Query-Aware Data Systems Tuning via Machine Learning

Connor Henderson

A thesis submitted to the
Faculty of Graduate Studies in partial
fulfillment of the requirements for the degree of

Master of Science

in

Graduate Program in Computer Science

York University
Toronto, Ontario
June 2023

© Connor Henderson, 2023

Abstract

Modern data systems such as IBM Db2 have hundreds of system configuration parameters, “knobs”, which heavily influence the performance of business queries. Manual configuration, “tuning,” by experts is painstaking and time consuming. We propose a query-informed tuning system called BLUTune which uses machine learning (ML)—deep reinforcement learning based on advantage actor-critic neural networks—to tune configurations within defined resource constraints. We translate high-dimensional *query execution plans* (QEPs) into a low-dimensional embedding space (QEP2Vec) and illustrate the usefulness of query embeddings for the downstream task of data systems tuning. To scale to complex and large workloads, we bootstrap the training process through transfer learning. We first train our model based on the estimated cost of queries; we then fine-tune it using actual query execution times. As part of a demonstration plan, we present an experimental study over various synthetic and real-world workloads with two models. The first has been trained on TPC-DS queries such that there are tables from the TPC-DS schema that are not seen during training time. The second is trained under specific resource constraints to show how the model performs when we limit the amount of memory the system has access to. Lastly we include a QTune model to highlight the robustness of our system.

Dedication

I dedicate this thesis to my parents whose love and support throughout my academic journey has been a constant source of encouragement. Their belief in me and the sacrifices they made to make my dreams a reality have filled me with gratitude.

I would also like to express my appreciation to my supervisor, Dr. Jarek Szlichta, for his guidance, mentorship, and passion for learning, which have been instrumental in shaping my academic growth. My time spent as his student has instilled in me a love for learning.

I am thankful to my York University and IBM colleagues, Associate Professor Dr. Parke Godfrey, Vincent Corvinelli, Calisto Zuzarte, and Piotr Mierzejewski, for their valuable knowledge and expertise in the field, and for their collaborative spirit, which helped me overcome many challenges.

Last but not least, I want to thank my fiancée Angela for her unwavering emotional support and belief in my abilities. Her patience and encouragement have been a source of strength during the highs and lows of this challenging journey.

I could not have accomplished this without all of you, and I am deeply grateful for your support and contributions to this work.

Acknowledgements

I would like to acknowledge Spencer Bryson of Ontario Tech University for his contributions towards the `BLUTune` system. We co-designed the reinforcement learning module described in Section 3.1; Spencer provided the implementation and I did the experimental evaluation. The subsequent query featurization framework outlined in Section 3.2 and beyond including the system demonstration were developed on my own.

Contents

Abstract **ii**

Dedication **iii**

Acknowledgements **iv**

Table of Contents **viii**

List of Figures **viii**

List of Abbreviations and Symbols **ix**

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 4

2 Background **9**

2.1 A Real-world Query Example 9

2.2 SQL, Db2, and Query Optimization 13

2.2.1 Query Optimization Steps 13

2.2.2 IBM Db2 Knobs 16

2.3 Query Embeddings 20

3 System Overview **25**

3.1 Deep Reinforcement Learning 27

3.2 Query Embedding 33

4 System Demonstration **41**

4.1 Demonstration of Knob Tuning Interface 41

4.2 Query-informed Tuning and Transfer Learning 43

4.3 IBM Tools and Resource Constraints 45

5 Experimental Evaluation **48**

5.1 TPC-DS Benchmark 48

5.1.1 TPC-H Benchmark 49

5.2 Query-informed Tuning 51

5.3 Transfer Learning 53

5.4 Effectiveness 54

6 Related Work **62**

7 Conclusion **65**

7.1 Conclusion 65

7.2 Future Work 66

List of Figures

1.1 TPC-DS Query #50. 3
1.2 TPC-DS Query #51. 4
1.3 TPC-DS Query #98 at Optimization level 1 5
1.4 TPC-DS Query #98 at Optimization level 5 6

2.1 Query Compilation Process 12
2.2 Original vs. Optimized Statement Text for TPC-DS Query 14
2.3 Query Explain Plan provided by IBM Db2's exfmt tool 17
2.4 Query Explain Plan Visualization provided by BLUTune 18
2.5 Example QEP Node 19
2.6 An Example Recommendation Encountered During Training 20
2.7 Featurization Strategy Employed by QTune 21
2.8 Example feature vector obtained via the representative strategy . . . 23

3.1 BLUTune architecture. 26
3.2 Actor network. 28
3.3 Critic network. 29
3.4 SQL query representation. 33

3.5 QEP2Vec embeddings. 34
3.6 Query #8 from TPC-DS Benchmark 38
3.7 Query #8 Template at Node 12 39
3.8 TPC-DS Query 98 with two different knob settings for sort heap and bufferpool size 39
3.9 TPC-DS Query 43 with two different knob settings for both sort heap and bufferpool size. 40

4.1 BLUTune Training Page. 42
4.2 BLUTune Tuning Page. 42
4.3 Positions of similar queries #2 (blue) and #23 (yellow) in embedding space 44
4.4 Query #2 from Similar Query Pair 46
4.5 Query #23 from Similar Query Pair 47

5.1 TPC-DS Fact Table from the TPC-DS Specification 50
5.2 TPC-H Schema Design from TPC-H Specification 57
5.3 Query level tuning 58
5.4 Workload level tuning 58
5.5 Cluster level tuning. 59
5.6 Query level tuning detailed on TPC-DS. 59
5.7 Fine-tuning performance. 60
5.8 Scalability study. 60
5.9 IBM tools & QTune-award. 61
5.10 Resource constraint for memory allocation. 61

List of Abbreviations and Symbols

A2C: .... Advantage Actor Critic

DBA: .... Database Administrator

DBMS: .... Database Management System

DBSCAN: .... Density Based Spatial Clustering of Applications with Noise

DRL: .... Deep Reinforcement Learning

FETCH: .... Index Scan

HSJOIN: .... Hash Join

IBM: .... International Business Machines Corporation

IXSCAN: .... Index Scan

LOLEPOP: .... LOw LEvel Plan OPerators

ML: .... Machine Learning

MSJOIN: .... Merge Join

NLJOIN: .... Nested Loop Join

OLAP: .... Online Analytical Processing

OLTP: .... Online Transaction Processing

PV-DBOW: .... Paragraph Vector - Distributed Bag Of Words

QEP: .... Query Execution Plan

QGM: .... Query Graph Model

SGD: .... Stochastic Gradient Descent

SQL: .... Structured Query Language

TBSCAN: .... Table Scan

Chapter 1

Introduction

1.1 Motivation

Analytical data systems such as `IBM Db2` have nowadays hundreds of "knobs" for configuration. Tuning the configuration can have an extreme impact on the performance of business queries [1], [2]. These knobs span system allocation of physical resources to modalities of the query optimizer which dictate the decisions it makes. In `IBM Db2`, such tuning knobs are found as *configuration parameters* [3] and as *registry variables* [4]. Knobs set the amount of working memory made available, such as the number of pages allocated to the bufferpool and sortheap, the degree of parallelism to be used, and even toggle specific functionalities of the query compiler, dialing down optimization features by setting an optimization level. As these tuning parameters determine the resources available to data system, they must be carefully tuned to achieve best performance [5].

Traditionally, so-called knob tuning is performed by a system administrator, or by an expert from the vendor themselves. Such manual knob tuning is, however,

a painstaking and time-consuming process, which often involves heuristic judgment, and lots of trial-and-error testing [6]. Three main challenges make knob tuning particularly difficult: (a) the sheer number of tuning parameters; (b) interdependence of the parameters; and (c) how the tuning parameters affect different workloads.

As there are many tuning parameters, it is unrealistic to expect even experts to understand the potential performance impact of each on the data system. Experts focus rather on high-profile, well understood configuration parameters, such as working memory, sortheap size, and bufferpool size. These often have rule-of-thumb guidelines associated with them, which guide the experts to tune for a given workload. For example, "the bufferpool should be ten times the size of the sortheap". These are only heuristics, however; there are likely better configurations for any given workload.

Understanding how any given configuration parameter behaves in isolation is itself a significant challenge. Understanding how they interact with each other is that much harder. Various knobs are interdependent, meaning that changing one affects the benefits of others [3]. This is not surprising when configuration parameters draw from the same physical resource; sortheap and bufferpool both are main memory allocations. This is also not unexpected when different settings can lead to the optimizer choosing different query plans for the queries. In `IBM Db2`, for example, the ratio of the size of the sortheap to the bufferpool leads the query optimizer to prioritize differently possible *query execution plans* (QEPs). This is illustrated in Figures 1.1 and 1.2 which plot for two queries from the TPC-DS benchmark [7] different resulting QEPs when varying the sizes of the sortheap and bufferpool. (The notation $1e6$ denotes 1×10^6.) These allocations affect the queries' plans' estimated costs non-uniformly, thus lead to different plans being chosen. The colors denote different QEPs. Another example

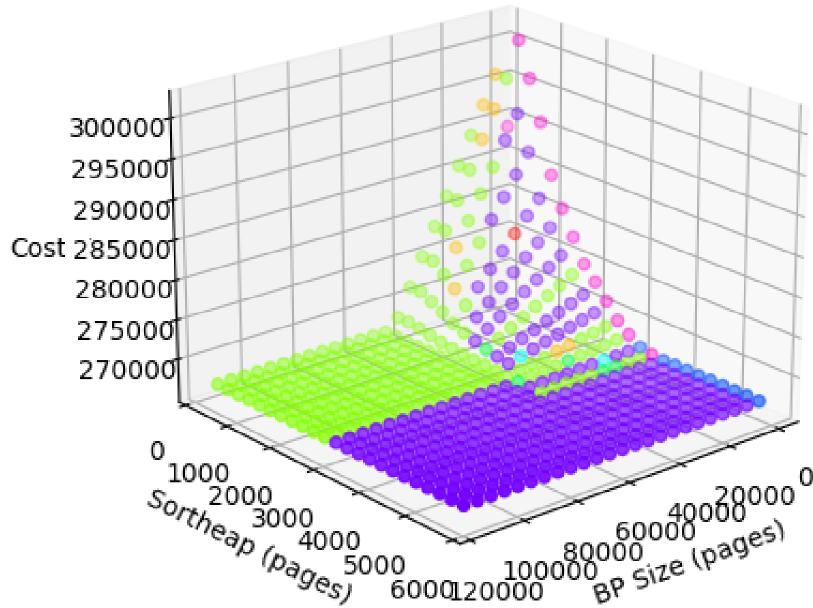


Figure 1.1: TPC-DS Query #50.

is the joint tuning of the degree of parallelism and sortheap size. As more system operations are performed in parallel, more sort consumers are allocated memory in the sortheap, eventually leading to resource over-allocation.

Another challenge is that there is no one best configuration that offers optimal performance for any query or workload. Tuning for performance is also dependent on the queries and workloads overall to be evaluated. This is demonstrated once again in Figures 1.1 and 1.2. Varying the sortheap and bufferpool sizes for the two queries from the TPC-DS have rather different impacts on their estimated costs. This shows that even queries from the same workload benefit differently from various tunings. This becomes even more pronounced when considering different and dynamic workloads.

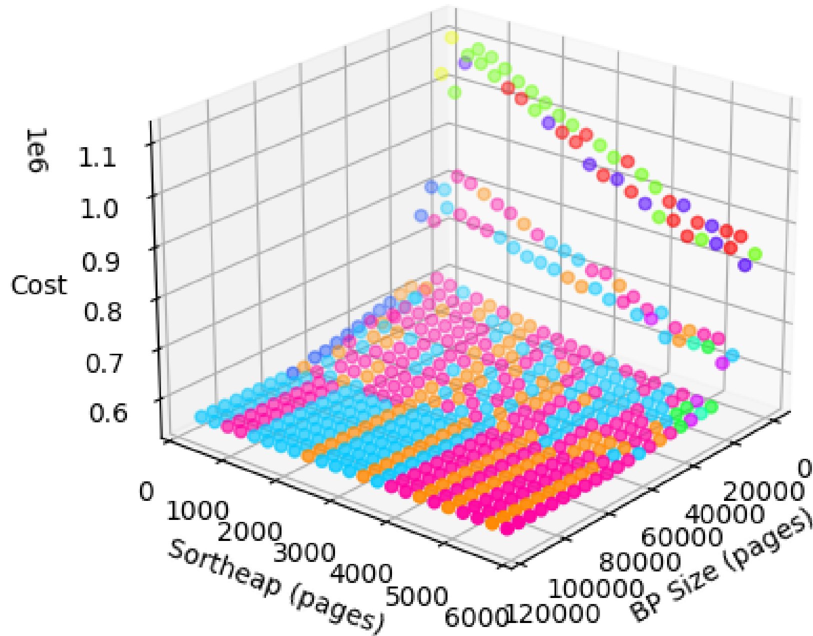


Figure 1.2: TPC-DS Query #51.

1.2 Contributions

We have designed and developed an automatic tuning system called BLUTune for IBM Db2. BLUTune is used by the IBM performance team to avoid the human burden of manual tuning by automatically recommending knob settings. The system has resulted in two publications; one for the 39th IEEE International Conference on Data Engineering (ICDE) [8] and one for the 31st ACM International Conference on Information & Knowledge Management (CIKM) [9] which resulted in a Best Paper Honourable Mention award. In Sec. 3, we provide a system overview. Our goal with BLUTune is to improve performance of business workloads. We make the following contributions.

1. **An Automatic Knob-Tuning System.**

   We present our automatic tuning system called BLUTune which uses deep rein-

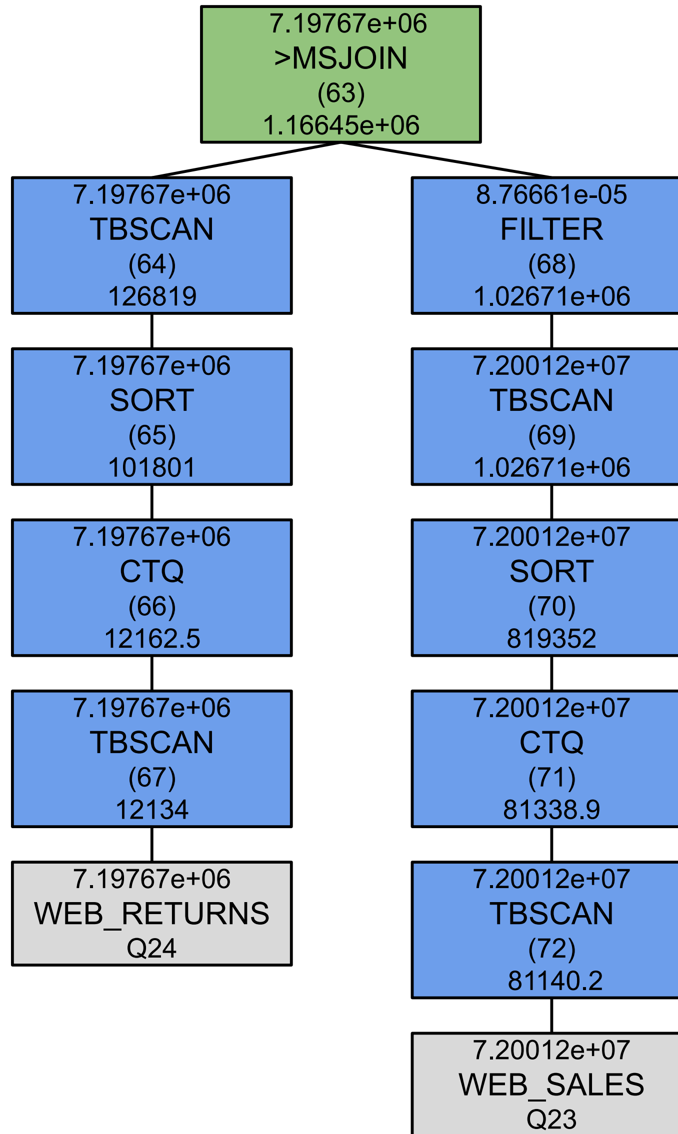


Figure 1.3: TPC-DS Query #98 at Optimization level 1

forcement learning (DRL)—specifically, the advantage actor-critic (`A2C`) agent approach—to produce effective knob recommendations. `A2C` captures how good an action (knob recommendation) is when compared against other actions, and decreases the variance in comparison to the regular actor-critic model used by

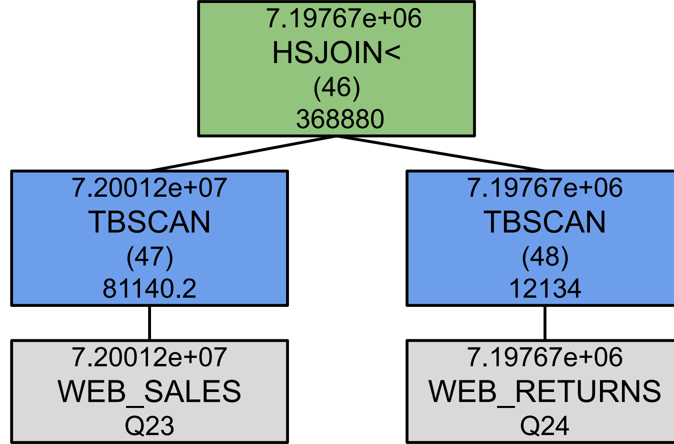


Figure 1.4: TPC-DS Query #98 at Optimization level 5

QTune and by CDBTune [10] [11]. Our approach additionally differs from these previous approaches in that our reward function as given to the agent includes the performance history, allowing us to converge more quickly.

- Since during training an ML agent must try thousands of knob configurations, this can be prohibitively expensive for complex and large workloads. To counter this, we employ a multi-stage process: first, the ML agent trains using the optimizer's cost estimates of the queries to bootstrap quickly the model; second, transfer learning is used to *fine-tune* the model based on actual query-execution times since estimates can also be inaccurate due to incorrect cardinality estimation by the optimizer[12]–[15]. DRL is used as a proof of concept, but any existing ML-tuning method that tunes based on the execution time (e.g., [10], [16], [17]) could be combined with our multi-stage process.
- We design a single model capable of tuning both continuous knobs (such as bufferpool and sortheap) and discrete knobs (such as optimization level and parallelism degree) simultaneously for query, workload, and cluster levels. These

recommendations are kept within user-defined resource constraints (e.g., limited memory for cloud computing).

