SkinnerMT: Parallelizing for Efficiency and Robustness in Adaptive Query Processing on Multicore Platforms

Ziyun Wei
Cornell University
Ithaca, NY, USA
zw555@cornell.edu

Immanuel Trummer
Cornell University
Ithaca, NY, USA
itrummer@cornell.edu

ABSTRACT

SkinnerMT is an adaptive query processing engine, specialized for multi-core platforms. SkinnerMT features different strategies for parallel processing that allow users to trade between average run time and performance robustness.

First, SkinnerMT supports execution strategies that execute multiple query plans in parallel, thereby reducing the risk to find near-optimal plans late and improving robustness. Second, SkinnerMT supports data-parallel processing strategies. Its parallel multi-way join algorithm is sensitive to the assignment from tuples to threads. Here, SkinnerMT uses a cost-based optimization strategy, based on runtime feedback. Finally, SkinnerMT supports hybrid processing methods, mixing parallel search with data-parallel processing.

The experiments show that parallel search increases robustness while parallel processing increases average-case performance. The hybrid approach combines advantages from both. Compared to traditional database systems, SkinnerMT is preferable for benchmarks where query optimization is hard. Compared to prior adaptive processing baselines, SkinnerMT exploits parallelism better.

PVLDB Reference Format:
Ziyun Wei and Immanuel Trummer. SkinnerMT: Parallelizing for Efficiency and Robustness in Adaptive Query Processing on Multicore Platforms. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at http://vldb.org/pvldb/format_vol14.html.

1 INTRODUCTION

Traditional query optimizers select join orders based on coarse-grained data statistics and many simplifying assumptions [24]. All too often, those assumptions (e.g., uniform data and uncorrelated query predicates) do not hold. In that case, query optimizers may generate query plans whose execution cost exceeds the optimum by orders of magnitude. Those long standing problems have recently motivated the use of machine learning for query optimization [17, 19, 22]. Most work in that domain focuses on "inter-query learning". This means that experiences gained from past queries are used to make better optimization decisions for the next. However, such methods suffer from a "cold start" problem and are not helpful for fresh data or unusual queries (e.g., queries introducing new user-defined functions).

Adaptive query processing [3, 4, 23] is another idea to mitigate the effects of unreliable query optimization. Here, the selected plan may change over the course of query execution. This allows integrating run time feedback into plan choices. While early forms of adaptive processing were targeted at stream data processing [3], allowing for longer convergence times, recent work [20, 29] has shown promising performance on standard benchmarks such as TPC-H as well. This paper presents SkinnerMT, a new engine for adaptive processing on multi-core platforms.

In the context of adaptive processing, there are two ways to exploit parallelism. First, alternative plans can be processed in parallel. Second, the same plan can be processed on different data partitions in parallel. Processing more data in parallel is beneficial if the executed plan is reasonably efficient. On the other side, exploring more plans in parallel can help to identify near-optimal plans faster. Of course, mixtures between the two extremes (i.e., processing multiple plans and multiple data partitions in parallel) are possible as well. SkinnerMT features different parallel processing strategies that cover all of the aforementioned possibilities. It has been built to systematically study the research question of how to exploit multicore parallelism best for adaptive processing. It is a fork of the recently proposed SkinnerDB system [29], an adaptive processing engine based on a reinforcement learning framework, which executes all joins sequentially.

SkinnerMT uses adaptive processing to find good join orders. Arguably, this is perhaps the most difficult and impactful choice made by the query optimizer. To enable fast and adaptive join order switching, SkinnerMT uses specialized join algorithms and data structures. It divides query processing into small time slices in which different join orders are tried. Measuring the amount of data processed per time slice allows obtaining (noisy) estimates of join order quality. Based on those run time performance measurements, SkinnerMT uses reinforcement learning to select which join order to try next. In doing so, it balances exploration (trying out join orders about which little is known) and exploitation (re-using join orders that are promising, according to current quality estimates) in an optimal manner.

SkinnerMT divides query processing into multiple phases (e.g., separating the processing of unary predicates and grouping or aggregation operations from join processing). For most of those phases, parallelization is trivial. This does not hold for the join processing phase. Parallelizing join processing is the focus of this paper

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

and SkinnerMT offers a rich set of parallelization strategies. As discussed later, those strategies allow users for optimizing for different metrics such as raw performance or performance robustness.

First, SkinnerMT can exploit multi-threading to explore multiple join orders in parallel. In the simplest version, the space of alternative join orders is split uniformly across different threads. A more sophisticated variant adapts the assignment from search space partitions to threads over time. The goal is to assign more threads to search space partitions that are likely to contain interesting join orders, based on performance observed so far. This avoids cases in which many threads explore parts of the search space that contain no near-optimal join orders. Processing multiple join orders in parallel may seem wasteful as it leads to redundant work. However, it can help for queries where finding good join orders is difficult. More importantly, it decreases performance variance. As adaptive processing is subject to random variations (due to randomized order of exploration and noisy performance feedback), limiting the scope of exploration per thread reduces convergence time.

Second, SkinnerMT can exploit multi-threading to process the same join order in parallel. A large body of prior work focuses on parallelizing traditional join operators. However, those join operators cannot deal with the requirements of adaptive processing with high-frequency join order switches [29] (which is why prior work on adaptive processing often opts for specialized join operators [3]). Instead, SkinnerMT uses a parallel, multi-way join operator that avoids creating large intermediate results (which would be costly to materialize when switching between join orders that generate different intermediate results). It exploits parallelism by assigning tuples in a specific table (the so called "split table") to specific threads. The choice of a split table is impactful and highly non-trivial. The best choice is determined by various factor, including the table position in join order (which changes over time) as well as data properties (e.g., value skew). Therefore, as shown later, simple strategies fall far short of realizing optimal speedups via parallelization. Instead, SkinnerMT uses cost-based optimization to select optimal split tables, based on run time feedback.

Finally, SkinnerMT supports hybrid strategies, exploiting parallel processing and parallel search at the same time. This includes a dynamic variant that gradually switches from parallel search to parallel processing.

We analyze all proposed algorithms formally and evaluate them in experiments. We evaluate on various benchmarks, ranging from benchmarks where optimization is easy (TPC-H benchmark [27], due to uniform data) over benchmarks where optimization is moderately difficult (join order benchmark [12]), up to benchmarks where optimization is hard (JCC-H, due to highly skewed data [7]). It turns out that parallel search leads to maximal robustness, minimizing the execution time variance when executing queries repeatedly. Parallel join processing reduces average run time significantly, in exchange for increased variance. The hybrid algorithm (in the dynamic variant) realizes attractive tradeoffs between average run time and variance. Compared to SkinnerDB, the most similar prior work, SkinnerMT scales well in the number of threads while SkinnerDB is bottlenecked by sequential joins. Compared to traditional database systems such as MonetDB, SkinnerMT performs better on benchmarks where query optimization is difficult (e.g., the join order benchmark). SkinnerMT's speedups via parallelization are comparable to the ones of traditional database systems. In summary, our original scientific contributions are the following:

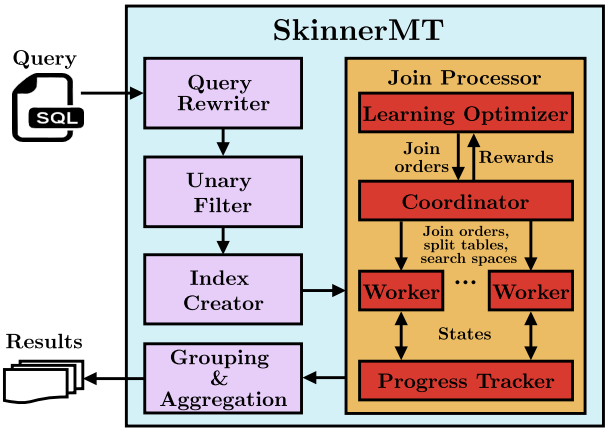


Figure 1: Overview of SkinnerMT.

- We describe SkinnerMT, an adaptive query processing engine featuring different parallelization strategies.
- We present several algorithms for parallel, adaptive processing, exploiting parallel processing in different ways.
- We formally analyze the algorithms and evaluate SkinnerMT with different configurations in experiments.

The remainder of this paper is organized as follows. Section 2 gives an overview of the SkinnerMT system. Section 3 introduces a data structure used to store execution state for specific join orders. Section 4 describes how SkinnerMT performs data-parallel processing. Section 5 describes how to leverage parallelism to explore more query plans in parallel instead. Section 6 proposes an algorithm that combines parallel exploration of join orders with parallel data processing. Section 7 analyzes all proposed processing strategies formally. In Section 8, we report experimental results. We discuss related work in Section 9 before we conclude.

