Distrib Parallel Databases (2011) 29: 185–216
DOI 10.1007/s10619-011-7082-y

ASTERIX: towards a scalable, semistructured data platform for evolving-world models

Alexander Behm · Vinayak R. Borkar · Michael J. Carey · Raman Grover · Chen Li · Nicola Onose · Rares Vernica · Alin Deutsch · Yannis Papakonstantinou · Vassilis J. Tsotras

Published online: 31 March 2011

Abstract ASTERIX is a new data-intensive storage and computing platform project spanning UC Irvine, UC Riverside, and UC San Diego. In this paper we provide an overview of the ASTERIX project, starting with its main goal—the storage and anal-

Communicated by: Brian Frank Cooper.

A. Behm · V.R. Borkar · M.J. Carey (✉) · R. Grover · C. Li · N. Onose · R. Vernica
University of California, Irvine, USA
e-mail: mjcarey@ics.uci.edu

A. Behm
e-mail: abehm@ics.uci.edu

V.R. Borkar
e-mail: vborkar@ics.uci.edu

R. Grover
e-mail: ramang@ics.uci.edu

C. Li
e-mail: chenli@ics.uci.edu

N. Onose
e-mail: onose@ics.uci.edu

R. Vernica
e-mail: rares@ics.uci.edu

A. Deutsch · Y. Papakonstantinou
University of California, San Diego, USA

A. Deutsch
e-mail: deutsch@cs.ucsd.edu

Y. Papakonstantinou
e-mail: yannis@cs.ucsd.edu

V.J. Tsotras
University of California, Riverside, USA
e-mail: tsotras@cs.ucr.edu

ysis of data pertaining to *evolving-world models*. We describe the requirements and associated challenges, and explain how the project is addressing them. We provide a technical overview of ASTERIX, covering its architecture, its user model for data and queries, and its approach to scalable query processing and data management. ASTERIX utilizes a new scalable runtime computational platform called Hyracks that is also discussed at an overview level; we have recently made Hyracks available in open source for use by other interested parties. We also relate our work on ASTERIX to the current state of the art and describe the research challenges that we are currently tackling as well as those that lie ahead.

Keywords Data-intensive computing · Cloud computing · Semistructured data · ASTERIX · Hyracks

1 Introduction

We live in an information-driven and information-rich age. Data and databases have been central to the functioning of large enterprises for several decades. Over the past ten years, however, digital information, transactions, and connectedness have become a key part of almost every facet of our everyday lives. E-commerce now accounts for well over $100 billion annually in retail sales in the U.S. [56]. E-mail dominates written correspondence today; in 2009, the average daily number of e-mails sent and received for business users and for consumers was approximately 110 and 45, respectively [53]. Print media are suffering while online news portals and blogs are thriving [42]. Over the past five years, social networks such as MySpace and Facebook have exploded in popularity. At the end of 2009, Facebook reported [26] over 350 million active users and over 55 million daily status updates; over 3.5 billion pieces of content (Web links, news posts, blog posts, etc.) were being added weekly together with over 3.5 million new events on a monthly basis. Nine months later, Facebook's active user count was over 500 millions, with more than half using the site on any given day, and the arrival rate for new content was over 30 billion pieces per month. Last but not least, Twitter has taken the world by storm over the past 1–2 years. It is estimated that one in five Internet users use Twitter or similar services to share or see status updates about themselves or others [47], and in early 2010 Twitter reported processing over 50 million tweets per day [55]. As evidenced by recent world news, Facebook and Twitter have transformed the way that people share and acquire information about unfolding events, especially in countries with closed governments.

Given these remarkable trends, what are the implications for databases and data analysis? There is a growing wealth of digital information being generated and flowing on a daily basis, information that—if captured and aggregated effectively—has great potential value for many purposes. Data warehouses have largely been an enterprise phenomenon, with large enterprises being unique in recording their day-to-day operations in databases and then warehousing and analyzing historical data to improve their business operations. We believe that there is tremendous value and insight to be gained by warehousing the emerging wealth of digital information and making it available for querying, analysis, and other purposes. In this paper we introduce the

ASTERIX project, a new, multi-campus effort to design, develop, and deliver in open source a highly scalable platform for information storage and data-intensive information analysis. Essentially, the ASTERIX platform aims to be a parallel semistructured data management system that is able to ingest, store, index, query, analyze, and publish massive quantities of semistructured information. We envision the primary uses of ASTERIX being for data management "within the cloud," as a highly scalable system for managing and analyzing the kinds of data being produced due to the aforementioned information trends.

The remainder of this paper is organized as follows. Section 2 delves deeper into the motivations for ASTERIX, including potential classes of applications, target workload and computing platform characteristics, and key challenges. Section 3 briefly surveys the current state of the art in three relevant technology areas. Section 4 provides a broad overview of ASTERIX, explaining its basic architectural approach. Section 5 explores the ASTERIX user model more deeply, including its approaches to data modeling, management, and querying. Section 6 looks under the hood, explaining our initial design and the components that are now under construction. Section 7 outlines some of the key research challenges that we face as we strive to fully realize the ASTERIX vision. Finally, Sect. 8 concludes the paper and summarizes the project's current status.

2 Motivation

In this section we further explore what it means, and some of what it will take, to warehouse and provide services around the new wealth of digital information.

2.1 Evolving-world models

Data modeling of a changing world: A database is a collection of data, typically describing the activities of an enterprise or an organization [50]. In essence, a given database usually represents an attempt to capture and store a digital model of a small portion of the world, usually the portion pertaining to the day-to-day operations of an enterprise—in other words, a typical database is an *enterprise model*. By analogy, creating a large base of information derived from diverse world data sources implies attempting to capture and persist a digital model of a portion of the "real world." Going even further, keeping the history of such information implies capturing the evolution of such a world model. The availability of the sorts of information sources described in the introduction thus leads to the possibility of capturing and exploiting *evolving-world models*, as illustrated by the following example.

Consider the case where we wish to create a repository of information about events and other aspects of the present and past states of a metropolitan area, e.g., the greater Los Angeles (LA) area. "Traditional" information about the LA area could include map data (e.g., at the level of Google maps), business listings, event listings, traffic data, population data, and so on. Additional information could include local online news stories, blogs, tweets whose geospatial coordinates or hashtags lie in or relate to the LA area, updates from MySpace and Facebook users who publicly list themselves as being in the area, online photos that are spatially tagged in the area, similarly for online videos, and so on.

A digital warehouse containing such information about the greater LA area would essentially be an evolving-world model of the LA area in the sense that it would change over time, with new kinds of data and data sets appearing, existing data slowly changing, and so on. Queries over current and past data would be useful for finding all sorts of facts of interest. There can be queries asking for the current status of the world. For example, when heading to see USC and UCLA face off in football at the Coliseum, an application might exploit a combination of information about maps, events, traffic, and tweets to guide its users on reasonable paths to the game and help them find the remaining affordable parking spaces. There can be "as of" queries asking for the status of the world for a specific time in the past, e.g., the traffic situation close to the LAX airport on a Saturday. There can even be queries on the history of the changing world, such as the total number of visitors who were looking for nearby restaurants within one year. Such historical queries and analyses would be useful in predicting future conditions and behaviors, in making important business decisions, and in assessing and improving how past circumstances were responded to and what ensued as a result of those responses.

One can easily imagine similar models being useful for handling major events (e.g., a Presidential debate in the LA area) or for natural disasters (e.g., an earthquake). Internet-based access to such models could help the general public to be aware of the current situation in the area as well as to themselves contribute to the shared knowledge of the current state of the area.

Data characteristics and needs: As illustrated by the preceding example, such evolving-world applications share the following characteristics in their data and information needs. (1) Various types of data: The information tends to have diverse and rich structures. Thus, successful information management will demand handling a mix of structured, semistructured, and even unstructured information [4]. (2) Frequent updates: both the data and its structure can keep changing constantly, which poses significant research challenges. Even for traditional data management, change management has been a persistent challenge with partial and elusive solutions. Schema changes in traditional systems involve upgrading a database's schema and contents from one version to another, and somehow simultaneously updating the various dependent application programs, interfaces, and reports as well. The popular phrase "the only constant is change" is an apt description of the challenges posed in warehousing the new wealth of digital information. (3) Large amounts of data: The volume of data is large and grows fast, and storing and querying the data goes well beyond the computing capabilities of a single machine. All these characteristics require new data models and approaches for managing such evolving-world information.

Dialing back the setting on our time machine, it should be clear that such models are a natural progression from today's modern enterprise warehouses. In the past, data and analyses have been fairly predictable and static, and relational databases and queries have sufficed. Today's online enterprises, such as Google, Facebook, eBay, and others, are capturing large volumes of user data including information about their online actions, searches, purchases, and other activities. That data forms a looser, richer model that is being stored and processed via non-relational means, e.g., as large distributed files with formats known to their associated analysis programs and using analysis paradigms such as Map/Reduce programs [7, 17, 20, 21]. An evolving-world model of the sort we envision would be similar in nature—its data would continuously arrive and add to the information base, requiring similar (or even greater) scale to that needed by large online enterprises, and analyses of the data would likely have similar computational requirements.

2.2 ASTERIX goals

A high-level objective of the ASTERIX platform is to provide support for the storage and analysis of data with varying structure pertaining to such evolving-world models.

Data, queries, and workloads: (1) ASTERIX must support data spanning the spectrum from the structured to the semi/unstructured, and for the more structured data, evolution must be expected and gracefully handled. (2) Data will continue to arrive, so data sets can become very large. (3) Updates will mostly involve insertion of new data or creating a new version of existing data (with older versions being retained); deletion will then cause versioned data to cease being current. (4) ASTERIX must support both stored (managed) data and external data so that it is not necessary to "load" all data prior to using it. (5) Target workloads will be mixed, including short and medium queries asking for current state-of-the-world information, long analyses looking for historical trends, together with updates of the sort just described. (6) In addition, we envision a demand to support standing queries that are watching for future conditions of interest. Throughout all of this, we have to develop scalable, parallel solutions due to large data volumes.

