Optimizing a Dataflow pipeline for cost efficiency: lessons learned at Orange

By Jérémie Gomez and Thomas Sauvagnat

BEAM SUMMIT

Austin, 2022

Meet the **team**

Jérémie Gomez
Data Cloud Consultant
Google

Thomas Sauvagnat
Data engineer
orange

Agenda

1. Use case
2. Architecture & initial decisions
3. The journey
4. Final results

Use case

01

Google Cloud

26 countries

and a global presence with Orange Business Services

Europe

Belgium
France
Luxembourg
Moldova
Poland
Romania
Slovakia
Spain

Africa and the Middle East

Botswana
Burkina Faso
Cameroon
Central African Republic
Côte d'Ivoire
Democratic Republic of the Congo
Egypt
Guinea
Guinea-Bissau
Jordan
Liberia
Madagascar
Mali
Mauritius
Morocco
Senegal
Sierra Leone
Tunisia

€42.3 billion

in revenues

259 million

customers

5

business activities

- Enhanced connectivity (retail and business customers)
- Business IT support services
- Wholesale services
- Cybersecurity
- Financial services

142,000

employees

The FigaroSI use case

Figaro probes collects various logs from Orange France home devices for retail customers :

- CPU/Mem usage
- Power consumption
- Temperature sensors
- Boot stats
- Process crash
- WAN and Homelan stats
- WiFi stats
- VoIP/VoWiFi stats

LiveBox
(Residential gateway)

Set Top Box

WiFi Extender

The FigaroSI use case

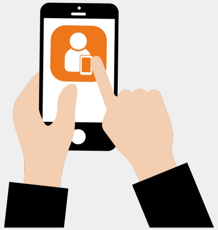
Performed operations :

- **parse and ingest Figaro probes data** (this is the topic of this presentation)
- computes daily KPIs
- enriches data
- delivers information to external systems

Purposes :

Execute proactive actions (reboot, push config)

Provide diagnostics labels for the customer service

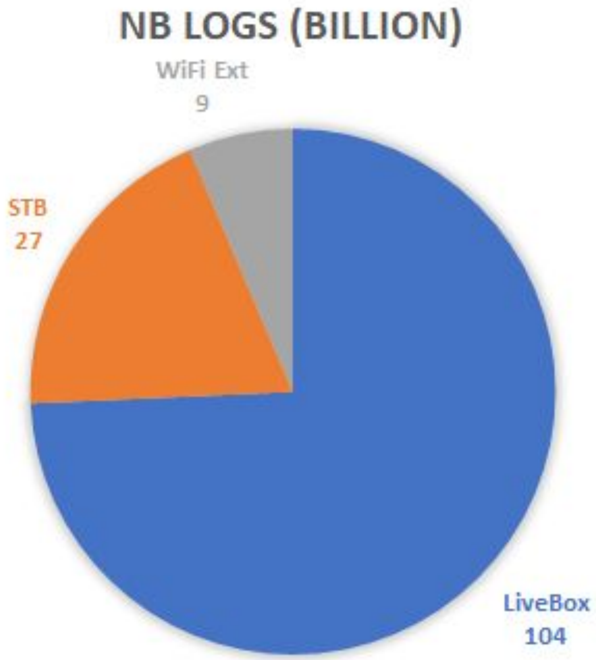


Provide KPIs for self-help troubleshooting through the “Orange et Moi” application

The FigaroSI use case

One of the main issues is the large volume of these logs :

- 15.6 million Orange France devices with Figaro probe
- 70 million enduser WiFi devices
- **140 billion logs per day**
- 33 TB BigQuery billable byte per day



Architecture & initial decisions

02

Google Cloud

The Dataflow pipeline

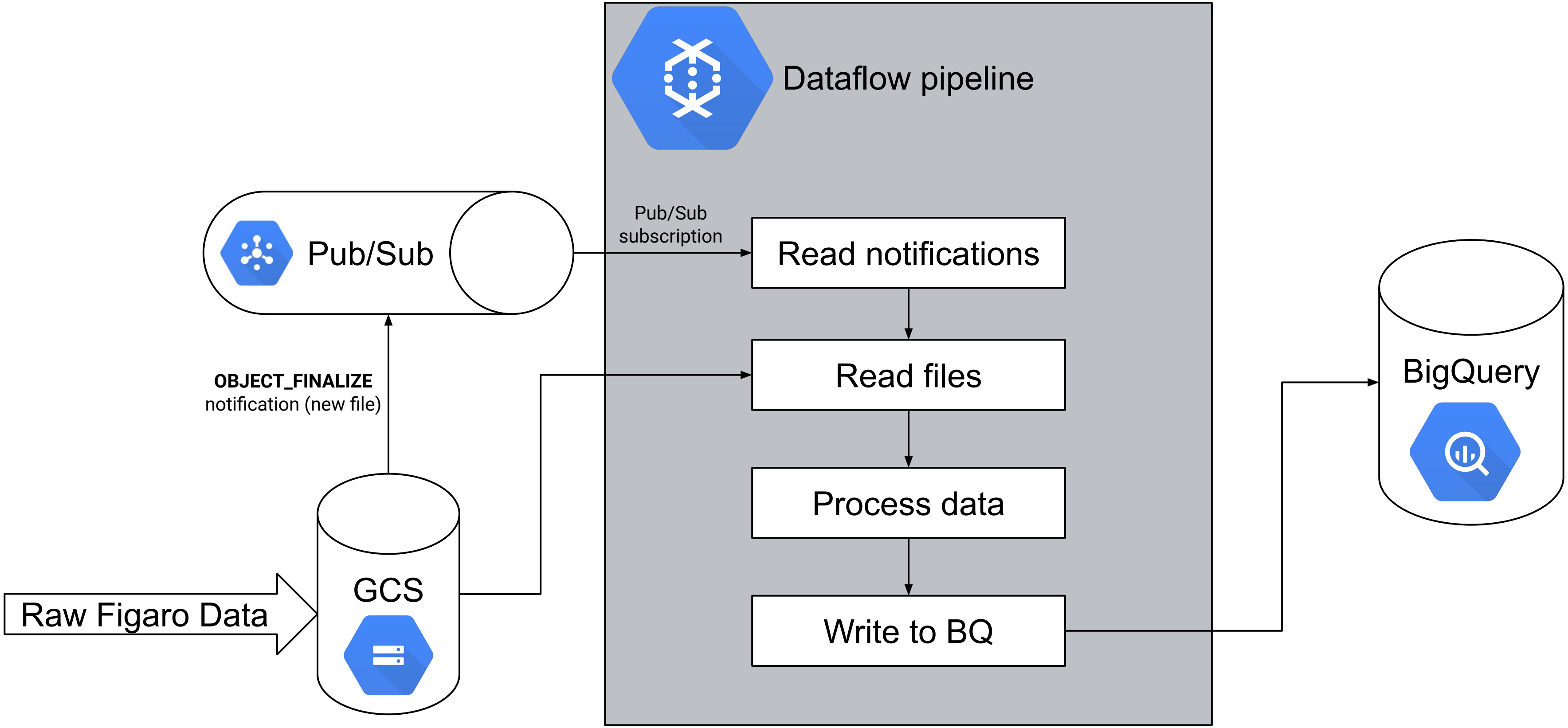
Why Dataflow?

