Exploiting Access Pattern Characteristics for Join Reordering

Nils L. Schubert
Technische Universitat Berlin, Germany

Philipp M. Grulich
Technische Universitat Berlin, Germany

Steffen Zeuch
Technische Universitat Berlin, Germany

Volker Markl
Technische Universitat Berlin, Germany
DFKI GmbH, Germany

ABSTRACT

With increasing main memory sizes, data processing has significantly shifted from secondary storage to main memory. However, choosing a good join order is still very important for efficient query execution in modern DBMS. This choice bases mainly on cardinality estimates for intermediate join results. However, the memory access pattern, e.g., sequential or random, on the intermediate state is an often neglected performance factor.

In this paper, we examine this impact on join query performance by evaluating the execution time, and cache misses for n-ary foreign-key joins. Based on this analysis, we propose a novel join reordering algorithm that detects the memory access pattern (using machine learning on hardware performance counters) and adapts the join order accordingly at runtime. By considering the access pattern, our evaluation shows that our adaptive reorder algorithm converges quickly to a good join order and reaches improvements of up to a factor of 5.7×.

ACM Reference Format:
Nils L. Schubert, Philipp M. Grulich, Steffen Zeuch and Volker Markl. 2023. Exploiting Access Pattern Characteristics for Join Reordering. In *19th International Workshop on Data Management on New Hardware (DaMoN '23), June 18–23, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3592980.3595304

1 INTRODUCTION

Over the last decades, memory bandwidth has become a major performance bottleneck of modern main memory database systems. To mitigate the bandwidth limitation, hardware vendors have increased the size of CPU caches and introduced hardware prefetchers to exploit the temporal and spatial locality of memory accesses [37]. Furthermore, researchers have revisited design decisions [6], cost-models [44], data-structures [20, 22, 24, 32], and operator implementations [9, 34] to improve memory and cache utilization.

Although there have been considerable improvements in designing hardware-conscious databases, finding an efficient join order is still a very important performance-critical decision [23, 25, 30]. For choosing the *best* join order, optimizers rely on cardinality and selectivity estimations. However, there are situations where the pure size of an intermediate result is not the only determining factor for performance but also *how* the data is accessed. For instance, if the individual probes of an n-ary join result in different memory access patterns (random vs. sequential), a join order that causes more sequential accesses will perform better. As a result, a join order that introduces a larger intermediate result might still be faster than a smaller one. Thus, an optimizer that relies solely on cardinality estimates might make suboptimal decisions.

We showcase one occurrence of this case in Figure 1, which represents the following SQL query.

```
SELECT * FROM A, B, C WHERE A.1 = B.1 and A.2 = C.2
```

We compare the performance of all join orders of three foreign-key joins, where each intermediate join result has the same selectivity of 22% and cardinality of 10 M tuples but different memory access patterns. We choose selectivity and cardinality to increase the impact of the different memory access patterns and omit the effect of the cache, respectively. In particular, one join probes always the same key (*repeated access*), one accesses keys sequentially (*sequential access*), and one randomly (*random access*). As shown, Permutation 1, with a repeated access pattern in the first join, is 5.5× faster than the worst order Permutation 6, with a random access pattern first. This join with very similar join cardinalities represents an obvious case where the optimizers must blindly choose a join order. However, there are other cases where the access pattern determines the final performance. For instance, when the intermediate result cardinalities are different, but one join uses a more efficient memory pattern (e.g., repeated access) and the other a highly inefficient (e.g., random pattern) [44].

This paper centers around exploiting access patterns to determine the join order. It follows the observation that each join permutation has a different data-dependent access pattern. As a result,

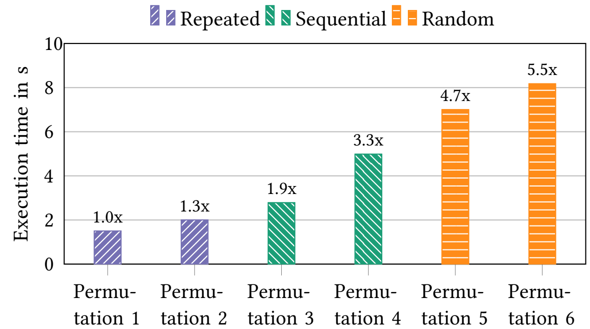


Figure 1: Execution time of all permutations of a three-way join with repeated, sequential, and random memory access patterns for the first join.

DaMoN '23, June 18–23, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0191-7/23/06…$15.00
https://doi.org/10.1145/3592980.3595304

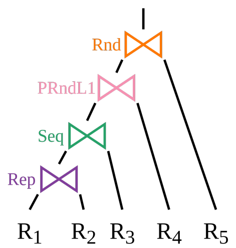


Figure 2: Query plan for the first permutation with the memory access pattern for each join.

