Automatic Knobs-Tuning for DB2 using Deep Reinforcement Learning

by

Spencer C. Bryson

A thesis submitted to the
School of Graduate and Postdoctoral Studies in partial
fulfillment of the requirements for the degree of

Master of Science in Computer Science

Faculty of Science
University of Ontario Institute of Technology (Ontario Tech University)
Oshawa, Ontario, Canada

December 2021

© Spencer C. Bryson, 2021

Thesis Examination Information

Submitted by: **Spencer C. Bryson**

Master of Science in **Computer Science**

Thesis title: Automatic Knobs-Tuning for DB2 using Deep Reinforcement Learning

An oral defense of this thesis took place on December 3, 2021 in front of the following examining committee:

Examining Committee:

Chair of Examining Committee	Dr. Shahram S. Heydari
Research Supervisor	Dr. Jarek Szlichta
Examining Committee Member	Dr. Heidar Davoudi
Thesis Examiner	Dr. Ying Zhu

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

Abstract

Modern database management systems have hundreds of different configuration parameters (knobs) that control various aspects of how they behave and perform. These knobs must be properly tuned in order to maximize the performance of the database for a given query workload. Traditionally, database administrators would be responsible for database performance tuning. However, manual configuration tuning is a difficult process for humans, as there are hundreds of different inter-dependent knobs to be tuned. Different queries and workloads also benefit from configurations differently, there is no one single database configuration that can fit all scenarios. We propose BLUTune, a system to automatically produce effective knob configuration for IBM DB2. BLUTune utilizes deep reinforcement learning and features a unique transfer-learning approach to training which allows for fast learning. In experimental validation, BLUTune demonstrates its capability of producing effective configurations across differing sizes of the TPC-DS OLAP benchmark in a timely manner.

Keywords: Database tuning; knob tuning; deep reinforcement learning; DB2

Author's Declaration

I hereby declare that this thesis consists of original work of which I have authored. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize the University of Ontario Institute of Technology (Ontario Tech University) to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize University of Ontario Institute of Technology (Ontario Tech University) to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.

______________________________ **Spencer C. Bryson**

Statement of Contributions

This work was done under supervision of Dr. Jarek Szlichta in collaboration with the IBM Centre for Advanced Studies [1] and Dr. Parke Godfrey from York University [2].

Collaborators from IBM provided critical guidance regarding DB2, their feedback helped shape the objectives and design decisions made throughout this work. Dr. Parke Godfrey also provided orientation, guidance and feedback throughout the development process. The development of the system was done in whole by myself.

This work is subject to publication at a future date but is not currently submitted, and the conference remains unknown.

[1] https://www.research.ibm.com/university/cas/

[2] http://www.cse.yorku.ca/~godfrey/

Acknowledgements

To begin, I would like to acknowledge my parents Shawn and Stephanie, my brother Brandon, and my girlfriend Alyssa, for supporting me throughout my entire university journey. You have somehow managed to put-up with me for several years and yet still love me unconditionally. I am proud to say that you inspired me to be the first member of our family to pursue a Masters degree. I would also like to say a very big thank-you to my supervisor Jarek Szlichta. As my professor, you saw potential in me and gave me my first research opportunity back in third-year of undergrad. Since then, we have worked together on various different problems and you have been a very important mentor me, not just in research but also in life. If it were not for you, there would have been a good chance I never pursued Masters at all. Similarly, a thank-you to Parke Godfrey for also providing guidance and advice throughout. I would like to thank the IBM Centre for Advanced Studies and my IBM collaborators Calisto Zuzarte, Vincent Corvinelli and Piotr Mierzejewski for providing technical insight and knowledge that had resulted in many development hours saved. Of course, thank-you as well to Ontario Tech University and all the professors that contributed positively to the person I am today. Finally, I am forever grateful for all my friends and family that stuck by my throughout the years - you have truly made my life enjoyable.

Contents

Abstract iii

Author's Declaration iv

Statement of Contributions v

Acknowledgements vi

Table of Contents xi

List of Tables xi

List of Figures xi

1 Introduction 1

1.1 Motivation 1

1.2 Real World Example 4

1.3 Contributions 8

2 Related Work 10

3 Automatic Knob Tuning **15**

3.1 System Overview 15

3.2 Query Representation 16

3.3 Deep Reinforcement Learning 24

3.3.1 Advantage Actor-Critic 25

3.4 Model design 29

3.5 Reward function design 32

3.5.1 Resource constraints 34

3.6 Training process 35

4 Experimental Study **42**

4.1 Evaluation of Our Techniques 44

4.1.1 Transfer learning evaluation 44

4.1.2 Resource constraints 50

4.2 Scalability 51

4.3 Effectiveness Comparison 53

5 Future Work **56**

6 Conclusion **59**

List of Tables

4.1 Cardinality (row count) of each table across three different sizes of TPC-DS benchmark databases. 43

List of Figures

1.1 Estimated cost of TPC-DS Query 50 as a result of varying both sortheap and bufferpool size knobs (4 KB pages). 3

1.2 Estimated cost of TPC-DS Query 51 as a result of varying both sortheap and bufferpool size knobs (4 KB pages). 4

1.3 Example of a Query Execution Plan (QEP) 5

1.4 Two different query execution plans for the same portion of TPC-DS Query 98 with two different knob settings for optimization level. . . . 6

3.1 High-level system overview of BLUTune. 16

3.2 Query representation process 19

3.3 TPC-DS Query 98 with two different knob settings for sort heap and bufferpool size. 20

3.4 TPC-DS Query 43 with two different knob settings for both sort heap and bufferpool size. 22

3.5 TPC-DS Query 68 with two different knob settings for both sort heap and bufferpool size. 23

3.6 Advantage Actor-Critic architecture utilized in BLUTune. 28

3.7 Network architecture for the Actor. 30

3.8 Network architecture for the Critic. 31

4.1 Latency comparison between training only on cost and training on both cost and execution time (transfer learning) for knobs well-represented by the optimizer. 45

4.2 Training time comparison between training only on cost and training on both cost and execution time (transfer learning) for knobs well-represented by the optimizer. 46

4.3 Latency comparison between training only on cost, on both cost and execution time (transfer learning), on execution time only and using the default settings 48

4.4 Training time comparison between training only on cost, on both cost and execution time (transfer learning) and on execution time only . . 49

4.5 Memory allocation in 4KB pages for two different memory constraints, 50000 and 250000. 51

4.6 Time spent training over three different sizes of the TPC-DS benchmark 53

4.7 Effectiveness comparison with existing IBM tools and QTune 54

Chapter 1

Introduction

1.1 Motivation

Relational database management systems have hundreds of parameters ("knobs") that control them [1], [2]. These knobs encompass many different aspects, from dictating how specific databases behave, to the database manager itself and server processes, and even influencing decisions made by the query optimizer. Some of these knobs serve high-level purposes, such as the amount of working memory, degree of parallelism, and the level of query optimization [3]. Other knobs are more coarse-grained, such as in the case of IBM DB2, the number of pages allocated to the *bufferpool* or *sortheap*, and even the ability to toggle specific functionalities of the query compiler such as requesting reduced optimization features or rigid use of optimization features at a specific optimization level [4].

Since these knobs control so many aspects of the database management system they must be properly tuned in order to achieve maximum performance, i.e. low latency and high throughput [5].

Traditionally, knob tuning is performed by a database administrator (DBA) or an expert from the database vendor themselves. However, manual knob tuning can be a difficult and long process, often involving heuristics or trial and error [6]. Three main challenges make knob tuning rather difficult: (a) the vast numbers of knobs to tune, (b) knob interdependence, and (c) how knobs impact different workloads in different ways.

As noted earlier, modern DBMS such as MySQL, PostgreSQL and IBM DB2 have hundreds of knobs that control various aspects of how they behave. It is unrealistic to expect most experts to understand the potential performance impact of each individual knob in the DBMS. Rather, experts focus on high-profile, well understood configuration parameters such as working memory, sortheap and bufferpool size to name a few. These knobs often have a rule-of-thumb associated with them which help the experts tune a suitable configuration for their given workload, e.g. bufferpool should be 20 times the size of the sortheap. However, these are only heuristics, meaning there is potentially better configurations to be found for a workload.

Understanding how hundreds of different knobs behave is a big challenge in itself, however understanding how they behave when they interact with each other is a whole different challenge. Various knobs are interdependent, meaning that changing one may affect the benefits of others. This is often the case when the knobs impact the consumption of the same resource or trigger access plan changes. An example of this in IBM DB2 would be the interaction between sortheap and bufferpool size, the size ratio of sortheap to bufferpool size may influence the query optimizer to prioritize different query execution plans over others. As seen in Figures 1.1 and 1.2, varying the size of the sortheap and bufferpool together affects the queries' estimated cost

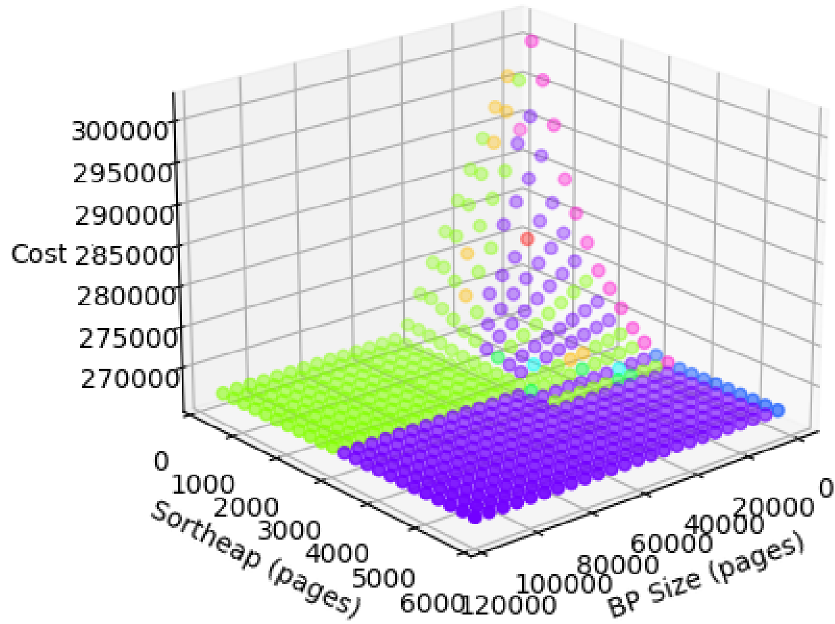


Figure 1.1: Estimated cost of TPC-DS Query 50 as a result of varying both sortheap and bufferpool size knobs (4 KB pages).

non-uniformly and also leads to various different plans being chosen (colour denotes the plan id). Another example would be tuning the degree of parallelism alongside the sortheap size. As more database operations are performed in parallel more sort consumers will allocate memory from the sortheap, potentially leading to resource over-allocation.

Another large challenge is that there is no single configuration that leads to the most optimal performance for every workload, if that were the case the default set of parameters that databases ship with would be sufficient for all users. Figures 1.1 and 1.2 demonstrate that varying the sortheap and bufferpool size for two separate queries have two largely different impacts on the estimated cost, which shows that even two queries from the same workload benefit differently from the knob configuration. This becomes more of an issue when considering *workload evolution*, businesses and their

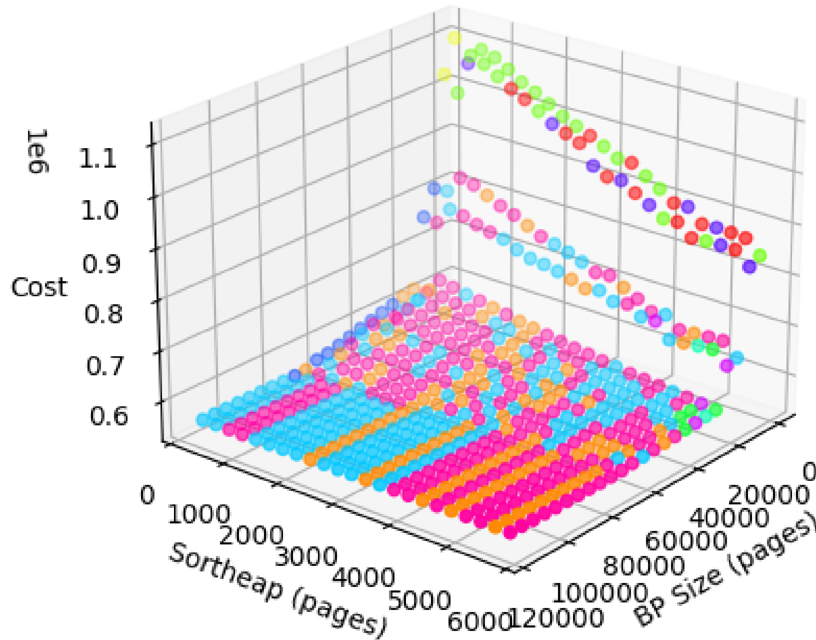


Figure 1.2: Estimated cost of TPC-DS Query 51 as a result of varying both sortheap and bufferpool size knobs (4 KB pages).

applications are not static, leading to changes in the workload overtime. As the workload evolves, so must the tuned configuration to maintain maximum performance.

1.2 Real World Example

One of the ways knobs affect queries and their respective costs and execution times is by directly influencing the choice and structure of the query execution plan (QEP) created by IBM DB2.

