BigDAWG Documentation

Release 0.1

BigDAWG Developers

Mar 26, 2017

Contents

CHAPTER 1

Welcome to BigDAWG documentation

Introduction

The Intel Science and Technology Center for Big Data is developing an open-source reference implementation of a Polystore database. The BigDAWG (Big Data Working Group) system supports heterogeneous database engines, multiple programming languages and complex analytics for a variety of workloads.

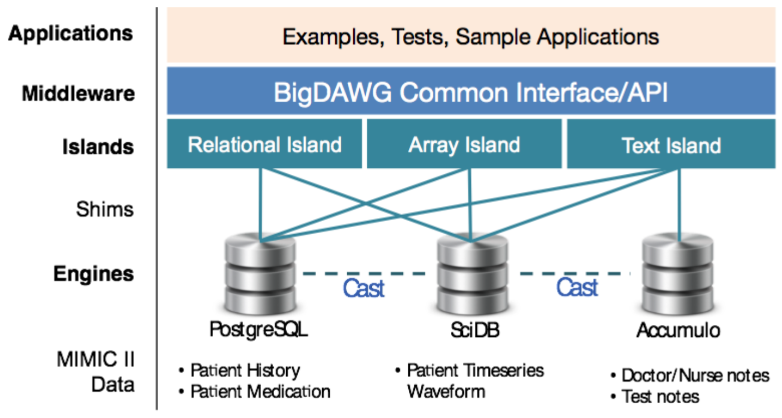


Fig. 1.1: BigDAWG Architecture

This BigDAWG release contains our initial prototype of a polystore middleware as well as support for 3 database engines: PostgreSQL, SciDB, and Accumulo. The architecture for this release is shown above.

Our goal with this release is to give end-users and database researchers an idea about what a Polystore database looks like. For the most part, we hope that you will download the release, experiment with the data we have distributed and create your own queries. Please do reach out to us if you have some bigger goals in mind or if you run into any issues while using this release - we are happy to help you navigate.

A simple example

Before we get into the details of what BigDAWG is, here is a very simple query example. This query execute a relational island query on a polystore storing MIMIC II data in the BigDAWG language:

```
curl -X POST -d "bdrel(select * from mimic2v26.d_patients limit 4;)" http://
↪localhost:8080/bigdawg/query/
```

Output:

```
subject_id  sex dob dod hospital_expire_flg
1039      M   3063-10-05 00:00:00.0   3147-04-05 00:00:00.0   Y
1010      F   2620-12-07 00:00:00.0   2688-07-30 00:00:00.0   Y
1000      M   2442-05-11 00:00:00.0   2512-03-02 00:00:00.0   Y
1038      M   2747-06-02 00:00:00.0   2807-11-13 00:00:00.0   N
```

For further details on what *islands* are, please refer to the *Introduction and Overview* section or refer to any one of our numerous publications that describe BigDAWG.

Get the code

What you need to get started is in :ref:‘getting-started‘section.

For (future) reference, the short version is:

The source source is available on GitHub.

Within the Docker toolbox, go into the *provisions* directory of the above repository and run setup_bigdawg_docker.sh:

```
./setup_bigdawg_docker.sh
```

This should start up three databases and middleware. You should now be able to execute a query such as the one above in a seperate window.

Contributing

We hope that you find this area of research as interesting as we do! We look forward to community invovlement. If you are interested in contributing, please let us know, we have many ideas where we could use help.

We have many ideas for new contributors such as adding new engines, islands and improving middleware capabilities. If this sounds interesting, let us know and we can set up a time to chat.

Website: http://bigdawg.mit.edu

The mailing list for the project is located at google groups: http://groups.google.com/group/bigdawg To contact the BigDAWG developers: bigdawg-help@mit.edu

Table of Contents

Introduction and Overview

Team

BigDAWG is an open source project from researchers within the Intel Science and Technology Center for Big Data (ISTC). Everything we do at the ISTC is open intellectual property so anyone is free to use whatever we produce.

The ISTC is based at MIT but includes researchers from Brown University, the University of Chicago, Northwestern University, the University of Washington, Portland State University, Carnegie Mellon University, the University of Tennessee, and, of course, Intel.

Polystore Systems

The slogan is now famous in the database community. “One size does not fit all”. If data storage engines match the data, performance of data intensive applications are greatly enhanced. We’ve done significant performance analsys and have found that using the right storage engine for the job can give you orders of magnitude in performance advantage. Even beyond performance advantages, often organizations already have their data spread across a number of storage engines. Writing connectors across N different systems can lead to a lot of work for developers and make the cost of adding a new system very high.

This has led us to develop database technologies we call “Polystore Systems.” A polystore system is any 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.

Obviously, a polystore must consist of **multiple** data stores. However, polystores should not to be confused with a distributed DBMS which consists of replicated 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. This is key since it means that in a polystore system, you can modify engines or the middleware managing them such that “the whole is greater than the sum of their parts.”

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.

The BigDAWG project described in this document is our reference implementation of this polystore concept. As we will see in the next section, BigDAWG uses the concepts of “islands” to balance these forces.

BigDAWG Approach

Figure 1 describes the overall BigDAWG architecture. This figure is a representation of the BigDAWG polystore system integrated with higher level components to solve end-user applications. At the bottom, we have a collection of disparate storage engines (we make no assumption about the data model, programming model, etc. of each of these engines). These are organized into a number of *islands*. An 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 is basically a translator that maps queries expressed in terms of the operations defined by an island into the native query language of a particular storage engine.

A key goal of a polystore system is for the 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,

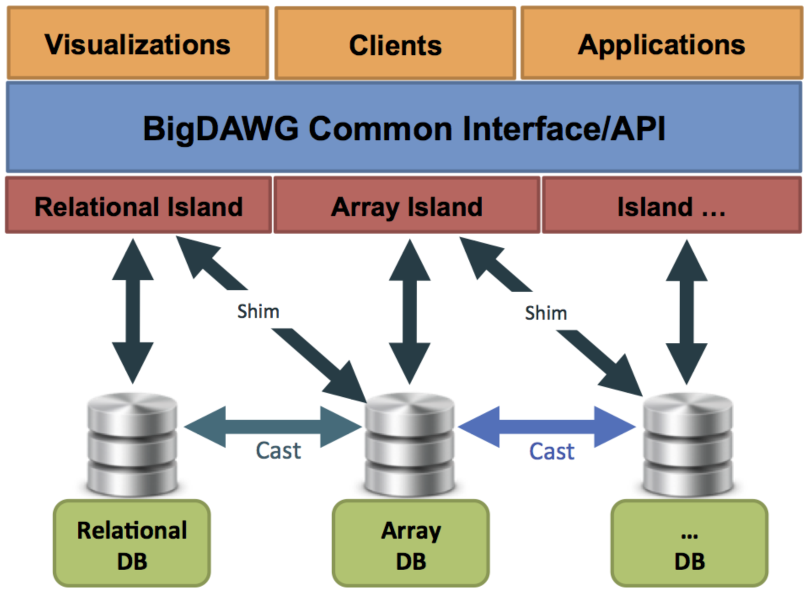


Fig. 1.2: BigDAWG Architecture

BigDAWG needs a capability to move data directly between storage engines. We do this with software components we call *casts*.

Major BigDAWG Components

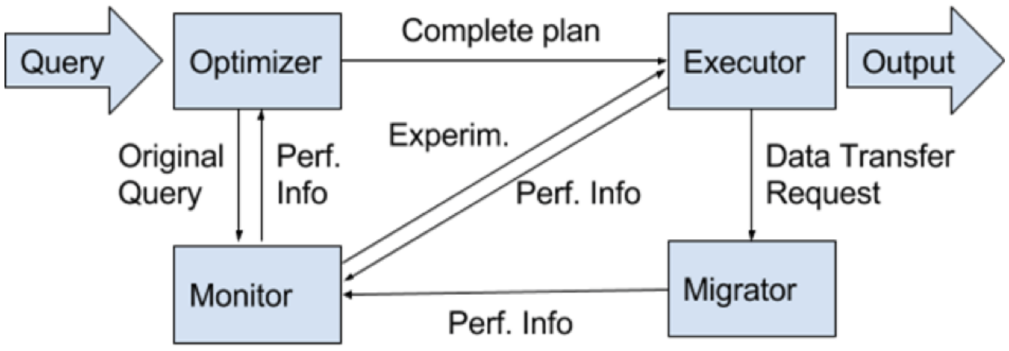


Fig. 1.3: Internal Components of the BigDAWG Middleware

BigDAWG is at its core middleware that supports a common application programming interface (API) to a collection of storage engines. The middleware contains a number of key elements:

- 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.

Each of these components will be described in more detail in a later section.

MIMIC II dataset

to demonstrate BigDAWG in action, we are using data collected by the PhysioNet group (https://physionet.org/mimic2/). The MIMIC II dataset contains medical data collected from medical ICUs over a period of 8 years. The MIMIC II datasets consists of structured patient data (for example, things filled in an electronic health record), unstructured data (for example, of the nurse/doctor reports), and time-series waveform data (for example, data collected from different machines one may be connected to while in the EHR). The MIMIC II dataset is a great example of where a polystore solution may work well. The structural parts of the data can sit well in a traditional relational database, the free-form text in a key-value store and the time series waveforms in an array database.

In this release, we provide simple scripts to download this data and load it into appropriate databases. While we only leveraging data the unrestricted parts of the data that do not require registration, we recommend you take a look at Getting Access to the Full Dataset . Also, if you are using any of their data in your results, please be sure to cite them appropriately.

Getting Started with BigDAWG

This section describes how to start a BigDAWG cluster, load an example dataset, and run several example queries.

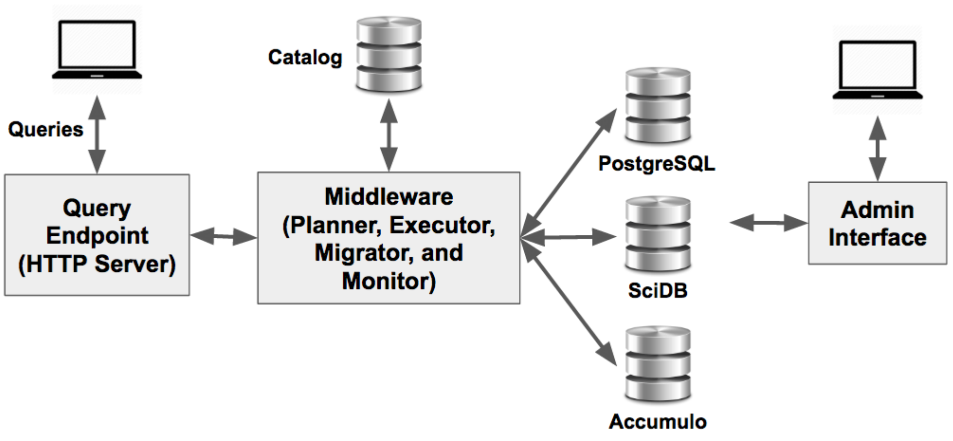

Queries
Catalog
Query Endpoint (HTTP Server)
Middleware (Planner, Executor, Migrator, and Monitor)
PostgreSQL
SciDB
Accumulo
Admin Interface

Fig. 1.4: BigDAWG Cluster Components

A BigDAWG cluster consists of the Middleware, Query Endpoint, Catalog, and multiple database engines. You can learn more about these components in the *BigDAWG Middleware Internal Components* section.

