Advanced Topics in Distributed Systems

Critical Review by Klas Segeljakt

April 8th 2020

This is a review of the papers I presented in the *Advanced Topics in Distributed Systems* course.

Instructions

For reference, the instructions for the critical review are:

> "Task 4: write a critical review of the papers that covers in particular the summary of contributions, solutions, significance, and technical/experimental quality." - Course instructions

Introduction

Distributed systems are systems which distribute state and computation among networked computers. The two pivotal reasons behind building distributed systems are scalability and reliability. More concretely, distributed systems achieve performance by scaling the number of computers against the problem size, and tolerance to failure by replicating and persisting state.

A sub-category of distributed systems are distributed data analytics systems - systems which optimize towards processing data at high throughput and low latency. These systems further subdivide into categories which depend on how the data is represented. At one axis, data can either be batch or streaming. Broadly, streaming systems achieve low-latency by processing data as soon as it arrives, while batch systems achieve high-throughput by processing data in bulks.

From another angle, data can also either be mutable or immutable [1]. For the former, computation is represented as cyclic or bidirectional graphs of processes which communicate

[1] Huang, Y., Yan, X., Jiang, G., Jin, T., Cheng, J., Xu, A., Liu, Z. and Tu, S., 2019. *Tangram: Bridging Immutable and Mutable Abstractions for Distributed Data Analytics.* In 2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19) (pp. 191-206).

by sending messages over channels. Examples include machine learning systems such as TensorFlow, where workers push and pull information from parameter servers, and vertex-centric graph analytics systems which scatter and gather vertex states. Conversely, systems with immutable data abstractions model computation as directed acyclic graphs, known as dataflow-graphs. Within these systems, data is flowing from sources to sinks, and through operators that apply natural transformations. For acyclic computational graphs, distributed iterative computation is not possible. This heavily limits the variety of workloads which can be executed. Therefore, dataflow systems typically allow loops to exist in the computational graph under tight constraints. On the other hand, by keeping the execution model simple, dataflow systems are capable of straggler mitigation through speculative execution, and efficient failure recovery.

Focusing on failure recovery, systems widely adopt different strategies, such as:

- Checkpoint-based solutions, which persists (snapshots) the complete state of the system at regular intervals, e.g. after having processed a mini-batch.
- Lineage-based solutions, which persists state partitions of individual nodes in the computational graph, allowing more fine-grained recovery.
- Replication-based solutions, where system state is kept either consistent or available by storing backups on multiple nodes.
- Message-replay solutions, where messages are logged and replayed on failure.

For failure recovery to be sound, it must satisfy the notion of *exactly-once semantics*, which further implies satisfying both *exactly-once processing* and *exactly-once delivery*. In the presence of failure, exactly-once processing means the system will eventually recover into a state that captures the effect of having processed every received message exactly-once. For exactly-once delivery, the system also needs to ensure that every message delivered to a sink is delivered exactly-once. In other words, the system must not only make consistency guarantees about its own state after recovery, but also ensure that the state of the outside world is not affected.

For the review, three papers were selected which in different ways address the problem of failure recovery in distributed immutable data stream analytics. The coming sections will cover the problem, solution, contributions, significance, experiment- and technical-quality of each paper.

Flink

The first paper to be reviewed is:

- Carbone, P. et al., 2017. State management in Apache Flink®: Consistent Stateful Distributed Stream Processing. Proceedings of the VLDB Endowment, 10(12), pp.1718-1729.

Problem

As of the time the Flink paper was written, systems for stream processing lacked state abstractions. For an application to contain stateful computation, users would either need to manually depend on an external database management system, or use micro batching. The former approach introduces operational challenges, since security, scaling, and code updates need to be managed between the internal and external system, while the latter adds processing latency. Between both approaches, the common denominator is that state management is separated from data processing. The question addressed in the Flink paper is if state management could instead be interleaved with processing to reduce both the operational costs and processing latency. Flink's solution is a global snapshotting protocol based on the Chandy-Lamport's consistent snapshots protocol, but with weakened assumptions.

