Optimizing Big Active Data Management Systems*

Shahrzad Haji Amin Shirazi¹, Xikui Wang², Michael J. Carey² and Vassilis J. Tsotras¹

¹University of California, Riverside ²University of California, Irvine

Abstract

Within the dynamic world of Big Data, traditional systems typically operate in a passive mode, processing and responding to user queries by returning the requested data. However, this methodology falls short of meeting the evolving demands of users who not only wish to analyze data but also to receive proactive updates on topics of interest. To bridge this gap, Big Active Data (BAD) frameworks have been proposed to support extensive data subscriptions and analytics for millions of subscribers. As data volumes and the number of interested users continue to increase, it is imperative to optimize BAD systems for enhanced scalability, performance, and efficiency. To this end, this paper introduces three main optimizations, namely: strategic aggregation, intelligent modifications to the query plan, and early result filtering, all aimed at reinforcing a BAD platform's capability to actively manage and efficiently process soaring rates of incoming data and distribute notifications to larger numbers of subscribers.

Keywords

big active data, scalable query processing, aggregation, optimization

1. Introduction

In today's fast-paced digital world, we are inundated with a tremendous amount of data every second. Managing and analyzing this ocean of data, widely known as Big Data, presents formidable challenges and numerous systems have been developed for addressing them. However, the majority of these systems operate in a *passive* mode, merely processing and returning data in response to user queries. This passive approach often falls short for users who not only want to analyze data but also **actively** receive updates on new data items that interest them, explore their **relationships** with other data, and even **enrich** them with additional information existing in different datasets. These demands have led to the creation of Big Active Data (BAD) frameworks [1, 2, 3] that aim to support extensive data subscriptions and analytics for millions of subscribers.

The BAD framework is designed to address several essential needs: (i) It ensures that data is not examined in isolation but rather in the context of related information, enhancing its overall significance. (ii) It enables data enrichment by enhancing newly arriving data with data from existing datasets, allowing for responses that enable personalized and actionable insights. (iii) It supports both real-time processing and retrospective Big Data analytics, facilitating deeper exploration and long-term analysis of stored data. By integrating these capabilities, a BAD framework eliminates the inefficiencies of cobbling together multiple independent systems (each dealing with a part of the needed processing, i.e., accessing Big Data, managing incoming streaming data, matching subscribers to information, etc). As we confront the relentless expansion of data volumes and the burgeoning number of users interested in this data, the challenge to manage ever-larger datasets becomes increasingly acute.

This work focuses on optimizing a BAD platform to better handle soaring rates of incoming data and a growing roster of subscriptions. By deploying a suite of optimization techniques including strategic aggregation, intelligent modifications to the query plan, and early result filtering we have enhanced the BAD framework to its most optimized form yet. These improvements not only streamline data processing without adding more resources, but also significantly broaden the system's ability to serve a larger and more diverse subscriber base, marking a significant stride forward in the ongoing quest to *activate* Big Data.

2. Related Work

Tapestry [4] first introduced Continuous Queries as queries that are issued once and then return results continuously as they become available. Tapestry also defined continuous query semantics and created rewrite rules for transforming user-provided queries into incremental database queries. Subsequent research has primarily concentrated on queries involving streaming data. NiagaraCQ [5] enhanced the scalability and efficiency of continuous queries by breaking them down into smaller, manageable components and clustering similar queries based on their expression signatures. It organized signature constants in a specialized table and used joins to process similar queries as a group. Furthermore, to enhance computational efficiency, the system employed delta files that allow for incremental updates and evaluations of the data. STREAM represents a research prototype that was designed to handle continuous queries across both data streams and persistent storage [6]. It provided a Continuous Query Language (CQL) for constructing continuous queries against streams and updatable relations [7]. Most continuous query projects have struggled with scalability, making them less suitable for Big Data use cases, as they often fail to scale effectively in such environments.

Streaming engines can be used for data processing and data customizing pipelines and to provide real-time analytics. Apache Kafka [8, 9], Apache NiFi [10], Apache Flink [11], and Amazon Kinesis [12] are prominent platforms designed to handle and process large-scale, real-time data streams. Similarly, Azure Stream Analytics [13] and Google Cloud Dataflow [14] are specialized for stream processing and real-time analytics. These systems are optimized for high-throughput and low-latency processing, enabling them to handle vast amounts of data generated in real-time. However, they are not inherently equipped to provide long-term data storage solutions. Instead, these systems typically re-

DOLAP 2025: 27th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data, co-located with EDBT/ICDT 2025, March 25, 2025, Barcelona, Spain

You can use this document as the template for preparing your publication. We recommend using the latest version of the ceurart style.
 Shaji013@ucr.edu (S. H. A. Shirazi); xikuiw@uci.edu (X. Wang); mjcarey@ics.uci.edu (M. J. Carey); tsotras@cs.ucr.edu (V. J. Tsotras)
 0009-0005-9259-4997 (S. H. A. Shirazi); 0000-0002-8540-3436 (X. Wang); 0000-0003-1561-4059 (M. J. Carey); 0000-0001-5462-9451 (V. I. Tsotras)

^{© 2025} Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

quire integration with external storage solutions to persist processed data for later use or analysis. This "glued-systems" approach has been shown to have performance disadvantages in [3].

Traditional publication/subscription (pub/sub) systems [15, 16, 17, 18, 19, 20] allow subscribers to register their interests in events and to be asynchronously notified about events from publishers. Although pub/sub services can handle a large number of subscribers, users often have to integrate such services with other systems for data processing, and complex computations across multiple data sources are not supported.

The systems mentioned above generally face challenges in scaling, persisting data, or handling complex subscription queries, which then requires integrating them with additional systems for effective data processing. The BAD platform, summarized in the next section, was designed and built to address these constraints. In this current work, we focus on further optimizing BAD. One of the key steps involves creating the BAD index, an idea that is conceptually similar to partial indexing [21]; it focuses on indexing only the most frequently accessed or queried segments of a dataset, yet is distinct from partial indexing in its implementation as explained in Section 4.3.