- Managed services: no infrastructure to manage, autoscaling
- Native IO connectors: GCS (file storage), Pub/sub (for continuous ingestion), BigQuery
- The Beam framework: can code in Java, concepts similar to Spark, can run the core of the code on other runners (Spark, Flink, etc.)

Transformations

- Combine 2 rows (header with compressed data representing 1 hour of logs from a device)
- Parse data (uncompress data, split data into logs, extract useful information)
- Adjust timezone (date format and timezone depends on device firmware version and device location [french overseas territories])

Architecture



Pipeline choices

```
rows.apply(
  "Write to BigQuery",
  BigQueryIO.writeTableRows()
    .to(String.format("%s:%s.%s", project, dataset, table))
    .withSchema(schema)
    .withCreateDisposition(CreateDisposition.CREATE_IF_NEEDED)
    .withWriteDisposition(WriteDisposition.WRITE_APPEND)
    .withMethod(BigQueryIO.Write.Method.STREAMING_INSERTS)
    .withAutoSharding()
    .ignoreInsertIds()
);
```

Pipeline choices

Initial choices

- Files arrive about every minute: chose a streaming job
- Default BigQueryIO in streaming jobs: the legacy streaming API
  => performance not sufficient

First improvements

- Activation of auto sharding (requires Streaming Engine)
  => performance improved, but hit the 100MB/s limit
- Stopped using insertIds()
  => performance ok, but without leeway (close to the 1 GB/s limit)

