Gobblin: Unifying Data Ingestion for Hadoop

Lin Qiao, Yinan Li, Sahil Takiar, Ziyang Liu, Narasimha Veeramreddy, Min Tu,
Ying Dai, Issac Buenrostro, Kapil Surlaker, Shirshanka Das, Chavdar Botev
LinkedIn Inc.
Mountain View, CA, USA
{lqiao,ynli,stakiar,ziliu,nveeramreddy,mitu,ydai,ibuenros,ksurlaker,sdas,cbotev}@linkedin.com

ABSTRACT

Data ingestion is an essential part of companies and organizations that collect and analyze large volumes of data. This paper describes Gobblin, a generic data ingestion framework for Hadoop and one of LinkedIn's latest open source products. At LinkedIn we need to ingest data from various sources such as relational stores, NoSQL stores, streaming systems, REST endpoints, filesystems, etc. into our Hadoop clusters. Maintaining independent pipelines for each source can lead to various operational problems. Gobblin aims to solve this issue by providing a centralized data ingestion framework that makes it easy to support ingesting data from a variety of sources.

Gobblin distinguishes itself from similar frameworks by focusing on three core principles: generality, extensibility, and operability. Gobblin supports a mixture of data sources out-of-the-box and can be easily extended for more. This enables an organization to use a single framework to handle different data ingestion needs, making it easy and inexpensive to operate. Moreover, with an end-to-end metrics collection and reporting module, Gobblin makes it simple and efficient to identify issues in production.

1. INTRODUCTION

A big data system is often referred to for the sheer volume of datasets it handles as well as the large processing power and new processing paradigms it is associated with. These big data challenges have fostered significant innovations on large scale computation platforms [2, 4, 5, 8, 14, 15, 19, 22, 24]. However, another important aspect of the complexity of a big data system comes from the coexistence of heterogenous data sources and sinks. In reality, data integration starts to cause big pain points a lot of times before developers or data engineers are able to solve traditional ETL or data processing at scale. It becomes more and more critical for a big data system to address the challenges of large data ingestion and integration with high velocity and quality.

Proceedings of the VLDB Endowment, Vol. 8, No. 12

With first hand experience on big data ingestion and integration pain points, we built Gobblin, a unified data ingestion framework. The development of Gobblin was mainly driven by the fact that LinkedIn's data sources have become increasingly heterogeneous. Data is constantly obtained and written into online data storage systems or streaming systems, including Espresso [23], Kafka [6], Voldemort [25], Oracle, MySQL, RocksDB [17], as well as external sites, including S3, Salesforce, Google Analytics, etc. Such data include member profiles, connections, posts and many other external and internal activities. These data sources crop daily terabytes worth of data, most of which needs to be loaded into our Hadoop clusters to feed business- or consumer-oriented analysis. We used to develop a separate data ingestion pipeline for each data source, and at one point we were running over a dozen different pipelines.

Having this many different data ingestion pipelines is like re-implementing the HashMap every time we need to use HashMap with a different type argument. Moreover, these pipelines were developed by several different teams. It is not hard to imagine the non-scalability of this approach, and the issues it brought in terms of maintenance, operability and data quality. Similar pains have been shared with us from engineers at other companies. Gobblin aims to eventually replace most or all of these ingestion pipelines with a generic data ingestion framework, which is easily configurable to ingest data from several different types of sources (covering a large number of real use cases), and easily extensible for new data sources and use cases.

The challenges Gobblin addresses are five-fold:

- **Source integration**: The framework provides out-of-the-box adaptors for all of our commonly accessed data sources such as MySQL, Kafka, Google Analytics, Salesforce and S3, etc.
- **Processing paradigm**: Gobblin supports both standalone and scalable platforms, including Hadoop and Yarn. Integration with Yarn provides the ability to run continuous ingestion in addition to scheduled batches.
- **Extensibility**: Data pipeline developers can integrate their own adaptors with the framework, and make it leverageable for other developers in the community.
- **Self-service**: Data pipeline developers can compose a data ingestion and transformation flow in a self-serviced manner, test locally using standalone mode and deploy the flow in production using scale-out mode without code change.