3. BAD Preliminaries

The BAD platform can enable millions of users to subscribe to data of interest and receive updates continuously. It is different than continuous queries, streaming engines and pub/sub systems as it also supports Big Data analytics with a declarative language, SQL++ (a SQL-inspired query language for semi-structured data [22, 23]).

An overview of the BAD platform is shown in Figure 1. BAD has five basic blocks [24]: (i) Data Feeds, which manage the ingestion of rapidly arriving data; (ii) Persistent Storage, responsible for storing the data; and an (iii) Analytical Engine, which enables analytic queries on the stored and incoming data to reveal useful information. Additionally, BAD includes (iv) Data Channels and (v) Brokers, which are important for this paper and are described in detail later in this section using an example application.



Figure 1: An overview of the BAD platform.

There are 3 different types of users for the BAD system; *Subscribers* who subscribe to channels so they can get the data of interest, *Developers* who create the BAD channels, and *Analysts* who run queries on the data stored in the system.

For this work we obtained a copy of the BAD open source platform and have extended it further. The existing implementation of the BAD platform [2, 25, 3] was built as an extension of Apache AsterixDB [26], a Big Data Management System (BDMS) that provides distributed data management for large-scale, semi-structured data. In AsterixDB, data is stored by creating a Datatype, which describes known aspects of the data being stored, and then a Dataset, which is a collection of records with a given datatype. As an example, Figure 2 shows the DDLs that create an open datatype named "EnrichedTweet" and a dataset named "EnrichedTweets" based on that type. As the name implies, this dataset includes enriched tweets that are derived from original tweets [27, 28, 29], augmented with additional fields that contain specific information extrapolated from the text of the tweets (like threatening_rate, weapon_mentioned etc.) The keyword "ACTIVE" used in the DDL for creating the dataset EnrichedTweets enables continuous query semantics.

CREATE TYPE EnrichedTweet AS {

tid:int, text:string, retweet_count:int, threatening_rate:int, hate_speech_rate:int, retweeted_status:string, weapon_mentioned:boolean, drug_activity:string, about_country:string, state:string, location:point, additional_info:string}; CREATE ACTIVE DATASET EnrichedTweets(EnrichedTweet) PRIMARY KEY tid;

Figure 2: DDLs for creating a Datatype and its Dataset.

3.1. A BAD Application Example

Tweets can offer valuable insights into public opinions and behaviors, with many people relying on them to stay informed about important topics. However, given the volume and velocity of tweets generated every second, it can be very challenging to extract useful information from them. Therefore, providing a way for users to focus on tweets that are specifically relevant to their interests is highly beneficial. People may be particularly interested in tweets related to specific topics such as sports, politics, or various types of crime. The BAD platform addresses this need by creating channels that users can subscribe to, delivering relevant data based on their preferences, and leveraging a broker network to ensure that tweets are delivered to subscribers in real time. This system enables different applications to effectively manage the vast flow of information, allowing users to focus on the content that matters most to them.

3.2. Brokers

To facilitate the distribution of results to millions of subscribers, the BAD platform integrates a broker sub-system to handle both subscription communication and result delivery [2]. Brokers can range from individual servers dedicated to relaying custom data to subscribers, to complex networks offering features like load balancing, subscription handover, and varied caching methods. Different brokers can be registered as HTTP endpoints in the BAD platform, and subscribing end users have the flexibility to select a broker that aligns with their specific requirements. Once results are prepared for subscribers, the BAD platform efficiently dispatches the relevant updates to all subscribed individuals (end users) through the designated brokers.

3.3. Data Channels

The BAD platform user model exploits the shared structure among subscriptions and offers it as a service, namely as a *data channel*. Data channels allow developers to activate parameterized queries as services for users to subscribe to and continuously receive their data of interest. In practice, the need for similar information among users would likely result in the creation of comparable queries.

Consider a scenario where users might want the system to "send them threatening tweets that relate to the US, are widely retweeted, and the sender's location is close to their location". The TweetsAboutCrime channel, depicted in Figure 3, allows its subscribers to receive nearby enriched tweets that concern the United States, possess a retweet_count greater than 10,000, and have a threatening_rate exceeding 5 (on a scale of 0 to 10). The term PE-RIOD refers to how often the channel is executed, which will be described later. To ensure that channels only process and deliver newly incoming tweets between executions, the is_new function is employed in the channel definition to exclude already processed tweets and thus implement continuous query semantics. The channel's parameters, such as MyUserName in the TweetsAboutCrime channel, allow the system to personalize results for each subscriber. Users can subscribe to channels of interest by providing parameters through DDL commands, as demonstrated in Figure 4. In this example, the user selects "user123" as the parameter. The system retrieves the subscriber's location from the UserLocations dataset by matching the username and will provide results for the user based on their location.

BAD data channels provide two modes for delivering data: *push* and *pull*. In push mode, the data of interest is pushed to brokers directly. In pull mode, the broker receives a notification from the channel whenever new data of interest becomes available for its subscribers; the subscribers can then request (pull) their data at any time. However, since many applications prefer immediate access to complete information rather than making requests to retrieve results, this paper focuses exclusively on push channels. In this approach, the broker receives data as soon as it is produced after each channel execution and immediately disseminates (pushes) it to subscribers. We defer the exploration of optimizing pull channels to future work.