2 SYSTEM OVERVIEW

SkinnerMT is an analytical, relational database management system for in-memory data processing. It is specialized for exploiting multicore parallelism via multiple parallel processing strategies. SkinnerMT is implemented in Java, primarily, while invoking fast C code via the Java Native Interface for operations such as filtering, index creation, and aggregation. The current prototype supports SQL features that are required by the benchmarks used in the experiments, including TPC-H and the join order benchmark.

Figure 1 shows an overview of the SkinnerMT system. Each incoming query is first rewritten to decompose it into a sequence of simple SPJGA (select, project, join, group-by, aggregation) queries. Each of the simple queries is processed in multiple phases. First, unary predicates are applied and the resulting tables are materialized. Second, in-memory hash indexes are created on columns that appear in equality joins. In order to enable arbitrary join orders, SkinnerMT creates hash indexes on any column that may be probed during join processing (i.e., compared to a system that prepares

hash indexes for a single join order, SkinnerMT typically creates indexes on around twice as many columns).

Third, SkinnerMT uses adaptive processing strategies to perform joins. It uses specialized multi-way join algorithms (explained in more detail later) to enable fast join order switching. SkinnerMT supports multiple parallel join processing strategies. They differ in how they exploit parallelism and they realize different tradeoffs between expected performance and performance robustness. For all variants, joins are executed by worker threads. Workers frequently suspend joins and resume join execution with a different join order. To avoid redundant work, workers communicate with the progress tracker component. This component uses a specialized data structure (discussed in more detail later) to concisely store execution state for multiple workers and multiple join orders. Furthermore, the progress tracker tries to merge progress achieved via different join orders. At the same time, workers communicate with a coordinator component. Depending on the parallelization strategy, this component assigns worker threads either to join orders, split tables (tables split into partitions for different worker threads), or partitions of the join order search space (or a combination thereof).

Join orders are selected via reinforcement learning. Performance statistics about join orders translate into reinforcement learning rewards. The reward function is the sum of two components. First, it measures the number of complete join result tuples produced per time unit. As all join orders produce the same total number of tuples, faster join orders produce more tuples per time unit in average. Second, to cover cases where join results are very sparse, the reward function measures the speed at which tuple combinations are explored. This speed is higher if, e.g., having selective predicates early in the join order allows the multi-way join to exclude tuple combinations in the remaining tables quickly. More precisely, the second reward component is calculated by $\sum_{1 \leq i \leq n} \delta_i / (\prod_{1 \leq k < i} c_k)$ where δ_i refers to the tuple selected in the i-th out of n tables in join order, capturing the gap between initial and final tuple index when executing that join order for a time slice, while c_i is the cardinality of the i-th table in join order. It can be shown [29] that the latter term sums up to one for a full execution of arbitrary join orders, i.e. faster join orders have again a higher average reward. The area of reinforcement learning [26] has produced a broad portfolio of algorithms that guarantee near-optimal expected rewards in the face of uncertainty. Using such algorithms and the aforementioned reward function, SkinnerMT converges to near-optimal join orders.

More precisely, the learning optimizer component uses the UCT algorithm [16] to choose interesting join orders to explore, balancing the need for exploration (trying new join orders) with the need for exploitation (trying join orders showing good performance so far). This algorithm builds a search tree over join orders, associating sub-trees with confidence bounds on average rewards (obtained when sampling join orders within that sub-tree). The algorithm builds the search tree gradually over time, adding at most one node per reinforcement learning sample. This is crucial to avoid prohibitive overheads when creating trees representing the entire join order space for large queries. For determining optimal split tables and search space partitions, specialized decision mechanisms are used, discussed in more detail later. Note that components in Figure 1 do not necessarily map to separate threads. For instance, depending on the parallelization variant, the same thread may process join orders and select join orders.

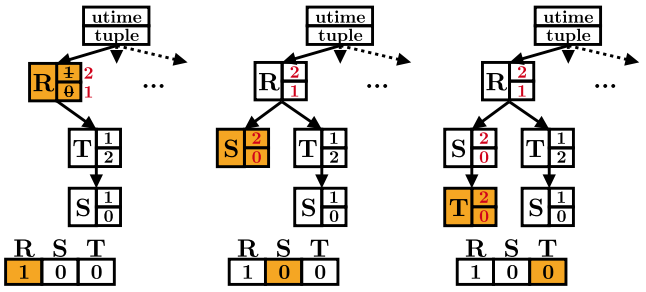
Algorithm 1 Retrieving and updating join execution state.

```
1:  // Store start tuple indices when resuming join order j
2:  // into vector v using progress tree with root p.
3:  procedure RESTORE(p, j, v)
4:      for i ← 1, ..., j.length do
5:          // Select child node in progress tree
6:          c ←GETCHILD(p, j_i)
7:          // Is child node outdated?
8:          if p.utime > c.utime then
9:              c ← null
10:         end if
11:         // Extract index of latest tuple
12:         if c = null then
13:             v_{j_i} ← 0
14:         else
15:             v_{j_i} ← c.tuple
16:         end if
17:         p ← c
18:     end for
19: end procedure
20: // Update progress tree with root p considering
21: // vector of latest tuple indices v for join order j.
22: procedure UPDATE(p, j, v)
23:     // Update progress tree
24:     for i ← 1, ..., j.length do
25:         c ←GETORCREATE(p, j_i)
26:         // Does child need an update?
27:         if c.tuple ≠ v_{j_i} ∨ p.utime > c.utime then
28:             c.tuple ← v_{j_i}
29:             c.utime ←THISUPDATETIME
30:         end if
31:         p ← c
32:     end for
33: end procedure
```

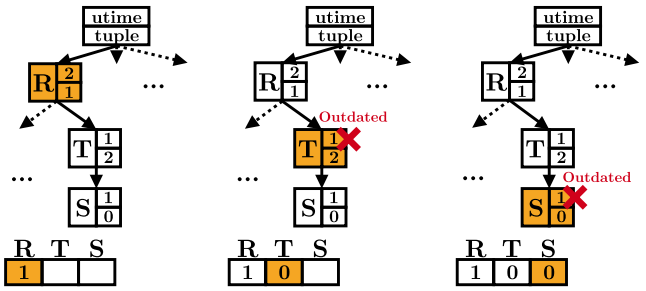
During join processing, join result tuples are collected from different threads. Keeping track of the tuple lineage (i.e., the vector of offsets of joined tuples in the base tables) allows eliminating tuples that are generated redundantly (e.g., by different threads or by the same thread via different join orders). After a full join result is generated, the join result is materialized and the grouping and aggregation phase starts. Here, group-by clauses and aggregates are processed to generate the final result. This result is either returned to the user or serves as input for the next query in the sequences of simple queries, resulting from query rewriting.

3 TRACKING PROGRESS

The execution state of the multiway join algorithm used by SkinnerMT (described in detail in the next section) is fully captured by one integer number per table. This number indicates the position (i.e., column array index) of the currently selected tuple. When selecting a new join order, worker threads start from the last execution state that is available for this order (or from the first tuple in each table, if the order is selected for the first time). All parallel



(a) Update progress for join order $R \bowtie S \bowtie T$ at Timestamp 2.



(b) Restore state for join order $R \bowtie T \bowtie S$ at Timestamp 3.

Figure 2: Storing and retrieving order-specific execution state.

processing strategies use the same data structure to store execution state for different join orders and threads concisely. This data structure, the progress tree, is described in the following.

Each node in the progress tree is associated with a join order prefix. Nodes in the i-th tree level represent prefixes of length i. Leaf nodes are associated with entire join orders. Each tree edge is associated with a query table. Traversing an edge appends the associated table to the join order prefix (i.e., the child node represents the prefix with appended table, compared to the parent node). For each node, we store two integers: a tuple index and an update timestamp. The tuple index represents execution state. It refers to the last table in the join order prefix associated with the corresponding node. It indicates which tuple was considered in that table when the join was interrupted last. When resuming processing for the corresponding join order, we start from that index.

