BigDAWG Version 0.1

Vijay Gadepally*† Kyle O'Brien* Adam Dziedzic§ Aaron Elmore§ Jeremy Kepner*†
Samuel Madden† Tim Mattson¶ Jennie Rogers‡ Zuohao She‡ Michael Stonebraker†

*MIT Lincoln Laboratory †MIT CSAIL ‡Northwestern University §University of Chicago ¶Intel Corporation

***Abstract*—A polystore system is a database management system composed of integrated heterogeneous database engines and multiple programming languages. By matching data to the storage engine best suited to its needs, complex analytics run faster and flexible storage choices helps improve data organization. BigDAWG (Big Data Working Group) is our prototype implementation of a polystore system. In this paper, we describe the current BigDAWG software release which supports PostgreSQL, Accumulo and SciDB. We describe the overall architecture, API and initial results of applying BigDAWG to the MIMIC II medical dataset.**

I. Introduction

Data comes in all shapes and sizes and it is unlikely that any single database management system will be able to efficiently query and analyze diverse data such as text, images, graphs, video, etc [1]. Thus, the need for database systems that leverage heterogenous data stores such as relational systems, key-value stores, graph databases, in-memory databases, array databases, etc.

Polystore systems are of great interest across the research community [2]. They integrate diverse database engines and multiple programming languages presenting them as a single system. The BigDAWG system [3] is our implementation of a polystore database. BigDAWG's architecture consists of four distinct layers: database and storage engines; islands; middleware and API; and applications. Our previous results described the development of core BigDAWG features, and its application to medical [4] and scientific datasets [5]. In this paper, we describe the open source release of BigDAWG (available at http://bigdawg.mit.edu under the BSD-3 license) and an analysis of the performance of BigDAWG queries applied to the MIMIC II medical dataset [6].

The remainder of the article is organized as follows: Section II expands on the concept of a polystore databases and the execution of polystore queries; Section III discusses the overall architecture of the BigDAWG system; Section IV discusses the specifics of our recent software release; Section V describes the current BigDAWG architecture and its application to the MIMIC II dataset, and Section VII describes performance results on an initial BigDAWG implementation. Finally, we conclude and discuss future work in Section VIII.

II. Polystore Systems

The "one size does not fit all" [1] slogan is now famous in the database community. If data storage engines match the data, performance of data intensive applications is greatly enhanced. In our previous work, we show that such benefits can often lead to orders of magnitude performance advantages [3]. Beyond performance, organizations may already have data spread across a number of storage engines and the data must reside in those original systems for policy or performance reasons. Writing connectors to move queries and data across N different systems could require up to $O(N^2)$ connectors, a fact that complicates adoption of polystore techniques.

A polystore system is a database management system (DBMS) that is built on top of multiple, heterogeneous, integrated storage engines. Each of these terms is important to distinguish a Polystore from conventional federated DBMS.

By our definition, a polystore must consist of **multiple** data stores. However, polystores should not to be confused with a distributed DBMS which consists of replicated or partitioned instances of a storage engine sitting behind a single query engine. The key to a polystore is that the multiple storage engines are distinct and accessed separately through their own query engine.

Therefore, storage engines must be **heterogeneous** in a polystore system. If they were the same, it would violate the whole point of polystore systems; i.e., the mapping of data onto distinct storage engines well suited to the features of components of a complex data set.

Finally, the storage engines must be **integrated**. In a federated DBMS, the individual storage engines are independent. In most cases, they are not managed by a single administration team. In a polystore system, the storage engines are managed together as an integrated set. The challenge in designing a polystore system is to balance two often conflicting forces.

- *Location Independence*: A query is written and the system figures out which storage engine it targets.
- *Semantic Completeness*: A query can exploit the full set of features provided by a storage engine.

BigDAWG is a prototype implementation of a polystore system. It is by no means, however, the only such system as a number of groups are also exploring different approaches to polystore DBMS systems [7], [8], [9], [10]. The remainder of this article concentrates on the open source release of the BigDAWG system and a range of implementation details.

