Database Engine Integration and Performance Analysis of the BigDAWG Polystore System

Katherine Yu*, Vijay Gadepally*†, Michael Stonebraker*

MIT CSAIL*, MIT Lincoln Laboratory†
kateyu@mit.edu, vijayg@ll.mit.edu, stonebraker@csail.mit.edu

***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). We present 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.**

I. 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 [1]. 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 [2], 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 may 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 [3], [4] 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[5] for time-series data, Apache Accumulo [6] for freeform text notes, and Postgres [7] 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. An example of an application of BigDAWG to a scientific analysis problem can be found in [8].

The current BigDAWG relational island only supports Postgres as a relational database. In this article, we describe our approach to integrating two relational databases, MySQL and Vertica, to the relational island. We validate the hypothesis that leveraging the relative strengths of different database engines for tasks that they are well suited to can improve overall query performance via the popular TPC-H [9] decision support benchmark.

The remainder of this article is organized as follows. In Section II, we describe the BigDAWG architecture and modules. In Section III, we detail the process for integrating new database engines into the BigDAWG system. In Section IV, we discuss performance experiments with their associated results and analysis. In Section V, we discuss areas of future work and conclude.

II. BigDAWG System Overview

The BigDAWG system [4] 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 multiple 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.

The islands of information serve as an abstraction to the client, each with its own query language, data model, and connectors, or *shims*, to each underlying database engine [10]. In this way, clients do not need to know anything about

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

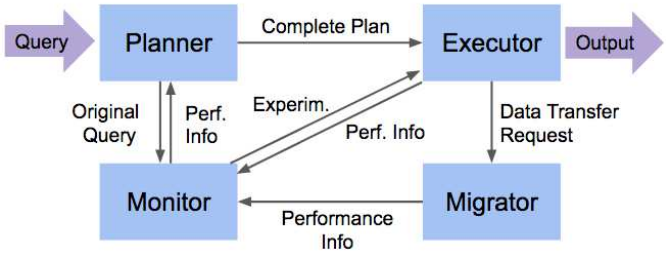

Fig. 1. Data flow across BigDAWG middleware architecture.

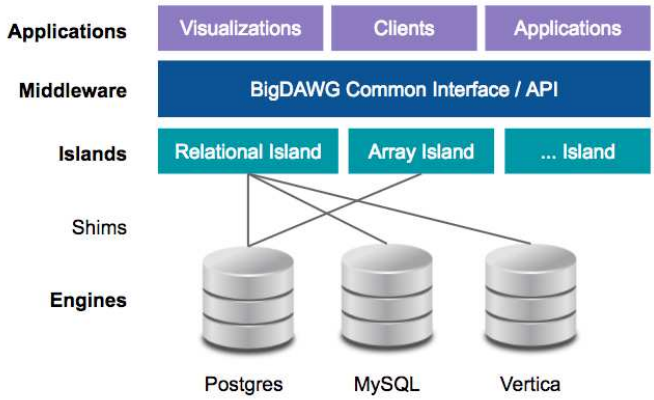

Fig. 2. A visualization of the BigDAWG architecture, including MySQL and Vertica. For simplicity, we have left out Accumulo and SciDB which are also currently supported by BigDAWG.

the underlying engines, and just need to use the common BigDAWG syntax to issue queries. The client can also instruct BigDAWG to migrate or cast data from one island to another. In this article, we focus on our work with the relational island which implements a subset of the Structured Query Language (SQL) [11] specification.

The BigDAWG middleware receives client requests and interacts with the underlying database engines. 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 1.

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 [12] 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.

The Executor [13] 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.

The Migrator [14] moves data between database engines in either CSV or binary format depending on performance constraints. The Migrator also reports performance metrics from each migration to the Monitor [12] in order to use these metrics to better plan future queries.

III. Integrating new database engines into BigDAWG

In this section, we describe the process for integrating new database support into the BigDAWG system. Specifically, we concentrate on the addition of two popular relational databases: MySQL and the columnar database Vertica.

MySQL [15] 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 [16] is the commerical version of the C-Store research prototype [17]. 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.

A. Implementation details

The BigDAWG middleware component is primarily written in Java, so this section will be described in the context of the language.

1) Connection to the database: When adding a new engine, one must implement a number of 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 [18] 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.

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

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

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) 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, we 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, we 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.

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.

IV. Performance analysis using the TPC-H benchmark

In this section, we discuss and analyze benchmark results and their implications going forward with BigDAWG.