The progress tree merges progress that was achieved according to different join orders. Join order prefixes belong to multiple join orders. Hence, the latest execution state for a first join order may partially override state of a second order. In those cases, it is important to understand which state is associated with what join order. It is for instance not admissible to mix the last tuples considered for different join orders in different tables. Resuming join processing from such a mixed state may lead to skipped join result tuples. Hence, we store in each tree node a timestamp, indicating when the tuple index was updated for the last time. A change in timestamp, when traversing the tree, indicates that tuple indices may refer to different join orders.

Algorithm 1 shows pseudo-code for storing and restoring execution state. Procedure Restore is called at the start of each time slice, when resuming join processing with a new join order. Procedure Update is called at the end of a time slice, to backup progress made with the current join order. Both procedures obtain as input the root node p of the progress tree, the join order j, and a vector v of base table tuple indices to store or to restore. When restoring execution state, we iterate over the tables of the input join order. At the same time, we traverse the progress tree, retrieving the edge associated with the next table using Function GetChild. The function returns **null** if no corresponding edge exists (i.e., if this join order was tried for the first time, covering the case p = **null** as well). Also, if update timestamps (field *utime*) indicate an outdated child node, we act as if no edge exists for the next table. A node becomes outdated if one of its predecessor nodes was updated more recently. This can happen if it belongs to a join order that has been outperformed by a different join order, sharing a join order prefix. In that case, the predecessor nodes associated with the other join order are updated while the outdated node is not. We restore tuple indices stored in the progress tree (field *tuple*) until one of the two aforementioned cases arises (or until the join order was completely processed).

Similarly, when backing up execution state, we traverse the progress tree according to the input join order. Function GetOrCreate retrieves child nodes in the progress tree, if they exists, and creates them otherwise. The update procedure is only called if the execution state stored in v is useful, meaning that it dominates progress already stored. Hence, we override existing progress where it differs from the input. Increasing the timestamps of nodes on the path representing the current join order makes other child nodes outdated. Hence, if possible, we want to avoid doing so. This motivates the condition in Line 27, checking whether stored progress differs from the input and whether the child is already outdated. In those cases, we override the tuple index and update the timestamp (Function ThisUpdateTime returns the same value during the same invocation of Function Update which increases strictly monotonically over different invocations).

Example 3.1. Figure 2 illustrates state retrieval and progress updates. Figure 2a shows the tree is updated to reflect the current execution state [1,0,0] (indexes of selected tuples in join order) for the join order $R \bowtie S \bowtie T$ with Timestamp 2. Timestamp and tuple index are updated for the node in the first tree level, capturing progress by all join orders starting with table R. Nodes representing join orders with prefix $R \bowtie S$ and the full join order $R \bowtie S \bowtie T$ are created successively (update steps from left to right). Figure 2b illustrates execution state retrieval for join order $R \bowtie T \bowtie S$. The tuple indexes of the current state (shown on the bottom of the figure) are retrieved in join order. Note that the tuple index retrieved for the first table, R, is due to the update from Figure 2a which refers to a *different* join order with an overlapping prefix. As the overlapping prefix only covers the first table, join processing must start from the first tuple in the remaining tables. This information is captured by the timestamps in the tree, indicating that tuple positions for T and S refer to an older update and cannot be used.

The data structure described so far stores progress for a single thread. If multiple threads process join orders in parallel on different data partitions (as described in the next section), progress made for the same join order may differ across threads. It is possible

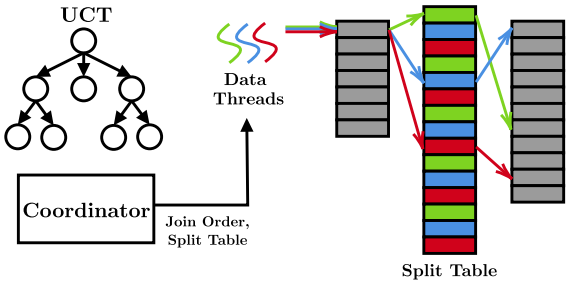


Figure 3: Worker threads execute a parallel multi-way join algorithm, using join orders and split tables selected by the coordinator.

to store one progress tree per thread. However, this means that space consumption increases linearly in the number of threads (and progress trees are stored in main memory). Instead, we opt to store for each join order progress made by the slowest thread alone (using relative tuple indexes, referring to each thread's specific data partition). This allows other threads to resume execution from the stored state without skipping tuples. On the other hand, it means faster thread may have to redo work. This tradeoff seems acceptable: if a join order allows some threads to advance significantly faster than others (e.g., due to data skew), it is not suitable for parallel processing (as the slower threads will become the bottleneck).

4 DATA PARALLELISM

SkinnerMT supports adaptive, data-parallel (DP) query processing. We discuss a parallel, multi-way join algorithm in Section 4.1. Conceptually, this algorithm partitions tuple in a specific table (the "split table") between threads. The choice of a good split table is critical for performance. In Section 4.2, we discuss a cost-based approach to select split tables based on run time feedback.

4.1 Parallel Multi-Way Join

Binary join algorithms generate a sequence of intermediate results, before producing the first complete join result tuple. If temporarily switching from a first to a second join order, the intermediate results of the first join order would have to be backed up (to avoid redundant work when selecting the first join order again). However, this causes significant cost in terms of space and materialization time. Instead, SkinnerMT uses a multi-way join algorithm that is optimized for quick join order switches. It aims at generating complete join result tuples as quickly as possible, never keeping more than one intermediate result tuple at the same time. Complete join tuples are not specific to the join order anymore (unlike intermediate result tuples as the set of intermediate results generated depends on the join order) and can therefore be saved without space penalty. The only execution state that is specific to join orders is the index of the currently selected tuple in each table. This state is stored in the data structure introduced in the previous section.

To parallelize processing, SkinnerMT splits data in one specific table across worker threads. Doing so does not require physical data re-organization (which would be costly). Instead, choosing a split table merely assigns tuple offsets to different threads, determining

Algorithm 2 Parallel multi-way join algorithm.

```
1:  // Propose next tuple index in i-th table of order j, given
2:  // execution state e and split table s.
3:  function Next(e, j, i, s)
4:      // Do we select tuples in split table?
5:      if j_i = s then
6:          return next matching tuple in thread scope s.t. index g.t. e_{j_i}
7:      else
8:          return next matching tuple in table t with index g.t. e_{j_i}
9:      end if
10: end function
11: // Perform parallel multi-way join on query q using order j with split
12: // table s for up to B steps, working with execution state e and focusing
13: // on i-th table in join order.
14: procedure PMWJoin(q, j, s, R, B, e, i)
15:     // While budget left and join not finished
16:     while B > 0 ∧ i ≥ 0 do
17:         // Do tuples selected in first i tables join?
18:         if e_{j_1} ⋈ ... ⋈ e_{j_i} satisfies join predicates then
19:             if i = q.nrTables then
20:                 // Insert join result tuple
21:                 R ← R ∪ {r}
22:             else
23:                 // Advance to next table
24:                 i ← i + 1
25:             end if
26:         end if
27:         // Advance to next tuple in current table
28:         e_{j_i} ←Next(e, j, i, s)
29:         // No tuples left in current table?
30:         if e_{j_i} = −1 then
31:             // Backtrack to previous table
32:             i ← i − 1
33:         end if
34:         B ← B − 1
35:     end while
36: end procedure
37: // Resume join with order j and split table s for query q for B steps,
38: // inserting result tuples into set R and using progress trackers P.
39: procedure ResumeJoin(q, j, s, R, B, P)
40:     // Select progress tracker for split table
41:     p ← P[s]
42:     // Retrieve last execution state e for join order
43:     Restore(p, j, e)
44:     // Add join result tuples until budget runs out
45:     PMWJoin(q, j, s, R, B, e, 0)
46:     // Update progress tracker tree with last state
47:     Update(p, j, e)
48: end procedure
```

which subset of tuples they consider. The selection of the split table has significant impact on performance and is highly non-trivial. A cost-based approach for split table selection is discussed in the next subsection.

