Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Apache Flink™: Streaming dataflows as a basis for universal data analytics

Authors

Abstract

Abstract TBD.

1. INTRODUCTION

Organizations rely more and more on the value of their data, which is directly correlated with its freshness. In addition, as the data infrastructure becomes more and more complex, development time is focused on integrating different systems together, rather than focusing on business logic. The ever-growing need for lower latency analytics and scale, creates a pressing need for more real-time, asynchronous, data processing infrastructure.

Various real-time analytics and CEP tools are currently in use by the industry, including TIBCO, IBM Infosphere Streams, Microsoft Streaminsight, Streambase, etc. These systems implement and extend the results of research which has been done in the database community in the last 15 years, with projects such as Aurora, Telegraph, STREAMS, etc. However, the era of Web-scale computing, spawned the development of internal "roll-your-own", solutions based on systems like Apache Storm, Samza, Kafka, Spark, etc. These systems provide very low level APIs and enforce only custom, application-specific solutions to be built on top. Moreover, in the lack of better solutions, the industry is using traditional batch processing platforms to serve streaming use cases. This has led to approaches such a "micro-batching" [], where a batch processor is used to simulate continuous processing, and the "lambda architecture", where a stream processor is aided by a batch processor to compensate for lack of performance and consistency guarantees.

We believe that there is still a need for a real-time analytics platform, that is not covered by the existing solutions described above. Our observation is that modern enterprises need a platform that integrates well in the Hadoop-related Big Data processing zoo, supports user-defined computations, while being able to perform efficient, scalable and consistent data analysis, on top of both streaming/unbounded and stored/batch data.

Apache Flink is a platform that was designed to deliver on these requirements. It is an open source project licensed by the Apache Software Foundation (ASF), with its community counting more than 120 developers, and growing fast []. Flink originates from the database research community, in particular the Stratosphere research project [], and incorporates several novel research contributions, some of which have been published in research venues in the past, and some of which have only been published in the form of blog posts or wiki pages. In this article, we attempt a short description of the architecture of the system, highlighting its novel aspects.

This paper is organized as follows: Section X presents a novel paradigm for distributed dataflows based on the notions of operators and logical intermediate results. Section Z presents a novel mechanism to provide fault tolerance while keeping the latency low and the throughput high. Section Y presents methods to overcome limitations of the JVM by careful engineering, and how to while extend cost-based optimization to user-defined functions [socc10, vldb12] and efficiently embed iterations within a DAG-based dataflow system [vldb12].

2. SYSTEM ARCHITECTURE

In this section we lay out the architecture of Flink as a software stack, and as a distributed system. While Flink's stack of APIs continues to grow, we can distinguish four main layers: deployment, core, APIs, and libraries.

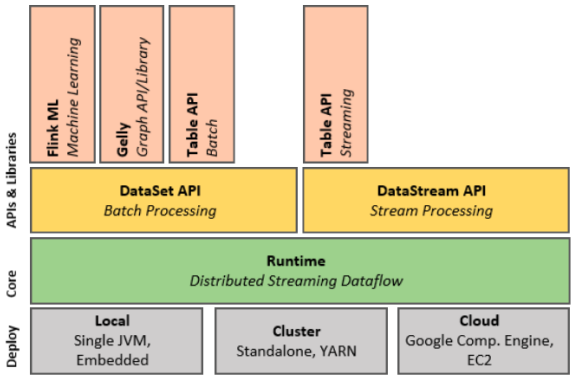

Figure 1: The Flink software stack.

Flink's Runtime and APIS. The core of Flink is the distributed dataflow engine, which executes dataflow programs called Job Graphs. A Job Graph is a DAG of operators and connections between operators. There are two "core" APIs in Flink:

the DataSet API for processing finite data sets (often referred to as "batch processing"), and the DataStream API for processing potentially unbounded data streams (often referred to as "stream processing"). Flink's core runtime engine can be seen as a streaming dataflow engine, and both the DataSet and DataStream APIs create programs (Job Graphs) accepted by this engine. On top of the core APIs, Flink bundles "domain-specific" libraries and APIs that generate DataSet and DataStream API programs. Currently these are the following: i) Gelly, an API and library for processing graphs, ii) FlinkML, an API and library for composing Machine Learning pipelines and iii) Table, an API similar in spirit to Microsoft's LINQ.

