Demonstrating the BigDAWG Polystore System for Ocean Metagenomic Analysis

Tim Mattson
Intel Corp.
2111 Northeast 25th Ave.
Hillsboro, Oregon
timothy.g.mattson@intel.com

Vijay Gadepally
MIT
244 Wood Street
Lexington, MA 02420
vijayg@ll.mit.edu

Zuohao She
Northwestern University
2145 Sheridan Rd.
Evanston, IL 60208
zuohaoshe2013@u.northwestern.edu

Adam Dziedzic
University of Chicago
5801 S. Ellis Ave.
Chicago, IL 60637
ady@uchicago.edu

Jeff Parkhurst
Intel Corp.
1900 Prairie City Rd.
Folsom, CA 95630
jeff.parkhurst@intel.com

ABSTRACT

In most Big Data applications, the data is heterogeneous. As we have been arguing in a series of papers, storage engines should be well suited to the data they hold. Therefore, a system supporting Big Data applications should be able to expose multiple storage engines through a single interface. We call such systems, *polystore systems*. Our reference implementation of the polystore concept is called BigDAWG (short for the Big Data Analytics Working Group). In this demonstration, we will show the BigDAWG system and a number of polystore applications built to help ocean metagenomics researchers handle their heterogenous Big Data.

CCS Concepts

•Information systems → DBMS engine architectures; Federated databases; Data federation tools; •Human-centered computing → *Visualization toolkits;*

Keywords

Polystore Data management systems; Data Integration

1. INTRODUCTION

Real applications, as opposed to simplified problems used for benchmarking, are complex. Data often spans multiple storage engines with queries over diverse data models. Common data integration techniques based on replicating data or forcing all data to fit into a single storage engine are not an effective long term solution.

We believe problems based on complex heterogeneous data are best addressed with *p*olystore architectures; integrated systems composed of multiple storage engines with queries that span multiple data models. We have been working on a reference implementation of a polystore architecture called BigDAWG [11] (short for the Big Data Analytics Working Group).

In this paper, we describe BigDAWG and its use for Big Data problems. Past work with BigDAWG focussed on medical informatics with the MIMIC-II [23] data set. This paper introduces a new problem for the BigDAWG project; an oceanographic data set used for metagenomic analysis. An additional contribution of this paper is a discussion of the BigDAWG middleware components and how they map queries and cast data between storage engines.

2. DATA SET: OCEAN METAGENOMICS

In our previous studies [14], we applied polystore systems to medical data and achieved significant speedups over traditional approaches based on a single storage engine [16]. In this paper, we consider a new problem for BigDAWG in collaboration with the Chisholm lab at MIT.

The Chisholm Lab (*https://chisholmlab.mit.edu/*) specializes in microbial oceanography and systems biology. Working with the GEOTRACES (*http://www.geotraces.org/*) consortium, they collect water samples during cruises across the globe and then release the marine chemical and hydrographic data [18] to the research community. The Chisholm Lab also archives frozen seawater samples from these cruises, and sequences these samples to understand the relationship between the diversity of marine cyanobacteria such as Prochlorococcus [5] and environmental variables. They work with large (many TB), diverse datasets that contain components:

- *G*enome Sequences: Paired-end sequence data which consists of two files (one from the beginning of the sequence and one from the end) with an average of 20 million unique sequences per sample.
- *S*ensor Metadata: Data from the GEOTRACES consortium with information about each sample and the sensors used for data collection.

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2017.
CIDR'17 Chaminade, CA, USA

- *S*ample Metadata: Data from the GEOTRACES consortium with information about hydrographic and chemical measurements of individual samples such as concentration of different metals in the water and information about the physical properties of the seawater (e.g., macro-nutrient concentration, temperature).
- *C*ruise Reports: Free-form text reports written by researchers on-board each cruise.
- *S*treaming Data: Data from the SeaFlow underway flow cytometer system [25]: a system to continuously measure the abundance and composition of microbial populations in near real-time.

Chisholm Lab researchers are interested in relationships between communities of cyanobacteria and environmental parameters (e.g. light, temperature and the chemical composition of the seawater). These relationships are contained in the data, but researchers often struggle with the management of such complex scientific datasets. Often, they build one-off solutions or solutions that do not scale with volume or the addition of new data products.

This ocean metagenomics dataset is an excellent candidate for a polystore solution. A single database management system (DBMS) or data model (relational, graph, array, document, key-value, etc.) will not be able to provide efficient access to all parts of the dataset. For example, the free-form text notes would work well with a key-value store while the structured metadata would benefit from a relational engine. Furthermore, the type of query may favor one store over another. For example, an engine such as SciDB [6] is an efficient choice for analytics that map onto linear algebra operations, but key-value stores such as Accumulo [15] provide high performance for genomics problems [10].

In this demo, we test our hypothesis that a polystore system is well suited for such ocean metagenomics problems. For this single data set, we leverage a number of DBMS engines. Structured data is stored in PostgreSQL, streaming data is directed to S-Store, text reports stored in Apache Accumulo, and genetic sequences stored in SciDB.

