BOSTON UNIVERSITY

METROPOLITAN COLLEGE

Master's Thesis

QUERY PLANNING FOR STREAM PROCESSING

by

APOORVA TAMASKAR

M.Sc., University of Glasgow, 2018
B.Sc. Hons, Chennai Mathematical Institute , 2017

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2020

Approved by

First Reader

Kia Teymourian, PhD
Assistant Professor of computer science

Second Reader

Eugene Pinsky, PhD
Assistant Professor of Computer Science

Third Reader

Reza Rawassizadeh, PhD
Associate Professor of Computer Science

Acknowledgments

Over the course of writing this thesis, I have received lots of support and assistance. First of all, I would like to thank my supervisor, Professor Kia Teymourian, who guided me and helped me formulate the research question, provided me with sources and methodology. Your insightful feedback pushed me to sharpen my thought process and helped me elevate the quality of my work.

I would like to thank the administrative staff at MET college, especially Ronette Lyle and the BU ISSO team who helped me through the administrative process.

I would like to acknowledge my friends and colleagues at Boston University for their wonderful feedback and discussions.

Also, I would like to thank my parents and my brother for their wise counsel, an empathetic ear and for always being there for me.

Apoorva Tamaskar

Metropolitan College

QUERY PLANNING FOR STREAM PROCESSING

APOORVA TAMASKAR

ABSTRACT

Our increasingly connected world has resulted in the generation of many data streams. With the rise in network traffic and increased interconnectivity, analyzing data and extracting key information has become challenging. Executing queries to extract insights from the data is now more essential than ever to make quick and efficient decisions. The flexibility of reinforcement learning, along with strong prediction properties, and precise mathematical model of Deep neural networks combined results in the technique called Deep reinforcement learning which can be a robust and promising method to reduce query processing times. While there is a lot of research on query optimization on data streams, none showcase the use of Deep reinforcement learning. The continuous nature of datastreams renders the conventional approaches inapplicable, due to increased processing time. The method implemented in this thesis is generic enough so that it applies to a wide range of data streams and queries to optimize the query processing time. The goal of this study is to act as a proof of concept. The thesis explores the use of deep reinforcement learning on a query from Linear road benchmark data. The proposed method is demonstrated to be effective in picking optimal query plan, and reducing the query execution time. We hope the work in this thesis encourages others to work on the development of a robust and more generalized system, capable of optimizing any query on any data stream.

Contents

Chapter 1

Introduction

In recent years, the ability to gather data has increased tremendously due to various sensory devices and the cheap cost of data storage. Some examples of data gathering and generating sources are:-

- **Finance**, Stock market, the continuous flow includes the price of any given stock at any time.
- **Network Management**, Network administrators need to continuously monitor the traffic flow on their network as they need to help maintain a level of Quality of Service for their clients.
- **Healthcare**, Monitoring the vitals of the patients is an important and continuous process.
- **National Security** Monitoring airspace, getting continuous information from various sources and inferring results from them.

For these examples, it is clear that there exists a need to continuously keep processing data stream and reporting back various statistics and any changes.

The increasingly complex nature of queries, high rate of data generation, and the need to have answers quickly pose a challenge for many businesses. The global growth in traffic makes finding an appropriate approach to optimize query execution prohibitively difficult. This increased traffic puts a tremendous amount of strain

on network equipments and can often lead to developers having to make choice as to whether reduce the accuracy of the results or to reduce the number of times an answer is returned to the user.

1.1 Deep reinforcement learning

Deep reinforcement learning is a mathematical model that is the combination of neural networks and reinforcement learning. Deep reinforcement learning can be thought of as a advanced version of traditional reinforcement learning, this better performance is achieved with the use of neural networks.

To get and intuitive idea for reinforcement learning, one can imagine playing a game without an opponent 1 move at a time. At each turn the player is required to make a move, the reinforcement learning model can estimate the probability with which each move will lead to the optimal outcome based on the game.

Simplicity and flexibility of reinforcement learning make it a very powerful tool to tackle problems with.

1.1.1 Applications

In recent time there are many applications of Deep reinforcement learning agents, a few of which are:-

- Self driving cars.
- Training bots to play games.
- Protein folding.
- Online recommendation systems, e.g. YouTube videos recommendation.

1.1.2 Properties

It is clear that deep reinforcement learning has a wide variety of applications. There is a tremendous amount of research being done in the area of deep reinforcement learning, hence we believe it to be a powerful tool to use. An important aspect of deep reinforcement learning comes from the fact that they use deep neural networks. The following is the theorem that states the possible accuracy neural networks can achieve if given enough examples.

Universal Approximation Theorem ($L1$ distance, ReLU activation, arbitrary depth, minimal width). For any Bochner-Lebesgue $p-$integrable function $\mho : \mathbb{R}^n \to \mathbb{R}^m$ and $\epsilon > 0$ there exists a fully-connected ReLU network F width exactly $d_m = \max\{n+1, m\}$ satisfying

$$\int_{\setminus^m} ||\mho(x) - F(x)||^p dx < \epsilon$$

1.1.3 Advantages

The approximation theorem essential means if given enough examples and time for training, the neural network can minimize the loss function to any degree near 0.

1.2 Research Problem

Traditional database management systems are designed to be efficient for solving queries on static data. They are unable to handle updates, deletion, and change in the data very well, which is a rather common occurrence in Data streams. The static dataset can be exploited to create various optimizations, these rules look at the schema of the data and the query being executed.

DBMS also tries to optimize the query execution by calculating various statistics about the data and make heuristic decisions based on them.

Optimizing the query execution plan is a difficult problem and can be proven to be $NP-$hard, i.e. there is an exponentially large search space and there is no known way to navigate efficiently.

Datastream management systems, on the other hand, do not have all the data available at once. They only have a chunk of data available to work with, which is specified using the window function. Datastream management systems are more complex by design as they have extra constraints. They often have multiple service quality metrics to meet, which puts additional constraints on them such as (Chakravarthy and Jiang, 2009) :-

- Achieve the maximal performance using bounded amount of resources.
- Be aware of unexpected overload situations.
- Guarantee user or application-specified QoS requirements for a query.
- Be implemented easily, and run efficiently with a low overhead.

A large number of systems around the world are generating continuous streams of data. Additional hardware is not enough to scale for the needs for the high throughput. The methods used by traditional database management systems are not scalable for datastreams. Whereas deep reinforcement learning is proven to be a powerful tool to help model and optimize various problems.

Some of the problems which Data streams face compared to traditional static data are (Chakravarthy and Jiang, 2009):-

- The continuous nature of data streams. In many cases the data first needs to be preprocessed and only then can an optimizer use it.
- Low latency is difficult to achieve. This is due to the optimizer needing preprocessed data.

- Gathering training data is difficult, training itself can be expensive.

1.3 Proposed Solution

We propose a method to gather enough training data use it for training to achieve low latency on queries.

When enough data, as defined by the windowing function, is taken as input we execute the query with multiple query plans. For each query plan records the time taken, the number of operations required to complete the execution, and few statistics about the data. Need to be careful of the statistics chosen and experiment with ones that do not require us to go over the data more than once.

We use this gathered data for training a Deep reinforcement learning model, to predict the optimal query plan and help improve our latency for executing the query.

1.4 Research Objective

The objective of this thesis is to act as a proof of concept for the application of Deep reinforcement learning for optimizing query plans for data streams. The proposed method should be general enough for its application to any other use case which can perhaps benefit from the use of deep reinforcement learning. To assess whether this method is useful, we will explore a query on the linear road benchmark data set.

This can be captured in the following research question:-

Is it possible to apply deep reinforcement learning to optimize query processing?

To answer the above question, it is helpful to understand the areas in which a Data stream system can be optimized. We look at each section individually and ask the question for the specific section. Optimization should be both safe and profitable.

Optimization is safe if it can be applied to a stream query without changing what it computes, as determined by the users requirements. An optimization is profitable if it makes the stream query faster, as measured by metrics that matter to the user, such as throughput, latency, or resource efficiency. (Martin Hirzel, 2018)

- **Batching**, reduces over-head by processing multiple data items together. (Carney et al., 2003), (Gordon et al., 2006), (Welsh et al., 2001)
- **Fusion**, combines smaller operators into a larger one, to avoid the overheadof data serialization and transport. (Gordon et al., 2006), (Tatbul et al., 2003), (Khandekar et al., 2009)
- **Placement**, assigns operators to hosts and cores to reduce communication costs or better utilize available resources. (Gordon et al., 2006), (Pietzuch et al., 2006), (Wolf et al., 2008)
- **State sharing**, attempts to avoid unnecessary copies of data. (Brito et al., 2008), (Arasu et al., 2006), (Sermulins et al., 2005)
- **Operator separation**, Operator separation splits a large computation into smaller steps. (Garcia-Molina et al., 2002), (Yu et al., 2009), (Ottoni et al., 2005)
- **Operator reordering**, A reordering optimization moves more selective operators, which reduce the data volume, upstream. (Garcia-Molina et al., 2002), (Graefe, 1990), (Avnur and Hellerstein, 2000)
- **Redundancy elimination**, eliminates superfluous computations. (Forgy, 1991), (Chen et al., 2000), (Pietzuch et al., 2006)
- **Load Balancing**, attempts to distribute workload evenly across resources.

(Gordon et al., 2006), (Arpaci-Dusseau et al., 1999), (Caneill et al., 2016), (Amini et al., 2006)

- **Algorithm selection**, uses a different algorithm to implement operator. (Garcia-Molina et al., 2002), (Welsh et al., 2001), (Wolf et al., 2008), (Abadi et al., 2005)
- **Load Shedding**, copes with high load by dropping data itemsto process. (Tatbul et al., 2003), (Gedik et al., 2008)
- **Fission**, often referred to as data parallelism, attempts to process multiple data items in parallel by replicating an operator. (Gordon et al., 2006), (Schneider et al., 2015), (Brito et al., 2008)

While the above-mentioned categories for optimization are not independent, this thesis focuses solely on operator reordering using deep reinforcement learning.

1.4.1 Operator reordering

A reordering optimization moves selective operators, which reduces the data volume, upstream. This has the benefit of reducing the data flowing into downstream computation, thus eliminating unnecessary work. However, care must be taken to preserve the desired semantics, and operators should only be re-ordered if the operations are commutative and their order of execution can be changed if they are associative. (Martin Hirzel, 2018)

The question to ask for operator reordering is

Is it possible to learn a model that predicts the optimal ordering, defined by a metric of choice, of operators for a given query on data streams using a deep reinforcement learning model trained on historical data?

1.5 Structure of thesis

The overall structure of the thesis chapter wise is as follows:-

- 2^{nd}:- Gives an in-depth view of the pipeline used by the current state of art technology for query optimization in traditional data bases including the mathematical knowledge for simplification and the overall framework.
- 2^{nd}:- Introduces the reader to data stream and how data management systems are used for them (called DSMS). We also explain few challenges DSMS face.
- 2^{nd}:- Introduced the reader to the mathematics required for Deep reinforcement learning and the algorithm for DQN.
- 3^{rd}:- Introduce the reader to stream optimization.
- 3^{rd}:- Look at the linear road, 2 of its queries, and few query plans for them.
- 4^{th}:- Show how linear road data is generated and explain the schema for it.
- 4^{th}:- Present our implementation to mimic the execution of SQL queries.
- 4^{th}:- Present our code for deep reinforcement learning.
- 5^{th}:- Present the assumption we made for our deep reinforcement learning and justification.
- 5^{th}:- Present our results and limitations.
- 6^{th}:- Present the summary of the thesis and provide ideas for further work.

1.6 Conclusion

In this chapter we introduced the reader to:-

- Challenges of inner workings of SQL.
- How Data streams are managed.
- General idea of a Deep reinforcement learning agent, its properties and advantages.
- The outline of this thesis.

In the next chapter we:-

- Present how SQL works in DBMS.
- We introduce the reader to stream processing.
- Give an outline of how Deep learning works.
- Give an outline of how Reinforcement learning works.
- Show how Deep learning and Reinforcement learning combines to give rise to DQNs.

Chapter 2

Related Work

2.1 Introduction, Query optimization

A database can be thought of as a list of tables, where in each table itself can be considered as a list of data points ordered initially in the sequence they are entered.

There are various tools which can be used to connect to a database, here we focus on structured query languages(SQL). A simple SQL query looks like this

```
1    SELECT column_name_1, column_name_2
2    FROM table_name
3    WHERE condition
```

This query is essentially asking to display the 2 columns from the table where the condition given is satisfied. This to particular query might be looking simple, but if the condition introduced is a complex one or if the table from which we need to return the output is complex, the question of how to execute the query optimally becomes difficult to answer.

2.2 Converting SQL queries to parse trees

The exact grammar for the convertion to parse tree is not listed below. But this step isn't simply conversion to a parse tree, the preprocessor which does the conversion also has several more functions.

For example, if the SQL query uses a "view" in the query as a relation, then each instance has to be replaced by the parse tree.

The preprocessor also has to conduct semantic checking, that is, check if relations used exist, check for ambiguity, and type checking. If a parse tree passes the preprocessing then it is said to be **valid**. In these parse trees, there are 2 types of nodes, one the atoms, which are essentially keywords in SQL, operators, constants and attributes. The second is Syntactic categories, these are names for families of subqueries in triangular brackets. Each of the syntactic category has unique expansion into atoms and further syntactic categories.

2.3 Relational algebra

As we saw above, order of operations matters, if the order of operations is not thoughtout and done blindly alot of redundant steps are executed and memory is moved around unnecessarily. There are few ways to atleast look and analyse the operations and how they can be simplied.(Garcia-Molina et al., 2002)

Let R,S be relations. Some simple laws, associativity and commutativity can easily be verified:-

- $R \times S = S \times R$
- $(R \times S) \times T = R \times (S \times T)$
- $R \bowtie S = S \bowtie R$
- $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
- $R \cup S = S \cup R$
- $(R \cup S) \cup T = R \cup (S \cup T)$
- $R \cap S = S \cap R$
- $(R \cap S) \cap T = R \cap (S \cap T)$

When applying associative law on relations, need to be careful whether the conditions actually makes sense after the order is changed.

While the above identities work on both sets and bags(bags allow for repeatition). To show that laws for sets and bags do differ an easy way is to consider the distributive property.

$$A \cap_S (B \cup_S C) = (A \cap_S B) \cup_S (A \cap_S C)$$
$$A \cap_B (B \cup_B C) \neq (A \cap_B B) \cup_B (A \cap_B C)$$

We can simply show it with an example. Let $A = \{t\}, B = \{t\}, C = \{t\}$. The LHS comes to be $\{t\}$, whereas RHS is $\{t, t\}$

2.3.1 Select operator σ

First we start with simple properties of the σ operator. Need to be careful about the attributes used in the select operator condition when pushing it down.

- $\sigma_{C_1 \wedge C_2}(R) = \sigma_{C_1}(\sigma_{C_2}(R))$
- $\sigma_{C_1 \vee C_2}(R) = (\sigma_{C_1}(R)) \cup_S (\sigma_{C_2}(R))$
- $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$
- $\sigma_C(R - S) = \sigma_C(R) - \sigma_C(S) = \sigma_C(R) - S$
- $\sigma_C(R \times S) = \sigma_C(R) \times S$
- $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$
- $\sigma_C(R \underset{D}{\bowtie} S) = \sigma_C(R) \underset{D}{\bowtie} S$
- $\sigma_C(R \cap S) = \sigma_C(R) \cap S$

2.3.2 Projection operator π

While for the Select operator(σ) the identities were quite straight forward with not many things to consider, the identities for Projection operator (π) are bit more involved.

- $\pi_L(R \bowtie S) = \pi_L(\pi_M(R) \bowtie \pi_N(S))$, where M, N are attributes required for the join or they are inputs to the projection.
- $\pi_L(R \underset{D}{\bowtie} S) = \pi_L(\pi_M(R) \underset{D}{\bowtie} \pi_N(S))$, similar to above identity/ law.
- $\pi_L(R \times S) = \pi_L(\pi_M(R) \times \pi_N(S))$
- $\pi_L(R \cup_B S) = \pi_L(R) \cup_B \pi_L(S)$
- $\pi_L(\sigma_C(R)) = \pi_L(\sigma_C(\pi_M(R))$

2.3.3 Duplicate Elimination operator δ

The δ operator eliminates duplicates from bags.

- $\delta(R) = R$, if R does not have any duplicates.
- $\delta(R \times S) = \delta(R) \times \delta(S)$
- $\delta(R \bowtie S) = \delta(R) \bowtie \delta(S)$
- $\delta(R \underset{D}{\bowtie} S) = \delta(R) \underset{D}{\bowtie} \delta(S)$
- $\delta(\sigma_C(R)) = \sigma_C(\delta(R))$
- $\delta(R \cap_B S) = \delta(R) \cap_B S$

2.3.4 Aggregation operator γ

It is difficult to give identities for the aggregation operator, like done for the above operators. This is mostly due to how the details of how the aggregation operator is used.

- $\sigma(\gamma_L(R)) = \gamma_L(R)$
- $\gamma_L(R) = \gamma_L(\pi_M(R))$, where M must at least contain the attributed used in L.

2.4 Converting Parse trees into logical expression

Till now, the only SQL related information presented is how to convert a Query into the parse tree, which is grammar dependent. Given the parse tree, need to substitute nodes by operators seen above, later this expression is optimized to be later converted to a physical query plan.

Now to convert the parse tree into the logical expression. First, look at the transformation of select-from-where statement.

- $< Query > \rightarrow < SFW >$.
- $< SFW > \rightarrow SELECT < SelList > FROM < FromList > WHERE < Condition >$.
- $< SelList > \rightarrow \pi_L$, where L is the list of attributes in $< SelList >$.
- $< Condition > \rightarrow \sigma_C$, where C is the equivalent of $< Condition >$

While there is nothign inherently wrong about the last statement, need to consider the case where $< Condition >$ involves subqueries. A simple explanation about why it isn't allowed is a convention normally the subscript has to be a boolean condition, and if it was allowed otherwise, it would be a very expensive operation, as the subscript

in the select operator has to evaluted at every element of the argument relation. This shows the redundance of it. While if this is allowed, it can be simplified and made efficient, but has to be done on a case by case basis with the use of $\bowtie$, $\times$ functions.

Overall it is a good idea to not use subquerying and rather using joins.

At this point, by making the substitutions mentioned above and using the algebraic identities, we obtain a starting logical query plan. The query has to be transformed into a query which the compiler believes to be the cheapest or the optimal. But, a thing which further complicates the process is the join order.

With the current knowledge, few optimizing rules are evident.

- **Selection repositioning**, Selections should be pushed down as much as possbile, but sometimes, might need to take the selection operator a level up first.
- Pushing projections down the parse tree, being careful with the new projections made in the process.
- Duplicate removal needs to be repositioned.
- σ combined with $\times$ below can result in equijoin, which is much more efficient.

