Towards Learned Access Path Selection:
Using Artificial Intelligence to Determine the Decision Boundary of Scan vs Index Probes in Data Systems

Angelo Kastroulis

A Thesis in the Field of Information Technology
for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

November 2019

Abstract

All data systems require optimization. Determining the most performant recipe for data retrieval is commonly referred to as *access path selection* (APS). This paper introduces the use of *artificial neural networks* for query optimization in APS.

The problems with query optimizers are well known. Promising research in other areas of query optimization using machine learning techniques continues at a rapid pace. Research has proposed machine learning solutions for join order enumeration, cardinality estimation, and predicting optimizer heuristics (such as cost estimation and table statistics). However, a practical method for learned APS remains an open research topic.

APS is difficult because an optimizer must be aware of the ever-changing system state, hardware, and data. Incorrect assumptions in any of those can be very costly, and finding a solution requires years of research.

In this thesis, I present an artificial intelligence-based approach to APS and introduce a learned optimization method using neural networks. Moreover, I explore the challenges inherent in applying generalized neural network techniques to APS. I empirically show that these networks significantly outperform the query optimizer's accuracy in determining the truly performant access path.

Dedication

This thesis is dedicated to my family and closest friends.

To my dearest friend, *Mark McCumber*, who keeps me sane and focused spiritually.

To my dear wife, *Náture* without whose loving support and encouragement I would be nothing.

To my children, *Elena* and *Evan*, who no longer share dad with Harvard.

To the memory of my father, *Aris Kastroulis*, whose genuine example still inspires me every day. This is for you.

Acknowledgements

I wish I had the words to express the gratitude and appreciation for the gift my thesis director, Professor Stratos Idreos, has bestowed upon me. He is not only my mentor and guide, but because of his kindness and attention, one of my personal friends. His courses (CS165/265, Data Systems) opened my eyes the way only the rigor of writing a database kernel can. He has always supported and encouraged my intellectual pursuits and I am grateful to be included among his research staff at the Harvard Data Systems Laboratory at the Paulson School of Engineering and Applied Science.

I am thankful to my fellow researchers (Manos Athanassoulis, Michael Kester, and Abdul Wasay) at the Harvard Data Systems Laboratory for access to their thoughts as I developed this concept.

I am indebted to my thesis advisor, Dr. Sylvain Jaume, for his support in writing this thesis.

Contents

List of Figures

List of Tables

List of Code

Chapter 1: Introduction

All data systems require optimization; whether for storage or access to information. For a given query, many decisions have to be made to determine the most performant method for retrieving results. The optimizer must determine *how* to read the data (indexes and algorithms) and *which* resources (parallelism, memory, and CPU) to use.

As Bruce Lindsay, Edgar F. Codd Innovations Award Winner in 2012, put it, "three things are important in the database world: performance, performance, performance". Performance is many times at odds with flexibility. There is a tradeoff between a generalized system providing flexibility at the cost of performance, and a specialized system delivering performance at the cost of flexibility.

A usable system must have at least some flexibility because it is difficult to know how a system will be used. Even in a known, specific application, there are variations in workload over time that require different tunings. Additionally, the point of any data system is to support any workload that applications require. Optimization is the result of years of research, shared knowledge, and testing. That makes systems incredibly complex and results in a necessarily opaque implementation, carefully exposing knobs that can be used to influence key aspects of the optimizer.

For example, a user can not tell the system precisely in which instances it should consider a scan or how many workers to use but can turn them on and off entirely. A user can influence how expensive the optimizer perceives multithreading

and other functions. The method can vary from system to system (MySQL allows per-query hinting while PostgreSQL supports only system-level knobs).

To make matters worse, the correct decisions an optimizer must make change with the query and the data itself. A hint that is right for a small table becomes wrong as the table grows and can even become anti-performant. All of the rich experience gained during the painful discovery process on one system does not make its way back to the project community for optimizer improvement.

A system specialized to a particular task is constrained to that scenario. The same issues mentioned above plague even this highly-tuned system; as data and hardware change, the system requires re-development.

Conceivably, system knobs have some optimal setting. It is impossibly complex to expose the inner workings of the optimizer to an expert user, but a system smart enough to improve itself could find the optimum configuration.

Today, artificial intelligence (AI) provides exciting opportunities given their exceptional capability for optimization problems. Machine learning and deep neural networks have shown that they can make recommendations, convert speech to text, or classify images with surprising accuracy.

AI could be used to replace the generalized optimizer with a specialized optimizer trained on a particular workload. Further, it could form the basis of a system that improves itself by continuously measuring results and re-training the model, creating a perpetually optimal system.

ID	...	Price
1	...	20
2	...	5
3	...	100
4	...	50
...	...	...
98	...	12
99	...	87
100	...	50

Table 1.1: An illustrative dataset.

1.1. Background

1.1.1 Access Path Selection

There are a variety of algorithms and data structures a data system can employ to return query results. Every approach has tradeoffs. The decision an optimizer has to make on how best to retrieve data is called *access path selection*.

Scan

One approach is to scan data to find rows matching criteria in the query's predicate. Consider the data in Table 1.1 and the following query:

```
SELECT Price FROM Dataset WHERE
  Price >= someMin AND Price < someMax
```

The query requests all of the prices between some minimum and maximum value. If $someMin = 1$ and $someMax = 101$, we would be requesting all of the price data in the table. The code for a typical scan used in a column-store database would look something like this:

```
int *output = new array(sizeOf(input))
      for (i=0;i<tuples;i++,input++)
```

```
            if *input>=low && *input<high
                    *output++=i
    return output
```

The tight `for loop` evaluates all values in memory in search of matching tuples. It works well for the scenario we outlined (known as the case of *high query selectivity*). Scanning is the fastest method of searching for results since we need to traverse the entire array, regardless.

Indexes

A scenario where $someMin = 10$ and $someMax = 12$ would result in very few rows returned (referred to as *low selectivity*). In such a case, a scan is highly inefficient because we have to scan many values (moving them in and out of memory) only to discard them. If we had some intuition as to exactly where they are, we could locate the few we need and move directly to their location in memory. Of course, the savings we would gain would have to be worth the computational and memory cost of building the intuition.

Indexes built on B+ Trees are one well-established technique for building intuition. A B+ Tree is a tree (as shown in Figure 1.1) in which each group of nodes contains n keys (where n is the number of keys able to fit in cache memory). Each group of nodes links to other nodes. We begin with the root node, which indicates that keys with values < 3 are located by traversing down one level to the left relation, keys between 3 and 5 to the middle, and keys > 5 to the right. Moving from node to node incurs a cache miss because the keys have to be evacuated from memory and replaced by the next batch. However, trees are shallow, allowing travel down the tree with only a hand full of cache misses. In comparison, a scan would incur a cache miss every time we move n values into memory.

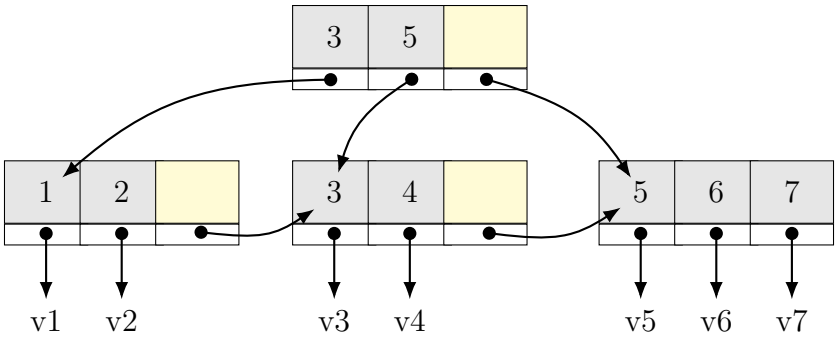


Figure 1.1: A example B+ Tree.

This kind of structure is not free. There is a cost to storing data redundantly, building the tree, and keeping it current. However, in the case of low selectivity, the cost is generally worth the savings in read performance.

Deciding Between Index and Scan

Determining the most performant route to take has been a heavily researched topic since the 1970s (Selinger et al., 1979). The conventional solution is to compute and store statistics of the data so that we predict how many records we must traverse (thus giving us some intuition as to the approach to take). If the query is highly selective, we can use the faster scan approach, and conversely, with low selectivity, choose the index.

Even with the guidance of statistics, the optimizer is still guessing, possibly incorrectly. Statistics may be out of date or poorly represent the distribution of the data. Recomputing them consumes valuable resources better spent on query workloads loads.

Recent work by Kester et al. (2017) shows that selectivity is not the only factor to be considered. That work proposes a decision boundary that takes concurrency into account, as shown in Figure 1.2. That algorithm is light-weight and fixed, identifying a mathematical proportion for a given set of parameters. Hardware characteristics would be set once upon system setup, then query concurrency, selectivity, and the

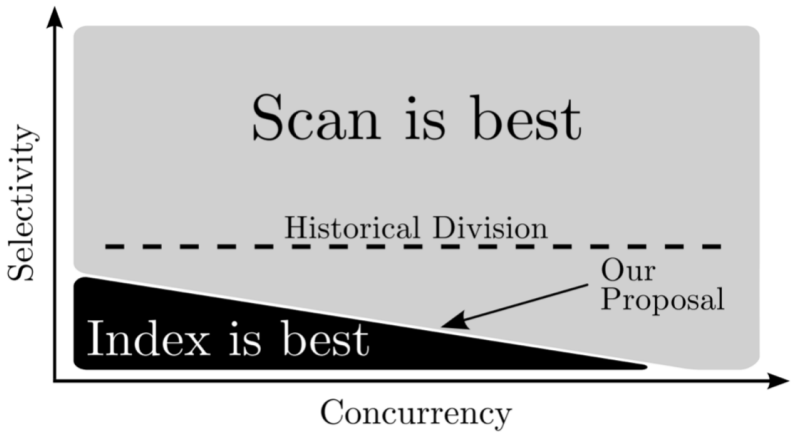


Figure 1.2: Concurrency is important in APS.

dataset could be monitored during the life of the system. The optimizer needs only to perform the computation to select the optimal access path. While, on the one hand, this is a fast computation, on the other, it can only take into account the parameters coded into the algorithm and is susceptible to factors that challenge the base assumptions.

Table 1.2 lists the parameters needed to compute the Access Path Selection (APS) cost proportion. Those parameters illustrate just how sensitive the algorithm is to changes in hardware, data, and the query. For example, when hardware evolves and is capable of running more concurrent queries effectively, the weight we assign to them needs to be adjusted. Fundamental changes to block size, CPU clocks, memory, or some other new capability also require new research to reformulate a new solution by hand.

Spending years reformulating understanding is not a hypothetical scenario. Consider the evolution of data systems. By the late 1990s, research on in-memory data layouts was underway to address the growing speed disparity between CPU and main memory (Abadi et al., 2013). Whereas processor time was the bottleneck in the past, it had now shifted to memory (size, access time, and latency). Table 1.3 shows

Workload	number of queries selectivity of query total selectivity of the workload
Dataset	data size tuple size
Hardware	L1 cache access (sec) LLC miss: memory access (sec) scanning bandwidth (GB/s) result writing bandwidth (GB/s) leaf traversal bandwidth (GB/s) the inverse of CPU frequency Factor accounting for pipelining
Scan and Index	result width (bytes per output tuple) tree fanout attribute width (bytes of the indexed column) offset width (bytes of the index column offset)

Table 1.2: APS algorithm parameters Kester et al. (2017)

Year	1980 (disk)	1990 (disk)	2000 (disk)	2010 (disk)	**2016 (mem)**	**F1 (mem)**	**F2 (mem)**
CPU (GHz)					2	4	4
HDD (ms)	10	8	2	2			
HDD (MB/s)	40	100	500	500			
Mem (ns)					180	100	20
Mem (GB/s)					40	160	80
# tuples	10^6	10^7	10^8	10^9	10^9	10^9	10^9
Tuple Size	200	200	200	4	4	4	4
Branching	250	250	250	250	21	21	21
Crossover	12.4%	6.2%	5.0%	0.1%	0.6%	0.3%	0.5%

Table 1.3: Access path selection crossover point evolution Kester et al. (2017).

how the selectivity-based boundary has been affected over time. In the 1980s, it made sense to switch from scans to indexes when the selectivity fell below 12.4%. However, advances in memory access made the penalty of moving data much lower, and scans became useful in more and more cases. As memory latency continued decreasing, we see indexes again start to make a resurgence in performance in larger datasets (a case that did not exist before fast memory and the massive sets found in the Big Data movement). Inevitably, the state of research in system memory will evolve again and alter the formulation.

Concurrency is complicated. It is not sufficient to say that processing more

queries concurrently increases system performance. Aside from the overhead in allocating threads to queries, concurrent processing has side-effects such as negating other techniques employed by data systems. For example, *zone maps* are an optimization that seeks to minimize the traversal of unnecessary blocks of data by viewing the block as a *zone*. Processing concurrent queries means that there are fewer irrelevant zones to skip since the area of interest is a union of the combined queries.

The hand-derived nature of the algorithm presents a problem to scalability, and raises some interesting questions:

- Can the decision boundary already described be improved? (or, described by a more sophisticated algorithm)?
- Can the performance of the model be improved?
- Are concurrency, hardware, and selectivity the only factors, or are there more dimensions to the problem space?

It is important to note that the *Access Path Selection* algorithm requires analysis and observation. Scientific breakthroughs in computing can also be the result of a data-driven solution (Hey et al., 2009). Can such a data-driven solution be automated?

Choosing the optimal path is critical to the performance of any system, not just in-memory data systems. The goal of every system is to find and return data as quickly as possible. The decisions made along the way determine the viability of the system.

1.1.2 Optimizers

In a data system, the *query execution engine* performs the work of executing operations and returning results. The *query optimizer* provides organized input to

```
SELECT * FROM lineitem WHERE
  l_extendedprice > 75000
  AND l_suppkey < 6000
  AND l_partkey < 1200000;
```

Listing 1.1: Sample query on `LINEITEM` table.

the execution engine. Since the execution engine executes any plan given, the plan must be accurate. The optimizer has three key components (Chaudhuri, 1998):

- A space of query plans called a *search space*.
- Some *cost estimation* technique to determine which plans are more costly than others.
- *Enumeration algorithms* to choose the best plan by exploring the search space.

The optimizer has to (1) ensure that optimal and efficient plans are in the search space, (2) have an accurate cost estimating technique, and (3) use an efficient enumeration algorithm. Each of these tasks is not trivial, and a flaw in any one causes the entire execution to be suboptimal. In practice, query optimizers are often suboptimal.

Search Space

When the optimizer builds a *search space*, it is collecting a list of possible query plans that include permutations of join sequencing, threading scenarios, and *access path*.

Consider the query in Listing 1.1 with an index on each of the columns in the predicate (`l_suppkey`, `l_partkey`, and `l_extendedprice`). The plans in the space include many variations, a few of which are shown in Listing 1.2 (PostgreSQL). The plan in Listing 1.2a performs a bitmap scan on `l_sk`, then filters (scans) `l_extendedprice`

```
Gather  (cost=68999.01..1326984.50 rows=161741 width=129)
  Workers Planned: 2
  -> Parallel Bitmap Heap Scan on lineitem (cost=67999.01..1309810.40 rows=67392)
       Recheck Cond: (l_suppkey < 6000)
       Filter :  ((l_extendedprice > '75000') AND (l_partkey < 1200000))
       -> Bitmap Index Scan on l_sk  (cost=0.00..67958.57 rows=3679734)
            Index Cond: (l_suppkey < 6000)
```

(a) Bitmap index on `l_suppkey` column, then filter.

```
Gather  (cost=1000.00..1443085.60 rows=764987)
  Workers Planned: 2
  -> Parallel Seq Scan on lineitem (cost=0.00..1365586.90 rows=318745)
       Filter :  ((l_extendedprice > '75000') AND (l_suppkey < 6000) AND (l_partkey < 1200000))
```

(b) Multithreaded table scan filtering on all columns.

```
Index Scan using l_sk  on lineitem  (cost=0.56..9681195.21 rows=161741)
 Index Cond: (l_suppkey < 6000)
 Filter :  ((l_extendedprice > '75000') AND (l_partkey < 1200000))
```

(c) Index scan on `l_suppkey` column, then filter.

Listing 1.2: Sample query plans for Listing 1.1.

and `l_partkey`. Another (Listing 1.2b) uses multiple workers to scan the entire table, looking for all columns in the predicate at once. Finally, we see a plan (Listing 1.2c) that performs a btree index probe in `l_sk`, then scans the remaining columns.

While conceptually simple, there is no perfect method for representing plans in a complex search space. Some systems use a calculus-oriented approach to analyze the structure of the executions in the plan while others use a query graph with nodes and edges representing operations (Chaudhuri, 1998).

If the best plan is not in the search space, the downstream portions of the optimizer must select the best of the available options (perhaps the worst options). If the plans in Listing 1.2 did not include 1.2b, the most effective plan in some cases would be missing (when returning more than .0099% of the rows).

Cost Estimate

There are many ways to represent a query as a set of operations. *Cost estimation* is the method of determining how costly components of the plan are in terms of memory, time, CPU, and I/O. This process must also be done efficiently since the

optimizer will repeatedly invoked it. The estimation framework must:

(a) Collect statistical summaries of the data stored in the system

(b) For each data stream:

    (a) determine the statistical summary of the output data stream

    (b) estimate the cost of executing the operation

Statistics collected while the system is idle accomplish the first task (in the meantime, they may drift from the actual). The latter task uses the collected statistics to predict the output distribution for estimating a cost to the various operations. If the estimates are inaccurate, the system executes a slower plan believing it to be optimal.

All plans in the space use the same statistical data and therefore should have the same output summaries (they all return the same data). However, that is not always the case. Listing 1.2b shows a very different estimation for rows than the other two plans in the space. In reality, none of them are correct. The actual number of rows that are returned by the query in Listing 1.1 are 153,849. All of the plans overestimate the number of rows returned.

Where plans significantly differ is the cost assigned to operational variations between plans. These costs are in an arbitrary unit of computation (although traditionally it is some representation in terms of disk page fetches set relative to the value of `seq_page_cost` parameter). The only reliable method to interpret them is to compare them relative to each other. From least to most expensive, the optimizer projects Plan 1.2a to cost 1326984.5 units, Plan 1.2b to cost 1443085.6 units, and Plan 1.2c to cost 9681195.21 units. Note that individual functions are assigned costs. For example, the scan on `l_sk` (line 6, Listing 1.2a) costs 67958.57 units, and the

cost of `GATHER` (combining the two worker threads) is 62825.9. Each cost in the tree includes the costs below them, so we need to subtract the cost of line 3 from line 1 to determine line 1's cost.

The cost only reflects things that the planner finds essential. Time spent transmitting result rows to the client or performing other factors that contribute to the real elapsed time of the query are ignored because the optimizer cannot change them by altering the plan.

Cost estimation remains one of the most challenging parts of the optimizer. It requires intimate knowledge of the underlying system's physical configuration and operator algorithms (for example, buffer pool hit ratios, memory access costs, cache latency, and co-location of data) (Chaudhuri, 1998).

A learning optimizer could observe the reality of execution, allowing all of the details to resolve naturally. If a genuinely accurate cost is predicted overall, the cost of the minutia does not matter. An optimizer executes the fasted plan, observes change, and adjusts.

It is vital to point out that the optimizer is making a few assumptions. First, each plan's total cost is the sum of its parts. That *could* be true if the environment were sterile and we were confident that no factors other than what is in the plan come into play. Second, the lowest cost is derivable from the statistical information *on hand*. What about concurrency (Kester et al., 2017) or some new algorithmic improvement? We cannot improve the algorithms automatically by adding some new hardware components. The optimizer needs to become intimately aware of its cost. A learned optimizer system could improve along with the underlying hardware.

Enumeration algorithm

The *enumeration algorithm* selects an inexpensive plan to execute. If it were simply a matter of executing every plan in the search space to determine which

enable_bitmapscan	seq_page_cost	geqo
enable_hashagg	random_page_cost	geqo_threshold
enable_hashjoin	cpu_tuple_cost	geqo_effort
enable_indexscan	cpu_index_cost	geqo_pool_size
enable_indexonlyscan	cpu_operator_cost	geqo_generations
enable_material	parallel_setup_cost	geqo_selection_bias
enable_mergejoin	parallel_tuple_cost	geqo_seed
enable_nestloop	min_parallel_relation_size	cursor_tuple_fraction
enable_seqscan	effective_cache_size	from_collapse_limit
enable_sort	default_statistics_target	join_collapse_limit
enable_tidscan	constraint_exclusion	force_parallel_mode
enable_gathermerge		

Table 1.4: Optimizer settings in PostgreSQL version 10.

is fastest, this would be trivial. Many modern versions are somewhat extensible, allowing the addition of new operators to the search space. The optimizer uses a rule-based algorithm to determine which plan to choose, suffering from the same issues plaguing any pre-coded approach. Systems expose "knobs," allowing experts to influence the optimizer. That poses a problem because the expertise needed to tune the knobs is very high and very time-consuming.

While knobs are helpful, the only influence exerted is limited to which knobs are exposed. Table 1.4 shows optimizer settings available in PostgreSQL 10. Consider PostgreSQL's genetic query optimizer, which uses a genetic algorithm to search the plan space over a rule-based approach (the name of the knobs begin with `geqo`). An expert can set values like the size of the genetic population and how many generations until convergence, but you can not tell it to use a different fitness function, mutation rate, or even a different algorithm entirely. You can take the inadvisable approach of turning it off completely, which illustrates the difficulty of expertly tuning systems. Finding the best genetic settings requires an exhaustive process of exploration. The documentation for PostgreSQL recommends not to touch it, negating any benefit an exposed knob would offer.

Since the row estimates are incorrect, the cost estimates that use them as a basis are also inaccurate. If the enumeration algorithm relies strictly on taking the lowest cost, it chooses the least optimal plan, in reality, Plan 1.2a instead of the better Plan 1.2b.

How Good are Query Optimizers?

Queries run on real systems with other things happening in the background and other queries running that permute the state of the system. At times, queries operate on the same data and share execution. A learned algorithm could better understand the observed reality of what happens when a queries run concurrently.

Finally, some parts of a system extremely challenging to estimate. For example, user-defined functions are opaque to the optimizer. A learned optimizer could effectively treat them as a black box and train on the observed behavior.

As I have shown, achieving perfection is difficult, while failure is easy. Given all of this, how does the optimizer truly perform? Consider the query in Listing 1.3. As before, this query could be executed in many ways. Figure 1.3 compares the performance of plans selected by the optimizer with the actual optimal plan as `l_suppkey` is varied (from 200 to 4,000) to increase selectivity (from 0.1988% to 3.9972%). The optimizer selects a bitmap scan on a single thread until point A (2.7985% selectivity), where it switches to a multithreaded bitmap scan. We can see that it should have switched much sooner. However, the true optimal was a different plan altogether; starting with a btree probe on `l_sk` until point B (0.7983% selectivity), then switching to a multithreaded table scan. Not only should it have switched to multithreading sooner, it should have never chosen a bitmap scan in the first place, let alone stick with it for eternity.

While this is a simple example, Chapter 2 demonstrates that an optimizer choosing the truly optimal plan is a rare occurrence.

```
SELECT * FROM lineitem WHERE l_suppkey < 200;
```

Listing 1.3: Query 3.

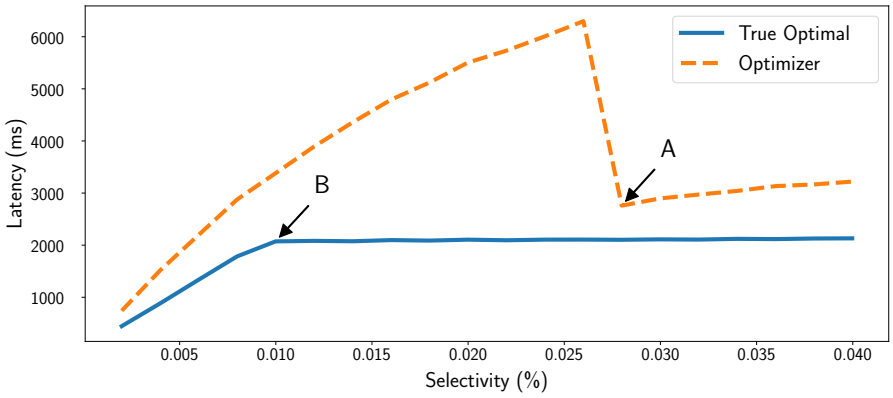


Figure 1.3: Ideal vs optimizer performance of Query 3.

1.1.3 Neural Networks

Neural networks have seen success in predicting complex problems. Rosenblatt first identified the *perceptron* as a probabilistic model in his 1958 paper (Rosenblatt, 1958), but it was not until more modern techniques improved the perceptron, making it a viable machine learning technique. Figure 1.4 illustrates the basic principles of the perceptron. Each input has a corresponding weight and bias applied (a linear regression). An activation function controls the perceptron's ability to "fire".

A biological neuron is an excellent metaphor because a neuron is either on or off. A network of many such neurons fires in observable patterns, and we can draw a decision boundary around the patterns.

The technique used to cause a neuron to fire or not is called the *activation function*. There are a variety of activation functions with varying properties as can be seen in Figure 1.5. The activation function defines the final output given an input.

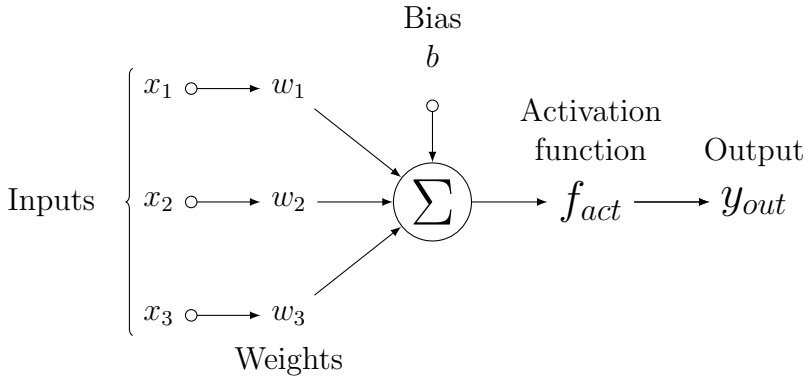
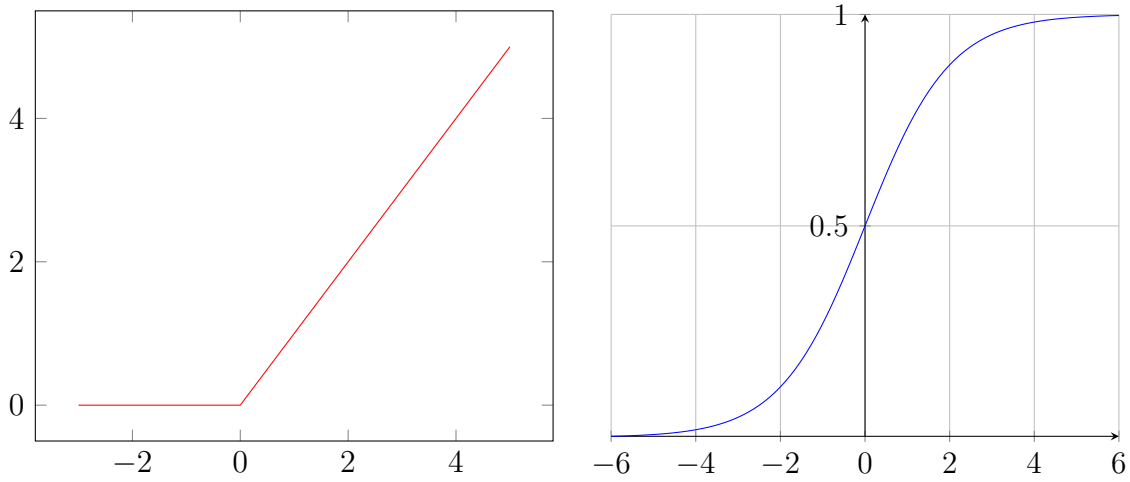


Figure 1.4: Rosenblatt's Perceptron.



(a) The ReLU activation function.

(b) Sigmoid activation function.

Figure 1.5: Common neural network activation functions.

Consider the functions in Figure 1.5. A *rectified linear unit* (ReLU) is the most popular neural network activation function (Ramachandran et al., 2017). It retains only the positive part of the input value (Figure 1.5a), effectively removing noisy negative neurons, allowing better gradient propagation to the next layer. A *sigmoid* (Figure 1.5b) effectively pushes values either toward one or zero (especially useful in classification problems, but problematic for values in the middle).

Many perceptrons could be clustered and arranged as layers. It is not uncommon to have thousands or millions of neurons in a network. A network of many layers

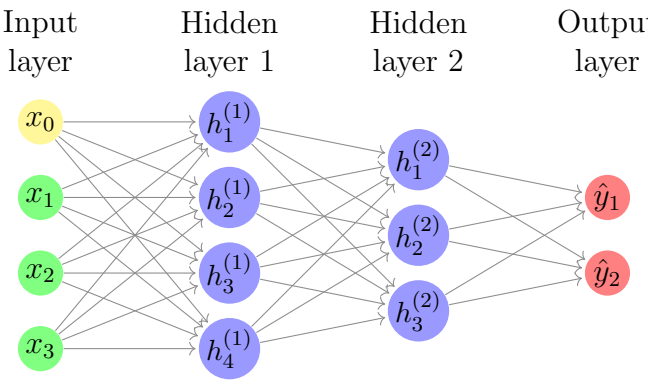


Figure 1.6: A multi-layered perceptron network (MLP).

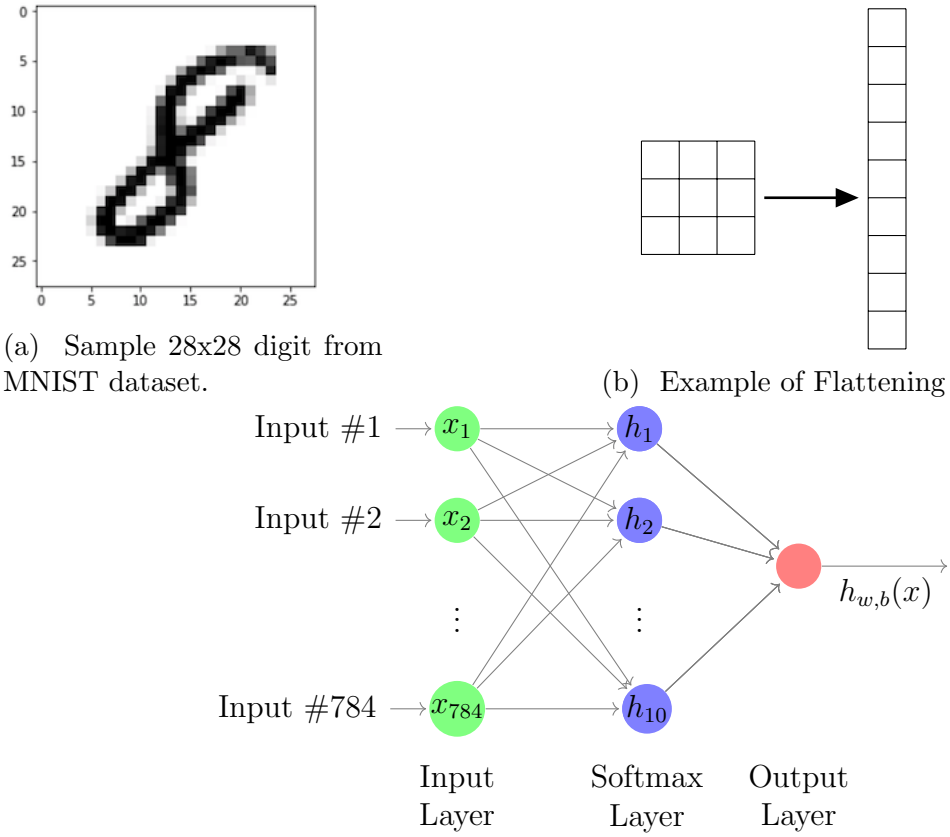
of perceptrons is appropriately called a *multilayer perceptron* network (MLP).

Figure 1.6 is an example of one such network used to classify handwritten digits. The MNIST dataset contains 70,000 handwritten digits in 28x28 pixel format along with a corresponding labeled classification (0-9) (see Figure 1.7a). This type of approach is known as *supervised learning* because we know what the actual prediction should be by comparing it to the label.

To feed the images to a neural network, we first *flatten* the image by converting it from a 28x28 pixel (784 total pixels) matrix to a 784-pixel vector (Figure 1.7b). We assign each pixel to a neuron in the input layer (a layer of 784 neurons) and connect each of those to every neuron in the next layer (Figure 1.7c). Such an architecture is appropriately called *fully connected*. The next layer of ten neurons (one for each classification of digits) uses Softmax (a multi-classification machine learning technique) to create probabilities for each class. Finally, in our output layer, we will select the highest probability as the predicted output.

We do not know what the values for the weights for each parameter should be. Finding an optimal combination for millions of weights on 70,000 images is a significant computing problem.

Backpropagation is a technique that allows the weights associated with the



(a) Sample 28x28 digit from MNIST dataset.

(b) Example of Flattening

(c) Simple network for classifying digits.

Figure 1.7: MNIST image classification.

neurons to be learned. The results of backpropagation are used by another function (such as stochastic gradient descent) to perform the learning. The backpropagation algorithm works as follows:

(a) Compute the scalar cost (or loss) during the forward pass through the network. For example, in a sum of squares approach.

(b) Compute the gradient of the cost function with respect to the parameters (using the chain rule of calculus).

1.2. Prior Work

Nowadays, there is a never-ending deluge of success stories in artificial intelligence. The term itself has not coalesced into a specific body of knowledge. In this thesis, I consider AI to encompass neural networks, reinforcement learning, evolutionary algorithms, and any machine learning technique that learns patterns in data. Every researcher wonders at some point if it is practical to create an artificial (learned) optimizer. The conclusion is natural since the knowledge needed to train such an algorithm exists in the data system somewhere.

The practicality of an artificial optimizer is difficult to determine. Practical in which terms? Is it technically possible, maintainable, and offer a fast enough response to incoming queries? How difficult will it be to train, and what is the performance impact? On one side of the debate, machine learning is deemed unnecessary to improve the quality of optimizers as they are only as good as the estimates given (Leis et al., 2015; Wu et al., 2013). On the other, recent research shows promising results (Kipf et al., 2018; Kraska et al., 2019; Krishnan et al., 2018; Marcus et al., 2019; Marcus & Papaemmanouil, 2018; Ortiz et al., 2018).

The problems with optimizers are well documented. In addition to sensitivity to cost estimates, system performance is highly dependent on configuration (Van Aken et al., 2017). Additionally, optimizers are highly susceptible to hardware changes (e.g., scans look more and more attractive as hardware improves). That presents the untenable situation of a research bottleneck. Research can take years, during which time hardware changes again, starting the cycle anew. Exciting research shows that the entire approach is even suspect when hardware changes because the algorithm has to account for concurrency where it did not before (Kester et al., 2017).

Even after decades of research, the generalization of an optimizer's cost pro-

cess is still very difficult. Conventional research methods are still the means that improvements make their way to optimizers, far behind state of the art.

