The BigDAWG Polystore System

Jennie Duggan
Northwestern

Aaron J. Elmore
Univ. of Chicago

Michael Stonebraker
MIT

Magda Balazinska
Univ. of Washington

Bill Howe
Univ. of Washington

Jeremy Kepner
MIT

Sam Madden
MIT

David Maier
Portland State Univ.

Tim Mattson
Intel

Stan Zdonik
Brown

ABSTRACT

This paper presents a new view of federated databases to address the growing need for managing information that spans multiple data models. This trend is fueled by the proliferation of storage engines and query languages based on the observation that "no one size fits all". To address this shift, we propose a *polystore* architecture; it is designed to unify querying over multiple data models. We consider the challenges and opportunities associated with polystores. Open questions in this space revolve around query optimization and the assignment of objects to storage engines. We introduce our approach to these topics and discuss our prototype in the context of the Intel Science and Technology Center for Big Data.

1. INTRODUCTION

In the past decade, the database community has seen an explosion of data models and data management systems, each targeted for a well-defined vertical market [3, 6]. These systems exemplify the adage that "no one size fits all" for data management solutions [21]. For example, relational column stores are poised to take over the data warehouse market; most of the major database vendors have embraced this technology. High-throughput OLTP workloads are largely moving to main memory SQL systems, with products available from multiple startups as well as Microsoft and SAP. Moreover, there has been a flood of NoSQL engines implementing a panoply of data models, and they typically operate on flexible storage formats such as JSON. The internet of things (IoT) calls for real-time stream processing and analytics that may be best served by either an OLTP engine or a stream processing engine. In addition, the jury is still out on the winning architecture for complex analytics and graph processing. Lastly, distributed file systems (a la HDFS) have become popular owing to their simple scalability and rich ecosystem of data processing tools.

Increasingly, we see applications that deploy multiple engines, resulting in a need to join data across systems. It is simply not reasonable for them to replicate all information on all platforms. This landscape calls for a new approach to federating data systems. At the Intel Science and Technology Center for Big Data, we have constructed a medical example of this new class of applications, based on the MIMIC II dataset [18]. These publicly available patient records cover 26,000 intensive care unit admissions at Boston's Beth Israel Deaconess Hospital. It includes waveform data (up to 125 Hz measurements from bedside devices), patient metadata (name, age, etc.), doctor's and nurse's notes (text), lab results, and prescriptions filled (semi-structured data). A production implementation would store all of the historical data augmented by real-time streams from current patients. Given the variety of data sources, this system must support an assortment of data types, standard SQL analytics (e.g., how many patients were given a particular drug), complex analytics (e.g., computing the FFT of a patient's waveform data and comparing it to "normal"), text search (e.g., finding patients who responded well to a particular treatment), and real-time monitoring (e.g., detecting abnormal heart rhythms).

Although it is conceivable to implement this entire application in a single storage system, the consequences of doing so would be dire. Not only would there be 1–2 orders of magnitude performance penalty on some of the workload, but the real-time requirements may not be achievable in a one-size-fits-all engine. In our reference implementation, which we call BigDAWG[1], we use SciDB [6] for archived time series data, Accumulo [1] for text, Postgres for patient metadata, and S-Store [8] for real-time waveform data. Obviously, there will be queries that span two or more storage engines. For example, to compare current waveforms to historical ones, one would query S-Store and SciDB. To find patient groups associated with particular kinds of prescriptions or doctor's notes, one would query Accumulo and Postgres. To run analytics on the waveforms from a particular cohort of patients, one would query Postgres and SciDB. It seems clear to us that a federation of multiple disparate systems will be a requirement in a myriad of future applications, especially those that are broad in scope like the MIMIC II example above.

We call such systems *polystores* to distinguish them from earlier federated databases that supported transparent access across multiple back ends with the same data model.

[1]So named after the ISTC Big Data Analytics Working Group

We begin by defining polystores and their architecture in Section 2. We then discuss our approach to query optimization within this framework in Section 3. After that, we turn to the monitoring system for query optimization and data placement. In Section 4, we discuss how our polystore model presents new challenges in assigning data to back ends. A discussion of prior work follows in Section 5.

