Building An End-To-End BAD Application

Shahrzad Haji Amin Shirazi
University of California, Riverside
shaji013@ucr.edu

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

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

ABSTRACT

Traditional big data infrastructures are *passive* in nature, passively answering user requests to process and return data. In many applications however, users not only need to analyze data, but also to subscribe to and actively receive data of interest, based on their subscriptions. Their interest may include the incoming data's content as well as its relationships to other data. Moreover, data delivered to subscribers may need to be enriched with additional relevant and actionable information. To address this Big *Active* Data (BAD) challenge we have advocated the need for building scalable BAD systems that continuously and reliably capture big data while enabling timely and automatic delivery of relevant and possibly enriched information to a large pool of subscribers. In this demo we showcase how to build an end-to-end active application using a BAD system and a standard email broker for data delivery. This includes enabling users to register their interests with the bad system, ingesting and monitoring data, and producing customized results and delivering them to the appropriate subscribers. Through this example we demonstrate that even complex active data applications can be created easily and scale to many users, considerably limiting the effort of application developers, if a BAD approach is taken.

CCS CONCEPTS

• **Information systems → Data management systems**.

KEYWORDS

big active data, active dataset,end to end active application

ACM Reference Format:
Shahrzad Haji Amin Shirazi, Michael J. Carey, and Vassilis J. Tsotras. 2021. Building An End-To-End BAD Application. In *The 15th ACM International Conference on Distributed and Event-based Systems (DEBS '21), June 28-July 2, 2021, Virtual Event, Italy*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3465480.3467840

1 INTRODUCTION

With the ever-increasing amounts of data being generated daily by social, mobile, and web applications, as well as the prevalence of the Internet of Things, it is critical to shift from passive to 'active' Big Data, to enable efficient data monitoring and timely delivery of personalized information to subscribers based on their subscriptions (indicated interests). In particular, a Big Active Data platform should address various BAD *desiderata* [5]; namely: (i) Incoming data items might not be important in isolation, but due to their **relationships** to other items in the data as a whole. That is, subscriptions need to consider **data in context**, not just a newly arrived item's content. (ii) Important information for users may be missing in the incoming data items, yet it may exist elsewhere in the data as a whole. Hence, the results delivered to subscribers must be able to be **enriched** with other existing data in order to provide **actionable notifications** that are individualized per user. (iii) 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. To this end, we have recently built a BAD platform supporting the above desiderata [8, 9, 13], using Apache AsterixDB [1] as its data serving foundation.

While BAD has similarities to continuous queries, triggers, streaming engines and pub/sub systems, these technologies do not offer all BAD desiderata or may not scale. In order to fully support BAD applications without BAD system, one would have to glue multiple existing systems together. This attempt would result in added management complexity, limited functionality, and integration difficulty [14]. These limitations and the time needed to glue different components together would further increase the difficulty of developing BAD applications. In contrast, in this demo we exhibit the easiness of creating active applications through BAD. We create as an example the My_Vaccine app which enables users to learn about their turn for getting the vaccine and nearby vaccine stations with vaccine availability based on their personal info.

Interactions between the BAD system and its end-users are possible through a broker network which delivers the data to the actual users. Depending on the application, a broker system with advanced caching and load-balancing strategies can be used to deliver data to subscribers. In this demo, we use for simplicity an email-based broker [11]. EMBA (*Em*ail *B*roker for *BAD*) is responsible for handling clients, subscriber registration, managing subscriptions, and delivering results to subscribers through emails. We proceed with Section 2 that discusses the limitations of related work. Section 3 introduces the My_Vaccine example application and Section 4 offers a quick overview of the main parts of the BAD system. EMBA and its layers are described in Section 5 while Section 7 presents conclusions.

2 RELATED WORK

Having users register their requests as persistent queries and subsequently being notified whenever new results become available is similar in context to work on Continuous Queries [6, 12]; nevertheless, big data poses new challenges for classic continuous query approaches due to their complexity and computational cost. Similarly, Triggers from traditional databases offer users the capability to react to events in a database under certain conditions [15]; however, triggers do not scale as the volume of data/users grows.

