Parallelizing Query Optimization on Shared-Nothing Architectures*

Immanuel Trummer and Christoph Koch
{firstname}.{lastname}@epfl.ch
École Polytechnique Fédérale de Lausanne

ABSTRACT

Data processing systems offer an ever increasing degree of parallelism on the levels of cores, CPUs, and processing nodes. Query optimization must exploit high degrees of parallelism in order not to gradually become the bottleneck of query evaluation. We show how to parallelize query optimization at a massive scale.

We present algorithms for parallel query optimization in left-deep and bushy plan spaces. At optimization start, we divide the plan space for a given query into partitions of equal size that are explored in parallel by worker nodes. At the end of optimization, each worker returns the optimal plan in its partition to the master which determines the globally optimal plan from the partition-optimal plans. No synchronization or data exchange is required during the actual optimization phase. The amount of data sent over the network, at the start and at the end of optimization, as well as the complexity of serial steps within our algorithms increase only linearly in the number of workers and in the query size. The time and space complexity of optimization within one partition decreases uniformly in the number of workers. We parallelize single- and multi-objective query optimization over a cluster with 100 nodes in our experiments, using more than 250 concurrent worker threads (Spark executors). Despite high network latency and task assignment overheads, parallelization yields speedups of up to one order of magnitude for large queries whose optimization takes minutes on a single node.

1. INTRODUCTION

Moore's law [15] is breaking and computer systems become more powerful by increasing their number of processing units (be it cores, CPUs, or cluster nodes) rather than by increasing clock rates. This means that all stages of query evaluation must exploit parallelism in order not to become the bottleneck in future systems.

Research on parallelizing query evaluation has so far mainly focused on how to parallelize the actual query processing stage, i.e. how to parallelize the execution of query plans. This is however insufficient as noted in prior work [9, 10, 26, 18]: in order to parallelize query evaluation, we must not only parallelize the execution of query plans but also the *generation* of query plans, i.e. we must develop parallel algorithms for the query optimization problem.

Query optimization is an NP-hard problem and even finding guaranteed near-optimal query plans is NP-hard [3]. The run time of all known algorithms increases exponentially in the number of joins and novel application scenarios (e.g., SPARQL query processing [6]) motivate queries with many joins. Furthermore, the complexity of the systems on which query processing takes place increases: the number of system components keeps increasing (as discussed before), flexible provisioning models and novel processing operators introduce new parameters by which query processing can be tuned (e.g., the number of machines to rent is such a parameter in a cloud scenario [14] or the sampling rate of a scan operator in the context of approximate query processing [1]). All those developments make query optimization harder since the size of the plan search space increases. In addition, many of the aforementioned developments motivate new cost metrics for comparing query plans (e.g., monetary fees in a cloud scenario or result precision in approximate query processing) in addition to execution time. Having multiple plan cost metrics makes query optimization however harder as well [22, 23, 24]. In summary, there are many ongoing developments that make query optimization harder and hence increase the need for parallel query optimization algorithms.

We propose a novel, parallel algorithm for query optimization in this work. Our goal is to obtain a query optimization algorithm that is future-proof in that it is able to exploit the ever-growing degree of parallelism forced by the breakdown of Moore's law. While prior parallel query optimization algorithms have been primarily designed for shared-memory architectures, we aim at parallelizing query optimization on shared-nothing architectures as well. Query plans are often executed on large clusters and, as query optimization must precede query execution, it is preferable to use all cluster nodes for query optimization rather than leave them idle until optimization has finished. Even for queries that are executed repeatedly on a single node, a cluster can be used for optimization before run time if optimization is expensive. The algorithm that we propose is however not specific

*This work was supported by ERC Grant 279804 and by a European Google PhD fellowship.

Proceedings of the VLDB Endowment, Vol. 9, No. 9
Copyright 2016 VLDB Endowment 2150-8097/16/05.

to shared-nothing architectures and can be applied in different scenarios as well.

Prior approaches for parallelizing query optimization assume that worker threads share common data structures [9, 10, 26, 5, 18], in particular big memotables storing subsets of query tables optimal join plans. They assume that a central master node distributes fine-grained optimization tasks to workers and that many interactions between master and worker threads take place during the optimization of a single query. In a shared-nothing architecture, sharing data between worker threads results in high communication overhead and each task assignment incurs setup overhead. We target extremely high degrees of parallelism, at least several hundreds of cluster nodes (while prior algorithms have not been evaluated on more than eight cores). Orchestrating that many nodes on the level of micro optimization tasks results in prohibitive communication and computation overhead on the master node.

Achieving our goals requires a radically different approach compared to prior work: instead of decomposing the query optimization problem into many small optimization tasks, we realize the most coarse-grained problem decomposition possible: the optimization of one query is mapped into exactly one task per worker node.

On a high level, our algorithm works as follows. Given a query to find an optimal plan for, the master optimizer node sends that query together with a plan space partition ID to each worker node. The partition ID is simply an integer between one and the number of workers such that each worker obtains a different number. Each worker node translates its partition ID into a set of constraints on join orders and only considers query plans that comply with those constraints. Each worker node therefore searches for an optimal plan within a plan space that is smaller than the original plan space. The worker nodes search the optimal plan within their respective plan space partition in parallel. No communication between workers or between workers and master node is required during that stage. Afterwards, the workers send the optimal plans back to the master node. The original plan space is the union over all plan space partitions. Comparing the plans returned by the workers, which are optimal within their respective partition and whose number is linear in the number of workers, yields therefore the globally optimal plan.

Our algorithm is designed to exploit very high degrees of parallelism. The time complexity of all serial processing steps, executed by the master node, is linear in the number of workers and in the query size. The amount of data sent over the network is also linear in the number of workers and in the query size. All plan space partitions have the same size which guarantees skew-free parallelization. For a fixed query, the run time as well as the consumption of main memory space per worker node decreases monotonically in the number of worker nodes. Furthermore, the number of partitions into which the plan space can be divided and therefore the maximal degree of parallelism grows in the query size and is in principle unlimited.

Our algorithm parallelizes one of the most popular dynamic programming schemes for query optimization [17]. It treats table sets of increasing cardinality and constructs optimal join plans for each table set out of optimal plans for table subsets that were previously generated. As it has been noted in prior work [9], this dynamic programming scheme belongs to the class of non-serial polyadic algorithms and is therefore difficult to parallelize. Certainly it is easier to parallelize randomized query optimization algorithms such as iterated improvement or simulated annealing [21, 12]. We nevertheless focus on parallelizing the dynamic programming approach. There are two reasons. First, unlike randomized algorithms, the dynamic programming approach formally guarantees to return optimal query plans. Second, by parallelizing Sellinger's classical dynamic programming scheme [17] we parallelize at the same time many query optimization algorithms that have been based on the same scheme and cover a multitude of scenarios (e.g., multi-objective query optimization [22, 23] or parametric query optimization [11]).

The time and space complexity of the classical dynamic programming algorithm depend on the number of table sets for which optimal join plans need to be found. We decompose the query optimization problem by introducing constraints on the join order that ultimately allow to reduce the number of table sets to consider.

We propose a partitioning scheme for the space of left-deep query plans and one partitioning method for bushy query plans. Left-deep query plans are characterized by the order in which tables are joined. We restrict join orders by constraints of the form $x \prec y$ where x and y are query tables: the semantics is that table x needs to be joined before table y. The constraint excludes any query plan producing an intermediate join result containing table y but not table x and hence we can neglect table sets containing y without x during dynamic programming. This reduces the number of table sets to consider by a factor of 3/4. If we assign the constraint $x \prec y$ to a first worker node and the complementary constraint $y \prec x$ to a second worker then the entire search space is covered. Furthermore, we can recursively decompose the resulting plan space partitions by applying similar constraints to other (disjoint) table pairs.

Bushy query plans are binary trees and cannot be represented as join orders anymore. However, if we fix an arbitrary table and follow its way from a leaf node in the plan tree to the root then we can order the other tables based on when they first appear in the sequence of intermediate results we encounter. Hence we restrict join orders for bushy plan spaces by constraints of the form $x \preceq y|z$ with the semantics that x appears no later than y when following table z to the plan tree root. This excludes join results that contain tables y and z but not table x.

