ASTERIX: An Open Source System for "Big Data" Management and Analysis (Demo)

Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Nicola Onose[*], Pouria Pirzadeh, Rares Vernica[†], Jian Wen[‡]
Information Systems Group, University of California, Irvine
mjcarey@ics.uci.edu[§]

ABSTRACT

At UC Irvine, we are building a next generation parallel database system, called ASTERIX, as our approach to addressing today's "Big Data" management challenges. ASTERIX aims to combine time-tested principles from parallel database systems with those of the Web-scale computing community, such as fault tolerance for long running jobs. In this demo, we present a whirlwind tour of ASTERIX, highlighting a few of its key features. We will demonstrate examples of our data definition language to model semi-structured data, and examples of interesting queries using our declarative query language. In particular, we will show the capabilities of ASTERIX for answering geo-spatial queries and fuzzy queries, as well as ASTERIX' data feed construct for continuously ingesting data.

1. INTRODUCTION

We started the ASTERIX project [1, 3] at UC Irvine approximately two and a half years ago. Our goal at the outset was to design and implement a highly scalable platform for information storage, search, and analytics. By combining and extending ideas from semistructured data management, parallel database systems, and first-generation data-intensive computing platforms (MapReduce and Hadoop), ASTERIX was envisioned to be a parallel, semistructured information management system with the ability to ingest, store, index, query, analyze, and publish very large quantities of semistructured data. ASTERIX is well-suited to handle use cases ranging all the way from rigid, relation-like data collections, whose types are well understood and invariant, to flexible and more complex data, where little is known a priori and the instances in data collections are highly variant and self-describing.

While ASTERIX began with the objective of creating a parallel, semistructured information management system, three reusable software layers have resulted from our work. The bottommost layer of the ASTERIX stack is a data-intensive runtime called Hyracks [4].

[*]now at Google
[†]now at HP Labs
[‡]at University of California, Riverside
[§]primary contact

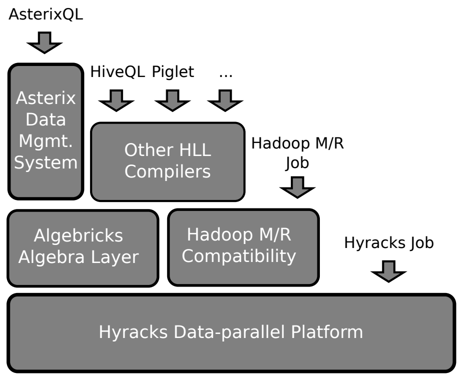


Figure 1: ASTERIX layers and entry points

Hyracks sits at roughly the same level of the architecture that MapReduce (Hadoop) does in implementations of higher-level data analysis languages such as Pig [8], Hive [2] or Jaql [5]. The topmost layer of the ASTERIX stack is a parallel DBMS, with a full, flexible data model (ADM) and a 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). In between these 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. Figure 1 summarizes the layers. Sections 2, 3 and 4 briefly explain how each layer does its job, and what its potential value is as a software resource for the "Big Data" analytics and management community. We will start at the bottom of the stack and work our way up to ASTERIX proper (i.e., to the complete end-user system).

2. THE HYRACKS RUNTIME

The Hyracks layer of ASTERIX is the bottom-most layer of the stack. Hyracks is the runtime layer (*a.k.a.* executor) whose job is to accept and manage data-parallel computations requested either by direct end users of Hyracks or (more likely) by 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, Figure 2 shows an example Hyracks job representing a TPCH-like query that performs a join between a partitioned file containing Customer records and another parti-

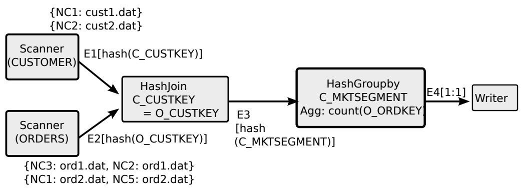


Figure 2: Example Hyracks job specification

tioned 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 2, 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.

Hadoop has quickly become the gold standard in the industry for 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 Hyracks' computational model, its implementation and perormance as well as the Hadoop compatibility layer are available in [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. To be useful for implementing arbitrary languages, Algebricks has been carefully designed to be agnostic of the model of its processed data. Logically, operators work 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 [9] 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.

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 compiler from Facebook, converting it to generate an Algebricks DAG that is then optimized and executed on Hyracks.

4. THE ASTERIX PARALLEL INFORMATION SYSTEM

At the top layer of the ASTERIX software stack sits the original goal, the ASTERIX parallel information management system itself (or simply ASTERIX, for short). Figure 3 provides an overview of how the various software components of ASTERIX map to nodes in a shared-nothing cluster and indicates how Hyracks serves as the runtime executor for query execution and storage management operations in ASTERIX.

