15}$$

$$I_{S.B_j} = \bigcup_{i=0}^{k-1} I^i_{S.B_j}. \tag{16}$$

Let

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

for all $i = 0, \dots, k-1$ and $j = 1, \dots, u$. Define

$$P_j = \bigcup_{i=0}^{k-1} P_j^i. \tag{18}$$

Let

$$P = \bigcap_{j=1}^{u} P_j. \tag{19}$$

Define

$$Q = \{r \circ (s.D_1, \dots, s.D_w) | r \in R \wedge s \in S \wedge (r.A, s.A) \in P\}. \tag{20}$$

Then $\pi_{*\backslash A}(R) \bowtie \pi_{*\backslash A}(S) = \pi_{*\backslash A}(Q)$ [11].

Note that calculation of P_j^i by (17) can be done in parallel on k different processors without data exchange. It ensures a near-linear speedup.

5 Performance Evaluation

The described approach was implemented as a prototype of DBMS coprocessor system. The source code of the program is openly available in the public GitHub repository [13]. Column indices and domain-interval fragmentation were evaluated using this prototype.

We generated a synthetic database, which consisted of two relations R and S with one common attribute B of integer type. In R relation, B was a primary key. In S relation, B was a foreign key. Numbers of tuples were following: $T(R) = 600\,000$ and $T(S) = 60\,000\,000$. Relation S was generated in two ways. First, we used uniform distribution for column $S.B$. Second, we used rule 80/20 [12] for column $S.B$. Fragmented column indices $I_{R.B}$ and $I_{S.B}$ was created for columns $R.B$ and $S.B$. All fragments of both indices were loaded into the memory of many-core coprocessor. Each pair of corresponding fragments of $I_{R.B}$ and $I_{S.B}$ was processed in separate thread by the merge join algorithm.

The experiments were done using the following equipment:

- NVIDIA Tesla K40m with 2880 CUDA Cores (maximum number of threads per block is 1024) and 12 Gb memory size;
- Intel Xeon Phi SE10X accelerator with 61 cores and 8 Gb memory size.

In all the experiments shown on figures 2–4, we varied the number of fragments, into which indices were splited (abscissa axis), and measured total time of join execution (ordinate axis).

In the first series of experiments, we used database with uniform distribution of B values in relation S. Using such a database, we investigated the influence of the number of threads per CUDA block for GPU during join processing. The results are presented in Fig. 2 a). We explored the following three cases: 128, 256 and 512 threads per CUDA block. The experiments show that maximum

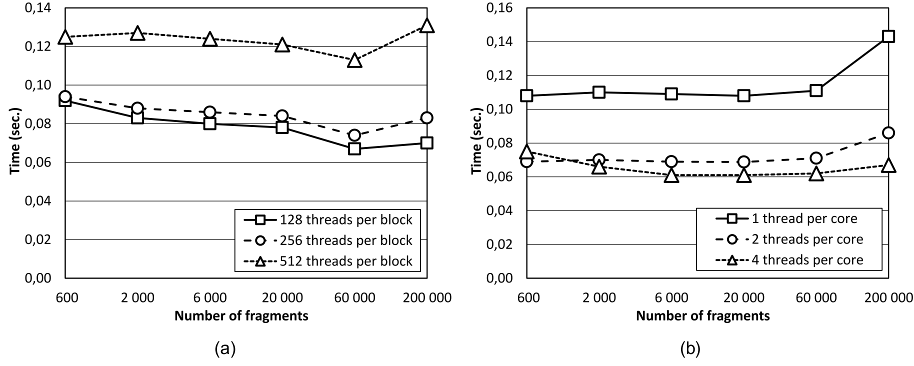

Fig. 2. Dependence of join execution time on the number of fragments for a) GPU and b) Xeon Phi (uniform distribution).

speedup on GPU is achieved for uniform distribution when we use 128 threads per CUDA block. The similar experiments were performed for Xeon Phi (see Fig. 2, a). The results show that maximum speedup on Xeon Phi is achieved for uniform distribution when we use 4 threads per core. In all cases, the performance of GPU is very close to the performance of Xeon Phi.

