Empirical study of parallel SQL query execution

Kirill Smirnov, George Chernishev
Saint-Petersburg University
Kirill.k.smirnov@math.spbu.ru, Chernishev@gmail.com

Abstract. In this paper we experiment with two major types of query parallelization techniques - intra and inter operator parallelism and their combinations. We evaluate these techniques applied to a query tree with a number of join operators in the multithreaded environment. In our experiments we vary thread count, buffer size, workload characteristics, measure the relative performance of several distinct join algorithms and observe several modes which appear in the series of experiments.

Keywords: database, experimentation, parallelism, multithreading, query execution

1. Introduction

Distributed database systems (DDBS) had appeared more than three decades ago and since that time there is constant demand in DDBS technologies. The driving force of this demand is the ever-growing volume of data which require processing. The scalability of distributed databases gives us a promise to handle this mass of data. The present demands new effective algorithms, models, architectural solutions so the field remains active and relevant.

In early 90s a substantial portion of research effort was dedicated to ascertainment of how we should divide the work (speaking more precisely a query execution plan or a part of it) between processing elements. This research effort was not specific to a distributed, but aimed for a wide range of database systems. Extensive experiments were conducted regarding the query plan execution techniques, and a pool of knowledge was formed. Now, query execution parallelism is the commonplace and is employed by a various commercial systems.

Despite these experiments, nowadays, the same questions had become relevant again due to the following reasons:

1) Shortage of research concerning utilization of threads and processes. Research papers usually had following distinct attitudes: use one process/thread per processing element, allocate much higher number of threads than processing elements [1] or not to touch this issue in detail.
2) The advancement of hardware – a few generations of CPUs had been developed. The improvements were not just clock speeds and cache size growth, but also more sophisticated ones. Notable examples are multithreading and multicore processing.
3) Advancement of software – mainly thread/process scheduling software involved.

Even alone items two and three can alter or invalidate principles of thread usage in database system.

The goal of this study is to determine how we should utilize threads designing a distributed database which runs on a modern hardware. We re-evaluate a few classical approaches to process joins in a distributed databases taking into consideration threading aspect. There are two ideas being considered: intra and inter operation parallelism and their combinations. The first one is how we can divide our work processing inside one operator treating our cores as separate processing elements. The latter covers matters of executing several operators in different threads.

This paper is organized as follows: in section 2 we briefly review some crucial aspects of DDBS which are related to our study; in section 3 we present our prototype of DDBS and provide some insights into implementation details. Section 4 reviews several possible ways of thread usage for parallelizing join operation in our system. Sections 5 and 6 describe our experiments and their interpretation results respectively.

2. Related work and DDBS technology overview

2.1 Overview of important technology concepts of DDBS

Extensive research in this field had been conducted for decades and a lot of knowledge had been accumulated. In this section we will review some part of it (by giving definitions at least), which will be referred to in this study.

The first aspect to consider designing a distributed database is its architecture. Here we imply the hardware architecture, how the data is distributed, stored and processed on the lowest level. Most popular architectures are shared-nothing, shared-disk and shared memory [1]. This choice impacts all further decisions regarding all constituent parts the design.

The second aspect is the system processing model. There are two basic alternatives and a lot of different combinations. This choice is the choice between pipelined execution and a materialized one. The former is the policy when a tuple passes through all operators of query tree without “stops”. The latter approach means that some “stops” are allowed. The stop is essentially a blocking operator [2], e.g. an operator which requires all tuples to carry out its task. The sort operator is the blocking one, while predicate filtering operator is the non-blocking. The pipelined execution has a huge benefit over materialization-based, namely all operations can be done in memory. While materialization often requires extensive disk operations which are caused by intermediate result being too large to fit into memory.