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

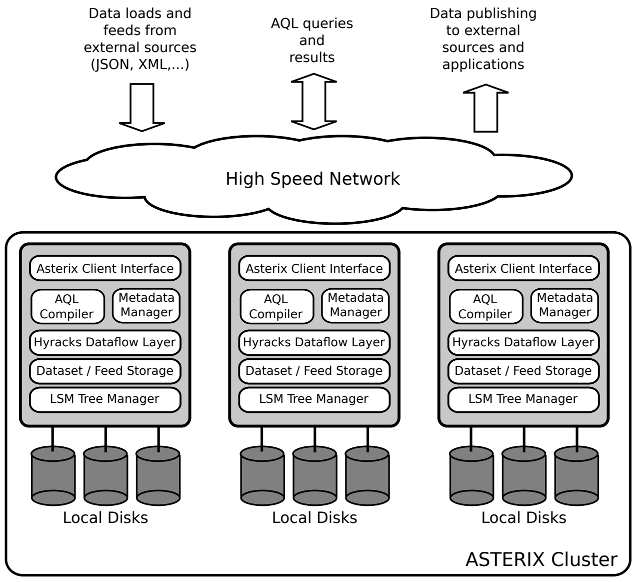


Figure 3: ASTERIX system schematic

```
create type TweetMessageType as open {
  tweet_id: string,
  user:{
     screen-name: string,
     lang: string,
     friends_count: int32,
     statuses_count: int32,
     name: string,
     folllowers_count: int32
  },
  sender-location:_point ?
  send-time: datetime,
  referred_topics: {{string}}
  message-text: string
};

create dataset TweetMessges(TweetMessageType)
    partitioned by key tweet_id;
```

(a) ADM type and dataset for tweets

```
for $tweet in dataset('TweetMessages')
where some $topic in $tweet.referred-topics
      satisfies contains ($topic,'verizon')
for $topic in $tweet.referred-topics
group by $topic with $tweet
return {
  "topic": $topic,
  "count": count($tweet)
}
```

(b) An example query over Tweet messages to filter and aggregate tweets

```
[
    { "topic": "verizon", "count": 3 },
    { "topic": "commercials", "count": 1 },
    { "topic": "att", "count": 1 },
    { "topic": "at&t", "count": 1 }
]
```

(c) Result of example query on example data.

```
{{ {
    "tweetid": "1023",
    "user": {
        "screen-name": "dflynn24",
        "lang": "en",
        "friends_count": 46,
        "statuses_count": 987,
        "name": "danielle flynn",
        "followers_count": 47
    },
    "sender-location": "40.904177,-72.958996",
    "send-time": "2010-02-21T11:56:02-05:00",
    "referred-topics": {{ "verizon" }},
    "message-text": "i need a #verizon phone :("
  },
  {
    "tweetid": "1024",
    "user": {
        "screen-name": "miriamorous",
        "lang": "en",
        "friends_count": 69,
        "statuses_count": 1068,
        "name": "Miriam Songco",
        "followers_count": 78
    },
    "send-time": "2010-02-21T11:11:43-08:00",
    "referred-topics": {{ "commercials", "verizon", "att" }},
    "message-text": "#verizon & #att #commercials,
        so competitive"
  },
  {
    "tweetid": "1025",
    "user": {
        "screen-name": "dj33",
        "lang": "en",
        "friends_count": 96,
        "statuses_count": 1696,
        "name": "Don Jango",
        "followers_count": 22
    },
    "send-time": "2010-02-21T12:38:44-05:00",
    "referred-topics": {{ "verizon", "at&t", "iphone" }},
    "message-text": "I think I may switch from
        #verizon to #at&t"
  } }}
```

(d) Example tweet data

```
for $tweet1 in dataset('TweetMessages')
for $tweet2 in dataset('TweetMessages')
where $tweet1.tweetid != $tweet2.tweetid
    and $tweet1.message-text~=$tweet2.message-text
return {
    "tweet1-text": $tweet1.message-text,
    "tweet2-text": $tweet2.message-text
}
```

(e) An example fuzzy query over Tweet messages to find similar tweets

Figure 4: Example AQL schemas, queries, and results

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 [6] and adding additional primitive types as well as type constructors borrowed from object databases [7]. Figure 4(a) illustrates ADM by showing how it could be used to define a record type for modeling Twitter messages. The record type shown there is an open type, meaning that the instances of this type will conform to its specification but are allowed to contain arbitrary additional fields that vary from one instance to the next. The example also illustrates how ADM includes features such as optional fields with known types (sender-location), nested collections of primitive values (referred-topics), and nested records (user). More information about ADM can be found in [3].

