Shared Workload Optimization

Georgios Giannikis Darko Makreshanski Gustavo Alonso Donald Kossmann

Systems Group, Department of Computer Science, ETH Zurich, Switzerland
{giannikg,darkoma,alonso,kossmann}@inf.ethz.ch

ABSTRACT

As a result of increases in both the query load and the data managed, as well as changes in hardware architecture (multicore), the last years have seen a shift from query-at-a-time approaches towards shared work (SW) systems where queries are executed in groups. Such groups share operators like scans and joins, leading to systems that process hundreds to thousands of queries in one go.

SW systems range from storage engines that use in-memory cooperative scans to more complex query processing engines that share joins over analytical and star schema queries. In all cases, they rely on either single query optimizers, predicate sharing, or on manually generated plans. In this paper we explore the problem of shared workload optimization (SWO) for SW systems. The challenge in doing so is that the optimization has to be done for the entire workload and that results in a class of stochastic knapsack with uncertain weights optimization, which can only be addressed with heuristics to achieve a reasonable runtime. In this paper we focus on hash joins and shared scans and present a first algorithm capable of optimizing the execution of entire workloads by deriving a global executing plan for all the queries in the system. We evaluate the optimizer over the TPC-W and the TPC-H benchmarks. The results prove the feasibility of this approach and demonstrate the performance gains that can be obtained from SW systems.

1. INTRODUCTION

The increasing and widespread use of data service is putting a strain on databases behind them. These data services need to support complex SQL queries for strategic decision making in industries as varied as travel reservation, financial, insurance or even social networking. With the number of internet users and web services increasing, these systems are faced with loads that often involve hundreds or thousands of queries submitted at the same time [26]. Conventional database engines deal with queries individually, trying to achieve the best performance for each query plan. With very high loads, these independent plans compete with each other for resources, causing a *load interaction* problem further aggravated by multi core architectures [21].

Proceedings of the VLDB Endowment, Vol. 7, No. 6

As a result, systems have started to appear where queries are processed in groups or batches, implementing different forms of cooperative and shared execution. Blink[19, 18] and Crescando [27] share scans at the storage engine level. QPipe[8], DataPath[1], SharedDB[6] and CJoin [2] add more complex operators and try to avoid repetitive work by sharing it across all queries. Instead of instantiating operators at runtime, most of these systems use a set of *always-on* operators in order to further reduce execution time and maximize sharing. Finally, MonetDB[29] implements sharing by caching intermediate results and reordering queries so that the cached results can be useful to more than one query.

These shared work (SW) systems are well established and quite sophisticated but lack a global optimizer. Some of them use conventional query-at-a-time optimization techniques, others rely on predicate sharing, and yet others use hand crafted plans. All of these SW systems would benefit from an optimizer capable of looking an entire workload and coming up with a shared execution plan that maximizes sharing.

In this paper we explore the problem of Shared Workload Optimization (SWO), and propose an algorithm that given a set of statements and their relative frequency in the workload, outputs a global plan over shared, always-on operators. Unlike what has been done until now, SWO concerns itself with entire workloads and the simultaneous optimization of all queries in each workload. SWO does not optimize queries individually and does not look for common predicates. Instead it focuses on running queries concurrently over a pool of shared operators and, consequently, must identify which operators to share and how to organize these operators into a global access plan.

Formally, identifying the shared operators and ordering them are not orthogonal decisions, which turns SWO into a bilinear optimization problem. As we will show in the paper, changing how many statements share the same operator affects the overall selectivity, which might require to reorder the operator. Additionally, the cost function is non-convex, as there are a lot of local optima. For instance, each independent query adds a local optimum, which is the cheapest way to execute it. The non-convex, bilinear nature of the problem renders exhaustive techniques, like brute-force and greedy optimization unsuitable. Exhaustive search will result in huge running times due to the enormous size of the solution space, while greedy optimization will most likely converge to a locally optimum solution. SWO is similar to the stochastic knapsack problem with uncertain weights [12], where the stochasticity comes from the fact that the cost or the weight of an item is variable and depends on all previous decisions.

In this paper we show how SWO can be tackled through a branch and bound optimization technique. To reduce the size of the solution space, we introduce two heuristics. The first one makes a

quick, yet effective, decision on how to share operators, while the second one gives guidelines on the ordering of these operators. We have used the resulting algorithm to generate global plans for the entire TPC-W and TPC-H benchmarks. The runtime of our algorithm is negligible, given that the generated query plan can be used for a long time, possibly the whole lifetime of the system. The experiments prove the feasibility of SWO and open up several interesting research directions for further extending the algorithm.

2. RELATED WORK & MOTIVATION

2.1 Query-at-a-time Optimization

The problem of optimizing a query plan is not a new one [17, 22]. Most of the early work focused on how to optimize individual query plans by trying to reduce the required processing power and I/O. This single query optimization problem boils down to exploring the whole solution space, which gets exponentially bigger as the number of operators involved increases. The solution space has a size of $O(\frac{(2N)!}{N!})$, where N is the number of involved relations. To make matters worse, this combinatorial problem is non-convex, meaning that there are a lot of local optima which, in most of the cases, are spread across the solution space.

Because the nature of the problem renders linear optimization unsuitable and exhaustive search comes with a high runtime, forward Dynamic Programming became the optimization technique of choice for single query plans [22]. In Dynamic Programming, *memoization* is used in order to avoid repeating work while exploring the solution space. The solution space is searched bottom up, by first examining all possible two way joins and then incrementally adding more relations. Several flavors of Dynamic Programming have been suggested depending on the nature and complexity of the queries [11]. Even though Dynamic Programming proved to be a successful option for single query optimization, it cannot be used in SWO because of the bilinear nature of the problem. An analysis of the bilinear nature of the problem is presented in Section 3.

2.2 Multi Query Optimization

Multi Query Optimization (MQO) was originally explored in [23]. The main idea is to identify common subexpressions across the set of running queries. Once common subexpressions are detected, the system can replace the original subqueries with a broader subquery that subsumes all of them or rearrange running queries such that common subexpressions are executed together.

The idea of MQO was extended and further improved in the Volcano optimizer [20]. This work uses materialized views to further benefit from commonalities across queries. Nevertheless, the additional overhead of maintaining the materialized views limits the applicability of this technique.

An alternative type of multiple query optimization that caches intermediate results has been suggested in [24] and further improved in [10, 15]. Caching (recycling) intermediate results of common operators (i.e. a very frequent join), has been shown to increase the performance as it comes with a number of advantages. Nonetheless, result caching cannot be applied to systems with high update loads, as the caches are invalidated very frequently. In order to support high update loads, the proposed algorithm does not rely on result caching at all.

Recent work by Zhou et al. [28] introduces a novel technique for detecting common subexpressions which allows MQO to be integrated in production systems. The experiments show that applying MQO across a few tens of queries is extremely efficient.

Although intuitively useful, when MQO is applied to SW systems, it limits the amount of queries that can share work. The goal of SW systems is to process hundreds to thousands of queries concurrently and we are not aware of any techniques that can detect common subexpressions for so many queries within a reasonable runtime. Additionally, subexpressions have to be detected every time a new set of queries arrive in the system, because the sharing of the previous set of queries is independent of the sharing of the next set of queries. Our approach to SWO avoids this limitations by not detecting common subexpressions at all. Instead we rely on identifying common operators across prepared statements. Moreover, SWO has to be executed only when the workload evolves (i.e. more prepared statements are added). This allows SWO to be applied not only to analytical workloads, but also to transactional workloads, where new queries arrive at a much faster rate. Our experiments on both TPC-H (analytical) and TPC-W (transactional) show that SWO is able to generate efficient plans in both cases. We are not aware of any work on MQO that is able to process the entire TPC-H benchmark, let alone TPC-W which has shorter running queries where MQO is not effective.

2.3 Shared Work Systems

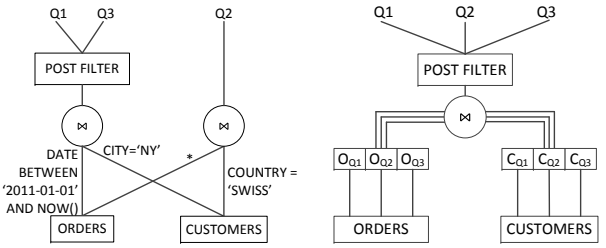
In the last years, there is an increasing shift towards database systems and database operators that process multiple queries at a time by sharing work (SW Systems). For instance, DB2 UDB [13], uses a cooperative scan operator where each table scan answers more than one query at the same time, independently of their selection predicates. Similar ideas have been implemented in MonetDB/X100 [29] and the Blink system [19, 18]. Crescando [27] allows thousands of queries to share the same scan, while maintaining predictable performance by indexing the running queries.