Algorithm 2 shows pseudo-code of the multi-way join algorithm. Worker threads execute specific join orders with specific split tables (both selected by the coordinator) for a fixed time budget of $\mathcal{B}$ steps. Execution is initiated by invoking the ResumeJoin function for one specific thread. First, the thread consults the progress tracker to

restore the last known state for that order. Note that SkinnerMT stores different progress trees for different split tables. Progress achieved using different split tables cannot be combined. Hence, each split table is associated with a separate progress tracker. Next, the thread executes the multi-way join for a limited time. Finally, the resulting execution state is stored in the progress tree. Procedure PMWJoin represents the core of the parallel join algorithm. During processing, this procedure manipulates four variables: e represents the execution state, indicating the index of the currently selected in each joined table, i is the size of the join order prefix for which a valid tuple combination (i.e., a combination of tuples satisfying all join predicates) has been found, R is the set of completed join result tuples, and $\mathcal{B}$ is the remaining join budget. Iterations continue until the join budget runs out or i reaches negative values (this indicates that a complete join result has been generated). In each iteration, the algorithm first verifies whether the currently selected tuples for the current join prefix (selected tuples in tables one to i in join order j, denoted as j_1 to j_i) satisfy all predicates (check in Line 18). If that is the case and the join "prefix" covers all tables (i.e., $i = q.nrTables$) then a complete join result tuple is found that is added to the result set R. If the currently selected tuple combination is valid but does not cover all tables, the prefix length (i) is incremented by one.

For the i-th table in join order (i.e., the end of the current join prefix), the algorithm advances to the next tuple by changing the corresponding tuple index in the execution state (i.e., e_{j_i}). Function Next recommends the next tuple index to consider. In case of equality join conditions, it uses previously generated indexes to find tuples that satisfy that join condition. It distinguishes the split table from other tables. If suggesting tuples in the split table, it selects the next tuple from the subset of tuples assigned to the current thread (this subset is defined by a hash value). Otherwise, it selects the next tuple from all tuples (with a tuple index larger than the currently selected tuple to avoid considering the same tuple combination twice).

Figure 3 illustrates parallel data processing in context. Worker threads execute the parallel multi-way join algorithm in parallel, using a join order and split table suggested by the coordinator. The join order is selected via reinforcement learning, summing up rewards received from different threads for the same join order. Processing finishes once all threads finish processing (i.e., condition $i < 0$ is satisfied in Algorithm 2) with the same split table. The split table is selected using the process described next.

4.2 Selecting Split Tables

First, we illustrate by the following example why split table choices matter for performance.

Example 4.1. Figures 4a to 4c illustrate processing of the same query and join order with different split tables. The query joins three tables, R, S, and T, via equality join conditions (connecting the only table columns). In the example, three threads process the parallel multi-way join algorithm discussed before. Different colors are associated with different threads. First, different cell background colors in the split table represent the scope of different threads. Second, colored numbers represent processing steps of different threads. More precisely, numbers represent the order in which different tuples are selected by the Next function in Algorithm 2. For instance, in Figure 4a, the green thread starts by selecting the first tuple in the first table (which is within its scope), then switches to the next table where it selects the first matching tuple that satisfies the (binary equality) join condition. Next, it advances to the last table in join order where it selects two matching tuples. After that, no matching tuples are left in the last table, prompting the thread to backtrack to the previous table (where it selects the next matching tuple in step number five).

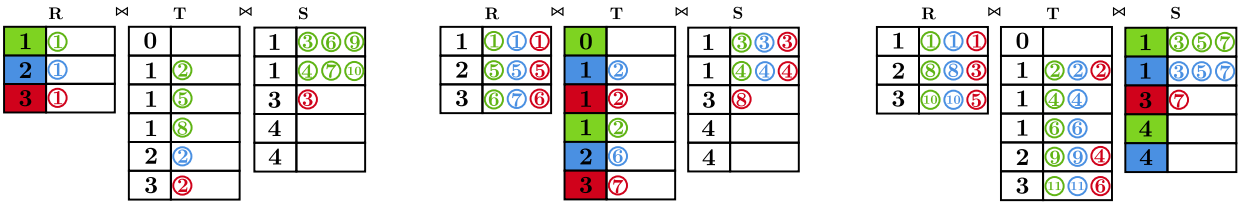
Note that the number of total steps differs across threads. This is due to the data not being completely uniform. In case of skew, selecting the wrong split table may cause a single thread to become the bottleneck. In the example, selecting table T (the second one in join order) minimizes the maximal number of steps over all threads. Hence, using this split table will be most efficient.

Up until the split table, all threads perform the same steps. This could be avoided by generating intermediate results once, then distributing the resulting tuples. However, as intermediate results are join order specific and cause overheads, SkinnerMT deliberately discards that option. This means that different threads perform redundant work before reaching the split table. If data are perfectly homogeneous, choosing the left-most table in join order therefore leads to optimal results. However, in practice, choosing a different table may improve performance due to skew. In the experiments, we will show that simple strategies do indeed not result in near-optimal performance.

Instead, we choose split tables according to the following model. Let $C(t)$ be the processing cost for a fixed query when choosing split table t. To finish query evaluation, each worker needs to finish its result partition for the current split table. Hence, it is $C(t) = \max_k(C_k(t))$ where $C_k(t)$ denotes processing cost for the k-th worker when splitting table t. Clearly, we want to minimize maximal per-worker costs.

For each worker, we can estimate remaining processing costs as follows. For a fixed split table, a worker finishes once it has advanced to the last tuple of the first table in join order. Hence, we can use the percentage of tuples covered in the first table as a coarse-grained measure of progress. By relating progress achieved so far with the time that has passed, we obtain an estimate of remaining processing costs. In a more fine-grained version, we also consider the number of tuples covered in other tables. E.g., the percentage of tuples covered in the second table in join order, scaled down by the cardinality of the first table (since it is specific to the currently selected tuple in the first table). We identify the slowest current worker based on those estimates. To select a split table, we only use statistics from that worker k^*.

The depth-first join algorithm switches focus between different tables (parameter i in Algorithm 2). We can decompose processing costs as $C_{k^*}(t) = \sum_i C^i_{k^*}(t)$ where $C^i_{k^*}(t)$ denotes processing costs associated with table i. The impact of split table choices on processing costs depends on data properties. Assuming that no reliable data statistics are available, this makes them hard to predict. We can however derive upper and lower bounds. Denote by $\underline{C_{k^*}}(t)$ lower and by $\overline{C_{k^*}}(t)$ upper bounds on processing costs when splitting t. At the very least, splitting table t scales down work required for that table by factor m (the number of workers). Hence, we obtain



(a) Split table R (b) Split table T (c) Split table S

Figure 4: Example of split table choices: three threads execute the join order $R \bowtie T \bowtie S$ with different split tables.

$$\overline{C_{k^*}}(t) = C^t/m + \sum_{i \neq t} C^i \tag{1}$$

where C^i denotes cost associated with table i without splitting. Threads perform work before the split table redundantly. Hence, we cannot hope to reduce costs associated with prior tables in join order. On the other side, assuming perfectly uniform data, cost associated with tables following the split table will decrease proportionally to the number of workers as well. Hence, we obtain

$$\underline{C_{k^*}}(t) = \sum_{i:\mathcal{A}(i,t,k^*)} (C^i/m) + \sum_{i:\neg\mathcal{A}(i,t,k^*)} C^i \tag{2}$$

as an optimistic cost bound, where $\mathcal{A}(i,t,k)$ is true if $i = t$ or if table i appears after table t in the join order executed by worker k. When selecting a split table, we prioritize the optimistic bound (Equation 2) and use the pessimistic bound (Equation 1) to break draws. Each worker collects per-sample statistics about the number of steps spent iterating over different tables. We use them to evaluate the prior formulas.

5 PARALLEL SEARCH

The data-parallel algorithm speeds up execution for the same join order via parallelization. This is helpful if that join order is of sufficiently high quality. On the other side, it does not help if execution time is prohibitive despite parallelization, due to a highly sub-optimal order. In such cases, it is better to exploit parallelism to speed up search for a near-optimal order. SkinnerMT supports approaches for parallel search, discussed in the following. Here, parallelism is used to execute more join orders in parallel, thereby speeding up convergence and reducing performance variance.

Figure 5 illustrates the main idea. Each worker is assigned to a search space partition. Instead of a single instance of the learning optimizer, one instance is spawned for each partition. At any point in time, each worker thread executes the most interesting join order, according to reinforcement learning, in each search space partition. The search space for join orders is represented as a tree by the UCT algorithm. Tree nodes correspond to join order prefixes. Edges connect one prefix to another, by selecting one more table. This means that each search tree level is associated with a join order prefix length. Different nodes at each level correspond to different join order prefixes.

