# Discretized Streams vs Apache Flink

Ottar Gudmundsson  Xin Ren

## I. Motivation

### A. Discretized Streams

Much of ”big data” is received in real time, and is most valuable at its time of arrival. To enable these low-latency processing applications, there is a need for streaming computation models that scale transparently to large clusters, which introduces two major problems: faults and stragglers. Unfortunately, existing streaming systems based on a continuous operator model, performing recovery through replication or upstream backup, have limited fault and straggler tolerance:

1) Replication costs 2x the hardware, while upstream backup takes a long time to recover, as the whole system must wait for a new node to serially rebuild the failed nodes state by rerunning data through an operator.
2) In upstream backup, a straggler must be treated as a failure, incurring a costly recovery step, while replicated systems use synchronization protocols like Flux to coordinate replicas, so a straggler will slow down both replicas.

### B. Apache Flink

Data-streaming processing and batch data processing were traditionally considered as two different applications, programmed using different models and APIs. Batch data analysis made up for the lions share of the use cases, data sizes, and market, while streaming data analysis mostly served specialized applications. However a huge number of today’s large-scale data processing use cases handle data that is produced continuously over time. Today’s setups ignore the continuous and timely nature of data production. Data records are batched into static data sets and then processed in a time-agnostic fashion. Architectural patterns such as the ”lambda architecture” combine batch and stream processing systems to implement multiple paths of computation: a streaming fast path for timely approximate results, and a batch offline path for late accurate results. All these approaches suffer from high latency, high complexity, as well as arbitrary inaccuracy, as the time dimension is not explicitly handled by the application.

## II. Contributions

### A. Discretized Streams

This paper presents a new stream processing model, discretized streams (D-Streams), which structures a streaming computation as a series of stateless, deterministic batch computations on small time intervals. D-Streams provides a more efficient recovery mechanism, parallel recovery, that runs faster than upstream backup without the cost of replication and tolerates stragglers, using speculative execution. They support a wide range of operators and can attain high per-node throughput, linear scaling to 100 nodes, sub-second latency, and sub-second fault recovery. They use the same execution model as batch platforms, and compose seamlessly with batch and interactive queries.

### B. Apache Flink

The contributions of this paper are as follows:

1) It makes the case for a unified architecture of stream and batch data processing, including specific optimizations that are only relevant for static data sets,
2) It shows how streaming, batch, iterative, and interactive analytics can be represented as fault-tolerant streaming dataflows,
3) It discusses how to build a full-fledged stream with a flexible windowing mechanisum, as well as a full-fledged batch processor on top of these dataflows.

## III. Solution

### A. Discretized Streams

Input stream is divided into small time interval batches and stored in Sparks memory as RDDs, which form a D-Stream. The data received in each interval forms an input dataset for that interval. Once the time interval completes, this dataset is processed via deterministic parallel operations: transformations produce new datasets representing intermediate state which is stored as RDDs and may then be processed along with the next batch of input data to produce a new dataset of updated intermediate states; output operations produce new datasets representing program outputs that may be pushed to an external system in a distributed manner. D-Streams support the same stateless transformations available in typical batch frameworks, including map, reduce, groupBy, and join and also several stateful transformations for computations spanning multiple intervals, based on standard stream processing techniques such as sliding windows.

Spark Streaming is implemented based on above model and consists of three components: a master that tracks the D-Stream lineage graph and schedules tasks to compute new RDD partitions; worker nodes that receive data, store the partitions of input and computed RDDs, and execute tasks; a client library used to send data into the system. Applications start by defining one or more input streams. The system can load streams either by receiving records directly from clients, or by loading data periodically from an external storage system. In the former case, new data is replicated across two worker nodes before sending an acknowledgement to the client library, because D-Streams require input data to be stored reliably to recompute results. If a worker fails, the client library sends unacknowledged data to another worker. All data is managed by a block store on each worker, with a tracker on the master to let nodes find the locations of blocks. Each block is simply given a unique ID, and any node that has that ID can serve it. The block store keeps new blocks in memory but drops them to disk if there is not enough memory. The user can also set a maximum history timeout bigger than the checkpoint interval, after which the system will simply forget old blocks without doing disk I/O. To decide when to start processing a new interval, it's assumed that the nodes have their clocks synchronized via NTP, and each node sends the master a list of block IDs it received in each interval when it ends. The master then starts launching tasks to compute the output RDDs for the interval and starts each task whenever its parents are finished. For optimizations, Spark Streaming pipelines operators that can be grouped into a single task, places tasks based on data locality and controls the partitioning of RDDs to avoid shuffling data across the network.