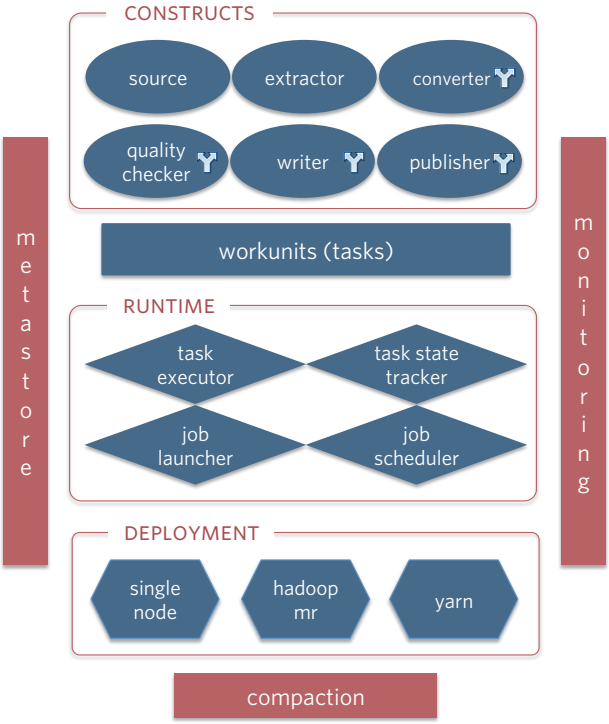

Figure 1: Gobblin Architecture

– **Data quality assurance**: The framework exposes data metrics collectors and data quality checkers as first class citizens which can be used to power continuous data validation.

Since we started the project in 2014, Gobblin has been launched in production and gradually replacing a number of ad-hoc data ingestion pipelines at LinkedIn. Gobblin was open sourced on Github as of February 2015. We aim to provide the community with a solid framework that meets the needs of many real world use cases, with desirable features that make the ingestion process as pleasant as eating a tasty data buffet.

2. ARCHITECTURE AND OVERVIEW

2.1 System Architecture and Components

The architecture of Gobblin is presented in Figure 1. A Gobblin *job* ingests data from a data source (e.g., Espresso) into a sink (e.g., HDFS). A job may consist of multiple *workunits*, or *tasks*, each of which represents a unit of work to be done, e.g., extracting data from an Espresso table partition or a Kafka topic partition.

2.1.1 Job Constructs

A job is formulated around a set of *constructs* (ellipses). Gobblin provides an interface for each of the constructs, thus a job can be customized by implementing these interfaces, or extending Gobblin's out-of-the-box implementations.

– A *Source* is responsible for partitioning the data ingestion work into a set of workunits and also specifying an Extractor per workunit. This resembles Hadoop's InputFormat, which partitions the input into Splits and specifies a RecordReader for each split. The algorithm for generating workunits should generally divide the work as evenly as possible to the workunits. In our implementations we mainly use hash based or bin packing based approaches.

– An *Extractor* does the work specified in a workunit, i.e., extracting certain data records. The workunit should have the information of where to pull data from (e.g., which Kafka partition, which DB table, etc.) as well as what portion of the data should be pulled. Gobblin uses a *watermark* object to tell an extractor what the start record (low watermark) and end record (high watermark) are. For example, for Kafka jobs the watermarks can be offsets of a partition, and for DB jobs the watermarks can be values of a column. A watermark can be of any type, not necessarily numeric, as long as there is a way for the Extractor to know where to start and where to finish.

– A *Converter* does data transformation on the fly while data is being pulled. There are often use cases that require such conversions, e.g., some fields in the schema need to be projected out for privacy reasons; or data pulled from the source are byte arrays or JSON objects and need to be converted to the desired output formats. A converter can convert one input record into zero or more records. Converters are pluggable and chainable, and it is straightforward to implement one, where all it takes is to implement two methods for converting schema and data records, respectively.

– A *Quality Checker* determines whether the extracted records are in good shape and can be published. There are two types of quality checking policies in terms of scope: record-level policies and task-level policies, which check the integrity of a single record and the entire output of a task, respectively. There are also two types of quality checking policies in terms of necessity: mandatory policies and optional policies. Violation of a mandatory policy will result in the record or task output being discarded or written to an outlier folder, while violation of an optional policy will result in warnings.