The purpose of this section is to guide you through the process of setting up a BigDAWG cluster with Docker, the open-source technology that allows you to deploy applications inside software containers. You will pull baseline *images* from our Dockerhub repository, run images as instantiated *containers*, and then run scripts to populate the engines with test data. The current release of BigDAWG includes images for PostgreSQL, SciDB, and Accumulo.

A video demonstration of these steps is also available to watch.

Prerequisites

To complete this guide, you will need basic knowledge of working with your computer's command prompt/terminal, Docker, and Linux commands. You will also need your computer's port 8080 available and will need administrator privileges on your system to install Docker.

Compatible Docker Installation

To follow the steps in this section, you will need to first install Docker on your system. If your system is running Mac OSX or Windows, you should install Docker Toolbox. Follow the download and installation steps from the Docker website.

Note: BigDAWG has been tested on these versions of Docker:

- Docker version 1.11.1, build 5604cbe (Tested on Ubuntu 14.04)
- Docker version 1.12.1, build 6f9534c (Tested on Docker Toolbox for Mac, version 0.8.1, build 41b3b25)
- Docker version 1.12.6, build 78d1802 (Tested on Docker Toolbox for Mac)

Note: Do not use "Docker for Mac" or "Docker for Windows", which are two alternative Docker applications, because of known networking limitations that interfere with this example. If your system is runnig Linux, then install Docker for Linux.

BigDAWG source code

Obtain the source code by cloning the git repository:

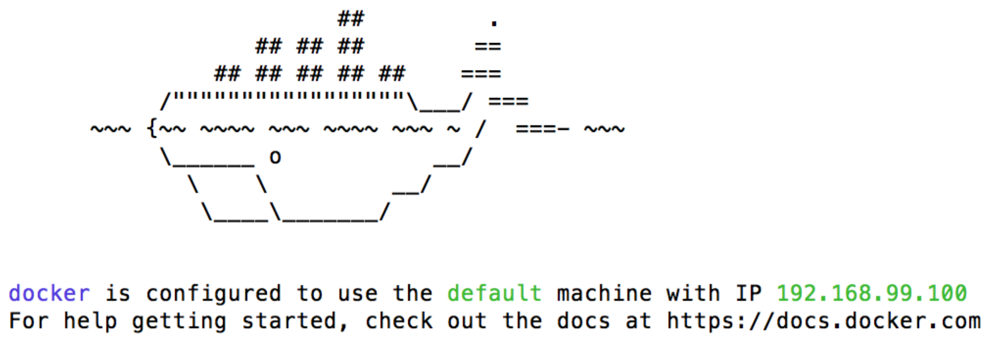
```
git clone https://github.com/bigdawg-istc/bigdawg.git
```

Alternatively, download the code directly from the website https://github.com/bigdawg-istc/bigdawg.git

BigDAWG Cluster Setup Steps

(Mac and Windows only) Open a Quickstart Terminal to Execute Docker Commands

Launch the Docker Quickstart Terminal application, which was installed when installing Docker Toolbox (this initialization can take some time). Launching this application will run a Docker host VM and open an initialized terminal window. Without this terminal, you will not be able to execute `docker` commands.



```
                        ##         .
                  ## ## ##        ==
               ## ## ## ## ##    ===
           /"""""""""""""""""\___/ ===
      ~~~ {~~ ~~~~ ~~~ ~~~~ ~~~ ~ /  ===- ~~~
           \______ o           __/
             \    \         __/
              \____\_______/


docker is configured to use the default machine with IP 192.168.99.100
For help getting started, check out the docs at https://docs.docker.com
```

Fig. 1.5: Docker Quickstart Terminal Successfully Initialized

The status shown above means that Docker was started successfully.

Navigate to the "provisions" directory of the source code root

The source code root is a directory called "bigdawg". All scripts executed in this tutorial assume that you are in the `bigdawg/provisions` directory.

Run the Docker setup script:

```
./setup_bigdawg_docker.sh
```

This script take will start a BigDAWG cluster using Docker containers. It can take up to 15-30 minutes to complete depending on your computer resources and internet connection. The script works in the following stages:

1. Create a Docker network called `bigdawg` that allows the containers to communicate with each other.
2. Pull "base" docker images from Docker Hub that encapsulate the database engines but contain no data.
3. Run the images as instantiated containers.
4. Download publically-available MIMIC II data. The BigDAWG project does not ship with any of data itself, so all data is downloaded from external sources.
5. Execute scripts on the contianers to insert data into the engines.
6. Start the BigDAWG Middleware on each container, and accept queries on the `bigdawg-postgres-catalog` container.

After the setup script completes, you will get a message:

```
Starting HTTP server on: http://bigdawg-postgres-catalog:8080/bigdawg/
2017-03-21 14:17:01,873 2767 istc.bigdawg.network.NetworkIn.receive(NetworkIn.
↪java:39) [pool-2-thread-1] null DEBUG istc.bigdawg.network.NetworkIn - tcp://*:9991
2017-03-21 14:17:02,072 2966 istc.bigdawg.network.NetworkIn.receive(NetworkIn.
↪java:43) [pool-2-thread-1] null DEBUG istc.bigdawg.network.NetworkIn - Wait for the␣
↪next request from a client ...
Mar 21, 2017 2:17:23 PM org.glassfish.grizzly.http.server.NetworkListener start
INFO: Started listener bound to [bigdawg-postgres-catalog:8080]
Jersey app started with WADL available at http://bigdawg-postgres-catalog:8080/
↪bigdawg/application.wadl
Hit enter to stop it...
```

If you hit any key, the Middleware execution will quit. Therefore, make sure to run any additional commands in a separate termainal window.

Optional setup verification

As an optional step, you can verify that the images were pulled successfully and check their running status.

To do this, create a separate Docker Quickstart terminal and run the following commands:

Check the status of all images:

```
docker images
```

```
user@local:~$ docker images
REPOSITORY          TAG       IMAGE ID       CREATED           SIZE
bigdawg/accumulo    latest    804fa44f5eb4   2 seconds ago     1.656 GB
bigdawg/scidb       latest    c1b578c504bb   8 seconds ago     1.237 GB
bigdawg/postgres    latest    1a2600f05cbb   12 seconds ago    1.086 GB
```

You should see the three images as shown above if the `pull` (phase 2 above) was successful.

Check the status of all running containers:

```
docker ps
```

```
user@local:~$ docker ps
CONTAINER ID   IMAGE               STATUS         PORTS                                ␣
↪              NAMES
```

```
ef66f13c4694   bigdawg/accumulo   Up 1 minute   0.0.0.0:42424->42424/tcp                ␣
↪              bigdawg-accumulo-proxy
3e02a26c9da5   bigdawg/accumulo   Up 1 minute   0.0.0.0:9999->9999/tcp, 0.0.0.0:50095-
↪>50095/tcp    bigdawg-accumulo-master
13deae26bff7   bigdawg/accumulo   Up 1 minute   0.0.0.0:9997->9997/tcp                  ␣
↪              bigdawg-accumulo-tserver0
c6e6b8185d7f   bigdawg/accumulo   Up 1 minute   0.0.0.0:2181->2181/tcp                  ␣
↪              bigdawg-accumulo-zookeeper
7d3135d17a7e   bigdawg/accumulo   Up 1 minute                                           ␣
↪              bigdawg-accumulo-namenode
3b1710639c09   bigdawg/scidb      Up 1 minute   0.0.0.0:1239->1239/tcp                  ␣
↪              bigdawg-scidb-data
4d119d50458c   bigdawg/postgres   Up 1 minute   0.0.0.0:5402->5402/tcp                  ␣
↪              bigdawg-postgres-data2
626ba8425e5b   bigdawg/postgres   Up 1 minute   0.0.0.0:5401->5401/tcp                  ␣
↪              bigdawg-postgres-data1
e4fe27b0c8ed   bigdawg/postgres   Up 1 minute   0.0.0.0:5400->5400/tcp, 0.0.0.0:8080->
↪8080/tcp      bigdawg-postgres-catalog
```

You should see all the containers running as shown above if the `run` (phase 3 above) was successful.

Run Example Queries

Warning: These commands will not work if you are using a VPN connection or cannot access the Docker host IP address. If VPN is necessary for your system, contact us for tips that you may be able to use to work around this.

Warning: Your system must have port 8080 available for the Middleware to initialize successfully.

Once the containers are running, the Catalog container will run the Query Endpoint (a simple HTTP server) listening on port 8080. The container is configured to publish its port 8080 to the Docker VM's port 8080, so that queries sent to that port will be routed to the Query Endpoint. You can then submit queries to this port like so:

```
$ curl -X POST -d "bdrel(select * from mimic2v26.d_patients limit 4;)" http://192.168.
↪99.100:8080/bigdawg/query/
```

Here, we are using `curl`, a shell command, to handle requests and responses to and from a web server, in this case the Query Endpoint, over the HTTP protocol.

Example Queries

In this section, we describe a few queries on the MIMIC II dataset that you can execute once you have successfully completed the above steps.

All queries use the following syntax:

```
$ curl -X POST -d "<query-goes-here>" http://192.168.99.100:8080/bigdawg/query/
```

We are making a `POST` request to send the query string as data to the Query Endpoint at the resource `/bigdawg/query/`. The IP address `192.168.99.100` is used by the Docker host VM, which is forwarding its port 8080 to the container running the Query Endpoint.

1) postgres only

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

This query uses the relational island (`bdrel`) to select 4 entries from the table `mimic2v26.d_patients`.

Here is the full curl command:

```
curl -X POST -d "bdrel(select * from mimic2v26.d_patients limit 4;)" http://192.168.
↪99.100:8080/bigdawg/query/
```

2) scidb only

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

This query uses the array island (`bdarray`) to filter all entries in the array `myarray` with `dim1` greater than 150. *Note* The SciDB connector is in beta mode.

Here is the full curl command:

```
curl -X POST -d "bdarray(filter(myarray,dim1>150));" http://192.168.99.100:8080/
↪bigdawg/query/
```

3) accumulo only

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

This query uses the text island (`bdtext`) to scan all entries in the Accumulo table `mimic_logs` with row keys between `r_0001` and `r_00015`.

Here is the full curl command:

```
curl -X POST -d "bdtext({ 'op' : 'scan', 'table' : 'mimic_logs', 'range' : { 'start'␣
↪: ['r_0001','',''], 'end' : ['r_0015','','']} });" http://192.168.99.100:8080/
↪bigdawg/query/
```

4) postgres to postgres

```
bdrel(select * from mimic2v26.additives,mimic2v26.admissions where mimic2v26.
↪additives.subject_id=mimic2v26.admissions.subject_id limit 10)
```

This query joins data stored in two seperate postgres instances. Essentially, the tables `mimic2v26.additives`, `mimic2v26.admissions` are split among two different postgres instances.

