Data Ingestion in AsterixDB

Raman Grover, Michael J. Carey

Department of Computer Science,
University of California- Irvine, CA, 92697
{ ramang, mjcarey}@ics.uci.edu

ABSTRACT

In this paper we describe the support for 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 new mechanism for having continuous data arrive into a BDMS from external sources and incrementally populate a persisted dataset and associated indexes. We add a new BDMS architectural component, called a *data feed*, that makes a Big Data system the caretaker for functionality that used to live outside, and we show how it improves users' lives and system performance.

We show how to build the *data feed* component, architecturally, and how an enhanced user model can enable sharing of ingested data. We describe how to make this component fault-tolerant so the system manages input in the presence of failures. We also show how to make this component elastic so that variances in incoming data rates can be handled gracefully without data loss if/when desired. Results from initial experiments that evaluate scalability and fault-tolerance of AsterixDB data feeds facility are reported. We include an evaluation of 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.

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 or via sharing on popular social websites [3]. Encouraged by low storage costs, enterprises today are aiming to collect and persist the available data and analyze it over time to extract useful information. Marketing departments use Twitter feeds to conduct sentiment analysis to get end user feedback on their company's products. As another example, utility companies have rolled out meters that measure the consumption of water, gas, and electricity and generate huge volumes of interval data that is analyzed over time. Traditional data management systems require data to be loaded and indexes to be created before data can be subjected to ad hoc analytical queries. To keep pace with "fast-moving" 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 (process) fetch data from an external data source, parse the data, and then invoke an insert statement per record or batch of records. This solution is limited to a single machine's computing capacity. Ingesting multiple data feeds would potentially require running and managing many individual programs/processes. The task of continuously retrieving data from external source(s), applying some pre-processing for cleansing, filtering data, and indexing the processed data today amounts to 'gluing' together different systems (e.g. [19]). It becomes hard to reason about the data consistency, scalability and fault-tolerance offered by such an assembly. Traditional data management systems have evolved over time to provide native support for services if the service offered by an external system is inappropriate or may cause substantial overheads [18, 10]. Responding to the new need then, it is natural for a BDMS to provide "native" support for data feed management.

1.1 Challenges in Data Feed Management

Let us begin by enumerating the key challenges involved in building a data feed facility.

C1) *Genericity* and *Extensibility*: A feed ingestion facility must be generic enough to work with a variety of data sources and high-level applications. A plug-and-play model is desired to allow extension of the offered functionality.

C2) *Fetch-Once, Compute-Many*: Multiple applications may wish to consume the ingested data and may wish the arriving data to be processed/persisted differently. It is desirable to receive a single flow of data from an external source and yet transform it in multiple ways to drive different applications concurrently.

C3) *Scalability and Elasticity*: Multiple feeds with fluctuating data arrival rates, coupled with ad hoc queries over the persisted data, imply a varying demand for resources. The system should offer *scalability* by being able to ingest increasingly large volumes of data (possibly from multiple sources) via the addition of resources. The system should demonstrate *elasticity* by auto-scaling in/out to meet the demand for resources.

C4) *Fault Tolerance*: Data ingestion is expected to run on a large cluster of commodity hardware that may be prone to hardware failures. It is desirable to offer the desired degree of robustness in handling failures while minimizing data loss.

10.5441/002/edbt.2015.61

1.2 Contributions

In this paper, we describe the support for data feed management in AsterixDB. AsterixDB provides a platform for the scalable storage and analysis of very large volumes of semi-structured data. The paper describes the approach adopted to address the aforementioned challenges. This paper also demonstrates the efficiency/flexibility achieved in having native support for feed ingestion in AsterixDB in comparison to the popular approach of 'gluing' together popular systems (e.g. Storm[6] and MongoDB[5]), which is the state of the art today. The paper offers the following contributions.

(1) *Concepts involved in Data Feed Management*: The paper 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*: We describe how a data feed is managed by associating an ingestion policy that controls the system's runtime behavior in response to failures and resource bottlenecks. Users may also opt to provide a custom policy to suit special application requirements.

(3) *Scalable/Elastic Data Feed Management*: We describe 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*: AsterixDB is available as open source software [2, 1]. The support for data ingestion in AsterixDB is extensible to enable future contributors to provide custom implementations of different modules.

(5) *Experimental Evaluation*: We describe 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 to evaluate scalability and our approach to fault-tolerance.

(6) *Improvement over State-of-the-Art*: We include an evaluation of a system created by coupling Storm (a popular streaming engine) and MongoDB (a popular persistence store) to draw a comparison with AsterixDB in terms of flexibility and scalability achieved in data feed management. We demonstrate and describe the inefficiencies involved in 'gluing' together such otherwise efficient systems; doing so is a current common practice in open-source community.

The rest of the paper is organized as follows. We discuss related work in Section 2 and provide an overview of AsterixDB in Section 3. Section 4 describes how a feed is modeled and defined at the language level in AsterixDB. The implementation details are described in Section 5. Section 6 describes the support for handling failures. Section 7 provides an experimental evaluation, and we conclude in Section 8.

2. RELATED WORK

Data feeds may seem similar to streams from the data streams literature (e.g. [7, 13]). There are important differences, however. Data feeds are a "plumbing" concept; they are a mechanism for having data flow from external sources that produce data continuously to incrementally populate and persist the data in a data store. Stream Processing Engines (SPEs) do not persist data; instead they operate with 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. In a similar spirit, Complex Event Processing (CEP) systems (Storm [6] and S4 [15]) can route, transform and analyze a stream of data. However, these systems do not persist the data or provide support for ad hoc analytical queries. These engines can be used in conjunction with a database (e.g MySql or MongoDB), making it possible to persist and run ad hoc queries.

In the past, ETL (Extract Transform Load) systems (e.g. [4]) have supported populating a Data Warehouse with data collected from multiple data sources. However, such systems operate in a "batchy" mode, with a "finite" amount of data transferred at periodic intervals coinciding with off-peak hours. Xu et al. in [19] described a Map-Reduce based approach for populating a parallel database system with an external feed. However, the system was tightly-coupled with Map-Reduce and required data to be put into HDFS, thus involving an additional copy.