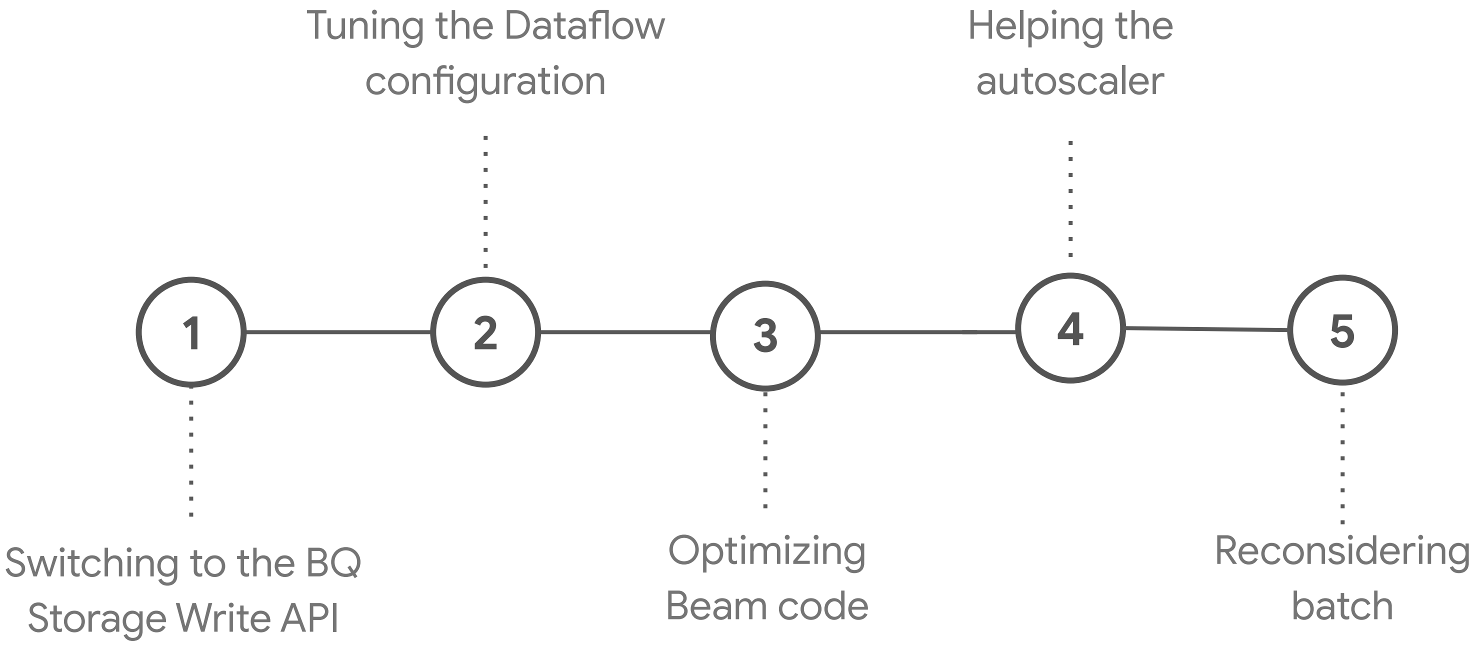
First cost projections

$5.8m / year

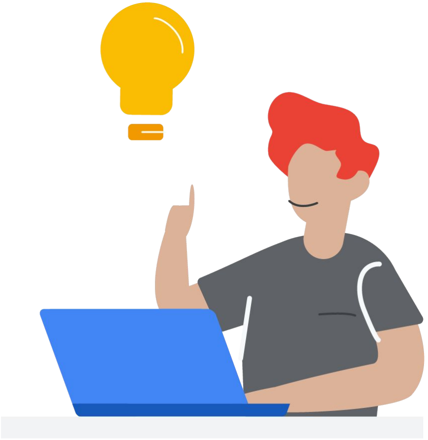
The journey

03

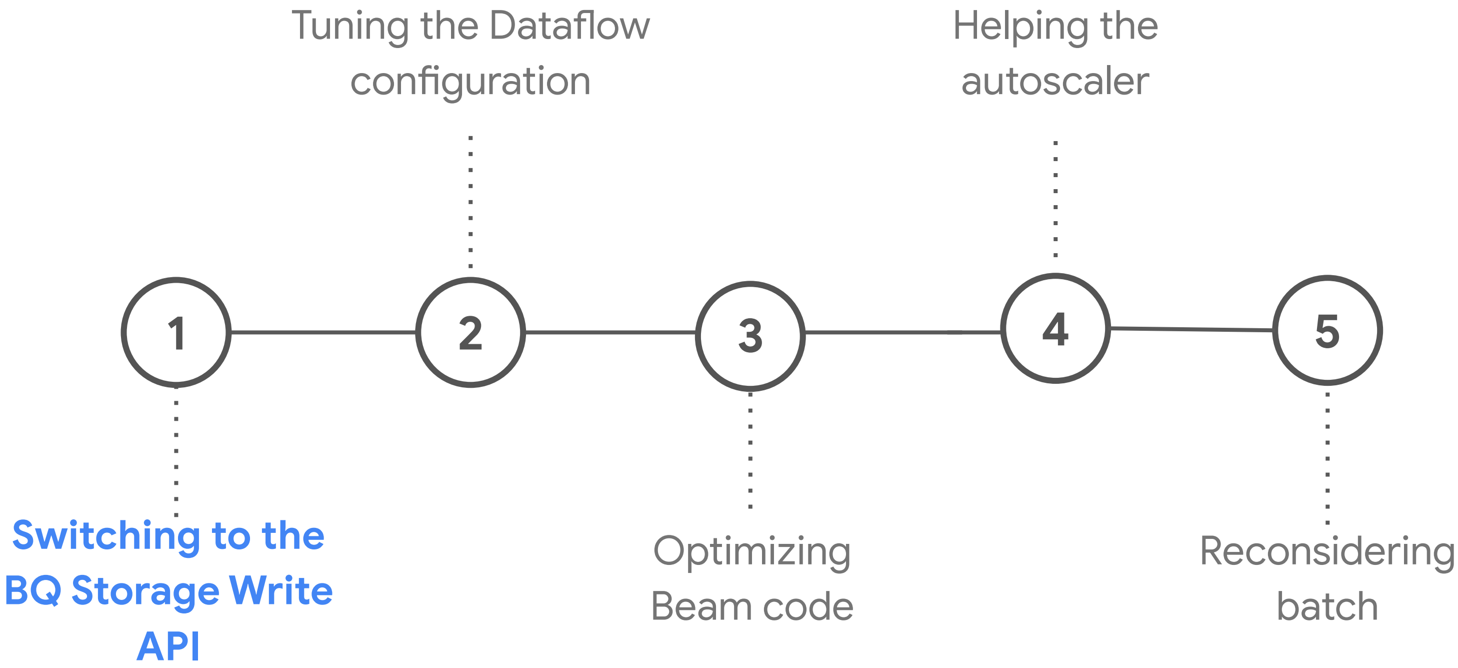
The journey timeline



Disclaimer: we recently re-ran tests to confirm gains for each step.
We grouped some of them for this presentation, so intermediate gains are only approximate.
We will say when gains from the project and from the re-run differed.



The journey timeline



1. Switching to the BQ Write Storage API

First, let's have a look at the available APIs to load data into BQ.

