Will it Scale?

The Secrets behind Scaling Stream Processing Applications

Navina Ramesh
Software Engineer, LinkedIn
Apache Samza, Committer & PMC
navina@apache.org
What is this talk about?

- Understand the *architectural choices* in stream processing systems that may impact performance/scalability of stream processing applications
- Have a high level comparison of two streaming engines (Flink/Samza) with a focus on scalability of the stream-processing application
What this talk is not about?

- Not a feature-by-feature comparison of existing stream processing systems (such as Flink, Storm, Samza etc)
Agenda

● Use cases in Stream Processing
● Typical Data Pipelines
● Scaling Data Ingestion
● Scaling Data Processing
  ○ Challenges in Scaling Data Processing
  ○ Walk-through of Apache Flink & Apache Samza
  ○ Observations on state & fault-tolerance
● Challenges in Scaling Result Storage
● Conclusion
Spectrum of Processing

Stream Processing

RPC

0 ms

Milliseconds to minutes

Synchronous

Response Latency

Batch Processing

Later. Typically, hours
Newsfeed
Cyber Security
Selective Push Notifications
Agenda

● Use cases in Stream Processing

● Typical Data Pipelines

● Scaling Data Ingestion

● Scaling Data Processing
  ○ Challenges in Scaling Data Processing
  ○ Walk-through of Apache Flink & Apache Samza
  ○ Observations on state & fault-tolerance

● Challenges in Scaling Result Storage

● Conclusion
Typical Data Pipeline - Batch

Ingestion Service → HDFS → Mappers → Reducers → HDFS/HBase → Query
Typical Data Pipeline - Batch

Data Ingestion

Ingestion Service

HDFS

Mappers

Reducers

HDFS/HBase

Query
Typical Data Pipeline - Batch
Typical Data Pipeline - Batch
Parallels in Streaming

Data Ingestion

HDFS

Processors

Partition 0
Partition 1
... Partition N

Mappers
Reducers

HDFS/HBase

Query

Result Storage / Serving

Ingestion Service

Query
Parallels in Streaming

Data Ingestion

Ingestion Service

HDFS

Partition 0
Partition 1
... Partition N

Processors

Mappers

Reducers

HDFS

KV Store

Query

Result Storage / Serving

Data Processing

HDFS/ HBase

Query
### Batch

- Data Processing on **bounded** data
- Acceptable Latency - order of **hours**
- Processing occurs at **regular intervals**
- Throughput trumps latency
- **Horizontal scaling** to improve processing time

### Streaming

- Data processing on **unbounded** data
- Low latency - order of **sub-seconds**
- Processing is **continuous**
- Horizontal scaling is **not straightforward** (stateful applications)
- Need tools to **reason about time** (esp. when re-processing stream)
Agenda

- Use cases in Stream Processing
- Typical Data Pipelines

**Scaling Data Ingestion**

- Scaling Data Processing
  - Challenges in Scaling Data Processing
  - Walk-through of Apache Flink & Apache Samza
  - Observations on state & fault-tolerance

- Challenges in Scaling Result Storage
- Conclusion
Typical Data Ingestion

- Typically, streams are partitioned
- Messages sent to partitions based on “Partition Key”
- Time-based message retention

![Diagram of stream ingestion with producers, partitions, and consumers.]
Scaling Data Ingestion

- Scaling “up” -> Increasing partitions
- Changing partitioning logic *re-distributes* the keys across the partitions
Scaling Data Ingestion

- Scaling “up” -> Increasing partitions
- Changing partitioning logic re-distributes* the keys across the partitions
- Consuming clients (includes stream processors) should be able to re-adjust!
- Impact -> Over-provisioning of partitions in order to handle changes in load
Agenda

- Use cases in Stream Processing
- Typical Data Pipelines
- Scaling Data Ingestion

**Scaling Data Processing**
- Challenges in Scaling Data Processing
- Walk-through of Apache Flink & Apache Samza
- Observations on state & fault-tolerance

- Challenges in Scaling Result Storage
- Conclusion
Scaling Data Processing

- Increase number of processing units → Horizontal Scaling
Scaling Data Processing

- Increase number of processing units → Horizontal Scaling

But more machines means more $$$

- Impact NOT only CPU cores, but "large" (order of TBs) stateful applications impact network and disk!!
Key Bottleneck in Scaling Data Processing

- Accessing State
  - Operator state
    - Read/Write state that is maintained during stream processing
    - Eg: windowed aggregation, windowed join
  - Adjunct state
    - To process events, applications might need to lookup related or ‘adjunct’ data.
Accessing Operator State: *Assemble Call Graph*

Service Calls

- homepage_service_call
- feed_service_call
- profile_service_call
- pymk_service_call
- ...