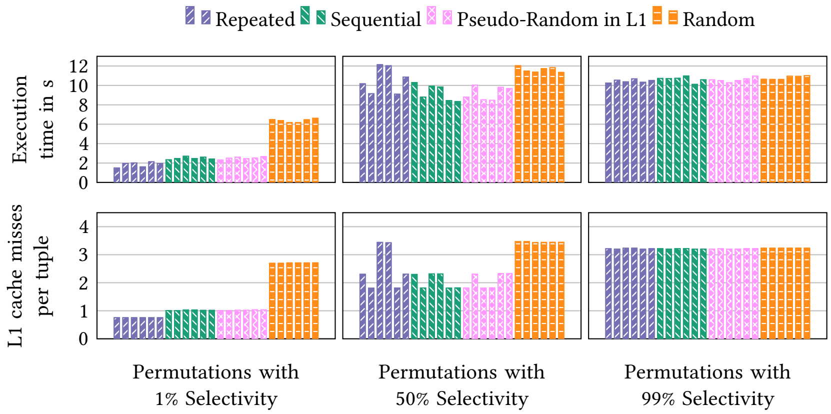


Figure 3: Execution time and L1 cache misses over all permutations for a four-way join across different selectivities.

the costs of the different join orders significantly differ, originating from their respective access patterns. To exploit these cases, we propose a novel join reordering algorithm that detects and optimizes the memory access pattern of n-ary joins. To this end, we collect hardware performance counters and introduce a machine-learning model to predict the memory access pattern at runtime. Based on this information, our algorithm [1] adapts the joins to mitigate inefficient orders. Overall, we make the following contributions:

(1) We provide a microarchitecture performance analysis of joins with similar selectivities and cardinalities (Section 3).
(2) We introduce a machine-learning model for classifying memory-access patterns based on lightweight hardware performance counters (Section 4).
(3) We propose an adaptive join algorithm that leverages memory-access patterns for choosing a good join order (Section 4).
(4) We evaluate our approach across different queries and data characteristics (Section 5).

2 BACKGROUND

In this section, we introduce hardware performance counters (see Section 2.1) and memory access patterns (see Section 2.2) as the two underlying concepts of our approach.

2.1 Hardware Performance Counters

Modern CPUs provide programmable registers, so-called performance monitoring units (PMUs). This enables us to analyze the utilization of the CPU and system resources of a program. Through hardware performance counters, we can determine the number of specific hardware events that occurred during a specific monitoring interval, e.g., executed instructions, cache accesses, or branch misses, directly in hardware. Due to their highly efficient hardware implementation, they introduce a negligible runtime overhead of less than 0.1% [44]. To use hardware performance counters, many performance analyzing tools have emerged over the last year, e.g., PERF [26] or VTUNE [18], as well as different libraries, e.g., PAPI [41] or pmu-tools [2]. We refer the reader to [17] for a detailed description of the performance counters and their implementations.

2.2 Memory Access Pattern

A memory access pattern describes the *stride* of consecutive memory accesses [16]. A stride is a difference in bytes between a sequence of data requests. Based on strides, we use the following notation throughout this paper.

- **Repeated (Rep),** stride = 0 byte: This memory pattern requests always the same data item. For example, a constant read to the same variable in a code fragment introduces this pattern.
- **Sequential (Seq),** stride = 1 byte: This memory pattern requests always the next data item. For example, a table scan or a loop over an array introduces this pattern.
- **Random (Rnd),** stride = unpredictable: This memory pattern always requests a different data item, and the stride does not follow a regular pattern that can be detected by the CPU prefetcher, e.g., lookups in a large hash table.
- **Pseudo-Random (PRndL1, PRndL2, PRndL3),** stride = size of L1, L2, or L3: This memory pattern represents a sequence of accesses that follow a regular pattern by performing *pseudo-random* memory accesses within a particular stride, e.g., L1, L2, or L3 cache size. We simulate them by pre-calculating memory accesses using the Fisher-Yates algorithm [11].

3 MICROARCHITECTURE ANALYSIS OF ACCESS PATTERN-AWARE JOINS

A good join order is crucial for the performance of most queries involving joins. Leis et al. [23] investigate join performance on a real-world data set using realistic n-ary join queries. They compare query optimizers relying on cardinality estimation to pick a good join order. In the following, we examine to which extent the memory access pattern influences an optimal join order.

Setup. We create a data set that follows the ideas of the *Join Order Benchmark*, proposed by Leis et al. [23]. We choose Query 17 as a representative four-way join query with similar cardinality. As we solely focus on the impact of the memory access pattern in this

[1] https://github.com/TU-Berlin-DIMA/apaj-hpc

analysis, we change the underlying tables so that the selectivity and, thus, the cardinality of each join result is identical for our experiment. The fact table consists of 350 M tuples of 48 byte, while each dimension tables consists of 10 M tuples of 16 byte.

Our query resembles a nested-leftmost hash-join (see Figure 2), probing a pre-built hash table sequence. We measure the L1 cache misses across four memory access patterns: *Repeated*, *Sequential*, *Random*, and *Pseudo-Random L1*. The memory access patterns for the hash probing can be provoked when generating the data, e.g., having the same value in the foreign-key column translates to a *Repeated* pattern while an autoincrement creates a *Sequential* access pattern. Furthermore, we run this experiment with three different selectivities: 1%, 50%, or 99%. We conduct our microarchitecture analysis on an *Intel Xeon Gold 5115* with 188 Gibyte RAM and show the results in Figure 3.

