PolyBench: The First Benchmark for Polystores

Jeyhun Karimov[1], Tilmann Rabl[1,2], and Volker Markl[1,2]

[1] DFKI, Germany
[2] TU Berlin, Germany

Abstract. Modern business intelligence requires data processing not only across a huge variety of domains but also across different paradigms, such as relational, stream, and graph models. This variety is a challenge for existing systems that typically only support a single or few different data models. Polystores were proposed as a solution for this challenge and received wide attention both in academia and in industry. These are systems that integrate different specialized data processing engines to enable fast processing of a large variety of data models. Yet, there is no standard to assess the performance of polystores. The goal of this work is to develop the first benchmark for polystores. To capture the flexibility of polystores, we focus on high level features in order to enable an execution of our benchmark suite on a large set of polystore solutions.

1 Introduction

Modern business questions frequently comprise complex analytical queries with multiple data types and data models, residing on several data storage and processing systems. This has led to a large number of domain-specific database engines with diverse capabilities since it is hard to support all kinds of heterogeneous queries within a single data processing engine [22]. For these setups, polystores have been proposed to combine systems that specialize in specific execution and data models.

Similarly, there is a growing community supporting one size might fit all, such as Apache Spark [27] and Weld [20]. These systems combine numerous analytics in a single engine to enable generic data representation and benefit from common intermediate representation for further optimization.

Despite the hype on heterogeneous analytics, whether on polystores or on single generic-purpose stores, there is no consistent evaluation method. As a result, each solution presents its own performance measurements. For example, some polystore solutions are built for a specific use-case [9], while others use TPC queries for their evaluation [12]. As a result, there is also no common workload, driver, and metrics for systems performing heterogeneous analytics. This makes it hard for a user to compare systems with different evaluation strategies. Although we concentrate on polystore evaluations in this paper, we also perform a thorough comparison between polystore and single, general-purpose engine.

We propose PolyBench, the first benchmark for heterogeneous analytics systems, especially for polystores, providing a complete evaluation environment. Our aim is to provide a benchmark suite with evaluation metrics and workloads, which will eventually lead to better baselines. Currently, a general accepted baseline for polystore

evaluation is a single, general-purpose engine. The outcome of previous performance comparisons between polystores and single store engines is that polystores outperform single stores [7, 21]. However, as we show in this paper, this is not always the case. We evaluate the trade-offs between polystores and single-stores with various workloads.

PolyBench features a driver which benchmarks polystores with three main use-cases. We also provide a set of metrics which are specific to polystores. Our use-cases operate with structured, semi-structured, and unstructured data types and support relational, stream, array, and graph data processing paradigms. Our benchmark solution is not tied to a specific polystore solution, rather, it is generic and high level enough to be applied to any polystore.

We list the main contributions of this paper below:

- We propose PolyBench, the first polystore benchmark. Our benchmark suite consists of three main use-cases and two test scenarios. We provide a set of metrics for PolyBench, to conduct a thorough analysis.
- The main idea behind polystores is to overcome performance bottlenecks of single general-purpose stores. We conduct an analysis of this idea and compare polystores and single general-purpose stores.
- We conduct an extensive experimental analysis. We evaluate the systems under test with different parameters and combinations of parameters, provided by our benchmark driver.

We structure the rest of the paper as follows. We provide background information about the systems under test in Section 2. In Section 3, we survey related work. We explain use cases and our data model in Section 4. Section 5 describes test scenarios we adopt in PolyBench. We demonstrate our experimental analysis in Section 6. In Section 7, we discuss the results of experiments and analyze possible directions to improve our benchmark as future work. Finally, we conclude in Section 8.

2 Background

In this section, we give brief definitions of terms we utilize in this paper.

A **polystore** is union of different specialized stores, possibly with distinct language and execution semantics, supporting wide range of data types and analytics. We adopt the term polystore from BigDAWG [7]; however, our definition of polystore is more general to cover wide range of solutions. We utilize the term query for single stores and **use case** with polystores.

A **member-store** is a fundamental unit of a polystore, specialized and optimized for specific workloads. A member-store contributes most of its features to overall feature set of a polystore. As a result, a polystore supports a set of features and capabilities of its underlying member-stores. Once a user executes a use case to a polystore, a polystore optimizer splits the use case into subqueries, each of which directly addresses a particular member-store. A subquery might also contain embedded invocations to specified member-store's native query interface.

We differentiate three main member-stores. The first one is source member-store. A **source member-store** is a member-store from which a polystore ingests input data from outside world. The second one is sink member-store. A **sink member-store**

is a member-store which reside in the last ring of the overall pipeline and provide the output of a given use case to the user. The third type is relay member-stores. A **relay member-store** is a member-store, except sink member-store, which ingests its input data from other member-stores.

In this paper, we consider a polystore as a blackbox and tune it only with high level APIs. For example, a connection between a member-store and a polystore, whether it is mediator-wrapper or grouped islands architecture, is a system-specific design decision and out of the scope of this paper.

A **single store** is a general-purpose store or engine, which might or might not be a specialized in one or many workloads, supporting various analytics. We adopt the term single store to differentiate it from member-stores. In our experimental setup a single store supports all required features to execute our workloads. This enables us to conduct a thorough analysis between single store and polystore.