2. **Query Representation Learning.**

   We devise `QEP2Vec`, an unsupervised neural-network framework to learn representations of QEPs that map their high dimensionality into a low-dimensional embedding space for use in the downstream ML task of knobs tuning. [18]. We follow the given success of recently proposed neural document embeddings such as `Word2Vec` and `Doc2Vec`, to view an entire QEP as a document, and decomposed templated query subplans as words. First, a set of subgraphs is extracted from each input graph. These subgraphs can be chosen based on various criteria, such as the presence of certain nodes (aggregate operators or expensive sorts), or based on structural properties of the graph like joins. Second, a neural network is trained to predict the context subgraphs given a target subgraph, using a skip-gram-like objective function. In contrast, the prior query-aware `QTune` system [17] uses a simple featurization of QEPs.

3. **System Demonstration.**

   We highlight three models in this work. The first model is trained on TPC-DS queries and tested on tables that were not part of the training set, showcasing the effectiveness of our table agnostic method. The second model is trained under memory constraints to evaluate its performance under limited resources. Finally, we demonstrate the robustness of our system by including a QTune model trained on one benchmark schema and evaluated on a different benchmark. Each of these models can be interactively used in a web-based application that is deployed through Docker.

4. **Experimental Evaluation.**

   We present a comprehensive experimental study to demonstrate the effectiveness of our approaches within, and the scalability of, `BLUTune` over complex and large workloads. Our experimental results illustrate that our system can learn knob configurations better than that other existing methods and `IBM` tools.

The necessary data systems background for understanding what `BLUTune` interacts with is outlined in Section 2. We provide an overview of the `BLUTune` system in Section 3 where the specifics of the reinforcement learning and query embedding processes are expounded upon. We go over a system demonstration that shows how users interact with the tool in Section 4 and go over the experimental results in Section 5. We describe related work in Section 6, and leave our concluding statements and future work in Section 7.

Chapter 2

Background

2.1 A Real-world Query Example

Knobs affect costs and execution times of queries by directly influencing the structure and characteristics of the QEPs created by the query optimizer in `IBM Db2`. The *optimization-level* knob, for example, specifies which classes of query optimization techniques are applied when preparing dynamic SQL statements. The higher the level, the more techniques are engaged. In Figure 1.3 and Figure 1.4, we illustrate two QEPs for a portion of TPC-DS's query #98 that result from different optimization-level settings. Each plan operator (e.g., `MSJOIN`) in `IBM Db2` is referenced in `IBM Db2` as a low level plan operator (LOLEPOP). The top-most decimal number (possibly in exponential notation) is the estimated cardinality, followed by operator type, and then the operator ID (the integer in parentheses). The bottom-most decimal number (again possibly in exponential notation) is the estimated cost; the estimated cost is reported as a proprietary unit of measure called a *timeron*. A timeron is a cost estimate calculated by evaluating the resources that will be used for a particular

operation. It is derived using both I/O cost and processor cost. To determine the relative importance of I/O versus CPU, weighting factors are applied that may vary based on the CPU model being used. The Db2 optimizer utilizes statistics, indexes, filter factors, and other available information to estimate the costs before applying the weighting factors.

The first plan in Figure 1.3 was produced under the optimization-level setting of 1 (which offers a set of basic optimizations applied during plan generation). This plan has an expensive merge-sort join (`MSJOIN`) between the tables `WEB_SALES` and `WEB_RETURNS`. A merge-join requires ordered input on the joining columns, provided either through index access or by sorting. Both tables are column organized, and are first read by a table scan (`TBSCAN`) operators. The optimizer chose then as the next step to convert the column-organized data into a row-organized format via a column-organized table queue (`CTQ`) operator. An effect of this is that all subsequent operations do not leverage the column organized vectors, compressed data and the many optimization implemented in the column engine. The row-organized data is sorted by the `SORT` operator, read then again by a `TBSCAN`. The intermediate result from the `WEB_SALES` table is filtered by a `FILTER` operator by a predicate before finally being joined with the sorted `WEB_RETURNS` table via a `MSJOIN`, with a resulting estimated cost of 1.16645e+06.

For this query, however, an optimization level of 5 works better. The resulting QEP in Figure 1.4 is much simpler, and is estimated to be less expensive with a total cost of only `368880`. The `WEB_SALES` and `WEB_RETURNS` tables are joined by a hash join operator (`HSJOIN`) which does not require ordered input on the joining columns. Provided the sortheap is sufficiently large enough to hold the lookup table required

for hashing the join column values, the `HSJOIN` is much faster and less expensive than the `MSJOIN` in this case. Not only does this avoid two large `SORT` operations, it also retains the data in a column-organized format, not needing the `CTQ` operator. This allows subsequent operations to be accelerated as the data is still compressed column-organized vectors and tuples. The resulting elapsed run-time, including execution and compile time, for a query optimization level of 1 was 157.86 seconds versus 64.21 seconds for an optimization level of 5, a significant 60% reduction.

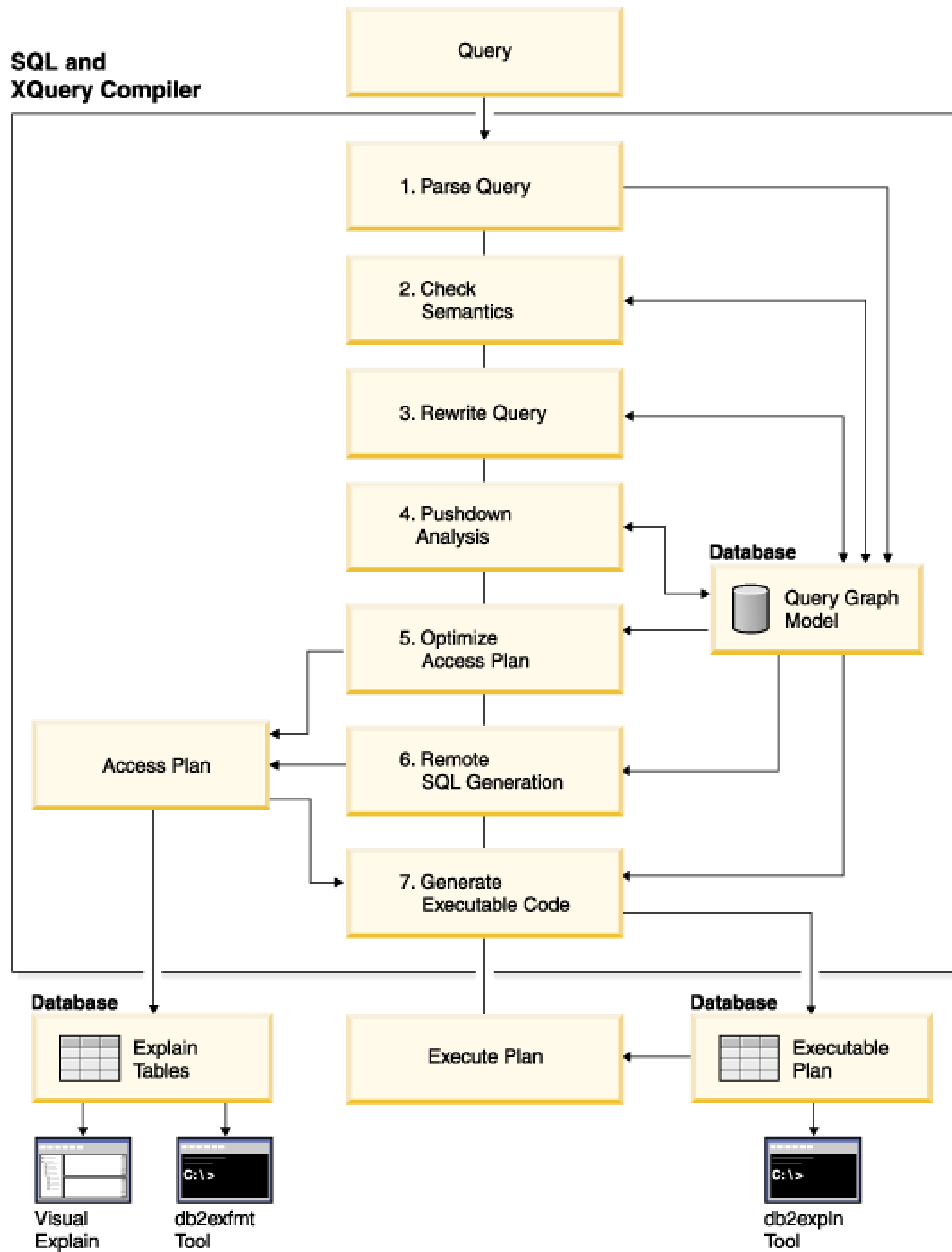


Figure 2.1: Query Compilation Process

2.2 SQL, Db2, and Query Optimization

2.2.1 Query Optimization Steps

In order to understand how QEP2vec parses the QEPs as well as how and why `BLUTune` uses that information to perform knobs tuning, it is important to understand the life-cycle of a query from the time it is written to the time that it is executed. The query compilation process goes through seven main steps outlined in Figure 2.1[19]. The steps are as follows:

1. **Parse Query**

   The first stage is to take the SQL query that is written by the user and parse the statement to ensure that the syntax of the query is correct. An internal representation is then stored using the *Query Graph Model* (QGM). The query text itself is currently not used in the representation but may be a valuable source of information as discussed in Section 7.2.

2. **Check Semantics**

   Once the compiler ensures that the query has the correct syntax, it checks for logical consistencies within the QGM. For instance, an operation that requires a piece of data to be a float would check that the column being specified to work with that operation actually contains a float.

   The query is still in a form that is specified by the SQL statement provided by the user. The compiler may now enforce some constraints on the state of the QGM to ensure that the integrity of the query semantics are maintained before moving on to the next step.

```
Original Statement:
------------------
-- query_00_01_96.sql
select
        count(*)
        from    store_sales,   household_demographics,   time_dim,   store
        where
                ss_sold_time_sk = time_dim.t_time_sk
            and ss_hdemo_sk = household_demographics.hd_demo_sk
            and ss_store_sk = s_store_sk
            and time_dim.t_hour = 16
            and time_dim.t_minute >= 30
            and household_demographics.hd_dep_count = 0
            and store.s_store_name = 'ese'
        order by
            count(*)
```

```
Optimized Statement:
-------------------
SELECT
  Q6.$C0
FROM
  (SELECT
     COUNT(*)
   FROM
     (SELECT
        Q1.S_STORE_SK
      FROM
        DS100.STORE AS Q1,
        DS100.TIME_DIM AS Q2,
        DS100.HOUSEHOLD_DEMOGRAPHICS AS Q3,
        DS100.STORE_SALES AS Q4
      WHERE
        (Q1.S_STORE_NAME = 'ese') AND
        (Q3.HD_DEP_COUNT = 0) AND
        (30 <= Q2.T_MINUTE) AND
        (Q2.T_HOUR = 16) AND
        (Q4.SS_STORE_SK = Q1.S_STORE_SK) AND
        (Q4.SS_HDEMO_SK = Q3.HD_DEMO_SK) AND
        (Q4.SS_SOLD_TIME_SK = Q2.T_TIME_SK)
     ) AS Q5
  ) AS Q6
ORDER BY
  Q6.$C0
```

Figure 2.2: Original vs. Optimized Statement Text for TPC-DS Query

3. **Rewrite Query**

Abiding by the constraints set in step 2, the optimizer will perform many different rewrite rules to make a query run more efficiently. After the rewrite stage, the query itself may not be in the optimal state but the statements are put into forms that can be better optimized. An example of the type of rewrite that can be performed is *general predicate pushdown* in which the compiler moves the level at which a predicate is evaluated. For instance, in Figure 2.4, a predicate that is acting on the GRPBY(3) may only be filtering results from Q1; pushing the predicate down to the TBSCAN at operator (10) may reduce the amount of computation required and speed up the query while maintaining the same semantics that were outlined by the QGM in step 2.

In general, aggregation, redistribution of rows, and correlated sub-queries, are significantly more expensive on partitioned data systems since the data required

may reside on different systems and might incur lots of communication costs. Query rewrite rules like general predicate pushdown may alleviate this by removing the need for communicating the results if the pushdown results on decorrelating the data such that each predicate only operates on data that is local to the data system.

Another example of query rewrite rule is *predicate translation*[20] where the compiler takes a predicate and turns it into a more efficient form. For instance, a query containing a sequence of 'OR' predicates may be switched to an 'IN' predicate as seen below:

```
select *
     from employee
     where
       deptno = 'D11' or
       deptno = 'D21' or
       deptno = 'E21'
```

translates to

```
select *
  from employee
  where deptno in ('D11', 'D21', 'E21')
```

Once the rewrite rules have been applied, the explain tables will capture both the original statement text as well as the optimized statement text, as can be seen in the explain plan format tool's output in figure 2.2.

4. Pushdown Analysis[1]

[1] This step is only taken for federated data systems.

A more generalized version of general predicate pushdown that aims to determine whether operations can be pushed down to be evaluated directly on the data source or remotely evaluated.

5. Optimize Access Plan

   Once the query is rewritten, the optimizer gathers statistics from the QGM regarding the components of the query and their estimated execution costs in order to generate alternative plans that may satisfy the query at lower costs. Additional forms of pushdown in the forms of aggregation pushdown and sort pushdown occur, and the system makes use of which buffer pools are available to generate the final version of the access plan.

   In figure 2.3 we can see a graphical depiction of the optimized access plan that is output by the IBM Db2 explain table format command, while in figure 2.4 we can see the access plan as visualized by the `BLUTune` system.

2.2.2 IBM Db2 Knobs

Sort heap is a temporary memory structure used by a data system to sort data during query processing. When a query involves an ORDER BY or GROUP BY clause, the data system needs to sort the relevant data before returning the results. To do this efficiently, the system will typically allocate a portion of memory called the sort heap, which is used to store and manipulate the data during the sorting process. The size of the sort heap can be configured by the database administrator or may be determined automatically based on the available system resources and the size of the data being sorted. A larger sort heap can improve sorting performance by reducing the need to spill data to disk, but can also consume more system resources.

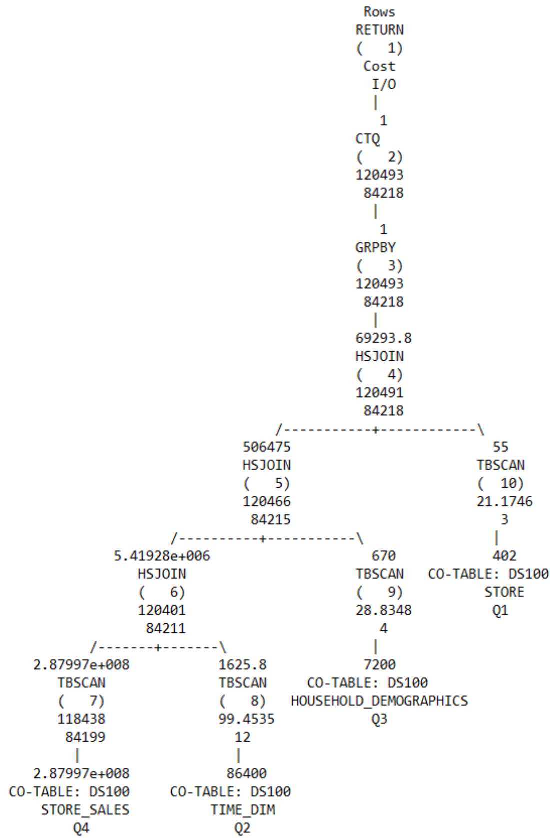


Figure 2.3: Query Explain Plan provided by IBM Db2's exfmt tool

A buffer pool, on the other hand, is a portion of memory used by a data system to cache frequently accessed data from disk. When a query needs to access data from a table or index that is stored on disk, the system will read the relevant data into the buffer pool, where it can be accessed more quickly by subsequent queries. The buffer pool helps to reduce the amount of disk I/O required to process queries and can significantly improve database performance. The size of the buffer pool can also be configured by the database administrator, and is typically based on the available system memory and the size of the data being accessed. A larger buffer pool can improve query performance by reducing the amount of disk I/O required to access

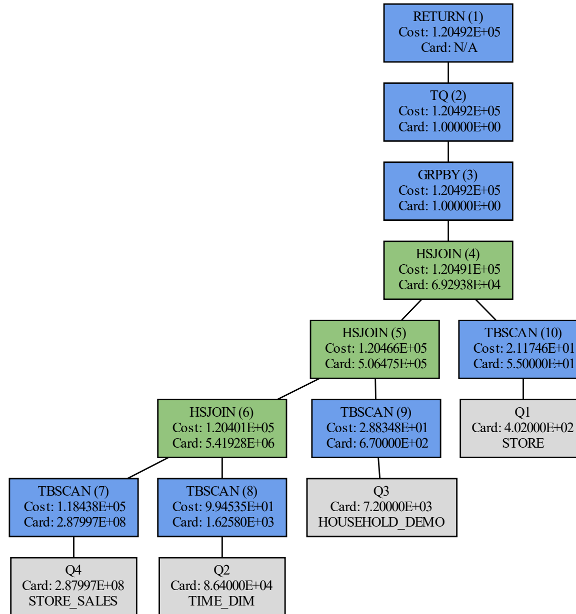


Figure 2.4: Query Explain Plan Visualization provided by BLUTune

data, but can also consume more system resources.

Both the sort heap and buffer pool are important components of data systems that help to optimize query processing and improve performance. The optimal sizes for these memory structures depend on many factors, including the size and complexity of the data, the available system resources, and the specific workload being processed. Since these knobs are continuous values, it can be time consuming and tedious to tune these by hand. In addition to that, there are a plethora of other knobs that can be hard to understand such as *query optimization hints*.

