CPSC 5660 - Research Project (Spring 2023)

Annotated Bibliography

Naveen Kumar Vadlamudi
Department of Mathematics & Computer Science
University of Lethbridge

24 March 2023

References

[1] S. Babu and J. Widom, "Continuous queries over data streams," *ACM Sigmod Record*, vol. 30, no. 3, pp. 109–120, 2001.

Problem statement: Streaming processing is one of the core sub domains in database community, where efficient handling and processing of it needs novel algorithms and frameworks to tackle the incoming data stream into system. So, the authors Jennifer Widom and Shivnath babu, were trying to address the problem of stream query processing by comprehending and correlating traditional RDBMS mechanisms like triggers and materialized views in the context of stream Processing. Which further led to the creation of a novel architecture.

Limitations of existing work: According to the authors, many computer scientists have proposed multiple variants of stream processing techniques, such as Xpath and Xyleme, which have their own query language and limited scope of working for their dedicated file format. Other variants like OpenCQ monitor continuous queries based on incremental view maintenance, and NiagaraCQ addresses scalability issues for grouping multiple queries. However, none of these techniques utilized triggers, materialized views, or specialized architecture in the context of stream processing and never leveraged their benefits.

Summary of proposed work: The authors defined streaming query (Q) as an entity that is submitted once and queried continuously. They proposed three specific scenarios for a network model problem as a concrete example that elaborates more on how the queries are processed internally in the context of stream processing. Then, they proposed an architecture consisting of four components (Stream, Store, Scratch, and Throw) and six steps to ensure that data objects queried are always consistent. In the first step, incoming tuples are sent to the stream component, while tuples not anticipated to be present in results are sent to the store area to make sure it only contains tuples not going to be included in the results. Similarly, the scratch component physically stores the tuples in memory to provide results if queried again. The last component, Throw, collects useless and garbage-collected tuples, reducing space over the system.

Summary of methodology and result of evaluation: The authors did not conduct any performance tests on the proposed architecture but discussed the methodology involved, such as using triggers and materialized views for storing and processing the incoming stream. The trigger is invoked on incoming data, and sections such as stream and store may remain empty, while scratch is used as a lookup to evaluate conditions. Similarly, for materialized views, the view itself is stored in the store section, and data not available in conventional tables are stored in scratch.

Research directions: The authors provided multiple research directions in the area of stream processing, where they evaluated many existing ideas of RDBMS techniques and proposed a novel architecture that optimizes the processing and handling of streaming data.

[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.

The Problem Being Addressed: The rise of big data has led to various innovations in the data domain, including the creation of novel tools such as Apache Spark and Hadoop that process huge amounts of data. Therefore, in their work, authors Carbone et al. document the architecture and functionalities of Apache Flink, an open-source project. They discuss the unified architecture of batch and stream processing and explain its fault-tolerant streaming data flows by detailing how they built fully-fledged stream and batch processing engines. They also mention specific optimizations that fit in the context of static and stream datasets.

Limitations of Existing Work: According to the authors, existing systems like Hadoop and Spark are extremely useful in the context of batch processing but lack the ability to efficiently process stream-based data models with low latency, as they rely on disk-based access methods. Also, the aforementioned systems lack support for event time processing, which is critical for evaluating time-sensitive data accurately.

Summary of Proposed Strategy: The authors proposed that Apache Flink is a unified processing engine that combines the strengths of both stream and batch processing systems. It uses a distributed dataflow model (a model that compiles a given program to a Directed Acyclic Graph that consists of all the necessary operators for query processing) to process data and has two core APIs. A finite dataset is referred to as the Dataset API (Batch processing), and unbounded data streams are named the DataStream API (Stream Processing). Similarly, every job, when submitted, is converted into a Dataflow Graph which is a DAG (Directed Acyclic Graph) that consists of tasks and subtasks that later perform all the compute operations in parallel. In a similar way, the data exchange in a data stream occurs in a distinct way, where for streaming data, Flink uses pipelined streams to avoid materialization. For batch processing, the blocking streams buffer all the data from the producing operator and give it to the consumer for consumption and are frequently checkpointed for maintaining fault tolerance using a technique named ABS (Asynchronous Barrier Snapshotting). A barrier is introduced, which are usually control records with a logical timestamp that performs periodic asynchronous data backup.

Summary of Methodology and Result of Evaluation: The authors tested a Flink cluster on latency and throughput parameters and mentioned that the system attained high throughput by configuring the buffers to a larger value and latency timeout to a minimal value. With the above parameters on 30 machines (120 cores), the system performed well, recording 80 million events per second with a 99-percentile latency of 50ms. Similarly, when the latency was 20ms, it recorded 1.5 million events per second.

Any Research Directions Provided The authors provide multiple research directions, including novel query processing methodologies that utilize the dataflow paradigm, distributed computing techniques for fault tolerance, and iterative processing for machine learning in stream and batch-based processing engines.

