UNIVERSITY OF CALIFORNIA,
IRVINE

Scalable Fault-Tolerant Elastic Data Ingestion in AsterixDB

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Raman Grover

Dissertation Committee:
Professor Michael J. Carey, Chair
Professor Chen Li
Professor Sharad Mehrotra

2015

DEDICATION

To my wonderful parents...

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my academic advisor – Prof. Michael J. Carey – who has been a source of inspiration throughout the course of my graduate studies. Working with Prof. Carey has had its positive impact on a lot of aspects that include writing skills, developing the right methodology for experimental evaluation, subsequent analysis of results and articulating thoughts in a clear manner. In the hindsight, Prof. Carey has been instrumental in teaching me the required skill of balancing between work and life and yet not miss any targets. I am deeply indebted to Prof Carey for his guidance.

I would like to thank Prof. Chen Li and Prof. Sharad Mehrotra for being on the dissertation committee. Prof. Li and Prof. Mehrotra introduced me to a wider range of real life scenarios that could benefit from the work done as part of my dissertation.

I have enjoyed being surrounded by talented and helpful colleagues and brainstorming sessions. As we worked towards a common goal of making AsterixDB robust, we developed mutual respect and bonding as a team. In particular, I would like to thank Pouria Pirzadeh and Yingyi Bu for being amazing office mates and extending help, cracking jokes, and having long technical discussions. I could not have asked more from such an amazing AsterixDB team.

This research is partially supported by the National I Science Foundation (NSF) IIS awards (0238586, 0331707, 0910989, 0844574 and 1030002), by University of California, Irvine, Department of Computer Science Chair's Fellowship, and by grants from Google, eBay, Facebook, Microsoft, Yahoo, HTC and Oracle.

I am extremely thankful to my friends in Irvine (from the IKG group) who have given me amazing moments to cherish and have been there to celebrate each small and big success.

My parents have played an immense role in shaping me and instilling in me the curiosity to learn. They have extended their unconditional love and care at all times. It would be impossible to sum up their contribution in text.

My stay at University of California, Irvine has gifted me great moments, friends and colleagues and most importantly a wonderful person, who would join me as a companion and be my better half, Payel. She has given me extreme care and support and has stood like a pillar to ensure a smooth ride for me. I feel extremely blessed to have found her.

CURRICULUM VITAE

Raman Grover

Doctor of Philosophy in Information and Computer Science **2015**
University of California, Irvine *Irvine, California*

Master of Science, Computer Science **2010**
University of California, Irvine *Irvine, California*

Bachelor of Technology, Computer Science **2004**
Indian Institute of Technology, Roorkee *Roorkee, India*

RESEARCH EXPERIENCE

Graduate Research Assistant **March, 2009–March, 2015**
University of California, Irvine *Irvine, California*

INDUSTRIAL EXPERIENCE

Research Intern **June, 2014–Sept, 2014**
Microsoft Research *Mountain View, California*

Research Scholar **June, 2012–Dec, 2012**
Walmart Labs *San Bruno, California*

Engineering Intern **June, 2012–Sept, 2012**
Facebook *Palo Alto, California*

HONORS and AWARDS

Google Fellowship in ICS **2015**
Yahoo! Key Scientific Challenge Award **2012**
SIGMOD Travel Grant **2012**
ICDE Travel Grant **2012**
Chair's Fellowship **2010-2012**

ABSTRACT OF THE DISSERTATION

Scalable Fault-Tolerant Elastic Data Ingestion in AsterixDB

By

Raman Grover

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2015

Professor Michael J. Carey, Chair

In this thesis, we develop the support for continuous data ingestion in AsterixDB, an open-source Big Data Management System (BDMS) that provides a platform for storage and analysis of large volumes of semi-structured data. Data feeds are a mechanism for having continuous data arrive into a BDMS from external sources and incrementally populate a persisted dataset and associated indexes. The need to persist and index "fast-flowing" high-velocity data (and support ad hoc analytical queries) is ubiquitous. However, the state of the art today involves 'gluing' together different systems. AsterixDB is different in being a unified system with "native support" for data ingestion.

We discuss the challenges and present the design and implementation of the concepts involved in modeling and managing data feeds in AsterixDB. AsterixDB allows the runtime behavior, allocation of resources, and the offered degree of robustness to be customized (by associating an ingestion policy) to suit the application(s) that wish to consume the ingested data. Results from experiments that evaluate the scalability and fault-tolerance of the AsterixDB data feeds facility are reported. We include an evaluation of the built-in ingestion policies and study their effect as well on throughput and latency. An evaluation and comparison with a 'glued' together system formed from popular engines — Storm (for streaming) and MongoDB (for persistence) — is also included.

Chapter 1

Introduction

A large volume of data is being generated on a "continuous" basis, be it in the form of click-streams, output from sensors, log files or via sharing on popular social websites [4]. Encouraged by low storage costs [43], data-driven enterprises today are aiming to collect and persist the available data and analyze it over time to extract hidden insightful information. Marketing departments use Twitter feeds to conduct sentiment analysis to determine what users are saying about the company's products. Location data combined with customer preference data from social networks enable retailers to target marketing campaigns based on buying history. Furthermore, utility companies have rolled out smart meters that measure the consumption of water, gas, and electricity and generate huge volumes of interval data that is required to be analyzed over time.

Data that is being generated at a high rate in a continuous fashion is typically referred to as *Fast Data* across academia and industry [26]. The act of collecting and persisting the data can be envisioned as producing a data repository that reaches the terabyte scale (*Big Data*) in due time and continues to grow beyond. Such a data repository is often classified as an example of *Big Fast Data* [34]. It is desired to subject the collected data to ad hoc queries that involve a mix of join, aggregation, group-by and sort operations. Such ad hoc analysis in a typical setting drives

overlying applications that generate reports and summaries, provide visualization over time, and facilitate complex tasks such as data mining and detecting anomalies. Such analyses go beyond the typical analysis done as part of "stream processing", which restricts analysis to smaller sets of data (e.g., a five minute window of data) and does not provide an opportunity for offline ad hoc processing.

The conventional way of doing ad hoc analysis over data is to use a data management system (database) and express the required analysis in a high-level language (SQL). However, traditional data management systems require data to be loaded and indexes to be created before data can be subjected to such ad hoc analysis. To keep pace with "fast-moving" high-velocity data, a Big Data Management System (BDMS) must be able to ingest and persist data on a continuous basis. A flow of data from an external source into persistent (indexed) storage inside a BDMS will be referred to here as a *data feed*. The task of maintaining the continuous flow of data is hereafter referred to as *data feed management*.

A simple way of having data being put into a Big Data Management System on a continuous basis is to have a single program (or process) fetch data from an external data source, parse the data, and then invoke an insert statement per record or per batch of records. This solution is limited to a single machine's computing capacity. Ingesting multiple data feeds would potentially require running and managing multiple individual programs/processes. The task of continuously retrieving data from external source(s), applying some pre-processing for cleansing, filtering or transforming the data may amount to 'gluing' together different systems (e.g., [44]). For the task at hand, it may suffice to employ a streaming engine that is efficient at routing data in a continuous fashion and use it in conjunction with a persistent data store that provides for the storage, indexing and querying of semi-structured data. A popular choice made within the open-source community is to use Storm [8] as a streaming engine coupled with MongoDB [7] as a data store.

A combination of individual systems that are otherwise efficient may not always prove to be optimal. (A thorough discussion with results from rigorous experimentation is deferred until Chapter 7

of this document.) As we will see later, it is harder to reason about the data consistency, scalability and fault-tolerance offered by such a 'glued' together assembly of individual systems. Traditional data management systems have evolved to provide native support for services if the service offered by an external system is inappropriate or may cause substantial overheads [37, 17]. Responding to the need of the hour, then, it is natural for a BDMS to provide "native" support for data feed management.

1.1 Challenges in Data Feed Management

We begin by enumerating the challenges involved in building a data ingestion facility and discussing on the desirable features of such a system.

1.1.1 Genericity and Extensibility

A data ingestion facility must be generic enough to work with a variety of data sources and data-driven applications. A tight-coupling with a data source or an application thereof is likely to result in a rigid system that is not adaptable to other scenarios or combinations of data sources and applications.

It is expected that a given data source may send data in its (proprietary) format that needs to be translated into a different format for data to be processed, stored and accessed within a data management system. The protocol for the transfer of data and the handshake and subsequent messages may be unique to a given data source. The transfer of data itself may occur in a push- or a pull-based manner. A push-based data source requires an initial handshake (message) that acts as a request for data and sets the necessary parameters (e.g. receiver's IP address) that are specific to the data transfer protocol. Data is then sent (pushed) at its regular rate thereafter without any subsequent requests. In contrast, a pull-based data source requires a separate request each time

(additional) data is required. A data source may be *durable* in the sense that it allows the receiver to selectively fetch a specific record. It may additionally offer a throttling mechanism that allows data to be sent at an adaptive rate in accordance with the rate at which data can be inserted into the data management system. The nature of the data source and the offered protocols/APIs have implications on how the flow of data can be managed. The properties of the data source, together with the set of APIs offered to facilitate data transfer, need to be taken into account when setting and managing the flow of data from the source to a data management system. A single strategy based on a set of assumptions about an external data source may not work or may be sub-optimal with respect to a given data source.

An application wishing to consume the fetched data may require it to be further processed, transformed, and persisted in a different form to suit the specific data analysis requirements of the application. Examples of such pre-processing include filtering of unwanted records or attributes, extraction of additional features for content-based classification, or performing sentiment analysis on text data. The application may further require QoS guarantees on the delivery of data and dictate the degree of robustness required in the handling of failures. As in the case of a data source, in order to avoid restricted usability, the properties or requirements of an application should not be assumed.

It is desired for a data ingestion facility to offer a plug-and-play model wherein individual modules can be custom-built and incorporated into the system to extend the offered functionality.

1.1.2 Fetch-Once Compute-Many Model

A data feed could simultaneously drive multiple applications that each require the arriving data to be processed/persisted differently. A naive way to achieve this is to independently fetch data for each application. Each such independent path would then need to perform the necessary step of translating the received data into the required format and optionally subjecting the translated

data to additional pre-processing prior to persistence. This strategy has multiple pitfalls. At first, it potentially employs additional resources (network bandwidth and CPU cycles) in setting and maintaining the flow of data across multiple paths between the external data source as the sender and the data management system as the receiver. Secondly, it leads to a loss of opportunity in optimizing the data flow wherein the common sequence of steps (e.g. translating the received records into a target format) is performed just once and subsequent pre-processing operations are built on top of each other to avoid any redundancy. Moreover, it causes additional overhead at the data source for disseminating data across multiple paths and/or handling of an increased number of requests (from multiple paths), particularly when the ingestion is pull-based.

It is desirable to have a single flow of data from an external source and yet be able to transform it in multiple ways to drive different applications concurrently.

1.1.3 Scalability and Elasticity

A data ingestion facility may involve multiple concurrent feeds each with their own demand for resources. A resource-constrained environment can offer a significant challenge in maintaining pace with an external data source that continues to generate data at its regular rate while the system load varies. In the case of a pull-based ingestion, the data management system can regulate the rate of arrival of data by limiting its requests for additional data. In contrast, push-based ingestion offers little or no control whatsoever on the rate of arrival of data. The inability to process data at its arrival rate translates to a potential data loss and/or a delay in persisting data. Such a scenario might be unacceptable for the overlying application. The uncertainty in the demand for resources can be further increased by the un-scheduled arrival of ad hoc queries that execute in parallel and compete for resources with concurrent data feeds.

It is hard to properly provision a system upfront prior to the initiation of data flow or the submission of queries. Provisioning a system requires the availability of known attributes related to each

data source that is sourcing a feed, a well-predicted rate of arrival of data, and a 'fixed' schedule of queries. Moreover, it is likely for a configured system to be either under-provisioned or be over-provisioned with respect to the workload at a given point in time. Given the possibility of subsequent addition of resources, a data ingestion facility must at a minimum, exhibit (linear) *scalability* in being able to handle an increasingly large number of data feeds by leveraging added resources.

A somewhat related aspect of a system is the ability to autonomously scale in or out with respect to the demand for resources; this is often referred to as *elasticity* of the system. An elastic system is particularly suitable in a cloud-based environment with a *pay-as-you-consume* model where relinquishing resources during lighter workloads can help save dollars.

Building a scalable data ingestion facility requires exploiting parallelism in managing the flow of data along a feed. As a pre-requisite for offering elasticity, the system must be able to monitor the demand and allocation of resources to detect any resource bottlenecks and form a corrective action by determining the level of resources required to handle the current workload.

1.1.4 Fault Tolerance

Systems in the Big Data era typically employ off-the-shelf commodity hardware as opposed to beefed up costly boutique hardware. Commodity computing, as it is widely known, offers the greatest amount of useful computation at low cost. Data ingestion is thus expected to run on a large cluster of commodity hardware. The down-side of using commodity hardware is the vulnerability to hardware failures, which become more probable given that data ingestion is a potentially never-ending activity. To emphasize the importance of handling and recovering from failures, consider the case of push-based ingestion wherein the external data source continues to send data irrespective of any failures that have occurred inside the data management system.

Failures can be categorized as *hard* and *soft* failures. The loss or crash of a physical node due to a network or disk failure is an example of a hard failure. Data ingestion involves translating and pre-processing the arriving data prior to persistence. A formatting error in the content or a runtime exception in processing a record (e.g., due to an expected or missing value for an attribute inside the record) may lead to runtime exceptions that are counted as soft failures. Data ingestion must be able to circumvent such *soft* failures and continue processing of arriving records.

It is worth noting that the desired degree of robustness in handling failures is dictated by the overlying application. Consider an application that intends to amass tweets to discover popular trends. Losing a small fraction of tweets may still be acceptable without producing erroneous results from analysis. In contrast, consider an application that aggregates clickstream data to compute hourly revenue contributed by clicks. The inability to process a record in such a scenario translates to loss of revenue or error(s) in reporting. It is worth noting that a single strategy to handling of failures can be a misfit or an overfit to the needs of the overlying application. It is desirable to offer the desired degree of robustness in handling failures i(in accordance with application requirements) while minimizing data loss.

1.2 Contributions

In this thesis, we describe the support for data feeds and data feed management in AsterixDB. AsterixDB is a Big Data Management System (BDMS) that provides a platform for the scalable storage and analysis of very large volumes of semi-structured data. We discuss the approach adopted to address the aforementioned challenges. The thesis offers the following contributions.

(1) *Concepts involved in Data Feed Management*: The thesis introduces the concepts involved in defining a data feed and managing the flow of data into a target dataset and/or to other dependent feeds to form a cascade network. It details the design and implementation of the involved concepts

in a complete system.

(2) *Policies for Data Feed Management*: The thesis describes how a data feed is managed by associating an ingestion policy that controls the system's runtime behavior in response to events such as software/hardware failures and resource bottlenecks. Users may also opt to provide a custom policy to suit special application requirements.

(3) *Scalable/Elastic Data Feed Management*: The thesis describes a dataflow approach that exploits partitioned-parallelism to scale and ingest increasingly large amounts of data. The dataflow exhibits elasticity by being able to monitor and dynamically re-structure itself to adapt to the rate of arrival of data. The system is fault-tolerant and provides *at least once semantics* as the strongest guarantee, if required.

(4) *Contribution to Open-Source*: The AsterixDB system, including data feeds, is available as open source [3, 2]. The support for data ingestion in AsterixDB is extensible to enable future contributors to provide custom implementations of different modules and form custom-designed policies to suit specific requirements.

(5) *Experimental Evaluation* The thesis provides an experimental evaluation that studies the role of different ingestion policies in determining the behavioral aspects of the system including the achieved throughput and latency. We also report on experiments that evaluate scalability and our approach to fault-tolerance.

(6) *Comparison with the State-of-the-Art*: We include an evaluation of a custom-created system created by coupling Storm (a popular streaming engine) and MongoDB (a popular persistence store) to draw a comparison with AsterixDB in terms of the flexibility and scalability achieved in data feed management. We demonstrate and analyze the inefficiencies involved in gluing together otherwise efficient systems, which is a widely followed practice in open-source community today.

1.3 Organization

The remainder of the thesis is organized as follows. We discuss related work in Chapter 2 and provide an overview of the AsterixDB and Hyracks systems in Chapter 3. Chapter 4 describes how a feed is modeled and defined at the language level in AsterixDB. The design aspects and implementation details involved in managing a data feed are then described in Chapter 5. Chapter 6 describes the support for handling failures. The mechanisms to deal with resource bottlenecks and sustaining the flow of data are described in Chapter 7. Chapter 8 covers some of the real-world use cases involving the use of data feeds. We conclude the thesis in Chapter 9.

Chapter 2

Related Work

In this chapter, we describe how the AsterixDB data ingestion facility compares with other systems that involve the handling and processing of high-velocity data and are faced with seemingly similar challenges. More detailed comparisons will be provided as appropriate in subsequent chapters. Related systems fall into several distinct categories – Stream Processing Engines (SPEs), Data Routing Engines, Extract Transform Load (ETL) systems.

2.1 Stream Processing Engines

Much work has been done on systems to support data stream processing [10, 12, 20, 24, 31]. On the surface, data feeds may seem similar to streams from the data streams literature. There are important differences, however. Data feeds are a "plumbing" concept; they are a mechanism for having data flow in from external sources that produce data continuously and to incrementally populate and persist it in a data management system. Stream Processing Engines (SPEs) do not persist data; instead they provide a sliding window on data (e.g., a 5 minute view of data), but the amount, or the time window, is usually limited by the velocity of the data and the available

memory. The windowing operations performed on in-flight data are different from the ad hoc analysis that is enabled by queries running over persisted data. The requirement that the data needs to be processed only once and is to be discarded thereafter drives the design and implementation aspects of a typical SPE. As such, an SPE does not deal with providing storage capabilities or transactional semantics with respect to maintaining indexes over such data.

With respect to providing fault-tolerance, an SPE faces the challenge of providing highly available parallel data-flows, SPE researchers have proposed several techniques (e.g., [36, 13]) for tackling them. These techniques rely on replication; the state of an operator is replicated on multiple servers and/or have multiple servers simultaneously process identical input streams (also referred to as *process-pairing*). Fault-tolerance is thus provided at a high cost as the number of required nodes are roughly doubled. Moreover, offering a single strategy (e.g. process-pairing) for fault-tolerance can be wasteful of resources in scenarios where the offered degree of robustness exceeds the requirements laid down by the overlying application that wishes to consume the arriving data. Such mechanisms are resource intensive and, as we shall describe later (Chapter 6) in detail, are not the preferred approach adopted in AsterixDB.

2.2 Data Routing Engines

Data routing engines can route, transform, and analyze a stream of data. Such systems do not provide a high-level language for expressing queries over the data, nor do they allow windowing operations similar to those supported by a stream processing engine. Prominent examples from the open-source community include Twitter's Storm [8], Yahoo!'s S4 [32] and LinkedIn's Samza [1]. A data routing engine typically allows constructing an arbitrary dataflow that resembles a directed acyclic graph (DAG) that consists of *nodes* connected together using *edges*, and it is typically constructed using APIs provided by the system. In such a dataflow, the nodes represent custom logic that is applied to data arriving at the node. The logic contained in a node is provided by the

end-user and is treated as a black box by the system. Edges represent the flow of data between a pair of nodes. A data routing engine such as Storm runs in a distributed manner on a cluster of commodity hardware. Storm also deals with the challenges involved in successfully moving data across the DAG in a fault-tolerant manner.

A typical use of a data routing engine (a use that overlaps with a data ingestion facility) is to receive data from an external source and have it processed through a data flow. Although a data routing engine does not provide for storage and indexing of data, it can still be used in conjunction with a data store (e.g., MongoDB) or a database (e.g., MySQL) such that the routed data output from the data routing engine can be re-directed to the data store (or a database) using its prescribed APIs. Such a setting then allows running ad hoc analytical queries over the persisted data, but with the underlying assumption that the data store can handle high-velocity data. On the surface, it seems that 'gluing' together a data routing engine with a data persistence store obviates the need for native support for data ingestion inside a data management system. However, building a combined system from otherwise efficient systems is not always optimal and does not offer a desirable end-user experience. Data ingestion is not treated as a first-class citizen in such a 'glued' together system. A detailed description of the pitfalls of 'glue' together with an experimental evaluation is deferred until Chapter 6.

2.3 Extract Transform Load (ETL) Systems

Extract Transform Load [40] (commonly referred as ETL) systems (e.g., [6, 11]) evolved naturally from the need to consolidate data residing across different homogeneous or heterogeneous data sources into a single common repository – a Data Warehouse. Data residing across systems may differ in representation, contained attributes, and format. Intuitively, ETL systems involve fetching data from these data sources (extraction), resolving any conflicts related to data representation and translating it into common format (transformation) and then putting the data into the data

warehouse (loading). A survey covering different ETL systems and the state-of-the-art can be found in [39].

ETL systems operate in a "batchy" mode, with a 'finite' amount of data transferred at periodic intervals coinciding with off-peak hours. Data the gets loaded via an ETL data flow is thus typically stale. The "batch" nature of such systems makes them inappropriate to the task at hand today - 'continuous' ingestion of data. Furthermore, ETL systems typically involve the use of high-end servers unlike a large cluster of commodity hardware, and are thus form an expensive solution. ETL tools are not known to exhibit linear scalability and their performance measured as rows inserted per second saturates at rates like 60k rows/second. As we are in the era of Big Data, ETL tools too have jumped on the bandwagon with an attempt to improve performance by employing MapReduce as the programming model and deploying a re-architect-ed ETL design on a cluster of commodity hardware. Xu et. al in [44] described a Map-Reduce based approach for populating a parallel database system with data arriving from an external source. However, the system formed a tight coupling with MapReduce and required the data to be initially put into the distributed file system (HDFS). Such an approach adds to storage costs and introduces additional delays due to the staging of data.

2.4 Other Systems

It is common in an industrial setting to have a large number of data sources that produce data in a continuous fashion. A typical example of continuously arriving data are the logs that are produced by web-servers and that need to be analyzed to detect anomalies or trigger alerts. Such logs are typically analyzed using batch-processing enabled by MapReduce. However, Hadoop (MapReduce) itself does not provide for the efficient and reliable collection of data from different sources. That limitation has resulted in the development of a multitude of systems (including Flume [21], Chukwa [19], Kafka [30] and Sqoop [29]) that support the reliable fault-tolerant collection of

data in a distributed/parallel manner and can scale the facility via the addition of resources. Such systems help bridge the gap between continuously arriving logs and their processing via Map-Reduce and avoid a repetition of effort in building a great deal of infrastructure to connect data sources with processing tools.

Such systems do not provide for the persistence, indexing, or ad hoc querying of continuously arriving data, but they do face a common subset of challenges in dealing with a wide variety of sources and constructing an arbitrary dataflow for movement of data. Next, we give a brief overview of these systems.

2.4.1 Flume

Flume [21] is a distributed system for efficiently collecting, aggregating and moving large amounts of data from many different sources to a centralized data store. As data sources in Flume are customizable, Flume can be used to transport massive quantities of event data (network traffic data or social media-generated data) from a variety of data sources. Flume is designed to be reliable and highly available while providing a simple, flexible, and intuitive programming model based on streaming data flows.

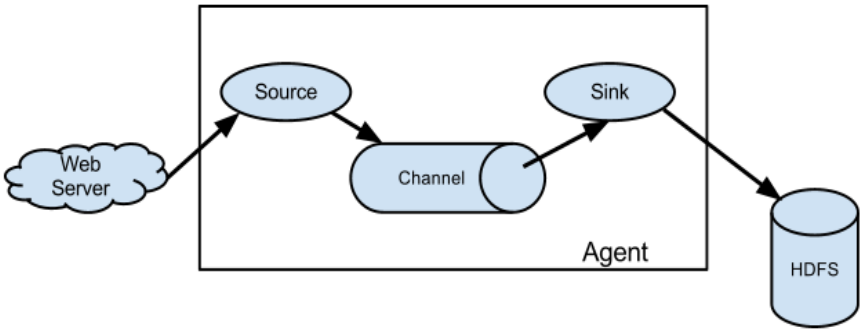


Figure 2.1: Dataflow inside a Flume Agent

A Flume data flow consists of one or more *agents*, with each agent configured with a *source*, a *sink*, and a *channel*. Inside an agent, data moves from the source to the sink via the channel (as

shown in Figure 2.1).

Similar to a data routing engine such as Storm, Flume enables the setting up of an arbitrary data flow but does not provide storage and indexing abilities natively. As such, it too needs to be glued together with a data store for persistence and providing ad hoc querying support, and it suffers from similar inefficiencies and limitations as were observed with a combination of systems such as Storm and MongoDB. Flume does not provide for the elastic scaling in or scaling out of any part of the dataflow to account for potentially dynamic demand for resources that vary in accordance with the rate of arrival of data.

2.4.2 Chukwa

Large distributed systems that are deployed over hundreds of nodes and constantly generate logs add to the complexity of the manual analysis and debugging of logs. Automated log-analysis is increasing the amount of information that can be extracted from logs, thus increasing their value. Large internet services companies use the Map Reduce programming model to process log data, so a number of log collection systems have been built to copy data into HDFS. These systems often lack a unified approach to failure handling, with errors being handled separately by each piece of the collection, transport, and processing pipeline. Apache Chukwa [19] is a scalable system for collecting logs and other monitoring data and processing the data with MapReduce.

Apache Chukwa is built on top of the Hadoop Distributed File System (HDFS) and MapReduce framework and inherits Hadoops scalability and robustness. Chukwa also includes a flexible and powerful toolkit for displaying, monitoring and analyzing results to make the best use of the collected data. Although Chukwa is not closely related with the data ingestion facility provided by AsterixDB, it still shares the spirit of providing a unified approach with an end-to-end delivery model for continuously generated data (logs) from multiple data sources (processes running in a distributed environment, each producing logs).

2.4.3 Kafka

Apache Kafka [30] is a distributed publish-subscribe messaging system. It was originally developed at LinkedIn and later on became an Apache project. Kafka is designed as a distributed system that is easy to scale out and offers high throughput for both publishing and subscribing. Dataflow in Kakfa involves a set of *producers*, *message brokers*, and *consumers*. A producer is anyone who can publish messages, which are a payload of bytes. Each message has an associated *topic*, which is a category or feed name to which messages are published. A consumer can subscribe to one or more topics and consume the published messages by pulling data from brokers. The brokers collectively constitute the Kafka cluster. Figure 2.2 shows the dataflow between the different entities in Kafka.

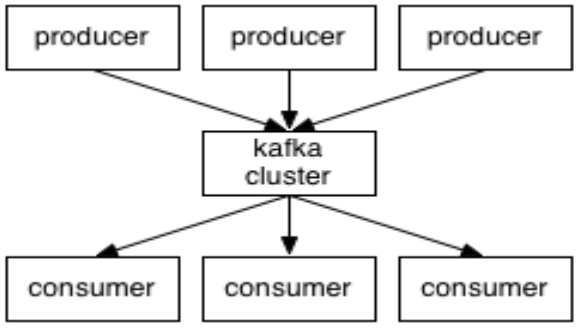


Figure 2.2: Dataflow inside Kafka

Kafka, by design, does not provide support for persistence, indexing, or querying of data. It is actually used in conjunction with Storm, wherein data from external source is pushed to Kafka and is subsequently pulled by Storm. Note that Storm does not support receiving data from push-based data sources, so pairing with Kafka is a popular method for overcoming this limitation. The 'glued' together system still needs further gluing with a data store to provide persistence and querying support.

2.4.4 Sqoop

Apache Sqoop [29] is a tool designed for efficiently transferring bulk data between Apache Hadoop and structured datastores such as relational databases. One can use Sqoop to import data from a relational database management system (RDBMS) such as MySQL or Oracle into the Hadoop Distributed File System (HDFS), transform the data in Hadoop MapReduce, and then export the transformed data back into an RDBMS. Sqoop automates most of this process, relying on the database to describe the schema for the data to be imported. Sqoop uses MapReduce to import and export the data, which provides for parallel operation as well as fault tolerance.

Sqoop, in a way, facilitates the execution of ad hoc analytical queries over data accumulated in HDFS by moving the data to an RDBMS. However, continuous ingestion of data from an external source would require persisting it into HDFS and creating an additional copy before it can be moved by Sqoop. Nonetheless, such creation of shadow data and batchy movement of introduces delays and involves large amounts of resources.

2.5 Summary

The need to be able to persist and index fast-flowing data is ubiquitous. This chapter has briefly reviewed a wide-spectrum covered the wide-spectrum of related systems that dealt with high-velocity data. However, these systems offered only a subset of the desired functionality, as none of them involved persistence of data or supported ad hoc analysis via queries. As such, these systems must be coupled with other systems to form an end-to-end solution[1]. Instead of 'gluing' together different systems, AsterixDB is different in being a unified system with "native", i.e., built-in support for data feed ingestion.

[1]The trade-offs and limitations of coupled systems in the context of data ingestion is described in greater detail in Chapter 7.

AsterixDB offers a generalized fault-tolerance approach that can be customized as per the application requirements and the expected degree of robustness. Furthermore, AsterixDB does not require data to be first staged to external storage (e.g., HDFS) before being ingested. To the best of our knowledge, AsterixDB is the first system to explore the challenges involved in building a data ingestion facility that is fault tolerant and employs partitioned parallelism to scale the facility and to couple it with high-volume and parallel external data sources.

Chapter 3

Background and Preliminaries

In this Chapter, we present a brief overview of AsterixDB and its execution layer – Hyracks – with an emphasis on the most important aspects and features from the perspective of this work. For a detailed description, the reader is referred to [2] and [18].

3.1 AsterixDB

Initiated in 2009, the AsterixDB project has been developing new technologies for ingesting, storing, indexing, querying, and analyzing vast quantities of semi-structured data. The project drew ideas from three distinct areas — semi-structured data management, parallel databases, and first generation Big Data platforms — to create an open-source software platform that scales by running on large, shared-nothing commodity computing clusters. The effort targeted a wide range of semi-structured use cases, ranging from "data" use cases — whose data is well-typed and highly regular — to "content" use cases — whose data is irregular, involves more text, and whose schema may be hard to anticipate a priori or may never exist. The initial results were released as an AsterixDB *beta* release in June of 2013, and the current open source release is available at [2].

To distinguish AsterixDB from current Big Data analytics platforms, which query but don't store or manage data, we classify AsterixDB as a *Big Data Management System* (BDMS). One of the project's mottos is "one size fits a bunch", and the hope is that AsterixDB will prove useful for a wider range of use-cases than are addressed by any one of the current Big Data technologies (e.g., Hadoop-based query platforms or key-value stores). We aim to reduce the need for "bubble gum and bailing wire" constructions involving multiple narrower systems and corresponding data transfers and transformations.

3.1.1 AsterixDB Architecture

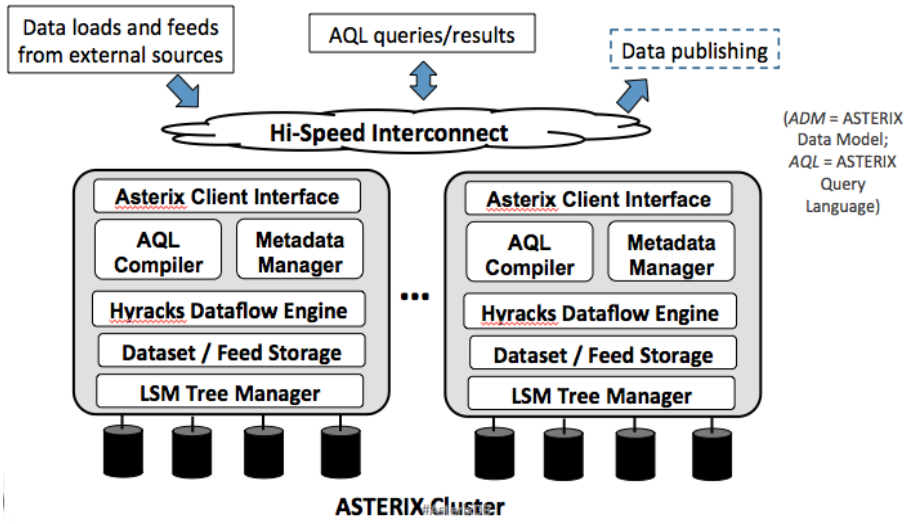


Figure 3.1: AsterixDB Architecture

Figure 3.1 provides an overview of how the various software components of AsterixDB map to nodes in a shared-nothing architecture. The topmost layer of AsterixDB is a parallel data manager, with a full, flexible data model (ADM) and query language (AQL) for describing, querying, and analyzing data. ADM and AQL support both native storage and indexing of data as well as analysis of external data (e.g., data in HDFS). The bottom-most layers from 3.1 provide storage facilities for datasets, which can be targets of ingestion. These datasets are stored and managed by AsterixDB as partitioned LSM-based B+-trees with optional LSM-based secondary indexes [33].

AsterixDB uses Hyracks [18] as its execution layer. Hyracks allows AsterixDB to express a computation as a DAG of data operators and connectors. Operators operate on partitions of input data and produce partitions of output data, while connectors repartition operators' outputs to make the newly produced partitions available at the consuming operators.

3.1.2 AsterixDB Data Model

AsterixDB defines its own data model (ADM) [16] designed to support semi-structured data with support for bags/lists and nested types. Listing 3.1 shows how ADM can be used to define a record type for modeling a raw tweet. The *RawTweet* type is an open type, meaning that its instances will conform to its specification but can contain extra fields that vary per instance. Listing 3.1 also defines a *ProcessedTweet* type. A processed tweet replaces the nested user field inside a raw tweet with a primitive string value (userId) and adds a nested collection of strings (referred_topics) to each tweet. Additionally, derived attributes about the tweet (e.g., sentiment and language) are included. The primitive location field types (latitude, longitude) and created_at are now expressed as their respective spatial (point) and temporal (datetime) datatypes. Note that ADM also allows specifying optional fields with known types (e.g., the location attribute in the *ProcessedTweet* type).

Data in AsterixDB is stored in *datasets*. Each record in a dataset conforms to the datatype associated with the dataset. Data is hash-partitioned (by primary key) across a set of nodes that form the *nodegroup* for a dataset. By default, all nodes in an AsterixDB cluster form the nodegroup for a dataset. Listing 3.2 shows the AQL statements for creating a pair of datasets—*Tweets* and *ProcessedTweets*. We create a secondary index on the location attribute of a processed tweet for more efficient retrieval of tweets on the basis of spatial location.

```
use dataverse feeds;
create type TwitterUser as open {
  screen_name: string,
  lang: string,
  friends_count: int32,
  statuses_count: int32,
  name: string,
  followers_count: int32
};

create type Tweet as open {
    id: string,
    user: TwitterUser,
    latitude: double?,
    longitude: double?,
    created_at: string,
    message_text: string,
    country: string?
};

create type ProcessedTweet as open {
    id: string,
    user_name: string,
    location: point?,
    created_at: datetime,
    message_text: string,
    country: string?,
    topics: [string],
    sentiment: double
};
```

Listing 3.1: Defining datatypes

```
use dataverse feeds;

create dataset Tweets(Tweet)
primary key id;

create dataset ProcessedTweets(ProcessedTweet)
primary key id;

create index locationIndex on
ProcessedTweets( location ) type rtree;
```

Listing 3.2: Creating datasets and associated indexes

3.1.3 Querying Data

AsterixDB queries are written in AQL, a declarative query language that was designed by borrowing the essence from XQuery [9]. As an example, consider the AQL query in Listing 3.3, which spatially aggregates tweets collected in the dataset *ProcessedTweets*. The query defines a bounding rectangle that spans over the geographic region covered by the US. It specifies the latitude and longitude increments to sub-divide this bounding rectangle into a grid-structure. The query begins by constraining the tweets to the bounding rectangle and those containing the hashtag "Obama". This step can be executed efficiently by using the secondary R-tree index on the location attribute (from Listing **??**). The location of each qualifying tweet together with the origin of the bounding rectangle and the latitude and longitude increments (to specify the resolution of the grid) are given to the spatial-cell function. The function returns the grid cell that the tweet belongs to. Tweets are then grouped according to their containing grid cells and the count function is applied to each cell. The result can be used draw a heat map showing the relative volume of tweets over a selected geographic region (Figure 3.2).

```
for $tweet in dataset ProcessedTweets
let $searchHashTag := "Obama"
let $leftBottom := create-point(33.13,-124.27)
let $rightTop := create-point(48.57,-66.18)
let $latResolution := 3.0
let $longResolution := 3.0
let $region := create_rectangle ($leftBottom, $rightTop)
where spatial_intersect ($tweet.location, $region) and
some $hashTag in $tweet.topics
 satisfies ($hashTag = $searchHashTag)
group by $c := spatial_cell ($tweet.location,
$leftBottom, $latResolution, $longResolution) with $tweet
return { "cell": $c, "count" : count($tweet) }
```

Listing 3.3: AQL query for spatial aggregation of tweets

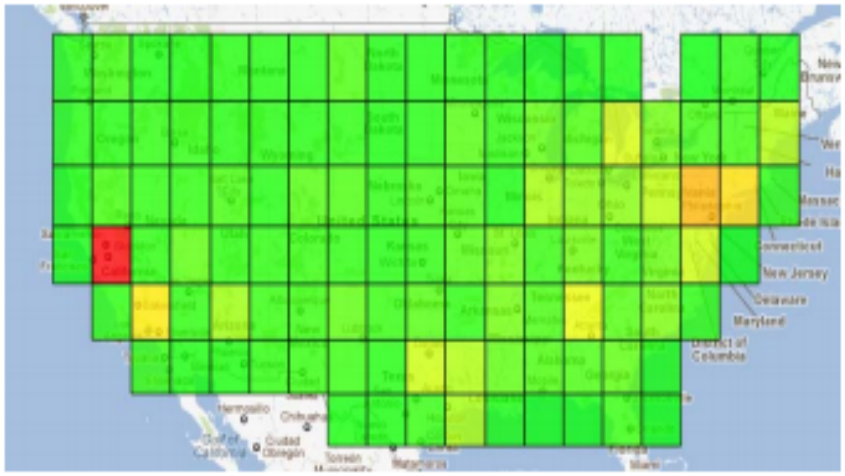


Figure 3.2: A visualization of the results of spatial aggregation query. The color of the cell indicates the tweet count.