However, *problems with machine learning approaches also exist.* They are slow to train, consume massive resources, and require large sets of data. Trained models are not portable and applicable to different scenarios (such as varying hardware, systems, algorithms) and require expensive retraining. Although research has been done on models of various sizes and architectures, establishing a best practice is still in the distance. Neural Networks and Reinforcement Learning are also coming of age, but the research for their applications to optimizers is still relatively new (Marcus & Papaemmanouil, 2018; Ganapathi et al., 2009; Zaheer et al., 2017). Despite the problems, the results are promising enough to warrant further exploration.

Opportunities

First, AI-based research in data systems is scattered, with most of the research centering around techniques to improve existing optimizers. Understandable, because they are very well researched and understood areas of the optimizer. They are also reasonably sized work that can establish viability for machine learning. Still, there is a significant gap in other areas of the optimizer or even the optimizer as a whole:

- Estimating cardinality for building sub plans (Ortiz et al., 2018)
- Join order enumeration (Kipf et al., 2018; Krishnan et al., 2018)
- Workload scheduling (Ganapathi et al., 2009)
- Data structures by means of new a kind of learned index (Kraska et al., 2018)
- Evolutionary approaches to access path selection and vertical fragmentation (Song & Gorla, 2000)

Second, even within machine learning and AI, one can approach the problem from several directions. One option is to leave the optimizer intact, using predictive models to improve estimates the optimizer relies on (Akdere et al., 2012). However, some researchers caution that better estimations have yet to demonstrate better plans, casting the entire line of reasoning into doubt (Marcus et al., 2019). Keeping the existing optimizer also inherits all of the problems in the current design. As some authors (Van Aken et al., 2017) point out, a system's performance is dependent on its configuration. The best you can do is tune the available knobs and hope for the best. Any other tuning is effectively unreachable. Additionally, different databases have different knobs.

Another option is to replace the optimizer entirely (Marcus et al., 2019). Research exists on micro components of the optimizer (for example, ReJOIN), but it is much more challenging (and intimidating) at a holistic level. However, by freeing us from the underlying system-specific algorithms, such an approach could translate more freely to various systems (perhaps even as a drop-in replacement for optimizers).

Even though problems with AI exist, all is not lost. Interesting research (Marcus & Papaemmanouil, 2018) shows that even a tiny model of only a few neurons can have a meaningful impact on join order enumeration using a Reinforcement Learning approach.

We have many open questions. Should we use a monolithic model for the entire system, or consider smaller, more purposeful models (Marcus et al., 2019)? How do we build such a system? Does the monitoring and training happen in-line with the system's other functions, or separately as another, even external, system (Gupta et al., 2008)?

1.3. Contributions

The main contribution of this thesis is an approach to access path selection using artificial neural networks that outperform a traditional approach to optimization. Secondarily, traditional loss functions for training neural networks suffer from the way they show accuracy without additional considerations.

As we have seen, approaches such as ReJOIN (Marcus & Papaemmanouil, 2018), use Reinforcement Learning models to predict optimal relation join order accurately. Such an approach can not be applied directly to the problem of access path selection because the major components of the approach (feature vectorization and policy gradient) are highly specific to the join problem. Further, the join order problem lends itself to an iterative approach because relations can be incrementally added and rewarded at each step.

I show that small neural networks of multi-layer perceptrons outperform the built-in query optimizer while avoiding the problems inherent in other solutions.

Problem: Reliance on the Optimizer

Other efforts, such as LEO (Stillger et al., 2001; Markl & Lohman, 2002) attempt to address the cost estimation problem by predicting cost estimates using machine learning techniques. While those rely on the existing optimizer, I show that a neural network can produce very accurate results without considering traditional optimizer costs.

Problem: Reliance on Computed Statistics

Improving the accuracy of table statistics is another method of addressing the cost estimate problem. We have seen that table statistics are not enough for access path selection (Kester et al., 2017). Research, like PQR (Gupta et al., 2008), predicts ranges of execution times using selectivity, among other things. In this thesis, I show

that a neural network can predict execution latency without using selectivity.

Problem: Computationally Expensive Predictions

Other research uses memory-intensive models, such as nearest-neighbor machine learning models (Ganapathi et al., 2009). These approaches require the entire training set to remain at hand during *prediction*, rather than *training* time. The resulting higher latency predictions make the approach unacceptable for access path selection. By their nature, neural networks require fewer resources at prediction time.

1.3.1 Summary of Contributions

- Demonstrate the performance of optimizers against the true ideal decision path on TPC-H data.
- Tune the optimizer to understand the difference between out-of-the-box settings and a highly tuned system.
- Compare various architectures and their effect on small and large neural networks.
- Show that identifying training loss is nuanced and identify lessons learned.
- Unlike LEO (Stillger et al., 2001; Markl & Lohman, 2002), which relies on the existing optimizer by predicting its cost estimates, I show that a neural network can produce accurate results without using selectivity and optimizer cost.
- Unlike PQR (Gupta et al., 2008), which predicts ranges of execution times, I show that an accurate latency prediction without selectivity is possible.
- Unlike research that focuses on resource-intensive models like nearest neighbor (Akdere et al., 2012; Ganapathi et al., 2009), I show system state, hardware,

query metadata, and observed latency train a very accurate model. I further show that a correct access path eliminates the need to estimate the minutia of a plan.

1.4. Outline

In Chapter 2, I describe the test methods and structure used and discuss how to find the true ideal and measure the optimizer's performance.

In Chapter 3, I describe the neural network architectures, and the process and various methods of extracting usable feature vectors from our performance data.

Chapter 4 is the main body of findings, where I measure the results of the monolithic and smaller, purpose-built neural networks. The chapter concludes with a comparison of the two approaches in terms of predictive ability and training time.

In Chapter 5, we briefly review some interesting nuances discovered between the way neural networks are conventionally trained and the peculiarities of access path selection.

In Chapter 6, I describe a possible approach to operationalizing this work.

Finally, in Chapter 7, I summarize conclusions and describe exciting future directions.

Chapter 2: Approach

2.1. Introduction

Generating a plan with the lowest latency is the ultimate goal of our model (and all optimizers). That makes latency a reasonable benchmark for comparison; if we can show that another approach can perform better than the optimizer, it is worth exploring. Of course, that is not the only benchmark. Other questions are worth asking:

- How long does it take to get to that conclusion?
- How complicated is it?
- How many resources does it consume to get there?

While these questions are of interest and explored somewhat, the primary objective of this thesis is to reduce latency by reach more accurate access path decisions. To that end, we need to measure optimizer performance as a baseline for comparison of our model. Then, with the optimizer disabled, we measure various plans through a range of selectivity with a *grid search* to establish a source of truth and discover the true *crossover point*. Recall that a crossover point is a point in which the decision for the access path should change from one plan to another. A grid search is a technique involving an exhaustive search of all possible combinations of parameters. Grid

searches are expensive, especially considering this test is on hundreds of thousands of queries.

We compute the ideal access path by locating the truly fastest plans by measured output. Finally, we compare the model's predictive power by determining how accurately they predict the truly optimal *paths* (not just the most accurate latency predictions). To locate the crossover point, we vary the selectivity of the queries by small increments. We also need to understand how multiple predicates interact and affect performance. The information contained in the query is not enough. We need to consider:

- Hardware characteristics (how much memory? how many cores? etc.)
- System configuration (what indexes do we have to work with)
- The method we should access (scan? btree? use multithreading?) and in which order
- How does multithreading affect the decision boundary?

To address these, we run the entire set of queries on a variety of hardware configurations and observe the variances in decision boundaries. The results form the basis of our training set. Rather than allow that to happen organically, we start with a curated suite of queries that exercise the system in the desired selectivity range between .02% to 4% (where the access path selection crossover typically occurs is usually between 1.5% and 2%, but gets lower as hardware improves).

To test the optimizer as a baseline, we perform two tests. First, we measure an untuned optimizer (Test 0) and its corresponding decisions and second, perform a grid search on the optimizer knobs to compare a highly tuned optimizer (Test 2). Appendix A lists all of queries for the various tables.

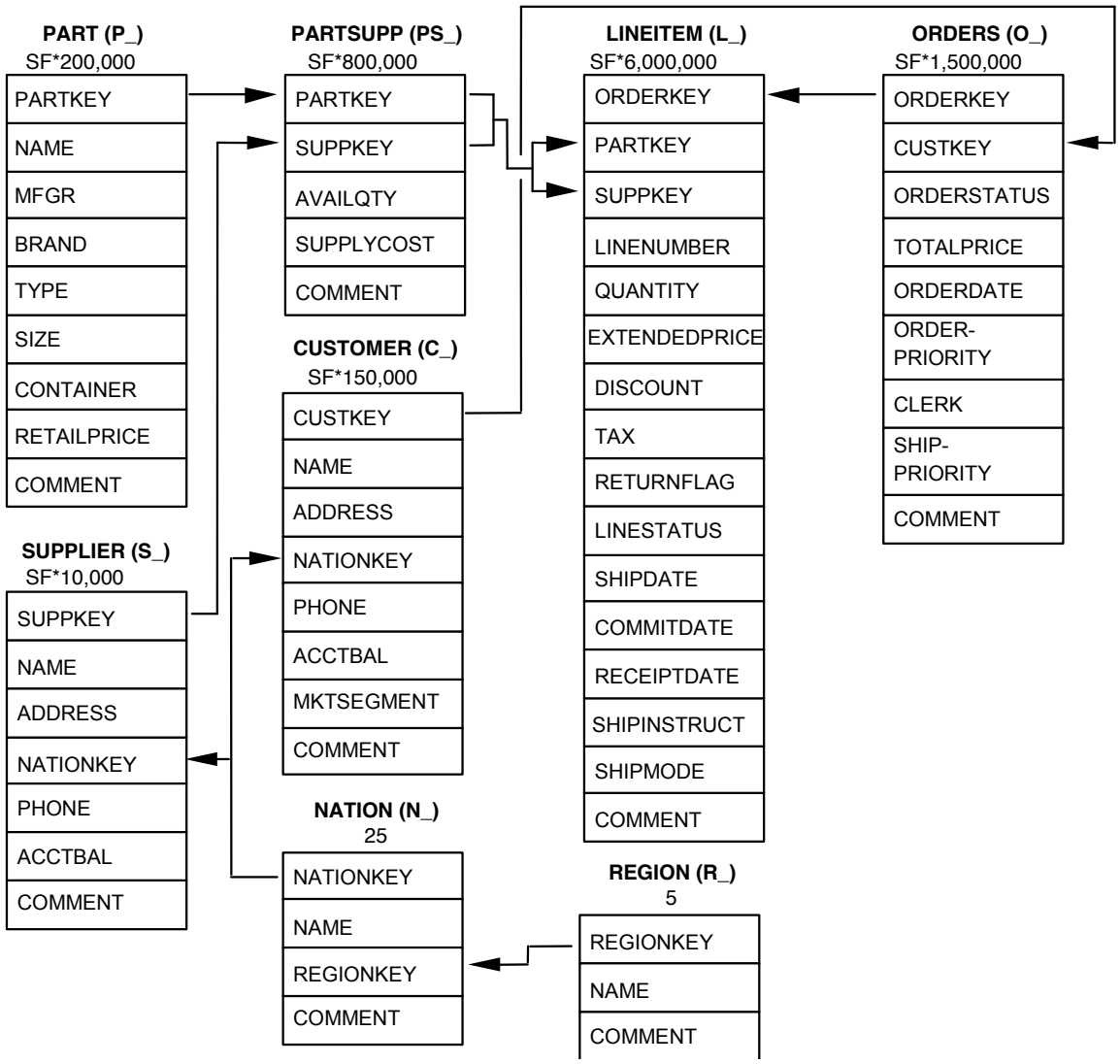


Figure 2.1: TPC-H table schema.

2.2. The TPC-H Benchmark

The TPC organization defines various benchmark standards. The TPC-H benchmark is a standard decision support benchmark and includes a set of tables and queries as well as tools to generate data to populate the tables. The widely used benchmark compares the performance of SQL-like data systems.

The benchmark includes tooling (`DBGEN`) to load the table with generated data,

at a scale factor selected. In the TPC-H schema, represented in Figure 2.1, below each table name is a "scale factor". For example, the `LINEITEM` table has a scale factor of 6,000,000, meaning that for each scale factor, 6,000,000 rows are generated. For the test setup, I used a scale factor of 10 on the four tables tested. For example, `LINEITEM` has 60 million rows ($6,000,000 \times 10$). Three large tables (`LINEITEM`, `PARTSUPP`, and `ORDERS`) and one smaller table (`PART`) were tested.

To get a space of query results in the target selectivity ranges, I derive the queries run from one of 100 base queries, applying variations to each of the query's predicate so that we achieve the desired granularity. Each query executes with only scan enabled, with bitmap, and with every combination of btree indexes. Each of those, in turn, runs with and without parallelism. Appendix C lists these combinations (**batches**).

2.2.1 Hardware

Tests executed on a variety of hardware configurations are reminiscent of tests done on APS in other papers (Kester et al., 2017). I use profiles that are **memory optimized**, **compute optimized**, and **general purpose**. Table B.1 (Appendix B) lists the three configurations and various technical features of each. Hardware setup must be carefully selected to ensure proper sizing. If it is too large, it obscures the differences between systems (because it executes the queries too quickly, through brute force). Some features are the same on all systems, but and are included for convenience (or future usefulness). Scalar features are normalized and treated as continuous values (such as processor clock speed and amount of RAM), while other features are categorical (such as the processor architecture and whether it supports SIMD). For easy reference, the shaded rows in Figure B.1 represent continuous values, while the rest are categorical.

Table	Column	Distribution
LINEITEM	L_PARTKEY	uniform
LINEITEM	L_PARTKEY	uniform
LINEITEM	L_PARTKEY	uniform
LINEITEM	L_PARTKEY	uniform
PARTSUPP	PS_AVAILQTY	uniform
PARTSUPP	PS_PARTKEY	uniform
PARTSUPP	PS_SUPPKEY	uniform
PARTSUPP	PS_SUPPLYCOST	uniform
ORDERS	L_CUSTKEY	uniform
ORDERS	L_ORDERKEY	uniform
ORDERS	L_TOTALPRICE	skewed right
PART	L_PARTKEY	uniform
PART	L_RETAILPRICE	symmetric
PART	L_SIZE	uniform

Table 2.1: TPC-H columns tested.

2.3. Test Queries

There are 100 **query templates** that form the basis of the test framework presented in this thesis. These execute against the larger TPC-H tables `LINEITEM`, `PARTSUPP`, and `ORDERS`; and the smaller `PART` table. Table 2.1 lists the columns, corresponding tables, and distributions of data therein. These columns were selected because they contain a large enough range of values for training. From those 100 queries, we derive 2,511 variations in granularity (collectively referred to as, **queries**) by incrementally adjusting the predicate values. Each variation is given a **run** number.

Table 2.2 demonstrates the run variations for query template 35 on the `PARTSUPP` table. The `PS_PARTKEY` column is stepped in 20 increments to give us a selectivity between approximately 0.02% and 4%. A query is identified by its query id (35) and the run number (1-20). For example, query 35-10 is the variation that yields a selectivity of 1.9995%.

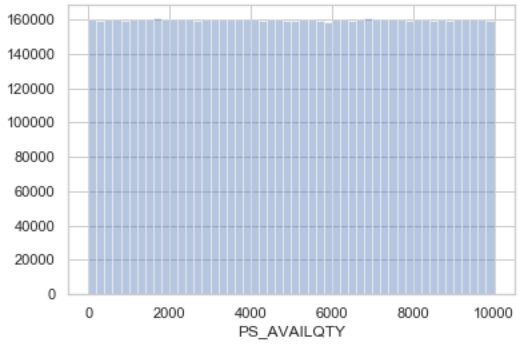
`SELECT * FROM partsupp WHERE PS_PARTKEY < ?`

Run	PS_PARTKEY	Selectivity (%)
1	4000	0.1999
2	7999	0.3999
3	11998	0.5999
4	15997	0.7998
5	19996	0.9997
6	23995	1.1997
7	27994	1.3997
8	31993	1.5996
9	35992	1.7996
10	39991	1.9995
11	43990	2.1995
12	47989	2.3994
13	51988	2.5993
14	55987	2.7993
15	59986	2.9993
16	63985	3.1992
17	67984	3.3992
18	71983	3.5991
19	75982	3.7990
20	79981	3.9990

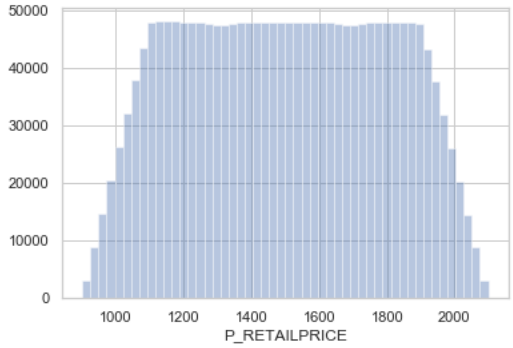
Table 2.2: Query 35 variations on `PARTSUPP`.

These incremental queries run against variations in system state called **batches**. Each batch represents a unique combination of access path (scan, index, or bitmap), multithreading, and available indexes. Appendix C lists the unique characteristics comprising the 673 unique batches.

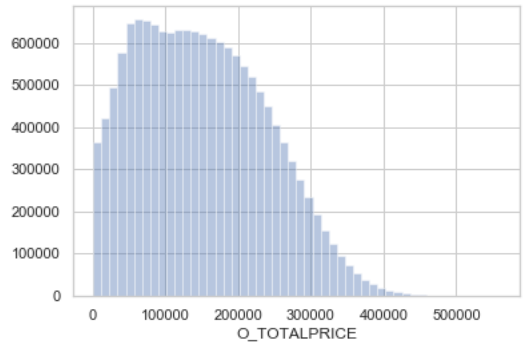
Each of the batches, in turn, run on three Amazon Web Services EC2 instances representing the three profiles: compute-optimized (`c5d.4xlarge`), memory-optimized (`r5d.4xlarge`), and general-purpose (`m5d.2xlarge`). All runs, batches, queries, servers, and permutations total 267,633 individual executions are comprising the training set. An execution is uniquely identified by the server, query, batch, and run.



(a) `PARTSUPP.PS_AVAILQTY` (uniform).



(b) `PART.P_RETAILPRICE` (symmetric).



(c) `ORDERS.O_TOTALPRICE` (skewed right).

Figure 2.2: Distributions of TPC-H data.

2.3.1 Data Distributions

TPC-H tools provide a mechanism for generating data by sampling from pre-set distributions for each column. That approach is particularly useful in determining a predictable, variable range of selectivity. For example, consider the distributions of data for the column `PARTSUPP.PS_AVAILQTY` in Figure 2.2. That column's data are uniformly distributed between zero and 10,000 (approximately 160,000 of each value, given a scale factor of 10). We can step the predicate values in equal increments to get the desired granularity (e.g., 4,000 keys produce 0.02% selectivity). Since the distribution is uniform, adding another 4,000 keys adds another 0.02% selectivity, and so on.

While PART.P_RETAILPRICE is not uniform, it is symmetrical and predictable. ORDERS.O_TOTALPRICE has a distribution that is skewed right, making it especially valuable for learning (demonstrating that the model can predict a variety of distributions).

2.3.2 Column Interaction (Order Enumeration)

Estimating selectivity is foundational to access path selection algorithms and, therefore, performance. We have seen that hardware also affects performance. In the simple case of a query with a single predicate, computing selectivity is trivial. However, in the case of a query with multiple predicate columns, the approach becomes a bit more challenging. The problem becomes even more complicated when we consider that some or all the columns may be indexed, and that system state mutates over time. The decision boundary for query plans becomes affected by the combination of when (order) and how (access path) each predicate is applied. Each of these permutations must be tested to understand their interaction.

For example, Table 2.3 details the variations of query template 100, which tests the interactions of columns O_TOTALPRICE, O_CUSTKEY, and O_ORDERKEY on table ORDERS. The predicate fixes O_TOTALPRICE while it varies the other two columns, measuring the latency effect and access path selected. The ideal plan across the entire selectivity range is:

$$\sigma_{TOTALPRICE} \rightarrow \sigma_{CUSTKEY} \rightarrow \gamma_{ORDERKEY}$$

The notation above describes a plan in which it first scans (σ) the entire table's O_TOTALPRICE column, then on the interim results scans O_CUSTKEY, and finally performs a btree probe (γ) on the remaining O_ORDERKEY. We yield the best results

```
SELECT * FROM ORDERS WHERE O_TOTALPRICE < ?  AND
O_CUSTKEY < ?  AND O_ORDERKEY < ?;
```

Run	O_TOTALPRICE	O_CUSTKEY	O_ORDERKEY	Selectivity (%)
1	120766	24750	990000	0.0107
2	120766	49499	1979999	0.0435
3	120766	74248	2969998	0.0993
4	120766	98997	3959997	0.1774
5	120766	123746	4949996	0.2798
6	120766	148495	5939995	0.4037
7	120766	173244	6929994	0.5508
8	120766	197993	7919993	0.7194
9	120766	222742	8909992	0.9097
10	120766	247491	9899991	1.1226
11	120766	272240	10889990	1.3578
12	120766	296989	11879989	1.6148
13	120766	321738	12869988	1.8976
14	120766	346487	13859987	2.2013
15	120766	371236	14849986	2.5243
16	120766	395985	15839985	2.8703
17	120766	420734	16829984	3.2401
18	120766	445483	17819983	3.6330
19	120766	470232	18809982	4.0482
20	120766	494981	19799981	4.4852

Table 2.3: Query 100 variations on `ORDERS`.

by pushing the btree probe as far down the stack as possible (minimizing the penalty for a cache miss when traversing the btree). The `O_TOTALPRICE` scan portion of the query includes roughly 25% of records. It pays to take the scan penalty since we are selecting so many records on that pass (following the intuition we have been building all along). But, why scan `O_TOTALPRICE` before `O_CUSTKEY`? They have the same proportional steps in the query predicate (thus, selectivity) and the columns have the same distribution. While the difference between the approaches may have been computable given perfectly accurate statistics and algorithms, there is no substitute for running the query and observing the latency (as we have done here). Optimizers cannot afford to do that at run time.

Switch	Effect
`SET force_parallel_mode = OFF;` `SET parallel_setup_cost = 1000000;`	disable parallelism
`SET enable_bitmapscan = OFF;`	disable bitmap
`SET enable_seqscan = OFF;`	disable scan
`SET enable_indexscan = OFF;`	disableindex

Table 2.4: Disabling plans in PostgreSQL optimizers (return to default to enable).

2.3.3 Test Execution

All queries were run on each of the servers directly with *psql* (the console application packaged with PostgreSQL). Queries are assembled into scripts that also created applicable indexes according to the batch requirements. The optimizer's path was forced a certain direction by enabling only certain options using switches such as those in Table 2.4, and by adding and deleting indexes. For example, Table 2.5 is a matrix representing options for executing Query 100. Each row represents a batch. The circle indicates that an option was enabled (or an index created). A scan is denoted by the absence of a mark in the index or bitmap column. We run the entire set represented in this table twice; once with parallelism enabled, and once without.

2.4. The Optimizer

As discussed, each of the 2,511 unique queries run in batches varying state on three servers. Computing the **true optimal** (or ideal) is a matter of selecting the lowest latency result of each query across batches (per server), building an **ideal optimizer** that made perfect choices. We quantify the performance of the actual optimizer by running the queries against a freshly installed system with indexes on every column so that it has the opportunity to select freely. Then, we compare the optimizer's results to that of the true optimal.

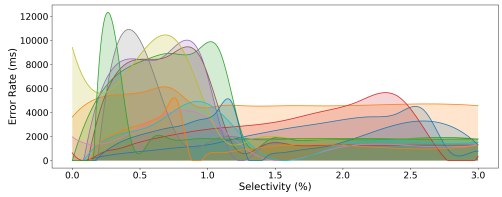
Index on				
O_TOTALPRICE	O_CUSTKEY	O_ORDERKEY	Index scan	Bitmap scan
			○	
				○
	○			
	○		○	
	○			○
		○		
		○	○	
		○		○
○	○			
○	○		○	
○	○			○
○		○		
○		○	○	
○		○		○
	○	○		
	○	○	○	
	○	○		○
○	○	○		
○	○	○	○	
○	○	○		○

Table 2.5: Access path options for Query 100.

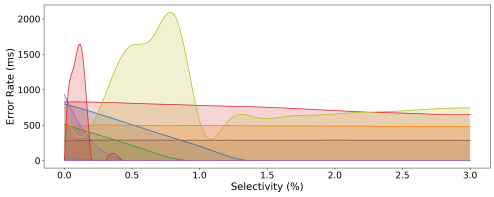
2.4.1 How Optimal is the Optimizer?

The results of query execution are shown in Figure 2.3. The difference between the latency of the ideal path and the path the optimizer chose is the **error rate**. Figures 2.3a and 2.3b show how queries performed on their respective tables. Figures 2.3c, 2.3d, and 2.3e illustrate the average error rate of those queries.

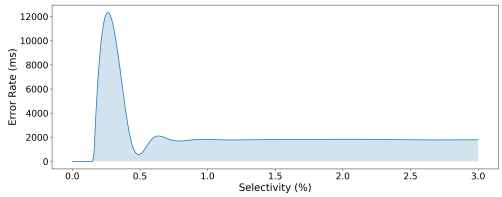
The error rate is particularly high in the low selectivity range (below 1.5%). That range is critical for access path selection because that is typically where the crossover from one plan to another occurs. Decision accuracy in this portion of the selectivity range is a critical factor in determining the viability of an AI approach. The penalty for such a decision are long-lasting. Not only is there a penalty of over 10 seconds, on average at the time of a decision (Figure 2.3c), but sticking with a poor decision continues to cost 2 seconds throughout the selectivity range.



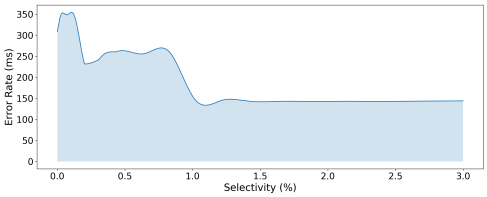
(a) Queries 1-16 `LINEITEM` table.



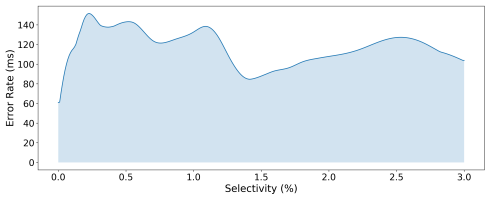
(b) Queries 86-100 `ORDERS` table.



(c) Average error rate on `LINEITEM` table.

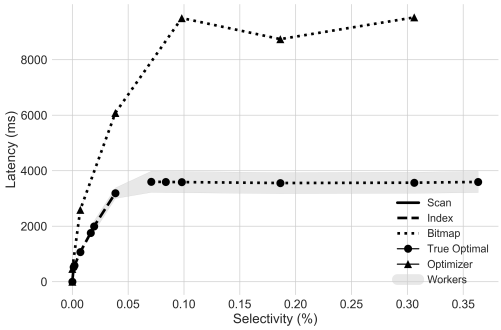


(d) Average error rate on `ORDERS` table.

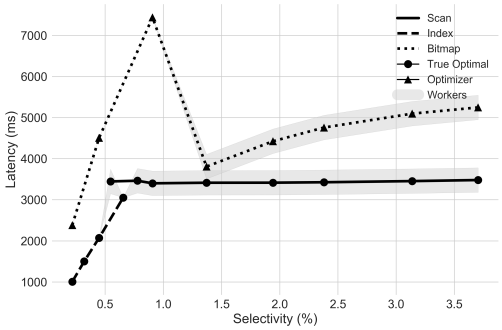


(e) Average error rate on `PARTSUPP` table.

Figure 2.3: Error rates for queries on TPC-H tables.



(a) Query 9 should have multithreaded.



(b) Query 10 should be index/scan.

Figure 2.4: Examples of poor decision heuristics.

2.4.2 Poor Decisions

Taking a deeper look at the lower selectivity range gives us an idea of the kinds of mistakes the optimizer made.

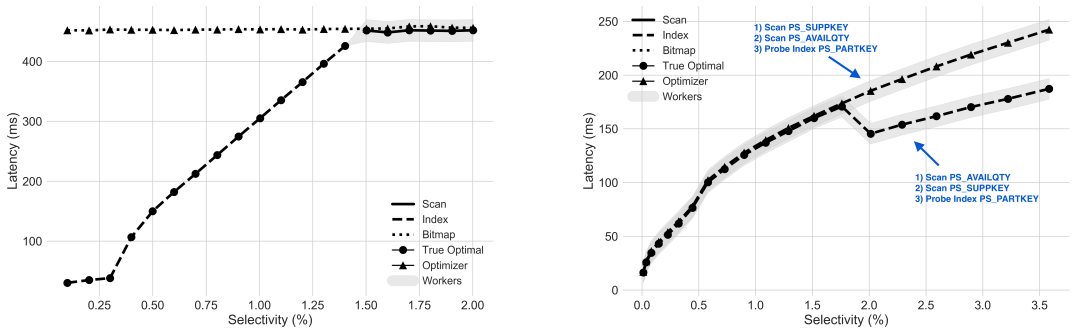
Parallelism

Parallelism is a crucial strategy in query execution. It involves adding workers to assist in the computational workload. That is not a trivial task; it consumes precious resources to allocate and manage workers and to combine their results. Additionally, a workload that can not be equitably divided causes workers to waste valuable cycles waiting for each other, often resulting in worse performance than without parallelism. Still, in the right circumstances, it is a powerful tool to leverage.

Query 9 varies column `l_extendedprice` (from 75,000 to 120,000) while fixing `l_partkey` at 100,000:

```
SELECT cast (count(l_extendedprice) as float)
        FROM lineitem WHERE l_extendedprice > 75000
        AND l_partkey < 100000;
```

Query 9 (Figure 2.4a) should have switched to multithreading at 0.05% selectivity, but the optimizer incorrectly surmised the cost was too high and avoided parallelism. The system could have used more workers traversing `l_extendedprice`



(a) Query 38 poor decisions up to 1.5%. (b) Query 70 switch column enumeration.

Figure 2.5: More examples of poor decision heuristics.

as more and more of the records became applicable to the result set.

Wrong Path

Query 10 explores the interaction of columns `l_extendedprice` and `l_suppkey`, fixing `l_suppkey` at 5,000 and varying `l_extendedprice` from 75,000 to 95,000:

```
SELECT cast (count(l_extendedprice) as float)
        FROM lineitem WHERE l_extendedprice > 75000
        AND l_suppkey < 50000;
```

Figure 2.4b shows that Query 10 should have started with a btree index scan, switching to a parallel table scan at 0.075%. Instead, the optimizer ignored those options, settling on a single-threaded bitmap index scan adding additional workers at 1.5% resulting in a very high penalty throughout the range.

Not Robust Enough

A poor decision is easily seen when the cost is many times higher (as in seconds), but even a few hundred milliseconds matter. Figure 2.5a shows the results of *Query 38* on `PARTSUPP` table. Query 38 varies only `ps_supplycost` column from 2 to 21:

```
SELECT cast (count(PS_PARTKEY) as float)
        FROM partsupp WHERE PS_SUPPLYCOST < 2;
```

Unlike the other tables, PARTSUPP is small in size and distribution range. Still, we can see that access path selection and optimizer choices are critical. The optimizer should have recognized that an index scan and additional workers at 1.5% selectivity was the optimal choice. Instead, the optimizer could see very little difference in the plans across the selectivity range (opting, again, for bitmap index scan). Across all tests, we see that the optimizer tends to overly favor bitmap index scans, with no way to influence the optimizer otherwise.

Sticking With a Plan Too Long

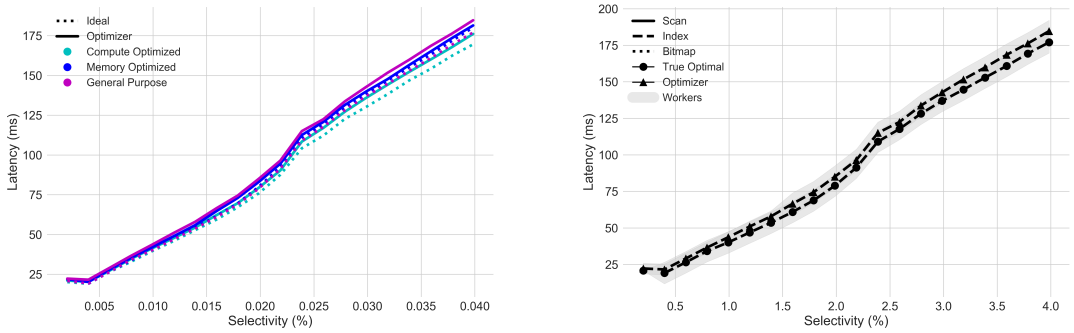
Query 70 on table PARTSUPP varies PS_PARTKEY from 33,000 to 659,981, PS_AVAILQTY from 165 to 3,281, and fixes PS_SUPPKEY at 33,000:

```
SELECT cast (count(PS_PARTKEY) as float) FROM partsupp
        WHERE PS_SUPPKEY < 33000
        AND PS_PARTKEY < 33000 AND PS_AVAILQTY < 165;
```

At first glance, Figure 2.5b appears to show that the optimizer and ideal plan are the same with different results. While it is true, they both determined that they should scan PS_SUPPKEY first, followed by a scan of PS_AVAILQTY, and finally a btree probe on PS_PARTKEY. However, at approximately 2% selectivity, the ideal plan varied in a subtle but essential way. It swapped the order of scans, performing the PS_AVAILQTY scan first, while the optimizer never varied from its original plan.

2.4.3 Good Decisions

The picture is not always so bleak. At times (albeit rarely, especially in complex cases), an optimizer does make the ideal decision. Figure 2.6 illustrates that



(a) Query 49 optimizer is ideal.

(b) Query 49 optimizer's correct decisions.

Figure 2.6: Sometimes optimizers makes ideal decisions.

Query 49 (fixing `PS_SUPPLYCOST` and varying `PS_PARTKEY` from 20,000 to 399,981) performed at optimal on all three hardware scenarios:

```
SELECT cast (count(PS_PARTKEY) as float)
        FROM partsupp where PS_SUPPLYCOST < 200
        AND PS_PARTKEY < 20000;
```

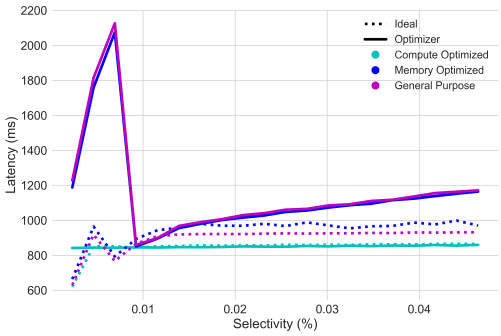
With this simple query, hardware did not have a noticeable impact on performance or to the decision process. Is that always the case?

