Inside "Big Data Management": Ogres, Onions, or Parfaits?

Vinayak Borkar
Computer Science Department
UC Irvine, USA

Michael J. Carey
Computer Science Department
UC Irvine, USA

Chen Li
Computer Science Department
UC Irvine, USA

ABSTRACT

In this paper we review the history of systems for managing "Big Data" as well as today's activities and architectures from the (perhaps biased) perspective of three "database guys" who have been watching this space for a number of years and are currently working together on "Big Data" problems. Our focus is on architectural issues, and particularly on the components and layers that have been developed recently (in open source and elsewhere) and on how they are being used (or abused) to tackle challenges posed by today's notion of "Big Data". Also covered is the approach we are taking in the ASTERIX project at UC Irvine, where we are developing our own set of answers to the questions of the "right" components and the "right" set of layers for taming the "Big Data" beast. We close by sharing our opinions on what some of the important open questions are in this area as well as our thoughts on how the data-intensive computing community might best seek out answers.

1. INTRODUCTION

The year is 2012, and everyone everywhere is buzzing about "Big Data". Virtually everyone, ranging from big Web companies to traditional enterprises to physical science researchers to social scientists, is either already experiencing or anticipating unprecedented growth in the amount of data available in their world, as well as new opportunities and great untapped value that successfully taming the "Big Data" beast will hold [9]. It is almost impossible to pick up an issue of anything from the trade press [8, 34], or even the popular press [53, 18, 28], without hearing something about "Big Data". Clearly it's a new era! Or is it...?

The database community has been all about "Big Data" since its inception, although the meaning of "Big " has obviously changed a great deal since the early 1980's when the work on parallel databases as we know them today was getting underway. Work in the database community continued until "shared nothing" parallel database systems were deployed commercially and fairly widely accepted in the mid-1990's. Most researchers in the database community then moved on to other problems. "Big Data" was reborn in the 2000's, with massive, Web-driven challenges of scale driving system developers at companies such as Google, Yahoo!, Amazon, Facebook, and others to develop new architectures for storing, accessing, and analyzing "Big Data". This rebirth caught most of the database commmunity napping with respect to parallelism, but now the database community has new energy and is starting to bring its expertise in storage, indexing, declarative languages, and set-oriented processing to bear on the problems of "Big Data" analysis and management.

In this paper we review the history of systems for managing "Big Data" as well as today's activities and architectures from the (perhaps biased) perspective of three "database guys" who have been watching this space for a number of years and are currently working together on "Big Data" problems. The remainder of this paper is organized as follows. In Section 2, we briefly review the history of systems for managing "Big Data" in two worlds, the older world of databases and the newer world of systems built for handling Web-scale data. Section 3 examines systems from both worlds from an architectural perspective, looking at the components and layers that have been developed in each world and the roles they play in "Big Data" management. Section 4 then argues for rethinking the layers by providing an overview of the approach being taken in the ASTERIX project at UC Irvine as well as touching on some related work elsewhere. Section 5 presents our views on what a few of the key open questions are today as well as on how the emerging data-intensive computing community might best go about tackling them effectively. Section 6 concludes the paper.

2. HISTORY OF BIG DATA

In the beginning, there was data – first in file sytems, and later in databases as the need for enterprise data management emerged [43] and as Tedd Codd's stone tablets, inscribed in 1970 with the rules of the relational model, began to gain commercial traction in the early 1980's. As for "Big Data", at least beyond the accumulation of scientific data, the need to manage "large" data volumes came later, first impacting the database world and then more recently impacting the systems community in a big way (literally).

2.1 Big Data in the Database World

In the database world, *a.k.a.* the enterprise data management world, "Big Data" problems arose when enterprises identifed a need to create data warehouses to house their historical business data and to run large relational queries over that data for business analysis and reporting purposes. Early work on support for storage and efficient analysis of such data led to research in the late 1970's on "database machines" that could be dedicated to such purposes. Early database machine proposals involved a mix of novel hardware architectures and designs for prehistoric parallel query processing techniques [37]. Within a few years it became clear that neither brute force scan-based parallelism nor proprietary hardware would become sensible substitutes for good software data structures and algorithms. This realization, in the early 1980's, led to the first generation of software-based parallel databases based on the architecture now commonly referred to as "shared-nothing" [26].

The architecture of a shared-nothing parallel database system, as the name implies, is based on the use of a networked cluster of individual machines each with their own private processors, main memories, and disks. All inter-machine coordination and data communication is accomplished via message passing. Notable first-generation parallel database systems included the Gamma system from the University of Wisconsin [27], the GRACE system from the University of Tokyo [29], and the Teradata system [44], the first successful commercial parallel database system (and still arguably the industry leader nearly thirty years later). These systems exploited the declarative, set-oriented nature of relational query languages and pioneered the use of divide-and-conquer parallelism based on hashing in order to partition data for storage as well as relational operator execution for query processing. A number of other relational database vendors, including IBM [13], successfully created products based on this architecture, and the last few years have seen a new generation of such systems (e.g., Netezza, Aster Data, Datallegro, Greenplum, Vertica, and ParAccel). Many of these new systems have recently been acquired by major hardware/software vendors for impressively large sums of money, presumably driven in part by "Big Data" fever.

So what makes "Big Data" big, i.e., just how big is "Big"? The answer is obviously something that has evolved over the thirty-year evolution of "Big Data" support in the database world. One major milestone occurred on June 2, 1986, the day on which Teradata shipped the first parallel database system (hardware and software) with 1 terabyte of storage capacity to Kmart, to power their then-formidable data warehouse, in a large North American Van Lines tractor-trailer truck [51]. In comparison, some of today's very large data warehouses (e.g., the data warehouse at eBay) involve multiple parallel databases and contain tens of petabytes of data.

Another form of "Big Data" suppport also emerged concurrently with the aforementioned query-oriented systems. Transaction processing (TP) systems underlie the online applications that power businesses' day-to-day activities and have been the main producers of the large volumes of data that are filling data warehouses. TP systems have also seen tremendous growth in demand, albeit more for processing power than for data storage. The same sort of shared-nothing architecture emerged as a significant contender on the TP side of the database world, most notably Tandem's NonStop SQL system [48] in the 1980's. To get a sense of what "Big" was and how it has evolved, Jim Gray and a collection of collaborators in the TP performance measurement world noted in 1985 that a transaction rate of 15 simple DebitCredit transactions per second was "common", 50 TPS was a "good" system, 100 TPS was "fast", and a 400 TPS system was characterized as being a very "lean and mean" system [10]. It was not until two years later that Jim and the NonStop SQL team achieved the "phenomenal result" of exceeding 200 TPS on a SQL relational DBMS [7]. In comparison, today's research prototypes on the TP side are hitting hundreds of thousands of (much heavier) transactions per second.

2.2 Big Data in the Systems World

In the distributed systems world, "Big Data" started to become a major issue in the late 1990's due to the impact of the world-wide Web and a resulting need to index and query its rapidly mushrooming content. Database technology (including parallel databases) was considered for the task, but was found to be neither well-suited nor cost-effective [17] for those purposes. The turn of the millenium then brought further challenges as companies began to use information such as the topology of the Web and users' search histories in order to provide increasingly useful search results, as well as more effectively-targeted advertising to display alongside and fund those results.