Chandy Lamport works by propagating markers between processes. When a process receives a marker for the first time, it must snapshot its internal state, begin recording in-flight messages on its (other) input channels, and forward the marker to each of its output channels. On additional markers, the process stops recording the receiving input channel. On system failure, processes restore their snapshotted state and messages, and resume normal execution. In cyclic graphs, safety (causal validity) requires that there is a single initiating process. Messages, and corresponding state updates, which supersede the marker sent by the process are not part of the snapshot. Liveness (termination) demands that channels are reliable, FIFO-ordered, and that the graph is strongly-connected. In other words, markers will eventually reach all processes on all input channels.

Solution

Flink relaxes Chandy Lamport's assumptions, allowing graphs to be weakly connected and consequently have multiple initiating processes. However, channels must be blockable and sources replayable. The idea of Flink's solution is to inject markers into the sources of the dataflow graph which cut the stream into epochs. When an operator receives a marker on an input channel, it blocks its output channels until all (other) input channels have received markers. As soon as all markers are received, the operator forwards one to each of its output channels, persists its local state and unblocks. On failure, the system rolls back its state to the epoch most recently committed by the sinks, and sources replay messages from that point onward. When cycles are present in the dataflow graph, Flink falls back to the Chandy

Lamport protocol. For Chandy Lamport's assumptions to hold, cycles must have an explicit head that acts as the single-initiating process of the cyclic subgraph. Data stream elements flow back to the head through an explicit iteration tail to guarantee strong connectivity.

In terms of safety guarantees, blocking is what guarantees exactly-once processing since messages from future epochs will not make it into the snapshotted state. However, blocking can optionally be disabled to allow for less expensive at-least-once processing. For exactly-once delivery, either idempotent or transactional sinks are needed. Apart from the algorithm definition, the paper also details how Flink supports pluggable state backends, queryable state, and asynchronous-incremental snapshots.

Contributions and Significance

In summary, the main contributions of the paper are:

- An overview of Flink's end-to-end design.
- A formal definition of Flink's consistent snapshot algorithm.
- A discussion about the algorithm's correctness, applications, and optimizations.
- A case study of Flink at King coupled with an evaluation of the algorithm.

The paper was published in 2017 and has since garnered 94 citations, with the most notable one being Ray [2]. Also, the original paper as of now stands at 784 citations. It is unclear to what degree Flink's snapshotting algorithm has inspired other work, but the system itself is highly influential. Additionally, since the paper was written, the protocol has been tweaked and optimized.

Technical and Experimental Quality

The paper is well written and uses both intuitive figures and pseudocode to explain how the system and protocol works. The only difficult part to read was section 4.1 *State Backend Support*, and section 4.2 *Asynchronous and Incremental Snapshots* which required background knowledge to grasp. The experiments first evaluate how different factors affect the snapshotting latency of the proposed algorithm under realistic workloads. Since snapshots are asynchronous, latency becomes independent of the global state size. The sole factor which impacts latency is alignment time, which for an operator is proportional to its number of input channels. While the results do not include any indication of significance (e.g. standard deviation or confidence interval), the experiments themselves are reproducible.

[2]Moritz, P. et. al., 2018. *Ray: A Distributed Framework for Emerging {AI} Applications.* In 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18) (pp. 561-577).

IBM Streams

The second paper to be reviewed is:

- Jacques-Silva, G. et al., 2016. Consistent regions: Guaranteed Tuple Processing in IBM Streams. Proceedings of the VLDB Endowment, 9(13), pp.1341-1352.

Problem

Global Snapshotting protocols such as Flink's provide end-to-end safety guarantees, i.e. guarantees that apply globally to the complete stream processing pipeline. Depending on the use-case, an application may not need stringent consistency guarantees for all its operators and channels. This is a concern, since snapshotting imposes overhead. Applications which rely on global snapshots might be paying for more than what they need. A motivating example from the paper is a streaming application which correlates banking transactions with data from Twitter feeds to predict marketing opportunities. Whereas every transaction must be correlated exactly-once, SLA:s are not breached if a tweet is processed more, or less, than once. IBM Streams asks the question whether different subgraphs of the dataflow graph can have different safety guarantees.

