A BAD Demonstration: Towards Big Active Data

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

Md Yusuf Sarwar Uddin
University of California, Irvine
msarwaru@uci.edu

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

Vagelis Hristidis
Univ. of California, Riverside
vagelis@cs.ucr.edu

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

N. Venkatasubramanian
University of California, Irvine
nalini@ics.uci.edu

ABSTRACT

Nearly all of today's Big Data systems are *passive* in nature. We demonstrate our Big *Active* Data ("BAD") system, a scalable system that continuously and reliably captures Big Data and facilitates the timely and automatic delivery of new information to a large population of interested users as well as supporting analyses of historical information. We built our BAD project by extending an existing scalable, open-source BDMS (AsterixDB [1]) in this active direction. In this demonstration, we allow our audience to participate in an emergency notification application built on top of our BAD platform, and highlight its capabilities.

1. INTRODUCTION

While some active software platforms, such as publish/subscribe systems [12] and streaming query systems [15] exist today, each fails to satisfy one or more key requirements for Big Active Data management due to limits in their data and query facilities. These key requirements are:

1. Incoming data items might not be important in isolation, but rather in their **relationships** to other data items as a whole. Subscriptions thus need to consider **data in context**, and not just newly arriving items' local content.

2. Information important to users may be absent in incoming items, but may exist elsewhere in the data as a whole. Subscription results should be **enrichable** using other relevant, related data to provide users with **actionable notifications**.

3. In addition to on-the-fly processing, later queries and analyses over the collected data may yield important insights. Thus, **retrospective Big Data analytics** must also be supported.

The rest of this paper is organized as follows: Section 2 discusses related work while Section 3 overviews the BAD system. Section 4 details the user experience for the audience of our demo, and Section 6 highlights the Impact and Significance of the demo.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.

Proceedings of the VLDB Endowment, Vol. 10, No. 12

Copyright 2017 VLDB Endowment 2150-8097/17/08.

2. 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 1 summarizes how our BAD vision fits into the overall active systems platform space.

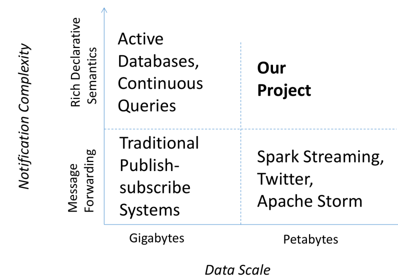

Figure 1: BAD in the context of other systems.

2.1 Big Data

First-generation Big Data projects resulted in MapReduce-based frameworks, many based on Hadoop for long-running data analytics; key-value storage management systems [11] for simple but high-performance record management; and various specialized systems (e.g. scalable graph analysis or data stream analytics [3, 6]). With the exception of data streams, Big Data remains "passive" in nature. Recent projects such as Apache Flink [2], Spark [18], and AsterixDB [1] have moved from MapReduce to algebraic runtime systems but they are essentially all still passive systems.

2.2 Active Data

The HiPac Project [10] pioneered ECA rules, also seen in later systems [14]. Big Active Data is a descendant of ECA rules and Triggers, but overcomes two key limitations. First, Triggers and ECA rules are really a "procedural sledgehammer" for a system: when event A happens, perform action B. We provide a more declarative (optimizable) way of 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 [4] 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.

2.3 Publish/Subscribe Systems

Pub/Sub systems seek to optimize the problems of identifying relevant data publications and delivering them to users in a scalable way. Modern Pub/Sub systems [12, 19] provide a rich, content-based subscription language. Our BAD platform vision goes beyond this in two ways, as mentioned in Section 1: First, whether or not newly arrived data is of interest to a user can be based on its relationship to other data. Second, notification(s) can be enriched by other data. [17] studied Pub/Sub and Database integration, but no scalability issues were addressed.

2.4 Continuous Query Engines

The seminal work on Continuous Queries was Tapestry [13], which focused on append-only databases and included the idea of monotonic queries. Subsequent work has mostly focused on streaming data [5, 3]. These systems build specialized data flows to process queries as non-persistent data streams through the system; queries relate to individual records or windows of records.

2.4.1 NiagaraCQ and Spatial Alarms

NiagaraCQ [9] 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 [7] 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.

3. A BAD SYSTEM

Our Big Active Data platform is designed to deliver data of interest to a scalable number of users without compromising on any of the three requirements from Section 1. Figure 2 shows the BAD system at a high level. Outside the BAD platform are data sources (Data Publishers) and end users (Data Subscribers). Within the platform itself, its components provide two broad areas of functionality – Big Data management and monitoring, handled by the BAD Data Cluster, and notification management and distribution, handled by the BAD Broker Network.

