UNIVERSITY OF CALIFORNIA,
IRVINE

Activating Big Data at Scale

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Xikui Wang

Dissertation Committee:
Professor Michael J. Carey, Chair
Professor Vassilis J. Tsotras
Professor Chen Li

2020

Portion of Chapter 2 © 2019 VLDB Endowment DOI 10.14778/3342263.3342628
All other materials © 2020 Xikui Wang

DEDICATION

To world peace.

TABLE OF CONTENTS

Page

LIST OF FIGURES **vi**

ACKNOWLEDGMENTS **x**

VITA **xi**

ABSTRACT OF THE DISSERTATION **xiii**

1 Introduction **1**

2 Enriching Big Data Actively at Scale **5**

2.1 Overview 5
2.2 Preliminaries 8
2.2.1 Apache AsterixDB 8
2.2.2 Hyracks 9
2.2.3 Data Ingestion 9
2.3 Big Data Enrichment 11
2.3.1 Motivation 11
2.3.2 UDFs for Data Enrichment 12
2.3.3 Utilizing Existing Knowledge 13
2.4 Data Enrichment for Analysis 15
2.4.1 Option 1 - Enrich during Querying 16
2.4.2 Option 2 - Enrich during Data Ingestion 16
2.4.3 More Complex Enrichment 18
2.5 Framework Building Blocks 24
2.5.1 Predeployed Jobs 24
2.5.2 Layered Ingestion Pipeline 26
2.5.3 Partition Holders 27
2.6 The New Ingestion Framework 28
2.6.1 Ingestion Life Cycle 28
2.6.2 New Ingestion Architecture 29
2.7 Experiments 31
2.7.1 Basic Data Ingestion 32
2.7.2 Data Enrichment with UDFs 34
2.7.3 Data Enrichment with Updates 37

2.7.4 Scale-out Experiments . . . 38
2.8 Related Work . . . 43
2.9 Summary . . . 46

3 Subscribing to Big Data Continuously at Scale 48
3.1 Overview . . . 48
3.2 Related Work . . . 52
3.3 Big Active Data . . . 53
3.3.1 A BAD World . . . 54
3.3.2 The BAD Building Blocks . . . 55
3.4 Repetitive BAD: BAD-RQ . . . 57
3.4.1 A BAD Repetitive Use Case . . . 57
3.4.2 Data Channel Evaluation . . . 63
3.5 Continuous BAD: BAD-CQ . . . 67
3.5.1 Approximately Continuous Queries . . . 68
3.5.2 BAD-CQ . . . 72
3.5.3 BAD-CQ Syntax and Optimization . . . 77
3.5.4 BAD-CQ Semantics . . . 81
3.6 GOOD: A Not BAD Approach . . . 85
3.6.1 The GOOD Architecture . . . 85
3.6.2 A GOOD System . . . 88
3.6.3 GOOD vs. BAD . . . 91
3.7 Experimental Results . . . 93
3.7.1 Active Dataset Scale-out Performance . . . 93
3.7.2 Channel Performance . . . 95
3.7.3 Good vs. BAD Performance . . . 98
3.7.4 BAD Scalability . . . 101
3.8 Summary . . . 103

4 Sharing Big Data Declaratively at Scale 105
4.1 Overview . . . 105
4.2 A Three-island Example . . . 106
4.2.1 BAD Island 1: Department of Homeland Security . . . 106
4.2.2 BAD Island 2: Orange County Sheriff's Department . . . 107
4.2.3 BAD Island 3: University of California-Irvine . . . 108
4.3 Island Hopping: Connecting BAD Islands . . . 109
4.3.1 Option 1: A BAD Continent . . . 110
4.3.2 Option 2: BAD Ferries . . . 111
4.3.3 Option 3: BAD Bridges . . . 112
4.4 Building BAD Bridges . . . 114
4.4.1 BAD Brokers . . . 114
4.4.2 BAD Feeds . . . 115
4.5 A Prototype of BAD Islands . . . 116
4.5.1 BAD Trinity Use Case . . . 117
4.5.2 The Trip of A Threatening Tweet . . . 128

4.6 System Demonstration: A BAD Islands Tour . . . 133
4.6.1 DHS Dashboard . . . 133
4.6.2 OCSD Dashboard . . . 137
4.6.3 UCI Dashboard . . . 141
4.7 Summary . . . 144

5 Conclusion and Future Work 145
5.1 Conclusion . . . 145
5.2 Future Work . . . 147

Bibliography 149

Appendix A Enrichment Functions 158

LIST OF FIGURES

Page

2.1 DDL statements for storing tweets 8
2.2 Translating a user query to a Hyracks job 9
2.3 Insert data into a dataset 10
2.4 Create a socket feed 11
2.5 Java UDF 1 for tweet safety check 13
2.6 SQL++ UDF 1 for tweet safety check 13
2.7 Java UDF 2 for tweet safety check 14
2.8 SQL++ UDF 2 for tweet safety check 15
2.9 An analytical query using SQL++ UDF 2 16
2.10 Enrich and insert collected tweets 17
2.11 Enrich and insert ingested tweets 18
2.12 Attach a SQL++ UDF to a data feed 18
2.13 Enrich and insert a constant record 19
2.14 Enrich and insert records from a feed 20
2.15 Case 1: Small SensitiveWords dataset 21
2.16 Case 2: Big SensitiveWords dataset 22
2.17 Case 3: Enrich with an available index 23
2.18 Enrichment UDF with a nested subquery 23
2.19 Ingestion pipeline using insert jobs 25
2.20 Parameterized predeployed job 25
2.21 Decoupled ingestion framework 26
2.22 Partition holders 28
2.23 The new ingestion framework 30
2.24 10M tweets ingestion speed-up over 24 nodes 33
2.25 1M tweets Ingestion with UDFs (log scale) 36
2.26 Refresh periods under different batch sizes 36
2.27 Reference data update 38
2.28 Reference data scale-out 39
2.29 UDF complexity comparison 41
2.30 100K tweets ingestion speed-up for 24 vs. 6 Nodes with different batch sizes 41
2.31 100K tweets ingestion speed-ups 43

3.1 A sample analytical query on collected tweets 54
3.2 A BAD system for a BAD world 57

3.3 Datatype and dataset definition for Tweets 59
3.4 Datatype and dataset definition for officer location updates 59
3.5 A data feed for ingesting tweets 60
3.6 A data feed for ingesting location updates 60
3.7 An SQL++ query looking for the 10 most hateful cities in each month in a given time frame 60
3.8 A UDF based on an analytical query 61
3.9 An UDF for counting hateful tweets near certain in-field officer given his/her officer ID 62
3.10 Creating a data channel based on a UDF with a parameter 62
3.11 Registering a broker to BAD 62
3.12 Subscribing to a channel with parameters on a broker 62
3.13 Data type definitions for brokers and subscriptions (internal to BAD) 64
3.14 A data sample for evaluating a data channel 64
3.15 An illustrative query for computing a channel 66
3.16 A query plan for channel evaluation 66
3.17 A UDF for adding ingestion time 69
3.18 A repetitive data channel looking for new nearby hateful tweets 70
3.19 Missing data due to scheduling delays 71
3.20 Missing tuple due to early timestamping 71
3.21 Storage format of an active record 73
3.22 Access active datasets with filters 74
3.23 Datatype and dataset definition for officer location updates 75
3.24 An illustration of active timestamp management 76
3.25 A continuous channel for new nearby hateful tweets 79
3.26 Query plan for new nearby hateful tweet channel 79
3.27 A continuous channel for unseen nearby hateful tweets 80
3.28 Expanding a continuous channel query with active functions 80
3.29 Query plan for unseen nearby hateful tweet channel 81
3.30 Officer u10 subscribing to CQNewNearbyHatefulTweets(u10) and officer u20 subscribing to CQNewNearbyHatefulTweets(u20) 82
3.31 Officer u10 subscribing to UnseenNearbyHatefulTweets(u10) and officer u20 subscribing to UnseenNearbyHatefulTweets(u20) 83
3.32 A continuous channel for new nearby hateful tweets 84
3.33 Officer u10 subscribing to NewNearbyHatefulTweetsForActiveOfficers(u10) and officer u20 subscribing to NewNearbyHatefulTweetsForActiveOfficers(u20) 85
3.34 GOOD Architecture 86
3.35 A concrete GOOD system 88
3.36 A GOOD example of sending hateful tweets to officers 92
3.37 Ingestion performance on active datasets 94
3.38 Query performance on active datasets 94
3.39 Datatype and dataset definition for Schools 96
3.40 A continuous channel for new local hateful tweets 96
3.41 A continuous channel for new local hateful tweets with schools 97

3.42 Maximum number of supportable subscribers under different incoming data rates . . . 97
3.43 Performance comparison of BAD-CQ and the GOOD system . . . 99
3.44 Cost of "NewLocalHatefulTweets" with 150,000 subscribers and 80 tweets/second on both the GOOD and BAD system . . . 101
3.45 Speed-up BAD-CQ with fixed incoming tweet rate and number of subscribers 102
3.46 Scale-out BAD-CQ with fixed incoming tweet rate . . . 102
3.47 Scale-out BAD-CQ with increasing incoming tweet rate . . . 103

4.1 An overview of the DHS Island . . . 107
4.2 An overview of the OCSD island . . . 108
4.3 An overview of the UCI island . . . 109
4.4 An illustration of a BAD continent . . . 110
4.5 An illustration of BAD ferries . . . 112
4.6 An illustration of BAD bridges . . . 113
4.7 Creating a BAD broker . . . 115
4.8 Creating a BAD feed . . . 116
4.9 Data type and dataset definition for tweets . . . 117
4.10 Definition for TweetFeed . . . 118
4.11 Data type and dataset definition for weapon registration information . . . 118
4.12 A Java UDF for detecting threatening rating . . . 119
4.13 Data type and dataset definition for weapon registration information . . . 120
4.14 Applying the enrichment UDF to the feed and starting the ingestion . . . 120
4.15 Definition of the ThreateningTweetsAt channel . . . 120
4.16 An overview of the DHS BAD system . . . 121
4.17 Data type and dataset definition for events . . . 121
4.18 Data type and dataset definition for local threatening tweets at Orange County 122
4.19 Definition, connect and start feed statements for LocalThreateningTweetFeed 122
4.20 Data type and dataset definition for officer location . . . 123
4.21 Definition, connect and start feed statements for OfficerLocationFeed . . . 123
4.22 Definition of the ThreateningEventsNear channel . . . 124
4.23 An overview of the OCSD BAD system . . . 124
4.24 Connecting UCI BAD to DHS BAD . . . 125
4.25 Data type and dataset definition of buildings . . . 126
4.26 Data type and dataset definition of security guard stations . . . 126
4.27 Definition of the AlertsOnCampus channel . . . 127
4.28 An overview of the UCI BAD system . . . 127
4.29 An overview of BAD island . . . 128
4.30 A sample raw threatening tweet . . . 129
4.31 The enriched threatening tweet . . . 129
4.32 The generated threatening tweet notification from DHS . . . 130
4.33 The generated threatening event notification from OCSD . . . 131
4.34 The generated on-campus alert from UCI . . . 132
4.35 An overview of the DHS dashboard . . . 133
4.36 DHS dashboard functions . . . 134

4.37 Exemplary data analytics 134
4.38 Visualization Panel of DHS dashboard 135
4.39 Tweet Panel for demonstrating raw and enriched tweets' content 136
4.40 An overview of the OCSD dashboard 137
4.41 OCSD Function Panel 138
4.42 Incoming threatening tweets from DHS BAD 139
4.43 Visualization Panel of OCSD dashboard 140
4.44 Notification Panel for displaying threatening event notifications to officers . . 140
4.45 An overview of the UCI dashboard 141
4.46 Demo Functions for UCI BAD 141
4.47 On-campus alerts 142
4.48 Security information of an alert 142
4.49 Visualization Panel of UCI BAD 143

ACKNOWLEDGMENTS

I would like to express my sincerest gratitude to my advisor, Professor Michael J. Carey, for his guidance and support during my Ph.D. study. He is a serious system researcher who drives me to investigate different systems' behaviors and performances in every detail. He is also a patient advisor who allows me to learn from exploration and helps me become a better researcher in all possible ways. He has made me realize the most important thing is to always make something good that you can be proud of. I'm fortunate to have the opportunity to work with Professor Carey. Everything that I've learned from him will benefit me forever.

I would like to thank Professor Vassilis J. Tsotras. We have been collaborating on the BAD project since 2018. Professor Tsotras is a very kind mentor. He has given me tremendous help both in my research work and personal life. Professor Tsotras is often the mediator in our discussions, and he is always willing to spend time scrutinizing my writings to make sure readers can easily follow. The BAD project wouldn't have been possible without his help.

I would like to thank Professor Chen Li for joining my dissertation committee and his help that allowed me to join the AsterixDB's group. I collaborated with Professor Li on my first database work and served as his teaching assistant for CS122A. He is a cautious thinker and always focuses on details from every aspect. I have benefited a lot from working with him.

During my time at UCI, I have had the fortune to work with many excellent people, including Professor Sharad Mehrotra, Professor Nalini Venkatasubramanian, Professor Md Yusuf Sarwar Uddin, Professor Heri Ramampiaro, Professor Liyan Zhang, Steven Jacobs, Phanwadee Sinthong, and Wail Alkowaileet. Many of them have offered me a lot of significant opportunities and lessons, which I will cherish forever.

My life at UCI wouldn't have been so rewarding and fun without my colleagues' help and my friends' company. I would like to thank Yingyi Bu, Jianfeng Jia, Taewoo Kim, Chen Luo, Shiva Jahangiri, Abdullah Alamoudi, Dmitry Lychagin, Till Westmann, Ian Maxon, Primal Pappachan, Dhrubajyoti Ghosh, Eunjeong Shin, Roberto Yus, Guoxi Wang, Xianwen Wang, Qin Han, Qiuxi Zhu, Xinwen Zhang, Fangqi Liu, Hang Nguyen, Xuyang Cheng, Zong Miao, Shan Jiang, Jiajun Xia. Let me know if you read this so that I can buy you a drink.

Most importantly, I would like to thank my wife, Jia, for lighting up my life. It takes so much more than just courage to switch from a stable career to joining me on a journey full of uncertainty. Also, I would like to thank my parents, Songyao and Xinyu, for supporting me in pursuing my dream. I am forever indebted to their love and support.

The work reported in this paper was supported by the Donald Bren Foundation (via a Bren Chair), NSF CNS award 1305430, NSF grants IIS-1447720, IIS-1838248, and CNS-1925610.

VITA

Xikui Wang

EDUCATION

Doctor of Philosophy in Computer Science **2020**
University of California Irvine *Irvine, California*

Master of Science in Computational Science **2014**
Zhejiang University *Hangzhou, Zhejiang*

Bachelor of Science in Software Engineering **2011**
Harbin Engineering University *Harbin, Heilongjiang*

RESEARCH EXPERIENCE

Graduate Student Researcher **2014–2020**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2015–2019**
University of California, Irvine *Irvine, California*

REFEREED JOURNAL PUBLICATIONS

BAD to the Bone - Big Active Data at its Core **2020**
The VLDB Journal (VLDBJ)

An IDEA: An Ingestion Framework for Data Enrichment in AsterixDB **2019**
Proceedings of the Very Large Database Endowment (PVLDB)

A BAD Demonstration: Towards Big Active Data **2017**
Proceedings of the Very Large Database Endowment (PVLDB)

REFEREED CONFERENCE PUBLICATIONS

End-to-End Machine Learning with Apache AsterixDB **Jun 2018**
DEEM: Workshop on Data Management for End-to-End Machine Learning @ SIGMOD'18

Enhancing Big Data with Semantics: The AsterixDB Approach (Poster) **Jan 2018**
The 12th IEEE International Conference on Semantic Computing (ICSC)

SOFTWARE

AsterixDB `http://asterixdb.ics.uci.edu`
An open source Big Data Management System that provides distributed data management for large-scale semi-structured data.

Big Active Data `http://asterix.ics.uci.edu/bigactivedata`
A platform that combines ideas and capabilities from both Big Data and Active Data and supports scalable data analytics and complex subscriptions regarding incoming and/or stored data.

ABSTRACT OF THE DISSERTATION

Activating Big Data at Scale

By

Xikui Wang

Doctor of Philosophy in Computer Science

University of California, Irvine, 2020

Professor Michael J. Carey, Chair

With our world being more digitized than ever, handling Big Data has become a fundamental challenge in building modern applications and services. Although both academia and industry have developed a plethora of systems in recent years to help developers working with Big Data, many of them still follow the pattern of *passively* responding to users' queries, rather than processing and delivering data to interested users *actively*. We need systems for *activating Big Data at scale* and reducing the users' effort in working with *Big Active Data*.

In this dissertation, we explore three problems related to activating Big Data at scale. We first investigate the problem of enabling data enrichment during data ingestion. We discuss the needs and challenges in enriching data during data ingestion, and we introduce a new ingestion framework into AsterixDB - *dynamic data feeds* - that supports complex data enrichment functions and captures relevant data changes in the system during ingestion. We show the design and implementation of the new ingestion framework and evaluate its performance using different enrichment use cases.

Then, we look at the Big Active Data (BAD) challenge. We describe a BAD world that consists of different types of users and requests, and we propose a BAD system for providing BAD services for BAD users. We first review the initial prototype of the BAD system - BAD-RQ - and we discuss its limitations in BAD continuous use cases. We introduce a new

BAD service - *BAD-CQ* - for providing continuous query semantics in the BAD system. Further, we use an alternative system constructed by gluing together multiple existing Big Data systems to show the challenges in providing BAD services without the BAD system. We measure the performance of BAD-CQ with various workloads and compare that with the alternative system's performance.

Last but not least, we study how to allow users to declaratively create scalable data sharing services between multiple BAD system instances without having to create and manage dedicated programs/services. We describe the notion of *BAD islands* that consist of multiple BAD instances and introduce new features to the BAD system for "bridging" multiple BAD instances together. We use a sample use case to illustrate how to create bridges between different BAD systems. To this end, we present a demonstration system that also involves the use of dynamic data feeds and BAD-CQ to show how BAD islands work.

Chapter 1

Introduction

Starting from the time when an ancient man carved the first notch on a bone stick to mark his day of work to now, when everyone can post a video on the Internet to share his/her morning coffee with the world, the ways of humans producing, recording, and sharing data have been evolving all the time [31]. The Digital Revolution converted data from analog to a digital format; this has vastly improved the efficiency of generating, replicating, and exchanging information [77, 97]. Advances in sensors, communications, and computation have created large collections of data [18]. The telescope of the Sloan Digital Sky Survey collected more data in its first few weeks than the entire history of astronomy [35]. The Digital Universe is expected to grow 40% annually into the next decade, and the volume of the world's digital data is estimated to reach 175 zettabytes (175 trillion gigabytes) in 2025 [30, 75]. Producing *Big Data* has become one of the daily routines of humankind.

Big Data comes with big challenges in storing, managing, and processing data. Developments in storage technologies, ranging from magnetic tapes to non-volatile random access memory (NVRAM), enable us to read and write huge data collections with trade-offs in capacity and speed [65]. In order to manage and make use of data, a series of Database management

systems (DBMS) have been developed to allow users to interact with data using declarative languages [11, 84, 85, 86]. The data models adopted by the DBMSs have been involving over time as well [12, 27, 59, 86]. Along the way, many special systems have been designed and developed for specific use cases, such as temporal databases [81], graph databases [95], spatial databases [47], and so on. More recently, to deal with the exponential growth of data volume, shared-nothing architectures and parallelism have become the best practice in modern Big Data Management Systems (BDMS) [4, 29, 60, 89].

While researchers and experts from academia and industry are continuously designing and implementing systems to support applications in the Big Data era, most of them are still passive in nature. They *passively* return data by replying to user queries instead of *actively* processing data and delivering it to interested users. Active databases provide mechanisms for reacting to events, but database systems provide active database features mostly in the form of triggers, which have limited functionality and scalability in Big Data use cases [70, 96]. Recent streaming engines support real-time data processing and analytics. Still, due to their streaming nature, streaming engines usually lack the ability to analyze historical data and do not provide a number of useful features that traditional database systems offer (such as indexes, declarative query languages, etc.) [19, 51, 100]. As more data is being generated by various activities, devices, and services everyday, and more users are interested in different aspects of data, it is critical to find ways to ingest, process, and serve information extracted from this massively growing sea of data to millions of users actively. We refer to this as *Activating Big Data at Scale*.

Activating Big Data creates a series of challenges related to ingesting, processing, and serving data. We need different technologies to support new active data services and to allow developers to easily create and manage these services without having to glue multiple systems together. This thesis focuses on the following three problems:

- **Enriching Big Data actively at Scale:** Data coming into a system often needs to be enriched with other information to support later analytical queries. For enrichments that will be frequently used, it can be advantageous to push their computation into the ingestion pipeline so that the enrichments can be stored with the data to avoid recomputation and can be used by analytical queries and other services immediately. As the enrichment computation may be complex and referenced information may change over time, we need an ingestion pipeline that can support various processing operations and adapt to such changes while guaranteeing the currency and/or correctness of the enriched data.

- **Subscribing to Big Data continuously at Scale:** Data ingested is important not only for the information that it contains but also for its relationships to other data and to interested users. In order to allow users to subscribe to Big Data continuously at scale, one would have to either heavily customize an existing passive system or to glue multiple systems together. Either choice would require significant effort from users and incur additional overhead. We need a system that allow users to easily create Big Data subscription services at scale, without having to glue together and manage multiple systems.

- **Sharing Big Data declaratively at Scale:** Data delivered to interested users sometimes may not come from a single system but may instead require combining information from multiple systems. Offering such data services requires sharing data between different systems, but this is not a trivial task due to management complexity and scalability requirements. Developers today often need to create dedicated programs for enabling data sharing between different systems. This introduces additional work for developing and managing these programs. We need a declarative way of enabling data sharing between systems.

This thesis is organized as follows. We first introduce a new data ingestion framework for

data enrichment in Chapter 2 that supports data ingestion at scale, enrichments requiring complex operations, and adaptiveness to referenced data updates. In Chapter 3, we introduce the Big Active Data (BAD) system for enabling Big Data subscriptions and introduce BAD-CQ, a new service for enabling continuous query semantics in BAD. Chapter 4 describes BAD islands, an architecture that allows users to declaratively share data among BAD systems without having to create and manage dedicated services. We conclude this thesis and discuss possible future directions in activating Big Data in Chapter 5.

Chapter 2

Enriching Big Data Actively at Scale

2.1 Overview

Traditionally, data to be analyzed has been obtained from one or more operational systems, fed through an Extract, Transform, and Load (ETL) process, and stored in a data warehouse [24]. In today's Big Data era, the data that people work with is no longer limited to the operational data from a company but also includes social network messages, sensor readings, user click-streams, etc. Data from these sources is generated rapidly and continuously. It becomes increasingly undesirable to stage the data in large batches, process it overnight, and then load it into a data warehouse due to the volume of the incoming stream and the need to analyze current data when making important decisions.

To support the ingestion of continuously generated data and provide near real-time data analysis, streaming engines have been introduced into the Big Data analysis architecture [16, 34, 94]. The incoming data is collected by a streaming engine and then pushed to (or periodically pulled by) a warehouse for later complex data analysis. Adding a streaming engine simplifies the ingestion process for the data warehouse, but it also introduces data routing

overhead between different systems. To minimize this overhead and simplify the architecture, some systems such as Apache AsterixDB [41] have chosen to provide an integrated ingestion facility, data feeds, to enable users to ingest data directly into the system.

The ingested data, such as sensor readings, is often not useful alone in high-level data analysis. To reveal more valuable insights, it needs to be enriched, e.g, by relating it to reference information and/or applying machine learning models. When the enriched data needs to be queried frequently, its computation is often pushed into the ingestion pipeline and the enriched data is then persisted [57]. This requires the ingestion framework to have the ability to process incoming data efficiently and to access reference data/machine learning models when needed.

Most streaming engines support incoming data processing, but some only support a limited query syntax [19, 51]. If a processing task requires existing data from a warehouse, most streaming engines would need to query the warehouse repeatedly, as otherwise they would have to keep a copy of that data locally. Frequent queries to the warehouse would increase the load on the system and incur latency, and maintaining multiple copies of the data would require data migration to keep the data consistent [57]. Both choices would slow down the enrichment/ingestion pipeline and increase the complexity of building the overall analysis platform.

The data feeds in AsterixDB support data enrichment during ingestion by allowing users to attach user-defined functions (UDFs) to the ingestion pipeline. UDFs are a longstanding and commonly available feature in databases. Data enrichment operations can be encapsulated in UDFs and be reused for different use cases. A Java UDF in AsterixDB can be used on a data feed to enrich the incoming data using existing information from resource files. A SQL++ UDF on a data feed can manipulate the incoming data declaratively using a SQL++ query. Currently, however, such a SQL++ function in AsterixDB must be limited to only accessing the content of a given input record, as the execution plan for a SQL++ UDF in

general can be stateful. If other data were accessed by a SQL++ function attached to an AsterixDB data feed, its query plan could fail to be evaluated or generate intermediate states that neglect changes in referenced data. This limits the expressiveness (usefulness) of data enrichment using SQL++ UDFs in AsterixDB today.

Considering the potential benefits of data enrichment during ingestion, we believe that a data ingestion facility should provide high-performance ingestion for incoming data, a full query syntax support to data enrichment, efficient access to existing data, and adaptiveness to changes in referenced data. With these requirements in mind, we have built a new ingestion framework for Apache AsterixDB. We have improved the scalability and stability of its data feeds and enabled users to attach declarative UDFs on data feeds with support for a full query capability. We have decoupled the ingestion pipeline into layers based on their functionality and life-cycle to improve ingestion efficiency and to allow ingestion pipelines to adapt to data changes dynamically.

The rest of this chapter is organized as follows. We introduce the background information about Apache AsterixDB, its Hyracks runtime engine, and rapid data ingestion in Section 2.2, and then we discuss how to enrich ingested data at scale in Section 2.3. In Section 2.4, we investigate different strategies for utilizing data enrichment for data analysis and the current limitations of each for providing current and correct data enrichment in time. We explain how we built the new ingestion framework and the techniques used in Section 2.5, and we elaborate on the details of the new ingestion framework in Section 2.6. We investigate its performance in Section 3.7, review related work in Section 2.8, and conclude our work in Section 2.9.

2.2 Preliminaries

We use Apache AsterixDB to highlight and address the challenges of enriching incoming data during data ingestion. Here we provide a brief introduction to Apache AsterixDB, its runtime engine Hyracks, and data ingestion.

2.2.1 Apache AsterixDB

Apache AsterixDB [4] is an open source Big Data Management System (BDMS). It provides distributed data management for large-scale, semi-structured data. It aims to reduce the need for gluing together multiple systems for Big Data analysis. AsterixDB uses SQL++ [23, 68] (a SQL-inspired query language for semi-structured data) for user queries and *the AsterixDB Data Model* (ADM) to manage the stored data. ADM is a superset of JSON and supports complex objects with nesting and collections. Before storing data in AsterixDB, a user can create a *Datatype*, which describes known aspects of the data being stored, and a *Dataset*, which is a collection of records of a datatype. AsterixDB allows a user to specify a datatype as "open" which makes it a minimal, extensible description of the stored data. As shown in the example in Figure 2.1, we can create an open datatype named "TweetType" with only two required attributes: "id" and "text". Tweets containing a variety of additional attributes can be stored and queried in this dataset as well.

```
CREATE TYPE TweetType AS OPEN {
    id : int64,
    text: string
};
CREATE DATASET Tweets(TweetType)
PRIMARY KEY id;
```

Figure 2.1: DDL statements for storing tweets

2.2.2 Hyracks

Hyracks [15] is a partitioned parallel computation platform that provides runtime execution support for AsterixDB. Queries from users are compiled into Hyracks jobs. A "job" is a unit of work that can be executed on Hyracks. A "job specification" describes how data flows and is processed in a job. It contains a DAG of operators, which describe computational operations, and connectors, which express data routing strategies. Data in a runtime Hyracks job flows in frames containing multiple objects. An operator reads an incoming data frame, processes the objects in it, and pushes the processed data frame to another connected operator through a connector. AsterixDB uses jobs to evaluate user queries. A query submitted to AsterixDB is parsed and optimized into a query plan and then compiled as a job specification to run on the Hyracks platform. Figure 2.2 shows an example of how a user query can be represented as a Hyracks job.

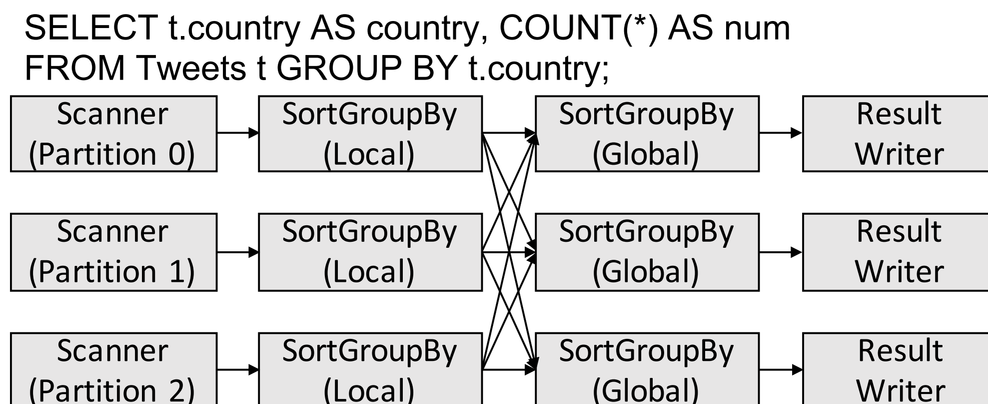


Figure 2.2: Translating a user query to a Hyracks job

2.2.3 Data Ingestion

In many contemporary data analysis use cases, data no longer stays on storage devices to be batched into a database system later. Instead, it enters the system rapidly and continuously. The traditional bulk loading technique cannot be applied due to the active nature of the

incoming data. Repeatedly issuing insert statements would be impractical because of its low efficiency.

To untangle database systems and users from the handling of rapidly incoming data, one popular solution is to couple a streaming engine with a database system and use the streaming engine to handle the data [57]. For example, one can set up a Kafka instance to ingest data from external sources, then create a program [64] or use the *kafka-mongodb-connector* [40] to transmit the ingested data to a MongoDB instance for later analysis. A naive solution to this problem could be to create an external program that obtains/receives these records from external sources and pushes them actively into a dataset using *INSERT* statements like the one in Figure 2.3.

```
INSERT INTO Tweets([
  {"id":0, "text": "Let there be light"}
]);
```

Figure 2.3: Insert data into a dataset

When data comes in at a rapid rate, it is impractical to issue repetitive insert statements for ingesting data into a dataset even if a user batches multiple records into a single insert statement. The processing cost of massive insert statements soon becomes a bottleneck in the system and cannot be scaled to handle faster incoming data. In addition, a robust and efficient facility for moving data from external sources into database systems is a common need for many users with similar data analysis use cases.

While streaming engines provide scalable and reliable data ingestion for database systems, they simultaneously introduce additional data routing costs. For example, the incoming data has to be persisted in Kafka first, then be pushed to the connected database system. Writing the same piece of data multiple times would cost more resources and also delay the demanding analytical queries awaiting the latest ingested data. In addition, configuring multiple systems and wiring them together could be challenging for data analysts, who

usually have little system management experience.

To simplify the process of ingesting data for end users and improve ingestion efficiency for analytical systems, some database systems provide integrated data ingestion facilities for handling rapid incoming data. Apache AsterixDB, for example, provides data feeds that allow users to assemble simple data ingestion pipelines with DDL statements [41]. A user can use the DDL statements in Figure 2.4 to create a data feed that receives incoming JSON formatted tweets using a socket server with a specified configuration. A data feed consists of two components: an adapter, which obtains/receives data from an external data source as raw bytes; and a parser, which translates the ingested bytes into ADM records. Compared with the "glue" solution using streaming engines, data feeds have no extra data routing overheads, and a user can easily assemble a basic data ingestion pipeline with declarative statements.

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

Figure 2.4: Create a socket feed

2.3 Big Data Enrichment

2.3.1 Motivation

Due to various restrictions, such as the limited bandwidth of infield sensors and predefined data formats from API providers, data coming from external sources may not contain all

the information needed for meaningful data analysis. In these scenarios, one can enrich the ingested data with existing knowledge (reference data) or machine learning models to reveal more useful information. For example, the IP address in a log record can be enriched by referencing IP reputation data to see whether there is known threat activity associated with that IP address [49]. Similarly, an incoming tweet can be enriched by referencing a sensitive key word list to see if its text is potentially threatening. It can also be processed by data mining techniques to extract sentiment and named entities for later analysis [33, 56]. Tweets can be enriched by utilizing linguistic processing, semantic analysis, and sentiment analysis techniques and can be used in societal event forecasting systems [33]. Raw data collected by sensor networks can be enriched to support higher level applications, such as improving training effects in sports [28] and building health-care services for medical institutions [71].

One way of expressing data enrichment requests is to use User-defined Functions (UDFs). This approach allows users to create functions with queries or programs and to reuse them. It enables users to modularize their data enrichment operations and to easily scale their computations on Big Data with the support of a BDMS, like AsterixDB. Since UDFs are available in most databases, the user model and system design around UDFs can be generalized to similar systems. Here we use the UDF framework in Apache AsterixDB for data enrichment.

2.3.2 UDFs for Data Enrichment

AsterixDB supports both Java and SQL++ in its UDF framework [3]. One can implement a Java UDF that utilizes the provided API to manipulate an input record, or create a SQL++ UDF to enrich input data using a declarative query. For example, we might want to create a UDF that checks whether a given tweet from the U.S. contains the keyword "bomb". If so, the UDF should add a new field, "safety_check_flag", to the tweet and set it to "Red".

If not, the UDF should add the same new field and sets it to “Green”. Figure A.4 shows an example of the Java UDF implementation (Java UDF 1) of such a UDF.

```
...
public void evaluate(IFunctionHelper functionHelper) throws Exception {
  JRecord inputRecord = (JRecord) functionHelper.getArgument(0);
  JString countryCode = (JString) inputRecord.getValueByName("country");
  JString text = (JString) inputRecord.getValueByName("text");

  safetyCheckFlag.setValue(countryCode.getValue().equals("US") &&
    text.getValue().contains("bomb") ? "Red":"Green");
  inputRecord.addField("safety_check_flag",safetyCheckFlag);
  functionHelper.setResult(inputRecord);
}
...
```

Figure 2.5: Java UDF 1 for tweet safety check

Although Java UDFs are powerful tools for enriching incoming data, especially when combined with machine learning models, constructing Java UDFs can be more complicated than writing SQL++ queries when expressing the same data enrichment requirements. In addition, a SQL++ UDF can be updated using an *UPSERT* statement instantly while updating a Java UDF requires a recompilation and redeployment process. Figure 2.6 shows an equivalent SQL++ UDF that performs the safety check for a given tweet (SQL++ UDF 1).

```
CREATE FUNCTION USTweetSafetyCheck(tweet) {
  LET safety_check_flag =
    CASE tweet.country = "US" AND contains(tweet.text, "bomb")
      WHEN true THEN "Red" ELSE "Green"
    END
  SELECT tweet.*, safety_check_flag
};
```

Figure 2.6: SQL++ UDF 1 for tweet safety check

2.3.3 Utilizing Existing Knowledge

In some use cases, a UDF needs to access existing knowledge, such as machine learning models or relevant stored information, for data enrichment. Both Java and SQL++ UDFs

can support utilizing existing knowledge. A Java UDF in AsterixDB can load external files during its initialization. A SQL++ UDF can access reference data stored in datasets. To expand on our tweet safety check example in Section 2.3.2, given a list of countries and their sensitive keywords, suppose we want to flag a tweet from a country if it contains one of the keywords associated with that country. For a Java UDF, we can put the country-to-keywords mappings into a local resource file and load it during the UDF initialization. Figure 2.7 shows a snippet of the implementation of this Java UDF (Java UDF 2). For a SQL++ UDF, we can store the mappings in a "SensitiveWords" dataset and use a SQL++ query to enrich the input data. Figure 2.8 shows its SQL++ implementation (SQL++ UDF 2).

```
...
@Override
public void initialize(IFunctionHelper functionHelper, String nodeInfo)
    throws IOException {
    ...
    BufferedReader fr = Files.newBufferedReader(
        Paths.get(keywordListPath));
    fr.lines().forEach(line -> {
        String[] items = line.split("\\|");
        keywordList.putIfAbsent(items[1], new LinkedList<>());
        keywordList.get(items[1]).add(items[2]);
    });
}
...
public void evaluate(IFunctionHelper functionHelper) throws Exception {
    JRecord inputRecord = (JRecord) functionHelper.getArgument(0);
    JString countryCode = (JString) inputRecord.getValueByName("country");
    JString text = (JString) inputRecord.getValueByName("text");
    List<String> keywords = keywordList.getOrDefault(
        countryCode.getValue(),Collections.emptyList());

    for (String keyword : keywords) {
        safetyCheckFlag.setValue(
            text.getValue().contains(keyword) ? "Red" : "Green");
    }
    inputRecord.addField("safety_check_flag", safetyCheckFlag);
    functionHelper.setResult(inputRecord);
}
...
```

Figure 2.7: Java UDF 2 for tweet safety check

```
CREATE FUNCTION tweetSafetyCheck(tweet) {
LET safety_check_flag = CASE
  EXISTS(SELECT s FROM SensitiveWords s
    WHERE tweet.country = s.country AND contains(tweet.text, s.word))
  WHEN true THEN "Red" ELSE "Green"
  END
SELECT tweet.*, safety_check_flag
};
```

Figure 2.8: SQL++ UDF 2 for tweet safety check

Loading external files in a Java UDF is commonly used for necessary configurations that are infrequently updated. However, when an update occurs, such as new keywords being added for a certain country, the resource files on every node will also need to be updated. A reference dataset used in a SQL++ UDF, in contrast, can easily be updated by *INSERT/UPSERT* [1] statements.

2.4 Data Enrichment for Analysis

The goal of data enrichment is to allow analysts to use the enriched information in analytical queries. One can enrich ingested data **lazily**, when constructing the analytical queries, or **eagerly** at ingestion time and then store the enriched results. Enriching data in analytical queries is good for one-time queries, while enriching and storing enriched results allows faster responses for future analytical queries. In this section, we show examples and discuss the implementation of both options.

