Managing Large State in Apache Flink®: An Intro to Incremental Checkpointing

January 23, 2018 Apache Flink, Flink Features

Stefan Richter

Apache Flink was purpose-built for *stateful* stream processing. Let's quickly review: what is state in a stream processing application? I defined state and stateful stream processing in a previous blog post, and in case you need a refresher, *state is defined as memory in an application's operators that stores information about previously-seen events that you can use to influence the processing of future events.*

State is a fundamental, enabling concept in stream processing required for a majority of interesting use cases. Some examples highlighted in the Flink documentation:

- When an application searches for certain event patterns, the state stores the sequence of events encountered so far.
- When aggregating events per minute, the state holds the pending aggregates.
- When training a machine learning model over a stream of data points, the state holds the current version of the model parameters.

However, stateful stream processing is only useful in production environments if the state is fault tolerant. "Fault tolerance" means that even if there's a software or machine failure, the computed end-result is accurate, with no data loss or double-counting of events.

Flink's fault tolerance has always been a powerful and popular attribute of the framework, minimizing the impact of software or machine failure on your business and making it possible to guarantee exactly-once results from a Flink application.

Core to this is checkpointing, which is the mechanism Flink uses to make application state fault tolerant. A checkpoint in Flink is a global, asynchronous snapshot of application state and position in the input stream that's taken on a regular interval and sent to durable storage (usually a distributed file system). In the event of a failure, Flink restarts an application using the most recently-completed checkpoint as a starting point.

Some Apache Flink users run applications with gigabytes or even terabytes of application state. These users have reported that with such large state, creating a checkpoint was often a slow and resource intensive operation, which is why in Flink 1.3 we introduced a new feature called 'incremental checkpointing.'

Before incremental checkpointing, every single Flink checkpoint consisted of the full state of an application. We created the incremental checkpointing feature after we observed that writing the full state for every checkpoint was often unnecessary, as the state changes from one checkpoint to the next were rarely that large. Incremental checkpointing instead maintains the differences (or 'delta') between each checkpoint and stores only the differences between the last completed checkpoint and the current application state.

Incremental checkpoints can provide a significant performance improvement for jobs with a very large state. Implementation of the feature by a production user with terabytes of state shows ***a drop in checkpoint time from more than 3 minutes per checkpoint down to 30 seconds per checkpoint after implementing incremental checkpoints***. This improvement is a result of not needing to transfer the full state to durable storage on each checkpoint.

How to Start

Currently, you can only use incremental checkpointing with a RocksDB state backend, and Flink uses RocksDB's internal backup mechanism to consolidate checkpoint data over time. As a result, the incremental checkpoint history in Flink does not grow indefinitely, and Flink eventually consumes and prunes old checkpoints automatically.

To enable incremental checkpointing in your application, I recommend you read the the Apache Flink documentation on checkpointing for full details, but in summary, you enable checkpointing as normal and also enable incremental checkpointing in the constructor by setting the second parameter to `true`.

Java Example

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

env.setStateBackend(new RocksDBStateBackend(filebackend, true));
```

Scala Example

```
val env = StreamExecutionEnvironment.getExecutionEnvironment()

env.setStateBackend(new RocksDBStateBackend(filebackend, true))
```

By default, Flink retains 1 completed checkpoint, so if you need a higher number, you can configure it with the following flag:

```
state.checkpoints.num-retained
```

How it Works

Flink's incremental checkpointing uses RocksDB checkpoints as a foundation. RocksDB is a key-value store based on 'log-structured-merge' (LSM) trees that collects all changes in a mutable (changeable) in-memory buffer called a 'memtable'. Any updates to the same key in the memtable replace previous values, and once the memtable is full, RocksDB writes it to disk with all entries sorted by their key and with light compression applied. Once RocksDB writes the memtable to disk it is immutable (unchangeable) and is now called a 'sorted-string-table' (sstable).

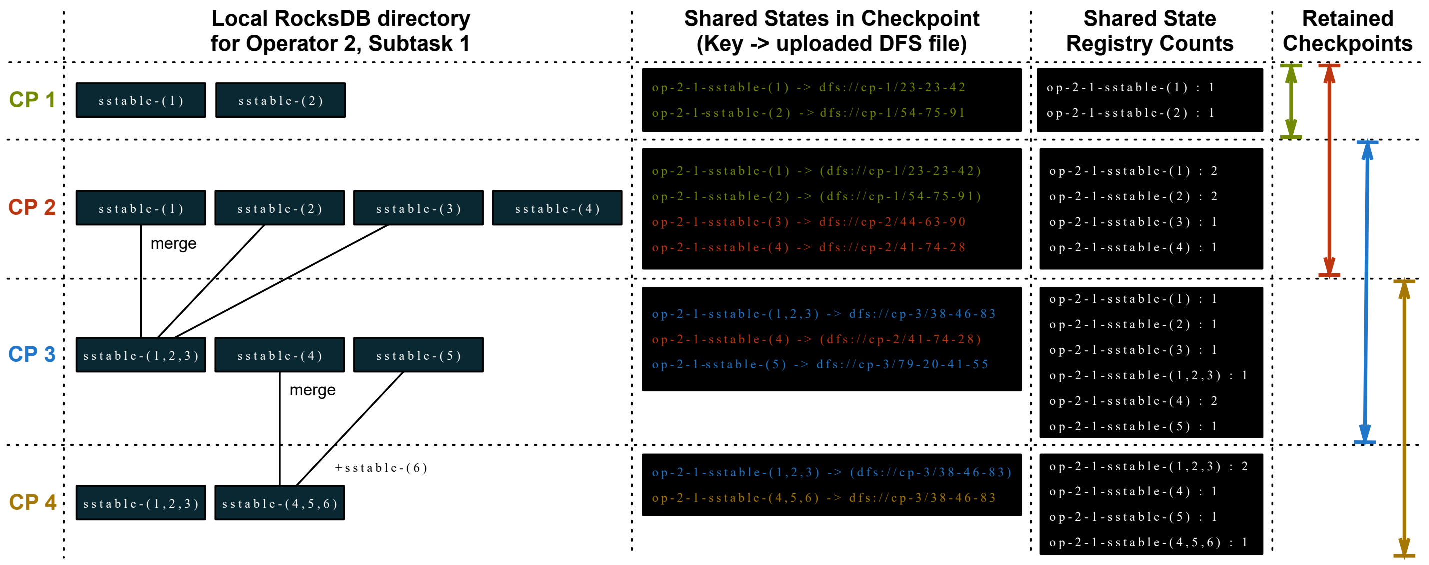
A 'compaction' background task merges sstables to consolidate potential duplicates for each key, and over time RocksDB deletes the original sstables, with the merged sstable containing all information from across all the other sstables.

On top of this, Flink tracks which sstable files RocksDB has created and deleted since the previous checkpoint, and as the sstables are immutable, Flink uses this to figure out the state changes. To do this, Flink triggers a flush in RocksDB, forcing all memtables into sstables on disk, and hard-linked in a local temporary directory. This process is synchronous to the processing pipeline, and Flink performs all further steps asynchronously and does not block processing.

Then Flink copies all new sstables to stable storage (e.g., HDFS, S3) to reference in the new checkpoint. Flink doesn't copy all sstables that already existed in the previous checkpoint to stable storage but re-references them. Any new checkpoints will no longer reference deleted files because deleted sstables in RocksDB are always the result of compaction, and it eventually replaces old tables with an sstable that is the result of a merge. This how in Flink's incremental checkpoints can prune the checkpoint history.

For tracking changes between checkpoints, the uploading of consolidated tables is redundant work. Flink performs the process incrementally, and typically adds only a small overhead, so we consider this worthwhile because it allows Flink to keep a shorter history of checkpoints to consider in a recovery.

(Click on the image below to open a full-size version in a new tab)


Local RocksDB directory for Operator 2, Subtask 1
Shared States in Checkpoint (Key -> uploaded DFS file)
Shared State Registry Counts
Retained Checkpoints
CP 1
CP 2
CP 3
CP 4
sstable-(1)
sstable-(2)
sstable-(3)
sstable-(4)
merge
sstable-(1,2,3)
sstable-(5)
+sstable-(6)
sstable-(4,5,6)
op-2-1-sstable-(1) -> dfs://cp-1/23-23-42
op-2-1-sstable-(2) -> dfs://cp-1/54-75-91
op-2-1-sstable-(1) : 1
op-2-1-sstable-(2) : 1
op-2-1-sstable-(1) -> (dfs://cp-1/23-23-42)
op-2-1-sstable-(2) -> (dfs://cp-1/54-75-91)
op-2-1-sstable-(3) -> dfs://cp-2/44-63-90
op-2-1-sstable-(4) -> dfs://cp-2/41-74-28
op-2-1-sstable-(1) : 2
op-2-1-sstable-(2) : 2
op-2-1-sstable-(3) : 1
op-2-1-sstable-(4) : 1
op-2-1-sstable-(1,2,3) -> dfs://cp-3/38-46-83
op-2-1-sstable-(4) -> (dfs://cp-2/41-74-28)
op-2-1-sstable-(5) -> dfs://cp-3/79-20-41-55
op-2-1-sstable-(1) : 1
op-2-1-sstable-(2) : 1
op-2-1-sstable-(3) : 1
op-2-1-sstable-(1,2,3) : 1
op-2-1-sstable-(4) : 2
op-2-1-sstable-(5) : 1
op-2-1-sstable-(1,2,3) -> (dfs://cp-3/38-46-83)
op-2-1-sstable-(4,5,6) -> dfs://cp-3/38-46-83
op-2-1-sstable-(1,2,3) : 2
op-2-1-sstable-(4) : 1
op-2-1-sstable-(5) : 1
op-2-1-sstable-(4,5,6) : 1

Take an example with a subtask of one operator that has a keyed state, and the number of retained checkpoints set at **2**. The columns in the figure above show the state of the local RocksDB instance for each checkpoint, the files it references, and the counts in the shared state registry after the checkpoint completes.

