UC Riverside
UC Riverside Previously Published Works

Title

Subscribing to big data at scale.

Permalink

Journal

Authors

Wang, Xikui

Carey, Michael

Tsotras, Vassilis

Publication Date

DOI

Subscribing to big data at scale

Xikui Wang[1] · **Michael J. Carey[1]** · **Vassilis J. Tsotras[2]**

Abstract

Today, data is being actively generated by a variety of devices, services, and applications. Such data is important not only for the information that it contains, but also for its relationships to other data and to interested users. Most existing Big Data systems focus on **passively** answering queries from users, rather than **actively** collecting data, processing it, and serving it to users. To satisfy both passive and active requests at scale, application developers need either to heavily customize an existing passive Big Data system or to glue one together with systems like *Streaming Engines* and *Pub-sub services*. Either choice requires significant effort and incurs additional overhead. In this paper, we present the BAD (Big Active Data) system as an end-to-end, out-of-the-box solution for this challenge. It is designed to preserve the merits of passive Big Data systems and introduces new features for actively serving Big Data to users at scale. We show the design and implementation of the BAD system, demonstrate how BAD facilitates providing both passive and active data services, investigate the BAD system's performance at scale, and illustrate the complexities that would result from instead providing BAD-like services with a "glued" system.

✉ Xikui Wang
  xikuiw@ics.uci.edu

  Michael J. Carey
  mjcarey@ics.uci.edu

  Vassilis J. Tsotras
  tsotras@cs.ucr.edu

[1] Donald Bren School of Information and Computer Sciences, University of California Irvine, Irvine, CA 92697, USA

[2] Computer Science and Engineering Department, University of California Riverside, Riverside, CA 92521, USA

1 Introduction

Big Data, without being analyzed, is merely a sequence of zeros and ones sitting on storage devices. To effectively utilize Big Data, researchers have developed a plethora of tools [1–4]. In many applications today, we want not only to understand Big Data, but also to deliver subsets of interest proactively to interested users. In short, users should not only be able to **analyze data** but also to **subscribe to data**. User subscription requests should not be limited to the incoming data's content but should also be able to consider its relationships to other data. Moreover, data to be sent should be allowed to include additional relevant and useful information. We refer to this as the *Big Active Data (BAD) challenge*. Due to the variety and volume of user requests, the data, and their relationships, analyzing, customizing, and delivering actionable data based on different user requests are not trivial tasks.

Traditionally, taking user requests and serving data continuously has been studied mostly in the context of Continuous Queries (e.g., Tapestry, NiagaraCQ) [5, 6]. Users there register their requests as persistent queries and are subsequently notified whenever new results become available. Although the continuous query concept overlaps significantly with the active data problem, Big Data poses new challenges for classic continuous query approaches due to their complexity and computational costs. Similarly, triggers from traditional databases offer users the capability to react to events in a database under certain conditions [7]. Users could try and take advantage of triggers to approach the active data challenge, but they soon become not applicable when the scale of the data, and thus the system, grows.

With the growth of streaming data and the need for real-time data analytics, Streaming Engines (e.g., Apache Kafka, Spark Structured Streaming) in recent years have been widely used in many active-data-related use cases [8–10]. Data is ingested and optionally processed in streaming engines on-the-fly and is 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 developers.

Delivering data of interest to many users also resonates with the publish/subscribe communication paradigm from the distributed systems community [11]. In the pub/sub paradigm, subscribers register their interests in incoming data items and will subsequently be notified about data published by publishers. Despite some similarity to the BAD challenge, pub/sub systems only forward data 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 must still be integrated with other Big Data systems (e.g., Data warehouses) in order to support historical analytical queries.

One significant goal of the BAD approach advocated here is that users should not only be able to **analyze data**—i.e., to issue queries and receive result

subsequently, but also to **subscribe to data**—i.e., to specify their interests in data and constantly receive the latest updates. Data of interest is not limited to **historical data**, but also includes **real-time data**, or **a combination of both**. Many (passive) systems today support data analytics, but very few of them provide the **active features** we need. In addition to that, we would like to allow users to subscribe to data without always having to write independent queries. Mastering query languages could be useful for data analysts with expertise, but it might be a burden for end-users interested only in receiving data. Although database features like stored procedures allow for the encapsulation of queries as executable units, they are still **passively** invoked by users. We need a system that allows users to analyze data declaratively and that enables users to subscribe to data actively with minimum effort.

In order to capture real-time data, we propose ***data feeds***, a data ingestion facility that allows users to reliably ingest fast incoming data at scale. In addition to that, users can also write declarative queries to process and enrich the incoming data on-the-fly, so it can later be used by other applications directly. The BAD system manages the data feeds, including ingestion protocol, data parsing, etc., for users without having them to write customized programs. In order to deliver the latest updates to end-users without asking them to construct queries and to "pull" data from the system constantly, we propose an abstraction— parameterized ***data channels***—to characterize user subscriptions. Users with expertise (e.g., application developers) can create data channels using declarative queries. Users with interest in data (e.g., end-users) can then subscribe to data channels with parameters and thus continuously receive new data. The BAD system runs data channels, manages their life-cycle, and offers them as active services.

Systems that handle real-time data today either process the incoming data on-the-fly (e.g., Streaming Engines) and then send it to other applications, or, persist the incoming data into storage and then continuously extract information from it (e.g., Continuous Queries). The BAD system provides both data feeds, which allow data processing on-the-fly, and data channels, which deliver information continuously, to cover the needs on both ends of the spectrum. Moreover, both the data feed and data channel abstractions effectively provide a declarative user model for activating Big Data without the need to manually glue multiple systems together and write customized programs.

Previously, we implemented an initial prototype of a BAD system— BAD-RQ— by enhancing Apache AsterixDB [12]. In BAD-RQ, we allow developers to create data channels using a declarative query language (SQL++) and enable users to subscribe to them by specifying their own parameters. Internally, channel queries are like parameterized prepared queries that are repetitively evaluated with subscription information and other relevant data. BAD-RQ computes them periodically on behalf of all users with all of the user-provided parameters and produces customized data for each subscribed user [13].

As BAD-RQ executes channel queries periodically, users may attempt to leverage them to approximate continuous query semantics— obtaining updates incrementally without retrieving the entire history or reporting redundant results [5]. For example, a continuous query *"send me **new** sensitive tweets"* can be loosely interpreted as a

repetitive channel query *"every 10 s, send me the sensitive tweets from the past 10 seconds"*. Although users can approximate continuous query semantics with repetitive channels, BAD-RQ does not guarantee continuous semantics, and data items could be missed or redundantly reported. To ensure continuous semantics, we want a systematic way of supporting continuous queries in BAD. We need (i) to make sure that users can receive incremental updates to data of interest with the guarantee of continuous semantics, (ii) to support different computational operations and indexes for accelerating evaluation, and (iii) to enhance the data channel model to provide a straightforward user model regarding continuous queries.

In this paper, we discuss Big Active Data in-depth, present the BAD system, and introduce BAD-CQ—a new BAD service that provides continuous query semantics. We show how BAD-CQ is designed and implemented, and we investigate its performance under different workloads at scale. This paper is organized as follows: We review work related to BAD in Sect. 2. In Sect. 3, we dive into the detailed vision of Big Active Data, discuss the settings of the BAD problem, and describe the building blocks of a BAD system. In Sect. 4, we present a repetitive BAD use case to demonstrate the BAD-RQ service and illustrate the BAD user model. We introduce continuous BAD in Sect. 5, discussing the limitations of approximating continuous BAD and presenting the design and implementation of the new BAD-CQ service. To compare a possible alternative approach with the BAD system, we introduce a GOOD (**G**luing **O**odles **O**f **D**ata platforms) system that consists of gluing together multiple Big Data systems in Sect. 6. We show how to use the GOOD system to provide BAD services and illustrate the challenges that developers would face in configuring, orchestrating, and managing such a glued system. We present a set of experimental results for the new BAD-CQ service and compare its performance with the glued system in Sect. 7. Section 8 concludes the paper.

2 Related work

Continuous queries are queries that are issued once and return results continuously as they become available. Tapestry [5] first introduced Continuous Queries over append-only databases, defined continuous semantics, and created rewriting rules for transforming user-provided queries into incremental queries. Much subsequent research has focused on queries over streaming data. STREAM is a research prototype for processing continuous queries over data streams and stored data [14]. It provides a Continuous Query Language (CQL) for constructing continuous queries against streams and updatable relations [15]. TelegraphCQ offers an adaptive continuous query engine that adjusts the processing during run-time and applies shared processing where possible [16]. NiagaraCQ splits continuous queries into smaller queries and groups queries with the same expression signature together. It stores signature constants in a table and utilizes joins to evaluate grouped queries together to improve scalability, and it uses delta files for incremental evaluation on changed data to improve computational efficiency [6]. Very few continuous query projects have been scaled out to a distributed environment. This limits their applicability in Big Data use cases.

Streaming engines allow low latency data processing and provide real-time analytics. Apache Storm is a distributed stream processing framework. It provides two primitives, "spouts" and "bolts", to help users create topologies for processing data in real-time [17]. Spark Structured Streaming is a stream processing engine built on top of Apache Spark. It divides incoming data into micro-batches of Resilient Distributed Datasets (RDDs) for fault-tolerant stream processing, and it offers a declarative API for users to specify streaming computations [10, 18]. Apache Kafka started as a distributed messaging system that allows collecting and delivering a high volume of log data with low latency. It later introduced a Streams API that enables users to create stream-processing applications [9, 19]. Apache Flink [20] (which originated from Stratosphere [8]) unifies both streaming and batch processing in one system and provides separate APIs (DataStream and DataSet) for creating programs running on a streaming dataflow engine [20]. Due to the nature of streaming data, streaming engines usually do not store data for the long-term. The incoming data is processed and then soon pushed to other systems for further processing or persistence.

Publish/subscribe services allow subscribers to register their interests in events and to be subsequently, asynchronously notified about events from publishers. There are three types of pub/sub schemes: topic-based, content-based, and type-based [11]. In topic-based pub/sub, publication messages are associated with topics, and subscribers register their interests to receive messages about topics of interest. Many systems in this domain focus on providing scalable and robust pub/sub services, including Scribe [21], SpiderCast [22], Magnet [23], and Poldercast [24]. Content-based pub/sub improves the expressiveness of pub/sub services by allowing subscriptions based on publications' content. Many research works in this area focus on improving the scalability and efficiency of matching publications to users' subscriptions, including XFilter [25], Siena [26], YFilter [27], BoXFilter [28], and BlueDove [29]. Type-based pub/sub groups publications based on their structure. It aims at integrating pub/sub services with (object-oriented) programming languages to improve performance [30]. P2P-DIET utilizes super-peers to serve data via continuous queries at scale [31]. While all these pub/sub services enable publishing data to a large number of subscribers, the expressiveness of subscriptions is limited and complex computation across multiple data sources are often not supported. In addition to that, users often have to integrate a pub/sub service with other systems for data processing.

3 Big active data

To better understand the Big Active Data (BAD) vision and the challenges in creating BAD services, in this section, we describe the BAD problem in detail, enumerate the requirements of a BAD system, and describe a set of BAD building blocks for fulfilling these requirements.

3.1 A BAD world

In a BAD world, data could come from various systems and services constantly and rapidly. Many users would like to acquire and share the data and use it for different purposes. Some users may want to analyze the collected incoming data for retrospective analysis. They may ask questions (*Analytical Queries*) like:

> find the top 10 cities in terms of hateful tweets for each of the nearest 6 months both before and after the Parkland shooting.

Other users may want to continuously receive updates regarding data that is of interest to them. Such requests (*Subscriber Queries*) may cover different aspects of the data. For example:

- *Data content* Receive data when its content contains certain values—"send me tweets that are hateful";
- *Data enrichment* Receive data enriched with relevant information—"send me hateful tweets and their nearby schools";
- *Relationship with other data* Receive data when it relates to other data—"send me hateful tweets if they are near my location".

Based on different needs of the users in the BAD world, we characterize three types of **BAD users**:

1. *Data analysts* issue queries to analyze collected incoming data and/or other relevant data.
2. *Information subscribers* make subscriptions and receive updates continuously using BAD applications.
3. *Application developers* create BAD applications and provide BAD services to subscribers.

A full-fledged BAD system needs to serve all three types of users— analysts, subscribers, and developers— and should be able to scale to support a massive volume of data and a huge number of users.

3.2 The BAD building blocks

In order to provide the features described in Sect. 3.1, a BAD system needs to have the following building blocks:

- *Persistent storage* In order to support retrospective analysis, data enrichment with relevant information, and customized data subscription, the BAD system should provide persistent storage to store collected incoming data, relevant data, and subscription information. It should be possible to add data to the BAD system through ingestion facilities, loading utilities, or applications' CRUD opera-

tions with ACID transaction support. Since data is persisted, developers should be able to utilize auxiliary data structures (like indexes) for accelerating data access.

- *Ingestion facility* A large volume of data, of the interest of either subscribers or analysts, may come into the BAD system rapidly. Some of the data may need to be filtered, processed, or combined with other data on-the-fly to be used by other applications. In order to handle such data, the BAD system should provide an ingestion facility to help continuously ingest data from various external data sources reliably, efficiently, and to be able to scale out according to the growing workload. BAD users should be able to easily create an ingestion pipeline in the BAD system without having to write low-level programs.

- *Analytical engine* Data analytics enables analysts to reveal useful information from data. To help analysts understand the incoming data and its relationship with other relevant information, the BAD system should provide an analytical engine with support for declarative queries.

- *Data channels* In traditional Big Data applications, subscribers, who want to get data, rely on developers to translate their interest (subscriptions) into queries and then to retrieve data on behalf of subscribers. In practice, many subscriptions have similar structures like "send me hateful tweets from city X", "send me hateful tweets near my location", etc. To simplify creating BAD applications using the BAD system, we extract the shared structure among subscriptions and offer that as a service, namely a data channel, for subscribers to subscribe to with parameters. Data channels can be created using declarative queries and are managed by the BAD system.

- *Broker network* Subscribers of a data channel expect the latest updates of their data of interest to be delivered to them continuously. The BAD system needs to handle millions of subscribers subscribing to a channel and to allow multiple channels to run concurrently. Due to the volume of data exchanges between the BAD system and subscribers, the BAD system should include a broker network with caching and load-balancing strategies.