– A *Writer* writes records extracted by a task to the appropriate sinks. It first writes records that pass mandatory record-level policies to a staging directory. After the task successfully completes and all records of the task have been written to the staging directory, the writer moves them to a writer output directory, which are pending audit against task-level quality checking policies and are to be published by the publisher. Gobblin's data writer can be extended to publish data to different sinks such as HDFS, Kafka, S3, etc., or publish data in different formats such as Avro, Parquet, CSV, etc., or even publish data in different folder structures with different partitioning semantics. For example, at LinkedIn we publish many datasets in "hourly" and "daily" folders, which contain records with timestamps in that hour or day.

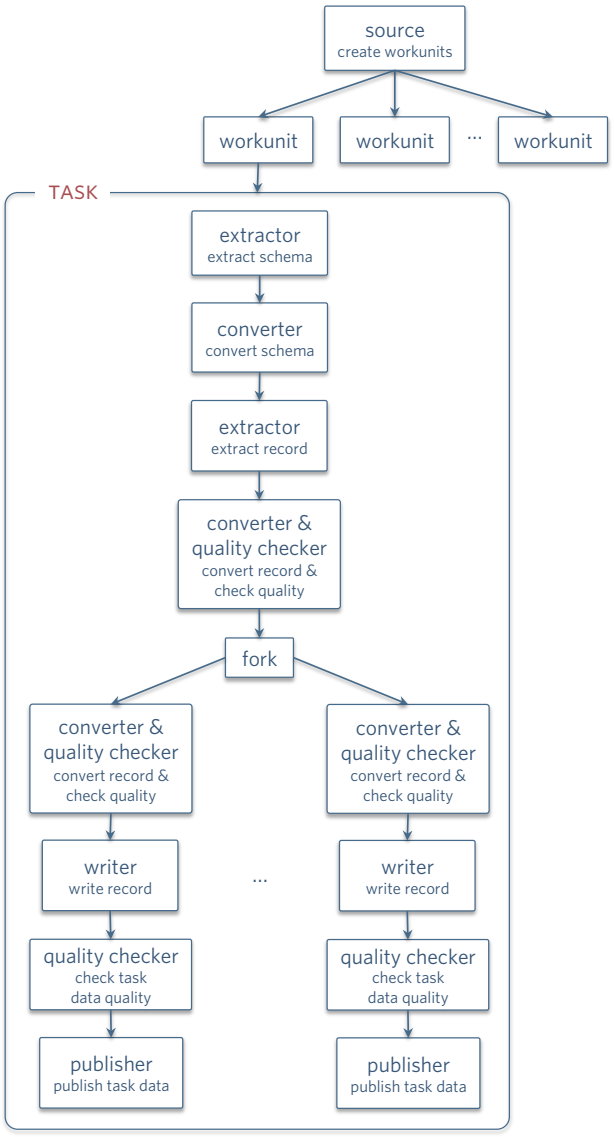

Figure 2: Flow of Gobblin Job Constructs in a Typical Job

– A *Publisher* publishes the data written by the Writers atomically to the final job output directory. Gobblin provides two commit policies: *commit-on-full-success* only commits data if all tasks succeeded, and *commit-on-partial-success* commits data for all succeeded tasks, while for each failed task, partial data will be committed if the task moved them from the staging directory to the output directory.

In addition to these constructs, Gobblin supports branching in a task flow through *fork operators* that allow an extracted record to be processed by different branches. Each fork may have its own converters, quality checkers, writers and publishers, and thus different forks may publish a record to different sinks, or to the same sink but in different formats. These constructs are annotated with a fork sign in Figure 1.

The flow of job constructs in a typical job is shown in Figure 2.

2.1.2 State Store

Gobblin employs a *metastore*, also known as *state store*, for managing job and task states across job runs. Upon completion of each run of a job, the high watermark along with other runtime metadata (e.g., latest data schema) of each task will be persisted into the state store. The next run of the same job will pick up the runtime metadata of the previous run from the state store and make the metadata available to the Source/Extractor so it can start from where the previous run left off.

Gobblin by default uses an implementation of the state store that serializes job and task states into Hadoop SequenceFiles, one per job run. Each job has a separate directory where the state store SequenceFiles of its runs are stored. Upon start, each run of a job reads the SequenceFile of the previous run in the corresponding directory to get its runtime metadata. To prevent the state store from growing indefinitely in size, Gobblin provides a utility for cleaning up the state store based on a configurable retention setting. Gobblin also allows users to plug-in their own state store implementations that conform to the StateStore interface through a configuration property.

