The BigDawg Monitoring Framework

by

Peinan Chen

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by .....
Michael R. Stonebraker
Adjunct Professor
Thesis Supervisor
May 20, 2016

Accepted by .....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee
May 20, 2016

The BigDawg Monitoring Framework

by

Peinan Chen

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I designed and implemented a monitoring framework for the BigDawg federated database system which maintains performance information on benchmark queries. As environmental conditions change, the monitoring framework updates existing performance information to match current conditions. Using this information, the monitoring system can determine the optimal query execution plan for similar incoming queries. A series of test queries were run to assess whether the system correctly determines the optimal plans for such queries.

Thesis Supervisor: Michael R. Stonebraker
Title: Adjunct Professor

Acknowledgments

I would first like to thank my thesis advisor Professor Michael R. Stonebraker. Professor Stonebraker provided me with guidance throughout the project. Whenever I was blocked or confused, he resolved the problem and helped me move full speed ahead.

I would also like to thank Zuohao She, Adam Dziedzic, and Ankush Gupta for their work on BigDawg. Whenever, I encountered problems, they were there to steer me in the right direction.

Lastly, I must express my very profound gratitude to my parents for their continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Peinan Chen

Contents

List of Figures

List of Tables

Chapter 1

Introduction

BigDawg is a system that utilizes a federated architecture to enable query processing over multiple databases, where each of the underlying storage engines may have a distinct data model. One important component of BigDawg is a monitoring system which keeps track of past queries' runtime information and utilizes this information to choose the best query plan for an incoming query. The main way the monitoring system associates incoming queries with benchmark queries is by utilizing a signature system.

Chapter two describes the architecture of the the BigDawg system, focusing on the design decisions of the monitoring system and the signature system in particular.

Chapter three analyzes the signature system, describing what kinds of queries the system works well for and the potential overhead of the system. Possible future extensions to the project are also discussed.

1.1 Motivations for BigDawg

It has become abundantly clear that there are a multitude of different features that people desire in their data storage engine. Currently, there are a variety of different storage engines that each have their own costs and benefits. For common Javascript applications that use a web browser paired with a backend database management system (DBMS), NoSQL engines tend to be well-suited. Similarly, relational col-

umn stores are the engine of choice for data warehouses while main memory SQL (NewSQL) systems are best suited for online transaction processing. In general, there needs to be a variety of different storage engines to satisfy different types of applications.

While the applications discussed so far are well suited to a single type of storage engine, there are many other applications that would be best implemented with a combination of different storage engines. For example, the Intel Science and Technology Center (ISTC) has built a medical application that utilizes the MIMIC II dataset [1]. This dataset contains patient metadata, text data (notes taken by medical professionals), semi-structured data (prescriptions and lab results), and waveform data (measurements such as heart rate from bedside devices). A medical application that utilizes this dataset would ideally support standard SQL analytics, complex analytics (such as computing the FFT of a patient's waveform data and comparing it to what is considered normal), text search (such as looking at patients that medical professions have used specific terms for in their notes), and real-time monitoring (such as detecting abnormal heart rhythms).

Although it is possible to implement such an application using a single storage engine, it would be much more efficient to utilize several different storage engines. Thus, to support datasets such as MIMIC II, we created a federated database system called BigDawg [2]. For the MIMIC II dataset, BigDawg uses SciDB to store the historical time series data, Accumulo for text, Postgres for patient metadata, and S-Store to store and process the real-time waveform data. Any queries that depend on multiple storage engines will query all necessary storage engines. For example, to compare current waveforms to historical ones, one will query S-Store and SciDB. To find metadata associated with particular kinds of prescriptions or doctor's notes, one will query Accumulo and Postgres. To run analytics on particular cohorts of patients, one would query Postgres and SciDB.

1.2 Overview of BigDawg

BigDawg is designed with three main goals in mind. First, BigDawg must provide location transparency. This transparency enables users to pose declarative queries that span several data management systems without becoming mired in the underlying data's present location or how to assign work to each storage engine. The architecture to support such location transparency is to write a *shim* for each storage engine that translates a query/update in the federation query language into the language supported by the underlying storage engine. We term such a location transparent federation an *island of information*. Since it is unlikely that a single island will offer the full functionality of all of the federation's database engines, our framework is designed for multi-island operations. Likewise, a storage engine is not restricted to a single island.

Secondly, each island will only support the intersection of the functionality of its composite storage engines. Hence, it is easy for users to express their queries in a single query language over multiple data stores. On the other hand, users do not want to lose any of the functionality their databases provided before federation. To provide the union of the capabilities of the federation's underlying storage engines, BigDawg offers degenerate islands. These islands have the full functionality of a single storage engine.

The third point we discuss concerns multiple islands of information over the same storage engines. In BigDawg, we have multiple groups working on islands. One is using a relational model [3] and the second is using associative arrays [4]. Since both islands include the same data (MIMIC II) stored in the same engines, we will have to support multiple islands over the same data. In general, we must expect multiple islands of information on overlapping subsets of storage engines.