2.4.4 Hardware

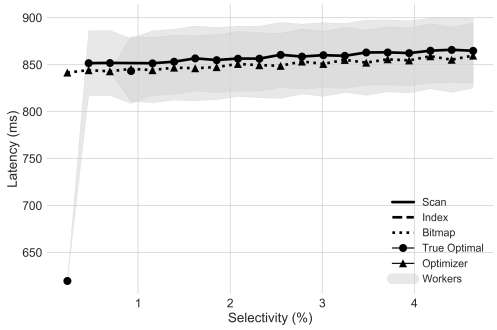
At times, hardware will feel different effects from the same decision. Consider *Query 90* (fixes `O_TOTALPRICE` to 73192 and varies `O_CUSTKEY` from 15000 to 299981):

```
SELECT cast (count(O_ORDERKEY) as float)
        FROM orders WHERE O_TOTALPRICE < 73192
        AND O_CUSTKEY < 15000;
```

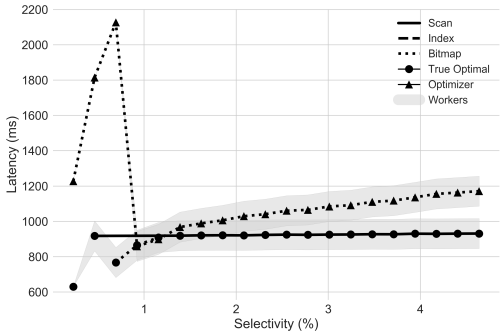
In Figure 2.7, we see the optimizers for Query 90 on all three systems made the same decision; start with a bitmap index scan and add parallelism at about 1%



(a) Hardware performs differently.



(b) Compute optimized performs near ideal.



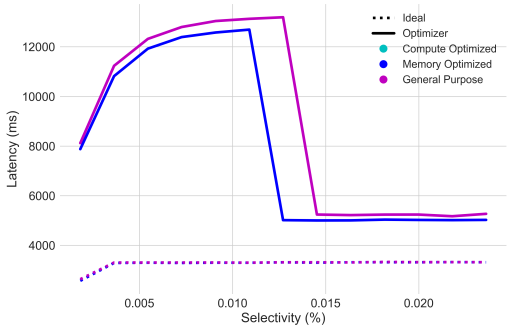
(c) General purpose and memory optimized feel the pain of delay in parallelism

Figure 2.7: Hardware makes decisions for Query 90 painfully obvious.

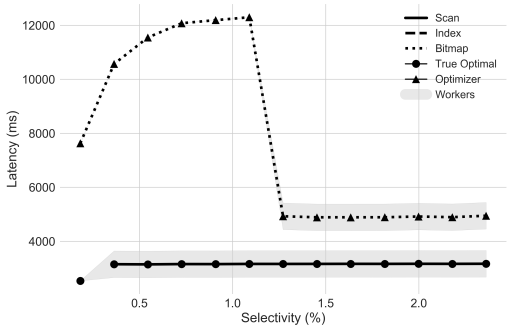
selectivity. The ideal performed a btree probe until 0.025% and switched to a parallel scan from then on. The compute optimized system performed almost as well as the ideal. The general purpose and memory-optimized systems, however, felt the pain of delay in parallelism much more as can be seen in the spike in latency.

It is much more common to see the effects demonstrated by *Query 5* (shown in Figure 2.8), where we see the crossover point move slightly between systems (unbeknownst to the optimizer):

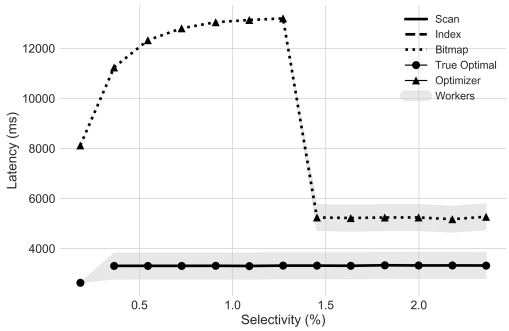
```
SELECT cast (count(l_extendedprice) as float)
        FROM lineitem WHERE l_discount < 0.01
        AND l_quantity < 2 ;
```



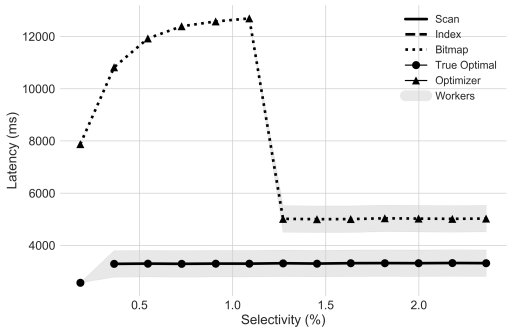
(a) Query 5 all hardware variations.



(b) Query 5 compute optimized hardware.



(c) Query 5 general purpose hardware.



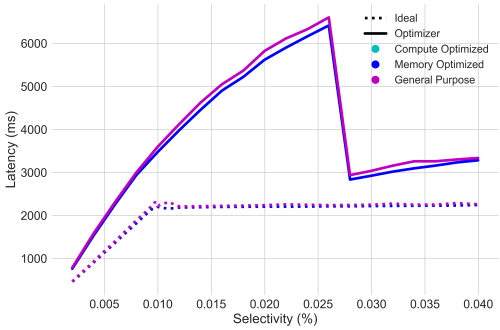
(d) Query 5 memory optimized hardware.

Figure 2.8: Hardware affects crossover point in optimizer vs ideal.

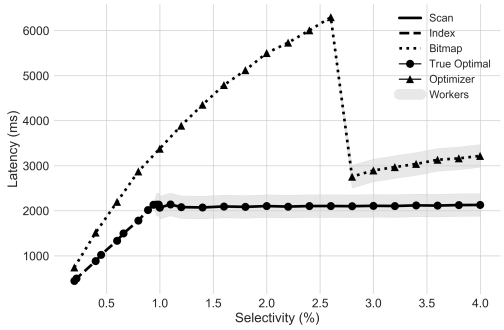
We fix `l_discount` and vary `l_quantity` from 2 to 12. The optimizers on the compute-optimized and general-purpose systems determined the appropriate time to introduce parallelism is at 1.25% selectivity. The memory-optimized system shifted the decision to 1.5%. While the true ideal was none of those options (and did not vary per system), we see that the underlying hardware can influence the optimizer.

It is not always true that hardware affects the optimizer. For example, Query 3 (shown earlier), shows no difference between optimizers on various hardware (see Figure 2.9).

It is clear that there are many ways in which the optimizer could make poor decisions, but there is only a single ideal decision. That is asking a lot from an



(a) All hardware variations on Query 3.



(b) Decisions on compute optimized hardware.

Figure 2.9: Hardware does not always affect optimizer decisions.

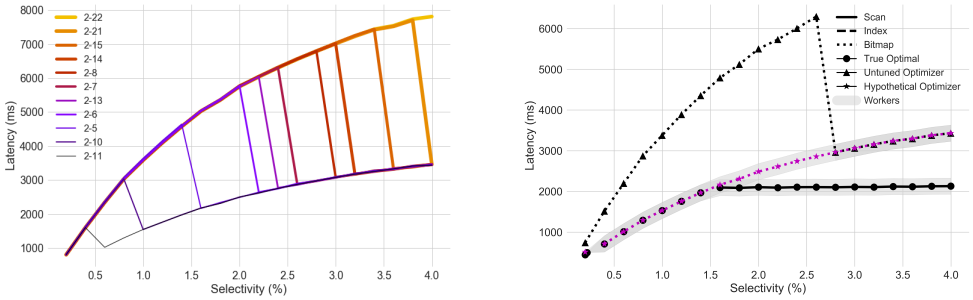
optimizer. We tested the optimizer with settings set by best practices. Tuning is tricky and changing knobs often has consequences on other queries without a single universally optimal solution. The best we can hope for is a setting that performs well, on average. Could a well-tuned optimizer perform better?

2.4.5 Tuning the Optimizer

Test 2 is designed to locate the ideal setting for PostgreSQL's optimizer knobs (shown in Table 1.4 in Chapter 1). The optimizer settings are varied to find the combination that yields the greatest performance for Query 3.

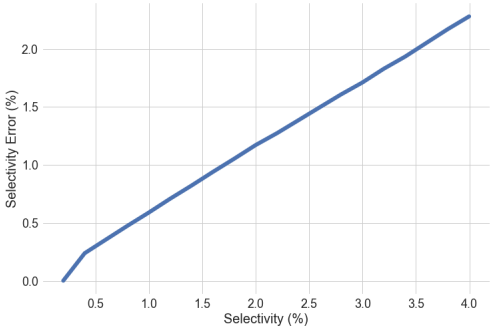
However, these are system-wide settings, and as shown in Table 2.6, there is no setting that is advantageous in all situations. Still, would a **hypothetical optimizer** that could constantly shift its system configuration, per query, perform ideally?

After running Test 2 on all tuning knob permutations, we can compute the best setting per query by again finding the results with the lowest latency and call that the hypothetical optimizer. Table 2.6 lists the results of the hypothetical optimizer at each selectivity point. *Selectivity* is the observed value, *Est Selectivity* is the selectivity computed by the actual optimizer's statistics, and *Cost* is the actual optimizer's



(a) Variations of settings and effect on crossover point.

(b) Optimizer, ideal, and hypothetical performance.



(c) Optimizer error in table statistics.

Figure 2.10: Using Query 3 to tune the optimizer.

assessment of the cost of this plan.

It is interesting to note that the optimizer's estimate for selectivity is always too low. Figure 2.10c shows us that the error rate (the difference between the estimated and actual selectivity) grew linearly as the selectivity grew, telling us that the estimates were getting worse as more records were selected. To some extent, the estimate does not matter once we pass the crossover point since there is no incentive to make any further changes to the plan (at that point, nothing beats). However, Figure 2.10a shows us that the settings affect the critical crossover point, causing it to shift (as can be seen from the color gradation).

Also interesting to note is that the cost estimates for the plans vary wildly. We

Latency	Selectivity	Est Selectivity	Cost	Batch
511.385	0.1988	0.1954	472431.66	2-4
712.776	0.3990	0.1590	587013.88	2-12
1013.439	0.5992	0.2408	774247.06	2-12
1293.960	0.7983	0.3226	915450.83	2-11
1532.453	0.9987	0.4086	1032826.27	2-12
1757.947	1.1992	0.4895	1110667.12	2-10
1967.899	1.3989	0.5760	1182007.94	2-11
2164.760	1.5992	0.6584	698724.21	2-5
2309.643	1.8001	0.7449	1270587.79	2-11
2487.979	1.9995	0.8272	820180.22	2-5
2612.282	2.1996	0.9248	1314746.65	2-10
2744.785	2.3990	1.0133	1323399.84	2-16
2857.196	2.5995	1.1017	1323648.42	2-13
2963.076	2.7985	1.1902	1331618.82	2-16
3063.461	2.9981	1.2870	1317797.14	2-19
3159.384	3.1985	1.3689	1324852.43	2-16
3241.860	3.3986	1.4641	1322325.92	2-10
3298.498	3.5987	1.5454	1294720.80	2-19
3379.634	3.7980	1.6267	1289191.27	2-20
3432.103	3.9972	1.7164	1298270.97	2-10

Batch Settings Legend

Batch	Non-default Setting	Value
2-4	seq_page_cost	5
2-5	random_page_cost	1
2-10	cpu_tuple_cost	.03
2-11	cpu_tuple_cost	.04
2-12	cpu_tuple_cost	.05
2-13	cpu_operator_cost	0.0035
2-16	cpu_operator_cost	0.0055
2-19	parallel_setup_cost	100
2-20	parallel_setup_cost	5000

Table 2.6: Settings for a hypothetical optimizer for Query 3.

discussed earlier that cost estimates are in undefined units and have little meaning on their own; however, relative to each other, they have significance. Some estimates continue in a linear ascent, which means that the optimizer understood their relationship to each other correctly, but that is not the case across the range.

Note that the *batch* column indicates that for the 20 queries, there are 19 changes in optimizer knobs necessary to achieve the performance of the hypothetical optimizer. While still not ideal, Figure 2.10b shows us that the hypothetical optimizer (purple line) would perform significantly better than the existing optimizer. Ultimately, it still suffers from the problems that plague the actual optimizer.

Chapter 3: Design

3.1. Introduction

There are two critical components to the design of a neural network; feature engineering and network architecture. *Feature engineering* is the method in which we design the features used to train a neural network. Recall that features are parameters on which weights are trained through a training phase and computed through back-propagation. *Network architecture* refers to the physical design of our network; how many layers, the number of neurons in each layer, and the techniques we use for activation and regularization.

3.2. Feature Engineering

Feature engineering is the process of creating individual, measurable properties from data that make machine learning algorithms work. The algorithms learn to weight features to compute a prediction. Loss, or *cost*, is the distance a prediction is from the actual value. The best model is the one with the lowest loss. Such an approach is also known as *supervised learning*. Care must be taken in the selection and engineering of features because the model is only able to learn from the features provided. The feature space consists of features in these categories:

- Hardware features (vary by hardware)

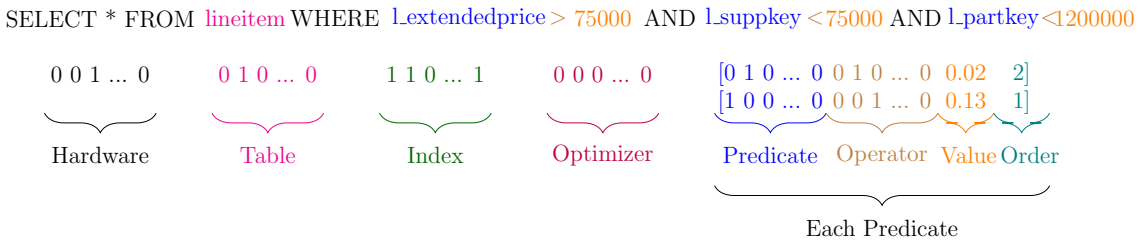


Figure 3.1: Feature vectorization of queries.

- System features (vary by database and includes indexes, etc.)
- Optimizer features (vary by configuration)
- Predicate features (vary by query)

Figure 3.1 shows an example of features presented in this thesis. **Hardware** features are attributes of the underlying physical system, such as processor type, supported instructions, and memory attributes. These are characteristics that could affect the latency of the query (or, of the prediction). When new hardware (such as a new processor instruction) becomes supported that could improve performance, its characteristics could be added to the feature vectors and used for training. **System** features are those that vary by implementation, such as the indexes available to the system.

Optimizer features are exposed knobs that the system would allow for tuning. While not required, they could include arbitrary metadata such as how much pressure was on the network or memory at the time of execution (that obviously would affect latency). These are fine-tuning fields that allow us to improve or research the model further.

Predicate features are specifically related to the query. They represent the column, operation, value, and ordinal position of the predicate in the plan. Some of these are found in the query (operation, column, and value), while others are

permuted (ordinal position).

Individual features can either be categorical or continuous values. *Categorical* features are ones whose value can take on one of a limited set. For example, there are only so many kinds of processors, or you either support a processor instruction, or you do not (for example, SIMD, an Intel processor extension that allows a single instruction execution on multiple data). *Continuous* variables can be measured and take on any value. For example, the clock speed of a CPU can be any value. The choice between the two can be subjective; for example, the number of processor cores. Is it continuous in that you can have any number of cores, or is it categorical because they only come in specific packages?

Another important consideration is that models learn significance in the magnitude or distance between values. In the processor core example, there is a linear relationship between 4 cores and 8 cores. Twice as many cores are twice as much computing power. There is value in preserving the meaning inherent in the magnitude of the values. In other cases, preserving the distance might lead to learning a false meaning. For example, level 2 cache is not twice as much as level 1 cache. They are simply labels, and it would not make sense to preserve those.

In such cases, we use a technique called *one-hot encoding* to recode categorical values to a series of binary features. Figure 3.2 illustrates the conversion of a categorical value "operator" to its one-hot encoded counterpart. Rather than represent each operator as a scalar value, we recode to a set of values representing all possible categories. The particular column desired would get a `1` value while the rest are encoded as `0`, denoting they are not applicable. Mathematically, they effectively cancel out by multiplying their respective trained weights by zero.

Some features in our model are one-hot encoded, including many of the predicate features (Figure 3.3). For example, the predicate `l_partkey < 1200000` is made

Operator
1 (Greater Than)
2 (Less Than)
3 (Equals)

Greater Than	Less Than	Equals
1	0	0
0	1	0
0	0	1

Figure 3.2: Example of one-hot encoding.

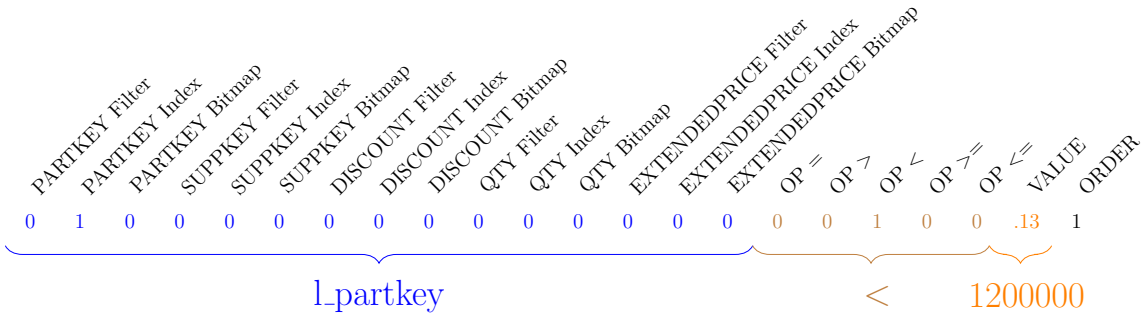


Figure 3.3: One-hot encoding of a predicate.

of four components:

- `l_partkey` *column*
- less than (`<`) *operator*
- `1200000` *value*
- ordinal *position*

The `l_partkey` portion represents the *method* with which the column is accessed; scan (or, filter), btree probe, or bitmap scan. We test each of these access methods by generating permutations of each access method. The **operator** is fixed by the query and is represented, as shown in Figure 3.3. Finally, the **value** is *normalized* to improve model accuracy.

Normalization is the adjustment of the values so that the entire distribution falls into alignment. One common method (and the one used in this thesis) is *L2 norm*, which is the square root of the sum of the squared values in the set:

$$\sqrt{\sum_{x=1}^{n} a_x^2}$$

The L2 norm can then be divided by each value to produce a value between 0 and 1 (and conversely multiplied by L2 norm to return the original value). The magnitude of the numbers is effectively reduced without distorting differences in the ranges of values.

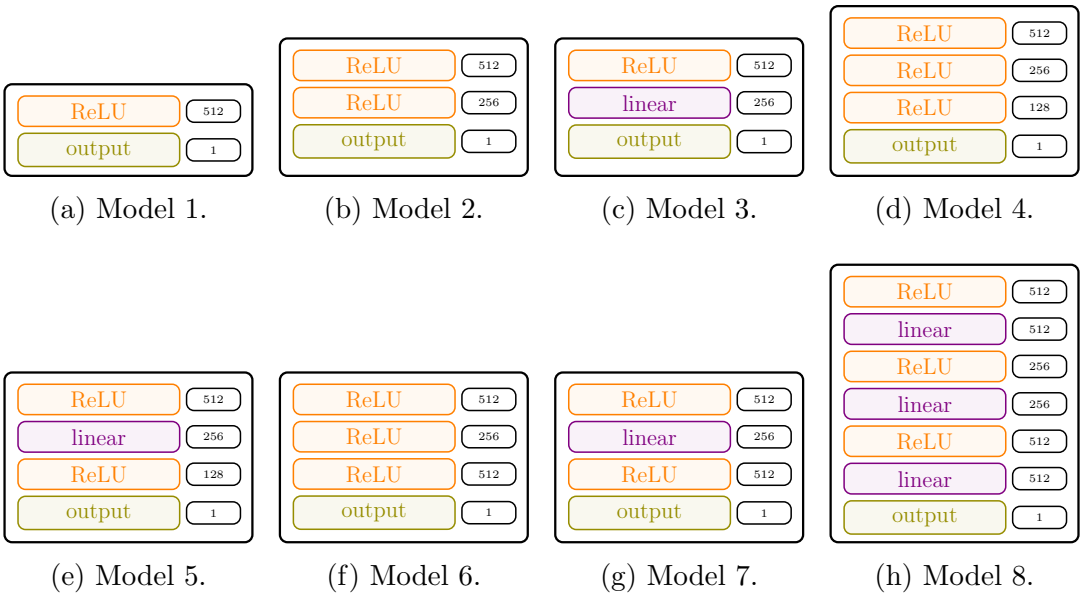
The final portion of each predicate is the **position**. As discussed, the permutations of each of the variations are also applied here. We need to generate $(m \times n)^n$ permutations (where m is the number of access methods, 3 in our case, and n is the number of columns) for each predicate.

3.3. Model Architecture

In Chapter 1 (§1.1.3), we introduced the multi-layer perceptron (MLP) deep neural network architecture. MLPs are incredibly versatile and exceptionally good at regression problems. They also have the added benefit of being straightforward to implement since each layer fully connected to the next with no other complex algorithms (such as convolutions). The bulk of the computation happens at training time, and with a trained network, extracting predictions no longer requires the original data.

Architecture is only one of the decisions made when designing a deep neural network. This paper addresses the size of the network, as well. Should the model be one **monolithic model** for the entire system, or would are many small, **specialized models** more advantageous?

Both approaches have attractive qualities. A large model is more simple and contains less moving parts, although its size consumes significantly more resources to



(a) Model 1. (b) Model 2. (c) Model 3. (d) Model 4.

(e) Model 5. (f) Model 6. (g) Model 7. (h) Model 8.

Figure 3.4: Model Architectures (scale factor=1).

train. Would its more general nature hinder its predictive power? The small models have fewer components per model and are more resilient (because we only need to retrain affected models). However, they have redundant features (such as hardware characteristics) in each model.

To test the hypothesis of which is better, I develop both approaches. We test eight deep neural network architectures on each of the small and large models with varying *hyperparameters*. Hyperparameters are settings that can be tuned in the machine learning algorithm itself to affect its behavior, for example, the number of layers, neurons in each layer, or epochs in which to train. An epoch is when all of the training vectors are used once to update the weights. In the architectures, we vary the number of layers, neurons, and epochs to find optimal parameters. We apply a variety of activation functions and observe the effect of *overfitting*. Overfitting occurs when the analysis corresponds too closely to the training dataset.

The architectures tested are based on the eight variations represented in Figure 3.4. The input layer is not represented in the figure for concision and varies per

Name	Parameters
Small 1 (512)	41,985
Small 2 & 3 (512)	173,057
Small 4 & 5 (512)	205,825
Small 6 & 7 (512)	304,897
Small 8 (512)	896,001
Large 1 (1024)	210,945
Large 2 & 3 (1024)	735,233
Large 4 & 5 (1024)	866,305
Large 6 & 7 (1024)	1,261,057
Large 8 (1024)	3,622,913

Table 3.1: Training parameters for each architecture.

training set. *Models 3, 5, 7, and 8* alternate the activation functions on the layers between linear and ReLU, similar to previous work (Kipf et al., 2018). *Models 6, 7, and 8* also follow that research in the way they structure their perceptrons in an hourglass shape (decreasing as we get toward the middle, then increasing them back to their original shape).

We scale the hidden layer (linear and ReLU layers) neurons in each of the hidden layers with three variations. Figure 3.4 shows the baseline scale referred to as factor 1. The small models test factors 1.0, 0.5, and 0.25, while the large model tests 2.0, 1.5, and 1.0. For example, a scale factor of 1.0 would have 512 neurons in the first hidden layer, 0.5 would be 256, and 2.0 would be 1024. The total number of trained parameters per model is in Table 3.1.

All models were trained for 200 epochs, and with and without L2 normalization on the latency label. The final output layer of all architectures produces a single prediction for latency.

The *parameters* shown in Table 3.1 grow exponentially with the number of neurons and layers added to the network. For example, All small models for `LINEITEM` table have 80 input features. Consider the resulting network for Model 8 in Table 3.2.

Layer (type)	Output Shape	Param Count
relu_1 (Dense)	128	10368
linear_2 (Dense)	128	16512
relu_3 (Dense)	64	8256
linear_4 (Dense)	64	4160
relu_5 (Dense)	128	8320
linear_6 (Dense)	128	16512
output_7 (Dense)	1	129
Total params: 64,257		

Table 3.2: Example `Lineitem` table neural network.

Our 80 features explode to 64,257 parameters. Recall that the formula for a regression is $Y = \beta_0 + \beta_1 X_i + \ldots \beta_n X_n$. Some number ($n$) of features ($X$) are multiplied by their respective weights ($\beta_{1..n}$) and added to the bias (β_0). That occurs for every neuron, so the total number of parameters in a layer are simply the sums of the regressions per neuron times the number of neurons ($Y_{layer} = \sum_{x=1}^{n_{neurons}} Y_x$). Consequently, the number of total parameters in the model are the sum of all the parameters in each layer ($Y_{tot} = \sum_{x=1}^{n_{layers}} Y_x$). Given that, for this model, we get:

$$relu\_1 = (80 \times 128) + 80 = 10,368 \tag{3.1}$$

$$linear\_2 = (128 \times 128) + 128 = 16,512 \tag{3.2}$$

$$relu\_3 = (128 \times 64) + 64 = 8,256 \tag{3.3}$$

$$linear\_4 = (64 \times 64) + 64 = 4,160 \tag{3.4}$$

$$relu\_5 = (64 \times 128) + 128 = 8,320 \tag{3.5}$$

$$linear\_6 = (128 \times 128) + 128 = 16,512 \tag{3.6}$$

$$output\_7 = (1 \times 128) + 1 = 129 \tag{3.7}$$

$$Total = 64,257 \tag{3.8}$$

Layer (type)	Output Shape	Param Count
relu_1 (Dense)	256	52480
linear_2 (Dense)	256	65792
relu_3 (Dense)	128	32896
linear_4 (Dense)	128	16512
relu_5 (Dense)	256	33024
linear_6 (Dense)	256	65792
output_7 (Dense)	1	257
Total params: 266,753		

Table 3.3: Example of monolithic neural network.

3.3.1 Monolithic and Specialized Models

In the **specialized model** approach, each table has its model, decreasing the feature space significantly. Smaller models should be faster to train, more accurate, and affect less of the system when change is detected.

The **monolithic model** must contain all of the data in each of the small models; thus, it has approximately 204 features, necessitating the larger scaling factors. As discussed earlier, training larger models has performance implications. The larger model has more neurons and significantly more parameters. Adding a single neuron also results in adding connections to every other neuron in the next layer, and so on. Table 3.3 summarizes the parameters at each layer. The 204 features result in 266,753 parameters to train.

3.3.2 Making Predictions

The trained network computes the latency for a given plan. A prediction for each query plan is required to determine the best plan. For example, given a btree index on column `l_suppkey` and the query (leaving multithreading off the table for a moment):

The lowest prediction of these models determines the best plan:

(A) **Scan** l_suppkey then **filter scan** l_extendedprice

(B) **Scan** l_extendedprice then **filter scan** l_suppkey

(C) **Index scan** l_suppkey then **filter scan** l_extendedprice

(D) **Index scan** l_suppkey then **bitmap scan** l_extendedprice

(E) **Scan** l_extendedprice then **bitmap scan** l_suppkey

(F) **Scan** l_extendedprice then **index scan** l_suppkey

(G) **Bitmap scan** l_suppkey then **filter scan** l_extendedprice

(H) **Bitmap scan** l_suppkey then **bitmap scan** l_extendedprice

(I) **Bitmap scan** l_extendedprice then **filter scan** l_suppkey

Chapter 4: Findings

In Chapter 2, I described the design of the neural networks and feature vectors. This chapter follows with a brief discussion on the training process (§4.1, continuing to the results of experiments on the small, purpose-built (§4.2) and the monolithic, system-wide models (§4.3). Section 4.4 concludes the chapter with a comparison between the approaches.

4.1. Training

Training neural networks follow a typical formula: clean the data, extract feature vectors, train models (tuning hyperparameters as you go), and take the model with the lowest loss as winner.

When tuning hyperparameters, there are a variety of successful methods (such as genetic or randomized methods) for determining optimal settings. *Grid search* attempts to find the optimal values of hyperparameters using an exhaustive method. While it takes significant time, it provides accurate, measured results. Epochs and the number of layers and neurons are examples of a few hyperparameters in our models. I addressed the numbers of layers and neurons in the careful design of 8 architectures described in Chapter 2. Since the goal of this thesis is not to find the optimal training time, I tested 50, 100, and 200 epochs on all models. Subsequently, I retrained all models at 200 epochs (the highest common denominator) for simplicity. All training

was done both on Google Colab and a fresh Apple MacBook Pro with the following specifications:

- Software: Python 3, Tensorflow 1.13.1
- Local Hardware: MacBook Pro 2.9 GHz Intel Core i9 (Mojave 10.14.6), 2400 MHz clock speed, 32 GB DDR4 RAM with SSD
- Cloud Computing: Google Colab
  - GPU: 1x Tesla K80 with 2496 CUDA cores, compute 3.7, 12GB (11.439GB Usable) GDDR5 VRAM.
  - CPU: 1x single core hyper threaded i.e(1 core, 2 threads) Xeon Processors @2.3Ghz (No Turbo Boost), 45MB Cache.

4.1.1 Overfitting

We see an example of overfitting in Figure 4.1. The training loss training continues to decline as it gets better and better at predicting the training set. Against the unseen validation set, however, it effectively loses its ability to make predictions on new data. This model is going to be as good as it is ever going to get at 60 epochs, and additional epochs are just a waste of time. Keras has a built-in mechanism to cope with this problem by saving the latest best model until a better model comes along. That frees us to train a network for as many epochs as we desire without fear of making matters worse. In the worst case, we wasted time.

4.1.2 Loss Functions and Accuracy

The two loss functions used to compare the model were *mean squared error* (MSE) and *mean average percentage error* (MAPE). Figure 4.2 compares the two on a sample model and data from the `PARTSUPP` table.

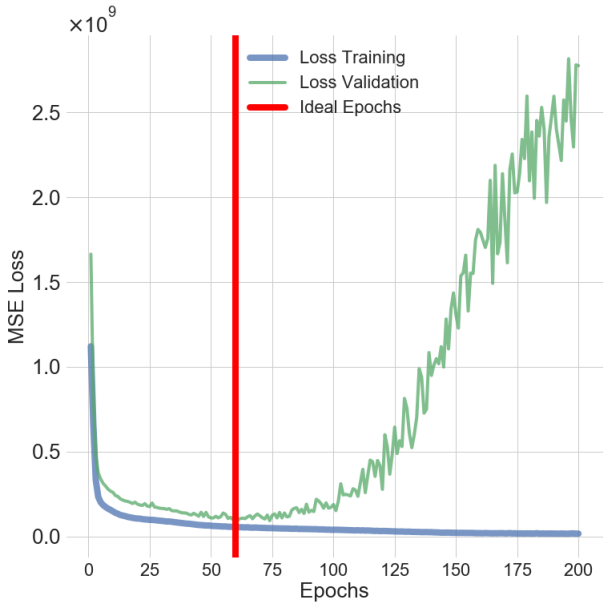
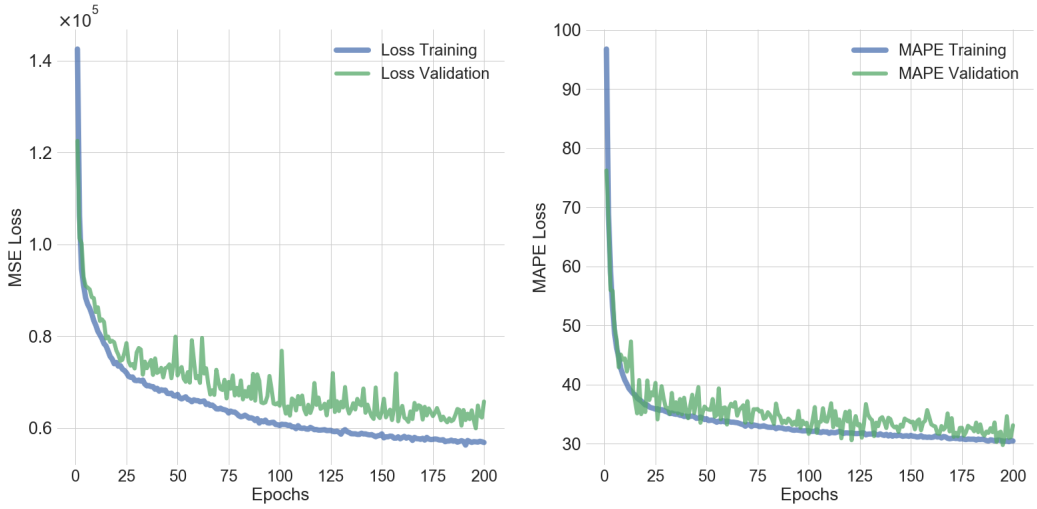


Figure 4.1: MSE loss for `LINEITEM` overfitting after 60 epochs



(a) Mean Squared Error (MSE) (b) Mean Average Pct Error (MAPE)

Figure 4.2: Training and validation loss for `PARTSUPP` over 200 epochs

MSE measures the average of the squares of the difference between the actual and the predicted latency (the *loss*, or *errors*) in an epoch in this way:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

Put another way, MSE is the *mean* ($\frac{1}{n}\sum_{i=1}^{n}$) of the *squares of the errors* $(Y_i - \hat{Y}_i)^2$, where Y_i is the actual value, and $\hat{Y}_i$ is the predicted value. We use MSE to compare the performance of an epoch to its peers. The winning recipe is the epoch with the lowest cost. MSE is somewhat interpretable but represented in squared units. For example, the MSE of the model in Figure 4.1 at epoch 100 is $64,822ms^2$. The average loss over the entire model ($\sqrt{MSE}$) is thus $254.6ms$.

MAPE expresses accuracy as a percentage of deviation from the actual. It is computed as follows:

$$MAPE = \frac{100\%}{n}\sum_{i=1}^{n}\left|\frac{Y_i - \hat{Y}_i}{Y_i}\right|$$

MAPE is the *percentage* ($\frac{100\%}{n}\sum_{i=1}^{n}$) of the *proportion the prediction that is error* ($\left|\frac{Y_i - \hat{Y}_i}{Y_i}\right|$), where Y_i is the actual value, and $\hat{Y}_i$ is the predicted value. The absolute value of the proportion is necessary to remove the effects of direction from the error (positive or negative). MAPE is easily interpretable. For example, the MAPE for epoch 100 in the same model is 31.63% (each prediction deviates from the true value by 31%, on average).

Both of these give different contexts and have different effects on training. MSE is very susceptible to outliers since they are squared and added to the sum, discarding a potentially excellent model. Thus, penalizing overly large prediction errors severely. MAPE, on the other hand, grows very high as the actual value approaches zero. For example, we would not consider a prediction of $200ms$ for an actual $100ms$ latency to be a substantial deviation (it could be much worse). Unfortunately, MAPE would lead us to believe that that is much worse than a prediction of *5 seconds* compared

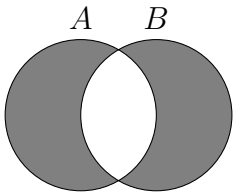


Figure 4.3: Overlapping Decisions (A & B) and Confusion Area (white)

to an actual *3 seconds*. When considering latencies that are very small (as we do in all fast-running systems), that poses a significant problem.

MAPE allows us to compare different models. For example, since the loss is a percentage, we can compare different architectures trained on vastly different sets (for example, a small model trained on a single table and a large model trained on all tables).

Loss does not tell the whole story. Since loss is a measurement of how far a prediction deviates from an expected value, the selection of what we are predicting is critical. In our approach, latency is an adequate substitution for our real objective: a correct plan. Our intuition tells us that the best plan has the lowest latency. Unfortunately, similar latencies between plans cause confusion. For example, consider the two hypothetical predictions illustrated in Figure 4.3. Let us assume that decision A is to scan, and decision B is to probe a btree. No model produces perfect decisions, so decision A, and decision B miss the center of the target by some acceptable *loss* (which we seek to minimize). However, the area in between in white (area of *confusion*) could belong to either A or B. Any value for A or B in the area of confusion may have the same loss, but drastically different decision points.

