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, 16(4): 905 - 917, 2022.
doi:10.14778/3574245.3574272

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/cornelldbgroup/skinnerdb/tree/skinnermt.

1 INTRODUCTION

Traditional query optimizers select join orders based on coarse-grained data statistics and many simplifying assumptions [34]. 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 [23, 25, 26, 31]. 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 [2, 3, 19, 33] 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 [2], allowing for longer convergence times, recent work [28, 39] 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 [39], 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. 16, No. 4 ISSN 2150-8097.
doi:10.14778/3574245.3574272

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 [39] (which is why prior work on adaptive processing often opts for specialized join operators [2]). 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 [37], due to uniform data) over benchmarks where optimization is moderately difficult (join order benchmark [14]), up to benchmarks where optimization is hard (JCC-H, due to highly skewed data [8]). 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 and Postgres, SkinnerMT performs better on benchmarks where query optimization is difficult

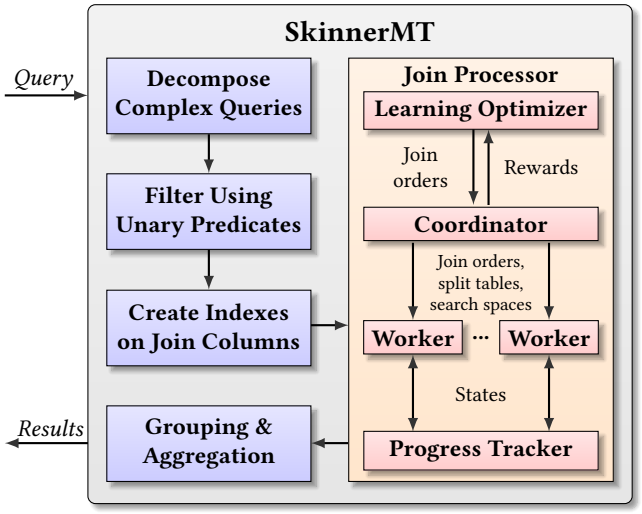


Figure 1: Overview of SkinnerMT.

(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:

- 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. A precise definition of the reward function is given in Section 4.1.

More precisely, the learning optimizer uses the UCT algorithm to choose interesting join orders to explore [22], 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 and starting with an empty tree. This is crucial to avoid prohibitive overheads when creating trees representing the entire join order space for large queries. It also means that the initial decisions are random while the algorithm converges to optimal decision over time [22].

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.

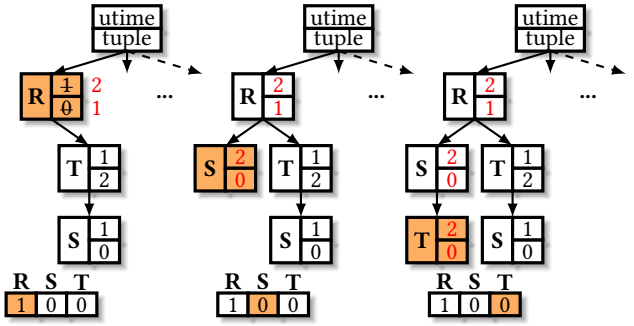
Algorithm 1 Retrieving and updating join execution state.

```
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
```

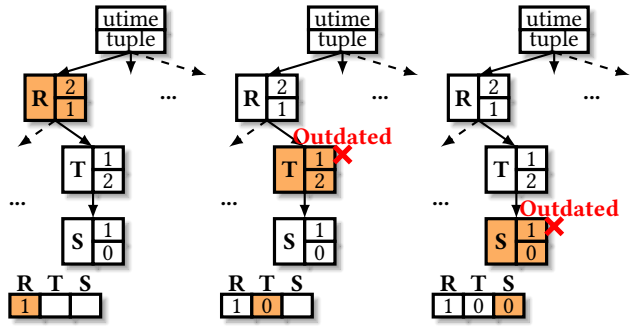
3 TRACKING PROGRESS

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 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



(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.

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 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).