Due to the above points, we define an island of information as:

- A data model that will likely exclude features that are difficult to map to underlying systems.
- A query language on that data model.

- A collection of storage engines for which shims exist to support the data model and query language.

In general, we expect to have a substantial number of islands defined on a collection of storage engines. There may be multiple islands, each with a different data model and query language. There will also be islands consisting of the full capabilities an underlying storage engine. Such a multi-island architecture is shown in Figure B-1.

BigDawg utilizes a scope-cast facility for queries that span multiple islands. When a user command cannot be supported by a single island system, BigDawg allows the user to break his query into multiple island languages - each of which is a subquery. To specify which island a subquery is intended for, the user indicates a SCOPE specification. A cross-island query will have multiple scopes to indicate what subqueries go where. In addition, when multiple islands support disparate storage engines, we will have to move datasets or intermediate results from one island system to another as needed to process a complete query. The full functionality of a multi-island system requires a CAST operation to perform such data movement. A user may insert a CAST operation to denote when an object should be accessed with a given set of semantics.

For example, consider a cross-island operation, such as a join between an object in an array island and one in a table island. Here, we can CAST the array to the table island and do a relational join or we can do the converse and perform an array join. Since the each of these options produces a different output, a user must specify the semantics he desires using SCOPE and CAST commands.

1.3 Query Optimization

For the purposes of this thesis, we will focus only on intra-island queries since our monitoring system is mainly utilized for intra-island query optimization. Specifically, we focus on relatively cheap operations. For such operations, we assume a general rule that local computation should always be done where possible. This is because it is expensive to move an object between engines because it will generally have to be

reformatted and/or converted from one representation to another.

We divide any query that spans multiple islands into stages. In Stage 1, we perform all possible local computations that do not require any data movement. At the end of Stage 1, we are left with computations on collections of objects at different sites. Divide such computations into independent subsets consisting of a collection of objects $O_1, ..., O_k$ along with "joining" specifications for how the objects are put together and computations on the output of such operations. Although the entire rest of the query can be one such collection, we look for collections of size two initially. Effectively, we look for a "bushy" tree of such computations, which will often be unique. There will be L such collections, each operating on non-overlapping sets of objects. In parallel at Stage 2, perform each of these L computations.

For each collection, pick a storage engine, E, and move the $O_1, ..., O_k$ objects to E and perform the computation at E. Continue with additional stages until the query is "solved". In this case, the fundamental query optimization decision is the choice of E for each collection.

1.4 Overview of the Monitoring System

The purpose of the monitoring system is to determine the choice of the storage engine, E, for each collection of objects. This system has two modes, training and production. In training mode, BigDawg has the liberty to run an operation at each local engine that contains data for the subquery and record the elapsed time in a BigDawg database. If a given (sub)query is run at these local engines, then BigDawg can identify the best location for this operation. Hence, training mode is effectively a training period where queries are run in multiple places and the best one location identified.

Since it is likely that many similar queries will be run over time, we construct a signature for each query and store these signatures in the BigDawg database noted above. Any similar queries we encounter in the future, which match an existing signature, do not need to use training mode, since the best location has already been

identified.

If the federation is in production mode, then the system simply chooses the storage engine arbitrarily for newly encountered queries. Over time, it will run the query with each of the other viable engines. Thus, the best location will be identified over a period of time rather than immediately before running the query.

We expect any given federation to be in production mode all the time or to start in training mode and then shift to production mode. In either case, we assemble a database of subqueries, their signatures, and their timings on various nodes. Over time, the system builds up a collection of queries, their signature, their elapsed time and what node ran them.

Chapter 2

System Architecture

The BigDawg system has four components: the query planning module (Planner), the performance monitoring module (Monitor), the data migration module (Migrator), and the query execution module (Executor). When BigDawg receives an incoming query, the Planner parses the query and creates a set of viable query plan trees with possible engines for each collection of objects. The Planner then passes these trees to the Monitor which uses existing performance information to determine the tree with the best engine for each collection of objects. This tree is given to the Executor which figures out how to best join the collections of objects and then executes the query, using the Migrator to move data from engine to engine when the plan calls for it. See Figure B-2 for a visual overview of the organization and workflow of the system.

In this section, I provide an overview of how each of the components works and focus on the design decisions of the Monitor.

2.1 Migrator

The Migrator provides efficient data migration between databases incorporated into BigDawg. The most generic approach to physical data migration is via the CSV format. Many databases support bulk loading as well as export of data in a CSV format. However, this process is compute bound, with parsing (finding new line and field delimiters) and deserialization (transformation from text to binary representa-

tion) constituting the biggest consumers of CPU cycles. CSV migration is the easiest approach but not the most efficient one.

Another approach is to migrate data using a binary format. Since there are many different binary formats used by databases, this approach requires intermediate binary transformation. Binary transformation also requires conversion of data types. For example, SciDB's data types represent a subset of PostgreSQL's data types and the mapping between the data types has to be specified explicitly.