III. BigDAWG

Figure 1 describes the overall BigDAWG architecture. At the bottom, we have a collection of disparate storage engines. These engines are organized into a number of *islands*. An

The corresponding author, Vijay Gadepally, can be reached at vijayg [at] ll.mit.edu.

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

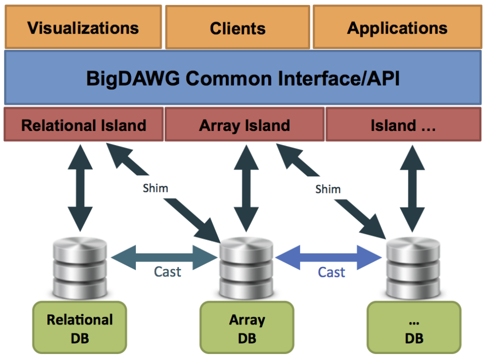

Fig. 1: BigDAWG Architecture.

island is composed of a data model, a set of operations and a set of candidate storage engines. An island provides location independence among its associated storage engines.

A *shim* connects an island to one or more storage engines. The shim translates queries expressed in terms of the island's operations into the native query language of a particular storage engine. For example, the `relational` island may convert a BigDAWG query into an *SQL* query that can be understood by PostgreSQL, MySQL or other relational database.

A key goal of a polystore system is for processing to occur on the storage engine best suited to the features of the data. We expect in typical workloads that queries will produce results best suited to particular storage engines. Hence, BigDAWG needs a capability to move data directly between storage engines. We do this with software components we call *casts*.

The BigDAWG common interface is middleware that supports a common application programming interface to a collection of storage engines via a set of islands. The middleware consists of a number of components:

- Optimizer: parses the input query and creates a set of viable query plan trees with possible engines for each subquery.
- Monitor: uses performance data from prior queries to determine the query plan tree with the best engine for each subquery.
- Executor: figures out how to best join the collections of objects and then executes the query.
- Migrator: moves data from engine to engine when the plan calls for such data motion.

These components are shown graphically in Figure 2. Given an incoming query, the planner parses the query into collections of objects and creates a set of possible query plan trees over collections of engines and objects. The planner sends these trees to the monitor which uses existing performance information to determine a tree with the best engine for each collection of objects (based on previous experience of a similar query). The tree is then passed to the executor which determines the best method to combine the collections of objects and executes the query. The executor can use the migrator to move objects between engines and islands, if

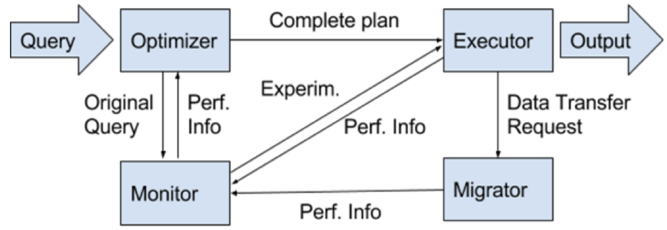

Fig. 2: Internal Components of the BigDAWG Middleware.

required, by the query plan.

IV. Current BigDAWG Release

In March 2017, we released BigDAWG version 0.1 licensed under the terms of the BSD license (http://bigdawg.mit.edu). This initial BigDAWG release supports three open-source database engines: PostgreSQL, Apache Accumulo [11], and SciDB [12] along with support for relational, array and text islands. The middleware is packaged in Docker containers. Figure 3 describes the software components of the BigDAWG software release.

Users primarily interact with the Query Endpoint, which accepts queries, routes them to the Middleware, and responds with results. The Catalog is a PostgreSQL engine containing metadata about the other engines, datasets, islands and connectors managed by the Middleware.

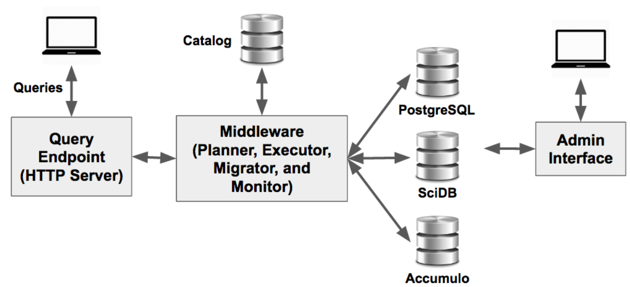

Fig. 3: BigDAWG software release system overview.