3 Related Work

There is a large body of work on polystores, each of which features a unique evaluation technique. In this section, we give an overview of existing polystore evaluation techniques. Below we categorize related works based on their main focus.

Language. Language design is an important component of polystores. It hides complex systems programming from users. Bondiombouy et al. propose a functional SQL-like query language that integrates data retrieved from different data stores [4]. Kolev et al. propose a similar SQL-like approach [14]. The authors provide specific queries for an evaluation of their solution. However, the member-store for data placement and query execution is hardcoded in the queries. We, on the other hand, formulate our use cases for polystores to be transparent both in terms of data placement and engine selection.

Tools. To enable data transparency between member-stores of a polystore, efficient data transfer and transformations are required. Dziedzic et al. analyze data migration between a diverse set of databases, including PostgreSQL, SciDB, S-Store, and Accumulo [8]. Pipegen features a similar approach automatically generating data pipes between DBMSs [11]. The authors of both papers evaluate their solutions with data migration-/transformation-specific use cases. These benchmarks are difficult to generalize for polystore evaluation as they are not high level enough to cover a polystore benchmark.

Optimizer. Workflow optimization is important to efficiently place and move data in polystores. Chen et al. focus on the optimization of the amount of data movement [6, 25]. The main limitation of this work is that data placement and member-stores are tightly coupled. Our benchmark on the other hand, has no prior assumption on data placement or migration. Because PolyBench considers systems under test as blackbox, our benchmark leaves all optimization decisions to the optimizer of a system under test. Jovanovic et al. soften the data placement condition in member-stores and develop an algorithm to choose member-stores [12]. The authors adopt TPC-H and TPC-DS queries for evaluation. MISO also adopts a similar evaluation method [15]. The main limitation is that TPC queries are not designed for heterogeneous analytics workloads.

Specialized benchmarks. There are also some works focusing on polystore performance analysis. However, these typically consider only a specific polystore and analyze its capabilities. Kolev et al. analyze the polystore built on the CloudMdsQL language [14] and conduct experiments on the main features of relational and NoSQL engines [13]. Yu et al. evaluate the performance of BigDAWG [7] with MySQL and Vertica member-stores [26]. The authors adapt TPC-H queries for their evaluation. The main limitation of previous work is that the benchmark design is specific to the proposed solution. We, on the other hand, propose a generic benchmark suite that can be applied to any polystore solution.

Currently, BigDAWG executes workloads comprising diverse queries by identifying sweet spots in member-stores. However, to effectively identify strengths and weaknesses of query processing capabilities of member-stores, a formalization the performance characteristics is required. According to one of the authors of BigDAWG, Jennie Rogers, an important step to solve this problem is to find minimal set of evaluation use cases [2]. For a better performance, monitoring framework should feed the evaluation results to a polystore optimizer. Our work is the first initiative to solve the more general issue incorporating a diverse set of polystore solutions.

Lu et al. propose their vision to benchmark polystores concentrating on data models [19, 17, 18]. The main limitation of this proposal is that data model conversion and transformation is only one facet of general polystore evaluation. Furthermore, to ensure black-box evaluation, the data models and conversion between them should be transparent to the benchmark driver. Our benchmark suite, on the other hand, performs analysis in with high level APIs and leave all low-level details to system under test. BigBench is an industry standard benchmark for big data analytics [10]. The focus of this benchmark is benchmarking big data processing systems. We, on the other hand, concentrate on benchmarking polystores, combination of big data processing systems.

In previous works many different evaluation methods were proposed, each of which is specific to either one polystore instance or one implementation aspect. Our work is the first to propose a generic, holistic polystore benchmark.

4 Data Model and Use Case

PolyBench is an application level benchmark and simulates a banking business model. We choose banking, since it features heterogeneous analytics and data types. PolyBench's data set comprises structured, semi-structured, and unstructured parts.

4.1 Data Model

Relational. From the Figure 1, (1) describes the list of bank customers. (2) is the list of people globally blacklisted. (3) is the customer transactions table.

Stream. The main characteristic of the stream data model is that data is continuously arriving, possibly infinitely. There is no standard streaming data format, it can be structured, semi-structured, and unstructured.

In Figure 1 (4) is a stream that represents online operations. This is necessary for analyzing and debugging potential problems in real-time. One example would be

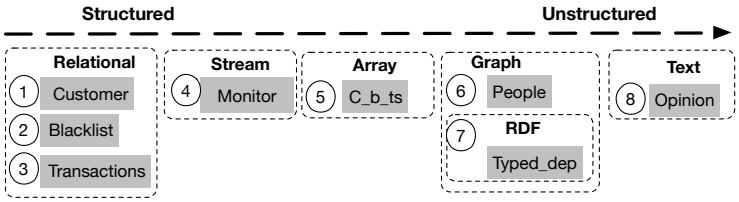


Fig. 1: PolyBench data model.

real-time fraud detection. Another example is monitoring exchange operations and updating exchange rates based on the current assets of the bank.

Array. While traditional DBMS platforms organize data in tables, array databases store data in array data model. The array model can have several dimensions, resulting in n-dimensional matrices. An array data model should be able to handle various scenarios, such as dense data (images), time series data, sparse arrays, and etc. The main goal is to fetch required data with few disk accesses by adjusting the tiling of the array to the access patterns. An array data model also tries to maintain a spatial proximity on disk, reducing the disk I/O during subsetting.

