Cross-Engine Query Execution in Federated Database Systems

by

Ankush M. Gupta

S.B., Massachusetts Institute of Technology (2016)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author .....
Department of Electrical Engineering and Computer Science
May 18, 2016

Certified by .....
Michael Stonebraker
Adjunct Professor
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Cross-Engine Query Execution in Federated Database Systems

by

Ankush M. Gupta

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Electrical Engineering

Abstract

Duggan et al.[5] have created 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 report, we implement and evaluate a concrete version of a cross-system Query Executor and its interface with a cross-system Query Planner. In particular, we focus on cross-engine shuffle joins within the BigDAWG system.

Thesis Supervisor: Michael Stonebraker
Title: Adjunct Professor

Acknowledgments

I would like to express my gratitude to my Thesis Advisor, Professor Michael Stonebraker, for his guidance, support, and patience with my project this past year. I would also like to thank my Research Supervisor, Vijay Gadepally, for his support and insight.

Finally, I would like to thank Jennie Duggan, Aaron Elmore, Adam Dziedzic, Jack She, and Peinan Chen (along with Mike, Vijay, and the rest of the BigDAWG team), whose efforts on BigDAWG provided the framework upon which I was able to complete my research.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Algorithms

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

The recent trend of development within the database community has been towards a proliferation of data models and data management systems, many targeted for a well-defined vertical market [2, 10]. Relational column stores are poised to take over the data warehouse market. High-throughput OLTP workloads are largely moving to main-memory SQL systems. There has been an explosion of NoSQL engines exploiting a range of data models (typically operating on flexible storage formats such as JSON). OLTP and stream processing engines are poised to satisfy the need for real-time stream processing and analytics for the internet of things. Because they are orders of magnitude faster for their specialized data models than other database systems, these specialized systems exemplify the adage that "one size does not fit all" for data management solutions[15].

It is not uncommon to see applications that take advantage of more than one of these systems to support their complex workloads. For example, such an infrastructure can be designed around MIMIC II[14], a publicly available dataset covering 26,000 intensive care unit admissions at Boston's Beth Israel Deaconess Hospital. This dataset includes waveform data (up to 125 Hz measurements from bedside devices), patient demographic information (name, age, etc.), doctor's and nurse's notes (raw text), clinic reports and lab results, and prescriptions filled (semi-structured data). In a real-world system, it would be prudent to split the waveform data even further into real-time streams and historic archive data, for both storage and compression

reasons, as well as the ability to have precomputed relevant analytics for faster access.

Given the variety of data sources, this system must support a cornucopia of data types and access methods – ranging from standard SQL (e.g., how many patients were given a particular drug), to complex analytics (e.g., computing the FFT of a patient's realtime waveform data and comparing it to the historic "normal"), and even text search (e.g., finding patients who responded well to a particular treatment).

Our reference implementation of the system uses SciDB[16] to store MIMIC II's time series data, Accumulo[1] to store text, and Postgres[17] to store patient information. The nature of these independent systems with correlated data implies that there will be queries that span two or more storage engines. For example, to locate the diagnoses of patients with irregular heart rhythms, one would query SciDB and Postgres. To find doctor's notes associated with a prescription drug, one would query Accumulo and Postgres. To run analytics on particular cohorts of patients, one would query Postgres and SciDB.

It is clear that a system designed to coordinate several distinct database engines will be required in a myriad of future applications, especially those that are broad in scope like MIMIC II. The concept of such a polystore federation has been discussed in the BigDAWG project[7, 5], and a concrete implementation of the BigDAWG system's Query Executor module is detailed in this report.

Chapter 2

Background

In this chapter, we introduce the basic concepts and structure of the BigDAWG system. We begin with an overview of its data storage and representation model. Next, we will explore an overview of the modules that allow BigDAWG to operate on these underlying data models.

The chapter concludes with a look at the challenges and opportunities presented by skew when attempting to execute queries on the underlying data.

2.1 BigDAWG System Architecture

The BigDAWG system is designed to offer users an interface that is agnostic to the location of data, so that users are able to interface with the underlying database engines while not being hampered by the specifics of which data is located on which engine. BigDAWG utilizes the concept of *islands of information* to support this idea. Each island serves as an abstraction for a federated system consisting of multiple database engines. Islands include a query language, data model, and mechanisms for interacting with the federated storage engines. Users are able to run queries against islands without requiring any knowledge of the underlying engines, and BigDAWG is responsible for performing them seamlessly. Users can also write queries that explicitly convert data from one Island to another, and BigDAWG handles the conversion as necessary.

2.1.1 System Modules

The BigDAWG system has four core modules: the Query Planner, the Performance Monitor, the Data Migration and the Query Executor (see Figure 2-1 for the organization and data flow of the system).