3. THE BigDAWG POLYSTORE SYSTEM

In Figure 1, we show the major components of the BigDAWG polystore system. This design adheres to the fundamental tenets of the project:

1. There is no single data model for constructing queries.
2. The complete functionality of each underlying storage engine is fully exposed.

Islands provide users with a number of programming and data model choices; *Shims* connect islands to databases; and *Cast* operations support migration of data from one engine or island to another. A more thorough description of individual components of the architecture can be found in [8, 16, 19, 14].

3.1 BigDAWG Components

3.1.1 Islands

Islands are a core concept in the BigDAWG polystore system. An island consists of a query language and a data model. Each island is associated with one or more storage engines and may connect to multiple instances of the same engine. Some islands use the language and the data model adopted by a specific database engine; we call these *degenerate islands*. Other islands do not conform to the language layout or data model of a specific engine; we call these *virtual islands*. Within each island, BigDAWG provides location transparency for all data sets which are registered to the BigDAWG middleware. The middleware maintains a global catalog that maps engines and their corresponding data objects to databases and islands.

A virtual island is typically the *intersection* of an island's constituent database engines: its query language supports common semantics found in the languages of all associated engines, and its data model implements features shared by the data models adopted by those engines. The virtual island is a unique construct proposed by the BigDAWG architecture; therefore, our development effort has focused on virtual islands. We currently support *relational* and *array* virtual islands.

In our current implementation, relational and array islands each implement a set of *BigDAWG operators*. The operators themselves represent logical semantic symbols found in database models such as the relational model governed by relational algebra. The implementation of these operators take the form of *Operator nodes*; data structures that, in addition to representing semantic symbols, hold information for generating executable queries. Operator nodes are organized in tree structures to represent queries issued to the island. We will denote these tree structures as Abstract Syntax Trees (ASTs). A sub-query of the original query therefore corresponds to a sub-tree of the AST. BigDAWG can swiftly manipulate the execution order and data movement of intermediate results of a complex query by reshaping the corresponding AST. This feature reduces the overhead costs associated with polystore query optimization.

While virtual islands handle intersecting semantics across multiple types of engines, *degenerate islands* cater to the semantics of a single engine. The degenerate island allows users to directly issue queries to an underlying engine, such as SciDB. Users compose queries to the engine in the native query language of the data model and can use operators uniquely available to that language. The BigDAWG middleware supports degenerate islands for each of the four engines supported: PostgreSQL, SciDB, S-Store [7], and Accumulo. Since PostgreSQL and SciDB use operator-based languages, their corresponding degenerate islands closely resemble relational and array virtual islands.

3.1.2 Supported engines

The BigDAWG middleware currently supports PostgreSQL, SciDB, S-Store and Accumulo. They differ from each other in terms of their language, data model and user facing interface. PostgreSQL is a relational database system used mainly for traditional relational queries. SciDB is an array database system optimized for data sets with a multidimensional array layout and is well-suited for queries that involve linear algebra operations. We provide partial support for SciDB's proprietary Array Functional Language (AFL). Accumulo is a distributed, persistent, multidimensional sorted map, key-value store that can be queried with the D4M language [17, 20]. S-Store is a stream engine that runs on a relational data model and can be queried with pre-

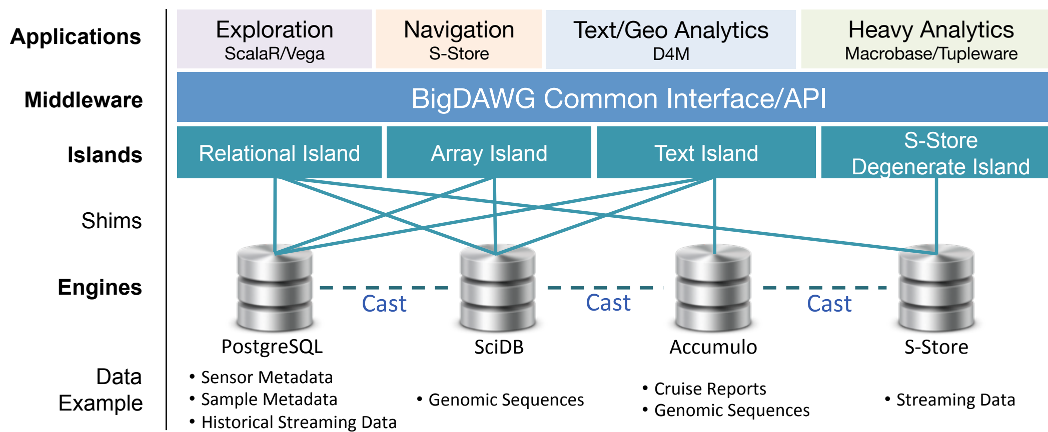

Figure 1: The BigDAWG polystore architecture consists of four layers - storage engines, islands, middleware and applications. For clarity, we have not drawn every CAST operation. To relate this to the ocean metagenomics dataset, we also mention where components of the dataset are stored. The solid lines between islands and engines correspond to Shims.

compiled SQL commands. It internally uses PostgreSQL to store archived streamed data or data too old to persist in memory.

