Database Engine Integration and Performance Analysis of the BigDAWG Polystore System

by

Katherine Yu

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

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author .....
Department of Electrical Engineering and Computer Science
May 26, 2017

Certified by .....
Michael Stonebraker
Professor, EECS
Thesis Supervisor

Accepted by .....
Christopher Terman
Chairman, Department Committee on Graduate Theses

Database Engine Integration and Performance Analysis of the BigDAWG Polystore System

by

Katherine Yu

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

The BigDAWG polystore database system aims to address workloads dealing with large, heterogeneous datasets. The need for such a system is motivated by an increase in Big Data applications dealing with disparate types of data, from large scale analytics to realtime data streams to text-based records, each suited for different storage engines. These applications often perform cross-engine queries on correlated data, resulting in complex query planning, data migration, and execution. One such application is a medical application built by the Intel Science and Technology Center (ISTC) on data collected from an intensive care unit (ICU). This thesis presents work done to add support for two commonly used database engines, Vertica and MySQL, to the BigDAWG system, as well as results and analysis from performance evaluation of the system using the TPC-H benchmark.

Thesis Supervisor: Michael Stonebraker
Title: Professor, EECS

Acknowledgments

I would like to express my gratitude towards my master's thesis advisor, Professor Michael Stonebraker, for providing guidance on my project this past year. I would also like to thank Dr. Vijay Gadepally, for his consistent support and guidance on my research.

I would also like to thank Zuohao She, Kyle O'Brien, Adam Dziedzic, and others on the BigDAWG development team for their help and work on the BigDAWG system, without which this work would not be possible.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

In the past decade, there has been a lot of development of database management systems specialized for narrower use cases, for example, relational column stores for data warehousing, in memory SQL systems for online transaction processing (OLTP) workloads, and NoSQL database engines for flexible data formats such as JSON, exemplifying the philosophy that “no one size fits all" for database management systems[18]. These specialized engines provide a performance benefit of several orders of magnitude compared to a single DBMS that tries to function as a catch-all for disparate types of data.

As a case study of an application with a complex workload, we consider a hospital application designed to handle the MIMIC II dataset[15], a publically available dataset containing 26,000 ICU admissions at the Beth Israel Deaconess Hospital in Boston. The dataset contains patient metadata, free-form text data (notes taken by medical professionals), semi-structured data (lab results and prescriptions), and waveform data (measurements on vitals like pulse and heartrate). Due to the variety in data sources, the application must rely on multiple different data access methods; for instance, an administrator might want to query the number of patients currently in the hospital with standard SQL, compute complex analytics on patient waveform data, and even perform text search on patient profiles for free-form data.

The Big Data Analytics Working Group (BigDAWG) system [7][9] has been developed in an effort to provide a unified interface for these disparate data models,

database engines and programming models. The current reference implementation developed around the MIMIC II dataset uses SciDB[17] for time-series data, Apache Accumulo[1] for freeform text notes, and Postgres[19] for clinical data. Complex datasets with correlated data will often require support for cross-engine queries. Some examples of such queries in the reference implementation include querying both SciDB and Postgres to locate patients with irregular heart rhythms, and querying Accumulo and Postgres to find doctor's notes to find doctor's notes associated with a particular prescription drug. The BigDAWG system provides the functionality to develop query plans, execute queries across database engines, and return the results to the user.

Currently, the BigDAWG system only supports a limited subset of engines including Postgres, SciDB, and Accumulo. However, there are myriad other database engines currently available, each suited for different use cases. I worked on making the BigDAWG polystore system compatible with two widely used database engines: MySQL and Vertica. Both database engines use the same relational interface as Postgres, so I was able to build on existing components to integrate them.

The goal of this piece of my research is to serve as a model for future same-island database integrations. Currently, the engines that are currently supported by BigDAWG were all integrated by other researchers on this project. However, it is not feasible to anticipate every database engine that users want support for. This work provides guidelines for how a user should modify BigDAWG to be compatible with a new database engine of their choice under the relational island.

Another goal is to analyze BigDAWG's performance for queries across multiple different database engines. While there has been some preliminary evaluation of the reference implementation on the MIMIC II dataset[10], the benchmarks were basic and use a setup with only two instances of PostgreSQL, which is not very interesting. In adding support for additional engines, I aimed to verify BigDAWG's premise: by leveraging the relative strengths of different database engines, certain queries will perform better under BigDAWG than on a single database engine. I detail results and analysis of performance testing on the BigDAWG system later in this report.

The rest of this thesis is organized as follows. In Chapter 2, I describe the Big-

DAWG architecture and modules. In Chapter 3, I detail the process for integrating new database engines into the BigDAWG system. In Chapter 4, I discuss the performance experiments I conducted and their associated results and analysis. In Chapter 5, I discuss future areas of work and potential improvements to the BigDAWG system based on my results. In Chapter 6, I conclude.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

BigDAWG architecture

In this chapter, I describe the design and architecture of the BigDAWG system. The BigDAWG system[10] is comprised of four layers: database engines, islands that provide data model abstraction, a middleware and API, and applications. The database engine layer consists of possibly many distinct database and storage engines, each with their own relative strengths for handling different workloads. The island layer provides a user-facing abstraction that informs the underlying engines how to interpret part of a query. There can be multiple database engines under a particular island, and a particular database engine can have differing functionality under multiple islands.

2.1 BigDAWG islands

The islands of information are another important part of the BigDAWG polystore system. The islands serve as an abstraction to the client, each with its own query language, data model, and connectors, or *shims*, to each underlying database engine[16]. In this way, clients do not need to know anything about the underlying engines, and just need to use the common BigDAWG syntax to issue queries. The client can also instruct BigDAWG to cast data from one island to another, but that is outside the scope of this thesis. I focus primarily on workloads that deal with different engines under the relational island, which implements a subset of the Structured Query Language[12] specification.

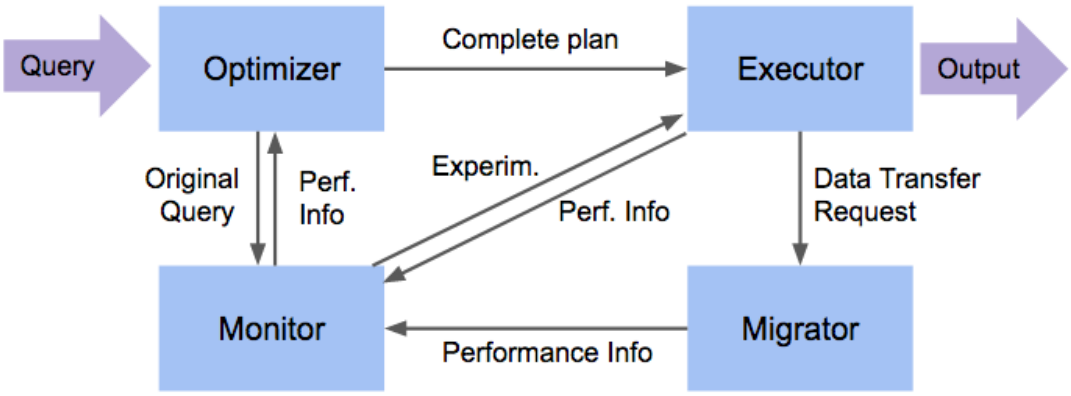

Figure 2-1: Data flow across BigDAWG middleware architecture.

2.2 BigDAWG middleware

My thesis primarily pertains to the BigDAWG middleware layer, which receives client requests and interacts with the underlying database engines themselves. It is comprised of four main modules: The query planning module (Planner), the query execution module (Executor), the performance monitoring module (Monitor), and the data migration module (Migrator), as shown in Figure 2-1.

2.2.1 Planner

The Planner parses incoming client queries and constructing logical query plans. These plans are in the form of dependency trees, where each node represents a task such as querying a particular database engine or performing a join. The root node in the plan represents the computation of the final query result. The planner also interfaces with the performance monitor to get historical performance data for certain queries and data migrations to help determine the optimal query plan for a particular query. Based on the response from the Monitor, the planner selects the best candidate query plan and dispatches it to the Executor to be executed.