Normally, parsers only have nodes with $0, 1, 2$ children, which corresponds to uniary and binary operators. But many of the operators(natural join, union, and intersection) are commutative and associative, so it helps combine them on a single level to provide the opportunity to prioritize which ones are done first. To do this step, the following guidelines are sufficient.

- We must replace the natural joins with theta-joins that equate the at tributes of the same name.
- We must add a projection to eliminate duplicate copies of attributes involved in a natural join that has become a theta-join

- The theta-join conditions must be associative.

2.5 Explain difficulties/ Time complexity

Currently, we have taken a query as an input and converted it into a logical plan, and then applied more transformation using relational algebra to make the optimal query plan. The next step is to convert it into a physical query plan which can be executed. To complete this step, need to compare the cost of various physical query plan derived from the logical query plan. This plan with the least cost is then passed query execution engine and executed. When trying to select a query plan, need to select:-

- An order for the bracketing for associative and commutative operations like joins, unions, and intersections.
- The underlying algorithm for each operator in the logical plan, for instance, deciding whether a nested-loop join or a hash-join should be used.
- Need to consider additional operators like scanning, sorting, and so on that are needed for the physical plan but that were not present explicitly in the logical plan.
- A method for passing information/ variable values fromt the output of one operator to the next, for instance, by storing the intermediate result on disk or by using iterators and passing an argument one tuple or one main-memory buffer at a time.

Finally after selecting all these methods need to actually check the time required to excute the plan, one obivious way to calculate the time is actually executing the plan to see the cost. But if we try to find the optimal plan by this method then we are

essentially executing every plan. That is expensive and redundant. The next few section presents few methods which can help in this task.

2.5.1 Estimating size and cost

$B(R) \coloneqq$ is the number of blocks needed to hold all the tuples of relation R.
$T(R) \coloneqq$ is the number of tuples of relation R.
$V(R, a) \coloneqq$ is the value count for attribute a of relation R, that is, the number of distinct values relation R has in attribute a.
$V(R, [a_1, a_2, ..., a_n]) \coloneqq$ is the number of distinct values R has when all of attributes $a_1, a_2, ..., a_n$ are considered together, that is, the number of tuples in $\delta(\pi_{a_1,a_2,...,a_n}(R))$

Compared to algorithms learnt in an introductory algorithms class, here the memory usage is huge and need to remember that intermidate/ temporary relations calculated will also incur a cost on the system. So need to take them into account as well because the physical plan is selected to minimize the estimated cost of evaluating the query. Over all the estimation method should pass the following sanity check:-

- Give accurate estimates. No matter what method is used for executing query plans.
- Are easy to compute.
- Are logically consistent; that is, the size estimate for an intermediate relation should not depend on how that relation is computed. For instance, the size estimate for a join of several relations should not depend on the order in which we join the relations.

Even though there is no agreed upon method to do this, it does not matter because the goal is to help select a query plan and not to find the exact minimum cost. So even if the estimated cost is wrong, but if it is wrong similarly then we will still have the least costing plan.

2.5.2 Estimation of Projection

The projection operator is a bag operation, so it does not reduce the number of tuples, only reduces the size of each tuple.

$T(R) = T(\pi(R))$

$B(R) \leq B(\pi(R))$

2.5.3 Estimation of Selection

Let $S = \sigma_{A=c}(R)$, here A is an attribute of R and c is a constant

$T(S) = \frac{T(R)}{V(R,A)}$, is it important to note that this is an estimate and not the actual value, this will be the actual value if all the attributes in A have equal occurance. An even better estimate can be obtained if the DBMS stores a statistic know as histogram.

The above calculate was easy because of the equality. if $S = \sigma_{A<c}(R)$, we simply estimate $T(S) = \frac{T(R)}{3}$

Now if $S = \sigma_{A\neq c}(R)$, can take $T(S) = T(R)$ or $T(S) = T(R) - \frac{T(R)}{V(R,A)}$

If $S = \sigma_{A=c_1 \vee c_2}$, take them to be independent conditions.

2.5.4 Estimation of Join, single attribute

For natural joins, we assume, the natural join of two relations involves only the equality of two attributes. That is, we study the join $R(X,Y) \bowtie S(Y,Z)$, but initially we assume that Y is a single attribute although X and Z can represent any set of attributes. But it is hard to find a good estimate as $T(R \bowtie S) \in [0, T(R)*T(S)]$, to help with this two assumptions are made.

- **Containment of Value Sets** If Y is an attribute appearing in several relations, then each relation chooses its values from the front of a fixed list of values $y_1, y_2, y_3, ...$ and has all the values in that prefix. As a consequence, if R and

S are two relations with an attribute Y, and $V(R,Y) \leq V(S,Y)$, then every Y-value of R will be a Y-value of S.

- **Preservation of Value Sets** If we join a relation R with another relation, then an attribute A that is not a join attribute (i.e., not present in both relations) does not lose values from its set of possible values. More precisely, if A is an attribute of R but not of S, then $V(R \bowtie S, A) = V(R, A)$. Note that the order of joining R and S is not important, so we could just as well have said that $V(S \bowtie R, A) = V(R, A)$.

Using the above assumptions we can claim,

$$T(R \bowtie S) = \frac{T(R) * T(S)}{max(V(R,Y), V(S,Y))}$$

Let $V(R,Y) \leq V(S,Y)$, then every tuple t of R has a chance of $\frac{1}{V(S,Y)}$ of joining with a given tuple of S. Since there are $T(S)$ tuples in S, the expected number of tuples that t joins with is $\frac{T(S)}{V(S,Y)}$. As there are $T(R)$ tuples of R, the estimated size of $R \bowtie S$ is $\frac{T(R)*T(S)}{V(S,Y)}$, if it was $V(R,Y) \geq V(S,Y)$, then $\frac{T(R)*T(S)}{V(R,Y)}$

Guidelines for other type of joins:-

- The number of tuples in the result of an equijoin can be computed exactly as for a natural join, after accounting for the change in variable names.

- Other theta-joins can be estimated as if they were a selection following a product.

2.5.5 Estimation of Join, multiple attribute

Now we assume Y in $R(X,Y) \bowtie S(Y,Z)$ represents multiple attributes. Say for example, $R(X, y_1, y_2) \bowtie S(y_1, y_2, Z)$. We again do a probability calculation.

Let $r \in R, s \in S$ be a tuples, the probability that r, s agree on y_1 is $\frac{1}{max(V(R,y_1),V(S,y_1))}$, similarly for y_2. So we get the expected value to be

$$\frac{T(R) * T(S)}{max(V(R, y_1), V(S, y_1)) * max(V(R, y_2), V(S, y_2))}$$

From this the pattern is clear, need to divide $T(R)*T(S)$ by maximum of $V(R, y), V(S, y)$ for each attribute.

But this is only the calculation for $T(R \bowtie S)$, need to calculate for $B(R \bowtie S)$ as well.

2.5.6 Multiple Joins

For this case we work with $S = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$

Here we have to make use of the containment assumption. Say the attribute A appears in k of R_i's and the values corresponding to $V(R_i, A)$ are $v_1 \leq v_2 \leq ... \leq v_k$, need to find the probability that tuples agree on A.

Consider the tuple t_1 chosen from the relation that has the smallest number of $A-$values, v_1 . By the containment assumption, each of these v_1 values is among the $A-$values found in the other relations that have attribute A. Consider the relation that has V_i values in attribute A. Its selected tuple t_i has probability $\frac{1}{v_1}$ of agreeing with t_1 on A. Since this claim is true for all $i \in \{2, 3, ..., k\}$, the probability that all k tuples agree on A is the product $\frac{1}{v_2 * v_3 * ... * V_k}$. This analysis gives us the rule for estimating the size of any join.

Start with the product of the number of tuples in each relation. Then for each attribute A appearing at least twice, divide by all but the least of the $V(R, A)$'s

2.5.7 Union

If U_B is used, it is the sum of the two individually.

If U_S is used, then number of tuples range from the max of two, to their sum. So take mean of the range.

2.5.8 Intersection

The number of tuples here ranges from 0 to the minimum of the two(in case of set intersection), so again can take the mean of this range.

2.5.9 Difference

The range for $R - S$ is $[T(R) - T(S), T(R)]$, so again mean of the range.

2.5.10 Duplicate Elimination

The range for $\delta(R)$ is $[1, T(R)]$, so again can take the mean, there can be other estimates as well.

A nice compromise is $min(\frac{T(R)}{2}, \prod V(R, a_i))$

2.5.11 Grouping and Aggregation

Same as duplicate.

2.6 Other tools

We assume that the "cost" of evaluating an expression is approximated well by the number of disk I/O's performed. The number of disk I/O's, in turn, is influenced by:

- The particular logical operators chosen to implement the query.
- The sizes of intermediate results, need to pass them to the next function.
- The physical operators used to implement logical operators, e.g., the choice of a one-pass or two-pass join, or the choice to sort or not sort a given relation.
- The ordering of similar operations, especially joins.

- The method of passing arguments from one physical operator to the next.

2.6.1 Histogram

In the earlier section the statistics were heavily used in calculations. Another statistic that can be stored is the histogram.

If $V(R, A)$ is not too large, then the histogram may consist of the number (or fraction) of the tuples having each of the values of attribute A. If there are a great many values of this attribute, then only the most frequent values may be recorded individually, while other values are counted in groups.

Equal width To start, select 2 parameters, w the width and v_0 a begining point of a column in the histogram, which will initally be the considered the lower bound and if an even lower value is noticed, make a smaller column as well and update the lower bound.

Equal height These are the common "percentiles". A percentile $p, 2p$, so on.

Most frequent values List the most common values and their numbers of occurrences. This information may be provided along with a count of occurrences for all the other values as a group, or we may record frequent values in addition to an equal width or equal height histogram for the other values.

2.6.2 Heuristics

One important use of cost estimates for queries or subqueries is in the application of heuristic transformations of the query.

Heuristics applied independent of cost estimates can be expected almost certainly to improve the cost of a logical query plan

However, there are other points in the query optimization process where estimating the cost both before and after a transformation will allow us to apply a transformation where it appears to reduce cost and avoid the transformation otherwise. In particular,

when the preferred logical query plan is being generated, we may consider a number of optional transformations and the costs before and after. Because we are estimating the cost of a logical query plan, so we have not yet made decisions about the physical operators that will be used to implement the operators of relational algebra, our cost estimate cannot be based on disk I / O's. Rather, we estimate the sizes of all intermediate results their sum is our heuristic estimate for the cost of the entire logical plan.

2.7 Enumeration Methods

The naive method of find the least costing plan is enumerating all the possible plans and calculating the cost for them and then selecting the minimum one.

Top-Down Here, we work down the tree of the logical query plan from the root. For each possible implementation of the operation at the root, we consider each possible way to evaluate its argument(s) , and compute the cost of each combination, taking the best.

Bottom-up For each subexpression of the logical-query-plan tree, we compute the costs of all possible ways to compute that subexpression. The possibilities and costs for a subexpression E are computed by considering the options for the subexpressions for E , and combining them in all possible ways with implementations for the root operator of E.

There is actually not much difference between the two approaches in their broadest interpretations, since either way, all possible combinations of ways to implement each operator in the query tree are considered. When limiting the search, a top-down approach may allow us to eliminate certain options that could not be eliminated bottom-up. However, bottom-up strategies that limit choices effectively have also been developed. there is an apparent simplification of the bottom-up method, where

we consider only the best plan for each subexpression when we compute the plans for a larger subexpression. This approach, called **dynamic programming**, is not guaranteed to yield the best plan, although often it does. The approach called Selinger-style (or System-R-style) optimization exploits additional properties that some of the plans for a subexpression may have, in order to produce optimal overall plans from plans that are not optimal for certain subexpressions.

2.7.1 Heuristic Selection

One option is to use the same approach to selecting a physical plan that is generally used for selecting a logical plan: make a sequence of choices based on heuristics. Few common ones are:-

- If the logical plan calls for a selection $\sigma_{A=c}(R)$, and stored relation R has an index on attribute A, then perform an index-scan to obtain only the tuples of R with $A-$value equal to c.
- if the selection involves one condition like A = c above, and other conditions as well, we can implement the selection by an index scan followed by a further selection on the tuples, which we shall represent by the physical operator filter.
- If an argument of a join has an index on the join attribute(s), then use an index-join with that relation in the inner loop.
- If one argument of a join is sorted on the join attribute(s) , then prefer a sort-join to a hash-join, although not necessarily to an index-join if one is possible.
- When computing the union or intersection of three or more relations, group the smallest relations first.

2.7.2 Branch-and-Bound

This approach, often used in practice, begins by using heuristics to find a good physical plan for the entire logical query plan. Let the cost of this plan be C. Then as we consider other plans for subqueries, we can eliminate any plan for a sub query that has a cost greater than C, since that plan for the sub query could not possibly participate in a plan for the complete query that is better than what we already know. Likewise, if we construct a plan for the complete query that has cost less than C, we replace C by the cost of this better plan in subsequent exploration of the space of physical query plans. An important advantage of this approach is that we can choose when to cut off the search and take the best plan found so far. For instance, if the cost C is small, then even if there are much better plans to be found, the time spent finding them may exceed C, so it does not make sense to continue the search. However, if C is large, then investing time in the hope of finding a faster plan is wise.

2.7.3 Hill Climbing

This approach, in which we really search for a "valley" in the space of physical plans and their costs, starts with a heuristically selected physical plan. We can then make small changes to the plan, e.g., replacing one method for an operator by another, or reordering joins by using the associative and/or commutativen laws, to find "nearby" plans that have lower cost. When we find a plan such that no small modification yields a plan of lower cost, we make that plan our chosen physical query plan .

2.7.4 Selinger-Style Optimization

This approach improves upon the dynamic-programming approach by keeping for each subexpression not only the plan of least cost, but certain other plans that have higher cost, yet produce a result that is sorted in an order that may be useful higher up in the expression tree. Examples of such interesting orders are when the result of

the subexpression is sorted on one of:

- The attribute(s) specified in a sort operator (τ) at the root.
- The grouping attribute(s) of a later group-by operator (γ).
- The join attribute(s) of a later join.

If we take the cost of a plan to be the sum of the sizes of the intermediate relations, then there appears to be no advantage to having an argument sorted. However, if we use the more accurate measure, disk I/O's, as the cost, then the advantage of having an argument sorted becomes clear if we can use one of the sort-based algorithms and save the work of the first pass for the argument that is sorted already.

2.8 Join Order

Join takes in two argument, while the end result is independent of the order of the two arguments, the method used to compute the result maybe dependent on the order. Perhaps most important, the one-pass join reads one relation preferably the smaller into main memory, creating a structure such as a hash table to facilitate matching of tuples from the other relation. It then reads the other relation, one block at a time, to join its tuples with the tuples stored in memory.

For instance, suppose that when we select a physical plan we decide to use a one-pass join. Then we shall assume the left argument of the join is the smaller relation and store it in a main-memory data structure. This relation is called the build relation. The right argument of the join, called the probe relation, is read a block at a time and its tuples are matched in main memory with those of the build relation. Other join algorithms that distinguish between their arguments include:

- Nested-loop join, where we assume the left argument is the relation of the outer loop.

- Index-join, where we assume the right argument has the index.

2.8.1 Join Trees

When we have the join of two relations, we need to order the arguments. We shall conventionally select the one whose estimated size is the smaller as the left argument. Notice that the algorithms mentioned above – one-pass, nested loop, and indexed - each work best if the left argument is the smaller. More precisely, one-pass and nested-loop joins each assign a special role to the smaller relation (build relation, or outer loop), and index-joins typically are reasonable choices only if one relation is small and the other has an index. It is quite common for there to be a significant and discernible difference in the sizes of arguments, because a query involving joins very often also involves a selection on at least one attribute, and that selection reduces the estimated size of one of the relations greatly.

When we need to join more than 2 relations, the order in which they are joined can be represented by a binary tree, where each node has either 0 or 2 children. A tree where the right child always a leaf node is called a left deep tree, one can similarly define a right deep tree. Any other tree is will be called **bushy**. We will stick with left deep tree due to their interaction with various common join algorithms. This introduced limitation also helps to reduce search space.

If one-pass joins are used, and the build relation is on the left, then the amount of memory needed at any one time tends to be smaller than if we used a right-deep tree or a bushy tree for the same relations.

If we use nested-loop joins, with the relation of the outer loop on the left, then we avoid constructing any intermediate relation more than once.

2.8.2 DP to decide join order

Suppose we need to calculate $R_1 \bowtie R_2 \bowtie ... \bowtie R_n$. For the DP algorithm construct a table with an entry for each subset of one or more of the n relations. In that table we put

- the estimated size of the join of these relations.
- the least cost of computing the join of these relations. Other, more complex estimates, such as total disk I/O's, could be used if we were willing and able to do the extra calculation involved.
- The expression that yields the least cost. This expression joins the set of relations in question, with some grouping. We can optionally restrict ourselves to left-deep expressions, in which case the expression is just an ordering of the relations.

2.8.3 Greedy algorithm for join order

Even the carefully limited search of dynamic programming leads to a number of calculations that is exponential in the number of relations joined. It is reasonable to use an exhaustive method like dynamic programming or branch-and-bound search to find optimal join orders of five or six relations. However, when the number of joins grows beyond that, or if we choose not to invest the time necessary for an exhaustive search, then we can use a join-order heuristic in our query optimizer.

The most common choice of heuristic is a greedy algorithm, where we make one decision at a time about the order of joins and never backtrack or reconsider decisions once made. We shall consider a greedy algorithm that only selects a left-deep tree. The "greediness" is based on the idea that we want to keep the intermediate relations as small as possible at each level of the tree.

Start with the pair of relations whose estimated join size is smallest. The join of these relations becomes the current tree. Find, among all those relations not yet included in the current tree, the relation that, when joined with the current tree, yields the relation of smallest estimated size. The new current tree has the old current tree as its left argument and the selected relation as its right argument.

2.9 Physical Query Plan

We have parsed the query, converted it to an initial logical query plan, and improved that logical query plan with transformations. Part of the process of selecting the physical query plan is enumeration and cost estimation for all of our options, then focused on the question of enumeration, cost estimation, and ordering for joins of several relations. By extension, we can use similar techniques to order groups of unions, intersections, or any associative/commutative operation

There are still several steps needed to turn the logical plan into a complete physical query plan.

- Selection of algorithms to implement the operations of the query plan, when algorithm-selection was not done as part of some earlier step such as selection of a join order by dynamic programming.

- Decisions regarding when intermediate results will be materialized (cre ated whole and stored on disk) , and when they will be pipelined (created only in main memory, and not necessarily kept in their entirety at any one time) .