Google's technical response to the challenges of Web-scale data management and analysis was simple, by database standards, but kicked off what has become the modern "Big Data" revolution in the systems world (which has spilled back over into the database world). To handle the challenge of Web-scale storage, the Google File System (GFS) was created [31]. GFS provides clients with the familiar OS-level byte-stream abstraction, but it does so for extremely large files whose content can span hundreds of machines in shared-nothing clusters created using inexpensive commodity hardware. To handle the challenge of processing the data in such large files, Google pioneered its MapReduce programming model and platform [23]. This model, characterized by some as "parallel programming for dummies", enabled Google's developers to process large collections of data by writing two user-defined functions, map and reduce, that the MapReduce framework applies to the instances (map) and sorted groups of instances that share a common key (reduce) – similar to the sort of partitioned parallelism utilized in shared-nothing parallel query processing.

Driven by very similar requirements, software developers at Yahoo!, Facebook, and other large Web companies followed suit. Taking Google's GFS and MapReduce papers as rough technical specifications, open-source equivalents were developed, and the Apache Hadoop MapReduce platform and its underlying file system (HDFS, the Hadoop Distributed File System) were born [2]. The Hadoop system has quickly gained traction, and it is now widely used for use cases including Web indexing, clickstream and log analysis, and certain large-scale information extraction and machine learning tasks. Soon tired of the low-level nature of the MapReduce programming model, the Hadoop community developed a set of higher-level declarative languages for writing queries and data analysis pipelines that are compiled into MapReduce jobs and then executed on the Hadoop MapReduce platform. Popular languages include Pig from Yahoo! [40], Jaql from IBM [5], and Hive from Facebook [4]. Pig is relational-algebra-like in nature, and is reportedly used for over 60% of Yahoo!'s MapReduce use cases; Hive is SQL-inspired and reported to be used for over 90% of the Facebook MapReduce use cases.

Not surprisingly, several divisions of Microsoft have faced similar requirements, so Microsoft developers have developed their own "Big Data" technologies in response. Microsoft's technologies include a parallel runtime system called Dryad [35] and two higher-level programming models, DryadLINQ [55] and the SQL-like SCOPE language [21], which utilize Dryad under the covers. Interestingly, Microsoft has also recently announced that its future "Big Data" strategy includes support for Hadoop.

We explained in the previous subsection that the database world faced both analytic and transactional "Big Data" challenges. The systems world has faced a similar dichotomy. In addition to MapReduce and Hadoop for analytics, the developers of very large scale user-facing Web sites and Web services have found a need for simple key-value stores that are fast, highly scalable, and reliable. At Google, this need led to the development of Bigtable, their massively scalable (and schema-flexible) table store built on top of GFS [22]. At Amazon, this need led to the development of Dynamo, which is Amazon's highly scalable and eventually consistent key-value store [25]. Again, driven by very similar requirements, the Apache open-source community has developed both HBase [3], an open-source Bigtable clone, and Cassandra [1], an open-source

Dynamo clone.

2.3 Big Data Today

Companies facing "Big Data" challenges today have multiple options. If a company faces traditional large query challenges, coupled with a traditional IT budget, they can opt to license a parallel database system from the database world. Unfortunately, as of today, despite the construction over the years of multiple parallel database system prototypes, no open-source parallel database offering exists. If a company has batch-style semistructured data analysis challenges, they can instead opt to enter the Hadoop world by utilizing one or several of the open-source technologies from that world. A lively early "parallel databases vs. Hadoop" debate captured the field's attention in the 2008-2009 timeframe and was nicely summarized in 2010 in a pair of papers written by the key players from the opposing sides of the debate [46, 24].

If a company instead has fairly simple key-based data storage and retrievel needs, they can choose to adopt one of the new open-source key-value stores from the systems world. Alternatively, they can opt to rely on application-level "sharding", addressing scaling by manually hash-partitioning their data storage and retrieval operations over a set of open-source MySQL or Postgres database instances. A more comprehensive treatment of the latter options (and the "NoSQL movement") can be found in [19].

3. ONIONS AND OGRES

We have briefly reviewed the histories of the database world and systems world regarding their responses to "Big Data" challenges. In this section we evaluate their architectural pros and cons.

Shrek: For your information, there's a lot more to ogres than people think.
Donkey: Example?
Shrek: Example... uh... ogres are like onions!
(holds up an onion, which Donkey sniffs)
Donkey: They stink?
Shrek: Yes... No!
Donkey: Oh, they make you cry?
Shrek: No!
Donkey: Oh, you leave 'em out in the sun, they get all brown, start sproutin' little white hairs...
Shrek: *(peels an onion)* NO! Layers. Onions have layers. Ogres have layers. Onions have layers. You get it? We both have layers.
(walks off)
Donkey: Oh, you both have LAYERS. Oh. You know, not everybody like onions. What about cake? Everybody loves cake!
Shrek: I don't care what everyone else likes! Ogres are not like cakes.
– from the 2001 Dreamworks movie "Shrek" [6]

3.1 Onions: Parallel Database Systems

Parallel database systems are like onions? Yes, indeed they are! Onions have layers, parallel databases have layers.

Figure 1 illustrates the layers found in the software architecture of a typical shared-nothing parallel SQL database system. At the bottom of the parallel database software stack is a record-oriented storage layer; this layer is made up of a set of local (row-oriented or column-oriented) storage managers, with one such storage manager per machine in a cluster. These local storage managers are orchestrated by the layers above to deliver partitioned, shared-nothing storage services for large relational tables. In the middle of the stack is a relational dataflow runtime layer that contains the system's collection of relational operators and utilizes them to run the query plans determined by the topmost layer of the stack. Lastly, the top layer is a SQL compiler, which accepts queries, optimizes them, and generates appropriate query plans for execution by the dataflow runtime layer. Parallel database applications have exactly one way into the system's architecture: SQL, which is available at the very top of the stack.

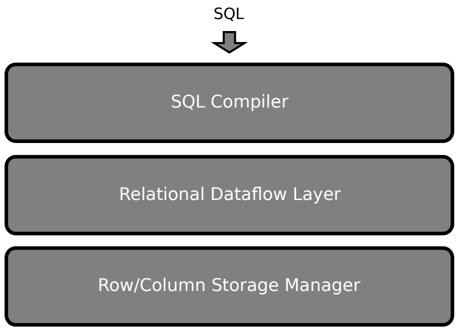

Figure 1: Parallel database software stack

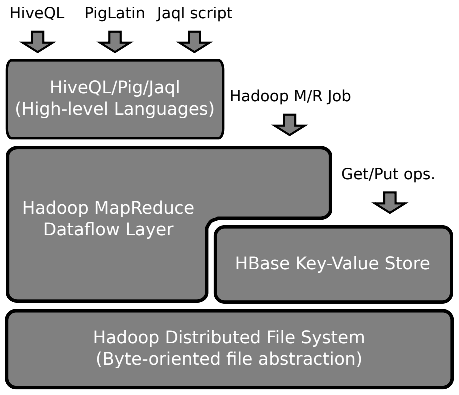

Figure 2: Hadoop software stack

Parallel database systems are actually like onions in several other ways as well. Not only do they have layers, but their users can see only the outermost layer (SQL). They are monolithic in nature – you can't safely cut into them if you just want access to the functionality of an inner layer. Also, like onions, they have been known to make people cry – most commonly when they look at their licensing costs, since there are no open source parallel database systems available today.