A. The TPC-H benchmark

TPC-H[9] is an industry standard decision support benchmark developed by the Transaction Processing Performance Council (TPC) that models an industry dataset involved with managing and distributing products worldwide. We ultimately decided to use queries from the TPC-H benchmark for evaluation purposes because it seemed sufficiently generalizable to the BigDAWG system and provides a lot of built-in tools to aid users in generating queries and datasets. We 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 some of the remaining queries, as BigDAWG does not fully support certain syntax such as `ORDER BY`s using aliases.

B. Experimental Setup

The following performance tests were performed on a machine running Ubuntu 14.04, with 64GB of RAM and 24 2GHz virtual processor cores. We conducted ech of the experiments using an installation of the BigDAWG middleware consisting of virtualized instances of the relevant database engines using Docker[19]. A detailed description of the middleware setup can be found in [20]. Further, we note that we did not leverage extensive database tuning for the purpose of our experiments.

TABLE I
SIZE OF TABLES FOR GENERATED 100GB TPC-H DATASET.

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

We generated data using the `tpchgen` tool, with a scaling factors of 10 and 100, resulting in 10GB and 100GB of data, respectively. The relative size of each of the tables for the 100GB dataset is shown in Table I.

The full dataset was loaded into each of the database engines, with the catalog modified accordingly to register these objects with the BigDAWG middleware. Since Postgres and MySQL are similar in architecture, we describe the results of queries between Postgres and Vertica intra-relational island queries.

C. Single-engine queries through BigDAWG

In this section, we compare the performance of queries from the TPC-H benchmark directly to a database engine against the performance of querying via the BigDAWG system. These results indicate that the overhead from querying through BigDAWG is minimal, and may even improve performance in certain cases.

Figures 3 and 4 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 3. For Vertica, the overhead added an extra 5 to 10 percent in overall runtime on the 10GB dataset, because the query runtimes were much shorter when performed on Vertica. The overhead incurred by querying through BigDAWG remains constant with respect to the database engine used, as the overhead shrinks proportionally when the dataset was increased to 100GB in Figure 4. These results indicate that BigDAWG can be suitable for interactive analysis such as visualization applications [21].

We can also see that some queries were actually executed faster when run through BigDAWG. Since 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, one of the filter clauses used in Query 5, which shows the most dramatic improvement under BigDAWG for Vertica, was

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

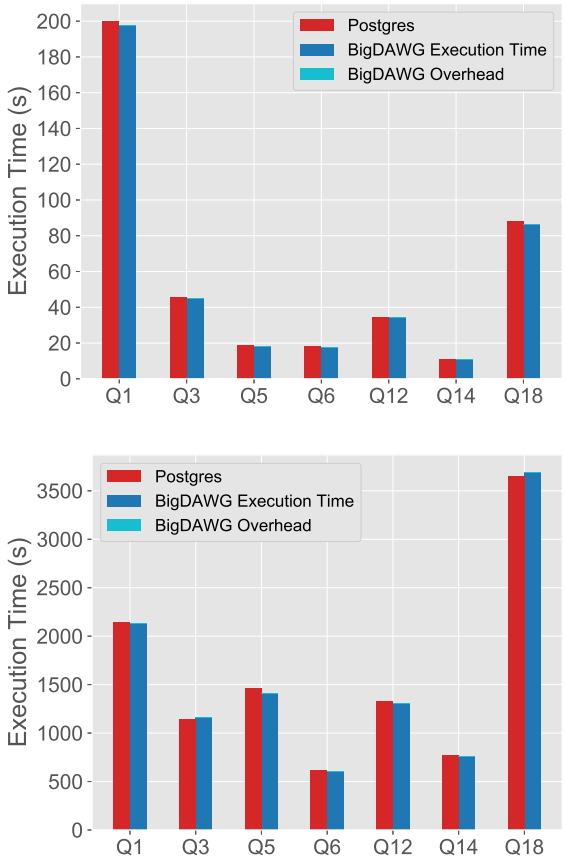

Fig. 3. TPC-H query performance for a single Postgres instance for 10GB and 100GB of data, respectively, with and without BigDAWG.