To understand how our model truly performs, we also compare the resulting *decisions* it recommends resulting from the lowest latency prediction with the *true*

ideal and the *actual optimizer*. I refer to the resulting decisions derived from the latency as the **learned optimizer**. The learned optimizer is created by composing predictions at each selectivity point and assembling them into a path.

In total, I trained 115 neural networks (23 small models per table and 23 larger models). Given that we are interested in the decisions the models make, we consider *error rate* and *path decisions* in addition to *training and validation loss*. Error rate is the difference between the actual latency of learned optimizer and the ideal path. Our goal is to do better than the actual optimizer at least. To do that, we review the decisions in detail. Each review proceeds as follows:

- Review the loss of all models
- Review the error rate of all models
- Review the decisions of the winning model

4.2. Specialized Models

There are four *specialized models*, each learning from queries on one of the tables tested. The 100 queries are as follows:

- `LINEITEM`: Queries 2-17
- `PART`: Queries 17-35
- `PARTSUPP`: Queries 36-85
- `ORDERS`: Queries 86-100

The small models had variations run on each architecture, identified by the number of neurons in the first hidden layer:

- Scale factor 1: 512 (on all models)
- Scale factor 0.5: 256 (on all models)
- Scale factor 0.25: 128 (omitted for model 8)

We identify a specific variation by the architecture number and variation (for example, Model 3-512).

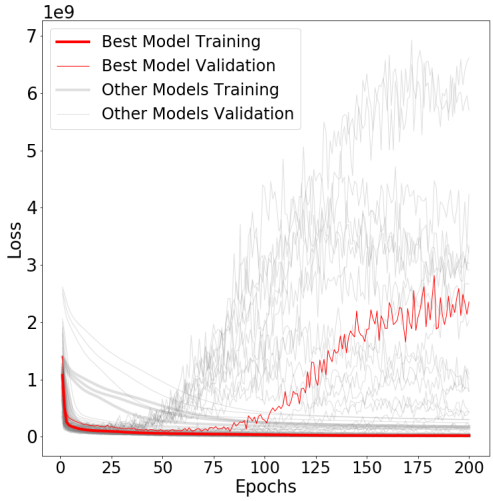
4.2.1 LINEITEM

Loss

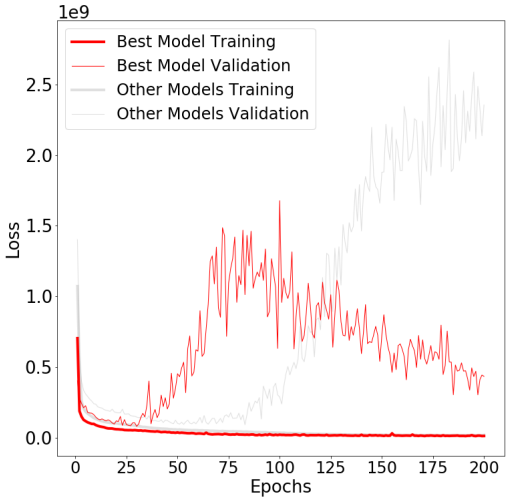
We begin by examining the loss of the model. Figure 4.4a shows the loss for all 23 models. We see that for `LINEITEM`, Model 3-512 performed best. After about 75 epochs, however, it suffered from overfitting. Despite overfitting almost immediately, Model 6-256 (Figure 4.4b) performed almost as well. At the cost of one second per epoch to train for only 25 epochs, it had remarkably good accuracy. MAPE, however, was abysmal (Figure 4.4c) showing losses at around 10,000%, reflecting the shortcomings of MAPE as a loss function. Comparing Model 3-512 with other members of the Model 3 family (256 and 128), we are misled to believe that they all performed about the same at 50 epochs (Figure 4.4d).

Error Rate

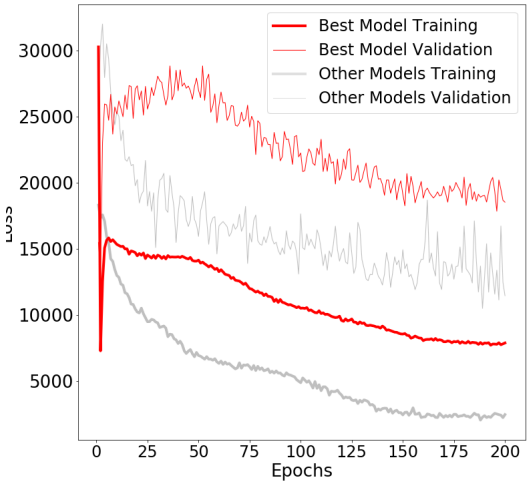
Figure 4.5 compares the performance of the optimizer (4.5a) with the winning model (4.5b), plotting the error rate for predictions of each query together. We can see that the predicted model performs better than the optimizer, even in the worst case.



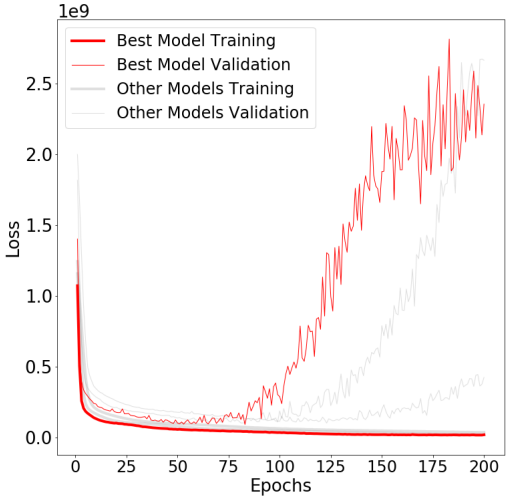
(a) MSE on all models (Model 6 in red)



(b) MSE on Model 6 (second best)



(c) MAPE on Model 3 and 6



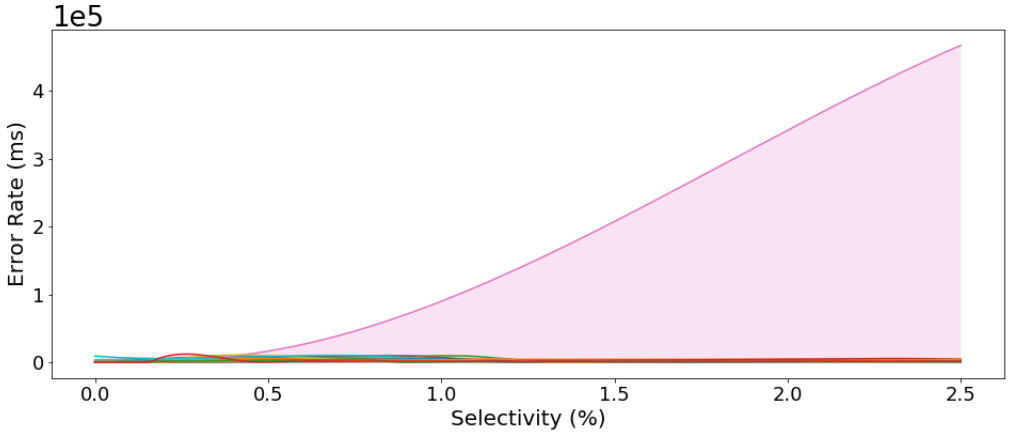
(d) MSE on all variations of Model 3

Figure 4.4: Loss on small `LINEITEM` table models

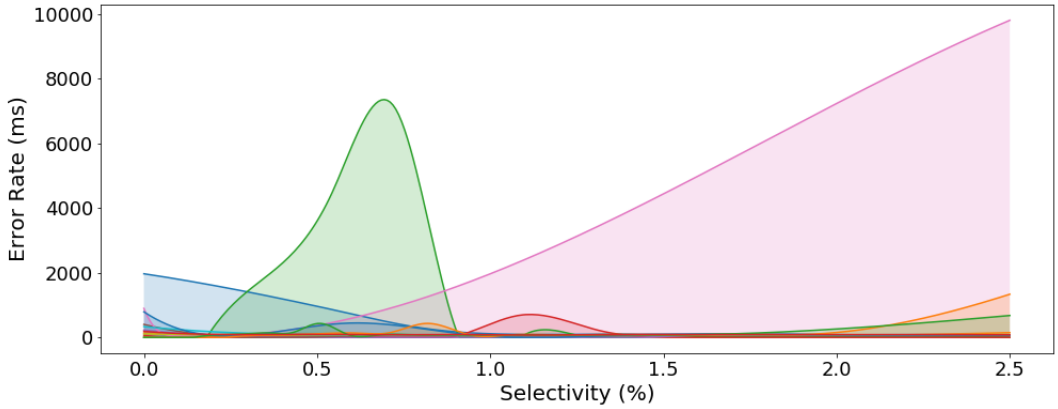
Decision Path

To determine the decision in a little more depth, we review decisions made for Query 6 and 13.

Query 6:



(a) Optimizer.



(b) Model 3-512.

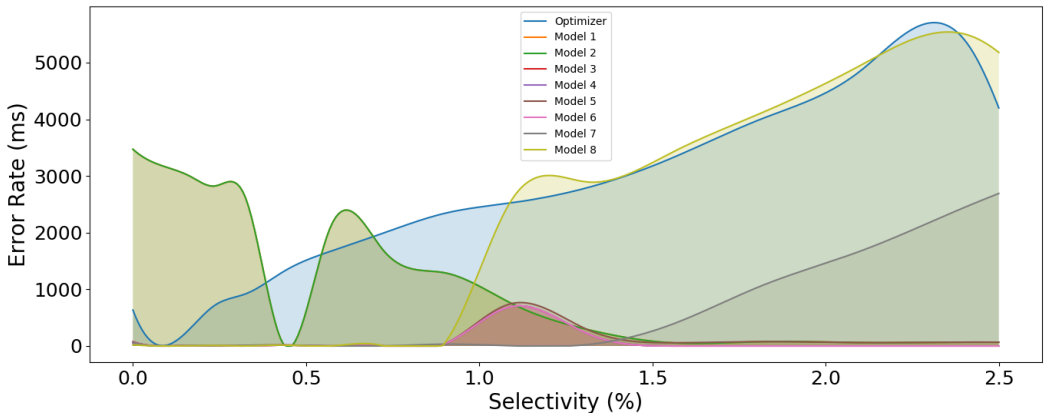
Figure 4.5: `LINEITEM` error rate by query.

```
SELECT count(l_partkey) FROM lineitem
        WHERE l_extendedprice > [100000-80000];
```

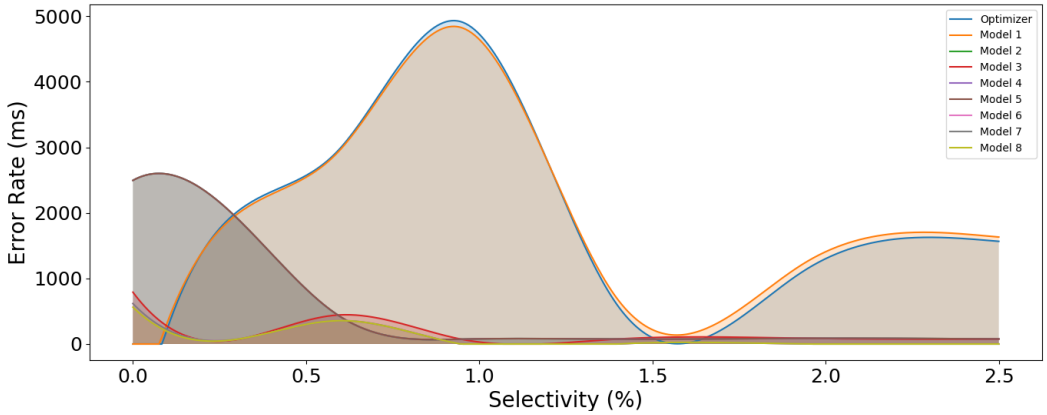
Query 13

```
SELECT count(l_partkey) FROM lineitem
        WHERE l_extendedprice >[75000-95000]
        AND l_suppkey < 75000 AND l_partkey < 1200000;
```

Figure 4.6 describes the error rate for Query 6, and 13 with each model plotted



(a) Query 6.



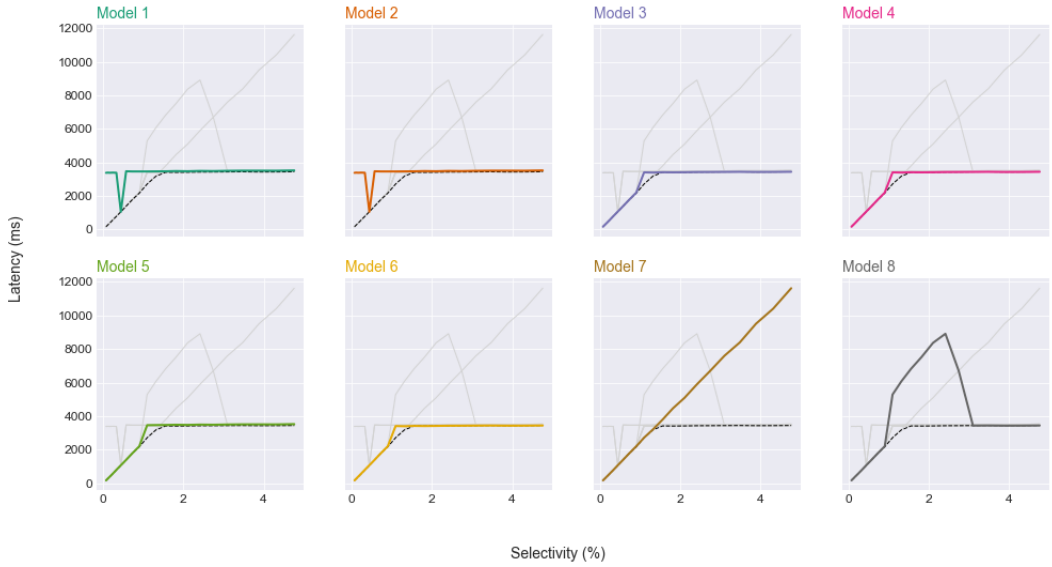
(b) Query 13.

Figure 4.6: `LINEITEM` error for all models by query (optimizer in blue).

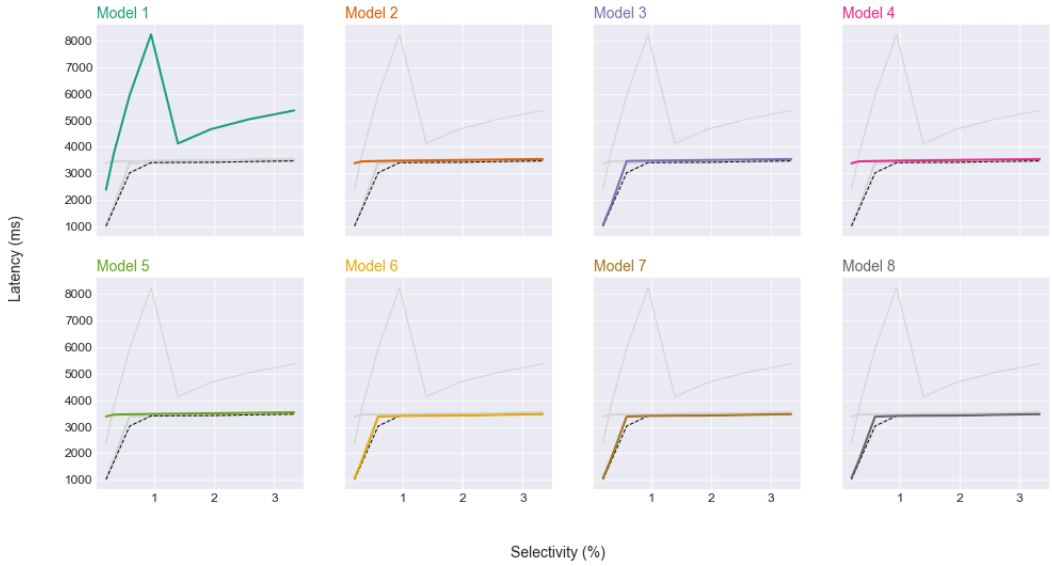
on a different line. Some models follow the optimizer (Model 8 in Figure 4.6a and Model 1 Figure 4.6b), while others perform exceptionally well (Model 3 and 6).

Consider Query 6 as shown in Figure 4.7. While none of the models performs perfectly, Models 3 through 6 roughly capture the correct path. Figure 4.8a shows the path in a bit more detail. We see that the models determined to switch to scanning just a little too soon. Still, much better than the optimizer that never comes to the correct conclusion.

The results of Query 13 shown in Figure 4.7b indicates Models 3, 6, 7, and 8



(a) Query 6.

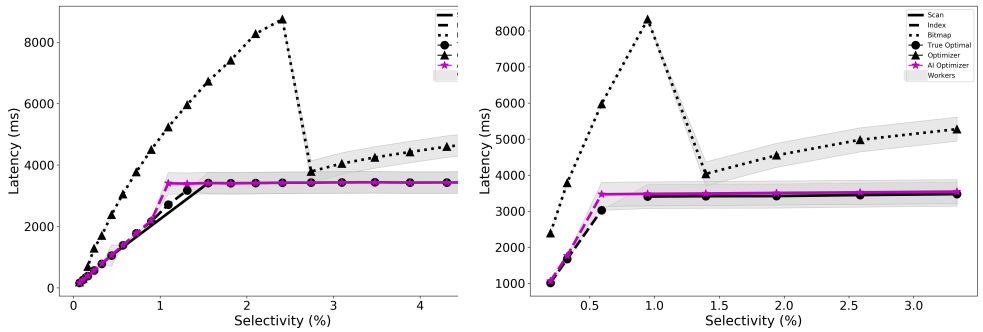


(b) Query 13.

Figure 4.7: `LINEITEM` model decisions compared to ideal.

performed well. Figure 4.8b clearly shows that Model 3 determined the access path perfectly.

Still, Model 3 is not without its flaws. Of the 17 queries in the tested table,



(a) Query 6. (b) Query 13.

Figure 4.8: `LINEITEM` Model 3-512 ("Learned Optimizer") decision detail.

it made mistakes on 3 of them (Queries 3, 5, and 6). Despite missing the target on those, it was only by 2 or 3 minor decision points (not switching fast enough) for a small latency loss. Again, the optimizer missed every one.

4.2.2 PART

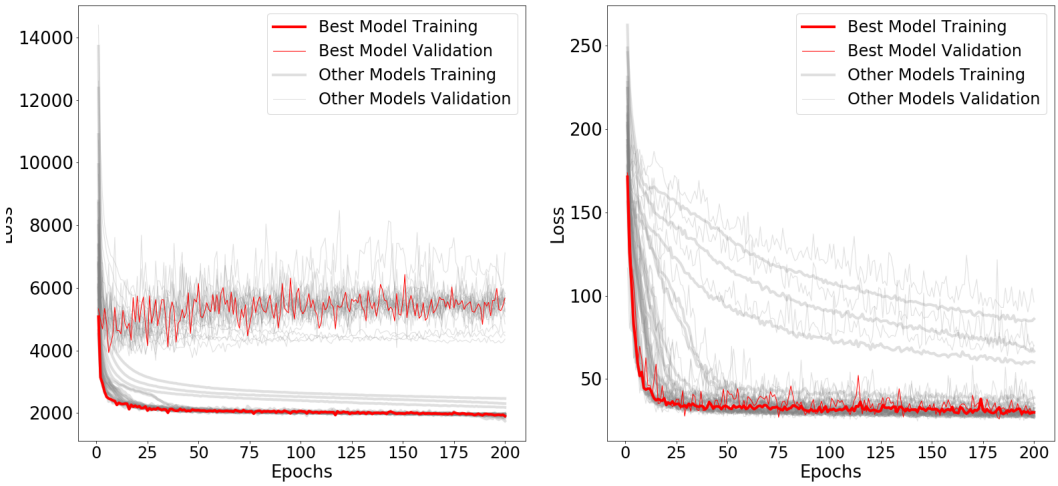
Recall that the `PART` table is small. How does our artificial optimizer fare?

Loss

In Figure 4.9a, we see that none of the models give us the satisfying loss progression that we would like to see. The validation loss stays stagnant and well above the training loss. In this smaller table, there are not as many training samples (intentionally) in the data. The MAPE (Figure 4.9b) is a bit more encouraging and gets into the teens. A closer look at winning Model 7-512 compared to its family (Figure 4.9c) does not give us much to go on. They seem as good as any other.

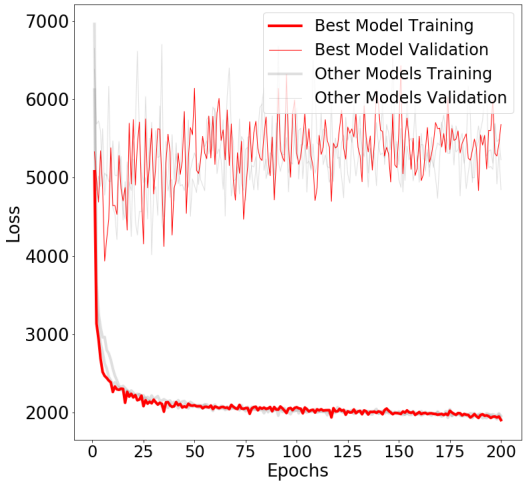
Error Rate

Figure 4.10 gives us some cause for concern. We see that Model 7-512 performs well except for one errant query that doubles the error of the optimizer. Even though we are led to believe that Model 7 performs well given its minimal error, it is the



(a) MSE on all models.

(b) MAPE on all models.



(c) MSE on all variations of Model 7.

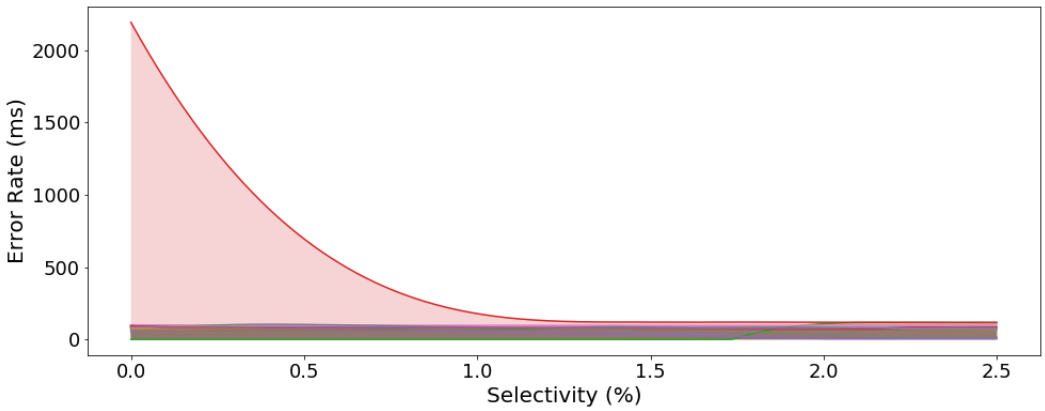
Figure 4.9: `PART` table models.

timing of decisions that is our main concern.

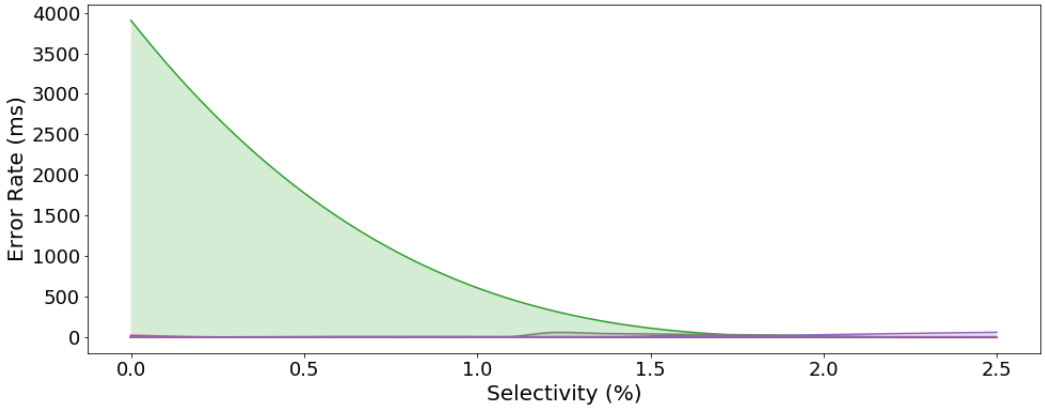
Decision Path

We will consider Queries 24 and 31 to determine the quality of decisions.

Query 24:



(a) Optimizer.



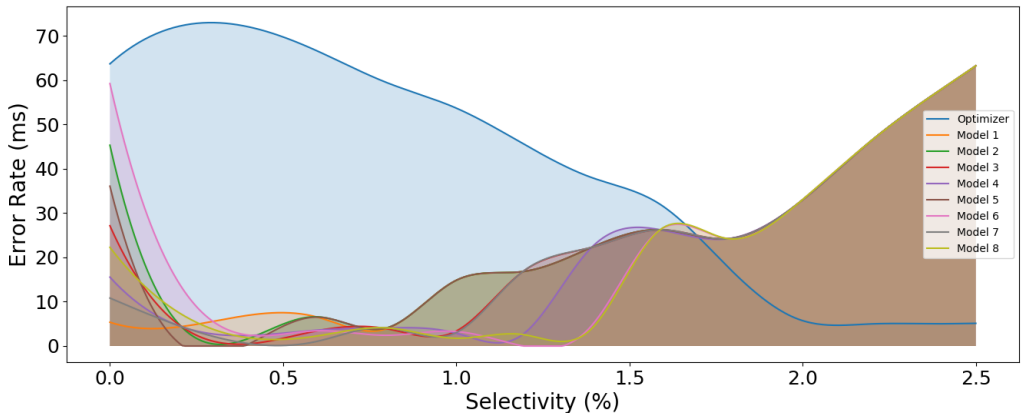
(b) Model 7.

Figure 4.10: `PART` error rate by query.

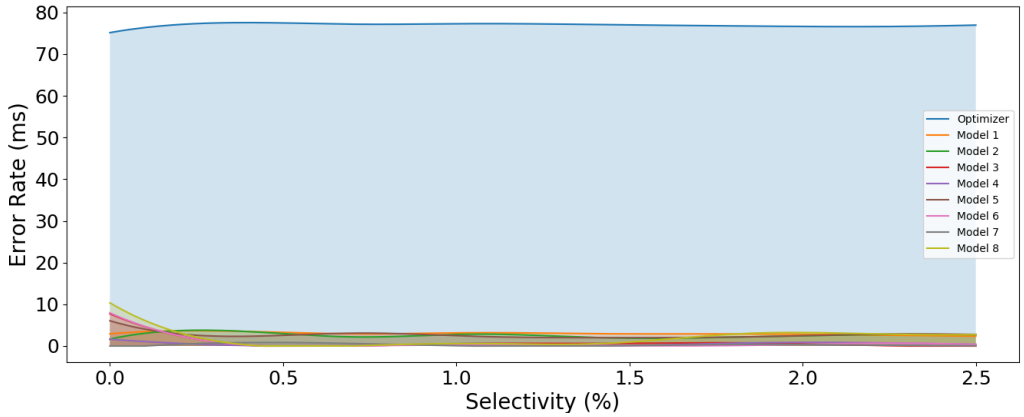
```
SELECT count(P_PARTKEY) FROM part
        WHERE P_PARTKEY < [100000-2000000]
        AND P_SIZE < 3;
```

Query 31:

```
SELECT count(P_PARTKEY) FROM part
        WHERE P_PARTKEY < 200000
        AND P_RETAILPRICE < [1100-2200]
```



(a) Query 24.



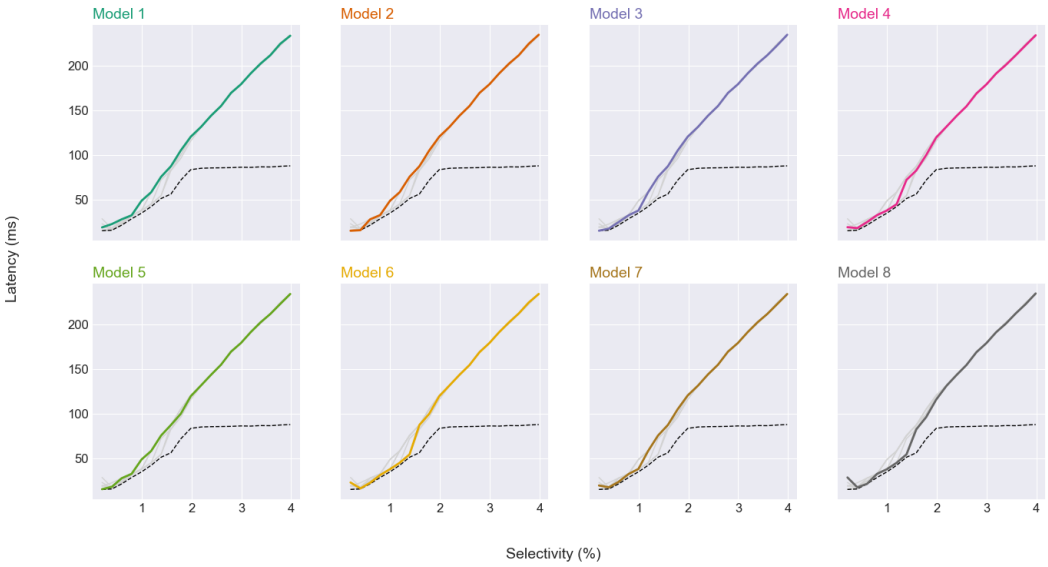
(b) Query 31.

Figure 4.11: `PART` error for all models by query (optimizer in blue).

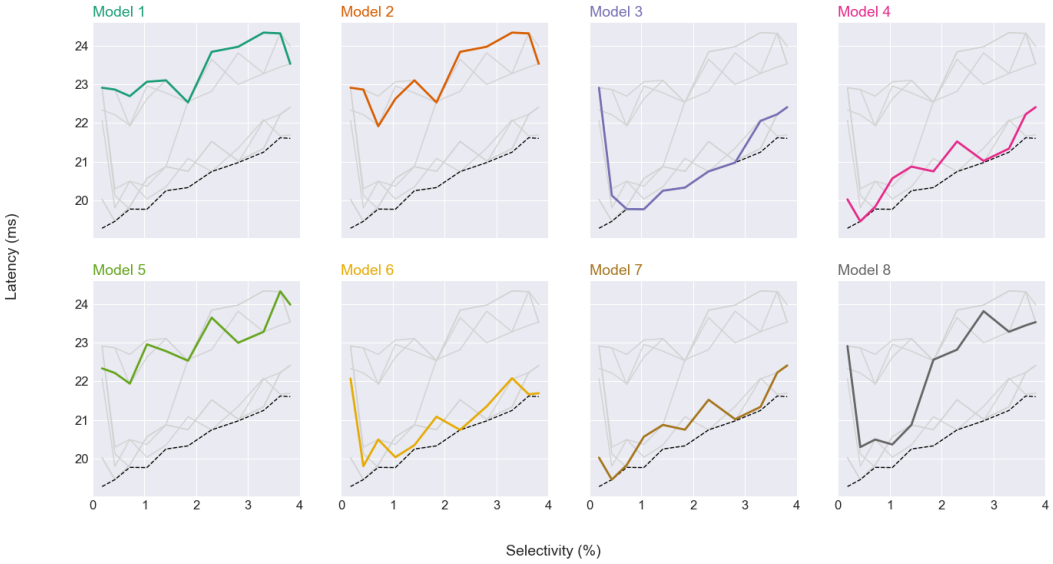
```
AND P_SIZE < [10-20];
```

The models in Figure 4.11a show good performance in the critical decision region (below 2%) but struggle in the higher selectivity range. In reality, that is the best possible place for error, since we are more confident that we should scan anything above the critical decision boundary. In Query 31 (Figure 4.11b), we see that all models outperform the optimizer.

Comparing the decisions across all models for Query 24, though, we get a very different picture. Even though the error rate is low, they all missed the critical decision



(a) Query 24.

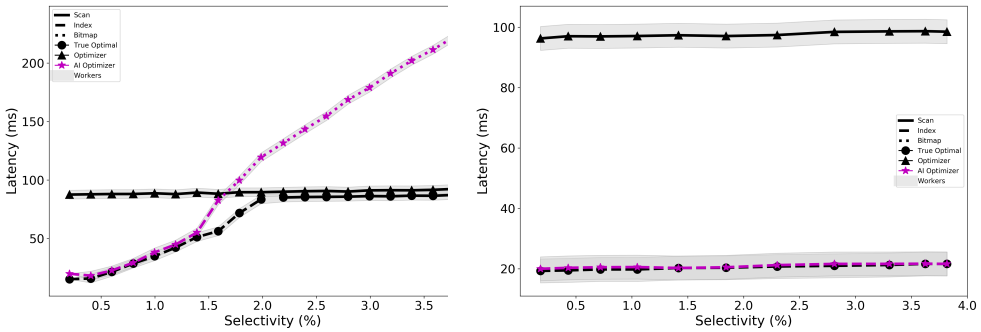


(b) Query 31.

Figure 4.12: `PART` model decisions compared to ideal.

point (Figure 4.12a). For Query 31, we see acceptable performance for Models 3, 4, 6, and 7.

Looking deeper at the decisions of the winning model, we see that Model 7-512



(a) Query 24. (b) Query 31.

Figure 4.13: `PART` Model 7-512 ("Learned Optimizer") decision detail.

missed the mark on Query 24 because it switched to a bitmap scan instead of a table scan. The optimizer made the right decision (Figure 4.13a). Figure 4.13b shows that for Query 31, we see optimal results (while the optimizer chose the wrong path).

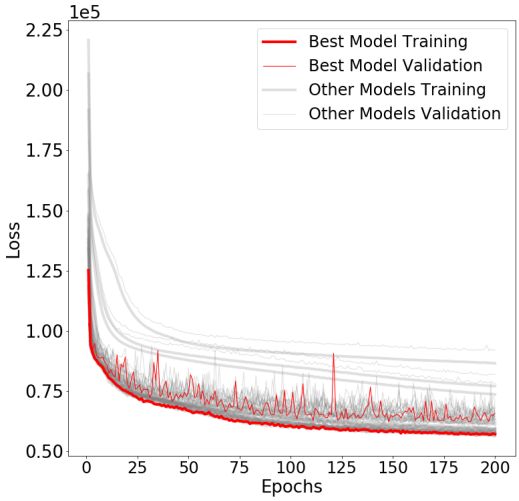
Of the 17 queries on `PART` table, we find 4 incorrect path selections (Queries 24, 27, 28, 32), albeit with minimal impact, while the optimizer misses 16 of them. Of interest is that the optimal paths for `PART` table usually stuck with one type of plan or another (a bitmap, or a scan) and rarely shifted direction. The AI optimizer learned that it should likewise stick to some decision and resist change. However, it was somewhat resilient in cases where change was necessary (Queries 24, 27, 28, 33).