A Flink cluster comprises of three types of processes: the client, the JobManager, and the TaskManager. The client takes the program code (written in a mixture of Flink's APIs), transforms it to a Job Graph, and submits it to the JobManager. The compilation process involves a type extraction and checking phase that generates serializers and comparators for all used types. DataSet programs, also go through a cost-based query optimization phase, similar to the physical optimizations performed by relational query optimizers (more details in Section X).

The JobManager is Flink's master node. It coordinates all message-passing during job execution by sending heartbeats to the TaskManagers, receiving statistics, controls the tasks' lifecycle, coordinates the fault tolerance machanisms. The actual data processing takes place in the TaskManagers, or worker processes that they execute several tasks in multiple threads, and maintain data structures shared by all tasks (e.g., buffer pools) executed by a TM. TMs communicate directly with each other using a multiplexed TCP connection per TM pair.

3. EXECUTION MODEL

The Job Graph. Flink's execution model is based on the Job Graph, a directed acyclic graph (DAG) that consists of nodes and edges. There are two classes of nodes: (stateful) operators, and (logical) intermediate results (IRs). For example, the graph below consists of five operators (op_1-op_5), and three intermediate results (ir_1-ir_3). Operators abstract computation (e.g., transformations, joins, etc), state (e.g., a persistent counter), as well as data sources (e.g. reading data from a file system, a socket, a message queue, etc.), and data sinks. Operators produce intermediate results, as well as updates to state. An intermediate result is a logical handle (pointer) to the data that is produced by one operator. An intermediate result can be consumed by one or more operators. Intermediate results are logical in the sense that the data they point to may or may not be materialized on disk.

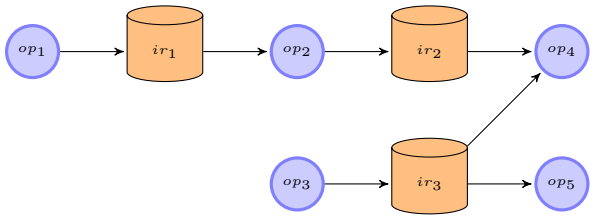

Figure 2: The JobGraph is the logical view of a Flink job.

Parallelized Job Graphs. When the JobGraph is scheduled in a cluster for execution, it is parallelized to form an ExecutionGraph. An ExecutionGraph consists of tasks (parallel instances of operators), and intermediate result partitions (IRPs), that represent logical partitions of intermediate results corresponding to different partitioning keys. The JobGraph in fig. 3, if it is partitioned using a parallelism of two, it results to the ExecutionGraph shown in fig. 3. Assume that op_1 and op_2 are data sources, op_2 is a map operator, op_4 is a join operator that needs both inputs partitioned by the same key, and op_5 is a filter. Assume also that the join is executed as a repartitioned join, i.e., both inputs are partitioned and shuffled over the cluster of 2 nodes (alternatives would be to broadcast one input or leverage a pre-existing partitioning if present). The resulting ExecutionGraph is the following:

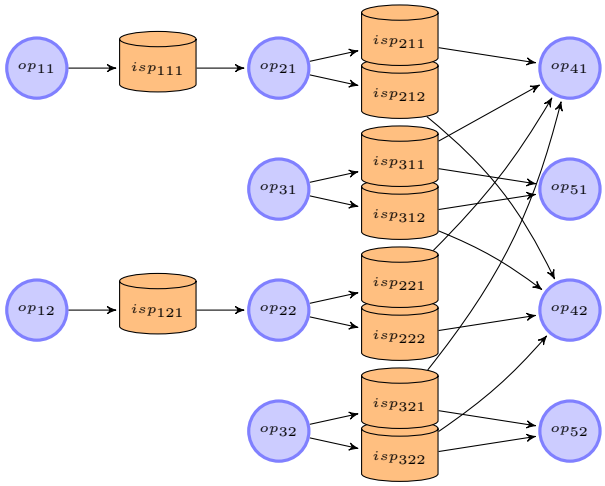

Figure 3: The ExecutionGraph is the physical view of a Flink job.

Data Transfer. The unit of data transfer in the Flink runtime is a buffer. Buffers contain one or more records, and a record can span multiple buffers. Buffers are requested and relinquished from local buffer pools, shared among operators that live in the same task manager. An IRP is simply a collection of buffers. Work in Flink progresses (i.e., records flow through the pipeline) as long as there are buffers available, essentially implementing distributed blocking queues (the logical streams) with bounded capacity (the amount of memory available to the buffer pools, which can be configured by the user). This mechanism, in addition to implementing record network transfers doubles down as a natural way to backpressure the flow in the case of slow operators (including external systems that consume data).