3.1 BAD Data Cluster

We have chosen Apache AsterixDB [1] as a foundation for BAD because it is openly available, intended for others to use for research, and has technical benefits including a rich declarative language (AQL) and a scalable distributed dataflow runtime system with continuous data ingestion support. We have enhanced AsterixDB with a new feature called Channels [8]. Channels are created as parameter-instantiable versions of queries that will execute continuously starting at their creation. As an example, consider the following continuous query: *"Select the message, impact zone, and nearby shelters for emergencies occurring near me."* We can create this query channel as a continuous geo-spatial join between users and emergencies, enriching the query result with emergency shelter information.

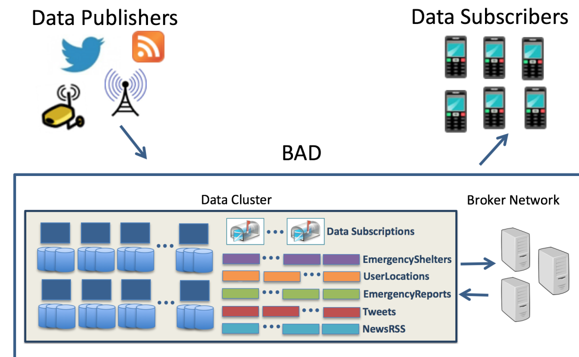

Figure 2: Big Active Data (BAD) System Overview

A user interested in this channel would subscribe using his or her user id. Internally, query evaluation takes a scalable data-centric approach by adapting techniques from prior active data systems as discussed in Section 2.4.1 [9, 7] to a more expressive and more capable query engine [8] to scale to a large number of subscriptions. As the channel executes, it will produce and stage individualized results for each active subscription. The Data Cluster will notify the BAD Broker network whenever new data is available.

We extend the RESTful API of AsterixDB to allow communication with Application Administrators (who create and manage channels) and BAD Brokers (the link between the BAD Data Cluster and the end user subscribers). The platform currently supports the following REST calls:

- `createbroker`: Register a broker as both a subscription generator on behalf of subscribers and a delivery endpoint for new data notifications.
- `createchannel`: Create a new channel, given a reference to the parameterized query (AsterixDB Function) to use.
- `subscribe`: Subscribe to a parameterized channel. (Sent by the broker on behalf of a subscriber, including the parameters for this subscription and the broker to handle its notifications.)
- `getresults`: Used by the broker to run queries against the staged results on the cluster, including searching by their time and subscription id.
- `unsubscribe`: Remove a subscription from a channel.
- `movesubscription`: Designate a new broker as the endpoint for notifications for a given subscription.
- `dropbroker`: Remove the broker as a subscription creator and notification endpoint.
- `dropchannel`: Stop execution of the channel.

3.2 BAD Broker Network

The BAD broker network consists of two components: the Broker Coordination Service and BAD broker nodes.

3.2.1 Broker Coordination Service (BCS)

The BAD broker network is managed by a BCS. When a new broker node joins the broker network, it registers with the BCS server. After registration, the broker can accept clients for possible subscriptions to channels. A client connects to the BCS server to receive the address of the broker to which it should connect for subsequent services. The selection of a broker can depend on many aspects, such as geographic locations and the current system load (the number of clients each broker is serving). Currently, we use

geo-distributed brokers where clients are assigned to their nearest brokers (with the IP to location mapping obtained from the MaxMind database).

3.2.2 BAD Brokers

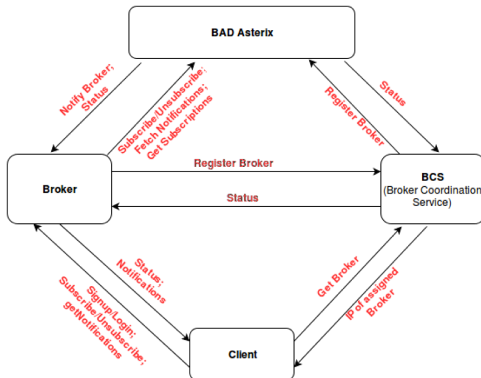

Figure 3: Broker Data Flow

The brokers are responsible for handling client (called BAD client) registration, managing subscriptions and delivering results for those subscriptions. Each broker has two parts: a "client-facing" part managing the clients and an "Asterix-facing" part handling interactions with the Asterix backend. A simple workflow of interaction between a client and the broker is as follows: The client registers via the broker and logs in. The client then subscribes to one or more available channels using desired parameter values, which are passed to the Asterix backend by the broker. The backend notifies the broker when new results are populated in the subscribed channels. The broker, in turn, notifies the client, and the client acts to fetch the results as desired.

Broker nodes are implemented as RESTful servers written in Python using the Tornado web framework. The server currently supports the following REST calls:

- `registerapplication`: Register a new application in the system (in a designated AsterixDB dataverse).
- `register`: Register/sign up a new user.
- `login`: Login an already signed up user.
- `subscribe`: Subscribe to a parameterized channel.
- `getresults`: Retrieve results for a given subscription.
- `listchannels`: List channels in the current dataverse.
- `listsubcriptions`: List subscriptions for a given user.
- `unsubscribe`: Unsubscribe from a subscription.
- `logout`: Logout from the current session.