For checkpoint 'CP 1', the local RocksDB directory contains two sstable files, and it considers these new and uploads them to stable storage using directory names that match the checkpoint name. When the checkpoint completes, Flink creates the two entries in the shared state registry and sets their counts to '1'. The key in the shared state registry is a composite of an operator, subtask, and the original sstable file name. The registry also keeps a mapping from the key to the file path in stable storage.

For checkpoint 'CP 2', RocksDB has created two new sstable files, and the two older ones still exist. For checkpoint 'CP 2', Flink adds the two new files to stable storage and can reference the previous two files. When the checkpoint completes, Flink increases the counts for all referenced files by 1.

For checkpoint 'CP 3', RocksDB's compaction has merged `sstable-(1)`, `sstable-(2)`, and `sstable-(3)` into `sstable-(1,2,3)` and deleted the original files. This merged file contains the same information as the source files, with all duplicate entries eliminated. In addition to this merged file, `sstable-(4)` still exists and there is now a new `sstable-(5)` file. Flink adds the new `sstable-(1,2,3)` and `sstable-(5)` files to stable storage, `sstable-(4)` is re-referenced from checkpoint 'CP 2' and increases the counts for referenced files by 1. The older 'CP 1' checkpoint is now deleted as the number of retained checkpoints (2) has been reached. As part of this deletion, Flink decreases the counts for all files referenced 'CP 1', (`sstable-(1)` and `sstable-(2)`), by 1.

For checkpoint 'CP-4', RocksDB has merged `sstable-(4)`, `sstable-(5)`, and a new `sstable-(6)` into `sstable-(4,5,6)`. Flink adds this new table to stable storage and references it together with `sstable-(1,2,3)`, it increases the counts for `sstable-(1,2,3)` and `sstable-(4,5,6)` by 1 and then deletes 'CP-2' as the number of retained checkpoints has been reached. As the counts for `sstable-(1)`, `sstable-(2)`, and `sstable-(3)` have now dropped to 0, and Flink deletes them from stable storage.

Race Conditions and Concurrent Checkpoints

As Flink can execute multiple checkpoints in parallel, sometimes new checkpoints start before confirming previous checkpoints as completed. Because of this, Flink must consider which of the previous checkpoints to use as a basis for a new incremental checkpoint. Flink only references state from a checkpoint confirmed by the checkpoint coordinator so that it doesn't unintentionally reference a deleted shared file.

Restoring Checkpoints and Performance Considerations

If you enable incremental checkpointing, there are no further configuration steps needed to recover your state in case of failure. If a failure occurs, Flink's `JobManager` tells all tasks to restore from the last completed checkpoint, be it a full or incremental checkpoint. Each `TaskManager` then downloads their share of the state from the checkpoint on the distributed file system.

Though the feature can lead to a substantial improvement in checkpoint time for users with a large state, there are trade-offs to consider with incremental checkpointing. Overall, the process reduces the checkpointing time during normal operations but can lead to a longer recovery time depending on the size of your state. If the cluster failure is particularly severe and the Flink `TaskManager`s have to read from multiple checkpoints, recovery can be a slower operation than when using non-incremental checkpointing. You'll need to plan for larger distributed storage to maintain the checkpoints and the network overhead to read from it.

There are some strategies for improving the convenience/performance trade-off, and I recommend you read the Flink documentation for more details.

Interested in learning more about checkpoints in Flink? Check out Stefan Richter's Flink Forward Berlin 2017 talk "A Look at Flink's Internal Data Structures and Algorithms for Efficient Checkpointing".

And you might also enjoy our CTO Stephan Ewen's Flink Forward San Francisco 2017 talk "Experiences Running Flink at Very Large Scale".

Tags: Apache Flink, Flink Features