Additionally, several systems implement relational operators capable of processing multiple queries at a time. Datapath [1], CJoin [2, 3] and SharedDB [6] are just a few of them. The common idea behind these systems is to share operators across queries. For instance, QPipe [8] creates dynamic pipelines of always on operators. As queries arrive in the system, they are attached to the active set of queries and are evaluated. Experimental results on these systems show that they outperform query-at-a-time systems, while providing better response time guarantees.

Even though these SW systems are quite sophisticated, they all currently rely on either single query optimization techniques, or on MQO, thus not taking full advantage of their capabilities. For instance, DataPath and QPipe rely on predicate sharing and MQO, a decision that limits their scalability to amount of sharing possible among concurrent queries. Because these systems rely on temporal overlap for enacting sharing, they are more suitable for analytical workloads where some queries take long enough to allow sharing with other queries. SharedDB and CJoin do not use MQO to avoid this limitation but they do not have an optimizer and rely on hand tuned query plans. A different type of optimization is hinted in [6] in order to automate hand tuning of query plans. In this two step optimization, single query optimization is used on each query and then the resulting access plans are overlapped. This reduces sharing opportunities as executing each query in the best locally way, might create a suboptimal global plan.

To tackle these shortcomings, our query optimization algorithm is designed specifically for these systems. The algorithm takes as an input the whole workload and produces an execution plan that minimizes the total amount of work necessary to evaluate it.

Finally, some of these systems impose additional constraints on the generated pipelines [4]. For instance, two complementary sort-merge join operator can cause a deadlock if their pipelines are used concurrently. These constraints can be safely integrated in our optimizer. The work of Dalvi et al. presents all the required methods



(a) MQO Generated Plan **(b)** SWO Generated Plan

Figure 1: Difference between MQO and SWO

for detecting and resolving a deadlock in such pipelines. Other systems, like CJoin and SharedDB, avoid deadlocks by explicitly not using operators processing two or more streams concurrently.

We will compare the effectiveness of the generated plans by running them on SharedDB [6]. Nevertheless, the algorithm is not specific to SharedDB and can be applied to other SW systems as well. SharedDB executes multiple queries by multiplexing their result tuples in shared result streams and driving them through a network of always on operators. The whole data-flow architecture operates in a push-based manner, where the lowest level operators push results to the highest level operators. Each shared operator is able to process possibly thousands of queries concurrently, and in fact, sharing more queries reduces the cumulative required work.

2.4 MQO vs SWO

To illustrate the differences between SWO and MQO, consider the following three queries:

```
Q1: SELECT * FROM ORDERS NATURAL JOIN CUSTOMERS
WHERE ORDERS.DATE BETWEEN '2012-01-01' AND NOW()
AND CUSTOMERS.CITY = 'NEW YORK';

Q2: SELECT * FROM ORDERS NATURAL JOIN CUSTOMERS
WHERE CUSTOMERS.COUNTRY = 'SWITZERLAND';

Q3: SELECT * FROM ORDERS NATURAL JOIN CUSTOMERS
WHERE ORDERS.DATE BETWEEN '2011-01-01' AND '2012-12-31'
AND CUSTOMERS.CITY = 'NEW YORK';
```

Implementing these three queries using MQO results in the execution plan of Figure 1a. The expressions of `Q1` and `Q3` are rewritten into a broader expression that selects tuples for both queries. As a result, the cost of accessing `ORDERS` is shared across these two queries. Post-filtering is required in order to separate tuples that answer each query. Executing this plan on a SW system is not optimal. First of all, the plan generation has to be repeated for every batch of queries. Also, the performance of the system depends on the query parameters; if there is no overlap across query predicates, broader subexpressions end up being a long disjunction of predicates. Finally, subexpression detection becomes expensive if hundreds of queries have to be taken into account. CJoin for instance has been shown to easily implement sharing across hundreds of concurrent queries and SharedDB case share work across thousands of queries while both systems maintain stable performance.

A plan that works better for most SW systems is shown in Figure 1b. In this case, a single shared join operator processes all three queries at the same time, regardless of commonalities. The predicates are pushed to the storage engines that execute all of them at the same time, while tagging tuples with the IDs of each query. There is no need to identify common subexpressions and, as a result, the plan optimizer does not have to examine the query parameters. This allows to create a global plan for all prepared statements of a workload. For instance, if `ORDERS` has to be accessed using full table scans, the same scan will answer all three queries. The lack of commonalities allows this plan to be used by any query that asks for `ORDERS` ⋈ `CUSTOMERS`. Finally, post filtering in this case is much faster. Instead of executing part of the original query, the filter selects tuples by the tagged query IDs and distributes them to the correct consumers.

Sharing as much as possible is not always optimal. To illustrate this consider three queries over two tables with these selections:

	Query 1	Query 2	Query 3
Customers	Select 10 Tuples	Fetch all 1,000 Tuples	Select 20 Tuples
Orders	Fetch all 10,000 Tuples	Select 100 Tuples	Fetch all 10,000 Tuples

Sharing the Customers ⋈ Orders operation across all queries requires building a hash table of 1,000 Customer tuples and probing it with 10,000 Order tuples. In other words, it requires computing the full join. During the build phase, there are a total of 1,030 tuples. The overlapping 30 tuples are processed only once. Similarly, for the probe phase, the processing of the 100 overlapping tuples is shared. The total number of shared tuples for this plan is 130.

An alternative plan is to share the join across `Q1` and `Q3` only. In this case, less tuples are shared. Yet, this plan may require less amount of work. The reason is that building a hash table is not as cheap as probing it. This plan requires building two hash tables of $(10+20)$ and 100 tuples respectively and probing them with 10,000 and 1,000 tuples. For most SW systems, $build(130)+probe(11K)$ is less expensive than $build(1K)+probe(10K)$. Also, by sharing the join across `Q1` and `Q3`, the probe phase of 10,000 tuples is shared and as a result all tuples are processed exactly once.

3. PROBLEM DEFINITION

The problem we are solving in this paper is query optimization by work sharing, where we search for the globally optimal plan to execute a whole workload. There are two dimensions in this problem: a. ordering of operators and b. sharing of operators. Unfortunately, these dimensions are not orthogonal. For instance, a decision to share a join operator across two queries, means that the selectivity of this operator will be affected and as a result, we should reorder the operators to achieve a better plan.

The ordering part of the problem is similar to single query optimization, a problem that has been extensively studied [17]. Given a query that involves n operators, the optimizer has to make n decisions on which order the operators should be executed. For every join operator, the optimizer has to additionally decide on which one is the inner relation and which one is the outer relation. This creates a solution space of $O(\frac{(2n)!}{n!})$ solutions. We can formulate this as a mixed integer optimization problem using a two dimensional matrix sel to store which operator o was selected on each step s, and a two dimensional matrix $cost_{s,o}$ that contains the cost of selecting operator o at step s. The problem of join ordering can be mathematically formulated by:

`Minimize:`

$$\sum_{s=1}^{n}\sum_{o=1}^{n} sel_{s,o} * cost_{s,o}$$

`under the constraints:`

$$\sum_{o=1}^{n} sel_{s,o} = 1, \forall s \in [1,n] \quad (1) \qquad \sum_{s=1}^{n} sel_{s,o} = 1, \forall o \in [1,n] \quad (2)$$

Here constraint 1 enforces that we choose only one operator on each step, while constraint 2 enforces that we eventually select all the required operators. This formulization of the problem assumes that there is only one possible implementation for each operator. In most systems there are different flavors of i.e. table access (scans or indexes), or joins (sort-merge and hash join). Adding all the possible flavors of operators in the formulization would only increase the dimension of the operators, without increasing the required steps. Nevertheless, the subproblems are overlapping and independent of

each other, so dynamic programming can be applied. The key idea is to isolate subproblems and find the optimal way to execute each subproblem, i.e. find the best way to join R relations. Then incrementally build on top of these subproblems.

Even with dynamic programming, a total of $O(3^n)$ joins have to be evaluated [16]. As a result, for large values of n the problem of join ordering requires a lot of processing and other techniques should be applied to simplify it. For instance, [11] suggests that for n greater than 10 (i.e. a ten-way join), heuristics should be used.

Adding operator sharing on top of ordering introduces a third dimension. The problem can be defined as:

`Given:` $\mathbb{Q}$ queries that involve a set of operators, $\mathbb{O}_{1...|\mathbb{Q}|}$,

`Find:` A global access plan GP that uses a set of operators selected from all the $\mathbb{O}_{1...|\mathbb{Q}|}$ sets, such that:

- all queries from $\mathbb{Q}$ can be answered, and
- the cost of GP is minimal.