2. POLYSTORE SYSTEMS

In this section, we present the semantic notion of polystores that we are exploring with our BigDAWG prototype. This approach is motivated by three goals. First, like previous federated databases, BigDAWG will support *location transparency* for the storage of objects. This transparency enables users to pose declarative queries that span several data management systems without becoming mired in the underlying data's present location or how to assign work to each storage engine. In this context, an *island of information* is a collection of storage engines accessed with a single query language. Section 2.1 explores this construct.

The second goal of our work is *semantic completeness*. A user will not lose any capabilities provided by his underlying storage engines by adding them to a polystore. Hence, BigDAWG will offer the union of the capabilities of its member databases. As we will see in Section 2.2, this calls for the system to support multiple islands.

Our third aim is to enable users to access objects in stored a single back end from multiple islands. For example, a user accessing MIMIC II waveform data may express real-time decision making in SQL with streaming semantics (e.g., "raise an alarm if the heart rate over this window exceeds some threshold"). On the other hand, he may express complex analytics using an array language with queries like, "compute the FFT over all heartrate waveforms, grouped by patient and day". For such a user, it is desirable to have the same data be queryable in both an array and a relational island.

Enterprises also have legacy systems that access newly federated datastores. Obviously, they want their existing applications to continue to run while simultaneously writing new ones in a more modern language. Again, we see a need for a single datastore to participate in many islands.

In summary, a polystore will support a many-to-many relationship between islands of information and data management systems over multiple, disparate data models and query languages. This framework is designed to minimize the burden associated with running many back ends while maximizing the benefits of the same.

2.1 Islands of Information

In BigDAWG we use islands of information as the application-facing abstraction with which a user interacts with one or more underlying data management engines. Specifically, we define an *island of information* as:

A data model that specifies logical, schema-level information, such as array dimensions or foreign key relationships.

A query language for that data model. It is with this language that users pose queries to an island.

A set of data management systems for executing queries written to an island. Here, each storage engine provides a *shim* for mapping the island language to its native one.

When an island receives a query, it first parses it to create an abstract syntax tree (AST) in the island's dialect. It also verifies that the query is syntactically correct. It then uses the query optimizer to slice the AST into subqueries, where subqueries are the unit with which the polystore assigns work to back ends. The island then invokes one or more shims to translate each subquery into the language of its target data store. The island next orchestrates the query execution over the engines and accumulates results. The island itself will do little computation other than concatenating query results from its data stores. This design permits us to take advantage of the optimizers in the back ends rather than managing data at the federator level.

Ideally, an island will have shims to as many databases as possible to maximize the opportunities for load balancing and query optimization. At the same time, individual users may be comfortable working in different query languages and data models. In general, we expect that users will desire islands for well-known data models, such as relational and streaming, and that these islands will overlap in practice. In BigDAWG, two systems developed by ISTC members will be used as initial islands, and we will build additional ones as we go along. The two initial systems are Myria [13], which uses relational-style semantics plus iteration, and D4M [15], which implements a novel data model based on associative arrays. In our prototype, both islands will mediate access to MIMIC II data stored in the same engines (Accumulo, SciDB, S-Store, and Postgres).

Neither island contains the complete functionality of any of the underlying engines. Differing support for data types, triggers, user-defined functions, and multiple notions of null are invariably system-specific. To satisfy our goal of semantic completeness, we require *degenerate* islands that have the complete functionality of each storage engine. Our initial prototype will have six islands: four degenerate ones (relational, array, streaming, and text) augmenting the two described above. In general BigDAWG will have as many islands as one wants, each providing location transparent access to one or more underlying storage engines.

2.2 Cross-Island Querying

An island within a polystore supports location transparency via a shim for each storage engine. For a storage engine to join an island, a developer writes a shim. If a single-island query accesses more than one storage engine, objects may have to be copied between local databases. Here, a *CAST* mechanism copies objects between back ends.