In Figure 1 (5) shows our array data. We store 3-dimensional (customer - balance - time) data in an array format, which stores the balance of a customer at a given time.

Graph. Similar to stream data model, there is no unified way to represent graph data. We use two graph datasets for graph data. (6) represents the relationships between customers. This is useful for calculating credit scores of customers. If a customer has a financial connection with someone, that person's name can appear on customer credit report. As a result, when a bank looks to the customer credit report, it also checks people the customer linked with. Thus, having financial connections with people with low credit score can affect customer credit score. (7) shows the RDF data extracted from (6) and (8).

Text. Text data is an unstructured information that lacks a pre-defined data model. (8) includes comments or public tweets about a bank. We use publicly available customer review data set [16].

As we can see, the overall input data consists of different data models, each of which with a separate **homogeneous** data set. Throughout the paper we utilize the term **heterogeneous input** for the union of several **homogeneous inputs** to a system under test. For example, a heterogeneous input may consist of a set of relational, stream, and array homogeneous inputs.

4.2 Use Cases

The amount of data stored by banks is rapidly increasing triggering banks to push new data processing technologies into their production environment [23]. To survive in a competitive world, it is necessary to adopt big data analytics as part of their core data processing strategy Apart from the volume, the diversity of data also increases, resulting in heterogeneous data and processing models. Inspired by this trend, we provide three use cases in Figures 2, 3 and 4.

```
INSERT INTO typed_dep VALUES (
CONVERT_INTO_RDF (
 SELECT *
 FROM Customer c
 WHERE c.updated >  arg  as u)
 UNION
 CONVERT_INTO_RDF (
 SELECT *
 FROM People p
 WHERE p IN u)
 UNION
 CONVERT_INTO_RDF (
 SELECT opinion_text
 FROM Opinion o
 WHERE o.ts > arg )
)
```

Fig. 2: Use Case 1

```
SELECT *
FROM (
 SELECT customer.userID
 FROM customer
 WHERE cutomer.work = null) AS c,
(SELECT userID
 FROM c_b_ts
 WHERE c_b_ts.balance > arg1
 AND c_b_ts.year=arg2) AS c2,
(SELECT p.userID
 FROM people p
 WHERE p sp blacklisted < arg3)
 AS p

WHERE p.userID = c.userID
AND c.userID=c2.userID
```

Fig. 3: Use Case 2

Bank multi-model data integration. In this use case we combine data residing in different sources to provide users a unified view. We integrate (1), (6), and (8) into (7), constructing a clear high level abstraction. The use case utilizes RDF as a target data type. At the sink operator of each engine, except the sink member-store, we put an additional operator, CONVERT_INTO_RDF. The operator converts relational data (Customer table) to RDF (id - columnName - columnValue). The conversion of

```
SELECT *
FROM customer c, transactions t, c_b_ts,
(SELECT *
FROM monitor m
WHERE m.userID IN blacklist.userID)
as fraud
WHERE c.userID = fraud.userID
AND t.userID = fraud.userID
AND fraud.userID = c_b_ts.userID
AND c_b_ts.ts within param_time
```

Fig. 4: Use Case 3

graph data model is in (sourcePersonID - relationName - destPersonID) format. For the text data, we extract (object - predicate - subject) patterns and construct RDF[3].

Customer background check. In this use case we check customer background to detect suspicious customers for further investigation. Schufa[4] is one example for customer background check. In our use case, if a customer is unemployed but has last year overall balance above some threshold and has very few connections to other people (for people having accounts in offshore banks) or some connections to blacklisted people, then the use case takes them into further consideration.

Continuous queries: fraud detection. In many financial applications, a data processing system may consume data in the form of continuous data streams, rather than finite stored data set. In this use case we consume and process realtime data and enrich it with other data sources. To be more precise, for every streaming tuple from (4) we check if the tuple ID is blacklisted. If so, we retrieve all transactions and balance information for the last week for the particular user for further investigation.

Based on the physical query execution plan of a polystore we categorize our use cases into two groups: dependent and independent polystore use cases. A **dependent polystore use case** is a use case, which consists of at least one relay member-store as a result of polystore deployment plan. An **independent polystore use case** is a use case which does not have any relay member-store as a result of a polystore deployment plan.

5 Benchmark Design

5.1 Metrics

Metrics are standard units to measure the performance of a system under test. Previous works generally adopt runtime as main metric for polystores. Although this is a proper metric for a polystore evaluation, it is not enough to get a good overview of polystore performance. Below we provide a set of metrics that we adopt for our benchmark.

[3] We partially benefitted from the library https://github.com/codemaniac/sopex
[4] https://www.schufa.de

Runtime. We use the term runtime for test scenarios consisting of batch use cases. Runtime is the time span between the polystore's start time, earliest start time of the member-stores, and end time, the latest end-time of member-stores, for processing the given use case. Thus, runtime is associated with the whole polystore system.

We use the term latency for interactive test scenarios consisting of continuous and batch use cases. We compute latency metric per tuple. The latency is the time span between tuple entering the source member-store and the related result emission time from sink member-store.

Individual runtime. Although we are interested mainly in the overall runtime of a use case, to perform a thorough analysis it is important to measure individual runtimes of subqueries running in different member-stores. Individual runtime is the runtime of each member-store in a polystore. We adopt the term individual latency for use cases containing continuous test scenarios.

