ASTERIX: Scalable Warehouse-Style Web Data Integration

Sattam Alsubaiee, Alexander Behm, Raman Grover, Rares Vernica
Vinayak Borkar, Michael J. Carey, Chen Li
University of California, Irvine
{salsubai,abehm,ramang,vborkar,mjcarey,chenli}@ics.uci.edu, rares.vernica@hp.com

ABSTRACT

A growing wealth of digital information is being generated on a daily basis in social networks, blogs, online communities, etc. Organizations and researchers in a wide variety of domains recognize that there is tremendous value and insight to be gained by warehousing this emerging data and making it available for querying, analysis, and other purposes. This new breed of "Big Data" applications poses challenging requirements against data management platforms in terms of scalability, flexibility, manageability, and analysis capabilities. At UC Irvine, we are building a next-generation database system, called ASTERIX, in response to these trends. We present ongoing work that approaches the following questions: How does data get into the system? What primitives should we provide to better cope with dirty/noisy data? How can we support efficient data analysis on spatial data? Using real examples, we show the capabilities of ASTERIX for ingesting data via feeds, supporting set-similarity predicates for fuzzy matching, and answering spatial aggregation queries.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems; H.2.7 [**DATABASE MANAGEMENT**]: Database Administration—*Data warehouse and repository*

General Terms

Design, Management, Performance

Keywords

Data-intensive computing, Cloud computing, Semistructured data, ASTERIX, Hyracks

1. INTRODUCTION

We started the ASTERIX project [1, 5] 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.

Traditionally, there have been two major approaches to integrating data from disparate sources [7]: Data warehousing, and virtual data integration (a.k.a. federation), both of which have been extensively studied and implemented. ASTERIX follows a "web warehousing" philosophy where social and web data are ingested into and analyzed in a single scalable platform. In this paper, we introduce specific features of ASTERIX that emerged from this goal.

Figure 1 provides an overview of how the various software components of ASTERIX map to nodes in a shared-nothing cluster. The bottom-most layer provides storage facilities for managed datasets based on LSM-trees, which can be targets of ingestion. Further up the stack lies our data-parallel runtime called Hyracks [8]. It sits at roughly the same level that Hadoop does in implementations of other high-level languages such as Pig [17] or Hive [3] or Jaql [11]. The topmost layer of ASTERIX is a parallel DBMS, with a full, 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 (e.g., in HDFS). As part of the AQL compiler, we have developed Algebricks, a model-agnostic, algebraic "virtual machine" for optimizing parallel queries. Algebricks is the target for AQL compilation, but it can also be the target for other declarative data languages.

1.1 EXAMPLE SCENARIO

Consider the upcoming US presidential elections. The elections are bound to generate a significant amount of online activity (tweets, blogs, etc) and are expected to be extensively covered by news media. Each tweet or news article related to the event contributes to a large information repository that can provide useful insights if subjected to analysis. We will use a simple example based on the election context to show the capabilities of ASTERIX.

The ASTERIX data model (ADM) is based on ideas from JSON with additional primitive types as well as type constructors borrowed from object databases [16]. Figure 2 shows how tweets and CNN news articles can be represented as records using ADM. Notice that the record types shown in the figure are open types, signifying that the instances of this type will conform to its specification but are allowed to contain arbitrary additional fields that vary

IIWeb '12 May 20 2012, Scottsdale, AZ, USA

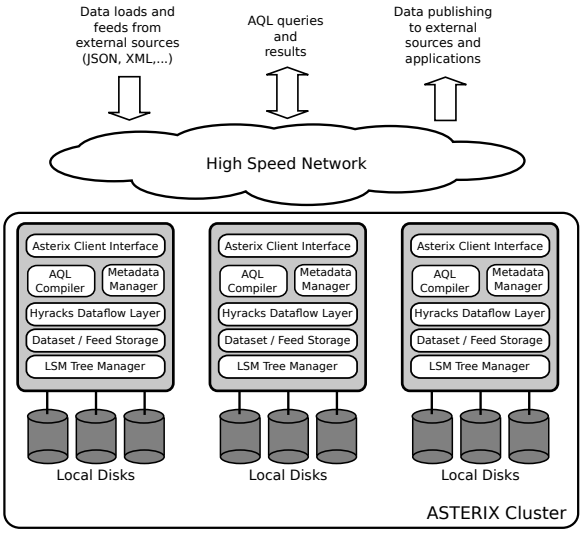

Figure 1: ASTERIX system architecture.

from one instance to the next. The example also illustrates how ADM includes features such as optional fields with known types (`location` in `TweetType`), and nested collections of primitive values (`hashTags` in `TweetType`).

```
create type TweetType
as open {
  id: string,
  username: string,
  location: point?,
  text: string,
  timestamp: string,
  hashTags: {{string}}?
};
```

```
create type NewsType
as open {
  id: string,
  title: string,
  description: string,
  link: string,
  topics: {{string}}?
};
```

Figure 2: Metadata definition for the running example.