The middleware is modular and allows additional islands or database engines to be supported. Soon, we will also support MySQL, the columnar store Vertica [13], and the streaming engine S-Store [14].

In addition to the core software, we developed polystore solutions for medical [4] and scientific [5] data. The current BigDAWG release includes scripts to download and load the MIMIC II medical dataset [6]. Using the default settings, patient history data is inserted into PostgreSQL, physiologic waveform data is inserted to SciDB, and free-form text data is inserted into Accumulo. We also include a number of example BigDAWG queries and an administrative interface to start, stop and view the status of a BigDAWG setup.

V. BigDAWG Components

As shown in Figure 1, BigDAWG is constructed from a number of components. In this paper we describe the middle-

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

ware responsible for receiving queries, translating them into statements that execute on actual data stores, and maintaining information about the system configuration. Other BigDAWG components are described in [3].

The BigDAWG middleware consists of four modules: query planning, performance monitoring, data migration and query execution. Information about the hardware configuration is maintained in the catalog. We describe these BigDAWG modules below.

A. Catalog

The Catalog stores metadata about the system. The Planner, Migrator, and Executor all rely on the Catalog for "awareness" of the BigDAWG components, such as the hostname and IP address of each engine, Engine to Island assignments, and the data objects stored on each engine.

The Catalog is maintained by a PostgreSQL instance with a database called the `bigdawg_catalog`. This database is made up of a number of tables which define items currently managed by the BigDAWG middleware:

- `engines`: Engine names and connection information.
- `databases`: Databases, their engine membership, and connection authentication information.
- `objects`: Data objects (*i.e.*, tables), field-names, and object-to-database membership.
- `shims`: Engines integrated with each island.
- `casts`: the available casts between engines.

Examples of contents of these tables are given in Figure 4. In order for BigDAWG to "see" new engines, databases or objects a user must add entries to relevant tables in the Catalog. BigDAWG includes scripts and an administrative interface to simplify modifications to the Catelog.

B. Planner

The Planner [15] coordinates all query execution. It has a single static function that initiates query processing for a given query and handles the result output. A relevant fragment of code from the Query Planning module is shown below:

```
package istc.bigdawg.planner;

public class Planner {
    public static Response processQuery(
        String userinput, boolean isTrainingMode
    ) throws Exception
}
```

The String `userinput` is the BigDAWG query. The *processQuery()* function first checks if the query is intended to interact with the Catalog. If so, the query is routed to a special processing module to parse and process Catalog-related queries. Otherwise, `processQuery()` parses and processes the query string. When the boolean of `isTrainingMode` is `true`, the Planner performs query optimization by enumerating all possible orderings of execution steps that produce an identical result. Then, the Planner sends the enumeration to the Monitor to gather query execution metrics. The Planner then picks the fastest plan to run returning the result to the Query Endpoint. When `isTrainingMode` is `false`, the Planner consults the Monitor to retrieve the best query plan based on past execution metrics.

Data retrieval queries are passed as inputs to the constructor of a `CrossIslandQueryPlan` object. A `CrossIslandQueryPlan` object holds a nested structure that represents a plan for inter-island query execution. An inter-island query execution is specified by `CrossIslandPlanNode` objects organized in tree structures: the nodes either carry information for an intra-island query or an inter-island migration.

Following the creation of the `CrossIslandQueryPlan`, the Planner traverses the tree structure of `CrossIslandPlanNode` objects and executes the intra-island queries, invokes migrations, and then produces the final result.

