ADVANCED DATABASE SYSTEMS

SURVEY ON STREAM DATA MANAGEMENT SYSTEMS

Nayeon Lee, 1337749
nayeonl@student.unimelb.edu.au
Jothe Krishnan, 1187902
jkrishnan@student.unimelb.edu.au
Junkai Xing, 1041973
junkaix@student.unimelb.edu.au
Thomas Tien-Hung Chen, 1290641
thomastienhu@student.unimelb.edu.au

Abstract

In this survey, we explore the development and the characteristics of the Data Stream Management System (DSMS). The limitations of traditional DBSM in supporting streaming data prompted the emergence of DSMS and we explore how DSMS has been altered and developed throughout the past few decades. In this survey, we specifically explore four different Data Stream Management Systems. Not only do we explore numerous different topics for each of the systems including its architecture, storage management, query processing, security, performance and application but the paper includes the comparison between systems and how it has developed and been improved.

1. Introduction

The concept of a continuous data stream and the Data Stream Management Systems emerged a few decades ago and has continuously developed and evolved, adapting to newer applications and the advancement in technology. A data stream is continuous, real-time, potentially unbounded in size, and ordered either implicitly or explicitly. It is not possible to regulate the sequence in which items come, and also very challenging to store a complete stream locally. One other characteristic is that the elements of stream data are scrapped or archived once it has been processed, making it challenging to retrieve data. Similarly, *queries* over streams are long-running, continuous and persistent as they execute continuously over time and return new results incrementally. Hence, the Data Stream Management System (DSMS) was developed to handle and manage such streaming data and continuous queries.

There are four different DSMS that are investigated in the survey; STREAM, Aurora, Spark and Flink. The survey explores two early systems, STREAM and Aurora, which were developed when the concept of streaming data emerged; and two later systems, Spark and Flink, which have been refined and developed further according to the changing needs and applications of streaming data. Section 2 of the survey examines the characteristics of each system in depth, referring to critical concepts of DSMS. The survey then investigates the difference between each of the systems, including the comparison of their architecture, query language, performance, and various other characteristics in Section 3. Not only does it compare the system's attributes, but it explores the process of development.

2. Stream database management systems

In this section, four different stream database management systems are explored, including the earlier stage STREAM and Aurora, and the later stage Spark and Flink.

2.1. STREAM

STREAM (for Stanford StREam DatA Manager) is a relational-based, "general-purpose" DSMS which is developed at Stanford University (STREAM Group, 2003). Unlike the one-time queries which were evaluated once with a snapshot of data in traditional DBMS, continuous queries are processed in real-time as new data arrives. STREAM was designed to handle "long-running, continuous queries over unbounded", high-volume streams of data. STREAM utilises a language called Continuous Query Language (CQL) to express queries. CQL is a modified version of the standard SQL with the main modification being able to increase the expressiveness of the query language for sliding windows. STREAM supports two types of inputs: "streams and relations". (Akhtar, 2011)

Architecture:

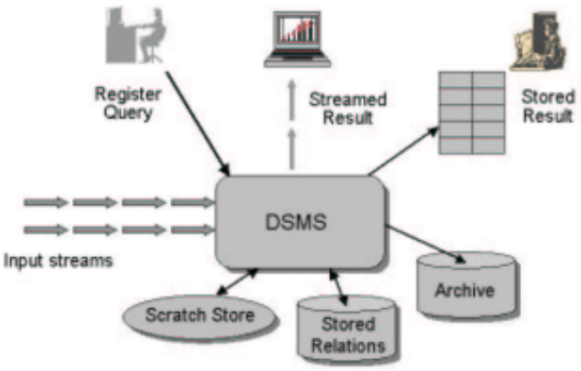

Figure 1 - Architecture of STREAM

Figure 1 shows a high-level summary of STREAM's architecture. When the input streams arrive, it drives the query processing procedure. Continuous query processing generally needs an interim state called 'Scratch Store'. Scratch Store consists of queues and synopses in queries and can be saved in memory or on the disc to be retrieved later (Arasu et al., 2003). Stream data can also be saved to an 'Archive' to preserve data and process costly analysis offline. The output of the queries is generally in the form of data streams, but it could also be in relational form updated over time (STREAM Group, 2003).

Query Processing:

The continuous queries are compiled into a query execution plan when registered into STREAM. For each query, a different plan is generated, and the system then integrates plans with corresponding or comparable sub-plans wherever possible to share memory and computation (Babcock et al., 2002). Query execution plans can also be directly submitted and manually combined with various optimisation algorithms. Query execution plans consist of query operators, inter-operator queues, and synopses. The operators are similar to ones found in conventional DBMSs; it reads each record from the input stream, processes them according to their semantics, and transfers the results to an output queue. The operators must be adaptive to deal with fluctuations in stream data, stream flow rates and the number of queries. The operators are connected by inter-operator queues and synopses are used to keep the record of their interim state associated with operators (Arasu et al., 2003). The execution of operators is coordinated by a central scheduler in STREAM. As memory management is particularly challenging for stream processing, the scheduler aims to carefully manage the run-time resources using a strategy called Chain Scheduling. This scheduling algorithm constructs query plans based on their effectiveness on memory usage and favours operations that take in as many inputs at a time and return few outputs (STREAM Group, 2003).