4.2.3 PARTSUPP

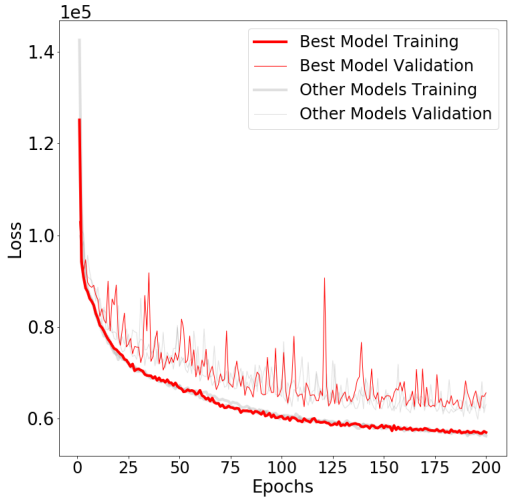
`PARTSUPP` was the largest table with the most complicated queries (50 queries with 4 indexed columns).

Loss

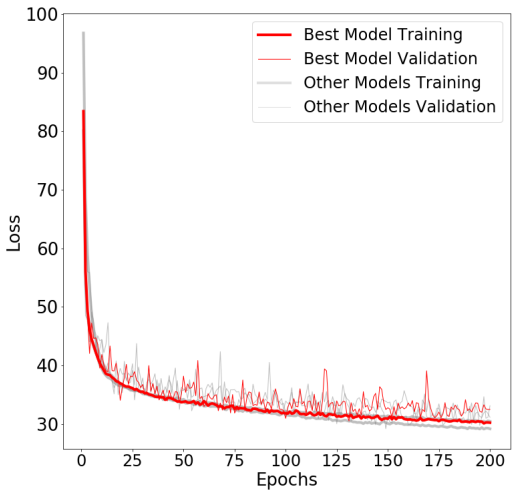
Loss follows the shape we would like to see during training (Figure 4.14a), with constantly improving training and validation loss as epochs progress (showing predictive power). A closer look at the best models (2-512 and 6-256) indicates



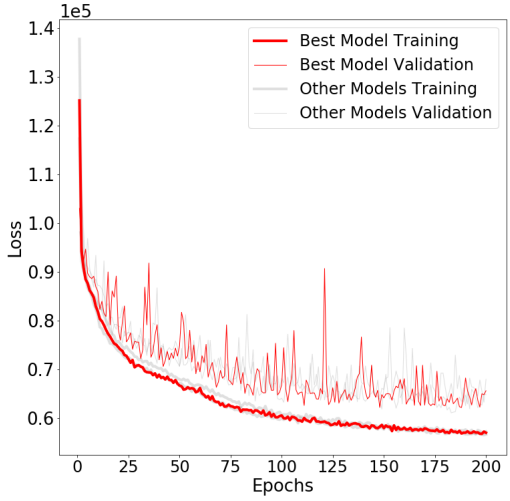
(a) MSE on all models.



(b) MSE on 4 best models.



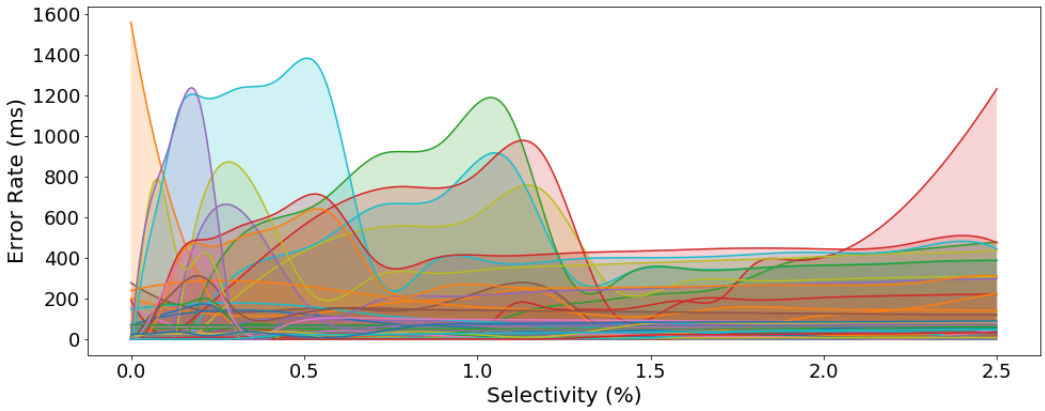
(c) MAPE on best 4 models.



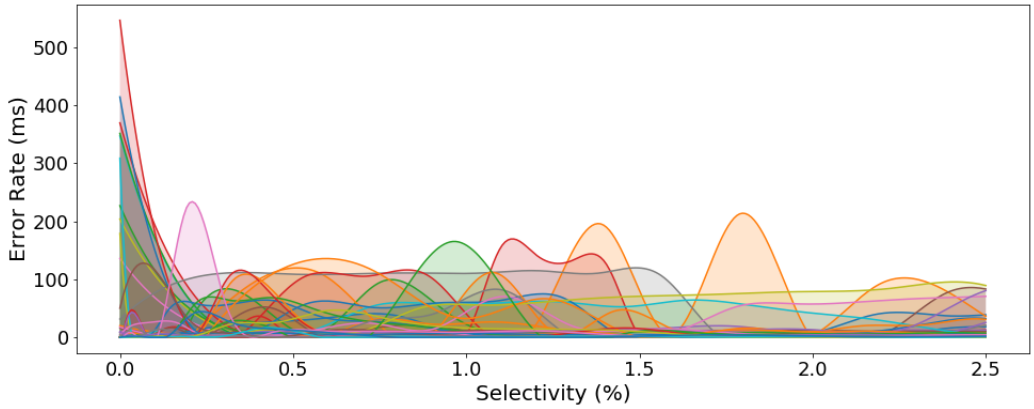
(d) MSE on all variations of Model 6.

Figure 4.14: `PARTSUPP` table models.

similarities, but Model 6-256 edges out (Figure 4.14b). They take about the same time to train; approximately 7 seconds per epoch (2-512) versus 6 seconds (6-256). MAPE (Figure 4.14c) supports the selection of Model 6-256. The Model 6 family also appears to be dead-even (Figure 4.14d).



(a) Optimizer.



(b) Model 6.

Figure 4.15: `PARTSUPP` error rate by query.

Error Rate

Comparing the performance of Model 6-256 on all of the queries on the `PARTSUPP` table, we see excellent performance (Figure 4.15b) in comparison to the much higher latency of the optimizer (Figure 4.15a).

Decision Path

We review the decisions on Query 67 and 74 in a bit more depth.

Query 67:

```
SELECT count(PS_PARTKEY) FROM partsupp
        WHERE PS_PARTKEY < 660000
        AND PS_SUPPKEY < [1650-32981]
        AND PS_SUPPLYCOST < [17-321];
```

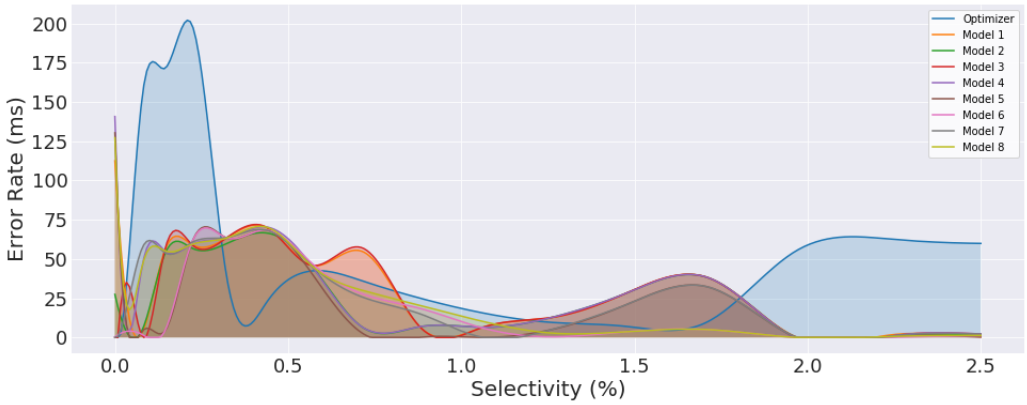
Query 74:

```
SELECT count(PS_PARTKEY) FROM partsupp
        WHERE PS_SUPPKEY < 33000
        AND PS_PARTKEY < [33000-659981]
        AND PS_SUPPLYCOST < [17-321];
```

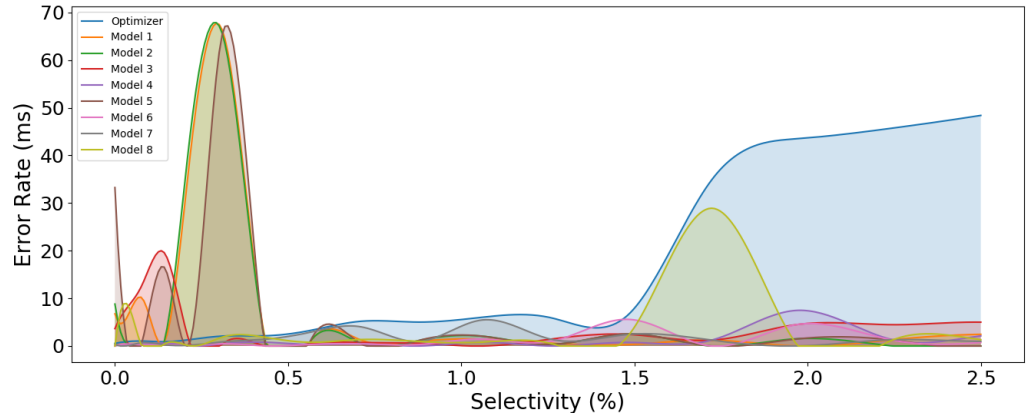
On most models, Query 67 (Figure 4.16a) performs well below the optimizer (in blue). Note that Model 2 (in green) struggles a bit in the critical decision point area below 1%. Query 74 tells a similar story (Figure 4.16b). Most models beat the optimizer, except for Model 1 and 2, which show an error spike (indicating a poor decision) at about .03% selectivity. Note that Model 2 failed at these two critical points, while Model 6 shows minimal error. Recall that they both showed the same score in training and validation loss earlier. We revisit this finding in the following chapter.

A look at how every model reacted to Query 67 (Figure 4.17a) tells us that no model did perfectly, but clearly, Model 6 made the fewest mistakes. In Figure 4.18a we see the cause for confusion were the few points before the 0.5% selectivity area. Still, it outperformed the optimizer, whereas the other models did not.

Turning our attention to Query 74 (Figure 4.17b), we see that Models 4 through 6 were the only models that perfectly followed the ideal. Interestingly, all of the models made the critical decision at 1.5% selectivity to multithread (as we discussed in Chapter2), while the optimizer missed that opportunity (Figure 4.18b).



(a) Query 67



(b) Query 74

Figure 4.16: `PARTSUPP` error for all models by query (optimizer in blue).

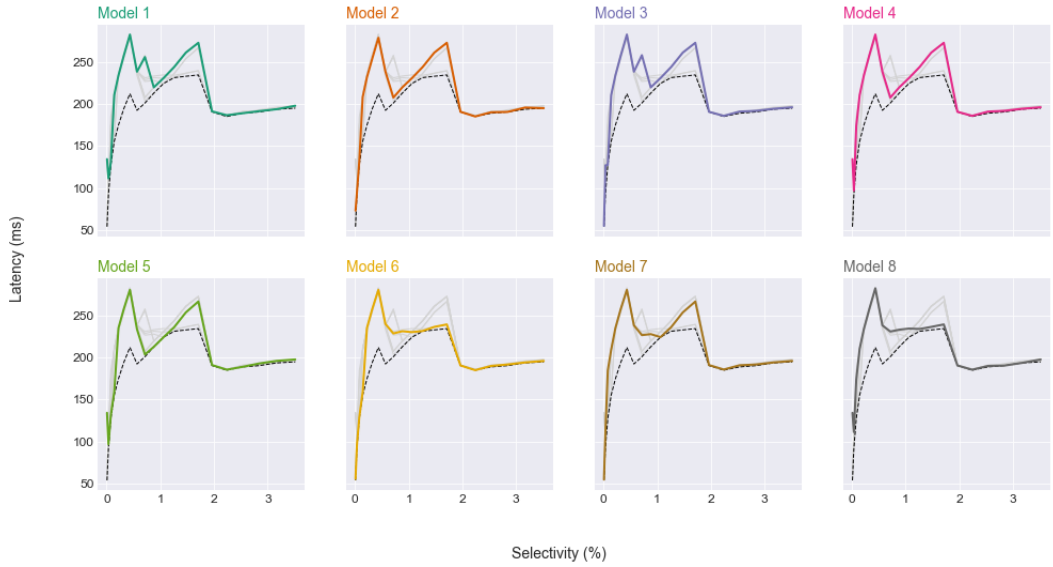
Of the 50 queries on this table, the winning model got about half (26) entirely correct, with minor mistakes in the others. It still outperformed the query optimizer.

4.2.4 ORDERS

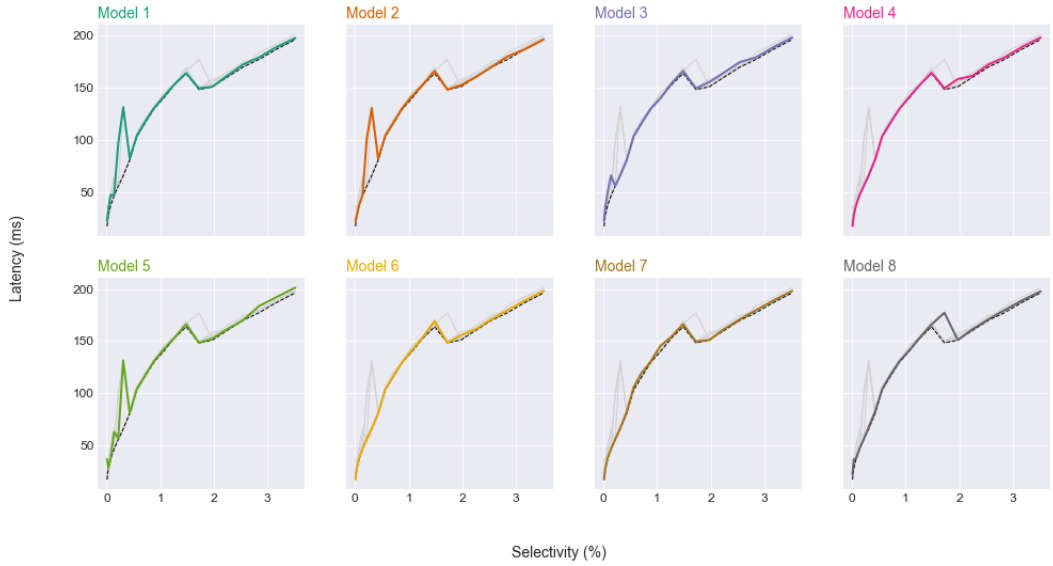
Many models accurately predicted the `ORDERS` table, making it a close race.

Loss

Most models had very little loss, both from an MSE (Figure 4.19a) and MAPE (Figure 4.19b) perspective. The winning model (Model 2-512) had a perfect decision



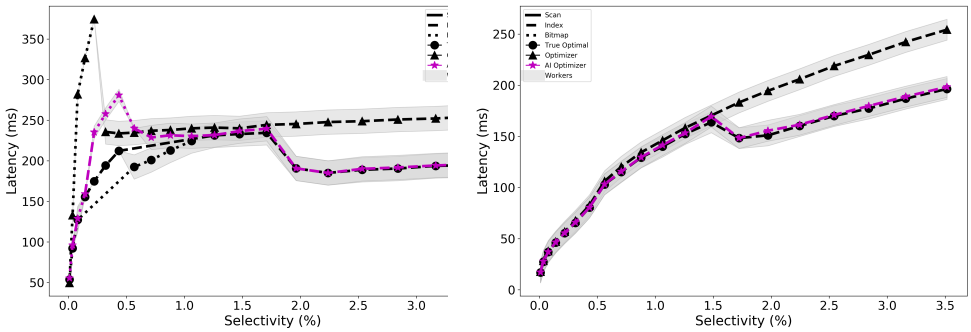
(a) Query 67.



(b) Query 74.

Figure 4.17: `PARTSUPP` model decisions compared to ideal.

record, edging out Model 8-256 by a single mistake (in Query 87). Model 2-512 took 7 seconds per epoch to train while Query 8-256 took 5 seconds. Both acceptable, considering Model 8-256's mistake cost 85 milliseconds of latency but saves 200 sec-



(a) Query 67. (b) Query 74.

Figure 4.18: PARTSUPP Model 6-256 ("Learned Optimizer") decision detai.l

onds of training. Additionally, the charts in Figure 4.19 show that training completed quickly at 50 epochs. Figure 4.19c compares the model to other members of its family. Unlike other tables, here, we see clear victory amongst its family members.

Error Rate

Error rates across all 14 queries on this table (Figure fig:orders-errorrate), indicate that all models significantly outperform the optimizer by order of magnitude.

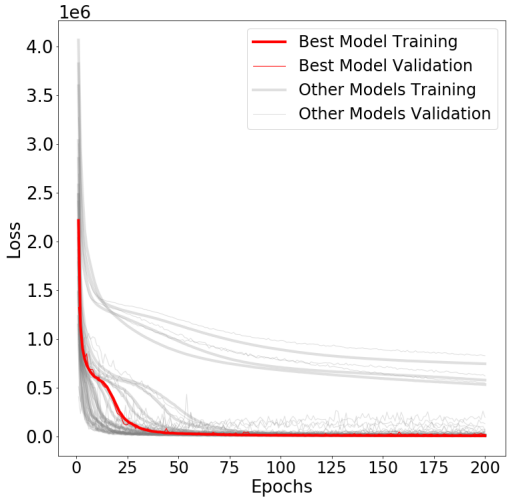
Decision Path

To understand the decisions made, we consider Queries 90 and 93. Recall that Query 90 was discussed in Chapter 2 and used O_TOTALPRICE and O_CUSTKEY varying only O_CUSTKEY.

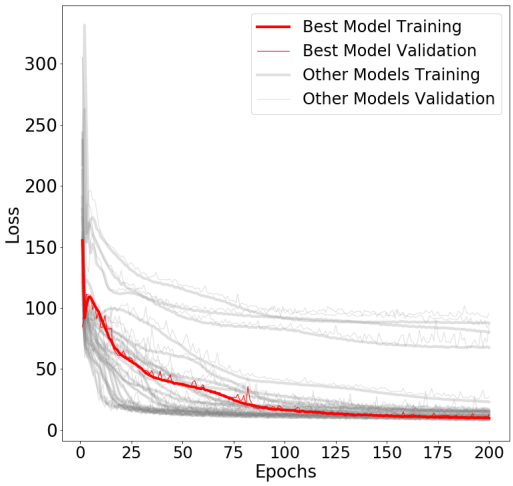
Query 93:

```
SELECT count(O_ORDERKEY) FROM orders
        WHERE O_CUSTKEY < 299999
        AND O_ORDERKEY < [600000-11999981];
```

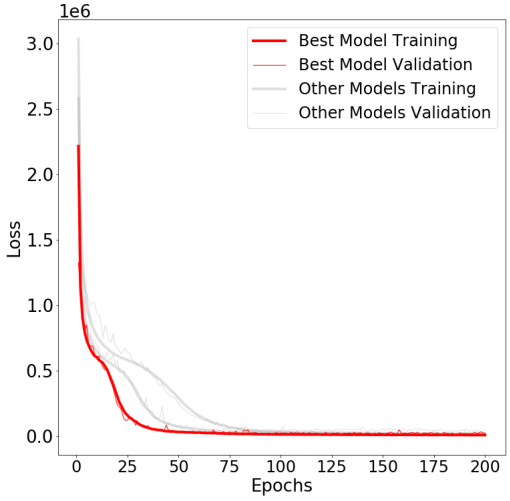
Figure 4.21a (Query 90) shows significant early confusion in Models 1 and 7 (costing approximately 1 second). The decisions made by Models 2 through 6 and 8



(a) MSE on all models.



(b) MAPE on all models.

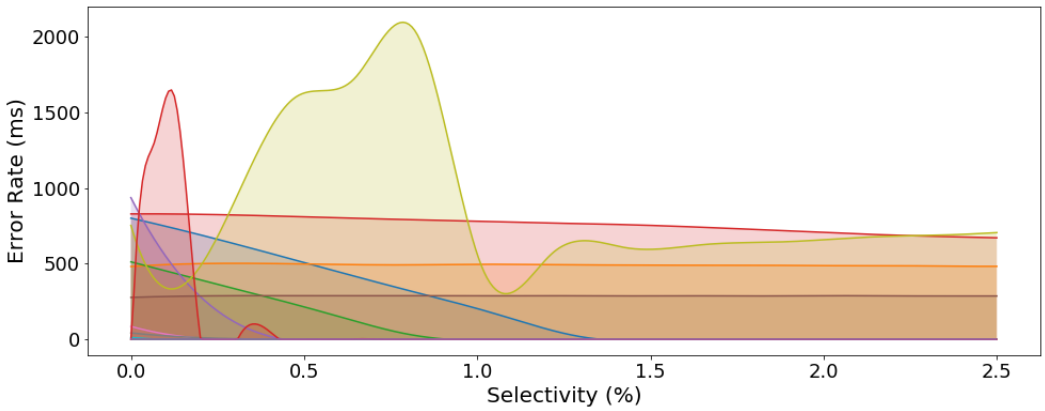


(c) MSE on all variations of Model 2.

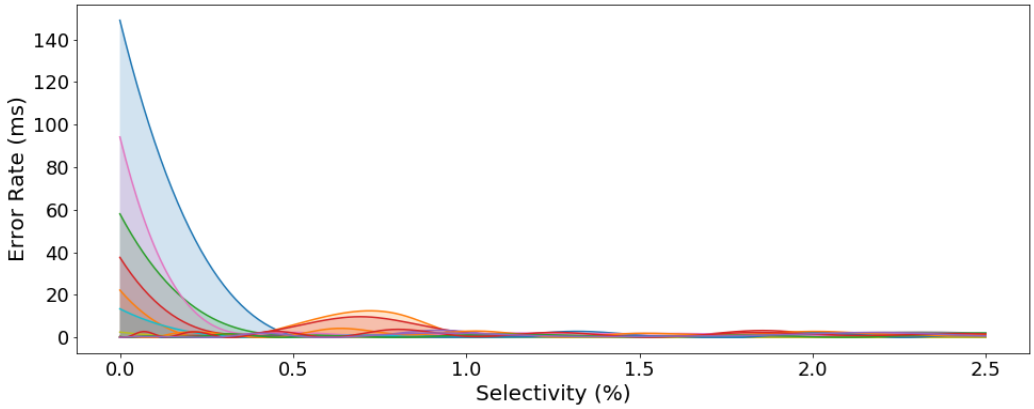
Figure 4.19: `ORDERS` table models.

follow the ideal (Figure 4.22a). Looking closely at the decisions, we see Model 2 (and the optimizer) were at ideal (Figure 4.23a).

Models 3 and 6 show early confusion on Query 93 (Figure 4.21b), while Models 1, 2, and 7 follow the ideal (Figure 4.22b). We find a minor mistake with the first



(a) Optimizer.



(b) Model 2.

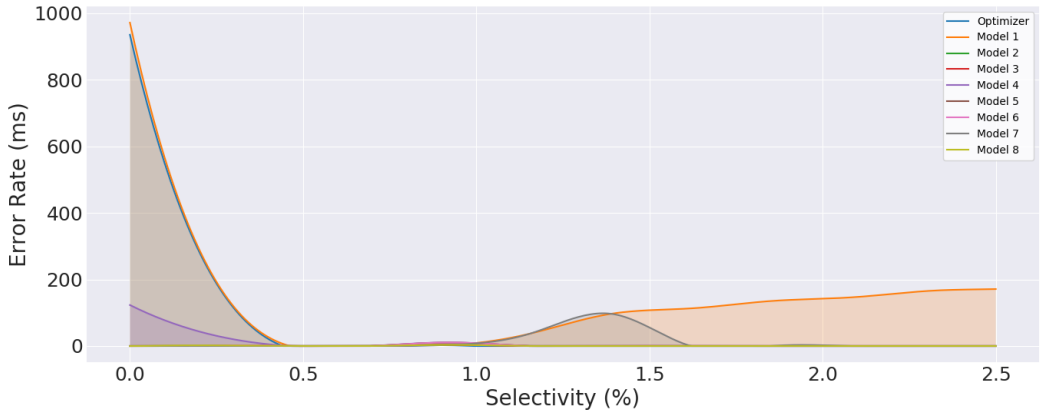
Figure 4.20: `ORDERS` error rate by query.

decision in Model 8. Again, we see the decisions of Model 2-512 follow the ideal, while the optimizer made a single mistake in the first decision point (Figure 4.23b).

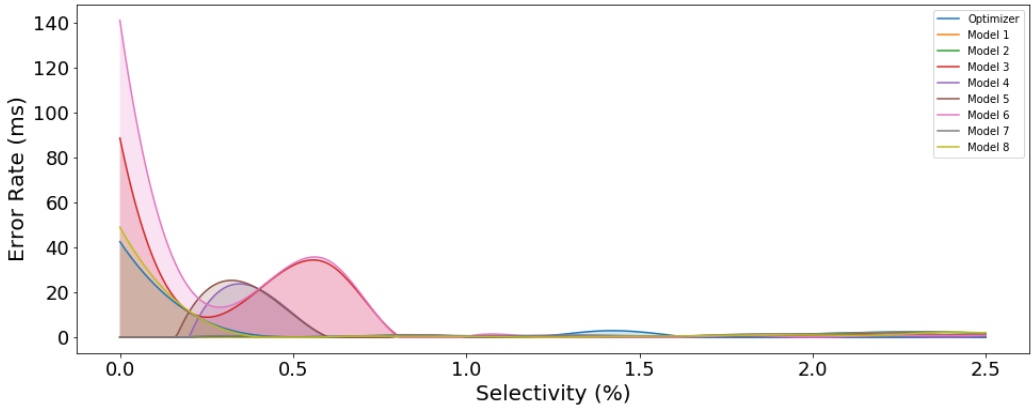
The best model for `ORDERS` is Model 2-512, with a perfect prediction record compared to the optimizer's 8 out of 14.

4.3. Monolithic Model

The monolithic model has the challenge of being adept at predicting a broader range of queries. To that end, additional features and a more extensive network are



(a) Query 90.



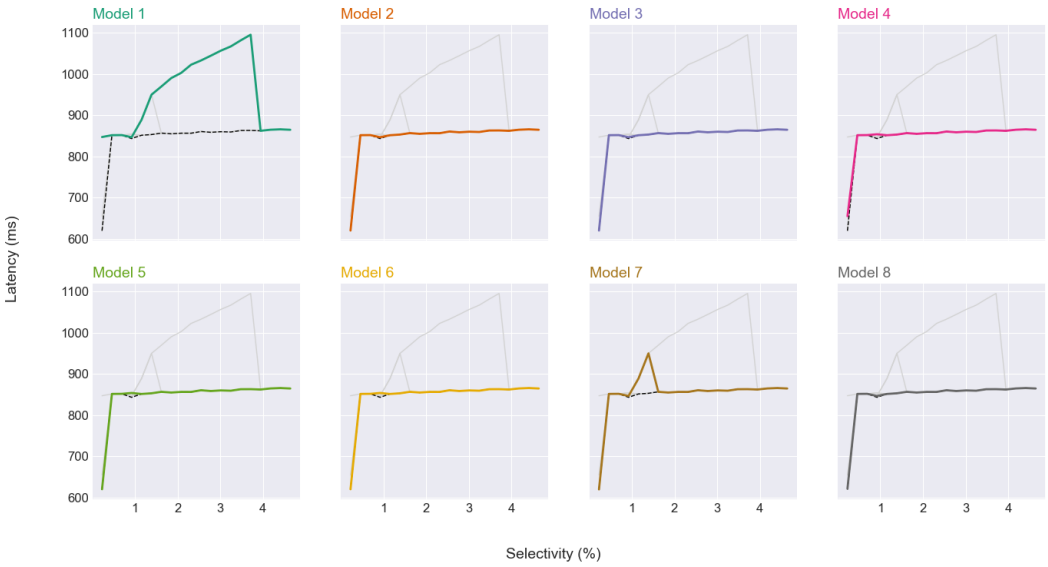
(b) Query 93.

Figure 4.21: `ORDERS` error for all models by query (optimizer in blue).

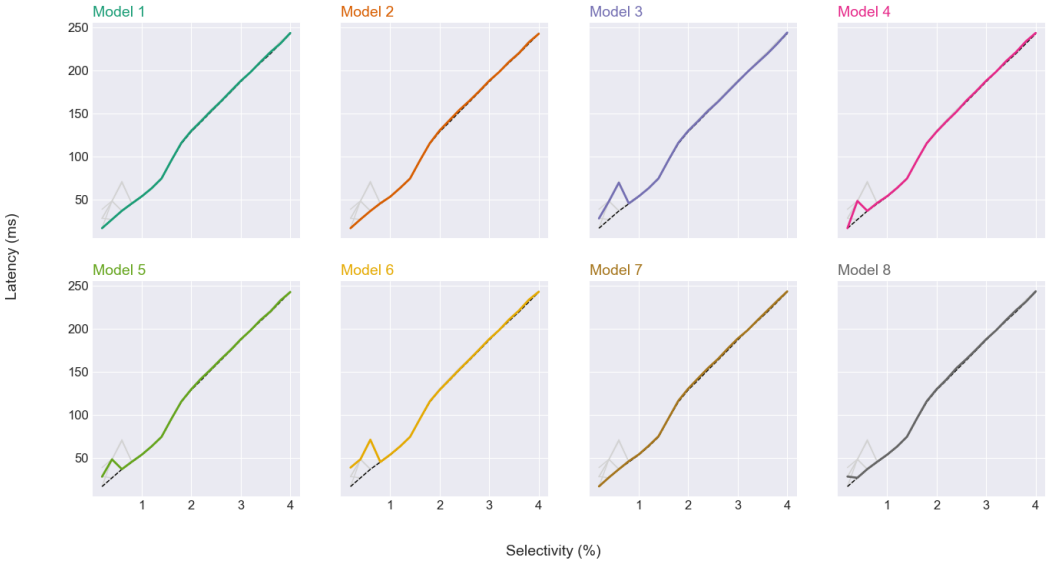
required. How does the larger network perform?

The architectures remain the same except for changes in scale factor, as discussed in Chapter 2. The monolithic model variations are:

- Scale factor 2: 1024 (on all models)
- Scale factor 1: 512 (on all models)
- Scale factor 0.5: 256 (omitted for model 8)



(a) Query 90

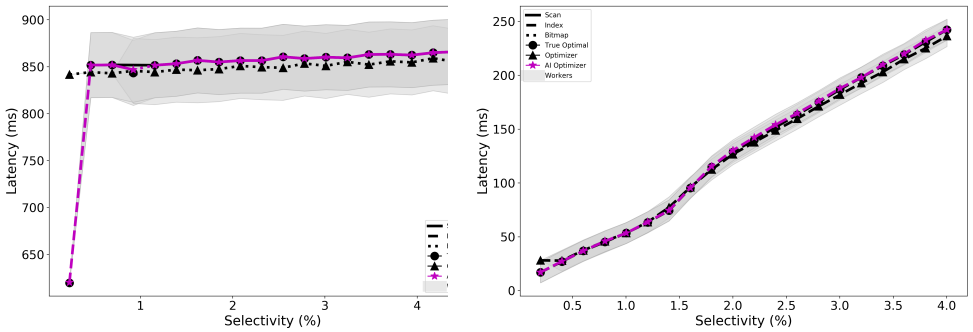


(b) Query 93

Figure 4.22: `ORDERS` model decisions compared to ideal.

4.3.1 Loss

We begin by comparing loss across the entire model range (Figure 4.24a), highlighting Model 2-256. Overall, the error is quite high with validation error wildly



(a) Query 90. (b) Query 93.

Figure 4.23: `ORDERS` Model 2-512 ("Learned Optimizer") decision detail.

out of control on many models. Model 2-256 is the best performing model, settling on $1,286,553ms^2$ MSE loss (approximately $1,134ms$). MAPE does not fare much better at 375%. Model 2-256 does outperform the rest of its family (Figure 4.24c). However, that does not tell us how the model fared in decision making.

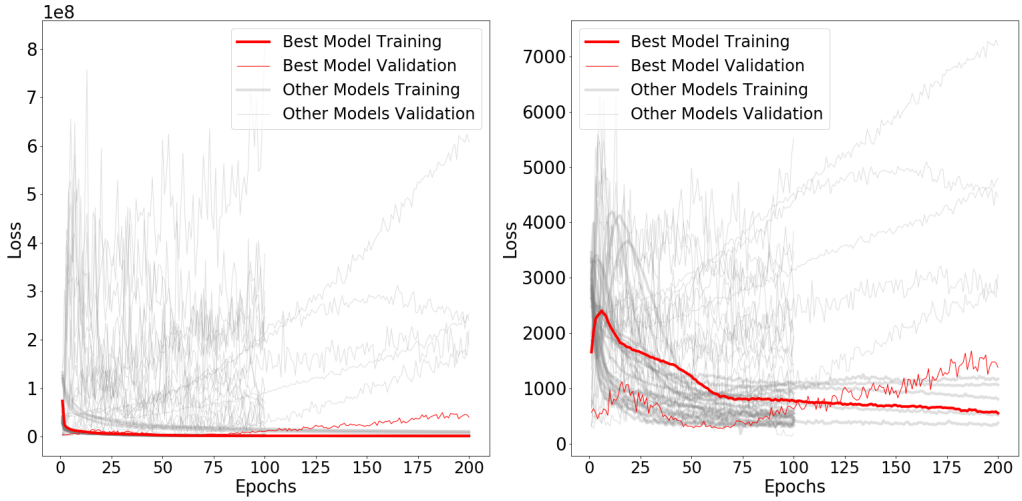
4.3.2 Error Rate

The optimizer shows a very high error rate against all queries, especially those in the critical decision range below 1.5% selectivity (Figure 4.25a). Most models outperform the optimizer (blue line), and a few have trouble after 2% selectivity (Figure 4.25b).

4.3.3 Decision Path

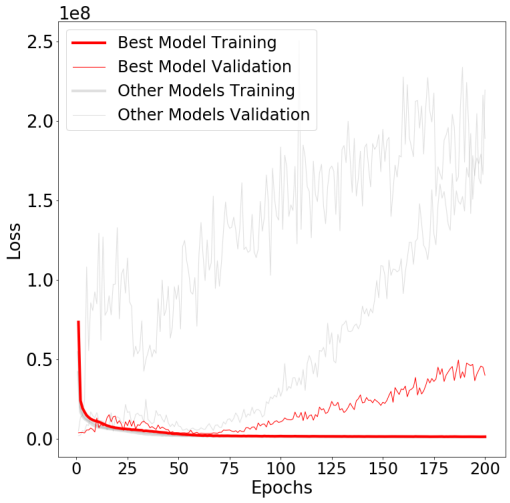
We see that Models 1, 2, 4, 6, and 7 struggle with Query 3, and Model 1 perfectly follows the optimizer (Figure 4.26a). Model 8 is the only model that found the correct decision point (Figure 4.26b).

Query 13 fared slightly better with Models 1, 2, 3, and 7 showing high error in the critical range (Figure 4.27a). The decisions of Models 5, 6, and especially 8



(a) MSE on all models.

(b) MAPE on all models.

