An IDEA: An *I*ngestion Framework for *D*ata *E*nrichment in *A*sterixDB

Xikui Wang
University of California Irvine
xikuiw@ics.uci.edu

Michael J. Carey
University of California Irvine
mjcarey@ics.uci.edu

ABSTRACT

Big Data today is being generated at an unprecedented rate from various sources such as sensors, applications, and devices, and it often needs to be enriched based on other reference information to support complex analytical queries. Depending on the use case, the enrichment operations can be compiled code, declarative queries, or machine learning models with different complexities. For enrichments that will be frequently used in the future, it can be advantageous to push their computation into the ingestion pipeline so that they can be stored (and queried) together with the data. In some cases, the referenced information may change over time, so the ingestion pipeline should be able to adapt to such changes to guarantee the currency and/or correctness of the enrichment results.

In this paper, we present a new data ingestion framework that supports data ingestion at scale, enrichments requiring complex operations, and adaptiveness to reference data changes. We explain how this framework has been built on top of Apache AsterixDB and investigate its performance at scale under various workloads.

PVLDB Reference Format:
Xikui Wang, Michael J. Carey. An IDEA: An Ingestion Framework for Data Enrichment in AsterixDB. *PVLDB*, 12(11): 1485-1498, 2019.
DOI: https://doi.org/10.14778/3342263.3342628

1. INTRODUCTION

Traditionally, data to be analyzed has been obtained from one or more operational systems, fed through an Extract, Transform, and Load (ETL) process, and stored in a data warehouse [11]. In today's Big Data era, the data that people work with is no longer limited to the operational data from a company but also includes social network messages, sensor readings, user click-streams, etc. Data from these sources is generated rapidly and continuously. It becomes increasingly undesirable to stage the data in large batches, process it overnight, and then load it into a data warehouse due to the volume of the incoming stream and the need to analyze current data when making important decisions.

To support the ingestion of continuously generated data and provide near real-time data analysis, streaming engines have been introduced into the Big Data analysis architecture [7, 14, 36]. The incoming data is collected by a streaming engine and then pushed to (or periodically pulled by) a warehouse for later complex data analysis. Adding a streaming engine simplifies the ingestion process for the data warehouse, but it introduces data routing overhead between different systems. To minimize this overhead and simplify the architecture, some systems such as Apache AsterixDB [17] have chosen to provide an integrated ingestion facility, data feeds, to enable users to ingest data directly into the system.

The ingested data, such as sensor readings, is often not useful alone in high-level data analysis. To reveal more valuable insights, it needs to be enriched, e.g, by relating it to reference information and/or applying machine learning models. When the enriched data needs to be queried frequently, its computation is often pushed into the ingestion pipeline and the enriched data is then persisted [24]. This requires the ingestion framework to have the ability to process incoming data efficiently and to access reference data/machine learning models when needed.

Most streaming engines support incoming data processing, but some only support a limited query syntax [9, 22]. If a processing task requires existing data from a warehouse, most streaming engines would need to query the warehouse repeatedly, as otherwise they would have to keep a copy of that data locally. Frequent queries to the warehouse would increase the load on the system and incur latency, and maintaining multiple copies of the data would require data migration to keep the data consistent [24]. Both choices would slow down the enrichment/ingestion pipeline and increase the complexity of building the overall analysis platform.

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

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342628

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

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

The rest of this paper is organized as follows. We introduce the background information about Apache AsterixDB, its Hyracks runtime engine, and rapid data ingestion in Section 2, and then we discuss how to enrich ingested data at scale in Section 3. In Section 4, we investigate different strategies for utilizing data enrichment for data analysis and the current limitations of each for providing current and correct data enrichment in time. We explain how we built the new ingestion framework and the techniques used in Section 5, and we elaborate on the details of the new ingestion framework in Section 6. We investigate its performance in Section 7, review related work in Section 8, and conclude our work in Section 9.

2. BACKGROUND

2.1 Apache AsterixDB

Apache AsterixDB [2] is an open source Big Data Management System (BDMS). It provides distributed data management for large-scale, semi-structured data. It aims to reduce the need for gluing together multiple systems for Big Data analysis. AsterixDB uses SQL++ [10, 27] (a SQL-inspired query language for semi-structured data) for user queries and *the AsterixDB Data Model* (ADM) to manage the stored data. ADM is a superset of JSON and supports complex objects with nesting and collections.

2.2 Hyracks

Hyracks [6] is a partitioned parallel computation platform that provides runtime execution support for AsterixDB. Queries from users are compiled into Hyracks jobs. A "job" is a unit of work that can be executed on Hyracks. A "job specification" describes how data flows and is processed in a job. It contains a DAG of operators, which describe computational operations, and connectors, which express data routing strategies. Data in a runtime Hyracks job flows in frames containing multiple objects. An operator reads an incoming data frame, processes the objects in it, and pushes the processed data frame to another connected operator through a connector. AsterixDB uses jobs to evaluate user queries. A query submitted to AsterixDB is parsed and optimized into a query plan and then compiled as a job specification to run on the Hyracks platform. Figure 1 shows an example of how a user query can be represented as a Hyracks job.

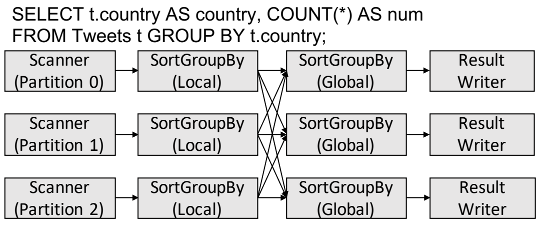


Figure 1: Translating a user query to a Hyracks job

2.3 Data Ingestion

In many contemporary data analysis use cases, data no longer stays on storage devices to be batched into a database system later. Instead, it enters the system rapidly and continuously. The traditional bulk loading technique cannot be applied due to the active nature of the incoming data. Repeatedly issuing insert statements would be impractical because of its low efficiency.

To untangle database systems and users from the handling of rapidly incoming data, one popular solution is to couple a streaming engine with a database system and use the streaming engine to handle the data [24]. For example, one can set up a Kafka instance to ingest data from external sources, then create a program [26] or use the *kafka-mongodb-connector* [16] to transmit the ingested data to a MongoDB instance for later analysis.