[1] A *UPSERT* statement in SQL++ inserts an object if there is no another object with the specified key. If not, it replaces the previous object with the new one.

2.4.1 Option 1 - Enrich during Querying

For data enrichment used in one-time analytical queries, one can apply enrichment UDFs directly when querying the data. A UDF in an analytical query can be optimized together with the query to produce an optimized query plan. For example, to find out how many tweets in each country are marked as "Red", a sample analytical query using a SQL++ UDF 2 is shown in Figure 2.9. Since the data enrichment is evaluated together with the analytical query, the query response time can be long in the case of complex enriching UDFs. Also, as the enriched data is not persisted, the same enrichment needs to be computed multiple times for each incoming analytical query.

```
SELECT tweet.country Country, count(tweet) Num
FROM Tweets tweet
LET enrichedTweet = tweetSafetyCheck(tweet)[0]
WHERE enrichedTweet.safety_check_flag = "Red"
GROUP BY tweet.country;
```

Figure 2.9: An analytical query using SQL++ UDF 2

2.4.2 Option 2 - Enrich during Data Ingestion

In common use cases, the enriched data may be used repeatedly in analytical queries at different points in time. In such use cases, enriching data **lazily** in each analytical query separately can be expensive, as it wastes time evaluating the same UDF on the same data multiple times. In such cases, it can be beneficial to instead persist the enriched data and use it for all future analytical queries with similar needs. To allow faster responses to those queries, data enrichment in such use cases is often completed **eagerly** during the data ingestion process. Here we discuss three different approaches to enriching data during ingestion using Apache AsterixDB.

Approach 1 - External Programs

A naive approach to enrich data during ingestion would be to set up an external program that obtains/receives data from data sources, issues DML statements to enrich the collected data, and then inserts the enriched data into a dataset. A sample insert statement that enriches data using SQL++ UDF 2 and inserts the result into a target dataset "EnrichedTweets" [2] is shown in Figure 2.10. However, as discussed in Section 2.2.3, issuing repeated insert statements has significant overheads and would not scale well.

```
INSERT INTO EnrichedTweets(
  LET TweetsBatch = ([{"id":0, ...}, {"id":1, ...}, ...])
  SELECT VALUE tweetSafetyCheck(tweet)
  FROM TweetsBatch tweet
);
```

Figure 2.10: Enrich and insert collected tweets

Approach 2 - External Programs w/ Data Feeds

A user can use the basic data feeds feature introduced in Section 2.2.3 to improve ingestion performance. The data can first be ingested into a dataset using data feeds, then enriched and stored in another dataset by applying UDFs. A user could set up an external program that repeatedly issues the DML statement in Figure 2.11 to initiate data enrichment for ingested data. Depending on the arrival rate of incoming data, the user may issue a new DML statement as soon as the previous one returns to catch up with the ingestion progress when the arrival rate is high, or wait for a certain period to batch the ingested data when the arrival rate is low. Benefiting from data feeds, this approach consumes the incoming data efficiently, even when the data comes in fast, but a user still needs to set up an external program that constantly initiates the data enrichment. In addition, the data is unnecessarily materialized twice since all information in the tweets is kept in the enriched tweets as well.

[2] One can create the "EnrichedTweets" dataset using a DML statement similar to Figure 2.1.

```
INSERT INTO EnrichedTweets(
  SELECT VALUE tweetSafetyCheck(tweet)
  FROM Tweets tweet WHERE tweet.id NOT IN
  (SELECT VALUE enrichedTweet.id FROM EnrichedTweets enrichedTweet)
);
```

Figure 2.11: Enrich and insert ingested tweets

Approach 3 - Data Feeds w/ UDFs

In order to avoid the unnecessary materialization of incoming data and make the enriched data available to users as soon as possible, we may attach the data enrichment operation directly to the ingestion pipeline so that the ingested data is enriched before it arrives at storage. Apache AsterixDB allows users to attach certain UDFs to data feeds. As an example, a user could attach SQL++ UDF 1 (in Figure 2.6) to a data feed using the DDL statement in Figure 2.12. Incoming tweets are first received by the feed adapter, then parsed by the feed parser, and then enriched by the attached UDF. Finally, they are stored in the connected dataset. A Java UDF, such as the UDF in Figure A.4, can also be attached to a data feed.

```
CONNECT FEED TweetFeed TO DATASET EnrichedTweets
  APPLY FUNCTION USTweetSafetyCheck;
```

Figure 2.12: Attach a SQL++ UDF to a data feed

2.4.3 More Complex Enrichment

Challenges

In AsterixDB today, UDF 1 can be attached to a data feed directly, as it only accesses the incoming record and does not create any intermediate states. We call this kind of UDF a ***stateless*** UDF. UDF 2 is different from UDF 1, as UDF 2 accesses external resources (the

"SensitiveWords" dataset in the case of SQL++, or the equivalent local resource files in the case of Java) and creates intermediate states (such as in-memory hash tables) used for data enrichment. We call this kind of UDF a ***stateful*** UDF.

Attaching a stateful UDF to a data feed can be problematic since in some cases the referenced data can itself be modified during the ingestion process, in which case the intermediate states based on the referenced data need to be refreshed accordingly. Also, not all complex and stateful SQL++ UDFs can be applied to a continuously incoming data stream directly. To illustrate the challenges of applying complex and stateful SQL++ UDFs in the ingestion pipeline, here we discuss three possible computing models for attaching UDF 2 to a data feed.

Model 1 - Evaluate UDF per Record

A simple computing model for applying a stateful UDF to a feed is to evaluate the attached UDF against each incoming record separately. An incoming record is received and parsed by the feed adapter and parser first, then enriched and persisted in storage. An equivalent insert statement for enriching and persisting one record is shown in Figure 2.13. In this model, each collected datum is treated as a new constant record. The attached UDF evaluates each record separately, and any intermediate states will be refreshed from record to record. This allows the UDF to see data changes during the ingestion process, and it imposes no limitations on the applicable query constructs in attached UDFs. However, evaluating the UDF on a per-record basis may introduce a lot of execution overhead. This model cannot be applied in situations where the data arrives rapidly.

```
INSERT INTO EnrichedTweets(
LET tweet = { "id": ... }
SELECT VALUE tweetSafetyCheck(tweet));
```

Figure 2.13: Enrich and insert a constant record

Model 2 - Evaluate UDF per Batch

To mitigate the execution overhead, one alternative is to batch the collected incoming records, apply the UDF to the batch, and store the enriched records. An equivalent insert statement for enriching a batch of records was shown in Figure 2.10. The records within one batch are enriched using the same reference data, and reference data changes are captured between batches. A larger batch leads to lower execution overhead but less immediate sensitivity to reference data changes; the converse is also true. A user may choose a balance between ingestion performance and sensitivity to reference data changes by tuning the batch size.

Model 3 - Stream Datasource

To further reduce execution overheads, the system could attempt to treat the incoming data stream as an infinite dataset and evaluate the attached UDF as if the stream is a normal dataset. An equivalent insert statement is shown in Figure 2.14[3].

```
INSERT INTO EnrichedTweets(
SELECT VALUE tweetSafetyCheck(t)
FROM FEED Tweets t);
```

Figure 2.14: Enrich and insert records from a feed

This model would be more efficient than the previous two, as the attached UDF is initialized once for all incoming data. Any pre-computation for enriching the incoming data occurs only once and is used for all incoming data. Although this model would provide the best ingestion performance since it has the smallest execution overhead, it cannot be used when the attached UDF is ***stateful***. Taking SQL++ UDF 2 as an example, when we attach this UDF to a data feed and use this model to compute it, the evaluation would become a

[3]The keyword "FEED" is not an actual supported datasource in SQL++, so one cannot run this DDL statement in the Apache AsterixDB system. Here we use it to conceptually denote a continuous feed datasource.

join operation between the "SensitiveWords" dataset and the never-ending feed data source. When there is a more complicated UDF, such as multi-level join and group-by, the evaluation could create more intermediate states and become even harder to evaluate using this model. Here we list three different scenarios of evaluating UDF 2 using Model 3, depending on the join algorithm and the size of the "SensitiveWords" dataset.

1. *Hash Join with a small "SensitiveWords" dataset*

   The evaluation of a hash join operation consists of two phases: build and probe [79]. In the build phase, the "SensitiveWords" dataset would be built into a hash table. In the probe phase, the data coming from the Twitter feed would then use the hash table to find the matching records in the "SensitiveWords" dataset.

   When the "SensitiveWords" dataset is small, the created hash table can be kept in memory. This allows incoming data to continuously probe the in-memory hash table for enrichment while the ingestion continues as shown in Figure 2.15. This appears to be a perfect model for this case, but it cannot incorporate the new changes to the "SensitiveWords" dataset, as the in-memory hash table would be built once and then used throughout the streaming ingestion process.

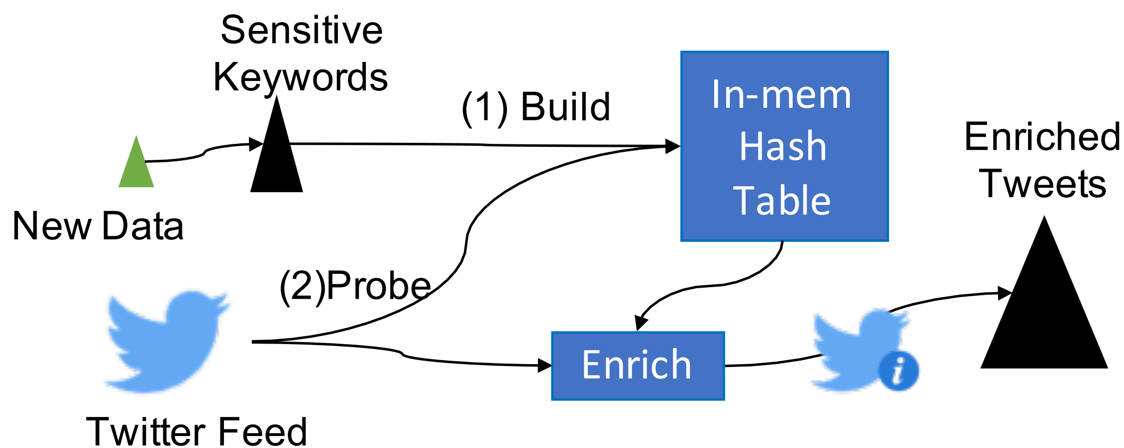


Figure 2.15: Case 1: Small SensitiveWords dataset

2. *Hash Join with a big "SensitiveWords" dataset*

When the "SensitiveWords" dataset is large, part of its data will be spilled to disk for the next round of the join [79]. This is shown in Figure 2.16. The hash join algorithm expects to process such spilled data recursively, after reading "all" data from Twitter, but of course the tweets will not stop coming. Thus, this model cannot be used in this case.

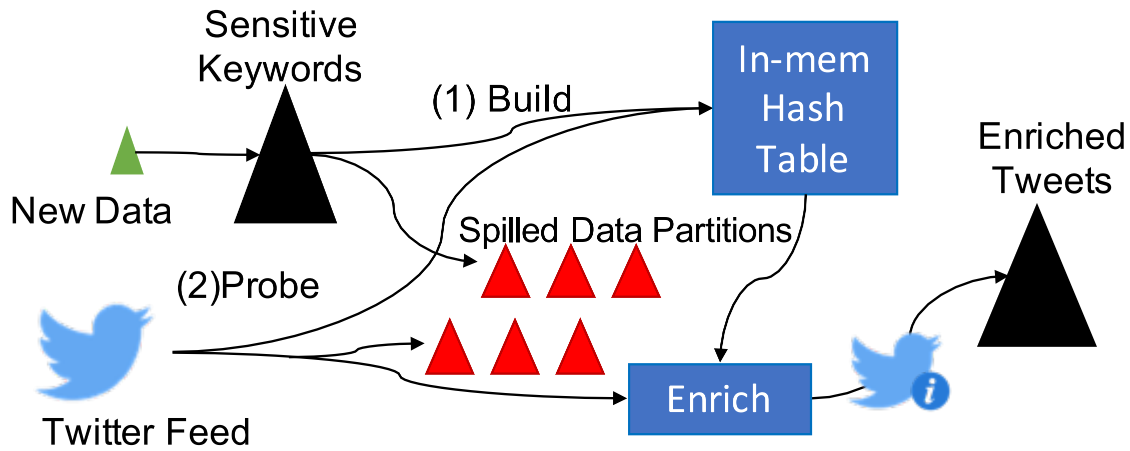


Figure 2.16: Case 2: Big SensitiveWords dataset

3. *Index Nested Loops Join*

If there is an index on the "country" attribute of the "SensitiveWords" dataset, the SQL++ query compiler may choose the index nested loop join algorithm to compute the join. In this case, the incoming data can be used to look in the index first, then find the matched records for enrichment, as shown in Figure 2.17. By choosing this join algorithm manually for this specific join case, one could avoid creating intermediate states during the enrichment operation and thus see the new data changes directly. However, this approach is not applicable to more general use cases where the indexes on referenced datasets may not always exist, and/or where an enrichment UDF contains other operations that create intermediate states. As an example, the function in Figure 2.18 red-flags a tweet if it comes from one of the top 10 countries containing

more keywords than others. The top 10 countries list would not be refreshed unless the attached UDF is evaluated again.

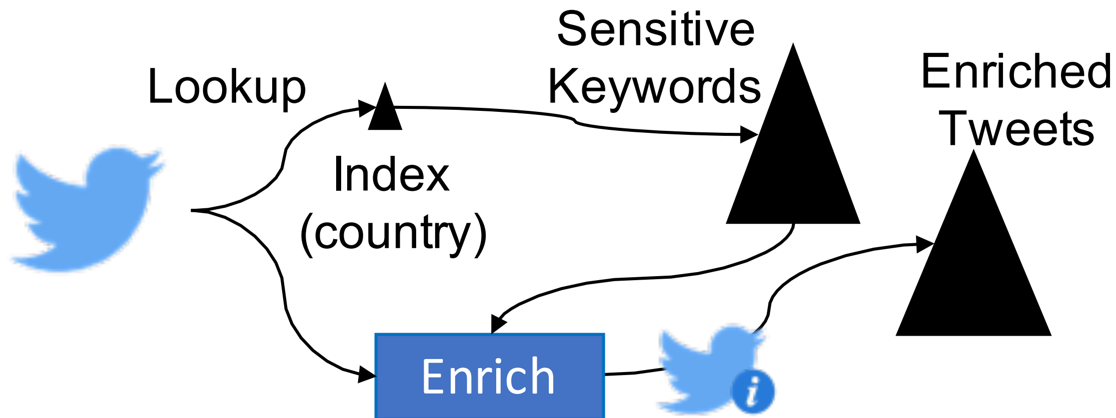


Figure 2.17: Case 3: Enrich with an available index

```
CREATE FUNCTION highRiskTweetCheck(t) {
  LET high_risk_flag = CASE
    t.country IN (SELECT VALUE s.country
      FROM SensitiveWords s GROUP BY s.country
      ORDER BY count(s) LIMIT 10)
    WHEN true THEN "Red" ELSE "Green"
  END
  SELECT t.*, high_risk_flag
};
```

Figure 2.18: Enrichment UDF with a nested subquery

In the current Apache AsterixDB release, data feeds actually use this streaming model to evaluate any attached UDFs on an ingestion pipeline, so the attached UDFs are limited to be **stateless**. In order to support **stateful** data enrichment UDFs and allow users to use the full power of SQL++ in more complex data enrichment use cases, we need to create a new data ingestion framework that evaluates attached complex **stateful** UDFs properly.

2.5 Framework Building Blocks

As discussed in Section 2.4.3, only models 1 and 2 support complex data enrichment during data ingestion and capture any reference data changes at the same time. We have thus built a new data ingestion framework based on model 2, as it provides flexibility by allowing users to choose the right batch sizes for their use cases. In this section, we describe the design of this new framework and the optimization techniques that we used for improving its performance.

2.5.1 Predeployed Jobs

Following model 2, our new ingestion pipeline consists of two independent Hyracks jobs: an *intake job* and an *insert job*. A sample ingestion pipeline on a three-node cluster is shown in Figure 2.19. The intake job contains the feed adapter and parser, and this job runs continuously throughout the lifetime of the ingestion process. The insert job takes a batch of records from the intake job, enriches them by applying the attached UDFs, and inserts the enriched records into a dataset. It runs repeatedly, being invoked once per batch, during ingestion. In each invocation, the insert job sees the updates to a referenced data record before it is first accessed. Updates after that are picked up by the next invocation[4].

The insert job in Figure 2.19 is constructed using the query in Figure 2.10. For every collected batch of records from the intake job, we replace the array of constant records (in TweetsBatch) with the collected batch and execute it. As discussed in Section 2.2.2, a query in AsterixDB is optimized and compiled into a job specification first, then distributed to the cluster for execution. Since the insert job is executed repeatedly, we utilize *parameterized predeployed jobs* to avoid redundant query compilation and job distribution costs.

[4]This follows the record-level consistency model provided in AsterixDB (and most other NoSQL databases).

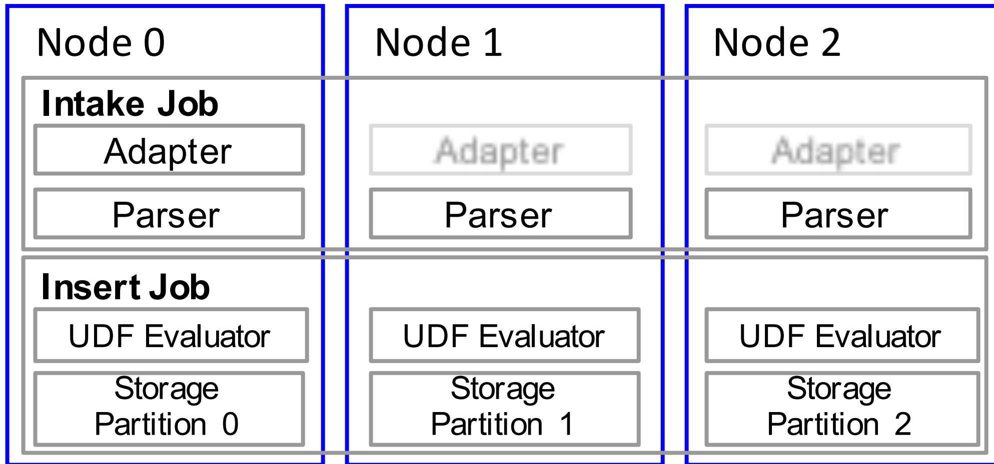


Figure 2.19: Ingestion pipeline using insert jobs

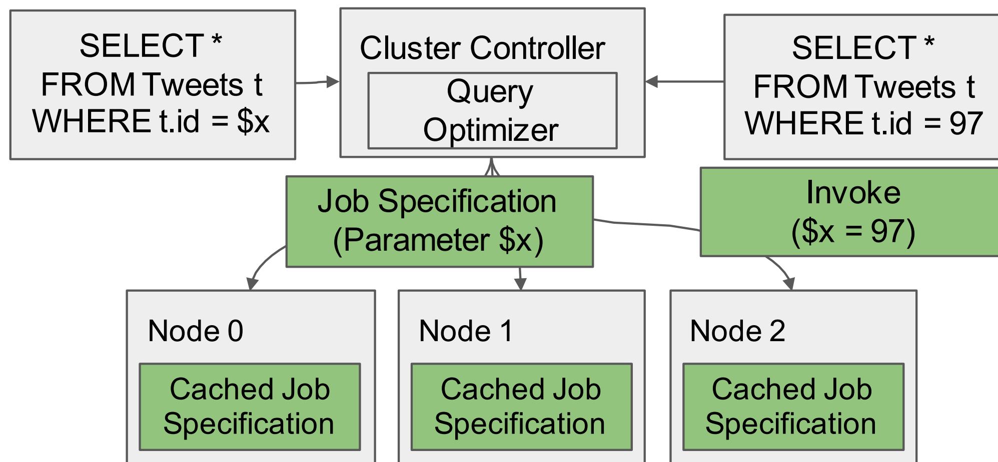


Figure 2.20: Parameterized predeployed job

Parameterized predeployed jobs are not unlike prepared queries in relational databases. As shown in Figure 2.20, a user can choose to predeploy a query with specified parameters. This query is optimized and compiled normally, and then the compiled job specification is predeployed to all nodes in the cluster. This job specification is then cached on the cluster nodes. When a user wants to run this query with a particular parameter, instead of repeating the entire query compilation and distribution process, an invocation message with the new invocation parameter is sent. Using this technique, Figure 2.19's insert job is distributed as a predeployed job in the cluster before the feed starts. When the intake job obtains a new batch of records, it invokes a new insert job with the collected batch as the parameter.

2.5.2 Layered Ingestion Pipeline

Repeatedly executing the insert job allows any intermediate states created in the UDF evaluation to be refreshed so that any data changes will be used for enriching the incoming data. It should be noted that the evaluation of an insert job, similar to the evaluation of an insert query, will have to wait for the storage log to be flushed to finish properly. Also, since the UDF evaluation and storage operations work sequentially in an insert job, UDF evaluation can be blocked while waiting for the downstream data to be written into storage. To fully utilize the cluster's computing resources and improve overall throughput, we further decompose the insert job into a computing job and a storage job so they can work concurrently. The decoupled ingestion framework is shown in Figure 2.21.

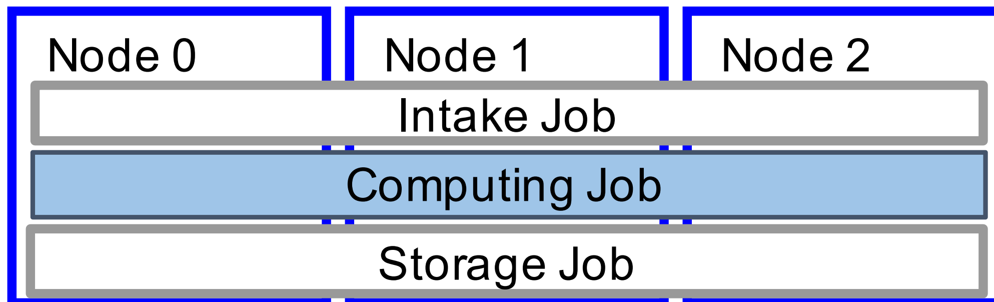


Figure 2.21: Decoupled ingestion framework

In the decoupled ingestion framework, the intake job handles data from external data sources, the computing job evaluates the attached UDFs, if any, and the storage job writes the enriched data into storage. The intake job and storage job begin to run when the data feed starts, while the computing job in between them is run repeatedly as data batches come in. As in the previous discussion, the computing job is distributed as a predeployed job to reduce execution overhead. Similar to the insert job in Section 2.5.1, an invocation of the computing job will see the updates to a referenced record before it is first accessed by the job.

2.5.3 Partition Holders

In the decoupled ingestion framework, data frames are passed from the intake job to a computing job, and then from the computing job to the storage job. Currently, data exchanges in Hyracks are limited to being within the scope of a job; one job cannot access data frames from another job at runtime. As data exchanges between jobs in the decoupled framework are frequent, we needed to add an efficient mechanism to allow data to be exchanged between jobs.

Considering that the operators in a job each work on data partitions, by aligning the output partitions of one job with the input partitions of the other job, data frames can be shipped from one job to the other efficiently through in-memory structures. For this, we introduce a new type of operator in Hyracks - a partition holder - to enable efficient data exchanges between jobs.

A partition holder operator "guards" a runtime partition by holding the incoming data frames in a queue with a limited size. There are two types of partition holders, active and passive, as shown in Figure 2.22. An active partition holder follows the default **push** strategy in Hyracks; it receives data frames from other jobs and pushes them to its downstream operators actively. A passive partition holder implements a **pull** strategy; it receives data frames from its upstream operators and waits for other jobs to pull them. Each partition holder has a unique ID that is associated with its partition number. When a new partition holder is created, it registers with the local partition holder manager. Jobs sending/receiving data to/from another job can locate the corresponding partition holders through local partition holder managers. In the decoupled ingestion framework, we add a passive partition holder to the tail of the intake job so that the computing jobs can request and receive data in batches. An active partition holder is added to the head of the storage job so that computing jobs can push the enriched data on to the storage job.

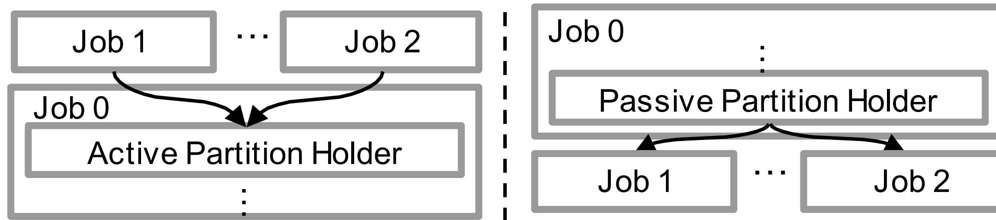


Figure 2.22: Partition holders

2.6 The New Ingestion Framework

Following the high-level design in Section 2.5, we now detail the new ingestion framework in AsterixDB to support data enrichment with reference data updates. We describe how we orchestrate different components in the new ingestion framework in Section 2.6.1, and we delve into the lower-level constructs of the framework in Section 2.6.2.

2.6.1 Ingestion Life Cycle

As we discussed earlier, AsterixDB is a parallel data management system that runs on a cluster of commodity machines. In an AsterixDB cluster, one (and only one) node runs the Cluster Controller (CC) that takes in users' queries and translates them into Hyracks jobs. Only the CC can start new jobs, and it keeps track of the progress of the running jobs in case of any failures. All worker nodes in the cluster run a Node Controller (NC) that takes computing tasks from the CC. The CC and NC can coexist on the same node.

In the new ingestion framework, there are two long running jobs, the intake and storage jobs; there is one short lived, but repeatedly invoked, computing job. When there are multiple data feeds running concurrently, each of them is compiled and executed independently. In order to monitor data feed jobs, we created an Active Feed Manager (AFM) on the CC to manage the lifecycle of data feeds. The AFM tracks all active data feeds and helps them to

invoke new computing jobs when new data batches arrive.

When a user submits a start feed request, the CC creates the intake, computing, and storage jobs based on a compiled job specification that is generated from a query template similar to Figure 2.10. The intake and storage job run directly, and the computing job is predeployed into the cluster for later invocations. The AFM maintains the mappings of data feeds to predeployed computing jobs so that it can invoke new computing jobs for each data feed separately.

When an intake job starts, it asks the AFM on the CC to invoke the first computing job and to keep invoking new computing jobs when the previous one finishes. After that, the intake job begins ingesting data from an external data source, adding data records into its queue, and waiting for the computing job to collect the ingested data. The current computing job takes a data batch from the intake job, enriches its records with an attached UDF if any, and then pushes the enriched data batch to the storage job. When this computing job finishes, the AFM on the CC will then start a new computing job to continue the processing.

When the user stops a feed, the intake job first stops taking new data and then adds a special "EOF" data record into its queue. When a computing job sees this record, it will finish its current execution with the collected data without waiting for a complete batch. The intake job finishes when all ingested data has been consumed. When the intake job finishes, it notifies the AFM to stop invoking new computing jobs for this feed. When the last computing job for the feed finishes, the storage job stops accordingly.

2.6.2 New Ingestion Architecture

The new ingestion framework consists of the intake, computing, and storage jobs. All jobs run on all nodes in an AsterixDB cluster. As explained in Section 2.2.2, a Hyracks job contains

operators and connectors. In order to demonstrate how data is processed and transported in the new ingestion framework, Figure 2.23 shows the composition of the framework running on three nodes at the operator and connector level:

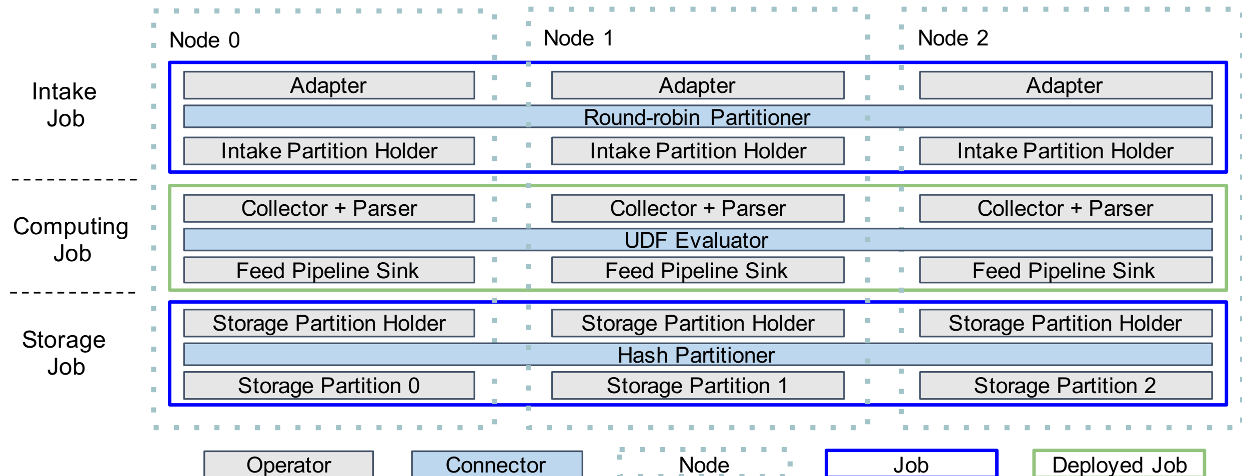


Figure 2.23: The new ingestion framework

- *The **intake job*** obtains/receives data from external data sources. The data enters the system through the Adapter. The Adapter collects data as raw bytes and arranges them into data frames for transportation purposes in the system. A user may choose to activate the Adapter on one or more nodes depending on the expected load. The ingested data frames are then fed through the Round-robin Partitioner to be distributed in a round-robin fashion. Since the attached UDFs can be expensive, distributing the incoming data evenly can help to minimize the overall execution time of the computing job. The partitioned data is forwarded to the Intake Partition Holder, which is implemented as a passive partition holder, which then waits for computing jobs to pull the data.

- *The **computing job*** evaluates the attached UDF to enrich data batches. A computing job starts by collecting a data batch from a local intake partition holder. The obtained data batch is first parsed by the Parser and then fed to the UDF Evaluator for data enrichment. Depending on the attached UDF, the UDF evaluator could be a Java

program that runs on each node independently, or it could be a group of operators produced by compiling a complex SQL++ UDF. In either case, local resource files (for Java UDFs) or reference datasets (for SQL++ UDFs) may be accessed, and/or intermediate states might be created as well. After being enriched, the data is pushed to the Feed Pipeline Sink to be forwarded to the storage job.

- *The* ***storage job*** receives enriched data and stores it to disk. The enriched data is first received by the Storage Partition Holder, which is implemented as an active partition holder. A feed pipeline has one Storage Partition Holder on each node, and the Storage Partition Holder receives the enriched data from all local partitions of the collocated computing job. The Storage Partition Holder pushes the received enriched data actively to the connected Hash Partitioner. The Hash Partitioner partitions the enriched data records by their primary keys so they can be stored in the appropriate Storage Partitions.

2.7 Experiments

In this section, we present a set of experiments that we have conducted to evaluate the new ingestion framework. We compared the basic ingestion performance of the new ingestion framework with that of the existing Apache AsterixDB ingestion framework. We examined the data enrichment performance of the new ingestion framework using various Java and SQL++ UDFs. Finally, we investigated the speed-up and scale-out performance of the new ingestion framework for more complex data enrichment workloads. Our experiments were conducted on a cluster which is connected with a Gigabit Ethernet switch. Each node had a Dual-Core AMD Opteron Processor 2212 2.0GHz, 8GB of RAM, and a 900GB hard disk.

2.7.1 Basic Data Ingestion

When ingesting data without an attached UDF, the computing job in the new ingestion framework simply moves data from the intake job to the storage job. By comparing the data ingestion performance of the new ingestion framework to that of the current AsterixDB ingestion framework without UDFs, we can examine the execution overhead introduced by managing and periodically refreshing the computing job in the new ingestion framework.

For this purpose, we compared the throughput of both the current and new frameworks for continuous tweet ingestion. We continuously fed tweets into both ingestion frameworks and measured the throughput of each while consuming 10,000,000 tweets. Each tweet record is around 450 bytes. The results are shown in Figure 2.24. To make sure a single intake node did not become a bottleneck for the ingestion performance, we also tested a "balanced version" of both the current and new ingestion framework by having all nodes in the cluster act as intake nodes. We refer to the experiments on the current framework as "Static Ingestion", on the new framework as "Dynamic Ingestion", on the balanced version of the current framework as "Balanced Static Ingestion", and on the balanced version of the new framework as "Balanced Dynamic Ingestion".

To explore how batch size affects the ingestion performance of the new ingestion framework, we experimented with three different batch sizes in Dynamic Ingestion, including 420 records/batch (1X), 1680 records/batch (4X), and 6720 records/batch (16X). Also, we varied the size of the cluster from 1 node to 24 nodes to see how ingestion performance varies with increased cluster sizes.

As we can see in Figure 2.24, the ingestion performance of Static Ingestion remained the same as the cluster size increased. This is because data intake and parsing are coupled in the current ingestion pipeline. In this case, the ingestion performance was limited by the parsing bottleneck on a single intake node. In contrast, Balanced Static Ingestion kept improving as

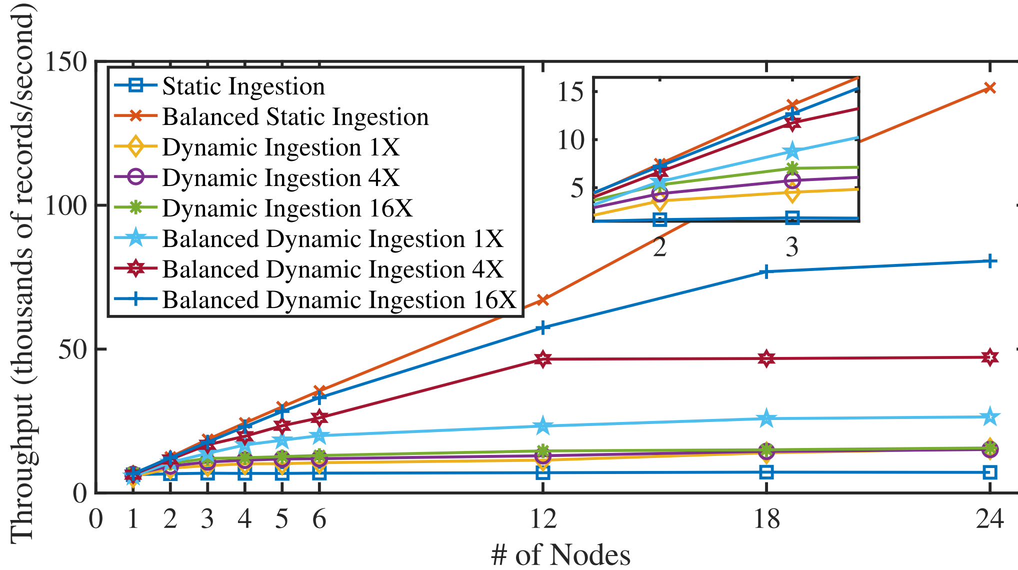


Figure 2.24: 10M tweets ingestion speed-up over 24 nodes

more nodes participated in parsing and ingesting the data. In the new ingestion framework, data parsing and intake are decoupled, so even for a single intake node (Dynamic Ingestion), the intake performance increased as more nodes participated when the cluster size is small.

Focusing on the results for Dynamic Ingestion, the ingestion performance improved as the batch size increased since there were fewer computing jobs initiated for data enrichment; different batch sizes' throughputs eventually converged to the same level, as they were limited by the available resources on the single intake node. For the Balanced Dynamic Ingestion, the intake load is split onto all nodes, so its throughput kept growing as nodes were added.

Comparing the performance difference of Balanced Static Ingestion and Balanced Dynamic Ingestion, we can see the execution overhead introduced by repeatedly invoking computing jobs in the new ingestion framework. The execution overhead of invoking computing job increased with the cluster size. As a result, Balanced Dynamic Ingestion had similar throughput as Balanced Static Ingestion when the cluster size is small but started to fall behind as the cluster continued growing. Note that given 24 nodes, the refresh rates (number of computing jobs per second) were 68, 27, and 10 for batch sizes of 1X, 4X, and 16X

respectively. We will further explore this with UDFs attached in next section.

2.7.2 Data Enrichment with UDFs

We now turn to the performance of the new ingestion framework in enriching data during data ingestion. We designed four sample use cases where the attached UDFs cover several common operations used in database queries, including join, group-by, order-by, similarity join, and spatial join. The four use cases are as listed below. The reference datasets are SafetyRatings, with 500,000 records and 74 bytes each, ReligiousPopulations, with 500,000 records and 137 bytes each, SuspectsNames, with 5,000 records and 150 bytes each, and MonumentList, with 500,000 records and 94 bytes each.

1. *Safety Rating:* Given a list of countries and their corresponding safety ratings, enrich a tweet with a safety rating based on its "country" field value. (Hash join. See Appendix A.1)

2. *Religious Population:* Given the population of each religion in every country, enrich a tweet with the overall religious population based on its "country" field value. (Group-by. See Appendix A.2)

3. *Largest Religions:* Given the population of each religion in every country, enrich a tweet with the three largest religions according to its "country" field value. (Order-by. See Appendix A.3)

4. *Fuzzy Suspects:* Given a list of suspects' names, enrich a tweet with the possible suspects whose name's edit distance to the tweet user's screen name, after removing all special characters, is less than five characters. (Java string processing, Similarity join. See Appendix A.4)

5. *Nearby Monuments:* Given a list of monuments and their coordinates, enrich a tweet with the monuments within 1.5 degrees of the tweet's location. (Index nested loop spatial join. See Appendix A.5)

All of these enrichment UDFs are stateful, and their evaluations involve the challenges that we discussed in Section 2.4.3. As we have mentioned, the current ingestion pipeline of AsterixDB doesn't support such stateful SQL++ UDFs on its ingestion pipeline. Java UDFs attached on the current AsterixDB ingestion pipeline can only handle reference data without updates. For comparison purpose, we experimented with Java UDFs in current AsterixDB and denote the results as "Static Enrichment w/ Java".