C. Migrator

The BigDAWG data migration module [16] exposes a single interface to other modules. Clients provide the connection information for source and destination databases as well as a name of the object (e.g. table, array) to be extracted from the source database, and a name of the object (e.g. table, array) to which the data should be loaded.

```
package istc.bigdawg.migration;

public class Migrator {
  public static MigrationResult migrate(
    ConnectionInfo connectionFrom, String
        objectFrom,
    ConnectionInfo connectionTo, String objectTo,
    MigrationParams migrationParams)
      throws MigrationException;
  }
}
```

Internally, the Migrator identifies the type of the databases by examining the connection information. The `ConnectionInfo` object is merely an *interface* so we check the actual type of the object. The connection object represents a specific database (e.g. PostgreSQL, SciDB, Accumulo or S-Store). Currently, we support migration between instances of PostgreSQL, SciDB, and Accumulo. There is an efficient binary data migration between PostgreSQL and SciDB. We also support binary migration to Vertica, but Vertica does not expose binary export. Future work will build a distributed migrator, tighter integration with S-Store, and a more efficient connection with Accumulo.

D. Executor

The Executor [17] executes intra-island queries through static functions. The static functions create instances of `PlanExecutor` objects that execute individual intra-island queries. A fragment of the executor definition is shown below:

```
package istc.bigdawg.executor;

public class Executor {
    public static QueryResult executePlan(
        QueryExecutionPlan plan,
        Signature sig,
        int index
```

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

eid [PK] serial	name character varying(15)	host character varying(40)	port integer	connection_properties character varying(100)
0	postgres0	bigdawg-postgres-catalog	5400	PostgreSQL 9.4.5
1	postgres1	bigdawg-postgres-data1	5401	PostgreSQL 9.4.5
2	postgres2	bigdawg-postgres-data2	5402	PostgreSQL 9.4.5
3	scidb_local	bigdawg-scidb-data	1239	SciDB 14.12
4	saw ZooKeeper	zookeeper.docker.local	2181	Accumulo 1.6

(a) Example Engines Table

dbid [PK] serial	engine_id serial	name character varying(15)	userid character varying(15)	password character varying(15)
0	0	bigdawg_catalog	postgres	test
1	0	bigdawg_schemas	postgres	test
2	1	mimic2	postgres	test
3	2	mimic2_copy	postgres	test
4	0	tpch	postgres	test
5	1	tpch	postgres	test
6	3	scidb_local	scidb	scidb123
7	4	accumulo	bigdawg	bigdawg

(b) Example Databases Table

oid [PK] serial	name character varying(50)	fields character varying(800)	logical_db serial	physical_db serial
0	mimic2v26.a_chartdurations	subject_id,icustay_id,itemid,	2	3
1	mimic2v26.a_iodurations	subject_id,icustay_id,itemid,	2	3
2	mimic2v26.a_meddurations	subject_id,icustay_id,itemid,	2	3
3	mimic2v26.additives	subject_id,icustay_id,itemid,	2	3

(c) Example Objects Table

shim_id [PK] serial	island_id serial	engine_id serial	access_method character varying(30)
0	0	0	N/A
1	0	1	N/A
2	0	2	N/A
3	1	3	N/A
4	2	4	N/A

(d) Example Shims Table

Fig. 4: BigDAWG catalog tables contain information about what engines, databases, tables and data objects are being managed by the middleware.

```
    ) throws ExecutorEngine.
        LocalQueryExecutionException,
        MigrationException;

    public static QueryResult executePlan(
        QueryExecutionPlan plan
    ) throws ExecutorEngine.
        LocalQueryExecutionException,
        MigrationException;

    public static CompletableFuture<Optional<
        QueryResult>> executePlanAsync(
        QueryExecutionPlan plan,
        Optional<Pair<Signature, Integer>>
            reportValues
    );
}
```

The `PlanExecutor` objects are created from `QueryExecutionPlan` objects that represent execution plans of an intra-island query. A `QueryExecutionPlan`, QEP, holds details of sub-queries required for their execution and a graph of dependency information among the sub-queries. The `PlanExecutor` takes information from a `QueryExecutionPlan` object and issues the sub-queries to their corresponding databases and calls the appropriate `Migrator` classes to migrate intermediate results.

