UNIVERSITY OF CALIFORNIA
RIVERSIDE

A BAD Thesis: The Vision, Creation, and Evaluation of a Big Active Data Platform

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Steven G. Jacobs

September 2018

Dissertation Committee:

Dr. Vassilis J. Tsotras, Co-Chairperson
Dr. Michael J. Carey, Co-Chairperson
Dr. Ahmed Eldawy
Dr. Walid A. Najjar

The Dissertation of Steven G. Jacobs is approved:

Committee Co-Chairperson

Committee Co-Chairperson

University of California, Riverside

Acknowledgments

I am grateful to my committee and, in particular, my chairs, for all of the advice and help provided throughout my work.

To Amanda, who worked and waited patiently through 10 years of college with me:

Best of wives, best of women.

To Margot Lily Jacobs: "Shall we look at the moon, my little loon, why do you cry?

Make the most of your life, while it is rife. While it is light." - Sufjan Stevens

To okapis, wolverines, and pangolins: For their majesty.

ABSTRACT OF THE DISSERTATION

A BAD Thesis: The Vision, Creation, and Evaluation of a Big Active Data Platform

by

Steven G. Jacobs

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2018
Dr. Vassilis J. Tsotras, Co-Chairperson
Dr. Michael J. Carey, Co-Chairperson

Virtually all of today's Big Data systems are *passive* in nature, responding to queries posted by their users. Instead, this thesis shows how to shift Big Data platforms from passive to *active*. A Big Active Data (BAD) system should continuously and reliably capture Big Data while enabling timely and automatic delivery of relevant information to a large pool of interested users, as well as supporting retrospective analyses of historical information. While various scalable streaming query engines have been created, their active behavior is limited to a (relatively) small window of the incoming data.

To this end, this thesis presents a BAD platform that combines ideas and capabilities from both Big Data and Active Data (e.g., Publish/Subscribe, Streaming Engines). It supports complex subscriptions that consider not only newly arrived items but also their relationships to past, stored data. Further, it can provide actionable notifications by enriching the subscription results with other useful data. The platform extends an existing open-source Big Data Management System, Apache AsterixDB, with an *active toolkit*. The toolkit contains features to rapidly ingest semistructured data, share execution pipelines

among users, manage scaled user data subscriptions, and actively monitor the state of the data to produce individualized information for each user.

This thesis describes the features and design of the current BAD system and demonstrates its ability to scale without sacrificing query capability or individualization.

One part of the BAD platform, the *Data Feed*, relies on storage mechanisms that allow for fast ingestion, namely the Log-Structured Merge-Tree (LSM-Tree). As such, this thesis also presents work on a formal evaluation and performance comparison of theoretical and existing LSM Merge policies for fast ingestion.

Contents

List of Figures

Chapter 1

Introduction

Work on Big Data management platforms has led to map-reduce and other frameworks that provide after-the-fact Big Data analytics systems [4, 11, 74] as well as NoSQL stores [1, 6, 9] that focus primarily on scalable key-based record storage and fast retrieval for schema-less data. There are also modern platforms that seek to provide the benefits of both analytics and NoSQL [16, 17, 79]. While these systems generally scale well, they remain mostly "passive" in nature, replying with answers when a user poses a query.

With the ever-increasing amounts of data being generated daily by social, mobile, and web applications, as well as the prevalence of the Internet of Things, it is critical to shift from passive to "active" Big Data, overcoming barriers in ingesting and analyzing this sea of data and continuously delivering real time personalized information to millions of users. Past work on active databases [35, 49, 48, 72, 77] was never built to scale to modern data sizes and arrival rates. Recently, various systems have been built to actively analyze and distribute incoming data; these include Publish/Subscribe systems [71, 81, 42, 64, 65, 50]

and Streaming Query engines [46, 29, 59, 3, 30, 59]. However, these approaches have achieved a compromise by accepting functional constraints in order to scale (e.g, limiting queries to a small window of data, or supporting a specific class of queries).

In contrast, this thesis advocates for Big Active Data (BAD), an approach that aims to leverage modern Big Data Management in order to scale without giving up on data or query capabilities, and in particular to address the following user needs:

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

2. Important information for users is likely to be missing in the incoming data items, yet it may exist elsewhere in the data as a whole. The results delivered to users must be able to be **enriched** with other existing data in order to provide **actionable notifications** that are individualized per user.

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

This thesis presents design decisions, implementations, evaluations, and enhancements of a complete BAD platform. The rest of this thesis is organized as follows: Chapter 2 gives the background for BAD, including the related work in multiple areas that motivated and informed the project. Chapter 3 presents the user model and high-level description of BAD. In Chapter 4 is a detailed description of the implementation under-the-hood of BAD

on top of an existing Big Data Platform (AsterixDB). Chapter 4 also presents several experiments for an initial evaluation of BAD against existing Big Data technology. Chapter 5 presents enhancements that were informed by the initial performance evaluation, including experiments to evaluate those enhancements. Chapter 6 introduces a formal evaluation of merge policies for improving fast ingestion of incoming data through Log-Structured Merge (LSM) storage, comparing existing policies with two newly implemented policies. Chapter 7 concludes the thesis.

Chapter 2

Background

The model for Big Active Data builds on knowledge from several areas, including modern Big Data platforms, early active database systems, and more recent active platform work on both Pub/Sub systems and Streaming Query systems. Figure 2.1 summarizes how the BAD vision fits into the overall active systems platform space.

This chapter discusses the past achievements in these related areas, including shortcomings when compared to Big Active Data. It also includes the few projects that have specifically focused on Big Active Data, and the limitations that they incurred, as they represent incremental steps in the direction of becoming fully BAD.

2.1 Big Data Platforms

First-generation Big Data management efforts resulted in various MapReduce-based [36] frameworks and languages, often layered on Hadoop [4] for long-running data analytics; in key-value storage management technologies [23, 20, 37, 41] that provide simple

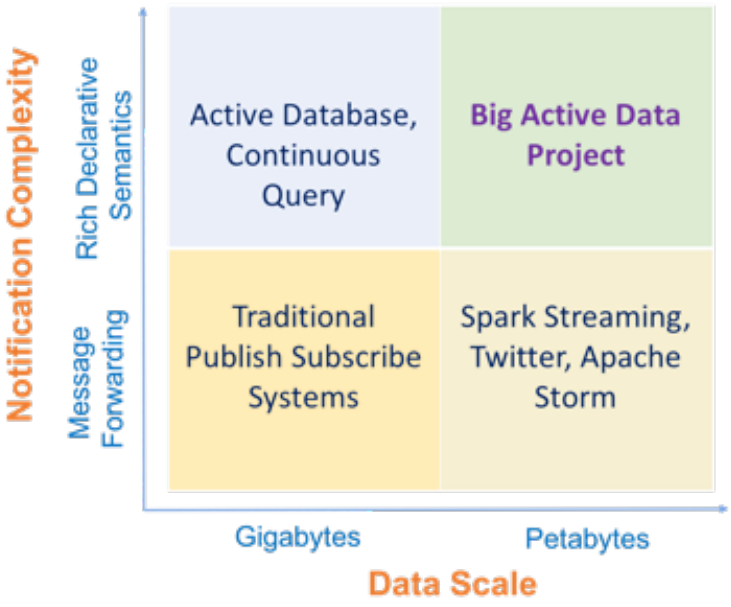


Figure 2.1: BAD in the context of other systems.

but high-performance record management and access; and, in various specialized systems tailored to tasks such as scalable graph analysis [78, 61, 60, 34] or data stream analytics [13, 21, 40, 45, 43, 62]. With the exception of data streams (which limit query capabilities in order to scale), these developments have been "passive" in nature – meaning that query processing, updates, and/or data analysis tasks have been scaled up to handle very large volumes of data, but these tasks run only when explicitly requested.

Several recent Big Data projects, including Apache Flink (Stratosphere) [3, 26], Apache Spark [7, 80], and Apache AsterixDB [2, 17], have made strides in moving away from the tradition of the MapReduce paradigm, moving instead towards new approaches based on algebraic runtime engines. Nevertheless, these approaches maintain a mostly-passive approach. *Data feed* mechanisms, such as those offered in AsterixDB [47], provide a step in

the direction of becoming active, and we have advanced and evolved them to become part of our Active Toolkit.

2.2 Active Data

The key foundations for active data (ECA rules, triggers) were arguably laid by the HiPac Project [35]. Many other systems contributed to the work on ECA rules, including TriggerMan [49], Ariel [48], Postgres [72], and Starburst [77]. There are, however, two issues when directly applying past active techniques on Big Data. First, triggers and ECA rules can be seen as a "procedural sledgehammer" for a system: when event A happens, perform action B. We seek a more declarative (optimizable) way of making Big Data active and detecting complex events of interest. Second, to the best of our knowledge, no one has scaled an implementation of triggers or ECA rules to the degree required for Big Data (in terms of either the number of rules or the scaled out nature of the data itself).

Work on Materialized View Maintenance (e.g., [31, 15, 70, 66]) is also related to Active Data. Nevertheless, materialized view implementations have generally been designed to scale on the order of the number of tables. Being more of a database performance tuning tool, the solutions developed in this area have not tried to address the level of scale that we anticipate for the number of simultaneous data subscriptions that should be the target for a BAD platform.

2.3 Publish/Subscribe Systems

In Pub/Sub Systems the data arrives in the form of publications, and these publications are of interest to specific users. Pub/Sub systems seek to optimize the problems of identifying the relevant publications and of delivering those publications to users in a scalable way. Early Pub/Sub systems were mostly topic-based (e.g., a user might be interested in sports or entertainment as publication topics). Modern Pub/Sub systems [71, 81, 42, 64, 65, 50] provide a richer, content-based subscription language, with predicates over the content of each incoming publication. Our BAD platform goes beyond this functionality in two ways: First, whether or not newly arrived data is of interest to a user can be based on not only its content, but on its relationships to *other* data. Second, the resulting notification(s) can include information drawn from other data as well.

There has been some work done to enable Pub/Sub systems to cache data in order to provide a richer subscription language and result enrichment [52, 76, 69], but this research has largely relied on limiting the size of the cached data (e.g., by storing a small window of recent history). This limitation prevents subscriptions from being applied to Big Data as a whole.

2.4 Continuous Query Engines

The seminal work in Tapestry [46] first introduced Continuous Queries over append-only databases, including a definition of monotonic queries. Most subsequent research has focused on queries over streaming data (e.g., STREAM [19], Borealis [14], Aurora [13], TelegraphCQ [27], and PIPES [57]). These systems are typically implemented internally by

building specialized data flows ("boxes and arrows") to process query results as new data streams through the system. Such approaches, as well as recent algebraic streaming query engines (e.g., Storm, Flink, and Spark Streaming) [30] are typically not designed to work with permanent storage, so their queries relate to individual incoming records or to recent windows of records.

2.5 Data-centric approaches

The most closely related work to BAD has been on supporting continuous queries via a *data-centric* approach, i.e., finding ways to treat user queries as "just data" rather than as unique data flows. To this end, NiagaraCQ [29] performed a live analysis of standing queries to detect "group signatures," which are groups of queries that perform a selection over the same attribute and that differ only by the constant of interest (e.g., age=19 vs. age=25). Given these group signatures, it created a dataset of the constants used and incrementally joined this dataset with incoming data to produce results for multiple users via a single data join. The growing field of Spatial Alarms [73, 22, 59] serves to issue alerts to users based on objects that meet spatial predicates. Spatial predicates are directly stored as objects (data) in an R-Tree, and incoming updates are then checked against all of the standing queries by simply performing a spatial join with this R-Tree.

The approach taken by NiagaraCQ and Spatial Alarms of treating continuous queries as data is one of the main inspirations for our own subscription scaling work. Both systems had limitations that we seek to relax in our work. NiagaraCQ was designed to operate using a very limited query language designed for XML data. Spatial Alarms focused

on one special use case (where the queries are locations) rather than on the problem as a whole. We build on these ideas for the more general world of Big Data, e.g., with horizontally partitioned data and a more fully expressive query language.

Chapter 3

Breaking BAD: A Data Serving Vision for Big Active Data

3.1 Introduction

This chapter presents the vision for leveraging a modern Big Data Management System, borrowing key ideas underlying the active capabilities offered by existing Pub/Sub and streaming query systems to build a complete BAD system. The Chapter is organized as follows: Section 3.2 gives a high level overview of the objectives, capabilities, and needs of a BAD platform, and it introduces an example BAD application that will be examined for the bulk of Chapters 3 and 4. Since a BAD system can be built starting from an existing passive BDMS, Section 3.3 discusses the shortcomings preventing a passive BDMS from interacting with users in an active way. In Section 3.4 the user capabilities of a BAD Active Toolkit are introduced.

3.2 A BAD overview

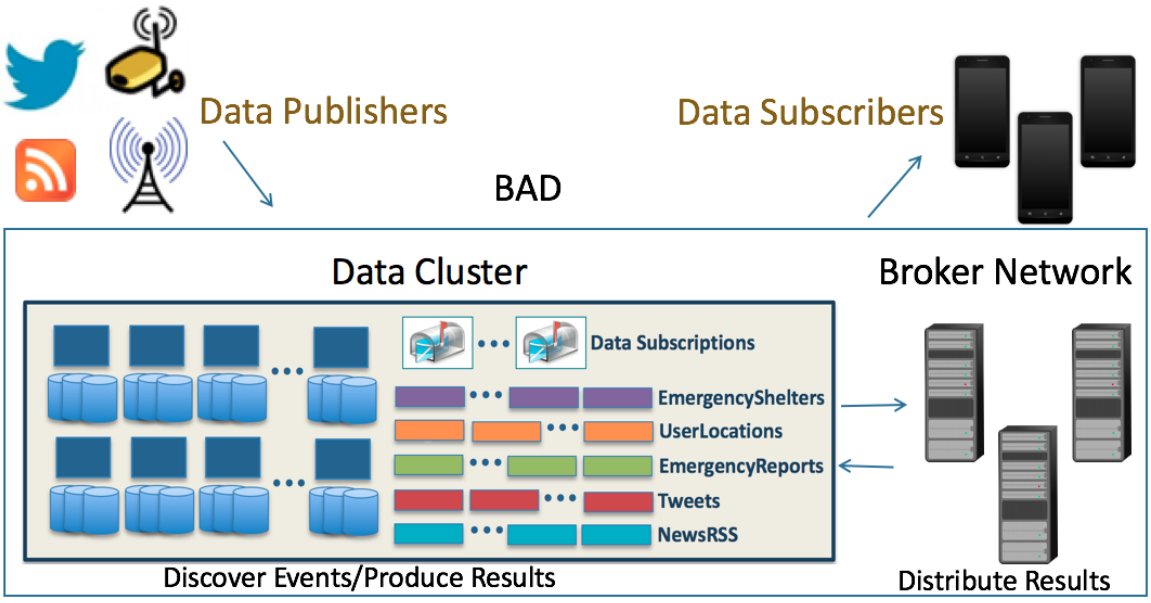


Figure 3.1: Big Active Data – System Overview.

Figure 3.1 provides a 10,000 foot overview of the BAD Platform. Outside of the platform, and the reason for its existence, are its data sources (Data Publishers) and its end users (Data Subscribers). Within the system itself, its components provide two broad areas of functionality – Big Data monitoring and management (handled by the BAD Data Cluster) and user notification management and distribution (handled by the BAD Broker Network).

3.2.1 A BAD Application

Consider as an example existing emergency notification services. One of the limitations of systems such as the USGS ShakeCast [12] system is that their notifications are

blanket statements for everyone (e.g., everyone receives the same flood warning or Amber Alert message). There is no individualization of messages to meet the needs of specific users (e.g., adding information relevant to the users' specific locations or needs). Such systems belong to the Pub/Sub system category, where users must manually select from a limited set of topics of interest.

In contrast, suppose a BAD system existed with the three capabilities sketched in Chapter 1. Rather than a user simply subscribing to emergency publications, now a user could ask something like "when there is an emergency near my child, and it is a flash flood, please notify me, and provide me with the contact information for the security on-duty at the school, as well as nearby safety shelter locations." Suddenly a user is not getting a simple publication, but a rich set of data specifically relevant to the user, including the enrichment of the emergency information with security personnel schedules, local shelter information, etc.

A similar example will be used for demonstration and evaluation purposes for the bulk of this Chapter (and the next). Specifically, the hypothetical example application will use three data sources: *UserLocations* (with records indicating the current location of each user), *Reports* (containing a record for each emergency generated as emergencies occur), and *Shelters* (holding the known locations of emergency shelters).

UserLocations and *Reports* will be continuously ingested into the data cluster, whereas *Shelters* will be loaded once and can be thought of as mainly a static, reference dataset (i.e., infrequently updated). The focus will be on users who want to know about emergencies occurring near them in space and time, and providing those users with in-

```
create dataverse emergencyNotifications;
use emergencyNotifications;

create type UserLocation as {
  location: circle,
  userName: string,
  timestamp: datetime
};
create type EmergencyReport as {
  reportId: uuid,
  Etype: string,
  location: circle,
  timestamp: datetime
};
create type Contact as {
  contactName: string,
  phone: int64,
  address: string?
};
create type EmergencyShelter as {
  shelterName: string,
  location: point,
  shelterType: string,
  contacts: {{ Contact }}?
};

create dataset UserLocations(UserLocation)
 primary key userName;
create dataset Shelters(EmergencyShelter)
 primary key shelterName;
create dataset Reports(EmergencyReport)
 primary key reportId autogenerated;

create index location_time on UserLocations(timestamp)
 type BTREE;
create index u_location on UserLocations(location)
 type RTREE;
create index s_location on Shelters(location)
 type RTREE;
create index report_time on Reports(timestamp)
 type BTREE;
```

Figure 3.2: Examples of ADM data types, datasets, and indexes

dividualized shelter information based on their locations. Note how the three user needs described in Chapter 1 apply to this example. The emergencies are only important to a user if they are near the known reported location of that user, and the provided notifications are enriched with shelter information before delivery. The emergency reports data will continue to grow over time and can be analyzed later to gain historical insights to help with long-term emergency service planning.

Figures 3.2 and 3.3 illustrates the data model and query capabilities for the example application, using SQL++ as the language; Figure 3.2 shows the data type, dataset, and index definitions that could be used for the example application, while Figure 3.3 shows

```
select report, u.userName from
( select value r from Reports r
  where r.timestamp > current_datetime() - day_time_duration(“PT10S”)) report,
UserLocations u
where spatial_intersect(report.location,u.location);
```

(a)

```
insert into Shelters (
{“shelterName” : “swan” ,
“shelterType” : “temporary” ,
“location” : point(“2437.34,1330.59”) ,
“contacts” : {{ { “contactName” : “Jack Shepherd”,“phone” : 4815162342 },
          { “contactName” : “John Locke”,“phone” : 1234567890 } }}
}
);
```

(b)

Figure 3.3: Examples of (a) an SQL++ query, and (b) an SQL++ INSERT statement

an SQL++ `SELECT` query and an SQL++ `INSERT` statement. The query in part (a) finds the emergencies that have been reported in the last ten seconds and joins them spatially with the locations of users, and part (b) shows how a new shelter could be added.

3.2.2 BAD Platform Prerequisites

A fully BAD Platform should take advantage of technologies and techniques that exist today for both Big Data performance and Scalable Delivery of results.

Big Data performance: For functionality and scalability, BAD should utilize the full capabilities of a modern BDMS; specifically, such systems can offer:

1. A rich, declarative query language.
2. Rich data type support that includes numeric, textual, temporal, spatial and semi-structured data.
3. Capabilities for fast data ingestion.
4. A data-partition-aware query optimizer.
5. A dataflow execution engine for partitioned-parallel execution of query plans.

Item (1) above prevents BAD applications from being limited in query capability, while (2) and (3) allow for the variety and velocity of modern Big Data, as well as active spatial-temporal queries. (4) and (5) enable the scaling of the data volume and optimizing of active tasks to run in parallel across the BAD platform.

Scalable Delivery of results: Publish/Subscribe systems continue to be a focus of research as the velocity and variety of Big Data continue to increase. Since a BAD data

cluster is useless unless its results actually reach its end users, a BAD platform must also offer the full capabilities of a Pub/Sub distributed delivery network, including:

1. Geo-distributed brokers that can scale dynamically to the demand of subscribers.
2. Dynamic heuristics for handling large influxes of subscribers and results.
3. Caching mechanisms designed with subscriber connectivity issues and commonality of interests in mind.
4. Enhancements for communicating with the underlying BAD data cluster efficiently.

3.3 Limitations of Passive BDMS

Big Data Management Systems platforms are limited from the perspective of the needs of an active framework. Most BDMS jobs are tied to an explicit user interaction, from start to finish. Compounding this problem is the fact that jobs are treated in isolation. Consider the use case where users want to know about emergencies near them as they occur. In a passive BDMS, information is only gained by directly requesting it (e.g., running a query to check recent emergencies near the user's current location). If a user wanted to continuously check for new information, the user would need to continuously request it (e.g., by sending a new request every 10 seconds to check for new emergencies from the last 10 seconds). This could be done in the following way:

1. The user sets up a cron job that runs every 10 seconds and calls the BDMS REST API.

2. At each execution, the script sends a query to the BDMS.

3. The BDMS treats this request as a new (never-before-seen) job, which must be parsed, compiled, and optimized. Then it is distributed to the nodes of the cluster.

4. The job for the query is finally executed.

5. The BDMS performs job cleanup, including the removal from all nodes of the information for the job.

6. Steps (2-5) are repeated ad infinitum.

This query model works well for a query that is run once, but clearly becomes wasteful when a job is repeated, resulting in significant shortcomings:

1. The work for steps (3) and (5) is repeated every ten seconds, even though it is exactly the same every time.

2. Every execution of the job requires explicit triggering by an outside source (the cron job in step (2)).

3. Both (1) and (2) are multiplied by the number of users who are performing the same task in parallel.

3.4 The Active Toolkit User Model

To overcome the above limitations, this chapter proposes the *Active Toolkit* User Model. It contains three language tools needed to allow users to interact actively with BAD, namely:

```
use emergencyNotifications;

create type UserLocationFeedType as {
 location: circle,
 userName: string
};
create type EmergencyReportFeedType as {
 Etype: string,
 location: circle
};
```

```
create feed LocationFeed with
{
  "adapter-name" : "socket_adapter",
  "sockets" : "bad_data_cluster.edu:10009",
  "address-type" : "IP",
  "type-name" : "UserLocationFeedType",
  "format" : "adm"
};
create feed ReportFeed with
{
  "adapter-name" : "socket_adapter",
  "sockets" : "bad_data_cluster.edu:10008",
  "address-type" : "IP",
  "type-name" : "EmergencyReportFeedType",
  "format" : "adm"
};
```

Figure 3.4: Create data feeds for the Reports and UserLocations

1. *Data feeds* to allow data publishers to rapidly add data. A data feed represents the flow of scalable rapid data into one dataset.

