Karlsruhe Reports in Informatics 2011,5
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Parallel SQL Query Auto-Tuning on Multicore

Victor Pankratius, Martin Heneka

2011

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Parallel SQL Query Auto-Tuning on Multicore

Victor Pankratius
Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
victor.pankratius@kit.edu
www.victorpankratius.com

Martin Heneka
Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
martin.heneka@student.kit.edu

ABSTRACT

Multicore processors with several processors on a chip are standard, so applications need to be parallel in order to exploit the performance potential. Relational database systems are important applications that can exploit new opportunities for parallelism within queries. Intra-query parallelism offers additional performance potential that could not be exploited easily on earlier hardware. Addressing this important issue, this paper focuses on a difficult scenario for performance improvement: the parallelization of joins in I/O intensive multi-join queries. Our approach has the significant advantage that is does not require a rewrite of existing query optimizers from scratch. We boost query speed on multicore systems using query execution plans that are generated by sequential optimizers. This is the first paper to (1) auto-tune parallel query performance by adjusting the structure of multi-threaded pipelines that are superimposed over sequential query plans; (2) employ double-pipelined hash joins that are multithreaded to boost performance; (3) let an auto-tuner decide how to adapt parallelism to the hardware environment by exchanging hash join algorithms in the query plan; (4) present a demonstration and working strategies for multithreaded query auto-tuning on shared-memory multicore systems. Our evaluations show that queries can execute up to a factor of 3 faster on a quad-core machine, and that queries from the industry TPC-H benchmark can execute up to 47% faster compared to sequential execution. The results are remarkable considering the I/O bound context. Using PostgreSQL's code as an example, we also discuss the software engineering issues for the adaptation of real-world database systems.

1. INTRODUCTION

Stagnating processor clock rates pushed multicore processors with several cores on the same chip into the mainstream. Consequently, desktop and server applications that need to increase performance now have to be parallel. Database Management Systems (DBMS) are important and widely used applications – almost any Web application or business application is built on top of a database. For DBMS, multicore processors open the door for additional parallelization potential within queries (so-called intra-query parallelism). This type of parallelism could not be exploited easily in the past due to limited capabilities of earlier hardware. Most parallel query optimizations in clusters and multiprocessors machines thus concentrated on coarse-grain parallelism (inter-query parallelism). However, modern multicore systems have distinguishing characteristics, such as multiple processors, shared caches, shared busses – all on the same chip. These subtle differences move the break-even point of overhead to a more favorable location. Query optimization strategies thus need to be reconsidered.

Unfortunately, database systems are complex applications that are not trivial to modify in order to exploit multicore parallelism. The algorithms implemented in query optimizers typically require years or even decades of research and experimentation. Rewriting everything from scratch requires a large investment and long-term commitment in research and product development. But what can we do in the short-run to improve query performance on multicore platforms?

This paper presents a novel query parallelization approach focusing on a difficult scenario for performance improvement: the parallelization of joins and relational multi-join queries. Joins belong to the most important and – in terms of performance – most expensive operations in relational query processing, due to I/O [5, 17]. We are motivated to tackle this problem because even small performance gains in this area will affect a large number of queries and become immediately visible in a multitude of applications for many users. Moreover, our approach uses execution plans that are generated by sequential optimizers, thus saving substantial investments in building optimizers from scratch. As shown later, our approach can be implemented right away in most of the existing relational DBMS and scales on various multicore platforms. We do not exclude that it can be used together with other optimization techniques.

In particular, the paper makes the following contributions. To our knowledge, we are the first to tackle the following issues on multicore systems and present data from experiments with a real implementation. We introduce inter-operator parallelism with non-linear multi-threaded pipelines superimposed over a sequential query plan. We employ an auto-tuner to empirically tune query performance on a target platform by adjusting the pipeline structure. The auto-tuner has a feedback-directed mechanism to iteratively adapt its query plan transformations to the hardware envi-

Technical Report 2011-5
Karlsruhe Institute of Technology, Germany
Multicore Software Engineering Young Investigator Group
February 8, 2011

ronment that executes a query. We introduce intra-operator parallelism by implementing multithreaded hash join algorithms; where appropriate, our auto-tuner can automatically exchange join algorithms in the query plan to improve performance in subsequent query executions. The paper demonstrates on several multicore platforms that auto-tuning heuristics work; they improve query performance of all benchmarked queries on all benchmarked platforms.

The paper presents to the parallel computing community that query processing is an important real-world application scenario for pipeline parallelism, patterns, and auto-tuning. Even though auto-tuning has been done in the past in other ways and mostly for numerical applications [7, 29, 13, 20], we want to exemplify that database query optimization is a fruitful new ground to expand the core concepts of auto-tuning. The way we employ auto-tuning is similar to offline tuning. We present the database community with a new opportunity of applying multithreading in query optimization, which on multicore systems represents an additional degree of freedom in the spectrum of query parallelization approaches. This is beyond earlier approaches [27, 28, 6, 22, 3, 2] that largely neglected intra-query parallelism due to lack of capable hardware.

The paper is organized as follows. Section 2 presents a background on join algorithms and discusses variants that are suitable for multithreading and pipelined processing. Section 3 introduces the key concepts of our multithreaded pipeline parallelization of relational queries. Section 4 describes our automatic query performance tuning approach. Section 5 discusses the evaluation environment and the performance results on four multicore platforms. Section 6 elaborates on directions for future extensions. Section 7 details the software engineering issues encountered in PostgreSQL. Section 8 contrasts our concept with related work. Section 9 provides a conclusion.

2. BACKGROUND ON JOIN ALGORITHMS

The join operation $R \bowtie_C S$ defined by Codd [5] in the relational data model combines tuples from separate relations R, S on a specified join condition C. So for example if R is a relation containing tuples representing social security numbers and person names, S a relation with tuples representing social security numbers and salaries, and C a condition matching social security numbers in both relations, the result would be a relation containing tuples of social security numbers, names, and salaries.

The naïve *nested loop join* implementation uses two nested loops over a designated "inner" and "outer" relation; a new relation is iteratively produced containing each tuple of the inner relation matching the join condition with a tuple of the outer relation. This algorithm can be inefficient, so other join algorithms have been developed [17].

Hash join algorithms (e.g., simple hash join [10], hybrid [16], GRACE [15]) work for so-called equijoins in which C contains just equality comparisons. These algorithms are used in commercial and open-source DBMS – as for example, in PostgreSQL [19]. Earlier work [8, 9] has pointed to these algorithms to potentially better exploit intra-query parallelism, but did not shed enough light on multicore performance with more than two cores and industry benchmark queries so far.