Another important aspect to consider is the tuple routing mechanisms. There are a few approaches, namely each tuple passes the operators in the same order, or not. The latter is known as adaptive execution [2]. The next major choice is the choice between data-driven (also called producer-consumer) and demand-driven (known as workflow) execution. Demand-driven execution works by “pumping” tuples from the leaves of the tree to the root, through the operators involved. In this scheme, each operator of a higher rank sends a notification request to its children to get tuples. Consequently, this request propagates to the leaf nodes, which usually deal with disk-based operations. The data-driven approach is the vice-versa: the leaf nodes generate intermediate result, notify its parents and submit the result. The data-driven approach is the more appealing one in case of distributed database system, because it has great potential for parallelization. Also, it offers more freedom and convenience implementing leaf operators. However, this approach requires great care programming it, due to synchronization and work distribution issues.

Yet another pair of aspects to consider designing a distributed database system is the operator parallelization methods. Considering a single query optimization task one can name the following ways: inter- and intra-operator parallelism. Intra-operation parallelism refers to the parallel execution of a single operator and inter-operation parallelism means executing a multiple operators in parallel. The first heavily relies on the data partitioning techniques and the latter is more architectural-dependent.

2.2 Threads and processes in databases

Threading and processing aspects are of critical importance and should be taken into consideration building high performance database engine. However, there is shortage of studies related to threading and its impact on operator parallelism. With the increasing popularity of multicore processors the demand for the reevaluation of these threading technologies had risen greatly. Let’s briefly review the threading in databases.

One of the first most prominent database systems which implemented threading was Volcano [3]. The following conclusions were presented [3]: it is beneficial to use multiple threads running concurrently on the same processor. However, running each operator in its own dedicated thread is unfeasible: synchronization and thread switching will eat up all the benefits. The same results presented here [4], also another important question is mentioned: how much memory we should provide to the thread. This question affects the number of threads which could be run effectively. In this paper [2] an adaptive query processing is considered, and shown that each adaptive symmetric hash join operator can work effectively in the separate thread.

2.3 Current state of parallelism in contemporary DBMS

In this section we deal not with the idea of running several transactions concurrently, but consider the possibility of executing several select queries in parallel. The idea of executing a single query in parallel, which in most cases implies inter or intra operator parallelism had become widely applied since 90. Now, almost all major DBMS have capabilities for query parallelism. In most systems the parallelism is controlled using the Degree Of Parallelism (DOP) mechanism. The DOP is the value which describes how much of logical processing elements we would like to dedicate to a query. User may specify the minimum and maximum values or use default settings. Then, the DBMS schedules the work and maps logical processing element into real ones – processors or cores. DBMS can also be tuned to cancel query execution if there are not enough free processing elements. The resulting performance heavily depends on the number of available processors, the data and query specifics and so on. Thus, it requires careful engine tuning to achieve good performance.

We can name Microsoft SQL Server [5], Oracle [6], IBM DB2 [7] products among the commercial systems that offer mechanisms for such parallelism. At the same time, to the best of our knowledge, neither MySQL, nor PostgreSQL have this feature. For these products, there are several third party frameworks exist. These frameworks offer a substitute to query parallelism which can be described as a layer which allows parallelization of a query via horizontal partitioning of the underlying data in a shared-nothing architecture.

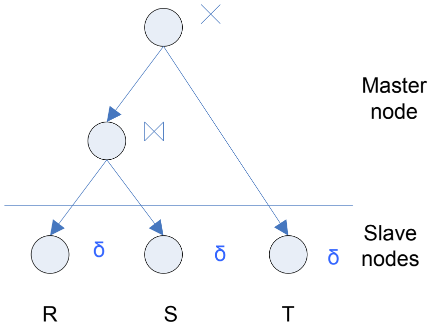
Unfortunately, the control interfaces of contemporary DBMS supporting query parallelism do not provide sufficient degree of freedom. We can't have much influence on how the threads would be used. Thus, we can't use these systems to perform our experiments.