2. *Data channels* to actively process data with respect to subscriber interests. A single channel is shared by a scalable number of users yet delivers individualized results.

3. *Procedures* to allow active execution of arbitrary SQL++ tasks.

3.4.1 Data Feeds

Since data in an active environment is being generated rapidly, it would not be efficient to insert records one by one through a typical `INSERT` or `UPSERT` statement. Alternatively, bulk data loading is useful when there is a large collection of new data sitting on the disk waiting to be imported, but for data incoming as a continuous stream, a different mechanism is required. Data feeds [47] provide a way to persist continuous data streams into datasets.

In the example application, it is assumed that the data being ingested into UserLocations and Reports is highly dynamic, as the user locations are being updated and reports are being generated frequently. Figure 3.4 depicts feeds being created for both datasets. Both feeds in this example expect data in ADM format. The default create-feed statement creates a feed with "UPSERT" semantics (i.e., insert if new, else replace). This setting favors datasets like UserLocations where a single record (the location for a certain user in this case) may be updated repeatedly. The DDL demonstrates a socket adapter on a designated host (e.g., "bad_data_cluster.edu:10008"). When clients come, they can connect to these endpoints and send their data directly.

Note that Figure 3.4 defines two additional datatypes, "UserLocationFeedType" and "EmergencyReportFeedType", for the feeds. As the incoming data from the data sources are not assumed to be timestamped (as in many application scenarios) these types will be used for the data received. In this example BAD application, the timestamp is an important field as it will be used later for generating the emergency notifications. To annotate the incoming data with proper timestamps, a user-defined function (UDF) can be created and attached it to a feed so that the incoming data is timestamped before it reaches the dataset. An "add_insert_time()" UDF is shown in Figure 3.5. This function utilizes the built-in SQL++ functions "current_datetime()" and "object_merge()" to add a new field with the current timestamp to an incoming record.

As the final step in setting up a data feed, the UDF is attached to the feed pipeline, the feed is connected to the dataset, and the feed is started. The DDL statements to accomplish this are shown in Figure 3.6. All incoming records for the UserLocations and

```
use emergencyNotifications;

create function add_insert_time(record) {

  object_merge({"timestamp": current_datetime()}, record)

};


/*

Sample Incoming Record:

{"Etype" : "storm",

"location" : circle("846.5, 2589.4, 100.0")}

Sample Output Record:

{"Etype" : "storm",

"location" : circle("846.5, 2589.4, 100.0"),

"timestamp" : datetime("2018-08-27T10:10:05")}

*/
```

Figure 3.5: Create the "add_insert_time" function

```
use emergencyNotifications;

connect feed LocationFeed to dataset UserLocations apply function add_insert_time;
connect feed ReportFeed to dataset Reports apply function add_insert_time;

start feed LocationFeed;
start feed ReportFeed;
```

Figure 3.6: Connect the data feeds to both datasets with function

Reports datasets will now be annotated with an arrival timestamp that will be used shortly in their associated *data channels*.

3.4.2 Channels

Here the notion of a channel is introduced as a scalable mechanism that allows users to subscribe to data. A channel is a shared entity that produces individualized data for users. In order to scale, a channel is implemented as a parameterized query with each user specifying their individual parameter values of interest.

Consider the example application, where users want to be notified when emergencies occur that intersect with their current locations. A natural implementation using passive AsterixDB would be through *polling*. Every user would explicitly poll the data cluster, at some interval, to see whether something new has occurred since the last poll. This would incur a steep penalty since every instance of every poll would be seen and compiled as a brand new query. The performance of such a passive approach is examined in Section

4.5.

AsterixDB already provides an interface (*functions*) for defining a passive parameterized query that polling users could utilize. The move from passive to active for users can be colloquialized as follows: "Rather than calling this function myself to check for data, I would like the function to call me when there is data of interest to me." Or, more succinctly, "You've got data!"

A *repetitive data channel* can be thought of as an active, shared version of a function (in fact the channel DDL makes use of the existing SQL++ function DDL) that utilizes an optimized deployed job to leverage shared interests but that produces individualized results for many users at once based on their individual parameters and sends notifications when new data is produced.

The SQL++ DDL extension for channels leverages AsterixDB parameterized function definitions. As an example, recall the query in Figure 3.3 that joined recent emergency reports with the UserLocations dataset. A function that will run a similar query on behalf of a single user can be seen in Figure 3.7(a). When a user calls RecentEmergenciesNearUser("dharma1") in Figure 3.7(b), the variable "userName" will be replaced with "dharma1" in the query, and then the query will be treated normally. This provides a nice way to describe exactly the type of shared query that users of the example application would want to run. Note that the query in Figure 3.7(a) also enriches (personalizes) the user's results with nearby shelter information.

Figure 3.8 shows how a channel can be created based on the function from Figure 3.7. Creating a repetitive channel requires two parts: a function for the channel to use

```
use emergencyNotifications;


create function RecentEmergenciesNearUser(userName) {
 (
  select report, shelters from
   ( select value r from Reports r where r.timestamp > current_datetime() - day_time_duration("PT10S"))
   report, UserLocations u
  let shelters = (select s.location from Shelters s where spatial_intersect(s.location,u.location))
  where u.userName = userName and spatial_intersect(report.location,u.location)
 )
};
```

(a)

```
RecentEmergenciesNearUser("dharma1");
```

(b)

Figure 3.7: DDL (a) for a function that finds recent emergencies near a given user, and an example invocation (b) of the function

```
use emergencyNotifications;

create repetitive channel EmergenciesNearMe using
RecentEmergenciesNearUser@1 period duration("PT10S");

create broker BADBrokerOne at "BAD_broker_one.edu";

subscribe to EmergenciesNearMe("dharma1") on BADBrokerOne;
subscribe to EmergenciesNearMe("johnLocke") on BADBrokerOne;
```

Figure 3.8: DDL to create a channel using the function RecentEmergenciesNearUser@1, DDL for creating a broker, and DDL for creating a subscription to the channel

and a time (repeat) period. Creating a channel will compile the query contained in the function into a single deployed job that then will be run repetitively based on the period provided (in this case every 10 seconds). Every time this channel runs it will produce a set of individualized results for all of the data subscribers.

In addition to the channel, Figure 3.8 also shows how to create a *broker* as a recognized subscription endpoint in BAD AsterixDB. In order to provide scalability, data subscribers connect to the cluster through BAD brokers, providing a one-to-many connection to BAD AsterixDB (brokers are discussed in more detail in Section 4.4). When a data subscriber subsequently subscribes to a channel via a broker, the broker acting on behalf of the subscriber will provide BAD AsterixDB with: (i) the parameters relevant for that subscriber (in this case the id of the user), and (ii) the name of the broker making the request (in this case *BADBrokerOne*). Once a subscription has been created, the subscriber will begin to receive results for emergencies near her changing location over time (discussed in

detail in Section 4.4).

The Case for Continuous Channels

Repetitive channels have the limitation that they rely on some fixed time interval to execute (e.g., ten seconds). While this is fine if data is produced and desired at a specific rate, it cannot handle two extreme but common cases: users wanting data at the moment of its creation (not waiting until the next channel execution), and events of interest occurring infrequently (not producing results for several executions, and thereby potentially wasting interim query processing resources). In this direction could be the next generation of channels, namely *continuous channels*, inspired by [46], which will execute on data changes rather than relying on fixed execution periods.

It is interesting to note that when events are time-driven and the channel functions are time-qualified (as in the example application), repetitive channels can provide a batch-y approximation to continuous channels, as they only execute on and produce a small set of data if the repeat interval is not too large.

3.4.3 Procedures

Procedures are another entity for a BAD application. Conceptually, a procedure can be seen as an active implementation of a function, but without the limitation of only performing queries (procedures can perform insertions and deletions as well). In order to accomplish active objectives using procedures (addressing the limitations discussed in Section 3.3), procedures allow users to specify an execution frequency (e.g., 24 hours), thus

```
use emergencyNotifications;

create procedure deleteStaleShelters() {
delete shelter from Shelters
where shelter.shelterType = ''ad-hoc''
} period duration(''PT24H'');

execute deleteStaleShelters();
```

Figure 3.9: DDL for creating and executing a repetitive procedure

allowing the creation of *repetitive procedures*. Here the user will only make one explicit call. Subsequent executions will then happen actively, with no user interaction, every 24 hours. Suppose that, during a multi-day ongoing hurricane disaster, ad-hoc shelters are built in new locations every morning and taken down each night. During the night, it may be important to delete the stale ad-hoc shelter information before the next day starts. An application administrator could create the first example procedure (*deleteStaleShelters*) in Figure 3.9. Once "execute" is called (also shown in the figure) this procedure will continue to run once every 24 hours. There are also many under-the-hood processes that procedures can be used for to help with BAD applications. More examples will be seen in Chapter 4.

3.5 Users of the Active Toolkit

The term "user" could loosely apply to three different types of users in the example application, namely, Application Administrators, Data Publishers and Data Subscribers.

3.5.1 Application Administrators

An **Application Administrator** builds applications using the BAD framework. They have direct access to the data cluster for hosting datasets and *data feeds*. They have knowledge of: (1) Database Administration for the data stored in the Data Cluster for their applications, and, (2) the common interests of their "users" (eventual Data Subscribers). Based on user interests, an Application Administrator will create and manage parameterized *channels* that can be subscribed to in the application.

In the example scenario, the Application Administrator is who will create the *emergencyNotifications* dataverse. She will then create the three datasets: *Reports* (the continuously ingested reports), *UserLocations* (the current location of each "user" (subscriber) of the application), and *Shelters* (the relatively static metadata for shelter information, initially bulk loaded with all known shelters by the administrator).

The Application Administrator will then proceed to make this a BAD application by creating the *data feeds* for both *Reports* and *UserLocations* (DDLs shown in Figure 3.4), and by creating the subscribe-able repetitive channels for the application (via the DDL shown in Figure 3.8). Lastly, she can create the relevant Procedures to help with the active management of the application (shown in Figure 3.9).

3.5.2 Data Publishers

Data Publishers provide data in the form of streams of incoming records. These streams enter the Data Cluster directly via *data feeds*. In a typical use case, the data publishers will be exteral services that are known/trusted by the Application Administrators

(such as news sites, social media data streams, or government agencies) and the incoming data will be broadly relevant to a given BAD application (e.g., emergency reports or weather broadcasts).

In the example scenario, the Application Administrator will provide the Emergency Report publisher with the cluster endpoint for sending reports to the *ReportFeed*, and the publisher will then send its reports to this endpoint (e.g., "bad_data_cluster.edu:10008").

3.5.3 Data Subscribers

The third category is **Data Subscribers**. They connect to BAD applications and subscribe to one or more of their channels. They are never given direct access to the data cluster, but instead perform all of their tasks via a nearby BAD Broker. This separation of the subscribers from the cluster provides several advantages. Rather than dealing with result data requests per subscriber, the cluster instead receives aggregated requests from brokers on behalf of many subscribers at once, and in the same way the cluster sends aggregate notifications to the brokers, rather than communicating with every subscriber. This limits the per-user impact on the cluster, freeing more resources for BAD tasks. In addition, this layered approach separates concerns, allowing brokers to focus on problems of result caching and communication issues with the subscriber, while the cluster deals with the data creation itself. A *subscription* can be created for a specific channel and indicates the specific parameters of individual interest to a subscriber. Each subscription will be registered in the BAD data cluster with a subscription id. After its creation, the broker for a subscription will begin to receive new results from the channel that the given subscriber has subscribed to.

There are cases where a data subscriber may also serve as a data publisher. As an example, an application (as in the example application) may want to allow users to provide (publish) their locations to the application (e.g., to enable subscriptions involving those locations). For such cases, an API is provided for a data feed that passes data from the subscribers through the brokers to the data cluster.

In the example application, users will want to subscribe to the *EmergenciesNearMe* channel, so they will allow the application to have access to their current locations. The application will send this data via the brokers to the *LocationFeed*. Users can then subscribe to the channel (their associated brokers will do so as shown in Figure 3.8), at which point they will start receiving relevant results.

Chapter 4

BAD to the Bone: Big Active Data at its Core

4.1 Introduction

At the core of the BAD platform is the BAD data cluster. This cluster has been built by leveraging the benefits of Apache AsterixDB, a modern Big Data Management System (BDMS) [2, 17] (namely, its scalability, declarative query language, flexible data model, and support for parallel data analytics), as well as borrowing key ideas underlying the active capabilities offered by existing Pub/Sub and streaming query systems. The complete architecture is described here, including how to use it to create real-world BAD applications. This work specifically targets environments that need to ingest scalable data publications and collect them for subsequent Big Data analytics, and need to handle scalable numbers of data subscribers who are interested in the information gleaned from not just

this ingested data, but from the state of the data as a whole. This chapter specifically shows by example the implementation of the example application introduced in Chapter 3.

This chapter is organized as follows: Section 4.2 gives an introduction to AsterixDB, the BDMS that has been used to implement the Active Toolkit. Section 4.3 delves into the under-the-hood implementation of the Active Toolkit introduced via the user model in Chapter 3, showing how the subscription and channel requests made by users are translated into a data and execution model. Section 4.4 delves into the three communication layers of BAD, used to maintain subscriber interests, discover complex data states of interest, and distribute this information to subscribers efficiently. Section 4.5 evaluates BAD performance using the example BAD application. In order to provide a comparison, the same application was built using existing passive Big Data technologies, and the performance shortcomings of such an implementation are shown.

4.2 AsterixDB

The aim here is to start with a passive BDMS, in this case AsterixDB, and then show how to transform it into a BAD system. As a preamble to creating an Active Toolkit (Section 4.3), this Section discusses the advantages inherited from using AsterixDB as a starting point for building a BAD platform.

Apache AsterixDB (Figure 4.1) is a full-function BDMS that supports all of the prerequisites listed in Section 3.2.2. The underlying runtime engine for executing its queries, written in SQL++, is the Hyracks data-parallel platform [25]. Queries are compiled and optimized via the Algebricks extensible algebraic parallel query planning and optimization

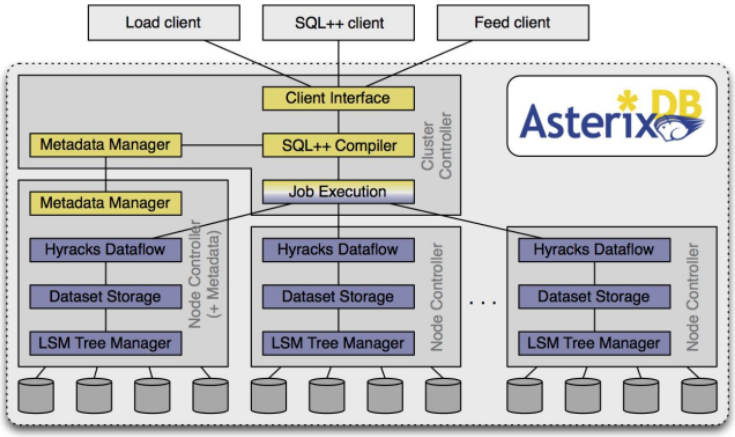


Figure 4.1: The architecture of passive AsterixDB

framework [24]. AsterixDB has fully developed support for rich Big Data types, including GeoJSON and other advanced spatial types, which fits well with location-oriented applications. Another feature of AsterixDB that makes it particularly suitable for becoming active is the provision of *data feeds* built on top of LSM (Log-Structured Merge) tree storage technology, allowing for fast data ingestion [18, 47].

4.3 The Active Toolkit Jobs

Chapter 3 gave the user model capabilities of the *Active Toolkit*. This Section discusses the underlying tools and enhancements added to AsterixDB that made the Active Toolkit realizeable. There are four main tools added for a BAD application, namely:

1. *Data feeds* allowing the flow of scalable rapid data into one dataset.

2. *Deployed jobs* that can perform arbitrary tasks. They get compiled and distributed once and used multiple times.

3. *Data channels* that manage a scalable number of subscriptions for a shared query. A single channel is compiled once as a deployed job, yet produces individualized staged results.

4. *Procedures* to use deployed jobs to perform other active management processes regularly and efficiently.

For each of these components, this Section delves into the implementation.

4.3.1 Data Feeds

[47] introduced the notions of a "primary feed", which gets data from an external data source, and "secondary feeds" that can be derived from another feed. In addition, either/both could have an associated user-defined function (UDF). Data feeds enable users to attach UDFs onto the feed pipeline so that the incoming data can be annotated (if needed) before being stored. A user could use that architecture to build a rich "cascade network" that routes data to different destinations for particular purposes.

While that initial architecture introduced a lot of flexibility for building feed dataflow networks, it also brought extra overheads related to persisting the data and additional complexities in maintaining the dataflow network. In a BAD application, the timeliness of data and the robustness of the network outweigh the user-level flexibility of defining a complex feed network. To meet the BAD requirements, the feed dataflow was *redesigned* into a more succinct yet equally powerful architecture (in terms of the set of addressible use cases), as depicted in Figure 4.2.

In the updated architecture, the previous cascade network was replaced by branch

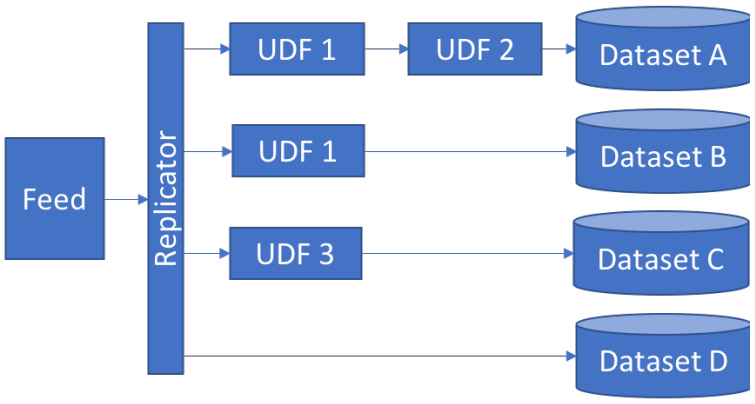


Figure 4.2: The updated feed dataflow

out sub-dataflows with a replicator. The sub-dataflows are isolated from one other so that the data movement in each sub-dataflow can proceed without interfering with the others. The UDFs attached to each path are evaluated separately as well.

A feed (on the far left in Figure 4.2) internally consists of an adapter and a parser. The adapter gets data from an external data source, and the parser translates the incoming data into ADM objects. Feeds were introduced with Socket, File, RSS, and Twitter adapters as well as JSON, ADM, Delimited Data, and Tweet parsers to handle common use cases. Building the BAD platform highlighted the need for adding many more adapters and parsers in order to handle a larger selection of data sources and formats. To address this issue, a pluggable adapter and parser framework was created so that users can add their own parsers and adapters and use them just like the native ones.

4.3.2 Deployed Jobs

The overhead of parsing, compiling, optimizing, and distributing a job (e.g., an "INSERT" or "QUERY" execution pipeline) can be especially time-consuming for small

jobs. For example, fetching a single record by primary key in AsterixDB currently takes around 20 milliseconds regardless of the size of the data cluster. This is because the process of parsing, compiling, optimizing, and distributing a job incurs a penalty of 10-20 milliseconds before the job even starts executing. This is "noise" for longer-running Big Data analytics queries, but for small jobs this process can become dominant.

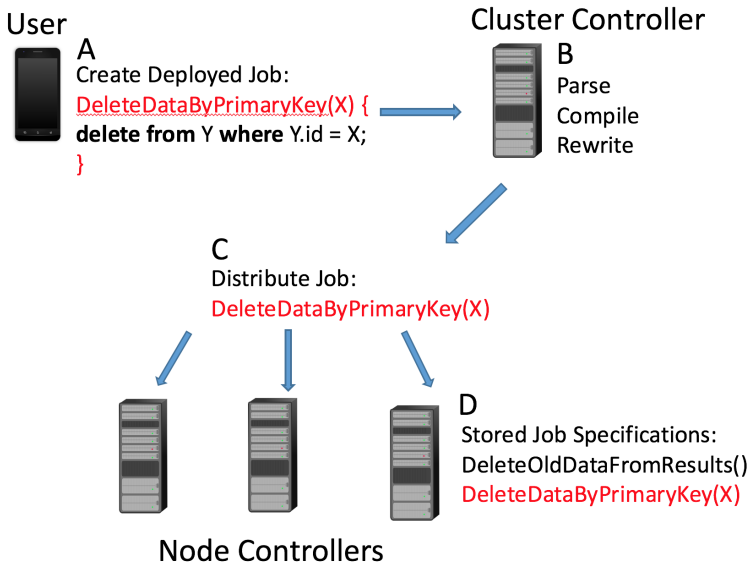


Figure 4.3: Deploying a Job

Deployed jobs were specifically created to address this overhead. When a deployed job is created (Figure 4.3), the SQL++ syntax for the job is provided (see step A). This is then parsed, compiled, and optimized once to produce a job specification (B). The resulting job spec is then distributed (C) and cached at each node (D).

When executing a deployed job (Figure 4.4), it is simply referenced by name (E). The cluster controller sends an "execute" command (F) to the nodes, which then execute

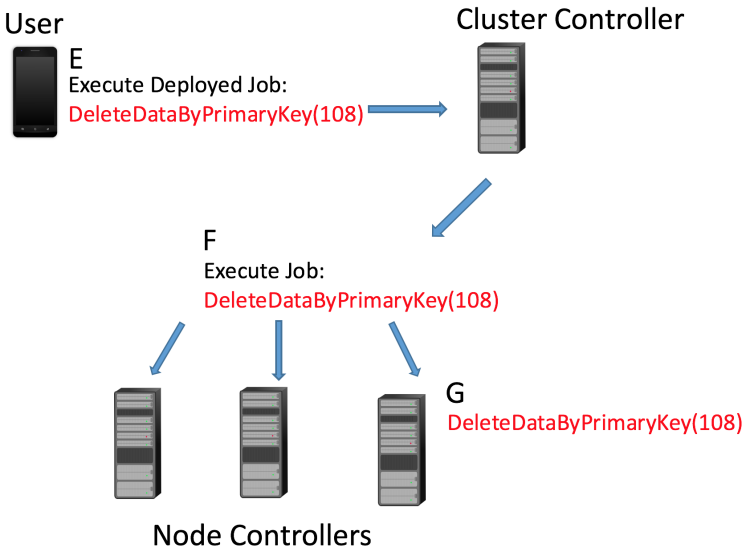


Figure 4.4: Executing a Deployed Job

the job (G) using the stored job specification.

