Breaking BAD: A Data Serving Vision for Big Active Data

Michael J. Carey
Univ. of California, Irvine
mjcarey@ics.uci.edu

Steven Jacobs
Univ. of California, Riverside
sjaco002@ucr.edu

Vassilis J. Tsotras
Univ. of California, Riverside
tsotras@cs.ucr.edu

ABSTRACT

Virtually all of today's Big Data systems are *passive* in nature. Here we describe a project to shift Big Data platforms from passive to *active*. We detail a vision for a scalable system that can continuously and reliably capture Big Data to enable timely and automatic delivery of new information to a large pool of interested users as well as supporting analyses of historical information. We are currently building a Big Active Data (BAD) system by extending an existing scalable open-source BDMS (AsterixDB) in this active direction. This first paper zooms in on the Data Serving piece of the BAD puzzle, including its key concepts and user model.

CCS Concepts

•**Information systems** → *Database management system engines;*

Keywords

Big Data, Active Data, Parallel Databases, Pub/Sub.

1. INTRODUCTION

We are past the time where "Big Data" simply refers to many computers holding data. Huge amounts of data are generated daily by social, mobile, and Web platforms and applications. In the emerging age of the Internet of Things (IoT), data is being collected, held, and used by an increasingly diverse set of devices. It is thus critical to find ways to ingest, analyze, and deliver information and insights from this massively growing sea of data to millions of users in real time. It is time to enter the era of Big Active Data.

Some active software platforms, such as Publish/Subscribe (Pub/Sub) systems and Streaming Query Systems, exist today. However, these systems fail to satisfy what we view as key requirements for Big Active Data management because of significant limits on the capabilities of the data and/or queries. Pub/Sub and Streaming Query systems, for example, scale by limiting their queries to examining one of (or a window of) the incoming records in isolation rather than the overall collection of data. In our view, such limitations lead to existing systems falling short in the following ways:

1. Incoming data items might not be important in isolation, but in their **relationships** to other items in the data as a whole. Subscriptions should consider **data in context**, not just newly arrived items' content.
2. Important information for users is likely to be absent in incoming data items, yet exist elsewhere in the data as a whole. Results delivered to users should be able to be **enriched** with other existing data in order to provide **actionable notifications**.
3. In addition to "in the moment" processing, it can be important to perform later queries and analyses over the collected data as a whole. Thus, retrospective **Big Data analytics** must also be well-supported.

We argue here for Big Active Data (BAD) as the future of Big Data systems. A BAD system should leverage the benefits of state-of-the-art Big Data Management Systems (BDMSs), including their scalability, declarative query language, flexible data model, and support for parallel data analytics. It should also leverage ideas from current Pub/Sub and streaming query systems. We envision a world where data is published by a large number of sources, and where it should be retained for later analyses. We envision millions of data subscribers who desire notifications that involve not just the incoming data, but the current state of other static or infrequently-changing data. Our goal is to enable the accumulation and monitoring of petabytes of data of potential interest to millions of end users; when "interesting" new data appears, actionable notifications should be delivered to the users in time frames measured in (100's of) milliseconds.

2. BAD SYSTEM OVERVIEW

Figure 1 provides our high level vision for how a scalable BAD platform should look. Outside of the platform are the data sources (Data Publishers) and end users (Data Subscribers). Within the system itself, its components provide two broad areas of functionality – Big Data monitoring and management (handled by the BAD Data Cluster) and notification management and distribution (handled by the BAD Broker Network). Before we further elaborate on the roles or details of these cooperating components, let us quickly consider the nature of a "typical" BAD use case.