Here is the full curl command:

```
curl -X POST -d "bdrel(select * from mimic2v26.additives,mimic2v26.admissions where␣
↪mimic2v26.additives.subject_id=mimic2v26.admissions.subject_id limit 10;)" http://
↪192.168.99.100:8080/bigdawg/query/
```

5) scidb to postgres

```
bdrel(select * from bdcast( bdarray(filter(myarray,dim1>150)), tab6, '(i bigint, dim1␣
↪real, dim2 real)', relational))
```

This query moves data from scidb to postgres. The `bdarray()` portion of the query filters all entries in the scidb array `myarray` with `dim1`>150. The `bdcast()` portion of the query tells the middleware to migrate this resultant array to a table called `tab6` with schema `(i bigint, dim1 real, dim2 real)` to a database in the relational island. The final `bdrel()` portion of the query selects all entries from this resultant table in postgres.

Here is the full curl command:

```
curl -X POST -d "bdrel(select * from bdcast( bdarray(filter(myarray,dim1>150)), tab6,
↪'(i bigint, dim1 real, dim2 real)', relational))" http://192.168.99.100:8080/
↪bigdawg/query/
```

6) postgres to scidb

```
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)))
```

This query moves data from postgres to scidb. The `bdrel()` portion of the array selects the columns `poe_id, subject_id FROM 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 this resultant array in scidb. *Note* The SciDB connector is in beta mode. We are having some problems with the current SciDB JDBC connector in which delivery of result arrays where dimensions span more than one chunk can lead to an error.

Here is the full curl command:

```
curl -X POST -d "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)));" http://192.168.99.100:8080/bigdawg/query/
```

7) accumulo to postgres

```
bdrel(select * from bdcast(bdtext({ 'op' : 'scan', 'table' : 'mimic_logs', 'range' :
↪{ 'start' : ['r_0001','',''], 'end' : ['r_0020','','']} }), tab1, '(cq1 text, mimic_
↪text text)', relational))
```

This query moves data from accumulo to postgres. The `bdtext()` portion of the query scans the accumulo table `mimic_logs` from row keys `r_0001` to `r_00020`. The `bdcast()` portion of the query tells the middleware to migrate these resultant key-value pairs to a table called `tab1` with schema `(cq1 text, mimic_text text)` in the relational island. The final `bdrel()` portion of the query selects all entries from this resultant table.

Here is the full curl command:

```
curl -X POST -d "bdrel(select * from bdcast(bdtext({ 'op' : 'scan', 'table' : 'mimic_
↪logs', 'range' : { 'start' : ['r_0001','',''], 'end' : ['r_0020','','']} }), tab1,
↪'(cq1 text, mimic_text text)', relational))" http://192.168.99.100:8080/bigdawg/
↪query/
```

8) postgres to accumulo

```
bdtext({ 'op' : 'scan', 'table' : 'bdcast(bdrel(select * from mimic2v26.icd9 limit 4),
↪ res, '', text)'})
```

This query moves data from postgres to accumulo. The `bdrel()` portion of the query select 4 entries from the table `mimic2v26.icd9`. The `bdcast()` portion of the query tells the middleware to migrate these entries to a text island table called `res`. Finally, the `bdtext()` portion of hte array scans this resultant table.

Here is the full curl command:

```
curl -X POST -d "bdtext({ 'op' : 'scan', 'table' : 'bdcast(bdrel(select * from␣
↪mimic2v26.icd9 limit 4), res, '', text)'})" http://192.168.99.100:8080/bigdawg/
↪query/
```

Output Logs

All logging is saved to a Postgres database called `logs` which resides on the `bigdawg-postgres-catalog` container. You can attach to the container by running the following Docker command in a separate Quickstart Terminal:

```
user@local:~$ docker exec -it bigdawg-postgres-catalog bash
postgres@bigdawg-postgres-catalog:/$
```

This command will attach to the `bigdawg-postgres-catalog` container, and logs you in as the user `postgres`, so you can execute psql queries from there.

```
postgres@bigdawg-postgres-catalog:/$ psql
psql (9.4.10)
Type "help" for help.

postgres=# \l
                                  List of databases
      Name        |  Owner   | Encoding | Collate |  Ctype  |   Access privileges
------------------+----------+----------+---------+---------+-----------------------
 bigdawg_catalog | postgres | UTF8     | C.UTF-8 | C.UTF-8 |
 bigdawg_schemas | postgres | UTF8     | C.UTF-8 | C.UTF-8 |
 logs            | pguser   | UTF8     | C.UTF-8 | C.UTF-8 |
 postgres        | postgres | UTF8     | C.UTF-8 | C.UTF-8 |
 template0       | postgres | UTF8     | C.UTF-8 | C.UTF-8 | =c/postgres          +
                 |          |          |         |         | postgres=CTc/postgres
 template1       | postgres | UTF8     | C.UTF-8 | C.UTF-8 | =c/postgres          +
                 |          |          |         |         | postgres=CTc/postgres
(6 rows)

postgres=# \c logs
You are now connected to database "logs" as user "postgres".

logs=# SELECT * FROM logs LIMIT 5;
 id | user_id |          time           |          logger          | level |          ␣
↪       message
----+---------+-------------------------+--------------------------+-------+----------
↪--------------------------------
  1 |         | 2017-03-21 20:36:11.342 | istc.bigdawg.LoggerSetup | INFO  | Logging␣
↪was configured!
  2 |         | 2017-03-21 20:36:11.427 | istc.bigdawg.Main        | INFO  | Starting␣
↪application ...
  3 |         | 2017-03-21 20:36:11.435 | istc.bigdawg.Main        | INFO  |␣
↪Connecting to catalog
  4 |         | 2017-03-21 20:36:11.452 | istc.bigdawg.Main        | INFO  | Checking␣
↪registered database connections
  5 |         | 2017-03-21 20:36:11.601 | istc.bigdawg.Main        | DEBUG | args 0:␣
↪bigdawg-scidb-data
(5 rows)

logs=# \q
postgres@bigdawg-postgres-catalog:/$ exit
user@local:~$
```

The `\q` command exits `psql` and returns you to the `bigdawg-postgres-catalog` container's shell. The subsequent `exit` command returns you to your local system shell.

Exporting logs

You can also dump the logs from the container into a text file on your local system with the following command:

```
docker exec -it bigdawg-postgres-catalog pg_dump -a -d logs -t logs > logs.txt
```

This will write the contents of the `logs` table of the `logs` database to a file called logs.txt on your local system.

Viewing the Catalog

You may view the contents of the Catalog database by sending queries to the Query Endpoint using the `bdcatalog()` syntax.

As an example, you may view the `engines` table of the Catalog database by executing the following:

```
curl -X POST -d "bdcatalog(select * from catalog.engines);" http://192.168.99.
↪100:8080/bigdawg/query/

eid name          host                     port  connection_properties
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
```

See the *Catalog Manipulation* section for more details about the query language, and see the *Catalog* section for more details about the contents and purpose of the Catalog.

Shutdown

When finished, stop and remove the containers:

```
./cleanup_containers.sh
```

Stopping a container means that the container ceases execution, but is still visible in the `docker ps -a` output list. *Removing* a container deletes all additional filesystem layers added to the associated image. In either case, the **image** is still present on your system, so that it doesn't need to be pulled from the Docker repository again.

After stopping and removing, you must run the `./setup_bigdawg_docker.sh` script to start the BigDAWG cluster again.

Additionaly, if you're using Docker Toolbox, you can stop the VM running Docker with the following command:

```
docker-machine stop default
```

Docker Networking and Container Reference

Below is a list of the Docker containers and the primary functions they serve:

bigdawg-postgres-catalog Runs the Catalog, Middleware, and Query Endpoint. The Query Endpoint listens for queries on `bigdawg-postgres-catalog` and port `8080`

bigdawg-postgres-data1 Runs PostgreSQL loaded with the MIMIC II patient dataset

bigdawg-postgres-data2 Runs PostgreSQL loaded with a copy of the Mimic II patient dataset. Used for demonstrating migration between 2 PostgreSQL instances

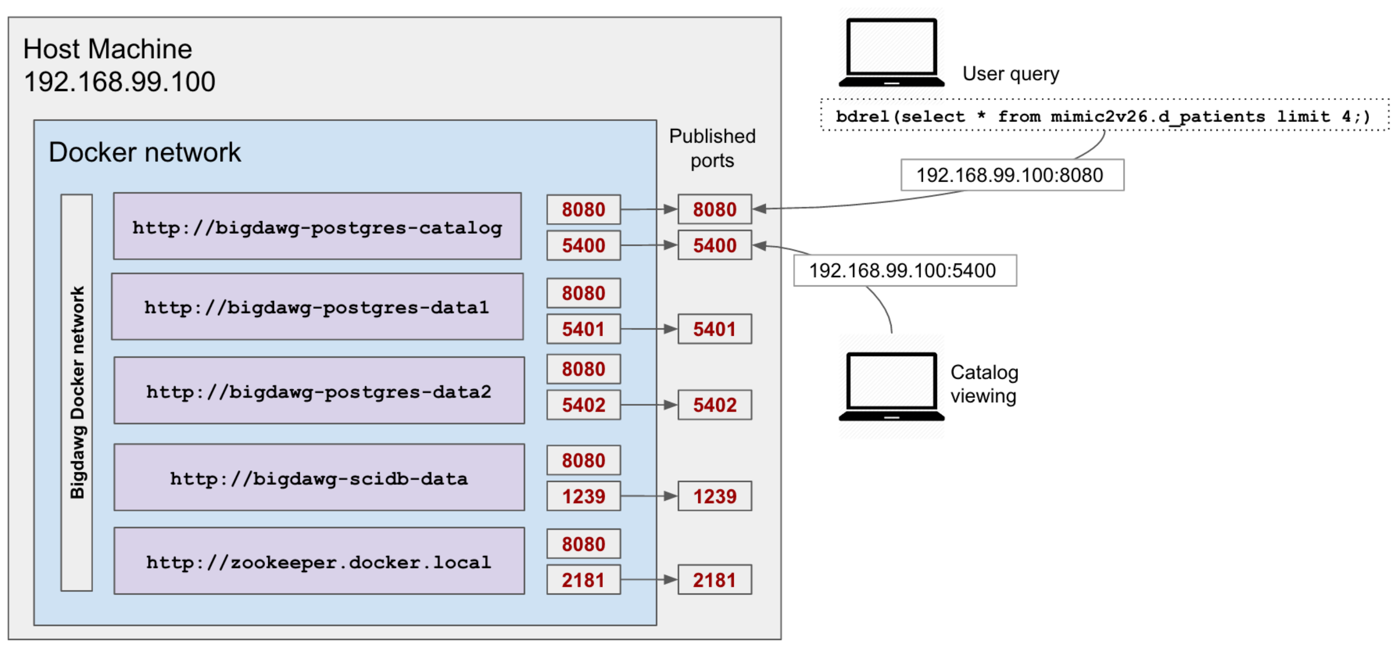


Fig. 1.6: Docker Networking Configuration

bigdawg-scidb Runs SciDB with MIMIC II waveform data

Accumulo containers: several containers support the Accumulo stack: bigdawg-accumulo-master: Master server bigdawg-accumulo-tserver0: Handles client reads and writes bigdawg-accumulo-zookeeper bigdawg-accumulo-namenode bigdawg-accumulo-proxy

In order for the containers to communicate with each other, they are connected to a Docker network named `bigdawg`, which was created with the `docker network create` command. In addition, each container *exposes* any required ports for other containers to connect to and *publishes* ports, which makes them available to both other containers and the Docker Host. This is all handled by the startup scripts above.