Considering the need to promptly inform subscribers, the update interval for the TweetsAboutCrime channel ("period" in Figure 3) has been set to every 10 minutes. Upon channel creation, a dataset named TweetsAboutCrimeSubscriptions is created for storing subscriptions. Each subscription is identified by a unique ID with an associated broker and subscription parameters. Every 10 minutes, a recurring query, as shown in Figure 5, generates channel results and matches them with the relevant subscriptions for distribution via brokers. The broker information is stored in the Broker metadata dataset. If results are not ready to be delivered to the broker before the deadline, the channel is halted.

```
// Users can subscribe to the channel using
// their usernames to get the threatening tweets
// posted near their location.
CREATE CONTINUOUS PUSH CHANNEL
TweetsAboutCrime(MyUserName)
PERIOD duration ("PT10M") {
SELECT t.text
FROM EnrichedTweets t, UserLocations u
WHERE spatial_distance(u.location,t.location)<10
AND u.username=MyUserName AND is_new(t)
AND t.about_country="US"
AND t.retweet_count>10000
AND t.threatening_rate>5 };
```

Figure 3: DDL for the TweetsAboutCrime channel.

4. Optimizing BAD

To address the challenges posed by escalating data volumes and a rising count of subscriptions, it is imperative to implement optimization strategies for the channels within the BAD platform. These optimizations are crucial for ensuring that the system can manage data effectively, accommodate all subscriber requests promptly, expedite query processing, and consistently meet designated delivery deadlines as its users and data scale. When examining the processing performed by the original implementation of the BAD platform, we discovered three patterns which offer opportunities for optimization. In particular, (1) Duplicate Processing: the original BAD platform processes results for subscriptions that ask for parameters that are identical as if they were distinct subscriptions, resulting in redundant computations. (2) **Overprocessing**: subscription queries are executed on the entire dataset that has accumulated since the last execution, even when new records do not match existing subscriptions, leading to superfluous processing. (3) Late Data Filtering: in the origial BAD platform, although the query and all predicates of a channel are defined during channel creation, which occurs prior to any channel execution, the system postpones processing and identifying the relevant data until execution time. For each of these cases, below we discuss the proposed solutions and their benefits, which are then showcased in the experimental section. More details about the proposed optimizations are available in [30]. SUBSCRIBE TO

TweetsAboutCrime("user123") **ON** BrokerA;

Figure 4: DDL for subscribing to the channel TweetsAboutCrime.

SELECT result, current_datetime()

as deliveryTime, sub.subscriptionId as sId **FROM** Metadata.`Broker` b,

KOM Metadata. Broker D,

TweetsAboutCrimeSubscriptions sub, TweetsAboutCrime(sub.param0) result

where result.BrokerName=b.BrokerName;

Figure 5: The query running under the hood for the channel TweetsAboutCrime.

4.1. Duplicate Processing: Aggregating Subscriptions

In systems designed to serve a large user base, multiple users often create similar queries, leading to redundant processing when retrieving results separately for each user. The BAD platform introduced data channels to group common user query patterns into a single parameterized query, allowing users to select their own parameters. However, further analysis has revealed additional areas where we can go a step further by implementing even more efficient sharing mechanisms. For instance, in use cases where users subscribe to specific content categories like trending topics or news alerts, the limited number of categories can result in numerous subscriptions requesting the same data, differentiated only by subscription IDs. This redundancy can burden the system with multiple subscriptions that, upon closer inspection, are requesting the same information. A similar issue is observed in the BAD platform. For example, in the TweetsAboutDrugs channel, shown in Figure 6, subscribers must specify their state. Since there are only a finite number of U.S. states, this setup often results in multiple records for the same state, with the only variation being the users IDs. To avoid the inefficiencies associated with storing duplicate records, we propose grouping subscriptions based on their parameters and associated brokers. As a result, we create subscription-group records that maintain the group's parameter and broker name, along with an array that includes all subscription IDs requesting the respective parameter and broker. Since groups may vary in size, these records are of variable length.

Figure 7 illustrates the transformed subscription dataset for the new optimized system. To minimize additional overhead when grouping subscriptions, the groups are created, and subscriptions are assigned to the appropriate group as they enter the system. The ID of each new incoming subscription is either allocated to a pre-existing group, or it initiates a new subscription-group if its parameter and broker combination is not yet represented.

Although grouping many subscriptions into a single record may seem practical, it introduces certain system challenges, which are discussed below along with potential solutions. Note that subscription aggregation can be applied in various domains where multiple users share common interests or query parameters. For instance, in a financial monitoring system, users may subscribe to get updates about stock market trends based on specific conditions such as price movements, trading volume, or company performance metrics. Grouping these subscriptions by similar thresholds or parameters allows the system to handle multiple users with similar interests more efficiently, reducing duplicate processing. Similarly, in a content recommendation system, users may subscribe to updates about certain genres, authors, or topics. Aggregating these subscriptions based on similar preferences ensures more efficient content delivery and query execution.

CREATE CONTINUOUS PUSH CHANNEL

TweetsAboutDrugs(Mystate) PERIOD duration ("PT10M") {
 SELECT t.text
 FROM EnrichedTweets t
 WHERE t.state=Mystate AND is_new(t)
 AND t.threatening_rate=10
 AND t.drug_activity="Manufacturing Drugs" };
Figure 6: DDL for the TweetsAboutDrugs channel.
 tata D = Multimize A train DD

4.1.1. Parallelism in AsterixDB

In all scalable database management systems, when data is overly aggregated, the ability to leverage the system's parallel architecture can be compromised, as fewer tasks can be distributed simultaneously, leading to potential bottlenecks and decreased performance efficiency. AsterixDB faces similar challenges. To delve deeper into the operational mechanics, we first note that the unit of data which is consumed and produced by different tasks in AsterixDB is called a *frame*. It is a fixed-size chunk of contiguous bytes which always contains complete records, ensuring that a record is not split across multiple frames. An operator that produces data packs a frame with a sequence of complete records and sends it to the consumer operator who then interprets the records.