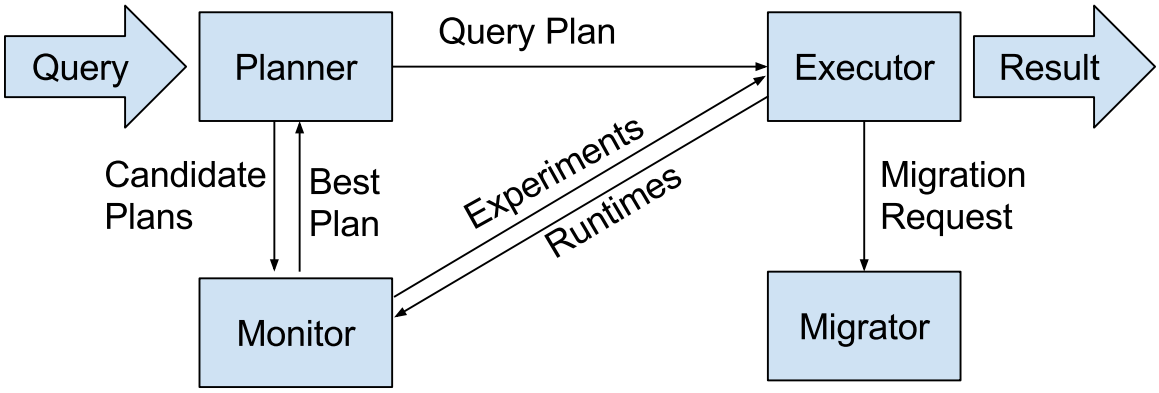

Figure 2-1: Data flow across all modules of BigDAWG

Planner

The Planner module of BigDAWG is responsible for parsing a user's input and constructing a logical query plan in the form of a dependency tree of action nodes. Actions in the logical plan include querying a specific engine, performing a Join operation, and performing a Union operation. The logical plan additionally specifies an end-destination (in the form of a table name and an engine) that designates where the query results should be stored.

The Planner generates multiple possible logical plans and interfaces with the Monitor module in order to determine which of the candidate logical query plans should be run by the Executor module.

Monitor

The Monitor module logs the execution times of logical query plans as they are performed by the Executor. It utilizes a query signature system to relate new plans to previously executed ones, and uses these relations in order to determine which of

the provided logical plan is the most efficient.

When the BigDAWG system is not under high load, the Monitor can also call the Executor directly with logical plans of its own in order to train its internal model.

Migrator

The Migrator module is responsible for migrating data between database engines. The Migrator can handle both intra-Island as well as inter-Island migrations, but inter-Island migrations may require additional parameters depending on the data models of the relevant Islands. The Migrator module is called by the Executor when needed to complete a logical query plan.

Executor

The Executor is responsible for performing the physical execution of a logical query plan as provided by the Planner module. The Executor performs as many nodes of the logical plan in parallel as possible, blocking on nodes with unfulfilled dependencies until they can be executed. When a node creates a table, the Executor also queries the database for the total row count, minimum, and maximum values in the resulting table. Special operations handled by the Executor include the Union operator (which is a trivial concatenation of the results of its dependencies) and the Join operator.

In this report, we provide a detailed overview and analysis of the Join operation when performed across engines within an Island in the BigDAWG system, with a specific focus on equijoin operation (henceforth referred to as simply a join)

Though the approaches implemented in the Executor module scale to any number of participating engines, the status of the Planner module at the time of writing does not present the Executor with opportunities to execute a single join across more than two engines even when the join predicate is the same. In the case of a complex join with a single predicate that spans multiple engines, the Planner module creates a logical plan containing a cascading sequence of operations involving exactly two engines.

2.1.2 Supported Queries

The BigDAWG system currently supports subsets of the Structured Query Language[9] for engines in the Relational Island (and other engines that expose a SQL-compatible interface) and Array Functional Language[3] for engines in the Array Island. Queries that are executable on a single engine are fully supported by passing the query string to the engine itself. Inter-engine operations that are supported include basic Select, Filter, and Project operations; basic aggregate functions such as Sum and Count; and finally Join and Union operators.

2.2 Skew in Data Distribution

Traditional parallel joins are vulnerable to the presence of skew in the underlying data. If the underlying data is sufficiently skewed, imbalances in computation load and network transfer time will become the bottleneck in performance, outweighing any of the gains that parallelism provides. 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. Unfortunately, the fact that the BigDAWG system manages data separated across various engines makes data movement inevitable.

Prior work has been done by Duggan et al. in order to mitigate the negative impacts of skew – and in some cases utilize it to improve performance – through parallelized shuffle joins across worker nodes within the SciDB database engine[6]. The Query Executor module of BigDAWG attempts to generalize approaches explored for SciDB to cross-engine equijoins in BigDAWG to efficiently compute these joins, taking advantage of skew where possible.

Chapter 3

Shuffle Join Framework

Once the Planner has determined a logical plan for evaluating a query, the Query Executor performs all nodes of the logical query plan in parallel, blocking until any necessary dependency nodes have been completed. For nodes which translate directly into queries on single engines, this is trivially done by delegating execution to the engine itself. Executing nodes that represent cross-engine join operations however, takes place in several phases, involving moving data between engines.