Load API	• For batch loads • Free with the shared slot pool • Buy PIPELINE slots for guaranteed capacity
(Legacy) streaming API	• For streaming loads • Pay per ingested volume (0.01$/200MB in US multiregion)
Storage write API	• For batch loads & streaming loads • Pay (half) per ingested volume (0.025$/1GB in US multiregion) • New capabilities • Recommended for streaming pipelines and high-performance batch pipelines

1. Switching to the BQ Write Storage API

Step 1/2

1 Action	2 Rationale	3 Obstacles	4 Impact
Use the BigQuery Write Storage API instead of the (legacy) BQ streaming API	This API is x2 cheaper, does not have the 1GB/s limit, is performant, has exactly-once ingestion.	Limit on regional tables (300 MB/s), no autosharding, limited documentation (number of streams is the one in your code * the number of tables written to)	Fewer workers needed (decreased RAM and CPU by 55%), ingestion costs decreased by 85%. Overall a 68% decrease. Increasing the number of streams did not bring improvement.

```
BigQueryIO.writeTableRows()
    .to(XXX).withSchema(XXX).withCreateDisposition(XXX).withWriteDisposition(XXX)
    .withMethod(BigQueryIO.Write.Method.STORAGE_WRITE_API)
    .withTriggeringFrequency(Duration.standardSeconds(30))
    .withNumStorageWriteApiStreams(90)
```

1. Switching to the BQ Write Storage API

Step 2/2

1 Action

For the sake of completeness, tried the BQ load API.

2 Rationale

This API is free.

3 Obstacles

The performance was insufficient: the latency kept increasing.

4 Impact

None (rollbacked).

```
BigQueryIO.writeTableRows()
    .to(XXX).withSchema(XXX).withCreateDisposition(XXX).withWriteDisposition(XXX)
    .withMethod(BigQueryIO.Write.Method.FILE_LOADS)
    .withTriggeringFrequency(Duration.standardMinutes(2))
```

1. Switching to the BQ Write Storage API

Keep in mind

- For high throughput, use the BQ Storage Write API with a multiregional destination table.
- As long as autosharding is not available, experiment with your number of streams: higher is not always better.
- More generally, make sure you are calling and using your external systems as optimally as possible.

The journey timeline

1. Switching to the BQ Storage Write API
   ~ 68% gain
2. **Tuning the Dataflow configuration**
3. Optimizing Beam code
4. Helping the autoscaler
5. Reconsidering batch

2. Tuning Dataflow configuration

What configuration can we change?

Machines configuration

- Machine family (n1, n2, ...) can be changed without cost change.
- Many sizes (n2-standard-8, n2-standard-16, ...) can be used.
- Number of threads can be chosen (e.g. default in Java: 300 threads per vCPU for streaming jobs, 1 thread per vCPU for batch jobs)

Streaming engine
Dataflow shuffle
Dataflow prime

- Streaming Engine moves state & shuffle to a backend service (smoother autoscaling, smaller machines required, less disk used)
- Dataflow shuffle is similar for batch jobs (smaller machines, performance improvements).
- Dataflow prime enables vertical autoscaling (for streaming Python) and right fitting.

2. Tuning Dataflow configuration

Step 1/4

1
Action

Change family type from n1 to n2.

2
Rationale

N2 machines have a more recent CPU and vCPUs cost the same.

3
Obstacles

No obstacle found

4
Impact

During the project, we observed upscaling became less aggressive, which improved the average number of workers. During the re-run, we did not observe any improvement.

```
--workerMachineType=n2-standard-16
```

2. Tuning Dataflow configuration

Step 2/4

1 Action	2 Rationale	3 Obstacles	4 Impact
Change machine size from n2-standard-16 to n2-standard-8	CPU and RAM might be underutilized (metrics suggested that CPU was not the limiting factor).	No obstacle found	Slight impact on the number of workers (we estimated a 5% decrease in cost).

```
--workerMachineType=n2-standard-8
```

2. Tuning Dataflow configuration

Step 3/4

1 Action	2 Rationale	3 Obstacles	4 Impact
Try different number of threads (more than 300 and less than 300)	The default parallelization might be tuned in order to better utilize vCPUs.	No obstacle found	No impact. Other use cases can see impact from doing this, in particular batch jobs.

```
--numberOfWorkerHarnessThreads=300
```

2. Tuning Dataflow configuration

Step 4/4

1 Action

Disabled the Streaming Engine.

2 Rationale

Streaming engine is priced by the volume of shuffled data. We might need more workers but the cost can be decreased.

3 Obstacles

No obstacle found

4 Impact

At the time of the project, we decreased costs by about 10% by switching it off. During re-run, we did not see an improvement.

