Exploring Learned Join Algorithm Selection in Relational Database Management Systems

by

Long Phi Nguyen

S.B., Computer Science and Engineering
Massachusetts Institute of Technology (2019)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2021

Author .....
Department of Electrical Engineering and Computer Science
December 15, 2020

Certified by .....
Ryan C. Marcus
Postdoctoral Associate
Thesis Supervisor

Certified by .....
Tim Kraska
Associate Professor of Electrical Engineering & Computer Science
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Exploring Learned Join Algorithm Selection in Relational Database Management Systems

by

Long Phi Nguyen

Submitted to the Department of Electrical Engineering and Computer Science
on December 15, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Query optimizers, crucial components of relational database management systems, are responsible for generating efficient query execution plans. Despite many advances in the database community over the last few decades, most popular relational database management systems today still use cost-based optimizers that do not always model the underlying data's characteristics accurately. These cost-based optimizers brutally slow down a query if they make even one gross underestimate of a database table's cardinality. In this work, we improve on native cost-based optimizer performance by identifying the most ideal join algorithms for query execution plans in two popular relational database management systems, PostgreSQL and Microsoft SQL. First, we gather baseline query execution times for the entire IMDb Join Order Benchmark under different subsets of usable join algorithms to show that no subset yields high performance across all queries. We then show that it is feasible to use deep reinforcement learning to choose one of these subsets for each query seen and achieve far better performance on the intensive JOB queries. Finally, we introduce the idea of k-edits, showing results that indicate that for some queries, isolating just 1 "bad" join and changing its join algorithm can yield better performance. Our work suggests that reinforcement learning with both coarse and fine decisions shows huge potential for the future of query optimization and relational database management systems.

Thesis Supervisor: Ryan C. Marcus
Title: Postdoctoral Associate

Thesis Supervisor: Tim Kraska
Title: Associate Professor of Electrical Engineering & Computer Science

Acknowledgments

Thank you, Ryan Marcus, for being a wonderful mentor and guiding me through the process of research. You've really shown me more about academia, and I feel that I am walking out of MIT stronger than ever and ready to tackle new challenges. Thank you for the technical guidance and the reassurance to help me realize that even in research, it's okay if not every approach yields something significant, so long as we remain steadfast and keep at it.

Thank you, Professor Tim Kraska, for your willingness to take in this novice researcher. I am humbled to have received this opportunity and have learned so much along the way.

Thank you, Montse, for staying by my side and getting me excited about the things I do. You have made me a better person, and I thank you for your never-ending love and support.

Và cuối cùng, con cảm ơn ba mẹ đã thương yêu con và nâng niu con trong những năm vừa qua. Sự thành công của con cũng là của ba mẹ. Con sẽ không bao giờ quên tình thương của ba mẹ.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Relational databases were inspired by the need for large, organized services capable of delivering data to users without exposing said data's internal representation. Codd first proposed this idea, formalizing data in terms of sets and their manipulation based on mathematical operations [2]. Functionally, relational databases divide their data into *relations*, tables of attributes grouped into rows. Relations can be compared and composed with various *relational algebra* operations, which form the basis for high-level query languages, such as SQL, that are used to deliver the underlying data to users.

A great amount of research over the last few decades has led to modern relational database management systems (RDBMSs) like Microsoft SQL Server, PostgreSQL, Amazon Relational Database Service, and more, giving computer science researchers and software developers great flexibility in constructing solutions tailored to their work. Each RDBMS implements its own *query optimizer* (see Figure 1-1 for an example), which attempts to determine and execute the most efficient strategy for each user-given query. In particular, the optimizer selects, for each join between two relations in a query, an *operator* that specifies through a *join algorithm* how the relations' data should be processed together. Note that each RDBMS offers its own set of join algorithms. For example, SQLite only offers the nested loop join, Postgres adds hashing and merge-sort on top of this, and MySQL/MariaDB go so far as to introduce batched key access (BKA) algorithms for efficient scanning of data.

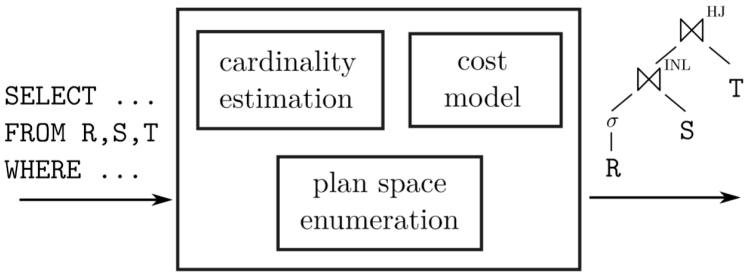

Figure 1-1: Traditional query optimizer architecture; taken from [6]. Given a query, the optimizer chooses the query plan it believes is best based on cardinality estimates for the relations involved in the query and the cost model's response to these estimates across possible plans that are semantically equivalent.

Why might it be useful to look at join algorithms in query execution plans? To answer this, we need to think about the following questions:

- How might cost-based query optimizers do poorly?
- Even with a good query plan, how much of a performance hit can we expect from choosing the wrong join algorithm to join two relations?

Consider that most cost-based query optimizers compute cardinality estimates for relations through heuristics based on simple assumptions such as data uniformity and independence [6]. They then choose the best query plan *according to these estimates*. However, choosing query plans this way can be devastating in practice if the data does not follow the optimizer's assumptions—a table T of size 10^6 might be estimated to have only 10^3 rows, thus impacting the actions the optimizer decides to take. The optimizer may choose a nested loop algorithm to join T and some U if it believes that T only has 10^3 rows, regardless of the overall query plan, due to nested loops being efficient for small amounts of data. This would result in a significant slowdown versus a plan that uses hashing to join T and U.

If we can do better than cost-based models that are fed wrong cardinality estimates by selecting join algorithms that are more reflective of the data and queries given, we may not have to compromise on performance. This thesis explores this problem of join algorithm selection and serves as a collection of experiments upon which to base

further work. We experiment with multiple techniques for choosing join algorithms using queries from the *Join Order Benchmark* [6] and identify several key points:

- If we isolate the join algorithms used across the entire DBMS to favor those that improve performance on more intensive queries, other queries will suffer as a result. There is no configuration that is ideal for every query.

- For queries involving gross optimizer underestimates (such as the example with T above), it is possible to achieve consistent performance gains through deep Bayesian reinforcement learning.

- For some intensive queries, suggesting a change of algorithm for just one join may yield significantly better performance.

In Chapter 2, we cover in more detail the building blocks for our methods and relevant work. In Chapter 3, we isolate different subsets of join algorithms used by the Postgres query optimizer and run, under each subset, all JOB queries to provide a baseline and further motivate the problem. In Chapter 4, we improve on the Postgres optimizer's performance by modeling join algorithm selection as a reinforcement learning problem (specifically the *contextual multi-armed bandit* problem), using a deep Bayesian neural network to choose a subset of join algorithms for each JOB query encountered. In Chapter 5, we move away from this broad approach of choosing subsets of join algorithms and begin to investigate the effects of modifying up to k join algorithms in a given query. In Chapter 6, we compare our results from Chapters 3 to 5. We conclude in Chapter 7 with a discussion on how the experiments shown in this work may lead to exciting progress in query optimization.

Chapter 2

Background and Related Work

In this chapter, we first provide more detail that serves as a reference for our methods in Chapters 3 to 5—we cover the `JOIN` operator, reinforcement learning, and the Join Order Benchmark. We then discuss the state-of-the-art in query optimization and continue to motivate the problem of efficient join algorithm selection in query execution plans. Note that we use relations and tables interchangeably.

2.1 The `JOIN` operator

`JOIN` is a binary operator that returns a combination of rows from two relations based on common attributes. Formally, given two relations $R_1 = \{A, B, C, \ldots\}, R_2 = \{X, Y, Z, \ldots\}$ and join conditions $C = \{R_1.A\,\theta\,R_2.X, ...\}$, where θ is an arbitrary relationship between two attributes, we generate R_3 with rows containing content from both R_1 and R_2 that satisfy C. The most common join type, **and the only one we are concerned with in this thesis**, is the inner join where, for each $c_i \in C$, θ represents equality ($=$). Conditions such as $<, \leq, >$, etc. are beyond the scope of our work. Some other types of joins are shown in Figure 2-1.

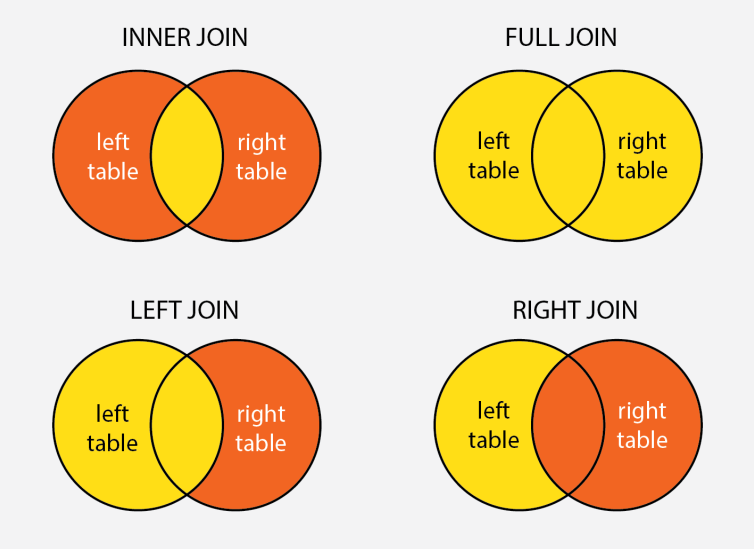

Figure 2-1: Four join types are shown here. The inner join is most commonly used, and support for other types of joins depends on the DBMS used.

2.1.1 Basic join algorithms

`JOIN` requires an algorithm for scanning and aggregating relations. Many such algorithms exist, allowing workloads to scale from single-server contexts to large, distributed networks on thousands of cores. Here, we describe the three basic join algorithms available in PostgreSQL and Microsoft SQL Server, our RBDMSs of interest in this work. In later chapters, we analyze performance in scenarios that involve only these three algorithms.

Nested loop join

The nested loop join is the most basic join algorithm, iterating through all rows of R_2 for each row in R_1—the Cartesian product—to find matches on join attributes. It is the most simple join algorithm to implement and runs in $O(|R_1| \times |R_2|)$ time.

Hash join

Let R_1 be the smaller of the two relations by cardinality. The hash join treats R_1 as the *build input* and R_2 as the *probe input*. Using memory, the algorithm builds a hash table over the build input and then scans the probe input for matches with the built

hash table. This algorithm runs in $O(|R_1|+|R_2|)$ time. If there is not enough memory in which to fit the entire build input, the algorithm proceeds in phases, working on portions at a time; this is known as a *grace hash join*.

Sort-merge join

The sort-merge join relies on both R_1 and R_2 being sorted, using two pointers to compare the join attribute desired. Both tables are first sorted (assuming they are unsorted) based on the join attribute. Both are then scanned in parallel linearly, with matching rows on the join attribute combined for the final output table.

There are three cases to consider when thinking about the runtime:

1. If both R_1 and R_2 are already sorted, sort-merge runs in $O(|R_1| + |R_2|)$ time, as there is no need to actually sort the tables.

2. If R_1 is sorted and R_2 is not, we only need to sort R_1, giving us $O(|R_1| \log |R_1| + |R_2|)$.

3. If both R_1 and R_2 are unsorted, we sort both, giving us $O(|R_1| \log |R_1| + |R_2| \log |R_2|)$.

Case 3 is, in practice, very slow, and as such, sort-merge join works ideally when one or both tables are already sorted.

2.2 Machine learning

Many human problems involve making decisions or predictions. *Machine learning* takes these problems to algorithmic models, using example data and past experience to optimize some criteria [1]. These models refine their knowledge over time, with their decision and prediction accuracy dependent on the type of data they are exposed to and the quality of their experience.

A basic example is object recognition—given an image of a cat and two possible classifications, dog or cat, can a machine learning model correctly identify the image as that of a cat?

Advanced examples include predicting stock market prices based on decades of financial data and defeating an opponent in chess.

2.2.1 Reinforcement learning

Reinforcement learning is a subset of machine learning that focuses on allowing some agent to make informed decisions in a complex environment. The environment is represented at time t by some state s_t conveyed to the agent. Based on what it knows about the environment and its own policy P for interpreting the environment, the agent takes an action $a_i \in A = \{a_0, a_1, a_2, \ldots, a_n\}$. After taking a_i, the agent is given reward $Q(a_i) = r_t$. This process continues with s_{t+1} and so on until there are no more actions left to take. One cycle of decision-making is called an *episode*, and the agent must learn to maximize its reward across episodes.

2.2.2 The exploration-exploitation dilemma & multi-armed bandits

When training a reinforcement learning model, there is inherently a conflict between *exploration* and *exploitation*. When attempting to maximize rewards across episodes, pure exploitation based on the best-known action drives the agent towards a locally optimal solution that may fail to account for other actions that could yield a better payoff in the long term despite poor performance in the short term. On the other hand, pure exploration, although beneficial for agent flexibility in a highly dynamic environment, reduces learning potential and thus reward potential for the agent.

Multi-armed bandits are one type of problem in which we encounter this exploration-exploitation dilemma. The agent chooses a_t, one of k arms, or actions, at each round. The environment samples reward r_t for the agent from a probability distribution unknown to the agent after the arm is pulled. Over time, the agent must balance playing the arm that historically gives large rewards and exploring other arms that may lead to bigger future payoffs.

2.3 The Join Order Benchmark

There are many moving parts in any RDBMS, but in this work, we focus only on the query optimizer. As mentioned in Chapter 1, the query optimizer attempts to generate the most efficient execution plan for any given query. However, what the optimizer believes to be the best plan may very well be a poor plan, sometimes orders of magnitudes worse than the actual best plan.