Hash join algorithms work in two phases: (1) build a table with hash values of all join attributes for one relation (i.e., the "inner" relation, usually the smaller one); (2) read tuples from the "outer" relation and compare the hash values of join attributes with the hash table of the inner relation; if hash values match, two tuples are joined if their join attributes also match. Joined tuples are passed on to the next operator in the query plan (see Figure 1), so in addition to one input stream of tuples from each join relation, there is an output stream of joined tuples. Obviously, the working principle is asymmetric since one of the two relations is handled in a different way. This has not been a problem so far on sequential processors, as tuples were passed one by one from one operator to another. However, asymmetry is a disadvantage for pipelined processing. The blocking behavior of the build phase could make the entire operator pipeline stall until some hash join operation reaches its second phase; only then will it feed joined tuples into the following pipeline stages.

As an improvement, the *double pipelined hash join* [12, 30] is symmetric and does not distinguish between a build phase and a probe phase. Both input relations have an own hash table (see Figure 1). Our approach to speedup up query processing uses one thread for each table as a means for fine-grain parallelization, so each double-pipelined hash join needs 2 threads. Tuples from one relation are hashed into a local table and probed with the other relation's hash table. Thus, a continuous stream of joined tuples is produced, and query optimizers don't need to designate an "inner" and "outer" relation.

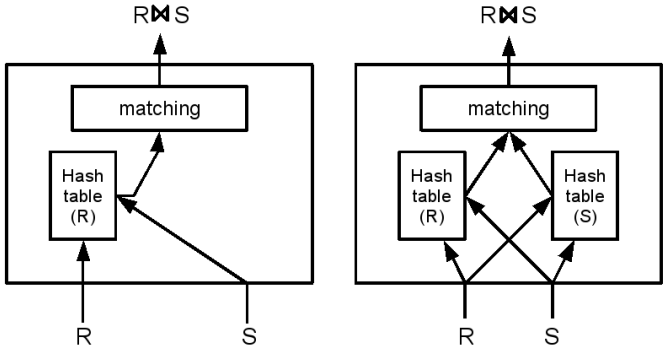


Figure 1: Dataflow comparison between asymmetric and symmetric hash joins [30].

3. PARALLELIZING QUERIES WITH NON-LINEAR MULTITHREADED PIPELINES

A DBMS transforms an SQL query into a sequential query execution plan, which is the initial data structure we use for parallelization. Figure 2 (a) shows a real query from the TPC-H benchmark and Figure 2 (b) depicts an excerpt of the sequential query plan, as generated by PostgreSQL.

The query plan can be represented as a tree with relational operators as nodes and relations as leaves. Operators are typically implemented as iterators [10] that establish producer–consumer relationships between nodes. Operators take tuples from child nodes, process them, and pass the result on to their parent node. The mainstream approach is to pass tuples sequentially, one by one, to reduce temporary storage of large intermediate results. PostgreSQL and most other systems use a demand-pull approach where the root node starts demanding tuples from its children. Looking at the whole tree, this technique establishes a non-linear pipeline structure over the whole query plan in which tuples flow up to the root, using buffers between operators [4]. The root node may perform additional transformations o the received tuples. Pipelined query execution can thus be treated as a dataflow problem [30]. However, the pipeline is processed sequentially. Our approach introduces multithreading at exactly this point, processing several operators in parallel on multicore systems. As will be shown later, the division of parallel work among threads is not trivial, because is cannot be done statically; a dynamic approach is needed that is aware of the target hardware characteristics.

The query operator pipeline can be parallelized as shown in Figure 2 (c). One or more connected nodes (i.e., query operators) can be assigned to a pipeline stage that can be processed by an own thread; thus, several stages can potentially be processed in parallel on several cores. To achieve speedups, it is imperative to avoid pipeline blocking – this may be done by ensuring that and adequate pipeline structure and suitable join algorithms are used. Potentially, there exist many pipeline structure configurations for a given query plan. We employ an auto-tuner to find one that leads to the best performance on a given multicore platform.

In general, query optimization is a difficult area because optimizations often work just for certain types of queries; this is because queries can differ in many respects. Our proposed parallelization technique is suitable for longer and more complex queries rather than short queries with short pipelines. We assume a transactional environment where a query can be executed more than one time (e.g., due to requests from different users) and where materialization would not pay off. In addition, queries with a more balanced or "bushier" query plan (produced by the sequential DBMS optimizer) tend to exhibit more parallelization potential. An important problem is that some join operation might become a bottleneck and slow down the entire pipeline, but our experiments show that it is possible to use stage aggregation to compensate such negative effects. Our experiments have also shown that the best configurations may depend on the specific query and multicore platform, and that a static prediction how to employ join algorithms and associate operators to pipeline stages is not promising. We therefore developed an iterative auto-tuning approach, as described next.

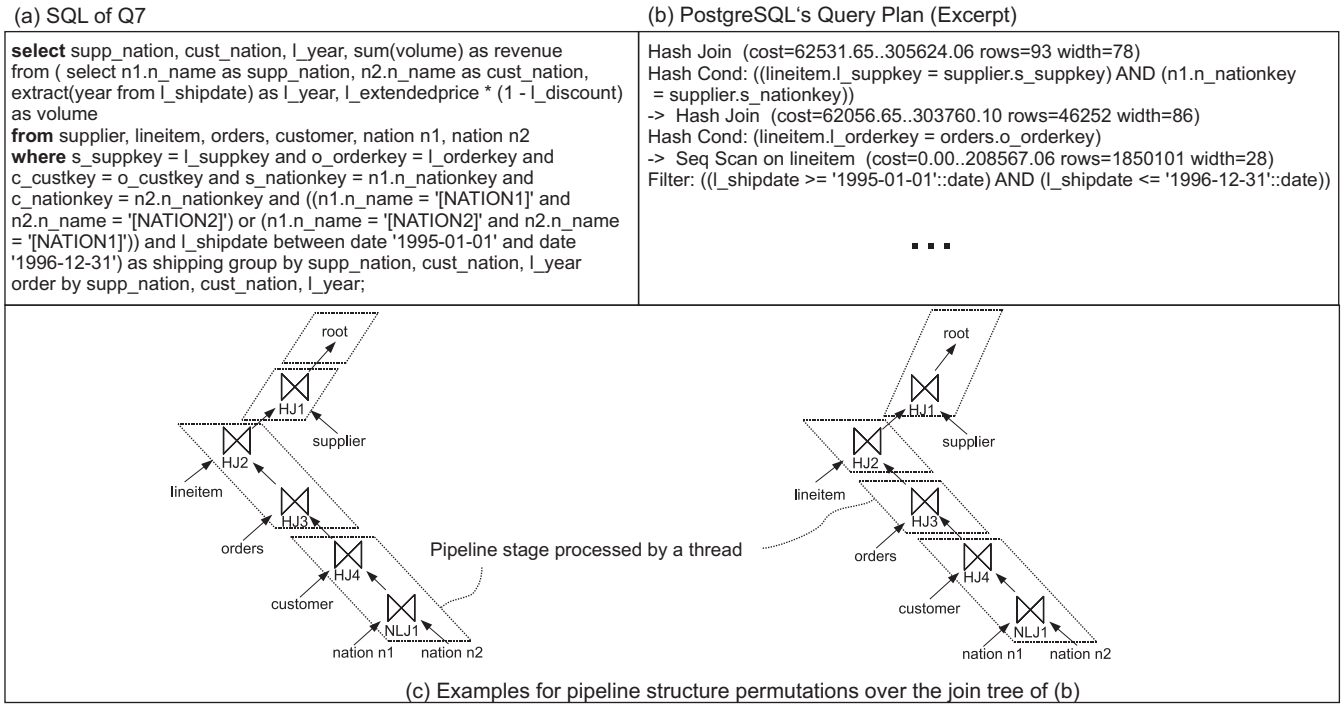


Figure 2: Example for TPC-H query 7: (a) SQL code; sequential query plan generated by PostgreSQL; (c) example stage configurations for a multithreaded pipeline.

