Google

No shard left behind

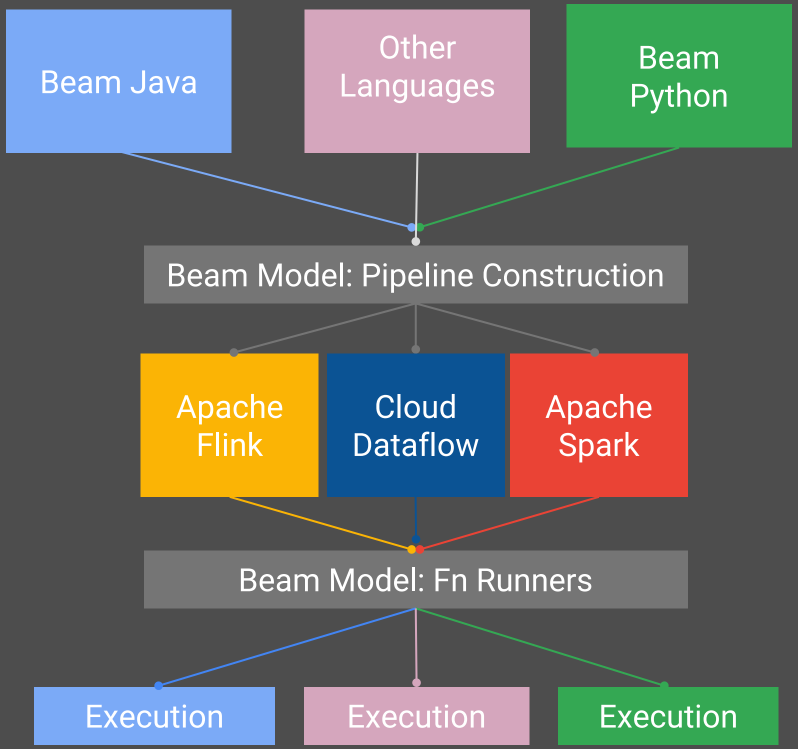
Dynamic Work Rebalancing and other adaptive features in Apache Beam

Malo Denielou (malo@google.com)

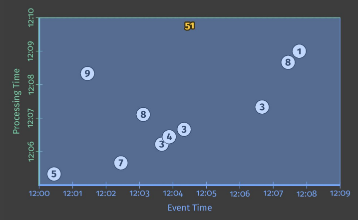
Apache Beam is a **unified** programming model designed to provide **efficient** and **portable** data processing pipelines.

Apache Beam

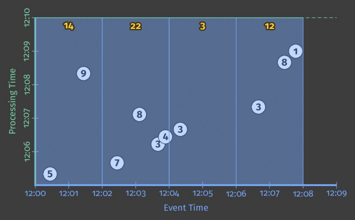
1. The Beam Programming Model
2. SDKs for writing Beam pipelines -- Java/Python/...
3. Runners for existing distributed processing backends
    - Apache Flink
    - Apache Spark
    - Apache Apex
    - Dataflow
    - Direct runner (for testing)