Idle time. The above metrics are related to the time span in which a polystore or a member-store performs data processing. However, member-stores might stay idle for some use cases. The idle time is a time span in which a member-store does not perform any computation. The reason is mainly a blocking upstream member-store, especially in dependent polystore use cases. Note that only elected member-stores, which are selected by polystore query optimizer for executing a given use case, are considered for this metric.

Load. In PolyBench the load is defined by the size of the heterogeneous input data. We adopt 10GB, 50GB, and 100GB heterogeneous input data each of which consists of different homogeneous input data sizes.

5.2 Test Scenarios

Our test scenarios categorize the use cases based on their mode, which can be *i*) one-shot scenarios, *ii*) continuous scenarios. We propose two main test scenarios for PolyBench. In our experiments, we analyze each test scenario separately and together. Because there are many parameters contributing to the performance of a polystore, we design our test scenarios to measure the best and the worst performance after parameter tuning.

Resource distribution. The first test scenario is resource distribution among member-stores. Member-stores reside in the left set and resources are in the right set. There is a many-to-many relation between the two sets. In this test scenario, we evaluate the result of different mapping strategies from member-stores set to resources set.

The resource distribution scenario receives the amount of overall resources as an input. In our case the resource includes nodes in a cluster, memory, and CPU. The test scenario assigns each resource to a particular member-store and ensures all resources are utilized by member-stores of a polystore. For the single store case, it assigns all resources to the resource manager of the single store engine.

One usage of this test scenario is scale-out/in scenarios. For example, a user has some information about the input data. She knows with the existing resources it is inefficient to process all of input data. So, once a user decides to add new resources, because of the performance issues, a polystore should distribute new resources among member-stores in an optimal way.

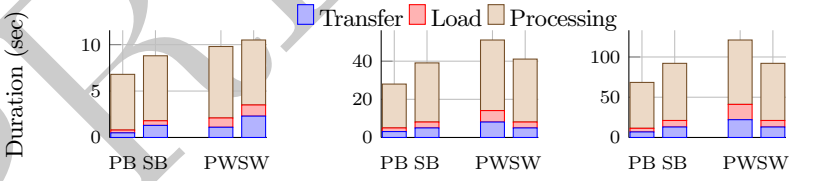
Load distribution. The second test scenario is load distribution among member-stores. There are two main factors contributing to the load of member-stores, being an input data size and assigned subqueries. Suppose a user submits a use case to a polystore. The polystore optimizer divides the use case to several subqueries, based on some meta-data and assigns subqueries to member-stores. As a result of the assignment if the performance of a particular member-store is a bottleneck to the whole use case, then there are several solutions. One option is to share the subquery with another member-store, which also supports all necessary features to execute the subquery. Another option is to recompile the use case and reassign subqueries to member-stores.

Because subquery assignment to member-stores is an internal process of a system under test and because we treat system under test as a blackbox, we concentrate on the second factor contributing the load distribution test scenario, being an input data size. Because the heterogeneous input consists of different homogeneous inputs, the idea of this test scenario is to tune the size of homogeneous inputs, find different ratio of homogeneous input sizes and ensure the size of heterogeneous input data is constant.

6 Experiments

6.1 Setup

We conduct experiments with the polystore BigDAWG v0.1 and single general purpose engine Apache Spark v2.3.0. We use Apache Giraph v1.2.0 [3] for workloads containing graph processing. We setup our experiments on a shared-nothing cluster. Our cluster consists of 20 nodes. Each node is equipped with 2.40GHz Intel(R) Xeon(R) CPU with 16 cores. System clocks in all machines throughout the cluster are synchronized via a local NTP server. Unless stated otherwise, we deploy all member-stores of a polystore to different cluster nodes. We utilize 10GB, 50GB, and 100GB datasets for benchmarking.



(a) 10GB data size (b) 50GB data size (c) 100GB data size

Fig. 5: Effect of tuning homogeneous data sizes with 10GB, 50GB, and 100GB heterogeneous input data size. PB stands for BigDAWG performance with tuned data distribution, SB stands for Spark performance with tuned data distribution (the same distribution as PB), PW stands for the worst BigDAWG performance, and SW means Spark performance with homogeneous input data same distribution as PW.

6.2 Use Case 1

We convert each tuple to RDF format in the sink operator of member-stores. Analyzing the deployment plan of BigDAWG we conclude that the use case belongs to the dependent polystore queries. To be more precise, the result of select operation from the People table depends on the output of the select operation from the Customer table. As a result, the latter is a blocking operation for the former.

As we discussed in Section 5, the input data distribution contributes to the member-store load. In the following experiment, we keep the deployment configurations of BigDAWG constant and change the size/ratio of homogeneous input data keeping the overall heterogeneous input data size constant. The idea of the use case is that an enterprise might lack prior knowledge of the statistics of input data sets.

Figure 5 shows the effect of different homogeneous input data sizes, keeping the heterogeneous data size constant, for systems under test. We consider two cases: *i*) the best case - input data distribution is tuned according to deployment of member-stores and *ii*) the worst case - the distribution of input data and member-stores deployment are uncorrelated. In the first case, we tune the homogeneous input data sizes to be executed by different member-stores to maximize to overall performance of BigDAWG. In the second case, we show the worst performance of BigDAWG. In both experiments we also evaluate the single general purpose store. We observe that once we tune the ratio of homogeneous inputs, then BigDAWG performs better than Spark, because each member-store is specialized for special workloads and we provide such a particular workload.