The new ingestion framework supports both Java and SQL++ UDFs and reference data with updates. We tested both Java and SQL++ UDFs and varied the batch sizes from 420 records/batch to 1680 records/batch to 6720 records/batch to see how batching in the new ingestion framework affects performance. We denote the Java cases as "Dynamic Enrichment w/ Java 1X", "Dynamic Enrichment w/ Java 4X", and "Dynamic Enrichment w/ Java 16X " and the SQL++ cases as "Dynamic Enrichment w/ SQL++ 1X", "Dynamic Enrichment w/ SQL++ 4X", and "Dynamic Enrichment w/ SQL++ 16X" respectively.

In order to measure the performances of both Static Enrichment and Dynamic Enrichment, we deployed the system on a 6-node cluster and fed tweets to the ingestion pipeline for data enrichment continuously. We measured the throughput (records / second) of the system spent while enriching 1,000,000 tweets, as shown in Figure 2.25 (in log scale). The refresh periods (i.e., execution time per batch) of Dynamic Enrichment w/ SQL++ are shown in Figure 2.26.

In most of the use cases, except for Nearby Monuments (to be discussed shortly), Static Enrichment offered higher throughput than Dynamic Enrichment. This is because Static Enrichment only loaded reference data once and then reused its stale intermediate states

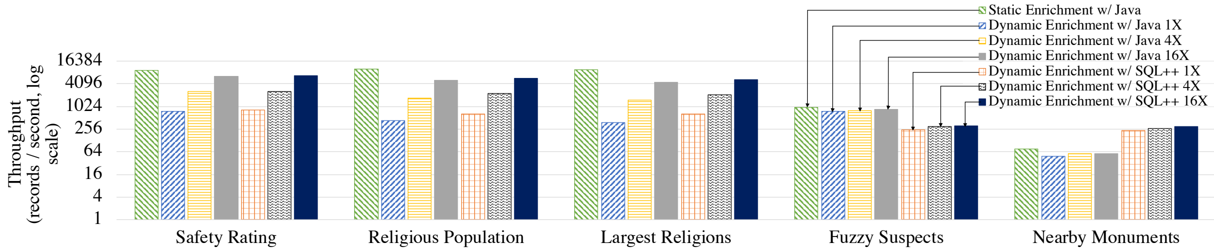


Figure 2.25: 1M tweets Ingestion with UDFs (log scale)

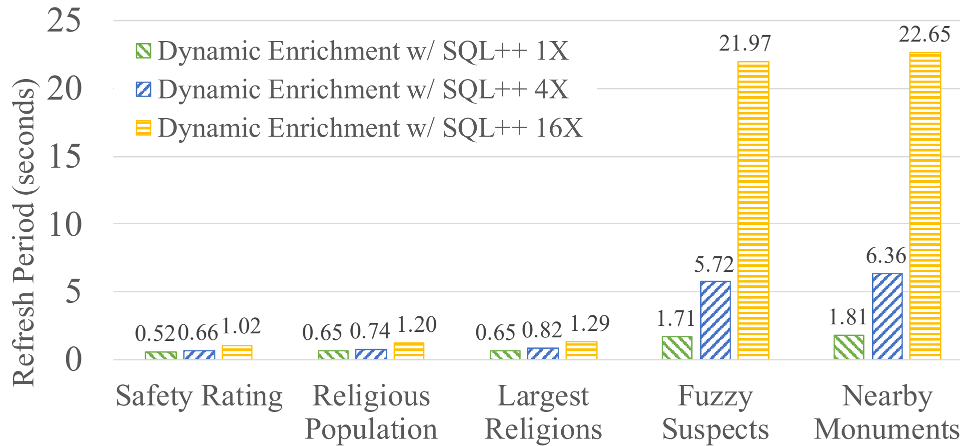


Figure 2.26: Refresh periods under different batch sizes

throughout the whole ingestion process. Dynamic Enrichment, however, refreshed and reconstructed those intermediate states from batch to batch. This allowed Dynamic Enrichment to capture data changes to reference data but with the overhead of repeatedly invoking computing jobs. For both Java and SQL++ UDFs, the throughput increased with larger batch sizes since they led to fewer computing jobs and a smaller execution overhead. Accordingly, the refresh periods grew, as there were more records to be enriched in larger batches.

For Fuzzy Suspects and Nearby Monuments, the throughput did not improved that much with increased batch size. The reason was that the computation costs of edit distance and spatial join were high and proportional to the cardinality of the incoming data. Job initialization and management overheads were small relative to these costs. Thus, increasing the batch size didn't improve the throughput significantly. In Fuzzy Suspects in particular, the attached SQL++ UDF not only computed edit distance but also invoked a Java UDF

for removing special characters. This introduced extra data serialization/deserialization and shuffling cost. In Nearby Monuments, we created an R-Tree index for the monuments' location in the reference dataset. The use of the index allowed the SQL++ UDF to outperform the Java UDF in this case by performing index lookups on partitioned reference data.

2.7.3 Data Enrichment with Updates

During data ingestion and enrichment, as the referenced data is being updated, the update rate may affect the ingestion performance. In order to further investigate this, we conducted an additional experiment. For each of the use cases (Safety Rating, Religious Population, Largest Religions, Fuzzy Suspects, Nearby Monuments), we created a client program that sends reference data updates to AsterixDB through a data feed. We measured the resulting throughput impact by varying the update rate (records/second) on a 6-node cluster during 100,000 tweets' ingestion and enrichment.

As we can see from Figure 2.27, reference data updates affect the ingestion and enrichment performance differently depending on the cardinality of the referenced dataset and the access method used in data enrichment operations. The throughputs of all cases dropped when the update rate first changed from none to one record per second. This was due to the resulting increased cost in accessing reference data. AsterixDB uses log-structured merge-trees (LSM Trees) in its storage [5]. Updates to a dataset will activate the in-memory component of its LSM structure and thereby change how the system accesses data even at the low rate of one record per second. This added additional data fetching, locking, and comparison costs to reference data access in computing jobs, which then slowed down the ingestion throughput. Since Fuzzy Suspect had the smallest reference dataset among all, it was the least affected by the updates. When increasing the update rate from there, the throughput then decreases gradually. For Nearby Monuments, the referenced dataset

was probed throughout a computing job (index join) for data enrichment whereas in other cases, the referenced dataset was scanned once at the beginning of each computing job (hash join). As a result, Nearby Monuments' performance was less affected at low update rates but started to slip when the update rate was high. The throughput of Nearby Monuments, under the 400 records/second update rate, was only 24% of that without updates. Compared with Safety Rating, the most affected among the other cases, the ratio was 52%.

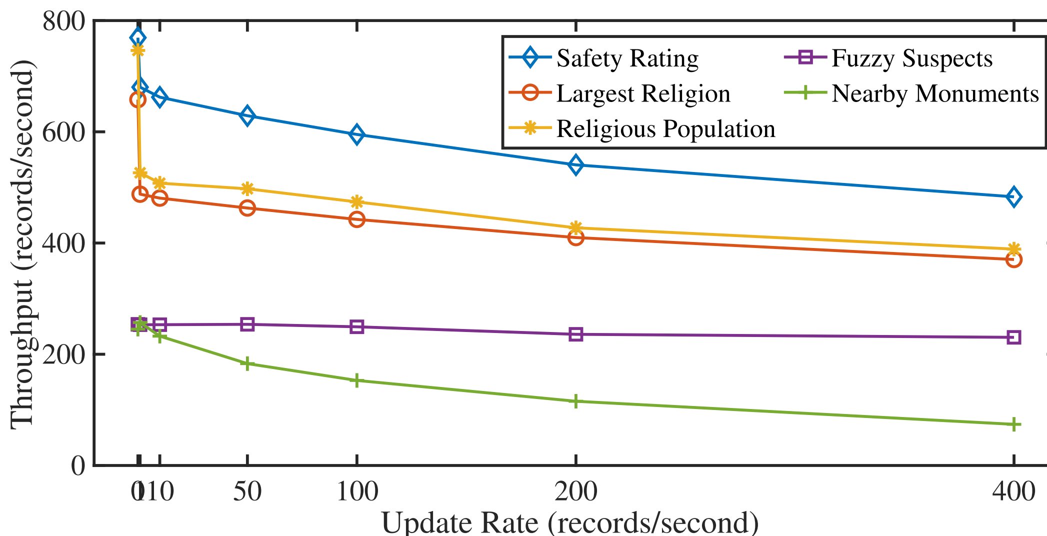


Figure 2.27: Reference data update

2.7.4 Scale-out Experiments

Reference Data Scale-out

In the new ingestion framework, since the intermediate states are refreshed repeatedly, the size of the reference data could become a important factor for the data ingestion and enrichment performance. In this section, we explore how the new ingestion framework scales with the size of the reference datasets. We started with the reference datasets in Section 2.7.2 and increased their sizes to 2X, 3X, and 4X, together with increasing the cluster size to 12 nodes, 18 nodes, and 24 nodes correspondingly. Similarly, we continuously fed tweet data

for data enrichment using the same set of SQL++ UDFs and measured the throughput after 1,000,000 tweets with 6720 records/batch. As we can see from the results in Figure 2.28, the throughput dropped slightly when we increased the size of the cluster due to the increasing execution overhead on a larger cluster. The new ingestion framework scaled well with the reference data size.

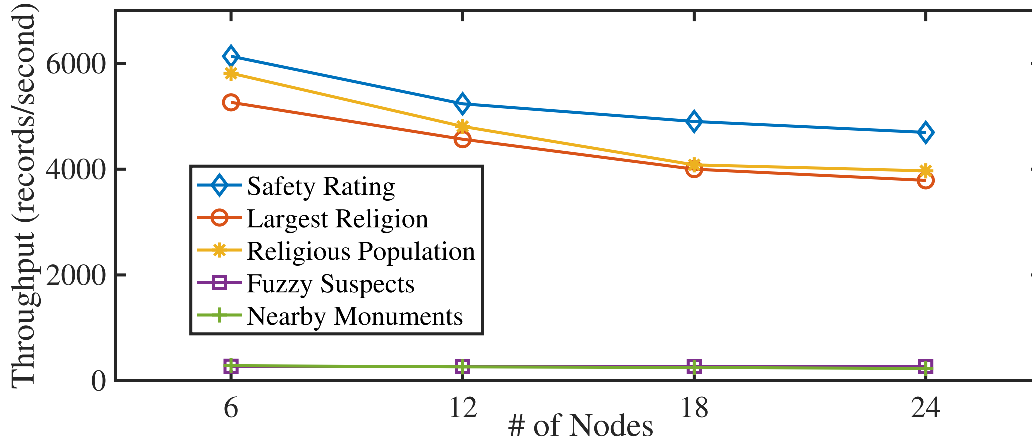


Figure 2.28: Reference data scale-out

Ingestion Data Scale-out

In order to further investigate the performance of the new ingestion framework for large scale data enrichment, we designed three new complex data enrichment use cases that add more information to the incoming tweets. The more complex a data enrichment function is, the more performance impact it will have on the whole ingestion framework. We tested the new framework with these new cases to see how it scales. The additional use cases are listed below. The reference datasets are ReligiousBuildings, with 10,000 records and 205 bytes each, Facilities, with 50,000 records and 142 bytes each, SensitiveNames, with 1,000,000 records and 155 bytes each, AverageIncome, with 50,000 records and 99 bytes each, DistrictArea, with 500 records and 121 bytes each, Residents, with 1,000,000,000 records and 124 bytes each, and AttackEvents, with 5,000 records and 179 bytes each.

6. *Suspicious Names*: Include the number of nearby facilities grouped by their types, the three closest religious buildings within three degrees of the tweet's location, and information about suspicious users who have the same name as the tweet's author. (See Appendix A.6)

7. *Tweet Context*: Include the average income for the district where the tweet was posted, the number of facilities in this district grouped by their types, and the ethnicity distribution of the residents in this district. (See Appendix A.7)

8. *Worrisome Tweets*: Include the religion names of the religious buildings within three degrees of the tweet and the number of terrorist attacks in the past two months that were related to that religion. (See Appendix A.8)

To demonstrate the complexity of these additional use cases, we compared their enrichment performance with that of "Nearby Monuments", the most complex UDF from the previous experiment. We measured their throughput on the new ingestion framework for 100,000 tweets enrichment on a 6-node cluster. As shown in Figure 2.29, the added use cases had different complexities, and different use cases benefited from batch size changes differently. In the Tweet Context use case, there were multiple expensive spatial joins between the referenced datasets before joining with the tweets. Increasing the batch size reduced the computation cost and thus increased the overall ingestion throughput. In the other cases, the tweets mostly joined with the reference datasets sequentially. Thus, increasing the batch sizes offered limited improvements in the Nearby Monuments, Suspicious Names, and Worrisome Tweets use cases.

The performance of scaling out the new framework is determined by the cluster size, batch size, and UDF complexity. Although increasing the number of nodes for computation can reduce the execution time of a computing job, it also introduces additional execution overhead for executing jobs on a larger cluster, so adding more resources may not always improve

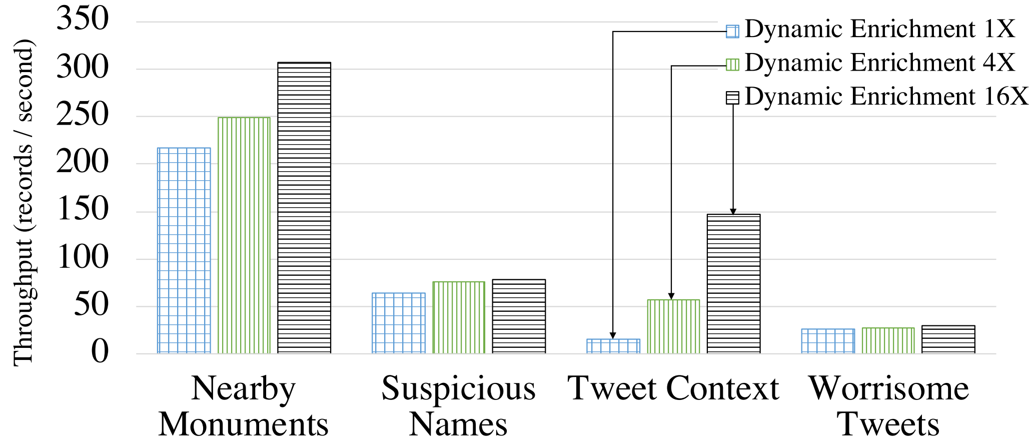


Figure 2.29: UDF complexity comparison

the overall ingestion time. Given a simple enrichment UDF, a small batch size, and a large cluster, the speed-up performance might be bounded by the batch execution overhead. To explore the relationship of these three factors, we experimented with the speed-up performance using different batch and cluster sizes.

We let the framework ingest and enrich 100,000 tweets using all seven UDFs. For each UDF, we measured its throughput on a 6-node cluster and a 24-node cluster separately and computed the resulting speed-up. We repeated this computation for each UDF for three different batch sizes, namely 420 records/batch (1X), 1680 records/batch(4X), and 6720 records/batch (16X), and we show the speed-up of each batch size in Figure 2.30.

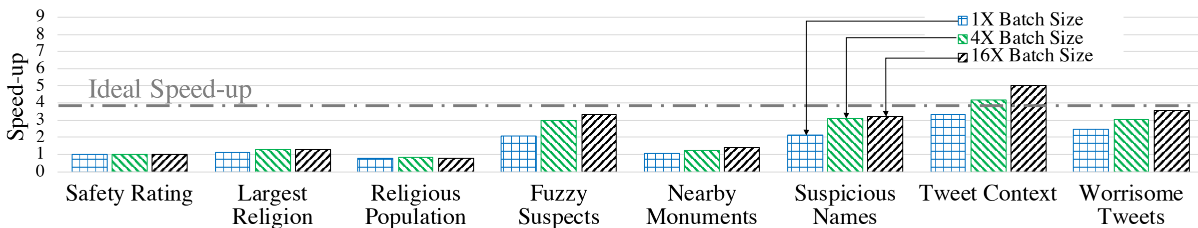


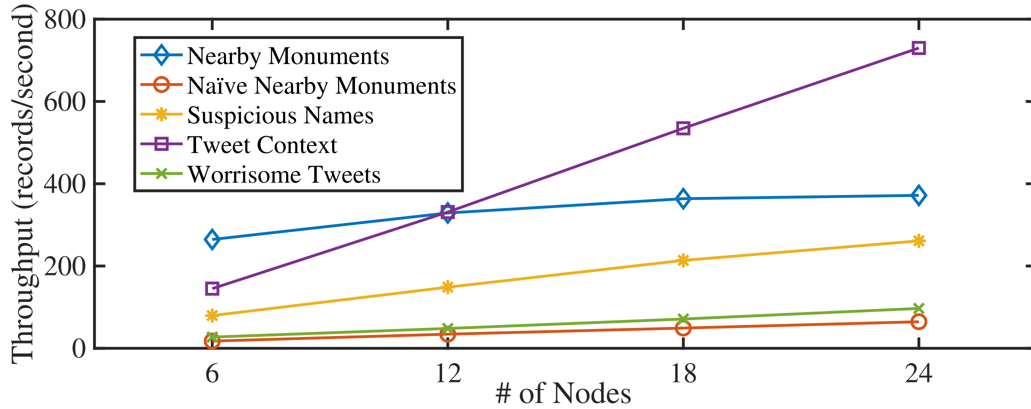
Figure 2.30: 100K tweets ingestion speed-up for 24 vs. 6 Nodes with different batch sizes

Since the UDFs in the Safety Rating, Religious Population, and Largest Religions use cases were relatively simple and their refresh period is already very low as shown in Figure 2.26, adding more resources yielded limited improvements to the execution times of their comput-

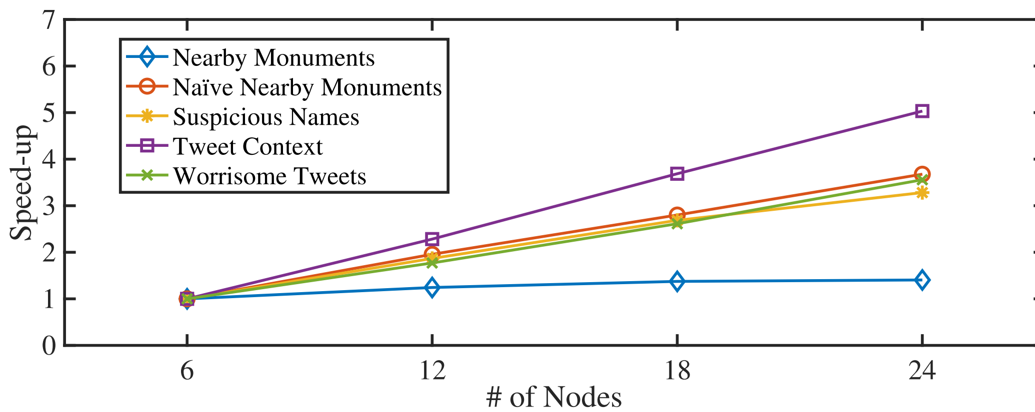
ing jobs. At the same time, their execution overhead grew as the cluster size increased. As a result, their speed-up is relatively poor. In Nearby Monuments, the Index Nested Loop Join didn't benefit from the batch size much as the overall index probing cost is not related to the batch size but mainly to the incoming data cardinality. In contrast, the other UDFs (Fuzzy Suspects, Suspicious Names, Tweet Context, and Worrisome Tweets) each improved with more resources. For Tweet Context in particular, not only were there 4x as many nodes participating in the computation, but the added resources (particularly memory) also allowed the join process to finish earlier. This enabled the system to obtain more than the ideal 4x speed-up. For a given volume of tweets, the bigger the batch size is, the fewer computing job invocations are needed for enriching these tweets, so the speed-up performance is better as the execution overhead increase from the cluster size growth is smaller.

In order to see how ingestion performance improves when adding more resources, we also evaluated the speed-up behavior of the four most complex UDFs (Nearby Monuments, Suspicious Names, Tweet Context, and Worrisome Tweets). To avoid the use of index in Nearby Monuments becoming a performance bottleneck, we used a query hint to add one Naive Nearby Monument use case that enriches the tweets with the same information without using the index. We fed the new ingestion framework 100,000 tweets and varied the cluster size from 6 nodes up to 24 nodes to see how throughput changes with at batch size of 6720 records/batch. The experimental results are shown in Figure 2.31.

As shown, the ingestion and enrichment performance improved, as more available computing resources were added. The performance gain started to level off when the cluster size kept increasing, as the query execution overhead of a larger cluster started to take away the speed-up benefits for the given reference data sizes. For Nearby Monuments in particular, the Index Nested Loop Join algorithm needed to broadcast the incoming tweets to all nodes to look for intersecting monuments. This limited its speed-up when the cluster size becomes large. In contrast, Naive Nearby Monuments started with a very low throughput which gradually



(a) Throughput



(b) Speed-up

Figure 2.31: 100K tweets ingestion speed-ups

increased as we grew the cluster. The reason was that its reference data monument list was split across more nodes that can then be joined with the incoming tweets concurrently.

2.8 Related Work

Data enrichment has been widely used in data analysis applications in which the collected data contains limited information and needs to be correlated with existing knowledge to revealing higher-level insights. Abel *et al.* proposed to construct Twitter user profiles by extracting semantics from tweets and relating them with collected news articles [1]. Moraru *et al.* introduced a framework for enriching sensor measurements with semantic concepts to

generate new features [63]. In the Big Active Data project [43], notifications delivered to users can be enriched with other existing data in order to provide actionable notifications that are individualized per user; for example, emergency notifications could be enriched with shelter information to help affected users. Our work here is aimed at providing a scalable framework that users can employ to perform such data enrichment operations in the ingestion pipeline so that the enriched data can be used as soon as it is persisted.

User-defined functions have been a long standing feature of database systems [54, 85]. UDFs allow users to register their own functions with the database system for customized data processing and then invoke them in declarative queries. Hellerstein and Stonebraker designed a predicate migration algorithm for moving expensive functions in a query plan to minimize the total cost of a query [42]. Rheinländer *et al.* surveyed optimization techniques for optimizing complex dataflows with UDFs [73]. In this chapter, we use the UDF feature as a tool for users to use to express their data enrichment operations. Prepared queries are a mechanism that caches compiled plans to improve query performance. The predeployed jobs technique that we employed here for reducing the execution time of computing jobs was inspired by this technique.

The traditional ETL process defines a workflow including data collection, extraction, transformation, cleansing, and loading that is performed for moving data from an operational system into a data warehouse [24]. Data is extracted from an operational system, cleaned and transformed into a defined schema for analysis, and loaded into a periodically refreshed data warehouse for querying and data analysis. The refreshment process is often executed in an off-line mode with a relatively long period in order to minimize the burden on the operational systems [92]. Bruckner *et al.* proposed a near real-time architecture which minimizes the delay of new data being loaded into the data warehouse after being created in the operational system [17]. In this chapter, our focus was building an efficient and succinct framework aimed at ingesting and enriching data at the same time. Note that a user can achieve part of the

ETL functionality by constructing appropriate UDFs, we do not consider the new ingestion framework to be a tool for solving general ETL problems. (Similarly, using a complex ETL suite for data enrichment would be overkill.) Our data feeds feature is related to the continuous data loading technique commonly used in near-real-time data warehouses [46].

The emerging category of **hybrid transactional/analyitical processing** (HTAP) aims to serving fast transactional data for analytical requests from large-scale real-time analytics applications. Özcan *et al.* recently reviewed emerging HTAP solutions and categorized HTAP systems based on different design options [69]. Some use the same engine to support both OLTP and OLAP requests [38, 72]. Other systems choose to couple two separate OLTP and OLAP systems to handle the different workloads separately. Wildfire [13], for example, provides the Wildfire Engine for ingesting fast transactional data, and integrates it with Spark for supporting analytical requests. HTAP systems, and similar data analytics services [14], focus on enabling data analytics on recent data. On the other hand, our system looks generally at improving data enrichment performance during the ingestion process so that later analytical queries can be evaluated more efficiently. The techniques that we used in this chapter can be adapted to HTAP systems for accelerating their OLAP requests as well.

Streaming engines were introduced to address a need for stream data processing and real-time data analysis. They can handle streaming data sources and provide stream data processing on-the-fly. Many streaming engines also allow users to access reference data during processing. Kafka [51] uses "change data capture" in combination with its Connect API to access reference data in databases. Flink [19] supports registering external resources as Tables and offers a DataStream API to process the streaming data. Spark Streaming [100] uses Discretized Streams to discretize an incoming stream into Resilient Distributed Datasets and allow users to transform the data using normal Spark operations. Since streaming engines are designed for stream processing but not for complex data analysis queries, the

processed results are often stored in connected data warehouses [57]. In this chapter, we have focused on data enrichment use cases where the reference data may be frequently accessed and changed, and where the enriched data needs to be stored in a data warehouse for timely data analysis. We sought to minimize the effort from users so they can create a data ingestion pipeline easily, with declarative statements, and apply enrichment UDFs without limitations. To achieve these goals, we have chosen to build a new ingestion framework that supports the full power of SQL++ for data enrichment operations inside AsterixDB . The batch processing model that we chose is commonly used in streaming engines as well.

2.9 Summary

In this chapter, we have investigated how to enrich incoming data during the data ingestion process. We discussed the challenges in data ingestion, presented possible computing models for evaluating stateful UDFs for data enrichment, and discussed the problems that may occur in different scenarios. We believe that an ingestion pipeline that supports efficient data ingestion and enrichment should be able to capture reference data changes during the ingestion process, maintain intermediate states properly, and support different enrichment operations with a full query language.

To achieve these goals, we created a new ingestion framework with multiple optimization techniques. Its layered architecture allows the ingestion pipeline to better utilize the cluster resources. Repeatedly executing computing jobs in the framework allows incoming data to be enriched correctly, and predeployed jobs and partition holders improve the execution efficiency of computing jobs. We implemented the proposed framework in an open-source DBMS - Apache AsterixDB - and conducted a series of experiments to examine its performance with different workloads and various scales. The results showed that the new ingestion framework can indeed be scaled to support a variety of data enrichment workloads

involving reference data and/or stateful operations. The techniques and designs illustrated in this chapter could also be applied in other systems to accelerate their ability to support analytical requests based on enriched data.

Chapter 3

Subscribing to Big Data Continuously at Scale

3.1 Overview

Big Data, without being analyzed, is merely a sequence of zeros and ones sitting on storage devices. To effectively utilize Big Data, researchers have developed a plethora of tools [67, 80, 89, 99]. In many applications today, we want not only to understand Big Data, but also to deliver subsets of interest proactively to interested users. In short, users should not only be able to **analyze data** but also to **subscribe to data**. User subscription requests should not be limited to the incoming data's content but should also be able to consider its relationships to other data. Moreover, data to be sent should be allowed to include additional relevant and useful information. We refer to this as the *Big Active Data (BAD) challenge*. Due to the variety and volume of user requests, the data, and their relationships, analyzing, customizing, and delivering actionable data based on different user requests are not trivial tasks.

Traditionally, taking user requests and serving data continuously has been studied mostly in the context of Continuous Queries [25, 88]. Users there register their requests as persistent queries and are subsequently notified whenever new results become available. Although the continuous query concept overlaps significantly with the active data problem, Big Data poses new challenges for classic continuous query approaches due to their complexity and computational costs. Similarly, triggers from traditional databases offer users the capability to react to events in a database under certain conditions [96]. Users could try and take advantage of triggers to approach the active data challenge, but they soon become not applicable when the scale of the data, and thus the system, grows.

With the growth of streaming data and the need for real-time data analytics, Streaming Engines in recent years have been widely used in many active-data-related use cases [2, 51, 100]. Data is ingested and optionally processed in streaming engines on-the-fly and then be pushed to other systems for later analysis. Streaming engines can be used for creating data processing and data customizing pipelines, but due to the nature of data streams, only a limited set of processing operations are available. As a result, streaming engines would need to be coupled with other systems for meeting the complete BAD challenge at scale. This would introduce additional performance overhead and integration complexity for users.

Delivering data of interest to many users also resonates with the publish/subscribe communication paradigm from the distributed systems community [37]. In the pub/sub paradigm, subscribers register their interests in incoming data items and will subsequently be notified about data published by publishers. Despite some similarity to the BAD challenge, pub/sub systems only forward data from publishers to subscribers without offering the capability to process it. Also, each data item is treated in isolation, so users' interests are limited to the data item itself (its topic, type, or content), but not its relationship to other data. In addition, pub/sub systems have to be integrated with other Big Data systems (e.g., Data warehouses) for supporting analytical queries.

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

In order to deliver the latest updates to end-users without asking them to construct queries and to "pull" data from the system constantly, we propose an abstraction - parameterized ***data channels*** - to characterize user subscriptions. Users with expertise (e.g., application developers) can create data channels using declarative queries. Users with interest in data (e.g., end-users) can then subscribe to data channels with parameters and thus continuously receive new data. The BAD system runs data channels, manages their life-cycle, and offers them as active services. This data channel abstraction provides a declarative user model for activating Big Data.

Previously, we implemented the initial prototype BAD system - BAD-RQ - on top of Apache AsterixDB [4]. In BAD-RQ, we allow developers to create data channels using a declarative query language (SQL++) and enable users to subscribe to them by specifying their own parameters. Internally, channel queries are like parameterized prepared queries that are repetitively evaluated with subscription information and other relevant data. BAD-RQ computes them periodically on behalf of all users with all user-provided parameters and produces customized data for each subscribed user [44].

As BAD-RQ executes channel queries periodically, users may attempt to leverage them to approximate continuous query semantics - obtaining updates incrementally without retrieving the entire history or reporting redundant results [88]. For example, a continuous query *"send me **new** sensitive tweets"* can be loosely interpreted as a repetitive channel query *"every 10 seconds, send me the sensitive tweets from the past 10 seconds"*. Although users can approximate continuous query semantics with repetitive channels, BAD-RQ does not guarantee continuous query semantics, and data items could be missed or redundantly reported. To ensure continuous query semantics, we want a systematic way of supporting continuous queries in BAD. We need to make sure that users can receive incremental updates of data of interest with the guarantee of continuous query semantics, to support different computational operations and indexes for accelerating evaluation, and to enhance the data channel model to provide a straightforward user model regarding continuous queries.

In this chapter, we discuss BAD in-depth, present the BAD system, and then introduce BAD-CQ - a new BAD service that provides continuous query semantics. We show how BAD-CQ is designed and implemented, and we investigate its performance under different workloads at scale. This chapter is organized as follows: We review work related to BAD in Section 2. In Section 3, we dive into the detailed vision of Big Active Data, discuss the settings of the BAD problem, and describe the building blocks of a BAD system. In Section 3.4, we present a repetitive BAD use case to demonstrate the BAD-RQ service and illustrate the BAD user model. We introduce continuous BAD in Section 3.5, discussing the limitations of approximating continuous BAD and presenting the design and implementation of the new BAD-CQ service. To compare a possible alternative approach with the BAD system, we introduce a GOOD (**G**luing **O**odles **O**f **D**ata platforms) system that consists of gluing together multiple Big Data systems in Section 3.6. We show how to use the GOOD system for providing BAD services and illustrate the challenges that users would face in configuring, orchestrating, and managing such a glued system. We present a set of experimental results for the new BAD-CQ service and compare its performance with the glued system in Section 3.7.

3.2 Related Work

Continuous Queries are queries that are issued once and return results continuously as they become available. Tapestry [88] first introduced Continuous Queries over append-only databases, defined continuous query semantics, and created rewriting rules for transforming user-provided queries into incremental queries. Much subsequent research has focused on queries over streaming data. STREAM is a research prototype for processing continuous queries over data streams and stored data [8]. It provides a Continuous Query Language (CQL) for constructing continuous queries against streams and updatable relations [9]. TelegraphCQ offers an adaptive continuous query engine that adjusts the processing during runtime and applies shared processing where possible [52]. NiagaraCQ splits continuous queries into smaller queries and groups queries with the same expression signature together. It stores signature constants in a table and utilizes joins to evaluate grouped queries together to improve scalability, and it uses delta files for incremental evaluation on changed data to improve computational efficiency [25]. Most continuous query projects have been initial research prototypes, and very few of them have been scaled out to a distributed environment. This limits their applicability in Big Data use cases.

Streaming Engines allow low latency data processing and provide real-time analytics. Apache Storm is a distributed stream processing framework. It provides two primitives, “spouts” and “bolts”, to help users create topologies for processing data in real-time [90]. Spark Structured Streaming is a stream processing engine built on top of Apache Spark. It divides incoming data into micro-batches of Resilient Distributed Datasets (RDDs) for fault-tolerant stream processing, and it offers a declarative API for users to specify streaming computations [10, 100]. Apache Kafka started as a distributed messaging system that allows collecting and delivering a high volume of log data with low latency. It later introduced a Streams API that enables users to create stream-processing applications [48, 51]. Apache Flink [19] (which originated from Stratosphere [2]) unifies both streaming and batch

processing in one system and provides separate APIs (DataStream and DataSet) for creating programs running on a streaming dataflow engine [19]. Due to the nature of streaming data, streaming engines usually do not store data for the long-term. The incoming data is processed and then soon pushed to other systems for further processing or persistence.

Publish/subscribe Services allow subscribers to register their interests in events and to be subsequently, asynchronously notified about events from publishers. There are three types of pub/sub schemes: topic-based, content-based, and type-based [37]. In topic-based pub/sub, publication messages are associated with topics, and subscribers register their interests to receive messages about topics of interest. Many systems in this domain focus on providing scalable and robust pub/sub services, including Scribe [22], SpiderCast [26], Magnet [39], and Poldercast [78]. Content-based pub/sub improves the expressiveness of pub/sub services by allowing subscriptions based on publications' content. Many research works in this area focus on improving the scalability and efficiency of matching publications to users' subscriptions, including XFilter [7], Siena [21], YFilter [32], and BlueDove [53]. Type-based pub/sub groups publications based on their structure. It aims at integrating pub/sub services with (object-oriented) programming languages to improve performance [36]. While all these pub/sub services enable publishing data to a large number of subscribers, the expressiveness of subscriptions is limited as each publication is treated in isolation, and users often have to integrate a pub/sub service with other systems for data processing.

3.3 Big Active Data

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

3.3.1 A BAD World

In a BAD world, data could come from various systems and services constantly and rapidly. Many users would like to acquire and share the data and use it for different purposes. Some users may want to analyze the collected incoming data for retrospective analysis. They may issue analytical queries like:

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

Figure 3.1: A sample analytical query on collected tweets

Other users may want to continuously receive updates regarding the data of interest to them. Users' interests may cover different aspects of the data. For example:

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

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

1. **Analysts** issue queries to analyze collected incoming data and/or other relevant data.
2. **Subscribers** make subscriptions and receive updates continuously using BAD applications.

3. **Developers** create BAD applications and provide BAD services to analysts and subscribers.

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

3.3.2 The BAD Building Blocks

In order to provide the features described in Section 3.3.1, a BAD system needs to have the following building blocks:

- **Persistent Storage**: In order to support retrospective analysis, data enrichment with relevant information, and customized data subscription, the BAD system should provide persistent storage to store collected incoming data, relevant data, and subscription information. It should be possible to add data to the BAD system through ingestion facilities, loading utilities, or applications' CRUD operations. Since data is persisted, developers should be able to utilize auxiliary data structures (like indexes) for accelerating data access.
- **Ingestion Facility**: Data of interest, for either subscribers or analysts, may come into the BAD system rapidly. In order to capture such data, the BAD system should provide an ingestion facility to help continuously ingest data from various external data sources reliably and efficiently. BAD users should be able to easily create an ingestion pipeline in the BAD system without having to write low-level programs.
- **Analytical Engine**: Data analytics enables analysts to reveal useful information from data. To help analysts understand the incoming data and its relationship with other

relevant information, the BAD system should provide an analytical engine with support for declarative queries.

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

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

We depict the BAD system and the BAD users it serves in Figure 3.2. To the best of our knowledge, there is no existing Big Data platform that provides all the functionality needed from a BAD system. Some platforms can fulfill certain building blocks in the BAD system, but one would have to hand-wire multiple systems together to get all desired BAD features. A well-designed, integrated, and efficient BAD system with support for a declarative language can significantly reduce the effort required to create BAD services.

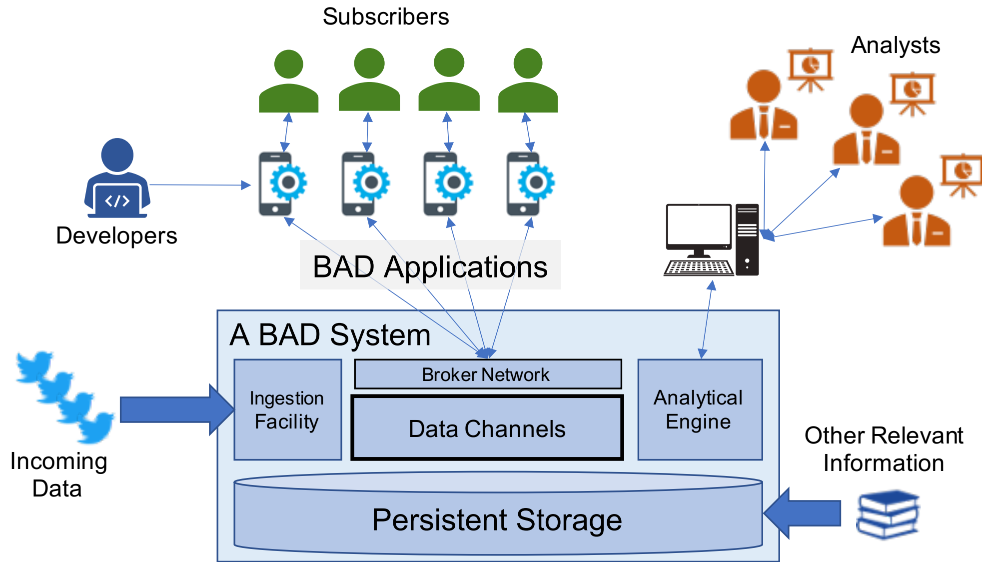


Figure 3.2: A BAD system for a BAD world

3.4 Repetitive BAD: BAD-RQ

A straightforward way of creating a BAD system is to "activate" an existing Big Data system by adding the missing features needed for BAD services and making sure it scales to millions of subscribers. We created the initial prototype BAD system - BAD-RQ - on top of Apache AsterixDB, an open-source Big Data Management System that provides distributed data management for large-scale, semi-structured data. In this section, we focus on the user model of BAD-RQ and illustrate how developers utilize it to create BAD services. Interested readers can refer to [20, 43, 44, 45, 91] for more information about the whole BAD project.