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, 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. More information about ADM can be found in [5].

2. DATA INGESTION/FEEDS

To amass data from services such as Twitter or CNN news, and serve as an effective platform for tracking and analyzing social media activity, ASTERIX supports continuous data ingestion via *data feeds*. Data can be collected from a wide variety of sources using wrappers (*adapters* in ASTERIX) that abstract away the mechanism of connecting with an external service, receiving data in either push or pull mode, and transforming the data into ADM records understood by ASTERIX. Continuously arriving data is persisted in ASTERIX as a feed dataset. Feed datasets differ from regular datasets only in where their data comes from; they are essentially append-only datasets bundled with connections to a data provider.

Figure 3 shows AQL DDL statements to create a Twitter feed and a CNN news feed. In the context of our running example, the Twitter feed `Tweets` contains ADM records that conform to the `TweetType` described in Figure 2. Similarly the CNN news feed `News` contains instances of `NewsType`. Each feed has an associated adapter and may be given adapter-specific configuration parameters. The `TwitterAdapter` in the example is configured with a time interval (seconds) between successive requests to the remote Twitter service for fetching tweets.

To facilitate analysis on the received data, a user may specify a pre-processing function which is applied to each feed record before persisting it to a dataset. For example, the `Twitter` feed utilizes a user-defined function `addHashTagsToTweet` that extracts all hash tags[1] in a tweet. The extracted hash tags are appended to the feed record and used subsequently for further analysis (Sections 3 and 4). Similar pre-processing to extract topics is applied to each news feed record. Like any ASTERIX dataset, a feed definition also requires a partitioning clause that informs ASTERIX about how to partition the receiving data. In our running example, we simply hash partition both feed datasets by their `id`.

```
create feed dataset Tweets(TweetType)
using TwitterAdapter ("interval"="10")
apply function addHashTagsToTweet
partitioned by key id;

create feed dataset News(NewsType)
using CNNFeedAdapter("topic"="politics","interval"="600")
apply function getTaggedNews
partitioned by key id;

create index locationIndex on Tweets(location) type rtree;

begin feed Tweets;
begin feed News;
```

Figure 3: Feed definitions for the running example.

Once a data feed has been defined, data ingestion can be triggered with a `begin feed` statement[2] (as shown in Figure 3). Feed datasets are first-class citizens; a user may write AQL queries against them, create secondary indexes, etc. In our example, we create an R-tree index on the `location` attribute of `TweetType`.

Runtime Execution: A feed ingestion workflow is a DAG executed as an ever-running parallel Hyracks job. The adapter associated with the feed may run on multiple nodes. Each adapter instance connects to the external feed source and receives data. The output ADM records are spread across the cluster nodes based on the partitioning criterion specified in the feed definition. Any pre-processing (user-defined function) is applied at the recipient nodes before the ADM instances are inserted in the local LSM-B+tree(s) for that dataset. Secondary indexes are also updated if necessary.

Related Work: Data feeds may seem similar to streams from the data stream management systems literature [2, 4] or to complex event processing systems [13, 24]. There are several important differences, however. Data feeds in ASTERIX are a "plumbing" concept; they are simply the mechanism for having data arrive into the ASTERIX system from external sources that produce data continuously, and to have that data incrementally populate a persisted dataset. To our knowledge, this will be the first system to explore the challenges involved in building a feed ingestion facility that deals with semi-structured data and employs partitioned parallelism in order to scale the facility and couple it with high-volume and/or parallel external data sources. The most closely related work is the AT&T Bistro data feed management system [21], but the focus of that work is on routing large amounts of file-based data from pre-determined feeds to the applications that need access to them.

3. FUZZY MATCHING

[1]Words beginning with #; hash tags in a tweet are symbolic of topics associated with the tweet

[2]ASTERIX also provides `suspend feed`, `resume feed` and `end feed` statements for controlling the lifecycle of a feed.

Having as a target use case the archiving, querying, and analysis of semistructured data drawn from Web sources, it is evident that data quality is an issue. Social network users often publish data informally, and can post tweets from mobile devices, resulting in many abbreviated keywords and typos. Fuzzy matching capabilities need to be added to ASTERIX, and we are in the process of adding fuzzy selection and join queries, described as follows.

Fuzzy Selection Queries: Many people mistype (perhaps purposely) the name of a currently prominent politician "Rick Santorum" as "Rick Sanitarium". Therefore, when querying the system for tweets about Rick Santorum, we should include similar matches as well, e.g., using edit distance. One may even infer the sentiment of the tweeter based on the misspelling. Currently, we are in the process of adding support for fuzzy selection queries.