Target computing platform: For obvious reasons, the target computing platform for ASTERIX is much like that for today's online enterprises, namely, very large shared-nothing clusters comprised of commodity processors, memories, disks, and networking hardware. In order to support mixed workloads involving potentially large numbers of simultaneous requests, we assume that the dynamic state of the system may vary greatly over time and that the current state of its resources will be an important factor in ASTERIX processing decisions. As a result, the development of effective, resource-aware task-scheduling techniques is a priority. The ASTERIX software architecture and techniques should also be applicable to emerging "elastic computing" infrastructures.

Challenges of scale: Dimensions of scale of interest to ASTERIX include very large data sets, workloads involving many concurrent requests, and individually large data-analysis tasks. Ensuing challenges on the ASTERIX research roadmap include support for large, self-managing data sets and index structures as well as query planning, processing, and scheduling approaches that scale and that are able to adapt to highly dynamic resource environments. Given the target scale, effective approaches to system management, monitoring, and logging will also be important.

3 Related work

To address its requirements, ASTERIX is drawing on and extending work in three areas: semistructured data, parallel database systems, and data-intensive computing.

3.1 Semistructured data

The database and XML communities have done significant research in the area of semistructured data management dating back to the Lore project [49] and other efforts in the mid- to late-1990s [1]. DataGuides [30] were proposed as one potential approach to understanding the contents of a (large) semistructured data set in the absence of static type information. In terms of the centralized management of semistructured data, much was accomplished, particularly related to query models and languages for self-describing data. As an outcome of that work and of the adoption of XML for data exchange in the industry, the XQuery language was developed and XML became a rallying point for most semistructured data management work. XQuery became an official W3C standard in 2007 [61] and is supported by most relational database vendors for querying data in XML-typed columns; several vendors (notably IBM and Oracle) have significant support for native XML storage and indexing. The past several years have seen the arrival and growth of other semistructured data formats; these include JSON [40], Google Protocol Buffers [31], Facebook's Thrift [27], and Avro [6].

In ASTERIX, we are focusing on extending previous work to the parallel realm. We are also designing a data model that is well-suited to handling a wide variety of semistructured data, covering the spectrum from completely unpredictable structures to highly regular (e.g., relational) structures.

3.2 Parallel database systems

Parallelization of relational data management [24] has been one of the biggest research and commercial successes of the database community. Much was accomplished in this area in the decade from the mid-1980s to the mid-1990s. Research systems such as Gamma [23] were successfully developed and showed that shared-nothing architectures and partitioned parallelism can deliver excellent speedup and scale-up results for many query workloads. Commercially successful parallel systems were developed in the same time frame, notably Teradata [8], which eventually became and remains the world's leading high-end data warehousing platform. Parallel joins, sorting, aggregation, and query execution strategies were all products of this work. Progress was also made on data-replication strategies that provide fault-tolerance and seek to minimize load imbalances under partial failures. Research quiesced in the mid- to late-1990s when the database community turned its attention to other areas such as data mining. The past few years have seen renewed commercial interest in this area, with a flurry of companies (AsterData, DatAllegro, Greenplum, Netezza, and others) encroaching on Teradata's parallel relational territory using lower-end commodity clusters.

In ASTERIX, we are building on the same foundation as these new companies, but for more complex, semistructured data, and to support fuzzy as well as traditional (precise) matching queries. In addition, we are aiming at much larger shared clusters and much longer query running times than those considered "large" in the 1990s. This increase in scale complicates questions such as how widely (and where) to parallelize a given query; it also implies a need to support successful query continuation in the presence of faults.

3.3 Data-intensive computing

As mentioned earlier, driven by a need to run analyses over very large data sets, Google's MapReduce [20] and Yahoo!'s open source variant, Hadoop [7], are rapidly gaining adoption as lighter-weight, scalable, parallel data-analysis platforms. These platforms make it possible for "normal programmers" to build scalable parallel programs as long as they can convert their problems and algorithms into a series of map and reduce steps for which they need only supply the corresponding map and reduce functions themselves. The resulting parallel programs can then execute with excellent data scale-up characteristics, utilizing hundreds or even thousands of nodes, and the platforms are designed to ensure forward progress and successful execution even in the face of occasional node failures during executions. However, because the MapReduce framework can be limiting and somewhat low-level as a programming model, efforts such as Pig at Yahoo! [46], Sawzall at Google [48], Jaql at IBM [38], and Hive at Facebook [54] have more recently sprung up. Each of these efforts seeks to provide a higher-level language or framework to developers for data analysis, and they then compile the resulting programs down to MapReduce or Hadoop for scaled-out execution. Microsoft's answer to MapReduce is Dryad [37], a general infrastructure for running data-parallel programs, on which their own higher-level systems like Dryad/LINQ [62] and Scope [15] have been built. Recently, Google has proposed FlumeJava [16], a Java library that helps users to express parallel operations over distributed collections, which are internally compiled into a MapReduce dataflow plan. Another Google project, Dremel [43], performs interactive analysis of read-only nested data using multi-level serving trees that can efficiently answer aggregation queries returning small and medium size results. Also related is the recently introduced Nephele/PACTS system [9]. Nephele/PACTS extends the MapReduce programming model by adding additional second-order operators (e.g., match, cross, and co-group) with additional "your function goes here" plug-in points for use by developers of parallel applications.

In ASTERIX, we are aiming to support similar analysis capabilities, but we are exploring runtime models inspired by parallel database system execution engines. We are doing this to avoid the inefficiencies of some of today's (e.g., Hadoop's) brute force approaches to execution staging and fault-tolerance. Our basic starting point is this simple observation: *A simplified parallel programming model designed for end users is not the runtime abstraction one would have targeted if designing an execution platform intended primarily to support one or more higher-level data analysis and/or query languages*. Given that many companies are adopting Hadoop for data analysis, and thus have IT organizations that are comfortable managing Hadoop clusters, it certainly makes sense for projects like Pig, Hive, and Jaql to target Hadoop with their compilers. On the other hand, since we are in research and taking a longer-term view, we feel that it is (at least) equally important to look at what the "right platform" might be to address the requirements of such languages—i.e., to ask questions such as "what if we'd meant to be building a runtime for higher-level data languages in the first place?" As we will explain later, there are significant performance and efficiency gains achievable by doing so. In addition, doing so is helping us to train a new generation of parallel database and data-intensive computing infrastructure researchers and engineers at UCI, UCSD, and UCR in ways that would not be possible if we limited ourselves to Hadoop.

4 ASTERIX overview

Our goal is to create a scalable platform to store, manage, and analyze large volumes of semistructured information so as to satisfy the requirements that we identified to support evolving-world models. En route to the goal, we are conducting research on various technical aspects of building such a platform, as well as constructing a software system to be shared with the database and data-intensive computing communities via open source distributions. In this section we provide a high-level overview of ASTERIX. More details on both its user model and its implementation will follow in Sects. 5 and 6, respectively.

Data model and query model: To support a wide variety of semistructured data formats, the ASTERIX data storage and query processing are based on its own semistructured model called the ASTERIX Data Model (ADM). Each individual ADM data instance is typed and self-describing. All data instances live in datasets (the ASTERIX analogy to tables), and datasets can be indexed, partitioned, and possibly replicated to achieve various "-ilities" such as scalability and availability. External datasets (i.e., datasets that reside in files that are not under ASTERIX control) are also supported. Datasets may have associated schema information that describes the core content of their instances. ASTERIX schemes are by default "open," in the sense that individual data instances may contain more information than what their dataset schema indicates and can differ from one another regarding their extended content. Data is accessed and manipulated through the use of the associated ASTERIX Query Language (AQL). AQL is designed to cleanly match and handle the data structuring constructs of ADM. It is inspired by XQuery, but omits its many XML-specific and document-specific features.

System architecture: The ASTERIX data storage, query processing, and computational capabilities are targeted at large, shared-nothing, commodity clusters. Figure 1 presents an overview of the main software components of ASTERIX. Its software architecture contains most of the "usual suspects," including a separation of query compilation from runtime plan execution. Areas where ASTERIX is different than other platforms include the compilation/execution division of labor as well as the management of data. (1) Regarding the division of labor, as will be discussed more later, AQL requests are compiled into jobs for an ASTERIX execution layer called Hyracks. ASTERIX concerns itself with the data details of AQL and ADM, turning AQL requests into Hyracks jobs, while Hyracks determines and oversees the utilization of parallelism based on information and constraints associated with the resulting

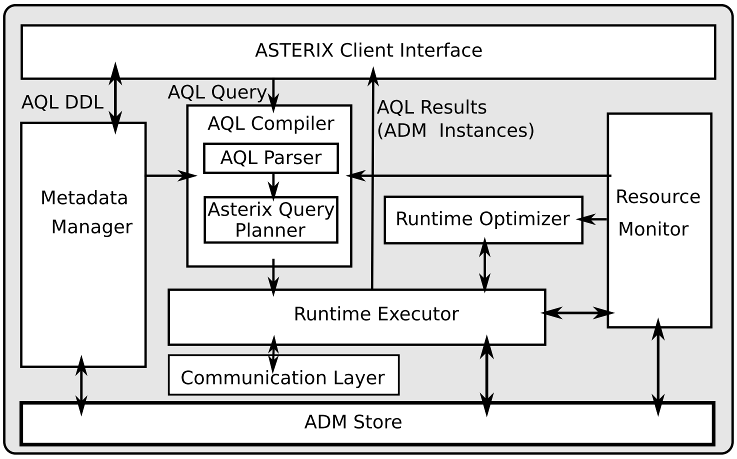

Fig. 1 ASTERIX system architecture

jobs' operators as well as on the runtime state of the cluster. Hyracks itself is intended to support AQL plans, MapReduce-style computation, and more general operator graphs in between, and is therefore multi-purpose in terms of its target customers or end users. (2) Regarding data management, today's data-intensive computing platforms focus on the analysis of data that is usually "just passing through." In contrast, more like a parallel DBMS, ASTERIX aims to store and manage large volumes of data and to simultaneously support short queries, longer analyses, and ongoing updates (the arrival of new versions) to the data in its datasets.