E. Monitor

The monitor [18] manages queries and records query performance of the BigDAWG system. A fragment of the monitor definitions is shown below:

```
class Monitor {
  public static boolean addBenchmarks(Signature
      signature, boolean lean);
  public static List<Long> getBenchmarkPerformance(
      Signature signature);
  public static Signature getClosestSignature(
      Signature signature);
}
```

The `signature` parameter identifies a query. The `addBenchmarks` method adds a new benchmark. If the `learn` parameter is `false`, the benchmark is immediately run over all possible query execution plans (henceforth referred to as QEP). The `getBenchmarkPerformance` method returns a list of execution times for a particular benchmark, ordered in the same order that the benchmark's QEPs are received. The best way to use the module is to add all of the relevant benchmarks using the *addBenchmarks* method and then retrieve information through `getBenchmarkPerformance`.

One of the more useful features is contained in the `getClosestSignature` method, which tries to find the closest matching benchmark for the provided signature. A user can add benchmarks that cover the majority of query use cases and then use the `getClosestSignature` method to find a matching benchmark and compare the QEP times to those of the current signature. If no matching signatures are found, the current signature is added as a new benchmark.

There are many opportunities to enhance this feature to improve the matching, possibly by utilizing machine learning techniques. The public methods in the `Monitor` class are the only API endpoints that should be used. In contrast, the `MonitoringTask` class updates the benchmark timings periodically and should be run in the background through a daemon.

VI. BIGDAWG API

BigDAWG queries are written with the BigDAWG Query language (BQL) which uses a functional syntax. At the highest level, a BigDAWG query follows the following format:

```
bdrel( ... )
```

A function token ('bdrel' in this case) indicates how the syntax within the parenthesis is interpreted. For example, the 'bdrel' function token indicates that this is a query for the relational island and any code between the parenthesis will be interpreted as SQL code.

Five function tokens are defined in BigDAWG. Three function tokens are used to issue queries/sub-queries within individual islands:

- `bdrel` – the query targets the relational island and uses PostgreSQL.
- `bdarray` – the query targets the array island and uses SciDB's AFL query language.
- `bdtext` – the query targets the text island and uses a syntax similar to the Accumulo syntax.

One function token deals with data migration between islands:

- `bdcast` – the query is a cast operation for inter-island data migration.

The remaining function token handles metadata for the polystore system:

- `bdcatalog` – the query targets the BigDAWG catalog using SQL.

The island and migration tokens are often nested within each other, while the `bdcatalog` token is only usable on its own. Sub-queries using the `bdcast` function token are always nested between a pair of island queries and can generally be used in place of an existing data object within a query.

A. Syntax Definitions

BigDAWG Query Syntax:

Code 1: BigDAWG Query Syntax

```
BIGDAWG_SYNTAX ::=
  BIGDAWG_RETRIEVAL_SYNTAX | CATALOG_QUERY

BIGDAWG_RETRIEVAL_SYNTAX ::=
  RELATIONAL_ISLAND_QUERY | ARRAY_ISLAND_QUERY |
      TEXT_ISLAND_QUERY
```

a) Catalog Manipulation: Catalog manipulation queries are used to directly view the content of the catalog.

Code 2: Catalog Manipulation Syntax

```
CATALOG_QUERY ::=
  { bdcatalog( catalog_table_name { [ column_name ]
      [, ...] }) }
  | { bdcatalog( query_applied_to_catalog_database )
      }
```

b) Relational Island: The Relational Island follows the relational data model with data organized into tables. The rows of a table are termed as *tuples* and columns simply as *columns*.

The Relational Island currently supports a subset of SQL used by PostgreSQL. It allows for single-layered *SELECT* query with filter, join, aggregation, sort and limit operations. Column projection, simple arithmetic SQL expression and basic aggregate functions (i.e. count, sum, avg, min and max) are supported.