This definition assumes that if multiple queries involve the same operator, then this operator can be chosen only once and serve all queries at the same time. The third dimension contains the decision on which queries share which operators and dramatically increases the decision space of our problem. Additionally, the number of required steps is not fixed. Sharing an operator across s queries means that we require $s-1$ fewer decisions. This three dimensional multi query optimization problem can be formulated as:

`Minimize:`
$$\sum_{s=1}^{S}\sum_{o=1}^{|\mathbb{O}|}\sum_{q=1}^{|\mathbb{Q}|} sel_{s,o,q} * cost(s,o,q)$$

`under the constraints:`
$$\sum_{o=1}^{|\mathbb{O}|}\sum_{q=1}^{|\mathbb{Q}|} \max(sel_{s,o,q}) = 1, \forall s \in [1,S] \quad (3)$$

$$\sum_{s=1}^{S}\sum_{q=1}^{|\mathbb{Q}|} \max(sel_{s,o,q}) = \mathbb{O}, \forall o \in [1,|\mathbb{O}|] \quad (4)$$

$$\sum_{s=1}^{S}\sum_{o=1}^{|\mathbb{O}|} \max(sel_{s,o,q}) = |\mathbb{O}_q|, \forall q \in [1,|\mathbb{O}_q|] \quad (5)$$

In the formulas, s is the current step, o denotes the operator, and q the query. The maximum number of required steps, S, is $\sum_{q=1}^{|\mathbb{Q}|} |\mathbb{O}_q|$, where $\mathbb{O}_q$ is the set of operators required by the qth query. The maximum number of operators, $|\mathbb{O}|$, is also equal to S. In this case, no operator could be shared and as a result we have to use all operators of all queries.

Constraint 3 enforces that we choose only one operator on each step. This operator might be shared across all or some queries, which introduces the $max()$ function. Constraint 4 enforces that we eventually choose all operators. Finally, constraint 5 enforces that all queries are answered. This mathematical formalization lacks one more constraint which cannot be described mathematically and limits the sharing of operators. In short, an operator cannot be shared across a set of queries unless all suboperators are also shared across the same set of queries. An analysis of this constraint will be given in Section 4.1.

Last but not least, the cost of subproblems is not fixed. In the two dimensional problem, the cost of selecting a join depends on the join and the previous decision (subproblems are independent). In the SWO problem, the cost of a decision depends on the join and on how many queries are sharing this join and all the underlying joins. In other words, the best way to join R relations depends on how many queries are sharing these $R-1$ joins. More interestingly, the cost changes if one of these joins is shared across some other query (that does not require all R relations). As mentioned before, the problem resembles a knapsack problem with stochastic weights.

Complexity Analysis

Adding sharing on top of ordering, a problem that was already NP-hard [9], heavily affects the complexity. A work sharing optimizer has to decide which operator should be executed next, as well as which combination of queries will use it. For every operator, a new dimension of $O(2^{ds-1})$ choices is introduced, where ds represents the degree of sharing, the number of queries that ask for the same operator.

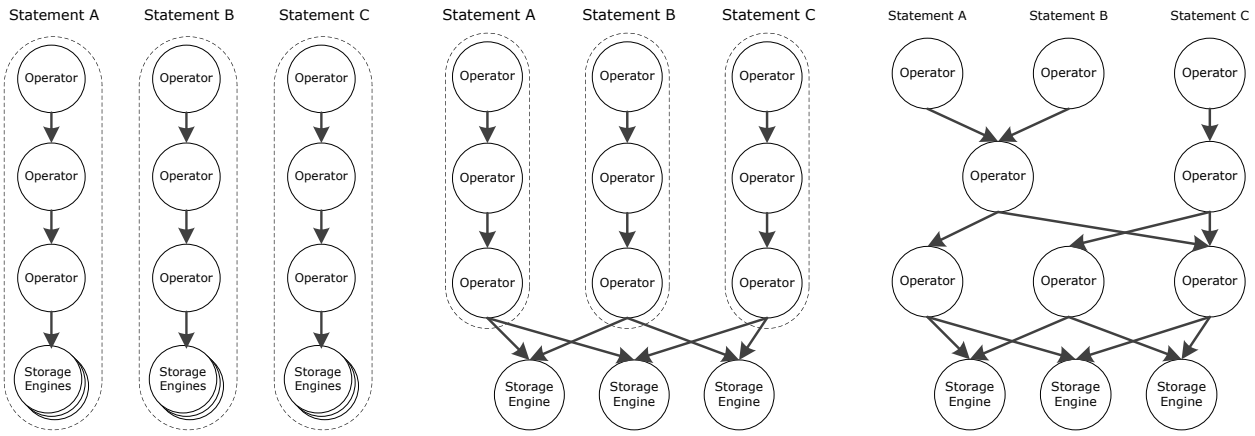
To make matters worse, in SWO we have to consider all operators involved by all queries. For instance, if a workload contains q queries, each of which involving the set of operators $\mathbb{O}_q$, the SWO optimizer will have to consider the union of all $\mathbb{O}_q$ operators. As a result the solution space of the presented problem has a size of $O(2^{ds-1} * \frac{(2*|\mathbb{O}|)!}{(|\mathbb{O}|)!})$, where $\mathbb{O} = \mathbb{O}_1 \cup \mathbb{O}_2 \cup \cdots \cup \mathbb{O}_q$. The problem can be classified as a non-convex, bilinear optimization problem, where the non-convex nature comes from the fact that there are a lot of local minima and the bilinearity comes from the fact that subproblems are not independent.

In order to solve this problem we considered a number of optimization methods. Exhaustive methods, like brute force or backtracking could not be used, due to the huge dimensions of the solution space. Additionally, greedy optimization methods, as well as convex optimization methods are useless. These methods would most likely converge to a local minimum. Furthermore, the dimensions of the problem do not have optimal substructure properties. Solving the ordering problem independently of the sharing problem will result on suboptimal solutions.

A class of optimization methods that is able to solve such problems is stochastic optimization, where generated random variables are applied into the objective function and the best encountered solution is chosen. Recent stochastic methods like Stochastic Optimization using Markov chains [14] provide some guarantees on the optimality of the solution by generating random variables in a smarter way. Yet, the runtime of such methods is quite high which make them not a good solution for this kind of problem.

Next we explored combinatorial optimization methods. Dynamic programming, a method successfully used in single query optimization is not suitable for this problem. In SWO, subproblems are not independent of each other as explained before. As a result, memoization is useless, thus dynamic programming cannot be applied. Branch and bound (B&B), an optimization method used to solve a wide variety of combinatorial problems, fits the constraints of our problem. B&B systematically enumerates a tree of candidate solutions, based on a bounded cost function. However, the huge number of variables of our problem means that branch and bound will need to examine on average $O((|\mathbb{O}| * ds)^{\frac{|\mathbb{O}|}{ds}})$ candidate solutions [25, 5]. In this case $|\mathbb{O}| * ds$ is the fan out of the solution tree which is the number of possible solutions to branch, and $\frac{\mathbb{O}}{ds}$ is the average depth of the tree. In order to further reduce the complexity of the problem, heuristics have to be applied.

The algorithm we present in this paper uses branch and bound with heuristics in order to find a globally optimized solution. The heuristics used are presented in Section 4.1. In order to simplify the problem, we will consider queries that only have two operators: hash joins and scans. This does not reduce the effectiveness or applicability of our algorithm, as it can be easily adapted to support more join methods and other operators, like sort and group by. We decided to work with joins, as these are more sensitive in terms of performance when it comes to ordering as well as sharing.



(a) No Sharing across Statements

(b) Share only Storage Engines across Statements

(c) Share all common Operators and Storage Engines across Statements

Figure 2: Different Approaches to Share Operators across Query Statements

4. WORK SHARING

The goal of our optimizer is to find an efficient global query plan for all input queries. As already mentioned, this requires two tasks: a. decide on whether to share a database operator across queries and b. decide on the ordering of the shared and not shared operators. What makes the problem interesting is that these two tasks interact with each other.

The task of ordering operators has been extensively studied [17, 22]. In this section we will study how operators can be shared and what are the limitations of sharing. We will first analyze the possible sharing strategies and then we will present two heuristics that are based on experimental data and simplify the problems of sharing and ordering operators.

4.1 Sharing Operators

As already mentioned in Section 2.3, sharing an operator across different statements reduces the amount of work required to execute the workload by not repeating the same work multiple times. However, sharing everything is not always optimal. For instance, sharing the execution of a full table join with a much smaller join means that the second statement will have to post filter more tuples on the higher levels of the query plan, as the result stream will contain tuples from both queries.

Based on the constraints of different SW systems, we can enumerate three different types of sharing:

Share Nothing: An example of a share nothing query plan is shown in Figure 2a. In this approach no work is shared across statements. Queries that are created from the same prepared statement will still share work with each other, however, scans, joins and other operations that are common with queries from other statements will be executed multiple times. Finding the globally optimal plan for this case is easier since traditional (single) query optimization techniques can be applied. Share Nothing is very common in systems that make use of heterogeneous replication. In this case, multiple replicas of the dataset and the query processing engine are present. Each replica is specialized in executing a different prepared statement. Thus, replicas that answer point queries will incorporate the appropriate indexes, while replicas that are dedicated to queries involving full table scans will omit these indexes and as a result avoid the cost of maintaining them. Another example of a share nothing approach is materialized views.