2.1.3 Job Execution

Once a job is created, the Gobblin *job execution runtime* (diamonds) is responsible for running the job on the platform of a given deployment mode. Common tasks such as job/task scheduling, job/task state management, error handling and retries, monitoring and reporting, etc., are handled by the runtime.

For error handling, Gobblin employs multiple levels of defenses against job and task failures. For job failures, Gobblin keeps track of the number of times a job fails consecutively and optionally sends out an alert email if the number exceeds a defined threshold, so that the owner of the job can jump in and investigate the failures. For task failures, Gobblin retries failed tasks in a job run up to a configurable maximum number of times. In addition, Gobblin also provides an option to enable retries of workunits corresponding to failed tasks across job runs, so that if a task fails after all retries fail, the workunit based on which the task gets created will be automatically included in the next run of the job if this type of retries is enabled. This is useful in handling intermittent failures such as those due to temporary data source outage.

For job scheduling, Gobblin can be integrated with job schedulers such as Oozie [7], Azkaban [10], or Chronos [12]. Gobblin also ships with a built-in job scheduler backed by a Quartz [16] scheduler, which is the default job scheduler in standalone deployment. Gobblin decouples the job scheduler and the jobs, so that different jobs may run in different deployment settings.

2.1.4 Metrics and Monitoring

Gobblin features an end-to-end system for metric collection and reporting using a generic metric library for monitoring purpose. The library supports various types of metrics including counters, gauges, histograms, meters, etc., and is able to report metrics to multiple backend systems, e.g., Kafka, Graphite, JMX, and logs. It also performs automatic metric aggregations though a unique hierarchical metric collection structure in which a metric at a higher level is

automatically updated upon any updates to lower level metrics with the same name. This helps reduce the amount of processing needed in the metric backend without incurring much additional overhead.

2.1.5 Compaction

There is a *compaction* module in Gobblin for data compaction. It provides two out-of-the-box compactors, one using Hive and the other using MapReduce, both performing deduplications. For example, at LinkedIn we pull event data into hourly folders, and compact them once a day into daily folders with duplicates removed. Compactors using different logics (e.g., a compactor that simply merges hourly data without deduplication, or a compactor that always picks records with the latest timestamp in deduplication) can be directly plugged in.

2.1.6 Deployment

Gobblin is capable of running in several modes of *deployment* (hexagons) on different platforms for different scalability and resource usage requirements. Currently Gobblin can run in standalone mode on a single machine using a dedicated thread pool to run tasks, as well as in MapReduce mode on Hadoop (both Hadoop 1 and Hadoop 2 are supported) using mappers as containers to run tasks. This enables one to test a flow in standalone mode and deploy it in production using MapReduce mode without code change. A third deployment option currently under development is to run Gobblin on Yarn [18] for more flexible and efficient resource management and the ability to run as a long-running service. This comes in handy for data ingestion from streaming data sources like Kafka. Gobblin on Yarn is designed and built upon Apache Helix [20] and Apache ZooKeeper [21].

More specifically, it uses Helix to manage workunits running in a cluster of Yarn containers with helps from ZooKeeper on coordination and messaging between the Yarn ApplicationMaster and the containers. ZooKeeper is also used as a central repository for metadata of running workunits. Helix, as a framework for managing distributed resources in a cluster, guarantees that workunits are evenly distributed among the available containers and is also responsible for rebalancing in case of container failures or shutdown. One important feature of Gobblin on Yarn that helps achieve more efficient resource management is the ability to adapt to changing workloads and resize the cluster of Yarn containers dynamically at runtime.

2.2 Gobblin in Production

Gobblin has been deployed in production at LinkedIn for over six months, and is currently responsible for ingesting over 300 datasets from 12 unique sources such as Salesforce and Google Analytics. These sources are of several different types including MySQL, SQL Server, SFTP Servers, S3, and REST endpoints. More and more datasets are being added. Adding a new dataset to be ingested is an easy task: our engineers just need to provide a simple configuration file in order to get their data ingested into HDFS. Our ETL and Solutions engineers no longer need to worry about data ingestion and freshness problems, and can instead solely focus on data usage.

In addition, Gobblin is also utilized for purging PII data from HDFS in order to meet data compliance requirements (e.g. delete a member's data within a certain number of days of account closure). This can be considered an HDFS to HDFS ingestion job with appropriate converters for filtering.

As Gobblin continues to mature, LinkedIn plans to run many more data ingestion jobs using Gobblin.