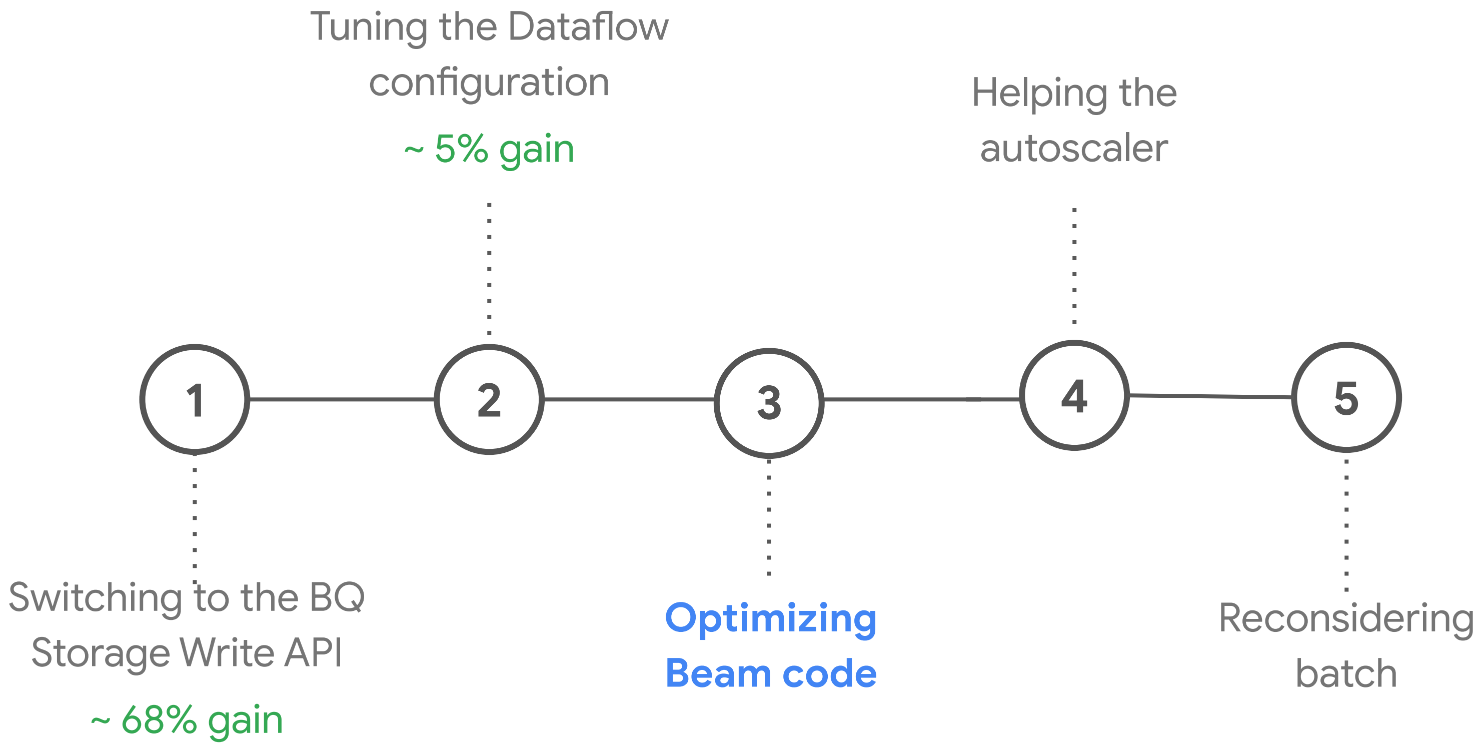
```
--enableStreamingEngine=false
```

2. Tuning Dataflow configuration

Keep in mind

- Your mileage may vary: some configuration changes may have big effects on some pipelines and no effect on others. Optimizing will require testing.

The journey timeline



3. Optimizing Beam code

The way you code your pipelines can have a huge impact on performance/cost.

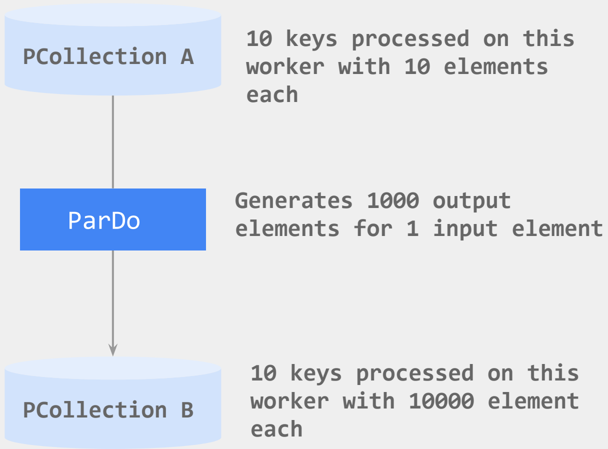
```
static class MatchWordWithRegexFn extends DoFn<String, String> {
  @Setup
  public void setup() {
    Pattern.compile(regexp) <----------------  yes

 @ProcessElement
  public void processElement(@Element String word, OutputReceiver<String> out) {
    Pattern.compile(regexp) <-----------------  no
  }
}
```

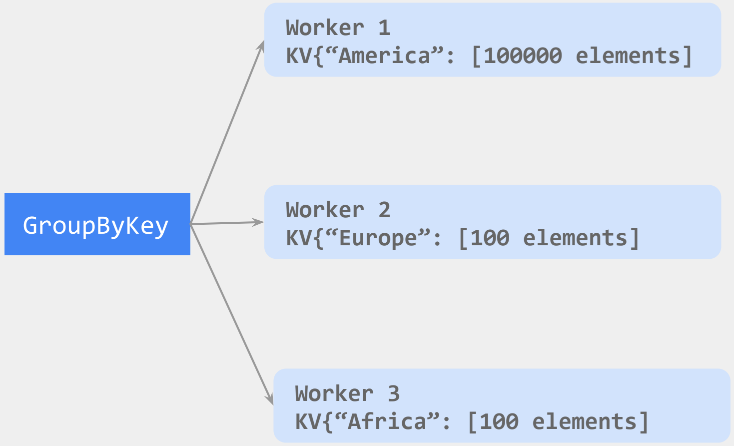
3. Optimizing Beam code

The way you code your pipelines can have a huge impact on performance/cost.

Stage fusion with high fanout transform



Skewed data



3. Optimizing Beam code

The way you code your pipelines can have a huge impact on performance/cost.