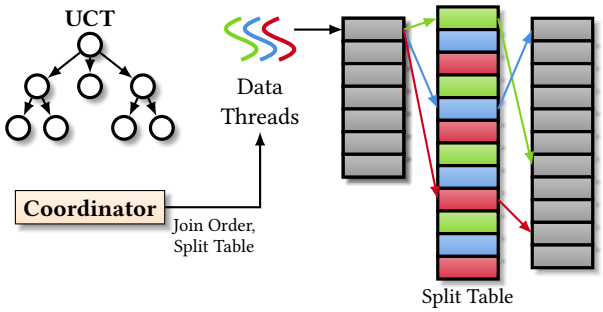


Figure 3: Worker threads execute a parallel multi-way join algorithm, using join orders and split tables selected by the coordinator.

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 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

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
```

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. Processing finishes once all threads finish processing (i.e., condition $i < 0$ is satisfied in Algorithm 2) with the same split table. The join order is selected via reinforcement learning, using the UCT algorithm described in Section 2. This algorithm is guided by a reward function that it tries to maximize. Here, rewards represent the quality of join orders. Reward samples are calculated after each invocation of Procedure ResumeJoin, using the same reward function as SkinnerDB [39]. The reward is the average of two components. The first component is the number of result tuples added to set R during the invocation. As all join orders must ultimately generate the same result, join orders generating more results per time unit are, in average, preferable. The second component is based on the speed at which the counters in vector e (representing the currently selected tuple for each table) change. Denoting by e and e' the execution start before and after the invocation, we denote by $\delta_i = e'_{j_i} - e_{j_i}$ the number of tuples by which we advanced for the i-th table in join order. Using c_i for the cardinality of the i-th table in join order, $\sum_{1 \leq i \leq n} \delta_i / (\prod_{1 \leq k < i} c_k)$ is the second component of the reward function. This measure of progress does not depend on the number of result tuples and is useful even for queries with empty result. For any join order, its values sum up to one over the course of the entire execution [39]. Hence, again, join orders with higher average rewards execute faster. Note that both reward metrics may vary across different invocations of the same join order (due to non-homogeneous data, for instance). The UCT algorithm can handle such variance and identifies join orders with highest average reward. 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.

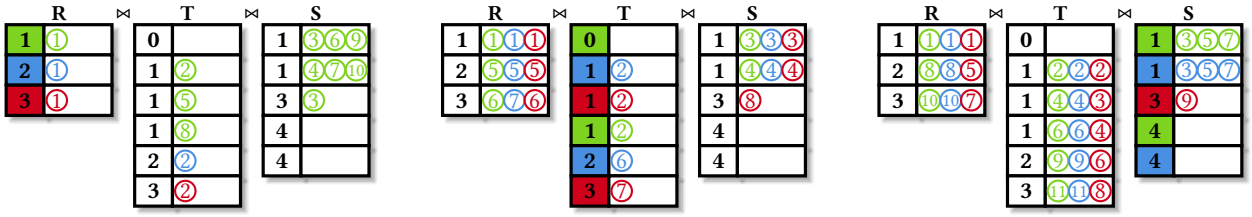
Example 4.1. Figures 4a to 4c illustrate processing of the same query and join order with different split tables. The query joins three tables, R, S, and T, via equality join conditions (connecting the only table columns). In the example, three threads process the parallel multi-way join algorithm discussed 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.

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^*.



(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.

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

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

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

SkinnerMT can exploit multi-threading to explore more join orders in parallel. This can be helpful if good join orders are hard to find. 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.

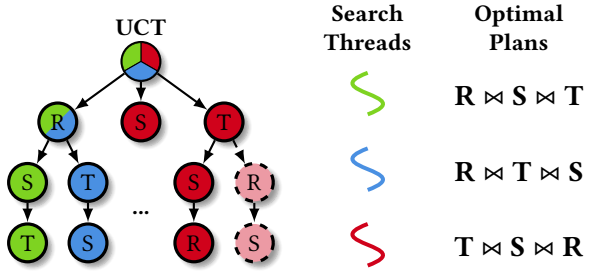


Figure 5: Dividing the search space: threads explore and execute join orders in parallel in different parts of the search space. Query evaluation ends once the first thread finishes.

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 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 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.

An improved version of the parallel search algorithm, SP-A, starts with uniform search space partitioning. Then, it adaptively changes the assignments from threads to search space partitions. Assignments are based on reward statistics in the UCT search tree. Ideally, the number of search threads per partition is proportional to average rewards observed in that partition. During re-assignments, SP-A successively divides threads into smaller and smaller groups that are associated with longer and longer join order prefixes (i.e., more and more fine-grained search space partitions). This process ends once the number of threads becomes too small to cover all possible expansions of a given join order prefix. As re-assignments create overheads, we increase the number of episodes between

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
```

reassignments according to an exponential scheme, waiting for α^i episodes between the i-th and $i-1$-th reassignment.