- Notation for physical-query-plan operators, which must include details regarding access methods for stored relations and algorithms for imple mentation of relational-algebra operators.

2.9.1 Choosing a Selection Method

One of the important steps in choosing a physical query plan is to pick algorithms for each selection operator.

Assuming there are no multidimensional indexes on several of the attributes, then each physical plan uses some number of attributes that each have an index, and are compared to a constant in one of the terms of the selection. We then use these indexes to identify the sets of tuples that satisfy each of the conditions. For simplicity, we shall not consider the use of several indexes in this way. Rather, we limit our discussion to physical plans that:

- Use one comparison of the form $A\theta c$, where A is an attribute with an index, c is a constant, and θ s a comparison operator such as $=$ or $<$.
- Retrieve all tuples that satisfy the comparison, using the index scan physical operator.
- Consider each tuple selected to decide whether it satisfies the rest of the selection condition. We shall call the physical operator that performs this step Filter; it takes the condition used to select tuples as a parameter, much as the rr operator of relational algebra does.

In addition to physical plans of this form, we must also consider the plan that uses no index but reads the entire relation (using the table-scan physical operator) and passes each tuple to the Filter operator to check for satisfaction of the selection condition.

We decide among the physical plans with which to implement a given selection by estimating the cost of reading data for each possible option. To compare costs of alternative plans we cannot continue using the simplified cost estimate of intermediate-relation size. The reason is that we are now considering implementations of a single

step of the logical query plan, and intermediate relations are independent of implementation. Thus, we shall refocus our attention and resume counting disk I/O's.

2.9.2 Choosing a Join Method

On the assumption that we know (or can estimate) how many buffers are available to perform the join, we can apply the formulas for sort/ indexed/ hash join. However, if we are not sure of, or cannot know, the number of buffers that will be available during the execution of this query (because we do not know what else the DBMS is doing at the same time), or if we do not have estimates of important size parameters such as the V (R, a)'s, then there are still some principles we can apply to choosing a join method. Similar ideas apply to other binary operations such as unions, and to the full-relation, unary operators, γ, δ

2.9.3 Pipelining Versus Materialization

The last major issue we shall discuss in connection with choice of a physical query plan is pipelining of results. The naive way to execute a query plan is to order the operations appropriately (so an operation is not performed until the argument(s) below it have been performed), and store the result of each operation on disk until it is needed by another operation. This strategy is called materialization, since each intermediate relation is materialized on disk. A more subtle, and generally more efficient, way to execute a query plan is to interleave the execution of several operations. The tuples produced by one operation are passed directly to the operation that uses it, without ever storing the intermediate tuples on disk. This approach is called pipelining, and it typically is implemented by a network of iterators whose functions call each other at appropriate times. Since it saves disk I/O 's, there is an obvious advantage to pipelining, but there is a corresponding disadvantage. Since several operations must share main memory at any time, there is a chance that algorithms with higher disk-

I/O requirements must be chosen , or thrashing will occur, thus giving back all the disk-I/O savings that were gained by pipelining, and possibly more.

2.9.4 Pipelining Unary Operations

Unary operations - selection and projection - are excellent candidates for pipelining. Since these operations are tuple-at-a-time, we never need to have more than one block for input, and one block for the output. We may implement a pipelined unary operation by iterators, The consumer of the pipelined result calls **GetNext()** each time another tuple is needed. In the case of a projection, it is only necessary to call **GetNext()** once on the source of tuples, project that tuple appropriately, and return the result to the consumer. For a selection σ_C (technically, the physical operator **Filter(C)**) , it may be necessary to call **GetNext()** several times at the source, until one tuple that satisfies condition c is found.

2.9.5 Pipelining Binary Operations

The results of binary operations can also be pipelined. We use one buffer to pass the result to its consumer, one block at a time. However, the number of other buffers needed to compute the result and to consume the result varies, depending on the size of the result and the sizes of other relations involved in the query. We shall use an extended example to illustrate the tradeoffs and opportunities.

2.9.6 Notation for Physical Query Plans

Operators for Leaves

Each relation R that is a leaf operand of the logical-query-plan tree will be replaced by a scan operator. The options are:

- **TableScan(R):** All blocks holding tuples of R are read in arbitrary order.

- **SortScan(R,L):** Tuples of R are read in order, sorted according to the attribute(s) on list L.

- **IndexScan(R,C):** Here, C i s a condition of the form $A\theta c$, where A is an attribute of R, θ is a comparison such as $=$ or $<$, and c is a constant. Tuples of R are accessed through an index on attribute A. If the comparison θ is not $=$, then the index must be one, such as a B-tree, that supports range queries.

- **IndexScan(R,A):** Here A is an attribute of R. The entire relation R is retrieved via an index on $R.A$. This operator behaves like **TableScan**, but may be more efficient in certain circumstances, if R is not clustered and/or its blocks are not easily found.

Physical Operators for Selection

A logical operator $\sigma_C(R)$ is often combined, or partially combined, with the access method for relation R, when R is a stored relation. Other selections, where the argument is not a stored relation or an appropriate index is not available, will be replaced by the corresponding physical operator we have called **Filter**.

Physical Sort Operators

Sorting of a relation can occur at any point in the physical query plan. We have already introduced the **SortScan(R, L)** operator, which reads a stored relation R and produces it sorted according to the list of attributes L. When we apply a sort-based algorithm for operations such as join or grouping, there is an initial phase in which we sort the argument according to some list of attributes. It is common to use an explicit physical operator $Sort(L)$ to perform this sort on an operand relation that is not stored. This operator can also be used at the top of the physical-query-plan tree if the result needs to be sorted because of an **ORDER BY** clause in the original query,

Other Relational-Algebra Operations

All other operations are replaced by a suitable physical operator. These operators can be given designations that indicate:

- The operation being performed, e.g., join or grouping.
- Necessary parameters, e.g., the condition in a theta-join or the list of elements in a grouping.
- A general strategy for the algorithm: sort-based, hash-based, or in some joins, index-based.
- A decision about the number of passes to be used: one-pass, two-pass, or multipass (recursive, using as many passes as necessary for the data at hand) . Alternatively, this choice may be left until run-time.
- An anticipated number of buffers the operation will require.

2.9.7 Ordering of Physical Operations

Our final topic regarding physical query plans is the matter of order of operations. The physical query plan is generally represented as a tree, and trees imply something about order of operations, since data must flow up the tree. However, since bushy trees may have interior nodes that are neither ancestors nor descendants of one another, the order of evaluation of interior nodes may not always be clear. Moreover, since iterators can be used to implement operations in a pipelined manner, it is possible that the times of execution for various nodes overlap, and the notion of "ordering" nodes makes no sense.

If materialization is implemented in the obvious store-and-later-retrieve way, and pipelining is implemented by iterators, then we may establish a fixed sequence of events whereby each operation of a physical query plan is executed.The following rules summarize the ordering of events i m pl icit in a phy sical query-plan tree:

- Break the tree into subtrees at each edge that represents materialization. The subtrees will be executed one-at-a-time.

- Order the execution of the subtrees in a bottom-up, left-to-right manner. To be precise, perform a preorder traversal of the entire tree. Order the subtrees in the order in which the preorder traversal exits from the subtrees.

- Execute all nodes of each subtree using a network of iterators. Thus, all the nodes in one subtree are executed simultaneously, with GetNext calls among their operators determining the exact order of events.

2.10 User defined function

User defined functions can be used to write SQL queries. Like any function in object oriented programming, UDF in SQL accept parameters of certain types as input and returns an output. These can help write queries quickly. UDFs can be modified independently of any other code.
SQL has the following types of UDFs:-

- **Scalar functions**:- As the name suggests, the output is scalar, in the sense it is $1D$, only a single value is output. These functions always have a return clause in them. These functions traditionally do not provide high performance.

- **Inline table-valued functions**:- These functions are optimized for performance.

- **Multi-statement table-valued functions**:- The syntax for these functions is similar to scalar fucntions, but these do allow for multiple outputs. The functions also do not provide high performance due to inner implementation.

In either of the above mentioned 3 types, no changes to the data base can be made, e.g. no new tables can be created and only a certain amounts of manipulating syntax is allowed.

2.10.1 Scalar

Scalar functions are created using the syntax **CREATE FUNCTION**. They are deleted using the syntax **DELETE FUNCTION**. Functions can be modified using the syntax **ALTER FUNCTION**. Note, a restriction on Scalar functions is that they can not modify databases. An oversimplified way of looking at working of Scalar functions is:-

- Take 0 or more arguments as input.
- Do computations based on the program written.
- Return as output a single scalar value.

The general syntax for a UDF is as follows:-

```
CREATE  FUNCTION [owner_name.]function_nam
    ([{@parameter_name[AS]scalar_parameter_data_type[=default]}[
   ,...n ])
RETURNS scalar_return_data_type
[ WITH < function_option> [ [,] ...n] ]
[ AS ] BEGIN
    function_body
    RETURN scalar_expression
END
```

Listing 2.1: Scalar function syntax

Function body normally contain control-of-flow statements and cursors, these are detrimental to the performance of the query and slow the execution down a lot.

When a Scalar UDF is used, the optimizer does not account for the cost of this function and treats it like a black box whereas it actually has some cost, there is a

memory and time requirement. This not accounting for the costs of scalar functions lead to bad plans.

The Scalar UDF is treated as multiple different queries, so the queries written inside as executed one after another, so each statement has an independent execution plan which does not account for previous or next statement.

Scalar UDF also makes the system unable to run things in parallel.

2.10.2 Inline

(Novick, 2004) Inline UDFs are a form of SQL view that accepts parameters. The param-eters are usually used in theWHEREclause to restrict the rows returned,but they can also be used in other parts of the UDFsSELECTstatement.Inline UDFs can be convenient, but they dont have all the features avail-able to views.

(Novick, 2004) The essential fact of inline UDFs is that they are a type of view, one that allows for parameters that can be used in the single SELECT statement that makes up the body of the inline UDF. Note, a inline UDF is very similar to view.
The general syntax for a UDF is as given in A.10

2.10.3 Multi Statement

Multi statement UDFs are a useful hybrid of a scalar and inline UDF. They return a table that is constructed by the T-SQL script in the body of the function. Multistatement UDFs cant execute stored procedures norcan they create or reference temporary tables or generate any messages. Their communication options are limited by design(Novick, 2004).
The lack of side effects is one of the features of multistatement UDFsthat distinguish them from stored procedures. In many ways, they are similar to a stored procedure that returns a single result set. If you can live with the restrictions, including the restriction on calling stored procedures, a multistatement UDF is a good substitute

for a procedure.The lack of side effects improves the maintainability of the code. The primary reason for using a multistatement UDF instead of aninline UDF is the need for the program logic in the function body(Novick, 2004). The general syntax for Multistatement UDFs is given thereA.11

2.10.4 Effects on Optimization

Overall these functions can be treated as black box operations as they have various restrictions on them which normal SQL operators don't. There is no internal optimization for them, but rather consider them as a separate operator and decide when to execute them.

There is a lot of research currently being done on UDF (Ramachandra et al., 2017), (Ramachandra and Park, 2019), (Gupta et al., 2020), these have improved the runtimes of Scalar UDF by huge amounts.

2.11 Introduction to Data Streams

A **Data Stream** can be thought of as a continuously generated never ending sequence of data. For example, network logistics, healthcare monitoring information, security cameras, all of these continuously produce data, these data sequences/ streams are crucial to their respective users/ organisations, realtime detection of anomally, pattern or changes thereof, or any other type of analytics is in need currently.

But there are many complications while trying to do this, the frequency of data generation isn't stable as it can keep changing throughout the cycle. Most of these service have a Quality of Service (or QoS) requirements defined which takes into account various metrics such as response time, memory usage, and throughput and more. Sometimes these requirements are so heavy and indulgent that it is infeasible to load the incoming data streams into a persistent store and process them effectively using DBMS tools. Hence special tools are required for these, similar to how DBMS

has a variety of steps, Data Stream Management Systems (DSMSs) have multiple steps as well processing of sensor data, pervasive computing, situation monitoring, real-time response, approximate algorithms, on-the-fly mining, complex event processing, and more. While it may seem that optimizing Data stream applications to their specific domains may make them difficult to study, it is possible the generalize the functions/ steps enough to have a basic architecture to study.

Queries used in traditional DBMS are called **ad hoc queries**, where are the queries used in DSMS are called **ContinuousQueries (CQ)**. Ad hoc queries are generally specified, optimized, and evaluated over a snapshot of a database. Whereas, CQs are specified once and evaluated repeatedly against new data over a specified life span or as long as there exists data in the stream. They are long-running queries that produce output continuously. The result is also assumed to be a stream possibly with differing rates and schema (as compared to the input). The difference between ad hoc queries and CQs can be best understood based on their relationship to the data over which they are processed. Different or changing ad hoc queries are processed over (relatively) static data in contrast to the same (or static) CQs that are processed repeatedly over frequently changing (or dynamic) data. It is clear that DBMS can't really cope with data streams in terms on high frequency updates, fulfilling QoS and continuous output. Hence we make use of Data Stream Management Systems (DSMSs).

2.12 Windowing and QoS

Most of the languages used for this are based on SQL, some of them are, Continuous Query Language (CQL), StreamSQL and ESP. Typically, continuous queries consist of relational operators 3 such as select, project, join, and other aggregation operators. A logical query plan, analogous to a query tree used in a traditional DBMS, can be

generated from the specification of a CQ. It can then be transformed into a query plan consisting of detailed algorithms used for computing each operator. Compared to DBMS where either pipelines or materialization is used, DSMS use a push or dataflow methodology. All operators in a DSMS compute on data items as they arrive and cannot assume the stream to be finite. This has significant implications for the computation of a number of operators such as join, sort, and some aggregation operators. These operations cannot be completed without processing the entire input data set (or sets) which poses problems due to the unbounded nature of streams. As a result, these operators will block and produce no output until the stream ends. Hence, they are termed blocking operators. For an operator to output results continuously (and hopefully smoothly) and not wait until the end of the stream, it is imperative that these blocking operators be converted into non-blocking ones. The notion of a window has been introduced to overcome the blocking aspect of a number of operators. Informally, a window defines a finite portion of the stream (as a relation) for processing purposes. A window specification, added to a continuous query specification, can produce time-varying, finite relations out of a stream.

QoS management is important and critical to the success of a DSMS. Few of the popular QoS metrices are Tuple latency, Memory usage, Throughput, Smooth or bursty nature of output streams, Accuracy of results in terms of error tolerance. Most of the metrices can be specified in the query written. Overall to correlate CQs and QoS a simple question can be asked

Given a set of CQs with their QoS specifications, what resources are needed to compute the given CQs and satisfy their QoS requirements?

2.13 Challenges of query optimization on data streams

Picking up on the question form the last section, it is important to be able to address the inverse as well, i.e. given resources, CQs and QoS, need to verify that QoS are satisfied. Like any verification, there are two general ways of tackling this problem.

- Try to model the load generated by the input stream and get a conservative estimate on the metrices used to measure the QoS.
- Continuously monitor the metrices while the program is running, on violation identify the bottleneck and improve upon it.

As seen in previous chapters, creating a model to predict the load can be very difficult and sometimes more expensive than the best result. So we focus on the second approach.

2.13.1 Resource Scheduling Strategies

Similar to the case of traditional DBMS, querying in DSMS involves multiple steps as well. The resource requirements for each step varies hence the need for a scheduling method which can determine an optimal schedule. To further complicate this, the various types of CQs have different QoS, hence the wiggle room for the performance is high, and as DSMS process queries in real time more considerations have to taken into account compared to traditional data bases, e.g. different operators require different amount of memory and time to process data.

2.13.2 Load Shedding and Run-Time Optimization

It is important to note that optimizing resource allocation using any state of the art strategy does not guarantee that resources will be sufficient, as there might be some points in the data stream which overloads particular query, hence wiggle room in the

performance/ accuracy, i.e. some of the data can be discarded to decrease the quality of the result and give an approximate result. This process of decreasing quality of result by not considering some tuple is called load shedding. Given that discarding tuples of the data stream is allowed, how do we decide which ones to discard?

2.13.3 Complex Event and Rule Processing

In the first section we said we proceed with the second strategy, i.e. monitor the metrics. Can extend that idea further to monitor the output of CQs to detect anomalies. Initially, Comple Event Processing was not a part of DSMS, but now many models have been developed to integrate the two. With the integration comes many additional functionalities and helps develop a theoretical base, but also puts additional burden on the system. This along with run time optimiations makes the implementation of DSMS quite complicated.

2.14 Deep Learning

We expand on linear regression model. Given $\{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$ as data points, the goal is to find a function $h_\theta(x)$ which is "as close as possible" to y_i.

One way to do that is, define a loss function J which measures the distance between prediction and actual value of $y^{(i)}$ as

$$J^{(i)}(\theta) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} J^{(i)}(\theta)$$

The goal then reduces to finding parametrize h_θ and finding the θ so that $J(\theta)$ is minimized.

Similarly in neural networks, we define a parameterization of h and then try to minimize the loss function.

Consider the simplest neural network, which consists of an input layer and a output layer with a single neuron. Say, the input is x, then the output neuron has input $h_\theta(x)$, and the output from the neuron is $A(h_\theta(x))$, where A is an activation function.

In the above example, if $x \in \mathbb{R}^n$ then, $h : \mathbb{R}^n \to \mathbb{R}$, which required us to find a single θ. Now say, $h : \mathbb{R}^n \to \mathbb{R}^m$, then we need to find $\theta_1, \theta_2, ..., \theta_m$. Which results in a matrix of dimension $n * m$. For each of the m neurons in the output layer, the input will then be $h_{\theta_i}(x)$ and the output of the m neurons will be a vector $A(h_{\theta_1}(x)), A(h_{\theta_2}(x)), ..., A(h_{\theta_m}(x)) \in \mathbb{R}^m$. This is an example of **Fully connected** $2-$**layer neural network**.

Multi-layer fully connected neural network, for this keep on adding one layer at a time to the neural network, and use the output from the last layer as the input.

At the end the problem is still reduced to finding parameters which minimize the loss and hence can solved using gradient descent and similar methods, but it is much more computationa intense.

2.15 Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward. Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning.

Reinforcement learning differs from supervised learning in not needing labelled input/output pairs be presented, and in not needing sub-optimal actions to be explicitly corrected. Instead the focus is on finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

Reinforcement learning has been successful in applications as diverse as autonomous helicopter flight, robot legged locomotion, cell-phone network routing, marketing strategy selection, factory control, and efficient web-pageindexing. Our study of reinforcement learning will begin with a definition of the Markov decision processes (MDP), which provides the formalism in which RL problems are usually posed.(Ng, 2020)

2.15.1 Markov decision processes

Markov decision process(MDP) is used to formalize various types of stochastic processes. In MDPs, the goal of the agent is to make a sequence of actions to optimize/ maximize an objective function (Xu et al., 2018).
Formally a MDP is a 4−tuple

$$\langle S, A, P(s,a), R(s,a)\rangle$$

$$S \rightarrow \text{Set of all possible states the agent can be in.}$$

$$A \rightarrow \text{Set of all possible actions the agent can take.}$$

$$P(s,a) \rightarrow \text{A probability distribution of going to various states given current state and action. } s^1 \sim P(s,a)$$

$$R(s,a) \rightarrow \text{Reward for taking action } a \text{ on state } s.$$

To the above 4 tuple, additional parameters such as starting state and discount factor can be added.

