The Stream Processor as a Database

Apache Flink

Stephan Ewen

@stephanewen

dataArtisans

Streaming technology is enabling the obvious: **continuous processing** on data that is **continuously produced**

Apache Flink Stack

Libraries

DataStream API
Stream Processing

DataSet API
Batch Processing

Runtime
Distributed Streaming Data Flow

Streaming and **batch** as **first class citizens**.

Programs and Dataflows

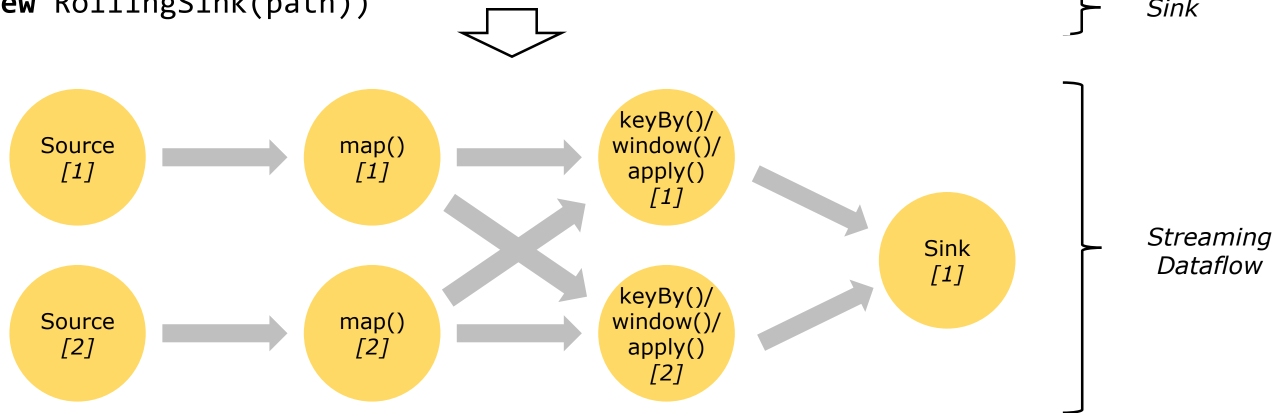
```
val lines: DataStream[String] = env.addSource(new FlinkKafkaConsumer09(…))

val events: DataStream[Event] = lines.map((line) => parse(line))

val stats: DataStream[Statistic] = stream
    .keyBy("sensor")
    .timeWindow(Time.seconds(5))
    .apply(new MyAggregationFunction())

stats.addSink(new RollingSink(path))
```

Source

Transformation

Transformation

Sink



Streaming Dataflow

What makes Flink flink?

True Streaming
- *Low latency*
- *High Throughput*
- *Well-behaved flow control (back pressure)*

Event Time
- *Make more sense of data*
- *Works on real-time and historic data*

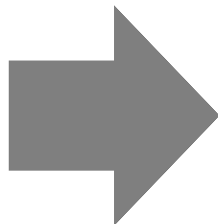
Stateful Streaming
- *Windows & user-defined state*
- *Exactly-once semantics for fault tolerance*
- *Globally consistent savepoints*

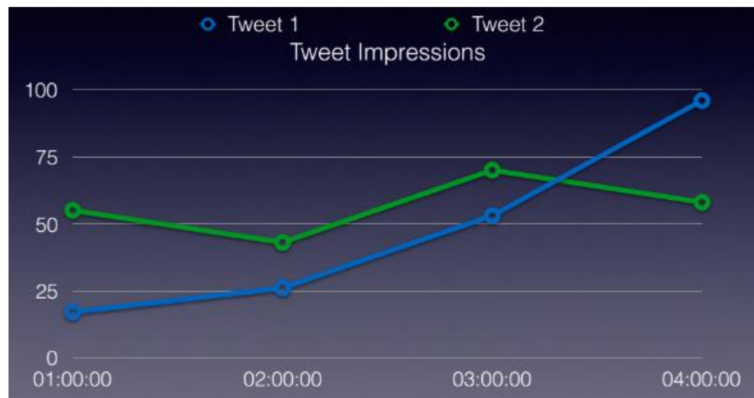
APIs Libraries
- *Complex Event Processing*
- *Flexible windows (time, count, session, roll-your own)*