[3] L. Deri, S. Mainardi, and F. Fusco, "tsdb: A compressed database for time series," in *Traffic Monitoring and Analysis: 4th International Workshop, TMA 2012, Vienna, Austria, March 12, 2012. Proceedings 4.* Springer, 2012, pp. 143–156.

The problem being addressed: Luca Deri et al. addressed the challenge of efficiently storing and retrieving time series data in network monitoring by examining existing solutions. This led to the creation of a novel time series database that is highly performant and efficient, capable of accommodating multiple metrics in parallel without compromising performance and compression.

Limitations of Existing Work: According to the authors, relational databases are not required in network monitoring applications as the data is not characterized by many relationships. Instead, the records are repeated over time at frequent intervals, increasing the cardinality of a table and making handling indexes extremely hard. Specialized databases like Round Robin Database (RRD) have been developed to handle time series data, where each RRD has a file with a size of 64 KB. However, RRD has performance limitations such as the inability to handle a large number of concurrent RRDs.

Summary of Proposed Strategy: The authors considered all the features and functionalities supported by existing RDBMS and RRDs, and drafted some core principles for their time series database (TSDB) that differentiate it from existing RRDs. These principles include (a) capacitating millions of time series using minimal append operations, (b) adding and removing time series as needed without reconfiguring data, (c) avoiding data consolidation during append, similar to RRD database, and (d) providing compressed data storage for all time series using a single file. To efficiently store and retrieve time series, the database designers implemented tables in TSDB as a series that shares the same time across the database. This method is immune to time manipulation issues and also aids in storing data effectively back to disk as a key-value pair. The index, which is maintained on a particular table, is preserved and reapplied when it's re-enabled after being dropped at a certain point in time. To compress data, the authors used the lossless QuickLZ algorithm, where the records are partitioned into 10k records per set and stored as X-Y format, where X is epoch and Y is chunk id.

Summary of Methodology and Results of Evaluation: The authors created TSDB to monitor the Italian DNS (Domain Name System) registry and tested the newly designed system with existing solutions, which used Redis, MySQL, and RRDtool as test variables. The performance of RRD started at 1600 updates/sec with small databases and decreased to 32 updates/sec after a peak concurrent load. Similarly, Redis's performance was acceptable but had some limitations, mainly regarding volatile memory, where when it was full, the write performance decreased significantly. MySQL performed better than Redis but was still inferior to TSDB, which outperformed every other existing solution with high throughput and minimal latency.

Any Research Directions Provided: The authors provided multiple research directions in the area of time series data management, lossless compression, and effective data retrieval mechanisms in the context of concurrent executions.

[4] Y. Fu and C. Soman, "Real-time data infrastructure at uber," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2503–2516.

Problem Being Addressed: In this paper, the authors Soman et al. discuss the challenges faced by Uber in managing and processing large volumes of real-time data generated by billions of events through customer activity. The lack of a unified, scalable, and fault-tolerant platform to process and consume real-time data for various applications, services, and stakeholders is the primary concern. The authors present the architecture, design, and implementation of the real-time data infrastructure (RTDI) that Uber developed to address these challenges instead of using existing off-the-shelf solutions.

Limitations of Existing Work: The authors identify several limitations of current real-time data processing platforms, including batch processing, limited scalability, complex deployment, and high latency. They argue that these limitations make it difficult to serve the diverse needs of real-time data processing, such as real-time analytics, machine learning, and stream processing. Therefore, the existing solutions could not handle the massive scale and complexity of Uber's data, which requires processing multiple billions of events per hour and trillions in a day.

Summary of Proposed Strategy: The authors present the customized architecture of RTDI, which consists of several components, including data ingestion, processing, storage, serving, and monitoring. They describe how each component works together to provide a unified and fault-tolerant platform for processing and serving real-time data. The key characteristics of RTDI include high scalability, low latency, and high availability, allowing Uber to process and serve real-time data for diverse applications and services. To achieve this, Uber uses Apache Flink for processing incoming data, Kafka for processing logs, HDFS for building data Lake, and Presto for querying data from Pinot tables.

Summary of Methodology and Results of Evaluation: As a technical paper focused on the architecture and design of RTDI, the authors did not provide any experiments or results. However, they mentioned several challenges they faced during the development and deployment of RTDI, including data consistency, network

latency, and hardware failures. They describe how they addressed these challenges using techniques such as replication, caching, and sharding.

Any Research Directions Provided: The authors suggest several research avenues for real-time data processing, including enhancing query latency, data partitioning optimization, and increasing fault tolerance. They also advise trying out fresh apps for real-time data processing, including fraud detection, real-time decision-making, and predictive analytics. The study contends that the integration of deep learning with machine learning can improve processing, discovering algorithms. Lastly, the authors recommend that future studies should concentrate on creating real-time data processing platforms that are more effective and scalable, capable of managing data sets that are considerably bigger and more complex.

[5] J. Kreps, N. Narkhede, J. Rao, *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, no. 2011. Athens, Greece, 2011, pp. 1–7.