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 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

We discuss a hybrid algorithm combining parallel search and execution. 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 once either *all* of the data threads or *one* of the search

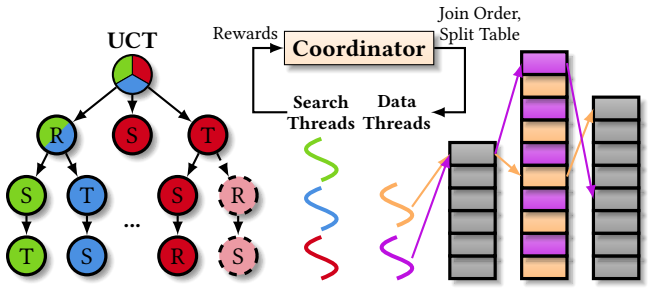


Figure 6: Hybrid algorithm: threads either explore search space partitions or execute join orders in parallel. The coordinator selects join orders and split tables for data threads, based on performance statistics collected by search threads.

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 and data threads remains fixed. Using more search threads leads to similar performance tradeoffs, compared to parallel search. Using more data threads has the opposite effect and approaches the performance of data-parallel processing in the limit.

A second version, "HP-A", gradually transitions from parallel search to parallel execution over the course of query execution. Intuitively, parallel execution is more useful after efficient join orders are known. Initially, HP-A partitions the search space equally among all search threads. In regular intervals, the algorithm selects half of the search threads, associated with the search space partitions with lowest average rewards, and re-assigns them for parallel join order execution (executing join orders selected by other search threads). This approach is similar, in spirit, to reinforcement learning variants that periodically discard actions with sub-optimal rewards (e.g., the sequential halving algorithm [20]). Ultimately, join order search focuses on a single search space partition, explored by a single search thread. The other threads execute selected join orders in parallel.

Algorithm 4 shows the associated pseudo-code. The algorithm depends on a tuning parameter, γ, determining how quickly the rate of reassignments decreases. 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).

7 ANALYSIS

At the start of execution, the UCT algorithm selects join orders with uniform random distribution. Over time, it converges to optimal