For queries targeting the relational island, the Planner sends the query to a dedicated Postgres instance that contains all the table schemas to generate a parse tree for the query. The Planner then uses the parse tree returned by Postgres to generate a

tree of dependencies, in which nodes may be combined or re-ordered for optimization purposes.

2.2.2 Monitor

The Monitor[5] records performance information for past queries, which is then used to aid the planner in selecting query plans for future similar queries. It uses a signature-based scheme to determine how similar two queries are. These signatures are comprised of three main components: a tree representing the structure of the query (sig-1), a set of the objects referenced and the predicates involved (sig-2), and a set of the constants in the query (sig-3). When the Monitor receives query plan from the Planner, it computes a signature for it and compares it to previous queries it has recorded. If it determines that the queries are similar enough, then it returns historical performance data back to the Planner. Otherwise, it logs the query as new information.

2.2.3 Executor

The Executor[11] determines the best way to physically execute the query plan provided by the Planner. It starts by executing the leaf nodes of the plan, and attempts to execute as many nodes in parallel as possible, moving up the tree as dependencies are satisfied. The Executor uses the Migrator module to move records from intermediate queries from one database engine to another in the case of cross-engine queries.

2.2.4 Migrator

The Migrator moves data between database engines in BigDAWG.The format used for most migrations between pairs of databases is in CSV format. The advantage to using CSV is that most database systems provide support exporting and loading data in this format. Some database engines, such as Postgres, Vertica and SciDB, also support a binary data format. Migration using binary data requires an explicit

mapping between datatypes of the two participating engines, and is not as widely supported by database systems as CSV, but is much more performant. The Migrator also reports performance metrics from each migration to the Monitor so that they can be used to evaluate future query plans.

Chapter 3

Integrating new Database Engines into BigDAWG

One of the main components of this thesis is the addition of support for two database engines, MySQL and Vertica, to the BigDAWG system. This chapter discusses the choice of the two engines, as well as details for how I implemented support for these engines.

3.1 Database engines used

MySQL[3] is an open-source relational database, commonly used in industry by companies such as Facebook, Twitter, and Ebay. It was a natural choice to start by integrating MySQL support into BigDAWG, as it implements a subset of the SQL standard, and is similar to Postgres in its architecture, and is a popularly used alternative due to its speed.

Vertica[13] is the commerical version of the C-Store research prototype[20]. While it shares the same relational interface as MySQL and Postgres, its architecture is fundamentally different in that records are stored as *projections*, or sorted subsets of the attributes of a table, rather than tuples of attributes. This allows Vertica to use a variety of encoding schemes on the data to significantly improve space efficiency and performance for analytic workloads.

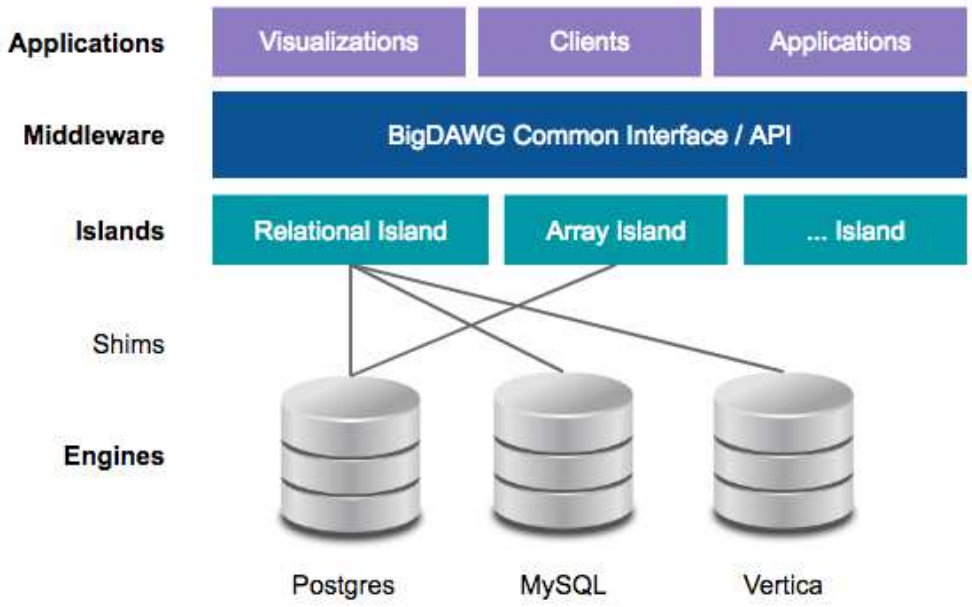

Figure 3-1: A visualization of the BigDAWG architecture, including MySQL and Vertica.

I chose both of these engines because of their support for the SQL standard, which is a superset of the operations handled by the relational island.

3.2 Implementation details

In this section, I detail the implementation of a few main components, enumerated below, for the BigDAWG system to successfully interact with the database engines. The BigDAWG middleware component is primarily written in Java, so this section will be described in the context of the language.

3.2.1 Connection to the database

When adding a new engine, one must implement a few interfaces that tell BigDAWG how to interact with the database. Primarily, one must implement the DBHandler interface, which is comprised of common operations that are required for BigDAWG to execute queries on the underlying engines. Both Vertica and MySQL have associated

JDBC[14] drivers, which allow developers to connect to each of these engines in Java-based programs.

Since most database engines each have their own unique sets of supported operations and nuances in syntax, it is necessary to let BigDAWG know how to perform operations such as creating tables to store intermediate query results, dropping tables to remove these results when it is done performing a query plan, and importing and exporting data for migration.

3.2.2 Query generation

BigDAWG must also know how to translate between its own syntax to the underlying syntax for each engine it supports. This is done through the SQLQueryGenerator class, which parses the client query and converts it into a version that can be executed directly on the underlying database engine. Since Vertica shares the same SQL parser as Postgres, it was able to use the existing class without any changes. MySQL, however, differs from Postgres in its syntax for certain operations such as `SELECT INTO`, and thus it is necessary to modify the QueryGenerator to take this into account when generating queries to be performed on MySQL.

3.2.3 Migration

When a client issues a query involving a cross-engine join, it is necessary to move tuples from one engine to another in order to compute the final result. To do so, I implemented Migrator classes between Postgres and each of the two engines. Each Migrator class facilitates the transfer of data in a single direction between a pair of database engines. Because binary data loading requires implementing a module that will translate data from the source database's binary format to that of the target database, I chose to export and load data in CSV format, which is the most widely used format by database systems for importing and exporting data and did not require translation beyond the data type conversions in the schema. These Migrators are used by the Executor when executing the query plan generated by the Planner.

Currently, my implementation only supports two-way migration between Postgres and Vertica, and Postgres and MySQL. BigDAWG is currently designed such that for every ordered pair of databases, there is a unique Migrator class that performs the migration, due to the specialized attributes of each database. The downside to this design choice is that the number of Migrator classes scales quadratically with the number of supported engines. In future iterations, we could generalize the migrator component to reduce this overhead. Another possibility is a two-stage migration, in which one engine, for example Postgres, serves as an intermediary between two other databases, so the number of Migrators scales linearly instead of quadratically, at the expense of increased migration execution time.

3.2.4 The BigDAWG catalog and schemas

Lastly, BigDAWG stores metadata about what database engines are available, and what tables are located on which engine in a catalog database (we currently use Postgres). It is necessary to insert metadata about each database engine into the catalog so BigDAWG can connect to them to issue queries and knows what tables exist. This metadata includes connection information (hostname, user, password, port), the island or islands that the engine can be queried through, and the tables located on that engine.

BigDAWG also stores schemas for each table in a separate database on the same Postgres instance. These schemas are used by the Planner when constructing query plans.

Chapter 4

BigDAWG performance analysis

In this chapter, I discuss and analyze benchmark results and their implications going forward with BigDAWG. First I cover the TPC-H benchmark, then I discuss the experimental setup used, results from the experiments, and analysis of the results.