Below is a listing of the ports published by each container.

hostname: bigdawg-postgres-catalog port 5400 for postgres, 8080 for accepting bigdawg queries

hostname: bigdawg-postgres-data1 port 5401 for postgres

hostname: bigdawg-postgres-data2 port 5402 for postgres

hostname: bigdawg-scidb port 1239 for scidb, 49901 for ssh

hostname: accumulo-data-master port 9999 for Master thrift server, 50095 for Monitor service

hostname: accumulo-data-tserver0 port 9997 for TabletServer thrift server

hostname: accumulo-data-tserver1 (no ports)

hostname: accumulo-data-zookeeper port 2181 for zookeeper client connections

hostname: accumulo-data-namenode (no ports)

hostname: accumulo-data-proxy (no ports)

If using docker-toolbox, the Docker Host will have IP address 192.168.99.100, which you can check using this command:

```
$ docker-machine ip default
> 192.168.99.100
```

Otherwise, if on Linux, the Docker Host IP is your own localhost IP.

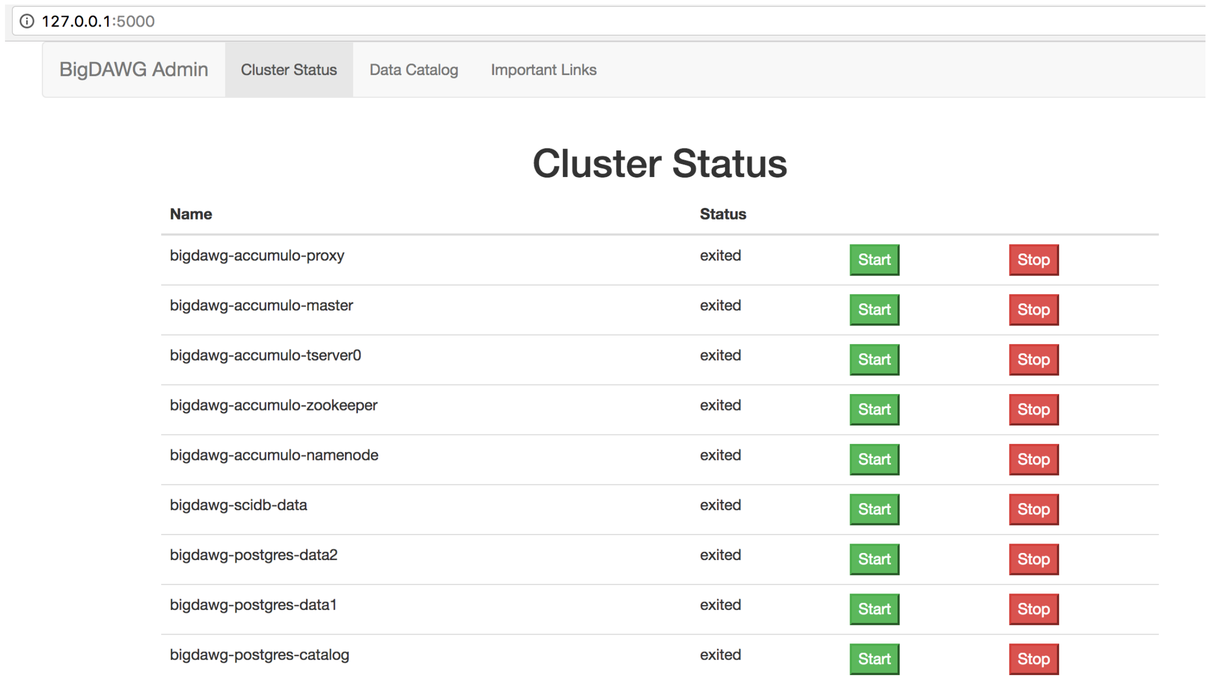
MIMIC II dataset

For the above examples, we are using data collected by the PhysioNet group (https://physionet.org/mimic2/). While we are only leveraging data the unrestricted parts of the data that do not require registration, we recommend you take a look at Getting Access to the Full Dataset . Also, if you are using any of their data in your results, please be sure to cite them appropriately.

Install the Administrative Web Interface

A very basic administrative web interface is included with this release, which will let you see the status of the Big-DAWG cluster of databases, start and stop containers, and view the Catalog objects table.

You can view a video demonstration here



Cluster Status

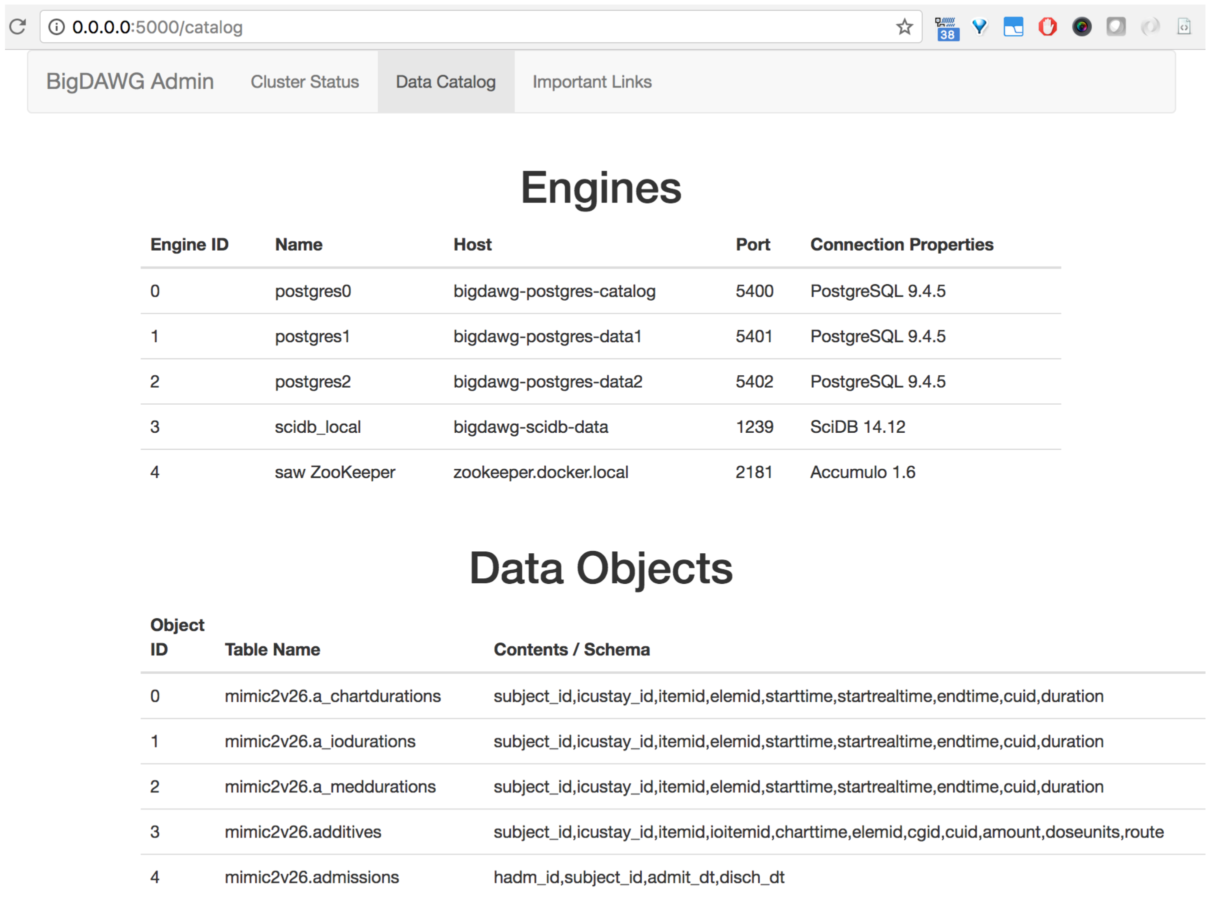
Name	Status		
bigdawg-accumulo-proxy	exited	Start	Stop
bigdawg-accumulo-master	exited	Start	Stop
bigdawg-accumulo-tserver0	exited	Start	Stop
bigdawg-accumulo-zookeeper	exited	Start	Stop
bigdawg-accumulo-namenode	exited	Start	Stop
bigdawg-scidb-data	exited	Start	Stop
bigdawg-postgres-data2	exited	Start	Stop
bigdawg-postgres-data1	exited	Start	Stop
bigdawg-postgres-catalog	exited	Start	Stop

Fig. 1.7: Container Status and Start/Stop Interface

Requirements:

You will need pip to install the python dependencies.

This interface has been tested with python versions 2.7.10, 2.7.11, and 3.5.2.



Engines

Engine ID	Name	Host	Port	Connection Properties
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

Data Objects

Object ID	Table Name	Contents / Schema
0	mimic2v26.a_chartdurations	subject_id,icustay_id,itemid,elemid,starttime,startrealtime,endtime,cuid,duration
1	mimic2v26.a_iodurations	subject_id,icustay_id,itemid,elemid,starttime,startrealtime,endtime,cuid,duration
2	mimic2v26.a_meddurations	subject_id,icustay_id,itemid,elemid,starttime,startrealtime,endtime,cuid,duration
3	mimic2v26.additives	subject_id,icustay_id,itemid,ioitemid,charttime,elemid,cgid,cuid,amount,doseunits,route
4	mimic2v26.admissions	hadm_id,subject_id,admit_dt,disch_dt

Fig. 1.8: Catalog Objects Interface

Installation instructions:

Note: If running on Mac or Windows, run the UI in a Docker Quickstart Terminal because Docker commands must be accessible by the Flask app.

Change directory to the "admin_ui" directory of the project root.