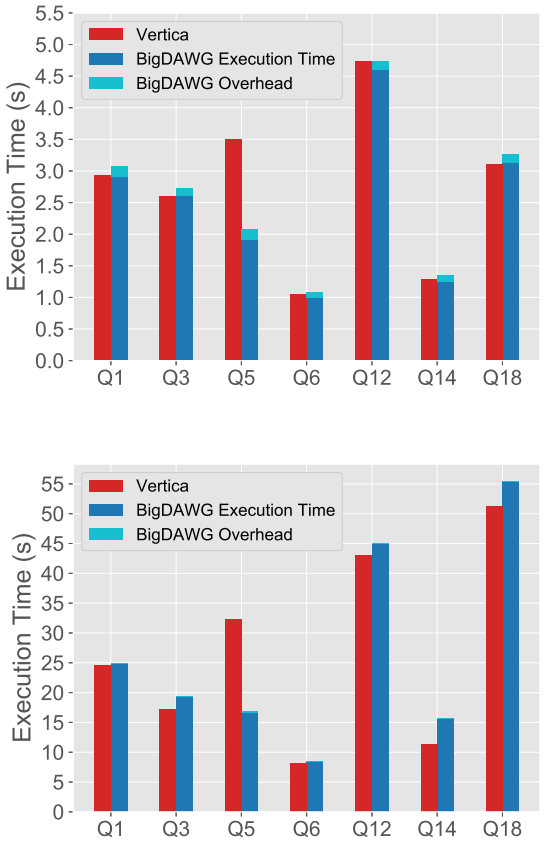

Fig. 4. TPC-H query performance for a single Vertica instance for 10GB and 100GB of data, respectively, with and without BigDAWG.

`o_orderdate < date '1993-01-01' + interval '1' year`. During the parsing process, BigDAWG computed the resulting date and rewrote the clause as `o_orderdate < date '1994-01-01'` before executing the query on Vertica directly, which resulted in a shorter execution time overall.

D. 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. To measure this performance, we instrumented the middleware to get a breakdown of the time taken by each of the middleware stages.

To illustrate these results, we present the relative performance of three TPC-H queries on the 100GB dataset in three separate configurations: a single Postgres instance, a single Vertica instance, and a cross-engine configuration where TPC-H tables are split between Postgres and Vertica, with the best-performing table assignment used. We also compute the time taken with the same split-table configuration, where the migration portion is done manually without BigDAWG. The goal of this last experiment is to emulate how queries perform without a broker such as BigDAWG. Below, we describe

```
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;
```

Fig. 5. The generated TPC-H Query 14 used in our experiments.

individual TPC-H queries along with a discussion of the performance results.

1) TPC-H Query 14: This query, shown in Figure 5, computes the response to a promotion like a television advertisement campaign. This query did not have to be modified to be used successfully on BigDAWG. For our experiments, we used a configuration that placed the `lineitem` table on Postgres and the `part` table on Vertica.

The query plan executed by BigDAWG roughly involves splitting the input query into `lineitem` and `part` queries,

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

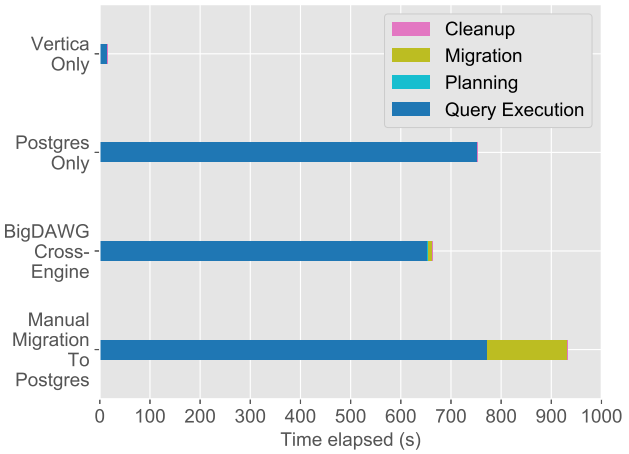

Fig. 6. The breakdown of the BigDAWG execution time for TPC-H query 14 with the lineitem table on Postgres and the part table on Vertica.

TABLE II
TPC-H QUERY 14 EXECUTION BREAKDOWN (IN S) WITH THE PART TABLE ON VERTICA AND THE LINEITEM TABLE ON POSTGRES.

Configuration	Execution	Migration	Other	Total	Tuples Migrated
Postgres Only	772.90	-	-	772.90	-
Vertica Only	11.31	-	-	11.31	-
Migrate to PG	772.90	159.16	-	932.06	20M
Migrate to V	11.31	4454.94	-	4466.25	600M
BigDAWG	653.73	7.850	1.884	663.46	7.5M

and materializing the intermediate results on their respective engines. Then, the resulting `lineitem` tuples (about 7.5 million) are migrated from Postgres to Vertica. Finally, a query joining the two intermediate tables is executed that produces the final result.