We mentioned that the intermediate results are logical handles to the data, rather than the data itself. Internally, intermediate results are abstract classes with many implementations. These implementations can perform pipelined data exchange, and blocking data exchange.

Pipelined Data Exchange. Pipelining (also called intra-operator parallelism), means that a producing and a consuming operator make progress at the same time, without the consumer waiting for the producer to finish. Pipelining is required in streaming systems, and is also used in batch systems to reduce latency. Flink implements pipelining by implementing intermediate results which activate network transfers between the producer task and consumer task, as soon as their first buffer is available. Flink allows the configuration of its buffers' size and timeout. A buffer is sent to its consumer task as soon it is filled, or as soon as a timeout is reached. Hence, by setting the buffer size and/or time-

out to lower values, one can reduce latency, while achieving the opposite effect (i.e., high throughput) by using higher values.

Blocking Data Exchange. Sometimes it is desirable to break a job into stages, scheduling and executing each stage individually (e.g., to enable interactive processing, and better staging of resources in large batch jobs). To do that, the system has to materialize intermediate results (in memory or disk). Flink implements blocking data exchange via an intermediate result that signals its availability only when all the buffers from the producer have been materialized. The cached buffers can double down as a materialized reusable intermediate result.

4. FAULT TOLERANCE VIA ASYNCHRONOUS SNAPSHOTS

Fault tolerance in Flink is achieved by taking a snapshot of the execution graph at regular intervals. When failure occurs the state of the execution is restored from the latest snapshot. Upon recovery, in order to guarantee exactly-once processing semantics, the records (events) consumed since the latest snapshot are reprocessed in the same order at each respective source. Normally this possible by using persistent message queueing systems, e.g. Kafka.

Flink's core mechanism for distributed snapshots is called "Asynchronous Barrier Snapshotting" (ABS) [arXiv paper]. It creates a global snapshot by collecting the state of each task in the execution graph in an asynchronous manner. It does so by superimposing snapshotting using injected markers, called "barriers", in the input data stream. Similar to Chandy-Lamport distributed snapshots [Chandy/Lamport], the barriers in ABS dictate the records that should be fully processed before each respected global snapshot, however, no records in transit are included in a snapshot, keeping the persisted state at a minimum. ABS achieves this by a special "aligning" phase which ensures that all records from all upstream tasks preceding broadcasted barriers have been fully consumed before processing the data stream further. Recovery from failure reverts the execution graph to the most recent snapshot and is fully consistent by respecting the causal dependency of records. Furthermore, ABS [arXiv paper] supports snapshotting of cyclic graphs. The operation of ABS is fully decoupled from the backend used for persisting states, thus, allowing multiple different backend implementations to be integrated. The following figure illustrates the snapshot process.

5. STREAM ANALYTICS ON TOP OF DATAFLOWS

Flink's DataStream API implements a full stream analytics framework on top of Flink's runtime, including the mechanisms to manage time (including out-of-order event processing), defining windows, and maintaining and updating user-defined state. The streaming API is based on the notion of a DataStream, a (possibly unbounded) immutable collection of elements of a given type. Since the Flink runtime already has pipelined data transfers, continuous stateful operators, and a fault tolerance mechanism for consistent state updates, overlaying a stream processor on top of it essentially boils down to implementing a windowing system and a state interface.The programming model of Flink's streaming API, builds on two basic abstract data types, namely data streams and window streams. Each of the two supports specific operations since they exhibit different properties. For example, windows support transformations that are only possible on bounded collections such as joins, groupings and aggregations. Other transformations, such as (flat)map are can be applied incrementally on data streams.

The API is designed with simplicity in mind, while at the same time providing powerful tools to deal with time and uncertainty. Here is, for example, a word count program on a simple time-based window:

Listing 1: WindowWordCount Scala implementation

```
object WindowWordCount {
  def main(args: Array[String]) {

    val env =
         StreamExecutionEnvironment.getExecutionEnvironment
    val text = env.socketTextStream("localhost", 9999)

    val counts = text
    .flatMap {line =>
          line.toLowerCase.split("\\W+")
              .filter { w => w.nonEmpty }}
    .map {w => (w, 1) }
    .keyBy(0)
    .timeWindow(Time.of(5, TimeUnit.SECONDS))
    .sum(1)

    counts.print
    env.execute("Window Stream WordCount")
  }
}
```

Flink distinguishes between two notions of time:

Event time is the time that an event actually happened (e.g., the time that a sensor emitted a signal, or the time that a person tapped on their smartphone).