Consider an example of a QEP shown in Figure 1.3 depicting a portion of TPC-DS query 98. This plan performs a join between the `WEB_SALES` (`Q23`) and `WEB_RETURNS` (`Q24`) tables. The plans are trees of operators— e.g. `MSJOIN`, `HSJOIN`, `TBSCAN`, `FILTER`, `SORT` and `CTQ` which are referred in IBM DB2 as a low level plan operator (LOLE-

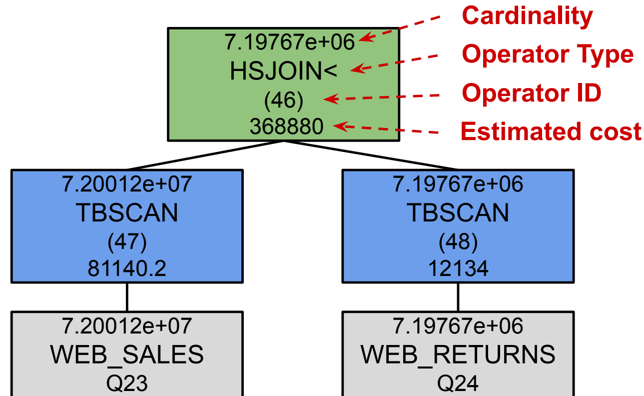
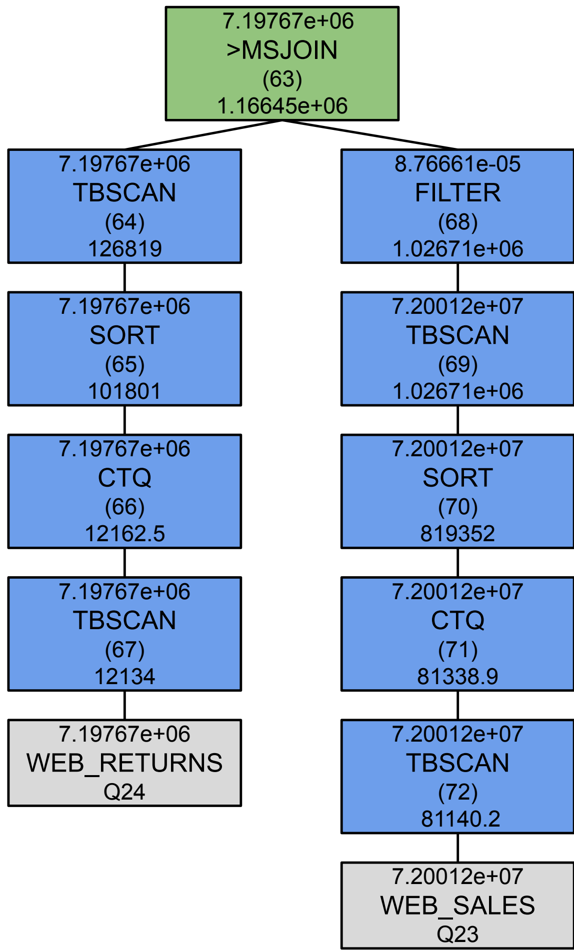


Figure 1.3: Example of a Query Execution Plan (QEP)

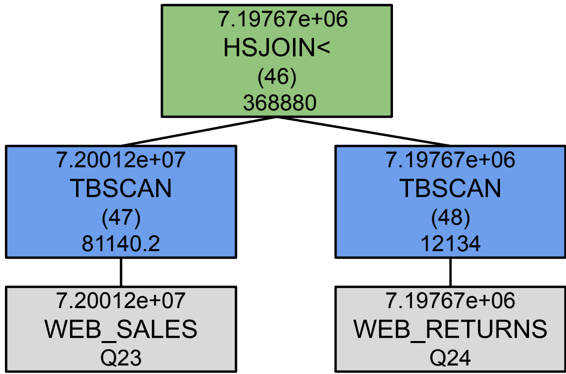
POP). The topmost decimal number of each LOLEPOP corresponds to the optimizer's estimated cardinality, the integer in parentheses represents the operator ID and the bottommost decimal number is the estimated total cost of the operation. For example, in Figure 1.3, the LOLEPOP with the `HSJOIN` has an estimated cardinality of 7.19767e+06 an operator ID of 46 and an estimated total cost of 368880. For base tables, the topmost decimal value corresponds to the table's *cardinality* (the number of rows in the table), and the value under the table name corresponds to the *instance* of the table. For example, the table `WEB_SALES` has an estimated cardinality of 7.20012e+07, and a table instance ID of `Q23`.

Now, consider the two different query execution plans in Figure 1.4 for the same portion of TPC-DS Query 98 with two different knob settings for optimization level. Both plans perform a join between the `WEB_SALES` (`Q23`) and `WEB_RETURNS` (`Q24`) tables, however this is achieved in two different ways. The first plan, Figure 1.4a, with an optimization level of 1 is more complicated than the second.

The *optimization level* knob specifies which class of query optimization techniques will be applied when preparing dynamic SQL statements. High levels of query op-



(a) Optimization level 1.



(b) Optimization level 2.

Figure 1.4: Two different query execution plans for the same portion of TPC-DS Query 98 with two different knob settings for optimization level.

timization may lead to better performing query execution plans, however this also may lead to longer compile times as the optimizations require time to compute themselves. The tradeoff between the reduced execution time of the optimized plan and the increased compile time is different for each query.

The first plan, Figure 1.4a, has an optimization level of 1, which is a set of basic optimizations that had been applied during plan generation. This plan has an expensive merge join, `MSJOIN`, between tables `WEB_SALES` and `WEB_RETURNS`. A merge join requires ordered input on the joining columns, either through index access or by sorting. Both tables are column organized and are first read by table scan, `TBSCAN`, operators. The optimizer decided that the next step is to convert the column-organized data into row-organized format through a column-organized table queue, `CTQ`, operator. The effect of this is that all subsequent operations will not leverage the compressed column-organized vectors and tuples, effectively not benefiting from IBM DB2 BLU Acceleration meant for OLAP workloads and data warehouses. The now row-organized data is sorted by the `SORT` operator, read again by a `TBSCAN` and the intermediate result from the `WEB_SALES` table is filtered by a `FILTER` operator on some predicate before finally joined with the sorted `WEB_RETURNS` table through a `MSJOIN` with a resulting cost of 1.16645e+06.

An optimization level of 2 is much bettered suited for this particular query, the resulting plan in Figure 1.4b is much simpler and much less expensive with only a total cost of 368880. The `WEB_SALES` and `WEB_RETURNS` table are joined by a hash join, `HSJOIN`, operator 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`. Not only does this avoid two large `SORT` operations, it also retains the data in a column-organized format with the absence of a `CTQ` operator. This allows subsequent operations to be accelerated as the data is still compressed column-organized vectors and tuples.

The resulting elapsed time, including execution and compile time, for a query optimization level of 1 was 157.86 seconds and 64.21 seconds for an optimization level of 2, a 59% reduction!

1.3 Contributions

Our main contributions in this paper are as follows:

- An automatic database knob tuning system for IBM DB2, utilizing deep reinforcement learning, specifically Advantage Actor-Critic (A2C), to produce effective knob recommendations.

- We design a single model capable of tuning both continuous and discrete knobs simultaneously. These recommendations are within user-defined resource constraints, i.e. limited memory, as our model is trained to abide to these restrictions.

- A multi-stage training process which utilizes the optimizer's cost estimates to quickly bootstrap a model with a sufficient understanding of the behaviour of most knobs followed by fine-tuning using Transfer Learning. Elapsed time is later used instead of estimates to fine-tune the agent's understanding based on real-life performance.

- A comprehensive experimental study evaluating and demonstrating the effectiveness of our approaches and the scalability of BLUTune over a complex OLAP workload. Experimental results show that BLUTune can learn configurations better than that of existing IBM tools.

In Section 2 we discuss the related work. In Section 3.1, we overview BLUTune's architecture. In Section 3.2, we describe how SQL statements can be represented as vectors or embeddings through the use of query execution plans. In Section 3.3, we detail our reinforcement learning approach, model design, reward function design, consider resource constraints and outline our training process. In Section 4, we provide a comprehensive experimental evaluation. In Section 5, we discuss future work and we conclude in Section 6.

Chapter 2

Related Work

Database performance tuning has been an interest of researchers, IBM and other database vendors for the past two decades. One area of research has been focusing on tuning the queries and the optimizer's plan rewrites themselves. OptImatch [7][8] and subsequently GALO [9][10] tune the performance of query execution plans by generating random permutations of the access plan until a better plan is found. The plan is stored as a guideline in a knowledge base to be later matched and applied to a query that will benefit. These tools aim to address the shortcomings of the query optimizer as it can be prone to inaccuracies such as cardinality estimation.

Database configuration parameter tuning is also a large area of interest for improving database peformance. IBM has a tool called the Configuration Advisor which allows DBAs to obtain recommended settings for common IBM DB2 configuration parameters to help alleviate administration requirements [1]. The tool requires three sets of information: **(1)** user specification of the database environment (e.g. type of workload, number of statements, memory percentage), **(2)** automatic sensing of system characteristics (e.g. number of CPUs, amount of RAM, number and size of

tables) and finally **(3)** expert heuristics for database configurations (as reported by experienced DBAs and tuning experts). Although it provides a sensible starting point for database performance tuning, it leaves a lot to be desired, mostly due to the requirement of human input to describe characteristics of the database and workload, but also due to the fact that it is an expert system built largely around heuristics. This leaves room for improved performance from better suited configurations that heuristics may have failed to capture. In DB2 version 9.1, IBM introduced the Self-Tuning Memory Manager (STMM), which provides adaptive self tuning of database memory allocation through heuristics [11]. STMM was not designed with column-organized tables in mind, meaning it is unable to tune certain parameters for our workload.

Similar to IBM's offerings, Oracle developed the Automatic Database Diagnostic Monitor which diagnoses database performance issues and recommends tuning actions to alleviate them [12]. This tool helps DBAs by recommending actions for a small set of configuration settings based off of heuristics. Oracle later developed the SQL Tuning Advisor, which takes one or more SQL statements and recommends statement-specific tuning actions to improve the performance [13]. These actions could be adding optimizer hints to the statement, statement rewrite, index creation, and of course changing the value of configuration parameters. Oracle fully automated the SQL tuning workflow of the SQL Tuning Advisor with a database feature called Automatic SQL Tuning [2].

iTuned is a database tuning tool that executes planned experiments through a cycle-stealing paradigm to gather performance data and statistics while the DBMS is not being fully utilized [14]. 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. iTuned's goal is to learn how a knob setting impacts performance for a given workload, and recommend a configuration that maximizes performance. It does not consider how different queries have differing performance impacts from configurations, it also requires training from scratch for each different workload, and can take tens of hours to find a suitable configuration for a single workload.

OtterTune, a more modern database configuration tuning tool, first identifies and ranks the knobs which have the strongest impact on the performance of the DBMS, maps the given workload to a repository of previously collected performance measurements, and utilizes a Gaussian Process model similar to iTuned to choise knob settings with the best expected performance [15]. OtterTune requires a large corpus of previous tuning sessions in order to leverage knowledge it has gained in the past. Relying heavily on this corpus does not allow OtterTune to generalize well to unseen workloads.

BestConfig is a system for finding the best configuration setting within a resource limit to maximize performance for a given system, application and workload. 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 database knob tuning [17]. CDBTune utilizes Deep Deterministic Policy Gradient (DDPG), a policy-based reinforcement learning method that is an extension of

Deep Q Networks. DDPG is an off-policy algorithm that is largely intended only for continuous action spaces and requires a large replay buffer of past experiences. The state/observation for the agent is the internal database metrics, such as counters for pages read from or written to disk. The critic network learns the expected Q-value given a state and tuning action. CDBTune does not consider the queries themselves and only acts to the anticipated change to database health metrics.

QTune is the most similar to our work. Similar to CDBTune, QTune also uses DDPG but places an emphasis on featurizing queries using their query execution plans [18]. For a query, statement details, table involvement, and cost information about each operator is extracted from the QEP and represented within a vector. This vector is used as input for a separate deep neural network which predicts the change in performance metrics as a result of executing the query, which is finally used as the input state for the agent. This additional network requires many high-quality training samples to pre-train. The overall performance of the agent is greatly tied to the predictions from the additional network and not a tangible measure like execution time.

Marcus *et al.* takea a unique approach to query performance prediction by using tree-structured neural networks that models the entire query execution plan as a tree of neural units [19]. The neural units are smaller neural networks that predict the latency of a given operator in the plan. Mapping a query execution plan to the resulting latency is something that is implicity done by our agent, however their approach allows for this mapping to be built separately offline.

Gur *et al.* propose a multi-model tuning solution utilizing DDPG to address deployments that consist of multiple, very different workloads [20]. They argue that

training data collected from a OLAP workload cannot be used to train a OLTP workload, meaning multiple DDPG models are required to handle the different workloads. A downside of this is that scenarios where many different models are needed is possible, which can make this a costly approach.

The most recent work in the database tuning space, ResTune, was published during the creation of this work. ResTune uses constrained Baysesian Optimization along with meta-learning to produce knob configurations [21]. The produced knob configuration are constrained by a Service Level Agreement, meaning there is restrictions on the resource usage. Similar to our work, this ensures knob configurations abide by a memory limit for instance. However, ResTune takes this further and attempts to minimize the amount of resources allocated as well.

Chapter 3

Automatic Knob Tuning

3.1 System Overview

BLUTune is a system to fully automate the database configuration (knob) tuning process for IBM DB2 with the goal of maximizing the performance for a given SQL workload. We define a *workload* to be a populated database and schema along with a collection of SQL queries that are regularly executed on a given database system instance. Performance is measured as the total elapsed time for compiling and executing a query until a result is returned.