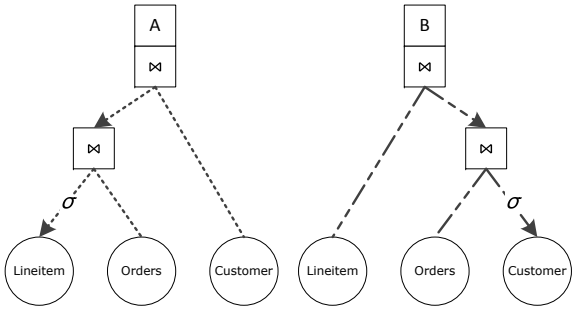
Nevertheless, a plan that shares nothing does not utilize the full potential of SW systems, as there are no sharing possibilities across statements. To make matters worse, share nothing implies that all storage engines have to be replicated for every statement that requires them. This requires additional locking and synchronization to ensure consistency which adds a considerable overhead in case of updates and makes the global query plan not scalable to the number of statements. Even though global query plans that share nothing are viable, none of the studied SW systems support them, thus our optimizer is not considering them.

Share Storage Engines: This approach overcomes the issue of storage engine replication by sharing all the storage engines in the system with all statements. An example of such a plan can be seen in Figure 2b. Database operators are still not shared across statements, while all storage engines are shared. Finding a globally optimal plan is still possible using traditional single query optimization techniques. Each statement has its own pipeline of operators that only interacts with other pipelines in the lowest level, making it possible to reorder operators within each pipeline.

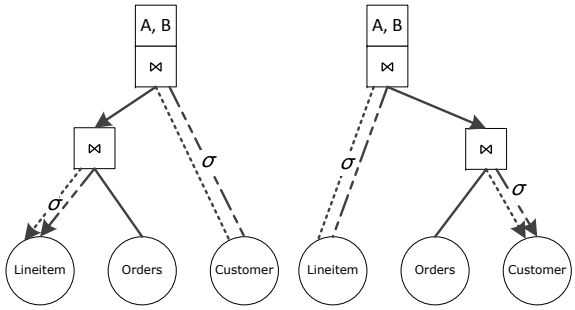
This sharing model is currently employed by a couple of research prototypes, like DataPath [1] and StagedDB [7]. The big advantage of this model is that any similar operation on the storage layer, which typically is the slowest one, is not repeated. For instance, a full table scan operator can run once in order to execute as many scan queries are currently pending. This is the main reason why shared (cooperative) scans have been established and implemented in a number of systems, like Blink [19, 18] and MonetDB [29].

Such query plans are considered in the proposed optimizer even though the amount of shared work is minimal. The optimizer considers building such disconnected pipelines of operators in cases where the statements are very diverse. For instance, consider a workload of two prepared statements, one of which is analytical and long running while the other is a transactional, short running query. Sharing, e.g., a join across these two statements means that the performance of the short running query will be dominated by the shared join which will be slowed down by the analytical statement. The overall cost of executing these two queries will also be greater, as filtering will be required in the smaller access plan in order to isolate the relevant small set of results out of the much larger stream of results.

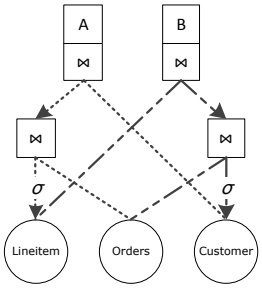
Share Everything: The last type of sharing is to share as much as possible across statements. Such a plan can be seen in Figure 2c. In this case, each operation that is common across all statements is shared. A number of recent research prototypes fit this model, like CJoin[2] and SharedDB[6]. The Share Everything model achieves



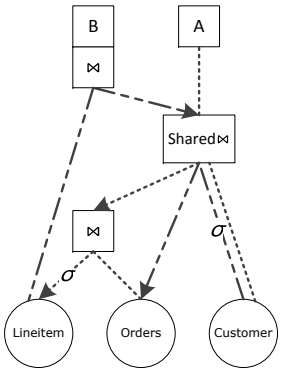
(a) Locally Optimal Plans for Queries of Section 4.1



(b) Two Possible Shared Plans for the Same Queries



(c) Share Storage Engines



(d) An Illegal Sharing of a Join Operator (Source Relations are Different)

Figure 3: Different Approaches to Share Two Query Statements. Arrow lines are build relations, plain lines are probe relations.

maximum sharing which means that work is never repeated. Practically, this approach reduces the total processing power required to execute a given workload. Nevertheless, if very diverse statements are involved, the pipeline of execution gets slower for the whole workload, and as a result shorter pipelines spend more time waiting for the longer pipelines to finish, rather than actually processing tuples. As a result, in such cases the short queries are penalized and the slowest query dominates the performance of the system.

While share everything seems optimal, it adds constraints on the ordering of operators. Shared operators should always have the same sub-operators for all the participating statements. To make this clearer, consider these queries:

```
A: SELECT * FROM LINEITEM JOIN ORDERS JOIN CUSTOMERS
WHERE LINEITEM.ITEM_ID = '?'

B: SELECT * FROM LINEITEM JOIN ORDERS JOIN CUSTOMERS
WHERE CUSTOMERS.ID = '?'
```

The locally optimal plans for these two queries can be seen in Figure 3a. Figure 3c illustrates a global plan that shares only the storage engines. The joins are processed independently and any work that is common, like hashing and probing of tuples as well as materialization, is performed two times. Additionally, Figure 3b shows two ways in which these statements can be executed using shared operators. Finally, the plan of Figure 3d is not legal because the sources of the shared operator are not the same. The shared operator is asked to build a hash table with tuples that are of type `LINEITEM ⋈ ORDERS` and of type `ORDERS` at the same time. Even though such a join operator could be implemented, it will require interpretation of every tuple, probably by means of a `switch` statement, in order to distinguish the tuple format. Adding a `switch` in the innermost loop of a hash join, adds considerable overhead, as well as a data dependency in the code. Thus the inner most loop cannot be fully optimized and branch miss-prediction would add a penalty on every tuple. To our knowledge, this is an implementation decision chosen by all SW systems.

In fact, our algorithm considers mostly hybrid approaches, where the operators that serve similar statements are shared, while operators that serve diverse statements are not shared. The optimizer can also take into account the probability (or the expected probability) of a prepared statement appearing in the workload and decides on whether to share an operator or not. This is not unreasonable even in real-life workloads, as nowadays, most database systems are behind web-services that have well known access patterns. The probability is then used to calculate the expected selectivity at any operator. For instance, in the example of Figure 3, if Query `B` appears with a probability of 0.05 and Query `A` with a probability of 0.95, then the left plan of Figure 3b would be more efficient.

4.2 Sharing Heuristic

In order to reduce the solution space of the multi query optimization problem, we introduce a heuristic that simplifies the sharing decision. The heuristic provides the optimization algorithm with the following two hints: a. which operators *should not* be shared, and b. which operators *can* be shared across which statements in the global access plan.

The first hint can be easily computed, provided that approximate statistics about the sizes of relations and the cardinalities of all involved attributes exist. The usefulness of this hint is easy to understand: the generated plan should avoid sharing operators across low selectivity and high selectivity subqueries at the same time, as already explained. The second hint requires analysis of each statement. The goal is to find what are the common operators across all statements. Even though this is something that can be computed by the algorithm at runtime, it is better to make this decision at a very early point. The reason is that this information is reused many times during optimization and it is better to precompute it. Additionally, it allows the optimizer to explore first the more popular operators. In contrast to the first hint, the second hint should only provide a guideline that can be ignored by the optimizer. For instance, the example of Figure 3d is one of the cases where the *can be shared* hint should be ignored.

The sharing heuristic analyzes all operators involved in all statements individually. First, it classifies all storage engine access for each query. We distinguish two classes: `H`igh Selectivity and `L`ow Selectivity. Subqueries that ask for the whole relation or request for big ranges of the relation are classified as `L`, while point queries or subqueries that ask for smaller ranges are classified as `H`. The assignment of a subquery to class `H` or `L` depends on the underlying implementation of the SW System. If the storage engine uses B-Tree lookups to access tuples, then the cost of retrieving all tree nodes should be taken into account. If the storage engine is implemented using full table scans, like for instance in SharedDB, then

the cost of retrieving 1 tuple or 100 tuples is very similar. In any case, properly identifying the threshold between L and H requires careful microbenchmarking of the specific SW System.

In the example of Figure 3, Lineitem access is H for query A and L for query B. Then, all higher level operators are classified accordingly. This means that three different classes for hash join operators are taken into account: H-H, L-L and L-H. The symmetrical class of the latter, the H-L (i.e. build on a big set of tuples and probe on a smaller) is not taken into account as it is always suboptimal. In this case, we always swap the build side with the probe side. The classification of joins always takes into consideration what can be filtered. For example, in query A of Figure 3, the Customers $\bowtie$ Orders is classified as L-H, as there is the potential to filter the Orders relation. In order to extend the heuristic to other flavors of join operators, we have to take into account the properties of the join. For instance, sort merge joins should only allow L-L or H-H classifications, as sort joins perform better if the sizes of the joined relations are similar.