Overall, the frame size in AsterixDB is selected to balance memory efficiency, data movement, and task execution performance within the system. The optimal frame size is typically chosen based on the characteristics of the data being processed, the available memory resources, and the specific requirements of the workload. Larger frame sizes are often preferred for workloads involving complex data processing or large records, as they can reduce the number of I/O operations and enhance network efficiency. However, this choice must be carefully managed to avoid memory contention or excessive garbage collection, which can degrade system performance.

After fixing the frame size, if a record is larger in size than the standard frame size, the particular frame is enlarged to include this record; as a result, there may be frames that are longer than the fixed size. Consolidating many subscriptions into a single, large record affects the distribution of processing tasks across different operators. Let f denote the fixed frame size in bytes and s_i the record size in bytes of subscription-group i. If the group consolidates many subscriptions and s i surpasses f, we must expand the frame size to fully encompass the subscription-group since a record cannot be fragmented across frames. This may lead to fewer but larger frames and potentially reduce parallelism. In this case dividing subscription-group *i* into smaller subgroups can be another option to improve parallelism. However, dividing subscription-group *i* into too many smaller subgroups will lead to significantly increasing the computational load, since the system will calculate the same result multiple times.

Clearly, there is a trade-off involved in optimizing the number of subscriptions within each group. This trade-off will be further examined in the experimental section.

Aggregated subscriptions in Subscriptions Dataset

	Aggregated subscriptions in Subscriptions Dataset			
	GroupID	Param0	BrokerName	sIDs
New subscription id:s6 asking for "CA"	1	CA	Broker1	[s1,s3,s4, <u>s6]</u>
	2	NV	Broker1	[s2,s5]
New subscription id:s7 asking for "AZ"	3	AZ	Broker1	<u>[s7]</u>

Figure 7: Aggregation of subscriptions based on matching parameters and brokers.

4.1.2. Broker Benefits

Aggregating subscriptions extends advantages beyond merely improving query execution times(the query execution time is defined as the duration measured from the start to the end of the query execution). It also offers significant advantages on the broker side by notably decreasing both the communication time between the BAD platform and the brokers, as well as the processing time required by brokers to manage the results prior to dispatching them to subscribers. Consider a scenario where a new EnrichedTweets instance (which is around 32 KB) pertains to drug-related activities in California, and suppose there are one million subscriptions for this state on the TweetsAboutDrugs channel. Previously, this would create 1 million individual but similar results, each corresponding to a subscription, thereby burdening the broker with the management of redundant outcomes. By aggregating the subscriptions according to channel parameters we substantially decrease the volume of results that need to be transmitted and processed by the broker. Instead of sending results for each individual subscription, results are sent out per group, decreasing the data volume from 32 GB to just 0.07756 GB.

4.2. Overprocessing: Augmenting the Query Plan to Align with User Preferences

Consider the MostThreateningTweets channel, shown in Figure 8, which allows users to learn about the most threatening (level 10) tweets in their state. The query plan for implementing this channel appears in Figure 9 (a). Originally, the BAD platform would completely scan the EnrichedTweets dataset, the channel's subscription dataset called MostThreateningTweetsSubscriptions, and the Brokers datasets, and eventually perform several steps leading up to a join between the selected EnrichedTweets and the MostThreateningTweetsSubscriptions dataset. This approach becomes wasteful if most of the incoming data rarely meets the subscription criteria. This is the case when only a tiny fraction of tweets pertain to criminal activities and require law enforcement's attention. Therefore, failing to consider subscription parameters early in the query execution process can lead to creating a large volume of results which will eventually be discarded due to not matching any subscription criteria. Additionally, it is important to note that some subscriptions might have similar parameters, and thus it is also crucial to structure the query process to avoid redundant computations. This scenario can arise in any system that tries to align users' interests with incoming or stored data. Overloading operators with irrelevant data that doesn't contribute to the desired results is inefficient and undesirable. We propose an approach that selects only the data satisfying at least one subscription. An initial solution might involve identifying relevant results by performing a join between the dataset containing the required parameters, e.g., the Enriched tweets dataset, and the subscription dataset in the first step. However this would result in a massive join operation between two datasets with millions of records but with a small number of results. For instance, in the case of the MostThreateningTweets example, this solution would involve joining EnrichedTweets with the subscriptions, leading to a large join that produces a small number of results.

CREATE CONTINUOUS PUSH CHANNEL

MostThreateningTweets(MyState) PERIOD duration ("PT10M") {
 SELECT t.text

FROM EnrichedTweets t

WHERE t.state=MyState AND is_new(t)

AND t.threatening_rate=10 };

Figure 8: DDL of a channel created for the most threatening tweets.



Figure 9: (a) Original vs. (b) Optimized channel plan.

4.3. Late Data Filtering: BAD Index

To address effectively, we introduce a "UserParameters" dataset (a dataset which will be created by the system when a channel is created), integrated into each channel's plan, replacing the direct use of subscription parameters. This dataset includes fields for the channel's parameter(s) and the number of subscriptions interested in each. These fields facilitate the dynamic addition or removal of parameters as subscriber interests evolve.

The updated query advances the join operation between the UserParameters and EnrichedTweets datasets to occur during the initial data scan, reducing processing overhead. To further enrich channel results with detailed subscription information, such as each subscriber's broker, an index nested loop join is applied between the outcomes derived from the above join and the MostThreateningTweetsSubscriptions dataset. The revised query plan is depicted in Fig 9 (b).

When a query is known in the system before execution, it creates the possibility for the system to filter out irrelevant incoming records based on the query's specific conditions, potentially improving execution efficiency. We can observe a similar scenario in the BAD system, where channels incorporate fixed selection criteria to ensure only relevant data is processed and delivered to subscribers.