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

To simplify the process of ingesting data for end users and improve ingestion efficiency for analytical systems, some database systems provide integrated data ingestion facilities for handling rapid incoming data. Apache AsterixDB, for example, provides data feeds that allow users to assemble simple data ingestion pipelines with DDL statements [17]. A data feed consists of two components: an adapter, which obtains/receives data from an external data source as raw bytes; and a parser, which translates the ingested bytes into ADM records. Compared with the "glue" solution using streaming engines, data feeds have no extra data routing overheads, and a user can easily assemble a basic data ingestion pipeline with declarative statements.

3. BIG DATA ENRICHMENT

3.1 Motivation

Due to various restrictions, such as the limited bandwidth of infield sensors and predefined data formats from API providers, data coming from external sources may not contain all the information needed for meaningful data analysis. In these scenarios, one can enrich the ingested data with existing knowledge (reference data) or machine learning models to reveal more useful information. For example, the IP address in a log record can be enriched by referencing IP reputation data to see whether there is known threat activity associated with that IP address [21]. Tweets can be enriched by utilizing linguistic processing, semantic analysis, and sentiment analysis techniques and can be used in societal event forecasting systems [13]. Raw data collected by sensor networks can be enriched to support higher level applications, such as improving training effects in sports [12] and building health-care services for medical institutions [29].

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

3.2 UDFs for Data Enrichment

AsterixDB supports both Java and SQL++ in its UDF framework. One can implement a Java UDF that utilizes the provided API to manipulate an input record, or create a SQL++ UDF to enrich input data using a declarative query. For example, we might want to create a UDF that checks whether a given tweet from the U.S. contains the keyword "bomb". If so, the UDF should add a new field, "safety_check_flag", to the tweet and sets it to "Red". If not, the UDF should add the same new field and sets it to "Green". Figure 2 shows an example of the Java UDF implementation (Java UDF 1) of such a UDF.

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

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

Figure 2: Java UDF 1 for tweet safety check

Although Java UDFs are powerful tools for enriching incoming data, especially when combined with machine learning models, constructing Java UDFs can be more complicated than writing SQL++ queries when expressing the same data enrichment requirements. In addition, a SQL++ UDF can be updated using an *UPSERT* statement instantly while updating a Java UDF requires a recompilation and redeployment process. Figure 3 shows an equivalent SQL++ UDF that performs the safety check for a given tweet (SQL++ UDF 1).

3.3 Utilizing Existing Knowledge

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

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

Figure 3: SQL++ UDF 1 for tweet safety check

UDF in AsterixDB can load external files during its initialization. A SQL++ UDF can access reference data stored in datasets. To expand on our tweet safety check example in Section 3.2, given a list of countries and their sensitive keywords, suppose we want to flag a tweet from a country if it contains one of the keywords associated with that country. For a Java UDF, we can put the country-to-keywords mappings into a local resource file and load it during the UDF initialization. For a SQL++ UDF, we can store the mappings in a "SensitiveWords" dataset and use a SQL++ query to enrich the input data. A snippet of the Java implementation of this UDF is included in [35]. Figure 4 shows its SQL++ implementation (SQL++ UDF 2).

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

Figure 4: SQL++ UDF 2 for tweet safety check

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

4. DATA ENRICHMENT FOR ANALYSIS

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

4.1 Option 1 - Enrich during Querying

For data enrichment used in one-time analytical queries, one can apply enrichment UDFs directly when querying the data. A UDF in an analytical query can be optimized together with the query to produce an optimized query plan. For example, to find out how many tweets in each country are marked as "Red", a sample analytical query using a SQL++ UDF 2 is shown in Figure 5. Since the data enrichment is evaluated together with the analytical query, the query response time can be long in the case of complex enriching UDFs. Also, as the enriched data is not persisted, the same enrichment needs to be computed multiple times for each incoming analytical query.

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

Figure 5: An analytical query using SQL++ UDF 2

4.2 Option 2 - Enrich during Data Ingestion

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

4.2.1 Approach 1 - External Programs

A naive approach to enrich data during ingestion would be to set up an external program that obtains/receives data from data sources, issues DML statements to enrich the collected data, and then inserts the enriched data into a dataset. A sample insert statement that enriches data using SQL++ UDF 2 and inserts the result into a target dataset "EnrichedTweets" is shown in Figure 6. However, as discussed in Section 2.3, issuing repeated insert statements has significant overheads and would not scale well.

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

Figure 6: Enrich and insert collected tweets

4.2.2 Approach 2 - External Programs w/ Data Feeds

A user can use the basic data feeds feature introduced in Section 2.3 to improve ingestion performance. The data can first be ingested into a dataset using data feeds, then enriched and stored in another dataset by applying UDFs. A user could set up an external program that repeatedly issues the DML statement in Figure 7 to initiate data enrichment for ingested data. Depending on the arrival rate of incoming data, the user may issue a new DML statement as soon as the previous one returns to catch up with the ingestion progress when the arrival rate is high, or wait for a certain period to batch the ingested data when the arrival rate is low. Benefiting from data feeds, this approach consumes the incoming data efficiently, even when the data comes in fast, but a user still needs to set up an external program that constantly initiates the data enrichment. In addition, the data is unnecessarily materialized twice since all information in the tweets is kept in the enriched tweets as well.

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

Figure 7: Enrich and insert ingested tweets

4.2.3 Approach 3 - Data Feeds w/ UDFs

In order to avoid the unnecessary materialization of incoming data and make the enriched data available to users as soon as possible, we may attach the data enrichment operation directly to the ingestion pipeline so that the ingested data is enriched before it arrives at storage. Apache AsterixDB allows users to attach certain UDFs to data feeds. As an example, a user could attach SQL++ UDF 1 (in Figure 3) to a data feed using the DDL statement in Figure 8. Incoming tweets are first received by the feed adapter, then parsed by the feed parser, and then enriched by the attached UDF. Finally, they are stored in the connected dataset. A Java UDF, such as the UDF in Figure 2, can also be attached to a data feed.

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

Figure 8: Attach a SQL++ UDF to a data feed

4.3 More Complex Enrichment

4.3.1 Challenges

In AsterixDB today, UDF 1 can be attached to a data feed directly, as it only accesses the incoming record and does not create any intermediate states. We call this kind of UDF a ***stateless*** UDF. UDF 2 is different from UDF 1, as UDF 2 accesses external resources (the "SensitiveWords" dataset in the case of SQL++, or the equivalent local resource files in the case of Java) and creates intermediate states (such as in-memory hash tables) used for data enrichment. We call this kind of UDF a ***stateful*** UDF.