A run of MDP looks like following:-

Start from the state s_0, choose an action $a_0 \in A$ to take, the action will result in transition in state from $s_0 \rightarrow s_1$, where s_1 is taken from the probability distribution $P(s_0, a_0)$. This change in state results in a reward $R(s_0, a_0)$. Then again an action a_1 is taken, which then results in a transition from state $s_1 \rightarrow s_2$, where s_2 is taken from the probability distribution $P(s_1, a_1)$. This change in state results in a reward

$\gamma R(s_1, a_1)$. and so on.

This will result in a total reward of

$$\sum_{i=0}^{n} \gamma^i R(s_i, a_i)$$

The goal with reinforcement learning is to maximize $\mathbb{E}[\sum_{i=0}^{n} \gamma^i R(s_i, a_i)]$.

Policy is a function $\pi : S \rightarrow A$. **Executing a policy** means at state s, action $a = \pi(s)$ is executed. This leads to the **value function** for policy π.

$$V^{\pi}(s) = \mathbb{E}[\sum_{i=0}^{n} \gamma^i R(s_i, a_i) | s_0 = s, \pi]$$

, This value function satisfies the Bellman Equation.

$$V^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} P(s, \pi(s))(s^{'}) V^{\pi}(s^{'})$$

Now we need to find the optimal policy, which can be formally written as

$$V^{*}(S) = \max_{\pi} V^{\pi}(S)$$

2.15.2 Finding Optima

```
for s in S:
    V(s)=0
while(not converged):
    for s in S:
        V(s)=R(s)+max over all action[gamma*(sum(P(s,a,s')V(s')))]
```

Listing 2.2: value iteration algorithm

The above algorithm is called the value iteration algorithm(Ng, 2020).

```
initialize random pi
while(not converged):
    V=V(pi)
    for s in S:
        pi(s)=max over all actions[sum(P(s,a,s')V(S'))]
```

Listing 2.3: Policy iteration algorithm

The second algorithm is called Policy iteration algorithm(Ng, 2020).

2.16 Deep Reinforcement Learning

Combining the two ideas as shown(Fan et al., 2020)

Algorithm 1 Deep Q-Network (DQN)

Input: MDP (S, A, P, R, γ), replay memory M, number of iterations T, minibatch size n, exploration probability $\epsilon \in (0, 1)$, a family of deep $Q-$networks $Q_\theta : S \times A \rightarrow R$, an integer T_{target} for updating the target network, and a sequence of stepsizes $\{\alpha_t\}_{t \geq 0}$
Initialize the replay memory M to be empty.
Initialize the $Q-$network with random weights θ.
Initialize the weights of the target network with $\theta^* = \theta$
Initialize the initial stateS_0.
for $t = 0, 1, ..., T$ **do**

With probability ϵ, choose A_t uniformly at random from A, and with probability $1 - \epsilon$, choose A_t such that $Q_\theta(S_t, A_t) = \max_{a \in A} Q_\theta(S_t, a)$.

Execute A_t and observe reward R_t and the next state S_{t+1}.

Store transition (S_t, A_t, R_t, S_{t+1}) in M.

Experience replay: Sample random minibatch of transitions $\{(s_i, a_i, r_i, s_i^{'})\}_{i \in [n]}$

For each $i \in [n]$, compute the target $Y_i = r_i + \gamma * \max_{a \in A} Q_{\theta^*}(s_i^{'}, a)$
Update the Q-network: Perform a gradient descent step

$$\theta \leftarrow \theta - \alpha_t * \frac{1}{n} \sum_{i \in [n]} [Y_i - Q_\theta(s_i, a_i)] * \nabla_\theta Q_\theta(s_i, a_i)$$

Update the target network: Update $\theta^* \leftarrow \theta$ every T_{target} steps.
Define policy $\bar{\pi}$ as the greedy policy with respect to Q_θ
Output Action-value function Q_θ and policy $\bar{\pi}$

2.17 Conclusion

To summarize this chapter, we illustrated the process of converting queries into a physical query plan for DBMS. This included various steps such as,

- Convert the given SQL query into a parse tree using the language specific grammar.
- Next have to check if the given parse tree is actually a member of the grammar, that is, sematic checking.
- Substituting the nodes of the parse tree with the proper operators for conversion to a logical plan.
- Next is optimizing the logical query plan by making the algebraic transformation from relation algebra.
- To prepare for cost based search, need to have statistics ready, so have to calculate them, E.g. histogram, the tuple size and more as mentioned earlier.
- Deciding a strategy for the joining and enumeration strategy.
- Lastly for execution, decide between pipeline and materialization.

Then we looked at DSMS and saw the challenged they face and how they overcome them. This essentially is the how the metrices used to measure QoS are inter-related and the windowing method.

In the next chapter we

- Formalize the problem statement.
- Formalize the approach we are planning on using.
- Give details of the two examples we are going to be working with.

Chapter 3

Stream Optimization

3.1 Introduction

A stream is an ordered sequence of data items, which are values that can range from simple numbers to flat tuples to more elaborate structured data that may be deeply nested and have variable size. Streams are conceptually infinite, in the sense that as the streaming computation unfolds over time, the sequence of data items is unbounded in length(Martin Hirzel, 2018).

Stream query optimization is the process of modifying a stream processing query, often by changing its graph topology and/or operators, with the aim of achieving better performance (such as higher throughput, lower latency, or reduced resource usage), while preserving the semantics of the original query.

An optimization should be both safe and profitable. An optimization is safe if it can be applied to a stream query without changing what it computes, as determined by the users requirements. An optimization is profitable if it makes the stream query faster, as measured by metrics that matter to the user, such as throughput, latency, or resource efficiency. There is a substantial literature on different stream query optimizations, with different safety and profitability characteristics. This entry lists the most common optimizations along with short descriptions.

Possible areas of optimization are, batch size, operation combining/ dividing, memory assignment, message passing, operation reordering, garbage collection. We focus mostly on operation reordering.

Last chapter showcased how Deep reinforcement learning can help solve problems by telling the best course of action. We are going to use this technique to improve our operation reordering.

3.2 Formalization Requirements

To even being to tackle the problem first we need to define the problem formally to be able to analyze it.
For the experiment to be reconstructible to the closest possible degree and to define a concrete problem statement for reinforcement learning, we will need to decide upon the following at least:-

- Data Soucre generator
- Query to execute
- The underlying algorithms have to be fixed
- Featurization method of DRL
- How will DRL change over time
- Output of DRL
- Evaluation

Before going on the give answers to the above, we also need to justify using this(DRL) technique.

3.3 Problem Statement

As discussed in the previous chapter, given a query there are multiple ways of executing the query to return the result, these ways correspond to a query plan and for

each query plan there is a cost associated to it, one component of the costs is the time required for execution.

We can represent the plans by P_i and the time corresponding to the plan as C_i. So we can represent them as $\{(P_i, C_i)\}_{i=1}^{n}$. Our goal is to find query plan P_i with the minimum C_i.

3.4 Approach

Currently we make the following assumptions:-

- We have our data source generator with say a fixed seed, which will ensure we have a constant source of data.
- The query to execute is given, that is the window size for data streams is given.
- The query has combination of data streams and static relations as well as operators such as selection, projections, groupby and more.

As mentioned eariler, we are going to focus on finding an order of operations to execute to reduce/ minimize the query execution time.

Say there is a data streams S with m attributes and relations $R_1, R_2, ..., R_n$ each with m_i attributes.

That is to say there are total of $(m + \sum_{i=1}^{n} m_i) = g$ attributes. We can use $1 - 0$ encoding so that any attribute or a combination in this schema can be represented by an array/vector v of length g.

$$v_i = \begin{cases} 1 & if, i \in combination \\ 0 & otherwise \end{cases}$$

This vector will help represent attributes listed in the query. We can further replace 1 with the entropy of the attribute.

For our Deep reinforcement learning data we will need the rewards, actions, current and next possible state, we will need to encode these in a 4−tuple, (F_c, F_n, Action, Reward). When we have enough of these examples we are use them to train our model and try to get a better ordering.

3.5 Example

We look at linear road benchmark to test our hypothesis.

3.5.1 Input

CarLocStr: Stream of car location reports. This forms primary input to the system.

```
CarLocStr(car_id,         /* unique car identifier       */
          speed,          /* speed of the car            */
          exp_way,        /* expressway: 0..10           */
          lane,           /* lane: 0,1,2,3               */
          dir,            /* direction: 0(east), 1(west) */
          x-pos);         /* coordinate in express way   */
```

AccBalQueryStr: Stream of account-balance adhoc queries. Each query requests the current account balance of a car.

```
AccBalQueryStr(car_id,
               query_id);/* id used to associate
                          * responses with queries      */
```

ExpQueryStr: Stream of adhoc queries requesting the expenditure of a car for the current day.

```
ExpQueryStr(car_id,
            query_id);
```

TravelTimeQueryStr: Stream of expected-travel-time adhoc queries.

```
TravelTimeQueryStr(query_id,
                   exp_way,
                   init_seg,      /* initial segment */
                   fin_seg,       /* final segment   */
                   time_of_day,
                   day_of_week);
```

CreditStr: The stream of credits to the account corresponding to a car.

```
CreditStream(car_id, credit);
```

Instead of assuming multiple streams for input we will combine them and assuming a single data stream as input.

3.5.2 Query

In this section we showcase the query statement for a simple and a complex query, convert it into a SQL query and show the complexity within. We use the query statement from linear road itself.

- **Query 1, Segment Average Speed** : The average speed of a particular car in a segment over the last 5 minutes.
- **Query 2, Toll Computation for Segments** : The toll for each segment depends on the average speed and volume of the cars in the segment, and on the presence of accidents in downstream segments.

Now to convert the query statement to actual syntax/ query and showcase few different query plans and how the cost might change.

3.5.3 Segment Average Speed

First we convert the simpler query.

```
SELECT exp_way, dir, seg, AVG(speed) as speed,
FROM CarSegStr [RANGE 5 MINUTES]
WHERE car_id = target
GROUP BY exp_way, dir, seg;
```

Listing 3.1: Custom Simple query from Linear Road

To convert this into a query plan, as seen in the previous chapter, we use relational algebra to convert into a query plan.

$$\gamma_{\text{exp\_way,dir,seg, AVG(speed)}}(\sigma_{\text{car\_id = target}}(\text{CarSegStr}))$$

Of course, the above mentioned method/ bracketing is one way of executing the query. A small modification from the rules in previous chapter can lead to the following improvement!

$$\gamma_{\text{exp\_way,dir,seg, AVG(speed)}}(\pi_{\text{exp\_way,dir,seg, speed}}(\sigma_{\text{car\_id = target}}(\text{CarSegStr})))$$

Now here we are assuming that $CarSegStr$ is a single stream, ready to be used, incase it was a more complicated, that is, it was obtained by joining 2 different streams/ relations with a condition C a further improvement is possible!

$$\gamma_{\text{exp\_way,dir,seg, AVG(speed)}}(\pi_{\text{exp\_way,dir,seg, speed}}(\sigma_{\text{car\_id = target}}(S_1 \underset{C \wedge (\text{car\_id = target})}{\bowtie} S_2)))$$

Of course further improvements can be made, but that will require us to look into attributes of the two streams S_1 and S_2

So to conclude, we made an improvement from the naive plan to a much more efficient plan

- $\gamma_{\text{exp\_way,dir,seg, AVG(speed)}}(\sigma_{\text{car\_id = target}}(\text{CarSegStr}))$
- $\gamma_{\text{exp\_way,dir,seg, AVG(speed)}}(\pi_{\text{exp\_way,dir,seg, speed}}(\sigma_{\text{car\_id = target}}(\text{CarSegStr})))$
- $\gamma_{\text{exp\_way,dir,seg, AVG(speed)}}(\pi_{\text{exp\_way,dir,seg, speed}}(\sigma_{\text{car\_id = target}}(S_1 \underset{C \wedge (\text{car\_id = target})}{\bowtie} S_2)))$

3.5.4 Toll Computation for Segments

Now to convert tackle the more complex query, **Query 2,**

Before going to that, note the steps to perform a query on streams requires to first generate a relation from stream, make relations from this derived relation and then output a stream from the derived relation. This is a 3 step process.

Now we look at the more complex query.

SegAvgSpeed: The average speed of the cars in a segment over the last 5 minutes.

```
SELECT exp_way, dir, seg, AVG(speed) as speed,
FROM CarSegStr [RANGE 5 MINUTES]
GROUP BY exp_way, dir, seg;
```

Listing 3.2: SEGAVGSPEED linear road query

SegVol: Relation containing the number of cars currently in a segment. The relation CurCarSeg is used to determine the cars in each segment.

```
SELECT exp_way, dir, seg, COUNT(*) as volume
FROM CurCarSeg
GROUP BY exp_way, dir, seg;
```

Listing 3.3: SEGVOL linear road query

SegToll:The toll for each segment. There are no entries in the relation for segments having no toll. A segment is tolled only if the average speed of the segment is less than 40, and if it is not affected by an accident. If a segment is tolled, its toll is basetoll $*(\#\text{cars} - 150) * (\#\text{cars} - 150)$. We modify this query to remove the not affected by accident condition.

```
SELECT S.exp_way, S.dir, S.seg, basetoll*(V.volume-150)*(V.volume
   -150)
FROM SegAvgSpeed as S, SegVol as V
WHERE S.exp_way = V.exp_way and S.dir = V.dir and S.seg = V.seg
      and S.speed <= 40;
```

Listing 3.4: SEGTOLL linear road query

These 3 steps combined give us the desired query. Where S in turn calculated via this process

$$\pi_{\text{exp\_way, dir, seg, speed}}(\gamma_{\text{exp\_way,dir,seg}}(\text{CarSegStr}))$$

Where S in turn calculated via this process

$$\pi_{\text{exp\_way, dir, seg, columm}}(\gamma_{\text{exp\_way,dir,seg}}(\text{CurCarSeg}))$$

Finally combining the two with the 3^{rd} query

$$\pi_{\text{S.exp\_way, S.dir, S.seg, S.toll}}$$

$$\sigma_{(\text{S.exp\_way} = \text{V.exp\_way})\wedge(\text{S.dir} = \text{V.dir})\wedge(\text{S.seg} = \text{V.seg})\wedge(\text{S.speed}\leq 40)}(\text{S,V})$$

We can push down the projection and then break down the selection condition, for ease of notation, substitute the 4 conditions as C_1, C_2, C_3, C_4 respectively.

$$\pi_{\text{S.exp\_way, S.dir, S.seg, S.toll}}(\sigma_{C_1}(\sigma_{C_2}(\sigma_{C_3}(\sigma_{C_4}(\text{S,V})))))$$

Now we can list few different query plans

$$\pi_{\text{S.exp\_way, S.dir, S.seg, S.toll}}(\sigma_{C_1}(\sigma_{C_2}(\sigma_{C_3}(\sigma_{C_4}(\text{S,V})))))$$

$$\pi_{\text{S.exp\_way, S.dir, S.seg, S.toll}}(\sigma_{C_1}(\sigma_{C_2}(\sigma_{C_4}(\sigma_{C_3}(\text{S,V})))))$$

$$\pi_{\text{S.exp\_way, S.dir, S.seg, S.toll}}(\sigma_{C_1}(\sigma_{C_3}(\sigma_{C_2}(\sigma_{C_4}(\text{S,V})))))$$

The goal is to try to figure out the best possible method of rearranging these associative operations, in this thesis we try out using deep reinforcement learning to try to predict the best possible move.

3.6 Conclusion

In this chapter we saw:-

- We introduced the need to optimizing queries for data streams.
- Formazlied the problem statement.
- Talked about the method we are planning on using.
- Looked at the Linear road data.

- Looked at the 2 queries, a simple and a complex one and saw multiple query plans for those.

In the next chapter we showcase our implementation of various stages of the pipeline.

- How the data is generated.
- How we process the data to mimic execution of query plans.
- How we generate data for DQNs.
- And lastly we look at how we train a DQN.

Chapter 4

Implementation

This chapter explores the setup required to conduct the experiments. The chapter is divided into

- Data Generation:- We generate the data stream for the Linear road benchmark.
- Query Execution:- We have implemented a query execution in C++ and extracted information while executing the query.
- Deep Reinforcement Learning:- Written a Deep reinforcement learning agent to help optimize the order of operations.

4.1 Data Generation

The data is generated using the walmart linear road code generator without mistim(Cory Stegemoller, 2016).
Linear Road tests Stream Data Management Systems (SDMS) by measuring how many expressways a SDMS can support by giving accurate and timely results to four types of queries that fit two categories: continuous and historical.(Cherniack et al., 2004)

```
java com.walmart.linearroad.generator.LinearGen [-o <output file>]
   [-x <number of xways>] [-m <dummy value to activate multi-
   threading>]
```

Listing 4.1: generate linear road data

Example output of this code

```
0,0,13,10,8,0,0,89,469920,-1,-1,-1,-1,-1,-1
0,0,17,10,8,0,1,65,348479,-1,-1,-1,-1,-1,-1
0,0,22,10,8,0,0,12,63360,-1,-1,-1,-1,-1,-1
0,0,33,10,8,0,1,94,501599,-1,-1,-1,-1,-1,-1
0,0,42,10,8,0,0,14,73920,-1,-1,-1,-1,-1,-1
0,0,4,10,7,0,0,61,322080,-1,-1,-1,-1,-1,-1
0,0,85,10,8,0,1,30,163679,-1,-1,-1,-1,-1,-1
0,0,11,10,6,0,1,41,221759,-1,-1,-1,-1,-1,-1
0,0,23,10,7,0,1,81,432959,-1,-1,-1,-1,-1,-1
0,0,15,10,6,0,0,5,26400,-1,-1,-1,-1,-1,-1
```

Listing 4.2: linear road data example output

Each line in the example indicates 1 data entry, the schema used in the generation of the above code as given.