3.2 Hyracks

Hyracks is a generalized alternative to infrastructures such as MapReduce [23], Hadoop [5] and Dryad [28] for solving data-parallel problems. It balances the need for expressiveness beyond Map-Reduce, which offers a very limited programming model based on a few user-provided functions, while providing out-of-the-box support for many commonly occurring communication patterns and operators needed in data-oriented tasks, which are absent in Dryad. Hyracks has been designed to work with a cluster of commodity computers. To do so in a robust manner, it has built-in support to detect and recover from possible system failures that might occur during the evaluation of a job through the use of heartbeats.

3.2.1 High-Level Architecture

Figure 3.3 provides an overview of the basic architecture of a Hyracks installation. Every Hyracks cluster is managed by a Cluster Controller process. The Cluster Controller accepts job execution requests from clients, plans their evaluation strategies (e.g., computing stages), and then schedules the job's tasks (stage by stage) to run on selected machines in the cluster. In addition, it is

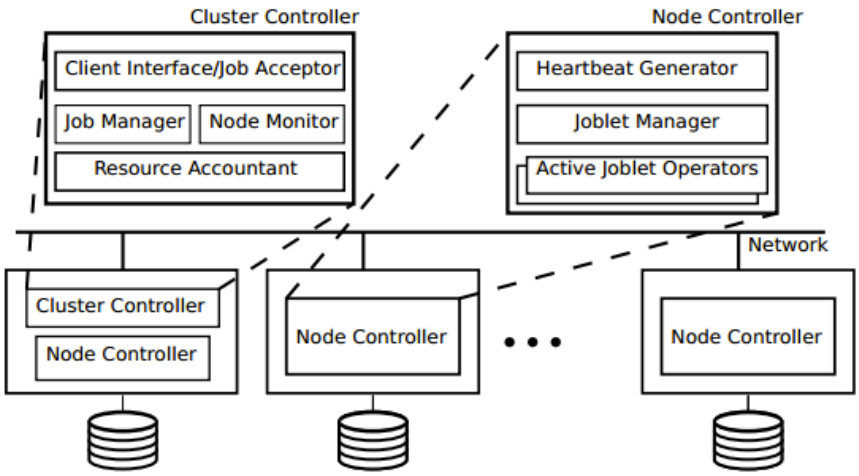


Figure 3.3: Hyracks Architecture

responsible for monitoring the state of the cluster to keep track of the resource loads at the various worker machines. The Cluster Controller is also responsible for re-planning and re-executing some or all of the tasks of a job in the event of a failure. Turning to the task execution side, each worker machine that participates in a Hyracks cluster runs a Node Controller process. The Node Controller accepts task execution requests from the Cluster Controller and also reports on its health (e.g., resource usage levels) via a heartbeat mechanism.

3.2.2 Execution Model

Hyracks provides a programming model and an accompanying infrastructure to efficiently divide computations on large data collections (spanning multiple machines) into computations that work on each partition of the data separately. A Hyracks job (the unit of work in Hyracks), submitted by a client, processes one or more collections of data to produce one or more output collections (also in the form of partitions). The job is a dataflow DAG comprising of operators (nodes) and connectors (edges). Operators represent job's partitioned-parallel computation steps and connectors represent the (re)-distribution of data from step to step. Internally, an individual operator consists of one or more activities (internal sub-steps or phases). At runtime, each activity of an operator is realized

as a set of (identical) tasks that are clones of the activity and that operate on individual partitions of the data flowing through the activity.

In Hyracks, the data flows between operators over connectors in the form of data frames containing physical records that can have an arbitrary number of fields. Hyracks provides support for expressing data-type-specific operations such as comparisons and hash functions. The type of each field is specified by providing an implementation of a descriptor interface that allows Hyracks to perform serialization and deserialization. For most basic types, e.g., numbers and text, the Hyracks library contains the required pre-existing type descriptors.

When Hyracks begins to execute a job, it takes the job specification and internally expands each operator into its constituent activities. This expansion of operators into constituent activities reveals to Hyracks the phases of each operator along with any sequencing dependencies among them. Activities that are transitively connected to other activities in a job only through dataflow edges are said to together form a stage. Intuitively, a stage is a set of activities that can be co-scheduled (to run in a pipelined manner, for example). A job's parallel execution details are planned in the order in which stages become ready to execute. (A given stage in a Hyracks job is ready to execute when all of its dependencies, if any, have successfully completed execution.)

A more detailed description of the execution model together with examples and an experimental evaluation can be found in [18].

3.3 Summary

AsterixDB is a Big Data Management System (BDMS) that provides for the storage, indexing and querying of semi-structured data. It uses a flexible data model – ADM – that supports rich data types such as bags, lists and nested types. Data analysis is expressed in a high-level language – the AsterixDB Query Language (AQL). AsterixDB employs a cluster of commodity hardware and

uses Hyracks as its runtime execution layer. An AQL (DDL) statement or a query is compiled into a single Hyracks job and scheduled to run on a cluster in a distributed manner.

In this chapter, we have covered sufficient ground to get a basic understanding of the data model and architecture used within AsterixDB. We shall next shift our focus towards the concepts involved in building a data ingestion facility. In subsequent chapters, we will describe how the data and execution model are leveraged to build a data ingestion facility that addresses the challenges enumerated in Chapter 1.

Chapter 4

Data Feed Basics

To provide support for continuous data ingestion, the AsterixDB query language (AQL) should have built-in support for data feeds. In this Chapter, we describe how an end-user may model a data feed and have its data be persisted and indexed in an AsterixDB dataset.

4.1 Collecting Data: Feed Adaptors

The first and foremost task in building a data ingestion facility is providing the ability to connect with an external data source and set up the required flow of data with the external source as the sender and the data management system (AsterixDB in the current context) as the receiver. As noted in Chapter 1, data sources are expected to use proprietary protocol(s) to facilitate data exchange and may send data in a specific format that requires parsing and translation into the AsterixDB Data Format (ADM). The functionality of establishing a connection with an external data source, receiving, parsing, and translating data into ADM records (for analysis and storage inside AsterixDB) is contained in a *Feed Adaptor*. AsterixDB does not concern itself with the semantics and connection parameters involved in initiating the flow of data from the external source; as such,

the Feed Adaptor is treated by the rest of the as a black box that outputs ADM records that are ready to be processed and stored within AsterixDB.

A Feed Adaptor is simply an implementation of a prescribed interface, and its details are specific to a given data source. An implementation of the interface provides AsterixDB with necessary details including the kind of data source (push versus pull) and the data type associated with the ADM records that are output by the adapter. To use a custom data source, an end-user simply needs to provide an implementation of the interface[1]. AsterixDB currently provides built-in adaptors for several popular data sources – namely Twitter, CNN, and RSS feeds. We are in the process of expanding the set to cover other popular data sources. AsterixDB additionally provides a generic socket-based feed adaptor that can be used to ingest data that is directed at a specified socket address.

We next illustrate how a feed adaptor is used to define a data feed. A data feed is defined in AQL using the *create feed* statement. Listing 4.1 illustrates the use of built-in adaptors in AsterixDB to define a pair of feeds. Note that an adaptor may optionally be given configuration parameters that are used in interfacing with the external source (establishing a connection) and parsing the received content. In Listing 4.1, the built-in pull-based Twitter adaptor is provided with a set of keywords that are used in filtering the set of received tweets. As configured, the pull-based Twitter adaptor will make a request for data every minute and make use of the Twitter search API. The CNN adaptor is used in constructing a feed that will consist of news articles from CNN that are related to any of the topics that are specified as part of its configuration.

It is possible that the protocol for data exchange between the external source and its feed adaptor allows for the transfer of data in parallel across multiple channels. The degree of parallelism in receiving data from an external source is determined by the feed adaptor in accordance with the data exchange protocol. The pull-based TwitterAdaptor uses a single degree of parallelism,

[1]Custom feed adaptors together with any dependencies are packaged into an AsterixDB library and installed into an AsterixDB instance using the AsterixDB management tool — Managix. The details for this are described in Appendix A.

```
create feed TwitterFeed using TwitterAdaptor
("query"="Obama", "interval"=60);

create feed CNNFeed using CNNAdaptor
("topics"=" politics , sports");
```

Listing 4.1: Defining a feed using some of the built-in adaptors in AsterixDB

whereas the CNNAdaptor uses a degree of parallelism determined by the number of topics that are passed as configuration. Corresponding to each topic (politics, sports, etc.) is an RSS feed that is fetched by an individual instance of the CNNAdaptor. Multiple instances of a feed adaptor may run as parallel threads on a single machine or on multiple machines across an AsterixDB cluster.

4.2 Pre-Processing Collected Data

A feed adaptor outputs ADM-formatted records that can be processed and stored within AsterixDB as part of a dataset. However, the incoming records, in terms of their content (contained attributes), may not be appropriate for storage or querying as per the requirements of the overlying application. The incoming records may require pre-processing that includes (but is not limited to) filtering of unwanted attributes (or even records), transforming content such as removing whitespaces or rounding up to certain precision, sampling or applying sophisticated processing such as sentiment analysis, or feature extraction for content-based classification.

A feed definition (*create feed statement*) may optionally include the specification of a user-defined function (UDF) that needs to be applied to each feed record prior to its persistence. The pre-processing is expressed as a user-defined function (UDF) that can be defined in AQL or in a programming language like Java. An AQL UDF is a good fit when pre-processing a record is very simple or requires the result of a query (join or aggregate) over data contained in AsterixDB dataset(s). More sophisticated processing such as sentiment analysis of text is better handled by

providing a Java UDF. A Java UDF has an initialization phase that allows the UDF to access any resources it may need to initialize itself prior to being used in a data flow. It's computation is assumed by the AsterixDB compiler to be stateless and thus usable as an embarassingly parallel black box (similar to the map function in Map Reduce). In contrast, the AsterixDB compiler can reason about an AQL UDF and even involve the use of indexes during its invocation. The pre-processing function for a feed is specified using the *apply function* clause at the time of creating the feed. This is illustrated in Listing 4.3.

The tweets collected by the TwitterAdaptor (Listing 4.1) conform to the *Tweet* datatype (Listing 3.1). The processing required in transforming a collected tweet to its lighter version (of type *ProcessedTweet*) involves extracting hash tags[2] (if any) in a tweet and collecting them under the referred-topics attribute for the tweet. This can be expressed as an AQL function as shown in Listing 4.2. More sophisticated pre-processing might require implementation in a programming language like Java. As an example, the CNNAdaptor (Listing 4.1) outputs records that each contain the fields - item, link, and description. The *link* field provides the URL of the news article on the CNN website. Parsing the HTML source provides additional information such as tags, images and outgoing links to other related articles. The extracted information could then be added to each record as additional fields to form an augmented version prior to persistence. Note that the return type of the function associated with a feed must conform to the datatype of the target dataset where the feed is to be persisted.

A feed adaptor and a UDF act as *pluggable* components that contribute towards providing a generic model for data ingestion and help to address challenge C1 from in Section 1.1. By providing an implementation of prescribed interfaces, the internal details of data feed management are abstracted from the end-user. These pluggable components can be packaged and installed as part of an AsterixDB library and subsequently be used in AQL statements. A detailed description of the steps involved in building a custom adaptor or a Java UDF and installing it within an AsterixDB instance

[2]Hash tags are words that begin with a #. In Twitter's jargon, these represent the topics associated with the tweet.

```
use dataverse feeds;

create function addHashTags($x){
  let $topics := (for $token in word-tokens($x.message_text)
                  where starts-with($token, "#")
                  return $token)
  return {"id":$x.id,
          "user": $x.user,
          "latitude": $x.latitude,
          "longitude": $x.longitude,
          "created_at": $x.created_at,
          "message_text": $x.message_text,
          "country": $x.country,
          "topics": $topics}
};
```

Listing 4.2: An AQL function to extract hash tags contained in a tweet's text and collect them into an ordered list that is added to the tweet as an additional attribute (topics)

```
create feed ProcessedTwitterFeed using TwitterAdaptor
("query"="Obama", "interval"=60)
apply function addHashTags;

create feed ProcessedCNNFeed using CNNAdaptor
("topics"="politics,sports")
apply function extractInfoFromCNNWebsite;
```

Listing 4.3: Defining a feed that involves pre-processing of collected data

is provided in Appendix A.

4.3 Building a Cascade Network of Feeds

Multiple overlying applications may wish to consume the data ingested from a given data feed. Each such application might perceive the feed in a different way and require the arriving data to be processed and/or be persisted differently. Building a separate flow of data from the external source for each application would be wasteful of resources as the pre-processing or transformations required by each application might overlap and could be done together in an incremental fashion to avoid redundancy. A single flow of data from the external source could provide data for multiple applications. To achieve this and address challenge C2 from Section 1.1, we introduce the notion of *primary* and *secondary* feeds in AsterixDB.

A feed in AsterixDB is considered to be a *primary* feed if it gets its data from an external data source. The records contained in a feed (subsequent to any pre-processing) are directed to a designated AsterixDB dataset. Alternatively or additionally, these records can be used to derive other feeds known as *secondary* feeds. A secondary feed is similar to its parent feed in every other aspect; it can have an associated UDF to allow for any subsequent processing, can be persisted into a dataset, and/or can be used to derive other secondary feeds to form a *cascade network* (see Figure 4.2). A primary feed and its dependent secondary feeds form a hierarchy.

```
create secondary feed ProcessedTwitterFeed from
feed TwitterFeed apply function addFeatures;

create secondary feed ProcessedCNNFeed from
feed CNNFeed apply function addInfoFromCNNWebsite;
```

Listing 4.4: Defining a secondary feed

As an example, Listing 4.4 shows the AQL statements that redefine the previously defined feeds

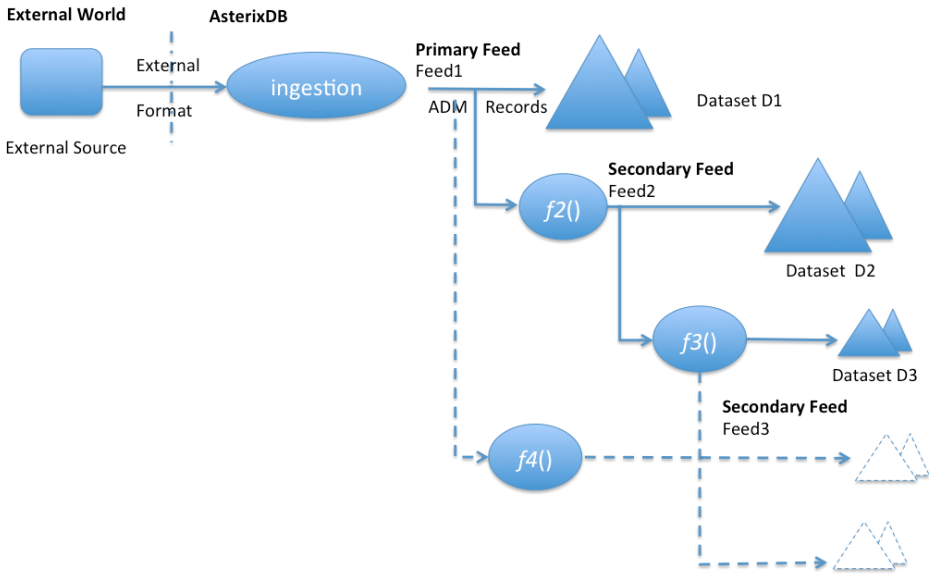


Figure 4.1: Building a cascade network of feeds. The solid lines represents the flow of data as constructed by creating a primary feed and additional secondary feeds that apply additional processing to form a cascade network. The dotted lines indicate example additions to the network with data flowing into newer secondary feeds and data sets.

— *ProcessedTwitterFeed* and *ProcessedCNNFeed* — in terms of their respective parent feeds from Listing 4.1. If a specific application needs to subject the tweets contained in the feed — ProcessedTwitterFeed — to additional processing (prior to persistence), it may do so by defining another secondary feed using ProcessedTwitterFeed as a parent feed and associating a UDF that does the additional processing.

4.4 Lifecycle of a Feed

A feed is a *logical* concept and is brought to life (i.e., its data flow is initiated) *only* when it is *connected* to a dataset using the *connect feed* AQL statement (Listing 4.5). Subsequent to a connect feed statement, the feed is said to be in the *connected* state. Multiple feeds can simultaneously be connected to a dataset such that the dataset represents the union of the connected feeds. In a possible but unlikely scenario, a feed may also be simultaneously connected to different datasets.

Note that connecting a secondary feed does not require the parent feed (or any ancestor feed) to be in the *connected* state. The order in which feeds that are related in a hierarchy are connected to their respective datasets is not important. Furthermore, additional secondary feeds can be added to an existing hierarchy and connected to a dataset at any time without impeding or interrupting the flow of data along a connected ancestor feed.

The *connect feed* statement in Listing 4.5 directs AsterixDB to persist the *ProcessedTweets* feed in the *ProcessedTweets* dataset. If it is required (by the high-level application) to retain the raw tweets obtained from Twitter, the end-user may additionally choose to connect the *TwitterFeed* to a different dataset. Having a set of primary and secondary feeds offers the the end-user the flexibility to do so. Let us assume that the high-level application indeed needs to persist *TwitterFeed* and that, to do so, the end-user makes use of the *connect feed* statement. A logical view of the continuous flow of data established by connecting the feeds to their respective target datasets is shown in Figure 4.2.

Referring to Figure 4.2, it is worth noting that data may not flow along the different paths (primary feed and secondary feed) at the same rate. For example, the application of a UDF along the secondary feed introduces an additional overhead that may have the effect of slowing the movement of data along that path. It is important that data feeds in a cascade network be isolated from each other in a way that guarantees a continuous flow of data independent of the rate of flow of data along other related feeds. AsterixDB provides such isolation by the mechanism of creating *Feed Joints*, as described in greater detail in Chapter 5 where we cover the physical aspects and implementation details of feeds.

```
connect feed ProcessedTwitterFeed to
dataset ProcessedTweets;

disconnect feed ProcessedTwitterFeed from
dataset ProcessedTweets;
```

Listing 4.5: Managing the lifecycle of a feed using *connect* and *disconnect* AQL statements

Contrary to the *connect feed* statement, the flow of data from a feed into a dataset can be terminated explicitly by use of the *disconnect feed* statement (Listing 4.5). Note that disconnecting a feed from a particular dataset does not interrupt the flow of data from the feed to any other dataset(s); neither does it impact other connected feeds in the lineage.

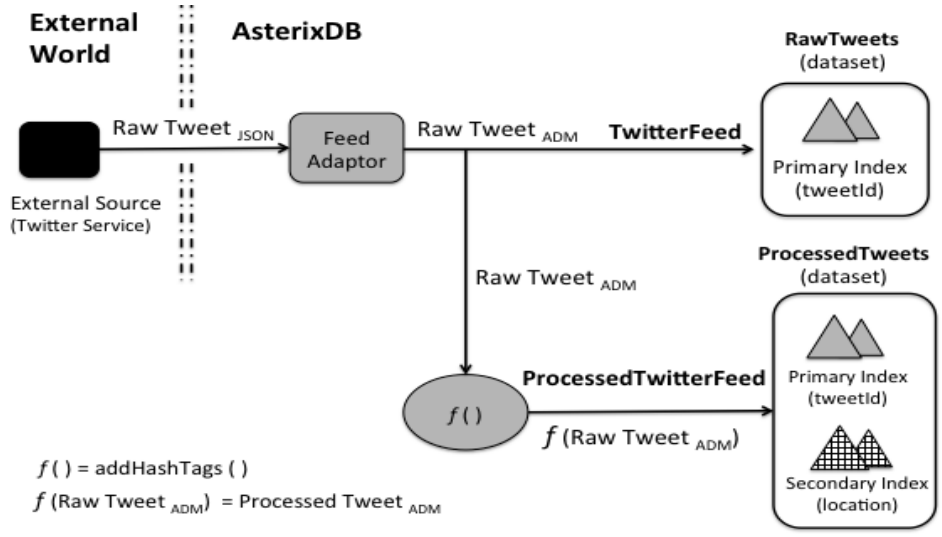


Figure 4.2: Logical view of the flow of data from external data source into AsterixDB datasets

The notion of a primary and a secondary feed allows the end-user to derive a feed from an existing feed and form a cascade network with data flowing from one feed to another and so on and so forth. This ability, coupled with the flexibility to connect each feed in a cascade network to a different dataset, helps in achieving the "Fetch-Once, Compute-Many" model described in Chapter 1. Furthermore, a cascade network can be structurally modified by connecting or disconnecting feeds in a non-disruptive manner wherein other connected feeds are isolated from structural changes occurring in the cascade network. Figure 4.2 illustrates the *Fetch-Once, Compute-Many* model where a record (tweet) is retrieved from the external source (Twitter) once but is pushed along multiple paths to be processed and persisted differently.

4.5 Policies for Feed Ingestion

Multiple feeds may be concurrently operational on an AsterixDB cluster, each competing for resources (including CPU cycles, network bandwidth, and disk IO) to maintain pace with their respective data sources. A data management system must be able to manage a set of concurrent feeds and make dynamic decisions related to the allocation of resources, resolving resource bottlenecks and the handling of failures. Each feed has its own set of constraints, influenced largely by the nature of its data source and the application(s) that intend to consume and process the ingested data. Consider an application that intends to discover the trending topics on Twitter by analyzing the *ProcessedTwitterFeed* feed. Losing a few tweets may be acceptable. In contrast, when ingesting from a data source that provides a click-stream of ad clicks, losing data would translate to a loss of revenue for an application that tracks revenue by charging advertisers per click.

AsterixDB allows a data feed to have an associated *ingestion policy* that is expressed as a collection of parameters and associated values. An ingestion policy dictates the runtime behavior of the feed in response to resource bottlenecks and failures. Note that during push-based feed ingestion, data continues to arrive from the data source at its regular rate. In a resource-constrained environment, a feed ingestion framework may not be able to process and persist the arriving data at the rate of its arrival. AsterixDB provides a list of policy parameters (Table 4.1) that help to customize the system's runtime behavior when handling excess records. AsterixDB provides a set of built-in policies, each constructed by setting appropriate value(s) for the policy parameter(s) from Table 4.1.

The handling of excess records by the built-in ingestion policies of AsterixDB is summarized in Table 4.2. Buffering of excess records in memory under the 'Basic' policy has clear limitations given that memory is bounded and may result in a termination of the feed if the available memory or the allocated budget is exhausted. The 'Spill' policy resorts to spilling the excess records to the local disk for deferred processing until resources become available again. Spilling is done

intermittently during ingestion when required, and the spillage is processed as soon as resources (memory) are available. In contrast, the 'Discard' policy causes the excess records to be discarded altogether until the existing backlog is cleared. However, this results in periods of discontinuity when no records received from the data source are persisted. This behavior may not be acceptable to an application wishing to consume the ingested data. A best-effort alternative is provided by the 'Throttling' policy, wherein records are randomly filtered out (sampled) to effectively reduce their rate of arrival. In addition, AsterixDB also provides the 'Elastic' policy, which attempts to scale-out/in by increasing/decreasing the degree of parallelism involved in processing of records. We shall revisit the built-in policies in Section 7.3 (Chapter 6) where we discuss the physical aspects and implementation details and evaluate the the impact of the chosen ingestion policy on ingestion throughput and latency.

Note that Listing 4.7 shows an example where a primary feed (*TwitterFeed*) and a dependent secondary feed (*ProcessedTwitterFeed*) are both connected using a common policy (Basic), but this is not a requirement. The ability to form a custom policy allows the runtime behavior to customized as per the specific needs of the high-level application(s) and helps address challenge C1 from Section 1.1.

A given end-user may choose to form a custom policy. E.g., it is possible in AsterixDB to create a custom policy that spills excess records to disk and subsequently resorts to throttling if the spillage crosses a configured threshold. In the example shown in Listing 4.6, a custom policy — Spill_then_Throttle — is created by extending the built-in Spill policy and overriding the appropriate parameters. The parameter "max.spill.size.on.disk" limits the spillage size in terms of bytes written to disk while the parameter "excess.records.throttle" allows the system to regulate the rate

Table 4.1: A few important policy parameters and their corresponding default value

Policy Parameter	Description	Default Value
excess.records.spill	Set to true if records that cannot be processed by an operator for want of resources (referred to as *excess records* hereafter) should be persisted to the local disk for deferred processing.	false
excess.records.discard	Set to true if *excess records* should be discarded.	false
excess.records.throttle	Set to true if rate of arrival of records is required to be reduced in an adaptive manner to prevent having any *excess records*.	false
excess.records.elastic	Set to true if the system should attempt to resolve resource bottlenecks by re-structuring and/or rescheduling the feed ingestion pipeline.	false
recover.soft.failure	Set to true if the feed must attempt to survive any runtime exception. A false value permits an early termination of a feed in such an event.	true
recover.hard.failure	Set to true if the feed must attempt to survive a hardware failure (loss of AsterixDB node(s)). A false value permits the early termination of a feed in the event of a hardware failure.	true

Table 4.2: Approach adopted by different policies in handling of excess records

Policy	Approach to handling of excess records
Basic	Buffer excess records in memory
Spill	Spill excess records to disk for deferred processing
Discard	Discard excess records altogether
Throttle	Randomly filter out records to regulate the rate of arrival
Elastic	Scale out/in to adapt to the rate of arrival

of inflow (throttle) in the event when the spillage crosses the configured threshold. In all cases, the desired ingestion policy is specified as part of the connect feed statement (Listing 4.7) or else the 'Basic' policy will be chosen as the default. Chapter 5 includes an evaluation of the built-in policies to study the impact of policy parameters on the runtime behavior during feed ingestion.

```
use dataverse feeds;

create ingestion policy Spill_then_Throttle from policy Spill
(("max.spill.size.on.disk"="512MB", "excess.records.throttle"="true"));
```

Listing 4.6: Specifying the ingestion policy for a feed

```
use dataverse feeds;

connect feed TwitterFeed to dataset RawTweets
using policy Basic;

connect feed ProcessedTwitterFeed to
dataset ProcessedTweets using policy Basic;
```

Listing 4.7: Specifying the ingestion policy for a feed

4.6 Summary

In this chapter, we have described the built-in support for data feeds in AQL We focused on the logical concepts involved in modeling a data feed and how these can be leveraged in building an extensible data flow that contains pluggable components (feed adaptors and UDFs) to cater to a wide variety of data sources and data-driven applications. We emphasized the level of abstraction provided to the end-user, whereby the runtime behavior for a data ingestion pipeline can be defined in a declarative manner by choosing or forming new ingestion policies. Such a design allows an end-user to customize the data ingestion task in accordance with the requirements of the overlying application(s).

We shall next focus on the physical aspects and describe the implementation details in building a data ingestion facility.

Chapter 5

Runtime for Data Ingestion

So far we have described, at a logical level, the user model and built-in support in AQL that enables the end-user to model a feed, manage its lifecycle, and dictate its runtime behavior by choosing an ingestion policy. In this chapter, we discuss the physical aspects and implementation details involved in building and managing the flow of data when a feed is connected to a dataset. We also describe how a feed cascade network is constructed and how its structure is dynamically modified when additional feeds are connected or active feeds are disconnected. We conclude the chapter with an experimental evaluation that compares data ingestion with batch inserts as mechanisms for putting data into indexed storage. We demonstrate the linear scalability offered by the data ingestion facility in terms of ingesting increasingly large volumes of data with additional of resources (hardware); in addition, we demonstrate the performance benefits gained from using a cascade network of feeds in maintaining a single flow of data from an external source while processing and persisting the collected data in different ways.

5.1 Feeds Metadata

Like a more conventional database system, AsterixDB has a system catalog (referred to as AsterixDB Metadata, hereafter) that contains definitions of different database objects including datasets, datatypes, indexes, data feeds, datasource adaptors, user-defined functions, etc. Instead of managing the Metadata outside the system, AsterixDB stores Metadata natively as a collection of AsterixDB datasets contained in a system-defined dataverse – the *Metadata* dataverse. Physically, the AsterixDB Metadata is stored at a chosen AsterixDB node, also known as the *MetadataNode*. All information related to the set of defined feeds and datasource adaptors is stored in the *Feeds* dataset and the *DatasourceAdapter* dataset respectively. In addition, information related to the set of user-defined functions is stored in the *Function* dataset.

The *DatasourceAdapter* dataset is pre-populated with the set of built-in adaptors. For each datasource adaptor, the corresponding dataset record captures the adaptor's factory class and the alias. The information contained per feed in the *Feed* dataset varies in accordance with the type of the feed (primary or secondary). In the case of a primary feed, the captured information includes the name (alias) of the associated adaptor together with the configuration parameters specified as part of the *create feed* AQL statement. In contrast, a secondary feed does not have any associated feed adaptor as it derives its data from another (primary or secondary) feed. The metadata for a secondary feed includes the name of the parent feed. For either kind of feed, the metadata also includes the name of the associated UDF (if any).

A data feed (defined using the *create feed* AQL statement) by itself is a logical concept and is not associated with any sort of runtime dataflow. The *create feed* AQL statement is only a Metadata operation that puts an additional record into the *Feeds* dataset. The actual flow of data is initiated only when the feed is connected to a target dataset using the *connect feed* AQL statement. Next, we describe in detail how the *connect feed* AQL statement is processed by the AsterixDB compiler to produce a dataflow that is referred to as a *data ingestion pipeline*.

5.2 Basic Runtime Components

AsterixDB uses Hyracks as its runtime execution engine. Hyracks allows AsterixDB to execute a directed acyclic graph (DAG) of operators and connectors (described in more detail below) that together form a dataflow. In this section, we discuss briefly the basic building blocks of a data ingestion pipeline. These are essentially the tools at hand for the AsterixDB compiler when processing a *connect feed* AQL statement and constructing the required data ingestion pipeline.

1. **Operator**: A Hyracks operator represents custom logic or a computation that may be applied in parallel to partitions of input data to produce partitions of output data. An operator can have an associated set of constraints (count or location constraints) that determine the degree of parallelism (number of parallel instances) at runtime and the specific location(s) where each instance may be scheduled to run. AsterixDB includes a set of built-in operators that apply specific logic to partitions of input data. As an example, the *IndexInsert* operator is used to insert data records into a primary or a secondary index.

2. **Connector**: A Hyracks connector represents the mode of exchange of data between a pair of operators that act as producers and consumers of data. Examples of connectors used in building a data ingestion pipeline include the *OneToOneConnector*, *HashPartitioningConnector*, and *RandomPartitioningConnector*.

3. **Feed Joint**: In addition to operators and connectors, a new building block for a data ingestion pipeline is the *feed joint*. At a high-level, a feed joint, when located at the output side of an operator, offers a mechanism for the output to be directed along multiple paths in a concurrent fashion. A feed joint is similar in spirit to a connector in the sense that it facilitates transfer of data between a pair of operator instances. However, unlike a connector, a feed joint allows for the transfer of data between operators that belong to different Hyracks jobs. This property makes it critical for use within a data ingestion pipeline. Furthermore, a feed joint allows an operator to (un)register itself dynamically for receiving data from a source

operator, a property that helps in constructing a cascade network of feeds and allowing it to expand or shrink dynamically.

A feed joint can be considered to loosely resemble a network tap that makes the data flowing through the data ingestion pipeline accessible and routable. In a Hyracks job, data is exchanged between a pair of operators in the form of fixed-size chunks known as *data frames*. Each operator in a Hyracks job is provided with an *IFrameWriter* handle that it uses to send output data frames downstream to the next operator in the data flow. Implementation-wise, a feed joint implements the IFrameWriter interface that provides the *nextFrame()* API. This allows the operator to remain agnostic of the internal implementation of the IFrameWriter and how its output data frames are being handled and sent downstream. Feed joints and their internal mechanism for transferring data to multiple recipient operator instances are discussed in greater depth in Section 5.4.

5.3 Building the Data Ingestion Pipeline

It is important for a data ingestion pipeline to offer flexibility in having the data be routed along multiple paths and enabling construction of a cascade network that involves multiple ingestion pipelines. Furthermore, in a cascade network, it is essential to insulate each pipeline from the other such that failure(s) or congestion (slow movement of data) along any path does not impede the flow of data through the rest of the network. Additionally, the cascade network must exhibit the ability to expand or contract by addition or removal of individual ingestion pipelines. In order to form such a flexible network, we designed a data ingestion pipeline to consist of a *head* section and a *tail* section, each constructed as a separate Hyracks job.[1]

The head section of a data ingestion pipeline provides the mechanism for collecting the set of

[1]The benefits of having a pair of Hyracks jobs instead of a single job are discussed in detail in later sections of this chapter and in the discussion of fault tolerance in Chapter 6.

data records that constitute the feed. To collect the records, the head section makes use of the feed adaptor to interface with the external data source. Output records from the head section are handled by the tail section that is responsible for directing the records to specific partitions of the target dataset. The specific target partition for a given data record is chosen by hashing the primary key contained in the record. The tail section also handles the pre-processing of records, if required, via the application of a UDF (specified as part of *create feed* statement) to each data record prior to persistence. A high-level view of a data ingestion pipeline with its head and tail sections is shown in Figure 5.1.

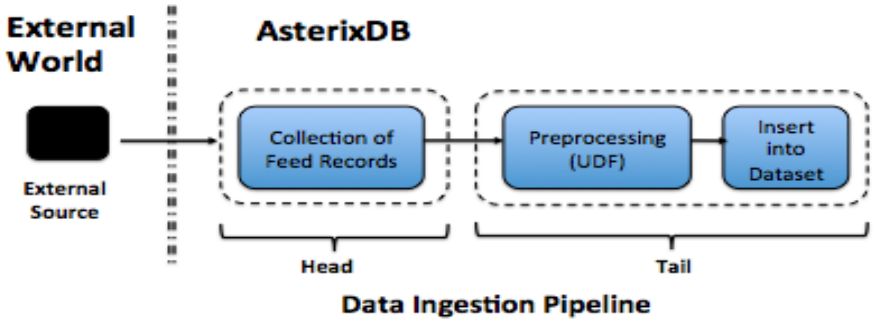


Figure 5.1: A high-level view of a data ingestion pipeline showing the *head* and *tail* sections and the functionality provided by each

Figure 5.1 represents the bare-bone skeleton structure of a data ingestion pipeline. It is worth noting that an ingestion pipeline can be fed with data that is moving across an existing ingestion pipeline to form a cascade network.This obviates the need of having a head section to retrieve records from the external source. The data ingestion pipeline then shares the head section but has a tail section of its own. Figure 5.2 shows a high-level view of a cascade network that involves a pair of data ingestion pipelines with a shared head section.

In order to understand the different steps involved in building and scheduling an ingestion pipeline, it is important to study different example scenarios. The differences in the structure of the constructed ingestion pipeline arise from the involvement of UDFs (Java or AQL) and the existing set of feeds that are currently in the *connected* state which can act as a source of data.