Code 3: Relational Island Syntax

```
RELATIONAL_ISLAND_QUERY ::=
  bdrel( RELATIONAL_SYNTAX )

RELATIONAL_SYNTAX ::=
  SELECT [ DISTINCT ]
  { * | { SQL_EXPRESSION [ [ AS ] output_name ] [,
      ...] } }
  FROM FROM_ITEM [, ...]
  [ WHERE SQL_CONDITION ]
  [ GROUP BY column_name [, ...] ]
  [ ORDER BY SQL_EXPRESSION [ ASC | DESC ]
  [ LIMIT integer ]

FROM_ITEM ::=
  { table_name | BIGDAWG_CAST } [ [ AS ] alias ]
```

The following is an example of a relational island query; it uses the relational island (`bdrel`) to select 4 entries from the table `mimic2v26.d_patients`.

```
bdrel(select * from mimic2v26.d_patients limit 4)
```

Code 4: Array Island Syntax

```
ARRAY_ISLAND_QUERY ::=
  bdarray( ARRAY_SYNTAX )

ARRAY_SYNTAX ::=
  { scan( ARRAY_ISLAND_DATA_SET ) }
  | { project( ARRAY_ISLAND_DATA_SET [, attribute ]
      [...]) }
  | { filter( ARRAY_ISLAND_DATA_SET,
      SCIDB_EXPRESSION ) }
  | { aggregate( ARRAY_ISLAND_DATA_SET,
      SCIDB_AGGREGATE_CALL [, ...] [, dimension]
      [...] ) }
  | { apply( ARRAY_ISLAND_DATA_SET {, new_attribute,
       SCIDB_NON_AGGREGATE_EXPRESSION} [...] ) }
  | { cross_join( ARRAY_ISLAND_DATA_SET [ as left-
      alias], ARRAY_ISLAND_DATA_SET [ as right-alias
       ] [, [left-alias.]left_dim1, [right-alias.]
      right_dim1] [...] ) }
  | { redimension( ARRAY_ISLAND_DATA_SET, {
      array_name | SCIDB_SCHEMA_DEFINITION } ) }
  | { sort( ARRAY_ISLAND_DATA_SET [, attribute]
      [...] } ) }

ARRAY_ISLAND_DATA_SET ::=
  array_name | ARRAY_ISLAND_SYNTAX | BIGDAWG_CAST
```

c) Array Island: The Array Island follows an array data model, where data is organized into arrays. Arrays are multi-dimensional grids, where each cell in the grid contains a number of fields. Each dimension of an array is referred to as a *dimension* and each field in a cell is termed an *attribute*. Dimensions assume unique values whereas attributes are allowed duplicates. A combination of dimension values across

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

all dimensions in an array uniquely identify an individual cell of attributes.

The Array Island currently supports a subset of SciDB's Array Functional Language (AFL). It allows for project, aggregation, cross_join, filter and schema reform (redimension). The Array Island also allows attribute sorting; however, at the moment, only sort in ascending order is supported.

An example array island query is shown below; it filters all entries in the array myarray with dimension dim1 greater than 150.

```
bdarray(filter(myarray,dim1>150))
```

d) Text Island: The Text Island logically organizes data in tables, and retrieves data in a key-value fashion. This is modeled after the data model of the Accumulo engine. When queried for a certain table, it returns a list of key-value pairs. The key contains row label, column family label, column qualifier label, and a time stamp. The value is a string.

The Text Island query syntax adopts a JSON format using single-quote for labels and entries. The user can issue full table scan or range retrieval queries.

Code 5: Text Island Syntax

```
TEXT_ISLAND_QUERY ::=
  bdtext( TEXT_ISLAND_SYNTAX )

TEXT_ISLAND_SYNTAX ::=
  { 'op' : 'TEXT_OPERATOR', 'table' : '(table_name |
       BIGDAWG_CAST)' [, 'range' : {
      TEXT_ISLAND_RANGE }] }
```

An example text island query is shown below; this query illustrates the text island (bdtext) to scan all entries in the Accumulo table mimic_logs with row keys between r_0001 and r_00015, matching any column family and column qualifier.

```
bdtext({ 'op' : 'scan', 'table' : 'mimic_logs', '
    range' : { 'start' : ['r_0001','',''], 'end' :
    ['r_0015','','']} })
```

e) Inter-Island Cast: Inter-island casts move data between different islands. The differences between two data models can give rise to ambiguities when migrating data between them. When issuing a Cast that invokes an Inter-Island migration, the user avoids such ambiguities by providing the schema used in the destination island.