Currently, the Migrator supports both variants and more testing needs to be done before determining the best format for each pair of databases. In addition to choosing the data format to migrate between databases, the Migrator also utilizes parallelization to decrease the loading time. It does this by exporting data in chunks from one database and loading them to another one in parallel.

The data migration process is also adaptive. After each migration, the Migrator reports fine-grained metrics to the Monitor. Later on, the Migrator retrieves the existing statistics and uses them to determine the optimal level of parallelism as well as the current best data format.

2.2 Planner

The Planner produces viable execution plans for a given query. A complete work cycle of the planner consists of three steps: analyzing the incoming query from the user, producing an optimized execution plan using the Monitor, and dispatching the plans to the Executer. For this section, I will discuss how the Planner handles intra-island queries. This is because the Planner breaks inter-island queries into constituent intra-island queries.

The Planner enumerates viable execution plans by iteratively constructing them into a tree structure which we denote as a Query Tree. In a Query Tree, each node represents either a piece of data or the result of a database-specific action (DSA), which entails executing a sub-query of the input query on a specific database instance. Examples of such sub-query include transmitting data, filtering according to

predicates and receiving and joining data. Edges represent the dependencies between the nodes.

To construct a forest of Query Trees that executes an input query, the Planner first converts the input query into a nested Abstract Syntax Tree (AST) according to the island scopes specified by the input query at each nested layer. Then the Planner breaks down the nested AST into sub-query tasks (such as filtering predicates, grouping relational tables by columns, or linear algebra functions). For each sub-query task, the Planner breaks the task into viable sets of DSAs and adds those DSAs as children nodes. Specifically, at the first iteration where the planner grows the Query Tree, the planner selects chunks of data from the innermost nested layer of the AST that relates to a viable DSA and makes partial Query Trees from those viable DSAs. At each subsequent iteration, the planner either combines two partial Query Trees from prior iterations to come up with a new partial Query Tree or adds a new DSA derived from the remaining sub-query tasks to a previously constructed Query Tree. Branches on the Query Trees that are strictly worse than others are pruned. For example, a branch is pruned if it ships data to an irrelevant database that does not provide extra functionality.

After constructing a forest of Query Trees, the Planner queries the monitor for performance information on each Query Tree. Based on the results from the Monitor, the Planner chooses the best Query Tree and sends it to the Executor to be executed. If no similar queries can be found in the Monitor, the Planner adds the query as a new benchmark for the Monitor. This way, if the query is encountered in the future, the Planner will have performance information for the query. If in training mode, the Monitor then tries each Query Tree and gives the Planner the fastest, while in production mode, the Planner chooses a Query Tree to run arbitrarily.

2.3 Executor

The Executor determines how to perform each cross engine query and executes a given query plan. This module takes as input a fully-formed Query Tree.

For a given Query Tree, the Executor begins by executing the leaf nodes of the tree which represent queries that depend on a single database system. As these actions complete, the Executor checks if the parents of the leaf nodes have had all of their dependencies satisfied. For any node that has its dependencies met, the Executor uses the Migrator to give the node the results of each of its dependencies.

After successfully migrating all necessary dependencies, the Executor determines how to perform the cross engine predicate on those dependencies. To this end, the Executor looks up the size of each operand and retrieves histogram information from the local system catalogs of each system. Using this information, it decides how to do each cross engine predicate. For joins, the Executor decides among moving the first operand, moving the second operand, using a bloom-filter semi-join, using a shuffle-join, etc. When the Executor has successfully run all of the nodes of the query plan tree, it sends the total running time to the Monitor.

2.4 Monitor

The Monitor maintains performance information on past intra-island queries, matches new intra-island queries to similar past queries, and stores migration metrics. For storing migration metrics, the Monitor simply maintains an API for storing and retrieving the metrics. The majority of the Monitor's functionality is devoted to query performance information.

Whenever BigDawg encounters a new intra-island query that is not similar to any previously seen queries, the Monitor creates an entry for each of the Query Trees generated by the Planner. Specifically, given an ordered list of Query Trees for a query, the Monitor stores the following for each Query Tree:

- The index of the Query Tree in the ordered list provided by the Planner.
- The query used to make the Query Trees.
- A signature for the query.
- The last time the Query Tree was run.

- The most recent cost, in terms of elapsed time, of running the Query Tree.

Since the Planner constructs Query Trees deterministically from a query together with a signature, the index of each Query Tree is fixed. Thus, it suffices to store the query, the signature, and the index to uniquely define each Query Tree.

To determine whether an incoming query is a new query or is similar enough to previous queries, the system uses signatures composed of the following:

- A tree representing the structure of the query (sig-1).
- A set of the objects referenced and the predicates involved (sig-2).
- A set of the constants in the query (sig-3).