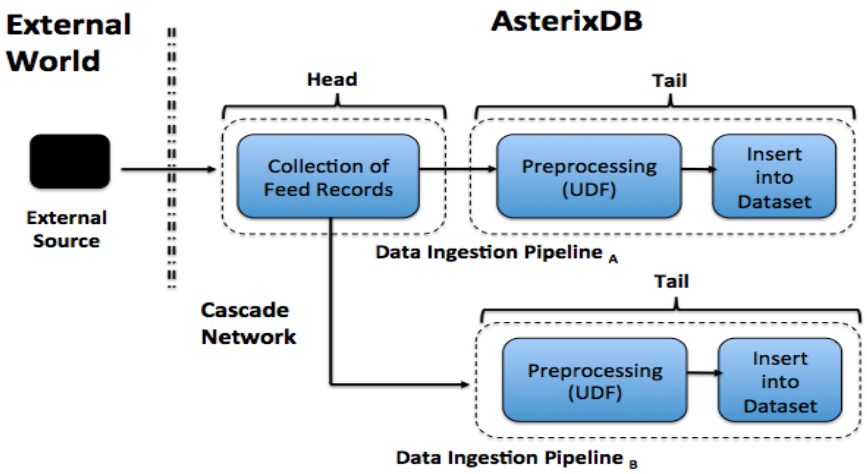


Figure 5.2: A high-level view of a cascade netowrk that involves a pair of ingestion pipelines with a shared head section and separate tail sections

In the discussion that follows, we assume an AsterixDB cluster where initially no feeds are actively running on the cluster. An AsterixDB cluster consists of a *master* node (Cluster Controller) and a set of *worker* nodes (Node Controllers). Each Node Controller additionally hosts a *FeedManager* that communicates with a single Central Feed Manager that is hosted by the Cluster Controller. The part played by the per node Feed Manager and the Central Feed Manager is further discussed, when appropriate, in subsequent sections of this chapter. We define and connect feeds in a step-wise manner.

- **Step 1**: *Primary Feed without a UDF*

  We start with a basic example of a primary feed that does not have an associated UDF.

- **Step 2**: *Secondary Feed with an AQL UDF*

  Subsequently, we define a secondary feed that extends our primary feed (from Step 1) and has an associated pre-processing that is expressed as an AQL UDF. Our choice of example allows us to describe in detail how the connected state of the parent primary feed is leveraged in building the data ingestion pipeline for a secondary feed.

- **Step 3**: *Secondary Feed with a Java UDF*

```
use dataverse feeds;

create feed TwitterFeed using TwitterAdaptor
(("type_name" = "Tweet"), ("query" = "Obama"));

connect feed TwitterFeed to dataset Tweets;
```

Listing 5.1: An example AQL statement to connect a primary feed (without an associated UDF) to a target dataset

Eventually, we define and connect another secondary feed that extends the secondary feed from Step 2. We use this example to describe how a secondary feed may act as a parent feed for another. To make things a bit different and emphasize the differences in handling of a Java UDF (as against an AQL UDF), we chose to associate a Java UDF with our secondary feed.

Following the steps above, we shall be constructing a cascade network of feeds in an incremental manner, starting with an idle cluster. At each step, we emphasize the methodology involved in building a data ingestion pipeline and how existing connected feeds are leveraged.

5.3.1 Primary Feed without a UDF

We begin by considering a basic example of a primary feed that does not have an associated UDF and is connected to a target dataset. Listing 5.1 shows the set of AQL statements that define and connect our example feed to a target dataset. We describe next, how the specific sections — *head* and *tail* — are constructed to build the ingestion pipeline.

Head Section

Given that we started with an idle cluster and that no other feeds are in the connected state, the records that constitute the *TwitterFeed* would need to be retrieved from the datasource (Twitter) using the feed adaptor associated with the *TwitterFeed*. As per the definition of our example feed, obtaining the data records for the feed requires the use of the TwitterAdaptor with the configuration parameters specified as part of the *create feed* statement (Listing 5.1). AsterixDB remains agnostic of the data transfer protocol followed by the data source (Twitter in the context of our example feed). The task of establishing a connection with the external source and parsing and translating data into ADM format is done by the feed adaptor (*TwitterAdaptor* in the case of our example feed). In doing so, the adaptor may choose to run as some number of parallel instances (by defining a *count* constraint) and may choose specific location(s) (AsterixDB node(s)) by defining a *location* constraint. Each feed adaptor has an associated *factory* class that allows AsterixDB to instantiate and configure an adaptor instance. The factory class also provides an API that is used by AsterixDB to obtain the constraints (count or location) for the adaptor. Writing a feed adaptor requires providing the associated factory class that includes an implementation of the *getConstraints()* API. In our current example, the TwitterAdaptor requires a single instance that may be scheduled on any node in the AsterixDB cluster.

The head section of a data ingestion pipeline corresponds to a Hyracks job, hereafter referred to as the *Feed Collect* job, and is shown in Figure 5.3. The FeedCollect job consists of *FeedCollect* operator that connects with a *NullSink* operator using a *one-to-one* connector.[2] A FeedCollect operator instance is responsible for housing an adaptor instance, managing its lifecycle, and using it to retrieve data from the external source. The count and location constraints for the FeedCollect operator are thus identical to those of the feed adaptor and they are obtained using the adaptor's factory class. Each FeedCollect operator instance is additionally provided with a handle to the factory class to enable the creation and configuring of an instance of the feed adaptor. The NullSink

[2]Each instance of FeedCollect operator is connected with a corresponding instance of the *NullSink* operator.

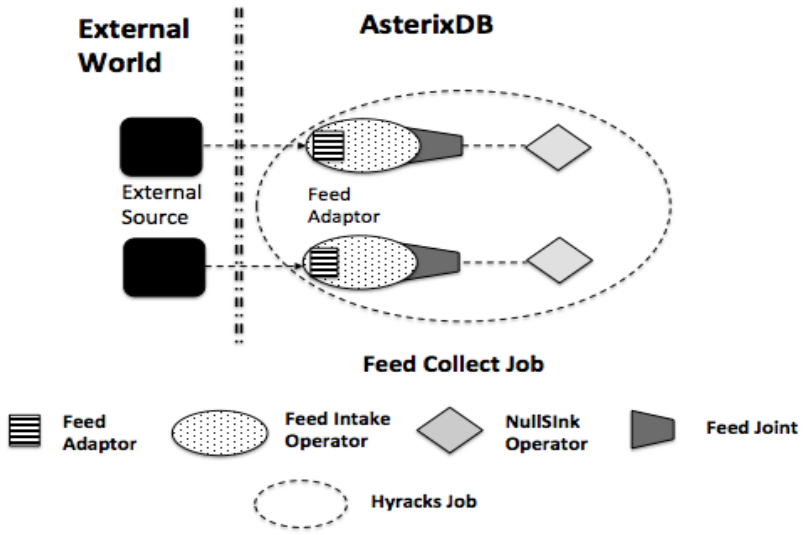


Figure 5.3: An example Feed Collect job that involves a pair of feed adaptor instances

operator in Figure 5.3 is a no-op in the sense that it doesn't process any data records at runtime.

At the output side of each FeedCollect operator instance is a feed joint. An operator instance with an associated feed joint at its output is known as a *subscribable* instance. A subscribable instance offers a simple subscription service for its output data and registers itself with the local Feed Manager using a unique ID. The unique ID (a string value) is chosen to symbolically represent the data records output by the subscribable instance. In general, a subscribable instance that outputs the result of the application of a sequence of functions $f_1(), f_i(), .., f_N()$ on a feed is assigned the ID – <*name of the feed*>: $f_1 : f_2, ..., f_{N-1} : f_N$. In our current example (refer to Figure 5.3), each FeedCollect operator instance is a subscribable instance that outputs records that represent the *TwitterFeed*. These instances are registered with the ID - "TwitterFeed". Our current example does not involve an associated UDF. In subsequent sections, when we discuss example feeds with an associated UDF, we shall look at other examples of IDs that have function name(s) embedded in them. Their use will also be discussed later in this chapter in the right context. Here, we continue our discussion using the ID "*TwitterFeed*".

The Feed Manager maintains a mapping between these unique IDs and their respective subscrib-

able operator instances, and it makes these discoverable by providing a simple search API. Any co-located operator instance may invoke the search API with an appropriate 'ID' to retrieve a handle to a co-located subscribable instance. The operator instance may then register itself as a recipient with the subscribable instance and begin receiving the output records from the subscribable instance. Note that a FeedCollect operator instance initially does not have any subscribers for its data. The creation and use of an adaptor instance by the FeedCollect operator instance is deferred until there is a request for the operator's output data to be routed (via the feed joint) to a recipient operator instance. The constructed head section of the data ingestion pipeline transits to the '*active*' state when the tail section has been setup. The construction of the *tail* section of a data ingestion pipeline is described next. To begin with, the feed joint will not have a registered set of recipient operators and will be marked as being in the *inactive* state.

Tail Section

While the head section is responsible for the collection of feed records, their subsequent processing to deposit them into the target dataset is handled by the *tail* section of the data ingestion pipeline. This task can be considered as similar to putting records into a dataset using the conventional *insert* statement supported in AQL, which is well understood by the AsterixDB compiler. In constructing the tail section, the AsterixDB compiler first rewrites the connect feed statement into an equivalent insert statement. The rewritten form is then compiled to generate a Hyracks job that contains the right set of operators and connectors to process and move data into the target dataset and update its indexes (if any).

Continuing with our example *connect feed* statement from Listing 5.1, we look at how the equivalent insert statement is generated by the compiler. Later in this section,we will describe the differences that arise from the use of different kinds of UDFs (Java or AQL) associated with a feed. Listing 5.2 shows the basic template for constructing the equivalent insert statement. The actual insert statement after substituting the appropriate values into the template for our example

```
insert into dataset < target_dataset > (
  for $x in feed_intake ("<source_feed>")
  return $x
)
```

Listing 5.2: Basic template followed for rewriting a connect feed statement when the feed does not have an associated UDF

```
use dataverse feeds;

insert into dataset Tweets (
  for $x in feed_intake ("TwitterFeed")
  return $x
)
```

Listing 5.3: An equivalent insert statement constructed by the AsterixDB compiler following the template from Listing 5.2

is shown in Listing 5.3.

Optimized Physical Plan

The *insert* statement shown in Listing 5.3 is processed by the AsterixDB compiler to produce an optimized physical plan that is shown in Listing 5.4. The plan can be sub-divided into distinct stages — *intake* and *store*. We describe these stages next.

- **Intake Stage**: The intake stage at the beginning of the tail section involves retrieving the feed data records from the head section of the data ingestion pipeline. This task is performed by the *FeedIntake* operator. In order to retrieve data records produced by the head section of the ingestion pipeline, the FeedIntake operator requires its instances to be co-located with the respective instances of the FeedCollect operator that belong to the head section of the pipeline. Recall that the FeedCollect operator instances are *subscribable* and, as such, allow their output to be routed along multiple paths. Being co-located with a respective FeedCollect operator instance allows the FeedIntake operator instance to retrieve a handle

```
commit
-- COMMIT  |PARTITIONED|
  project  ([$$3])
  -- STREAM_PROJECT  |PARTITIONED|
    exchange
    -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
      insert  into  feeds:Tweets
      -- INSERT_DELETE  |PARTITIONED|
        exchange

----------STORE STAGE ----------
        HASH_PARTITION_EXCHANGE [$$5]  |PARTITIONED|
----------INTAKE STAGE ----------

          assign  [$$3] <- [function-call:  asterix : field -access-by-index
          -- ASSIGN  |PARTITIONED|
            exchange
            -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
              data-scan []<-[$$0] <- feeds:TwitterFeed
              -- DATASOURCE_SCAN  |PARTITIONED|
                exchange
                -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                  empty-tuple-source
                  -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
```

Listing 5.4: Optimized plan for a connect feed statement when the feed has no associated UDF

(using the local Feed Manager's search API) to the respective FeedCollect operator instance and to register itself as a recipient of data frames it outputs.

- **Store Stage**: Subsequently, as part of the store stage, the records output by the preceding intake stage are hash-partitioned on the basis of primary key and put into the target dataset; secondary indexes, if any, are also updated accordingly.[3]

Figure 5.4 provides a pictorial representation of tail section of the data ingestion pipeline. Our current example does not involve any pre-processing of feed records prior to their persistence. As such, the output from FeedIntake operator connects with the IndexInsert operator using an M:N hash-partitioning connector.

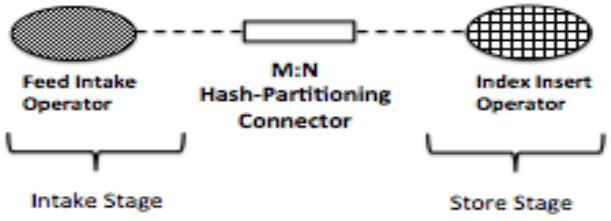


Figure 5.4: Tail section of a data ingestion pipeline

So far we have described the individual construction of the head and tail sections of a data ingestion pipeline in terms of their corresponding Hyracks jobs. Scheduling the tail section of the pipeline requires careful selection of its location constraints. Figure 5.5 shows the data ingestion pipeline wherein data is collected by the head section and processed thereafter by the connected tail section.

The *FeedIntake* operator instances from the tail section are set with location constraints that are identical to the *FeedCollect* operator instances from the head section. Being co-located with a FeedCollect operator instance allows the instance to discover the instance by querying the local Feed Manager and subsequently invoking the *subscribe* API provided by the *subscribable* instance.

[3]Note that secondary indexes in AsterixDB are partitioned and co-located with the corresponding primary index partition. The insert of a record into the primary and any secondary indexes uses write-ahead logging and offers record-level ACID semantics.

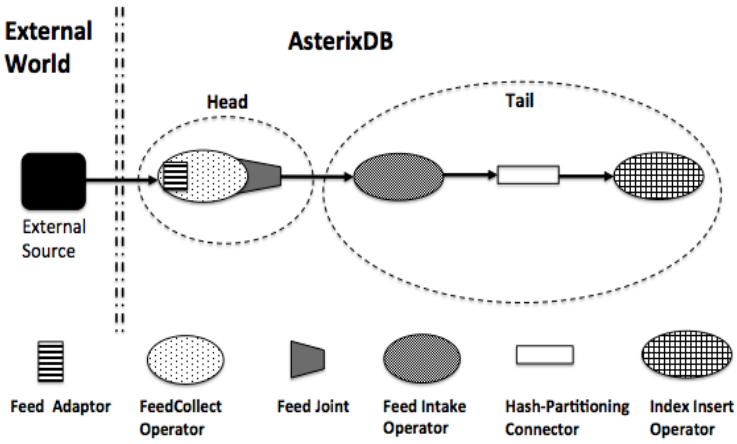


Figure 5.5: Data ingestion pipeline showing the flow of data between the head and the tail sections.

The remaining part of the tail section is the store stage, which is a set of IndexInsert operator instances. Each of these instances is co-located with a stored partition of the target dataset. In AsterixDB, the partition for a dataset exists on each AsterixDB node by default.[4]

Figure 5.6 shows an example physical layout of the data ingestion pipeline that involves 3 nodes from our example AsterixDB cluster. In this example, the FeedIntake operator instance is located at Node A and forms the head section or the intake stage for the data ingestion pipeline. An instance of the IndexInsert operator is located at each of the nodes B and C such that each is co-located with a dataset partition. Together these instances form the store stage and are a part of the tail section of the data ingesstion pipeline. The transfer of data between the head and the tail sections is facilitated by the FeedCollect operator instance that is co-located with the FeedIntake operator instance at Node A.

5.3.2 Secondary Feed with AQL UDF

Recall that different overlying data-driven applications may require the data from a feed to be processed differently and persisted in different datasets. To this effect, AsterixDB has the notion

[4]The set of AsterixDB nodes that store partitions of a dataset are collectively referred to as the *nodegroup* of the dataset.

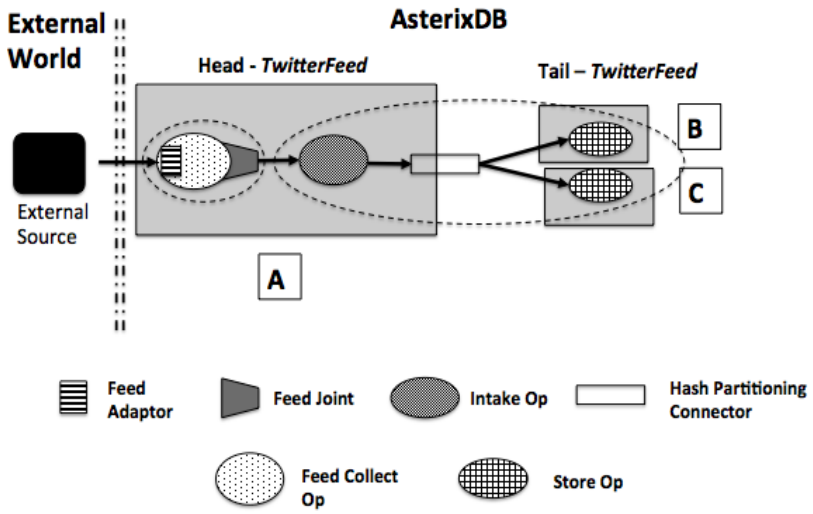


Figure 5.6: An example physical layout of the constructed data ingestion pipeline that involves 3 AsterixDB nodes

of a secondary feed that derives its data from another feed and can be independently processed and targeted at a different dataset. Previously, we described the construction of a data ingestion pipeline when a primary feed is connected to a dataset. Continuing with our example, we next define a secondary feed, one that derives its data from the primary feed, which involves additional pre-processing of data and requires the resulting data records to be persisted into a different target dataset. We first consider the case when an AQL function is required to be applied on each feed data record. Our example AQL function looks for the hashtags (if any) contained in the text of the tweet and collects them as an additional attribute (an ordered list) that is appended to the tweet. The required processing is contained in a user-defined AQL function, as shown in Listing 5.5.

Note that we have created a secondary feed *ProcessedTwitterFeed* and associated the created UDF using the *apply function* clause. We describe here, how the components – *head* and *tail* – for this data ingestion pipeline are constructed.

```
use dataverse feeds;

create function processTweet($x){
  let $topics := (for $token in word-tokens($x.message_text)
                  where starts-with($token, "#")
                  return $token)
  return {"id":$x.id,
          "user": $x.user,
          "latitude": $x.latitude,
          "longitude": $x.longitude,
          "created_at": $x.created_at,
          "message_text": $x.message_text,
          "country": $x.country,
          "topics": $topics}
};

create secondary feed ProcessedTwitterFeed from feed TwitterFeed
apply function processTweet;

connect feed ProcessedTwitterFeed to dataset ProcessedTweets;
```

Listing 5.5: An example of a feed with an associated AQL UDF.

Constructing the Head

Typically, the head section of a data ingestion pipeline involves the use of the feed adaptor to retrieve records from the external data source. However, given that the parent feed *TwitterFeed* is already in the connected state (from Figure 5.6), the task of interfacing with the external data source and retrieving data records need not be repeated. Thus, the head section does not need to be re-constructed. Instead, the data records flowing through the head section of the ingestion pipeline for *TwitterFeed* can be simultaneously routed to an additional path. These records can then flow through the newly constructed tail section for *ProcessedTwitterFeed* where they are processed further and eventually inserted into the target dataset, which is *ProcessedTweets* in our example. We thus reuse the existing head section of the parent feed pipeline and construct only the tail section for our secondary feed ingestion pipeline.

Constructing the Tail

In processing a *connect feed* statement that connects a secondary feed, the AsterixDB compiler identifies a *source feed* that can be used to derive the records constituting the secondary feed. In general, if $feed_{m+1}$ denotes a secondary feed that is a child of $feed_m$, then a feed $feed_i$ can be obtained from an ancestor feed $feed_k$ $(k < i)$ by subjecting each record from $feed_k$ to the sequence of UDFs associated with each child feed $feed_j$ $(j = k+1, ..., i)$. Note that $i-k$ denotes the 'distance' from $feed_k$ to $feed_i$ and is indicative of the additional processing steps (UDFs) required to produce $feed_i$ from $feed_k$. To minimize the processing involved in forming a feed, it is desired to source the feed from the nearest ancestor feed that is in the connected state. The feed joint(s) available along the ingestion pipeline of an ancestor feed are then used to access the flowing data and subject it to the additional processing needed to form the desired feed. AsterixDB keeps track of the available feed joints and uses them in preference over creating a new feed adaptor instance in sourcing a feed.

The AsterixDB compiler traverses the hierarchy of the secondary feed to find the closest ancestor feed that is in a *connected* state. In the case when none of the ancestor feeds are in a connected state, the source feed defaults to the primary feed that is at the root of the hierarchy. The AsterixDB compiler additionally constructs an ordered list of UDFs that need to be applied to the records from the *source feed* in order to obtain the desired secondary feed. We denote this ordered set of functions as $f_i(), i = 1, ..$N, where $f_1()$ is the UDF associated with the source feed, $f_N()$ is the UDF associated with the secondary feed that is being connected, and $f_j(), 1 < j < N$ are the UDFs corresponding to the other parent feeds in hierarchical order. The template followed is a slightly modified version of the template followed for a primary feed (shown earlier in Listing 5.2) and is shown in Listing 5.6.

As a first step towards constructing the tail section, the *connect feed* AQL statement is rewritten as an equivalent *insert* statement that is based on the template from Listing 5.6. The body of any

```
use dataverse feeds;

insert into dataset < target_dataset > (
  for $x in feed_intake ("<name_of_the_source_feed>")
  let $y{i}:= f{i}($x)
  let $y{i+1} := f{i+1}($y{i})
   ...
   ...
  let $y{N} := f{N}($y{N−1})
  return $y{N}
)
```

Listing 5.6: Generic template followed for constructing an equivalent insert statement in the case when a secondary feed is connected to a dataset.

AQL function is looked up from the AsterixDB Metadata and 'inlined' in the template to form the required *insert* statement, as shown in Listing 5.7. Note that the argument to the *feed_intake* built-in function is the parent feed - *TwitterFeed*. The *insert* statement from Listing 5.7 is compiled by the AsterixDB compiler to produce an optimized physical plan. Besides having an *intake* and a *store* stage, the resulting plan includes a ***compute stage*** that involves the application of the associated UDF to each feed record. Listing 5.8 illustrates the stages in the optimized plan.

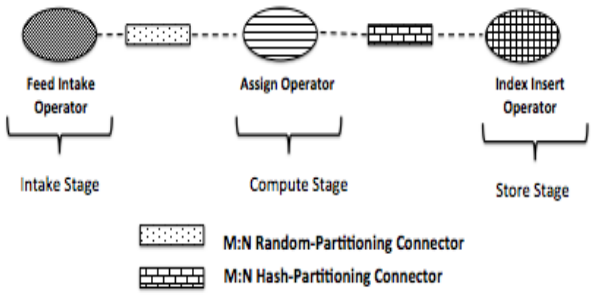


Figure 5.7: Tail section of a data ingestion pipeline when a feed involves a preprocessing UDF

Figure 5.7 gives a simplified representation of the constructed tail section. Output data from the *FeedIntake* operator is randomly partitioned across a set of *Assign* operator instances that form the *compute* stage. Each input record received by an *Assign* operator instance is processed by a subplan (not shown in the Figure) that corresponds to the body of the AQL function. The result from the

```
use dataverse feeds;

insert into dataset ProcessedTweets (
  for $x in feed_intake ("TwitterFeed")
  let $y:= let $topics :=
                (for $token in word-tokens($x.message_text)
                 where starts-with($token, "#")
                 return $token)
           return {"id":$x.id,
                   "user": $x.user,
                   " latitude ": $x. latitude ,
                   "longitude" : $x.longitude ,
                   " created_at " : $x. created_at ,
                   "message_text": $x.message_text,
                   "country": $x.country,
                   "topics": $topics}
  return $y
)
```

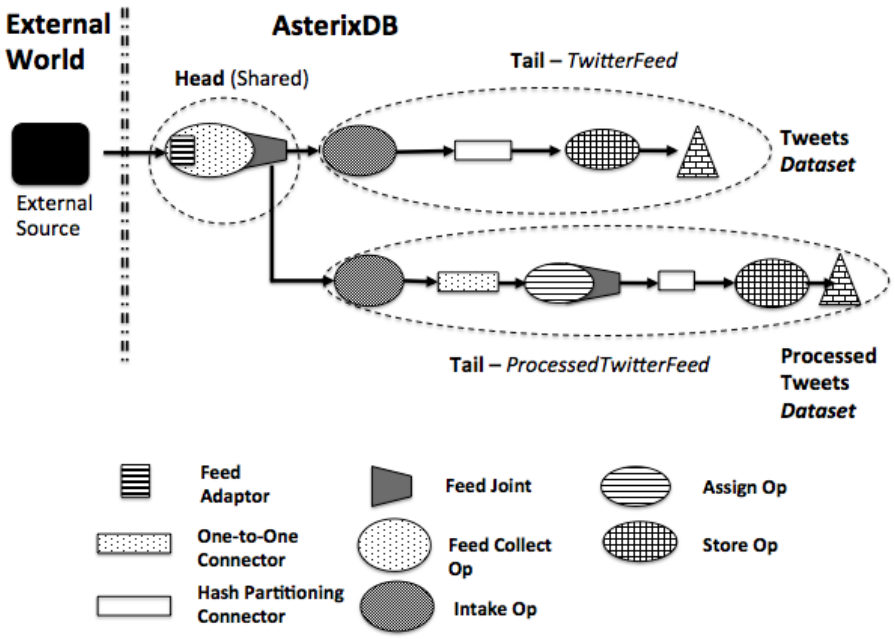
Listing 5.7: Equivalent ProcessedTweets insert statement for a connect feed statement (Listing 5.5) as constructed by the AsterixDB compiler

invocation of the function is hash-partitioned across a set of *IndexInsert* operator instances that form the *store* stage. At the store stage, records are inserted into the primary index partition and its secondary indexes (if any) are also updated.

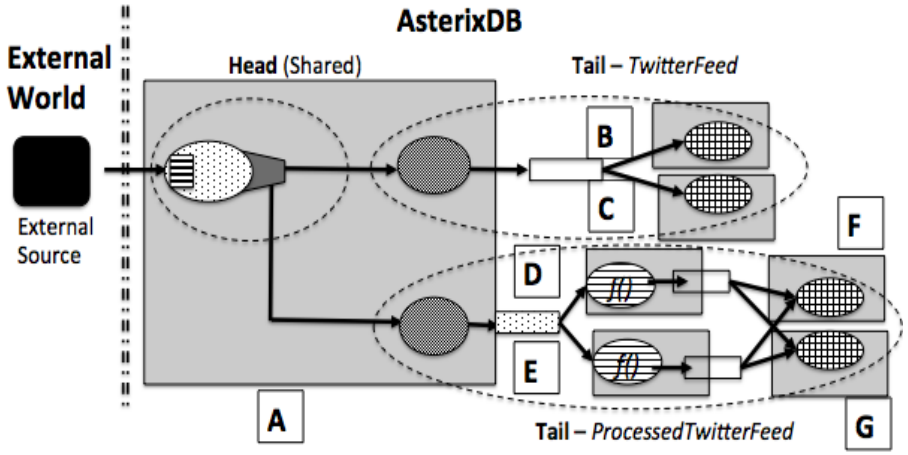
The Hyracks job representing the feed's tail section of the data ingestion pipeline is subsequently scheduled to run in a distributed fashion on an AsterixDB cluster. Note that the data ingestion pipelines for the parent *Twitter* feed and the descendant *ProcessedTwitterFeed* share a common head section but have separate tail sections to account for the differences in the way that records need to be processed prior to persistence. To facilitate data transfer and build a cascade network spanning the pair of ingestion pipelines, the count and location constraints for the FeedIntake operator instances are chosen carefully to coincide with the FeedCollect operator instances from the head section of the data ingestion pipeline for the parent *TwitterFeed*. The resulting cascade network is shown in Figure 5.8(a). Figure 5.8(b) shows an example physical layout of the cascade network that involves nodes A-G in our AsterixDB example cluster.

```
commit
-- COMMIT |PARTITIONED|
  project ([$$18])
  -- STREAM_PROJECT |PARTITIONED|
    exchange
    -- ONE_TO_ONE_EXCHANGE |PARTITIONED|
      insert into feeds:ProcessedTweets
      -- INSERT_DELETE |PARTITIONED|
        ------STORE STAGE ----------
        -- HASH_PARTITION_EXCHANGE [$$18] |PARTITIONED|
        ------COMPUTE STAGE ----------
    assign [$$18] <- [function-call: asterix : field -access-by-index]
        -- ASSIGN |PARTITIONED|
          project ([$$20])
          -- STREAM_PROJECT |PARTITIONED|
            assign [$$20] <- [function-call: asterix :cast-record,
            -- ASSIGN |PARTITIONED|
              project
              -- STREAM_PROJECT |PARTITIONED|
                subplan {
                    subplan {
                        aggregate [$$8] <- [function-call: asterix : listify ]
                        -- AGGREGATE |LOCAL|
                            select ( function-call: asterix : starts -with)
                            -- STREAM_SELECT |LOCAL|
                              unnest $$1 <- function-call: asterix :scan-
    collection , Args:[ function-call: asterix :word-tokens]
                                -- UNNEST |LOCAL|
                                  -- NESTED_TUPLE_SOURCE |LOCAL|
                    }
                  -- SUBPLAN |LOCAL|
                    nested tuple source
                    -- NESTED_TUPLE_SOURCE |LOCAL|
                }
            -- SUBPLAN |PARTITIONED|
              assign <- [function-call: asterix : field -access-by-index]
              -- ASSIGN |PARTITIONED|
                ------COMPUTE STAGE ----------
                -- RANDOM_PARTITION_EXCHANGE |PARTITIONED|
                ----------INTAKE STAGE----------
                  data-scan []<-[$$0] <- feeds:TwitterFeed
                  -- DATASOURCE_SCAN |PARTITIONED|
                    exchange
                    -- ONE_TO_ONE_EXCHANGE |PARTITIONED|
                      empty-tuple-source
                      -- EMPTY_TUPLE_SOURCE |PARTITIONED|
```

Listing 5.8: Optimized plan to the tail section of a data ingestion pipeline when a feed involves an AQL UDF (from Listing 5.5)



(a) Cascade network constructed from a shared head section and separate tail sections for *TwitterFeed* and *ProcessedTwitterFeed*



(b) An example physical layout of the cascade network (from Figure 5.8(a)) that involves given AsterixDB nodes

Figure 5.8: Physical Layout: Cascade network involving the *TwitterFeed* and *ProcessedTwitterFeed* with a shared head section and separate tail sections

Note that the output from the compute stage constitutes the records that define the *ProcessedTwitterFeed*. This secondary feed could itself be used as a parent for other descendant feeds. It is thus required that the output feed records from the compute stage are made accessible for a possible routing, in the event that the end-user chooses to create and connect a descendant of *ProcessedTwit-*

terFeed. The *Assign* operator instances use a feed joint at their output and register themselves with the local Feed Manager using the ID - "TwitterFeed:processTweets". In a data ingestion pipeline, a feed joint is placed at a location where records that constitute the feed are being output. For a data ingestion pipeline that does not involve a UDF, this happens at the output of the *intake* stage. In the other case, a feed's records are produced at the output of its *compute* stage.[5]

5.3.3 Feed with a Java UDF

A secondary feed is similar to a primary feed in the sense that it too can act as a parent feed for other feeds. The data ingestion pipeline for the secondary *ProcessedTwitterFeed* (refer to Figure 5.8(a)) offers subscribable instances at its compute stage. To illustrate how these subscribable instances may be used to route data along multiple paths, we define an example secondary feed, *SentimentFeed* that extends the *ProcessedTwitterFeed* and subjects the records to additional processing before persisting them in a target dataset. The additional pre-processing involves computing a sentiment (a double value $\in [0, 1]$) associated with the text of the tweet and adding it as an additional attribute to the tweet. Such pre-processing is not feasible via AQL but is expressed by composing a UDF in a programming language such as Java and plugging-in the function (a.k.a. installing the function) via the AsterixDB external library feature. A detailed tutorial on using Java UDFs with AsterixDB is provided in Appendix A. For now, we assume the availability of such a UDF and associate it with our secondary feed, *SentimentFeed*. The set of AQL statements that define our secondary feed and connect it to a target dataset is shown in Listing 5.9. Note that a Java UDF is referred to by its fully qualified name, which includes the name of the containing library (*tweetlib* in the example shown in Listing 5.9).

By definition, the records for the *SentimentFeed* can be obtained by accessing the parent feed — *ProcessedFeed* — and applying the Java UDF (*sentimentAnalysis*) to each record. As the *Pro-*

[5]Figure 5.8(b) shows a generic case. However, the default placement in an AsterixDB cluster would have a partition of the target dataset at each worker node and a co-located instance of the store operator.

```
use dataverse feeds;

create secondary feed SentimentFeed from ProcessedTwitterFeed (
apply function "tweetlib#sentimentAnalysis";

connect feed SentimentFeed to dataset TwitterSentiments;
```

Listing 5.9: An example of a feed with an associated Java UDF.

cessedFeed is already in a connected state, building the data ingestion pipeline for *SentimentFeed* does not require constructing a separate head section. We shift our focus to construction of the tail section of the data ingestion pipeline and bring out the subtle differences introduced due to the involvement of a Java UDF as against an AQL UDF.

Tail Section

In translating the *connect feed* statement to an equivalent *insert* statement, the AsterixDB compiler uses the template, described earlier in Listing 5.6. The resulting *insert* statement is shown in Listing 5.11. Note that the secondary feed *ProcessedTwitterFeed* acts as the source feed and is passed as the argument to the *feed_intake* built-in function. The optimized plan is shown in Figure 5.9. Note that the Java UDF is treated as a black box as the compiler has very limited understanding of its semantics. Unlike the case of an AQL function, the AsterixDB compiler does not have an opportunity to optimize application of a Java UDF by the use of indexes etc. The tail section of the data ingestion pipeline for *SentimentFeed* can be pictorially represented in a similar way, as shown in Figure 5.7.

The tail section for the data ingestion pipeline is fed with data that is being produced at the output side of the compute stage of its parent *ProcessedTwitterFeed*. A logical view of this flow of data and the state of the cascade network is show in Figure 5.9(a). Figure 5.9(b) presents a physical view showing an example placement of different operator instances across the nodes in our ex-

```
use dataverse feeds;

insert into dataset TwitterSentiments (
  for $x in feed_intake ("ProcesseedTwitterFeed")
  let $y:= tweetlib#sentimentAnalysis ($x)
  return $y
)
```

Listing 5.10: An equivalent TwitterSentiments insert statement for the connect feed statement shown in Listing 5.9

```
commit
-- COMMIT |PARTITIONED|
   project ([$$5])
  -- STREAM_PROJECT |PARTITIONED|
     exchange
    -- ONE_TO_ONE_EXCHANGE |PARTITIONED|
       insert into feeds:SentimentTweets
      -- INSERT_DELETE |PARTITIONED|

---------------STORE STAGE ------------------------
        -- HASH_PARTITION_EXCHANGE [$$5] |PARTITIONED|
---------------COMPUTE STAGE ----------------------

           assign [$$5] <- [function-call: asterix : field -access-by-index
           -- ASSIGN |PARTITIONED|
              project ([$$1])
             -- STREAM_PROJECT |PARTITIONED|
                assign [$$1] <- [function-call: feeds : tweetlib#sentimentAnalysis
                -- ASSIGN |PARTITIONED|

---------------COMPUTE STAGE ----------------------
                  -- RANDOM_PARTITION_EXCHANGE |PARTITIONED|
---------------INTAKE STAGE -----------------------

                   data-scan []<-[$$0] <- feeds:ProcessedTwitterFeed
                    -- DATASOURCE_SCAN |PARTITIONED|
                       exchange
                       -- ONE_TO_ONE_EXCHANGE |PARTITIONED|
                          empty-tuple-source
                          -- EMPTY_TUPLE_SOURCE |PARTITIONED|
```

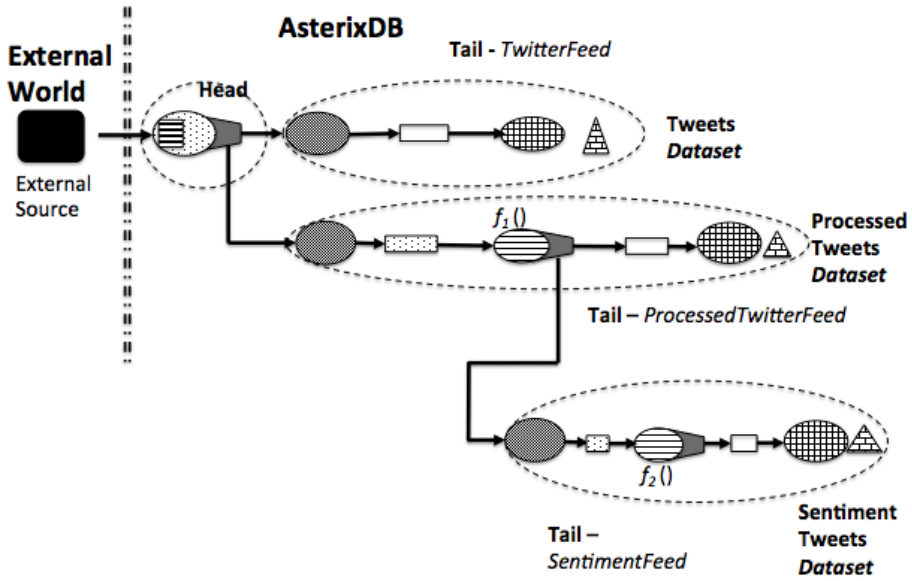
Listing 5.11: Optimized plan for the tail section of the data ingestion pipeline when the feed involves a Java UDF (Listing 5.9)

ample AsterixDB cluster.[6] The FeedIntake job, as constructed by the AsterixDB compiler, has its FeedIntake operator instances strategically co-located with the subscribable Assign operator instances from the compute stage of *ProcessedTwitterFeed*. Recall that the Assign operator instances are discoverable by the search API provided by the local Feed Manager. Note that each Assign operator instance on the ingestion pipeline for *SentimentFeed* has a feed joint at its output side. The feed joint makes the data records constituting the *SentimentFeed* to be accessible and allows these records to be routed along additional path(s) when the cascade network is expanded. For example, if an end-user creates a secondary feed as a descendant of *SentimentFeed*, then the constructed ingestion pipeline will use the this set of feed joints as their source(s) of data.