In this chapter, we provide an overview of the process by which cross-engine joins are performed in the BigDAWG Executor, through a pipeline called the shuffle join optimization framework.

3.1 Execution Pipeline

Our shuffle join optimization framework assigns *join-units*, or small non-overlapping ranges of tuples, to participating engines in order to make efficient use of network bandwidth and to balance the tuple comparison load. These join-units are typically represented by buckets in a histogram. Each join-unit consists of a fraction of the predicate space, 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 all 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 the phases of the shuffle join framework:

1. **Skew Examination** is the first phase, in which the distribution of the join attributes on each participating engine is extracted. In some join strategies (detailed in Chapter 5, this step may be skipped altogether.

2. **Join-Unit Assignment** is the next phase, in which the distribution of attributes is utilized to assign tuples to be joined to specific participating engines.

3. **Join-Unit Colocation** is the phase in which the join units are physically migrated to the engines to which they are assigned. This phase relies on the Migrator (see Figure 2-1) to move portions of the join tables in parallel.

4. **Tuple Comparison** is the phase in which the Join operation is performed as parallel local queries with the tuples that have been migrated to their assigned engines.

5. **Join Result Union** is the final phase, where 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 speed with which the Executor is able to complete all of the above phases determines the total execution time of a cross-engine join. 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.

It is in the Skew Examination and Join Unit Assignment phases that the Executor makes decisions which are critical in optimizing the performance of the Join operation in the face of potentially skewed data. In the remainder of this report, we detail and evaluate various strategies for completing these two phases of the Shuffle Join Framework.

Chapter 4

Skew Examination

In this chapter, we begin by describing the Skew Examination phase in detail. We follow by describing two particular strategies that were implemented in order to extract information about the degree of skew within a particular table: full table scans and table sampling. The sampling strategy is further broken down into sampling performed by the Executor, and sampling performed by the engine itself.

4.1 Overview

The Skew Examination phase is the first step in shuffle join execution. In this phase, the Executor retrieves information about the data distribution of each engine involved in the join. This information can then be utilized by join-unit assignment strategies to make informed decisions about how join-units should be 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 retrieves after evaluating the plan nodes that created the tables being joined, or are retrieved by querying 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 statistics are extracted from an engine's internal planner, and if that planner provides this information.

A skew examination function must return histogram data, but may not necessarily return hotspot data. In the assignment strategies and analysis presented in this report, 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.

The resulting information is then used in the tuple assignment phase.

4.2 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.

4.3 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.

Depending on the schema of the table (particularly indexes and table sorting), randomly accessing rows may prove to be costly, even compared with the full table scan approach. This does not prove to be a problem with workloads which join on results of subqueries however, because the intermediate tables created by the Planner have no indexes or sorting parameters.

4.3.1 Engine-collected Statistics

In many cases[17, 11, 1], the database engine itself performs a sampling routine for its internal statistics, which are often utilized by the engine's query optimizer. Several such databases (particularly relational databases) also record hotspots in their statistics.

Where possible, utilizing the engine-provided statistics is preferred to performing a query to sample the tables. Not only does this ensure that the local engine has up-to-date statistics for performing the join on the data in the tables, but empirically, we have seen that updating and extracting engine analytics is faster than than performing queries for random sampling. In some cases, the histogram data extracted from the engine is not of the desired bucket size. In these cases, the buckets are resized assuming uniform distribution within buckets.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Join-Unit Assignment

In this chapter, we start by detailing the considerations which must be taken into account when determining how to assign join-units to engines. We continue by detailing join-assignment strategies in two categories: those which assign entire tables to engines, and those which assign tuples to engines depending on their join-attribute values.

5.1 Overview

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 3.1).

Some assignment approaches will be more computationally intensive than others, and can take a longer time to compute, depending on the characteristics of the underlying data to be joined. In cases of simpler assignment strategies, coarser skew examination strategies may be utilized, leading to a performance increase in the Skew Examination phase.

Another impact of join-unit assignment comes from the fact that migration of tuples between engines is far from an instantaneous process. The network is typically

the most scarce resource for joins in a shared-nothing architecture[12]. In addition, there is a nontrivial amount of overhead associated with utilizing the Migrator module for even a small amount of data. 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 a direct impact on the overall runtime of the Join. Assignments which require migrating fewer tuples can begin the Tuple Comparison phase more quickly than those which involve migrating a 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 the 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, various potential Join-Unit Assignment strategies are explored, and their tradeoffs with respect to the above factors are discussed and evaluated.

5.2 Full Table Assignment

In this section, we detail two join-unit assignment strategies that assign entire tables to engines, without grouping any tuples by their join attributes.

This category of assignment can be viewed as treating the entirety of a table as a join-unit, which allows these assignments to be performed with a much coarser degree of skew examination (if any at all). By doing so, runtime savings are also achieved by limiting the total number of migrations involved in the join process.