3. Our system

The experiments which are described in this work were conducted using our prototype of distributed query engine developed by authors. This system is devised for simple SPJ conjunctive queries [2] and is capable of serving simultaneously a substantial number of clients. The architecture of the system is described on a Pic. 1 and it is essentially a shared-nothing design with a number of processing nodes distributed among network. We partition nodes into two types: master and slave. There is only one master node per setup and it is responsible for handling incoming queries. For purpose of paper's experiments we run slave processes on the same computer.

Our system is based on assumption that data are pre-distributed and reside on slave nodes. The data are collection of relations, where each attribute may be either integer value or string. Each relation is partitioned horizontally and may be replicated.

The overall design of the system resembles a classical Volcano [4], where QEP (query execution plan) is defined as a tree of various operators and candidate tuples have to pass them in order for result to be produced. In our system there are the following operators: selection (actually consisting of both selection and projection), join and cross-product. Details regarding our implementation of join operator can be found in part 4, but it is essential to note that the overall architecture was designed for non-blocking operations, so the join operator should be non-blocking too.

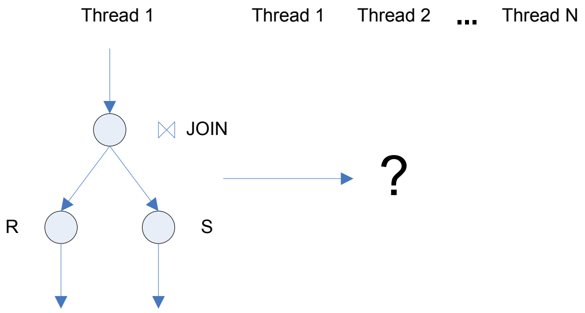


Picture 1: architecture

The QEP generation procedure also is a very unpretentious and is based entirely on heuristics, which resemble classical ones described [8]. The plan generation doesn't affect our benchmarks much because we use very specific loads (described in part 5) to assess the performance. Some details regarding implementation can be found in [9]. To conclude, we should mention that despite its simplicity our system is sufficient for use in our benchmarks to reach our goal.

4. Considered approaches

We considered the following approaches to incorporation of threading into join processing. Our initial algorithm was essentially nested loop join algorithm [4] which was modified to cope with the networking issues. The main difference is the bulk processing technique which provides bufferization to lessen the communication burden (as opposed to per record processing). When the both sets of tuples are received, we sort one of them and use the second as the probing one. Also, we employ in-operation caching to lessen the network stress further. This caching works as follows: if one of the participating relations is small enough to fit in memory, it is put into inner operation cache. This baseline method is working inside a single thread (all joins and cross-products are working inside one thread), Pic. 2.

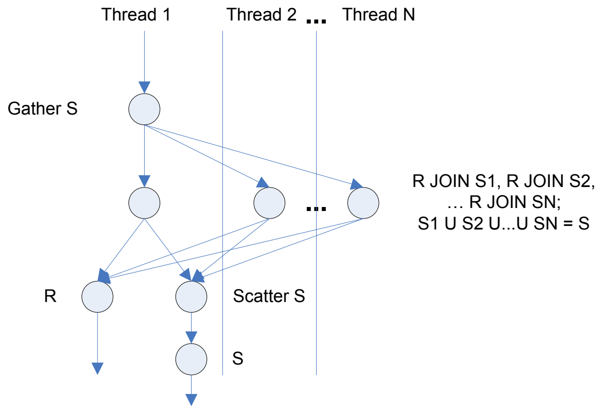


Picture 2: initial join design

To introduce parallelism we consider the technique which employs a number of threads for data processing. The core idea is to partition data supplied by one of the child nodes, while leaving the other one intact, then use different threads to process it (shown on Pic. 3).