Adaptivity:

If query processing is not adaptable, the performance of STREAM may decrease when adjustments are made. The 'StreaMon' is an adaptive query processing infrastructure that is part of the STREAM system. StreaMon can reduce run-time memory requirements by maximising the use of patterns of stream data and it can perform adaptive join ordering for multiway stream joins. There are three main components within StreaMon: an Executor, Profiler and Reoptimizer. The Executor performs query

plans to produce outputs and the Profiler gathers and preserves information about “stream and plan characteristics”. The Reoptimizer guarantees the “most optimised and efficient plans and memory structures”. (Babu & Widom, 2004)

Approximation Techniques:

As mentioned previously, the incoming stream data can flow continuously without bound, exceeding the DSMS’s memory capacity and ability to produce accurate answers for the active queries. Approximation technologies emerged for the effective process of executing queries over rapid data streams (Babcock et al., 2002). Such techniques imply the trade-off between accuracy and storage memory, with the additional constraint of keeping processing time per item to a minimum. One way of approximating data stream is sampling methods such as hashing method, sampling methods, and batch processing. Such methods can be used when the arrival rate of data is so high that there is not enough CPU time to process. Another method is to analyse the query using a sliding window of current data from the streams rather than the complete prior history so that memory utilisation can be minimised by reducing the size of the synopsis.

Transaction Management:

Although it is not explicitly stated that STREAM supported transactions, the mechanism relied on logical timestamps that may be considered as transaction IDs. Input data with the same timestamp were executed atomically to provide isolation (Meehan, 2015).

Performance:

STREAM is a general-purpose DSMS but its targeted applications include real-time monitoring and financial analysis. Akhtar (2011) explored the performance of utilising STREAM in two different domains: Road Traffic analysis and Habitat Monitoring Analysis. The result was that STREAM was very robust as not a single tuple was dropped. It was also concluded that the output was very accurate, with the error percentage of about 0.125% to 0.025%. However, there was also the downside of STREAM where the robustness level dropped when “the number of simultaneous queries” increased to about 8 and above. Also, it was shown that the system crashed frequently on some aggregation operations and complex queries at high speed. Although the system did not perform perfectly, the system acted as a pioneer of DSMS. The STREAM project officially ended in 2006, however many other DSMSs emerged with the advancements in technology.

2.2. Aurora

This section referred to the papers of Abadi et al. (2003) with other papers for supplementary information. Aurora is a data-flow system that processes stream data with a technique called *boxes and arrows* paradigm, with processing operations as the boxes and the direction of data-flow as the arrows.

Workflow:

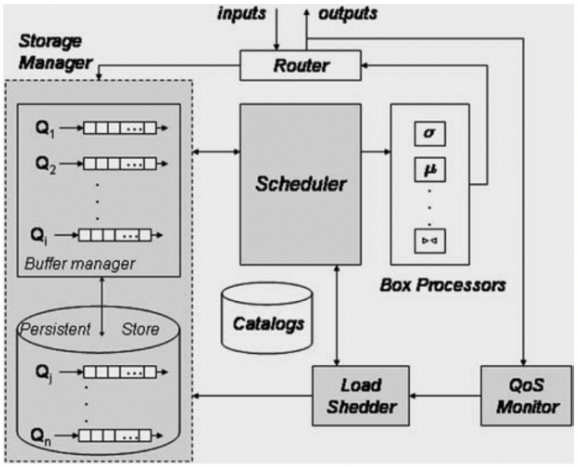

Figure 2 - Workflow of Aurora

In Figure 2, the router receives the data flows from the input sources and the output of box processors and then decides whether to output them to the applications or push them into the storage manager. The storage manager deals with the buffer for data to be pushed to box processors. Scheduler arranges boxes to be executed depending on the priorities, order, and the processing data length with QoS specified by the external application (Carney et al., 2003). The box processors then receive the indexes of the boxes required for processing, execute the operation and send outputs to the router. QoS monitor constantly monitors the system and is triggered if the system is at low-performance quality, the load shedder is then activated and shed the system to reach a satisfied status (Abadi et al., 2003). "Quality-of-service (QoS)" specification is defined by the requirements from the external applications, and it is utilised as an indicator when making decisions on how dynamic resources are allocated (Abadi et al., 2003).