Following coding best practices

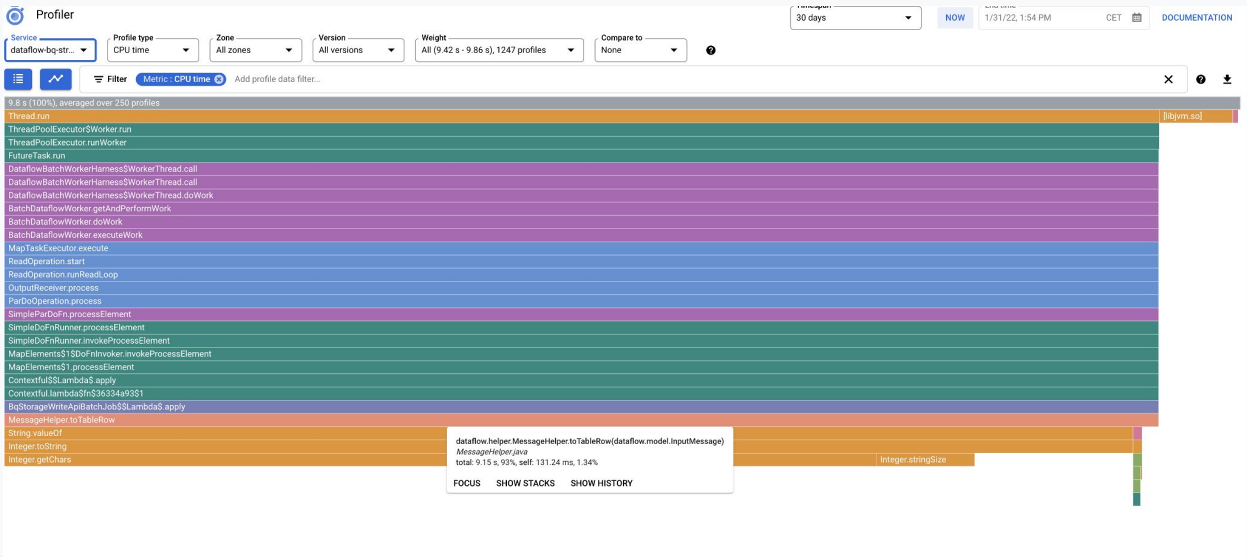
- Filter first (especially before shuffle operations)
- Do not instantiate your costly operations (regex compilation, database connections, etc.) in the processElement method. Rather use the setUp method.
- Use efficient coders (e.g not SerializableCoder, for Java).
- Use side inputs instead of CoGroupByKey when one side of the join is small.
- Be aware of stage fusion, small key space and data skew
- If possible, do not use non-distributable compressed files like gzip.
- Be careful with excessive logging.
- Java is usually more performant than Python.

3. Optimizing Beam code

You can use code profiling in order to finely determine CPU/memory bottlenecks.

Profiling your code

- Use the flag to profile the code:
  `--dataflowServiceOptions=enable_google_cloud_profiler`
- Enables to use directly the Cloud Profiler on GCP



3. Optimizing Beam code

Step 1/1

1 Action

Profiled the code and improved the most CPU-intensive parts (mostly regular expressions)

2 Rationale

We may need fewer workers if our code consumes fewer CPU cycles.

3 Obstacles

No obstacle found

4 Impact

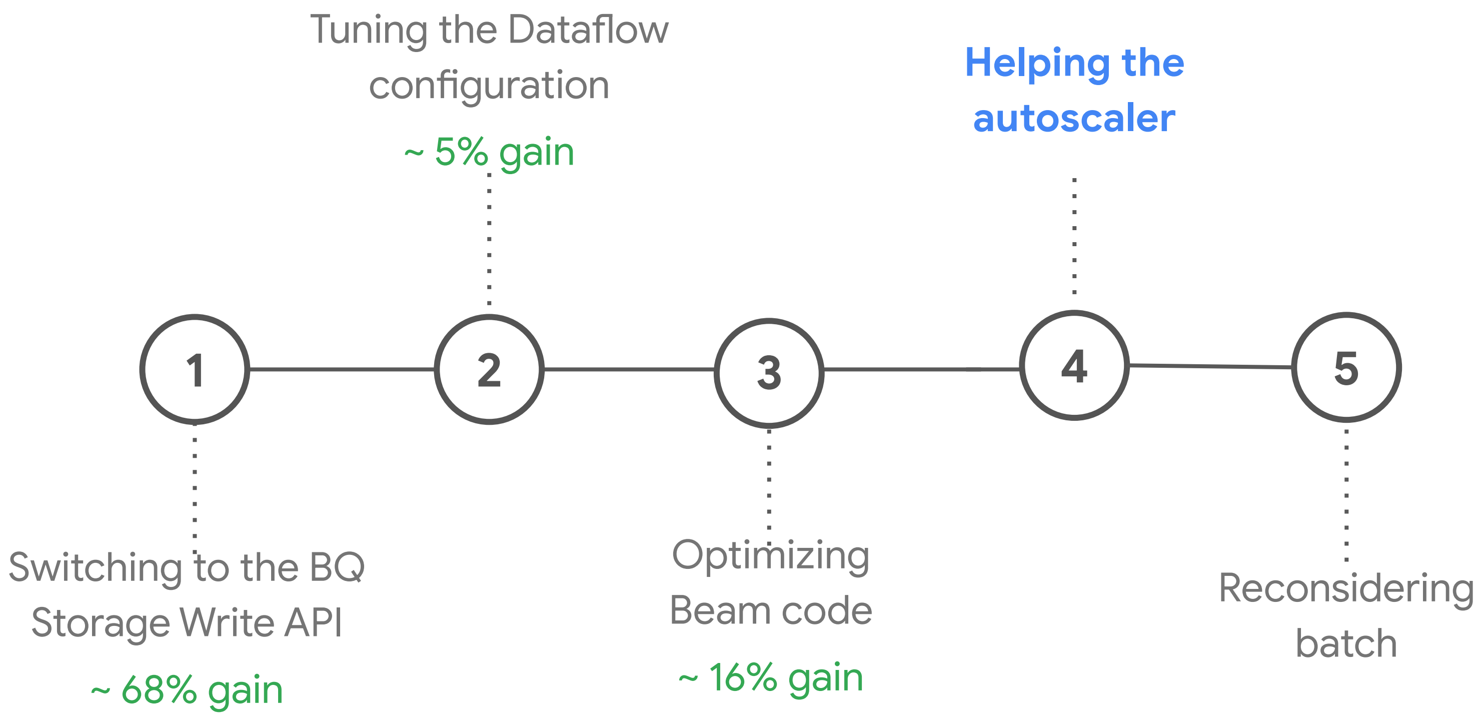
We saw an impact on vCPU-time consumed. During the project, we did not see a significant decrease in number of workers, but we did in the re-run (about 16% cost decrease).