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, assign partitions
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 D 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:         end if
23:         // Increase time until next reassignment
24:         b ← b · γ
25:     end while
26: end procedure
```

(i.e., maximum reward) join orders [22]. To analyze the impact of parallelization, we assume a simplifying model in which that transition happens instantaneously. This divides query execution into two phases: finding optimal orders via random exploration and executing them. The first phase ends once an optimal join order is found (i.e., we simplify by assuming query execution does not end with sub-optimal orders).

We denote by T_C^S (time for convergence) and T_E^S (time for execution) random variables modeling time needed, without parallelization, for the first and second phase respectively (for an arbitrary but fixed query in the following). Consistent with that model, we assume that T_C^S follows an exponential distribution [5], modeling time until a rare event occurs. The distribution is parameterized by a parameter λ, capturing the event rate. Here, we consider discovering an optimal join order via random exploration a rare event and set λ to the ratio of optimal join orders within the join order space (i.e., $\lambda = n^*/n$ where n^* and n are the number of optimal and all join orders respectively). Due to the general properties of the exponential distribution, the expected value of T_C^S is therefore $1/\lambda$ and its variance $1/\lambda^2$. We assume that T_E^S follows a normal distribution (a common assumption in the domain of non-adaptive query execution time prediction [15, 47]) with mean μ and variance σ^2. For analyzing variance, we assume that T_C^S and T_E^S are independent random variables (as the time required for finding a join order does not directly relate to its execution time).

Next, we analyze how parallelization via different strategies, namely SP-U (parallel search with uniform partitioning), DP-L (data-parallel execution splitting left table in join order), and HP-F (hybrid parallelism with a fixed number of search and parallel processing threads), influences expected time ($\mathbb{E}$) and variance ($\mathbb{V}$). We denote the number of worker threads by m. We simplify by neglecting inter-thread synchronization overheads (which are shown to be moderate in Section 8).

Theorem 7.1. *SP-U has expected time* $\mathbb{E}(T_C^S)/m + \mathbb{E}(T_E^S)$ *and variance* $\mathbb{V}(T_C^S)/m^2 + \mathbb{V}(T_E^S)$.

Proof. We uniformly partition the search space across m threads. Denote by λ_1 to λ_m the ratio of optimal join orders in the corresponding search space partition. It is $\lambda = \sum_i \lambda_i/m$ (the average ratio over all equal-sized partitions equals the ratio in the entire search space). Threads proceed independently and execution finishes once any thread discovers an optimal order and finishes its execution. Variable T_{Ci} for $1 \leq i \leq m$ models time until thread i finds an optimal join order. It follows an exponential distribution with parameter λ_i. Then, $T_C^P = \min_{1 \leq i \leq m}(T_{Ci})$ models time until the first thread finds an optimal join order. The minimum of independent, exponentially distributed random variables follows an exponential distribution as well [6]. More precisely, T_C^P follows an exponential distribution with parameter $\sum_i \lambda_i = \lambda \cdot m$. The expected time before convergence is therefore $\mathbb{E}(T_C^P) = 1/(\lambda \cdot m) = \mathbb{E}(T_C^S)/m$, the variance $\mathbb{V}(T_E^P) = 1/(\lambda \cdot m)^2 = \mathbb{V}(T_E^P)/m^2$. As different threads execute joins independently, parallel execution time T_E^P follows the same distribution as T_E^S. It is $\mathbb{E}(T_C^P + T_E^P) = \mathbb{E}(T_C^P) + \mathbb{E}(T_E^P)$ and $\mathbb{V}(T_C^P + T_E^P) = \mathbb{V}(T_C^P) + \mathbb{V}(T_E^P)$ as T_C^P and T_E^P are independent. □

For the next theorems, we assume that data is skew-free.

Definition 1. *A database is skew-free with regards to the join* $R_1 \bowtie \ldots R_n$ *if replacing relation* R_1 *by a subset* R_S *of rows with cardinality ratio* $|R_S|/|R_1| = r$ *scales down the sizes of all intermediate results proportionally (e.g.,* $|R_S \bowtie R_2|/|R_1 \bowtie R_2| = r$ *and* $|R_S \bowtie R_2 \bowtie R_3|/|R_1 \bowtie R_2 \bowtie R_3| = r$*).*

We assume that processing time per thread is proportional to the total number of intermediate result rows it processes. This is consistent with the definition of the cost budget in Algorithm 2 (reducing the budget by one whenever the next intermediate result tuple is selected) and the classical C_{out} cost metric [10].