4.1 The TPC-H benchmark

TPC-H[4] is an industry standard decision support benchmark developed by the Transaction Processing Performance Council (TPC) that models a an industry involved with managing and distributing products worldwide. It has been widely used in industry by database vendors as well as researchers to evaluate database systems. The benchmark is comprised of a set of business-related analytical queries.

Ultimately, I decided to use queries from the TPC-H benchmark for evaluation purposes. It seemed sufficiently generalizable to the BigDAWG system, and provides a lot of built-in tools to aid users in generating queries and datasets. I did not follow the strict TPC-H specification for how to run the tests, because it is mainly targeted towards commerical database vendors running on high-end hardware. Furthermore, only a subset of the queries were compatible with BigDAWG, and small modifications needed to be made in order to execute a few of the remaining queries, as BigDAWG does not fully support certain syntax such as `ORDER BY`s using aliases. A full list of queries used can be found in Appendix A.

Other benchmarks I considered include the TPC-C benchmark, another widely-used benchmark developed by the TPC. However, TPC-C was inferior for my purposes because it is both older than TPC-H and does not provide tools for data and query generation. I also considered using the Yahoo! Cloud Serving Benchmark (YCSB), which has been developed to evaluate "cloud OLTP" systems, such as BigTable, PNUTS, Cassandra, or HBase, whose workload might differ from traditional workloads that are simulated by TPC-C[6]. Since it is designed with cloud serving systems in mind, it ultimately did not seem like a good fit for our use case as it focused more on evaluating scalability than speed.

4.2 Experimental Setup

The following performance tests were run on a machine running Ubuntu 14.04, with 64GB of RAM and 24 2GHz virtual processor cores. Each of the experiments was conducted using an installation of the BigDAWG middleware consisting of virtualized instances of the relevant database engines using Docker[2]. The databases used were not tuned for performance in any particular way. For simplicity, I use a single-node installation of Vertica.

I generated data using the `tpchgen` tool, with a scaling factors of 10 and 100, resuling in 10GB and 100GB of data, respectively. The relative size of each of the tables is shown in Table B.1 and Table B.2.

The full dataset was loaded into each of the database engines, with the catalog modified accordingly to inform BigDAWG what locations to use for each table for each experiment. I primarily chose to focus on Postgres and Vertica because Postgres was used in the initial BigDAWG development process, and Vertica differs much more from Postgres compared to MySQL in its architecture.

4.3 Single-engine queries through BigDAWG

In this section, I discuss the results of evaluating the performance of queries from the TPC-H benchmark when querying a single database engine directly, or through the BigDAWG middleware. The results show that the overhead from querying through BigDAWG is minimal, and may even improve performance in certain cases.

Figures 4-1 and 4-2 show the relative performance of the selected TPC-H queries on a single instance of Postgres and a single instance of Vertica, respectively. Since Postgres performed worse than Vertica on the workload, the overhead incurred by using BigDAWG was comparatively smaller, typically adding less than 1 percent to the overall runtime, as shown in Figure 4-1. For Vertica, the overhead added an extra 5 to 10 percent in overall runtime on the 10GB dataset, because Vertica performed significantly better overall. The overhead incurred by querying through BigDAWG remains relatively constant with respect to the database engine used, as the overhead shrinks proportionally when the dataset was increased to 100GB in Figure 4-2. The underlying data can be found in Appendix B.

We can also see that some queries were actually executed faster when run through BigDAWG than when Vertica was queried directly. Because TPC-H aims to emulate queries used in business analysis of an ad-hoc nature, there are portions of the queries that are randomized, such as the date ranges used. For example, Query 5 (as seen in A.3), which shows the most dramatic improvement under BigDAWG for Vertica, adds an interval of 1 year to the date range to query over. During the parsing process, BigDAWG rewrites the original query to compute the resulting date before executing the query on Vertica directly, which actually results in a shorter execution time overall.

4.4 Cross-engine queries through BigDAWG

While BigDAWG performs well with a single engine, we are primarily interested in how BigDAWG performs in a multi-engine scenario. I instrumented the middleware

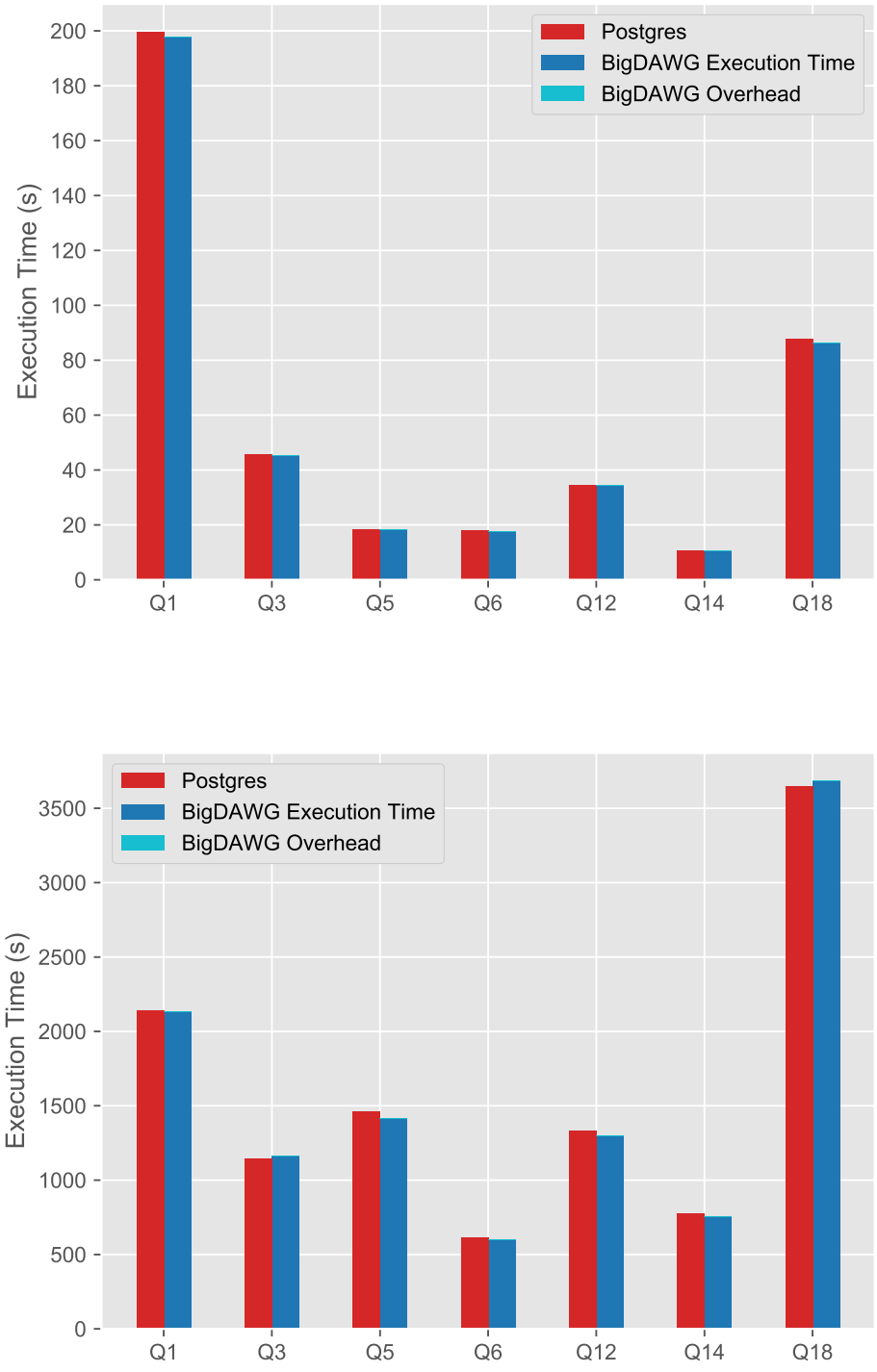

Figure 4-1: TPC-H query performance for a single Postgres instance for 10GB and 100GB of data, respectively, with and without BigDAWG.