The classification of operators alone is enough to provide a rough guideline of when operators should be shared or not. Operators that belong to different classes should never be shared, as this will result in too much unnecessary work for some queries. For instance, consider the left plan of Figure 3b. In this case Orders $\bowtie$ Lineitem is L-H for Query A and L-L for Query B. Our heuristic suggests that this join should not be shared across these queries. Instead, the operator has to be replicated and one instance should be used exclusively for Query A, while the second one should be dedicated to Query B, as show in Figure 3c.

In most cases, the sharing heuristic manages to reduce the solution space by an important factor. To quantify, consider a popular join that is asked by ds different prepared statements. An exhaustive search algorithm will have to make the decision on which of these ds statements should share the operator. This creates a combinatorial problem with a total of $2^{ds-1} - 1$ combinations (the combination that no operator shares the join is void). The Sharing Heuristic groups queries into 3 different groups. Queries that belong to different groups *should not* share the same operator, which reduces the number of combinations to $2^{\frac{ds}{3}-1} - 1$, assuming the average case of uniform distribution.

Even though this heuristic seems quite abrupt, it always hints towards the optimal solution. In Section 6.1 we present experimental results on the effectiveness of this heuristic.

4.3 Ordering Heuristic

In addition to the sharing heuristic, our multi query optimizer uses a second heuristic, the ordering heuristic, which provides the optimizer with a hint on how to explore the solution space. Based on the classification of the Sharing Heuristic, we are able to make a draft decision on the ordering of operators in the global query plan.

The heuristic exploits the observation that if a sequence of join operations is requested, executing the smaller joins (in terms of cost and number of tuples) first results in faster runtime. The benefits come mainly from the fact that using smaller sets of tuples first, we essentially perform more filtering at the early stages. This means that fewer tuples have to be joined and materialized.

In order to faster evaluate the heuristic, we reuse the classification of the Sharing Heuristic. Based on it, the optimizer will explore first the plans that have H-H hash joins on the lower level, L-H joins on the higher and the L-L join on the top level of the query plan. This means that before executing any full table join (L-L), we have filtered as much as possible the inner and outer relations.

The Ordering Heuristic is not reducing the solution space by omitting sub-optimal decisions, but instead gives a greedy direction on how to start exploring the solution space. The worst case is that all solutions need to be explored. Nonetheless, our experimental results show that the Ordering Heuristic helps in converging towards the best solution faster.

5. SWO ALGORITHM

In this section we present the shared workload optimization algorithm. The goal of the algorithm is to produce a globally efficient shared access plan, given a whole workload. In this context, globally dictates that we are interested in the cost of executing the whole workload, rather than each query independently.

Shared Workload Optimization is a non-convex bilinear global optimization problem. The objective (cost) function contains a lot of local minima. For instance, the best way to execute a single query is a local minimum. Additionally, subproblems are not independent of each other. The decision to share a hash join across two queries affects the cost of both queries. Consider a join across two relations, A and B, and two queries. Q1 asks for a join of the set of tuples $\sigma_1(A)$ with the set of tuples $\sigma_1(B)$, while Q2 asks for the join of $\sigma_2(A)$ with $\sigma_2(B)$. If the operator is shared, we have to calculate the cost of joining $\sigma_1(A) \cup \sigma_2(A)$ with $\sigma_1(B) \cup \sigma_2(B)$. Because of the union operator, if the selection are the same for both queries, then the shared join will have exactly the same cost as the not shared joins. Reordering join operators also affects the overall cost and may require to un-share certain operators, as already explained in Section 4.1.

The optimizer algorithm is based on the branch and bound optimization method. Branch and bound is used by most optimization solvers due to its effectiveness, especially if the type of optimization is discrete [12]. In branch and bound the solution space is greedily explored under the assumption that there is a theoretical bound in the solution. The theoretical bound is usually the solution to the same problem with relaxed constraints. Using this bound as a guide, the method proceeds by calculating the cost of intermediate illegal solutions (i.e. solutions that violate some of the constraints) and modifying the variables until a valid solution is reached.

Before presenting the algorithm, we will define the objective function to be optimized, as well as identify the bounded problem and the algorithm to estimate the cost of bounded solutions.

5.1 The Objective Function

The cost or objective function quantifies how good a solution is over other solutions. In most existing single query optimizers, the objective function consists of two variables: the number of I/O operations required and the CPU cost. These two metrics were sufficient for single query systems, where the queries had to be executed as fast as possible.

On shared work systems these metrics are not applicable as multiple queries are executed concurrently. A shared full table scan for instance, requires a lot of I/O operations, but now it serves a lot of queries. Adding another query in the load will not increase the number of I/O operations. The same applies to all shared operators in all SW Systems that implement them. What is important in SW systems is the number of tuples that have to be processed. Since some of these systems share resources across queries, the cost function is not linear. For example, if a query in the mix requests a full table scan of n tuples while another one requests a single tuple (point query), the total number of tuples fetched is going to be n instead of $n + 1$.

As a result, the objective function of our optimizer is based only on tuple count. The number of tuples is then multiplied by a factor that is different for each operator. For example, full table scans have a factor of 1 per tuple, while key value lookups have a factor that is equal to the number of tuples that fit in a (memory or

disk) page. Hash joins have different factors for the build and probe phase. The build phase has a factor of 4, as in our implementation four memory reads and writes are required to read and insert a tuple in the hash table. On the other hand, the probe phase has a factor of 1. These factors originate mainly from experimental micro benchmarks, which is part of micro tuning the objective function for a particular SW System. Applying our algorithm to different SW Systems, requires proper analysis of each individual operator.

Finally, the goal is to minimize the objective function. The optimal solution should have the minimum number of processed tuples, while answering the whole given workload.

5.2 Identifying the Bound

As already explained before, the bounded solution of the B&B method is the optimal solution for a relaxed problem, thus an invalid solution for the actual problem. The bounded solution should have a fast runtime as it is used only as an approximation of how good each step is. Of course, choosing a very relaxed problem increases the number of iterations required, as most of the intermediate steps will be very suboptimal compared to the approximation.

The bounded problem that we used in our optimizer originates from the same optimization problem with the relaxed constraint that every operator can and will be shared across all queries. A solution to the bounded problem violates the ordering constraint that was described in Section 4.1, but removes the non-linearity of the original problem. Essentially, the bounded problem is the overlapping of the access plans of all the queries, with no limitations. For instance, consider a workload with three queries, `Q1` that requires two joins, `A` $\bowtie$ `B` and `A` $\bowtie$ `C`, `Q2` that requires `A` $\bowtie$ `B` and `Q3` that requires `A` $\bowtie$ `C`. The relaxed problem allows both `A` $\bowtie$ `B` and `A` $\bowtie$ `C` to be shared. In contrast, the original SWO problem would only allow one of these joins to be shared. Sharing `A` $\bowtie$ `B` means that we cannot share `A` $\bowtie$ `C`, because `Q1` is now interested in the join `(A` $\bowtie$ `B)` $\bowtie$ `C`. Additionally, in order to reduce the complexity of the problem, we approximate that an operator that has to execute a set of n queries $\{Q_1, Q_2, ..., Q_n\}$ that ask for $T_1, T_2, ..., T_n$ tuples respectively, has a cost that is equal to $max(T_1, T_2, ..., T_n)$.

The relaxed problem is a combinatorial optimization problem that can be solved with traditional single query optimization techniques, like dynamic programming. Memoization can be applied as the subproblems and decisions are independent of each other. As we explore the solution space of the relaxed problem, we keep the cost of every subproblem. For instance, in a workload that requires 5 hash joins, we will keep the minimum cost of all two-way, three-way and four-way joins. This will help us bound the solution in the branching part of the algorithm. As the B&B visits intermediate (invalid) solutions, i.e. solutions that validate some of the constraints, it can estimate how good this intermediate solution is. The cost of every intermediate solution is equal to the actual cost of the part of the solution that is valid, plus the theoretically limited cost of the invalid solution.

5.3 Branch & Bound Algorithm

In this section we present the proposed multi query optimization algorithm that is based on the Branch and Bound method. The algorithm uses "nodes" to keep intermediate states. There are three types of nodes: the solution nodes, the live nodes and the dead nodes. A solution node contains a solution to the problem without violating any constraint. The cost of a solution node comes directly from the cost function. The algorithm may reach multiple solution nodes as it explores the solution space. The solution with the lowest cost is the output of the algorithm.

Live nodes contain the solution to some problem that violates some constraints and they can be expanded into other nodes that violate less constraints. Once expanded, a live node is turned into a dead node, meaning that we do not have to remember it any more. In order to calculate the cost of a live node, we use the cost function for the part of the solution that does not violate the constraints, and the bound cost function for the remaining part of the solution. A feature of Branch and Bound is that once we have reached a solution node, we can prune all live nodes that have a cost higher than the cost of the solution node. This does not affect the optimality of the algorithm because the cost of a live node means that as we explore this node and fully expand all children, we will never reach a solution with a lower cost. In other words, the cost of a live node is the theoretical bound of the subtree of nodes.