Note that the deployed job in Figures 4.3 and 4.4 has a parameter representing the primary key of the record to be deleted. This parameterization is another enhancement that makes deployed jobs more robust. There are cases where different users may want to run a similar job that differs by only some set of parameters (in this case the id of the record). *Parameterized* deployed jobs were implemented to support this. The parameter values are passed to the node controller for the given job execution by the cluster controller. At runtime, an added operator in the job's ongoing plan fetches the value for a given parameter. At job cleanup, the stored parameter values are removed for the job. Allowing users to share parameterized jobs will be examined further in the Sections on channels and procedures.

A deployed job is one limited special case of an active job. More generally, a BAD platform should be able to support a scalable number of users (through a simple interface)

who subscribe to data of interest to them. This implies actively processing data as it changes, storing new results as they are found, and delivering them to the data subscribers. This is achieved through the next feature of the Active Toolkit: *repetitive data channels*.

4.3.3 Channels

Subscriptions

subscriptionId	brokerName	parameter0
27ea79fd-8009-4937-9a5a-6eac6c0bb1aa	Broker1	"johnLocke"
5b957e9a-17b5-4651-8a28-ec778db67af6	Broker2	"dharma1"
b634ae17-6b60-4a1d-b716-d080a1a76b07	Broker1	"MREcho"
17c130b5-bd9f-4539-a6cb-3ee5d9c893de	Broker3	"henryGale"
...	...	...

Results

subscriptionId	deliveryTime	result
27ea79fd-8009-4937-9a5a-6eac6c0bb1aa	2015-11-25 15:10:00	{"**Etype**":"flood", "**location**": circle("2719.47,967.5 1000.0"), "**shelters**": [{ "location": point("2872.98, 957.52") }, { "location": point("2913.07,954.89") }]...}
17c130b5-bd9f-4539-a6cb-3ee5d9c893de	2015-11-25 15:10:00	{"**Etype**": "fire", "**location**": circle("2208.19,2559.09 500.0"), "**shelters**": [{ "location": point("2023.99,2505.02") }]...}
5b957e9a-17b5-4651-8a28-ec778db67af6	2015-11-25 15:10:00	{"**Etype**": "flood", "**location**": circle("2830.75,1332.91 1000.0"), "**shelters**": [{ "location": point("2594.22,1078.11") }, { "location": point("2567.07,1104.86") }]...}
5b957e9a-17b5-4651-8a28-ec778db67af6	2015-11-25 15:10:00	{"**Etype**": "fire", "**location**": circle("2123.42,1297.09 500.0"), "**shelters**": [{ "location": point("2594.22,1078.11") }, { "location": point("2567.07,1104.86") }]...}
...	...	...

Figure 4.5: The subscription and results tables for the EmergenciesNearMe channel

This Section discusses in detail how BAD AsterixDB creates and manages data

channel work-flows under the hood using an *EmergenciesNearMe* channel as an example. When this channel is created, two new internal datasets will be created: *EmergenciesNearMeSubscriptions* and *EmergenciesNearMeResults*. The subscriptions dataset contains one record for each subscription that has been created. This includes three important pieces of information: (a) an automatically generated id for the subscription, (b) the name of the broker servicing the subscriber of the subscription, and (c) the channel parameter values for the subscription. The results dataset is where the result records for the channel will be stored (including their subscription ids). An example of these datasets appears in Figure 4.5. Although these datasets will start out empty, they are depicted with data to illustrate how their data could look over time. There are four subscriptions shown, along with results produced for some of these subscriptions.

Once these two datasets have been created, the channel will be compiled, optimized, and distributed to the node controllers as a deployed job. Rather than running the function separately for each subscription, a join is created between the function body and the *EmergenciesNearMeSubscriptions* dataset on the parameter values. The results produced are inserted into the *EmergenciesNearMeResults* dataset. This process starts with an SQL++ insert statement, shown in Figure 4.6. The call to RecentEmergenciesNearUser will be directly inlined by the AsterixDB parser into the body of the function, which joins Reports with the UserLocations based on the parameter values and adds the shelter information (recall Figure 3.7). The join with the Broker dataset is used to retrieve the broker endpoint information, which is used to send notifications out.

The "returning" clause capability shown was added to AsterixDB for the BAD

```
insert into EmergenciesNearMeChannelResults as r (
with channelExecutionTime as current_datetime()
select result, channelExecutionTime, sub.subscriptionId as subscriptionId,current_datetime() as deliveryTime
from EmergenciesNearMeSubscriptions sub,
Metadata.Broker b,
RecentEmergenciesNearUser(sub.param0) result
where b.BrokerName = sub.BrokerName
and b.DataverseName = sub.DataverseName
) returning r;
```

Figure 4.6: The SQL++ INSERT statement for the EmergenciesNearMe Channel

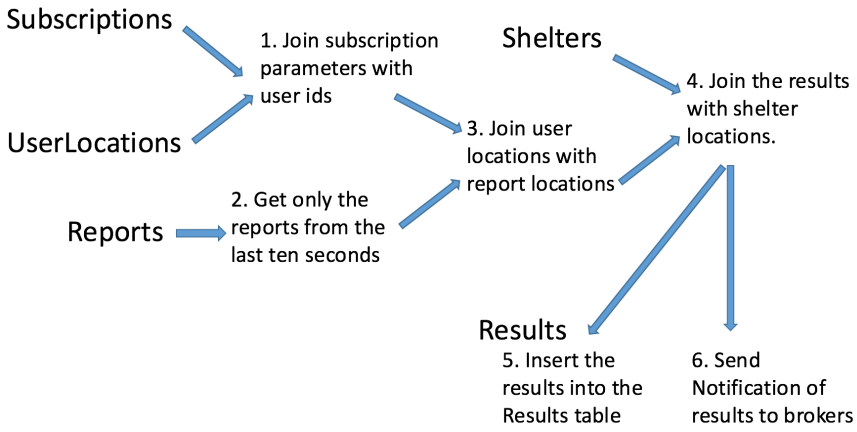


Figure 4.7: The deployed job for executing the channel EmergenciesNearMe

platform to allow results of INSERT statements to be distributed to users, and is also used under-the-hood for the "subscribe" statement, which returns the subscription id of the newly created subscription to the broker (discussed further in Section 4.11). In this

case, the returning clause will actually be rewritten by the BAD Optimizer (an enhanced version of the AsterixDB Algebricks optimizer) into a set of operators that will send the notifications of new results to the brokers.

This plan is then optimized into a plan DAG by AsterixDB's rule-based job optimizer (see Figure 4.7) that includes the following steps:

1. Join the Subscriptions and UserLocations datasets to find the locations of the subscribers.
2. Utilize the time index of Reports to fetch only the recent Reports (last ten seconds, from the function query).
3. Perform a spatial join between the results of steps (1) and (2).
4. Enhance the result with the nearby shelters (also spatially joined).
5. Insert the results into the results dataset.
6. Send the brokers notifications that new results have been created.

Once the job for a channel is deployed to the cluster, execution is synchronized by a timer on the cluster controller (Figure 4.8). When the timer goes off (A) the cluster controller communicates (B) with the node controllers to execute the channel. Then the node controllers use the stored job specification to execute (C) their portions of the channel job.

It is important to note the advantages of this approach over a passive polling method. If users were polling individually, each poll request would produce an individual

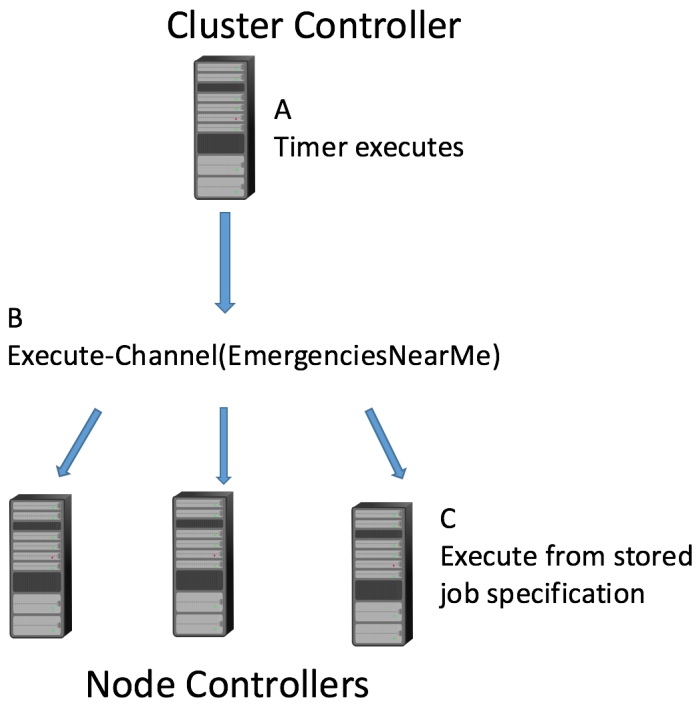


Figure 4.8: Repetitive execution of a channel

job very similar to Figure 4.7, but with the *Subscriptions* dataset being replaced by a single input value (the id of the user making the request) and with the *results* being delivered directly back to that user. In contrast, the deployed channel job executes once every ten seconds (the period of the channel) on behalf of all subscribers, potentially produces new results, and requires no intervention from the users. Section 4.4 discusses in more detail how the results (and the subscriptions) are communicated from the data cluster to each subscriber (and vise-versa).

4.3.4 Procedures

Under the hood, a procedure is translated into a corresponding deployed job, as in Figures 4.3 and 4.4. When a procedure is repetitive, the cluster controller triggers periodic execution on a timer, in the same manner as a channel (Figure 4.8).

Underlying channel datasets are a great candidate for using procedures. For example, brokers might want to retrieve lists of their current subscriptions to a given channel. Rather than having each broker send such a request as a new job each time, an application administrator could create the first example procedure (*CountBrokerSubscriptions*) in Figure 4.9 that can then be used multiple times by multiple brokers. This also shows how one can provide a parameter to a procedure (in this case the name of the broker of interest). The value of the parameter is then passed when "execute" is called (also shown in the figure).

Managing a channel results dataset provides a use-case for a repetitive procedure. The dataset can be thought of as a log of results being continually appended. This data might (depending on the type of application) be considered to be stale after some time threshold. In the example application, where users are notified of emergencies on an ongoing basis, it might be desirable to keep the old results in a broker-retrievable form for only one day.

An application administrator can easily set up a procedure for cleaning up the results dataset using the DDL and DML for the second procedure (*deleteStaleResults*) in Figure 4.9. The body of this procedure deletes channel results that are more than 24 hours old. Note that this procedure can be given an execution interval (24 hours). The "execute" call to initiate the active procedure will only need to be called once. The procedure will

```
create procedure CountBrokerSubscriptions(brokerName) {
select array_count(
(select sub from EmergenciesNearMeSubscriptions sub where sub.BrokerName = brokerName))
};

execute CountBrokerSubscriptions(''BADBrokerOne'');

create procedure deleteStaleResults() {
delete result from EmergenciesNearMeResults
where result.channelExecutionTime < current_datetime() - day_time_duration(''PT24H'')
} period duration(''PT24H'');

execute deleteStaleResults();

create procedure SubCountsForEmergenciesNearMe(){
insert into SubscriptionStatistics (
select current_datetime() as timestamp, b.BrokerName,
(select value array_count((select sub from EmergenciesNearMeSubscriptions sub
 where sub.BrokerName = b. BrokerName))) as subscriptions
from Metadata.'Broker' b)
} period duration(''PT1H'');

execute SubCountsForEmergenciesNearMe();
```

Figure 4.9: DDL for creating and executing three procedures (with the latter two being repetitive)

then continue to repeat every 24 hours. There are many other needs that procedures would be useful for in the application as well. For example, procedures could also be used to help evaluate broker utilization. The third procedure in Figure 4.9 (*SubCountsForEmergencies-NearMe*) will query the subscription counts for the *EmergencyChannel* for every broker, on an hourly basis, and insert the results into a *SubscriptionStatistics* dataset. Retrospective analytics can then be used on this dataset to tune the broker network itself.

There are many ways that procedures could enhance a BAD Platform, including: gathering statistics on the types of emergencies that are frequently producing results, finding the average number of results produced per execution, etc.

The dataset in Figure 4.10 summarizes the abilities and differences of the tools in the Active Toolkit. Data Channels and Procedures are both extended implementations of deployed jobs.

	Compiled Only Once	Can Run Continuously	Used For Ingesting Rapid Data	Allows Users To Share One Execution	Produces Individualized Results
Data Feeds	X	X	X	X	
Deployed Jobs	X				
Data Channels	X	X		X	X
Procedures	X	When Repetitive			

Figure 4.10: Comparison of Active Toolkit tools

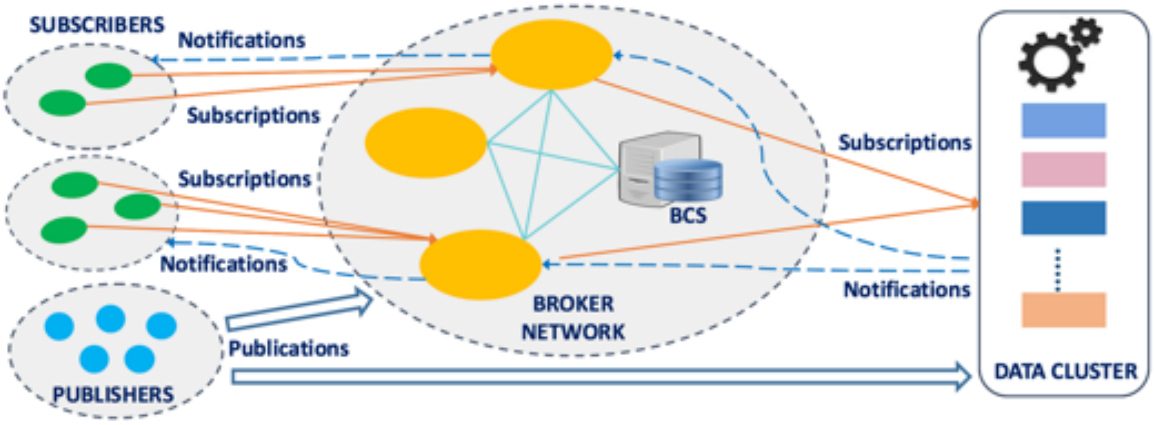


Figure 4.11: Communication in the BAD system

4.4 BAD Layers

As shown in Figure 4.11, there are three layers of communication for a BAD application. This provides a separation of concerns and allows each layer to perform tasks that are optimal for such a layer. Specifically, the Broker network is focused on efficiently handling both subscription communication and result delivery. Since the Broker network is structurally similar to a Pub/Sub Broker network, it can utilize state-of-the-art Pub/Sub scaling techniques and heuristics. On the other hand, the Data Cluster layer is built directly on a state-of-the-art Big Data Platform, and it can therefore capitalize on its capabilities (e.g., the distributed query engine). These details are discussed in further depth below.

4.4.1 Subscriber Network

When a subscriber joins the application, that user will be assigned a broker to communicate through. This assignment may change as the user moves or the broker utilization changes. The Application Administrator can decide to what level the subscribers

will be aware of the underlying channels and communications, e.g., whether they are choosing from a list of channels to subscribe to or simply registering *interests* using a higher-level interface.

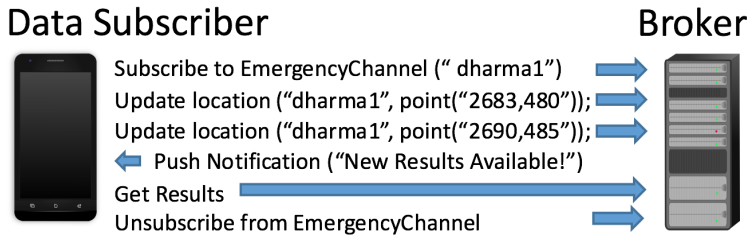


Figure 4.12: Communications between a subscriber's device and a broker

The communications between a subscriber's device and a broker are as follows:

1. The user's application will send a *subscribe* request to the broker (and will get back a subscription id that represents that subscription).

2. In the background, the application will continue to send the broker location updates.

3. When there are new results, the broker will send a *push notification* to the application.

4. The application will send a *get results* request to retrieve these results from the broker (including the subscription id).

5. When a subscriber is no longer interested in a channel, the device can send an *unsubscribe* request to the broker.

The interactions made by the user will ultimately be translated into requests made to the broker from the subscriber's device application. Figure 4.12 illustrates the types of

communication between a broker and a subscriber's ("dharma1") device. There are five main interactions that will occur between the broker and a subscriber to the *EmergenciesNearMe* channel. The reason for using a pull-based architecture is to allow the device application to be intermittently connected and to use its own heuristics (including network connection, battery power, etc.,) to determine when and how to fetch results. In addition, the application may use heuristics to tell the broker how it wants to be able to get results (e.g., if I have been disconnected for a long time, just give me the newest result), which the broker can use to manage result caching.

The end subscriber does not need to be aware of the data cluster at all, or of how the channels and locations are being maintained. This separation of concerns is repeated in the broker-to-data-cluster interaction.

4.4.2 Broker Network

The broker network is comprised of a scalable number of nodes, each designed to provide a one-to-many connection between the data cluster and the end user data subscribers. The broker network can capitalize on current Pub/Sub research for heuristics on subscriber distribution, subscription management, and result caching and delivery (as noted in Section 3.2.2). Such techniques are examined in [75] and lie outside the scope of this paper. This Section focuses on the communication layer between a broker and the data cluster.

Requests made by an end data subscriber are handled (directly or indirectly) by requests made by the broker to the data cluster controller, using the following operations:

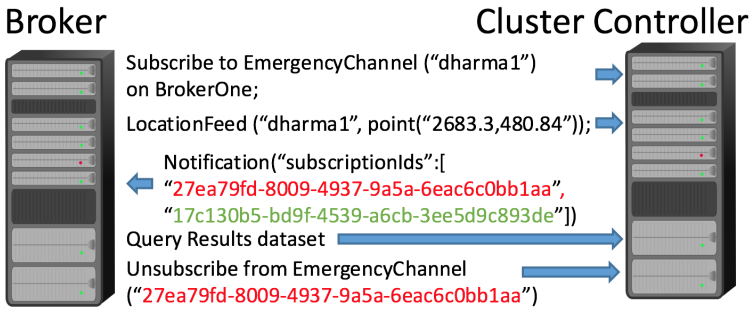


Figure 4.13: Communications between a broker and the data cluster

1. The broker sends a *subscribe* request on behalf of a user (again, getting back a subscription id).

2. The broker sends newly reported user locations directly to the *LocationFeed* data feed.

3. When the channel executes, the cluster sends a notification to the broker if there are new results, including the subscription ids for which there are new results.

4. The broker can run queries on the *Results* dataset to retrieve results.

5. The broker can *unsubscribe* on behalf of a user.

It is up to the broker to determine when and how to query the *Results* dataset based on the needs and availability of its users. In the simplest case, the broker can simply request results individually when they are requested by the user. The broker could fetch all of its users' results every time it gets a notification, so these results would be cached for when users request them. A broker could also opt to only request and cache results for users who ask for results often, and hold off on other results [75].

As an optimization, a broker could also share subscriptions among its users. For example, if there were a channel to find emergencies of a certain type in a certain city, there

might be several users subscribing to tornadoes in San Francisco. Rather than creating a new subscription per user, the broker could share a single subscription on the cluster for these users but provide end user subscribers with individual ids to communicate their requests to the broker (which will use the shared subscription on the cluster).

4.4.3 Data Cluster

At the bottom of the Active Toolkit's software stack, the data cluster has the advantage of treating everything as "just data". Static and dynamic datasets, subscription data, result data, and user data all end up as scalable, distributed data that can capitalize on AsterixDB as a performant Big Data platform. Hence the abstract notions of *subscribe* and *unsubscribe* are translated at this level into simple cases of `INSERT` and `DELETE`. The application administrator can also build indexes on the datasets involved in the query, including the *Subscriptions* and *Results* datasets themselves. Temporal indexes can help with fetching recent data. The execution of the channel then becomes an optimized, scalable Big Data join.

4.5 Experimental Evaluation

The separation of concerns between the data subscribers, the brokers, and the data cluster, allows us to separate their performance evaluations. For example, one could look at the end-to-end performance experienced by users, the caching and user distribution performance of brokers, or the data cluster itself. As this thesis is focused on the techniques and research of the BAD data cluster, the experiments focus on that layer. Specifically, the

evaluation looks at three performance aspects of the data cluster: Ingesting scalable data, processing channel jobs for a scalable number of users, and delivery of results to the brokers. This is a critical factor of the performance overall, as all other aspects of performance will depend on it. Since users are just "data" from the data cluster perspective, the end users can be essentially removed from the performance picture and treat the brokers as the "clients" of the BAD data cluster.

4.5.1 Experimental Setup

The experiments use the example application that has been discussed throughout this Chapter and the last, including its data model, datasets, feeds, and the repetitive EmergenciesNearMe channel with a period of 10 seconds (Figure 3.8).

In order to provide realistic movement models for people in the real world, the Opportunistic Network Environment (ONE) simulator [10, 55, 39] was used. The ONE Simulator allows for maps to be built representing real cities, including map graphs of pedestrian paths, roads, tram routes, etc., and for creating classes of "hosts" that can represent cars, pedestrians, and commuters by providing movement models and graphs for each class, thereby simulating a realistic flow of human movement within that city. The publicly available ONE Simulator comes with a pre-built simulation of the city of Helsinki (using the actual roads and metro routes), which was used for the experiments.

The simulator simulated a total of 501,000 users with different subsets of the total population being used for different experiments, broken up into the following groups: 167,000 pedestrians, 83,500 cars, and 250,500 metro commuters. The metro commuters are further broken into three equal groups using three different metro routes. Each user's

location was reported every ten seconds in the simulated world.

Hosts are also used to simulate the emergencies by creating a group of ten emergency creators with movement paths and speeds allowing them to traverse rapidly and randomly around the map between reported locations (simulated emergencies). They reported four potential types of emergencies: floods (with a radius of about 1/8th of the city and a probability of 50%), fires (with a radius of about 1/16th of the city and a probability of 30%), storms (with a radius of about 1/4th of the city and a probability of 10%), and car crashes (with a radius of about 1/100th of the city and a probability of 10%). Lastly, the hosts were used to create a set of statically located shelters in Helsinki by having them start at random points and recording their starting positions. A distribution of the 200 shelters is shown in Figure 4.14.