Solution

The solution proposed by IBM Streams is to create a Stream Processing Language (SPL), and C++/Java framework alongside a runtime API, that gives more control over how dataflow graphs are constructed. In SPL, users define and connect custom operators to form a dataflow graph. Operators can be annotated as either `@consistent` or `@autonomous` to define their safety guarantees. The annotations respectively indicate the start and end of *consistent regions*. A consistent region is a subgraph of the dataflow graph which is snapshotted. Hence, sources, i.e. entry-nodes, in consistent regions replay messages on failure. Conversely, sources in *autonomous-regions*, i.e. regions which are not snapshotted, never replay messages. The effect is that any operator within a consistent region will process messages received from other operators in the region exactly-once. Messages from sources in autonomous regions may be lost, and are therefore processed at-most-once. Messages from sources in consistent regions may be processed at-least-once unknowingly by downstream operators in autonomous regions. SPL hence gives the user the choice to decide between exactly-once, at-least-once, and at-most-once processing at the granularity of individual operators.

To handle cyclic dataflow, SPL introduces the concept of ports. Channels connect with operators through input and output ports. Input ports are modeled as either data ports, or control-ports. Whereas a control-port may only mutate internal state, a data-port may in addition output new messages. The idea is to model feedback loops with control-ports. Control-ports guarantee that messages which flow upstream do not flow back downstream again. In other words, the dataflow graph becomes acyclic, and can be snapshotted with weaker assumptions than Chandy Lamport.

The snapshotting algorithm is initiated by a *consistent region controller* - a process which sends snapshot notifications to each of the sources in the region to be snapshotted. The sources proceed to 1) drain in-flight messages stored within the operator (e.g. buffers) 2) block outgoing channels, 3) persist local state, and 4) forward *drain markers*. When an operator receives a drain marker, it will 1) drain in-flight messages, 2) block outgoing channels, block ingoing channels from autonomous regions, 3) forward drain markers, 4) wait for drain markers to arrive on all control ports, 4) persist local state, and finally 6) notify the consistent region controller of termination. Like Chandy Lamport, the protocol relies on the assumption that channels are FIFO ordered, and that drain markers are not forwarded by failing operators (byzantine errors). Additionally, sources must be replayable, and sinks must be able to retract messages after having written them to an external system.

While the algorithm temporarily shuts down processing within the consistent region, other regions in the dataflow graph are not halted. Although, what the paper does not emphasize is how poorly the algorithm performs in the case where the whole graph is a consistent region, which would require halting the entire pipeline. The argument that Flink does not support fine-grained fault tolerance which the paper makes is not entirely true. Users could potentially connect multiple Flink pipelines with mixed levels of safety to accomplish the same goal as IBM streams.

Contributions and Significance

The main contributions of the paper are:

- A Stream Processing Language (SPL) which models pipeline safety properties through consistent regions, and feedback-loops through ports.
- A protocol for distributed snapshots which can be executed on individual regions of the dataflow graph.
- An evaluation of the protocol's performance and correctness, under realistic workloads.

As of now, the paper has 18 citations, of which Samza [3] and Flink [4] are the most discernible. Unlike Flink, IBM Streams is proprietary work, and as a result, the paper does not cover much of the implementation details. After having read the paper, it is difficult to tell how the framework, runtime, and the underlying IBM Streams platform works. For this reason, I think the work is less significant when compared to its open source counterparts.

[3] Noghabi, S.A. et al. 2017. Samza: stateful scalable stream processing at LinkedIn. Proceedings of the VLDB Endowment, 10(12), pp.1634-1645.

[4] Carbone, P. et al. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. Proceedings of the VLDB Endowment, 10(12), pp.1718-1729.

Technical and Experimental Quality

The paper is for the most part well written. The main weak point is the lack of figures. There is for example no figure present which explains the concept of ports, the architecture of the system, nor the abstract syntax of SPL. For the experimental evaluation, the paper first perform a large set of unit-tests to validate the correctness of the algorithm. The remaining experiments test the throughput against the snapshotting frequency, snapshotted state size, and topology complexity. Unsurprisingly, throughput degrades under high snapshotting frequencies and elaborate topologies. As in Flink, throughput is not impacted by the state size as long as the state is persisted asynchronously.

Spark Structured Streaming

The final paper to be reviewed is:

- Armbrust, M. et al., 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).