Previous research has generated many benchmarks for challenging query optimizers. In particular, the *Join Order Benchmark* [6] is a set of 113 SQL queries that select data across 21 tables from the Internet Movie Database, IMDb. These queries are structured such that they might be asked by movie enthusiasts. For example, Query 13d returns ratings and release dates for all movies produced in the US.

The Join Order Benchmark shows that even industrial-strength query optimizers built with decades of developer experience often produce large errors for cardinality estimation [6]. These errors in cardinality estimation heavily influence many parts of the final query plan chosen by the optimizer, such as the join order of the relations in the query, the join algorithms used between relations, and the types of operators used to scan database indices.

What differentiates the JOB from other benchmarks was the authors' purpose in writing it: to explicitly model and show the difficulty of *join ordering*, a problem related to join algorithm selection. These queries are challenging to execute efficiently due to how many joins are involved in each and actual correlations in the data that classical query optimizers do not account for in their cost models. Thus, these queries are useful for our work, as we want to investigate join algorithm selection on queries that expose optimizers' weaknesses when handling many joins.

2.4 State-of-the-art in query optimization

Prior to the widespread adoption of machine learning in computer science research, work in query optimization did not address two key problems:

1. Many query optimizers do not guarantee efficient, hands-free operation—experts in the field with years of experience must manually tune optimizers to improve performance for production purposes and attempt to offset any errors these optimizers might make.

2. Characteristics of queries with poor performance are not taken into account for future optimization on new queries—most optimizers work from scratch with static assumptions about each query.

Several systems have been developed that address shortcomings in either the entire query optimizer or parts of it with machine learning methods. Neo replaces the default query optimizer end-to-end, using deep neural networks to determine join orders, join algorithms, and access paths (index scan, table scan, etc.) [9]. DQ [5] and ReJOIN [10] specialize, using reinforcement learning to find the best join ordering for given queries. These three systems demonstrate significant improvements over Postgres's query optimizer and prove that there is a space for machine learning methods in query optimization that address the two problems above.

Despite their improvements over Postgres, however, Neo, DQ, and ReJOIN do not isolate join algorithm selection. Bao, another recent system, applies per-query optimization hints that steer the query optimizer to an ideal query plan [8]. Bao handles only boolean options available in the RDBMS used, such as disabling nested loop joins, forcing index usage, etc. Our experiments in Chapter 4 use the same technique of applying query hints at runtime, but unlike Bao, we consider only hints related to join algorithms and work with a deep Bayesian neural network and not a tree-based convolutional neural network.

Chapter 3

Baseline Experiment with Join Algorithm Subsets in Postgres

In this chapter, we conduct our baseline experiment on the Join Order Benchmark in Postgres. For every element of the power set of the three join algorithms available in Postgres *except the empty set*, we restrict Postgres to only use the algorithms present in that element when generating query plans. We then run *all* JOB queries under that element and analyze their execution times. We use our results to show that on top of cardinality estimation errors, join algorithm selection introduces non-negligible performance loss when done incorrectly.

3.1 Hardware setup

The experiment described here was conducted on the Windows Subsystem for Linux, Ubuntu 18.04.5 flavor, in Windows 10. Appendix A describes the hardware specifications of our machine. We expect different hardware than that used here to produce different numerical results. However, given that the IMDb dataset on which the JOB runs is small enough (3.6GB) to fit into memory on modern machines (those with at least 4GB of RAM), the overall performance trends noted in our experiment should extrapolate well to other hardware configurations—Postgres should still choose the same types of operators, and performance should be affected more by the CPU used

and disk I/O.

3.2 Postgres configuration

Our experiment uses Postgres 12.4. All settings are untouched and kept similar to those at the time of installation, that is, no "pre-tuning" is done. On top of the base Postgres installation, we also have `pg_hint_plan` [11] installed as an extension, which allows users to fine-tune query plans with run-time hints; we use `pg_hint_plan` later in Chapter 5.

3.3 Method

Let L represent nested loop joins, H represent hash joins, and M represent sort-merge joins. There are $2^3 - 1$ non-empty subsets to consider: $\{L\}, \{H\}, \{M\}, \{L, H\}, \{L, M\}, \{H, M\}, \{L, H, M\}$.

Our method can be summarized in the following steps:

1. Enable the query planning settings [4] that correspond to the algorithms present in the current subset and disable all others. These include

   - `enable_hashjoin`,
   - `enable_mergejoin`,
   - `enable_nestloop`.

2. Run all 113 JOB queries and document the performance of each in a Pandas DataFrame. **The metric for performance here is wall clock time, equivalent to elapsed real time.** Ensure that Postgres runs each query with a cold start to maintain independence. This is done by restarting Postgres and clearing all system page caches with `echo 1 > /proc/sys/vm/drop_caches`.

3. Repeat until all join algorithm subsets are tested.

An extended method to account for parallelization involves running the same steps, but with `max_parallel_workers_per_gather` set to 0 in Postgres to see what happens when we restrict parallelization. This extended procedure tests the effect of modern CPUs with many cores and threads on query execution.

3.4 Results

3.4.1 Query execution times and remarks

Following the procedure in Section 3.3, we ran all 113 queries under all 7 different join algorithm subsets. A Pandas DataFrame is available for inspection and data analysis. Table A.1 shows the numbers from this DataFrame.

Note that for some queries in the DataFrame, several subsets of enabled join algorithms yield roughly the same performance, with differences of a second or two. This is due to Postgres's prioritization of some algorithms over others when there are multiple options available. For example, Postgres still uses only nested loops for Query 16b and runs it in about 70s when restricted to $\{L\}, \{L, H\}, \{L, M\}$ and $\{L, H, M\}$ due to its severe underestimation of table cardinalities. The underestimates imply that using nested loops would be better than hashing or sorting and merging a small amount of data, guiding the query optimizer to a bad choice. The final nested loop operator for the query plan with $\{L, H, M\}$ enabled predicts a cardinality of 2960, *but the actual cardinality is 3710592*, making the prediction off by a factor of over 1000. Hashing here clearly would have been the better decision.

We do not show empirical data regarding parallelization, as investigating parallelization is tangential to the goals of this work, but a summary of what we observed can be found in Section 3.4.2.

3.4.2 Key observations

From our results, we note the following:

- **There is no single subset of join algorithms that yields the best per-**

formance on all 113 JOB queries. To achieve good performance, one must empirically compare all 7 configurations as we have done here and set a specific configuration for each query before execution.

- **Poor cardinality estimation can steer the query optimizer to disastrous results [6].** In particular, Postgres severely underestimates the size of the final output and some of the intermediate tables for query 16b from the JOB, which creates a 100% bias for the nested loop algorithm when it is available. Only by disabling the nested loop algorithm can we achieve fast performance on this query and others like it. Ultimately, the cost model may be a good one, but if its cardinality inputs are not, query execution suffers in snowballing fashion.

- **Parallelization makes a huge difference for algorithms that support it.** With just two parallel workers under a hash join operator, Postgres achieves an almost 3x increase in performance over having a single worker. This is, of course, dependent on the hardware used in practice.

With these observations in mind, the problem of join algorithm selection becomes better motivated. We have shown that under default Postgres conditions, even with potentially optimal join orders (assuming some of the plans chosen by Postgres had the ideal orders) as described in existing research [5, 8, 10], join operators that use the wrong algorithms due to bad cardinality estimates introduce large performance overheads. This extends tangentially into the question of what hardware can add to performance once the right algorithms are chosen—parallelization may yield further improvement depending on how many CPU cores are available for workers and the cost model's parameters for parallel operations.

Chapter 4

Join Algorithm Selection as a Reinforcement Learning Problem

In this chapter, we formulate join algorithm selection as a *contextual* multi-armed bandit problem [7] in which each bandit arm represents a Postgres configuration that enables a subset of the three join algorithms covered in Chapter 2 and used in Chapter 3. We train a deep Bayesian network that uses Thompson sampling to learn the appropriate join algorithm subset for any JOB query given at runtime. Our numbers suggest the following:

1. For more intensive queries, such as 16b, a learned approach does quite well across at least 3000 episodes and 15-20 encounters with each query. Our deep Bayesian network allows Postgres to prioritize hash joins consistently on queries where large cardinality underestimates are made.

2. For less intensive queries that execute relatively quickly to begin with, a learned approach performs either the same as or worse than Postgres.

4.1 Reinforcement learning formulation

Formally, given agent A, which is the neural network, at time step / episode t, we provide A with context C_t representing an aggregate of information about the query

given, q_t. Specifically, we provide

1. an n-dimensional binary vector R_t (the *relation vector*), where n is the number of relations in the database queried and $R_{t_i} = 1$ if relation i is involved in any joins in q_t

2. and the upper triangular matrix (including the diagonal to handle self-joins) T_t of the binary symmetric square matrix M_t (the *join predicate matrix*) of size $n \times n$, where $M_{t_{ij}} = 1$ and $M_{t_{ji}} = 1$ if relations i and j are paired in a join predicate in q_t. We have $|T_t| = n(n+1)/2$.

Given $C_t = (R_t, M_t)$, A chooses an arm/action a_t corresponding to enabling a join algorithm subset and receives reward $r_t = -(s_t^{1.5})$, where s_t is the actual execution time of q_t after choosing a_t. Note that r_t is intended to punish poor execution time exponentially. A then moves on to episode $t+1$ and repeats this process, the ultimate goal of which is to minimize total query execution time across all episodes experienced so far.

This formulation can be extended to include the other configurable Postgres options, similar to what Bao has done.

4.2 Neural network and episode overview

We conduct our experiment with the Python `space_bandits` library [3], which provides a collection of deep Bayesian neural network implementations that use Thompson sampling, an effective method for solving contextual multi-armed bandit problems.

Our chosen network (the class `NeuralBandits`, link available **here**) performs full Bayesian linear regression on the last layer. All numbers are based on 1) the database with $n = 21$ tables created from the IMDb dataset and 2) the Join Order Benchmark queries. Relevant details include:

- 7 actions possible (configurations corresponding to the join algorithm subsets),

- input vectors of length $n + n(n+1)/2 = 252$ representing the concatenation of R_t and the flattened form of T_t from Section 4.1,
- initial learning rate 0.1,
- hidden layer sizes $[128, 128, 64]$,
- the Adam optimization algorithm,
- and 100 initial blind round-robin pulls on each arm before sampling actions from the posterior.

Note that because we have 100 initial round-robin pulls on each arm, the network does not actually start to make decisions on its own until episode 701, with the initial 700 acting as "base knowledge".

An episode is shown visually in Figure 4-1 and is summarized as follows:

1. The network receives input vector V of size 252 generated from a randomly sampled JOB query Q from the designated training set.
2. The network chooses one of the seven actions available. Each action corresponds to one of the subsets in the set of non-empty join algorithm subsets discussed in previous chapters. *Until each action has been chosen at least 100 times, we select one in round-robin fashion.*
3. All join algorithm types are disabled in Postgres.
4. The join algorithms in the subset for the action chosen are enabled in Postgres.
5. Postgres runs Q with a cold cache and reports execution time s.
6. We reward the network with $-(s^{1.5})$ to punish poor decisions more than good ones.
7. The network updates its history and its Bayesian linear regressor.

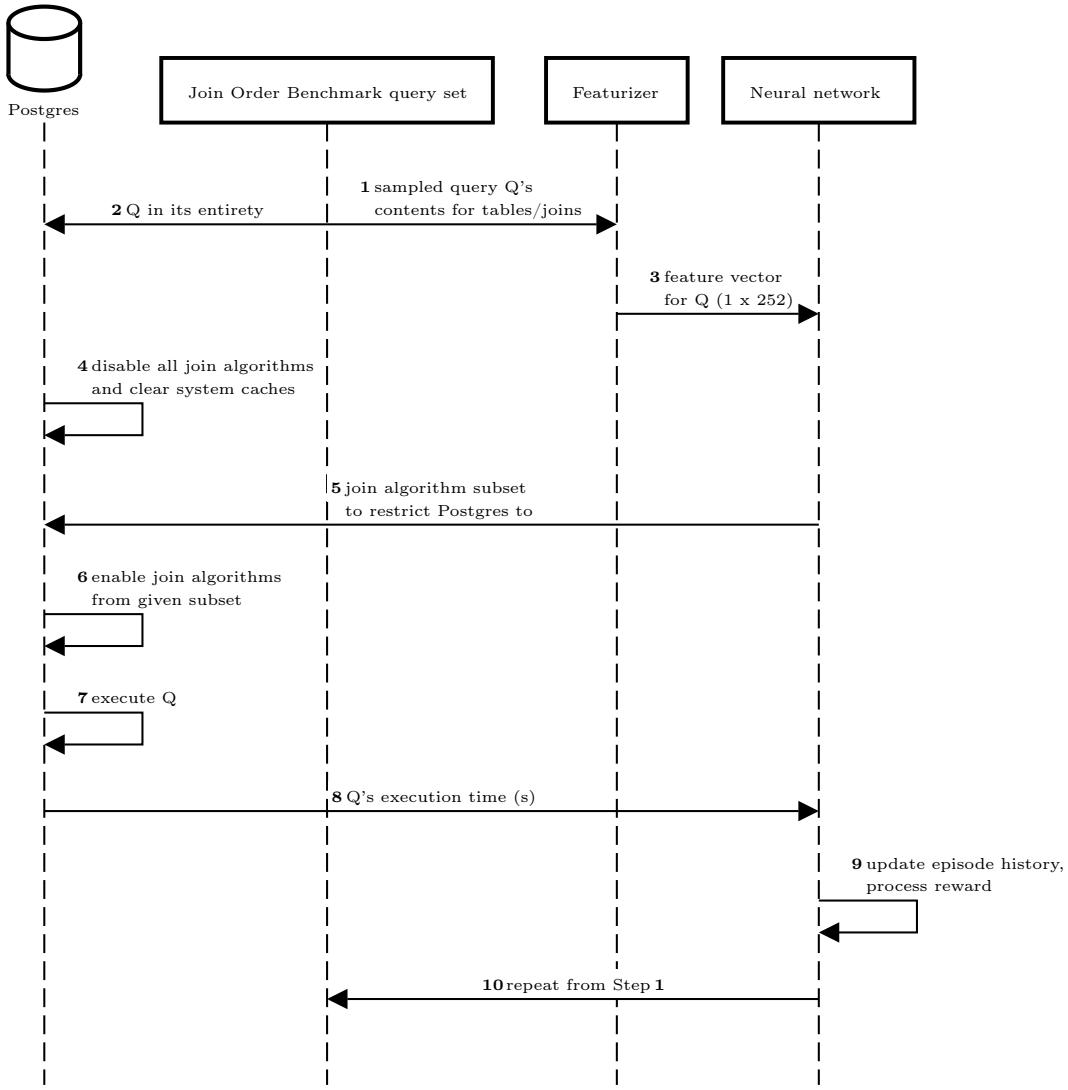

Figure 4-1: Overview of reinforcement learning method with join algorithm subsets. This flow diagram corresponds holistically with the steps outlined in Section 4.2.