4.1.1 Schema

The above generated data can be interpretated as follows:-

Column1	Tells the type of query. 0: position report 2: account balance request 3: daily expenditure request 4: travel time request
Column2	Timestamp position.
Column3	Vehicle identification number
Column4	Speed of the vehicle
Column5	Express way number
Column6	Lane ID $(0, ..., 4)$
Column7	Direction of movement(0 =East or 1 =West)
Column8	Segment ID $(0, ..., 99)$
Column9	Position of the vehicle.
Column10	Query identifier
Column11	Start Segment
Column12	End Segment
Column13	Day of the week
Column14	Minute of the day
Column15	Day in the past 10 weeks

4.2 Query Execution

After the data is prepared, we need to execute the queries listed and explained in the previous chapter. We simplified and presented multiple plans of execution and we will explore them.

As suggested, a SDMS can take in multiple data stream as input but we were going to treat it as a single input. Below is the implementation of the 2 queries in $C++$.

4.2.1 Simpler Query

The simpler query

```
1    SELECT exp_way, dir, seg, AVG(speed) as speed,
2    FROM CarSegStr [RANGE 5 MINUTES]
3    WHERE car_id = target
4    GROUP BY exp_way, dir, seg;
```

Listing 4.3: simple query

The goal of implementing this query is to get a sense of time required for executing queries, the time for taking input and time for providing an output.

The code can be broken into 3 parts:-

- Intialization, code overview
- Reading input
- executing the query

A.1

To start, the input file is specified, the window size is specified and the start time of the clock is initialized. The input file is treated as a stream. While many things are to be considered while taking input, as the focus of this thesis was in other area, we simply read from the file part by part.

We continue to read the file until we reach the end of it, everytime we have read the

window size amount of data points, we execute the query.

A window size was fixed which determined the number of entries to read at once, the window size was interpretted in two ways,

- The number of entries to read
- The time interval for which to read the entries

A.2

This information along with the value was passed to the read function A.2 everytime.

This input function treats the window size as specifying the number of entries to read.

The input was taken by simply reading a file and storing the attribute values important to the query. We only store the information necessary for our queries. This was done due to limited RAM and for expediting the process.

A.3

This part prepares the data for aggreation functions. The data is first inserted into a vector. It then essentially groups the rows together depending on the order provided by the means of the sort function(my_sort in this case). The sort function helps group up entries with same express way ID, segment ID and which are going in the same direction.

A.4

Finally do the aggregation, simply check for the continuity of the group and update the measures of the group as you iterate. When you arrive the end of the group simply insert the answer for the aggregate function along with the rest of required attributes into an output and return the table.

4.2.2 Complex Query

The complex query(Widom et al., 2004)

```
SELECT car_id, speed, exp_way, lane, dir, (x-pos/52800) as seg
FROM CarLocStr;
```

Listing 4.4: CarSegStr

```
SELECT car_id, exp_way, dir, seg
FROM CarSegStr [PARTITION BY car_id ROWS 1], CurActiveCars
WHERE CarSegStr.car_id = CurActiveCars.car_id;
```

Listing 4.5: CurCarSeg

```
SELECT exp_way, dir, seg, AVG(speed) as speed,
FROM CarSegStr [RANGE 5 MINUTES]
GROUP BY exp_way, dir, seg;
```

Listing 4.6: SegAvgSpeed

```
SELECT exp_way, dir, seg, COUNT(*) as volume
FROM CurCarSeg
GROUP BY exp_way, dir, seg;
```

Listing 4.7: SegVol

```
SELECT S.exp_way, S.dir, S.seg, basetoll*(V.volume-150)*(V.volume
   -150)
FROM SegAvgSpeed as S, SegVol as V
WHERE S.exp_way = V.exp_way and S.dir = V.dir and S.seg = V.seg
      and S.speed <= 40;
```

Listing 4.8: SegToll

The goal of implementing this query is to obtain data to train a deep reinforcement learning model and to be able to test the model. We start by giving the general outline of the code.

A.5

Similar to the simple query, we fix a window size, the data file and we start the loop

In the loop, each time we read "window size" number of data entries, calculate 3 vectors, **curcarseg**, **segavgspeed**, **segvol**. After calculating these for the data in the data window, we execute the final query of **segtoll**

For $CurCarSeg$, simply filter the cars for the last 5 mins.

A.6

For $SegAvgSpeed$, we first filter the cars which were active in the last 5 mins. Then sort to group data entries. Then lastly apply aggregate function.

A.7

Next, calculate $SegVol$, first sort the vector by the attributes in the group by statement. Then apply the aggregation operation. After obtaining both $SegVol$ and $SegAvgSpeed$, finally proceed to calculate $SegToll$ and obtain data for reinforcement learning.

A.8

Given $SegVol$ and $SegAvgSpeed$, need to do the following steps:-

- Calculate column wise entropy and store it.
- Store size of $SegVol$ and $SegAvgSpeed$.
- For all $4! = 24$ orderings of operations(due to them being associative) execute the $SegToll$ query.
- Store the time require for execution and the number of operations required for each order.

Here is a part of output generated by the code

The first line represents the entropy of the 8 columns,

The next line represents the size of $SegAvgSpeed$ and $SegVol$ respectively.

Then for the next 48 lines, it alternates between the order of operations executed and then the time required in microseconds and the number of operations required.

```
3.32187 0.99998 6.63484 3.29218 2.92342 3.32185 0.99998 6.63482
1916 1914
```

```
0 1 2 3
171487 3672963
0 1 3 2
175067 3672963
0 2 1 3
176562 3672963
0 2 3 1
175096 3672963
```

Listing 4.9: Data for DQN

The example above can be interpretted as

The entropy of columns are 3.32187, 0.99998, 6.63484, 3.29218, 2.92342, 3.32185, 0.99998, 6.63482

The size of $SegAvgSpeed$ is 1916, the size of $SegVol$ is 1914.

The order line can be interpretted as

- The first operation executed is the 0^{th}(S.exp_way = V.exp_way).
- The second operation executed is 1^{st}(S.dir = V.dir).
- The third operation executed is 2^{nd}(S.seg = V.seg).
- The last operation executed is 3^{rd}(S.speed <= 40)

When this order is executed, the time taken is 171487 mircoseconds and the number of operations required is 3672963

Next 2 lines can be interpretted as

- The first operation executed is the 0^{th}(S.exp_way = V.exp_way).
- The second operation executed is 1^{st}(S.dir = V.dir).
- The third operation executed is 3^{rd}(S.speed <= 40).
- The last operation executed is 2^{nd}(S.seg = V.seg)

When this order is executed, the time taken is 175067 mircoseconds and the number of operations required is 3672963.

The fact that the number of operations remains same but the time required changes is due to the internal scheduling by the kernel.

4.3 Deep Reinforcement Learning

The input for MDP is of the format (S, A, S^*, R). But as this game is 1 move game, all tuples will have $S^* = s_f$. When trying to understand a Deep reinforcement learning model, the things one needs to understand about the model are:-

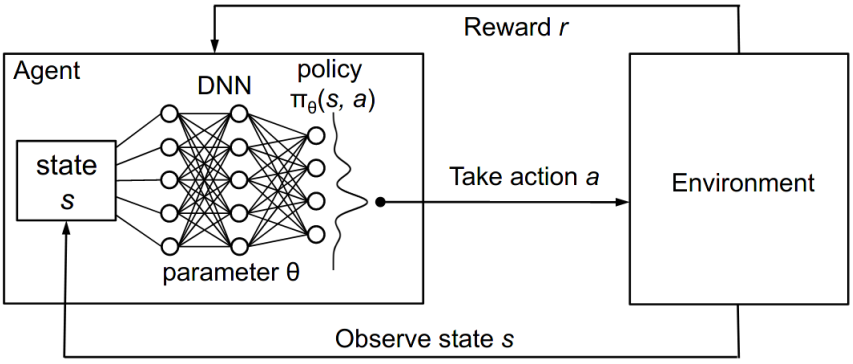
- Actions allowed
- State space
- Transition function
- The reward system
- The assumptions made along the way

The following diagram helps visualize a DQN framework 4·1. The code for the DQN is given here A.9

4.3.1 Actions Allowed

The actions in this scenario, that is the DQN model, is the ordering of the operations we choose to execute. That is, we have $4 = 24$ possible actions on any given input. These are constructed using all possible permutations of $\{0, 1, 2, 3\}$. We then transform this information into 1 hot encoding for ease of use.

Figure 4·1: Deep Reinforcement learning framework



Consider the set of permutations of $\{0, 1, 2, 3\}$, we can order them lexicgraphically. Given a permutation $\mathbb{P}$, it will have a rank r in the ordering.

To convert the permutation into a 1hot encoding, start with a 24 element vector v with all 0s.Then update

$$v[r] = 1$$

4.3.2 State Space

We want the information extracted, i.e. the column wise entropy and the size of the relations, to represent the current state of the environment. Hence, any state is going to be represented as a vector of length 10, the first 8 are the column wise entropy and the $9^{th}, 10^{th}$ are the sizes of the two relations.

Note the makes the image of the state space to be a subset of $\mathbb{R}^{10}$. But as entropy is never going to be 0 and so won't the size of the relations, we can trim it down to $(0, \infty)^{10}$ which is a continuous space.

Discretization is a process that divides numeric features into categorical ones using intervals. Depending on the application and predictive model being used, discretization can bring several benefits, including faster computation time as discrete variables are usually easier to handle compared to numeric ones; and decreases the chances of overfitting since the feature space becomes less complex(Gomes et al., 2019) . Target-

ing feature discretization from data streams, a significant milestone was the Partition Incremental Discretization algorithm (PiD)(Pinto and Gama, 2005). While we have not used PiD, it is a powerful tool. PiD discretizes numeric features in two layers. The first layer is responsible for computing a high number of intervals given the arriving data, while the second uses the statistics calculated in the first layer to compute equal frequency partitions.

4.3.3 Reward System

The reward system being used currently is simply taking the number of operations and time required(in micro seconds) as the reward. A more complex state will be when the rewards are not readily available, then another method can be applied as shown in (Eysenbach et al., 2019),(Guo, 2017).

In traditional Q-learning, there is a discount factor involved to discount the moves done later on in the sequence of the game and there is a utility function involved to calculate the utility of each move for each state. The goal of the system is to maximize the expected utility.

As the game is a single move game, there is no discounting factor involved neither is a utility function required. We can use the negative of time required or the number of moves required to maximize or minimize with te current positive values. We chose to minimize the positive values.

4.3.4 Transition function

The transition function is the DQN, i.e. the reward determining system, along with the move choosing method.

The DQN will output a list of reward, which corresponds to the 24 moves. Then to choose the optimal action based on reward, we simply scan a chunk of 24 continuous rewards and associate them to moves.

As the game is a single state game, the model of DQN becomes rather simple, and makes it not necessary to choose a discount factor nor do we need to include it into the transition model.
The method to choose the optimal move is described in the next chapter formally.

4.3.5 Assumptions

Note the amount of information, features extracted from the querying do not fully represent the data set our Neural network can't perform very well. To tackle this, we look at whether the shift in the predicted rewards is similar cross all the moves.
Then given data points $(x_i, y_i), (x_j, y_j)$ and the DQN Q_θ the condition to check becomes:-

$$y_i \leq y_j \Rightarrow Q_\theta(x_i) \leq Q_\theta(x_j)$$

If this condition holds, then the predicted optimal move is the actual optimal move.
As optimal move implies

$$y_{\text{optimal}} \leq y_i \forall i \in [24]$$

This with the previous condition will imply

$$Q_\theta(x_{\text{optimal}}) \leq Q_\theta(x_i) \forall i \in [24]$$

We train a Deep neural network with 70%, use the rest of the 30% data for testing and record the results.

4.4 Conclusion

In this chapter we saw :-

- The method use to generate data for the linear road benchmark test cases.
- The method implemented to take input, converted SQL queries to $C++$ and

extracted features during execution and stored them

- Showcased how the extracted features can help with deep reinforcement learning
- Showcased the method used to evaluate the performance of DQN.

In the next chapter, we

- Formalize the assumptions we made in our evaulation.
- Present the categorize/ classification of the data set. Divided into training and testing data.
- Visualize the confusion matrix.
- Give the true positive, true negative, false positive false negative as well the analysis of the confusion matrix for all 24 orderings.
- Lastly we compare how our predictions fared against the optimal and worst orderings.

present the results of our experiments, visualize them The method to compute the results are given in the next chapter.

Chapter 5

Evaluation

5.1 Optimality checking

Now we have learnt a model Q_θ, we also have our test cases $\{(x_i, y_i)\}_{i\in[n]}$.
Each data window corresponds to 24 data entries. The t^{th} window corresponds to

$$\{x_{24t}, x_{24t+1}, ..., x_{24t+23} | t \in \mathbb{I}\}$$

Each data window has 24 data points as we have tried all possible(24 of them) combinations of ordering of operations. For each data window we need to find the ordering of operations which requires the minimum moves according to our predictions.
To do this for the t^{th} window of data, find the index of the minimum of

$$Q_\theta(x_{24t}), Q_\theta(x_{24t+1}), ..., Q_\theta(x_{24t+23})$$

, say k.
To check if the move this corresponds to is the actual optimal move, find the index of the minimum of $y_{24t}, y_{24t+1}, ..., y_{24t+23}$, say l. If $k = l$ then the move we predicted as optimal is indeed optimal.

We train the model multiple times on the training data and measure its performance on the test data each time.
To measure the performance we use the predicted optimal move vs the actual optimal

move.

5.2 Justification

The things considered while determining the neural network to use for training the DQN are :-

- To achieve a value as close as possible to the global minima for the optimization function (Adam optimzer)
- The time required to predict the optimal move should not exceed the time saved by using it.
- The time and resources required for training should not exceed the capacity of the system while it is running the query processing in the background.

What we found was:-

- Adding additional layers improve the prediction of the optimal moves but not by significant margin.
- Adding additional layers resulted in the time spent predicting the answer overshadowing the time saved by executing the optimal move.

But note, these 2 are only query and data specific findings.

5.3 Results

In this section, we present:-

- The frequency distribution of the entire dataset.
- Present our confusion matrix, heat map and the values extracte from it.

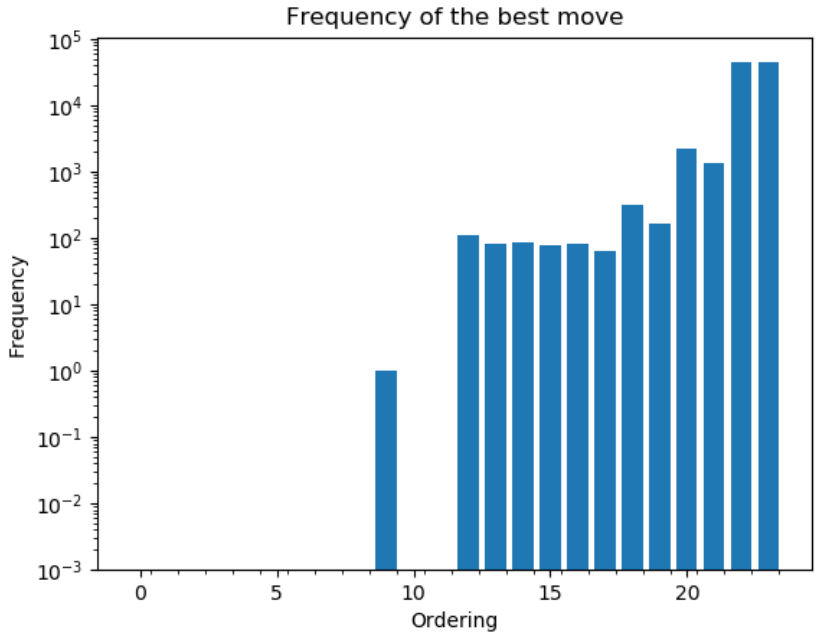


Figure 5·1: This figure shows the frequency distribution of the number of cases where each move is optimal

- Analyse various predictions for various classes.
- Present how our model performed compared.

5.3.1 Data Distrbution

As visible from the below diagram 5·1 the disparity in the frequency distribution is massive. Learning a model based on biased data is often difficult as the smaller classes tend to be ignored. As visible the data is higly biased towards the last 2 orderings.

The actual values of the above distribution are presented here5.1.

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 110, 83, 84, 79, 80, 63, 307,
   165, 2209, 1367, 43849, 43872]
```

Listing 5.1: Frequencies of optimal moves in total data

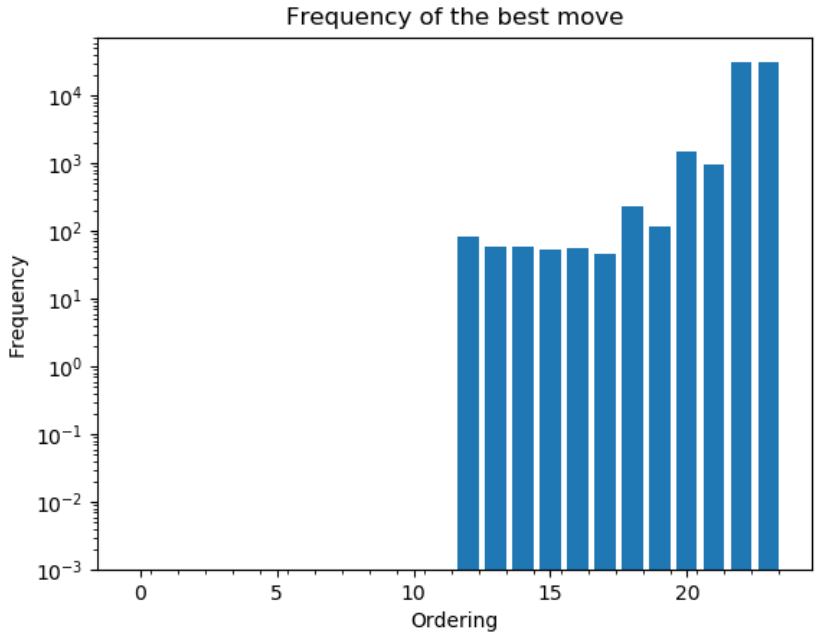


Figure 5·2: This figure shows the frequency distribution of cases where each move is optimal in the training dataset

For training purposes, we divided the data in 70% for training and 30% for testing. Cross validation did not have a significant impact on training parameters nor on the results for prediction of test data. We visualized the training and testing data below 5·2 and 5·3

The actual values for the frequency distribution are given in 5.2 and 5.3

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 81, 60, 59, 54, 57, 45, 227,
    119, 1526, 966, 30811, 30583]
```