4. AUTO-TUNING QUERY PARALLELISM

Our auto-tuning approach consists of several parts. We start with a discussion of findings from pilot studies that influenced the decisions on key aspects of the tuning process. We present patterns and transformations on query plans and use a microbenchmark approach to quantify performance impacts. We then describe how to detect performance-relevant query plan patterns and detail the principles of the automated performance optimization process.

4.1 Pilot Studies

Before developing automation strategies, we first conducted several pilot studies to assess the feasibility and performance potential of our approach. In particular, we analyzed speedup potential of superimposing non-linear multithreaded pipelines over sequential query plans, as well as the applicability of multithreading within symmetric hash joins. The studies were conducted using queries from the industry TPC-H benchmark as well as a multitude of other queries.

The results indicated that our approach is feasible and worthwhile to implement in a full-fledged environment. The results have also shown that the structure of the pipeline had the most significant impact on parallel query performance, and that increasing the number of threads per stage had a negligible performance effect (so this parameter would not require auto-tuning). Further results from the pilot studies implied that if certain patterns are detected in the sequential query plan, they can be exploited to introduce associated transformations. This technique reduces the search space for the definition of the multithreaded pipeline structure. We identified six important patterns that lead to the most significant speedup gains; other patterns showed little improvements.

4.2 Patterns in the Query Execution Plan

We now describe performance-relevant patterns in the sequential query execution plan. Each pattern is associated with a query plan transformation that introduces parallelism. The query execution plan is assumed to be represented by a tree data structure in which the size of relations on the right side of a node is smaller than the size of relation on the left side of that node; this is a usual property of plans [10] generated by many existing sequential optimizers (e.g., PostgreSQL).

In total, we employ six patterns as sketched in Figure 3. The patterns identify locations where to insert a boundary for pipeline stage or where to replace an asymmetric hash join algorithm by our multithreaded symmetric hash join. We assume that for each node that has two leaves, there is a starting boundary for a pipeline stage that includes that node (but not the leaves).

- **Pattern 1**: Identify one asymmetric hash join node that has a right child that is also an asymmetric hash join node. Insert a boundary for a pipeline stage between the two joins. This transformation introduces parallelism by overlapping tuple computations in time.
- **Pattern 2**: Identify one asymmetric hash join A in a left subtree, so that following the path up to the root, there exists one other asymmetric hash join node B (select the first one encountered). Introduce a boundary for a pipeline stage between A and its direct parent. This transformation can speed up the creation of hash tables in the join further up in the tree.
- **Pattern 3**: Identify an asymmetric hash join A in a right subtree that has a direct parent B that is also an asymmetric hash join. In addition, B has a left child with a large input relation. Starting from B and following the path up to the root, there exists one additional asymmetric hash join C. Introduce a boundary for a pipeline stage between joins A and B. This transformation avoids creating a pipeline stage in a wrong place and thus avoids creating a bottleneck through which many tuples have to pass.
- **Pattern 4**: Identify an equi-join node in which both children are sub-trees (i.e., no leaves $\rightarrow$ no table scans, no index scans). Replace the join algorithm by a multithreaded symmetric hash join algorithm.
- **Pattern 5**: Identify an equi-join node whose left child is a sub-tree (i.e., no leaf) and whose right child is a leaf. Replace the join algorithm by a multithreaded symmetric hash join algorithm.
- **Pattern 6**: Identify an equi-join node with arbitrary left and right children (i.e., either sub-trees or leaves) for which the following condition holds: $relationsize_{right}/relationsize_{left} > 0.9$ (the value was determined in our exploratory experiments). Replace the join algorithm by a multithreaded symmetric hash join algorithm.

All patterns assume that the size of input relations to the entire query is not too small, so the additional overhead that comes with the proposed transformations can be compensated. This estimation can be deduced by the optimizer by static query analysis, but we also employ dynamic data to improve estimates.

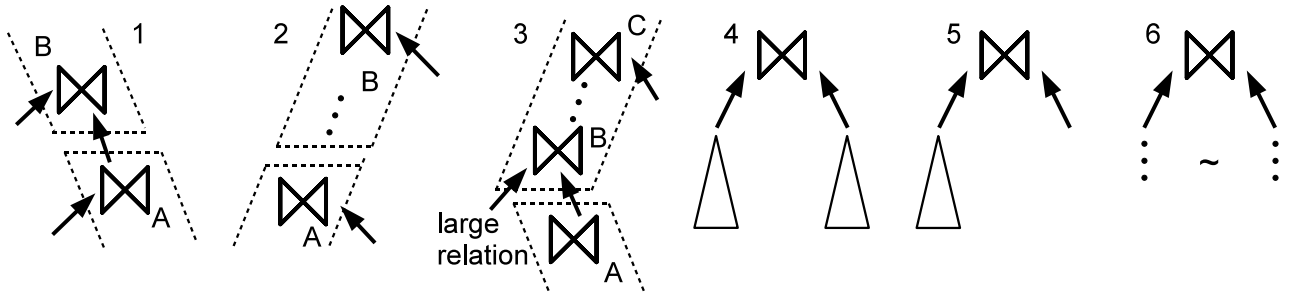


Figure 3: Patterns identified in the query execution plan.

4.3 Microbenchmarks and Pattern Ranks

We run an initial microbenchmark on every new hardware platform to estimate the performance impact of each pattern. This is done exactly once for each unknown platform.

The microbenchmark consists of synthetic queries that lead to query plans with the aforementioned patterns. For each query in which one of the six patterns is detected, the auto-tuner applies the defined transformations and measures the average speedup gained after running the parallel query several times. Our auto-tuner also performs warm-up measurements to avoid distortions.

A normalized rank metric $-1 \leq r \leq 1$ is calculated for each pattern based on the measured performance improvement in relation to sequential execution. This rank describes the relative performance impact of a pattern in comparison to all patterns; higher values of r indicate that higher speedups can be expected. Our experiments have also shown that the rank measure can be used to compare the impact of a pattern on different multicore architectures.