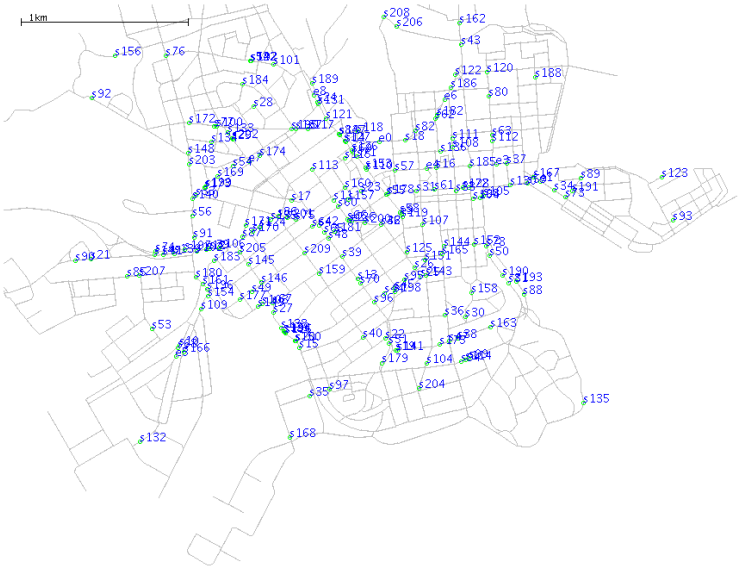


Figure 4.14: Distribution of 200 shelters in Helsinki

With this emergency-heavy scenario set up, which one might characterize as "Hell"-

sinki, the ONE simulator was run and the logs were then converted into data records in the AsterixDB data model (ADM) form. With these ADM files, the given "Hellsinki" scenario was replayed on the BAD data cluster by first loading all of the shelter data and then feeding the *UserLocations* and *Reports* in real time from the logged ONE data.

4.5.2 Data

To start with a non-empty state, the *Reports* dataset was preloaded with a history of over 2 million reports. Since the channel is index-optimized and uses only recent reports, this data size should not harm the performance of the channel, but is is still pre-loaded so as to show that this holds true. The *Shelters* dataset was also preloaded to contain the 200 shelters scattered across Hellsinki.

In the experiments, the rate at which reports arrive was varied from one per second to thousands per second. The user data was varied by changing the number of users for whom to send locations over the data feed. This was reflected in the sizes of both the subscriptions dataset (1 subscription per user) and the *UserLocations* dataset. Only the newest location of each user was recorded, so the stored dataset had one record per user in the experiments.

Figure 4.15 shows a snapshot in time of "Hellsinki". Here is shown the locations of 1000 users along with the locations of four emergencies that occurred. The dense areas of users are commuters on the city's tram routes.

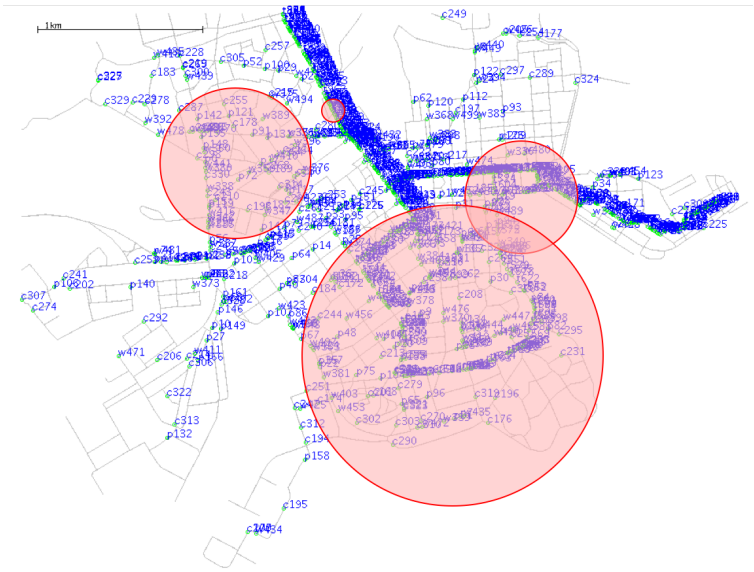


Figure 4.15: A BAD moment in "Hellsinki"

4.5.3 Building a Comparable Passive Platform

In order to support the same scenario using only passive AsterixDB, the broker would need to send explicit requests on behalf of the users. This is implemented, with the assumption that the broker has a dedicated thread that is tasked with getting the results for users (assumed for the BAD scenario as well). The broker thread could achieve the same goal as the BAD application by querying for nearby emergencies on behalf of every user one at a time within the same interval as the repetitive channel (10 seconds). The query to accomplish this is shown in Figure 4.16. The users can also be divided amongst multiple broker polling threads (each serving a fraction of the users) to allow passive AsterixDB to multi-thread the requests and scale more gracefully.

In the experiments BAD AsterixDB is referred to as the *active* mode, and vanilla AsterixDB the *passive* mode. Both modes do the same initial load on the *Shelters* and

```
use emergencyNotifications;

select r, shelters
from Reports r
let shelters = (select s.location from Shelters s where
spatial_intersect(s.location, circle("2437.3,1330.5 100.0")))
where r.timestamp > current_datetime() - day_time_duration("PT10S")
and spatial_intersect(r.location, point("2437.3,1330.5"))
```

Figure 4.16: The polling query for a single user (at location "2437.3,1330.5")

Reports datasets, and then start feeding the *Reports* dataset

In the active (i.e., BAD) mode the feed on the *UserLocations* dataset is also started. The experiments are based on measuring how well the channel is able to scale in terms of the number of supportable users, i.e., how many users and how much data it can handle within the desired inter-results period of 10 seconds (After that the system would be in "overload" mode since it fails to operate within the interval specified for the channel). To instrument the experiment, the total time (t1) is recorded for each execution of the channel ('Channel Execution Time', i.e. the time to produce results). After the broker receives a notification of results, the time (t2) spent to fetch the new results on behalf of the users ('Result Fetching Time') is also recorded. The total time (t3= t1+t2) for result creation and delivery must be less than the 10-second deadline. For each data setting, the times are averaged over many executions.

In the passive (i.e., user polling) mode, the broker starts to send requests by reading

logs of user locations and polling one-by-one on behalf of each user using the query in Figure 4.16. Each user is polled sequentially once per 10-second interval. The performance is measured based on how many user queries the poller can perform on average during this interval (averaged over many executions).

4.5.4 Cluster Hardware Setup

The experiments used six Intel NUC (BOXNUC5I7RYH) machines, each with an i7-5557U CPU processor (4 cores per machine), 16 Gigabytes of RAM, and a 1 TB hard drive. The BAD data cluster ran on three of these machines. The other three machines on the same local network acted as the brokers.

4.5.5 Feeds *vs.* Manual Inserts

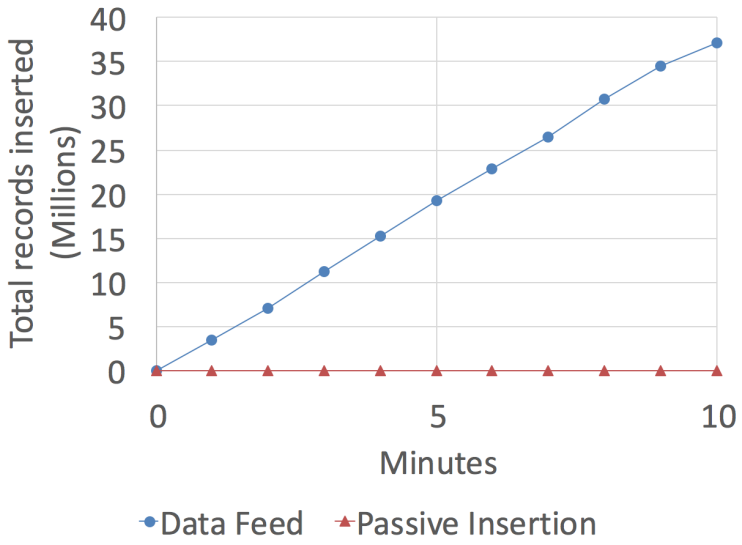


Figure 4.17: Feeds vs. Manual Inserts

Active data feeds were used for the insertions for both Reports and UserLocations. Data feeds are an important part of scalable channels, as they provide the mechanism used by the BAD platform to ingest data at scale. For illustration, Figure 4.17 provides a comparison of feed ingestion speed versus using traditional insert statements. In the feed case, a single thread sends data to the feed socket. For the insert case, there is also a single thread that continuously sends single-record INSERT statements to the Asterix HTTP API. After 10 minutes, the feed version can insert over 37 million records, while the passive insert version can just hit 5,000 records.

4.5.6 Channels *vs.* Polling

In order to see how the degrees of intersection between users and emergencies can affect performance, there are four scenarios used when comparing the active (BAD) and passive (polling) modes, namely:

- Case 1: A large percentage of users intersect with emergencies, and a large percentage of the emergencies intersect with users.
- Case 2: A large percentage of users intersect with emergencies, but a small percentage of the emergencies intersect with those users.
- Case 3: A small percentage of users intersect with emergencies, while a large percentage of the emergencies intersect with those users.
- Case 4: A small percentage of users intersect with emergencies, and a small percentage of the emergencies intersect with those users.

In order to demonstrate these scenarios, there are two other cities created outside of Hellsinki: Tartarusinki, where lots of emergencies will happen but (fortunately) no one resides, and Elysinki, where lots of people reside but (also fortunately) no emergencies ever happen. In all cases, the breaking-point analysis is used: For a given arrival rate of reports, how many users can the system serve within the 10-second window?

Case 1 - "Hellsinki" Alone

For Case 1, it is assumed that Helsinki is the only city (as in Figure 4.15). All emergencies take place there, and all users reside there. As the number of incoming emergencies per second increases, Helsinki gradually becomes more and more of an apocalyptic "Hell"-sinki.

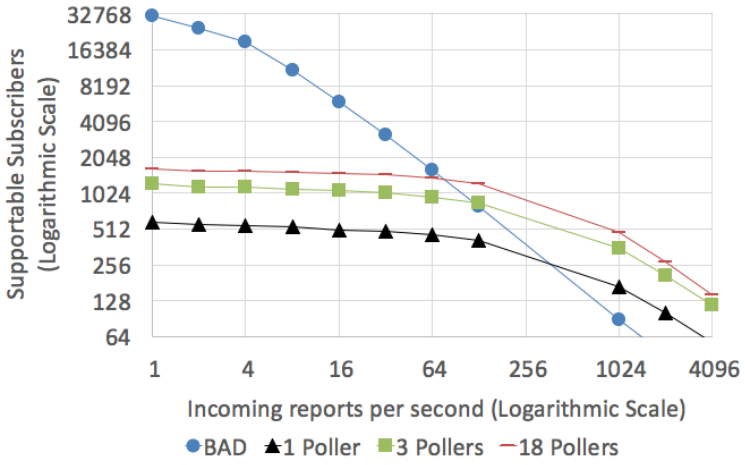


Figure 4.18: Case 1 - "Hellsinki" Alone

Figure 4.18 gives the performance results for this scenario. On the horizontal axis the rate of reports is varied. On the vertical axis the maximum number of users that can

be served while staying within the 10-second channel delivery deadline is shown. Note that both axes are on a logarithmic scale.

The passive mode shows the performance for three modes of execution: 1 poller, 3 pollers, and 18 pollers. Notice that the total number of users that can be served increases as the number of pollers increases. This is because AsterixDB is able to multi-thread the parallel requests to serve multiple queries at the same time. Also notice, however, that there are diminishing returns as the number of pollers increases. The bottleneck in the passive mode comes from the time that it takes to execute each request. The passive performance remains relatively stable on the left side of the graph. This is because the poll query execution time does not increase much until the amount of data processed by the query gets relatively large.

In the active mode, there are two main factors involved in the performance: the time to perform the channel job itself (including the time to store the results), and the I/O time to read (fetch) the result data by the broker. When the rate of reports is low, the result data per user will be small, so the number of users is the limiting factor (as all users will be joined with the recent reports each time). The maximum number of users that can be handled in Case 1 is around 32,000. When the rate of reports is high, the data volume per user is correspondingly high, so the number of supportable users is only a few hundred or less. The time spent writing and reading the user data is now the limiting factor.

Clearly, when there are less than 10 incoming emergency reports per second (already a high rate for a single city) the active mode is strikingly better, outperforming the passive mode by supporting up to 64 times (more than an order of magnitude higher) as

many users. The passive mode outperforms the active for much higher (and probably unrealistic) rates (hundreds of reports per second). This is discussed this further in Section 4.5.7.

Case 2 - Hellsinki and Tartarusinki

If Tartarusinki is added to the map and 90% of all of the emergencies occur there, all of the users will potentially receive notifications, but most of the emergencies that occur "worldwide" will not contribute to those notifications. Figure 4.19 shows this scenario.

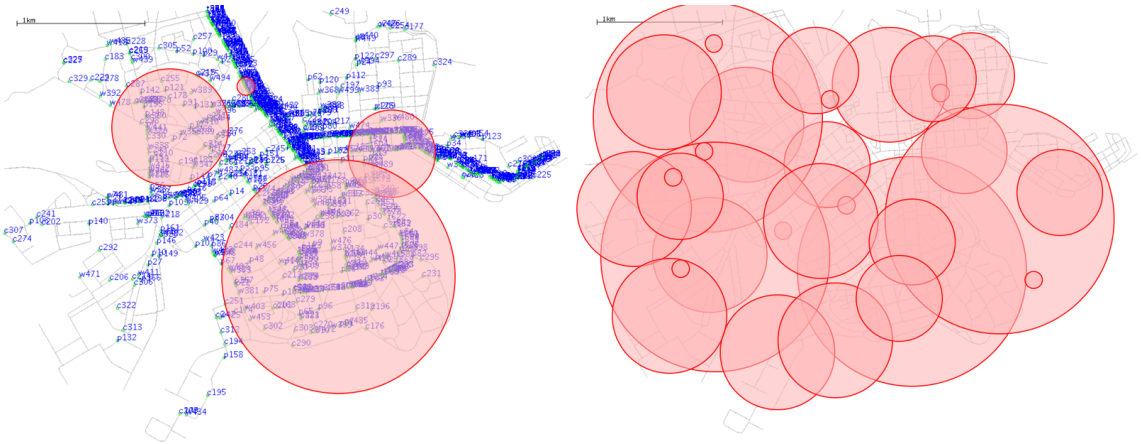


Figure 4.19: Hellsinki (Left) next to Tartarusinki (Right)

The performance results are depicted in Figure 4.20. When compared to Case 1, the same number of users and rate of emergency reports will produce a smaller number of results. Hence the active mode performs better until the rate becomes greater than 1000 reports per second. As in Case 1, for practical rates the active mode can support up to 64 times as many users as the passive mode.

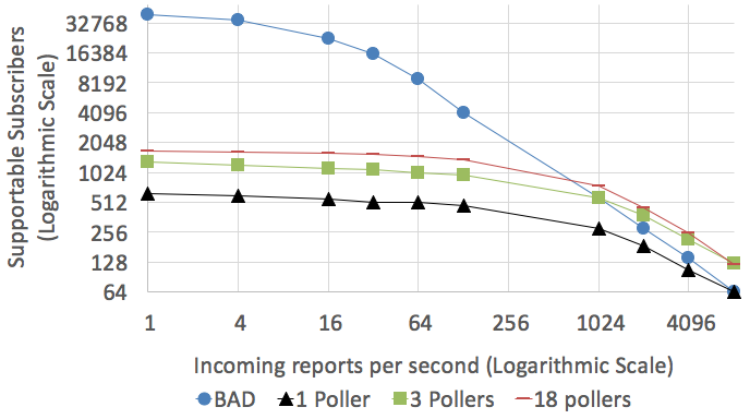


Figure 4.20: Case 2

Case 3 - Hellsinki and Elysinki

If instead Elysinki is added as the second city, all of the emergencies will be in Hellsinki. Since only 10% of the users are in Hellsinki now, most users will not receive emergency notifications. Figure 4.21 shows this scenario.

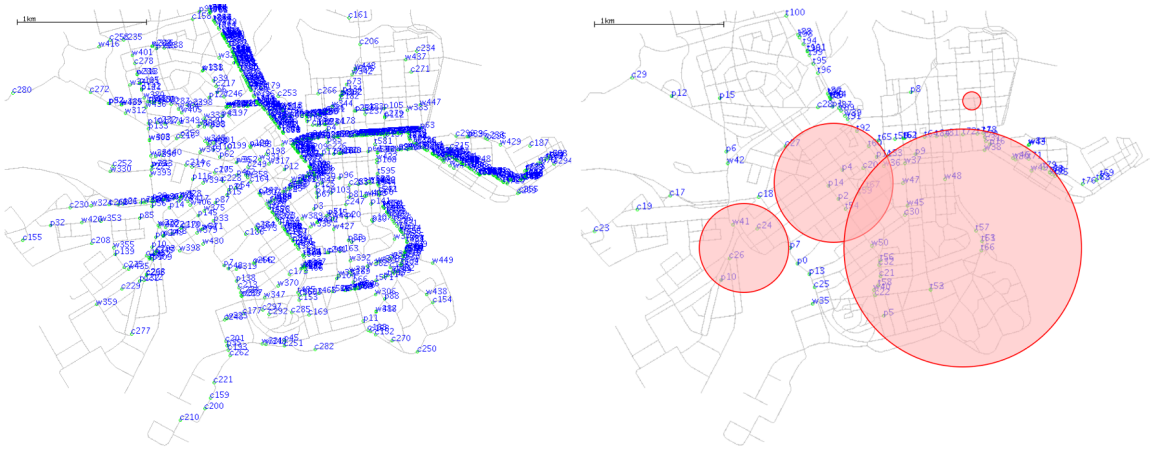


Figure 4.21: Elysinki (Left) next to Hellsinki (Right)

Figure 4.22 shows the related performance results. Although the users outside of

Hellsinki will be involved in the channel joins, they produce no results, and therefore they do not contribute to the size of the data, essentially becoming "free" from a results data perspective. Once again, the passive mode only slightly improves (the users with no results will have shorter queries). This leads to the active mode performing almost two orders of magnitude better than the passive mode (serving up to 260K users).

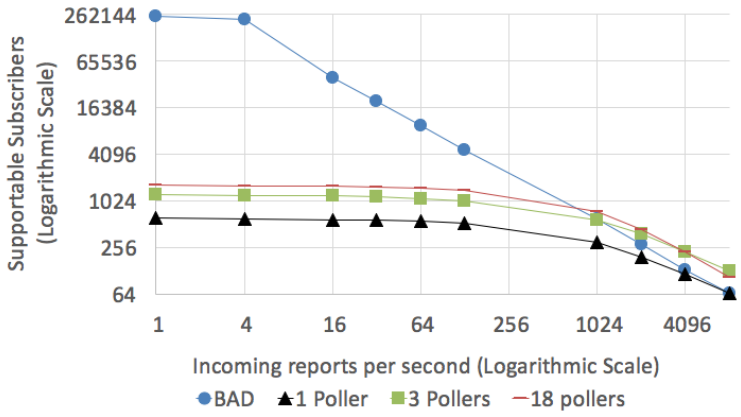


Figure 4.22: Case 3

Case 4 - All three cities

Lastly is simulated a world where 90% of the emergencies occur in Tartarusinki while 90% of the users reside in Elysinki, and an unlucky few remain in Hellsinki. In this case emergency notifications will exist for only 10% of both emergencies and users. Figure 4.23 depicts this scenario.

The active mode (Figure 4.24) continues to outperform the passive mode even when the disaster rate is scaled to thousands of emergencies per second. Again, for low report

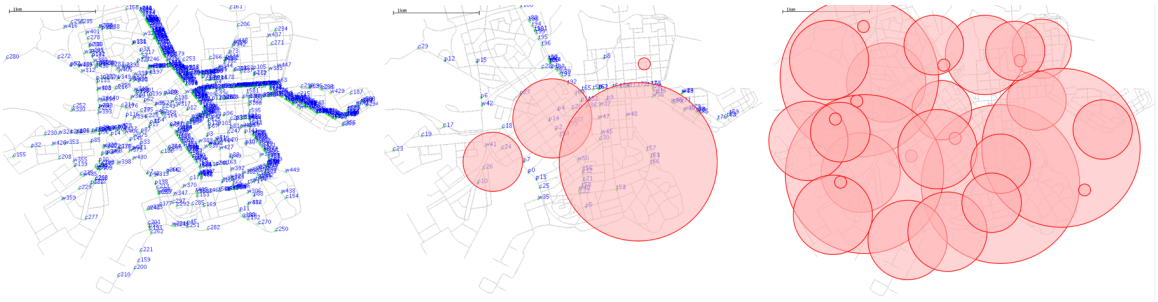


Figure 4.23: Elysinki(Left), Hellsinki(Center), and Tartarusinki(Right)

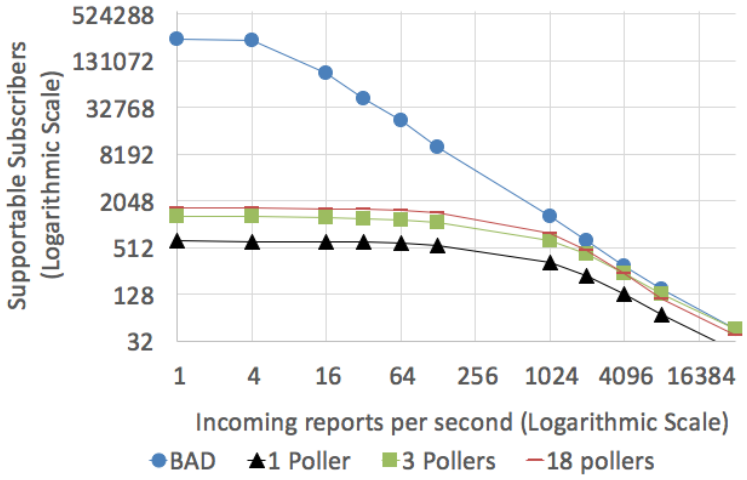


Figure 4.24: Case 4

rates the active mode provides more than two orders of magnitude better performance.

Discussion

All previous graphs were on logarithmic scales. The difference between the active and the passive modes can be seen more clearly in Figure 4.25, which plots the same graph for Case 4 but using linear scales. Finally, Figure 4.26 summarizes the active mode performance for all four cases. Case 1 represents the most 'apocalyptic' scenario where

all events and all users have the potential to intersect and thus the number of supported subscribers is the lowest. In contrast, in Case 4 few reports and users intersect, thus the system can serve a far greater number of subscribers.