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

4.3.2 Model 1 - Evaluate UDF per Record

A simple computing model for applying a stateful UDF to a feed is to evaluate the attached UDF against each incoming record separately. An incoming record is received and parsed by the feed adapter and parser first, then enriched and persisted in storage. An equivalent insert statement for enriching and persisting one record is shown in Figure 9. In this model, each collected datum is treated as a new constant record. The attached UDF evaluates each record separately, and any intermediate states will be refreshed from record to record. This allows the UDF to see data changes during the ingestion process, and it imposes no limitations on the applicable query constructs in attached UDFs. However, evaluating the UDF on a per-record basis may introduce a lot of execution overhead. This model cannot be applied in situations where the data arrives rapidly.

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

Figure 9: Enrich and insert a constant record

4.3.3 *Model 2 - Evaluate UDF per Batch*

To mitigate the execution overhead, one alternative is to batch the collected incoming records, apply the UDF to the batch, and store the enriched records. An equivalent insert statement for enriching a batch of records was shown in Figure 6. The records within one batch are enriched using the same reference data, and reference data changes are captured between batches. A larger batch leads to lower execution overhead but less immediate sensitivity to reference data changes; the converse is also true. A user may choose a balance between ingestion performance and sensitivity to reference data changes by tuning the batch size.

4.3.4 *Model 3 - Stream Datasource*

To further reduce execution overheads, the system could attempt to treat the incoming data stream as an infinite dataset and evaluate the attached UDF as if the stream is a normal dataset. An equivalent insert statement is shown in Figure 10[1].

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

Figure 10: Enrich and insert records from a feed

This model would be more efficient than the previous two, as the attached UDF is initialized once for all incoming data. Any pre-computation for enriching the incoming data occurs only once and is used for all incoming data. Although this model would provide the best ingestion performance since it has the smallest execution overhead, it cannot be used when the attached UDF is ***stateful***. Taking SQL++ UDF 2 as an example, when we attach this UDF to a data feed and use this model to compute it, the evaluation would become a join operation between the "SensitiveWords" dataset and the never-ending feed data source. When there is a more complicated UDF, such as multi-level join and group-by, the evaluation could create more intermediate states and become even harder to evaluate using this model. Here we list three different scenarios of evaluating UDF 2 using Model 3, depending on the join algorithm and the size of the "SensitiveWords" dataset.

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

   The evaluation of a hash join operation consists of two phases: build and probe [32]. In the build phase, the "SensitiveWords" dataset would be built into a hash table. In the probe phase, the data coming from the Twitter feed would then use the hash table to find the matching records in the "SensitiveWords" dataset.

   When the "SensitiveWords" dataset is small, the created hash table can be kept in memory. This allows incoming data to continuously probe the in-memory hash table for enrichment while the ingestion continues as shown in Figure 11. This appears to be a perfect model for this case, but it cannot incorporate the new changes to the "SensitiveWords" dataset, as the in-memory hash table would be built once and then used throughout the streaming ingestion process.

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

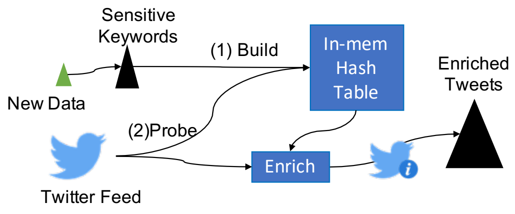


Figure 11: Case 1: Small SensitiveWords dataset

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

   When the "SensitiveWords" dataset is large, part of its data will be spilled to disk for the next round of the join [32]. This is shown in Figure 12. The hash join algorithm expects to process such spilled data recursively, after reading "all" data from Twitter, but of course the tweets will not stop coming. Thus, this model cannot be used in this case.

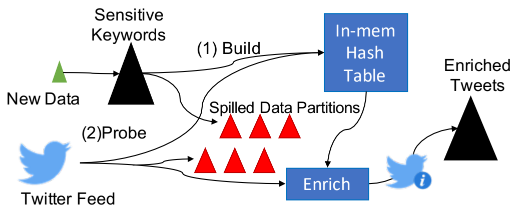


Figure 12: Case 2: Big SensitiveWords dataset

3. *Index Nested Loops Join*

   If there is an index on the "country" attribute of the "SensitiveWords" dataset, the SQL++ query compiler may choose the index nested loop join algorithm to compute the join. In this case, the incoming data can be used to look in the index first, then find the matched records for enrichment, as shown in Figure 13. By choosing this join algorithm manually for this specific join case, one could avoid creating intermediate states during the enrichment operation and thus see the new data changes directly. However, this approach is not applicable to more general use cases where the indexes on referenced datasets may not always exist, and/or where an enrichment UDF contains other operations that create intermediate states.

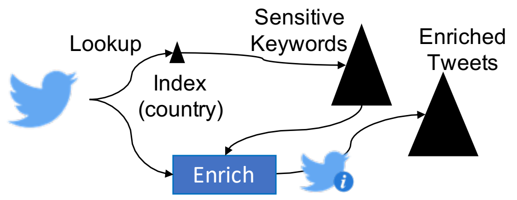


Figure 13: Case 3: Enrich with an available index

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

5. FRAMEWORK BUILDING BLOCKS

As discussed in Section 4.3, only models 1 and 2 support complex data enrichment during data ingestion and capture any reference data changes at the same time. We have thus built a new data ingestion framework based on model 2, as it provides flexibility by allowing users to choose the right batch sizes for their use cases. In this section, we describe the design of this new framework and the optimization techniques that we used for improving its performance.

5.1 Predeployed Jobs

Following model 2, our new ingestion pipeline consists of two independent Hyracks jobs: an *intake job* and an *insert job*. A sample ingestion pipeline on a three-node cluster is shown in Figure 14. The intake job contains the feed adapter and parser, and this job runs continuously throughout the lifetime of the ingestion process. The insert job takes a batch of records from the intake job, enriches them by applying the attached UDFs, and inserts the enriched records into a dataset. It runs repeatedly, being invoked once per batch, during ingestion. In each invocation, the insert job sees the updates to a referenced data record before it is first accessed. Updates after that are picked up by the next invocation[2].

The insert job in Figure 14 is constructed using the query in Figure 6. For every collected batch of records from the intake job, we replace the array of constant records (in TweetsBatch) with the collected batch and execute it. As discussed in Section 2.2, a query in AsterixDB is optimized and compiled into a job specification first, then distributed to the cluster for execution. Since the insert job is executed repeatedly, we utilize *parameterized predeployed jobs* to avoid redundant query compilation and job distribution costs.