With respect to providing fault-tolerance, stream processing systems also faced the challenge of providing highly available parallel data-flows and have proposed several techniques [17, 8]. The process-pairs approach, used in Flux and StreamBase, involves a high overhead when the system needs to scale. These techniques rely on replication; the state of an operator is replicated on multiple servers or have multiple servers simultaneously process identical input streams. Fault-tolerance is provided at a high cost, as the number of nodes is thus at least doubled due to replication. It was thus not considered for use in AsterixDB. Moreover, offering a single strong strategy for fault-tolerance can be wasteful of resources in scenarios where the offered degree of robustness exceeds the application's requirements.

Data ingestion and stream-processing data-flows are typically associated with fluctuating data arrival rates that cause a varying demand for resources. An elastic behavior with the ability to scale in/out in adaptation to the demand for resources is desirable. Such mechanisms have been studied and evaluated before. Elastic reconfiguration in [16] is triggered when the data arrival rate exceeds a pre-calculated saturation rate by some fraction (e.g., 5% in their papers). It is not clear how an appropriate saturation rate would be statically calculated in a dynamic environment with concurrent feeds and queries over the ingested data. AsterixDB follows a different approach by dynamically monitoring the rate of flow of data and the availability of resources across the participant nodes. This allows detecting resource bottlenecks and triggering corrective actions in accordance with measured values.

We have explored the challenges involved in building a data ingestion facility. In doing so, we added a new BDMS architectural component, called a *data feed*, that makes a Big Data system the caretaker for functionality that used to live outside, and we show how it improves users' lives and system performance. In this paper, we describe how to build this component, architecturally, so that it provides continuous load-like performance (i.e., low overhead) - and how an enhanced user model can enable sharing of ingested data. We identify a number of different QoS options that users might want, depending on the nature of their application, and we show how to deliver them via dynamic monitoring of the system state. We also show how to make this new *data feed* component fault-tolerant so the system manages input (and the user doesn't have to) in the presence of failures. We show how to make this component elastic so that variances in incoming data rates can be handled gracefully without data loss if/when desired. We added that functionality to AsterixDB, and we demonstrate that it works (and how well).

3. BACKGROUND: 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. It combines the ideas from three distinct areas—semi-structured data, parallel databases, and data-intensive computing—in order to create an open-source software platform that scales by running on large, shared-nothing commodity computing clusters.

3.1 AsterixDB Architecture

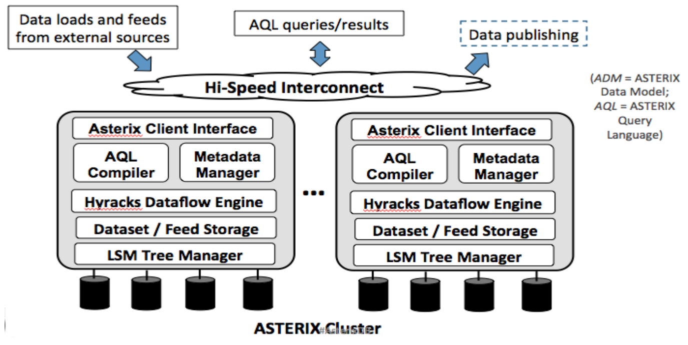


Figure 1: AsterixDB Architecture

Figure 1 provides an overview of how the various software components of AsterixDB map to nodes in a shared-nothing cluster. The topmost layer of AsterixDB is a parallel DBMS, with a full, flexible AsterixDB Data Model (ADM) and AsterixDB 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 Figure 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. A detailed description of AsterixDB and results from experimental evaluation can be found in [12].

AsterixDB uses Hyracks [11] 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.

3.2 AsterixDB Data Model

AsterixDB defines its own data model (ADM) [9] designed to support semi-structured data with support for bags/lists and nested types. Figure 2 shows how ADM can be used to define a record type for modeling a raw tweet. *RawTweet* type is an open type, meaning that its instances will conform to its specification but can contain extra fields that vary per instance. Figure 2 also defines a *ProcessedTweet* type. A processed tweet replaces the nested user field inside a raw tweet with a primitive userId value and adds a nested collection of strings (referred topics) to each tweet. Derived attributes about the tweet (e.g. sentiment and language) are also included. The primitive location field types (location-lat, location-long) and send-time are expressed as their respective spatial (point) and temporal (datetime) datatypes. ADM also allows specifying optional fields with known types (e.g. location).

Data in AsterixDB is stored in *datasets*. Each record in a dataset conforms to the datatype associated with the dataset. Data is hash-partitioned (primary key) across a set of nodes that form the *nodegroup* for a dataset, which defaults to all nodes in an AsterixDB cluster. Figure 3 shows the AQL statements for creating a pair of datasets—*RawTweets* 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.

```
create type RawTweet                 create type TwitterUser
as open {                            as open {
   tweetId: string,                     screen-name: string,
   user: TwitterUser,                   lang: string,
   location-lat: double?,               friends_count: int32,
   location-long: double?,              statuses_count: int32,
   send-time: string,                   name: string,
   message-text: string                 followers_count: int32
};                                   };

create type ProcessedTweet as open {
   tweetId: string,
   userId: string,
   location: point?,
   send-time: datetime,
   message-text: string,
   referred-topics: {{string}},
   sentiment: double,
   language: string
};
```