`enable_nestloop`, `enable_hashjoin`, and `enable_mergejoin` are the Postgres settings used to enable the algorithms in any given subset. Join algorithm subsets in this process are analogous to Bao's hint sets [8]: each algorithm in a subset implies a hint to allow that algorithm during query execution.

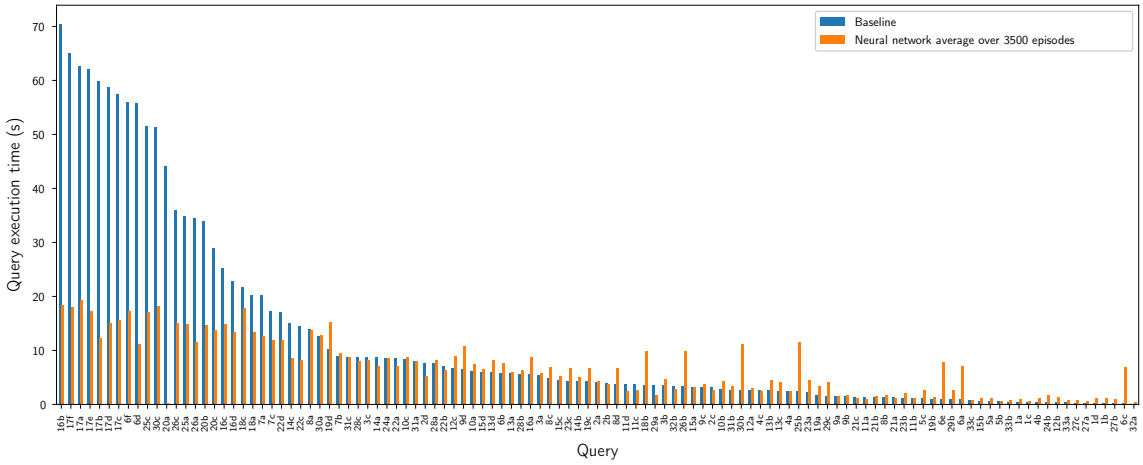

Figure 4-2: A query execution time comparison of Postgres's query optimizer and our deep Bayesian neural network. Queries are sorted in descending order by baseline Postgres execution time. The orange data point for each query is the average of its execution times (when encountered by our neural network) across all 3500 episodes.

4.3 Results and discussion

We trained our deep Bayesian network from scratch for 3500 episodes on the entire JOB. Training overall was quick, taking about 10 hours. We did not split the JOB into training and testing sets, opting instead to think of our training process as a stream of arbitrary user inputs from which to build knowledge of the ideal decisions. Thus, each episode's query was a random sample from the JOB. Over time, as the network encountered each query more and more, its performance increased dramatically, with many intensive queries such as 16b showing consistent speed-ups after only 10-15 encounters. We show the baseline time versus the average of all the encounters across the 3500 episodes for each query in Figure 4-2. The average baseline execution time across all 113 queries was 12.00s, and the average execution time across all 3500 episodes with our neural network was 6.92s, a 42% decrease from the baseline.

Note that many queries which were already efficient under Postgres's query optimizer (the right side of Figure 4-2) performed worse under our neural network. This is due to the network learning that hash joins are, on average, more ideal than nested loops, which leads to occasional decisions to use hash joins (by restricting L, and per-

haps M, from the chosen subset) even on queries for which nested loops are the most efficient. Still, with a 42% decrease in average execution time, our network arguably performs well above the baseline.

Given our results, we see this particular method as a viable use case for read-heavy databases that encounter some fixed set of frequent queries. Upon initial encounters, some of these queries would indeed run slowly, but after several more, the network would know what the approximately ideal join algorithm subsets are for each. With this method comes potential weakness for queries that are already sufficiently optimized, but the gains for more intensive queries offset this weakness.

Chapter 5

k-edits for Join Algorithm Selection

In Chapter 4, we provided *coarse hints*, those that affect query execution on a broader level, to the query optimizer through the seven join algorithm subsets. Each of these subsets served as a collection of hints to, system-wide for Postgres, disable those not included and enable those included.

In this chapter, we shift our focus to *fine hints* and address the following question: is it possible to identify and fix some number of "bad joins" in a query plan without disabling any join types at all? We first describe how specific joins can be modified in both Postgres and SQL Server. We then introduce the idea of *k-edits*, suggesting k modifications to the join algorithms in each query plan chosen by the query optimizer. We finally conduct an experiment that shows that, across all the JOB queries, 1-edits are optimal for performance improvements, but are only doable efficiently with more fluid systems like Postgres. We also make a case for the intractability of the set of k-edits for $k \geq 3$, which renders the exploitation of anything beyond 1 and 2-edits impractical.

5.1 Specifying join hints in Postgres and SQL Server

Postgres. We use `pg_hint_plan` [11], a third-party Postgres extension, to provide explicit hints as to how certain tables should be joined together. As far as we know, no official solution from Postgres exists, short of modifying the original source code,

to provide run-time query hints.

As a practical example of `pg_hint_plan`'s operation, take Query 17f, shown in Listing 5.1. Each line containing `AND` and two tables represents a potential join where the condition is equality between the columns mentioned.

Listing 5.1: Query 17f from the original Join Order Benchmark.

```
SELECT
    MIN(n.name) AS member_in_charnamed_movie
FROM
    cast_info AS ci,
    company_name AS cn,
    keyword AS k,
    movie_companies AS mc,
    movie_keyword AS mk,
    name AS n,
    title AS t
WHERE
    k.keyword = 'character-name-in-title'
    AND n.name LIKE '%B%'
    AND n.id = ci.person_id
    AND ci.movie_id = t.id
    AND t.id = mk.movie_id
    AND mk.keyword_id = k.id
    AND t.id = mc.movie_id
    AND mc.company_id = cn.id
    AND ci.movie_id = mc.movie_id
    AND ci.movie_id = mk.movie_id
    AND mc.movie_id = mk.movie_id;
```

We can explicitly state which join algorithms to use for pairs of tables by prepending a comment to the query. We might prepend `/*+ HashJoin(n ci) */` or, if we want to specify more than one, `/*+ MergeJoin(ci mc) NestLoop(ci mk) */`. At runtime, `pg_hint_plan` detects these prepended comments and changes the query optimizer's output.

SQL Server. Unlike Postgres, which requires extensions or source-code-level modification, SQL Server provides built-in syntax to explicitly force join algorithms on pairs of tables. However, with this benefit comes stricter standards on how the SQL should be written for every query: to specify explicit join algorithms, queries must be ANSI-compliant, that is, inner joins must be declared with `INNER JOIN` and

not implicitly with `FROM-WHERE` clauses as 17f does in Listing 5.1. We can rewrite every `INNER JOIN` as `INNER {J} JOIN`, where `J` can be among `LOOP`, `HASH`, and `MERGE`.

Note that the original JOB queries are *not* ANSI-compliant—we rewrite them in Section 5.3 to work with the SQL Server part of our experiment.

5.2 k-edits

We define k-edits as *query plans for which exactly k join algorithms are explicitly specified*.

More formally, let Q be a query of interest. Let n be the number of join predicates—explicit join statements between pairs of tables, ex. `AND n.id = ci.person_id` and `INNER JOIN keyword k ON mk.keyword_id = k.id`—in Q. Let $k \leq n$ be the number of predicates whose join algorithms we want to modify and m be the number of distinct join algorithms available (for Postgres and SQL Server, $m = 3$).

Given k, m, and n, we have $\binom{n}{k}$ total possible combinations of predicates to modify. Each combination has k predicates that each needs to be assigned one of m join algorithms. This gives us a search space, or equivalently, the set of k-edits, of size $\binom{n}{k}m^k$. One can see that this search space grows exponentially with k.

5.3 Method

Here, we divide our experiment into two parts, as what needs to be done differs significantly between Postgres and SQL Server.

Postgres. We first run any given JOB query with a cold cache to establish a baseline for comparison with our k-edits method. We then parse its contents to find all join predicates of the form `r1.attr1 = r2.attr2` and extract the set R_J of all relations involved in ≥ 1 join predicate. We then enumerate the set of 1-edits involving elements from R_J and prepend each 1-edit to the query before running the

query with a cold cache. For each 1-edit, if the query plan Postgres generates is still the same with that 1-edit included, we skip it. After every execution, we restore the query to how it was before to prepare for the next 1-edit.

SQL Server. We run the given query as is with a cold cache for the baseline time. We convert the query to ANSI-compliant form (multiple solutions exist on the Web for automatically rewriting queries). After conversion, we parse the new, semantically equivalent query to find all instances of the form `INNER JOIN {x}` that involve only two relations. With SQL Server's built-in query hint functionality, we enumerate the set of 1-edits and add the relevant algorithms in place of `{x}`. As with Postgres, we run the query with a cold cache to eliminate bias.

5.4 Results

We conducted, for Postgres and SQL Server, their respective methods. All results were gathered on the same machine described in Appendix A. Separate baseline times are available in Table A.2 to serve as a comparison between the Postgres and SQL Server query optimizers.

Postgres. Our Postgres results in Listing B.1 (too large to fit here) show, for each JOB query, the baseline time, hints applied that resulted in new query plans, query execution time under those hints, and total number of hints skipped. Of the 113 queries, 63, or 55%, were sped up with 1-edits compared to the baseline. The average improvement was 7.9s across all queries that executed faster. Notably, almost all hints generated were skipped, with some queries skipping *all* hints generated for them entirely. It is clear that our method was not enough to force the query optimizer to make many changes to the default query plans.

We also tested $k = 3$ on Query 6f, where $m = 3$ and $n = 5$, giving us a total of $\binom{5}{3}3^3 = 270$ 3-edits. Listing 5.2 shows an excerpt of these edits and their execution times. There was no large difference between all the 3-edits, with all unable to bring the execution time under 19 seconds. We believe $k \geq 3$ to be highly intractable—assuming, in the best-case scenario, each 3-edit takes 20 seconds to execute and that

there are about 100 unskipped 3-edits, that gives us $100 \cdot 20 = 2000$ seconds, or 33 minutes, just to explore the set of 3-edits for one query. Repeating this for every query at runtime is clearly not feasible, and for successive k, the search space grows exponentially.

Listing 5.2: 3-edits on Query 6f. We show 10 out of the 108 3-edits used for brevity.

```
Working with query 6f.sql.
Baseline time: 63.51s.
6f.sql: Hint HashJoin(ci mk) HashJoin(ci n) HashJoin(k mk) resulted in a runtime of
    20.75s.
6f.sql: Hint HashJoin(ci mk) HashJoin(ci n) MergeJoin(k mk) resulted in a runtime of
    19.73s.
6f.sql: Hint HashJoin(ci mk) NestLoop(ci n) HashJoin(k mk) resulted in a runtime of
    20.5s.
6f.sql: Hint HashJoin(ci mk) NestLoop(ci n) MergeJoin(k mk) resulted in a runtime of
    19.84s.
6f.sql: Hint HashJoin(ci mk) MergeJoin(ci n) HashJoin(k mk) resulted in a runtime of
    20.62s.
6f.sql: Hint HashJoin(ci mk) MergeJoin(ci n) MergeJoin(k mk) resulted in a runtime of
     19.96s.
6f.sql: Hint NestLoop(ci mk) HashJoin(ci n) HashJoin(k mk) resulted in a runtime of
    20.61s.
6f.sql: Hint NestLoop(ci mk) HashJoin(ci n) MergeJoin(k mk) resulted in a runtime of
    19.86s.
6f.sql: Hint NestLoop(ci mk) NestLoop(ci n) HashJoin(k mk) resulted in a runtime of
    20.42s.
6f.sql: Hint NestLoop(ci mk) NestLoop(ci n) MergeJoin(k mk) resulted in a runtime of
    19.76s.
. . .
. . .
. . .
Skipped 162 hints out of 270 available.
```

SQL Server. Our SQL Server method showed no improvements over the baseline and instead slowed execution by at least two orders of magnitude; we conducted this method on only Query 16b and did not continue further. For Query 16b, we found the rewriting process extremely complex—to isolate all joins, basic conversion to the ANSI standard was not enough, as demonstrated by the `JOIN-ON-AND` clauses in Listing 5.3, which still have not isolated all pairs of tables being joined. We rewrote the query again in a nested fashion that corresponds to the join order given by SQL

Server, shown in Listing 5.4, but SQL Server largely changed parts of the query plan when we specified explicit joins, even for queries outside of 16b.

Figure B-1 shows the default SQL Server plan for 16b. When the full join-isolated form of 16b (Listing 5.4) is analyzed, we get the same query plan as the default. When, however, explicit nested loop joins are indicated for every `INNER JOIN` to match the default plan, SQL Server introduces table spools, shown in Figure B-2. Ultimately, we terminated execution when the first join in Figure B-2 continued to run for over five hours. The same occurs even if table spools are disabled.