One such hint in IBM Db2 is the *NO_SORT_MGJOIN* hint that can be used to instruct the query optimizer to use a merge join algorithm instead of a sort-merge join algorithm. A merge join is a join algorithm that can be used to join two sorted

```
3) GRPBY : (Group By)
        Cumulative Total Cost:          120493
        Cumulative CPU Cost:            1.08496e+011
        Cumulative I/O Cost:            84218
        Cumulative Re-Total Cost:       120493
        Cumulative Re-CPU Cost:         1.08496e+011
        Cumulative Re-I/O Cost:         84218
        Cumulative First Row Cost:      120493
        Estimated Bufferpool Buffers:   0

        Arguments:
        ---------
        AGGMODE : (Aggregation Mode)
                HASHED COMPLETE
        GROUPBYC: (Group By columns)
                FALSE
        GROUPBYN: (Number of Group By columns)
                0
        MAX CARD: (Maximum Cardinality)
                1

        Input Streams:
        -------------
                11) From Operator #4

                        Estimated number of rows:       69293.8
                        Number of columns:              1
                        Subquery predicate ID:          Not Applicable

                        Column Names:
                        ------------
                        +Q5.S_STORE_SK
```

Figure 2.5: Example QEP Node

input streams, and is generally faster than a sort-merge join algorithm. When the *NO_SORT_MGJOIN* hint is used, the query optimizer will try to use a merge join algorithm if possible, without first sorting the input streams. The sort-merge join algorithm, on the other hand, involves sorting the input streams before joining them, which can be expensive for large data sets. By using the *NO_SORT_MGJOIN* hint, the query optimizer may be able to generate a more efficient execution plan for the query.

Query optimization hints can have unintended consequences and may not always result in improved query performance. The optimal execution plan for a query depends on many factors, including the structure of the query, the size and distribution

```
-------------- epoch 24: iter 131 start: --------------
Inferred from ['0-103', '6-161', '1-128', '129-0', '130-0', '101-14']
reward: 5.99421015732687 cost_reward: 3.79392E+01                    best_cost_reward: -4.00579E+00
cost:                           2.24814E-01
sortheap                      :  2456.5838277339935
bufferpool                    :  145558.11882019043
query_opt                     :  7
COLJOIN                       :  1
XGBPART ON FULL NONHDIR       :  0
GY_DELAY_EXPAND 1000          :  0
NI2NE_WITH_NULLS OUTER        :  0
PDAOPT_VAL ON,PDAOPT_PREP ON  :  0
NO_HSJN_BUILD_FACT ON 1       :  0
NO_TQ_FACT ON 1               :  1
NO_SORT_MGJOIN                :  1
INJ_DUMMY_PRD4JOIN ON         :  1
CDE_PDA_CACHING               :  1
EGAD 0                        :  0
ep:24, iter:131, stable_counter:0, queries:['c:/blutune/data/DS100/queries/query_00_62_41.sql']
```

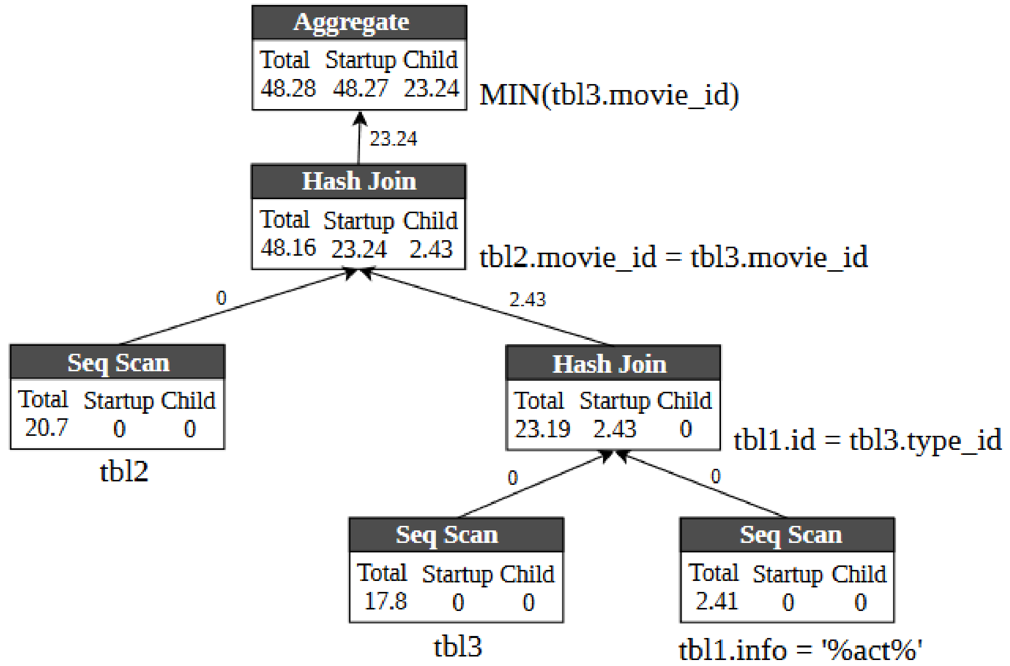
Figure 2.6: An Example Recommendation Encountered During Training

of the data, and the available system resources. It is generally best to rely on the query optimizer to generate the most efficient execution plan, and to use query optimization hints only when necessary and based on careful testing and analysis. A list of the various continuous valued knobs as well as hints used in tuning can be seen in Figure 2.6

2.3 Query Embeddings

Given that the intention is to use the queries in a machine learning pipeline, they must first be put into some type of fixed length representation. A query does not have a natural fixed length representation since a query can form a graph that is arbitrarily large. Query featurization is the process of converting a SQL query into a set of numerical features that can be used as input to machine learning models for query optimization. Historically, database tuning systems are agnostic to the information contained within a query. Huawei's `QTune` system uses query featurization to generate

```
SELECT    MIN(tbl3.movie_id)
FROM      tbl1, tbl2, tbl3
WHERE     tbl1.info = '%act%'
    AND   tbl1.id = tbl3.type_id
    AND   tbl2.movie_id = tbl3.movie_id
```



Insert	Delete	Update	Select	tbl1	tbl2	tbl3 ... tbl8		Hash_Join	Seq_Scan	Aggregate	...
[0	0	0	1	1	1	1 ...	0	68.92	40.91	25.04	...]
(1) DML				(2) Tables				(3) Operation Costs			

Insert	Delete	Update	Select	tbl1	tbl2	tbl3 ... tbl8		Hash_Join	Seq_Scan	Aggregate	...
[0	0	0	1	1	1	1 ...	0	0.1401	-0.166	-0.2423	...]

Normalized Feature Vector

Figure 2.7: Featurization Strategy Employed by QTune

features that capture various aspects of a SQL query, such as the type of query operation being performed, the tables and columns involved, and the aggregated then normalized costs associated to each operator. This process is detailed in Figure 2.7.

In order to accomplish our goal of turning our queries into feature vectors without the pitfalls of former methods, we come up with the notion of *query templating* and then use graph-based embedding techniques (Section 3), the benefits of which are outlined in Section 4.2). The query is first sent to the DBMS to produce the QEP before it is then broken down into its constituent elements. Each node in the QEP (an example of the information contained within a node is shown in Figure 2.5) is parsed for certain details. First, we extract the operator for a given node and the nodes relationship to other nodes within the graph. Starting at the nodes connected to the tables, we work our way up the QEP and keep track of the relative depth of the nodes within the joins, creating canonical labels for referencing these components of the tree. Each node is represented within a pair of parentheses; the first number represents the depth, and the second number represents the relative position of the node within the input streams. For instance, the subgraph located at node 6 within the Figure 2.3 would take the form *'(0-TBSCAN-0)-(1-HSJOIN-0),(0-TBSCAN-1)-(1-HSJOIN-0)'* which would act as a key to a dictionary; the sub-graph starts at depth 0 with nodes 7 and 8 which are both TBSCAN's that connect to a HSJOIN. The first TBSCAN is the inner-most stream giving it the suffix -0, while the second TBSCAN is the input stream that follows it, giving it a suffix of -1. The resulting label is called a *template*.

Not all templates are considered equal by virtue of their operators alone, however. In addition to the operators and the relationships between them, certain aspects of

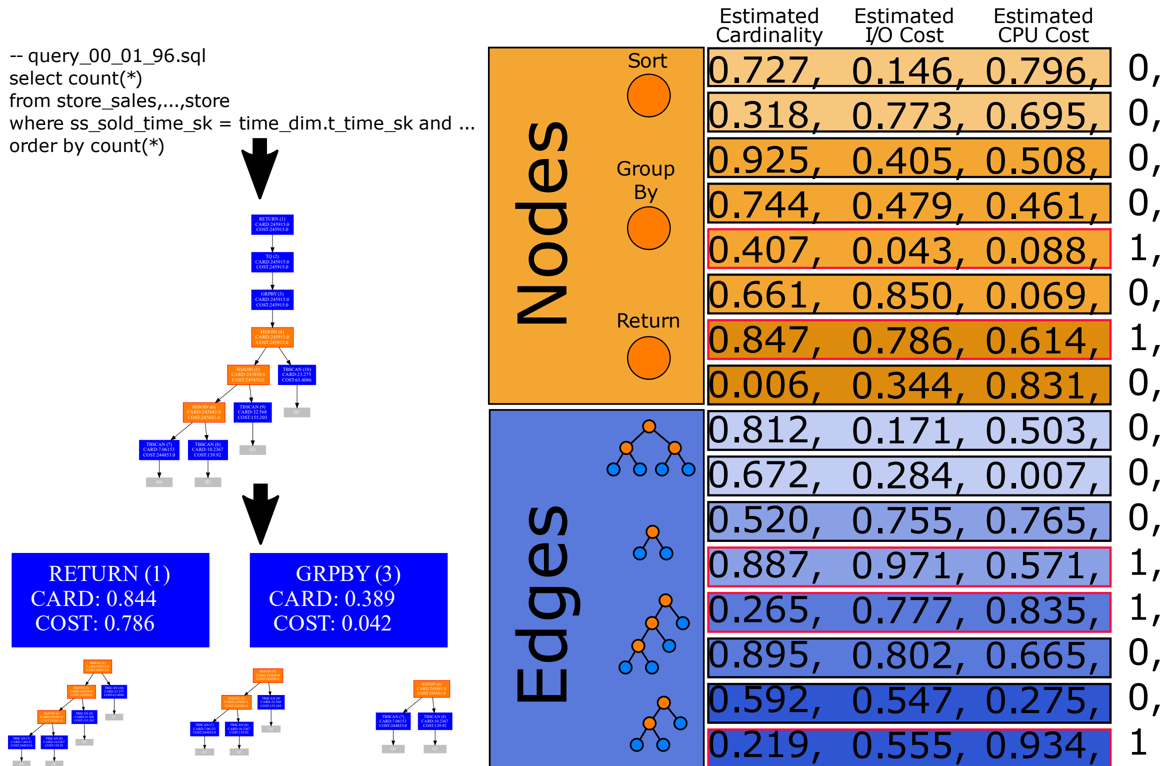


Figure 2.8: Example feature vector obtained via the representative strategy

the costs at that node are included such as the total cost, the CPU cost, the I/O cost, and the First Row cost. These amounts are cumulative, however, and so the costs must be subtracted from the costs that are associated to the input streams; this means subtracting the costs from the HSJOIN at node 4 from the proceeding GRPBY at node 3 in the provided QEP. The cost vectors themselves are continuous values and would create far too many templates should we choose to include them all. To rectify this, we instead choose to keep a list of the costs associated to a template and only add new cost values to this list if we have not yet encountered a cost vector that is within a certain percentage based threshold. When we have a cost associated to a template, we call this a *representative*.

Before we perform the reinforcement learning algorithm to determine the optimal knob settings, we first have to perform a representation learning algorithm in order to learn a fixed-length feature vector for our arbitrarily sized query graphs. The process is described in more depth in Section 3 but the inputs to the representation learning algorithm are the queries and the list of graph features that are present within the query. We can see an example of the representative strategy in Figure 2.8. In this figure, the query is broken down into the elements of interest (the 1-,2-, and 3-way joins, as well as select nodes) with their extracted costs. The block labeled 'Nodes' and 'Edges' represents the dictionary that is created during the representation learning stage; here we can see the 5 components that are extracted are matched to their closest representatives within the dictionary which results in a string of 0's and 1's that represent that specific query.

Chapter 3

System Overview

A *workload* is defined by a database schema, the data (populated with respect to the schema), and a collection of SQL queries that are periodically executed. A *query execution plan* (QEP) is constructed by the query optimizer for an SQL query, to be evaluated at runtime [21]. Within `IBM Db2`, query plans are represented in the *query graph model* (QGM) [22], [23]. An SQL query is first parsed into a QGM representation; then that QGM is rewritten by `IBM Db2`'s query *rewrite engine* to simplify the query, but preserving the query's semantics. The rewritten QGM is passed to `IBM Db2`'s *cost-based optimizer*, which annotates the QGM into a full-fledged QEP. The QEP profiles the access paths chosen by the optimizer, the chosen join types (e.g., sort-merge join), the join order, and index usage.

`BLUTune` is a system to automate fully the knobs tuning process for `IBM Db2` with the goal of maximizing the performance for a given SQL workload. Runtime is measured as the total elapsed time for compiling and executing a query until a result is returned. An architectural overview of `BLUTune` is illustrated in Figure 3.1.

The collection of SQL queries from the workload in question is processed through

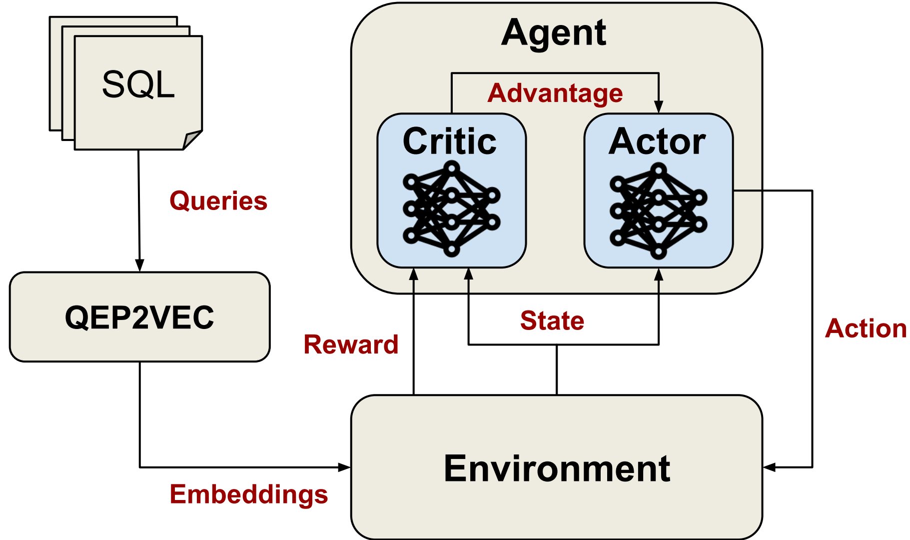


Figure 3.1: BLUTune architecture.

the `IBM Db2` environment, retrieving information about QEPs for the queries, but without executing the queries themselves. A QEP contains information about the query, including the operators and their costs, and information about the underlying data statistics. This information is fed into `QEP2Vec`, which vectorizes the high-dimensional QEPs into a low-dimensional embedding space (discussed in Section 3.2). Each query can be handled and tuned for individually, or the entire cluster / workload can be tuned for. When performing tuning for a given workload, the average embedding of each query within the workload is used as input to the reinforcement learning model. The recommended knobs are then applied and used for the workload as a whole. If a user instead wishes to use clustering, we can perform principal component analysis to reduce the dimensionality of the embedding space, infer the embedding for each query within the workload, then use DBSCAN to perform clustering. Each query cluster is then treated like a workload as described above to obtain a knob recommendation for that query cluster.

Given a QEP embedding vector as part of the input, `BLUTune`'s deep reinforcement learning agent produces a knob configuration and applies it to the data system (Section 3.1). The resulting performance metrics for queries under this new tuning configuration are reported back to the agent to guide the learning.

3.1 Deep Reinforcement Learning

Reinforcement learning (RL) is a type of machine learning where an agent interacts with an environment, performs an action, and receives rewards based on the actions taken, with the goal of learning a policy that maximizes cumulative rewards over time through trial and error. In Deep reinforcement learning (DRL), artificial neural networks (ANN) are used as function approximators to guide the network when trying to estimate the reward for performing a given action. The RL network is typically trained using techniques such as Q-learning or policy gradient methods, and the ANNs allows it to learn complex and abstract representations of the environment and its dynamics. RL techniques can be categorized into two types: value-based and policy-based. Value-based methods aim to learn the optimal value function, which is a mapping between an action and its corresponding value. Deep Q Learning is an example of a value-based method, where a learned function $Q(s, a)$ estimates the expected reward an agent can get in a state s by taking an action a. The larger the Q-value, the better the action is considered to be. Policy-based methods, on the other hand, aim to learn the optimal policy directly, which is a mapping between a state and an action, without using Q-values. Policy gradient methods are an example of policy-based methods. Value-based methods are considered more sample-efficient and stable, while policy-based methods work better for continuous environments and

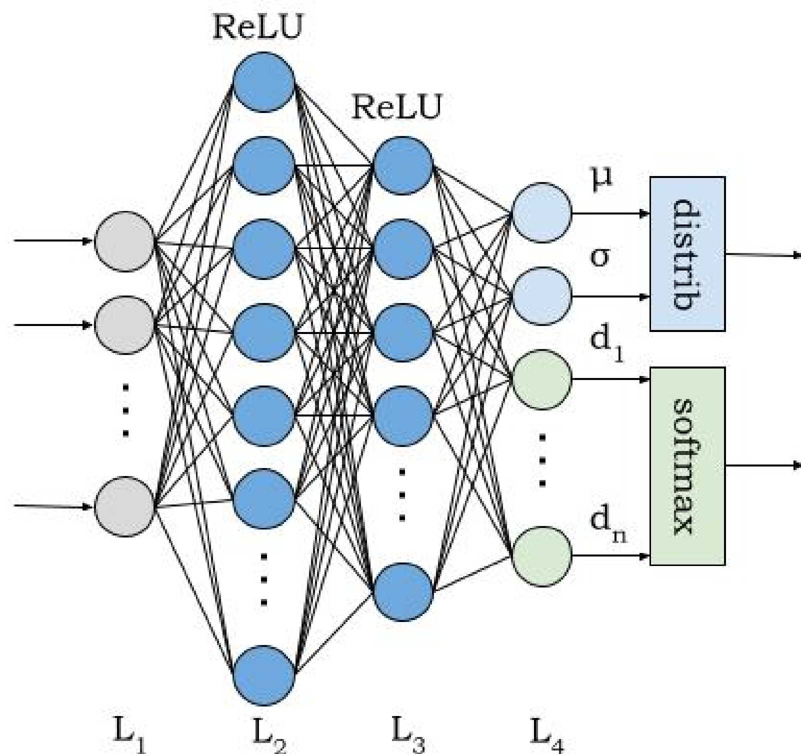


Figure 3.2: Actor network.

converge faster.