DEBS '21, June 28-July 2, 2021, Virtual Event, Italy

ACM ISBN 978-1-4503-8555-8/21/06.
https://doi.org/10.1145/3465480.3467840

Recently, Streaming Engines have been widely used in many active-data-related use cases [4, 10, 16]. Data is ingested and optionally processed in streaming engines on-the-fly and then pushed to other systems for later analysis. Streaming engines can be used for creating data processing and data customizing pipelines, but due to the nature of data streams, only a limited set of processing operations are available. As a result, streaming engines would need to be coupled with other systems for meeting the complete BAD challenge at scale. This would introduce additional performance overhead and integration complexity for users [14].

Delivering data of interest to many users also resonates with the publish/subscribe communication paradigm [7]; here, subscribers register their interests in incoming data items and will subsequently be notified about data published by publishers. However, typical pub/sub systems only forward data items from publishers to subscribers without offering the capability to process it. Also, each data item is treated in isolation, so users' interests are limited to the data item itself (its topic, type, or content), but not its relationship to other data. In addition, pub/sub systems have to be integrated with other Big Data engines for supporting analytical queries[14].

3 MY_VACCINE APPLICATION

In this demo we will mimic building an application for Covid vaccinations (referred below as the My_Vaccine app). We consider two groups of users, the vaccine station managers (or content providers) who publish available vaccine information about their station and patients (the end users) who want to be notified about vaccine availability. For versatility purposes we let the two user groups interact with the app in different ways (interface vs EMBA).

When appointments become available in a vaccine station, its station manager will use the My_Vaccine application interface, created using the Django framework [2], to publish the station's information (including station name, address, number of vaccine appointments available and the eligible group for those vaccines). We assume that the station manager updates a station's information which changes happen using the same interface.

In our example, there are two available *notification services* that patients can subscribe to. The first allows patients to be notified when new appointments are available within a distance (patient specified, say 10 kilometers) from a given static location (also patient specified, say some landmark or the patient's parents' house). Patients that subscribe to that service will also be notified if information changes (e.g. number of appointments) at an existing station within the required distance. The second service allows patients to be notified about appointments in currently nearby vaccine stations (based on the patient's last known location) where they can be vaccinated based on group eligibility (patient specified, say based on a given age range or profession). As patients move around (and update their locations) the notifications they receive may contain different stations. Patients interact with the My_Vaccine app through emails to and from EMBA. Using emails they can learn about available notification services (in our example two), register their interests on these services (by specifying the notification service they are interested in and their parameters), update their location and, get back notifications.

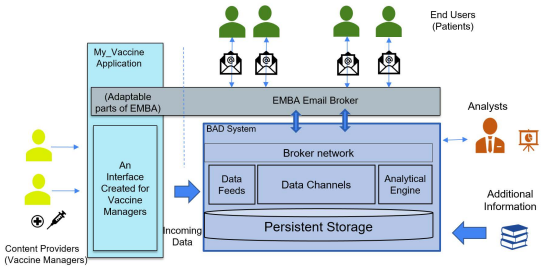

Figure 1: An overview of an application implemented using BAD and EMBA system

Figure 1 shows an overview of the users, the app, and its interactions with BAD. BAD has five basic blocks: (i) Data Feeds, (ii) Data Channels, (iii) Persistent Storage, (iv) Analytical Engine, and (v) Broker Network. We use Data feeds to capture the incoming data created by the station managers (when new appointments appear or are updated) and the patients' personal information (when patients register in the system or when their location changes). Each Feed is connected to a dataset so that the ingested data can be persisted in storage. Data Channels provide a scalable mechanism that allows users to subscribe to data they are interested in. Data Channels translate user subscriptions into parameterized queries and efficiently retrieve the appropriate data on behalf of the subscribers. The Analytical Engine supports declarative queries and enables analysts to reveal useful information from incoming data and its relationships with other stored data. Subscribers communicate with BAD through brokers connected with the Broker Network. Any data that needs to be sent to the subscribers of channel is sent through a broker which is registered as an HTTP or socket endpoint in the BAD system. In our demo we use EMBA as the broker; in general, one can use other brokers to deliver data in different ways, say by text, etc.