This search space can be naturally partitioned by fixing a join order prefix. This means that different threads explore join orders that differ by the first few tables. In the simplest variant, we can

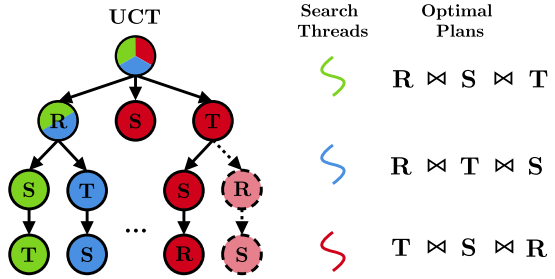


Figure 5: Dividing the search space: threads explore and execute join orders in parallel in different parts of the search space. Query evaluation ends once the first thread finishes.

divide the space of join orders uniformly. This means that, approximately, each thread searches a space of the same size. Processing terminates whenever the first thread terminates. Unlike in the case of data-parallel processing, threads do not consider different data splits. Instead, each thread is working towards generating a full join result.

The variant described above, called SP-U (for uniform partitioning) in the following, achieves some improvements via parallelization, as shown in Section 8. It suffers, however, from the following problem. Typically, good join orders are not distributed uniformly over different search space partitions. Instead, fixing the first few tables often has a significant impact on the average quality of the associated join orders. In such scenarios, the benefit of uniform parallel search is limited. Most of the threads are working within search space partitions that do not contain near-optimal join orders. Hence, those threads cause overheads (by using up cores and by reducing memory bandwidth available to threads making useful progress) without contributing to faster execution.

An improved version of the parallel search algorithm, called SP-A in the following, starts with uniform search space partitioning. However, as query processing proceeds, it adaptively changes the assignment from threads to search space partitions. More precisely, it increases the density of assigned threads for search space parts that receive high average reward values.

Algorithm 3 shows the recursive function used regularly to partition the search space based on rewards. It updates the field *threads*, associated with each UCT tree (if a single thread is assigned to the root of a sub-tree, the thread assignments within the sub-tree will not be updated further). Initially, all threads are assigned to

Algorithm 3 Assigning threads to search space partitions.

```
1:  // Recursively partition threads over child nodes of n.
2:  procedure PartitionThreads(n)
3:      // Do we have multiple threads to partition?
4:      if |n.threads| > 1 then
5:          // Collect unassigned threads
6:          U ← n.threads
7:          // Assign threads to child nodes, proportional to reward
8:          for c ∈ n.children in decreasing order of average reward do
9:              // Calculate desired number of threads
10:             m ← ⌈|n.threads| · c.r/(∑_{c∈n.children} c.r)⌉
11:             // Are enough unassigned threads left?
12:             if |U| > m then
13:                 c.threads ← pick m threads from U
14:             else if U ≠ ∅ then
15:                 c.threads ← pick max(|U| − 1, 1) threads from U
16:             else
17:                 c.threads ← pick one thread from previous child
18:             end if
19:             U ← U \ c.threads
20:         end for
21:         // Recursively partition threads for child nodes
22:         for c ∈ n.children do
23:             PartitionThreads(c)
24:         end for
25:     end if
26: end procedure
```

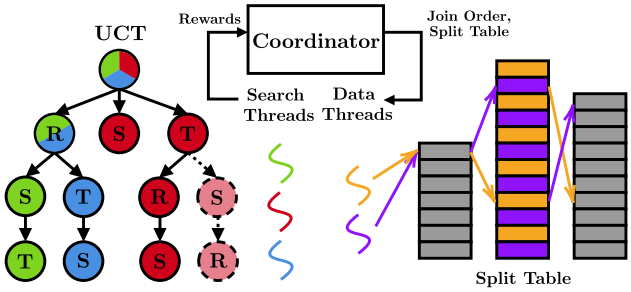


Figure 6: Hybrid algorithm: threads either explore search space partitions or execute join orders in parallel. The coordinator selects join orders and split tables for data threads, based on performance statistics collected by search threads.

the root node of the UCT tree. Then, Function PartitionThreads is invoked with the root node as parameter. In each invocation, the function tries to partition threads over child nodes to cover high-reward regions by more threads (field $c.r$ denotes average reward of a node c, $n.children$ denotes the set of n's child nodes). The procedure ensures that every part of the search tree has at least one thread assigned. It is recursive and partitions threads over more and more fine-grained parts of the UCT search tree.

6 HYBRID PARALLELISM

In the previous sections, we have seen that multi-core parallelism can be used to execute more join orders in parallel, or to execute one join order more quickly. In this section, we discuss hybrid versions compromising between those two extremes. Figure 6 illustrates the idea. We divide threads into two groups: search threads and data threads. The primary task of search threads is to explore the space of join orders. The primary task of data threads is to execute promising join orders in parallel.

Search threads are assigned to non-overlapping parts of the search space. They execute join orders to obtain performance estimates, then report those statistics back to the coordinator. The coordinator analyzes statistics collected by the search threads to identify the most promising join order overall (maximizing average reward). The coordinator assigns that join order to data threads, selecting a split table via the method discussed in Section 4.2. Processing ends either once either *all* of the data threads or *one* of the search threads finishes join processing. The latter case is rare as search threads do not use parallel processing.

In the simplest case (called "HP-F" in the following), the number of search threads and data threads remains fixed over time. If so, the number of search and data threads becomes an important tuning parameter. Using more search threads leads to similar performance tradeoffs, compared to parallel search. I.e., it increases performance robustness while average performance decreases. Using more data threads has the opposite effect and approaches the performance of data-parallel processing in the limit. We will explore those tradeoffs in the experiments in Section 8.

A more sophisticated version (called "HP-A" in the following) adapts the number of search and data threads over the course of query processing. Intuitively, increased exploration of join orders is most useful at the beginning of query execution. At that point, join order search has not yet converged. Discovering efficient join orders is more useful as they can be applied to a larger amount of (unprocessed) data. On the other side, parallel processing is most useful when applied to a near-optimal join order. This is why the adaptive version of the hybrid algorithm gradually transforms search to data threads.

Algorithm 4 shows the associated pseudo-code. The algorithm depends on a tuning parameter, γ, determining how quickly the rate of reassignments decreases (in the experiments, we use $\gamma = 10$). Initially, all threads are assigned as search threads (Line 7). The main loop ends once one of two conditions is satisfied: either one of the search threads terminates ($s.term$ is the termination of search thread s) or all data-parallel threads terminate for a specific split table ($d.term(s)$ indicates whether data thread d terminated for split table s). Iteratively, the coordinator retrieves the optimal join order found by the search threads (Line 12), calculates the best split table via the approach from Section 4.2 (Line 13) and instructs the data threads to execute with the corresponding join order and split table (while search thread continue exploration in the background). Periodically, search threads associated with the search space partitions with lowest average reward values are reassigned to become data threads (Lines 18 to 21). Their search space partitions are equally distributed over the remaining search threads.

7 ANALYSIS

We analyze the proposed algorithms formally. Section 7.1 analyzes time complexity. Section 7.2 focuses on space complexity.

Algorithm 4 Hybrid algorithm: coordinator assigns threads for exploring join orders and executing promising orders in parallel.

```
1: // Search optimal join orders and execute query q in parallel and
2: // periodically assign threads T to search and execution tasks.
3: procedure HPCoordinator(q, T)
4:     // Initialize samples until next order assignment
5:     b ← γ
6:     // Initialize search and data threads
7:     S ← T, D ← ∅
8:     // While join processing not finished
9:     while ∄t ∈ q.tables : ∀d ∈ D : d.term(t) ∧ ∄s ∈ S : s.term) do
10:        // Iterate until budget runs out
11:        for b iterations do
12:            j ← best join order found by search threads S
13:            s ← best split table for join order j
14:            Instruct data threads T to execute one episode with j and s
15:            In parallel, search threads S continue exploration
16:        end for
17:        // Reassign search threads if required
18:        if |S| > 1 then
19:            R ← pick half of search threads with lowest reward
20:            S ← S ∪ R
21:            D ← D \ R
22:            Partition search space across remaining search threads
23:        end if
24:        // Increase time until next reassignment
25:        b ← b · γ
26:    end while
27: end procedure
```

7.1 Time Complexity

We make several simplifying assumptions. First, we assume that execution time is proportional to the number of UCT samples. This seems reasonable as the number of atomic steps, executed by Algorithm 2 per thread and sample, is bounded by a constant budget $\mathcal{B}$. Second, we analyze parallel time complexity relative to the complexity of sequential processing, using the same processing algorithms. We simplify by modeling adaptive processing as two distinct phases: time T_C for converging to a (near-)optimal join order and time T_E for executing it. This model complies with observations with prior adaptive processing engines, namely the SkinnerDB system [29]: most execution time is spent with a single join order and UCT tree growth slows over time.