Theorem 7.2. *DP-L has expected time* $\mathbb{E}(T_C^S) + \mathbb{E}(T_E^S)/m$ *and variance* $\mathbb{V}(T_C^S) + \mathbb{V}(T_E^S)/m^2$.

Proof. DP-L does not explore more join orders in parallel, compared to the sequential algorithm. Hence, convergence time does not change. On the other side, DP-L executes join orders in parallel on different data subsets. While executing an optimal order, DP-L selects the left-most table as split table. Hence, the tuples in the left-most table are equally partitioned across threads. For skew-free data, this means that the sizes of all intermediate results are partitioned equally (i.e., scaled down by factor m) as well. We assume that execution cost depends on the number of tuples, therefore parallel execution time is $T_E^P = T_E^S/m$. Hence, $\mathbb{E}(T_E^P) = \mathbb{E}(T_E^S)/m$ and $\mathbb{V}(T_E^P) = \mathbb{V}(T_E^S)/m^2$. Expected values and variance of T_C^P and T_E^P can be added since we assume independence. □

For the analysis of HP-F, m_S denotes the number of search threads and m_D the number of data-parallel processing threads.

Theorem 7.3. *HP-F has expected time* $\mathbb{E}(T_C^S)/m_S + \mathbb{E}(T_E^S)/m_D$ *and variance* $\mathbb{V}(T_C^S)/m_S^2 + \mathbb{V}(T_E^S)/m_D^2$.

Proof. We combine reasoning from prior theorems. As HP-F explores m_S join orders in parallel, it shortens expected convergence time and the associated variance. As it executes the optimal order using m_D threads in parallel, it shortens expected time and reduces variance in the second (execution) phase as well. □

Based on those results, we expect SP to perform better, compared to DP, on queries where sequential execution time is dominated by time for finding optimal join orders (T_C). On the other side, DP should perform better on queries where sequential execution time is dominated by the time for executing join orders (T_E), not searching them. HP combines aspects from both algorithms and we expect it to perform well with both types of queries. We present further analysis in our extended technical report.

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. We include the detailed results in an extended technical report [44].

8.1 Experimental Setup

We evaluate on three benchmarks: the TPC-H benchmark [37], the join order benchmark (JOB) [14], and JCC-H [8]. 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 of 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), reporting 95% confidence bounds. We calculate speedups by the ratio of sequential to parallel join phase time for each performance metric. We measure robustness via the T-Ratio: the ratio of maximal to minimal run time over ten runs. I.e., if RT with $|RT| = 10$ is the set of run times for a specific query, the T-ratio is $\max(RT)/\min(RT)$. For each benchmark, we report the arithmetic average of T-ratios over all queries. We use a join budget of $\mathcal{B} = 500$ steps per time slice, set α and γ to 10. 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 [30] with the maximum heap space of 200 GB for performance testing.

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

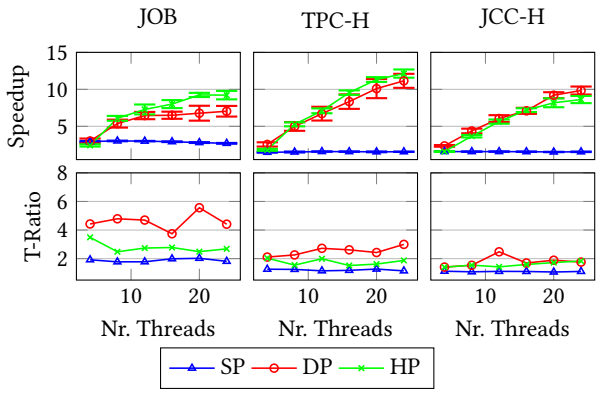


Figure 7: Time speedups and T-ratios for different strategies.

8.2 Evaluating SkinnerMT

We compare variants for different parallel strategies on the join order benchmark. For SP, we compare the uniform (SP-U) and adaptive (SP-A) search space partitions described in Section 5. SP-A is faster than SP-U by factor 1.17X for four threads and by factor 1.05X for 24 threads. Also, SP-A reduces T-ratio, compared to SP-U, by factor 1.59X for four threads and factor 1.08X for 24 threads. This means that redirecting more search threads to promising search space partitions reduces time and variance. For DP, we vary the split table selection policy. We compare our main variant (DP-C) against selecting the left-most table (DP-L) and selecting the largest table (DP-M). Cost-based splitting leads to optimal performance for any number of threads, achieving up to 2.3X speedup over DP-L and 1.4X speedup over DP-M. The space-efficient progress tracker presented in Section 3 reduces space consumption from 2.3 to 4.3 times, compared to the original SkinnerDB progress tracker. Finally, we compare HP-A to a simplified variant, HP-F, that fixes the number of search threads (while using the remaining threads for data processing). HP-A achieves speedups between factor 1.4X (minimal number of search threads) and 3.5X (maximal number of search threads) over HP-F and achieves comparable (worse by at most 20%) or better T-ratios. This means that adapting the distribution of threads over tasks (search versus execution) is preferable. Parallel search helps initially but becomes useless once optimal join orders are found. The extended report (Figures 12 to 15) provides details.

