Towards Real-Time Analysis of ID-Associated Data

Guodong Jin[1,2], Yixuan Wang[1,2], Xiongpai Qin[1,2(✉)], Yueguo Chen[1,2], and Xiaoyong Du[1,2]

[1] School of Information, Renmin University of China, Beijing, China
qxp1990@ruc.edu.cn
[2] DEKE Key Laboratory, Renmin University of China, MOE, Beijing, China

Abstract. ID-associated data are sequences of entries, and each entry is semantically associated with a unique ID. Examples are user IDs in user behaviour logs of mobile applications and device IDs in sensor records of self-driving cars. Nowadays, many big data applications generate such types of ID-associated data at high speed, and most queries over them are ID-centric (on specific IDs and ranges of time). To generate valuable insights from such data timely, the system needs to ingest high volumes of them with low latency, and support real-time analysis over them efficiently. In this paper, we introduce a system prototype designed for this goal. The system designed a parallel ingestion pipeline and a lightweight indexing scheme for the fast ingestion and efficient analysis. Besides, a fiber partitioning method is utilized to achieve dynamic scalability. For better integration with Hadoop ecosystem, the prototype is implemented based on open source projects, including Kafka and Presto.

Keywords: Real-time analytics · ID-associated data
Real-time ingestion

1 Introduction

ID-associated data are sequences of entries, and each entry is semantically associated with a unique ID. Examples are user IDs in user behaviour logs of mobile applications and device IDs in sensor records of self-driving cars. Nowadays, ID-associated data are ubiquitous in many big data applications. They are generated at high speed, and real-time analysis of them is critical to gain valuable business insights timely. Take user behaviour analysis in mobile applications as an example, each time a user clicks inside a mobile application, a log entry with the user ID recording the user's behaviours is generated automatically and collected by the vendor. The vendor applies real-time analysis over newly collected

This work is supported by Science and Technology Planning Project of Guangdong under grant No.2015B010131015, 863 key project under grant No.2015AA015307, and the National Science Foundation of China under grants No.61472426, U1711261, 61432006.

© Springer Nature Switzerland AG 2018
C. Woo et al. (Eds.): ER 2018 Workshops, LNCS 11158, pp. 26–30, 2018.
https://doi.org/10.1007/978-3-030-01391-2_6

data to identify abnormal user accesses, and study behaviours of a particular group of users over the latest days for better content recommendations.

In many such cases, ID-associated data come at high speed. Typically, most queries over ID-associated data are ID-centric – they retrieve and analyze data of a specified group of IDs over a period of time. To gain valuable insights timely, the system needs to ingest high volumes of data quickly and execute ID-centric queries efficiently. ElasticSearch [5], the distributed search engine, builds inverted indices over data to support real-time searches. However, due to the high latency of indexing, it cannot support data ingestion in real time. SQL-on-Hadoop systems like Spark SQL [2] perform well at batch processing, but they lack the ability of data ingestion. Hive [1] is proven to be a powerful tool for ETL(extract, transform and load) on HDFS. It converts the ETL pipeline into a batch of map reduce jobs, which is slow due to frequent reads and writes of intermediate files.

In this paper, we introduce a system prototype tailored for the real-time analysis of ID-associated data. It integrates ingestion of ID-associated data with ID-centric relational processing. To support this, we designed a parallel ingestion pipeline and a lightweight indexing scheme. The pipeline offers data ingestion with low latency, and a well-designed in-memory columnar store, enabling efficient relational processing over data being loaded. And the indexing scheme helps the system avoid a full scan of the relational table. Combined with the fiber partitioning mechanism, the system provides dynamic scalability. For better integration with Hadoop ecosystem, the prototype is implemented based on an open source messaging system (Apache Kafka [7]), and a popular SQL-on-Hadoop engine (Facebook Presto [8]).

2 System Architecture

As shown in Fig. 1, our system prototype contains six modules (in gray): **Collector**, **Loader**, **Indexer**, **Metadata**, **Connector** and **Coordinator**.

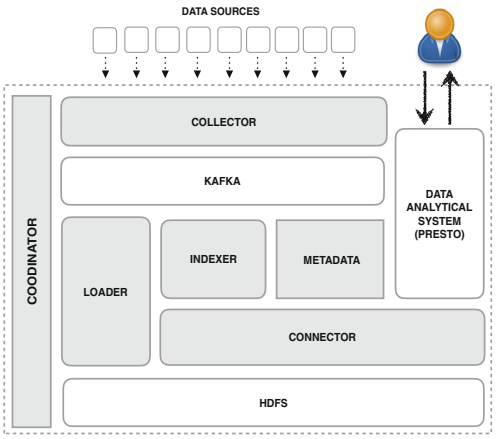

Fig. 1. System architecture

Collector. *Collector* servers as the front-end of the system, which is responsible for collecting data from various sources. Data collected are applied to our fiber partitioning method (details are discussed in Sect. 3), and sent into Kafka with a one-to-one correspondence between fibers and Kafka partitions. Kafka assigns each record with a unique offset to allow re-read of data from a specified offset. We make checkpoints on offsets of loaded data to guarantee data lossless during ingestion.