These assignment methods comes 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 (discussed in Section 7.2), the current implementation of these strategies does not do so.

5.2.1 Destination Full Broadcast

This type of Full Table Assignment simply moves all data to the end-destination engine specified by the logical plan generated by the Planner (see Section 2.1.1), regardless of the size of the participating tables. The join is then computed locally on the end 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, it 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.

5.2.2 Minimal Full Broadcast

This planning model moves the smaller tables to the engine of the largest table. The skew examination phase for this assignment strategy can treat 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 can avoid situations where a single large table dominates the 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.

5.3 Join-Attribute Assignment

In this section, we detail three join-unit assignment strategies that assign tuples to engines based on the values of their join attributes. 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, where n is the number of engines participating in the shuffle join.

5.3.1 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 n engines, the Executor determines the engine e to which the tuple is assigned by:

$$e = H(a_t) \mod n$$

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 is expected to have a balanced number of migrations between participating engines.

The Hash Assignment strategy does not, however, take into account any potential imbalances between tuple comparison performance on participating engines, and also

can result in very inefficient allocations of tuples when faced with larger degrees of skew in the data.

5.3.2 Minimum Bandwidth Heuristic

The first skew-aware assignment strategy we explore is the the Minimum Bandwidth Heuristic, adopted from SciDB[6]. 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 call the chosen engine for each join unit the unit's "center of gravity".

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

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

and assigns the join unit to engine e.

This heuristic provably 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, where network constraints do not impact runtime.

Though the MBH minimizes the migrations involved in the Join-Unit Colocation phase, this assignment strategy does nothing to address unequal load in the Tuple Comparison phase, and does not attempt to perform any optimization based on the final size of the join.

5.3.3 Tabu Search

The final join-unit assignment strategy detailed in this report is based on the variant of the Tabu Search[8] proposed for SciDB skew-aware join handling[6]. This assignment strategy is also skew-aware, and begins with the assignment provided by the Minimum Bandwidth Heuristic 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. The key insight of the Tabu Search is that once a join-unit has been assigned to a given engine for the first time, the unit-to-engine pair is noted in the "Tabu" list, ensuring that this assignment is never considered again. Once it is impossible to improve on the plan by reducing the load of the engines with high cost, the Tabu Search completes.

Algorithm 1 Tabu Search

```
function TabuSearch(E)
Input: E                                          ▷ Skew Examination results
Output: A                                         ▷ Join-Unit to Engine assignments
    tabuList ← ∅
    A ← MinimumBandwidthAssignment(E)
    tabuList.insert(A)
    engineCosts ← ComputeEngineCosts(A)
    A′ ← null
    while A ≠ A′ do
        A′ ← A
        for all j ∈ E.engines do
            if engineCosts.get(j) ≥ mean(engineCosts) then
                A ← UnburdenEngine(j, A)
        engineCosts ← ComputeEngineCosts(A)
    return A
```

Algorithm 2 Engine Unburdening

```
function UnburdenEngine(j, A, tabuList)
Input: j, A, tabuList          ▷ Engine to unburden, current assignments, Tabu List
Output: A′                                  ▷ Assignment of reduced cost by unburdening j
    for all i ∈ A.joinUnitsForEngine(j) do
        for all j′ ∈ A.engines s.t. ¬(j′ = j ∨ tabuList.contains(i, j)) do
            working ← A
            working.assign(i, j)
            if ComputeTotalCost(working) ≤ ComputeTotalCost(A) then
                A ← working
                tabuList.put(i, j)
    return A
```

Pseudocode for the modified Tabu Search algorithm utilized by SciDB and implemented in the BigDAWG executor for join-unit assignment is shown in Algorithms 1 and 2. The algorithm initializes using the MBH assignment. The Tabu list is populated with this initial allocation, and per-engine cost estimates are calculated for the

assignment, as defined below in Section 5.3.3.

After processing the initial assignment, the algorithm loops until it can no longer improve upon the initial plan. All engines involved in the join are iterated over, and those with high cost are evaluated for potential improvements.

Attempting to lighten the load of a given engine consists of iterating over each join-unit assigned to that engine, and determining if assigning the join-unit to a less burdened engine decreases the cost of the overall plan, as defined below in Section 5.3.3. If the change does reduce the total cost of the assignment, the new assignment is marked as optimal, and the reassignment is noted in the Tabu List. Once the algorithm reaches a local optimum, it stops executing.

The approach of formulating the Tabu List as a join-unit to engine mapping is borrowed from SciDB's shuffle join handling. By formulating the Tabu List in this way, we no longer have to examine the exponential $O(2^{n*u})$ possible assignments of u join-units to n engines. Instead, we can reduce the search space to the polynomial $O(n * u) = O(\max(n, u)^2)$.

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 engines.

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.