Sigificant optimizations and changes were made to Spark to support streaming. These included: data plane rewritten to use asynchronous I/O to let tasks with remote inputs fetch them faster; scheduler modified to allow submitting tasks from the next timestep before the current one has finished; multiple optimizations to task scheduler, such as hand-tuning the size of control messages, to be able to launch parallel jobs of hundreds of tasks every few hundred milliseconds; storage layer rewritten to support asynchronous checkpointing of RDDs and to increase performance; scheduler modified to forget lineage after an RDD has been checkpointed, so that its state does not grow arbitrarily, and other data structures that grew without bound were given a periodic cleanup process; support added for recovering the Spark masters state if it fails.

To recover from faults and stragglers, both D-Streams and RDDs track their lineage. When a node fails, D-Streams allow the state RDD partitions that were on the node, and all tasks that it was currently running, to be recomputed in parallel on other nodes. The system periodically checkpoints some of the state RDDs, by asynchronously replicating them to other worker nodes. Then, when a node fails, the system detects all missing RDD partitions and launches tasks to recompute them from the last check point. D-Streams mitigate stragglers by running speculative backup copies of slow tasks which are detected whenever a task runs more than 1.4x longer than the median task in its job stage. To tolerate failures of Sparks master, the state of the computation is wrote reliably when starting each timestep and workers connect to a new master and report their RDD partitions to it when the old master fails. Upon recovery, the new master reads D-Stream metadata (the graph of the users DStreams and Scala function objects representing user code, the time of the last checkpoint and the IDs of RDDs since the checkpoint) that stored in an HDFS file that is updated through an atomic rename on each timestep to find where it left off, and reconnects to the workers to determine which RDD partitions are in memory on each one. It then resumes processing each timestep missed.

To handle out-of-order input, the system can wait for a limited slack time before starting to process each batch and user programs can correct for late input at the application level. D-Streams provide exactly-once consistency semantics, because time is naturally discretized into intervals, and each intervals output RDDs reflect all of the input received in that and previous intervals. D-Streams follow the same processing model, data structures, and fault tolerance mechanisms as batch systems, the two can be unified with powerful features provided by Spark Streaming: D-Streams can be combined with static RDDs computed using a standard Spark job; users can run a D-Stream program on previous historical data using a batch mode; users run ad-hoc queries on D-Streams interactively by attaching a Scala console to their Spark Streaming program and running arbitrary Spark operations on the RDDs there.

## *B. Apache Flink*

There are four main layers in Flink: deployment, core, APIs and libraries. The core of Flink is the distributed dataflow engine. A Flink runtime program is a DAG of stateful operators connected with data streams. Two core APIs  Dataset API for batch processing and DataStream API for stream processing. Flink's core runtime engine can be seen as a streaming dataflow engine and both APIs create runtime programs executable by the engine. On top of the core APIs, Flink bundles domain-specific libraries and APIs that generate DataSet and DataStream API programs. A Flink cluster comprises three types of processes: the client which transforms the program code to a dataflow graph and submits it to the Job Manager; the Job Manager which coordinates the distributed execution of the dataflow, tracks the state and progress of each operator and stream, schedules new operators and coordinates checkpoints and recovery; at least one Task Manager where the actual data processing takes place.

The dataflow graph is a DAG consists of stateful operators and data streams, produced by an operator and available to be consumed by other operators. Operators are parallelized into one or more subtasks and streams are split into stream partitions (one per subtask). Intermediate data streams are the core abstracton for data-exchange between operators. It represents a logical handle to the data that may or may not be materialized on disk. Flink uses pipelined streams between consumers and producers for continuous streaming programs, and many parts of batch dataflows, to avoid materialization when possible. They propagate backpressure, modulo some elasticity via intermediate buffer pools, to compensate for short-term throughput fluctuations. Blocking streams are applicable to bounded data streams, which do not propagate backpressure by buffering all of the producing operators data before making it available for consumption. When a data record is ready on the producer, it's serialized and split into buffers that can be forwarded to consumers as soon as a buffer is full or a timeout is reached, so Flink can achieve high throughput by setting high buffer size and low latency by setting low buffer timeout. Besides data, streams can communicate control events that injected by operators, delivered within a partition and reacted by the receiving operators. Assuming data sources are persistent and replayable, Flink deals with failures via checkpointing and partial re-execution, and offers reliable execution with strict exactly-once-processing consistency guarantees by building the checkpointing on the notion of distributed consistent snapshots. To bound recovery time, Flink takes a snapshot of the state of operators, including the current position of the input streams at regular intervals. Asynchronous Barrier Snapshotting is used to take a consistent snapshot of all parallel operators without halting the execution of the topology. Recovery from failures reverts all operator states to their respective states taken from the last successful snapshot and restarts the input streams starting from the latest barrier for which there is a snapshot.