Figure 4(d) shows an example of a set of `TweetMessageType` instances. 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 (such as the `TweetMessages` dataset declared at the bottom of Figure 4(a)), which are stored and managed by ASTERIX as partitioned LSM-based B+ trees with optional 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.

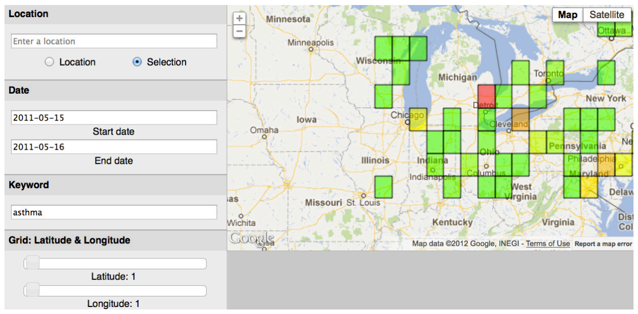
ASTERIX queries are written in AQL (the ASTERIX Query Language), a declarative language that is designed by taking the essence of XQuery [11], most importantly its FLWOR expression constructs and its composability, and simplifying and adapting it to query the types and data modeling constructs of ADM. Figure 4(b) illustrates AQL by example. This query runs over the `TweetMessages` dataset, which contains `TweetMessageType` instances, to compute, for those tweets that refer to "verizon", the number of tweets that refer to each topic that appears in those tweets.

Figure 4(c) shows what this query's results would look like when run against the sample data of Figure 4(d). To process queries like this one, 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 that uses the operators and connectors of Hyracks to compute the desired query result.

5. DEMONSTRATION DETAILS

We will demonstrate the following features of ASTERIX.

1. Asterix Data Model (ADM): We will demonstrate how ADM 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.
2. Asterix Query Language (AQL): We will demonstrate the capabilities and expressive power of AQL by running different kinds of queries against a running instance of the ASTERIX system. For each of the queries, we plan to give a walk-through tour of the system, describing how a logical plan for a query if formed, translated and optimized into an efficient parallel physical plan by the underlying language-neutral Algebrics layer and eventually executed as a DAG by the Asterix runtime - Hyracks.
3. Geo-spatial queries: Spatial data types (point, line, polygon, etc.) are treated as first-class citizens in ASTERIX, providing built-in support for geo-spatial (i.e., location-based) data. We will demonstrate, via a very simple Web-based user interface example (Figure 5), how ASTERIX might be used for spatial analysis of social data (from Twitter) to derive knowledge useful to society.
4. Fuzzy queries: An important use case for ASTERIX is querying, and analysis 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 plan to demonstrate. An example fuzzy matching query is shown in Figure 4(e). This example query is executed in parallel based on principles that we developed while studying how to perform fuzzy joins in the context of Hadoop [10].
5. Data Feeds: ASTERIX supports continuous data ingestion via data feeds and can can be used to amass data from services such as Twitter or RSS feed-based services such as CNN news. We will demonstrate how ASTERIX can serve as an effective platform for archiving, tracking and analyzing social media activity.


Location

Enter a location

[ ] Location [✓ Selection]

Date

2011-05-15

Start date

2011-05-16

End date

Keyword

asthma

Grid: Latitude & Longitude

Latitude: 1

Longitude: 1

Figure 5: A visualization of the results of a spatial aggregation query. The color of each cell indicates the tweet count.

6. STATUS OF ASTERIX

Currently, the ADM/AQL layer of ASTERIX is able to run parallel queries including lookups, large scans, parallel joins (regular and fuzzy), and parallel aggregates for data stored in partitioned LSM B+ trees and indexed via secondary indexes such as LSM-based R-trees. The system's external data access and data feed features are also operational. We plan to offer a first open-source release of ASTERIX during the latter part of 2012, and we are now seeking early partners who would like to try ASTERIX on their favorite "Big Data" problems. Our ongoing work includes hardening and documenting the ASTERIX code base for initial public release, adding indexing support for fuzzy selection queries, improving the performance of spatial aggregation, adding support for continuous queries, extending AQL with windowing features, and starting to work with a few early users and use cases to learn by experience where we should go next.

Acknowledgements: This 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.

7. REFERENCES

[1] ASTERIX Website. http://asterix.ics.uci.edu/.
[2] Apache Hive, http://hadoop.apache.org/hive.
[3] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. Asterix: Towards a Scalable, Semistructured Data Platform for Evolving-World Models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.
[4] 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.
[5] Jaql, http://www.jaql.org.
[6] JSON. http://www.json.org/.
[7] Object database management systems. http://www.odbms.org/odmg/.
[8] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a Not-so-Foreign Language for Data Processing. In *SIGMOD*, pages 1099–1110, 2008.
[9] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. WCB/McGraw-Hill, 2002.
[10] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD*, pages 495–506, 2010.
[11] XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/.