We formally analyze time and space complexity and the network bandwidth required by our algorithm. We show that each constraint reduces time complexity by factor 3/4 for linear and by factor 21/27 for bushy plan spaces. We show that those reduction factors are actually optimal within a restricted design space of partitioning methods. Prior algorithms achieved near linear speedups until a low number of threads within a shared-memory architecture. Our speedups are not linear but very steady up to very high degrees of parallelism and within a shared-nothing architecture. In our experiments, we demonstrate continuous scaling up to more than 250 concurrent worker threads on a large cluster over various query sizes and for single as well as multi-objective query optimization. As our algorithm scales even in this challenging scenario, we believe that it scales on many other architectures as well.

The original scientific contributions of this paper are in summary the following:

- We propose a novel algorithm for massively-parallel query optimization on shared-nothing architectures.
- We formally evaluate that algorithm in terms of time and space complexity and in terms of the required network traffic.
- We evaluate the algorithm experimentally on a large cluster, demonstrating its scalability for up to more than 250 concurrent worker threads.

The remainder of this paper is organized as follows. We compare against related work in Section 2. In Section 3, we introduce our formal problem model. We present our algorithms for parallel query optimization in left-deep and bushy plan spaces in Section 4. In Section 5, we analyze time and space complexity as well as the growth in network traffic. In Section 6, we experimentally demonstrate the scalability of our algorithms on a large cluster.

2. RELATED WORK

The term *parallel query optimization* sometimes refers to serial optimization algorithms generating plans that are executed in parallel [4]. We use the term in a different sense: we propose a parallel algorithm for generating query plans.

Our work connects to prior work that parallelizes the classical dynamic programming based query optimization algorithm [9, 10, 26, 27, 5, 18]. Prior algorithms have however implicitly been designed for shared-memory architectures that do not scale beyond a certain degree of parallelism [20]. Prior algorithms have been evaluated on up to maximally eight cores while we demonstrate scalability of our algorithm on a shared-nothing architecture using over 250 workers. We outline some of the factors that distinguish prior algorithm from our algorithm and limit their scalability.

Prior algorithms assume that all threads share common data structures (e.g., the memotable containing partial plans) and can access data generated by other threads. This would lead to huge communication overhead on shared-nothing architectures (e.g., the size of the memotable is exponential in the query size) while our algorithm does not require any communication between workers. Furthermore, prior algorithms use a central coordinator which assigns rather fine-grained optimization tasks to worker threads (e.g., the master thread assigns specific pairs of join operands to generate plans for). This has two disadvantages. First, a lot of communication is required between master and workers. Second, the time complexity for managing the workers is high, so the master itself will eventually become the bottleneck as the degree of parallelism increases.

We assign tasks at the coarsest possible level: each worker receives exactly one task per query. The time complexity of the algorithm executed on the master is linear in the number of worker nodes and in the query size and so is the total amount of data that needs to be sent over the network. Finally, only one round of communication between workers and master is required per query by our algorithm while prior algorithms usually require many rounds of communication. Having only one round of communication is advantageous in scenarios where distributing tasks to workers and receiving the results is associated with overheads. We compare against a typical representative of prior algorithms in our experimental evaluation.

Our work is generally relevant for all areas of query optimization in which algorithms based on dynamic programming have been proposed. This includes, for instance, multi-objective query optimization [22, 24], parametric query optimization [7, 13], and multi-objective parametric query optimization [24]. Our method of partitioning the join order space is generic and can be applied to all of those scenarios.

3. PROBLEM MODEL

As it is standard in query optimization, we use a simplified query and query plan model to describe our algorithms. Extending the model and the algorithms towards richer query languages and plan spaces is however straightforward and can be achieved via standard techniques [17].

A query is a set Q of tables that need to be joined. We denote by $\textsc{Scan}(q)$ for $q \in Q$ a query plan that scans a single table and call such a plan a *scan plan*. By $\textsc{Join}(p_L, p_R)$ we designate a plan that joins the result produced by plan p_L with the result produced by p_R and uses p_L as outer and p_R as inner operand. We use the terms left and right operand as synonyms for outer and inner operand respectively. We do not incorporate alternative scan and join operator implementations into our model to simplify the presented pseudocode. The extension is however easy and our implementation of our algorithm considers all standard operators.

We distinguish two types of query plans. *Left-deep plans* are plans in which the right operand of every join is a scan plan. All other plans are *bushy plans*. Bushy plans can be represented as labeled binary trees where leaf nodes correspond to single tables and inner nodes correspond to join results. The tree shape of left-deep plans is fixed and the join order of a left-deep plan is fully described by the order in which table leaf nodes are encountered in a traversal (e.g., in post-order) of the plan tree. This is why we can represent left-deep plans by a sequence of query tables.

For a fixed query, the set of all bushy plans is the *bushy plan space* and the set of all left-deep plans is the *left-deep* or *linear plan space*. We assume that a cost model is available that associates query plans with cost estimates. Our pseudocode encapsulates that cost model in a pruning function that discards the plan with higher cost among several compared plans. The goal of query optimization is to find the cost-optimal plan either in the space of left-deep or in the space of bushy plans.

4. ALGORITHM

We present an algorithm for massively-parallel query optimization. The algorithm is well suited for shared-nothing architectures as it minimizes the amount of sychronization and communication overhead. The same properties are however beneficial in shared-memory scenarios. Our algorithm is not specific to shared-nothing architectures and can be used to parallelize query optimization over the nodes of a cluster or over the cores of a single computer all the same.

The presented algorithm solves the traditional query optimization problem, meaning that it compares alternative query plans according to single point cost estimates in one cost metric. The method by which we partition the plan space is however very generic and it is in fact straightforward to extend our algorithm to handle multiple plan cost metrics [22, 23] or plan cost functions that depend on unknown parameters [13, 7] or both together [24]. This is possible since algorithms have been proposed for all of the aforementioned query optimization variants that use the same dynamic programming scheme as the classical algorithm by Selinger [17]; only the pruning function, the way in which different query plans are compared, differs between them. The algorithm presented next can therefore easily be transformed into an algorithm handling other query optimization variants by essentially replacing the pruning function.

We present two variants of our algorithm: the first variant finds the optimal left-deep query plan for a given query while the second variant finds the optimal plan within a bushy plan space. Before discussing the pseudo-code, we illustrate informally how our algorithm works by means of a simplified example. This example refers to the algorithm variant searching left-deep plan spaces.

Example 1. *Assume we want to find the optimal left-deep plan for answering the join query $R \bowtie S \bowtie T \bowtie U$. Further assume that four worker nodes are available over which query optimization is parallelized. Upon reception of the query, the master nodes sends the query together with the total number of plan space partitions (four) and the respective partition ID (between one and four) to each worker node. Consider the worker node that partition three is assigned to. Knowing that the total number of partitions is four, the worker node derives that it should use $\log_2 4 = 2$ constraints to restrict the join order space. The two constraints refer to the order in which the four tables are joined. The first constraint refers to the ordering between the first pair of tables, R and S, and establishes which of them appears first in the join order. The second constraint refers to T and U. The binary representation of the partition ID encodes the concrete set of constraints to use. For the considered worker node, the partition ID is 10 in binary representation. The first bit of the binary representation is zero so the worker node orders R before S. As the second bit is one, the worker orders U before T. Note that other workers will use complementary constraint sets based on their respective partition ID such that the whole join order space is covered. The worker that we focus on finds the best plan whose join order complies with the given constraints. It returns that plan to the master which compares the plans returned by all workers to determine the globally optimal plan.*

We present pseudo-code for the high-level algorithm that is executed by the master and the worker nodes in Section 4.1. The code of the sub-functions that the workers use to infer constraints on the join order from the partition ID and to find join orders that comply with the constraints are discussed in Section 4.2.