Install the python requirements with pip:

```
pip install -r requirements.txt
```

Edit the text file "catalog_config.txt" and configure the following credentials to connect to the Catalog database:

```
database=bigdawg_catalog
user=pguser
password=test
host=192.168.99.100
port=5400
```

Run the server with:

```
export FLASK_APP=app.py
flask run --host=0.0.0.0
```

The output will specify the local host and IP:

```
$ flask run --host=0.0.0.0
> * Serving Flask app "app"
> * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Navigate to the address shown above in a web browser and it will display the web interface.

See usage instructions in the administration section.

BigDAWG Middleware Internal Components

This section describes each Middleware component and their interaction in more technical detail. It is meant for contributors to BigDAWG or for adaptation of the Middleware to your own project or Polystore implementation.

The major components of the BigDAWG middleware are shown in the figure above. The sections below provide a technical description of each.

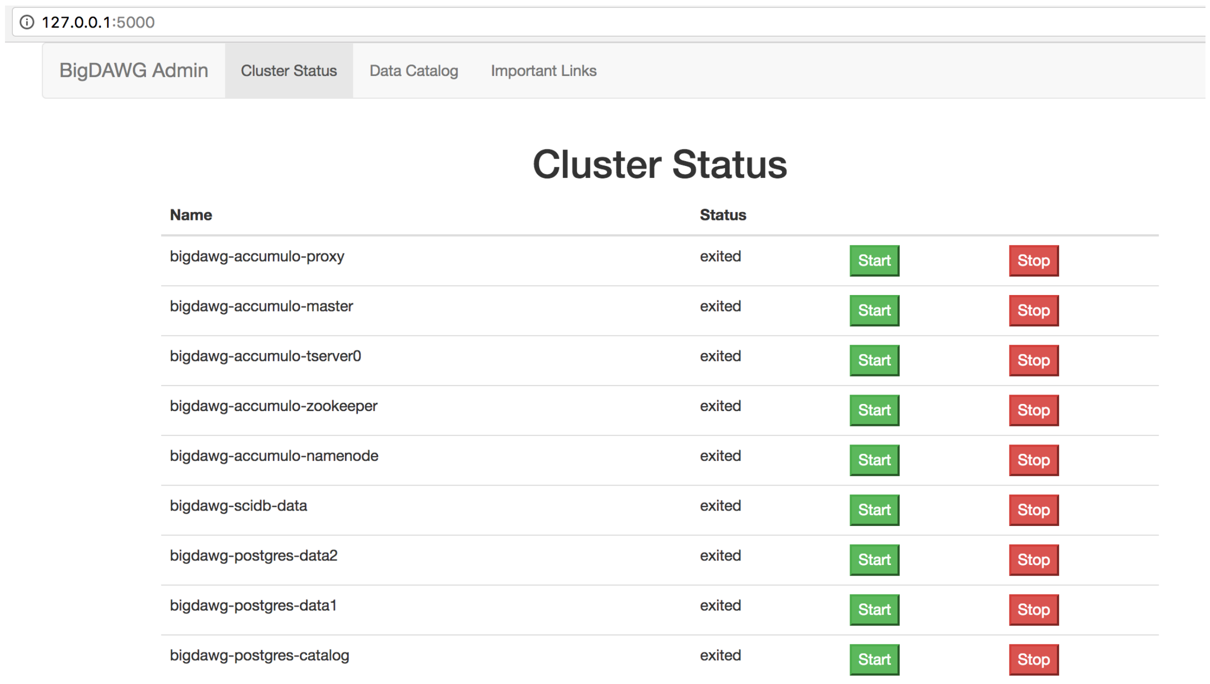
Query Endpoint

The Query Endpoint is responsible for accepting user queries, passing them to the Middleware, and responding with results.

The Query Endpoint is a simple HTTP server that's executed by the `istc.bigdawg.main()` method. The hostname/IP address and port used by this server is configurable by setting the following configuration properties:

```
grizzly.ipaddress=localhost
grizzly.port=8080
```

See the *Getting Started with BigDAWG* section or example queries that can be passed to the Query Endpoint. For more information on the syntax of query langage, refer to *BigDAWG Query Language*.



Cluster Status

Name	Status		
bigdawg-accumulo-proxy	exited	Start	Stop
bigdawg-accumulo-master	exited	Start	Stop
bigdawg-accumulo-tserver0	exited	Start	Stop
bigdawg-accumulo-zookeeper	exited	Start	Stop
bigdawg-accumulo-namenode	exited	Start	Stop
bigdawg-scidb-data	exited	Start	Stop
bigdawg-postgres-data2	exited	Start	Stop
bigdawg-postgres-data1	exited	Start	Stop
bigdawg-postgres-catalog	exited	Start	Stop

Fig. 1.9: Administrative Web Interface

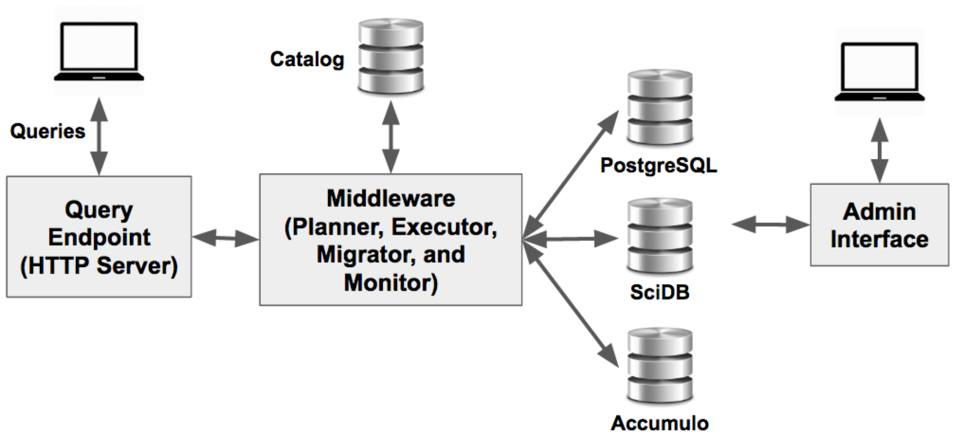

Queries
Catalog
Query Endpoint (HTTP Server)
Middleware (Planner, Executor, Migrator, and Monitor)
PostgreSQL
SciDB
Accumulo
Admin Interface

Fig. 1.10: System Overview

Middleware Components

The middleware has four components: the query planning module (planner), the performance monitoring module (monitor), the data migration module (migrator) and the query execution module (executor). Given an incoming query, the planner parses the query into collections of objects and creates a set of possible query plan trees that also highlights the possible engines for each collection of objects. The planner then 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 required, by the query plan. Some of the implementation details of each of these components are described below. Please refer to the publications section to learn more.

Catalog

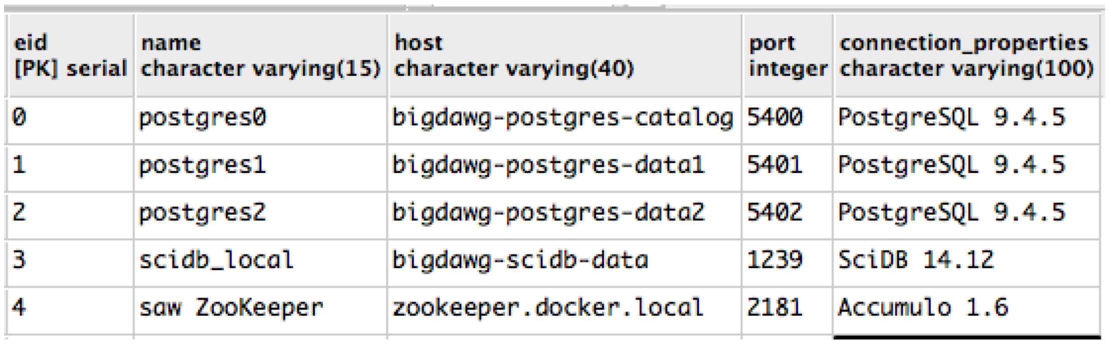
The Catalog is responsible for storing metadata about the polystore and its data objects. The Planner, Migrator, and Executor all rely on the Catalog for "awareness" of the BigDAWG's components, such as the hostname and IP address of each engine, Engine to Island assignments, and the data objects stored in each engine.

The Catalog is itself a PostgreSQL cluster with 2 databases: `bigdawg_catalog` and `bigdawg_schemas`.

`bigdawg_catalog` Database

This database contains the following tables.

- `engines` table: Engines currently managed by the Middleware, including engine name and connection information.



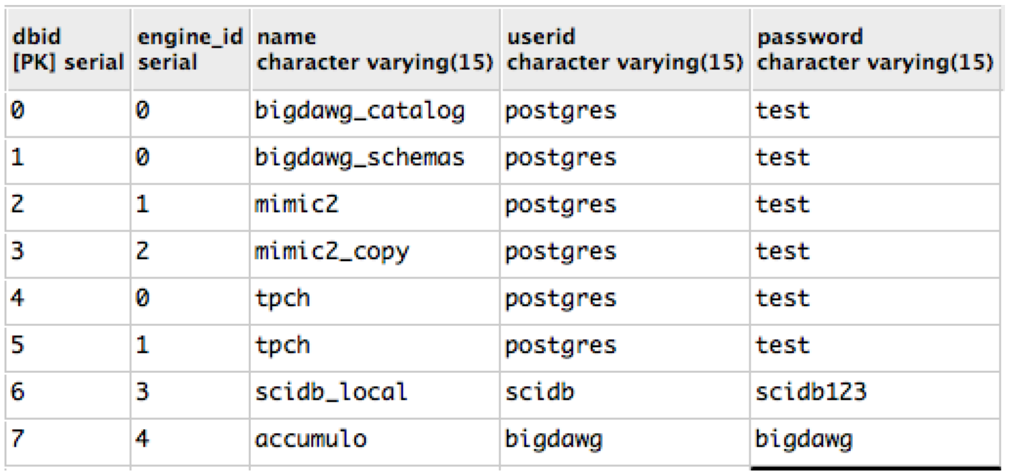
eid [PK] serial	name character varying(15)	host character varying(40)	port integer	connection_properties character varying(100)
0	postgres0	bigdawg-postgres-catalog	5400	PostgreSQL 9.4.5
1	postgres1	bigdawg-postgres-data1	5401	PostgreSQL 9.4.5
2	postgres2	bigdawg-postgres-data2	5402	PostgreSQL 9.4.5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

Fig. 1.11: Example Engines Table

- `databases` table: Databases currently managed by the Middleware, their corresponding engine membership, and connection authentication information.
- `objects` table: Data objects (*i.e.*, tables) currently managed by the Middleware, including fieldnames and object-to-database membership.
- `shims` table: Shims describing which engine is integrated into each island.
- `casts` table: information about what casts are available between each engine.

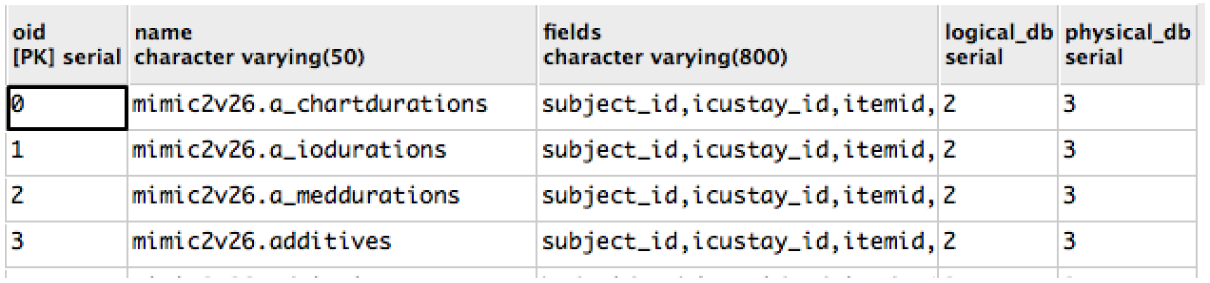
`bigdawg_schemas` Database

This database is made up of tables whose column schema define the schema of each data object. For example, the table `d_patients` from the MimicII dataset has the following schema in the `bigdawg_schemas` database.



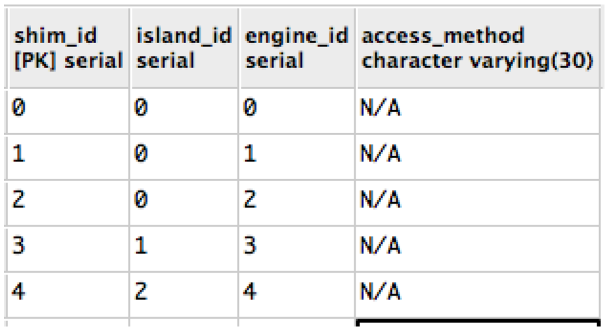
dbid [PK] serial	engine_id serial	name character varying(15)	userid character varying(15)	password character varying(15)
0	0	bigdawg_catalog	postgres	test
1	0	bigdawg_schemas	postgres	test
2	1	mimic2	postgres	test
3	2	mimic2_copy	postgres	test
4	0	tpch	postgres	test
5	1	tpch	postgres	test
6	3	scidb_local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

Fig. 1.12: 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

Fig. 1.13: 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

Fig. 1.14: Example Shims Table

```
CREATE TABLE mimic2v26.d_patients
(
  subject_id integer,
  sex character varying(1),
  dob timestamp without time zone,
  dod timestamp without time zone,
  hospital_expire_flg character varying(1)
)
```

Planner

This section details the Planner. The Planner coordinates all query execution. It has a single static function that initiates query processing for a given query and handles the result output.