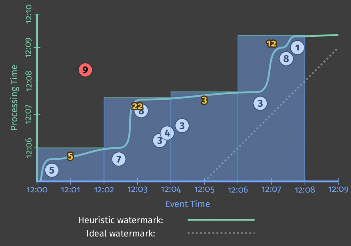
Apache Beam use cases



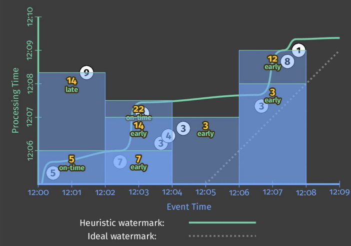
1.Classic Batch



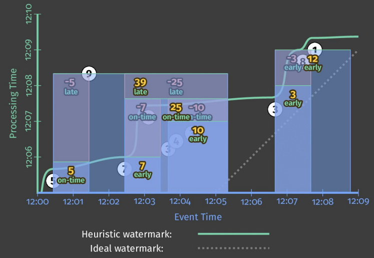
2. Batch with Fixed Windows



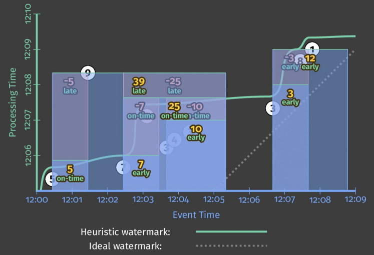
3. Streaming



4. Streaming with Speculative + Late Data

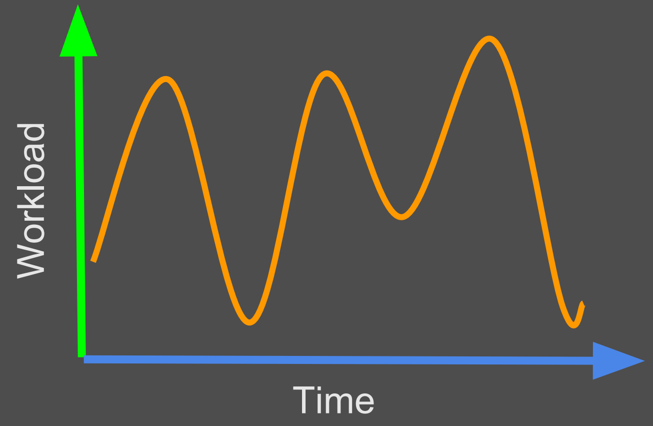


5. Streaming With Retractions

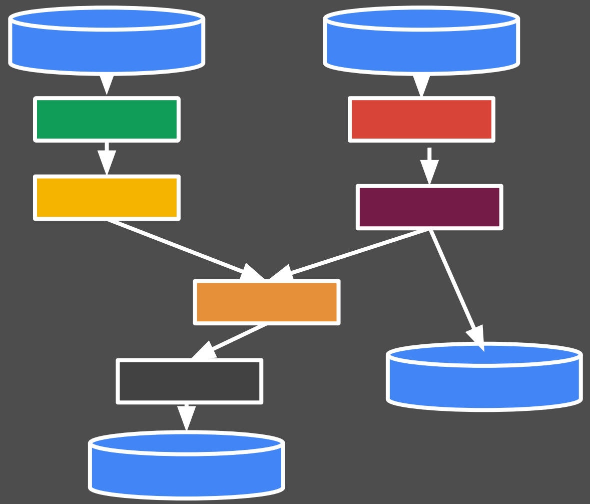


6. Streaming With Sessions

Data processing for realistic workloads

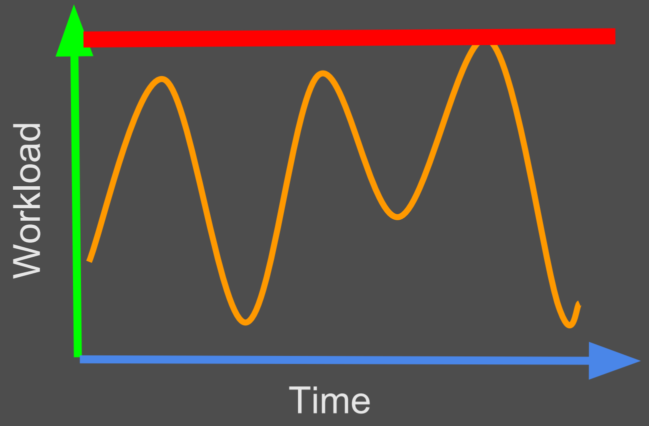


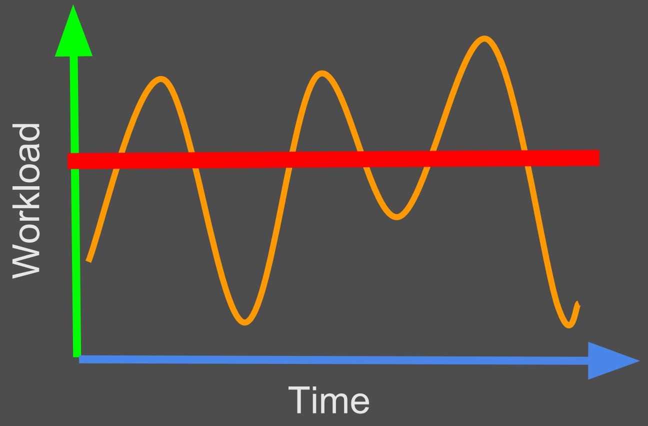
Streaming pipelines have variable input



Batch pipelines have stages of different sizes

The curse of configuration



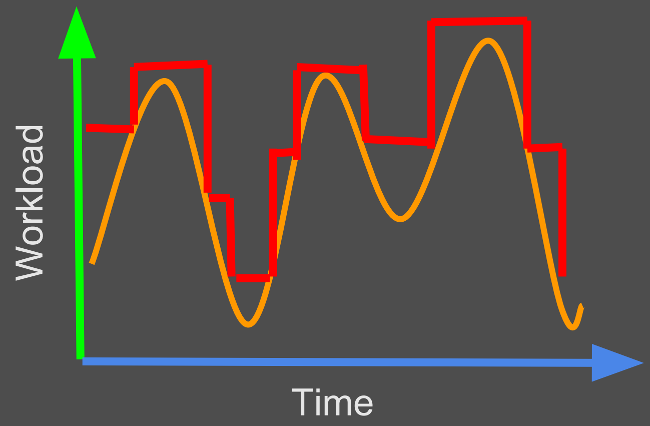


Over-provisioning resources?

Under-provisioning on purpose?

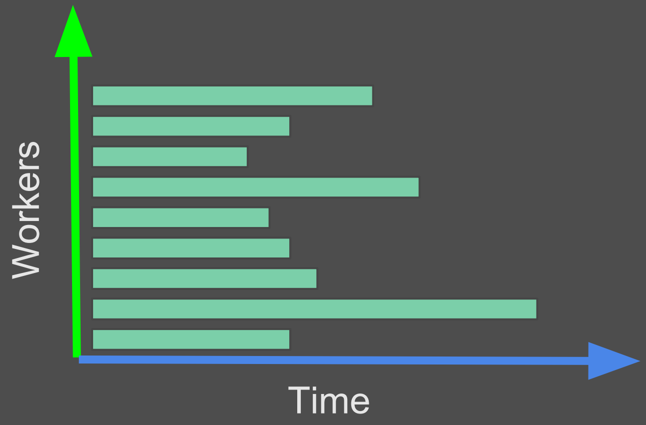
A considerable effort is spent to finely tune all the parameters of the jobs.