Using these components, the system outputs a number from 0 to 1 representing how close the two signatures are where 0 means the two are identical and 1 means they are completely different. Specifically, the system first determines the tree edit distance, d, between the two queries' sig-1s using a robust tree edit distance algorithm [5]. It divides this distance by the maximum possible tree edit distance for the two sig-1s. To do this, the system finds the cost of constructing each tree, d_1 and d_2, from scratch. Using these 3 distances, the system computes $t_1 = \frac{d}{\max(d_1,d_2)}$, which is the tree edit distance divided by the maximum possible tree edit distance for the two sig-1s.

Next, the system determines the number of predicates, s, shared between the two queries' sig-2s as well as the maximum l_{max} of the number of predicates in both sig-2s. Using these numbers, it computes $t_2 = \frac{s}{l_{max}}$ which is the proportion of predicates shared between the two queries.

For sig-3, the Monitor determines the difference in the number of constants for the two queries' sig-3s. Since we expect changes in individual constants to not affect the relative ordering of Query Trees for a given query, we want to examine whether the number of constants remains unchanged. To do this the Monitor computes $t_3 = 1 - \frac{\min(a_1,a_2)}{\max(a_1,a_2)}$ where a_1 and a_2 are the number of constants in each respective query.

In other words, the system finds the query with less constants and divides its number of constants by that of the other query.

After computing all of these values, the Monitor returns the resulting value $v = \frac{\sum_i t_i * c_i}{\sum_i c_i}$ where c_i is a constant weight that can be set. Let two queries be similar if v is less than some constant and different otherwise.

Upon receiving a query from the Planner, the Monitor compares the new query's signature to that of each existing query. If the minimum v is such that the two queries are similar, the Monitor gives the Planner the performance information for the closest existing query. In the case that no existing queries are similar, the Monitor stores the query as a new benchmark.

If the Monitor is in training mode when a new query is added, the Monitor calls the Executor on each of the Query Trees for that query and stores those runtimes. In the case that the Monitor is in production mode, the Monitor stores each of the Query Trees without determining their runtimes and just initializes all of the runtimes to a null value. After adding the new query, the Monitor gives the Planner the runtimes for that query. In the case that all of the runtimes are null, the Planner should pick an arbitrary Query Tree to execute.

It is likely that as BigDawg continues to function, the performance of individual queries will change. This could happen for a variety of factors. Perhaps some of the storage engines scale with number of entries better than other storage engines, so as entries are added, the initial timings would no longer be valid. Another possibility is that many queries utilize a specific storage engine rather than other storage engines, making those engines overburdened. In order to address this problem, the Monitor periodically reruns existing queries to update their timings based on the current state of the system. To ensure that these timing updates do not interfere with normal BigDawg functions, the Monitor only reruns queries when it is not busy. Specifically, for each island, the Monitor checks periodically whether the load average of that island is under some threshold. If so, the Monitor finds a query in that island and reruns that query.

Currently, BigDawg is only running on a single physical system, so this feature of

the Monitor is very primitive. Ideally, the Monitor should maintain a task on each physical system that periodically reports the load average to the Monitor keyed by the island of that system. The Monitor should take all of these load averages to compute the load average for each island. Instead, the current Monitor simply checks the load average of the single physical system periodically and utilizes that as the load average of each island.

Chapter 3

Analysis

The Monitor should ideally speed up queries by determining the correct engine for each collection of objects. In this section, we evaluate whether the Monitor actually helps with query performance.

For this section, I use the following setup. I have two PostgreSQL instances on the same physical machine containing the MIMIC II dataset. One of the instances contains roughly half of the tables and the other instance contains the remaining tables. Table A.1 shows the size of each table as well as the PostgreSQL instance for each table. The focus of the analysis is determining if the Monitor matches signatures effectively. To this end, we want to determine if the Monitor should match queries where a table is replaced with a similar sized table as well as queries where a constant is replaced. Thus, the size of each of the tables is more important than their content. Similarly, for each of the queries tested, the choice of tables being joined is more important than the meaning of the queries. As a result, we will describe each of the test queries by the tables they join:

1. A query joining *icustayevents*, *labevents*, and *poe_order*. *icustayevents* is replaced with *icustay_detail* in some queries.

2. A query joining *additives*, *deliveries*, and *totalbalevents*. *additives* is replaced with *a_iodurations* in some queries.

3. A query joining *a_iodurations*, *ioevents*, and *deliveries*. *a_iodurations* is replaced with *a_meddurations* in some queries.

4. A query joining *comorbidity_scores*, *demographicevents*, and *procedureevents*. *comorbidity_scores* is replaced with *icd9* in some queries.

5. A query joining *icustay_detail*, *demographicevents*, *drgevents*, and *procedureevents*. *demographicevents* is replaced with *microbiologyevents* in some queries.

6. A query joining *icustay_days*, *medevents*, *a_iodurations*, and *deliveries*. *a_iodurations* is replaced with *a_meddurations* in some queries.

7. A query joining *d_patients*, *demographicevents*, *chartevents*, and *deliveries*. *demographicevents* is replaced with *microbiologyevents* in some queries.

8. A query joining *a_meddurations*, *d_chartitems*, *icustayevents*, and *totalbalevents*. *totalbalevents* is replaced with *medevents* in some queries.