For example, the TweetsAboutCrime channel appearing in Figure 3 has three predicates that involve the incoming data (combined with AND): t.about_country="US", t.retweet_count>10000, t.threatening_rate>5. Knowing these fixed predicates in advance allows for the integration of an additional pre-execution filter, streamlining the query evaluation process by focusing only on relevant data from the outset before executing the main logic of the query. This filter identifies an incoming record that meets all fixed selection predicates specified in the channel query's WHERE clause and adds this record's id in a dedicated secondary index, known as the channel's **BAD index**. The method for creating the BAD index is described below.

CREATE CONTINUOUS PUSH CHANNEL

TweetsAboutCrime(MyUserName) PERIOD duration ("PT10M") {
 SELECT t.text
 FROM UserLocations u, EnrichedTweets t
 WHERE spatial_distance(u.location,t.location)<10
 AND u.username=MyUserName AND is_new(t)
 AND t.about_country="US" //(I)
 AND t.about_country="US" //(I)</pre>

c.recweet_counts roooo	//(11)
t.hate_speech_rate>5	//(III)
t.threatening_rate>5	//(IV)
t.weapon_Mentioned=true	$//(V) \ \};$
	<pre>t.hate_speech_rate>5 t.threatening_rate>5 t.weapon_Mentioned=true</pre>

Figure 10: $\ensuremath{\mathsf{TweetsAboutCrime}}$ channel DDL with extra conditions.

When a channel query is submitted to the system, a BAD index is created for each active dataset involved in the channel that has fixed selection predicates applied to it. The fixed conditions for each dataset are grouped together and added to an ordered list, called conditionsList, created for that active dataset, and when the channel is deleted from the system its conditions are also deleted. Conditions lists help determining whether incoming data satisfies the conditions for any channel. Each new incoming record will be checked against all groups of conditions in this list, and if it satisfies all conditions for a given channel, it will be added to that channel's BAD index. For example, when the TweetsAboutCrime channel is created (Figure 3), it includes three fixed conditions on the active EnrichedTweets dataset. Consequently, the channel's conditions are added to the EnrichedTweets dataset's conditionsList and a BAD index called TweetsAboutCrimeB-ADindex is created for the channel. Any new tweet that meets all three predicates in the TweetsAboutCrime channel will be added to the TweetsAboutCrimeBADindex. Algorithm 1 demonstrates the procedure for adding entries to BAD indexes for an active dataset.

Unlike general-purpose indexes, which store data for all records, the BAD index focuses solely on records that meet the fixed predicates of a channel's query. The BAD index offers a major improvement over traditional indexes by filtering and retrieving only the data relevant to a specific query (we call this 'early result filtering'), thus avoiding the inefficiencies of indexing all records in a dataset. This selective indexing reduces storage overhead and eliminates the need to scan irrelevant data, resulting in faster query execution. This indexing approach facilitates the application of time filters [31] to the BAD indexes, enabling efficient retrieval of only the most recent tweets when the is_new function is present the query. The time filter employs the timestamp from the channel's most recent execution to guarantee that the index search includes only records that have arrived at or since the time of the last execution. As a result, a BAD index boosts channel execution by quickly locating data that meets the channel criteria, eliminating the need to scan the entire dataset (EnrichedTweets in the above example). Algorithm 1 BAD Index Record Insertion

15: return

16: return True

Note that this pre-processing is reminiscent of, but different from, Partial Indexing [21], which focuses on indexing only the most frequently accessed or queried segments of a dataset. A BAD index does not index any attributes of the dataset; instead it consolidates the primary keys of all records that satisfy all the fixed predicates of a channel. This enables quicker access to the relevant records as opposed to scanning the entire dataset or using a traditional index which includes entries for all records, not just the ones that meet the specific criteria of the channel. To ensure that each channel query effectively utilizes the BAD index, the query plan must be adjusted to replace full dataset scans or the use of other secondary indexes with the BAD index. As explained earlier, regular secondary indexes contain both relevant and irrelevant records, making the BAD index a more efficient choice. Additionally, since the fixed predicates are already processed when data enters the system, they should be removed from the query plan to avoid redundant computations.

5. Experimental Evaluation

We proceed with outlining a series of experiments designed to assess the performance improvements of the optimized BAD platform as compared to its original configuration. Further experiments are detailed in [30]. We first evaluate the performance of each individual optimization and subsequently examine how the collective implementation of all optimizations provides comprehensive benefits. It should be highlighted that, previous research has already established the superiority of the BAD platform over a traditional, pieced-together solution with similar functionalities [3]. Given this, we can assert that an optimized version of the BAD platform would also outperform other comparable systems. Therefore, we have chosen not to repeat those earlier experiments here in this study.

For the following experiments, we used the example application that has been discussed throughout this paper, including its data model, datasets, and data channels. For all experiments we deployed the BAD platform on a 4-node cluster. Each node has an Intel(R) Xeon(R) CPU E5-2603 v4 @1.70GHz processor, with 64 GB of RAM, 10 TB of HDD, and 2×6-core processors.

5.1. Data

We initialize our experiments by loading the BAD platform with an initial EnrichedTweets dataset that contains 2 million synthetic tweets. This preloading was performed to ensure that the size of the EnrichedTweets dataset does not impact the performance of channel execution. Following the system's initiation, the BAD platform consistently receives 2000 EnrichedTweets per second, with each EnrichedTweet being approximately 30 KB in size. For each channel, we utilized datasets containing 1 million subscribers. Detailed descriptions of these subscriptions will be provided for each experiment. We have also created a dataset called UserLocations which is being used in the TweetsAboutCrime channel and includes people's usernames and their locations; each such record is 38 bytes. We assume that this dataset is continuously updated as the data is received as subscribers change their device's location. Only the most recent location for each user is maintained in the system.

In this study, synthetic tweets were used to assess system performance, allowing precise manipulation of field attributes to evaluate their impact on channel and query operations. By altering these parameters, we gained insights into system behavior under varying conditions, analyzing the efficiency and scalability of the indexing mechanism. To demonstrate the applicability of our optimizations, experiments were also conducted with a real-world Twitter dataset, detailed in Section 5.6.