2.1 Use Cases

Many application domains have "active needs" that could benefit from a BAD software platform rather than hav-

DEBS '16, June 20-24, 2016, Irvine, CA, USA

DOI: http://dx.doi.org/10.1145/2933267.2933313

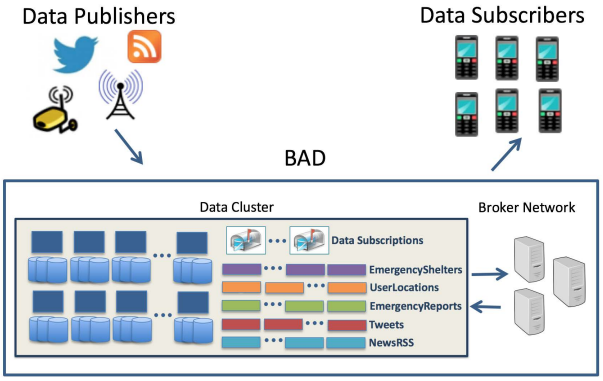

Figure 1: Big Active Data – System Overview.

ing to awkwardly glue together multiple existing systems. These include emergency management, homeland security, public health monitoring, product marketing and management, and national mood monitoring in a Presidential election year, to name but a few of the possibilities. As an example consider the USGS ShakeCast service [3], which was created to collect sensor data about earthquakes in real-time from thousands of sources and to deliver "ShakeMap" reports to subscribing agencies and facilities. A BAD platform could facilitate the creation of a richer, future, ShakeCast++.

In addition to sensor data, ShakeCast++ could use geotagged social data. As it arrives, its earthquake emergency relevance could be evaluated not only based on its direct content, but by combining it with other, more static, geographic and law enforcement data sets. ShakeCast++ could push enriched notifications, with information like nearby hospitals and shelters, in a timely manner to a large mobile user base. A data-oriented BAD subscription language would enable hundreds of non-government agencies (e.g., the Red Cross) and private entities (e.g., utility companies, hospitals, schools) to each register appropriate data subscription profiles. Notifications could be "personalized" by subscriber, using additional data sets, to enable each one to meet their individual response needs based on the information in their notifications. Rapid delivery of such information could be crucial to neutralizing hazards as well as of ensuring safety for first responders and the populous within the vicinity.

2.2 Requirements and Approach

Returning to Figure 1, the BAD platform must have extensible support for a variety of high-volume **data feeds** so that data from many different Data Publishers can be continuously and reliably ingested and stored for use by BAD applications. Given a set of registered data interests from the universe of Data Subscribers, the BAD platform must capture and monitor their potentially many interests against the evolving state of the stored data – in an efficient, shared, and scalable manner – in order to detect the need to generate and send out new notifications. To that end, the platform can provide a notion of information **channels**, based on application-defined parameterized queries, that can be subscribed to by users. In our BAD architectural vision, as shown in Figure 1, these are the data-serving responsibilities of the subsystem referred to as the BAD Data Cluster.

Figure 1 shows a second major subsystem, the BAD Broker Network. When new notifications are generated by the BAD Data Cluster, they must be disseminated to their target users. Users will be geographically distributed and may connect to a given BAD application via a variety of mobile and Web-based devices. The data distribution responsibilities of the BAD platform are handled by a network of cooperating BAD **brokers** which manage the system's connections and communications with end users and their devices.

3. RELATED WORK

Our model for Big Active Data builds on knowledge from several areas, including modern Big Data platforms, early active database systems, and more recent active platform work on both Pub/Sub systems and Streaming Query systems. Figure 2 summarizes how we see our BAD vision as fitting into the overall active systems platform space.

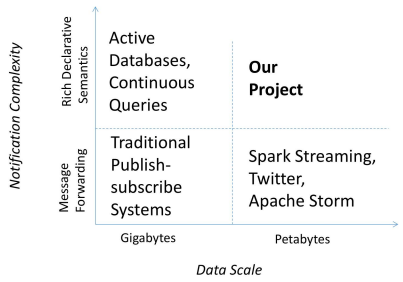

Figure 2: BAD in the context of other systems.

3.1 Big Data

First-generation Big Data projects resulted in MapReduce-based frameworks and languages, many based on Hadoop for long-running data analytics; key-value storage management systems (e.g., [14]) for simple but high-performance record management; and various specialized systems tailored to tasks such as scalable graph analysis or data stream analytics (e.g., [4, 9, 15, 17]). With the exception of data streams, these developments have been "passive" in nature. Recent projects such as Apache Flink [2], Spark [28], and AsterixDB [1] have moved from MapReduce to algebraic runtime systems but they are essentially all still passive systems.