Query Processing:

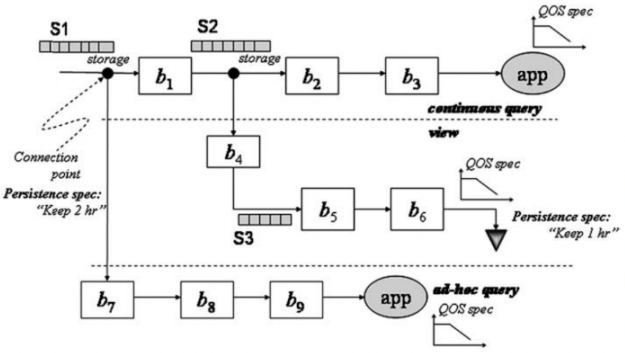

Figure 3 - Query processing of Aurora

In figure 3, the dark circles in the graph are the *connection points.* Boxes can be added or removed from the connection points, which enables the network to do dynamic modifications. Also, connection points themselves can be viewed as a stored data set. The stored data in the connection points provide frequently used data for operations. The top path in the graph represents the *continuous query* which is operating in isolation. Stream data flows into boxes for processing and boxes can do basic operations such as filtering, mapping, aggregate and join. Ad hoc query is represented by the bottom path. Ad hoc query refers to the query that emerges due to new requirements and cannot be solved with its implementation at that time. In Aurora, an ad hoc query is added to connection points with subsequent boxes. It works as a continuous query for a period of time, and the duration is indicated in the persistence specification (Abadi et al., 2003).

Storage management:

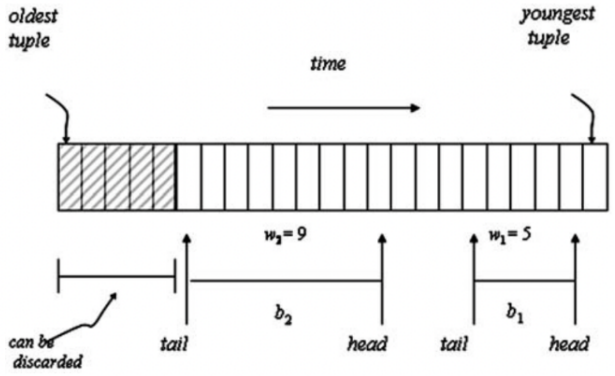

Figure 4 - Data structure in Aurora

Aurora Storage Manager needs to deal with two kinds of storage in the system; the storage for data-flow in the network and the storage for the connection points. Aurora storage manager utilises a queue, which is shown in figure 4, as the data structure to store temporary data in the buffer manager, hence it is also referred to as queue management. A queue receives output from the last box and can have multiple successor boxes, and each of them has two pointers which specify the range of data the box requires, also known as window. The data that has been processed by all successor boxes will be removed from the queue to ensure no space is wasted (Abadi et al., 2003).

For the optimization purpose, the storage manager needs to allocate the space in the main memory for data used by high priority boxes, and the priorities of boxes are determined by the scheduler. The communication between the storage manager and scheduler is done by a tabular data structure to store information each row for a box. The historical data are stored in the persistent storage with B-tree data structure. Aurora can execute the corresponding indexed search based on the data required by the operators (Abadi et al., 2003).

Load shedding:

Load shedding is used when the system is overloaded and tries to reduce the size of data in processing. The loss of data potentially results in unnecessary degradation or affects the semantics of the system, while the two issues are addressed (Abadi et al., 2003). With techniques like the greedy algorithm and QoS information, Aurora manages to do load shedding while ensuring the best cost-performance ratio and dropping less important data using filters and algorithms (Babcock et al., 2003).

2.3. APACHE SPARK

Apache Spark is an open-source, big data processing tool that possesses both Batch and Stream processing capabilities that allows parallel data processing on computer clusters. Unlike models that read from and write back to the disk for each computation like MapReduce, Spark offers ‘full in-memory computation’, which guarantees better performance and lower latency. However, Spark can also perform traditional disk-based processing. Spark works with a unique data type called Resilient Distributed Data (RDD) which is an immutable, fault-tolerant collection of data, partitioned across different clusters in the application.

Some notable features of Spark include in-memory computing, high speed, auto-scaling, cross-platform support and real-time stream processing.

Spark architecture and ecosystem:

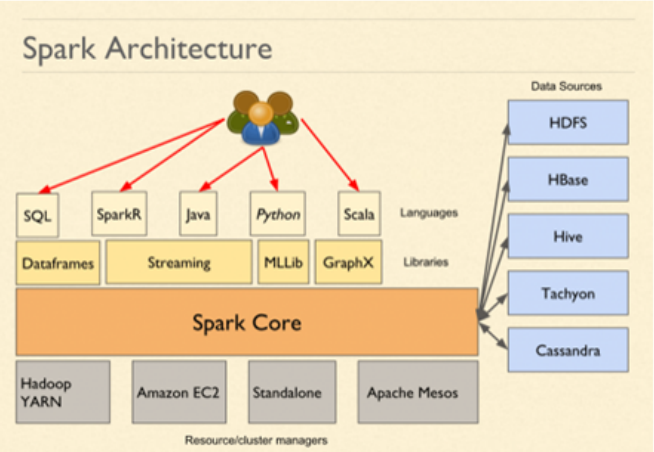

Figure 5 - Spark Ecosystem

The Spark core forms the foundation for several functionalities to be built on top of it. It offers a multitude of APIs and applications that support different programming languages like Python, Java, Scala, .NET, etc. SparkSQL, formerly known as Shark, has a data abstraction component called DataFrames which supports both structured and semi-structured data. It can perform query processing on data obtained from Hive, JSON or any RDBMS. Spark streaming allows the data to be processed in real-time by converting the data to mini-batches unlike its other streaming components like Flink which processes event by event.