Figure 2: Defining datatypes

```
create dataset RawTweets(RawTweet) primary key tweetId;

create dataset ProcessedTweets(ProcessedTweet)
primary key tweetId;

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

Figure 3: Creating datasets and associated indexes

4. DATA FEED BASICS

AQL has built-in support for data feeds. In this section, we describe how an end-user may model a data feed and have its data be persisted/indexed into an AsterixDB dataset.

4.1 Collecting Data: Feed Adaptors

The functionality of establishing a connection with a data source and receiving, parsing and translating its data into ADM records (for storage inside AsterixDB) is contained in a *feed adaptor*. A feed adaptor is an implementation of an interface and its details are specific to a given data source. An adaptor may optionally be given parameters to configure its runtime behavior. Depending upon the data transfer protocol/APIs offered by the data source, a feed adaptor may operate in a *push* or a *pull* mode. Push mode involves just one initial request by the adaptor to the data source for setting up the connection. Once a connection is authorized, the data source "pushes" data to the adaptor without any subsequent requests by the adaptor. In contrast, when operating in a pull mode, the adaptor makes a separate request each time to receive data.

AsterixDB currently provides built-in adaptors for several popular data sources—Twitter, CNN, and RSS feeds. AsterixDB additionally provides a generic socket-based adaptor that can be used to ingest data that is directed at a prescribed socket. Figure 4 illustrates the use of built-in adaptors in AsterixDB to define a pair of feeds. The *TwitterFeed* contains tweets that contain the word "Obama". As configured, the adaptor will make a request for data every minute. The *CNNFeed* will consist of news articles that are related to any of the topics that are specified as part of the indicated configuration.

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

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

Figure 4: Defining a feed using some of the built-in adaptors in AsterixDB

The degree of parallelism in receiving data from an external source is determined by the feed adaptor in accordance with the data exchange protocol offered by the data source. The external source may allow transfer of data in parallel across multiple channels. For example, CNN as a data source offers an RSS feed corresponding to each topic (politics, sports, etc). The CNNFeed can thus employ a degree of parallelism as determined by the number of topics that are passed as configuration. Multiple instances will then run as parallel threads on a single machine or on multiple machines. In contrast, the TwitterAdaptor uses a single degree of parallelism.

4.2 Pre-Processing Collected Data

A feed definition may optionally include the specification of a user-defined function that is to be applied to each feed record prior to persistence. Examples of pre-processing might include adding attributes, filtering out records, sampling, sentiment analysis, feature extraction, etc. 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 requires the result of a query (join or aggregate) over data contained in AsterixDB datasets. 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 is assumed by the AsterixDB compiler to be stateless and thus usable as an embarassingly parallel black box. In constrast, the AsterixDB compiler can reason about an AQL UDF and involve the use of indexes during its invocation.

The tweets collected by the TwitterAdaptor (Figure 4) conform to the *RawTweet* datatype (Figure 2). The processing required in transforming a collected tweet to its lighter version (of type *ProcessedTweet*) involves extracting the hash tags (if any) in a tweet and collecting them in the referred-topics attribute for the tweet. Attributes associated with a tweet (sentiment, language) are derived from analyzing the text. In the case of the *CNNFeed*, the CNNAdaptor (Figure 4) 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. This information can then be added to each record as additional fields prior to persistence. The pre-processing function for a feed is specified using the *apply function* clause at the time of creating the feed (Figure 5).

A feed adaptor and a UDF act as *pluggable* components. These contribute towards providing a generic ‘plug-and-play‘ model where custom implementations can be provided to cater to specific requirements. This helps address challenge C1 from Section 1.1. By providing implementation of the prescribed interfaces, the internal

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

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

Figure 5: Defining a feed that involves pre-processing of collected data

details of data feed management are abstracted from end users. A feed adaptor or a Java UDF can be packaged and installed as part of an AsterixDB library and subsequently be used in AQL statements. A tutorial on building a custom feed adaptor or a UDF with a description of the interfaces to be implement can be found at [1].

4.3 Building a Cascade Network of Feeds

Multiple high-level applications may wish to consume the data ingested from a data feed. Each such application might perceive the feed in a different way and require the arriving data to be processed and/or persisted differently. Building a separate flow of data from the external source for each application is 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, we introduce the notion of *primary* and *secondary* feeds in AsterixDB to address challenge C2 from Section 1.1.

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 made to derive other secondary feeds to form a *cascade network*. A primary feed and a dependent secondary feed form a hierarchy. As an example, Figure 6 shows the AQL statements that redefine the previous feeds—*ProcessedTwitterFeed* and *ProcessedCNNFeed*—in terms of their respective parent feeds from Figure 4.

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

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

Figure 6: Defining a secondary feed

4.4 Lifecycle of a Feed

A feed is a *logical* artifact that 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 (Figure 7). 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 contents of the dataset represent the union of the connected feeds. In a supported but unlikely scenario, one feed may also be simultaneously connected to different target 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 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/interrupting the flow of data along a connected ancestor feed.

```
connect feed ProcessedTwitterFeed to
dataset ProcessedTweets;

disconnect feed ProcessedTwitterFeed from
dataset ProcessedTweets;
```

Figure 7: Managing the lifecycle of a feed

The *connect feed* statement in Figure 7 directs AsterixDB to persist the *ProcessedTwitterFeed* feed in the *ProcessedTweets* dataset. If it is required (by the high-level application) to also retain the raw tweets obtained from Twitter, the end user may additionally choose to connect *TwitterFeed* to a (different) dataset. Having a set of primary and secondary feeds offers the flexibility to do so. Let us assume that the application needs to persist *TwitterFeed* and that, to do so, the end user makes another 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 8. The flow of data from a feed into a dataset can be terminated explicitly by use of the *disconnect feed* statement (Figure 7). Disconnecting a feed from a particular dataset does not interrupt the flow of data from the feed to any other dataset(s), nor does it impact other connected feeds in the lineage.

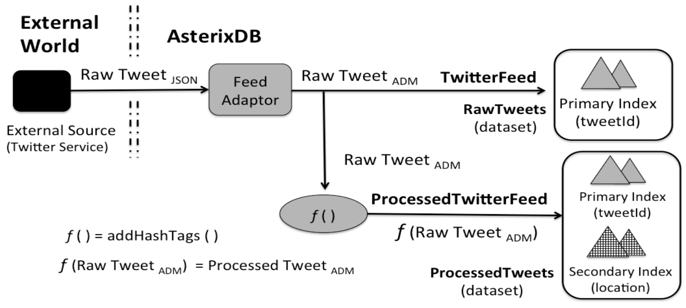


Figure 8: Logical view of the flow of data from external data source into AsterixDB datasets

4.5 Policies for Feed Ingestion

Multiple feeds may be concurrently operational on an AsterixDB cluster, each competing for resources (CPU cycles, network bandwidth, 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 1) that help 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 1.

The handling of excess records by the built-in ingestion policies of AsterixDB is summarized in Table 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 will discuss the built-in policies to Section 5.3, where we cover the physical aspects and their implementation details.

Note that the 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 all cases, the desired ingestion policy is specified as part of the connect feed statement (Figure 9) or else the 'Basic' policy will be chosen as the default. It is worth noting that a feed can be connected to a dataset at any time, which is independent from other related feeds in the hierarchy. As such the *connect feed* statements shown in Figure 9 are not required to be executed together.

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.

```
connect feed TwitterFeed to dataset RawTweets
using policy Basic;

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

Figure 9: Specifying the ingestion policy for a feed

5. RUNTIME FOR FEED 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 define a feed,

Table 1: A Few Important Policy Parameters

Policy Parameter	Description	Default
excess.records.spill	Set to true if records that cannot be processed by an operator for lack 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 failures (loss of AsterixDB node(s)). A false value permits the early termination of a feed in the event of a hardware failure.	true

manage its lifecycle, and dictate its runtime behavior by selecting a policy. Next, 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.