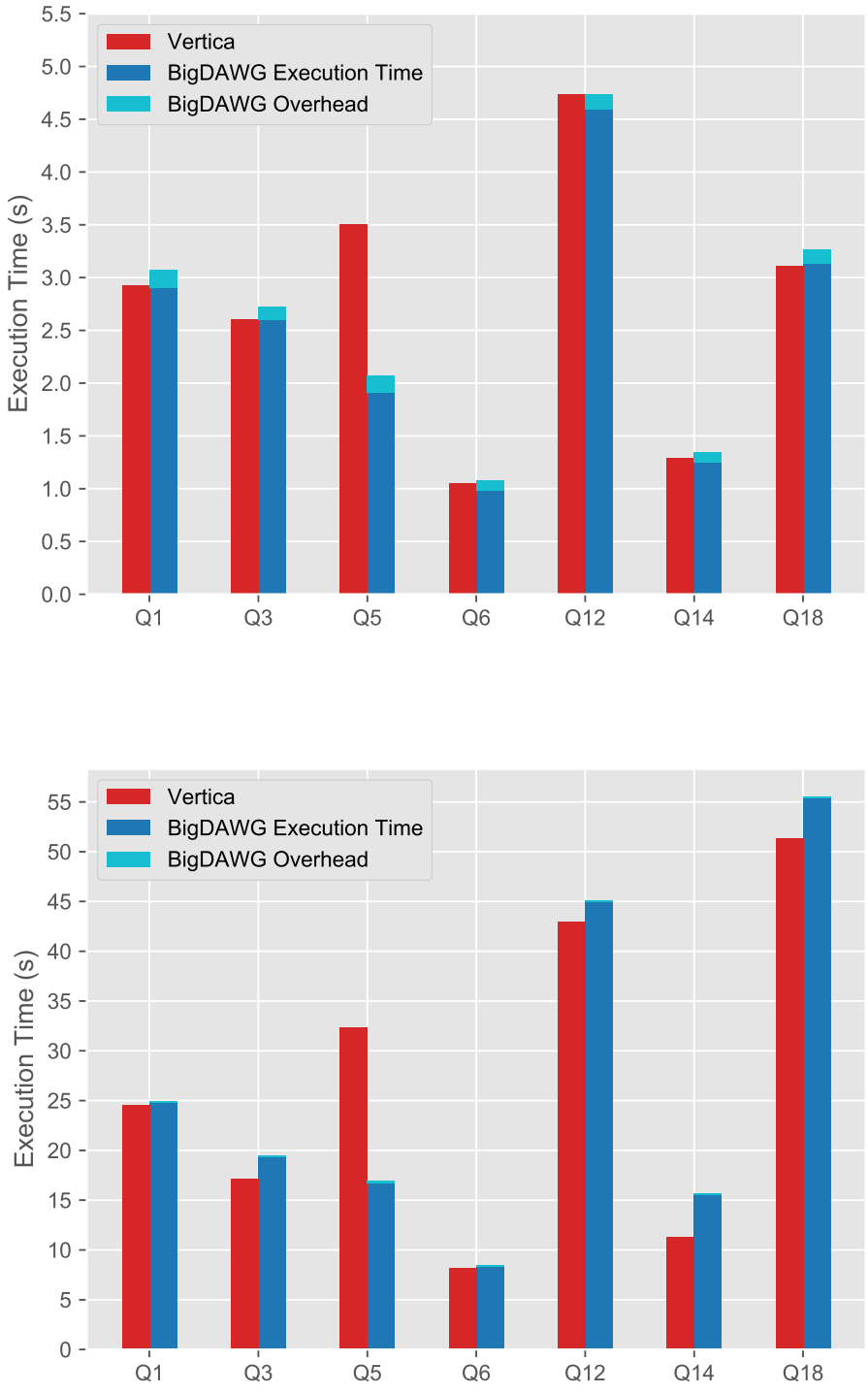

Figure 4-2: TPC-H query performance for a single Vertica instance for 10GB and 100GB of data, respectively, with and without BigDAWG.

code so that I could get a breakdown of how much time each stage of the total BigDAWG execution time takes.

In this section, I illustrate the relative performance of three TPC-H queries on the 100GB dataset in three separate configurations: on a single Postgres instance, on a single Vertica instance, and a cross-engine configuration where relevant tables are split between Postgres and Vertica, with the best-performing table assignment used. I also compute the time it would take with the same split-table configuration, except instead of using BigDAWG, I copy the entirety of the relevant tables to one of the engines, and then compute the query on the target engine, emulating how a cross-engine query would work without the use of BigDAWG.

Some results have been omitted from the following figures to preserve their scaling, but the complete set of data can be found in Appendix B.

4.4.1 TPC-H Query 14

TPC-H Query 14, which can be seen in A.6, computes the response to a promotion like a tv advertisement campaign. This query did not have to be modified to be used successfully on BigDAWG.

The primary execution stages for the query plan developed by the Planner were roughly as follows:

1. Materialize the results of running a subquery on the lineitem table with the lineitem-specific filters on Postgres
2. Materialize the results of running the subquery on the part table on Vertica
3. Migrate the resulting lineitem tuples (about 7.5 million) from Postgres to Vertica
4. Perform a query joining the two intermediate results to produce the final result.

Figure 4-3 shows that the cross-engine configuration performed worse than both Vertica but better than Postgres. It also shows that BigDAWG performs better than the naive solution of migrating the entire part table to Postgres and executing the

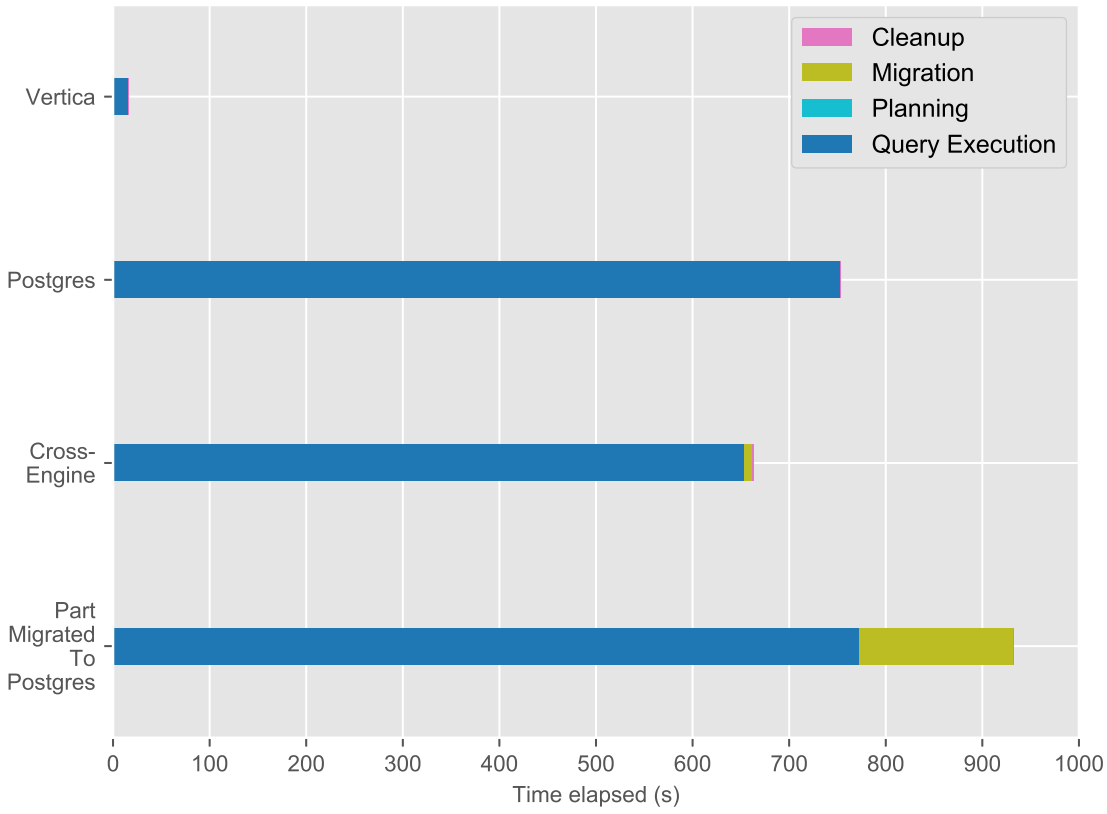

Figure 4-3: The breakdown of the BigDAWG execution time for TPC-H query 14 with the lineitem table on Postgres and the part table on Vertica.

original query there by limiting the amount of data that needs to be transferred. BigDAWG also significantly outperforms a migration of the entire lineitem table to Vertica before execution of the query, as shown in Table B.12.