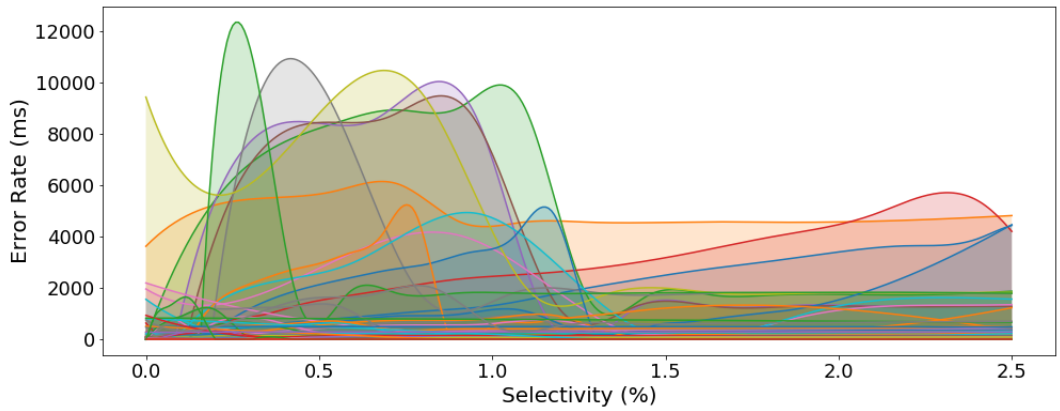


(c) MSE on all variations of Model 2.

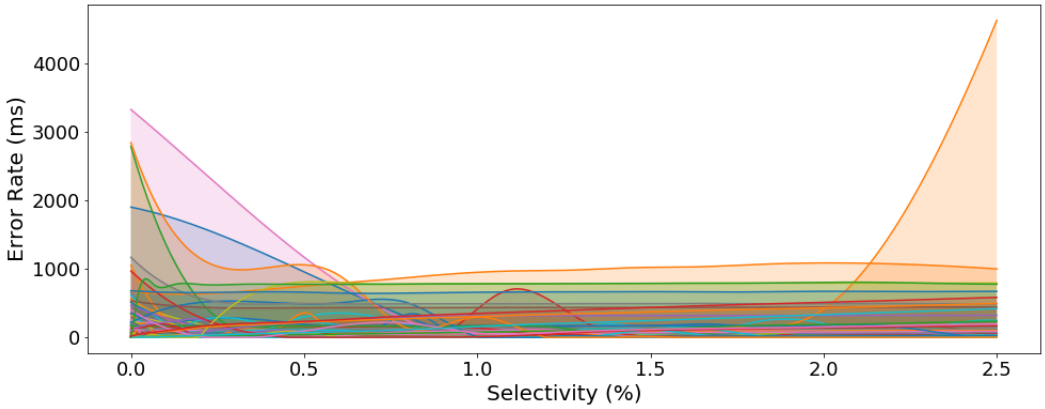
Figure 4.24: Error rate on monolithic models.

performed well (Figure 4.27b).

Figure 4.28a indicates that the error for Query 24 is very low on most models. The optimizer (blue) errors cost $75ms$ in the lower selectivity range, on average. Nearly all models outperform the optimizer in that range, but all of them missed the



(a) Optimizer (Query 6 removed).

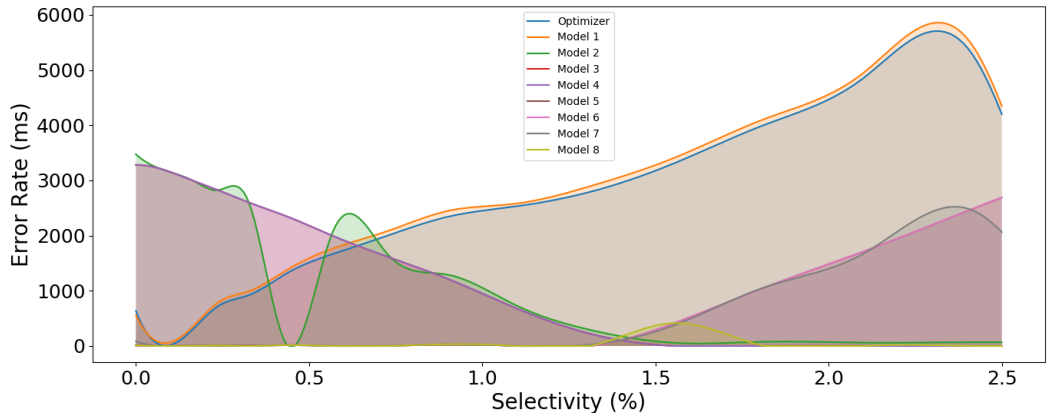


(b) Model 2 (Query 6 removed).

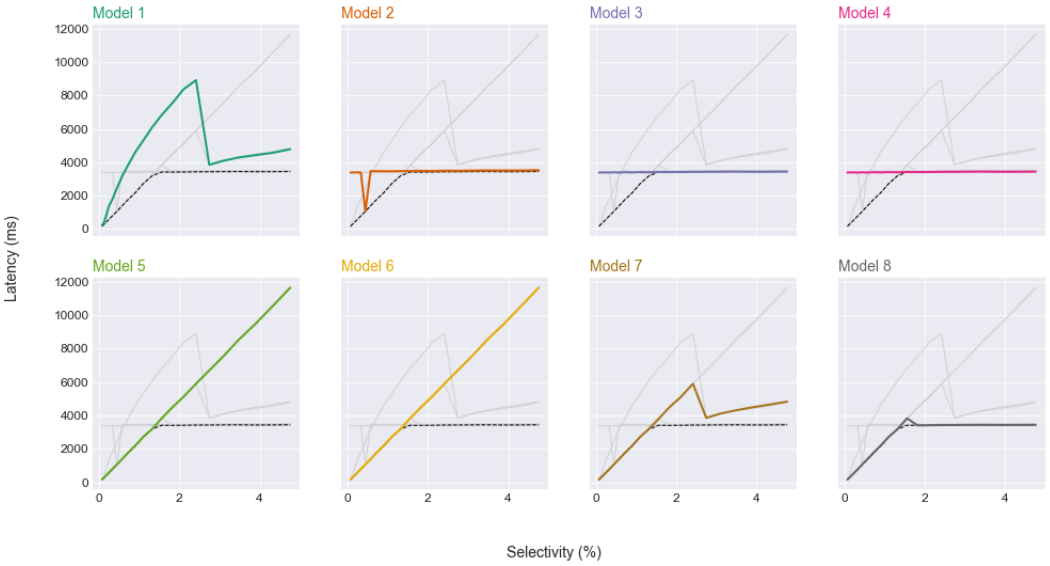
Figure 4.25: Monolithic error rate by query.

critical decision point (Figure 4.28b). Their low error is because the simple query plan does not require change until outside the typical range (Figure 4.35a). Here, the error rate gives us a false sense of reality.

Model 31 is a straightforward plan; it requires no decision change (Figure 4.35b). Unfortunately, all models made errors here (Figure 4.29b). Model 6 performed best on Query 67, but missed the early decisions (Figure 4.30b). Still, it outperformed the optimizer (Figure 4.36a). Models 2, 4, and 6 found ideal performance on Query 74. The spike of Model 3's poor decisions early on (Figure 4.31) caused problems for this



(a) Error across all models (optimizer in blue).

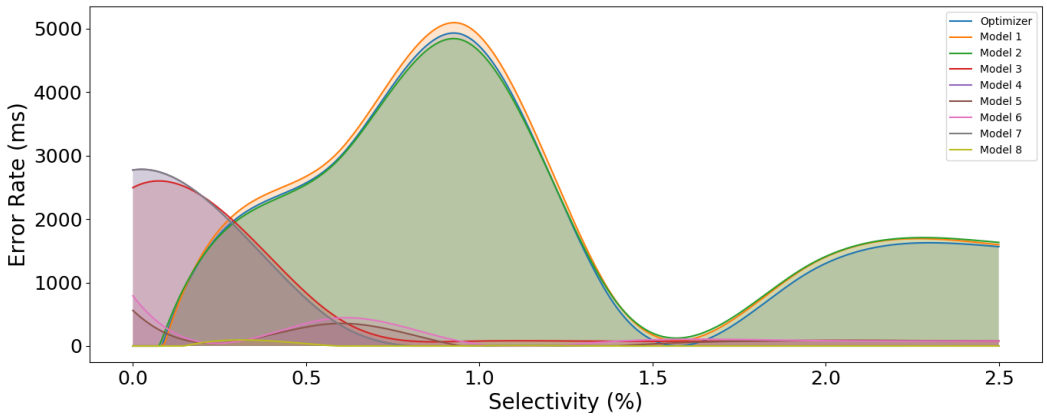


(b) Decisions per model (ideal is dotted).

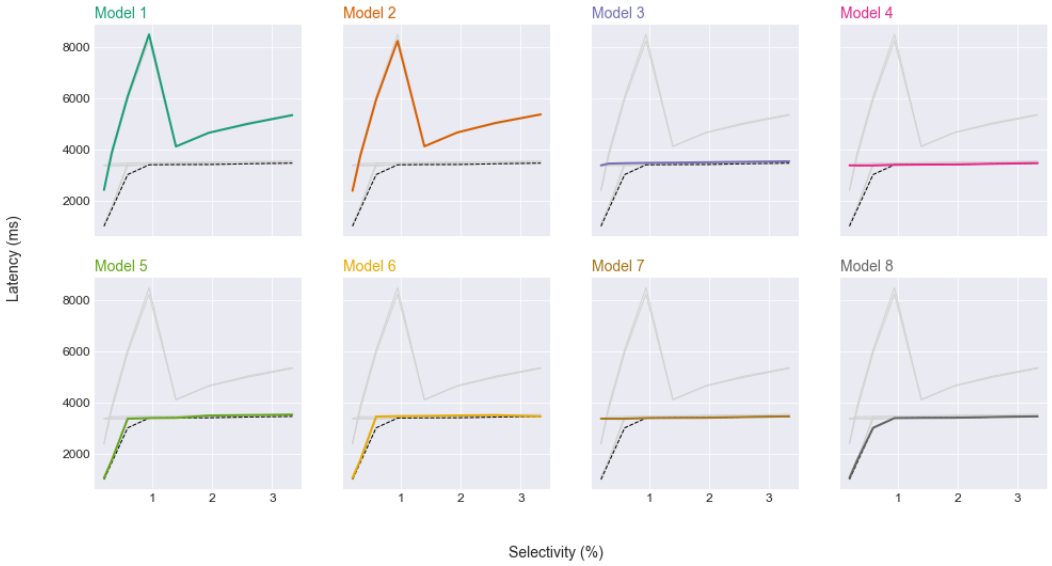
Figure 4.26: Query 6 monolithic model comparison.

query. Only Model 2 correctly surmised the optimal path for Query 90 (Figure 4.32). Models 1, 2, and 3 performed very well on Query 93 (Figure 4.33).

Despite not performing well on some of the examples above, Model 2 performed best overall (with 30 out of 100 ideal paths and many other minor errors). However, other large models performed better on the various tables:



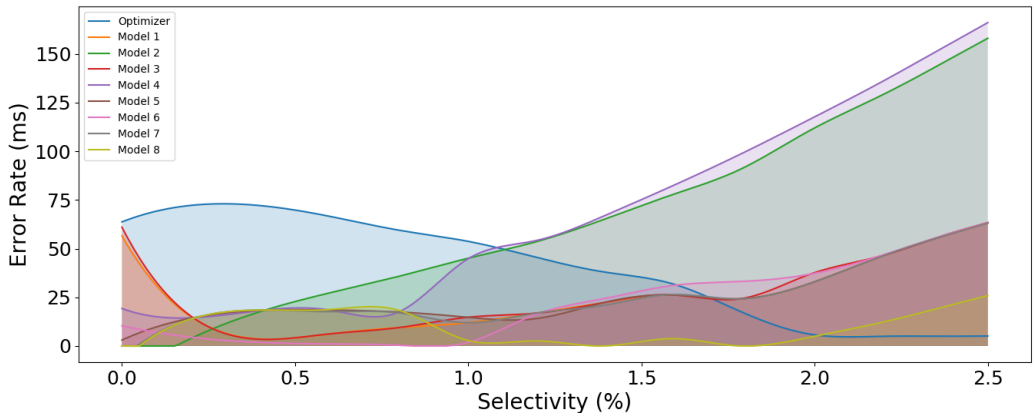
(a) Error across all models (optimizer in blue).



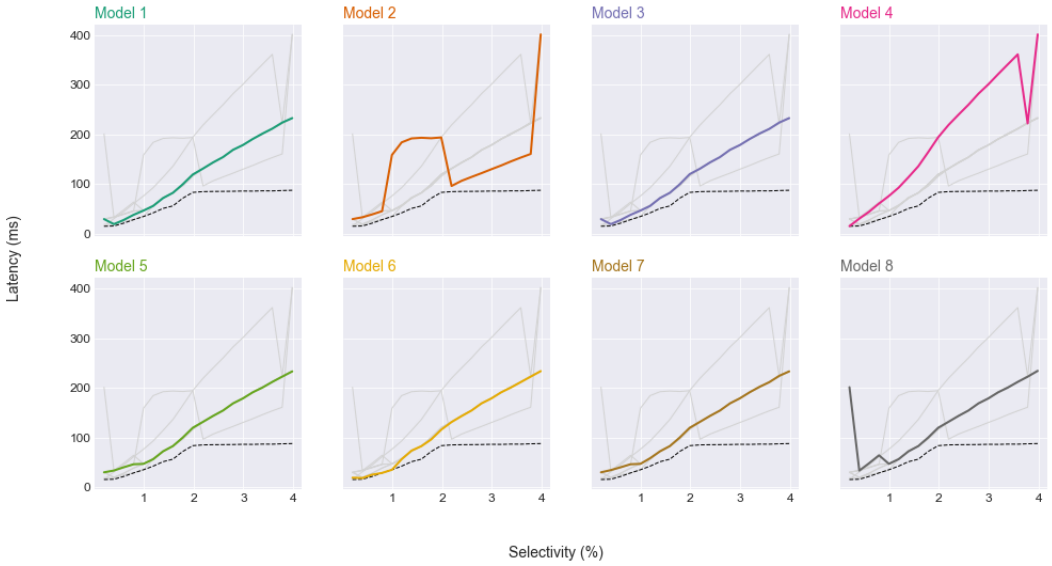
(b) Decisions per model (ideal is dotted).

Figure 4.27: Query 13 monolithic model comparison.

- `LINEITEM`: Model 5-1024
- `PART`: 4-1024
- `PARTSUPP`: 2-1024
- `ORDERS`: 3-1024



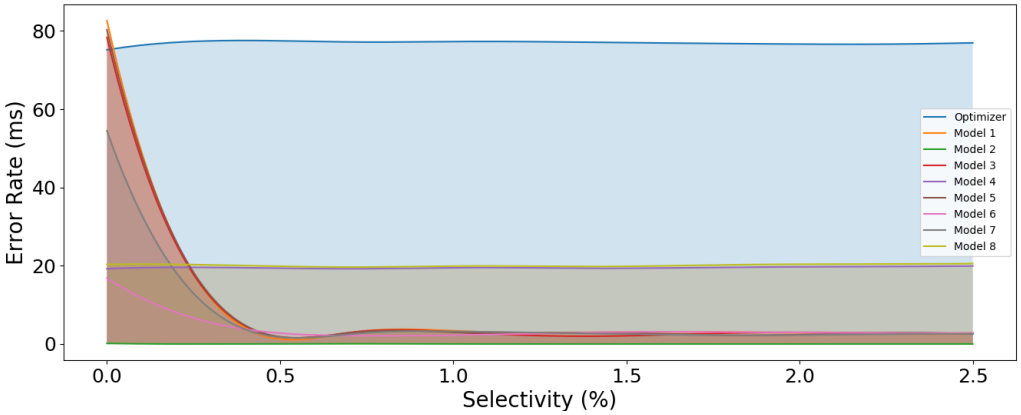
(a) Error across all models (optimizer in blue).



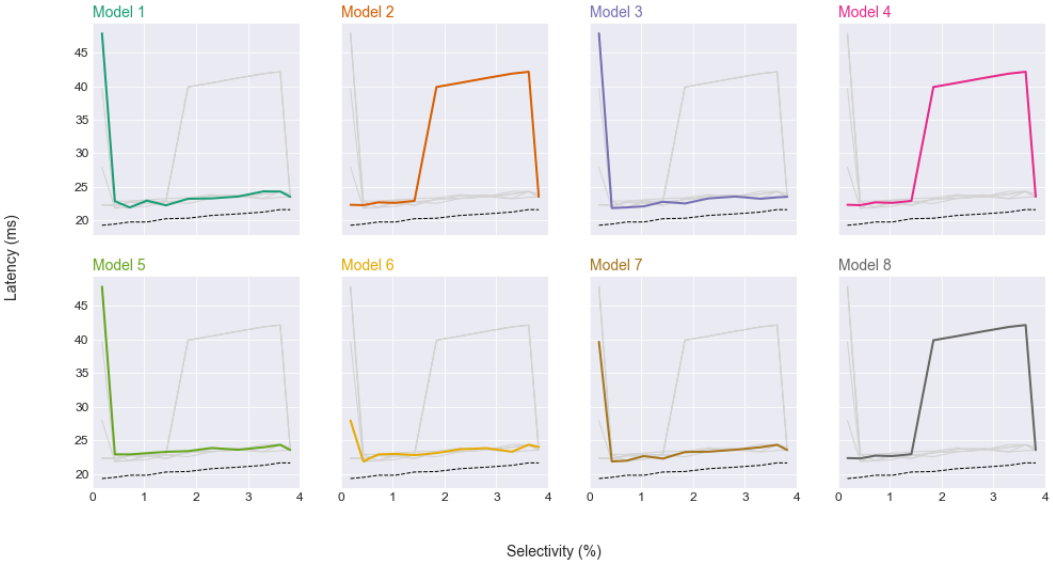
(b) Decisions per model (ideal is dotted).

Figure 4.28: Query 24 monolithic model comparison.

The results of the queries above are not atypical (See Appendix E). How did the model perform against its smaller counterparts?

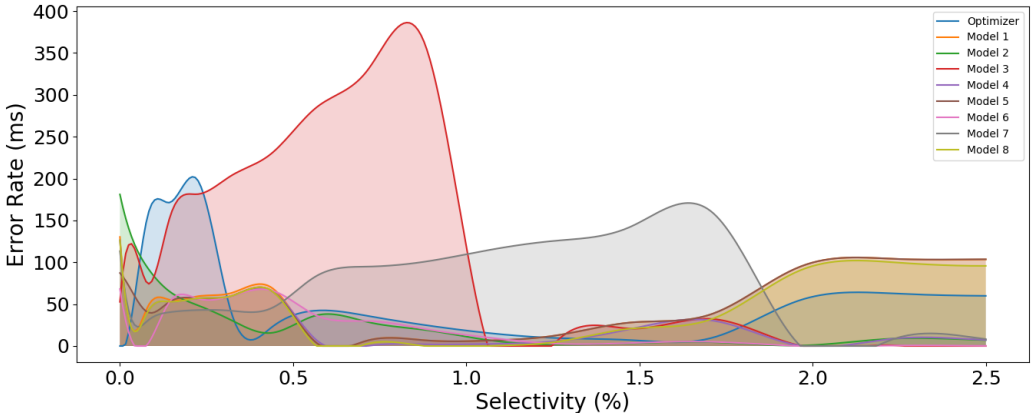


(a) Error across all models (optimizer in blue).

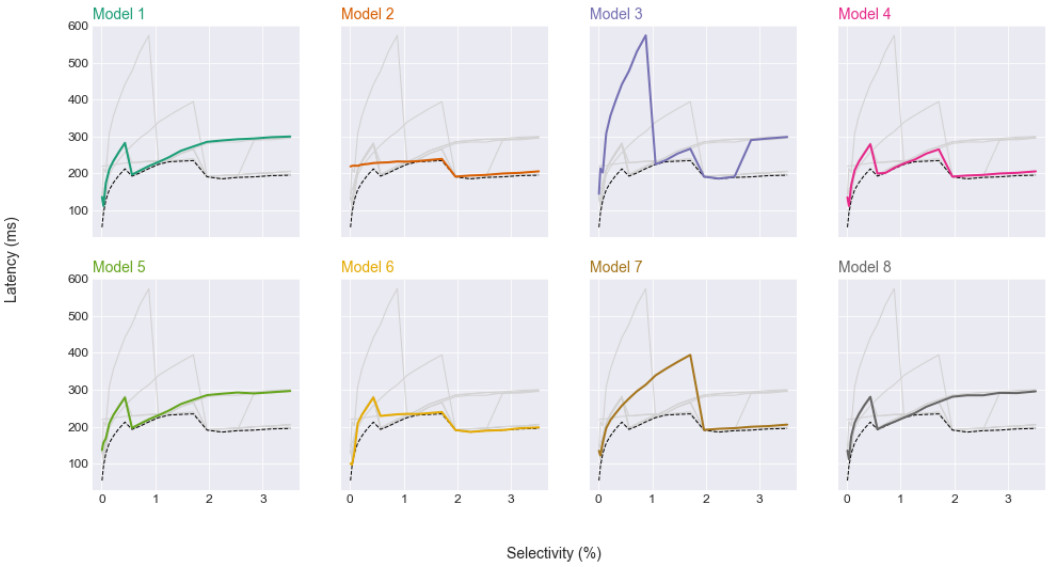


(b) Decisions per model (ideal is dotted).

Figure 4.29: Query 31 monolithic model comparison

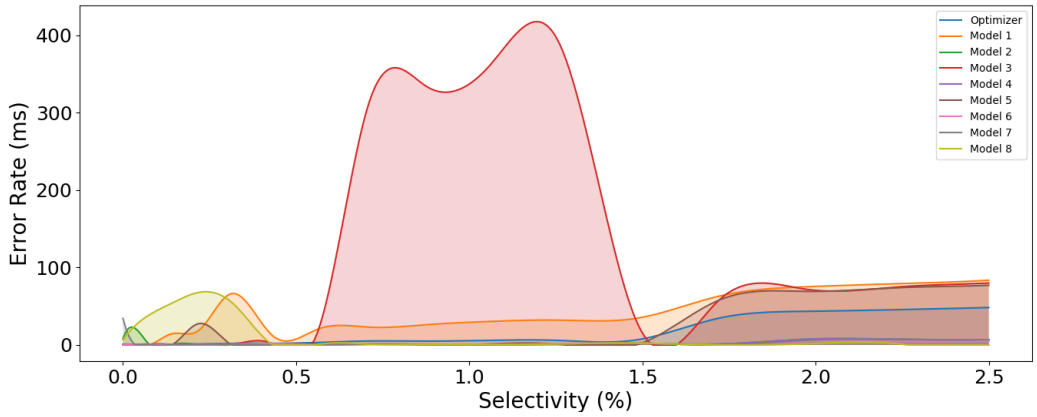


(a) Error across all models (optimizer in blue).

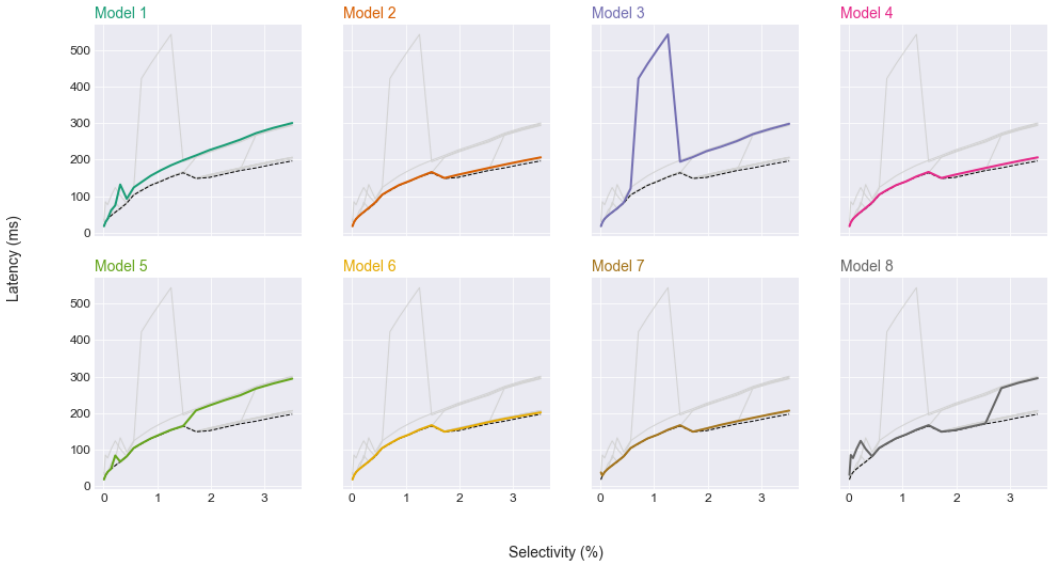


(b) Decisions per model (ideal is dotted).

Figure 4.30: Query 67 monolithic model comparison.



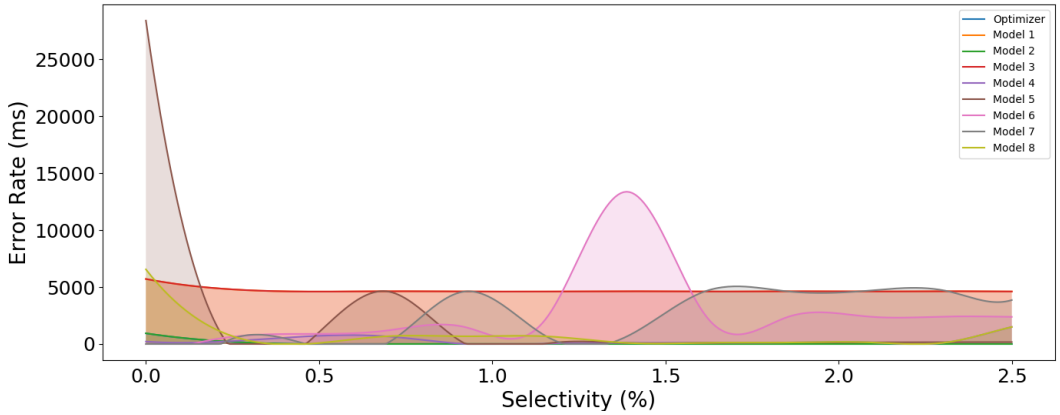
(a) Error across all models (optimizer in blue).



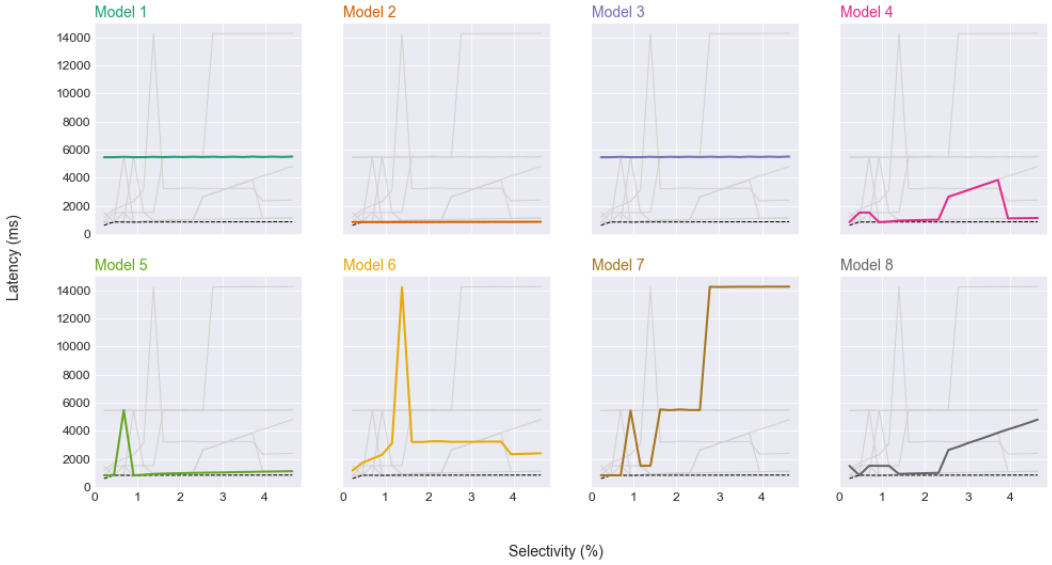
(b) Decisions per model (ideal is dotted

).

Figure 4.31: Query 74 monolithic model comparison.

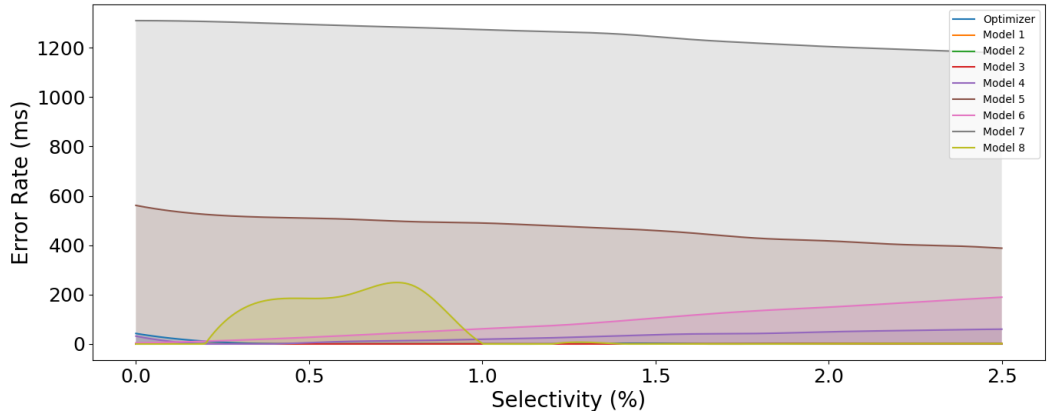


(a) Error across all models (optimizer in blue).

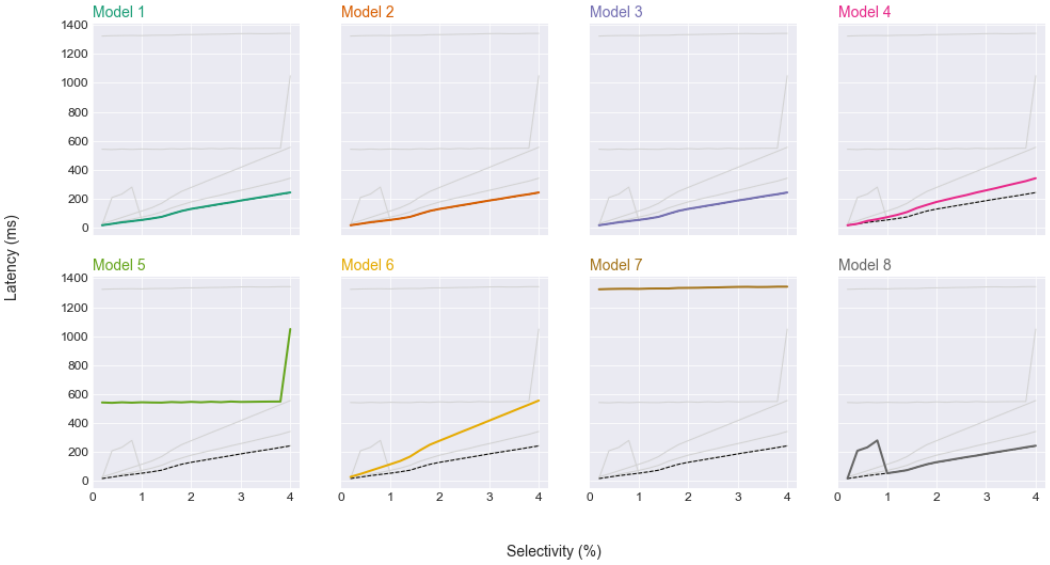


(b) Decisions per model (ideal is dotted).

Figure 4.32: Query 90 monolithic model comparison.

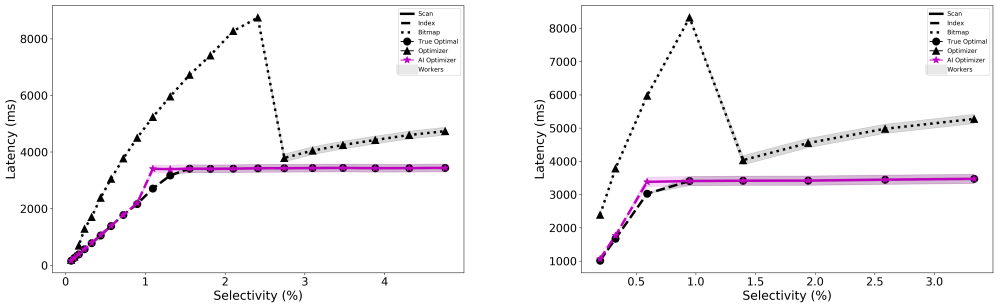


(a) Error across all models (optimizer in blue).



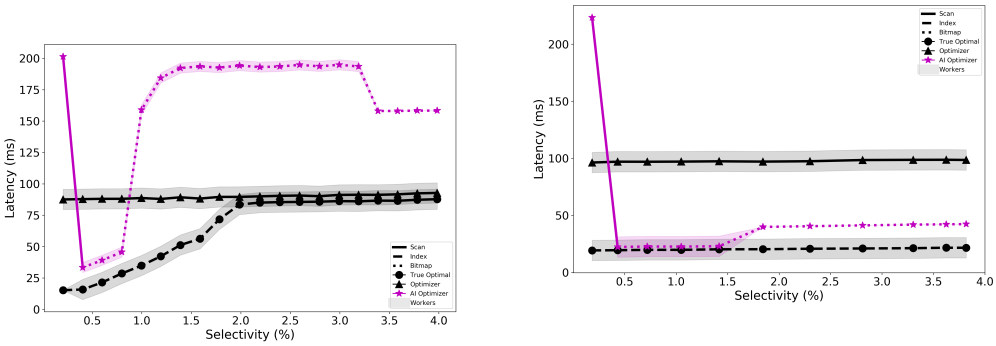
(b) Decisions per model (ideal is dotted).

Figure 4.33: Query 93 monolithic model comparison.



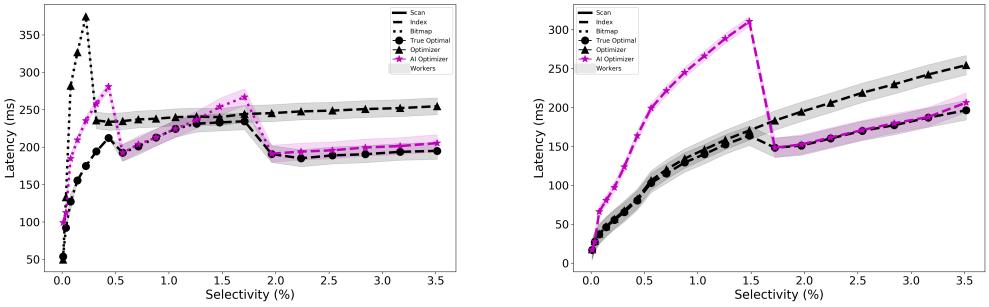
(a) Query 6. (b) Query 13.

Figure 4.34: Monolithic model 2-256 decisions on `LINEITEM`.



