Dataflow Streaming
What’s New & What’s Next?

Iñigo San Jose, Tom Stepp
Google
Agenda

- Overview
- Autotuning
- GCP PubSub Integration
- Observability
- Other Projects
Overview
Overview: Streaming @ Google

- History of Streaming @ Google
- Streaming Appliance vs Streaming Engine
- Streaming Basics
History of Streaming @ Google

- Everything was batch
- MapReduce
- First streaming systems were designed for Ads
- Streaming MapReduce
- MillWheel
- Streaming Flume
- Windmill (Dataflow)
History of Streaming @ Google

- **2004**: MapReduce
  - Simple distributed data processing

- **2010**: Flume
  - Logical pipelines & optimization

- **2013**: Millwheel
  - Low-latency streaming

- **2015**: Cloud Dataflow + Apache Beam
  - Batch + Streaming Serverless Cloud
Streaming Appliance vs Streaming Engine

Streaming Appliance

- Worker
- User Code
- Windmill
- Worker(s)

Streaming Engine

- User Worker(s)
- Windmill Worker(s)
Streaming Engine vs Streaming Appliance

Streaming Appliance

- Worker
  - User Code
    - DoFn
    - DoFn
    - DoFn
  - Windmill
    - Disk
    - Disk
    - Disk
- Worker(s)

Streaming Engine

- User Worker(s)
  - DoFn
  - DoFn
  - DoFn
- Windmill Worker(s)
  - Bigtable
Benefits of Streaming Engine:

- More efficient use of User Workers
- No need for Persistent Disks
- More responsive Horizontal Autoscaling
- Improved supportability and visibility
Streaming Basics

Pipeline example

Pubsub Read
  ↓
ParDo
  ↓
Window
  ↓
GroupByKey
  ↓
ParDo
  ↓
Sink

What Dataflow Streaming Sees

Fused Stage 1
- Pubsub Read
- ParDo
- Window
- GroupByKey (Write)

Fused Stage 2
- GroupByKey (Read)
- ParDo
- Sink
• Every message has a key assigned to it
• Keys can be user defined or system defined
• Keys are hashed
• Elements are processed in the context of a key
• Keys are the basic unit of parallelism
- Keys belong to key-ranges
- Key ranges are assigned to workers
- Key ranges can be split and sent to different workers

**Fused Stage 1**
- Pubsub Read
- ParDo
- Window
- GroupByKey (Write)

**Fused Stage 2**
- GroupByKey (Read)
- ParDo
- Sink

**Fused Stage 1**
- [00, FF]

**Fused Stage 2**
- [00, FF]

**NOTE:** all range boundaries are hexadecimal values.
- Keys belong to key-ranges
- Key ranges are assigned to workers
- Key ranges can be split and sent to different workers
Worker 2

1729_Pear
A007_Banana
C3PO_Steak

Worker 1

[00, 78)
[78, AB)
[AB, FF]

Fused Stage 1

[00, 78)
[78, AB)
[AB, FF]

Fused Stage 2

[00, 78)
[78, AB)
[AB, FF]

Streaming Basics
Streaming Basics

Fused Stage 1
- [00, 78)
- [78, AB)
- [AB, FF]

Fused Stage 2
- [00, 78)
- [78, AB)
- [AB, FF]

Worker 1
- [00, 78)
- [78, AB)
- [78, AB]

Worker 2
- [00, 78)
- [AB, FF]

9696_Meat
3141_Fruit

3141_Fruit
Autotuning
Past: Scaling backend workers linearly with user workers.

Present: Scaling each worker pool independently.
Past: Unconditionally throttling user worker upscale if < 20% CPU utilization.

Present: Throttle user worker upscale on key parallelism limits (number of keys).
Past: Only consider the current state (backlog, throughput, etc.)

Present: Track scaling frequencies, downscale slower when yo-yoing detected (frequent up/down scaling in short time frame).
Past: When autoscale events happen, new workers need to load the pipeline state from persistence. This can take time and lead to backlog and latency.