We can see in Figure 5 that Spark is more robust to the changes in input data set, than BigDAWG. The reason is that Spark utilizes all dedicated resources, as opposed to a member-store which utilizes only a portion of the resources dedicated to BigDAWG. As a result, a member-store is more prone to become a bottleneck (to the whole polystore) than Spark. Indeed, if there are more bottleneck member-stores, then the overall performance of a polystore degrades significantly.

We also observe a serious performance degradation for BigDAWG once we play with the amount of homogeneous input data. Moreover, with increasing heterogeneous data size, the gap between the best and the worst case increases as well. One reason behind this behavior is scheduling. In BigDAWG a member-store can belong to only one island, although it might feature several characteristics of different islands. As a result, the BigDAWG scheduler is unable to share a subquery to member-stores residing in different islands. This causes limitations when the size of one homogeneous input data is larger than others.

In Spark, data sources reside in the upstream of the source operator. The source operators receive data from external data sources while other operators pull input data from upstream operators. When there is a blocking operation in the upstream operator, the global scheduler of Spark, DAGScheduler, assigns a non-blocking task to downstream task schedulers. The global scheduler might also eliminate downstream operator for some time allocating more resources to blocking upstream operator.

Similarly, in BigDAWG source member-stores receive input from external data sources and other member-stores obtain input data from upstream member-stores. The main limitation is that BigDAWG scheduler is not as dynamic as Spark scheduler. As a result, especially for dependent polystore use cases, an upstream member-store can easily become a bottleneck. That is, the result of the selection query in People

table depends on the result of the selection query from Customer table. As a result, member-store associated with the former table stays idle until the member-store linked with the latter table finishes. The problem increases with larger input data sizes.

Another reason behind poor performance, we observe in Figure 5, of BigDAWG with non-tuned data distribution is data partitioning. For a single system selecting an optimal data partitioning is a non-trivial task [24]. Performing so for a polystore is more challenging task as the task includes *i*) partitioning data among member-stores and *ii*) partitioning data within separate member-stores.

BigDAWG setup's transfer and load times contribute significantly to the overall use case runtime. The impact increases with increasing input data size. The main reason is that efficient data transfer strategies between different member-stores requires n-to-m connections between one member-store with n instances and another one with m instances. This is non-trivial as it requires changing an engine's communication internals and can cause synchronization issues. In Spark, on the other hand, these details are automatically handled transparent to a user.

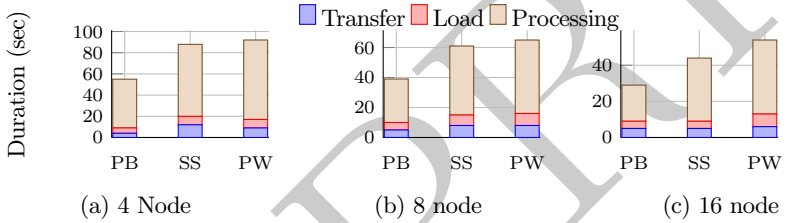


Fig. 6: Effect of scaling out in Spark and BigDAWG with 4, 8, and 16-node configurations. PB stands for the best performance of scaling out BigDAWG, SS means scaling out Spark, and PW mean the worst performance for scaling out BigDAWG.

Figure 6 shows the main idea behind scaling out in BigDAWG and Spark environment. In this case, we fix both heterogeneous and homogeneous data size and consider the number nodes in cluster as a variable. As a result, a user should be able to benefit from the performance of the systems under test with adding more resources. In this case we accept heterogeneous and homogeneous data as a constant variable.

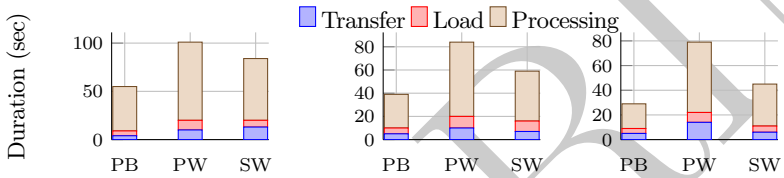
We can observe that once a user has knowledge about the input data domain and engine characteristics of the member-stores, then tuning BigDAWG for scaling out results with the best performance compared to Spark. Engine characteristics of member-stores refers to an estimation of each member-store performance with more resources.

We see a consistent scale-out performance for Spark. As we add more nodes to the cluster, the duration of computation improves. Although there are several parameters to tune manually such as garbage collection, serialized RDD storage, level of parallelism, and memory usage of reduce tasks, Spark performs the main network and I/O tuning transparent to the user. From this perspective, the required systems expertise is less for tuning the single engine for scaling out.

We note that the worst case scaling out scenario for BigDAWG causes performance problems. Worst case scaling out occurs when we increase the amount of

resources to a set of member-stores without having enough information about the characteristics of member-stores. A lightweight monitoring system, which BigDAWG currently lacks, might be a solution for this problem, where the framework monitors the performance of operators inside member-stores and member-stores as a whole and feed the information to the optimizer which assigns available resources among member-stores and among instances of particular member-store in an optimal way.