Results. Figure 3 shows the runtime and L1 cache misses over all 24 permutations for three different selectivities, 1%, 50%, or 99%. Note that we omit the number of L2, and L3 cache misses, as they show similar behavior. The observations are two-fold. First, the join order that induces more cache misses also has a larger execution time, which aligns with previous results [44]. Second, the memory-access pattern of the first join (indicated by the color) mainly impacts the performance of the entire set of permutations, with subsequent patterns inducing only marginal impact. In particular, for 1% selectivity, the execution time and cache misses can be clustered into four groups, each consisting of six nearly similar fast orders. The main reason for that behavior is the position of the respective access pattern in the join order. The first six join orders have the fastest pattern (repeated) at the front and the other subsequent. Because each join has a selectivity of 1% and the selectivity of the n-th join is the product of all selectivities $1..n$, each consecutive join processes fewer tuples. In our current experiment, the first join qualifies 3.5M tuples, the second 35K tuples, the third join 35, and the last is a single tuple. As a result, the first join has the highest impact on the overall execution time, and all following joins have a decreasing impact. This explains the clustering and the different execution time and cache misses in the experiment. For larger selectivities, the impact of the first join reduces and nearly vanishes for 99% selectivity. Additionally, the execution time increases for all permutations regardless of the first join memory access pattern. The cache misses similarly follow the execution times for the different permutations. The main reason is that larger selectivities result in more lookups, and thus the impact of the other joins increases for the overall execution time and cache misses. For a selectivity of 50%, we observe that the execution time varies, with some permutations even reaching the worst-case execution time. We attribute this behavior to the fact that the selectivity is high enough such that the other joins also impact the performance but not too high so that essentially all tuples are processed. Additionally, the branch predictor also plays a role, as with a selectivity of 50%, it is difficult to predict the following branch outcome. In general, the maximum speedup decreases with larger selectivities. We measure 4.4×, 1.5×, and 1.1× speedup for 1%, 50%, and 99%.

Summary. Our microarchitecture analysis confirms our assumption that joins with different access patterns lead to significantly different execution times, especially for small selectivities. Furthermore, we reveal that different join orders lead to different hardware performance counter values. Thus, we use the opportunity to learn from hardware performance counters and take action based on this in the next section.

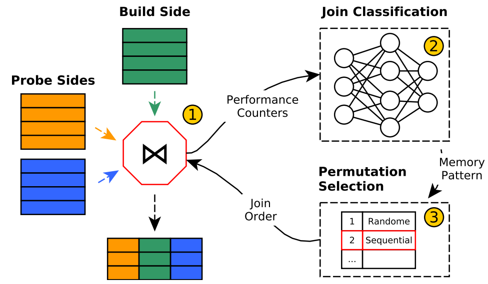


Figure 4: Access Pattern-Aware Join.

4 ACCESS PATTERN-AWARE JOINS

In the previous section, we revealed cases in which the memory access pattern impacts the query execution time. Following this insight, we present a novel access pattern-aware join (APAJ) algorithm for n-ary foreign-key joins. It consists of two steps. First, we train a machine learning model in an offline fashion. Second, our algorithm detects the memory access pattern and adapts the join order accordingly at runtime. In particular, our algorithm follows a three-step approach, as illustrated in Figure 4. Initially , our access pattern-aware join executes a random join order and gathers hardware performance counters, i.e., L1, L2, and L3 misses. Using this profiling information, it uses the machine-learning model to infer the memory-access pattern of the first join . Based on the access pattern, our algorithm adjusts the join permutation and selects a more efficient join order . To this end, we propose a greedy algorithm that stores previous predictions and chooses the join order that has a predicted *Random* access pattern as late as possible in the join order. In the remainder of this section, we discuss our memory-access pattern classification model (see section 4.1) and the reordering algorithm in detail (see Section 4.2).

4.1 Access pattern classification

As a fundamental building block of our APAJ algorithm, we predict the access pattern of the first join. We have investigated inferring multiple joins, but it was shown to be insignificant. In the following, we formulate access pattern detection as a classification problem and then introduce our machine-learning (ML) model in detail.

Access-Pattern classification. As we saw in Section 3, the access pattern of the first join impacts the execution performance the most. To this end, we investigate the impact of different memory access patterns for a 3-way foreign-key join to formulate our classification problem. Figure 5 shows the execution time, and the L1 cache misses for different access patterns across all join permutations and join selectivities, i.e., between 1% and 99% with a 1% step-size. We omit the L2, and L3 cache misses, as they show similar behavior and group the data by the memory access pattern of their first join. As shown in Figure 5, we see a correlation between the execution time and the memory access pattern. Additionally, we observe that with an increase in randomness of the memory access

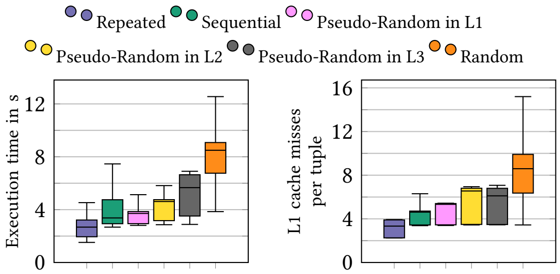


Figure 5: Execution time and L1 cache misses for all permutations of a three-way join grouped by the memory access pattern of their first join.

pattern (*Rep* →*Seq* →*PRndL1-PRndL3* →*Rnd*), the average execution time increases as well. As a result, an optimal memory access pattern can lead to up to 4× faster execution time, as it improves cache locality. Furthermore, the *Random* access pattern has the most significant performance overhead, except for some outliers. Thus, our algorithm should perform join probes with a random access pattern as late as possible. To accomplish this, we model our join classifier as a binary classification problem with the two classes `Random` and `Non-Random`. We choose to only differentiate between two cases, as additional classes require more training data and time.

ML-Model. We leverage a neural network to tackle this binary classification problem, as they perform well in classification tasks [21, 33, 36]. There are many different neural network types, and we decide to use a multilayer perception model [15] that consists of one hidden layer with four neurons and one output neuron. We found this to be an adequate model for our binary classification task. As an activation function, we use *relu* for the hidden layer to provide good generalization and no susceptibility to vanishing gradients [12]. We use the *sigmoid* function to differentiate between two classes. As an input, our model receives a vector of hardware performance counters `[L1_TCM, L2_TMC, L3_TMC]` and produces a probability for both classes.