9. A query joining *admissions*, *noteevents*, *a_chartdurations*, and *medevents*. *admissions* is replaced with *comorbidity_scores* in some queries.

10. A query joining *additives*, *deliveries*, *comorbidity_scores*, and *microbiologyevents*. *comorbidity_scores* is replaced with *admissions* in some queries.

3.1 Training Mode Gains

From Table A.2, we can see that running a query in training mode requires more than 3 times as long as running a query in production mode. This is because in training mode, the Monitor tries every Query Tree for a query before allowing the Executor to run the fastest Query Tree. Each of the queries generate 2 Query Trees, so intuitively, training mode should take at least 3 times as long as production mode for such queries.

Clearly, running a query once the Monitor already has training data for the query is faster than constructing the training data and then running the query. However, the

high cost of running a query in training mode begs the question of whether training mode is necessary in the first place. The "Avg Time Without Monitor" column of Table A.2 shows the time required to the run the query in production mode without any training data. In this case, since BigDawg does not have any information on which Query Tree performs better, it randomly selects a Query Tree. While the cost of running the query in production mode varies a lot compared to that of without the Monitor, we can see that in the best case, we can run a query in about 2/3 as much time and in the worst case, we can run a query in about the same amount of time. Thus, assuming queries are often rerun or we see similar queries often, running a query in training mode initially can save a lot of time later on.

3.2 Matching Signatures

One of the main components of the Monitor module is how it matches previous queries to new queries and uses the performance information of those previous queries to determine the optimal query plan for the new queries. To test this, for each of our 10 queries, we constructed the following 4 types of similar queries:

- A query where the order of the predicates of the original query are changed.
- A query where one of the constants of the query is replaced with a similar constant. We measure similarity by looking at the number of entries in each table that has that constant. Two constants are similar if the number of entries is approximately the same for both constants.
- A query where one of the constants of the query is replaced with a dissimilar constant.
- A query where one of the tables involved in the query is replaced with a similar sized table.

Tables A.3-A.12 shows the time to run each Query Tree for the 10 test queries as well as any similar queries. I also applied load to the testing environment while

running queries to see how the runtime of the queries changed. Specifically, I ran multiple queries in the background on PostgreSQL instance 1 for one of the loads, and ran queries on PostgreSQL instance 2 for the other load. This is represented by the load 1 and load 2 columns respectively.

If the Monitor matches queries correctly, it should recognize that changing the order of the predicates does not actually change the query. During my testing, I found that queries where the order of the predicates are changed are always matched correctly. Specifically, the Monitor believes that $v = 0$ for such queries as expected. Furthermore, examining Tables A.3-A.12, we can see that changing the order of predicates does not impact the runtime of the query as expected. Thus, for queries where the structure of the query changed but all else is identical, the Monitor is able to match them correctly with existing queries.

In general, the Monitor should not expect queries where a table has been swapped to perform similarly to the original queries. This is because the Monitor does not have any context on which tables are actually similar other than through the size of the tables and the names of the tables. While it is possible that similar sized tables have similar data, the Monitor has no way of ensuring this. Furthermore, it is difficult to identify whether two tables are similar through their names.

Examining Tables A.3-A.12, we can see that while swapping a table sometimes results in similar runtimes, it can also result in completely disparate runtimes. Table A.7 shows an example of how changing the table can result in completely different outcomes. While initially Query Tree 2 performs better than Query Tree 1, after swapping tables, we can see the opposite is true. A.10-A.12 also provide examples of where swapping a table can change which Query Tree is best. Currently, the Monitor does not match queries where a table was changed for any of the 10 queries examined.

Currently, the Monitor matches all queries where the constants in the query are changed. It seems obvious that replacing constants with similar constants should result in nearly unchanged runtimes, and my results in Tables A.3-A.12 confirms this. In general, replacing a constant with a similar constant resulted in very similar runtimes for all of the test queries.

While replacing constants with similar constants will not affect the runtime significantly, it is unclear whether this would also be the case with skewed constants. Examining Tables A.3-A.12, we can see that in general the relative order of Query Trees is preserved with skewed constants. Queries where both Query Trees run in approximately the same amount of time, as shown in tables Tables A.5 and A.6, continue to do so after swapping constants. Similarly, when Query Tree 1 runs faster than Query Tree 2, such as in Tables A.4 and A.7-A.9, this continues to be the case with a skewed constant. Furthermore, the ratio between the runtimes of the 2 Query Trees tends to be similar after replacement with a skewed constant. For example, the ratio between Query Trees in Table A.10 is 0.45 initially and 0.4 after using a skewed constant. The ratio noticeably differs in Tables A.3, A.4, and A.9. However, for these 3 queries, either the initial query or the query after replacement ran in a trivial period of time. Thus it is possible, that when the query is too simple, the differences in the runtimes between Query Trees are dominated by constant factors. To account for this, as long as users initially train the Monitor on queries that take a non-trivial amount of time, replacing constants with skewed constants should result in queries that exhibit either similar behavior or run in a trivial amount of time regardless of which Query Tree is chosen.