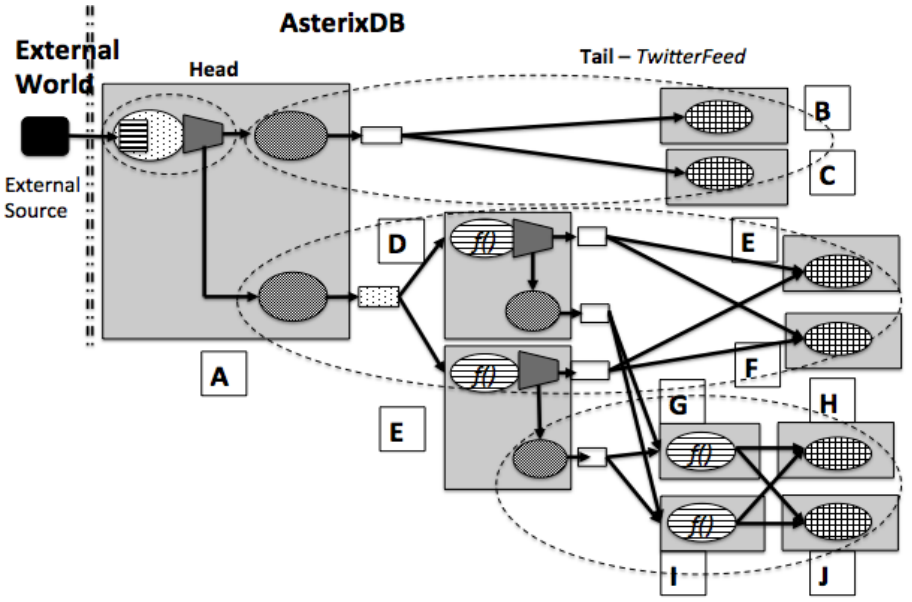
5.4 Inside a Feed Joint

A feed joint has loose similarity to a subscription service that allows interested subscribers to register interest in data, i.e., feed records, and subsequently receive and process them. Implementation-wise, a feed joint is a shared queue attached at the end of an operator such that all data frames output by the operator are deposited into the queue. These data frames are required to be routed to a set of recipient operator instances that have registered as 'subscribers' and act as a bridge for data to flow from an ingestion pipeline to another. Each Node Controller (NC) has an associated *Feed Manager*, a data structure that holds all runtime metadata about the active components of a data ingestion pipeline that are hosted by the NC. This metadata includes the set of operator instances and the available feed joints. An *intake* operator instance uses the Feed Manager API to gain access to an available feed joint and register itself as a subscriber using an API provided by the feed joint. Thereafter, data flowing through the feed joint begins to be routed to the subscriber operator instance.

[6]The default placement puts a store operator instance at each worker node such that it is co-located with a respective partition of the target dataset. Recall that, by default, a dataset has a partition on each worker node.



(a) Logical View: Cascade network showing a shared head section and separate tail sections for *TwitterFeed*, *ProcessedTwitterFeed* and *SentimentFeed*



(b) Physical View: Physical layout of the cascade network (Figure 5.9(a)) involving eleven AsterixDB nodes

Figure 5.9: Logical and physical view of a cascade network involving *TwitterFeed*, *ProcessedTwitterFeed* and *SentimentFeed*

5.4.1 Modes of Operation

A feed join may operate in two possible modes, namely — *Shared* mode and *Short-Circuited* mode, which are described next.

1. **Shared Mode**:

   A feed joint operates in a *shared* mode when the data flowing through it is required to be routed to multiple paths in a concurrent fashion. Recall that records output by an operator are packaged into fixed-size chunks known as data frames. When operating in a shared mode, a feed joint must ensure the following:

   - *Guaranteed Delivery*:

     A feed joint must ensure that a data frame is received by each subscribing operator instance. In providing such guarantee it must not require the subscribers to operate in a synchronized lock-step mode, but should instead allow each consuming operator to operate at its own pace.

   - *Congestion Isolation*:

     A feed just must ensure that sluggish or slow movement of data[7] along a particular path does not impede the flow of data across other path(s).

   In order to achieve the above objectives (Guaranteed Delivery and Congestion Isolation), a feed joint tracks a data frame by wrapping it in a holder object (referred hereafter as a *Data Bucket*). A Data Bucket additionally contains a counter that is initialized to the number of registered subscribers that must consume the content (data frame) inside the bucket. Each subscriber has an associated input queue that delivers the next data frame to be consumed. Subsequent to wrapping a data frame in a data bucket, the feed joint deposits the data bucket to the input queue associated with each subscriber. Note that a single instance of Data

[7] Sluggish of slow movement of data along a path is typically the result of resource-intensive computation or processing being applied along any of the downstream operator instances on the path.

Bucket is being 'shared' by all subscribers. Consumption of a Data Bucket (processing of its content) by the subscriber is done in an *asynchronous* manner. This ensures congestion isolation, as delays in the processing of a Data Bucket by a subscriber does not prevent other subscribers from consuming other Data Buckets from their respective input queues. Each subscriber extracts the data frame contained inside the data bucket at its own pace and decrements the associated counter when it is done processing the data frame. The Data Bucket continues to stay in the queue until it has been processed by all the subscribers, i.e., the counter reaches zero. This ensures guaranteed delivery. Subsequently, the Data Bucket is reclaimed and returned to a pool only to be retrieved later and initialized with another output frame from the producing operator.

2. **Single or Short-Circuited Mode**:

   In the case when a feed joint has a single subscriber, it doesn't need to do any sort of book-keeping or tracking of data frames to ensure Congestion Isolation or Guaranteed Delivery. Thus, the working of a feed joint is significantly simplified. A feed joint then does not use a Data Bucket to wrap a data frame; instead, data frames are sent to the receiving operator in a synchronous manner. This mode is also known as the *Short-Circuited* mode.

   The mode of operation (single or short-circuited) for a feed joint is dynamically determined in accordance with the number of associated subscribers, which may increase or decrease dynamically.

5.5 Disconnecting a feed

So far, we have described in detail the methodology adopted for constructing a data ingestion pipeline when a given feed is connected to a target dataset. We illustrated the *Fetch Once, Compute Many Model* and described how the AsterixDB compiler optimizes the construction of a data ingestion pipeline by reusing computations done as part of ingestion of other feeds related in hier-

archy. Next, we shift our focus to what is semantically the opposite task – terminating the flow of data by disconnecting a feed from its target dataset. This task is supported by the *disconnect feed* AQL statement.

Let us assume that our example set of feeds (*TwitterFeed*, *ProcessedTwitterFeed*, and *Sentiment-Feed*) are all in the connected state. Beginning with the cascade network that was shown in Figure 5.9(a), we disconnect the feed *TwitterFeed* from the dataset *Tweets*. The AQL statement is shown in Listing 5.12. At the time of submission of a *disconnect* statement, the operator instances on a data ingestion pipeline are involved in continuous processing of arriving records and pushing them downstream. As the system is not at rest, we expect in-flight data records traveling across the wire to reach destination operator instances for further processing until they eventually reach the dataset index. Disconnecting a feed essentially translates to terminating this flow of data from the external source to the target dataset. Disconnecting a feed is designed to be graceful in the sense that AsterixDB will stop receiving data from external source but allow the already received but unprocessed records to traverse the length of the ingestion pipeline and reach the target dataset.

AsterixDB keeps track of the set of locations (AsterixDB nodes) where operators from each stage are running. The task of disconnecting a feed requires the FeedIntake operator instances for the ingestion pipeline to discontinue receiving any more data and convey the end of the data flow to the downstream operator instances. Recall that FeedIntake operator instances receive data from a co-located *subscribable* operator instance, which is either a FeedCollect operator instance (from the intake stage of the source feed) or an Assign operator instance (from the compute stage of the source feed). Contrary to the *subscribe* API, a *subscribable* operator instance also provides the *unsubscribe* API to allow registered subscribers to discontinue receiving data. The FeedIntake operator instances invoke this method but ensure that records in their respective input buffers have been pushed downstream. Subsequently each FeedIntake operator instance invokes the *close()* method on the downstram operator instance, indicating the intent to terminate the flow of data. This sequence of processing the input buffer, sending records downstream, and invoking the *close()*

method is repeated by each successive operator instance in the ingestion pipeline. Subsequent to a invocation of the *close()* method, an operator instance terminates.

It is worth noting that disconnecting *TwitterFeed* from dataset *Tweets* does not impact the flow of data along other pipelines in the cascade network. This is because the subscribers that receive data from a shared feed joint operate in an independent manner without the need of any synchronization amongst each other. As such, the registered set of subscribers continue to receive data frames (each wrapped inside a Data Bucket) while the unregistered subscriber simply ceases to receive any additional Data Buckets in its input queue.

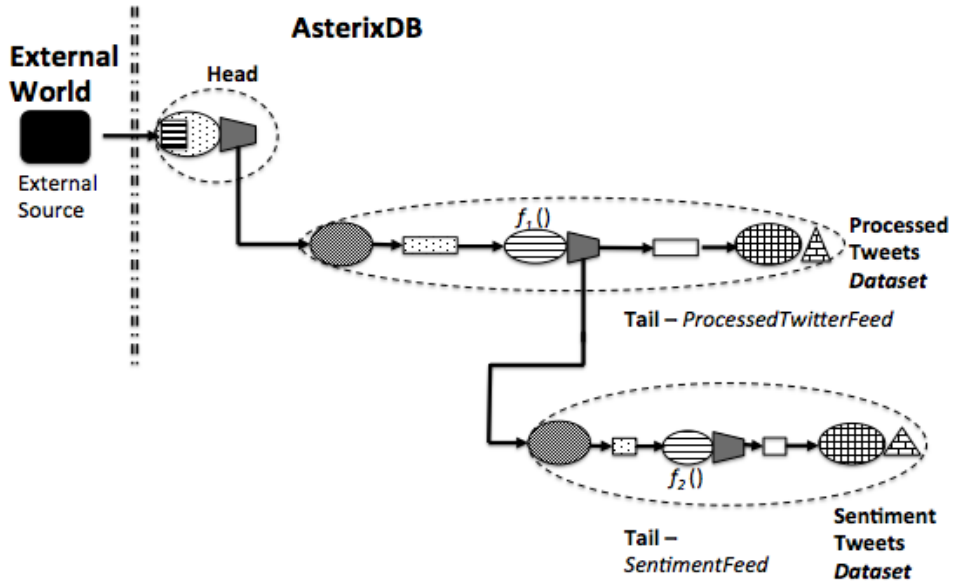
The resulting cascade network, after disconnecting *TwitterFeed*, is shown in Figure 5.10(a). At this stage, let us assume that the secondary feed *ProcessedTwitterFeed* needs to be disconnected as well now. This is achieved by the use of a *disconnect feed* AQL statement that is similar to Listing 5.12. Recall that *ProcessedTwitterFeed* is the parent feed for *SentimentFeed*, so the Assign operator instances on the ingestion pipeline are serving as the source of data records that are consumed by the ingestion pipeline for *SentimentFeed*. Disconnecting *ProcessedTweets* causes only partial dismantling of its ingestion pipeline, as the *IndexInsert* operator instances are allowed to terminate but the *Assign* operator instances continue to stay alive and process records. The resulting flow of data is shown in Figure 5.10(b).

Note that if at this stage, we choose to connect *ProcessedTwitterFeed* to its respective target dataset (*ProcessTweets*) as before, the head section for the ingestion pipeline would not have to be re-constructed. For feeds that are form a hierarchy, connecting *any* feed along the lineage to a dataset results in constructing the head section that provides a feed joint for the arriving records to be routed along multiple paths. For example, if we do connect *ProcessedTwitterFeed* to a dataset again, the resulting cascade network would be identical to one shown in Figure 5.10(a).

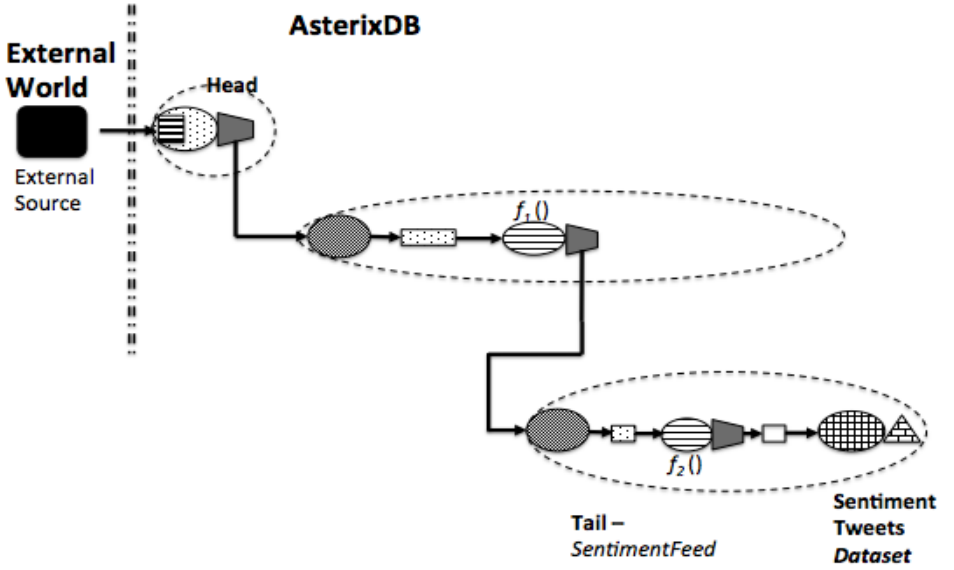
```
use dataverse feeds;

disconnect feed TwitterFeed from dataset Tweets;
```

Listing 5.12: AQL statement for disconnecting TwitterFeed from its target dataset.



(a) Data flow after disconnecting the primary feed *TwitterFeed*



(b) Data flow after disconnecting the secondary feed *ProcessedTwitterFeed*

Figure 5.10: Runtime modifications to a cascade network when participant feeds are diconnected

5.6 At Least Once Semantics

An application may demand stronger guarantees on the processing of records by requiring each arriving record to be processed *at least once* through the ingestion pipeline, despite any failures. Such a requirement is expressed through the *at.least.once.enabled* policy parameter. To provide *at least once semantics*, each record arriving from the data source is augmented with a *tracking id* at the intake stage. Subsequent to persisting a record (log record has been written to the local disk), the store operator instance constructs an *ack* message with the *tracking id*. Over a fixed-width time-window, the ack messages for all records that were sourced from a given feed adaptor instance (identified from the tracking id) are grouped and encoded together as a single message by each store operator instance. Grouping of multiple ack messages and subsequent encoding reduces the number of bytes exchanged over the network. A record that has been output by the intake stage is held at its intake node until an ack message for the record is received from the store stage. When an ack is received, the record is dropped and memory is reclaimed. On a timeout, the records without an ack are replayed. *At least once* semantics are not guaranteed if *throttling* or *discarding* of records is enabled by the policy.

5.7 Experimental Evaluation

Next, we present an experimental evaluation of the support for data ingestion in AsterixDB. Our evaluation begins by emphasizing the benefits derived from continuous ingestion over the alternative approach of batch inserts. We then shift focus to the flexibility offered by AsterixDB in constructing a cascade network of feeds and demonstrate the performance benefits thereof. We conclude our evaluation by measuring the ability of our system to scale and ingest an increasingly larger volume of data through the addition of resources.

Experimental Setup: We ran experiments on a 10-node IBM x3650 cluster. Each node had one

```
<workload xmlns=”workload”>
 <repeat>
   < iterations >5</iterations >
   <pattern>
      <duration>400</duration>
      <rate>300</rate>
   </pattern >
   <pattern>
      <duration>400</duration>
      <rate>600</rate>
   </pattern >
 </repeat>
</workload>
```

Listing 5.13: An example pattern descriptor that defines the specific pattern to be followed for generation of data by TweetGen

Intel 2.26GHz processor, 8GB of RAM, and a 300GB hard disk. The following were the steps taken to prepare the experimental setup.

- ***Modeling a Continuous External Data Source***: We wrote a custom tweet generator, hereafter referred to as *TweetGen*. TweetGen runs as a standalone process (JVM) and can be configured to output synthetic but meaningful tweets (in JSON format). TweetGen allows configuring the pattern for data generation with a predefined rate of generation of tweets (tweets/sec or **twps**) and respective time intervals. The Tweet datatype from Listing 3.2 from Chapter 3 showed the ADM representation for a tweet output by TweetGen. TweetGen listens for a request for data at a pre-determined port that is passed as an argument. Additionally, an XML file (referred to as a ‘pattern descriptor’) that describes the pattern to be followed for generating tweets is provided as an argument. Listing 5.13 shows an example of a pattern description XML file. The example pattern described there defines a cycle with two 400 second intervals with the respective rates of generation of tweets being 300 twps and 600 twps. As defined in the descriptor, the cycle is repeated 5 times. In our experiments, we shall use different versions of the pattern descriptor XML to vary the workload.

  Initiating the generation and the flow of data requires an initial handshake (by an interested

receiver) subsequent to which data is "pushed" to the receiver at a constant rate (twps).

- ***Creating a feed***: To ingest data from TweetGen, we used a custom socket-based adaptor, *TweetGenAdaptor*. The adaptor is configured with the location(s) (socket addresses) where instance(s) of TweetGen is/are running.

5.7.1 Batch Inserts versus Data Ingestion

An alternative mechanism for putting a large amount of data into a data store is to use the conventional *insert* statement on a batch of records and do so in a repeated fashion at an application level to achieve continuous ingestion. This is what database applications must do today in lieu of a feed facility. Such an approach is likely to incur delays due to overheads associated with the execution of each standalone insert statement which includes the costs of statement compilation and execution. We experimentally evaluated the two mechanisms — batch inserts and continuous data ingestion. Our goal was to experimentally determine the more efficient method in terms of time consumed in putting a given set of records into a target dataset inside AsterixDB.

We used ADM to model an end-user of a social networking website. For the scenario in this experiment, Listing 5.14 shows the AQL statements that define the end-user type (UserType) and a dataset to hold data about each user. For the target dataset, we generated a total of 590 million records that measured 162 GB in terms of size. These records were loaded into the target dataset (Users from Listing 5.14) up front using the *load dataset* AQL statement. The experiment itself then aimed at continuously ingesting a total of 2.2 GB worth of additional ADM records (8,185,185 records) into the pre-populated dataset. The ingestion task can be performed by using (batch) inserts or by defining and creating a data feed that consists of records to be inserted:

- **Batch Inserts**: A given number of records n, which is the batch-size, is read and an insert statement is constructed that follows the template shown in Listing 5.15. The insert statement

```
use dataverse feeds;

create type EmploymentType as {
 organization_name: string,
 start_date : date,
 end_date: date?
}

create type UserType as {
 id: int64,
 id_copy: int64,
  alias : string,
 name: string,
  user_since : datetime,
  user_since_copy : datetime,
  friend_ids : {{ int64 }},
 employment: [EmploymentType]
}

create dataset Users(UserType)
primary key id;

create index usrSinceIdx on Users( user_since );
```

Listing 5.14: AQL statements to define datatypes and datasets for experimental evaluation and comparison of batch inserts and continuous data ingestion

```
use dataverse feeds;

insert into dataset Users
(
 for $t in [
  <FULL_ADM_REC1>,
  <FULL_ADM_REC2>,
  . . .,
  <FULL_ADM_RECn>
 ]
 return $t
);
```

Listing 5.15: A template for insert statement that includes a batch of records

is executed and a new statement is constructed using a fresh batch of records. This cycle is repeated until all records have been processed. The number of iterations is dependent on the batch-size, n. Each iteration incurs the cost of compilation of the insert statement to produce the required Hyracks job and the execution cost (setup and cleanup, etc.) of the job. The cost involved in creating the insert statement per batch is increasingly amortized as the batch size is increased.

Method for Experimentation:

The records to be inserted were generated a priori and stored as a single file on disk. We wrote a synthetic stand-alone Java application that parsed the file to produce a fixed-size batch of records, construct and submit an insert statement, and wait for its execution before constructing the next batch and so on until all records contained in the file are processed.

- **Continuous Data Ingestion**: The records to be inserted are received by a feed adaptor and pushed downstream along a data ingestion pipeline into the target dataset. This method incurs the initial cost of setting up the data ingestion pipeline which is amortized over the entire set of records received and inserted.

  Method for Experimentation:

  In practice, a data feed is obtained from an external source that generates data in a continuous

```
use dataverse feeds;

create feed UsersOnDisk using file_based_feed (
("type_name"= "UserType"),
("format"= "adm"),
("path"= "<absolute path of source file >")
}

connect feed UsersOnDisk to dataset Users;
```

Listing 5.16: AQL statements to define and connect our example feed used for experimental evaluation

manner and provides a push- or a pull-based method for retrieving the data records. For experimental evaluation, we created a simulated feed that utilized the disk-resident data file as an external data source and required writing a custom adaptor that received records by parsing the content of a given file.[8] The set of AQL statements that define and connect the feed to a target dataset is shown in Listing 5.16. The adaptor (referred to in Listing 5.16 by its alias – file_based_feed) accepts the path of the file containing data records as a configuration parameter. We defined the feed to make use of the built-in adaptor in ingesting the set of 8 million records (2.2 GB worth of data) that formed the insert workload for the experiment.

Table 5.1: Execution time for different methods for insertion of records

Method	Avg time per record (msec)
Batch Insert (Batch Size = 1)	73.75
Batch Insert (Batch Size = 20)	6.2
Data Feed	0.03

We measured the average time taken to insert a record using a batch insert method (with varying batch size) or using a feed. Table 5.1 summarizes the results. The tasks of compiling an insert

[8]The adaptor served the purpose of simulating a data feed and was written strictly for experimental evaluation.

statement to form a Hyracks job, scheduling the job, and performing the job cleanup after completion represent a significant overhead. A batch size of 1 is the worst case since as many individual insert statements are required as the number of records, maximizing the negative impact of the per-statement overhead involved in the insert approach. As the batch size is increased to 20, the number of individual insert statements required reduces by that factor thus reducing the overhead via amortization. In principle, setting the batch size to as high as the number of records to be inserted should result in minimum execution time as a single Hyracks job needs to be constructed and scheduled. The overhead is minimal. However, this is not feasible, as in practice, the set of records that need to be inserted is derived from a continuously generating data source and is thus not finite. Furthermore, it is not practical or efficient to form large-sized batches from continuously arriving records as these would need to be held in memory to form a batch before it is complete and can be processed.

The data feeds mechanism provides the most efficient alternative to ingest data with an average execution time per record of less than a tenth of a millisecond which improves over the insert via batch (size = 20) approach by **two orders of magnitude**. The data feeds mechanism gains by incurring an initial fixed-cost for setting up a data ingestion pipeline and then using it to insert a potentially never-ending sequence of records. Such an approach eliminates the intermediate costs involved in compiling an insert statement, scheduling the resulting job, and cleaning up the job after its execution. Data sources that generate data in a continuous manner are ubiquitous, and the conventional way of batch-inserts does not fit well with such sources of data, and these performance results make it clear that that a data management system should provide a mechanism for "continuous inserts" vias a built-in support for data feeds.

5.7.2 Fetch Once, Compute Many Model

The support for data ingestion in AsterixDB provides a flexible Fetch Once, Compute Many model that allows routing data arriving from given data feed along multiple paths, applying different computations or transformations along each path to form additional secondary feeds that can be persisted in different datasets. Such a dataflow is referred to as a "*cascade* network and its net effect is to transform and persist data from a given data source in different ways and do so with sharing and in parallel. An alternate way of achieving the same end-result would be to establish multiple independent connections with the external source, apply transformations to data arriving on each connection, and persist the result into a separate dataset. Such a basic dataflow is referred to as an *independent* network as data flows between the external source and target datasets along independent paths. We provide here an experimental evaluation that demonstrates the performance benefits of having and using the 'Fetch Once Compute Many' model by comparing a cascade network of feeds to feeding data from an external source along multiple independent paths.

Figure 5.11 shows a cascade network configuration with the pair of feeds ($Feed_A$ and $Feed_B$), each applying the required transformation, prior to persistence of records. The alternate configuration (an independent network) that is logically equivalent to the cascade network configuration from Figure 5.11 is shown in Figure 5.12. In either configuration, the goal is to receive data records from an external source in a continuous manner, pre-process the records using two different sequences of computations, and persist the resulting records from each sequence into the respective target dataset. Note that the figure's different sets of computations have an overlap such that one can be considered as an extension of the other.

Method for Experimentation:

We modeled an external data source using our custom data generator, TweetGen and defined $Feed_A$ in the figures as a primary feed that makes use of the TweetGen adaptor to retrieve data from TweetGen. $Feed_A$ involved the application of a UDF, $f_1()$, prior to persistence of records. $Feed_B$

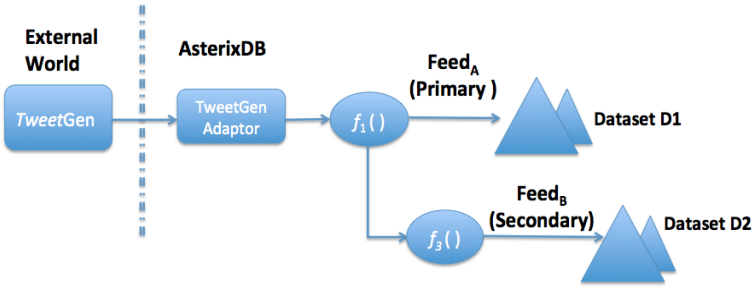


Figure 5.11: Cascade Network: Dataflow constructed using a cascade network with common connection to external data source

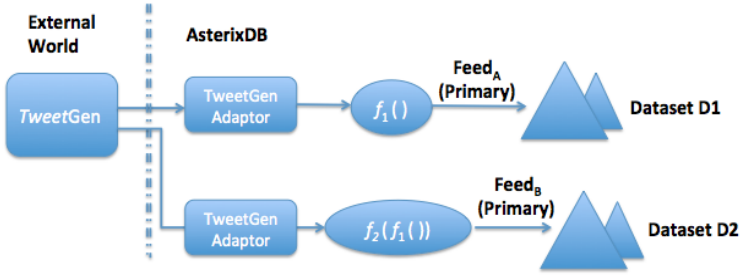


Figure 5.12: Indpendent Network: Dataflow constructed using separate connections to the external data source

required the output records to be subjected to further processing, which is expressed as $f_2()$. Note that in the cascade network (Figure 5.11), $Feed_A$ and $Feed_B$ share the computation of $f_1()$. For each record emitted by TweetGen, $f_1()$ and $f_2()$ are each invoked once. In contrast, for a given record emitted by TweetGen, the independent network from Figure 5.12 involves the invocation of $f_1()$ twice (once for each $Feed_A$ and $Feed_B$) and an additional invocation of $f_2()$ along $Feed_B$. The potential performance gain due to the sharing of computation in Figure 5.11 is dependent on the relative complexity of $f_1()$ in comparison to $f_2()$.

For our experiment here, the functions $f_1()$ and $f_2()$ were modeled to have a varying degree of computational intensity. One extreme case is when $f_1()$ is an extremely lightweight function while $f_2()$ is computationally expensive and can potentially bottleneck on the availability of CPU cycles. The opposite case is when $f_1()$ is computationally intensive and $f_2()$ is extremely lightweight; this scenario forms the other extreme end of the sharing opportunity spectrum. To alter the computational complexity of a UDF ($f_1()$ or $f_2()$), we defined a synthetic Java UDF to involve a busy spin

```
use dataverse feeds;

create feed Feed_A using TweetGen
(("server"="10.1.0.1:9000"))
apply function tweetlib#f1;

create secondary feed Feed_B from feed Feed_A
apply function tweetlib#f2;

connect feed Feed_A to dataset D1 using policy Discard;
connect feed Feed_B to dataset D2 using policy Discard;
```

Listing 5.17: AQL statements to construct the cascade network configuration from Figure 5.11

```
use dataverse feeds;

create feed Feed_A using TweetGen
(("server"="10.1.0.1:9000"))
apply function tweetlib#f1;

create feed Feed_B using TweetGen
(("server"="10.1.0.1:9000"))
apply function tweetlib#f3; // f3 is semantically equivalent to f2(f1)

connect feed Feed_A to dataset D1 using policy Discard;
connect feed Feed_B to dataset D2 using policy Discard;
```

Listing 5.18: AQL statements to construct the cascade network configuration from Figure 5.12

loop that runs for a given number of iterations and increments a long value in each iteration. A higher number of iterations yields an increased computational cost and a longer delay introduced per invocation of the function.

For the experimental environment, our test cluster involved a set of 10 worker nodes, each having two physical cores. We configured TweetGen to generate data at a sufficiently high rate such that at each compute node, the data arrival rate exceeds the rate at which the records can be processed. To ensure that the excess records on each node do not increase the memory footprint and introduce non-determinism (via arbitrary triggering of garbage collection), we selected the 'Discard' ingestion policy (refer to Table 4.2 from Chapter 4) to discard the excess records. As a result, our

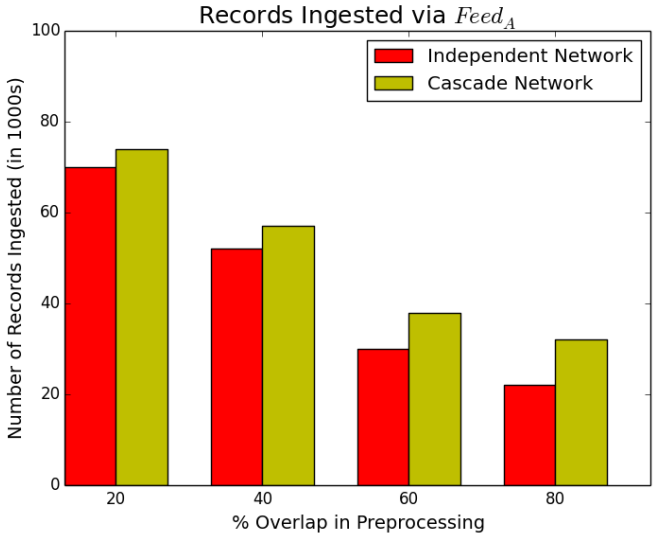
performance metric is the number of records successfully ingested in a given fixed time window during which the data source (TweetGen) emitted records in a continuous fashion. Recall that the compute stage in a data ingestion pipeline applies the associated UDF in an embarrassingly parallel manner. The default degree of parallelism used is the same as the number of available worker nodes. Each node then hosted an instance of the compute operator that applied the UDF to each input record.

Listing 5.17 shows the set of AQL statements used to construct the cascade network configuration from Figure 5.11. The corresponding set of AQL statements for constructing the independent network configuration is shown in Listing 5.18. TweetGen was configured to run for a fixed duration of 400 seconds and generated tweets at 5000 twps. At the end of the 400 second period, we measured the number of records persisted in each of the target datasets D1 and D2. We varied the relative computational complexity of $f_1()$ and $f_2()$ as shown in Table 5.2. Recall that by definition, $Feed_A$ is formed by applying $f_1()$ on each record received from TweetGen, while $Feed_B$ is formed by applying $f_2(f_1())$ per record. We represent the composition $f_2(f_1())$ as yet another function – $f_3()$ – as seen in Figure 5.12. Notice that the ratio $f_1()$/$f_3()$ is the fraction of the computation that can be shared and applied just once when forming the cascade network of feeds (Figure 5.11). Table 5.2 reports this fraction, which we vary, as a percentage value that is hereafter referred to as $\%_{OVERLAP}$.

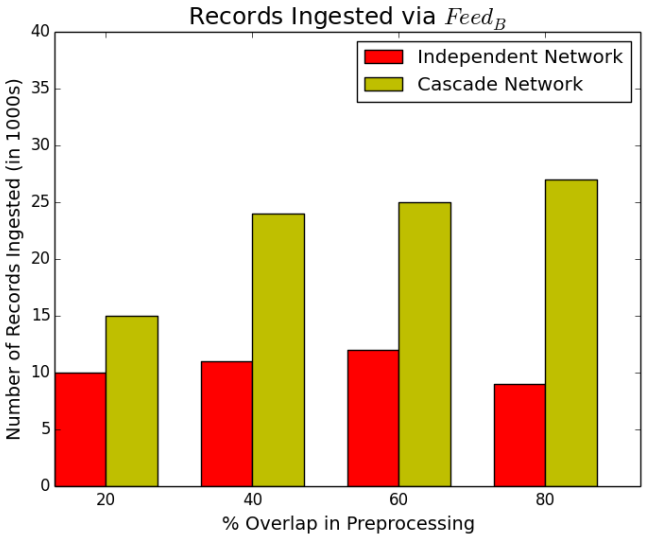
Table 5.2: Execution time for functions in milliseconds and the % computation that can be shared when constructing the feeds – $Feed_A$ and $Feed_B$ in a cascade network

$f_1()$	$f_2()$	$f_2(f_1())$ *or* $f_3()$	% $f_1()$/$f_3()$ or $\%_{OVERLAP}$
10	40	50	20
20	30	50	40
30	20	50	60
40	10	50	80

We repeated our workload sharing experiment for different values of $\%_{OVERLAP}$ (20, 40, 60 and



(a) A comparison of the total number or records persisted via $Feed_A$ in the independent and cascade network configurations



(b) A comparison of the total number or records persisted via $Feed_B$ in the independent and cascade network configurations

Figure 5.13: A comparison of the total number of records persisted (and indexed) from each feed (Feed_A and Feed_B) in a Cascade network (Figure 5.11) and an Independent network (Figure 5.12) configuration. The $\%_{OVERLAP}$ between the preprocessing required by each feed forms the x axis.

80 %). For each given value of $\%_{OVERLAP}$ between the pre-processing required by $Feed_A$ and $Feed_B$, we measured the total number of records persisted via each feed in the case of the independent network (Figure 5.11) and the cascade network (Figure 5.12) configurations. A sufficiently

high rate of generation of tweets by TweetGen (5000 twps) caused maximum utilization of CPU at each worker node across all runs of our workload experiment. Figure 5.13 summarizes the results. The following are the most noteworthy observations:

- *Records persisted by* $Feed_A$:

  As shown in Figure 5.13(a), for each value of $\%_{OVERLAP}$, the records persisted via $Feed_A$ in a cascade network configuration exceeds the corresponding value for the independent network configuration. The cascade configuration reduces the amount of computation required for producing the secondary $Feed_B$ as the computation done as part of ingestion in $Feed_A$ is not repeated. This shared computation includes the parsing and translation of received records at the intake stage and the application of the UDF $f_1()$ at the compute stage of the data ingestion pipeline for $Feed_A$. The reduced computation required in producing $Feed_B$ in a cascade network lowers the demand for resources (CPU cycles) at each worker node. This allows a greater number of records from $Feed_A$ to be processed per unit time.