We analyze the time complexity of SP-U in the following. This requires reasoning about how a reduction in search space size influences convergence time of the UCT algorithm. We use the formula $B^{D/2}$, proposed as a pessimistic estimate [16]. Here, B is the branching factor and D the UCT tree depth. Also, we denote by n the number of joined tables and by m the number of threads.

Theorem 7.1. *SP-U is in $O(T_C/\sqrt{n^{\lfloor \log_n(m) \rfloor}} + T_E)$ time.*

Proof. By assigning workers to fixed join order prefixes, we partition the search space equally among workers. Join ordering has a branching factor in $O(n)$ as well as a tree depth of $O(n)$ (this analysis is pessimistic and neglects for instance the effects of avoiding Cartesian product heuristics). Partitioning the search space over m threads reduces the tree depth by at least $\lfloor \log_n(m) \rfloor$ levels. Hence, we expect convergence in $O(n^{(n-\lfloor \log_n(m) \rfloor)/2})$ and, assuming $T_C = n^{n/2}$, convergence is achieved in $O(T_C/\sqrt{n^{\lfloor \log_n(m) \rfloor}})$. We pessimistically assume that execution time is not reduced via progress sharing, compared to sequential execution. □

Intuitively, parallel search is most useful if the performance gap between join orders is significant and good join orders are sparse. We conduct another analysis of SP-U in this very scenario. We assume now that all join orders have negligible average reward while one single join order performs significantly better. Exploration essentially ends once the high-performance order has been tried for the first time and its performance was observed. We denote by $s = B^D$ the total size of the join order space.

Theorem 7.2. *If high-performance join orders are extremely sparse, SP-U is in $O(T_C/m + T_E)$ time.*

Proof. The UCT algorithm (as well as other reinforcement learning algorithms) selects actions with uniform random distribution, as long as observed rewards do not differ across actions. Hence, each join order selection corresponds to an independent random draw from the total space of join orders. With probability $1/s$, the high-performance join order is found and exploration ends. As draws are identically distributed and independent, and as the chance of finding the high-performance join order are small, the expected time to find the high-performance order can be modeled by an exponential probability distribution with $\lambda = 1/s$ and expected value $1/\lambda = s$. If the search space is partitioned uniformly across m threads, the mean decreases to s/m. □

Under the same assumptions, we can analyze performance variance. We denote by V_C and V_E the variance in time for convergence and execution phase.

Theorem 7.3. *If high-performance join orders are extremely sparse, SP-U has variance in $O(V_C/m^2 + V_E)$.*

Proof. Using the same reasoning as in the previous proof, we can model the time until finding a good join order via an exponential distribution with parameter $\lambda = 1/s$. The variance of an exponential distribution is given by $1/\lambda^2$. Hence, decreasing the size of the search space per thread (via uniform partitioning) reduces the variance of the convergence phase quadratically as a function of the number of threads (m). □

Next, we analyze the data-parallel processing algorithm (DP). We make several additional assumptions. For the final join order, we assume that cost associated with specific tables is either completely negligible or high enough to measure cost during each invocation of the join algorithm. This assumption entails a fixed choice for the split table. We also assume that all tables are large enough to partition work associate directly with their tuples equally over all workers. We denote by T_C and T_E sequential convergence and execution time, m is the number of workers. First, we analyze time complexity of the data parallel algorithm (DP) assuming uniform data in each table and a uniform distribution of processing cost over tables.

Theorem 7.4. *DP is in $O(T_C + T_E/m)$ time in the uniform data scenario.*

Proof. Cost is uniformly distributed over tables, hence no table has negligible cost. Hence, splitting the first table is the only choice minimizing optimistic cost. If splitting the first table, the size of all following intermediate results is scaled down by the number of workers due to uniform data. The size of the search space does not change, compared to sequential intra-query learning. Hence, we assume the same convergence time T_C. □

Next, we assume highly skewed data in one table and uniform data in the others. We assume that one heavy hitter tuple has many matches in that table, while other tuples have none. We assume that cost associated with the skewed table is decisive, meaning that cost associated with prior tables in the final join order is negligible.

Theorem 7.5. *DP is in $O(T_C + T_E/m)$ time in the single skewed table scenario.*

Proof. We assume convergence time T_C for the same reasons as before. Let t be the skewed table, producing many matches for the heavy hitter. The slowest worker handles the heavy hitter. Choosing t as well as prior tables in the final join order minimizes optimistic cost. However, only choosing t minimizes pessimistic cost among them (since cost before the skewed table is negligible). Hence, t is chosen as split table. We assume uniform data, except for the split table. Hence, cost associated with all intermediate results starting from t is scaled down by m. □

The analysis provides some evidence that parallel search can perform well in scenarios where the join order space is large and good join orders are sparse. In particular, it explains why search space parallelism can reduce variance under these circumstances. For data-parallel processing, the analysis shows potential for significant speedups via parallelization, even in the presence of skewed data. The findings translate to the hybrid algorithm as well (which may inherit performance properties from both algorithms, depending on the number of search and data threads). All results are based on simplifying assumptions and will be verified in experiments.

7.2 Space Complexity

Next, we analyze space complexity. We denote by r the result size, and by T total run time, measured as the number of UCT samples (to account for limited tree growth as a function of execution time). The data structures we consider grow gradually over time. Hence, we derive bounds based on run time as well as on query properties.

Lemma 7.6. *The progress tree is in $O(\min(n \cdot T, n!))$ space.*

Proof. The maximal extent of the progress tree is limited by the number $n!$ of join orders for n tables. More precisely, it is the maximal number of leaf nodes. It is only achieved if each tree node has the maximally possible branching factor. As the branching factor is at least two for inner nodes, the number of leaf nodes dominates the aggregated number of inner nodes. At the same time, the progress tree can grow by at most n nodes after each UCT sample (if a new join order was tried). Hence, the tree size is also in $O(T \cdot n)$. □

Lemma 7.7. *The UCT tree is in $O(\min(T, n!))$ space.*

Proof. The UCT tree contains at most one leaf node per join order. Also, it is expanded by at most one node per sample. □

Lemma 7.8. *The join result set is in $O(r \cdot n)$ space.*

Proof. The join result is stored as a set of vectors, indicating tuple indices for each of the n tables. Due to set semantics, it contains each result tuple only once. □

Next, we derive space bounds for all parallel processing strategies.

Theorem 7.9. *Parallel search is in $O(r \cdot n + \min(T \cdot n, n!))$ space.*

Proof. Different search threads share the same join result whose size is given by Lemma 7.8. Different threads cover non-overlapping search space partitions. Hence, the sum of the associated UCT and progress tree partitions do not exceed the bounds given in Lemma 7.6 and 7.6. □

Theorem 7.10. *Data-parallel processing is in $O(r \cdot n + n \cdot \min(T, n!))$ space.*

Proof. Different worker threads generate non-overlapping parts of the join result. Hence, the bound from Lemma 7.8 applies. Different progress trees are stored for different split tables. Hence, space for storing progress is in $O(n \cdot n!)$ but still bounded by $O(T \cdot n)$ since only one split table is tried per sample. □

Corollary 7.11. *The hybrid algorithm is in $O(r \cdot n + n \cdot \min(T, n!))$ space.*

Proof. This follows from the previous two theorems (as the hybrid algorithm can be tuned to behave like any of the two analyzed algorithms). □

8 EXPERIMENTAL EVALUATION

We describe the experimental setup in Section 8.1. In Section 8.2, we analyze performance of different parallelization strategies in SkinnerMT's join phase. In Section 8.3, we compare SkinnerMT against other database systems and adaptive processing baselines.

8.1 Experimental Setup

We evaluate on three benchmarks: the TPC-H benchmark [27], the join order benchmark (JOB) [12], and JCC-H [7]. Those benchmarks cover different degrees of challenge for traditional query optimizers, ranging from TPC-H (easy optimization due to uniform data), over JOB (moderately difficult optimization due to slightly skewed data), up to JCC-H (hard due to highly skewed data). We omit three TPC-H and JCC-H queries, Q13, Q15, and Q22, from the following comparisons as they are currently not supported by SkinnerMT (lack of support for outer joins, views, and substring functions). We use scaling factor 10 for TPC-H and JCC-H. SkinnerMT is a fork of the original SkinnerDB system[1] which does not parallelize the join phase.