We depict the BAD system and the BAD users in Fig. 1. Rapidly incoming data (e.g., live tweets) is captured by the *Ingestion Facility* to be actively processed and persisted. Reference data (e.g., sensitive accounts, important facilities) is stored in the *Persistent Storage* and used in *Data Channels* and the *Analytical Engine*. *Data Channels* combine the persisted incoming active data and other reference data to provide customized information for *Information Subscribers* at scale via the *Broker Network*, following the design of *Application Developers* (e.g., to send threatening tweets near schools to police officers nearby); the *Analytical Engine* combines persisted active data and reference data to support complex analytical queries issued by *Data Analysts* (e.g., to list the accounts sending the most threatening tweets in the past month).

To the best of our knowledge, there is no existing Big Data platform that provides all the functionality needed from a BAD system. Some platforms can fulfill certain building blocks in the BAD system, but one would have to hand-wire multiple systems together to get all desired BAD features. A well-designed, integrated,
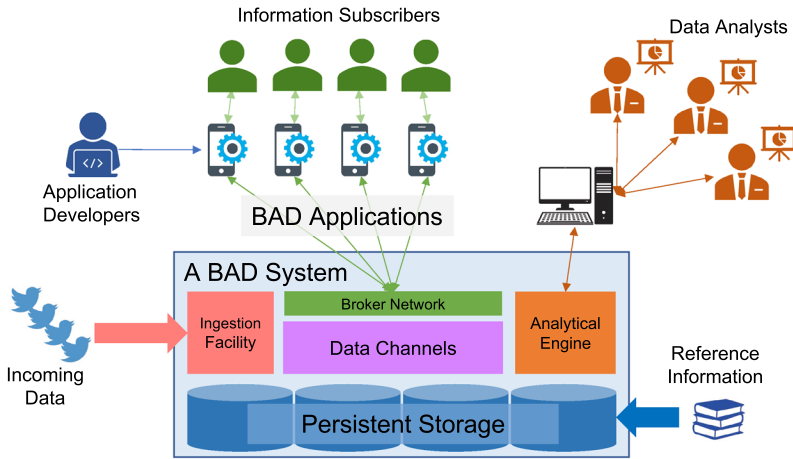
Fig. 1 A BAD system for a BAD world

and efficient BAD system with support for a declarative language can significantly reduce the effort required to create BAD services. In the following sections, we will introduce how we have built such a BAD system and how it fulfills the BAD requirements, in particular supporting *Subscriber Queries* at scale, as outlined in Sect. 3.1.

4 Repetitive BAD: BAD-RQ

We created the initial prototype BAD system, namely Repetitive BAD (BAD-RQ), by approximating continuous semantics using repetitive queries [13]. This was done by enhancing Apache AsterixDB, an open-source Big Data Management System that provides distributed data management for large-scale, semi-structured data. In this section, we present the user model of BAD-RQ and a high-level illustration of its internal evaluation to provide a context for BAD-CQ. Interested readers can refer to [13, 32–34] for a more detailed description of the whole BAD project.

4.1 A BAD repetitive use case

To illustrate BAD-RQ, we use a sample scenario in which we want to provide BAD services to police officers around tweets.[1] Users of these services include investigative officers as **analysts** who want to study tweets about certain events, and in-field officers as **subscribers** who patrol around the city and want to receive live tweets meeting certain requirements. Tweets come into BAD-RQ from an external system continuously, and each contains a hateful flag provided by the datasource indicating

---

[1] This example is for illustrating the BAD system and how it supports both analytical and active queries at scale. Similar use cases can be found in many other Big Data applications, such as COVID-19 alerting, monitoring, and tracking, Internet of things (IoT) device management, etc.

```
CREATE TYPE Tweet AS OPEN {
  tid: bigint,
  area_code: string,
  location: point
};
CREATE DATASET Tweets(Tweet) PRIMARY KEY tid;
CREATE INDEX t_location ON Tweets(location) TYPE RTREE;
```

Fig. 2 Datatype and dataset definition for Tweets

```
CREATE TYPE OfficerLocation AS OPEN {
  oid: int,
  location: point
};
CREATE DATASET OfficerLocations(OfficerLocation) PRIMARY KEY oid;
CREATE INDEX o_location ON OfficerLocations(location) TYPE RTREE;
```

Fig. 3 Datatype and dataset definition for officer location updates

whether this tweet is hateful and may relate to a potential crime. Location updates of patrolling in-field officers are also sent to BAD-RQ constantly to show their latest location. We describe the implementation of BAD building blocks in BAD-RQ and demonstrate how **developers** can utilize them for creating BAD services.

4.1.1 Persistent storage

In order to support analytical queries from analysts and subscriptions from subscribers, both incoming tweets and location updates need to be persisted in the BAD system. BAD-RQ offers the same storage functionality as AsterixDB, including all data types and indexes. AsterixDB organizes data under *dataverses* (similar to databases in an RDBMS). Without loss of generality, all data discussed in this section is stored in the "BAD" dataverse.

To store data in the BAD dataverse, we (as developers) need to create a *datatype*, which describes the stored data, and a *dataset*, which is a collection of records of a datatype. We define both the Tweet and OfficerLocation data types as "open", which makes the stored data extensible. The "hateful_flag" attribute, indicating whether a tweet is hateful, is not specified in the data type and thus it is an open (optional) field. When "hateful_flag" is not provided but needed for a BAD application, a developer could use a enrichment user-defined function (UDF) to enrich tweets during data ingestion [35]. We create a dataset *Tweets* for storing incoming tweets, a dataset *OfficerLocations* for storing location updates, and two R-Tree indexes on the location attribute of each dataset for more efficient data access. The DDL statements for creating both datasets are shown in Figs. 2 and 3 respectively.

4.1.2 Ingestion facility

Since tweets and location updates may come at a very rapid rate, the BAD system needs to intake such "fast" incoming data efficiently. AsterixDB provides data feeds for data ingestion from various data sources with different data formats [36].

Fig. 4 A data feed for ingesting tweets

```
CREATE FEED TweetFeed WITH {
  "type-name" : "Tweet",
  "adapter-name": "socket_adapter",
  "format" : "JSON",
  "sockets": "127.0.0.1:10001",
  "address-type": "IP",
  "insert-feed" : true
};
CONNECT FEED TweetFeed TO DATASET Tweets;
START FEED TweetFeed;
```

```
CREATE FEED LocationFeed WITH {
  "type-name" : "OfficerLocation",
  "adapter-name": "socket_adapter",
  "format" : "JSON",
  "sockets": "127.0.0.1:10002",
  "address-type": "IP",
  "insert-feed" : false
};
CONNECT FEED LocationFeed TO DATASET OfficerLocations;
START FEED LocationFeed;
```

Fig. 5 A data feed for ingesting location updates

```
LET stime = datetime("2017-07-14T10:10:00"),
  etime = datetime("2018-08-14T10:10:00")
FROM Tweets t WHERE t.timestamp > stime AND t.timestamp < etime
GROUP BY print_datetime(t.timestamp, "Y-M")
GROUP AS TweetsByMonth
SELECT print_datetime(t.timestamp, "Y-M") AS Month, (
  SELECT VALUE tbm.t.area_code FROM TweetsByMonth tbm
  GROUP BY tbm.t.area_code ORDER BY count(1) DESC LIMIT 10
) MostHatefulCities;
```

Fig. 6 An SQL++ query looking for the 10 most hateful cities in each month in a given time frame

We create a socket data feed to intake JSON formatted tweets using the statements shown in Fig. 4. Similarly, we create a data feed for intaking location updates sent by in-field officers in Fig. 5. In this use case, we deliver in-field officers nearby hateful tweets based only on their current location, so we create an UPSERT (i.e., insert if new, else replace) data feed by setting *"insert-feed"* to false. In cases where officers' entire movement history is needed, one can also create an INSERT data feed like the one used for tweets.

4.1.3 Analytical engine

BAD-RQ supports data analytics using the query engine in AsterixDB. It provides SQL++ [37, 38] (a SQL-inspired query language for semi-structured data) for users to construct analytical queries. SQL++ supports standard SQL query operations (SELECT, JOIN, GROUP BY, ORDER BY, etc.), spatial-temporal queries, operations designed for semi-structured data, etc. One can use the SQL++ query shown in Fig. 6 to answer the analytical query in Sect. 3.1. For a query executed multiple times with different constant expressions, analysts can also define it as a SQL++

```
CREATE FUNCTION mostHatefulCitiesByMonth(stime,etime) {
  FROM Tweets t WHERE t.timestamp > stime AND t.timestamp < etime
  GROUP BY print_datetime(t.timestamp, "Y-M")
  GROUP AS TweetsByMonth
  SELECT print_datetime(t.timestamp, "Y-M") AS Month, (
    SELECT VALUE tbm.t.area_code FROM TweetsByMonth tbm
      GROUP BY tbm.t.area_code ORDER BY count(1) DESC LIMIT 10
  ) MostHatefulCities
};
mostHatefulCitiesByMonth(datetime("2017-07-14T10:10:00"),
  datetime("2018-08-14T10:10:00"));
```

Fig. 7 A UDF based on an analytical query

```
CREATE FUNCTION RecentNearbyHatefulTweetsCount(oid) {
  FROM OfficerLocations o, Tweets t
  WHERE o.oid = oid AND t.hateful_flag = true
    AND spatial_distance(t.location, o.location) < 5
    AND t.timestamp > current_datetime() - day_time_duration("PT1H")
  SELECT count(*) AS HatefulTweetsNum, current_datetime() AS CurrentTime
};
RecentNearbyHatefulTweetsCount("0907");
```

Fig. 8 An UDF for counting hateful tweets near certain in-field officer given his/her officer ID

```
CREATE REPETITIVE CHANNEL RecentNearbyHatefulTweetCountChannel
  USING RecentNearbyHatefulTweetsCount@1 PERIOD duration("PT10M");
```

Fig. 9 Creating a data channel based on a UDF with a parameter

UDF and invoke it with parameters instead of re-constructing the same query every time. As an example, the analytical query in Fig. 6 can be encapsulated in the SQL++ UDF shown in Fig. 7.

4.1.4 Data channels

Since queries can be encapsulated as a UDF with parameters, and subscriptions with a similar structure can also be interpreted as a parameterized query, we can use a SQL++ UDF to group these subscriptions together and "activate" it as a data channel. Developers can create data channels based on SQL++ UDFs and offer them as services, and subscribers can subscribe to them with parameters to receive data of interest subsequently. As an example, if in-field officers want to know the number of hateful tweets near their current location in the past hour, we can first create the UDF in Fig. 8, which can be invoked using an officer's ID and returns the number of recent hateful tweets nearby. We "activate" this UDF using the statement in Fig. 9 by creating a data channel using this UDF. This channel has a configurable period "10 mins" indicating that it computes every 10 mins. In-field officers who subscribed to this channel will receive the number of nearby hateful tweets in the past hour every

```
CREATE BROKER BROKER_A AT "http://BROKER_A_HOST:BROKER_A_PORT/API";
```

Fig. 10 Registering a broker to BAD

```
SUBSCRIBE TO RecentNearbyHatefulTweetCountChannel("0907") ON ROKER_A;
```

Fig. 11 Subscribing to a channel with parameters on a broker

10 mins.[2] We will further discuss how a channel evaluation produces customized data for each subscriber in Sect. 4.2.

4.1.5 Brokers and subscriptions

The BAD system includes a broker sub-system for managing the communication with a large number of subscribers. A broker could be a single server that only forwards customized data to subscribers or a broker network that provides load balancing, subscription migration, and different caching strategies. Interested readers can refer to [34, 39] for more details. A developer can choose a broker suited for the use case and register it as an HTTP endpoint in the BAD system as in Fig. 10. A subscriber can then subscribe to a channel in the BAD system on this broker using the statement in Fig. 11. A given channel execution can produce customized data for subscribers subscribed on different brokers, and the customized data is sent to the corresponding brokers based on which brokers the subscriptions are subscribed on. A broker receives the customized data from channel executions and then disseminates it to its subscribers.

4.2 Data channel evaluation

As the core feature of the BAD system, data channels combine incoming data, relevant information, subscriptions, and broker information to produce customized data for each subscriber. In this section, we describe how BAD-RQ evaluates data channels to support a large number of subscriptions at scale.

4.2.1 Modeling brokers and subscriptions

As we mentioned in Sect. 3.2, subscribers subscribe to a data channel with parameters, and there could be millions of subscribers for a data channel. Given the large volume of subscriptions, separately evaluating a channel query (the underlying UDF of a channel) for each subscriber would be too computationally expensive. Inspired by [6], BAD-RQ stores subscriptions as data and evaluates the channel query using the analytical query engine. Benefiting from the query optimization, indexes, and

---

[2] While such precise requirement of sending nearby hateful tweets to every police officer is uncommon in practice, we use this as a synthetic example for illustrating the BAD system and for later demonstrating how the BAD system can scale regarding such complex use cases.

Fig. 12 Data type definitions for brokers and subscriptions (internal to BAD)

```
CREATE TYPE Brokers AS {
  dataverse_name: string,
  broker_name: string,
  broker_end_point: string
};
CREATE TYPE Subscription AS {
  subscription_id: uuid,
  broker_name: string,
  dataverse_name: string
};
```
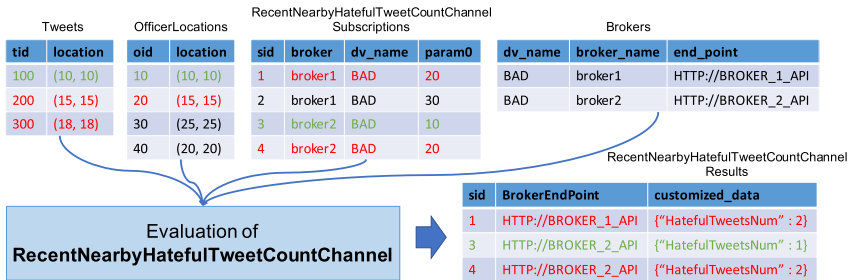

Fig. 13 A data sample for evaluating a data channel

distributed evaluation in AsterixDB, BAD-RQ can compute a channel query with a lot of subscriptions efficiently, and the channel evaluation process can also take advantage of the shared computation among subscriptions in order to serve more subscribers.

BAD-RQ uses the data types defined in Fig. 12 to store the broker and subscription information internally. Broker information is decoupled from subscriptions, so a broker record can be modified without having to update all related subscriptions. The subscription data type is defined as *open*, and the parameters of a subscription are stored as open attributes and named as *param0*, *param1*, *etc.* This allows a data channel to support an arbitrary number of parameters with arbitrary data types. The broker dataset is a permanent part of the BAD-RQ metadata. The subscription dataset is tied to the life-cycle of a data channel. When a developer creates a data channel (e.g., *RecentNearbyHatefulTweetCountChannel*), a corresponding subscription dataset (*RecentNearbyHatefulTweetCountChannelSubscriptions*) is also created, and this will be removed when the channel is dropped. Whenever a subscriber subscribes to the channel, a new subscription record is inserted into the subscription dataset.