- *Records persisted by* $Feed_B$:

  By definition, constructing $Feed_B$ requires subjecting each record received from TweetGen to the functions $f_1()$ and $f_2()$ in that sequence. In a cascade network configuration (Figure 5.11), the required computation is reduced to just applying the function $f_2()$ on records that are being output by the compute stage of the ingestion pipeline for $Feed_A$. As described earlier, the experimental setup involved a sufficiently high rate of arrival of data together with computationally expensive UDFs placed along the ingestion pipeline for each feed. Each worker node operated in a resource-constrained environment with CPU cycles forming the bottleneck. The cascade network configuration demands less CPU cycles because of the reduced computation. The records persisted by $Feed_B$ thus increases in such a configuration.

- *Shared Computation* ($\%_{OVERLAP}$):

  The $\%_{OVERLAP}$ parameter determines the fraction of computation that is applied as part of producing $Feed_A$ and that need not be repeated in producing $Feed_B$. As $\%_{OVERLAP}$

```
use dataverse feeds;

create feed TweetGenFeed using TweetGenAdaptor
(("datasource" = "10.1.0.1:9000,
    10.1.0.2:9000,  10.1.0.3:9000,
    10.1.0.4:9000,  10.1.0.5:9000,  10.1.0.6:9000"))
apply function addFeatures;

connect feed TweetGenFeed to dataset ProcessedTweets;
```

Listing 5.19: AQL to create a pair of primary and secondary feeds used in evaluating scalability

is increased from 20 to 80 (percentage value), the gap between the cascade network and an independent network configuration also widens, as measured in terms of the number of records persisted. This is observed for each of the feeds – $Feed_A$ and $Feed_B$. A higher value of $\%_{OVERLAP}$, by definition, signifies a greater amount of resource-savings in a cascade network configuration, which manifests as an increase in the number of records persisted.

5.7.3 Evaluating Scalability

In our final experiment in this chapter, we provide an experimental evaluation of the scalability offered by the AsterixDB feed ingestion facility. We again used TweetGen as an external data source and defined a primary feed, TweetGenFeed, that made use of the TweetGenAdaptor to receive tweets generated by TweetGen. We associated a custom UDF (Java) with TweetGenFeed that collects the hash tags contained in the tweet in an ordered list and appends it as an additional attribute to the tweet. The set of AQL statements to define and connect our experimental feed is shown in Listing 5.19, and the logical flow of data is shown in Figure 5.14. The adaptor is configured with the location(s) (socket address) where instances of TweetGen are running. Each instance of TweetGen receives a request for data from a corresponding instance of *TweetGenAdaptor*, thus enabling ingestion of data in parallel.

Referring to Figure 5.14, if the record arrival rate exceeds the rate at which they can be processed

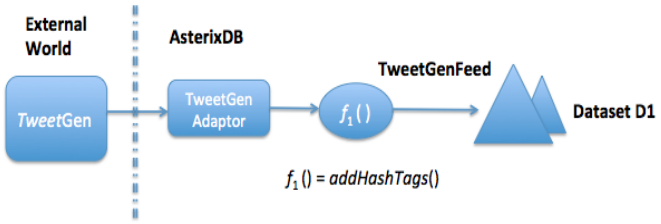


Figure 5.14: Scalability: Number of records (tweets) successfully ingested (persisted and indexed) as the cluster size is increased.

and ingested in AsterixDB, the 'excess' records are handled as dictated by the ingestion policy associated with the feed (Table 4.2 from Chapter 4). Through initial experiments, we determined the ingestion capacity offered by a single node AsterixDB cluster as 20k twps. For this data source and UDF, we used six parallel instances of TweetGen and configured each such that the aggregate rate of generation of tweets ($20k * 6, 120k$ twps) far exceeded the ingestion capacity that we measured experimentally. Initiating the flow of data from TweetGen at an aggregate rate that is higher than the ingestion capacity of the recipient AsterixDB cluster results in 'excess' records and cause maximum CPU utilization across the worker nodes. We selected the Discard ingestion policy to discard the excess records; this again makes the persisted % data volume a good performance metric.

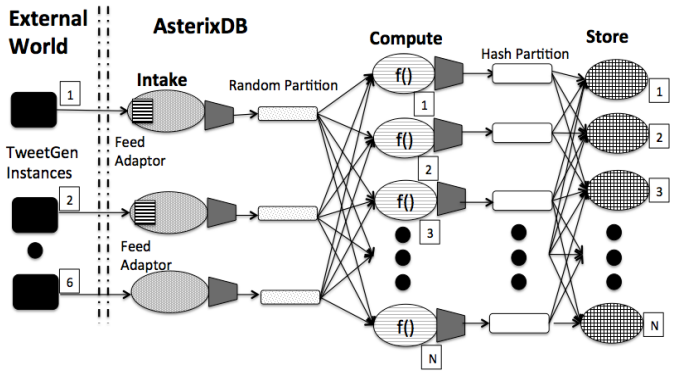


Figure 5.15: Evaluating Scalability: A physical view of the flow of data through the intake, compute, and store stages of the data ingestion pipeline. The degree of parallelism at the compute and store stages is determined by the number of nodes in the AsterixDB cluster.

We evaluated the ability of the AsterixDB feed facility to scale and ingest an increasingly large volume of data as additional resources are added. To do so, we varied the size of our AsterixDB

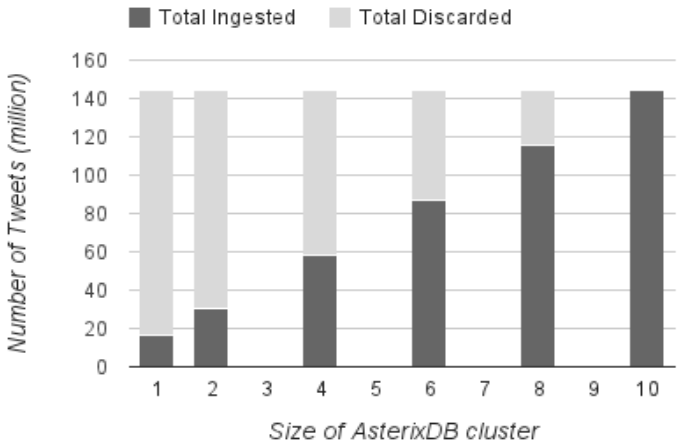


Figure 5.16: Measring Scalability: Number of records (tweets) successfully ingested (persisted and indexed) as the cluster size is increased.

cluster (1 to 10 nodes) and measured the total number of records that were successfully persisted in the 20 minute time window during which the TweetGen instances continued to push data into AsterixDB. Figure 5.15 shows the physical data flow and the parallelism involved at the compute and store stages for the data ingestion pipeline. As we add more resources (AsterixDB nodes) to the cluster, the degree of parallelism at the compute and store stage increases accordingly, as each node hosts a pair of compute and store operator instances. Note that the parallelism at the intake stage remains fixed at 6, which is the number of TweetGen instances.

The experimental results are shown in Figure 5.14. A significant proportion of records were discarded for lack of resources on a small size cluster of 1 to 4 nodes. On a bigger cluster, the proportion of discarded tweets declines, indicating that the system that can indeed ingest an increasingly high volume of data as additional resources (nodes and cores) are added to the system. The results in Figure 5.16 indicate **linear scaleup** characteristics. Linear scale-up is one of the most critical features required of a data ingestion facility as it allows the system to handle an increased workload simply by addition of resources.

5.8 Summary

In this chapter, we have described the basic internals of the data ingestion support in AsterixDB with an emphasis on the physical aspects and underlying design considerations. We described the methodology adopted for constructing the runtime for a data ingestion pipeline and covered various example scenarios to illustrate the different steps involved in constructing the head and the tail sections of the pipeline separately. We also described how a cascade network is constructed to involve two or more data ingestion pipelines and how it is dynamically modified when active feeds are disconnected or additional feeds are connected.

The chapter evaluated the basic promise of this thesis – that data ingestion can and should be an effective new feature in BDMS. We compared feeds in AsterixDB with the conventional approach of inserting batches of records, a major improvement that feeds provide. We demonstrated the benefits of having and using the *Fetch Once, Compute Many* model, and the scalability offered by the data ingestion facility in AsterixDB.

A wide variety of sources of data emit records in a continuous fashion; examples include sensors, electrical meters, social media, clickstreams, processes that generate logs, etc. In order to ingest data from such sources, it is required to have a built-in mechanism for continuous insert of records, which at minimum, performs better than conventional batch inserts and offers scalability. In subsequent chapters, we will discuss other desirable features of such a system that include fault-tolerance and elasticity.

Chapter 6

Fault-Tolerant Data Ingestion

Data ingestion is a long running task, so it is bound to encounter hardware failure(s) as it continues to run "forever" on a cluster of commodity hardware. Furthermore, parts of a data ingestion pipeline include pluggable user-provided modules (feed adaptor and a pre-processing function) that may cause soft failures in the form of runtime exceptions. Sources of such runtime exceptions include unexpected data format, unexpected null values for an attribute, or simply inherent bugs in the user-provided source code that show up for certain kind(s) of data values. We categorize the failures occurring from processing of data as soft failures, and those arising from loss of a physical machine either due to a disk, network or power failure as hard failures. In this chapter, we describe how a data ingestion pipeline may recover from soft and hard failures and particularly address the challenge C4 from Section 1.1 of Chapter 1. Note that the kind of failures that a data ingestion pipeline must attempt to handle and survive is dictated by its associated ingestion policy.

6.1 Soft Failures

A runtime exception encountered by an operator in processing an input record in a typical Hyracks job carries non-resumable semantics and causes the dataflow to cease and the job to terminate. It is essential to guard the data ingestion pipeline from such exceptions by executing each participant operator in a sandbox-like environment, such that the data ingestion pipeline is insulated from any runtime exception(s) that a participant operator instance may throw. It is equally important that any mechanism to handle and recover from failures need not be embedded into the definition of the operator itself. This adheres to the popular *Separation of Concerns (SoC)* [41] design principle and ensures that operators remain simple to build and reusable as part of other Hyracks jobs. Next, we introduce the *MetaFeed* operator and describe its role in handling soft failures.

6.1.1 Executing An Operator in a Sandbox

Each operator in a data ingestion pipeline is an implementation of an interface that provides to the Hyracks runtime, a set of APIs to initialize the operator, pass data frames for processing, and manage the operator's lifecycle. The MetaFeed operator was introduced as a wrapper operator that mimics it's enclosed operator (hereafter referred to as the *core* operator) in implementing an identical interface. The Hyracks runtime remains agnostic of such wrapping provided by the MetaFeed operator and continues to invoke the (identical) interface methods (API), as before. The MetaFeed operator delegates invocation of all methods to the *core* operator but provides additional functionality to enable fault-tolerance.

The runtime of a core operator receives input data as a sequence of frames each comprising of records. An exception thrown by the core operator in processing an input record is caught by the wrapping MetaFeed operator. The MetaFeed operator slices the original input frame to form a subset frame that excludes the processed records and the exception generating record. The remnant

subset frame is then passed to the core-operator as the next frame to be processed. The operator has, in effect skipped past the exception-generating record. If a record in the truncated frame causes an exception, the act of slicing and weeding out the error generating tuple is repeated as many times as required.

6.1.2 Logging of an Exception

A runtime exception may be indicative of the inability of the core operator in handling a given kind of record, e.g., one that has null values for an attribute, or may even reflect a bug in the source code or invalidate an assumption made about the external data source. It is important to allow the end-user to revisit the exception(s) thrown during data ingestion for diagnosis and take subsequent corrective action, if any, or simply report data that could not be persisted and indexed.

The MetaFeed operator provides different options for logging a runtime exception. At minimum, the exception and the causing record are appended to the standard AsterixDB error log file. Alternatively, the information may also be persisted into a dedicated AsterixDB dataset. The logging support for a feed is determined from the ingestion policy[1] that is specified as part of the *connect feed* AQL statement.

In a possible scenario, every record may result in a similar exception; this situation would be indicative of a bug or an invalid assumption made about the external data source. A cycle of handling and logging an exception in such a case would be never-ending and wasteful of resources. To avoid such a situation, a feed ingestion policy can be configured with an upper bound on the number of consecutive records that can be "skipped" by an operator. Upon reaching the limit, an exception raised on the next incurring record causes the faulty feed to end.

[1]Persisting the details about a soft exception in a dedicate ddataset is enabled by setting the parameter "soft.failure.log.data" to "true" in the ingestion policy.

6.2 Hard Failures

In this section, we describe the mechanism by which AsterixDB handles the loss of one or more of the AsterixDB nodes involved in a data ingestion pipeline. The operators participating in a data ingestion pipeline operate in a pipelined fashion such that an upstream operator 'pushes' data frames downstream to a consuming operator. For a long running task such as data ingestion that runs in a distributed environment built from commodity hardware, the failure of a node can be considered as a norm rather than an exception. Loss of a participating node in a typical Hyracks job setting carries non-resumable semantics. Such semantics are unacceptable for a data ingestion pipeline. Any interruption in the flow of data from an external source into AsterixDB can potentially introduce a lag wherein external source continues to generate records but these cannot flow into AsterixDB. In the worst case, such a period of discontinuity can potentially result in a loss of data, as the ability to request old data may not be supported by the external data source. AsterixDB as a receiver of data may never catch pace with an external data source that continues to generate and send records agnostic and irrespective of the failures happening inside an AsterixDB cluster.

6.2.1 Detecting and Identifying a Failure

As as a distributed execution engine, Hyracks allows for subscribing to events and being notified of their occurrence. These events are broadly classified as *job-events* that are related with the lifecycle of a job (examples include creation and termination of a job) and *cluster-events* that are related to cluster membership (examples include joining and leaving of nodes). Recall that an AsterixDB clsuter operates in a master-slave configuration, with a central Cluster Controller (*master*) or CC and a set of *worker* nodes referred to as Node Controllers or NCs. Co-located with the Cluster Controller is a Central Feed Manager (referred to as CFM, hereafter) that oversees the execution of each active data inegstion pipeline in the cluster. The CFM also subscribes to job-events and cluster-events. In addition, an AsterixDB cluster also includes a set of Feed Managers (FMs), one

per each Node Controller. The FMs can communicate with the CFM but do not communicate with each other. The communication with the CFM is via *control* messages that travel separate from any data path and are exchanged as bytes sent at designated (but configurable) socket address where the CFM listens.

Hyracks requires each Node Controller (NC) to heartbeat its 'live' status to the Cluster Controller (CC) with a configurable periodicity. A failure in receiving a heartbeat for a configurable threshold duration is assumed by the CC as a node failure. A cluster-event with information of the failed set of node(s) is dispatched to the set of interested subscribers including the Central Feed Manager (CFM). The CFM keeps track of the location for each operator instance that is participating in a data ingestion pipeline. On being notified of a node failure, the CFM identifies the set of data ingestion pipelines that are affected by the loss of the node. The affected data ingestion pipelines (if any) include those that had a participating operator instance running on the a failed node. Subsequent to detecting a failed node, a fault-tolerance protocol is triggered. At a high-level, the protocol works by substituting the failed node with another live node and re-scheduling the ingestion pipeline to involve the substitute node. When restructuring an ingestion pipeline, care is taken to prevent or minimize the loss of data. Next, we describe the fault-tolerance protocol in detail and emphasize the physical aspects and intricacies involved. We include a set of example scenarios and describe their handling using the protocol.

6.2.2 The Fault-Tolerance Protocol

When an AsterixDB node fails, its operator instances are lost and are referred to as *dead* instances. Subsequently, the other operator instances in the same pipeline that are running on other (alive) AsterixDB nodes are notified of the pipeline failure. The fault-tolerance protocol defines the set of actions that are required to be taken by each kind of operator instance (*collect, intake, assign, store*) upon receiving such notification. Before describing the set of actions, we define the possible

states that an operator instance may transit into as part of the fault-tolerance protocol.

1. **Zombie Instance**

   An operator instance may be allowed to terminate, but prior to termination, its runtime state (which consists of the input buffer) is saved with the local Feed Manager by the wrapping MetaFeed operator. The operator instance is referred to as a *Zombie*, as it is not alive as a thread processing data frames, but its state is available for retrieval when the pipeline is resurrected and dataflow is resumed.

2. **Alive Instance**

   An operator instance may be allowed to continue to process arriving data frames irrespective of the failures that have occurred elsewhere in the ingestion pipeline. Such an instance is referred to as an *alive* instance, and it may operate in either of two modes, namely – *forward* and *buffer*. In *forward* mode, the alive instance sends or forwards its output data frames to the downstream consuming operator. In the buffer mode, the output data frames from an operator are instead held in memory and not yet sent downstream to the next operator instance in the pipeline.

Next, we describe the precise behavior for each kind of operator when a data ingestion pipeline is handling a failure.

1. **Collect**: A *collect* operator instance continues to stay *alive* and operates in the 'forward' mode wherein output data frames are pushed downstream.

2. **Intake**: An *intake* operator instance continues to stay *alive* and receives data from the corresponding *collect* operator instance. However it transits to operate in the *'buffer'* mode, meaning that records received are held in memory and not sent downstream.

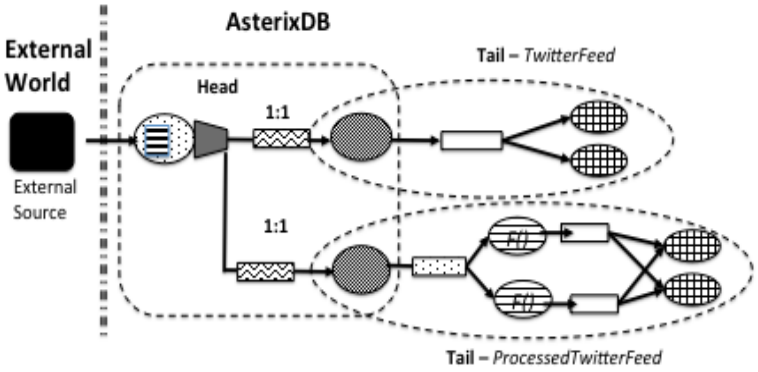
3. **Assign (or Compute)**: An *assign* operator instance transits to a *zombie* instance.

4. **Store**: A *store* operator instance behaves similar to an *assign* operator instance and transits to a *zombie* instance.

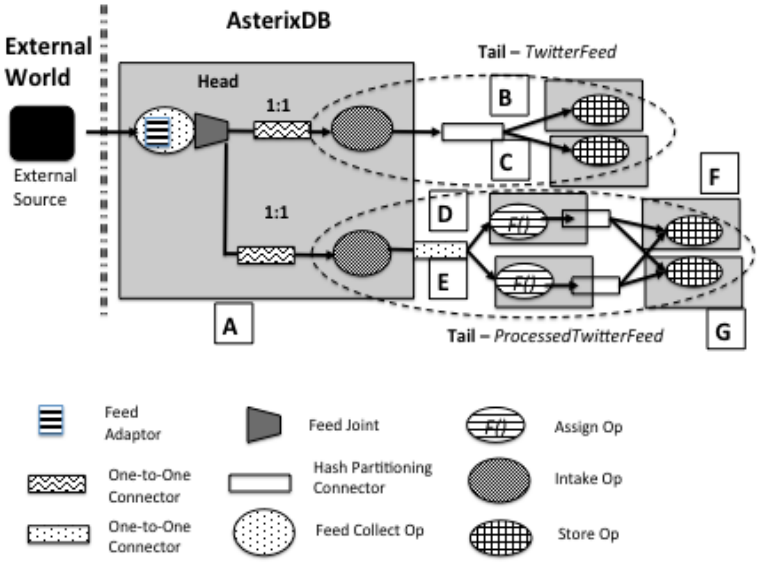
When each operator instance has transitioned into the prescribed state, the local Feed Manager on each participant node notifies the Central Feed Manager (CFM) running at the master node. The CFM chooses a node to substitute each failed node. Being a substitute implies that each operator instance at a failed node will need to be rescheduled to run at the substitute node. The CFM creates a revised structure of the ingestion pipeline wherein the failed node is replaced with its substitute. The revised pipeline is subsequently scheduled to run on the cluster. In order to describe the mechanism for restructuring and resumption of flow of data subsequent to a failure, we consider an example cascade network that involves a pair of ingestion pipelines. We discuss different failure scenarios and illustrate how the above described protocol works to overcome node failures.

6.2.3 Example Failure Scenarios

Figure 6.1(a) shows a cascade network involving a pair of data ingestion pipelines — the primary *TwitterFeed* — and its descendant secondary *ProcessedTwitterFeed*. The pipelines share a common head section that retrieves raw tweets from the external source (Twitter in the current context). We assume a 10 node AsterixDB cluster that includes a master node (where Cluster Controller and Central Feed Manager run) and a set of nine worker nodes – nodes [A-I]. An example physical layout of the cascade network showing the operator locations on our example cluster is shown in Figure 6.1(b). Note that nodes H and I are not used initially and not shown in Figure 6.1(b). Next, we describe different failure scenarios.



(a) A logical view of the flow of data across an example cascade network



(b) Physical view of the flow of data showing the placement of operator instances across an AsterixDB clusters

Figure 6.1: An example cascade network to describe the fault-tolerance protocol

External Source Failure

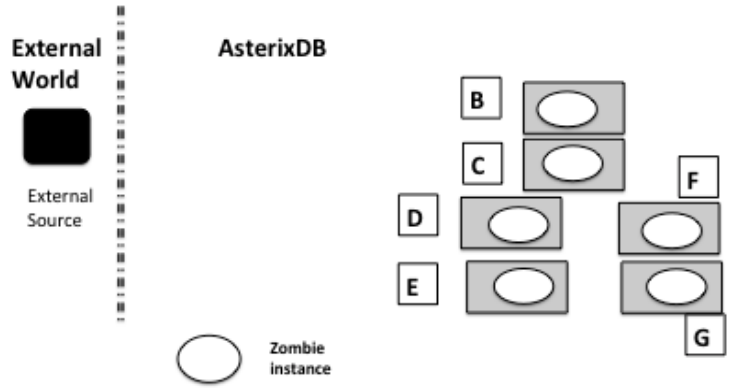
The external data source runs on just another physical machine outside the AsterixDB cluster. A power or disk failure at the external machine or lost network connectivity would interrupt the regular flow of data into an ingestion pipeline. For example, Twitter or CNN as a data source may itself experience an outage. AsterixDB doesn't understand the semantics of the data transfer protocol and remains agnostic of the set of APIs offered by the external source. The recovery logic subsequent to a failure at an external source is required to be provided by the feed adaptor.

An adaptor may resort to reconnecting after a wait or connecting to a different server/machine offered as part of agreed communication protocol. However, if the adaptor discovers reconnecting as a futile or infeasible exercise, it must convey this to AsterixDB by throwing an exception, in which case AsterixDB terminates the feed and relinquishes any involved resources. On the contrary, if the adaptor is able to recover from the failure (via reconnecting, switching to a different source node etc.), the flow of data resumes. It is worth noting that in this case, AsterixDB remains agnostic of the failure and the recovery action taken by the feed adaptor. The individual operator instances may simply not receive additional data records to process and this may manifest as a transient drop in the data ingestion throughput.

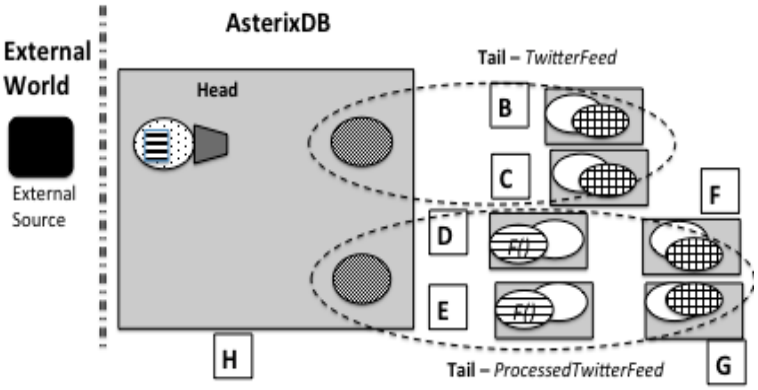
Collect or Intake Operator Failure

Referring to our example data flow shown in Figure 6.1(b), we assume the loss of node A while the feeds — TwitterFeed and ProcessedTwitterFeed — are active. Node A houses the *intake* operator instance from the tail section of each of the two ingestion pipelines. Additionally the co-located *collect* operator instance from the shared head section of each pipeline is also lost. This interrupts the flow of data into AsterixDB as the connection between the source (as the producer) and the *collect* operator instance (as the receiver) is broken. The remaining operator instances (*assign* and *store*) on each ingestion pipeline, which are running elsewhere on healthy nodes, are notified. Subsequently these transition into *zombie* instances, as dictated by the fault-tolerance protocol. This state is shown in Figure 6.2(a).

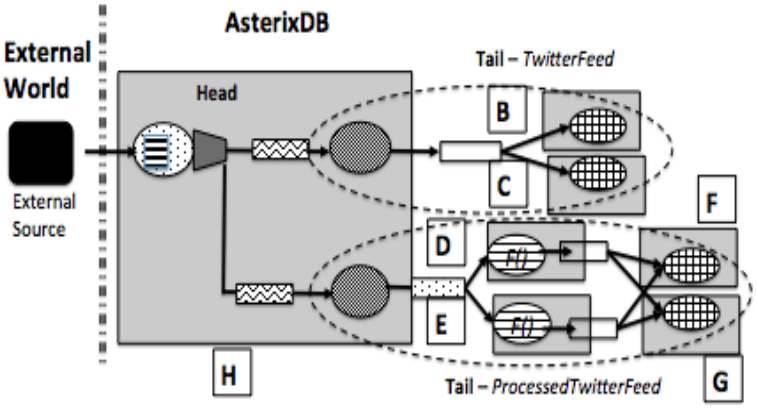
Next, the Central Feed Manager must choose a node as a substitute for the failed node (A). To illustrate, we show node H as being the chosen substitute for node A, but any of the nodes (B-I) could be chosen. The *FeedCollect* Hyracks job is rescheduled with a *FeedCollect* operator instance placed at the substitute node (H). This resurrects the head section. The respective *FeedIntake* jobs that represent the tail sections for *TwitterFeed* and *ProcessedTwitter* are then revived. In doing so, the location constraints for each operator instance are carefully chosen to be identical to the



(a) State of the data ingestion pipeline subsequent to loss of intake node. The zombie instances on alive node remain to register their state with the local Feed Manager



(b) Recovery Phase: A revised ingestion pipeline is scheduled with location for each operator instance chosen carefully. The head section is yet to connect with the external source and resume the flow of data



(c) Post Recover: Revised data ingestion pipeline showing resumed dataflow

Figure 6.2: Handling of the loss of an intake node in a cascade network

previous execution. Thus, the *assign* and *store* instances are co-located with the respective *zombie* instances from the previous execution. Such a placement allows a newly created live instance of an operator to retrieve the state (input buffer containing unprocessed data) corresponding to its zombie instance from the local Feed Manager. These live instances shall process the unprocessed data records from the previous execution in an effort to minimize the loss of data from pipeline failure.

Figure 6.2(b) shows a transient state showing opreator instances as being co-located with their respective zombie counterparts. The tail section for each reconstructed pipeline is able to subscribe to the output of the shared head section. This happens at the substitute node H. At this stage, the *collect* operator instances at the head make use of the feed adaptor to establish connection with the external source and resume the flow of data. The recovered state is shown in Figure 6.2(c). It is similar to the state shown in Figure 6.1(b) except that Node A has been replaced with Node H. Note that the loss of *FeedCollect* operator instance results in a period when AsterixDB is unable to receive data from the external source.

Assign Operator Failure

Next, we assume the failure of node D that is part of the compute stage for *ProcessedTwitterFeed*. Following the fault-tolerance protocol, other operator instances belonging to the data ingestion pipeline transition to the prescribed states. This intermediate state is shown in Figure 6.3(a). The *intake* operator instance at the head section continues to stay alive but transitions to the *buffer* mode, that is, it holds the arriving data in memory instead of sending downstream. Loss of node D does not impact the flow of data along the data ingestion pipeline for *TwitterFeed*, although the pipelines share a head section. This behavior is an example of the *fault isolation* principle that keeps other ingestion pipelines that are part of a common cascade network insulated and functional despite failures elsewhere in the cascade network.

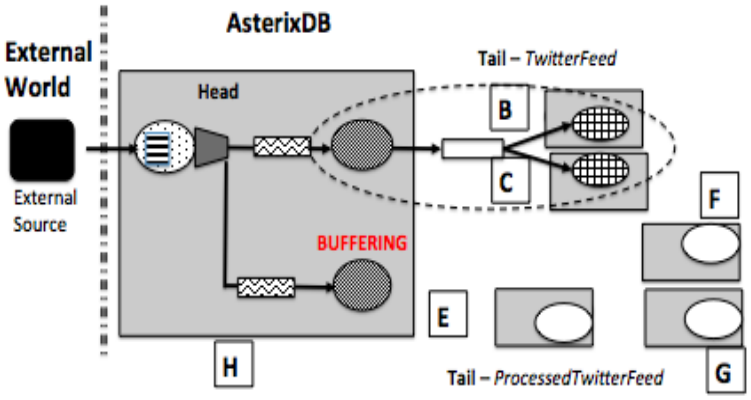
Once the state shown in Figure 6.3(a) is achieved, the Central Feed Manager is notified and a substitute for the failed node is chosen. For illustration, we chose node I as the substitute, but any of the other nodes could be chosen. Figure 6.3(b) shows a transient state with Node I as the substitute and alive operator instances from the revised pipeline being co-located with their respective zombie instances from previous pipeline. The *intake* operator instance takes ownership of the input buffer used by the alive instance from previous execution and continues to send the arriving data downstream. The previous *intake* operator instance is then allowed to terminate. The revised cascade network is shown in Figure 6.3(c). It is similar to 6.1(b) except that node D has been replaced with Node I.

Store Operator Failure

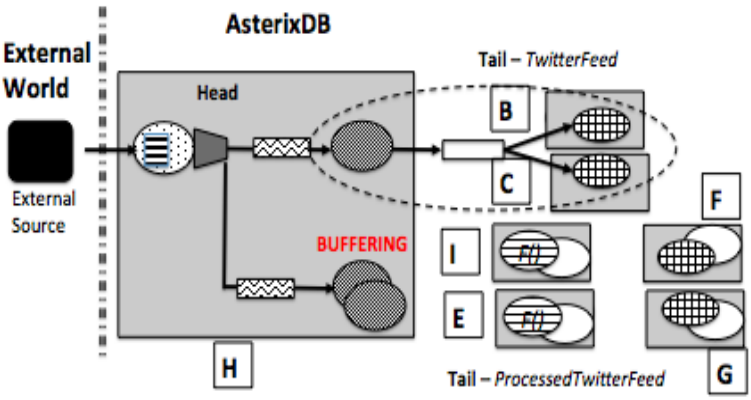
The loss of a store node translates to the loss of a partition of the dataset that is receiving the feed. AsterixDB does not yet support data replication. In absence of the replica(s), there does not exist a substitute for the lost dataset partition. In the current implementation, a store node failure therefore results in an *early* termination of an associated feed. As and when the failed store node re-joins the cluster and becomes available[2], the data ingestion pipeline is rescheduled. New operator instances in the rescheduled pipeline take ownership of the state left behind by their respective zombie instances from the previously failed execution. Data replication is on the road map of AsterixDB. An AsterixDB node hosting an in-sync replica of the lost data partition would become the preferred choice for being an immediate substitute; the recovery phase would then involve rescheduling the pipeline to involve the replica.

With respect to data preservation midst hardware failures, AsterixDB does not guarantee lossless ingestion of data. Although, operator instances save the frames from their input buffers with the local feed manager following a pipeline failure, termination of the pipeline results in the loss of

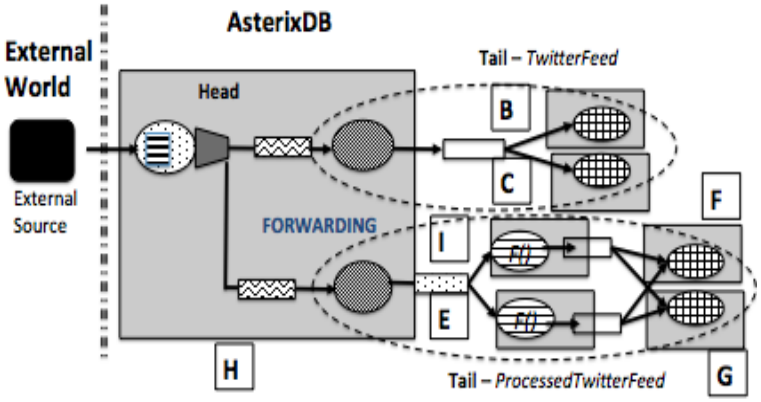
[2]In AsterixDB, a failed node upon re-joining the cluster undergoes log-based recovery to ensure that all hosted dataset partitions are in a consistent state.



(a) State of a data ingestion pipeline subsequent to the loss of a compute node. The zombie instances remain to register their state with the local Feed Manager. The shared head section continues to receive data.



(b) Recovery phase: The impacted data ingestion pipeline is revised to involve a substitute for the lost node.



(c) Post Recovery: The impacted data ingestion pipeline resumes receiving data from the head section.

Figure 6.3: Handling the loss of a compute node in a cascade network

the in-flight records that failed to reach their destination. It would be possible to preserve the in-flight records by use of checkpointing to coordinate the flow of data between operators [36]. Such support could then be conditionally incorporated in a data ingestion pipeline if the associated ingestion policy requested lossless movement of data.

6.2.4 Under the Hood

The operators participating in a data ingestion pipeline are reusable components that can be employed as part of other Hyracks jobs. It is essential to keep these operators simple and generic so that data concerns are kept separate from fault-tolerance concerns. The MetaFeed operator that was introduced in Section 5.7.2 provides much of the functionality that makes an ingestion pipeline tolerant of hardware failures. As a MetaFeed operator instance wraps around a *core* operator instance, it can intercept all invocations (made by Hyracks runtime) of the core operator's runtime API. Subsequent to a failure in a normal Hyracks job, the Hyracks runtime notifies the job'e other alive operator instances. This is done via the invocation of the *close()* method that is implemented by each operator, which instructs the operator to terminate. In a data feed job, the invocation of the *close()* method on an operator interface is intercepted by the wrapping MetaFeed operator. The MetaFeed operator instance implements the fault-tolerance protocol and transitions the core operator into the required state, as prescribed by the protocol. The functionality of retrieving the state from the previous zombie instance and handling of the unprocessed records is also provided by the MetaFeed operator instances.

6.3 Experimental Evaluation

We have evaluated the ability of the AsterixDB feed subsystem to recover from single and multiple hardware failures while continuing to ingest data. Earlier, in Section 5.7.2 from Chapter 5, we

```
use dataverse feeds;

connect feed ProcessedTweetGenFeed to
dataset ProcessedTweets using policy FaultTolerant ;

connect feed TweetGenFeed to
dataset RawTweets using policy FaultTolerant ;
```

Listing 6.1: AQL statemens that connect each feed to its respective dataset

described the use of a custom-built data generator – *TweetGen* – to act as an external data source that could generate synthetic tweets at a prescribed rate. We reuse TweetGen here as an external source in setting up a cascade network that involves a pair of feeds – *TweetGenFeed* (primary) and *ProcessedTweetGenFeed* (secondary). *ProcessedTweetGenFeed* involves the use of a Java UDF that collects the hash-tags in a tweet's text in to an ordered list and appends it to the tweet as an extra field.

This experiment involved having a pair of TweetGen instances configured to generate data at 5000 tweets/sec (twps). Each instance ran on a separate machine outside the AsterixDB cluster. We connected the feeds –*TweetGenFeed* and *ProcessedTweetGenFeed* – to their respective target dataset and used the built-in policy *Fault-Tolerant* when doing so. Figure 6.1 shows the set of AQL statements. The nodegroup associated with each dataset included a pair of nodes each. To make things interesting and show that the order of connecting related feeds is not important, we connected *ProcessedTweetGenFeed* prior to connecting its parent feed *TweetGenFeed*. In this scenario, as the primary feed (*TweetGenFeed*) is not active at first, building the ingestion pipeline for *ProcessedTweetGenFeed* requires constructing the head section that would interface with the external source (pair of *TweetGen* instances). Data begins to flow with the construction of the tail section. The ingestion pipeline offers feed joints (labelled as 'A' and 'B' in Figure 6.4). Subsequently, in building the data ingestion pipeline for the primary feed, the head section from *ProcessedTweetGenFeed* is reused and the output data flowing through feed joints 'A' is routed to the tail section for *TweetGenFeed* (Figure 6.4).