Listing 5.3: Query 16b in ANSI-compliant form.

```
SELECT
  MIN(an.name) AS
       cool_actor_pseudonym,
  MIN(t.title) AS
       series_named_after_char
FROM aka_name an
  INNER JOIN cast_info ci
    ON an.person_id = ci.person_id
  INNER JOIN movie_companies mc
    ON ci.movie_id = mc.movie_id
  INNER JOIN company_name cn
    ON mc.company_id = cn.id
  INNER JOIN movie_keyword mk
    ON ci.movie_id = mk.movie_id
    AND mc.movie_id = mk.movie_id
  INNER JOIN keyword k
    ON mk.keyword_id = k.id
  INNER JOIN name n
    ON an.person_id = n.id
    AND n.id = ci.person_id
  INNER JOIN title t
    ON ci.movie_id = t.id
    AND t.id = mk.movie_id
    AND t.id = mc.movie_id
WHERE
  cn.country_code = '[us]'
    AND k.keyword = 'character-name
         -in-title';
```

Listing 5.4: Query 16b in full join-isolated form. The nested join order here is the same as that provided by SQL Server's query optimizer.

```
select min(j.name), min(j.title) from (
select ci.name, ci.title from name n inner join (
select an.name, ci.person_id, ci.title from aka_name an inner join (
select an.id, ci.person_id, ci.title from aka_name an inner join (
select ci.person_id, t.title from cast_info ci inner join (
select ci.id, t.title from cast_info ci inner join (
select mc.title, mc.t_id as id from company_name cn inner join (
select mc.company_id, mc2.title, mc2.t_id from movie_companies mc inner
     join (
select mc.id, t.title, t.id as t_id from movie_companies mc inner join (
select title, id from title t inner join (
select movie_id from movie_keyword mk inner join (
select mk.id from (
select id from keyword k where k.keyword = 'character-name-in-title') k
     inner join
movie_keyword mk on mk.keyword_id = k.id
) mk2 on mk.id = mk2.id
) mk on mk.movie_id = t.id
) t on mc.movie_id = t.id
) mc2 on mc.id = mc2.id
) mc on mc.company_id = cn.id where cn.country_code = '[us]'
) t on ci.movie_id = t.id
) t on ci.id = t.id
) ci on an.person_id = ci.person_id
) ci on an.id = ci.id
) ci on n.id = ci.person_id
) j;
```

In Section 5.5, we discuss insights from this k-edits experiment and describe changes to our methodology that may generate clearer results.

5.5 Discussion

Our experiment with k-edits suggests two ideas to consider:

1. To increase query performance as efficiently as possible when performing manual join algorithm selection, fluid systems like Postgres that do not enforce standards on execution are ideal. To adapt to any kind of query, including those that are not ANSI-compliant, we cannot rely on a system like SQL Server that forces rewrites and changes query plans entirely when even one edit is made. The goal is to take *any* query given and achieve efficient execution with as little processing overhead as possible—implementing an intelligent database query rewriter to accommodate the quirks of various RBDMSs is beyond the scope of our work.

2. To improve Postgres performance with 1-edits even further and prevent skipping of hints, any `pg_hint_plan` hint comment blocks must also account for the default join order provided by the query optimizer. Just providing the 1-edits by themselves causes Postgres to alter only the leaf node joins in a query plan that match an edit, as any join above the initial one is between intermediate join results (not original tables mentioned in the 1-edits) and another original table. This suggests that our method in this chapter may be more practical for bushy query plans with many leaf nodes than for left-deep query plans.

   To have more hints take effect, we must add a `Leading(t5 (t4 (t3 (...))))` construct from `pg_hint_plan` that matches the exact join order provided by Postgres for every query. We expect this method to require significantly more complex parsing, as the join algorithm hints must take a different form as well.

With our results, it is not immediately clear that k-edits can provide consistent, large speed-ups, but our numbers are a step in the right direction.

Chapter 6

Comparison of Results from Experiments

In this chapter, we compare the results of our three experiments on Postgres and single out reinforcement learning with coarse hints as the best-performing method for query speed-ups *among our methods in this work*.

We show a comparison of our three experiments for the six slowest JOB queries in Figure 6-1. Data for all 113 queries is available, but the six queries shown here are representative enough of the more intensive queries from the JOB. We do not account for potential changes to our methods—it may very well be the case that a more refined approach to k-edits, such as the one we describe at the end of Chapter 5, outperforms reinforcement learning. We also do not go further with our SQL Server results, as we worked with SQL Server in only Chapter 5.

6.1 Deep Bayesian neural network versus Postgres

Across the 6 queries in Figure 6-1, our neural network performed the best, with significant speed-ups compared to default Postgres after about 10-20 encounters with each query. In addition, of the 18 intensive queries from the JOB that each took at least 25 seconds to execute under default conditions (Table C.1), 100% executed in less than half their original times after learning from scratch from 0 to 3500 episodes.

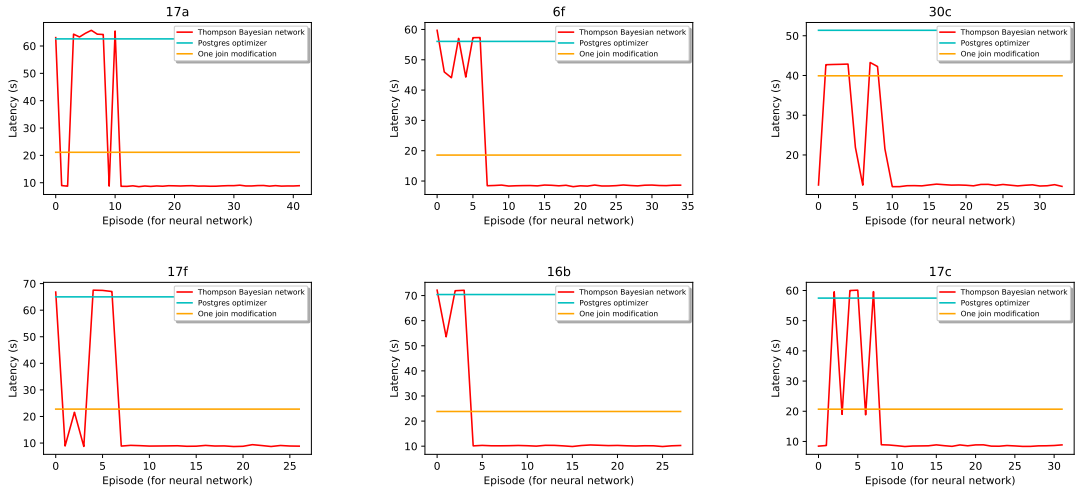

Figure 6-1: Postgres performance comparison of the Postgres query optimizer, our deep Bayesian network, and 1-edits on the six slowest JOB queries. The blue represents Postgres's optimizer, the red represents our deep Bayesian network, and the orange represents the best execution time with 1-edits. The x-axis represents sequential encounters with the query for the neural network; it is a nonfactor for default Postgres and 1-edit times.

20a, an extreme improvement, went from 44.14s to just 0.21s. Least noticeable was 16c, which went from 25.32s to 9.32s.

6.2 1-edits versus Postgres

1-edits also beat default Postgres, but not as much as we had hoped, with our neural network still outperforming 1-edits significantly. Notably, for all six queries in Figure 6-1, our neural network converged to a time below half that of the best 1-edit. We suspect that the changes outlined in Section 5.5 would increase the amount of hints explored by Postgres and, consequently, its best time for most JOB queries.

Chapter 7

Conclusion

We showed in this thesis that poor cardinality estimation produces a snowball effect in query plan generation and execution, that reinforcement learning with coarse hints is quick to bootstrap and quick to outperform, and that 1-edits independent of join orders provide decent, but not ideal, speed-ups.

A significant consequence of this work is highlighting machine learning's potential for query optimization research. We hope our methods and data will be used to explore different approaches to guiding conventional cost-based query optimizers towards the best query plans. For example, we demonstrated a brute-force search over the set of 1-edits for each JOB query in this work, but what about training a neural network to learn the most ideal 1-edits *with join orders accounted for* given information about a query? What about a hybrid approach that marries both join algorithm subset isolation and k-edits, where intensive queries are improved with either coarse hints or fine edits chosen by a neural network? These exciting next steps and our methods in this work share in common the ideas of *exploiting past knowledge* about both failures and successes and *looking for correlations* where conventional research does not—naturally, these ideas play well into machine learning.

We also see tremendous potential for developers and their productivity. Should a machine learning system that can accurately and consistently improve performance across any database and query set (not just for IMDb and the Join Order Benchmark) be made easily accessible to the public, developers would be able to shift gears

from hours of manual RDBMS tuning to hours of mission-critical programming. We envision many models, each trained on a separate production database. With reinforcement learning, these models can improve as more and more queries arrive, decreasing production latency over time.

Appendix A

Baseline JOB execution times in PostgreSQL and Microsoft SQL Server

All the results in these tables were obtained on the Windows Subsystem for Linux (v2.0) with the following hardware:

- Intel Core i9-10900K, 10 cores / 20 threads @ 4.0GHz,
- 64GB DDR4 2400MHz RAM, 24GB of which was allocated to WSL,
- Samsung 850 EVO 1TB SSD,
- NVIDIA GeForce RTX 2080, 8GB GDDR6 memory.

All numbers shown are in seconds.

Table A.1: Baseline JOB execution times in Postgres under all seven join algorithm subsets.

	$\{L\}$	$\{H\}$	$\{M\}$	$\{L,H\}$	$\{L,M\}$	$\{H,M\}$	$\{L,H,M\}$
1a	0.28	1.56	8.03	0.39	0.34	1.50	0.39
1b	0.07	1.51	8.21	0.23	0.08	1.51	0.23
1c	0.28	1.53	1.77	0.33	0.29	1.55	0.35
1d	0.07	1.47	7.27	0.23	0.08	1.59	0.24
2a	3.68	1.91	11.16	4.19	4.21	1.97	4.20
2b	3.24	1.93	10.94	3.78	3.89	1.93	3.87
2c	3.22	1.19	4.13	3.24	3.15	1.23	3.18
2d	7.69	2.01	9.64	7.90	8.07	2.01	7.97
3a	5.38	5.19	11.79	5.05	5.52	5.14	5.34
3b	3.48	4.85	11.35	3.01	3.44	4.70	3.42
3c	8.78	5.11	16.83	8.45	8.72	5.22	8.47
4a	2.21	1.65	7.91	2.29	2.31	1.68	2.42
4b	1.48	1.63	2.22	0.31	1.57	1.70	0.31
4c	2.14	1.64	8.44	2.58	2.60	1.65	2.59
5a	0.46	0.63	10.58	0.44	0.46	0.67	0.45
5b	0.43	0.62	4.76	0.44	0.45	0.66	0.46
5c	1.08	5.01	10.80	1.01	1.03	4.99	0.98
6a	0.75	8.10	10.36	0.72	0.86	7.97	0.84
6b	5.80	8.35	3.96	5.45	5.64	8.02	5.39
6c	0.11	8.34	3.94	0.11	0.12	8.26	0.11
6d	57.03	7.93	10.82	57.08	56.38	8.56	56.72
6e	0.85	8.00	10.26	0.84	0.83	7.87	0.68
6f	57.25	8.57	43.81	56.12	56.79	8.47	56.64
7a	21.22	8.85	11.63	20.49	21.28	8.75	20.69

Continued on next page

Table A.1: Baseline JOB execution times in Postgres under all seven join algorithm subsets.

	$\{L\}$	$\{H\}$	$\{M\}$	$\{L,H\}$	$\{L,M\}$	$\{H,M\}$	$\{L,H,M\}$
7b	9.19	8.99	11.14	9.14	9.18	8.92	8.82
7c	17.45	10.11	14.60	17.03	17.73	10.28	16.47
8a	14.22	8.25	14.49	14.22	14.33	8.48	14.08
8b	1.14	8.14	8.66	1.03	1.12	8.25	1.04
8c	41.25	9.32	20.80	4.87	14.50	9.42	4.74
8d	35.55	8.93	20.90	3.71	13.10	8.61	3.62
9a	1.51	9.56	22.37	1.49	1.54	9.73	1.48
9b	1.41	9.12	15.90	1.44	1.47	9.65	1.46
9c	3.14	9.69	23.24	3.03	3.20	9.45	3.06
9d	19.24	9.76	26.57	6.61	17.14	10.02	6.46
10a	11.17	7.69	19.62	6.47	10.01	8.01	6.54
10b	4.66	7.96	9.96	2.74	3.93	7.78	2.73
10c	5.77	7.87	16.54	8.09	30.58	8.16	8.07
11a	1.22	2.02	0.81	1.14	1.21	0.53	1.23
11b	0.97	1.89	1.48	1.18	1.18	1.17	1.19
11c	2.95	1.94	0.63	3.23	3.16	0.50	3.02
11d	3.18	1.84	0.74	3.28	3.73	0.40	3.68
12a	2.41	4.97	5.98	2.47	2.46	4.90	2.47
12b	0.09	5.03	6.71	0.25	0.11	5.24	0.25
12c	17.65	4.85	12.55	6.82	17.64	4.72	6.79
13a	15.07	5.09	11.40	5.74	14.94	5.12	5.73
13b	7.60	5.23	7.34	2.46	7.41	5.15	2.49
13c	7.39	4.93	7.20	2.43	7.38	4.94	2.44
13d	13.35	6.71	13.07	5.87	13.24	6.49	5.90

Continued on next page

Table A.1: Baseline JOB execution times in Postgres under all seven join algorithm subsets.