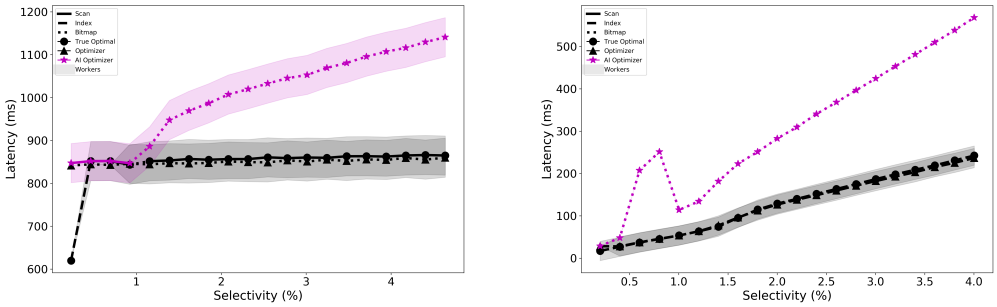
(a) Query 24. (b) Query 31.

Figure 4.35: Monolithic model 2-256 decisions on `PART`.



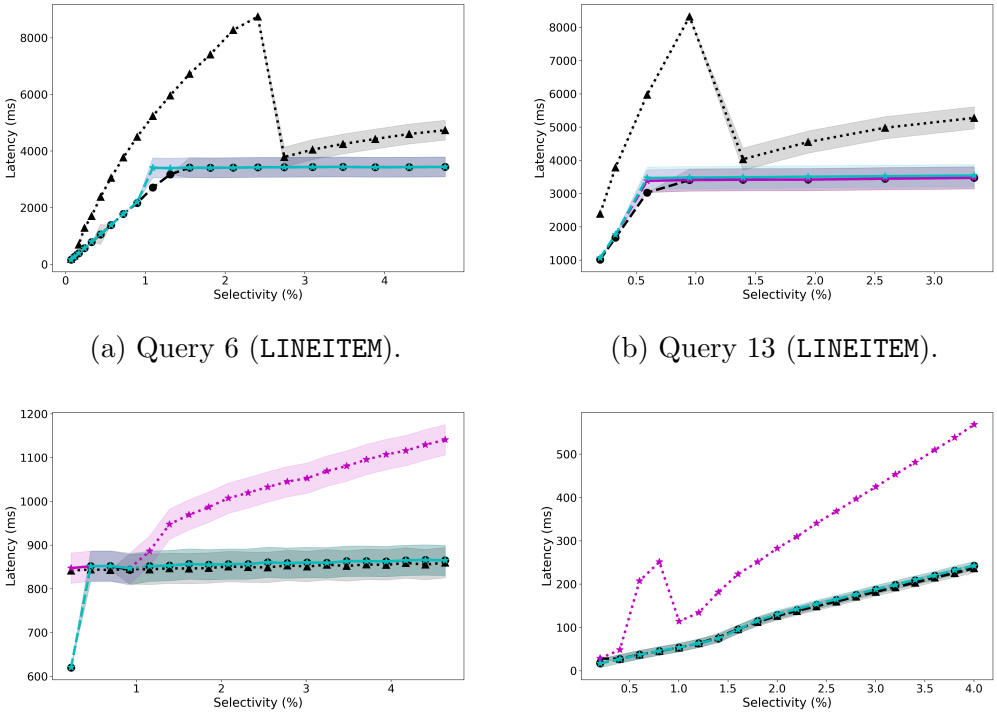
(a) Query 67. (b) Query 74.

Figure 4.36: Monolithic model 2-256 decisions on `PARTSUPP`.



(a) Query 90. (b) Query 93.

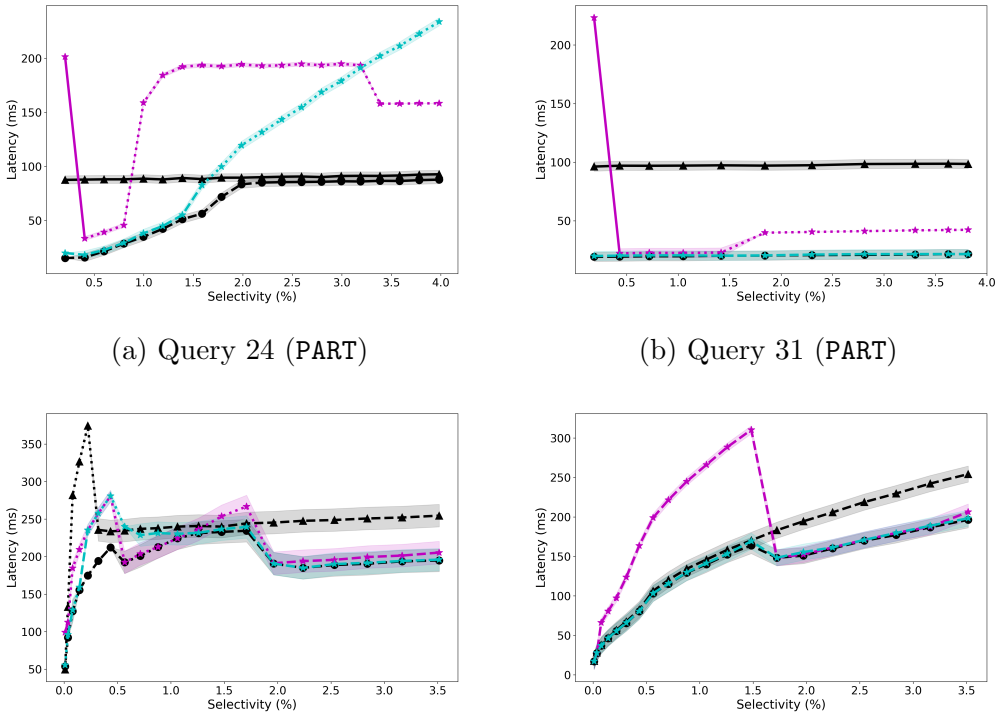
Figure 4.37: Monolithic model 2-256 decisions on `ORDERS`.



(a) Query 6 (`LINEITEM`). (b) Query 13 (`LINEITEM`).

(c) Query 90 (`ORDERS`). (d) Query 93 (`ORDERS`).

Figure 4.38: Monolithic (red) vs specialized (blue) model on `LINEITEM` and `ORDERS`.



(a) Query 24 (PART)

(b) Query 31 (PART)

(c) Query 67 (PARTSUPP)

(d) Query 74 (PARTSUPP)

Figure 4.39: Monolithic (red) vs specialized (blue) model on PART and PARTSUPP.

4.4. Best Approach

We compare the monolithic and specialized models using two approaches; the *predictive power* of the model (how well it made decisions compared to the ideal), and how *expensive* the model was to train.

4.4.1 Predictive Power

We again look at the pairs of representative queries on our four tables and compare the large and small models together. While at times both models struggle (Query 67, Figure 4.39c and Query 24, Figure 4.39a), the smaller model is a better fit.

For Query 6 (Figure 4.38a) and Query 13 (Figure 4.38b)

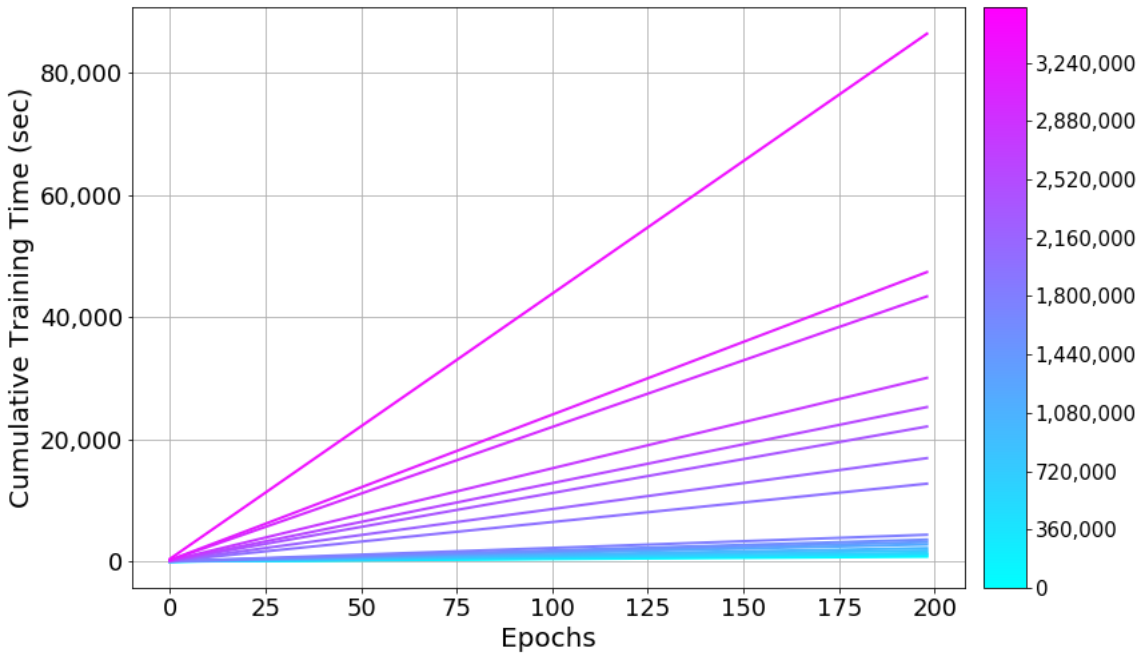


Figure 4.40: Model parameters dramatically increase training time

For table `LINEITEM` (Query 6 and 13) and table `ORDERS` (Queries 90 and 93) we see both models perform well (Figure 4.38). The same is true for large model predictions for other queries (see Appendix E).

We see on table `PARTSUPP` (Queries 67 and 74) and `PART` (Queries 24 and 31) that the smaller models outperform the larger almost every time (Figure 4.39).

4.4.2 Training Time and Resource Consumption

The performance of models is also essential. Training consumes valuable resources. Figure 4.40 shows the relationship between the number of model parameters and time. As the number of parameters to train increases, so does the amount of time (and computing power) needed to accomplish the task.

Chapter 5: Additional Findings

In Chapter 4 we saw that loss did not accurately predict the best model. Additionally, we saw that even computing the error rate for each query is not sufficient. The models that appeared to perform well are not the best. Determining the best model took inspection of the decisions.

5.1. Traditional Methods of Selecting Models Do Not Work

We can further visualize this by comparing the average error of queries for each table between the models (Figure 5.1). We see that the specialized model outperforms the monolithic model on tables `PARTSUPP` and `ORDERS`. However, despite the better average latency error, we have shown that the large model has many more mistakes at the decision point. Traditional means of measuring accuracy are not sufficient.

As we have seen, the larger model does not perform as well as the error rate and loss would lead us to believe. That is because the methods focus on the accuracy of predictions, treating each prediction of equal value. That is not the case. The error of pivotal decisions are much more important and impactful and should be minimized, weighing other errors less.

(a) `LINEITEM` table errors.

(b) `PART` table errors.

(c) `PARTSUPP` table errors.

(d) `ORDERS` table errors.

Figure 5.1: Monolithic (red) vs specialized (blue) average errors, per table

5.2. A New Loss Function

Figure 5.2 shows the computational process in an example neural network (such as Architecture 3). Computed weights are passed to the loss function which, in turn, updates the gradients. Instead of MSE or MAPE, a new loss function that takes into account properly weighting decisions in the critical area before 2% selectivity more heavily, thus rewarding good decisions.

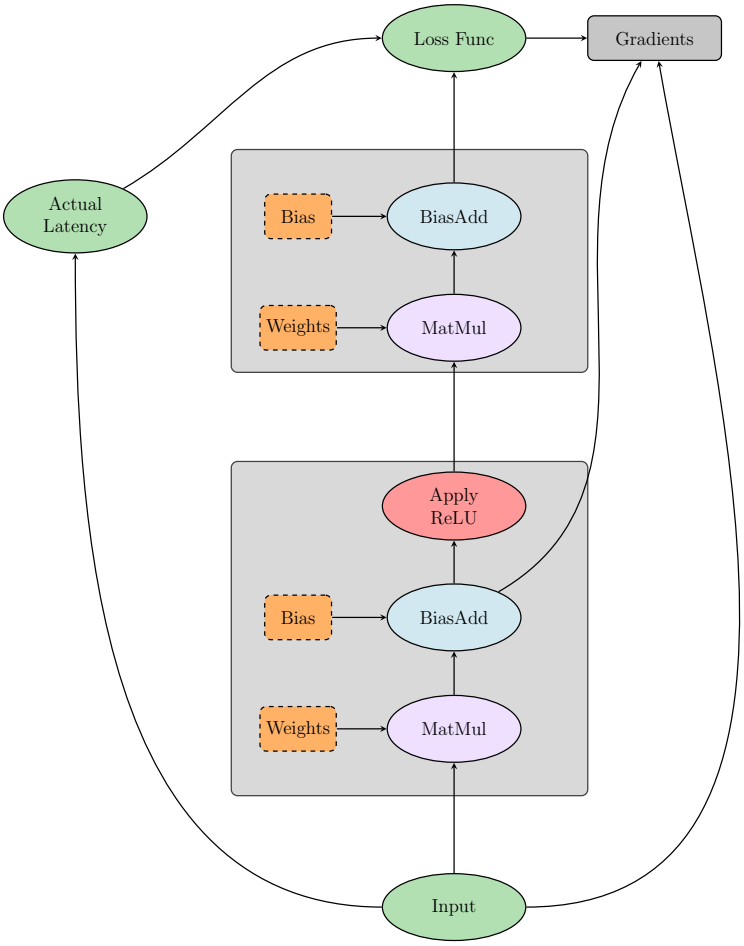


Figure 5.2: The computational process.

Chapter 6: A Learning System

6.1. System Overview

Neural networks can become quite large and increasingly slower to train. To give us as much flexibility as possible while still being sensitive to the performance implications of neural networks, a system consisting of two core components: **a learned optimizer** and a **learning core** (see Figure 6.1) can be conceived.

The *learned optimizer* ultimately selects the best plan for hand-off to the executor. The optimizer is modular, allowing it to be a drop-in replacement for components in various data systems while still allowing system-specific coding and query execution to remain intact.

The *learning engine* generates and trains the model, decoupling the dependency on the data system's internal workings. It also allows a degree of freedom to research entirely new models without disrupting the core optimizer code. It gathers training data by either observing execution at run time, or by a bulk data run. The bulk run executes queries found in Appendix refappendix.queries, while the queries at run time are ad-hoc.

As changes to the underlying hardware or schema occur, the model is periodically retrained and reintroduced to the optimizer. This separation of concerns also fits more naturally to the tooling necessary for each task. Models are trained with more conventional neural network tools (for example, Tensorflow and Keras). The

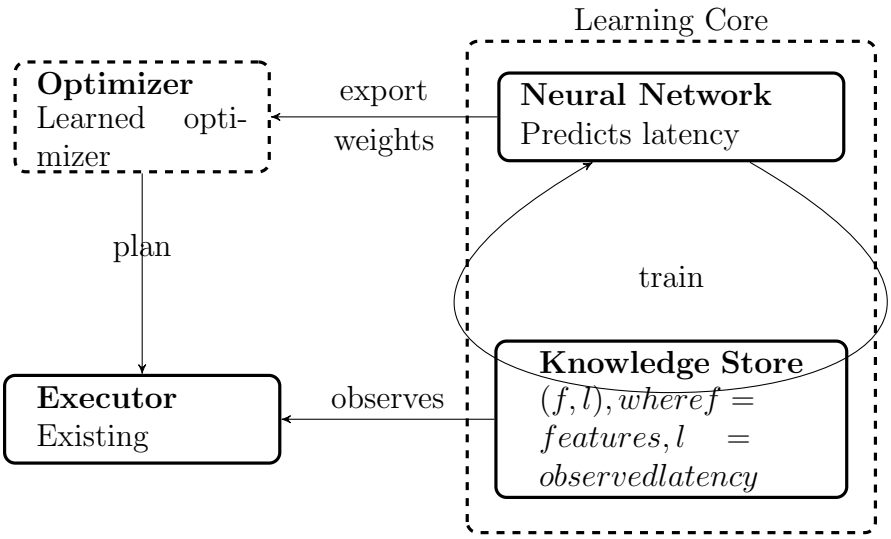


Figure 6.1: An AI learning optimizer system design.

exported raw numeric weights mean that retraining requires changes to the state of the optimizer, but not to its core code.

6.2. The Learning Core

The **learning core**, in turn, is made of two core components; the **deep neural network** and the **knowledge store**. The main contribution of this paper is the neural network described in previous chapters. The *knowledge store* keeps the data needed for training encoded and properly shaped as a tuple:

$$\mathcal{T} = ([feature\_vectors], observed\_latency)$$

Where tuple $\mathcal{T}$ is a set of *features* and the *observed latency* of the executed query.

6.2.1 Training Data

Deep, fundamental changes to configuration require data reshaping and model retraining. In most cases, however, a majority of the data remains valid. For example, adding a new index or column means the portion of the data affected needs reshaping, but the data from previous runs is still valuable for training.

When a system is first setup (or a new hardware profile introduced), there is no training data. This case is referred to as the **bootstrap problem**, while the case of incremental changes to a running system is called the **runtime problem**. Figure 6.2 illustrates the decision tree for determining the type of operation necessary.

Bootstrapping

Run Time

Since our requirements necessitate a fast, accurate response, models train during downtime. The system needs to monitor and collect data for the training set continually. Additional observed data is added to the training set for the next training phase to improve the predictive power of the model.

6.2.2 Plan Manager

The plan manager's primary function is to create permutations of plans based on a given query.

6.2.3 Cost Estimator

The cost estimate provides a prediction for the given plan (feature vectors) and returns the normalized latency prediction. The cost estimator contains very little code other than what is necessary to load, cache, and run the model.

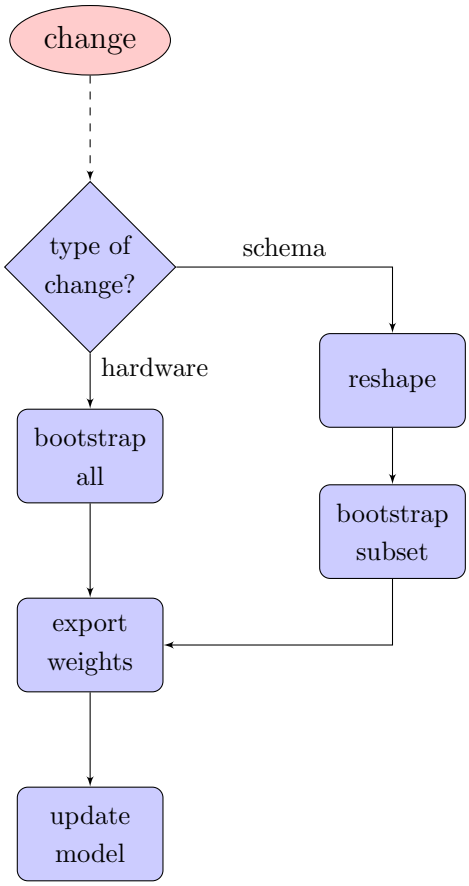


Figure 6.2: Change detection flowchart.

Algorithm 6.1 Enumeration algorithm.

```
1: V(p) = predict(p), for all p ∈ 𝒫        ▷ Prediction for each plan in the space P
2: best = V(0)
3: for each p ∈ 𝒫 do                       ▷ For each plan in the plan space
4:     if V(p) < best then
5:         best ← V(p)
6:     end if
7: end for
8: return best
```

6.2.4 Enumeration Algorithm

The enumeration algorithm also becomes simplified because it does not need to determine the cost of components of the plan, nor which operations are required. It returns the lowest latency plan from the predicted set (Algorithm 6.1).

Chapter 7: Summary and Conclusions

All data systems require optimization. The critical decisions in access path selection involved complex patterns that traditional rule-based approaches cannot solve. In this thesis, I introduce a learned optimizer for determining the most efficient path using small, specialized neural networks. I show that small, specialized neural networks:

- Outperform the current, state of the art optimizer
- Outperform larger, monolithic neural networks
- Do not require selectivity or other table statistics
- Do not need to consider the inner workings of the query plan (for example, sorting).

The requirement for sizable training sets still limits this approach. Also, predicting latency is not an ideal choice. Predicting latency requires that plan options be estimates and compared. Perhaps a more direct approach producing a single prediction of all of the correct plan options could be found. Perhaps reframing the problem in an iterative, reinforcement approach would prove beneficial.

I categorize the directions this work can take as improvement, systematization, and other applications. We already addressed the *improvements* that could be made to address the limitations.

Systematization is the arrangement of this research into a functioning, cohesive system like the one described in Chapter 6. Such a system would have two major components; a learned optimizer that uses the neural network (replacing the existing optimizer), and a knowledge store that trains the network.

Further applications include NoSQL systems such as Apache Cassandra, Lucene-based search, and Kafka. Finally, this approach can automate research to guide algorithmic changes and reduce research time from years to weeks. Artificial intelligence has a place in data systems, and the future of these directions is exciting.

References

Abadi, D., Boncz, P., Harizopoulos, S., Idreos, S., Madden, S., et al. (2013). The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, 5(3), 197–280.

Akdere, M., Çetintemel, U., Riondato, M., Upfal, E., & Zdonik, S. B. (2012). Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering* (pp. 390–401).: IEEE.

Chaudhuri, S. (1998). An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (pp. 34–43).: ACM.

Ganapathi, A., Kuno, H., Dayal, U., Wiener, J. L., Fox, A., Jordan, M., & Patterson, D. (2009). Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *2009 IEEE 25th International Conference on Data Engineering* (pp. 592–603).: IEEE.

Gupta, C., Mehta, A., & Dayal, U. (2008). Pqr: Predicting query execution times for autonomous workload management. In *2008 International Conference on Autonomic Computing* (pp. 13–22).: IEEE.

Hey, A. J., Tansley, S., Tolle, K. M., et al. (2009). *The fourth paradigm: data-intensive scientific discovery*, volume 1. Microsoft research Redmond, WA.

Kester, M. S., Athanassoulis, M., & Idreos, S. (2017). Access path selection in main-memory optimized data systems: Should i scan or should i probe? In *Proceedings of the 2017 ACM International Conference on Management of Data* (pp. 715–730).: ACM.

Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P., & Kemper, A. (2018). Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*.

Kraska, T., Alizadeh, M., Beutel, A., Chi, E. H., Ding, J., Kristo, A., Leclerc, G., Madden, S., Mao, H., & Nathan, V. (2019). Sagedb: A learned database system.

Kraska, T., Beutel, A., Chi, E. H., Dean, J., & Polyzotis, N. (2018). The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data* (pp. 489–504).: ACM.

Krishnan, S., Yang, Z., Goldberg, K., Hellerstein, J., & Stoica, I. (2018). Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*.

Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., & Neumann, T. (2015). How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3), 204–215.

Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., & Tatbul, N. (2019). Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711*.

Marcus, R. & Papaemmanouil, O. (2018). Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (pp.$\tilde{3}$).: ACM.

Markl, V. & Lohman, G. (2002). Learning table access cardinalities with leo. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* (pp. 613–613).: ACM.

Ortiz, J., Balazinska, M., Gehrke, J., & Keerthi, S. S. (2018). Learning state representations for query optimization with deep reinforcement learning. *arXiv preprint arXiv:1803.08604*.

Ramachandran, P., Zoph, B., & Le, Q. V. (2017). Searching for activation functions. *arXiv preprint arXiv:1710.05941*.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6), 386.

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., & Price, T. G. (1979). Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data* (pp. 23–34).: ACM.

Song, S.-K. & Gorla, N. (2000). A genetic algorithm for vertical fragmentation and access path selection. *The Computer Journal*, 43(1), 81–93.

Stillger, M., Lohman, G. M., Markl, V., & Kandil, M. (2001). Leo-db2's learning optimizer. In *VLDB*, volume 1 (pp. 19–28).

Van Aken, D., Pavlo, A., Gordon, G. J., & Zhang, B. (2017). Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (pp. 1009–1024).: ACM.

Wu, W., Chi, Y., Zhu, S., Tatemura, J., Hacigümüs, H., & Naughton, J. F. (2013). Predicting query execution time: Are optimizer cost models really unusable? In *2013 IEEE 29th International Conference on Data Engineering (ICDE)* (pp. 1081–1092).: IEEE.

Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R. R., & Smola, A. J. (2017). Deep sets. In *Advances in neural information processing systems* (pp. 3391–3401).

Appendix A: TPC-H Test Queries

The tables in this appendix are the portions of the query predicate for the columns indicated and should be read as follows:

- A blank in the column cell indicates that column was not used in the predicate
- A static value indicates that the value is used for every variation
- A range (in brackets) indicates that each variation will change in this format: [*start*, *end*, *step*].
- A positive step indicates a greater than comparison, and a negative is a "less than" comparison.

Number	l_extendedprice	l_partkey	l_suppkey	l_quantity	l_discount
2				[2,4,1]	
3			[200,4000,200]		
4				2	[0.01,0.15,0.01]
5				[2,14,1]	0.01
6	[100000,80000,-1000]				
7	80000		[5000,75000,5000]		
8	80000	[100000,2000000,100000]			
9	[75000,95000,2000]	100000			
10	75000	1200000	[75000,6000,-3000]		
11	75000	[100000,1100000,100000]	75000		
12	[75000,95000,2000]		75000		
13	[80000,95000,1000]	[1100000,2000000,60000]	75000		
14	[80000,95000,1000]	1200000	[55000,75000,1000]		
15	750000	[250000,900000,50000]	[25000,90000,5000]		

Table A.1: LINEITEM Table queries.

Number	P_PARTKEY	P_SIZE	P_RETAILPRICE
18	[4000,80000,4000]		
19	[1996000,1920000,-4000]		
10		[49,48,-1]	
21		[2,3,1]	
22			[990,1026,2]
23			[1973,2027,2]
24	[100000,2000000,10000]	3	
25	200000	[1,20,1]	
26		[2,21,1]	1100
27		47	[1060,1660,20]
28	[20000,440000,20000]	10	50000
29	200000	[2,21,1]	50000
20	200000	[10,20,1]	[1000,2200,100]
31	200000	[3,20,1]	[1100,2200,110]
32	[30000,200000,10000]	50000	[3,20,1]
33	[40800,209100,5100]	[960,4920,120]	20
34	[30000,111000,3000]	[1200,4440,110]	[10,37,1]

Table A.2: PART table queries.

Number	O_TOTALPRICE	O_ORDERKEY	O_CUSTKEY
86	[2077,25599,1238]		
87		[120000,2399981,119999]	
88			[3000,59981,2999]
89	73192	[600000,11999981,599999]	
90	73192		[15000,299981,14999]
91		12000000	[15000,299981,14999]
92	[4455,73178,3617]	12000000	
93		[600000,11999981,599999]	299999
94		[4455,73178,3617]	299999
95	120766	19800000	[24750,494981,24749]
96	120766	[990000,19799981,989999]	494999
97	[6834,120758,5997]	19800000	494999
98	6834	[990000,19799981,989999]	494999
99	6834	19800000	[24750,494981,24749]
100	120766	[990000,19799981,989999]	24750

Table A.3: ORRERS table queries.

Number	PS_PARTKEY	PS_SUPPKEY	PS_AVAILQTY	PS_SUPPLYCOST
35	[4000,79981,3999]			
36		[200,3981,299]		
37			[20,381,19]	
38				[2,21,1]
39	400000	[1000,19981,999]		
40	400000		[100,1981,99]	
41	400000			[10,181,9]
42	[20000,399981,9999]	20000		
43		20000	[100,1981,99]	
44		20000		[10,181,9]
45	[20000,399981,9999]		1999	
46		[1000,19981,999]	1999	
47			1999	[10,181,9]
48			[100,1981,99]	200
49	[20000,399981,9999]			200
50		[1000,19981,999]		200
51	660000	33000		[17,321,16]
52	660000		3299	[17,321,16]
53	33000		3299	[17,321,16]
54	660000	[1650,32981,1649]		330
55	660000	[1650,32981,1649]	3299	
56		[1650,32981,1649]	3299	330
57	660000	33000	[165,3281,164]	
58	660000	[165,3281,164]		330
59	33000		[165,3281,164]	330
60	[33000,659981,32999]	33000	3299	
61	[33000,659981,32999]		3299	330
62	[33000,659981,32999]	33000		330
63	660000		[165,3281,164]	[17,321,16]
64		33000	[165,3281,164]	[17,321,16]
65	660000	[1650,32981,1649]	[1650,32981,1649]	
66		[1650,32981,1649]	[165,3281,164]	330
67	660000	[1650,32981,1649]		[17,321,16]
68		[1650,32981,1649]	3299	[17,321,16]
69	[33000,659981,32999]		[165,3281,164]	330
70	[33000,659981,32999]	33000	[165,3281,164]	
71	[33000,659981,32999]	[1650,32981,1649]		330
72	[33000,659981,32999]	[1650,32981,1649]	3299	
73	[33000,659981,32999]		3299	[17,321,16]
74	[33000,659981,32999]	33000		[17,321,16]
75	940000	47000	4699	[24,461,23]
76	940000	[2350,46981,2349]	4699	470
77	940000	47000	[235,4681,234]	470
78	[47000,939981,46999]	47000	4699	470
79	940000	47000	[235,4681,234]	[24,461,23]
80	940000	[2350,46981,2349]	[235,4681,234]	470
81	940000	[2350,46981,2349]	4699	[24,461,23]
82	[47000,939981,46999]	47000	[235,4681,234]	470
83	[47000,939981,46999]	[2350,46981,2349]	4699	470
84	[47000,939981,46999]	47000	4699	[24,461,23]
85	[47000,939981,46999]	[2350,46981,2349]	[235,4681,234]	470

Table A.4: PARTSUPP table queries.

Appendix B: AWS Hardware Configurations

	c5d.4xlarge CPU Optimized	r5d.4xlarge Memory Optimized	m5d.2xlarge General Purpose
h_mem System Memory (GB)	30	128	32
h_cpu CPU Architecture	Intel(R) Xeon(R) Platinum 8124M	Intel(R) Xeon(R) Platinum 8175M	Intel(R) Xeon(R) CPU E5-2686 v4
h_cores Number of Cores	16	16	16
h_clk CPU Clock Speed (Mhz)	3405.776	3110.95	2300.107
h_l1d Level1 Cache Size for Data	32	32	32
h_l1i Level1 Cache Size for Instructions	32	32	32
h_l2 Level2 Cache Size	1024	1024	256
h_l3 Level3 Cache Size	25344	33792	46080
h_numa NUMA Aware?	✓	✓	✓
h_avx2 Supports SIMD?	✓	✓	✓
h_avx512bw SIMD 512 Bit Words?	✓	✓	✓
h_avx512vl SIMD Vector Length Extensions?	✓	✓	✓
h_avx512cd SIMD Conflict Detection?	✓	✓	✓
h_avx512dq SIMD Double and Quad Words?	✓	✓	
h_bogomips Linux bogus MIPS Score	6000	5000	4600
h_ss Self Snoop?	✓	✓	
h_3dnowprefetch 3D Now Prefetch?	✓	✓	
h_bmi2 Bitmanipulation Instruction Set?	✓		✓
h_erms Enhanced Rep Movsb/Stosb?	✓		✓
h_invpcid Supports Invalidate Process Context Identifiers?	✓		✓

Table B.1: AWS hardware configurations.

Appendix C: Batches

C.1. Optimizer Tuning

Optimizer tuning batches in Table C.1 tune the following PostgresSQL settings:

- `seq_page_cost (SPC)`
- `random_page_cost (RPC)`
- `cpu_tuple_cost (CTC)`
- `cpu_operator_cost (COC)`
- `parallel_setup_cost (PSC)`
- `parallel_tuple_cost (PTC)`
- `min_parallel_table_scan_size (MPTSS)`
- `effective_cache_size (ECS)`

C.2. Table Batches

BATCH	SPC	RPC	CTC	COC	PSC	PTC	MPTSS	ECS
2-1	2							
2-2	3							
2-3	4							
2-4	5							
2-5		1						
2-6		2						
2-7		3						
2-8		5						
2-9			.02					
2-10			.03					
2-11			.04					
2-12			.05					
2-13				0.0035				
2-14				0.0015				
2-15				0.0005				
2-16				0.0055				
2-17					2000			
2-18					10			
2-19					100			
2-20					5000			
2-21					8000			
2-22					10000			
2-23						.01		
2-24						.001		
2-25						.5		
2-26						.2		
2-27								
2-28								1
2-29								2
2-30								8
2-31								16
2-32							1MB	
2-33							4MB	
2-34							16MB	

Table C.1: Optimizer tuning batches.

BATCH	L_PARTKEY	L_SUPPKEY	L_DISCOUNT	L_QTY	L_EXTENDED_PRICE	bitmap on	parallel on	index on
3-1a								
3-1b							Y	
3-2a	Y							Y
3-2b	Y						Y	Y
3-3a		Y						Y
3-3b		Y					Y	Y
3-4a		Y	Y					Y
3-4b		Y	Y				Y	Y
3-5a		Y	Y	Y			Y	Y
3-5b		Y	Y	Y				Y
3-6a		Y		Y				Y
3-6b		Y		Y			Y	Y
3-7a				Y	Y			Y
3-7b				Y	Y		Y	Y
3-8a	Y				Y			Y
3-8b	Y				Y		Y	Y
3-9a		Y			Y			Y
3-9b		Y			Y		Y	Y
3-10a	Y	Y						Y
3-10b	Y	Y					Y	Y
3-11a	Y	Y			Y			Y
3-11b	Y	Y			Y		Y	Y
3-12a	Y					Y		
3-12b	Y					Y	Y	
3-13a		Y				Y		
3-13b		Y				Y	Y	
3-14a		Y	Y			Y		
3-14b		Y	Y			Y	Y	
3-15a		Y	Y	Y		Y		
3-15b		Y	Y	Y		Y	Y	
3-16a		Y		Y		Y		
3-16b		Y		Y		Y	Y	
3-17a				Y	Y	Y		
3-17b				Y	Y	Y	Y	
3-18a	Y			Y	Y	Y		
3-18b	Y			Y	Y	Y	Y	
3-19a		Y			Y	Y		
3-19b		Y			Y	Y	Y	
3-20a	Y	Y				Y		
3-20b	Y	Y				Y	Y	
4-1a	Y	Y	Y	Y	Y	Y	Y	Y
4-2b	Y	Y	Y	Y	Y		Y	Y
4-3a	Y	Y	Y	Y	Y			Y
4-4b	Y	Y	Y	Y	Y	Y		Y
4-5a	Y	Y	Y	Y	Y	Y	Y	Y

Table C.2: `LINEITEM` table batches.

BATCH	P_PARTKEY	P_RETAILPRICE	P_SIZE	bitmap on	parallel on	index on
6-1a						
6-1b					Y	
6-2a	Y	Y	Y			Y
6-2b	Y	Y	Y		Y	Y
6-3a	Y	Y	Y	Y		
6-3b	Y	Y	Y	Y	Y	
6-4a	Y	Y				Y
6-4b	Y	Y			Y	Y
6-5a	Y	Y		Y		
6-5b	Y	Y		Y	Y	
6-6a	Y					Y
6-6b	Y				Y	Y
6-7a	Y					
6-7b	Y			Y	Y	
6-8a	Y		Y			Y
6-8b	Y		Y		Y	Y
6-9a	Y		Y	Y		
6-9b	Y		Y	Y	Y	
6-10a		Y	Y			Y
6-10b		Y	Y		Y	Y
6-11a		Y	Y	Y		
6-11b		Y	Y	Y	Y	
6-12a						Y
6-12b					Y	Y
6-13a				Y		
6-13b				Y	Y	

Table C.3: `PART` table batches.