Note that the situation where the passive mode behaves better implies hundreds of emergencies per second; this would create a practically unusable amount of data for each user. A user would be expected to read hundreds or thousands of results during one execution (every 10 seconds). In the more 'realistic' scenarios, where users might each get one or fewer notifications during an execution, the active mode can perform orders of magnitude better than the passive mode.

Lastly, it is important to note that in the active case, there is the potential to run other jobs in parallel to the channel execution and the result fetching. whereas the multi-threaded passive case bottlenecks the query throughput at all times, meaning that no other queries could be run on the system while maintaining this user scale. The multi-threaded passive case also bottlenecks the threads on the broker, preventing the broker from performing any other tasks. The active mode only uses at most one broker thread, and only during result fetching.

4.5.7 Effect of Result Size on the Performance

An important factor affecting the BAD system's performance is the result size per user. This is because the BAD platform performs an extra step of writing and reading the results: the results are first staged in the results dataset, then the same results are read from this dataset by the broker. Figure 4.27 (note the logarithmic y-axis scale) illustrates

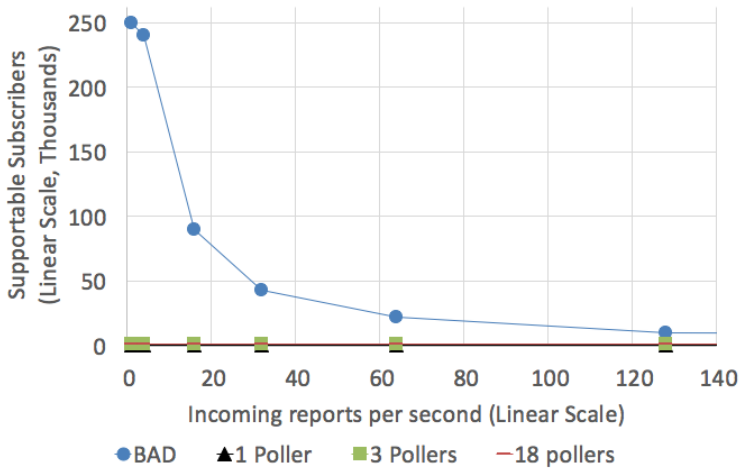


Figure 4.25: Case 4 on a linear scale

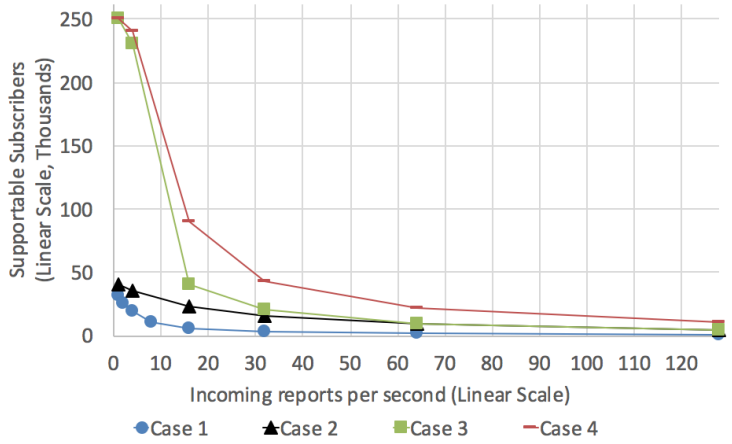


Figure 4.26: Active performance in the four cases

the result size per user for Case 1 and Case 4 as the number of reports per second increases. The 'apocalyptic' scenario of Case 1 produces hundreds and then thousands of results per user, whereas in Case 4 the results remain more than an order of magnitude smaller per user. This is consistent with the performance difference seen between these two cases in

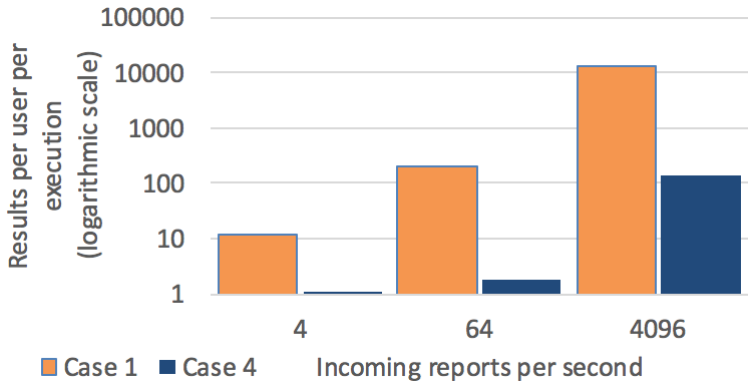


Figure 4.27: Result size comparison (Cases 1 and 4)

Figure 4.26.

Recall that the overall execution time consists of the 'Channel Execution Time' (t1) and the 'Result Fetching Time' (t2). Figure 4.28 and Figure 4.29 depict the relative percentage of time spent on each task, for Cases 1 and 4 respectively. In Case 1 the time taken to retrieve the hundreds of thousands of total results (i.e., across all users) accounts for 40-60% of the total time, whereas in Case 4 (Figure 4.29), where the result size per user is smaller, result fetching is less than 10% of the overall execution, thus allowing the system to serve more subscriptions.

The above observation further explains the scenarios where the passive mode behaves better. The passive mode avoids the extra step of staging the results, as the results of each polling query are sent immediately and directly to the broker. Since the amount of result data can become a limiting factor, and the aggregate result size grows with the report rate, the passive mode can perform better when there are many results (in the hundreds) per user per execution.

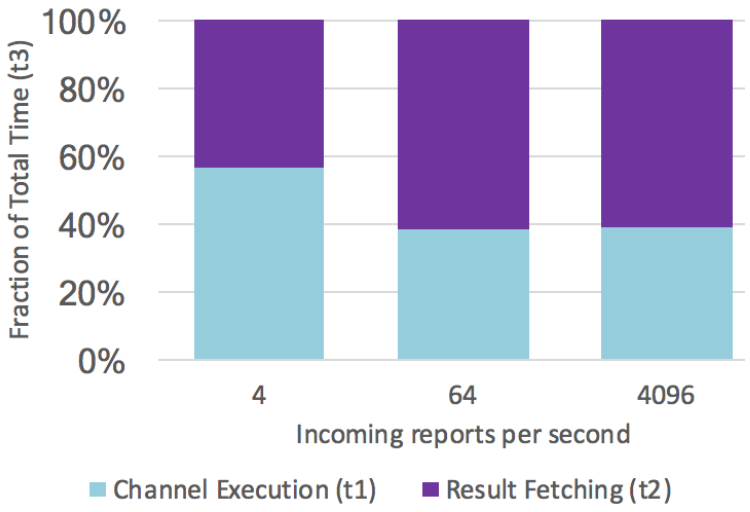


Figure 4.28: Fraction of Total Time spent on Channel Execution and Result Fetching (Case 1)

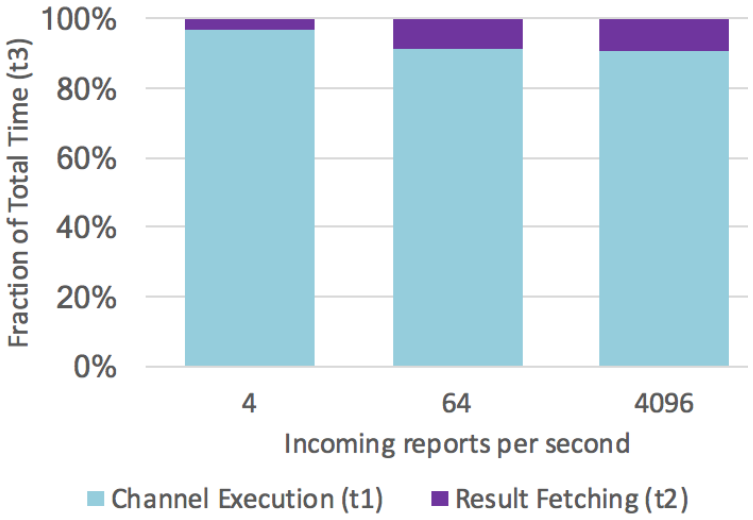


Figure 4.29: Fraction of Total Time spent on Channel Execution and Result Fetching (Case 4)

4.5.8 Increasing Pollers Ad Infinitum

Based on the fact that increasing the number of pollers increases the number of users that the passive case can serve, it might be assumed that the passive case can always do as well as the active case if there were enough enough pollers. Figure 4.30 shows that

this is not the case. The gains quickly flatline well before reaching 20 pollers. This is because AsterixDB (as with any Big Data platform) has limits on the amount of parallel query processing that it can do.

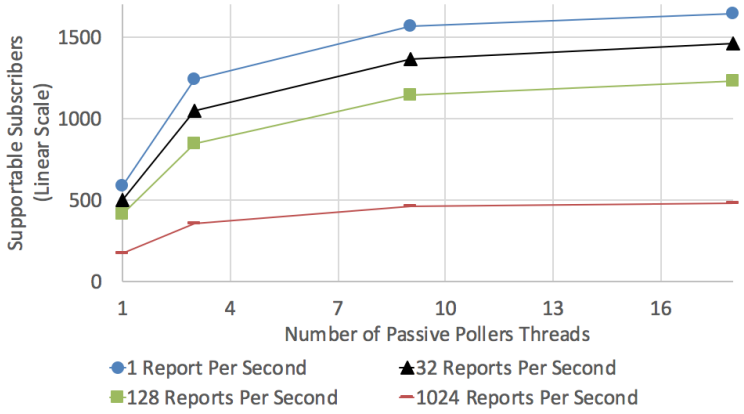


Figure 4.30: Supportable User gains as the number of polling threads is increased (Using Case 1)

4.5.9 Cluster Scaling Experiments

Speed-up: In order to show how the channel execution time can scale with the cluster size, additional experiments were performed on a 16-node cluster with dual-core AMD Opteron processors. Each machine has 8 GB of memory and dual 1 TB 7200 RPM SATA disks. The Case 1 scenario was used with 1000 user subscriptions. Sub-clusters of size 2, 4, 8, and 16 were used and the channel execution time (t1) was measured relative to the rate of emergency reports. The speed-up results are depicted in Figure 4.31. The channel execution scales well; for example, the 16-node cluster executes in half of the time of the 8-node cluster.

Scale-up: Using the same cluster and Case 1 scenario, the scale up was measured

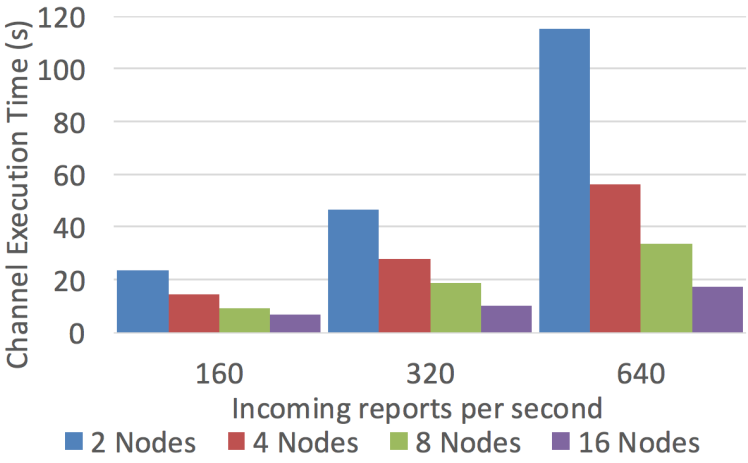


Figure 4.31: Channel speed-up for different cluster sizes

by fixing the report rate per node while increasing the cluster size, thus doubling the total report rate each time the cluster size was doubled The report rates per node were 160, 320, and 640. The results in figure 4.32 show that doubling the cluster size allows us to double the rate of reports without sacrificing channel execution time.

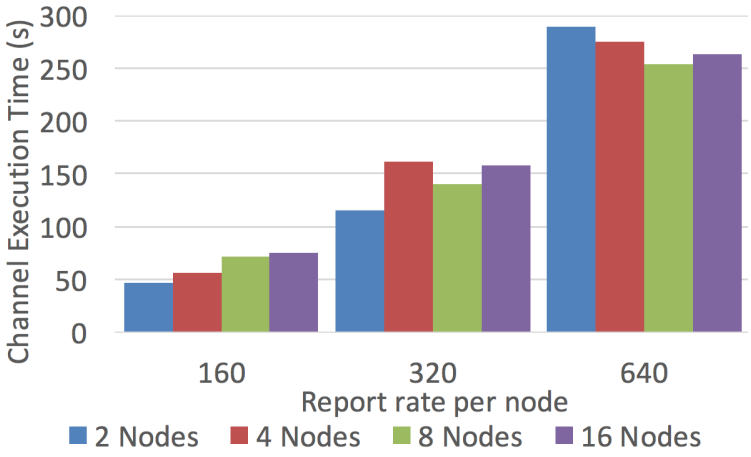


Figure 4.32: Channel scale-up for different cluster sizes

Chapter 5

From BAD to worse: Stability and Optimization of BAD

5.1 Introduction

Evaluating the first implementation of the BAD platform revealed some interesting research directions. In particular, this Chapter examines three topics. First we consider how to provide active recovery and continue active jobs in the presents of system failures. Section 5.2 discusses how this recovery was implemented, including allowing for the redeploying of a given deployed job. Second, we provide the motivation for and implementation of a push-based channel model in Section 5.2.1, which can improve performance when the channel result data is large. We include a performance comparison between push-based and pull-based channels using the scenario from Chapter 4. Third, we examine the advantage of sharing subscription data for users interested in the same information in Section 5.4.

We show the implementation of a redesign of subscription management in order to capitalize on common subscription needs, and provide a performance evaluation for this shared subscription model.

5.2 Handling system failures

Once a deployed job has been compiled and optimized, the specification for the job is cached on the cluster nodes to be re-used whenever the job is executed. This will work fine as long as no nodes ever have failures, but as soon as a single node is restarted this cached specification is lost on that node. AsterixDB is already able to handle failures like this from a data perspective, bringing the cluster back to a valid state after the cluster controller detects a failure. In order to allow BAD deployed jobs (including procedures and channels) to survive the same failures, the AsterixDB recovery manager needed a new task of redeploying these jobs.

The deployed job specification is the result of taking an initial INSERT statement (Figure 4.6) and compiling it into an optimal Hyracks job (Figure 4.7). The specification for this job is cached at each node, meaning that it will be lost when the node goes down. Although this loss occurs, the metadata for the channel or procedure is permanently stored on disk. In order to re-deploy these jobs, the original statement body (e.g., the SQL++ in Figure 4.6) is now stored as a string field in the metadata. When recovery is taking place, the statements for existing deployed jobs are parsed, compiled, and optimized, and then redeployed to the failed nodes. This way the job is once again ready to be executed as before. In the case of channels and repetitive procedures, the job period is also stored

in the metadata, so periodic execution of these jobs can begin again automatically as well, although there may be some down period of the job if the nodes were down for a significant amount of time. This allows BAD applications to continue through cluster failures without any need for application administrators to take actions.

5.2.1 Adapting to data changes

The advantage of compiling deployed jobs only once has one drawback: the jobs are optimized for the state of the data at the time the job was created. Changes to the data, specifically the addition of new indexes, may alter what would be considered the optimal DAG for performing the job. In order to allow automatic re-optimization of jobs as the system changes, a check is added at index creation for whether the new index affects the datasets used by any existing deployed jobs, which could allow the performance of the jobs to improve (the dependencies are stored in the metadata as well). If so, the job is re-optimized and redeployed to utilize the new index.

5.3 Push-based channels

When the number of subscription results per user becomes large (seen most dramatically in Case 1 of the experiments, Figure 4.27) the time taken for a broker to fetch the cached results from the results dataset takes up a large portion of the time (around 50% in Case 1, Figure 4.28). This provided motivation for adding a *push-based* channel model. Rather than sending the brokers a relatively small notification that there are new staged results, a push-based channel sends the new results directly to the brokers as they occur.

```
create repetitive push channel EmergenciesNearMe using
RecentEmergenciesNearUser@1 period duration(“PT10S”);
```

Figure 5.1: DDL to create a push-based channel using the function RecentEmergencies-NearUser@1

The modified channel definition can be seen in Figure 5.1. The addition of the keyword “push” lets the optimizer know what kind of channel this is.

Unlike the pull-based model, which has two steps of execution (channel execution and result fetching), the push-based model only has a single step, the channel execution, which includes the forwarding of the results directly to the brokers. This is the difference in message between “you’ve got results!” and “here are your new results!” Since the pull-based model had the disadvantage over passive polling that the results are read and written one additional time, the push-based model will excel in cases where the number of results is very large (as in Case 1). Note also that the push-based model does not need to maintain a results dataset. From solely a performance perspective, it might seem that a push-based model will always be the right choice. However, there are reasons why a pull-based model might be preferred.

The push-based channel does not guarantee delivery of the results to the broker. If the broker is down, the results will not be able to be sent. In a pull-based model, this would not cause a potential data loss problem, since the data will still be stored in the results dataset even if the broker communication failed. Since a push-based model does not store the results, there is no permanent version of results to fall back on after a broker recovers from a failure. It is worth noting however, that if the BAD platform uses a keep-everything

data model (as AsterixDB does), results may able to be reproduced manually by a broker simply by running an appropriate historical query. An advantage of a pull-based model is that it gives brokers the ability to optimize result fetching based on available resources and subscriber needs and connectivity. The push-based model instead forces the results onto the broker whether the broker wants them currently or not. Although these trade-offs should be considered, there are still many scenarios where a push-based model might be preferred, e.g., in RSS feeds or social media platforms where 100% delivery might not be critical.

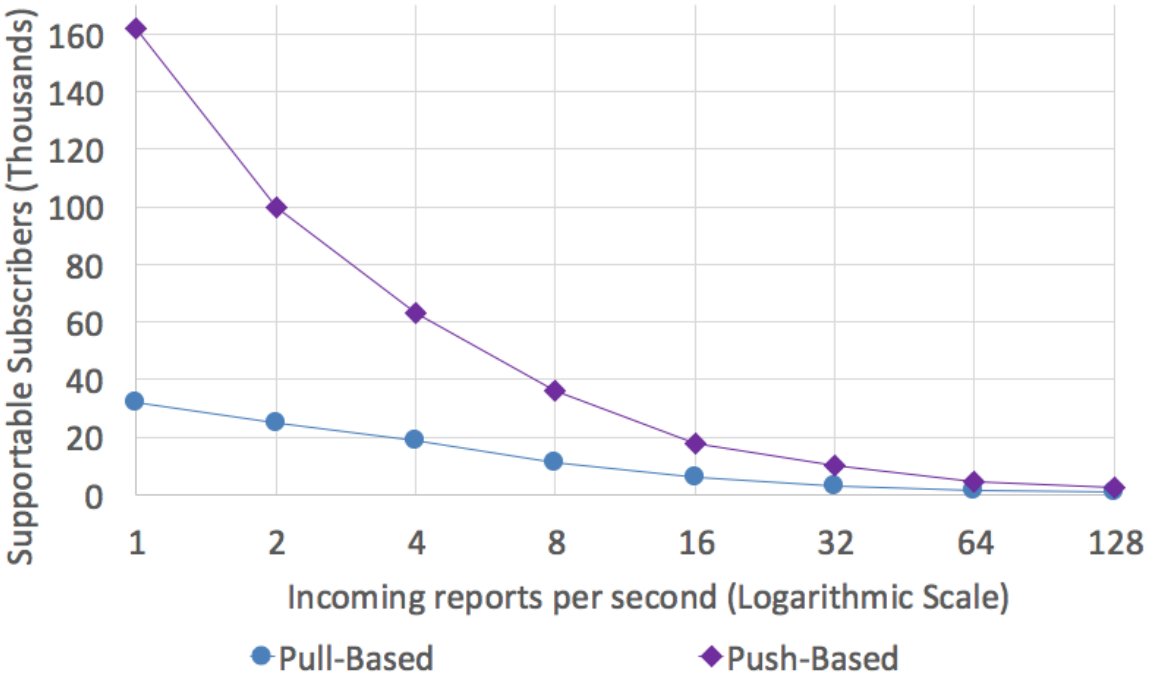


Figure 5.2: Push-based vs pull-based performance for Case 1: Hellsinki alone

In order to demonstrate the potential gains of using push-based channels, Figure 5.2 compares the pull-based data from Case 1 of Chapter 4 with the same data scenario using a push-based channel. Again the performance is measured based on how many subscriptions

can be handled while successfully meeting the 10-second channel deadline. The pull-based channel model was only able to handle a maximum of 32,000 subscriptions (when the report rate was only 1 per second). The push-based model is able to handle between 3X and 5X as many subscriptions, up to 162,000 subscriptions.

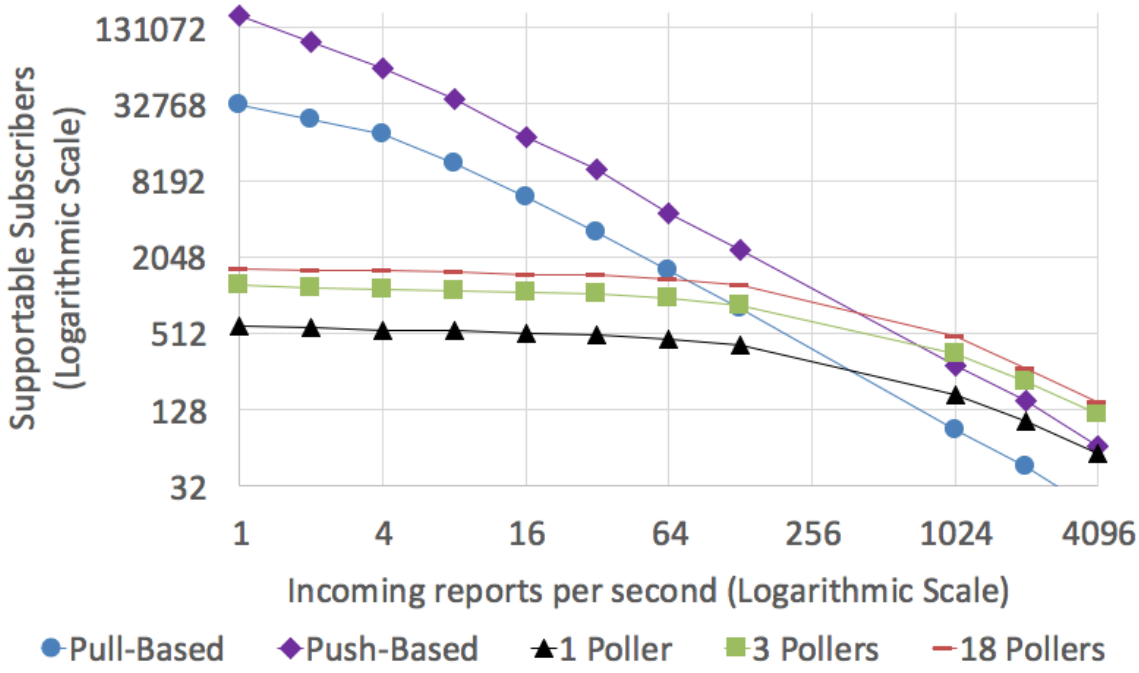


Figure 5.3: The Case 1 graph from Chapter 4 with the push-based channel added

Figure 5.3 shows the push-based channel in context of the original data from Chapter 4, including the passive polling data. As was shown in Figure 4.18, the pull-based model was outperformed by a single poller once the rate of incoming reports passed a few hundred per second. The push-based model, on the other hand, continues to outperform the data for 1 poller, even at 1000's of reports per second. It is also worth noting here that since the push-based model does not cache results, it should in theory be able to outperform all

of the passive results (not just the single-threaded one) since it is essentially delivering the same results but doing it in a batched manner. It is also worth noting that the push-based model eventually stops outperforming the multi-polling cases for very large rates of reports, suggesting there is still room for improvement (left as future work).