3.4.1 A BAD Repetitive Use Case

To illustrate BAD-RQ, we use a sample scenario in which we want to provide BAD services to police officers around tweets. Users of these services include investigative officers as **analysts** who want to study tweets about certain events, and in-field officers as **subscribers** who patrol around the city and want to receive live tweets meeting certain requirements. Tweets

come into BAD-RQ from an external system continuously, and each contains a hateful flag provided by the datasource indicating whether this tweet is hateful and may relate to a potential crime. Location updates of patrolling in-field officers are also sent to BAD-RQ constantly to show their latest location. We describe the implementation of BAD building blocks in BAD-RQ and demonstrate how **developers** can utilize them for creating BAD services.

Persistent Storage

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

To store data in the BAD dataverse, we (as developers) need to create a *datatype*, which describes the stored data, and a *dataset*, which is a collection of records of a datatype. We define both the Tweet and OfficerLocation data types as "open", which makes the stored data extensible. The "hateful_flag" attribute, indicating whether a tweet is hateful, is not specified in the data type and thus it is an open (optional) field. When "hateful_flag" is not provided but needed for a BAD application, a developer could use a enrichment user-defined function (UDF) to enrich tweets during data ingestion [93]. We create a dataset *Tweets* for storing incoming tweets, a dataset *OfficerLocations* for storing location updates, and two R-Tree indexes on the location attribute of each dataset for more efficient data access. The DDL statements for creating both datasets are shown in Figures 3.3 and 3.4 respectively.

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

Figure 3.3: Datatype and dataset definition for Tweets

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

Figure 3.4: Datatype and dataset definition for officer location updates

Ingestion Facility

Since tweets and location updates may come at a very rapid rate, the BAD system needs to intake such "fast" incoming data efficiently. AsterixDB provides data feeds for data ingestion from various data sources with different data formats [41]. We create a socket data feed to intake JSON formatted tweets using the statements shown in Figure 3.5. Similarly, we create a data feed for intaking location updates sent by in-field officers in Figure 3.6. In this use case, we send in-field officers nearby hateful tweets based only on their current location, so we create an UPSERT (i.e., insert if new, else replace) data feed by setting *"insert-feed"* to false. In cases where officers' entire movement history is needed, one can also create an INSERT data feed like the one used for tweets.

Analytical Engine

BAD-RQ supports data analytics using the query engine in AsterixDB. It provides SQL++ [23, 68] (a SQL-inspired query language for semi-structured data) for users to construct analyt-

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

Figure 3.5: A data feed for ingesting tweets

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

Figure 3.6: A data feed for ingesting location updates

ical queries. SQL++ supports standard SQL query operations (SELECT, JOIN, GROUP BY, ORDER BY, etc.), spatial-temporal queries, operations designed for semi-structured data, etc. One can use the SQL++ query shown in Figure 3.7 to answer the analytical query from Figure 3.1. For a query executed multiple times with different constant expressions, analysts can also define it as a SQL++ UDF and invoke it with parameters instead of reconstructing the same query every time. As an example, the analytical query in Figure 3.7 can be encapsulated in the SQL++ UDF shown in Figure 3.8.

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

Figure 3.7: An SQL++ query looking for the 10 most hateful cities in each month in a given time frame

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

Figure 3.8: A UDF based on an analytical query

Data Channels

Since queries can be encapsulated as a UDF with parameters, and subscriptions with a similar structure can also be interpreted as a parameterized query, we can use a SQL++ UDF to group these subscriptions together and "activate" it as a data channel. Developers can create data channels based on SQL++ UDFs and offer them as services, and subscribers can subscribe to them with parameters to receive data of interest subsequently. As an example, if in-field officers want to know the number of hateful tweets near their current location in the past hour, we can first create the UDF in Figure 3.9, which can be invoked using an officer's ID and returns the number of recent hateful tweets nearby. We "activate" this UDF using the statement in Figure 3.10 by creating a data channel using this UDF. This channel has a configurable period "10 minutes" indicating that it computes every 10 minutes. In-field officers who subscribed to this channel then will receive the number of nearby hateful tweets in the past hour every 10 minutes. We will further discuss how a channel evaluation produces customized data for each subscriber in Section 3.4.2.

Brokers and Subscriptions

The BAD system includes a broker sub-system for managing the communication with a large number of subscribers. A broker could be a single server that only forwards customized data to subscribers or a broker network that provides load balancing, subscription migration,

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

Figure 3.9: An UDF for counting hateful tweets near certain in-field officer given his/her officer ID

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

Figure 3.10: Creating a data channel based on a UDF with a parameter

and different caching strategies. Interested readers can refer to [66, 91] for more details. A developer can choose a broker suited for the use case and register it as an HTTP endpoint in the BAD system as in Figure 3.11. A subscriber can then subscribe to a channel in the BAD system on this broker using the statement in Figure 3.12. A given channel execution can produce customized data for subscribers subscribed on different brokers, and the customized data is sent to the corresponding brokers based on which brokers the subscriptions are subscribed on. A broker receives the customized data from channel executions and then disseminates it to its subscribers.

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

Figure 3.11: Registering a broker to BAD

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

Figure 3.12: Subscribing to a channel with parameters on a broker

3.4.2 Data Channel Evaluation

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

Modeling Brokers and Subscriptions

As we mentioned in Section 3.3.2, subscribers subscribe to a data channel with parameters, and there could be millions of subscribers for a data channel. Given the large volume of subscriptions, separately evaluating a channel query (the underlying UDF of a channel) for each subscriber would be too computationally expensive. Inspired by [25], BAD-RQ stores subscriptions as data and evaluates the channel query using the analytical query engine. Benefiting from the query optimization, indexes, and distributed evaluation in AsterixDB, BAD-RQ can compute a channel query with a lot of subscriptions efficiently, and the channel evaluation process can also take advantage of the shared computation among subscriptions in order to serve more subscribers.

BAD-RQ uses the data types defined in Figure 3.13 to store the broker and subscription information internally. Broker information is decoupled from subscriptions, so a broker record can be modified without having to update all related subscriptions. The subscription data type is defined as *open*, and the parameters of a subscription are stored as open attributes and named as param0, param1, etc. This allows a data channel to support an arbitrary number of parameters with arbitrary data types. The broker dataset is a permanent part of the BAD-RQ metadata. The subscription dataset is tied to the life-cycle of a data channel. When a developer creates a data channel (e.g., RecentNearbyHatefulTweetCountChannel), a corresponding subscription dataset (RecentNearbyHatefulTweetCountChannelSubscriptions) is

also created, and this will be removed when the channel is dropped. Whenever a subscriber subscribes to the channel, a new subscription record is inserted into the subscription dataset.

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

Figure 3.13: Data type definitions for brokers and subscriptions (internal to BAD)

An example of Channel Evaluation

In order to illustrate how BAD-RQ periodically computes a channel and produces customized data for each subscriber using broker and subscription information, we show a small data sample in Figure 3.14 for the channel defined in Section 3.4.1, which returns the number of hateful tweets near a particular in-field officer. For illustrative simplicity, we assume all three tweets are posted within one hour and are hateful, and attributes not used for evaluation are not shown in the figure.

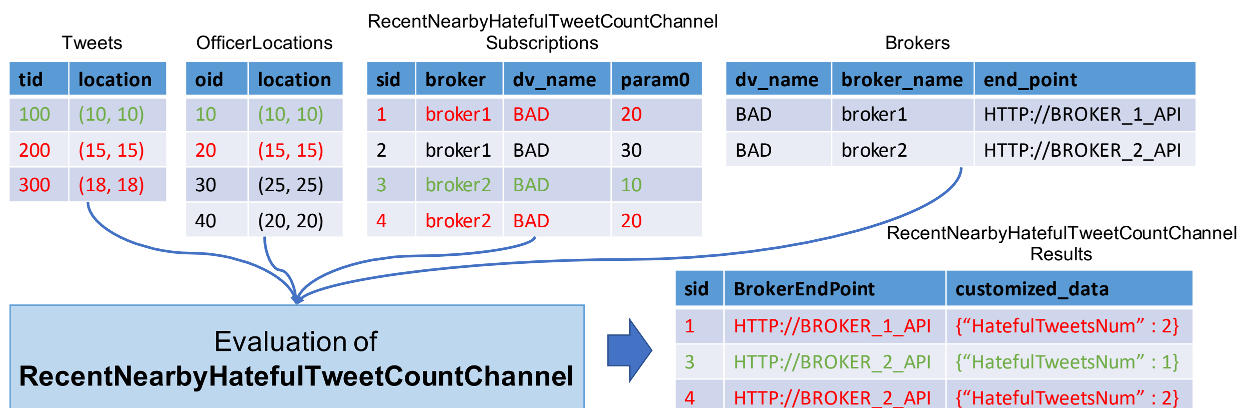


Figure 3.14: A data sample for evaluating a data channel

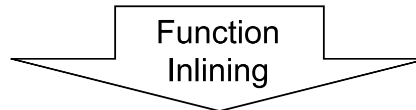
The channel evaluation combines information from four datasets, including *OfficerLocations*, *Tweets*, *RecentNearbyHatefulTweetCountChannelSubscriptions*, and *Brokers*, and it produces the customized data shown in the *RecentNearbyHatefulTweetCountChannelResults* dataset. Related tuples are colored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

Channel Evaluation Internals

Evaluating a channel is equivalent to evaluating a query where we apply the underlying UDF to parameters from subscriptions to produce customized data. For example, evaluating the channel defined back in Figure 3.10 is equivalent to evaluating the query shown in Figure 3.15. In this query, we apply the UDF in Figure 3.9 on parameters from subscriptions and nest the return value of the UDF into a "customized_data" field. The UDF can be inlined into the query, as shown, and be compiled and optimized together with it. The broker endpoint and subscription ID are also attached to each customized data record. The broker endpoint is used for the channel to send the result to a corresponding broker API, and the subscription ID is used by brokers to identify which subscriber the customized data should be delivered to.

Since the query used for evaluating a channel is computed on the analytical engine of AsterixDB, it can be optimized by the query optimizer and be accelerated by utilizing efficient algorithms and indexes. Under the hood, the query in Figure 3.15 compiles into a query plan as shown in Figure 3.16. BAD can use use an R-Tree index to accelerate the spatial

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




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

Figure 3.15: An illustrative query for computing a channel

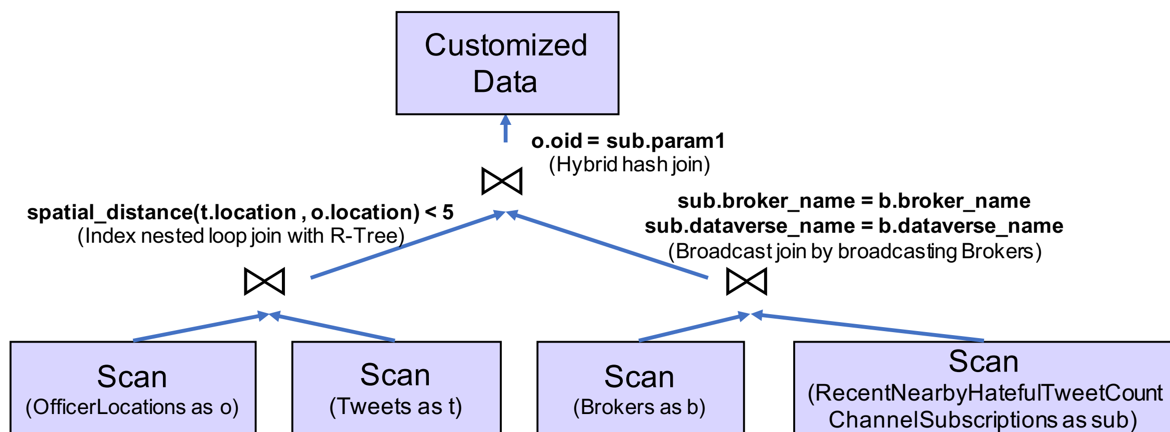


Figure 3.16: A query plan for channel evaluation

join between *Tweets* and *OfficerLocations*. Also, since the number of brokers is small compared with subscriptions, it can broadcast the Brokers to avoid unnecessary shuffling of the Subscriptions dataset. It can use a hybrid hash join to join the two intermediate results in parallel on all nodes in the cluster.

Customized Data Delivery

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

3.5 Continuous BAD: BAD-CQ

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

In some use cases, subscribers may want the latest information delivered **incrementally**.

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

3.5.1 Approximately Continuous Queries

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

BAD Timestamps

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

In some cases (like tweets), incoming data comes with a "timestamp" attribute which indi-

cates when was a data item created (a.k.a., valid time [81] or event time [19]). This attribute could potentially be used for differentiating new tweets from old ones. However, this would introduce additional complexity in handling out-of-order arrivals. Besides, when such an attribute is not provided in the incoming data, we still need to find another solution[1]. BAD-RQ allows developers to attach timestamps to incoming tuples during data ingestion by attaching a UDF to the ingestion pipeline. For this use case, we can create the UDF shown in Figure 3.17 and attach it to the tweet data feed defined in Figure 3.5. This UDF adds an "ingested_timestamp" attribute to each incoming tweet, which marks the current date time when the tweet first enters the pipeline (a.k.a, ingestion time [19]). We can utilize this timestamp to determine whether or not a tweet is **new**.

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

Figure 3.17: A UDF for adding ingestion time

A Repetitive Approximation

By assigning timestamps to incoming tweets during ingestion, we can use the ingestion timestamp to infer the arrival order of tweets and differentiate "new" tweets from "old". We can construct a repetitive data channel with a designated channel period, as shown in Figure 3.18. In this channel, we look for hateful tweets ingested in the past 10 seconds from the time when the channel executes. These tweets are new and thus haven't been examined yet. We join them with officers' current locations and look for nearby new hateful tweets for each subscribed in-field officer. The channel is defined to execute every 10 seconds, so subscribers can continuously receive new nearby hateful tweets. This allows us to approx-

[1]Streaming Engines (such as Spark Structured Streaming) that compute with event time offer watermarking to handle late arrivals. BAD-RQ with BAD timestamps (and BAD_CQ later introduced in Section 3.5.2) can provide similar functionality with proper channel queries. Here we focus on the general use cases without assuming the existence of event time.

imate continuous (incremental) semantics with a repetitively executed channel query that runs every 10 seconds and looks back 10 seconds.

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

Figure 3.18: A repetitive data channel looking for new nearby hateful tweets

Challenges in Approximation

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

- *Scheduling Delay:* We approximate the continuous query semantics by examining data ingested in the past execution period (e.g., 10 seconds) from the current channel execution time. To perfectly approximate a continuous query, we rely on BAD-RQ to schedule the channel execution *on time* to make sure that all incoming data is examined. However, this is impractical in practice, especially in a distributed environment. If a scheduling delay happens, some data can be missed by the channel, as shown in Figure 3.19. This channel executes every 10 seconds and examines data ingested from the past 10 seconds. If the actual channel execution 1 is delayed from T = 20 to 20.5, the data ingested from T = 10 to 10.5 will not be examined and thus missed.

- *Early Timestamping:* The approximation of BAD-RQ uses the ingestion timestamp for determining whether ingested data should be examined in a channel execution.

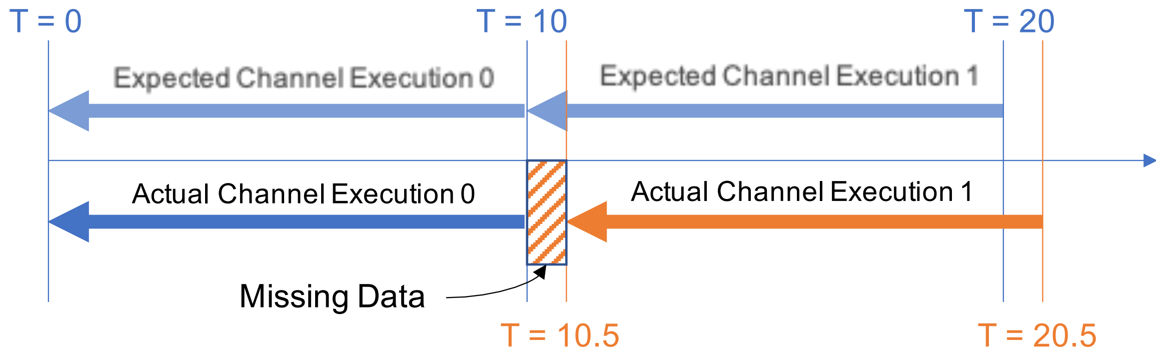


Figure 3.19: Missing data due to scheduling delays

However, since the ingested (timestamped) data does not become visible to channel execution instantaneously due to delays in data transmission, data enrichment (if any), secondary index(es) updating (if any), primary index updating, and waiting for the storage transaction to complete, there is a chance that a running channel execution could miss the data just ingested, even if the channel execution is scheduled on time. This is illustrated in Figure 3.20, where channel execution 1 starts at $T = 10$ and a tuple t100 is ingested at $T = 10 - \sigma$ and later persisted and becomes visible to queries at $T = 10 + \delta$ due to the delay [2]. Channel execution 1 does not examine t100 because the tuple is not in storage yet, and channel execution 2 will not examine t100 either, because the tuple has an ingested_timetsamp that is smaller than 10 (i.e., too old). Thus, tuple t100 is missed.

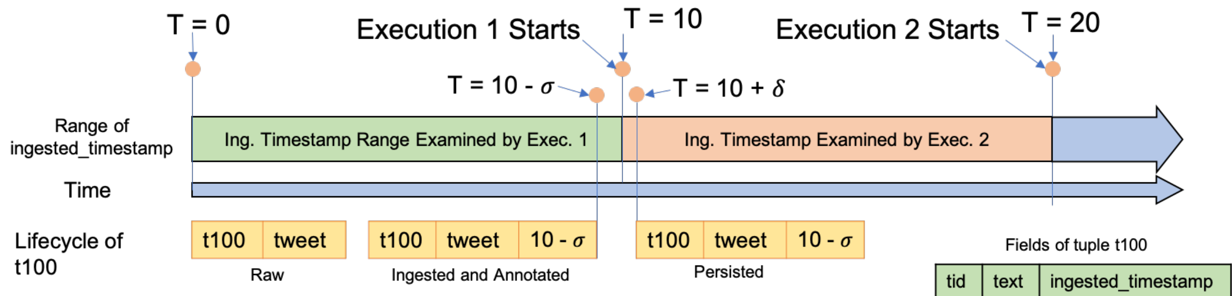


Figure 3.20: Missing tuple due to early timestamping

[2]In practice, this time gap is very small. We emphasize the delay in Figure 3.20 for illustration purposes.

- *Risk of User Data "Corruption":* We have attached an timestamp ("ingested_timestamp") attribute to explicitly mark the ingestion time of incoming tweets. This attribute then exists as part of the user data, and other users of the BAD system can access it. This raises the potential risk that other users may accidentally modify this attribute and cause data channels to fail. Additionally, this auxiliary information may cause confusion for non-channel users such as data analysts.

- *Complex Approximation Query:* In order to approximate continuous query semantics, we have chosen the same time period in the temporal predicate and the channel execution period, as shown in Figure 3.18. Such a correspondence needs to be managed manually and carefully by developers. When channel queries become more complex and involve multiple incoming data sources, constructing a proper approximation query can be challenging. One would have to add proper temporal predicates for each of the data sources, and when there are joins between these data sources, which portion of the collected data from one data source should be joined the other one needs to be carefully specified with temporal predicates. These temporal predicates would increase the query complexity and make such queries very difficult to write.

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

3.5.2 BAD-CQ

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

Active Datasets

As we have discussed in Section 3.5.1, BAD persists all data to support retrospective analysis. To help data channels differentiate new data from old, we need to timestamp incoming data and use timestamps for proper continuous channel evaluation. To avoid the previously mentioned drawbacks of adding an ingestion timestamp to user data, we introduce a new type of datasets - *Active Datasets* - in BAD-CQ. Unlike regular datasets in AsterixDB, a record (active record) stored in an active dataset contains not only user data but also a "hidden" active attribute: "_active_timestamp". This helps BAD-CQ to evaluate continuous channel queries. This attribute is stored alongside users' data but separated from the regular record content. It is "invisible" to users and can only be accessed using active functions (to be discussed soon). The storage layout of an active record is shown in Figure 3.21.

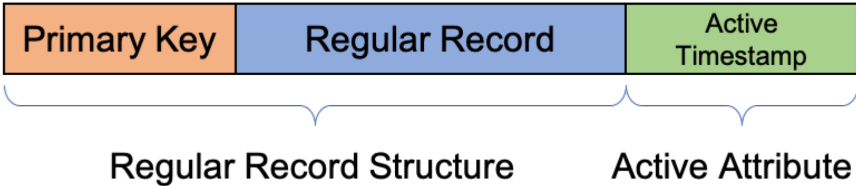


Figure 3.21: Storage format of an active record

As the BAD system runs in a distributed environment, which clock to use to assign active timestamps needs careful consideration. One might first consider using a single clock to assign all active timestamps. This would be convenient because then all active timestamps would be directly comparable, and we would only need to figure out one active timestamp range to identify all the new data. However, having a master clock would require either routing all data to a single node, which would create a bottleneck in the system, or synchronizing clocks on multiple nodes, which can be very challenging in a distributed environment. In BAD-CQ, we instead use the local clock on each node to assign active timestamps to the active records stored on it for scalability. Active timestamps are assigned in the storage engine, after the locking phase. This makes sure that incoming records will become visible

to running queries as soon as they are timestamped. Although the **new data** on each node may now have a different active timestamp range, we can introduce an active timestamp management mechanism with additional query optimization rules to make sure that channel queries are evaluated correctly. We will further discuss this in Section 3.5.3.

Considering that active timestamps often need to be compared in channel queries, we can optimize these comparisons to improve channel performance. One might consider creating a secondary index on active timestamps, but this would take additional disk space and incur additional access overhead when the selectivity is high [55]. As the active timestamps of an active dataset grow monotonically, we can instead utilize the filter feature in the AsterixDB storage engine to avoid accessing irrelevant data [6]. The BAD storage engine uses Log-Structured Merge (LSM) Trees as its storage structure [5]; they perform batch updates into components to avoid the cost of random writes and then read them sequentially for data access. One can designate a filter attribute when creating a dataset, and every LSM component of this dataset is then decorated with the maximum and minimum attribute values of its stored records. When a query containing a filter attribute comparison comes, it can quickly skip irrelevant components by examining their maximum and minimum filter values. For active datasets, we use the active timestamp as the filter attribute to accelerate channel queries, as shown in Figure 3.22. The *active_timestamp(t)* function reveals the active timestamp of the tuple t stored in the active dataset *Tweets*, as will further be discussed in Section 3.5.3.

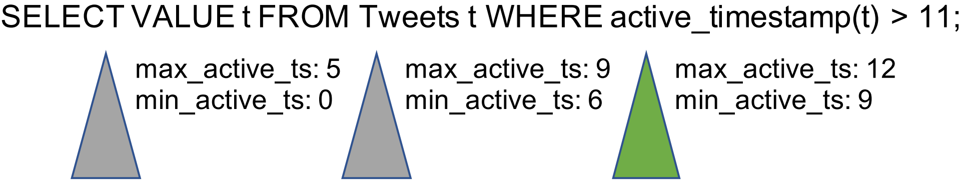


Figure 3.22: Access active datasets with filters

The syntax for creating active datasets is straightforward. An active dataset can be created with a regular data type, and the active attribute and filter are automatically configured

behind-the-scene. One can create two active datasets *Tweets* and *OfficerLocations* using the statements in Figure 3.23. Active datasets can also be accessed in regular queries just like non-active datasets. There is an extra overhead when reading active datasets due to the additional space for storing active timestamps. We will see from later experiments that this overhead is relatively small. When not used in query evaluation, active timestamps are projected out from the active records as early as possible to avoid potential transmission overhead.

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

Figure 3.23: Datatype and dataset definition for officer location updates

Active Timestamp Management

With active datasets, we now need to "teach" channels to utilize the active timestamps to recognize **new** data and to guarantee continuous query semantics. The basic idea is straightforward: keep track of the channel execution times and compare them with active timestamps to find the new data. As mentioned in Section 3.5.2, each node uses the local time to assign active timestamps, so we also need to use local time for tracking channel execution times and make sure they are properly compared with active timestamps. We create a local active timestamp manager on each node to keep track of the *previous channel execution time* and the *current channel execution time* under the local clock. When a channel executes on a node, these two timestamps are used to determine which portion of the stored data should be considered for this execution.

To demonstrate how multiple local active timestamp managers can work to offer continuous semantics, we consider the channel defined in Section 3.5.1 that looks for **new** nearby tweets for in-field officers. We show an illustrative channel execution example in Figure 3.24. In this example, we use the cluster controller (CC) time as the (conceptual) cluster time. Since not

all nodes are synchronized on time, current timestamps on different nodes can be different. In this case, when CC starts the first channel execution at time T_0, Node A marks the channel start time under its local time as T_0^A, which is *"logically before"* T_0, and Node B marks the channel start time under its local time as T_0^B, which is *"logically after"* T_0. When the CC invokes the first channel execution at T_1, every node examines the tweets ingested and persisted from the previous channel execution time to the current channel execution time. From Node A's perspective, all tweets ingested from T_0^A to T_1^A are examined. From Node B's perspective, tweets ingested from T_0^B to T_1^B are examined. Although T_1, T_1^A, and T_1^B are different, from the CC's (and subscribers') perspective, only nearby hateful tweets from T_0 to T_1 are reported to subscribers. This guarantees the continuous semantics for this channel. The channel's previous channel execution and current channel execution time are each progressed with each channel execution. They are updated instantly when a channel execution job first accesses an active dataset used for the channel. This makes sure that all incoming tweets that were persisted before the current channel execution can all be safely examined in the current channel execution.

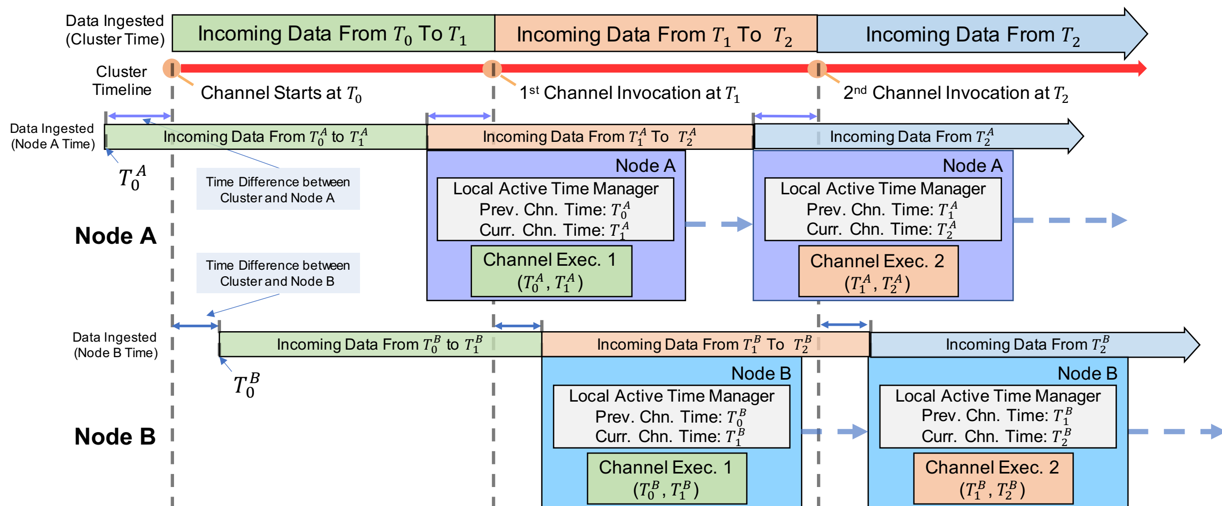


Figure 3.24: An illustration of active timestamp management

The active timestamp manager enables BAD-CQ to provide continuous query semantics in a distributed environment without time synchronization. The monotonically increased active

timestamps on each node in fact act like sequence numbers. The local active time manager marks the range of sequence numbers for each channel execution (as its previous and current channel execution time) and allows it to find the new data.

3.5.3 BAD-CQ Syntax and Optimization

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

- ***active_timestamp(t)*** reveals the active timestamp of the tuple t.
- ***previous_channel_time(t)*** returns the previous channel execution time on the node where the tuple t is persisted, as defined in Section 3.5.2. Note that every node has its own (local) channel time for a channel, and dataset *Tweets*'s tuples could be persisted on multiple nodes, so this function is evaluated locally on each node at run time, and tuples from Tweets used in the channel could have different previous channel times.
- ***current_channel_time(t)*** returns the current channel execution time of the tuple t, as defined in Section 3.5.2. Similar to *previous_channel_time*, *current_channel_time* is also computed locally at run time, and tuples from *Tweets* could have different current channel time.
- ***is_new(t)*** returns a boolean value indicating whether tuple t is new to the current channel execution. The return value of *is_new(t)* is equivalent to the following ex-

pression: *previous_channel_time(t)* < *active_timestamp(t) AND active_timestamp(t)* < *current_channel_time(t)*.

With active functions, a developer can conveniently construct continuous channels with continuous query semantics. Here we show an example for the use case described in Section 3.5.1, where subscribers would like to receive new tweets near in-field officers. We use a different user model in BAD-CQ. Data channel definition in BAD-CQ is not based on UDFs, since active functions are not meaningful outside. Executing *previous_channel_time* and *current_channel_time* functions in regular queries return 0 and current cluster time respectively. Using BAD-CQ's active functions, a developer can create a continuous channel for the new nearby hateful tweets using the statement shown in Figure 3.25.

In order to assist channel evaluation with active functions and to improve channel performance, we introduce two new query optimization rules into BAD-CQ. First, when compiling a continuous channel query, we push the *current_channel_time* function into the leaf node of the query plan - the data scan operator of an active dataset - as the filter's maximum value. This is because when an active dataset is accessed in a channel execution, only data before the current channel execution time is relevant. We use this to quickly skip data coming after the current execution starts. Second, we push the *previous_channel_time* function down towards the leaf of the query as much as possible, and we use it as the filter's minimum value for active datasets when applicable. Whether this function can be pushed into the data scan operator depends on the specific channel query. For the channel query defined in Figure 3.25, we can indeed push *previous_channel_time(t)* into the *Tweets* scan operator and use it as the minimum filter, as shown in Figure 3.26. [3]

When the *previous_channel_time* function cannot be pushed all the way down into a data scan operator, we need to attach its node-dependent value (i.e., the previous channel execu-

[3] In this channel, we only need officers' latest location, so there is no lower bound on active timestamps of *OfficerLocations*. We will introduce another example in Section 3.5.4 which requires recent location updates and utilizes the minimum filter on *OfficerLocations*.

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

Figure 3.25: A continuous channel for new nearby hateful tweets

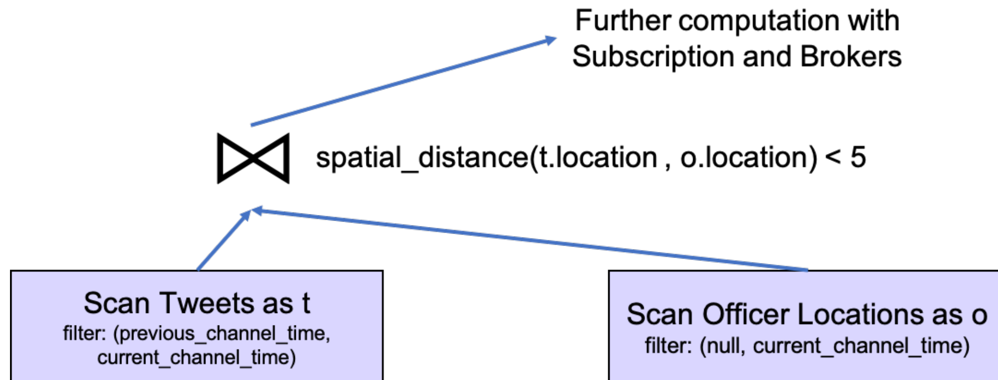


Figure 3.26: Query plan for new nearby hateful tweet channel

tion time on a node) to the active records read from this node. In this case, the comparison between active timestamps and the *previous_channel_time* function is rewritten into a comparison between active timestamps and this attached previous channel execution time value. This makes sure that even if active records are shuffled around in the cluster, the comparison between their active timestamps and previous channel time will be evaluated correctly. To explain how the second rule works in this scenario, we introduce another continuous use case, where *"in-field officers (as subscribers) would like to receive nearby hateful tweets he/she has not seen before"*. In this case, we not only need to consider a new tweet posted near an in-field officer, but also tweets that were not nearby but that become nearby due to the movement of in-field officers. We can create a continuous channel for this use case as shown in Figure 3.27.

In this continuous channel query, the active functions *is_new(o)* and *is_new(t)* are expanded to the corresponding query predicates based on active timestamps, previous channel execution time, and current channel execution time as shown in Figure 3.28. Following the

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

Figure 3.27: A continuous channel for unseen nearby hateful tweets

first optimization rule, the current time timestamp of both *Tweets* and *OfficerLocations* are pushed into the corresponding data scan operators. However, the previous channel execution time cannot be pushed thoroughly, because the disjunctive predicate "*active_timestamp(t) > previous_channel_time(t) OR active_timestamp(o) > previous_channel_time(o)*" also needs data from before the previous channel execution time from both datasets. Following the second optimization rule, this continuous channel query can be compiled into the plan shown in Figure 3.29. The disjunctive predicate is evaluated in the join operation that is computed across all nodes, and data is shuffled around in this process [4]. Notice now that since the previous channel execution time is attached to active records, we can compare the active timestamp with the channel execution time under the same local clock, even if records are shipped to another node.

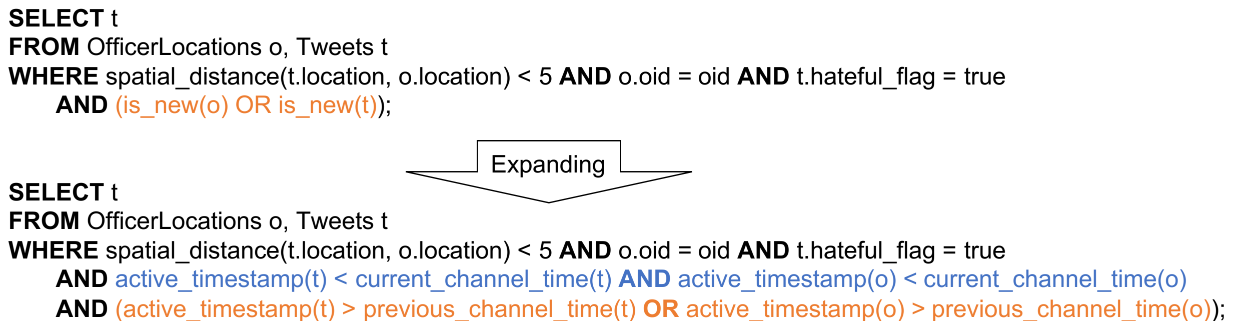


Figure 3.28: Expanding a continuous channel query with active functions

Different from the implicit query rewriting in Tapestry [88] and the delta files in NiagaraCQ [25], BAD-CQ allows developers to construct queries using active functions that are best suited for their use cases, and it takes advantage of the storage engine for accel-

[4]Depending on the workload, the execution plan for the channel query can choose either to broadcast *Tweets* or *OfficerLocations*.

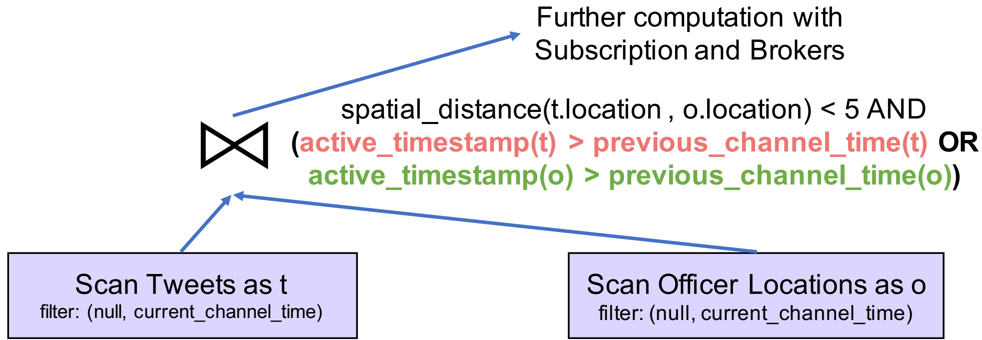


Figure 3.29: Query plan for unseen nearby hateful tweet channel

erating channel queries without having to introduce additional data structures. Developers can write a query using the *is_new* function and let the query compiler rewrite it into an incremental query, or they can use the *active_timestamp* function to expose the active timestamps and directly compare them with channel times or other times. The BAD-CQ user model uses datasets to hold the collected incoming data and other existing data. This provides developers with a unified query model and lets them to reuse all dataset processing operations when defining channels. The principles underlying the BAD-CQ approach are general - i.e., other database systems supporting declarative queries could also be adapted to provide continuous query semantics like BAD-CQ.

3.5.4 BAD-CQ Semantics

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

New Nearby Hateful Tweets

We first look at the example from Section 3.5.1, where in-field officers would like to receive new nearby hateful tweets. The channel is defined in Figure 3.25. We use the *is_new* function to look for **new** tweets that have not been sent to subscribers, and we use the officers' latest locations to look for nearby tweets.

In Figure 3.30, we show a channel execution example with several sample data records. In order to focus on the channel execution process, irrelevant attributes of tweets and officer location updates are not shown in the figure. The channel starts at time 0, and in-field officers u10 and u20 have initial location at time 0 of (0, 0) and (0, 10), respectively. At 9s, the first tweet t100 arrives and its location is (0, 3). When the channel first executes at 10s, only tweet t100 is near in-field officer u10, so the channel produces one notification for u10. After that, u20 updates his/her location to (0, 7) at 13s. When the channel executes at 20s, as there is no new tweet after the previous channel execution, no notification is produced. Later, u10 updates his/her location to (0, 3) at 22s, and a new tweet t200 located at (0, 4) arrives at 28s. When the channel executes at 30s, both u10 and u20 have t200 nearby, so the channel produces two notifications for each of the corresponding officers.