When a user command cannot be expressed in a single island's semantics, he must convey his query in multiple island languages, each of which is a subquery. To specify the island for which a subquery is intended, the user encloses

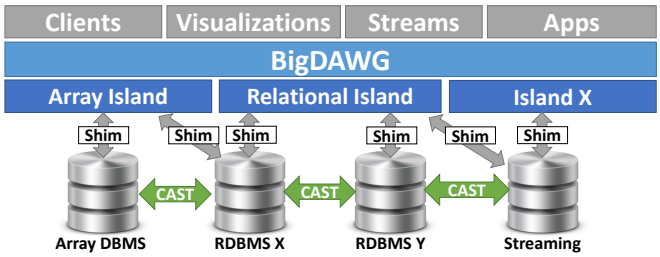

Figure 1: Polystore Architecture

his query in a *SCOPE* specification. A cross-island query will have multiple scopes to indicate the expected behavior of its subqueries. Likewise, a user may insert a CAST operation to denote when an object should be accessed with a given set of semantics.

For example, consider a cross-island operation, such as a join between an object in an array island and one in a table island. Obviously, we can CAST the array to the table island and do a relational join or we can do the converse and perform an array join. Since the each of these options produces a different output, a user must specify the semantics he desires using SCOPE and CAST commands. If he elects relational semantics, his query might be:

```
RELATIONAL(SELECT *
  FROM R, CAST(A, relation)
  WHERE R.v = A.v);
```

Here, the user specifies that the query will execute in a relational scope. From the user's perspective, the query will produce output as if A were translated into a table, shipped to a relational engine, and executed there. Because A is an array, the CAST converts it into a table when it is initially accessed. The user does not care whether the query is executed in the array store or relational one provided his prescribed island semantics are obeyed. Many queries may have implicit CASTS, and the polystore will insert these operations automatically as needed.

The full BigDAWG architecture is shown in Figure 1. This diagram shows multiple applications using a BigDAWG instance with three islands. Each island speaks to one or more engines through the use of shims. Objects can be CAST between engines, and only a subset of casts are shown for clarity. Here, a user issues a query to BigDAWG, and he specifies his commands using one or more island languages. Within an island, the polystore calls the shims needed to translate the query into the language(s) of the participating storage engines. When the query uses multiple islands, data may be shuffled among them using CAST operations.

As a result, the BigDAWG query language consists of the above SCOPE-CAST facility for a collection of islands over an overlapping set of storage engines. For simplicity, we leave the control of redundant copies of data objects for future work. In the rest of this paper we discuss our approach to query optimization and data placement.

3. QUERY OPTIMIZATION

In this section, we first introduce our approach to optimizing single-island queries. After that, we outline a mechanism for generating and managing the statistics needed for query optimization and data placement within a polystore. The section closes with a discussion of generalizing the optimizer to multi-island query planning.

3.1 Single Island Planning

Traditional query optimization is simply not capable of supporting cross-database queries. First, cost-based optimizers [19] require the planner to maintain a model for each operation to estimate its resource needs. This essentially obligates the optimizer to understand all of the operations in all storage engines as well as how the various shims work. Moreover, the primitives in each of the underlying storage engines may not map neatly to the operators in the island language. For example, a distributed array store may implement matrix multiplication with a purpose-built set of scatter-gather operations whereas a relational engine might categorize it as a group by aggregate. Reconciling these models would be non-trivial. Also, the optimizer would have to adapt whenever a new storage engine is added to the polystore. Lastly, conventional optimizers assume metadata about objects is available, such as their distribution of values. Local engines may or may not expose such information. As a result, we propose a black box approach in this section, whereby no information about the local optimizer is assumed.

If our query optimizer cannot be made robust using this approach, then we will selectively add more sophisticated knowledge of individual systems, recognizing that this may make adding new storage engines to a polystore more challenging. More detailed knowledge might include the sizes of operands, their data distribution, available access methods, and explanations of query plans.

We first consider simple queries, ones which have comparable performance on all of an island's storage engines. We anticipate that many select-project-join queries will be in this category and we will examine in how to identify such queries in Section 3.2.

Simple Queries For such queries we propose to minimize data movement among storage engines, so as to avoid costly data conversions and unnecessary network traffic. Rather, we should bring computation to the data whenever possible. Hence, we divide any simple query optimization into stages. In Stage 1 we perform all possible local computations that do not require any data movement. At the end of Stage 1, we are left with computations on collections of objects, where each one is located on a different storage engine. Since we have done all single-DBMS subqueries, we expect the number of remaining objects to be modest. In the next section we describe how to process this "remainder".