Although benchmarking scenarios individually is important, testing the performance of systems under test with combinations of different test scenarios gives us more insights. In Figure 7, we benchmark the performance of the systems under test with combination of both test scenarios: resource distribution and load distribution. The scenario occurs when a user is not an expert in BigDAWG and there is a little knowledge about input data. The result is that the performance gap between BigDAWG and Spark increases more than in the above experiments.



(a) 4 Node (b) 8 node (c) 16 node

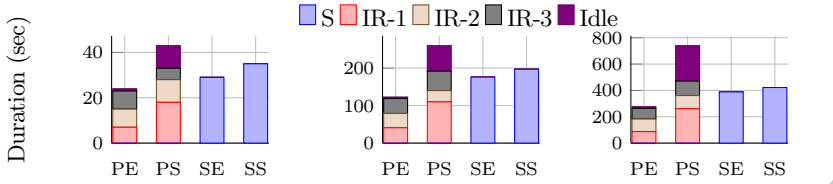
Fig. 7: Effect of scaling with different homogeneous data distribution for BigDAWG and Spark. PB stands for the best performance for BigDAWG. PW stands for the worst BigDAWG performance. SW stands for the performance of Spark. The heterogeneous input data size is constant.

6.3 Use Case 2

From the deployment plan of BigDAWG, we note that Use Case 2 is an independent polystore use case. BigDAWG divides the use case into sub-queries, submits to the relevant member-stores and merges once the results of all member-stores are ready. Independent polystore use cases spend less time for data transformation (from one member-store format to another) and reduce the amount of idle stay waiting for an upstream member-store.

We analyze the load distribution test scenario with BigDAWG and compare it with Spark. Figure 8 shows the results of engine load for Use Case 2. Idle time is the sum of periods in which member-stores stay idle. For the equal load in the figure, we configure member-stores such that the overall idle time is minimized. For skewed load, on the other hand, we arrange the distribution of the load to be random and to be different from each other at least 20%.

We observe that with a shared load, BigDAWG performs better than Spark. In this experiment, we measure the performance of a member-store with different loads and select a load combination which ensures the best performance for the whole



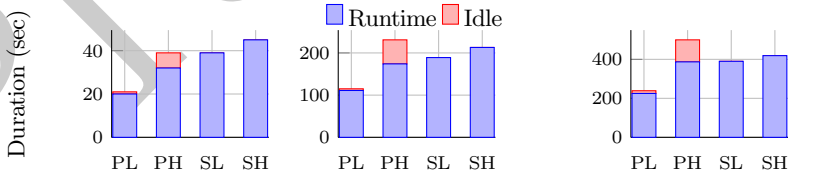
(a) 10GB input size (b) 50GB input size (c) 100GB input size

Fig. 8: Effect of engine load. PE refers to BigDAWG with equal load for member-stores, PS refers to BigDAWG with skewed load, SE refers to Spark with same load as PE, and SS refers to the performance of Spark with the same load as PS. Legends: S refers to runtime of Spark, IR-n refers to the individual runtime of *nth* member-store, and Idle refers to the overall idle time of BigDAWG.

polystore. The main reason behind the better performance of BigDAWG with shared load is that each member-store is specialized in assigned workload, resulting in overall improved performance.

We also perform experiment with skewed load. As a result, we can observe significantly increased idle times. Moreover, as the data size increases, the impact of idle time increases. We also observe a correlation between runtime of an individual member-stores and idle time.

We can also see that Spark is less susceptible to skewed load than BigDAWG. The reason is better scheduling and adaptive resource allocation in Spark. Dynamic resource allocation and scheduling is simpler in single engine environment. As a result, idle time duration in a cluster and the impact of skew is minimized in Spark. To avoid data skew, Spark, adopts TreeReduce and TreeAggregate methods and new aggregation communication pattern based on multi-level aggregation trees. At the beginning of the job, Spark's DAGScheduler assigns task schedulers to combine partial aggregates on local executors. Then, Spark shuffles the locally aggregated data to pre-scheduled reducers. For BigDAWG case, on the other hand, the optimizer lacks similar features, which in turn results in relatively poor performance.



(a) 10GB input size (b) 50GB input size (c) 100GB input size

Fig. 9: Effect of engine selectivity for BigDAWG and Spark. PL stands for BigDAWG with low selective subqueries, PH means BigDAWG with high selective subqueries, SL stands for Spark with low selective operators, and SH means Spark with high selective operators.

Similarly, in Figure 9 we analyze the effect of subquery selectivity in BigDAWG and operator selectivity in Spark. We define low selectivity being $s \leq 0.2$ and high selectivity being $s \geq 0.8$.