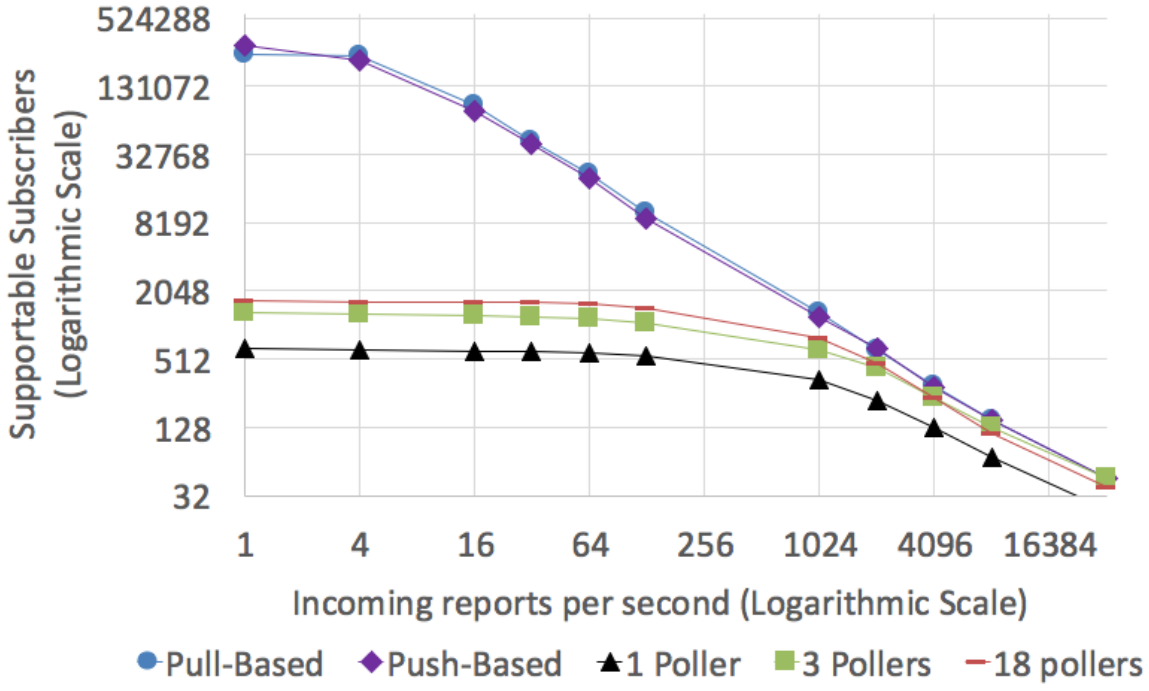


Figure 5.4: The Case 4 graph from Chapter 4 with the push-based channel added

Although, as expected, the push-based model has clear advantages when the result size is very large per user, this advantage might be negligible when the result size is small. Consider Case 4, where there were very few intersections between user and emergency report locations. In this case, the data per user was very small (Figure 4.27), and up to 97% of the time was spent on the channel execution itself (Figure 4.28). Figure 5.4 shows the push-based channel added to the results for this case. In such cases, the pull-based model might

be preferred for the advantages discussed previously (e.g., allowing the broker to decide when and how to fetch results).

5.4 Result sharing for common subscriptions

Although the BAD application discussed in Chapter 4 used a channel with parameters that would likely be different for each subscription (since the user id was a parameter), there are many channels that could be created where many users will subscribe with the same parameters. The most basic example, using the example emergency application, would be for users to subscribe to emergencies based on the type of emergency. In addition to the emergency itself, users may also be interested in the shelters that are nearby that emergency. The function and channel DDL and subscription statements in Figure 5.5 provide such a channel.

As many people may be interested in the same type of emergency, this channel will end up creating many records in the subscriptions and results dataset that are essentially the same. This is illustrated in Figure 5.6. All four subscriptions are for the same type of emergency, and the four results shown therefore contain the same report and shelter information. This incurs the following effects on channel performance:

1. The same parameter is joined with the report data four times.
2. The same information is stored four times in the results dataset.
3. When the broker fetches results, it will need to read essentially the same data four times.

```
create function RecentEmergenciesOfType(emergencyType) {
 (
    select report, shelters from
    (select value r from Reports r where
    r.timeStamp > current_datetime() - day_time_duration("PT10S")) report
  let shelters = (select s.location from Shelters s where spatial_intersect(s.location,report.location))
    where report.Etype = emergencyType
 )
};

create repetitive channel EmergencyTypeChannel using RecentEmergenciesOfType@1 period duration("PT10S");

subscribe to EmergencyTypeChannel("flood") on BrokerOne;
```

Figure 5.5: Channel for emergencies of a type

Subscriptions

subscriptionId	brokerName	parameter0
27ea79fd-8009-4937-9a5a-6eac6c0bb1aa	Broker1	"flood"
5b957e9a-17b5-4651-8a28-ec778db67af6	Broker1	"flood"
b634ae17-6b60-4a1d-b716-d080a1a76b07	Broker1	"flood"
17c130b5-bd9f-4539-a6cb-3ee5d9c893de	Broker1	"flood"
...	...	...

Results

subscriptionId	deliveryTime	result
27ea79fd-8009-4937-9a5a-6eac6c0bb1aa	2015-11-25 15:10:00	{"**Etype**":"flood", "**location**": circle("2719.47,967.5 1000.0"), "**shelters**": [{ "location": point("2872.98, 957.52") }, { "location": point("2913.07,954.89") }]...}
17c130b5-bd9f-4539-a6cb-3ee5d9c893de	2015-11-25 15:10:00	{"**Etype**":"flood", "**location**": circle("2719.47,967.5 1000.0"), "**shelters**": [{ "location": point("2872.98, 957.52") }, { "location": point("2913.07,954.89") }]...}
b634ae17-6b60-4a1d-b716-d080a1a76b07	2015-11-25 15:10:00	{"**Etype**":"flood", "**location**": circle("2719.47,967.5 1000.0"), "**shelters**": [{ "location": point("2872.98, 957.52") }, { "location": point("2913.07,954.89") }]...}
5b957e9a-17b5-4651-8a28-ec778db67af6	2015-11-25 15:10:00	{"**Etype**":"flood", "**location**": circle("2719.47,967.5 1000.0"), "**shelters**": [{ "location": point("2872.98, 957.52") }, { "location": point("2913.07,954.89") }]...}
...	...	...

Figure 5.6: Channel datasets where many users subscribe with the same parameters

These shortcomings can be overcome by instead aggregating these four subscriptions into one underlying channel subscription, and providing some way to identify this underlying subscription corresponding to a given broker subscription.

5.4.1 Potential data models

In the modern Big Data NoSQl world there are two basic ways to store a one-to-many relationship between two types of records.

Option 1: Flat data

There is the traditional, flat technique of having the primary key of one dataset be a foreign key to a second dataset. There could be one dataset of channel subscriptions and a second dataset of broker (end-user) subscriptions, where each channel subscription (e.g., "flood") may have several corresponding broker subscriptions (one for each user interested

in floods). In this case, when the channel executes, the (much smaller) channel subscriptions dataset is used to produce results, and these results may be fetched for individual broker subscriptions simply by joining the results with the broker subscriptions dataset.

Option 2: Nested data

The NoSQL alternative is to use a nested data model. In this case there would still only be one subscriptions dataset, but each record would have a nested object containing the information for each individual broker (end-user) subscription. In this case, the channel would work in much the same way as it did before, but the records for subscriptions being joined with emergency reports would be much fewer and much larger. When a broker fetched results, it would need to use the nested information in subscriptions to find which results are relevant to its end-user subscriptions.

The flat data model using two datasets was chosen here. The intention of sharing subscriptions is to enable hundreds or thousands or users to be able to share a single subscription. If the nested data model were used, the subscription records being joined with the reports would be huge objects, which is sub-optimal for performing joins. In addition, without array indexes (which are currently not available in AsterixDB) the job of finding results for a given broker subscription would also be sub-optimal.

Figure 5.7 shows the datasets and example data for a shared subscription model. In addition to modifying the data, the channel job itself will be slightly different to allow for the two subscriptions datasets. The modified job is illustrated in Figure 5.8.

Channel Subscriptions

channelSubscriptionId	parameter0
27ea79fd-8009-4937-9a5a-6eac6c0bb1aa	"flood"
5b957e9a-17b5-4651-8a28-ec778db67af6	"fire"
b634ae17-6b60-4a1d-b716-d080a1a76b07	"storm"
...	...

Broker Subscriptions

channelSubscriptionId	brokerSubscriptionId	brokerName
27ea79fd-8009-4937-9a5a-6eac6c0bb1aa	36aeaca4-f29d-ad0f-3c38-664b3c68fb74	Broker1
5b957e9a-17b5-4651-8a28-ec778db67af6	2faeaca4-e39d-ad0f-3d06-74cc528efa6c	Broker2
27ea79fd-8009-4937-9a5a-6eac6c0bb1aa	67aeaca4-fa9d-ad0f-3e37-536aa57a8f46	Broker1
27ea79fd-8009-4937-9a5a-6eac6c0bb1aa	1398cefc-6753-4365-c463-923f93225e33	Broker3
b634ae17-6b60-4a1d-b716-d080a1a76b07	2a0c39a4-0689-7930-37a3-6f71df17767e	Broker1
5b957e9a-17b5-4651-8a28-ec778db67af6	egaeaca4-779d-ad0f-3fad-11e094be1feb	Broker2
...		...

Results

channelSubscriptionId	deliveryTime	result
27ea79fd-8009-4937-9a5a-6eac6c0bb1aa	2015-11-25 15:10:00	{"**Etype**":"**flood**", "**location**": circle("2719.47,967.5 1000.0"), "**shelters**": [{ "location": point("2872.98, 957.52") }, { "location": point("2913.07,954.89") }]...}
5b957e9a-17b5-4651-8a28-ec778db67af6	2015-11-25 15:10:00	{"**Etype**": "**fire**", "**location**": circle("2208.19,2559.09 500.0"), "**shelters**": [{ "location": point("2023.99,2505.02") }]...}
27ea79fd-8009-4937-9a5a-6eac6c0bb1aa	2015-11-25 15:10:00	{"**Etype**": "**flood**", "**location**": circle("2830.75,1332.91 1000.0"), "**shelters**": [{ "location": point("2594.22,1078.11") }, { "location": point("2567.07,1104.86") }]...}
5b957e9a-17b5-4651-8a28-ec778db67af6	2015-11-25 15:10:00	{"**Etype**": "**fire**", "**location**": circle("2123.42,1297.09 500.0"), "**shelters**": [{ "location": point("2594.22,1078.11") }, { "location": point("2567.07,1104.86") }]...}
...	...	...

Figure 5.7: The channel datasets (with sample data) for the new subscription model

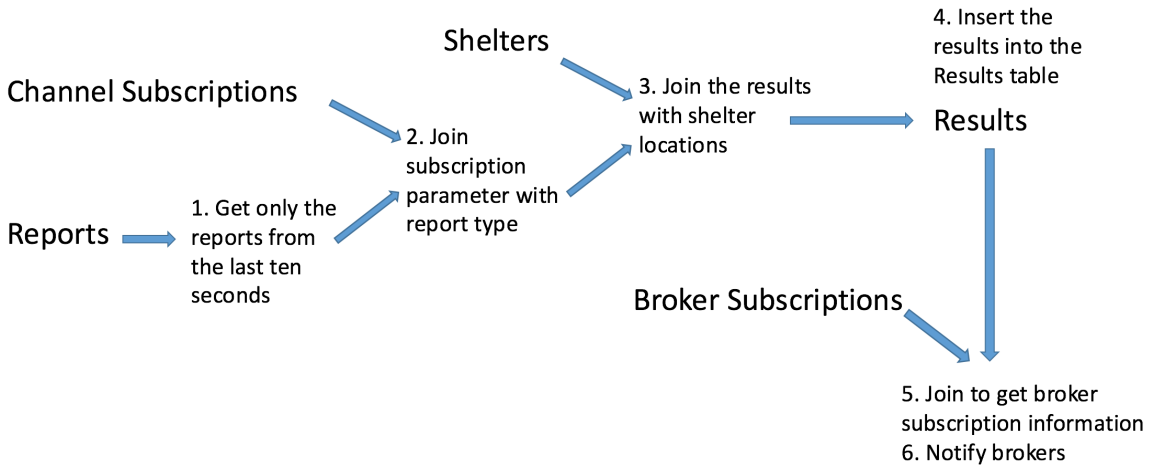


Figure 5.8: The deployed job for executing the channel EmergencyTypeChannel using the new subscription model

5.4.2 Preventing inconsistencies

Using a flat data model as shown, in an environment where multi-statement transactions are not allowed (as in AsterixDB) has the drawback that potential inconsistencies

could occur as subscriptions are added and removed. In order for channels to operate correctly, subscribe and unsubscribe operations are performed in the following ways:

Subscribe: A subscribe will be performed in the following two steps:

1. Query the channel subscriptions to see whether an existing subscription exists with the same set of parameters. If so, the table remains the same. If not, add a new record with the new set of parameters.

2. Insert into the broker subscriptions one new record, first querying the channel subscriptions dataset to find the channelSubscriptionId for a record with the correct parameters. Return to the subscriber the brokerSubscriptionId.

If step (1) happens concurrently the first time a new set of parameters is used, there could potentially be more than one record created for the same set of parameters. This is unfortunate, but not inconsistent, and unavoidable with current transaction limitations. Once the ongoing concurrent first instances are done, future broker subscriptions will always use an existing channel subscription.

Unsubscribe: This is done in a single step: Simply delete the record in broker subscriptions with the corresponding brokerSubscriptionId. At first it would seem prudent to check for channel subscriptions with no corresponding remaining broker subscriptions and delete them, but this is a case where not having a complete transactional model would make this potentially inconsistent by deleting a channel subscription at the same time a new corresponding broker subscription is being created. By leaving this step out, consistency is maintained, although the model is less clean. Essentially there will left over unneeded channel subscriptions for every unique set of parameters that has been used. This means

that the join with the channel subscriptions may do extra work and produce unneeded results. One way to avoid this would be to periodically perform a cleanup of the dataset, and prevent new subscription statements while the cleanup is being performed.

5.4.3 Evaluation of shared subscriptions

To demonstrate the advantages of sharing, an experiment was performed on the same cluster and using the same data setup as in the experiments in Chapter 4, but using the channel introduced in Figure 5.5. There were four types of emergencies for the experiments: fire, flood, crash, and storm. These types are equally distributed among the subscriptions for this experiment. This means that there will be four total channel subscriptions (one for each type), with each having a corresponding 1/4 of the total broker subscriptions.

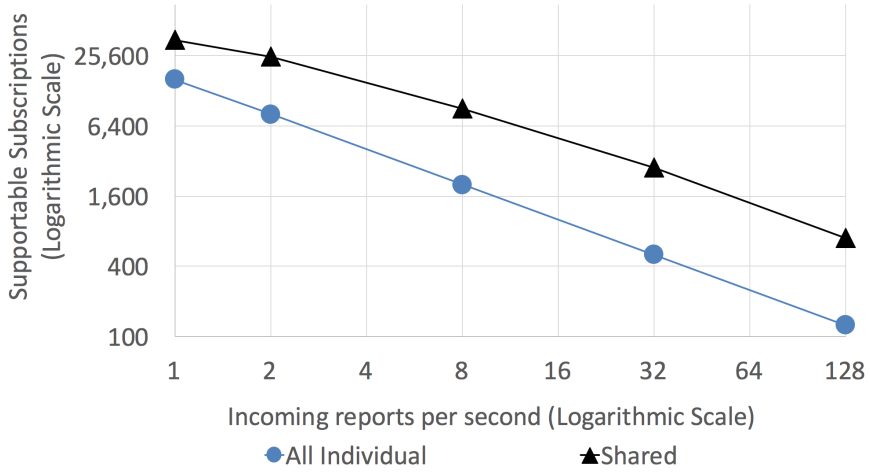


Figure 5.9: Channel performance of shared vs non-shared data models

As in Chapter 4, the number of supportable subscriptions in measured against the rate of incoming emergency reports. The old model of all individual subscriptions is

compared with the new shared subscription model. The new model is able to handle up to 5.6X as many subscriptions (Figure 5.9).

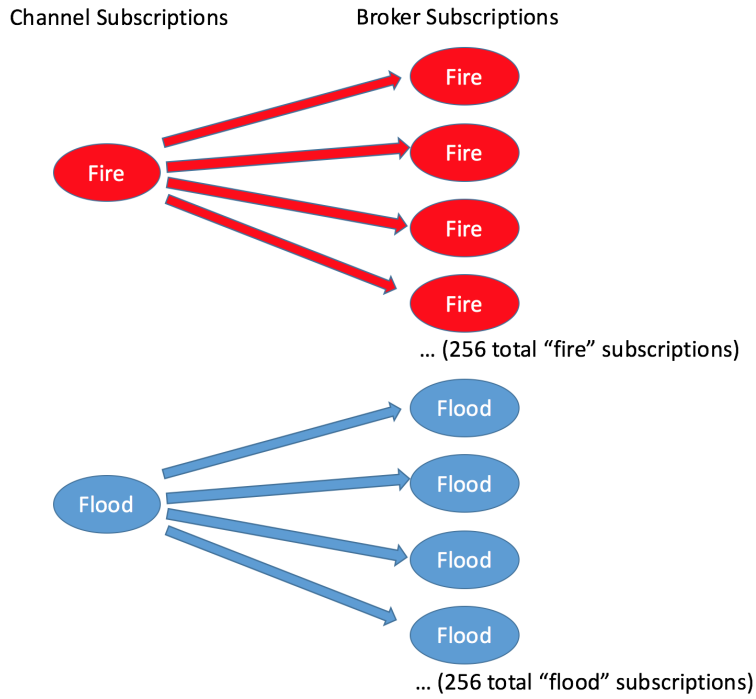


Figure 5.10: Complete Sharing: All subscriptions with the same parameter share a single channel subscription

To better see how the number of users that share a subscription impacts the performance, another experiment was performed to examine sharing in a constant data setting. In order to do so, the number of broker subscriptions and rate of reports were both held constant (1024 and 32 per second, respectively), but the number of channel subscriptions per emergency type was varied. This was done under the hood by artificially adding multiple channel subscriptions for the same type of emergency and sharing these equally among the broker subscriptions, e.g., 8 channel subscriptions for "fire" with 32 broker subscriptions

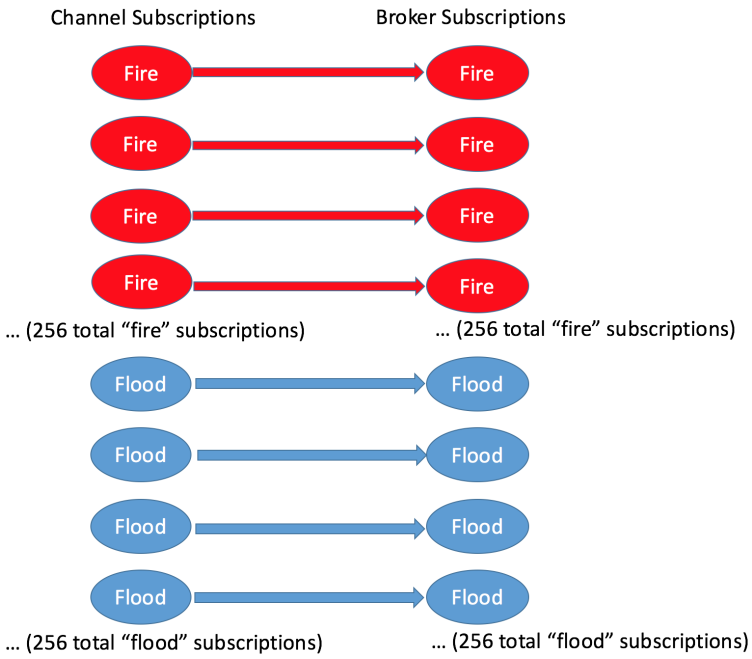


Figure 5.11: No Sharing: Each user receives a unique channel subscription

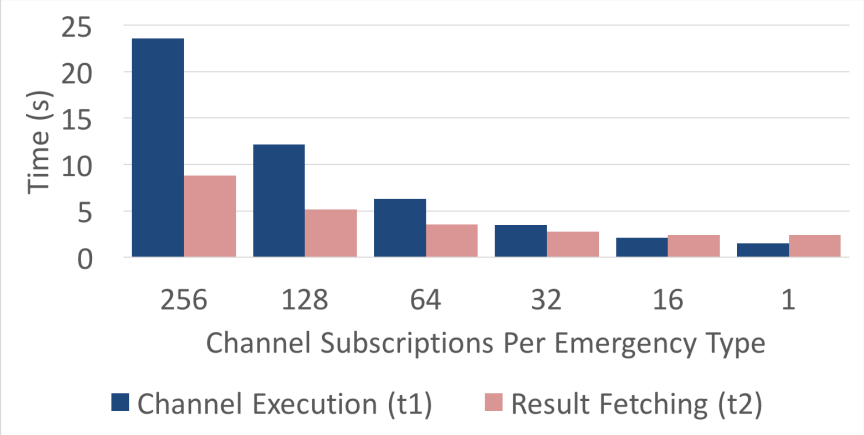


Figure 5.12: Channel performance of shared data models

each, for a total of 256 “fire” subscriptions (there are four types of emergencies for a total of 1024 subscriptions). The number of channel subscriptions per emergency type was varied between 256, 128, 64, 32, 16, and 1. In the lower extreme (complete sharing), all broker

subscriptions for a given type of emergency shared a single channel subscription (Figure 5.10). In the upper extreme (no sharing), each broker subscription had a unique channel subscription (Figure 5.11). Complete sharing is the actual result of the new data model, but the points in between illustrate the performance effects of sharing more fully. Both the execution time (t1) and the result fetching time (t2) are shown in Figure 5.12. The amount of time taken for each step begins to double as the number of channel subscriptions doubles.