ML-Training. We collect profiling data from 12 foreign-key queries with three memory access patterns and four selectivities to train the model. As shown in Figure 3, the impact of the first join reduces for larger selectivities. Thus, we choose small local join selectivities of 1%, 11%, 22%, and 33%, as otherwise, our model can not learn the access pattern of the first join. We have tested multiple queries with different access pattern combinations regarding the memory access patterns. We choose the three patterns `Rep/Seq/PRndL1/PRndL2`, `Rep/PRndL1/Rnd/Rnd`, and `Rep/Seq/Rnd/Rnd`, as they have shown to train our model with the same accuracy as more extensive data sets consisting of up to 3000 queries.

We have presented our machine-learning model that classifies the first join access pattern through a vector of the hardware performance counters.

4.2 Adaptive Reordering

Our access pattern-aware join adjusts the join order at runtime based on the predicted memory access pattern. To this end, we

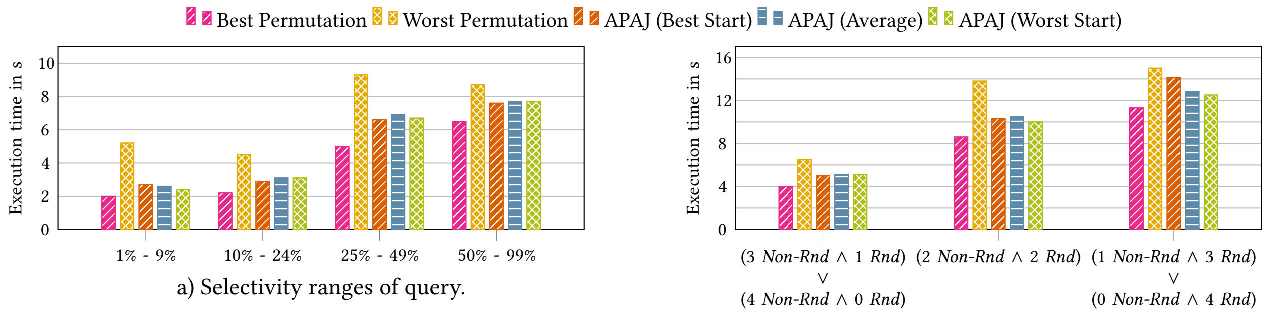
Algorithm 1: Checks if the current permutation has a better join order than another permutation

```
curPerm = bookkeeping[curPermPos];
otherPerm = bookkeeping[otherPermPos];
for join = 0 to numberOfJoins − 1 do
    curJoinPattern := curPerm[join];
    otherJoinPattern := otherPerm[join];
    if curJoinPattern = RANDOM and otherJoinPattern ≠
      RANDOM then
        return False;
    end
    if curJoinPattern ≠ RANDOM and otherJoinPattern =
      RANDOM then
        return True;
    end
end
return False;
```

propose the following adaptive reordering algorithm. In this section, we first introduce the high-level concept and describe our greedy permutation selection algorithm.

High-Level Concept. Our access pattern-aware join generally resembles a nested-leftmost hash-join, which probes a sequence of pre-built hash tables. As shown in Figure 4, APAJ processes a subset of the data from the probe relation and collects hardware performance counters, applies our classification model, and changes the join order if necessary. By swapping the permutation, we change the order in which each hash table is accessed. Initially, our approach starts with a random join permutation. Using this permutation, our approach executes the join on a record batch and collects hardware performance counters. We found a batch size of 1 k tuple to be a good compromise between the possibility of staying too long in an inefficient permutation and the accuracy of our prediction. After executing a batch, we infer the first join's memory access pattern with our machine-learning model. Our approach stays with the same permutation if the model predicts a `Non-Random` access pattern. Suppose the model predicts a `Random` access pattern. In that case, our approach executes the greedy algorithm to choose a permutation that moves the `Random` access pattern to a later join lookup.

Join Order Selection. To determine a new join order, we introduce a greedy algorithm that leverages the outcome of multiple prediction rounds to reduce the search space. We use a bookkeeping structure that stores the memory access pattern for each join order. After we process a batch of thousand tuples, we feed the collected hardware performance counters into our classification model and update our data structure with the predicted access pattern. This work assumes that once our algorithm predicts a memory access pattern for a given join, it does not change and stays consistent, and a changing access pattern is out of the scope of this work. Figure 7 illustrates an example of filling our bookkeeping structure. We predict the outermost join of a permutation to have a `Rep` access pattern. Therefore, we insert `Rnd` six times, translating to the six possible permutations.



a) Selectivity ranges of query.

b) Memory-access patterns of each query.

Figure 6: Execution time over different selectivity ranges and memory accesses.

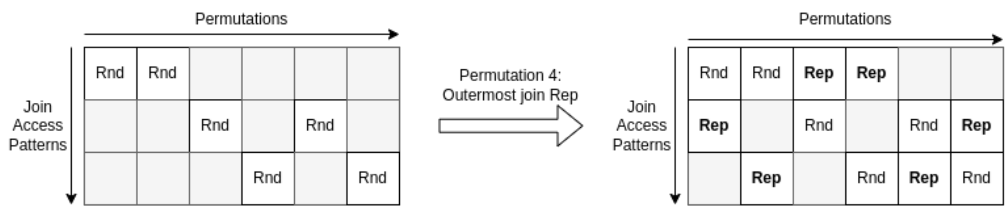


Figure 7: Two examples of filling our bookkeeping structure with predicted memory access patterns.