Although the query plan used seems reasonable, step 2 is unnecessary because the executor essentially duplicates the entire part table into another temporary table. This behavior would make sense in case the part table needs to be filtered and migrated to another node, but in this case, it is already at its final destination. In this case, the overall runtime was probably not affected because this subquery was run concurrently with the Postgres query in step 1, which took a longer time to complete. However, this may not always be the case, so modifying the query executor to identify these unnecessary nodes would probably improve performance and reduce computation overhead.

4.4.2 TPC-H Query 12

TPC-H Query 12, which can be seen in A.5, determines whether the choice of shipping mode affects whether more parts in critical-priority orders are received by customers after the commited date. The original query included `and l_shipmode in (SHIPMODE1, SHIPMODE2)` as a clause in the filter, but BigDAWG does not yet support the `in` keyword, so the modified query computes its effect for all shipping modes.

The primary execution stages for the query plan developed by the Planner were roughly as follows:

1. Materialize the results of running the subquery on the lineitem table with lineitem-specific filters on Postgres
2. Materialize the results of running the subquery on the orders table on Vertica
3. Migrate the resulting lineitem tuples (about 10 million) from Postgres to Vertica
4. Perform a query joining the two intermediate results to produce the final result.

Figure 4-4 and Table B.11 show that the cross-engine configuration performed worse than Vertica but better than Postgres. Since both the lineitem and orders tables

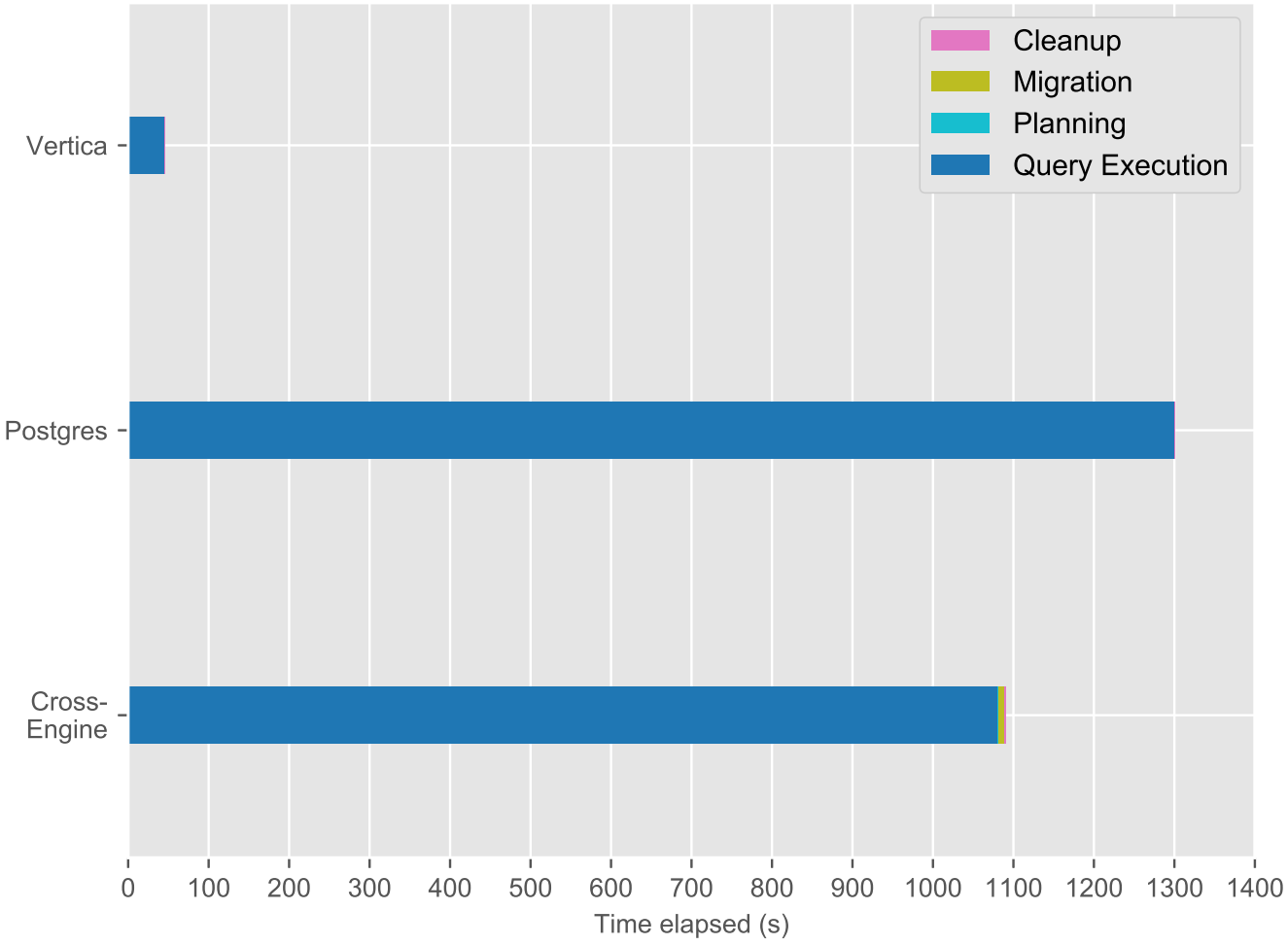

Figure 4-4: The breakdown of the BigDAWG execution time for TPC-H query 12 with the lineitem table on Postgres and the orders table on Vertica.