3.2 Active Data

The HiPac Project [13] pioneered an approach (ECA rules) that is also seen in later systems, including TriggerMan [21], Ariel [20], Postgres [25], and Starburst [27]. Big Active Data is a descendant of ECA rules and Triggers, but they have two key problems. First, Triggers and ECA rules are really a "procedural sledgehammer" for a system: when event A happens, perform action B. We seek a more declarative (optimizable) way of making Big Data active and detecting complex events of interest. Second, to the best of our knowledge, no one has scaled an implementation of Triggers or ECA rules to the degree required for Big Data (in terms of the number of rules or the scaled-out nature of the data).

A Materialized View [6, 24] is a cached result of a given query that is made available for querying like a stored table. Materialized view implementations have been designed to scale on the order of the number of tables and have not addressed the level of scale that we expect for the number of data subscriptions in the BAD platform context.

3.3 Publish/Subscribe Systems

Pub/Sub Systems address a basic use case: data arrives in the form of publications, and publications are of interest to specific users. Pub/Sub systems seek to optimize the problems of identifying relevant publications and delivering them

to users in a scalable way. Modern Pub/Sub systems [16, 23, 29] provide a rich, content-based subscription language. Our BAD platform vision goes beyond this in two ways: First, whether or not newly arrived data is of interest to a user can be based on its relationship to other data. Second, the resulting notification(s) can include information based on other data. [26] studied the integration of Pub/Sub with Databases, but no scalability issues were addressed.

3.4 Continuous Query Engines

The seminal work on Continuous Queries was Tapestry [18], which defined Continuous Queries over append-only databases, including a definition of monotonic queries. Most of the subsequent work has focused on streaming data (e.g., STREAM [8], Borealis [5], Aurora [4], TelegraphCQ [11], and PIPES [22]). These systems build specialized data flows to process queries as new data is streamed through the system. Not designed to work with permanent storage, their queries relate to individual records or to windows of records.

3.4.1 NiagaraCQ and Spatial Alarms

NiagaraCQ [12] turned queries into data by finding groups of queries that do selections on the same attribute but differ by the constant(s) of interest (e.g., age=19 vs. age=25). Given these groups, they create a dataset of the constants and join it with incoming data to produce results for multiple users via a single join. This **data-centric** approach of treating continuous queries as data has inspired our own subscription scaling work. Spatial Alarms [10] also used this idea. Spatial Alarms issue alerts to users based on objects that meet spatial predicates. The spatial predicates are stored as objects in an R-Tree, and incoming updates are spatially joined with this R-Tree of standing queries.

4. WHAT'S BAD FOR USERS?

4.1 Classes of Users

We envision three different kinds of BAD users.

As indicated earlier in Figure 1, the first category of users are **data publishers** who provide data in the form of streams of incoming records. These streams enter the Data Cluster via *data feeds*. In a typical use case, the data publishers will be known/trusted sources of data (e.g., news sites, social media data streams, or government agencies) and the incoming data will be broadly relevant to a given BAD application (e.g. emergency reports and weather broadcasts).

The second category of BAD users are **application managers**. They know about the data stored in the Data Cluster for their applications. Based on the expected user interests, an application manager will create and manage parameterized *channels* that can be subscribed to in their application.

The third category is **data subscribers**, who are aware of the channels created for a BAD application and subscribe to one or more of them. A *subscription* can be created for a specific channel and indicates the parameter values of interest. Each subscription will be registered in the BAD Data Cluster with a subscription id. After its creation, a subscription will begin to receive new results from the channel.

4.2 Data Cluster vs. Broker Responsibilities

To support many data subscriptions and subscribers, the BAD architecture has a distributed *Broker Network* as the scalable interface between subscribers and the Data Cluster as well as distributing notification results produced by the Data Cluster to the geographically distributed subscribers.