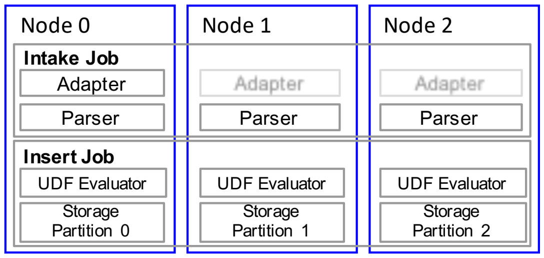


Figure 14: Ingestion pipeline using insert jobs

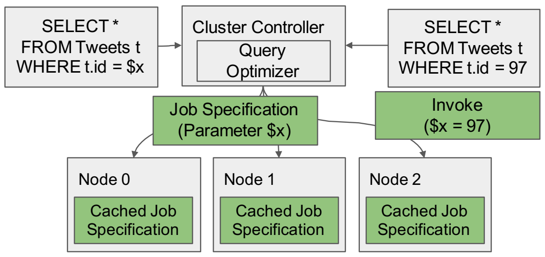


Figure 15: Parameterized predeployed job

Parameterized predeployed jobs are not unlike prepared queries in relational databases. As shown in Figure 15, a user can choose to predeploy a query with specified parameters. This query is optimized and compiled normally, and then the compiled job specification is predeployed to all nodes in the cluster. This job specification is then cached on the cluster nodes. When a user wants to run this query with a particular parameter, instead of repeating the entire query compilation and distribution process, an invocation message with the new invocation parameter is sent. Using this technique, Figure 14's insert job is distributed as a predeployed job in the cluster before the feed starts. When the intake job obtains a new batch of records, it invokes a new insert job with the collected batch as the parameter.

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

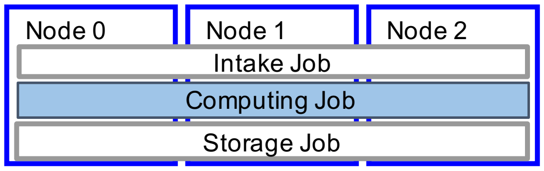


Figure 16: Decoupled ingestion framework

5.2 Layered Ingestion Pipeline

Repeatedly executing the insert job allows any intermediate states created in the UDF evaluation to be refreshed so that any data changes will be used for enriching the incoming data. It should be noted that the evaluation of an insert job, similar to the evaluation of an insert query, will have to wait for the storage log to be flushed to finish properly. Also, since the UDF evaluation and storage operations work sequentially in an insert job, UDF evaluation can be blocked while waiting for the downstream data to be written into storage. To fully utilize the cluster's computing resources and improve overall throughput, we further decompose the insert job into a computing job and a storage job so they can work concurrently. The decoupled ingestion framework is shown in Figure 16.

In the decoupled ingestion framework, the intake job handles data from external data sources, the computing job evaluates the attached UDFs, if any, and the storage job writes the enriched data into storage. The intake job and storage job begin to run when the data feed starts, while the computing job in between them is run repeatedly as data batches come in. As in the previous discussion, the computing job is distributed as a predeployed job to reduce execution overhead. Similar to the insert job in Section 5.1, an invocation of the computing job will see the updates to a referenced record before it is first accessed by the job.

5.3 Partition Holders

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

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

A partition holder operator "guards" a runtime partition by holding the incoming data frames in a queue with a limited size. There are two types of partition holders, active and passive, as shown in Figure 17. An active partition holder follows the default **push** strategy in Hyracks; it receives data frames from other jobs and pushes them to its downstream operators actively. A passive partition holder implements a **pull** strategy; it receives data frames from its upstream operators and waits for other jobs to pull them. Each partition holder has a unique ID that is associated with its partition number. When a new partition holder is created, it registers with the local partition holder manager. Jobs sending/receiving data to/from another job can locate the corresponding partition holders through local partition holder managers. In the decoupled ingestion framework, we add a passive partition holder to the tail of the intake job so that the computing jobs can request and receive data in batches. An active partition holder is added to the head of the storage job so that computing jobs can push the enriched data on to the storage job.

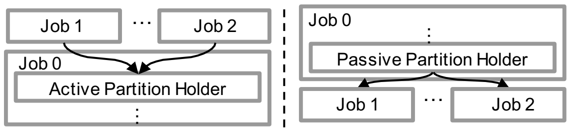


Figure 17: Partition holders

6. THE NEW INGESTION FRAMEWORK

Following the high-level design in Section 5, we now detail the new ingestion framework in AsterixDB to support data enrichment with reference data updates. We describe how we orchestrate different components in the new ingestion framework in Section 6.1, and we delve into the lower-level constructs of the framework in Section 6.2.

6.1 Ingestion Life Cycle

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

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

When a user submits a start feed request, the CC creates the intake, computing, and storage jobs based on a compiled job specification that is generated from a query template similar to Figure 6. The intake and storage job run directly, and the computing job is predeployed into the cluster for later invocations. The AFM maintains the mappings of data feeds to predeployed computing jobs so that it can invoke new computing jobs for each data feed separately.

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

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

6.2 New Ingestion Architecture

The new ingestion framework consists of the intake, computing, and storage jobs. All jobs run on all nodes in an AsterixDB cluster. As explained in Section 2.2, a Hyracks job contains operators and connectors. In order to demonstrate how data is processed and transported in the new ingestion framework, Figure 18 shows the composition of the framework running on three nodes at the operator and connector level:

- *The* ***intake job*** obtains/receives data from external data sources. The data enters the system through the Adapter. The Adapter collects data as raw bytes and arranges them into data frames for transportation purposes in the system. A user may choose to activate the Adapter on one or more nodes depending on the expected load. The ingested data frames are then fed through the Round-robin Partitioner to be distributed in a round-robin fashion. Since the attached UDFs can be expensive, distributing the incoming data evenly can help to minimize the overall execution time of the computing job. The partitioned data is forwarded to the Intake Partition Holder, which is implemented as a passive partition holder, which then waits for computing jobs to pull the data.
- *The* ***computing job*** evaluates the attached UDF to enrich data batches. A computing job starts by collecting a data batch from a local intake partition holder. The obtained data batch is first parsed by the Parser and then fed to the UDF Evaluator for data enrichment. Depending on the attached UDF, the UDF evaluator could be a Java program that runs on each node independently, or it could be a group of operators produced by compiling a complex SQL++ UDF. In either case, local resource files (for Java UDFs) or reference datasets (for SQL++ UDFs) may be accessed, and/or intermediate states might be created as well. After being enriched, the data is pushed to the Feed Pipeline Sink to be forwarded to the storage job.
- *The* ***storage job*** receives enriched data and stores it to disk. The enriched data is first received by the Storage Partition Holder, which is implemented as an active partition holder. A feed pipeline has one Storage Partition Holder on each node, and the Storage Partition Holder receives the enriched data from all local partitions of the collocated computing job. The Storage Partition Holder pushes the received enriched data actively to the connected Hash Partitioner. The