For skewed distribution (rule 80/20) of B values in relation S, we are seeing the very opposite picture (see Fig. 3). In such a way, we have 20% of very

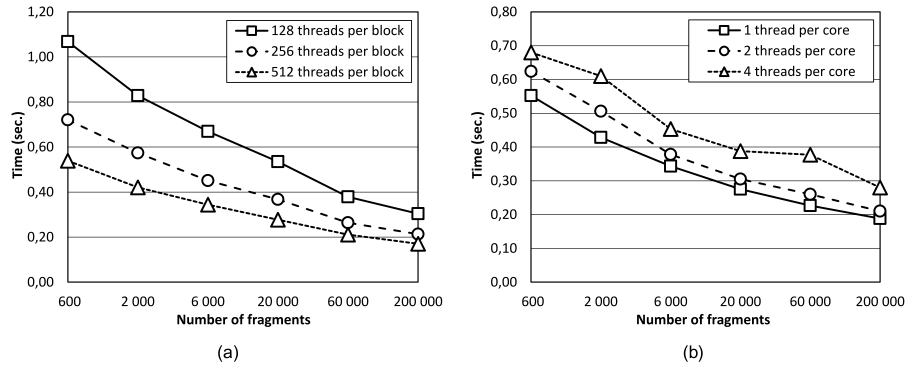

Fig. 3. Dependence of join execution time on the number of fragments for a) GPU and b) Xeon Phi (80/20 rule).

"big" fragments and 80% of very "small" fragments for column index $I_{S.B}$. In this situation, the maximum performance is achieved on GPU, when we use the greater number of threads per CUDA block. For the skewed data, the maximum performance is achieved on Xeon Phi, when we use the smaller number of threads

per core. And again, the performance of GPU is very close to the performance of Xeon Phi for skewed data.

In the last series of experiments, we investigated how our algorithm is robust with respect to data skew. To simulate data skew, 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 distinct value of $S.B$, some weight coefficient p_i $(i = 1, \ldots, N)$ is assigned by the formula

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

where N is the number of distinct values for attribute $S.B$ and $H_N^s = 1^{-s} + 2^{-s} + \ldots + N^{-s}$ is the N-th harmonic number of order s. The case of $\theta = 0$ corresponds to uniform distribution. The case of $\theta = 0.5$ corresponds to 45/20 rule, in accordance with which 20% distinct values have 45% occurrences in column B of relation S. The case of $\theta = 0.73$ corresponds to 65/20 rule and the case of $\theta = 0.86$ corresponds to 80/20 rule. In these experiments, we used 512 threads per CUDA block for GPU and 1 thread per core for Xeon Phi. The results are presented in Fig. 4. We see that load balancing can be effectively managed

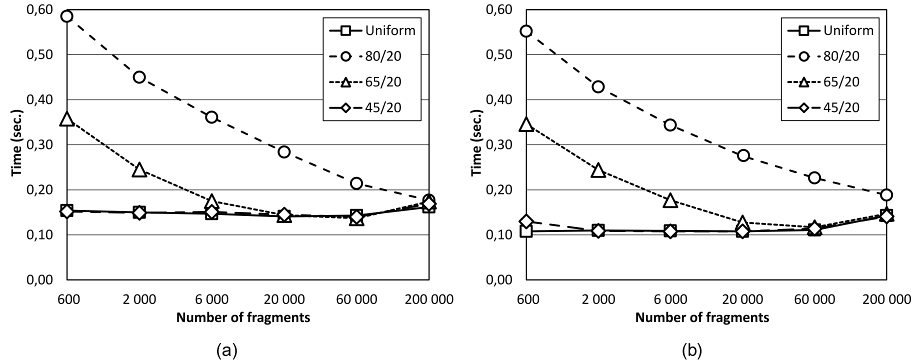

Fig. 4. The influence of number of fragments on load balancing for a) GPU and b) Xeon Phi.

by increasing the number of fragments, into which we split the column indices on GPU as well as on Xeon Phi. When the number of fragments much greater than the number of threads, one thread can handle many small fragments, while another thread will process one big fragment. If the number of fragments equals to the number of threads, we have no such a possibility.

Performed experiments let us make three main conclusions. First, the proposed approach based on fragmented column indices allows to perform resource-intensive join operator for $T(I_{R.B}) = 600\,000$ and $T(I_{S.B}) = 60\,000\,000$ during less then 0.2 second on Xeon Phi coprocessor or NVIDIA Tesla GPU. Second, the

performance of Xeon Phi is very close to the performance of NVIDIA Tesla GPU for such kind of workload. Third, the described approach eliminates data transfer, hence we may expect a near-linear speedup on computing cluster systems with thousands nodes equipped with many-core accelerators.

6 Related Work

Binary table model was introduced in the paper [14]. On the basis of this model, several column-oriented DBMS were designed. As it was demonstrated by work [15] and [16], column-oriented systems offer an order-of-magnitude performance improvement over traditional row-oriented systems for analytical processing workloads, such as those found in data warehouses or decision support systems. One of the main disadvantages of column-oriented DBMS is lacking the optimization technique, which is intrinsic to relational (row-oriented) DBMS. The work [6] investigated column-oriented simulation in a relational DBMS via the following techniques: vertical partitioning, index-only plans and materialized views. The investigation showed that such techniques do not improve the performance of row stores for analytical processing workloads. To overcome the problems faced with work [6], the work [17] introduced two new operators: Index Merge and Index Merge Join. The algorithms presented in this paper were designed specifically to take advantage of parallel processing whenever possible. Another approach was proposed in work [18]. This paper introduced a new index type, column store indexes, where data is stored column-wise in compressed form. Column store indexes are intended for data-warehousing workloads where queries typically process large numbers of rows but only a few columns. To further speed up such queries, the paper [18] also introduced a new query processing mode, batch processing, where operators process a batch of rows (in columnar format) at a time instead of a row at a time.