Ideal case



A system that adapts.

The straggler problem in batch



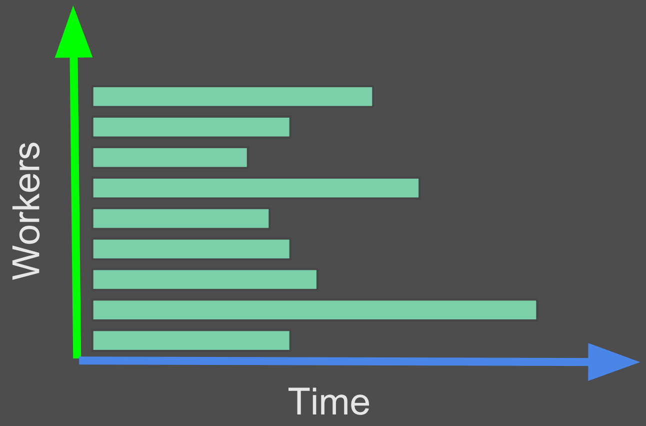
Tasks do not finish evenly on the workers.

- Data is not evenly distributed among tasks
- Processing time is uneven between tasks
- Runtime constraints

Effects are cumulative per stage!

Common straggler mitigation techniques

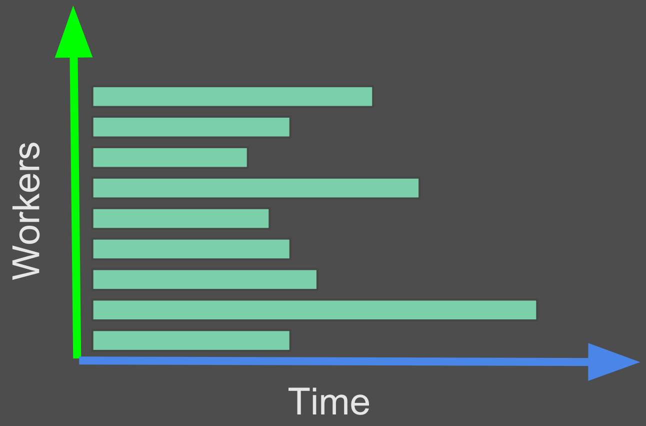
- Split files into equal sizes?
- Pre-emptively over split?
- Detect slow workers and reexecute?
- Sample the data and split based on partial execution



All have major costs, but do not solve completely the problem.

Common straggler mitigation techniques

- Split files into equal sizes?
- Pre-emptively over split?
- Detect slow workers and reexecute?
- Sample the data and split based on partial execution



All have major costs, but do not solve completely the problem.

« The most straightforward way to tune the number of partitions is experimentation: Look at the number of partitions in the parent RDD and then keep multiplying that by 1.5 until performance stops improving. »

From [blog]how-to-tune-your-apache-spark-jobs

No amount of upfront heuristic tuning (be it manual or automatic) is enough to guarantee good performance: the **system will always hit unpredictable situations** at run-time.

A system that's able to **dynamically adapt and get out of a bad situation** is much more powerful than one that **heuristically hopes to avoid** getting into it.

Fine-tuning execution parameters goes against having a truly **portable** and **unified** programming environment.

Beam abstractions empower runners

A **bundle** is group of elements of a PCollection processed and committed together.

APIs (ParDo/DoFn):

- `setup`()
- `startBundle`()
- `processElement`() n times
- `finishBundle`()
- `teardown`()

Streaming runner:

- **small bundles**, low-latency **pipelining** across stages, **overhead** of frequent commits.

Classic batch runner:

- **large bundles**, fewer **large commits**, more **efficient**, **long synchronous stages**.

Other runner strategies may strike a different balance.

Beam abstractions empower runners

Efficiency at runner's discretion

"Read from this source, **splitting it 1000 ways**"

→ **user** decides

"Read from this source"

→ **runner** decides

APIs for portable Sources:

- `long getEstimatedSize()`
- `List<Source> splitIntoBundles(size)`

Beam abstractions empower runners

Efficiency at runner's discretion

"Read from this source, **splitting it 1000 ways**"

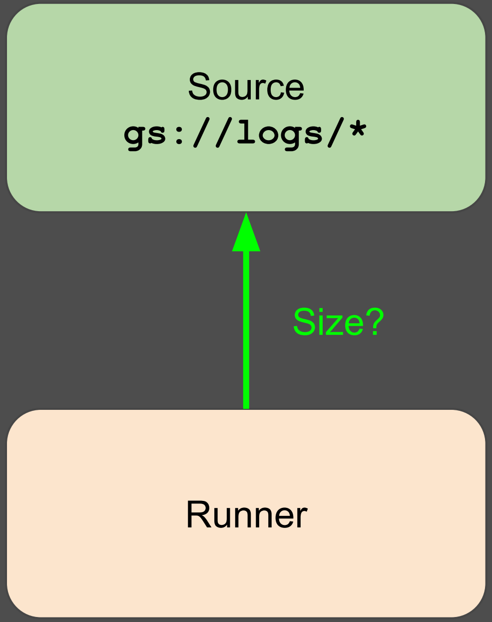
→ **user** decides

"Read from this source"

