Abstract

Apache Beam is a unified programming model capable of expressing a wide variety of both traditional batch and complex streaming use cases. By neatly separating properties of the data from run-time characteristics, Beam enables users to easily tune requirements around completeness and latency and run the same pipeline across multiple runtime environments. In addition, Beam's model enables cutting edge optimizations, like dynamic work rebalancing and autoscaling, giving those runtimes the ability to be highly efficient.

This talk will cover the basics of Apache Beam, touch on its evolution, and describe the main concepts in its powerful programming model. We'll include detailed, concrete examples of how Beam unifies batch and streaming use cases, and show efficient execution in real-world scenarios.
Using **Apache Beam** for **Batch**, **Streaming**, and **Everything in Between**

Dan Halperin (@dhalperi)
Apache Beam PMC
Senior Software Engineer, Google
Expresses data-parallel batch and streaming algorithms with one unified API.

Cleanly separates data processing logic from runtime requirements.

Supports execution on multiple distributed processing runtime environments.

Integrates with the larger data processing ecosystem.
Announcing the First Stable Release

The Apache Software Foundation Announces Apache® Beam™ v2.0.0

Open Source unified programming model for batch and streaming Big Data processing in use at Google Cloud, PayPal, and Talend, among others.

Forest Hill, MD, May 17, 2017 (GLOBE NEWSWIRE) -- The Apache Software Foundation (ASF), the all-volunteer developers, stewards, and incubators of more than 350 Open Source projects and initiatives, announced today the availability of Apache® Beam™ v2.0.0, the first stable release of the unified programming model for both batch and streaming Big Data processing.

An Apache Top-Level Project (TLP) since December 2016, Beam includes Java and Python software development kits used to define data processing pipelines and runners to execute them on Apache Apex, Apache Flink, Apache Spark, and Google Cloud Dataflow, among other execution engines.

Apache Beam has its roots in Google’s internal work on data processing over the last decade, evolving from the initial MapReduce system, through FlumeJava and MillWheel, into Google Cloud Dataflow v1.x, which defined the unified programming model that became the heart of Apache Beam.
Apache Beam at this conference

Using Apache Beam for Batch, Streaming, and Everything in Between
  • Dan Halperin @ 10:15 am

Apache Beam: Integrating the Big Data Ecosystem Up, Down, and Sideways
  • Davor Bonaci, and Jean-Baptiste Onofré @ 11:15 am

Concrete Big Data Use Cases Implemented with Apache Beam
  • Jean-Baptiste Onofré @ 12:15 pm

Nexmark, a Unified Framework to Evaluate Big Data Processing Systems
  • Ismaël Mejía, and Etienne Chauchot @ 2:30 pm
Apache Beam at this conference

Apache Beam Birds of a Feather
• Wednesday, 6:30 pm - 7:30 pm

Apache Beam Hacking Time
• Time: all-day Thursday
• 2nd floor collaboration area
• (depending on interest)
This talk: Apache Beam introduction and update
This talk: **Apache Beam** introduction and update

**Apache Beam** is a **unified** programming model designed to provide **efficient** and **portable** data processing pipelines.
The **Beam Model**: Asking the Right Questions

*What* results are calculated?

*Where* in event time are results calculated?

*When* in processing time are results materialized?

*How* do refinements of results relate?
1. Classic Batch

2. Batch with Fixed Windows

3. Sessions

4. Streaming

5. Streaming with Speculative + Late Data
What is Apache Beam?

The Beam Programming Model
• What / Where / When / How

SDKs for writing Beam pipelines
• Java, Python

Beam Runners for existing distributed processing backends
• Apache Apex
• Apache Flink
• Apache Spark
• Google Cloud Dataflow
Apache Beam is a unified programming model designed to provide efficient and portable data processing pipelines.
Simple clickstream analysis pipeline

**Data:** JSON-encoded analytics stream from site

- `{"user":"dhalperi", "page":"apache.org/feed/7", "tstamp":"2016-08-31T15:07Z", ...}

**Desired output:** Per-user session length and activity level

- dhalperi, 33 pageviews, 2016-08-31 15:04-15:25
Simple clickstream analysis pipeline

**Data:** JSON-encoded analytics stream from site

- \{"user":"dhalperi", "page":"apache.org/feed/7", "tstamp":"2016-08-31T15:07Z", ...\}

**Desired output:** Per-user session length and activity level

- **dhalperi**, 33 pageviews, **2016-08-31 15:04–15:25**
Two example applications

Streaming job consuming Kafka stream
• Uses 10 workers.
• Pipeline lag of a few seconds.
• With a 2 million users over 1 day.