4.1 High-Level Algorithm

We present pseudo-code for the high-level algorithms that are executed on the master node and on the workers. As it is common in the area of query optimization, we simplify the presented pseudo-code by considering only SPJ queries. There are however standard methods by which such algorithms can be extended to support richer query languages [17] (e.g., queries with aggregates or nested queries).

```
// Parallelizes optimization of query $Q$ over $m$ machines.
function Master($Q, m$)
    // Generate best plan for each partition in parallel
    parfor $partID \in \{1, \ldots, m\}$ do
        $bestInPart[partID] \leftarrow$Worker($Q, partID, m$)
    end parfor
    // Prune plans and returns best plan
    $bestPlan \leftarrow bestInPart[1]$
    for $partID \in \{2, \ldots, m\}$ do
        FinalPrune($bestPlan, bestInPart[partID]$)
    end for
    return $bestPlan$
end function
```

Algorithm 1: Function executed by master node for parallel query optimization on shared-nothing architectures.

As announced before, we present two algorithm variants, one treating the space of left-deep plans, the other one treating the space of bushy plans. The pseudo-code that we discuss in this subsection is however the same for both variants such that we do not need to distinguish between them.

Our algorithm consists of two parts: the first part is executed by the master node which orchestrates the worker nodes. The second part of our algorithm runs on the worker nodes. Algorithm 1 shows the code that is executed on the master. The input is a query Q, for which we want to find an optimal query plan, and the number m of available worker nodes. We assume in the following that m is a power of two (the reason will become apparent in the following). The output of the Master function is the optimal plan for Q.

The master node executes two phases. In the first phase, the master sends the query together with a unique partition ID to each of the workers[1]. We discuss the pseudo-code of the Worker function a bit later. All worker invocations happen in parallel as indicated by the keyword **parfor**. The partition ID identifies a partition of the plan search space. The task of each worker is to find the optimal plan within its respective partition and to return it to the master. The master collects the returned plans in the array *bestInPart* (we use the standard notation *bestInPart*[x] to represent an access to the x-th field of that array). In the second phase, the master node compares all collected plans to identify the globally-optimal plan. Function FinalPrune, whose pseudo-code we do not specify, represent a standard pruning function that replaces *bestPlan* by the better plan among the two input plans. Having considered all plans returned by the workers, the best plan must be globally optimal.

Note that workers need access to metadata (e.g., cardinality and value distribution statistics) to estimate plan execution costs. Either the master node sends query-specific statistics to the workers together with each query (e.g., selectivity estimates for the query predicates) or all relevant statistics are regularly distributed to and stored on the worker nodes. Which approach is preferable depends on the amount of metadata and its update frequency.

Algorithm 2 shows the code of the function that runs on worker nodes and is invoked by the master. The input is the query Q to optimize, the total number m of plan space partitions, and the identifier *partID* of the partition that is

[1] If worker nodes are heterogeneous then the number of partitions treated by a worker should be proportional to its performance.

```
// Generate best plan for query $Q$ in partition with
// ID $partID$ out of $m$ partitions.
function Worker($Q, partID, m$)
    // Decode partition ID into a set of constraints
    $constr \leftarrow$PartConstraints($Q, partID, m$)
    // Generate admissible intermediate results
    $joinRes \leftarrow$AdmJoinResults($Q, constr$)
    // Initialize best plans for single tables
    for $q \in Q$ do
        $P[q] \leftarrow$Scan($q$)
    end for
    // Iterate over join result cardinality
    for $k \in \{2, \ldots, |Q|\}$ do
        // Iterate over admissible join results
        for $q \in joinRes : |q| = k$ do
            // Try splits of $q$ into two join operands
            TrySplits($q, constr, P$)
        end for
    end for
    // Return best plan for query $Q$
    return $P[Q]$
end function
```

Algorithm 2: Generate best query plan within specific partition of either linear or bushy plan space.

assigned to the respective worker. The output is the optimal plan within the corresponding partition. Each worker node executes the following three steps. First, knowing the total number m of partitions, the specific partition ID $partID$ can be translated into a set of constraints on the join order. Function PartConstraints, whose code is discussed later, accomplishes the translation. Second, function AdmJoinResults translates the set of constraints into an admissible set of table sets that can appear as join results within a query plan whose join order respects the constraints. Finally, the worker node uses a dynamic programming approach to find the optimal query plan among all plans that produce only admissible join results. We assume, without explicitly writing out the corresponding code, that the result sets generated by function AdmJoinResults have been indexed by their cardinality such that Algorithm 2 can efficiently retrieve all sets with a given cardinality k.

Variable P is an array storing optimal query plans and $P[Q]$ designates the optimal query plan for joining the table set Q. We initialize P by inserting the scan plan for each single query table $q \in Q$. We simplify the pseudo-code by assuming only one scan plan per table but the generalization is straight-forward. After that, the algorithm calculates optimal plans for table sets of increasing cardinality, using the optimal plans that were stored in prior iterations. The algorithm considers only table sets that represent admissible join results. For each admissible join result, function TrySplits tries all ways of splitting the join result into two admissible operands and stores the best resulting plan in P.

4.2 Plan Space Partitioning

We discuss the sub-functions invoked by the Worker function. In contrast to the previous subsection, we now need to distinguish between the two algorithm variants that we present. In the following pseudo-code, we use the notation F$[LINEAR]$ to indicate that function F is specific to the algorithm searching linear (or left-deep) search spaces. Analogously, F$[BUSHY]$ indicates a function that is specific to the algorithm generating bushy plans. The code of all other functions is the same for both variants.

```
// Generate constraint on $i$-th table pair of
// query $Q$ using precedence order $precOrd$.
function Constraint[Linear]($Q, i, precOrd$)
    if $precOrd = 0$ then
        return $Q_{2\cdot i} \prec Q_{2\cdot i+1}$
    else
        return $Q_{2\cdot i+1} \prec Q_{2\cdot i}$
    end if
end function
// Generate constraint on $i$-th table tuple of
// query $Q$ using precedence order $precOrd$.
function Constraint[Bushy]($Q, i, precOrd$)
    if $precOrd = 0$ then
        return $Q_{3\cdot i} \preceq Q_{3\cdot i+1}|Q_{3\cdot i+2}$
    else
        return $Q_{3\cdot i+1} \preceq Q_{3\cdot i}|Q_{3\cdot i+2}$
    end if
end function
// Decode partition ID $partID$ into a set of constraints
// restricting the plan space for query $Q$. The total
// number of partitions is $m$ and $partID \leq m$.
function PartConstraints($Q, partID, m$)
    // Initialize constraint set
    $constr \leftarrow \emptyset$
    // Iterate over constraints
    for $i \in \{0, \ldots, \log_2(m) - 1\}$ do
        // $i$-th bit encodes precedence order
        $precOrd \leftarrow$Bit($partID, i$)
        // Generate constraint on $i$-th subset of $Q$
        $c \leftarrow$Constraint($Q, i, precOrd$)
        // Add new constraint into set
        $constr \leftarrow constr \cup c$
    end for
    return $constr$
end function
```

Algorithm 3: Translate the partition ID into a set of constraints that restrict the plan search space.

Algorithm 3 shows the code for translating a partition ID into a set of constraints. Function PartConstraints obtains as input the query, the number of partitions, and the partition ID. The output is a set of constraints on the join order that define the plan space partition that the current worker needs to treat.

When generating constraints, we use the notation Q_x with $x \in \mathbb{N}$ to designate the x-th table in query Q. This notation assumes that query tables have been numbered consecutively from 0 to $|Q|-1$. The algorithm can use an arbitrary table numbering but it is important that all workers use the same numbering in order to guarantee that the whole plan space is covered by the ensemble of workers.

The form of the generated constraints differs depending on whether we search for left-deep or bushy plans. Constraints for the left-deep plan space are defined on table pairs while constraints on bushy plans are defined on triples of tables. Constraint restricting the linear plan space are of the form $Q_x \prec Q_y$. This means that the x-th table must appear before the y-th table in an admissible join order (the join order of a left-deep plan can be represented as a sequence of tables and the constraints refer to that representation).

Constraints restricting bushy plan spaces are of the form $Q_x \preceq Q_y|Q_z$ with the semantic that when considering the intermediate join results containing table Q_z in ascending order of cardinality, table Q_y must not appear before table Q_x. We assume that constraints have been indexed such that all constraints concerning a given set of tables can be retrieved efficiently.