3.2 Ogres: Open-Source Big Data Stacks

Open-source "Big Data" stacks are like ogres? Yes, in our opinion they are. They are awkwardly assembled, as if by accident, and are quite strong but can also be clumsy. And like ogres, at times, they stink – at least for some tasks.

Figure 2 illustrates the layers found in the software architecture of today's Hadoop stack. At the bottom of the Hadoop software stack is HDFS, a distributed file system in which each file appears as a (very large) contiguous and randomly addressible sequence of bytes. For batch analytics, the middle layer of the stack is the Hadoop MapReduce system, which applies map operations to the data in partitions of an HDFS file, sorts and redistributes the results based on key values in the output data, and then performs reduce operations on the groups of output data items with matching keys from the map phase of the job. For applications just needing basic key-based record management operations, the HBase store (layered on top of HDFS) is available as a key-value layer in the Hadoop stack. As indicated in the figure, the contents of HBase can either be directly accessed and manipulated by a client application or accessed via Hadoop for analytical needs. Finally, as mentioned earlier, in the History section, many clients of the Hadoop stack prefer the use of a declarative language over the bare MapReduce programming model. High-level language compilers are thus the topmost layer in the Hadoop software stack for such clients, as indicated on the left-hand side of Figure 2.

3.3 Are Ogres Good or Evil?

Watching the trends in the industry, it seems that ogres (variations on the Hadoop stack) are preferable to onions (monolithic parallel SQL database systems) for a number of modern "Big Data" use cases. But are ogres actually the "right" answer? We believe not. We believe instead that, while the ogres of today are important and even convenient, the "right" answer still lies ahead: The database community should be making and serving the world parfaits by bringing our years of experience with data management and scalable data systems to bear on defining the next generation of "Big Data" management and analysis systems. In order to do this, we need to assess the ogres in light of what we have learned from growing and cooking with our onions.

We believe that today's Hadoop stack has a number of strongly desirable properties that we need to learn from and internalize:

1. Open source availability.
2. Non-monolithic layers or components.
3. Support for access to file-based external data (rather than mandating data loading).
4. Support for automatic and incremental forward-recovery of jobs with failed tasks.
5. Ability to schedule very large jobs in smaller chunks, at least under high loads, rather than being forced to completely co-schedule entire large pipelines.
6. Automatic data placement and rebalancing as data grows and machines come and go.
7. Support for replication and machine fail-over without operator intervention.

We believe that today's Hadoop stack also has a number undesirable properties that should be avoided in future "Big Data" stacks.

1. Similar to problems cited years ago in [45], which discussed the appropriateness of OS file systems for database storage, it makes little sense to layer a record-oriented data abstraction on top of a giant globally-sequenced byte-stream file abstraction. For example, because HDFS is unaware of record boundaries, instead of dealing with fixed-length file splits, there will be a "broken record" – a record with some of its bytes in one split and some in the next – with very high probability at the end of each partition in a typical HDFS file.
2. Also similar is the questionable amount of sense associated with building a parallel data runtime on top of an impoverished unary operator model (map, reduce, and perhaps combine). For example, look at what it takes to perform operations such as joins when operating strictly in the world according to MapReduce [15, 30].
3. Building a key-value store layer with possibly remote query access happening at the next layer up makes little sense, as it places a potentially remote boundary at one of the worst possible places in an efficient data management architecture [33]. That is, pushing queries down to data is likely to outperform pulling data up to queries.
4. A general lack of schema information, typical today, is flexible but also a recipe for future difficulties. For example, future maintainers of applications that today's developers are building are likely to have a very difficult time finding and fixing bugs that might be related to changes in or assumptions about the structure of some data file in HDFS. (This was one of the very early lessons in the database world [43].)
5. Based on our recent experiences measuring current "Big Data" stack performance, it seems as though open-source stack implementators have forgotten almost everything about single system performance, focusing solely on scale-out. Recall from the History section that database machine researchers discovered early on that basic single-system performance is also critical; this lesson was reiterated in [42, 46] and needs to be remembered going forward.

It is our feeling that many academics are being too "shy" about questioning and rethinking the "Big Data" stack forming around Hadoop today, perhaps because they are embarrassed about having fallen asleep in the mid-1990's (with respect to parallel data issues) and waking up in the world as it exists today. Rather than trying to modify the Hadoop code base to add indexing or data co-clustering support, or gluing open-source database systems underneath Hadoop's data input APIs, we believe that database researchers should be asking "Why?" or "What if we'd meant to design an open software stack with records at the bottom and a strong possibility of a higher-level language API at the top?"

4. MAKING PARFAITS

Donkey: You know what ELSE everybody likes? Parfaits! Have you ever met a person, you say, "Let's get some parfait," they say, "Hell no, I don't like no parfait"? Parfaits are delicious!
Shrek: NO! You dense, irritating, miniature beast of burden! Ogres are like onions! End of story! Bye-bye! See ya later.
Donkey: Parfait's gotta be the most delicious thing on the whole damn planet!
– from the 2001 Dreamworks movie "Shrek" [6]

4.1 The ASTERIX Parfait

The ASTERIX project at UC Irvine began in early 2009 with the objective of creating a new parallel, semistructured information management system. In the process of designing and building the system, three distinct and reusable architectural layers have emerged in what we hope will someday be viewed as the "ASTERIX parfait". Figure 3 summarizes the layers, and this section of the paper explains what each layer in the ASTERIX stack does, roughly how it does it, and also what its potential value is as an open-source software resource.

The bottom-most layer of the ASTERIX stack is a data-intensive runtime called Hyracks [16]. Hyracks sits at roughly the same level of the architecture that Hadoop does in the previous section's discussion of higher-level data analysis languages such as Pig and

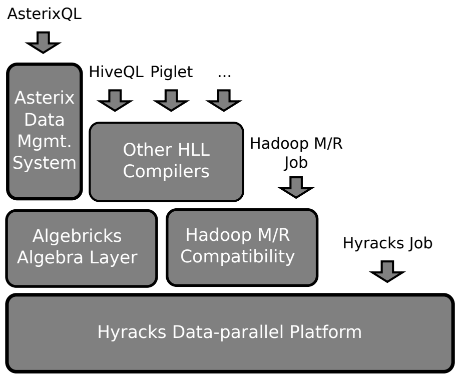

Figure 3: ASTERIX software stack

Hive. The topmost layer of the ASTERIX stack is a full parallel DBMS with a flexible data model (ADM) and query language (AQL) for describing, querying, and analyzing data. AQL is comparable to languages such as Pig, Hive, or Jaql, but ADM and AQL support both native storage and indexing of data as well as access to external data residing in a distributed file system (e.g., HDFS). Last but not least, in between these two layers sits Algebricks, a model-agnostic, algebraic "virtual machine" for parallel query processing and optimization. Algebricks is the target for AQL query compilation, but it can also be the target for other declarative data languages (e.g., we have HiveQL compiling on top of Algebricks today, and we also have a small Pig subset called Piglet to help show Algebricks users how to use Algebricks when implementing a new language).

4.2 The Hyracks Runtime Layer

The Hyracks layer of ASTERIX is the bottom-most layer of the stack. Hyracks is the runtime layer whose job is to accept and manage data-parallel computations requested either by direct end-users of Hyracks or (more likely) by the layers above it in the ASTERIX software stack.