Figure 3.1 depicts a high-level system overview of BLUTune. A collection of SQL queries from the workload are sent through IBM DB2 to retrieve information about their query execution plans (QEP) without executing the queries themselves. These plans contain a significant amount of information about a query, the operators and their costs, and also information about the underlying data in the table. This information is fed into `QEP2Vec` which vectorizes the query execution plans to be used as input to our reinforcement learning agent. These queries can be sent as a

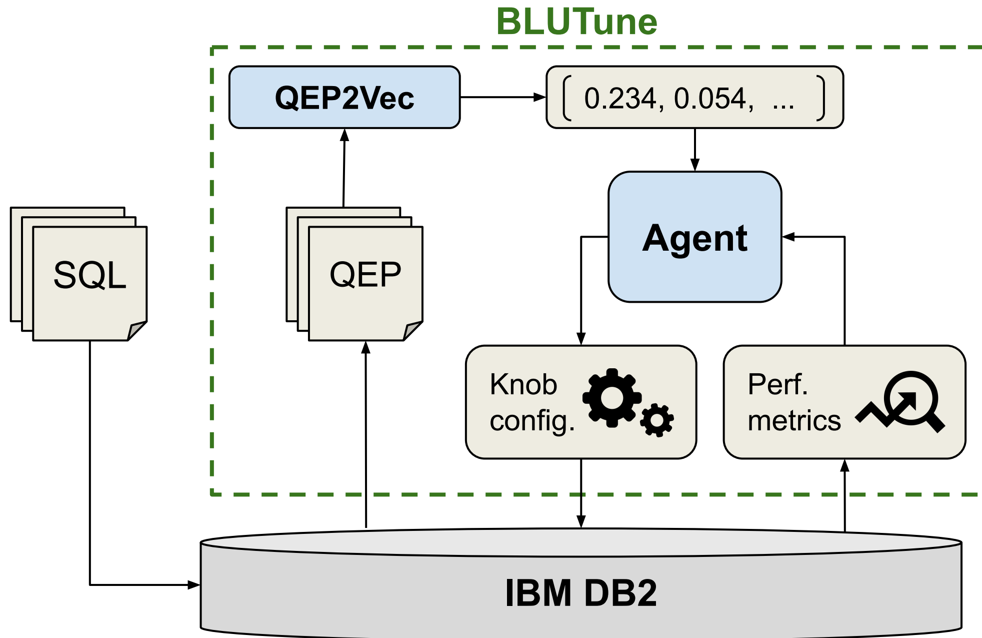


Figure 3.1: High-level system overview of BLUTune.

group where their vectors are aggregated and represented by a single vector and tuned together as group, or they can be handled one-by-one and tuned for individually. The way query execution plans are vectorized is detailed in Section 3.2.

The reinforcement learning agent takes the QEP vector as input, produces a knob configuration and applies it to the database instance. The query themselves are subsequently executed under the new configuration and the resulting performance metrics are reported back to agent to guide the learning of the agent.

3.2 Query Representation

Queries are traditionally represented as SQL statements. A SQL statement itself can have quite a bit of information contained within it, e.g. statement type (SELECT,

UPDATE, DELETE, INSERT), the tables involved and some SQL operator information (predicates, group-by, etc). While this information appears to useful, it is difficult to represent in a form that our agent and it's neural networks can easily interpret, such as a numerical vector.

SQL is a declarative programming language, meaning it expresses the logic of a computation without actually specifying the control flow. Queries, when sent to a database system for execution, are first optimized by the query optimizer and a query execution plan is created. This plan is comprised of LOLEPOPs, which are low-level plan operators, that specify how a query is processed step-by-step to produce the desired result. An example of a visual representation of a QEP was provided earlier in Figure 1.3. These plans contain useful information such as the plan operators and their respective estimated costs, join orders, table cardinalities and much more.

Our initial approach is to focus mainly on the plan operators themselves and their respective costs. The operators involved in a query give a general indication of how the query will behave, and the estimated cost for each operator represents how expensive computationally it will be. The estimated cost is measured in *TIMERONs*, calculated by a proprietary algorithm in IBM DB2 that estimates the cost by using statistics, knob settings, indexes, filter factors and other information at its disposal to produce the weighted sum of the I/O and CPU cost. From this we get a way to represent queries and their expected performance impact upon execution, which forms the basis of the input to our reinforcement learning model.

In IBM DB2 there exists 31 distinct LOLEPOPs for query execution plans to be composed of. The `QEP2Vec` component extracts the plan operators and their costs from the QEP and outputs a vector of length 31, with a position for each operator's

estimated cost. Operators may appear more than once in a plan, thus the estimated costs for each occurrence is summed together. This vector is standardized by Z-score normalization to improve the learning process of our model. The result is a vector that captures the involvement of each operator and the extent of their impact on performance. This process is depicted in Figure 3.2. The time it takes for the optimizer to create, consider and choose a query execution plan (compile time) is normally very quick, a matter of milliseconds. For complex OLAP workloads the compile time is negligible, however for fast executing OLTP workloads the compile time can contribute to the overall elapsed time of executing a query. A key realization is that in order to execute a query a query execution plan must be created first, meaning our query representation adds little overhead as we can fetch the details from internal `EXPLAIN` tables.

The query vector not only represents a query and it's anticipated costs, but it also captures how specific knobs will improve a plan and it's performance as many knobs are considered by the query optimizer when plans are chosen. This means our representation will also reflect the expected performance improvement from various knob configurations, as reflected in the estimated cost for the query and it's operators.

In Figure 1.4 we had shown that a particular knob, optimization level, is capable of producing very different query execution plans with different join types and estimated costs. Another example of differing knob settings affecting query execution plans is shown in Figure 3.3. A portion of the query execution plan for Query 98 is shown again here, changing the sort heap size and bufferpool size knobs from 1000 and 2000 to 10000 and 20000 respectively. This particular change in knob settings does not trigger a change in the structure of the query execution plan, however it directly impacts the

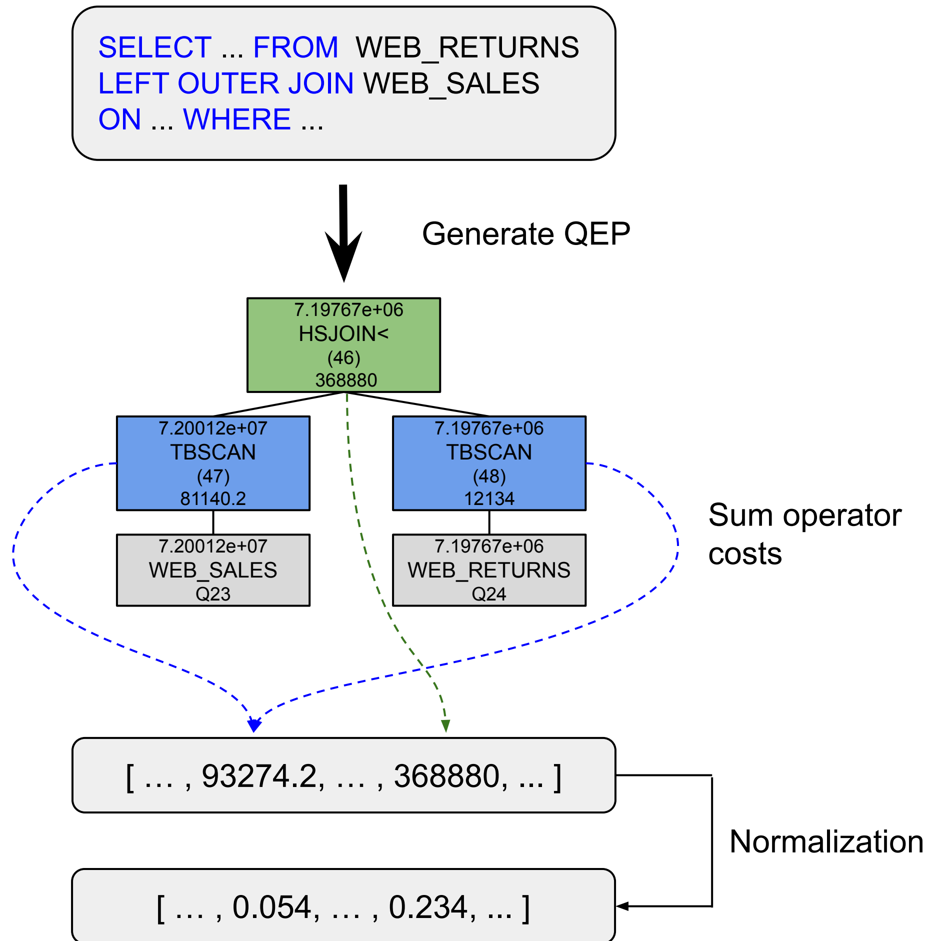
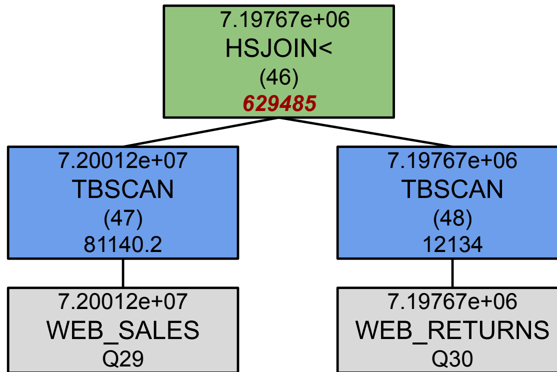
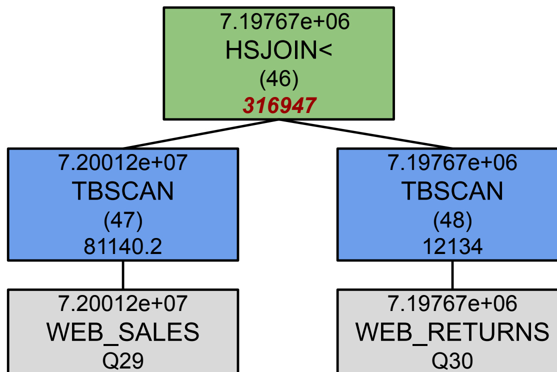


Figure 3.2: Query representation process



(a) sortheap = 1000,
bpsize = 2000



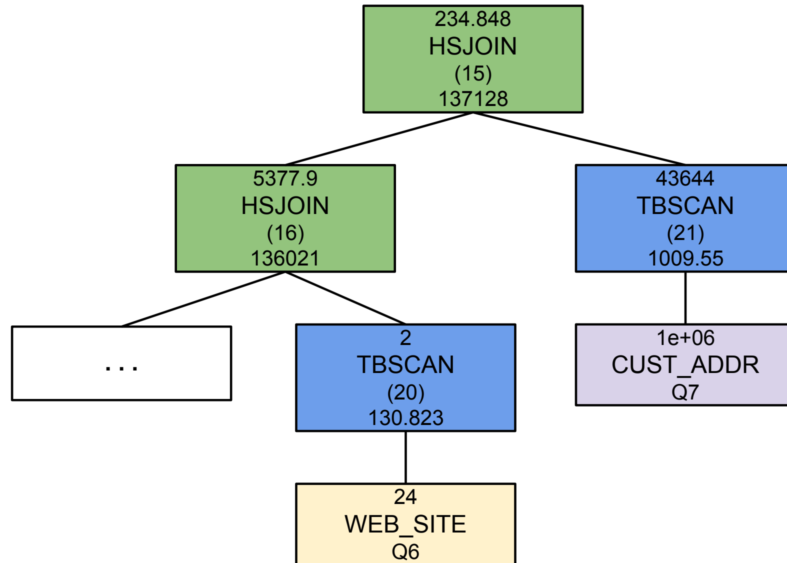
(b) sortheap = 10000,
bpsize = 20000

Figure 3.3: TPC-DS Query 98 with two different knob settings for sort heap and bufferpool size.

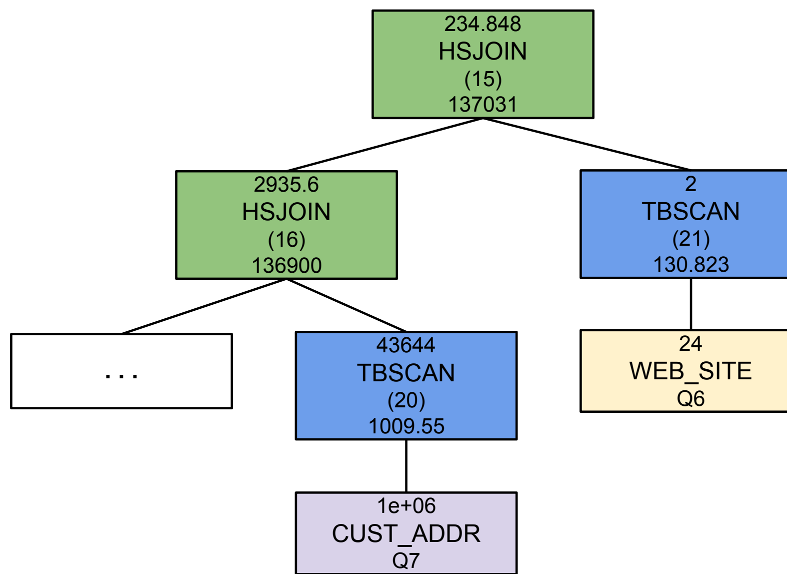
estimated cost of the `HSJOIN` (`46`) operator, highlighted in red. Figure 3.3b shows a near 50% decrease in estimated cost between the two configurations. This decrease in estimated cost is due to reduced I/O cost, since more memory is available for the bufferpool and sort consumers, less reads will needed to be performed.

Similarly, in Figure 3.4 the effect of altering the sortheap and bufferpool size knobs is shown. The main difference between the two portions of the plan is the join order, tables `WEB_SITE` (`Q6`) and `CUST_ADDR` (`Q7`) are swapped. Although the `HSJOIN` (`15`) only saw a small estimated cost decrease, the overall plan cost was reduced from 2.15188e+07 to 1.98359e+06, a 90% decrease. This further shows that many knobs influence the query execution plans and operators costs, making both excellent for the capturing the impact the knobs have on the queries.