If we predict a `Random` memory access pattern, we search for a new join order as our current one is not optimal. We iterate through all possible join orders and compare the memory access patterns to those of our current join order. Algorithm 1 shows the respective pseudo-code. First, we retrieve the memory access patterns for both permutations from our bookkeeping structure (Lines 1-2). Afterward, we iterate through the access patterns and compare the two patterns to check which permutation has a more favorable join order (Lines 3-11). We define a more favorable join order where `Random` access pattern lookups occur as late as possible (Line 6-10).

As shown in Figure 7, our algorithm updates all possible positions that the current outermost join can appear with the predicted memory access pattern. Additionally, we assume that our prediction does not change in the following rounds. Therefore, we have inferred the first join's memory access pattern for all different join orders after predicting the number of probes in our join. Therefore, we can choose the best permutation based on our predictions for the rest of the tuples. Thus, our algorithm has a runtime of $O(n)$, with n being the number of probes, for example, four predictions for a five-ary join.

Summary. This section presents the adaptive reordering algorithm for our access pattern aware join. After training a machine learning classifier offline, we detect the memory access pattern and reorder the join lookups adaptively. It leverages the outcome of previous prediction rounds to reduce the search space for finding a new join order once we predict a non-optimal access pattern at the first join.

5 EVALUATION

In this section, we experimentally evaluate our access pattern-aware join. Section 5.1 describes our experimental setup, followed by conducting end-to-end experiments and evaluating our approach. Then, we conduct end-to-end experiments and evaluate our approach. First, we perform an end-to-end experiment (see Section 5.2) to evaluate the performance impact of our approach on the execution time. Then, we study the impact of different start permutations (see Section 5.3) to assess the robustness of our approach.

5.1 Setup

In the following section, we present our experimental setup (see Section 5.1.1), our workloads (see Section 5.1.2), and the setup of our classification model (see Section 5.1.3).

5.1.1 Hardware and Software. We perform all experiments on an Intel Xeon Gold 5115 (Skylake) processor with ten physical and 20 logical cores. The server has a total of 188 Gibyte main memory. Our implementation is written in C++17 and compiled with GCC 11.2.0 with O3, march, and mtune compiler flags to produce optimized code for the underlying hardware. We use PAPI [39] version 6.0.0 to read hardware counters and use tensorflow [1] version 2.1.0 in combination with python version 3.10.

5.1.2 Workloads. Throughout the evaluation, we use 100 randomly generated four-way join queries. Each query is uniformly drawn from all possible combinations of the discussed memory access patterns in Section 2 and join selectivities between 1% to 99%. If not stated otherwise, all queries access a fact table of 350 M records and dimension tables of 10 M records, which results in a working set of 17 Gbyte. Additionally, we use batches of 1 k tuples for prediction if not otherwise stated. Following our findings from Section 3, we define an optimal join order as a permutation with no random memory access in the first join (as the first one has the highest performance impact). All queries consist of a join that resembles a nested-leftmost hash join, which probes pre-built hash tables.

5.1.3 Neural Network Setup. For training, we use 12 selected queries that are shown in Section 4, which results in more `Non-Random` data points than `Random`. This results in an unbalanced data set, and thus a risk of failing to generalize exists as the model could learn to stick to predicting one class. Therefore, we duplicate the `Random` data points until there is an equal number of `Non-Random` and `Random` labels. With this, we end up with 228 data points for each classification class. We perform an extensive parameter evaluation and report below the best. We choose a 72%, 8%, 20% split for training, validation and test set. We use Adam [19] as an optimizer with default values in combination with a learning rate of 10^{-4} and *AMSGrad*, as it works well in practice [19]. Additionally, we implement an *EarlyStopping callback* on the validation loss with a patience of 100 and a minimum delta of 0.001 to reduce the risk of overfitting. This enables us to train our model in under a minute and run inference in under 60 ms. With an overall execution time of at least a second, the overhead of running inference is negligible in our experiments.

5.2 Performance of Access Pattern-Aware Join

In this experiment, we study the performance impact of our approach. As baselines, we report the execution time of the best and worst join permutations. The execution time of APAJ is the average among all 24 possible start permutations. The execution time of APAJ (Best Start) and APAJ (Worst Start) is the execution time if we start in the best or worst permutation but run our adaptive join reordering scheme. We group the queries according to their selectivities and memory access patterns.

Results. Figure 6 shows the best and worst cases among all permutations and our approach for each query. We have further divided the adaptive approach into an average across all 24 start permutations, and the execution time started in the best and worst permutations. Across all experiments, we measure an average speedup of $1.1-2.2\times$ compared to the mean execution time for each query.

In Figure 6a, the experiment reveals that the more selective a query is, the more the difference between the best and worst case reduces with an overall increase in execution time. The largest speedup of 2.2× originates for selectivities less than 10%, while our approach only results in a speedup of 1.2× for selectivities larger than 50%. In general, the more selective a query is, the more tuples must be processed, resulting in more work being performed in total. Furthermore, APAJ shows similar execution times, regardless if we start in the best or worst permutation. Thus, we show that our approach improves the performance compared to the worst case regardless of the selectivity, with smaller selectivities leading to a higher speedup.