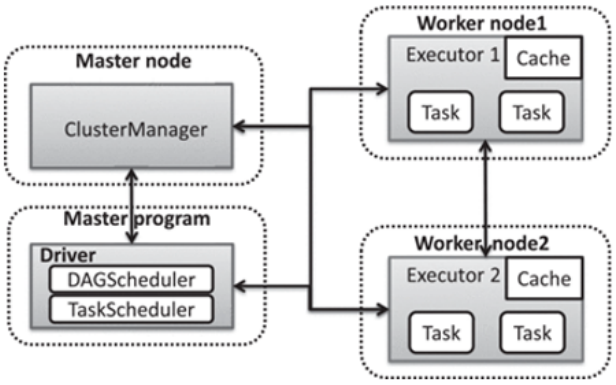

Figure 6 - Spark Architecture

Although the streaming component uses the same technology as batch processing, where each mini-batch is converted to RDD, it incurs some latency. It also provides a separate streaming technology based on datasets called structured streaming. MLib provides a vast range of machine learning algorithms to use like regression, classification, clustering, etc. Due to Spark's in-memory cluster computation, it offers parallel processing of these algorithms iteratively and offers pipeline functionalities. GraphX is a graph computation application that allows building, transforming and manipulating graphs over RDDs. Since RDDs are immutable, Graphs are also immutable.

Figure 6 shows the architecture of Spark, on a cluster. Every Spark application creates a new Master node called the driver, where the Master program in the driver is responsible for task scheduling. The Master program provides a Spark context which acts as a gateway to all the Spark functionalities that could be used by the program. The hierarchy of job scheduling is jobs, stages and finally tasks. A stage is a subset of jobs where the stages are interconnected, where one stage could be Map and the other is the Reduce part of a MapReduce job. A job can either be split into stages or tasks. The DAGScheduler in the Master node is responsible for allocating the jobs and monitoring the stages. It

keeps track of the output and the RDDS at each stage. The TaskScheduler is responsible for scheduling low-level tasks in each stage. It schedules and submits the task at each stage, to the cluster. The worker nodes are responsible for executing the tasks. When an RDD is created in a Spark context, it is sent to the worker nodes to execute and is temporarily stored in the cache. Once the task is executed, they return to the Spark context.

Query processing and execution:

The traditional query processing engine in Spark, Spark SQL works on Dataframes, Datasets and RDDs. A dataset is a distributed collection of data and a dataframe is a Dataset, organised into named columns which is like relational tables but with better optimization techniques. The data could be derived from various sources like Hive tables, external databases, structured data files or just existing RDDs. To process a query, a dataframe or a dataset is created using APIs and uses the SQL functions of Spark session. Query execution in Spark follows a tree structure where the logical plan of the query is converted to a physical plan. And the logical operators are converted to physical operators like union, join, filter, etc.

Spark has transformation and action RDD functions, where the transformation changes an existing RDD to a different version of RDD, like performing a join or an intersection on input RDDs. Action functions on an RDD give an output after performing some computations on the RDDs like count, sum, etc. Spark SQL has in-built optimization techniques that are used to optimise the query to minimise cost of execution.

Although the Spark SQL API performs fairly well with its inbuilt query execution functions and query optimizers, there have been some notable developments by researchers in the field to compare and build better query processing systems. Agarwal (2014) proposed an “approximate query processing system, atop Shark and Spark”. Their system worked by maintaining an adaptive optimization framework that has multi-dimensional samples of raw data, and “dynamically selecting samples based on accuracy and response time of queries”. Naacke (2016) compared four SparQL query processing methods and concluded that a hybrid that combines partition join and broadcast joins showed the best performance for processing queries.

With recent growing attention in incremental BigData analysis, Fegaras (2016) came up with an incremental query processing system on a distributed platform. Their prototype model of ‘MRQL streaming’ on top of Apache Spark was developed to process large stream data incrementally, where instead of using the standard micro-batching method of Spark, their system processes data in batches incrementally using MRQL algebra.