Realtime Counts and Aggregates

The (Classic) Use Case

(Real)Time Series Statistics





stream of events

realtime statistics

The Architecture

kafka

Flink

hadoop HDFS

S3

cassandra

MySQL

mongoDB

ORACLE

APACHE HBASE

collect *log* *analyze* *serve & store*

The Flink Job

```
case class Impressions(id: String, impressions: Long)

val events: DataStream[Event] =
               env.addSource(new FlinkKafkaConsumer09(…))

val impressions: DataStream[Impressions] = events
    .filter(evt => evt.isImpression)
    .map(evt => Impressions(evt.id, evt.numImpressions)

val counts: DataStream[Impressions]= stream
    .keyBy("id")
    .timeWindow(Time.hours(1))
    .sum("impressions")
```

The Flink Job

Kafka Source → filter() → map() → *keyBy()* → window()/ sum() → Sink

Kafka Source → filter() → map() → *keyBy()* → window()/ sum() → Sink

Putting it all together

realtime queries

KafkaConsumer
map()
filter()

group

Flink event time windows

Periodic trigger + watermark trigger

Periodically (every second) flush new aggregates to Redis

How does it perform?

Latency

Throughput

Number of Keys

Yahoo! Streaming Benchmark

Latency

(lower is better)

Spark Streaming 1.5

Storm 0.10

Flink 0.10

99th Percentile Latency (sec)

9

8

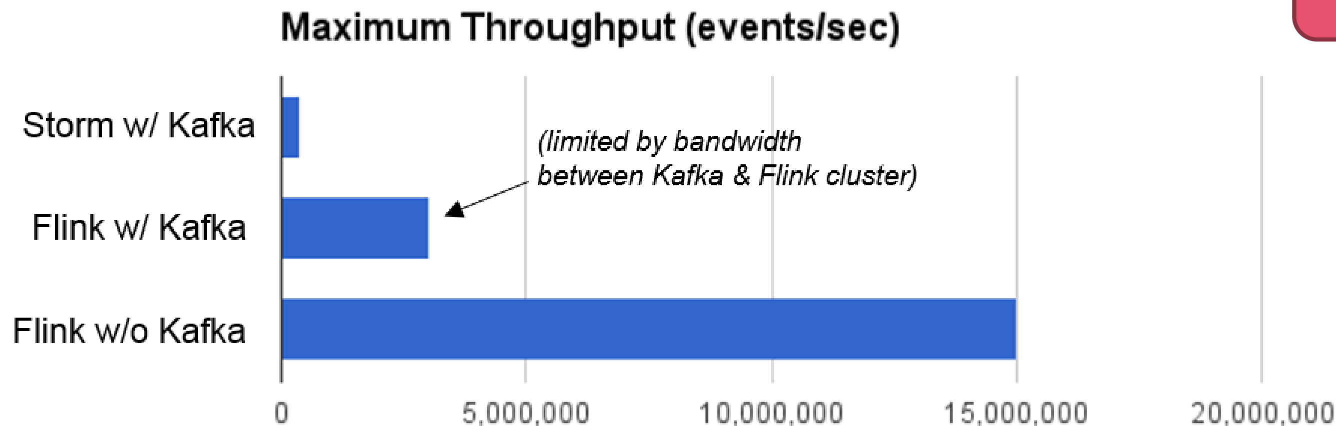
2

1

60 80 100 120 140 160 180

Throughput (1000 events/sec)