In Figure 6b, we see that more random access patterns result in a higher execution time for different memory access patterns. This aligns with our investigation in Section 3. APAJ shows a speedup of $1.1-1.9\times$ compared to the worst case for all memory access patterns. For queries with the same number of `Non-Random` and `Random` access patterns, the execution time of the worst case is close to the execution time of queries with more `Random` than `Non-Random` access patterns. We do not see this behavior for our approach, meaning it can still find a viable join order and avoids the worst permutation.

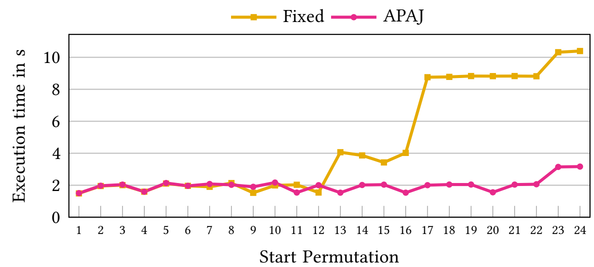


Figure 8: Execution time over different starting permutations with a fixed permutation or APAJ.

Summary. APAJ improves the performance regardless of the distribution of the memory access pattern and the selectivities. We measure a maximum speedup of 2.2× for selectivities below 10% and an average speedup of $1.1-2.2\times$ across all experiments. Thus, APAJ shows robustness against different selectivities and distribution of memory access patterns for n-ary joins.

5.3 Different starting permutations

In this experiment, we investigate the behavior of our access pattern-aware joins on different starting permutations to highlight the possibilities of APAJ. We create a four-way join and construct the fact table such that a memory access pattern of *Repeated*, *Repeated*, *Random*, and *PRand2* exists. Additionally, we change all tables to get a selectivity of 49% for all four joins lookups.

Results. Figure 8 shows the execution time over different starting permutations for a fixed permutation and with APAJ. We observe that different starting permutations result in different execution times, which aligns with our findings of Section 3. Additionally, we measure a maximum speedup of 5.7× compared to no join reordering. This shows that a query remains in its starting permutation, which can cause a significant execution time overhead. Figure 8 shows that with join reordering enabled, APAJ always sticks to the overall best permutation. Additionally, our access pattern-aware join only changes to another permutation if it detects a *Random* memory access pattern on its first join. For this example query, our approach improves performance for the last 12 permutations with an additional sharp increase in performance for the last eight permutations. Furthermore, for cases where a join reordering would not be ideal, i.e., first join type *Repeated* or *PRndL1*, the execution time is almost the same with and without join reordering. This highlights the neglectable performance overhead of access pattern-aware joins but, at the same time, shows the significant performance improvement potential.

Summary. The experiments show that our approach is robust against different starting permutations. It finds a viable join order and avoids sticking to the worst permutation.

5.4 Discussion

In our experiments, we evaluated our access pattern-aware joins for an end-to-end benchmark with a different number of joins. We show that our approach does not incur any measurable overhead, shows robustness for different queries, and corrects the worst permutation quickly. Furthermore, we measure a maximal speedup of 5.7×, which shows cases where our approach offers significant performance improvements. In a final experiment, we highlight that the access pattern-aware joins does not depend on the starting permutation. In summary, exploiting the memory access pattern for determining the best join order is highly beneficial and should be considered by today's query optimizers.

6 RELATED WORK

We structure the related work into self-driving databases and adaptive join optimization.

Self-Driving Databases. Self-driving or self-tuning databases are database management systems that usually collect statistics. Upon the collection, it can perform self-optimization on multiple aspects, such as query optimization or layout orientation[4, 7, 8].

Pavlo et al. [31] reveal that databases are still tuned with a human in the loop, which results in a reactionary fixing of problems after they occur. Thus, they propose a fully self-driving database system named Peleton that utilizes deep learning. Our work focuses on finding a viable join order, whereas they propose a system.

Further approaches exist towards self-tuning query optimizers [38, 42, 47]. Stillger et al. [38] describe a feedback loop that monitors previously executed queries and then compares the estimated cardinality with the actual. They improve query optimization by providing a better cardinality estimation. Woltmann et al. [42] accelerate the training phase for machine learning-based approaches by executing example queries over pre-aggregated data. They all have in common that they provide an improved model for cardinality estimation. Dutt et al. [10] propose an approach that performs query processing without estimating the selectivities. Their approach chooses a subset (bouquets) of the optimal plans and then discovers the actual selectivities during the partial execution of the bouquet plans. In contrast, access pattern-aware joins does not utilize cardinalities but instead predicts the memory access pattern using performance counters.

Zeuch et al. [44] used hardware performance counters to reorder multi-selection queries, whereas we focus on utilizing hardware performance counters to reorder joins. Similarly, Grulich et al. [14] leverage hardware performance counters to optimize stream processing queries at runtime.

Trummer et al. [40] propose using reinforcement learning to choose a good join order. They divide the execution of a query into small time slices and try out different join orders for each slice. In contrast, we do not employ reinforcement learning or try different join orders. However, instead, we switch to another join order upon adaptively detecting a non-optimal memory access pattern.