System schematic: Figure 2 presents a schematic indicating how the ASTERIX components are mapped to a cluster of computers. Some of the cluster nodes are designated as *Metadata Nodes*, since they have access to information about the managed datasets as well as information about the state of the system's resources. Query compilation and planning is handled by Metadata Nodes. Other nodes are designated as *Compute Nodes*, since they contain less of the overall ASTERIX software stack, containing just what is needed to store and manage dataset partitions and participate in the execution of ASTERIX query plans and other Hyracks jobs.

5 ASTERIX user model

In this section we provide more information about the user model of ASTERIX, which consists of its data model (ADM) and query language (AQL) targeting semi-structured data. We will illustrate the features of ADM and AQL via a simple scenario involving a hypothetical online service that allows users to register themselves along with their interests. Users are allowed to form special interest groups (SIGs), and SIGs are allowed to have chapters that are local to a geographical region. Once in a while users get together to organize events under the auspices of one or more SIGs. These SIGs are said to be the sponsors of the event. Users are allowed to be members

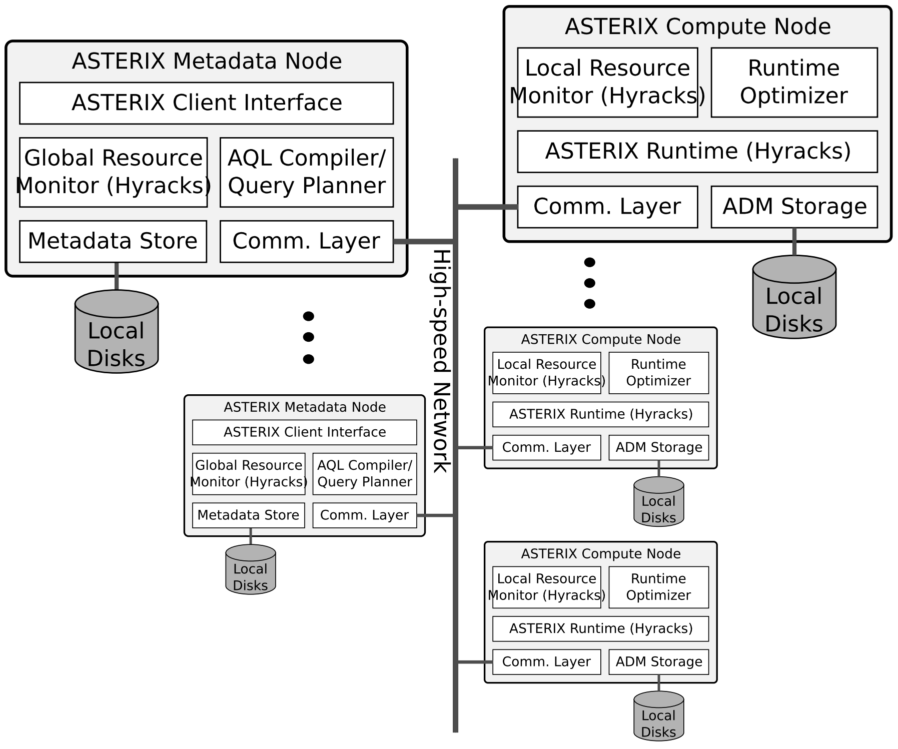

Fig. 2 ASTERIX system schematic

of the SIGs. All users are required to provide certain core pieces of information, such as their email address and regular mailing address, in order to be users of the service. SIGs may add more requirements regarding the information that a user needs to provide in order to become a member of their SIG. For example, a SIG for world travelers might require its members to list the places they have traveled to, while a SIG for movie enthusiasts might require its members to list their top favorite movies.

5.1 ASTERIX data model (ADM)

Basic data types: The ASTERIX data model (ADM) is inspired by the features of popular formats for semistructured data, such as JSON [40] and Avro [6], as well as by the structured data types of object-database systems [50] from which ADM gets bulk types such as sets (actually bags). We rejected XML as a candidate data model for ASTERIX because it brings too many document-oriented concepts (e.g., document order, node identity, mixed content, and its attribute vs. element distinction) that we found undesirable from a clean and simple data-modeling standpoint. We call an instance of the ASTERIX data model a *value*. The type of a value can either be a *primitive* type (int32, int64, string, time, etc., or null) or a *derived* type, which may include:

– *Enum*: an enumeration type, whose domain is defined by listing the sequence of possible values.
– *Record*: a set of fields, where each field is described by its name and type. A record type is either *open* or *closed*. Open records can contain fields that are not part of the type definition, while closed records cannot. Syntactically, record constructors are surrounded by curly braces "{ ... }".
– *Ordered list*: a sequence of values for which the order is determined by creation or insertion. Ordered-list constructors are denoted by brackets: "[...]".
– *Unordered list*: an unordered sequence of values, similar to bags in SQL. Unordered-list constructors are denoted by angular brackets: "$\langle \dots \rangle$".
– *Union*: describes a choice between a finite set of types (as opposed to values in *Enum*). It appears in statically declared or inferred type information, but not necessarily in types seen when inspecting actual data values at runtime. Union type declarations are defined as "$Union(T_1, \dots, T_n)$" where $T_1, \dots, T_n$ are the choices.

Figure 3 shows one possible way to model the data backing the example scenario in ASTERIX. User information is modeled as a `UserType` record with core fields such as the user's name, email address, interests, address, SIG membership, and a unique user_id. The user record type is indicated as open to allow additional structured content to be associated with a user based on their SIG membership. A membership in SIGs is modeled as an unordered list of records identifying the SIG and the chapter the user belongs to, along with the date when they initially became a member.

The `SIGType` describes records representing special interest groups. Notice that the containment of chapters in SIGs is modeled as an unordered list of nested records. The `AddressType` describes records representing addresses. The `EventType` describes event records. The "sponsoring_sigs" field is defined as an ordered list to model the fact that the ordering of sponsors is important. The "?" symbol means that the "price" field is optional. The "*double?*" notation is actually just syntactic sugar for *Union(double, null)*.

Container model: A named collection of data in ASTERIX is called a *dataset*. A dataset may be partitioned over multiple hosts in a cluster. Partitions could be replicated on other hosts to allow parallel access as well as to provide fault tolerance. An ASTERIX dataset is loosely analogous to a table in a relational database; i.e., a dataset is a target for AQL queries and updates and is also the attachment point for indexes. A collection of datasets related to an application are grouped into a namespace called a *dataverse*, which is analogous to a database in the relational world. Figure 3(e) shows DDL (data definition language) commands to create a dataverse "SIGService" with datasets "User", "SIGroup", and "Event". As shown, the DDL includes the specification of a primary key for each dataset (currently required for all datasets) as well as a partitioning key for the system to use in logically distributing the data across the nodes of a cluster. By default, the partitioning key is also the primary key. (The current system also accepts additional physical placement specifications, relating to nodes and node groups, much like the table DDL features in current parallel databases.) Also, as mentioned earlier, external datasets are supported as well; they are very similar to external tables in Hive [54] and require the provision of location- and format-related information in their definitional DDL. That is, as in Hive, external datasets are proxies for data in files (Hadoop files or local files) that lie outside

```
declare type UserType as open {
  user_id: int32,
  name: string,
  email: string,
  interests: <string>,
  address: AddressType,
  member_of: <
    {
      sig_id: int32,
      chapter_name: string,
      member_since: date
    }
  >
}
```

(a)

```
declare type EventType as closed {
  event_id: int32,
  name: string,
  location: AddressType?,
  organizers: <
    {
      name: string,
      role: string?
    }
  >,
  sponsoring_sigs: [
    {
      sig_id: int32,
      chapter_name: string
    }
  ],
  interest_keywords: <string>,
  price: double?,
  start_time: datetime,
  end_time: datetime
}
```

(b)

```
declare type AddressType as closed {
  street: string,
  city: string,
  zip: string,
  latlong: point2d
}
```

(c)

```
declare type SIGType as closed {
  sig_id: int32,
  name: string,
  interests: <string>,
  chapters: <
    {
      name: string,
      created_on: date,
      location: AddressType
    }
  >
}
```

(d)

```
create dataverse SIGService;
use dataverse SIGService;

create dataset User(UserType)
   partitioned by key user_id;

create dataset SIGroup(SIGType)
   partitioned by key sig_id;

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

(e)

Fig. 3 Metadata definition for the running example

ASTERIX control and involve the specification of readers to interpret the file data as collections of ADM object instances.

Open data types: While it is able to express nested records and collections, ADM is different from nested relational [2] and object-oriented [45] data models which are strongly (and statically) typed. Open records give ASTERIX the flexibility to cover a full range of typing possibilities, ranging from statically untyped data (e.g., an open record type with no a priori declared fields) to strongly and statically typed data (e.g., a completely closed record type). The ability to have open schemes is critical for ASTERIX to address some of the key requirements for evolving-world modeling, including flexible typing and also tolerance for evolution (e.g., data sets can have ADM instances that have more fields than were initially anticipated). Open data types with self-describing data also provide excellent support for use cases involving collections of highly variant and/or "sparse" objects.

In Fig. 4, we exemplify our data model by showing a few records that could appear in the Event dataset. Datasets are always unordered collections, and we denote this in the figure by using angular brackets. Note in the figure that the optional "role" field is

```
< {
    "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" )
  } >
```

Fig. 4 Data from the Event dataset

missing from all of the unordered lists of organizers, and in event 1023 the optional price field is also missing. The ADM record for the organizer of event 941 also has an "affiliation" field. This extra field at the instance level is allowed since ADM records are open by default.

5.2 ASTERIX Query Language (AQL)

The ASTERIX Query Language (AQL) borrows from XQuery 1.0 [61] and Jaql 0.1 [39] their programmer-friendly declarative syntax that describes bulk operations such as iteration, filtering, and sorting. As a result, AQL is comparable to those languages in terms of expressive power. The major difference with respect to XQuery and XML is AQL's focus on data-centric use cases at the expense of built-in support for mixed content for document-centric use cases. In ASTERIX, there is no notion of document order or node identity for data instances, and (messy) distinctions such as attributes versus elements and support for duplicate elements as a way to model lists have been eliminated, both in ADM and AQL. Differences between AQL and Jaql stem from the usage of the languages: ASTERIX data is stored in and managed by the ASTERIX system, while Jaql (like Hadoop) runs against data stored externally in Hadoop files or in the local file system.

In this section we use examples to illustrate key features of the AQL language. The examples are asked against records stored in the datasets from Sect. 5.1. Datasets can be accessed using the built-in "*dataset*" function. For instance, the function call *dataset('User')* exposes a collection of users with the type "UserType".

5.2.1 Basic queries

Q1. (*Simple AQL Query*) Find the names of all users who are interested in movies:

```
for $user in dataset('User')
where some $i in $user.interests
     satisfies $i = "movies"
return { "name": $user.name }
```

Note that AQL uses the classic dot "." syntax for field access and that record construction takes an expression (in this case the string constant "name") for a field name.

Q2. (*Grouping and nesting in AQL*) Out of the SIGroups sponsoring events, find the top 5, along with the number of events they have sponsored, cumulatively and broken down by chapter. (If two chapters of a SIGroup sponsor an event, it counts as 2 for the SIGroup):

```
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 }
```

This query starts by constructing a record for each sponsored event and then groups these records by the id of the sponsoring SIG. The first "`group by`" re-binds the "$es" variable via its "`with`" clause to the list of events grouped under the same sponsor id. The list is passed, on the following line, to the "`count`" aggregate, used for sorting the output. The chapter breakdown is computed in a nested query that similarly groups each list of sponsored events by their sponsoring chapter names. The outer query includes an order by clause and a limit clause, a combination expected to be common (and important to optimize) when querying evolving-world data, as such data may often be too voluminous for un-limited querying.

The result of running Q2 over the input data from Fig. 4 is the following ordered collection containing two records corresponding to the two SIGs with ids 14 and 31, respectively:

```
[ {
    "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 } ]
   } ]
```

Q3. (*Fuzzy set-similarity in AQL*) For each user, find the 10 most similar users based on their interests. As described in Sect. 5.1, the users' interests are represented as an unordered list of strings. To decide if two users have similar interests, we treat the list of interests as a set, and compute the Jaccard similarity[1] between the two sets of interests (the strings representing each interest have to match exactly between the sets). Two users are similar if their interest-set similarity is above a certain threshold (e.g., 0.75); ~= is the similarity operator in AQL, and its with-clause specifies the desired similarity measure and threshold.

```
for $user in dataset('User')
let $similar_users :=
    for $similar_user in dataset('User')
    let [$match, $similarity] :=
       $user.interests  ~= $similar_user.interests
       with simfunction 'jaccard', simthreshold '.75'
    where $match
    order by $similarity desc
    limit 10
    return { "user_name" : $similar_user.name, "similarity" : $similarity }
return { "user_name" : $user.name, "similar_users" : $similar_user }
```

For each user, the query asks for other users who have a Jaccard similarity of at least 0.75 based on their sets of interests. The candidate users are ordered decreasingly by the similarity score, and only the top 10 are returned.

5.2.2 Update and versioning queries

Updates and transactions in ASTERIX have been designed only to a very preliminary extent. Like other systems focused on scalability [18, 22], ASTERIX plans to support the option to implement updates by creating new versions of data rather than modifying existing data instances "in place." For transactions, we are currently working to figure out how to turn Helland's prescription for scalability [34] into a transaction model that ASTERIX users can hope to understand and use. For versioned datasets, similar to transaction-time temporal databases [51], ASTERIX will associate with each version the completion time of the transaction that created the version. Even for non-versioned datasets, since it has been long known that multiversion concurrency control dramatically improves multiuser performance [14] for workload mixes like those anticipated for managing evolving-world models (i.e., small update requests running concurrently with much larger read-only requests), a transient multiversion strategy will likely be employed. In AQL, one can specify over which versions of the data the query criteria should be applied by giving a time interval that includes the transaction times of the desired versions. The following examples illustrate these concepts.

Q4. (*Simple update in AQL*) Update the user named *John Smith* to contain a field named *favorite-movies* with a list of his favorite movies:

[1] The Jaccard similarity between two sets x and y is $jaccard(x, y) = \frac{|x \cap y|}{|x \cup y|}$.

```
replace $user in dataset('User')
where $user.name = "John Smith"
with (
    add-field($user, "favorite-movies", ["Avatar"])
)
```

The “replace” clause in Q4 replaces each item in the “in” list that matches on the “where” condition with the value produced by the expression in the “with” clause. The built-in function “add-field()” takes as the first argument a record, a field name as the second argument, and the field value as the third. It returns a record that has the new field with the given value in addition to the fields of the input record. In effect, we replace the “User” record whose name field has the value “John Smith” with the contents of the same record and a new field called “favorite-movies”. Note that the *favorite-movies* field is not present in the UserType definition, but the update query is legal because we declared UserType as an open record.

Q5. (*Historical data access in AQL*) List the SIGroup records added in the last 24 hours:

```
for $current_sig in dataset('SIGroup')
where
     every $old_sig in dataset('SIGroup',
          getCurrentDateTime() - dtduration(0, 24, 0, 0))
     satisfies $old_sig.sig_id != $current_sig.sig_id
return $current_sig
```

The “*dataset*” function takes a second argument of datetime type and returns the latest version of the data items prior to that datetime value. The day-time function call “dtduration(0, 24, 0, 0)” constructs a duration of 24 hours which is subtracted from the result of a function call that returns the current time from ASTERIX as of the start of the query’s execution. The second (inner) invocation of “dataset()” yields all records in the SIGroup dataset as of 24 hours ago. The fact that AQL allows users to access multiple versions of a dataset within one query makes it possible to inspect the evolution of data over time; this is another necessary feature to support evolving-world models.

6 ASTERIX under the Hood

To process distributed queries over large collections of data, we face several challenges. At compile time, we need to produce a good-quality plan that can be run efficiently over the potentially large cluster. This task is generally difficult, given that the values of some parameters are unknown at compile time and the space of possible plans is large [35]. At runtime, we need to make good use of the available resources and to be flexible in planning decisions, as resource availability may change.

The implementation of the overall ASTERIX system is currently underway. We have built the first version of its distributed runtime platform, called Hyracks [12], which provides support for executing AQL queries and performs runtime scheduling and execution coordination. In addition to its role as the runtime for AQL, Hyracks is standalone and re-usable in the sense that it can also be used to execute other data-parallel computations, e.g., MapReduce tasks and generalizations thereof. An initial version of Hyracks is now available in open source form on Google Code for interested early adopters [36]. ADM storage and AQL compilation are less far along, but basic storage and queries are now running end-to-end and their implementation is being refined and the fullness of their DDL and DML support is expanding. ASTERIX itself will also be released via open source once the system is sufficiently complete and robust. This section describes our initial design choices for implementing AQL and explains how we schedule and execute parallel jobs using the Hyracks runtime.

6.1 Implementing AQL

Algebra and operators: ASTERIX builds a logical plan for each query. A plan is a directed acyclic graph (DAG) of algebraic operators that are similar to what one might expect to find in a nested-relational algebra [44], e.g., select, project, join, group-by, unnest, and other operations over streams of tuples. The algebra is accompanied by an expression language that encodes data-model-specific computations such as filter predicates or construction of new data values. This separation of "bulk type operations" and "instance operations" was motivated by a desire to make the algebra reusable. In addition to AQL, we would like to offer query and analytic access to ASTERIX data through currently popular languages such as Hive or Pig, and we believe that having a reusable algebra layer will make it relatively easy to support other ASTERIX "skins" such as those as well as preparing the system architecturally to someday support user-defined primitive types.

For space reasons, we do not describe the query algebra formally here. Instead, we present an example involving the logical plan (shown in Fig. 5) obtained via direct translation of query Q2 described in Sect. 5. Each operator receives as input a sequence of tuples and produces a sequence of tuples. The operator syntax used is:

$$\text{opname list-of-arguments} \rightarrow \text{output-columns.}$$

In Fig. 5, column names start with a "\$" sign in order to highlight their relation to the AQL query. Tuples represent simultaneous bindings of values to AQL variables. The tuple stream flows bottom-up. It starts with an empty tuple (Line 30 in the figure), and populates the columns \$*event* and \$*sponsor* (Lines 29 and 28). Note that the sponsor instances start out being directly nested inside the event records. Lines 26–27 show how the \$*es* column is initialized with newly created records containing pairs of events and sponsors. The *sig_id* field is extracted from \$*sponsor* and used by the *group by* clause that computes a nested plan, delimited by curly braces (Lines 21–24). The nested plan concatenates all \$*es* values (the *listify* operator), as required by the semantics of the `group by` clause in AQL, and stores this list in the $\$tempo_0$ column of all the tuples. As we will see in the optimized plan (Fig. 6), it would be more efficient to bypass list materialization and compute the *count* aggregate directly inside the *group by*. The *subplan* operator, corresponding to the traditional dependent product, computes the breakdown by chapter in its nested plan.

Query planning: AQL queries first go through a static optimization process that uses heuristics and information about physical-operator characteristics including partition- and order-related properties. The result is a plan (i.e., a Hyracks job) where all physical operators are bound. Hyracks is free to change the degree of parallelism, the

Fig. 5 Logical plan for query Q2