We perform a parameter sensitivity analysis. We vary the cost budget $\mathcal{B}$ per episode, SP's α parameter, influencing the frequency of search space reassignments, and HP's γ parameter, influencing the speed at which search threads are converted into data processing threads. Varying $\mathcal{B}$ between 50 and 5,000, and α and γ between five and 100 lead to run time variations of at most 49% and variations in T-ratio of at most 47% (considering all benchmarks, parameters, and values). Figure 21 in the technical report contains details.

Figure 7 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

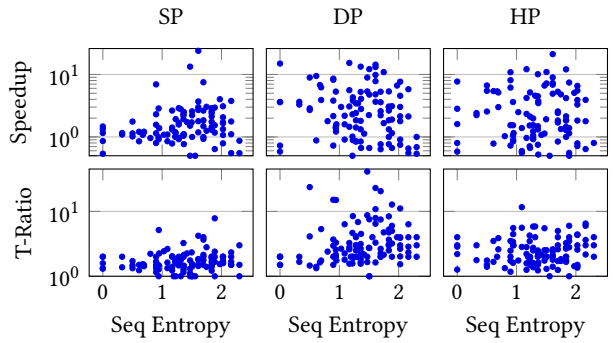


Figure 8: Time speedups and T-ratios as a function of join order selection entropy for JOB.

Table 1: Relative performance of DP and HP, compared to SP, broken down by sequential entropy.

Method	S-Low	S-High	R-Low	R-High
DP/SP	2.92	1.76	2.17	3.14
HP/SP	2.21	1.91	1.68	1.53

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.

We perform additional experiments to analyze the source of performance improvements. We hypothesize that speedups are due to a reduction in the number of episodes (rather than a reduction in the time per episode). We find a strong Pearson correlation between time and the number of episodes for all benchmarks indeed with R^2 values of at least 0.92 and p-values of at most 0.01. This is consistent with SkinnerMT's reward model, rewarding join orders with more progress per episode while keeping the cost budget per episode constant. We hypothesize that reductions in T-ratio are due to more stable join order selections (rather than a reduction in processing time variance per order). We quantify "instability" via the entropy of join order selections. I.e., for a given query, we record the last selected join order for ten runs, then calculate the entropy of the associated probability distribution. A high entropy means less stable join order choices (i.e., no convergence). Indeed, T-ratio shows a significant Pearson correlation with entropy, achieving R^2 values of 0.87 (JOB), 0.79 (TPC-H), and 0.36 (JCC-H) with p-values below 0.01 respectively. Table 2 in the technical report provides more details.

Next, we analyze how query properties influence performance gains by different parallelization methods. Our formal analysis from Section 7 suggests relative performance gains of SP over DP, the longer convergence takes without parallelization. We apply the same entropy measure as before to the queries of the join order benchmark, executed sequentially (via SkinnerDB). Here, we use entropy as a proxy for the degree of convergence. As entropy is maximal for a uniform distribution and minimal once a single join order is selected with probability one, we expect advantages of SP for high-entropy queries. Figure 8 reports results for each algorithm. Each dot represents one query from the join order benchmark, sorted by sequential entropy on the x-axis. In particular DP, the only algorithm not using parallel search, shows decreasing performance (i.e., lower speedups and higher T-ratio) as entropy increases. Table 1 reports average results for queries with high (i.e., above average) and low entropy for speedups (S) and T-ratios (R). The table reports relative results, scaling performance results of DP and HP to the corresponding number of SP. Clearly, relative to SP, the performance of DP worsens with increasing entropy, according to both metrics. HP is more robust to queries where join order search is difficult. This is expected as it uses parallel search as well.

Finally, we quantify synchronization overheads by executing a fixed number of episodes with different threads in parallel or sequentially for each of the three parallelization methods. Comparing average execution time per thread in the parallel to the sequential scenario, overheads increase from 8% for four threads to 60% for 24 threads. We consider these results reasonable for a research prototype implementation. Figure 20 in the technical report contains full details.

8.3 Comparison to Other Systems

Figure 9 compares SkinnerMT to HP, Postgres [32] (version 12.11), MonetDB [9] (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).