Structured streaming in Spark is based on continuous query processing, where it reads the input data from a source like Apache kafka, kinesis or other databases and writes to an output sink. The input data is in the form of tables and Spark performs incremental query processing on the stream. New data is appended to the input everytime it comes. M.Armbrust (2018) developed a declarative API that performs structured streaming with Spark streaming. Their system combines the query optimization

techniques of Spark SQL with their logic that avoids system crashing with incoming data and recomputing the output after incrementally appending input data. Their system proved to be much faster than Spark and almost twice as fast as Flink.

Since Spark is an in-memory computation system, it doesn't have a database. It uses external databases to store and load data like HDFS, Hive tables.

Although Spark boasts some notable advantages over its competitors, it also has some disadvantages to consider. In comparison to Hadoop, Spark has poor security. As it takes time to familiarise with the APIs in the system, Spark poses a steep learning curve. It also consumes a high volume of data since the data needs to be converted to RDD for computation.

2.4. APACHE FLINK

Apache Flink is an open-source stream data processing Framework. It can process both stream and batch data at a large scale and provide insightful analysis of the data in real time. Unlike STREAM and Aurora, which are the early prototypes of stream data processing systems and are no longer being developed, Apache Flink is up-to-date and trending for business and research (Hesse & Lorenz, 2015). Although Apache Flink is not as widely used as Apache Spark in business applications, it provides lots of great functionalities including stream and batch data processing, state management, exact-once message processing guarantee, and great scalability. Apache Flink even overtook Apache Spark in streaming capability thanks to its underlying design (Nazari et al., 2019). In this section, we would discuss Apache Flink in detail.

Runtime, APIs, and Architecture:

Apache Flink runtime and APIs can be categorised into three categories, which are core, APIs & libraries, and deployment. (Carbone et al., 2015) The core of Flink is a "distributed dataflow engine" that executes "dataflow programs". The two core APIs of the Flink framework are the DataSet API and the DataStream API. Both APIs would "create runtime programs" executable by the dataflow engine, while DataSet API is used to handle the batch data and the DataStream API is used to handle stream data. This enables Flink for both batch processing and stream processing. It is worth noting that Flink also has libraries that support machine learning, graph processing, and SQL execution. Flink can be deployed in a local machine, as a cluster, or on the cloud.

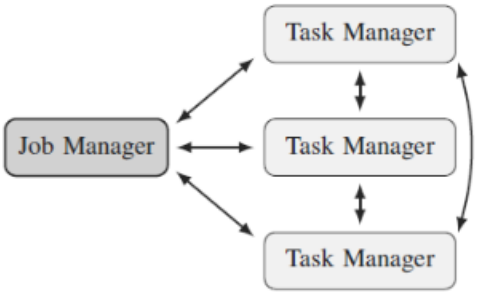

Figure 7 - Architecture of Apache Flink

The architecture of the Flink runtime is shown in Figure 7. Flinks framework consists of two major components, which are the JobManager and the TaskManager. A client is not a "part of the runtime and program execution" and it is used to "prepare and send a dataflow to the JobManager" ("Flink architecture," n.d.). After receiving a job from the client, the JobManager would schedule and assign the work to the TaskManagers to be

executed. When the work is being executed by the TaskManagers, the JobManagers would monitor the “overall execution status and the state” of the TaskManagers. The JobManager periodically creates checkpoints with states and metadata information and saves it in persistent storage. On system failures or restarts, the JobManager can “reconstruct the checkpoint and recover the dataflow execution from there” (Carbone et al., 2015). This gives Flink fault tolerance capability.

Each TaskManager has multiple task slots which are used to execute tasks in parallel. (“Flink architecture,” n.d.). Although the number of task slots in each TaskManager can be manually configured, it is recommended to set it to the number of CPU cores in “every TaskManager node” (Hesse & Lorenz, 2015). The task manager can exchange data streams with each other. The computation result and statistics are sent from the TaskManagers through the JobManager to the client.

State management:

For stream data management systems, it is often important to store intermediate computation results as they might be used for future computation with subsequent stream data. The intermediate computation results are called ‘state’ in Flink and it is maintained both temporarily in memory and persistently in external storage. There are two types of state in Flink, named “in-flight state” and “state snapshots”. (“Using rocksdb state backend,” n.d.) The “in-flight state” is the working state of a job. It is stored in “local memory” in a TaskManager, which is supported by the embedded key-value database “RocksDB”. The “state snapshots”, on the other hand, is the checkpoint record used for state recovery. State snapshots are generated periodically with the latest state information of jobs and saved to an external persistent database. On system failure, the snapshot is used to recover from the last checkpoint. (Carbone et al., 2017)

Optimization:

In contrast to the micro-batching processing in Spark, which groups arriving data into “micro-batches” and processes them accordingly, Flink offers native streaming capability supported by DataStream API (Nazari et al., 2019). This allows Flink to process the input stream continuously as soon as it arrives. Since the input stream does not need to wait to be grouped, Flink can start processing input faster, which results in lower latency than Spark. However, Flink can support “MiniBatch Aggregation” for optimization purposes. As mentioned before, Flink keeps and updates state information in real-time. By default, the simple aggregation will need to read, update, and then store the updated state information for each arriving data. This can cause lots of overhead in the state backend like RocksDB. The “MiniBatch Aggregation” would cache “a bundle of inputs in a buffer” and then execute them together to only update the state once for each MiniBatch. This significantly reduces the overhead in the state backend. However, it can increase the latency as the streamed data is no longer processed “in an instant” (“Performance tuning,” n.d.)