Listing 5.2: Frequencies of optimal move in training data

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 29, 23, 25, 25, 23, 18, 80, 46,
    683, 401, 13038, 13289]
```

Listing 5.3: Frequencies of optimal move in testing data

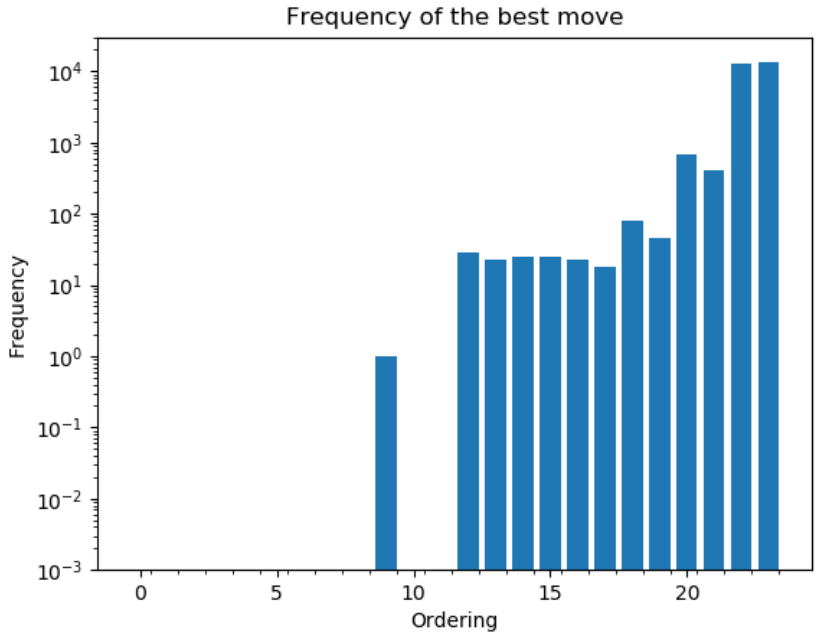


Figure 5·3: This figure shows the frequency distribution of cases where each move is optimal in the testing dataset

5.3.2 Confusion matrix

Confusion matrix or error matrix is a table that aids with visualizing the performance of an algorithm, usually a supervised algorithm, in this case for classificaiton purpose. Each row of the matrix represented a class, specifically meant to represent the predicted class by the classification algorithm, while the column is meant to represent the actual class.

In the confusion matrix $\mathbb{A}$, the entry $\mathbb{A}[i, j]$ is interpretted as, the number of data points of class j predicted to be i.

As we trained the DQN over multiple iterations, we present two of it's results here 5·4 and 5·5.

As most of the variation in the matrix is at the bottom right, we present the close up visualization of that section here 5·6 and 5·7

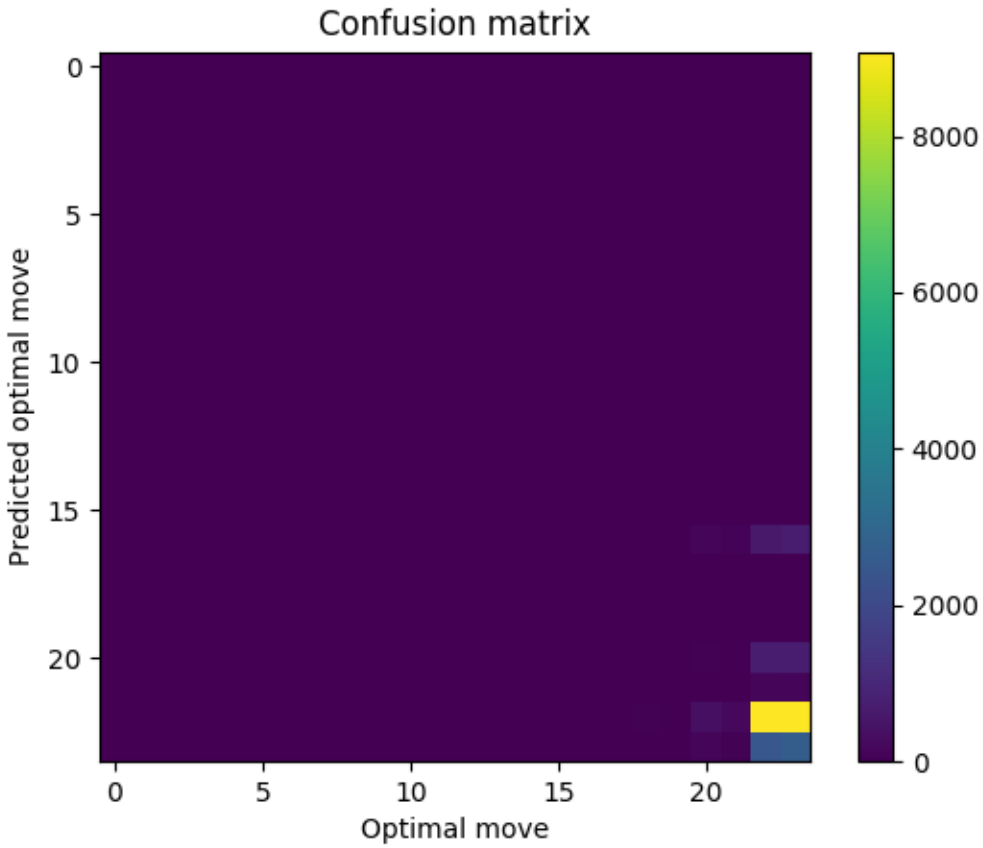


Figure 5·4: DQN Run 1 confusion matrix

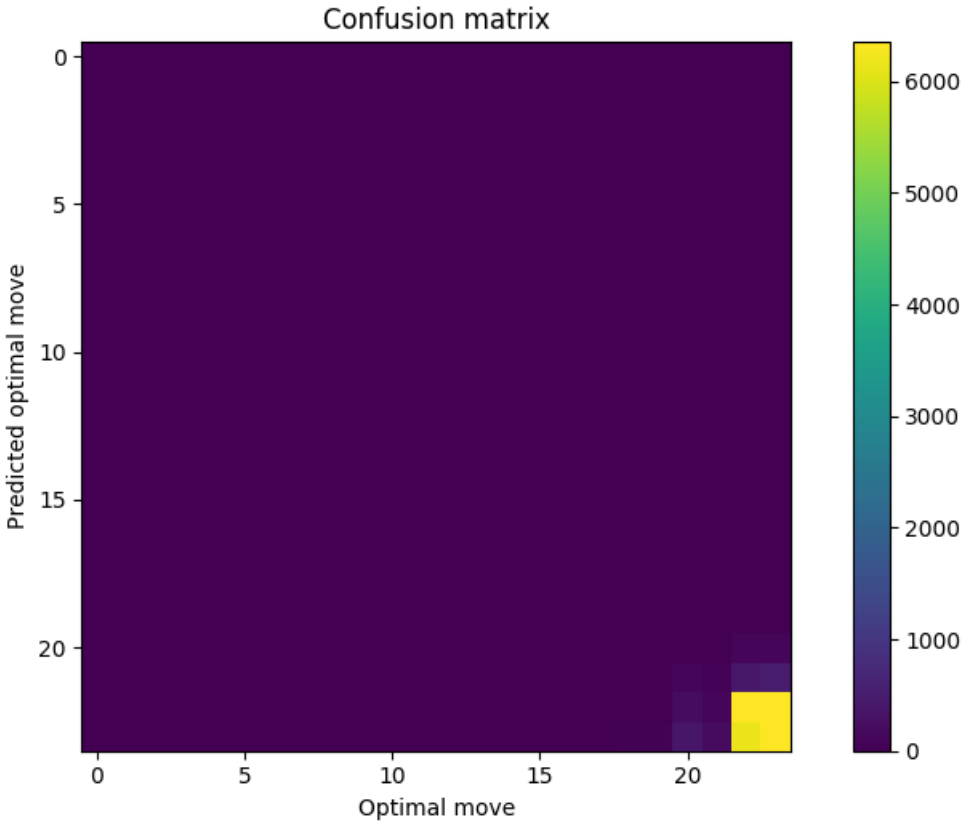


Figure 5·5: DQN Run 2 confusion matrix

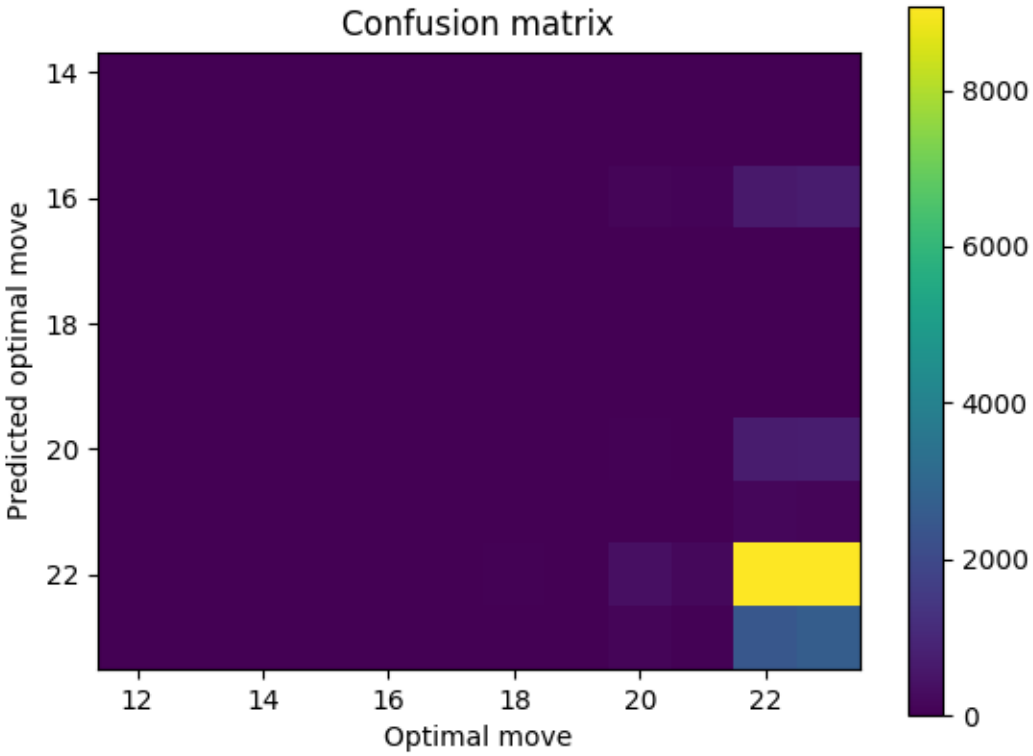


Figure 5·6: DQN Run 1 confusion matrix closeup

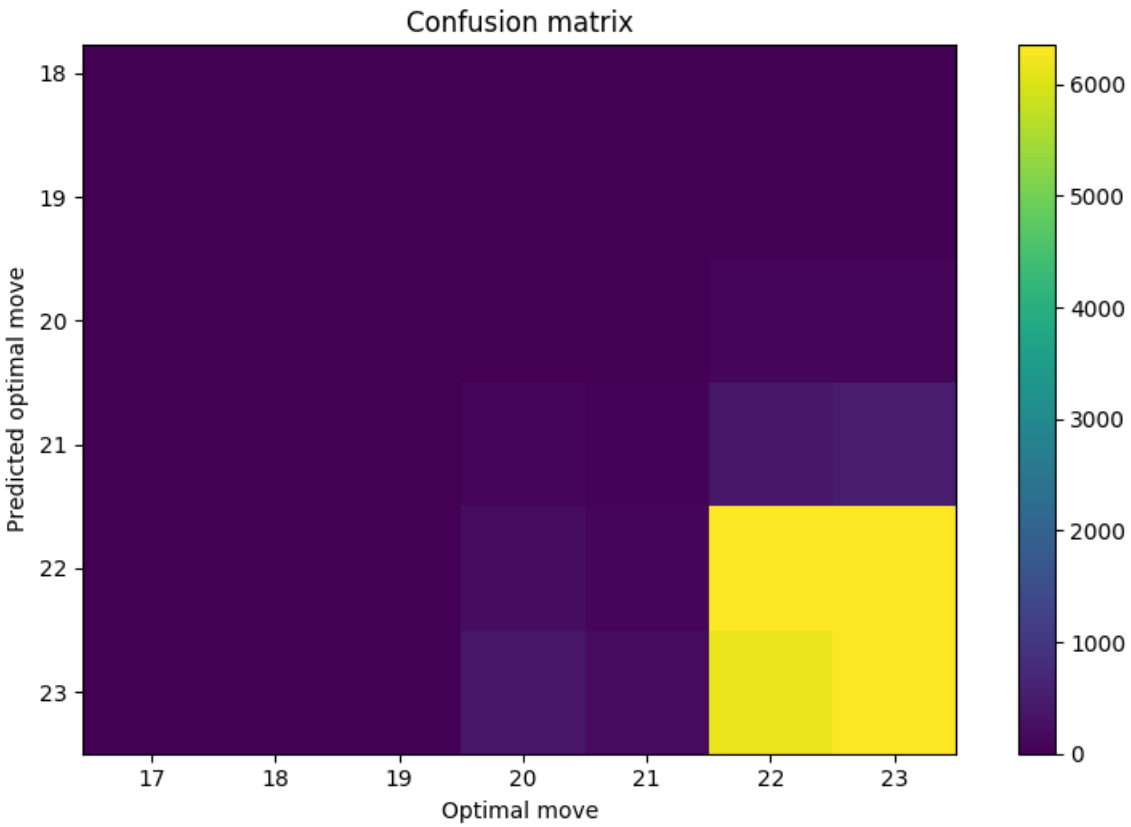


Figure 5·7: DQN Run 2 confusion matrix closeup

The acutal values used for heatmap are also presented here, due to most of the upper half not having any special variation, we truncated it and presented here 5.4.

```
[[   0    0    0    0    0    0    0    0    0    0    0    1    0]
 [   0    0    0    0    0    0    0    0    0    0    0   16   13]
 [   0    0    0    0    0    0    0    0    0    0    0   18    5]
 [   0    0    0    0    0    0    0    0    0    0    0   13   12]
 [   0    0    0    0    0    0    0    0    0    0    0   12   13]
 [   0    0    0    0    0    0    0    0    0    0    0   17    6]
 [   0    0    0    0    0    0    0    0    0    0    0    9    9]
 [   0    0    0    0    0    0    0    0    0    0   10   21   49]
 [   0    0    0    0    0    0    0    0    0    3    4   13   26]
 [   0    0    0    0    0    0    0    1    0   18   77  213  374]
 [   0    0    0    0    0    0    0    0    0   19   69  118  195]
 [   0    0    2    0    0    0    0    0    0  116  406 6347 6167]
 [   0    0    1    0    0    0    0    6    0  113  476 6338 6355]]
```

Listing 5.4: Confusion matrix for DQN classificaiton

While the data was highly biased towards the last 2 orderings, our predictions on the otherhand were comparatively diverse.

5.3.3 Analysis

A label based analysis is obtained by using "metrics.classification_report" from scikit-learn (Pedregosa et al., 2011).

```
              precision    recall  f1-score   support

           9       0.000     0.000     0.000         1
          12       0.000     0.000     0.000        29
          13       0.000     0.000     0.000        23
          14       0.000     0.000     0.000        25
          15       0.000     0.000     0.000        25
          16       0.000     0.000     0.000        23
          17       0.000     0.000     0.000        18
          18       0.000     0.000     0.000        80
          19       0.000     0.000     0.000        46
          20       0.067     0.026     0.038       683
          21       0.066     0.172     0.096       401
```