Using the BAD system and EMBA, application development is greatly simplified. To create the My_Vaccine app, a developer would have to: (1) create an online user interface for the station managers, (2) create the application datasets, the feeds and channels, and (3) adapt the mail broker (EMBA) to the application's channels. The creation of the appropriate datasets, feeds and data channels for the My_Vaccine app is described in Section 4 while the description of EMBA's layers appears in Section 5.

4 CREATING BAD FEEDS AND CHANNELS

For the My_Vaccine app, we defined an open datatype `Station` to describe the required attributes about vaccination stations; such attributes include the station id, the station's name, address, telephone and GPS coordinates (the DDL appears in Figure 2 using SQL++ statements). We have created an active dataset `Stations` (with key sid) to persist such data. The available appointments in each station are described by the open datatype `AvailableAppt`, containing attributes like station id, number of appointments and the eligibility group. The active dataset `AvailableAppt` is created for persisting this data. Here we assume that a station can offer vaccines for different eligibility groups, thus the key is the combination of sid

and eligibility group. We also defined an open datatype `Patient` to describe the minimum required attributes for each Patient (including first and last name, patient id and location coordinates) and created an active dataset `Patients` to persist this data. Active datasets, different from normal datasets, enable continuous query semantics [12] in channels (discussed below).

```
CREATE TYPE Station AS OPEN {sid: bigint,
stationName: string,
stationAddress: string,
phoneNumber: string,
latitude: double,
longitude: double};
CREATE ACTIVE DATASET Stations(Station)PRIMARY KEY sid;

CREATE TYPE AvailableAppt AS OPEN {sid: bigint,
NumOfAppts: bigint,
eligibilityGroup: String};
CREATE ACTIVE DATASET AvailableAppts(AvailableAppt)
PRIMARY KEY sid,eligibilityGroup;

CREATE TYPE Patient as {patient_FirstName:string,
patient_LastName:string,
patientID:bigint,
latitude: double,
longitude: double};
CREATE ACTIVE DATASET Patients(Patient)PRIMARY KEY patientID;
```

Figure 2: DDL for Datasets

```
CREATE FEED AvailableApptsFeed WITH {
"adapter-name": "socket_adapter",
"sockets": "127.0.0.1:10001",
"address-type": "IP",
"type-name": "AvailableAppt",
"format": "JSON"};
CONNECT FEED AvailableApptsFeed TO DATASET AvailableAppts;
START FEED AvailableApptsFeed;
CREATE FEED PatientsFeed WITH {
"adapter-name": "socket_adapter",
"sockets": "127.0.0.1:10004",
"address-type": "IP",
"type-name": "Patient",
"format": "JSON"};
CONNECT FEED PatientsFeed TO DATASET Patients;
START FEED PatientsFeed;
```

Figure 3: DDL for Data Feeds

Data Feeds. In any application, data of interest may arrive rapidly. To capture such data, BAD provides *data feeds*, an ingestion facility to help continuously ingest data from various external data sources reliably and efficiently. In this example, we assume that datasets `AvailableAppts` and `Patients` are updated rapidly and thus we create feeds `AvailableApptsFeed` and `PatientsFeed` for them. The feed DDLs are depicted in Figure 3. Each feed populates the relevant dataset using a socket adapter and specifying the incoming data's format as JSON. In contrast we assume that the `Stations` dataset is updated infrequently and thus no feed is created.