	$\{L\}$	$\{H\}$	$\{M\}$	$\{L,H\}$	$\{L,M\}$	$\{H,M\}$	$\{L,H,M\}$
14a	8.59	5.15	6.40	8.56	8.44	5.35	8.72
14b	4.32	5.41	5.80	3.83	3.79	5.19	4.20
14c	15.39	5.35	11.85	14.99	15.52	5.23	15.34
15a	3.32	5.27	14.20	3.27	3.31	5.60	3.31
15b	0.49	5.67	8.09	0.48	0.51	5.64	0.44
15c	4.49	5.21	14.79	4.59	4.20	5.04	4.43
15d	5.61	5.63	14.48	5.60	5.74	5.40	5.50
16a	6.60	8.95	29.73	5.78	6.57	8.71	5.78
16b	70.95	9.95	52.65	70.53	70.52	10.07	69.86
16c	25.58	9.16	51.54	26.33	26.11	9.05	25.53
16d	23.22	9.20	51.30	22.62	22.85	8.96	22.06
17a	64.12	8.72	21.04	63.62	63.66	8.83	63.71
17b	62.02	9.05	20.06	61.01	60.51	8.54	61.25
17c	59.71	8.72	18.79	58.82	59.11	8.74	58.89
17d	61.76	8.48	18.92	60.22	60.40	8.55	59.81
17e	64.02	9.20	52.04	64.21	64.01	9.14	64.18
17f	66.60	8.91	21.43	66.65	66.56	9.00	67.02
18a	22.18	11.94	19.16	20.47	22.36	11.32	20.62
18b	3.67	11.36	20.45	3.67	5.03	11.12	3.60
18c	64.19	11.60	21.24	22.18	22.25	11.39	22.99
19a	1.52	12.86	25.74	1.54	1.56	12.61	1.53
19b	0.86	12.29	16.93	0.87	0.93	13.03	0.97
19c	8.39	13.03	26.42	4.39	4.91	12.85	7.90
19d	44.00	13.57	30.87	13.27	13.56	13.62	13.13

Continued on next page

Table A.1: Baseline JOB execution times in Postgres under all seven join algorithm subsets.

	$\{L\}$	$\{H\}$	$\{M\}$	$\{L,H\}$	$\{L,M\}$	$\{H,M\}$	$\{L,H,M\}$
20a	0.07	0.18	0.20	0.06	0.09	0.20	0.08
20b	34.19	9.06	13.96	33.12	34.08	9.25	33.66
20c	29.45	9.17	20.54	29.44	29.45	9.14	30.00
21a	0.93	5.55	0.96	1.28	1.28	1.30	1.29
21b	1.23	5.47	0.97	1.23	1.15	1.21	1.10
21c	1.25	5.57	1.43	1.15	1.02	1.05	1.22
22a	8.69	5.62	7.24	8.56	8.74	5.39	8.36
22b	7.07	5.59	7.05	6.85	7.35	5.54	7.18
22c	15.02	5.79	9.18	15.05	15.02	5.91	14.82
22d	17.92	5.77	13.82	18.49	17.84	5.47	16.72
23a	7.07	5.48	8.78	2.16	7.41	5.62	2.30
23b	1.15	5.33	8.44	1.13	1.17	4.86	1.18
23c	10.37	5.56	8.59	4.22	10.38	5.34	4.25
24a	8.00	13.09	19.89	6.54	9.12	13.05	7.68
24b	0.28	12.93	17.76	0.29	0.31	13.53	0.53
25a	37.44	11.83	21.63	36.37	35.91	12.26	35.22
25b	2.43	11.99	13.21	2.43	2.58	12.16	2.38
25c	52.65	12.24	22.59	52.03	53.92	12.39	52.57
26a	23.22	9.42	21.23	13.09	27.30	9.65	20.80
26b	3.73	9.52	21.73	2.42	3.90	9.61	2.86
26c	62.49	9.60	21.32	40.24	31.04	9.71	29.89
27a	0.94	5.68	0.98	0.22	0.23	0.58	0.25
27b	1.24	5.56	0.94	0.21	0.17	1.21	0.19
27c	0.41	5.76	1.04	1.26	0.76	1.11	0.26

Continued on next page

Table A.1: Baseline JOB execution times in Postgres under all seven join algorithm subsets.

	$\{L\}$	$\{H\}$	$\{M\}$	$\{L, H\}$	$\{L, M\}$	$\{H, M\}$	$\{L, H, M\}$
28a	6.41	5.79	14.74	19.61	18.47	5.89	6.50
28b	5.33	5.62	12.63	6.73	5.12	5.73	7.17
28c	13.97	5.98	14.56	8.55	13.91	6.04	7.47
29a	0.48	14.43	15.42	3.42	0.40	14.61	0.68
29b	0.55	14.78	15.93	0.58	0.37	14.94	1.24
29c	3.25	14.05	28.33	1.24	1.23	14.97	4.56
30a	17.42	12.33	21.50	20.68	20.38	12.15	20.08
30b	2.72	12.23	13.88	2.65	2.68	12.00	2.68
30c	41.45	12.42	21.58	42.07	42.40	12.27	41.88
31a	8.78	12.83	21.76	8.29	5.96	12.29	8.39
31b	2.75	12.55	13.30	2.61	2.73	12.49	2.74
31c	9.04	12.56	22.68	8.17	5.91	12.35	8.35
32a	0.08	0.72	2.70	0.07	0.08	0.69	0.08
32b	3.58	1.64	2.68	3.50	3.29	1.35	3.10
33a	0.37	1.89	1.82	0.55	0.58	2.71	0.28
33b	0.44	1.90	1.61	0.40	0.46	1.86	0.47
33c	0.58	1.92	3.50	0.81	0.40	1.86	0.38

Table A.2: Baseline times in Postgres and SQL Server with all joins enabled and no other settings changed. This is equivalent to running all queries under $\{L, H, M\}$.

Query	PostgreSQL	Microsoft SQL Server
1a	0.45	0.70

Continued on next page

Table A.2: Baseline times in Postgres and SQL Server with all joins enabled and no other settings changed. This is equivalent to running all queries under $\{L, H, M\}$.

Query	PostgreSQL	Microsoft SQL Server
1b	0.36	2.51
1c	0.47	0.56
1d	0.38	1.31
2a	4.72	3.14
2b	3.71	3.11
2c	3.62	3.06
2d	8.78	3.23
3a	5.30	2.55
3b	4.12	1.42
3c	10.33	4.39
4a	2.84	0.75
4b	0.41	0.55
4c	2.61	0.79
5a	0.57	3.77
5b	0.61	0.40
5c	1.19	3.87
6a	1.05	0.42
6b	6.66	1.55
6c	0.21	0.16
6d	65.57	6.66
6e	1.07	0.43
6f	66.09	6.56
7a	24.75	4.46

Continued on next page

Table A.2: Baseline times in Postgres and SQL Server with all joins enabled and no other settings changed. This is equivalent to running all queries under $\{L, H, M\}$.

Query	PostgreSQL	Microsoft SQL Server
7b	10.76	5.19
7c	19.78	74.87
8a	16.07	4.86
8b	1.26	1.07
8c	5.18	7.49
8d	4.18	6.54
9a	1.76	4.98
9b	1.72	2.19
9c	3.49	5.64
9d	7.17	7.00
10a	7.20	3.16
10b	3.18	2.94
10c	9.29	4.98
11a	1.49	0.34
11b	1.48	0.60
11c	4.26	0.53
11d	4.46	0.55
12a	2.88	1.68
12b	0.38	1.80
12c	7.45	3.45
13a	6.37	7.58
13b	2.89	1.49
13c	2.84	1.38

Continued on next page

Table A.2: Baseline times in Postgres and SQL Server with all joins enabled and no other settings changed. This is equivalent to running all queries under $\{L, H, M\}$.

Query	PostgreSQL	Microsoft SQL Server
13d	6.38	7.17
14a	9.73	6.92
14b	4.33	5.38
14c	17.87	7.39
15a	3.64	1.76
15b	0.65	0.84
15c	4.94	3.11
15d	6.35	5.11
16a	7.43	1.31
16b	81.67	35.98
16c	29.54	12.18
16d	27.26	10.82
17a	73.72	22.58
17b	71.34	13.79
17c	69.46	12.63
17d	69.82	14.87
17e	73.96	22.75
17f	77.03	18.47
18a	22.98	6.47
18b	4.31	3.38
18c	24.48	12.41
19a	1.76	9.86
19b	1.21	1.12

Continued on next page

Table A.2: Baseline times in Postgres and SQL Server with all joins enabled and no other settings changed. This is equivalent to running all queries under $\{L, H, M\}$.

Query	PostgreSQL	Microsoft SQL Server
19c	9.21	27.76
19d	14.80	330.01
20a	0.17	2.08
20b	38.94	5.17
20c	33.24	5.42
21a	1.48	0.43
21b	1.58	0.40
21c	1.40	0.46
22a	10.14	9.04
22b	8.24	8.95
22c	16.77	9.38
22d	20.05	8.62
23a	2.58	1.94
23b	1.42	1.50
23c	4.91	1.95
24a	8.20	2.63
24b	0.65	0.45
25a	40.43	12.13
25b	2.84	1.06
25c	59.63	20.49
26a	26.05	5.69
26b	3.28	1.56
26c	33.40	6.83

Continued on next page

Table A.2: Baseline times in Postgres and SQL Server with all joins enabled and no other settings changed. This is equivalent to running all queries under $\{L, H, M\}$.

Query	PostgreSQL	Microsoft SQL Server
27a	0.36	0.42
27b	0.28	0.35
27c	0.36	0.41
28a	7.03	7.58
28b	8.62	6.18
28c	9.21	7.99
29a	0.94	0.42
29b	1.73	0.41
29c	4.92	1.95
30a	22.85	8.00
30b	3.06	1.18
30c	47.92	8.94
31a	10.40	11.26
31b	3.15	27.38
31c	10.30	13.83
32a	0.17	0.10
32b	3.69	1.30
33a	0.39	13.25
33b	0.51	0.28
33c	0.48	0.23