As the speeds at which each distinct engine is able to complete these steps varies between engines, it is necessary to determine a cost estimation mechanism that is able to handle varying costs per engine. The approach utilized is shown in Algorithm 3. The cost for a specific engine is determined by the sum of the most expensive inbound migration (since migrations are parallelized) 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, utilizing predetermined constants.

Algorithm 3 Tabu Search per-engine costs

```
  function ComputeEngineCosts(A)
Input: A                                              ▷ Current assignments
Output: C                                  ▷ Map of engines to per-engine costs
    A ← {}
    for all j ∈ 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 ← 0
    n ← 0
    for all j' ∈ A.engines do
        m ← A.getAssignedTuples(j', j).count()
        (c1, c2) ← MigrationConstants.get(j, j')
        maxMigrationCost ← max(c1 * m^2 + c2 * m, maxMigrationCost)
        n ← n + m
    (c3, c4) ←ComparisonConstants.get(j)
    tupleComparisonCost ← c3 * n^2 + c4 * n
    cost ← maxMigrationCost + tupleComparisonCost
    return cost
```

As illustrated in Algorithm 4, 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 4 Tabu Search total cost

```
  function ComputeTotalCost(A)
    costs ← ComputeEngineCosts(A)
    return Max(costs)
```

The downside to this cost model is that its correlation with real-world execution time clearly depends on the constants for each participating engine. In cases where the performance with respect to the number of tuples differs significantly from the cost estimates, Tabu Search will assign join-units suboptimally. In our experiments,

we use cost model parameters determined empirically by performing a regression across varying workload sizes in order to reflect the performance characteristics of the participating databases.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Experimental Results

In this chapter, we analyze the performance of the Skew Examination strategies described in Chapter 4 and the Join-Unit Assignment strategies described in Chapter 5.

We start by describing the experimental setup. Then, we analyze the results of our performance evaluation on the Skew Examination strategies. We close by examining the performance of the different Join-Unit Assignment strategies under varying degrees of skew.

6.1 Experimental Setup

Though the approaches implemented in the Query Executor module scale to any number of participating engines, the status of the Query Planner module at the time of writing does not present the Executor with opportunities to execute shuffle joins across more than two engines. In the case of a complex join across multiple engines that share a single predicate, the Planner module creates a logical plan containing a cascading sequence of operations involving exactly two engines, despite the potential benefits of parallelization through shuffling across multiple engines.

As a result, we conducted our skew examination and join-unit assignment evaluations on an installation of BigDAWG consisting of two virtualized PostgreSQL instances of different capacity. The larger engine had 150GB of data with four 3.4GHz processor cores and 32GB RAM, while the other had a 75GB of data with a single

3.4GHz processor core and only 8GB RAM. Both were using a switched network and SATA disks. The joins are evaluated on the schema shown in Figure 6-1, and the cross-engine Join workload of `SELECT * FROM A, B WHERE A.id = B.id`. Because typical joins in the BigDAWG system are across intermediate un-indexed results of Select, Project, and Filter operations, no indexes were used. A trial consists of this workload run once with the logical plan's destination engine specified as the larger engine, and once with the the smaller. Each trial was started with a cold cache and executed 5 times, with the average duration reported, segmented by phase of execution.

```
CREATE TABLE bigdawgdev.test_table(
    ID   BIGSERIAL        NOT NULL,
    NAME VARCHAR (20)     NOT NULL,
    AGE  INT              NOT NULL,
    ADDRESS  VARCHAR (40)
    PRIMARY KEY (ID)
);
```

Figure 6-1: Test database schema

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

6.2 Skew Examination Strategy Evaluation

The first strategies that were evaluated were the skew examination strategies. In order to compare the Full Table Scan and Sampling skew examination strategies, we utilize them to execute the workload first using the Minimum Bandwidth Heuristic, 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.