3.1.3 Shims

A shim is a bridge between an island and an engine. It consists of a query parser, a query generator, and a query dispatcher. Currently, PostgreSQL has a shim to the relational island, SciDB has one to the array island, and each of the four engines has a shim that connects to its own degenerate island. The shim for PostgreSQL is implemented based on JSQLParser [26] and the PostgreSQL JDBC. The SciDB shim is implemented with the help of the SciDB JDBC. The shims of Accumulo and S-Store were developed in-house. In the future, we expect to implement new virtual islands for the intersecting semantics of existing islands. An operator node in the new virtual island will inherit fields and methods from single or multiple operator nodes of each island. As a result, a developer will be able to quickly develop new shims for existing engines. If each operater node from the new island maps to one in an existing island, the developer will be able to reuse an existing shim. Otherwise, they may construct a new shim by first creating a mapping between ASTs in the new island and ASTs in the old island, and then wrap an existing shim in this new mapping layer.

3.1.4 Loader, Migrator, Planner and Executor

The BigDAWG middleware coordinates operations across engines using islands, shims and casts. In addition, the middleware uses four other components to supports its operation:

1. *T*he data loader: ingests data in parallel to improve the performance of the underlying engine.

2. *T*he planner: tracks the locations of data sets, and creates query execution plans within and among different islands.

3. *T*he executor: coordinates with multiple instances of the same engine to carry out intra-island queries.

4. *T*he migrator: collaborates with the Planner and Executor to move data sets between two instances of supported database engines.

We discuss how these four components function to support a query in the next section.

3.2 BigDAWG processes

3.2.1 Data Loading

The first step in the data analysis pipeline is the data loading process. The raw ocean metagenomic data is stored in CSV or FASTQ format, both of which are text formats. The raw sequence data (genomics data) represents the largest portion of the ocean metagenomic dataset and is stored in the FASTQ format. Most DBMSs can load data directly from CSV files, however, the FASTQ format has to be transformed before it can be loaded into a database. The data in the FASTQ format has to be parsed during transformation, which constitutes the biggest consumer of CPU cycles. Our approach is to transform the FASTQ format into a binary format which can be loaded directly into a database. Databases, such as PostgreSQL, SciDB or Vertica, expose native external binary formats which enable faster data loading. An additional optimization is a division of the data into many separate pipes or files during transformation. Thus, the genomics data can be transformed into a binary format and loaded, for example, into SciDB in parallel from many clients on separate nodes to fully leverage multi-core and multi-node parallelism. The other types of data in the ocean metagenomic data set, the discrete sample metadata or sensor metadata, are much smaller than the genomic data and their loading is not on the critical path. Finally, the streaming real-time navigation data is sent directly to S-Store, where it can be transformed and migrated to other engines within BigDAWG.

3.2.2 Query execution

A user query may refer to many data sets distributed on different engines across multiple islands. A sub-query issued to many datasets on the same island is automatically

handled by the planner and the executor. To move an intermediate result from one island to another, the user makes an explicit *cast* call that invokes the migrator to move data across islands.

We begin our discussion with intra-island queries (or subqueries). A query issued to the relational or array island can be directly mapped to a tree expression using a set of well-defined BigDAWGoperators. This allows the BigDAWG planner to employ a variety of polystore query optimization techniques. We denote islands such as relational and array virtual islands as *operator-based* islands.

Other islands do not naturally support well-defined operator trees. For example, the Accumulo text island frequently receives queries that are filled with procedures and functions specified by the D4M language [17].

The S-Store degenerate island does not support ad hoc queries; only precompiled, stored queries. In these cases, the queries are passed-through unmodified to the underlying database engine; hence, we refer to these as *pass-through* islands.

We focus our discussion of query execution on the more interesting operator-based islands. Further, since the approach used for query execution with the array island is directly analogous to that used for the relational island, we restrict our discussion to the relational island.

When the planner received an SQL query, the query is first sent to a dedicated PostgreSQL instance to be optimized as a single-instance query. This step creates a query execution plan whose filters are pushed down and unused columns eliminated. The planner then converts the query execution plan into an aforementioned AST. The BigDAWG planner then begins constructing the intra-island execution plan. Each sub-tree of the AST, which represents a query that is readily executable on a single engine, is marked as *pruned*. Each pruned sub-tree is wrapped in a separate query *container*, which also identifies the database engine at which the sub-query will be executed. The roots of these sub-trees are subsequently replaced by scans of intermediate results from the containers. After pruning, the planner either allows an underlying engine to perform subsequent query optimization (if all tables are on the same engine) or interprets the original AST as a stub whose leaves refer to intermediate results executed on different engines. Non-leaf nodes in the AST signify how intermediate results are to be combined and processed. The new AST is now denoted as a *remainder* of the pruning.