Jobs are submitted to Hyracks in the form of directed acyclic graphs that are made up of "Operators" and "Connectors". To illustrate the key ideas in Hyracks via a familiar example, Figure 4 shows an example Hyracks job representing a TPCH-like query that performs a join between a partitioned file containing Customer records and another partitioned file containing Order records. The result of the join is aggregated to count the popularities of orders by market segment. Nodes in the figure represent Operators, and edges represent Connectors. In Hyracks, Operators are responsible for consuming partitions of their inputs and producing output partitions. Connectors perform redistribution of data between different partitions of the same logical dataset. For example, in Figure 4, the file scanners that scan the Customer and Order files are each connected to the Join Operator by means of an M:N Hash Partitioning Connector. This Connector ensures that all Customer (Order) records reaching the Join Operator partitions agree on the hash-value of the Customer's (Order's) CID attribute, thereby enabling each partition of the Join Operator to peform a local join to produce output partitions. In order to perform aggregation on the MKT_SEGMENT attribute, the Join Operator's output partitions are redistributed to the partitions of the GroupBy Operator using another M:N Hash Partitioning Connector; this one hashes on the MKT_SEGMENT attribute to ensure that all records that match on the grouping attribute are directed to the same grouping partition. Finally, the GroupBy Operator's output is written to a file by the FileWriter Operator. The use of a 1:1 Connector between the GroupBy Operator and the FileWriter Operator results in the creation of as many result partition files as GroupBy Operator partitions.

Figure 5 illustrates the first step in the execution of the submitted Job by Hyracks. Operators in Hyracks are first expanded into their constituent activities. For example, the Join Operator in the example is made up of two activities, the Join Build Activity and the Join Probe Activity. This expansion is made possible by APIs that Hyracks provides to Operator implementors to enable them to describe such behavioral aspects of an operator. As shown in Figure 5, the Join Build Activity has a dashed edge to the Join Probe Activity, which represents the blocking of the probe operation until the build operation is complete. Although Hyracks does not understand the specifics of the various activities of an operator, exposing the blocking characteristics of Operators provides important scheduling information to the system. Also note that the GroupBy Operator is similarly made up of two activities, which denotes the constraint that it can produce no output until all of its input has been consumed.

Hyracks analyzes the Activity graph produced by the expansion described above to identify the collections of Activities (Stages) that can be executed at any time (while adhering to the blocking requirements of the Operators). Each Stage is parallelized and executed in the order of dependencies. Figure 6 shows the partitioned runtime graph of the example. The dashed polygons around the Operator Partitions represent the stages that were inferred in the previous step of job execution. More details about Hyracks' computational model, as well as about its implementation and perormance, are available in [16].

Hadoop [11] has quickly become a "gold-standard" in industry as a highly scalable data-intensive MapReduce platform. Any system that hopes to displace it must provide a low-cost migration path for existing Hadoop artifacts. To that end, we have built an adapter on top of Hyracks that accepts and executes Hadoop MapReduce jobs without requiring code changes from the user. More details about the Hadoop compatibility layer can be found in [16]. Based in part on this layer, we have also conducted experiments to measure the relative performance of Hadoop and Hyracks. In Figure 7 we show running times for the TPCH-like example query as the

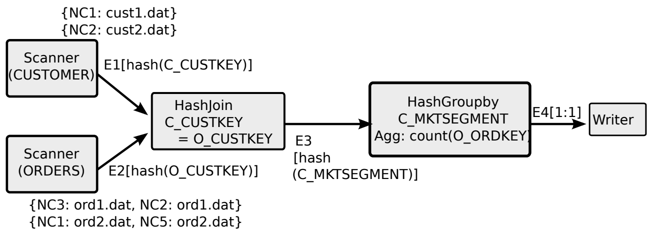

Figure 4: Example Hyracks Job specification

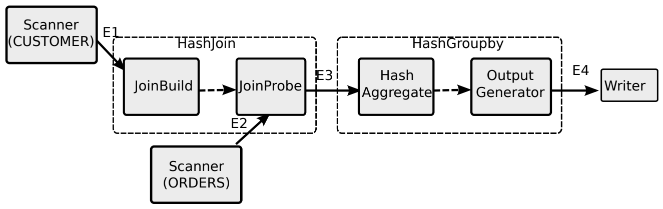

Figure 5: Example Hyracks Activity Node graph

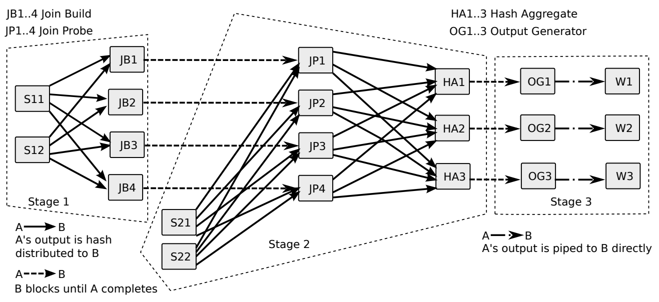

Figure 6: Parallel instantiation of the example

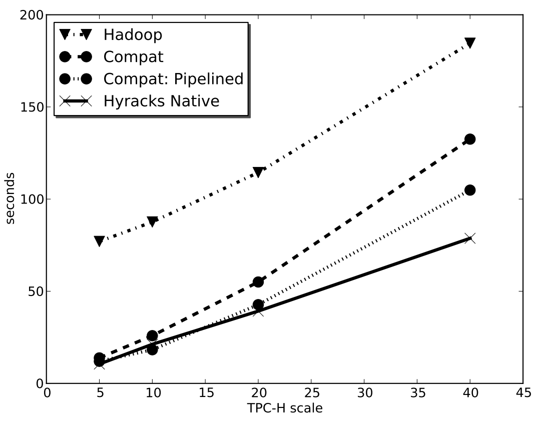

Figure 7: TPC-H query performance

amount of data increases, running natively both on Hadoop and on Hyracks. To shed light on where the gains come from, we also show running times measured when running the Hadoop MapReduce job on Hyracks using its Hadoop compatibility layer. Due to a tighter platform implementation, the MapReduce job running on the Hadoop compatibility layer of Hyracks already performs better than Hadoop. In addition, the native Hyracks formulation of the query shows additional performance improvement as the input data size grows. More experiments, as well as a careful analysis of their results, can be found in [16].

4.3 The Algebricks Algebra Layer

Algebricks is a model-agnostic, algebraic layer for parallel query processing and optimization. This layer's origin was as the center of the AQL compiler and optimizer of the ASTERIX system. However, it soon became clear that such a layer could benefit other high-level declarative language implementations for data-parallel computation, and Algebricks was reborn as a public layer in its own right.

To be useful for implementing arbitrary languages, Algebricks has been carefully designed to be agnostic of the data model of the data that it processes. Logically, operators operate on collections of tuples containing data values. The data values carried inside a tuple are not specified by the Algebricks toolkit; the language implementor is free to define any value types as abstract data types. For example, a user implementing a SQL compiler on top of Algebricks would define SQL's scalar data types to be the data model and would implement interfaces to perform operations such as comparison and hashing. ASTERIX has a richer set of data types, and these have been implemented on top of the Algebricks API as well.

Algebricks consists of the following parts:

1. A set of logical operators,
2. A set of physical operators,
3. A rewrite rule framework,
4. A set of generally applicable rewrite rules,
5. A metadata provider API that exposes metadata (catalog) information to Algebricks, and,
6. A mapping of physical operators to the runtime operators in Hyracks.