6.4 Use Case 3

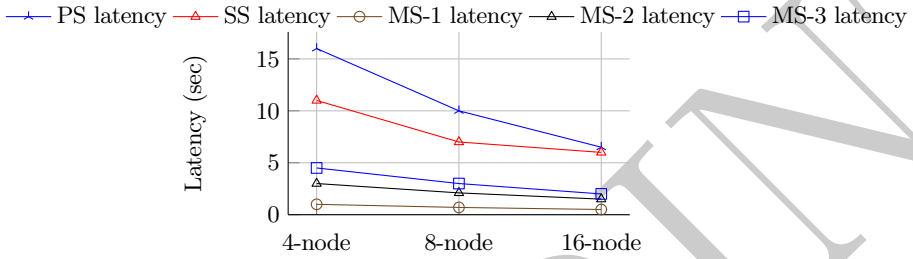


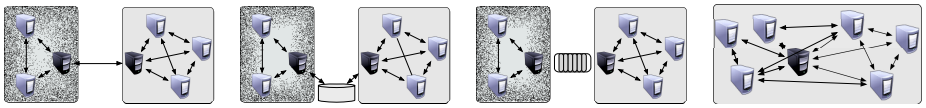
Fig. 10: Effect of running continuous queries on BigDAWG and Spark. PS latency stands for the latency of BigDAWG , SS latency means the latency of Spark, MS-1 is S-Store, MS-2 is PostgreSQL and MS-3 is SciDB.

Figure 10 shows the latency of input tuples. We can observe the skew in the latency distribution among member-stores. For example, the streaming engine in BigDAWG has the lowest latency. Because relational and array databases in our polystore are not optimized for streaming workloads, we observe a relatively high latency for the particular member-stores. Another reason for this behavior is synchronization and scheduling overhead among member-stores. For this type of queries BigDAWG would have benefited from caching feature among member-stores. Spark, on the other hand, provides automatic caching of frequently used RDDs.

We notice that the input/output semantics of member-stores is prone to be a bottleneck, especially with workloads including continuous queries. For example, the streaming member-store adopts a pull-based approach to ingest the input data and push based approach to output. It is not desirable to accumulate data inside the engine because once an operator state gets bigger, the performance degrades. For relational databases the input/output semantics are more relaxed. Depending on the size of the output, the system can stream or save the data in a temporary table for later use. Because backpressure mechanism is not available in polystores, adding flow controls to each engine, causes an additional latency because of the synchronization overhead among member-stores.

7 Discussion and Future Work

In this section, we summarize the findings of our experimental results. There is a need for better query optimizers and automation tools for polystores. Although this is not a new area in database research, existing optimizers work best for a specific



(a) A polystore data exchange via direct socket.

(b) A polystore data exchange through disc.

(c) A polystore data exchange via intermediate system.

(d) Data transfer for single store.

Fig. 11: Data transfer strategies for polystore and single store.

set of workloads. The overhead and required knowledge for tuning polystores is considerably higher than single store engines. Tuning a system is important to optimize and homogenize the performance. Although there is a large amount of research on self-tuning systems [5], performing so for a single-store system is still non-trivial. However, having information about input data can ease the work of database admin significantly. Tuning polystores for a given workload is much harder problem than tuning a single store systems. Tuning a polystore includes tuning all member-stores individually. Moreover, once there is a correlation among member-store workloads, as we saw in our experiments, tuning a polystore becomes even more complicated. For example, if the query involves interchanging the data between member-stores, the exact tuning decision is non-trivial at compile time.

A two-level scheduler (local for member-store and global for polystore) leads to non-negligible idle times. Relaxing the border between different layers of schedulers would increase polystore performance considerably.

In our experiments, we notice data representation and transfer to be a significant issue. While some works [11, 20] use a single data representation [1], others have multiple data representation [7]. The same is true for data transfer. Especially for continuous queries data transfer layers can easily become a bottleneck. The reason is that backpressure is non-trivial to implement for polystores, which would lead to gathering massive amounts of data in data transfer layer. Figure 11 shows three possible data transfer strategies for polystores and the data transfer for a single store. Although all different strategies have their own advantages and limitations, selecting the best option for the given workload is essential to improve the performance of polystores. BigDAWG, for example, supports the transfer strategies depicted in Figures 11a and 11b; however, these are hardcoded in the implementation and, thus, are not considered as a variable for an optimizer.

8 Conclusion

Polystores are designed to overcome the limitations of single general purpose data stores. To fill various gaps in data processing, there is an increasing number of polystores, with member-stores featuring different data models and execution models. This makes the solutions challenging to benchmark. In this paper we present PolyBench, the first benchmark for polystores. Our benchmark is generic and high level to support wide range of existing polystore solutions. We conduct an experimental analysis on a single store and a polystore and provide a comparative analysis. Our key finding

is that, although polystores are a key solution for most enterprise use cases, there are significant limitations for evaluation in previous works. Firstly, polystores perform better with tuned load and resource distribution. Secondly, current polystore designs are not compatible with continuous queries. We identify the main reasons for the above behaviours as lack of advanced optimizer, scheduler, and data transfer layer.

Considering that this work proposes the first benchmark for polystores, there is still a research to be carried for a complete and standard benchmark. A useful extension to our benchmark would be to support specialized polystores. Examples are graph based polystores and ML based polystores. An improvement would be to add workloads with all possible combinations of the above. An extension for measuring individual components would be to support benchmarking polystore tools, such as data transfer and data representation tools. Moreover, important metrics such as ease of use, maintainability, high availability, and performance robustness are key in production environment, which is part of the future work.