Actor-Critic is a hybrid technique that combines both policy-based and value-based methods, aiming to leverage the advantages of both. The basic idea is to divide the agent model into two parts, an Actor and a Critic. The Actor learns an optimal policy π (policy-based), and the Critic learns the Q-value $Q(s, a)$ of each state s and action a (value-based). The actor network can be seen in Figure 3.2 with the discrete and continuous outputs while the critic network is outlined in Figure 3.3. The interaction between the actor and critic occurs during training, where the agent interacts with the environment, receives rewards, and updates its policy and value estimates based on the observed experiences. The critic provides feedback to the

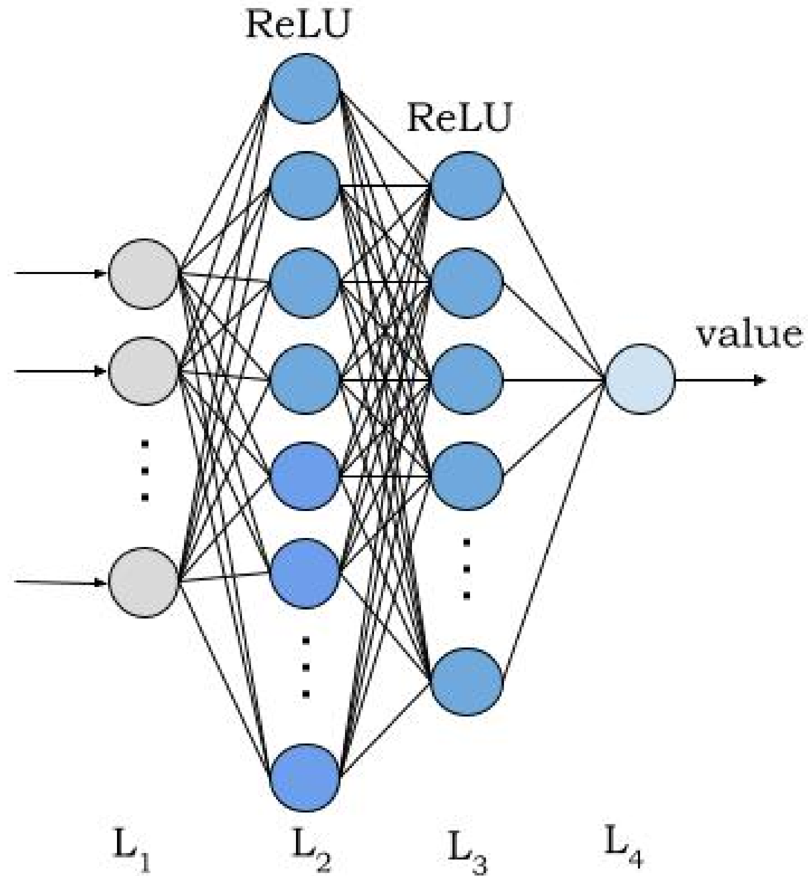


Figure 3.3: Critic network.

actor on the quality of its actions by estimating the value function of state-action pairs. The actor uses this feedback to improve its policy by selecting actions that have a higher estimated value. In turn, the actor's policy affects the critic's value estimates. As the actor explores the environment and tries new actions, it provides the critic with new data to improve its value estimates. `BLUTune` uses a specific type of DRL architecture called Advantage Actor-Critic (A2C). In traditional Actor-Critic algorithms, the value function is estimated using temporal difference (TD) learning, and the policy is updated based on the estimated value function using the policy gradient method. However, in "Advantage Actor-Critic" (A2C), the value function

is estimated as the advantage function, which is the difference between the expected value and the value of a state-action pair, and is used to update the policy directly.

$$Q(s, a) = r(s, a) + \gamma V(s') \tag{3.1}$$

$$\mathbf{A}(s, a) = Q(s, a) - V(s) \tag{3.2}$$

$$\mathbf{A}(s, a) = r(s, a) + \gamma V(s') - V(s) \tag{3.3}$$

Hence, we only have to learn the state value function V. By learning the advantage instead of the Q-value function, we can simplify the input to our model *and* also reduce the variance within the policy network, resulting in a more stable model. The weights for the policy (θ) and value function (w) are updated following Equations 3.4 and 3.5, respectively. For an action a and a state s, the policy π is updated by multiplying the advantage with the gradient of the log probability of the action chosen. The value function V_w is updated by multiplying the squared advantage with the gradient of itself.

$$\Delta\theta = A(s, a)\nabla_\theta \log \pi_\theta(a \mid s); \tag{3.4}$$

$$\Delta w = A(s, a)^2 \nabla_w V_w(s) \tag{3.5}$$

Algorithm 1 Parsing graph for labels

Input: Query of interest
Output: A list of the labels and associated costs for the given query

```
execute query Q while data system is in explain mode
node_list ← Query explain tables for nodes in QEP
edge_list ← [], join_list ← [], join_candidates ← [], nodes ← dict()
for node in node_list do
    if node.parent != null then
        edge_list.append(node.parent, node)
        nodes[node] ← (operator, depth,  children, label)
    end if
end for
for node in node_list do
    if node.parent.child_count   1 then
        node.parent.relative_depth = 0
        if node.parent not in join_list then
            join_candidates.append(node.parent)
        end if
    end if
    node.relative_depth ← node.parent.relative_depth + 1
end for
for i ← 0 to len(join_candidates) do
    depth ← 0
    candidates ← [ join_candidates[i] ]
    while candidates != [] do
        new_candidates ← []
        for candidate in candidates do
            for child in nodes[candidate].in_oper do
                new_candidates.append(child)
                join_labels[i] ← label
                join_costs[i] ← node.costs - node.children.costs
            end for
        end for
        candidates ← new_candidates
        depth += 1
    end while
end for
return join_labels, join_costs
```

Algorithm 2 Getting Representatives from Graph

```
Input: graph,
representatives ← dict(),
Output: A list of the representatives present in the given graph
  % Get the list of operators/joins that are of interest
  operator_candidates ← []
  join_candidates ← [graph.join_labels, graph.join_costs]
  nodes ← graph.nodes
  for node in nodes do
      if nodes[node].operator in {'GRPBY','SORT','RETURN'} then
          operator_candidates.append([node, nodes[node].operator,
              nodes[node].costs])
      end if
  end for

  % Using the operator/join labels, find the representatives
  results ← [ ]
  feature_vector ← operator_candidates + join_candidates
  for vector in feature_vector do
      % vector[1] contains the labels; vector[2] contains the costs
      % Add the vector if it's a new template
      if vector[1] not in representatives then
          % Add the cost to the list of representatives for that label
          representatives[vector[1]] ← [vector[2]]
          % For this graph, store the label appended by the index of the
          % closest template match (only the 0th exists)
          results.append(vector[1] + "0")
      else
          closest_idx = idx of representatives[vector[1]] entry
              with smallest dist() to vector[2]
          results.append(vector[1] + closest_idx)
      end if
  end for
  return results
```

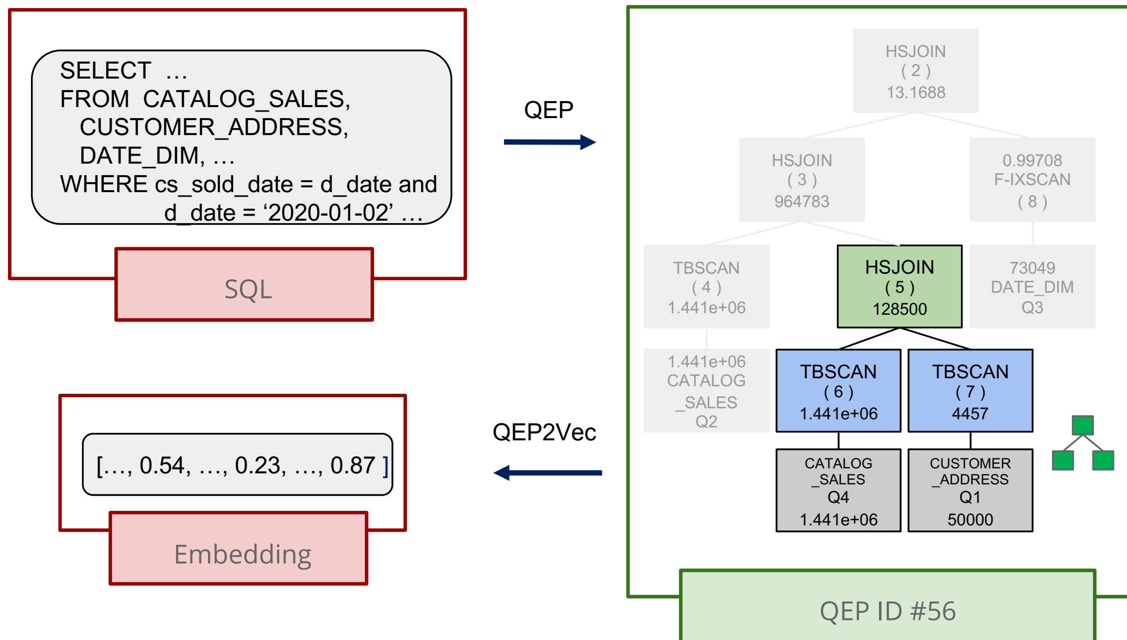


Figure 3.4: SQL query representation.

3.2 Query Embedding

We present how to produce embeddings for SQL queries. First, for a given QEP, all sub-plans—contiguous fragments of the QEP, so corresponding to sub-queries—are generated, up to a predefined size threshold (number of joins). In Figure 3.4, an SQL query is illustrated which consists of a two-way join between TPC-DS benchmark schema tables `CATALOG_SALES` and `CUSTOMER_ADDRESS`. A sub-plan projects the join and local predicates from the original QEP that are applicable to the sub-plan's selected tables. For the threshold, we have verified that, in practice, up to four-way join is optimal. This process is recursively applied until the stopping `LOLEPOP`, denoted as root, is reached in the QEP.

For abstraction, the table and column names in the plans are replaced by canonical

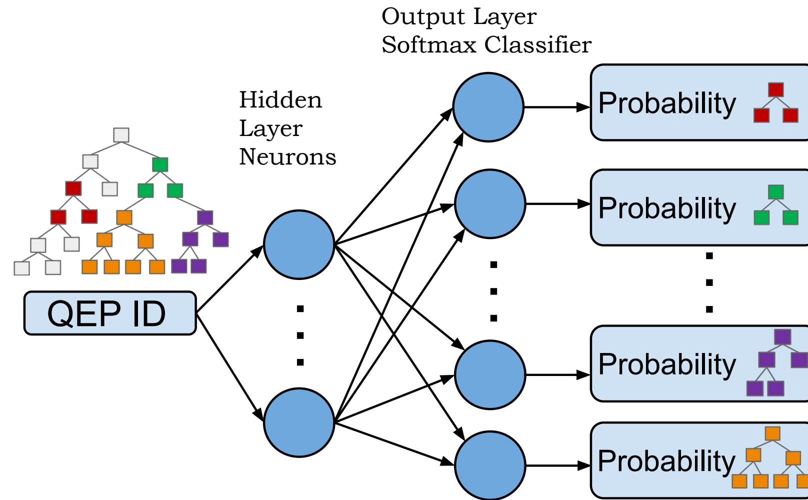


Figure 3.5: QEP2Vec embeddings.

symbol labels. Sub-plans are grouped into *templates* with similar ranges of lower and upper-bounds of estimated costs and cardinalities. Thus sub-plans with similar substructure and characteristics, but with different context of tables and column names, can match, even across different workloads. We can see an example of query execution plan and a highlighted join that we would like to create a template for in Figure 3.6. The associated template and the costs associated to that template are seen in Figure 3.7. In Figure 3.8 we can see an example of a template match that is detected for a query run at two different settings; the differences in estimated costs is rather stark and is thus more likely to cause the query to create two different representatives for the given query. Figure 3.9 highlights a similar scenario but the differences in sortheap and bufferpool have cause the join order to swap. Since the templating method is agnostic to tables and the operators are otherwise the same, these would have the same template base but different representatives due to cost differences.

Since query plans are in a high-dimensional space, we transform them into a low-dimensional vector space via embeddings. In `QEP2Vec`, we consider QEPs and the derived sub-plan templates in analogy with documents and words, respectively. The `Doc2Vec` method [18] uses an instance of the skip-gram model called a *paragraph vector distributed bag of words* (`PV-DBOW`) to learn representations of word sequences of arbitrary length in documents.

To compute our `QEP2Vec` embeddings we construct a fake task to calculate the probability for every 2-, 3-, and 4-way join to appear in a QEP using a skip-gram neural network. This is illustrated in Fig. 3.5. In truth, we are not interested about these probabilities. We ignore the second layer; the first hidden layer matrix weights become our embeddings. So, given a set of QEPs $\mathscr{Q} = \{q_1, q_2, \ldots, q_n\}$ and a sequence of 2-, 3-, and 4-way join sub-plan templates $\texttt{sub}(q_i) = \{t_1, t_2, \ldots, t_m\}$ from each QEP $q_i \in \mathscr{Q}$, `QEP2Vec`'s skip-gram learns a θ-dimensional embedding of the QEP. The model maximizes the following *log likelihood* by considering a sub-plan template $t_k \in \texttt{sub}(q_i)$ to appear in the context of QEP q_i:

$$\sum_{k=1}^{m} \log P(t_k \,|\, q_i) \tag{3.6}$$

The key idea is that structurally similar QEPs will be close in the embedding space; i.e., given a pair of QEPs q_i and q_j, the training will lead to their embeddings $\Omega(q_i)$ and $\Omega(q_j)$ being close if they consist of similar 2-, 3-, and 4-way join sub-plan templates, but being distant, otherwise. The procedure to learn query plan embeddings is presented in Algorithm 3. We first extract 2-, 3-, 4-way joins sub-plan templates in each of the QEPs (Lines 3–4), parametrized as K-way joins. We then learn the corresponding QEPs embeddings in multiple epochs maximizing the log likelihood

(Lines 2-7). We use a stochastic gradient descent optimizer to learn the parameters in Lines 6–7. We estimate the derivatives by using back-propagation with a learning rate α that is tuned empirically.

Algorithm 3 QEP2Vec(Q, δ, α, E, K)

Input: QEPs $Q = \{q_1, q_2, ..., q_n\}$, embedding size δ, learning rate α, # of epochs E and degree of K-way joins
Output: Matrix representation of QEPs Ω
  for $e = 1$ to E **do**
    for each $q_i \in Q$ **do**
      for k = 0 to K **do**
        Add to $sub(q_i)$ K-way join subplan templates
      end for
      $J(\Omega) = -\sum_{k=1}^{m=|\texttt{sub}(q_i)|} \log P(t_k \mid \Omega(q_i))$
      $\Omega = \Omega - \alpha \frac{\partial J}{\partial \Omega}$
    end for
  end for

As the number of sub-plan templates in a complex query workload can be large, we use a negative sampling strategy. For each training round in Algorithm 3, for a given QEP, we select a predefined number of random sub-plan templates. Consequently, this can be computed efficiently by updating only embeddings of the negative sample, rather than the entire lexicon.

Once the `QEP2Vec` embeddings are computed, they can be used in variety of applications, including automatic knob tuning and query clustering. We illustrate in our experiments in Section 5 that, by using our `QEP2Vec` embeddings, we achieve significantly better performance than `QTune` [17], which uses a simple featurization of the QEP based on the SQL query's features, aggregated costs of operators, and table and column names.

Query clustering can be performed by obtaining the embeddings for each query

within a query workload and using the average of the embedding vectors to represent the embedding of the workload as a whole. Knob recommendations can then be obtained for the query workload as a whole by providing the workload embedding to the `BLUTune` system the same way we obtain a recommendation for an individual query.

Using the workload embedding to provide knob recommendations will provide good catch-all settings for the queries, however, if one of the queries requires drastically different settings than the rest of the workload than there will be a degradation in performance across the batch as a whole. To that end, we can perform clustering within the embedding space. We first reduce the dimensionality of our training queries embeddings via Principal Component Analysis (PCA) in order to make clustering within the space more feasible, then using Density-based spatial clustering of applications with noise (DBSCAN) we perform the clustering to assign each query within the workload to a cluster. The query workload is then split into separate workloads based on which cluster they are present in, and the knob recommendations for each cluster are provided.