Pattern ranks define the order in which patterns are chosen and applied by the auto-tuner. After executing the microbenchmark, the resulting ranks remain constant throughout the auto-tuning process.

4.4 Pattern Detection

Patterns 1–5 are detected after one traversal of the query plan, using static information such as node type (e.g. hash join, table scan), node position (e.g., left sub-tree, relative to another node), the estimated total size of relations per subtree (to estimate if a node is well-balanced). In addition, statistics of the sizes of relations of sub-trees are collected from previous query executions and made available during pattern detection (e.g., to improve detection of pattern 6). These statistics also help to correct misestimations of the initial static analysis.

The efficiency and performance impact of each pattern varies from system to system. In addition, a combination of several patterns on a query plan can improve the results. We develop an automated approach to successively select and apply the most promising patterns, as described next.

4.5 The Performance Optimization Process

Automated performance optimization occurs in an iterated and feedback-directed fashion. The auto-tuner assumes that a query with the same structure can be executed more than once; this is a scenario that often occurs in Web server or business information systems that handle many similar requests from different users. The tuning process has two phases: Phase I performs coarse-grain tuning by iteratively detecting patterns and applying transformations; phase II starts with fine-tuning when no more transformations can be applied.

Phase I. In each iteration, the auto-tuner tries to find and apply a new pattern to the query plan. In the first iteration, the auto-tuner starts with the sequential query plan, and every iteration works on the query plan resulting from the preceding iteration. Patterns are successively applied in each iteration as long as there are applicable patterns that were not selected so far. The selection of the next pattern is determined by its rank. In cases where two or more patterns have the same rank, the auto-tuner randomly selects one of them with equal probability. A pattern is skipped if its transformation has already been applied in an earlier iteration. The auto-tuner logs in each iteration the speedup after the application of a new pattern transformation. If a pattern leads to a better speedup compared to the previous iteration, the query plan transformation is kept for the next iteration, otherwise the transformation is undone.

Phase II. If no other patterns can be applied, the auto-tuner tries using random transformations to further improve performance. A fine-tuning transformation can be either an insertion of a pipeline stage boundary at a random location or a buffer size change between two stages. Buffering tuples between stages can save synchronization overhead, but the optimum buffer size depends on the concrete situation. Thus, the auto-tuner randomly selects sizes $\in \{64, 128, 256, 512, 1024\}$ tuples per buffer (each one can be picked with equal probability).

The tuning process terminates when the speedup improvement is less than a pre-defined constant ϵ. In principle, the tuning process can be stopped after any iteration; it will then use the configuration that was considered best until that particular point. This property provides a high flexibility for many real-world applications.

We remark that an advantage of our approach being designed to work on query plans is that slight modifications of a query's SQL code (e.g., in selections, projections, or other filters) between two executions would still make it possible to apply performance-improving transformations based on information from previous executions, as long as the structure of the query plan remains similar.

5. EVALUATION ON MULTICORE

This Section presents the experimental setup, benchmark databases and queries, and the empirical performance results on several multicore platforms.

5.1 Environment

Our query auto-tuner is fully implemented in C++ in an environment oriented at PostgreSQL. We evaluated the parallelization of real-world SQL queries in a controlled environment using the sequential query plans as generated by PostgreSQL. We chose PostgreSQL because it is a well-documented open-source DBMS that is also employed in productive environments.

Our auto-tuner creates independent threads for pipeline stages using the Pthreads library. Threads are started during the initialization phase of the iterators associated with a tree node. Once threads are running, the associated pipeline stages start producing tuples. We read all data from hard disk, and all our measurements include I/O showing realistic results in a productive scenario. For testing purposes, our system also allows users to manually alter the initial query plan input to assess the impact of modifications on multicore performance.

Our experiments are carried out on following multicore platforms:

1. **4-Core Intel Machine**: Intel Core 2 Quad Q6600 processor, clocked at 2.4 GHz, equipped with 8 GB RAM, running Linux 2.6.32. The system has 256 KB L1 cache, and 8 MB L2 cache.

2. **4-Core AMD Machine**: AMD Phenom II X4 810 processor, clocked at 2.6 GHz, equipped with 6 GB

RAM, running Linux 2.6.34. The system has 512 KB L1 cache, 2 MB L2 cache and 4 MB L3 cache.

3. **8-Core Intel Machine**: Intel with 2x Quadcore Xeon E5320 processor, 1 thread per core, clocked at 1.86 GHz, equipped with 8 GB RAM, running Linux 2.6.32. The system has 512 KB L1 cache, and 8 MB L2 cache.

4. **8-Core Sun Machine**: SUN UltraSPARC T1 processor (Niagara 1), 8 cores, 4 threads per core, clocked at 1 GHz, equipped with 16 GB RAM, running Solaris 5.10. The processor has 192 KB L1 cache, and 3 MB L2 cache.

We use databases generated by the TPC-H benchmark revision 2.12.0 [24], with different sizes for the input data (determined by so-called "scale factors"). Experiments with scale factor SF1 have input relations with a total size of about 1 GB; for SF2, total input relation size is about 2 GB.

5.2 Benchmarks on Join Queries

We first conduct experiments using the largest table (*lineitem* has ∼750MB on scale factor 1). This type of query has important practical applications, e.g., for bill of materials explosion, computing transitive closures, and truck routes. We execute queries with $n \in \{1, 2, \ldots, 6\}$ self-joins on this table. Resulting query plans have n hash joins.

For each of these queries, we take the sequential plan as produced by PostgreSQL and create all possible rebalanced versions by hand (this is the reason why $n \leq 6$ in our setting). Each version is then parallelized with our approach. We measure the best and worst query run-time reduction (along with the best and worst speedup), in comparison to sequential execution. The results of these experiments are summarized in Table 1, after 10 tuning iterations.

Results show that the difference between the best and worst results for each experiment (shown in a table cell) is low. In addition, speedups are remarkable. For example, a speedup of three on a quad-core platform is good, given that I/O processing and hard disk reading are fully included in all measurements. Auto-tuning can exploit machine-specific properties and improve performance on the eight-core machine up to a factor of 3.9. However, query-inherent properties and machine-dependent properties make it difficult to improve performance even further.

5.3 Benchmark on TPC-H Queries

We analyze all TPC-H queries for which PostgreSQL produces a sequential query plan with more than one hash join. This approach selects six queries; Figures 2 and 4 show the SQL code of these queries.

Query Q2 has 3 hash joins. The query plan is an imbalanced, right-deep operator tree generated by PostgreSQL sequential optimizer. The size of relations on the left can differ significantly from the relation sizes on the right (e.g., on one join node, the left relation has 168 MB for SF1 and 337MB for SF2, while the results produced on the right have about 10 times less size). Inappropriate stage configurations may lead to pipeline stalls.