Repartitioner

(Partition events by “tree id”)

Assembler

(Aggregate events by “tree id”)
**Accessing Operator State:** *Assemble Call Graph*

- In-memory structure to aggregate events until ready to output
- **Concerns:**
  - Large windows can cause overflow!
  - Restarting job after a long downtime can increase memory pressure!
Accessing Operator State: *Assemble Call Graph*

**Concerns:**
- **Remote RPC is Slow!!** (Stream: ~1 million records/sec; DB: ~3-4K writes/sec)
- **Mutations can’t rollback!**
  - Task may fail & recover
  - Change in logic!
Accessing Operator State: *Push Notifications*

- **Online Apps**
- **User Action Data**
- **B2**
- **Relevance Score**

**Task**

(Generate active notifications - filtering, windowed-aggregation, external calls etc)

**Notification System** (Scheduler)
Accessing Operator State: \textit{Push Notifications}

- Stream processing tasks consume from multiple sources - offline/online

- Performs multiple operations
  - Filters information and buffers data for window of time
  - Aggregates / Joins buffered data

- Total operator state per instance can easily grow to \textit{multiple GBs per Task}

\begin{itemize}
  \item \textbf{Online Apps}
  \item \textbf{User Action Data}
  \item \textbf{Task}
  \item \textbf{Notification System (Scheduler)}
  \item \textbf{B2}
  \item \textbf{Relevance Score}
\end{itemize}

(Generate active notifications - filtering, windowed-aggregation, external calls etc)
Accessing Adjunct Data: AdQuality Updates

Stream-to-Table Join

(task)

(Look-up memberId & generate AdQuality improvements for the User)

Read Member Data

Member Info

AdClicks

Task

AdQuality Update
Accessing Adjunct Data: *AdQuality Updates*

**Stream-to-Table Join**

AdClicks → Task → AdQuality Update

*(Look-up memberId & generate AdQuality improvements for the User)*

Read Member Data

**Member Info**

**Concerns:**
- Remote look-up Latency is high!
- DDoS on shared store - *MemberInfo*
Accessing Adjunct Data using Cache: *AdQuality Updates*

- **AdClicks** → **Task** → **AdQuality Update**

**Stream-to-Table Join**

- Maintain a cache of member Info & do local lookup

**Read Member Data**

- **Member Info**
Accessing Adjunct Data using Cache: *AdQuality Updates*

**Concerns:**
- Overhead of maintaining cache consistency based on the source of truth (*MemberInfo*).
- Warming up the cache after the job’s downtime can cause temporary spike in QPS on the shared store.
Agenda

● Use cases in Stream Processing
● Typical Data Pipelines
● Scaling Data Ingestion
● Scaling Data Processing
  ○ Challenges in Scaling Data Processing
  ○ Walk-through of Apache Flink & Apache Samza
    ○ Observations on state & fault-tolerance
● Challenges in Scaling Result Storage
● Conclusion
Apache Flink
Apache Flink: Processing

- Dataflows with *streams* and *transformation* operators
- Starts with one or more source and ends in one or more sinks
Apache Flink: Processing

- **JobManager** (*Master*) coordinates distributed execution such as, checkpoint, recovery management, schedule tasks etc.
- **TaskManager** (*JVM Process*) execute the subtasks of the dataflow, and buffer and exchange data streams
- Each *Task Slot* may execute multiple *subtasks* and runs on a separate thread.
Apache Flink: State Management

- **Lightweight Asynchronous Barrier Snapshots**
- Master triggers checkpoint and source inserts *barrier*
- On receiving barrier from all input sources, each operator stores the entire state, acks the checkpoint to the master and emits snapshot barrier in the output
Apache Flink: State Management