Figure 6 indicates that for this particular query, the cross-engine configuration performed worse than using only Vertica but better than using only Postgres. It also shows that BigDAWG performs better than the naïve solution of migrating the entire part table to Postgres and executing the original query there by limiting the amount of data that needs to be transferred. BigDAWG also vastly outperforms a migration of the entire `lineitem` table to Vertica before execution of the query, as shown in Table II.

2) TPC-H Query 12: This query, shown in Figure 7, 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

TABLE III
TPC-H QUERY EXECUTION BREAKDOWN (IN S) WITH THE ORDERS TABLE ON VERTICA AND THE LINEITEM TABLE ON POSTGRES.

Configuration	Execution	Migration	Other	Total	Tuples Migrated
Postgres Only	1329.81	-	-	1329.81	-
Vertica Only	42.936	-	-	42.936	-
Migrate to PG	1329.81	1264.44	-	2594.25	150M
Migrate to V	42.936	4454.94	-	4497.87	600M
BigDAWG	1080.91	7.922	1.521	1090.40	10M

```
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;
```

Fig. 7. The generated and modified TPC-H Query 12 used in our experiments.

TABLE IV
TPC-H QUERY 3 EXECUTION BREAKDOWN (IN S) WITH THE LINEITEM AND ORDERS TABLES ON VERTICA AND THE CUSTOMER TABLE ON POSTGRES.

Configuration	Execution	Migration	Other	Total	Tuples Migrated
Postgres Only	1264.45	-	-	1264.45	-
Vertica Only	17.14	-	-	17.14	-
Migrate to PG	1264.45	8752.58	-	10017.03	750M
Migrate to V	17.14	61.49	-	78.63	15M
BigDAWG	1223.28	1.429	3.131	1227.81	3M

`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. For our experiments, we used a configuration that placed the lineitem table on Postgres and the orders table on Vertica.

The query plan executed by BigDAWG for this query was similar to that used for Query 14; it splits the input query into `lineitem` and `orders` queries, and materializes the intermediate results on their respective engines. Then, the resulting `lineitem` tuples (about 10 million) are migrated from Postgres to Vertica. Finally, a query joining the two intermediate tables is executed to produce the final result.

Table III shows that the cross-engine configuration performed worse than Vertica but better than Postgres. Since both the lineitem and orders tables are the largest tables in the dataset, BigDAWG performs especially well compared to the naïve solution of migrating all tables to one engine before executing the query.

3) TPC-H Query 3: This query, shown in Figure 8, is meant to return the 10 unshipped orders with the highest value. However, BigDAWG does not yet support ordering by aliases, so we removed `ORDER BY revenue` from the

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

```
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;
```

Fig. 8. The generated and modified TPC-H Query 3 used in our experiments.

original generated query. For our experiments, we used a configuration that placed the customer table on Postgres and the lineitem and orders tables on Vertica.

The query plan BigDAWG used for this query involved first breaking the query into separate customer, lineitem, and orders subqueries, and materializing the result of running those subqueries on each of their respective engines. Then the resulting 3 million customer tuples were migrated from Postgres to Vertica. The Planner then executed two intermediate nodes, one joining customers and orders tuples, and one joining the lineitem and orders tuples and materialized those results. Finally, a query joining those two intermediate tables was executed to produce the final result.

As shown in Table IV, 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 seen in Table I, taking nearly 1000 seconds to perform.

Furthermore, the intermediate joins following the migration 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 naïve solution of migrating the entire customers table, one of the smaller tables in the dataset, to Vertica and then executing the entire query directly on Vertica outperforms BigDAWG by a factor of about 15.

E. 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 significant 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 for which Vertica is optimized. 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 IV-C.

While the results of our experiments indicate that Vertica alone performs better than cross-engine BigDAWG queries, this is not unexpected. BigDAWG query execution time is dominated by underlying database execution performance; we expect that BigDAWG will perform in between the performance of the underlying systems. In the future, the Monitor module of BigDAWG will automatically highlight such queries and attempt to perform them on a single engine if possible.