```
14              22       0.483     0.487     0.485     13038
15              23       0.481     0.478     0.479     13289
16
17     micro avg       0.462     0.462     0.462     27681
18     macro avg       0.084     0.089     0.084     27681
19  weighted avg       0.461     0.462     0.461     27681
```

Listing 5.5: Statistics for various orderings

Precision, for each class is the ratio of the number of data entries correctly predicted in this category to the number of data points predicted to be of this class. So the above can be the precision column can be interpreted as:-

- For ordering 23, of the data points predicted to have the optimal ordering as 23, 48.1% actually had it as the optimal ordering.
- For ordering 22, of the data points predicted to have the optimal ordering as 22, 48.3% actually had it as the optimal ordering.
- For ordering 21, of the data points predicted to have the optimal ordering as 21, 6.6% actually had it as the optimal ordering.

Recall, for each class if the ratio of the number of data points correctly predicted to have this class to the number of data points which actually had this as the optimal ordering. So in the above table, the recall column can be interpreted as:-

- For ordering 23, of all the data points which had ordering 23 as their optimal ordering, we predicted 47.8% to have ordering 23 as the optimal ordering.
- For ordering 22, of all the data points which had ordering 22 as their optimal ordering, we predicted 48.7% to have ordering 22 as the optimal ordering.
- For ordering 21, of all the data points which had ordering 21 as their optimal ordering, we predicted 17.2% to have ordering 21 as the optimal ordering.

F1-score, for any classification system, it is desirable to have high precision and recall while not overfitting. **F1-score** for a class, is the harmonic mean of precision and recall. It aims to give preference to neither precision nor recall. All the various metrics aim to capture various information from the confusion matrix. F1 score is better measure when False negatives and False positives are important and if there is a class imbalance.

Micro F1 is calculated by calculating **micro precision** and **micro recall** and taking their harmonic mean.
Micro Precision is calculated by adding True positive values across all the classes and adding False positives across all classes and treating them as new true positive and false positives to calculate precision.
Micro Recall is calculated by adding True positive values across all the classes and adding False negatives across all classes and treating them as new true positive and false negatives to calculate precision.

Macro F1 is a simple arithmetic mean of f1 scores of all the classes.
Macro Precision is a simple arithmetic mean of precisions of all the classes.
Macro Recall is a simple arithmetic mean of recalls of all the classes.

Weighted F1 is weighted arithmetic mean of f1 scores across all classes, with the class strength as weight.
Weighted Precision is weighted arithmetic mean of precisions across all classes, with the class strength as weight.
Weighted Recall is weighted arithmetic mean of f1 recalls across all classes, with the class strength as weight.

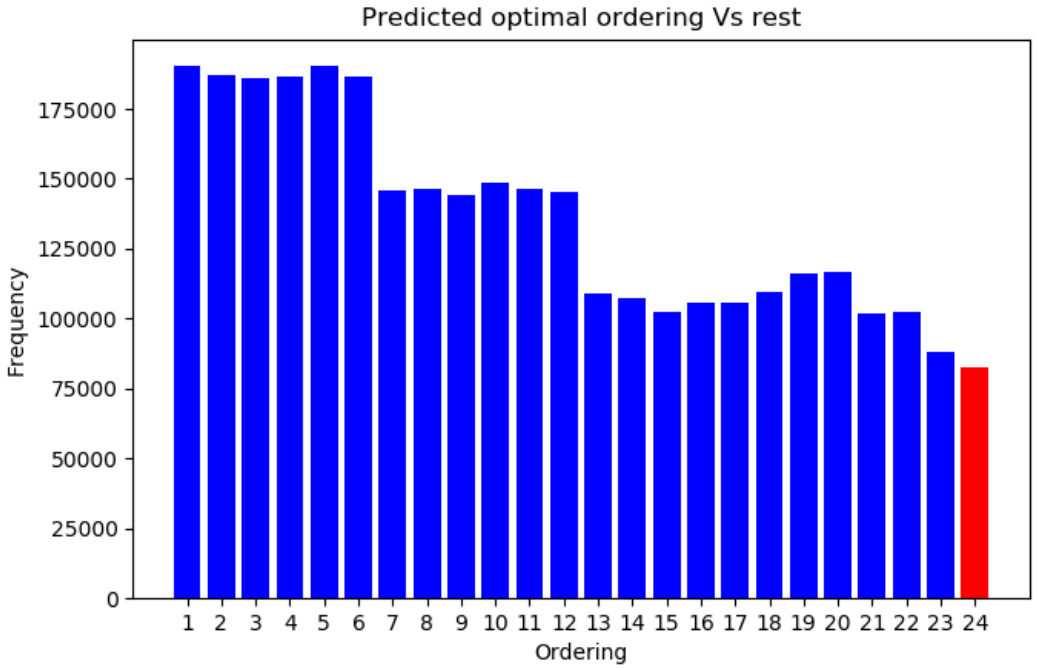


Figure 5·8: The figure shows the number of operations required to execute the query depending on the ordering of the selection operators chosen. The predicted optimal ordering is shown in red.

5.3.4 Performance

From the confusion matrix, we calculate the sum of true positives across all the classes. We see that 12978 data points had predictions correct out of 27681 data points, that is around 47%. While the total accuracy isn't very high, looking solely at the number of correct predictions isn't enough, as we are more interested in the improvement in performance offered by this model.

Below we have few examples of our predictions. The bar chart shows the number of operations required by each ordering, the predicted optimal ordering is in red. 5·8 and 5·9

Given that we have the numbr of operations required to be executed by each ordering, we can look at the sum of number of operations of the orders predicted by

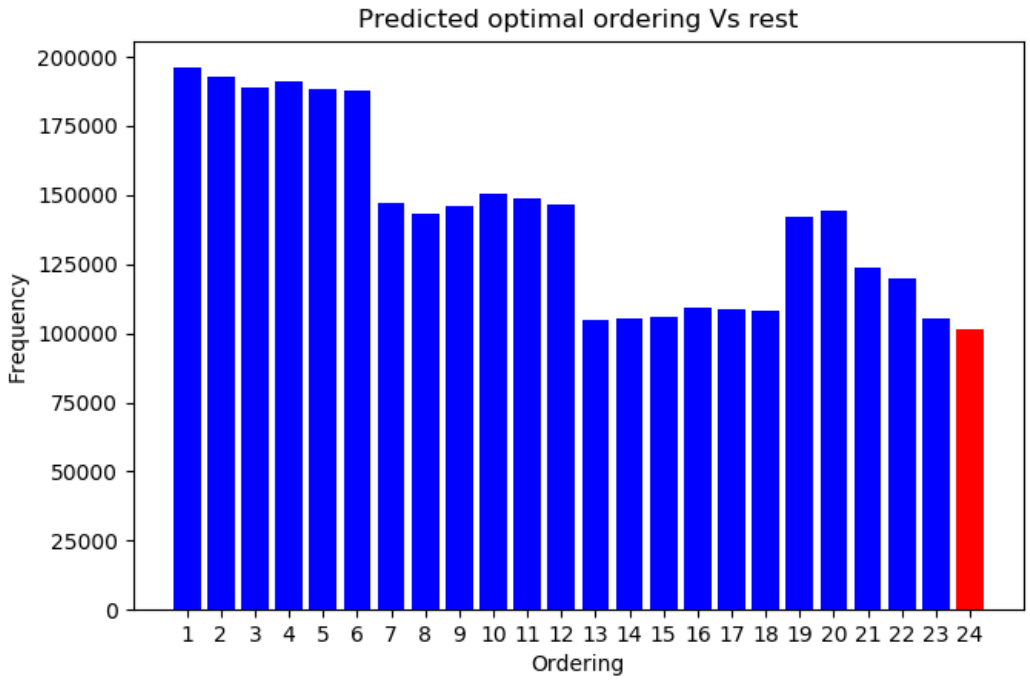


Figure 5·9: The figure shows the number of operations required to execute the query depending on the ordering of the selection operators chosen. The predicted optimal ordering is shown in red.

our model.

We have the actual optimal ordering, the actual worst ordering, the predicted best ordering from the model.

We add the number of operations required by following each of the above 3 and look at the sum of them.

- The sum of operations required by the optimal ordering 890640075.0
- The sum of operations required by the predicted ordering 900269878.0
- The sum of operations required by the optimal ordering 2310227856.0

This tells us that our predicted orderings required approximately 39% operations of the number of operations required by the worst ordering executed always and 102% operations of the number operations required by the best ordering executed always.

Below we have few examples of our predictions

5.4 Discussion

Learning from data streams is a continuous process. The learning systems that act in dynamic environments, where working conditions change and evolve, need to monitor their working conditions. They need to monitor the learning process for change detection, emergence of novel classes, changes in the relevance of features, changes in the optimal parameters settings, and others (Gomes et al., 2019).

We divide this section into the following parts:-

- **Interpretations** We will interpret the above stated results and see what they mean.

- **Implications** Why do these results matter and what can they lead to.
- **Limitations** the limitations we experienced in the experiments carried out in this thesis.

5.4.1 Interpretations

Despite the biased input data classes, the final strategy we got from the DQN gave us a performance of 101% of the optimal solution, we can safely say, DQN have proved to be useful in this case. The successive epoches of DQN also can be seen to improve the results via the diagram.

But contrary to the expected result, the number of data points for which we were able to predict to optimal move correctly are roughly 48%, the lack of accuracy across the 24 classes is concerning which can perhaps be addressed to the lack of features used in training the DQN.

Overall the improvement of optimal strategy selection is inline with the powerful nature of deep neural networks and iterative nature of reinforcement learning. The high amount of data available from data streams can definitely be a huge plus point for online training methods for DQN for query optimization.

5.4.2 Implications

The fact that the application of DQN improved performance by to such a margin, serves to prove that DQNs can be applied in at least some cases for optimizing queries on data stream. Methods for using DQN for query optimization on static data are shown here (Krishnan et al., 2019) and we use it as an motivation to apply it on data streams.

This makes a strong case for further work in this field, application of deep reinforcement learning to optimize query processing on stream data, to explore more SQL

like implementation and integration of the DQN model along with online learning feature.

As we saw in this experiment, we achieved a model which tells us an strategy for ordering which requires only 1% more operations than the optimal ordering. Application of DQN on other queries can similarly provide a huge speed up and perhaps reduction in resources consumed.

It is popular in recent AI research to try end-to-end learning, where problems that were traditionally factored into subproblems (e.g., self-driving cars involve separate modelsfor localization, obstacle detection and lane-following) are learned in a single unified model. One can imagine a similar architectural ambition for an end-to-end learning query optimizer, which simply maps subplan features to measured runtimes. Thiis would require a significant corpus of run-time data to learn from, and changes to the featurization and perhaps the deep network structure we used here. DQ is a pragmatic middle ground that exploits the structure of the join optimization problem(Krishnan et al., 2019).

5.4.3 Limitations

The work presented in this thesis has a number of limitations.

- The neural network is unable to achieve low loss for the test data, this perhaps speak the lack of features extracted during the querying time.
- The data sample we have is highly biased, leading to rather baised DQNs.
- The entire method need to be parallelized and set up on a scalable infrastructure to see the actual effects of DQN.
- The DQN used is simulating a single move game rather than a multi move game.
- Not looked at how DQNs can be trained online.

5.5 Conclusion

In this chapter we saw :-

- The method for selecting the optimal move and the assumption for it.
- The classificaiton of the data and how it is split between training and testing.
- Analyzed and visualized the confusion matrix on different runs of DQN.
- The performance of the DQN on predicting.

In the next chapter we:-

- Present the overall results of the thesis.
- Present the take aways.
- Present the ways of improving upon this work.

Chapter 6

Conclusion and Further work

6.1 Conclusion

The goal of this thesis is to act as a proof of concept for the application of Deep Reinforcement Learning for optimization of query processing on streams.
Say, for a particular window of data

- The size of SegVol $= x$.
- The size of SegAvgSpeed $= y$.

Due to the query used in the experiments, the number of operations required to execute the query lie between

$$[x * y, 4 * x * y]$$

The worst and best possible cases can both arise in the same window depending on the order used and the data in the query.
Of the 24 possible operation orders, we were able to predict the optimal order of operations around 48% cases, reducing the number of operations to around 39% of the worst observed(which is better than the actual worst).

$$x * y \leq \text{optimal} \leq \text{predicted} \leq \text{worst observed} \leq 4 * x * y$$

6.1.1 Take aways

We hope the material in this thesis is sufficient for anyone to understand the complexities of SQL and be able to work on their own versions and other improvements,

overall we hope the thesis provided the read with the following:-

- The read should now have to tools to understand inner workings of SQL
- Be familiar with Operator Algebra.
- Understand how queries are executed in SQL.
- Understand how physical plans are structed/ constructed in SQL.
- Get familiar with what data streams are and what they are used for.
- Understand what deep learning, and reinforcement learning is.
- And see how DQN are being applied for improving operation ordering.

6.2 Further Work

This thesis proposed a method for optimization of query execution on data stream and provided a demo implementation for experimentation. There are still many areas which can be improved and developed further.

6.2.1 Data Generation

As shown in chapter 5, the data generated is highly biased, there are ways to try to tackle it(Losing et al., 2018). Imbalanced datasets are characterized by one class outnum-bering the instances of the other one. The later is referred to as the minority class, while the former is identified as the majority class. These concepts can be generalized to multi class classification and other learning tasks, e.g.,regression. The imbalance may be inherent to the problem (intrinsic) or caused by some fault in the data acquisition (extrinsic). Learning from imbalanced datasets is challenging as most learning algorithms are designed to optimize for generalization, and as a consequence, the minority class may be completely ignored.(Gomes et al., 2019)

A source producing more diverse data and a query on that may lead to more interesting results. Real world data stream can be used to get better variety in the data and take input via different methods.

Some examples of data generators are :-

- A real-estate website tracks a subset of data from consumers mobile devices and makes real-time property recommendations of properties to visit based on their geo-location.(Team,)
- A solar power company has to maintain power throughput for its customers, or pay penalties. It implemented a streaming data application that monitors of all of panels in the field, and schedules service in real time, thereby minimizing the periods of low throughput from each panel and the associated penalty payouts. (Team,)
- A media publisher streams billions of clickstream records from its online properties, aggregates and enriches the data with demographic information about users, and optimizes content placement on its site, delivering relevancy and better experience to its audience.(Team,)
- An online gaming company collects streaming data about player-game interactions, and feeds the data into its gaming platform. It then analyzes the data in real-time, offers incentives and dynamic experiences to engage its players.(Team,)

6.2.2 Data Storage

As demonstrated in the implementation chapter, currently for executing the query, we store data in vectors. SQL systems use B-Trees which are cache optimized, this

can reduce the runtime by huge margin. The choice of data storage, method of access and the target used for reinforcement learning can lead to a variety of learnt models.

Few considerations that can lead to changes are:-

- Use of B-trees to improve runtime and access time.
- Use of filesystems such as AWS S3 bucket, Google Cloud Platform can affect memory access time.
- In cloud storage, the amount of traffic can often lead to delays, so the network setup is also a factor to consider.
- An experiment time requirements for query execution with various types of filesystems(HDFS, NTFS, EXT2, EXT3, EXT4, XFS, ZFS, BTRFS) can be done.

6.2.3 Features extraction

The idea behind the feature extraction was to have a fixed sized vector for the input to the neural network. Entropy is a measure of information and using the entropy itself in the neural network results in a loss of information. So different feature extractions can lead to better results. One method that can be used is:-

- For each column, we replace the entropy with a vector of $i + 1$ numbers, which represents the normalized frequency of the i most frequent elements and the last one is the entropy.

This method will allow for more information about the data to be passed on to the neural network and to get better training results.

6.2.4 Deep Reinforcement Learning

The Deep Reinforcement Learning model used was simplified, in the sense that it was a 1 move game. We start with a given state and can execute any of the 24 possible operations orders and we end the game in the final state and report the time of execution and number of operations required.

A better reinforcement learning model as created here(Krishnan et al., 2019) can be used, it represent the transition, the state changes and more information is passed along and helps train a better model. It is important to note the initial training time and memory requirement are not system breaking, it is the online learning part which needs to be efficient

6.2.5 Integration

As mentioned, the code implemented in the thesis is meant to serve as a proof of concept. The code presented mimics the execution of a SQL query on a very high level. A system can be developed which:-

- Parses text in the for of SQL query and convert it into actual SQL query.
- Applied various optimizations used in a SQL server.
- Additionally give the option to use custom optimizer at various levels to test of various implementation.
- Helps scalability, improves reliability and resilience.
- Gives options to input various evaluation metrics.

Appendix A

Code Implementation and Description

```
1
2  int main(int argc, char** argv)
3  {
4      /*
5      Tells the query execution time in microseconds for each window
       of data.
6
7      Arguments
8      ---------
9      The path to location of data file.
10
11     */
12     //The size of window we consider.
13     int window_size=20000;
14
15     FILE *pFile;
16     // The first arguement is the location of the stream.
17     pFile = fopen (argv[1],"r");
18
19     //We will store data in this array.
20     int database[window_size][4];
21
22     //Passing an empty vector to functions to help with memory
23     std::vector<int> vect;
24     vect.clear();
25     std::vector<std::vector<int>> output;
26
27     //Read till the end of input file
28     while(!feof(pFile))
29     {
30         //Start timer of current data set
31         auto start = std::chrono::high_resolution_clock::now();
```

```
32
33          //Take input of window size number of entries
34          input_database(pFile, database, window_size);
35
36          //Extract information, features
37          segAvgSpeed(database, output, vect, window_size);
38
39          //clear memory
40          output.clear();
41          vect.clear();
42
43          //get the end time
44          auto stop = std::chrono::high_resolution_clock::now();
45          auto duration = std::chrono::duration_cast<std::chrono::
    microseconds>(stop - start);
46
47          //print the time required
48          std::cout << duration.count() << std::endl;
49      }
50      fclose(pFile);
51      return 0;
52  }
```

Listing A.1: code overview for simple query

```
1  void input_database(FILE * pFile, int d[][4],int window_size)
2  {
3      /*
4      Read the file and stores the information extracted into the
    array d.
5
6      Arguments
7      ---------
8      The location of the file
9      Array to store information in
10     The amount of data entries to read
11     */
12     int i,j;
13     for(i=0;i<window_size;i++)
14     {
15         fscanf (pFile, "%i,%i,%i,%i,%i,%i,%i,%i,%i,%i,%i,%i,%i,%i
    ", &j,
16         &j,&j,&d[i][0],&d[i][1],&j,&d[i][2],&d[i][3],&j,&j,&j,&j,&j
```

```
    ,&j,&j);
17   }
18 }
```

Listing A.2: File reading function for simple query

```
1  // Adds elements of the array d into a 2d vector "output"
2  // Then sorts the vector to mimic a groupby statement
3  for(i=0;i<window_size;i++)
4  {
5      if(d[i][2]==0)
6      {
7          vect.push_back(d[i][1]);
8          vect.push_back(d[i][2]);
9          vect.push_back(d[i][3]);
10         vect.push_back(d[i][0]);
11         output.push_back(vect);
12     }
13     vect.clear();
14 }
15 std::sort(output.begin(),output.end(), my_sort);
```

Listing A.3: groupby statement for the simple query

```
1  /*
2  function return a 2d vector with an aggregation function applied
       after a groupby function is used
3  */
4  int count = 1;
5
6  //OP is the output 2d vector obtained after aggregation.
7  std::vector<std::vector<int>> op;
8
9  op.push_back(output[0]);
10 for(i=1;i<output.size();i++)
11 {
12     //This is condition to check if we are in the same group
13     if((output[i][0]==output[i-count][0]))
14     {
15         if(output[i][1]==output[i-count][1])
16         {
17             if((output[i][2]==output[i-count][2]))
18             {
```

```
                // incase of same group make adjustments in preparation for aggregation function
                op.back()[3]=op.back()[3]+output[i][3];
                count++;
            }
        }
    }
    else
    {
        // if new group has started apply the aggregation to the last group
        op.back()[3]=op.back()[3]/(count);
        op.push_back(output[i]);
        count = 1;
    }
}
op.back()[3]=op.back()[3]/count;
output.clear();
output=op;
op.clear();
```

Listing A.4: applying aggregation function for simple query

```


int main(int argc, char** argv)
{
    /*
    Function outputs the data required to perform deep reinforcement learning for query optimization

    Arguments
    ---------
    The path to location of data file.