A data consumer, the end user of a given BAD application, will see a user-friendly list of the available channels on his/her device (cell phone, tablet, etc.) and may subscribe, for example, to the EmergencyWatch channel. This channel's parameters might be the type of emergency (tornado, fire, ...) and the location (city) of interest. Using the application, the data consumer might specify interest in tornadoes in Iowa City. The application would then communicate via the Broker network to create the corresponding subscription at the Data Cluster. Given the presence of the Broker network, the data consumers are abstracted away from the Data Cluster. Rather than managing details about end users, the Data Cluster just needs to issue subscription ids and keep track of the Broker responsible for receiving and forwarding the notifications for each subscription. Thus, data consumers are users from the Broker Network's perspective but not from the Data Cluster's perspective. The Data Cluster's "users" are just the Brokers themselves.

Our BAD architecture separates result creation (which occurs within the Data Cluster) from result delivery (which is the responsibility of the Brokers). With this separation, data consumers can be allocated to different brokers as they move around, and Brokers can make notification delivery decisions based on factors such as data consumer availability (online/offline), location, battery consumption, or current device. (The Broker network could also make decisions about whether to aggregate data subscriptions and/or notifications for groups of users, etc.) In the remainder of this paper we will focus primarily on the features required of an active Data Cluster and its interactions with the data publishers, the application managers, and the Brokers.

5. A BAD USER EXPERIENCE

Consider a scenario where users (data subscribers) are interested in getting information about emergency reports near some user-specified location. Emergency reports are created continuously and include both temporal and spatial attributes. Data publishers in this case could include News Broadcasters, Weather Forecasters, Government Agencies, and Police Departments. In addition to emergency reports, subscribers would like the system to include other useful information related to the emergency (e.g., the location of the nearest shelter for an emergency) in its notifications. To this end, the application maintains additional data sets (which are relatively static) containing locations and other information about Emergency Shelters.

5.1 Leveraging AsterixDB

Instead of starting from scratch, we are using an existing BDMS and adding features to make it active. We selected Apache AsterixDB [1, 7] as a foundation for BAD because it is openly available, intended for others to use for research, and has the following technical benefits:

1. A rich, declarative query language (AQL)
2. A scalable distributed dataflow platform (runtime)
3. Rich data types (including semi-structured data)
4. A fast data ingestion mechanism (data feeds)

Figure 3 gives AsterixDB DDL for two data sets for our example application: EmergencyReports and EmergencyShel-

ters. It shows AsterixDB's rich data types, including support for *nested records* (EmergencyReport.reportingStation), *unordered lists* (EmergencyShelter.amenities) and *optional fields* (EmergencyReport.affectedCities). This is important for the kinds of BAD applications that we envision. Optional fields enable support for data where record contents can vary (e.g., some reports may not have all information or may have extra); nesting permits embedding subrecords within notifications to support enhanced notifications.

```
create type ReportingStation as
 {id:uuid, name:string?, location:point}

create type EmergencyReport as {
 id:uuid, emergencyType:string, impactZone:circle,
 state:string, affectedCities:{{string}}?,
 message:string, severityLevel:int, timestamp:datetime,
 expirationTime:datetime, reportingStation:ReportingStation
}

create type EmergencyShelter as
 {id:uuid, name:string, location:point, amenities:{{string}}}

create dataset EmergencyShelters(EmergencyShelter)
 primary key id;

create dataset EmergencyReports(EmergencyReport)
 primary key id;
```

Figure 3: DDL to create datasets.

5.2 Data Feeds

To deliver Emergency Reports for rapid ingestion, a BAD data producer can create a data feed adapter to stream incoming reports directly into AsterixDB, and it can be used to feed the EmergencyReports dataset. The DDL to create a data feed using AsterixDB's data feed support would be:

```
create EmergencyFeed using EmergencyFeedAdapter;
connect feed EmergencyFeed to dataset EmergencyReports;
```

The create statement defines the Emergency Report feed, and the connect statement starts the continuous flow of data into the target data set. For more on data feeds, see [19].