Query Q3 has 2 hash joins. The query plan is a right-deep operator tree, similar to Q2, but Q3 has additional selection operators on input relations that can be processed in parallel. Relation sizes in Q3 are larger than in Q2. These selections reduce the amount of data to be processed by joins. This slight difference can significantly affect the overhead and influence the sweet spot for parallelization.

Query Q7 has 4 hash joins. The query plan is a tree that is more balanced than that of Q2 and Q3. This is a case in which an addition of too many pipeline stages and threads might introduce unnecessary overheads and thus lead to slowdowns or just minor performance improvements.

Query Q8 has 7 hash joins. The query plan is a tree with 3 joins left-deep, 4 joins right-deep; the join operator at the inflection point processes a 750 MB relation in SF1 and 1.52 GB relation in SF2, so it can easily become a bottleneck. The large amount of data to be processed may quickly reach the memory wall on some machines. Moreover, the join at the inflection point of the operator tree may cause problems if pipeline stages are chosen inappropriately. Introducing a pipeline stage right before this join causes overhead that does not pay off, because many tuples have to be streamed through one buffer between stages. If this join is placed in a stage with other joins, tuples get joined with other relations in the same stage, thus reducing the number passed over to other stages.

Query Q9 has 4 hash joins. The query plan is a tree with 3 left-deep joins, but the remaining joins are balanced down in the tree. Thus, several hash tables can potentially be built up in parallel.

Query Q11 has 2 hash joins. The query plan is similar to query Q2, except that the temporary results and the final result are smaller.

5.4 Results

In summary, our auto-tuning parallelization works and shows remarkable results. The run-time reductions on 4-core platforms are shown in Figures 5(a) and 5(b) and for 8-core platforms in Figures 6(a) and 6(b). Each graph illustrates a particular query executed on a certain machine. The x-axis shows the number of iterations that the auto-tuner needs to achieve a certain performance improvement. As execution times vary from platform to platform, the y-axis shows the performance improvement (i.e, run-time reduction) in relation to the sequential execution time on that particular machine. The solid line shows the results with our auto-tuning approach. To ensure that the results are stable, the entire measurement process was repeated 10 times.

The graphs illustrate that the auto-tuner is able to im-

	Min/max run-time reduction in %, max speedup after 10 iters					
#Joins	1	2	3	4	5	6
4-core AMD	43/43, **1.8**	59/60, **2.5**	63/67, **3.0**	59/65, **2.8**	61/66, **2.9**	57/66, **2.9**
4-core Intel	43/43, **1.8**	58/58, **2.4**	65/67, **3.0**	63/65, **2.8**	61/66, **2.9**	60/66, **2.9**
8-core Intel	42/42, **1.7**	58/59, **2.4**	65/66, **3.0**	64/70, **3.4**	63/73, **3.6**	66/74, **3.9**

Table 1: Join Query Performance on Largest TPC-H Table.

Q2

```
select s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address, s_phone, s_comment
from part, supplier, partsupp, nation, region
where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_size = [SIZE] and p_type like '%[TYPE]' and s_nationkey = n_nationkey and
n_regionkey = r_regionkey and r_name = '[REGION]' and ps_supplycost = ( select min(ps_supplycost) from partsupp, supplier, nation, region where
p_partkey = ps_partkey and s_suppkey = ps_suppkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = '[REGION]' )
order by s_acctbal desc, n_name, s_name, p_partkey;
```

Q3

```
select l_orderkey, sum(l_extendedprice*(1-l_discount)) as revenue, o_orderdate, o_shippriority
from customer, orders, lineitem
where c_mktsegment = '[SEGMENT]' and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < date '[DATE]' and l_shipdate >
date '[DATE]' group by l_orderkey, o_orderdate,
o_shippriority order by revenue desc, o_orderdate;
```

Q8

```
select o_year, sum(case when nation = '[NATION]' then volume else 0 end) / sum(volume) as mkt_share
from ( select extract(year from o_orderdate) as o_year, l_extendedprice * (1-l_discount) as volume,n2.n_name as nation
from part, supplier, lineitem, orders, customer, nation n1, nation n2, region
where p_partkey = l_partkey and s_suppkey = l_suppkey and l_orderkey = o_orderkey and o_custkey = c_custkey and c_nationkey =
n1.n_nationkey and n1.n_regionkey = r_regionkey and r_name = '[REGION]'
and s_nationkey = n2.n_nationkey and o_orderdate between date '1995-01-01' and date '1996-12-31' and p_type = '[TYPE]' ) as all_nations
group by o_year order by o_year;
```

Q9

```
select nation, o_year, sum(amount) as sum_profit from (select n_name as nation, extract(year from o_orderdate) as o_year,
 l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
from part, supplier, lineitem, partsupp, orders, nation
where s_suppkey = l_suppkey and ps_suppkey = l_suppkey and ps_partkey = l_partkey and p_partkey = l_partkey and o_orderkey = l_orderkey
and s_nationkey = n_nationkey and p_name like '%[COLOR]%' ) as profit group by nation, o_year order by nation, o_year desc;
```

Q11

```
select ps_partkey, sum(ps_supplycost * ps_availqty) as value from partsupp, supplier, nation
where ps_suppkey = s_suppkey and s_nationkey = n_nationkey and n_name = '[NATION]'  group by ps_partkey having sum(ps_supplycost *
ps_availqty) > (
select sum(ps_supplycost * ps_availqty) * [FRACTION]
from partsupp, supplier, nation
where ps_suppkey = s_suppkey and s_nationkey = n_nationkey and n_name = '[NATION]'
) order by value desc;
```

Figure 4: TPC-H Queries with multiple hash joins, benchmarked in addition to Q7 in Fig. 2

prove the performance of every query on every platform. The exact quantities of performance improvements vary for each query; this is because hardware differs and the auto-tuner adapts its strategy on which patterns to apply to improve performance. It is worth pointing out that the auto-tuner requires just about 10 iterations for any query to converge.

The charts also illustrate that some queries have more potential for pipeline parallelization, due to their structure. But even for short queries (such as Q2) with less parallelization potential, the performance has been improved by several percent. The approach works well especially for more complex queries; for example Q9 executes on the AMD platform 47% faster than sequential.

Compared to the Niagara platform, the performance gains on the two Intel machines and the AMD machine are higher. Several factors can provide an explanation. The Niagara has a lower clock frequency than the other machines. Moreover, Niagara's hardware is better optimized for I/O. Nevertheless, the auto-tuner is still able to improve performance, though with lower deltas.

Queries Q7 and Q8 already show performance improvements with the current approach, however, they can be made faster with an additional technique (marked as "alt. plan" in the graphs). Our manual experimentation revealed that slightly re-balancing the plan generated by PostgreSQL made more patterns applicable, so these queries performed better. Future work will investigate approaches to extend our technique accordingly.