3. Optimizing Beam code

Keep in mind

- The way you code your pipeline can have a big impact on performance & cost.
- Profiling your code can complement following best practices.
- Using metrics from the Dataflow UI can also help you determine where you should focus your efforts.

The journey timeline



4. Helping the autoscaler

The autoscaler follows a certain algorithm, you may have to help it a little to adapt to your case.

Autoscaler decisions (streaming)

- Scales up when average CPU utilization is > 20% and the backlog is > 15 seconds for a couple of minutes.
- Scales down if the average CPU utilization is < 75% and the backlog is < 10 seconds for a couple of minutes.

Help the autoscaler

- Streaming Engine usually provides a more reactive and smoother autoscaling.
- Setting a good number of initial and max workers is a good idea.
- Setting a minimum number of workers is experimental with `--experiment=min_num_workers=N`

4. Helping the autoscaler

Step 1/1

1 Action

Experimented and set a good number of initial & max number of workers

2 Rationale

The autoscaler behavior is not necessarily the best for us, it scales too much and stays high for a long time. We are ok to have some peak latency.

3 Obstacles

No obstacle found

4 Impact

Increasing the min number of workers to 100 and leaving the max at 300 decreased costs by about 30%. Further tuning of these parameters (70 initial and 70 max workers) yielded another 12% of decrease in costs.

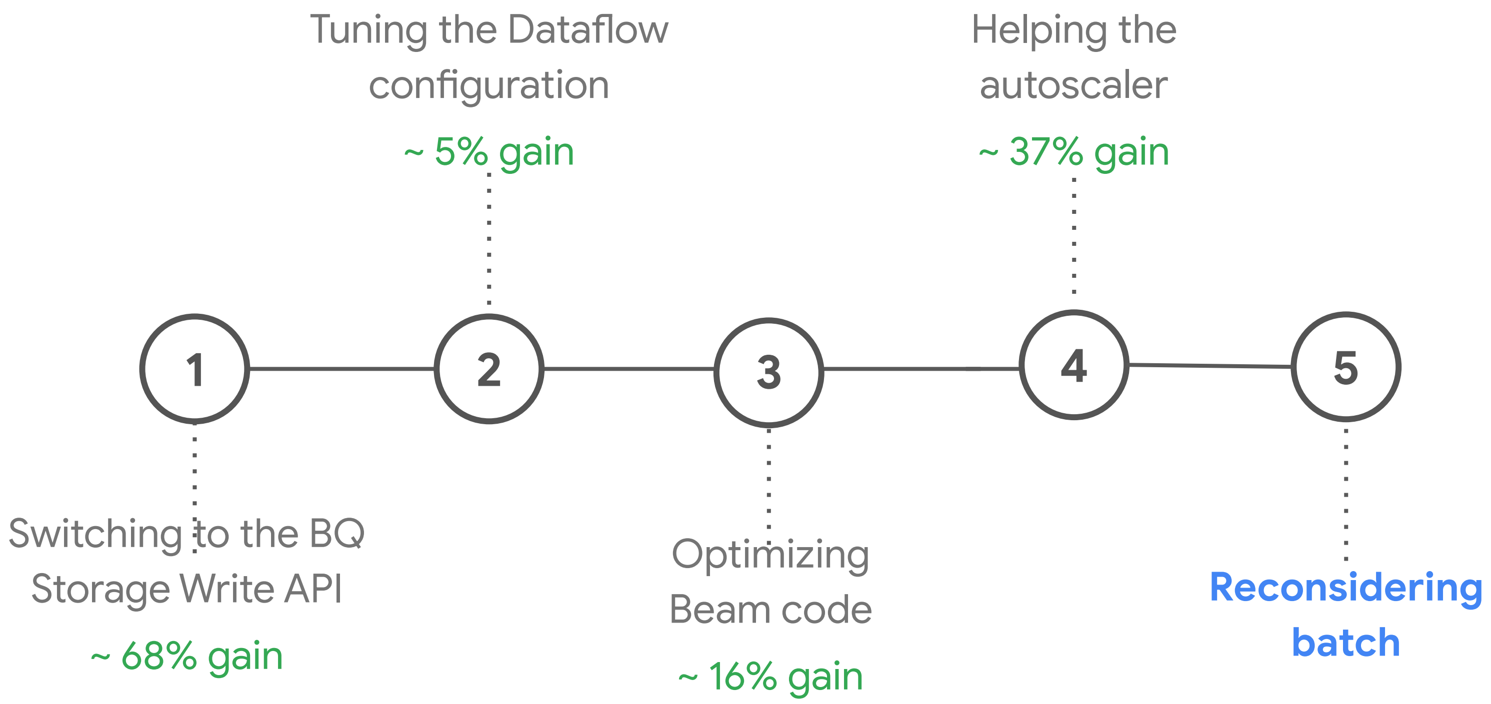
```
--numWorkers=70
--maxNumWorkers=70
```

4. Helping the autoscaler

Keep in mind

- Even if the autoscaler is very useful, it is not yet very customizable.
- Helping the autoscaler to have a behavior that matches your use case can decrease costs significantly.