```
package istc.bigdawg.planner;

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

The String `userinput` is the string of a BigDAWG query.

When the boolean of `isTrainingMode` is `true`, the Planner will perform query optimization by enumerating all possible orderings of execution steps that will produce an identical result. Then, the Planner sends the enumeration to the Monitor to gather query execution metrics. The Planner will then pick the fastest plan to run and return the result to the Query Endpoint. When `isTrainingMode` is `false`, the Planner will consult the Monitor to retrieve the best query plan based on past execution metrics.

The `processQuery()` function first checks if the query is intended to interact with the Catalog. If so, the query is routed to a specical processing module to parse and process these Catalog-related queries. Otherwise, `processQuery()` proceeds to parse and processing the query string.

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.

Migrator

The data migration module for the BigDAWG polystore system exposes a single convenient 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;

 /**
 * The main interface to the migrator module.
 */
```

```
public class Migrator {
  /**
   * General method (interface, also called facade) for other modules to
   * call the migration process.
   *
   * @param connectionFrom Information about the source
   * database (host, port, database name, user name,
   * user password) from which the data should be
   * extracted.
   *
   * @param objectFrom The name of the object
   * (e.g. table, array) which should be extracted
   * from the source database.
   *
   * @param connectionTo Information about the
   * destination database (host, port, database name,
   * user name, user password) to which the data
   * should be loaded.
   *
   * @param objectTo The name of the object
   * (e.g. table, array) which should be loaded to
   * the destination database.
   *
   * @param migrationParams Additional parameters for the migrator,
   * for example, the "create statement" (a statement to create an object:
   * table/array) which should be executed in the database
   * identified by connectionTo; data should be loaded to this new
   * object, the name of the target object in the create statement
   * has to be the same as the migrate method parameter: objectTo
   *
   * @return {@link MigrationResult} Information about
   * the results of the migration process (e.g. number of
   * extracted elements (rows, cells) from the destination database,
   * number of loaded elements (rows, cells) to the destination database,
   * the duration of the migration in milliseconds.
   *
   * @throws MigrationException Information why the migration failed (e.g. no access␣
↪to one
   * of the databases, schemas are not compatible, etc.).
   *
   */
    public static MigrationResult migrate(
      ConnectionInfo connectionFrom, String objectFrom,
      ConnectionInfo connectionTo, String objectTo,
      MigrationParams migrationParams)
         throws MigrationException;
    }
 }
```

An example of how the data migrator module can be called is presented below.

```
public class UseMigrator {
   public static void Main(String ... args) {
      logger.debug("Migrating data from PostgreSQL to PostgreSQL");
      FromDatabaseToDatabase migrator = new
          FromPostgresToPostgres();
      ConnectionInfo conInfoFrom = new
          PostgreSQLConnectionInfo("localhost", "5431",
          "mimic2", "pguser", "test");
      ConnectionInfo conInfoTo = new
          PostgreSQLConnectionInfo("localhost", "5430",
          "mimic2", "pguser", "test");
      MigrationResult result;
      try {
          result = migrator.migrate(conInfoFrom,
              "mimic2v26.d_patients",
          conInfoTo, "mimic2v26.d_patients");
      } catch (MigrationException e) {
          logger.error(e.getMessage());
      }
      logger.debug("Number of extracted rows: "
          + result.getCountExtractedElements()
          + " Number of loaded rows: " +
          result.getCountLoadedElements());
  }
}
```

Internally, the Migrator identifies the type of the databases by examinig the connection information. The `ConnectionInfo` object is merely an *interface* and we check what the real type of the object is. 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 work on distributed migrator (at present it works between instances of PostgreSQL) and tighter integration with S-Store as well as more efficient connection with Accumulo.

Binary migration

The data transformation module, which converts data be- tween different (mainly binary) formats, is the important part of the data migrator. This module is implemented in C/C++ to achieve high performance. The binary formats require operations at the level of bits and bytes. Many data formats apply encoding to values of attributes in order to decrease storage footprint.

To build the C++ migrator navigate to: `bigdawgmiddle/src/main/cmigrator/buil` in the maven project. We use CMake to build this part of the project. Simply execute:

```
cd bigdawgmiddle/src/main/cmigrator/build
cmake ..
make
```

Executor

The Executor executes intra-island queries through static functions. The static functions create instances of `PlanExecutor` objects that execute individual intra-island queries.

```
package istc.bigdawg.executor;

public class Executor {
    public static QueryResult executePlan(
        QueryExecutionPlan plan,
        Signature sig,
        int index
    ) 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` holds details of sub-queries that are required for their execution and a graph that provides 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.

```
package istc.bigdawg.executor;

class PlanExecutor {
    /**
     * Class responsible for handling the execution of a single QueryExecutionPlan
     *
     * @param plan
     *            a data structure of the queries to be run and their ordering,
     *            with edges pointing to dependencies
     */
    public PlanExecutor(
        QueryExecutionPlan plan
    )
}
```

Monitor

The BigDAWG monitor is responsible for managing queries.

```
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 is provided to identify a query.

The `addBenchmarks` method adds a new benchmark. If the `lean` parameter is `false`, the benchmark is immediately run over all of its possible query execution plans (henceforth referred to as QEP).

The `getBenchmarkPerformance` method returns a list of execution times for a particular benchmark, ordered in same order that the benchmark's QEPs are received.

The best way to use the module is to add all of the relevant benchmarks first 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. In this way, a user can add many benchmarks that are believed to cover the majority of query use cases. Then you use the `getClosestSignature` method to find a matching benchmark and compare the QEP times to your current signature's QEPs. On missing any matching signatures, you can add the current signature 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.

BigDAWG Query Language

Fundamentally, BigDAWG is middleware that provides a common application programming interface to a collection of distinct storage engines. To the typical user, BigDAWG is viewed as a query engine for the polystore system; hence, understanding how these queries are written is key to understanding BigDAWG.

BigDAWG queries are written with the BigDAWG Query language which uses a functional syntax:

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 indicate the islands targeted by a query:

- 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 either SQL or D4M.

The remaining function tokens deal with metadata for the polystore system and the migration of data between islands:

- bdcatalog – the query targets the BigDAWG catalog using SQL.
- bdcast – the query is a cast operation for inter-island data migration.

Queries using the 'bdcast' function token behave differently than queries based on the other function tokens. A 'bdcast' query is always nested inside other queries to indicate migration of data between islands.

In the next few subsections, we summarize operations supported by each island and provide a formal definition of the BigDAWG query syntax. See *Example Queries* for examples of BigDAWG queries.

BigDAWG Syntax Definitions

BigDAWG Query

BigDAWG Query Syntax:

```
BIGDAWG_SYNTAX ::=
  BIGDAWG_RETRIEVAL_SYNTAX | CATALOG_QUERY
```

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

Catalog Manipulation

Catalog manipulation queries are used to directly view the content of the catalog.

You may find the list of `catalog_table_name` in *Catalog*.

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

Inter-Island Cast

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.

Cast Syntax:

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

Relational Island

The Relational Island follows the relational data model, where data is 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 you to issue single-layered `SELECT` query with filter, aggregation, sort and limit operations.

Relational Island supports the following data types: integer, varchar, timestamp, double, float

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 ]
```

```
SQL_EXPRESSION ::=
  SQL_NON_AGGREGATE_EXPRESSION
  | SQL_AGGREGATE
```

```
SQL_NON_AGGREGATE_EXPRESSION ::=
  literal
  | column_name
  | { SQL_NON_AGGREGATE_EXPRESSION SQL_BINARY_ALGEBRAIC_FUNCTION  SQL_NON_AGGREGATE_
↪EXPRESSION }
```

```
  | { - SQL_EXPRESSION }
  | {( SQL_EXPRESSION )}
  | SQL_CONDITION
```

```
SQL_BINARY_ALGEBRAIC_FUNCTION ::=
  + | - | * | / | %
```

```
SQL_CONDITION ::=
  { SQL_NON_AGGREGATE_EXPRESSION SQL_CONDITION_OPERATOR
       SQL_NON_AGGREGATE_EXPRESSION }
  | { SQL_NON_AGGREGATE_EXPRESSION SQL_BINARY_LOGICAL_OPERATOR
       SQL_NON_AGGREGATE_EXPRESSION }
```

```
SQL_CONDITION_OPERATOR ::=
  = | < | > | <= | >= | !=
```

```
SQL_BINARY_LOGICAL_OPERATOR ::=
  AND
```

```
SQL_AGGREGATE ::=
  { SQL_AGGREGATE_NAME( [ DISTINCT ] SQL_NON_AGGREGATE_EXPRESSION [ , ... ] ) }
  | { count( { * | SQL_NON_AGGREGATE_EXPRESSION } )}
  | { width_bucket( SQL_NON_AGGREGATE_EXPRESSION, double_precision_number,
      double_precision_number, integer ) }
```

```
SQL_AGGREGATE_NAME ::=
  sum | avg | min | max
```

```
POSTGRES_SCHEMA_DEFINITION ::=
  ( { column_name sql_data_type POSTGRES_COLUMN_CONSTRAINT } [, ...] )
```

```
POSTGRES_COLUMN_CONSTRAINT ::=
  { [ PRIMARY KEY ]
    | [ REFERENCES table_name [( column_of_table_referenced )] ] }
  [ [ NOT ] NULL ]
```

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 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. Array Island also allows attribute sorting; however, at the moment, only sort in ascending order is supported.