Overall, the Monitor matches queries where constants have been replaced or when the order of predicates of the queries are changed. Based on my results, for all of these matched queries, the Monitor provides useful metrics as long as the initial benchmark queries run in non-trivial amounts of time. For queries where a table has been replaced, the Monitor cannot reliable provide useful metrics and thus does not match such queries.

3.3 Response to Environmental Changes

Table A.3 shows an example of how the relative order of Query Trees can change depending on load. While Query Tree 1 initially runs faster than Query Tree 2 (265 ms compared to 296 ms), after applying load to PostgreSQL instance 1, the opposite

is true (378 ms compared to 293 ms). Since the performance of individual queries will change as BigDawg continues to function, it is necessary to evaluate whether the Monitor adapts to changes in the environment.

In general, the Monitor is able to update its performance metrics for its queries over time. After applying load, I observed that the Monitor adapted within a few seconds. However, the rate at which the Monitor adapts linearly increases with the number of queries in the system. That is, the more queries stored by the Monitor, the longer it takes for a given query to stay up to date. This is because the way the Monitor updates its performance metrics is by re-running the least recently updated query whenever the load average is under some threshold. Thus, if the environment changes faster than the rate that the Monitor can run all of its queries, the Monitor will always provide outdated metrics. This would be problematic for queries where several Query Trees perform similarly. For such queries, it is possible that the Monitor will always rank the Query Trees incorrectly if it is always outdated. For queries where some Query Trees perform vastly better than others, it is unlikely that changes in the environment can cause the better performing queries to become significantly worse than the previously underperforming queries.

3.4 Future Work

While the Monitor is effective for the existing environment, there is still room for improvement. The main avenues of improvement are reducing overhead and extending the Monitor's ability to respond to environmental changes.

Currently, the Monitor finds matching queries fairly inefficiently. Specifically, the Monitor finds the distance, v, between every unique query in the system and the incoming query. It is clear that the overhead necessary to match an incoming query will increase linearly with the number of benchmark queries supported by the Monitor. While this is not a large problem at the moment since we have very few queries, this can become a much bigger problem in the future. One possible way to address this is by keying queries by one of the signatures, such as sig-1, and only determining

v for queries that match that signature. The downside of this is method is that it is possible that the Monitor can provide useful metrics even for queries that differ on that signature. More testing will need to be done to determine if there are any signatures that need to be identical between queries for the queries to be considered similar.

Another problem with the Monitor is that it is possible for the Monitor to always provide outdated metrics. In order to deal with this problem, it might be a good idea to select old queries to run in a different order than least recently updated. Instead, one could try choosing old queries to run in a randomized manner, weighted by time since last updated. More testing needs to be done to determine the best order to select queries to re-run. Furthermore, in line with re-running old queries, the way the Monitor determines the load average of the system needs to be extended as the number of physical systems increases. This is described in more detail at the end of the Monitor section of Chapter 2.

Appendix A

Tables

Table A.1: MIMIC II Tables

PostgreSQL Instance	Table Name	Table Size
1	censusevents	284
1	d_careunits	22
1	d_chartitems	4832
1	d_codeditems	3339
1	d_demographicitems	88
1	deliveries	874
1	demographicevents	1069
1	microbiologyevents	3157
1	noteevents	6566
1	procedureevents	989
1	ioevents	106491
1	labevents	153025
1	medevents	51157
1	poe_med	16161
1	poe_order	13286
1	totalbalevents	14826
2	a_chartdurations	43713
2	a_iodurations	4703
2	a_meddurations	2611
2	additives	1170
2	admissions	181
2	chartevents	1385468
2	comorbidity_scores	181
2	d_patients	143
2	demographic_detail	181
2	drgevents	181
2	icd9	1966
2	icustay_days	1442
2	icustay_detail	219
2	icustayevents	219

Table A.2: Training vs Production Mode

Query No.	Training Mode Time	Production Mode Time	Time Without Monitor
1	826 ms	265 ms	281 ms
2	882 ms	190 ms	346 ms
3	62539 ms	20559 ms	20990 ms
4	491 ms	160 ms	166 ms
5	6592 ms	1977 ms	2308 ms
6	24294 ms	6146 ms	9074 ms
7	28165 ms	7648 ms	10259 ms
8	19073 ms	4496 ms	7289 ms
9	15806 ms	4652 ms	5577 ms
10	78487 ms	23496 ms	27496 ms

Table A.3: Query 1 Average Runtimes

	No Load		Load 1		Load 2	
	Tree 1	Tree 2	Tree 1	Tree 2	Tree 1	Tree 2
Base query	265 ms	296 ms	378 ms	293 ms	359 ms	414 ms
Predicate order	254 ms	296 ms	387 ms	309 ms	335 ms	406 ms
Similar constant	278 ms	290 ms	369 ms	312 ms	355 ms	431 ms
Skewed constant	346 ms	906 ms	448 ms	1055 ms	514 ms	1170 ms
Table swap	542 ms	619 ms	723 ms	727 ms	620 ms	751 ms