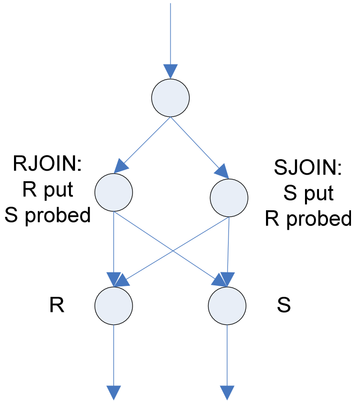
Also, we consider a hash join operator. In this case, the processing of a join operation is separated into two separate phases: a build phase and a probe phase. During the build phase, a table which was chosen as a build one is used in building a hash-table. Then, the next phase is essentially a traversal of a probe table to locate the match. The evident drawback of this technique is the temporary "freeze" of the system. This is caused by the build stage, e.g. the dependent operators should wait until the build phase is finished. This drawback is a critical issue, because it breaks [2] the pipelining execution, or this join operator becomes a blocking operator. If we are dealing with the QEP of a sufficient depth the blocking operators can become the source of a major slowdown in processing of the whole query. Also, this blocking especially adversely affects inter-operation parallelism (threads will wait for tuples to arrive). Thus, blocking operations should be taken into consideration.

To parallelize this join we employ the similar technique: we partition the build relation into a few fragments and each fragment is processed inside its dedicated thread.



Picture 3: join parallelization

To overcome the blocking nature of the previous approach we would consider a symmetric hash-join operator [2]. The core idea is to have two join nodes, which have different building relations. The processing happens as follows: given a tuple from relation R we probe it against build relation of S join operator and at the same time, we use this tuple in building relation of R join operator. Likewise, given a tuple from relation S we use it to probe it in R and build it in S operators. This join is correct due to the fact that if a tuple arrives too early (e.g. no relevant tuples from other relation had been built) it would be probed later, because it is built in other join operator. This type of hash join operator is a non-blocking operator. However, this advantage comes at a price – additional memory and time constraints (twice as much as a simple hash join) and more sophisticated programming techniques are required.



Picture 4: symmetric hash join operator design

To implement data partitioning in our system we introduced two new operators (much alike exchange operator in volcano system), for distribution and collection of the results. These operators are inserted in QEP around join operator which is going to be parallelized. We call helper threads the threads which hold the additional join nodes which will appear as the result of our parallelization.

5. Experiments

5.1 Parameters used in experiments

We implemented the proposed approaches and examined them using our system on a variety of workloads. First we characterize the data used in our experiments:

1) Size of the tables are 150Mb (first type of workload)
2) Each table contained three attributes: primary key, integer attribute used for joining and predicate which defines selectivity. We used only 4 byte integer data in our experiments and it is uniformly distributed.

The processing details:

1) Number of helper threads per processing node (QEP one): 1, 2, 5, 10
2) Methods (node types) of join:
    a. Hash join

    b. Block nested loop (sort)
    c. Block multithreaded nested loop (sort)
    d. Block multithreaded nested loop (hash)
    e. Symmetric hash join
3) Operation cache (also, size of bulk which is transferred from one QEP node to another): 2, 5, 10 MB
4) Join operators use results of a sequential scan over table (no index is used)

Query type details:

1) Query to evaluate intra-operation parallelism: a single join of two tables
2) Query to evaluate inter-operation parallelism: a QEP which contains 2 joins and 3 tables

5.2 Hardware and software

The following setup was used: Intel(R) Pentium(R) D CPU 3.00GHz, RAM 3Gb, which run GNU/Linux 2.6.35.10 x86, and we used gcc 4.5.2.

NPTL threads are used for multithreading, threads communicate via chunk of shared memory. Notifications are implemented using POSIX pipes along with select core. For hashes we used c++0x unordered map STL container.

5.3 Experiments performed

We had evaluated our algorithms on two distinct groups of workloads, namely workloads which evaluate intra-operator parallelism and inter-operator parallelism. The first ones evaluate our algorithms using QEP which contains only one join operation, which is fueled by two scan operations. The Table 1 contains summarized results over all runs in this group of experiments (here we measured execution time in seconds). Each experiment was repeated 100 times and the average value was used. Detailed results are presented in diagrams 1-3, where each histogram shows results for a fixed cache size.