5.1 Runtime Components

In processing a connect feed statement, the AQL compiler retrieves the definitions of the involved components—feed, adaptor, function, policy, and the target dataset from the AsterixDB Metadata. The compiler translates a *connect feed* statement into a Hyracks job that is subsequently scheduled to run on an AsterixDB cluster. The resulting dataflow is referred to as a feed ingestion pipeline. A Hyracks *data operator* forms a major building block of an ingestion pipeline and is useful in executing custom logic on partitions of input data to produce partitions of output data. It may employ parallelism in consuming input by having multiple instances that run in parallel across a set of nodes in an AsterixDB cluster. *Data connectors* repartition operators' outputs to make the newly produced partitions available at the consuming operator instances. In addition, an ingestion pipeline provides *feed joints* at specific locations. A *feed joint* is like a network tap and provides access to the data flowing along an ingestion pipeline. It helps in building a cascade network of feeds by allowing data from an ingestion pipeline to be simultaneously routed along multiple paths.

A feed ingestion pipeline involves 3 stages—*intake*, *compute* and *store*. The *intake* stage involves creating an instance of the associated feed adaptor, using it to initiate the transfer of data and transforming it into ADM records. If the feed has an associated preprocessing function, it is applied to each feed record as part of the

Table 2: Policies for 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

compute stage. Subsequently, as part of the *store* stage, the output records from the preceding *intake* or a *compute* stage are put into the target dataset and its secondary indexes[1] (if any) are updated accordingly. Each stage is handled by a specific data-operator, hereafter referred to as an *intake*, *compute*, and *store* operator respectively. Next, we describe how operators, connectors and joints are assembled together to construct a feed ingestion pipeline.

Figure 10 contains some example AQL statements that define and connect a pair of feeds to respective target datasets. The second statement in Figure 10 connects the primary feed, CNNFeed. As determined by the number of topics specified in its configuration, the feed involves the use of a pair of instances of the CNNFeedAdaptor. Each adaptor instance is managed by an instance of the *intake* data operator. As *CNNFeed* does not involve any preprocessing, the output records from each adaptor instance thus constitute the feed. These are then partitioned across a set of store operator instances using the hash-partitioning data-connector. In the constructed pipeline, a feed joint is located at the output of each intake operator instance. In general, a feed joint is placed at the output side of an operator instance that produces records that form a feed. In the case where a feed involves pre-processing, a feed joint is placed at the output of its *compute* operator instances.

The last statement in Figure 10 connects the secondary feed, *ProcessedCNNFeed*. By definition, this feed can be obtained by subjecting each record from the *CNNFeed* to the associated UDF (*addInfoFromCNNWebsite*). In general, if $feed_{m+1}$ denotes the immediate child of $feed_m$, a child 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 $feed_j$ $(j = k + 1, ..., i)$. $i - k$ 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 its 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 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 cascade network for ingestion of *CNNFeed* and *ProcessedCNNFeed* is shown in Figure 11. Note that disconnecting a feed from a dataset does not necessarily remove the set of feed joints located along the ingestion pipeline. Referring to Figure 11, disconnecting CNNFeed at this stage removes the tail of the pipeline that includes the compute and store operator instances but will retain the intake operator instances. This is because the feed joints (labeled as 'A') at the output of the intake operator instances each have an existing

[1]Secondary indexes in AsterixDB are partitioned and co-located with the corresponding primary index partition. Inserting a record into the primary and any secondary indexes uses write-ahead logging and offers ACID semantics.

path (ingestion pipeline for ProcessedCNNFeed) that requires the output records to keep flowing in an uninterrupted manner.

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

connect feed CNNFeed to dataset RawArticles;

create secondary feed ProcessedCNNFeed from
feed CNNFeed apply function addInfoFromCNNWebsite;