A typical declarative language compiler parses a user's query and then translates it into an algebraic form. When using Algebricks, the compiler uses the provided set of logical operators as nodes in a directed acyclic graph to form the algebraic representation of the query. This DAG is handed to the Algebricks layer to be optimized, parallelized, and code-generated into runnable Hyracks operators.

Algebricks provides all of the traditional relational operators [43] such as **select**, **project**, and **join**. In addition, Algebricks enables the expression of correlated queries through the use of a **subplan** operator. The **groupby** operator in Algebricks allows complete nested plans to be applied to each group.

The Algebricks optimizer uses Logical-to-Logical rewrite rules to create alternate logical formulations of the initially provided DAG. Logical-to-Physical rewrite rules then generate a DAG of physical operators that specify the algorithms to use to evaluate the query. For example, a **join** operator might be rewritten to a **hash-join** physical operator. This process of rewriting into a physical DAG uses partitioning properties and local physical properties of input data sources. Data **exchange** operators [32] are used to perform redistribution of data between partitions.

We expect that most language implementers using Algebricks will need a rewriting framework to perform additional useful optimizations. The Algebricks toolkit contains a rewriting framework that allows users to write their own rewrite rules but also comes with a number of "out of the box" rules that the user can choose to reuse for compiling their high-level language. Examples of rules that seem likely to apply for most languages include:

- **Push Selects**: Rule for pushing filters lower in the plan to eliminate data that is not useful to the query.
- **Introducing Projects**: Rule to limit the width of intermediate tuples by eliminating values that are no longer needed in the plan.
- **Query Decorrelation**: Rule to decorrelate nested queries to use joins when possible.

We are on a path to demonstrating the general applicability of Algebricks by using it to build multiple languages ourselves as well as interacting with outside groups with similar desires. The top layer of ASTERIX, based on ADM and AQL, is a data management system that is built on top of Algebricks. In addition, as a strong proof of concept, we have ported the Hive [12] compiler from Facebook, converting it to generate an Algebricks DAG, which is then optimized and executed on Hyracks. The decision to carve out Algebricks was also motivated in part as a result of feedback from potential collaborators. A team at Yahoo! research is starting to use Algebricks to implement a declarative language for machine learning [52], and a team at the San Diego Supercomputing Center is working to use Algebricks to implement a query processor to process queries over large RDF graphs.

4.4 The ASTERIX System Layer

The topmost layer of the ASTERIX software stack, the original project goal, is the ASTERIX parallel information management system itself (or ASTERIX, for short), pictured at a high level in Figure 8. Figure 9 indicates how the software components of ASTERIX map to the nodes in a cluster and indicates how Hyracks serves as the runtime executor for query execution and storage management operations in ASTERIX. Figure 10 provides a diagrammatic overview of an ASTERIX metadata node with a focus on its query planning and execution components.

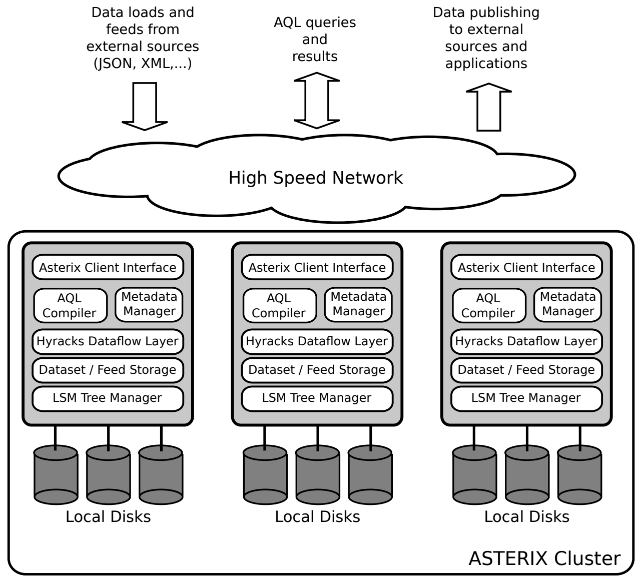

Figure 8: ASTERIX system overview

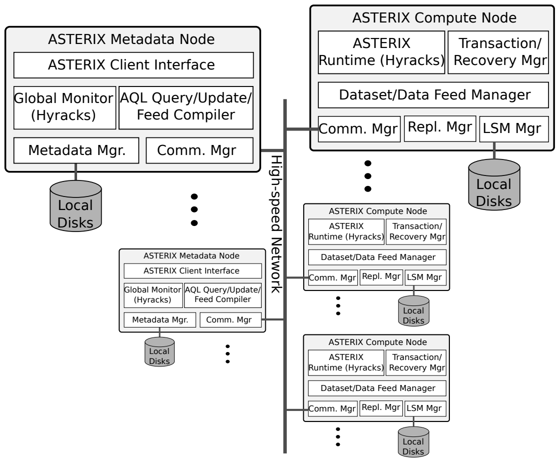

Figure 9: ASTERIX system schematic

Data in ASTERIX is based on a semistructured data model. As a result, ASTERIX is well-suited to handling use cases ranging from rigid, relation-like data collections, whose types are well understood and invariant, to flexible and potentially more complex data where little is known a priori and the instances in data collections are highly variant and self-describing. The ASTERIX data model (ADM) is based on borrowing the data concepts from JSON [36] and adding additional primitive types as well as type constructors

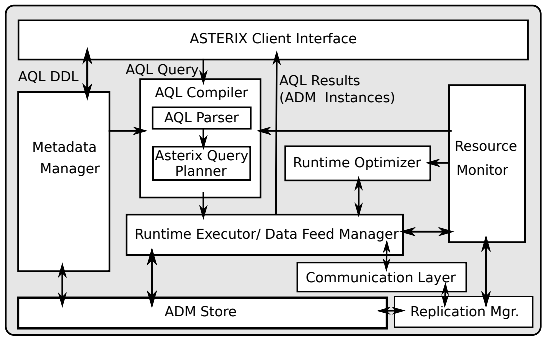

Figure 10: ASTERIX query processing schematic

borrowed from object databases [20, 39]. Figure 11(a) illustrates ADM by showing how it could be used to define a record type for modeling events being put on by special interest groups. The record type shown there is an "open" type, meaning that all instances of this type will conform to its type specification but may also contain arbitrary additional fields that can vary from one instance to the next. The example also shows how ADM includes such features as nested records (location), optional fields with specified types (price and location), nested record sets (organizers), nested record lists (sponsoring_sigs), and nested sets of primitive values (interest_keywords).

Figure 11(d) shows an example of what a set of data instances of type EventType would look like. Data storage in ASTERIX is based on the concept of a dataset, a declared collection of instances of a given type. ASTERIX supports both system-managed datasets (like the Event dataset declared in Figure 11(a)), which are internally stored and managed by ASTERIX as partitioned LSM B+ trees [41] with optional LSM-based secondary indexes, and external datasets, where the data can reside in existing HDFS files or collections of files in the cluster nodes' local file systems.