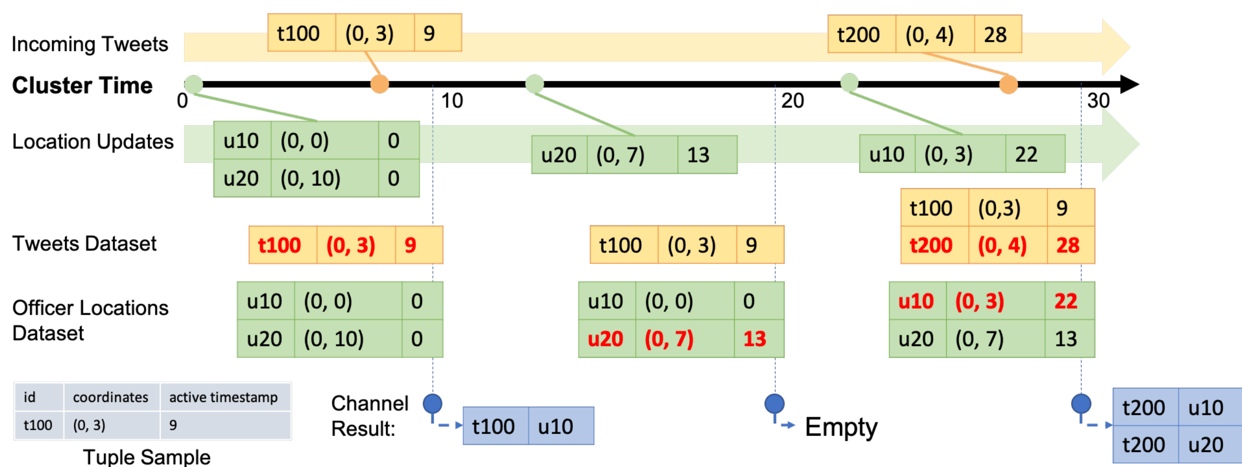


Figure 3.30: Officer u10 subscribing to CQNewNearbyHatefulTweets(u10) and officer u20 subscribing to CQNewNearbyHatefulTweets(u20)

Unseen Nearby Hateful Tweets

In the previous use case, in-field officers receive a hateful tweet only if the tweet is temporally new. In another use case, officers may also be interested in older nearby hateful tweets that they have not seen before (which could contain useful information). The channel definition for this use case is shown in Figure 3.27.

We use the same data sample in Section 3.5.4 to explain how this channel works. As shown in Figure 3.31, the channel acts the same way as the previous one and produces one notification for u10 in the first channel execution. In the second channel execution, the location update of u20 from (0, 10) to (0, 7) makes t100 become nearby to u20, so the channel produces one notification for u20 at 20s to notify this officer about this previously unseen tweet. The third channel execution starts at 30s and produces two notifications for u10 and u20, respectively, as both in-field officers have not seen this new tweet.

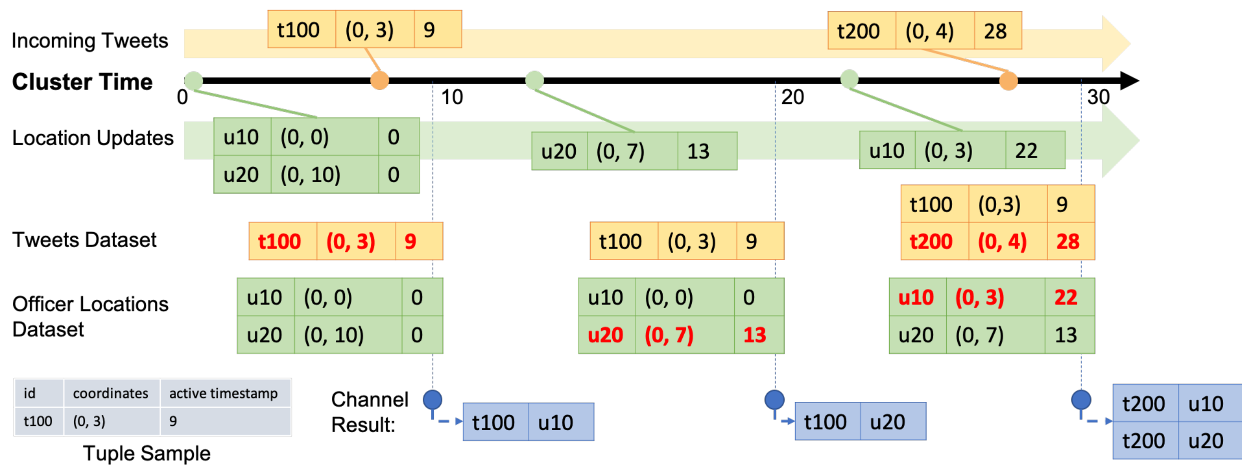


Figure 3.31: Officer u10 subscribing to UnseenNearbyHatefulTweets(u10) and officer u20 subscribing to UnseenNearbyHatefulTweets(u20)

New Nearby Hateful Tweets for Active Officers

In the previous use cases, even if an officer is not updating his/her location constantly (e.g., in order to reduce power/data plan consumption), the channel can still be producing notifications for them based on their last known location. When the officer reconnects, the broker sub-system can pull notifications that were produced "offline" from the BAD storage engine and send them out. If we want to produce notifications only to "active" in-field officers (who are their updating the locations to the system regularly), one can create the continuous channel defined in Figure 3.32. Different from the channel defined in Figure 3.25, we now only look for new hateful tweets for officers who have recently updated their locations instead of all officers. Those who are not updating their locations "actively" will not receive nearby hateful tweets while they are inactive.

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

Figure 3.32: A continuous channel for new nearby hateful tweets

Following our data sample used in previous use cases, the execution process of this channel is shown in Figure 3.33. Similarly, the first channel execution produces one notification based on u10 about t100. In the second channel execution, no notification is generated since there is no new incoming tweet. In the third channel execution, we produce one notification about the new tweet t200 for u10 who has recently updated his/her location. Although t200 is also near u20, we do not produce a notification for him/her since u20 is not "active". As we can see from these sample use cases, active functions offer the flexibility and expressiveness of working with both the *new* and *historical* data. Developers can use active functions to conveniently construct a wide range of suitable queries for their BAD applications.

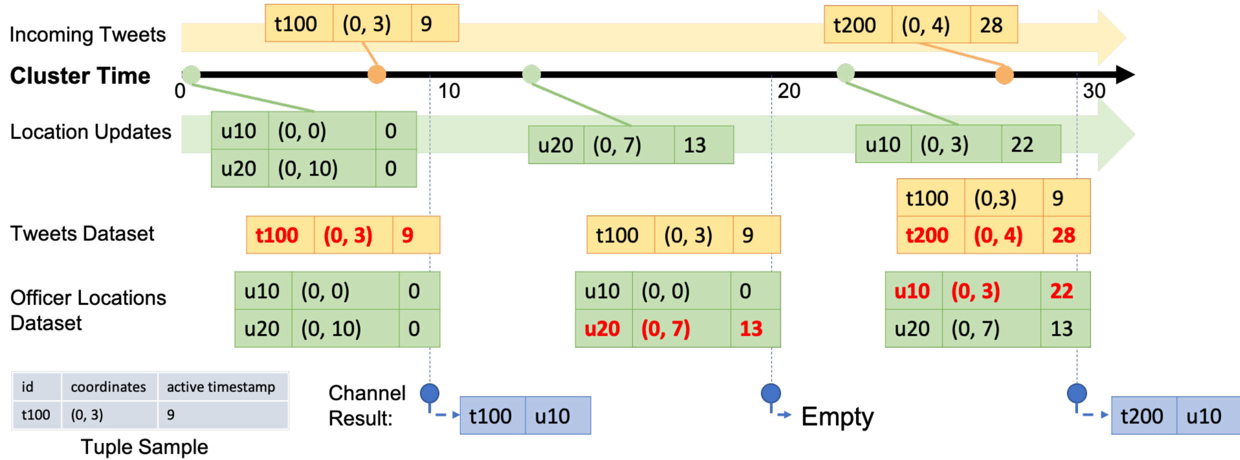


Figure 3.33: Officer u10 subscribing to NewNearbyHatefulTweetsForActiveOfficers(u10) and officer u20 subscribing to NewNearbyHatefulTweetsForActiveOfficers(u20)

3.6 GOOD: A Not BAD Approach

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

3.6.1 The GOOD Architecture

Following our discussion in Section 3.3.1, a GOOD system also needs to serve all three types of BAD users: **Subscribers** who want to customize data and receive constant updates, **Developers** who create BAD applications to serve subscribers, and **Analysts** who analyze data using declarative queries. Such a system should provide the following features:

- Efficient data ingestion for rapid incoming data.

- Data customization based on a large volume of subscriptions.
- Data analytics with a declarative language.
- Persistent storage for incoming data and other relevant information.
- Customized data delivery to a large number of subscribers.

An existing Big Data system alone can only fulfill a portion of the BAD requirements. For example, Apache Spark Structured Streaming offers on-the-fly data processing but lacks persistent storage. Amazon's Simple Notification Service (SNS) supports cloud-based pub-/sub, but the expressiveness of subscriptions is limited to the content of publications. A user wanting to build BAD applications would thus have to glue multiple systems together. We can break down a proposed GOOD system architecture into different components and categorize existing Big Data systems with respect to this GOOD architecture, as shown in Figure 3.34. We describe the functionality of each component as follows:

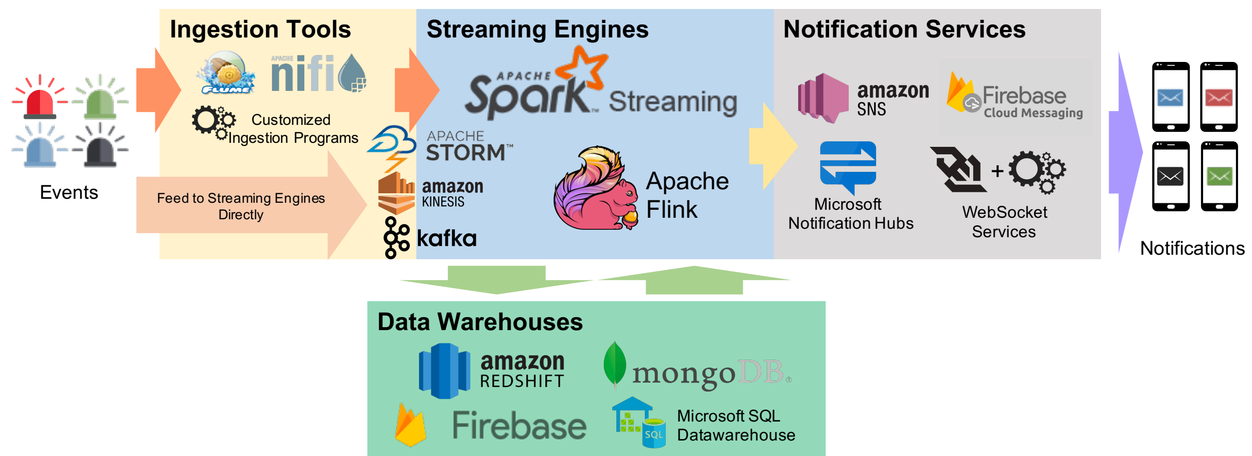


Figure 3.34: GOOD Architecture

- **Ingestion tools** collect data from external data sources and help distribute the data to downstream components. In some cases, users could implement their own ingestion programs to handle specific ingestion protocols. With the growth of stream processing,

many ingestion tools now also support on-the-fly data processing (with various limitations). This blurs the boundary between ingestion tools and streaming engines. Here we still consider them as different GOOD components to emphasize their functional differences.

- **Streaming engines** today come in two different flavors. One (e.g., Apache Storm, Apache Kafka) focuses on efficient and reliable data distribution and allows users to hang data processing units onto the pipeline. The other (e.g., Apache Flink, Apache Spark Structured Streaming) focuses on enabling real-time data analytics as if working with non-streaming data. Users could glue multiple streaming engines together to benefit from both flavors (such as gluing Kafka with Spark Structured Streaming). GOOD can use streaming engines to combine incoming data, subscription information, and other relevant data to produce customized notifications.

- **Data warehouses** provide data persistence and support data analytics. We use data warehouses as the storage engine of the GOOD system. Incoming data is persisted in data warehouses for retrospective analysis. Subscriptions and other relevant data used for producing customized data are also persisted in data warehouses and loaded into streaming engines for processing. We do not replicate data in both streaming engines and data warehouses to avoid data inconsistency and the constant migration of updates between them.

- **Notification services** deliver customized data produced by streaming engines to interested subscribers. Users could choose cloud-based services such as Amazon SNS or Firebase Cloud Messaging to send notifications to subscribers via SMS or Email, or they also could build their own notification services based on WebSocket.

Every component of the GOOD system must be horizontally scalable to ensure that it can support a large number of subscribers, just like the BAD system. Even with this scalable

architecture, it would be impractical for the GOOD system to compute/customize an incoming data item for every subscriber independently, especially when the incoming data arrives rapidly. In order to best approach the BAD system's scalability requirement, we also adopt the data channel model in our GOOD system architecture by grouping similar subscriptions into a data channel and evaluating them together. Next, we will consider a sample GOOD system to explain how it can receive, customize, and deliver data.

3.6.2 A GOOD System

The GOOD architecture offers a way to approximate the BAD system by gluing multiple existing Big Data systems together. One could choose various combinations among the options in Figure 3.34 for creating a GOOD system. In order to compare the GOOD system with the BAD system toe-to-toe, we have constructed a sample GOOD system using several component systems that have been widely used in practice, as shown in Figure 3.35. We choose Apache Kafka for data ingestion and use Spark Structured Streaming for data processing, as suggested in the Spark Structured Streaming documentation [83]. Although Kafka also supports several data processing operations via Kafka Streams [48], we choose Spark Structured Streaming for its richer query semantics, which is closer to the BAD system's offering. Further, we use MongoDB for persistence and analytics and AmazonSNS for notification delivery. Each component of the GOOD system can be described and configured as follows:

Figure 3.35: A concrete GOOD system

- **Apache Kafka** is a distributed streaming platform that allows applications to publish and subscribe to data streams reliably. We connect external data sources to Kafka using producer APIs. For each data source, we can create a topic in Kafka to allow downstream consumers (Spark Structured Streaming and MongoDB) to access the incoming data.

- **MongoDB** is a document-based distributed database. We connect MongoDB to Kafka as a consumer via the mongodb-kafka connector [62] provided by MongoDB. Incoming data records from a Kafka topic (i.e., an external data source) are persisted in a corresponding MongoDB collection as JSON-like documents for retrospective analysis. Besides incoming data, subscriptions specifying subscribers' interest and other relevant information used for data customization and data analytics are also stored in MongoDB.

- **Apache Spark Structured Streaming** is a scalable stream processing engine built on top of the Spark SQL engine. It supports Dataframe/Dataset APIs for users to express streaming computations the same way one would express a batch computation on static data. We connect Spark Structured Streaming to Kafka as a consumer through the spark-streaming-kafka connector [82] provided by Spark. Incoming data from a Kafka topic is mapped into a data stream in Spark Structured Streaming. One can implement a data channel as a Spark application that runs continuously for producing customized data. Relevant information and subscriptions stored in MongoDB can be loaded into Structured Streaming as DataFrames through a mongodb-spark connector [61] provided by MongoDB.

- **Amazon SNS** is a notification service provided in Amazon Web Services for delivering messages to subscribed endpoints or clients. It allows users to create Amazon SNS topics and publish notifications through APIs. Other systems and end-users can subscribe to these topics and receive published data. Amazon SNS provides filter poli-

cies in subscriptions to allow subscribers to filter notifications by their content. We can use the filter policy to send notifications to certain channel subscribers by using their subscription IDs as the filter value. We map a data channel to an Amazon SNS topic, and whoever subscribes to this data channel also becomes a subscriber to the Amazon SNS topic with its subscription ID as the filter attribute. Customized data generated by the Spark channel application is published to the Amazon SNS topic with subscription IDs attached.

Due to its glued nature, the GOOD system needs "cooperation" between different components to provide BAD services. Taking the new nearby hateful tweet example described in Section 3.5 (the equivalent BAD channel defined in Figure 3.25), one would have to complete the following steps for providing the channel service in the GOOD system:

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

- For each newly subscribed subscriber, we add the subscription information into MongoDB for data customization, and we also create a corresponding Amazon SNS subscription with the subscription ID as the filter attribute.

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

3.6.3 GOOD vs. BAD

As we have mentioned, streaming engines have to age historical data out to restrain their resource usage. This limits the query semantics that can be supported by the GOOD system. Consider the new nearby hateful tweets channel defined in Figure 3.25, where we send new nearby hateful tweets to in-field officers based on their last known location by utilizing an UPSERT feed. That channel can produce notifications for a temporarily "offline" officer and later send these "missed" notifications to him/her when the officer reconnects, as discussed in Section 3.5.4.

In the GOOD system, if an officer has not sent location updates for a some time, his/her location information would be aged out by the streaming engine. Due to this limitation, a GOOD user can only look for location updates back to a limited time for a new incoming tweet. To better approximate the BAD channel, one could consider persisting all historical location updates in MongoDB and pulling the latest locations into Spark Structured Streaming in each channel execution. However, this would lose the timeliness of streaming data and introduce additional data access overhead.

To illustrate the query semantics of the GOOD system and compare that with BAD, we show an alternative new nearby hateful tweets use case. As Spark Structured Streaming does not support spatial joins on data streams, we use "area_code" to represent tweets' and officers' locations. We consider a tweet to be nearby to an officer if it is posted from the same area code as the officer. In this modified use case, we send a new hateful tweet to the nearby in-field officers who have recently (within 10 seconds) updated their locations. This use case will also be used in the later performance comparison between BAD and GOOD. An illustrative example of the modified channel execution using the data sample in Section 3.5.4 is shown in Figure 3.36. When t100 arrives at 9s, we examine the location updates in the past 10 seconds and find two officers u10 and u20 who recently updated their locations. We check the area codes of t100, u10, and u20 and produce a notification for u10. When t200 arrives at 28s, we look back in a 10-second window and find the location update from u10 at 22s, so we produce a notification for u10. Note that the location update from u20 at 13s is not "used". When t200 come at 28s, this location update of u20 is too old for the the tweet [5].

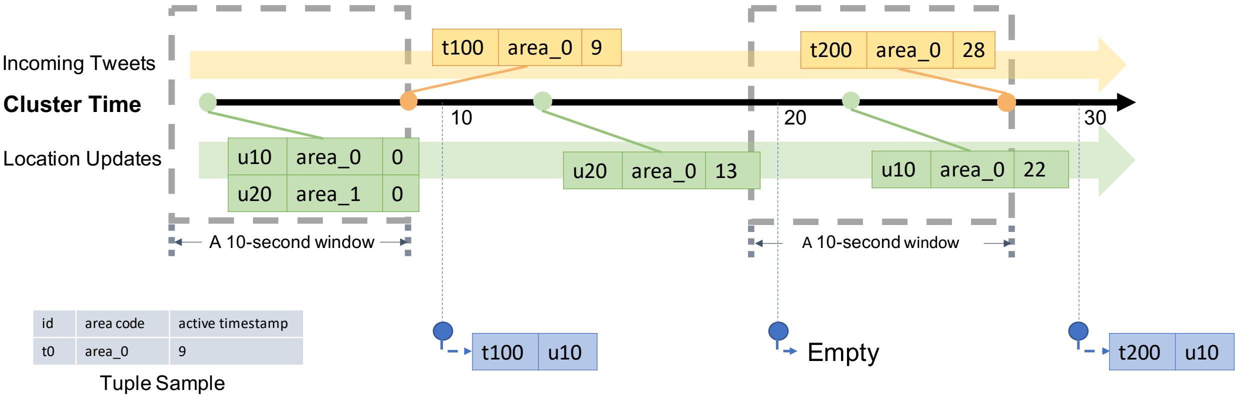


Figure 3.36: A GOOD example of sending hateful tweets to officers

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

3.7 Experimental Results

In this section, we present a set of experiments conducted to evaluate the performance of the BAD system. We focused on the performance of BAD-CQ and compared that with the GOOD system described in Section 3.6.3. We first examined the basic ingestion and query performance of active datasets. Then, we investigated BAD-CQ's continuous channel performance regarding supportable subscribers in different use cases. Also, we compared the performance of the GOOD and the BAD systems using the same use cases. Finally, we investigated the speed-up and scale-out performance of BAD-CQ when it is given more resources. Our experiments were conducted on a cluster connected using a Gigabit Ethernet switch (up to 16 nodes). Each node had a Dual-Core AMD Opteron Processor 2212 2.0 GHz, 8 GB of RAM, and a 900 GB hard disk drive.

3.7.1 Active Dataset Scale-out Performance

Since active datasets store active timestamps with records for continuous channel evaluation, writing and reading active datasets will have the same additional cost due to the additional bytes. In order to examine the performance impact of that, we conducted ingestion and query performance experiments with active datasets. We used two types of data: the Tweets and OfficerLocations defined in Figure 3.3 and Figure 3.4 respectively. Each tweet was around 140 bytes, and each user location record was around 60 bytes. An active timestamp was 9 bytes long (1 byte for data type and 8 bytes for epoch time). For both scale-out experiments, we started with 100 million records on a 2-node cluster and increased that to 400 million records on a 8-node cluster. For the ingestion performance experiments, we measured the ingestion throughput. For the query performance experiments, we measured the average time over 50 query executions for scanning all records in a dataset. The results are shown in Figure 3.37 and Figure 3.38 respectively.

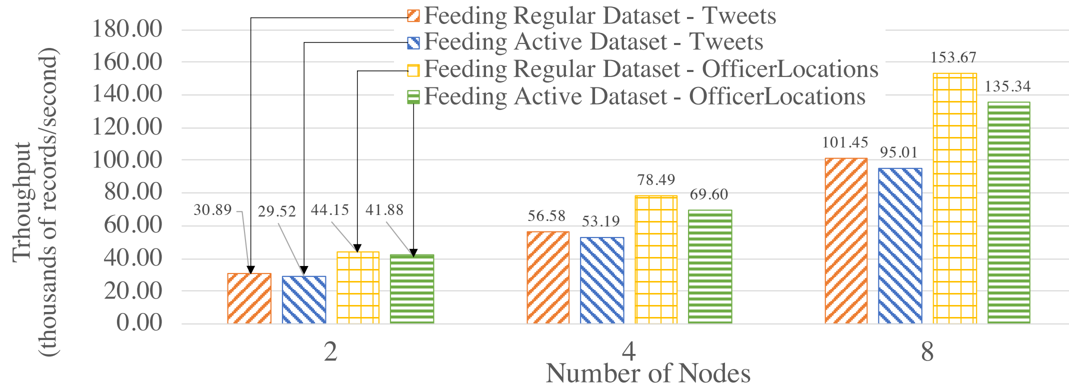


Figure 3.37: Ingestion performance on active datasets

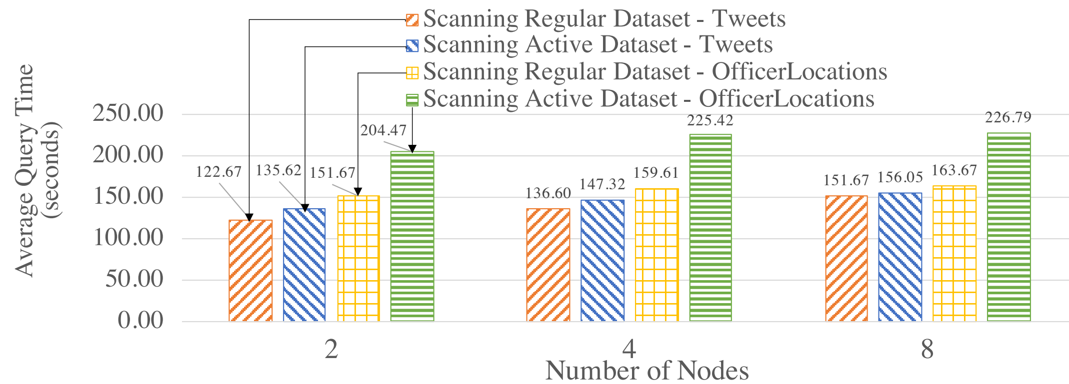


Figure 3.38: Query performance on active datasets

When ingesting data into active datasets, the additional work comes from attaching active timestamps to incoming data records and persisting them into the storage engine. As we can see from Figure 3.37, the ingestion throughput on both the Tweets and OfficerLocations datasets have some regression compared with the regular datasets. The throughput regression is proportional to the size ratio between an incoming record and the active timestamp. When an incoming record is big, the performance impact is relatively small and vice versa. With more nodes in the cluster, the throughput increases since more resources (CPU and storage bandwidth) can be used for parsing and storing incoming data.

When scanning active datatasets, the query time increases due to the additional cost of reading the large records with active timestamps from disk. Similarly, the query time increase is proportional to the size ratio between a stored record and the active timestamp. As the

cluster size grows, the query time increases slightly due to the increased query execution cost on a larger cluster, but overall it remains stable since AsterixDB shards its stored data across all nodes.

3.7.2 Channel Performance

As a channel runs periodically at a user specified period, it requires the channel evaluation to finish within that given period of time. The channel execution time depends on the channel query complexity and the size of the data involved (e.g., the number of tweets and subscribers). In order to examine the performance of data channels, we measured the maximum number of subscribers that can be supported by a channel within a given period. For these use cases, we introduce a new dataset **Schools**, defined in Figure 3.39, to store schools' information as relevant auxiliary information. A list of schools can be attached to hateful tweets to provide additional information for use by the responding in-field officers. The Schools dataset contains 10,000,000 records, and each record is around 70 bytes. We used the following four use cases to examine channel performance:

1. ***NewLocalHatefulTweets***: Send me new hateful tweets from a certain area (defined in Figure 3.40).
2. ***NewLocalHatefulTweetsWithSchools***: Send me new hateful tweets from a certain area together with information about schools in that area (defined in Figure 3.41).
3. ***NewNearbyHatefulTweets***: Send me new hateful tweets nearby (defined in Figure 3.25).
4. ***UnseenNearbyHatefulTweets***: Send me nearby hateful tweets that I've not seen before (defined in Figure 3.27).

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

Figure 3.39: Datatype and dataset definition for Schools

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

Figure 3.40: A continuous channel for new local hateful tweets

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

In all four use cases, we fixed the incoming tweet rate and searched for the maximum number of supportable subscribers in the given 10-second channel execution period while both tweets and location updates were coming. We varied the incoming tweet rate to see how channel performance changed. For the "NewNearbyHatefulTweets" channel in particular, we chose two algorithms (broadcast nested loop join and index nested loop join) to evaluate the spatial join between the incoming tweets and officers' locations. (We broadcast data from the Tweets dataset and utilized the R-Tree index on the location attribute of the OfficerLocations dataset.) We deployed BAD-CQ on a 6-node cluster and the performance results are shown in Figure 3.42. (Note the use of a log scale for the y-axis.)

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

Figure 3.41: A continuous channel for new local hateful tweets with schools

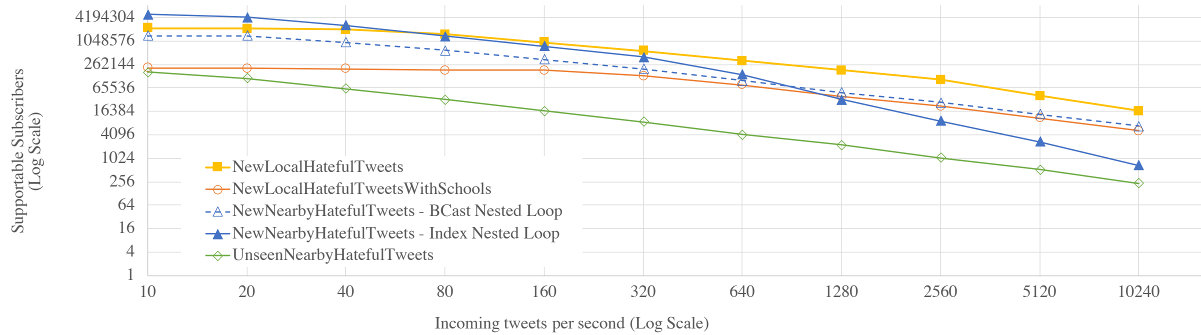


Figure 3.42: Maximum number of supportable subscribers under different incoming data rates

Depending on the channel query complexity, the maximum number of supportable subscribers varies. For all four use cases, the maximum number of supportable subscribers decreases as the incoming tweet rate increases; this is due to the increased cost of producing and persisting [6] more customized notifications. Comparing the results for "NewLocationHatefulTweets" and "NewLocalHatefulTweetsWithSchools", we see that the latter one has lower performance, as adding in school information incurs more computational and persistence cost. Comparing "NewNearbyHatefulTweets - BCast Nested Loop" and "NewNearbyHatefulTweets - Index Nested Loop", we see that the use of the index offers much better performance than scanning the whole OfficerLocations dataset when the incoming tweet rate is low. As the incoming tweet rate grows, however, the performance of the index nested loop join becomes worse than the broadcast join. The reason is that, with more incoming tweets, the maximum number of supportable subscribers decreases due to the increased cost of computing customized data. For the join operation between tweets and officer locations,

[6] As mentioned in Section 3.4.2, BAD persists customized data to disk by default to allow brokers to pull later.

then, having more tweets and fewer actual subscribers (in-field officers) increases the query's selectivity for OfficerLocations. Since the index nested loop join accesses the primary index through a secondary index, when the selectivity becomes high, the performance of using that index becomes worse than just scanning the primary dataset. Interested readers may refer to [55] for a more detailed analysis of the underlying storage engine's performance benchmarks.

3.7.3 Good vs. BAD Performance

The BAD system enables developers to create BAD services with declarative statements. The GOOD system, in contrast, requires developers to manually glue multiple systems together and orchestrate them programmatically to create BAD services. In order to show that the BAD system not only alleviates developers' effort when creating BAD services, but can also provide better performance compared with a GOOD system, we chose several use cases supported by both the BAD and GOOD systems and measured their performance on both.

We used the GOOD system detailed in Section 3.6.2 for these experiments. As we discussed in Section 3.6.3, the GOOD system cannot provide all query semantics supported in the BAD system. Not all use cases in Section 3.7.2 can be supported directly in the GOOD system. Spark Structured Streaming does not support spatial joins between streams, so we used area code to represent the location of tweets and officers. The use cases used for comparing the performance of the BAD system and the GOOD system are as follows:

1. ***NewLocalHatefulTweets***: Send me new hateful tweets from a certain area (same as Section 3.7.2).
2. ***NewLocalHatefulTweetsWithSchools***: Send me new hateful tweets from a certain area together with the schools in that area (same as Section 3.7.2).
3. ***NewHatefulTweetsForLocalActiveUsers***: Send me new hateful tweets from the

same area as my current location (similar to NewNearbyHatefulTweets in Section 3.7.2, but modified to use area_code for this experiment).

In use cases 1 and 2, subscribers subscribe to a channel with the area codes of interest. In use case 3, subscribers subscribe to the channel with the their officer IDs. Due to the high overhead of integrating Spark Structured Streaming with MongoDB, we tuned down the size of the **Schools** dataset by 5x to 2,000,000. To demonstrate the advantages that the BAD system as of utilizing indexes and different query evaluation algorithms, we picked the "NewHatefulTweetsForLocalActiveUsers" use case, and we experimented with hash join, broadcast nested loop join, and index nested loop join. In this experiment, we focused on the processing core of both systems without including result delivery using brokers. The generated notifications were persisted in storage, as in the default pull mode. All incoming data was persisted as well for retrospective analysis. The performance results in terms of the number of supportable subscribers are shown in Figure 3.43. (Note the use of a log scale for the y-axis.)

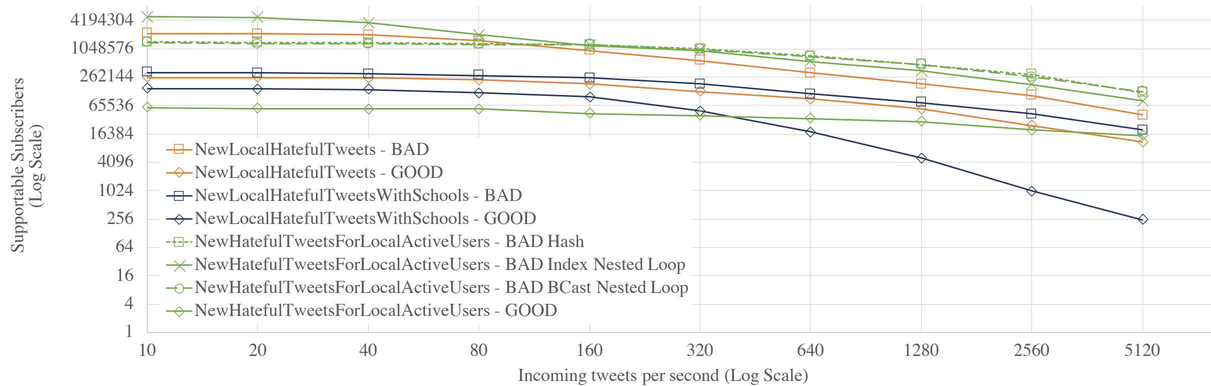


Figure 3.43: Performance comparison of BAD-CQ and the GOOD system

In all three cases, BAD-CQ outperforms the GOOD system. As the incoming tweet rate grows, the performance of both systems drop because of the increased cost of producing and persisting more notifications. Similar to Section 3.7.2, both systems have better performance for "NewLocalHatefulTweets" (colored in orange) than for "NewLocalHatefulTweetsWith-

Schools" (colored in blue) due to the additional cost of attaching relevant school information. In particular, the "NewLocalHatefulTweetsWithSchools" use case for GOOD suffers more from the increased incoming tweet rate, as the cost of persisting notifications with schools into MongoDB becomes high when there are many notifications. For "NewHatefulTweets-ForLocalActiveUsers", we see a similar performance benefit for utilizing an index and the same performance regression when the incoming tweet rate becomes high. Hash join offered only a slight advantage over a broadcast nested loop join in this case, as the total number of tweets for each channel execution is relatively small.

In order to better understand the cost of the GOOD system, we chose the "NewLocalHatefulTweets" use case with 150,000 subscribers and 80 tweets/second and measured the time consumed by each stage of its channel execution on both the GOOD and BAD system. The result is shown in Figure 3.44, which also includes the overall channel execution time. As can be seen, the GOOD channel execution spent much of its time loading Subscriptions from MongoDB. This is a consequence of the overhead of gluing different systems together, as shipping data from one sub-system to another incurs additional serialization/deserialization and data transformation and transmission costs. One could consider maintaining copies of the relevant data and subscriptions in Spark Structured Streaming as well, to accelerate the processing, but then developers would have to handle consistency challenges and need to migrate updates back and forth between Spark Structured Streaming and MongoDB. In contrast, BAD-CQ spent much less time on subscription loading. Since tweets were being ingested at the same time, there was a bit of read/write contention on the *Tweets* dataset that caused the tweet loading time to be higher than the subscription loading time on BAD-CQ.

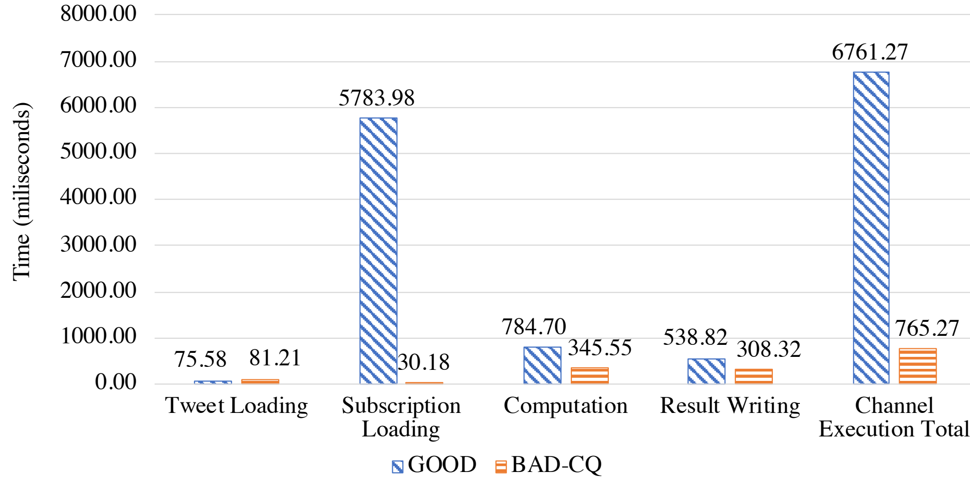


Figure 3.44: Cost of "NewLocalHatefulTweets" with 150,000 subscribers and 80 tweets/second on both the GOOD and BAD system

3.7.4 BAD Scalability

Finally, we investigated the scalabilty of BAD-CQ from two angles: *speed-up* - given a fixed workload, see if the performance improves with more resources, and *scale-out* - increase the workload together with available resources to see if the performance remains stable. We chose the "NewNearbyHatefulTweets - Bcast Nested Loop" channel and increased the channel's period to 30 seconds for this experiment. All other settings were the same as Section 3.7.2.

Speed-up experiments: The channel workload is determined by the incoming tweets per second and the number of subscribers (in-field officers). In this experiment, we fixed the incoming tweet rate to 160 tweets per second and had 140,000 subscribers. We increased the cluster size from 2 nodes to 4, 8, and 16 nodes, and we measured the channel execution times, as shown in Figure 3.45. When the cluster grows, the channel execution time is almost halved because the subscribers' locations are stored on twice as many machines. Since each node now has less data, the join between incoming tweets and officer's locations, which computes on all nodes, can finish sooner. As tweets are broadcast to all nodes in the cluster and the execution overhead also grows with the cluster size, the speed-up gain gradually decreases with larger cluster sizes.