For performance, we compare algorithms in terms of per-query and per-benchmark run time (average of ten runs with one warmup run). We calculate speedups by the ratio of sequential to parallel join phase time for each performance metric. For robustness, we

[1] https://github.com/cornelldbgroup/skinnerdb

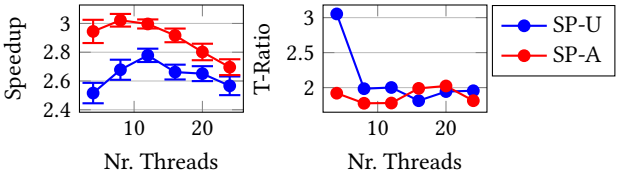


Figure 7: Comparison of parallel search strategies.

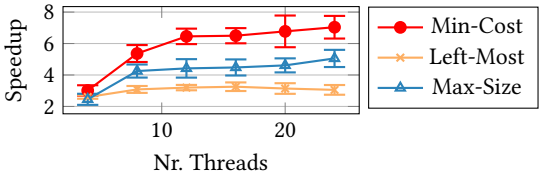


Figure 8: Comparing split table selection methods.

report the ratio of maximal to minimal join time for each query over ten runs. We show 95% confidence bounds for all plots showing arithmetic averages (confidence bounds are not applicable to plots showing ratios). We use a join budget of $\mathcal{B} = 500$ steps per time slice. We use a reward function that combines input and output reward with weights 0.5 respectively. All experiments were conducted on a 24-core server (two 2.30 GHz 12-core Intel(R) Xeon(R) Gold 5118 CPUs) running Ubuntu 20.04.4 LTS and the OpenJDK 64-Bit Server JVM 15.0.2. The total DRAM size is 252 GB. In order to reduce garbage collection (GC) overheads, we use Epsilon GC[1] with the maximum heap space of 200 GB for performance testing.

8.2 Join Performance

Figure 7 compares approaches for parallel search on JOB in terms of time (Speedup) by parallelization. We compare the uniform (SP-U) and adaptive (SP-A) search space partitions described in Section 5. SP-A outperforms SP-U in performance and robustness, especially when the number of threads is limited. Using more threads, the gap decreases. We use SP-A as the representative for search space parallelism in the following experiments.

Figure 8 compares variants of the data parallel (DP) algorithm on JOB. We vary the policy according to which the split table is selected. We compare our main variant (Min-Cost) against selecting the left-most table (Left-Most) and selecting the largest table (Max-Size). Cost-based splitting leads to optimal performance.

DP stores join order state for different threads and for different split tables separately. Space-efficient progress tracking is therefore crucial. Figure 9 compares space consumption between the newly proposed tree-based and the progress tracker used in the prior SkinnerDB system. Clearly, the new tracker reduces space consumption significantly (in particular on JOB which tends to have larger search spaces than the other benchmarks). As access times are equivalent, we use the new version in the following.

Figure 10 compares variants of the hybrid algorithm in terms of speedups and robustness. For HP-F, the number of search threads remains fixed (and is shown on the x-axis). Increasing the number of search threads trades speedups for more robustness. HP-A is the adaptive, hybrid algorithm described in Section 6. For most settings,

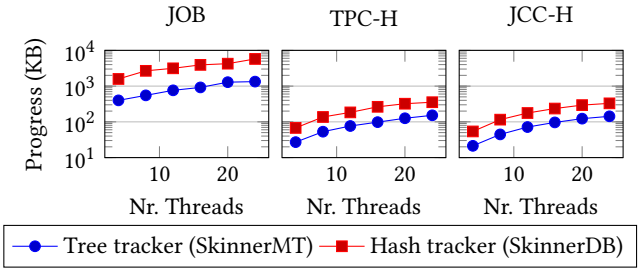


Figure 9: Impact of space-efficient progress tracker for data parallel algorithm.

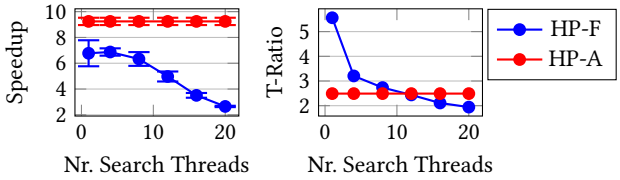


Figure 10: Fixed number of threads and vary number of search threads.

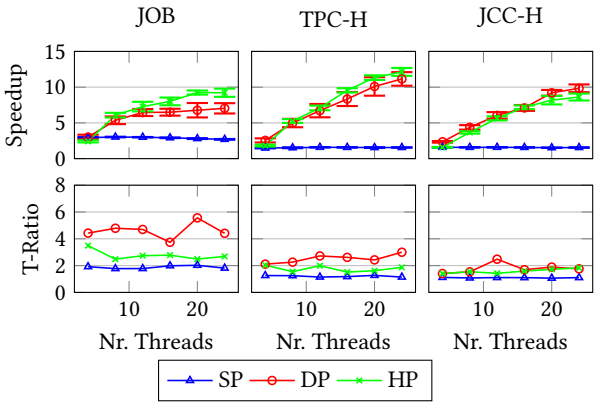


Figure 11: Time speedups for different parallel join strategies.

it achieves dominant tradeoffs between speedups and robustness. For the maximal number of search threads, HP-F is equivalent to search parallelism and achieves better robustness.

Figure 11 compares the best variant of each proposed algorithm for all three benchmarks. Generally, increasing the degree of parallelism leads to speedups for DP and HP. For SP, only a limited number of threads can be exploited efficiently. In terms of robustness, SP reduces per-query run time variance to at most factor two over all queries. DP and HP achieve similar speedups on most benchmarks while HP reduces variance. On JOB, the benchmark with the largest join order space, HP improves speedups, compared to DP, as well. As HP is overall preferable, it is used for the end-to-end comparison with other systems in the next subsection.

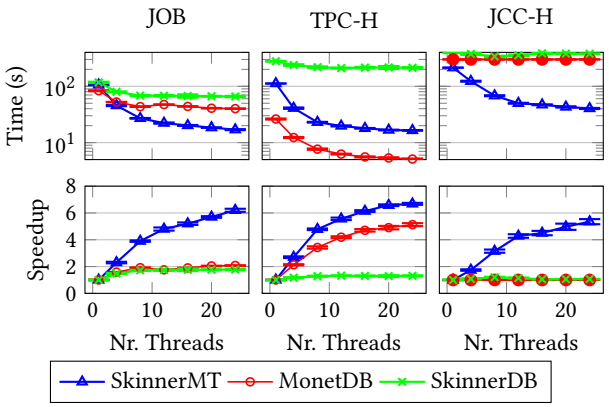


Figure 12: End-to-end performance of different systems.

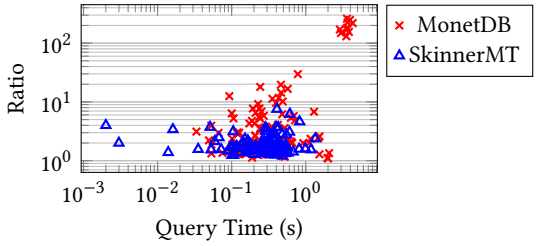


Figure 13: Performance robustness for JOB queries with regards to semantically equivalent query rewritings.

8.3 End-to-End Comparison

Figure 12 compares end-to-end performance of SkinnerMT to HP, MonetDB [8] (DB Server 4 v11.43.13), and SkinnerDB. We use a timeout of 300 seconds per benchmark and baseline.

MonetDB performs best on the TPC-H benchmark. Here, cardinality estimation is easy due to uniform data. Adaptive processing leads to overheads that do not pay off in this scenario. For JOB, MonetDB and SkinnerMT are similarly fast. Finally, SkinnerMT is the best approach for the JCC-H benchmark where query optimization is hard due to skewed data (the baselines hit our per-benchmark timeout). SkinnerMT clearly benefits from parallelization, achieving comparable speedups to MonetDB (except for JCC-H where MonetDB reaches the timeout).

The discussion of robustness has so far focused on mitigating performance variance, caused by adaptive plan selection. Traditional systems which select and execute a single query plan do not suffer from this variance. However, a well documented problem with such approaches are drastic performance changes, caused by different plan selections when applying small changes to queries. In the extreme case, those changes do not even change the query semantics. In a final experiment, we rewrite JOB queries by duplicating unary predicates. This does not change the query semantics but impacts selectivity estimates by the optimizer. SkinnerMT relies on run time feedback alone to select join orders. It is therefore insensitive to the way in which a query is written. MonetDB, however, shows significant variations in processing performance. Figure 13 compares the two systems (each dot represents one query, calculating the maximal time ratio over ten runs).