3. CASE STUDY

In this section we share some details on how Gobblin is used to extract data from Kafka and JDBC sources at LinkedIn, through which we hope to explain by examples how to extend Gobblin for different use cases.

3.1 Kafka to HDFS Ingestion

At LinkedIn we ingest more than 3000 topics with tens of thousands of partitions constantly from Kafka to Hadoop clusters. These topics include member activities (such as page views and sending InMails), metrics, as well as a variety of internal events that need to be tracked. Kafka ingestion is currently handled by Camus [11], an open source pipeline specifically designed for Kafka-HDFS ingestion. Gobblin's Kafka adapter is replacing Camus at LinkedIn for better performance, stableness, operability and data integrity.

Gobblin currently ingests Kafka records in small, continuous batches. Once the integration with Yarn is complete, we will be able to run Kafka ingestion in long-running, streaming mode. Next we explain some details about how each construct in the Kafka adapter is designed and implemented.

Source. Recall that Source is responsible for generating workunits. To ingest from Kafka, one needs to implement a Source that generates Kafka workunits. A Kafka workunit should contain for each partition the start and end offsets (i.e., low watermark and high watermark) to be pulled.

For each topic partition that should be ingested, we first retrieve the last offset pulled by the previous run, which should be the start offset of the current run. We also retrieve the earliest and latest offsets currently available from the Kafka cluster and verifies that the start offset is between the earliest and the latest offsets. The latest available offset is the end offset to be pulled by the current workunit. Since new records are constantly published to Kafka and old records are deleted based on retention settings, the earliest and latest offsets of a partition changes constantly. This adds pressure to the data ingestion pipeline, since its speed must be faster than the speed records are published to Kafka.

For each partition, after the start and end offsets are determined, an initial workunit is created. After obtaining all initial workunits, one for each partition, two steps are performed using bin-packing based algorithms: (1) Merge some workunits corresponding to partitions of the same topic into bigger workunits, so that these partitions can write to the same output file, which reduces the number of small files published on HDFS and thus reduces pressure on the NameNode; (2) Assign workunits to containers and balance the workload of the containers.

Extractor. As explained above, a Kafka workunit has one or more partitions of the same topic. An Extractor is created for each workunit which pulls partitions one by one, and if a partition cannot be pulled, it skips the partition (and reports the failure in the metrics) but does not fail the task, so that different partitions are isolated and do not affect each other.

After all partitions are pulled, the Extractor reports the last offset pulled for each partition to a state object which

will be persisted to the state store by Gobblin runtime.

Converter. LinkedIn's Kafka clusters mainly store records in Avro format. Gobblin currently has a few Avro converters that can be used for ingestion from Kafka and any other source with Avro data, including a converter for filtering Avro records based on field values, a converter for removing fields from Avro records, a converter for converting Avro records to Json objects, etc. The most often used one at LinkedIn is the one for removing fields, since certain sensitive fields need to be removed from Avro records before they can be published.

Quality Checker. Since we ingest Kafka data into time partitioned (i.e., hourly and daily) folders, we use a mandatory record-level quality policy which checks whether each record has a timestamp field. If so, it can be written to the appropriate folder; otherwise it has to be discarded or written to an outlier folder. We also use another policy to verify whether the primary key field of a record exists and is non-null.

Writer and Publisher. For Kafka ingestion we implement a time partitioned writer and a time partitioned publisher, which write and publish each record to the appropriate time partitioned folder based on its timestamp.

3.2 JDBC to HDFS Ingestion

The JDBC protocol offers a clean abstraction for using Java code to query and manipulate a variety of relational data stores. Today, LinkedIn pulls about 250 tables from both MySQL and SQL Server.

An important difference between JDBC ingestion and Kafka ingestion is that database tuples are mutable while Kafka records are not. Therefore, JDBC ingestion not only needs to pull new tuples but also needs a way to handle updated tuples. We will explain how this is done after discussing the constructs for JDBC ingestion in this subsection.

Source. Gobblin provides two Sources for JDBC: a MySQL Source and a SQL Server Source. They use the same algorithm for generating workunits, and the only difference is that the MySQL Source uses the MySQL Extractor and the SQL Server Source uses the SQL Server Extractor.