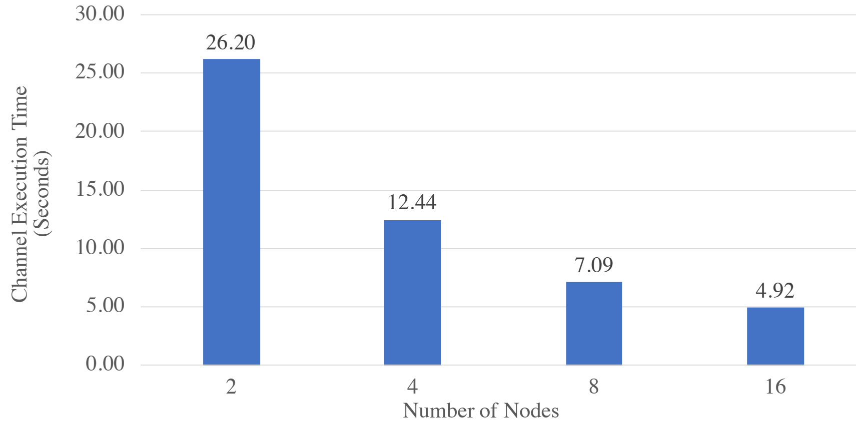


Figure 3.45: Speed-up BAD-CQ with fixed incoming tweet rate and number of subscribers

Scale-out experiments: We used two experiments to evaluate the scale-out performance of BAD-CQ. We first fixed the incoming tweet rate to 160 tweets/second and increased the cluster size from 2 nodes to 4, 8, and 16 nodes to see how many subscribers could be supported in each setting. The result is shown in Figure 3.46. As we double the size of the cluster, the maximum number of supportable subscribers almost doubles. Similar to the speed-up experiment, twice many nodes allow the join operation to handle more data in the given time period.

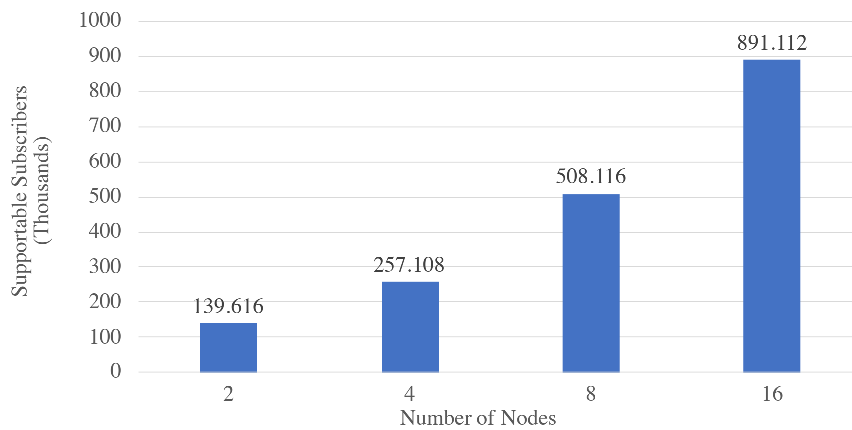


Figure 3.46: Scale-out BAD-CQ with fixed incoming tweet rate

In the second experiment, we increased the incoming tweet rate together with the cluster size. We started with a 2-node cluster with 80 incoming tweets per second, and we increased the cluster size and the incoming tweet rate by the factor of two, up to 16 nodes and 640

tweets per second. The result is shown in Figure 3.47. The channel performance maintains relatively stable as we increase the workload and add more resources at the same time.

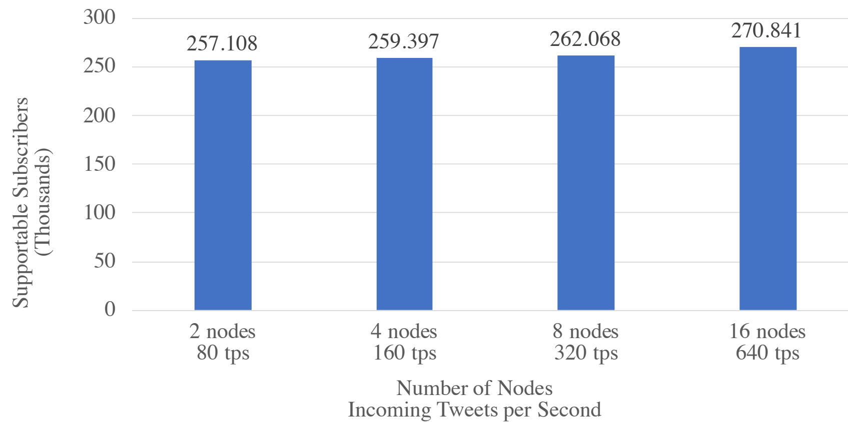


Figure 3.47: Scale-out BAD-CQ with increasing incoming tweet rate

3.8 Summary

In this chapter, we considered a world where Big Data is no longer just bytes sitting on storage devices, waiting to be analyzed, but is valuable information surrounded by active requests asking for continual "news updates". In such a Big Active Data (BAD) world, developers often need to create and manage data services to support analysts in working with declarative queries and subscribers looking for the latest updates. In order to reduce the effort for developers creating BAD services, we have built the BAD system, consisting of BAD-RQ, which "activates" a parameterized query as a data channel for subscribers to receive periodic query results of interest, and BAD-CQ, which introduces continuous (incremental) query semantics into data channels and optimizes the channel infrastructure for continuous use cases. We showed the user model, design, and implementation of our system and illustrated how developers can use it to create BAD services declaratively. To demonstrate the complexity of creating BAD services without BAD, we also presented a "GOOD" system created by gluing multiple Big Data systems together. We examined the perfor-

mance of the BAD system under different workloads and compared that with an instance of a GOOD system. The results for the use cases examined showed that the BAD system could support up to four millions subscribers on a six-node cluster, was able to horizontally scale out with more resources, and offered significantly better performance as compared with the GOOD system. In all, the BAD system provides a systematic solution for creating BAD services at scale.

Chapter 4

Sharing Big Data Declaratively at Scale

4.1 Overview

Developments in information technology have changed the ways of producing, persisting, and exchanging information. These advancements allow research institutions, companies, government agencies, and other organizations a new way to cooperate by sharing digitized information electronically [50]. Besides the ethical and legal issues, data sharing is difficult due to many challenges related to management, interoperability, security, and infrastructure [98]. It usually requires dedicated infrastructures and collaborative efforts from all participants. Researchers from academia have developed projects that unify institutional repositories from different organizations for sharing research datasets [58, 74, 87]. Companies have also created platforms and systems based on Big Data projects to improve business efficiency and consolidate resources for better services [76]. In this chapter, we focus on data sharing in a Big Active Data (BAD) world [20, 44]. In particular, we characterize a BAD

world as a group of BAD islands, where each organization runs an independent BAD island, and we “bridge” different BAD islands to create scalable data sharing services.

This chapter is organized as follows. In Section 4.2, we introduce a BAD islands example involving three organizations. We describe each individual BAD island’s settings, including its users’ requests, hosted data, and provided services. In Section 4.3, we discuss possible ways of bridging different BAD islands together for supporting BAD services. In Section 4.3.3, we introduce new variants of BAD brokers and BAD feeds that are designed for bridging BAD islands and show how to bridge different islands using declarative statements. In Section 4.5, we present a three-island prototype and use an example to show how data flows between different BAD islands. In Section 4.6, we present an actual demonstration system built on top of this prototype to illustrate how BAD islands help developers create BAD applications based on shared data. We conclude this chapter in Section 4.7.

4.2 A Three-island Example

In order to illustrate the BAD islands use case with multiple organizations, we choose a three-island example with the following organizations: the Department of Homeland Security, the Orange County Sheriff’s Department, and the University of California-Irvine. Each organization hosts an independent BAD system and serves its own BAD users with localized information.

4.2.1 BAD Island 1: Department of Homeland Security

The Department of Homeland Security (DHS) is a federal agency responsible for ensuring public security. In our example, DHS has access to all tweets posted in the United States. These tweets cannot be shared with other organizations directly due to licensing and privacy

concerns, except for the tweets that are related to potential threats. The BAD system at DHS supports data analytics by DHS analysts on collected tweets, and it allows DHS field agents to subscribe to tweets through data channels.

Since raw tweets from Twitter may not contain all necessary information, DHS might need to enrich them with other relevant data. As an example, DHS could collect weapon registration information for some sensitive twitter account holders and attach that information to tweets to provide important additional information for interested subscribers. In addition, DHS could also utilize Machine Learning (ML) algorithms to estimate the threatening level of the tweets' text and use that for later analysis. An overview of the DHS island is shown in Figure 4.1.

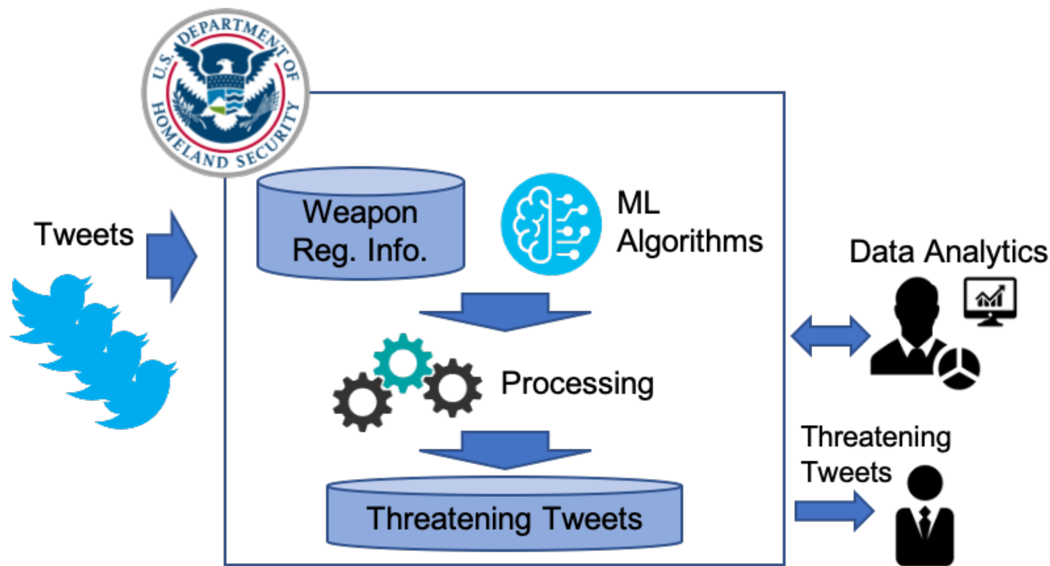


Figure 4.1: An overview of the DHS Island

4.2.2 BAD Island 2: Orange County Sheriff's Department

The Orange County Sheriff's Department (OCSD) is the local law enforcement agency that ensures safety and responds to potential crimes in Orange County. In our use case, OCSD wants to monitor major local events and ensure the safety of the event and its participants.

There are some in-field officers who patrol around the county. These officers continuously report their locations back to OCSD so that OCSD can send them instructions based on their locations (e.g., when an emergency happens, send nearby officers for help).

To prevent potential threats to local events, OCSD would like to obtain the threatening tweets posted at Orange County. When a local threatening tweet is detected, OCSD could then find important events and nearby in-field officers and then notify the officers about the event and the tweet so they could further investigate it. Additionally, OCSD wants to support data analytics on collected threatening tweets and other data in the system. An overview of the OCSD island is shown in Figure 4.2.

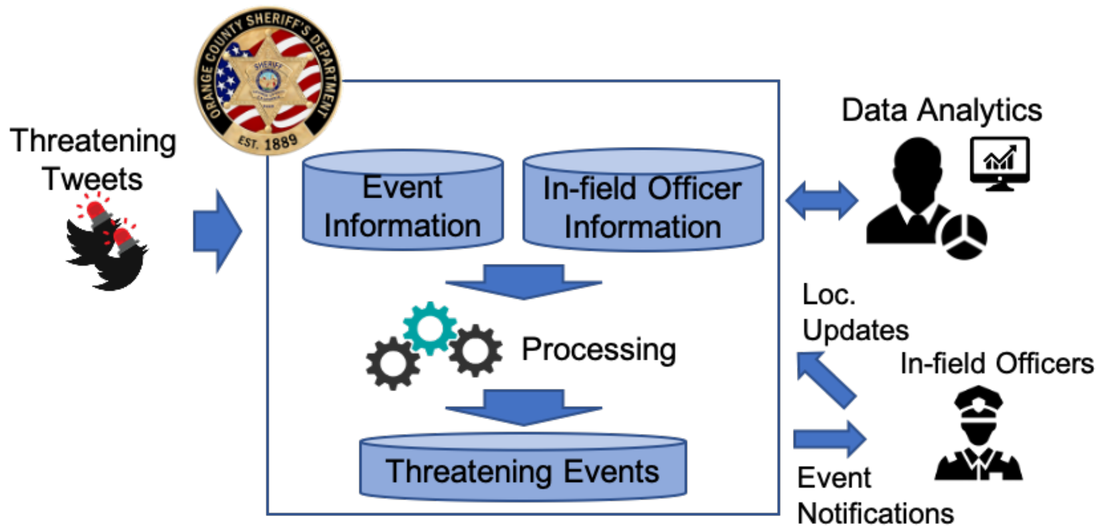


Figure 4.2: An overview of the OCSD island

4.2.3 BAD Island 3: University of California-Irvine

The University of California-Irvine (UCI) is a public university located in Irvine, a city in Orange County. The university often hosts various activities and events in different buildings on campus. To ensure students' and others' safety, the university has its own university police officers that are placed at various safety guard stations on campus, and students can seek

help from them when an emergency happens. The buildings on campus all have notice boards for showing important notifications and alerts. The university also has an alerting service - zotAlert - which delivers important messages to people (subscribers) on-campus through text messages and emails.

UCI would like to acquire the threatening tweets posted near the UCI campus and notify people in the buildings around those tweets to raise attention. An alert could include the information about nearby safety guard stations of the tweet, so that people in an emergency situation could quickly seek help. Data analytics on threatening tweets and other data in the system for school officials are also to be supported. An overview of the UCI island is shown in Figure 4.3.

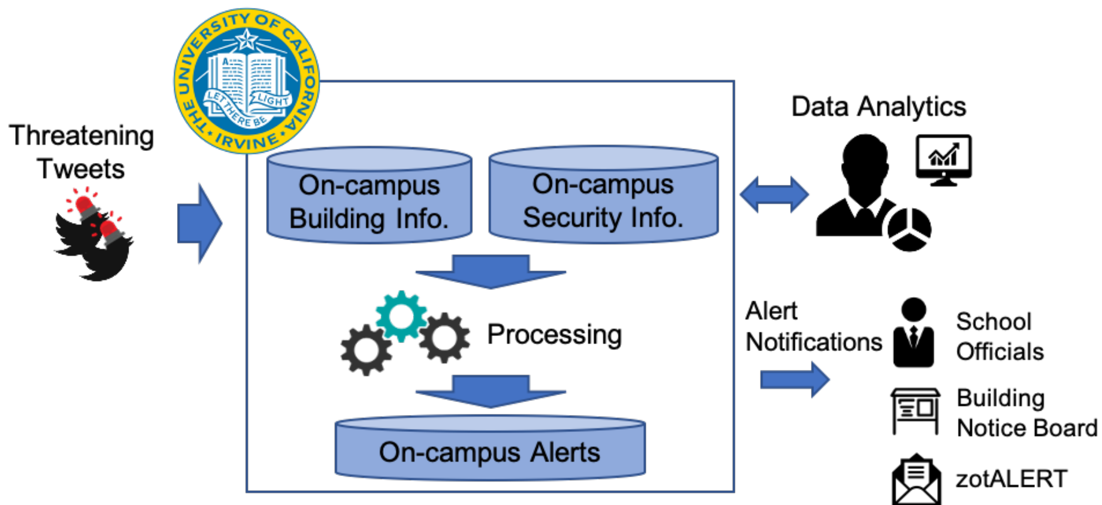


Figure 4.3: An overview of the UCI island

4.3 Island Hopping: Connecting BAD Islands

In order to support the BAD services at OCSD and UCI described in Section 4.2, we need to enable the sharing of threatening tweets detected at DHS with OCSD and UCI. These tweets can be combined with the location information at Orange County and UCI, respectively, and then be used for creating localized notifications for subscribers on each island. Below we

consider three options for sharing the threatening tweets among these islands, namely: (1) combining all islands together into one (a BAD Continent), (2) creating direct connections between the individual islands as needed (BAD Ferries) and (3) utilizing the channel idea for islands to subscribe to what they need (BAD Bridges). Below we discuss the three options in detail.

4.3.1 Option 1: A BAD Continent

Instead of sharing threatening tweets between multiple BAD islands, one could create a big BAD island, namely a BAD continent, that holds not only the data at DHS but also the local data from OCSD and UCI, as shown in Figure 4.4. In this case, all services at OCSD and UCI could be integrated into this BAD continent, and all subscribers then would subscribe to this BAD continent directly. Now, all information is in the same system. Developers from different organizations could easily create BAD services without having to share data across systems.

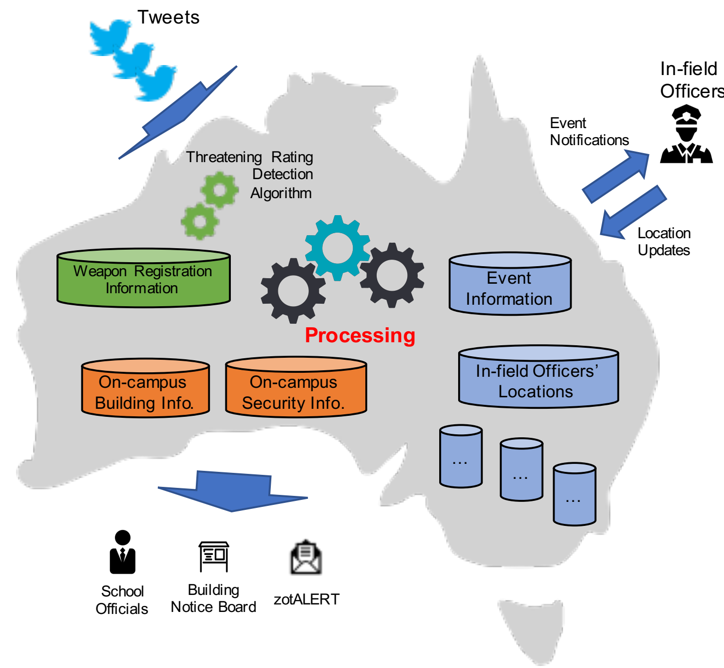


Figure 4.4: An illustration of a BAD continent

A one-for-all BAD continent could be conceptually easy to build, and it avoids the complexity of connecting different BAD islands. Although the resulting BAD system could be scaled to support the volume of data and users from multiple organizations, such integration would introduce significant management and administration overheads, especially for the service provider (DHS in this case). For the three-island example, not only would all local information (including local events, campus building layouts, etc.) need to be stored in the BAD continent, but all updates (location updates, event updates, etc.) would need to be forwarded to the system. Managing all local data at DHS could be very complex and would require sophisticated access control. When more organizations join, such a database would have to manage all kinds of additional local information while receiving updates from multiple parties; this system would quickly become impractical to maintain by one organization. Additionally, such global information sharing may not be permitted (by law) between different agencies in all cases.

4.3.2 Option 2: BAD Ferries

A different way of supporting the required BAD services at OCSD and UCI, without combining everything together, would be to send the requested data from DHS to OCSD and UCI, as shown in Figure 4.5. DHS could send the threatening tweets detected in Orange County and near UCI campus to OCSD and UCI, respectively, and OCSD BAD and UCI BAD could then combine the threatening tweets with their local information to produce localized notifications for their subscribers.

In order to share the data cleanly and efficiently, DHS would need to create a dedicated server program that allows other organizations to access the shared data in DHS. Also, OCSD and UCI would need to develop corresponding client programs connected to the DHS server program and obtain shared data. Data exchanges between the server and clients could

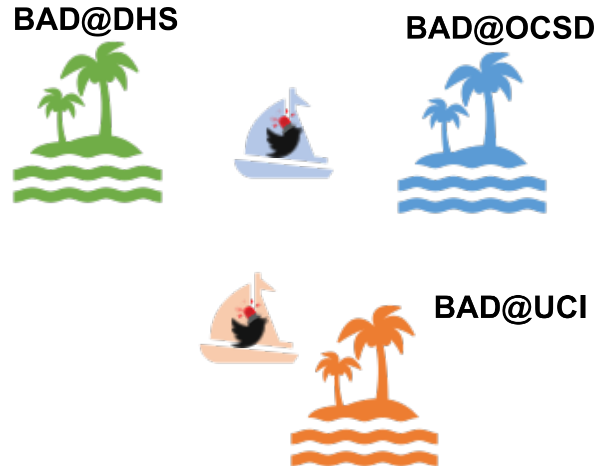


Figure 4.5: An illustration of BAD ferries

be frequent, and there could be many more clients who would like to access the shared data. Thus, the server program would need to be efficient, robust, and scalable for handling a massive number of clients and a large volume of data. Implementing the server and client programs would require significant efforts from these organizations.

4.3.3 Option 3: BAD Bridges

An important observation is that this data exchange pattern, where we have an island serving data and multiple islands constantly requesting data of interest, resonates well with the BAD user model, where subscribers subscribe to data and constantly receive updates. Inspired by this, we could characterize a BAD island as being a BAD subscriber of another island and utilize data channels and data feeds to share data at scale, as shown in Figure 4.6.

Following our example, we could create a data channel on DHS BAD, which serves threatening tweets by areas, namely via a threateningTweetsAt channel, and other islands interested in local threatening tweets from an area could then subscribe to this channel with the area name. OCSD BAD, as a subscriber, can subscribe to this channel with the parameter "OC", and UCI BAD, as another subscriber, can also subscribe to this channel with the parameter

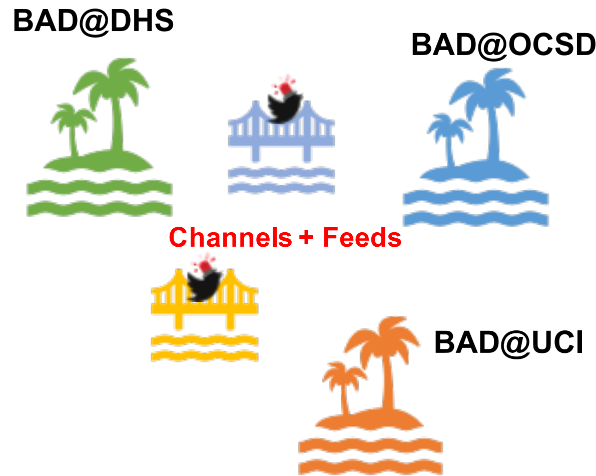


Figure 4.6: An illustration of BAD bridges

"UCI". Since subscribers like OCSD and UCI BAD all have persistent storage, we could use a push channel (discussed in Section 3.4.2) to push notifications directly. This would enable OCSD and UCI BAD to receive local threatening tweets right after a channel execution and would ensure that all data is persisted (in different BAD systems' storage).

On OCSD and UCI BAD, we could utilize data feeds to receive threatening tweets detected by the threateningTweetsAt channel on DHS BAD. Taking OCSD BAD as an example, we could create an HTTP feed and connect it to a local OCSD dataset for persisting the threatening tweets. We could register the feed's HTTP endpoint as a broker in DHS BAD and then subscribe to the threateningTweetsAt channel with the parameter "OC". With this feed, broker, and subscription, threatening tweets posted at Orange County and detected by DHS would then be sent to the feed's endpoint by threateningTweetsAt channel executions. These threatening tweets would then be persisted in OCSD BAD and be used for its local notification services. For other BAD islands, like UCI BAD, we could also repeat this process and easily obtain threatening tweets from their areas of interest. Since the BAD system is scalable and can support a large number of subscribers with a large volume of data, bridging BAD systems using data channels and feeds can be scaled out to support many more islands connecting to DHS. This allows developers to declaratively create data

sharing services without additional programming and gluing together multiple systems.

4.4 Building BAD Bridges

In Section 4.3, we discussed possible ways of connecting multiple BAD islands and proposed the use of data channels and feeds provided in BAD to enable data sharing at scale. Following the BAD Bridges approach, we introduce *BAD brokers* to enhance data exchanges between BAD islands and *BAD feeds* to help users create and manage bridges.

4.4.1 BAD Brokers

The broker sub-system in BAD manages the communication between the BAD system and its subscribers. A broker registers itself as an HTTP endpoint in the BAD system. Notifications produced by the BAD back-end are delivered to this broker endpoint, and the broker then disseminates these notifications to subscribers who subscribed on this broker. In order to allow brokers to parse notifications conveniently, data channels produce notifications as JSON objects, and more complex data types in the AsterixDB Data Model (ADM) (such as datetimes, points, etc.) are transported as strings, arrays, and other JSON data types. Since BAD islands are "brokers" that can also process ADM records, we could deliver their notifications in the ADM format to maintain the richer data type information and avoid additional data encoding overheads.

To achieve this, we introduce a new notion of *BAD brokers* and a new syntax for creating brokers in BAD, as shown in Figure 4.7. Users can add an optional WITH statement for providing additional information about the broker. While we only support "broker-type" for now, this can be further extended to support other features in the future. When there is no WITH statement or the broker-type is set to "general", we create a general broker

that takes JSON formatted data. When the broker-type is set to "BAD", we create a BAD broker that takes ADM records. In general, a channel could have subscriptions from both types of brokers. In that case, channel executions will send JSON formatted data to general brokers and ADM formatted data to BAD brokers.

```
CREATE BROKER BROKER_NAME AT "http://BROKER_HOST:PORT_NUM" WITH {
  "broker-type" : "BAD"
};
```

Figure 4.7: Creating a BAD broker

4.4.2 BAD Feeds

Bridging a BAD island A to another BAD island B and obtaining data from island B requires several steps: create a data feed on island A; register the feed as a BAD broker on island B; and make a subscription on island B on the created broker. Also, removing island A from island B's subscription list requires unsubscribing from the channel and removing the BAD broker. In order to simplify the process of bridging BAD islands and help users manage the life-cycles of bridges, we introduce the notion of *BAD feeds* into the system.

One can create a BAD feed on island A and connect it to a channel on island B using the statements in Figure 4.8. Unlike regular data feeds, users would need to specify several additional configuration entries for connecting to a data channel on the other BAD island. In particular, the "bad-host", "bad-channel", and "bad-dataverse" configuration parameters help the system locate the data channel on the other island, while "bad-channel-parameters" contains subscription parameters as a quote-escaped string for subscribing to the channel. When a channel takes multiple parameters, we use commas to separate each of them. When a data feed wants to subscribe to a channel with different parameters (e.g., OCSD BAD subscribes to threatening tweets from both Orange County and UCI), we can concatenate them using semicolons.

```
CREATE FEED A_SAMPLE_BAD_FEED WITH {
  "adapter-name" : "http_adapter",
  "address-type" : "IP",
  "format" : "ADM",
  "addresses" : "ISLAND_A_HOST:ISLAND_A_FEED_PORT",
  "type-name" : "INCOMING_DATA_TYPE",
  "bad-host" : "ISLAND_B_HOST",
  "bad-channel" : "ISLAND_B_CHANNEL_NAME",
  "bad-channel-parameters": "CH_PARAM_1-1,
     ↪ CH_PARAM_1-2;CHANNEL_PARAM2-1, CH_PARAM_2-2",
  "bad-dataverse": "ISLAND_B_CHANNEL_DATAVERSE"
};
```

Figure 4.8: Creating a BAD feed

The bridge's information is persisted in the BAD system's metadata with a feed's configuration when the feed is created. When a user starts a BAD feed on a local BAD system (island A), the system (island A) creates a broker on the specified remote BAD system (island B) using the feed's endpoint and subscribes to the channel using the provided parameters automatically. When a user stops a BAD feed, the local system (island A) unsubscribes from the channel and then removes the broker from the remote BAD system (island B). We tie the start and stop events of a data feed to the subscribe and unsubscribe actions on the remote BAD system, so when the feed is not running, the remote BAD system would not need to compute and deliver data to this BAD feed.

4.5 A Prototype of BAD Islands

In order to show how to bridge multiple BAD islands together, we now describe a prototype of BAD islands that supports the use cases described in Section 4.2. We first illustrate how to create and connect three BAD systems using declarative statements, and then we use an example to show how data flows among these islands.

4.5.1 BAD Trinity Use Case

This prototype consist of three coupled BAD systems. We use the dynamic data feeds introduced in Chapter 2 to add additional information to incoming data, and we use BAD-CQ as introduced in Chapter 3 to ensure continuous query semantics in channel computations. Each BAD system can be created as follows.

DHS BAD

DHS BAD intakes tweets from external data sources. In order to ingest and persist incoming tweets, we first create a data type "Tweet" and a dataset "Tweets", as shown in Figure 4.9. The Tweet data type is open, so later enriched attributes can also be persisted with the tweet records. We use an active dataset for Tweets, which makes sure all incoming tweets can be processed continuously in the later channel executions.

```
USE dhs;
CREATE TYPE Tweet AS {
  tid: bigint,
  uid: bigint,
  text: string
};
CREATE ACTIVE DATASET Tweets(Tweet) PRIMARY KEY tid;
```

Figure 4.9: Data type and dataset definition for tweets

We create an HTTP data feed for intaking tweets, as shown in Figure 4.10. Tweets come into the system as JSON documents, and they are parsed into ADM records and then persisted in BAD storage. In order to provide additional information for subscribers, we enrich the incoming tweets using User-defined Functions (UDFs) to be discussed shortly. Since the enrichment computations are stateful, and the reference data used for data enrichment can be updated from time to time, we use the dynamic data feed introduced in Chapter 2 with adjustable batch size.

```
USE dhs;
CREATE FEED TweetFeed WITH {
  "adapter-name" : "http_adapter",
  "addresses" : "DHS_HOST:10011",
  "address-type" : "IP",
  "type-name" : "Tweet",
  "format" : "JSON",
  "batch-size" : "BATCH_SIZE",
  "dynamic": true
};
```

Figure 4.10: Definition for TweetFeed

We enrich incoming tweets with two additional attributes. First, we enrich an incoming tweet with the tweet's user's weapon registration records (if any). In order to store the weapon registration records of sensitive tweet users, we create a data type "WeaponRegistration" and dataset "WeaponRegistrations", as shown in Figure 4.11. A user may register multiple weapons.

```
CREATE TYPE WeaponRegistration AS {
  wrid: uuid,
  uid: bigint,
  weapon_name: string
};
CREATE DATASET WeaponRegistrations(WeaponRegistration)
  PRIMARY KEY wrid AUTOGENERATED;
```

Figure 4.11: Data type and dataset definition for weapon registration information

Second, we create a Java program as a Java UDF to detect the threatening rating of a tweet's text using a list of threatening words, as shown in Figure 4.12. In this UDF, we load an external list of threatening words and use the number of threatening words in the given text as its threatening rating.

In order to actually attach the enrichment to the incoming data and pre-process the incoming tweets for later computation, we create a SQL++ UDF using the statements shown in Figure 4.13. In this UDF, we transform the epoch time of a tweet's "created_at" attribute

```
...
@Override
public void evaluate(IFunctionHelper functionHelper) throws Exception {
    JString input = (JString) functionHelper.getArgument(0);
    JInt output = (JInt) functionHelper.getResultObject();
    String tweetText = input.getValue();
    int threateningRating = 0;
    String[] words = tweetText.split(" ");
    for (String word : words) {
        if (threateningWordList.contains(word.replaceAll("[,.]", ""))) {
            threateningRating++;
        }
    }
    output.setValue(threateningRating);
    functionHelper.setResult(output);
}

@Override
public void initialize(IFunctionHelper functionHelper) throws Exception {
    threateningWordList = new ArrayList<>();
    functionParameters = functionHelper.getParameters();
    wordListPath = functionParameters.get("wordListPath");
    Files.lines(Paths.get(wordListPath)).
        forEach(keyword -> threateningWordList.add(keyword));
}
...
```

Figure 4.12: A Java UDF for detecting threatening rating

into a datetime attribute “timestamp”, and we create a point attribute “location” using the array of coordinates. These ADM data attributes can be useful, as they do not need to be constructed in computations like spatial joins every time. We use the Java UDF defined in Figure 4.12 to extract the threatening rating of the tweet's text and attach it as a “threatening_rating” attribute. We use a sub-query to look for the weapon registration information of the tweet's user and nest the registered weapons into a “user_registered_weapon” attribute. These attributes can be merged into the tweet and be persisted for producing notifications.

We apply the function defined in Figure 4.13 to TweetFeed, connect the feed to the Tweets dataset, and start the ingestion using the statement in Figure 4.14. With the enriched threatening tweets, we can then serve threatening tweets from areas by creating the continuous data channel “ThreateningTweetsAt” defined in Figure 4.15. To put everything together,

```
USE dhs;
CREATE FUNCTION EnrichTweet(tweet) {
  object_merge(tweet, {
    "timestamp" : datetime_from_unix_time_in_ms(tweet.created_at),
    "location" :
      create_point(tweet.coordinates[0], tweet.coordinates[1]),
    "threatening_rating" : threateningRating(tweet.text),
    "user_registered_weapon": (SELECT VALUE w.weapon_name
      FROM WeaponRegistrations w WHERE w.uid = tweet.uid)
  })
};
```

Figure 4.13: Data type and dataset definition for weapon registration information

```
USE dhs;
CONNECT FEED TweetFeed to DATASET Tweets APPLY FUNCTION EnrichTweet;
START FEED TweetFeed;
```

Figure 4.14: Applying the enrichment UDF to the feed and starting the ingestion

```
USE dhs;
CREATE CONTINUOUS PUSH CHANNEL ThreateningTweetsAt(area_name)
   ↪ PERIOD duration("PERIOD_DURATION") {
    SELECT t.area_name, t.text, t.location, t.threatening_rating,
       ↪ t.user_registered_weapon
    FROM Tweets t
    WHERE t.area_name = area_name
      AND t.threatening_rating > 0
      AND is_new(t)
};
```

Figure 4.15: Definition of the ThreateningTweetsAt channel

an overview of the DHS BAD system is shown in Figure 4.16.

OCSD BAD

OCSD BAD receives threatening tweets posted at Orange County and notifies in-field officers about nearby threatening tweets if these tweets are close to important local events. To persist event information in OCSD BAD, we create a data type “Event” and a dataset “Events” as

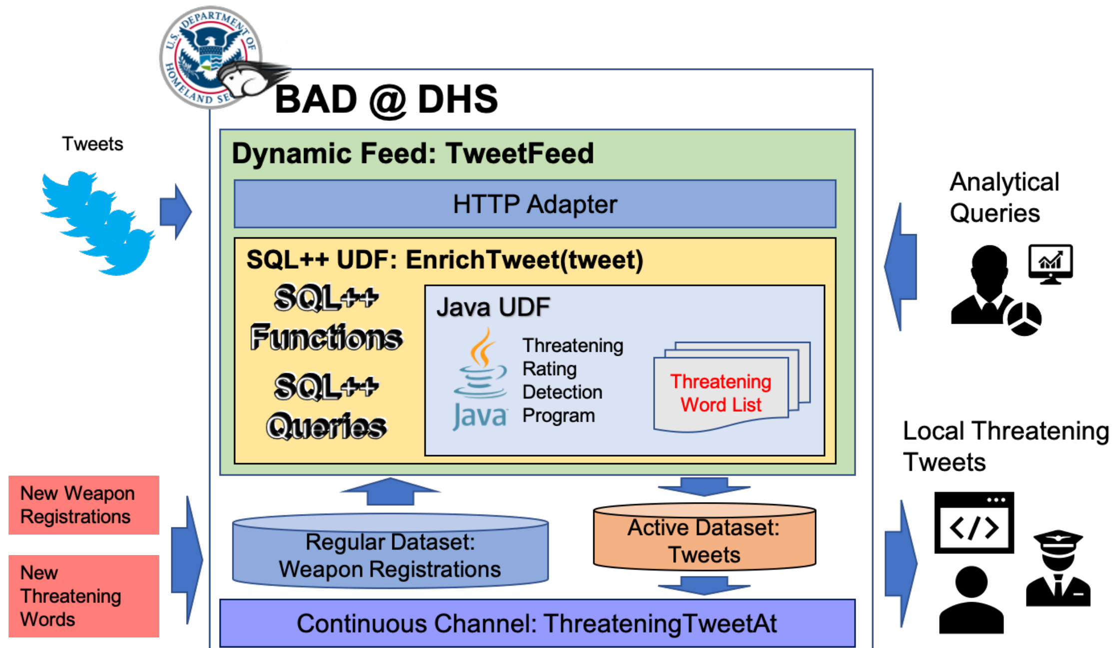


Figure 4.16: An overview of the DHS BAD system

```
USE ocpd;
CREATE TYPE Event AS {
  eid: uuid,
  name: string,
  location: point,
  event_duration: duration,
  radius_km: double
};
CREATE DATASET Events(Event) PRIMARY KEY eid;
```

Figure 4.17: Data type and dataset definition for events

shown in Figure 4.17.