connect feed ProcessedCNNFeed to
dataset ProcessedArticles;
```

Figure 10: Example AQL statements

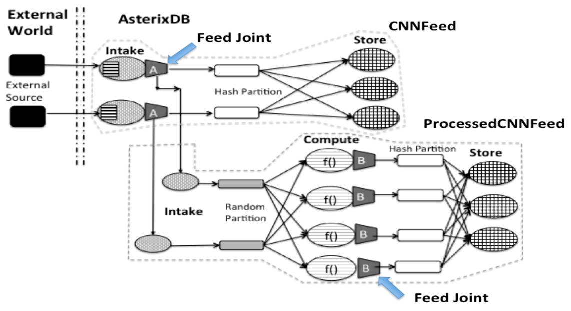


Figure 11: An example of a feed cascade network. The cascade network provides two sets of feed joints - labeled as 'A' & 'B' - that provide access to CNNFeed and CNNProcessedFeed respectively. If at this stage, the end-user creates (and connects) a secondary feed that derives from *ProcessedCNNFeed*, then its intake stage would involve receiving records from each of the 4 feed joints (kind B) provided by the ingestion pipeline for *ProcessedCNNFeed*.

5.2 Scheduling a Feed Ingestion Pipeline

Scheduling a feed ingestion pipeline on a cluster requires determining the desired degree of parallelism of each operator and mapping each instance of an operator to an AsterixDB node. An AsterixDB cluster consists of a manager node and a set of worker nodes. Scheduling decisions for a feed ingestion pipeline are taken by the *Central Feed Manager* (CFM) that is hosted by the manager node. Each worker node hosts a *Feed Manager* (FM). The CFM keeps track of the load distribution across the cluster from the periodic reports sent by each FM. These reports contain vital statistics including CPU usage. Each feed pipeline operator may define a cardinality constraint (degree of parallelism) and/or a location constraint at the Hyracks job level. The location and cardinality constraints for the intake operator are determined by the feed adaptor. If no constraints are specified, the Central Feed Manager will choose to run a single instance of the operator on a node of its choice. The constraints for the store operator are pre-determined and derived from the nodegroup associated with the target dataset. Recall that the nodegroup of a dataset refers to the set of nodes that hold the partitions of the dataset.

The compute operator in a feed pipeline doesn't offer any constraints. Instead, the level of parallelism for a compute operator is determined as described below. The partitioned parallelism employed at the compute and store stages helps the system ingest increasingly large volumes of data. Additional resources (physical machines) can be added at the compute and/or store stage to scale out the system. This helps address challenge C5 from Section 1.1. The appropriate degree of parallelism is therefore dependent on the rate of arrival of data and on the complexity associated with the UDF. To begin with, the cardinality at the compute stage is matched with that of the store stage to offer the same degree of parallelism. However, as we describe in the following section, a feed ingestion pipeline, as dictated by the ingestion policy, may be re-structured in accordance with the demand for resources.

5.3 Managing Congestion

An expensive UDF and/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 can cascade upstream to completely 'lock' the flow of data along the pipeline. A 'locked' state creates excessive demand for resources to buffer the data that is arriving at its regular rate from the data source(s). At a feed joint located on a 'locked' pipeline, insufficient resources can also impede the flow of data along other pipelines originating from the joint. To avoid such an undesirable situation, AsterixDB takes a different approach by resolving back-pressure at its originating operator and preventing it from escalating upstream. This isolates other operators in the pipeline/cascade network from the created congestion. In this sub-section, we describe the methodology adopted for detecting resource bottlenecks and taking corrective action (challenge C3 from Section 1.1).

The records arriving at an operator are buffered in memory; this functionality is provided by a *MetaFeed* operator that wraps around the actual operator (referred to as the *core* operator hereafter). Having a MetaFeed operator as a wrapper ensures that the core operators remain simple, generic and reusable elsewhere as part of other (non-feed) jobs. In addition to buffering, the MetaFeed operator periodically measures the size of the input buffer, the rate of arrival of records R_A (records/sec), and the rate of processing of records R_P (records/sec) by the core operator. Note that R_P varies with the availability of resources at the operator location and the size of the records that need to be processed. If R_A exceeds R_P, the buffer is expanded to accommodate for the deficit. The total available memory (JVM heap size) is bounded and is shared by operators serving multiple concurrent feeds or queries. To ensure sufficient resources for concurrent queries, a fixed (configurable) limit is imposed on the total memory allocated for feed input buffers at each worker node.

Table 1 from Section 4.5 described 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 mechanisms' downside is delayed processing of records (buffering/spilling) or losing some of them altogether (discarding/throttling).

Congestion that occurs due to a compute operator (i.e., due to use of an expensive UDF) can be cleared in yet another way – via 'global' resolution. Global resolution exploits the stateless and therefore embarrassingly parallel nature of the UDF. 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 to the Central Feed Manager (CFM) by each FM. Using mechanisms similar to those detailed later for handling failures during ingestion, the CFM re-structures the pipeline to have increased parallelism at the compute tier. 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. Contrary to dynamically scaling out an operator, AsterixDB also provides for auto-scaling-in if the current degree of parallelism is greater than that required to handle the flow of data. The required increase/decrease is derived from the reported values of R_A and R_P from the compute operator instances running across the cluster.

5.4 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. Once the record has been persisted (log record on disk), an *ack* message with the *tracking id* is constructed. 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. 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.

6. FAULT TOLERANT FEED INGESTION

Feed ingestion is a long running task running on a commodity cluster, so it is eventually bound to encounter hardware failure(s). Also, portions of a feed ingestion pipeline include pluggable user-provided modules (feed adaptor and a pre-processing function) that may cause soft failures (runtime exceptions). Sources of such an exception may include unexpected data format/values or simply inherent bugs in the user-provided source code. Next, we describe how a feed may recover from software and hardware failures and thereby address the challenge C4 from Section 1.1.

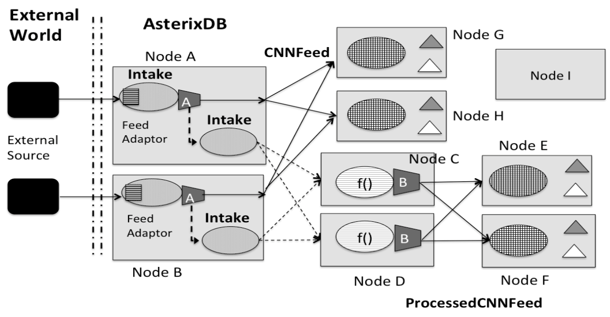
6.1 Handling Software Failures

A runtime exception encountered by an operator in processing an input record in a normal AsterixDB insert setting carries non-resumable semantics and causes the insert operation's dataflow to cease. It is essential to guard feed pipelines from such exceptions by executing each of their operators in a sandbox-like environment. The MetaFeed operator (introduced in Section 5.3) acts as a shell around each operator to provide such an environment. Recall that the operator that is wrapped is referred to as the *core* operator. The runtime of a core operator receives input data as a sequence of frames containing one ore more 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 includes the unprocessed records minus the exception generating record. The subset frame is then passed to the core-operator which continues to process input frames and has, in effect, skipped past the exception-generating record.

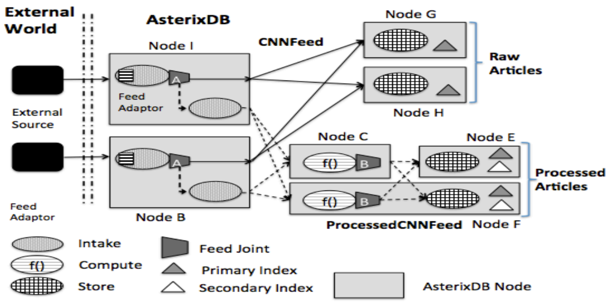
6.2 Handling Hardware Failures

We next describe the mechanism that handles the loss of one or more of the AsterixDB nodes involved in a feed ingestion pipeline. Corresponding to the operation being performed, a node is referred to as an *Intake*, *Compute* or a *Store* node. An AsterixDB cluster node may simultaneously act as an intake, compute or a store node for one or more feeds. To illustrate a failure scenario, we use an example ingestion pipeline (Figure 12(a)) that executes on a 10 member AsterixDB cluster (nodes A-I). In this particular data flow, node I is not used initially. To be considered *alive*, each node is required to send periodic heartbeats to the master node (not shown in the figure). We assume a concurrent failure of an intake node (node A) and a compute node (node D). A node failure is detected by the master node through the heartbeat mechanism. Each operator instance in the ingestion pipeline is notified of the pipeline failure. On being notified, the operator instance saves the contents of its input buffer with the local Feed Manager. The operator instance also has an option to save state information that may help in resuming operation once the pipeline is rescheduled. The operator instance then notifies the Central Feed Manager (CFM) and terminates itself. An *intake* operator instance behaves differently; it begins to buffer/spill the arriving records and not forward them downstream.

A revised feed ingestion pipeline is constructed with identical operators and feed joints. An operator instance is co-located with its respective instance from the previous failed execution if the node is still available. An intake operator instance is co-located with the corresponding *live* instance from the previous execution. Each operator then enters a 'hand-off' stage where it retrieves the input buffer and any state saved with the local Feed Manager by the corresponding instance from the failed pipeline. The functionality of registering with the Feed Manager and saving/retrieving any state across failures is provided by the *MetaFeed* operator that wraps around each core operator. An operator instance that has a *dead* instance from the previous execution can be scheduled to run at an AsterixDB node chosen by the CFM.



(a) An example dataflow for describing the fault tolerance protocol: Node A and Node D fail



(b) Restructured pipeline post recovery: Node I takes over Node A operations; Node F takes over Node D operations

Figure 12: Recovering from compute node failure.

When substituting a failed node, the CFM considers the load distribution across the cluster. Recall that the FMs periodically report vital statistics (including CPU usage) about a worker node to the

CFM. Figure 12(b) shows the revised pipeline with node I (which was idle) substituted for node A while Node F substituted for node D and thereafter acted as a compute and a store node. A more detailed description of the handling of different failure scenarios during feed ingestion can be found in [14]. Essentially the same machinery is used to handle the scaling-in or out of a feed pipeline when an elastic policy is chosen for handling data congestion (or decongestion) and that CFM determines that a scaling action is required.

A store node failure translates to the loss of a partition of the dataset that is receiving the feed. AsterixDB does not (yet!) support data replication. In the absence of replica(s), a store node failure will result in an *early* termination of an associated feed. When the failed store node later re-joins the cluster, it will undergo a log-based recovery to ensure that all of its hosted dataset partitions are in a consistent state. Subsequently, the feed ingestion pipeline will be rescheduled to involve the joined node.

7. EXPERIMENTAL EVALUATION

In this section, we provide an initial experimental evaluation of the scalability and fault-tolerance offered by the AsterixDB feed ingestion facility. We include a study of the impact of the ingestion policy (parameters) on the runtime behavior (throughput and latency) under different workload conditions. We also include a comparison with a custom built 'glued' system using Storm (as a processing engine) and MongoDB (as a persistence engine) and discuss the tradeoffs.

Experimental Setup: We ran experiments on a 10-node IBM x3650 cluster. Each node had one Intel 2.26GHz processor with four cores, 8GB of RAM, and a 300GB hard disk. We wrote a custom stand-alone tweet generator (*TweetGen*) that can output synthetic (JSON) tweets at a rate (*tweets per second* - ***twps***) that follows a configurable pattern. The *RawTweet* datatype created in Figure 2 showed the equivalent ADM representation for a tweet output by TweetGen. Next, we wrote a custom socket-based adaptor—*TweetGenAdaptor*. The adaptor is configured with the location(s) (socket address) where instance(s) of TweetGen is/are running. Each instance of TweetGen receives a request for data from a corresponding instance of *TweetGenAdaptor*, thus enabling ingestion of data in parallel. We used the AQL statements shown in Figure 3 (from Section 3.2) to create the target datasets (and indexes) for persisting the feed. The definition for the set of two feeds used in our experiment is shown in Figure 13. In this example, a pair of instances of TweetGen are running and listening on a specific for request to initiate (push-based) transfer of data.