ASTERIX queries are written in AQL (the ASTERIX Query Language), a declarative query language that we designed by taking the essence of XQuery [54], most importantly its FLWOR expression constructs and its composability, and simplifying and adapting it to query the types and modeling constructs of ADM (vs. XML). Figure 11(b) illustrates AQL via an example. This query runs over the Event dataset, containing EventType instances, to compute the top five events sponsored by special interest groups, along with the number of events they have sponsored, both cumulatively and broken down by chapter. Figure 11(c) shows what this query's results would be when run on the sample data shown in Figure 11(d).

To process queries like the one in Figure 11(b), ASTERIX compiles an AQL query into an Algebricks program. This program is then optimized via algebraic rewrite rules that reorder the Algebricks operators as well as introducing partitioned parallelism for scalable execution, after which code generation translates the resulting physical query plan into a corresponding Hyracks job. The resulting Hyracks job uses the operators and connectors of Hyracks to compute the desired query result. Figure 12 shows what the Hyracks job resulting from the compilation of the example query would look like. (Some of the operators shown in the figure's Hyracks job are primitive operators from the Hyracks library, while others are instances of a meta-operator that permits pipelines of smaller Algebricks operators to be bundled into a single Hyracks operator for efficiency.)

ASTERIX is targeting use cases of archiving, querying, and anal-

```
declare open type EventType as {
  event_id: int32,
  name: string,
  location: AddressType?,
  organizers: {{ {
    name: string,
    role: string
  } }},
  sponsoring_sigs: [ {
    sig_name: string,
    chapter_name: string
  } ],
  interest_keywords: {{string}},
  price: decimal?,
  start_time: datetime,
  end_time: datetime
}

create dataset Event (EventType)
      partitioned by key event_id;
```

(a) AQL Schema for EventType

```
for $event in dataset('Event')
for $sponsor in $event.sponsoring_sigs
let $es := {
    "event": $event,
    "sponsor": $sponsor
}
group by $sig_id := $sponsor.sig_id
     with $es
let $sig_sponsorship_count := count($es)
let $by_chapter :=
    for $e in $es
    group by $chapter_name :=
        $e.sponsor.chapter_name with $es
    return { "chapter_name": $chapter_name,
                "count": count($es) }
order by $sig_sponsorship_count desc
limit 5
return {
    "sig_id": $sig_id,
    "total_count": $sig_sponsorship_count,
    "chapter_breakdown": $by_chapter
}
```

(b) Example aggregation query

```
[ {
    "sig_id": 14,
    "total_count": 3,
    "chapter_breakdown": [
        { "chapter_name": "San Clemente", "count": 1 },
        { "chapter_name": "Laguna Beach", "count": 2 }
    ]
  },
  {
    "sig_id": 31,
    "total_count": 1,
    "chapter_breakdown": [
        { "chapter_name": "Huntington Beach", "count": 1 }
    ]
  } ]
```

(c) Result of aggregation query

```
{{ {
    "event_id": 1023,
    "name": "Art Opening: Southern Orange County Beaches",
    "organizers": {{ { "name": "Jane Smith" } }}
    "sponsoring_sigs": [
        { "sig_id": 14, "chapter_name": "San Clemente" },
        { "sig_id": 14, "chapter_name": "Laguna Beach" }
    ],
    "interest_keywords": {{
        "art",
        "landscape",
        "nature",
        "vernissage"
    }}
    "start_time": datetime( "2011-02-23T18:00:00:000-08:00" ),
    "end_time": datetime( "2011-02-23T21:00:00:000-08:00" )
  },
  {
    "event_id": 941,
    "name": "Intro to Scuba Diving",
    "organizers": {{
        {
            "name": "Joseph Surfer",
            "affiliation": "Huntington Beach Scuba Assoc."
        }
    }}
    "sponsoring_sigs": [ {
        "sig_id": 31,
        "chapter_name": "Huntington Beach"
    } ],
    "interest_keywords": {{ "scuba", "diving", "aquatics" }}
    "price": 40.00,
    "start_time": datetime( "2010-10-16T9:00:00:000-08:00" ),
    "end_time": datetime( "2010-10-16T12:00:00:000-08:00" )
  },
  {
    "event_id": 1042,
    "name": "Orange County Landmarks",
    "organizers": {{ { "name": "John Smith" } }}
    "sponsoring_sigs": [ {
        "sig_id": 14,
        "chapter_name": "Laguna Beach"
    } ],
    "interest_keywords": {{ "architecture", "photography" }}
    "price": 10.00,
    "start_time": datetime( "2011-02-23T17:00:00:000-08:00" ),
    "end_time": datetime( "2011-02-23T19:00:00:000-08:00" )
  } }}
```

(d) Example Event data

```
for $event1 in dataset('Event')
for $event2 in dataset('Event')
where $event1.interest_keywords ~= $event2.interest_keywords
      and $event1.event_id != $event2.event_id
return { 'event1': $event1, 'event2': $event2 }
```

(e) Example Fuzzy-Join query

Figure 11: Example AQL schemas, queries, and results

ysis of semistructured data drawn from Web sources (e.g., Twitter, social networking sites, event calendar sites, and so on), so it is a given that some of the incoming data will be dirty. Fuzzy matching is thus a key feature of ASTERIX that we are very actively developing. In addition, analyzing such data, such as to make recommendations or to identify sub-populations and trends in social networks, often requires the matching of multiple sets of data based on set-similarity measures. For all these reasons, AQL includes an approximate matching capability, as illustrated by the example query shown in Figure 11(e). This example AQL query finds the Events that are similar in terms of their associated interest keyword sets, and such queries are executed in parallel based on principles that were developed [49] while studying how to perform fuzzy joins in the context of Hadoop [50]. AQL has default similarity metrics

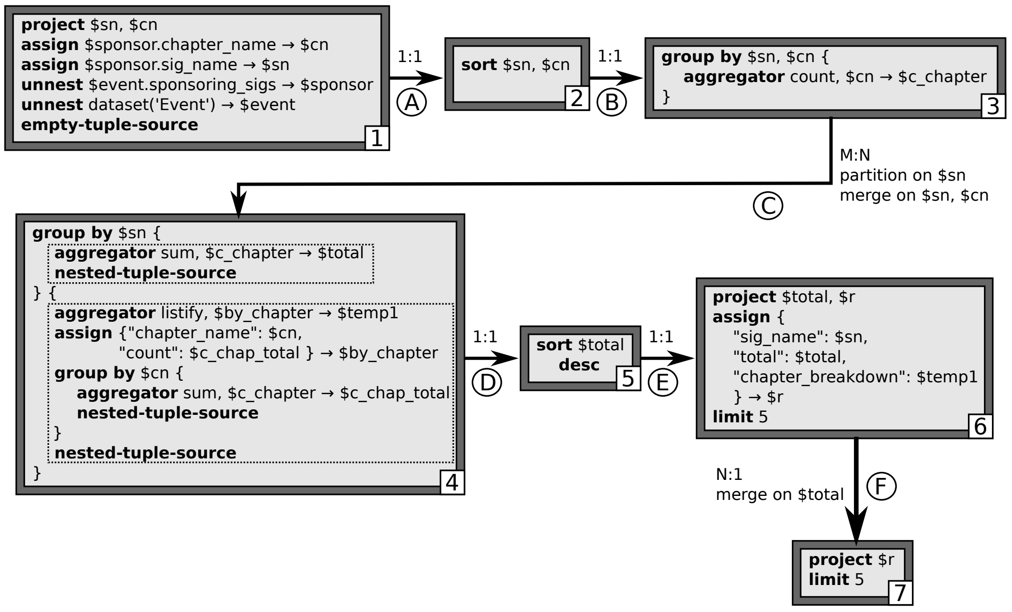
project $sn, $cn
assign $sponsor.chapter_name → $cn
assign $sponsor.sig_name → $sn
unnest $event.sponsoring_sigs → $sponsor
unnest dataset('Event') → $event
empty-tuple-source
1
1:1
A
sort $sn, $cn
2
1:1
B
group by $sn, $cn {
aggregator count, $cn → $c_chapter
}
3
M:N
partition on $sn
merge on $sn, $cn
C
group by $sn {
aggregator sum, $c_chapter → $total
nested-tuple-source
} {
aggregator listify, $by_chapter → $temp1
assign {"chapter_name": $cn,
"count": $c_chap_total } → $by_chapter
group by $cn {
aggregator sum, $c_chapter → $c_chap_total
nested-tuple-source
}
nested-tuple-source
}
4
1:1
D
sort $total desc
5
1:1
E
project $total, $r
assign {
"sig_name": $sn,
"total": $total,
"chapter_breakdown": $temp1
} → $r
limit 5
6
N:1
merge on $total
F
project $r
limit 5
7