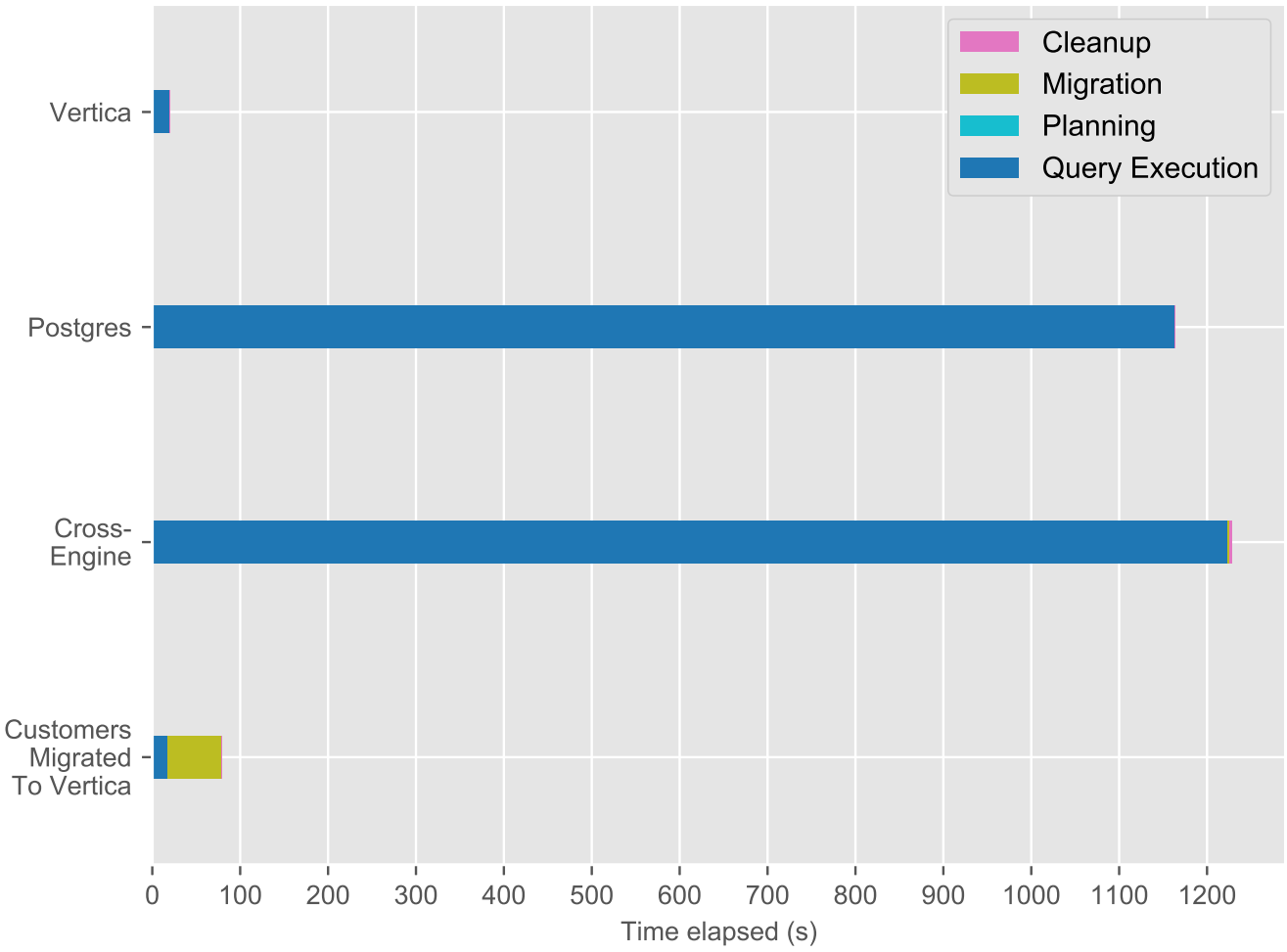

Figure 4-5: The breakdown of the BigDAWG execution time for TPC-H Query 3 with the customers table on Postgres and the lineitem and orders tables on Vertica.

are the largest tables in the dataset, BigDAWG performs especially well compared to the naive solution of migrating all tables to one engine before executing the query. The execution plan used is very similar to that of query 14, and shares the same issue of an unnecessary materialization of the entire orders table on Vertica. Similarly, the longer runtime of the concurrently executed Postgres materialization masks the negative effects, if any.

4.4.3 TPC-H Query 3

TPC-H Query 3, which can be seen in A.2, is meant to return the 10 unshipped orders with the highest value. However, BigDAWG does not yet support ordering by aliases, so I removed `ORDER BY revenue` from the original generated query.

The primary execution stages for the query plan developed by the Planner were roughly as follows:

1. Materialize the results of running the subquery on the orders table with the customer-specific filter on Postgres
2. Materialize the results of running the subquery on the customer table with the order-specific filter on Vertica
3. Materialize the results of running the subquery on the lineitem table with the lineitem-specific filters on Vertica
4. Migrate the resulting customer tuples (about 3 million) from Postgres to Vertica
5. Materialize the results of joining the intermediate customers and orders tables on Vertica.
6. Materialize the results of joining the intermediate lineitem and orders tables on Vertica.
7. Compute the final result by joining the tuples from two intermediate join nodes on Vertica.

As shown in Figure 4-5, the cross-engine setup did not perform well relative to Postgres or Vertica. Because this query uses three tables rather than two, the query plan and execution plan generated by BigDAWG includes more steps than the ones used for Queries 12 and 14, with much more time spent on executing intermediate queries.

While this query plan makes sense given the tree of dependencies, like those of Queries 12 and 14, there are unnecessary materializations of the orders and lineitem tables, the largest tables in the TPC-H dataset by far, as shown in Table B.2, taking nearly 1000 seconds to perform.

Furthermore, the intermediate joins in steps 5 and 6 probably do not need to be explicitly materialized, as all the data needed is already colocated on Vertica by that point. Deferring the execution of the three-table computation to the Vertica query optimizer would probably also improve performance. In fact, the naive solution of migrating the entire customers table, which is comparatively small, to Vertica and then executing the entire query directly on Vertica outperforms BigDAWG by a factor of about 15, as shown in Table B.10.

4.4.4 Discussion

These results show that for some queries, executing a query through BigDAWG with two engines can outperform a single engine. The BigDAWG system itself does not add too much overhead, with query execution on the databases themselves taking the majority of the execution time. The overhead incurred by the planning stage remains constant even as the size of the dataset increases. It is unsurprising that Vertica performs better on the TPC-H query set, which is primarily comprised of analytical queries, which Vertica is optimized for. The primary gains in performance over Postgres are likely from performing the final analytical computations, such as aggregates using `SUM`, on Vertica, which significantly outperforms Postgres for these types of queries, as seen in Section 4.3.

These results also show that by identifying and migrating only the tuples that will be needed for the query, BigDAWG outperforms the naive solution of migrating entire tables to one destination engine and executing the query there. Large tables, such as the lineitem table in the TPC-H dataset, can be costly to migrate. For every incoming query, the Planner determines what objects are necessary to perform the query, and then generates a query plan based on those dependencies. If the query involves filtering, it can limit the tuples migrated to only those that satisfy the filter condition, reducing the amount of data that must be transmitted between engines. BigDAWG does this dependency checking for its users so that they do not have to know about the distribution of data among the underlying engines or the optimal set of tuples to move, which is especially convenient in workloads containing ad-hoc

queries, in which the data that should be moved may change from query to query. Thus, queries that involve selective filters should perform well on BigDAWG.

Despite these results, the cross-engine BigDAWG configuration does not perform as well as it could be. One major factor is that the nodes of the query plan tree are generated based on an initial single-engine parse tree from a Postgres instance. Since Postgres is a row store, it does not incur much extra cost by fetching full rows even if some columns are not needed. In my experimentation, the Postgres query optimizer typically performed projections at the top level of the query tree, only waiting until the end of the execution to reduce the number of columns.

Conversely, since Vertica is a column store, it is much more expensive to fetch extraneous columns, which can have large performance implications, especially as seen in the execution of TPC-H query 3 in 4.4.3. Furthermore, retaining unused columns in intermediate results caused extra data to be transmitted during the migration step, also increasing runtime as well as compute and bandwidth usage. To address this problem, the Planner should be modified to identify which columns are unnecessary to compute the final query result, and remove those columns from intermediate nodes in the query plans it produces.

Another problem is the unnecessary materialization of intermediate nodes during the execution of query. This behavior makes sense if the intermediate result is to be migrated to another database engine later, but the query executor seems to do so indiscriminately, even if the table being materialized is already present on its destination engine. This results in extraneous computation and disk space usage, as well as an increase in execution time. This problem does not arise in single-engine configurations, where the entire query is propagated to the engine itself to be executed. The Planner and Executor should identify nodes that involve computing intermediates that are already on their terminal destination engine, and instead combine them with the final query that is executed directly on the underlying database engine. Further improvements are discussed in more detail in Chapter 5.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Future work

In this chapter, I discuss a few areas where the BigDAWG system could be improved based on my research.

5.1 Streamlining the process for adding new engines

The process for adding support for new engines to BigDAWG could be streamlined and better defined. This is important as we open usage of the BigDAWG system to new users, who may want to use currently unsupported databases. Currently, there are a large number of disparate classes and components that need to be implemented and modified in order for BigDAWG to successfully interact with a new database engine. Refactoring the system by defining more specific interfaces that explicitly makes clear what functionality should be implemented will make the process easier moving forward.

5.2 Generalizing the migration process

Currently, the migration between a pair of database engines requires the implementation of six classes: two Migrators, two Export classes, and two Load classes. As the number of engines that BigDAWG supports increases, the number of these classes scales quadratically under the current design. Because the primary format for im-

porting and exporting data is CSV, it should be possible to refactor the Migrator components to generalize to most databases, rather than writing new unique classes for each one.