To persist threatening tweets coming from DHS BAD, we create a data type "LocalThreateningTweet" and an active dataset "LocalThreateningTweets" in Figure 4.18. We use an active dataset for threatening tweets to ensure continuous query semantics in later local channel computations (to avoid missing threatening tweets). We create a BAD feed in Figure 4.19 to receive threatening tweets near both Orange County and UCI from DHS BAD. This BAD feed subscribes to the threateningTweetsAt channel with "OC" and "UCI" as its

parameters, which correspond to two separate subscriptions in DHS BAD. Since there is no data enrichment during ingestion, we can use a static data feed to maximize the throughput and we can connect this feed to LocalThreateningTweets directly.

```
USE ocpd;
CREATE TYPE LocalThreateningTweet AS {
  channelExecutionEpochTime: bigint,
  dataverseName: string,
  channelName: string
};
CREATE ACTIVE DATASET LocalThreateningTweets(LocalThreateningTweet)
  PRIMARY KEY channelExecutionEpochTime;
```

Figure 4.18: Data type and dataset definition for local threatening tweets at Orange County

```
USE ocpd;
CREATE FEED LocalThreateningTweetFeed WITH {
  "adapter-name" : "http_adapter",
  "addresses" : "OCSD_HOST:10013",
  "address-type" : "IP",
  "type-name" : "LocalThreateningTweet",
  "format" : "adm",
  "bad-host" : "DHS_HOST",
  "bad-channel" : "ThreateningTweetsAt",
  "bad-channel-parameters": "\"OC\";\"UCI\"",
  "bad-dataverse": "dhs",
  "dynamic": false
};
CONNECT FEED LocalThreateningTweetFeed TO
  DATASET LocalThreateningTweets;
START FEED LocalThreateningTweetFeed;
```

Figure 4.19: Definition, connect and start feed statements for LocalThreateningTweetFeed

In OCSD BAD, in-field officers continuously send their location updates to the OCSD BAD system so that it can notify in-field officers about nearby threatening tweets based on their current location. We create a data type "OfficerLocation" and a dataset "OfficerLocations" for describing and persisting in-field officers' location updates, as shown in Figure 4.20. We use an active dataset for officer locations as well to ensure continuous query semantics in later channel computations. For this use case, we only need the latest location of an officer,

so we choose the officer id (oid) as the primary key of OfficerLocations and use an UPSERT data feed for intaking officers' location updates, as shown in Figure‘4.21. One could instead use an auto-generated primary key to keep the whole movement history if needed. As there is no enrichment for location updates, this feed can be static as well.

```
USE ocpd;
CREATE TYPE OfficerLocation AS {
  oid: bigint,
  location: point
};
CREATE ACTIVE DATASET OfficerLocations(OfficerLocation)
  PRIMARY KEY oid;
```

Figure 4.20: Data type and dataset definition for officer location

```
USE ocpd;
CREATE FEED OfficerLocationFeed WITH {
  "adapter-name" : "http_adapter",
  "addresses" : "OCSD_HOST:10012",
  "address-type" : "IP",
  "type-name" : "OfficerLocation",
  "format" : "adm",
  "upsert" : true,
  "dynamic": false
};
CONNECT FEED OfficerLocationFeed TO DATASET OfficerLocations;
START FEED OfficerLocationFeed;
```

Figure 4.21: Definition, connect and start feed statements for OfficerLocationFeed

With the local threatening tweets, event information, and officers' locations, we can now create a continuous channel for in-field officers to subscribe to nearby threatening tweets close to local events (threatening events), as shown in Figure 4.22. The incoming notification contains threatening tweets as an array in the "results" attribute, so we use the UNNEST operation to access each independent threatening tweet. We calculate the distance between the officer and the tweet, the event and the tweet, and the officer and the event. If the officer is near a threatening tweet and the threatening tweet is near an event, we create a notification for the officer. We include the tweet's content, the event information, the

distance between the officer and the tweet, and the distance between the officer and the event in the notification to help the officer take further actions. An overview of the OCSD BAD system is shown in Figure 4.23.

```
USE ocpd;
CREATE CONTINUOUS PUSH CHANNEL ThreateningEventsNear(oid) PERIOD
    ↪ duration("PT5S") {
  FROM LocalThreateningTweets tn, OfficerLocations o, Events e
  UNNEST tn.results threatening_tweet
  LET tweet_loc = threatening_tweet.result.location,
  officer_tweet_dist = spatial_distance(o.location, tweet_loc),
  event_tweet_dist = spatial_distance(e.location, tweet_loc),
  officer_event_dist = spatial_distance(o.location, e.location)
    WHERE is_new(tn) AND oid = o.oid AND officer_tweet_dist < 0.1 AND
        ↪ event_tweet_dist < e.radius_km / 100
  SELECT oid, threatening_tweet.result tweet_content, e event_info,
    officer_tweet_dist * 100 as tweet_distance_km, officer_event_dist *
        ↪ 100 as event_distance_km
};
```

Figure 4.22: Definition of the ThreateningEventsNear channel

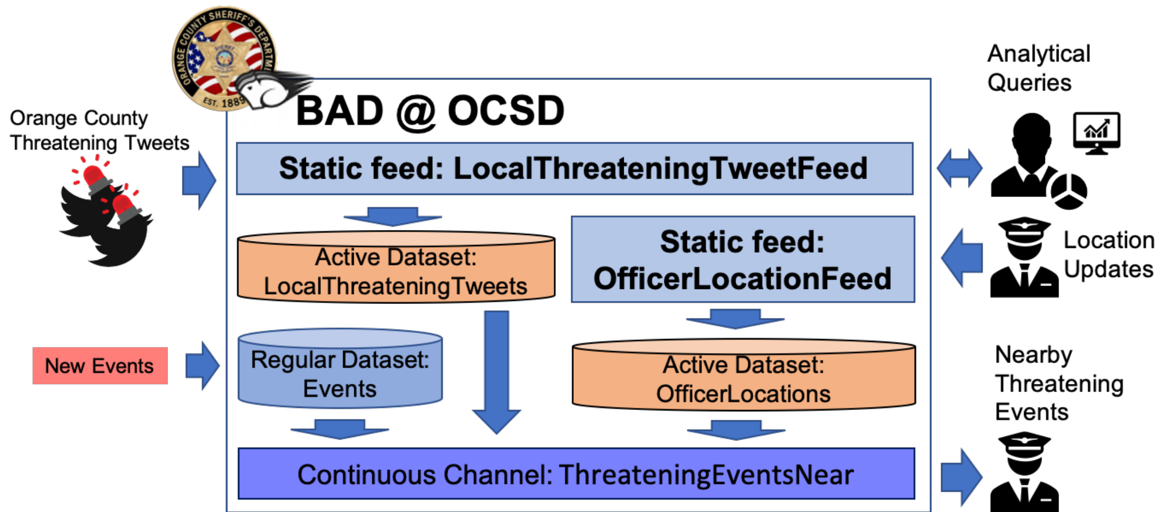


Figure 4.23: An overview of the OCSD BAD system

UCI BAD

UCI BAD receives threatening tweets posted at UCI and checks whether a threatening tweet is near an on-campus building. If so, it creates a notification about the threatening tweet together with the nearby security guard stations' information. Like OCSD BAD, in order to receive threatening tweets at UCI, we need to create a data type, a dataset, and a BAD feed. The statements for connecting to DHS BAD and obtaining threatening tweets at UCI are shown in Figure 4.24.

```
USE uci;
CREATE TYPE LocalThreateningTweet AS {
  channelExecutionEpochTime: bigint,
  dataverseName: string,
  channelName: string
};

CREATE ACTIVE DATASET LocalThreateningTweets(LocalThreateningTweet)
  PRIMARY KEY channelExecutionEpochTime;

CREATE FEED LocalThreateningTweetFeed WITH {
  "adapter-name" : "http_adapter",
  "addresses" : "127.0.0.1:10014",
  "address-type" : "IP",
  "type-name" : "LocalThreateningTweet",
  "format" : "adm",
  "bad-host" : "127.0.0.1",
  "bad-channel" : "ThreateningTweetsAt",
  "bad-channel-parameters": "\"UCI\"",
  "bad-dataverse": "dhs",
  "dynamic": false
};
CONNECT FEED LocalThreateningTweetFeed TO DATASET LocalThreateningTweets;
START FEED LocalThreateningTweetFeed;
```

Figure 4.24: Connecting UCI BAD to DHS BAD

In order to store the information about on-campus buildings and use it for checking whether there is a threatening tweet nearby, we create a data type "Building" and a dataset "Buildings", as shown in Figure 4.25. Also, we create a data type "SecurityStation" and a dataset "SecurityStations" in Figure 4.26 for providing additional information in notifications.

```
USE uci;
CREATE TYPE Building AS {
  bid: uuid,
  name: string
};
CREATE DATASET Buildings(Building) PRIMARY KEY bid AUTOGENERATED;
```

Figure 4.25: Data type and dataset definition of buildings

```
CREATE TYPE SecurityStation AS {
  sid: bigint,
  location: point
};
CREATE DATASET SecurityStations(SecurityStation) PRIMARY KEY sid;
```

Figure 4.26: Data type and dataset definition of security guard stations

With the local threatening tweets, on-campus building information, and security guard station information, we can create a continuous channel called "AlertsOnCampus" to provide on-campus alerts about threatening tweets near buildings with security guard stations' information attached using the statement shown in Figure 4.27. Like the ThreateningEventsNear channel in OCSD BAD, we first UNNEST threatening tweets from incoming notifications. Then, we check whether a threatening tweet is posted in an on-campus building. If so, we attach the security guard station information to the threatening tweet, with stations ordered by their distances to the tweet's location, and generate an alert. An overview of the UCI BAD system is shown in Figure 4.28.

```
USE uci;
CREATE CONTINUOUS PUSH CHANNEL AlertsOnCampus() PERIOD
   ↪ duration("PERIOD_DURATION") {
  FROM LocalThreateningTweets tn, Buildings b
  UNNEST tn.results threatening_tweet
  LET tweet_loc = create_point(threateningTweet.result.location[0],
     ↪ threateningTweet.result.location[1]),
    station_dist = (FROM SecurityStations s
      LET dist = spatial_distance(tweet_loc, s.location)
      SELECT s stationInfo, dist * 100 dist_km
       ORDER BY dist
    )
  WHERE is_new(tn) AND spatial_intersect(tweet_loc, b.area)
  SELECT threateningTweet.result tweet_content, b building_info,
     ↪ station_dist
};
```

Figure 4.27: Definition of the AlertsOnCampus channel

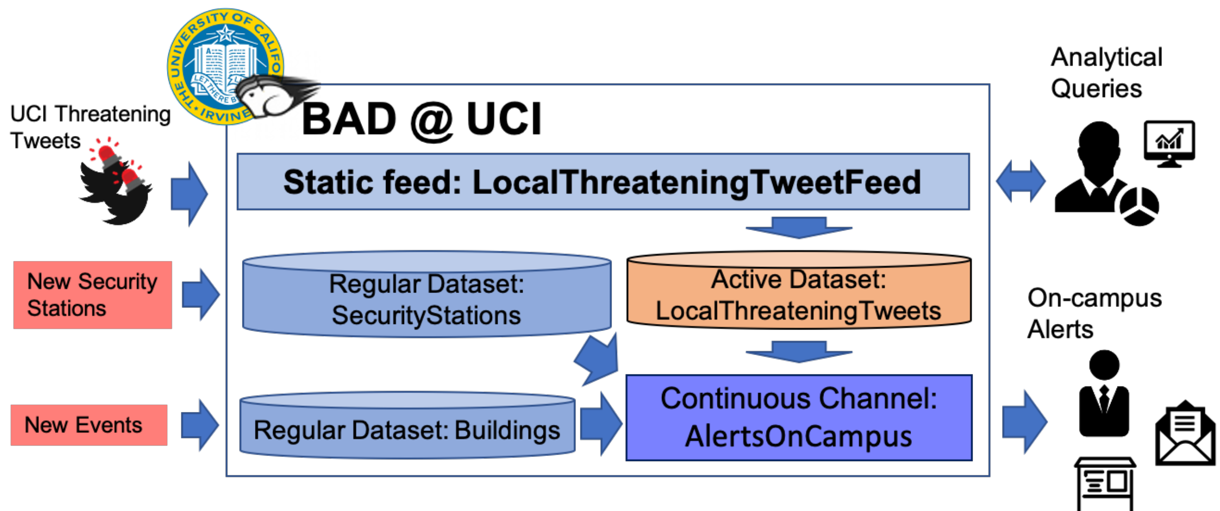


Figure 4.28: An overview of the UCI BAD system

4.5.2 The Trip of A Threatening Tweet

In order to illustrate how BAD islands interact with BAD bridges, we pick a sample tweet and show how it flows through the three islands and their bridges and produces notifications with local information for the subscribers on each island. An overview of the three-island prototype is shown in Figure 4.29. The circled numerical labels in the figure will be used later for illustrating the data content at different stages of the workflow.

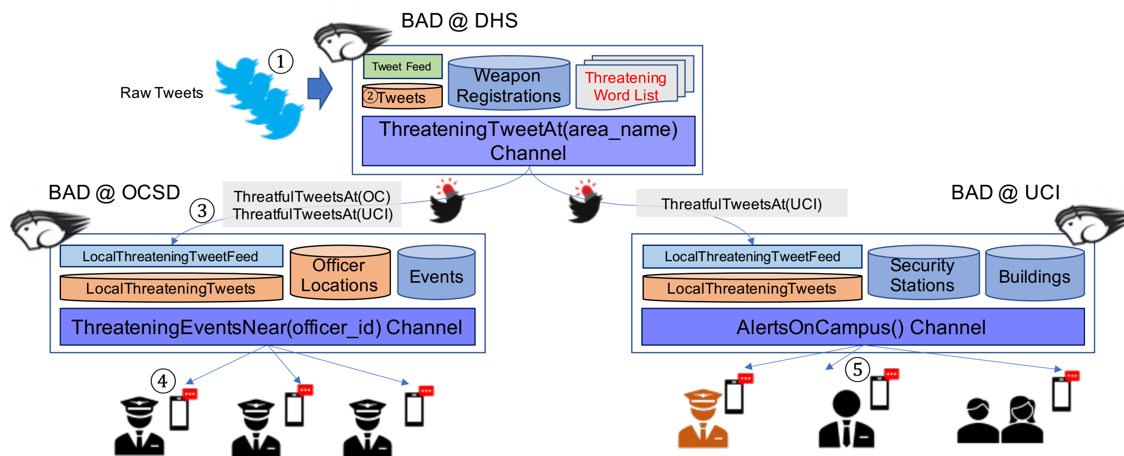


Figure 4.29: An overview of BAD island

We use the raw tweet in Figure 4.30 (labeled 1 in Figure 4.29) as the example. This tweet is posted on the UCI campus, and it contains the tweet's geolocation as a JSON array of coordinates and the epoch timestamp of when the tweet was created as a JSON number. This raw tweet is ingested by the TweetFeed defined in Figure 4.10 and then enriched by the UDF defined in Figure 4.13. After that, the enriched tweet is persisted in the Tweets dataset, with the enriched attributes, as shown in Figure 4.31 (labeled 2 in Figure 4.29). Enriched tweets contain a threatening rating detected by the Java UDF, an array of registered weapons of the tweet's user by looking up the WeaponRegistrations dataset using the "uid" attribute, a timestamp as a datetime attribute, and a location as a point attribute.

Since both the OCSD and UCI BAD systems subscribe to threatening tweets at UCI, they each receive a notification from DHS BAD about this threatening tweet. Figure 4.32 shows

```
{
  "tid": 1593142018123,
  "uid": 73,
  "area_name": "UCI",
  "text": "Saul Goodman builds SKS, and Todd Alquist fires AK47,
    ↪ but Skyler White sells Cabbage.",
  "coordinates": [ 33.64921228736088, -117.84181977473024 ],
  "created_at": 1593142018123
}
```

Figure 4.30: A sample raw threatening tweet

```
{
  "tid": 1593142018123,
  "uid": 73,
  "area_name": "UCI",
  "text": "Saul Goodman builds SKS, and Todd Alquist fires AK47,
    ↪ but Skyler White sells Cabbage.",
  "coordinates": [ 33.64921228736088, -117.84181977473024 ],
  "created_at": 1593142018123,
  "threatening_rating": 2,
  "user_registered_weapon": [ "AR10" , "AK47", "GLOCK21" ],
  "timestamp": datetime("2020-06-26T03:26:58.123Z"),
  "location": point("33.64921228736088,-117.84181977473024")
}
```

Figure 4.31: The enriched threatening tweet

the notification sent to the OCSD BAD (labeled 3 in Figure 4.29). If there is another threatening tweet posted at Orange County at the same time, the "results" array would include that tweet with a different subscription ID, as OCSD BAD has two separate subscriptions to the ThreateningTweetAt channel with "OC" and "UCI", respectively. Since UCI BAD also subscribes to the channel but with a different subscription on another broker (pointed to UCI's BAD feed), the notification for UCI BAD is produced and sent separately.

In the OCSD BAD ThreateningEventsNear channel, threatening tweets are combined with local event information and officer location information to produce threatening event notifications for in-field officers. There is one local event "OC Marathon" near the threatening tweet in Figure 4.31, and there is an in-field officer 0 nearby, so the OCSD BAD pro-

```
{
  "dataverseName": "dhs",
  "channelName": "ThreateningTweetsAt",
  "channelExecutionEpochTime": 1593142019521,
  "results": [
    {
      "result": {
        "text": "Saul Goodman builds SKS, and Todd Alquist fires
           ↪ AK47, but Skyler White sells Cabbage.",
        "area_name": "UCI",
        "location": point("33.64921228736088,-117.84181977473024"),
        "threatening_rating": 2,
        "user_registered_weapon": [ "AR10" , "AK47", "GLOCK21" ]
      },
      "channelExecutionTime": datetime("2020-06-26T03:26:59.521Z"),
      "subscriptionId":
         ↪ uuid("82e61d25-f7ad-0632-3b9a-9c26e681ad84"),
      "deliveryTime": datetime("2020-06-26T03:26:59.522Z")
    }
  ]
}
```


Figure 4.32: The generated threatening tweet notification from DHS

duces one notification about the tweet and the event for this officer. Figure 4.33 shows the threatening event notification (labeled 4 in Figure 4.29). This notification contains the event information as the "event_info" attribute, the threatening tweet's information as the "tweet_content" attribute, and the distances from the officer 0 to the tweet and to the event as the "event_distance_km" and "tweet_distance_km" respectively.

In the UCI BAD AlertsOnCampus channel, threatening tweets are combined with on-campus building information and security guard station information to produce alerts. The threatening tweet in Figure 4.31 is near the building "Student Center", so we produce a notification to notify people around this building as shown in Figure 4.34 (labeled 5 in Figure 4.29). The building information is attached to the notification. There are two security guard stations near the threatening tweet's location, so the system attaches their information with their distances, ordered by their distances to the threatening tweet. Everyone subscribed to the

AlertsOnCampus channel will receive this notification.


```
{
  "dataverseName": "ocsd",
  "channelName": "ThreateningEventsNear",
  "channelExecutionEpochTime": 1593142020436,
  "results": [
    {
      "result": {
        "event_info": {
          "eid": uuid("82e61d25-4cad-0632-3d8d-148e71cb50bf"),
          "name": "OC Marathon",
          "location": point("33.66100302712824, -117.83950620703125"),
          "event_duration": duration("PT10S"),
          "radius_km": 3.57746886883645
        },
        "tweet_distance_km": 4.854786471222485,
        "event_distance_km": 5.6839370484947755,
        "oid": 0,
        "tweet_content": {
          "text": "Saul Goodman builds SKS, and Todd Alquist fires AK47,
             ↪ but Skyler White sells Cabbage.",
          "area_name": "UCI",
          "location": point("33.64921228736088,-117.84181977473024"),
          "threatening_rating": 2,
          "user_registered_weapon": [ "AR10" , "AK47", "GLOCK21" ]
        }
      },
      "channelExecutionTime": datetime("2020-06-26T03:27:00.436Z"),
      "subscriptionId": uuid("82e61d25-47ad-0632-3e5c-22b3cb7d7df4"),
      "deliveryTime": datetime("2020-06-26T03:27:00.437Z")
    }
  ]
}
```


Figure 4.33: The generated threatening event notification from OCSD

```
{
  "dataverseName": "uci",
  "channelName": "AlertsOnCampus",
  "channelExecutionEpochTime": 1593142024344,
  "results": [
    {
      "result": {
        "buildingInfo": {
          "bid": uuid("82e61d25-43ad-0632-45d0-0ba5366832d9"),
          "name": "Student Center",
          "area": rectangle("33.64811430275051, -117.84332027249145
             ↪ 33.649382536086605,-117.84153928570557")
        },
        "stationDist": [
          {
            "stationInfo": {
              "sid": 1,
              "location": point("33.64792551859947, -117.84013290702327"),
              "name": "Station # 1"
            },
            "dist_km": 0.21216259109805177
          },
          {
            "stationInfo": {
              "sid": 0,
              "location": point("33.646866723393266,
                 ↪ -117.84170161534618"),
              "name": "Station # 0"
            },
            "dist_km": 0.23485382616041114
          }
        ],
        "tweetContent": {
          "text": "Saul Goodman builds SKS, and Todd Alquist fires AK47,
             ↪ but Skyler White sells Cabbage.",
          "area_name": "UCI",
          "location": point("33.64921228736088,-117.84181977473024"),
          "threatening_rating": 2,
          "user_registered_weapon": [ "AR10" , "AK47", "GLOCK21" ]
        }
      },
      "channelExecutionTime": datetime("2020-06-26T03:27:04.344Z"),
      "subscriptionId": uuid("82e61d25-0ead-0632-4717-e17b6a912fa6"),
      "deliveryTime": datetime("2020-06-26T03:27:04.345Z")
    }
  ]
}
```


Figure 4.34: The generated on-campus alert from UCI

4.6 System Demonstration: A BAD Islands Tour

In order to illustrate the kinds of BAD applications one could build with BAD islands and to visualize the process of data flowing through multiple systems and becoming notifications for subscribers at each organization, we have created a demonstration system based on the three-island prototype that consists of three independent dashboards supporting the BAD users from the different organizations. On each dashboard, we provide functions for manipulating local data in an organization and simulating local BAD users' actions (subscribing/unsubscribing/moving). Demonstration users can use those functions to see how different elements (the system, data, and users) interact on BAD islands.

4.6.1 DHS Dashboard

The overview of the DHS dashboard is shown in Figure 4.35. This dashboard contains four panels: the Function Panel, Analytics Panel, Visualization Panel, and Tweet Panel.

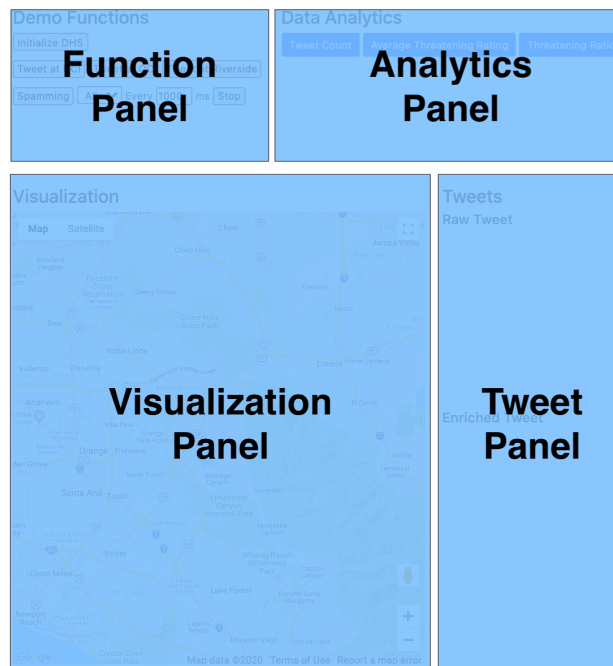


Figure 4.35: An overview of the DHS dashboard

The Function Panel, as shown in Figure 4.36, provides several functions for initializing the DHS BAD and configuring tweet feeding process. For demonstration purposes, we implement a tweet generating program that produces tweets with real geo-locations. We create tweets from three areas: Orange County, UC Irvine, and Riverside. A tweet's text is generated using a template with random subjects, verbs, and objects, and each tweet may contain up to three threatening words. A tweet's user is drawn from a pool in which 10% of the users have one or more weapon registration records. Demonstration users can create a tweet from one of the three areas or continuously feed tweets from one or all areas at a specified rate.

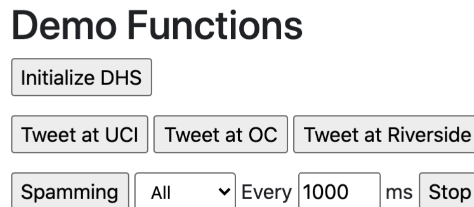
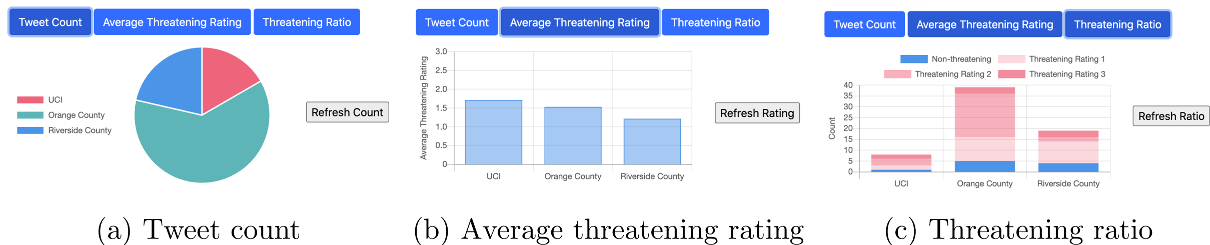


Figure 4.36: DHS dashboard functions

The Analytics Panel shows visualized results of three exemplary analytical queries, as shown in Figure 4.37. Demonstration users can use buttons to switch the analytical results. In Figure 4.37a, we count the number of tweets from each area. In Figure 4.37b, we calculate the average threatening rating of tweets from each area. In Figure 4.37c, we group the tweets by their area names and threatening ratings to show the ratio of tweets with different ratings. These queries are simple examples for demonstrating the Data Analytics feature of the BAD system. With the support of BAD's persistent storage and SQL++ query language, BAD users could also create more complex analytical queries.



(a) Tweet count (b) Average threatening rating (c) Threatening ratio

Figure 4.37: Exemplary data analytics

The Visualization Panel in Figure 4.38 visualizes incoming tweets with their geo-locations. When a new tweet comes, we place a Twitter icon on the map with the tweet's location. Each tweet icon shows up for 30 seconds and then is removed from the map to reduce resource usage. Note that the tweet's record is not removed from the system when the Twitter icon is removed. All tweets are persisted in BAD storage and can be accessed by analytical queries at any time. Tweet icons on the map are clickable elements. Clicking a tweet icon on the map shows more information about the tweet in the Tweet Panel.

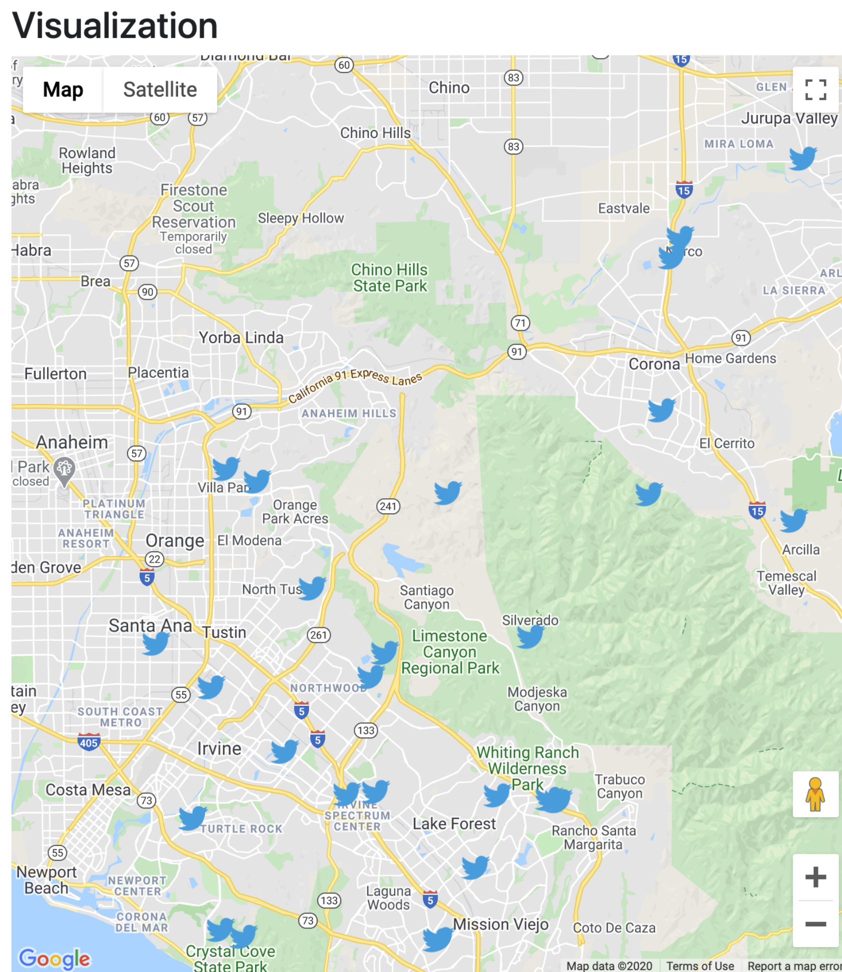


Figure 4.38: Visualization Panel of DHS dashboard

The Tweet Panel shows the content of the selected raw tweet on the map and its enriched version from BAD storage, as shown in Figure 4.39. We show the raw tweet and its enriched version in formatted JSON for better visualization. The enriched attributes in the Enriched

Tweet section are highlighted in red. Since JSON doesn't support all data types in ADM, we encode those ADM attributes as strings for demonstration purposes. The stored records are still in ADM format and contain attributes of various ADM types. The "timestamp" in the enriched tweet is a datetime attribute. The "location" is a point attribute, and the "user_registered_weapon" is an array attribute.

Tweets

Raw Tweet


```
{
  "tid": 1593279626502,
  "uid": 71,
  "area_name": "OC",
  "text": "Hank Schradaer sells Potato, and
Mike Ehrmantraut hides AR10, but Marie
Schrader sells AR10.",
  "coordinates": [
    33.61032194694224,
    -117.69039044985263
  ],
  "created_at": 1593279626502
}
```


Enriched Tweet


```
{
  "tid": 1593279626502,
  "uid": 71,
  "text": "Hank Schradaer sells Potato, and
Mike Ehrmantraut hides AR10, but Marie
Schrader sells AR10.",
  "timestamp": "2020-06-27T17:40:26.502Z",
  "location":
"point('33.61032194694224,-117.69039044985263')",
  "threatening_rating": "2",
  "user_registered_weapon":
"SKS,AK47,GLOCK21",
  "area_name": "OC",
  "coordinates": [
    33.61032194694224,
    -117.69039044985263
  ],
  "created_at": 1593279626502
}
```


Figure 4.39: Tweet Panel for demonstrating raw and enriched tweets' content

4.6.2 OCSD Dashboard

The overview of the OCSD dashboard is shown in Figure 4.40. It contains four panels: the Function Panel, Incoming Threatening Tweet Panel, Visualization Panel, and Officer Notification Panel. In order to visualize the threatening tweets in Orange County without constantly pulling from the system, this dashboard also subscribes to the threateningTweetsAt channel in DHS BAD but using a general broker for JSON formatted data, as ADM records cannot be parsed in the front-end system.

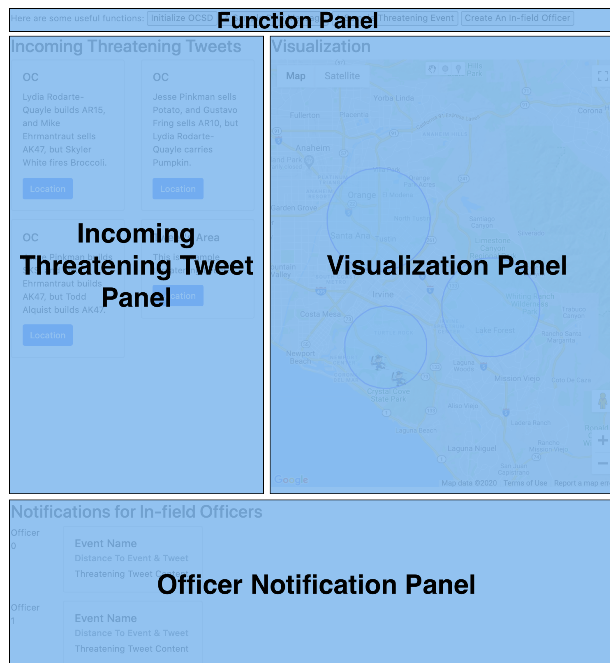


Figure 4.40: An overview of the OCSD dashboard

The Function Panel in Figure 4.41 contains several demonstration features. Users can use the "Initialize OCSD" function to initialize the OCSD BAD and connect to the DHS BAD. "A Sample DHS Message" and "A Sample Threatening Event" function generate sample threatening tweets and events, without getting data from DHS BAD, for demonstration users to see how the dashboard changes when new notifications arrive. "Create An In-field Officer" function places an in-field officer on the map who also subscribes to the ThreateningEventsNear channel in OCSD BAD.

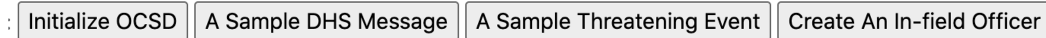


Figure 4.41: OCSD Function Panel

The Incoming Threatening Tweet Panel in Figure 4.42 shows the incoming threatening tweets received from DHS BAD. Since OCSD subscribes to threatening tweets from both Orange County and UC Irvine, demonstration users will see threatening tweets from both areas. When a new threatening tweet comes, it is added to the panel as a new card. The card contains the area name as the title and the tweet's text as the content. Clicking the "Location" button on the card will place a pin on the map to highlight the selected threatening tweet.

The Visualization Panel in Figure 4.43 is the core feature of the OCSD dashboard. It shows the incoming threatening tweets, local events, produced threatening events, and in-field officers' movements. The map contains a control bar at the top (highlighted in a blue box) for users to navigate the map, add a new event, and add a new in-field officer. A new event can be added by drawing a circle on the map indicating the event's area. An officer can be added by dropping an officer icon on a preferred location on the map. Information about the created events and officers is updated in the BAD system accordingly. A created officer moves around the map randomly, and it continuously sends his or her current location to the BAD system. Demonstration users can change an officer's location by dragging the officer's icon to a new place.

We use a red tweet icon to mark the threatening tweets received from DHS BAD and a black tweet icon for the threatening events detected by OCSD BAD. When an in-field officer receives a threatening event notification (as highlighted in the red circles in the figure), the officer randomly decides whether to go for further investigation or to stay at his or her current location. The officer's decision pops up as a small information window, as shown in the figure. If the officer decides to go, he or she moves gradually towards the tweet's

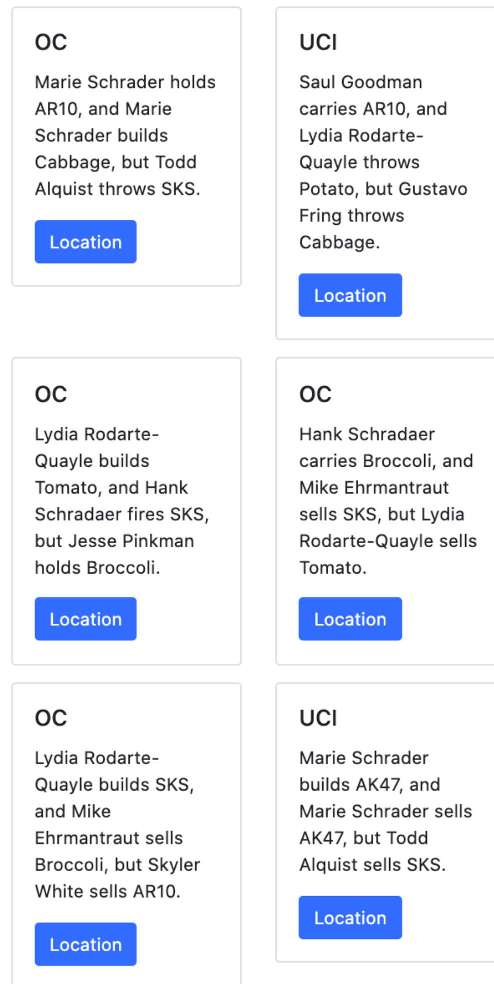


Figure 4.42: Incoming threatening tweets from DHS BAD

location, as the upper officer does in Figure 4.43.

The Officer Notification Panel in Figure 4.44 shows the threatening event notifications received by in-field officers from OCSD BAD. When a new in-field officer is added on the map, a new row is added to this panel. Each row shows the notification's content for the in-field officer. A threatening event notification is added as a new card to an officer's row. The card contains the event's name, the distances from the officer to the event and to the threatening event respectively, the tweet's text, the tweet's geolocation, and the event's geolocation.

Visualization

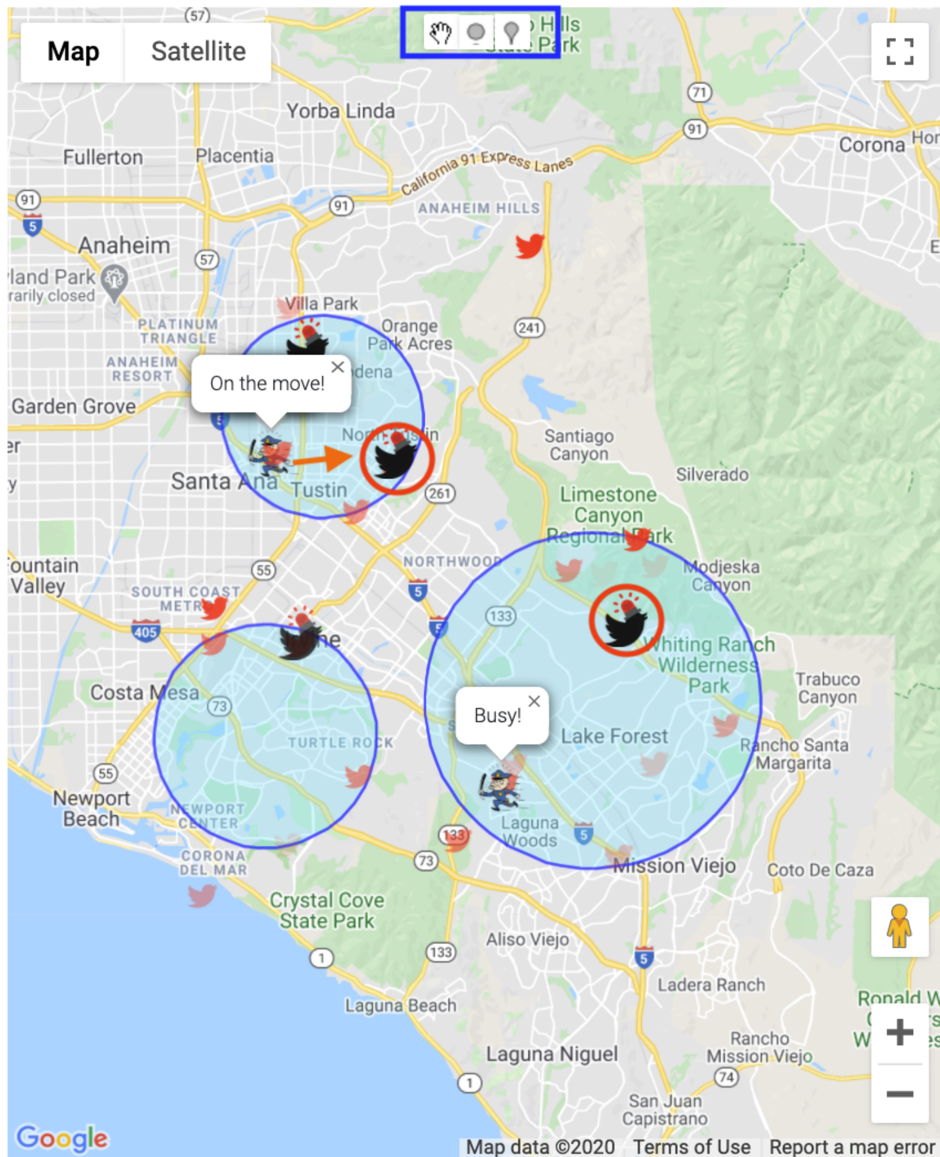


Figure 4.43: Visualization Panel of OCSD dashboard

Notifications for In-field Officers

Officer
0

OC Music Festival
EDist.: 9.88 TDist: 9.44
Skyler White sells Cabbage, and Gustavo Fring throws Potato, but Jesse Pinkman sells Grenade.
ELoc.: [33.77, -117.82], TLoc.: [33.76, -117.81]

OC Marathon
EDist.: 3.93 TDist: 3.56
Lydia Rodarte-Quayle fires Pumpkin, and Lydia Rodarte-Quayle builds AR10, but Todd Alquist sells AR15.
ELoc.: [33.65, -117.85], TLoc.: [33.68, -117.83]

Figure 4.44: Notification Panel for displaying threatening event notifications to officers

4.6.3 UCI Dashboard

An overview of the UCI dashboard is shown in Figure 4.45. It contains three panels: the Function Panel, Alert Panel, and Visualization Panel. Like the OCSD Dashboard, the UCI Dashboard also subscribes to the threateningTweetsAt channel in DHS BAD on a general broker for visualization purposes.

