FastDAWG: Improving Data Migration in the BigDAWG Polystore System

Xiangyao Yu[1], Vijay Gadepally[2], Stan Zdonik[3],
Tim Kraska[1], and Michael Stonebraker[1]

[1] Massachusetts Institute of Technology,
Computer Science and Artificial Intelligence Laboratory
[2] Massachusetts Institute of Technology, Lincoln Laboratory
[3] Computer Science Department, Brown University

Abstract. The problem of data integration has been around for decades, yet a satisfactory solution has not yet emerged. A new type of system called a polystore has surfaced to partially address the integration problem. Based on experience with our own polystore called BigDAWG, we identify three major roadblocks to an acceptable commercial solution. We offer a new architecture inspired by these three problems that trades some generality for usability. This architecture also exploits modern hardware (i.e., high-speed networks and RDMA) to gain performance. The paper concludes with some promising experimental results.

Keywords: Polystore · BigDAWG · Migration · RDMA.

1 Introduction

The database landscape has been plagued for decades by problems of data integration. Operational data is stored in multiple heterogeneous database management systems (DBMSs) each of which may differ in their data model, their vendor, or their schema. Typically, the component systems run on different machines adding communication problems to this nightmare. How can a user extract and combine information from multiple heterogeneous sources? Most systems have tried to solve this problem in its full generality, by integrating arbitrary databases. But what if we could simplify the notion of what can be integrated in a way that is still useful and that addresses some of the long-time impediments to wide-spread adoption? The architecture presented in this paper attempts to do just that.

A number of recent collection of papers popularized the notion of polystores [5, 7, 8, 11, 18]. The concept was to allow K islands of information each of which supports a common data model and a common island query language along with a candidate set of database engines. Examples include a relational island, an array island, and a key-value island. An individual DBMS, call it D, would join an island by constructing a wrapper that maps between the island query language and the local query language of D and a local-cast that converts D's data representation to the standard island format. In addition, there is an an island-cast from each island to every other island (very often, direct casts are made between particular engines of interest). Hence, when system A needs to send data to system B, then A converts its data to standard island format,

an island-cast is applied to get to the other island representation, and then finally the data is converted to the local dialect via a local-cast. For more information consult the individual papers that discuss the components of the middleware in detail [4, 6, 9, 14]

We have implemented this island concept in a system called BigDAWG. At present, we have two islands in operation, a relational one with Postgres, MySQL, and Vertica as members and an array island composed of SciDB. We have identified the following problems with our initial architecture:

1. **Learning multiple query languages is a daunting task.** Our previous proposal assumed that a BigDAWG programmer would know multiple query languages. In practice, this is a lot to ask. Most programmers have a main language in which they are competent. A realistic polystore proposal should not require a multi-lingual facility.
2. **Data movement is too slow.** Moving one record at a time using an insert in SQL is exceeding slow. Even connecting to the bulk load facility of the various systems is quite slow. We need a faster way to move data between islands.
3. **Wrappers are inefficient and hard to write.** The idiosyncrasies of the various query languages make wrappers tedious to write. Also, different type systems, treatment of nulls, integrity constraints, and the complexity of SQL just make matters worse. To add MySQL to the BigDAWG island took multiple months of effort. We need a simpler architecture that makes it easier to add new DBMSs to a polystore system.

In this paper, we propose a new polystore architecture that addresses all three concerns above. Section 2 presents a simpler overall architecture which does not require a user to learn multiple query languages and effectively addresses Challenge 1 above. Section 3 continues with a data movement system using networks with remote direct memory access (RDMA) supports and solves Challenges 2 and 3. Section 4 shows experimental results that make us optimistic about the success of this new architecture. Finally, Section 5 discusses some previous work, Section 6 talks about future work, and Section 7 concludes the paper.

2 A New Polystore Architecture

One issue with the current BigDawg system is that a user has to learn the query languages of multiple systems in order to run queries across them. The architecture proposed in this section requires a user to learn only one query language which will be translated to different systems, which effectively solves Challenge 1 discussed in the previous section.

We assume that a user has a main query language and most of his data is in systems that support some dialect of his main query language. Hence, an analyst might know an array query language, and a business intelligence expert would know some relational query language, such as Postgres or Vertica or Redshift. We will term this system as the user's **main** system.

Most sophisticated systems support the notion of **foreign** objects; for example, RDBMSs support the notion of external tables. We assume that all foreign objects (e.g.,

arrays, graphs) will be specified using the foreign object interface of the main system. We require this external interface to be extended with simple notions of indexing, so the query optimizer of the main system can do a complete query plan for any user query in the dialect of the main system. This query plan will be executed locally until foreign objects must be dealt with.

Assume the main system is relational. In this case, assume joins are coerced to run locally. Hence, the operation that is associated with an external table is a predicate and perhaps a projection. The one-table query is the result of pushing down all predicates and collecting them into a single query with the combined predicate.

Listing 1 shows an example of a query `Q` executed over two machines. Table R1 resides on the main system and table R2 is an external table residing on a different machine. In BigDAWG, query `Q` is broken down into three subqueries, `Q1`, `Q2`, and `Q3`, as shown in Listing 2. At the local machine, `Q1` filters table R1 and stores the results into a temporary table `T1`. Meanwhile at the remote machine, `Q2` filters table R2 and stores the results into a temporary table `T2`. After both filtering operations finish, BigDAWG migrates `T2` from the remote machine to the local machine and performs the join locally.

Listing 1. Example Query Execution.

```
Q:  SELECT *
    FROM R1, R2
    WHERE R1.A1=a AND
          R2.A1=b AND
          R2.A2=R1.A2
```

Listing 2. Query Q is broken down into subqueries Q1, Q2, and Q3.

```
Q1: SELECT *     // filter R1 locally
    INTO T1
    FROM R1
    WHERE R1.A1 = a

Q2: SELECT *     // filter R2 remotely
    INTO T2
    FROM R2
    WHERE R2.A1 = b

Q3: SELECT *     // migrate T2 and perform the join locally
    FROM T1, T2
    WHERE T1.A2 = T2.A2
```

The data movement discussed in the next section deals with the remote subquery (`Q2`) of the example above. A filtered table is returned to the main system which is stored and query execution continues. Hence, the data for all joins and aggregates can be fetched by pushing the predicates to the remote node. In this case, the entire query semantics is that of the main system.

3 Data Movement and Semantic Transformations

In this section, we propose an RDMA-based data movement design that addresses Challenges 2 and 3 discussed in Section 1. Specifically, Section 3.1 describes how datatype conversion works in the new architecture. Section 3.2 discusses how the system uses RDMA to accelerate data transfer and datatype transformation. Section 3.3 shows the execution of an example query. Finally, Section 3.4 compares the proposed data movement strategy with data migration solution in the current BigDAWG system.

3.1 Semantic Transformations

In this section, we describe the required steps in manipulating data across systems. Consider a main system `M` and a second system `S`. If a user wishes to interact with an object in the second system, he must enter an external object schema into the catalog at `M`. Thus, when the query execution engine needs data from `S`, it will look it up in `M`'s catalog where it will find the external object definition. The data conversion logic will be included in the definition.

Assuming a polystore system that supports a relational database, an array store, a graph database, and a key-value store, the data conversion logic for each type of main system is discussed below:

Relational database: A key-value store, a graph database, and an array store can all be "table-ized" in a straightforward way and be queried using SQL. Hence, it is easy for the owner of any of these systems to export a collection of tables, whose schema information can be used to access remote objects by entering it into remote catalogs.

Array store: Relational tables are a special case of arrays. A graph store can easily export an incidence matrix and a node matrix. Similarly, a key-value store is a degenerate array.

Graph database: Each tuple in a relational database can be considered a node of a graph and the foreign key relationship can be considered an edge of the graph. Similarly, a collection of key-value stores and arrays can be considered separate nodes in a graph.

Key-value store: A table can be exported as multiple key-value stores with each attributed exported as a key-value pair. An array or a graph database can be table-ized and then exported the same way.

As a result, **data wrappers** at this level are straightforward to construct. Most systems support the notion of predicates so a predicate on the main system can usually be converted straightforwardly to one on `S`, so predicates can be pushed into the remote system. Of course `M` and `S` may have different semantics for nulls and operators on single objects. Hence, pushing predicates has to be optional.

3.2 RDMA-Based Data Movement

We assume that all nodes are connected by networks with remote direct memory access (RDMA) support. RDMA allows a computer to directly access data in a remote computer's main memory without the intervention of the remote CPUs or the operating system (OS). Compared to traditional TCP/IP networks, RDMA can provide an order

of magnitude lower latency and higher network bandwidth [3], making it a promising technology to replace TCP/IP based networks in small to medium sized computer clusters.

Besides high bandwidth and low latency, the next generation of InfiniBand network interface controller (NIC) is equipped with embedded compute capability (e.g., Mellanox BlueField SmartNIC integrates the NIC with ARM processors [2] and the Innova Flex adapters integrate the NIC with FPGA [1]). As a result, type conversion can be done in the processor within the NIC, thereby not requiring CPU involvement. Null conversion can be similarly accomplished. We first discuss the protocol for local execution of query `Q` with potentially remote data:

The site `M` sends an RPC request over RDMA to `S` to fetch remote object `O`. We use the term object to mean any collection data type (e.g., relation, array, graph) that is supported by `S`. Site `S` does the appropriate casting to the data model of `M` as noted above and returns a set of locations in `S`'s memory where the result is located. `M` then directly reads the memory locations on `S` using RDMA and continues with the local query plan.

If a predicate `P` is being pushed, then the RPC request is `P(O)`. In this case, `S` does casting plus filtering and returns a collection of memory locations as above.

3.3 An Example of Query Execution

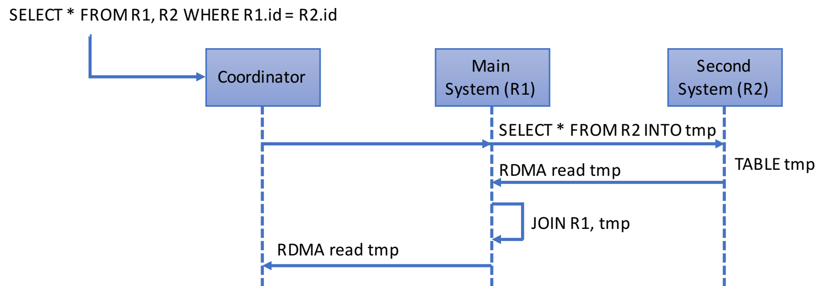

Fig. 1. An example query executed over two systems, main and second systems, with tables R1 and R2 respectively.

Consider a simplified version of the query example presented in Section II. To understand the proposed architecture, we use an example of two relational tables, `R1` and `R2`, located on two nodes, main system and second system, respectively. The coordinator node (which can be co-located with the main system) is responsible for receiving client queries and dispatching them to other nodes. When a query, such as `select * from R1,R2 where R1.id=R2.id` is received, the coordinator breaks the query into two pieces, similar to the breakdown in Listing 2, and dispatches it to the two systems for execution. Figure 1 describes the flow of messages for this example.

When the coordinator receives the query from the client, it first instructs the second system to do a part of the query locally (`select * from R2 into tmp`). In this

example, `tmp` is the location in the second system's memory that can be directly accessed by the main system through RDMA. Once this is complete, the main system is notified and performs the join of table R1 and table `tmp` locally. Finally, the results are sent back to the coordinator and the client.

In the case of two systems with dissimilar organizations (e.g., arrays or key-value stores), we assume that the remote systems (the second system in the above example) can be table-ized (i.e, we can consider the object foreign to the main system as a table). When there are dissimilar datatypes the above message between the main and remote systems will need to include information about how the type conversion is to be performed. The processor within the NIC can then perform the type conversion.

3.4 Comparison with writing BigDAWG connector

We believe that the proposed system can significantly reduce the amount of effort to integrate a new database system. Adding a new database engine to BigDAWG is a non-trivial task. As described in [19], there are a number of steps to be followed to add a new system (within an existing island). First, the developer must define a connection to the database. With relational systems, one can leverage a JDBC driver. The classes for the new database engine must then be generated. Next, a query generator is required that can translate one query language to another. In the case of different datatypes, some common representation must be defined. Once the engine definitions are complete, islands are modified in order to see this new engine. Assuming that queries will move data from one system to another, developers need to write export and load classes that can be used for migration. Again, this needs to take datatype conversion into account. Next, migrators are needed for all possible migrations. New migrators are registered in the middleware. Finally, catalog entries are required in the BigDAWG middleware. For MySQL and Vertica, adding each new engine was on the order of 2000 lines of code (each).

In short, adding a new engine to a system such as BigDAWG requires three major development efforts: 1) Connections, 2) Query generation / data conversion and 3) Migration. We believe that the proposed architecture can greatly simplify this process. By removing the need for users to translate queries from one global query language to a local query, we drastically simplify the query generation step from the description above. Additionally, writing custom data migrators between each system can be time consuming and difficult. Our proposed system relies on the well-defined RDMA protocol that can be used on a variety of interconnects.

4 Performance analysis

4.1 Experimental Setup

In this section, we look at the relative performance of using RDMA vs. TCP/IP for data movement. We use predicted performance when comparing against the BigDAWG migrator.

The experiments in this section are performed on two machines, each with an Intel Xeon CPU E5-2660 v2 processor and 256 GB of main memory and runs Ubuntu

14.04.1. Both machines are equipped with a Mellanox Connect IB EDR NIC, which supports a theoretical bandwidth of 100 Gigabit per second. Each machine is also equipped with an Ethernet NIC that supports a theoretical bandwidth of 1 Gigabit per second.

For the BigDAWG experiments, we installed BigDAWG v0.1 [8] on both servers. Each server runs PostgreSQL [16] as the native database system.

4.2 RDMA vs. TCP

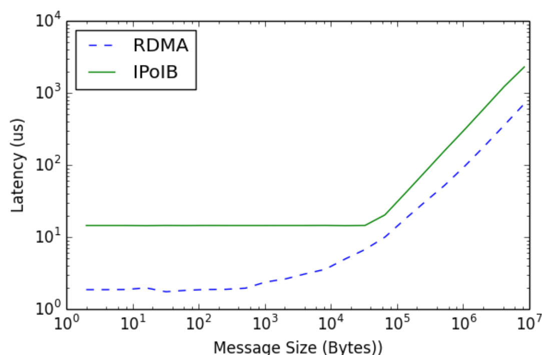

Fig. 2. Network latency with different message sizes.

We now compare the performance of RDMA with TCP/IP over the same InfiniBand network. In this case, TCP is implemented using IP over InfiniBand (IPoIB). As discussed in Section 3.2, one advantage of RDMA over TCP is the lower network latency due to bypassing the network stack. This is shown in Figure 2 where we measure the network latency of both RDMA and TCP at different message sizes.

With small messages, the latency of a TCP message stays constant at 14 μs, which is primarily the time to process the message in the operating system. RDMA, in contrast, incurs only 1.9 μs latency with small messages. This is because RDMA queries do not require the involvement of the OS and only minor CPU computation is required at the client side. As the message size increases, the latency for both RDMA and TCP increases. But the latency of RDMA remains lower than that of TCP.

4.3 InfiniBand vs. Ethernet

In this experiment, we compare the network bandwidth of InfiniBand and Ethernet. The results are shown in Figure 3. With both network settings, the bandwidth consumption increases as messages get larger, until the bandwidth saturates when the message size reaches 2KB. Regardless of the message size, the bandwidth of InfiniBand is around 2 orders of magnitude higher than that of Ethernet. This matches the theoretical bandwidth gap between these two types of networks (i.e., 1 Gigabit vs. 100 Gigabit).

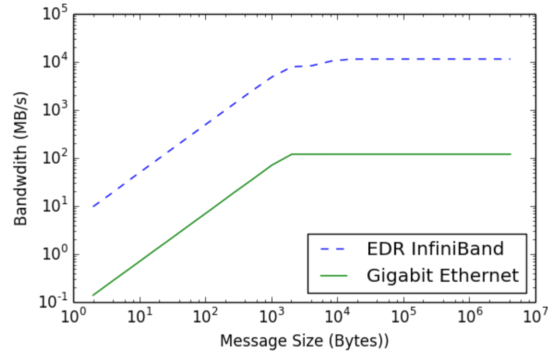

Fig. 3. Network bandwidth of InfiniBand and Ethernet with different message sizes.

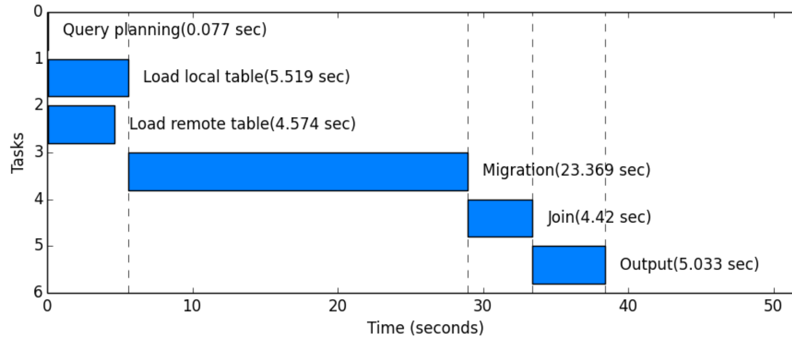

Fig. 4. The runtime breakdown of Query Q1 running on BigDawg.

4.4 BigDAWG Comparison

In this section, we investigate the performance bottleneck of BigDAWG and study how much performance improvement RDMA can bring to BigDAWG. To perform this study, we deployed two PostgreSQL tables, S at the local machine and R at the remote machine. Both tables have the same schema:

Table S s_key: integer, s_value: char (1000)
Table R r_key: integer, r_value: char (1000)

Each table contains one million rows which corresponds to roughly 1 GB of storage. The BigDawg system runs the following query over tables S and R where table S is on the main server and table R is on the remote server.

```
Query Q1:
    SELECT * from S, R
    WHERE S.s_key = R.r_key
```

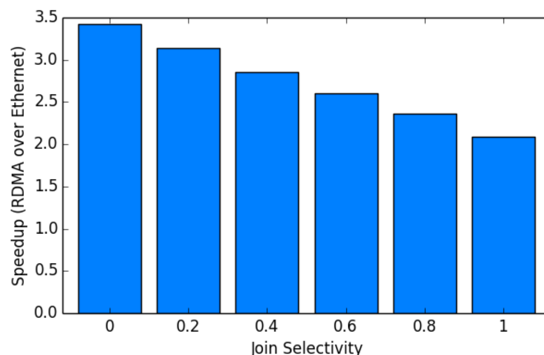

Fig. 5. Predicted speedup of BigDawg when replacing Ethernet with RDMA.

Within BigDAWG, the query is broken down into three subqueries: (1) selecting the local table S, (2) selecting the remote table R and performing the migration, and (3) performing the join locally. The breakdown of execution time of Q1 is shown in Figure 4. The majority of the execution time is spent on the migration process, namely, migrating one table from the remote machine and perform the join operation locally. After integrating RDMA into BigDAWG, this portion of execution time can be largely eliminated, leading to about $3\times$ performance improvement.

In Figure 5, we demonstrate the performance improvement of BigDAWG when using RDMA as the selectivity of the join operation changes from 0 to 1. As the selectivity increases, the performance gain of using RDMA decreases. This is because higher selectivity leads to a larger joined table. Therefore, the portion of execution time spent on the join operation increases and the portion of time spent on migration decreases, limiting the potential improvement of RDMA which can accelerate only the migration but not the join.

4.5 Alternate Architecture and Proof of Concept

Many relational systems support the notion of a foreign table. An alternate architecture uses the concept of a foreign table on S. In this model, we can design a foreign data wrapper that can communicate with the node containing table R. The wrapper is used to define server connections, fetch/update data, and return to the executor. When a query is issued that requires data from R, the foreign data wrapper can fetch this remote data via an RDMA connection. The architecture of Section 3 leverages RDMA as a generalized migrator for polystore systems. This alternate architecture, while likely requiring increased developer effort, can reduce latency, improve query planning and optimization and can be used to largely hide the details of data movement from the end-user.

To demonstrate the viability of this RDMA-based join, we wrote a simple program to manually perform the join using RDMA and compare its performance to a local join. In this experiment, we used two tables S and R that have the same schema and size as in Section 4.4 where each row in table S joins with exactly one row in table R. For the

local join, both tables reside on the same machine which performs a hash join. For the distributed join, the host machine builds a hash table using the local table, and then uses one-sided RDMA read operations to load individual records from the second table over the network and performs the join operation.

Table 1. Execution Time of Local Join vs. RDMA-based Remote Join.

	Local Join	Remote Join
Execution Time	0.798 sec	0.806 sec

Table 1 shows the runtime of both the local join and the RDMA join. Overall, the RDMA join is only 1% slower than the local join, although 1 GB of data is transferred over the network during that period of time. For this type of data intensive query, the computation is the high pole in the tent and RDMA does not limit the system performance at all. Note that the performance number in Table 1 is much better than the performance number measured on PostgreSQL. This is because the hand-optimized join code does not have some overhead in PostgreSQL.

5 Previous Work

The notion of making database systems that interoperate has a long history. In the early 1980s the topics of multi-databases and federated systems were the focus of much research [10, 13, 15]. The high-level vision of these systems was very much like the concept of a polystore, but the implementation challenges were different.

First, a multi-database was built from a loosely-coupled collection of computers that communicate via TCP. Queries that required a lot of very expensive data motion, typically made the system unusable. Second, The multi-database system needed to be capable of integrating any data model and complete with any queries making interoperability a difficult task. Foreign database schemas were converted first to a common intermediate representation and then translated to the target systems schema/model which was complex and slow.

We address the data movement problem by using RDMA to make remote data access competitive in speed to local RAM. We address the complexity issue by limiting the models that can interoperate. We show that we can cover the common cases even with limited models.

More recently, a number of polystore systems have been developed to address the downsides of multi-databases; examples include BigDAWG [5, 7], CloudMDsQL [12], Myria [18], and Apache Drill [11]. While these systems all have different query interfaces and heterogeneous execution strategies [17], all of them used software-based data wrappers and migration policies. The RDMA based fast migration solution that we propose in this paper is able to benefit all of these systems above.

6 Future Work

Compared to BigDAWG, one limitation of FastDAWG is that a query is executed only on the main system—the other systems in the polystore handle data conversion and predicates, but not query execution. BigDAWG, in contrast, can execute different parts of a query using different systems, thereby potentially achieves better overall performance. While a FastDAWG user can still choose the main system for maximal performance, the level of flexibility is more limited. In practice, however, we believe the majority of most queries are optimized by running on a single system, in which case BigDAWG and FastDAWG will have similar execution plans. We plan to study the performance implication of FastDAWG by comparing its performance to BigDAWG on a variety of queries.

This paper has demonstrated the performance potential of FastDAWG using relational database systems. As future work, we plan to study how data type conversion and predicates can be pushed down to the new-generation of RDMA hardware. Hopefully, we can demonstrate the simplicity and performance improvement of this design.

7 Summary

We have described a new polystore architecture that addresses three shortcomings that we observed in our own polystore called BigDAWG, They are (1) the need to learn multiple query languages, (2) the data movement is too slow, and (3) wrappers are inefficient and hard to write. We make a conscious decision to limit interoperability between systems in order to make the system easier to use. We believe that our choice for how to maintain this balance is a sweet-spot. Our next step is to follow this up with some real-world deployments.

References

1. Innova-2 Flex Programmable Network Adapter. https://goo.gl/xNzVD1 (2018)
2. Mellanox BlueField SmartNIC. https://goo.gl/dic6HH (2018)
3. Binnig, C., Crotty, A., Galakatos, A., Kraska, T., Zamanian, E.: The End of Slow Networks: It's Time for a Redesign. Proceedings of the VLDB Endowment **9**(7), 528–539 (2016)
4. Chen, P., Gadepally, V., Stonebraker, M.: The Bigdawg Monitoring Framework. In: High Performance Extreme Computing Conference (HPEC), 2016 IEEE. pp. 1–6. IEEE (2016)
5. Duggan, J., Elmore, A.J., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T., Zdonik, S.: The Bigdawg Polystore System. ACM Sigmod Record **44**(2), 11–16 (2015)
6. Dziedzic, A., Elmore, A.J., Stonebraker, M.: Data Transformation and Migration in Polystores. In: High Performance Extreme Computing Conference (HPEC), 2016 IEEE. pp. 1–6. IEEE (2016)
7. Elmore, A., Duggan, J., Stonebraker, M., Balazinska, M., Cetintemel, U., Gadepally, V., Heer, J., Howe, B., Kepner, J., Kraska, T., et al.: A Demonstration of the Bigdawg Polystore System. Proceedings of the VLDB Endowment **8**(12), 1908–1911 (2015)
8. Gadepally, V., O'Brien, K., Dziedzic, A., Elmore, A., Kepner, J., Madden, S., Mattson, T., Rogers, J., She, Z., Stonebraker, M.: BigDAWG Version 0.1. In: High Performance Extreme Computing Conference (HPEC), 2017 IEEE. pp. 1–7. IEEE (2017)

9. Gupta, A.M., Gadepally, V., Stonebraker, M.: Cross-Engine Query Execution in Federated Database Systems. In: High Performance Extreme Computing Conference (HPEC), 2016 IEEE. pp. 1–6. IEEE (2016)
10. Hammer, M., McLeod, D.: On Database Management System Architecture. Tech. rep., MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE (1979)
11. Hausenblas, M., Nadeau, J.: Apache drill: Interactive Ad-Hoc Analysis at Scale. Big Data **1**(2), 100–104 (2013)
12. Kolev, B., Bondiombouy, C., Levchenko, O., Valduriez, P., Jimenez-Péris, R., Pau, R., Pereira, J.: Design and Implementation of the CloudMdsQL Multistore System. In: CLOSER: Cloud Computing and Services Science. vol. 1, pp. 352–359 (2016)
13. McLeod, D., Heimbigner, D.: A Federated Architecture for Database Systems. In: Proceedings of the May 19-22, 1980, national computer conference. pp. 283–289. ACM (1980)
14. She, Z., Ravishankar, S., Duggan, J.: Bigdawg Polystore Query Optimization through Semantic Equivalences. In: High Performance Extreme Computing Conference (HPEC), 2016 IEEE. pp. 1–6. IEEE (2016)
15. Sheth, A.P., Larson, J.A.: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. ACM Computing Surveys (CSUR) **22**(3), 183–236 (1990)
16. Stonebraker, M., Rowe, L.A.: The Design of Postgres, vol. 15. ACM (1986)
17. Tan, R., Chirkova, R., Gadepally, V., Mattson, T.G.: Enabling Query Processing across Heterogeneous Data Models: A Survey. In: Big Data (Big Data), 2017 IEEE International Conference on. pp. 3211–3220. IEEE (2017)
18. Wang, J., Baker, T., Balazinska, M., Halperin, D., Haynes, B., Howe, B., Hutchison, D., Jain, S., Maas, R., Mehta, P., et al.: The Myria Big Data Management and Analytics System and Cloud Services. In: CIDR (2017)
19. Yu, K., Gadepally, V., Stonebraker, M.: Database Engine Integration and Performance Analysis of the BigDAWG Polystore System. In: High Performance Extreme Computing Conference (HPEC), 2017 IEEE. pp. 1–7. IEEE (2017)