```
listify $temp_5 → $temp_6
assign { "sig_id", $sig_id,
         "total_count", $sig_sponsorship_count,
         "chapter_breakdown", $temp_4 } → $temp_5
limit 5
order by $sig_sponsorship_count desc
subplan {
  listify $temp_3 → $temp_4
  assign { "chapter_name", $chapter_name,
           "count", count($temp_2)} → $temp_3
  group by $chapter_name {
    listify $es → $temp_2
    nested-tuple-source
  }
  assign $temp_1.sponsor.chapter_name
         → $chapter_name
  unnest $temp_0 → $temp_1
  nested-tuple-source
}
assign count($temp_0) → $sig_sponsorship_count
group by $sig_id {
  listify $es → $temp_0
  nested-tuple-source
}
assign $sponsor.sig_id → $sig_id
assign { "event", $event,
         "sponsor", $sponsor } → $es
unnest $event.sponsoring_sigs → $sponsor
unnest dataset('Event') → $event
empty-tuple-source
```

amount of resources provided to the operators, and the location for the execution of the operators on the cluster subject to constraints provided by the query optimizer. Hyracks job planning is done in stages at runtime; after each stage, new optimization decisions may therefore kick in. The basic plan topology (e.g., operator and connector choices) created during static optimization in ASTERIX is unchanged, however.

In the future, Hyracks jobs created by ASTERIX may also include *choose-plan* [32] operators that depend on parameter values to be bound later, i.e., at runtime. A job could therefore start with a pre-phase that runs specifically with the intent of filling in these parameters, for instance, by sampling or by gathering statistics (possibly during earlier job stages). Executing each of the subsequent stages produces parameter values that are fed to the choose-plan operator so that the best decision based on current information can be taken. As will be mentioned in Sect. 7, experimental exploration of dynamic query processing techniques, as well as their interactions with fault-tolerance, are among the major research-agenda items for the ASTERIX project.

Storage and indexing: Items in a dataset can be stored and indexed based on values of their type components on any nesting level. For example, we could index events in our running example on their name and/or also on their organizer names. Currently, storage and indexing in ASTERIX are based on the use of primary and secondary B+ tree indices. Each dataset is represented as a key-partitioned collection of local B+ tree primary indexes; these indexes are keyed on the primary key of the dataset and their leaves contain the ADM instances themselves. ADM instances are stored in a binary format that factors out the declared portions of both closed and open types for space efficiency. Secondary B+ tree indices are currently local-only, meaning that secondary indexes for a dataset are local B+ tree indexes that are co-located with the primary B+ tree and map (only) that partition's secondary key values to their corresponding primary keys. The storage manager layer is both "distribution unaware" and "index unaware," meaning that the AQL compiler owns the responsibility of generating loading, query, and insert/update/delete plans that involve all the affected partitions and keep the full set of relevant indexes in sync with the primary data values. In terms of supported index types, in addition to B+ trees, keyword indexes and spatial indexes are both currently under development. We are also working on adding indexing capabilities to efficiently support fuzzy queries; we have already studied the problem of supporting fuzzy queries on a single node using external indexing [11].

6.2 AQL runtime

As explained before, an AQL query undergoes translation and then optimization. Last but not least, it is handed to the runtime module to be evaluated. To execute AQL queries, ASTERIX uses our generic parallel runtime platform, Hyracks, which we describe here through the use of an example. We also describe the Hadoop compatibility interface provided on top of Hyracks, which enables existing Hadoop applications for data-intensive analyses to be easily migrated to a Hyracks cluster and to co-exist on the cluster with ASTERIX-generated Hyracks jobs. A more extensive description of Hyracks, along with a first evaluation of its performance, can be found in [12].

6.2.1 Hyracks overview

Hyracks is a generalized alternative to infrastructures such as MapReduce, Hadoop, and Dryad for solving data-parallel problems. It balances the need for expressiveness beyond MapReduce, which offers a very limited programming model based on a few user-provided functions, while providing out-of-the-box support for many commonly occurring communication patterns and operators needed in data-oriented tasks, which are absent in Dryad [12].

Hyracks has been designed to work with a cluster of commodity computers. To do so in a robust manner, it has built-in support to detect and recover from possible system failures that might occur during the evaluation of a job through the use of heartbeats. A Hyracks cluster has two kinds of nodes: *cluster controllers* and *node controllers*. The cluster controllers monitor the health of the node controllers while the latter are used to evaluate user jobs. Cluster controllers are also responsible for interacting with clients and providing them with a single system image (SSI). As shown in Fig. 2, when Hyracks is running in the context of ASTERIX, the Metadata Nodes house the cluster controller modules in the Global Resource Monitor.

Hyracks jobs: Hyracks jobs are directed acyclic graphs (DAGs) of *Hyracks Operator Descriptors* (HODs). An HOD is similar to the execution-plan-tree node described in [28]. ConnectorDescriptors (CDs) are used to connect HOD nodes. Note that Hyracks has no visibility into the logic or semantics of an HOD node or a CD. The connectors between the HOD nodes provide Hyracks with data-dependency information. The connector encapsulates the exact data-distribution logic to use at runtime. In order to provide Hyracks with control dependencies in a job, we use Hyracks Activity Nodes (HANs). An HOD can expose to Hyracks the distinct set of activities it performs through the creation of one or more HANs. Hyracks uses HANs to divide the job into stages and decide in what order the stages are to be evaluated. Once Hyracks has determined the amount of parallelism for each stage, the HANs are asked to create a set of *Hyracks Operator Nodes* (HONs). The HONs are essentially clones that are responsible for the actual execution of the activities of the operations and for producing results.

Hyracks includes a library of HOD and CD implementations that are generally applicable to building data-oriented programs. For example, the sort HOD used in the example below is a part of the core library. Similarly, connectors such as "1:1" and "M:N hash-merge" (used below) are also components of the library. ASTERIX uses these core Hyracks operators and connectors when applicable and then implements a set of additional HOD nodes that are specialized to its needs.

We will use the example query Q2 from Sect. 5.2 to describe the metamorphosis of a job in Hyracks that leads finally to its parallel evaluation. Execution of Q2 entails the following logical steps:

– For each partition of Events, produce a local grouping on sig_id and chapter_name along with the count of Events that agree on this pair of values.
– Redistribute the output to a set of machines such that all values with the same sig_id arrive at the same machine.
– Sum the local counts for each unique value of sig_id to compute the total count of chapter_names and also compute per-chapter counts.
– Merge the outputs to one machine and limit the results to 5 records.

Figure 6 depicts the Hyracks job specification of the running example. Hyracks is oblivious to the internal logic of each HOD node, but for ease of exposition, we list the logical operators from the ASTERIX optimized plan inside each HOD node. Some of the HOD nodes correspond directly to Hyracks operators (e.g., sort), while others are ASTERIX container operators that hold short sequences of micro-operators that are locally chainable. The job in the example gets its input from the Event dataset. To keep the example small, we assume that the Event dataset is partitioned on three hosts.

HOD 1 contains the logic to scan a partition of the Event dataset and project the *sig_id* and *chapter_name* fields from each record. HOD 1 is connected to HOD 2 using a 1:1 connector. The 1:1 connector tells Hyracks that the amount of parallelism for HOD 2 is the same as that for HOD 1. Each runtime instance of HOD 2 is a sort

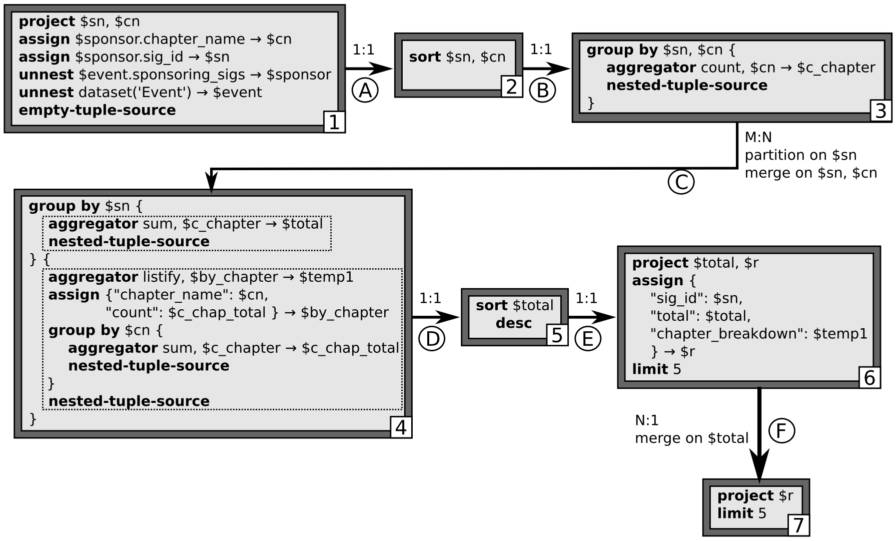
project $sn, $cn
assign $sponsor.chapter_name → $cn
assign $sponsor.sig_id → $sn
unnest $event.sponsoring_sigs → $sponsor
unnest dataset('Event') → $event
empty-tuple-source
1
1:1
A
sort $sn, $cn
2
1:1
B
group by $sn, $cn {
aggregator count, $cn → $c_chapter
nested-tuple-source
}
3
M:N
partition on $sn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

Fig. 6 Hyracks job specification for query Q2

operator that locally sorts the stream of ($*sn*, $*cn*) pairs generated by one instance of HOD 1. The output of the sorter is sent to the next operator, HOD 3, also over a 1:1 connector. This operator performs grouping on ($*sn*, $*cn*) and computes the number of $*cn* values.[2]

At the output of HOD 3, we have, for each partition of Events, a stream of ($*sn*, $*cn*, $*c_chapter*) triples that is sorted on ($*sn*, $*cn*). After sorting, these streams across all partitions are merged using an $M{:}N$ hash-merge connector. This connector distributes each datum produced by each of M senders using a provided hash function (in this case hash over $*sn*) to pick one of N receivers. At each receiver, it merges all incoming data to maintain the specified sort order, which is over ($*sn*, $*cn*) in the example.

Each instance of HOD 4 receives a partition of data such that it contains all instances that agree on a particular $*sn*. HOD 4 groups this stream on $*sn* and computes two results. (The group-by operator in ASTERIX is able to evaluate multiple nested plans for each created group.) $*total* contains the sum of local counts computed in HOD 3, which is equivalent to the total number of chapters of the SIG $*sn* that are sponsors of events in the Event dataset. $$temp_1$ is bound to a list of records containing *chapter_name* and *count* fields—one record for each unique *chapter_name* value. Note that the inner group-by exploits the secondary sort on $*cn* maintained by the hash-merge connector.

The output of HOD 4 is sent to a local sorter (HOD 5) to sort the stream in a descending order of $*total*, through a 1:1 connector. The output of the local sort is sent to HOD 6 also through a 1:1 connector. HOD 6 limits the results to the first five

[2]In this example the group by operator is assumed to be a pre-sorted variant that takes its input sorted on the grouping values.

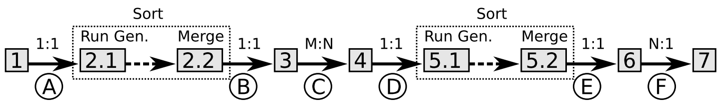

Fig. 7 Example Hyracks Activity Node graph for Q2

records produced by HOD 5. This limit was introduced by the compiler, owing to the limit clause in the query, to throw away irrelevant results early. A further optimization (not shown for presentation purposes) would be to push the limit functionality into the sorter so that excess comparisons or possible materialization can be eliminated.

Finally, the results of HOD 6 instances are merged into one instance of HOD 7 as indicated by the N:1 merge connector. The sort order maintained by the merging connector is on \$*total*. HOD 7 once again limits the input to five objects and performs a projection on the result \$*r*.

Scheduling and execution: The next step in job planning is to expand each HOD into a set of HANs. Figure 7 shows the HANs for the HOD graph in Fig. 6. Notice that the sort HODs 2 and 5 have been replaced with two activity nodes each. This is because a sort operation works in two phases: in the first phase, it builds sorted runs, and in the second phase, it merges the runs to produce output. The second phase (merge) of the sort cannot begin until the run-generation phase completes. This requirement is indicated in the HAN graph by the use of a dotted (blocking) edge from the run-generation phase to run-merging phase. All other HODs are composed of a single activity each, hence there is exactly one HAN node for each of them. All HANs that are connected only by non-blocking (solid) edges are grouped into a stage, splitting the complete job into independent stages. The stages are planned and executed by Hyracks in such a way that all of a stage's dependencies are complete before it is executed. Hyracks can select the amount of parallelism and the placement of the HAN instances based on node affinities and resource requirements. For example, the HAN corresponding to HOD 1 would indicate that it should be instantiated where its input dataset partitions are located. Once a stage is planned, the HAN instances for the stage are instantiated in participating Hyracks node controllers to form a DAG of Hyracks Operator Nodes (HONs) at runtime.

Both planning the degree of parallelism and placing HONs for a stage are delayed until the stage is ready to run, allowing current cluster conditions to play a role in the process. The HONs are responsible for consuming data from their inputs, performing computation, and sending outputs to their consuming HONs. Figure 8 shows the HON graph for the example. Here, we assume that the Event dataset was stored as three partitions, while Hyracks decided to use four machines to perform the final grouping. Although we show the completely expanded graph with all operator nodes for all stages, in reality, this expansion is performed stage by stage. Once a stage completes the execution, its resources are relinquished. The dotted rectangles in the figure show all operator nodes that belong to a stage.

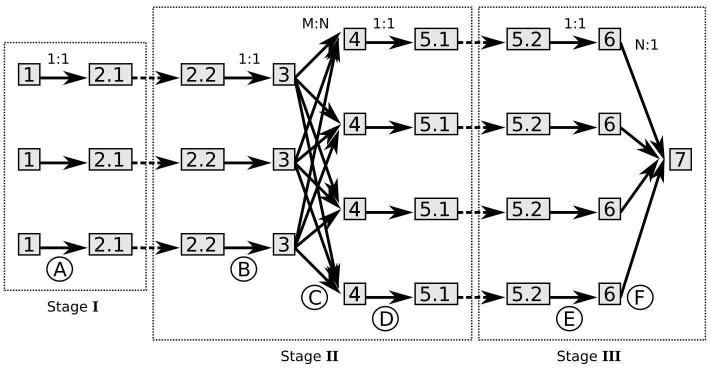

Fig. 8 Hyracks Operator Node graph for Q2

6.2.2 Hadoop compatibility

Given the adoption of Hadoop as a platform for data-analysis tasks, we believe that ASTERIX must provide an easy migration path for existing Hadoop projects in order to have any shot at being a serious replacement candidate. In that spirit, we have implemented a Hadoop compatibility layer on top of Hyracks so that existing Hadoop programs can be executed atop Hyracks. In this section we first give a brief introduction to MapReduce and Hadoop, then describe our emulation layer.

MapReduce has become a very popular programming paradigm for data-intensive parallel computing on shared-nothing clusters. Example applications for MapReduce include processing crawled Web documents, analyzing large Web-server logs, and so on. In the open source community, Hadoop is by far the most popular implementation of this paradigm. In MapReduce and Hadoop, data is initially partitioned across the nodes of a cluster and stored in a distributed file system (DFS). Data is represented as `(key, value)` pairs. The computation is expressed using two functions:

```
map    (k1,v1)        → list(k2,v2);
reduce (k2,list(v2))  → list(k3,v3).
```

The computation starts with a map phase in which the `map` functions are applied in parallel on different partitions of the input data. The `(key, value)` pairs output by each `map` function are hash-partitioned on the `key`. For each partition the pairs are sorted by their `key` and then sent across the cluster in a shuffle phase. At each receiving node, all the received partitions are merged in sorted order by their `key`. All pair values that share a given key are passed to a single reduce call. The output of each `reduce` function is written to a distributed file in the DFS.

To allow users to run MapReduce jobs developed for Hadoop on Hyracks, we developed two extra HODs that can wrap the `map` and `reduce` functionality. The data tuples provided as input to the *hadoop_mapper* HOD are treated as `(key, value)` pairs and, one at a time, are passed to the `map` function. The *hadoop_reducer* HOD also treats the input as `(key, value)` pairs, and groups the pairs by `key`. The sets of `(key, value)` pairs that share the same `key` are passed, one set at a time, to

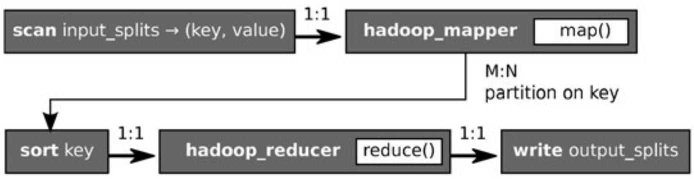

Fig. 9 Hyracks plan for Hadoop jobs

the `reduce` function. The outputs of the `map` and `reduce` functions are directly output by the HODs. To emulate a MapReduce framework we also use a sorter HOD and a hash-based distribution of data. The Hadoop compatibility layer of Hyracks includes a Hadoop-to-Hyracks plan generator that takes a given Hadoop job description, together with code for its `map` and `reduce` functions, and generates a Hyracks plan that uses the given `map` and `reduce` implementations and provides the same computational functionality.

Figure 9 shows the Hyracks plan for running a MapReduce job. After data is read by the scan HOD, it is fed into the *hadoop_mapper* HOD using a 1:1 edge. Next, using an $M{:}N$ hash-based distribution edge, data is partitioned based on `key`. The "M" value represents the number of maps, while the "N" value represents the number of reducers. After distribution, data is sorted using the sort HOD and passed to the *hadoop_reducer* HOD using a 1:1 edge. Finally, the *hadoop_reducer* HOD is connected using a 1:1 edge to a file-writer HOD.

One of the unique goals of ASTERIX is to support parallel fuzzy queries. In [57] we described several techniques for performing set-similarity joins in MapReduce. We divided the problem into three stages, each one using one or two MapReduce jobs. The techniques presented there were implemented in Hadoop. As a case study for testing Hyracks compatibility with Hadoop, we have successfully run the Hadoop code developed in [57] unchanged inside Hyracks.

6.2.3 Hyracks performance

To provide a sense of the efficacy of the Hyracks platform, Fig. 10 shows the results of a performance experiment from [12] comparing the execution times for a simple TPC-H-like query on Hyracks versus Hadoop. For each platform, the goal for their respective data-parallel jobs was to compute the equivalent of the following SQL query over various scales of TPC-H data:

```
select C_MKTSEGMENT, count(O_ORDERKEY)
from CUSTOMER join ORDERS on C_CUSTKEY = O_CUSTKEY
group by C_MKTSEGMENT
```

In Hyracks, for each data source (CUSTOMER and ORDERS), a file scanner operator was used to read the source data. A hash-based join operator received the resulting streams of data (one with CUSTOMER instances and another with ORDERS instances) and produced a stream of CUSTOMER-ORDERS pairs that matched on the specified condition (C_CUSTKEY = O_CUSTKEY). The result of the join was then

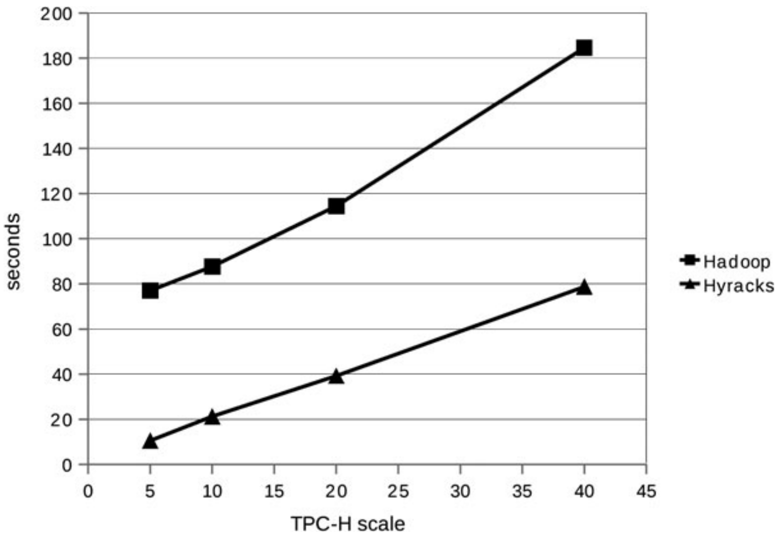

Fig. 10 TPC-H query with Hadoop and Hyracks

aggregated using a hash-based group operator on the value of the C_MKTSEGMENT field. This group operator was provided with a COUNT aggregation function to compute the count of O_ORDERKEY occurrences within a group. Finally, the output of the aggregation was written out to a file using a file writer operator.

For Hadoop, two MapReduce jobs were used to perform the join followed by the aggregation. The Mapper of the first job processed the source data files and generated a value field as a tagged output indicating whether the record came from the CUSTOMER source or the ORDER source. The associated key value was generated by extracting either the C_CUSTKEY field (from CUSTOMEER records) or the O_CUSTKEY (from ORDER records). The Reducer of the first job then received all of the records (from both CUSTOMER and ORDER) that matched on the (join) key; its task was to generate CUSTOMER-ORDER pairs that became the input for the second MapReduce job. The second MapReduce job performed the aggregation to compute the desired count, grouping the data on the value of the C_MKTSEGMENT field. The Mapper of the second job produced the value of C_MKTSEGMENT as its output key and the CUSTOMER-ORDER pair as the associated value. The second job's Reducer then received a list of CUSTOMER-ORDER pairs for a given value of C_MKTSEGMENT, which it counted; the final output was emitted as pairs of C_MKTSEGMENT and count values.

The performance results in Fig. 10, covered in detail in [12], were produced on a 40-core/40-disk cluster at UC Irvine. As shown, Hyracks significantly outperforms Hadoop for such computations. Reasons for Hyracks' superior performance include its use of pipelining versus Hadoop's reliance on HDFS and temporary files for data transmission, Hyracks' support for binary operators such as joins versus Hadoop's unary operator model (necessitating tricks such as tagging/stacking Hadoop's join inputs), Hyracks' ability to utilize hash-based algorithms versus Hadoop's strictly sort-based Reducer contract, Hyracks' support for arbitrarily long operator chains versus Hadoop's limit of one Mapper and one Reducer per job, and a heavy focus in Hyracks on minimization of data copying and Java object generation in its runtime operators and data paths. More information about this experiment can be found in [12] together with a number of related experiments involving computations such as K-means and fuzzy joins and one initial experiment exploring the tradeoffs related to query execution in the presence of faults for Hyracks versus Hadoop.

7 Research challenges

In this section, we outline some of the research challenges that we are attempting to tackle under the ASTERIX umbrella.

7.1 Modern storage and indexing

The ASTERIX project is working to innovate in two dimensions related to storage and indexing technologies. First, on the basic storage front, existing storage management techniques tend to be "black or white" with respect to structures; most are designed either for highly regular, statically typed data, where schema and data are separate, or for self-describing, non-statically typed data, where each datum must be completely self-describing. We are exploring the continuum in between, looking at ways for commonalities to be factored out of the data (automatically, in an ideal world), so that storage costs depend on how much the instances in a data set have in common rather than on a 0/1 decision between self-describing and non-self-describing formats. Second, on the scalable storage and indexing front, again existing systems fall mostly into one of two camps. In the parallel database camp [24], tables and indexes are explicitly partitioned by the DBA, who must either pick "over all nodes" as the answer to how to partition a table or else explicitly micromanage the layout of the partitions across the cluster. In the Web-scale distributed file system camp [17, 29], the data distribution is self-managing, but access to the data is typically limited to access via a primary key or full file scans. The scale of our target environment demands that we seek a hybrid approach, with richer storage and access options à la parallel databases, but with the self-managing placement characteristics of modern distributed file systems and "big table" managers.

7.2 Fuzzy data and queries

Supporting fuzzy queries (queries with predicates involving string similarity or set-similarity conditions) is a very important aspect of a data management system, especially when the data is coming from the Web and contains mismatches and inconsistencies. A wide variety of main-memory-single-node solutions have been proposed in the literature for various kinds of fuzzy queries (selection queries with similarity thresholds [41], ranking queries [58], and join queries [60]). More recent work proposed external-memory-single-node solutions for answering fuzzy selection queries [10] and MapReduce-based solutions for answering fuzzy join queries [57]. In ASTERIX, we are treating fuzziness as a first-class operator. We are innovating in two directions related to fuzzy query processing. First, we have designed efficient, parallel, disk-based indexing techniques and query-answering algorithms for fuzzy queries over indexed data [11] that we plan to incorporate into ASTERIX. Second, we are designing efficient fuzzy-query-answering techniques for ad-hoc queries on unindexed data.

7.3 Dynamic parallel query processing

Most existing parallel database management systems employ a rather static strategy for query execution. The query compiler compiles and optimizes a given query using its best guesses based on costs and initial data sizes. After this compilation phase, the executor executes the query on a cluster of nodes. During the execution phase, the algorithmic choices of operators (decided by the compiler) are generally not revisited based either on actual available resources or observed data sizes. The dynamic nature of our target environment demands that we explore systematic ways of deferring query planning, interspersing it with query evaluation, so as to consider the state of the world when the operator is ready to run. Options under consideration range from completely incremental planning as in Ingres [59] to Graefe's *choose-plan* operator [32]. Architecturally, we are seeking to determine the appropriate frequency and nature of interactions between the AQL compiler, its produced Hyracks jobs, and the Hyracks scheduler. The Hyracks Operator Descriptor interface includes provisions for the operators chosen for use in executing an AQL query to inform Hyracks of their resource (e.g., CPU, I/O, and memory) requirements to enable Hyracks to exploit that information when making parallelism and operator placement decisions.

7.4 Selective fault-tolerance

The aim of ASTERIX is to run user queries over data residing on very large clusters. As the number of components in a system grows, the chances of faults grow as well. As Google has reported, fault-tolerance must be integral in very large systems to mitigate the effects of faults [29]. Systems such as GFS and Hadoop provide a static brute-force answer to this problem—e.g., making three copies of all results after every job. Such a technique is simple and effective, but it adds both space and time overheads to all computations. In contrast, traditional databases restart the entire transaction, which may be an expensive strategy when running long computations on large clusters.

We believe that a more selective fault-tolerance mechanism is called for in ASTERIX, one that takes the actual task producing the intermediate results into consideration. If job A feeds results to job B and both jobs are small, running the jobs concurrently and pipelining results from one to the other might be more efficient than running job A, materializing the results, and then having job B read them. If there is a failure, the two jobs could simply be re-run from the beginning at a fairly low cost. For longer-running jobs, however, materialization would be critical to avoid costly restarts. The trend towards more sophisticated fault-tolerance mechanisms is acknowledged by [52], which mentions current research on enabling "operator level" restart. We plan to quantify the notion of faults and develop a cost model to evaluate different techniques for making progress in their presence. Integrating the fault model with the query processor's cost model will allow the query planner to selectively place materialization and/or distribution operations with the goal of intelligently minimizing the effects of faults. Some very preliminary results that illustrate the basic tradeoffs can be found in [12].

7.5 Text data and queries

ASTERIX aims to admit arbitrary mixes of structured and unstructured (text) data whose consuming applications are not pre-defined. To support this flexibility, we must provide developers with a rich query language that provides a seamless combination of structural query and text-search primitives. We envision an expressive text-search extension of AQL with arbitrary boolean combinations of full-text predicates [13], i.e., predicates involving the position of certain keywords in the text (e.g., "return SIG mission statements mentioning keywords *data* and *knowledge within a window of at most 10 keywords* and not mentioning *artificial intelligence*"). Since top-K result pruning is a crucial requirement at the targeted scale of ASTERIX storage, a key contribution will be to allow application developers to specify their own relevance ranking/scoring functions. We are developing a generic framework that supports a wide class of scoring algorithms, including algorithms seen in the literature as well as user-defined scoring. On the data side, we also need to validate ADM's suitability (e.g., as opposed to XML) for sufficiently representing the kinds of data that such applications will wish to handle.

One of the most important challenges that this goal raises is due to the limitations of existing algebraic full-text techniques proposed in the database community. Full-text algebras benefit from database-style dynamic optimizers, and are suitable for integrated DB-Full-Text systems such as XQuery Full Text [5], but so far they have been used only with optimizers that do not consider scoring. Top-K pruning and other optimizations require scoring early in plans. Optimizers must preserve the semantics of early scoring when applying other optimizations. If scoring semantics are not preserved, scoring can become *inconsistent*. Given the effort usually put by application owners into ranking search results and the competitive advantage such rankings often contain, inconsistent scoring is not only a technical challenge but also a serious business issue.

7.6 Standing queries and pub/sub