BATCH	O_TOTALPRICE	O_ORDERKEY	O_CUSTKEY	bitmap on	parallel on	index on
8-1a						
8-1b					Y	
8-2a	Y			Y		
8-2b	Y			Y	Y	
8-3a	Y					Y
8-3b	Y				Y	Y
8-4a	Y	Y		Y		
8-4b	Y	Y		Y	Y	
8-5a	Y	Y				Y
8-5b	Y	Y			Y	Y
8-6a	Y	Y	Y	Y		
8-6b	Y	Y	Y	Y	Y	
8-7a	Y	Y	Y			Y
8-7b	Y	Y	Y		Y	Y
8-8a	Y		Y	Y		
8-8b	Y		Y	Y	Y	
8-9a	Y		Y			Y
8-9b	Y		Y		Y	Y
8-10a			Y	Y		
8-10b			Y	Y	Y	
8-11a			Y			Y
8-11b			Y		Y	Y
8-12a		Y	Y	Y		
8-12b		Y	Y	Y	Y	
8-13a		Y	Y			Y
8-13b		Y	Y		Y	Y
8-14a		Y		Y		
8-14b		Y		Y	Y	
8-15a		Y				Y
8-15b		Y			Y	Y

Table C.4: `ORDERS` table batches.

BATCH	PS_PARTKEY	PS_AVAILQTY	PS_SUPPLYCOST	PS_SUPPKEY	bitmap on	parallel on	index
7-1a							
7-1b						Y	
7-2a	Y				Y		
7-2b	Y				Y	Y	
7-3a	Y						Y
7-3b	Y					Y	Y
7-4a	Y	Y			Y		
7-4b	Y	Y			Y	Y	
7-5a	Y	Y					Y
7-5b	Y	Y				Y	Y
7-6a	Y	Y	Y		Y		
7-6b	Y	Y	Y		Y	Y	
7-7a	Y	Y	Y				Y
7-7b	Y	Y	Y			Y	Y
7-8a	Y	Y	Y	Y	Y		
7-8b	Y	Y	Y	Y	Y	Y	
7-9a	Y	Y	Y	Y			Y
7-9b	Y	Y	Y	Y		Y	Y
7-10a		Y	Y	Y	Y		
7-10b		Y	Y	Y	Y	Y	
7-11a		Y	Y	Y			Y
7-11b		Y	Y	Y		Y	Y
7-12a			Y	Y	Y		
7-12b			Y	Y	Y	Y	
7-13a			Y	Y			Y
7-13b			Y	Y		Y	Y
7-14a				Y	Y		
7-14b				Y	Y	Y	
7-15a				Y			Y
7-15b				Y		Y	Y
7-16a	Y			Y	Y		
7-16b	Y			Y	Y	Y	
7-17a	Y			Y			Y
7-17b	Y			Y		Y	Y
7-18a	Y	Y		Y	Y		
7-18b	Y	Y		Y	Y	Y	
7-19a	Y	Y		Y			Y
7-19b	Y	Y		Y		Y	Y
7-20a		Y		Y	Y		
7-20b		Y		Y	Y	Y	
7-21a		Y		Y			Y
7-21b		Y		Y		Y	Y
7-22a		Y	Y		Y		
7-22b		Y	Y		Y	Y	
7-23a		Y	Y				Y
7-23b		Y	Y			Y	Y
7-24a	Y		Y		Y		
7-24b	Y		Y		Y	Y	
7-25a	Y		Y				Y
7-25b	Y		Y			Y	Y

Table C.5: `PARTSUPP` table batches.

Appendix D: Neural Network Example Code

D.1. Keras, Tensorflow and Python

Listing D.1: modeltrain.py

```
import models as m
import pandas as pd
import matplotlib.pyplot as plt

epochs = 500

#import cleaned and feature vectorized input
X = pd.read_hdf("x.h5")
Y = pd.read_hdf("y.h5")

path = "Lineitem_Models/ModelMLP-"

#Architecture 1
model = m.get_model([512], ["relu"])
hist = m.train_model(model, f"{path}1-512", X, Y, epochs)

model = m.get_model([256], ["relu"])
hist = m.train_model(model, f"{path}1-256", X, Y, epochs)

model = m.get_model([128], ["relu"])
hist = m.train_model(model, f"{path}1-128", X, Y, epochs)

#Architecture 2
model = m.get_model([512, 256], ["relu", "relu"])
hist = m.train_model(model, f"{path}2-512", X, Y, epochs)

model = m.get_model([256, 128], ["relu", "relu"])
hist = m.train_model(model, f"{path}2-256", X, Y, epochs)

model = m.get_model([128, 64], ["relu", "relu"])
hist = m.train_model(model, f"{path}2-128", X, Y, epochs)

#Architecture 3
model = m.get_model([512, 256], ["relu", "linear"])
hist = m.train_model(model, f"{path}3-512", X, Y, epochs)
```

```
model = m.get_model([256, 128], ["relu", "linear"])
hist = m.train_model(model, f"{path}3-256", X, Y, epochs)

model = m.get_model([128, 64], ["relu", "linear"])
hist = m.train_model(model, f"{path}3-128", X, Y, epochs)

#Architecture 4
model = m.get_model([512, 256,128], ["relu", "relu","relu"])
hist = m.train_model(model, f"{path}4-512", X, Y, epochs)

model = m.get_model([256, 128, 64], ["relu", "relu","relu"])
hist = m.train_model(model, f"{path}4-256", X, Y, epochs)

model = m.get_model([128, 64,32], ["relu", "relu","relu"])
hist = m.train_model(model, f"{path}4-128", X, Y, epochs)

#Architecture 5
model = m.get_model([512, 256,128], ["relu", "linear","relu"])
hist = m.train_model(model, f"{path}5-512", X, Y, epochs)

model = m.get_model([256, 128, 64], ["relu", "linear","relu"])
hist = m.train_model(model, f"{path}5-256", X, Y, epochs)

model = m.get_model([128, 64,32], ["relu", "linear","relu"])
hist = m.train_model(model, f"{path}5-128", X, Y, epochs)

#Architecture 6
model = m.get_model([512, 256,512], ["relu", "relu","relu"])
hist = m.train_model(model, f"{path}6-512", X, Y, epochs)

model = m.get_model([256, 128,256], ["relu", "relu","relu"])
hist = m.train_model(model, f"{path}6-512", X, Y, epochs)

model = m.get_model([128, 64,128], ["relu", "relu","relu"])
hist = m.train_model(model, f"{path}6-512", X, Y, epochs)

#Architecture 7
model = m.get_model([512, 256, 512], ["relu", "linear","relu"])
hist = m.train_model(model, f"{path}7-512" , X, Y, epochs)

model = m.get_model([256, 128,256], ["relu", "linear","relu"])
hist = m.train_model(model, f"{path}7-256" , X, Y, epochs)

model = m.get_model([128, 64,128], ["relu", "linear","relu"])
hist = m.train_model(model, f"{path}7-128" , X, Y, epochs)

#Architecture 8
model = m.get_model([512, 512, 256 ,256, 512, 512], ["relu", "linear","relu", "
    linear","relu", "linear"])
hist = m.train_model(model, f"{path}8-512", X, Y, epochs)
```

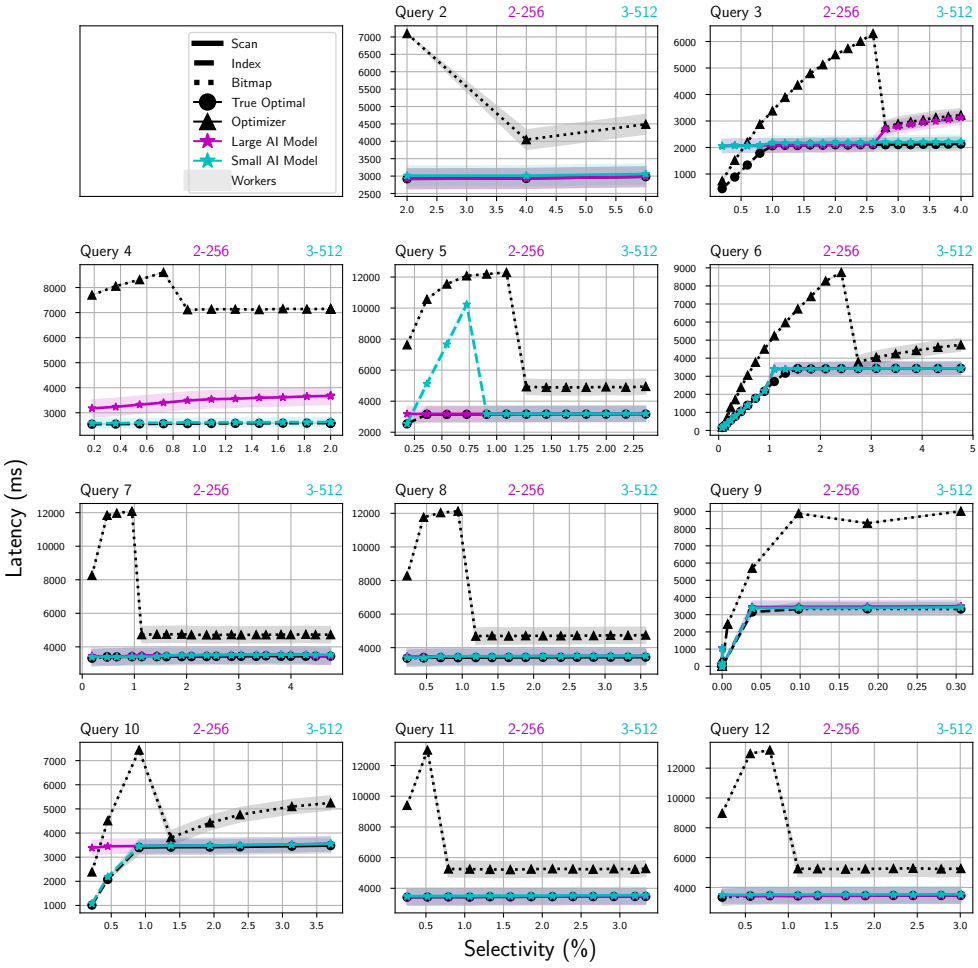
```
model = m.get_model([256, 256, 128 ,128, 256, 256], ["relu", "linear","relu", "
    linear","relu", "linear"])
hist = m.train_model(model, f"{path}8-256", X, Y, epochs)
```

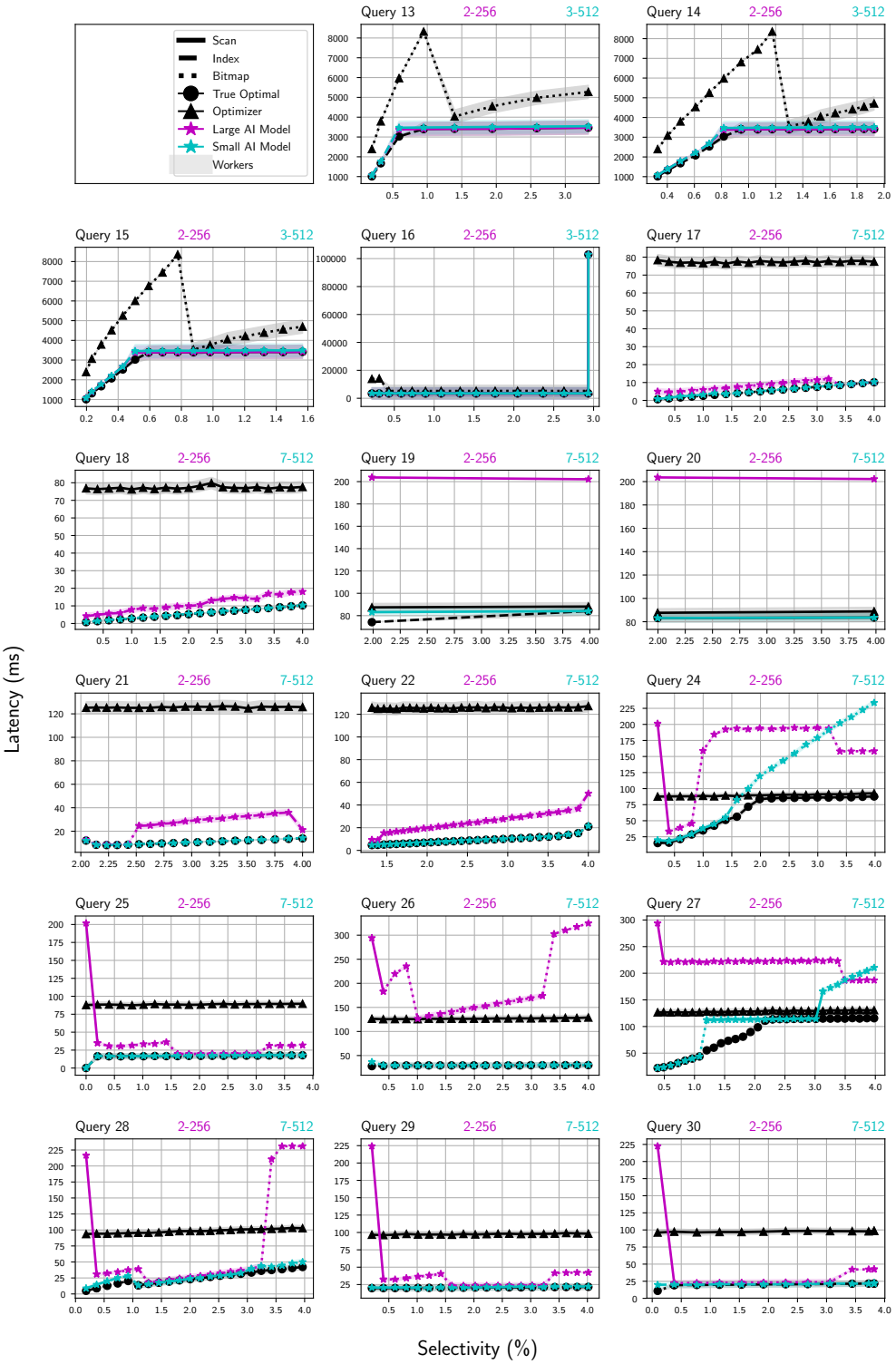
Listing D.2: models.py

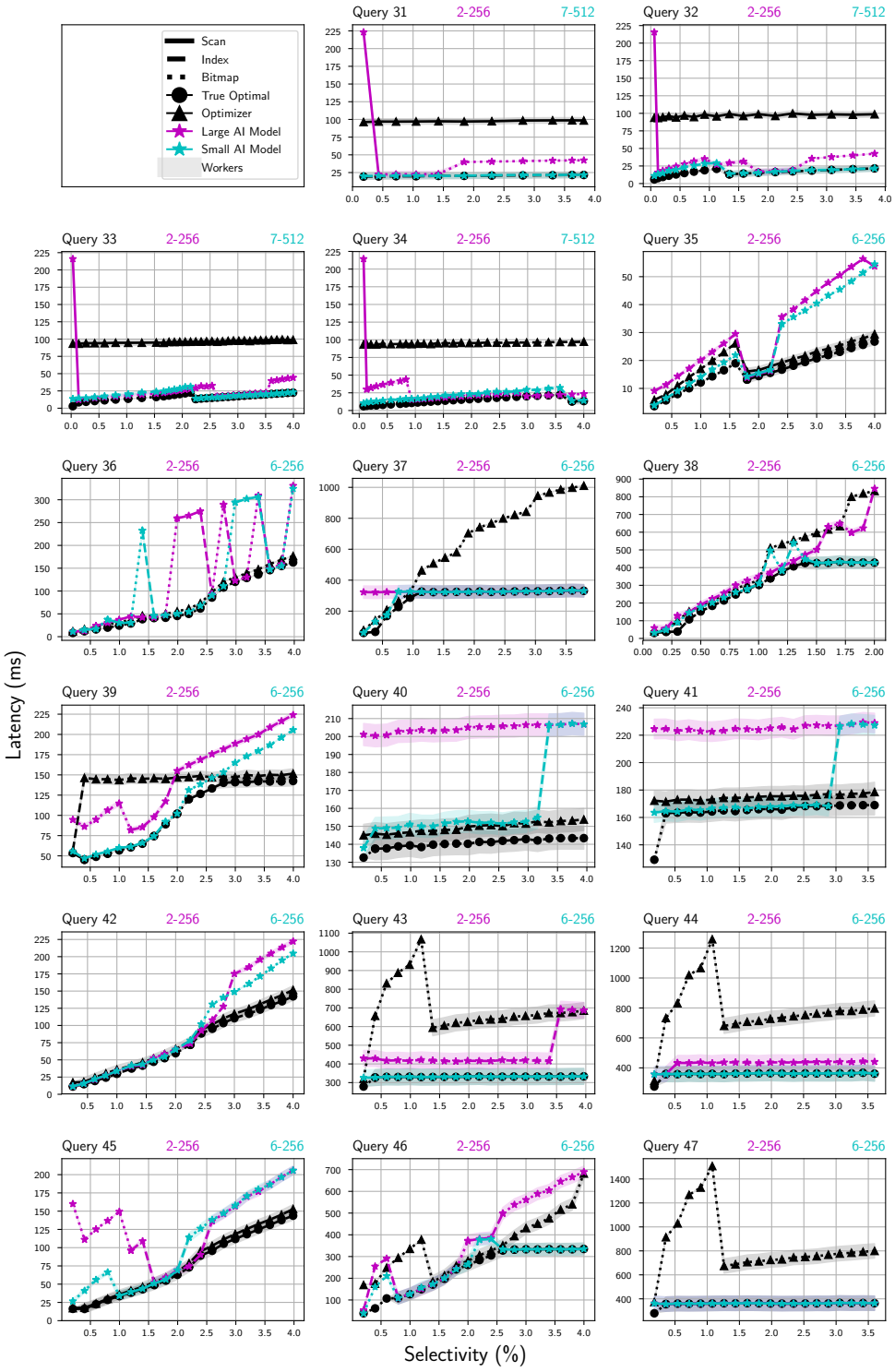
```
import pandas as pd
import tables as pt
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

import keras
from keras.layers import Input, Dense
from keras.models import Model, Sequential
from keras.callbacks import CSVLogger

#takes 2 lists... a list of counts of neurons per layer, and a list of activations
     for the layer, and a loss function (default is MSE)
def get_model(neurons, activations, loss='mean_squared_error'):
    keras.backend.clear_session()

    model = Sequential()

    for (n, a) in zip(neurons, activations):
        model.add(Dense(n, activation=a))

    #model.add(Dense(n_1, activation=act_1))
    #model.add(Dense(n_2, activation=act_2))
    model.add(Dense(1, kernel_initializer='normal'))

    model.compile(loss=loss, optimizer='adam', metrics=['mse', 'mae', 'mape', '
    cosine'])

    return model

def train_model(m, filename, x, y, epochs, batch_size=None):
    cb = keras.callbacks.ModelCheckpoint(f"{filename}.h5", monitor='val_loss',
    verbose=0, save_best_only=True, save_weights_only=False, mode='auto', period
    =1)
    hist = m.fit(x=x.values, y=y.values, epochs=epochs, batch_size=batch_size,
    shuffle=True, validation_split=.2, callbacks=[cb])

    print("Evaluating Model")
    print (m.evaluate(x.values, y.values))

    pd.DataFrame(hist.history).to_csv(f"{filename}-hist.csv")

    return hist

#this function will help plot the test/train accuracy
def plot_hist(hist):
    fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
    fig.set_figheight(15)
    fig.set_figwidth(15)
```

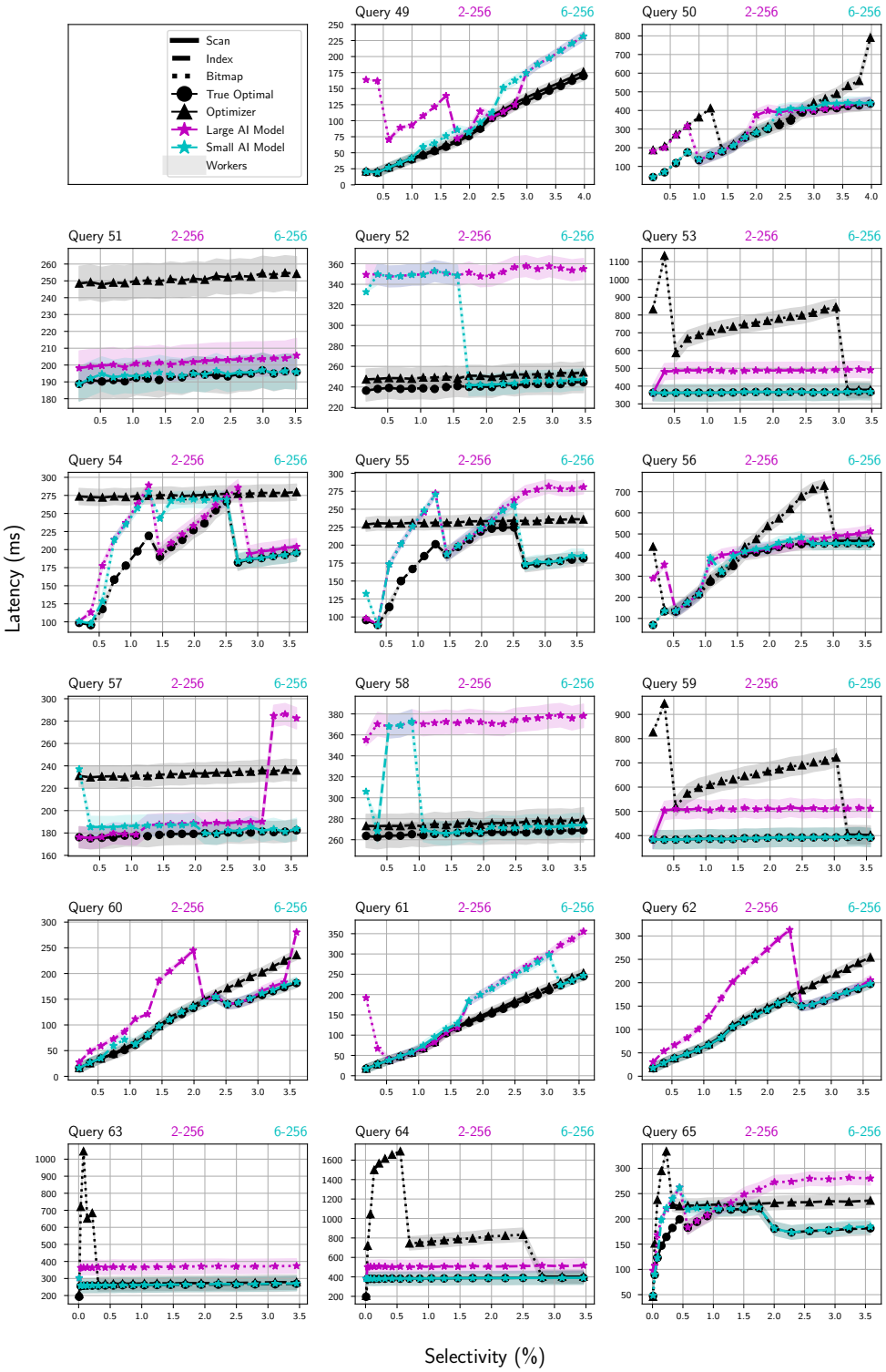
```
epochs = len(hist.history["loss"])

x = range(1, epochs + 1)

#plot the loss as circles and the accuracy as a line
ax1.plot(x, hist.history['loss'], "bo", label="Loss Training")
ax1.plot(x, hist.history['val_loss'], "b", label="Loss Validation")

ax2.plot(x, hist.history['mean_absolute_error'], "bo", label="MAE Training")
ax2.plot(x, hist.history['val_mean_absolute_error'], "b", label="MAE
Validation")

ax3.plot(x, hist.history['mean_absolute_percentage_error'], "bo", label="MAPE
Training")
ax3.plot(x, hist.history['val_mean_absolute_percentage_error'], "b", label="
MAPE Validation")

ax4.plot(x, hist.history['cosine_proximity'], "bo", label="Cosine Training")
ax4.plot(x, hist.history['val_cosine_proximity'], "b", label="Cosine
Validation")


ax1.set_title('Training and Validation Loss')
ax1.set_xlabel('Epochs')
ax1.set_ylabel('Loss')
ax1.legend()

ax2.set_title('Training and Validation MAE')
ax2.set_xlabel('Epochs')
ax2.set_ylabel('MAE')
ax2.legend()

ax3.set_title('Training and Validation MAPE')
ax3.set_xlabel('Epochs')
ax3.set_ylabel('MAPE')
ax3.legend()

ax4.set_title('Training and Validation Cosine')
ax4.set_xlabel('Epochs')
ax4.set_ylabel('Cosine')
ax4.legend()

plt.show()
```

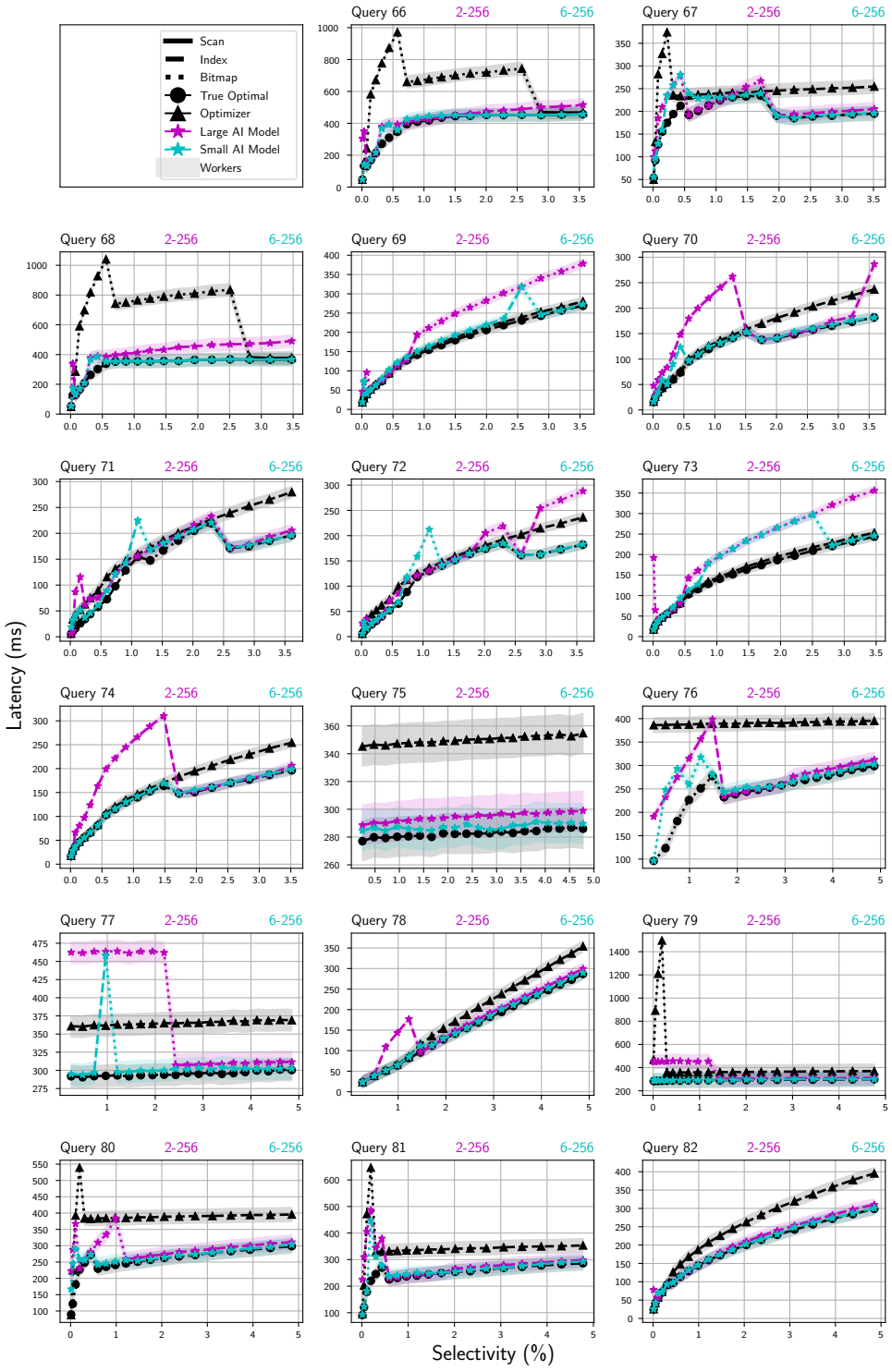
Appendix E: Model Decisions

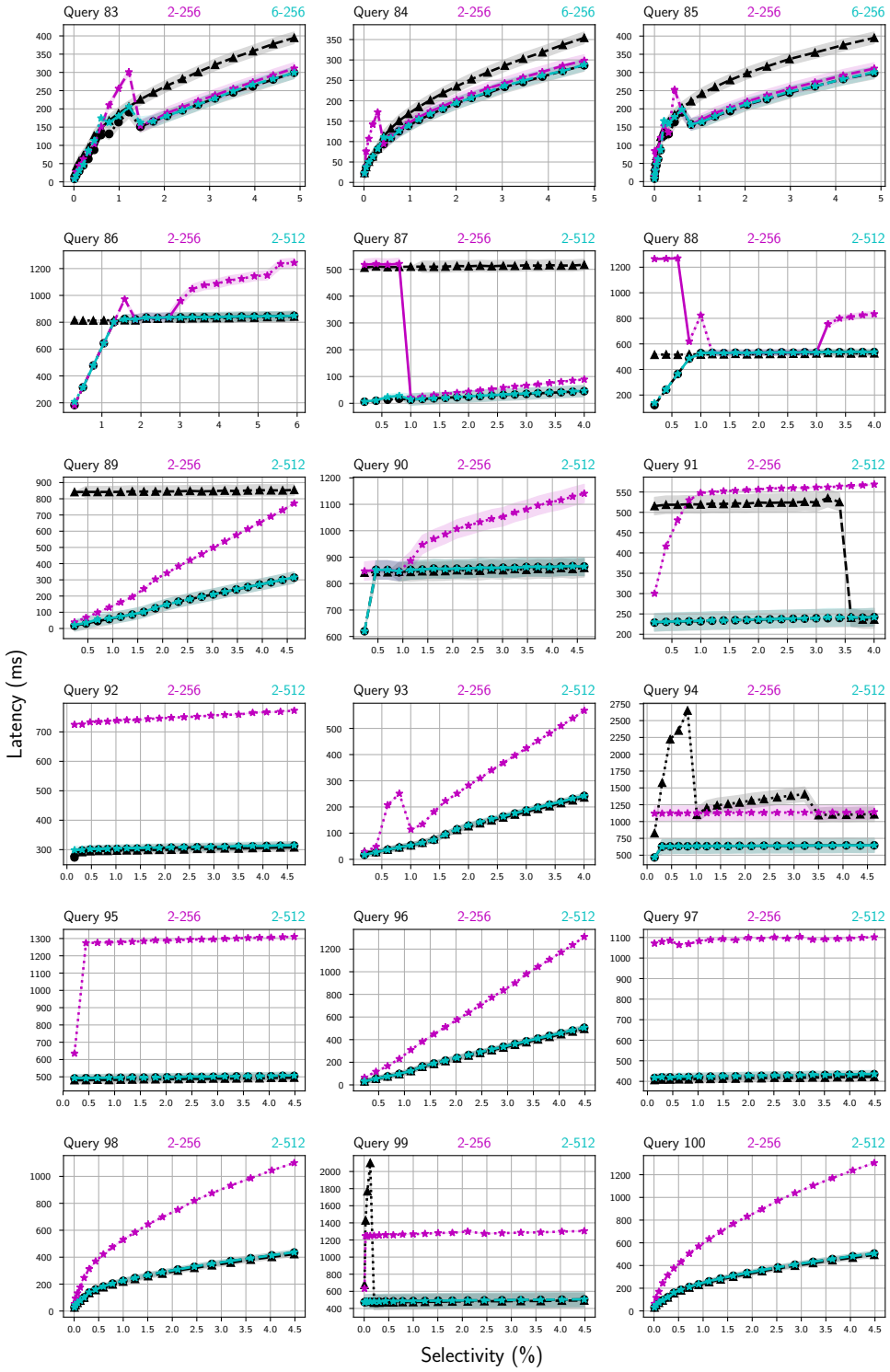



Scan
Index
Bitmap
True Optimal
Optimizer
Large AI Model
Small AI Model
Workers
Query 13 2-256 3-512
Query 14 2-256 3-512
Query 15 2-256 3-512
Query 16 2-256 3-512
Query 17 2-256 7-512
Query 18 2-256 7-512
Query 19 2-256 7-512
Query 20 2-256 7-512
Query 21 2-256 7-512
Query 22 2-256 7-512
Query 24 2-256 7-512
Query 25 2-256 7-512
Query 26 2-256 7-512
Query 27 2-256 7-512
Query 28 2-256 7-512
Query 29 2-256 7-512
Query 30 2-256 7-512
Latency (ms)
Selectivity (%)


Scan
Index
Bitmap
True Optimal
Optimizer
Large AI Model
Small AI Model
Workers
Query 31 2-256 7-512
Query 32 2-256 7-512
Query 33 2-256 7-512
Query 34 2-256 7-512
Query 35 2-256 6-256
Query 36 2-256 6-256
Query 37 2-256 6-256
Query 38 2-256 6-256
Query 39 2-256 6-256
Query 40 2-256 6-256
Query 41 2-256 6-256
Query 42 2-256 6-256
Query 43 2-256 6-256
Query 44 2-256 6-256
Query 45 2-256 6-256
Query 46 2-256 6-256
Query 47 2-256 6-256
Latency (ms)
Selectivity (%)


Scan
Index
Bitmap
True Optimal
Optimizer
Large AI Model
Small AI Model
Workers
Query 49
Query 50
Query 51
Query 52
Query 53
Query 54
Query 55
Query 56
Query 57
Query 58
Query 59
Query 60
Query 61
Query 62
Query 63
Query 64
Query 65
2-256
6-256
Latency (ms)
Selectivity (%)


Scan
Index
Bitmap
True Optimal
Optimizer
Large AI Model
Small AI Model
Workers
Query 66 2-256 6-256
Query 67 2-256 6-256
Query 68 2-256 6-256
Query 69 2-256 6-256
Query 70 2-256 6-256
Query 71 2-256 6-256
Query 72 2-256 6-256
Query 73 2-256 6-256
Query 74 2-256 6-256
Query 75 2-256 6-256
Query 76 2-256 6-256
Query 77 2-256 6-256
Query 78 2-256 6-256
Query 79 2-256 6-256
Query 80 2-256 6-256
Query 81 2-256 6-256
Query 82 2-256 6-256
Latency (ms)
Selectivity (%)


Query 83 2-256 6-256
Query 84 2-256 6-256
Query 85 2-256 6-256
Query 86 2-256 2-512
Query 87 2-256 2-512
Query 88 2-256 2-512
Query 89 2-256 2-512
Query 90 2-256 2-512
Query 91 2-256 2-512
Query 92 2-256 2-512
Query 93 2-256 2-512
Query 94 2-256 2-512
Query 95 2-256 2-512
Query 96 2-256 2-512
Query 97 2-256 2-512
Query 98 2-256 2-512
Query 99 2-256 2-512
Query 100 2-256 2-512
Latency (ms)
Selectivity (%)