Problem

As mentioned in the introduction, systems for data analytics are at large distinguished by the type of data they process. Systems such as IBM Streams are solely focused on processing streaming data whilst others try to retrofit one data abstraction onto the other. This is non-trivial, because each data type comes with a different set of operations. For example, rewriting a query over batch data to be compatible with streaming data might insist incrementalizing expressions. Users consequently need to think about how queries are executed, which takes focus away from the business logic. As the paper notes, there is a need in modern day analytics to not only retrofit, but also combine operations over streaming and batch data in the same pipeline, e.g. joins and interactive queries. Meanwhile systems must also be capable of handling failures, migrating between clusters, graceful shutdown, code updates, straggler mitigation, and dynamic rescaling. The paper further claims that throughput takes precedence over latency in large-scale streaming applications as latency sensitive applications should ideally be run on scale-up architectures.

Solution

As a response to the aforementioned problems, the paper presents Spark Structured Streaming - a new high-level API in Apache Spark for batch and stream processing. Programs in Structured Streaming are written as either SQL queries, or operations on Spark DataFrames and DataSets, and are independent of whether the data source is stream or batch. The core component behind this achievement is an “incrementalizer” subsystem which rewrites batch operations into equivalents which work on streaming data.

Additionally, Structured Streaming comes with two execution modes. Streaming queries can be executed either as minibatches, akin to Spark Streaming’s discretized streams, or through a new mode known as Continuous Processing. The former mode optimizes for dynamic load balancing, fine-grained failure recovery, straggler mitigation, rescaling, and high-throughput, while the latter favors low-latency. The most interesting prospect of the paper is how the two modes can be used together in the same application. The first example is that streaming applications often need to execute large window computations multiple times per day. As cloud providers charge by the minute, keeping the streaming system fully awake during the off-hours lead to lost profits. In Structured Streaming, streaming windows can instead be executed as batch-jobs for cost savings. The second example is to use the minibatch execution mode adaptively in cases where the system needs to catch-up with the backlog, e.g. after failure-recovery or during spikes in user-activity. Although continuous processing appears promising, there are three major limitations which sets it back compared to other engines: 1) cyclic dataflow is not possible, 2) all sinks need to be idempotent for exactly-once delivery, and 3) the current version of does not support shuffle operations.

Structured Streaming's failure-recovery protocol is checkpoint-based. Spark's master node periodically cuts the input into epochs. In minibatch execution, each epoch is partitioned and executed by multiple short-running tasks. Inversely, tasks in continuous processing are long-running and execute multiple epochs per task. Whenever an epoch enters or leaves the system, it is logged by the corresponding source or sink operator. Stateful operators, such as aggregators, asynchronously backup their internal state. On failure, operator state is rolled back to the most recent checkpoint. Because backups are asynchronous, epochs may have been committed by sinks after operator state was checkpointed. Therefore, the system replays any epochs committed after the checkpoint while disabling sink outputs. Normal execution is resumed when the final, uncommitted, epoch has been replayed (with sink outputs enabled).

Contributions and Significance

The core contributions of the paper are:

- A declarative API which unifies batch and stream processing.
- An incrementalizer which is able to rewrite programs written on streams into ones on batch, and vice versa.
- A checkpointing algorithm.
- Two execution engines which optimize for different workloads.

The paper currently has 44 citations, with [5] being the most significant. Spark is by itself possibly the most influential highly data analytics system currently available with 5010 citations as of now. The solution the paper proposes has limitations but contains ideas which will likely influence other systems that try to unify the batch and streaming paradigm.

Technical and Experimental Quality

The paper has a good structure which explains with clarity what problems the system can solve and what the contributions are. More details could have been put however into describing how the system works. There are for example only few hints given about how the continuous processing mode functions.

The first experiment in the evaluation benchmarks Structured Streaming's performance in minibatch mode against that of Flink and Kafka Streams with the Yahoo! Streaming Benchmark (YCSB). Boldly, the paper claims that Structured Streaming can outperform Flink by up to 2x in throughput. There is however no mention about how they compare

[5] Traub, J. et al. 2019. Efficient Window Aggregation with General Stream Slicing. In EDBT (pp. 97-108).

in latency. The second experiment measures is how throughput scales with the number of nodes in YCSB and indicates almost a linear scaling from 1 to 255 cores. The final experiment investigates how continuous processing compares with minibatch execution on Extract Transform Load (ETL) workloads. Continuous processing has lower latency than minibatch execution for mid-level workloads. Surprisingly, continuous processing also has a higher maximum attainable throughput than minibatch mode. Conclusively, the experiments do not show indications of statistical significance, but seem reproducible since they use publicly available benchmarks.