We plan to support query-based triggers and subscriptions against large volumes of ASTERIX data. Triggering events can include data modifications (such as updates, inserts, or deletes) or temporal events. They might also be composite, i.e., a combination or sequence of single events. While extensive work exists on relational triggers [19] and their scalability [33], few efforts have addressed triggers in parallel or distributed systems, or their support against semistructured data, which is our focus here. We also plan to provide support for query subscriptions in a Publish-Subscribe environment. In the XML messaging community, many approaches have been presented for providing efficient filtering of XML data against large sets of continuous queries within a centralized server [25] and more recently using distributed XML dissemination [3]. However, distributed XML systems typically suffer from load balancing or from the overhead incurred from updating large indexes. In ASTERIX, we plan to explore efficient techniques for incrementally evaluating trigger/subscription conditions against highly partitioned and semistructured data, including conditions that span many partitions and/or involve non-co-located data. One of the challenges here will be sorting out our transaction and consistency models and exploring their impact on trigger and subscription semantics. In addition, of course, massive data is likely to be accompanied by large numbers of users, so any of the chosen techniques will have to work well in the face of many active triggers and subscriptions.

7.7 End-to-end data connections

An interesting question for a scalable data management platform such as ASTERIX is how one can most effectively "feed" such a system. E.g., what are the channels and interfaces through which data arrives, and how is data converted from an unstructured form (from the Web) into a queryable, semistructured form? On the output side, how are large batch query results managed, and how are the results of standing queries (subscriptions) disseminated to a large user base? These questions are currently beyond the scope of our effort, but we are starting to partner with another group at UCI and with a group at the San Diego Supercomputing Center at UCSD to begin addressing them based on requirements that those groups are bringing to the table.

8 Conclusions and status

In this paper we have described ASTERIX, a new data-intensive storage and computing platform initiative involving researchers at UC Irvine as well as UC San Diego and UC Riverside. We provided a top-down description of the effort beginning with its main goal—the storage and analysis of large data sets pertaining to evolving-world models. We outlined the platform's requirements and the associated challenges and then discussed how the project is addressing them. We provided a technical overview including the software architecture, the current version of the user model for data and queries, and the approach to scalable query processing and data management being implemented for this user model. ASTERIX utilizes a new scalable runtime platform called Hyracks that we also described, both in terms of its roles as the ASTERIX query runtime and as a lower-level data-intensive computing platform in its own right (including its Hadoop compatibility mode). Finally, we described a handful of the key research challenges that we are currently tackling as well as some of the additional challenges that are still lying in wait.

The ASTERIX effort has officially been underway as a multi-campus NSF project since September of 2009 (though some work had begun 7–8 months earlier on a shoestring budget). The Hyracks runtime layer is now up and running on our 40-core/40-disk cluster at UC Irvine; its preliminary performance results versus Hadoop appear extremely encouraging, and we have made an early open source drop of Hyracks available on Google Code. With respect to the end-to-end ASTERIX system, it is now up and limping on our cluster as well. The basic AQL compiler, the logical algebra, and the runtime operators (physical operators) have been implemented over a simple initial B+ tree based storage and indexing model along with support for hash-partitioned data distribution and processing. We are aiming to have fairly solid initial implementations of ADM and AQL, supporting both managed and external datasets as well as much of the AQL design (sans versioning), running with respectable performance on our cluster sometime in the latter half of calendar year 2011. Of course, given that we have a substantial list of open research problems related to ASTERIX and systems in its class, we expect that we will be building and refining various components of ASTERIX for several more years (at a minimum).

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.

References

1. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web: From Relations to Semistructured Data and XML. Morgan Kaufmann, San Mateo (1999)
2. Abiteboul, S., Fischer, P.C., Schek, H.-J.: Nested Relations and Complex Objects in Databases (LNCS). Springer, Berlin (1989)
3. Abiteboul, S., Manolescu, I., Polyzotis, N., Preda, N., Sun, C.: Xml processing in dht networks. In: ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering, pp. 606–615. IEEE Computer Society, Washington (2008)
4. Agrawal, R., et al.: The Claremont report on database research. Commun. ACM **52**(6), 56–65 (2009)
5. Amer-Yahia, S., Botev, C., Buxton, S., Case, P., Doerre, J., Dyck, M., Holstege, M., Melton, J., Rys, M., Shanmugasundaram, J.: XQuery and XPath full text 1.0. W3C Candidate Recommendation, July 9 (2009)
6. Apache Avro, http://hadoop.apache.org/avro/
7. Apache Hadoop, http://hadoop.apache.org
8. Ballinger, C.: Born to be parallel. Why parallel origins give teradata. Database an enduring performance edge. http://www.teradata.com/library/pdf/eb3053.pdf
9. Battré, D., Ewen, S., Hueske, F., Kao, O., Markl, V., Warneke, D.: Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In: SoCC '10: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 119–130. ACM, New York (2010)
10. Behm, A., Ji, S., Li, C., Lu, J.: Space-constrained gram-based indexing for efficient approximate string search. In: ICDE (2009)
11. Behm, A., Li, C., Carey, M.: Answering approximate string queries on large data sets using external memory. Technical report, Department of Computer Science, UC Irvine (under submission) (July 2010)
12. Borkar, V., Carey, M., Grover, R., Onose, N., Vernica, R.: Hyracks: a flexible and extensible foundation for data-intensive computing. In: ICDE (2011)
13. Botev, C., Amer-Yahia, S., Shanmugasundaram, J.: Expressiveness and performance of full-text search languages. In: EDBT, pp. 349–367 (2006)
14. Carey, M.J., Muhanna, W.A.: The performance of multiversion concurrency control algorithms. ACM Trans. Comput. Syst. **4**(4), 338–378 (1986)
15. Chaiken, R., Jenkins, B., Larson, P.-Å., Ramsey, B., Shakib, D., Weaver, S., Zhou, J.: SCOPE: easy and efficient parallel processing of massive data sets. PVLDB **1**(2), 1265–1276 (2008)
16. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R.R., Bradshaw, R., Weizenbaum, N.: Flumejava: easy, efficient data-parallel pipelines. In: PLDI, pp. 363–375 (2010)
17. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. 26(2) (2008)
18. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., Yerneni, R.: Pnuts: Yahoo!'s hosted data serving platform. PVLDB **1**(2), 1277–1288 (2008)

19. Dayal, U., Blaustein, B., Buchmann, A., Chakravarthy, U., Hsu, M., Ledin, R., McCarthy, D., Rosenthal, A., Sarin, S., Carey, M.J., Livny, M., Jauhari, R.: The HiPAC project: combining active databases and timing constraints. SIGMOD Rec. **17**(1), 51–70 (1988)
20. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
21. Dean, J., Ghemawat, S.: MapReduce: a flexible data processing tool. Commun. ACM **53**(1), 72–77 (2010)
22. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: SOSP, pp. 205–220 (2007)
23. DeWitt, D.J., Ghandeharizadeh, S., Schneider, D.A., Bricker, A., Hsiao, H.-I., Rasmussen, R.: The Gamma database machine project. IEEE Trans. Knowl. Data Eng. **2**(1), 44–62 (1990)
24. DeWitt, D.J., Gray, J.: Parallel database systems: the future of high performance database systems. Commun. ACM **35**(6), 85–98 (1992)
25. Diao, Y., Altinel, M., Franklin, M.J., Zhang, H., Fischer, P.M.: Path sharing and predicate evaluation for high-performance xml filtering. ACM Trans. Database Syst. **28**(4), 467–516 (2003)
26. Facebook press room—statistics. http://www.facebook.com/press/info.php?statistics
27. Facebook Thrift. http://incubator.apache.org/thrift
28. Garofalakis, M.N., Ioannidis, Y.E.: Parallel query scheduling and optimization with time- and space-shared resources. In: VLDB, pp. 296–305 (1997)
29. Ghemawat, S., Gobioff, H., Leung, S.-T.: The Google file system. In: SOSP, pp. 29–43 (2003)
30. Goldman, R., Widom, J.: Dataguides: enabling query formulation and optimization in semistructured databases. In: VLDB, pp. 436–445 (1997)
31. Google protocol buffers. http://code.google.com/apis/protocolbuffers/
32. Graefe, G.: Query evaluation techniques for large databases. ACM Comput. Surv. **25**(2), 73–170 (1993)
33. Hanson, E.N., Carnes, C., Huang, L., Konyala, M., Noronha, L., Parthasarathy, S., Park, J.B., Vernon, A.: Scalable trigger processing. In: ICDE, pp. 266–275 (1999)
34. Helland, P.: Life beyond distributed transactions: an apostate's opinion. In: CIDR, pp. 132–141 (2007)
35. Hong, W., Stonebraker, M.: Optimization of parallel query execution plans in XPRS. In: PDIS, pp. 218–225 (1991)
36. Hyracks project on Google code. http://code.google.com/p/hyracks
37. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: EuroSys, pp. 59–72 (2007)
38. Jaql, http://www.jaql.org
39. Jaql 0.1. http://www.jaql.org/release/0.1/jaql-overview.html
40. JSON. http://www.json.org/
41. Li, C., Lu, J., Lu, Y.: Efficient merging and filtering algorithms for approximate string searches. In: ICDE (2008)
42. MarketWatch, The Wall Street Journal. Will the news survive? http://www.marketwatch.com/story/will-the-news-survive-2009-12-08
43. Melnik, S., Gubarev, A., Long, J.J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T.: Dremel: interactive analysis of web-scale datasets. PVLDB **3**(1), 330–339 (2010)
44. Moerkotte, G.: Building query compilers. Manuscript, 2009
45. Object database management systems. http://www.odbms.org/odmg/
46. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: a not-so-foreign language for data processing. In: SIGMOD Conference, pp. 1099–1110 (2008)
47. Pew Internet & American Life Project. Twitter and status updating, Fall 2009. http://www.pewinternet.org/Reports/2009/17-Twitter-and-Status-Updating-Fall-2009.aspx
48. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data: parallel analysis with Sawzall. Sci. Program. **13**(4), 277–298 (2005)
49. Quass, D., Widom, J., Goldman, R., Haas, K., Luo, Q., McHugh, J., Nestorov, S., Rajaraman, A., Rivero, H., Abiteboul, S., Ullman, J.D., Wiener, J.L.: Lore: a lightweight object repository for semistructured data. In: SIGMOD Conference, p. 549 (1996)
50. Ramakrishnan, R., Gehrke, J.: Database Management Systems. WCB/McGraw-Hill, Boston (2002)
51. Snodgrass, R.T., Ahn, I.: A taxonomy of time in databases. In: SIGMOD Conference, pp. 236–246 (1985)
52. Stonebraker, M., et al.: MapReduce and parallel DBMSs: friends or foes? Commun. ACM **53**(1), 64–71 (2010)

53. The Radicati Group Inc. Business user survey, 2009. http://www.radicati.com/wp/wp-content/uploads/2009/11/Business-User-Survey-2009-Executive-Summary1.pdf
54. Thusoo, A.: Hive—a petabyte scale data warehouse using Hadoop. http://www.facebook.com/note.php?note_id=89508453919
55. Twitter blog. Measuring tweets, Feb. 2010. http://blog.twitter.com/2010/02/measuring-tweets.html
56. U.S. Department of Commerce, Washington: Quarterly retail e-commerce sales, 4th quarter 2008. http://www2.census.gov/retail/releases/historical/ecomm/08Q4.html
57. Vernica, R., Carey, M., Li, C.: Efficient parallel set-similarity joins using MapReduce. In: SIGMOD Conference (2010)
58. Vernica, R., Li, C.: Efficient top-k algorithms for fuzzy search in string collections. In: KEYS, pp. 9–14 (2009)
59. Wong, E., Youssefi, K.: Decomposition—a strategy for query processing (abstract). In: Author, J.B.R. Jr. (ed.) Proceedings of the 1976 ACM SIGMOD International Conference on Management of Data, Washington, DC, June 2–4, 1976, p. 155. ACM, New York (1976)
60. Xiao, C., Wang, W., Lin, X.: Ed-join: an efficient algorithm for similarity joins with edit distance constraints. In: VLDB (2008)
61. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/
62. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P.K., Currey, J.: DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In: OSDI, pp. 1–14 (2008)