5.3 Channels

Based on the expected interests of their users, as mentioned earlier, a BAD application manager will create a set of channels – continuous queries with parameters – that the users can subscribe to. Since we are building upon AsterixDB, we can leverage another of its features: AQL *user-defined functions* (UDFs). UDFs allow AQL queries with parameters to be defined and named for later reuse, so we can utilize them to specify the queries for BAD channels.

As an example of an AQL UDF, consider the following English query: "Select the message and impact zone for tornadoes occurring within the last week within my state." We can encapsulate such a query in AsterixDB as the function tornadoesInStateLastWeek in Figure 4. This function takes an argument to indicate the state where the caller resides. Its query body uses a temporal clause to select emergency reports from within the last seven days, filters them by state and type "tornado", and returns the requested fields. (Note that we could also have made the emergency type a parameter rather than specializing the function for tornadoes.)

AQL UDFs provide a nice basis for our BAD channel DDL. Channel definitions can be based on functions, and subscriptions can be specified by identifying a channel and the parameter value(s) of interest. A subscription represents

```
create function tornadoesInStateLastWeek($state) {
  from $report in dataset EmergencyReports
  let $historyStart :=
    current-datetime() - day-time-duration("P7D")
  where $report.timestamp >= $historyStart
  and $report.emergencyType = "tornado"
  and $report.state = $state
  select {
    "message":$report.message,
    "impactZone":$report.impactZone
  }
};
```

Figure 4: DDL to Create an AQL function.

a parameterized version of a function that will continue to execute over time; a channel query can be compiled and optimized once and shared by all subscriptions.

Based on use cases we see two useful types of channels: *Repetitive* and *Continuous*. A **Repetitive Channel** is like a "data chron job": it executes the channel function periodically (e.g., every five minutes) starting at its time of creation. The resulting notifications contain the *full* result of the function at the time of each execution. In contrast, a **Continuous Channel** can be thought of as executing whenever any of its underlying data sets change; it checks whether the changes contribute new results and, if so, it issues notifications containing only the *differential* results.

5.3.1 Repetitive Channels

To elaborate on the semantics of repetitive channels, assume that a user is interested in the following query: "Every week, give me the list of all tornadoes that occurred in my state that week." The DDL for this channel appears in Figure 5(a) and uses the function tornadoesInStateLastWeek. The DDL provides a name for the new channel and indicates the function upon which the channel is based (i.e., the function tornadoesInStateLastWeek with one parameter[1]). The channel definition DDL also specifies how often the channel will run ("P7D" stands for seven days). Once created, this channel will continue to produce new results every seven days for all of the created subscriptions. In general a repetitive channel's function may be based on any AQL query.

```
create repetitive channel tornadoesInState
using tornadoesInStateLastWeek@1 period duration("P7D");
```

(a)

```
subscribe to tornadoesInState("IA") on Broker1;
subscribe to tornadoesInState("KS") on Broker2;
subscribe to tornadoesInState("WI") on Broker1;
```

(b)

Figure 5: (a) Repetitive Channel; (b) Subscriptions.

Recall that a user will connect through a broker to create subscriptions. Thus, a subscription request includes the name of the responsible broker, the channel name and the parameter values. Figure 5 (b) shows DDL requests for three subscriptions from user(s) interested in tornadoes in Iowa, Kansas, and Wisconsin; two originated through Broker1 and one through Broker2. Running each subscription DDL will return the ID for the newly created subscription, which is unique per channel and serves as the key by which the Broker can reference the subscription in the future.

[1]AsterixDB permits function name overloading: the unique name for a function is its given name plus a parameter count indicator.