We remark that we also compared for each query on each platform if the best auto-tuned result is better than a completely random selection of pipeline stages and join algorithms. Results confirmed that auto-tuning is better (i.e., it leads to better performance in fewer iterations). So building query auto-tuners is worth the effort.

In summary, optimizing I/O intensive queries differs from computationally intensive applications – especially in what speedups are realistic to expect. This insight is important for the community to realize. Embarrassingly parallel numerical applications can generate many compute threads and typically achieve almost linear speedups. In our scenarios, we a re I/O bound; threads are useful to hide latency. We

typically do not reach peak computation performance but rather hit the memory wall.

6. OUTLOOK

This paper demonstrates that parallel query performance can be improved on multicore with various multithreading techniques. There are two opportunities for further extensions that we consider most promising.

One direction to investigate is how to introduce an intermediate step to re-balance the query plans to further improve parallel performance. This is not a trivial task, as tree balancing depends on several factors, such as relation sizes and target hardware. Our first experiments with alternative plans (Q7 and Q8 "alt plan" in Figures 5 and 6) imply that additional performance can be exploited this way when more pipeline threads run in parallel.

In this paper, we mainly investigated the potential of pipeline parallelism in the context of join processing. Our approach can be extended with other forms of intra-operator parallelism, such as data parallelism.

7. SOFTWARE ENGINEERING ISSUES IN DATABASE MANAGEMENT SYSTEMS.

Introducing multithreaded pipelined processing as proposed in this paper requires relational operators to be implemented in a thread-safe way. Unfortunately, this is not the case in many DBMS implementations. Reengineering fundamental code is required. Taking PostgreSQL as an example, changes for inter-operator parallelization have to be made in the so-called "backend" of PostgreSQL, which is not multithreaded. The code has to be improved to get rid of single-core assumptions (e.g., current stack overflow detection is not valid in a multithreaded context) and avoid possible race conditions in different layers (e.g., memory management, storage management). Shared data structures (e.g., "contexts" and "states" during query execution) have to be privatized or synchronized. The good news for PostgreSQL is that it has good code quality. Most shortcomings that stem from sequentiality assumptions are relatively easy to detect by code inspection. Most problems can be resolved using thread local storage, e.g., in the code for error logging and reporting. PostgreSQL's implementation of the sequential pipeline and the operator implementations are separated and well-documented. The pipeline parallelization affects only a few files in the so-called "executor" module implementing sequential pipeline code. Adding the double pipelined hash join algorithms can be done incrementally, reusing code from the existing implementation.

8. RELATED WORK

In the past, auto-tuning has mostly been applied to numerical problems, such as in [7, 29, 13, 20]; these approaches are domain-specific (e.g., for matrix multiply or FFT) and not directly applicable to auto-tuning database queries as done in this paper.

In the context of databases, self-tuning has been explored by [27, 28, 6, 22, 3, 2], however, the proposed strategies have different goals and are outside the context of modern multicore processors. To our knowledge, our approach is the first to use auto-tuned structures of multithreaded pipelines over query plans, use multithreaded symmetric hash joins, and automated join algorithm exchange throughout the optimization of a query plan. In addition, a hypothesis of our approach is to propose an incremental enhancement to existing sequential query optimizers that would not require a major rewrite of large parts of database system code. This contrasts other proposals that require major changes to column-oriented storage, such as [11, 18, 26]. Other approaches such as [23] require more radical rewrites of current DBMS implementations, which are still row-based. Of course, starting from scratch offers more potential for improvements, however, such endeavors are costly, risky, and take a long time; many evaluations need still to be done for multicore performance.

A special operator is used in [1] to partition the query tree, but the evaluation is mainly hypothetical and shows just a few selective on a two-processor machine. Prior work on hash join parallelization [9, 14, 11, 21] concentrates on the asymmetric hash join. However, this two-phase algorithm has disadvantages for parallel pipelines: the blocking behavior of the first phase introduces pipeline stalls and may lead to bad performance. This paper makes a novel contribution studying parallel query execution with symmetric hash join algorithms, which overcome the blocking problem, on current multicore hardware. Among sort-merge-based and hash-based join algorithms, the latter were found to be more suitable on multicore systems [14]. Parallel join algorithms running on graphics cards were implemented in CUDA in [11, 25], however, assuming column-oriented storage and manipulating only columns of floating point values. The work of [25] has shown that join processing on GPUs is disadvantageous when queries operating on large data cause heavy memory transfers.

Other approaches such as [23, 31] require more radical rewrites of current DBMS implementations, which are still row-based; of course, starting from scratch offers more potential for improvements, but such endeavors are costly, risky, and take a long time. Until we get there, our technique can be used to achieve first scalability on standard multicore hardware.

9. CONCLUSION

Database systems are key applications that can take advantage of the wide availability of multicore computers. Multithreading is now an additional degree of freedom that query optimizers should consider in addition to other query optimizations. This paper presents a new approach for query performance improvement that does not require a major rewrite of sequential query optimizers from scratch. Using sequential query plans, auto-tuning shows great potential to introduce adaptive parallelism and exploit specific potentials on each hardware platform. In particular, our experimental evaluations on real hardware show that such adaptations pay off: the performance of all benchmarked queries was improved on all our platforms.

Acknowledgements. We thank the Excellence Initiative and the Landesstiftung Baden-Wuerttemberg for their financial support. We also thank Peter Lockemann for the fruitful discussions.

10. REFERENCES

[1] R. Acker et al. Parallel query processing in databases on multicore architectures. In *Proc. ICA3PP*, pages 2–13. Springer, 2008.

[2] S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 3–14. VLDB Endowment, 2007.

[3] S. Chaudhuri and G. Weikum. Foundations of automated database tuning. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1265–1265. VLDB Endowment, 2006.

[4] J. Cieslewicz and K. A. Ross. Database optimizations for modern hardware. *Proc. IEEE*, 96(5):863–878, 2008.

[5] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[6] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic sql tuning in oracle 10g. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1098–1109. VLDB Endowment, 2004.

[7] M. Frigo and S. Johnson. Fftw: an adaptive software architecture for the fft. In *Proc. IEEE ICASSP'98*, volume 3, pages 1381–1384, 1998.

[8] P. Garcia and H. F. Korth. Database hash-join algorithms on multithreaded computer architectures. In *Proc. ACM CF*, pages 241–252, 2006.

[9] P. Garcia and H. F. Korth. Pipelined hash-join on multithreaded architectures. In *DaMoN*, 2007.

[10] H. Garcia-Molina et al. *Database systems*. Prentice Hall, 2002.

[11] B. He et al. Relational joins on graphics processors. In *Proc. ACM SIGMOD*, pages 511–524, 2008.

[12] Z. G. Ives et al. An adaptive query execution system for data integration. *ACM SIGMOD Rec.*, 28(2):299–310, 1999.

[13] T. Katagiri, K. Kise, H. Honda, and T. Yuba. Fiber: A generalized framework for auto-tuning software. In *Proc. ISHPC*, 2003.