→ **runner** decides

APIs:

- `long getEstimatedSize()`
- `List<Source> splitIntoBundles(size)`



Beam abstractions empower runners

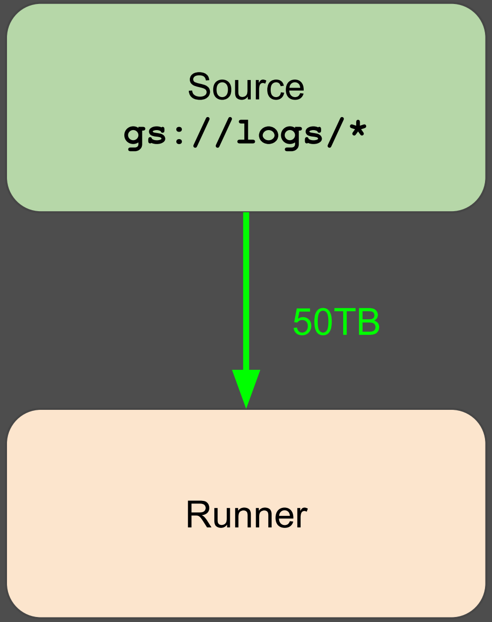
Efficiency at runner's discretion

"Read from this source, **splitting it 1000 ways**"
→ **user** decides

"Read from this source"
→ **runner** decides

APIs:

- `long getEstimatedSize()`
- `List<Source> splitIntoBundles(size)`



Beam abstractions empower runners

Efficiency at runner's discretion

"Read from this source, **splitting it 1000 ways**"

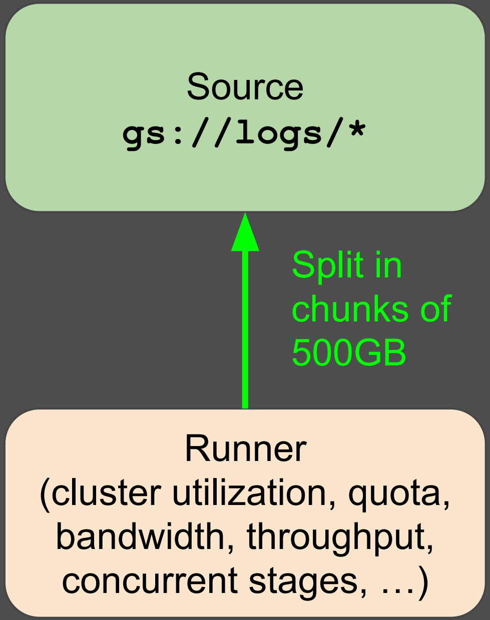
→ **user** decides

"Read from this source"

→ **runner** decides

APIs:

- `long` **`getEstimatedSize`**`()`
- `List<Source>` **`splitIntoBundles`**`(size)`