Extended Benchmark: Throughput



- 10 Kafka brokers with 2 partitions each
- 10 compute machines (Flink / Storm)
  - Xeon E3-1230-V2@3.30GHz CPU (4 cores HT)
  - 32 GB RAM (only 8GB allocated to JVMs)
- 10 GigE Ethernet between compute nodes
- 1 GigE Ethernet between Kafka cluster and Flink nodes

Scaling Number of Users

- Yahoo! Streaming Benchmark has 100 keys only
  - Every second, only 100 keys are written to key/value store
  - Quite few, compared to many real world use cases
- Tweet impressions: millions keys/hour
  - Up to millions of keys updated per second

Number of Keys

Performance

Number of Keys

Max Throughput (events/sec)

100 keys

1,000,000 keys

0 5000000 10000000 15000000 20000000

The Bottleneck

realtime queries

KafkaConsumer
map()
filter()

group

Flink event time windows

Periodic trigger + watermark trigger

Writes to the key/value store take too long

Queryable State

Queryable State

realtime
queries

KafkaConsumer
map()
filter()

group

Flink event time
windows

Archive
Database

Queryable State

realtime queries

KafkaConsumer
map()
filter()

group

Flink event time windows

Archive Database

Optional, and only at the end of windows

Queryable State Enablers

- Flink has state as a **first class citizen**
- State is **fault tolerant** (exactly once semantics)
- State is **partitioned** (sharded) together with the operators that create/update it
- State is **continuous** (not mini batched)
- State is **scalable** (e.g., embedded RocksDB state backend)

Queryable State Status

- [FLINK-3779] / Pull Request #2051 :
  Queryable State Prototype
- Design and implementation under evolution
- Some experiments were using earlier versions of the implementation
- Exact numbers may differ in final implementation, but order of magnitude is comparable

Queryable State Performance

Max Throughput (events/sec)

100 keys

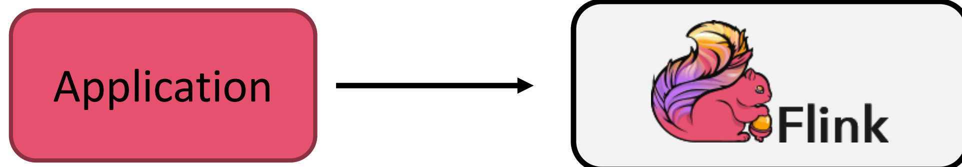
1,000,000 keys

Queryable State 1,000,000 keys

0 5000000 10000000 15000000 20000000

Queryable State: Application View

Application only interested in latest realtime results



Queryable State: Application View

Application requires both latest realtime- and older results

Application → Query Service

current time windows → Flink

past time windows → Database

realtime results

older results

Apache Flink Architecture Review

TaskManager

Task Slot | Task Slot | Task Slot
Task | Task

Memory & I/O Manager

Network Manager

Actor System

Data Streams

TaskManager

Task Slot | Task Slot | Task Slot
Task | Task

Memory & I/O Manager

Network Manager

Actor System

Flink Program

Program code

Program Dataflow

Optimizer / Graph Builder

Client

Actor System

Dataflow graph

Status updates

Statistics & results

Submit job (send dataflow)

Cancel / update job

Task Status

Heartbeats

Statistics

...

Deploy/Stop/ Cancel Tasks

Trigger Checkpoints

...

JobManager

Actor System

Dataflow Graph

Scheduler

Checkpoint Coordinator

Queryable State: Implementation

Query: /job/operation/state-name/key

Query Client

(1) Get location of "key-partition" for "operator" of" job"

(3) Respond location

(4) Query state-name and key

(2) Look up location

State Location Server

ExecutionGraph

Job Manager

deploy

status

State Registry

window()/ sum()

Task Manager

State Registry

window()/ sum()