In terms of sending out notifications to BAD clients, the broker node currently supports three types of clients: (1) *Web clients*: Notifications pushed through web sockets. (2) *Desktop clients*: Notifications are managed by RabbitMQ, an active messaging system. (3) *Android clients*: Notifications pushed through FCM (Firebase Cloud Messaging).

4. A BAD DAY IN MUNICH

In this demonstration, we mimic an "emergency notification system" where users can receive information about emergencies (for example, earthquakes, floods, shootings, etc.) via their subscriptions. Emergency reports (containing useful information related to emergency situations, and including temporal and spatial attributes) may be published by agencies. Notifications to users can be enhanced [16] with additional data such as nearby shelters and their locations.

We will utilize a web application built using HTML, CSS3, Javascript, and the Angular framework. Initially, the application administrator communicates directly with the BAD Data Cluster to set up the application dataverse with relevant datasets and channels. The demo application will consist of pre-built parameterized channels that enable users to monitor emergencies in several ways, detailed below.

Our interactive demo enables the audience members to have the option of being either a data subscriber or a data publisher, by using their own personal computers or phones.

4.1 Data Subscriptions

When accessing Emergency Reports website, the user will have the option to select a home city. Once logged in, the user will automatically begin moving randomly around the city. (To see/update her current location, she can go to the Data Notifications screen (Figure 6), discussed later).

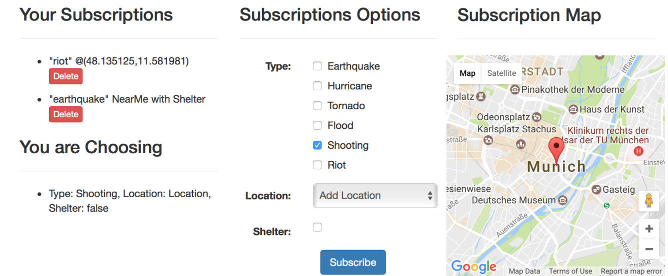

Figure 4: Data Subscription Interface

Figure 4 shows the User Subscriptions screen. Here she can perform two tasks:

- View/Remove existing subscriptions
- Create new subscriptions

To begin with, the new user will not have any subscriptions. She can create as many subscriptions as she would like by selecting several options for each subscription:

- The emergency type(s) she is interested in (e.g., riots)
- Whether to monitor emergencies at a static location (e.g. Munich) or to monitor emergencies occurring near her dynamically updated location ("near me")
- Whether to enrich the result with shelter information

Once she has made these choices, she confirms her subscription, which gets added to "Your subscriptions." She can create additional subscriptions or delete existing ones.

4.2 Data Publishers: Loki strikes

For more mischievous audience members, we allow them to log into Loki accounts. Rather than subscribing to emergencies of interest, Loki users are data publishers who can create emergencies when the urge strikes (Figure 5).

To create an emergency, Loki has the following choices:

- Which type of emergency to create
- Where the emergency will occur (either type the location or specify it by moving a Loki icon on the map)

Loki can continue creating emergencies over time, in ever-changing locations.

4.3 Data Notifications

On the Data Notifications screen (Figure 6), a user can see the live notifications created when the emergencies created by Loki users intersect with her subscriptions in time and space. On this screen, she can choose to manually move

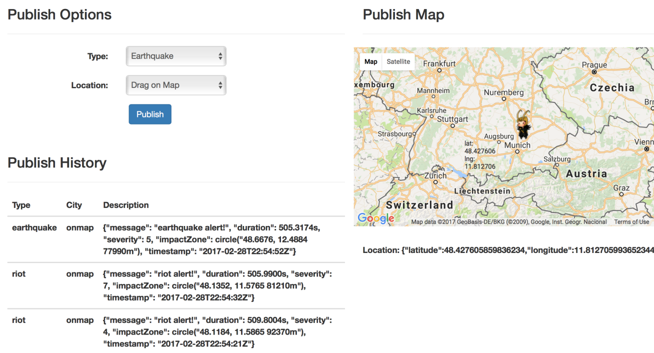

Figure 5: Loki interface

herself to any location on the map. The application will then begin a random walk from the current location to slowly move the user around.