4.2.2 An example of channel evaluation

In order to illustrate how BAD-RQ periodically computes a channel and produces customized data for each subscriber using broker and subscription information, we show a small data sample in Fig. 13 for the channel defined in Sect. 4.1.4, which returns the number of hateful tweets near a particular in-field officer. For illustrative simplicity, we assume all three tweets are posted within one hour and are hateful, and attributes not used for evaluation are not shown in the figure. The channel

```
SELECT customized_data,
  current_datetime() AS ChannelExecutionTime,
  sub.subscription_id AS SubscriptionId,
  b.broker_end_point AS BrokerEndPoint
FROM RecentNearbyHatefulTweetCountChannelSubscriptions sub,
  Brokers b,
  RecentNearbyHatefulTweetsCount(sub.param0) customized_data
WHERE sub.broker_name = b.broker_name AND sub.dataverse_name = b.dataverse_name;
```

Function
Inlining
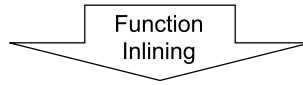
```
SELECT customized_data,
  current_datetime() AS ChannelExecutionTime,
  sub.subscription_id AS SubscriptionId,
  b.broker_end_point AS BrokerEndPoint
FROM RecentNearbyHatefulTweetCountChannelSubscriptions sub,
  Brokers b,
  (SELECT count(*) AS HatefulTweetsNum FROM OfficerLocations o, Tweets t
  WHERE spatial_distance(t.location, o.location) < 5 AND o.oid = sub.param0 AND t.hateful_flag = true
    AND t.timestamp > current_datetime() - day_time_duration("PT1H")) customized_data
WHERE sub.broker_name = b.broker_name AND sub.dataverse_name = b.dataverse_name;
```

Fig. 14 An illustrative query for computing a channel

evaluation combines information from four datasets, including *OfficerLocations*, *Tweets*, *RecentNearbyHatefulTweetCountChannelSubscriptions*, and *Brokers*, and it produces the customized data shown in the *RecentNearbyHatefulTweetCountCh-annelResults* dataset. Related tuples are colored the same. Taking red tuples as an example, we find two tweets near officer with oid 20s current location at (15, 15): tweet 200 at (15, 15) and tweet 300 at (18, 18). Also, there are two subscriptions (subscription 1 and subscription 4) subscribe to the nearby hateful tweet number of officer 20 (having param0 equal to 20). Subscription 1 is on broker 1, and subscription 4 is on broker 2. Based on the above information, BAD produces two notifications, one for each subscriber, and sends them to their corresponding broker APIs.

4.2.3 Channel evaluation internals

Evaluating a channel is equivalent to evaluating a query where we apply the underlying UDF to parameters from subscriptions to produce customized data. For example, evaluating the channel defined back in Fig. 9 is equivalent to evaluating the query shown in Fig. 14. In this query, we apply the UDF in Fig. 8 on parameters from subscriptions and nest the return value of the UDF into a "customized_data" field. The UDF can be inlined into the query, as shown, and be compiled and optimized together with it. The broker endpoint and subscription ID are also attached to each customized data record. The broker endpoint is used for the channel to send the result to a corresponding broker API, and the subscription ID is used by brokers to identify which subscriber the customized data should be delivered to.

    Since the query used for evaluating a channel is computed on the analytical engine of AsterixDB, it can be optimized by the query optimizer and be accelerated by utilizing
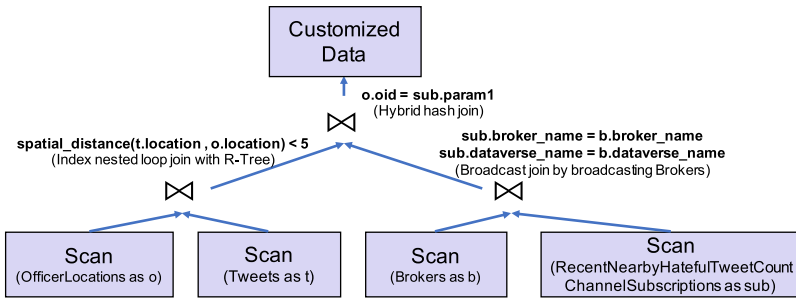
Fig. 15 A query plan for channel evaluation

efficient algorithms and indexes. Under the hood, the query in Fig. 14 compiles into a query plan as shown in Fig. 15. BAD can use use an R-Tree index to accelerate the spatial join between *Tweets* and *OfficerLocations*. Also, since the number of brokers is small compared with subscriptions, it can broadcast the Brokers to avoid unnecessary shuffling of the Subscriptions dataset. It can use a hybrid hash join to join the two intermediate results in parallel on all nodes in the cluster.

4.2.4 Customized data delivery

A data channel executes on a specified period (time interval) to generate customized data. Depending on subscribers' preferences, the customized data can either be eagerly or lazily delivered. In the eager (push) mode, the produced data is pushed to brokers directly so they can immediately disseminate the data to subscribers. As the produced subscription result data is not persisted in BAD-RQ in this mode, brokers have to be fault-tolerant to avoid data loss. In the lazy (pull) mode, the customized data is first persisted in the BAD-RQ storage engine. The channel then sends a notification to the brokers whose subscribers have customized data that was produced in this channel execution. A broker that receives such a notification then pulls the customized data from BAD-RQ and distributes it to the subscribers. To this end, a result dataset (*RecentNearbyHatefulTweetCountChannelResults*) is created for persisting produced customized data when a "lazy" channel is created. The result dataset has an index on the "ChannelExecutionTime" attribute for accelerating result pulling. Since the customized data is persisted in the storage engine in this mode, brokers then have the flexibility to choose when to disseminate the notifications to subscribers, and the storage engine ensures data safety. BAD-RQ uses the pull (broker-initiated) mode as the default mode for its channels.

```
CREATE FUNCTION AddIngestionTime(incoming_record) {
  object_merge({"ingested_timestamp": current_datetime()}, incoming_record)
};
```

Fig. 16 A UDF for adding ingestion time

5 Continuous BAD: BAD-CQ

BAD-RQ "activates" a UDF (a parameterized query) to create a data channel that allows subscribers to constantly receive updates of interesting data. Although BAD-RQ demonstrates how to transform a "passive" Big Data system into a basic "active" one for creating BAD services, it faces several limitations when users have more requirements.

In some use cases, subscribers may want the latest information delivered **incrementally**. Examples include "send me new hateful tweets on campus", "notify me when an emergency happens around me", and "let me know when crimes happen near my house". We call such use cases **Continuous BAD**. In order to support them, data channels in BAD need to provide continuous semantics, in which they continuously return incremental updates. Developers using BAD-RQ could try to approximate continuous semantics using repetitive channels, but such approximations would face challenges due to the lack of native support for true continuous semantics. In this section, we look at an example of continuous BAD and demonstrate how to use BAD-RQ to approximate it. We discuss the limitations of this approximation and then introduce a new BAD service—BAD-CQ—designed for supporting continuous BAD.

5.1 Approximately continuous queries

To illustrate continuous BAD and its BAD-RQ approximation, we look at a simple continuous use case where *"in-field officers (subscribers) want to know **new** hateful tweets near their current location"*. We introduce the setup for approximating continuous semantics in BAD-RQ and show how to construct a repetitive channel query for this approximation.

5.1.1 BAD timestamps

As subscribers are interested in **new** tweets, BAD-RQ needs to determine which portion of the collected tweets are new (i.e., tweets ingested but not yet reported). Different from streaming engines where all data in the engine is **new**, and **old** data is aged out, BAD-RQ keeps all data in the storage for supporting other services (e.g., data analytics). In order to differentiate new data from old, BAD-RQ needs to utilize timestamps.

In some cases (like tweets), incoming data comes with a "timestamp" attribute which indicates when was a data item created (a.k.a., valid time [40] or event time [20]). This attribute could potentially be used for differentiating new tweets from old ones. However, this would introduce additional complexity in handling out-of-order

```
CREATE FUNCTION NewNearbyHatefulTweets(oid){
  SELECT t
  FROM OfficerLocations o, Tweets t
  WHERE spatial_distance(t.location, o.location) < 5
    AND o.oid = oid AND t.hateful_flag = true
    AND t.ingested_timestamp >
      current_datetime() - day_time_duration("PT10S")
};
CREATE REPETITIVE CHANNEL NewNearbyHatefulTweetsChannel
  USING NewNearbyHatefulTweets@1
PERIOD duration("PT10S");
```

Fig. 17 A repetitive data channel looking for new nearby hateful tweets

arrivals. Besides, when such an attribute is not provided in the incoming data, we still need to find another solution[3]. BAD-RQ allows developers to attach timestamps to incoming tuples during data ingestion by attaching a UDF to the ingestion pipeline. For this use case, we can create the UDF shown in Fig. 16 and attach it to the tweet data feed defined in Fig. 4. This UDF adds an "ingested_timestamp" attribute to each incoming tweet, which marks the current date time when the tweet first enters the pipeline (a.k.a, ingestion time [20]). We can utilize this timestamp to infer the arrival order of tweets and differentiate **new** tweets from **old**.

5.1.2 A repetitive approximation

With BAD timestamps, we can then construct a repetitive data channel to serve new nearby hateful tweets with a designated channel period, as shown in Fig. 17. In this channel, we look for hateful tweets ingested in the past 10 s from the time when the channel executes. These tweets are new and thus haven't been examined yet. We join them with officers' current locations and look for nearby new hateful tweets for each subscribed in-field officer. The channel is defined to execute every 10 s, so subscribers can continuously receive new nearby hateful tweets. This allows us to approximate continuous (incremental) semantics with a repetitively executed channel query that runs every 10 s and looks back 10 s.

5.1.3 Challenges in approximation

Although developers could use BAD-RQ to approximate continuous semantics just as shown, such an approximation is not perfect in practice and could fail to have continuous semantics in some circumstances. Also, due to the lack of native syntax support for continuous semantics, constructing an approximation query can become very complex. Challenges include:

- *Scheduling delay* We approximate the continuous semantics by examining data ingested in the past execution period (e.g., 10 s) from the current channel execu-

---

[3] Streaming Engines (such as Spark Structured Streaming) that compute with event time offer watermarking to handle late arrivals. BAD-RQ with BAD timestamps (and BAD_CQ later introduced in Sect. 5.2) can provide similar functionality with proper channel queries. Here we focus on the general use cases without assuming the existence of event time.
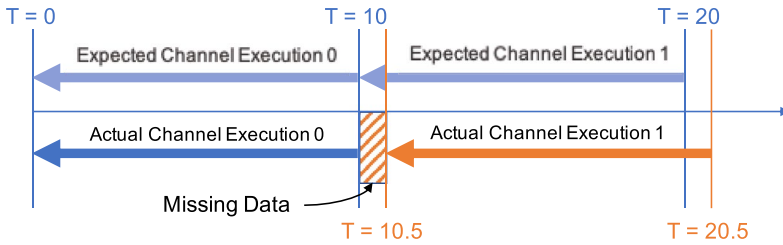
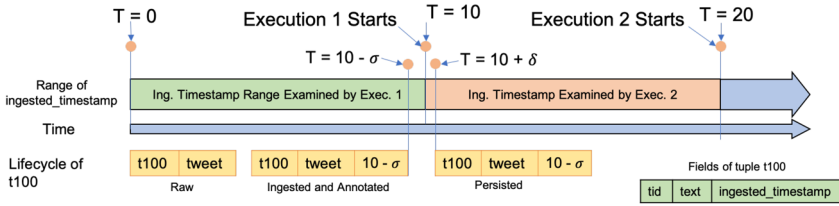**Fig. 18** Missing data due to scheduling delays

**Fig. 19** Missing tuple due to early timestamping

tion time. To perfectly approximate a continuous query, we rely on BAD-RQ to schedule the channel execution *on time* to make sure that all incoming data is examined. However, this is impractical in practice, especially in a distributed environment.

If a scheduling delay happens, some data can be missed by the channel, as shown in Fig. 18. This channel executes every 10 s and examines data ingested from the past 10 s. If the actual channel execution 1 is delayed from T = 20 to 20.5, the data ingested from T = 10 to 10.5 will not be examined and thus missed.

- *Early timestamping* The approximation of BAD-RQ uses the ingestion timestamp for determining whether ingested data should be examined in a channel execution. However, since the ingested (timestamped) data does not become visible to channel execution instantaneously due to delays in data transmission, data enrichment (if any), secondary index(es) updating (if any), primary index updating, and waiting for the storage transaction to complete, there is a chance that a running channel execution could miss the data just ingested, even if the channel execution is scheduled on time. This is illustrated in Fig. 19, where channel execution 1 starts at $T = 10$ and a tuple t100 is ingested at $T = 10 - \sigma$ and later persisted and becomes visible to queries at $T = 10 + \delta$ due to the delay.[4] Channel execution 1 does not examine t100 because the tuple is not in storage yet, and channel execution 2 will not examine t100 either, because the tuple has an ingested_timetsamp that is smaller than 10 (i.e., too old). Thus, tuple t100 is missed.

---

[4] In practice, this time gap is very small. We emphasize the delay in Fig. 19 for illustration purposes.

Fig. 20 Storage format of an active record
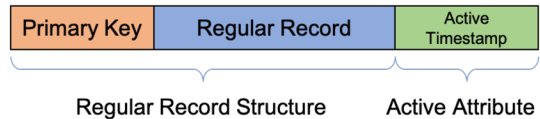
- *Inappropriate data access* We have attached an explicit timestamp ("ingested_timestamp") attribute to mark the ingestion time of incoming tweets.

  This attribute then exists as part of the user data, and other users of the BAD system can access it. This raises the potential risk that other users may accidentally modify this attribute and cause data channels to fail. Additionally, this auxiliary information may cause confusion for non-channel users such as data analysts.
- *Complex approximation query* In order to approximate continuous semantics, we have chosen the same time period in the temporal predicate and the channel execution period, as shown in Fig. 17. Such a correspondence needs to be managed manually and carefully by developers. When channel queries become more complex and involve multiple incoming data sources, constructing a proper approximation query can be challenging. One would have to add proper temporal predicates for each of the data sources, and when there are joins between these data sources, which portion of the collected data from one data source should be joined the other one needs to be carefully specified with temporal predicates. These temporal predicates would increase the query complexity and make such queries very difficult to write.