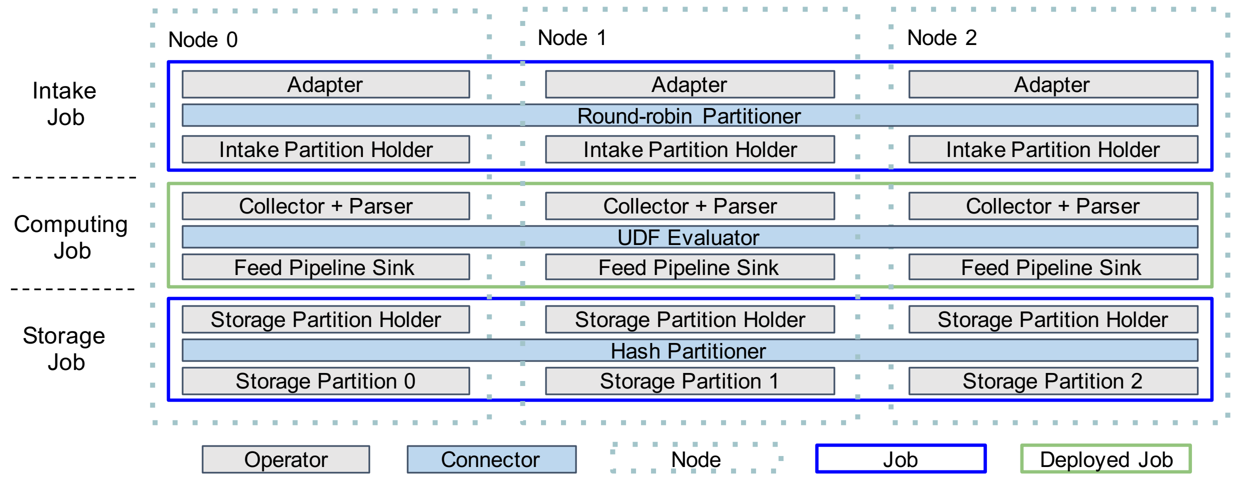


Figure 18: The new ingestion framework

Hash Partitioner partitions the enriched data records by their primary keys so they can be stored in the appropriate Storage Partitions.

7. EXPERIMENTS

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

7.1 Basic Data Ingestion

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

For this purpose, we compared the throughput of both the current and new frameworks for continuous tweet ingestion. We continuously fed tweets into both ingestion frameworks and measured the throughput of each while consuming 10,000,000 tweets. Each tweet record is around 450 bytes. The results are shown in Figure 19. To make sure a single intake node did not become a bottleneck for the ingestion performance, we also tested a "balanced version" of both the current and new ingestion framework by having all nodes in the cluster act as intake nodes. We refer to the experiments on the current framework as "Static Ingestion", on the new framework as "Dynamic Ingestion", on the balanced version of the current framework as "Balanced Static Ingestion", and on the balanced version of the new framework as "Balanced Dynamic Ingestion".

To explore how batch size affects the ingestion performance of the new ingestion framework, we experimented

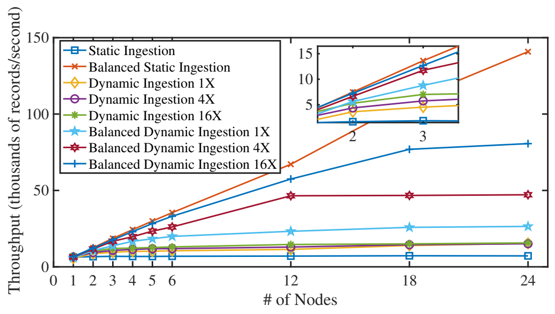


Figure 19: 10M tweets ingestion speed-up over 24 nodes

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

As we can see in Figure 19, the ingestion performance of Static Ingestion remained the same as the cluster size increased. This is because data intake and parsing are coupled in the current ingestion pipeline. In this case, the ingestion performance was limited by the parsing bottleneck on a single intake node. In contrast, Balanced Static Ingestion kept improving as more nodes participated in parsing and ingesting the data. In the new ingestion framework, data parsing and intake are decoupled, so even for a single intake node (Dynamic Ingestion), the intake performance increased as more nodes participated when the cluster size is small.

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

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

7.2 Data Enrichment with UDFs

We now turn to the performance of the new ingestion framework in enriching data during data ingestion. We designed four sample use cases where the attached UDFs cover several common operations used in database queries, including join, group-by, order-by, similarity join, and spatial join. The four use cases are as listed below, and their complete queries can be found in [35]. The reference datasets are SafetyRatings, with 500,000 records and 74 bytes each, ReligiousPopulations, with 500,000 records and 137 bytes each, SuspectsNames, with 5,000 records and 150 bytes each, and MonumentList, with 500,000 records and 94 bytes each.

1. *Safety Rating:* Given a list of countries and their corresponding safety ratings, enrich a tweet with a safety rating based on its "country" field value. (Hash join)
2. *Religious Population:* Given the population of each religion in every country, enrich a tweet with the overall religious population based on its "country" field value. (Group-by)
3. *Largest Religions:* Given the population of each religion in every country, enrich a tweet with the three largest religions according to its "country" field value. (Order-by)
4. *Fuzzy Suspects:* Given a list of suspects' names, enrich a tweet with the possible suspects whose name's edit distance to the tweet user's screen name, after removing all special characters, is less than five characters. (Java string processing, Similarity join)
5. *Nearby Monuments:* Given a list of monuments and their coordinates, enrich a tweet with the monuments within 1.5 degrees of the tweet's location. (Index nested loop spatial join)

All of these enrichment UDFs are stateful, and their evaluations involve the challenges that we discussed in Section 4.3. As we have mentioned, the current ingestion pipeline of AsterixDB doesn't support such stateful SQL++ UDFs on its ingestion pipeline. Java UDFs attached on the current AsterixDB ingestion pipeline can only handle reference data without updates. For comparison purpose, we experimented with Java UDFs in current AsterixDB and denote the results as "Static Enrichment w/ Java".