In case of a left-deep plan space there are two complementary constraints for each pair of tables, namely $Q_x \prec Q_y$ and $Q_y \prec Q_x$. In order to guarantee that the whole plan space is covered by the ensemble of workers, we need to consider complementary constraints by different workers. All workers use constraints on the same table pairs but the direction of those constraints (which of the two tables to join first) differs among workers. Each worker uses the binary representation of the partition ID to derive which of the two possible constraints to consider for each table pair. We use the notation Bit$(partID, i)$ to represent the i-th bit of the binary representation (it does not matter whether we start with the lowest order bits or with the highest order bits). Each bit determines the direction for one constraint.

The treatment of bushy plan spaces is analogue. Constraints are defined on table triples but for each triple of tables there are still just two complementary constraints and each worker picks between them based on the partition ID. We define two variants of the function Constraint that generates the actual constraints: one for the linear and one for the bushy plan space. The high-level algorithm for generating constraint sets does not differ between them.

Note that we have assumed that the number of workers is a power of two and that the number of query tables is a multiple of two for left-deep plans and a multiple of three for bushy plans. Those assumptions simplify our pseudocode while the extension to the general case (i.e., using only a subset of workers whose cardinality is a power of two) are straight-forward. The number of workers that can be efficiently exploited by our algorithm is however indeed restricted to powers of two and the maximal number of workers is additionally restricted as a function of the query size. We analyze those restrictions in more detail in Section 5.

Constraints restrict the admissible join orders and join trees. We are however ultimately interested in restricting the number of intermediate results, i.e. join result table sets, that can appear in admissible plans. The time and space complexity of the dynamic programming algorithm executed by the workers depends on that.

We must translate sets of constraints into sets of intermediate results that admissible plans can use. Algorithm 4, more precisely function AdmJoinResults, accomplishes the translation. The input is the query and a set of constraints. The output is the set of intermediate results that can appear in plans that comply with those constraints.

Function AdmJoinResults iterates over all subsets of query tables that constraints can refer to. For left-deep plans those are all pairs of tables with consecutive numbers. For bushy plans those are all triples of consecutive tables. In each iteration of the for loop, the function extends the admissible table sets stored in R by subsets of the table pair (or table triple) considered in the current iteration using a Cartesian product for the extensions. The auxiliary function ConstrainedPowerSet returns for a given pair (respective triple) or tables all subsets that comply with the constraints. More precisely, if table Q_x needs to be joined before table Q_y in case of left-deep plans then (non-singleton) table sets containing Q_y but not table Q_x do not need to be considered. Equally for bushy plans, if table Q_x must appear before table Q_y when enumerating all table sets containing Q_z then table sets containing Q_y and Q_z but not Q_x are not admissible as join results.

```
// Returns pairs of consecutive tables in query $Q$
function Subsets[Linear]($Q$)
    return $\{\{Q_{2\cdot i}, Q_{2\cdot i+1}\}|0 \leq i \leq |Q|/2-1\}$
end function
// Returns triples of consecutive tables in query $Q$
function Subsets[Bushy]($Q$)
    return $\{\{Q_{3\cdot i}, Q_{3\cdot i+1}, Q_{3\cdot i+2}\}|0 \leq i \leq |Q|/3-1\}$
end function
// Part of power set of $S$ respecting constraints $C$
function ConstrainedPowerSet[Linear]($S, C$)
    return Power($S$)$\setminus\{\{Q_y\}|(Q_x \prec Q_y) \in C\}$
end function
// Part of power set of $S$ respecting constraints $C$
function ConstrainedPowerSet[Bushy]($S, C$)
    return Power($S$)$\setminus\{\{Q_y, Q_z\}|(Q_x \preceq Q_y|Q_z) \in C\}$
end function
// Returns all potential join results (table subsets
// of query $Q$) that comply with constraints $C$.
function AdmJoinResults($Q, C$)
    // Initialize result sets
    $R \leftarrow \{\emptyset\}$
    // Iterate over subsets of $Q$
    for $S \in$Subsets($Q$) do
        // Extend join results using Cartesian product
        $R \leftarrow R \times$ConstrainedPowerSet($S, C$)
    end for
    return $R$
end function
```

Algorithm 4: Generate all table subsets that comply with the constraints defining a search space partition.

Example 2. *Assume that* $Q = \{Q_1, Q_2, Q_3, Q_4\}$ *and that we have the two constraints* $C = \{Q_1 \prec Q_2, Q_4 \prec Q_3\}$, *hence we consider left-deep plans. Then the set of admissible join result sets is generated in function* AdmJoinResults *as follows. In the first iteration of the for loop, we extend the elements contained in R (initially this is only the empty set) with the admissible subsets of the first table pair* $\{Q_1, Q_2\}$. *The admissible subsets are* $\{\{\}, \{Q_1\}, \{Q_1, Q_2\}\}$ *and this is at the same time the content of R after the first iteration. The algorithm considers admissible subsets of* $\{Q_3, Q_4\}$ *in the second iteration (which are the sets* $\{\}$, $\{Q_4\}, \{Q_3, Q_4\}$*) and extends each element with all of the admissible subsets. Hence* $R = \{\{\}, \{Q_1\}, \{Q_1, Q_2\}, \{Q_4\}, \{Q_1, Q_4\}, \{Q_1, Q_2, Q_4\}, \{Q_3, Q_4\}, \{Q_1, Q_3, Q_4\}, \{Q_1, Q_2, Q_3, Q_4\}\}$ *after the second iteration.*

Note that the admissible table sets generated by function AdmJoinResults do not include all singleton table sets. While all singleton sets must be considered to generate any plan (since we need to select scan plans for each table), singleton sets are treated separately in Algorithm 2 and it does not matter which of them are included in the result of function AdmJoinResults.

Algorithm 5 shows the function trying out different splits and generating corresponding plans that applies for left-deep

```
1: // Try all splits of $U \subseteq Q$ into two operands respecting
2: // constraints $C$, generate associated plans and prune.
3: function TrySplits[Linear]$(Q, U, C, P)$
4:     // Iterate over potential inner operands
5:     for $u \in U$ do
6:         // Check if operand choice satisfies constraints
7:         if $\nexists v \in U : (u \prec v) \in C$ then
8:             $p \leftarrow$Join$(P[U \setminus u], P[u])$
9:             Prune$(P, p)$
10:        end if
11:    end for
12: end function
```

```
13: // Try all splits of $U \subseteq Q$ into two operands respecting
14: // constraints $C$, generate associated plans and prune.
15: function TrySplits[Bushy]$(Q, U, C, P)$
16:     // Determine admissible operands
17:     $A \leftarrow \{\emptyset\}$
18:     // Iterate over set of table triples
19:     for $T \in$Subsets[Bushy]$(Q)$ do
20:         // Restrict triple to tables in join result
21:         $S \leftarrow T \cap U$
22:         // Form power set of remaining triples
23:         $S \leftarrow$Power$(S)$
24:         // Take out sets violating constraints
25:         $S \leftarrow S \setminus \{\{Q_y, Q_z\} | (Q_x \preceq Q_y | Q_z) \in C\}$
26:         // Remove complement of inadmissible sets
27:         $S \leftarrow S \setminus \{\{Q_x\} | (Q_x \preceq Q_y | Q_z) \in C; Q_y, Q_z \in U\}$
28:         // Extend admissible splits by Cartesian product
29:         $A \leftarrow A \times S$
30:     end for
31:     // Full set and empty set do not qualify as operands
32:     $A \leftarrow A \setminus \{\emptyset, U\}$
33:     // Iterate over admissible left operands
34:     for $L \in A$ do
35:         // Generate plans associated with splits
36:         $p \leftarrow$Join$(L, U \setminus L)$
37:         // Discard suboptimal plans
38:         Prune$(P, p)$
39:     end for
40: end function
```

Algorithm 5: Generate and prune query plans that correspond to different splits of a join result into two operands.

plans. This function is called by Algorithm 2 for each admissible join result. The function iterates over all tables in the join result set U and tries all of them as inner join operands as long as none of the constraints is violated. Plans corresponding to admissible splits are generated and function Prune, whose pseudo-code we do not specify, compares the newly generated plan against the best plan known so far that produces the same tuples in the same order [17] as the new one. Sub-optimal plans are discarded. The pruning function used by the workers might differ from the one used by the master (called FinalPrune in Algorithm 1) as the tuple ordering is for instance only relevant as long as it can reduce the cost of future operations and does not need to be taken into account anymore for completed plans.