Processing time is the wall-clock time of the machine that is processing the data.

In distributed systems, there is an arbitrary lag between event-time and processing-time [dataflow]. This may mean arbitrary delays for getting an answer based on event-time semantics. To avoid arbitrary delays these systems regularly insert special events called "low watermarks" that include a time attribute t indicating that all events lower than t have already entered the system. The watermarks aid the execution engine to process events in the correct event order.

Flink programs that are based on processing time, rely on the machine clocks, and hence a less reliable notion of time, but exhibit the best latency. Programs that are based on event time provide the most reliable semantics, but may exhibit latency due to event time-processing time lag. Flink includes a third notion of time as a special case of event time called ingestion time, which is the time that events enter the system. This provides a lower latency than event-time semantics, and avoids the arbitrary processing time semantics.

Streaming Windows. A window defines a logical group of records that are processed together. Flink's windowing system largely follows the Dataflow model [Dataflow] proposed by Google. A window definition consists of three building blocks: a window assigner, optionally a trigger, and optionally an evictor. The assigner predefines the logical groups to which each record belongs, e.g. count-based or time-based periodic windows. The trigger defines when the window operation is performed on each group. The evictor defines which records to keep in each group. The following example defines windows that operate in event

time of 6 secondst hat slide every 2 seconds (Assigner). The window results are computed once the watermark passes the end of the window (Trigger). Flink also offer shart hands for commonly used functionalities.

Listing 2: Window API functionalities

```
//general window API
stream
  .window(SlidingTimeWindows.of(Time.seconds(6),
       Time.seconds(2))
  .trigger(TimeTrigger.create())

//equivalent short hand for common use case
stream.timeWindow(Time.seconds(6), Time.seconds(2))
```

A global window creates a single logical group. The following example defines a global window (assigner) that invokes the operation on every 1000 events (trigger) while keeping the last 100 elements (evictor).

Listing 3: Global Window usage

```
stream
  .window(GlobalWindow.create)
  .trigger(Count.of(1000))
  .evict(Count.of(100))
```

Note that streams are already partitioned on a key before windowing, so the above is a local operator that does not require coordination between machines. This mechanism can be used to implement a wide variety of windowing functionality [Dataflow], and Flink comes bundled with syntactic sugar for the most common window definitions (e.g., time- and count-based windows).

Simple programs in Flink's DataStream API looks like functional, side effect-free programs consisting of transformations on unbounded immutable collections. We incorporate mutable state in the API by providing interfaces to users to register any local variable within a transformation with the system's checkpointed mechanism and freely use this variable in their code.

6. BATCH ANALYTICS ON TOP OF DATAFLOWS

A bounded data set is a special case of an unbounded data stream. Thus, a streaming program that inserts all of its input data in a window can form a "batch" program and "batch analytics", or "batch processing" should be fully covered by Flink's features that we presented above. However, i) the syntax, i.e., the API for batch computation can be simplified (not need for window definitions, simpler joins and loops) and ii) we can simplify the fault tolerance mechanisms and iii) we can apply query optimization borrowing ideas from MPP database systems.

For these reasons, Flink treats specially batch computations, by implementing the above optimizations. For enabling batch computations on top of it's streaming runtime, Flink embeds blocking versions of its operators (sorts, joins, etc) within its runtime. These operators simply block until they have received all of their input. Moreover, Flink currently disables the asynchronous snapshotting mechanism for batch programs, and simply uses backtracking based recovery as first described in Dryad [Dryad].

With these established, we look that the batch-specific optimizations mentioned above: query optimization, and query processing on paged (managed) memory.

Query Optimization. Flink's optimizer builds on techniques from parallel database systems such as plan equivalences, cost models and interesting properties. However, the arbitrary UDF-heavy DAGs that make up Flink's dataflow programs, do not allow a traditional optimizer to employ database techniques out of the box [black-boxes], since the operators hide their semantics from the optimizer. For the same reason, cardinality and cost estimation methods are equally difficult to employ. Flink's optimizer employs a number of novel methods for overcoming these issues [blackboxes, stratosphere-journal, iterations] for which we provide a short overview below. Flink's runtime supports various execution strategies including repartition/broadcast data transfer, as well as sort-based grouping and sort- and hash-based join implementations. Flink's optimizer enumerates different physical plans based on the concept of interesting properties propagation [scope-optimizer], using a cost-based approach to choose among multiple physical plans. The cost includes network/disk I/O and CPU cost. To overcome the cardinality estimation issues that were mentioned earlier, Flink's optimizer uses hints that are provided by the programmer.