- **Lightweight Asynchronous Barrier Snapshots**
- Periodically snapshot the entire state to snapshot store
- Checkpoint mapping is stored in Job Manager
- **Snapshot Store** (typically, HDFS)
  - operator state (windows/aggregation)
  - user-defined state (checkpointed)
Apache Flink: State Management

- Operator state is primarily stored In-Memory or local File System
- Recently added RocksDB
- Allows user-defined operators to define state that should be checkpointed
Apache Flink: Fault Tolerance of State
Apache Flink: Fault Tolerance of State

- Full restore of snapshot from last completed checkpointed state
- Continues processing after restoring from the latest snapshot from the store
Apache Flink: Summary

● State Management Primitives:
  ○ Within task, local state info is stored primarily in-memory (recently, rocksdb)
  ○ Periodic snapshot (checkpoints + user-defined state + operator state) written to Snapshot Store

● Fault-Tolerance of State
  ○ Full state restored from Snapshot Store
Apache Flink: Observations

- Full snapshots are expensive for large states.
- Frequent snapshots that can quickly saturate network.
- Applications must trade-off between snapshot frequency and how large a state can be built within a task.
Apache Samza
Apache Samza: Processing

- **Samza Master** handles container life-cycle and failure handling
Apache Samza: Processing

- Samza Master handles container life-cycle and failure handling
- Each **container** (JVM process) contains more than one **task** to process the input stream partitions
Apache Samza: State Management

- Tasks checkpoint periodically to a **checkpoint stream**
- *Checkpoint* indicates which position in the input from which processing has to continue in case of a container restart
Apache Samza: State Management

- State store is local to the task - typically RocksDB (off-heap) and In-Memory (backed by a map)
- State store contains any operator state or adjunct state
- Allows application to define state through a Key Value interface
Apache Samza: State Management

- State store is continuously replicated to a changelog stream
- Each store partition is mapped to a specific changelog partition
Apache Samza: Fault Tolerance of State

Samza Master

Machine A
Container
Task
Task

Checkpoint Stream

Machine B
Container
Task
Task

Changelog Stream

Container Failure
Apache Samza: Fault Tolerance of State

- When container is recovered in a different host, there is no state available locally.
Apache Samza: Fault Tolerance of State

- When container comes up in a different host, there is no state available locally
- Restores from the beginning of the changelog stream → **Full restore!**
Apache Samza: Fault Tolerance of State

- State store is *persisted to local disk* on the machine, along with info on which offset to begin restoring the state from changelog.

The diagram illustrates the flow of data and state in a Samza cluster. The state store is persisted to local disk on the machine, along with information on which offset to begin restoring the state from the changelog.
Apache Samza: Fault Tolerance of State

- Samza Master tries to re-allocate the container on the same host.
- The feature where the Samza Master attempts to co-locate the task with their built-up state stores (where they were previously running) is called **Host-affinity**.
Apache Samza: Fault Tolerance of State

- Samza Master tries to re-allocate the container on the same host.
- The feature where the Samza Master attempts to co-locate the task with their built-up state stores (where they were previously running) is called **Host-affinity**.
- If container is re-allocated on the same host, state store is **partially restored** from changelog stream (*delta restore*)
Apache Samza: Fault Tolerance of State

- Once state is restored, checkpoint stream contains the correct offset for each task to begin processing.
Persisting state on local disk + host-affinity effectively **reduces the time-to-recover state from failure (or) upgrades** and continue with processing.
Persisting state on local disk + host-affinity effectively reduces the time-to-recover state from failure (or) upgrades and continue with processing.

Only a subset of tasks may require full restore, thereby, reducing the time to recover from failure or time to restart processing upon upgrades!
Apache Samza: Summary

● State Management Primitives
  ○ Within task, data is stored in-memory or on-disk using RocksDB
  ○ Checkpoint state stored in checkpoint-stream
  ○ User-defined and operator state continuously replicated in a changelog stream

● Fault-Tolerance of State
  ○ Full state restored by consuming changelog stream, if user-defined state not persisted on task’s machine
  ○ If locally persisted, only partial restore
Apache Samza: Observations

- State recovery from changelog can be time-consuming. It could potentially saturate Kafka clusters. Hence, partial restore is necessary.