5.3 Smarter query planning and execution

When the Planner constructs candidate plans for a particular query on the relational island, it currently parses the parse tree returned by issuing `EXPLAIN` statement for the client query on a single Postgres instance containing all the table schemas to determine the dependencies in the query plan. Due to the initial dependence on the Postgres query optimizer, the query trees that the Planner constructs may be far from optimal for databases with divergent architectures from Postgres. For example, while Postgres does not does not lose much efficiency by fetching full rows rather than the projections needed to compute a query, column stores such as Vertica incur a large performance penalty. Furthermore, preserving these unused columns may cause more bandwidth and computation power to be used in a migration, because extraneous data is transmitted. Identifying unused columns and removing them from intermediate queries during the planning stage should improve performance significantly.

Another problem I identified is that in the course of executing cross-engine queries, intermediate results tend to get unnecessarily materialized, as discussed in 4.4. Modifying the Executor to identify nodes that are already present on their destination engines can improve runtime and reduce compute and disk usage. The Planner and Executor should also aim to defer query optimization to the underlying destination engines if possible, because they may have vastly differing architectures from Postgres, and thus optimization strategies used by Postgres may not perform as well as native query optimization. For example, the Planner could condense colocated nodes in a query plan into a single node to be executed as one single query and take advantage of the target engine's query optimization.

5.4 Considering a larger set of query plans

The set of query plans that the Planner considers is currently limited based on where tables are located. For example, if a table is located on a Postgres instance, the Planner will only consider a query plan that executes solely on that database engine. However, Vertica vastly outperforms Postgres in analytical queries using aggregates such as `SUM` or `COUNT`, so such a query may be executed faster by simply migrating relevant tuples to Vertica or another analytics-based database and computing the final result there, even if none of the data was originally present on that engine.

Another possibility would be to supplement the BigDAWG API to allow the user to have more control over the query execution plan; an example would be letting the user specify that analytical queries on data stored in Postgres should be performed in Vertica. While there exists functionality to tell BigDAWG to move data across engines using a bdcast, this is only available for inter-island queries.

In this way, we can further leverage the relative strengths of the database engines available through BigDAWG.

5.5 Migration using the binary data transformer for Postgres to Vertica

The current implementation only supports CSV-based migration for Postgres and Vertica. However, CSV data loading can be a highly CPU intensive process due to parsing and deserialization, and it is likely that we can achieve a significant speedup by using a binary data format for importing and exporting data[8]. BigDAWG currently uses a module written in C++ to perform transformation of binary data formats for migrations between Postgres and SciDB.

Using this module to perform conversions from the binary data formats of Postgres to that of Vertica will allow us to take advantage of Vertica's binary data loading capabilities. Unfortunately, Vertica does not support binary export, so we will only

be able to improve execution times for Postgres-to-Vertica migrations. Even though the bulk of the execution time currently comes from executing queries directly on the underlying database engines, implementing this change should result in a decent performance improvement.

5.6 Further evaluation for inter-island queries

My thesis focused primarily on evaluating the BigDAWG system for intra-island cross-engine queries using the TPC-H benchmark on the relational island. Inter-island queries require explicit migration instructions and are not as easily constructed, but a systematic evaluation may demonstrate the power of the BigDAWG system even better.

Chapter 6

Conclusion

In this thesis, I have described the process for adding support for new database engines to the relational island in the BigDAWG system. I also presented results from a performance evaluation of the system using the TPC-H benchmark.

The BigDAWG system performs well in a single-engine scenario, even outperforming native queries to Postgres and Vertica in certain cases. There is minimal planning overhead, which remains constant as the size of the dataset increases. For cross-engine queries, BigDAWG's performance seems correlated with the complexity of the query plan, but seems to achieve reasonable results for queries involving two tables located on separate engines, with multiple queries showing improved performance with a configuration using both Postgres and Vertica when compared to a single Postgres instance alone. Compared to the naive solution for the two-engine scenario of migrating all tables to one of the engines and computing the final result there, BigDAWG also shows a marked improvement by identifying and migrating only the data that is necessary to compute the final result.

The experiments detailed in this report use query runtime as a primary measure of performance; while the results do not show the multi-engine configuration outperforming Vertica alone, it may still be advantageous to use BigDAWG for reasons outside the scope of this thesis. BigDAWG does not yet support insertions, but one area where Postgres could outperform Vertica is with transactional workloads, characterized by a large number of small transactions (usually updates or insertions) in a

short period of time. In scenario with both transactional and analytic workloads on the same dataset, we can use the relative strengths of each database engine to achieve better performance overall than with any single engine.

Overall, the BigDAWG system has shown promising results in leveraging the strengths of its component database engines. From my research, I have identified certain weaknesses in the current design. Addressing these weaknesses should further improve usability and performance.

Appendix A

TPCH Queries Used

A.1 Query 1

```
select
  l_returnflag,
  l_linestatus,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
  sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
  avg(l_quantity) as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order
from
  lineitem
where
  l_shipdate <= date '1998-12-01' - interval '65' day
group by
  l_returnflag,
  l_linestatus
order by
  l_returnflag,
  l_linestatus
LIMIT 1;
```

A.2 Query 3 (ORDER BY removed)

```
select
  l_orderkey,
  sum(l_extendedprice * (1 - l_discount)) as revenue,
  o_orderdate,
  o_shippriority
from
  customer,
  orders,
  lineitem
where
  c_mktsegment = 'MACHINERY'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < date '1995-03-01'
  and l_shipdate > date '1995-03-01'
group by
  l_orderkey,
  o_orderdate,
  o_shippriority
LIMIT 10;
```

A.3 Query 5 (ORDER BY removed)

```
select
  n_name,
  sum(l_extendedprice * (1 - l_discount)) as revenue
from
  customer,
  orders,
  lineitem,
  supplier,
  nation,
  region
where
  c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and l_suppkey = s_suppkey
  and c_nationkey = s_nationkey
  and s_nationkey = n_nationkey
  and n_regionkey = r_regionkey
  and r_name = 'MIDDLE␣EAST'
  and o_orderdate >= date '1993-01-01'
  and o_orderdate < date '1993-01-01' + interval '1' year
group by
```

```
  n_name
LIMIT 1;
```

A.4 Query 6

```
select
  sum(l_extendedprice * l_discount) as revenue
from
  lineitem
where
  l_shipdate >= date '1993-01-01'
  and l_shipdate < date '1993-01-01' + interval '1' year
  and l_discount between 0.03 - 0.01 and 0.03 + 0.01
  and l_quantity < 25
LIMIT 1;
```

A.5 Query 12 (IN filter removed)

```
select
  l_shipmode,
  sum(case
    when o_orderpriority = '1-URGENT'
      or o_orderpriority = '2-HIGH'
      then 1
    else 0
  end) as high_line_count,
  sum(case
    when o_orderpriority <> '1-URGENT'
      and o_orderpriority <> '2-HIGH'
      then 1
    else 0
  end) as low_line_count
from
  orders,
  lineitem
where
  o_orderkey = l_orderkey
  and l_commitdate < l_receiptdate
  and l_shipdate < l_commitdate
  and l_receiptdate >= date '1994-01-01'
  and l_receiptdate < date '1994-01-01' + interval '1' year
group by
  l_shipmode
```

```
order by
  l_shipmode
LIMIT 1;
```

A.6 Query 14

```
select
  100.00 * sum(case
    when p_type like 'PROMO%'
      then l_extendedprice * (1 - l_discount)
    else 0
  end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue
from
  lineitem,
  part
where
  l_partkey = p_partkey
  and l_shipdate >= date '1994-11-01'
  and l_shipdate < date '1994-11-01' + interval '1' month
LIMIT 1;
```

A.7 Query 18 (ORDER BY removed)

```
select
  c_name,
  c_custkey,
  o_orderkey,
  o_orderdate,
  o_totalprice,
  sum(l_quantity)
from
  customer,
  orders,
  lineitem
where
  o_orderkey in (
    select
      l_orderkey
    from
      lineitem
    group by
      l_orderkey having
        sum(l_quantity) > 315
```

```
  )
  and c_custkey = o_custkey
  and o_orderkey = l_orderkey
group by
  c_name,
  c_custkey,
  o_orderkey,
  o_orderdate,
  o_totalprice
order by
  o_totalprice desc,
  o_orderdate
LIMIT 100;
```

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix B

TPCH Data

Table	Number of Tuples	Size (bytes)
lineitem	59986052	7.2G
orders	15000000	1.7G
partsupp	8000000	1.2G
part	2000000	231M
customer	1500000	233M
supplier	100000	14M
nation	25	2.2K
region	5	384

Table B.1: Size of tables for generated TPC-H dataset for 10GB of data.