5.4.4 Observations

It may at first seem that sharing should have an even larger impact on performance, e.g. if 1000 users share a subscription then the channel can handle 1000x as many users. This is not true for two reasons: First, when the channel executes, it still joins the data with the full broker subscriptions dataset in order to send the proper notifications to the brokers. Although this is done after the joins with reports and shelters are done, it still has an impact on the execution time. Second, when the broker fetches results, although it only gets one copy of each result, it needs to join these with the broker subscriptions to know which results are for which broker subscription, and therefore the records read will contain a list of brokerSubId values as well (a sample query is in Figure 5.13), although this is still far less data than having a duplicate of the entire result for every broker subscription. It is left as potential future work for the broker itself to store the relationships between broker and channel subscriptions, and thereby fetch a very small amount of data when the degree of sharing is very high.

```
select * from EmergencyTypeChannelResults r
let subs = (select value s.brokerSubId from EmergencyTypeChannelBrokerSubscriptions s
  where s.channelSubId = r.channelSubId and s.BrokerName = "my_name")
where r.channelExecutionTime = datetime("2018-11-19T23:49:23.938");
```

Figure 5.13: Broker query to fetch shared results

Chapter 6

Performance Evaluation of LSM merge policies

6.1 Introduction

Modern NoSQL systems use log-structured-merge (LSM) architectures [67] for maintaining high write throughput. To insert a new record, a WRITE operation simply appends the record in the MemTable (also called the in-memory component [28, 17]). Rather than requiring seeks for deletions/updates (as traditional relational data-bases would), the cost of a deletion/update is amortized by using sequential I/O. To achieve this, deletions are not in-place, rather, an 'anti-matter' record about the deletion is appended to the MemTable (transforming a deletion to a sequential insertion). Similarly, updates are not in-place. A record update is implemented as an 'upsert' which simply appends (i.e., sequential insertion) a record with the updated value.

When the MemTable reaches capacity, it is flushed to disk, to create an immutable file (an SSTable, also known as a disk component [28, 17]). This process can continue, creating many SSTables over time. Each READ operation searches the MemTable and SSTables to find the most recent value written for the given key. Hence, the time per READ grows with the number of components. (Bloom filters mitigate this effect.) In order to improve READ performance, the system periodically *merges* SSTables to reduce their number (and also reconcile any anti-matter records in these components). A *merge policy* (also known as a compaction policy) determines exactly when and how this is done.

Two variations of policies exist. In *level-based* policies ([67]), components have fixed size and are organized in levels. When a level is full (defined by a parameter), a component is selected to be merged with the components of the level below it. An example is LevelDB [38], where the highest level contains the fewest number of components, and each lower level contains more components than the level above it. Flushed records are moved first to the highest level. Read operations first check the MemTable, then check the components at each level, starting from the highest.

In contrast, *stack-based* (or "component-based") policies have a single stack of components, which may vary in size. When the MemTable is flushed to disk, it becomes the highest (newest) SSTable on the stack. Merges are triggered which combine several SSTables into a new, larger component. Several popular NoSQL systems today use stack-based merge policies. These include (among others) Bigtable [28], Accumulo [54, 68], AsterixDB [17], Cassandra [58], HBase [44, 56, 68], Hypertable [53], and Spanner [33]. This Chapter focuses on stack-based policies due to their popularity.

Although many NoSQL platforms use an LSM architecture, they generally use merge policies based on heuristics, which contain several tuning parameters. Though these policies may be intuitive, they are not generally based on a principled cost model, and are hence arguably suboptimal. As of the writing of this thesis, there has been no formal mathematical analysis of how to create an ideal merge policy. Even for a simple scenario with uniform insertions (i.e., with fixed Memtable size), no deletes, and a goal of limiting the maximum number of components to some k, the existing implemented merge policies provide no optimality analysis or guarantees.

In this thesis we implement and experiment with two recently proposed novel merge policies (MinLatencyK and Binomial) which have been built on a principled cost model. This cost model ([63]) makes two specific assumptions: first, that there is no shrinkage (i.e., it is assumed that the size of a newly merged component equals the sum of the sizes of the merged components), and second, that the read cost is proportional to the number of components. While no-shrinkage may seem to be a limiting assumption, it is a practical assumption as long as updates and deletes are relatively infrequent, and allows the merge cost to be predictable.

The key contribution of this chapter is the implementation on a common platform and experimental evaluation of existing policies (including those used by Google Bigtable [8] and Apache HBase [5]) and the two new proposed policies (MinLatencyK and Binomial). This is particularly important, given that *(a)* existing policies operate on different NoSQL systems, and *(b)* the proposed new policies were never implemented before (they were previously only theoretically analyzed).

This chapter also validates a merge cost model. Specifically, it verifies that the merge time is proportional to the sum of the sizes of the input components. Further, it shows that different policies will have relatively consistent read times for the same stack size (making them equally good choices from a read perspective, and allowing the comparison to focus on write cost), although the role of stack size on read time in real systems is not as direct as the one conjectured in previous theoretical work [63].

To create a relatively level playing field for all policies, this work focuses on the scenario where there is a maximum number k of allowed components, that is, going over k forces an immediate merge. This allows the experiments to be focused mainly on the write performance, which is shown to vary widely by policies. In contrast, the read cost varies much less when all policies share the same maximum k. The experiments support these observations.

The contributions of this Chapter are summarized as follows:

1. It validates a cost model for LSM merges, for the first time to the best of the knowledge of the author.

2. It implements several existing and two new stack-based merge policies on a common platform, specifically Apache AsterixDB [2].

3. It experimentally compares the various policies using the Yahoo! Cloud Serving Benchmark (YCSB) benchmark. The experiments show that the new policies significantly outperform the state-of-the-art policies.

4. It shows how a simulation framework, based on the cost model, can accurately estimate the performance of merge policies. This provides an opportunity to quickly test the

performance of new policies without having to implement them and evaluate them on a real system.

The rest of this Chapter is organized as follows: Section 6.2 provides background on LSM merging, while Section 6.3 describes the cost model and a formal definition for the problem of building a stack-based merge policy. Section 6.4 describes policies used in practice by systems today, policies that were implemented for the evaluation in this Chapter. Section 6.5 describes the two new proposed policies. The experimental results, on AsterixDB and on the simulator, are presented in Section 6.6. Section 6.7 presents concluding remarks.

6.2 LSM and Merges

The rest of this Chapter focuses on stack-based merge policies. A component (whether MemTable or SSTable) is typically implemented as a B^+-tree. Figure 6.1 illustrates an LSM *flush* operation. Once a *MemTable* (number 5) reaches capacity, its records are flushed to disk, producing the newest SSTable. Then a new (6th) component will be started as the new MemTable.

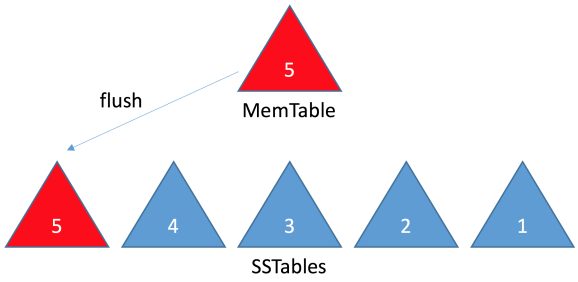


Figure 6.1: Flushing a MemTable to disk

As the number of SSTables increases, sequential groups of SSTables may be *merged* (Figure 6.2) together based on some *merge policy*. Note that new components always become the top of the stack (the left side in the figures), so the next MemTable flushed to disk would still be placed after component 5 (and become the 6th flushed component) in the Figure. This maintains a stack order that is based on freshness of records.

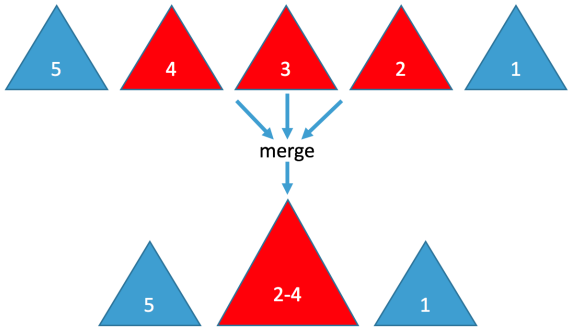


Figure 6.2: Merging a sequence of SSTables

It is important to note here that typical NoSQL systems (including the one used for the experiments) use upsert (insert if new, else replace) semantics, at least as an option if not as the default. This (combined with the fact that components maintain freshness order) means that there is no need for duplicate-key checking during insertion (which would incur an extra key lookup for every insert). Upserts can be handled simply as new inserts in the MemTable. Throughout this Chapter, "insert" will thus actually mean "upsert" semantics.

SSTables of an LSM-tree are immutable, meaning that deletes must be handled by adding 'anti-matter' entries into the MemTable. During merges, older versions of records can be ignored when creating the new SSTable, as can older records that are now marked as deleted.

During a query scan, resolution of deletes and updates can occur by using a heap

of sub-cursors sorted on (key, component id) where each sub-cursor operates on a single component. This allows records with duplicate keys to be resolved together (since only the version on the newest component is valid).

Successful key lookups can be further optimized. Since components are ordered on freshness, a key lookup simply accesses the components from newest to oldest, until the first instance (the newest version) of a key is found.

When compared to B^+-trees, LSM-trees make sacrifices on read performance in order to acheive scalable write throughput [51]. Specifically, the time per read will increase as the size of the component stack increases. This is mitigated by merges (because they reduce the stack size). As will be discussed fully in Section 6.3, merge (or "compaction") operations take time proportional to the size of the components being merged. This means that writes incur some amortized merge cost on the system. Any given merge policy will make some trade-offs between the total read performance and the total write performance.

6.3 Problem Definition

Two key operations in NoSQL databases are reads and writes of (key,value) pairs, where the value may consist of one or several attributes. For example, given a table of tweets, the key operations are *write(tweetid, {userid, body, time, ... })* and *read(tweetid).*

A read in a stack-based LSM system accesses one component at a time, starting from the top ones (most recent), until the searched key is found. Bloom filters are often used to minimize the number of non-matching components searched. When accessing a component, the system navigates the component's local index to find the search key. For

instance, in AsterixDB, each component is implemented internally as a B+-tree on the key attribute. Intuitively, as the component stack gets bigger, the read cost also increases, as the read may have to access more components to find a given key. This chapter focuses on policies that limit the read cost (as well as the maximum read latency) by ensuring that the stack size never exceeds a given parameter k.

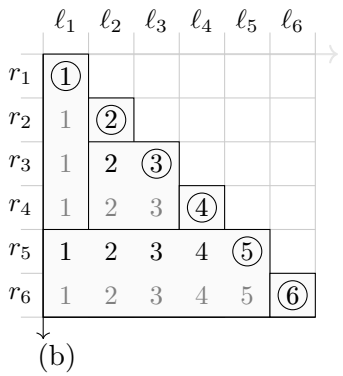
A write in LSM has a constant upfront cost, as it appends a record to the MemTable, which is periodically flushed to the disk. An additional significant indirect cost of writes is the merge (compaction) cost, which is proportional to the sizes of the components that are being merged. That is, $merge\_cost = \sum_{F \in \sigma_t \setminus \sigma_{t-1}} \ell(F)$, where $l(F)$ is the size of component F, summed over all components that existed before the merge but not after the merge (that is, the ones participating in the merge). This cost formula is validated in Section 6.6.2.

Stack-based LSM Compaction (SLC) Problem: The key decisions that a merge policy must make are (a) when to initiate a merge, and (b) which components should participate in a merge. The inputs are the writes (which lead to MemTable flushes) and the reads, and the goal is to find a balance between the cost of the merges and their benefit (the read cost decreases when more merges occur, given that merges decrease the size of the LSM stack). Below we give a formal definition of the k-SLC problem taken from [63].

Problem 1 (k-SLC) *The input is the parameter k and a sequence $\ell_1, \ell_2, \ldots, \ell_n$ of non-negative integers, where ℓ_t is the size (in bytes) of the t-th MemTable cache flush. A solution is a sequence $\sigma = \{\sigma_1, \ldots, \sigma_n\}$, called a* merge schedule*, where σ_t is a partition of $\{1, 2, \ldots, t\}$ with at most k parts, called* components*, and σ_t is refined by $\sigma_{t-1} \cup \{\{t\}\}$ (where*

t	stack at time t	time t merge cost	stack depth
1	$\sigma_1 = \{\mathbf{1}\}$	ℓ_1	1
2	$\sigma_2 = \{1\}\{\mathbf{2}\}$	ℓ_2	2
3	$\sigma_3 = \{1\}\{\mathbf{2}, \mathbf{3}\}$	$\ell_2 + \ell_3$	2
4	$\sigma_4 = \{1\}\{2,\ 3\}\{\mathbf{4}\}$	ℓ_4	3
5	$\sigma_5 = \{\mathbf{1},\ \mathbf{2},\ \mathbf{3}\ ,\mathbf{4}\ ,\mathbf{5}\}$	$\ell_1 + \ell_2 + \ell_3 + \ell_4 + \ell_5$	1
6	$\sigma_6 = \{1,\ 2,\ 3,\ 4,\ 5\}\{\mathbf{6}\}$	ℓ_6	2

(a)



(b)

Figure 6.3: **(a)** An eager, stable schedule σ ($k = 3$). **(b)** The graphical representation of σ. Each shaded rectangle is a component (over time). Row t is the stack at time t.

for convenience $\sigma_0 = \emptyset$). The length *of any component F is $\ell(F) = \sum_{t \in F} \ell_t$ — the sum of the lengths of the cache flushes that comprise F. The goal is to minimize σ's* merge cost*, $\sum_{t=1}^{n} \delta(\sigma_t, \sigma_{t-1})$, where $\delta(\sigma_t, \sigma_{t-1}) = \sum_{F \in \sigma_t \setminus \sigma_{t-1}} \ell(F)$ is the sum of the lengths of the new components at time t. This problem is illustrated in Figure 6.3.*

A schedule σ is *eager* if at each time t exactly one component F is new, where a component F is new if $F \in \sigma_t \setminus \sigma_{t-1}$. (Hence, in an eager schedule, at each time t, either the new MemTable flush creates its own component, or the flushed file is merged with some existing components into one new component.) A schedule is *stable* if every component is of the form $\{i, i+1, \ldots, j\}$ for some $i < j$ (so the components in the stack are always naturally ordered oldest to newest; this greatly eases the search for the most recent value inserted for a key). The rest of the chapter considers only stable policies.

A deterministic policy P for k-SLC is a function mapping each input instance ℓ to a schedule $\sigma = P(\ell)$. P is *eager (stable)* if every schedule it produces is *eager (stable)*. In practice, policies must be *online*, meaning that each σ_t is determined by k and $\ell_1, \ldots, \ell_t$, the flush sizes up to time t. An online policy is *static* if each σ_t is determined *solely by*

k and t, and is otherwise independent of the input. (In a static policy, the merge at each time t is predetermined — for example, for $t = 1$ always merge the top two components, for $t = 2$ merge just the flushed file, and so on.)

Model limitations: This paper focuses on policies that use a fixed stack size k to control read cost (and latency). A natural generalization, out of scope here, allows a finer accounting of read costs [63]. The experiments consider k values in the range 3 to 10 (many implementations in practice use k within this range). With this constraint on the stack size, read costs are expected to be similar across policies, allowing the focus to be on merge cost as the key performance differential.

Neither the model, nor the proposed policies, take into account the possibility that merging multiple components may produce a component whose length is smaller than the sum of the lengths of the merged components. Such "shrinkage" can be significant in workloads with many deletions or upserts of existing keys, which allow redundant records to be discarded.

Some systems (including AsterixDB) flush the cached MemTable (for the primary index key) when it reaches a preconfigured size, although very rare smaller flushes may be triggered. In this common case, ℓ will be generally *uniform* — each ℓ_t will be the same — and the proposed policies provide theoretical near-optimal absolute merge cost. Other systems (including Bigtable) flush the cache for a variety of other reasons not under the control of the merge policy, including cache pressure and inter-system coordination. In this more general setting (not tested here), the proposed policies ensure near-optimal worst-case cost by an appropriate theoretical metric [63].

Although the new policies outperform the existing policies in the experiements, more sophisticated models and policies may be developed in the future for other workloads and goals.

6.4 Existing Policies

There are several stable, fixed-k policies that are in common use with Big Data Management Platforms today. This chapter focuses on the following set of policies.

6.4.1 Constant Policy

The constant policy works as follows: when a flush occurs causing the stack size to reach k, merge all disk components into one. This eager, static policy has been shown in [18] to have a negative impact on ingestion performance since large merges are CPU and I/O intensive operations. In fact, its performance was a blocker for the first open-source release of AsterixDB.

6.4.2 Exploring policy

The exploring policy (the default for HBase) aims to find specific sequences to merge based on three parameters: λ (default 1.2), C (default 3), and D (default 10). If a valid sequence isn't found, the policy always ensures that the stack size never passes k. The exploring policy works as follows:

When a flush occurs, consider all contiguous sequences of components in the stack. Find all sequences s satisfying:

1. The number of components is at least C and at most D

2. The size of the largest component in s is at most λ times the sum of the size of the other components in s.

If there is at least one such sequence, merge the sequence with minimum average component size (if there are more than k components on the stack) or the longest sequence (otherwise). Otherwise, if there are more than k components on the stack, merge the sequence of C components having minimum total size. This policy is stable and online, but not eager.

6.4.3 Bigtable Policy

Like the constant merge policy, the Bigtable merge policy (used by Google's Bigtable platform) only performs merges when the stack size reaches k. However, rather than always merging all components, it tries to merge a sequence of newer components. It works as follows:

When a flush adds component F_t, causing $k+1$ total components in the stack, merge the i newest components (including F_t), where $i \geq 2$ is the minimum such that, afterwards, the size of each component F exceeds the sum of the sizes of all components newer than F. This policy is stable, eager, and online.

6.5 Proposed Policies

This Section provides a brief overview of two new merge policies that are formally proposed in [63]. For the full motivation and details of these policies, please see [63]. The

key intuition of these policies is that they represent a merge schedule as a binary search tree whose structural properties correspond to the cost of the schedule. The two policies generate the optimal such binary search tree, each under slightly different assumptions: MinLatencyK only focuses on minimizing the overall merge cost (and hence maintains a stack size close to k in the steady state), while Binomial also tries to keep the stack size relatively small to facilitate faster read operations. These policies are stable, eager, and static.

Both policies utilize the function $B(d, h, t)$ (for $0 \leq t < \binom{d+h}{h}$) defined by the recurrence

$$B(d,h,t) = \begin{cases} 0 & \text{if } t = 0, \text{ else} \\ B(d-1,h,t) & \text{if } t < \binom{d+h-1}{h}, \\ 1 + B(d,h-1,t-\binom{d+h-1}{h}) & \text{if } t \geq \binom{d+h-1}{h}. \end{cases}$$

6.5.1 MinLatencyK policy

In response to a flush at time t: Let $d' = \min\{d : \binom{d+k}{d} > t\}$ and $i = B(d', k, t)$. Merge the i-th oldest component with all newer components (resulting in stack size i).

Note that if the stack has length less than k just before flush t adds component F_t, then MinLatencyK just puts the flushed file on the top of the stack. That is, if $|\sigma_{t-1}| < k$, then $\sigma_t = \sigma_{t-1} \cup \{F_t\}$.

6.5.2 Binomial policy

In response to a flush at time t: Let $T_k(d) = \sum_{i=1}^{d} \binom{i+\min(i,k)-1}{i}$. Let $d' = \min\{d : T_k(d) \geq t\}$. Let $i = 1 + B(d, \min(d,k) - 1, t - T_k(d-1) - 1)$ for B as defined above. Merge

the i-th oldest component with all newer components (resulting in stack size i).

6.5.3 Discussion

The Binomial policy is designed to improve read performance by producing a schedule σ that minimizes the total merge cost *plus* the sum of all the stack sizes (that is, plus $\sum_{t=1}^{n} |\sigma_t|$). This is chosen to roughly minimize the total (worst-case) computational costs of merges *plus* reads, assuming one read per merge. In the range $n \ll 4^k$, Binomial achieves a smaller average stack size than does MinLatencyK, with only a small increase in merge cost. In this range, Binomial guarantees a maximum stack size of about $\log_2(n)/2 \leq k$, and a worst-case total merge cost of at most about $(\sum_t \ell_t) \log_2(n)/2$. In the range $n \gg 4^k$, no such optimization is possible. There, in order to achieve near-optimal merge cost (e.g., for uniform ℓ), any schedule *must* have an average stack size near k. In this range, Binomial behaves similarly to MinLatencyK. In contrast to MinLatencyK, it is reasonable to use Binomial with $k = \infty$ in the non-fixed-k setting.

6.6 Experimental Evaluation

The following experiments will compare the two new merge policies with the policies discussed in Sections 6.4.1 to 6.4.3. In the experiments, the AsterixDB data feed is used as the insert mechanism for the dataset, to handle rapid ingestion of data. The feed uses an upsert data model, meaning that records will be ingested without requiring a duplicate key check, making ingestion faster. This model also makes feeds more robust for "at least once" data sources.

6.6.1 Experimental Setup

The experiments are run on an Intel i3–4330 CPU (4 cores) machine running CentOS 7 with 8 GB of RAM and a 1TB hard drive. The RAM was reduced to 8 GB in order to make the data scale grow larger than available memory quickly. Since the experiments focus on how merge policies affect the performance of the writes on a single dataset, a single dataset is created with a single AsterixDB data feed for each experimental run. The size of the MemTable is 4 MB (The default used by HBase). This gives about 3,000 maximum records for the MemTable.

The experiments for each policy run with values for k as 3, 5, 6, 8, and 10. The default values are used for all other parameters (as discussed in Section 6.4) with one exception. For the Exploring policy, C is set to 2 (rather than the default of 3) when k is 6 or less. This gave the policy better performance and a fair comparison should use the best performance for each policy.

The Yahoo! Cloud Serving Benchmark (YCSB) [32] was used to generate the workload. Each experiment ran for a total of seven hours. The first five hours was spent doing purely writes as quickly as possible. For the last two hours, the operations are split between reads and writes with 99.95% of operations being writes (in order to make the number of flushes be on the order of the number of reads, and allow for reads to be measured on many different stacks).