Data Channels. To simplify creating applications using the BAD system, we extract the shared structure among subscriptions and offer it as a service, namely a *data channel*, for subscribers to subscribe to with parameters. When creating a channel, developers construct a channel query to describe the data of interest for subscribers and specify the channel's period to indicate how often [1] the channel query should be evaluated for subscribed users. Data channels can be created using declarative queries and are managed by the BAD system. All subscriptions of a channel are periodically evaluated together to allow the system to exploit shared computations.

```
CREATE CONTINUOUS PUSH CHANNEL
AllStationsNearAddress (latitude,longitude,distance)
PERIOD duration("PT10S"){
SELECT s.stationName, s.stationAddress,s.phoneNumber,
a.NumOfAppts,a.eligibileGroup
FROM Stations s,AvailableAppts a
WHERE a.sid=s.sid AND is_new(a) AND spatial_distance(
create_point(latitude,longitude),
create_point(s.latitude,s.longitude))<distance};

CREATE CONTINUOUS PUSH CHANNEL StationsForMe
(patientID,eligibility) PERIOD duration("PT10S"){
SELECT s.stationName, s.stationAddress,s.phoneNumber
,distanceFromPatient
FROM Stations s,AvailableAppts a,Patients p
LET distanceFromPatient=spatial_distance
(create_point(s.latitude,s.longitude),
create_point(p.latitude,p.longitude))
WHERE p.patientID=patientID AND is_new(a) AND
a.eligibilityGroup=eligibility AND a.sid=s.sid
ORDER BY distanceFromPatient LIMIT 5};
```

Figure 4: DDL for Channels

For the My_Vaccine application, we have created the channel `AllStationsNearAddress`, as shown in Figure 4, which allows patients to learn about available appointments in stations within a certain distance from a location. A patient can subscribe to this channel by providing three parameters: the latitude and longitude of the location of interest and the required distance. As an example, a user might only be interested in available appointments within 5 kilometers from her parents' house. The channel will run repetitively based on the period provided (in this case every 10 seconds– shown in the channel's PERIOD). After subscribing, the results delivered to her include the stations with available appointments, their eligibility group and number of available appointments, for all stations within the user-specified distance from the user-specified address; results are also *enriched* with existing data about these stations, including the station's name, telephone, and address. In general, this enrichment could also include other data, like directions to each station from the subscribed address, etc. The channel definition uses the **is_new** function to look for the new data which has not been sent to the user yet. The active dataset `AvailableAppts` provides the support for continuous query semantics to make sure every qualified new information about an appointment will be delivered to subscribed users. BAD provides two different channel modes for delivering data, Push and Pull. In the Push mode (as in our example), the complete new data of interest is sent to the subscriber, while in the Pull mode, the subscriber will only receive a notification from the channel that new data is available (and it is then up to the

[1] Channels are evaluated in a batched continuous manner, similar to the batches in Spark Streaming [16].

user whether he/she wants to get the data). For this application we have also created the channel `StationsForMe`, which allows users to get the name, address, phone number and distance to the five closest stations (to the patient's current location) with available appointments for the patient's eligibility. Users subscribe with their patientID and eligibility.

5 EMBA OVERVIEW

EMBA (BAD email broker) enables users to get info about available channels, register for channels, and get related notifications, by just sending and receiving emails. Figure 5 summarizes EMBA's layers.

The Receiver Layer. This layer is responsible for processing incoming email requests. EMBA deals with the following requests: (1) users ask about available channels and/or the characteristics of an existing channel, (2) users ask to register on available channels, and, (3) users update their location information. For collecting a users' registration information, EMBA provides two options: the standard approach is by email; there is also the possibility of collecting that information directly on Google Forms [3] and passing it to AsterixDB.

Interacting With BAD. This layer sends subscribe and unsubscribe requests to the BAD system as well as requests asking for information about a channel. As an example, when a patient (with id 222 and interest in eligibility "65 and older") requests by email to subscribe to the `StationsForMe` channel, this layer will send to the BAD system the statement in Figure 6. It is also responsible for receiving new notifications from channels and forwarding them to the Sender Layer. Finally, this layer will send patient location updates received via emails to the BAD feed.

The Sender Layer. The sender layer uses four different types of email to the users. (1) *Overview Email:* which summarizes the info an email should contain for the different services. For instance, when a user wants to ask about all available channels, it is enough to write the sentence "All channels" as the subject of the email. (2) *Available Channels:* EMBA uses this email to send the name and the description (describing the aim of the channel and the possible outputs) of all available channels. (3) *Subscription Information:* which contains the parameters needed for registering with a specific channel. As an example, if a user asks about the `StationsForMe` channel, he/she will receive an email with the two parameters (eligibility, patientID) needed for subscribing to this channel. (4) *Channel Results:* results from different channels will be emailed to the relevant users.