Appendix B

k-edits Results

Listing B.1: 1-edits performed on all 113 JOB queries in Postgres. The output lines show for each query 1) the query processed, 2) the baseline execution time with no modification, 3) edits used, and 4) the total count for skipped edits.

```
- - - - - - - - - - - - -
Working with query 33a.sql.
Baseline time: 0.36s.
Hints used for modifications: [].
Skipped 57 hints out of 57 available.
 - - - - - - - - - - - - -
Working with query 14a.sql.
Baseline time: 8.41s.
14a.sql: Hint HashJoin(k mk) resulted in a runtime of 5.96s.
14a.sql: Hint MergeJoin(k mk) resulted in a runtime of 5.95s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 28 hints out of 30 available.
 - - - - - - - - - - - - -
Working with query 1c.sql.
Baseline time: 0.41s.
1c.sql: Hint NestLoop(it mi_idx) resulted in a runtime of 0.34s.
1c.sql: Hint MergeJoin(it mi_idx) resulted in a runtime of 0.81s.
Hints used for modifications: ['NestLoop(it mi_idx)', 'MergeJoin(it mi_idx)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - - -
Working with query 8a.sql.
Baseline time: 14.31s.
8a.sql: Hint NestLoop(mc cn) resulted in a runtime of 14.4s.
8a.sql: Hint MergeJoin(mc cn) resulted in a runtime of 14.72s.
Hints used for modifications: ['NestLoop(mc cn)', 'MergeJoin(mc cn)'].
```

```
Skipped 22 hints out of 24 available.
 - - - - - - - - - - - -
Working with query 3a.sql.
Baseline time: 5.3s.
3a.sql: Hint HashJoin(k mk) resulted in a runtime of 2.21s.
3a.sql: Hint MergeJoin(k mk) resulted in a runtime of 2.65s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 10 hints out of 12 available.
 - - - - - - - - - - - -
Working with query 32b.sql.
Baseline time: 3.02s.
32b.sql: Hint HashJoin(mk k) resulted in a runtime of 0.6s.
32b.sql: Hint MergeJoin(mk k) resulted in a runtime of 0.61s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 16 hints out of 18 available.
 - - - - - - - - - - - -
Working with query 13b.sql.
Baseline time: 2.61s.
13b.sql: Hint NestLoop(it miidx) resulted in a runtime of 7.31s.
13b.sql: Hint MergeJoin(it miidx) resulted in a runtime of 2.04s.
Hints used for modifications: ['NestLoop(it miidx)', 'MergeJoin(it miidx)'].
Skipped 31 hints out of 33 available.
 - - - - - - - - - - - -
Working with query 21c.sql.
Baseline time: 1.07s.
21c.sql: Hint HashJoin(mk k) resulted in a runtime of 0.23s.
21c.sql: Hint MergeJoin(mk k) resulted in a runtime of 0.23s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 40 hints out of 42 available.
 - - - - - - - - - - - -
Working with query 6b.sql.
Baseline time: 5.5s.
6b.sql: Hint HashJoin(k mk) resulted in a runtime of 0.96s.
6b.sql: Hint MergeJoin(k mk) resulted in a runtime of 0.95s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - -
Working with query 15c.sql.
Baseline time: 4.37s.
15c.sql: Hint NestLoop(it1 mi) resulted in a runtime of 4.39s.
15c.sql: Hint MergeJoin(it1 mi) resulted in a runtime of 4.45s.
Hints used for modifications: ['NestLoop(it1 mi)', 'MergeJoin(it1 mi)'].
Skipped 40 hints out of 42 available.
 - - - - - - - - - - - -
Working with query 26c.sql.
```

```
Baseline time: 27.67s.
Hints used for modifications: [].
Skipped 51 hints out of 51 available.
 - - - - - - - - - - - -
Working with query 10a.sql.
Baseline time: 6.41s.
10a.sql: Hint NestLoop(cn mc) resulted in a runtime of 9.37s.
10a.sql: Hint MergeJoin(cn mc) resulted in a runtime of 6.47s.
Hints used for modifications: ['NestLoop(cn mc)', 'MergeJoin(cn mc)'].
Skipped 19 hints out of 21 available.
 - - - - - - - - - - - -
Working with query 22d.sql.
Baseline time: 16.66s.
22d.sql: Hint HashJoin(k mk) resulted in a runtime of 8.23s.
22d.sql: Hint MergeJoin(k mk) resulted in a runtime of 8.29s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 46 hints out of 48 available.
 - - - - - - - - - - - -
Working with query 2d.sql.
Baseline time: 7.45s.
2d.sql: Hint HashJoin(mk k) resulted in a runtime of 3.06s.
2d.sql: Hint MergeJoin(mk k) resulted in a runtime of 3.0s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - -
Working with query 9a.sql.
Baseline time: 1.59s.
9a.sql: Hint HashJoin(an n) resulted in a runtime of 1.77s.
9a.sql: Hint MergeJoin(an n) resulted in a runtime of 2.23s.
Hints used for modifications: ['HashJoin(an n)', 'MergeJoin(an n)'].
Skipped 25 hints out of 27 available.
 - - - - - - - - - - - -
Working with query 9d.sql.
Baseline time: 6.56s.
9d.sql: Hint NestLoop(mc cn) resulted in a runtime of 6.68s.
9d.sql: Hint MergeJoin(mc cn) resulted in a runtime of 6.58s.
9d.sql: Hint HashJoin(ci rt) resulted in a runtime of 11.48s.
9d.sql: Hint MergeJoin(ci rt) resulted in a runtime of 11.91s.
Hints used for modifications: ['NestLoop(mc cn)', 'MergeJoin(mc cn)', 'HashJoin(ci rt
    )', 'MergeJoin(ci rt)'].
Skipped 23 hints out of 27 available.
 - - - - - - - - - - - -
Working with query 15a.sql.
Baseline time: 3.29s.
15a.sql: Hint NestLoop(cn mc) resulted in a runtime of 3.29s.
```

```
15a.sql: Hint MergeJoin(cn mc) resulted in a runtime of 3.28s.
Hints used for modifications: ['NestLoop(cn mc)', 'MergeJoin(cn mc)'].
Skipped 40 hints out of 42 available.
 - - - - - - - - - - - -
Working with query 4b.sql.
Baseline time: 0.39s.
4b.sql: Hint NestLoop(it mi_idx) resulted in a runtime of 1.42s.
4b.sql: Hint MergeJoin(it mi_idx) resulted in a runtime of 1.39s.
Hints used for modifications: ['NestLoop(it mi_idx)', 'MergeJoin(it mi_idx)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - - -
Working with query 12b.sql.
Baseline time: 0.32s.
12b.sql: Hint NestLoop(mi_idx it2) resulted in a runtime of 0.16s.
12b.sql: Hint MergeJoin(mi_idx it2) resulted in a runtime of 0.99s.
Hints used for modifications: ['NestLoop(mi_idx it2)', 'MergeJoin(mi_idx it2)'].
Skipped 28 hints out of 30 available.
 - - - - - - - - - - - - -
Working with query 6a.sql.
Baseline time: 0.84s.
6a.sql: Hint HashJoin(k mk) resulted in a runtime of 0.83s.
6a.sql: Hint MergeJoin(k mk) resulted in a runtime of 1.91s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - - -
Working with query 6f.sql.
Baseline time: 54.46s.
6f.sql: Hint HashJoin(k mk) resulted in a runtime of 17.9s.
6f.sql: Hint MergeJoin(k mk) resulted in a runtime of 18.56s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - - -
Working with query 6c.sql.
Baseline time: 0.19s.
6c.sql: Hint HashJoin(k mk) resulted in a runtime of 0.9s.
6c.sql: Hint MergeJoin(k mk) resulted in a runtime of 0.92s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - - -
Working with query 12c.sql.
Baseline time: 6.78s.
12c.sql: Hint NestLoop(mi_idx it2) resulted in a runtime of 16.98s.
12c.sql: Hint MergeJoin(mi_idx it2) resulted in a runtime of 7.73s.
Hints used for modifications: ['NestLoop(mi_idx it2)', 'MergeJoin(mi_idx it2)'].
Skipped 28 hints out of 30 available.
```

```
 - - - - - - - - - - - -
Working with query 19b.sql.
Baseline time: 0.98s.
19b.sql: Hint HashJoin(t mc) resulted in a runtime of 0.93s.
19b.sql: Hint MergeJoin(t mc) resulted in a runtime of 0.88s.
Hints used for modifications: ['HashJoin(t mc)', 'MergeJoin(t mc)'].
Skipped 37 hints out of 39 available.
 - - - - - - - - - - - -
Working with query 16a.sql.
Baseline time: 6.13s.
16a.sql: Hint HashJoin(mk k) resulted in a runtime of 2.31s.
16a.sql: Hint MergeJoin(mk k) resulted in a runtime of 1.7s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 31 hints out of 33 available.
 - - - - - - - - - - - -
Working with query 13d.sql.
Baseline time: 5.91s.
13d.sql: Hint NestLoop(it miidx) resulted in a runtime of 13.19s.
13d.sql: Hint MergeJoin(it miidx) resulted in a runtime of 6.08s.
Hints used for modifications: ['NestLoop(it miidx)', 'MergeJoin(it miidx)'].
Skipped 31 hints out of 33 available.
 - - - - - - - - - - - -
Working with query 29c.sql.
Baseline time: 4.47s.
Hints used for modifications: [].
Skipped 84 hints out of 84 available.
 - - - - - - - - - - - -
Working with query 31a.sql.
Baseline time: 8.54s.
31a.sql: Hint HashJoin(k mk) resulted in a runtime of 7.16s.
31a.sql: Hint MergeJoin(k mk) resulted in a runtime of 7.21s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 58 hints out of 60 available.
 - - - - - - - - - - - -
Working with query 32a.sql.
Baseline time: 0.15s.
32a.sql: Hint HashJoin(mk k) resulted in a runtime of 0.19s.
32a.sql: Hint MergeJoin(mk k) resulted in a runtime of 0.19s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 16 hints out of 18 available.
 - - - - - - - - - - - -
Working with query 22a.sql.
Baseline time: 8.32s.
22a.sql: Hint HashJoin(k mk) resulted in a runtime of 5.2s.
22a.sql: Hint MergeJoin(k mk) resulted in a runtime of 5.2s.
```

```
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 46 hints out of 48 available.
 - - - - - - - - - - - -
Working with query 16d.sql.
Baseline time: 21.85s.
16d.sql: Hint HashJoin(mk k) resulted in a runtime of 7.16s.
16d.sql: Hint MergeJoin(mk k) resulted in a runtime of 6.7s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 31 hints out of 33 available.
 - - - - - - - - - - - -
Working with query 13a.sql.
Baseline time: 5.84s.
13a.sql: Hint NestLoop(it miidx) resulted in a runtime of 14.73s.
13a.sql: Hint MergeJoin(it miidx) resulted in a runtime of 4.41s.
Hints used for modifications: ['NestLoop(it miidx)', 'MergeJoin(it miidx)'].
Skipped 31 hints out of 33 available.
 - - - - - - - - - - - -
Working with query 15d.sql.
Baseline time: 5.75s.
15d.sql: Hint NestLoop(it1 mi) resulted in a runtime of 5.89s.
15d.sql: Hint MergeJoin(it1 mi) resulted in a runtime of 5.85s.
Hints used for modifications: ['NestLoop(it1 mi)', 'MergeJoin(it1 mi)'].
Skipped 40 hints out of 42 available.
 - - - - - - - - - - - -
Working with query 14b.sql.
Baseline time: 3.87s.
14b.sql: Hint HashJoin(k mk) resulted in a runtime of 2.51s.
14b.sql: Hint MergeJoin(k mk) resulted in a runtime of 2.5s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 28 hints out of 30 available.
 - - - - - - - - - - - -
Working with query 3b.sql.
Baseline time: 3.33s.
3b.sql: Hint HashJoin(k mk) resulted in a runtime of 1.57s.
3b.sql: Hint MergeJoin(k mk) resulted in a runtime of 1.98s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 10 hints out of 12 available.
 - - - - - - - - - - - -
Working with query 6e.sql.
Baseline time: 0.85s.
6e.sql: Hint HashJoin(k mk) resulted in a runtime of 0.82s.
6e.sql: Hint MergeJoin(k mk) resulted in a runtime of 1.95s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - -
```

```
Working with query 31c.sql.
Baseline time: 8.58s.
31c.sql: Hint HashJoin(k mk) resulted in a runtime of 7.21s.
31c.sql: Hint MergeJoin(k mk) resulted in a runtime of 7.2s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 58 hints out of 60 available.
 - - - - - - - - - - - -
Working with query 18c.sql.
Baseline time: 22.23s.
18c.sql: Hint NestLoop(it2 mi_idx) resulted in a runtime of 61.19s.
18c.sql: Hint MergeJoin(it2 mi_idx) resulted in a runtime of 22.99s.
Hints used for modifications: ['NestLoop(it2 mi_idx)', 'MergeJoin(it2 mi_idx)'].
Skipped 25 hints out of 27 available.
 - - - - - - - - - - - -
Working with query 30c.sql.
Baseline time: 39.92s.
Hints used for modifications: [].
Skipped 63 hints out of 63 available.
 - - - - - - - - - - - -
Working with query 24b.sql.
Baseline time: 0.63s.
Hints used for modifications: [].
Skipped 54 hints out of 54 available.
 - - - - - - - - - - - -
Working with query 1a.sql.
Baseline time: 0.44s.
1a.sql: Hint NestLoop(it mi_idx) resulted in a runtime of 0.39s.
1a.sql: Hint MergeJoin(it mi_idx) resulted in a runtime of 0.83s.
Hints used for modifications: ['NestLoop(it mi_idx)', 'MergeJoin(it mi_idx)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - -
Working with query 11d.sql.
Baseline time: 3.14s.
11d.sql: Hint HashJoin(mk k) resulted in a runtime of 0.26s.
11d.sql: Hint MergeJoin(mk k) resulted in a runtime of 0.26s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 28 hints out of 30 available.
 - - - - - - - - - - - -
Working with query 17d.sql.
Baseline time: 58.62s.
17d.sql: Hint HashJoin(mk k) resulted in a runtime of 19.48s.
17d.sql: Hint MergeJoin(mk k) resulted in a runtime of 20.94s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 25 hints out of 27 available.
 - - - - - - - - - - - -
```

```
Working with query 23a.sql.
Baseline time: 2.33s.
23a.sql: Hint NestLoop(cct1 cc) resulted in a runtime of 2.33s.
23a.sql: Hint MergeJoin(cct1 cc) resulted in a runtime of 2.37s.
Hints used for modifications: ['NestLoop(cct1 cc)', 'MergeJoin(cct1 cc)'].
Skipped 46 hints out of 48 available.
 - - - - - - - - - - - -
Working with query 21b.sql.
Baseline time: 1.08s.
21b.sql: Hint HashJoin(mk k) resulted in a runtime of 0.26s.
21b.sql: Hint MergeJoin(mk k) resulted in a runtime of 0.26s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 40 hints out of 42 available.
 - - - - - - - - - - - -
Working with query 11b.sql.
Baseline time: 0.99s.
11b.sql: Hint HashJoin(mk k) resulted in a runtime of 0.21s.
11b.sql: Hint MergeJoin(mk k) resulted in a runtime of 0.21s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 28 hints out of 30 available.
 - - - - - - - - - - - -
Working with query 29b.sql.
Baseline time: 1.41s.
Hints used for modifications: [].
Skipped 84 hints out of 84 available.
 - - - - - - - - - - - -
Working with query 8d.sql.
Baseline time: 3.78s.
8d.sql: Hint NestLoop(an1 n1) resulted in a runtime of 3.67s.
8d.sql: Hint MergeJoin(an1 n1) resulted in a runtime of 3.63s.
8d.sql: Hint NestLoop(mc cn) resulted in a runtime of 3.93s.
8d.sql: Hint MergeJoin(mc cn) resulted in a runtime of 3.82s.
8d.sql: Hint HashJoin(ci rt) resulted in a runtime of 9.13s.
8d.sql: Hint MergeJoin(ci rt) resulted in a runtime of 10.5s.
Hints used for modifications: ['NestLoop(an1 n1)', 'MergeJoin(an1 n1)', 'NestLoop(mc
    cn)', 'MergeJoin(mc cn)', 'HashJoin(ci rt)', 'MergeJoin(ci rt)'].
Skipped 18 hints out of 24 available.
 - - - - - - - - - - - -
Working with query 9c.sql.
Baseline time: 3.46s.
9c.sql: Hint NestLoop(an n) resulted in a runtime of 3.26s.
9c.sql: Hint MergeJoin(an n) resulted in a runtime of 5.99s.
Hints used for modifications: ['NestLoop(an n)', 'MergeJoin(an n)'].
Skipped 25 hints out of 27 available.
 - - - - - - - - - - - -
```

```
Working with query 17b.sql.
Baseline time: 59.15s.
17b.sql: Hint HashJoin(mk k) resulted in a runtime of 19.9s.
17b.sql: Hint MergeJoin(mk k) resulted in a runtime of 21.22s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 25 hints out of 27 available.
 - - - - - - - - - - - -
Working with query 30b.sql.
Baseline time: 2.78s.
Hints used for modifications: [].
Skipped 63 hints out of 63 available.
 - - - - - - - - - - - -
Working with query 26b.sql.
Baseline time: 2.72s.
Hints used for modifications: [].
Skipped 51 hints out of 51 available.
 - - - - - - - - - - - -
Working with query 2c.sql.
Baseline time: 3.07s.
2c.sql: Hint HashJoin(mk k) resulted in a runtime of 0.2s.
2c.sql: Hint MergeJoin(mk k) resulted in a runtime of 0.2s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - -
Working with query 15b.sql.
Baseline time: 0.54s.
15b.sql: Hint HashJoin(cn mc) resulted in a runtime of 0.61s.
15b.sql: Hint MergeJoin(cn mc) resulted in a runtime of 0.64s.
Hints used for modifications: ['HashJoin(cn mc)', 'MergeJoin(cn mc)'].
Skipped 40 hints out of 42 available.
 - - - - - - - - - - - -
Working with query 17f.sql.
Baseline time: 64.28s.
17f.sql: Hint HashJoin(mk k) resulted in a runtime of 22.18s.
17f.sql: Hint MergeJoin(mk k) resulted in a runtime of 22.77s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 25 hints out of 27 available.