There are actually two mechanisms by which partitioning reduces the time complexity per worker. So far we have focused on the first one: partitioning reduces the time complexity per worker since fewer potential join results need to be considered. An additional advantage of partitioning is however that it allows to reduce the number of splits of join results into two join operands, leading to different query plans that need to be generated and compared.

The potential for saving computation time by reducing the number of splits is higher for bushy plan spaces since the number of possible splits grows exponentially in the size of the join result. For left-deep plans, the number of splits grows only linearly in the cardinality of the join result as the right join operand is limited to singleton table sets.

This is why we invest more effort in case of bushy than in case of left-deep plans into properly exploiting the reduction of admissible splits. For left-deep plans, we basically enumerate all possible splits and check whether they comply with the constraints. The complexity of that approach remains linear in the number of possible splits and not in the lower number of admissible splits. The algorithm for bushy plans is more sophisticated as it avoids generating non-admissible splits for bushy plans in the first place. Hence its complexity is linear in the number of admissible rather than possible splits.

Function TrySplits[Bushy] generates all admissible splits in a bushy plan space and generates and prunes the associated query plans. The algorithm first generates all admissible join operands and stores them in variable A. Each admissible join operand corresponds to the union of one admissible subset for each table triple (constraints are defined on triples of tables). This is why we iterate over all table triples, determine all admissible subsets of the current triple, and combine in each iteration each operand in A with each admissible subset of the current triple (using a similar approach as in Algorithm 4). For a given triple of query tables, we only consider the ones that are included in the join result U that needs to be split. If no constraints are defined on the current triple then the entire power set of the contained table is admissible. Otherwise, we must remove subsets violating the precedence constraints (line 25) but we must also remove subsets whose complement (in the contained triple tables) violates the precedence constraints (line 27) as the second join operand is the complement of the first one.

Having determined all admissible join operands (whose complement is admissible, too), we iterate over all of them, generate plans and discard sub-optimal plans.

In principle, our partitioning method can parallelize query optimization algorithms that do not implement the classical dynamic programming scheme. The classical dynamic programming scheme seems however particularly amenable to partitioning since run time is guaranteed to be proportional to the number of intermediate results. This means that run time does not vary significantly across workers, thereby avoiding skew. It is a-priori unclear by how much query optimization algorithms without that property, e.g. the Volcano algorithm [8], benefit from partitioning.

5. COMPLEXITY ANALYSIS

We analyze the asymptotic amount of data sent over the network in Section 5.1, the consumed amount of main memory in Section 5.2, and the execution time in Section 5.3.

Throughout Sections 5.1 to 5.3, we simplify the analysis by assuming only one scan and join operator, one cost metric, and no interesting orders. We generalize the analysis in Section 5.4. In Section 5.5, we discuss the question of whether our partitioning methods can be improved and show that they are optimal at least within a restricted space.

We denote by $n = |Q|$ the number of query tables to join and by m the number of worker machines. We assume that $m \leq 2^{\lfloor n/2 \rfloor}$ for linear plan search spaces and $m \leq 2^{\lfloor n/3 \rfloor}$ for bushy plan spaces. We denote by $l = \lfloor \log_2(m) \rfloor$ the number of constraints per plan space partition. By b_q we designate the byte size of the input query. By b_p we denote the byte size of a corresponding plan.

5.1 Network Communication

We analyze the communication overhead per query.

THEOREM 1. *The amount of data sent over the network is in $O(m \cdot (b_q + b_p))$.*

PROOF. Different workers do not communicate with each other so data is only sent between master and workers. Initially, the query and two integer numbers are sent to each worker. If statistics are sent with the query, we assume that their size is proportional to the query size. Hence, the input size per worker is in $O(b_q)$. We consider one plan cost metric and no interesting orders (while extensions are discussed later). The output of each worker is therefore a single query plan with space consumption b_q. □

5.2 Main Memory

We analyze the amount of main memory that each worker requires during optimization. Note that the main memory consumption of the master is negligible as it delegates optimization. The main memory consumed per worker node depends on the number of admissible join results.

THEOREM 2. *Each linear plan space partition restricted by l constraints has $O(2^n \cdot (3/4)^l)$ admissible join results.*

PROOF. The proof is an induction over the number of constraints l. For $l = 0$ (induction start), all subsets of Q are admissible and their number is in $O(2^n)$. Assume the induction holds up to L constraints. We will see that it holds for $L + 1$ constraints as well. All constraints refer to different tables. Hence the first L constraints do not influence the occurrence frequency of the two tables x and y that the $L+1$-th constraint refers to. More precisely, among the table sets that remain admissible after considering the first L constraints, the fraction of table sets containing x and y, x but not y, y but not x, and neither x nor y, is always 1/4. Denote by $x \prec y$ the $L+1$-th constraint stating that we must join x before y. Then join results containing y but not x are inadmissible, the number of admissible table sets is reduced by factor 3/4, and the induction holds. □

THEOREM 3. *Each bushy plan space partition restricted by l constraints has $O(2^n \cdot (7/8)^l)$ admissible join results.*

PROOF SKETCH. The proof follows closely the one of Theorem 2 with the difference that each constraint of the form $x \preceq y|z$ excludes all table sets that contain y and z but not x and their fraction is always 1/8 among the table sets satisfying all other constraints. □

THEOREM 4. *The main memory consumption per node is in $O(2^n \cdot (3/4)^l)$ for linear plan spaces and $O(2^n \cdot (7/8)^l)$ for bushy plan spaces.*

PROOF. The main memory consumption per worker dominates the consumption of the master. The variable with dominant space consumption are the ones storing admissible join results (variable *joinRes* in Algorithm 2) and the one assigning table sets to optimal plans (variable P). We currently assume one plan cost metric and therefore only one plan is optimal per table set. Storing plans generally takes $O(n)$ space but here each plan can be represented by at most two pointers to optimal sub-plans stored for table subsets which requires only $O(1)$ space. The total main memory consumption follows from Theorems 2 and 3. □

5.3 Execution Time

We analyze time complexity. Note that the pseudo-code presented in Section 4 is rather abstract and does not contain certain steps that are crucial for efficiency: as we mentioned in Section 4, we assume for instance that constraints are indexed such that we can find all constraints in which a given table appears in constant time. For the analysis, we assume that such commonsense optimizations have been applied. We first analyze execution time on the master.

THEOREM 5. *The master requires $O(m \cdot (b_q + b_p))$ time.*

PROOF. The master distributes the query and the partition ID to all m workers. Assuming that the required time is proportional to the amount of data being sent, distributing tasks takes $O(mb_q)$ time and collecting plans from the workers is in $O(mb_p)$. After receiving all plans, the master iterates over the m plans that were returned from the workers (and whose cost was already calculated) and determines the one with minimal cost. This has complexity $O(m)$. □

We analyze time complexity on the worker nodes.

THEOREM 6. *The time complexity for processing a linear plan space partition by one of the workers is $O(n \cdot 2^n \cdot (3/4)^l)$.*

PROOF. A worker performs three main steps per invocation: translating the partition ID into constraints, translating constraints into admissible join result sets, and determining the optimal plan among the plans using only admissible join results. The operation with dominant time complexity is the determination of the optimal plan. For each admissible join result set, we iterate over less than n inner join operands. The number of admissible join result sets is in $O(2^n \cdot (3/4)^l)$ according to Theorem 2. Generating a plan from two sub-plans, calculating its cost via recursive formulas, and comparing it with the best previously generated plan joining the same tables requires only constant time. □

THEOREM 7. *The time complexity for processing a bushy plan space partition by one of the workers is $O(3^n \cdot (21/27)^l)$.*

PROOF. Finding an optimal plan in a plan space partition is the operation with dominant time complexity. Its complexity is proportional to the number of considered join operand pairs. For each table there are in general three possibilities for how it appears in a pair of join operands: either it appears in the left operand or in the right operand or it does not appear (neither in the operands nor in the join result). Join operands are constructed from admissible subsets of table triples. If no constraint is defined on a given triple then all splits are admissible which makes $3^3 = 27$ possible pairs. If a constraint is defined on a triple then some of those 27 possibilities are not admissible. If the constraint is $x \preceq y|z$ then the following six splits of triple $\{x, y, z\}$ are excluded: all splits whose union contains y and z but not x (this applies to four splits) and all splits that assign y and z to one operand and x to the other one (this applies to two splits). The ratio of admissible to possible splits is therefore 21/27 for each triple with a constraint on it. □

As the time complexity of the worker processes dominates the complexity of the master process and as all workers execute in parallel, the time complexity of a single worker is the complexity of the entire optimization process.