The Problem Being Addressed: Log data plays a crucial role in technology companies where they collect, process, and analyze patterns of data from accrued logs. Therefore, handling and processing log data is a pivotal part of every technological organization. In addressing the problem of handling and processing large volumes of log data generated by consumer internet companies, Jay Kreps et al. created a novel distributed messaging system that handles log data with minimal latency and high throughput.

Limitations of Existing Work: The authors indicate that existing messaging systems, such as IBM WebSphere, purely follow transaction-based processing, which guarantees atomicity while inserting log messages in a queue, creating overhead and performance issues for collecting log data. Similarly, the Java-based JMS service does not have the ability to batch a huge volume of data, which leads to the problem of low throughput. Additionally, many messaging systems lack the features of distributed computing and online processing of data.

Summary of Proposed Strategy: The authors proposed a novel distributed log processing architecture consisting of three main components: producer, consumer, and broker. The producer generates a message, and the consumer consumes it. The broker stores all messages as topics and serves them to multiple consumers reliably. Therefore, all messages are transferred as a pull-based system instead of a push-based strategy that other frameworks follow. Every log message in the broker is stored as a partitioned segment file with an offset based on the topic it is subscribed to. Partitions are distributed across the cluster or can be present within a single broker. To achieve better performance, data is flushed to disk after a certain level of log messages have been published. Later, it builds a sorted segment index in memory based on the offset. Once the data is in a consistent state after all balancing operations, it can be readily consumed by downstream users (consumers). Additionally, the distributed engine is stateless, leveraging the underlying file system cache and implementing built-in API to transfer data from a broker to a socket channel (consumer).

Summary of Methodology and Results of Evaluation: The authors conducted an experimental study with other messaging systems like RabbitMQ and ActiveMQ using two Linux machines (each with 2 GHz cores, 16 GB memory, 1 GB network link, and RAID 10), where one machine is used as a broker and another as a consumer. Testing the producer with 10 million messages, Kafka performed extremely well, achieving 2x throughput with a batch size of 1-50, whereas the other two messaging systems did not perform as expected due to using acknowledgement-based protocols. Similarly, on the second test with the consumer, Kafka demonstrated 4x better performance consuming 22000 messages per second, which is much higher than the other two messaging systems.

Any Research Directions Provided The authors provided multiple research directions in the field of distributed computing that involve distributed file storage and log processing in the context of streaming data.

[6] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

The problem being addressed: Through constant innovation and implementation, many organizations have had the opportunity to introduce multiple tools to handle big data. However, every tool has its own limitations, specifically for their use case. Therefore, a unified tool that can handle all the problems in a big data domain is a much-needed effort. The authors Zaharia et al. attempt to document and explain Apache Spark, a unified big data processing tool they developed to solve multiple problems that originate in a big data domain.

Limitations of Existing Work: According to the authors, previously, every domain in the data community used to have a separate processing engine for a specific technical requirement. For example, SQL was used for warehousing and descriptive analytics, streaming for real-time ingestion, machine learning for predictive analytics, and graph mining for graph processing. However, this kind of framework resulted in multiple issues for the teams working on varied data that originates from multiple sources and has multiple forms. Data teams were expected to have coordination and expertise in multiple areas, which was difficult to achieve.

Summary of Proposed Strategy: As per the authors, Spark was created to process varied data in a single engine that has the capabilities of multiple ones. It uses a programming model that is similar to map-reduce (a model that executes an action using multiple executors and aggregates the data at the end) and is known as RDD (Resilient Distributed Datasets). Similarly, it uses distributed computing to process jobs. However, the core difference lies in the computation part of RDD. As it performs compute operations using lazy evaluation and leverages lineage graph mechanism to process the submitted query, by optimizing it on the go. Spark can use any storage system, such as Hadoop file system (HDFS), cloud object storage from any public vendor (S3, Azure Blob Storage), and can query data based on file format. Although Spark does support various file formats, the most frequent one that is used by the community is Parquet. Additionally, the source mentions around 1000 contributors to

the project, and various organizations are using it for their productionized big data environments ranging from biotechnology to the finance industry.

Summary of Methodology and Result of Evaluation: In the view of authors, this is a survey paper that doesn't have any specific methodology to discuss. Still, a few evaluation results were discussed, such as training an iterative model. When a model is trained on the system using Hadoop and Spark, Spark processed its job much faster compared to Hadoop because it uses an efficient technique that does not spill data to the underlying disk from a cluster. Instead, it only loads the blocks once and proceeds to evaluate every consecutive step in memory, utilizing the previously computed results. Due to this reason, even when a job fails, it recomputes and retries in memory instead of fetching blocks from the distributed disks. Overall, Spark performed blazingly fast in contrast with Hadoop.

Any Research Directions Provided: The authors touched upon many research areas that range from distributed computing, big data technologies, and cluster computing programming models in the context of RDD (Resilient Distributed Dataset), reflecting the core ideas of this paper.