Security:

Security is a very important part of any stream data management system. The Flink framework itself does not come with any security components. It achieves security in authentication and encryption by incorporating external components such as Hadoop/Kerberos and SSL/TLS. Kerberos is an "authentication system" designed for an "open network computing environment" (Steiner et al., 1988). In Flink, Kerberos can be used to authenticate a user's identity before establishing secure data access to jobs. ("Kerberos Authentication Setup," n.d.) SSL and TLS are two cryptographic protocols used for web communication encryption to ensure data integrity. (Thomas, 2000) Since there is both internal communication between the JobManagers and the TaskManagers and external communication between the client and the Flink system, SSL and TLS are used to encrypt and protect the transferred data to keep a secure communication among all subjects.

Usage in business:

Although Flink is commercially used by some companies for its high performance. A typical use case is found in Alibaba, which is the largest e-commerce company in China. Alibaba uses Flink as their real-time "search and recommendation" engine because of the low latency and high throughput of the system. Flink was able to handle "472 million transactions per second" during peak business hours (Feng, 2019). Aside from the search and recommendation engine, Flink can also be used in "Fraud detection", "Anomaly detection", and "Quality monitoring of Telco networks" ("Use cases," n.d.).

3. Comparison between systems

In this section, we perform comparisons of the systems mentioned above in two stages. We first compare the two older systems (STREAM and Aurora) and two newer systems (Apache Spark and Apache Flink) and then compare all four systems. The comparison is carried out in this manner to contrast the systems from two different generations.

3.1. STREAM vs Aurora

STREAM and AURORA both use "workflow-based solutions". They both have a layer in their system where they take in the inputting streams and apply operations on them such as data partitioning or load shedding. However, the input of Aurora is restricted to stream data whereas STREAM is capable of handling stream data as well as relations as its input. They both have a compiler, which creates executable query plans, and a scheduler which employs a technique to allocate operators across processing units.

The biggest difference between Aurora and STREAM is the implementation of CQL language in the STREAM application (Kotto-Kombi et al., 2015). With the help of CQL, STREAM is able to process Stream-to-Relation operations which enables it to execute SQL queries. In contrast, Aurora can only perform SQL-like queries with the combination of seven predefined simple queries. Furthermore,

STREAM manages to utilise chain scheduling algorithms to reduce the memory usage in run time also with the support of CQL.

They both utilise an extra data structure for quick access to the required data. Aurora stores such data in a buffer manager with a queue data structure, including the data from the data-flow and the historical information extracted from persistent storage. STREAM stores such data in the Scratch Store with queue data structure, including data from data-flow and synopsis (Botan et al., 2009). Synopsis is used to store the continuous input data-flow at a certain moment so that traditional relations language can be used.

Both Aurora and STREAM are geared towards small-scale applications. Due to the limitations of memory and core capacity, Aurora is limited to small-scale applications. Aurora is designed specifically for stream data from monitoring and hence targeted applications like sensor network monitoring whereas STREAM is a more general-purpose database management system targeted applications which handle real-time monitoring and financial analysis.

3.2. Apache Spark vs Apache Flink

Both Apache Spark and Apache Flink are trending stream data management systems that are used in many domains including business and research. The Spark project is mainly written in Java, Scala, and Python, while the Flink project is mainly written in Java and Scala. Both systems have APIs for Java and Scala programming languages that can be integrated by developers to build new features (Hesse & Lorenz, 2015). One of the most significant differences between the two systems is the way they handle the input streams. Flink supports “native streaming”, which means the streamed data is individually and continuously processed in real-time as soon as it is generated. In contrast, Spark only supports “Micro-Batching”, where the continuous input stream is split into small batches to be processed later (Gorasiya, 2019). Both systems follow the exactly-once “message delivery guarantee”, meaning that it is guaranteed that all the input streams will be processed once and only once. (Alkatheri et al., 2019). Both systems support “in-memory computing”, which leads to better performance as there is no need to access disks. However, the content in memory might be lost when the machine is turned off (Alkatheri et al., 2019). Luckily, this is addressed by the state management mechanisms. For stream data management systems, intermediate computation results of the processed streams are usually saved to be further used with the incoming input streams. The intermediate computation results are called state. Both systems will periodically save the state's information along with the metadata, such as system configurations and the offsets on processed streams, to persistent storage (Gorasiya, 2019). The saved states and metadata information are called checkpoints in Spark and snapshots in Flink. On system failures, the checkpoints and snapshots help it roll back to the last successful state to continue running. This fault-tolerant design also helps guarantee all streams will be processed exactly one time. Although both systems have the ability to scale horizontally, Apache Spark provides auto-scaling where it can add workers on the fly depending on the load. Once the load