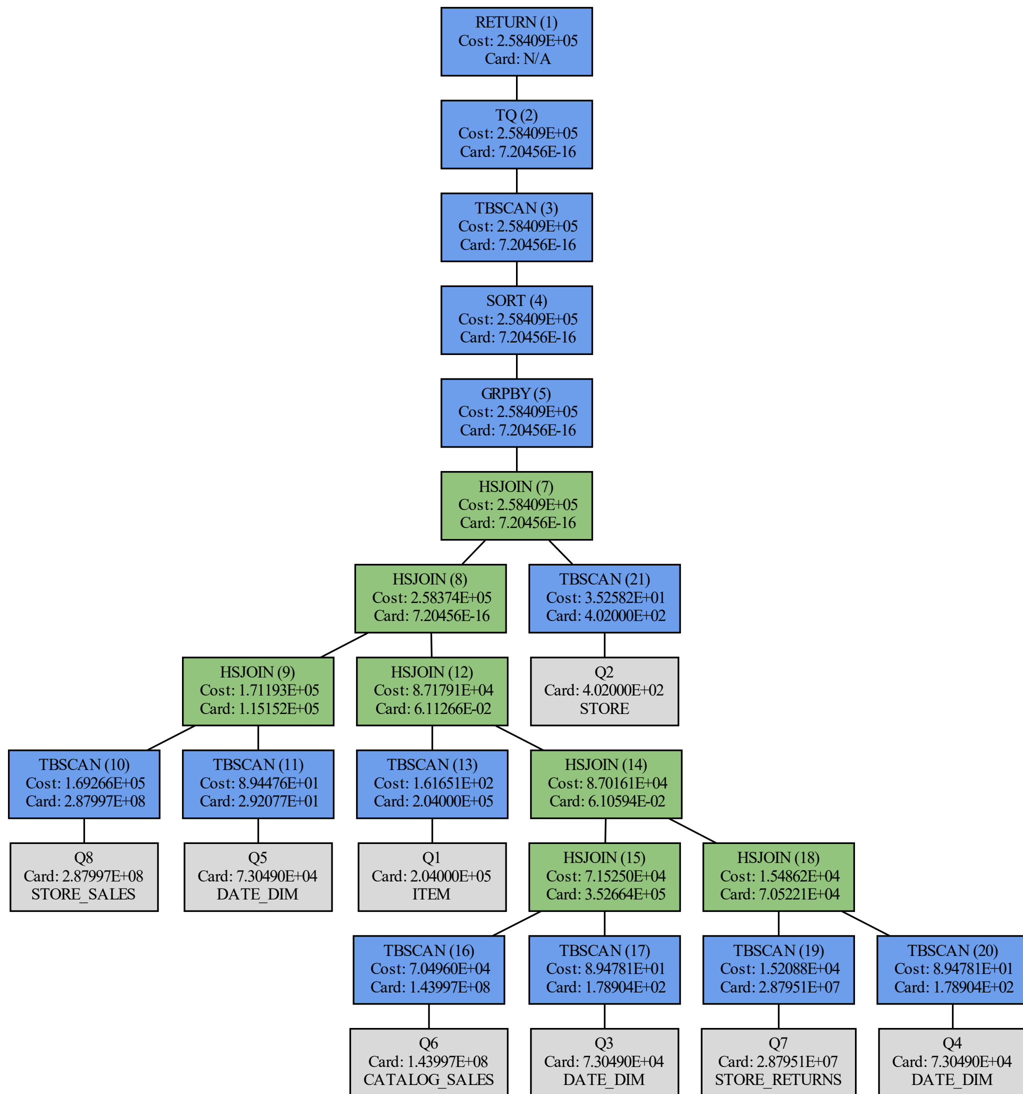


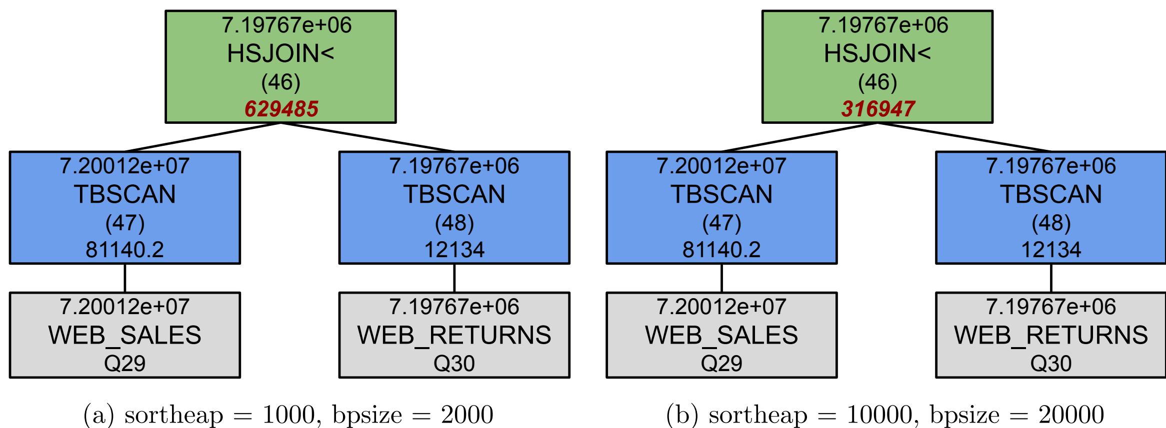
Figure 3.6: Query #8 from TPC-DS Benchmark

Template label:

['(0-HSJOIN-0)-(1-HSJOIN-0)',
'(0-HSJOIN-0)-(1-HSJOIN-1)',
'(1-HSJOIN-0)-(2-TBSCAN-0)',
'(1-HSJOIN-0)-(2-TBSCAN-1)',
'(1-HSJOIN-1)-(2-TBSCAN-0)',
'(1-HSJOIN-1)-(2-HSJOIN-1)',
'(2-HSJOIN-1)-(3-HSJOIN-0)',
'(2-HSJOIN-1)-(3-HSJOIN-1)',
'(3-HSJOIN-0)-(4-TBSCAN-0)',
'(3-HSJOIN-0)-(4-TBSCAN-1)',
'(3-HSJOIN-1)-(4-TBSCAN-0)',
'(3-HSJOIN-1)-(4-TBSCAN-1)']

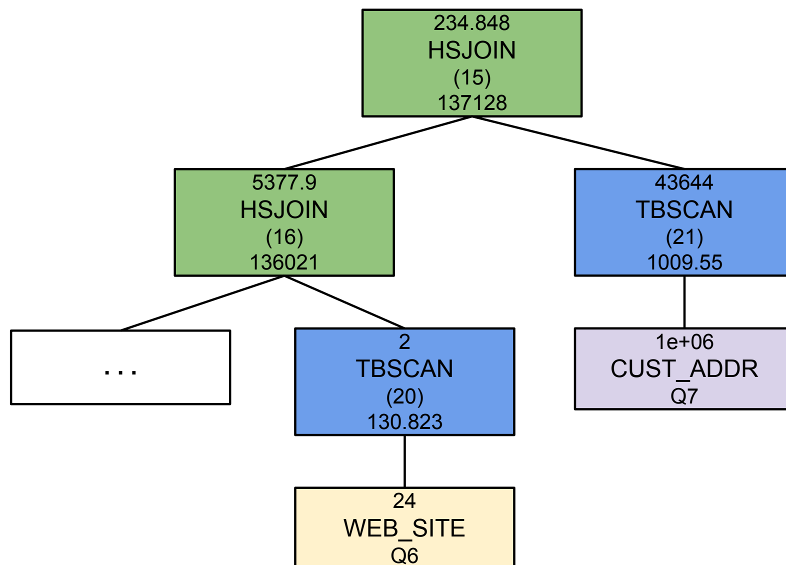
Costs:
[87179.078125, 63270.0, 65207181312.0]

Figure 3.7: Query #8 Template at Node 12

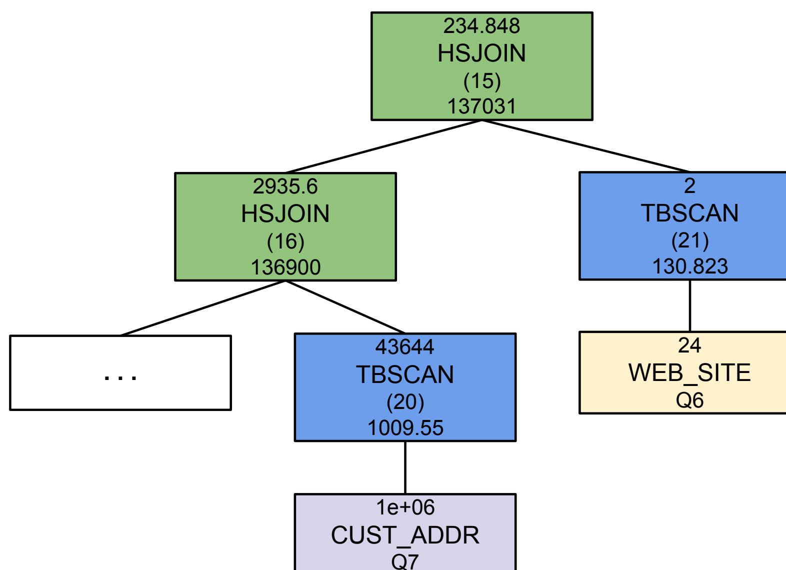


(a) sortheap = 1000, bpsize = 2000 (b) sortheap = 10000, bpsize = 20000

Figure 3.8: TPC-DS Query 98 with two different knob settings for sort heap and bufferpool size



(a) sortheap = 500, bpsize = 1500



(b) sortheap = 5000, bpsize = 6000

Figure 3.9: TPC-DS Query 43 with two different knob settings for both sort heap and bufferpool size.

Chapter 4

System Demonstration

We deploy the synthetic workloads of TPC-DS (with 99 queries) and TPC-H (with 19 queries), and an `IBM` real-client OLAP workload (with 116 queries) up to 100GB.

4.1 Demonstration of Knob Tuning Interface

As an example, in `IBM Db2`, varying the ratio of the size of the sortheap to the bufferpool leads the optimizer to prioritize different QEPs as can be seen in Figures 1.1 and 1.2 for two queries from the TPC-DS benchmark. (The notation $1e6$ for cost denotes 1×10^6.) The colors denote different query plans being chosen. A second example is the joint tuning of the degree of parallelism and sortheap size. As more system operations are performed in parallel, more sort consumers are allocated memory in the sortheap, eventually leading to resource over-allocation.

Figure 4.1 shows the interface for training the DRL model for selecting tuning knobs, queries trained, and the `QEP2Vec` embedding representation. Figure 4.2 illustrates the recommendations for a sample Query #91 from the TPC-DS benchmark.

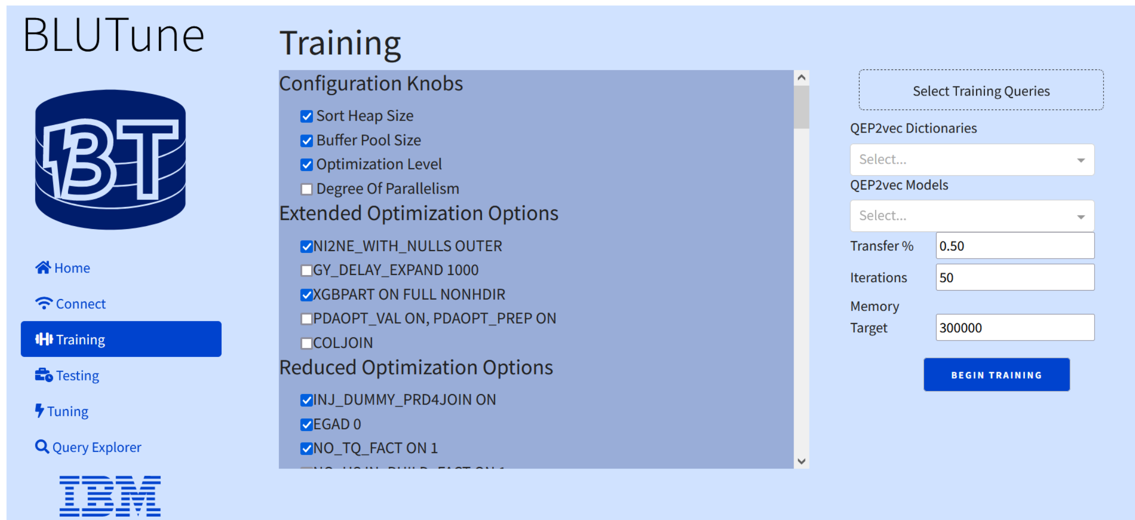


Figure 4.1: BLUTune Training Page.

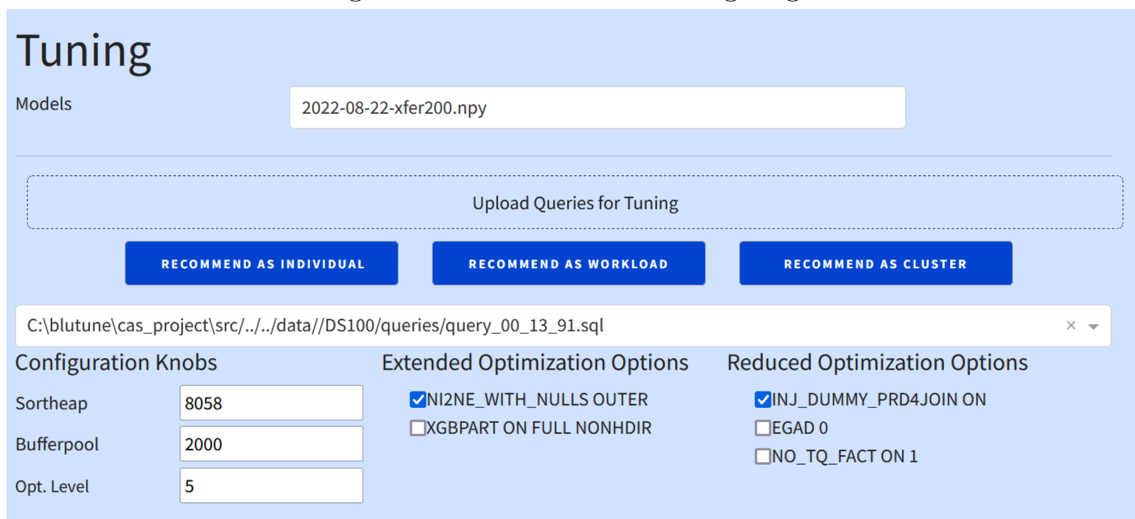


Figure 4.2: BLUTune Tuning Page.

To highlight the challenges of modern workloads, a user can choose a number of non-trivial large, complex queries. A user can then select individual queries or query workload, to obtain recommendations and retrieve an efficient QEP using the capabilities offered by `BLUTune`. Users can visualize the QEPs within `BLUTune`. The queries selected in the TPC-DS benchmark focus on the interesting challenges that promote

performance, including efficient join type and order selection, cost and cardinality estimation, random I/O reads, indexing, and sorting. For instance, for four queries presented to IBM experts, the replacement knobs values found by `BLUTune` resulted in 35% average performance speed-up.

4.2 Query-informed Tuning and Transfer Learning

With the tool, users can compare `BLUTune`'s `QEP2Vec` embeddings against query featurization used by `QTune` [17], denoted as `QTune–featurization`, first at a query level. `BLUTune`'s `QEP2Vec` improves performance over `QTune–featurization` on average by 46% over TPC-DS, 463% over TPC-H, and 448% over IBM real-client workloads. `QTune–featurization` leverages "memorizing" table names and relies on costs as aggregated over an entire query. This results in overfitting, and underperforms on queries with previously unseen schema names during the training (such as `CATALOG_SALES` and `INVENTORY`) and query subplan patterns. This is addressed by `QEP2Vec` via abstraction: table and column names in QEPs are replaced by canonical names, and matching patterns to the 2,3,4-way join sub-plans; Based on `QEP2Vec` embeddings, TPC-DS Query #2 and Query #23 are correctly identified as similar. As seen in Figure 4.4 and Figure 4.5, both of these queries have four `HSJOIN`'s close in costs that are preceded by `TBSCAN`'s, end with `SORT`, `TBSCAN`, and table queue (`TQ`) operators, and also have group by `GRPBY` and `RETURN` operators with similar costs, despite that the involved tables differ. We can also see that the two queries are their closest neighbours within the embedding space in Figure 4.3. In contrast, by `QTune-featurization`, TPC-DS Query #2 and Query #86 are incorrectly classified as similar largely as a result of the same operators and the large overlap in the tables

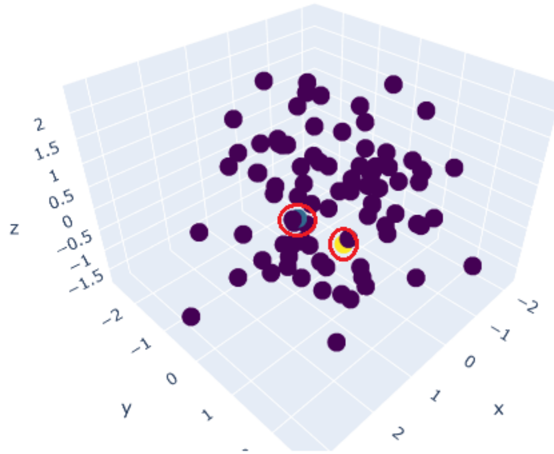


Figure 4.3: Positions of similar queries #2 (blue) and #23 (yellow) in embedding space

between the two queries despite having nearly an order of magnitude of difference between the costs.

With access to both TPC-DS and TPC-H schemas, users can see the versatility of `BLUTune` when being evaluated in new environments. Our performance gain is 21% trained over TPC-DS but tested over TPC-H, and 437% trained over the IBM real-client workload, but tested over TPC-DS.

When tuning for the workload or cluster, `BLUTune` sums up the individual query embeddings and takes their average. We aggregate in a similar fashion for `QTune-featurization`. Participants will be able to observe dramatic performance improvements at the workload and per cluster (comparable to the ones at the query level for the similar reasons).

We can also evaluate the effectiveness of transfer learning. Training a model on the optimizer's estimated cost results in a latency of only 106 seconds. Using transfer learning with fine-tuning based by the execution time results in a latency of 95 seconds, an 11% improvement. Training on execution time only, however—which

is extremely time consuming—results in only a marginal 2% further improvement. It is up to the administrator to decide whether the additional time for training based on fine-tuning by executing queries is worthwhile for the extra performance gain.