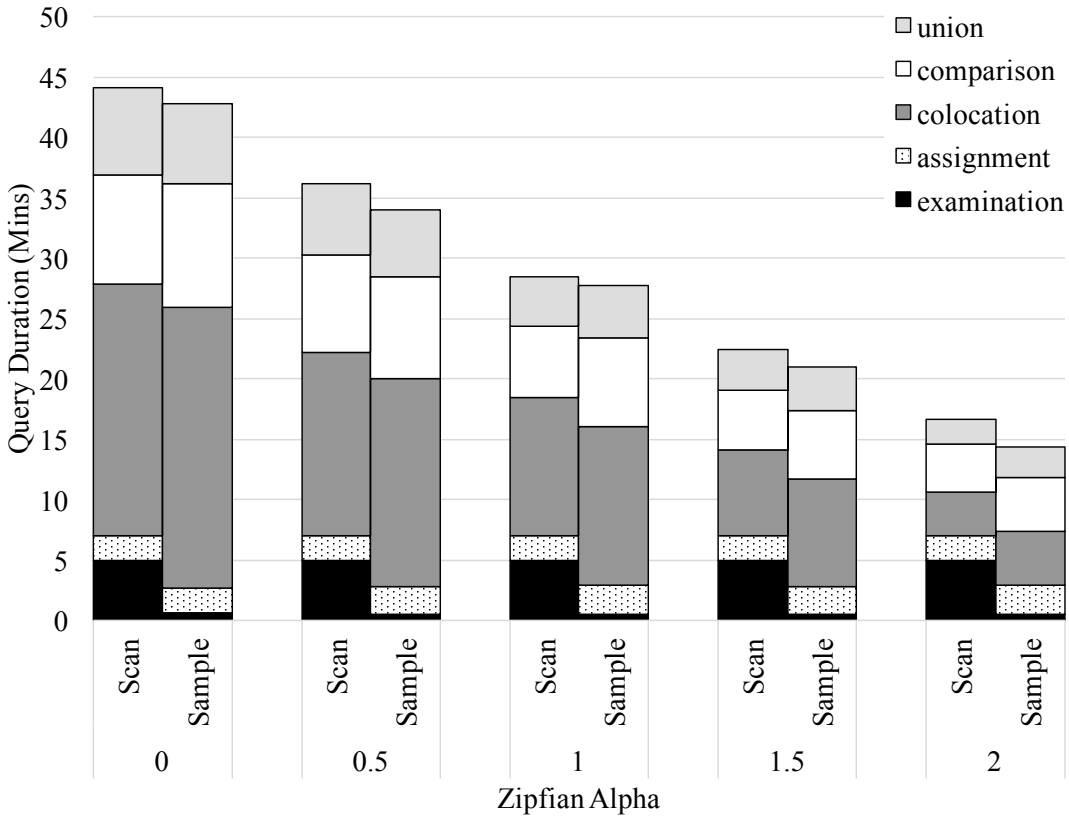

Figure 6-2: Join performance with varying skew and skew examination strategies, using Minimum Bandwidth Heuristic assignment

The benchmark results for both skew examination strategies with the Minimum Bandwidth Heuristic assignment are shown in Figure 6-2, while those for Tabu Search assignment are shown in Figure 6-3. The figures illustrate the total runtime of the benchmark query under different Zipf distributions, segmented by time taken for each phase. As expected, the portion of total runtime consumed by the Skew Examination phase (shown in black) is significantly superior with the sampling strategy than with 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 join. As indicated by the results, when the skew of the data is low, the sampling strategy may result to less efficient assignments for both the MBH and Tabu

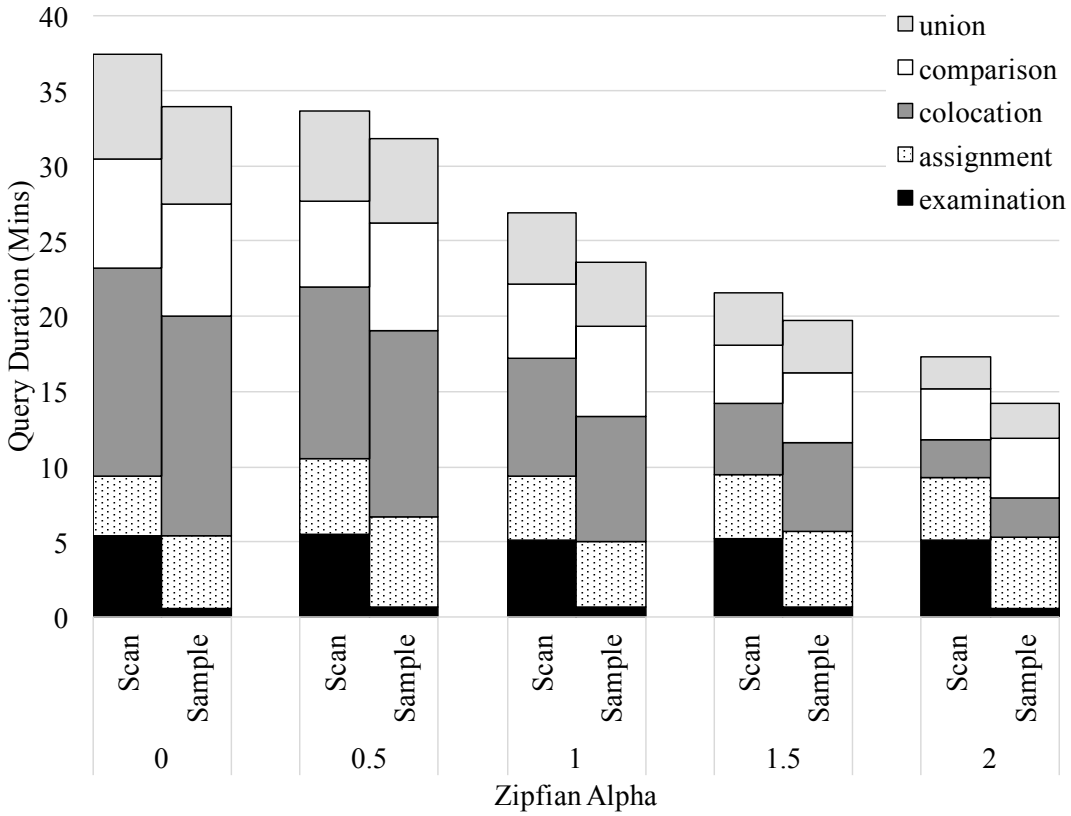

Figure 6-3: Join performance with varying skew and skew examination strategies, using Tabu Search assignment

assignment strategies. As skew increases, we notice that the post-examination phases have approximately equivalent runtimes regardless of skew examination strategy. 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.