Table	Number of Tuples	Size (bytes)
lineitem	600037902	74G
orders	150000000	17G
partsupp	80000000	12G
part	20000000	2.3G
customer	15000000	2.3G
supplier	1000000	136M
nation	25	2.2K
region	5	384

Table B.2: Size of tables for generated TPC-H dataset for 100GB of data.

Configuration	Q1	Q3	Q5	Q6	Q12	Q14	Q18
Native	189.8771	45.5855	18.4707	17.9423	34.6521	10.7365	87.9068
BigDAWG Total	197.9404	45.1839	18.4158	17.686	35.0654	10.5532	86.2032
BigDAWG Execution	197.6742	45.0731	18.2619	17.5991	34.8483	10.4782	86.06

Table B.3: Runtime (in s) of running TPCH queries with 10GB data on Postgres.

Configuration	Q1	Q3	Q5	Q6	Q12	Q14	Q18
Native	2.9266	2.6009	3.5059	1.0500	4.3671	1.2874	3.1099
BigDAWG Total	3.0771	2.7262	2.0775	1.0798	4.513	1.3458	3.2638
BigDAWG Execution	2.9063	2.6056	1.9129	0.9864	4.3693	1.2497	3.1338

Table B.4: Runtime (in s) of running TPCH queries with 10GB data on Vertica

Configuration	Q1	Q3	Q5	Q6	Q12	Q14	Q18
Native	2141.917	1142.892	1458.791	614.992	1329.813	772.902	3648.222
BigDAWG Total	2132.725	1163.713	1412.64	602.095	1300.951	753.526	3687.175
BigDAWG Execution	2132.378	1163.493	1412.378	601.913	1300.747	753.321	3686.961

Table B.5: Runtime (in s) of running TPCH queries with 100GB data on Postgres.

Configuration	Q1	Q3	Q5	Q6	Q12	Q14	Q18
Native	24.908	19.446	16.900	8.449	45.083	15.630	55.535
BigDAWG Total	24.523	17.147	32.341	8.139	42.936	11.309	51.307
BigDAWG Execution	24.783	19.326	16.685	8.326	44.943	15.516	55.371

Table B.6: Runtime (in s) of running TPCH queries with 100GB data on Vertica

Configuration	Planning	Execution	Migration	Cleanup	Total	Tuples Migrated
BigDAWG	0.24156	72.790	0.2634	0.577	74.7042	~300,000
Migrate to Postgres	-	45.5855	840.098	-	885.683	74,986,052
Migrate to Vertica	-	2.6009	5.2693	-	7.8702	1,500,000

Table B.7: Query execution breakdown (in s) for TPCH query 3 with 10GB data with the lineitem and orders tables on Vertica and the customer table on Postgres.

Configuration	Planning	Execution	Migration	Cleanup	Total	Tuples Migrated
BigDAWG	0.1976	28.2748	0.8573	0.2384	29.1535	~1,000,000
Migrate to Postgres	-	34.6521	113.7116	-	148.3637	15,000,000
Migrate to Vertica	-	4.3671	399.7762	-	404.1433	59,986,052

Table B.8: Query execution breakdown (in s) for TPCH query 12 with 10GB data with the orders table on Vertica and the lineitem table on Postgres.

Configuration	Planning	Execution	Migration	Cleanup	Total	Tuples Migrated
BigDAWG	0.1143	10.2639	0.9634	0.0677	11.4095	~750,000
Migrate to Postgres	-	10.7365	17.8245	-	28.561	2,000,000
Migrate to Vertica	-	1.2874	399.7762	-	401.0636	59,986,052

Table B.9: Query execution breakdown (in s) for TPCH query 14 with 10GB data with the part table on Vertica and the lineitem table on Postgres.

Configuration	Planning	Execution	Migration	Cleanup	Total	Tuples Migrated
BigDAWG	0.2327	1223.28	1.429	2.898	1227.81	~3,000,000
Migrate to Postgres	-	1264.45	8752.58	-	10017.033	750,037,902
Migrate to Vertica	-	17.147	61.49	-	78.637	15,000,000

Table B.10: Query execution breakdown (in s) for TPCH query 3 with 100GB data with the lineitem and orders tables on Vertica and the customer table on Postgres.

Configuration	Planning	Execution	Migration	Cleanup	Total	Tuples Migrated
BigDAWG	0.1505	1080.9147	7.9227	1.3705	1090.4035	~10,000,000
Migrate to Postgres	-	1329.81	1264.448	-	2594.258	150,000,000
Migrate to Vertica	-	42.936	4454.94	-	4497.876	600,037,902

Table B.11: Query execution breakdown (in s) for TPCH query 12 with 100GB data with the orders table on Vertica and the lineitem table on Postgres.

Configuration	Planning	Execution	Migration	Cleanup	Total	Tuples Migrated
BigDAWG	0.1426	653.7294	7.8498	1.7418	663.46	~7,500,000
Migrate to Postgres	-	772.90	159.16	-	932.06	20,000,000
Migrate to Vertica	-	11.309	4454.94	-	4466.25	600,037,902

Table B.12: Query execution breakdown (in s) for TPCH query 14 with 100GB data with the part table on Vertica and the lineitem table on Postgres.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Accumulo. https://accumulo.apache.org/.

[2] Docker. https://www.docker.com/.

[3] Mysql. https://www.mysql.com/.

[4] The tpc-h benchmark. http://www.tpc.org/tpch/.

[5] Peinan Chen, Vijay Gadepally, and Michael Stonebraker. The bigdawg monitoring framework. In *High Performance Extreme Computing Conference (HPEC)*, 2016.

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[7] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. The bigdawg polystore system. *SIGMOD Rec.*, 44(2):11–16, August 2015.

[8] Adam Dziedzic, Aaron J Elmore, and Michael Stonebraker. Data transformation and migration in polystores. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–6. IEEE, 2016.

[9] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, S. Madden, D. Maier, T. Mattson, S. Papadopoulos, J. Parkhurst, N. Tatbul, M. Vartak, and S. Zdonik. A demonstration of the bigdawg polystore system. *Proc. VLDB Endow.*, 8(12):1908–1911, August 2015.

[10] Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Tim Mattson, and Michael Stonebraker. The bigdawg polystore system and architecture. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–6. IEEE, 2016.

[11] A Gupta, Vijay Gadepally, and Michael Stonebraker. Cross-engine query execution in federated database systems. In *High Performance Extreme Computing Conference (HPEC)*, 2016.

[12] Carolyn J. Hursch and Jack L. Hursch. *SQL: The Structured Query Language*. TAB Books, Blue Ridge Summit, PA, USA, 1988.

[13] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *Proc. VLDB Endow.*, 5(12):1790–1801, August 2012.

[14] Oracle. Java jdbc api. https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/.

[15] Mohammed Saeed, Mauricio Villarroel, Andrew T Reisner, Gari Clifford, Li-Wei Lehman, George Moody, Thomas Heldt, Tin H Kyaw, Benjamin Moody, and Roger G Mark. Multiparameter intelligent monitoring in intensive care ii (mimic-ii): a public-access intensive care unit database. *Critical care medicine*, 39(5):952, 2011.

[16] Zuohao She, Surabhi Ravishankar, and Jennie Duggan. Bigdawg polystore query optimization through semantic equivalences. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–6. IEEE, 2016.

[17] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The architecture of scidb. In *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management*, SSDBM'11, pages 1–16, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] Michael Stonebraker and Ugur Cetintemel. One size fits all": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[19] Michael Stonebraker and Greg Kemnitz. The postgres next generation database management system. *Commun. ACM*, 34(10):78–92, October 1991.

[20] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.