5.2. Subscription Aggregation Experiments

Firstly, we examine the effectiveness of subscription grouping in the TweetsAboutDrugs channel (Figure 6). We begin by examining the trade-off presented in Section 4.1.1 regarding the optimal size for subscription groups. We conducted an experiment with a specially created dataset of 1 million subscriptions, all of which are interested in "CA". This setup allows us to examine the impact of large and small subscription-group sizes on system performance. In this experiment the AsterixDB frame size f is set to 40 KB while each original subscription is around 40 bytes.

Initially, we consolidate all "CA" subscriptions into a single subscription-group (that is 1024 times bigger than the size of the frame), which we then systematically halve to evaluate smaller subgroup sizes. Execution times for these various configurations are depicted in Figure 11. Each execution time is the time involved in running the channel after receiving 10 minutes of incoming tweets (at 2K tweets/sec). The left side (denoted as 1024 f) corresponds to having a single subscription-group of 1M subscriptions, while the right side (f/1024) corresponds to splitting this subscription-group into subgroups with 1 subscription each.

As it is shown in the figure, the execution time decreases when we have more than a single group, benefiting from enhanced parallelism, but the performance begins to suffer when we create many smaller subgroups, due to increased computational demands. The channel achieves the shortest execution times when the size of each subgroup matches the frame size, effectively balancing computational load and parallelism efficiency. As a result, we see that larger subscription-groups should be split into smaller ones to match the frame size.



Figure 11: Determining the ideal subgroup subscription size relative to frame size *f*.

Next, we compare the execution time of the original BAD channel to the optimized BAD channel, which incorporates aggregated subscriptions with subgroup sizes of *f*. Both configurations assume a total of 1 million subscriptions distributed across the 50 U.S. states, reflecting real-world population densities. Consequently, highly populous states such as California and Texas have a greater number of subscribers, whereas less populous states, such as Wyoming, have fewer. For instance, California's subscription group consists of 118,118 subscriptions divided into 116 subgroups, while Wyoming's group contains 1,723 subscriptions divided into 2 subgroups. Execution times demonstrate significant improvements with grouped subscriptions, decreasing from 255.23 seconds in the original BAD system to 57.23 seconds, highlighting the efficiency of aggregation.

5.3. Augmenting The Query Plan Experiments

In this section, we undertake an experiment utilizing the MostThreateningTweets channel (Figure 8) to demonstrate the impact of augmenting the query plan to integrate the UserParameters dataset. In particular we focus on varying the proportion of relevant EnrichedTweets and examine the system's responsiveness and efficiency while adjusting the percentage of EnrichedTweets that align with subscribers' preferences. We use three subscription datasets that are specifically designed to match a certain percentage of relevant tweets: in set 1 10% of the subscriptions are matching tweets, in set 2 15%, and in set 3 20% of the subscriptions are matching tweets.

As depicted in Figure 12, re-configuring the query plan to address user interests early on significantly decreases channel execution times for all sets. This optimization is particularly crucial for set 3 due to the higher proportion of relevant data, which necessitates producing a larger number of results. This need for efficiency is much higher when running the original BAD plan. For instance, with the third set of subscriptions, the optimized plan is essential as it allows the channel to process data efficiently and meet timesensitive deadlines, capabilities that are not as achievable with the original query plan.



Figure 12: The execution time of channel MostThreatening Tweets with different sets of subscriptions.

5.4. BAD Index Experiments

To test the BAD index, we employ a variation of the channel TweetsAboutCrime where we have added more fixed predicates (see Figure 10). The goal is to illustrate the impact of the BAD index under different levels of channel selectivity. The full version of the channel contains 5 predicates (marked I through V). To enhance the selectivity of the channel the predicates are incrementally applied to the channel query. The first three predicates (I,II and III) each have selectivity 50%, while the next two conditions (IV and V) each have selectivity 20%. By having only the first two conditions we will select around 17% of the incoming tweets. After applying the first three conditions, the channel selects about 10% of the incoming tweets. When a fourth condition is added, the selection is reduced to approximately 4.2%. Finally, applying all five conditions narrows it down significantly to just 0.07%.

We maintain a dataset called UserLocations, which includes the saved locations of users. This dataset contains 1 million entries. Note that changing the subscription dataset does not impact the comparison of execution times between the original BAD system and the optimized version since the BAD index is updated as new tweet records are ingested.



Figure 13: The execution time of the channel TweetsAboutCrime under varying conditions.

To ensure an equitable comparison, we assess the channel's execution times using a traditional index crafted on the attribute that is most selective under the given conditions. As an example, for the scenario with 2 conditions (I+II), this index is based on the retweet_count and when having 4 conditions, the index will be based on the threatening_rate as these are the most selective conditions respectively. Figure 13 shows the channel execution time measured with a traditional index as well as the BAD index. The creation of the BAD index significantly enhances the data retrieval capabilities. As illustrated in the figure, it reduces the execution time across various channel selectivities. Notably, the benefits of the BAD index increase as the channels become more selective, enabling more efficient filtering of unsatisfactory records prior to channel execution.

5.5. Comprehensive Performance of Optimized BAD

We now proceed with an experiment that synthesizes all three distinct optimization strategies that were proposed to enhance the BAD channel performance. It is important to note that each optimization may be particularly beneficial for specific scenarios. For instance, the BAD index proves most advantageous in channels which have very selective fixed conditions, the subscription aggregation excels in channels with a limited range of possible parameter values, while customizing the query plan is most effective when the user parameters significantly restrict the dataset that needs to be processed. An important comparison metric is the maximum number of subscriptions that the optimized channel can support; more supported subscriptions reflects the system's improved functionality to process and deliver timely and accurate results to an expanded number of subscribers within set deadlines (in our environment, the maximum number of subscribers that can be supported in the 10 minutes between subsequent channel invocations).