The above challenges of using BAD-RQ to approximate continuous semantics introduce risks of missing data and cause difficulties for developers in creating continuous BAD applications. In order to properly support continuous BAD, we introduce a new BAD service— BAD-CQ—with native support for continuous query semantics.

5.2 BAD-CQ

In this section, we first introduce the new building blocks needed for providing continuous semantics in BAD-CQ, and then we show how to utilize them to create continuous data channels for continuous BAD.

5.2.1 Active datasets

As we have discussed in Sect. 5.1.1, BAD persists all data to support retrospective analysis. To help data channels differentiate new data from old, we need to

Fig. 21 Access active datasets with filters

SELECT VALUE t FROM Tweets t WHERE active_timestamp(t) > 11;

max_active_ts: 5   max_active_ts: 9   max_active_ts: 12
min_active_ts: 0   min_active_ts: 6   min_active_ts: 9

```
CREATE ACTIVE DATASET Tweets(Tweet) PRIMARY KEY oid;
CREATE ACTIVE DATASET OfficerLocations(OfficerLocation) PRIMARY KEY oid;
```

Fig. 22 Datatype and dataset definition for officer location updates

timestamp incoming data and use timestamps for proper continuous channel evaluation. To avoid the previously mentioned drawbacks of adding an ingestion timestamp to user data, we introduce a new type of datasets—*Active Datasets*—in BAD-CQ. Unlike regular datasets in AsterixDB, a record (active record) stored in an active dataset contains not only user data but also a "hidden" active attribute: "_active_timestamp". This helps BAD-CQ to evaluate continuous channel queries. This attribute is stored alongside users' data but separated from the regular record content. It is "invisible" to users and can only be accessed using active functions (to be discussed soon). The storage layout of an active record is shown in Fig. 20.

As the BAD system runs in a distributed environment, which clock to use to assign active timestamps needs careful consideration. One might first consider using a single clock to assign all active timestamps. This would be convenient because then all active timestamps would be directly comparable, and we would only need to figure out one active timestamp range to identify all the new data. However, having a master clock would require either routing all data to a single node, which would create a bottleneck in the system, or synchronizing clocks on multiple nodes, which can be very challenging in a distributed environment. In BAD-CQ, we instead use the local clock on each node to assign active timestamps to the active records stored on it for scalability. Active timestamps are assigned in the storage engine, after the locking phase. This makes sure that incoming records will become visible to running queries as soon as they are timestamped. Although the **new data** on each node may now have a different active timestamp range, we can introduce an active timestamp management mechanism with additional query optimization rules to make sure that channel queries are evaluated correctly. We will further discuss this in Sect. 5.3.

Considering that active timestamps often need to be compared in channel queries, we can optimize these comparisons to improve channel performance. One might consider creating a secondary index on active timestamps, but this would take additional disk space and incur additional access overhead when the selectivity is high [41]. As the active timestamps of an active dataset grow monotonically, we can instead utilize the filter feature in the AsterixDB storage engine to avoid accessing irrelevant data [42]. The BAD storage engine uses Log-Structured Merge (LSM) Trees as its storage structure [43]; they perform batch updates into components to avoid the cost of random writes and then read them sequentially for data access. One can designate a filter attribute when creating a dataset, and every LSM component of this dataset is then decorated with the maximum and minimum attribute values of its stored records. When a query containing a filter attribute comparison comes, it
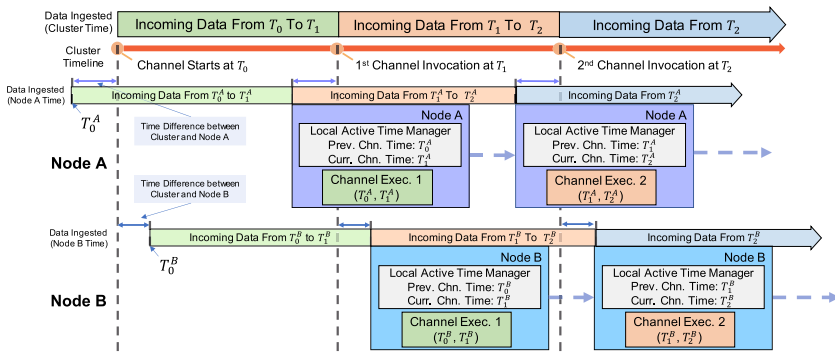
Fig. 23 An illustration of active timestamp management

can quickly skip irrelevant components by examining their maximum and minimum filter values. For active datasets, we use the active timestamp as the filter attribute to accelerate channel queries, as shown in Fig. 21. The *active_timestamp(t)* function reveals the active timestamp of the tuple t stored in the active dataset *Tweets*, as will further be discussed in Sect. 5.3.

The syntax for creating active datasets is straightforward. An active dataset can be created with a regular data type, and the active attribute and filter are automatically configured behind-the-scene. One can create two active datasets *Tweets* and *Officer-Locations* using the statements in Fig. 22. Active datasets can also be accessed in regular queries just like non-active datasets. There is an extra overhead when reading active datasets due to the additional space for storing active timestamps. We will see from later experiments that this overhead is relatively small. When not used in query evaluation, active timestamps are projected out from the active records as early as possible to avoid potential transmission overhead.

5.2.2 Active timestamp management

With active datasets, we now need to "teach" channels to utilize the active timestamps to recognize **new** data and to guarantee continuous semantics. The basic idea is straightforward: keep track of the channel execution times and compare them with active timestamps to find the new data. As mentioned in Sect. 5.2.1, each node uses the local time to assign active timestamps, so we also need to use local time for tracking channel execution times and make sure they are properly compared with active timestamps. We create a local **active timestamp manager** on each node to keep track of the *previous channel execution time* and the *current channel execution time* under the local clock. When a channel executes on a node, these two timestamps are used to determine which portion of the stored data should be considered for this execution.

To demonstrate how multiple local active timestamp managers can work to offer continuous semantics, we consider the channel defined in Sect. 5.1 that looks for

new nearby tweets for in-field officers. We show an illustrative channel execution example in Fig. 23.

In this example, we use the cluster controller (CC) time as the (conceptual) cluster time. Since not all nodes are synchronized on time, current timestamps on different nodes can be different. In this case, when CC starts the first channel execution at time T_0, Node A marks the channel start time under its local time as T_0^A, which is *"logically before"* T_0, and Node B marks the channel start time under its local time as T_0^B, which is *"logically after"* T_0. When the CC invokes the first channel execution at T_1, every node examines the tweets ingested and persisted from the previous channel execution time to the current channel execution time. From Node A's perspective, all tweets ingested from T_0^A to T_1^A are examined. From Node B's perspective, tweets ingested from T_0^B to T_1^B are examined. Although T_1, T_1^A, and T_1^B are different, from the CC's (and subscribers') perspective, only nearby hateful tweets from T_0 to T_1 are reported to subscribers. This guarantees the continuous semantics for this channel. The channel's previous channel execution and current channel execution time are each progressed with each channel execution. They are updated instantly when a channel execution job first accesses an active dataset used for the channel. This makes sure that all incoming tweets that were persisted before the current channel execution can all be safely examined in the current channel execution.

The active timestamp manager enables BAD-CQ to provide continuous semantics in a distributed environment without time synchronization. The monotonically increased active timestamps on each node in fact act like sequence numbers. The local active time manager marks the range of sequence numbers for each channel execution (as its previous and current channel execution time) and allows it to find the new data.

5.3 BAD-CQ syntax and optimization

Active datasets and active timestamp management allows BAD-CQ to provide continuous semantics. In order to enable users to use active timestamps and channel execution times for constructing channel queries, we introduce several active functions in this section. Each active function takes a parameter that refers to tuples from active datasets. Applying active functions on normal datasets will cause a query compilation exception. In order to describe the functionalities of active functions, we use a tuple *t* from the active dataset *Tweets* as an example. The active functions are as follows:

- *active_timestamp(t)* reveals the active timestamp of the tuple *t*.
- *previous_channel_time(t)* returns the previous channel execution time on the node where the tuple *t* is persisted, as defined in Sect. 5.2.2. Note that every node has its own (local) channel time for a channel, and dataset *Tweets*'s tuples could be persisted on multiple nodes, so this function is evaluated locally on each node at run time, and tuples from Tweets used in the channel could have different previous channel times.

```
CREATE CONTINUOUS CHANNEL CQNewNearbyHatefulTweets(oid)
  PERIOD duration("PT10S") {
  SELECT t
  FROM OfficerLocations o, Tweets t
  WHERE spatial_distance(t.location, o.location) < 5
    AND o.oid = oid AND t.hateful_flag = true AND is_new(t)
};
```

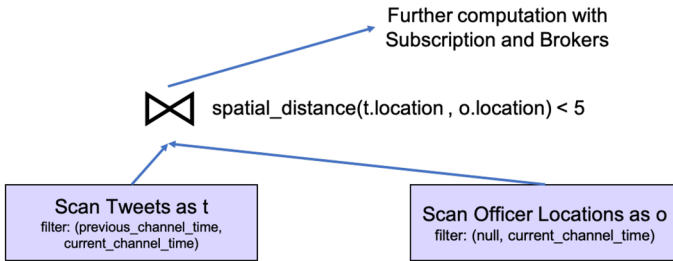**Fig. 24** A continuous channel for new nearby hateful tweets

Fig. 25 Query plan for new nearby hateful tweet channel

- **current_channel_time(t)** returns the current channel execution time of the tuple *t*, as defined in Sect. 5.2.2. Similar to *previous_channel_time*, *current_channel_time* is also computed locally at run time, and tuples from *Tweets* could have different current channel time.
- **is_new(t)** returns a boolean value indicating whether tuple *t* is new to the current channel execution. The return value of *is_new(t)* is equivalent to the following expression:

  previous_channel_time(t) < active_timestamp(t) AND active_timestamp(t) < current_channel_time(t).

With active functions, a developer can conveniently construct continuous channels with continuous semantics. Here we show an example for the use case described in Sect. 5.1, where subscribers would like to receive new tweets near in-field officers. We use a different user model in BAD-CQ. Data channel definition in BAD-CQ is not based on UDFs, since active functions are not meaningful outside. Executing *previous_channel_time* and *current_channel_time* functions in regular queries return 0 and current cluster time respectively. Using BAD-CQ's active functions, a developer can create a continuous channel for the new nearby hateful tweets using the statement shown in Fig. 24.

In order to assist channel evaluation with active functions and to improve channel performance, we introduce two new query optimization rules into BAD-CQ. First, when compiling a continuous channel query, we push the *current_channel_time* function into the leaf node of the query plan—the data scan operator of an active dataset—as the filter's maximum value. This is because when an active dataset is accessed in a channel execution, only data before the current channel execution time is relevant. We use this to quickly skip data coming after the current execution

```
CREATE CONTINUOUS CHANNEL UnseenNearbyHatefulTweets(oid)
  PERIOD duration("PT10S") {
  SELECT t
  FROM OfficerLocations o, Tweets t
  WHERE spatial_distance(t.location, o.location) < 5 AND o.oid = oid
    AND t.hateful_flag = true
    AND (is_new(o) OR is_new(t))
};
```

Fig. 26 A continuous channel for unseen nearby hateful tweets

SELECT t
FROM OfficerLocations o, Tweets t
WHERE spatial_distance(t.location, o.location) < 5 **AND** o.oid = oid **AND** t.hateful_flag = true
    AND (is_new(o) OR is_new(t));

Expanding

SELECT t
FROM OfficerLocations o, Tweets t
WHERE spatial_distance(t.location, o.location) < 5 **AND** o.oid = oid **AND** t.hateful_flag = true
    AND active_timestamp(t) < current_channel_time(t) **AND** active_timestamp(o) < current_channel_time(o)
    AND (active_timestamp(t) > previous_channel_time(t) **OR** active_timestamp(o) > previous_channel_time(o));

Fig. 27 Expanding a continuous channel query with active functions

starts. Second, we push the *previous_channel_time* function down towards the leaf of the query as much as possible, and we use it as the filter's minimum value for active datasets when applicable. Whether this function can be pushed into the data scan operator depends on the specific channel query. For the channel query defined in Fig. 24, we can indeed push *previous_channel_time(t)* into the *Tweets* scan operator and use it as the minimum filter, as shown in Fig. 25. [5]

When the *previous_channel_time* function cannot be pushed all the way down into a data scan operator, we need to attach its node-dependent value (i.e., the previous channel execution time on a node) to the active records read from this node. In this case, the comparison between active timestamps and the *previous_channel_time* function is rewritten into a comparison between active timestamps and this attached previous channel execution time value. This makes sure that even if active records are shuffled around in the cluster, the comparison between their active timestamps and previous channel time will be evaluated correctly. To explain how the second rule works in this scenario, we introduce another continuous use case, where *"in-field officers (as subscribers) would like to receive nearby hateful tweets he/she has not seen before"*. In this case, we not only need to consider a new tweet posted near an in-field officer, but also tweets that were not nearby but that become nearby due to the movement of in-field officers. We can create a continuous channel for this use case as shown in Fig. 26.

In this continuous channel query, the active functions *is_new(o)* and *is_new(t)* are expanded to the corresponding query predicates based on active timestamps, previous channel execution time, and current channel execution time as shown in

---

[5] In this channel, we only need officers' latest location, so there is no lower bound on active timestamps of *OfficerLocations*. We will introduce another example in Sect. 5.4 which requires recent location updates and utilizes the minimum filter on *OfficerLocations*.
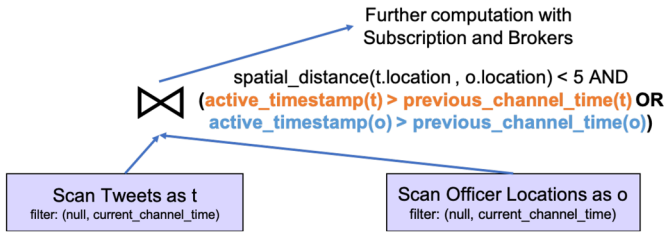
Fig. 28 Query plan for unseen nearby hateful tweet channel

Fig. 27. Following the first optimization rule, the current time timestamp of both *Tweets* and *OfficerLocations* are pushed into the corresponding data scan operators. However, the previous channel execution time cannot be pushed thoroughly, because the disjunctive predicate "*active_timestamp(t) > previous_channel_time(t) OR active_timestamp(o) > previous_channel_time(o)*" also needs data from before the previous channel execution time from both datasets. Following the second optimization rule, this continuous channel query can be compiled into the plan shown in Fig. 28. The disjunctive predicate is evaluated in the join operation that is computed across all nodes, and data is shuffled around in this process [6]. Notice now that since the previous channel execution time is attached to active records, we can compare the active timestamp with the channel execution time under the same local clock, even if records are shipped to another node.