To generate workunits, similar as the Kafka Source, for each table that should be pulled, we retrieve the latest timestamp of a tuple pulled by the previous run and use it as the low watermark; the current time is used as the high watermark. We then divide the interval from low watermark to high watermark into a configurable number of partitions of equal length. A workunit is created for each partition.

Extractor. The MySQL and SQL Server Extractors get the tuples with the specified timestamp values from the databases using SQL queries, and return them as JSON elements. Similar as the Kafka Extractor, they report the latest timestamp pulled, which will be persisted to the state store and available for the next run.

Converter and Quality Checker. We store ingested data in HDFS in Avro format. Therefore, we use a converter to convert JSON elements into Avro records. We also use a task-level quality checking policy to ensure that the number of records written is the same as the number of rows returned by a count query.

Writer and Publisher. Unlike Kafka ingestion, we do not write database tuples into time partitioned folders for JDBC ingestion, since users of the data often need to access an entire table rather than tuples in a specific hour or day. We use a simple writer and publisher which publish the records of each table in a single folder.

Handling Tuple Updates. As mentioned above, in JDBC ingestion we need a way to handle tuple updates. A trivial method is to pull each table in its entirety in each run, which is hardly feasible for large tables and inefficient for tables with only a small portion of updated tuples between two runs.

We use an improved approach for such tables which involves multiple job types for JDBC ingestion. A *snapshot* job, which pulls an entire table, is scheduled at a relatively low frequency. An *append* job only pulls the tuples whose modification timestamps are later than the latest timestamp pulled by the previous run (which may be either a snapshot job or an append job). Append jobs are scheduled at higher frequencies. Note that this requires the table has a modification timestamp column. Also note that different versions of a tuple may exist in the extracted snapshot and append data files.

Users may directly consume the extracted snapshot and append files if their applications can live with the existence of multiple versions of a tuple, or if they have a way to pick the latest version of a tuple if multiple versions exist. Otherwise, Gobblin has a compaction module introduced in Section 2.1.5 which can be used to compact snapshot and append files. Alternatively, users can run a *snapshot-append* job which runs an append job followed by compaction.

Apache Sqoop [9] is another data ingestion framework whose focus is SQL to Hadoop data ingestion. Compared to Sqoop, in Gobblin we aim to provide the right granularity of operators, which can be easily composed into an ingestion flow while creating the maximum leverage of the building blocks, as well as easily extended for different use cases. Converter, quality checker and fork operator are a few examples of such operators, which are not currently available in Sqoop. Besides, as explained, Gobblin supports multiple types of ingestions as well as compaction. While Sqoop supports incremental ingestion, it does not have something comparable to snapshot-append jobs, nor does it support compaction.

4. RELATED WORK

The most comparable system to Gobblin is Apache Sqoop [9], which is originally designed to be a SQL-Hadoop ingestion tool and has since become a generic framework for multiple types of sources and destinations. Next we summarize the main differences between Gobblin and Sqoop in a few categories.

- In terms of goals, one of the goals of Gobblin from the beginning was the unification of continuous streaming ingestion with scheduled batch ingestion. We believe that the two modes of ingestion do not need completely different, specialized engines and instead they can be handled by a single framework with an appropriate level of abstraction on how data ingestion workunits are defined. This is one of the big differentiators with Sqoop which is architected to be a batch oriented ingestion solution. The streaming capability of Gobblin based on Yarn is being developed and will be available in the coming months.

- In terms of modularity and componentization, Sqoop does not provide some of the modules in Gobblin as

building blocks for data ingestion jobs, such as Converter, quality checker and fork operator, as explained in Section 3.2.

- In terms of data semantics, Gobblin supports multiple types of ingestions as well as compaction, as mentioned in Section 3.2. These are not supported in Sqoop.
- In terms of job/task state management and failure handling, a state store is maintained internally in Gobblin for managing job states as explained in Section 2.1.2. Gobblin uses watermarks for managing incremental pulls, which can be both simple and complex types. By contrast, Sqoop provides a less powerful feature for incremental import, which is based on a single numeric value on a single attribute. To the best of our knowledge Sqoop also doesn't provide an easy way to maintain such checkpoints and other job states.
- In terms of operability, Gobblin has a complete end-to-end metrics collection and reporting system explained in Section 2.1.4. This is to our knowledge not available in other data ingestion frameworks including Sqoop.