drops, the cluster scales down to the original number of nodes it was spun up. Whereas Flink does not support auto-scaling and the only way to scale jobs in Flink is to shut down the job with a savepoint and restart the job from the savepoint with new parallelism (Alkatheri et al., 2019). Comparing the performance of both systems, while Spark is faster than Hadoop, it is much slower than Flink, where Spark takes a few seconds to complete a job, and Flink only needs a sub-second (Inoubli et al., 2018). On data partitioning in both systems, Spark has a Hash partitioner and a range partitioner, where the hash is the default partitioner. Flink does not have a default partitioner.

3.3. Overall comparison of STREAM, Aurora, Apache Spark, and Apache Flink

Features	STREAM	Aurora	Apache Spark	Apache Flink
Release Date (First Stable Version)	2003	2003	2014	2016
Community	Discontinued	Discontinued	Popular and up-to-date	Popular and up-to-date
Input	Raw stream data and Relations	Raw stream data	DStream	Data stream
Execution Support	Centralized multi-core	Centralized multi-core	Distributed	Distributed
Computation Mode	CPU and memory based computation	Memory and disk based computation	In-memory computation	In-memory computation
Fault Tolerance	None	None	Checkpoint	Snapshots
Scalability	None	None	Yes (Automatic)	Yes (Manual)

Figure 8 - Overall Comparison of STREAM, Aurora, Apache Spark, and Apache Flink

Figure 8 shows the detailed comparison over the four systems. STREAM and Aurora are the pioneers of the stream data management systems. At the earlier stage, the concept of window comes out. Researchers started to use it to deal with continuous time-series dataflow. Nonetheless, issues related to security, fault tolerance and scalability were not addressed due to the restriction of knowledge and technology at that time. As time passed, those issues were addressed with the emergence of modern technologies and extensive research on stream data management systems. The input data vary from one to another to support the functionality and improve the performance. With the distributed systems across different servers, stream data management is able to share the burden of heavy computation and do calculations locally. Fault tolerances are implemented in different ways to make sure the systems do not corrupt due to a failure value in the continuous data. Last but not least, as the size of stream data grows, researchers engage in the area of scalability to ensure that stream data management systems can flex the systems to cater to the stream data size.

4. Limitations and Future Works

There are numerous modern stream management systems in the market, such as Apache storm and Amazon Kinesis. However, we only covered four representative stream data management systems throughout history in this survey due to the limitations of timeframe and word count. We also found that there is limited research on security and transaction management in Stream Data Management Systems, and some of the existing systems suffer from “not having specific optimization at each step

of query processing" (Kotto-Kombi et al., 2015). Those limitations are expected to be addressed with future research.

5. Conclusion

This report surveys four of the notable systems for stream data management. Starting with two of the foundational models, STREAM and Aurora, we see that these two applications were pioneers in Stream Data Management Systems. Introducing new techniques like load shedding to combat rapid incoming data and strategies like Approximation and Adaptivity, these systems set a good baseline idea for future systems. While both applications had a lot of similarities, it should be mentioned that STREAM had the capability to operate on stream data as well as relational data with its query processing tool, CQL. Whereas, Aurora was limited to stream input data. On comparing the two newer Apache systems, Spark and Flink, we conclude that Flink is much faster than Spark due to its architecture and improved APIs. Although Spark has a wider community with more contributors and Flink is less preferred, Flink outperforms Spark in terms of its streaming capabilities, due to its native streaming feature and Spark might incur higher latency due to its micro-batching technique. With the promise of lower latency and better scalability, Flink could be the future of real-time streaming.

6. References

Alkatheri, S., Abbas, S. A., & Siddiqui, M. A. (2019). A Comparative Study of Big Data Frameworks. *International Journal of Computer Science and Information Security*.

Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., & Zdonik, S. (2003). Aurora: A new model and Architecture for Data Stream Management. *The VLDB Journal The International Journal on Very Large Data Bases*, *12*(2), 120–139. https://doi.org/10.1007/s00778-003-0095-z

Akhtar, N. (2011). Statistical data analysis of continuous streams using stream dsms. *Proceedings of the International Journal of Database Management Systems*, *3*(2), 89-99.

Agarwal, S., Milner, H., Kleiner, A., Talwalkar, A., Jordan, M., Madden, S., ... & Stoica, I. (2014, June). Knowing when you're wrong: building fast and reliable approximate query processing systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (pp. 481-492).

Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., ... & Widom, J. (2003, June). STREAM: the stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (pp. 665-665).

Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., ... & Widom, J. (2016). Stream: The stanford data stream management system. In *Data Stream Management* (pp. 317-336). Springer, Berlin, Heidelberg.