Different from the implicit query rewriting in Tapestry [5] and the delta files in NiagaraCQ [6], BAD-CQ allows developers to construct queries using active functions that are best suited for their use cases, and it takes advantage of the storage engine for accelerating channel queries without having to introduce additional data structures. Developers can write a query using the *is_new* function and let the query compiler rewrite it into an incremental query, or they can use the *active_timestamp* function to expose the active timestamps and directly compare them with channel times or other times. The BAD-CQ user model uses datasets to hold the collected incoming data and other existing data. This provides developers with a unified query model and lets them to reuse all dataset processing operations when defining channels. The principles underlying the BAD-CQ approach are general - i.e., other database systems supporting declarative queries could also be adapted to provide continuous semantics like BAD-CQ.

5.4 BAD-CQ semantics

To better understand the query semantics provided in BAD-CQ, we dive into the details of several continuous BAD use cases in this section. We focus on the scenario where in-field officers would like to get nearby hateful tweets with different

---

[6] Depending on the workload, the execution plan for the channel query can choose either to broadcast *Tweets* or *OfficerLocations*.
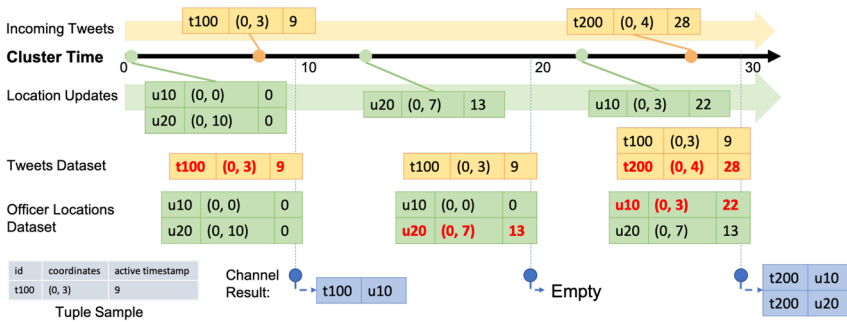
Fig. 29 Officer u10 subscribing to CQNewNearbyHatefulTweets(u10) and officer u20 subscribing to CQNewNearbyHatefulTweets(u20)
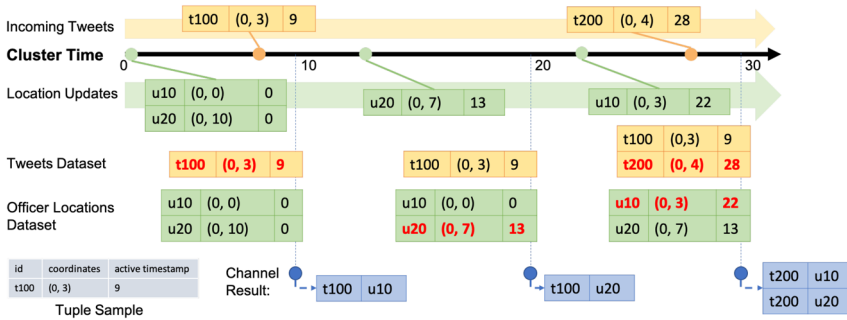

Fig. 30 Officer u10 subscribing to UnseenNearbyHatefulTweets(u10) and officer u20 subscribing to UnseenNearbyHatefulTweets(u20)

preferences, and we use data samples to show how BAD-CQ produces notifications for different channels.

5.4.1 New nearby hateful tweets

We first look at the example from Sect. 5.1, where in-field officers would like to receive new nearby hateful tweets. The channel is defined in Fig. 24. We use the *is_new* function to look for **new** tweets that have not been sent to subscribers, and we use the officers' latest locations to look for nearby tweets.

In Fig. 29, we show a channel execution example with several sample data records. In order to focus on the channel execution process, irrelevant attributes of tweets and officer location updates are not shown in the figure. The channel starts at time 0, and in-field officers u10 and u20 have initial location at time 0 of (0, 0) and (0, 10), respectively. At 9s, the first tweet t100 arrives and its

```
CREATE CONTINUOUS CHANNEL NewNearbyHatefulTweetsForActiveOfficers(oid)
  PERIOD duration("PT10S") {
    SELECT t
    FROM OfficerLocations o, Tweets t
    WHERE spatial_distance(t.location, o.location) < 5
      AND o.oid = oid AND t.hateful_flag = true AND is_new(t) AND is_new(o)
};
```

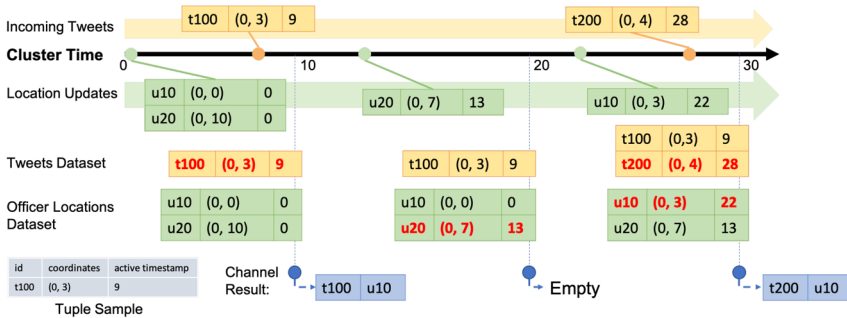**Fig. 31** A continuous channel for new nearby hateful tweets

Fig. 32 Officer u10 subscribing to NewNearbyHatefulTweetsForActiveOfficers(u10) and officer u20 subscribing to NewNearbyHatefulTweetsForActiveOfficers(u20)

location is (0, 3). When the channel first executes at 10s, only tweet t100 is near in-field officer u10, so the channel produces one notification for u10. After that, u20 updates his/her location to (0, 7) at 13s. When the channel executes at 20s, as there is no new tweet after the previous channel execution, no notification is produced. Later, u10 updates his/her location to (0, 3) at 22s, and a new tweet t200 located at (0, 4) arrives at 28s. When the channel executes at 30s, both u10 and u20 have t200 nearby, so the channel produces two notifications for each of the corresponding officers.

5.4.2 Unseen nearby hateful tweets

In the previous use case, in-field officers receive a hateful tweet only if the tweet is temporally new. In another use case, officers may also be interested in older nearby hateful tweets that they have not seen before (which could contain useful information). The channel definition for this use case is shown in Fig. 26.

We use the same data sample in Sect. 5.4.1 to explain how this channel works. As shown in Fig. 30, the channel acts the same way as the previous one and produces one notification for u10 in the first channel execution. In the second channel execution, the location update of u20 from (0, 10) to (0, 7) makes t100 become nearby to u20, so the channel produces one notification for u20 at 20s to notify this officer about this previously unseen tweet. The third channel execution starts at 30s and produces two notifications for u10 and u20, respectively, as both in-field officers have not seen this new tweet.

5.4.3 New nearby hateful tweets for active officers

In the previous use cases, even if an officer is not updating his/her location constantly (e.g., in order to reduce power/data plan consumption), the channel can still be producing notifications for them based on their last known location. When the officer reconnects, the broker sub-system can pull notifications that were produced "offline" from the BAD storage engine and send them out. If we want to produce notifications only to "active" in-field officers (who are their updating the locations to the system regularly), one can create the continuous channel defined in Fig. 31. Different from the channel defined in Fig. 24, we now only look for new hateful tweets for officers who have recently updated their locations instead of all officers. Those who are not updating their locations "actively" will not receive nearby hateful tweets while they are inactive.

Following our data sample used in previous use cases, the execution process of this channel is shown in Fig. 32. Similarly, the first channel execution produces one notification based on u10 about t100. In the second channel execution, no notification is generated since there is no new incoming tweet. In the third channel execution, we produce one notification about the new tweet t200 for u10 who has recently updated his/her location. Although t200 is also near u20, we do not produce a notification for him/her since u20 is not "active". As we can see from these sample use cases, active functions offer the flexibility and expressiveness of working with both the *new* and *historical* data. Developers can use active functions to conveniently construct a wide range of suitable queries for their BAD applications.

6 GOOD: a not BAD approach

In order to fully support BAD applications without the BAD system, one would have to glue multiple existing Big Data systems together. In this section, we discuss a Not-BAD approach, which we call GOOD— **G**luing **O**odles **O**f **D**ata platforms— approach to approximate the BAD system. We introduce a GOOD system that consists of several Big Data systems, illustrate how to configure it for creating BAD services, and compare it with the BAD system.

6.1 The GOOD architecture

Following our discussion in Sect. 3.1, a GOOD system also needs to serve all three types of BAD users: **Subscribers** who want to customize data and receive constant updates, **Developers** who create BAD applications to serve subscribers, and **Analysts** who analyze data using declarative queries. Such a system should provide the following features:

- Efficient data ingestion for rapid incoming data.
- Data customization based on a large volume of subscriptions.
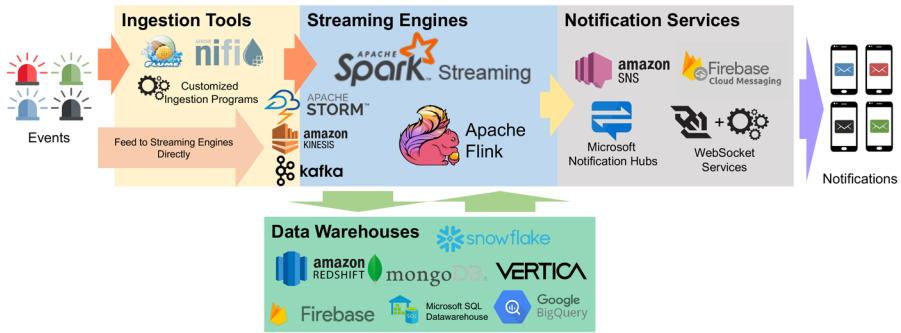- Data analytics with a declarative language.

Fig. 33 A GOOD architecture

- Persistent storage for incoming data and other relevant information with ACID guarantees.
- Customized data delivery to a large number of subscribers.

An existing Big Data system alone can only fulfill a portion of the BAD requirements. For example, Apache Spark Structured Streaming offers on-the-fly data processing but lacks persistent storage that provides ACID transaction support. Amazon's Simple Notification Service (SNS) supports cloud-based pub/sub, but the expressiveness of subscriptions is limited to the content of publications. A user wanting to build BAD applications would thus have to glue multiple systems together. We can break down a proposed GOOD system architecture into different components and categorize existing Big Data systems with respect to this GOOD architecture, as shown in the overview Fig. 33. Users need to pick one or more systems from each component to assemble a bundled GOOD system to meet the BAD requirements. We describe the functionality of each component as follows:

- *Ingestion tools* collect data from external data sources and help distribute the data to downstream components. In some cases, users could implement their own ingestion programs to handle specific ingestion protocols. With the growth of stream processing, many ingestion tools now also support on-the-fly data processing (with various limitations). This blurs the boundary between ingestion tools and streaming engines. Here we still consider them as different GOOD components to emphasize their functional differences.
- *Streaming engines* today come in two different flavors. One (e.g., Apache Storm, Apache Kafka) focuses on efficient and reliable data distribution and allows users to hang data processing units onto the pipeline. The other (e.g., Apache Flink, Apache Spark Structured Streaming) focuses on enabling real-time data analytics as if working with non-streaming data. Users could glue multiple streaming engines together to benefit from both flavors (such as gluing Kafka with Spark Structured Streaming). GOOD can use streaming engines to combine incoming data, subscription information, and other relevant data to produce customized notifications.

Fig. 34 A concrete GOOD system

- *Data warehouses* (and database systems[7]) provide data persistence and support for data analytics. We want a data analytics capability as a part of the GOOD system. Thus, incoming data should be persisted for retrospective analysis. Subscriptions and other relevant data used for producing customized data are also persisted in the data warehouse and loaded into the streaming engine for processing. We choose not to replicate data in both the streaming engine and the data warehouse to avoid data inconsistencies and a need for constant migration of updates between them.
- *Notification services* deliver customized data produced by streaming engines to interested subscribers. Users could choose cloud-based services such as Amazon SNS or Firebase Cloud Messaging to send notifications to subscribers via SMS or Email, or they also could build their own notification services based on Web-Socket.

Every component of the GOOD system must be horizontally scalable to ensure that it can support a large number of subscribers, just like the BAD system. Even with this scalable architecture, it would be impractical for the GOOD system to compute/customize an incoming data item for every subscriber independently, especially when the incoming data arrives rapidly. In order to best approach the BAD system's scalability requirement, we also adopt the data channel model in our GOOD system architecture by grouping similar subscriptions into a data channel and evaluating them together. Next, we will consider a sample GOOD system to explain how it can receive, customize, and deliver data.

6.2 A GOOD system

The GOOD architecture offers a way to approximate the BAD system by gluing multiple existing Big Data systems together. One could choose various combinations among the options in Fig. 33 for creating a GOOD system. In order to compare the GOOD system with the BAD system toe-to-toe, we have constructed a sample GOOD system using several component systems that have been widely used in practice, as shown in Fig. 34. These component systems were chosen based on our perception of their popularity. Each of them provides good performance for their targeted use cases. We use them here to illustrate how to build a GOOD system off-the-shelf. More importantly, as we will see later in the discussion and experiments,

---

[7] Here we use the term Data Warehouse loosely, focusing more on its data management part.

there are computing and performance limitations due to bundling different systems together for the BAD use case in addition to the effort required to glue them together.