Flinks DataStream API implements a full stream-analytics framework. It distinguishes between two notions of time: event-time and processing time, and includes a third notion of time called ingestion-time, which is the time that events enter Flink, to achieve a lower processing latency than event-time and more accurate results than processing-time. Operators in DataStream API support efficient stateful computations. State is made explicit and is incorporated in the API by providing: i) operator interfaces or annotations to statically register explicit local variables within the scope of an operator and ii) an operator-state abstraction for declaring partitioned key-value states and their associated operations. Users can also configure how the state is stored and checkpointed using the StateBackend abstractions provided by the system. Incremental computations over unbounded streams are evaluated over continously evolving logical views called windows. Flink incorporates windowing within a stateful operator that is configured via a flexiblle declaration composed out of three core functions: a window assigner which assigns records to windows, a optional trigger which defines when the operation associated with the window definition is performed and an optional evictor which determines which records to retain within each window.

A DataSet API is provided for batch computations, which is simplified and optimized in following ways: The runtime executable may be parameterized with blocked data streams to break up large computations into isolated stages that are scheduled successively; Periodic snapshotting is turned off when its overhead is high. Instead, fault recovery can be achieved by replaying the lost stream partitions from the latest materialized intermediate stream; Blocking operators use managed memory provided by Flink and can spill to disk if their inputs exceed their memory bounds; Flink serializes data into memory segments and uses type inference and custom serialization mechanisms to handle arbitrary objects. By keeping the data processing on binary representation and off-heap, Flink manages to reduce the garbage collection overhead, and use cache-efficient and robust algorithms that scale gracefully under memory pressure; A query optimization layer transforms a program into an efficient executable. It enumerates different physical plans based on the concept of interesting properties propagation, using a cost-based approach to choose among multiple physical plans and use hints provided by the programmer to overcome the cardinality estimation issues in the presence of UDFs.

Iterations in Flink are implemented as iteration steps, special operators that themselves can contain an execution graph. Flink allows for iteration head and tail tasks that establish an active feedback channel to the iteration step and provide coordination for processing data records in transit within this feedback channel. In DataStream API, to comply with fault-tolerance guarantees, feedback streams are treated as operator state within the implicit-iteration head operator and are part of a global snapshot. And in DataSet API, Flinks execution model allows for any type of structured iteration logic to be implemented on top, by using iteration-control events and Flink introduces further novel optimisation techniques such as the concept of delta iterations to exploit sparse computational dependencies.

## IV. Your Opinions

1) Both D-Steams and Flink implement a DAG based execution engine, provide an SQL optimizer, perform driver-based iterations, and treat unbounded computation as micro-batches.
2) Both D-Steams and Flink handle failover with checkpointing and partial re-execution, but with different techniques. D-Steams deal with stragglers, but Flink does not.
3) D-Steams provide stateless computations, while Flink provides stateful computations.
4) Both D-Streams and Flink serves both batch and streaming, and provide exactly-once consistency.
5) Flink supports all known window types while D-Streams only supports periodic time-window.
6) Both D-Steams and Flink handles out-of-order input. In Flink, it is handled in operator implementation, while in D-Streams, it can be handled in application level.
7) In D-Streams paper, the performance is measured, but other metrics such as memory/CPU usage is not displayed which might indicate other interesting trade offs in performance. And Flink paper can be improved by measuring performance including comparing the performance of DataSet API and DataStream API on batch processing.
8) Both papers have hyperlinks to the figures, notes and references.
9) In D-Streams paper, details on previous systems and future works are described and Flink paper can be improved on this.
10) D-Streams paper looks more formal with two columns format and Flink paper can be improved on this.
11) Flink paper states contributions explicitly and D-Streams paper can be improved on this.
12) Some abbreviations in Flink paper are not explained.