References

1. Apache arrow: A cross-language development platform for in-memory data. https://arrow.apache.org/ Last accessed 24 Feb 2018
2. Query modeling and optimization in the bigdawg polystore system. http://istc-bigdata.org/index.php/query-modeling-and-optimization-in-the-bigdawg-polystore-system/ Last accessed 10 March 2018
3. Avery, C.: Giraph: Large-scale graph processing infrastructure on hadoop. In: Proceedings of the Hadoop Summit. Santa Clara. vol. 11, pp. 5–9 (2011)
4. Bondiombouy, C., Kolev, B., Levchenko, O., Valduriez, P.: Integrating big data and relational data with a functional sql-like query language. In: International Conference on Database and Expert Systems Applications. pp. 170–185. Springer (2015)
5. Chaudhuri, S., Narasayya, V.: Self-tuning database systems: a decade of progress. In: Proceedings of the 33rd international conference on Very large data bases. pp. 3–14. VLDB Endowment (2007)
6. Chen, Y., Xu, C., Rao, W., Min, H., Su, G.: Octopus: Hybrid big data integration engine. In: Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on. pp. 462–466. IEEE (2015)
7. Duggan, J., Elmore, A.J., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T., Zdonik, S.: The bigdawg polystore system. ACM Sigmod Record **44**(2), 11–16 (2015)
8. Dziedzic, A., Elmore, A.J., Stonebraker, M.: Data transformation and migration in polystores. In: High Performance Extreme Computing Conference (HPEC), 2016 IEEE. pp. 1–6. IEEE (2016)
9. Gadepally, V., Chen, P., Duggan, J., Elmore, A., Haynes, B., Kepner, J., Madden, S., Mattson, T., Stonebraker, M.: The bigdawg polystore system and architecture. In: High Performance Extreme Computing Conference (HPEC), 2016 IEEE. pp. 1–6. IEEE (2016)
10. Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., Jacobsen, H.A.: Bigbench: towards an industry standard benchmark for big data analytics. In: Proceedings of the 2013 ACM SIGMOD international conference on Management of data. pp. 1197–1208. ACM (2013)
11. Haynes, B., Cheung, A., Balazinska, M.: Pipegen: Data pipe generator for hybrid analytics. In: Proceedings of the Seventh ACM Symposium on Cloud Computing. pp. 470–483. ACM (2016)
12. Jovanovic, P., Simitsis, A., Wilkinson, K.: Engine independence for logical analytic flows. In: Data Engineering (ICDE), 2014 IEEE 30th International Conference on. pp. 1060–1071. IEEE (2014)

13. Kolev, B., Pau, R., Levchenko, O., Valduriez, P., Jiménez-Peris, R., Pereira, J.: Benchmarking polystores: the cloudmdsql experience. In: Big Data (Big Data), 2016 IEEE International Conference on. pp. 2574–2579. IEEE (2016)
14. Kolev, B., Valduriez, P., Bondiombouy, C., Jiménez-Peris, R., Pau, R., Pereira, J.: Cloudmdsql: querying heterogeneous cloud data stores with a common language. Distributed and parallel databases **34**(4), 463–503 (2016)
15. LeFevre, J., Sankaranarayanan, J., Hacigumus, H., Tatemura, J., Polyzotis, N., Carey, M.J.: Miso: souping up big data query processing with a multistore system. In: Proceedings of the 2014 ACM SIGMOD international conference on Management of data. pp. 1591–1602. ACM (2014)
16. Leskovec, J., Sosič, R.: Snap: A general-purpose network analysis and graph-mining library. ACM Transactions on Intelligent Systems and Technology (TIST) **8**(1), 1 (2016)
17. Lu, J.: Towards benchmarking multi-model databases. In: CIDR (2017)
18. Lu, J., Holubová, I.: Multi-model data management: What's new and what's next? In: EDBT. pp. 602–605 (2017)
19. Lu, J., Liu, Z.H., Xu, P., Zhang, C.: Udbms: road to unification for multi-model data management. arXiv preprint arXiv:1612.08050 (2016)
20. Palkar, S., Thomas, J.J., Shanbhag, A., Narayanan, D., Pirk, H., Schwarzkopf, M., Amarasinghe, S., Zaharia, M., InfoLab, S.: Weld: A common runtime for high performance data analytics. In: Conference on Innovative Data Systems Research (CIDR) (2017)
21. Simitsis, A., Wilkinson, K., Castellanos, M., Dayal, U.: Optimizing analytic data flows for multiple execution engines. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. pp. 829–840. ACM (2012)
22. Stonebraker, M., Cetintemel, U.: " one size fits all": an idea whose time has come and gone. In: Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on. pp. 2–11. IEEE (2005)
23. Sun, N., Morris, J., Xu, J., Zhu, X., Xie, M.: icare: A framework for big data-based banking customer analytics. IBM Journal of Research and Development **58**(5/6), 4–1 (2014)
24. Valduriez, P.: Parallel database systems: open problems and new issues. Distributed and parallel Databases **1**(2), 137–165 (1993)
25. Xu, C., Chen, Y., Liu, Q., Rao, W., Min, H., Su, G.: A unified computation engine for big data analytics. In: Big Data Computing (BDC), 2015 IEEE/ACM 2nd International Symposium on. pp. 73–77. IEEE (2015)
26. Yu, K., Gadepally, V., Stonebraker, M.: Database engine integration and performance analysis of the bigdawg polystore system. In: High Performance Extreme Computing Conference (HPEC), 2017 IEEE. pp. 1–7. IEEE (2017)
27. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. HotCloud **10**(10-10), 95 (2010)