In this experiment we varied the following parameters: operation cache and number of helper threads. The last six methods describe parallelized join methods (for threads amount of 2, 5, 10 respectively) of two distinct methods: sort and hash. Sort method is a plain block nested loop, where one relation's block is sorted before probing by another.

In multithreaded tests we observed several modes. This phenomenon can be explained by system thread scheduler fluctuations. The distributions of measured values for block nested loop are presented in histograms 1-2, bin size is determined by Sturges' formula in each case. The Table 1 presents main mode value.

Method/Cache size	2Mb	5Mb	10Mb
Block NL Hash	50.69	50.77	56.60
Block NL Sort	57.50	56.56	57.45
Symmetric HJ	67.60	66.98	68.16
Block NL Hash MT 2	47.96	45.28	44.78
Block NL Sort MT 2	48.36	45.74	45.43
Block NL Hash MT 5	46.75	45.56	44.32
Block NL Sort MT 5	47.31	45.05	47.64
Block NL Hash MT 10	45.34	45.19	45.94
Block NL Sort MT 10	51.62	44.23	46.47

Table 1: Intra-operation parallelism

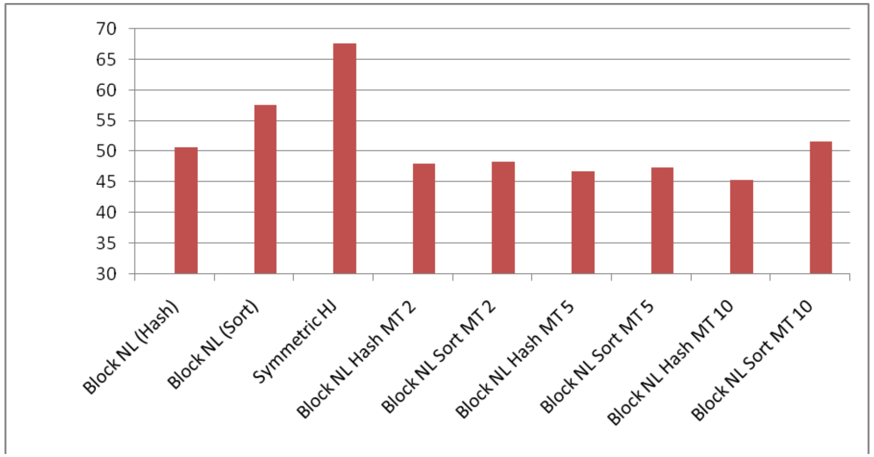


Diagram 1: 2 Mb cache

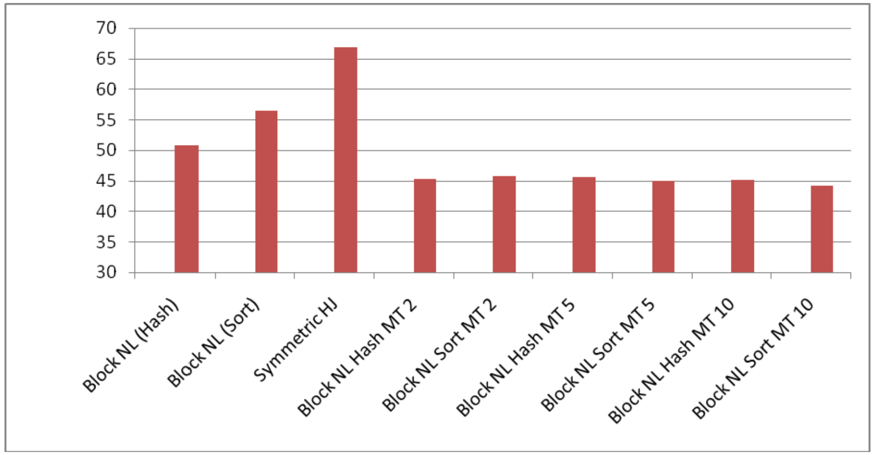


Diagram 2: 5 Mb cache

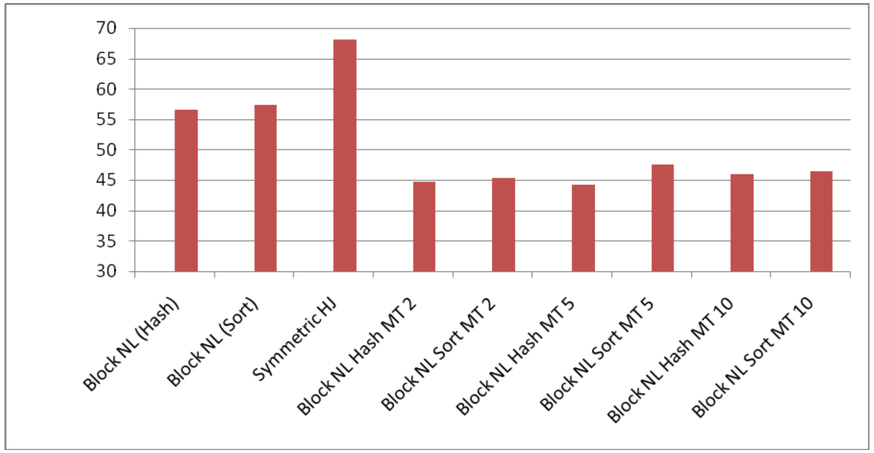
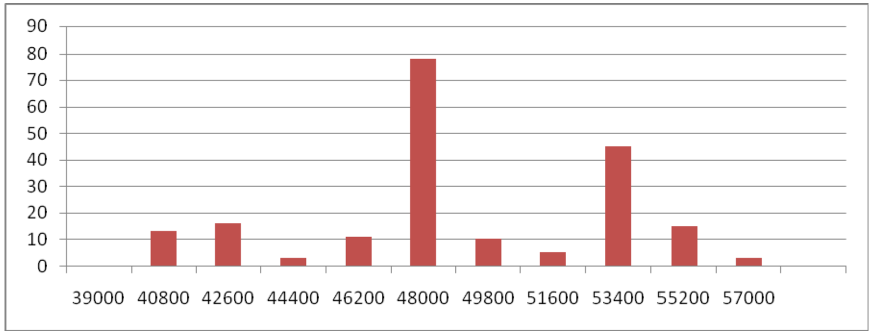
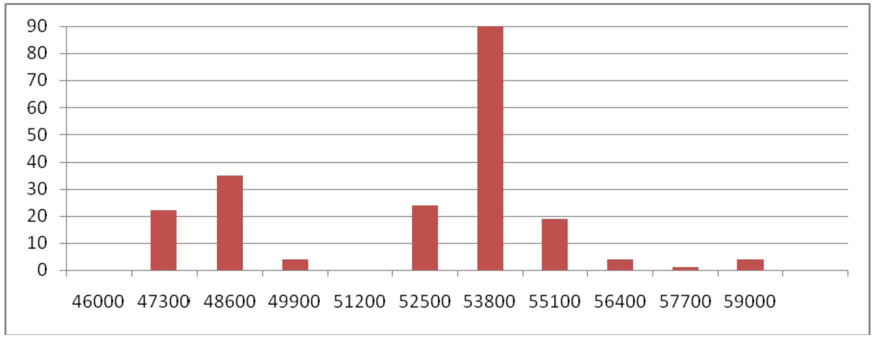


Diagram 3: 10 Mb cache



Histogram 1: 2 threads



Histogram 2: 10 threads

The Block NL Hash MT version of the operator is constructed in such way, that hash relation is always hashed and fully resides in the memory. So, it can be considered a true hash join. We can draw the following conclusions:

1) Parallelization does help to speed-up join processing on modern hardware, under several conditions. Our initial experiments were conducted with bulk size of 1Mb and parallelized joins had shown worse results that single-threaded ones.
2) Increasing bulk size does increase performance, up to some point. Our assumption is the following: there is a tradeoff between a number of data transfers and degree of thread utilization, e.g. increasing amount of work one may ensure that communication does not happens too often, but we have to communicate frequently enough in order to have a benefits of parallelism and pipelining.
3) Increasing number of threads further does not improve performance on dual core CPU.
4) Parallelization leads to multi-mode distribution of measured execution time. This behavior may affect estimation of parallel plan execution time.
5) Examining single-threaded join methods (first three mentioned operators) had shown us the following:
    a. Hash join better than nested loop

b. Symmetric hash join is actually worse than nested loop

6) Other results (not shown here) state the following: hash-based parallel algorithms a bit worse than sort-based ones on workload which converges to cross-product and vice versa (join which has both tables filtered on the same field and value). Our hypothesis is the following: this behavior could be attributed to our implementation of sort-based algorithm (we implemented a quicksort algorithm) which is tightly coupled with pointers for efficiency and to a large amount of collisions which arise considering this workload.

Another group of experiments we considered was the group which tested the QEP containing 2 join nodes. The goal was to run each join in separate threads. We restrict ourselves to having both join operators of the same type. The results of these experiments were the following:

1) There are not much intermediate results of the first join processing to load second one in such way to see different results. So is unfeasible to use the original dataset in order to evaluate these approaches. To cope with this problem we raised scan selectivity 5 times.
2) Under the new conditions we got the following results:
   a. Difference between non-threaded hash join and symmetric hash join is about 5% (hash join is better). Overall performance of QEP had increased over the same case of the first group (8% loss). We presume that this happens due to enabling of pipelining.
   b. Threaded versions worse 2.5 times compared to single-threaded hash join on the smallest buffer size (2MB). Unfortunately, now, it is unclear whether our programming technique was sufficient to implement this threading correctly or it is the hardware processing specifics guarantee us these bad results (different operators cause constant cache resets). We can also say for sure, that it is not thread scheduler problem, because adding more threads does not have any effect.

   c. Bulk size has a huge impact on performance in such case also. It is becoming even more evident here. Increasing bulk size increases performance of threaded joins almost 2 times (single-threaded ones get approximately 10%), but still threaded joins lose to single threaded ones.

6. Conclusions and future work

We presented results of evaluation of a set of join algorithms. The results imply that thread usage is beneficial for join processing and can give up to 50% increase in performance on simple queries. However, these approaches require extra programming effort, extensive parameter tuning and heavily rely on thread scheduler. Symmetric hash join has a promise for deep QEPs over threaded joins on such hardware.

This is a relatively small piece of work for this vast and evolved topic. A lot of aspects require our attention and they hadn’t been touched at all in this paper. The future work will include parallelization of symmetric hash join algorithm, deeper join trees for more thorough analysis of threading effects to pipelining, proper evaluation of scale-up, and possibly evaluation on more productive equipment.

References

[1] M. Tamer Özsu and Patrick Valduriez. 1999. Principles of Distributed Database Systems (2nd Ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[2] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive query processing. Found. Trends databases 1, 1 (January 2007), 1-140.

[3] Donald Kossmann. 2000. The state of the art in distributed query processing. ACM Comput. Surv. 32, 4 (December 2000), 422-469.

[4] Goetz Graefe. 1993. Query evaluation techniques for large databases. ACM Comput. Surv. 25, 2 (June 1993), 73-169.

[5] http://msdn.microsoft.com/en-us/library/ms188611.aspx Degree of parallelism SQL Server 2008 R2, access date 14.06.2011

[6] Oracle (R) Database Datawarehousing Guide 10g Release 2 (10.2)

[7] Yun Wang. 1995. DB2 Query Parallelism: Staging and Implementation. In Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95), Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 686-691.

[8] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access path selection in a relational database management system. In Proceedings of the 1979 ACM SIGMOD international conference on Management of data (SIGMOD '79). ACM, New York, NY, USA, 23-34.

[9] Kirill Smirnov and George Chernishev. 2011. Networking and multithreading architectural aspects of distributed dbms (in Russian), Software and Systems 1(93) (March 2011), 164-168