The algorithm, described in Figure 4, uses a heap to maintain the set of live nodes sorted by their cost. The first node that enters the heap is the root node, a node that contains the solution of the relaxed problem described in Section 5.2. The root node dictates that all expanded sub-nodes will have a cost that is equal or higher

Algorithm 1: Multi Query Optimizer Algorithm

```
Data: Heap heap;                          // a cost-sorted heap of all live nodes
Data: Node boundSolution;                 // the relaxed problem solution
Data: Node e;                             // the currently expanded node
Data: Node solution;                      // the solution node
/* create the root node                                                    */
e ← boundSolution;
e.validOperators ← 0;
e.cost ← BoundCostFunc(boundSolution);
Push(heap, e);
solution.cost ← ∞;
while ¬ IsEmpty(heap) do
    e ← Pop(heap);
    if IsValidSolution(e) then
        // The current node is a valid solution. Keep it only
        // if it is the cheapest so far.
        if e.cost < solution.cost then
            solution ← e;
    else if e.cost < solution.cost then
        // If the current node is cheapest than the best
        // solution, expand it and look for better solutions.
        Expand(e, h)
return solution;
```

Function IsValidSolution(Node e)

```
if e.validOperators = e.totalOperators then
    /* If the sequence of all operators is valid, we have reached a
       solution                                                          */
    return true;
else
    return false;
```

Function Expand(Node e, Heap h)

```
Node[] children ← ChildrenOf(e);
foreach Node c ∈ children do
    c.validOperators ← e.validOperators + 1;
    c.cost ← CalculateCost(c);
    Push(heap, c);
```

Function CalculateCost(Node e)

```
Result: Cost cost ;                       // The cost of the given node
// Find the cost of the valid part
Cost c1 ← CostFunction(c, c.validOperators);
// Find the bounded cost of the invalid part
Cost c2 ← BoundFunction(c, c.validOperators);
return c1 + c2
```

Figure 4: Multi Query Optimization Algorithm

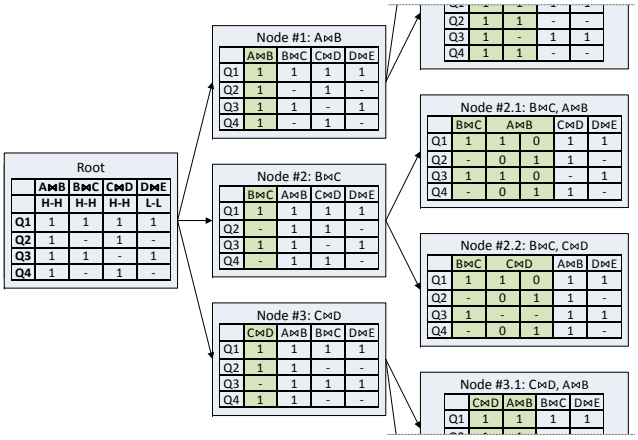


Figure 5: SWO Optimization Algorithm at Runtime

to the cost of the root node. The algorithm proceeds by removing the first node of the heap, and testing if it is a valid solution or not. In case it is a solution, and it is cheaper than any solution we have seen before, we keep it. In the case that the active node is not a solution, it is further expanded and all child nodes are inserted into the heap. As already explained, we can prune a node and the whole subtree if it has a cost that is higher than the best solution seen so far. This is because expanding a node will result in nodes with a cost equal or higher to the one of the parent node.

The algorithm implementation uses a two dimensional matrix to store the state of a node. The first (horizontal) dimension describes which operator is used, while the second one the queries that share this operator. Additionally every node remembers which part of the bitmap is the valid solution and which has yet to be expanded. The root node has no valid operators, as constraints are violated.

Figure 5 shows an example of how our algorithm explores the workspace by expanding nodes. The root node holds a matrix that describes which queries use which operators. The values of the matrix are tristate. A value of '1' means that the respective query is interested for this operator, and in this node it is using it. A value of '0' means that the query is interested for the operator, but in this step it is not using it. Finally, a value of '–' means that the query is not interested in this operator.

In the example of Figure 5 expanding the root node results in three child nodes. These child nodes have the first operator fixed, meaning that as they expand, this part of the solution will remain constant. Their cost is calculated by combining the cost of the first operator using the cost function with the cost of the remaining operators using the bound function.

During node expansion all constraints must be taken into account. In our example the expansion of the root node creates only 3 child nodes. The $\texttt{D} \bowtie \texttt{E}$ operator is not part of the permutation, because it is a L–L operator and the ordering heuristic suggests that the selective operators should be executed first. Also, as node #2 expands into node #2.1, the $\texttt{A} \bowtie \texttt{B}$ operator is duplicated. This is done in order to adhere to the sharing constraint that was explained in Section 4.1. The first replica serves only Q1 and Q3 and it is able to join tuples of (BC) with tuples of (A), while the second replica serves Q2 and Q4 and it is able to join tuples of (B) with tuples of (A). For reasons of simplicity, we have not taken into account different build and probe order in this example.

Finally, at this point we should notice the importance of the heuristics. Without the sharing heuristic, node #1 would be split into 8 different nodes that contain all the possible values of the bitmap $\{1, 1, 1, 1\}$. The optimizer would have had to consider instantiating the $\texttt{A} \bowtie \texttt{B}$ operator just for query 1 ($\{1, 0, 0, 0\}$), then just for query 2 ($\{0, 1, 0, 0\}$) and so on, for a total of 2^4 nodes. The benefits of the ordering heuristic is less obvious. The root node expands only to 3 nodes instead of 4, as the hint says that the L–L joins should be executed after the H–H joins.

6. EXPERIMENTAL ANALYSIS

In order to evaluate the performance of our work sharing multi query optimization algorithm, we carried out a series of performance experiments. First we present experimental results to validate the correctness of the two heuristics of our algorithm and then we present results on the generated plans of two well known workloads, the TPC-W and the TPC-H benchmarks.

The baseline we compared against is multi query optimization, where only subexpressions of the same prepared statements are executed together (predicate sharing). Executing the same workload on a query-at-a-time database system, like MySQL, would be fruitless. SW systems outperform query-at-a-time systems, especially under high load, as already shown in the experiments of related work [6, 7]. We decided not to perform any tests with a query-at-a-time system, as this paper focuses on the techniques of optimizing entire workloads info an globally efficient access plan.

The generated plans were implemented on a cooperative relational database system, SharedDB. In SharedDB each operator is assigned to one or more CPU cores that are dedicated to it. Queries are pushed from the top of the query plan until a storage engine is reached, and then results are pushed all the way to the original client. SharedDB batches queries on every operator and executes these batches one at a time. Any work that is common within a batch, is not repeated. Additional information on how SharedDB processes queries can be found in [6].

The baseline plans were also implemented on SharedDB, with the additional constraint that each batch can contain only queries that originate from the same prepared statement (i.e. they have common subexpressions). The sharing that occurs in baseline plans resembles the one of Figure 2b, while our algorithm shares more work by creating plans similar to the one of Figure 2c. In order to generate the baseline plans, we used dynamic programming to optimize each individual query. Then we combined the locally optimal plans to create the global query plan.

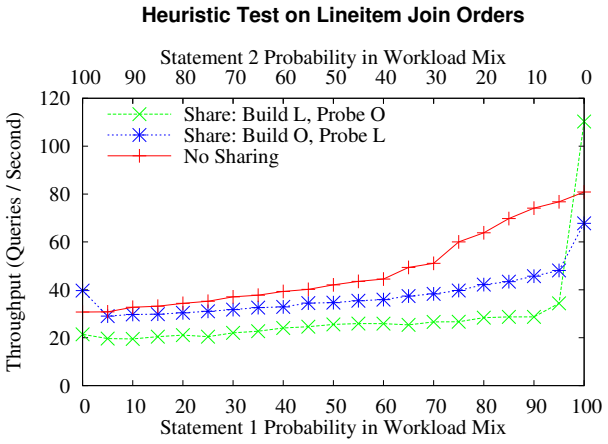
The global query plans were generated on a dual core laptop, as the runtime of the optimization algorithm is very small. We discuss the exact optimization time of each experiment as we present the results. The same laptop was used to generate the baseline query plans. In all experiments reported in this paper, we used a multi core machine to evaluate the efficiency of the generated plans. This machine features four eight core Xeon CPUs. Each core has a 2.4 GHz clock frequency and is hyper threaded. The hyper threaded cores were used to run the clients. This did not affect the performance in any benchmark, as clients mostly wait for the results to arrive and perform no additional work.

6.1 Heuristic Test

The first part of our experimental study focuses on the correctness of the heuristics. We used two small microbenchmarks, one that involves a two way join across two relations, and one that involves two overlapping three-way joins. In both cases, we used two prepared statements with very diverse selectivities.