To further illustrate that query execution plans can be greatly impacted by tuning knobs, we demonstrate the effects of tuning only sortheap and bufferpool for a variation of TPC-DS query 68 with it's predicates altered in Figure 3.5. These two plans make heavy use of `TEMP` operators, which refers to temporary tables. These temporary tables were created earlier in the query execution plan, if a result is required multiple times in a single plan then it is temporarily stored to reduce the computational requirement. The obvious differences between the two plans is where and when each `TEMP` table is joined through a series of *Nested-Loop Join* (`NLJOIN`) and `HSJOIN` operators. Changing the join order can lead to more efficient joins and contribute to a reduction in estimated cost. However, the largest reduction in cost between the two plans is the `TEMP` table (coloured yellow) which is operator `21` in the first plan and `30` in the second. These two temp tables have the same result, as indicated by their same cardinality of `1.99885e+06`, but very different estimated costs of `6.83944e+06`

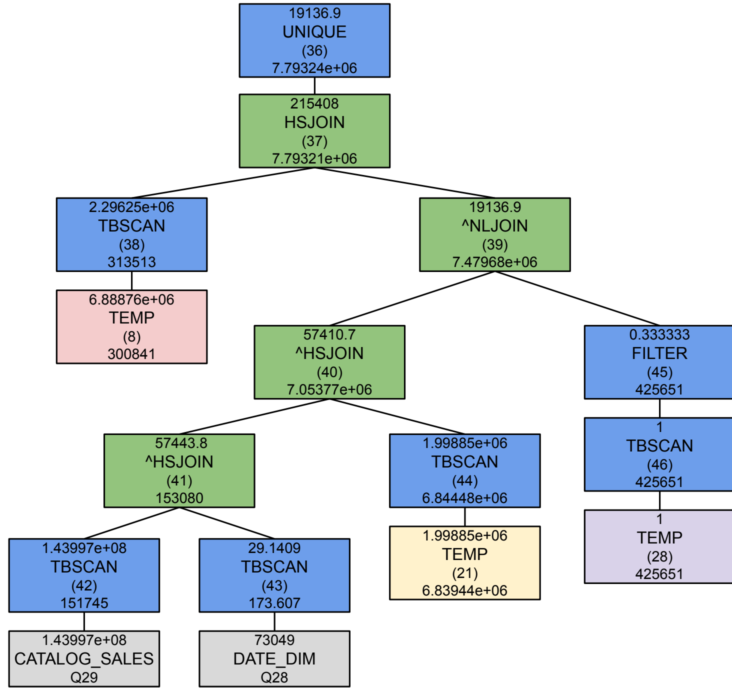


(a) sortheap = 500, bpsize = 1500

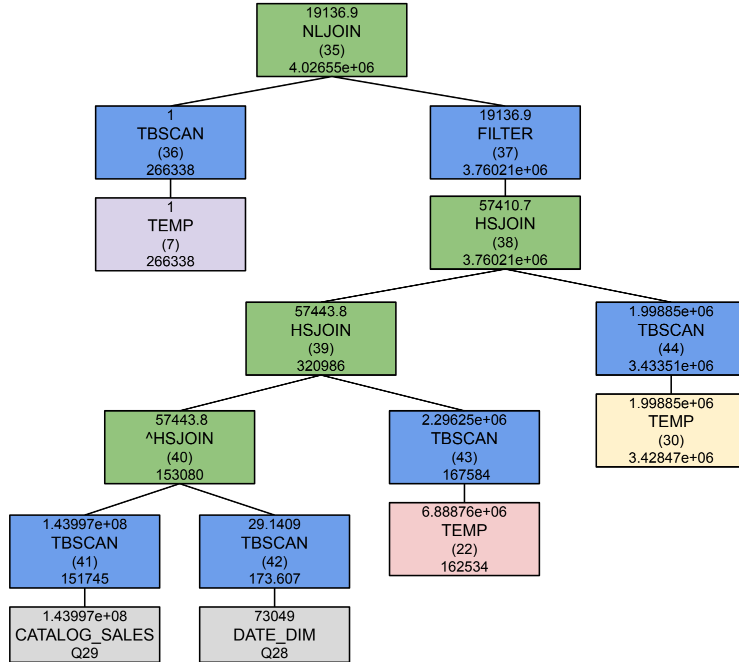


(b) sortheap = 5000, bpsize = 6000

Figure 3.4: TPC-DS Query 43 with two different knob settings for both sort heap and bufferpool size.



(a) sortheap = 1,000, bpsize = 3,200



(b) sortheap = 20,000, bpsize = 40,000

Figure 3.5: TPC-DS Query 68 with two different knob settings for both sort heap and bufferpool size.

vs. `3.42847e+06` (approx. 50% reduction). Inspecting the corresponding sub-graph of these temp tables in each plan reveals that they are identical as they both perform a `HSJOIN` between the `STORE_SALES` and `CUSTOMER` table, followed by a *Group-By* (`GRPBY`). These two tables are quite large, and the increased availability of bufferpool and sort memory greatly reduces the I/O cost of the operators.

Other works have used the effects the query has upon the performance metrics as input, i.e. number of pages read or written to disk [17], but we argue it fails to capture information to succinctly differentiate between queries the system attempts to tune for.

3.3 Deep Reinforcement Learning

Reinforcement learning (RL) is a suitable approach to solving the problem of automatic database configuration tuning. Reinforcement learning is comprised of a machine learning agent and an environment. The agent takes actions within the environment with the goal of maximizing it's cumulative reward. At it's core, it is a Markov Decision Process, where at each timestep t, the agent in a current state S_t chooses an action A_t and transitions into a new state S_{t+1} and receives some reward R_{t+1}, where S_t belongs to set of states S, which represents every configuration of the environment and A_t belongs to set of actions A, which is every valid action an agent can take within the environment.

As one can imagine, with a very large state space S and large action space A, it can be difficult to learn an optimal policy that can grasp the action-state space to maximize expected cumulative reward. This is where deep reinforcement learning (DRL) has an advantage. Deep reinforcement learning combines reinforcement

learning with deep learning. Incorporating a neural network in our agent allows us to handle large, complex action-state spaces.

There is no theoretical limit on the size of our state space S. Our state is the query vector from Section 3.2, with the estimated costs of the 31 operators, each their own continuous value. Similarly, our action space represents each possible knob configuration, which in our system is a combination of both discrete and continuous values. Thus, deep reinforcement learning and neural networks are required to learn an optimal policy.

A major reason DRL was chosen is because it learns through trial-and-error. The agent combines both exploration (discovering uncharted territory) with exploitation (utilizing learned knowledge). This means it does not require high-quality data samples to learn off of, it interacts and learns from the environment completely on it's own. Collecting this data in first place would be challenging, as mentioned earlier a knob configuration affects different queries in different ways, and creating a large enough corpus of samples to capture this would be a difficult task on its own. Also, considering different workloads would need their own data samples makes this even more unrealistic.

3.3.1 Advantage Actor-Critic

BLUTune utilizes deep reinforcement learning, specifically Advantage Actor-Critic, to solve the task of automatic database configuration tuning. Reinforcement learning methods are either *value-based* or *policy-based.* Value-based methods attempt to learn the optimal value function, a mapping between an action and a value. An example of this would be Deep Q Learning, where a learned function $Q(s, a)$ estimates the

overall expected reward for an agent in state s performing an action a, and continues following some policy π until the end of the episode. The larger the Q-value, the better the action is considered to be. Policy-based methods attempt to learn the optimal policy directly, a mapping between state and an action, without the use of a Q-value. An example of policy-based methods would be policy gradient methods. Value-based methods are regarded as being more sample efficient and steady, whereas policy-based methods work better for continuous environments and converge faster.

Actor-Critic is a combination of both policy-based and value-based methods with the intent of combining the advantages of both methods. The main idea is to separate the model of the agent into two components, 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). Both the actor and the critic interact with each other improving the effectiveness of both together.

Advantage Actor-Critic (A2C) is an improvement on the original Actor-Critic algorithm. The Q-value function $Q(s, a)$ quantifies how good a particular action is for a given state. However, $Q(s, a)$ alone fails to capture how good an action is with respect to other possible actions. For example, with $Q(s_1, a_1) = 10$, a_1 may appear to be a good action for the given state s_1, however if there exists a different action a_2 such that $Q(s_1, a_2) = 100$ then a_2 is clearly much better than a_1. It would be useful to know what is the *advantage* of taking a specific action.

The advantage of a given state-action pair is defined in Equation 3.1 as $A(s, a)$ the difference between the Q-value $Q(s, a)$ and the state's Value function $V(s)$. The value function $V(s)$ is a measure of the expected reward from the agent being in state s and continues following some policy π until the end of the episode. $V(s)$ is similar

to $Q(s, a)$ except it does not consider any actions.

$$A(s, a) = Q(s, a) - V(s) \tag{3.1}$$

A natural assumption would be that we now have to learn both $Q(s, a)$ and $V(s)$, making learning more complex. However, this is not the case. The Q-value as mentioned earlier is the expected reward for taking action a in state s, and continue following some policy π until the end of the episode. This can be written in Equation 3.2 as the sum of the immediate reward $r(s, a)$ and the value $V(s')$ of the next state s'. Using this definition, we can rewrite the advantage as shown in Equation 3.3. This is also known as the TD error, which is an unbiased estimate of the advantage function.

$$Q(s, a) = r(s, a) + \gamma V(s') \tag{3.2}$$

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

Following this, it becomes apparent we only have to learn the state value function V and use it twice – for the current state s and the next state s'. Instead of learning $Q(s, a)$ which requires both a state and action as input, we only have to learn $V(s)$ which does not require the action as input. 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 Equation 3.4 and Equation 3.5 respectively. For an action a and a state s the policy π is

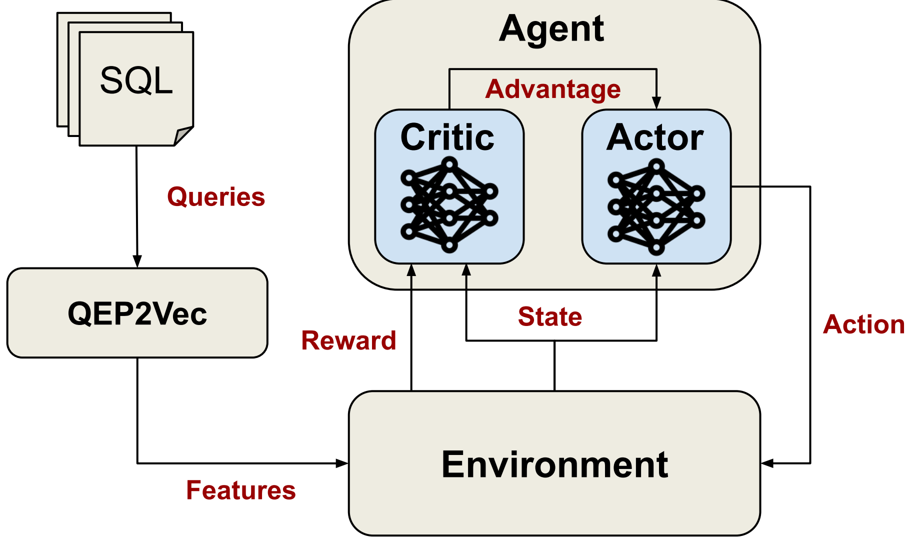


Figure 3.6: Advantage Actor-Critic architecture utilized in BLUTune.

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 advantage with the gradient of itself.

$$\begin{aligned}\Delta\theta &= (r(s,a) + \gamma V_w(s') - V_w(s))\nabla_\theta log\pi_\theta(a|s) \\ &= A(s,a)\nabla_\theta log\pi_\theta(a|s)\end{aligned} \tag{3.4}$$

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

Our Advantage Actor-Critic model is depicted in Figure 3.6. The `Environment` in our case is IBM DB2. Our `Agent` is composed of both an `Actor` network and a `Critic` network. The SQL statements are transformed into QEPs given a default

knob configuration and turned into a numerical vector by `QEP2Vec` to be used within our networks. The `Actor` observes the initial state from the `Environment`, which is a query vector in this case. Utilising it's neural network, the `Actor` recommends a knob configuration that is subsequently applied to the `Environment` (IBM DB2) causing a state transition. A reward is produced based on the change in query performance as reported by the `Environment`. Given the initial state, the action, it's reward and the resulting state, the `Critic` learns how good the action was and how it compares to others at a given state. The `Critic` calculates the advantage value and feeds it to `Actor` so the `Actor` can learn whether or not the action it took was appropriate for the particular state. This process loops until the agent converges or a iteration limit is reached.

3.4 Model design

Database configuration parameters exist in both discrete or continuous values. For instance, in IBM DB2 the optimization level knob has 7 discrete choices: [0, 1, 2, 3, 5, 7, 9]. The `sortheap` parameter can be any integer between `16` and `4,294,967,295` which is considered a continuous value for our purposes, since we can not feasibly represent that many choices in our model.

Our `Actor` network can handle both continuous and discrete actions simultaneously, it's architecture is depicted in Figure 3.7. The input to the model L_1 is the current state, or in our case the vector of summed operator costs from the QEP (detailed in Section 3.2), which is pre-standardized through Z-score normalization as the operator costs can be very large. This feeds through two hidden layers L_2 and L_3, both with ReLU (Reftified Linear Unit) activation. L_2 is the largest layer, which

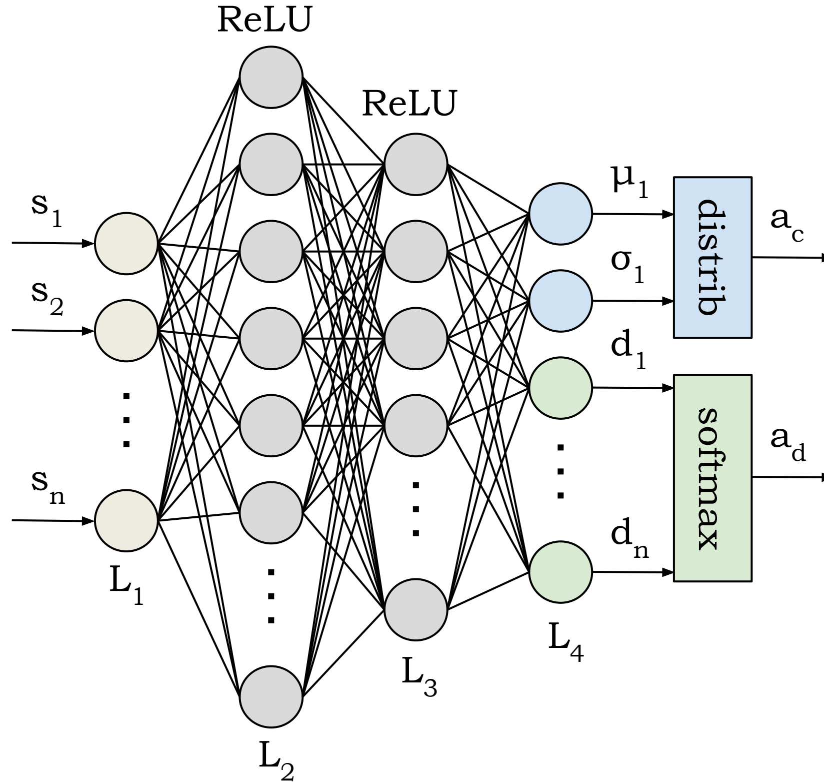


Figure 3.7: Network architecture for the Actor.

learns low-level features and patterns from the input layer L_1. This information is then distilled and generalized by a smaller layer L_3. The output layer L_4 produces the intermediate action vector, which must be broken apart and transformed into actions usable by our agent.