Figure 10 compares plans generated by different optimizer baselines in the Postgres database system. We compare plans generated by the original Postgres optimizer to join orders selected by SkinnerMT (executing queries in SkinnerMT and forcing the Postgres optimizer to execute the orders SkinnerMT converges to, setting join_collapse_limit to one) and to plans selected by an adaptive processing baseline, AQO [19], as well as BAO [25], another optimizer based on reinforcement learning. Unlike SkinnerMT, both BAO and AQO learn from prior queries. Hence, for those baselines and for each benchmark, we measure execution time after ten training runs. As the order of queries may matter for BAO's training phase, we experiment with the original query order (BAO) as well as with the inverse order (BAO-I), and randomly selected query orders (BAO-R). Figure 10 reports total benchmark run time and the number of timeouts (using a timeout of 60 seconds per query). BAO can perform better than SkinnerMT on JOB but its performance depends on the query order. It improves performance over the default optimizer on JCC-H whereas SkinnerMT performs best for JCC-H and TPC-H overall (e.g., it achieves significant improvements over the Postgres optimizer plan for TPC-H Query 2). BAO does not directly optimize join order but influences the Postgres optimizer by setting flags, e.g. enabling or disabling specific join operators. By selecting join

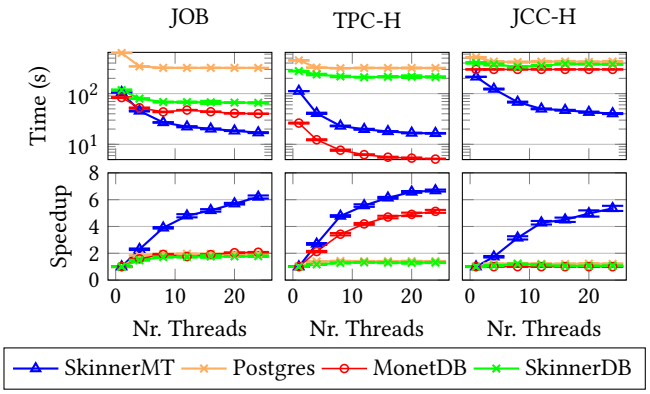


Figure 9: End-to-end performance of different systems.

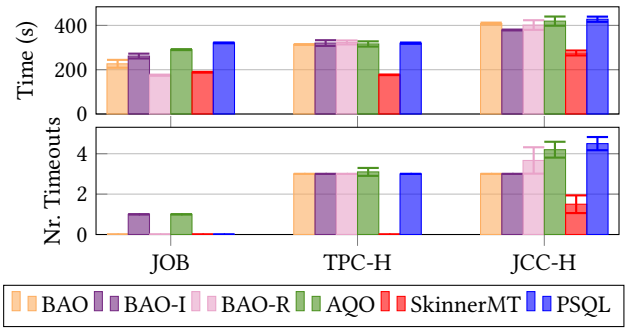


Figure 10: Performance of different query plans in Postgres.

orders directly, SkinnerMT is able to influence optimization in a more fine-grained manner. AQO achieves slight performance improvements over default plans on JOB but is otherwise dominated by the other baselines.

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 11 compares the two systems (each dot represents one query, calculating the maximal time ratio over ten runs).

9 RELATED WORK

SkinnerMT is based on SkinnerDB [39] and uses the same reinforcement learning approach to select join orders. Our focus in this paper is parallelization. SkinnerDB processes joins sequentially, severely limiting its performance as we show in the experiments. We

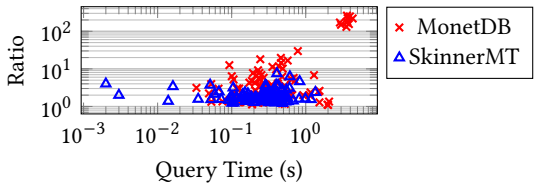


Figure 11: Performance robustness for JOB queries with regards to semantically equivalent query rewritings.

propose various algorithms to exploit parallelism for adaptive processing, covering the full spectrum from parallel join order search to parallel execution, along with formal and experimental analysis. Along with those algorithms, we solve several technical problems that arise specifically in the context of parallel processing. First, to avoid memory bottlenecks when trying many join orders in parallel, we propose a space-efficient data structure storing execution states for join orders. Second, we describe a parallel version of SkinnerDB's multi-way join algorithm to enable data-parallel processing. Third, we propose a cost-based optimization approach for adaptive data partitioning, based on run time feedback. Our work is complementary to other research on using learning for query optimization [21, 23, 26, 27, 31, 36, 45]. 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. We experimentally compare against one representative [27]. Traditional [34] and re-optimization [4, 7, 46] require initial data statistics.

Our work connects to prior work on parallel query execution [1, 12]. Unlike SkinnerMT, prior work on parallel query execution does not typically consider the possibility to execute multiple join orders concurrently. Our cost-based partitioning strategy relates to prior work on optimal data partitioning [18, 35]. However, our partitioning decisions are based on statistics (i.e., the frequency at which different tables are visited) that are specific to multi-way joins. Such statistics are not considered for data partitioning in prior work as it focuses on traditional, binary join operators.

Our work relates to prior work on adaptive query processing strategies for data streams [2, 11, 40, 42]. However, prior work in this domain has not systematically considered possibilities to trade data parallelism for search parallelism which is the focus in this paper. Doing so helps to reduce convergence time and variance (which may be less interesting for long-running queries on data streams). Prior work on parallelizing query optimization [16, 17, 38, 43, 48] 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 [13, 24] or worst-case guarantees [29, 41]).

10 CONCLUSION

SkinnerMT exploits parallelism for adaptive processing. We presented different parallelization strategies, ranging from parallel search to parallel execution. In our experiments, SkinnerMT compares favorably against various baselines.

REFERENCES

[1] 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

[2] Ron Avnur and Jm Hellerstein. 2000. Eddies: continuously adaptive query processing. In *SIGMOD*. 261–272. https://doi.org/10.1145/342009.335420