Loader and Indexer. *Loader* ingests data from fibers assigned by *Coordinator*. It pulls data from Kafka and writes them into HDFS after processing. Generally, data on HDFS are stored in columnar formats such as ORC [3] and Parquet [4] to speed up queries. Data pulled from Kafka are applied with user-defined filters and transformers. Then, data are sorted by their associated IDs and generation time, and appended into a *memory store* (details are discussed in Sect. 3). Finally, the *memory store* flushes data into HDFS as columnar files. In particular, for data in the *memory store*, *Indexer* maintains bloom filters to track existing IDs and records key statistics, such as min and max values of each column (including generation time). When data are flushed into HDFS, these lightweight indices are embedded into files. To efficiently load and index data, a parallel data ingestion pipeline is designed, which is described in Sect. 3.

Metadata, Connector and Coordinator. *Metadata* maintains critical metadata for the system, including definitions of relational tables and columns, user-defined functions, and fiber storage information. When executing queries, reading tasks are pushed down to *Connector* to fetch data needed by query engines. *Connector* is able to fetch data from the *memory store* and HDFS altogether, and make use of existing indices to filter useless files. Thus, analytical systems can be boosted to run queries on both of the most recent data in memory and historical ones on disk. *Coodinator* monitors all running processes in the cluster, and coordinates executions of *Loaders* by dynamically assigning fibers to achieve good scalability and load balance.

3 Key Techniques

Fiber Partitioning. Our fiber partitioning scheme is based on the consistent hashing [6]. In ID-associated data, each entry is associated with a unique ID, and a hash function is applied to get a hash value of the ID. The range of hash values is split into k intervals, mapping to k fibers. Due to the possibly uneven distribution of data, some fibers may contain much data while some may comprise few. During data ingestion, *Coordinator* collects load metrics from *Loaders*. When loads are seriously skewed, *Coordinator* greedily migrates fibers from the node with the heaviest loads to other ones. Further, fibers can be tuned in fine granularity to merge and split dynamically. Based on this partitioning scheme, data are further clustered by their associated IDs and indexed efficiently.

Parallel Ingestion Pipeline. Data pulled from Kafka are maintained as fiber streams separately, and in each stream data are processed with user-defined

filters and transformers inside a thread. After filtering and transformation, data are inserted into a *sorted buffer*. Inside the buffer, entries are ordered by their associated IDs and generation time. Once the sorted buffer reaches its max size or user-defined lifetime (elapsed time since the last reset), it appends all data into the *memory store* and resets. The lifetime represents the latency of data ingestion. Data inside the *memory store* is organized as immutable segments and ready to be queried by analytical systems. When the memory store reaches its threshold in size, it flushes segments as columnar files into HDFS. To gain benefits from sequential disk I/O and avoid I/O competitions, a single writer thread is utilized to handle all writing requests.

In-Memory Columnar Storage. *Memory store* organizes data as segments, inside which data are clustered by their associated IDs, and stored in a columnar format to be optimized for cache lines. To reduce memory footprints, lightweight compression schemes such as run length encoding, bit packing, and dictionary encoding are utilized to compress segments. With little overhead, queries can run directly on these lightweight compressed segments. Memory store maintains storage information of each segment, including the storage level (i.e., on-heap, off-heap, on-disk) and location (i.e., object reference, memory address, file path). In-memory segments can be stored on-heap and off-heap based on user configurations. Off-heap storage is recommended by default to avoid GC overheads.

4 Demonstration

The demonstration is set up on a cluster, in which our *Collectors* are deployed along with Kafka, and *Loaders* are distributed with Hadoop and Presto. We design a data generator to generate records based on the result of joining *lineitem* and *orders* table from the TPC-H benchmark. Besides, we add an attribute, called *generation_time*, to identify the generation time of each record. The generator runs in a streaming manner – records are generated and sent to *Collectors* one by one in real time. During the demonstration, a web interface is presented with detailed information of data ingestion, such as throughput of ingestion and loads of each node. Also, users can choose queries from our provided query set or compose some in our web interface. The interface has a query client embedded to issue queries to the Presto and collect results through JDBC. Our provided query set consists of ID-centric queries, which are based on TPC-H queries with time range conditions and designated IDs. The example query[1] analyzes quantity, price and discount of orders made by customer *k* from *t1* to *t2*. Once a query is submitted, we can track its execution in our web interface. Our system prototype is open source on the Github[2].

[1] select sum(quantity), sum(totalprice), min(discount), max(discount), avg(extended price), count(*) from test where custkey = k and *generation_time* > $t1$ and *generation_time* < $t2$ order by linestatus.

[2] https://github.com/dbiir/paraflow.

References

1. Apache Hive (2011). http://hive.apache.org
2. Spark SQL: relational data processing in spark (2015). http://spark.apache.org/sql/
3. Apache ORC (2018). https://orc.apache.org
4. Apache Parquet (2018). https://parquet.apache.org
5. Gormley, C., Tong, Z.: ElasticSearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine. O'Reilly Media Inc, Sebastopol (2015)
6. Karger, D., et al.: Web caching with consistent hashing. Comput. Netw. **31**(11–16), 1203–1213 (1999)
7. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: a distributed messaging system for log processing. In: Proceedings of the NetDB, pp. 1–7 (2011)
8. Traverso, M.: Presto: interacting with petabytes of data at facebook (2013). Accessed 4 Feb 2014