5.4 Extensions

So far we considered one plan cost metric, no interesting orders, and no alternative operator implementations. Now we sketch out how to generalize the analysis.

Considering multiple alternative operator implementations for scan and join operations influences only time complexity. Time complexity grows linearly in the number of operators as each join operator implementation must be considered for each possible pair of join operands. Annotating the operations within query plans by an operator ID does neither change asymptotic main memory consumption nor asymptotic communication overhead.

Considering interesting orders means that we have to store one optimal plan per interesting order and per table set. Considering multiple cost metrics has a similar effect as we need to store a set of Pareto-optimal plans for each table set. The number of plans sent from workers to master, and therefore communication overhead, increases linearly in the number of plans stored per table set. Main memory consumption also increases linearly in the number of plans. Time complexity increases proportionally to the cube of the number of plans per table set: when searching for the optimal plan within each plan space partition, we need to consider all pairs of optimal plans for each split of a table set into two join operands [22]. This accounts for a quadratic increase in complexity. Additionally, pruning might take longer as we need to compare plans not against one but multiple optimal plans. This implies a cubic complexity growth.

5.5 Optimality of Partitioning

Execution time and main memory consumption both depend on the number of intermediate join results that need to be treated by each worker. With our partitioning scheme, the number of join results per worker reduces by factor 3/4 in case of linear plan spaces and by factor 7/8 for bushy plans, each time that the number of workers doubles. As the ideal factor of 1/2 is not reached there must be join results that are assigned to multiple workers. This raises the question of whether we can do better and reduce the number of intermediate results per worker node by a lower factor.

We answer that question for partitioning methods that are similar to the one we apply. Those are methods that divide the power set of query tables into subsets based on which out of two, respective three, fixed tables are present. Each of the resulting subsets is assigned to part of the workers and each worker generates all plans whose join results are contained in its assigned subsets (each worker constructs scan plans for all single tables, independently from the assigned join results). Workers do not exchange partial plans and hence must generate completed plans and start optimization from scratch. We study the case of two workers in the following but the reasoning can be generalized.

THEOREM 8. *Doubling the number of workers cannot reduce the maximal number of join results per worker by less than factor 3/4 in linear plan spaces.*

PROOF. For a fixed pair of tables $\{x, y\}$ out of all query tables, we denote by $\overline{x}y$ the set of table sets containing y but not x, by xy the sets containing both tables, by $\overline{xy}$ sets containing neither x nor y, and by $x\overline{y}$ the remaining sets. Each worker must obtain subset xy in order to generate complete plans. The cardinality of the set of joined tables can only increase by one from one join to the next in a left-deep plan space. Each worker needs therefore either join results from $\overline{x}y$ or from $x\overline{y}$ in order to generate any valid plan. Set $\overline{xy}$ must be assigned to at least one worker since the plan space partitioning is otherwise incomplete. □

THEOREM 9. *Doubling the number of workers cannot reduce the maximal number of join results per worker by less than factor 7/8 in bushy plan spaces.*

PROOF SKETCH. For a triple of tables $\{x, y, z\}$, we use a similar notation as before to characterize join result sets and denote for instance by $x\overline{y}z$ all sets containing x and z but not y. Both workers require xyz for the same reason as before. Assume that we do not assign the set $\overline{xyz}$ to both workers. The worker to which $\overline{xyz}$ is assigned is the only worker that can consider plans joining the other tables besides x, y, and z independently before joining with the triple tables. This means that this worker needs to cover all possible join orders for x, y, and z. Hence it requires all join result sets which defeats the purpose of partitioning.

Assume now that we do not assign the set $x\overline{yz}$ to the first worker. Then the second worker is the only one that can consider plans of the form $(x \bowtie \ldots) \bowtie (y \bowtie \ldots)$ and hence requires $\overline{x}y\overline{z}$ and by analogue reasoning also $\overline{xy}z$ in addition to $x\overline{yz}$ in order to make sure that the whole plan space is covered. As the second worker is at the same time the only one that can consider plans of the form $((x \bowtie \ldots) \bowtie y) \bowtie \ldots$, it requires at the same time $xy\overline{z}$ and $x\overline{y}z$. Since only the second worker can treat plans of the form $(x \bowtie \ldots) \bowtie (y \bowtie z)$, it requires also $\overline{x}yz$. So the second worker obtains at least 7 sets of join results. The same happens when not assigning $\overline{x}y\overline{z}$ or $\overline{xy}z$ to the first worker. We have the option of not assigning one of the three sets containing two out of the three tables $\{x, y, z\}$ to the first worker in which case we need to assign the other two to the second worker. The maximal number of intermediate result splits per worker remains 7/8. □

6. EXPERIMENTAL EVALUATION

We evaluate the scalability of our query optimization algorithm on a large cluster with 100 nodes. Parallelizing query optimization on clusters is useful if query plans are also executed on a cluster: it is preferable to use all available resources for optimization instead of leaving nodes idle until serial optimization finishes. While parallelizing query optimization on a cluster is hence a relevant application scenario, we also selected it specifically because it is a very challenging scenario for parallelization due to high communication cost and setup overhead. The fact that our algorithm scales even on a cluster provides strong evidence for that it scales in a multitude of other scenarios, too.

Section 6.1 describes our experimental setup and Section 6.2 our experimental results.

6.1 Experimental Setup

We evaluate our algorithm on a cluster with 100 nodes. Each node is equipped with two Intel Xeon E5-2630 v2 CPUs featuring six cores each running at 2.60GHz; 128 GB of main memory and 20 TB of hard disk capacity are available per node. The cluster runs Ubuntu Linux, version 14.04.

All benchmarked algorithms use Spark 1.5 on Yarn 2.7.1 and are implemented in Java 1.7. Master and worker nodes send serialized Java objects over the network. We implemented the algorithm from Section 4 and abbreviate it by MPQ (for massively parallel query optimization). We compare against an algorithm representing the fine-grained approaches to parallelizing query optimization proposed so far. They were targeted at shared-memory architectures and moderate degrees of parallelism [9, 10]. We call that algorithm SMA (for shared-memory approach). In this algorithm, the master node assigns to each worker a set of join results for which to find optimal plans using the optimal plans that were generated by other workers. This means that intermediate results need to be shared between workers and that the master needs to assign multiple rounds of tasks to the workers. For both algorithms, the master initially sends query-specific statistics (e.g., predicate selectivity values) to each worker. The comparison between MPQ and SMA is unfair as both were developed for different architectures. We are however unaware of other query optimization algorithms for shared-nothing architectures.

We use up to 256 Spark executors and reserve up to 40 GB of main memory per executor (query optimization requires large amounts of memory, in particular in case of multiple plan cost metrics [22]). We set the maximum message size to 1 GB (SMA needs to send large messages).

We compare algorithms in linear and bushy plan spaces. We do not heuristically restrict the use of cross products as this might miss optimal plans [16]. As we allow cross products, the number of intermediate results to consider and hence performance of our optimization algorithms does not critically depend on the structure of the query join graphs. We generate queries with equality predicates and star-shaped join graphs (unless noted otherwise). We choose table cardinalities and attribute domain sizes by the method introduced by Steinbrunn et al. [19] which is commonly used for query optimization benchmarks [2, 23, 24]. In a first series of experiments, we consider execution time as only cost metric and use standard cost formulas [19] to estimate the cost of standard join operators such as block-nested loop join, hash join, and sort-merge join. In a second series of experiments, we consider two plan cost metrics and the goal is hence to approximate the set of Pareto-optimal plans (a plan is Pareto-optimal if no other plan has better cost according to all cost metrics [22]). Our second cost metric (in addition to execution time) is the buffer space consumption. Those two cost metrics are frequently used for benchmarking multi-objective query optimization algorithms [22, 23].