Task Manager

register

local state

Contrasting with key/value stores

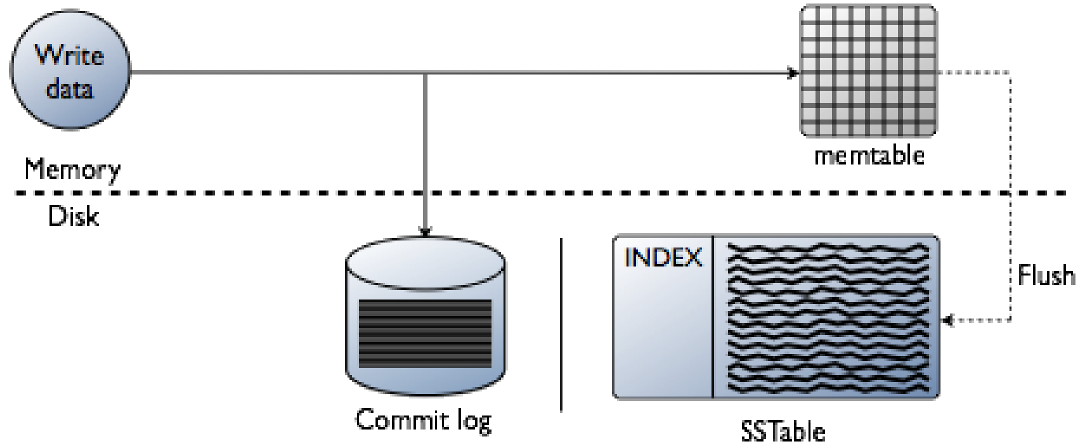
Turning the Database Inside Out

- Cf. Martin Kleppman's talks on re-designing data warehousing based on log-centric processing

- This view angle picks up some of these concepts

- Queryable State in Apache Flink = (Turning DB inside out)++

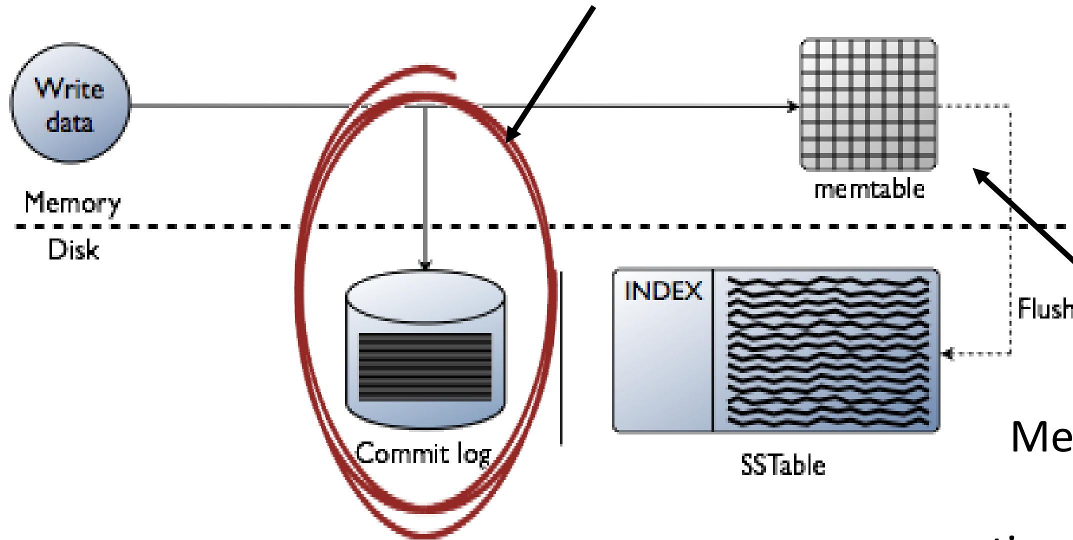
Write Path in Cassandra (simplified)



From the Apache Cassandra docs

Write Path in Cassandra (simplified)

First step is durable write to the commit log
(in all databases that offer strong durability)



Memtable is a re-computable
view of the commit log
actions and the persistent SSTables)

From the Apache Cassandra docs

Write Path in Cassandra (simplified)