Table A.4: Query 2 Average Runtimes

	No Load		Load 1		Load 2	
	Tree 1	Tree 2	Tree 1	Tree 2	Tree 1	Tree 2
Base query	502 ms	190 ms	669 ms	260 ms	707 ms	225 ms
Predicate order	475 ms	169 ms	685 ms	277 ms	718 ms	213 ms
Similar constant	479 ms	206 ms	738 ms	334 ms	610 ms	212 ms
Skewed constant	93023 ms	92321 ms	96560 ms	95412 ms	96903 ms	96430 ms
Table swap	1156 ms	180 ms	1548 ms	219 ms	1620 ms	301 ms

Table A.5: Query 3 Average Runtimes

	No Load		Load 1		Load 2	
	Tree 1	Tree 2	Tree 1	Tree 2	Tree 1	Tree 2
Base query	20599 ms	21421 ms	21443 ms	22267 ms	27003 ms	27201 ms
Predicate order	20047 ms	21309 ms	21985 ms	23099 ms	27003 ms	27321 ms
Similar constant	20402 ms	21965 ms	21151 ms	22098 ms	26327 ms	26921 ms
Skewed constant	411 ms	413 ms	578 ms	576 ms	641 ms	656 ms
Table swap	4049 ms	3902 ms	4922 ms	4643 ms	4723 ms	4759 ms

Table A.6: Query 4 Average Runtimes

	No Load		Load 1		Load 2	
	Tree 1	Tree 2	Tree 1	Tree 2	Tree 1	Tree 2
Base query	171 ms	160 ms	367 ms	368 ms	330 ms	318 ms
Predicate order	165 ms	156 ms	356 ms	356 ms	331 ms	322 ms
Similar constant	180 ms	180 ms	268 ms	249 ms	268 ms	264 ms
Skewed constant	226 ms	210 ms	547 ms	523 ms	547 ms	543 ms
Table swap	620 ms	621 ms	898 ms	889 ms	919 ms	939 ms

Table A.7: Query 5 Average Runtimes

	No Load		Load 1		Load 2	
	Tree 1	Tree 2	Tree 1	Tree 2	Tree 1	Tree 2
Base query	2638 ms	1977 ms	4551 ms	2901 ms	3736 ms	3109 ms
Predicate order	2563 ms	2054 ms	4342 ms	2869 ms	3834 ms	3134 ms
Similar constant	2625 ms	2015 ms	4043 ms	2916 ms	3801 ms	2815 ms
Skewed constant	4486 ms	3851 ms	5098 ms	4512 ms	5374 ms	4828 ms
Table swap	29781 ms	32474 ms	32132 ms	36428 ms	32244 ms	39086 ms

Table A.8: Query 6 Average Runtimes

	No Load		Load 1		Load 2	
	Tree 1	Tree 2	Tree 1	Tree 2	Tree 1	Tree 2
Base query	12002 ms	6146 ms	12880 ms	6664 ms	12629 ms	7032 ms
Predicate order	12694 ms	6177 ms	12944 ms	6644 ms	12704 ms	6985 ms
Similar constant	12312 ms	6102 ms	12706 ms	6598 ms	12712 ms	7210 ms
Skewed constant	147256 ms	80089 ms	169093 ms	82895 ms	182392 ms	83093 ms
Table swap	7991 ms	3304 ms	8221 ms	3432 ms	8223 ms	3512 ms

Table A.9: Query 7 Average Runtimes

	No Load		Load 1		Load 2	
	Tree 1	Tree 2	Tree 1	Tree 2	Tree 1	Tree 2
Base query	12869 ms	7648 ms	13093 ms	7720 ms	13505 ms	7883 ms
Predicate order	12754 ms	7678 ms	13042 ms	7842 ms	13524 ms	7884 ms
Similar constant	12827 ms	7680 ms	13097 ms	7696 ms	13458 ms	7869 ms
Skewed constant	323 ms	275 ms	456 ms	320 ms	431 ms	403 ms
Table swap	2481 ms	1272 ms	2601 ms	1350 ms	2620 ms	1502 ms

Table A.10: Query 8 Average Runtimes

	No Load		Load 1		Load 2	
	Tree 1	Tree 2	Tree 1	Tree 2	Tree 1	Tree 2
Base query	4496 ms	10081 ms	4750 ms	10600 ms	4519 ms	10995 ms
Predicate order	4483 ms	10335 ms	4652 ms	11017 ms	4504 ms	11036 ms
Similar constant	4425 ms	10383 ms	4705 ms	10006 ms	4514 ms	11333ms
Skewed constant	8456 ms	21385 ms	8828 ms	23345 ms	8613 ms	24537 ms
Table swap	664 ms	317 ms	922 ms	545 ms	832 ms	587 ms