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

In order to measure the performances of both Static Enrichment and Dynamic Enrichment, we deployed the system on a 6-node cluster and fed tweets to the ingestion pipeline for data enrichment continuously. We measured the throughput (records / second) of the system spent while enriching 1,000,000 tweets, as shown in Figure 20 (in log scale). The refresh periods (i.e., execution time per batch) of Dynamic Enrichment w/ SQL++ are shown in Figure 21.

In most of the use cases, except for Nearby Monuments (to be discussed shortly), Static Enrichment offered higher throughput than Dynamic Enrichment. This is because Static Enrichment only loaded reference data once and then reused its stale intermediate states throughout the whole ingestion process. Dynamic Enrichment, however, refreshed and reconstructed those intermediate states from batch to batch. This allowed Dynamic Enrichment to capture data changes to reference data but with the overhead of repeatedly invoking computing jobs. For both Java and SQL++ UDFs, the throughput increased with larger batch sizes since they led to fewer computing jobs and a smaller execution overhead. Accordingly, the refresh periods grew, as there were more records to be enriched in larger batches.

For Fuzzy Suspects and Nearby Monuments, the throughput did not improved that much with increased batch size. The reason was that the computation costs of edit distance and spatial join were high and proportional to the cardinality of the incoming data. Job initialization and management overheads were small relative to these costs. Thus, increasing the batch size didn't improve the throughput significantly. In Fuzzy Suspects in particular, the attached SQL++ UDF not only computed edit distance but also invoked a Java UDF for removing special characters. This introduced extra data serialization/deserialization and shuffling cost. In Nearby Monuments, we created an R-Tree index for the monuments' location in the reference dataset. The use of the index allowed the SQL++ UDF to outperform the Java UDF in this case by performing index lookups on partitioned reference data.

7.3 Data Enrichment with Updates

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

As we can see from Figure 22, reference data updates affect the ingestion and enrichment performance differently depending on the cardinality of the referenced dataset and the access method used in data enrichment operations. The throughputs of all cases dropped when the update rate first changed from none to one record per second. This was due to the resulting increased cost in accessing reference data. AsterixDB uses log-structured merge-trees (LSM Trees) in its storage [3]. Updates to a dataset will activate the in-memory component of its LSM structure and thereby change how the system accesses data even at the low rate of one record per second. This added additional data fetching, locking, and comparison costs to reference data access in computing jobs, which then slowed down the ingestion throughput. Since Fuzzy Suspect had the smallest reference dataset among all, it was the least affected by the updates. When increasing the update rate from there, the throughput then decreases grad-

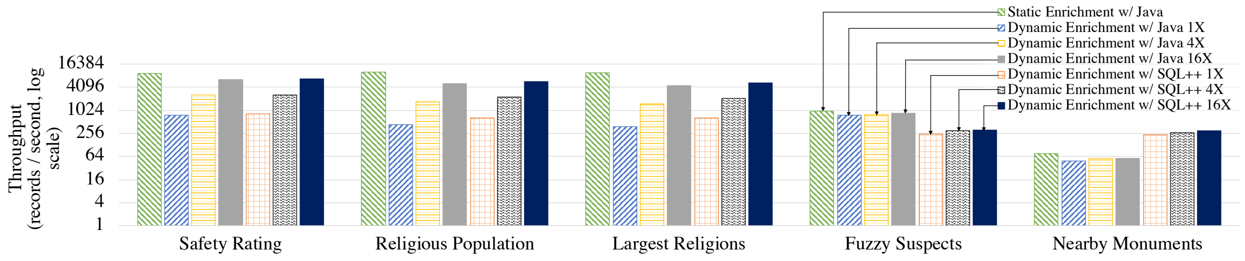


Figure 20: 1M tweets Ingestion with UDFs (log scale)

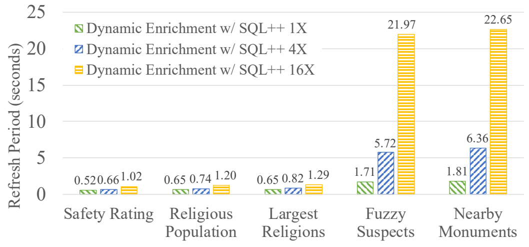


Figure 21: Refresh periods under different batch sizes

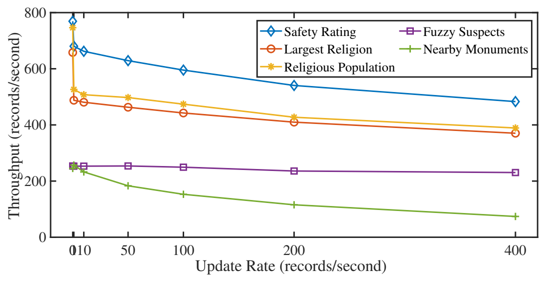


Figure 22: Reference data update

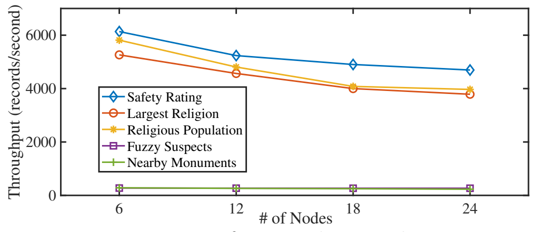


Figure 23: Reference data scale-out

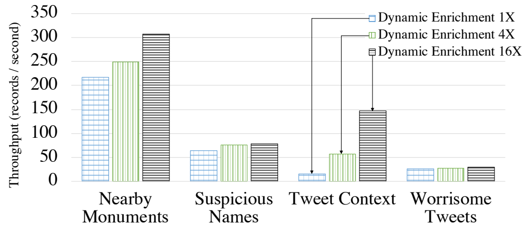


Figure 24: UDF complexity comparison

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

7.4 Scale-out Experiments

7.4.1 *Reference Data Scale-out*

In the new ingestion framework, since the intermediate states are refreshed repeatedly, the size of the reference data could become a important factor for the data ingestion and enrichment performance. In this section, we explore how the new ingestion framework scales with the size of the reference datasets. We started with the reference datasets in Section 7.2 and increased their sizes to 2X, 3X, and 4X, together with increasing the cluster size to 12 nodes, 18 nodes, and 24 nodes correspondingly. Similarly, we continuously fed tweet data for data enrichment using the same set of SQL++ UDFs and measured the throughput after 1,000,000 tweets with 6720 records/batch. As we can see from the results in Figure 23, the throughput dropped slightly when we increased the size of the cluster due to the increasing execution overhead on a larger cluster. The new ingestion framework scaled well with the reference data size.