Array Island supports the following data Types: string, int64, datetime, double, float

Array Island Syntax:

```
ARRAY_ISLAND_QUERY ::=
  bdarray( ARRAY_SYNTAX )
```

```
ARRAY_SYNTAX ::=
  { scan( array_name ) }
  | { 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
```

```
SCIDB_EXPRESSION ::=
  SCIDB_AGGREGATE_CALL
  | SCIDB_NON_AGGREGATE_EXPRESSION
```

```
SCIDB_BINARY_ALGEBRAIC_FUNCTION ::=
  + | - | * | / | %
```

```
SCIDB_CONDITION ::=
  { SCIDB_NON_AGGREGATE_EXPRESSION SCIDB_CONDITION_OPERATOR SCIDB_NON_AGGREGATE_
↪EXPRESSION }
  | { SCIDB_NON_AGGREGATE_EXPRESSION SCIDB_BINARY_LOGICAL_OPERATOR SCIDB_NON_
↪AGGREGATE_EXPRESSION }
  | { regex( { attribute_name | dimension_name }, 'regex_expression') }
  | { iif ( SCIDB_BINARY_PREDICATE, SCIDB_ALGEBRAIC_EXPRESSION, SCIDB_ALGEBRAIC_
↪EXPRESSION) }
```

```
SCIDB_NON_AGGREGATE_EXPRESSION ::=
  literal
  | dimension
  | attribute
  | { SCIDB_NON_AGGREGATE_EXPRESSION SCIDB_BINARY_ALGEBRAIC_FUNCTION SCIDB_NON_
↪AGGREGATE_EXPRESSION }
  | { - SCIDB_EXPRESSION }
  | {( SCIDB_EXPRESSION )}
  | SCIDB_CONDITION
```

```
SCIDB_CONDITION_OPERATOR ::=
  = | < | > | <= | >= | !=
```

```
SCIDB_BINARY_LOGICAL_OPERATOR ::=
  AND
```

```
SCIDB_AGGREGATE_CALL ::=
  SCIDB_AGGREGATE_FUNCTION( dimension )
```

```
SCIDB_AGGREGATE_FUNCTION ::=
  sum | avg | min | max
```

```
SCIDB_SCHEMA_DEFINITION ::=
  <{attribute_name: data_type} {, ...}>
  \[ { dimension_name = { integer_lower_bound | * } : { integer_upper_bound | * } ,␣
↪integer_cell_size, integer_overlap} [, ...] \];
```

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.

Text Island supports the following data Types: string

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 }] }
```

```
TEXT_ISLAND_RANGE ::=
  TEXT_ISLAND_RANGE_START_KEY
  | TEXT_ISLAND_RANGE_END_KEY
  | (TEXT_ISLAND_RANGE_START_KEY, TEXT_ISLAND_RANGE_END_KEY )
```

```
TEXT_ISLAND_RANGE_START_KEY ::=
  'start' : \['start_row','[start_column_family]','[start_column_qualifier]'\]
```

```
TEXT_ISLAND_RANGE_END_KEY ::=
  'end' : \['end_row','[end_column_family]','[end_column_qualifier]'\]
```

```
TEXT_OPERATOR ::=
  scan
```

```
TEXT_SCHEMA_DEFINITION ::=
  ()
```

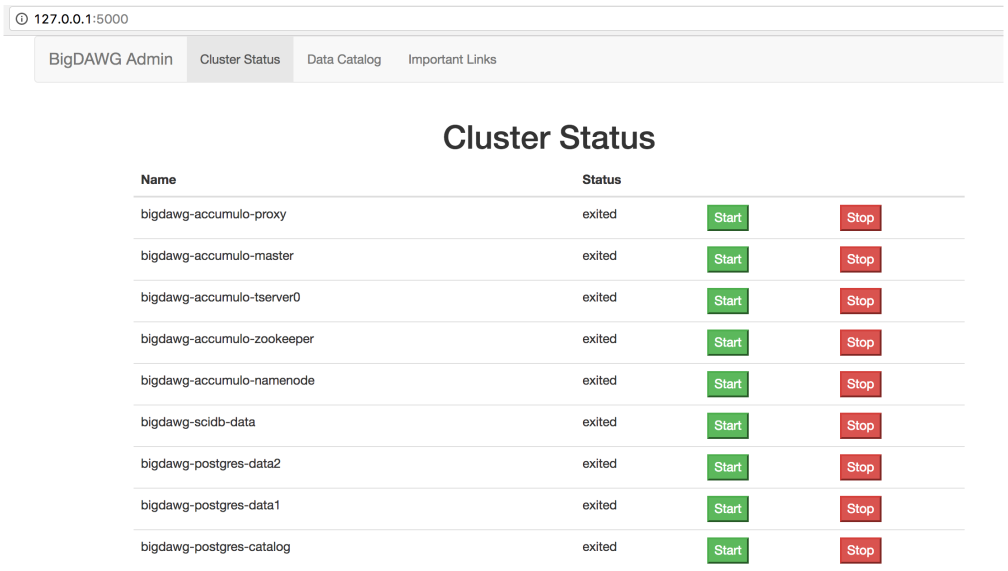
This section provides some tips on how you can adapt the BigDAWG system for your own data. Specifically, we describe how to use the administrative web interface, add your own database engine, add your own tables/databases and tips on how to construct your own island. Some of these may require some level of expertise so please do not hesitate to contact us if you have any questions!

Personalizing the setup

Administrative Web Interface:

A very basic administrative web interface is included with this release, which will let you see the status of the BigDAWG cluster of databases, start and stop containers, and view the Catalog objects table.

You can view a video demonstration here



Cluster Status

Name	Status		
bigdawg-accumulo-proxy	exited	Start	Stop
bigdawg-accumulo-master	exited	Start	Stop
bigdawg-accumulo-tserver0	exited	Start	Stop
bigdawg-accumulo-zookeeper	exited	Start	Stop
bigdawg-accumulo-namenode	exited	Start	Stop
bigdawg-scidb-data	exited	Start	Stop
bigdawg-postgres-data2	exited	Start	Stop
bigdawg-postgres-data1	exited	Start	Stop
bigdawg-postgres-catalog	exited	Start	Stop

Fig. 1.15: Container Status and Start/Stop Interface

Formulating Example Queries:

todo: (Add information about writing other queries)

Adding your own data:

You can register a new database with a BigDAWG cluster by adding information about the database to the Catalog. Once the Catalog is updated, the Middleware is aware of the new database and can perform all island-compatible queries on it.

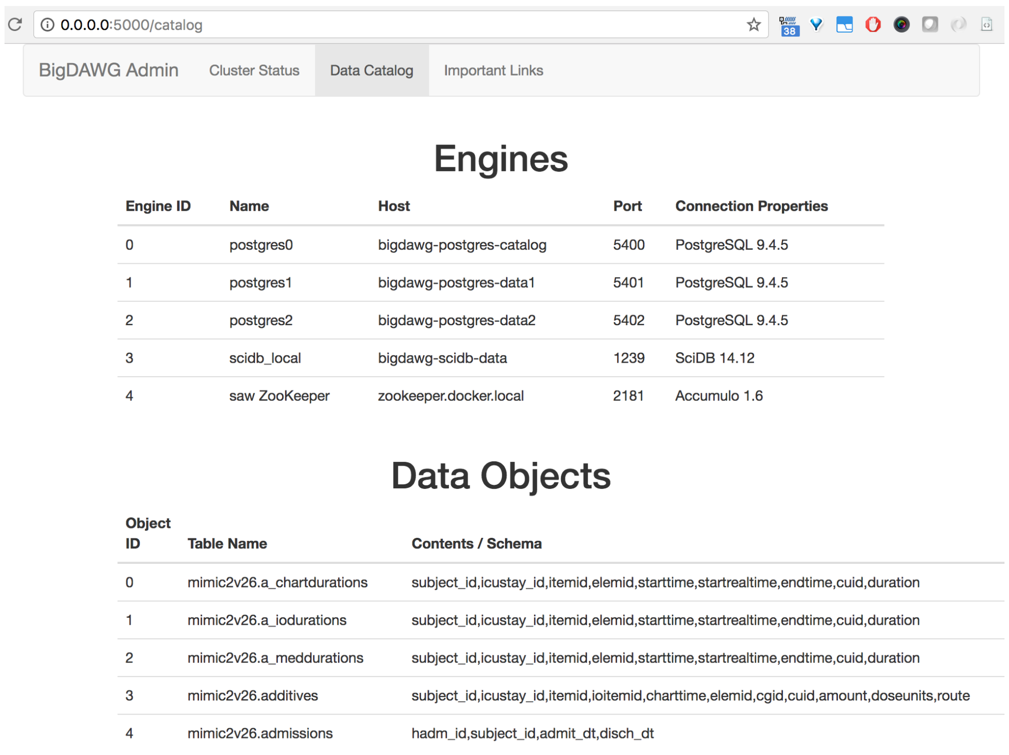
For example, assume that you can add a simple relational database named `inventory` with a table named `products` consisting of product information such as the following:

ItemNumber	ItemName	Price
1	Banana	0.99
2	Apple	1.25
3	Carrot	1.30

There are 3 parts of the Catalog that must be updated. Recall that the Catalog itself is a Postgres database named `bigdawg_catalog`.

1.) The `databases` table requires the following fields:

- `dbid`: serial integer for referring to the database by ID
- `engine_id`: serial integer for referring to the type of engine that this database corresponds to. This ID should be read from the `eid` value of the
- `engines` table in the Catalog.
- `name`: name of the database. In this example, this value would be "inventory".



Engines

Engine ID	Name	Host	Port	Connection Properties
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

Data Objects

Object ID	Table Name	Contents / Schema
0	mimic2v26.a_chartdurations	subject_id,icustay_id,itemid,elemid,starttime,startrealtime,endtime,cuid,duration
1	mimic2v26.a_iodurations	subject_id,icustay_id,itemid,elemid,starttime,startrealtime,endtime,cuid,duration
2	mimic2v26.a_meddurations	subject_id,icustay_id,itemid,elemid,starttime,startrealtime,endtime,cuid,duration
3	mimic2v26.additives	subject_id,icustay_id,itemid,ioitemid,charttime,elemid,cgid,cuid,amount,doseunits,route
4	mimic2v26.admissions	hadm_id,subject_id,admit_dt,disch_dt

Fig. 1.16: Catalog Objects Interface

- `userid`: the username used to log into the new database
- `password`: the password used to log into the new database

For example, an `INSERT` statement would look like this:

```
INSERT INTO catalog.databases values(8, 0, inventory, postgres, test);
```

2.) The `objects` table requires the following fields:

- `oid`: serial integer for referring to the new table.
- `name`: name for the new data object. In this example, the value would be "products"
- `fields`: A comma-separated string of column names in the `products` table
- `logical_db`: An ID referencing the database ID from the `databases` table
- `physical_db`: An ID referencing the database ID from the `databases` table

For example, an `INSERT` statement would look like this:

```
INSERT INTO catalog.objects values(52, products, ItemNumber,ItemName,Price, 8, 8);
```

3.) `bigdawg_schemas` table:

```
CREATE TABLE products (ItemNumber integer, ItemName varchar(40), Price real);
```

Adding your own engine

This guide provides you a starting point to integrate a database with JDBC driver into the BigDAWG middleware. For other types of databases, please reach out to us and we will work with you.

1. Find the associated JDBC driver, and add it as a dependency to pom.xml
2. Create the associated `ConnectionInfo`, `DBHandler`, `DBInstance`, etc. classes for the database engine. (See Postgresql package for reference.)
3. Create a new query generator if existing ones are not fully compatible. Also might need some sort of utility class to convert datatype names to some common representation (e.g. Postgresql datatypes – see)
4. Modify `islands.TheObjectThatResolvesAllTheDifferencesAmongTheIslands.java -` EngineEnum, getQConnectionInfo(), getQueryGenerator(), and anywhere else that would be appropriate.
5. Create Export and Load classes for the Database engine (under migration)
6. Create migrators to/from Postgres (or any other engines you want to migrate to/from)
7. Register the new migrators in Migrator.java
8. When setting up your BigDAWG instance, make sure to add an entry to the catalog to let it know your database engine exists. Also add entries for the schemas for tables stored on that index.

Connecting to existing databases

Use can use the middleware distributed in this release to connect to an existing database. For this example, we assume that you have an existing Postgres instance that you would like to connect to. Let's assume that the database name if `foo` and that this database has two tables `foo_table1` and `foo_table2`.

1. Clone the git repository to a system that can connect to the Postgres database (from https://github.com/bigdawg-istc/bigdawg):
2. In the Postgres database, create two new databases: 1) `bigdawg_catalog` with schema `catalog` and 2) `bigdawg_schemas`. The `bigdawg_catalog` database contains a variety of information such as connection properties, names of tables and schema. Look at `/provisions/cluster_setup/postgres-catalog/bdsetup/catalog_inserts.sql` for an example of what tables are filled for connecting to the various MIMIC II tables. You will need to add the engine connection information in catalog. In this case, you will add a row to catalog.engines for the existing Postgres database; entries in catalog.databases for the `bigdawg_catalog`, `bigdawg_schemas`, and `foo` databases. You will also need to add information about the tables `foo_table1` and `foo_table2` to the catalog.objects table.

3) In the `bigdawg_schemas` database, create empty schemas for the `foo` database similar to what we did for the MIMIC II database: `./provisions/cluster_setup/postgres-catalog/bdsetup/mimic2_schemas_ddl.sql`

4. Now, you can compile the code you downloaded.

First, you need to edit file `profiles/dev/dev-config.properties` so that the middleware knows where to look for the Postgres engine. Specifically look at the following lines to modify:

```
# ==================
# Catalog database
# ==================

postgresql.url=jdbc:postgresql://host:port
postgresql.user=XXXXXXXXX
postgresql.password=XXXXXXXXX
```

Once you are done editing this file, close and save it and you are ready to package the JAR in the root directory using the following command:

```
mvn package -P mit -DskipTests -f pom.xml -q
```

3. Now that you have packaged the jar, you should be ready to execute it using the following command:

```
mvn exec:java -f pom.xml -P mit -q
```

The above command will start the bigdawg instance on the current node you are running on.

4. If you are running the Postgres engine on another host, you need to launch the middleware on that host as well. For example, you can ssh into that node and use the same command as above to run it.

```
ssh node
mvn exec:java -f pom.xml -P mit -q
```

5. Now, you should be ready to issue a query

```
curl -X POST -d "bdrel(select * from foo.table1);" http://localhost:8080/bigdawg/query
```

Adding your own island

This guide provides a road-map for adding new islands to the BigDAWG system. Creating an island involves four general steps: determine the language and functionalities supported by the island, implement supports for the island language and logical representations of the functionalities in the BigDAWG context, creating shims between the island and the database engines, and create a front-end support for other BigDAWG components. We will elaborate on these steps using the current Text Island as an example.

1. Determining the language and functionalities

   We model our island on the functionalities of Apache Accumulo. It is therefore by design to support only complete or ranged table scans. Therefore, we need to only support one operation: Scan, with optional range parameters. Consequently, there will not be nested expressions. As with other islands, we will not reformat the results.

2. Implement supports for the language and its functionalities in the BigDAWG context

   For query optimization purposes, functionalities of an island are represented by implementation of Java interface *Operator* and its extensions, such as *SeqScan*, or sequential scan. In our case, we want to implement a Text Island operator that scans a table, with optional specification of ranges. Therefore, we want to create the class *TextScan* that implements *SeqScan* interface.

   Note that to retain extendibility for the Text Island, we first created a parent abstract class named *TextOperator* that implements the *Operator* interface; we extended the *TextOperator* class to create our *TextScan*.

   Language support entails parsing user query into an Abstract Syntax Tree (AST) with *Operator* nodes. In our case, each query will consist of a single *TextScan* and there will not be branches.

   We therefore use the JSON to implement our language. In a JSON object, we require the user to provide a field of table name and an optional JSON object to specify range in the query. We use the *org.json.simple.parser.JSONParser* in our language parser to create *TextScan* operators.

3. Creating shims for BigDAWG Query Executor

   At the moment, we only want to connect Accumulo to the Text Island. Therefore we implement the *Shim* Java interface to create our shim, *TextToAccumulo* shim. The virtual functions listed in *Shim* provides a very good guideline of what needs to be done to connect Accumulo to the Text Island.

4. Creating planner and executor facing front

   We begin by creating the *TextIsland* interface used by the Planner and Executor. The *TextIsland* class implements Java interface *Island*. In the *TextIsland* class, we need to define the default database to which an inter-island intermediate result could be migrated. This is done by looking up the database's *dbid* in the Catalog. The setup and tear down virtual functions are intended for creating and destroying temporary tables used for inter-island query execution. The virtual function for creating Literal and Constant Signature asks for a list of constants, therefore we return a list of values used in the range specification.

   We then implement *IntraIslandQuery* Java interface to create the logical intra-island execution plan of the Text Island. Here, we make use of the setup and tear down functions created in TextIsland to create support for new tables migrated from another island.

   In other islands, an operator such as a Join could take multiple table inputs. The intra-island execution plan needs to create 'cut points' in the AST to divide the AST into containers – sub-queries using naturally co-located tables – and a remainder – a skeleton AST that executes with migrated intermediate results. The *traverse* virtual function is designated to recursively mark natural locations of a table or sub-query and create containers out of any sub-query whose children are not co-located. *pruneChild* is used to mark a node in an AST so that a sub-query starting from the node is used to create a container. It is hinted that a *remainderLoc* with a positive value indicate all input tables co-locate and no containers are constructed; a zero value indicate that at least two containers exist.

   getQEPs function lists all viable Query Execution Plans (QEPs) composed from permutations of the query. A permutation produces the same result as does the original query, yet it has a different order for Joins. The different permutations are run used by the monitor, which then records performance information with regard to each permutation. *getQEP* (without s) is used to extract a specific QEP.

   At last, we modify *IslandAndCastResolver* to finish the integration, and add new entries to the BigDAWG Catalog to make them usable.

Selected BigDAWG Publications

Overall architecture:

"The BigDAWG Polystore System and Architecture", Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Tim Mattson, Michael Stonebraker. IEEE High Performance Extreme Computing, 2016.

   BigDAWG overall architecture and details of various middleware components along with some performance results.

"The Big Dawg Polystore System", J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, S. Zdoânik. ACM Sigmod Record, 44(3), 2015.

   Original vision paper on BigDAWG architecture.

BigDAWG applications:

"Demonstrating the BigDAWG Polystore System for Ocean Metagenomic Analysis", Tim Mattson, Vijay Gadepally, Zuohao She, Adam Dziedzic, Jeff Parkhurst CIDR'17 Chaminade, CA, USA

   This paper describes a second application based on BigDAWG; an oceanography dataset including integration with the S-Store system for streaming data.

"A Demonstration of the BigDawg Polystore System", A. Elmore, J. Duggan, M. Stonebraker, 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, S. Zdonik. Proceedings of VLDB, 2015.

This paper describes our performance measurements with the MIMCII dataset.

BigDAWG Middleware:

"The BigDAWG Monitoring Framework", Peinan Chen, Vijay GAdepally, Michael Stonebraker IEEE High Performance Extreme Computing, 2016.

This paper describes the BigDAWG monitoring framework.

"BigDAWG Polystore Query Optimization Through Semantic Equivalences", Zuohao She, Surabhi Ravishankar, Jennie Duggan IEEE High Performance Extreme Computing, 2016.

This paper describes query optimization in BigDAWG.

"Cross-Engine Query Execution in Federated Database Systems", Ankush M. Gupta, Vijay Gadepally, Michael Stonebraker (MIT) IEEE High Performance Extreme Computing, 2016.

This paper describes how queries are split between different islands.

"Data Transformation and Migration in Polystores", Adam Dziedzic, Aaron J. Elmore, Michael Stonebraker

This paper describes how the casts work in BigDAWG.

"Integrating Real-Time and Batch Processing in a Polystore", John Meehan, Stan Zdonik Shaobo Tian, Yulong Tian, Nesime Tatbul, Adam Dziedzic, Aaron Elmor

This paper provides details behind the integration of the S-Store streaming system with BigDAWG.

Contributors

Acknowledgement

The development of BigDAWG was supported by the Intel Science and Technology Center (ISTC) for Big Data. The authors also wish to thank collaborators as Brown University, University of Washington, Portland State University, Universtiy of Tennessee and Intel for their collaboration.

Contributors

There are a number of people involved in developing the current version of the codebase:

Adam Dziezdzic

Aaron Elmore

Vijay Gadepally

Jeremy Kepner

Kyle O'Brien

Sam Madden

Timothy Mattson

Jennie Rogers

Mike Stonebraker

Zuohao She

Alumni/Collaborators

We are fortunate to have a number of collaborators who have helped us along the way:

Magdalena Balazinska, University of Washington

Leilani Battle, MIT CSAIL

Ugur Cetintemel, Brown University

Peinan Chen, MIT CSAIL

Ankush Gupta, MIT CSAIL

Brandon Haynes, University of Washington

Jeffrey Heer, University of Washington

Bill Howe, University of Washington

Tim Kraska, Brown University

David Maier, Portland State University

Stavros Papadopoulos, Intel

Jeff Parkhurst, Intel

Surabhi Ravishankar, Northwestern University

Ran Tan, North Carolina State University

Nesime Tatbul, Intel and MIT

Kristin Tufte, Portland State University

Manasi Vartak, MIT CSAIL

Katherine Yu, MIT CSAIL

Stan Zdonik, Brown University

Frequently Asked Questions

1.) What is BigDAWG?:

BigDAWG (short for Big Data Working Group) is a reference implementation of a Polystore database. Essentially, BigDAWG provides the middleware needed to talk to multiple disparate engines (for example, SQL, NoSQL and NewSQL engines) while using multiple data model and programming languages (for example, SQL, AFL, AQL). More details about BigDAWG can be found in our publications.

2.) Where do I download get started and download everything I need to see what you have released?:

See *Getting Started with BigDAWG* for details.

3.) How do I modify the queries or make my own?:

See Section *BigDAWG Query Language*

4.) How do I add my own engine?:

See Section *BigDAWG Query Language* and contact us for help! Perhaps someone has already integrated (or is in the process of integrating) the engine of interest.

5.) How do I add my own data/tables?:

We've distributed a handy Python script that can help you load data. See administration for details on how to use this. If you have more questions, of course, email us!

6.) What is the query API?:

Section *BigDAWG Query Language* addresses the query language we use.

7.) How do I create a new island?:

Looks at Section administration for insight on how to do this. Please feel free to reach out to us if you have any other questions or comments.

8.) How do I contact the development team with bugs, questions, etc.?

Email us at `bigdawg-help@mit.edu`

9.) How is BigDAWG licensed?

The BigDAWG middleware is licensed under the terms of the BSD 3-clause license. Please note that external componenents such as database management engines may have their own license agreements. Please reach out to us for specific licensing questions.