Beam abstractions empower runners

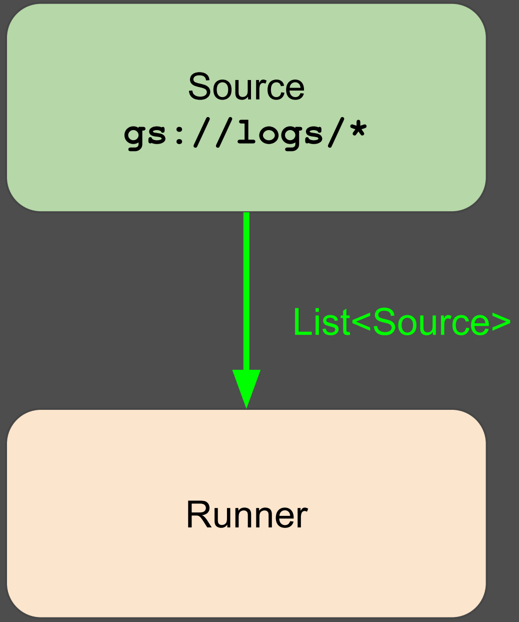
Efficiency at runner's discretion

"Read from this source, **splitting it 1000 ways**"
→ **user** decides

"Read from this source"
→ **runner** decides

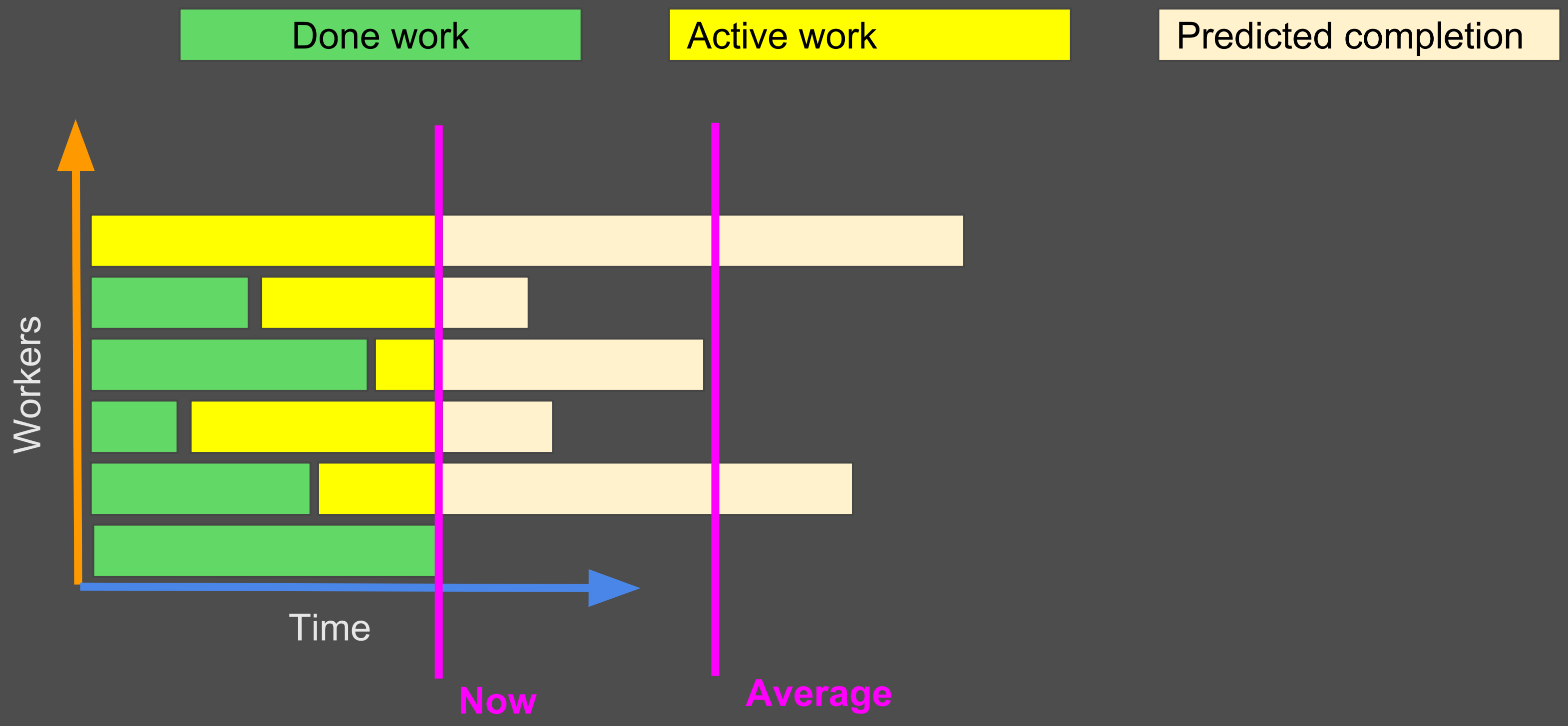
APIs:

- `long getEstimatedSize()`
- `List<Source> splitIntoBundles(size)`

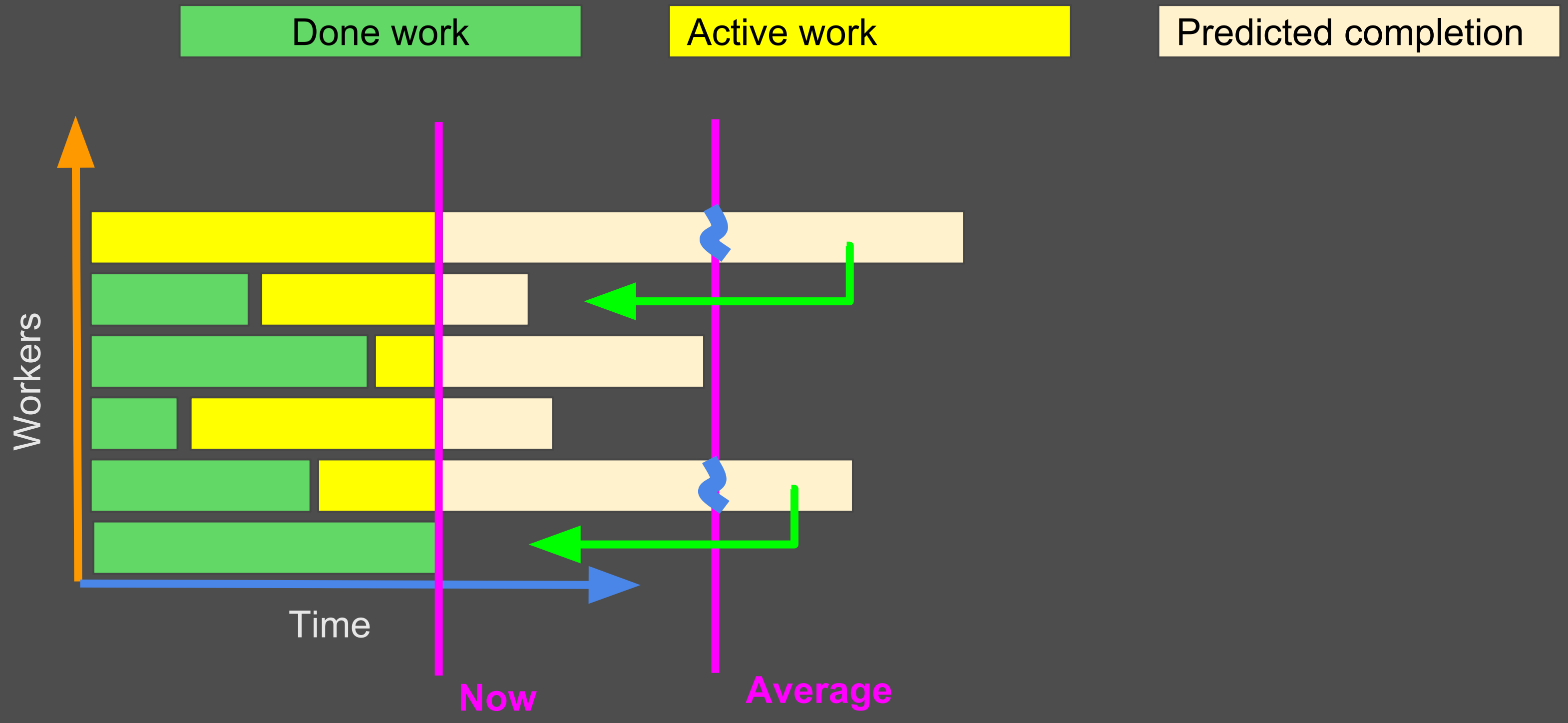


Solving the straggler problem: Dynamic Work Rebalancing

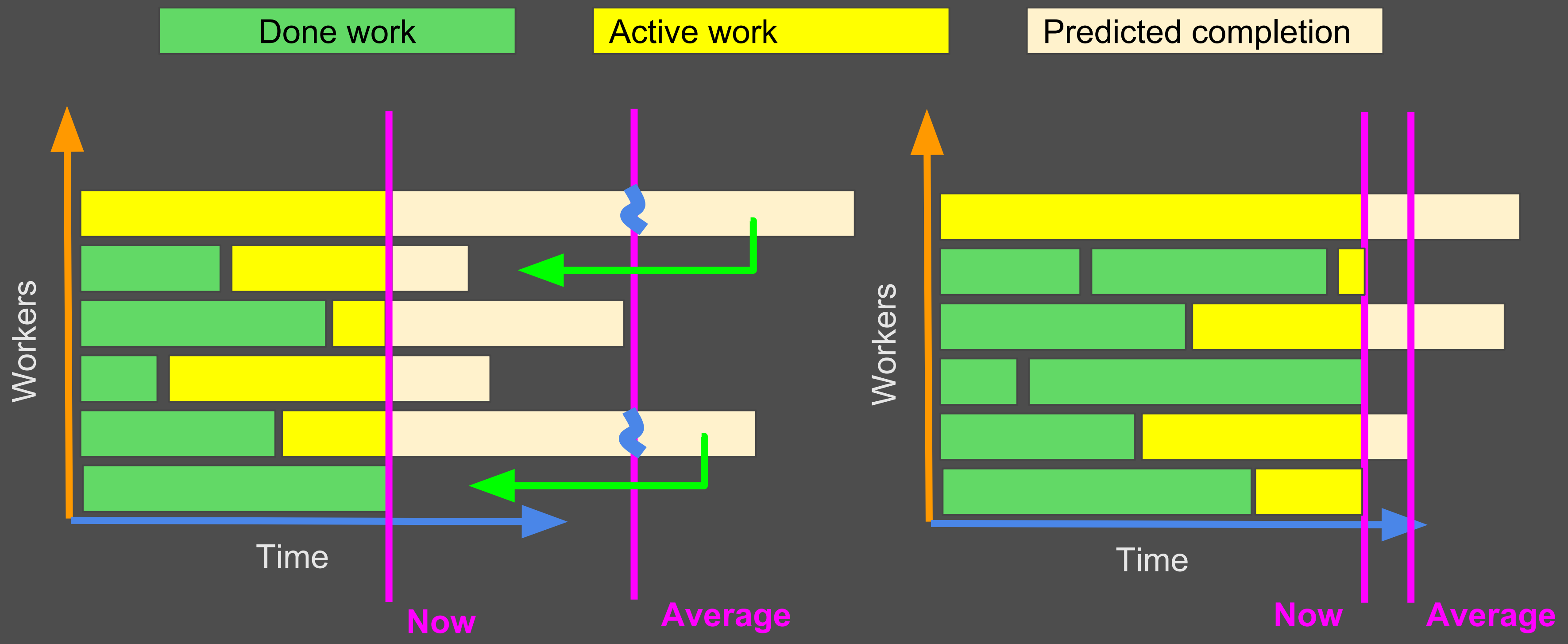
Solving the straggler problem: Dynamic Work Rebalancing