Fuzzy Join Queries: In addition, analyzing such data, e.g., 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 example, suppose a news provider like CNN wants to optimize its web page layout by analyzing the success of past news stories. Apart from article-specific measures, such an analysis could take into account the general impact of news stories on certain combinations of topics on the Twitter community. The AQL query in Figure 4 illustrates a possible query formulation. Intuitively, the query returns the top ten most popular articles based on their relevance to topics mentioned in tweets. The first part of the query (up to `group by`) generates all pairs of tweets and news articles that have similar topics based on the Jaccard similarity of their topic-lists. The $\sim=$ operator means "similar to", and has been qualified with `set simfunction` and `set simthreshold`. The second part of the query counts the number of related tweets per news article, and returns the top ten articles with the highest tweet count.

```
set simfunction "jaccard"
set simthreshold "0.5f"

for $tweet in dataset('Tweets')
for $article in dataset('News')
where $tweet.hashTags ~= $article.topics
group by $a := $article.id with $article
order by count($article)
limit 10
return {"article": $article, "popularity": count($article)}
```

Figure 4: A set-similarity join to find the top ten most popular news articles based on their relevance to topics in tweets.

Runtime Execution: Executing queries with set-similarity (or string-similarity) predicates is challenging on large amounts of data. First, the predicates themselves are expensive, rendering brute-force solutions impractical. For example, computing the Jaccard similarity requires the union and intersection of two sets, and the classic edit distance algorithm uses dynamic programming, all of which are computationally expensive. Second, standard divide-and-conquer strategies based on hash partitioning are not directly applicable, because similar items may not have the same hash value. Currently, ASTERIX can execute fuzzy joins efficiently based on principles that we developed while studying how to perform fuzzy joins in the context of Hadoop [23, 22] (without pre-existing indexes). We have recently started implementing indexed support for such fuzzy queries based on our earlier work in [6]. We expect to speed up both selection and join queries with secondary indexes.

Related Work: There have been many studies on set-similarity and string-similarity selection [15, 10], and join queries [20, 25], some in the context of relational database systems [9]. Our work in ASTERIX complements previous work in that we are extending the core (single machine) techniques to implement end-to-end support for set-similarity queries in a parallel database system, including parallel joins and optimizations via distributed secondary indexes.

4. SPATIAL AGGREGATION

Large volumes of events and social data can be aggregated and analyzed to derive knowledge valuable to businesses, governments, and society. For instance, consider the case where the campaign manager of a presidential candidate such as Mitt Romney wants to know how potential voters are reacting to the Republican presidential primaries in a certain geographic area.

A useful piece of information is the level of voters' interest in the rival Rick Santorum in February/March, 2012 (close to Super Tuesday of the primary election) in different geographical regions. Such information is clearly valuable to the decision-making process of the campaign. The increasing availability and popularity of social data and event data make such information more readily available and more real time. We can derive such knowledge by doing a *spatial aggregation* on Twitter data as follows: We formulate a query to find the tweets mentioning "Santorum" posted from February 15, 2012 to March 1, 2012, group them on a grid structure, and compute the number of such tweets in each cell in the grid. By doing this spatial analysis, the campaign staff could gain an understanding of the public opinion, and make informed decisions such as broadcasting more political ads in certain areas.

Given the importance of spatial aggregation queries, we have added capabilities for them in ASTERIX. Such a query specifies a grid structure including a spatial range and a resolution and asks for a density distribution (histogram) of data within the grid. The query may optionally include a time interval and keywords and do an aggregation on the data records satisfying these additional conditions. A spatial aggregation query partitions data records into groups and applies an aggregation function to all records in each group. Figure 5 shows a color-coded density grid on the map that visualizes the results of a spatial aggregation query using the Google Maps API.

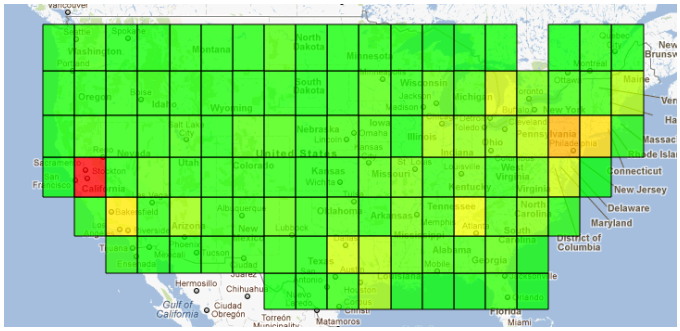

Figure 5: A visualization of the results of a spatial aggregation query. The color of each cell indicates the tweet count.