Figure 12: Hyracks job specification for aggregation query

(e.g., Jaccard) and match thresholds, but it also includes syntax extensions to allow users to specify these in their queries.

There are several other modern data features presently under development for the ASTERIX system that are similarly aimed at supporting data drawn from (or alternatively pushed to ASTERIX from) around the Web, including the increasingly popular and important mobile side of the Web. These include support for location-based (i.e., geospatial) data as well as support for automatic data ingestion APIs (via a new ASTERIX feature called datafeeds) that channel data coming from continuous Web sources such as Twitter and RSS-based news services into affiliated ASTERIX datasets for either immediate or later searching and analysis.

4.5 The Stratosphere Cake

Donkey: Oh, you both have LAYERS. Oh. You know, not everybody like [sic] onions. What about cake? Everybody loves cake!
– *from the 2001 Dreamworks movie "Shrek"* [6]

Another data-intensive computing project in which the nature of the layers is being reexamined is the Stratosphere effort [14]. Running in parallel with ASTERIX, but in Europe (Germany), the Stratosphere project is investigating "Information Management on the Cloud". The Stratosphere system itself has two layers, a new programming model called PACT and a parallel execution engine called Nephele. Beneath Nephele lies the usual HDFS-based distributed data storage.

Nephele is roughly analogous to Hyracks, but at a slightly lower level (e.g., it is a bit more like Microsoft's Dryad [35] engine). The job of Nephele is to execute dataflow graphs in a massively parallel and flexible manner. Sitting above Nephele in the Stratosphere architecture is the PACT programming model. PACT is a generalization and extension of the MapReduce programming model that retains MapReduce's "your simple function goes here" nature, but PACT has a richer set of operators that include several binary operators. In particular, in addition to the familiar operators Map and Reduce, PACT's available operators include Cross, CoGroup, and Match, giving PACT sufficient natural expressive power to cover the relational algebra with aggregation. The less-constrained model aims to provide for easier definition of complex parallel data analysis tasks. A cost-based optimizer then compiles PACT programs down to Nephele dataflow graphs. Future plans for Stratosphere include a higher-level declarative language whose query syntax will be translated into PACT dataflow graphs and then optimized by the PACT compiler and executed by the Nephele engine.

We classify the Stratosphere effort here as cake, another dessert, because the project is thoughtfully reconsidering the MapReduce layer's content; it is not shying away from replacing Hadoop with a generalization based on a new model, and it is also developing an entirely new code base. We stop short of classifying the Stratosphere effort as a parfait because its architectural changes are more conservative (i.e., less global) than what we believe to be the full set of ingredients for making parfait.

4.6 ASTERIX Parfait Status

We are now 2.5 years into our initial 3-year, NSF-sponsored ASTERIX effort. The Hyracks layer is maturing nicely, being the oldest part of the system. Hyracks is able to significantly outperform Hadoop on typical data-intensive computing problems based on our experimental results, and we have recently tested it on clusters with up to 200 nodes (with 800 disks and 1600 cores). The Algebricks layer emerged in its own right as a result of our first-hand experience in supporting both AQL and HiveQL on Hyracks. Algebricks was created by factoring its functionality out of our AQL implementation in order to share it generally with Hive, which is based on a different data model and a different query language that is much more SQL-like. We have measured typical performance improvements of 3-4x in preliminary experiments comparing HiveQL on Algebricks and Hive itself on Hadoop, and several colleagues at Yahoo! are exploring Hyracks and Algebricks as a basis for their work on supporting data-parallel machine learning computations (and possibly parallel Datalog). Last but not least, the ADM/AQL layer, i.e., ASTERIX proper, is coming together as well. It is able to run parallel queries including lookups, large scans, parallel joins, and parallel aggregrates efficiently for data stored in our partitioned LSM B+ tree storage component and indexed via LSM B+ tree and LSM-based R-tree indexes. We are planning a first open-source release of the ASTERIX system and its component layers sometime during the latter part of 2012.

5. OPEN QUESTIONS

We certainly live in interesting times! Some of the more interesting open questions today, at least in our view, are the following:

1. For a quarter of a century, the database field operated with essentially one data model that "stuck" and a few standard architectures for such systems. In response to the more recent software and hardware trends discussed in the History section, as well as some other observations, Stonebraker and colleagues have been arguing over the last few years that those "one size fits all" days are simply over [47]. The upper layers of current "Big Data" platforms (both analytic and key-value systems) seem to agree, as there are nearly as many data models, languages, and APIs as there are systems at present. Which of these two situations was transient, which was normal, and what should the future hold in that regard?

2. On the analytical side of the early parallel database days, and even recently on the transactional side, there has been much debate about shared-nothing versus shared-disk architectures as being "right" to support the load balancing and failure-handling needs of scalable data management. It appears that there is a similar tradeoff to be considered, and a similar debate to be had, with respect to where data and its processing should reside in cloud-based architectures involving "Big Data". What will the answer be?

3. Related to this issue, while stateless computations are elastic, data tends to be anything but elastic. Can truly elastic scalability be realized in the "Big Data" world, and if so, how – and elastic on what time scale (minutes, hours, days, weeks)?

4. When we started the ASTERIX project, we conducted an informal survey of a set of colleagues at some of the large "Big Data" players (e.g., Teradata, Facebook, and eBay) to see where they thought systems research attention was needed. One consistent theme that we heard was that real "Big Data" clusters involve multi-user, multi-class (i.e., mixed job size) workloads – not just one analysis job at a time, or a few analysis jobs at a time, but a mix of jobs and queries of widely varying importance and sizes. Can the next generation of "Big Data" systems learn to manage such workloads effectively? This has been a long-standing (and difficult!) open problem in the parallel database world, and it has also largely been ignored or at least under-addressed in the world of Hadoop and friends.

5. Also related to the "one size" issue, on the programming model side, a variety of models are being proposed and used for different classes of "Big Data" problems – log analysis (MapReduce and its higher-level languages), machine learning tasks (Iterative MapReduce extensions), graph computations (e.g., Pregel [38] and other "bulk synchronous processing" frameworks), and so on. How will this unfold? How can we best meet the needs of data pipelines that involve multiple of these paradigms' requirements?