Adaptive Join Optimization. Adaptive join optimization is a well-studied field in database research [13, 30]. Avnur et al. [3] propose continuously reordering operators during query processing. They introduce eddies, which route tuples to each operator, and once all operators have processed the tuple, the eddy emits it to the output. In contrast, we reorder based on hardware performance counters while they use the throughput of different operators. Markl et al. [27] propose POP, a progressive query optimization. POP monitors the actual cardinalities and compares them to their estimated values, and upon detecting a significant difference, a re-optimization is performed. In contrast to POP, we do not need to recompile the whole query during re-optimization. Li et al. [25] rearranges joins according to their rank, calculated on the join cardinality costs. Răducanu et al. [35] contribute a new $\epsilon-$greedy learning algorithm that chooses an alternative function implementation. Common to our approach, they execute batches of tuples and, if needed, change to a different flavor of the query. Michalke et al. [29] tackle the challenge of energy-efficient joining of data streams. They introduce an adaptive join algorithm that adjusts the batch size of the data on incoming data stream rates and a user-provided constraint. However, all approaches are not using hardware performance counters in their estimates, which have a negligible overhead as discussed in Section 2.1.

Menon et al. [28] propose a bridge between just-in-time compilation and adaptive query processing. They transform a query execution plan into a permutable compiled query such that a different permutation can be executed later without recompiling. In contrast, our approach changes to a new permutation based on hardware performance counters, while Menon et al. utilize cardinalities. Zhu et al. [46] propose Lookahead Information Passing (LIP) that uses a lookahead filter, e.g., Bloom Filter[5], to decide for a better join order. They show that LIP ensures that execution times for the best and worst-case planes are far closer than without LIP. In contrast, APAJ does not require additional complex data structures to store information about each join.

7 CONCLUSION

In this paper, we have introduced the first access pattern-aware join (APAJ). We show that joins with different memory access patterns can lead to significantly different execution times, especially for smaller selectivities. Additionally, we reveal that different join orders lead to different hardware performance counter values, allowing us to learn from them and take action. Our APAJ algorithm uses a multilayer perception model to predict the memory access pattern of the first join and, if applicable, chooses a better join permutation. Through an experimental evaluation, we have shown that APAJ does not incur a measurable overhead and is robust versus different starting permutations. Furthermore, we have shown instances when APAJ offers significant performance gains of up to 5.7×. Thus, today's query optimizers should consider exploiting the memory access pattern. In the future, we plan on integrating our findings into the state-of-the-art stream processing system NebulaStream [43, 45] and conduct further research in this area.

ACKNOWLEDGMENTS

This work was funded by the European Union's H2020 research and innovation program - ELEGANT (No. 957286), the German Federal Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and 01IS18037A), DFG Priority Program (MA4662-5) and moreEVS (410830482), and the Software Campus (01IS17052).

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard,

Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Andi Kleen. [n. d.]. PMU Tools. Accessed Mar 9, 2023. https://github.com/andikleen/pmu-tools.

[3] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. *SIGMOD Rec.* 29, 2 (may 2000), 261–272. https://doi.org/10.1145/335191.335420

[4] Philip A. Bernstein, Michael L. Brodie, Stefano Ceri, David J. DeWitt, Michael J. Franklin, Hector Garcia-Molina, Jim Gray, Gerald Held, Joseph M. Hellerstein, H. V. Jagadish, Michael E. Lesk, David Maier, Jeffrey F. Naughton, Hamid Pirahesh, Michael Stonebraker, and Jeffrey D. Ullman. 1998. The Asilomar report on database research. *ArXiv* cs.DB/9811013 (1998).

[5] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (jul 1970), 422–426. https://doi.org/10.1145/362686.362692

[6] Peter Boncz, M. Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. *2nd Biennial Conference on Innovative Data Systems Research, CIDR 2005.*

[7] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 3–14.

[8] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with ITuned. *Proc. VLDB Endow.* 2, 1 (aug 2009), 1246–1257. https://doi.org/10.14778/1687627.1687767

[9] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the Impact of Memory Allocation on High-Performance Query Processing. In *Proceedings of the 15th International Workshop on Data Management on New Hardware.* ACM. https://doi.org/10.1145/3329785.3329918

[10] Anshuman Dutt and Jayant R. Haritsa. 2014. Plan Bouquets: Query Processing without Selectivity Estimation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1039–1050. https://doi.org/10.1145/2588555.2588566

[11] Ronald Aylmer Fisher and Frank Yates. 1948. *Statistical tables for biological, agricultural and medical research* (3rd ed., rev. and enl ed.). Oliver and Boyd, London.

[12] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep Sparse Rectifier Neural Networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 15)*, Geoffrey Gordon, David Dunson, and Miroslav Dudík (Eds.). PMLR, Fort Lauderdale, FL, USA, 315–323. https://proceedings.mlr.press/v15/glorot11a.html

[13] Anastasios Gounaris, Norman W. Paton, Alvaro A. A. Fernandes, and Rizos Sakellariou. 2002. Adaptive Query Processing: A Survey. In *Advances in Databases*, Barry Eaglestone, Siobhán North, and Alexandra Poulovassilis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 11–25.

[14] Philipp M Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient stream processing through adaptive query compilation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 2487–2503.

[15] Trevor Hastie, Robert Tibshirani, Jerome Friedman, and James Franklin. 2004. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Vol. 27. 83–85 pages. https://doi.org/10.1007/BF02985802

[16] J.K. Hughes. 1986. *PL / I Structured Programming.* Wiley. https://books.google.de/books?id=WCEiAQAAIAAJ

[17] Intel. 2021. *Intel® 64 and IA-32 ArchitecturesOptimization Reference Manual.* Intel Corp.

[18] Intel Corporation. [n. d.]. Intel® VTune™ Profiler. Accessed Mar 9, 2023. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html.