6.6.2 Model Validation

Before comparing the policies, it is important to verify that the assumption from Section 6.3 holds true: Namely, the cost (in time) of a merge should be proportional to the total size of the components being merged. Figure 6.4 gives the time spent for a merge vs the total size of the components before being merged for a series of merge operations during the experiments. Indeed, the time spent grows proportionally to the total size. (The R^2 value is shown.)

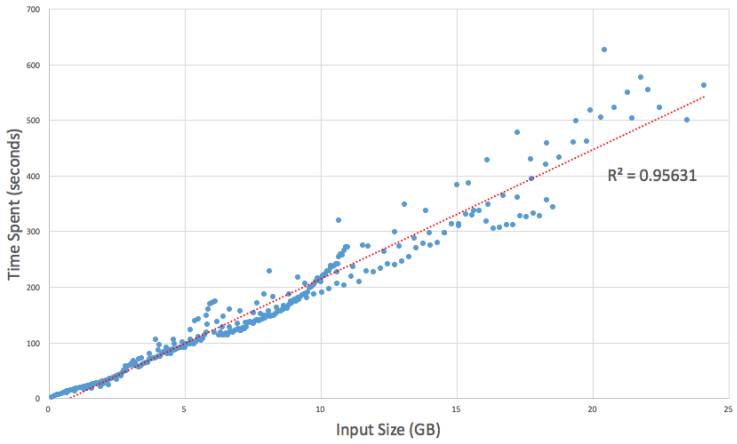


Figure 6.4: Merge Time vs Total Size of Components Being Merged

Comparing policies based on their amortized write costs alone misses part of the full picture. In systems with frequent reads, it is important not to sacrifice read performance in exchange for better write performance. As such, the policies compared should have similar read performance for the same YCSB workload and maximum stack size k. Figure 6.5 shows the average read time for a random key lookup vs k for the experiments, excluding reads that occurred during merge operations (to ignore large outliers when the system resources

are occupied with a merge).

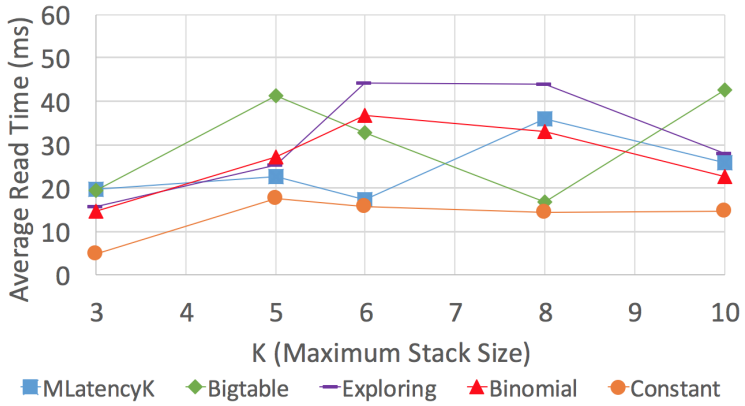


Figure 6.5: Average Read Time vs k (Maximum Stack Size)

As seen in the figure, no realistic policy has a clear advantage over the others for average read time. The constant policy seems to give slightly better read times. This is likely because components have the longest lifetime with the constant policy, so caching is helping a little more. It will be seen, however, that the constant policy is always the worst choice from a write cost perspective (in fact it is not used in practice by systems). Note also that there is no obvious correlation here between read time and k, which agrees with the previous theoretical work [63].

6.6.3 Experimental Evaluation of Merge Cost

The amortized write cost of a policy can be seen by the accumulated cost (in total bytes merged) of merges over a workload of writes. Amortized cost is measured fairly across policies by recording the accumulated merge cost vs the number of flushes that have occurred. Figures 6.6 through 6.10 show the results over several experiments, with the

maximum stack size k set to 3, 5, 6, 8, and 10 respectively.

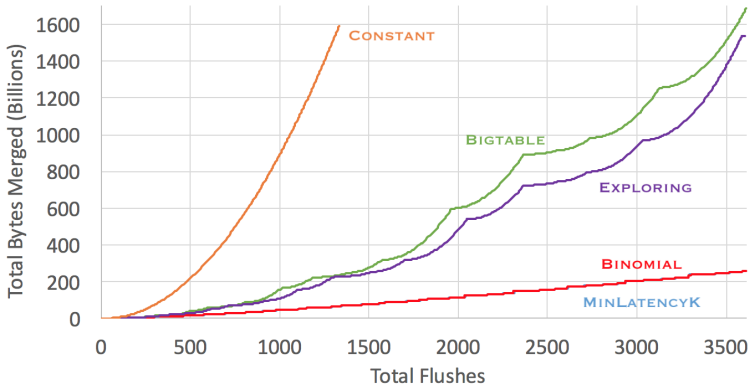


Figure 6.6: Merge Cost, $k = 3$

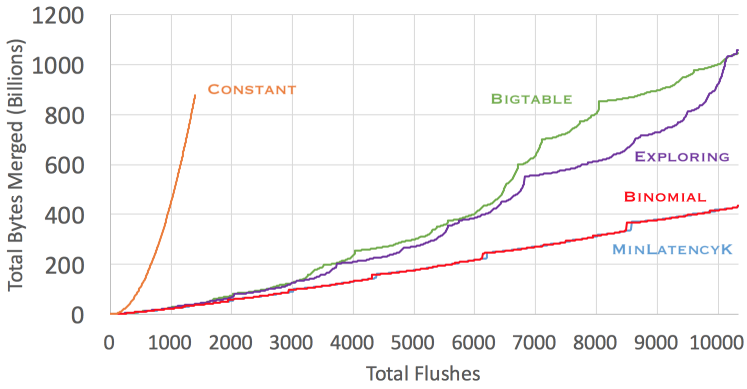


Figure 6.7: Merge Cost, $k = 5$

Discussion The first thing to be noted from the data is that the two proposed policies (MinLatencyK and Binomial) give better performance overall than the Bigtable and Exploring policies. In some of the graphs, the two lines are indistinguishable. The merge cost

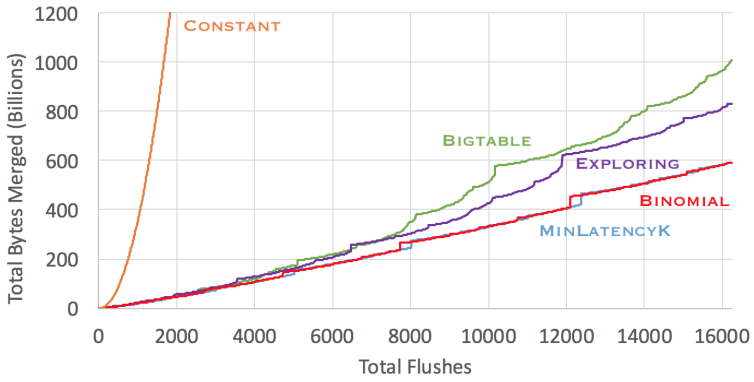


Figure 6.8: Merge Cost, $k = 6$

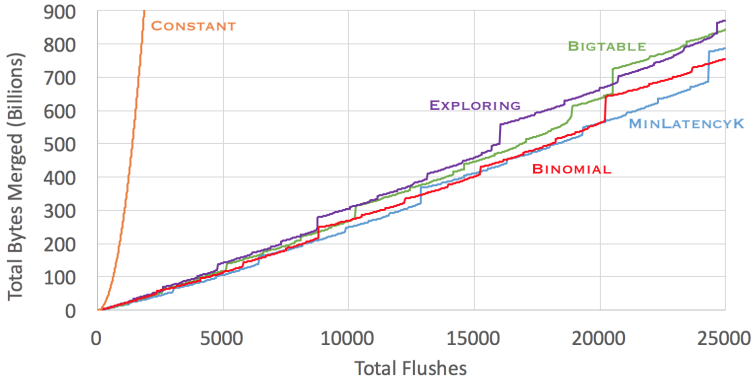


Figure 6.9: Merge Cost, $k = 8$

of the MinLatencyK and Binomial policies appears to be the same for small k (although they might differ slightly at the times of large merges), but starts to diverge when k gets larger. This is because the Binomial policy is specifically designed to balance read and write costs, so it will not utilize the entire potential stack size until the data grows to a certain point (around $n = 4^k$ flushes, $1,000,000$ flushes in the case of $k = 10$). The difference between the proposed and existing policies is also less clear for higher values of k. These algorithms are designed to work with data that is growing over weeks and months, but the experiments are running over hours, so the complete asymptotic behavior for higher values

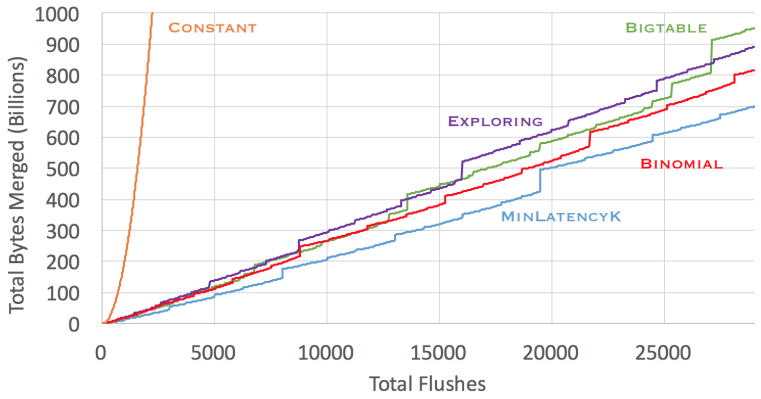


Figure 6.10: Merge Cost, $k = 10$

of k is not seen.

6.6.4 Simulated Merge Cost

Because these merge policies follow precise mathematical models, only depending on the input of a set of flushed components for their behavior, these models can be used to show how these policies will act after 10's or 100's of thousands of flushes. A Python-based simulation was used to better show the long term performance of these merge policies. In order to verify the simulation accuracy with the real data, Figure 6.11 shows the simulated performance for the policies for k = 3 up to 3500 flushes. Note that this figure matches well with the actual results in Figure 6.6. There will be slight differences in individual points between the two graphs because the simulated version assumes no shrinkage and precise uniform flush size, but the Big-O performance and overall merge cost matches well with the actual data.

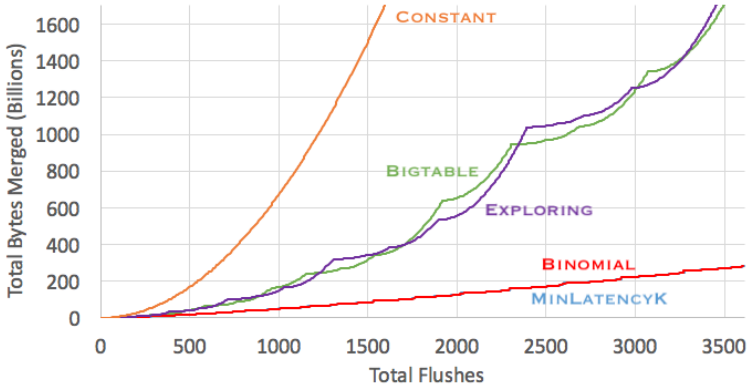


Figure 6.11: Simulator Validation, $k = 3$

Simulation Results The simulator can show how the policies will perform in the long term for k values of 8 and 10. For $k = 8$, the difference can be seen once the number of flushes is shown through 100,000. This can be seen in Figure 6.12. In the case of $k = 10$, the change is not obvious until the number of flushes approaches 1 Million (Figure 6.13). Once again, the proposed policies perform asymptotically better than the existing policies, as well as being asymptotically the same as each other.

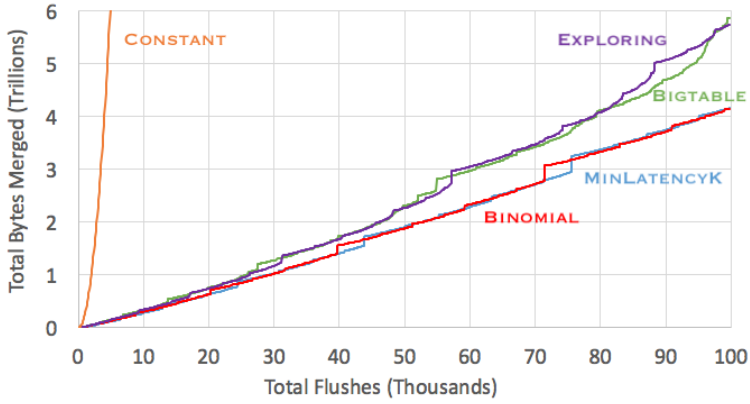


Figure 6.12: Simulated Merge Cost, $k = 8$

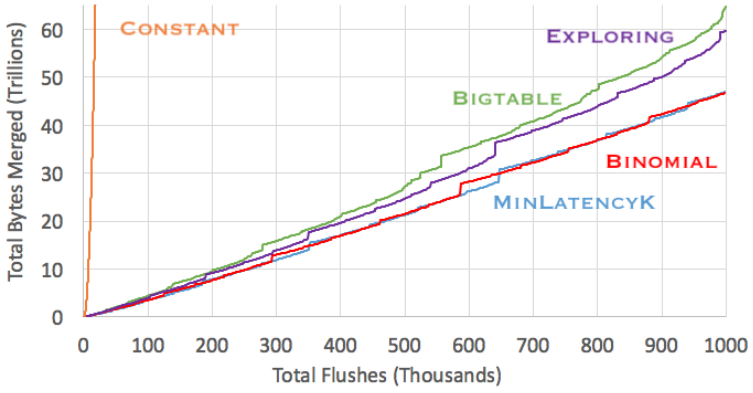


Figure 6.13: Simulated Merge Cost, $k = 10$

6.7 Concluding remarks

The Binomial and MinLatencyK policies were both able to outperform existing heuristic policies from real systems using an established data benchmark (YCSB). This highlights the potential advantages of using models rather than heuristics to create merge policies for LSM-based storage. However, this chapter focused on a simple workload model, and only scratched the surface for the design of merge policies based on specific workloads. Recall that the theoretical models made assumptions on uniform insertion, read cost, shrinkage, and (in the case of the binomial policy) the ratio of reads to merges, which may or may not be true for some workloads. A more advanced analysis of read vs write costs would also be helpful when considering read-heavy workloads (this is also where the difference between the two proposed policies can be made more clear).

In the future, more dynamic policies should be studied that take into account the possible shrinkage in the size of a compacted component in the presence of significant amount of antimatter (upserts or deletions), as well as alternate workloads.

Chapter 7

Conclusions

This thesis has described a new paradigm for big data, namely: *Big Active Data* which merges Big Data Management with active capabilities. It also described the implementation of a BAD system prototype using a modern Big Data Platform (AsterixDB), and showed how it can outperform passive Big Data by orders of magnitude in many practical scenarios. BAD can consider data in context and enrich results in ways unavailable in other active platforms, in addition to allowing for retrospective Big Data analytics by basing its subscription paradigm on queries over stored data rather than transient data streams. In addition, this thesis explored several research directions for improving the performance and scalability of a BAD platform.

From this point, BAD can be improved in a myriad of ways. For example, this thesis only scratched the surface of the research for Broker to User communication and scalability. Other future research directions include multi-channel optimization by sharing more work between channels, as well as moving from the repetitive channel model to the

continuous channel model discussed in Chapter 3.

My hope is that this thesis can provide the groundwork for a wide range of future BAD research and systems.

Bibliography

[1] http://www.couchbase.com/.

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

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

[4] Apache Hadoop. `http://hadoop.apache.org`.

[5] Apache HBASE. `https://hbase.apache.org`.

[6] Apache HBase Website. `http://hbase.apache.org/`.

[7] Apache Spark. `http://spark.apache.org`.

[8] Google Bigtable. `https://cloud.google.com/bigtable/`.

[9] Mongodb website. `http://www.mongodb.org/`.

[10] ONE Simulator. `https://akeranen.github.io/the-one/`.

[11] Pig Website. `http://hadoop.apache.org/pig`.

[12] U.S. geological survey – Shakecast, 2014. `http://earthquake.usgs.gov`.

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

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

[15] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous view maintenance for VLSD databases. In *ACM SIGMOD*, 2009.

[16] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[17] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, et al. AsterixDB: A scalable, open source bdms. *Proceedings of the VLDB Endowment*, 7(14):1905–1916, 2014.

[18] S. Alsubaiee et al. Storage management in AsterixDB. *Proc. VLDB Endowment*, 2014.

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

[20] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS*, 2012.

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

[22] B. Bamba, L. Liu, S. Y. Philip, G. Zhang, and M. Doo. Scalable processing of spatial alarms. In *High Performance Computing (HiPC)*. 2008.

[23] D. Borkar, R. Mayuram, G. Sangudi, and M. Carey. Have your data and query it too: From key-value caching to big data management. In *Proceedings of the 2016 International Conference on Management of Data*, pages 239–251. ACM, 2016.

[24] V. Borkar, Y. Bu, E. P. Carman, Jr., N. Onose, T. Westmann, P. Pirzadeh, M. J. Carey, and V. J. Tsotras. Algebricks: A data model-agnostic compiler backend for big data languages. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 422–433, New York, NY, USA, 2015. ACM.

[25] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.

[26] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

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

[28] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

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

[30] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, et al. Benchmarking streaming computation engines: Storm, Flink and Spark Streaming. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1789–1792. IEEE, 2016.

[31] R. Chirkova, J. Yang, et al. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[32] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, New York, NY, USA, 2010. ACM.

[33] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[34] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. *Proceedings of the VLDB Endowment*, 1(1):1189–1204, 2008.

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

[36] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04*, 2004.

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

[38] A. Dent. *Getting Started with LevelDB*. Packt Publishing Ltd, 2013.

[39] M. S. Desta, E. Hyytiä, A. Keränen, T. Kärkkäinen, and J. Ott. Evaluating (geo) content sharing with the one simulator. In *Proceedings of the 11th ACM international symposium on Mobility management and wireless access*, pages 37–40. ACM, 2013.

[40] N. Dindar, B. Güç, P. Lau, A. Özal, M. Soner, and N. Tatbul. DejaVu: Declarative pattern matching over live and archived streams of events. In *ACM SIGMOD*, 2009.

[41] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: A distributed, searchable key-value store. *ACM SIGCOMM*, 42(4), 2012.

[42] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2), 2003.

[43] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system S declarative stream processing engine. In *ACM SIGMOD*, 2008.

[44] L. George. *HBase: the definitive guide*. O'Reilly Media, 2011.

[45] L. Golab, T. Johnson, and V. Shkapenyuk. Scheduling updates in a real-time stream warehouse. In *IEEE ICDE*, 2009.

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

[47] R. Grover and M. J. Carey. Data ingestion in AsterixDB. In *EDBT*, pages 605–616, 2015.

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

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

[50] H. Jafarpour, B. Hore, S. Mehrotra, and N. Venkatasubramanian. Subscription subsumption evaluation for content-based publish/subscribe systems. In *Middleware*, 2008.

[51] C. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. *The VLDB JournalThe International Journal on Very Large Data Bases*, 16(4):417–437, 2007.

[52] Y. Jin and R. Strom. Relational subscription middleware for internet-scale publish-subscribe. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, pages 1–8. ACM, 2003.

[53] D. Judd. Scale out with HyperTable. *Linux magazine, August 7th*, 2008.

[54] J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Hubbell, P. Michaleas, J. Mullen, A. Prout, A. Reuther, A. Rosa, and C. Yee. Achieving 100,000,000 database inserts per second using Accumulo and D4m. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept. 2014.

[55] A. Keränen, J. Ott, and T. Kärkkäinen. The ONE Simulator for DTN Protocol Evaluation. In *SIMUTools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, New York, NY, USA, 2009. ICST.

[56] A. Khetrapal and V. Ganesh. HBase and Hypertable for large scale distributed storage systems. *Dept. of Computer Science, Purdue University*, pages 22–28, 2006.

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

[58] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[59] K. Lee, L. Liu, B. Palanisamy, and E. Yigitoglu. Road network-aware spatial alarms. *IEEE Transactions on Mobile Computing*, 15(1):188–201, 2016.

[60] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[61] G. Malewicz et al. Pregel: A system for large-scale graph processing. In *Proc. ACM SIGMOD Conf.* ACM, 2010.

[62] A. Markowetz, Y. Yang, and D. Papadias. Keyword search on relational data streams. In *ACM SIGMOD*, 2007.

[63] C. Mathieu, C. Staelin, N. E. Young, and A. Yousefi. Bigtable Merge Compaction. *ArXiv e-prints*, July 2014.

[64] T. Milo, T. Zur, and E. Verbin. Boosting Topic-Based Publish-Subscribe Systems with Dynamic Clustering. In *ACM SIGMOD*, 2007.

[65] M. M. Moro, P. Bakalov, and V. J. Tsotras. Early profile pruning on XML-aware publish/subscribe systems. In *VLDB*, 2007.

[66] M. Nikolic, M. ElSeidy, and C. Koch. Linview: incremental view maintenance for complex analytical queries. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 253–264. ACM, 2014.

[67] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[68] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. Lpez, G. Gibson, A. Fuchs, and B. Rinaldi. YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 9. ACM, 2011.

[69] M. A. Qader and V. Hristidis. DualDB: An efficient LSM-based publish/subscribe storage system. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2017.

[70] D. Quass and J. Widom. On-line warehouse view maintenance. In *ACM SIGMOD Record*, 1997.

[71] S. Saigaonkar, M. Rao, and S. Mantha. Publish subscribe system based on ontology and XML filtering. In *Computer Research and Development (ICCRD), 2011 3rd International Conference on*, volume 1, pages 154–158. IEEE, 2011.

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

[73] G. S. Thakur et al. Planetsense: A real-time streaming and spatio-temporal analytics platform for gathering geo-spatial intelligence from open source data. *arXiv preprint arXiv:1507.05245*, 2015.

[74] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2), 2009.

[75] M. Y. S. Uddin and N. Venkatasubramanian. Edge caching for enriched notifications delivery in big active data. *Submitted for publication.*

[76] X. Wang, W. Zhang, Y. Zhang, X. Lin, and Z. Huang. Top-k spatial-keyword publish/subscribe over sliding window. *The VLDB Journal*, 26(3):301–326, 2017.

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

[78] D. Yan, Y. Bu, Y. Tian, A. Deshpande, and J. Cheng. Big graph analytics systems. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2241–2243. ACM, 2016.

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

[80] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache Spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[81] Y. Zhao, K. Kim, and N. Venkatasubramanian. Dynatops: A dynamic topic-based publish/subscribe architecture. DEBS '13, 2013.