4.3 IBM Tools and Resource Constraints

We can also compare the knob configuration `BLUTune` produces against those produced by existing `IBM` tools. `IBM Db2` is shipped with a basic default knob configuration, which is often suitable for simple workload installations. The `IBM Db2 Configuration Advisor` can semi-automatically recommend configurations based on expert specification of the system environment and characteristics, and by using heuristics for system configuration. The `BLUTune` system has an order-of-magnitude improvement over `IBM Db2` default settings. When working with large and complex workloads, it is necessary to tune these knobs to achieve satisfactory performance. `BLUTune` latency runtime is also 38% better than `Configuration Advisor`. Hence, our system can free experts time, while producing more optimized settings.

Data systems are often constrained; e.g., in cloud computing services by the available memory, the # of CPU cores, etc. To demonstrate that our agent learns actions within user-specified constraints effectively, we can select some knobs as an example, such as the `sortheap` and `bpsize`, which each allocate 4KB pages from the system memory heap. Tuning under a memory constraint of 35,000 pages for instance, resulted in a memory allocation of 30,400 pages and a memory constraint of 100,000 pages resulted in a memory allocation of 89,658 pages. These makes sense, as we do not want our agent allocating the entire resource limit, if it offers no benefit over the other actions. We verified this is the case.

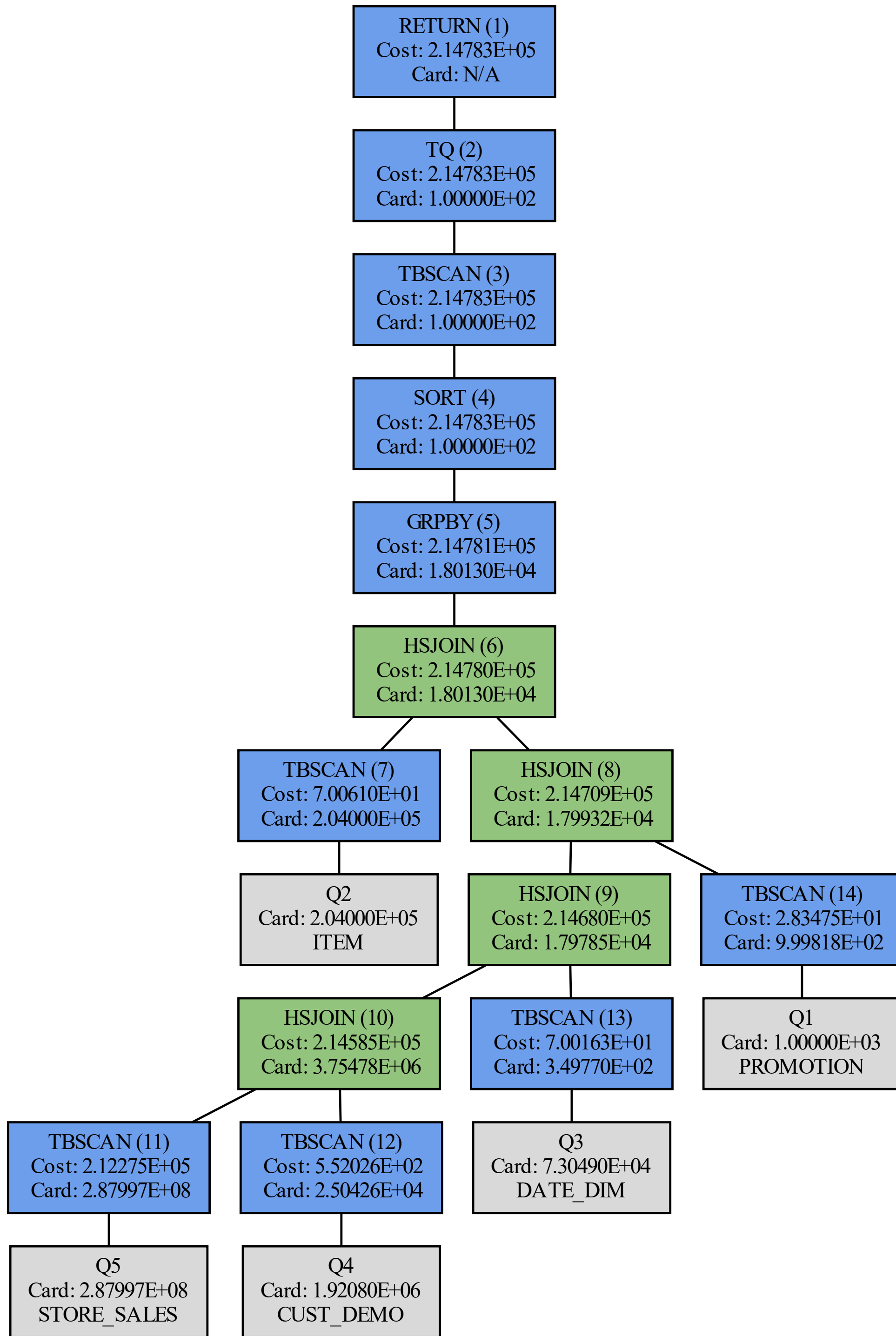


Figure 4.4: Query #2 from Similar Query Pair

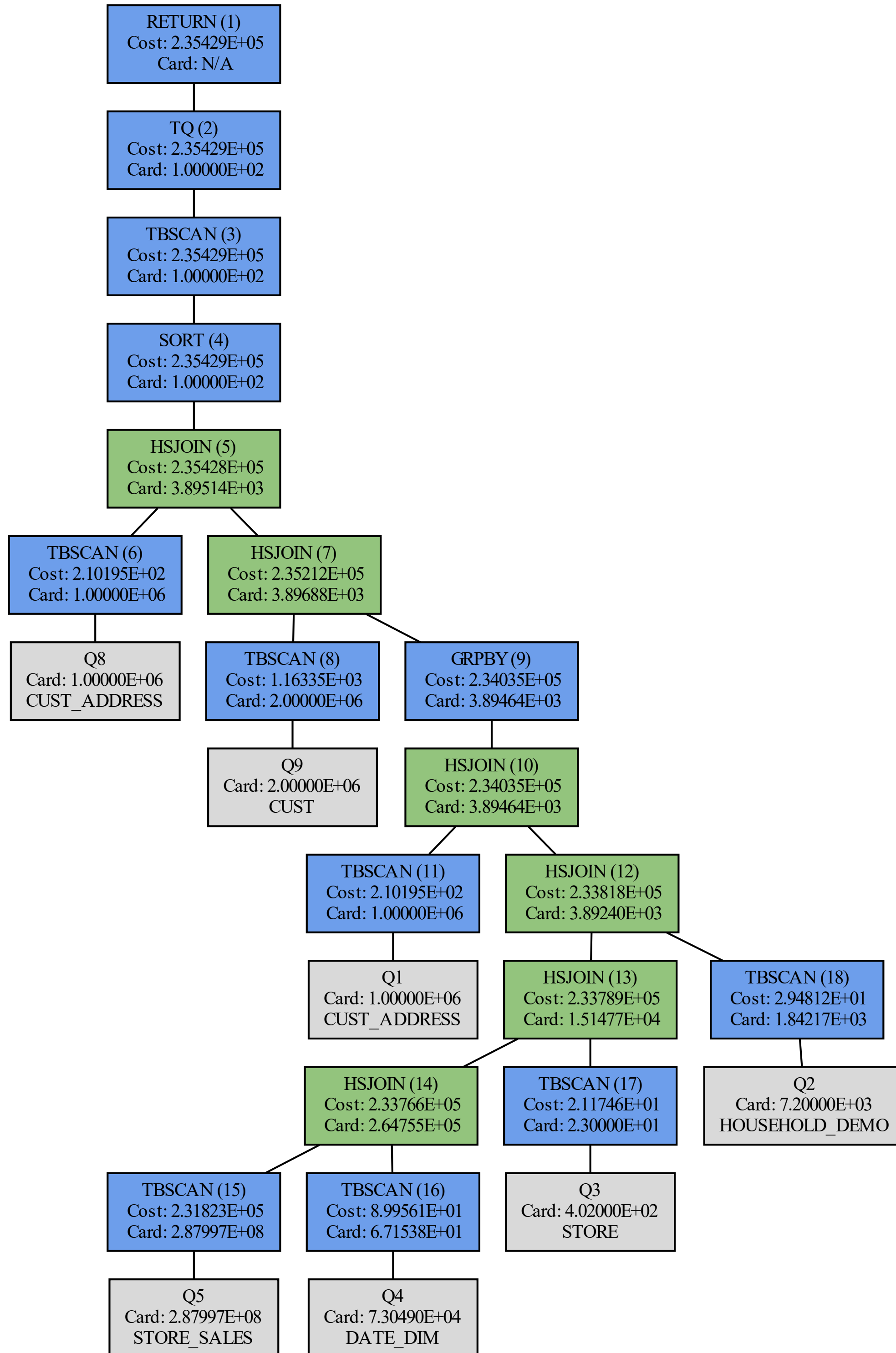


Figure 4.5: Query #23 from Similar Query Pair

Chapter 5

Experimental Evaluation

We present an experimental evaluation of our `BLUTune` system to demonstrate its efficiency, effectiveness, and scalability. We use a server running `IBM Db2` 11.5 which is an 8-core Intel Xeon E5-2630 v3 2.4GHz CPU with 64GB of DDR4 RAM. We conducted the experiments over the synthetic TPC-DS (with 99 OLAP queries) and TPC-H (with 19 queries) benchmarks [7], and a real-world `IBM` client OLAP workload (denoted as *IBM Real*, with 116 queries). Each database is up to 100GB of data. We used a 80/20 split, with 80% of queries for training and 20% for testing, when evaluating over the same workload. `BLUTune` is written in Python, and employs the *PyTorch* package to facilitate ML.

5.1 TPC-DS Benchmark

The TPC-DS benchmark (Transaction Processing Performance Council Decision Support) is a standard benchmark designed to measure the performance of decision support systems (DSS). DSS are systems that are used for analyzing large volumes of

data to help with business decision-making. The schema for the TPC-DS benchmark is quite large and complex, but an example of the relationships established in the fact table for the *store_sales* table can be seen in figure 5.1

The TPC-DS benchmark involves running a series of SQL queries against a database, simulating a typical DSS workload. The benchmark consists of 99 queries in total, which are designed to test various aspects of the system's performance, including:

- Complex queries involving multiple joins, sub-queries, and aggregation functions
- Queries with a high degree of parallelism and large data sets
- Queries that require complex calculations and transformations of data
- Queries that simulate real-world scenarios, such as product sales and customer orders

The benchmark is designed to measure both query throughput (the number of queries that can be processed per unit time) and query response time (the time taken to execute a single query). The TPC-DS benchmark is widely used in the industry to compare the performance of different database systems and hardware configurations for decision support workloads. However, it is important to note that the benchmark may not always accurately reflect the performance of a system in real-world scenarios, so it should be used as a guideline rather than a ground truth for performance.

5.1.1 TPC-H Benchmark

The TPC-H benchmark is another widely used benchmark in the field of data warehousing and decision support. The TPC-DS and TPC-H benchmarks are both de-

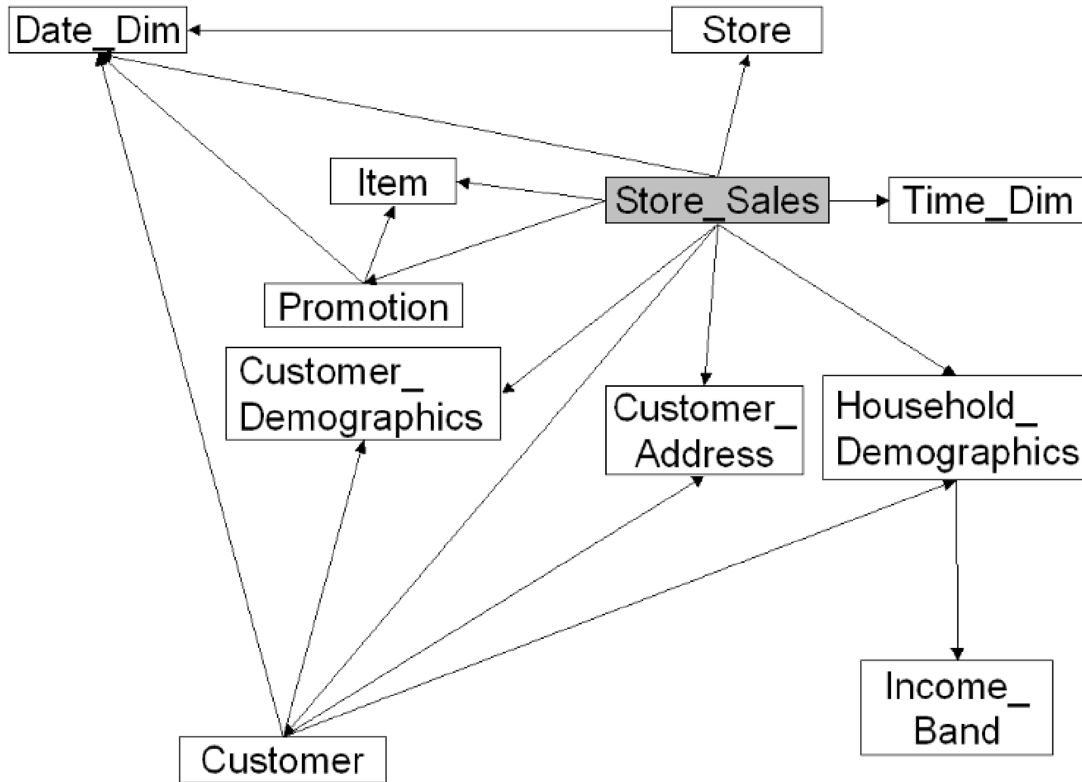


Figure 5.1: TPC-DS Fact Table from the TPC-DS Specification

signed to measure the performance of database systems when executing decision support queries, but they have some differences in terms of their objectives and design. TPC-H is designed to measure the performance of a database system when executing complex ad-hoc queries typical of decision support and business intelligence workloads. The schema for the TPC-H benchmark can be seen in figure 5.2.

The benchmark consists of a set of 22 queries that model a typical data warehousing scenario involving complex queries that analyze large volumes of data. The queries are divided into different groups based on their complexity, with the simplest ones involving a single table and the most complex ones involving many tables and

subqueries.

The TPC-H benchmark is designed to test various aspects of the system's performance, including:

- Complex query processing involving multiple joins, aggregations, subqueries, and data transformations
- The ability to handle large data volumes
- The ability to execute queries with a high degree of parallelism
- The ability to optimize queries using indexing, query rewriting, and other techniques

One key difference between the two benchmarks is the number and type of queries they include. TPC-DS has a larger number of queries, including more complex data mining and statistical analysis queries, while TPC-H has a smaller set of ad-hoc queries. Another difference is the way the results are reported, with TPC-DS reporting query throughput as well as query response time.

5.2 Query-informed Tuning

Exp-1: Query Level Tuning. We first evaluate our query-informed tuning `BLUTune` system at the *query* level. We compare `BLUTune`'s `QEP2Vec` embeddings against query featurization used by `QTune`, denoted as `QTune-featurization` (described in Section 3.2). We observe dramatic performance improvements, as reported in Figures 5.3 and 5.4 over workloads with 100GB of data.

BLUTune's QEP2Vec improves performance over QTune-featurization by 46% over TPC-DS, 463% over TPC-H, and 448% over IBM Real workloads. Performance for *each* query improved; this is shown in the zoom-in view in Figure 5.6 for the TPC-DS benchmark. QTune-featurization leverages "memorizing" table names and relies on costs as aggregated over an entire query. This results in overfitting, and under-performs on queries with previously unseen schema names during the training (such as CATALOG_SALES, INVENTORY and CATALOG_RETURNS in TPC-DS) and unseen query subplan patterns. This is addressed by QEP2Vec via abstraction; table and column names in QEPs are replaced by canonical names, and matching patterns is to the 2,3,4-way join sub-plans.

Also, our performance gain is 21% when trained over TPC-DS but tested over TPC-H,[1] and 437% when trained over IBM Real, but tested over TPC-DS. This illustrates the versatility of BLUTune when operating in unfamiliar environments.

Exp-2: Query Workload & Cluster Level Tuning. When operating to tune for the workload or per cluster, BLUTune sums up the individual query embeddings and takes their average to obtain the semantic equivalent of the workload / cluster encoding. We aggregate in a similar fashion for QTune-featurization. (This is quite standard, and is how QTune was evaluated for clusters in [17].)

The average performance improvement of BLUTune's QEP2Vec over QTune-featurization when tuned for the entire workload or by cluster, shown in Figures 5.4 and 5.5, respectively, is again significant, with improvements of 10% and 8% over TPC-DS, 470% and 77% over TPC-H, 335% and 360% over IBM Real, 51% and 29% when TPC-DS trained and TPC-H tested, aswell as 263% and 290% when IBM Real

[1] This is denoted with an arrow; e.g., as TPC-DS → TPC-H.

trained and TPC-DS tested. The reasons for these performance improvements are similar to those for Exp-1.

5.3 Transfer Learning

Exp-3: Fine-tuning performance. To evaluate the effectiveness of transfer learning for knobs tuning, we further divide the 80% of queries used for training from the TPC-DS workload with 100GB into a 70/30 split with 70% used for training by estimated cost, and remaining 30% for training by execution time.

The cumulative runtime of the testing queries is reported in Figure 5.7. Training a model based on the optimizer's estimated cost only leads to a latency of 106 secs. Using transfer learning with fine-tuning based on the execution time (denoted as Cost + Execution Time) can offer some further gain. By using transfer learning with fine-tuning based on the execution time leads to a latency of 95 secs, an 11% improvement. Training based on execution time only, however, which is extremely time consuming (Exp-4), results in only a marginal 2% further improvement.

This demonstrates that training solely by the optimizer's estimated costs works remarkably well, with a significant speed up for training. This is also nearly independent of the data size, and so it "scales" with size. Employing transfer learning in this scenario does offer some benefit (e.g., the seen 11% improvement for TPC-DS / 100GB). In scenarios like this, it is up to the administrator to decide whether the additional time for training based on fine-tuning by executing queries (Exp-4) is worthwhile for extra performance gain.