Memory Management. Building on database technology, Flink, instead of storing objects in the JVM's heap, serializes objects into a memory segments. These memory segments resemble database blocks into which, java objects representing the tuples that go through the runtime are serialized. Operations such as sorting, and joining, operate as much as possible on the binary data, keeping the de/serialization overhead at a minimum and partially spilling data to disk when needed. To handle arbitrary objects, Flink uses type inference, and custom serialization mechanisms. By keeping the data processing on binary representation and off-heap, Flink manages to reduce the garbage collection overhead, and implement cache-efficient and robust algorithms that scale gracefully in under memory constraints.

Native Iterations on top of Dataflows. The final aspect of Flink on which we focus, is how to implement loops on top of the Flink's dataflow engine. Some approaches execute iterations by submitting new jobs for each iteration or by adding additional nodes to a running DAG [Hadoop iterative works, Spark], hiding from the engine that it is executing an iterative program. The approach, implemented in Naiad [] adds feedback edges in the dataflow graph, supporting graphs with cycles, that allow for nested iterations. A third approach was to design specialized engines around iterative processing along (e.g., Giraph, and GraphLab) allowing to reduce the number of computations in each iteration [].

Flink follows an approach that maintains the DAG-based runtime, but allows for special "head" and "tail" tasks to signify the beginning and end of iteration, that exchange data via shared memory. This approach simulates a cyclic dataflow within a DAG engine making Flink i) competitive with specialized graph engines [], and outperforms the driver-based approach []. Flink supports delta iterations [], which exploit sparse computational dependencies, and are used, as the basis for Gelly, Flink's Graph API. Finally, Flink offers Bulk Synchronous Parallel (BSP) iterations in its DataSet API, and asynchronous iterations in its DataStream API.

7. RELATED WORK

There is, by now, a wealth of engines for distributed batch and stream analytical processing. We categorise these systems below.

Batch Processing. Apache Hadoop [] is the most popular open source system for large-scale data analysis that is based

on the MapReduce paradigm []. Dryad [], introduced embedded user-defined functions in general DAG-based dataflows and was used by SCOPE [] which added a language and an SQL optimizer on top of it. Apache Tez [] can be seen as an open source implementation of the ideas proposed in Dryad. MPP databases [], and recent open-source implementations [Apache Drill, Impala] restrict their API to SQL variants. Very similar to Flink, Apache Spark [] is a very popular data processing framework that implements a DAG-based processing framework, provides an SQL optimizer, performs driver-based iterations, and treats stream computations as micro-batches. In contrast, Flink is the only system that i) sports an optimizer that can optimize DAG programs beyond SQL queries, ii) is able to perform very efficient iterative processing natively, iii) includes stream processing as a first-class citizen, enabling more complex use cases than micro-batches.

Stream Processing. There is a wealth of work on streaming systems, that includes commercial systems like StreamBase, Microsoft StreamInsight, and IBM InfoSphere Streams. Many of these systems are based on research in the database community in projects such as Telegraph [], Aurora/Borealis [], Stanford STREAMS [], Trill [], and IBM System S []. Most of the above systems are either i) academic prototypes, ii) closed-source commercial products, or iii) do not scale the computation horizontally on clusters of commodity servers. Newer open source streaming systems that scale horizontally, such as Apache Storm and Apache Samza, provide very low level APIs and offer only at-least-once and at-most-once guarantees. MillWheel [], a closed-source system at Google provides exactly-once guarantees with low latency and is used by Google Dataflow [] to implement out-of-order stream analytics. To the best of our knowledge, Flink is the only open-source project that i) offers high level APIs and processing libraries, ii) provides state management with exactly-once guarantees and iii) achieves high throughput and low latency, serving both batch and true streaming computations equally well.

8. CONCLUSION

In this paper we presented Apache Flink, a platform that implements one universal dataflow engine designed to perform both stream and batch analytics. Flink's dataflow engine treats operator state and logical intermediate results as first class citizens, and is used by the batch and a data stream APIs. The streaming API that is built on top of Flink's streaming dataflow engine, provides the means to keep recoverable state, and to partition, transform, discretize and aggregate a data stream. While batch computations are, in theory, a special case of a streaming computations, Flink treats them specially, by optimizing their execution with a novel query optimizer, and by implementing blocking operators that gracefully spill to disk in the absence of memory.

9. REFERENCES