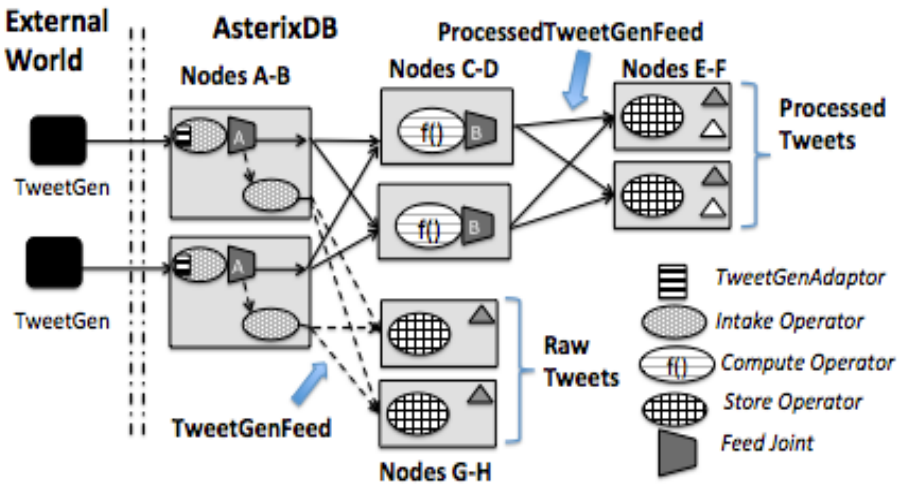


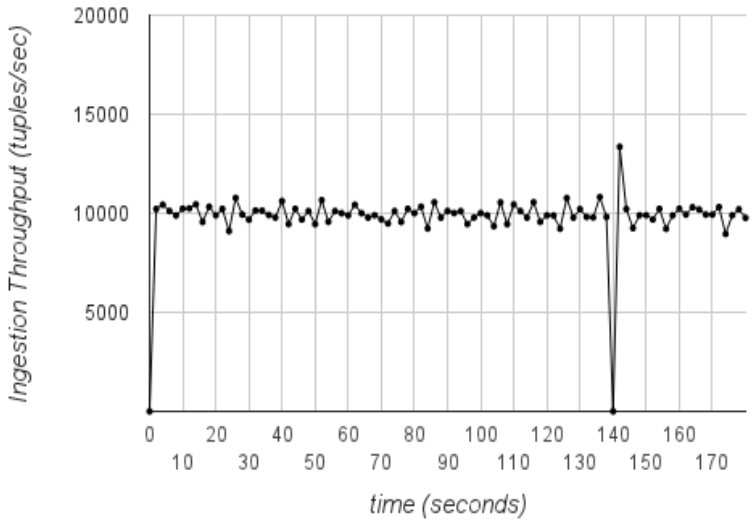
Figure 6.4: Feed cascade network for fault-tolerance experiment

We measured the number of records inserted into each target dataset during consecutive two-second intervals to obtain the instantaneous ingestion throughput for the associated feed. We introduced a compute node failure (node C in Figure 6.4) at time t=70 seconds. This was followed by a concurrent failure of both an intake node (node A) and a compute node (node D) at time t=140 seconds. The instantaneous ingestion throughput for each feed as plotted on a timeline is shown in Figure 6.5.

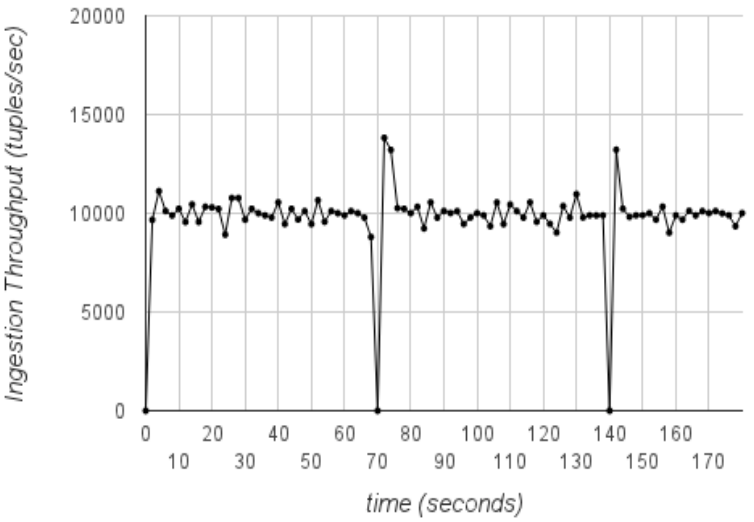
The following are the most noteworthy observations:

(i) **Recovery Time**: The failures are reflected as a drop in the instantaneous ingestion throughput at the respective times. Each failure was followed by a recovery phase that reconstructed the ingestion pipeline and resumed the flow of data into the target dataset within 2-4 seconds.

(i) **Fault Isolation**: Data continues to arrive from the external source at the regular rate, irrespective of any failures in an AsterixDB cluster. During the recovery phase for *ProcessedTweetGenFeed*, the FeedIntake operator instances buffer the records until the pipeline is resurrected. The FeedIntake operator instances corresponding to *TweetGenFeed* continue to receive and send the data records downstream at the regular rate. Fault isolation helps in "**localizing**" the impact of a pipeline failure and is a desirable feature of the system. The ingestion pipeline for *TweetGen* feed remains insulated from the failure of Node C. As shown in Figure 6.5(a), *TweetGenFeed* is not



(a) TweetGenFeed



(b) ProcessedTweetGenFeed

Figure 6.5: Instantaneous ingestion throughput with interim hardware failures: Node C fails at t=70 seconds; Node A and Node D fail at t=140 seconds

impacted by the failure of node C at t = 70 seconds.

6.4 Other Approaches to Fault-Tolerance

Stream processing work in the past also faced challenges related to *availability* and sustaining the flow of data, irrespective of node failures. In this section, we look at some of the known methods that provide fault-tolerance in Stream Processing Engines (referred to as SPEs hereafter). We discuss their different approaches and provide comparisons with the approach adopted in AsterixDB.

6.4.1 Replication-Based Approach

The replication-based method, also known as *Process Pairs* approach [27], involves replicating each participating operator in a given dataflow to form cloned versions that are scheduled to run in parallel. An operator instance is chosen as being *primary*, whereas its replica instances are referred to as *secondary*. The Process Pairs approach is further classified as involving a '*passive standby*' or an '*active standby*'.

- **Passive Standby**:

  In the 'passive standby' approach, a primary operator instance periodically checkpoints its state and sends that checkpoint to secondary replica instance(s). The state includes any data maintained by the operator instance and any records stored in queues between operator instances. In practice, sending the entire state at every checkpoint is not necessary. Instead, each primary operator instance periodically performs only a delta-checkpoint. During a delta-checkpoint, the primary operator instance updates the backup by copying only the difference between its current state and the state at the time of the previous checkpoint. Because of these periodic checkpoints, a backup (secondary operator instance) always has its primarys state as of the last checkpoint.

  To detect failures, each replica operator instance sends heartbeat requests to its primary and assumes that the primary has failed if the number of consecutive attempts that do not return

within a timeout period crosses a pre-configured threshold. If the primary fails, the secondary recovers by restarting from its saved copy of the primary's state and reprocessing the input records that the primary processed since the last checkpoint. To enable backups to reprocess such input tuples, all primary operator instances log their output records. If a downstream primary fails, each upstream primary instance re-sends its output records to the downstream backup.

- **Active Standby**:

  In the case of active standby, the replica operator instances actually process all records in parallel. An upstream operator instance logs its output and forwards the output to its current set of downstream operator instances. Each recipient operator instance sends periodic acknowledgments to its upstream operator instance to indicate that it has received the input stream up-to a certain point. An acknowledgment indicates that the input need not be resent in the case of a failure; this allows producers to truncate their output logs.

6.4.2 Upstream Backup Approach

In the upstream-backup approach, the upstream nodes act as backups for the downstream nodes by logging tuples in their output queues until all downstream nodes process their tuples completely. The upstream log is trimmed periodically using acknowledgments sent by downstream primaries. In case of failure, the upstream primaries replay their logs, and the secondary nodes rebuild the missing state before serving other downstream nodes. Compared to the Passive Standby approach, upstream backup does not involve creating replicas of an operator instance, neither does it require operator instances to take periodic checkpoints. In comparison to the Passive Standby approach, this strategy requires longer recovery, but has lower runtime overheads.

6.4.3 Flux

The Flux system [36] presented a technique for correctly coordinating replicas of individual operator partitions within a larger parallel dataflow. The proposed method address the challenges of avoiding long stalls and maintaining exactly-once, in-order delivery of the input to these replicas during failure and recovery.

The goal of Flux was to make the in-flight data and transient operator state fault-tolerant and highly available. In-flight data consists of all records in the system from acknowledged input from the source to unacknowledged output to the client. This in-flight data includes intermediate output generated from operators within the dataflow that may be in local buffers or within the network itself. Inspired by the process-pairs technique [27], this approach provides fault-tolerance and high availability by properly coordinating redundant copies of the dataflow computation. Redundant computation allows quick fail-over and thus gives high availability.

However, consistency is difficult to maintain during failure and after recovery, as connections can lose in-flight data and operators may not be perfectly synchronized. Thus, the Flux techniques sought to maintain the following two invariants to achieve this goal.

1. **Loss-Free**: No tuples in the input stream sequence are lost.
2. **Duplicate-Free**: No tuples in the input stream sequence are duplicated.

To maintain these invariants, the Flux method introduced intermediate operators that connect existing operators in a replicated dataflow. Between every producer-consumer operator pair that communicate via a network connection, intermediate operators were introduced to coordinate copies of the producer and consumer. Abstractly, the protocol is as follows: To keep track of in-flight tuples, each tuple was assigned a sequence number. The intermediate operator on the consumer side received input from its producer, and acknowledged the receipt of input (with the sequence number) to the producers copy. The intermediate operator at the producer's copy stored in-flight tuples in an

internal buffer and ensured the invariants in case the original producer failed. Acknowledgements tracked the consumers progress and were used to drain the buffer and filter duplicates.

6.4.4 Borealis Stream Processing Engine

The Borealis Stream Processing Engine used the Process-Pairs approach in providing fault-tolerance. In a Process-Pairs approach, it is assumed that each operator is deterministic in nature in the sense that a given set of input records shall always produce identical output irrespective of the time of arrival of the input records. The Process-Pairs approach requires that each of the $N-1$ replicas for an operator receives and processes the arriving records in an identical order. To compute an order without the overhead of inter-replica communication, the Borealis system introduced a data serializing operator – *SUnion* – that could receive multiple input streams, order tuples on an embdedded time_stamp value and place each in statically-sized buckets. An operator is required to emit a boundary_tuple. Arrival of a boundary_tuple at a receiving operator signifies that all tuples with a timestamp less than that of the boundary_tuple have been received. All participating nodes and the data source(s) in the distributed execution of the dataflow are required to be synchronized with respect to time. Furthermore, the data sources are required to implement the fault-tolerance protocol by embedding time_stamp values in output tuples and inserting boundary_tuples in the output stream. The tuples arriving at an operator are buffered. It is assumed that an input buffer is large enough to hold all of the records that arrive during the window when a failure is being repaired.

The Process-Pairs approach is expensive in terms of the requirement for resources; each operator now has $N-1$ replicas that need to be scheduled, thus increasing the demand for resources. The requirement of the datasource implementing the prescribed fault-tolerance protocol (use of time_stamp and boundary_tuples) cannot be assumed. It is actually unlikely that a source like Twitter would opt to provide a customized stream. The approach thus cannot be termed as being

generic and extensible and capable of catering to a wide variety of data sources.

The Process Pairs approach, as implemented in Borealis, does offer the end-user a trade-off between consistency and availability. The Borealis approach assigns to each node a maximum delay that it may introduce to ensure a bounded overall delay in processing a record received from the data source. It is unclear how the delay, X, is determined per node. A non-blocking operator is allowed to continue processing of records even if a subset of its receiving only a subset of its input streams subsequent to a node failure. The records produced are termed as *tentative* in Borealis' terminology. An uninterrupted flow of records through the dataflow offers higher availability at the cost of consistency. As and when the failure is repaired and the operator begins to receive its missing set of input streams, it must send output records that are reconciled at the receiving operator to amend the state produced from earlier processing of tentative records.

The Process-Pairs approach is in contrast to the approach followed in AsterixDB. AsterixDB does not have a notion of *tentative* records and does not actually aim at maintaining an uninterrupted flow of data while a failure is being repaired. An interrupted flow of data is acceptable if the mean time to repair a failure is small enough to not violate any QoS guarantees required by the overlying application. Instead of using an increased amount of resources in running replicated versions and using of additional operators to enforce ordered processing of records, AsterixDB's approach is generic as it does not require the data source(s) to adhere to a specific protocol or to use special tuples (punctuation marks) in their respective output streams.

6.5 Summary

Large-scale computing today typically involves commodity hardware that is prone to failures. In an extensible system that involves use of pluggable components written by end-users, the presence of software bugs cannot be ignored either. Interfacing with a variety of external sources and any

assumptions made regarding the format or presence of certain attributes in the received data contribute towards soft runtime exceptions. It is thus highly desirable to build a fault-tolerant data ingestion facility. In this chapter, we have described the fault-tolerance protocol implemented in AsterixDB to safeguard a data ingestion pipeline from both soft and hard failures.

We described the feed fault-tolerance protocol in view of different example scenarios involving single and multiple node failures. An experimental evaluation emphasized two important properties of the protocol - "fault-isolation" and "mean time to recovery". We also provided a qualitative comparison of our approach to fault-tolerance with alternate mechanisms developed for distributed systems and adapted for stream processing engines.

Chapter 7

Dealing with Data Indigestion

Data ingestion is a potentially long running task that is likely to experience a dynamic environment with varying demand and availability of resources. Each data feed has an associated uncertainty in terms of the rate of arrival of records and the computational complexity of pre-processing involved, particularly in the case when the pre-processing is done by means of a Java UDF.[1] Furthermore, the number of connected feeds and queries that may execute concurrently on shared hardware cannot be statically fixed. In a resource-constrained environment, a data management system such as AsterixDB may not be able to process data at its rate of arrival. This results in *congestion*, or *data indigestion*, wherein the movement of data along an ingestion pipeline is slowed or stalled. Table 4.2 from Chapter 4 described the user-level choices for dealing with data indigestion. In this chapter, we study the effects of congestion and evaluate the alternate approach as offered in AsterixDB to resolve resource bottlenecks and facilitate continuous movement of data.

[1] A Java UDF is considered as a black box by AsterixDB, and as such its computational complexity and requirements for resources remains hidden from AsterixDB. This is in contrast to an AQL UDF whose definition is visible to the AsterixDB compiler and can potentially be optimized by use of indexes.

7.1 Congestion or Data Indigestion

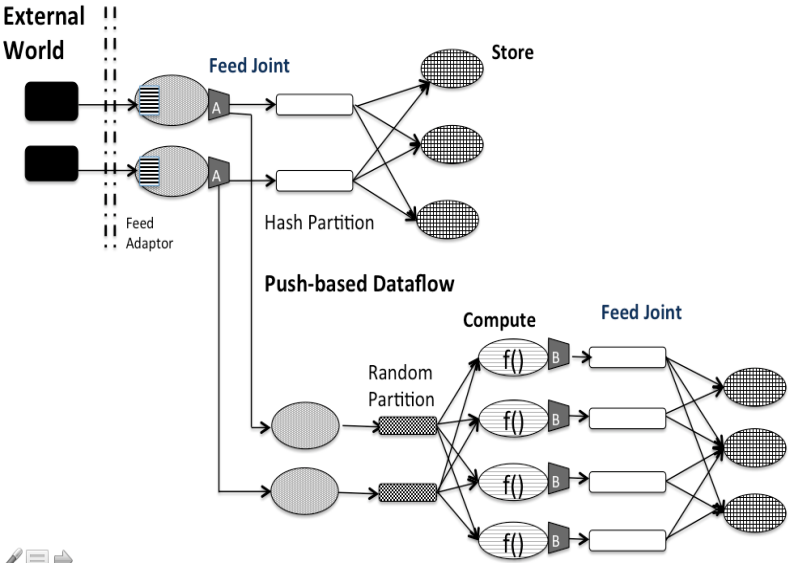
An expensive UDF or an increased rate of arrival of data may lead to an excessive demand for resources leading to delays in the processing of records. Left unchecked, the created *back-pressure* at an operator instance can cascade upstream to completely 'lock' the flow of data along the ingestion pipeline.

To illustrate the potential ill effects of congestion, we consider an example cascade network that involves a pair of primary and secondary feeds, as shown in Figure 7.1(a). We assume that the UDF associated with the secondary feed is computationally expensive and that under a given resource-constrained environment, the records arriving from the external data source cannot be processed via the UDF at their rate of arrival. Under such a scenario, the operators in the secondary compute stage (where the UDF is applied) become 'overloaded' in the sense that the input buffer associated with each operator instance fills up. This precludes the parent operator instances (from the *intake* stage) from pushing data downstream and causes their respective input buffers to fill up; as a result the shared parent operator instances will soon be in a similar 'overloaded' state. At this stage, the data ingestion pipeline for the secondary feed is in a 'locked' state, wherein the *intake* and *compute* stages are 'overloaded'. The secondary *store* stage may still have a backlog from the set of records that are being sent downstream by the compute stage. However, as the compute stage is not receiving additional input, the finite backlog at the compute stage gets processed in due time, subsequent to which, the compute stage discontinues sending data. This causes the downstream secondary store operator instances to become 'idle' with no data to process.

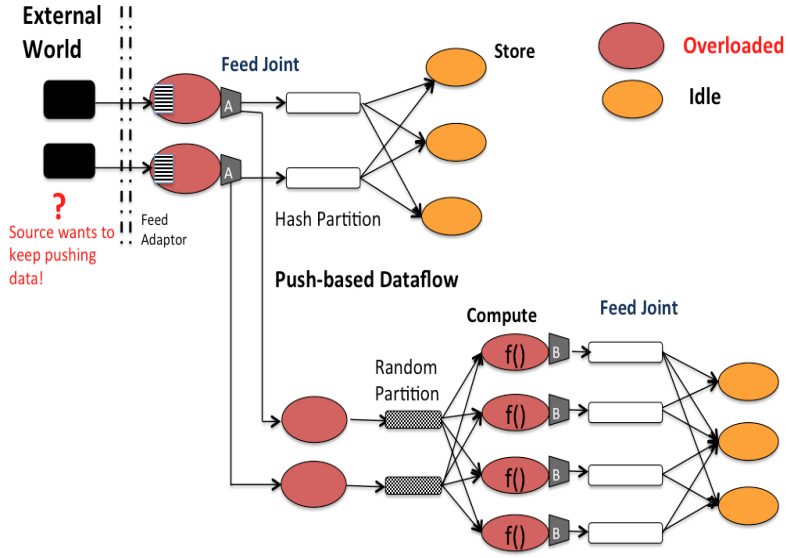
Note that 'locked' state of the data ingestion pipeline for the secondary feed increases memory pressure at the source feed joints (labeled 'A' in Figure 7.1(a)), as data frames at the feed joint are required to be held in memory until consumed. If the external data source is pull-based, AsterixDB can regulate the rate of arrival of data at the intake stage by not requesting data.[2] In contrast, in the

[2]Note that such a strategy is not optimal, as it restricts the flow of data to every other data ingestion pipeline that is receiving data from the source.

case of push-based ingestion, the data source continues to send data at its regular rate irrespective of the ability of the receiver (AsterixDB in the current context) to consume or digest the data. Popular data sources such as Twitter follow a policy of disconnecting the receiver if it is unable to receive data at the required rate.



(a) Store Stage



(b) Intake Stage

Figure 7.1: Congestion in a Data Ingestion Pipeline

As shown in Figure7.1(b), the congestion created from application of an expensive UDF along a data ingestion pipeline has cascaded across the other ingestion pipeline and resulted in a 'locked'

state with no flow of data into AsterixDB. To avoid such an undesirable situation, AsterixDB takes a different approach by resolving congestion at the overloaded operator and preventing it from escalating upstream as back-pressure. This isolates other operators in the pipeline or in the cascade network from the created congestion. However, to take a corrective action, it is important to be able to detect congestion and locate its origin. This requires monitoring the dataflow.

7.2 Monitoring a Data Ingestion Pipeline

In this section, we describe the methodology for monitoring a data ingestion pipeline to detect and locate a resource bottleneck, which is a pre-requisite to taking corrective action (challenge C3 from Section 1.1). An AsterixDB node runs as a Java process (JVM) that is configured with a limit to the amount of available memory. Besides supporting the flow of data along an ingestion pipeline, an AsterixDB node also participates in execution of queries that involve aggregation, joining, and sorting of data and manages their memory usage. The total available memory is shared by operators serving multiple concurrent feeds or queries. Operators participating in a data ingestion pipeline are collectively referred to as feed operators. To ensure sufficient resources for concurrent queries, a fixed (but configurable) limit is imposed on the total memory that can be allocated to the operators (intake, collect, store, etc.) at each worker node (Node Controller)[3] that is participating in a data ingestion pipeline.

During normal operation, when the rate of arrival of records is lower than what an ingestion pipeline can consume and persist, the participant operator instances use a limited amount of memory as each stores input and output frames in reusable buffers. However, feed records may be pushed to AsterixDB at a rate that is higher than what the ingestion pipeline may consume. To prevent data loss in such a situation, additional memory is required by the intake operator instances

[3]Parameter *feed.memory.budget* defined in AsterixDB configuration defines the memory budget in terms of number of bytes.

to hold the arriving records in memory-resident buffers until the preceding records are processed and sent downstream. Each operator in a data ingestion pipeline is provided with an in-memory buffer (referred to as *input buffer*, hereafter) that is used to stage the arriving records before these are picked up for processing by the operator. The input buffer associated with each operator is expandable as long as the total memory allocation across all feed input buffers is below the configured threshold.

We define R_P as the rate of processing of records by an operator measured as records/sec. R_P is a function of the computational complexity of the task being performed by the operator and is influenced by the availability of resources at the node hosting the operator instance. Additionally, we denote the rate of arrival of records at an operator's input as R_A, measured as records/sec. In the scenario when the rate of arrival of records (R_A) at an operator's input exceeds the rate of processing of records (R_P) by the operator, additional memory is required to hold the arriving data frames. This may require the input buffer to be expanded. Exhaustion of the memory budget is a symptom of congestion or the inability of the operator to process data frames at the required rate, an undesirable situation that needs to be detected early. An operator is thus monitored to periodically measure R_A, R_P, and the length of its input buffer. The monitoring functionality at an operator is provided by a ***MetaFeed*** operator that wraps around the actual operator (also known as the *core* operator).

Data shall continue to flow along a data ingestion pipeline as long as for every pair of a sender (upstream) and receiver (downstream) operator, there is sufficient memory at the receiver to allow the sender to continue to push data at its regular rate. The data arrival rate for a given feed may not be statically determined. AsterixDB does not attempt to forecast the data arrival rate or the demand for additional memory. It introduces a notion of a *congested* state. An operator for which, the length of the input buffer increases beyond a configured threshold and stays above for a threshold duration, is marked to be in a *congested* state. A congested state of an operator, if left unchecked can potentially exhaust the memory budget, preclude the corresponding sender opera-

tor from pushing data downstream and thus lock of the data flow along the pipeline, in a fashion similar to the example shown earlier in Figure 7.1(b).

7.3 Ingestion Policies

In a congested state, the records that are staged in the *input buffer* of an operator cannot be processed for lack of resources and are referred to as *excess* records. On detecting a *congested* state of an operator, the corrective action to be taken with respect to the excess records is determined by the ingestion policy associated with the data feed. To study the different alternatives and evaluate the benefits and downsides of each in terms of their impact on throughput and latency, we constructed an experiment that involved creating an 'artificial' bottleneck in a data ingestion pipeline.

The experiment involved a 10 node AsterixDB cluster. Additionally, a pair of physical machines, located outside the AsterixDB cluster, ran our custom data generating application – *TweetGen*, that emulated the Twitter service by allowing clients to request and have artificial tweets streamed to a configured socket address. *TweetGen* allowed configuring a pattern for generation of tweets wherein the rate of sending tweets (tweets/sec or **twps**) could be varied during defined intervals of time. We configured the pair of instances of *TweetGen* to collectively generate and send tweets (to a requesting client) such that the aggregate rate of generation of tweets followed the pattern shown in Figure 7.2. The pattern involves equi-width workload-phases with mid, high and low activity in terms of the rate of arrival of tweets. These workload-phases are referred to as W_{MID}, W_{HIGH} and W_{LOW} respectively and the corresponding rate (twps) is denoted by R_{MID}, R_{HIGH} and R_{LOW}. T_{start} and T_{stop} denote the time when data source starts and stops pushing data respectively. In our experiment, T_{stop} was 1200 seconds.

In order to retrieve data from our example external source (*TweetGen*), we wrote a custom data feed adaptor – *TweetGenAdaptor*. We created a target dataset – *Tweets* to persist the retrieved

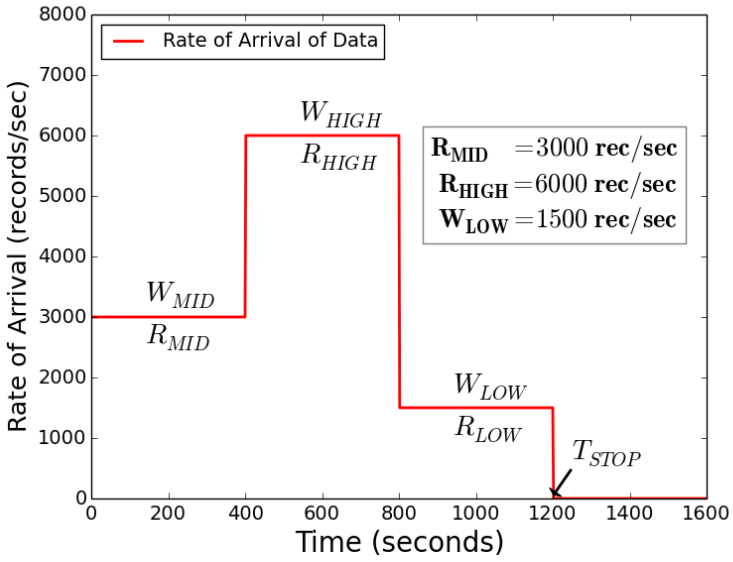


Figure 7.2: Rate of Arrival of Data

data. For the AQL statements for defining the dataset and the associated datatype, the reader is referred back to Listing 3.1 from Section 3.1.2 of Chapter 3. Additionally, we associated with the feed, a Java UDF (*introduceDelay* that involved a busy-spin loop to introduce a delay of ~3ms per invocation. Listing 7.1 shows the *create feed* AQL statement. The intake stage of the data ingestion pipeline involved two instances of *TweetGenAdaptor*, each running at different AsterixDB nodes. The target dataset had a partition on a disk at each node. The store stage thus involved a store operator instance on each node. The AQL statement for connecting the *TweetGen* feed to the target dataset followed the template shown in Listing 7.2.

The compute stage (as constructed by the AsterixDB compiler) offered a similar degree of parallelism and involved a compute operator instance on each node. Application of the UDF (with its ~3ms execution time) by a compute operator instance gave each one a maximum processing capacity of ~300 twps. The aggregate capacity from 10 parallel instances was thus limited to 3000 twps (referred to as $\text{Compute}_{\text{LIMIT}}$). In the workload (Figure 7.2), we have $R_{\text{HIGH}} > \text{Compute}_{\text{LIMIT}}$ during W_{HIGH}. This leads to congestion, a situation where records cannot be processed at their rate of their arrival. We repeated the experiment using each of the built-in ingestion policies (Table 4.2

```
use dataverse feeds;

create feed TwitterFeed using "tweetlib#TweetGenAdaptor"
(("server" = "server1:9001, server2:9002"),
 ("type_name" = "Tweet"))
 apply function tweetlib#includeBusySpinLoop;
```

Listing 7.1: Definition of the feed used in the workload experiment

```
use dataverse feeds;

connect feed TwitterFeed to dataset Tweets
using policy <name of ingestion_policy >;
```

Listing 7.2: A template for a *connect* feed statement that uses a given ingestion policy

from Section 4.5 under Chapter 4) in the template shown in Listing 7.2.

Table 7.1 lists the symbols and metrics we use when describing this experiment and its results. The intake stage involved a pair of feed adaptor instances each receiving records from a separate TweetGen instance located outside the AsterixDB cluster. Each TweetGen instance pushed data for a continuous duration of 1200 seconds ($T_{start} - T_{stop}$). We measured the *instantaneous throughput* as the number of tweets persisted in each 2 second interval over the duration of the experiment. We also measured the *ingestion latency* (Table 7.1(b)) for each tweet received by the feed adaptor during each workload-phase (W_{MID}, W_{HIGH} and W_{LOW}). Next, we discuss the results.

7.3.1 Basic Policy

The *Basic* policy dictates that the *input buffer* at an operator be expanded to accommodate the arriving records until the total memory allocated reaches the configured memory budget. Under

Table 7.1: Symbols and Metrics

(a) Symbol Definitions

Symbol	Definition
T_{start}, T_{stop}	Time when data source starts/stops pushing data
$T_{intake}(i)$	Time when Tweet(i) is received by the feed adaptor
$T_{indexed(i)}$	Time when Tweet(i) is indexed in storage
N_{total}	Total number of tweets received by feed adaptor
$N_{indexed}$	Total number of tweets indexed
T_{done}	Time when ingestion activity completes.

(b) Metric Definitions

Metric	Definition
Instantaneous Throughput (t)	$(N_{indexed}(t) - N_{indexed}(t-w))/w$, $w = 2\ seconds$
Ingestion Latency (i)	$T_{indexed}(i) - T_{intake}(i)$
Ingestion Coverage	$N_{indexed}/N_{total}$

this policy, if the rate of arrival of records (R_A) does not decrease or the rate or processing (R_P) does not increase, the operator continues to be in the congested state, and its memory footprint continues to increase. When the memory footprint reaches the threshold, the arriving records cannot be staged in the operator's input buffer and are required to be discarded, thus resulting in data loss. Note that the upstream sender operator remains agnostic of the congested state of the downstream operator and continues to send records at the regular rate. As such, it is isolated from the congestion occurring downstream.

Figure 7.3 shows the instantaneous throughput plotted on a timeline for the Basic policy. The figure also cites the *average ingestion latency* (L_{Avg}) during each workload-phase. It is desirable to maximize the *ingestion coverage* (Table 7.1(b)), minimize the average ingestion latency for each workload-phase and have $T_{DONE} \sim T_{STOP}$. During W_{MID}, the rate of arrival of records at the compute stage is ~3000 twps, which is within the $Compute_{LIMIT}$ offered by the data ingestion pipeline. As such, no excess records are created during this workload phase. This is also evident from a low value of *Average Latency* (L_{Avg} (W_{MID}) = 0.65 seconds). However, with the beginning of the W_{HIGH} phase, the data arrival rate doubles to ~6000 twps, which is twice the $Compute_{LIMIT}$ offered

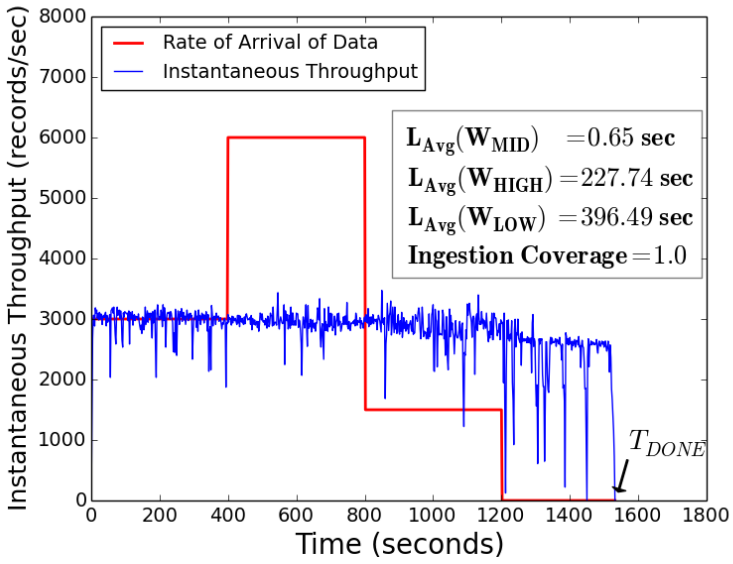


Figure 7.3: Basic Policy

by the cluster. As R_A is twice of R_P, the excess records begin to accumulate (and wait) in the input buffer associated with each operator belonging to the compute stage. This behavior results in an increased value of *Average Latency* ($L_{Avg}(W_{HIGH})$ = 227.74 seconds) for records belonging to this heavy-traffic high activity phase. During the final workload phase (W_{LOW}), R_A drops by a factor of 4 and becomes half of R_P. As $R_A < R_P$, the congested operator instances are able to process the accumulated excess records and clear the backlog from the previous phase. As the excess records from W_{HIGH} are processed prior to the records arriving in W_{LOW} phase, the records received during the W_{LOW} phase incur a longer waiting period resulting in a higher *Average Latency* ($L_{Avg}(W_{LOW})$ = 396.49 seconds).

The Basic policy was able to ingest all records (ingestion coverage = 1.0) which is attributed to the drop in R_A during W_{LOW} that provided an opportunity to process previous backlog, reclaim the input buffer and thus reduce the memory footprint. In an adverse situation, if R_A did not decrease, the Basic policy would have resulted in crossing the memory budget threshold thus leading to data loss arising from discarding of the excess records. Also note that T_{DONE} exceeded T_{STOP} due to the *excess records* created during W_{HIGH}.

7.3.2 Spill Policy

The Spill policy requires an operator to 'spill' to the local disk the *excess records* when the in-memory input buffer has been exhausted and cannot be expanded further. As arriving records are redirected to the local disk, the operator is given an opportunity to clear its backlog by processing records from its input buffer thus reducing its memory footprint. As and when the input buffer has empty slots, records from the spillage on disk are fetched into the input buffer for processing. Note that the arriving records are processed in their order of arrival. The Spill policy allows the end-user to limit the amount of spillage (measured as bytes on disk) to limit disk usage.

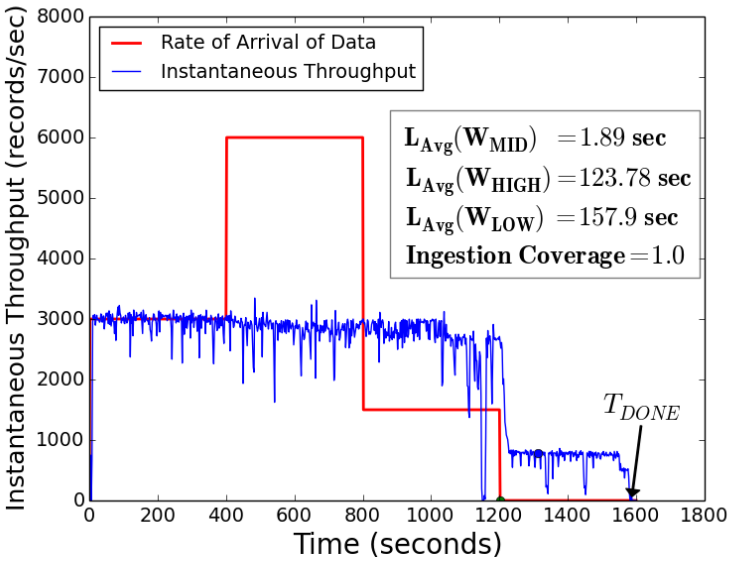


Figure 7.4: Spill Policy

Figure 7.4 shows the instantaneous throughput plotted on a timeline for the Spill policy. *Excess records* are generated during the W_{HIGH} workload phase when rate of arrival of records at the compute stage exceeds the Compute $_{\text{LIMIT}}$. As these are spilled to disk and thus made to wait, the average ingestion latency – $L_{Avg}(W_{HIGH})$ – increases (123.78 seconds). The spillage also causes T_{DONE} to exceed T_{STOP}. Note that unlike the Basic policy, the Spill policy uses a cheaper resource (local disk) for storage of excess records. This allows the operator to maintain a low memory footprint and additionally provides a greater capacity for storage of excess records before a limit is

reached, subsequent to which records would need to be discarded.

