High-throughput, low-latency, and exactly-once stream processing with Apache Flink

Sep 15
Zehao Wu
streaming infrastructure

- popularity of stream data platforms is skyrocketing
- two main properties
  - high throughput across a wide spectrum of latencies
  - strong consistency guarantees even in the presence of stateful computations
What makes a good stream system?

- Exactly-once guarantees
- Low latency
- High throughput
- Powerful computation model
- Low overhead of the fault tolerance mechanism in the absence of failures
- Flow control
Record acknowledgements

- Apache Storm
- sends back the previous an acknowledgement
- source keeps a backup of all the tuples
- received acknowledgements from all, discarded
- not all have been received, replayed
Micro batches

- Apache Storm Trident, Apache Spark Streaming
- broken down in a series of small, atomic batch jobs called micro-batches
- either succeed or fail
- at a failure, the latest can be simply recomputed
Transactional updates

- Google Cloud Dataflow
- Details will be covered in the next presentation
Distributed Snapshots

- Apache Flink
- Chandy Lamport algorithm, regular processing keeps going, while checkpoints happen in the background
- draw a consistent snapshot of that state
- storing that snapshot in durable storage
- restore from durable storage, rewind the stream source and hit the play button again
data stream

newer records

checkpoint barrier \( n \)

part of checkpoint \( n+1 \)

checkpoint barrier \( n-1 \)

part of checkpoint \( n \)

older records

part of checkpoint \( n-1 \)

stream record (event)

1. align barriers
2. checkpoint state
3. emit barrier \( n \) and continue
<table>
<thead>
<tr>
<th>Feature</th>
<th>Record acks (Storm)</th>
<th>Micro-batching (Spark Streaming, Trident)</th>
<th>Transactional updates (Google Cloud Dataflow)</th>
<th>Distributed snapshots (Flink)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guarantee</td>
<td>At least once</td>
<td>Exactly once</td>
<td>Exactly once</td>
<td>Exactly once</td>
</tr>
<tr>
<td>Latency</td>
<td>Very Low</td>
<td>High</td>
<td>Low (delay of transaction)</td>
<td>Very Low</td>
</tr>
<tr>
<td>Throughput</td>
<td>Low</td>
<td>High</td>
<td>Medium to High (Depends on throughput of distributed transactional store)</td>
<td>High</td>
</tr>
<tr>
<td>Computation model</td>
<td>Streaming</td>
<td>Micro-batch</td>
<td>Streaming</td>
<td>Streaming</td>
</tr>
<tr>
<td>Overhead of fault tolerance mechanism</td>
<td>High</td>
<td>Low</td>
<td>Depends on throughput of distributed transactional store</td>
<td>Low</td>
</tr>
<tr>
<td>Flow control</td>
<td>Problematic</td>
<td>Problematic</td>
<td>Natural</td>
<td>Natural</td>
</tr>
<tr>
<td>Separation of application logic from fault tolerance</td>
<td>Partially (timeouts matter)</td>
<td>No (micro batch size affects semantics)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Lacking

- High availability
- Event time and watermarks
- Improved monitoring of running jobs
Pros

- Fast
- Reliable
- Expressive
- Easy to use
- Scalable
- Hadoop-compatible
Cons

- Too many traditional database designs
- forced declarative
- harder to program
- the community not as active as Spark