```
create feed TweetGenFeed using TweetGenAdaptor
("datasource"="10.1.0.1:9000, 10.1.0.2:9000"));

create secondary feed ProcessedTweetGenFeed from
feed TweetGenFeed apply function addFeatures;
```

Figure 13: Feed definitions for experimental evaluation

7.1 Scalability

We evaluated the ability of the AsterixDB feed ingestion support to scale and ingest an increasingly large volume of data when additional resources are added. If the data arrival rate exceeds the rate at which it can be ingested in AsterixDB, the excess records are either buffered in memory, spilled to disk or discarded altogether. The precise behavior is chosen by the associated ingestion policy. By design, we chose to discard data here and not defer its processing (via spilling/buffering). This helped in evaluating the ability to successfully ingest data as a function of available resources.

In this experiment, we chose the amount of data loss as our performance metric. A total of 6 instances of TweetGen were run on machines outside the test cluster and were configured to generate tweets at a constant rate (20k twps) for a duration of 20 minutes. We measured the total number of ingested (persisted and indexed) tweets and repeated the experiment by varying the size of our test cluster. We increased the hardware till there is no data loss (the ideal behavior). The experimental results are shown in Figure 14. A significant proportion of records were discarded for lack of resources on a small size cluster of 1–4 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 when additional resources (nodes) are added.

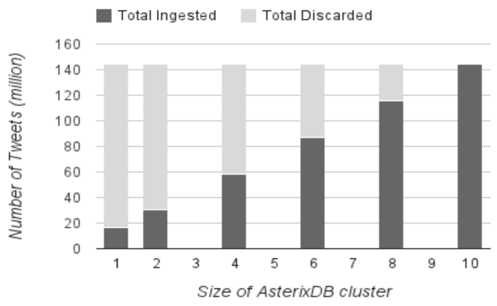


Figure 14: Scalability: Number of records (tweets) successfully ingested (persisted and indexed) as the cluster size is increased.

7.2 Fault Tolerance

We next evaluated the ability of the system to recover from single/multiple hardware failures while continuing to ingest data. This experiment involved a pair of TweetGen instances (twps=5000), each running 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* (Figure 15). 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 the absence of an available feed joint, the ingestion pipeline for *ProcessedTweetGenFeed* is constructed using the feed adaptor (Figure 16). The physical layout of the dataflow as scheduled on our AsterixDB cluster during this experiment is shown in Figure 16. The ingestion pipeline for *TweetGenFeed* is sourced from the feed joints (kind A) provided by *ProcessedTweetGenFeed*.