Complex Queries Now consider expensive operations, parts of the workload that have at least an order of magnitude

performance improvement relative to "one size fits all" architectures. Put differently, such operations are usually O(N^2) or O(N^3), and dwarf the cost of conventional data management commands, such as joins and aggregates. For example, the Genbase benchmark [22] demonstrated that complex analytics like linear regression enjoy a 10X speedup when run on an array store in comparison to a column store. For these queries, it will often make sense to move the data to a site where the high-performance implementation is available–which we learn empirically. The cost of the such moves will be more than amortized by the reduced execution time. To ensure a move is not too expensive, we will explore tactics for moving an object based on the engines involved, disk activity at both sites, and overall network activity.

In summary, we must pay careful attention to expensive operations. For polystores, we start by looking for select-project-join subqueries that are local to an object, and perform them in place as above. The "remainder" is expanded to include expensive functions, that we consider moving to a different site that offers high performance execution. The next section indicates how BigDAWG learns a preference list for assigning simple and complex queries to engines.

3.2 Workload Monitoring

To make efficient query optimization and data placement decisions, BigDAWG relies on black box performance profiling of its underlying systems. To rapidly acquire this information, the polystore query executor will have three modes of operation: training, optimized, and opportunistic. In training mode, BigDAWG has the liberty to run a subquery on all engines after inserting any casts that are needed. Optimized mode assigns the work of an incoming query to just one back end. Opportunistic mode measures the performance of the polystore engines by exploiting idle resources with an active learning approach.

Training For subqueries running in training mode, the federator records its elapsed time on each engine in an internal performance catalog. It will first run the leafs and branches from the AST at all possible locations in parallel to accumulate feedback. Since branches may involved multiple engines, there are many ways that a branch can be executed. Specifically, if there are N engines involved in a query, then there are $N!$ possibilities. In general we expect N to be small, and hence not a source of scalability issues.

Since training mode uses all engines, the optimizer will naturally have a comprehensive profile for each leaf and branch. Hence, it can develop a ranked *preference list* to determine the engine(s) best suited for each subquery. After that, new subqueries matching the characteristics of this subquery will need no additional experimentation.

Optimized If a query is run in optimized mode, then BigDAWG will execute no redundant computations for it. This mode is invoked in one of two cases: either the subquery has an existing preference list to guide its decision or there are not enough spare resources for expansive training. In the latter case, BigDAWG selects a database at random for query execution. Over time, the polystore builds up the same performance profiles offered more quickly by training mode.

Opportunistic BigDAWG maintains partial profiles on subqueries from optimized executions for further evaluation during periods of low system utilization. The system opportunistically executes the stored subqueries on new engines when their resources become available.

We expect a polystore to be in optimized mode most of the time or for it to start in training mode and gradually shift to optimized executions. In any case, over time BigDAWG assembles a database of subqueries and their duration on various engines. This monitoring framework is used to guide query optimization and data placement.

To accommodate complex operations, the experiments noted above will have a timeout. Hence, if an engine has not completed its trial within N multiples of the fastest running option it is halted. If objects are so large that data movement is problematic, we propose running a subquery on a statistical subset of the data to contain costs whenever possible. A subquery is then considered complex if there is significant variance in its runtime on different back ends.

3.3 Multi-Island Planning

We now consider queries that span multiple polystores. Here, the user is explicitly stating his semantics using CAST and SCOPE commands. Changing the order of these directives arbitrarily will generally result in ambiguous semantics in the query's execution. We will not support such indeterminateness. Instead, our main optimization opportunity is to identify circumstances where multiple islands have overlapping semantics. Consider an extension to the example query in Section 2.2:

```
RELATIONAL(SELECT count(*)
  FROM R, CAST(A, relation)
  WHERE R.v = A.v);
```

Here, the user issues the appropriate SCOPE-CAST query. This query, however, produces the same result, regardless of the join's island semantics. In other words, the join is sensitive to scope, but the subsequent aggregate masks these semantics. Hence, the query can be converted to the single-island query that is potentially amenable to traditional optimization. We will look for more instances where an island query may be simplified in this fashion.

We can also examine optimization opportunities between islands by analyzing their shims. For example, if two islands both have shims to the same back end, the optimizer may merge the output of their shims, deferring more of the query planning to a single storage engine's optimizer.