- Host-affinity allows for faster failure recovery of task states, and faster job upgrades, even for large stateful jobs

- Since checkpoints are written to a stream and state is continuously replicated in changelog, frequent checkpoints are possible.
Agenda

- Use cases in Stream Processing
- Typical Data Pipelines
- Scaling Data Ingestion
- Scaling Data Processing
  - Challenges in Scaling Data Processing
  - Walk-through of Apache Flink & Apache Samza
  - **Observations on state & fault-tolerance**
- Challenges inScaling Result Storage
- Conclusion
### Comparison of State & Fault-tolerance

<table>
<thead>
<tr>
<th></th>
<th>Apache Samza</th>
<th>Apache Flink</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Durable State</strong></td>
<td>RocksDB</td>
<td>FileSystem (Recently added, RocksDB)</td>
</tr>
<tr>
<td><strong>State Fault Tolerance</strong></td>
<td>Kafka based Changelog Stream</td>
<td>HDFS</td>
</tr>
<tr>
<td><strong>State Update Unit</strong></td>
<td>Delta Changes</td>
<td>Full Snapshot</td>
</tr>
<tr>
<td><strong>State Recovery Unit</strong></td>
<td>Full Restore + Improved recovery with host-affinity</td>
<td>Full Restore</td>
</tr>
</tbody>
</table>
Agenda

- Use cases in Stream Processing
- Typical Data Pipelines
- Scaling Data Ingestion
- Scaling Data Processing
  - Challenges in Scaling Data Processing
  - Walk-through of Apache Flink & Apache Samza
  - Observations on state & fault-tolerance

- Challenges in Scaling Result Storage

- Conclusion
Challenges in Scaling Result Storage / Serving

- Any fast KV store can handle very small (order of thousands) QPS compared to the rate of stream processing output rate (order of millions)

- Output store can DoS due to high-throughput
Challenges in Scaling Result Storage / Serving

Online + Async Processor

Processor Downtime ~30 min

Processor Restarts
Scaling Result Storage / Serving

- Stream Processing
  - Change Stream
- Offline Processing
  - Bulk Load

Distributed Queue (Kafka) → Serving Platform
  (Derived Data)

Query → Online Apps
Agenda

● Use cases in Stream Processing
● Typical Data Pipelines
● Scaling Data Ingestion
● Scaling Data Processing
  ○ Challenges in Scaling Data Processing
  ○ Walk-through of Apache Flink & Apache Samza
  ○ Observations on state & fault-tolerance
● Challenges in Scaling Result Storage

● Conclusion
Conclusion

- Ingest/Process/Serve should be wholistically scalable to successfully scale stream processing applications.
- The notion of a “locally” accessible state is great to scale stream processing applications for performance. It brings in the additional cost of making the state fault-tolerant.
References

- Apache Flink - http://flink.apache.org
- https://blog.acolyer.org/2015/08/19/asynchronous-distributed-snapshots-for-distributed-dataflows/
Contribute!

Exciting features coming up in Apache Samza:

- **SAMZA-516** - Standalone deployment of Samza (independent of Yarn)
- **SAMZA-390** - High-level Language for Samza
- **SAMZA-863** - Multithreading in Samza

Join us!

- Apache Samza - samza.apache.org
- Mailing List - dev@samza.apache.org
- JIRA - [http://issues.apache.org/jira/browse/SAMZA](http://issues.apache.org/jira/browse/SAMZA)
Thanks!
Additional Slides
Revisit with Samza: *Assemble Call Graph*

---

**Service Calls**
- homepage_service_call
- feed_service_call
- profile_service_call
- pymk_service_call
- ...

**Managed Memory with RocksDb**

**Repartitioner**

**Assembler**
- Periodically trigger output and TTL stale records

---

**Homepage_service_call** (tree id: 10)
- Pymk_service_call (tree id: 10)
- Profile_service_call (tree id: 10)
- Feed_service_call (tree id: 10)
Revisit with Samza: *AdQuality Updates*

Diagram:
- AdClicks → Task
- Task → AdQuality Update
- Buffer state on-disk with change-capture
- Member Info
Company Standardization

Concerns:
1. Re-processing Latency!

Diagram:
- Updates to Company Info
- Company Info
- Task
- Task Ver. 2 (New Algorithm)
- Update Search Index
- Update Serving Store

Note: Re-process & update entire data set!!!
Revisit with Samza: *Company Standardization*