Code 6: Cast/Migration Syntax

```
BIGDAWG_CAST ::=
  bdcast( BIGDAWG_RETRIEVAL_SYNTAX,
      name_of_intermediate_result, {
    {, POSTGRES_SCHEMA_DEFINITION, relational}
    | {, SCIDB_SCHEMA_DEFINITION, array}
    | {, TEXT_SCHEMA_DEFINITION, text}} )
```

The following is an example of moving data between array and relational islands. This query moves data from PostgreSQL to SciDB. The bdrel() portion of the query selects the columns poe_id and subject_id from table mimic2v26.poe_order. The bdcast() portion of the query tells the middleware to migrate this data to an array called poe_order_copy with schema <subject_id:int32>[poe_id=0:*,10000000,0] in the array island. The final bdarray() portion of the query scans and returns to the user this resultant array in SciDB.

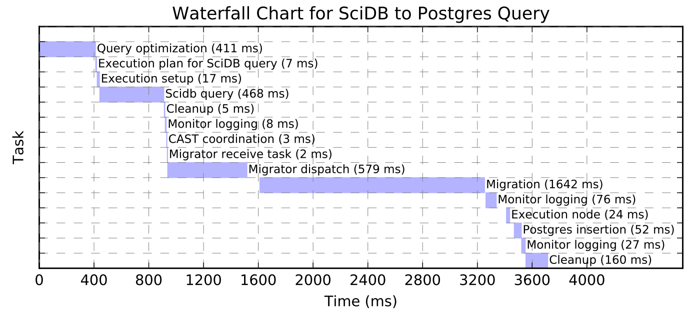

Fig. 5: Task timings for array and relational inter-island query.

```
bdarray(
  scan(
    bdcast(
      bdrel(SELECT poe_id, subject_id FROM mimic2v26
          .poe_order LIMIT 5)
      , poe_order_copy
      , '<subject_id:int32>[poe_id=0:*,10000000,0]'
      , array)))
```

VII. Query Performance Analysis

To characterize BigDAWG queries, we analyze system log data at the DEBUG level with millisecond precision. This data is not meant to benchmark raw performance, since the experiments were run on a single laptop running all database engines in Docker containers. Instead, the goal is to compare relative performance of subtasks within a query and with respect to query complexity.

Figure 5 shows timings an inter-Island query between SciDB and Postgres in which an operation is performed on data stored in SciDB and the resultant array is CAST into Postgres. Tasks are shown on the vertical axis and execution times (i.e. lengths of bars) on the horizontal axis.

This plot shows the relative performance of each stage in a complex query. Note that the bulk of the time is spent waiting for two database execution tasks (*Scidb query*), dispatching the remote procedure call over the network(*Migrator dispatch*), and transmitting data across the network (*Migration*). Together, these tasks consume about 75% of the total execution time. The overhead associated with BigDAWG itself is mostly in the initial query optimization, which is about 10% of the total execution time.

Figure 6 presents sample statistics for query execution times as a boxplot for eight queries run 50 times. The vertical axis shows the query execution time and the horizontal axis shows the query type. This plot shows query latencies and speeds for queries that require and queries that do not require migration between engines. As expected, queries that require migration take more time than single island/engine queries. Inter-island

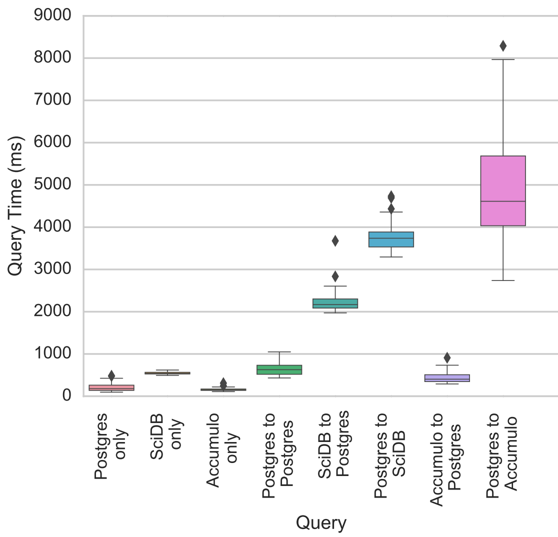

Fig. 6: Task timings for array and relational inter-island query.

migration between different island types require data format translation and transmission dispatching of remote procedures across the network, hence increasing query latency. However, migration between databases in the same island is fast since we can take advantage of native selection and insertion routines.