We will also investigate how to identify areas of intersection between multiple shims associated with the same storage engine. By identifying areas of the various island languages that produce the same query in the storage en-

gine's native dialect, we can rewrite the query in a single "intersection" island and further leverage the storage engine's query optimizer. More broadly, we will examine techniques for finding equivalences between the grammars of multiple islands. This will involve probing the space of queries in a manner similar as described for monitoring.

4. DATA PLACEMENT

Our polystore is an environment in which objects reside in some local execution engine. In a location transparent system it is possible to move objects without having to change any application logic. There are two reasons to move an object: load-balancing and optimization. In the former scenario, operations may take longer than necessary because one or more engines are overloaded. To shed load, objects must be relocated. In addition, for complex analytics the system would be more responsive if one or more objects were moved elsewhere because a faster implementation of the object's common workload is available. Since ultimately DBAs are in control of their data, our interface will have a mechanism for specifying that some objects are "sticky" and cannot be moved so that local control is supported. Non-sticky objects are available for relocation. For simplicity, we will initially consider a single client and later explore placement and load-balancing in the presence of concurrent queries. It is worth noting that if all access to underlying engines does not go through BigDAWG, then data placement decisions are likely to suboptimal.

We estimate the relative duration of each operation on each storage engine. From this, BigDAWG will predict the effect of placing object O_i on engine E_j. Its what-if analysis will estimate the effect of this move on both the performance of the source engine and any proposed destination. Hence, BigDAWG learns a matrix estimating the resource utilization of each object O_i on each storage engine E_j.

We initially assume that the various engines are not overloaded, and that early numbers represent an unloaded state. The polystore may identify overload situations by comparing the runtimes of queries to similar ones that were run in the past. For every engine E_j, we compute an average percentage deterioration of response time over all possible objects. When the percentage exceeds a threshold for a site E_j, we choose an object stored on that back end for relocation.

Our initial algorithm greedily selects the object O_i with maximum benefit to E_j and relocates it to the engine E_k with maximum headroom. Clearly this will not necessarily be the optimal choice, and we will explore other options.

So far, we have been assuming we are choosing the location of objects within a single island of information. As noted in the Section 2, there will typically be multiple islands with overlapping storage engines. This island abstraction imposes additional limitations on the operation of our polystore. Specifically, any object that is accessible in multiple islands may be moved to a different storage manager, but only to one with shims to all of its islands. Otherwise, some application programs may cease to work.

Each island maintains a set of metadata catalogs that contain the objects the island knows about along with the storage manager that houses that object. BigDAWG must be able to read all of these catalogs to implement its CAST-SCOPE mechanisms. As such, it is capable of identifying the collection of storage managers that may host a given object. This information is used to restrict data placement.

5. RELATED WORK

The federation of relational engines, or distributed DBMSs, was studied in depth during the 1980s and early 1990s. Here, Garlic [7], and TSIMMIS [9], and Mariposa [20] were early prototypes. The research community also explored distributed query processing, distributed transactions, and replica management. There was also research on federating databases with different schemas, and it focused on issues such as coalescing disparate schemas [4] and resolving semantics between applications [14]. More recently, Dataspaces were proposed as a mechanism for integrating data sources [12], however this approach has not seen large-scale adoption.

BigDAWG's approach is similar to the use of mediators in early federated database systems [7, 9, 23]. The use of shims to integrate individual databases is modeled on wrappers from these systems, and islands are similar to mediators in that they provide unified functionality over a set of disparate databases. However, mediators were often used to provide domain-specific functionality and did not span multiple data models. These systems focused on integrating independent sources, and did not consider the explicit controls, such as placement or replication, found in BigDAWG. We also will extend work on executing queries across several disjoint execution sites [5, 10]. This includes the ability to decompose queries into subqueries for partial execution [17], and adaptive query processing to handle uncertainty in database performance [2].

Historically, there has been limited interest in data federations inside the enterprise. Most enterprises have independent business units that are each in charge of their own data. To integrate disparate data sources, enterprises often use extract, transform and load (ETL) systems to construct a common schema, clean up any errors, transform attributes to common units, and remove duplicates. This *data curation* is an expensive process for constructing a unified database. In the past it was common practice to simply load curated information into a relational data warehouse for querying.