Both continuous and discrete actions are embedded in the output layer L_4. The blue nodes in L_4 are associated with a single continuous action, outputting a mean μ_1 and standard deviation σ_1. A normal distribution is created from μ_1 and σ_1, and a continuous action a_c is sampled from this distribution. This distribution facilitates exploration around the learned value μ_1 by adjusting σ_1. To facilitate additional continuous actions, a multivariate normal distribution can be used by learning multiple

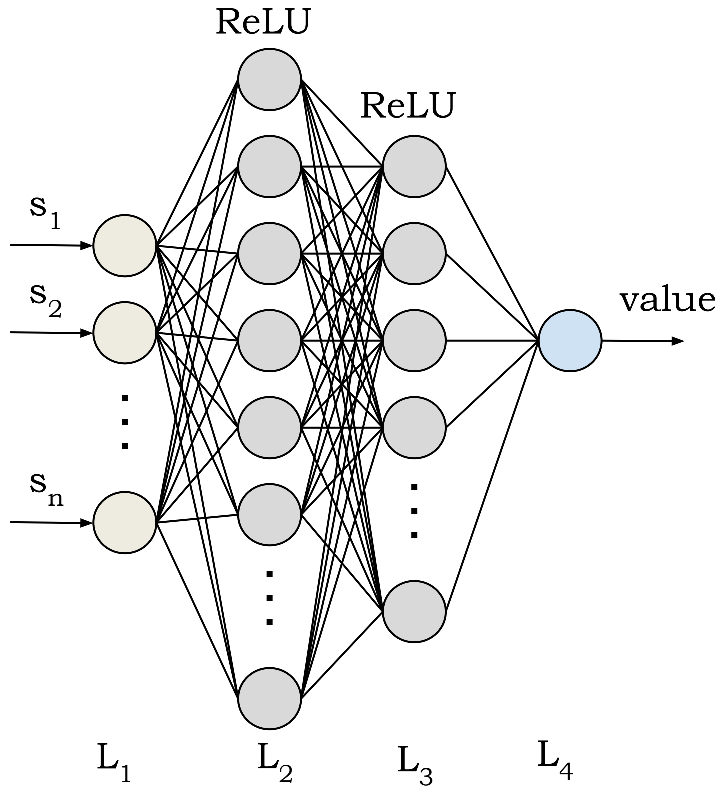


Figure 3.8: Network architecture for the Critic.

μ and σ values.

A discrete action is depicted by the green nodes in L_4. For each discrete action, there are n fixed choices. Thus, in L_4 there is an output node for each choice, d_1 ... d_n. In order to make a recommendation for which choice should be selected these values are fed through a `softmax` function, which produces a probability distribution summing to 1. From this categorical distribution we can sample a choice for a discrete action a_d.

Our `Critic` network is responsible for producing the state value for the advantage function to guide the `Actor`'s learning. Given a state and an action, the `Critic` evaluates how good the action is and how better it is compared to other actions

based on the received reward. Figure 3.8 depicts the network architecture for the `Critic`. The network takes only the state as input in layer L_1, as the actions are not considered by the state value function. The state feeds through fully-connected layers L_2 and L_3, each with ReLU activation applied. The `Critic` outputs the state value in layer L_4, which is used to calculate the advantage.

3.5 Reward function design

The reward given to the `Agent` by the `Environment` for taking a tuning action at a specific state is critical to guiding the learning of the agent. The reward is utilized by the `Critic` to evaluate an action for a given state in order to learn an optimal advantage value.

The main goal of the agent is to produce a knob configuration that minimizes a cost metric for a given query or workload. Our reward is a function of the execution time or the estimated cost of a query as these are good indicators of performance. If a configuration results in the estimated cost or execution time being reduced, then we give our agent positive reward since this is desired behaviour. Conversely, a rise in cost or time leads to a negative reward.

$$\mathbf{r} = r_{history} + r_{best} \tag{3.6}$$

Our reward, Equation 3.6, has two main components: $r_{history}$ and r_{best}.

$$r_{history} = -1.0 * \sum_{n=1}^{N} \frac{x - c_n}{c_n} * e^{-\lambda n} \tag{3.7}$$

$r_{history}$ is a measure of how the current cost compares to the cost received in previ-

ous tuning iterations, shown in Equation 3.7. We keep a cost history of the previous N iterations. For each cost c_n in the history we calculate the relative difference between the current cost x and c_n, then apply exponential decay to each value and finally sum all the values together. Exponential decay is used to make costs further back in the cost history contribute less to the reward. We keep a history rather than just the single previous cost to help steer the agent and ensure it is making meaningful progress in the correct direction. The goal is to have the agent take a tuning action that is better than the last, or reach the point where the agent cannot find an action that is significantly better than the last N actions, or in other words reach convergence.

Although $r_{history}$ is a good measure of how the agent is performing compared to the past N iterations, there can be instances where the agent finds itself in a situation where the current action and all of the past N actions are rather poor. This can lead to the agent making little progress, trapping it in a local minima. In order to avoid this a second component, r_{best} is introduced to the reward.

$$r_{best} = -1.0 * \frac{x - c_{best}}{c_{best}} \tag{3.8}$$

r_{best} is a measure of how the current cost from a tuning actions compares to the best known cost of the best known action. As the agent explores and tries different tuning actions, we make note of the best cost c_{best} encountered so far for a given query or workload. Equation 3.8 shows the relative difference calculated between the current cost x and the best cost c_{best}. This gives a second goal to the agent, to produce tuning actions with similar or better results than that of the best known action. In the case where the agent is trapped in a local minima and is receiving little

information from the $r_{history}$ component, it can be guided by the r_{best} component to escape and seek actions that are similar to what is the best known action.

The decimal values from both r_{best} and $r_{history}$ are each clipped within [-1,1], meaning the total reward r is clipped to [-2,2] as a result.. This was done because there is no determinable upper and lower bound on these values. The relative difference between two given costs can sometimes have a very large and unbound magnitude. Large reward magnitudes can lead to large gradients, hampering the learning efficiency of our agent.

3.5.1 Resource constraints

One of the objectives of BLUTune is to account for hardware limitations on database instances and produce knob configurations that abide by these constraints. This objective arose as a necessity as our agent quickly realized that indefinitely increasing the value of specific knobs led to better estimated costs and execution times. For example, sortheap and bufferpool size are specified in terms of 4KB pages. These pages are allocated from the main database instance memory, *working_mem*. It is possible to have a large enough sortheap that when many sort consumers allocate heaps at once all the allocated memory is exhausted, leading to the failure of a query's execution. It is important to generate knob configurations that do not exceed system resources and lead to query execution failures. Thus, the agent must learn to operate within these constraints.

These constraints can be conveyed to the agent through the reward function. The purpose of the reward is to inform the agent how good a particular action was. If the tuning action exceeds a predefined limit or leads to an error, the action is clearly

poor and the reward should be negative to deter similar actions from being taken.

$$r_{res} = a * \frac{(\sum_{n=1}^{|G|} k_n) - limit}{b} \tag{3.9}$$

A resource reward function r_{res} in Equation 3.9 addresses this. Given a set of knobs $G = (k_1, k_2, ..., k_n)$ of length $|G|$ that consume the same resource res, i.e. memory pages, we sum the total amount of the resource allocated across each knob in G and calculate the difference between the sum and the resource $limit$. This value is divided by a constant b and multiplied by another constant a, which adjusts how large the reward penalty will be for exceeding the limit.

To impose a memory constraint on the agent, we chose to limit the number of 4Kb pages that were being allocated through the sortheap and bufferpool size knobs. For instance, 100,000 4 Kb pages would be 400 Mb worth of memory.

Once the sum of $sortheap$ and bufferpool ($bpsize$) exceeds some defined 4Kb page $limit$, the previous reward r is replaced by r_{res} from Equation 3.9. For every 1000 pages (b) over the $limit$, a reward of -0.25 (a) is given. This negative reward scales linearly, to communicate to the agent that increasing the knob values further only leads to worse rewards. This results in the agent quickly learning not to exceed the memory constraint as there is only negative rewards to be had.

3.6 Training process

Training the agent only requires a *workload*: a set of queries, their respective schema and a database instance. Training consists of N episodes, where in each episode a random query or collection of queries is sampled from the workload to be tuned for.

The current knob configuration is reset to some specified default values and applied to the database at the beginning of the episode. The main training loop is shown in Algorithm 3.1 and begins by first running the sampled queries through an EXPLAIN statement to the IBM DB2 database, this produces a query execution plan (QEP) that serves as the basis for our query representation along with a cost estimate from the optimizer. The QEP is transformed through our `QEP2Vec` component into a numerical vector that can be used as the state for input to our model. If multiple queries were sampled the resulting vector is averaged as a representative of the collection and the estimated total cost is summed across each query. The query vector or State is fed through our agent which produces a set of actions, a recommended knob configuration, which is subsequently applied to the database instance. The queries are once again ran through an EXPLAIN statement which factors in the new knobs to generate the resulting QEP and the new estimated cost. The new QEP is transformed through `QEP2Vec` to produce the new state and the new estimated cost which is the key performance metric of our reward function. The initial state, the actions taken, the resulting reward and the new state is used by our agent to update the weights of the model. This process is repeated for the episode until the model converges, i.e. the resulting reward for the past n iterations has changed very little if at all, or until a user defined iteration limit is reached. Once the episode has reached a terminal state, the next episode starts with a different sample of queries from the workload and follows the same process.

The agent's action selection process is detailed in Algorithm 3.2. Given some state observation *State* and the actor's policy network π_θ the agent produces an action recommendation. This is achieved by first feeding the *State* through the

Algorithm 3.1 Training BLUTune

Input: Set of training *queries*
Output: Trained *Agent*

```
1: for episode ← 1 to N do
2:     Knobs ← Reset to default
3:     Query ← Sample query from queries                ▷ Could also be a workload
4:     while !converged or iterationlimit not reached do
5:         QEP ← EXPLAIN Query given Knobs
6:         State ← QEP2Vec(QEP)
7:         Knobs ← Agent.choose_action(State)
8:         QEP′ ← EXPLAIN Query given new Knobs
9:         State′ ← QEP2Vec(QEP′)
10:        Reward ← Calculate using estimated cost
11:        Agent.learn(State, Knobs, Reward, State′)
12:    end while
13: end for
```

policy network π_θ to produce an intermediate output which contains all of the learned values required for each knob. For each continuous action there is a learned mean μ and variance σ within the output. For each discrete action, there is an entry in the output vector for each possible option for that particular knob. The output is split into the learned values for each knob for both continuous actions A_c and discrete actions A_d. For each discrete action and their respective action vector a_n from A_d, apply a Softmax function over the action vector so all the elements sum to 1, suitable for a probability distribution. A categorical distribution is created from the result and an *action* is sampled randomly. At this point we calculate the log likelihood of the particular *action* given the distribution and store it for the learning phase. Finally, *action* is recommended as the knob setting for the discrete knob. For selecting continuous actions, we create a vector of means $\hat{\mu}$ and a vector of variances $\hat{\sigma}$. Using $\hat{\mu}$ and a diagonal matrix from $\hat{\sigma}$ we create a multivariate normal distribution. From this distribution we can sample multiple continuous actions randomly at once.

Once again, we calculate the log likelihood of selecting *actions* with respect to the multivariate distribution.

Algorithm 3.2 Agent action selection

Input: Observed $State$, Actor's policy network π_θ
Output: Knob recommendations

1: $Output \leftarrow \pi_\theta(State)$ ▷ Feed State through Actor
2: Split $Output$ into action vectors for all continuous A_c and discrete actions A_d
3: **for** each discrete action vector a_n in A_d **do**
4: $m \leftarrow SoftMax(a_n)$
5: $distrib \leftarrow$ create a categorical distribution from m
6: $action \leftarrow$ random sample from $distrib$
7: Calculate the log likelihood of $action$ w.r.t. $distrib$
8: Recommend $action$ as the knob setting
9: **end for**
10: $\hat{\mu} \leftarrow$ learned means from A_c for all cont. actions
11: $\hat{\sigma} \leftarrow$ learned variances from A_c for all cont. actions
12: $distrib \leftarrow$ multivariate normal distribution from $\hat{\mu}$ and $\text{diag}(\hat{\sigma})$
13: $actions \leftarrow$ random sample from $distrib$
14: Calculate the log likelihood of $actions$ w.r.t. $distrib$
15: Recommend $actions$ as the knob setting

The learning process detailed in Algorithm 3.3 is rather simple and closely follows Equations 3.4 and 3.5 in Section 3.3.1. Given the initial $State$, $Knobs$ recommended, resulting $Reward$, resulting $State'$ we update the weights of the policy π_θ (θ) and value function V_w (w). First, the advantage is calculated using Equation 3.3. The policy weight update $\Delta\theta$ follows Equation 3.4, using the log likelihoods calculated in Algorithm 3.2 multiplied by the advantage. The value function weight update Δw closely follows Equation 3.5, with the goal of minimizing the mean squared error between the predicted state value and the TD estimate. Since advantage is already the difference between the TD estimate and the state value, we can just square the advantage to calculate the MSE. For the actor network, we apply gradient ascent as

we wish to maximize the advantage received from taking a specific action in a specific state. With the critic network, we perform gradient descent as we wish to minimize the TD error.

Algorithm 3.3 Agent learning

Input: $State$, $Knobs$, $Reward$, $State'$, Policy π_θ with weights θ, Value function V_w with weights w

Output: Updated actor and critic network weights

1: $Advantage \leftarrow Reward + \gamma V_w(State')$ - $V_w(State)$
2: $\Delta\theta \leftarrow \nabla_\theta log\pi_\theta(Knobs|State) * Advantage$
3: $\Delta w \leftarrow \nabla_w V_w(State) * Advantage^2$
4: Update actor policy network weights: $\theta \leftarrow \theta + \alpha_\theta \Delta\theta$
5: Update critic value network weights: $w \leftarrow w - \alpha_w \Delta w$

The training process outlined in Algortihm 3.1 trains the model to minimize the optimizer's estimated cost of the workload. This is advantageous because IBM DB2 compiles a query execution plan and an estimated cost in a matter of milliseconds – in some cases applying a knob configuration takes longer such as having to resize memory pools. This speed allows the agent to interact with its environment and explore thousands of knob configurations very quickly, making the training very efficient.