We choose Apache Kafka for data ingestion and use Spark Structured Streaming for data processing, as suggested in the Spark Structured Streaming documentation [44]. Although Kafka also supports several data processing operations via Kafka Streams [19], we choose Spark Structured Streaming for its richer query semantics, which is closer to the BAD system's offering. We use MongoDB as the data warehouse for persisting incoming data as well as other reference data used for analytics. We chose it because it is the most popular database for "modern data" (NoSQL), which is the kind of data we expect many BAD applications to need. Later (Sect. 7) we will also consider using Delta Lake [45], the storage layer of Databricks' new Lakehouse architecture, in this role because of its relationship to the Spark ecosystem, where Delta lake is gaining usage for providing ACID transactions in Spark as the data storage. Finally, we choose AmazonSNS for notification delivery. Each component of the GOOD system can be described and configured as follows:

- *Apache Kafka* is a distributed streaming platform that allows applications to publish and subscribe to data streams reliably. We connect external data sources to Kafka using producer APIs. For each data source, we can create a topic in Kafka to allow downstream consumers (Spark Structured Streaming and MongoDB) to access the incoming data.
- *MongoDB* is a document-based distributed database. We connect MongoDB to Kafka as a consumer via the mongodb-kafka connector [46] provided by MongoDB. Incoming data records from a Kafka topic (i.e., an external data source) are persisted in a corresponding MongoDB collection as JSON-like documents for retrospective analysis. Besides incoming data, subscriptions specifying subscribers' interest and other relevant information used for data customization and data analytics are also stored in MongoDB.
- *Apache spark structured streaming* is a scalable stream processing engine built on top of the Spark SQL engine. It supports Dataframe/Dataset APIs for users to express streaming computations the same way one would express a batch computation on static data. We connect Spark Structured Streaming to Kafka as a consumer through the spark-streaming-kafka connector [47] provided by Spark. Incoming data from a Kafka topic is mapped into a data stream in Spark Structured Streaming. One can implement a data channel as a Spark application that runs continuously for producing customized data. Relevant information and subscriptions stored in MongoDB can be loaded into Structured Streaming as DataFrames through a mongodb-spark connector [48] provided by MongoDB.
- **Amazon SNS** is a notification service provided in Amazon Web Services for delivering messages to subscribed endpoints or clients. It allows users to create Amazon SNS topics and publish notifications through APIs. Other systems and end-users can subscribe to these topics and receive published data. Amazon SNS provides filter policies in subscriptions to allow subscribers to filter notifications by their content. We can use the filter policy to send notifications to certain channel subscribers by using their subscription IDs as the filter value. We map a data channel to an Amazon SNS topic, and whoever subscribes to this data channel also becomes a

subscriber to the Amazon SNS topic with its subscription ID as the filter attribute. Customized data generated by the Spark channel application is published to the Amazon SNS topic with subscription IDs attached.

Due to its glued nature, the GOOD system needs "cooperation" between different components to provide BAD services. Taking the new nearby hateful tweet example described in Sect. 5 (the equivalent BAD channel defined in Fig. 24), one would have to complete the following steps for providing the channel service in the GOOD system:

- Configure and deploy Apache Kafka to the cluster. Create adaptor programs as Kafka producers that publish data into Kafka topics for tweets and for officer location updates separately.
- Configure and deploy MongoDB to the cluster. Create collections for tweets, location updates, and subscriptions, and make sure all collections are sharded across the cluster.
- Create and configure an Amazon SNS topic on Amazon Web Services for sending notifications.
- Configure and deploy Apache Spark to the cluster. Create a Spark application as a data channel and connect it to Kafka, MongoDB, and Amazon SNS separately. Implement data customization by joining tweets, officer locations, and subscriptions using stream processing operations.
- Deploy the channel application onto the Spark cluster and make sure all services are running and connected.
- For each newly subscribed subscriber, we add the subscription information into MongoDB for data customization, and we also create a corresponding Amazon SNS subscription with the subscription ID as the filter attribute.

Compared with the BAD system, the GOOD system requires a significant amount of effort from developers to configure, orchestrate, and manage different components for providing BAD services. Besides the administration complexity, due to the limitation of the components in the GOOD system, not all of the query semantics provided by the BAD system can be conveniently supported by the GOOD system.

6.3 GOOD vs. BAD

As we have mentioned, streaming engines have to age historical data out to restrain their resource usage. This limits the query semantics that can be supported by the GOOD system. Consider the new nearby hateful tweets channel defined in Fig. 24, where we send new nearby hateful tweets to in-field officers based on their last known location by utilizing an UPSERT feed. That channel can produce notifications for a temporarily "offline" officer and later send these "missed" notifications to him/her when the officer reconnects, as discussed in Sect. 5.4.

In the GOOD system, if an officer has not sent location updates for a some time, his/her location information would be aged out by the streaming engine. Due to this limitation, a GOOD user can only look for location updates back to a limited time
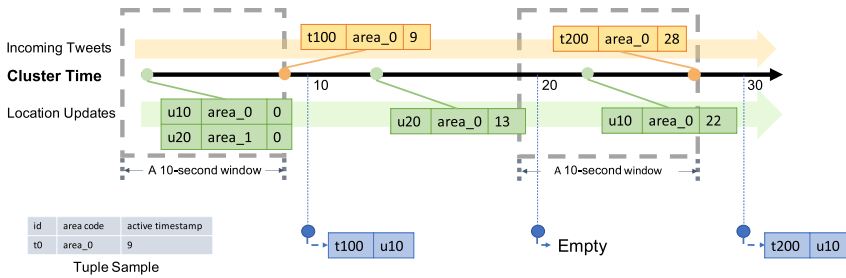
Fig. 35 A GOOD example of sending hateful tweets to officers

for a new incoming tweet. To better approximate the BAD channel, one could consider persisting all historical location updates in MongoDB and pulling the latest locations into Spark Structured Streaming in each channel execution. However, this would lose the timeliness of streaming data and introduce additional data access overhead.

To illustrate the query semantics of the GOOD system and compare that with BAD, we show an alternative new nearby hateful tweets use case. As Spark Structured Streaming does not support spatial joins on data streams, we use "area_code" to represent tweets' and officers' locations. We consider a tweet to be nearby to an officer if it is posted from the same area code as the officer. In this modified use case, we send a new hateful tweet to the nearby in-field officers who have recently (within 10 seconds) updated their locations. This use case will also be used in the later performance comparison between BAD and GOOD. An illustrative example of the modified channel execution using the data sample in Sect. 5.4 is shown in Fig. 35.

When t100 arrives at 9s, we examine the location updates in the past 10 s and find two officers u10 and u20 who recently updated their locations. We check the area codes of t100, u10, and u20 and produce a notification for u10. When t200 arrives at 28s, we look back in a 10-second window and find the location update from u10 at 22s, so we produce a notification for u10. Note that the location update from u20 at 13s is not "used". When t200 come at 28s, this location update of u20 is too old for the the tweet.[8]

7 Experimental results

In this section, we present a set of experiments conducted to evaluate the performance of the BAD system. We focused on the performance of BAD-CQ and compared that with the GOOD system described in Sect. 6.3. We first examined the basic ingestion and query performance of active datasets. Then, we investigated BAD-CQ's continuous channel performance regarding supportable subscribers in

---

[8] For illustrative simplicity, here we only look for location updates for new tweets. One may consider to look for tweets for new location updates and notify u20 about t100, but t200 for u20 would still be missing. Increasing the window size would work for this example but couldn't be applied for general cases.
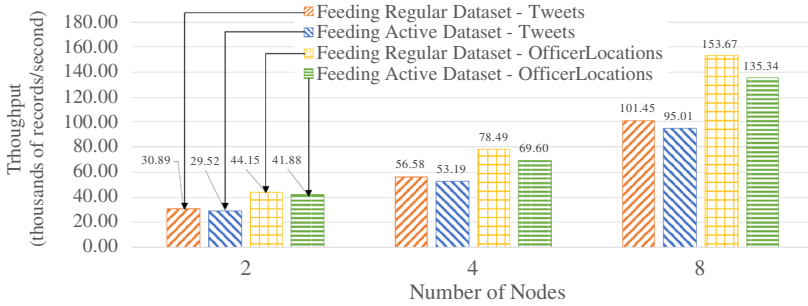
Fig. 36 Ingestion performance on active datasets
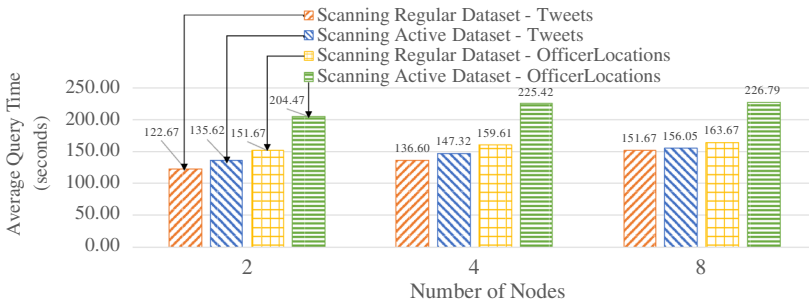
Fig. 37 Query performance on active datasets

different use cases. Also, we compared the performance of the GOOD and the BAD systems using the same use cases. Finally, we investigated the speed-up and scale-out performance of BAD-CQ when it is given more resources. Our experiments were conducted on a cluster connected using a Gigabit Ethernet switch (up to 16 nodes). Each node had a Dual-Core AMD Opteron Processor 2212 2.0 GHz, 8 GB of RAM, and a 900 GB hard disk drive.

7.1 Active dataset scale-out performance

Since active datasets store active timestamps with records for continuous channel evaluation, writing and reading active datasets will have the same additional cost due to the additional bytes. In order to examine the performance impact of that, we conducted ingestion and query performance experiments with active datasets. We used two types of data: the Tweets and OfficerLocations defined in Figs. 2 and 3 respectively. Each tweet was around 140 bytes, and each user location record was around 60 bytes. An active timestamp was 9 bytes long (1 byte for data type and 8 bytes for epoch time). For both scale-out experiments, we started with 100 million records on a 2-node cluster and increased that to 400 million records on

Fig. 38 Datatype and dataset
definition for Schools

```
CREATE TYPE School AS OPEN {
  sid: int,
  area_code: string,
  name: string
};
CREATE DATASET Schools(School) PRIMARY KEY sid;
```

```
CREATE CONTINUOUS CHANNEL NewLocalHatefulTweets(area_code)
  PERIOD duration("PT10S") {
    SELECT t FROM Tweets t
    WHERE t.area_code = area_code AND is_new(t)
};
```

**Fig. 39** A continuous channel for new local hateful tweets

```
CREATE CONTINUOUS CHANNEL NewLocalHatefulTweetsWithSchools(area_code)
 PERIOD duration("PT10S") {
  SELECT t,
  (SELECT VALUE s FROM Schools s WHERE s.area_code = t.area_code)
    AS nearby_schools
  FROM Tweets t
  WHERE t.area_code = area_code AND is_new(t)
};
```

Fig. 40 A continuous channel for new local hateful tweets with schools

a 8-node cluster. For the ingestion performance experiments, we measured the ingestion throughput. For the query performance experiments, we measured the average time over 50 query executions for scanning all records in a dataset. The results are shown in Figs. 36 and 37 respectively.

When ingesting data into active datasets, the additional work comes from attaching active timestamps to incoming data records and persisting them into the storage engine. As we can see from Fig. 36, the ingestion throughput on both the Tweets and OfficerLocations datasets have some regression compared with the regular datasets. The throughput regression is proportional to the size ratio between an incoming record and the active timestamp. When an incoming record is big, the performance impact is relatively small and vice versa. With more nodes in the cluster, the throughput increases since more resources (CPU and storage bandwidth) can be used for parsing and storing incoming data.

When scanning active datatasets, as shown in Fig. 37, the query time increases due to the additional cost of reading the larger records with active timestamps from disk. Similarly, the query time increase is proportional to the size ratio between a stored record and the active timestamp. As the cluster size grows, the query time increases slightly due to the increased query execution cost on a larger cluster, but overall it remains stable since AsterixDB shards its stored data across all nodes.
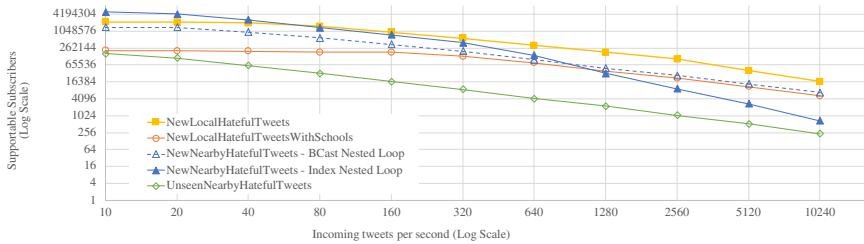
Fig. 41 Maximum number of supportable subscribers under different incoming data rates

7.2 Channel performance

As a channel runs periodically at a user specified period, it requires the channel evaluation to finish within that given period of time. The channel execution time depends on the channel query complexity and the size of the data involved (e.g., the number of tweets and subscribers). In order to examine the performance of data channels, we measured the maximum number of subscribers that can be supported by a channel within a given period. For these use cases, we introduce a new data-set **Schools**, defined in Fig. 38, to store schools' information as relevant auxiliary information. A list of schools can be attached to hateful tweets to provide additional information for use by the responding in-field officers. The Schools dataset contains 10,000,000 records, and each record is around 70 bytes. We used the following four use cases to examine channel performance:

1. *NewLocalHatefulTweets* Send me new hateful tweets from a certain area (defined in Figure 39).
2. *NewLocalHatefulTweetsWithSchools* Send me new hateful tweets from a certain area together with information about schools in that area (defined in Figure 40).
3. *NewNearbyHatefulTweets* Send me new hateful tweets nearby (defined in Figure 24).
4. *UnseenNearbyHatefulTweets* Send me nearby hateful tweets that I've not seen before (defined in Figure 26).

In use cases 1 and 2, subscribers subscribe to a channel with their interested area codes. In use cases 3 and 4, subscribers subscribe with their officer IDs and their locations are spatial data mapped to IDs. All channels were configured to execute every 10 seconds. To approximate incoming data in practice, we set up external programs to continuously send tweets and officer location updates into the data feeds we created in Figs. 4 and 5. For tweets, the client program sent them at a configurable rate (tweets / second), and 10% of the incoming tweets were hateful. For location updates, we created client programs to simulate the movement of subscribers (in-field officers) on the map and to send location updates on behalf all of them, and an average of 1/3 of the in-field officers updated their locations every 10 s. Both programs ran on machines outside of the BAD cluster.

In all four use cases, we fixed the incoming tweet rate and searched for the maximum number of supportable subscribers in the given 10-s channel execution period while both tweets and location updates were coming. We varied the incoming tweet rate to see how channel performance changed. For the "NewNearbyHatefulTweets" channel in particular, we chose two algorithms (broadcast nested loop join and index nested loop join) to evaluate the spatial join between the incoming tweets and officers' locations. (We broadcast data from the Tweets dataset and utilized the R-Tree index on the location attribute of the OfficerLocations dataset.) We deployed BAD-CQ on a 6-node cluster and the performance results are shown in Fig. 41. (Note the use of a log scale for the y-axis.)