These data warehouses are incapable of dealing with real-time data feeds, as they are optimized for bulk loading. In addition, most are not particularly adept at text, semi-structured data, or operational updates. As such, we expect revitalized interest in federated databases as enterprises cope with these shortcomings.

Recently, many vendors have advocated "data lakes" wherein an organization dumps its data sources into a repository–often

based on the Hadoop ecosystem. However, Hadoop's lack of transactions, batch-oriented processing, and limited functionality means it is not often an ideal solution. As such, it is a plausible historical archive, but it would still require an additional polystore for the interactive and real-time components of the workload. Several recent research prototypes have explored multi-database systems that couple Hadoop with relational databases [11, 16].

6. SUMMARY

We believe that polystore systems will be a critical component as data management needs diversify. Here we presented our vision for a next-generation data federation, BigDAWG, that offers full functionality and location transparency through the use of explicit scopes and casts. With these primitives, we outlined tractable research agendas in query optimization and data placement.

7. REFERENCES

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

[2] L. Amsaleg, A. Tomasic, M. J. Franklin, and T. Urhan. Scrambling query plans to cope with unexpected delays. In *Fourth International Conference on Parallel and Distributed Information Systems, 1996*, pages 208–219. IEEE, 1996.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16. ACM, 2002.

[4] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[5] L. Bouganim, F. Fabret, C. Mohan, and P. Valduriez. A dynamic query processing architecture for data integration systems. *IEEE Data Eng. Bull.*, 23(2):42–48, 2000.

[6] P. G. Brown. Overview of scidb: large scale array storage, processing and analysis. In *SIGMOD*, pages 963–968. ACM, 2010.

[7] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, and D. Petkovic. Towards heterogeneous multimedia information systems: The Garlic approach. In *Data Engineering: Distributed Object Management*, pages 124–131. IEEE, 1995.

[8] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, and N. Tatbul. S-Store: A Streaming NewSQL System for Big Velocity Applications. *PVLDB*, 7(13), 2014.

[9] S. Chawathe, H. G. Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *IPSJ*, 1994.

[10] A. Deshpande and J. M. Hellerstein. Decoupled query optimization for federated database systems. In *ICDE*, pages 716–727. IEEE, 2002.

[11] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in polybase. SIGMOD, pages 1255–1266, 2013.

[12] M. Franklin, A. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *Sigmod Record*, 34(4):27–33, 2005.

[13] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, et al. Demonstration of the Myria big data management service. In *SIGMOD*. ACM, 2014.

[14] R. Hull. Managing semantic heterogeneity in databases: a theoretical prospective. In *PODS*, pages 51–61. ACM, 1997.

[15] J. Kepner, W. Arcand, W. Bergeron, N. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, and J. Kurz. Dynamic distributed dimensional data model (d4m) database and computation system. In *ICASSP*. IEEE, 2012.

[16] J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, J. Tatemura, N. Polyzotis, and M. J. Carey. MISO: souping up big data query processing with a multistore system. In *SIGMOD*, pages 1591–1602, 2014.

[17] L. M. Mackinnon, D. H. Marwick, and M. H. Williams. A model for query decomposition and answer construction in heterogeneous distributed database systems. *Journal of Intelligent Information Systems*, 11(1):69–87, 1998.

[18] M. Saeed, M. Villarroel, A. T. Reisner, G. Clifford, L.-W. Lehman, G. Moody, T. Heldt, T. H. Kyaw, B. Moody, and R. G. Mark. Multiparameter Intelligent Monitoring in Intensive Care II (MIMIC-II): A public-access intensive care unit database. *Critical Care Medicine*, 39:952–960, 2011.

[19] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34. ACM, 1979.

[20] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: a wide-area distributed database system. In *The VLDB Journal*, volume 5, pages 48–63. Springer, 1996.

[21] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose time has come and gone. In *ICDE*, pages 2–11, 2005.

[22] R. Taft, M. Vartak, N. R. Satish, N. Sundaram, S. Madden, and M. Stonebraker. Genbase: A complex analytics genomics benchmark. In *SIGMOD*, pages 177–188. ACM, 2014.

[23] G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, pages 38–49, 1992.