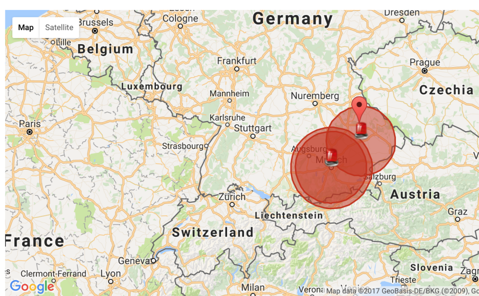

Figure 6: Data Notifications Screen

When an emergency intersects with one of the user's subscriptions, a red circle indicating the emergency impact zone will appear on the map. She can click on these alerts to see the full notification, including the shelter information (if part of the subscription). Users also have the option of a list view, which appends new results to a list rather than showing them on a map. On the History Screen (Figure 7), users can see the full history of results for their subscriptions.

Type	Severity	Mesaage	Coordinates	Radius	Duration	TimeStampv
EARTHQUAKE	5	earthquake alert!	48.667612,12.488365	77990m	505.317350s	02/28/2017 at 2:54:52PM
RIOT	7	riot alert!	48.135163,11.576500	81210m	505.990000s	02/28/2017 at 2:54:32PM
RIOT	4	riot alert!	48.118434,11.586456	92370m	509.800400s	02/28/2017 at 2:54:21PM

Figure 7: History Screen

5. BAD IN SOCIAL MEDIA

We will also demonstrate a Twitter-based mobile application which allows a user to subscribe to her Twitter friends (i.e. followers and/or followees), and if the user, a subset of her friends and a coffee shop are in close proximity, the application will notify the user that they can meet in that coffee shop. We get the latest location of all the users and their friends in real-time when they post geo-tagged tweets. Coffee shop locations are stored in an AsterixDB table. The BAD subscription channel checks for this condition periodically (e.g., every 5 seconds) and notifies the broker once a meetup is possible. The developed Android app shows the locations of the users and of the coffee shop on a map.

6. IMPACT AND SIGNIFICANCE

Users of this system are able to see how we achieve the three key requirements in Section 1:

1. Emergencies are not important in and of themselves but because of their relationships to user location data (another rapidly changing dataset).
2. Notifications to the user are enriched with information from a third dataset (the emergency shelters).
3. The History Screen enables the application user to do a posthumous query to determine a full history of subscription and emergency intersections.

Our project is advocating a shift from passive Big Data to an era of Big Active Data platforms. In this demonstration we present our implementation of such a system and show how its features can be used to build an interesting active application involving Big Data. Our vision and the initial design decisions for the BAD platform appear in [8].

7. ADDITIONAL AUTHORS

Yao Wu (UCI, Renmin U China (NSFC No. 61532021), `yao.wu@uci.edu`), Syed Safir (UCI, `fsyedsha@uci.edu`), Purvi Kaul (UCI, `kaulp@uci.edu`), Xikui Wang (UCI, `xikuiw@uci.edu`), Mohiuddin Abdul Qader (UCR, `mabdu002@cs.ucr.edu`) and Yawei Li (UCR, `yli119@ucr.edu`)

8. REFERENCES

[1] Apache AsterixDB (https://asterixdb.apache.org/).
[2] Apache Flink (https://flink.apache.org).
[3] D. J. Abadi et al. Aurora: a new model and architecture for data stream management. *VLDB J.*, 2003.
[4] P. Agrawal et al. Asynchronous view maintenance for VLSD databases. 2009.
[5] A. Arasu et al. Stream: The Stanford stream data manager. *IEEE Data Eng. Bull.*, 2003.
[6] S. Babu and J. Widom. Continuous queries over data streams. *ACM SIGMOD*, 2001.
[7] B. Bamba, L. Liu, P. S. Yu, G. Zhang, and M. Doo. Scalable processing of spatial alarms. *HiPC*, 2008.
[8] M. Carey, S. Jacobs, and V. Tsotras. Breaking BAD: A data serving vision for big active data. *DEBS*, 2016.
[9] J. Chen et al. NiagaraCQ: a scalable continuous query system for internet databases. *ACM SIGMOD*, 2000.
[10] U. Dayal et al. The HiPAC project: Combining active databases and timing constraints. *SIGMOD*, 1988.
[11] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. *ACM SOSP*, 2007.
[12] P. T. Eugster et al. The many faces of publish/ subscribe. *ACM Comput. Surveys*, 2003.
[13] D. Goldberg et al. Using collaborative filtering to weave an information Tapestry. *Comm. of ACM*, 1992.
[14] E. N. Hanson et al. Scalable trigger processing. *IEEE ICDE*, 1999.
[15] C.-Q. JI et al. Analysis and management of streaming data: A survey. *Journal of software*, 2004.
[16] M. Y. S. Uddin et al. RichNote: Adaptive selection and delivery of rich media notifications to mobile users. *Distributed Computing Systems (ICDCS)*, 2016.
[17] L. Vargas, J. Bacon, and K. Moody. Event-driven database information sharing. *BNCOD*, 2008.
[18] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI*, 2012.
[19] Y. Zhao et al. DYNATOPS: A dynamic topic-based publish/ subscribe architecture. *DEBS*, 2013.