[14] C. Kim et al. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proc. PVLDB*, 2(2):1378–1389, 2009.

[15] M. Kitsuregawa et al. Application of hash to database machine and its architecture. *New Generation Computing*, 1(1), 1983.

[16] L. Liu et al. US patent 6263331 hybrid hash join process, July 2001.

[17] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Comput. Surv.*, 24(1):63–113, 1992.

[18] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proc. ACM SIGMOD*, pages 1–2, 2009.

[19] PostgreSQL Global Development Group. *PostgreSQL 8.4.1 Documentation*, 2009.

[20] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. Spiral: code generation for dsp transforms. *Proceedings of the IEEE*, 93(2), 2005.

[21] L. Rashid et al. Exploiting multithreaded architectures to improve the hash join operation. In *Proc. ACM MEDEA*, pages 46–53, 2008.

[22] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. Colt: continuous on-line tuning. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 793–795, New York, NY, USA, 2006. ACM.

[23] M. Stonebraker et al. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[24] Transaction Processing Performance Council (TPC). *TPC BENCHMARK H Standard Specification 2.12.0*, 2010.

[25] K. Tsakalozos et al. Using the graphics processor unit to realize data streaming operations. In *Proc. MDS*, pages 1–6, 2009.

[26] P. Vaidya and J. J. Lee. Characterization of tpc-h queries for a column-oriented database on a dual-core amd athlon processor. In *Proc. ACM CIKM*, 2008.

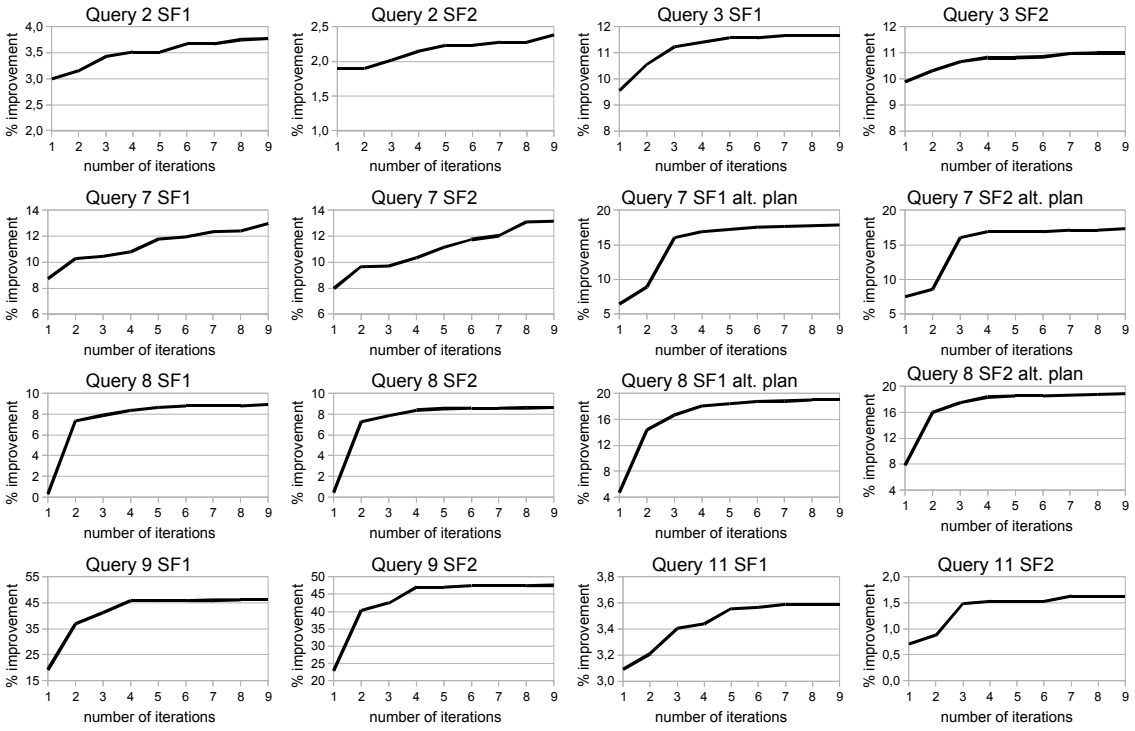
[27] G. Weikum, C. Hasse, A. Mönkeberg, and P. Zabback. The comfort automatic tuning project. *Information Systems*, 19(5):381–432, 1994.

[28] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 20–31. VLDB Endowment, 2002.

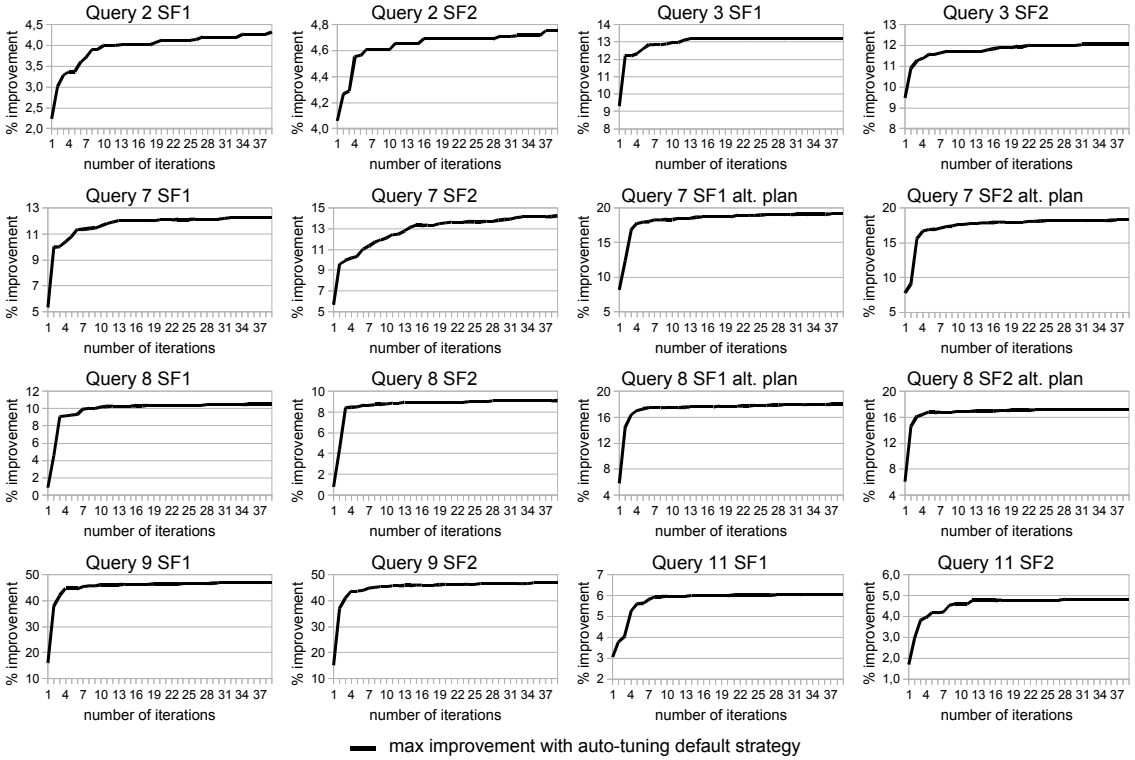
[29] C. R. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, January 2001.