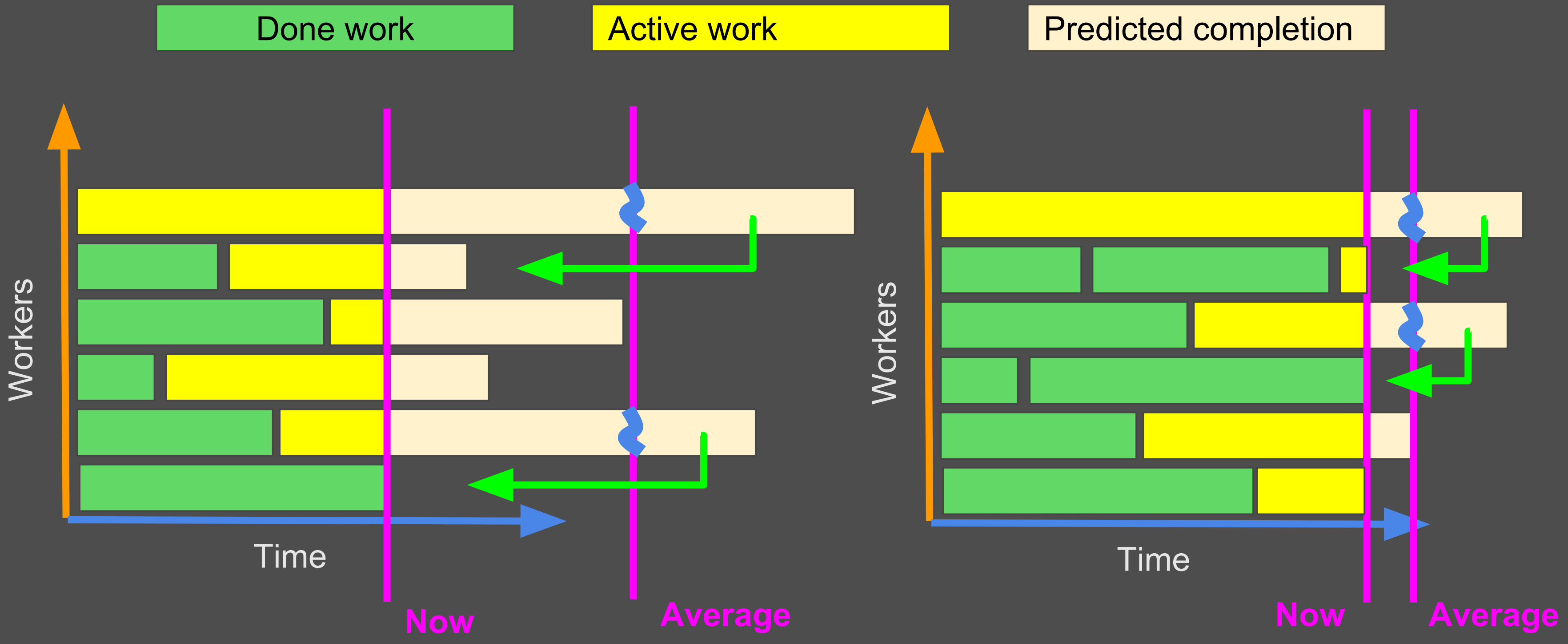
Solving the straggler problem: Dynamic Work Rebalancing



Solving the straggler problem: Dynamic Work Rebalancing



Solving the straggler problem: Dynamic Work Rebalancing

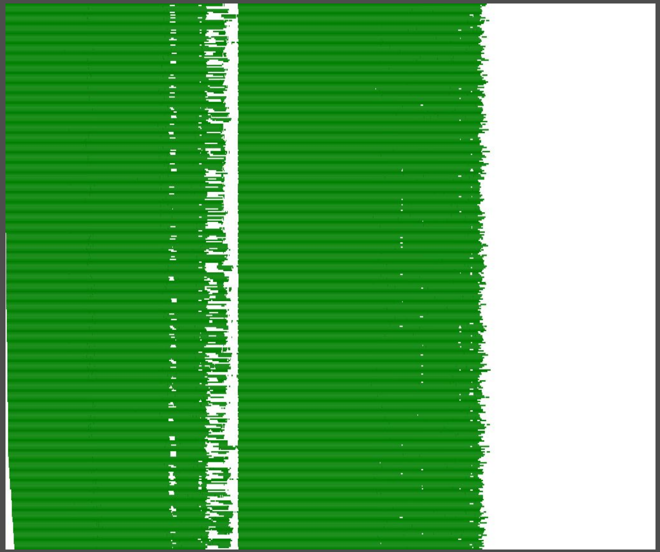


Dynamic Work Rebalancing in the wild

A classic MapReduce job (read from Google Cloud Storage, GroupByKey, write to Google Cloud Storage), 400 workers. Dynamic Work Rebalancing disabled to demonstrate stragglers.
X axis: time (total ~20min.); Y axis: workers

