The BigDawg Monitoring Framework

Peinan Chen*, Vijay Gadepally*†, Michael Stonebraker*

*MIT CSAIL †MIT Lincoln Laboratory
chenp@mit.edu, {vijayg, stonebraker}@csail.mit.edu

***Abstract*—BigDAWG is a polystore database system designed to work with heterogenous data that may be stored in disparate database and storage engines. A central component of the BigDAWG polystore system is the ability to submit queries that may be executed in different data engines. This paper presents a monitoring framework for the BigDawg federated database system which maintains performance information on benchmark queries. As environmental conditions change, the monitoring framework updates existing performance information to match current conditions. Using this information, the monitoring system can determine the optimal query execution plan for similar incoming queries. We also describe a series of test queries that were run to assess whether the system correctly determines the optimal plans for such queries.**

I. Introduction

BigDAWG is a system that utilizes a polystore architecture to enable query processing over multiple databases, where each of the underlying storage engines may have a distinct data model. One important component of BigDawg is a monitoring system which keeps track of past queries' runtime information and utilizes this information to choose the best query plan for an incoming query. The main way the monitoring system associates incoming queries with benchmark queries is by utilizing a signature system.

It has become abundantly clear that there are a multitude of different features that people desire in their data storage engine. Currently, to suit people's interests, there are a variety of different storage engines that each have their own costs and benefits. For common Javascript applications that use a web browser paired with a backend database management system (DBMS), NoSQL engines tend to be well-suited. Similarly, relational column stores are the engine of choice for data warehouses while main memory SQL (NewSQL) systems are best suited for online transaction processing. In general, there needs to be a variety of different storage engines to satisfy different types of applications.

While there are many applications that may be well suited to a single type of storage engine, there are many other applications that would be best implemented with a combination of different storage engines. For example, the Intel Science and Technology Center (ISTC) has built a medical application that utilizes the MIMIC II dataset [1]. This dataset contains patient metadata, text data (notes taken by medical professionals), semi-structured data (prescriptions and lab results), and waveform data (measurements from bedside devices such as heart rate, pulse, etc.). A medical application that utilizes this dataset would ideally support standard SQL analytics, complex analytics (such as computing the FFT of a patient's waveform data and comparing it to what is considered normal), text search (such as looking at patients that medical professions have used specific terms for in their notes), and real-time monitoring (such as detecting abnormal heart rhythms).

Although it is possible to implement such an application using a single storage engine, it can be much more efficient to utilize several different storage engines [2]. To support datasets such as MIMIC II, we developed a polystore database management system architecture called BigDAWG [3]. For the MIMIC II dataset, BigDAWG uses SciDB [4] to store the historical time series data, Apache Accumulo [5] for text, Postgres for patient metadata, and the streaming database S-Store [6] to store and process the real-time waveform data. With the BigDAWG architecture, any query that depends on data stored in multiple storage engines will query all necessary storage engines. For example, to compare current waveforms to historical ones, one can query S-Store and SciDB. To find metadata associated with particular kinds of prescriptions or doctor's notes, one can query Accumulo and Postgres. To run analytics on particular cohorts of patients, one can query Postgres and SciDB. The BigDAWG system automatically develops query plans, executes queries on individual storage or database engines and collates the resulting data back to the end-user. An overview of the BigDAWG architecture is shown in Figure 1. In this architecture, BigDAWG can support multiple islands of information which encapsulate a data model, programming model and candidate set of database engines. In our current implementation, we support a relational island [7], text or associative array island [8], [5], [9], and an array island. Queries that are received by the BigDAWG system can be executed by a single island or multiple islands. The BigDAWG islands also support semantically complete functionality of underlying engines through *degenerate* islands that support only a single engine.

Since similar datasets reside in different islands, BigDAWG must be able to execute queries across different islands. For example, the same piece of data from the MIMIC II dataset may be stored in Accumulo which is accessed throug the associative array island and in PostGRES which is accessed through the relational island. When a query comes in, the BigDAWG system must be able to determine which island and engine to execute queries and/or analytics for highest performance. Often, these queries may also span different islands and BigDAWG must be able to determine the most efficient intra-island and inter-island query plan.

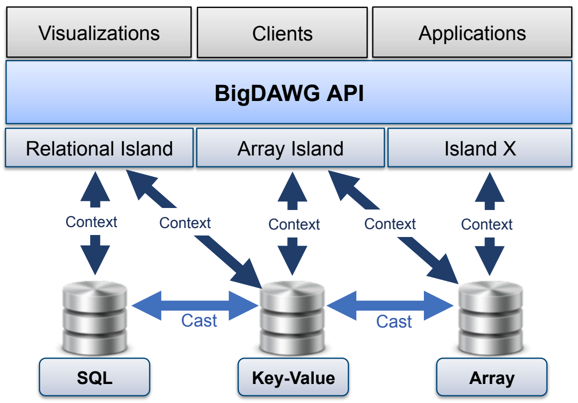

Fig. 1. Multi-Island Data Federation through BigDAWG

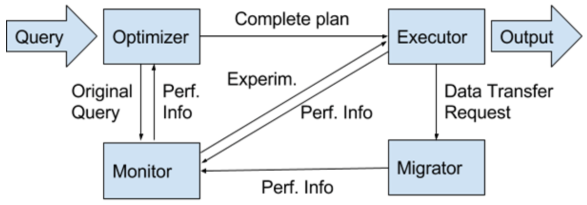

Fig. 2. BigDAWG Middleware Components.

This article describes the implementation and performance results of the BigDAWG monitor. For the remainder of the paper, we refer to this as the BigDAWG monitor which is a part of the BigDAWG middleware. The article is structured as follows. Section II describes our approach to optimizing queries across different islands and disparate data stores. Section III describes the architecture of the BigDAWG middleware. Section IV describes the results of our prototype implementation. Finally, in Section V we discuss further work and conclude.

II. BigDAWG Query Optimization

In this section, we discuss how a query that comes to the BigDAWG system is executed and optimized across the different islands and stores.

Any query that spans multiple islands is divided into stages. In the first stage, we perform all possible local computations that do not require any data movement. At the end of the first stage, we are left with computations on collections of objects at different sites. These computations are divided into independent subsets consisting of a collection of objects $O_1, ..., O_k$ along with "joining" specifications for how the objects are put together and computations on the output of such operations. Although the remainder of the query can be one such collection, we look for collections of size two initially. Effectively, we look for a bushy tree [10] of such computations, which will often be unique.

Suppose that there are L such collections, with each collection operating on non-overlapping sets of objects. In the second stage, we perform each of these L computations in parallel. For each of the L computations, pick a storage engine, E, and move the required objects $O_1, ..., O_k$ to E and perform the computation at E. In this case, the fundamental query optimization decision is the choice of engine E for each collection.

The purpose of the monitoring system is to determine the choice of the storage engine, E, for each collection of objects. The monitoring system has two modes, *training* and *production* mode. In training mode, BigDAWG has the liberty to run an operation at each local engine that contains data for the subquery and record the elapsed time in a BigDAWG database. If a given (sub)query is run at these local engines, then BigDAWG can identify the best location for this operation. Hence, training mode is effectively a training period where queries are run in multiple places and the best location can be identified.

Since it is likely that many similar queries will be run over time under similar system parameters, we construct a signature for each query and store these signatures in the BigDAWG database noted above. Any similar queries we encounter in the future, which match an existing signature, do not need to use training mode, since the best location has already been identified.

In production mode, the system chooses the storage engine arbitrarily for newly encountered queries. Over time, it will run the query with each of the other viable engines in the background. Thus, the best location will be identified over a period of time rather than immediately before running the query.

We expect any given federation to be in production mode all the time or to start in training mode and then shift to production mode. In either case, we assemble a database of subqueries, their signatures, and their timings on various engines. Over time, the BigDAWG monitor builds up a collection of queries, their signature, their elapsed time and what nodes the queries have been run on.

III. BigDAWG Monitor Architecture

The BigDAWG 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). When BigDAWG receives an incoming query, the Planner parses the query and creates a set of viable query plan trees with possible engines for each collection of objects. The Planner then passes these trees to the Monitor which uses existing performance information to determine the tree with the best engine for each collection of objects. This tree is given to the Executor which figures out how to best join the collections of objects and then executes the query, using the Migrator to move data from engine to engine when the plan calls for it. See Figure 2 for a visual overview of the organization and workflow of the system. In this article, we concentrate on the architecture of the Monitor.

The Monitor maintains performance information on past intra-island queries, matches new intra-island queries to similar past queries, and stores migration metrics. For storing migration metrics, the Monitor simply maintains an API for storing and retrieving the metrics. The majority of the Monitor's functionality is devoted to query performance information.

Whenever BigDAWG encounters a new intra-island query that is not similar to any previously seen queries, the Monitor creates an entry for each possible Query Tree of that query. In a Query Tree, each node represents either a piece of data or the result of a database-specific action (DSA), which entails executing a sub-query of the input on a specific database instance. Examples of such sub-queries include transmitting data, filtering according to predicates and receiving and joining data. Edges represent the dependencies between the nodes.

The Monitor generally receives an ordered list of Query Trees for each query from the Planner. For each Query Tree, the Monitor stores the following:

- The index of the Query Tree in the ordered list provided by the Planner.
- The query used to make the Query Trees.
- A signature for the query.
- The last time the Query Tree was run.
- The most recent cost, in terms of elapsed time, of running the Query Tree.

Since the Planner constructs Query Trees deterministically from a query together with a signature, the index of each Query Tree is fixed. Thus, it suffices to store the query, the signature, and the index to uniquely define each Query Tree.

To determine whether an incoming query is new or similar to a previous query, the system uses signatures composed of the following elements:

- A tree representing the structure of the query (sig-1).
- A set of the objects referenced and the predicates involved (sig-2).
- A set of the constants in the query (sig-3).

Using these components, the system outputs a number from 0 to 1 representing how close the two signatures are where 0 means the two are identical and 1 means they are completely different. Specifically, the system first determines the tree edit distance, d, between the two queries' sig-1s using a robust tree edit distance algorithm [11]. It divides this distance by the maximum possible tree edit distance for the two sig-1s. To do this, the system finds the cost of constructing each tree, d_1 and d_2, from scratch. Using these 3 distances, the system computes $t_1 = \frac{d}{\max(d_1,d_2)}$, which is the tree edit distance divided by the maximum possible tree edit distance for the two sig-1s.

Next, the system determines the number of predicates, s, shared between the two queries' sig-2s as well as the maximum l_{max} of the number of predicates in both sig-2s. Using these numbers, it computes $t_2 = \frac{s}{l_{max}}$ which is the proportion of predicates shared between the two queries.

For sig-3, the Monitor determines the difference in the number of constants for the two queries' sig-3s. Since we expect changes in individual constants to not affect the relative ordering of Query Trees for a given query, we want to examine whether the number of constants remains unchanged. To do this the Monitor computes $t_3 = 1 - \frac{\min(a_1,a_2)}{\max(a_1,a_2)}$ where a_1 and a_2 are the number of constants in each respective query. In other words, the system finds the query with less constants and divides its number of constants by that of the other query.

After computing all of these values, the Monitor returns the resulting value $v = \frac{\sum_i t_i * c_i}{\sum_i c_i}$ where c_i is a constant weight that can be set. Let two queries be similar if v is less than some constant and different otherwise.

Upon receiving a query from the Planner, the Monitor compares the new query's signature to that of each existing query in the monitor. If the minimum v is such that the two queries are similar, the Monitor gives the Planner the performance information for the closest existing query. In the case that no existing queries are similar, the Monitor stores the query as a new benchmark.

If the Monitor is in training mode when a new query is added, the Monitor calls the Executor on each of the Query Trees for that query and stores the performance information (runtimes). In the case that the Monitor is in production mode, the Monitor stores each of the Query Trees without determining their runtimes and initializes all of the runtimes to a null value. After adding the new query, the Monitor gives the Planner the runtimes for that query. In the case that all of the runtimes are null, the Planner picks an arbitrary Query Tree to execute.

It is likely that as BigDAWG continues to function, the performance of individual queries will change. This could happen for a variety of factors. For example, some storage engines may scale with the number of entries better than other storage engines, so as entries are added, the initial timings may no longer be valid. Another possibility is that many queries utilize a specific storage engine rather than other storage engines, making those engines have a high load. In order to address this problem, the Monitor periodically reruns existing queries to update their timings based on the current state of the system. To ensure that these timing updates do not interfere with normal BigDAWG functions, the Monitor only reruns queries when engines are not busy. For each island, the Monitor periodically checks whether the load average for that island is under some predetermined threshold. In these cases, the Monitor finds a query in that island and reruns that query.

IV. Analysis

In this section, we describe initial implementation results of the Monitor. Ideally, the Monitor should speed up queries by determining the best engine for each collection of objects received (the engine that minimizes runtime). For the experiments described, we use the following setup: two PostGRES instances on the same physical machine containing the MIMIC II dataset. One of the PostGRES instances contains roughly half of the MIMIC II tables and the other instance contains the remaining tables. The full schema of the MIMIC II dataset is available in [12]. Ten different queries, each with two distinct Query Trees, were used for testing.

TABLE I
TRAINING VS PRODUCTION MODE AVERAGE RUNTIMES

Query No.	Training Mode	Production Mode	Without Monitor
1	826 ms	265 ms	281 ms
2	882 ms	190 ms	346 ms
3	62539 ms	20559 ms	20990 ms
4	491 ms	160 ms	166 ms
5	6592 ms	1977 ms	2308 ms
6	24294 ms	6146 ms	9074 ms
7	28165 ms	7648 ms	10259 ms
8	19073 ms	4496 ms	7289 ms
9	15806 ms	4652 ms	5577 ms
10	78487 ms	23496 ms	27496 ms

A. *Training Mode Gains*

From Table I, we see that running a query in training mode takes more than three times as long as running a query in production mode. This is because in training mode, the Monitor tries every Query Tree for a query before allowing the Executor to run the fastest Query Tree. Each query generates two Query Trees which results in the expected longer query time.

Clearly, running a query once the Monitor already has information from the training mode for the query is faster than generating all of this information and then running the query. At first, it may seem that the training mode cost is prohibitively high and it may be better to perform the query without this mode. The “Without Monitor” column of Table I shows the time required to execute the query in production mode without having any information on the query's Query Trees. In this case, since BigDAWG does not have any information on which Query Tree performs better, it randomly selects a Query Tree to execute.

While the cost of running the query in production mode varies compared to that of running without the Monitor, we can see that in the best case, we can run a production mode query in about 60% of the time as running without the monitor. Thus, assuming queries are often rerun or we often see similar queries, the expense of the training mode run can be justified in the long run.

B. *Response to Environmental Changes*

Another factor that plays a large part in performance is that the relative order of Query Tree execution can change depending on the environment. A Query Tree that initially performs better than others can perform worse relative to other Query Trees depending on factors such as increased load on specific engines. Thus, it is necessary to evaluate whether the Monitor adapts to changes in the environment.

In general, the Monitor is able to update its performance metrics for queries over time. After applying load, we observed that, in general, the monitor adapts within a few seconds. However, the rate at which the Monitor adapts depends on the number of queries in the system. That is, the more queries stored by the Monitor, the longer it takes for a given query to stay up-to-date. This is due to the Monitor updating its performance metrics by re-running the least recently updated query when the load average is under a predetermined threshold. Thus, if the environment changes faster than the rate that the Monitor can run all of its queries, the Monitor may provide outdated metrics for a subset of the queries stored. This may be problematic for queries where several Query Trees have similar performance characteristics. For such queries, it is possible that the Monitor will always rank the Query Trees incorrectly if it is always outdated. For queries where some Query Trees perform vastly better than others, it is unlikely that changes in the environment can cause the better performing queries to become significantly worse than the previously underperforming queries.

C. *Matching Signatures*

One of the main components of the Monitor module is in matching new queries with existing queries using the performance information of previous queries to determine the optimal query plan for new queries. In order to verify whether the Monitor matches queries in such a way that existing queries provide useful information for new queries, for each of the 10 queries, we constructed the following four types of queries:

- A query where the order of the predicates of the existing query are changed
- A query where one of the constants of the query are replaced with a similar constants. Here, we can measure similarity by comparing the number of entries in each table that has that constant. Two constants are similar if the number of entries is approximately the same for both constants.
- A query where one of the constants of the query are replaced with a dissimilar constants.
- A query where one of the tables involved in the query is replaced with a similar sized table.

If the Monitor matches queries correctly, it should recognize that changing the order of the predicates does not actually change the query. During our testing, we found that queries where the order of the predicates are changed are always matched correctly. A subset of the results from executing the previously mentioned 10 queries are summarized in Tables II-VI. we can see that changing the order of predicates does not impact the runtime of the query as expected. Thus, for queries where the structure of the query changed but all else is identical, the Monitor is able to match them correctly with existing queries.

In general, the Monitor should not expect queries where a table has been swapped to perform similarly to the original queries. This is because the Monitor does not have any context on the similarity of tables apart from size and names of the tables. While it is possible that similar sized tables have similar data, the Monitor has no way of ensuring this. Furthermore, it is difficult to identify whether two tables are similar through their names.

TABLE II
QUERY 1 AVERAGE RUNTIMES

Query Type	Query Tree 1	Query Tree 2
Base query	265 ms	296 ms
Predicate order	254 ms	296 ms
Similar constant	278 ms	290 ms
Skewed constant	346 ms	906 ms
Table swap	542 ms	619 ms

TABLE III
QUERY 2 AVERAGE RUNTIMES

Query Type	Query Tree 1	Query Tree 2
Base query	502 ms	190 ms
Predicate order	475 ms	169 ms
Similar constant	479 ms	206 ms
Skewed constant	93023 ms	92321 ms
Table swap	1156 ms	180 ms

TABLE IV
QUERY 4 AVERAGE RUNTIMES

Query Type	Query Tree 1	Query Tree 2
Base query	171 ms	160 ms
Predicate order	165 ms	156 ms
Similar constant	180 ms	180 ms
Skewed constant	226 ms	210 ms
Table swap	620 ms	621 ms

TABLE V
QUERY 5 AVERAGE RUNTIMES

Query Type	Query Tree 1	Query Tree 2
Base query	2638 ms	1977 ms
Predicate order	2563 ms	2054 ms
Similar constant	2625 ms	2015 ms
Skewed constant	4486 ms	3851 ms
Table swap	29781 ms	32474 ms

TABLE VI
QUERY 8 AVERAGE RUNTIMES

Query Type	Query Tree 1	Query Tree 2
Base query	4496 ms	10081 ms
Predicate order	4483 ms	10335 ms
Similar constant	4425 ms	10383 ms
Skewed constant	8456 ms	21385 ms
Table swap	664 ms	317 ms

Our results from Tables II- VI support this. We can see that while swapping a table often results in similar runtimes, it can also result in completely disparate runtimes. Table V shows an example of how changing the table can result in completely different outcomes. For the base query, Query Tree 2 performs better than Query Tree 1, after swapping tables, we can see that Query Tree 1 outperforms Query Tree 2. Table VI provides another example of where swapping a table can change which Query Tree is best. In general, we cannot predict the result of swapping a table, so the Monitor treats queries with tables swapped as dissimilar.

Currently, the Monitor successfully matches all queries where the constants in the query are changed and can be verified by inspecting Tables II- VI. In general, replacing a constant with a similar constant also results in very similar runtimes for all of the test queries. Furthermore, the Monitor preserves the relative order of Query Trees with skewed constants.

To summarize the overall results for all 10 queries, we present the ratio of runtime in executing under Query Tree 1 with Query Tree 2 in Figure 3. For most queries, we would expect that similar queries maintain similar relative runtime (this indicates that the monitor is correctly identifying similar queries). Recall that the Monitor considers all four types of constructed queries to be similar other than queries where a table has been swapped. Thus, we would expect that the ratio of Query Tree 1 runtimes to Query Tree 2 runtimes will stay constant across the first four bars of each query. For the most part, we do see this result. For example, for query 8, the ratio of runtimes is nearly the same across the first four bars.

However, there a few exceptions. For queries 1 and 2 either the initial query or the query after replacement ran for a trivial amount of time (for example, from Table III, we see that the overall runtime of the query before changing to a skewed constant is in the 100s of milliseconds). In these cases, the differences in the runtimes between Query Trees are largely dominated by constant factors such as Monitor overhead. To avoid these problems, users can train the Monitor on queries that take a non-trivial amount of time. This guarantees that replacing the query with any of the 3 similar types of queries will result in queries that exhibit either similar behavior or run in a trivial amount of time.

V. DISCUSSION AND CONCLUSION

Currently, the Monitor finds matching queries fairly inefficiently. Specifically, the Monitor finds the distance, v, between every unique query in the system and the incoming query. It is clear that the overhead necessary to match an incoming query will increase linearly with the number of benchmark queries supported by the Monitor. While this is not a large problem at the moment since we have very few queries, this can become a much bigger problem in the future. One possible way to address this is by keying queries by one of the signatures, such as sig-1, and only determining v for queries that match that signature. The downside of this is method is that it is

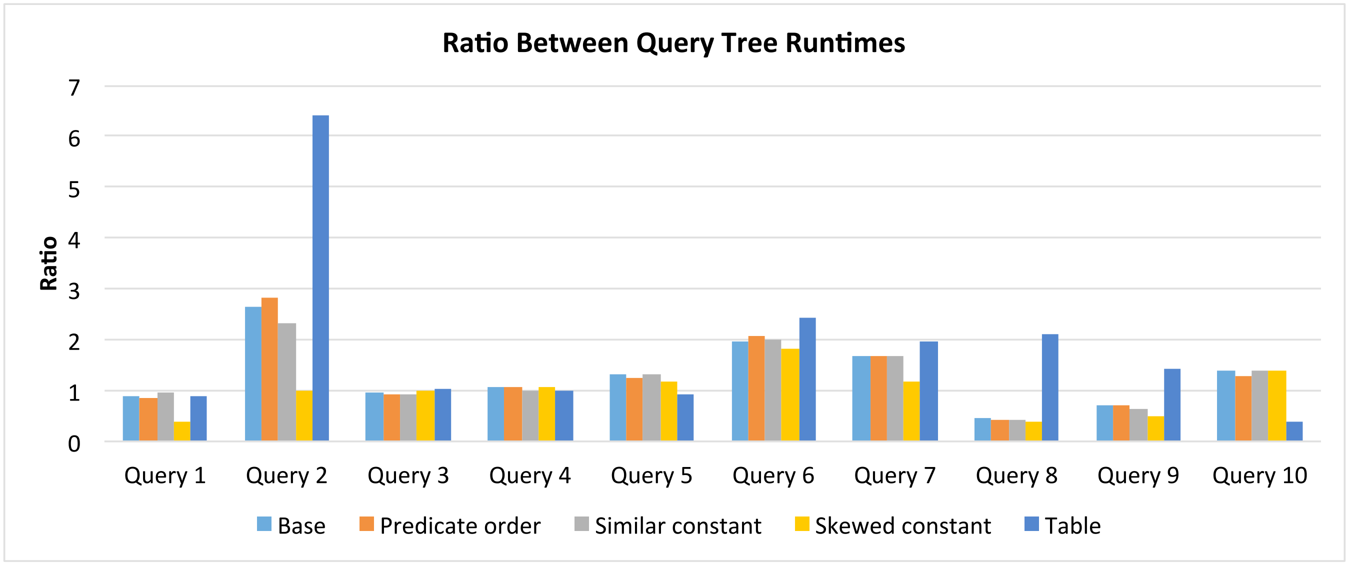

Fig. 3. Ten polystore queries applied to the MIMIC II dataset. The y-axis indicates the ratio between running a particular query with two different query plans. Ideally, the ratio of runtimes should be nearly the same for a similar query and query type.

possible that the Monitor can provide useful metrics even for queries that differ on that signature.

Another problem with the Monitor is that it is possible for the Monitor to always provide outdated metrics. In order to deal with this problem, it might be a good idea to select old queries to run in a different order than least recently updated. Instead, one could try choosing old queries to run in a randomized manner, weighted by time since last updated. More testing needs to be done to determine the best order to select queries to re-run.

Overall, the Monitor matches queries where constants have been replaced or when the order of predicates of the queries are changed. Based on my results, for all of these matched queries, the Monitor provides useful metrics as long as the initial benchmark queries run in non-trivial amounts of time. For queries where a table has been replaced, the Monitor cannot reliable provide useful metrics and thus does not match such queries.

Acknowledgment

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

References

[1] M. Saeed, M. Villarroel, A. T. Reisner, G. Clifford, L.-W. Lehman, G. Moody, T. Heldt, T. H. Kyaw, B. Moody, and R. G. Mark, "Multiparameter intelligent monitoring in intensive care ii (mimic-ii): a public-access intensive care unit database," *Critical care medicine*, vol. 39, no. 5, p. 952, 2011.

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

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

[4] M. Stonebraker, P. Brown, D. Zhang, and J. Becla, "Scidb: A database management system for applications with complex analytics," *Computing in Science Engineering*, vol. 15, no. 3, pp. 54–62, May 2013.

[5] J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Hubbell, P. Michaleas, J. Mullen, A. Prout, A. Reuther, A. Rosa, and C. Yee, "Achieving 100,000,000 database inserts per second using accumulo and d4m," in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, Sept 2014, pp. 1–6.

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

[7] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker *et al.*, "Demonstration of the myria big data management service," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 881–884.

[8] V. Gadepally, J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, L. Edwards, M. Hubbell, P. Michaleas, J. Mullen, A. Prout, A. Rosa, C. Yee, and A. Reuther, "D4m: Bringing associative arrays to database engines," in *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, Sept 2015, pp. 1–6.

[9] J. Kepner, W. Arcand, W. Bergeron, N. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz, A. McCabe, P. Michaleas, A. Prout, A. Reuther, A. Rosa, and C. Yee, "Dynamic distributed dimensional data model (d4m) database and computation system," in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, March 2012, pp. 5349–5352.

[10] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 1998, pp. 34–43.

[11] M. Pawlik and N. Augsten, "Rted: A robust algorithm for the tree edit distance," in *Proceedings of the VLDB Endowment (PVLDB), Vol. 5, No. 4*, December 2011, pp. 334–345.

[12] G. D. Clifford, D. J. Scott, and M. Villarroel, "User guide and documentation for the mimic ii database," *MIMIC-II database version*, vol. 2, p. 95, 2009.