Table A.11: Query 9 Average Runtimes

	No Load		Load 1		Load 2	
	Tree 1	Tree 2	Tree 1	Tree 2	Tree 1	Tree 2
Base query	4652 ms	6502 ms	4923 ms	6654 ms	4832 ms	6893 ms
Predicate order	4753 ms	6634 ms	5343 ms	7098 ms	5208 ms	7129 ms
Similar constant	4423 ms	6742 ms	4828 ms	6928 ms	4787 ms	7188 ms
Skewed constant	32454 ms	67121 ms	34852 ms	68323 ms	33947 ms	68932 ms
Table swap	1647 ms	1165 ms	1923 ms	1239 ms	1832 ms	1387 ms

Table A.12: Query 10 Average Runtimes

	No Load		Load 1		Load 2	
	Tree 1	Tree 2	Tree 1	Tree 2	Tree 1	Tree 2
Base query	31495 ms	23496 ms	34293 ms	24933 ms	33298 ms	25209 ms
Predicate order	30938 ms	23922 ms	35980 ms	25096 ms	34454 ms	26748 ms
Similar constant	28381 ms	20487 ms	30092 ms	21933 ms	30212 ms	22117 ms
Skewed constant	4201 ms	3017 ms	4821 ms	3428 ms	4782 ms	3689 ms
Table swap	6029 ms	15991 ms	6492 ms	16021 ms	6332 ms	16118 ms

Appendix B

Figures

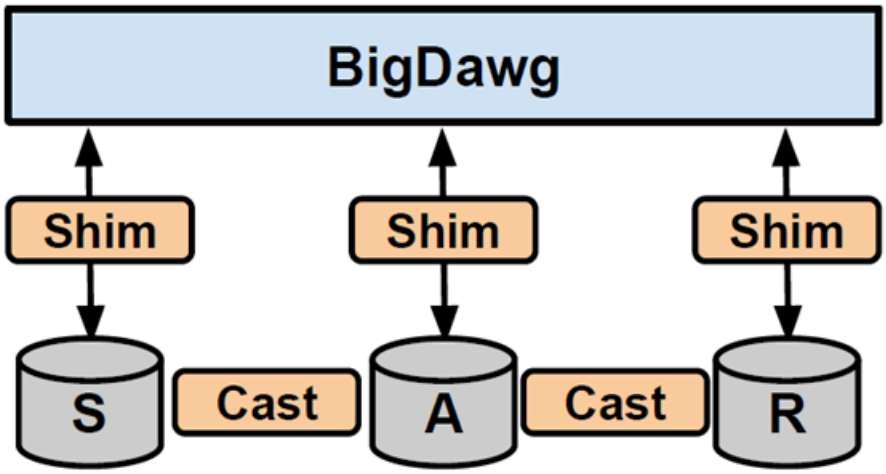

Figure B-1: Multi-Island Data Federation

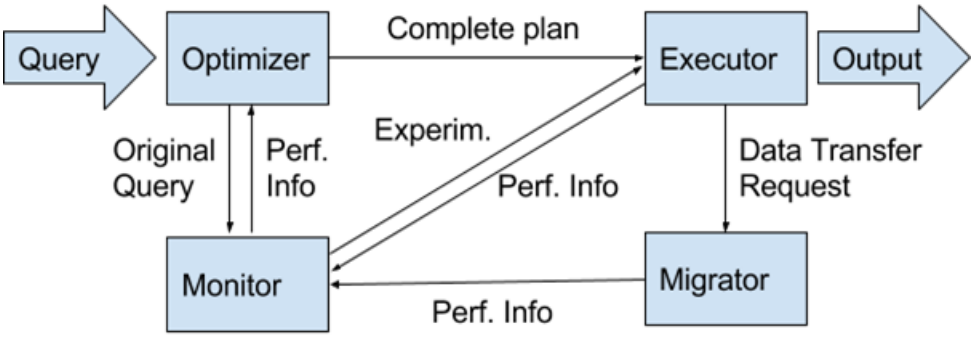

Figure B-2: BigDawg Workflow

[1] M. Saeed, M. Villarroel, A. T. Reisner, G. Clifford, L.-W. Lehman, G. Moody, T. Heldt, T. H. Kyaw, B. Moody, and R. G. Mark. Multiparameter intelligent monitoring in intensive care ii (mimic-ii): A public-access intensive care unit database. Critical Care Medicine, 39:952-960, May 2011.

[2] J. Duggan, A. Elmore, T. Kraska, S. Madden, T. Mattson, and M. Stonebraker. The BigDawg Architecture and Reference Implementation. 2015.

[3] D. Halperin, V. T. de Almeida, L. L. Choo, S. Chu, et al. Demonstration of the myria big data management service. In SIGMOD, pages 881-884, 2014.

[4] J. Kepner, W. Arcand, W. Bergeron, N. Bliss, et al. Dynamic distributed dimensional data model (d4m) database and computation system. In ICASSP, pages 5349-5352. IEEE, 2012.

[5] M. Pawlik and N. Augsten. RTED: A Robust Algorithm for the Tree Edit Distance. Proceedings of the VLDB Endowment (PVLDB), Vol. 5, No. 4, pp. 334-345, 2011.