[30] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distr. and Par. Databases*, 1(1):103–128, 1993.

[31] M. Zukowski. Parallel query execution in Monet on SMP machines. Master's thesis, Vrije Universiteit Amsterdam, Warsaw University, 2004.

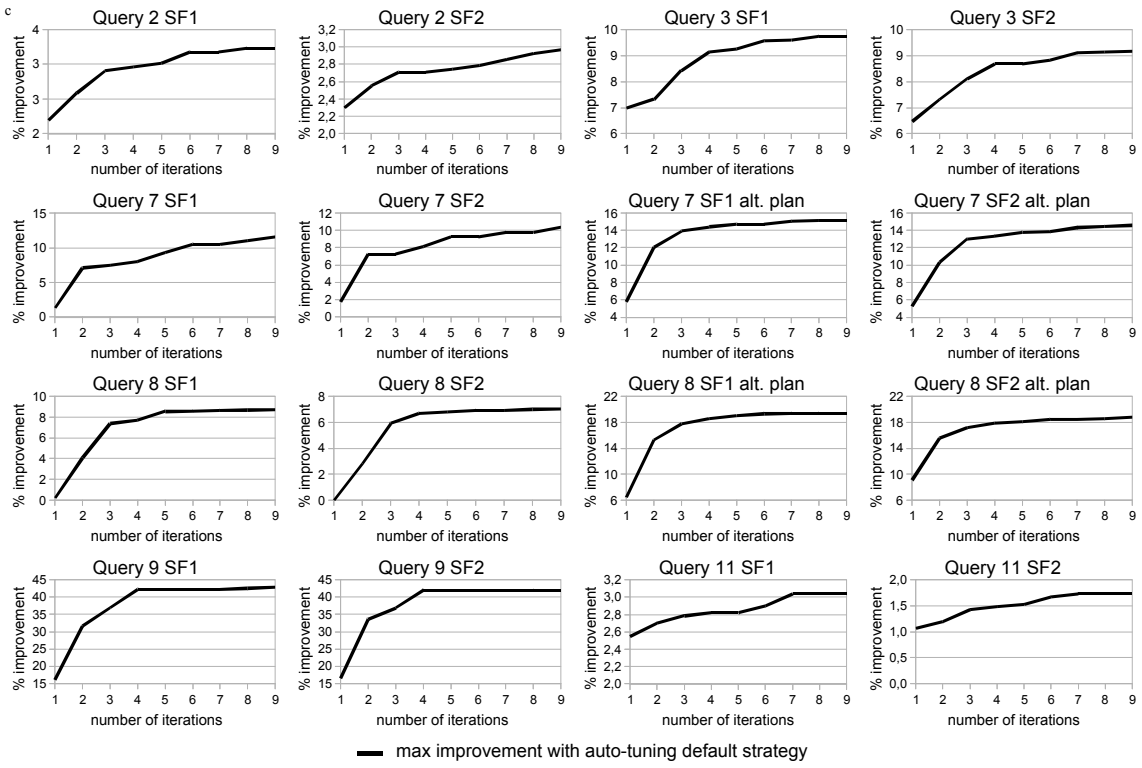


9(a) Intel Core2 Quad

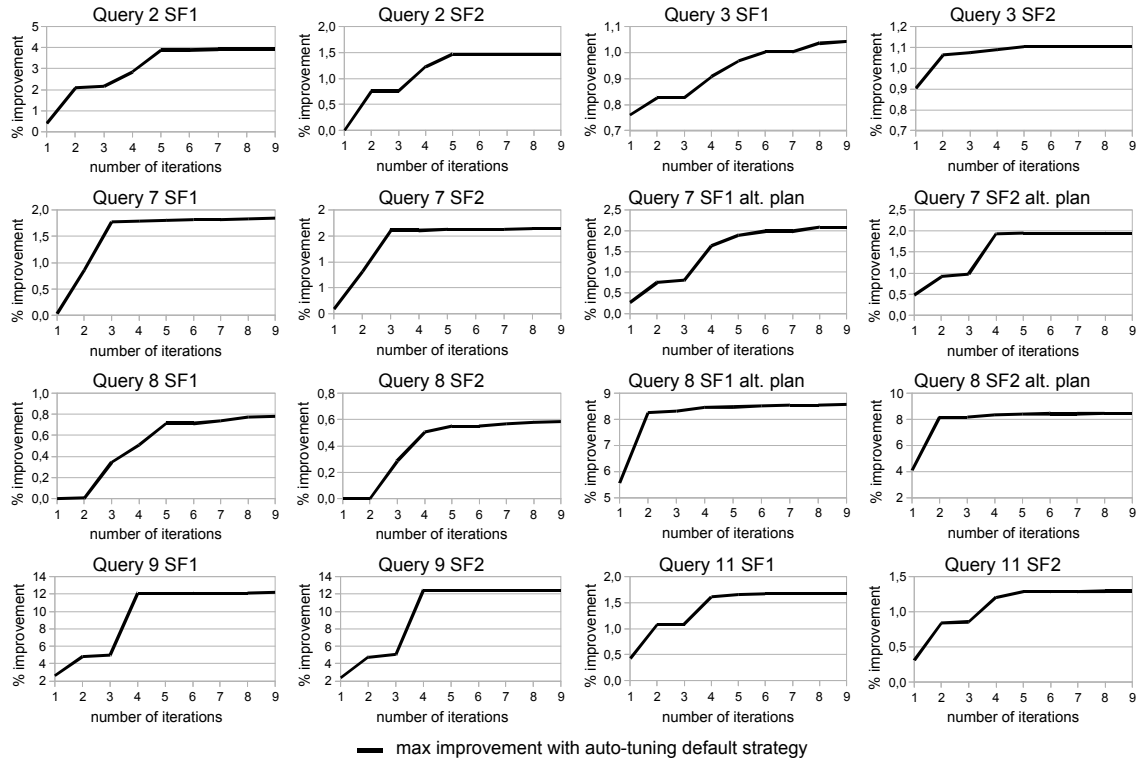


9(b) AMD Quadcore Phenom II

Figure 5: Auto-tuned multithreaded TPC-H on 4-core platforms.



9(a) Intel 8-core machine



9(b) Sun Niagara 8-core machine

Figure 6: Auto-tuned multithreaded TPC-H on 8-core platforms.