Depending on the channel query complexity, the maximum number of supportable subscribers varies. For all four use cases, the maximum number of supportable subscribers decreases as the incoming tweet rate increases; this is due to the increased cost of producing and persisting [9] more customized notifications. Comparing the results for "NewLocationHatefulTweets" and "NewLocalHatefulTweets-WithSchools", we see that the latter one has lower performance, as adding in school information incurs more computational and persistence cost. Comparing "NewNearbyHatefulTweets - BCast Nested Loop" and "NewNearbyHatefulTweets - Index Nested Loop", we see that the use of the index offers much better performance than scanning the whole OfficerLocations dataset when the incoming tweet rate is low. As the incoming tweet rate grows, however, the performance of the index nested loop join becomes worse than the broadcast join. The reason is that, with more incoming tweets, the maximum number of supportable subscribers decreases due to the increased cost of computing customized data. For the join operation between tweets and officer locations, then, having more tweets and fewer actual subscribers (in-field officers) increases the query's selectivity for OfficerLocations. Since the index nested loop join accesses the primary index through a secondary index, when the selectivity becomes high, the performance of using that index becomes worse than just scanning the primary dataset. Interested readers may refer to [41] for a more detailed analysis of the underlying storage engine's performance benchmarks.

7.3 GOOD vs. BAD performance

The BAD system enables developers to create BAD services with declarative statements. The GOOD system, in contrast, requires developers to manually glue multiple systems together and orchestrate them programmatically to create BAD services. In order to show that the BAD system not only alleviates developers' effort when creating BAD services, but can also provide better performance compared with a GOOD system, we chose several use cases supported by both the BAD and GOOD systems and measured their performance on both. As the BAD system aim to provide an out-of-box solution for meeting all the BAD requirements outlined in Sect. 3, we compare the end-to-end performance of the BAD system and the GOOD system.

---

[9] As mentioned in Sect. 4.2.4, BAD persists customized data to disk by default to allow brokers to pull later.
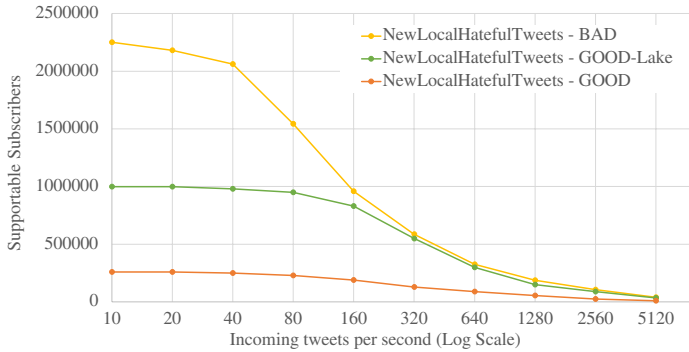
Fig. 42 BAD-CQ vs. GOOD on *NewLocalHatefulTweets*
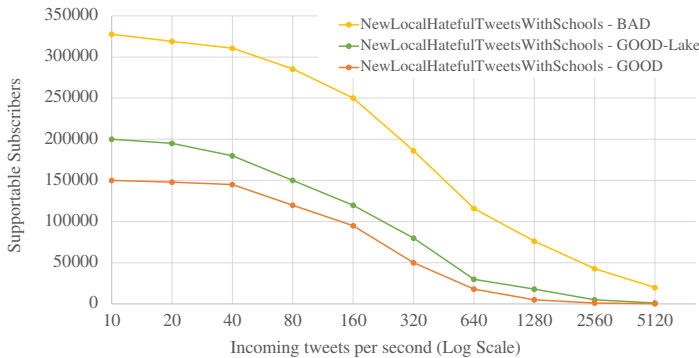
Fig. 43 BAD-CQ vs. GOOD on *NewLocalHatefulTweetsWithSchools*

We used the GOOD system detailed in Sect. 6.2 for these experiments. For comparison purposes, we also measured the performance of an altered GOOD system by replacing MongoDB with Delta Lake. As we discussed in Sect. 6.3, the GOOD system cannot provide all query semantics supported in the BAD system. Not all use cases in Section 7.2 can be supported directly in the GOOD system. Spark Structured Streaming does not support spatial joins between streams, so here we use area code to represent the location of tweets and officers. The use cases used for comparing the performance of the BAD system and the GOOD system are as follows:

1. *NewLocalHatefulTweets* Send me new hateful tweets from a certain area (same as Sect. 7.2).
2. *NewLocalHatefulTweetsWithSchools*: Send me new hateful tweets from a certain area together with the schools in that area (same as Sect. 7.2).
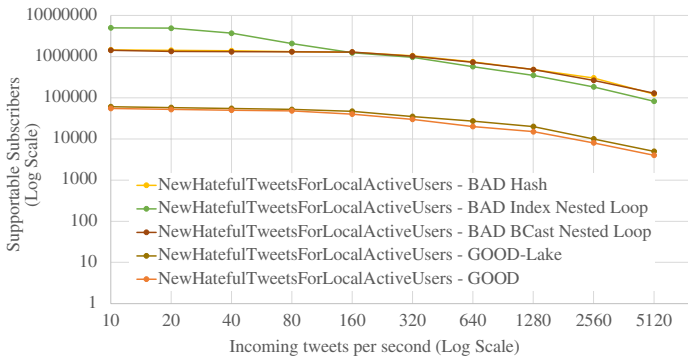
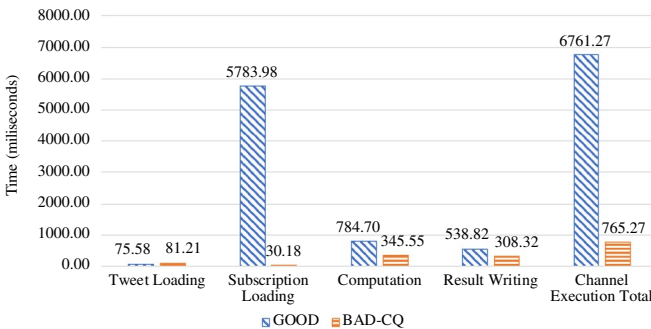**Fig. 44** BAD-CQ vs. GOOD on *NewHatefulTweetsForLocalActiveUsers*

Fig. 45 Cost of "NewLocalHatefulTweets" with 150,000 subscribers and 80 tweets/second on both the GOOD and BAD system

3. ***NewHatefulTweetsForLocalActiveUsers*** Send me new hateful tweets from the same area as my current location (similar to NewNearbyHatefulTweets in Sect. 7.2, but modified to use area_code for this experiment).

In use cases 1 and 2, subscribers subscribe to a channel with the area codes of interest. In use case 3, subscribers subscribe to the channel with the their officer IDs. Due to the high overhead of integrating Spark Structured Streaming with MongoDB, we tuned down the size of the **Schools** dataset by 5x to 2,000,000. To demonstrate the advantages that the BAD system as of utilizing indexes and different query evaluation algorithms, we picked the "NewHatefulTweetsForLocalActiveUsers" use case, and we experimented with hash join, broadcast nested loop join, and index nested loop join. In this experiment, we focused on the processing core of both systems without including result delivery using brokers. The generated notifications were persisted in storage, as in the default pull mode. All incoming data was persisted as well for retrospective analysis. The performance results in terms of the number of supportable subscribers are shown in Figs. 42, 43, and 44 respectively.
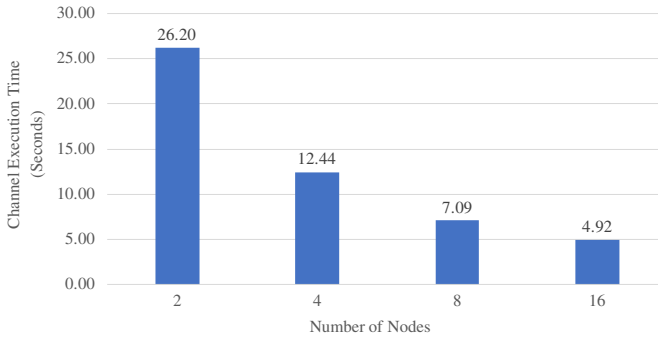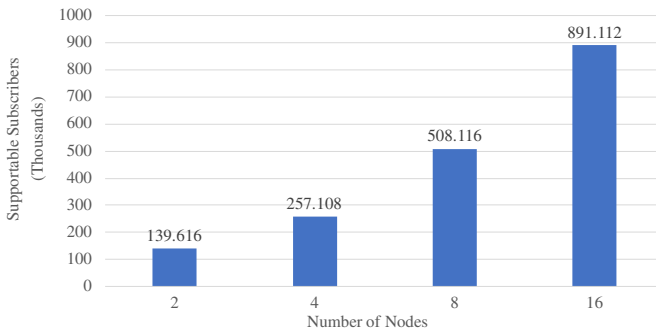
Fig. 46 Speed-up BAD-CQ with fixed incoming tweet rate and number of subscribers

In all three cases, BAD-CQ outperforms both versions of the GOOD system. The GOOD system on Delta Lake (GOOD-Lake) is slightly better than that on MongoDB (GOOD), as the cost of shipping data across systems (serailization, deserialization, etc.) is much less, but it's still not comparable to the BAD system. As the incoming tweet rate grows, the performance of both systems drop because of the increased cost of producing and persisting more notifications. Similar to Sect. 7.2, both systems have better performance for "NewLocalHatefulTweets" (colored in orange) than for "NewLocalHatefulTweetsWithSchools" (colored in blue) due to the additional cost of attaching relevant school information. In particular, the "NewLocalHatefulTweetsWithSchools" use case for GOOD suffers more from the increased incoming tweet rate, as the cost of persisting notifications with schools becomes high when there are many notifications. For "NewHatefulTweetsForLocalActiveUsers", the GOOD system on Delta Lake (GOOD-Lake) has a limited advantage compared to that on MongoDB (GOOD) because the computation cost dominated the maximum number supported subscribers, which leads to many fewer subscriptions being read and fewer notifications being persisted. For the BAD system, we see a similar performance benefit for utilizing an index and the same performance regression when the incoming tweet rate becomes high. Hash join offered only a slight advantage over a broadcast nested loop join in this case, as the total number of tweets for each channel execution is relatively small.

In order to better understand the cost of the GOOD system, we chose the "NewLocalHatefulTweets" use case with 150,000 subscribers and 80 tweets/second and measured the time consumed by each stage of its channel execution on both the GOOD and BAD system. The result is shown in Fig. 45, which also includes the overall channel execution time. As can be seen, the GOOD channel execution spent much of its time loading Subscriptions from MongoDB. This is a consequence of the overhead of gluing different systems together, as shipping data from one subsystem to another incurs additional serialization/deserialization and data transformation and transmission costs. One could consider maintaining copies of the relevant data and subscriptions in Spark Structured Streaming as well, to accelerate the processing, but then developers would have to handle consistency challenges and need

Fig. 47 Scale-out BAD-CQ with fixed incoming tweet rate
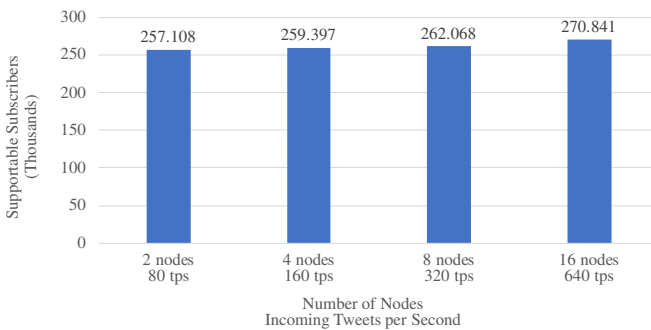
Fig. 48 Scale-out BAD-CQ with increasing incoming tweet rate

to migrate updates back and forth between Spark Structured Streaming and MongoDB. In contrast, BAD-CQ spent much less time on subscription loading. Since tweets were being ingested at the same time, there was a bit of read/write contention on the *Tweets* dataset that caused the tweet loading time to be higher than the subscription loading time on BAD-CQ.

7.4 BAD scalability

Finally, we investigated the scalabilty of BAD-CQ from two angles: *speed-up* - given a fixed workload, see if the performance improves with more resources, and *scale-out* - increase the workload together with available resources to see if the performance remains stable. We chose the "NewNearbyHatefulTweets - Bcast Nested Loop" channel and increased the channel's period to 30 seconds for this experiment. All other settings were the same as Sect. 7.2.

Speed-up experiments The channel workload is determined by the incoming tweets per second and the number of subscribers (in-field officers). In this experiment, we fixed the incoming tweet rate to 160 tweets per second and had 140,000 subscribers. We increased the cluster size from 2 nodes to 4, 8, and 16 nodes, and

we measured the channel execution times, as shown in Fig. 46. When the cluster grows, the channel execution time is almost halved because the subscribers' locations are stored on twice as many machines. Since each node now has less data, the join between incoming tweets and officer's locations, which computes on all nodes, can finish sooner. As tweets are broadcast to all nodes in the cluster and the execution overhead also grows with the cluster size, the speed-up gain gradually decreases with larger cluster sizes.

Scale-out experiments We used two experiments to evaluate the scale-out performance of BAD-CQ. We first fixed the incoming tweet rate to 160 tweets/second and increased the cluster size from 2 nodes to 4, 8, and 16 nodes to see how many subscribers could be supported in each setting. The result is shown in Fig. 47. As we double the size of the cluster, the maximum number of supportable subscribers almost doubles. Similar to the speed-up experiment, twice many nodes allow the join operation to handle more data in the given time period.

In the second experiment, we increased the incoming tweet rate together with the cluster size. We started with a 2-node cluster with 80 incoming tweets per second, and we increased the cluster size and the incoming tweet rate by the factor of two, up to 16 nodes and 640 tweets per second. The result is shown in Fig. 48. The channel performance maintains relatively stable as we increase the workload and add more resources at the same time.

8 Conclusions and future work

In this work, we considered a world where Big Data is no longer just bytes sitting on storage devices, waiting to be analyzed, but is valuable information surrounded by active requests asking for continual "news updates". In such a Big Active Data (BAD) world, developers often need to create and manage data services to support analysts in working with declarative queries and subscribers looking for the latest updates. In order to reduce the effort for developers creating BAD services, we have built the BAD system, consisting of BAD-RQ, which "activates" a parameterized query as a data channel for subscribers to receive periodic query results of interest, and BAD-CQ, which introduces continuous (incremental query) semantics into data channels and optimizes the channel infrastructure for continuous use cases. We showed the user model, design, and implementation of our system and illustrated how developers can use it to create BAD services declaratively. To demonstrate the complexity of creating BAD services without BAD, we also presented a "GOOD" system created by gluing multiple Big Data systems together. We examined the performance of the BAD system under different workloads and compared that with an instance of a GOOD system. The results for the use cases examined showed that the BAD system could support up to four millions subscribers on a six-node cluster, was able to horizontally scale out with more resources, and offered significantly better performance as compared with the GOOD system. In all, the BAD system provides a systematic solution for creating BAD services at scale.