6. CONCLUSION

In this paper we have reviewed the history of systems for managing "Big Data" in the database world and (more recently) the systems world, as well as examining recent "Big Data" activities and architectures, all from the perspective of three "database guys" who are currently working to create a next-generation "Big Data" platform of our own. Our focus here has been on architectural issues, particularly on the components and layers that have been developed in open source and how they are being used to tackle the challenges posed by today's "Big Data" problems. We also presented the approach being taken in our ASTERIX project at UC Irvine, and hinted at our own answers to the questions regarding the "right" components and the "right" set of layers for taming the modern "Big Data" beast. Lastly, we shared our views on what we see as some of the key open questions and issues.

As a part of this paper's journey through the landscape of "Big Data" related architectures, we formulated a set of deeply technical analogies regarding the different approaches:

Parallel database systems are like onions. They have layers internally, but users can only see their whole form from the outside (SQL). Tear-evoking smells result if you try to cut them open.

The open-source Hadoop stack is like an ogre. It has layers, and it is quite powerful. However, it is arguably ugly, and its layers are not organized in a very elegant or even a very sensible manner, particularly when considering the stack from the perspective of an architecture for a high-level language-based "Big Data" platform.

The database community should be making parfaits. Our community has significant expertise in declarative data processing and how to do it efficiently as well as to make it scale. We should be applying our expertise to the design of more appropriate and more efficient future "Big Data" platforms. (We are trying to do exactly this in the ASTERIX project.)

To close, we would like to leave the EDBT community with two thoughts. Our first thought is that those of us on the database side of the emerging "Big Data" field need to not be shy – we need to question everything and rethink everything rather than assume that Hadoop, which arrived while we were napping, is now "the answer" that we need to adapt to. After all, if not us, then who? And if we don't teach our students to ask such questions, who will ask them when they need to be asked yet again? Our second thought is actually an invitation: The ASTERIX project is an open-source effort with an Apache-style license, and a release is planned in 2012. We are starting to look both for early users, i.e., collaborators who would like a flexible platform for managing and analyzing their application's "Big Data", as well as other potential ASTERIX contributors, i.e., developers who would like to try building new components or experimenting with their own alternative algorithmic ideas within some component. Who would like to play with us going forward?

Acknowledgements: The ASTERIX project is supported by NSF IIS awards 0910989, 0910859, 0910820, and 0844574, a grant from the UC Discovery program, and a matching donation from eBay. Vinayak Borkar was supported in part by a Facebook Fellowship award. We are also grateful to Yahoo! for their collaborative support and for their provision of access to one of their research clusters.

7. REFERENCES

[1] Apache Cassandra website. `http://cassandra.apache.org`.

[2] Apache Hadoop website. `http://hadoop.apache.org`.

[3] Apache HBase website. `http://hbase.apache.org`.

[4] Apache Hive website. `http://hive.apache.org`.

[5] jaql: Query language for JavaScript Object Notation (JSON). `http://code.google.com/p/jaql/`.

[6] Memorable quotes for Shrek (2001). *IMDB.com*.

[7] Jim Gray – industry leader. Transaction Processing Performance Council (TPC) web site, April 2009. http://www.tpc.org/information/who/gray.asp.

[8] The big data era: How to succeed. *Information Week*, August 9, 2010.

[9] Data, data everywhere. *The Economist*, February 25, 2010.

[10] Anon Et Al. A measure of transaction processing power. In *Technical Report 85.2*. Tandem Computers, February 1985.

[11] Apache Hadoop, http://hadoop.apache.org.

[12] Apache Hive, http://hadoop.apache.org/hive.

[13] C. Baru, G. Fecteau, A. Goyal, H.-I. Hsiao, A. Jhingran, S. Padmanabhan, W. Wilson, and A. G. H.-I. Hsiao. DB2 parallel edition. *IBM Systems Journal*, 34(2), 1995.

[14] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *SoCC*, pages 119–130, New York, NY, USA, 2010. ACM.

[15] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in MapReduce. In *Proc. of the 2010 International Conference on Management of Data*, SIGMOD '10, New York, NY, USA, 2010. ACM.

[16] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

[17] E. A. Brewer. Combining systems and databases: A search engine retrospective. In J. M. Hellerstein and M. Stonebraker, editors, *Readings in Database Systems, Fourth Edition*. MIT Press, 2005.

[18] M. Calabresi. The Supreme Court weighs the implications of big data. *Time*, November 16, 2011.

[19] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39:12–27, May 2011.

[20] R. G. G. Cattell, editor. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.

[21] R. Chaiken, B. Jenkins, P. A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[22] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[23] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04*, pages 137–150, December 2004.

[24] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53:72–77, Jan. 2010.

[25] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[26] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[27] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - a high performance dataflow database machine. In *VLDB*, pages 228–237, 1986.

[28] C. Freeland. In big data, potential for big division. *New York Times*, January 12, 2012.

[29] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the system software of a parallel relational database machine GRACE. In *VLDB*, pages 209–219, 1986.

[30] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of MapReduce: the Pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[31] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. 19th ACM Symp. on Operating Systems Principles*, SOSP '03, New York, NY, USA, 2003. ACM.

[32] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[33] R. B. Hagmann and D. Ferrari. Performance analysis of several back-end database architectures. *ACM Trans. Database Syst.*, 11, March 1986.

[34] B. Hopkins. Beyond the hype of big data. *CIO.com*, October 28, 2011.

[35] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[36] JSON. http://www.json.org/.

[37] W. Kim. *Special Issue on Database Machines*. IEEE Database Engineering Bulletin, December 1981.

[38] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[39] Object database management systems. http://www.odbms.org/odmg/.

[40] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[41] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33:351–385, June 1996.

[42] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Sotonebrakeotonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.

[43] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. WCB/McGraw-Hill, 2002.

[44] J. Shemer and P. Neches. The genesis of a database computer. *Computer*, 17(11):42 –56, Nov. 1984.

[45] M. Stonebraker. Operating system support for database management. *Commun. ACM*, 24, July 1981.

[46] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel

DBMSs: friends or foes? *Commun. ACM*, 53:64–71, Jan. 2010.

[47] M. Stonebraker and U. Cetintemel. One size fits all: An idea whose time has come and gone. *Data Engineering, International Conference on*, 0:2–11, 2005.

[48] The Tandem Database Group. Nonstop SQL: A distributed, high-performance, high-availability implementation of SQL. *Second International Workshop on High Performance Transaction Systems*, September 1987.

[49] R. Vernica. *Efficient Processing of Set-Similarity Joins on Large Clusters*. Ph.D. Thesis, Computer Science Department, University of California-Irvine, 2011.

[50] R. Vernica, M. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD Conference*, 2010.

[51] T. Walter. Teradata past, present, and future. In *UCI ISG Lecture Series on Scalable Data Management*, October 2009. http://isg.ics.uci.edu/scalable_dml_lectures2009-10.html.

[52] M. Weimer, T. Condie, and R. Ramakrishnan. Machine learning in scalops, a higher order cloud computing language. In *NIPS 2011 Workshop on parallel and large-scale machine learning (BigLearn)*, December 2011.

[53] D. Weinberger. The machine that would predict the future. *Scientific American*, November 15, 2011.

[54] XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/.

[55] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In R. Draves and R. van Renesse, editors, *OSDI*, pages 1–14. USENIX Association, 2008.