For the series of experiments with two plan cost metrics, we replace the standard pruning function by a pruning function that was used in prior work for multi-objective query optimization with formal near-optimality guarantees [22, 23]. That pruning function is parameterized by an approximation factor α, we set $\alpha = 10$ unless noted otherwise.

6.2 Experimental Results

We show only an extract of our full experimental results. The presented results are however representative and we observed the same tendencies in additional experiments.

We start by discussing the results for traditional query optimization with one plan cost metric. Figure 1 shows a comparison between MPQ and SMA in terms of optimization time and in terms of the amount of data exchanged between cluster nodes. Each data point in the plots corresponds to the median of the results for twenty randomly generated queries. We compare algorithms for different plan spaces (linear and bushy) and for different query sizes (number of joined tables). We try different degrees of parallelism for each plan space, adapting the maximal parallelism to the search space size (we scaled up to the maximal degree of parallelism that MPQ can exploit based on the number of disjoint table pairs or triples) up to a maximum of 128 workers. We try smaller query sizes for the bushy plan space than for the linear plan space as the size of the bushy search space grows faster in the number of query tables. Note that we also consider Cartesian product joins in contrast to prior evaluations of parallel query optimization algorithms. This makes the plan space much larger for the same number of tables. Still the search spaces treated in Figure 1 are of moderate size and we try larger search spaces in the following.

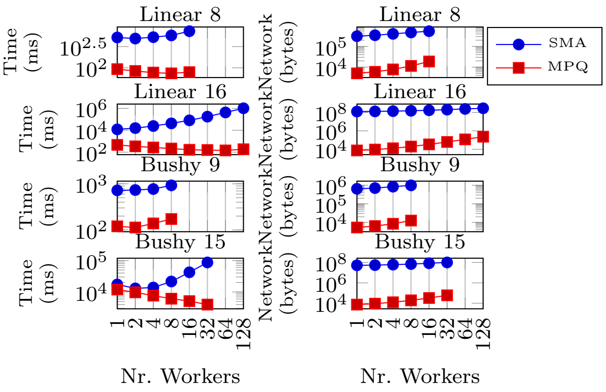

Figure 1: MPQ outperforms SMA by up to four orders of magnitude in terms of optimization time; scalability of MPQ is limited due to the query sizes.

MPQ outperforms SMA by up to four orders of magnitude in optimization time. The reason is the large amount of data that SMA has to send over the network, due to the need for sharing intermediate results between workers, and the overheads on the master node by fine-grained task management. The amount of data sent by SMA reaches several hundreds of megabytes while our algorithm sends at most 234 kilobytes and in most cases significantly less than that. SMA is not designed for shared-nothing scenarios and the performance gap between the algorithms is expected.

The search space sizes in Figure 1 represent approximately the limit of what the competitor algorithm SMA can treat within reasonable amounts of time. For our MPQ algorithm, the considered search spaces are actually too small to justify parallelization. This is why we see in most plots in Figure 1 no decrease in optimization time for MPQ with growing degree of parallelism. The network traffic and the management overhead increase for both algorithms in the number of workers. SMA can only benefit in few cases from parallelization and only up to a degree of parallelism of four.

The computation time of SMA increases quickly in the query size and in the degree of parallelism as well (reaching more than 15 minutes per test case for 16-table joins). This is why we exclude it from the following series of experiments.

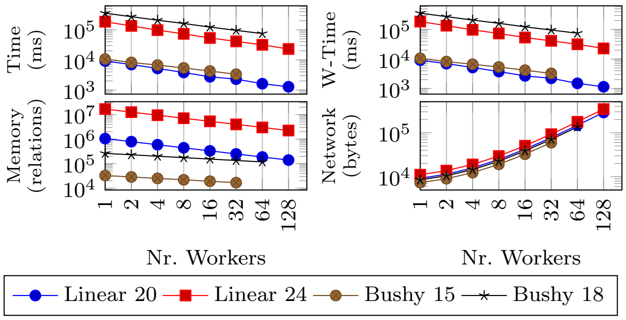

Figure 2: MPQ scales steadily for sufficiently large search spaces and one plan cost metric.

Figure 2 shows results for larger search spaces and only for MPQ. The figure shows total optimization time (measured on the master node) as well as the maximal optimization time measured over all workers ("W-Time" in the figure). The fact that the difference between both is small indicates that the management overhead on the master node is negligible. We show network traffic and additionally the maximal main memory consumption over all of the workers (the master does not perform optimization itself and its main memory consumption is negligible). We scale for each query size up to the maximal degree of parallelism supported by our algorithm (determined by the number of table pairs for linear plans and the number of table triples for bushy plans) and maximally up to 128 workers.

As search space sizes are large enough, we see steady scaling for all degrees of parallelism that are theoretically supported by our algorithm without diminishing returns for higher number of workers. The scaling is slightly better for linear plans than for bushy plans which matches precisely our theoretical predictions from Section 5 (execution time decreases by factor 3/4 for linear plans but only by factor 21/27 for bushy plans, each time that the degree of parallelism doubles). Unlike for SMA, the network traffic created by MPQ depends only marginally on the query size as no intermediate results have to be exchanged between workers or between workers and master. Only the query itself and the final plan generated by each worker are sent. The maximal main memory consumption on the workers (measured by the number of relations for which to store optimal plans) equally decreases steadily with increasing parallelization. Here the decrease for bushy plans is slower than for linear plans which again matches our theoretical results.

If we use one worker then MPQ is equivalent to the classical query optimization algorithms [25] as it treats the same table sets in the same order. Hence we compare the optimization time when executing our algorithm on a single worker (not measuring master computation time and communication overheads) to the optimization time of the parallel version (including master computation time and communication overheads) to obtain the speedup of our algorithm compared to serial query optimization. With 128 workers, we obtain for left-deep plans a speedup of 8.1 for 24 query tables and a speedup of 7.2 for 20 tables. With 32 workers we have a speedup of 3.2 for 15-table joins and bushy query plans and a speedup of 4.8 for 18-table joins and 64 workers.

All results presented so refer to queries with star-shaped join graphs generated according to the method by Steinbrunn et al. [19]. There are query optimization algorithms for which optimization time depends heavily on query properties such as join graph and predicate types. The two algorithms that we compare are however both based on a classical dynamic programming scheme that examines for a given query size always the same number of intermediate results, independently of other query properties. Figure 3 shows for instance optimization time (arithmetic averages and 95% confidence intervals) for different join graph structures. Corroborating theoretical guarantees, the impact of the join graph structure on optimization time is negligible.

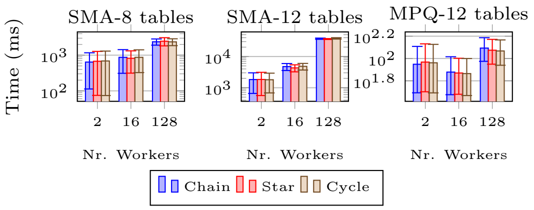

Figure 3: Query properties like the join graph structure have negligible impact on optimization time.

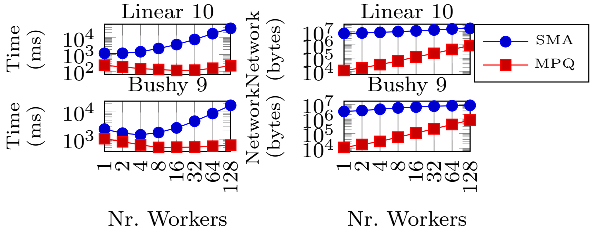

Figure 4: MPQ outperforms SMA but its scalability is limited by small query sizes.

Note finally that our Java-based implementation is not optimized for maximum efficiency. It is rather optimized for modularity, allowing to "plug-in" different search spaces and cost metrics. This enables us to execute experiments over a broad range of scenarios. We believe that optimization time can be reduced by specializing the algorithm.