    */
    int window_size=10000;

    //Location of the data file
    FILE *pFile;
    pFile = fopen (argv[1],"r");

    //Output file
```

```
20      std::ofstream op_file;
21      op_file.open("q3out.txt");
22
23      std::vector<std::vector<int>> database, curCarSeg, segAvgSpeed,
    segVol;
24
25      int read_times=0;
26      int temp;
27
28      //Read the file till end
29      while(!feof(pFile))
30      {
31          //Take window size number of inputs
32          input_database(pFile, database, window_size);
33          read_times++;
34          printf("read times is, %d\n", read_times);
35
36          //Calculate the cucarseg relation
37          curcarseg(database, curCarSeg);
38          temp=curCarSeg.size();
39          printf("    CurCarSeg size is, %d\n", temp);
40
41          //Calculate the segavgspeed relation
42          segavgspeed(database,segAvgSpeed);
43          temp=segAvgSpeed.size();
44          printf("    SegAvgSpeed size is, %d\n", temp);
45
46          //calculate the segvol relation
47          segvol(curCarSeg,segVol);
48          temp=segVol.size();
49          printf("    SegVol size is, %d\n", temp);
50
51          // Execute the segtoll query.
52          segtoll(segAvgSpeed,segVol,op_file);
53
54          //clear memory
55          curCarSeg.clear();
56          segAvgSpeed.clear();
57          segVol.clear();
58          database.clear();
59
60      }
```

```
    fclose(pFile);
    op_file.close();
    return 0;
}
```

Listing A.5: Code plan overview for complex query

```
void segavgspeed(std::vector<std::vector<int>> &d, std::vector<std::
    vector<int>> &s)
{
    /*
    Function outputs the segavgspeed relation

    Arguments
    ---------
    The database
    The output vector variable

    */

    //d=(Time, carID, expressway, Dir, Seg, Position,speed)

    std::vector<std::vector<int>> temp;
    int cur_time=0;
    int i,j;
    std::vector<int> vect;
    vect.resize(4);

    //Figure the latest time in the database
    for(i=0;i<d.size();i++)
    {
        if(cur_time<d[i][0])
        {
            cur_time=d[i][0];
        }
    }

    //Filter out the entries in the last 5 mins
    for(i=0;i<d.size();i++)
    {
        if(d[i][0]>cur_time-300)
        {
            vect[0]=d[i][2];
```

```
36              vect[1]=d[i][3];
37              vect[2]=d[i][4];
38              vect[3]=d[i][6];
39              temp.push_back(vect);
40          }
41      }
42
43      //sort to mimic applying a groupby function
44      std::sort(temp.begin(),temp.end(), my_sort);
45
46      //(express,dir,seg, average speed)
47      int count = 1;
48      s.push_back(temp[0]);
49
50      //Apply the aggregation function
51      for(i=1;i<temp.size();i++)
52      {
53          if(temp[i][0]==temp[i-1][0] and temp[i][1]==temp[i-1][1] and
        temp[i][2]==temp[i-1][2])
54          {
55              s.back()[3]=s.back()[3]+temp[i][3];
56              count++;
57          }
58          else
59          {
60              s.back()[3]=s.back()[3]/(count);
61              s.push_back(temp[i]);
62              count = 1;
63          }
64      }
65      s.back()[3]=s.back()[3]/count;
66      /*for(i=0;i<s.size();i++)
67      {
68          for(j=0;j<s[i].size();j++)
69          {
70              printf("%d ",s[i][j]);
71          }
72          printf("\n");
73      }*/
74      temp.clear();
75  }
```

Listing A.6: Calculating the SegAvgSpeed relation

```
void segvol(std::vector<std::vector<int>> &d, std::vector<std::
    vector<int>> &s)
{
    /*
    Function outputs the segvol relation

    Arguments
    ---------
    The database
    The output vector variable

    */

    //d=(carid,exp,dir,seg)
    //sort the vector inorder to mimic the application of groupby
    std::sort(d.begin(),d.end(), my_sort2);

    int i,j;
    s.push_back(d[0]);
    s.back()[0]=1;

    //Apply the aggregation function
    for(i=1;i<d.size();i++)
    {
        if(d[i][3]==d[i-1][3] and d[i][1]==d[i-1][1] and d[i][2]==d[
    i-1][2])
        {
            s.back()[0]++;
        }
        else
        {
            s.push_back(d[i]);
            s.back()[0]=1;
        }
    }
}
```

Listing A.7: Calculating the SegVol relation

```
void segtoll(std::vector<std::vector<int>> &s, std::vector<std::
    vector<int>> &v, std::ofstream &op_file)
{
    /*
```

```
    Function mimics the execution of the segtoll query which will
   generate the data required for training and evaluating the DQN


    Arguments
    ---------
    The segVol relation
    The SegAvgSpeed relation
    The file to generated data

    */

    int c1,r1,i,j,k,operation;

    //Caluclate the entropy of the two relations column wise
    std::vector<float> entropy;
    entropy=entropy_calc(s,v);

    //Store the columnwise entropy in the file
    for(i=0;i<entropy.size();i++)
    {
       // fprintf(op_file, "%f ", entropy[i]);
        op_file<<entropy[i]<<" ";
    }


    op_file<<std::endl;
    //Store the size of the 2 relations
    op_file<<s.size()<<" "<<v.size()<<std::endl;
    entropy.clear();

    //v=(count of cars,exp,dir,seg)
    //s=(express,dir,seg, average speed)
    int times;
    std::vector<int> order;
    order.push_back(0);
    order.push_back(1);
    order.push_back(2);
    order.push_back(3);

    int max_times=24;
    int num_op;
```

```
45      bool b;
46
47      //Fix the order of operations to execute and start the loop
48      for(times=0;times<max_times;times++)
49      {
50          num_op=0;
51          //start the timer
52          auto start = std::chrono::high_resolution_clock::now();
53          for(i=0;i<s.size();i++)
54          {
55              for(j=0;j<v.size();j++)
56              {
57                  b=false;
58                  for(k=0;k<4;k++)
59                  {
60                      operation=order[k];
61                      num_op++;
62
63                      //Look at the operation to execute and do the
    needful
64                      switch(operation)
65                      {
66                          case 0:
67                              if(s[i][0]!=v[j][1])
68                              {
69                                  b=true;
70                              }
71                          case 1:
72                              if(s[i][1]!=v[j][2])
73                              {
74                                  b=true;
75                              }
76                          case 2:
77                              if(s[i][2]!=v[j][3])
78                              {
79                                  b=true;
80                              }
81                          case 3:
82                              if(s[i][3]>=40)
83                              {
84                                  b=true;
85                              }
```

```
                    }
                    // if any of the cases above give true, stop.
                    if(b)
                    {
                        break;
                    }
                }
            }
        }

        // get the end time and calculate the time required.
        auto stop = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::
    microseconds>(stop - start);

        //Store the order of operations
        for(i=0;i<4;i++)
        {
            op_file<<order[i]<<" ";
        }

        //Store the amount of time taken to execute the query and
    the number of operations required.
        op_file<<std::endl<<duration.count()<<" "<<num_op<<std::endl
    ;

        //Generate the next permutation which is the next order of
    operations to try.
        next_perm(order);
     }
}
```

Listing A.8: The SegToll query which will generate data for DQNs

```
def categorize(a):
    '''
    Returns the frequencies of the best moves/ordering

    Arguments
    ---------
    The data to categorize
    '''
    category=[0]*24
```

```
    seen=0
    m=0
    for j in range(len(a)):
        if(seen == 0):
            m=j
        if(a[j]<a[m]):
            m=j
        if(seen==23):
            seen=-1
            category[m%24]+=1
        seen+=1
    return(category)

def analysis(p,y):
    '''
    The function returns an analysis of the based on the predictions
    p and the actual values y

    Arguments
    ---------
    predictions p
    actual values y
    '''
    #Start with a confusion matrix
    mat=np.zeros((24,24))

    seen=0
    correct=0
    best=0
    pred=0
    wors=0
    act=[]
    pre=[]

    #need to track the total of best moves, total of worst moves,
   total of predicted moves
    #need to record the predicted best move vs the actual best move
   for confusion matrix

    for j in range(len(p)):
        if(seen==0):
            mi1=j
```

```
            mi2=j
            ma=j
        if(p[j]<p[mi1]):
            mi1=j
        if(y[j]<y[mi2]):
            mi2=j
        if(y[j]>y[ma]):
            ma=j
        if(seen==23):
            seen=-1
            best+=y[mi2]
            pred+=y[mi1]
            wors+=y[ma]
            act.append(mi2%24)
            pre.append(mi1%24)
            mat[mi1%24,mi2%24]+=1
            if(mi1==mi2):
                correct+=1
        seen+=1

    #print out the times our prediction was correct
    print(correct, len(p)/24)

    #the sum of moves for best, predicted, worst and their ratios.
    print(best, pred, wors, pred/best, pred/wors)

    #print the confusion matrix
    print(metrics.confusion_matrix(act,pre))

    #print the analysis of the confusion matrix
    print(metrics.classification_report(act, pre, digits=3))

    #Calculate the true positives
    tp = np.asarray([mat[i,i] for i in range(24)])

    #calculate the false positives
    fp = np.sum(mat, axis=1)-tp

    #calculate the False negatives
    fn = np.sum(mat, axis=0)-tp

    #calculate the true negatives
```

```
    tn = np.sum(mat)-(tp+fp+fn)

    #print them out
    for i in range(24):
        print(tp[i],fp[i],fn[i],tn[i])

    #plot the confusion matrix
    plt.imshow(mat)
    plt.colorbar()
    plt.show()

def reinforcement_learning(fin_train,save_loc,epoch_number):
    '''
    This function trains a DQN and tests its predictions on another
   set of data and returns the results found

    Arguments
    ---------
    The total data
    location to save the model
    number of times to train

    '''
    data_points = len(fin_train)/24
    bp = int(data_points*0.7)

    bp = bp*24

    x_tr=fin_train[:bp,:-2]
    #time
    y1_tr=fin_train[:bp,-1]
    #operations
    y2_tr=fin_train[:bp,-2]

    x_te=fin_train[bp:,:-2]
    y1_te=fin_train[bp:,-1]
    y2_te=fin_train[bp:,-2]

    #modeling our DNN
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Dense(34,activation='relu'))
    model.add(tf.keras.layers.Dense(68,activation='relu'))
```

```
    model.add(tf.keras.layers.Dense(68,activation='relu'))
    model.add(tf.keras.layers.Dense(34,activation='relu'))
    model.add(tf.keras.layers.Dense(1))
    model.compile(optimizer='adam', loss='mse')

    #Training data classification
    cat1=categorize(y2_tr)
    print(cat1)
    #Testing data classification
    cat2=categorize(y2_te)
    print(cat2)

    times = int(epoch_number)
    for i in range(times):
        history=model.fit(x_tr, y2_tr, batch_size=100, epochs=1)
        ans=model.predict(x_te)
        analysis(ans,y2_te)
```

Listing A.9: The code to train a DQN for the complex query

```
SET QUOTED_IDENTIFIER ON
SET ANSI_NULLS ON
GO

CREATE FUNCTION dbo.<inline_function_name, sysname, udf_> (
<@parm1, sysname, @p1> <parm1_data_type, , int> -- <
   parm1_description, ,>
, <@parm2, sysname, @p2> <parm2_data_type, , int> -- <
   parm2_description, ,>
, <@parm3, sysname, @p3> <parm3_data_type, , int> -- <
   parm3_description, ,>
)   RETURNS TABLE
    WITH SCHEMABINDING -- Or relevant comment.
/*
* description goes here
* Related Functions:
* Attribution: Based on xxx by yyy found in zzzzzzzzzzzz
* Maintenance Notes:
* Example:

select * FROM dbo.<inline_function_name, sysname, udf_>
    (<parm1_test_value, , 1>, <parm2_test_value, , 2>, <
   parm3_test_value, , 3>)
```

```
* Test:
* Test Script: TEST_<inline_function_name, sysname, udf_>
* History
* When          Who     Description
* ------------- ------- ----------------------------------------
* <date created,smalldatetime, YYYY-MM-DD>             <your
   initials,char(8), XXX>
Initial Coding
****************************************************************/
AS RETURN

SELECT
    FROM
    WHERE
    GROUP BY
    HAVING
    ORDER BY
GO

GRANT SELECT ON [dbo].[<inline_function_name, sysname, udf_>]
    TO [PUBLIC]
-- GRANT INSERT, UPDATE, DELETE ON
--        [dbo].[<inline_function_name, sysname, udf_>]  TO [PUBLIC]
GO
```

Listing A.10: Inline function template

```
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_NULLS ON
GO

CREATE FUNCTION dbo.<table_function_name, sysname, udf_> (
    <parm1, sysname, @p1> <parm1_data_type, , int> -- <
   parm1_description, ,>
  , <parm2, sysname, @p2> <parm2_data_type, , int> -- <
   parm2_description, ,>
  , <parm3, sysname, @p3> <parm3_data_type, , int> -- <
   parm3_description, ,>
)   RETURNS TABLE (
    <col1_Name, , [ID]>   <col1_Type, ,int>-- <col1_description, ,>
    , <col2_Name, , [Desc]> <col2_Type, ,int>-- <col2_description,
   ,>
```

```
13      , <col3_Name, , [xxxx]> <col3_Type, ,int>-- <col3_description,
        ,>
14      , PRIMARY KEY (<col1_Name, , [ID]>)
15          )
16  WITH SCHEMABINDING -- Or comment about why not
17  /*
18  * description goes here
19  * Related Functions:
20  * Attribution: Based on xxx by yyy found in zzzzzzzzzzzz
21  * Maintenance Notes:
22  * Example:
23      SELECT * FROM dbo.<table_function_name, sysname, udf_>(<
        value_for_@param1, , 1>,
24      <value_for_@param2, , 2>, <value_for_@param3, , 3>)
25      * Test Script: TEST_<table_function_name, sysname, udf_>
26      * History:
27      * When          Who     Description
28      * ------------- -------
        ----------------------------------------
29      * <date created,smalldatetime, mm/dd/yy>              <your
        initials,char(8), XXX>
30      Initial Coding
31      ************************************************************
        */
32      AS BEGIN
33
34      DECLARE
35
36      RETURN
37      END
38      GO
39
40      GRANT SELECT ON [dbo].[<table_function_name, sysname, udf_>]
41          TO [PUBLIC]
42      GO
```

Listing A.11: Multistatement UDF template

References

Abadi, D., Ahmad, Y., Balazinska, M., etintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. (2005). The design of the borealis stream processing engine. In *CIDR*.

Amini, L., Jain, N., Sehgal, A., Silber, J., and Verscheure, O. (2006). Adaptive control of extreme-scale stream processing systems. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, pages 71–71.

Arasu, A., Babu, S., and Widom, J. (2006). The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121142.

Arpaci-Dusseau, R. H., Anderson, E., Treuhaft, N., Culler, D. E., Hellerstein, J. M., Patterson, D., and Yelick, K. (1999). Cluster i/o with river: Making the fast case common. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, IOPADS '99, page 1022, New York, NY, USA. Association for Computing Machinery.

Avnur, R. and Hellerstein, J. M. (2000). Eddies: Continuously adaptive query processing. *SIGMOD Rec.*, 29(2):261272.

Brito, A., Fetzer, C., Sturzrehm, H., and Felber, P. (2008). Speculative out-of-order event processing with software transaction memory. In *Proceedings of the Second International Conference on Distributed Event-Based Systems*, DEBS '08, page 265275, New York, NY, USA. Association for Computing Machinery.

Caneill, M., El Rheddane, A., Leroy, V., and De Palma, N. (2016). Locality-aware routing in stateful streaming applications. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, New York, NY, USA. Association for Computing Machinery.

Carney, D., Çetintemel, U., Rasin, A., Zdonik, S., Cherniack, M., and Stonebraker, M. (2003). Operator scheduling in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, page 838849. VLDB Endowment.

Chakravarthy, S. and Jiang, Q. (2009). *Stream Data Processing:A Quality of Service Perspective*. Springer.

Chen, J., DeWitt, D. J., Tian, F., and Wang, Y. (2000). Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, page 379390, New York, NY, USA. Association for Computing Machinery.

Cherniack, M., Galvez, E., Maskey, A. S., Pedan, I., Ryvkina, E., Arasu, A., Maier, D., Stonebraker, M., and Tibbetts, R. (2004). Linear road.

Cory Stegemoller, M. M. (2016). Linear road data generator.

Eysenbach, B., Tyo, J., Gu, S., Salakhutdinov, R., Lipton, Z., and Levine, S. (2019). Reinforcement learning with unknown reward function.

Fan, J., Yang, Z., Xie, Y., and Yang, Z. (2020). A theoretical analysis of deep q-learning.

Forgy, C. (1991). Rete: a fast algorithm for the many pattern/many object pattern match problem. *Expert Systems*, pages 324–341.

Garcia-Molina, H., Ullman, J. D., and Widom, J. (2002). *Database Systems:The Complete Book*. Prentice Hall.

Gedik, B., Wu, K., and Yu, P. S. (2008). Efficient construction of compact shedding filters for data stream processing. *2008 IEEE 24th International Conference on Data Engineering*, pages 396–405.

Gomes, H. M., Read, J., Bifet, A., Barddal, J. P., and Gama, J. a. (2019). Machine learning for streaming data: State of the art, challenges, and opportunities. 21(2):622.

Gordon, M. I., Thies, W., and Amarasinghe, S. (2006). Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGOPS Oper. Syst. Rev.*, 40(5):151162.

Graefe, G. (1990). Encapsulation of parallelism in the volcano query processing system. *SIGMOD Rec.*, 19(2):102111.

Guo, X. (2017). Deep learning and reward design for reinforcement learning.

Gupta, S., Purandare, S., and Ramachandra, K. (2020). Aggify: Lifting the curse of cursor loops using custom aggregates. In *ACM SIGMOD 2020*. ACM.

Khandekar, R., Hildrum, K., Parekh, S., Rajan, D., Wolf, J., Wu, K.-L., Andrade, H., and Gedik, B. (2009). Cola: Optimizing stream processing applications via graph partitioning. In Bacon, J. M. and Cooper, B. F., editors, *Middleware 2009*, pages 308–327, Berlin, Heidelberg. Springer Berlin Heidelberg.

Krishnan, S., Yang, Z., Goldberg, K., Hellerstein, J. M., and Stoica, I. (2019). Learning to optimize join queries with deep reinforcement learning.

Losing, V., Hammer, B., and Wersing, H. (2018). Incremental on-line learning: A review and comparison of state of the art algorithms. *Neurocomputing*, 275:1261–1274.

Martin Hirzel, Robert Soule, B. G. (2018). Stream query optimization.

Ng, A. (2020). Reinforcement learning and control.

Novick, A. (2004). *Transact-SQLUser-DefinedFunctions*. Wordware Publishing.

Ottoni, G., Rangan, R., Stoler, A., and August, D. (2005). Automatic thread extraction with decoupled software pipelining. pages 12 pp.–.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., and Seltzer, M. (2006). Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49.

Pinto, C. and Gama, J. (2005). Partition incremental discretization. *2005 portuguese conference on artificial intelligence*, pages 168–174.

Ramachandra, K. and Park, K. (2019). Blackmagic: Automatic inlining of scalar udfs into sql queries with froid. *Proceedings of VLDB*, 12(12):1810–1813.

Ramachandra, K., Park, K., Emani, K. V., Halverson, A., Galindo-Legaria, C., and Cunningham, C. (2017). Froid: Optimization of imperative programs in a relationaldatabase.

Schneider, S., Hirzel, M., Gedik, B., and Wu, K. (2015). Safe data parallelism for general streaming. *IEEE Transactions on Computers*, 64(2):504–517.

Sermulins, J., Thies, W., Rabbah, R., and Amarasinghe, S. (2005). Cache aware optimization of stream programs. *SIGPLAN Not.*, 40(7):115126.

Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., and Stonebraker, M. (2003). Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, page 309320. VLDB Endowment.

Team, A. Streaming data.

Welsh, M., Culler, D., and Brewer, E. (2001). Seda: An architecture for well-conditioned, scalable internet services. 35(5):230243.

Widom, J., Motwani, R., Arasu, A., Babcock, B., Babu, S., Srivastava, U., Wang, Z., Cieslewicz, J., Datar, M., Ito, K., McAlister, J., Nishizawa, I., Rosenstein, J., Srinivasan, V., and Thomas, D. (2004). Stream project page for linear road benchmark.

Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.-L., and Fleischer, L. (2008). Soda: An optimizing scheduler for large-scale stream-based distributed computer systems. volume 5346, pages 306–325.

Xu, H., Li, Y., Tian, Y., Darrell, T., and Ma, T. (2018). Algorithmic framework for model-based reinforcement learning with theoretical guarantees. *CoRR*, abs/1807.03858.

Yu, Y., Gunda, P. K., and Isard, M. (2009). Distributed aggregation for data-parallel computing: Interfaces and implementations. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 247260, New York, NY, USA. Association for Computing Machinery.