While the estimated cost from the optimizer is a good indicator of the resources required for a executing a given query and the underlying knob configuration, it is measured in TIMERONs which is an abstract unit of measure that does not directly equate to execution time. Estimates can also be prone to inaccuracy due to factors such as incorrect cardinality estimation by the optimizer [7]. Furthermore, some configuration knobs are not factored into cost estimation, such as the degree of parallelism, as performing operations in parallel is not any less expensive resource-wise, but it can greatly reduce the resulting execution time of a query.

Finding a knob configuration that minimizes the execution time of a query work-

load is the goal of BLUTune. Training a model from scratch to learn a suitable knob configuration can take a long time when you must execute queries after every single training step to produce a new state and reward. This is apparent when tuning over a workload with complex OLAP queries and large data warehouses. This issue is further exacerbated by the fact that the initial knob configurations that the agent produces during training are rather poor as it has not explored enough yet, which can lead to execution times in hours. The agent needs to explore thousands of options meaning execution time can make training prohibitively expensive for large workloads.

We propose a multi-stage learning process that uses the idea of *Transfer Learning* to combine the benefits of using both the estimated cost and execution time of a query as our key performance metrics for training. Transfer learning focuses on storing knowledge gained while solving one task and apply it to solve a different, but related second task [22]. First, we train our model on the first task of minimizing the optimizer's estimated cost in each episode as shown in Algorithm 3.1. After some number of episodes, we switch from using estimated cost as the performance metric to the overall elapsed time of executing a query. The model initially trained on minimizing estimated cost is now solving a second different, but related task of minimizing the elapsed time of executing the queries. When we refer to execution time, we mean the *elapsed time* it takes to submit a query and receive a result. It is important to make this distinction because DB2's internal measure of execution time does not capture the time it took to compile or prepare a query for execution. Some knobs, such as query optimization level, can increase the amount of time it takes the query optimizer to compile a query execution plan as the optimizer considers more plan choices and computations. By using elapsed time we can get a more accurate

measure of how all knobs affect the DBMS as a whole.

We first train a model that has considered thousands of configurations for a number of queries in a relatively small amount of time using estimated cost. This model has a good, but incomplete understanding of the behaviour of most knobs. Leveraging the large amount of knowledge already gained on the first task, the second task of minimizing execution time is easier for the agent, since although the estimated costs do not directly equate execution time they are still a good indication of performance improvement from the knob configuration. This allows us to spend less time training on execution time and fine-tune our model to improve the knob recommendations. This process is nearly identical to that described in Algorithm 3.1, instead of retrieving the estimated cost from the QEP the query is executed and the execution time is used as the performance metric for the reward.

Chapter 4

Experimental Study

We present an experimental study of BLUTune for an *evaluation* of our techniques, a *scalability* study and an *effectiveness comparison*.

1. *Evaluation*. We evaluate our techniques: transfer learning and resource constraints.
2. *Scalability*. We demonstrate the scalability of BLUTune with respect to the number of queries trained on and the resulting training time and performance.
3. *Effectiveness Comparison*. We compare the effectiveness of BLUTune to existing IBM tools and other related works.

We conducted experiments over the synthetic OLAP TPC-DS benchmark [23], with 99 queries and varying database sizes of 1GB, 10GB and 100GB. All the tables in the database are column-organized. The tables and their respective cardinalities are listed in Table 4.1 for each database size.

The server running IBM DB2 version 11.5 was an 8-core Intel Xeon E5-2630 v3 2.4GHz CPU with 64 GB DDR4 RAM.

BLUTune was written in Python, and relies on the package PyTorch to facilitate machine learning.

Table name	1GB row count	10GB row count	100GB row count
INVENTORY	11,745,000	133,110,000	399,330,000
STORE_SALES	2,880,404	28,800,991	287,997,024
CATALOG_SALES	1,441,548	14,401,261	143,997,065
WEB_SALES	719,384	7,197,566	72,001,237
STORE_RETURNS	287,514	2,875,432	28,795,080
CUSTOMER_DEMO.	1,920,800	1,920,800	1,920,800
CATALOG_RETURNS	144,067	1,439,749	14,404,374
WEB_RETURNS	71,763	719,217	7,197,670
CUSTOMER	100,000	500,000	2,000,000
CUSTOMER_ADDR.	50,000	250,000	1,000,000
ITEM	18,000	102,000	204,000
TIME_DIM	86,400	86,400	86,400
DATE_DIM	73,049	73,049	73,049
CATALOG_PAGE	11,718	12,000	20,400
HOUSEHOLD_DEMO.	7,200	7,200	7,200
PROMOTION	300	500	1,000
WEB_PAGE	60	200	2,040
STORE	12	102	402
REASON	35	45	55
WEB_SITE	30	42	24
CALL_CENTER	6	24	30
INCOME_BAND	20	20	20
SHIP_MODE	20	20	20
WAREHOUSE	5	10	15

Table 4.1: Cardinality (row count) of each table across three different sizes of TPC-DS benchmark databases.

4.1 Evaluation of Our Techniques

4.1.1 Transfer learning evaluation

A key contribution of our work is using the idea of transfer learning: first train a model on the objective of minimizing the estimated cost from the optimizer, then using the first model begin training on the goal of minimizing execution time.

For experimentation, 50 of the most computationally expensive queries were selected from the TPC-DS workload. These queries are divided into a 80/20 training/testing split, meaning 40 queries are used for training and 10 are used for testing. From these 40 queries for training, they are further split 70/30 with 28 used for training on estimated cost and 12 for training on execution time.

Exp-1: Transfer learning for knobs considered by the optimizer cost estimates. Many configuration parameters (such as sortheap, bufferpool size, etc) are captured by the optimizer cost model and tuning them directly impacts query execution plan generation and estimated costs. Although the estimated cost is an artificial measure in TIMERONs and does not directly represent the expected execution time, reducing the estimated cost of a query is still a good idea and can lead to faster execution times.

We demonstrate this approach by following the methodology outlined at the beginning of this section and tuning on two continuous knobs (sortheap & bpsize) and one discrete knob (query optimization level).

The cumulative latency (execution time) of the testing queries for three configurations is reported in Figure 4.1. Using the default configuration for DB2 results in a cumulative latency of 1067.4 seconds. Training a model on 28 queries where the goal

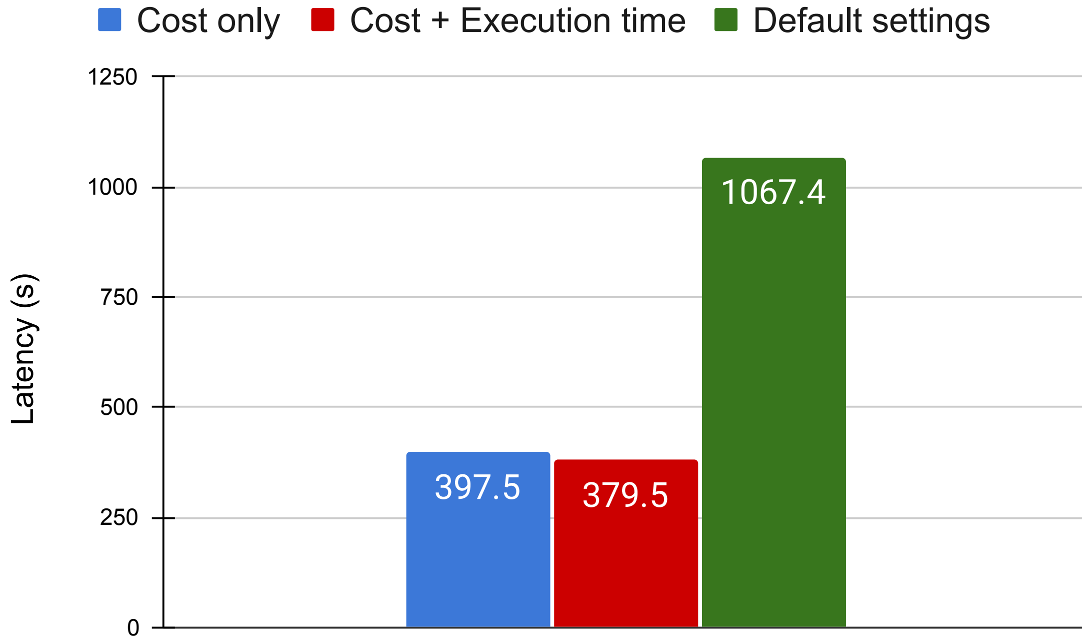


Figure 4.1: Latency comparison between training only on cost and training on both cost and execution time (transfer learning) for knobs well-represented by the optimizer.

is to minimize the optimizer's estimated cost leads to a latency of 397.5 seconds, approximately a 63% improvement over the default settings. By using transfer learning, we further train the existing model on the task of minimizing execution time. This results in a configuration that has a cumulative latency of 379.5 seconds, approximately a 5% improvement on the first learned configuration and a 64% improvement over the default settings.

The reported results demonstrate that training a model on the task of minimizing the estimated cost of the query can in fact lead to lower execution times. When the tuned knobs are accurately considered in the cost estimates the agent can learn a suitable configuration off cost alone. Employing transfer learning in this scenario

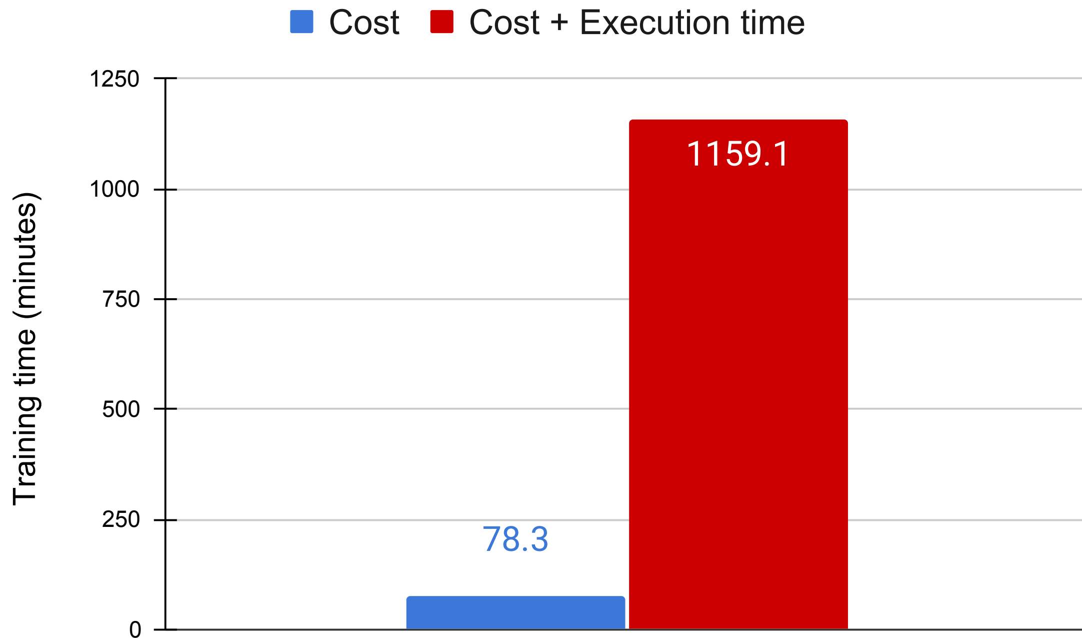


Figure 4.2: Training time comparison between training only on cost and training on both cost and execution time (transfer learning) for knobs well-represented by the optimizer.

does offer some benefit, as fine-tuning the model on the task of minimizing execution time does come with a 5% improvement. However, training on execution time can be a much longer process as it requires the queries to be executed thousands of times. This is shown in Figure 4.2, where 78.3 minutes (1.3 hours) was spent training on minimizing the cost of 28 queries. Further training the model on execution time for 12 queries takes another 1080.8 minutes for a total of 1159.1 minutes (19.3 hours), a massive increase over cost alone. The large training time is caused by the execution of the queries themselves, also only a limited number of knobs were tuned, meaning the performance could be much better. In scenarios like this, it is up for the DBA to decide if the additional time of transfer learning is worth it for the performance

improvement.

Exp-2: Transfer learning for knobs not considered by the optimizer. While many knobs are captured in the optimizer cost estimates, some are not. A high-profile knob that can improve performance but is not reflected in the cost estimates in DB2 is *dft_degree*, the default degree of intrapartition parallelism within queries. The query degree does not reduce the estimated cost since the amount of work needed to be done does not change, but rather the query is divided into parts and certain operations are performed in parallel leading to faster execution times.

To demonstrate how our approach handles knobs that are not accurately reflected in estimates, we take the same approach as in Exp-1 however in addition to tuning two continuous knobs (sortheap & bpsize) and one discrete knob (query optimization level) we also tune another discrete knob: degree of parallelism. We use the same testing and training sets from Exp-1 as a basis for comparison.

The cumulative latency of the testing queries for four configurations is reported in Figure 4.3. Once again, using the default configuration for DB2 results in a latency of 1067.4 seconds. Training a model only on the cost estimates leads to a cumulative latency of 399.6 seconds, a 63% improvement. This improvement is largely attributed to the three knobs that are reflected by the estimates, as the resulting configuration is similar to that in *Exp-1*. Further training the model on the task of minimizing execution time results in a latency of 168.6 seconds, a 58% improvement over cost-only, for a total improvement of 84%. For comparison, training all 40 queries only on the task on minimizing execution time results in a cumulative latency of 157.1 seconds, a 7% improvement over transfer learning and a 85% improvement overall.