7.4.2 *Ingestion Data Scale-out*

In order to further investigate the performance of the new ingestion framework for large scale data enrichment, we designed three new complex data enrichment use cases that add more information to the incoming tweets. The more complex a data enrichment function is, the more performance impact it will have on the whole ingestion framework. We tested the new framework with these new cases to see how it scales. The additional use cases are listed below, and their complete queries can be found in [35]. The reference datasets are ReligiousBuildings, with 10,000 records and 205 bytes each, Facilities, with 50,000 records and 142 bytes each, SensitiveNames, with 1,000,000 records and 155 bytes each, AverageIncome, with 50,000 records and 99 bytes each, DistrictArea, with 500 records and 121 bytes each, Residents, with 1,000,000,000 records and 124 bytes each, and AttackEvents, with 5,000 records and 179 bytes each.

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

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

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

To demonstrate the complexity of these additional use cases, we compared their enrichment performance with that of "Nearby Monuments", the most complex UDF from the previous experiment. We measured their throughput on the

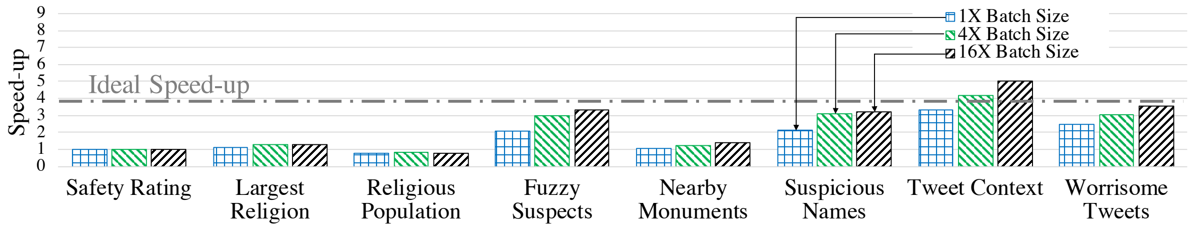


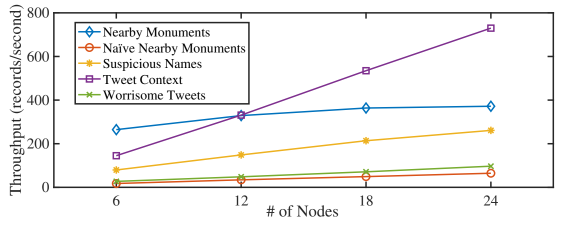
Figure 25: 100K tweets ingestion speed-up for 24 vs. 6 Nodes with different batch sizes

new ingestion framework for 100,000 tweets enrichment on a 6-node cluster. As shown in Figure 24, the added use cases had different complexities, and different use cases benefited from batch size changes differently. In the Tweet Context use case, there were multiple expensive spatial joins between the referenced datasets before joining with the tweets. Increasing the batch size reduced the computation cost and thus increased the overall ingestion throughput. In the other cases, the tweets mostly joined with the reference datasets sequentially. Thus, increasing the batch sizes offered limited improvements in the Nearby Monuments, Suspicious Names, and Worrisome Tweets use cases.

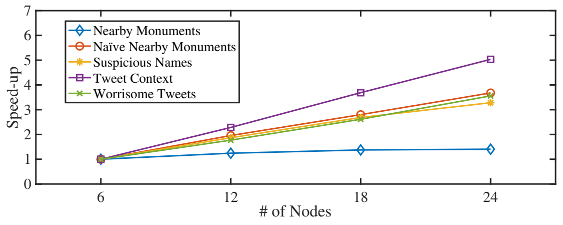
The performance of scaling out the new framework is determined by the cluster size, batch size, and UDF complexity. Although increasing the number of nodes for computation can reduce the execution time of a computing job, it also introduces additional execution overhead for executing jobs on a larger cluster, so adding more resources may not always improve the overall ingestion time. Given a simple enrichment UDF, a small batch size, and a large cluster, the speed-up performance might be bounded by the batch execution overhead. To explore the relationship of these three factors, we experimented with the speed-up performance using different batch and cluster sizes.

We let the framework ingest and enrich 100,000 tweets using all seven UDFs. For each UDF, we measured its throughput on a 6-node cluster and a 24-node cluster separately and computed the resulting speed-up. We repeated this computation for each UDF for three different batch sizes, namely 420 records/batch (1X), 1680 records/batch(4X), and 6720 records/batch (16X), and we show the speed-up of each batch size in Figure 25.

Since the UDFs in the Safety Rating, Religious Population, and Largest Religions use cases were relatively simple and their refresh period is already very low as shown in Figure 21, adding more resources yielded limited improvements to the execution times of their computing jobs. At the same time, their execution overhead grew as the cluster size increased. As a result, their speed-up is relatively poor. In Nearby Monuments, the Index Nested Loop Join didn't benefit from the batch size much as the overall index probing cost is not related to the batch size but mainly to the incoming data cardinality. In contrast, the other UDFs (Fuzzy Suspects, Suspicious Names, Tweet Context, and Worrisome Tweets) each improved with more resources. For Tweet Context in particular, not only were there 4x as many nodes participating in the computation, but the added resources (particularly memory) also allowed the join process to finish earlier. This enabled the system to obtain more than the ideal 4x speed-up. For a given volume of tweets, the bigger the batch size is, the fewer computing job invocations are needed for enriching these tweets, so the speed-up performance is better as the execution overhead increase from the cluster size growth is smaller.



(a) Throughput



(b) Speed-up

Figure 26: 100K tweets ingestion speed-ups

In order to see how ingestion performance improves when adding more resources, we also evaluated the speed-up behavior of the four most complex UDFs (Nearby Monuments, Suspicious Names, Tweet Context, and Worrisome Tweets). To avoid the use of index in Nearby Monuments becoming a performance bottleneck, we used a query hint to add one Naive Nearby Monument use case that enriches the tweets with the same information without using the index. We fed the new ingestion framework 100,000 tweets and varied the cluster size from 6 nodes up to 24 nodes to see how throughput changes with at batch size of 6720 records/batch. The experimental results are shown in Figure 26.