```
connect feed ProcessedTweetGenFeed to
dataset ProcessedTweets using policy FaultTolerant;

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

Figure 15: Connected feeds to respective dataset

We measured the number of records inserted into each target dataset during consecutive 2 second intervals to obtain the instantaneous ingestion throughput for the associated feed. We caused a compute node failure (node C in Figure 16) at t=70 seconds. This was followed by a concurrent failure of both an intake node (node

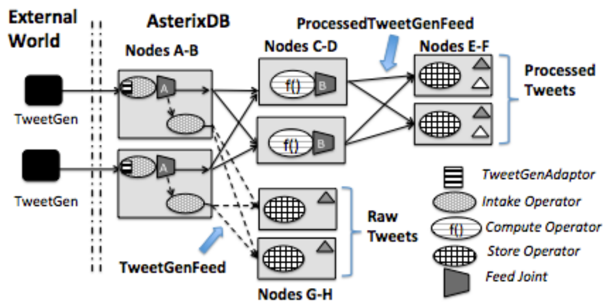
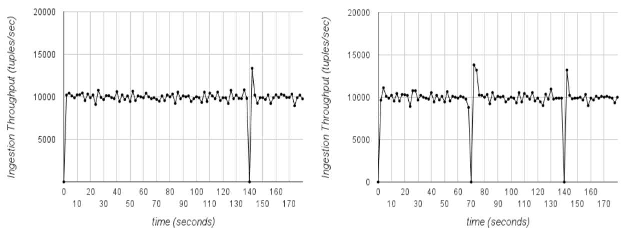


Figure 16: Feed cascade network for fault tolerance experiment: Node C fails at t=70 seconds; Node A and Node D fail at t=140 seconds



(a) TweetGenFeed (b) ProcessedTweetGenFeed

Figure 17: Instantaneous ingestion throughput with interim hardware failures: in Figure 16 Node C fails at t=70 seconds; Node A and Node D fail at t=140 seconds

A) and a compute node (node D) at t=140 seconds. The instantaneous ingestion throughput for each feed as plotted on a timeline is shown in Figure 17. Following are the 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 feed joint(s) buffer the records until the pipeline is resurrected but allow the records to flow (at their regular rate) into any other ingestion pipeline that does not involve a failed node. This helps in "**localizing**" the impact of a pipeline failure and is a desirable feature of the system. As shown in Figure 17(a), *TweetGenFeed* is not impacted by the failure of node C at t = 70 seconds.

7.3 Throughput and Latency: Impact of Ingestion Policy

We evaluated the impact of the ingestion policy on the runtime behavior and performance characteristics of interest — *throughput* and *ingestion latency* — under different conditions of rate of arrival of data. We configured two instances of TweetGen to generate tweets at a rate that followed the pattern shown in Figure 18(a). 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 $\mathrm{W_{MID}}$, $\mathrm{W_{HIGH}}$ and $\mathrm{W_{LOW}}$ respectively and the corresponding rate (tweets/second or twps) is denoted by $\mathrm{R_{MID}}$, $\mathrm{R_{HIGH}}$ and $\mathrm{R_{LOW}}$. The pre-processing of tweets involved a UDF that simply executed a busy spin loop to consume CPU cycles and cause a compute delay of about 3ms per tweet.

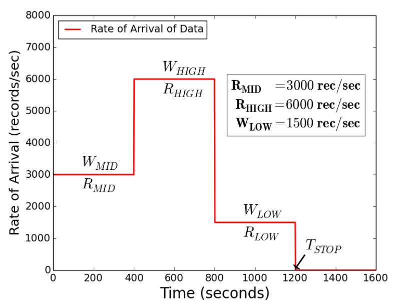
We used our 10 node AsterixDB cluster. Table 3 lists the symbols and metrics we use when describing this experiment and its results. The intake stage involved a pair of feed adaptor instances each receiving records from a separate TweetGen instance located outside the AsterixDB cluster. Each TweetGen 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 3(b)) for each tweet received by the feed adaptor during each workload-phase ($\mathrm{W_{MID}}$, $\mathrm{W_{HIGH}}$ and $\mathrm{W_{LOW}}$). 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 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 tweets/sec. The aggregate capacity from 10 parallel instances was thus limited to 3000 twps (referred to as $\mathrm{Compute_{LIMIT}}$). In the workload (Figure 18(a)), we have $R_{\mathrm{HIGH}} > \mathrm{Compute_{LIMIT}}$ 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 2, Section 4.5).

Figure 18 shows the instantaneous throughput plotted on a timeline for each policy. Each figure also cites the *average ingestion latency* ($\mathrm{L_{Avg}}$) during each workload-phase. It is desirable to maximize the *ingestion coverage* (Table 3(b)), minimize the average ingestion latency for each workload-phase and have $T_{\mathrm{DONE}} \sim T_{\mathrm{STOP}}$. The Basic and Spill policies were able to ingest all records (ingestion coverage = 1.0). However, T_{DONE} exceeded T_{STOP} due to the excess records created during W_{HIGH}. The Discard and Throttle policies had $T_{\mathrm{DONE}} \sim T_{\mathrm{STOP}}$, provided low ingestion latency but at the cost of reduced ingestion coverage (~0.66). During W_{HIGH}, Throttle policy reduced the effective tweet arrival rate at each compute node from ~600 tweets/second to ~300 tweets/second (50% sampling rate)[2]. The Elastic policy acted differently by restructuring the pipeline to involve 20 compute operator instances during W_{HIGH} as the tweet arrival rate doubled. Each node then had two compute operator instances that provided a better utilization of the cores (4) on each node, effectively increasing the $\mathrm{Compute_{LIMIT}}$ of the cluster. This helped provide a complete ingestion coverage (1.0), a minimum average ingestion latency for each workload-phase and had $T_{\mathrm{DONE}} \sim T_{\mathrm{STOP}}$. Recall that the MetaFeed operator dynamically evaluates the record arrival and processing rate at each core operator. Throttle and Elastic policies make use this periodic evaluation to derive the $\mathrm{Compute_{LIMIT}}$ and adapt the sampling rate/degree of parallelism respectively.

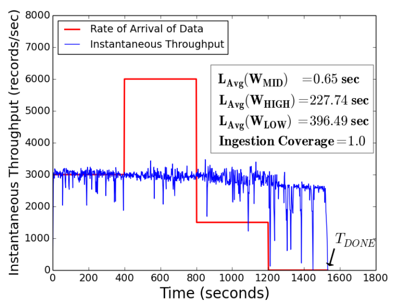
7.4 Comparison with Storm + MongoDB

An alternative way of supporting data ingestion today 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. We used our 10 node cluster to host Storm and MongoDB. 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

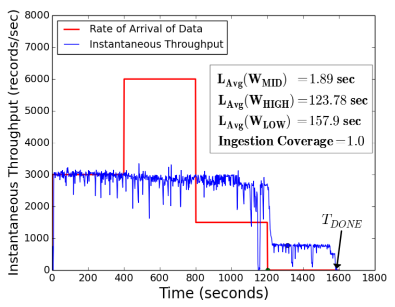
[2]Records arrive in fixed-size frames that contain varying number of records. Each frame is sampled to randomly select a subset of records for processing.