In this scenario, transfer learning has a larger benefit than in Exp-1. Since the

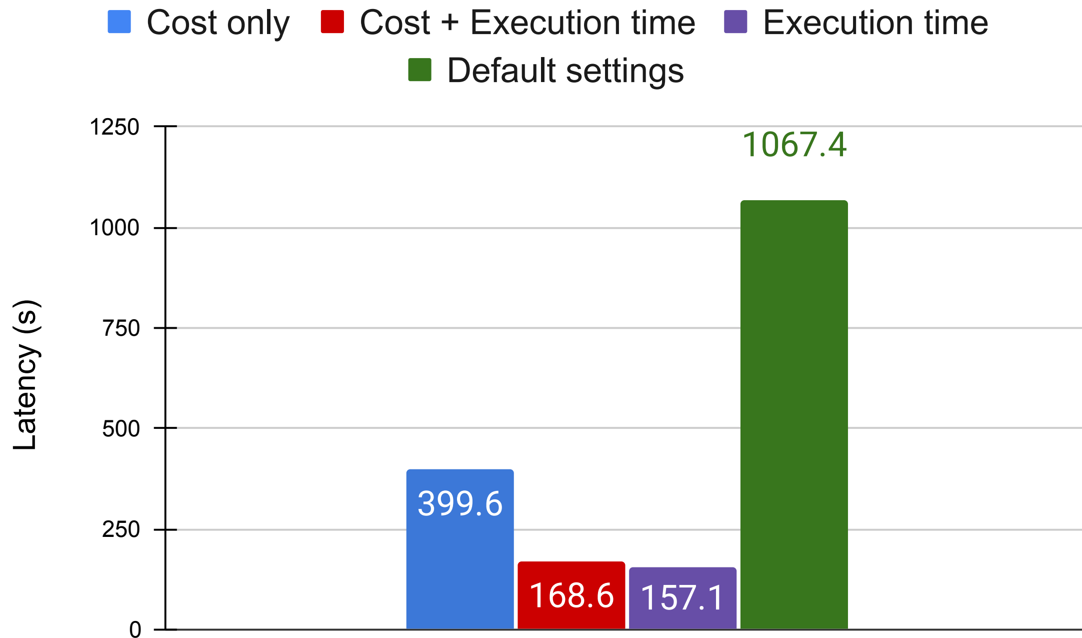


Figure 4.3: Latency comparison between training only on cost, on both cost and execution time (transfer learning), on execution time only and using the default settings

degree of parallelism is not reflected in the estimates there is no information for the agent to learn about that particular knob in the first phase of training. Training on execution time in the second phase is required to capture information pertaining to the knob and how it impacts the query performance.

Having to train on execution time does negatively impact the training time, but in this scenario it is required and likely worth the extra time spent training to see increased performance. Figure 4.4 shows the time it took to train the specific models. Training using the cost estimates takes only 88.2 minutes (1.47 hours) but fails to learn an understanding of the query degree knob. Transfer learning takes an additional 371.5 minutes for a total of 460.1 minutes (7.67 hours). Since learning the degree of parallelism knob requires training over execution time a DBA likely would not want

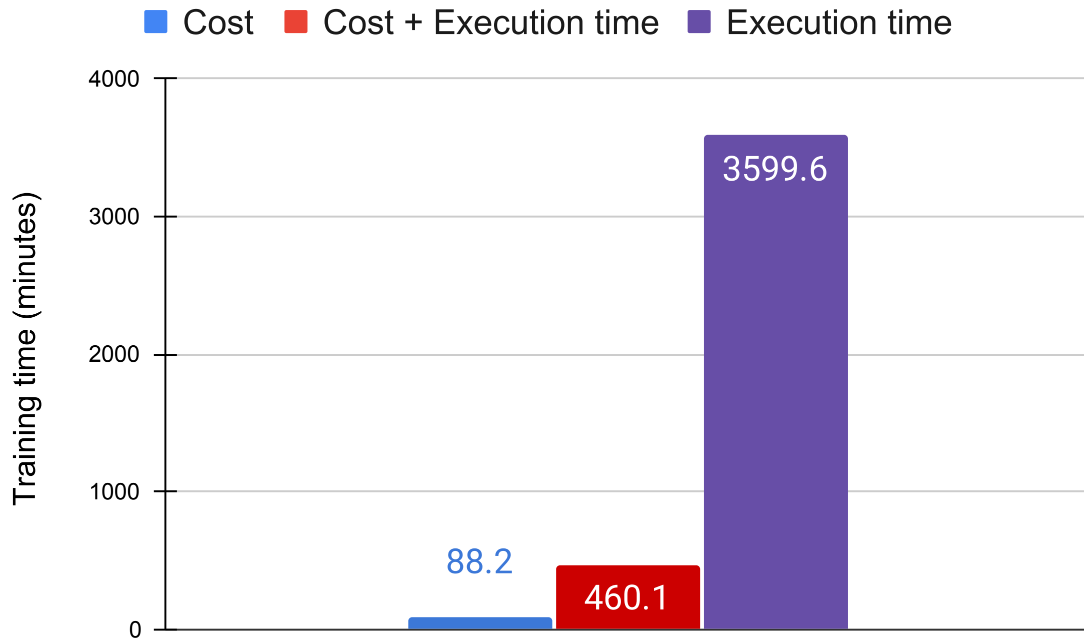


Figure 4.4: Training time comparison between training only on cost, on both cost and execution time (transfer learning) and on execution time only

to use the cost-only model. We compare the time of our transfer learning approach with that of training only on execution time for the full 40 episodes, which took 3599.6 minutes (60 hours), a 682% increase in training time!

These experiments demonstrate that our transfer learning approach of first training on minimizing the estimated cost then switching to minimizing execution time can produce effective configurations in a reasonable amount of time. Knobs that are considered in the optimizer's cost estimates and also knobs that are not can both be effectively learned by our approach.

4.1.2 Resource constraints

Database systems do not have an infinite amount of resources to pull from, they are often constrained in terms of the available memory, number of CPU cores, etc. Without relaying a constraint to the agent, it is free to try and learn any configuration, even if it allocates an unrealistic amount of resources. Our agent is trained to learn actions within user-specified resource constraints through our resource reward function detailed in Section 3.5.1.

Exp-3: Training under resource constraints. To demonstrate the effectiveness of our approach, we select the sortheap and bpsize knobs which both allocate 4KB pages from the database memory heap. We train our agent to produce only these two actions. We place a constraint on the memory usage by specifying a resource limit in our reward function. Figure 4.5 shows the resulting combined sortheap and bpsize usage for the same set of queries with two different resource limits: 50,000 and 250,000 4KB pages. Training under a memory constraint of 50,000 pages resulted in an average memory allocation of 30,400 pages, with the maximum being 35,774 pages. A memory constraint of 250,000 pages resulted in an average memory allocation of 89,658 pages, with a maximum of 100,873 pages. These averages both come in under their limits, with the limit of 250,000 resulting in a configuration well below. This is acceptable as we do not want our agent allocating the whole resource limit if it offers little benefit over the previous actions.

This shows that our agent is effective at training and producing tuning recommendations within user-specified resource constraints.

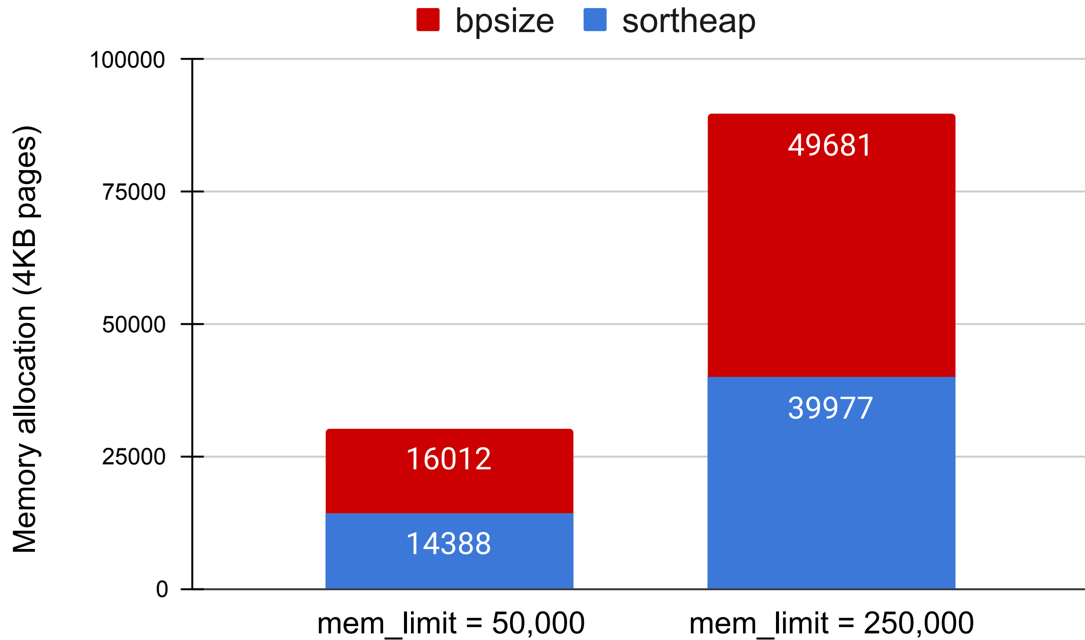


Figure 4.5: Memory allocation in 4KB pages for two different memory constraints, 50000 and 250000.

4.2 Scalability

BLUTune was designed to tune and produce knob configuration for large, complex analytical workloads. To demonstrate how our approach scales alongside the number of rows in the database, we generate three different TPC-DS workloads with differing sizes: 1GB, 10GB and 100GB. The row counts for each database is listed in Table 4.1.

Exp-4: Scalability of BLUTune. We trained our agent to tune four knobs (sortheap, bpsize, query degree and optimization level) for the three separate database sizes following the same setup as *Exp-2*. The resulting training times of both the cost-only phase and the execution time phase is shown for each database size in Figure 4.6.

Across all three database sizes, the time spent on the first phase of training solely

to minimize estimated cost is roughly the same. The time required to train on cost is independent of the size of the database, this is because we use estimates and not the actual execution time. The estimates are generated by the optimizer in milliseconds and training is instead bottle-necked by the time it takes to apply a new configuration (e.g. resizing a memory pool). The actual discrepancy in time is due to the randomness and certain convergence conditions, leading to earlier termination. Training on the cost estimates allows for our agent to quickly learn a suitable configuration regardless of database size. The better the learned configuration is from the cost estimates, the less time the agent will spend on the second phase involving actual query execution.

The second phase of training requires queries to be executed and their resulting execution time observed. The time spent between executing a query and receiving a result takes a lot longer than fetching a cost estimate. As the size of the database increases, the estimated cost and execution time of queries grows with it. It is natural to assume that a specific query will take longer to execute over a 100GB database rather than a 10GB variation of it. As a result, the second phase of training is not independent of the size of the database, as shown in Figure 4.6. Training over the 1GB database took 167.2 minutes, increasing the size by ten times to 10GB only results in a total training time of 195.4 minutes, only a 17% increase. Increasing the database size by a factor of 10 again to 100GB results in 460.1 minutes of training time, only a 135% increase. The training time scales sub-linearly with the size of the database.

BLUTune has a unique training process that scales well to large databases. The first phase of training (minimizing cost) is invariant to the size of the database and

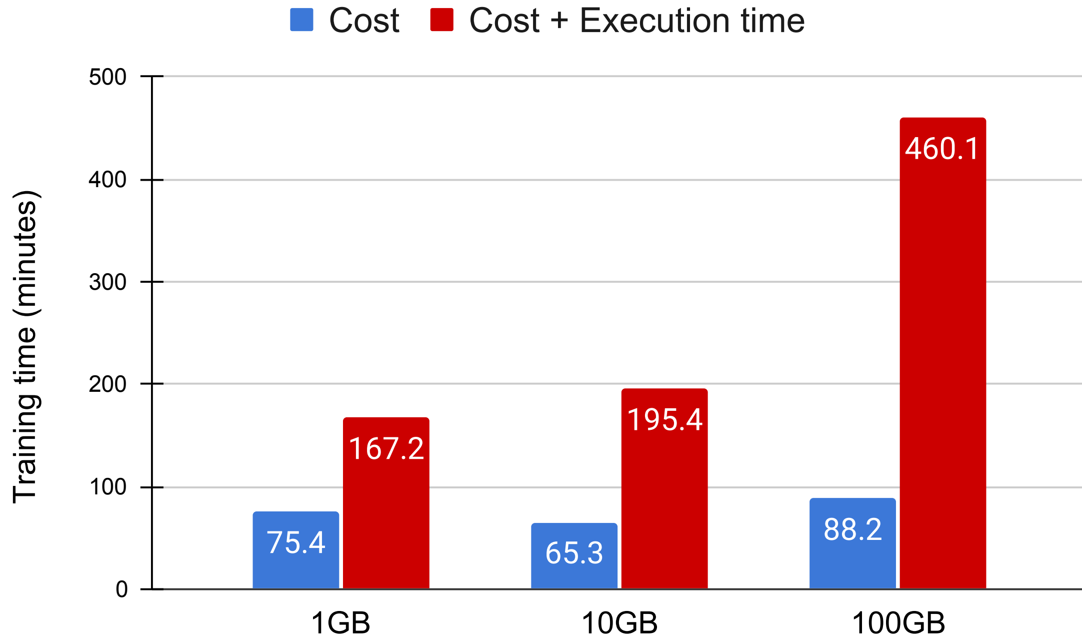


Figure 4.6: Time spent training over three different sizes of the TPC-DS benchmark

the intermediate learned configuration is a suitable starting point for the second phase of training (minimizing execution time) to reduce the overall time spent.

4.3 Effectiveness Comparison

Exp-5: Effectiveness Comparison of BLUTune. To demonstrate and test the effectiveness of BLUTune and the knob configurations it produces, we compare our configuration against those produced by existing IBM tools and QTune, which is a related work most similar to our approach.