VIII. Conclusion and Further Work

The future of data analytics will increasingly depend on data distributed among disparate database management systems. Current practice based on federated database engines cannot meet the needs of future high performance since they are largely limited to single data or programming models. Polystore systems go well beyond data federation systems by supporting multiple query languages and disparate yet integrated DBMSs.

In this paper we described the open source release of the BigDAWG polystore version 0.1. As shown in prior results [3] with queries over the MIMIC II medical dataset, BigDAWG provides dramatic performance advantages by using multiple storage engines optimized for particular operations and data models. We build on that earlier study in this paper showing that for complex queries, BigDAWG adds overhead of about 10% of the total execution time. Future work on BigDAWG will expand its capabilities by adding additional islands and storage engines. We plan to support more complex query planning and additional execution capabilities. We also plan to improve inter-island query performance by running monitor logging and cleanup tasks on background threads as well as using multithreaded execution for long-running tasks. Finally, we will look at access control and security [19] techniques for polystores such as BigDAWG.

Acknowledgement

This work was supported in part by the Intel Science and Technology Center for Big Data. The authors wish to thank ISTC collaborators Kristin Tufte, Jeff Parkhurst, Stavros Papadopoulos, Nesime Tatbul, Magdalena Balazinska, Bill Howe, Jeffrey Heer, David Maier, Tim Kraska, Ugur Cetintemel, and Stan Zdonik.

References

[1] M. Stonebraker and U. Çetintemel, ""one size fits all": an idea whose time has come and gone," in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*. IEEE, 2005, pp. 2–11.

[2] E. Begoli, D. Kistler, and J. Bates, "Towards a heterogeneous, polystore-like data architecture for the us department of veteran affairs (VA) enterprise analytics," in *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 2016, pp. 2550–2554.

[3] 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*. IEEE, 2016, pp. 1–6.

[4] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska *et al.*, "A demonstration of the bigdawg polystore system," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1908–1911, 2015.

[5] 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.

[6] 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.

[7] V. Spyropoulos, C. Vasilakopoulou, and Y. Kotidis, "Digree: A middleware for a graph databases polystore," in *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 2016, pp. 2580–2589.

[8] A. Maccioni, E. Basili, and R. Torlone, "Quepa: Querying and exploring a polystore by augmentation," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 2133–2136.

[9] E. Kharlamov, T. Mailis, K. Bereta, D. Bilidas, S. Brandt, E. Jimenez-Ruiz, S. Lamparter, C. Neuenstadt, O. Özçep, A. Soylu *et al.*, "A semantic approach to polystores," in *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 2016, pp. 2565–2573.

[10] S. Dasgupta, K. Coakley, and A. Gupta, "Analytics-driven data ingestion and derivation in the awesome polystore," in *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 2016, pp. 2555–2564.

[11] J. Kepner and V. Gadepally, "Adjacency matrices, incidence matrices, database schemas, and associative arrays," in *International Parallel & Distributed Processing Symposium Workshops (IPDPSW). IEEE*, 2014.

[12] P. G. Brown, "Overview of scidb: large scale array storage, processing and analysis," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 963–968.

[13] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear, "The vertica analytic database: C-store 7 years later," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1790–1801, 2012.

[14] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul *et al.*, "S-store: A streaming newsql system for big velocity applications," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1633–1636, 2014.

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

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

[17] A. M. Gupta, V. Gadepally, and M. Stonebraker, "Cross-engine query execution in federated database systems," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE, 2016, pp. 1–6.

[18] P. Chen, V. Gadepally, and M. Stonebraker, "The bigdawg monitoring framework," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE, 2016, pp. 1–6.

[19] V. Gadepally, B. Hancock, B. Kaiser, J. Kepner, P. Michaleas, M. Varia, and A. Yerukhimovich, "Computing on masked data to improve the security of big data," in *Technologies for Homeland Security (HST), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 1–6.