Present: Transfer info directly from workers, reducing latency
Past: When autoscale events happen, new workers need to load the pipeline state from persistence. This can take time and lead to backlog and latency.

Present: Transfer info directly from workers, reducing latency
Autotuning: Scaling Actuation Latencies

**Past:** When autoscale events happen, new workers need to load the pipeline state from persistence. This can take time and lead to backlog and latency.

**Present:** Transfer info directly from workers, reducing latency.
Past: When autoscale events happen, new workers need to load the pipeline state from persistence. This can take time and lead to backlog and latency.

Present: Transfer info directly from workers, reducing latency
Autotuning: Scaling Actuation Latencies

Latency Disabled (top) vs Enabled (bottom)
Autotuning: Scaling Actuation Latencies

User Workers Disabled vs Enabled
Past: If a key range has a disproportionate amount of input rate, its worker would have more load than others, potentially accumulating backlog and wasting resources on other workers.

Present: We can split key ranges dynamically and rebalance them across workers based on their throughput.

<table>
<thead>
<tr>
<th>Key Range</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001-key1</td>
<td>500mb/s</td>
</tr>
<tr>
<td>6002-key2</td>
<td>300mb/s</td>
</tr>
<tr>
<td>BCDF-key3</td>
<td>50mb/s</td>
</tr>
</tbody>
</table>

Worker1:
- [00, AB] 800mb/s

Worker2:
- [AB, FF] 50mb/s
Past: If a key range has a disproportionate amount of input rate, its worker would have more load than others, potentially accumulating backlog and wasting resources on other workers.

Present: We can split key ranges dynamically and rebalance them across workers based on their throughput

<table>
<thead>
<tr>
<th>Key Range</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001-key1</td>
<td>500mb/s</td>
</tr>
<tr>
<td>6002-key2</td>
<td>300mb/s</td>
</tr>
<tr>
<td>BCDF-key3</td>
<td>50mb/s</td>
</tr>
</tbody>
</table>

Worker1
- [00, 55): 500mb/s

Worker2
- [55, AB): 300mb/s
- [AB, FF]: 50mb/s
Autotuning: Range Rebalancing

Feature Enabled
**Past:** Autosharding was only available for Streaming Inserts / File Loads and was load agnostic, which could lead to wasted resources in case of dynamic destinations.

**Present:** StorageAPI gets autosharding option, using backlog and throughput as metric.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>200mb/s</th>
<th>1000 shards</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2</td>
<td>100mb/s</td>
<td>1000 shards</td>
</tr>
<tr>
<td>Table 3</td>
<td>1mb/s</td>
<td>1000 shards</td>
</tr>
</tbody>
</table>
**Past:** Autosharding was only available for Streaming Inserts / File Loads and was load agnostic, which could lead to wasted resources in case of dynamic destinations.

**Present:** StorageAPI gets autosharding option, using backlog and throughput as metric.

<table>
<thead>
<tr>
<th>Table</th>
<th>Throughput</th>
<th>Shards</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>200mb/s</td>
<td>800</td>
</tr>
<tr>
<td>Table 2</td>
<td>100mb/s</td>
<td>400</td>
</tr>
<tr>
<td>Table 3</td>
<td>1mb/s</td>
<td>4</td>
</tr>
</tbody>
</table>
Autotuning: BigQuery Autosharding

Streaming Inserts + Autosharding

Storage API

Storage API + Autosharding
GCP PubSub Integration
**Past:** Pipelines used old Pubsub API Unary Pull

**Present:** Pipelines use newer Pubsub API Streaming Pull, improving throughput and latency
**Past:** Pipelines used old Pubsub API Unary Pull

**Present:** Pipelines use newer Pubsub API Streaming Pull, improving throughput and latency
PubSub Streaming Pull

Latency and Backlog Improvements
Usage improvements

Latest worker status: Autoscaling: Raised the number of workers to 300 so that the pipeline can catch up with its backlog and keep up with its input rate.
Observability
• Collecting many new Streaming Engine metrics
• Some integrated into Dataflow UI
• All available in Monitoring UI
• Available dashboard template for easy detailed job performance monitoring