7 Conclusions

In this article, we presented a decomposition of the natural join operator based on the column indices and the domain-interval fragmentation. Our approach was evaluated using the prototype DBMS coprocessor system. Experiments showed its efficiency for a resource-intensive natural join operator. Proposed approach can be used on computing cluster systems with many-core accelerators. Described technique is suitable for data warehouse workloads as well as for OLTP workloads.

As a direction of a future research, we are going to use described approach for the decomposition of another relational operators and compare speedup with existing DBMS.

Acknowledgments. The study was supported by the Ministry of education and science of Russia under Federal targeted program “Research and development

in priority fields of scientific and technological complex of Russia in 2014-2020" (Governmental contract No. 14.574.21.0035).

References

1. Turner, V., Gantz, J.F., Reinsel, D., Minton, S.: The digital universe of opportunities: rich data and the creasing value of the internet of things. IDC, White Paper (2014). http://idcdocserv.com/1678
2. Sokolinsky, L.B.: Survey of architectures of parallel database systems. Programming and Computer Software. Vol. 30, No. 6, pp. 337–346 (2004)
3. Lepikhov, A.V., Sokolinsky, L.B.: Query processing in a DBMS for cluster systems. Programming and Computer Software. Vol. 36, No. 4, pp. 205–215 (2010)
4. Pan, C.S., Zymbler, M.L.: Taming elephants, or how to embed parallelism into PostgreSQL. In: Decker, H., Lhotská, L., Link, S., Basl, J., Tjoa, A M. (eds.) DEXA 2013. LNCS, vol. 8055, pp. 153–164 (2013)
5. Besedin, K.Y., Kostenetskiy, P.S.: Simulating of query processing on multiprocessor database systems with modern coprocessors. In: 37th International Convention, MIPRO 2014, pp. 1835–1837. IEEE (2014)
6. Abadi, D.J., Madden, S.R., Hachem, N.: Column-Stores vs. Row-Stores: How Different Are They Really? In: SIGMOD'08, pp. 967–980. ACM, New York (2008)
7. Abadi, D.J., Madden, S.R., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: SIGMOD'06, pp. 671–682 (2006)
8. Fang, J., Varbanescu, A.L., Sips, H.: Sesame: a user-transparent optimizing framework for many-core processors. In: 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid2013, pp. 70–73. IEEE (2013)
9. Breß, S., Beier, F., Rauhe, H., Sattler, K.-U., Schallehn, E., Saake, G.: Efficient Co-Processor Utilization in Database Query Processing. Information Systems. Vol. 38, No. 8, pp. 1084–1096 (2013)
10. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book (2nd Edition). Prentice Hall, New Jersey (2008)
11. Ivanova, E.V., Sokolinsky, L.B.: Decomposition of Intersection and Join Operations Based on the Domain-Interval Fragmented Column Indices. Bulletin of the South Ural State University. Series "Computational Mathematics and Software Engineering". Vol. 4, No. 1, pp. 44–56 (2015)
12. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.: Quickly Generating Billion-record Synthetic Databases. In: SIGMOD'94, pp. 243–252 (1994)
13. Prototype of DBMS coprocessor system, https://github.com/elena-ivanova/colomnindices
14. Copeland, G.P., Khoshafian, S.N.: A decomposition storage model. In: SIGMOD 1985, pp. 268–279. ACM, New York (1985)
15. Boncz, P.A., Zukowski, M., Nes, N.: MonetDB/X100: Hyper-Pipelining Query Execution. In: CIDR 2005, pp. 225–237. (2005)
16. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., ONeil, E., ONeil, P., Rasin, A., Tran, N., Zdonik, S.: C-Store: A Column-Oriented DBMS. In: VLDB 2005, pp. 553–564. VLDB Endowment (2005)
17. El-Helw, A., Ross, K.A., Bhattacharjee, B., Lang, C.A., Mihaila, G.A.: Column-oriented query processing for row stores. In: DOLAP 2011, pp. 67–74. ACM, New York (2011)

18. Larson, P., Clinciu, C., Hanson, E.N., Oks, A., Price, S.L., Rangarajan, S., Surna, A., Zhou, Q.: SQL server column store indexes. In: SIGMOD Conference 2011, pp. 1177–1184. ACM, New York (2011)