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

8. RELATED WORK

Data enrichment has been widely used in data analysis applications in which the collected data contains limited information and needs to be correlated with existing knowledge to revealing higher-level insights. Abel *et al.* proposed to construct Twitter user profiles by extracting semantics from tweets and relating them with collected news articles [1]. Moraru *et al.* introduced a framework for enriching sensor measurements with semantic concepts to generate new features [25]. In the Big Active Data project [19], notifications delivered to users can be enriched with other existing data in order to provide actionable notifications that are individualized per user; for example, emergency notifications could be enriched with shelter information to help affected users. Our work here is aimed at providing a scalable framework that users can employ to perform such data enrichment operations in the ingestion pipeline so that the enriched data can be used as soon as it is persisted.

User-defined functions have been a long standing feature of database systems [23, 33]. UDFs allow users to register their own functions with the database system for customized data processing and then invoke them in declarative queries. Hellerstein and Stonebraker designed a predicate migration algorithm for moving expensive functions in a query plan to minimize the total cost of a query [18]. Rheinländer *et al.* surveyed optimization techniques for optimizing complex dataflows with UDFs [31]. In our work, we use the UDF feature as a tool for users to use to express their data enrichment operations. Prepared queries are a mechanism that caches compiled plans to improve query performance. The predeployed jobs technique that we employed here for reducing the execution time of computing jobs was inspired by this technique.

The traditional ETL process defines a workflow including data collection, extraction, transformation, cleansing, and loading that is performed for moving data from an operational system into a data warehouse [11]. Data is extracted from an operational system, cleaned and transformed into a defined schema for analysis, and loaded into a periodically refreshed data warehouse for querying and data analysis. The refreshment process is often executed in an off-line mode with a relatively long period in order to minimize the burden on the operational systems [34]. Bruckner *et al.* proposed a near real-time architecture which minimizes the delay of new data being loaded into the data warehouse after being created in the operational system [8]. In our work, our focus was building an efficient and succinct framework aimed at ingesting and enriching data at the same time. Note that a user can achieve part of the ETL functionality by constructing appropriate UDFs, we do not consider the new ingestion framework to be a tool for solving general ETL problems. (Similarly, using a complex ETL suite for data enrichment would be overkill.) Our data feeds feature is related to the continuous data loading technique commonly used in near-real-time data warehouses [20].

The emerging category of **hybrid transactional/analyitical processing** (HTAP) aims to serving fast transactional data for analytical requests from large-scale real-time analytics applications. Özcan *et al.* recently reviewed emerging HTAP solutions and categorized HTAP systems based on different design options [28]. Some use the same engine to support both OLTP and OLAP requests [15, 30]. Other systems choose to couple two separate OLTP and OLAP systems to handle the different workloads separately. Wildfire [4], for example, provides the Wildfire Engine for ingesting fast transactional data, and integrates it with Spark for supporting analytical requests. HTAP systems, and similar data analytics services [5], focus on enabling data analytics on recent data. On the other hand, our system looks generally at improving data enrichment performance during the ingestion process so that later analytical queries can be evaluated more efficiently. The techniques that we used in this paper can be adapted to HTAP systems for accelerating their OLAP requests as well.

Streaming engines were introduced to address a need for stream data processing and real-time data analysis. They can handle streaming data sources and provide stream data processing on-the-fly. Many streaming engines also allow users to access reference data during processing. Kafka [22] uses "change data capture" in combination with its Connect API to access reference data in databases. Flink [9] supports registering external resources as Tables and offers a DataStream API to process the streaming data. Spark Streaming [37] uses Discretized Streams to discretize an incoming stream into Resilient Distributed Datasets and allow users to transform the data using normal Spark operations. Since streaming engines are designed for stream processing but not for complex data analysis queries, the processed results are often stored in connected data warehouses [24]. In this paper, we have focused on data enrichment use cases where the reference data may be frequently accessed and changed, and where the enriched data needs to be stored in a data warehouse for timely data analysis. We sought to minimize the effort from users so they can create a data ingestion pipeline easily, with declarative statements, and apply enrichment UDFs without limitations. To achieve these goals, we have chosen to build a new ingestion framework that supports the full power of SQL++ for data enrichment operations inside AsterixDB . The batch processing model that we chose is commonly used in streaming engines as well.

9. CONCLUSIONS

In this paper, we have investigated how to enrich incoming data during the data ingestion process. We discussed the challenges in data ingestion, presented possible computing models for evaluating stateful UDFs for data enrichment, and discussed the problems that may occur in different scenarios. We believe that an ingestion pipeline that supports efficient data ingestion and enrichment should be able to capture reference data changes during the ingestion process, maintain intermediate states properly, and support different enrichment operations with a full query language. To achieve these goals, we created a new ingestion framework with multiple optimization techniques. Its layered architecture allows the ingestion pipeline to better utilize the cluster resources. Repeatedly executing computing jobs in the framework allows incoming data to be enriched correctly, and predeployed jobs and partition holders improve the execution efficiency of computing jobs. We implemented the proposed framework in an open-source DBMS - Apache AsterixDB - and conducted a series of experiments to examine its performance with different workloads and various scales. The results showed that the new ingestion framework can indeed be scaled to support a variety of data enrichment workloads involving reference data and/or stateful operations. The techniques and designs illustrated in this paper could also be applied in other systems to accelerate their analytical requests based on enriched data.

10. ACKNOWLEDGMENTS

We would like to thank Chen Luo and Vassilis J. Tsotras for their feedback on this paper. The work reported in this paper was supported by the Donald Bren Foundation (via a Bren Chair) and the NSF CNS award 1305430.

11. REFERENCES

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

[2] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[19] S. Jacobs, M. Y. S. Uddin, M. J. Carey, V. Hristidis, V. J. Tsotras, N. Venkatasubramanian, Y. Wu, S. Safir, P. Kaul, X. Wang, M. A. Qader, and Y. Li. A BAD demonstration: Towards big active data. *PVLDB*, 10(12):1941–1944, 2017.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[35] X. Wang and M. J. Carey. An IDEA: An ingestion framework for data enrichment in AsterixDB. *arXiv preprint arXiv:1902.08271*, 2019.

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

[37] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: A unified engine for Big Data processing. *Commun. ACM*, 59(11):56–65, 2016.