 - - - - - - - - - - - -
Working with query 21a.sql.
Baseline time: 1.05s.
21a.sql: Hint HashJoin(mk k) resulted in a runtime of 0.28s.
21a.sql: Hint MergeJoin(mk k) resulted in a runtime of 0.31s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 40 hints out of 42 available.
 - - - - - - - - - - - -
```

```
Working with query 18a.sql.
Baseline time: 21.13s.
18a.sql: Hint NestLoop(it2 mi_idx) resulted in a runtime of 21.54s.
18a.sql: Hint MergeJoin(it2 mi_idx) resulted in a runtime of 21.2s.
Hints used for modifications: ['NestLoop(it2 mi_idx)', 'MergeJoin(it2 mi_idx)'].
Skipped 25 hints out of 27 available.
 - - - - - - - - - - - -
Working with query 24a.sql.
Baseline time: 7.36s.
Hints used for modifications: [].
Skipped 54 hints out of 54 available.
 - - - - - - - - - - - -
Working with query 8c.sql.
Baseline time: 4.75s.
8c.sql: Hint NestLoop(a1 n1) resulted in a runtime of 4.69s.
8c.sql: Hint MergeJoin(a1 n1) resulted in a runtime of 4.65s.
8c.sql: Hint NestLoop(mc cn) resulted in a runtime of 4.98s.
8c.sql: Hint MergeJoin(mc cn) resulted in a runtime of 4.89s.
8c.sql: Hint HashJoin(ci rt) resulted in a runtime of 8.93s.
8c.sql: Hint MergeJoin(ci rt) resulted in a runtime of 11.03s.
Hints used for modifications: ['NestLoop(a1 n1)', 'MergeJoin(a1 n1)', 'NestLoop(mc cn
    )', 'MergeJoin(mc cn)', 'HashJoin(ci rt)', 'MergeJoin(ci rt)'].
Skipped 18 hints out of 24 available.
 - - - - - - - - - - - -
Working with query 27a.sql.
Baseline time: 0.4s.
Hints used for modifications: [].
Skipped 63 hints out of 63 available.
 - - - - - - - - - - - -
Working with query 5c.sql.
Baseline time: 1.09s.
5c.sql: Hint NestLoop(ct mc) resulted in a runtime of 1.13s.
5c.sql: Hint MergeJoin(ct mc) resulted in a runtime of 1.1s.
Hints used for modifications: ['NestLoop(ct mc)', 'MergeJoin(ct mc)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - -
Working with query 28b.sql.
Baseline time: 6.96s.
Hints used for modifications: [].
Skipped 69 hints out of 69 available.
 - - - - - - - - - - - -
Working with query 3c.sql.
Baseline time: 7.87s.
3c.sql: Hint HashJoin(k mk) resulted in a runtime of 3.52s.
3c.sql: Hint MergeJoin(k mk) resulted in a runtime of 4.09s.
```

```
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 10 hints out of 12 available.
 - - - - - - - - - - - -
Working with query 31b.sql.
Baseline time: 2.85s.
31b.sql: Hint HashJoin(k mk) resulted in a runtime of 0.5s.
31b.sql: Hint MergeJoin(k mk) resulted in a runtime of 0.5s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 58 hints out of 60 available.
 - - - - - - - - - - - -
Working with query 29a.sql.
Baseline time: 0.79s.
Hints used for modifications: [].
Skipped 84 hints out of 84 available.
 - - - - - - - - - - - -
Working with query 6d.sql.
Baseline time: 52.98s.
6d.sql: Hint HashJoin(k mk) resulted in a runtime of 18.05s.
6d.sql: Hint MergeJoin(k mk) resulted in a runtime of 17.51s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - -
Working with query 20b.sql.
Baseline time: 31.82s.
20b.sql: Hint HashJoin(k mk) resulted in a runtime of 34.61s.
20b.sql: Hint MergeJoin(k mk) resulted in a runtime of 34.52s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 34 hints out of 36 available.
 - - - - - - - - - - - -
Working with query 17c.sql.
Baseline time: 56.47s.
17c.sql: Hint HashJoin(mk k) resulted in a runtime of 19.11s.
17c.sql: Hint MergeJoin(mk k) resulted in a runtime of 20.69s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 25 hints out of 27 available.
 - - - - - - - - - - - -
Working with query 28a.sql.
Baseline time: 5.85s.
Hints used for modifications: [].
Skipped 69 hints out of 69 available.
 - - - - - - - - - - - -
Working with query 26a.sql.
Baseline time: 20.88s.
Hints used for modifications: [].
Skipped 51 hints out of 51 available.
```

```
 - - - - - - - - - - - -
Working with query 11c.sql.
Baseline time: 3.12s.
11c.sql: Hint HashJoin(mk k) resulted in a runtime of 0.97s.
11c.sql: Hint MergeJoin(mk k) resulted in a runtime of 1.01s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 28 hints out of 30 available.
 - - - - - - - - - - - -
Working with query 4a.sql.
Baseline time: 2.52s.
4a.sql: Hint HashJoin(k mk) resulted in a runtime of 3.43s.
4a.sql: Hint MergeJoin(k mk) resulted in a runtime of 3.44s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - -
Working with query 13c.sql.
Baseline time: 2.62s.
13c.sql: Hint NestLoop(it miidx) resulted in a runtime of 7.57s.
13c.sql: Hint MergeJoin(it miidx) resulted in a runtime of 1.92s.
Hints used for modifications: ['NestLoop(it miidx)', 'MergeJoin(it miidx)'].
Skipped 31 hints out of 33 available.
 - - - - - - - - - - - -
Working with query 22b.sql.
Baseline time: 7.31s.
22b.sql: Hint HashJoin(k mk) resulted in a runtime of 3.37s.
22b.sql: Hint MergeJoin(k mk) resulted in a runtime of 3.38s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 46 hints out of 48 available.
 - - - - - - - - - - - -
Working with query 19a.sql.
Baseline time: 1.59s.
19a.sql: Hint HashJoin(n an) resulted in a runtime of 2.01s.
19a.sql: Hint MergeJoin(n an) resulted in a runtime of 2.39s.
Hints used for modifications: ['HashJoin(n an)', 'MergeJoin(n an)'].
Skipped 37 hints out of 39 available.
 - - - - - - - - - - - -
Working with query 2a.sql.
Baseline time: 4.29s.
2a.sql: Hint HashJoin(mk k) resulted in a runtime of 1.92s.
2a.sql: Hint MergeJoin(mk k) resulted in a runtime of 1.59s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - -
Working with query 7a.sql.
Baseline time: 21.56s.
```

```
7a.sql: Hint HashJoin(lt ml) resulted in a runtime of 21.44s.
7a.sql: Hint MergeJoin(lt ml) resulted in a runtime of 21.45s.
Hints used for modifications: ['HashJoin(lt ml)', 'MergeJoin(lt ml)'].
Skipped 31 hints out of 33 available.
 - - - - - - - - - - - -
Working with query 18b.sql.
Baseline time: 3.78s.
18b.sql: Hint NestLoop(it2 mi_idx) resulted in a runtime of 3.78s.
18b.sql: Hint MergeJoin(it2 mi_idx) resulted in a runtime of 5.25s.
Hints used for modifications: ['NestLoop(it2 mi_idx)', 'MergeJoin(it2 mi_idx)'].
Skipped 25 hints out of 27 available.
 - - - - - - - - - - - -
Working with query 17a.sql.
Baseline time: 64.87s.
17a.sql: Hint HashJoin(mk k) resulted in a runtime of 21.13s.
17a.sql: Hint MergeJoin(mk k) resulted in a runtime of 20.94s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 25 hints out of 27 available.
 - - - - - - - - - - - -
Working with query 4c.sql.
Baseline time: 2.3s.
4c.sql: Hint HashJoin(k mk) resulted in a runtime of 3.48s.
4c.sql: Hint MergeJoin(k mk) resulted in a runtime of 3.44s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - -
Working with query 19d.sql.
Baseline time: 13.73s.
19d.sql: Hint HashJoin(rt ci) resulted in a runtime of 19.04s.
19d.sql: Hint MergeJoin(rt ci) resulted in a runtime of 17.68s.
Hints used for modifications: ['HashJoin(rt ci)', 'MergeJoin(rt ci)'].
Skipped 37 hints out of 39 available.
 - - - - - - - - - - - -
Working with query 8b.sql.
Baseline time: 1.21s.
8b.sql: Hint HashJoin(mc cn) resulted in a runtime of 1.14s.
8b.sql: Hint MergeJoin(mc cn) resulted in a runtime of 1.16s.
Hints used for modifications: ['HashJoin(mc cn)', 'MergeJoin(mc cn)'].
Skipped 22 hints out of 24 available.
 - - - - - - - - - - - -
Working with query 33b.sql.
Baseline time: 0.55s.
Hints used for modifications: [].
Skipped 57 hints out of 57 available.
 - - - - - - - - - - - -
```

```
Working with query 14c.sql.
Baseline time: 15.63s.
14c.sql: Hint HashJoin(k mk) resulted in a runtime of 7.67s.
14c.sql: Hint MergeJoin(k mk) resulted in a runtime of 7.85s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 28 hints out of 30 available.
 - - - - - - - - - - - -
Working with query 33c.sql.
Baseline time: 0.47s.
Hints used for modifications: [].
Skipped 57 hints out of 57 available.
 - - - - - - - - - - - -
Working with query 7c.sql.
Baseline time: 17.63s.
7c.sql: Hint HashJoin(it pi) resulted in a runtime of 5.91s.
7c.sql: Hint MergeJoin(it pi) resulted in a runtime of 6.15s.
Hints used for modifications: ['HashJoin(it pi)', 'MergeJoin(it pi)'].
Skipped 31 hints out of 33 available.
 - - - - - - - - - - - -
Working with query 23b.sql.
Baseline time: 1.26s.
23b.sql: Hint HashJoin(k mk) resulted in a runtime of 1.0s.
23b.sql: Hint MergeJoin(k mk) resulted in a runtime of 2.39s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 46 hints out of 48 available.
 - - - - - - - - - - - -
Working with query 10b.sql.
Baseline time: 2.87s.
10b.sql: Hint NestLoop(cn mc) resulted in a runtime of 4.41s.
10b.sql: Hint MergeJoin(cn mc) resulted in a runtime of 2.93s.
Hints used for modifications: ['NestLoop(cn mc)', 'MergeJoin(cn mc)'].
Skipped 19 hints out of 21 available.
 - - - - - - - - - - - -
Working with query 20a.sql.
Baseline time: 0.15s.
Hints used for modifications: [].
Skipped 36 hints out of 36 available.
 - - - - - - - - - - - -
Working with query 25a.sql.
Baseline time: 35.75s.
25a.sql: Hint HashJoin(k mk) resulted in a runtime of 32.64s.
25a.sql: Hint MergeJoin(k mk) resulted in a runtime of 32.06s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 40 hints out of 42 available.
 - - - - - - - - - - - -
```

```
Working with query 30a.sql.
Baseline time: 19.03s.
Hints used for modifications: [].
Skipped 63 hints out of 63 available.
 - - - - - - - - - - - -
Working with query 5b.sql.
Baseline time: 0.55s.
5b.sql: Hint NestLoop(ct mc) resulted in a runtime of 0.53s.
5b.sql: Hint MergeJoin(ct mc) resulted in a runtime of 0.54s.
Hints used for modifications: ['NestLoop(ct mc)', 'MergeJoin(ct mc)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - -
Working with query 9b.sql.
Baseline time: 1.56s.
9b.sql: Hint HashJoin(an n) resulted in a runtime of 15.69s.
9b.sql: Hint MergeJoin(an n) resulted in a runtime of 15.76s.
Hints used for modifications: ['HashJoin(an n)', 'MergeJoin(an n)'].
Skipped 25 hints out of 27 available.
 - - - - - - - - - - - -
Working with query 16b.sql.
Baseline time: 72.48s.
16b.sql: Hint HashJoin(mk k) resulted in a runtime of 23.82s.
16b.sql: Hint MergeJoin(mk k) resulted in a runtime of 23.76s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 31 hints out of 33 available.
 - - - - - - - - - - - -
Working with query 2b.sql.
Baseline time: 3.27s.
2b.sql: Hint HashJoin(mk k) resulted in a runtime of 1.8s.
2b.sql: Hint MergeJoin(mk k) resulted in a runtime of 1.9s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - -
Working with query 28c.sql.
Baseline time: 7.71s.
Hints used for modifications: [].
Skipped 69 hints out of 69 available.
 - - - - - - - - - - - -
Working with query 5a.sql.
Baseline time: 0.55s.
5a.sql: Hint NestLoop(ct mc) resulted in a runtime of 0.55s.
5a.sql: Hint MergeJoin(ct mc) resulted in a runtime of 0.54s.
Hints used for modifications: ['NestLoop(ct mc)', 'MergeJoin(ct mc)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - -
```

```
Working with query 27c.sql.
Baseline time: 0.31s.
Hints used for modifications: [].
Skipped 63 hints out of 63 available.
 - - - - - - - - - - - -
Working with query 23c.sql.
Baseline time: 4.4s.
23c.sql: Hint NestLoop(cct1 cc) resulted in a runtime of 4.38s.
23c.sql: Hint MergeJoin(cct1 cc) resulted in a runtime of 4.53s.
Hints used for modifications: ['NestLoop(cct1 cc)', 'MergeJoin(cct1 cc)'].
Skipped 46 hints out of 48 available.
 - - - - - - - - - - - -
Working with query 11a.sql.
Baseline time: 1.0s.
11a.sql: Hint HashJoin(mk k) resulted in a runtime of 0.24s.
11a.sql: Hint MergeJoin(mk k) resulted in a runtime of 0.25s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 28 hints out of 30 available.
 - - - - - - - - - - - -
Working with query 12a.sql.
Baseline time: 2.6s.
12a.sql: Hint NestLoop(mi_idx it2) resulted in a runtime of 2.62s.
12a.sql: Hint MergeJoin(mi_idx it2) resulted in a runtime of 2.81s.
Hints used for modifications: ['NestLoop(mi_idx it2)', 'MergeJoin(mi_idx it2)'].
Skipped 28 hints out of 30 available.
 - - - - - - - - - - - -
Working with query 25b.sql.
Baseline time: 2.57s.
25b.sql: Hint HashJoin(k mk) resulted in a runtime of 2.6s.
25b.sql: Hint MergeJoin(k mk) resulted in a runtime of 2.59s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 40 hints out of 42 available.
 - - - - - - - - - - - -
Working with query 27b.sql.
Baseline time: 0.25s.
Hints used for modifications: [].
Skipped 63 hints out of 63 available.
 - - - - - - - - - - - -
Working with query 22c.sql.
Baseline time: 14.81s.
22c.sql: Hint HashJoin(k mk) resulted in a runtime of 9.0s.
22c.sql: Hint MergeJoin(k mk) resulted in a runtime of 8.95s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 46 hints out of 48 available.
 - - - - - - - - - - - -
```

```
Working with query 10c.sql.
Baseline time: 8.19s.
10c.sql: Hint NestLoop(chn ci) resulted in a runtime of 8.17s.
10c.sql: Hint MergeJoin(chn ci) resulted in a runtime of 8.84s.
10c.sql: Hint NestLoop(cn mc) resulted in a runtime of 8.18s.
10c.sql: Hint MergeJoin(cn mc) resulted in a runtime of 8.09s.
Hints used for modifications: ['NestLoop(chn ci)', 'MergeJoin(chn ci)', 'NestLoop(cn
    mc)', 'MergeJoin(cn mc)'].
Skipped 17 hints out of 21 available.
 - - - - - - - - - - - -
Working with query 20c.sql.
Baseline time: 29.16s.
20c.sql: Hint HashJoin(k mk) resulted in a runtime of 32.88s.
20c.sql: Hint MergeJoin(k mk) resulted in a runtime of 32.69s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 34 hints out of 36 available.
 - - - - - - - - - - - -
Working with query 1b.sql.
Baseline time: 0.31s.
1b.sql: Hint NestLoop(it mi_idx) resulted in a runtime of 0.14s.
1b.sql: Hint MergeJoin(it mi_idx) resulted in a runtime of 0.37s.
Hints used for modifications: ['NestLoop(it mi_idx)', 'MergeJoin(it mi_idx)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - -
Working with query 17e.sql.
Baseline time: 64.36s.
17e.sql: Hint HashJoin(mk k) resulted in a runtime of 21.16s.
17e.sql: Hint MergeJoin(mk k) resulted in a runtime of 20.96s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 25 hints out of 27 available.
 - - - - - - - - - - - -
Working with query 19c.sql.
Baseline time: 8.33s.
19c.sql: Hint HashJoin(rt ci) resulted in a runtime of 23.56s.
19c.sql: Hint MergeJoin(rt ci) resulted in a runtime of 23.34s.
Hints used for modifications: ['HashJoin(rt ci)', 'MergeJoin(rt ci)'].
Skipped 37 hints out of 39 available.
 - - - - - - - - - - - -
Working with query 7b.sql.
Baseline time: 8.09s.
7b.sql: Hint HashJoin(lt ml) resulted in a runtime of 9.35s.
7b.sql: Hint MergeJoin(lt ml) resulted in a runtime of 9.22s.
Hints used for modifications: ['HashJoin(lt ml)', 'MergeJoin(lt ml)'].
Skipped 31 hints out of 33 available.
 - - - - - - - - - - - -
```

```
Working with query 1d.sql.
Baseline time: 0.31s.
1d.sql: Hint NestLoop(it mi_idx) resulted in a runtime of 0.15s.
1d.sql: Hint MergeJoin(it mi_idx) resulted in a runtime of 0.37s.
Hints used for modifications: ['NestLoop(it mi_idx)', 'MergeJoin(it mi_idx)'].
Skipped 13 hints out of 15 available.
 - - - - - - - - - - - -
Working with query 25c.sql.
Baseline time: 52.76s.
25c.sql: Hint HashJoin(k mk) resulted in a runtime of 49.2s.
25c.sql: Hint MergeJoin(k mk) resulted in a runtime of 49.19s.
Hints used for modifications: ['HashJoin(k mk)', 'MergeJoin(k mk)'].
Skipped 40 hints out of 42 available.
 - - - - - - - - - - - -
Working with query 16c.sql.
Baseline time: 26.07s.
16c.sql: Hint HashJoin(mk k) resulted in a runtime of 8.16s.
16c.sql: Hint MergeJoin(mk k) resulted in a runtime of 7.7s.
Hints used for modifications: ['HashJoin(mk k)', 'MergeJoin(mk k)'].
Skipped 31 hints out of 33 available.
```