5.3.2 *Continuous Channels*

Suppose that a user is interested in the query: "Whenever I'm in the impact zone of some emergency, notify me with the message for the emergency, its impact zone, and all emergency shelters that are within that impact zone." This user is interested in emergencies as they happen, so we use a continuous channel. This channel analyzes changes as they occur and delivers new results as they are produced. In addition, users here are essentially also data publishers, as they will update their locations as they move around. Location tracking is achieved by creating a new data set and a data feed for User Locations, as shown in Figure 6. User devices can continuously update their locations by having the application send them to the User Locations data feed. With this data feed in place, UserLocations becomes yet another data set that can be referred to in a channel function's query.

```
create type UserLocation as
{id:uuid, userId:int, location:point, timestamp:datetime}

create dataset UserLocations(UserLocation) primary key id;

create feed UserLocationsFeed using UserLocationsFeedAdapter
connect feed UserLocationsFeed to dataset UserLocations;
```

Figure 6: BAD DDL for User Location Tracking.

The function EmergenciesNearUser in Figure 7 takes a single parameter (the id of the user who subscribes) and performs a spatial join between three datasets. In addition to joining Emergency Reports with User Locations – using the userid parameter to find the user's current location, which is checked for inclusion in the emergency's impact area – this channel enhances the notification results by giving the subscribing user a list of nearby shelters.

```
create function EmergenciesNearUser($userid) {
  from $emergency in dataset EmergencyReports
  from $userlocation in dataset UserLocations
  where $userlocation.user-id = $userid
    and spatial-intersect(
      $emergency.impactZone,$userlocation.location)
    and $userlocation.timestamp >= $emergency.timestamp
    and $userlocation.timestamp <= $emergency.expirationTime
  select {
    "message":$emergency.message,
    "impactZone":$emergency.impactZone,
    "shelter locations":
      from $shelter in dataset EmergencyShelters
      where spatial-intersect
        ($emergency.impactZone,$shelter.location)
      select $shelter.location
}};

create continuous channel EmergenciesNearUserChannel
using EmergenciesNearUser@1;

subscribe to EmergenciesNearUserChannel("12345") on Broker3;
```

Figure 7: Continuous Channel.

Our continuous channel semantics follow Tapestry's [18] in several ways. We focus on *append-only* datasets, where new tuples can be added but existing tuples are not deleted or modified (or else corresponding notifications are not needed). An example is UserLocations, which is essentially a timestamped log of users' locations. Moreover, we focus on *monotonic* queries where previous query results cannot be invalidated by considering new tuples. Given that three datasets are joined in our example, there are three distinct ways that new results could be produced for a given user:

1. The user enters the impact zone of an active emergency (new location added to UserLocations dataset).
2. A new emergency arises with the user's location in its impact zone (new report added to EmergencyReports).
3. A triage center is set up in the impact zone of an active emergency (new shelter in EmergencyShelters).

This illustrates the BAD advantage of subscribing to data. Pub/Sub, in contrast, is based on filtering individual events (i.e., the arrival of a new tuple in a stream) and cannot detect these sorts of events (which involve data joins) – at least not without gluing together additional systems.

5.4 Brokers

As explained earlier, brokers provide a go-between layer between the BAD Data Cluster and the actual, geographically distributed, end users of a given BAD application.

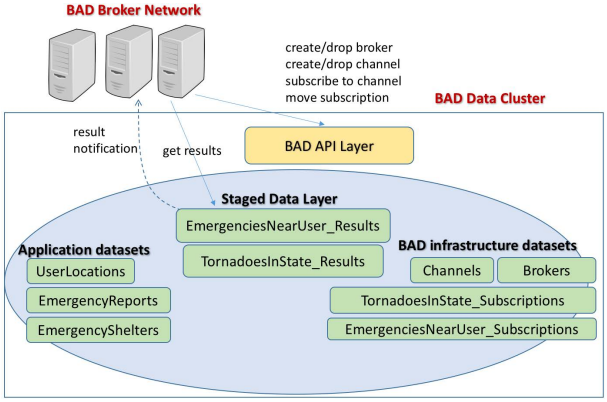

Figure 8: The BAD Data Cluster

Figure 8 provides an overview of the interactions between the BAD Data Cluster and the brokers as well as the datasets involved in our emergency-related BAD example application. In addition to the three application datasets, execution of the create channel and create broker statements add entries about new channels and new brokers to the system's Channels and Brokers datasets, respectively. Whenever a new channel is created, two other datasets are also created internally to support the individual channel, namely: Channel_Subscriptions and Channel_Results (shown for two channels in the Staged Data Layer). The schema for each channel's Channel_Subscriptions dataset includes the parameter values for that subscription as well as the subscriptionId and the name of the current hosting Broker for the subscription. The schema for each channel's Channel_Results dataset contains the subscriptionId, the deliveryTime, and the individual result value for each subscription to the channel.