Figure 14 showcases the enhanced capacity of three specific channels to manage subscriptions using different optimizations; the original BAD, each optimization alone, and the fully optimized BAD as shown. As illustrated in the figure, implementing any combination of the proposed optimizations, or all of them, increases the capacity of each channel to support more subscriptions in the BAD platform.



Figure 14: Maximum number of subscriptions supported.

5.6. Experiments with Real-world Data

To demonstrate the real-world applicability of our methods, this section utilizes actual tweets collected from Twitter [32]. According to the study [33], the most common languages in the overall dataset of real tweets are English, Japanese, Spanish, Arabic, and Portuguese. In the subset of the dataset used for our experiments, English was the dominant language, followed by Portuguese, with the other languages being less represented. As a result, we focused on the two most prevalent languages English and Portuguese. The first channel, called EnglishTrendingTweetsInACountry, shown in Figure 15, sends subscribers trending tweets in English (those with retweet_counts greater than 100,000). Similarly, the second channel, PortugueseTrendingTweetsInACountry, targets trending tweets in Portuguese.

Users can subscribe to either channel based on their country of interest, and every 10 minutes, they receive trending tweets from that location. For this experiment, the tweet inflow rate was set to 6,000 tweets per second, which aligns with the average tweet generation rate reported in [34]. Each tweet averages about 3.5 KB, including details like user data, retweet counts, and location. In total, 1 million subscriptions were generated, with the distribution proportional to each country's population, meaning more populous countries had a larger number of subscribers. Figure 16 shows the execution times for both channels. In this figure, we compare the performance of the original BAD system with a traditional index on the most selective field, retweet count, along with various optimizations. As depicted, each optimization reduces execution time for both channels. Notably, the BAD index offers greater time reduction when the PortugueseTrendingTweetsInACountry channel. This occurs because most tweets are in English, making Portuguese queries more selective and leading to a significant difference in the number of records stored in a traditional index compared to the BAD index. It is important to note that channels can be created with varying levels of complexity, from simple to highly sophisticated, depending on specific needs. Different channels may benefit from particular optimizations based on their characteristics. However, the experiment in Figure 16 demonstrates that even for simpler channels like EnglishTrendingTweetsInACountry and PortugueseTrendingTweetsInACountry, that use nonenriched tweets, the execution time is reduced by 62% and 70%, highlighting the benefits of our optimizations.

CREATE CONTINUOUS PUSH CHANNEL

EnglishTrendingTweetsInACountry(countryName)
PERIOD duration ("PTIOM") {
 SELECT t.text
 FROM Tweets t
 WHERE t.country=countryName AND is_new(t)
 AND t.retweet_count>100000 AND t.lang="en" };

Figure 15: TrendingTweetsInACountry channel DDL.



Figure 16: TrendingTweetsInACountry execution time in different conditions.

6. Conclusions And Future Work

In this paper, we concentrated on enhancing scalability, performance, and efficiency of a Big Active Data (BAD) platform. We discussed different example use cases where users use BAD services to monitor a high speed incoming data source like tweets. In order to reduce the execution time and increase the supportable number of users, we introduced three different approaches including: (i) strategically consolidating subscriptions, (ii) revising (augmenting) query plans, and (iii) implementing the BAD index (for early result filtering). Our findings demonstrate a significant enhancement in system performance. In this paper, our optimization efforts have focused solely on individual channels. Looking ahead, our future work will focus on strategies for optimizing multiple channels concurrently, which holds great potential. For example, we will explore grouping channels and refining the BAD index to synchronize indexing activities across channels.

Acknowledgements

This work was supported in part by NSF awards CNS-1925610, CNS-1924694, NIFA award 2024-67022-43695, an industrial gift from Google, and funding from the Donald Bren Foundation (via a Bren Chair at UC Irvine).

References

- M. J. Carey, S. Jacobs, V. J. Tsotras, Breaking BAD: a data serving vision for big active data, in: Proc. of ACM DEBS, 2016, pp. 181–186. URL: https://doi.org/10.1145/ 2933267.2933313. doi:10.1145/2933267.2933313.
- [2] S. Jacobs, X. Wang, M. J. Carey, V. J. Tsotras, M. Y. S. Uddin, BAD to the Bone: Big Active Data at its Core, VLDB J. 29 (2020) 1337–1364. doi:s00778-020-00616-7.
- [3] X. Wang, M. J. Carey, V. J. Tsotras, Subscribing to big data at scale, Distributed and Parallel Databases 40 (2022) 475–520. URL: https: //doi.org/10.1007/s10619-022-07406-w. doi:10.1007/ S10619-022-07406-W.
- [4] D. Terry, D. Goldberg, D. Nichols, B. Oki, Continuous queries over append-only databases, SIGMOD Rec. 21 (1992) 321–330. URL: https://doi.org/10.1145/141484. 130333. doi:10.1145/141484.130333.
- [5] J. Chen, D. J. DeWitt, F. Tian, Y. Wang, NiagaraCQ: A scalable continuous query system for internet databases, in: Proc. ACM SIGMOD, 2000, pp. 379–390. URL: https://doi.org/10.1145/342009.335432. doi:10.1145/342009.335432.
- [6] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, J. Widom, STREAM: The Stanford Data Stream Management System, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 317–336. URL: https: //doi.org/10.1007/978-3-540-28608-0_16. doi:10.1007/ 978-3-540-28608-0_16.
- [7] A. Arasu, S. Babu, J. Widom, The CQL continuous query language: semantic foundations and query execution, The VLDB Journal 15 (2006) 121–142. URL: https://api.semanticscholar.org/CorpusID:6407934.
- [8] J. Kreps, N. Narkhede, J. Rao, Kafka: A distributed messaging system for log processing, in: Proceedings of the NetDB, 2011, pp. 1–7.
- [9] Apache Kafka, Kafka streams, https://kafka.apache. org/documentation/streams/, 2010. Accessed: 2024-10-10.
- [10] Apache NiFi, Apache NiFi Documentation, 2006.
- [11] A. Katsifodimos, S. Schelter, Apache Flink: Stream analytics at scale, in: 2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW), 2016, pp. 193–193. doi:10.1109/IC2EW.2016.56.
- [12] Amazon Web Services, Inc., Amazon Kinesis, https: //aws.amazon.com/kinesis/, 2013. Accessed:2024-04-10.
- [13] Azure Stream Analytics, https://azure.microsoft.com/ en-us/services/stream-analytics/, 2008.
- [14] Google LLC, Google Cloud Dataflow Documentation, https://cloud.google.com/dataflow/docs, 2024. Accessed: 2024-05-10.
- [15] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Comput. Surv. 35 (2003) 114–131. URL: https://doi.org/ 10.1145/857076.857078. doi:10.1145/857076.857078.
- [16] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, T. D. Chandra, Matching events in a content-based subscription system, in: Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '99, Association for Computing Machinery, New York, NY, USA, 1999, p. 53–61. URL: https://doi.org/10.1145/301308.301326. doi:10.1145/