7.3.3 Discard Policy

The Discard policy follows a simplistic strategy of not processing any *excess records* by discarding them altogether. Intuitively this can result in data loss resulting in a lower *Ingestion Coverage* . Figure 7.5 plots the instantaneous throughput over a timeline when Discard policy was in use.

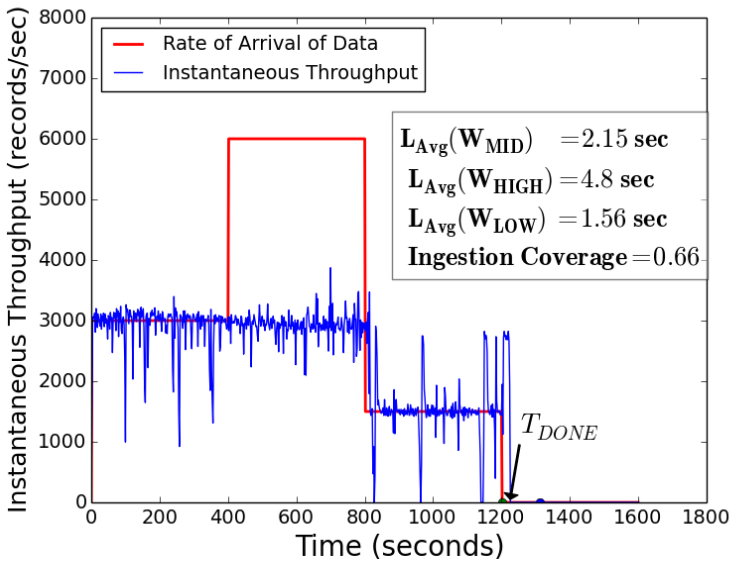


Figure 7.5: Discard Policy

During W_{MID}, the rate of arrival of records is within the cluster $Compute_{LIMIT}$. This phase observes a low value for ingestion latency ($L_{Avg} = 2.15$ sec). In contrast the W_{MID} results in generation of *excess* records as rate of arrival of records far exceeds the cluster $Compute_{LIMIT}$. These records are not made to staged in memory or the local disk. The average *ingestion latency* (L_{AVG} remains low during each of the workload phases. However the flip side is a low *ingestion coverage* of 0.66 which is indicative of a loss of a third of the ingested records. Note that since there isn't any backlog created during the heavy traffic W_{HIGH} workload phase, T_{DONE} equals T_{STOP}.

7.3.4 Throttle Policy

The Throttle policy requires the rate of arrival of data at an operator to be dynamically regulated in accordance with the processing capacity of the operator such that no excess records are generated. This is achieved by periodic monitoring of the processing capacity of an operator (records/sec) and sampling each arriving input data frame to effectively reduce the data arrival rate. Considering our example workload, the processing capacity (R_P) for an operator belonging to the *compute* stage is $\sim$ 300 records/sec . During the heavy traffic W_{HIGH} phase, the rate of arrival of records at each operator instance is 600 records/sec. The sampling rate (R'_A) is determined by R_P / R_A, which is 0.5 in our example workload. This for each data frame, only one half of the records are retained for processing via random selection.

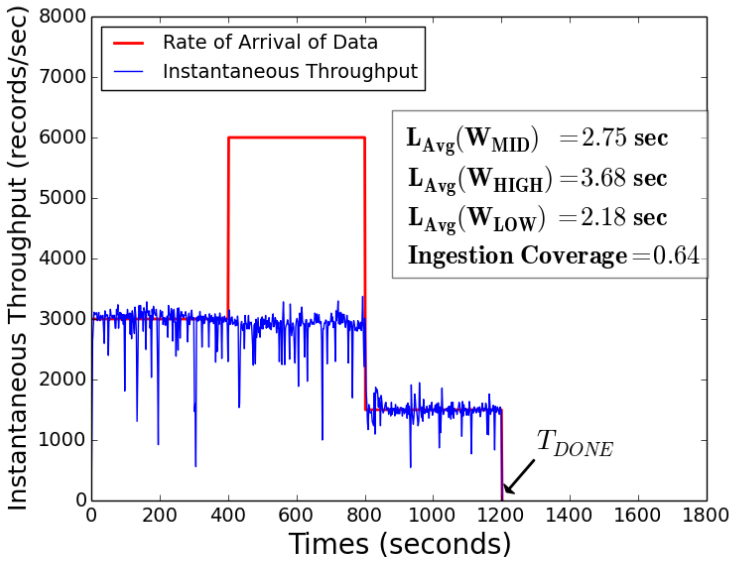


Figure 7.6: Throttle Policy

Figure 7.6 shows the instantaneous throughput plotted on a timeline for the Throttle policy. The average ingestion latency during each workload phase remains low, as operators are able to process the input records without introducing any waiting delays.

7.3.5 Elastic Policy

So far, we have described the use of buffering, spilling, discarding or throttling as mechanisms for dealing with congestion. These mechanisms constitute 'local' resolution and remain hidden from the upstream operators that continue to send data seamlessly. The downside of the described mechanisms is a delay in processing of records (buffering/spilling) or a loss of some of them altogether (discarding/throttling).

Congestion that occurs specifically due to application of an expensive UDF at the compute stage can be cleared in yet another way – 'global' resolution. Congestion at an operator can be considered as a performance failure and recovering from it then involves increasing the degree of parallelism at the operator. This requires re-structuring the ingestion pipeline. In an opposite scenario, the data arrival rate at an operator can be less than the rate at whcih operator can consume records (a.k.a processing capacity measured in terms of records/sec). In such a case, the degree of parallelism for the operator (number of concurrent instances) can be reduced without causing any congestion. The act of increasing the degree of parallelism is referred to as *scaling-out* whereas the opposite act of reducing the degree of parallelism is referred to as *scaling-in*.

The Elastic policy allows for re-structuring of an ingestion pipeline in accordance with the rate of arrival of data. Increasing or decreasing the parallelism during application of an UDF exploits the stateless and therefore embarrassingly parallel nature of the UDF. Next we describe the mechanism for such dynamic re-structuring (scaling in and scaling out) in both kinds of scenarios and study the result obtained from running our test workload.

Elastic Scale-out Protocol

Recall that each operator in a data ingestion pipeline is an instance of the *MetaFeed* operator that wraps around an inner operator a.k.a the *core* operator. Arriving data frames are processed by the

core operator while the MetaFeed operator provides the functionality for dynamic monitoring of data arrival rate, the input buffer and the data processing capacity of the enclosed core operator. Such a design ensures that core operators remain simple and reusable for other Hyracks jobs, while the common functionality (monitoring) is abstracted out and not repeated when implementing core operators.

The MetaFeed operator reports a *congested* state of a compute operator to the local Feed Manager (FM) together with the last measured values for R_A and R_P. Congested states occurring across the cluster are reported by the FM at each worker node (Node Controller) to the Central Feed Manager (CFM) located at the master node (Cluster Controller). The Central Feed Manager uses reported values of R_A and R_P, to compute the required degree of parallelism at the compute stage, N', as $(avg(R_A)/avg(R_P))$; such evaluation ensures that with the adjusted degree of parallelism (N'), the rate of arrival of records at an operator instance (at compute stage) is $\sim R_P$ and thus the arriving data can be processed without causing any congestion.

Subsequently, the Central Feed Manager forms a revised Hyracks job specification for the ingestion pipeline to reflect the increased degree of parallelism at the compute stage. In doing so, the additional compute operator instances may run on idle nodes from the cluster or be scheduled on the current set of nodes to utilize additional cores. The location constraints for other operators are set in a way to ensure that the instances are located at their respective locations from the previous run. The Central Feed Manager sends a message — *STALL* — to the respective Feed Manager on each of the worker nodes (Node Controllers). The *STALL* message indicates the intention to restructure the ingestion pipeline and is forwarded by each Feed Manager to each of the (local) MetaFeed operator instances that are a part of the ingestion pipeline.

The responsive action taken by a MetaFeed operator instance differs based on the stage (intake, compute, store) that it's part of. At the intake stage, the MetaFeed operator instance switches to a *buffer-only* mode where data records are held in memory and not forwarded to the compute stage. This cuts off the flow of records downstream. At the compute and the store stages, the MetaFeed

operator instances continue to consume the (existing) records from their respective input buffers. The goal here is to drain the pipeline of any data frames such that the operators at the compute and store stage become idle and are left with no pending data frames to process. At this stage, the Central Feed Manager is notified and a revised pipeline is scheduled to run on the AsterixDB cluster. Next, we describe how the revised pipeline resumes the flow of data.

Let $Op_{stg}^{old(i)}$ denote the $i'th$ instance of an operator that belongs to stage - stg - from the previous (old) run. The corresponding operator from the restructured pipeline is denoted by $Op_{stg}^{new(i)}$. Note that the location for $Op_{stg}^{old(i)}$ is same as that for $Op_{stg}^{new(i)}$ $\forall stg \in (intake, compute, store)$. Furthermore, $Op_{intake}^{old(i)}$ $\forall i$ is in the ***buffer-only*** mode and $Op_{compute}^{old(i)}$ and $Op_{store}^{old(i)}$ $\forall i$ do not have any pending data frames to process. These instances are allowed to end gracefully and release any resources (input buffers) back to the local pool for reuse. At the intake stage, $Op_{intake}^{new(i)}$ takes ownership of the input buffer associated with $Op_{intake}^{old(i)}$ such that the records flowing into $Op_{intake}^{old(i)}$ are now accessible to $Op_{intake}^{new(i)}$. $Op_{intake}^{new(i)}$ begins to forward the records downstream to the compute stage which now has an increased degree of parallelism. In the event that the rate of arrival of records further increases and causes a congestion at the compute stage, the Central Feed Manager is notified with meta-information (R_A and R_P) and the pipeline is revised using the mechanism, we just described.

Elastic Scale-in Protocol

Contrary to dynamically scaling out an operator, AsterixDB also provides for auto-scaling-in at the compute stage, if the current degree of parallelism is greater than that required to handle the flow of data. The possibility of reducing the degree of parallelism at the compute stage is evaluated by monitoring of the arrival rate (R_A), the processing rate (R_P) and the size of the input buffer at each of the operator instances belonging to the compute stage. If $R_A < R_P$, then the degree of parallelism at the compute stage can be reduced by a factor – $N_{REDUCTION} = R_P/R_A$. However scaling-in at the compute stage should only be triggered if it offers significant benefit by a substantial reduction in the number of employed operator instances. In AsterixDB, we follow a basic

heuristic that triggers a scale-in only if $N_{REDUCTION} > 1.33$. Expressed otherwise, scale-in is triggered only if the degree of parallelism can be reduced by at least 25%. The threshold value for $N_{REDUCTION}$ can be set as an ingestion policy parameter.

The mechanism for scaling-in is similar in part to the mechanism followed for scaling-out. The Central Feed Manager sends a *STALL* message to the Feed Manager at each of the worker nodes (Node Controllers). This message is then forwarded to each of the MetaFeed operator instances running on each Node Controller by the local Feed Manager. As in the case of scaling-out, the data ingestion pipeline is made to process all pending records while the intake stage restricts forwarding of any received records downstream. At the stage when the compute and store stage are idle (operators have empty input buffers), the Central Feed Manager is notified and a revised ingestion pipeline (a Hyracks job) is scheduled to run on the cluster. The location for operators in the revised pipeline is set in way such that intake and store stage completely overlap with the corresponding locations in previous ingestion pipeline. The intake stage operator instances in the revised pipeline take ownership of the input buffer from their corresponding instances from the previous execution. Note that these input buffers are continuously being appended with records received from the external data source. These records are now allowed to be sent downstream for processing at the compute stage thus resuming normal flow of data, but with lesser degree of parallelism at the compute stage.

Experimental Evaluation

We revisit our multi-phase workload experiment described in Section 7.3 and repeat it using 'Elastic' as the feed ingestion policy. Figure 7.7 shows the instantaneous throughput plotted on a timeline when the Elastic policy is in use. The figure also cites the *average ingestion latency* (L_{Avg}) during each workload-phase (W_{MID}, W_{HIGH} and W_{LOW}) and the achieved *ingestion coverage* from running the complete workload. During the initial phase (W_{MID}) of the workload , the data arrival rate is within digestible limit (Compute $_{LIMIT}$) of the cluster with no observed congestion. However,

as the data arrival rate increases (becomes twice) to 6000 twps, the degree of parallelism (10) at the compute stage is insufficient to keep pace with the arriving data. The Elastic policy triggers scale-out at this stage. The required degree of parallelism at compute stage is increased by a factor that is given by $\lceil R_A/R_P \rceil$ (which evaluates to $\lceil (6000/3000) \rceil$ = 2).

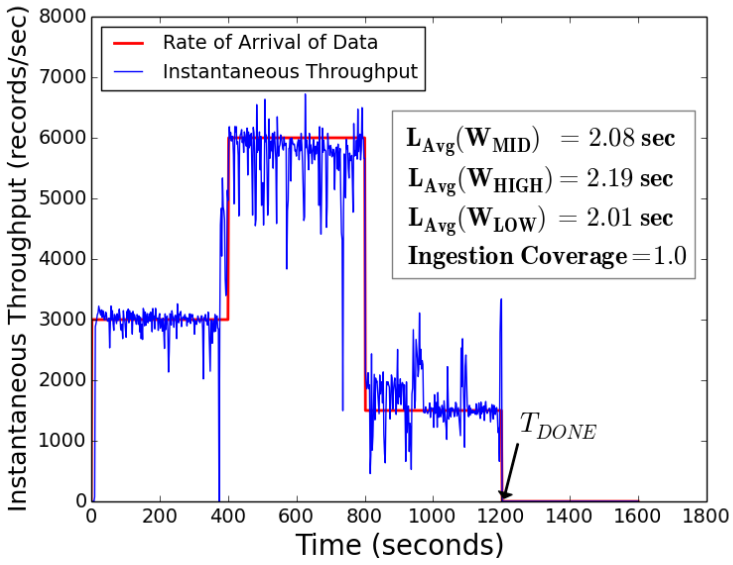


Figure 7.7: Elastic Policy

The ingestion pipeline is restructured to have 20 compute operator instances at the compute stage raising the Compute $_{\text{LIMIT}}$ of the cluster to 6000 twps. Each node then had two compute operator instances that provided a better utilization of the cores (4) on each node. This change is observed in Figure 7.7 at the beginning of W_{HIGH} workload phase. Unlike any other policy, the instantaneous throughput matches the rate of arrival of data during this phase. As the workload enters the low traffic phase (W_{LOW}), the data arrival rate falls much below the Compute $_{\text{LIMIT}}$ offered by the cluster. In the quest to relinquish resources, the Elastic policy triggers a scaling-in at the compute stage such that the offered Compute $_{\text{LIMIT}}$ (6000 twps) matches the data arrival rate (1500 twps). The degree of parallelism at the compute stage is thus decreased by a factor of 4 with only 5 compute operator instances at the compute stage.

The Elastic policy was evaluated to be superior to the rest. Dynamic scaling out of the compute

stage to match the data arrival rate helped provide a complete ingestion coverage (1.0) as no data records were required to be discarded. The average ingestion latency observed for each workload-phase did not vary across the different workload phases, while the workload was still completed with $T_{\text{DONE}} \sim T_{\text{STOP}}$.

7.4 Discard versus Throttle

The Discard policy and the Throttle policy are both closely related in the sense that both may result in data loss and provide a lower value for *ingestion coverage*. However, there is a subtle difference. With respect to the external data source, use of Discard policy can result in a continuous period when none of the records received by AsterixDB are persisted. To understand and demonstrate the difference, we conducted an experiment that involved a single instance of TweetGen and a 3 node AsterixDB cluster. We chose each stage of the data ingestion pipeline to not involve any parallelism.

We configured the single instance of TweetGen to generate data as per the pattern shown in Figure 7.8. This pattern is similar to the workload pattern from our earlier experiment (Figure 7.2), but with different values for the rate of arrival of data during the three workload phases – W_{MID}, W_{HIGH} and W_{LOW}. To cause congestion, we made use of a UDF that involved a busy spin loop to introduce a per-invocation delay of ~3 ms. Application of the UDF (with its ~3ms execution time) by a single compute operator instance gave it a maximum processing capacity of ~300 twps (which was also the $\text{Compute}_{\text{LIMIT}}$), given that the compute stage did not have any parallelism. In the workload (Figure 7.8), we have $R_{\text{HIGH}} > \text{Compute}_{\text{LIMIT}}$ during W_{HIGH}. This leads to congestion, a situation where records cannot be processed at their rate of their arrival.

Each tweet was appended at the intake stage with an additional attribute – record ID – which was chosen to be a monotonically increasing integer starting with 1. Following the data generation

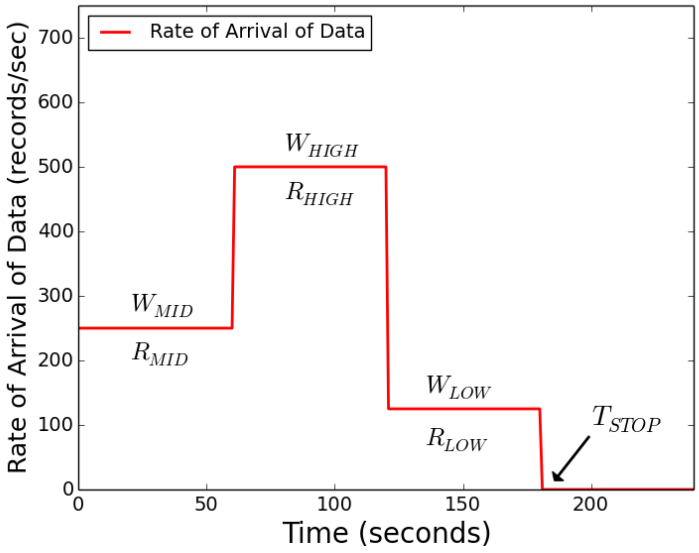


Figure 7.8: Rate of arrival of data

pattern, the TweetGen instance discontinued sending records at t = 180 secs. At this stage, we examined the set of record IDs that were successfully persisted. We conducted the experiment using both Discard and Throttle as the ingestion policy. Figure 7.9 illustrates the pattern formed by plotting a value of 1 corresponding to a record ID (a monotonically increasing integer value) if the record was persisted and 0 vice versa.

Next, we discuss the results.

- **Discard**:

  During the workload phase – W_{MID}, the rate of arrival of data (R_A) is less than Compute $_{\text{LIMIT}}$. There is no congestion and as such all records received during this period are persisted. Record IDs corresponding to the workload phase W_{MID} belonged to the range (1-15000). This resulted in a straight-line pattern till record ID – 15000. However, as we enter the high-activity workload phase – W_{HIGH}, R_A increases beyond the Compute $_{\text{LIMIT}}$ resulting in congestion. As dictated by the Discard policy, the excess records are discarded. The action of discarding records results in period of time when no records are persisted. Discarding

the arriving records also provides an opportunity to clear the existing backlog. Clearing the backlog allows the arriving records to be processed and be persisted. However, as R_A continues to exceed the Compute $_{\text{LIMIT}}$ during this workload phase, the lack of resources causes records to be discarded. A repeated pattern is produced, as shown in Figure7.9.

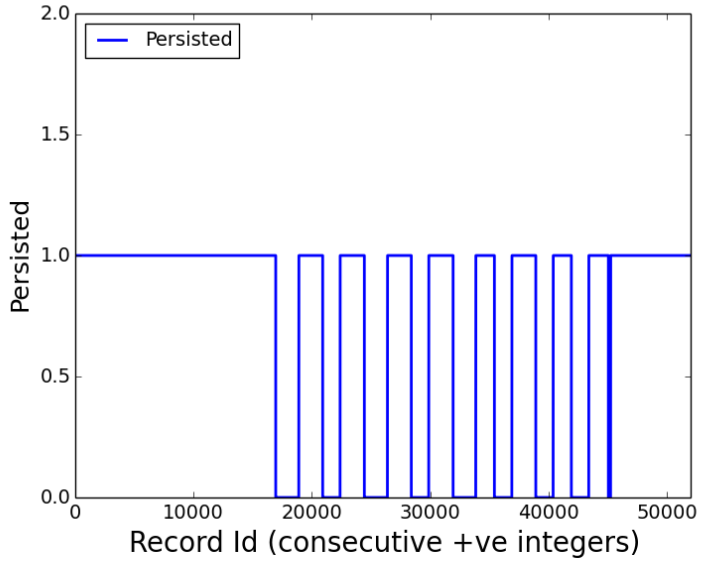
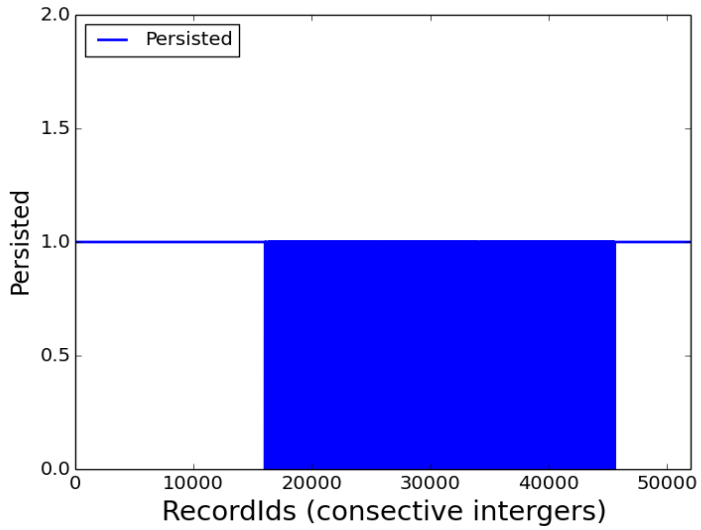


Figure 7.9: Handling of excess records by Discard policy: Pattern formed by plotting a value of 1 for a persisted record ID and 0 otherwise

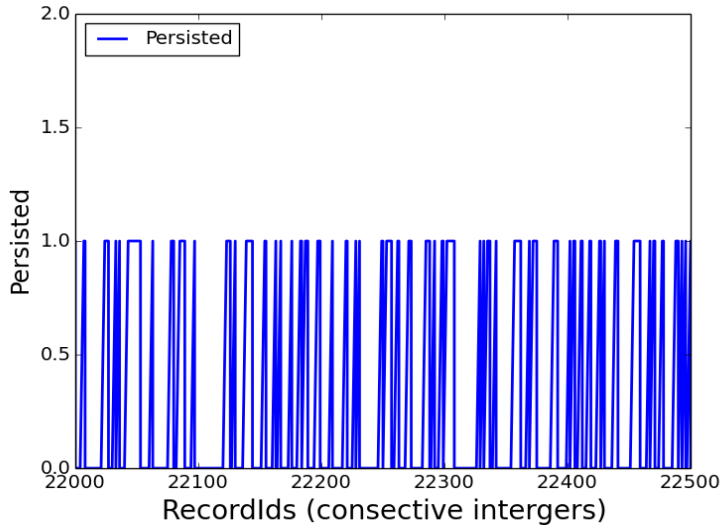
- **Throttle**:

  The effects of the chosen ingestion policy come to play only during the high-activity workload phase — W_{HIGH}, when the rate of arrival of data R_A exceeds the Compute$_{\text{LIMIT}}$ of the cluster. Figure 7.10(a) summarizes the behavior when Throttle policy is chosen. During W_{HIGH}, the Throttle policy works by regulating the rate of arrival of data at the compute stage (Note that data continues to arrive at the intake stage at its regular rate – 500 twps). Each data frame is reduced to a smaller-sized random sample of the contained records to effectively match the rate of arrival of data with Compute$_{\text{LIMIT}}$ (300 twps) offered by the cluster. The throttling behavior appears as persisted record IDs interspersed with the discarded ones, as shown in the region – (10000 - 30000) on the record ID horizontal axis in

Figure 7.10(a). The exact pattern is visible only when viewed under a higher resolution. Figure 7.10(b) illustrates the pattern, depicting it for a small range of record IDs (22000-22500) to bring out the hidden detail.



(a) Pattern formed by plotting a value of 1 for a persisted record ID and 0 otherwise



(b) A high-resolution view of Figure 7.10(a) showing a smaller range of records

Figure 7.10: Handling of excess records by Throttle Policy

Loss of data that corresponds to a continuous stretch of time may not be acceptable to the overlying application that wishes to consume the persisted data. In contrast, the Throttle

policy ensures that for every data frame, a fraction of the contained records (as determined by R_P/R_A) are persisted. The different between Discard and Throttle policies in treatment of excess records becomes less significant when the stages (compute, store) in a data ingestion pipeline involve a higher degree of parallelism. This is because, the individual operator instances at the compute or store stage apply the policy (Discard or Throttle) to their local input and do so independently of other instances. There is no correlation between the set of record IDs dropped across the multiple operator instances.

To demonstrate the subtle difference between the Discard and Throttle policies, we chose to run the data ingestion pipeline without any parallelism at the intake, compute or the store stage.

7.5 Comparison with Storm + MongoDB

An alternative way of supporting data ingestion is to 'glue' together a streaming engine (e.g., Storm) with a persistent store (e.g., MongoDB) that supports queries over indexed semi-structured data. While Storm and MongoDB are popular for their respective functionality of streaming and persisting/indexing data. However using systems together to combine their functionality doesn't necessarily translate to an efficient system. In this subsection, we intend to explore the combined system and evaluate it using our artificial workload from Figure 7.2.

We begin by giving a brief overview of Storm and MongoDB and how the systems can be combined to support data ingestion. A Storm dataflow offers spouts (that act as sources of data) and bolts (that act as operators) that can be connected to form a dataflow. A spout implements a method, *nextTuple()* that is invoked by Storm in a 'pull-based' manner for obtaining the next record from a data source. However, this method is not compatible with the common scenario of a 'push-based' ingestion where data continues to arrive from the data source at its natural rate. To support push-based ingestion, it is necessary to buffer the arriving records from the data source and then forward

them to Storm on each invocation of the *nextTuple()* method. Another strategy commonly used by the community is to use yet another system — Redis, Thrift, or Kafka — as services (more 'gluing'!) so that records can be pushed to them and then a spout can pull them.

The data records output by Storm can be directed to a persistent store such as MongoDB, that forms a preferred choice in the open-source community for its support for semi-structured data and the ability to construct indexes for faster query execution. In contrast to our declarative support for defining/managing feeds, where the AsterixDB compiler constructs the dataflow, a Storm+MongoDB user must programmatically connect together spouts and bolts and statically specify the degree of parallelism for each. Storm does not offer elasticity, nor does it allow associating ingestion policies to customize the handling of congestion and failures. Interfacing with MongoDB requires the bolts to be parameterized with the locations of MongoDB Query Routers, which are processes running on specific nodes in a MongoDB cluster that accept insert statements/-queries. The end-user is thus required to understand the layout of the cluster and include specific information in the source code.

Experimental Evaluation

We used our 10 node cluster to host Storm and MongoDB. Our 'glued' solution emulated the stages from an AsterixDB ingestion pipeline. The constructed dataflow involves a pair of spouts, each receiving records from a separate TweetGen instance. Each spout's output is randomly partitioned across a set of 10 bolts, one on each node. Each node also hosted a MongoDB Query Router to allow the co-located bolt to submit an insert statement to the local Query Router. Each node also hosted a MongoDB partition server. The MongoDB collection (dataset) was sharded (hashed by primary key) across the partition servers.

MongoDB provides a varying level of durability for writes. The lowest level (non-durable) allow submitting records for insertion asynchronously with no guarantees or notification of success.

The Storm+MongoDB coupling then acts as a pure streaming engine with minimal overhead from (de)serialization of records. However, it becomes hard to reason about the consistency and durability offered by the system. The durable-write mode in MongoDB is a fair comparison with AsterixDB, as it provides ACID semantics for data ingestion. However, to provide a complete picture, we ran the workload of Figure 7.2 using both kinds of writes for MongoDB.

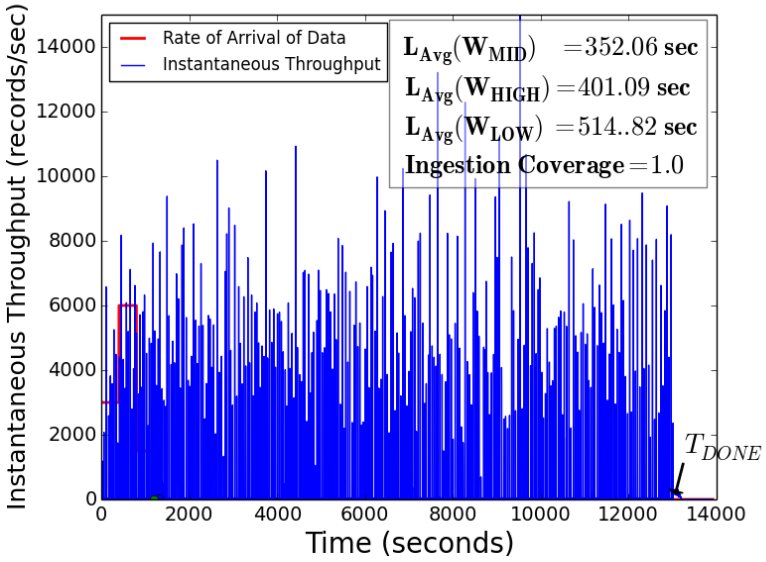


Figure 7.11: Instantaneous Throughput for Storm+MongoDB: Durable Write

The durable-write mode (Figure 7.5) in the Storm+MongoDB coupling provides complete ingestion coverage. However, when compared to Basic, Spill and Elastic policies from AsterixDB (with similar ingestion coverage), the time taken for the ingestion activity to complete (T_{DONE} - T_{START}) increased by a factor of ten — meaning that Storm+MongoDB coupling was unable to keep up with the workload. The average ingestion latency observed in each workload-phase for Storm+MongoDB compared with the Elastic policy was worse by two orders of magnitude. To isolate the cause, we switched to using non-durable writes (Figure 7.12) wherein the system behaves like a pure streaming engine with a de-coupled unreliable persistent store (asynchronous writes). We then obtained $T_{\text{DONE}} \sim T_{\text{STOP}}$. This ruled out inefficient streaming of records within Storm as a possible reason for the low throughput.

To better understand the results, we must consider the processing strategy used by MongoDB. MongoDB optimized for maximum single-record throughput and write-concurrency but at the cost of an increased wait time (~50ms) per write when full durability is requested. This created congestion at the output of the compute stage of our Storm+MongoDB combination and contributed to the high latency and low ingestion throughput. The situation is expected to worsen when 'at least once semantics' are required. Storm achieves such semantics by replaying a record if it does not traverse the dataflow within a specified time threshold. Owing to an increased wait time per write, additional failures would be assumed and records would begin to be replayed; this cycle can repeat endlessly, leading to system instability.

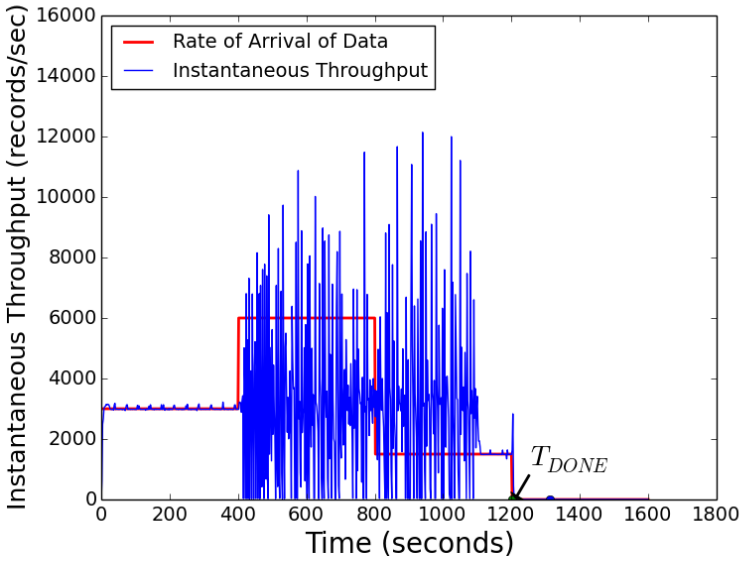


Figure 7.12: Instantaneous Throughput for Storm+MongoDB: Non-Durable Write

7.6 Other Approaches to Dealing with Congestion

A data ingestion pipeline is similar to a stream processing dataflow in the sense that each represents a directed acyclic graph with operators that represent computation and connectors that represent different modes of exchange of data across operator instances. Congestion introduced in a data

ingestion pipeline can also occur in a stream dataflow given the similar dynamic nature in terms of fluctuating rate of arrival of data and application of expensive computations. Stream processing engines (SPEs) too resort to elastic scale out of operators when the data cannot be processes at its current rate of arrival. The elastic reconfiguration protocols can be classified as being operational at an operator-level or at cluster-level. We discuss both strategies and compare with the approach adopted in AsterixDB.

7.6.1 Load Shedding

As we have discussed, the rate of arrival of data may fluctuate in an unexpected manner, raising the demand for resources and putting the system under stress where the available resources are no longer sufficient. The ideal resolution in such a scenario is to add more resources in a dynamic fashion. However, in general, adding more resources may not be feasible at runtime because of multiple reasons. At first, one may not have additional hardware at disposal that could be added to create additional worker nodes (e.g., Node Controllers in AsterixDB). Secondly, it may not be financially feasible even in a cloud setting that offers a flexible 'pay as you go' model. Moreover, the system used for processing of data may itself not have the ability to add worker nodes on the fly and incorporate them in redistributing the load. Finally, the overlying data-driven application that wishes to consume the ingested data may permit data loss and consider the error from executing queries over incomplete data as being acceptable.

A popular method that is useful in such scenarios is *Load Shedding*, which in general terms is defined as the process of dropping excess load from the system. The basic idea there is to not process each and every data record. Data records could be dropped at different stages in a data flow, but of course dropping them earlier reduces the wasted effort. AsterixDB offers two built-in ingestion policies – Discard and Throttle – that allow an end-user to configure load shedding as the method for resolving congestion. Load shedding is not a new idea. It has been widely

studied in the fields of networking [45] and multimedia [22]. Adaptation of the method has also been applied to Stream Processing Engines (e.g., [38]). The use of load shedding as a congestion control mechanism in SPEs is worthy of a discussion given that both stream processing and data ingestion involve dealing with 'push-based' data sources that do not offer an option to regulate the rate of arrival of data from the source.

An SPE workload typically involves a known set of standing queries that are evaluated in a continuous fashion over streams of moving data. In a scenario, where the queries are fixed, the requirement for resources as a function of data arrival rate can be worked out using repeated experimentation [38]. Each continuous query dataflow is then annotated with *load coefficients* that provide a method to quantify the load on the system and compare it against the known capacity of the system expressed in similar units.

The overlying data-driven applications (that are also statically known) may express their requirements (QoS guarantees) in terms of bounds on fraction of data dropped and even declare what kinds of data are not important for the application. For e.g., a sensor-driven application that monitors heart-beats of patients in a hospital may assign more importance to outliers that represent abnormal behavior. Such records may be required to preserve when choosing to discard records during congestion. The load shedding approach adopted in [38] makes use of the inputs provides by the overlying application in statically constructing a list of load-shedding plans (referred to as Load Shedding Road Map – LSRM). Each LSRM describes the placement of *drop* operators in the dataflow whose only purpose is to selectively drop records during periods of congestion. The data arrival rate measured dynamically is used as input to determine the existing load and derive the desired structure of the dataflow that would help resolve congestion.

In dealing with congestion in a data ingestion pipeline, AsterixDB does not assume static knowledge of the workload in terms of number of feeds that would be active or the set of concurrent queries that would compete for resources. In absence of any assumptions or knowledge gained via controlled experimentation, the approach adopted in AsterixDB differs significantly as being

dynamic in nature. AsterixDB resorts to runtime monitoring of each operator in a data ingestion pipeline, the length of input buffers and the amount of memory allocated to each operator from the configured budget.