[3] 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

[4] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive re-optimization. In *SIGMOD*. 107–118. https://doi.org/10.1145/1066157.1066171

[5] K. Balakrishnan. 2019. *Exponential distribution: theory, methods and applications*.

[6] Markus Bibinger. 2013. Notes on the sum and maximum of independent exponentially distributed random variables with different scale parameters. (2013), 1–9. arXiv:1307.3945 http://arxiv.org/abs/1307.3945

[7] 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

[8] 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

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

[10] Sophie Cluet and Guido Moerkotte. 1995. On the complexity of generating optimal left-deep processing trees with cross products. In *ICDT*. 54–67. http://link.springer.com/chapter/10.1007/3-540-58907-4_6

[11] 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

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

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

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

[15] Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. 2008. PQR: Predicting query execution times for autonomous workload management. *5th International Conference on Autonomic Computing, ICAC 2008* (2008), 13–22. https://doi.org/10.1109/ICAC.2008.12

[16] 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

[17] 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

[18] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a Partitioning Advisor for Cloud Databases. In *SIGMOD*. 143–157. https://doi.org/10.1145/3318464.3389704

[19] Oleg Ivanov and Sergey Bartunov. 2017. Adaptive cardinality estimation. *arXiv preprint arXiv:1711.08330* (2017).

[20] Zohar Karnin, Tomer Koren, and Oren Somekh. 2013. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*. PMLR, 1238–1246.

[21] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: estimating correlated joins with deep learning. In *CIDR*. 1–8. arXiv:1809.00677 http://arxiv.org/abs/1809.00677

[22] 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

[23] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2020. Learning to optimize join queries with deep reinforcement learning. In *aiDM*. 1–6. arXiv:1808.03196 http://arxiv.org/abs/1808.03196

[24] 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

[25] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making learned query optimization practical. *ACM SIGMOD Record* 51, 1 (2022), 6–13.

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

[27] Tim Marcus, Ryan and Negi, Parimarjan and Mao, Hongzi and Tatbul, Nesime and Alizadeh, Mohammad and Kraska. 2022. Bao: Making Learned Query Optimization Practical. In *ACM SIGMOD Record*, Vol. 51. 5. https://doi.org/10.1145/3542700.3542702

[28] 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

[29] Hung Q Ngo and Christopher Ré. 2014. Beyond Worst-case Analysis for Joins with Minesweeper. In *PODS*. 234–245.

[30] OpenJDK. 2022. JEP 318: Epsilon: A no-op garbage collector (Experimental). https://openjdk.java.net/jeps/318 Accessed: 2022-08-30.

[31] 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

[32] PostgreSQL. 2022. Group, The PostgreSQL Global Development. https://www.postgresql.org. Accessed: 2022-08-30.

[33] 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

[34] 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

[35] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. *Vldb* 10, 4 (2016), 445–456. https://doi.org/10.14778/3025111.3025125

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

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

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

[39] 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.

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

[41] 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

[42] 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

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

[44] Ziyun Wei and Immanuel Trummer. 2022. *SkinnerMT: Parallelizing for Efficiency and Robustness in Adaptive Query Processing on Multicore Platforms*. Technical Report. https://github.com/cornelldbgroup/skinnerdb/blob/skinnermt/skinnermt.pdf

[45] Lucas Woltmann, Claudio Hartmann, Maik Thiele, and Dirk Habich. 2019. Cardinality estimation with local deep learning models. In *aiDM*. 1–8.

[46] 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

[47] Wentao Wu, Xi Wu, Hakan Hacigümüş, and Jeffrey F. Naughton. 2014. Uncertainty aware query execution time prediction. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1857–1868. https://doi.org/10.14778/2733085.2733092 arXiv:1408.6589

[48] 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