Exp-4: Scalability. `BLUTune` is designed to tune knobs for complex and large workloads. To demonstrate how our approach scales with the size of data, we generate

three versions of the synthetic TPC-DS benchmark with different sizes: 1GB, 10GB, and 100GB. The resulting training times of both the cost-only phase and the fine-tuning phase based on execution time are shown in Figure 5.8.

Across the sizes, the time spent on the first phase of training solely to minimize estimated cost unsurprisingly is similar, as it is based on data statistics. The estimates are generated by the optimizer in milliseconds. The training's main bottleneck is the time needed to apply new knob configurations; e.g. resizing a memory pool. Discrepancies in timing are also due to randomness, and certain convergence conditions, leading to earlier termination. Training on the cost estimates allows our agent to learn quickly a suitable configuration, regardless of the workload size.

The better the learned configuration is from the cost estimates, the less time the agent spends on the fine-tuning phase involving query execution. As the size of the data increases, execution time of queries grows with it. Thus, fine-tuning by using transfer learning is dependent on the data size. The training time scales sub-linearly, however; e.g., increasing the size by a factor of ten from 10GB to 100GB results in only a 2.35 increase in training time. Thus, our `BLUTune` system scales well to large workloads. In contrast, training fully based on the execution time is extremely expensive, and is not reported in Fig. 5.8, as it takes around 2,623 miutes (44 hours).

5.4 Effectiveness

Exp-5: IBM Tools. We compare the knob configuration `BLUTune` produces against those produced by existing `IBM` tools over the TPC-DS benchmark with 100GB. `IBM Db2` is shipped with a basic default knob configuration, which is often suitable for simple workload installations. The `IBM Db2 Configuration Advisor` can semi-

automatically recommend configurations based on expert specification of the system environment and characteristics, and by using heuristics for system configuration (see details in Sec. 6).

We report the results in Figure 5.9. The `BLUTune` system has an order-of-magnitude improvement over `IBM Db2` default settings. This illustrates than when working with large and complex workloads, it is necessary to tune these knobs to achieve satisfactory performance. `BLUTune` latency runtime is also 38% better than `Configuration Advisor`. Hence, our system can free experts time, while producing more optimized settings.

Exp-6: Reward. We evaluate our `BLUTune`'s reward function against `QTune`'s reward, as both systems use a deep reinforcement learning approach. We found that `QTune`-reward function did not perform well. It is, on average, 56% slower under the TPC-DS workload with 100GB, as reported in Fig. 5.9 (and 48% slower under the IBM Real workload). During both training on cost and execution time, the magnitudes of the reward were high, leading to high gradients. `QTune`'s reward was originally evaluated over workloads with simpler queries and smaller data [17]. Given that TPC-H queries are simpler than the other workloads tested, this is seen in `QTune`-reward more competitive performance (11% slower) over TPC-H.

We attempted to address this by modifying their reward function to reduce the magnitude and variance. However, this still did not act as an effective reward signal. The agent had trouble converging as `QTune`'s reward is only based on the performance change compared against that at initial time and the previous time. `BLUTune`'s reward function, on the other hand, consists of the performance history, using exponential decay and best performance. The lack of convergence led to a training time of 24.5

hours, compared to `BLUTune`'s 7.5 hours. Thus, the `QTune` approach does not extend well to complex OLAP workloads, which is the key focus of `BLUTune`.

Exp-7: Resources. Data systems are often constrained, e.g., in cloud computing services by the available memory and the # of CPU cores, etc. Our agent learns actions within user-specified resource constraints via our resource reward function. To demonstrate the effectiveness of user-specified resource constraints, we select the `sortheap` and `bpsize` knobs as an example, which each allocate 4KB pages from the system memory heap. Fig. 5.10 shows the resulting combined `sortheap` and `bpsize` usage for the same set of queries with two different resource limits: 35,000 and 100,000 4KB pages.

Training under a memory constraint of 35,000 pages resulted in an average memory allocation of 30,400 pages A memory constraint of 100,000 pages resulted in an average memory allocation of 89,658 pages. These results makes sense, as we do not want our agent allocating the entire resource limit if it offers little or no benefit over the other actions. We verified this is the case. This demonstrates `BLUTune`'s effectiveness for tuning within resource constraints.

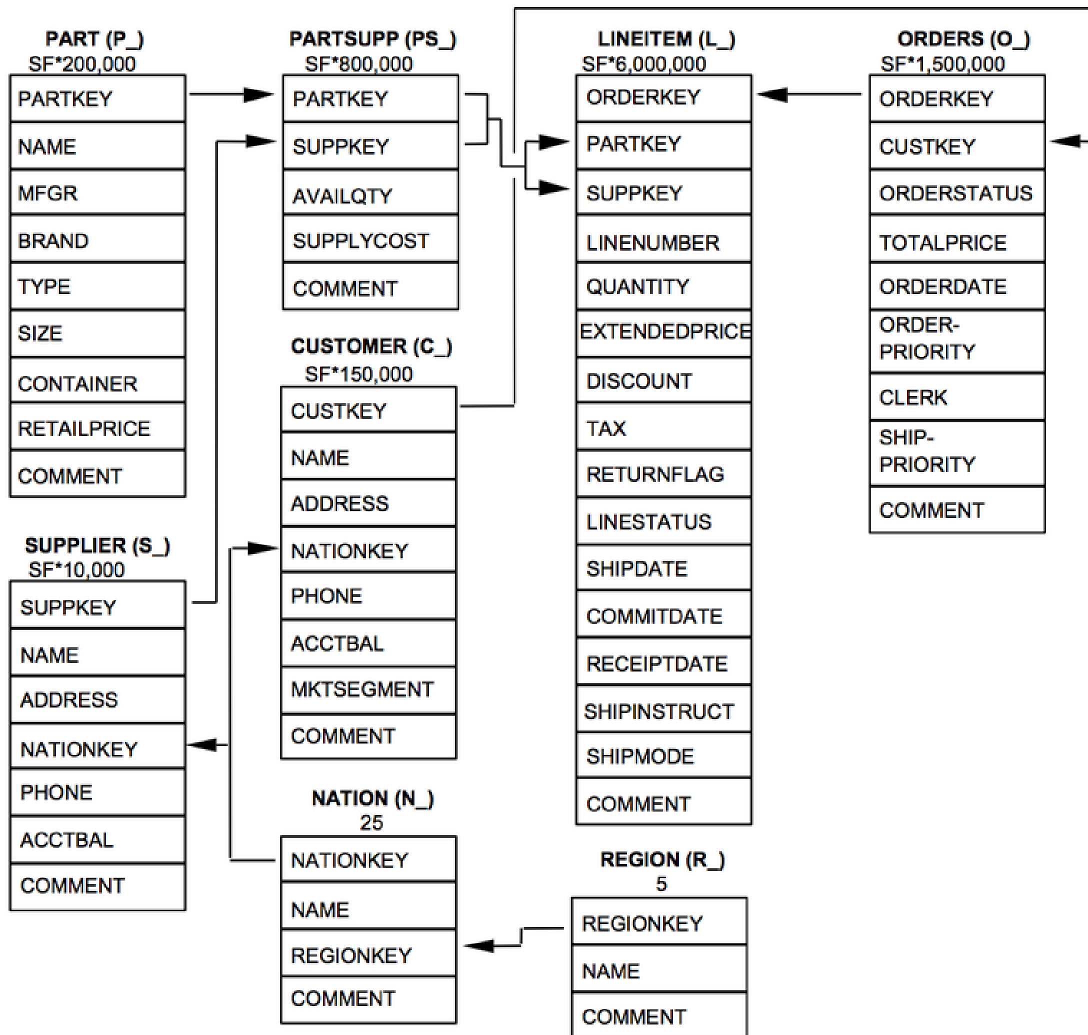


Figure 5.2: TPC-H Schema Design from TPC-H Specification

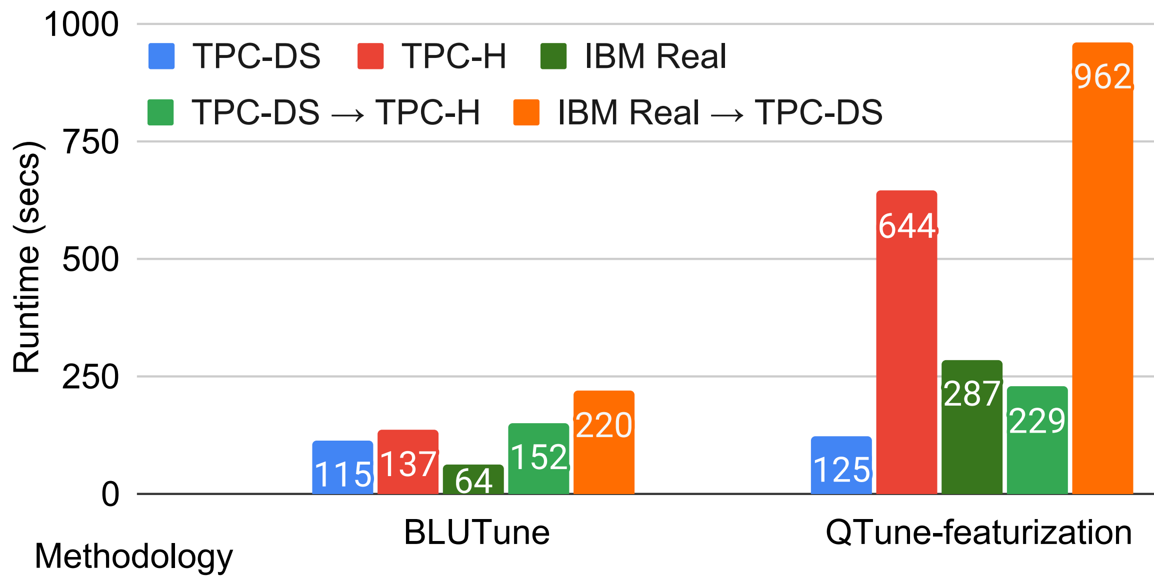


Figure 5.3: Query level tuning

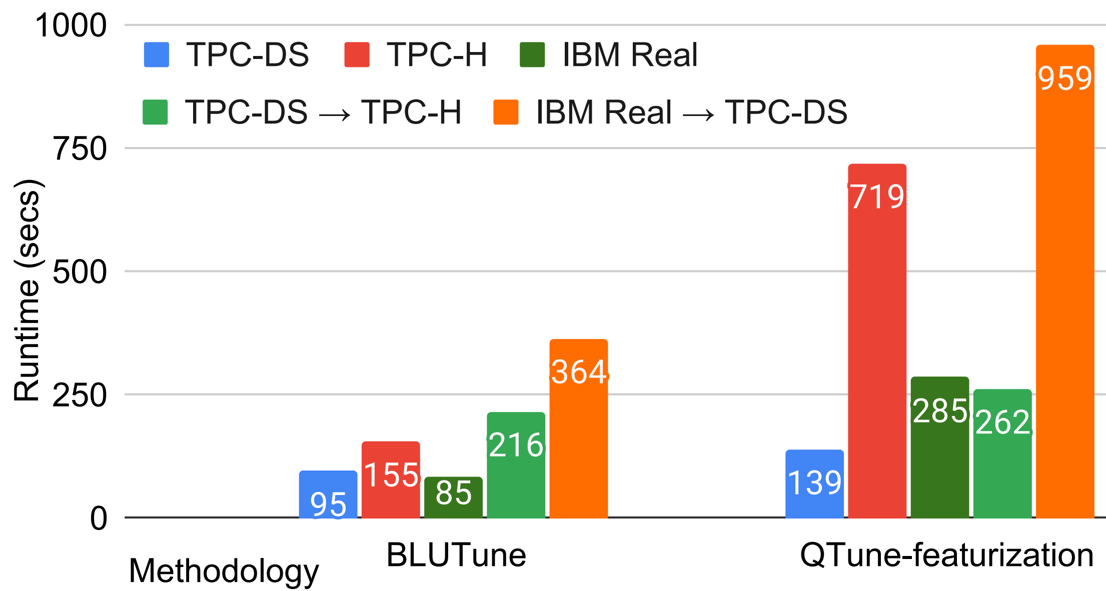


Figure 5.4: Workload level tuning

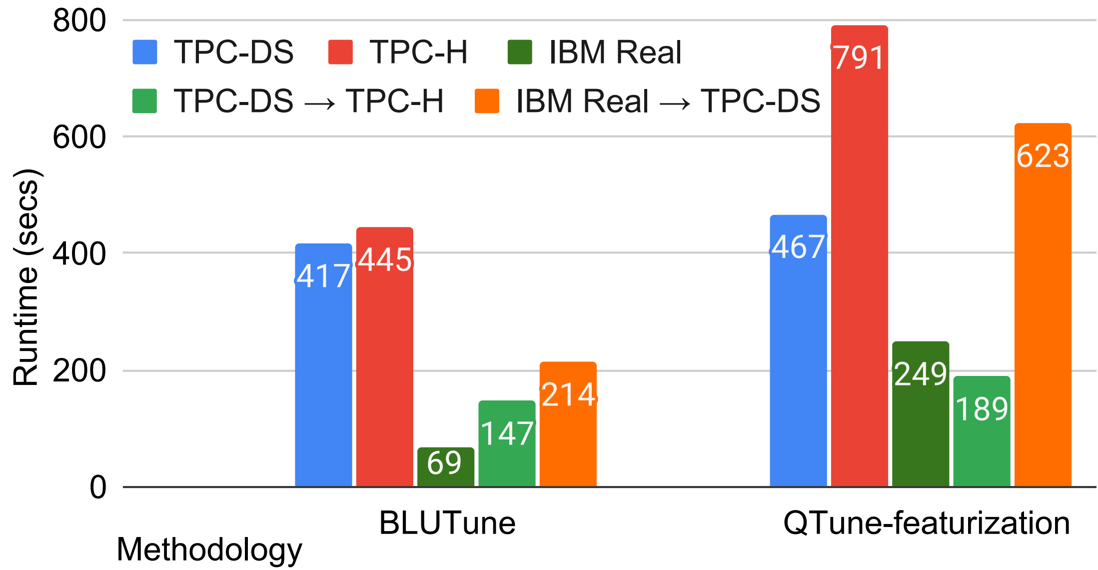


Figure 5.5: Cluster level tuning.

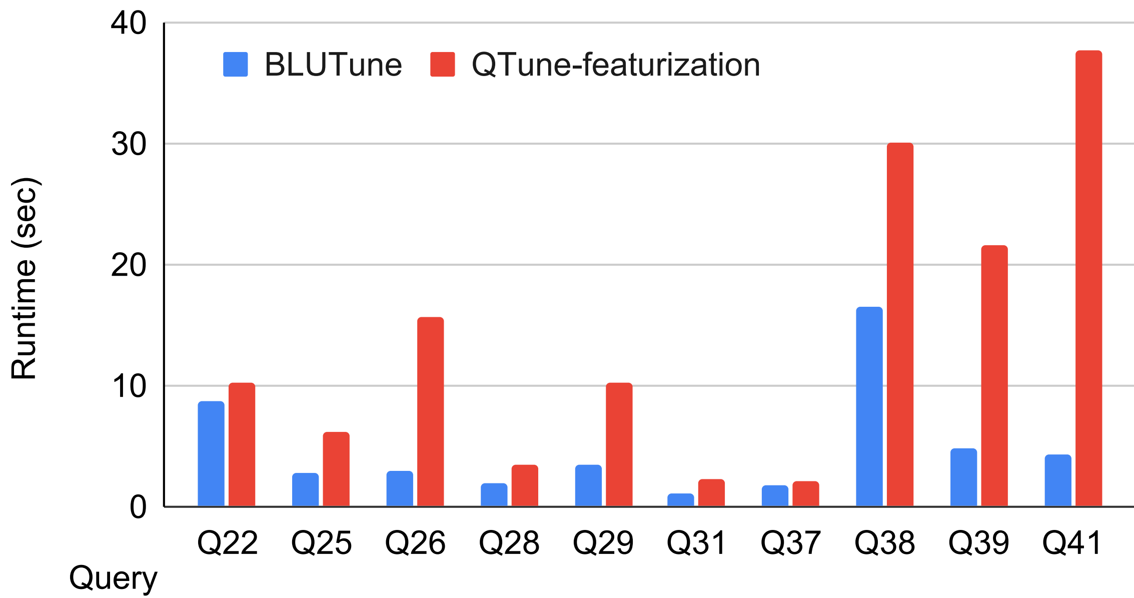


Figure 5.6: Query level tuning detailed on TPC-DS.

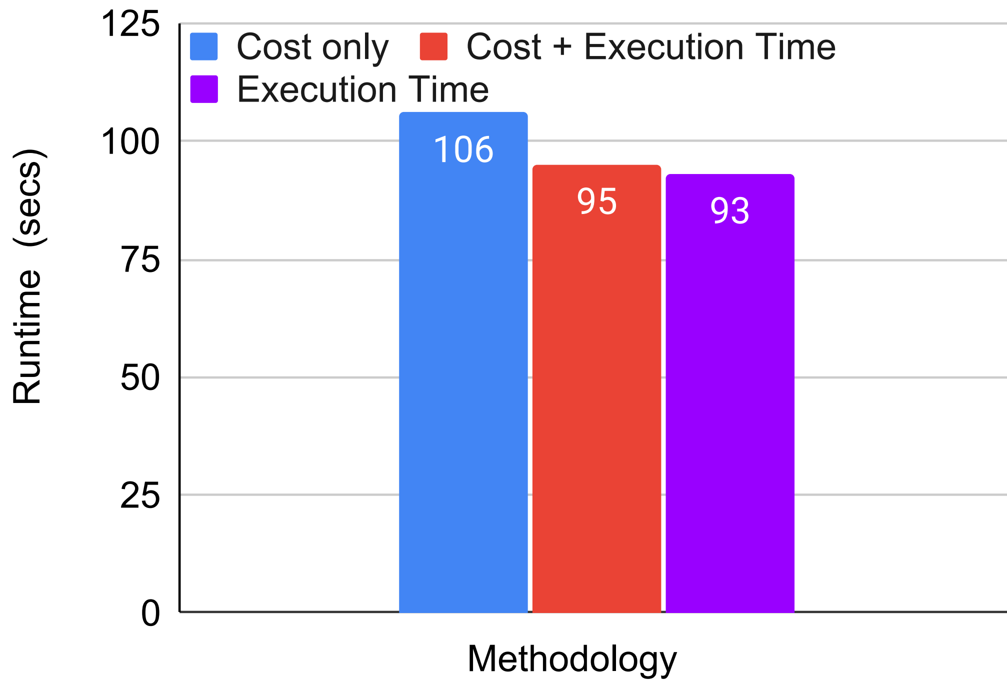


Figure 5.7: Fine-tuning performance.