New Metrics

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duplicates Filtered</td>
<td>/job/duplicates_filtered_out_count</td>
</tr>
<tr>
<td>Processing Parallelism</td>
<td>/job/processing_parallelism_keys</td>
</tr>
<tr>
<td>Backlog Bytes</td>
<td>/job/backlog_bytes</td>
</tr>
<tr>
<td>Backlog Seconds</td>
<td>/job/estimated_backlog_processing_time</td>
</tr>
<tr>
<td>Timers Processed</td>
<td>/job/timers_processed_count</td>
</tr>
<tr>
<td>Timers Resident</td>
<td>/job/timers_pending_count</td>
</tr>
<tr>
<td>Status of Streaming Pull connections</td>
<td>/job/pubsub/streaming_pull_connection_status</td>
</tr>
<tr>
<td>The number of bytes produced by this pi-transform</td>
<td>/job/estimated_bytes_produced_count</td>
</tr>
<tr>
<td>Checkpoint bytes written</td>
<td>/job/streaming_engine/persistent_state/write_bytes_count</td>
</tr>
<tr>
<td>Checkpoint bytes read</td>
<td>/job/streaming_engine/persistent_state/read_bytes_count</td>
</tr>
<tr>
<td>Checkpoint Latency</td>
<td>/job/streaming_engine/persistent_state/write_latencies</td>
</tr>
<tr>
<td>User Processing Latency</td>
<td>/job/bundle_user_processing_latencies</td>
</tr>
<tr>
<td>Key (Range) Availability</td>
<td>/job/streaming_engine/key_processing_availability</td>
</tr>
<tr>
<td>The number of bytes consumed by this pi-transform</td>
<td>/job/estimated_bytes_consumed_count</td>
</tr>
<tr>
<td>The number of bytes being processed by pi-transform</td>
<td>/job/estimated_bytes_active</td>
</tr>
<tr>
<td>Pubsub Pull to Ack Latency</td>
<td>/job/pubsub/pulled_message_ages</td>
</tr>
<tr>
<td>Persistent State Usage</td>
<td>/job/streaming_engine/persistent_state/stored_bytes</td>
</tr>
<tr>
<td>Late pubsub messages</td>
<td>/job/pubsub/late_messages_count</td>
</tr>
<tr>
<td>Target workers</td>
<td>/job/target_worker_instances</td>
</tr>
<tr>
<td>Pubsub Publish Messages/Errors</td>
<td>/job/pubsub/published_messages_count</td>
</tr>
</tbody>
</table>
### Observability: Dataflow UI

**Metrics**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>REDUCE_PERCENTILE_50</td>
<td>50th Percentile</td>
<td>0</td>
</tr>
<tr>
<td>REDUCE_PERCENTILE_95</td>
<td>95th Percentile</td>
<td>0.11min</td>
</tr>
<tr>
<td>REDUCE_PERCENTILE_99</td>
<td>99th Percentile</td>
<td>0.16min</td>
</tr>
</tbody>
</table>

**Processing**

- **User processing latencies heatmap**
  - Create alerting policy
  - UTC-7, 10:50 AM to 12:10 PM

**Chart**

- Backlog
- Errors
- Throughput
- System latency
- Data freshness

**Menu**

- JOB GRAPH
- EXECUTION DETAILS
- JOB METRICS
- COST
- RECOMMENDATIONS
- AUTOSCALING

---

Launched

---

BEAM SUMMIT NYC 2023
Observability: Dashboard Template

Preview of first few graphs

Importing template
Other Projects
We wanted to test the throughput of sources and sinks without any special settings. We got to 10 GB/s for these I/O combos:

- Pubsub to BQ
- Pubsub to Pubsub
- Pubsub to GCS*
- Kafka to GCS*
- Kafka to BQ
Out of the box

Pubsub to GCS example
Collection of +190 self-contained Dataflow pipelines ready to use, including most common sources, sinks, and use cases.

https://github.com/GoogleCloudPlatform/dataflow-cookbook
QUESTIONS?