These results also show that by identifying and migrating only the tuples that will be needed for the query, BigDAWG outperforms the naïve 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, we believe there is room for improvement in the cross-engine BigDAWG configuration. 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 that serves as the basis for candidate query plans, whose nodes may be combined or re-ordered for optimization purposes. 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 our experiments, the Postgres query optimizer typically performs 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 more expensive to fetch extraneous columns, which can have large performance implications, especially as seen in the execution of TPC-H query 3 in IV-D3. Furthermore, retaining unused

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

columns in intermediate results causes 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.

The unnecessary materialization of intermediate nodes during query execution may also be hindering performance. This behavior makes sense if the intermediate result will be migrated to another database engine, but currently, the query executor does so indiscriminately. 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.

V. Conclusion

In this article, we described the process for integrating new database engines to the relational island of the BigDAWG system. We also presented results from a performance evaluation of the system using a subset of the queries from 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 achieves 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 naïve solution for the two-engine scenario of migrating all tables to one of the engines and computing the final result there, BigDAWG also shows marked improvement by identifying and migrating only the data that is necessary to compute the final result for cross-engine queries with filters.

The experiments detailed in this article use query runtime as a primary measure of performance; while the results do not show the multi-engine configuration outperforming Vertica alone, this is not unexpected. Since the cross-engine BigDAWG query execution time is dominated by underlying database execution performance, we expect that BigDAWG will perform in between the performance of the underlying systems. Areas of future work identified by our findings include eliminating unnecessary materializations of nodes already present on their terminal destination engines, pruning extraneous columns from query plans, and using the binary data transformer[14] for migrations from Postgres to Vertica.

Acknowledgment

This work was made possible in part by the Intel Science and Technology Center for Big Data. The authors would also like to thank Aaron Elmore, Adam Dziedzic, Jennie Rogers, Zuohao She, Kyle O'Brien, and Sam Madden for their help in developing this work.

References

[1] M. Stonebraker and U. 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, (Washington, DC, USA), pp. 2–11, IEEE Computer Society, 2005.

[2] 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*, vol. 39, no. 5, p. 952, 2011.

[3] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik, "The bigdawg polystore system," *SIGMOD Rec.*, vol. 44, pp. 11–16, Aug. 2015.

[4] V. Gadepally, P. Chen, J. Duggan, A. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker, "The bigdawg polystore system and architecture," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pp. 1–6, IEEE, 2016.

[5] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman, "The architecture of scidb," in *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management*, SSDBM'11, (Berlin, Heidelberg), pp. 1–16, Springer-Verlag, 2011.

[6] "Accumulo." https://accumulo.apache.org/.

[7] M. Stonebraker and G. Kemnitz, "The postgres next generation database management system," *Commun. ACM*, vol. 34, pp. 78–92, Oct. 1991.

[8] T. Mattson, V. Gadepally, Z. She, A. Dziedzic, and J. Parkhurst, "Demonstrating the bigdawg polystore system for ocean metagenomic analysis," in *Conference on Innovative Database Research (CIDR)*, 2017.

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

[10] Z. She, S. Ravishankar, and J. Duggan, "Bigdawg polystore query optimization through semantic equivalences," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pp. 1–6, IEEE, 2016.

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

[12] P. Chen, V. Gadepally, and M. Stonebraker, "The bigdawg monitoring framework," in *High Performance Extreme Computing Conference (HPEC)*, 2016.

[13] A. Gupta, V. Gadepally, and M. Stonebraker, "Cross-engine query execution in federated database systems," in *High Performance Extreme Computing Conference (HPEC)*, 2016.

[14] A. Dziedzic, A. J. Elmore, and M. Stonebraker, "Data transformation and migration in polystores," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pp. 1–6, IEEE, 2016.

[15] "Mysql." https://www.mysql.com/.

[16] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear, "The vertica analytic database: C-store 7 years later," *Proc. VLDB Endow.*, vol. 5, pp. 1790–1801, Aug. 2012.

[17] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: A column-oriented dbms," in *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pp. 553–564, VLDB Endowment, 2005.

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

[19] "Docker." https://www.docker.com/.

[20] V. Gadepally, K. O'Brien, A. Dziedzic, A. Elmore, J. Kepner, S. Madden, T. Mattson, J. Rogers, Z. She, and M. Stonebraker, "BigDAWG version 0.1," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pp. 1–6, IEEE, Submitted.

[21] A. Dziedzic, J. Duggan, A. J. Elmore, V. Gadepally, and M. Stonebraker, "Bigdawg: a polystore for diverse interactive applications," in *Data Syst. Interactive Anal. Workshop 2015*, 2015.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE