Cross-Engine Query Execution in Federated Database Systems

Ankush M. Gupta*, Vijay Gadepally*†, Michael Stonebraker*

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

***Abstract*—We have developed a reference implementation of the BigDAWG system: a new architecture for future Big Data applications, guided by the philosophy that "one size does not fit all". Such applications not only call for large-scale analytics, but also for real-time streaming support, smaller analytics at interactive speeds, data visualization, and cross-storage-system queries. The importance and effectiveness of such a system has been demonstrated in a hospital application using data from an intensive care unit (ICU). In this article, we describe the implementation and evaluation of the cross-system Query Executor. In particular, we focus on cross-engine shuffle joins within the BigDAWG system, and evaluate various strategies of computing them when faced with varying degrees of data skew.**

I. Introduction

In the recent past, the database community has produced a wide variety of data models and data management systems. Many of these new systems have targeted vertically scaled SQL engines such as those that rely on in-memory online transaction processing (OLTP) workloads [1], [2]. In parallel, there has been an explosion of NoSQL engines exploiting a range of data models (typically operating on flexible storage formats such as JSON or key-value pairs). OLTP and stream processing engines are poised to satisfy the need for real-time stream processing and analytics for domains such as the internet of things. Specialized data management systems are often orders of magnitude faster for their specialized data models than other database systems, these systems exemplify the adage that "one size does not fit all" for data management solutions [3].

It is now common to see applications that must take advantage of disparate database management systems to support complex analytics and applications. However, this often requires the construction of one-off connectors and shims to enable such analytics. Recently, we have developed the Big Data Analytics Working Group (BigDAWG) polystore system [4], [5] to provide a single interface for disparate data models, database engines and programming models. The BigDAWG stack was applied to a a complex medical dataset - MIMIC II [6] and supports SciDB[7] to store MIMIC II time-series data, Apache Accumulo [8], [9], [10] to store freeform text notes, and Postgres[11] to store clinical data.

Complex datasets and analytics that touch independent database engines requires support for cross-engine queries. These independent systems with correlated data. For example, in our reference implementation, to locate the diagnoses of patients with irregular heart rhythms, one will need to extract data from both SciDB and Postgres. To find doctor's notes associated with a prescription drug, one would query Accumulo and Postgres. Such cross-engine queries rely on complex query planning, execution strategies and data migration.

In this article, we describe the process in which cross-engine queries are evaluated in the BigDAWG system. Specifically, we design a multi-step framework for performing "shuffle joins" – joins where data is migrated to multiple different engines and computed in parallel. We then implement and evaluate various methods for completing each step of the shuffle join framework, particularly focusing on handling varying degrees of skewness in the distribution of data.

A. BigDawg System Modules

The BigDAWG system has four core modules: the Query Planner, the Performance Monitor, the Data Migration and the Query Executor (see [12], [13], [14], [15] for a more thorough exploration of these modules and their interfaces). In this article, we concentrate on the query Executor.

The Executor is responsible for performing the physical execution of a logical query plan as provided by the Planner module. The planner module provides the Executor with a list of nodes and tasks that must be performed and their dependencies. The Executor performs as many nodes of the logical plan in parallel on different engines as possible, blocking on nodes with unfulfilled dependencies until they can be executed. Currently, we special operations such as the Union operator and Join operator are handled by the Executor.

This article concentrates on the equijoin operation (for simplicity, we refer to this simply as "join") performed across different engines within an Island of the BigDAWG polystore system. Currently, the Planner module only provides the Executor with binary joins, however the approach and technique described in this article can scale to any number of engines.

Since parallel joins are vulnerable to the presence of skew in the underlying data, we also discuss our approach to model the distribution of skew and take advantage of skew when possible. The most efficient way to achieve fast, parallelizable join execution of large tables across multiple engines is for the matching tuples of the two inputs to always be hosted on the same engine. This allows for parallel tuple comparison where each engine joins local data. Specifically, we extend prior work

on skew performance mitigation [16] to generalize approaches for shuffle-joins across worker nodes within SciDB.

II. Shuffle Join Framework

Given a shuffle join operation, the BigDAWG Planner determines a logical plan for evaluating a query and passes this to the Executor. The Executor performs all nodes of the logical query plan received in parallel, blocking until any necessary dependency nodes have been completed. For nodes which translate directly into queries on a single engine, this is done by delegating execution to the engine itself. Executing nodes that represent cross-engine operations, however, takes place in several phases, moving data between engines by leveraging the data migrator. This section details the execution pipeline.

A. Execution Pipeline

The shuffle join optimization framework assigns *join-units*, small non-overlapping ranges of tuples (rows in PostgreSQL, cells in SciDB, key-value pairs in Accumulo, etc.), to participating engines in order to efficiently use network bandwidth and balance the tuple comparison load. Each join-unit consists of a fraction of the full query predicate, and tuples are assigned to a join-unit based on the value of their join attribute. Tuples belonging to a single join-unit may be distributed over any number of engines participating in the join, but must be brought to the same engine in order for the join to be computed. The following steps are used when executing a query:

1) **Skew Examination:** The distribution of the join attributes on each participating engine is extracted. In some join strategies (detailed in Section II-C), this step may be skipped altogether.
2) **Join-Unit Assignment** The distribution of attributes is utilized to assign tuples to be joined on specific participating engines.
3) **Join-Unit Colocation** The join units are physically migrated to the engines to which they are assigned. This phase relies on the Migrator [14] to migrate portions of the join tables in parallel.
4) **Tuple Comparison** The Join operation is performed as parallel local queries with the tuples that have been migrated to their assigned engines.
5) **Join Result Union** The outputs of each participating engine's tuple comparison step are migrated and Unioned at the single end-destination engine specified in the logical query plan.

The total execution time of a cross engine join is largely determined by the time taken to perform the above steps. The Executor delegates the Join-Unit Colocation phase and Join Result Union phases to the Migrator module, and delegates the Tuple Comparison to a given database engine. For the Executor, a majority of the time is taken in determing the skew distribution and join-unit assignments. We detail these two phases in the following sections.

B. Skew Examination Strategies

The Skew Examination phase is the first step of the shuffle join execution. In this phase, the Executor retrieves information about the data distribution of each engine involved in the join which can be utilized in determinig how join-units are assigned and migrated in future stages of the shuffle join. In this phase, a histogram detailing the distribution of the join attribute for each table is either created or extracted from the internal statistics utilized by each engine.

A skew examination strategy consists of a function for each distinct database engine that retrieves the following types of data:

> **Histogram data** consists of a series of histogram buckets. The number of buckets is a specified as a constant, and remains the same for all histograms within the same join operation. The bounds for each bucket is determined by utilizing the minimum and maximum values the Executor stores after evaluating the plan nodes that created the tables being joined, or are retrieved from the engine if the tables already existed prior to the execution of the query plan. The value for each bucket is the count of elements on the engine whose join attribute falls within the given range.
>
> **Hotspot data** consists of data that is of much finer granularity than a histogram bucket. A hotspot is a single value of a join attribute that has a particularly high occurrence in a given engine. The value associated with each hotspot value is the count of items that have an identical value for their join attribute. Hotspot data is only utilized if an statistics are extracted from an engine's internal planner, and if that engine's planner provides this information.

A skew examination returns histogram data, but not necessarily hotspot data. The resulting information is then used in the next step, the tuple assignment.

In the assignment strategies and analysis presented in this article, both histogram buckets and hotspot values are treated as join-units by the shuffle join planner. If there exists a hotspot value that is also within the range contained by a histogram bucket, that histogram bucket is treated as if it does not contain the hotspot, and the hotspot is treated as if it were a separate bucket.

1) Full Table Scan: The full table scan strategy involves scanning every element involved in the join, for every participating table, constructing a completely accurate histogram of the distribution of join attributes on each engine.

This strategy allows for perfectly accurate histogram data, but comes at the expense of slower performance. Depending on the circumstances, the overhead from the increased time in skew examination may result in better overall performance if it allows for join units to be more efficiently allocated.

2) Table Sampling: An alternative approach to generating distribution information is to sample the data in the participating join tables rather than performing a full table scan. In this

method, a fixed number or proportion of tuples are sampled at random from each table, and a histogram is populated with the results. The histogram bucket counts are then rescaled with respect to the total number of tuples on the local engine. Wherever possible, we utilize any sampling statistics provided by the engine's built-in query planner instead of sampling the table by performing a query.

C. Join-Unit Assignment Strategies

Once information on the skew of each participating table has been collected, the Executor must determine how to best allocate each join-unit represented in the tables to specific engines. The strategy of assigning of tuples to participating engines can have large performance implications on the runtime of each phase of the shuffle join framework (listed in Section II-A). Some assignment approaches can be more computationally intensive depending on the characteristics of the underlying data to be joined. In cases of simple assignment strategies, coarser skew examination strategies that lead to large performance benefits may be utilized.

Another impact of the join-unit assignment comes from migrating tuples between engines. We have found that the network is often a scarce resource for joins in a shared-nothing architecture [17]. In addition, there is often a significant amount of overhead associated with utilizing the Migrator module, even for relatively small datasets. Because computation of a join at an engine is blocked until all tuples assigned to that engine have successfully been migrated, the number of migrations that an assignment of join-units requires has direct impact on the overall runtime of the Join. Assignments which require migrating fewer tuples can begin the Tuple Comparison phase much more quickly than those which involve migrating a considerably larger amount of tuples.

A final impact of join-unit assignment on overall performance is in the runtime of the Tuple-Comparison Phase. The more tuples that are assigned to a specific participating engine, the longer that engine will take to complete its tuple-comparison phase. Though tuple comparison phase is parallelized across all participating engines, the total runtime of the join is bottlenecked by the engine which takes the longest to complete its Tuple Comparison phase and send its results to the destination specified by the Planner module.

In the following sections, we look at potential Join-Unit assignments and their tradeoffs with respect to the above factors.

1) Full Table Assignment: This category of assignment can be viewed as treating an entire table as a join-unit. This allows assignments to be performed with minimal skew examination. By doing so, runtime savings can be achieved by limiting the total number of migrations involved in the join process.

These assignment methods come with the tradeoff of potentially migrating excess data in the form of tuples that have no matching values with which to join at their destination engine. Though probabilistic data structures such as Bloom Filters can be used to mitigate this factor. in our current implementation we have not addressed this.

a) Destination Full Broadcast: This type of Full Table Assignment moves all data to the engine specified by the logical plan generated by the Planner, regardless of the size of the participating tables. The join is then computed locally on this destination engine. This assignment strategy eliminates the need for the Skew Examination and Join Result Union phase. Additionally, the simplicity of this assignment strategy allows this assignment to be computed efficiently and quickly.

This assignment strategy may perform slowly when non-destination tables are excessively large and skewed, or when the destination engine performs the join at a significantly slower pace than alternative engines. In these cases, is possible that the cost of migrating unnecessary tuples to a single potentially slower engine dominates the savings from not needing to perform a finer-grain skew examination and corresponding assignment strategy.

b) Minimal Full Broadcast: This planning model moves smaller tables involved in the join to the engine of the largest table. The skew examination phase for this assignment strategy treats the universe of all potential join attribute values as a single join-unit, eliminating the skew examination phase in favor of a simple count operation.

Compared to the Destination Full Broadcast assignment strategy, the Minimum Full Broadcast avoids situations where a single large table dominates migration times. The downside to this approach is that the computed join result may need to be migrated to the destination engine specified in the Planner's logical plan.

2) Join-Attribute Assignment: Based on join attributes, there may be many join-unit assignemtn strategies. Specifically, we discuss the skew-agnostic Hash Assignment, and the skew-aware Minimum Bandwidth Heuristic and Tabu Search assignments.

The skew-aware strategies rely on data provided by the Skew Examination phase to make better informed assignments of the data being joined.

Though join-attribute assignments can result in more balanced migration and execution times, it is important to note that every participating engine could have migrations both to and from other participating engines, resulting in $O(n^2)$ total simultaneous migrations with respect to the number of engines.

a) Hash Assignment: The Hash Assignment strategy is a skew-agnostic strategy. It randomly assigns tuples to each participating engine in the join such that all tuples with the same join attribute value end up on the same engine. It does so by simply hashing each tuple's join attribute, and mapping it to an engine using a common hash function $H(x)$ shared across all participating engines.

For a given tuple, t, having join attribute a_t over k engines, the Executor determines the engine n to which the tuple is assigned by: $n = H(a_t) \mod k$

This join strategy does not require the Skew Examination phase to take place. If the data has a uniform distribution across all participating engines, the Hash assignment strategy has a balanced number of migrations between participating engines in expectation.

b) Minimum Bandwidth Heuristic: The first skew-aware assignment strategy we explore is the the Minimum Bandwidth Heuristic (MBH), adopted from SciDB [16]. The MBH assumes that the overhead of migration dominates the overall runtime, and attempts to optimize runtime by greedily assigning join-units to the engine that already possesses the majority of the tuples for that join-unit. We refer to the chosen engine for each join unit as the unit's *center of gravity*.

For a given join unit, i, having frequency $\{f_{i,1}, f_{i,2}, \ldots, f_{i,k}\}$ over k engines, the Executor identifies the engine n with the most tuples from join unit i:

$$n = \arg\max_{e=[1,k]} f_{i,e}$$

and assigns the join unit to engine n.

This heuristic minimizes the amount of data transmitted in the Join-Unit Colocation phase. In doing so, it attempts to complete the Join-Unit Colocation phase as quickly as possible, so that the parallelized tuple comparison process can begin on each participating engine, unbound by network constraints.

c) Tabu Search: The final join-unit assignment strategy detailed in this article is based on a variant of the Tabu Search [18] proposed for SciDB skew-aware join handling [16]. This assignment strategy is skew-aware, and begins with the assignment provided by the MBH approach. It then searches for a locally optimal result by exploring engines with a higher-than-average cost associated with them, and attempting to reassign their join-units to engines with lower cost. Once a join-unit has been assigned to a given engine, the unit-to-engine pair is noted in the Tabu list which ensures that this assignment is never considered again. The Tabu search completes once it is impossible to improve the plan by reducing the load of the engines. Pseudocode for the modified Tabu Search algorithm utilized by SciDB and implemented in the BigDAWG executor for join-unit assignment is shown in Algorithm 1.

The approach of formulating the Tabu List as a join-unit to engine mapping is borrowed from SciDB's shuffle join handling. This strategy significantly reduces the size of the sample space from the exponential $O(2^{i*j})$ to the polynomial, $O(i * j)$, where i is the total number of join-units and j is the total number of engines participating in the shuffle join. Because the algorithm unburdens a single engine at a time, the bottleneck engine is likely to change during a single round of unburdening. This formulation of the Tabu List also prevents the algorithm from getting stuck in loops, wherein a single join-unit is cyclically reassigned between multiple non-bottleneck nodes.

Cost Estimation: The nature of Tabu Search necessitates a mechanism of evaluating the cost of a given query plan. The primary functions that the cost estimation should consider include the cost of migrating join units to a given engine, and the cost of comparing the migrated tuples once the migrations have completed.

Since the rate at which each distinct engine completes these steps varies between engines, a cost estimation mechanism that is able to handle varying costs per engine is needed. The approach we utilize is given in Algorithm 1. The cost for a specific engine is determined by the sum of the most expensive inbound migration and the cost of comparing all tuples being joined on the given engine. Both costs are modeled as a simple quadratic function of the number of tuples.

Algorithm 1 Tabu Search per-engine costs

function COMPUTEENGINECOSTS(A)
Input: A ▷ Current assignments
Output: C ▷ Map of engines to per-engine costs
  $A \leftarrow \{\}$
  for all $j \in A$.engines **do**
    A.put(COMPUTESINGLEENGINECOST(j, A))
  return A

function COMPUTESINGLEENGINECOST(j, A)
Input: j, A ▷ Desired engine, Current assignments
Output: cost ▷ Cost associated with engine j
  maxMigrationCost $\leftarrow 0$
  n $\leftarrow 0$
  for all $j' \in A$.engines **do**
    $m \leftarrow A$.getAssignedTuples(j', j).count()
    $(c_1, c_2) \leftarrow$ MigrationConstants.get(j, j')
    maxMigrationCost $\leftarrow \max(c_1 * m^2 + c_2 * m,$ maxMigrationCost)
    $n \leftarrow n + m$
  $(c_3, c_4) \leftarrow$ComparisonConstants.get(j)
  tupleComparisonCost $\leftarrow c_3 * n^2 + c_4 * n$
  cost $\leftarrow$ maxMigrationCost + tupleComparisonCost
  return cost

As illustrated in Algorithm 2, the cost associated with the overall plan is simply equivalent to the maximum of all engines. This is because every engine must complete its local executions before the results can be be used as inputs to the Union operator and returned to the user.

Algorithm 2 Tabu Search total cost

function COMPUTETOTALCOST(A)
  costs $\leftarrow$ COMPUTEENGINECOSTS(A)
  return MAX(costs)

III. PERFORMANCE RESULTS

In this section, we analyze the performance of the Skew Examination strategies described in Section II-B and the Join-Unit Assignment strategies described in Section II-C.

A. Experimental Setup

For the purpose of this article, we concentrate on binary joins. However, the techniques outlined in previous sections can scale to any number of concurrent nodes. As a result, we perform our skew examination and join-unit assignment

evaluations on a system consisting of two virtualized PostgreSQL instances of different capacity. The larger engine has approximately 150GB of data and runs on four 3.4GHz processor cores with 32GB RAM, while the other instance has approximately 75GB of data with a single 3.4GHz processor core and 8GB RAM. Both systems use a switched network and SATA disks. We consider the the cross-engine Join workload of `SELECT * FROM A, B WHERE A.id = B.id`. A trial consists of this workload run once with the destination engine specified as the larger engine, and once with the the smaller. Each trial was started with a cold cache and executed five times. We report the average duaration of each phase of execution.

Our experiments use synthetic data sampled from a Zipf (power law) distribution to control the level of skew in the tables. This distribution's skew is characterized by the parameter α [19]. Higher values of α denote greater imbalance data distribution. The experiments begin with uniformly distributed data ($\alpha = 0$), where all join-units are the same size, and gradually increase the skew by 0.5 until reaching a maximum of $\alpha = 2.0$.

B. Skew Examination Strategy Evaluation

In order to compare the Full Table Scan and Sampling skew examination strategies, we utilize skew examination strategies to execute the workload by first using the Minimum Bandwidth Heuristic (MBH), and then using the Tabu Search assignment strategy. We then compare the differences in execution times for skew examination, and observe the impact of skew examination strategy on the performance of other phases involved in the shuffle join.

The benchmark results for both skew examination strategies with the Minimum Bandwidth Heuristic assignment are shown in Figure 1. The figures illustrate the total runtime of the benchmark query under different Zipf distributions (parametrized by α) and segmented by time taken for each phase. As expected, the portion of total runtime consumed by the Skew Examination phase (shown in black) is lower when using the sampling strategy rather than the full table scan.

We now attempt to determine if there is a significant difference in the quality of information generated by each strategy by examining their impact on the future phases of the overall join. As observed in Figure 1, when the skew of the data is low, the sampling strategy may result in less efficient assignment strategies. As skew increases, we notice that the post-examination phases have approximately equivalent runtimes. In almost all cases, the fact that Skew Examination under sampling is faster than under a full table scan more than makes up for any discrepancy between the later phases as a result of suboptimal join-unit assignments, resulting in lower runtimes while sampling. Similar results were found with the Tabu Search assignment strategy as well.

From these results, we conclude that the Sampling skew examination strategy is superior to the Full Table Scan strategy for the Skew Examination phase. Further, the total runtime is either comparable to or better than the Table Scan strategy

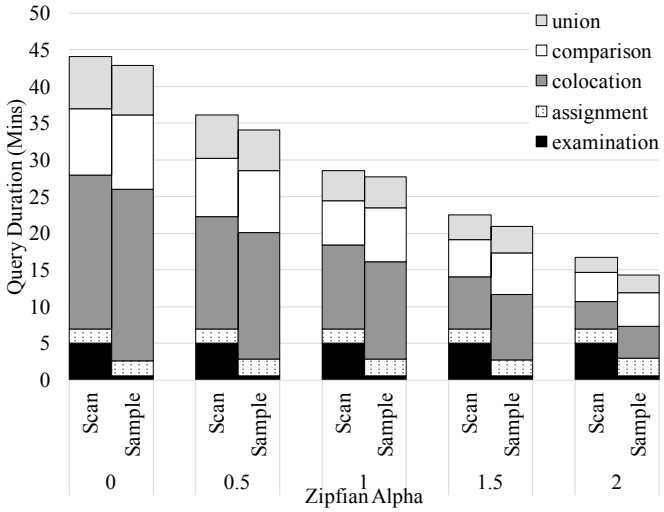

Fig. 1: Join performance with varying skew and skew examination strategies, using Minimum Bandwidth Heuristic assignment

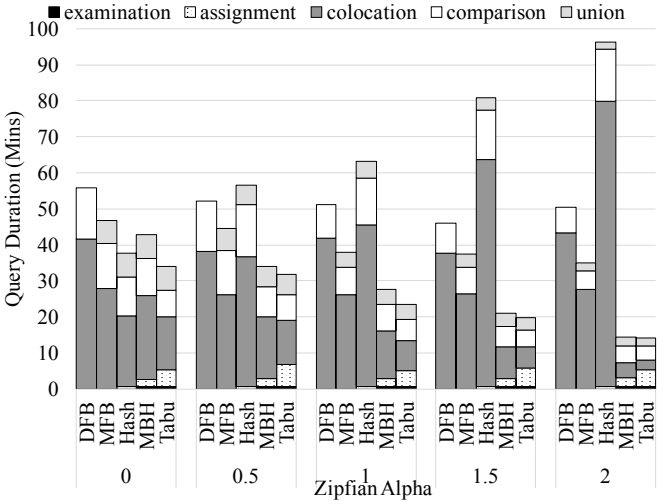

Fig. 2: Join with varying skew and assignment strategies for each of the five phases described in Section II-A.

when accounting for all phases involved in the shuffle join. This relative difference is more pronounced in high-skew workloads.

C. Join-Unit Assignment Strategy Evaluation

Next, we discuss the performance differences between all Join-Unit Assignment strategies described in II-C as the skew factor (parametrized by α) is increased. Since the sampling skew examination strategy outperforms the full table scan strategy, we concentrate on the Sampling strategy whenever skew examination is required. Figure 2 illustrates the performance of the plan generated by all assignment strategies given the experimental workload as the skew of the distribution varies.

From Figure 2, we observe that when $\alpha = 0$, the Hash strategy performs relatively well. It has tuple colocation performance virtually identical to the MBH strategy, due to the uniform distribution of the data. The only strategy that outperforms the Hash strategy at $\alpha = 0$ is the Tabu Search strategy, due to its ability to more effectively account for the fact that the different engines have differing performance in the tuple comparison phase.

We observe that as α increases, the time needed to migrate the data based on skew-agnostic assignments (DFB, MFB, and Hash) grows uncontrollably. Conversely, the MBH and Tabu strategies are able to leverage the skew in the data, and perform more efficient tuple assignments as the degree of skew increases. The Tabu approach can further utilize its cost estimation information to balance tuple comparison load across the two engines, and is competitive with the next-best assignment strategy with all ranges of skew.

IV. Conclusion and Future Work

In this article, we introduce a shuffle join Executor that is able to efficiently execute operations that span across multiple database engines. Initial results tested with cross-engine joins perform well in the presence of skewed data. The shuffle join methodology includes a multi-step process that exploits random sampling and effective join-unit assignment strategies to minimize the computational and network resources required to compute joins. The Executor can conduct the necessary migrations of data across participating engines, and execute all necessary operators for exploiting any available skew in the data in order to efficiently compute Join results. Empirical results indicate that this framework can consistently achieve significant performance improvements compared to skew-agnostic approaches in the presence of skewed data, while suffering no significant performance difference compared to skew-agnostic methods when utilized over non-skewed, uniform distributions.

Future work for improving the shuffle join framework in BigDAWG include utilizing probablistic data structures to minimize data transfer, determining the constants used in the Tabu Search cost model by interfacing with the Monitor, and abstracting the strategies to function across islands in the BigDAWG system.

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, Peinan Chen, and Sam Madden for their help in developing this work.

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '02. New York, NY, USA: ACM, 2002, pp. 1–16. [Online]. Available: http://doi.acm.org/10.1145/543613.543615

[2] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-store: A high-performance, distributed main memory transaction processing system," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1496–1499, Aug. 2008. [Online]. Available: http://dx.doi.org/10.14778/1454159.1454211

[3] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik, "One size fits all? part 2: Benchmarking results," *Proc. CIDR*, 2007.

[4] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, S. Madden, D. Maier, T. Mattson, S. Papadopoulos, J. Parkhurst, N. Tatbul, M. Vartak, and S. Zdonik, "A demonstration of the bigdawg polystore system," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1908–1911, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824098

[5] 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 Rec.*, vol. 44, no. 2, pp. 11–16, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2814710.2814713

[6] M. Saeed, M. Villarroel, A. T. Reisner, G. Clifford, L.-W. Lehman, G. Moody, T. Heldt, T. H. Kyaw, B. Moody, and R. G. Mark, "Multiparameter intelligent monitoring in intensive care ii (mimic-ii): A public-access intensive care unit database," *Critical care medicine*, vol. 39, no. 5, pp. 952–960, 05 2011. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3124312/

[7] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman, "The architecture of scidb," in *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management*, ser. SSDBM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032397.2032399

[8] "Accumulo," https://accumulo.apache.org/.

[9] 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.

[10] 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.

[11] M. Stonebraker and G. Kemnitz, "The postgres next generation database management system," *Commun. ACM*, vol. 34, no. 10, pp. 78–92, Oct. 1991. [Online]. Available: http://doi.acm.org/10.1145/125223.125262

[12] V. Gadepally, P. Chen, J. Duggan, A. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker, "The bigdawg polystore system," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, Submitted.

[13] S. Zuohao and J. Duggan, "Bigdawg polystore query optimization through semantic equivalences," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, Submitted.

[14] A. Dziedzic, A. Elmore, and M. Stonebraker, "Data transformation and migration in polystores," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, Submitted.

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

[16] J. Duggan, O. Papaemmanouil, L. Battle, and M. Stonebraker, "Skew-aware join optimization for array databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 123–135. [Online]. Available: http://doi.acm.org/10.1145/2723372.2723709

[17] M. Mehta and D. J. DeWitt, "Data placement in shared-nothing parallel database systems," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 6, no. 1, pp. 53–72, 1997.

[18] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Comput. Oper. Res.*, vol. 13, no. 5, pp. 533–549, May 1986. [Online]. Available: http://dx.doi.org/10.1016/0305-0548(86)90048-1

[19] V. Gadepally and J. Kepner, "Using a power law distribution to describe big data," in *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*. IEEE, 2015, pp. 1–5.

[20] A. Gupta, V. Gadepally, and M. Stonebraker, "Cross-engine query execution in federated database systems," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, Submitted.
[21] J. K. Mullin, "Optimal semijoins for distributed database systems," *Software Engineering, IEEE Transactions on*, vol. 16, no. 5, pp. 558–560, 1990.
[22] C. J. Hursch and J. L. Hursch, *SQL: The Structured Query Language*. Blue Ridge Summit, PA, USA: TAB Books, 1988.
[23] P. G. Brown, "Overview of scidb: large scale array storage, processing and analysis," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 963–968.
[24] R. E. Burkard and E. Cela, *Linear assignment problems and extensions*. Springer, 1999.
[25] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear, "The vertica analytic database: C-store 7 years later," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1790–1801, 2012.