7.6.2 Operator-Level Elasticity

Operator-level elasticity is targeted at dynamically tweaking the degree of parallelism used within each instance of a given operator in processing the input. The total number of operator instances executing across a cluster remains fixed in this strategy. An operator is modeled to have an input queue for holding the arriving data, a set of worker threads that can do apply an embarrassingly parallel task independently on each data record, a dispatcher thread that distributed input records across the worker threads and an alarm thread that wakes up periodically to measure the processing rate and determine if it is required to alter the degree of parallelism. The operator allocates a pool of worker threads that remain suspended unless they are woken up when the operator needs more threads, that is when it needs to increase the degree of parallelism. In an opposite act of scaling-in, currently active worker threads may be suspended and returned to the pool.

An operator is periodically monitored to measure its performance measured as its rate of processing of records (records/sec) at a given thread-level. The degree of parallelism (number of active threads) is increased by one and the performance is re-evaluated. This cycle of increasing parallelism and evaluating performance is repeated until a further increases causes the performance to not change or decline. At this stage, the operator is said to be performing at its stable peak rate. Any change in the workload or availability of resources results in drop in performance which is detected in the next evaluation cycle. The elastic reconfiguration protocol does not attempt to increase/decrease the number of operator instances that are running across a cluster. Instead it only concerns itself in determining the optimal number of threads for each operator instance. As such, the protocol does not benefit from idle machines lying elsewhere which are not hosting an instance

of the operator, as it does not attempt to create new instances of the operator and restructure the dataflow. A detailed description of this approach can be found in [**?**].

7.6.3 Cluster-Level Elasticity

An alternate approach to elastic reconfiguration of a dataflow is to increase/decrease the number of operator instances executing across a given cluster. This approach, also known as cluster-level elasticity, allows for creating additional operator instances on nodes or relinquishing nodes by eliminating operator instances and is also the approach adopted in AsterixDB. Another system that follows a similar approach is StreamCloud [25]. StreamCloud is a stream processing engine that support elastic scale-out/in of data flows that correspond to the execution of a continuous query on streaming data. The elastic reconfiguration protocol in StreamCloud deals with stateless operators (map, filter etc.) and stateful operators (aggregation, windowing etc.). Data ingestion in AsterixDB does not involve the use of stateful operators such as aggregation. As such, the common ground between the elastic configuration protocol followed by each system is the handling of stateless operators. StreamCloud attempts to maintain the CPU utilization at each node within statically configured upper and lower limits. An elastic scale-out is triggered when the average CPU utilization across multiple nodes that are participating in a continuous query, increases beyond the threshold. Likewise, the opposite action of scaling-in is triggered when the average CPU utilization falls below the lower limit.

Triggering of a scale-out/in action on the basis of measurement of CPU utilization assumes that there is a single query or data flow and that CPU is the bottleneck in its execution. In a dynamic environment, where multiple queries are actively running, measuring of CPU utilization alone does not provide sufficient information as to which particular query and its specific operator needs to be elastically scaled-out (or scaled-in). Furthermore, a query may involve a user-defined-function (UDF) that is blocked on network or disk. Tracking CPU utilization may actually give false alarms

that trigger reconfiguration or may defer reconfiguration at times when it is required to alter the degree of parallelism.

7.7 Summary

In this chapter, we discussed the undesirable scenario of data congestion wherein data arrival rate exceeds the consumption capacity of the data ingestion pipeline. The inability of a data ingestion facility to auto-detect and take corrective action can result in an unacceptable delay in persisting data and making it available for querying by the overlying application. In the worst scenario, the data store may never be able to catch pace with the external data source and may even cause the external source to discontinue sending data (a policy followed by Twitter).

The strategy to deal with data congestion is largely determined by the requirements laid down by the overlying application that wishes to consume the ingested data. In the case when ingesting each received record is not a strict requirement, the end-user may choose the 'Discard' or 'Throttle' policy and have a subset of the data be ingested with minimal delay (ingestion latency). In contrast, for an application that does not permit losing records, an end-user can opt to use the 'Spill' or the 'Basic' policy, but with a downside of an increased ingestion latency. However, if the introduced delays thereof are not acceptable, the 'Elastic' policy acts as a superior alternative that provides for complete ingestion coverage with minimal ingestion latency. The caveat here is that there is a limit to the achieved degree of parallelism, as determined by the number of cores in a given cluster. However, in a pay as you go cloud-based environment, the 'Elastic' policy can provide the required scale-out by having the data ingestion facility add additional nodes dynamically. AsterixDB, currently does not support dynamic addition of nodes to benefit from a cloud-based environment.

In this chapter, we also brought to surface, an interesting result with respect to 'gluing' of other-

wise efficient system to form an combined system that is capable of ingesting data. However, as we observed, the combined uber system loses efficiency arising from the optimization techniques adopted by involved systems that tend to work contrary to each other. We also looked at other strategies for elastic reconfiguration that worked at the level of an operator and allowed a dynamic number of threads to be utilized in the computation performed by each instance of the operator.

Chapter 8

Use Cases

In this chapter, we briefly describe how the data ingestion facility in AsterixDB enables some of the real-life scenarios that involve real-time data ingestion and that require ad hoc analyses of the persisted data via queries. The goal of this chapter is to bring to light some of the practical use cases for data ingestion.

8.1 Knowledge Base Acceleration

Knowledge bases (KB) ([14, 15]) have become indispensable sources of information. Specifically, Wikipedia has been widely used in various information access contexts, including named entity recognition and disambiguation, query modeling and expansion, question answering, entity linking, and entity retrieval. In many of these cases, the role of a knowledge base, such as Wikipedia, is to serve as a "semantic backbone", a repository of entities and their relations.

Keeping up with changes and maintaining up-to-date knowledge is in everyone's best interest. However, this requires a continuous effort by the editors and content managers, and keeping up is becoming increasingly demanding as information is being produced at an ever-growing rate. With

this context in mind, *Knowledge Base Acceleration* (KBA) systems seek to help humans expand knowledge bases like Wikipedia by automatically recommending edits based on incoming content streams. Identifying content items, such as news articles, blog posts and other document sources, that may imply modifications to the attributes or relations of a given target entity is one of the basic steps to be performed by any KBA system.

As a prerequisite, a KBA system requires ingesting data from a variety of data sources, processing and consolidating the collected data into a persisted indexed form. The data ingestion facility in AsterixDB provides this required functionality and enables ad hoc analysis of the collected data via its query language – AQL.

8.2 Publish-Subscribe

A Publish-Subscribe system allows messages from a sender to be delivered to interested receivers where the set of receivers are not known to the senders and may grow and shrink dynamically. Messages are characterized into classes, without knowledge of what, if any, subscribers there may be. Similarly, subscribers express interest in one or more classes, and only receive messages that are of interest, without knowledge of what, if any, publishers there are.

Consider a publish-subscribe system with end-users interfacing with the system using a mobile device and creating and registering subscriptions using the device. The system will also have a set of registered senders or data sources. The system is required to continuously ingest data from the set of sources and have it be persisted and indexed. Querying the collected data allows retrieving the required information that needs to be matched and disseminated to the set of interested subscribers. In an extended version of such a system, the data arriving from external sources could be augmented with additional information that adds value to the message. Such enriched data is also referred to as “actionable data”, as it contains information that enables receiver to take necessary

action. As an example, a user could subscribe to 'events' like accidents, fire, earthquakes, disasters nearby. An augmented version of such a message would also include detours, information on medical services etc.

The data ingestion support provided by AsterixDB can play a key role by enabling the collection of data from external sources. Requests for registration and subscription for specific interests can also be received and persisted as a data feed, as can dynamic situational information such as users' locations. Such a "Big Active Data" system is currently under construction at UC Irvine and UC Riverside, and feeds are providing the required "input side" functionality.

8.3 Analysis of a Twitter Feed

Twitter has proved to be a great data resource for a multitude of data analysis tasks ranging from studying general user sentiment to predicting elections, crime, and flu trends across the globe. The support for data ingestion in AsterixDB can enable continuous collection and storage of tweets. The persisted and indexed data then allows for potential data analyses to train models and carry out predictions for real-world applications.

Some applications being currently considered as part of ongoing project at UC Irvine include (1) identifying and predicting high stress geographical areas over the United States, and (2) investigating the usefulness of Twitter in predicting traffic accidents. After narrowing down to a specific application, machine learning models (e.g., classification models) can be trained on the Twitter data in AsterixDB to make future predictions on new incoming Twitter data. After a sufficient time period, these models should be re-trained on the newly arrived twitter data to ensure that the prediction models are current and have sufficiently high accuracy. As the data arrives via a feed, the output from the prediction will be added to the tweets via a UDF and stored in AsterixDB twitter datasets as additional fields of the same record, for future access and viewing.

The requirement to maintain a reference state (the model) at each UDF and refresh the state dynamically introduces the need to support stateful functions and a mechanism that triggers the state to refresh across each worker node in the compute tier (where the UDF is being applied) to be reloaded.

8.4 Event Shop

Event Shop is a system being developed at UC Irvine that aims at enabling the real time detection of events by analysis of social data feeds (Twitter), and environmental data such as weather, rainfall, pollen count, wind speed, road traffic, and air quality data over different geographical areas of interest. Additional sources of data include news articles, and personal activities such as sharing of location on social media, etc.

The data of interest in Event Shop is spatio-temporal in nature and can be grouped into a buckets in the time domain or into a two-dimensional grid structure based on latitude and longitude. Furthermore, data sources often emit data at different granularities – (city, state, country) – along the spatial domain and along the time domain (mins, hourly, daily). AsterixDB has built-in support for spatio-temporal data, thus providing rich querying capability. The data ingestion support of AsterixDB allows for interfacing with external sources and continuous collection of data. The Event Shop project is now beginning to explore the use of AsterixDB in order to scale their system to "Big Data" settings.

8.5 Summary

The amount of data that gets generated on a continuous basis has grown significantly, and it is being further propelled by the notion of the Internet of Things (IoT) that enables devices to emit

data and add to the ever-going data repository. The ability to be able to ingest data in real time and offer rich querying support appears to have become the need of the hour. In this chapter, we have provided a brief overview of only a few of the many data driven applications that can benefit from the continuous ingestion support now provided by AsterixDB.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

We are living in an era of "Big Data", which is a broad term for data sets being so large or complex that traditional data processing applications are inadequate. Challenges include analysis, capture, curation, search, sharing, storage, transfer, and visualization. The past decade has seen a phenomenal rise in the quest to collect and analyze increasingly larger volumes of data. Data-driven enterprises today view the ability to extract useful information from this influx of data as the key to their success. A multitude of systems have been developed across industry and academia to address the challenges imposed by the volume of data that needs to be processed and mined.

Apart from volume, data also has an associated velocity aspect as it continues to be generated from a wide variety of sources in an uninterrupted fashion. Data that is being generated at a high rate in a continuous fashion is typically referred to as *Fast Data*. The act of collecting and persisting the data can be envisioned as producing a data repository that reaches the many terabyte scale (*Big Data*) in due time and continues to grow beyond. Such a data repository is often classified as an example of *Big Fast Data*. It is desired to subject the collected data to ad hoc queries that involve

a mix of joins, aggregation, group-by and sort operations. Such ad hoc analyses in a typical setting enables overlying applications that generate reports and summaries, provide visualizations over time, and facilitate complex tasks such as data mining and detecting anomalies. Such analyses go beyond the typical analysis done as part of "stream processing", which restricts the analysis to smaller sets of data (e.g., a five-minute window of data) and does not provide an opportunity for offline ad hoc retrospective processing.

To enable efficient analysis of Big Fast data, it is required for a data management system to allow for continuous ingestion of data such that there is minimal delay introduced in making the data queryable via indexes. In this thesis, we have described the support for continuous data ingestion in AsterixDB, a Big Data Management System (BDMS) that allows storage and analysis of semi-structured data. AsterixDB has a built-in notion of a data feed, which is defined as the flow of data from an external source into indexed storage inside a data management system. The thesis provided a detailed description of the concepts involved in data feed management and efficiently and reliably managing the flow of data. It included a detailed description of the design, implementation, and physical aspects involved in building a scalable, fault-tolerant, and elastic data ingestion facility.

In this work, we have emphasized the need for the genericity and flexibility offered by a data ingestion facility in interfacing with a wide variety of data sources and formats. We have shown how a data feed can be managed by associating an ingestion policy that controls the systems runtime behavior in response to failures and resource bottlenecks. We have included an experimental evaluations that studied the role of different ingestion policies in determining the behavioral aspects of the system, including the achieved, throughput latency, and effectiveness at data capture We also reported experiments to evaluate scalability and our approach to fault-tolerance. The thesis also included a brief evaluation of a "current day" solution created by coupling Storm (a popular streaming engine) and MongoDB (a popular persistent store) to draw a comparison with AsterixDB in terms of flexibility and scalability achieved in data feed management. We demonstrated and described the inefficiencies involved in gluing together such otherwise efficient systems.

Throughput this thesis, we have emphasized the need for "native" (built-in) support for data ingestion in a data management system; a thought that is contrary to the currently accepted practice of 'gluing' together multiple systems as a means to achieve similar result. AsterixDB treats data ingestion as a first class citizen. It abstracts the underlying details and intricacies involved in data flow management and presents only a logical view to the end-user that allows modeling and managing data feed(s) in a declarative way at a language-level. AsterixDB is available as open-source software [2] and all of the work describe here will be released in that form in second quarter of 2015. The support for data ingestion in AsterixDB is extensible to enable future contributors to provide custom implementations of different modules.

9.2 Future Work

We have developed end-to-end support for data ingestion that is scalable and fault-tolerant. In this section, we present some of the natural extensions to our work.

9.2.1 Continuous Queries

The data ingestion facility is AsterixDB in scalable and fault-tolerant and allows ad hoc analysis of persisted (indexed) data via its query language (AQL). AQL supports analytical queries thay may involve aggregation, joining or sorting of data. Analytical queries over data peristed by data feeds enable historical analysis of data as old data is not discarded and thus remains visible for processing by queries. Such a model is in contrast to the stream processing model, where data is considered transient and retained in memory for a short (typically fixed) duration where it can be included as part of the result for a query. The queries in that model are often termed as standing continuous queries that constitute a pre-defined query set.

Data ingestion involves constructing a data flow that allows data from an external source to flow in

a continuous manner prior to being persisted into an indexed storage. The AsterixDB data ingestion framework provides an opportunity for both models – continuous standing queries on moving data and ad hoc queries over persisted indexed data – to co-exist and address diverse analysis allows treating data differently as it moves through the data flow and gets deposited into indexed storage. Such architecture is popularly known as the *Lambda Architecture* [42] and is designed to handle massive quantities of data by taking advantage of both batch- and stream-processing methods. This approach to architecture attempts to balance latency, throughput, and fault-tolerance by providing comprehensive and accurate analysis of persisted data, while simultaneously using real-time stream processing to provide views of online data.

The AsterixDB Query Language (AQL) offers a rich support for ad hoc analytical queries. A natural extension to it would be the support for continuous queries and windowing queries.

9.2.2 Data Replication

AsterixDB does not yet support data replication. Data native to AsterixDB – the target for feeds – is stored in datasets that are horizontally partitioned on the basis of their primary key (hash). Having a single replica of each partition of a dataset adds vulnerability, as a node failure translates to unavailability of data. Data replication is a necessity in typical large deployments which involve commodity hardware that is prone to failures. Data replication of course is faced with the challenge of synchronizing the replicas to present a consistent view of data. Solutions to replication come in two variants – synchronous replication and asynchronous replication. The former strives to keep replicas in sync at all times by having each write request to be propagated to all replicas and marking an operation as complete when all replicas have successfully recorded the change. In contrast, asynchronous replication completely decouples the replicas, and as each is allowed to be out-of-sync and process write requests at their own pace without impacting other replicas. Synchronous replication favors consistency over write-latency, whereas asynchronous replication offers a loose

definition of consistency (aka eventual consistency) and optimizes for write-throughput.

The challenges faced when replicating data (synchronously or asynchronously) are aggravated in an environment that involves continuously arriving high-velocity data. A high rate of arrival of data requires highly optimized writes. Asynchronous replication is expected to offer a higher write-throughput when compared to synchronous replication. However substituting a lost dataset partition by an in-sync replica becomes a challenge as replicas are allowed to diverge during normal operation. As part of future work, we wish to support data replication and address the aforementioned challenges that surface when ingesting high-velocity data into AsterixDB datasets via data feeds. Also of interest are the opportunities that replication and its infrastructure might offer for extending the elasticity of data feeds into the storage layer.

Bibliography

[1] "Apache Samza," http://samza.apache.org/.

[2] Asterixdb http://asterix.ics.uci.edu.

[3] "AsterixDB source," https://code.google.com/p/asterixdb.

[4] "Data on Big Data," http://marciaconner.com/blog/data-on-big-data/.

[5] Hadoop http://hadoop.apache.org/.

[6] "Informatica PowerCenter" http://www.informatica.com/in/etl/.

[7] "MongoDB," http://www.mongodb.org/.

[8] "Twitter's Storm," http://storm-project.net.

[9] "XQuery," http://www.w3.org/tr/xquery/.

[10] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, et al. Aurora: A Data Stream Management System. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 666–666. ACM, 2003.

[11] H. Agrawal, G. Chafle, S. Goyal, S. Mittal, and S. Mukherjea. An enhanced extract-transform-load system for migrating data in telecom billing. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 1277–1286. IEEE, 2008.

[12] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, 2002.

[13] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)*, 33(1):3, 2008.

[14] K. Balog and H. Ramampiaro. Cumulative Citation Recommendation: Classification vs. Ranking. *Proceedings of ACM SIGIR)*, 2013.

[15] K. Balog, N. Takhirov, H. Ramampiaro, and K. Norvaag. Multi-step classification approaches to cumulative citation recommendation. *Open Research Areas in Information Retrieval*, 2013.

[16] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[17] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Mobile Data Management*, pages 3–14. Springer, 2001.

[18] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1151–1162. IEEE, 2011.

[19] J. Boulon, A. Konwinski, R. Qi, A. Rabkin, E. Yang, and M. Yang. Chukwa, a large-scale monitoring system. In *Proceedings of CCA*, volume 8, pages 1–5, 2008.

[20] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams: A New Class of Data Management Applications. *Proceedings of the 28th international conference on Very Large Data Bases*, pages 215–226, 2002.

[21] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[22] C. L. Compton and D. L. Tennenhouse. Collaborative Load Shedding for Media-Based Applications. *Multimedia Computing and Systems, 1994., Proceedings of the International Conference on*, pages 496–501, 1994.

[23] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[24] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134. ACM, 2008.

[25] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *Parallel and Distributed Systems, IEEE Transactions on*, 23(12):2351–2365, 2012.

[26] D. C. Hansen. Fast data: Go big. go fast. *Oracle Blogs*, oct 2012.

[27] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 779–790. IEEE, 2005.

[28] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[29] A. Jain. *Instant Apache Sqoop*. Packt Publishing Ltd, 2013.

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

[31] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system, 2002.

[32] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[33] P. E. O'neil, E. Cheng, D. Gawlick, and E. J. O'neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33, 1996.

[34] O. Rafal. From big data to fast data. *Information Management*, Aug 2013.

[35] S. Schneider et al. Elastic scaling of data parallel operators in stream processing. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009.

[36] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 827–838. ACM, 2004.

[37] M. Stonebraker. Operating system support for database management. *Commun. ACM*, 24, 1981.

[38] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 309–320, 2003.

[39] P. Vassiliadis. A survey of extract–transform–load technology. *International Journal of Data Warehousing and Mining (IJDWM)*, 5(3):1–27, 2009.

[40] Wikipedia. Extract, transform, load — wikipedia, the free encyclopedia, 2014. [Online; accessed 12-November-2014].

[41] Wikipedia. Separation of concerns — wikipedia, the free encyclopedia, 2014. [Online; accessed 22-January-2015].

[42] Wikipedia. Lambda architecture — wikipedia, the free encyclopedia, 2015. [Online; accessed 24-February-2015].

[43] M. Wojtasiak. Intelligent infrastructure. *Big Data*, April 2014.

[44] Y. Xu, P. Kostamaa, Y. Qi, J. Wen, and K. K. Zhao. A hadoop based distributed loading approach to parallel data warehouses. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1091–1100. ACM, 2011.

[45] C.-Q. Yang and A. V. Reddy. A Taxonomy for Congestion Control Algorithms in Packet Switching Networks. *Network, IEEE*, 9(4):34–45, 1995.

Appendices

A Feed Management Console

AsterixDB provides a *Feed Management Console* that enlists the set of connected feeds in AsterixDB. For each listed field, the console shows information that includes the start timestamp, target dataset, set of nodes participating at each stage (intake, compute, store) of the data ingestion pipeline. In addition the aggregate rate of inflow of data records at the intake stage (referred hereafter as *intake rate*) and the rate of persistence of records at the store stage (referred hereafter as *store rate*), each measured in terms of records/sec. Figure A.1 shows a screen shot of the Feed Management Console that lists a pair of feeds in the connected state.

A part of the information shown per connected feed is static and includes the name of the feed and the target dataset. The scheduling information that shows the location for each operator instance may change over time if the parts of the data ingestion pipeline are re-structured (to resolve congestion) or re-located (to resolve hard failures). Other dynamic information includes the intake rate and store rate. Each operator at the intake and the store stage reports these metrics to the local Feed Manager, which subsequently forwards them to the Central Feed Manager, where these are aggregated.

The Feed Management Console simply provides a read-only view of the connected feeds. We are currently working on enabling the end-user to manage lifecycle for a feed using the console.

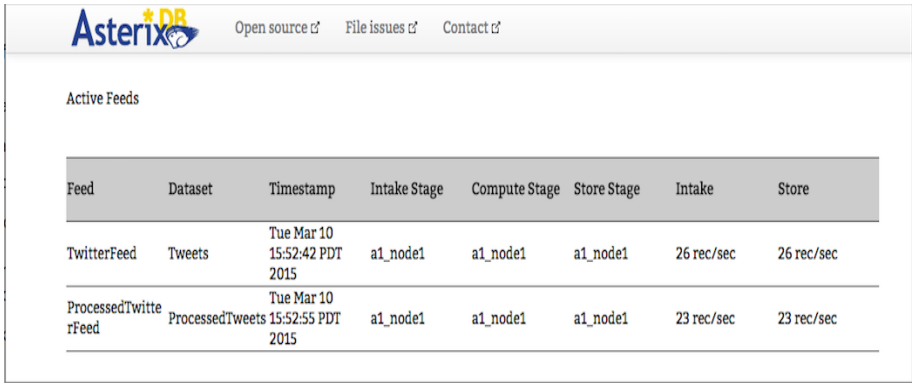


Figure A.1: A screen shot of the Feed Management Console that shows a pair of connected feeds. The primary TwitterFeed retrieved records from Twitter using the streaming API which offers a low rate of arrival of data. For each feed, the physical nodes participating at the intake compute and store stages are shown. The respective instantaneous rates at which data is received and persisted are also shown.

B Writing a Custom Adaptor

Feed adaptors in AsterixDB are pluggable components that can be custom built by end-users to cater to their specific needs. In this section, we illustrate how an end-user can write a custom adaptor, package it as being part of an AsterixDB library and install it with an AsterixDB instance. The basic API implemented by each adaptor is shown in Listing B.1. The API is low-level and requires understanding of the binary format used within AsterixDB to represent a data record. To abstract away the low-level details and ease the task of writing a custom adaptor, AsterixDB provides default implementation of the API that only requires the end-user to implement high-level APIs that do not require significant knowledge of the internals of AsterixDB. We discuss this abstraction next in context of a push and pull based feed adaptor. Recall that we classify a feed adaptor as being *push* or *pull* based depending upon how it communicates with the data source for transfer of data.

```
/**
 * Triggers the adapter to begin ingesting data from the external source.
 *
 * @param partition
 *            The adapter could be running with a degree of parallelism.
 *            partition corresponds to the i'th parallel instance.
 * @param writer
 *            The instance of frame writer that is used by the adapter to
 *            write frame to. Adapter packs the fetched bytes (from
 *            external source), packs them into frames and forwards
 *            the frames to an upstream receiving operator using the
 *            instance of IFrameWriter.
 * @throws Exception
 */
public void start(int partition, IFrameWriter writer) throws Exception;
```

Listing B.1: Basic API required to be implemented by a Feed Adaptor

B.1 Push-Based Adaptor

A push-based adaptor makes an initial request to the data source to establish a connection and provide any configuration parameters for the connection. Thereafter, the data source begins to send data to the adaptor instance without requiring any additional requests. Data from a push-based data source is typically made available via a channel (an instance of *InputStream*) that provides access methods to retrieve the next byte. The sequence of bytes delivered on the InputStream needs to be parsed to construct the equivalent representation of data records in ADM format. AsterixDB provides a collection of built-in data parsers for popular formats that include delimited-text (e.g., CSV), ADM and JSON. If the data source uses a different format, AsterixDB allows end-users to use a custom parser implementation. The format for data and the parser implementation (in case when specified format is not natively supported by AsterixDB) are specified as configuration parameters alongside the feed adaptor as part of the *create feed* AQL statement.

To cater to this common case where data is received on an instance of InputStream, AsterixDB provides an abstract class – *StreamBasedAdaptor* that hides away much of the complexity involved

```
public class MyPushBasedAdaptor extends StreamBasedAdaptor {

    @Override
    public InputStream getInputStream(int partition ) throws IOException {
     InputStream in = /*code to obtain input stream from data source*/;
       return inputStream;
    }

}
```

Listing B.2: An example implementation of a push-based feed adaptor that extends the built in StreamBasedAdaptor

in retrieving and parsing bytes from the stream and translating them to ADM formatted records. Listing B.2 shows a custom adaptor that extends *StreamBasedAdaptor* and is required to override the method *getInputStream()*. The base class *StreamBasedAdaptor* takes care of constructing ADM records, packing them as data frames and sending the frames downstream.

B.2 Pull-Based Adaptor

A pull-based adaptor works by polling the data source for additional data and thus involves repeated request messages exchanged between the adaptor and the source. Similar to the case of a push-based adaptor, AsterixDB provides much of the functionality contained in the abstract class -*PullBasedFeedAdapter*. In pull-based data ingestion, AsterixDB as a receiver needs to initiate a request it wishes to receive data. As AsterixDB remains agnostic of the intricacies involved in interfacing with the external source, it relies on a *IFeedClient* implementation that contains the required functionality of creating a request and receiving data as a response from the external source. Note that the IFeedClient implementation is specific to the data source. Listing B.3 shows the IFeedClient interface. End-users may simply write a custom implementation for IFeedClient interface and use it in a custom adaptor. Listing B.4 shows an example pull-based adaptor that uses a custom implmentation (Listing B.3) of IFeedClient.

```
public interface IFeedClient {

    public enum InflowState {
    /** all data from the data source has been received **/
        NO_MORE_DATA,

        /** new data was received from the data source **/
        DATA_AVAILABLE,

        /** no more data is expected from data source; end of feed **/
        DATA_NOT_AVAILABLE
    }

    /**
     * Writes the next fetched tuple into the provided instance of DatatOutput.
     * Invocation of this method blocks until new tuple has been written or
     * the specified time has expired.
     *
     * @param dataOutput
     *              the output channel used by a feed client to write ADM
     *              records.
     * @return InflowState
     *          the state of the data source
     * @throws AsterixException
     */
    public InflowState nextTuple(DataOutput dataOutput) throws AsterixException;

}
```

Listing B.3: The IFeedClient interface

```
public class MyPullBasedAdaptor extends PullBasedAdaptor {

@Override
  public IFeedClient getFeedClient(int partition ) throws Exception{
    return new MyFeedClient(..);
  }
}
```

Listing B.4: An example implementation of a pull-based adaptor that extends the built-in PullBasedAdaptor and provides the required implmentation of the abstract method - getFeedClient.

```
public class MyFeedClient implements IFeedClient {

    public InflowState nextTuple(DataOutput dataOutput, int timeout) throws AsterixException
    {
    /** construct the next tuple (ADM Record) and write its
        serialized form to the DataOutput instance
     **/
     return /* instance of Inflow state as appropriate */
    }

}
```

Listing B.5: An example implementation of the IFeedClient interface (Listing B.3)

B.3 AdaptorFactory

Following the *factory* design pattern, an adaptor has an associated *AdaptorFactory* that allows creation and configuration of an adaptor instance. The factory is an implementation of the *IAdaptorFactory* interface and contains *getters* that provide AsterixDB with the following information.

- **Alias or Name of the Adaptor**: This is the unique name for the adaptor by which it will be referred to in AQL statements.
- **Count/Location Constraints**: The number of parallel instance (count constraint) and any specific set of locations (AsterixDB nodes) where the adaptor instances are required to run.
- **Output Datatype**: The datatype that is representative of the data received from the data source.

In addition, an adaptor factory also provides the *configure* method that allows AsterixDB to initialize an adaptor instance with the set of configuration parameters, passed as part of the *create feed* AQL statement.

C Writing an External Java Function

AsterixDB allows pluuging in an external Java function for its use within AQL statements or queries. In this section, we provide an overview of the support for Java UDFs and illustrate how one may define a custom UDF with an example.

A Java UDF has a lifecycle and is required to implement the IExternalScalarFunction interface. The specific methods required to be implemented are *initialize()*, *evaluate()* and *deinitialize()* that allow AsterixDB to manage the lifecycle of a Java UDF and invoke it with the required set of arguments.

- *initialize()*: The *initialize()* method is called only once prior to any invocation of the UDF by AsterixDB. This method allows the UDF to perform any sort of initialization that is required before it becomes usable for invocations.
- *evaluate()*: The *evaluate()* function contains the actual computation that the function performs on each input record.
- *deinitialize()* The *deinitialize()* method is invoked as part of cleanup or tear down where any resources (e.g. file handle(s), connection(s), etc.) used by the function are relinquished.

The lifecycle of a Java UDF consists of an invocation of the *initialize()* method followed by one or more invocations of the *evaluate()* method and eventually the invocation of the *deinitialize()* method. As described before, a Java UDF forms a pluggable component that is installed with an AsterixDB instance by packaging into an AsterixDB library. Relevant information about the UDF including the type associated with its arguments and return value is captured as an entry in the descriptor XML that is associated with the library. Listing D.14 shows an example UDF implemented in Java.

The relevant information about the UDF as included in the library descriptor XML is shown in

Listing D.7. As defined in the descriptor, the function operates on an input record of type Tweet and returns as output, a record of similar type. The function collects the hash tags contained in the *message-text* attribute of the input tweet and collects them into an ordered list that is appended as an additional attribute to the tweet. Note that the data in AsterixDB is represented using the AsterixDB Data Model (ADM) which is different from the data model followed in Java. AsterixDB provides a binding that includes a one-to-one mapping between ADM data types (records, lists, and primitive types such as int, long, string etc.) and their equivalent representative types that are meant to be used when implementing an external UDF. The binding ensures backward compatibility wherein existing UDFs need not be recompiled to work with future versions of AsterixDB that may bring changes to the AsterixDB data model.

D Installing a Pluggable - Adaptor/Function

Pluggable components such as a user-defined feed adaptor or a java function need to be packaged as an AsterixDB library and installed with an AsterixDB instance before these can be used in AQL statements. This section describes the steps for packaging and installation of an AsterixDB libary. An AsterixDB library has a pre-defined structure which has the following components.

- **Jar File**: The source code for the user-defined function or feed adaptor is compiled into class files that need to packaged as jar file.
- **Library Descriptor** An AsterixDB library contains a descriptor document (XML file) that contains essential information about the contents of the library, that is the set of functions and/or feed adaptors that will be installed. As part of installation, relevant information about each installed artifact is extracted and put into the AsterixDB Metadata. Listing D.7 shows an example library descriptor that describes a pair of library function (hashTags) and a feed adaptor (MyAdaptor) as the components that need to be installed.

```
package edu.uci.ics.asterix.external.library;

import edu.uci.ics.asterix.external.library.java.JObjects.JRecord;
import edu.uci.ics.asterix.external.library.java.JObjects.JString;
import edu.uci.ics.asterix.external.library.java.JObjects.JUnorderedList;
import edu.uci.ics.asterix.external.library.java.JTypeTag;

public class HashTagsFunction implements IExternalScalarFunction {

    private JOrderedList list = null;

    @Override
    public void initialize(IFunctionHelper functionHelper) {
        list = new JOrderedList(functionHelper.getObject(JTypeTag.STRING));
    }

    @Override
    public void evaluate(IFunctionHelper functionHelper) throws Exception {
        JRecord inputRecord = (JRecord) functionHelper.getArgument(0);
        JString text = (JString) inputRecord.getValueByName("message-text");

        String[] tokens = text.getValue().split(" ");
        for (String tk : tokens) {
            if (tk.startsWith("#")) {
                JString newField = (JString) functionHelper.
                                        getObject(JTypeTag.STRING);
                newField.setValue(tk);
                list.add(newField);
            }
        }
        inputRecord.addField("topics", list);
        functionHelper.setResult(inputRecord);
    }

    @Override
    public void deinitialize() {
    }

}
```

Listing C.6: An example Java UDF

```
<externalLibrary xmlns="library">
  <libraryFunctions>
    <libraryFunction>
      <language>JAVA</language>
      <function_type>SCALAR</function_type>
      <name>hashTags</name>
      <arguments>Tweet</arguments>
      <return_type>ProcessedTweet</return_type>
      <definition>edu.uci.ics.asterix.external.udf.HashTagsFunctionFactory
      </definition>
    </libraryFunction>
  </libraryFunctions>
  <libraryAdapters>
    <libraryAdapter>
      <name>tweetgen_adaptor</name>
      <factory_class>edu.uci.ics.asterix.external.library.adaptor.TweetGenAdaptorFactory
  </factory_class>
    </libraryAdapter>
  </libraryAdapters>
</externalLibrary>
```

Listing D.7: An example library descriptor XML. The library contains a Java UDF and a feed adaptor as the components that get installed as part of the library.

- **Dependency Jars** If the user-defined component requires additional dependency jars, these are added a "lib" directory and are made available at runtime.

A library is a zip archive with contents as described above. Listing D.8 shows the contents of an example library (tweetlib.zip) that contains a Java UDF and a feed adaptor as the components that need to be installed. The library contains the class files packaged as *twitter_components*.jar and a library descriptor xml (tweet.xml) which is identical to the XML shown in Listing D.7.

Next, we describe the steps involved in installing a library archive (zip bundle) with an AsterixDB instance. We assume here that an AsterixDB instance by the name "my_asterix" exists. A detailed description on how to set up an AsterixDB instance is beyond the sceope of this document, but the reader may find the instructions at **Managix - Creating and Managing an AsterixDB instance**. AsterixDB offers a management tool – *Managix* that allows end-user to create an AsterixDB in-

```
$ unzip -l ./tweetlib.zip
Archive:  ./tweetlib.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
   760817  04-23-14 17:16   twitter_components.jar
      405  04-23-14 17:16   tweet.xml
---------                     -------
   761222                     2 files
```

Listing D.8: Contents of an external library zip bundle as listed using the unzip command

stance, manage its lifecycle and perform management operations that includes (un)installation of AsterixDB library. The steps to install a library are as follows.

- **Step 1**: Stop the AsterixDB instance if it is in the ACTIVE state.

```
$ managix stop -n my_asterix
```

Listing D.9: Stopping an AsterixDB instance using the managix stop command

- **Step 2**: Install the library using Managix install command.

  For illustration purpose, we use the help command to look up the syntax.

```
$ managix help -cmd install
 Installs a library to an asterix instance.
Arguments/Options
-n Name of Asterix Instance
-d Name of the dataverse under which the library will be installed
-l Name of the library
-p Path to library zip bundle
```

Listing D.10: Using the managix help command to output the usage of any managix comman. In the current listing we use the install command as an example.

Above is a sample output and explains the usage and the required parameters. Each library has a name and is installed under a dataverse. We provide a name for our library - ”tweetlib”,

but ofcourse, you may choose another name. To install the library, use the Managix install command. An example is shown below.

```
$ managix install -n my_asterix -d feeds -l tweetlib -p <put the absolute path of the
    library zip bundle here>
```

Listing D.11: Using the managix install command to install an external library into an AsterixDB instance

You should see the following message:

```
INFO: Installed library tweetlib
```

Listing D.12: Example output obtained on using the managix install command to install a library

We shall next start our AsterixDB instance using the start command as shown below.

```
$ managix start -n my_asterix
```

Listing D.13: Starting an AsterixDB instance using the managix start command

You may now use the AsterixDB library in AQL statements and queries. To look at the installed artifacts, you may execute the following query at the AsterixDB web-inerface.

```
for $x in dataset Metadata.Function
return $x

for $x in dataset Metadata.Library
return $x
```

Listing D.14: A set of queries to extract the metadata information on the installed functions and libraries.