This work leads to a number of interesting opportunities for future investigation:

- *Connecting multiple BAD systems* In a BAD world, there could be multiple BAD systems running and managed by different organizations. In some use cases, developers may need to share information between different organizations and combine it with local data to create applications. Building a scalable data sharing service from scratch requires a lot of work. With BAD, we could allow developers to connect multiple BAD systems via data channels and feeds. Developers of multi-site applications could then benefit from the BAD approach and could create data sharing services with very little implementation and management overhead.

- *Exploiting shared computation among data channels* In the current BAD system, data channel queries are processed, compiled, and optimized independently. While shared computation arises from evaluating the parameterized requests within a given channel together, more exploitation of sharing is possible.

    Similar to [6], we could analyze multiple data channel queries, split them into smaller parts, discover shared computations, and reuse intermediate results to improve channel performance by avoiding redundant computation.

- *Fault-tolerant channel evaluation* As the BAD system contains a *Persistent Storage* subsystem, which provides ACID transaction support, all data stored in the BAD system are persistently stored and can be recovered when failure happens. Despite that, when a failure happens during a channel execution, the run-time data will be lost and the channel execution has to restart from scratch when the system come back online. Following the shared computation work discussed in the previous bullet, we could persist some of the intermediate results depending on the computational complexity and the evaluation footprint, so when a failure happens, the channel evaluation can resume from a previously persisted state to reduce the recovery time.

- *Resource management & scheduling of channel executions* Currently, every channel execution is scheduled based on its period. Each channel execution runs as an independent job in the analytical engine, and an internal resource manager manages the resource usage of all jobs running in the system. When there is resource contention, certain channel executions may be delayed and cause a channel to terminate (as we require channel executions to finish within the given period to meet the channel's time requirement). Given different channel periods and users' quality of service requirements, it should be possible to develop a smarter scheduling strategy in which we allow more flexible channel execution schedules based on the available resources and obtain better resource utilization at the same time.

- *Scalable experiments on cloud infrastructure* With the popularity of the cloud, more and more systems are offering out-of-the-box services with databases built on top of different cloud infrastructures. Such deployments enable users to quickly set up the environment and scale out or scale down based on their loads accordingly. We could deploy the BAD system onto a cloud environment, investigate its performance when given a large number of virtual nodes, and explore how to offer the system as a service to different users in a cloud environment with further optimizations.

References

1. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop distributed file system. In: Khatib, M.G., He, X., Factor, M. (eds.) IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, pp. 1–10. Lake Tahoe, Nevada, USA, 3–7 May (2010). https://doi.org/10.1109/MSST.2010.5496972

2. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: Wang, J.T. (ed.) Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, pp. 1099–1110. Vancouver, BC, Canada, 10–12 June (2008). https://doi.org/10.1145/1376616.1376726

3. Thusoo, A., Sarma, J.S., Jain, N., et al.: Hive: a warehousing solution over a map-reduce framework. PVLDB **2**(2), 1626–1629 (2009). https://doi.org/10.14778/1687553.1687609

4. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Gribble, S.D., Katabi, D. (eds.) Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, , pp. 15–28. San Jose, CA, USA, 25–27 Apr (2012)

5. Terry, D.B., Goldberg, D., Nichols, D.A., Oki, B.M.: Continuous queries over append-only databases. In: Stonebraker, M. (ed.) Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, pp. 321–330. San Diego, California, USA, 2–5 June (1992). https://doi.org/10.1145/130283.130333

6. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: A scalable continuous query system for internet databases. In: Chen, W., Naughton, J.F., Bernstein, P.A. (eds.) Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pp. 379–390. Dallas, TX, USA, 16–18 May (2000). https://doi.org/10.1145/342009.335432

7. Widom, J., Ceri, S. (eds.): Active Database Systems: Triggers and Rules For Advanced Database Processing (1996)

8. Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M.J., Schelter, S., Höger, M., Tzoumas, K., Warneke, D.: The stratosphere platform for big data analytics. VLDB J. **23**(6), 939–964 (2014). https://doi.org/10.1007/s00778-014-0357-y

9. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: a distributed messaging system for log processing. In: Proceedings of the NetDB, pp. 1–7 (2011)

10. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: fault-tolerant streaming computation at scale. In: Kaminsky, M., Dahlin, M. (eds.) ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, pp. 423–438. Farmington, PA, USA, 3–6 Nov (2013). https://doi.org/10.1145/2517349.2522737

11. Eugster, P.T., Felber, P., Guerraoui, R., et al.: The many faces of publish/subscribe. ACM Comput. Surv. **35**(2), 114–131 (2003). https://doi.org/10.1145/857076.857078

12. Alsubaiee, S., Altowim, Y., Altwaijry, H., et al.: Asterixdb: a scalable, open source BDMS. PVLDB **7**(14), 1905–1916 (2014). https://doi.org/10.14778/2733085.2733096

13. Jacobs, S., Wang, X., Carey, M.J., Tsotras, V.J., Uddin, M.Y.S.: Bad to the bone: big active data at its core. VLDB J. **29**, 1337 (2020)

14. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: Stream: the stanford data stream management system. Technical Report 2004-2020, Stanford InfoLab (2004)

15. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB J. **15**(2), 121–142 (2006). https://doi.org/10.1007/s00778-004-0147-z

16. Krishnamurthy, S., Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Madden, S., Reiss, F., Shah, M.A.: TelegraphCQ: an architectural status report. IEEE Data Eng. Bull. **26**(1), 11–18 (2003)

17. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., Ryaboy, D.V.: Storm@twitter. In: Dyreson, C.E., Li, F., Özsu, M.T. (eds.) International Conference on Management of Data, SIGMOD 2014, pp. 147–156. Snowbird, UT, USA, 22–27 June (2014). https://doi.org/10.1145/2588555.2595641

18. Armbrust, M., Das, T., Torres, J., Yavuz, B., Zhu, S., Xin, R., Ghodsi, A., Stoica, I., Zaharia, M.: Structured streaming: a declarative API for real-time applications in apache spark. In: Das, G., Jermaine, C.M., Bernstein, P.A. (eds.) Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, pp. 601–613. Houston, TX, USA, 10–15 June (2018). https://doi.org/10.1145/3183713.3190664

19. Kafka, A.: Kafka streams. https://kafka.apache.org/documentation/streams/ (2020). Accessed 8 May 2020

20. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache Flink™: stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015)

21. Castro, M., Druschel, P., Kermarrec, A., Rowstron, A.I.T.: Scribe: a large-scale and decentralized application-level multicast infrastructure. IEEE J. Sel. Areas Commun. **20**(8), 1489–1499 (2002). https://doi.org/10.1109/JSAC.2002.803069

22. Chockler, G.V., Melamed, R., Tock, Y., Vitenberg, R.: Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication. In: Jacobsen, H., Mühl, G., Jaeger, M.A. (eds.) Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems, DEBS 2007, pp. 14–25. Toronto, Ontario, Canada, 20–22 June, 2007. ACM International Conference Proceeding Series, vol. 233 (2007). https://doi.org/10.1145/1266894.1266899

23. Girdzijauskas, S., Chockler, G.V., Vigfusson, Y., Tock, Y., Melamed, R.: Magnet: practical subscription clustering for internet-scale publish/subscribe. In: Bacon, J., Pietzuch, P.R., Sventek, J., Çetintemel, U. (eds.) Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, pp. 172–183. Cambridge, UK, 12–15 July (2010). https://doi.org/10.1145/1827418.1827456

24. Setty, V., van Steen, M., Vitenberg, R., Voulgaris, S.: Poldercast: Fast, robust, and scalable architecture for P2P topic-based pub/sub. In: Narasimhan, P., Triantafillou, P. (eds.) Middleware 2012 - ACM/IFIP/USENIX 13th International Middleware Conference, Montreal, QC, Canada, 3–7 Dec 2012. Proceedings. Lecture Notes in Computer Science, vol. 7662, pp. 271–291 (2012). https://doi.org/10.1007/978-3-642-35170-9_14

25. Altinel, M., Franklin, M.J.: Efficient filtering of XML documents for selective dissemination of information. In: Abbadi, A.E., Brodie, M.L., Chakravarthy, S., Dayal, U., Kamel, N., Schlageter, G., Whang, K. (eds.) VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, pp. 53–64, Cairo, Egypt. 10–14 Sept (2000)

26. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and evaluation of a wide-area event notification service. ACM Trans. Comput. Syst. **19**(3), 332–383 (2001). https://doi.org/10.1145/380749.380767

27. Diao, Y., Fischer, P.M., Franklin, M.J., To, R.: Yfilter: Efficient and scalable filtering of XML documents. In: Agrawal, R., Dittrich, K.R. (eds.) Proceedings of the 18th International Conference on Data Engineering, pp. 341–342. San Jose, CA, USA, 26 Feb–1 Mar (2002). https://doi.org/10.1109/ICDE.2002.994748

28. Moro, M.M., Bakalov, P., Tsotras, V.J.: Early profile pruning on XML-aware publish-subscribe systems. In: Proceedings of the 33rd international conference on very large data bases (VLDB '07). VLDB Endowment, pp. 866–877 (2007).

29. Li, M., Ye, F., Kim, M., Chen, H., Lei, H.: A scalable and elastic publish/subscribe service. In: 25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, pp.

1254–1265. Anchorage, Alaska, USA, 16–20 May 2011—Conference Proceedings (2011). https://doi.org/10.1109/IPDPS.2011.119

30. Eugster, P.: Type-based publish/subscribe: concepts and experiences. ACM Trans. Program. Lang. Syst. **29**(1), 6 (2007). https://doi.org/10.1145/1180475.1180481

31. Idreos, S., Koubarakis, M., Tryfonopoulos, C.: P2P-DIET: an extensible P2P service that unifies ad-hoc and continuous querying in super-peer networks. In: Weikum, G., König, A.C., Deßloch, S. (eds.) Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 933–934. Paris, France, 13–18 June (2004). https://doi.org/10.1145/1007568.1007704

32. Carey, M.J., Jacobs, S., Tsotras, V.J.: Breaking BAD: a data serving vision for big active data. In: Gal, A., Weidlich, M., Kalogeraki, V., Venkasubramanian, N. (eds.) Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, pp. 181–186. Irvine, CA, USA, 20–24 June (2016). https://doi.org/10.1145/2933267.2933313

33. Jacobs, S., Uddin, M.Y.S., Carey, M.J., et al.: A BAD demonstration: towards big active data. PVLDB **10**(12), 1941–1944 (2017). https://doi.org/10.14778/3137765.3137814

34. Uddin, M.Y.S., Venkatasubramanian, N.: Edge caching for enriched notifications delivery in big active data. In: 38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, pp. 696–705. Vienna, Austria, 2–6 July (2018). https://doi.org/10.1109/ICDCS.2018.00073

35. Wang, X., Carey, M.J.: An IDEA: an ingestion framework for data enrichment in AsterixDB. PVLDB **12**(11), 1485–1498 (2019). https://doi.org/10.14778/3342263.3342628

36. Grover, R., Carey, M.J.: Data ingestion in AsterixDB. In: Alonso, G., Geerts, F., Popa, L., Barceló, P., Teubner, J., Ugarte, M., den Bussche, J.V., Paredaens, J. (eds.) Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, pp. 605–616. Brussels, Belgium, 23–27 Mar (2015). https://doi.org/10.5441/002/edbt.2015.61

37. Chamberlin, D.: SQL++ for SQL users: a tutorial. Couchbase, Inc. (2018). (Available at Amazon.com)

38. Ong, K.W., Papakonstantinou, Y., Vernoux, R.: The SQL++ query language: configurable, unifying and semi-structured (2014). http://arxiv.org/abs/1405.3631

39. Nguyen, H., Uddin, M.Y.S., Venkatasubramanian, N.: Multistage adaptive load balancing for big active data publish subscribe systems. In: Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, DEBS 2019, pp. 43–54. Darmstadt, Germany, 24–28 June (2019). https://doi.org/10.1145/3328905.3329508

40. Snodgrass, R.T., Ahn, I.: Temporal databases. IEEE Comput. **19**(9), 35–42 (1986). https://doi.org/10.1109/MC.1986.1663327

41. Luo, C., Carey, M.J.: Efficient data ingestion and query processing for LSM-based storage systems. PVLDB **12**(5), 531–543 (2019). https://doi.org/10.14778/3303753.3303759

42. Alsubaiee, S., Carey, M.J., Li, C.: LSM-based storage and indexing: An old idea with timely benefits. In: Mouratidis, K., Renz, M., Emrich, T., Züfle, A., Janowicz, K. (eds.) Second International ACM Workshop on Managing and Mining Enriched Geo-Spatial Data, GeoRich@SIGMOD 2015, pp. 1–6. Melbourne, VIC, Australia, 31 May (2015). https://doi.org/10.1145/2786006.2786007

43. Alsubaiee, S., Behm, A., Borkar, V.R., Heilbron, Z., Kim, Y., Carey, M.J., Dreseler, M., Li, C.: Storage management in asterixdb. Proc. VLDB Endow. **7**(10), 841–852 (2014). https://doi.org/10.14778/2732951.2732958

44. Spark, A.: Structured streaming programming guide. http://spark.apache.org/docs/latest/structured-streaming-programming-guide.html (2020). Accessed 13 Apr 2020

45. ...Armbrust, M., Das, T., Paranjpye, S., Xin, R., Zhu, S., Ghodsi, A., Yavuz, B., Murthy, M., Torres, J., Sun, L., Boncz, P.A., Mokhtar, M., Hovell, H.V., Ionescu, A., Luszczak, A., Switakowski, M., Ueshin, T., Li, X., Szafranski, M., Senster, P., Zaharia, M.: Delta lake: high-performance ACID table storage over cloud object stores. Proc. VLDB Endow. **13**(12), 3411–3424 (2020). https://doi.org/10.14778/3415478.3415560

46. MongoDB: MongoDB Kafka Connector. https://docs.mongodb.com/kafka-connector/current/ (2020). Accessed 8 May 2020

47. Spark, A.: Spark Streaming + Kafka Integration Guide. https://spark.apache.org/docs/latest/streaming-kafka-integration.html (2020). Accessed 8 May 2020

48. MongoDB: MongoDB Connector for Spark. https://docs.mongodb.com/spark-connector/master/ (2020). Accessed 8 May 2020

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.