301308.301326.

- [17] M. Hong, A. J. Demers, J. E. Gehrke, C. Koch, M. Riedewald, W. M. White, Massively multi-query join processing in publish/subscribe systems, in: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07, Association for Computing Machinery, New York, NY, USA, 2007, p. 761–772. URL: https://doi.org/10.1145/1247480. 1247564. doi:10.1145/1247480.1247564.
- [18] M. M. Moro, P. Bakalov, V. J. Tsotras, 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, 2007, p. 866–877.
- [19] A. Carzaniga, D. S. Rosenblum, A. L. Wolf, Design and evaluation of a wide-area event notification service, ACM Trans. Comput. Syst. 19 (2001) 332–383. URL: https://doi.org/10.1145/380749.380767. doi:10.1145/ 380749.380767.
- [20] A. Carzaniga, M. Papalini, A. L. Wolf, Content-based publish/subscribe networking and information-centric networking, in: Proceedings of the ACM SIGCOMM Workshop on Information-Centric Networking, ICN '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 56–61. URL: https://doi.org/10.1145/ 2018584.2018599. doi:10.1145/2018584.2018599.
- [21] M. Stonebraker, The case for partial indexes, SIGMOD Rec. 18 (1989) 4–11. URL: https://doi.org/10.1145/74120. 74121. doi:10.1145/74120.74121.
- [22] M. J. Carey, D. Chamberlin, A. Goo, K. W. Ong, Y. Papakonstantinou, C. Suver, S. Vemulapalli, T. Westmann, SQL++: We Can Finally Relax!, in: Proceedings of the 40th IEEE International Conference on Data Engineering, Utrecht, Netherlands, 2024. Invited paper.
- [23] J. Fang, D. Lychagin, M. J. Carey, V. J. Tsotras, A new window clause for SQL++, VLDB J. 33 (2024) 595–623. URL: https://doi.org/10.1007/s00778-023-00830-z. doi:10.1007/s00778-023-00830-z.
- [24] S. Jacobs, M. Y. S. Uddin, M. Carey, V. Hristidis, V. J. Tsotras, N. Venkatasubramanian, Y. Wu, S. Safir, P. Kaul, X. Wang, M. A. Qader, Y. Li, A BAD demonstration: towards Big Active Data, Proc. VLDB Endow. 10 (2017) 1941–1944. URL: https://doi.org/10.14778/ 3137765.3137814. doi:10.14778/3137765.3137814.
- [25] S. H. A. Shirazi, M. Carey, V. Tsotras, Building an end-to-end BAD application, in: Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems, DEBS '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 184–187. URL: https://doi.org/10.1145/3465480. 3467840. doi:10.1145/3465480.3467840.
- [26] Apache AsterixDB, https://asterixdb.apache.org/, 2009.
- [27] W. Y. Alkowaileet, S. Alsubaiee, M. J. Carey, et al., Enhancing big data with semantics: The AsterixDB approach (poster), in: Proceedings of the 12th IEEE International Conference on Semantic Computing (ICSC), IEEE Computer Society, 2018, pp. 314–315.
- [28] R. Grover, M. J. Carey, Data ingestion in AsterixDB, in: Proceedings of the 18th International Conference on Extending Database Technology (EDBT), 2015.
- [29] X. Wang, M. J. Carey, An IDEA: An ingestion framework for data enrichment in asterixdb, Proceedings of the VLDB Endowment (PVLDB) 12 (2019) 1485–1498.

- [30] S. H. A. Shirazi, X. Wang, M. J. Carey, V. J. Tsotras, Optimizing big active data management systems, arXiv preprint arXiv:2412.14519 (2024). URL: https://arxiv.org/abs/2412.14519.
- [31] S. Alsubaiee, M. J. Carey, C. Li, LSM-based storage and indexing: An old idea with timely benefits, GeoRich'15, Association for Computing Machinery, New York, NY, USA, 2015. URL: https://doi. org/10.1145/2786006.2786007. doi:10.1145/2786006. 2786007.
- [32] Twitter, Twitter Platform, 2024. URL: https://twitter. com, [Accessed: 2024-09-05].
- [33] T. M. Luu, S. Ram, F. Rangel, P. Rosso, F. Menczer, On the persistence of linguistic inequality: English and the global balance of languages in wikipedia, EPJ Data Science 10 (2021). URL: https://doi.org/10. 1140/epjds/s13688-021-00271-0. doi:10.1140/epjds/ s13688-021-00271-0.
- [34] X Corp. Engineering, New Tweets per Second Record, and How, 2013. URL: https://blog.x.com/engineering/en_us/a/2013/ new-tweets-per-second-record-and-how, [Accessed: 2024-09-05].