There are also a few specialized open-source tools for data ingestion, such as Apache Flume [3] for ingesting log data, Aegisthus [1] for extracting data from Cassandra, and Morphlines [13] which is a data transformation tool similar as Gobblin's converter, and which can be plugged into data ingestion frameworks such as Sqoop and Flume. In comparison, Gobblin is designed to be far more general and extensible which enables an organization to use a single framework for different types of data ingestions.

5. FUTURE WORK

Gobblin is under active development. We are extending Gobblin in the future along the following directions:

- To achieve the goal of unifying processing paradigm between batch and streaming ingestion flows, we are building a new execution framework based on Yarn natively. The new execution framework will deliver much better SLA, load-balancing and elasticity.
- Gobblin currently uses record-level abstraction in its execution flow, i.e., the unit to be pulled is a record. We will extend Gobblin beyond record-level abstraction, and add file-level abstraction to Gobblin, which will enable Gobblin to be used as a high-performance file transfer service across various file systems, including NFS, HDFS, S3, etc. as well as across data centers or regions.
- We are planning to expand the integration points with workflow schedulers for Gobblin jobs beyond native Quartz scheduler and Azkaban. For example, Oozie and Chronos are on top of our list.

6. CONCLUSION

We presented Gobblin, a data ingestion framework for Hadoop. Gobblin is designed to be generic and extensible so that it can be used for a variety of data ingestion use cases, as well as easily operable and monitorable. It is currently used to ingest from a number of data sources at LinkedIn into Hadoop, and is slated to take over several specific-purpose data ingestion pipelines currently being used as it further matures.

7. ACKNOWLEDGEMENT

We'd like to extend our appreciation to our partner teams for their strong support and valuable help during the development and deployment of Gobblin at LinkedIn. They are Data Services Operation team (Neil Pinto, Sandhya Ramu, Tu Tran, Teja Thotapalli, Vamshi Hardageri and Dhawal Chohan), Hadoop Operation team (Bob Liu, Anil Alluri, Adam Faris and Chen Qiang), BI and Solution team (Daisy Lu, Qun Li, Ranjith Prabu and Subbu Sankar). We also thank our alumni, Henry Cai and Ken Goodhope, for their significant contribution to Gobblin.

8. REFERENCES

[1] Aegisthus. https://github.com/Netflix/aegisthus.
[2] Apache Flink. https://flink.apache.org/.
[3] Apache Flume. https://flume.apache.org/.
[4] Apache Giraph. http://giraph.apache.org/.
[5] Apache Hadoop. https://hadoop.apache.org/.
[6] Apache Kafka. http://kafka.apache.org/.
[7] Apache Oozie. http://oozie.apache.org/.
[8] Apache Spark. https://spark.apache.org/.
[9] Apache Sqoop. http://sqoop.apache.org/.
[10] Azkaban: Open-source Workflow Manager. http://azkaban.github.io/.
[11] Camus. https://github.com/linkedin/camus.
[12] Chronos. http://nerds.airbnb.com/introducing-chronos/.
[13] Morphlines. http://cloudera.github.io/cdk/docs/current/cdk-morphlines/index.html.
[14] Pinot. https://github.com/linkedin/pinot.
[15] Presto. https://prestodb.io/.
[16] Quartz Scheduler. http://quartz-scheduler.org/.
[17] RocksDB. http://rocksdb.org/.
[18] YARN. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.
[19] S. Chen. Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce. *PVLDB*, 3(2):1459–1468, 2010.
[20] K. Gopalakrishna, S. Lu, Z. Zhang, A. Silberstein, K. Surlaker, R. Subramonian, and B. Schulman. Untangling Cluster Management with Helix. In *SoCC*, 2012.
[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX*, 2010.
[22] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*, 2015.
[23] L. Qiao, K. Surlaker, S. Das, T. Quiggle, B. Schulman, B. Ghosh, A. Curtis, O. Seeliger, Z. Zhang, A. Auradkar, C. Beaver, G. Brandt, M. Gandhi, K. Gopalakrishna, W. Ip, S. Jagadish, S. Lu, A. Pachev, A. Ramesh, A. Sebastian, R. Shanbhag, S. Subramaniam, Y. Sun, S. Topiwala, C. Tran, J. Westerman, and D. Zhang. On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform. In *SIGMOD Conference*, pages 1135–1146, 2013.
[24] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *ICDE*, pages 60–69, 2008.
[25] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving Large-scale Batch Computed Data with Project Voldemort. In *FAST*, 2012.