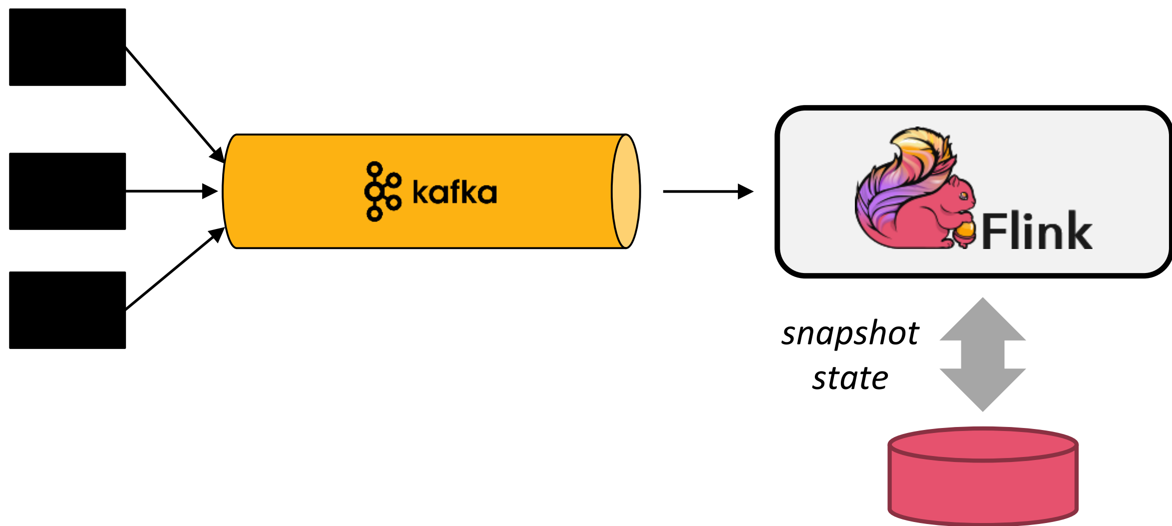
First step is durable write to the commit log
(in all databases that offer strong durability)

Replication to Quorum before write is acknowledged

Memtable is a re-computable view of the commit log actions and the persistent SSTables)

From the Apache Cassandra docs

Durability of Queryable state



Durability of Queryable state

Queryable state
re-computable
from checkpoint and log

kafka

Flink

snapshot state

Snapshot replication can happen in the background

Event sequence is the ground truth and is durably stored in the log already

Performance of Flink's State

Events flow without replication or synchronous writes

Performance of Flink's State

Trigger checkpoint

Inject checkpoint barrier

Source / filter() / map()

window()/ sum()

Performance of Flink's State

Take state snapshot

RocksDB:
Trigger state
copy-on-write

Source /
filter() /
map()

window()/
sum()

Performance of Flink's State

Persist state snapshots

Processing pipeline continues

Durably persist snapshots asynchronously

Source / filter() / map()

window()/ sum()

Conclusion

Takeaways

- Streaming applications are often not bound by the stream processor itself. **Cross system interaction** is **frequently biggest bottleneck**

- **Queryable state mitigates a big bottleneck**: Communication with external key/value stores to publish realtime results

- **Apache Flink**'s sophisticated support for state makes this possible

Takeaways

Performance of Queryable State

- Data persistence is fast with logs (*Apache Kafka*)
  - Append only, and streaming replication
- Computed state is fast with local data structures and no synchronous replication (*Apache Flink*)
- Flink's checkpoint method makes computed state persistent with low overhead

Go Flink!

Low latency

High Throughput

Make more sense of data

Works on real-time and historic data

Well-behaved flow control (back pressure)

True Streaming

Event Time

Windows & user-defined state

Stateful Streaming

APIs Libraries

Complex Event Processing

Exactly-once semantics for fault tolerance

Globally consistent savepoints

Flexible windows (time, count, session, roll-your own)