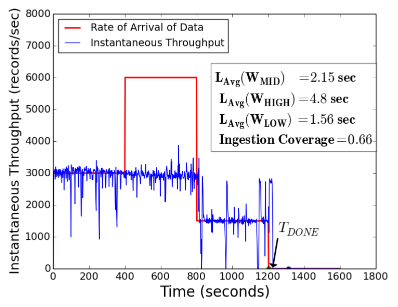
(a) Rate of Arrival of Data



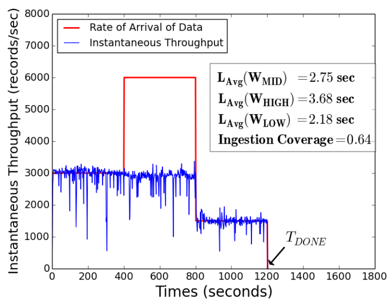
(b) Basic Policy



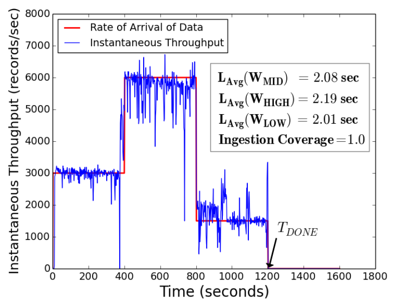
(c) Spill Policy



(d) Discard Policy



(e) Throttle Policy



(f) Elastic Policy

Figure 18: Impact of Ingestion Policy on Runtime Behavior

Table 3: 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.

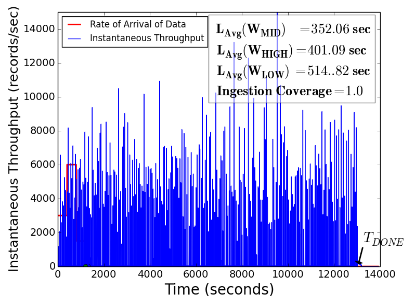
(b) Metric Definitions

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	$N_{indexed}/N_{total}$

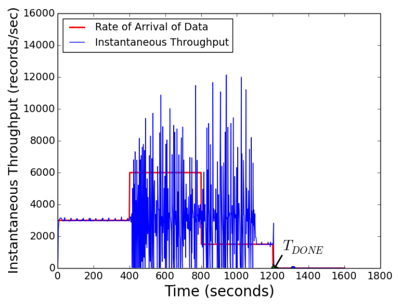
source. This method is not compatible with the common scenario of a 'push-based' ingestion where data continues to arrive from the data source at its natural rate. To support push-based ingestion, it is necessary to buffer the arriving records from the data source and then forward them to Storm on each 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.

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. Our 'glued' solution emulates 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 18(a) using both kinds of writes for MongoDB. The durable-write mode (Figure 19(a)) 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,



(a) Durable Write



(b) Non-Durable Write

Figure 19: Instantaneous Throughput for Storm+MongoDB.

we switched to using non-durable writes (Figure 19(b)) 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.

8. CONCLUSION

We have described the support for data feed management in AsterixDB (an open-source BDMS) and how it addresses the challenges involved in building a fault-tolerant data ingestion facility that scales through partitioned parallelism. We described how a feed may be defined and managed using a high-level language (AQL). A generic plug-and-play model helps AsterixDB cater to a wide variety of data sources and applications. We described the system's internal architecture and also provided a preliminary evaluation of the system, emphasizing its ability to scale to ingest increasingly large volumes of data and to handle failures during ingestion. A custom-built solution formed by 'gluing' together Storm and MongoDB was evaluated but did not compare well with the ingestion support provided by AsterixDB, neither in terms of user-experience nor its performance characteristics.

9. ACKNOWLEDGEMENTS

AsterixDB work has been supported by a UC Discovery grant and NSF IIS awards 0910989, 0910859, 0910820, and 0844574 and CNS awards 1305430, 1059436, and 1305253. Industrial supporters have included Amazon, eBay, Facebook, Google, HTC, Microsoft, Oracle Labs, and Yahoo!.

10. REFERENCES

[1] Asterixdb http://asterix.ics.uci.edu.
[2] "AsterixDB source," https://code.google.com/p/asterixdb.
[3] "Data on Big Data," http://marciaconner.com/blog/data-on-big-data/.
[4] "Informatica PowerCenter" http://www.informatica.com/in/etl/.
[5] "MongoDB," http://www.mongodb.org/.
[6] "Twitter's Storm," http://storm-project.net.
[7] D. Abadi et al. Aurora: A Data Stream Management System. *Proc. SIGMOD Conf.*, 2003.
[8] M. Balazinska et al. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *Proc. SIGMOD Conf.*, 2005.
[9] A. Behm et al. ASTERIX: Towards a Scalable, Semi-structured Data Platform for Evolving-World Models. *Proc. DAPD*, 29, 2011.
[10] P. Bonnet et al. Towards Sensor Database Systems. *Mobile Data Management*, 2001.
[11] V. R. Borkar et al. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. *Proc. ICDE Conf.*, 2011.
[12] M. Carey et al. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.*, 7(14), 2014.
[13] B. Gedik et al. SPADE: The System S Declarative Stream Processing Engine. *Proc. SIGMOD Conf.*, 2008.
[14] R. Grover and M. J. Carey. Scalable Fault-Tolerant Data Feeds in AsterixDB. *arXiv:1405.1705, CoRR*, 2014.
[15] L. Neumeyer et al. S4: Distributed Stream Computing Platform. *ICDM Workshops*, 2010.
[16] Z. Qian, Y. He, et al. Timestream: Reliable Stream Computation in the Cloud. *ACM EuroSys*, 2013.
[17] M. A. Shah et al. Highly Available, Fault-Tolerant, Parallel Dataflows. *Proc. SIGMOD Conf.*, 2004.
[18] M. Stonebraker. Operating System Support for Database Management. *Communication. ACM*, 24, 1981.
[19] Y. Xu et al. A Hadoop Based Distributed Loading Approach to Parallel Data Warehouses. *Proc. ICDE Conf.*, 2011.