Two Way Joins

For the first microbenchmark, we used two relations, L and O. L was loaded with approximately 1 GB of data that were generated from the LINEITEM relation of TPC-H, while O was loaded with approximately 230 MB of the ORDER relation. We considered a



(a) Experimental Results

PS1	PS2	Maximum	Optimal	Actual
0	10	13,500,000	13,500,000	13,500,000
5	5	78,005,035	30,004,860	30,004,860
10	0	54,010,935	54,010,935	54,010,935

(b) Number of Shared Tuples for 10 Queries

Figure 6: Testing Heuristic on a 2-Way Join

natural join across these two relations and issued a workload of these two prepared statements:

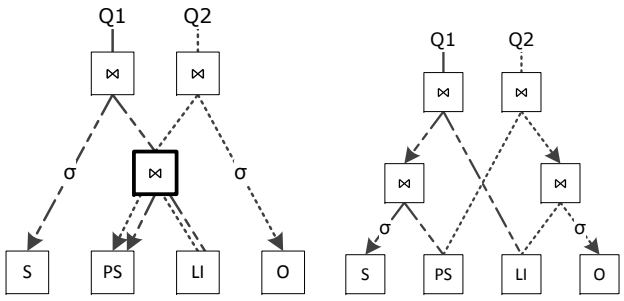
```
PS1:  SELECT * FROM L JOIN O on L.o_id = O.id
      WHERE L.part = ?
PS2:  SELECT * FROM L JOIN O on L.o_id = O.id
      WHERE O.id = ?
```

The query parameters were also generated based on the TPC-H specification. We used a total of 160 client threads that issued queries generated from these prepared statements in a closed loop.

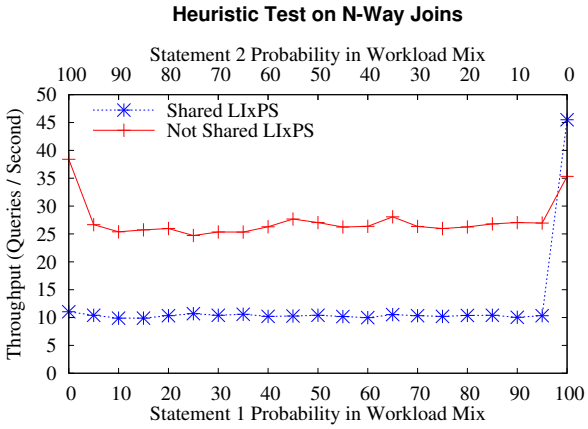
Based on our operator classification presented in Section 4.2, these two statements have a L-H join. We considered three different global plans to execute this workload. The first plan creates an operator that builds a hash table with tuples from the L relation and probes with tuples from O and shares the operator across both statements. The second one builds on O, probes on L and shares the operator across both statements as well. Finally, the third possible plan uses both of these join operators, one for PS1 and one for PS2. All three plans use the same amount of resources, in terms of number of CPU cores and memory buffers.

We varied the probability of a prepared statement appearing in the workload and measured the achieved throughput on all possible plans. Figure 6a shows the results. In the left-most extreme, where no queries generated from PS1 were executed, the second plan is better, while in the right-most extreme, where only queries of type PS1 were executed, the first plan is better. However, in these extreme cases we would never consider work sharing. If the probability of PS1 is 0, there is no need to create a shared operator that would serve PS1 and PS2.

The area of the plot between the two extremes is the interesting region. The "No Sharing" plan has a higher performance in this area. This means that whenever two queries with diverse selectivities are present, the optimizer should never attempt to share the operator across them. This is exactly the hint that the sharing heuristic provides. Finally, the table of Figure 6b shows the number of shared tuples while executing 10 of these queries. We see that in all cases the measured number of shared tuples is equal to the optimal. The optimal sharing is computed by evaluating all possible plans and computing their costs.



(a) Plan with Shared LI ⋈ PS **(b)** Plan with no Sharing



(c) Experimental Results

PS1	PS2	Maximum	Optimal	Actual
0	10	61,210,935	61,210,935	61,210,935
5	5	122,421,870	54,409,720	54,409,720
10	0	61,210,935	61,210,935	61,210,935

(d) Number of Shared Tuples for 10 Queries

Figure 7: Testing Heuristic on a N-Way Join

N-Way Joins

The next microbenchmark studies the correctness of the heuristic in more complicated scenarios. We used a schema consisting of four relations taken from the TPC-H benchmark, Supplier, PartSupp, LineItem and Orders. Similar to the 2-way experiments, we used a workload of the following two statements:

```
PS1:  SELECT * FROM S JOIN PS on S.id = PS.s_id
      JOIN LI on PS.ps_id = LI.ps_id WHERE S.id = ?
PS2:  SELECT * FROM O JOIN LI on O.id = LI.o_id
      JOIN PS on LI.ps_id = PS.ps_id WHERE O.id = ?
```

To answer this workload, we considered the two global plans of Figures 7a and 7b. The first plan uses a shared operator to evaluate Lineitem ⋈ PartSupp. This is against the sharing heuristic, as in the first prepared statement, tuples from PartSupp can be filtered, while in the second prepared statement, tuples from LineItem can be filtered. Also, the ordering heuristic would instruct the optimizer to execute LI ⋈ PS at the end, as it is a L-L join. In order to ensure fairness in our comparisons, we assigned twice the amount of resources (CPU cores, buffer size) to the shared operator of Figure 7a.

We measured the throughput of the two global query plans, as we modified the probability of a statement appearing in the mix. The results are presented in Figure 7c. The results show that not sharing the LI ⋈ PS operator and executing it after filtering as many tuples as possible is better for all cases, except for the extreme case where only queries of the second statement are executed. In

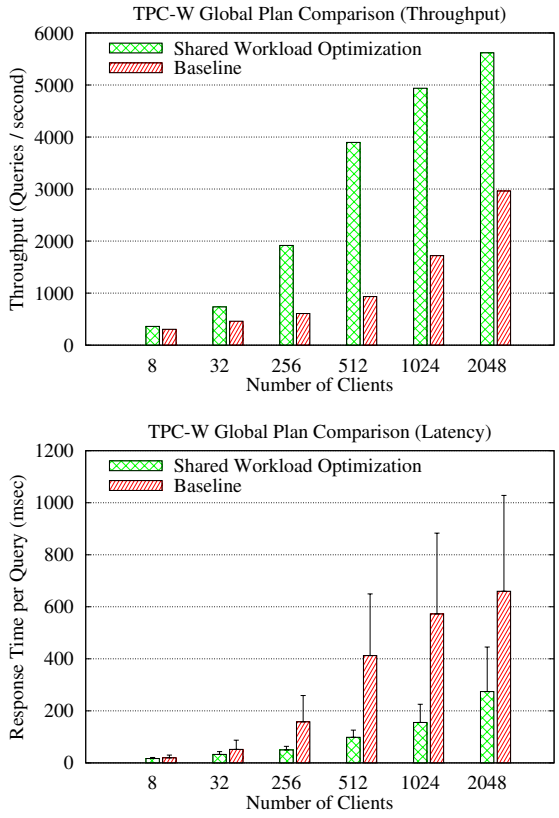


Figure 8: Testing the TPC-W Browsing Workload

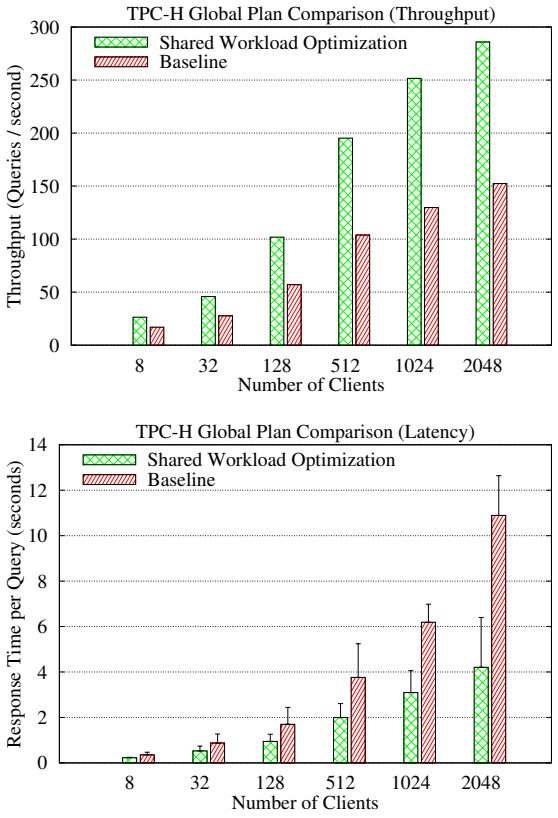


Figure 9: Testing the TPC-H Workload

this case, the two cores that were assigned to the shared operator, give an advantage to the shared plan. Yet, if the optimizer knows that `PS1` will never appear in the workload, there will be no need for a decision of sharing this join or not. Additionally, the number of shared tuples when executing 10 queries can be seen in 7d. As with before, the measured shared tuples match the optimal, something that is expected given that the solution space is very small.