The journey timeline




Tell me the truth...I'm...I'm ready
to hear it.
You don't need streaming

5. Reconsidering batch

It is easy to switch between batch & streaming with Beam, and the cost might be quite different.

Streaming vs batch

- Streaming workers are 15% more expensive in Dataflow.
- Using a BQ load with a streaming job is possible but not efficient

Our use case

- We chose streaming for a technical reason: we thought batch loads would not be efficient enough for our throughput
- No business reason to choose streaming (we are ok with a few hours latency as long as we do not accumulate latency)

5. Reconsidering batch

Step 1/1

1 Action	2 Rationale	3 Obstacles	4 Impact
Change the IO to make a batch job, and use the BQ load API.	If performance is sufficient, batch workers will be cheaper (15%) and BQ load API is free.	No obstacle found	The job runs every 30 minutes and actually takes only 18mn with 30 workers (240 vCPU). This removed the ingestion cost, and batch workers run less time and are less expensive. This decreased costs by 68%.

5. Reconsidering batch

Keep in mind

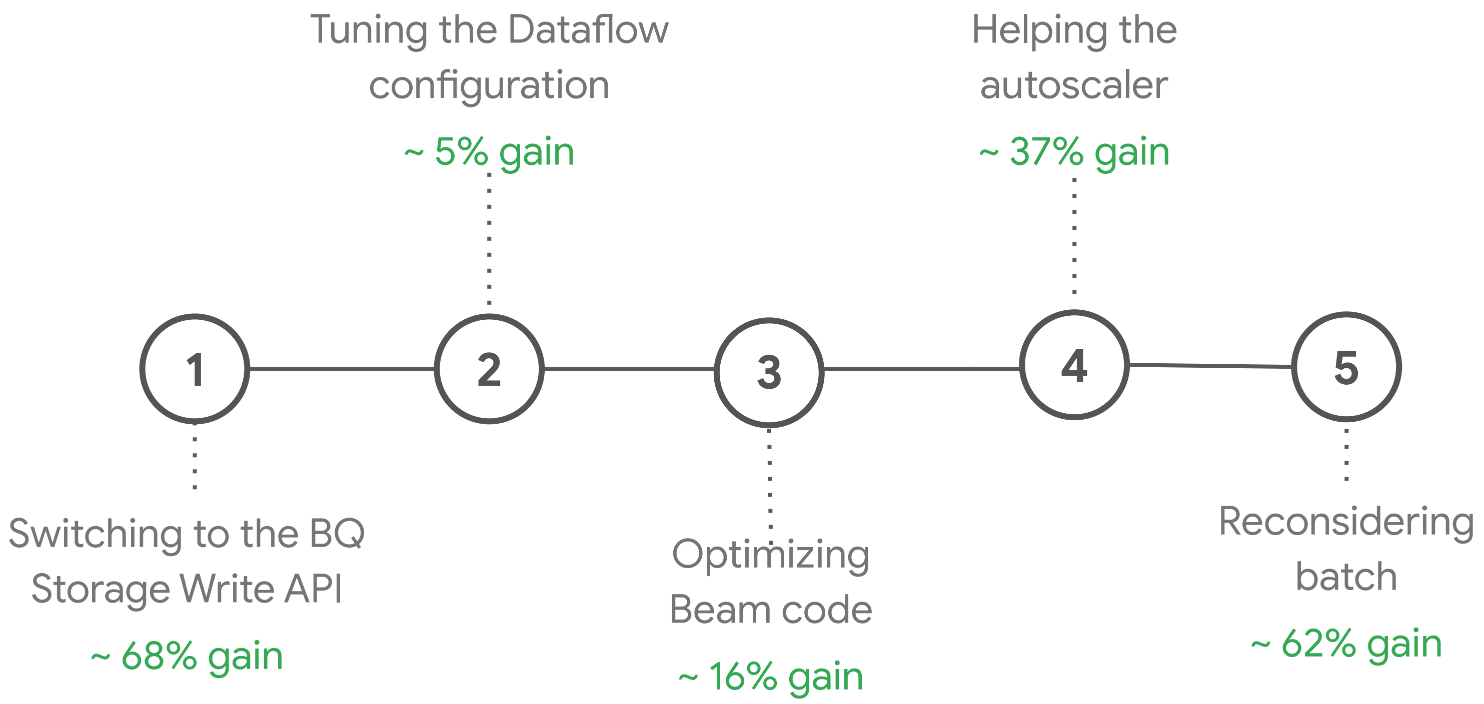
- If your business case does not require low latency processing, do not assume you need streaming for throughput reasons.

Final results

04

Google Cloud

The journey timeline



Final win by keeping a streaming pipeline

Decreased costs by ~5x

(~$5.8m/y to ~$1.2m/y)

Final win by switching to a batch pipeline

Decreased costs by 13x+

(~$5.8m/y to ~$440k/y)

Now the cost distribution is around 35% for the Dataflow batch pipeline and 65% for BigQuery storage (1.15PB for 35 days)

Final thoughts

Some leads we did not need to try on this use case:

- Using FlexRS
- Using Dataflow prime

Your mileage will vary

- Some steps that had a minor impact on this pipeline might be major for yours, depending on the pipeline and the order in which you take the steps. It did for us!

Experiment, experiment, experiment.

Questions?

www.linkedin.com/in/thomassauvagnat
www.linkedin.com/in/jeremiegomez

BEAM SUMMIT Austin, 2022