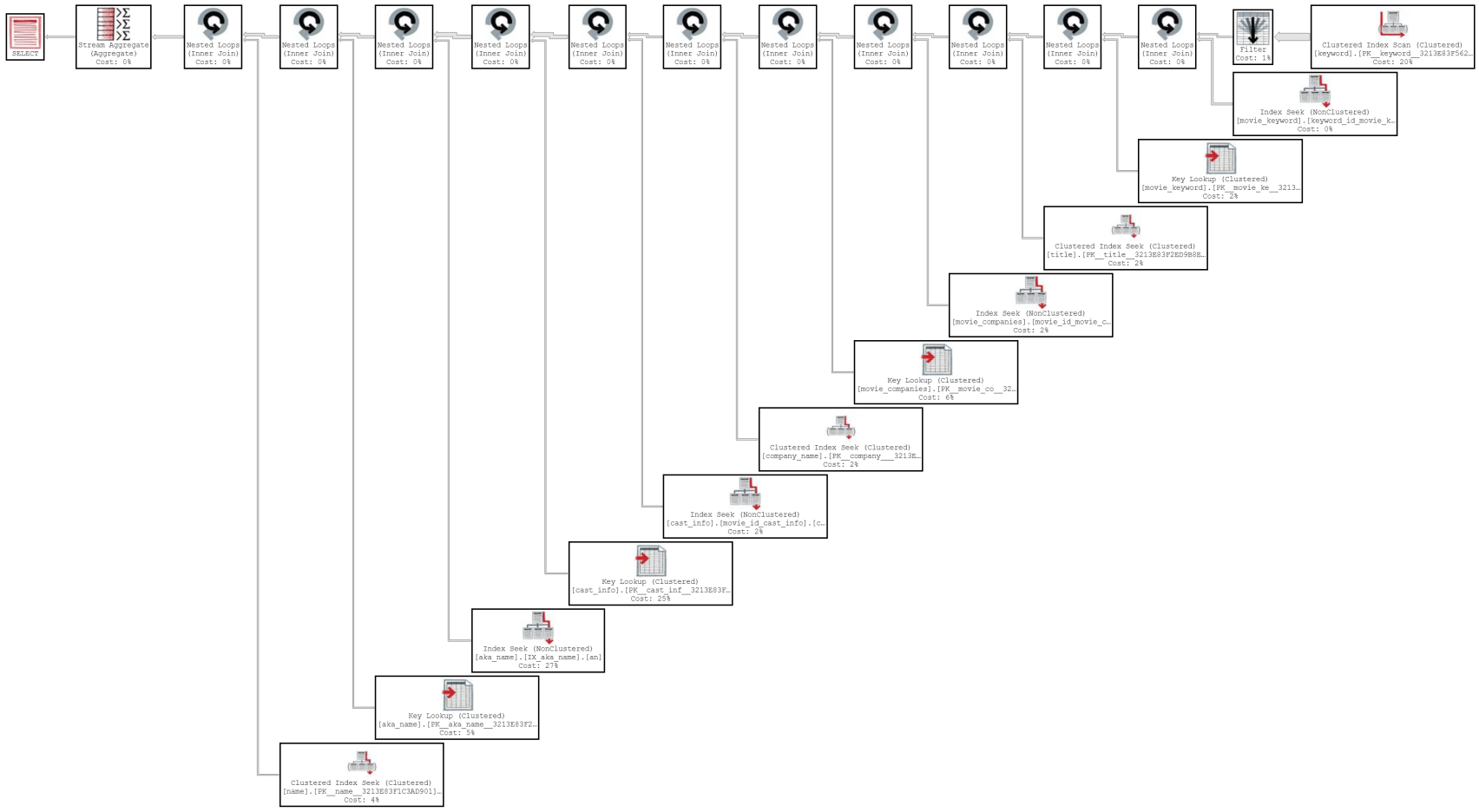

Figure B-1: Default SQL Server query plan for JOB Query 16b. Every operator for joins uses the nested loop algorithm.

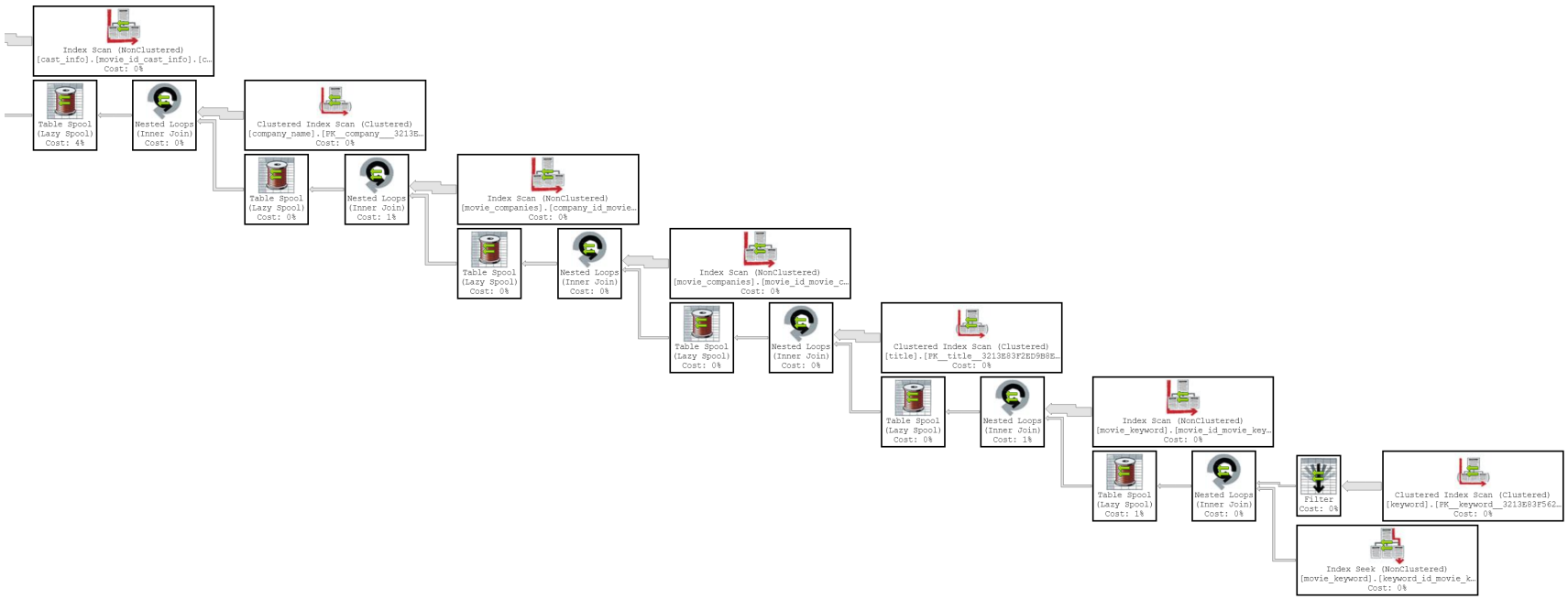

Figure B-2: Explicitly specified nested loop join plan for 16b. 16b was rewritten in join-isolated, nested loop join operator form for this query plan.

Appendix C

Performance Comparison Charts

Table C.1: Comparison of execution times for JOB queries that took more than 25s and their maximum speed-ups with our deep Bayesian neural network. There are 18 queries shown here.

	Postgres	Neural network
6d	55.75	8.10
6f	56.06	8.63
16b	70.41	10.27
16c	25.32	9.32
17a	62.60	8.91
17b	59.93	9.17
17c	57.48	8.86
17d	58.75	8.44
17e	62.07	8.89
17f	65.01	8.83
20a	44.14	0.21
20b	33.87	9.59

Continued on next page

Table C.1: Comparison of execution times for JOB queries that took more than 25s and their maximum speed-ups with our deep Bayesian neural network. There are 18 queries shown here.

	Postgres	Neural network
20c	28.88	9.44
25a	34.96	12.32
25c	51.58	11.90
26a	34.49	9.81
26c	36.02	9.67
30c	51.37	12.09

Bibliography

[1] Wei-Lun Chao. Machine learning tutorial. 2012.

[2] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[3] Fellowship.AI. Spacebandits: Deep contextual bandits models made simple. `https://github.com/fellowship/space-bandits`.

[4] PostgreSQL Global Development Group. Postgresql: Documentation - query planning. https://www.postgresql.org/docs/9.2/runtime-config-query.html. Accessed: December 15, 2020.

[5] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[6] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.*, 9(3):204–215, November 2015.

[7] Tyler Lu, David Pal, and Martin Pal. Contextual multi-armed bandits. volume 9 of *Proceedings of Machine Learning Research*, pages 485–492, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. JMLR Workshop and Conference Proceedings.

[8] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Learning to steer query optimizers, 2020.

[9] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo. *Proceedings of the VLDB Endowment*, 12(11):1705–1718, Jul 2019.

[10] Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. *CoRR*, abs/1803.00055, 2018.

[11] Nippon Telegraph and Telephone Corporation. `pg_hint_plan`. `https://pghintplan.osdn.jp/pg_hint_plan.html`.