6.2 TPC-W Analysis

As seen in the previous experiments, sharing as much work as possible (greedily) is not optimal. The two heuristics of our algorithm make sure that we do not share too many operators. In most realistic workloads, removing these cases still leaves enough opportunities for sharing. To demonstrate this, we used the TPC-W workload with 10,000 items which contains a total of 11 prepared statements that use a total of 17 join operators. The prepared statements were issued based on the frequencies defined by the TPC-W Browsing mix. The query parameter were generated as defined in the TPC-W specification. What makes this workload interesting is that it contains queries that have deep pipelines of operators (analytical queries) as well as short running queries that join only a couple of tuples every time. In order to focus only on the query plan efficiency, we used only the prepared statements and skipped the web interface that is part of the TPC-W specification.

Since our implemented algorithm focuses on hash join and scan operators, we removed all other relational operators from the prepared statements. All selections were executed as part of the scans. Yet, this does not limit the applicability of our algorithm. The same technique can be easily applied to all relational operators, like `SORT` and `GROUPBY`. We limited our implementation to join operators because of their sensitivity to optimization decisions. Additionally, adding different storage access (i.e. index lookup) and different flavors of joins is orthogonal to our algorithm and has been extensively examined in prior work [22, 11].

The runtime of our SWO optimizer for the whole TPC-W was less than 3 seconds, which is quite fast, given it requires analysis of 17 join operators across 11 statements. To put it into perspective, a brute force optimizer would have to consider a total of $2^{16} * \frac{(2*11)!}{11!} = 4.39 * 10^{16}$ possible solutions. The generated plan contained only 10 join operators, some of which were shared across multiple statements. The baseline plan, where operators are shared across queries of the same statements only, was generated in 500 msec and requires a total of 17 join operators. In traditional MQO, this plan has to be generated every time a new set of queries arrives in the system. We did not include this overhead in our measurements. The measured latency and throughput account only for the query execution.

We implemented the two plans in SharedDB and used a variable number of TPC-W clients in a closed system with no thinking time, as we measured the throughput and latency of the system. Each experiment was run for one hour. The results, shown in Figure 8, clearly demonstrate that the shared workload optimization algorithm outmatches the traditional multi query optimization techniques. Sharing operators across statements independently of whether there are common subexpressions or not is beneficial both in terms of throughput and in terms of latency. For the latter, we can also observe that work sharing reduces the variance of the response time. The plot illustrates the 50th and 90th percentiles.

6.3 TPC-H Analysis

Next, we used the TPC-H benchmark with a scale factor of 1.0 to measure the quality of the work sharing global query plans. As

with TPC-W we simplified the queries by removing all relational operators except joins and full table scans. We used all 22 statements that are part of the specification, even the ones that require just a full table scan and no join operators, like `Q1` and `Q6`. A total of 49 hash join operators were considered.

The SWO algorithm needs around 30 seconds to optimize this workload, which is acceptable, given the size of the problem and the fact that the generated plan can be used for the whole lifetime of the system. The generated plan contained only 34 operators, some of which were shared. The baseline plan for a set of 22 queries requires on average 3 seconds to be generated. As with the TPC-W benchmark, our metrics do not include the overhead of generating a plan every time a new set of queries arrives.

The results of our experiment are shown in Figure 9, and clearly show the advantages of work sharing. Any work that is common across statements is executed only once, which explains the higher throughput and the lower response time of the work sharing plan.

As already explained, a query-at-a-time system database would not be able to handle such a load. Since, all table accesses are full table scans, the storage engine overhead would dominate the performance of such a system. Additionally, the number of clients is prohibitive. Executing 2,048 queries concurrently would either require queuing, or queries will fight with each other for resources. In both cases, performance would be significantly lower compared to the performance of a SW system.

7. CONCLUSION

This paper presented a global query optimization algorithm that can be used to optimize the whole workload on modern cooperative database systems. The problem is twofold. The optimizer has to order the relational operators as well as decide which queries should share them. The generated plan does not necessarily answer each query in the best possible way. Nevertheless, the cost of executing the whole workload is optimized.

What makes the problem interesting is that state-of-the-art query optimization techniques cannot be applied. The solution space is so big that exhaustive techniques cannot be used. Additionally, the problem does not have optimal substructure and as a result, it cannot be broken into smaller subproblems. To tackle these challenges, our optimizer uses the branch and bound optimization method. In order to reduce the solution space, we introduce two heuristics that prune suboptimal solutions and guide the optimizer to explore first the most promising part of the solution space. Our experimental results on the generated plans of different workloads, show that work sharing outperforms traditional multi query optimization techniques that share work only across common subexpressions.

Acknowledgments. This work was funded by the Enterprise Computing Center of ETH Zurich (www.ecc.ethz.ch). The work of G. Giannikis was supported by a Swiss National Science Foundation grant as part of its Pro-Doc program.

8. REFERENCES

[1] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The DataPath System: A Data-Centric Analytic Processing Engine for Large Data Warehouses. In *Proc. SIGMOD 2010*, pages 519–530.

[2] G. Candea, N. Polyzotis, and R. Vingralek. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. In *Proc. VLDB 2009*, pages 277–288.

[3] G. Candea, N. Polyzotis, and R. Vingralek. Predictable Performance and High Query Concurrency for Data Analytics. *VLDB Journal*, 20(2):227–248, 2011.

[4] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. Pipelining in Multi-Query Optimization. In *Proc. PODS 2001*, pages 59–70.

[5] L. Devroye and C. Zamora-Cura. On the Complexity of Branch and Bound Search for Random Trees. *Random Struct. Algorithms*, 14(4):309–327, July 1999.

[6] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one Thousand Queries with one Stone. *Proc. VLDB Endow.*, 5(6):526–537, Feb. 2012.

[7] S. Harizopoulos and A. Ailamaki. StagedDB: Designing Database Servers for Modern Hardware. *IEEE Data Eng. Bull. 2005*, 28(2):11–16.

[8] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *Proc. SIGMOD 2005*, pages 383–394.

[9] T. Ibaraki and T. Kameda. On the Optimal Nesting Order for Computing N-Relational Joins. *ACM Trans. Database Syst.*, 9(3):482–502, Sept. 1984.

[10] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves. An Architecture for Recycling Intermediates in a Column-Store. In *Proc. SIGMOD 2009*, pages 309–320.

[11] D. Kossmann and K. Stocker. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. *ACM Trans. on Database Systems*, 25:2000, 1998.

[12] S. Kosuch and A. Lisser. Upper Bounds for the 0-1 Stochastic Knapsack Problem and a B&B Algorithm. *Annals OR 2010*, 176(1):77–93.

[13] C. Lang et. al. Increasing Buffer-Locality for Multiple Relational Table Scans through Grouping and Throttling. *Proc. ICDE 2007*.

[14] A. Lecchini, J. Lygeros, and J. M. Maciejowski. Stochastic Optimization on Continuous Domains with Finite-Time Guarantees by Markov Chain Monte Carlo Methods. *IEEE Transactions on Automatic Control*, 55:2858–2863, Dec. 2010.

[15] S. Manegold, A. Pellenkoft, and M. Kersten. A Multi-Query Optimizer for Monet. Technical report, Amsterdam, The Netherlands, 2000.

[16] K. Ono and G. M. Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. In *Proc. VLDB 1990*, pages 314–325.

[17] F. Palermo. A Database Search Problem. In *Proc. of the 4th Symposium on Computer and Information Science 1974*, pages 67–101.

[18] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-Memory Scan Sharing for Multi-Core CPUs. In *Proc. VLDB 2008*, pages 610–621.

[19] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *Proc. ICDE 2008*, pages 60–69.

[20] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proc. SIGMOD 2000*, pages 249–260.

[21] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. Database Engines on Multicores, Why Parallelize when you can Distribute? In *Proc. EuroSys 2011*, pages 17–30.

[22] 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 *Proc. SIGMOD 1979*, pages 23–34.

[23] T. K. Sellis. Multiple-Query Optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.

[24] K.-L. Tan, S.-T. Goh, and B. C. Ooi. Cache-on-Demand: Recycling with Certainty. In *Proc. Data Engineering 2001*, pages 633–640.

[25] N. Thakoor, V. Devarajan, and J. Gao. Computation Complexity of Branch-and-Bound Model Selection. In *Proc. Computer Vision 2009*, pages 1895–1900.

[26] P. Unterbrunner. *Elastic, Reliable, and Robust Storage and Query Processing with Crescando/RB*. PhD thesis, ETH Zürich, 2012.

[27] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable Performance for Unpredictable Workloads. In *Proc. VLDB 2009*, pages 706–717.

[28] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner. Efficient Exploitation of Similar Subexpressions for Query Processing. In *Proc. SIGMOD 2007*, pages 533–544.

[29] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *Proc VLDB 2007*, pages 723–734.