We discuss the results for multi-objective query optimization. Figure 4 shows a comparison between multi-objective versions of SMA and MPQ (both algorithms use the same pruning function that we reconfigured to consider two cost metrics). The tendencies are similar as for single-objective query optimization. Optimization times and network traffic are significantly lower for MPQ than for SMA. The network traffic of MPQ has however increased when comparing to the results for single-objective query optimization. The reason is that each worker must now send the set of all Pareto-optimal plans in its respective plan space partition back to the master instead of only one plan. The median number of complete Pareto-optimal plans per query was 21 for 12-table joins when considering left-deep plans and 16 for 9-table joins in a bushy plan space.

Instead of exploiting a high degree of parallelism, SMA suffers significantly once the number of workers increases due to network traffic and coordination overhead. The maximal degree of parallelism that was beneficial to SMA is eight. This is also the number of threads that prior algorithms were maximally evaluated on. MPQ benefits from

Table 1: Minimal parallelism required to reach precision α within fixed optimization time budget.

Time (s)	Tables	Approximation Precision α						
		1.01	1.05	1.25	1.5	2	5	10
10	14	16	4	1	1	1	1	1
	16	∞	∞	64	64	32	16	8
	18	∞	∞	∞	∞	∞	∞	∞
	20	∞	∞	∞	∞	∞	∞	∞
30	14	1	1	1	1	1	1	1
	16	64	16	4	2	1	1	1
	18	∞	∞	128	128	64	32	32
	20	∞	∞	∞	∞	∞	∞	∞
60	14	1	1	1	1	1	1	1
	16	8	1	1	1	1	1	1
	18	∞	128	32	16	16	8	4
	20	∞	∞	∞	∞	∞	∞	128

parallelism up to 32 workers for 10-table joins and left-deep plans, for up to 64 workers for 12-table joins, and for up to eight workers for 9-table joins and bushy plan spaces which corresponds to the number of disjoint table pairs respective triples. The absolute run times of MPQ are however so low that parallelization is unnecessary.

Figure 5 shows results for MPQ on queries that are sufficiently large to exploit large degrees of parallelism. The scaling is steady and without noticeable diminishing returns effects up to the maximal number of 256 workers. Note that the run times of MPQ in Figure 5 are lower than the run times of SMA in Figure 4, even though we consider significantly larger search spaces in Figure 5. We tested scalability for bushy plans and more than 9 query tables and saw steady scaling up to the number of table triples in the query. We omit those results due to space restrictions.

Our algorithm is for one worker equivalent to a classical algorithm for multi-objective query optimization [22]. We calculate speedups in a similar way as before and obtain a speedup of 5.1 for 16-table joins, 5.5 for 18-table joins, and 9.4 for 20-table joins. We have seen that parallelization can decrease optimization time. Alternatively, parallelization can increase result quality for a given optimization time budget. Table 1 shows the degree of parallelism that is required to reach a certain approximation factor α within a given optimization time window for two cost metrics and linear plans (i.e., it shows the minimal degree of parallelism for which at least eight out of 15 test cases were solved). The entry ∞ indicates that the maximal degree of parallelism that we tried (128 workers) was insufficient. As in prior work [22], approximation quality improves as α approaches 1 and the algorithm guarantees to generate a plan with cost vector at most $\vec{c} \times \alpha$ if a plan with cost vector $\vec{c}$ is possible. Table 1 shows that a higher degree of parallelism results in higher approximation quality for a fixed time budget.

Increasing the degree of parallelism using our algorithm is not always helpful if optimization time on a single node is significantly below one second. Otherwise, in all our experiments, the lowest optimization time (respective highest quality) was obtained by choosing the highest degree of parallelism that can be exploited by our algorithm for a given query size (denoting by n the number of tables, this is $2^{n/2}$ for linear and $2^{n/3}$ for bushy plans).

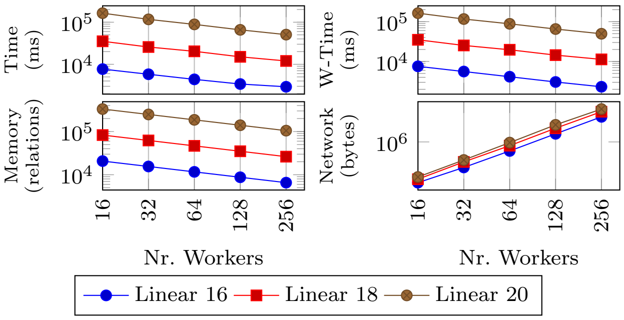

Figure 5: MPQ scales steadily using up to 256 workers for linear plan spaces and two plan cost metrics.

7. CONCLUSION

We presented a generic plan space decomposition method for query optimization that is applicable for single- and multi-objective query optimization and for other variants. We demonstrated scalability using up to 256 workers.

8. REFERENCES

[1] S. Agarwal, A. Iyer, and A. Panda. Blink and it's done: interactive queries on very large data. In *VLDB*, volume 5, pages 1902–1905, 2012.

[2] N. Bruno. Polynomial heuristics for query optimization. In *ICDE*, pages 589–600, 2010.

[3] S. Chatterji and S. Evani. On the complexity of approximate query optimization. In *PODS*, pages 282–292, 2002.

[4] C. Chekuri, W. Hasan, and R. Motwani. Scheduling Problems in Parallel Query Optimization. In *PODS*, pages 255–265, 1995.

[5] Y. Chen and C. Yin. Graceful Degradation for Top-Down Join Enumeration via similar sub-queries measure on Chip Multi-Processor. *Applied Mathematics and Information Sciences*, 941(3):935–941, 2012.

[6] O. Cure and G. Blin. *RDF Database Systems: Triples Storage and SPARQL Query Processing.* 2014.

[7] S. Ganguly. Design and analysis of parametric query optimization algorithms. In *VLDB*, pages 228–238, 1998.

[8] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.

[9] W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl. Parallelizing query optimization. In *VLDB*, pages 188–200, 2008.

[10] W.-S. Han and J. Lee. Dependency-aware reordering for parallelizing query optimization in multi-core CPUs. In *SIGMOD*, pages 45–58, 2009.

[11] A. Hulgeri and S. Sudarshan. AniPQO: almost non-intrusive parametric query optimization for nonlinear cost functions. In *VLDB*, pages 766–777, 2003.

[12] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In *SIGMOD*, pages 312–321, 1990.

[13] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric Query Optimization. *VLDBJ*, 6(2):132–151, may 1997.

[14] H. Kllapi, E. Sitaridi, M. M. Tsangaris, and Y. E. Ioannidis. Schedule Optimization for Data Processing Flows on the Cloud. In *SIGMOD*, 2011.

[15] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[16] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, pages 314–325, 1990.

[17] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[18] M. a. Soliman, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, R. Baldwin, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, and F. Rahman. Orca: A modular query optimizer architectur for big data. In *SIGMOD*, pages 337–348, 2014.

[19] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDBJ*, 6(3):191–208, 1997.

[20] M. Stonebraker. The Case for Shared Nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, 1986.

[21] A. Swami. Optimization of large join queries: combining heuristics and combinatorial techniques. *SIGMOD*, pages 367–376, 1989.

[22] I. Trummer and C. Koch. Approximation schemes for many-objective query optimization. In *SIGMOD*, pages 1299–1310, 2014.

[23] I. Trummer and C. Koch. An incremental anytime algorithm for multi-objective query optimization. In *SIGMOD*, pages 1941–1953, 2015.

[24] I. Trummer and C. Koch. Multi-objective parametric query optimization. *VLDB*, 8(3):221–232, 2015.

[25] B. Vance and D. Maier. Rapid Bushy Join-Order Optimization with Cartesian Products. *SIGMOD*, 1996.

[26] F. M. Waas and J. M. Hellerstein. Parallelizing extensible query optimizers. In *SIGMOD*, page 871, 2009.

[27] W. Zuo, Y. Chen, F. He, and K. Chen. Optimization Strategy of Top-Down Join Enumeration on Modern Multi-Core CPUs. *Journal of Computers*, 6(10):2004–2012, oct 2011.