From these results, we conclude that the Sampling skew examination strategy results in performance that is strictly superior to the Full Table Scan strategy in the Skew Examination phase. The total runtime under sampling is almost always superior to the Full Table Scan strategy when accounting for all phases involved in the shuffle join, especially when performed over skewed data.

6.3 Join-Unit Assignment Strategy Evaluation

The second performance analysis was examining the performance differences between all Join-Unit Assignment strategies described in Chapter 5 as the skew measure is increased. As we saw in Section 6.2, the Sampling skew examination strategy almost always outperformed the Full Table Scan strategy. As a result, in the following experimentation, the Sampling strategy is utilized wherever skew examination is required.

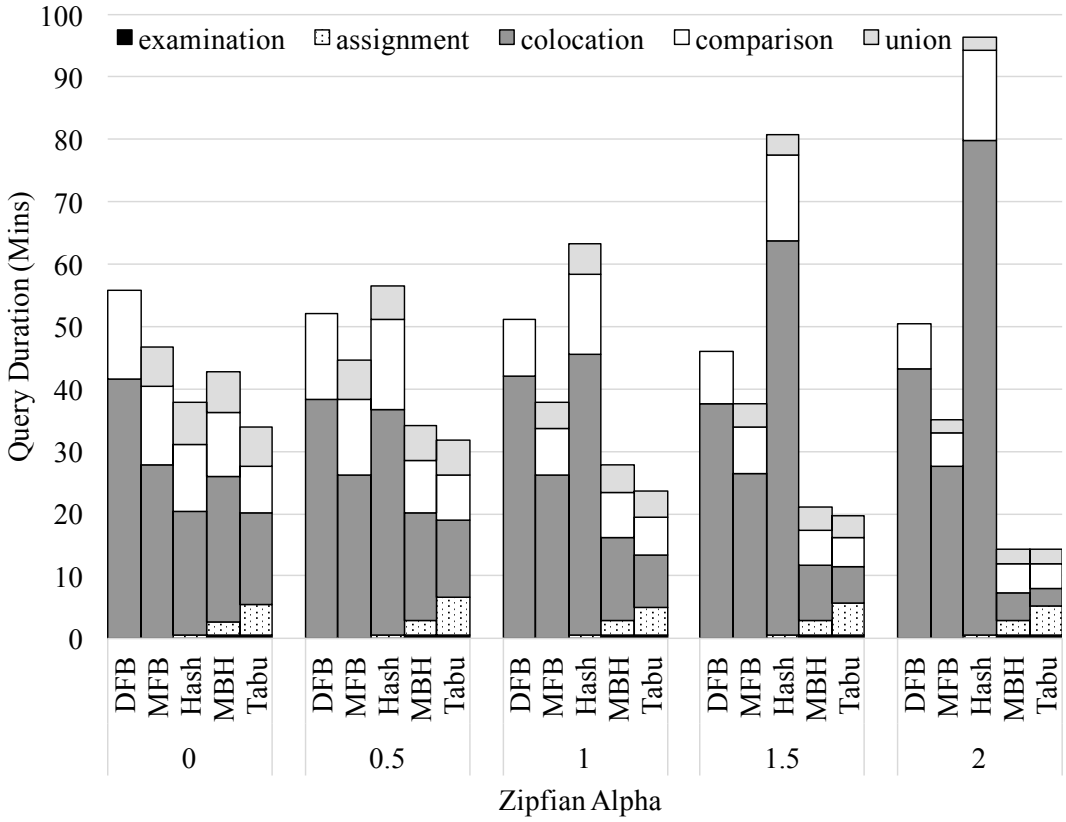

Figure 6-4: Join with varying skew and assignment strategies

Figure 6-4 illustrates the performance of the plan generated by all assignment strategies given the experimental workload as the skew of the distribution varies. We note that the skew examination phase is nonexistant for the Destination Full Broadcast (DFB), Minimum Full Broadcast (MFB), and Hash assignment strategies, while the performance of the examination phase (using the Sampling strategy) for the Minimum Bandwidth Heuristic (MBH) and Tabu Search strategies is identical to

that in Figures 6-2 and 6-3.

The assignment phase is also not used with the DFB or MBH strategies. The computation of Hash assignments has a very minimal footprint, due to the fact that it is delegated to the participating engines and piggybacked onto the join-unit colocation phase. We observe that the MBH and Tabu Search strategies have additional overhead in the assignment phase, with the Tabu Search assignment determination taking roughly two to three times as long.