9 RELATED WORK

SkinnerMT explores the use of multi-core parallelism for adaptive processing. In that, it differs from SkinnerDB [29], the closest prior work, which performs sequential joins. Our work is complementary to other research on using learning for query optimization [15, 17, 19, 22, 25, 34]. Prior work implements inter-query learning: knowledge gained from past queries is applied to optimize future queries. As opposed to our approach, this requires representative training workloads. Traditional [24] and re-optimization [5, 6, 35] requires initial data statistics.

Our work relates to prior work on adaptive query processing strategies for data streams [3, 9, 30, 32]. We compare against one representative. Our work also connects to prior work on parallelizing query optimization [13, 14, 28, 33, 36] as well as query execution [2, 10]. However, prior work on data-parallel execution typically assumes that join order and partitioning are fixed. Instead, we adapt join orders and partitioning dynamically, based on reinforcement learning and a dynamic, cost-based partitioning strategy. Prior work on parallelizing query optimization assumes that optimization operates on an intermediate result lattice. Instead, SkinnerMT operates on a partial UCT search tree. Our work is complementary to other work using specialized multi-way join algorithms for different purposes than fast join order switching (e.g., fast approximation [11, 18] or worst-case guarantees [21, 31]).

10 CONCLUSION

We presented the SkinnerMT system: an adaptive processing engine that exploits multi-core parallelism in multiple ways. The experiments show that SkinnerMT can realize different tradeoffs between average performance and performance robustness, depending on the selected processing strategy. Compared to baselines, SkinnerMT achieves high performance on benchmarks where optimization is difficult. Furthermore, its performance is more robust to query rewriting than non-adaptive systems.

Going forward, we plan to transfer the adaptive, parallel processing strategies tested in SkinnerMT to other systems, too. High-level approaches such as parallel plan search and hybrid variants (exploiting parallel search and parallel processing) should be transferable to other adaptive processing systems as well, in particular adaptive systems that use multi-way join algorithms. Finally, we plan to explore strategies for trading robustness for performance via parallel multi-order execution for traditional database systems as well.

REFERENCES

[1] [n.d.]. https://openjdk.java.net/jeps/318
[2] MC Albutiu, Alfons Kemper, and T Neumann. 2012. Massively parallel sort-merge joins in main memory multi-core database systems. *VLDB* 5, 10 (2012), 1064–1075. http://dl.acm.org/citation.cfm?id=2336678
[3] Ron Avnur and Jm Hellerstein. 2000. Eddies: continuously adaptive query processing. In *SIGMOD*. 261–272. https://doi.org/10.1145/342009.335420
[4] Shivnath Babu. 2005. Adaptive query processing in the looking glass. In *CIDR*. 238 – 249. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.98.3279
[5] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive re-optimization. In *SIGMOD*. 107–118. https://doi.org/10.1145/1066157.1066171
[6] P. Bizarro, N. Bruno, and D.J. DeWitt. 2009. Progressive parametric query optimization. *KDE* 21, 4 (2009), 582–594. https://doi.org/10.1109/TKDE.2008.160

[7] Peter Boncz, Angelos Christos Anatiotis, and Steffen Kläbe. 2018. JCC-H: Adding join crossing correlations with skew to TPC-H. *LNCS* 10661 (2018), 103–119. https://doi.org/10.1007/978-3-319-72401-0_8

[8] P.A. Boncz, Kersten M.L., and Stefacn Mangegold. 2008. Breaking the memory wall in MonetDB. *CACM* 51, 12 (2008), 77–85.

[9] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2006. Adaptive Query Processing. *Foundations and Trends® in* 1, 1 (2006), 1–140. https://doi.org/10.1561/1900000001

[10] David J Dewitt, Donovan Schneider, and Rick Rasmussen. 1990. The Gamma database machine project. *KDE* 2, 1 (1990), 44–62.

[11] Alin Dobra, Chris Jermaine, Florin Rusu, and Fei Xu. 2009. Turbo-Charging Estimate Convergence in DBO. *PVLDB* 2, 1 (2009), 419–430.

[12] Andrey Gubichev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *PVLDB* 9, 3 (2015), 204–215.

[13] Wook-Shin Han, Wooseong Kwak, Jinsoo Lee, Guy M. Lohman, and Volker Markl. 2008. Parallelizing query optimization. In *VLDB*. 188–200. https://doi.org/10.14778/1453856.1453882

[14] Wook-Shin Han and Jinsoo Lee. 2009. Dependency-aware reordering for parallelizing query optimization in multi-core CPUs. In *SIGMOD*. 45–58. https://doi.org/10.1145/1559845.1559853

[15] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: estimating correlated joins with deep learning. In *CIDR*. arXiv:1809.00677 http://arxiv.org/abs/1809.00677

[16] Levente Kocsis and C Szepesvári. 2006. Bandit based monte-carlo planning. In *European Conf. on Machine Learning*. 282–293. http://www.springerlink.com/index/D232253353517276.pdf

[17] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv:1808.03196* (2018). arXiv:1808.03196 http://arxiv.org/abs/1808.03196

[18] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. *SIGMOD* 46, 1 (2016), 615–629. https://doi.org/10.1145/2882903.2915235

[19] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In *aiDM*. 3. arXiv:arXiv:1803.00055v2

[20] Prashanth Menon, Amadou Ngom, Lin Ma, Todd C. Mowry, and Andrew Pavlo. 2020. Permutable compiled queries: Dynamically adapting compiled queries without recompiling. *Proceedings of the VLDB Endowment* 14, 2 (2020), 101–113. https://doi.org/10.14778/3425879.3425882

[21] Hung Q Ngo and Christopher Ré. [n.d.]. Beyond Worst-case Analysis for Joins with Minesweeper Categories and Subject Descriptors. ([n. d.]), 234–245.

[22] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *DEEM*. arXiv:1803.08604 http://arxiv.org/abs/1803.08604

[23] Vijayshankar Raman Vijayshankar Raman, A. Deshpande, and J.M. Hellerstein. 2003. Using state modules for adaptive query processing. In *ICDE*. 353–364. https://doi.org/10.1109/ICDE.2003.1260805

[24] PG G Selinger, MM M Astrahan, D D Chamberlin, R A Lorie, and T G Price. 1979. Access path selection in a relational database management system. In *SIGMOD*. 23–34. http://dl.acm.org/citation.cfm?id=582095.582099

[25] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *PVLDB*. 19–28.

[26] RS Richard S Sutton and Andrew G Ag Barto. 1998. *Reinforcement learning: an introduction*. https://doi.org/10.1109/TNN.1998.712192

[27] TPC. 2013. TPC-H Benchmark. http://www.tpc.org/tpch/

[28] Immanuel Trummer and Christoph Koch. 2016. Parallelizing query optimization on shared-nothing architectures. In *VLDB*. 660–671.

[29] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: regret-bounded query evaluation via reinforcement learning. In *SIGMOD*. 1039–1050.

[30] Kostas Tzoumas, Timos Sellis, and Christian S Jensen. 2008. *A reinforcement learning approach for adaptive query processing*. Technical Report.

[31] Todd L. Veldhuizen. 2012. Leapfrog Triejoin: a worst-case optimal join algorithm. (2012), 96–106. https://doi.org/10.5441/002/icdt.2014.13 arXiv:1210.0481

[32] Stratis D Viglas, Jeffrey F Naughton, and Josef Burger. 2003. Maximizing the output rate of multi-way join queries over streaming information sources. In *PVLDB*. 285–296. http://dl.acm.org/citation.cfm?id=1315451.1315477

[33] Florian M. Waas and Joseph M. Hellerstein. 2009. Parallelizing extensible query optimizers. In *SIGMOD*. 871–882. https://doi.org/10.1145/1559845.1559938

[34] Lucas Woltmann, Claudio Hartmann, Maik Thiele, and Dirk Habich. 2019. Cardinality estimation with local deep learning models. In *aiDM*.

[35] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. 2016. Sampling-based query re-optimization. In *SIGMOD*. 1721–1736. arXiv:1601.05748 http://arxiv.org/abs/1601.05748

[36] Wanli Zuo, Yongheng Chen, Fengling He, and Kerui Chen. 2011. Optimization strategy of top-down join enumeration on modern multi-core CPUs. *Journal of Computers* 6, 10 (oct 2011), 2004–2012. https://doi.org/10.4304/jcp.6.10.2004-2012