Consider the AQL query in Figure 6 which spatially aggregates election-related tweets. It starts by constraining tweets to a bounding rectangle inside the US, a datetime window, and those containing the hashtag "Santorum". The `spatial-cell` function determines which grid cell a tweet belongs to. This function receives the location of the tweet, the origin of the bounding rectangle, and the latitude and longitude increments (to specify the resolution of the grid). It returns the cell (represented by a rectangle) that the tweet belongs to. Those tweets are then grouped according to their containing grid cells. Finally the `count` function is applied to each

group of tweets to return the final answer as pairs of cell and number of tweets (that satisfy the predicates) in that cell.

```
for $tweet in dataset('Tweets')
let $searchHashTag := "Santorum"
let $leftBottom := create-point(33.13,-124.27)
let $rightTop := create-point(48.57,-66.18)
let $latResolution := 3.0
let $longResolution := 3.0
let $region := create-rectangle($leftBottom,$rightTop)
where spatial-intersect($tweet.location, $region) and
$tweet.time > datetime("2012-02-15T00:00:00Z") and
$tweet.time < datetime("2012-03-01T23:59:59Z") and
some $hashTag in $tweet.hashTags
  satisfies ($hashTag = $searchHashTag)
group by $c := spatial-cell($tweet.location,
  $leftBottom, $latResolution, $longResolution)
  with $tweet
return { "cell": $c, "count": count($tweet) }
```

Figure 6: Spatial aggregation query over tweets that were generated by US users close to Super Tuesday of the Republican primary election, containing the hashtag "Santorum".

Runtime Execution: Since ASTERIX provides rich spatial support, spatial aggregation queries are executed efficiently by using a secondary R-tree index. Thus, all records outside of the query bounding region are filtered quickly. One path that we are investigating to further boost performance is to incrementally pre-aggregate the data into a spatial index (akin to a materialized view).

Related Work: There are existing studies on answering spatial aggregation queries [18, 12], and spatio-temporal aggregation [19], where their goal is to do an aggregation based on space and time conditions simultaneously. Mathioudakis et al. [14] proposed a framework to identify spatial burstiness assuming the space is decomposed using a grid-based layout. Spatial Aggregation in ASTERIX is different from these earlier studies since we are interested in finding the density distribution of spatial objects, possibly with textual and temporal predicates. Moreover, in the ASTERIX system, spatial aggregation queries are part of a general spatial framework, where the goal is to support different types of useful spatial queries rather than supporting specific queries in an ad-hoc way.

5. FUTURE/ONGOING WORK

In this paper, we presented three key features of the ASTERIX system to warehouse and analyze social and Web data: Data feeds, fuzzy matching, and spatial aggregation. We have described our data feed mechanism for continuously ingesting data and showed how ASTERIX can help with inferring useful information via fuzzy matching and spatial aggregation. 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.

6. REFERENCES

[1] ASTERIX Website. http://asterix.ics.uci.edu/.

[2] D. J. Abadi et al. The design of the Borealis stream processing engine. In *In CIDR*, 2005.

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

[4] A. Arasu et al. Stream: The stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[5] 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*, 2011.

[6] A. Behm, C. Li, and M. J. Carey. Answering approximate string queries on large data sets using external memory. In *ICDE*, 2011.

[7] P. A. Bernstein and L. M. Haas. Information integration in the enterprise. *Commun. ACM*, 51(9):72–79, Sept. 2008.

[8] V. R. Borkar et al. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.

[9] L. Gravano et al. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[10] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, 2008.

[11] Jaql, http://www.jaql.org.

[12] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *SIGMOD*, 2001.

[13] M. Li, M. Mani, E. A. Rundensteiner, and T. Lin. Complex event pattern detection over streams with interval-based temporal semantics. In *Proc. of the 5th ACM Int. Conf. on Distrib. Event-Based Systems*, pages 291–302, 2011.

[14] M. Mathioudakis, N. Bansal, and N. Koudas. Identifying, attributing and describing spatial bursts. *PVLDB*, 3(1), 2010.

[15] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

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

[17] C. Olston et al. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[18] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In *SSTD*, 2001.

[19] D. Papadias, Y. Tao, P. Kalnis, and J. Zhang. Indexing spatio-temporal data warehouses. In *ICDE*, 2002.

[20] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, 2004.

[21] V. Shkapenyuk, T. Johnson, and D. Srivastava. Bistro data feed management system. SIGMOD, 2011.

[22] R. Vernica. *Efficient Processing of Set-Similarity Joins on Large Clusters*. Ph.D. thesis, UC Irvine, 2011.

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

[24] D. Wang, E. A. Rundensteiner, R. T. Ellison, and H. Wang. Active complex event processing infrastructure: Monitoring and reacting to event streams. In *ICDE Workshops*, 2011.

[25] C. Xiao, W. Wang, and X. Lin. Ed-join: An efficient algorithm for similarity joins with edit distance constraints. In *VLDB*, 2008.