The BigDAWG Planner takes the remainder as a naïve intra-island execution plan and re-orders and combines its nodes to optimize the plan. The containers and the updated remainder collectively make up the final intra-island execution plan to be delivered to the Executor. The containers are first executed and materialized in parallel and then asynchronously joined or concatenated by the order specified in the remainder. The Executor decides how a pair of engines will migrate its result and how the results are combined. In addition, since the executing engines may not support the language of the island, the Executor will use shims to generate sub-queries coded in a langauge native to each executing engine.

After the Executor performs an intra-island query, the result is either returned or cast by the user to another data model for further processing. To invoke a cast, the user provides a creation statement in the language of the destination island to provide type and data structure conversion information. In the future, we envision that the cast will be automatic: the user may choose not to provide any conversion information and let the system decide what to do. In case of ambiguity, the planner can prompt the user for clarification.

3.2.3 Casts

A cast moves an intermediate result of an intra-island query from one island to another to enable inter-island queries. To call a cast, the user needs to name the destination island and provide information necessary to convert data types and layouts of the data set. In our current implementation, the user provides a complete creation statement in the language of the destination island. The BigDAWG planner internally migrates the intermediate result to a default database in the destination island as soon as it is materialized. In the future, casting will become an interactive process: instead of providing all details of the data conversion, a user may only specify what will be cast and where it will be sent. The BigDAWG planner will consult the catalog for suitable type and data structure conversions and attempt to migrate the intermediate result to the engine that will immediately make use of the data. If it encounters ambiguities in the process, the BigDAWG planner will prompt the user to provide clarifications.

3.2.4 Data migration

The BigDAWG middleware utilizes data migration in two ways: (1) transfer of intermediate results from query executions between database engines, and (2) migration of data as the workload evolves to improve performance. We investigate data migration between PostgreSQL, SciDB, S-Store and Accumulo.

Data migration is a pipelined three-stage process. First, source data is exported from a source database. In this stage, we extract the database object (e.g. table or array) and its metadata (e.g. names and types). Second, the exported data must be transformed and moved to the destination database. This can involve changing the data model and format. Finally, the transformed data is loaded to the destination and if it does not exist, the target object is created with the extracted metadata.

The data transformation module, which converts data between different formats, is the core module of the data migrator. This module is implemented in C/C++ and tuned with Intel VTune Amplifier [1] to eliminate bottlenecks and achieve high performance. This is a complex piece of software since 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. Furthermore, in BigDAWG this is extended to metadata in the binary files. For instance, PostgreSQL's binary format stores the number of fields at the beginning of each row and this information in the current version of PostgreSQL never changes; thus, it can be easily encoded to further minimize the data size.

New data formats will arise from time to time. Hence, extending the transformation module must be simple. To this end, we extract common parts of data formats, (e.g. file (or row) header and footer, attribute metadata and values), and design generic interfaces so when a new data format is added, only the read and write interfaces have to be imple-

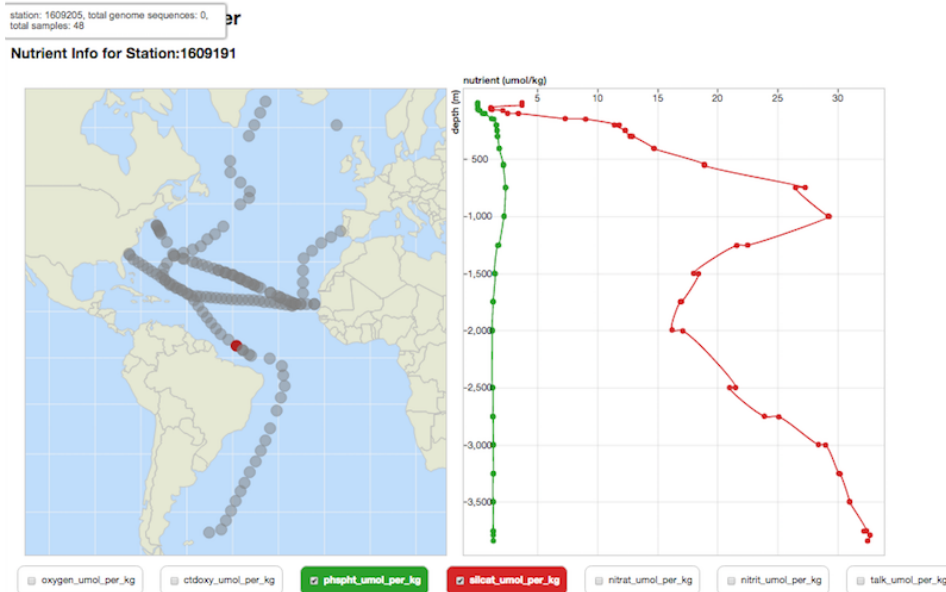

Figure 2: Screenshot of Ocean metagenomic explorer application.

mented and plugged into the transformation module.

Within the data transformation module, a tight loop iterates over values in the input file and produces the output file. This loop is optimized by removing unnecessary branches. At a conceptual level, a list of attributes with proper types is prepared before the main data processing. Inside the loop a value is read from a given input, transformed according to its type, and finally written to the output binary file. Hence, the type selection is done only once and memory consumption is kept to a minimum.

4. DEMONSTRATION

4.1 BigDAWG Demo Architecture

To address the challenges presented by the dataset described in section 2, we have developed a number of polystore applications that can be used by oceanographers to explore, query, and analyze ocean metagenomic data.

The architecture of this demonstration system is shown in Figure 1. As described in the figure, we currently leverage the open source database engines Accumulo, PostgreSQL, S-Store, and SciDB. In our demonstration, we will leverage multiple PostgreSQL instances used to store structured components of the data such as sensor and sample metadata as well as historical data collected from the SeaFlow instrument. Accumulo is used to store free-form text reports as well as certain pieces of the genomics dataset. S-Store is leveraged for recording and processing streaming SeaFlow data and SciDB is used for storing and processing components of the genomic data.

Interaction with these engines occurs through a number of islands, casts and shims as described previously. We also leverage a number of visualization and analysis engines. The open source Vega environment [24] and ScalaR [4] are used for interactive visualizations; Tupleware [9] for data analysis and machine learning tasks; and Macrobase [3] for automated outlier detection.

4.2 Demonstration Applications

The demonstration includes four applications.

1. *Exploration*: The exploration application leverages the BigDAWG relational island, PostgreSQL, and the visualization engine Vega. This application is designed to give researchers a quick look at the overall dataset. For each data collection station, users can plot depth vs. nutrient concentration. A screenshot of the application is given in Figure 2
2. *Navigation*: The navigation application uses streaming data from SeaFlow to help reduce the cost of data collection cruises. In this screen, a researcher can enter a parameter of interest and our system will use historical tracks or real-time streaming tracks to offer navigational suggestions. For example, for greater concentration of a parameter of interest, turn left. This application utilizes the streaming data store S-Store, relational database PostgreSQL and BigDAWG streaming and relational islands. A screenshot of the application is given in Figure 3.
3. *Text and Geo Analytics*: This application helps researchers explore the free form text components of the dataset. Users can look for details about cruise stations and search cruise logs within a particular region to look for keywords of interest. This application utilizes the key-value store Accumulo, D4M [17], PostgreSQL and the BigDAWG text island.
4. *Heavy analytics*: The final component of our demonstration will describe our approach to analytics that cut across metadata and genomics data. We developed an outlier detection system and a spatial predictive model. The outlier detection system leverages Macrobase using BigDAWG to look for outliers in metadata stored in PostgreSQL and genomic data migrated from SciDB. The spatial predictive model utilizes Tupleware for analyzing track data and sample descrip-

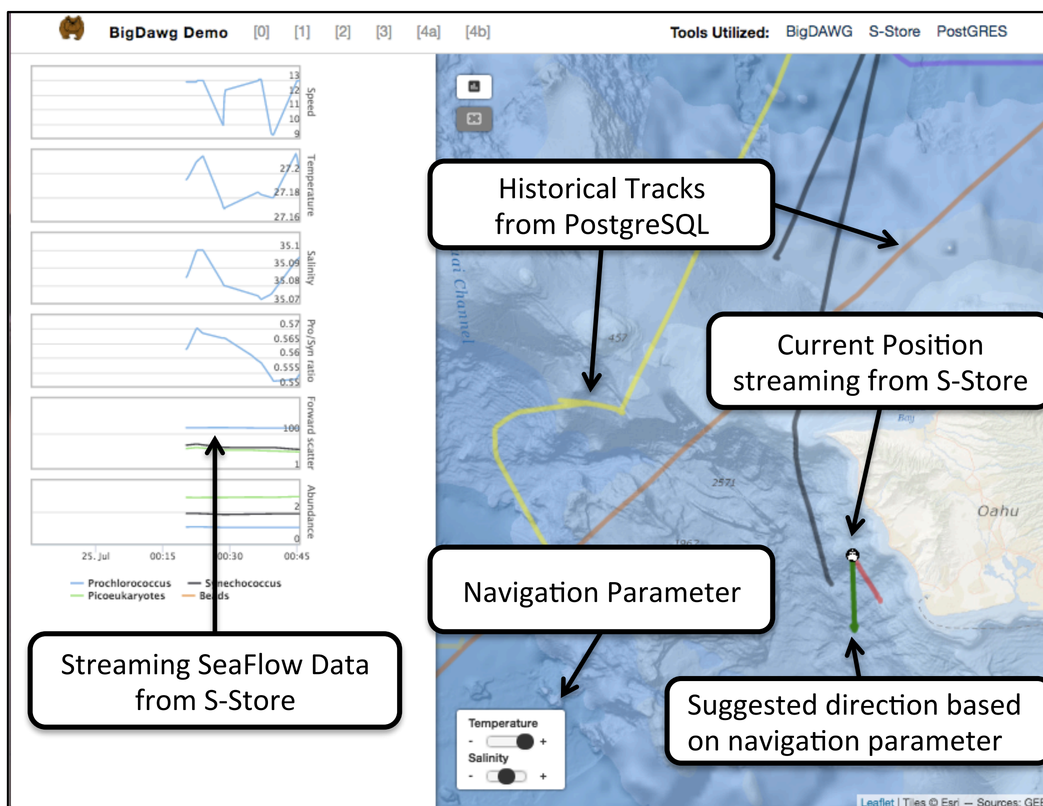

Figure 3: Screenshot of Ocean metagenomic navigation application.

tions stored in separate PostgreSQL instances. These applications utilize the BigDAWG relational and array islands.

All of the applications run on the MIT Engaging One cluster using the SuperCloud database management system [22]. A video of the current state of the demonstration can be viewed at: http://tiny.cc/iul1dy.

4.3 Cross Island Queries

Cross-island queries are the backbone of BigDAWG polystore applications. They allow us to query multiple engines, migrate intermediate results, and mix languages in a single query. Consider the following cross-island query (simplified for the article):

```
bdarray(filter(
  bdcast(
    bdrel(select bodc_station, time_stamp
          , interpolated_salinity
          from sampledata.main)
    , intrp_sali
    , '<bodc_station:int64
        , time_stamp:datetime
        , interpolated_salinity:double>
       [i=0:*,1000,0]')
    , interpolated_salinity < 35))
```

We now break this query down into its components to explain the structure of the query in more detail. At the top level, this query is a filter operator on the array island (*bdarray*) to return all readings near a BODC station that have a salinity level less than 35.

```
bdarray(filter(
      <<< array >>>
    , interpolated_salinity < 35))
```

The array is generated by a cast from data residing in a PostgreSQL database within the relational island. The data in question is all BODC stations for which salinity data is available. This is formed by the following SQL query addressed to the relational island (*bdrel*).

```
bdrel(select bodc_station, time_stamp
      , interpolated_salinity
      from sampledata.main)
```

The cast operation (*bdcast*) takes the result from the SQL query and casts it into an array named *intrp_sali* with three attributes for the BODC station, a time stamp and the salinity value.

```
bdcast(
  bdrel( <<< SQL query>>>)
  , intrp_sali
  , '<bodc_station:int64
      , time_stamp:datetime
      , interpolated_salinity:double>
     [i=0:*,1000,0]')
```

The final line describes the layout of data rows within SciDB.

Cross-island queries provide an expressive, compact notation for queries that span multiple data models. The cast operations play a key role by letting the result from one island directly feed into a subquery that targets a different island.

4.4 Data Migration

We have investigated many techniques which can accelerate data migration. We describe three in detail: migration in different data formats, task-level and data-level parallelism.

4.4.1 *CSV, binary with transformation, and direct binary migration*

In the development of our data migrator, we implemented and tested a number of potential migrators [12]. First, a CSV based migrator: it is highly portable but inefficient as it consumes a significant amount of CPU cycles during the parsing (finding delimiters and other special characters) and deserialization (data type conversion from text to binary representation). Second, we developed a migrator based on external binary formats, which are exposed by many DBMSs for faster data loading and export. Each database requires clients to adhere to its own binary format, thus this type of migration necessitates an intermediate binary transformation. This approach removes the need for type conversion; however, it incurs an additional conversion step. Third, we changed source code of one of the databases to directly generate binary data that can be processed by both source and destination DBMSs. In our experiments, the migration with intermediate binary transformation is nearly 3X faster than CSV migration and the direct binary migration is about 4X faster than CSV migration.

4.4.2 *Task-level parallelism*

To achieve rapid data migration, parallelization is required for each stage of data migration [12, 13, 21]. First, PostgreSQL allows us to load data in parallel from many clients to a single table but does not support parallel export. We present how we enable parallel export from PostgreSQL. Second, data partitioning is an inherent property of S-Store, so we can easily harness parallelism by exporting data from separate partitions simultaneously.

The default implementation of data export from PostgreSQL executes a full table scan, processes data row by row, and outputs a single file. We parallelized this process by distributing data on the level of database pages. Each page is distributed in round-robin fashion to separate files. We run many database clients, each of which processes and exports (to a single file) part of a table (a range of pages), instead of the whole table. Additionally, the consecutive batches of pages can be processed by each client to improve sequential I/O. To illustrate the performance of our method, we extract 10 GB of TPC-H data from PostgreSQL using our parallel page-wise export. The parallel export is even 20X faster than the traditional single-threaded data extraction from PostgreSQL (Fig. 4)[1].

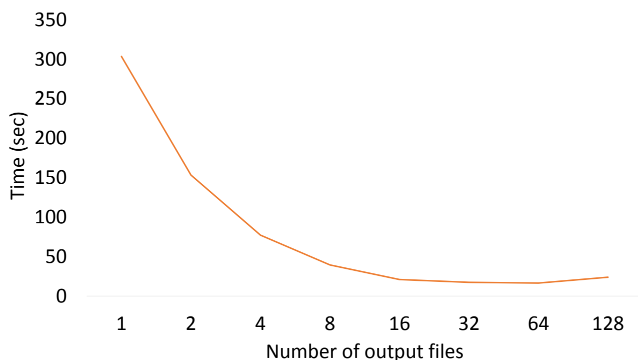

Figure 4: Parallel data export from PostgreSQL (data from TPC-H [2] benchmark of size about 10 GB). The degree of parallelism is equal to the number of output files.

We extended the S-Store database to export data in the binary formats supported by PostgreSQL and SciDB. The partitioned data in S-Store enables us to export each partition in a separate process. Our experimental results indicate that binary migration is much faster than a CSV-based migration for low degrees of parallelism, however, the speed of two methods converges as parallelism increases.

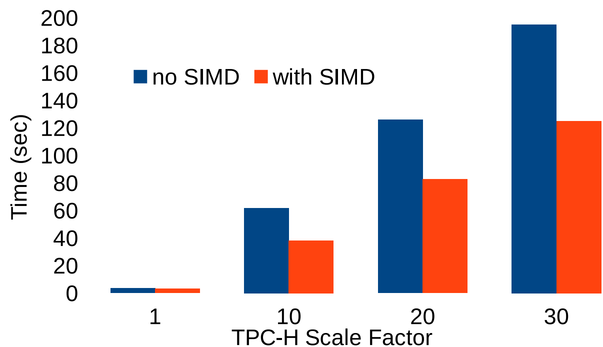

Figure 5: The execution time for parsing (finding new lines) with and without SIMD. The achieved speed-up is about 1.6X. (during loading of the *lineitem* table from the TPC-H [2] benchmark).

4.4.3 *Data-level parallelism*

We investigated how SIMD (SSE 4.2) can be used for faster CSV parsing during data loading to PostgreSQL. Parsing in PostgreSQL is split in two phases: (1) extract lines, (2) identify fields. First, it scans through the input file and searches for new lines. Once a new line is found, it is copied to a separate line buffer and sent to the next stage which finds fields (within the same line buffer). It occurs that the most time consuming part of the parsing process is extracting new lines. The process of finding lines for a plain text and CSV format is fused into a single block of code whereas there is a separate processing of fields for both plain text and CSV format. A new buffer is prepared for each line during parsing. The size of the line is not known beforehand (as there can be fields of varying size), thus we have to check if the new characters to be added do not exceed the size of the buffer and re-size it from time to time.

Finding lines requires many comparisons for each character. Thanks to SIMD, we skip many if-else comparisons for each character, as they are executed much faster using the *equal any* programming model from SSE 4.2. Moreover,

[1]The experiment was conducted on Intel® Xeon® CPU E7-4830 @ 2.13GHz (32 physical cores), 32 KB L1, 256 KB L2, 24 MB L3 cache shared, 256 GB of RAM, running Ubuntu Linux 12.04 LTS (64 bit) with kernel version 3.2.0-23. RAID-0 disk bandwidth about 480 MB/sec for writes and 1.1 GB/sec for reads. For all of our experiments, we compile PostgreSQL 9.6 (dev) with -O2 and -march=native flags (gcc 4.9.4).

skipping many characters means avoiding many checks for the line buffer and coping more characters to the buffer at a time. For line parsing, we keep the SIMD registers full - we always load and process 16 characters in a SIMD register until the end of the file. In our micro-benchmark, we changed the parsing of lines in PostgreSQL and measured directly its performance while loading the *lineitem* table from the TPC-H benchmark. The achieved speed-up for finding new lines with SIMD is about 1.6X (Fig. 5)[2].

5. CONCLUSION

This paper describes a demonstration of the BigDAWG polystore system. We considered real data from the the Chisholm Lab at MIT and analysis workflows from the ocean metagenomic community.

The unique contribution of this paper is its focus on the shims and casts within BigDAWG and how they interact with queries to support polystore applications. Other papers introduce these BigDAWG components [8, 16, 19, 14], but this paper is unique in that it describes how they work together to solve real problems.

We describe BigDAWG as a “reference implementation” of the polystore concept. Polystore systems are a relatively new approach for integrating data in complex systems. A great deal of work is required to fully develop the polystore concept. As just one example, consider the “N-squared” casting problem. As each new storage engine is added to a polystore system, the total number of engines (N) grows and the number of cast operations grows as the square of N. For all but the smallest values of N, this can make the cost of adding new storage engines prohibitively expensive. Further research will explore ways to automate generation of cast operations.

The greatest value of a polystore system is the ability to issue queries that span multiple storage engines. This has always been possible at the application level by issuing independent queries, materializing the results, and integrating them together on a case by case basis. We expect and have shown elsewhere [16], however, that an additional benefit of polystore systems will be enhanced performance from matching data to the storage engine. We do not have a diverse range of examples, however, demonstrating this performance advantage. Future research will be needed to validate and fully characterize these performance advantages.

6. ACKNOWLEDGMENTS

This work was supported in part by the Intel Science and Technology Center (ISTC) for Big Data. The authors wish to thank our ISTC collaborators: Jennie Duggan, Aaron Elmore, Kristin Tufte, Stavros Papadopoulos, Nesime Tatbul, Magdalena Balazinska, Bill Howe, Jeffrey Heer, David Maier, Jeremy Kepner, Michael Stonebraker, Samuel Madden, Tim Kraska, Ugur Cetintemel, and Stan Zdonik. We also wish to thank the members of the Chisholm Lab, Penny Chisholm, Steve Biller, and Paul Berube for their scientific advice. Finally, we wish to thank the MIT Lincoln Laboratory Supercomputing Center for their infrastructure support.

[2]The experiment was conducted on Intel® Core™ i7-5557U CPU @ 3.1GHz (4 physical cores), 32 KB L1, 256 KB L2, 4 MB L3 cache shared, and 16 GB RAM, running Ubuntu Linux 14.04 LTS (64bit) with kernel version 3.16.0-38. 250GB SATA SSD: bandwidth about 250 MB/sec for writes and 500 MB/sec for reads.

7. REFERENCES

[1] Intel VTune Amplifier 2016. http://software.intel.com/vtune/.

[2] TPC-H Benchmark: Standard Specification. http://www.tpc.org/tpch/.

[3] P. Bailis, D. Narayanan, and S. Madden. MacroBase: Analytic Monitoring for the Internet of Things. *arXiv preprint arXiv:1603.00567*, 2016.

[4] L. Battle, M. Stonebraker, and R. Chang. Dynamic reduction of query result sets for interactive visualizaton. In *Big Data, 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[5] S. J. Biller, P. M. Berube, D. Lindell, and S. W. Chisholm. Prochlorococcus: the structure and function of collective diversity. *Nature Reviews Microbiology*, 13(1):13–27, 2015.

[6] 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*, pages 963–968. ACM, 2010.

[7] 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*, 7(13):1633–1636, 2014.

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

[9] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Cetintemel, and S. B. Zdonik. Tupleware: “Big” Data, Big Analytics, Small Clusters. In *CIDR*, 2015.

[10] S. Dodson, D. O. Ricke, and J. Kepner. Genetic sequence matching using D4M big data approaches. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6. IEEE, 2014.

[11] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. the bigdawg polystore system. *Sigmod Record*, 44(3):11–16, 2015.

[12] A. Dziedzic, A. Elmore, and M. Stonebraker. Data Transformation and Migration in Polystores. In *High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016.

[13] A. Dziedzic, M. Karpathiotakis, I. Alagiannis, R. Appuswamy, and A. Ailamaki. DBMS Data Loading: An Analysis on Modern Hardware. In *Proceedings of the 7th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*, 2016.

[14] 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*, 8(12):1908–1911, 2015.

[15] A. Fuchs. Accumulo–Extensions to Google’s Bigtable Design. *National Security Agency, Tech. Rep*, 2012.

[16] 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*, pages 1–6. IEEE, 2016.

[17] V. Gadepally, J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, L. Edwards, M. Hubbell, P. Michaleas, J. Mullen, et al. D4M: Bringing associative arrays to database engines. In *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, pages 1–6. IEEE, 2015.

[18] G. Group et al. The GEOTRACES intermediate data product 2014. *Marine Chemistry*, 177:1–8, 2015.

[19] 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*, pages 1–6. IEEE, 2016.

[20] J. Kepner, W. Arcand, W. Bergeron, N. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz, et al. Dynamic distributed dimensional data model (D4M) database and computation system. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5349–5352. IEEE, 2012.

[21] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant loading for main memory databases. *Proc. VLDB Endow.*, 6(14), 2013.

[22] A. Prout, J. Kepner, P. Michaleas, W. Arcand, D. Bestor, B. Bergeron, C. Byun, L. Edwards, V. Gadepally, M. Hubbell, et al. Enabling on-demand database computing with MIT SuperCloud database management system. In *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, pages 1–6. IEEE, 2015.

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

[24] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE transactions on visualization and computer graphics*, 22(1):659–668, 2016.

[25] J. E. Swalwell, F. Ribalet, and E. V. Armbrust. SeaFlow: A novel underway flow-cytometer for continuous observations of phytoplankton in the ocean. *Limnology & Oceanography Methods*, 9:466–477, 2011.

[26] T. Warneke. Jsqlparser. https://github.com/JSQLParser/JSqlParser, 2011–2016.