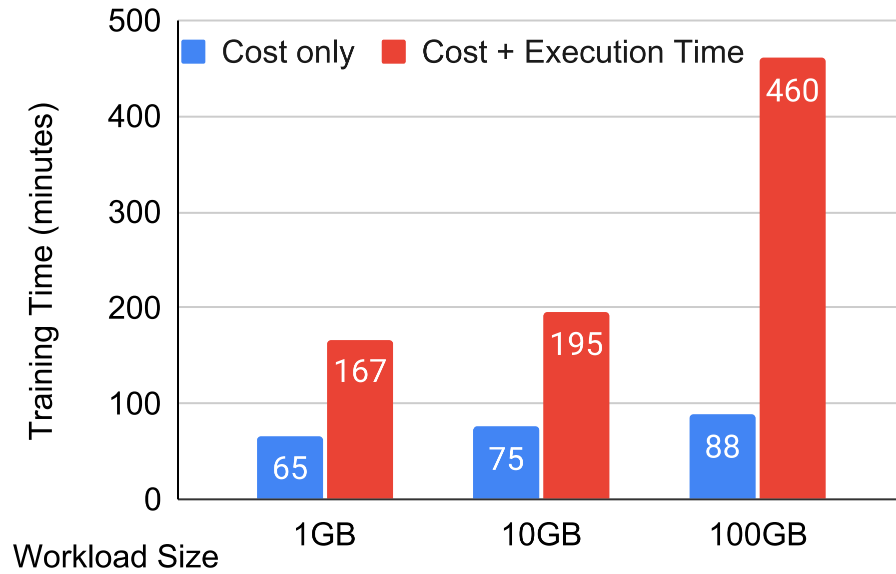


Figure 5.8: Scalability study.

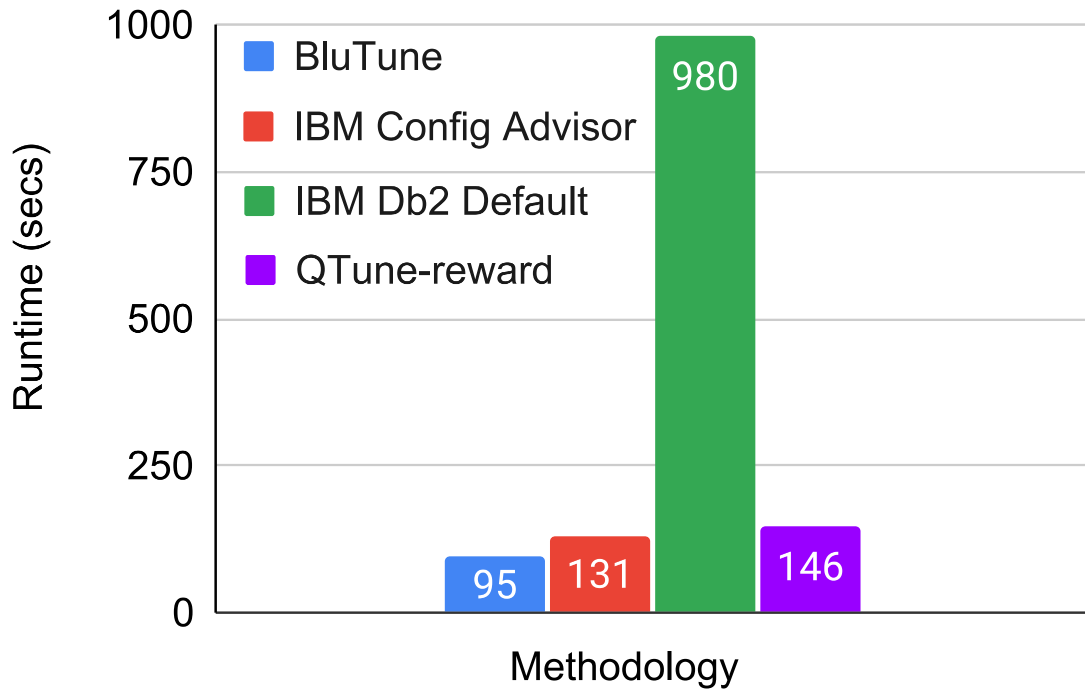


Figure 5.9: IBM tools & QTune-award.

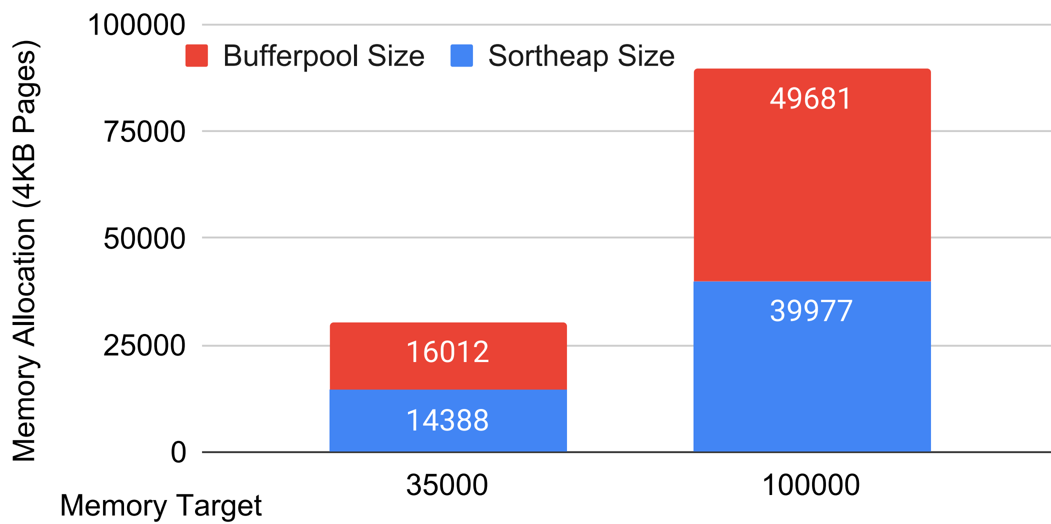


Figure 5.10: Resource constraint for memory allocation.

Chapter 6

Related Work

Automatic knobs tuning has been an interest of researchers and vendors for the past decades [24]. IBM has a tool called the Configuration Advisor, which allows administrator to obtain recommended settings for common IBM DB2 configuration parameters [1]. The tool requires three sets of information: (1) user specification of the system environment (e.g. type of workload, number of statements and memory percentage), (2) expected system characteristics (e.g. number of CPUs, amount of RAM, number and size of tables) and finally (3) expert specified heuristics for system configurations. Although it provides a sensible starting point for performance tuning, it is limited mostly due to the requirement of human input. In Db2 version 9.1, IBM introduced the Self-Tuning Memory Manager (STMM), which provides adaptive self tuning of database memory allocation through heuristics [25]. However, STMM was not designed with column-organized tables in mind.

Similar to IBM's offerings, Oracle developed the Automatic Database Diagnostic Monitor which diagnoses performance issues to recommend tuning actions [26]. This tool recommends actions for a small set of configuration settings based

on heuristics. Oracle later developed the `SQL Tuning Advisor`, which recommends statement-specific tuning actions to improve the performance [2], [27].

The `iTuned` system is a tuning tool that executes planned experiments through a cycle-stealing paradigm to gather performance data and statistics, while the data system is not being fully utilized [28]. A Gaussian process model is used to approximate the performance surface from the set of experiments, providing a way to produce knob recommendations with positive expected performance impact. It requires training from scratch for each different workload, and can take tens of hours to find a suitable configuration.

`OtterTune`, a more modern configuration tuning tool, first identifies and ranks the knobs which have the strongest impact on the performance, maps the given workload to a repository of previously collected performance measurements, and utilizes a Gaussian process model similar to `iTuned` to choose knob settings [29], [30]. `OtterTune` requires a large corpus of previous tuning sessions in order to leverage knowledge it has gained in the past.

`BestConfig` utilizes a divide-and-diverge sampling method to represent the action space and the recursive bound-and-search algorithm to recommend configurations [16]. `BestConfig` does not learn a model, nor store information from previous tuning sessions, meaning it must be ran from scratch for each workload.

`CDBTune` was the first tool to adopt deep reinforcement learning for the task of knob tuning [10]. `CDBTune` utilizes deep deterministic policy gradient (DDPG), a policy-based reinforcement learning method. DDPG is an off-policy algorithm that is largely intended only for continuous action spaces and requires a large replay buffer of past experiences. `CDBTune` does not consider the queries themselves and only acts

to the anticipated change to data system health metrics, such as counters for pages read from or written to disk.

Similar to `CDBTune`, `QTune` also uses DDPG, but places an emphasis on featurizing queries using their QEP [17]. For a query, statement details, table involvement, and cost information about each operator is extracted from QEP and represented as a vector. The overall performance of the agent is greatly tied to the simplistic reward function (also used in `CDBTune`) and featurization of the QEPs, which does not capture well the structure of queries.

`Gur` *et al.* propose a multi-model tuning solution utilizing DDPG to address deployments that consist of multiple, different workloads [31]. A downside of this is that scenarios where many different models are needed is possible, which can make this a costly approach.

Chapter 7

Conclusion

7.1 Conclusion

In this work, we created a system, BLUTune, to solve the problem of automatic knob tuning for IBM DB2. Unlike existing IBM tools which largely use heuristics to recommend knob configurations, BLUTune utilizes Deep Reinforcement Learning to directly learn the impact knobs have on the performance of queries in order to suggest knob configurations that maximize the performance for a given workload. We specifically used Advantage Actor-Critic, where our implementation can handle both continuous and discrete knobs simultaneously in one single model. Queries are represented by their query execution plan operators and costs which provide insight into how a query behaves and also capture the effects of various knobs. The custom reward function for our agent allows for fast convergence and also enforces system resource constraints such as limited memory. BLUTune has a novel training strategy utilizing the concept of transfer learning. The agent first trains on the task of minimizing the query optimizer's estimated cost for various queries, which allows for thousands of

different knob configurations to be considered in a very short amount of time. We continue training the same model but on a different task of minimizing execution time. This second phase captures knobs that are not considered by the cost estimates and allow for fine-tuning of the original model to improve its effectiveness. In our experimental evaluation, we demonstrated the effectiveness of our transfer learning approach. When tuning knobs that are considered by cost estimates, the cost-only phase can find a suitable configuration without even needing to train on minimizing execution time, although doing so nets a small performance boost. Introducing a knob not reflected by the optimizer highlights the importance of the execution time phase, but by employing transfer learning we greatly reduce the training time as opposed to training on minimizing execution time alone. Our approach allows for BLUTune to scale to large databases while producing knob configurations that are better than that of IBM's existing tools. BLUTune serves as a strong foundation for a truly fully automated tuning solution for IBM DB2.

7.2 Future Work

`BLUTune` is designed to aid DBAs when a problem in query/workload performance arises; it would be beneficial if `BLUTune` was capable of anticipating changes in the workload and taking proactive measures to tune the database. In other systems, certain patterns, such as cycles, growth spikes, and workload evolution, have been identified in business query workloads and applications. To address this, a query forecaster has been proposed that employs recurrent neural networks, specifically long short-term memory modules. We believe that a similar approach could enhance the efficiency of BLUTune as a fully automated knob-tuning system.

The training process of BLUTune is significantly extended as query execution during the learning approach takes a lot of time. To expedite the training process, several components of BLUTune can be parallelized. Currently, BLUTune processes queries one by one during training. The order in which queries are encountered and learned from does not have a significant impact. Asynchronous Advantage Actor-Critic (A3C) is an extension of A2C developed by Google DeepMind. A2C has only one agent that interacts with the environment, whereas A3C allows for concurrent support of multiple agents to accelerate training.

References

[1] E. Kwan, S. Lightstone, K. B. Schiefer, A. J. Storm, and L. Wu, "Automatic database configuration for DB2 universal database: Compressing years of performance expertise into seconds of execution," in *Datenbanksysteme für Business, Technologie und Web, BTW-Konferenz*, 2003, pp. 620–629.

[2] P. Belknap, B. Dageville, K. Dias, and K. Yagoub, "Self-tuning for sql performance in oracle database 11g," in *ICDE*, 2009, pp. 1694–1700.

[3] I. D. Documentation, *Configuration parameters summary*, https://www.ibm.com/docs/en/db2/11.5?topic=parameters-configuration-summary, Accessed: 2022-05-01, 2022.

[4] I. D. Documentation, *Db2 registry and environment variables*, https://www.ibm.com/docs/en/db2/11.5?topic=variables-registry-environment, Accessed: 2021-10-04, 2022.

[5] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback, "Self-tuning database technology and information services: From wishful thinking to viable engineering," in *VLDB*, 2002, pp. 20–31.

[6] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer, "Using probabilistic reasoning to automate software tuning," in *SIGMETRICS*, 2004, pp. 404–405.

[7] M. Pöss, R. O. Nambiar, and D. Walrath, "Why you should run TPC-DS: A workload analysis," in *VLDB*, 2007, pp. 1138–1149.

[8] C. Henderson, V. Corvinelli, P. Godfrey, P. Mierzejewski, J. Szlichta, and C. Zuzarte, "Blutune: Tuning up ibm db2 with ml," in *Proceedings of the 39th IEEE International Conference on Data Engineering*, ser. ICDE '23, Los Angeles, CA, USA: Institute of Electrical and Electronics Engineers, 2023. DOI: `10.1145/3511808.3557117`. [Online]. Available: `https://doi.org/10.1145/3511808.3557117`.

[9] C. Henderson, S. Bryson, V. Corvinelli, *et al.*, "Blutune: Query-informed multi-stage ibm db2 tuning via ml," in *Proceedings of the 31st ACM International Conference on Information amp; Knowledge Management*, ser. CIKM '22, Atlanta, GA, USA: Association for Computing Machinery, 2022, pp. 3162–3171, ISBN: 9781450392365. DOI: `10.1145/3511808.3557117`. [Online]. Available: `https://doi.org/10.1145/3511808.3557117`.

[10] J. Zhang, Y. Liu, K. Zhou, *et al.*, "An end-to-end automatic cloud database tuning system using deep reinforcement learning," in *SIGMOD*, 2019, pp. 415–432.

[11] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[12] G. Damasio, P. Mierzejewski, J. Szlichta, and C. Zuzarte, "Query performance problem determination with knowledge base in semantic web system optimatch," in *EDBT*, 2016, pp. 515–526.

[13] G. Damasio, P. Mierzejewski, J. Szlichta, and C. Zuzarte, “Optimatch: Semantic web system for query problem determination,” in *ICDE*, 2016, pp. 1334–1337.

[14] G. Damasio, V. Corvinelli, P. Godfrey, *et al.*, “Guided automated learning for query workload re-optimization,” *PVLDB*, vol. 12, no. 12, pp. 2010–2021, 2019.

[15] G. Damasio, S. Bryson, V. Corvinelli, *et al.*, “GALO: guided automated learning for re-optimization,” *PVLDB*, vol. 12, no. 12, pp. 1778–1781, 2019.

[16] Y. Zhu, J. Liu, M. Guo, *et al.*, “Bestconfig: Tapping the performance potential of systems via automatic configuration tuning,” in *SoCC*, 2017, pp. 338–350.

[17] G. Li, X. Zhou, S. Li, and B. Gao, “Qtune: A query-aware database tuning system with deep reinforcement learning,” *PVLDB*, vol. 12, no. 12, pp. 2118–2130, 2019.

[18] Q. V. Le and T. Mikolov, “Distributed representations of sentences and documents,” in *ICML*, 2014, pp. 1188–1196.

[19] *The sql and xquery compiler process*, `https://www.ibm.com/docs/en/db2/11.5?topic=optimization-sql-xquery-compiler-process`, Accessed: 2022-12-20.

[20] *Query rewriting methods and examples*, `https://www.ibm.com/docs/en/db2/11.5?topic=process-query-rewriting-methods-examples`, Accessed: 2022-12-21.

[21] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann, “Cardinality estimation done right: Index-based join sampling,” in *CIDR*, 2017.

[22] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, "Access path selection in a relational database management system," in *SIGMOD*, 1979, pp. 23–34.

[23] A. Mihaylov, V. Corvinelli, P. Godfrey, P. Mierzejewski, J. Szlichta, and C. Zuzarte, "Scalable learning to troubleshoot query performance problems," in *CIKM*, 2021, pp. 4016–4025.

[24] D. V. Aken, D. Yang, S. Brillard, *et al.*, "An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems," *PVLDB*, vol. 14, no. 7, pp. 1241–1253, 2021.

[25] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra, "Adaptive self-tuning memory in db2," in *VLDB*, 2006.

[26] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood, "Automatic performance diagnosis and tuning in oracle," in *CIDR*, 2005, pp. 84–94.

[27] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin, "Automatic SQL tuning in oracle 10g," in *VLDB*, 2004, pp. 1098–1109.

[28] S. Duan, V. Thummala, and S. Babu, "Tuning database configuration parameters with ituned," *Proc. VLDB Endow.*, pp. 1246–1257, 2009.

[29] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *SIGMOD*, 2017, pp. 1009–1024.

[30] B. Zhang, D. V. Aken, J. Wang, *et al.*, "A demonstration of the ottertune automatic database management system tuning service," *PVLDB*, vol. 11, no. 12, pp. 1910–1913, 2018.

[31] Y. Gur, D. Yang, F. Stalschus, and B. Reinwald, “Adaptive multi-model reinforcement learning for online database tuning,” in *EDBT*, 2021, pp. 439–444.