IBM DB2 is shipped with a very basic initial knob configuration, which is suitable for small or new database installations, however when working with complex databases and workloads it is necessary to tune these knobs to achieve satisfactory

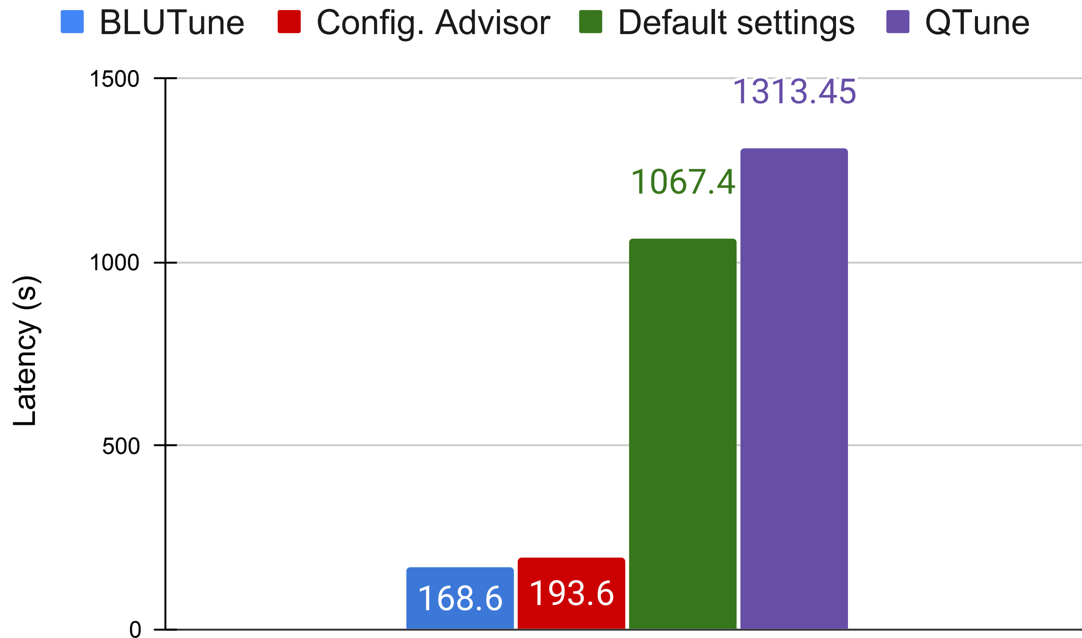


Figure 4.7: Effectiveness comparison with existing IBM tools and QTune

performance, as shown in Figure 4.7 the default settings results in a total execution time of 1067.4 seconds over the testing set. The IBM DB2 Configuration Advisor can automatically recommend configurations based on heuristics and database statistics. For evaluation, we took only the recommendations for the same four knobs that BLUTune tuned, this resulted in a latency of 193.6, a 13% increase over the learned configuration.

QTune is the most similar to our work, as we both take a Deep Reinforcement Learning-based approach with a focus of using query execution plans as a means to represent the queries. The authors of QTune did not make their full approach available so an approximation of their system was made. We utilized their implementation of query representation, their reward function, and trained the system following the

same approach as BLUTune. We found that QTune's reward function did not perform correctly under our TPC-DS workload. During both training on cost and also training on execution time, the magnitudes of the reward were extremely high - leading to exploding gradients. As a result, no effective configurations could be learned, as shown in Figure 4.7 it resulted in 1313.45 seconds. We attempted to address this issue by modifying their reward function such that the magnitude and variance was reduced. However, it still did not act as an effective reward signal as the agent had trouble converging. The lack of convergence led to a total training time of 24.5 hours compared to BLUTune's 7.6 hours. QTune was originally evaluated over OLTP workloads, meaning the execution time and costs of the queries were likely low, it is possible that their approach does not extend to complex OLAP workloads, which was the main focus of BLUTune.

Chapter 5

Future Work

BLUTune is a major step towards solving automatic knobs tuning for IBM DB2. Early on we identified many tasks we felt should be completed in order to reach a fully automatic knob-tuning system. Some of these included workload forecasting, improved query representation through graph embeddings, and asynchronous actor-critic for parallel training.

Database tuning is often a *reactive* task, meaning it is not performed until database performance issues arise. Throughout this paper, we identified that different queries or workloads do not always benefit in the same way from specific knob configurations. Additionally, businesses and their databases may have many different workloads that are ran over the course of a day, weeks or even months. It would be very beneficial if BLUTune was able to predict, or forecast, upcoming changes to the workload and *proactively* tune the database before any performance problems can arise. Ma *et al.* [24] outline various patterns that exist within business query workloads and applications (cycles, growth & spikes and workload evolution), and propose a query forecaster using an ensemble of Linear Regression and RNN (specifically LSTMs).

We believe a similar approach would improve the effectiveness of BLUTune as a fully automatic knobs-tuning system.

The query representation used by BLUTune is sufficient in capturing the types of operators involved in a query execution plan and their estimated costs. However, it is a high-level summary of the query execution plan, and fails to capture some useful information such as the structure of table joins and the structure of the plans themselves. This information can help improve the characterization and representation of our query execution plans. Query execution plans are represented by a Query Graph Model (QGM) which is a graph of plan operators. It is desirable to retain the structure of these graphs in our query representation such that similar queries have similar representatives. This would allow our reinforcement learning agent to have a better understanding of the queries themselves, and correctly recall previously learned knob configurations for similar queries in the future. A potential approach would be building on the idea of *doc2vec*, which is a Natural Language Processing tool for representing documents a vector[25]. However, instead of a document we have a QGM, and instead of words we have sub-graphs of plan operators. If two queries contain the same sub-graphs and operators, then they are likely similar behaving queries that hopefully benefit similarly from the same knob configuration. Furthermore, this would allow for the possibility of clustering similar queries into self-identified workloads. These clusters can be tuned for separately in order to maximize performance.

BLUTune's training process is greatly lengthened due to the time required to execute queries during the execution phase of our transfer learning approach. Several parts of BLUTune could be parallelized in order to speed up the training process. For example, in our training approach BLUTune tackles one query at a time. The

order in which the system encounters the queries and learns from them largely does not matter. Advantage Actor-Critic (A2C) has an extension called Asynchronous Advantage Actor-Critic (A3C) devised by Google DeepMind [26]. A2C has a single agent which interacts with the environment, A3C can support many agents concurrently to speedup training. Each agent has their own neural network and copy of the environment, a database instance in this case, and the experience of each agent is independent of the experience of others. A global network is maintained such that the gradient updates of each individual agent are applied to the global network, which makes the final predictions. BLUTune was not initially designed to be able to tune queries that execute concurrently, a possible extension could be multiple agents tuning different queries over the same environment to simulate a concurrent database environment.

Chapter 6

Conclusion

In this work we created a system, BLUTune, to solve the problem of automatic knobs 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 maximizes 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 provides insight on how a query behaves and also captures 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 allows for fine-tuning of the the original model to improve the 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 to a truly fully automated tuning solution for IBM DB2.

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 *BTW 2003, Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW-Konferenz, 26.-28. Februar 2003, Leipzig*, G. Weikum, H. Schöning, and E. Rahm, Eds., ser. LNI, vol. P-26, GI, 2003, pp. 620–629. [Online]. Available: https://dl.gi.de/20.500.12116/30086.

[2] P. Belknap, B. Dageville, K. Dias, and K. Yagoub, "Self-tuning for sql performance in oracle database 11g," in *2009 IEEE 25th International Conference on Data Engineering*, 2009, pp. 1694–1700. DOI: 10.1109/ICDE.2009.165.

[3] *Configuration parameters summary*, https://www.ibm.com/docs/en/db2/11.5?topic=parameters-configuration-summary, Accessed: 2021-10-04.

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

[5] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback, "Self-tuning database technology and information services: From wishful thinking to viable engineering," in *Proceedings of 28th International Conference on Very Large Data Bases,*

VLDB 2002, Hong Kong, August 20-23, 2002, Morgan Kaufmann, 2002, pp. 20–31. DOI: 10.1016/B978-155860869-6/50011-1. [Online]. Available: http://www.vldb.org/conf/2002/S02P02.pdf.

[6] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer, "Using probabilistic reasoning to automate software tuning," in *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004, June 10-14, 2004, New York, NY, USA*, E. G. C. Jr., Z. Liu, and A. Merchant, Eds., ACM, 2004, pp. 404–405. DOI: 10.1145/1005686.1005739. [Online]. Available: https://doi.org/10.1145/1005686.1005739.

[7] G. Damasio, P. Mierzejewski, J. Szlichta, and C. Zuzarte, "Query performance problem determination with knowledge base in semantic web system optimatch," in *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*, E. Pitoura, S. Maabout, G. Koutrika, A. Marian, L. Tanca, I. Manolescu, and K. Stefanidis, Eds., OpenProceedings.org, 2016, pp. 515–526. DOI: 10.5441/002/edbt.2016.49. [Online]. Available: https://doi.org/10.5441/002/edbt.2016.49.

[8] ——, "Optimatch: Semantic web system for query problem determination," in *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, IEEE Computer Society, 2016, pp. 1334–1337. DOI: 10.1109/ICDE.2016.7498338. [Online]. Available: https://doi.org/10.1109/ICDE.2016.7498338.

[9] G. Damasio, V. Corvinelli, P. Godfrey, P. Mierzejewski, A. Mihaylov, J. Szlichta, and C. Zuzarte, "Guided automated learning for query workload re-optimization,"

Proc. VLDB Endow., vol. 12, no. 12, pp. 2010–2021, 2019. DOI: 10.14778/3352063.3352120. [Online]. Available: http://www.vldb.org/pvldb/vol12/p2010-damasio.pdf.

[10] G. Damasio, S. Bryson, V. Corvinelli, P. Godfrey, P. Mierzejewski, J. Szlichta, and C. Zuzarte, "GALO: guided automated learning for re-optimization," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 1778–1781, 2019. DOI: 10.14778/3352063.3352064. [Online]. Available: http://www.vldb.org/pvldb/vol12/p1778-damasio.pdf.

[11] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra, "Adaptive self-tuning memory in db2," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, ser. VLDB '06, Seoul, Korea: VLDB Endowment, 2006, pp. 1081–1092.

[12] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood, "Automatic performance diagnosis and tuning in oracle," in *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, www.cidrdb.org, 2005, pp. 84–94. [Online]. Available: http://cidrdb.org/cidr2005/papers/P07.pdf.

[13] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin, "Automatic SQL tuning in oracle 10g," in *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, Morgan Kaufmann, 2004, pp. 1098–1109. DOI: 10.1016/B978-012088469-8.50096-6. [Online]. Available: http://www.vldb.org/conf/2004/IND4P2.PDF.

[14] S. Duan, V. Thummala, and S. Babu, "Tuning database configuration parameters with ituned," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 1246–1257, Aug. 2009, ISSN: 2150-8097. DOI: 10.14778/1687627.1687767. [Online]. Available: https://doi-org.uproxy.library.dc-uoit.ca/10.14778/1687627.1687767.

[15] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," ser. SIGMOD '17, Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 1009–1024, ISBN: 9781450341974. DOI: 10.1145/3035918.3064029. [Online]. Available: https://doi.org/10.1145/3035918.3064029.

[16] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: Tapping the performance potential of systems via automatic configuration tuning," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17, Santa Clara, California: Association for Computing Machinery, 2017, pp. 338–350, ISBN: 9781450350280. DOI: 10.1145/3127479.3128605. [Online]. Available: https://doi.org/10.1145/3127479.3128605.

[17] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li, "An end-to-end automatic cloud database tuning system using deep reinforcement learning," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19, Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 415–432, ISBN: 9781450356435. DOI: 10.1145/3299869.3300085. [Online]. Available: https://doi-org.uproxy.library.dc-uoit.ca/10.1145/3299869.3300085.

[18] G. Li, X. Zhou, S. Li, and B. Gao, "Qtune: A query-aware database tuning system with deep reinforcement learning," *Proc. VLDB Endow.*, vol. 12, no. 12,

pp. 2118–2130, Aug. 2019, ISSN: 2150-8097. DOI: 10.14778/3352063.3352129. [Online]. Available: https://doi.org/10.14778/3352063.3352129.

[19] R. C. Marcus and O. Papaemmanouil, "Plan-structured deep neural network models for query performance prediction," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1733–1746, 2019. DOI: 10.14778/3342263.3342646. [Online]. Available: http://www.vldb.org/pvldb/vol12/p1733-marcus.pdf.

[20] Y. Gur, D. Yang, F. Stalschus, and B. Reinwald, "Adaptive multi-model reinforcement learning for online database tuning," in *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, Y. Velegrakis, D. Zeinalipour-Yazti, P. K. Chrysanthis, and F. Guerra, Eds., OpenProceedings.org, 2021, pp. 439–444. DOI: 10.5441/002/edbt.2021.48. [Online]. Available: https://doi.org/10.5441/002/edbt.2021.48.

[21] X. Zhang, H. Wu, Z. Chang, S. Jin, J. Tan, F. Li, T. Zhang, and B. Cui, "Restune: Resource oriented tuning boosted by meta-learning for cloud databases," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds., ACM, 2021, pp. 2102–2114. DOI: 10.1145/3448016.3457291. [Online]. Available: https://doi.org/10.1145/3448016.3457291.

[22] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu, "A survey on deep transfer learning," in *Artificial Neural Networks and Machine Learning - ICANN 2018 - 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part III*, V. Kurková, Y. Manolopoulos, B. Hammer, L. S. Iliadis, and I. Maglogiannis, Eds., ser. Lecture Notes in

Computer Science, vol. 11141, Springer, 2018, pp. 270–279. DOI: 10.1007/978-3-030-01424-7_27. [Online]. Available: https://doi.org/10.1007/978-3-030-01424-7%5C_27.

[23] M. Pöss, R. O. Nambiar, and D. Walrath, "Why you should run TPC-DS: A workload analysis," in *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C.-C. Kanne, W. Klas, and E. J. Neuhold, Eds., ACM, 2007, pp. 1138–1149. [Online]. Available: http://www.vldb.org/conf/2007/papers/industrial/p1138-poess.pdf.

[24] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, "Query-based workload forecasting for self-driving database management systems," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds., ACM, 2018, pp. 631–645. DOI: 10.1145/3183713.3196908. [Online]. Available: https://doi.org/10.1145/3183713.3196908.

[25] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, ser. JMLR Workshop and Conference Proceedings, vol. 32, JMLR.org, 2014, pp. 1188–1196. [Online]. Available: http://proceedings.mlr.press/v32/le14.html.

[26] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement

learning," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, M.-F. Balcan and K. Q. Weinberger, Eds., ser. JMLR Workshop and Conference Proceedings, vol. 48, JMLR.org, 2016, pp. 1928–1937. [Online]. Available: http://proceedings.mlr.press/v48/mniha16.html.