Armbrust, M., Das, T., Torres, J., Yavuz, B., Zhu, S., Xin, R., ... & Zaharia, M. (2018, May). Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data* (pp. 601-613).

Babcock, B., Babu, S., Datar, M., Motwani, R., & Widom, J. (2002, June). Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (pp. 1-16).

Babcock, B., Datar, M., & Motwani, R. (2003). Load shedding in Data Stream Systems. *Advances in Database Systems*, 127–147. https://doi.org/10.1007/978-0-387-47534-9_7

Babu, S., & Widom, J. (2004, June). StreaMon: an adaptive engine for stream query processing. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (pp. 931-932).

Botan, I., Alonso, G., Fischer, P. M., Kossmann, D., & Tatbul, N. (2009). Flexible and scalable storage management for data-intensive stream processing. In Proceedings of the 12th International Conference on Extending Database Technology Advances in Database Technology - EDBT '09 (pp. 934–945). EDBT. https://doi.org/10.1145/1516360.1516467

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull., 38*, 28-38.

Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., & Tzoumas, K. (2017). State Management in Apache Flink®. *Proceedings of the VLDB Endowment*, *10*(12), 1718–1729. https://doi.org/10.14778/3137765.3137777

Carney, D., Çetintemel, U., Rasin, A., Zdonik, S., Cherniack, M., & Stonebraker, M. (2003). Operator

Feng, W. (2019, August 2). *Why did alibaba choose apache flink anyway?* Alibaba Cloud Community. Retrieved May 22, 2022, from https://www.alibabacloud.com/blog/why-did-alibaba-choose-apache-flink-anyway_595190

Flink architecture. Flink Architecture | Apache Flink. (n.d.). Retrieved May 22, 2022, from https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/concepts/flink-architecture/

Gorasiya, D. (2019). Comparison of Open-Source Data Stream Processing Engines: Spark Streaming, Flink and Storm.

H. Naacke, B. Amann, and O. Cure, "Sparql graph pattern processing with apache spark," in Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems. ACM, 2017, p. 1.

Hesse, G., & Lorenz, M. (2015). Conceptual survey on Data Stream Processing Systems. *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. https://doi.org/10.1109/icpads.2015.106

Inoubli, W., Aridhi, S., Mezni, H., Maddouri, M., & Nguifo, E. M. (2018). A Comparative Study on Streaming Frameworks for Big Data.

Kerberos Authentication Setup and Configuration. Kerberos | Apache Flink. (n.d.). Retrieved May 22, 2022, from https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/deployment/security/security-kerberos/

Kotto-Kombi, R., Lumineau, N., Lamarre, P., & Caniou, Y. (2015). Parallel and Distributed Stream Processing: Systems Classification and Specific Issues.

L. Fegaras, "Incremental Query Processing on Big Data Streams," in IEEE Transactions on Knowledge and Data Engineering, vol. 28, no. 11, pp. 2998-3012, 1 Nov. 2016, doi: 10.1109/TKDE.2016.2601103.

Meehan, J., Tatbul, N., Zdonik, S., Aslantas, C., Cetintemel, U., Du, J., ... & Wang, H. (2015). S-store: Streaming meets transaction processing. *arXiv preprint arXiv:1503.01143*.

N., & Zdonik, S. (2003). Aurora: a new model and architecture for data stream management. *The VLDB Journal the International Journal on Very Large Data Bases*, *12*(2), 120–139. https://doi.org/10.1007/s00778-003-0095-z

Nazari, E., Shahriari, M. H., & Tabesh, H. (2019). Bigdata analysis in Healthcare: Apache Hadoop , Apache Spark and Apache Flink. *Frontiers in Health Informatics*, *8*(1), 14. https://doi.org/10.30699/fhi.v8i1.180

Performance tuning. Performance Tuning | Apache Flink. (n.d.). Retrieved May 22, 2022, from https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/dev/table/tuning/

Steiner, J.G., Neuman, B.C., & Schiller, J.I. (1988). Kerberos: An Authentication Service for Open Network Systems. *USENIX Winter*.

STREAM Group. (2003). *STREAM: The stanford stream data manager*. Stanford InfoLab.

Scheduling in a Data Stream Manager. *Proceedings 2003 VLDB Conference*, 838–849. https://www.academia.edu/50165479/Operator_Scheduling_in_a_Data_Stream_Manager

Tang, S., He, B., Yu, C., Li, Y., & Li, K. (2020). A survey on spark ecosystem: Big data processing infrastructure, machine learning, and applications. *IEEE Transactions on Knowledge and Data Engineering*.

Thomas, S. (2000). *Ssl and Tls Essentials: Securing the web*. John Wiley & Sons.

Use cases. Apache Flink. (n.d.). Retrieved May 22, 2022, from https://flink.apache.org/usecases.html

Using rocksdb state backend in Apache Flink: When and how. Apache Flink. (n.d.). Retrieved May 22, 2022, from https://flink.apache.org/2021/01/18/rocksdb.html