Dynamic Work Rebalancing in the wild

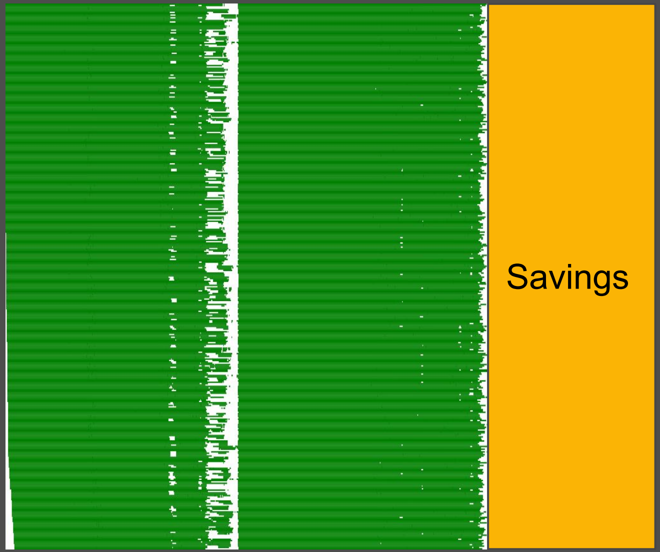
A classic MapReduce job (read from Google Cloud Storage, GroupByKey, write to Google Cloud Storage), 400 workers. Dynamic Work Rebalancing disabled to demonstrate stragglers.
X axis: time (total ~20min.); Y axis: workers



Same job,
Dynamic Work Rebalancing enabled.
X axis: time (total ~15min.); Y axis: workers

Dynamic Work Rebalancing in the wild

A classic MapReduce job (read from Google Cloud Storage, GroupByKey, write to Google Cloud Storage), 400 workers. Dynamic Work Rebalancing disabled to demonstrate stragglers.
X axis: time (total ~20min.); Y axis: workers



Same job,
Dynamic Work Rebalancing enabled.
X axis: time (total ~15min.); Y axis: workers

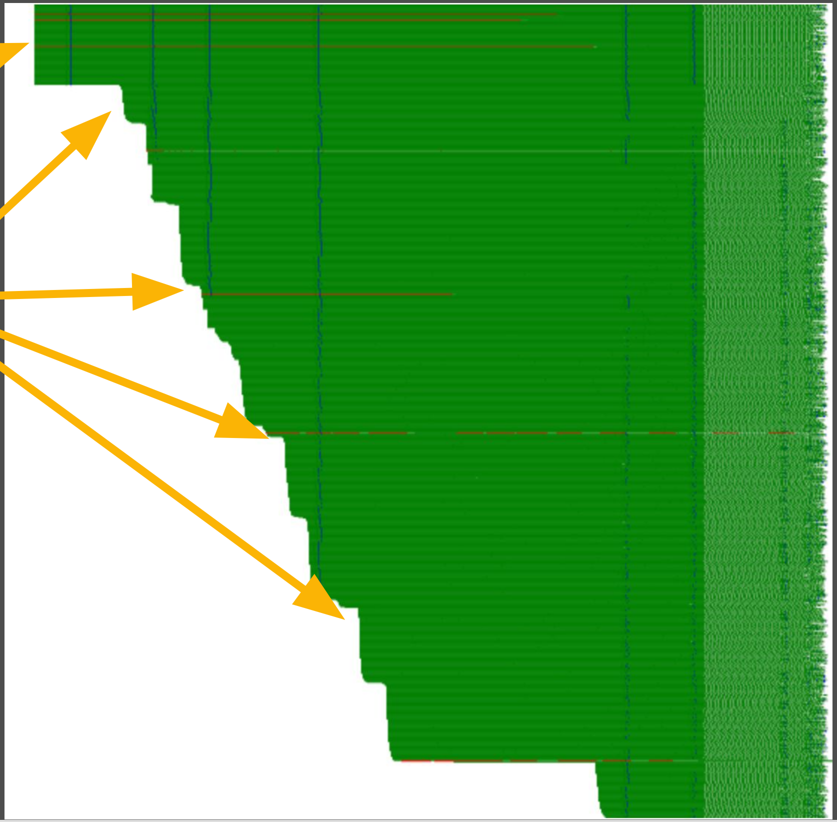
Dynamic Work Rebalancing with Autoscaling

Initial allocation of 80 workers, based on **size**

Multiple rounds of **upsizing**, enabled by dynamic work rebalancing

Upscales to 1000 workers.

- tasks are balanced
- no oversplitting or manual tuning



Apache Beam enable dynamic adaptation

Beam Source Readers provide **simple progress signals**, which enable runners to take action based on execution-time characteristics.

All Beam runners can implement Autoscaling and Dynamic Work Rebalancing.

APIs for how much work is pending.

- **bounded:** `double getFractionConsumed()`
- **unbounded:** `long getBacklogBytes()`

APIs for splitting:

- **bounded:**
  - `Source splitAtFraction(double)`
  - `int getParallelismRemaining()`
- **unbounded:**
  - Coming soon ...

Apache Beam is a **unified** programming model designed to provide **efficient** and **portable** data processing pipelines.

To learn more

Read our blog posts!

- No shard left behind: Dynamic work rebalancing in Google Cloud Dataflow
  https://cloud.google.com/blog/big-data/2016/05/no-shard-left-behind-dynamic-work-rebalancing-in-google-cloud-dataflow
- Comparing Cloud Dataflow autoscaling to Spark and Hadoop
  https://cloud.google.com/blog/big-data/2016/03/comparing-cloud-dataflow-autoscaling-to-spark-and-hadoop

Join the Apache Beam community!
https://beam.apache.org/