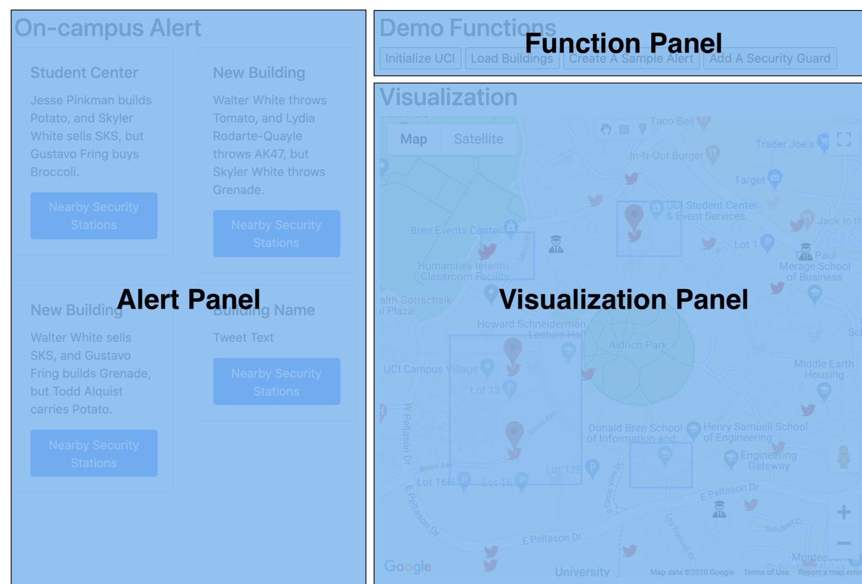


Figure 4.45: An overview of the UCI dashboard

The Function Panel in Figure 4.46 contains several demonstration functions for users to interact with. The "Initialize UCI" function initializes the UCI BAD and connects to the DHS BAD channel. The "Load Buildings" function inserts several UCI buildings into the system. The "Create A Sample Alert" creates a sample alert on the UCI campus without getting threatening tweets from DHS BAD. Finally, the "Add A Security Guard" function creates a new security guard station at a random on-campus location.

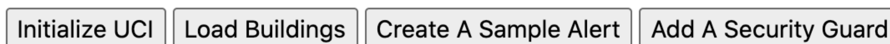


Figure 4.46: Demo Functions for UCI BAD

The Alert Panel in Figure 4.47 shows the on-campus alert generated based on threatening tweets. When an alert is created, it is added to the panel as a new card. The card contains the name of the building where a threatening tweet is posted, the threatening tweet's content, and a button "Nearby Security Stations". Clicking on the button of an alert shows security guard stations' information of the alert, as shown in Figure 4.26. It contains the stations and their geo-locations ordered by their distances to the threatening tweet.

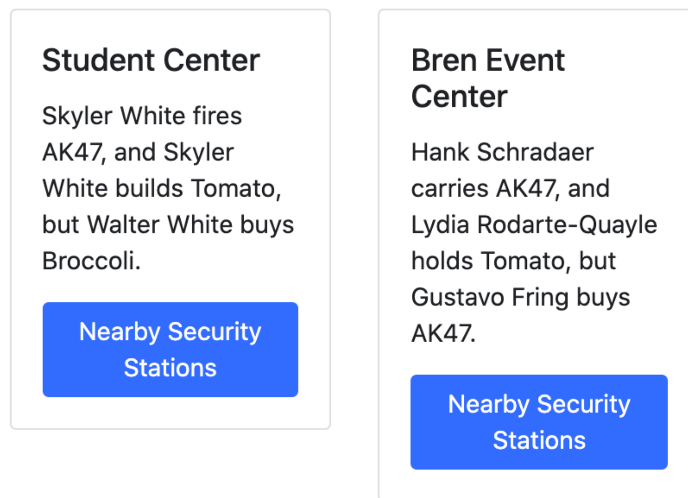


Figure 4.47: On-campus alerts

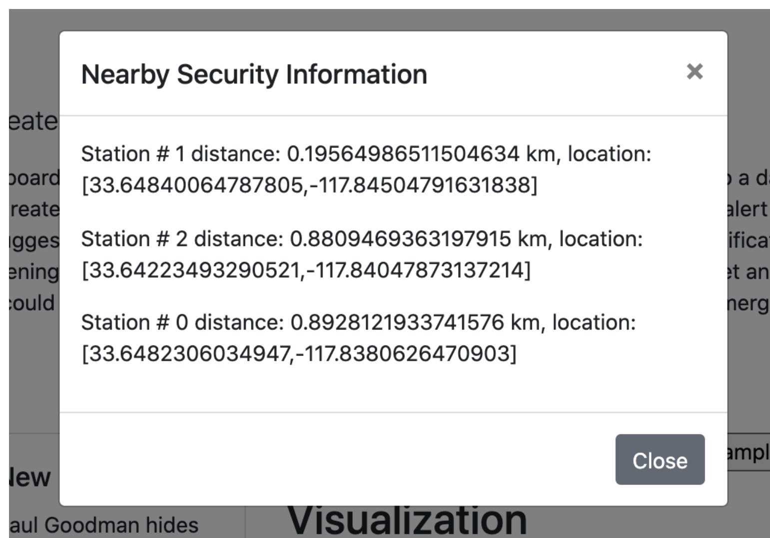


Figure 4.48: Security information of an alert

The Visualization Panel in Figure 4.49 visualizes the incoming threatening tweets from DHS BAD, security stations, buildings, and on-campus alerts generated by UCI BAD. The control bar on the top of the map (highlighted in a blue box) can be used to navigate the map, add new buildings, and add new security stations. Demonstration users can draw a rectangle for adding a new building on the map or placing a new security guard station at a preferred location on the map. Correspondingly, any added buildings and security guard stations are updated in the BAD system as well. Threatening tweets are marked using red tweet icons. When an on-campus alert is generated, we place a pin at the threatening tweet's location. Security guard stations are marked using an officer icon. Unlike in-field officers in the OCSD dashboard, security guard stations are not moving, but demonstration users can relocate them. Their new locations are updated in the back-end system accordingly.

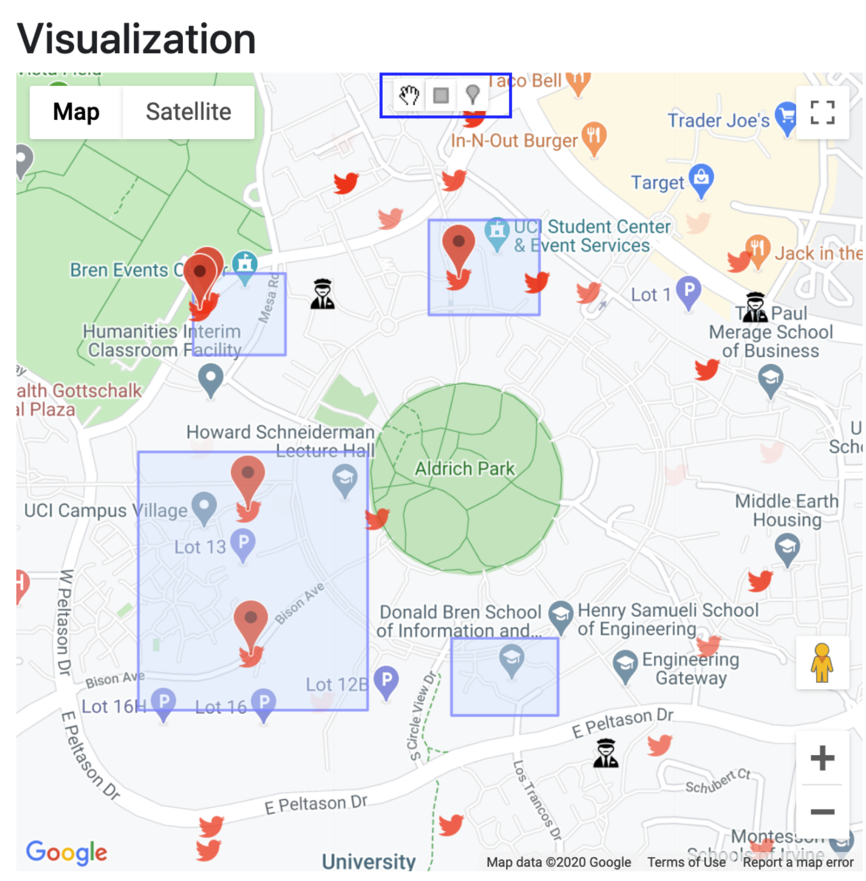


Figure 4.49: Visualization Panel of UCI BAD

4.7 Summary

In this chapter, we have focused on enabling users to declaratively create scalable data sharing services between different BAD systems. We looked at an example use case in which two local organizations (OCSD and UCI) would like to get data from a third organization (DHS) in order to provide BAD services to their subscribers. We discussed several possible ways of supporting this use case and proposed using data feeds and data channels for bridging BAD systems. We extended the BAD system with *BAD brokers* to enhance data exchanges between channels and feeds and *BAD feeds* to help users more easily create bridges between different BAD systems. We detailed a three-island prototype to show how BAD islands can be bridged together. We demonstrated how users can easily build such a prototype with declarative statements, and we used a sample incoming data item to show how data and events flow within the three-island system. We created a runnable demonstration system based on this three-island prototype to concretely illustrate how BAD islands share data with each other and support BAD applications with localized information. By utilizing the dynamic ingestion framework from Chapter 2 and the continuous channels from Chapter 3, we showcased how BAD islands can help developers to declaratively create multiple end-to-end active pipelines for supporting BAD applications at scale without having to glue and manage multiple Big Data systems.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we have studied different challenges related to activating Big Data at scale and we introduced techniques to enable users to actively enrich, to continuously subscribe to, and to declaratively share Big Data, all without having to glue, configure, and manage multiple Big Data systems.

In Chapter 3, we discussed the need to enrich data during ingestion and the challenges posed by directly attaching stateful enrichment functions to an ingestion pipeline. In order to support stateful enrichment functions, to allow users to attach any SQL++ queries onto an ingestion pipeline, and to enable enrichment functions to capture updates from referenced data, we introduced a new dynamic ingestion framework into AsterixDB, a framework that decouples the ingestion pipeline into different jobs (intake, computing, and storage) and executes the computing job repeatedly to support stateful computation and observes reference information updates. We examined the ingestion performance of the new framework and designed multiple data enrichment use cases to evaluate its enrichment performance at

different scales.

In Chapter 4, we defined Big Active Data (BAD) and described the challenges in providing active data services to BAD users in a BAD world. In order to support BAD applications, we proposed a BAD system consisting of the ingestion facility, persistent storage, analytical engine, data channels, and broker network. We reviewed the initial prototype of the BAD system - BAD-RQ - using a BAD use case, illustrated how BAD-RQ works, and explained in detail how BAD-RQ supports subscribing to Big Data at scale. We discussed issues related to using BAD-RQ for approximating continuous query semantics, and we introduced BAD-CQ for properly supporting continuous queries. We described the implementation details of BAD-CQ for providing continuous query semantics and the optimization techniques employed for improving its computing efficiency. In order to show the complexity and challenges of supporting BAD applications, we presented a GOOD system created by gluing multiple Big Data systems together. We evaluated the performance of BAD-CQ using various use cases at different scales and compared that with the GOOD system.

In Chapter 5, we investigated a use case of the BAD system involving data sharing. We introduced a sharing architecture based on BAD islands, where an island is an independent BAD system operated by an organization where data needs to be shared between different BAD islands and combined with local information on each island to provide local BAD services to its users. We discussed several possible methods of enabling data sharing and proposed the use of data channels and data feeds to "bridge" BAD islands together. In order to improve the expressiveness and efficiency of BAD bridges and help users manage bridges' life-cycles, we introduced BAD brokers, which take ADM records from data channels, and BAD feeds, which connect to remote data channels automatically. We showed how to use these BAD brokers and BAD feeds to enable declarative data sharing at scale among multiple BAD islands, and we developed a runnable demonstration system to show the kinds of functionalities that can be supported by BAD islands.

5.2 Future Work

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

- **Exploiting shared computation among data channels:** In the current BAD system introduced in Chapter 3, data channel queries are processed, compiled, and optimized independently. While shared computation arises from evaluating the parameterized requests within a given channel together, more exploitation of sharing is possible. Similar to [25], we could analyze multiple data channel queries, split them into smaller parts, discover shared computations, and reuse intermediate results to improve channel performance by avoiding redundant computation.

- **Resource management and scheduling of channel executions:** Currently, every channel execution in the BAD system in Chapter 3 is scheduled independently based on its period. Each channel execution runs as an independent job in the analytical engine, and AsterixDB's internal resource manager manages the resource usage of all jobs running in the system. When there is resource contention, certain channel executions may be delayed and could cause a channel to terminate (since we require channel executions to finish within the given period to meet the channel's time requirement). Given different channel periods and users' quality of service requirements, it should be possible to develop a smarter scheduling strategy in which we allow more flexible channel execution schedules based on the available resources and obtain better resource utilization at the same time.

- **Stream computing over batches:** The dynamic ingestion framework introduced in Chapter 2 used a layered architecture to decouple the intake, computing, and storage components of the ingestion pipeline. The incoming data batches in the dynamic ingestion framework are like the *micro-batches* from Spark Structured Streaming. One could utilize the incoming data batches in the dynamic ingestion framework to build a

stream-like computing engine within AsterixDB to provide richer real-time computation. Since this computing engine would be integrated into AsterixDB, we could avoid the additional cost of pulling existing data from other systems that streaming engines suffer from. Currently, the data enrichment computations on the dynamic ingestion pipeline only involves incoming data from one data feed (stream). It could be extended to take data from multiple data feeds (streams) and issue time-based, batch-based, or event-based executions and thus to provide stream computations on multiple streams.

- **Enhancing channel-broker communication:** We created the BAD brokers in Chapter 4 to allow developers to specify more information about a broker to improve the richness of the data being sent to the broker. Currently, the BAD system treats a broker as a data endpoint and only delivers generated data to the broker and lets it disseminate the data to subscribers. One could add more APIs to the broker and specify them when creating the broker so that the BAD system could communicate with its brokers and share more information about the back-end system. The broker then could utilize such information to initiate better strategies for handling the incoming channel data and managing subscriptions.

Bibliography

[1] F. Abel, Q. Gao, G.-J. Houben, and K. Tao. Semantic enrichment of Twitter posts for user profile construction on the social web. In *Extended semantic web conference*, pages 375–389. Springer, 2011.

[2] A. Alexandrov, R. Bergmann, S. Ewen, J. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The stratosphere platform for Big Data analytics. *VLDB J.*, 23(6):939–964, 2014.

[3] W. Y. Alkowaileet, S. Alsubaiee, M. J. Carey, et al. End-to-end machine learning with Apache AsterixDB. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 6:1–6:10, 2018.

[4] S. Alsubaiee, Y. Altowim, H. Altwaijry, et al. AsterixDB: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

[5] S. Alsubaiee, A. Behm, V. R. Borkar, Z. Heilbron, Y. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in AsterixDB. *PVLDB*, 7(10):841–852, 2014.

[6] S. Alsubaiee, M. J. Carey, and C. Li. LSM-based storage and indexing: An old idea with timely benefits. In *Second International ACM Workshop on Managing and Mining Enriched Geo-Spatial Data, GeoRich@SIGMOD 2015, Melbourne, VIC, Australia, May 31, 2015*, pages 1–6, 2015.

[7] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K. Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 53–64. Morgan Kaufmann, 2000.

[8] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[9] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. Technical Report 2, 2006.

[10] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative API for real-time applications in apache spark. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 601–613. ACM, 2018.

[11] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. K. III, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[12] M. P. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. B. Zdonik. The object-oriented database system manifesto. In H. Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, page 395. ACM Press, 1990.

[13] R. Barber, M. Huras, G. M. Lohman, C. Mohan, R. Müller, F. Özcan, H. Pirahesh, V. Raman, R. Sidle, O. Sidorkin, A. J. Storm, Y. Tian, and P. Tözün. Wildfire: Concurrent blazing data ingest and analytics. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2077–2080, 2016.

[14] S. Bharadwaj, L. Chiticariu, M. Danilevsky, S. Dhingra, S. Divekar, A. Carreno-Fuentes, H. Gupta, N. Gupta, S. Han, M. A. Hernández, H. Ho, P. Jain, S. Joshi, H. Karanam, S. Krishnan, R. Krishnamurthy, Y. Li, S. Manivannan, A. R. Mittal, F. Ozcan, A. Quamar, P. Raman, D. Saha, K. Sankaranarayanan, J. Sen, P. Sen, S. Vaithyanathan, M. Vasa, H. Wang, and H. Zhu. Creation and interaction with large-scale domain-specific knowledge bases. *PVLDB*, 10(12):1965–1968, 2017.

[15] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 1151–1162, 2011.

[16] I. Botan, Y. Cho, R. Derakhshan, N. Dindar, L. Haas, K. Kim, and N. Tatbul. Federated stream processing support for real-time business intelligence applications. In *International Workshop on Business Intelligence for the Real-Time Enterprise*, pages 14–31. Springer, 2009.

[17] R. M. Bruckner, B. List, and J. Schiefer. Striving towards near real-time data integration for data warehouses. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 317–326. Springer, 2002.

[18] R. Bryant, R. H. Katz, and E. D. Lazowska. "big-data computing": creating revolutionary breakthroughs in commerce, science and society, 2008.

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

[20] M. J. Carey, S. Jacobs, and V. J. Tsotras. Breaking BAD: a data serving vision for big active data. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016*, pages 181–186, 2016.

[21] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[22] M. Castro, P. Druschel, A. Kermarrec, and A. I. T. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE J. Sel. Areas Commun.*, 20(8):1489–1499, 2002.

[23] D. Chamberlin. *SQL++ For SQL Users: A Tutorial.* Couchbase, Inc., 2018. (Available at Amazon.com).

[24] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.

[25] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 379–390, 2000.

[26] G. V. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication. In H. Jacobsen, G. Mühl, and M. A. Jaeger, editors, *Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems, DEBS 2007, Toronto, Ontario, Canada, June 20-22, 2007*, volume 233 of *ACM International Conference Proceeding Series*, pages 14–25. ACM, 2007.

[27] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[28] K. Conroy and M. Roantree. Enrichment of raw sensor data to enable high-level queries. In *International Conference on Database and Expert Systems Applications*, pages 462–469. Springer, 2010.

[29] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In C. Thekkath and A. Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 251–264. USENIX Association, 2012.

[30] I. D. Corporation. The digital universe of opportunities: Rich data and the increasing value of the internet of things, 2014. [Online; accessed Jul-4th-2020].

[31] J. De Heinzelin. Ishango. *Scientific American*, 206(6):105–118, 1962.

[32] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In R. Agrawal and K. R. Dittrich, editors, *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 341–342. IEEE Computer Society, 2002.

[33] A. Doyle, G. Katz, K. Summers, C. Ackermann, I. Zavorin, Z. Lim, S. Muthiah, P. Butler, N. Self, L. Zhao, et al. Forecasting significant societal events using the embers streaming predictive analytics system. *Big Data*, 2(4):185–195, 2014.

[34] L. Duan and Y. Xiong. Big Data analytics and business analytics. *Journal of Management Analytics*, 2(1):1–21, 2015.

[35] Economist. Data, data everywhere, 2010. [Online; accessed Jul-4th-2020].

[36] P. Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst.*, 29(1):6, 2007.

[37] P. T. Eugster, P. Felber, R. Guerraoui, et al. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[38] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[39] S. Girdzijauskas, G. V. Chockler, Y. Vigfusson, Y. Tock, and R. Melamed. Magnet: practical subscription clustering for internet-scale publish/subscribe. In J. Bacon, P. R. Pietzuch, J. Sventek, and U. Çetintemel, editors, *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010*, pages 172–183. ACM, 2010.

[40] H.-P. Grahsl. Kafka connect MongoDB sink, 2016. [Online; accessed 23-December-2018].

[41] R. Grover and M. J. Carey. Data ingestion in AsterixDB. In *EDBT Conf.*, 2015.

[42] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993.*, pages 267–276, 1993.

[43] S. Jacobs, M. Y. S. Uddin, M. J. Carey, et al. A BAD demonstration: Towards big active data. *PVLDB*, 10(12):1941–1944, 2017.

[44] S. Jacobs, X. Wang, M. J. Carey, V. J. Tsotras, and M. Y. S. Uddin. Bad to the bone: Big active data at its core. *arXiv preprint arXiv:2002.09755*, 2020.

[45] S. Jacobs, X. Wang, M. J. Carey, V. J. Tsotras, and M. Y. S. Uddin. Bad to the bone: Big active data at its core. *VLDB J.*, 2020.

[46] C. S. Jensen, T. B. Pedersen, and C. Thomsen. Multidimensional databases and data warehousing. *Synthesis Lectures on Data Management*, 2(1):1–111, 2010.

[47] K. Johnston, J. M. Ver Hoef, K. Krivoruchko, and N. Lucas. *Using ArcGIS geostatistical analyst*, volume 380. Esri Redlands, 2001.

[48] A. Kafka. Kafka streams, 2020. [Online; accessed May-8th-2020].

[49] E. D. Knapp and J. T. Langill. *Industrial Network Security (Second Edition)*. Syngress, Boston, 2015.

[50] B. R. Konsynski and F. W. McFarlan. Information partnerships–shared data, shared scale. *Harvard Business Review*, 68(5):114–120, 1990.

[51] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

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

[53] M. Li, F. Ye, M. Kim, H. Chen, and H. Lei. A scalable and elastic publish/subscribe service. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*, pages 1254–1265. IEEE, 2011.

[54] V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch, and M. Wallrath. Design and implementation of an extensible database management system supporting user defined data types and functions. In *Fourteenth International Conference on Very Large Data Bases, August 29 - September 1, 1988, Los Angeles, California, USA, Proceedings.*, pages 294–305, 1988.

[55] C. Luo and M. J. Carey. Efficient data ingestion and query processing for LSM-based storage systems. *PVLDB*, 12(5):531–543, 2019.

[56] S. Mansmann, N. U. Rehman, A. Weiler, and M. H. Scholl. Discovering OLAP dimensions in semi-structured data. *Information Systems*, 44:120–133, 2014.

[57] J. Meehan, C. Aslantas, S. Zdonik, N. Tatbul, and J. Du. Data ingestion for the connected world. In *CIDR*, 2017.

[58] W. K. Michener, S. Allard, A. E. Budden, R. B. Cook, K. Douglass, M. Frame, S. Kelling, R. Koskela, C. Tenopir, and D. Vieglais. Participatory design of dataone - enabling cyberinfrastructure for the biological and environmental sciences. *Ecol. Informatics*, 11:5–15, 2012.

[59] J. J. Miller. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, volume 2324, 2013.

[60] MongoDB. Mongodb, 2020. [Online; accessed Jul-4th-2020].

[61] MongoDB. Mongodb connector for spark, 2020. [Online; accessed May-8th-2020].

[62] MongoDB. Mongodb kafka connector, 2020. [Online; accessed May-8th-2020].

[63] A. Moraru and D. Mladenić. A framework for semantic enrichment of sensor data. *Journal of computing and information technology*, 20(3):167–173, 2012.

[64] A. Morgan. MongoDB & data streaming – implementing a MongoDB Kafka consumer, 2016. [Online; accessed 23-December-2018].

[65] C. H. Museum. Timeline of computer history - storage memory. [Online; accessed Jul-4th-2020].

[66] H. Nguyen, M. Y. S. Uddin, and N. Venkatasubramanian. Multistage adaptive load balancing for big active data publish subscribe systems. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, DEBS 2019, Darmstadt, Germany, June 24-28, 2019*, pages 43–54. ACM, 2019.

[67] C. Olston, B. Reed, U. Srivastava, et al. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1099–1110, 2008.

[68] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ query language: Configurable, unifying and semi-structured. *arXiv preprint arXiv:1405.3631*, 2014.

[69] F. Özcan, Y. Tian, and P. Tözün. Hybrid transactional/analytical processing: A survey. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1771–1775, 2017.

[70] N. W. Paton and O. Díaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, 1999.

[71] S. Qanbari, N. Behinaein, R. Rahimzadeh, and S. Dustdar. Gatica: Linked sensed data enrichment and analytics middleware for IoT gateways. In *2015 3rd International Conference on Future Internet of Things and Cloud*, pages 38–43. IEEE, 2015.

[72] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[73] A. Rheinländer, U. Leser, and G. Graefe. Optimization of complex dataflows with user-defined functions. *ACM Comput. Surv.*, 50(3):38:1–38:39, 2017.

[74] R. Rice. DISC-UK datashare project. In *IASSIST 2008 - Technology of Data: Collection, Communication, Access and Preservation, Stanford, CA, USA, May 28-30, 2008.* IASSIST, 2008.

[75] D. R.-J. G.-J. Rydning. The digitization of the world from edge to core. *Framingham: International Data Corporation*, 2018.

[76] E. Scaria, A. Berghmans, M. Pont, C. Arnaut, and S. Leconte. Study on data sharing between companies in europe. *A study prepared for the European Commission Directorate-General for Communications Networks, Content and Technology by everis Benelux*, 24, 2018.

[77] S. E. Schoenherr. The digital revolution, 2004. [Online; accessed Jul-4th-2020].

[78] V. Setty, M. van Steen, R. Vitenberg, and S. Voulgaris. Poldercast: Fast, robust, and scalable architecture for P2P topic-based pub/sub. In P. Narasimhan and P. Triantafillou, editors, *Middleware 2012 - ACM/IFIP/USENIX 13th International Middleware Conference, Montreal, QC, Canada, December 3-7, 2012. Proceedings*, volume 7662 of *Lecture Notes in Computer Science*, pages 271–291. Springer, 2012.

[79] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986.

[80] K. Shvachko, H. Kuang, S. Radia, et al. The Hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10, 2010.

[81] R. T. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, 1986.

[82] A. Spark. Spark streaming + kafka integration guide, 2020. [Online; accessed May-8th-2020].

[83] A. Spark. Structured streaming programming guide, 2020. [Online; accessed April-13th-2020].

[84] M. Stonebraker and L. A. Rowe. The design of Postgres. In C. Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986*, pages 340–355. ACM Press, 1986.

[85] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Trans. Knowl. Data Eng.*, 2(1):125–142, 1990.

[86] R. W. Taylor and R. L. Frank. CODASYL data-base management systems. *ACM Comput. Surv.*, 8(1):67–103, 1976.

[87] C. Tenopir, S. Allard, K. Douglass, A. U. Aydinoglu, L. Wu, E. Read, M. Manoff, and M. Frame. Data sharing by scientists: practices and perceptions. *PloS one*, 6(6):e21101, 2011.

[88] D. B. Terry, D. Goldberg, D. A. Nichols, et al. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992*, pages 321–330, 1992.

[89] A. Thusoo, J. S. Sarma, N. Jain, et al. Hive - A warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[90] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy. Storm@twitter. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 147–156. ACM, 2014.

[91] M. Y. S. Uddin and N. Venkatasubramanian. Edge caching for enriched notifications delivery in big active data. In *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*, pages 696–705, 2018.

[92] P. Vassiliadis. A survey of extract-transform-load technology. *IJDWM*, 5(3):1–27, 2009.

[93] X. Wang and M. J. Carey. An IDEA: an ingestion framework for data enrichment in AsterixDB. *PVLDB*, 12(11):1485–1498, 2019.

[94] H. J. Watson. Tutorial: Big Data analytics: Concepts, technologies, and applications. *CAIS*, 34:65, 2014.

[95] J. Webber. A programmatic introduction to neo4j. In G. T. Leavens, editor, *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25, 2012*, pages 217–218. ACM, 2012.

[96] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.

[97] Wikipedia. Digital revolution. [Online; accessed Jul-4th-2020].

[98] S. Wolfert. Study on data sharing between companies in europe, 2018. [Online; accessed Jul-12th-2020].

[99] M. Zaharia, M. Chowdhury, T. Das, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012.

[100] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In M. Kaminsky and M. Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 423–438. ACM, 2013.

Appendix A

Enrichment Functions

A.1 Safety Rating

In statements shown in Figure A.1, we enrich a tweet with the safety rating of the country where the tweet comes from.

```
CREATE TYPE SafetyRatingType AS open {
  country_code : string,
  safety_rating: string
};

CREATE DATASET SafetyRatings(SafetyRatingType)
  PRIMARY KEY country_code;

CREATE FUNCTION enrichTweetQ1(t) {
  LET safety_rating = (SELECT VALUE s.safety_rating
    FROM SafetyRatings s WHERE t.country = s.country_code)
  SELECT t.*, safety_rating
};
```

Figure A.1: Safety rating enrichment

A.2 Religious Population

In statements shown in Figure A.2, we enrich an incoming tweet with the total religious population in its country.

```
CREATE TYPE ReligiousPopulationType AS open {
  rid : string,
  country_name : string,
  religion_name : string,
  population: int
};
CREATE DATASET ReligiousPopulations(ReligiousPopulationType)
  PRIMARY KEY rid;

CREATE FUNCTION enrichTweetQ2(t) {
  LET religious_population =
    (SELECT sum(r.population) FROM ReligiousPopulations r
    WHERE r.country_name = t.country)[0]
  SELECT t.*, religious_population
};
```

Figure A.2: Religious population enrichment

A.3 Largest Religions

In statements shown in Figure A.3, we enrich a tweet with the 3 largest religions in the country that the tweet came from.

```
CREATE TYPE ReligiousPopulationType AS open {
  rid : string,
  country_name : string,
  religion_name : string,
  population: int
};
CREATE DATASET ReligiousPopulations
(ReligiousPopulationType) PRIMARY KEY rid;

CREATE FUNCTION enrichTweetQ3(t) {
  LET largest_religions =
    (SELECT VALUE r.religion_name FROM ReligiousPopulations r
    WHERE r.country_name = t.country ORDER BY r.population LIMIT 3)
  SELECT t.*, largest_religions
};
```

Figure A.3: Largest religions enrichment

A.4 Fuzzy Suspects

In statements shown in Figure A.4 and Figure A.5, we use a Java UDF to remove the special characters in a Twitter user's screen name and find the related suspects whose name's edit distance to the processed screen name is within five characters.

```
...
@Override
public void evaluate(IFunctionHelper functionHelper)
    throws Exception {
    JString originalName = (JString) functionHelper.getArgument(0);
    String cleanedString = originalName.getValue().
        replaceAll("[^a-zA-Z]+", "").toLowerCase();
    cleanedName = (JString) functionHelper.getResultObject();
    cleanedName.setValue(cleanedString);
    functionHelper.setResult(cleanedName);
}
...
```

Figure A.4: Java UDF for removing special characters

```
CREATE FUNCTION annotateTweetQ4(x) {
  LET related_suspects=(
    SELECT s.sensitiveName, s.religionName
    FROM SensitiveNamesDataset s
    WHERE edit_distance(testlib#removeSpecial(x.user.screen_name),
      s.sensitiveName) < 5)
  SELECT x.*, related_suspects
};
```

Figure A.5: Fuzzy Suspects

A.5 Nearby Monuments

In statements shown in Figure A.6, we enrich an incoming tweet with the monuments that are within 1.5 degree of the tweet's location.

```
CREATE TYPE monumentType AS open {
  monument_id: string,
  monument_location: point
};
CREATE DATASET monumentList(monumentType)
  PRIMARY KEY monument_id;

CREATE FUNCTION enrichTweetQ4(t) {
  LET nearby_monuments =
    (SELECT VALUE m.monument_id FROM monumentList m
    WHERE spatial_intersect(m.monument_location,
      create_circle(create_point(t.latitude, t.longitude), 1.5)))
  SELECT t.*, nearby_monuments
};
```

Figure A.6: Nearby monuments enrichment

A.6 Suspicious Names

In statements shown in Figure A.7, we enrich a tweet with the number of nearby facilities grouped by their types, three closest religious buildings within three degrees of the tweet's location, and information about suspicious users who have the same name as the tweet's author.

```
CREATE TYPE ReligiousBuildingType AS open {
  religious_building_id : string,
  religion_name : string,
  building_location : point,
  registered_believer: int
};
CREATE DATASET ReligiousBuildings(ReligiousBuildingType)
  PRIMARY KEY religious_building_id;

CREATE TYPE FacilityType AS open {
  facility_id: string,
  facility_location: point,
  facility_type: string
};
CREATE DATASET Facilities(FacilityType) PRIMARY KEY facility_id;

CREATE TYPE SuspiciousNamesType AS open {
  suspicious_name_id: string,
  suspicious_name: string,
  religion_name: string,
  threat_level: int
};
CREATE DATASET SuspiciousNames(SuspiciousNamesType)
  PRIMARY KEY suspicious_name_id;

CREATE FUNCTION enrichTweetQ5(t) {
  LET nearby_facilities = (
    SELECT f.facility_type FacilityType, count(*) AS Cnt
    FROM Facilities f
    WHERE spatial_intersect(create_point(t.latitude, t.longitude),
      create_circle(f.facility_location, 3.0))
    GROUP BY f.facility_type),
  nearby_religious_buildings = (
    SELECT r.religious_building_id religious_building_id,
      r.religion_name religion_name
    FROM ReligiousBuildings r
    WHERE spatial_intersect(create_point(t.latitude, t.longitude),
      create_circle(r.building_location, 3.0))
    ORDER BY spatial_distance(create_point(t.latitude, t.longitude),
      r.building_location) LIMIT 3),
  suspicious_users_info = (
    SELECT s.suspicious_name_id suspect_id,
      s.religion_name AS religion, s.threat_level AS threat_level
    FROM SuspiciousNames s
    WHERE s.suspicious_name = t.user.name)
  SELECT t.*, nearby_facilities, nearby_religious_buildings,
    suspicious_users_info
};
```

Figure A.7: Enrich a tweet with nearby facilities and suspicious user information

A.7 Tweet Context

In statements shown in Figure A.8, we enrich a tweet with the average income of the district where the tweet is posted, the number of facilities in this district grouped by their types, and the ethnicity distribution of the residents in this district based on a resident sampling.

```
CREATE TYPE DistrictAreaType AS open {
  district_area_id : string,
  district_area : rectangle
};
CREATE DATASET DistrictAreas(DistrictAreaType)
  PRIMARY KEY district_area_id;

CREATE TYPE FacilityType AS open {
  facility_id: string,
  facility_location: point,
  facility_type: string
};
CREATE DATASET Facilities(FacilityType) PRIMARY KEY facility_id;

CREATE TYPE AverageIncomeType AS open {
  district_area_id: string,
  average_income: double
};
CREATE DATASET AverageIncomes(AverageIncomeType)
  PRIMARY KEY district_area_id;

CREATE TYPE PersonType AS open {
  person_id: string,
  ethnicity: string,
  location: point
};
CREATE DATASET Persons(PersonType) PRIMARY KEY person_id;

CREATE FUNCTION enrichTweetQ6(t) {
  LET area_avg_income = (
    SELECT VALUE a.average_income
    FROM AverageIncomes a, DistrictAreas d1
    WHERE a.district_area_id = d1.district_area_id
      AND spatial_intersect(create_point(t.latitude, t.longitude), d1.district_area)),
  area_facilities = (
    SELECT f.facility_type, count(*) AS Cnt
    FROM Facilities f, DistrictAreas d2
    WHERE spatial_intersect(f.facility_location, d2.district_area)
      AND spatial_intersect(create_point(t.latitude, t.longitude), d2.district_area)
    GROUP BY f.facility_type),
  ethnicity_dist = (
    SELECT ethnicity, count(*) AS EthnicityPopulation
    FROM Persons p, DistrictAreas d3
    WHERE spatial_intersect(create_point(t.latitude, t.longitude), d3.district_area)
      AND  spatial_intersect(p.location, d3.district_area)
    GROUP BY p.ethnicity AS ethnicity)
  SELECT t.*, area_avg_income, area_facilities, ethnicity_dist
};
```

Figure A.8: Enrich a tweet with the average income, facility numbers, and ethnicity distribution in the area where the tweet is posted

A.8 Worrisome Tweets

In statements shown in Figure A.9, we enrich a incoming tweet with the names of religions within three degrees of the tweet's location and the number of terrorist attacks in the past two months related to that religion.

```
CREATE TYPE ReligiousBuildingType AS open {
  religious_building_id : string,
  religion_name : string,
  building_location : point,
  registered_believer: int
};
CREATE DATASET ReligiousBuildings(ReligiousBuildingType)
  PRIMARY KEY religious_building_id;

CREATE TYPE AttackEventsType AS open {
  attack_record_id: string,
  attack_datetime: datetime,
  attack_location: point,
  related_religion: string
};
CREATE DATASET AttackEvents(AttackEventsType)
  PRIMARY KEY attack_record_id;

CREATE FUNCTION enrichTweetQ7(t) {
  LET nearby_religious_attacks = (
    SELECT r.religion_name AS religion, count(a.attack_record_id)
      AS attack_num
    FROM ReligiousBuildings r, AttackEvents a
    WHERE spatial_intersect(create_point(t.latitude, t.longitude),
      create_circle(r.building_location, 3.0))
      AND t.created_at  < a.attack_datetime + duration("P2M")
      AND t.created_at  > a.attack_datetime
      AND r.religion_name = a.related_religion
    GROUP BY r.religion_name)
  SELECT t.*, nearby_religious_attacks
};
```

Figure A.9: Enrich a tweet with nearby religions and the recent terrorist attacks related to them