Brokers use the channel and subscription DDL to communicate with the BAD Data Cluster's API. When results are produced for a given subscription, a notification will be sent to the broker hosting that subscription. It can then retrieve the results from the channel's results dataset when it wishes.

5.5 Data, Channel, and Subscription Data

To clarify all of these concepts, let us look at how they are represented in terms of datasets in the BAD Data Cluster.

Figure 9 shows our example application's datasets' contents at a point in time. Figure 10 shows the subscriptions to the repetitive channel tornadoesInState and results from datetime ("2015-11-25 15:10:00"). There were three tornadoes in the last week, two in Kansas and one in Iowa. The Kansas subscription yielded two results (subscriptionId: "...-

EmergencyReports

timestamp	emergencyType	state	message	ImpactZone	...
09:00	tornado	KS	Please proceed to the nearest shelter	circle("300,20 12.0")	...
09:02	tornado	IA	Please proceed to the nearest shelter	circle("100,5 10.0")	...
09:04	tornado	KS	Please proceed to the nearest shelter	circle("300,10 5.0")	...
09:05	earthquake	IA	Shelters will provide water to affected victims	circle("105,15 50.6")	...
...	...	...	...	...	...

EmergencyShelters

shelterName	location
Downtown Evacuation Center	point("100,10")
Public Shelter 152	point("100,20")
Public Shelter 148	point("100,100")

UserLocations

timestamp	userId	location
2015-11-25 09:08:00	1	point("101,12")
2015-11-25 09:09:00	2	point("105,22")
2015-11-25 09:15:00	3	point("113,115")

Figure 9: Application Datasets

Subscriptions

subscriptionId	brokerName	parameter0
...6eac6c0bb1aa	Broker1	"IA"
...ec778db67af6	Broker2	"KS"
...d080a1a76b07	Broker1	"WI"
...3ee5d9c893de	Broker3	"IA"

Results

subscriptionId	deliveryTime	result
...6eac6c0bb1aa	15:10	{"message":" Please proceed to the nearest shelter", "impactZone":circle("100,5 10.0")}
...3ee5d9c893de	15:10	{"message":" Please proceed to the nearest shelter", "impactZone":circle("100,5 10.0")}
...ec778db67af6	15:10	{"message":" Please proceed to the nearest shelter", "impactZone":circle("300,20 12.0")}
...ec778db67af6	15:10	{"message":" Please proceed to the nearest shelter", "impactZone":circle("300,10 5.0 ")}

Figure 10: Repetitive Channel Datasets

Subscriptions

subscriptionId	brokerName	parameter0
...7dc98721a977	Broker1	1
...781f06522ffa	Broker2	2

Results

subscriptionId	deliveryTime	result
...7dc98721a977	09:02	{"message":Please proceed to the nearest shelter", "impactZone":circle("100,5 10.0"),"shelter locations":[point("100,10")]}
...7dc98721a977	09:05	{"message":"Shelters will provide water to affected victims"," impactZone": circle("105,15 50.6"),"shelter locations": [point("100,10"),point("100,20")]}
...781f06522ffa	09:05	{"message":"Shelters will provide water to affected victims"," impactZone":circle("105,15 50.6"),"shelter locations": [point("100,10"),point("100,20")]}

Figure 11: Continuous Channel Datasets

ec778db67af6"), one per Kansas tornado. The Iowa tornado led to one result for each of the two Iowa subscriptions.