[19] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[20] Thomas Kowalski, Fotios Kounelis, and Holger Pirk. 2020. High-Performance Tree Indices: Locality matters more than one would think. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2020, Tokyo, Japan, August 31, 2020*, Rajesh Bordawekar and Tirthankar Lahiri (Eds.). 1–7. http://www.adms-conf.org/2020-camera-ready/ADMS20_03.pdf

[21] Hemalatha Kulala and K. Rani. 2017. Advancements in Multi-Layer Perceptron Training to Improve Classification Accuracy. *International Journal on Recent and Innovation Trends in Computing and Communication* 5 (06 2017), 353–357.

[22] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data.* 311–326.

[23] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9 (2015), 204–215.

[24] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE).* IEEE, 38–49.

[25] Quanzhong Li, Minglong Shao, Volker Markl, Kevin Beyer, Latha Colby, and Guy Lohman. 2007. Adaptively Reordering Joins during Query Execution. In *2007 IEEE 23rd International Conference on Data Engineering.* IEEE. https://doi.org/10.1109/icde.2007.367848

[26] Linux Kernel Organization, Inc. [n. d.]. Perf Wiki. Accessed Mar 9, 2023. https://perf.wiki.kernel.org/.

[27] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust Query Processing through Progressive Optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 659–670. https://doi.org/10.1145/1007568.1007642

[28] Prashanth Menon, Amadou Ngom, Lin Ma, Todd C. Mowry, and Andrew Pavlo. 2020. Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling. *Proc. VLDB Endow.* 14, 2 (Oct. 2020), 101–113. https://doi.org/10.14778/3425879.3425882

[29] Adrian Michalke, Philipp M. Grulich, Clemens Lutz, Steffen Zeuch, and Volker Markl. 2021. An Energy-Efficient Stream Join for the Internet of Things. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)* (Virtual Event, China) *(DAMON'21)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. https://doi.org/10.1145/3465998.3466005

[30] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 677–692. https://doi.org/10.1145/3183713.3183733

[31] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR 2017, Conference on Innovative Data Systems Research.* https://db.cs.cmu.edu/papers/2017/p42-pavlo-cidr17.pdf

[32] Shangfu Peng, Yin Yang, Zhenjie Zhang, Marianne Winslett, and Yong Yu. 2012. DP-tree. In *Proceedings of the 2012 international conference on Management of Data - SIGMOD '12.* ACM Press. https://doi.org/10.1145/2213836.2213972

[33] Weibiao Qiao, Mohammad Khishe, and Sajjad Ravakhah. 2021. Underwater targets classification using local wavelet acoustic pattern and Multi-Layer Perceptron neural network optimized by modified Whale Optimization Algorithm. *Ocean Engineering* 219 (2021), 108415.

[34] Suprio Ray, Catherine Higgins, Vaishnavi Anupindi, and Saransh Gautam. 2020. Enabling NUMA-aware Main Memory Spatial Join Processing: An Experimental Study. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2020, Tokyo, Japan, August 31, 2020*, Rajesh Bordawekar and Tirthankar Lahiri (Eds.). 1–8. http://www.adms-conf.org/2020-camera-ready/ADMS20_04.pdf

[35] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. 2013. Micro Adaptivity in Vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1231–1242. https://doi.org/10.1145/2463676.2465292

[36] Jagsir Singh and Jaswinder Singh. 2021. Malware Classification Using Multilayer Perceptron Model. In *International Conference on Innovative Computing and Communications*, Deepak Gupta, Ashish Khanna, Siddhartha Bhattacharyya, Aboul Ella Hassanien, Sameer Anand, and Ajay Jaiswal (Eds.). Springer Singapore, Singapore, 155–168.

[37] Alan Jay Smith. 1982. Cache Memories. *Comput. Surveys* 14, 3 (sep 1982), 473–530. https://doi.org/10.1145/356887.356892

[38] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 19–28.

[39] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009.* Springer Berlin Heidelberg, 157–173. https://doi.org/10.1007/978-3-642-11261-4_11

[40] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB. In *Proceedings of the 2019 International Conference on Management of Data.* ACM. https://doi.org/10.1145/3299869.3300088

[41] University of Tennessee. [n. d.]. Performance Application Programming Interface. Accessed Mar 9, 2023. https://icl.utk.edu/papi/.

[42] Lucas Woltmann, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2022. Aggregate-based Training Phase for ML-based Cardinality Estimation. *Datenbank-Spektrum* 22, 1 (2022), 45–57. https://doi.org/10.1007/s13222-021-00400-z

[43] Steffen Zeuch, Ankit Chaudhary, Bonaventura Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform: Data and Application Management for the Internet of Things. In *Conference on Innovative Data Systems Research (CIDR)*.

[44] Steffen Zeuch, Holger Pirk, and Johann-Christoph Freytag. 2016. Non-invasive progressive optimization for in-memory databases. *Proceedings of the VLDB Endowment* 9, 14 (oct 2016), 1659–1670. https://doi.org/10.14778/3007328.3007332

[45] Steffen Zeuch, Eleni Tzirita Zacharatou, Shuhao Zhang, Xenofon Chatziliadis, Ankit Chaudhary, Bonaventura Del Monte, Dimitrios Giouroukis, Philipp M Grulich, Ariane Ziehn, and Volker Mark. 2020. NebulaStream: Complex analytics beyond the cloud. *The International Workshop on Very Large Internet of Things (VLIoT 2020)* (2020).

[46] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust: Making the Initial Case with in-Memory Star Schema Data Warehouse Workloads. *Proc. VLDB Endow.* 10, 8 (apr 2017), 889–900. https://doi.org/10.14778/3090163.3090167

[47] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. arXiv:2011.09022 [cs.DB]