6 DEMO DESCRIPTION

Our demo enables the audience to witness the functionalities and efficiency of the BAD system through the My_Vaccine app. We will have imaginary patients who want to register with different channels and receive their info when a new suitable appointment pops up. The audience will be able to participate as vaccine station managers or as patients and receive notifications by email. The demo video appears at: https://youtu.be/T3V_vN9GuTY .

7 CONCLUSIONS

In this demo we showcase how to create an end-to-end application over Big Active data. We present how a developer can create such an application using the BAD system that we have developed and an email broker (EMBA) which enables users to register their interests, update data and receive notifications, all through emails. We created an email broker so as to showcase the easiness of completing such an end-to-end application (from data arriving into BAD to data received by the users). Other brokers (i.e. sending texts) could be used as well. Overall, creating a BAD application is quite easy; the developer needs only work on creating the appropriate services (through channels) and adapting the broker at hand. In return, the BAD system deals with data storage and manipulates, and creating notifications, while guaranteeing that the application can scale to large numbers of users (in the thousands) over terabytes of data.

Acknowledgements: This research was partially supported by NSF grants IIS-1838222, IIS-1838248, CNS-1924694 and CNS-1925610.

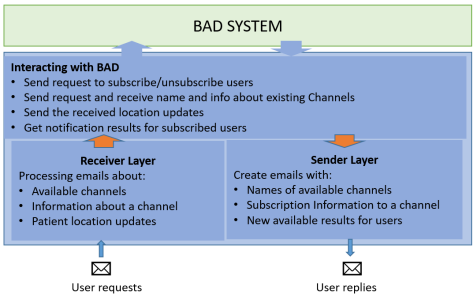

Figure 5: EMBA Layers

```
SUBSCRIBE TO StationsForMe(222,"65 and older")ON EmailBroker;
```

Figure 6: DDL for creating a channel subscription

REFERENCES

[1] Apache AsterixDB (https://asterixdb.apache.org/).
[2] Django (https://djangoproject.com).
[3] Google forms (https://www.google.com/forms).
[4] A. Alexandrov et al. The Stratosphere platform for big data analytics. *VLDB J.*, 23(6):939–964, 2014.
[5] M. J. Carey, S. Jacobs, and V. J. Tsotras. Breaking BAD: a data serving vision for big active data. In *Proc. of ACM DEBS*, pages 181–186, 2016.
[6] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. ACM SIGMOD*, pages 379–390, 2000.
[7] P. T. Eugster et al. The many faces of publish/ subscribe. *ACM Comput. Surveys*, 35(2):114–131, 2003.
[8] S. Jacobs, M. Y. S. Uddin, M. J. Carey, et al. A BAD demonstration: Towards Big Active Data. *PVLDB*, 10(12):1941–1944, 2017.
[9] S. Jacobs, X. Wang, M. J. Carey, V. J. Tsotras, and M. Y. S. Uddin. BAD to the Bone: Big Active Data at its Core. *VLDB J.*, 29(6):1337–1364, 2020.
[10] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proc. of NetDB*, pages 1–7, 2011.
[11] H. Nguyen, M. Y. S. Uddin, and N. Venkatasubramanian. Multistage adaptive load balancing for big active data publish subscribe systems. In *Proc. of ACM DEBS*, pages 43–54, 2019.
[12] D. B. Terry et al. Continuous queries over append-only databases. In *Proc. of ACM SIGMOD*, pages 321–330, 1992.
[13] X. Wang, M. Carey, and V. Tsotras. Bridging BAD islands: Declarative data sharing at scale. In *IEEE BigData - Workshop on Scalable Cloud Data Mngmt*, 2020.
[14] X. Wang, M. J. Carey, and V. J. Tsotras. Subscribing to big data at scale, CoRR abs/2009.04611, 2020.
[15] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.
[16] M. Zaharia et al. Discretized streams: fault-tolerant streaming computation at scale. In *Proc. of ACM SOSP*, pages 423–438, 2013.