The DFB and MFB executions start off immediately with their colocation phase. These two strategies migrate every tuple from one engine onto the other. As a result, they are unable to realize any benefits from parallelization of the Migration process and the tuple comparison phase. In the portion of the workload where the smaller engine is the desired destination for the join result, the DFB strategy migrates every tuple from the larger engine to the smaller engine, resulting in a longer colocation and assignment phase. As a result, it consistently has higher migration times than MFB.

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.

Chapter 7

Future Work

Though the basics of the shuffle join executor show promising results, there are several areas in which it can improve.

7.1 Shuffle Joins across 3+ Engines

Though the approaches described in this report are generalized to operations across any number of engines in the Executor module, the Planner currently models more complex cross-engine operations as a cascading sequence of operations involving exactly two engines, even if there is a single join predicate shared among all such joins.

The shuffle join framework detailed in this report can perform joins across multiple engines if they share a common predicate, allowing for an even larger amount of savings by increased parallelization. Additionally, the shuffle join framework could utilize multiple engines within the same island to distribute computational load if there are performance gains to be realized by doing so.

7.2 Probabilistic Data Structures

The Full Table Assignment methods discussed in Section 5.2 come with the downside of potentially migrating excess data in the form of tuples that have no matching values with which to join at their destination engine.

The amount of unnecessarily migrated tuples could be significantly reduced by constructing probabilistic data structures such as Bloom Filters[13] to represent the data contained at each participating engine, and only migrating join attributes which test positive.

7.3 Cost Estimation by Training

The Tabu Search join-unit assignment strategy can significantly outperform other strategies in the right circumstances. Unfortunately, the current implementation requires constants to be determined in advance. The natural next step to solve this problem would be to increase and leverage the data collected by the Monitor module to train a model to determine these constants for each distinct engine in the system.

7.4 Parallelizing Alignment and Comparison

Currently, interaction with the Data Migration module is through a mechanism that returns when the entire migration has either succeeded or failed. Only after any necessary migrations are fully completed does the tuple comparison phase of query execution start.

Query processing time may be improved if BigDAWG performed Joins across streams of data as the data being migrated, but this brings both design and consistency concerns that have not yet been fully explored and are beyond the scope of this report.

7.5 Inter-Island Shuffle Join Execution

The current goal of the Query Executor is to implement efficient query execution for intra-Island, cross-engine joins and rely on explicit Migration commands for queries that span Islands. The strategies in this report could be used to intelligently plan Joins that span across Islands, but doing so would require a more thorough under-

standing of the costs associated with cross-Island Migrations.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 8

Conclusion

We have introduced a shuffle join Executor that is able to efficiently execute joins spanning across multiple database engines, and performs 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 indicated 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.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Accumulo. `https://accumulo.apache.org/`.

[2] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[3] Paul G Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 963–968. ACM, 2010.

[4] Rainer E Burkard and Eranda Cela. *Linear assignment problems and extensions*. Springer, 1999.

[5] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. The bigdawg polystore system. *SIGMOD Rec.*, 44(2):11–16, August 2015.

[6] Jennie Duggan, Olga Papaemmanouil, Leilani Battle, and Michael Stonebraker. Skew-aware join optimization for array databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 123–135, New York, NY, USA, 2015. ACM.

[7] 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.*, 8(12):1908–1911, August 2015.

[8] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Comput. Oper. Res.*, 13(5):533–549, May 1986.

[9] Carolyn J. Hursch and Jack L. Hursch. *SQL: The Structured Query Language*. TAB Books, Blue Ridge Summit, PA, USA, 1988.

[10] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker,

Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, August 2008.

[11] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[12] Manish Mehta and David J DeWitt. Data placement in shared-nothing parallel database systems. *The VLDB JournalThe International Journal on Very Large Data Bases*, 6(1):53–72, 1997.

[13] James K Mullin. Optimal semijoins for distributed database systems. *Software Engineering, IEEE Transactions on*, 16(5):558–560, 1990.

[14] Mohammed Saeed, Mauricio Villarroel, Andrew T Reisner, Gari Clifford, Li-Wei Lehman, George Moody, Thomas Heldt, Tin H Kyaw, Benjamin Moody, and Roger G Mark. Multiparameter intelligent monitoring in intensive care ii (mimic-ii): A public-access intensive care unit database. *Critical care medicine*, 39(5):952–960, 05 2011.

[15] Michael Stonebraker, Chuck Bear, Uğur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stan Zdonik. One size fits all? part 2: Benchmarking results. *Proc. CIDR*, 2007.

[16] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The architecture of scidb. In *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management*, SSDBM'11, pages 1–16, Berlin, Heidelberg, 2011. Springer-Verlag.

[17] Michael Stonebraker and Greg Kemnitz. The postgres next generation database management system. *Commun. ACM*, 34(10):78–92, October 1991.