Figure 11 shows the subscriptions and results for the EmergenciesNearUserChannel. Two emergencies have matched existing subscriptions. An Iowa tornado (at datetime("2015-11-25 09:02:00")) was near the location of user 1 (subscriptionId "...7dc98721a977"), yielding the first result. An earthquake (at datetime("2015-11-25 09:05:00")) was near both users and led to a result record for each subscription.

6. CONCLUSIONS

We have called for a shift from passive Big Data to an era of Big Active Data and shared a vision for addressing the challenges of a BAD world. We described how we are building a BAD platform by adding active capabilities to Apache AsterixDB. We presented language extensions to support brokers, channels, and subscriptions in the BAD Data Cluster component of the platform and the Broker Network layer. Our initial version of the BAD Data Cluster – "BAD Asterix" – has support for creating a broker, creating/subscribing/unsubscribing/retrieving results from a repetitive channel, and dropping brokers and channels.

Acknowledgments: This work was partially supported by NSF grants: IIS-1447826 and IIS-1447720.

7. REFERENCES

[1] Apache AsterixDB. `https://asterixdb.apache.org`.

[2] Apache Flink. `https://flink.apache.org`.

[3] U.S. geological survey – Shakecast, 2014. `http://earthquake.usgs.gov/research/software/shakecast/`.

[4] D. J. Abadi et al. Aurora: a new model and architecture for data stream management. *VLDB J.*, 2003.

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

[6] P. Agrawal et al. Asynchronous view maintenance for VLSD databases. In *ACM SIGMOD*, 2009.

[7] S. Alsubaiee et al. AsterixDB: A scalable, open source BDMS. *Proc. VLDB Endowment*, 2014.

[8] A. Arasu et al. Stream: The Stanford stream data manager. *IEEE Data Eng. Bull.*, 2003.

[9] S. Babu and J. Widom. Continuous queries over data streams. *ACM SIGMOD Record*, 30(3), 2001.

[10] B. Bamba, L. Liu, P. S. Yu, G. Zhang, and M. Doo. Scalable processing of spatial alarms. In *HiPC*. 2008.

[11] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[12] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for internet databases. In *ACM SIGMOD*, 2000.

[13] U. Dayal et al. The HiPAC project: Combining active databases and timing constraints. *ACM SIGMOD Record*, 17(1), 1988.

[14] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *ACM SOSP*, 2007.

[15] N. Dindar et al. DejaVu: Declarative pattern matching over live and archived streams of events. In *ACM SIGMOD*, 2009.

[16] P. T. Eugster et al. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2), 2003.

[17] B. Gedik et al. Spade: the System S declarative stream processing engine. In *ACM SIGMOD*, 2008.

[18] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using collaborative filtering to weave an information Tapestry. *Comm. of the ACM*, 35(12), 1992.

[19] R. Grover and M. J. Carey. Data ingestion in AsterixDB. In *EDBT Conf.*, 2015.

[20] E. N. Hanson. The design and implementation of the Ariel active database rule system. *IEEE Trans. Knowl. Data Eng.*, 8(1), 1996.

[21] E. N. Hanson et al. Scalable trigger processing. In *IEEE ICDE*, 1999.

[22] J. Krämer and B. Seeger. Pipes: a public infrastructure for processing and exploring streams. In *ACM SIGMOD*, 2004.

[23] T. Milo, T. Zur, and E. Verbin. Boosting topic-based Publish-Subscribe systems with dynamic clustering. In *ACM SIGMOD*, 2007.

[24] D. Quass and J. Widom. On-line warehouse view maintenance. *ACM SIGMOD Record*, 26(2), 1997.

[25] M. Stonebraker and L. A. Rowe. The design of POSTGRES. In *ACM SIGMOD*, 1986.

[26] L. Vargas, J. Bacon, and K. Moody. Event-driven database information sharing. In *BNCOD*, 2008.

[27] J. Widom, R. Cochrane, and B. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *VLDB*, 1991.

[28] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[29] Y. Zhao, K. Kim, and N. Venkatasubramanian. DYNATOPS: A dynamic topic-based publish/subscribe architecture. In *DEBS*, 2013.