• Want fresh, correct results at low latency
• Okay to use more resources
Two example applications

Streaming job consuming Kafka stream
- Uses **10** workers.
- Pipeline lag of a few seconds.
- With a 2 million users over 1 day.
- Want **fresh, correct results** at low latency
- Okay to use **more resources**

Batch job consuming HDFS archive
- Uses **200** workers.
- Runs for 30 minutes.
- Same input.
- **Accurate results at job completion**
- **Batch efficiency**
Two example applications

Streaming job consuming Kafka stream
- Uses 10 workers.
- Pipeline lag of a few seconds.
- With a 2 million users over 1 day.
- Want fresh, correct results at low latency
- Okay to use more resources

Batch job consuming HDFS archive
- Uses 200 workers.
- Runs for 30 minutes.
- Same input.
- Accurate results at job completion
- Batch efficiency

What does the user have to change to get these results?
Two example applications

Streaming job consuming Kafka stream
- Uses 10 workers.
- Pipeline lag of a few seconds.
- With a 2 million users over 1 day.
- Want fresh, correct results at low latency
- Okay to use more resources

Batch job consuming HDFS archive
- Uses 200 workers.
- Runs for 30 minutes.
- Same input.
- Accurate results at job completion
- Batch efficiency

What does the user have to change to get these results?
A: O(10 lines of code) + Command-line Arguments
Quick overview of the Beam model

Clean abstractions hide details

PCollection – a parallel collection of timestamped elements that are in windows.
Quick overview of the Beam model

Clean abstractions hide details

**PCollection** – a parallel collection of *timestamped elements* that are in *windows*.

**Sources & Readers** – produce PCollections of timestamped elements and a *watermark*.
Quick overview of the Beam model

Clean abstractions hide details

**PCollection** – a parallel collection of *timestamped elements* that are in *windows*.

**Sources & Readers** – produce PCollections of timestamped elements and a *watermark*.

**ParDo** – flatmap over elements of a PCollection.
Quick overview of the Beam model

Clean abstractions hide details

**PCollection** – a parallel collection of *timestamped elements* that are in *windows*.

**Sources & Readers** – produce PCollections of timestamped elements and a *watermark*.

**ParDo** – flatmap over elements of a PCollection.

**(Co)****GroupByKey** – shuffle & group \( \{K: V\} \rightarrow \{K: [V]\} \).
Quick overview of the Beam model

Clean abstractions hide details

**PCollection** – a parallel collection of **timestamped elements** that are in **windows**.

**Sources & Readers** – produce PCollections of timestamped elements and a **watermark**.

**ParDo** – flatmap over elements of a PCollection.

**(Co)** **GroupByKey** – shuffle & group $\{\{K: V\}\} \rightarrow \{K: [V]\}$.

**Side inputs** – global view of a PCollection used for broadcast / joins.
Quick overview of the Beam model

Clean abstractions hide details

**PCollection** – a parallel collection of *timestamped elements* that are in *windows*.

**Sources & Readers** – produce PCollections of timestamped elements and a *watermark*.

**ParDo** – flatmap over elements of a PCollection.

**(Co)** **GroupByKey** – shuffle & group \{\{K: V\}\} → \{K: [V]\}.

**Side inputs** – global view of a PCollection used for broadcast / joins.

**Window** – reassign elements to zero or more windows; may be *data-dependent*. 
Quick overview of the Beam model

Clean abstractions hide details

**PCollection** – a parallel collection of *timestamped elements* that are in *windows*.

**Sources & Readers** – produce PCollections of timestamped elements and a *watermark*.
**ParDo** – flatmap over elements of a PCollection.
**(Co)**GroupByKey – shuffle & group \( \{\{K: V\}\} \rightarrow \{K: [V]\} \).

**Side inputs** – global view of a PCollection used for broadcast / joins.

**Window** – reassign elements to zero or more windows; may be *data-dependent*.
**Triggers** – user flow control based on *window, watermark, element count, lateness*. 
Quick overview of the Beam model

Clean abstractions hide details

**PCollection** – a parallel collection of *timestamped elements* that are in *windows*.

**Sources & Readers** – produce PCollections of timestamped elements and a *watermark*.

**ParDo** – flatmap over elements of a PCollection.

**(Co)**GroupByKey – shuffle & group \{\{K: V\}\} → \{K: [V]\}.

**Side inputs** – global view of a PCollection used for broadcast / joins.

**Window** – reassign elements to zero or more windows; may be *data-dependent*.

**Triggers** – user flow control based on window, watermark, element count, lateness.

**State & Timers** – cross-element data storage and callbacks enable complex operations
1. Classic Batch

2. Batch with Fixed Windows

3. Sessions

4. Streaming

5. Streaming with Speculative + Late Data
Simple clickstream analysis pipeline

```java
PCollection<KV<User, Click>> clickstream =
    pipeline.apply(IO.Read(...))
    .apply(MapElements.of(new ParseClicksAndAssignUser()));

PCollection<KV<User, Long>> userSessions =
    clickstream.apply(Window.into(Sessions.withGapDuration(Minutes(3)))
                       .triggering(AtWatermark().withEarlyFirings(AtPeriod(Minutes(1))))
                       .apply(Count.perKey()));

userSessions.apply(MapElements.of(new FormatSessionsForOutput()))
    .apply(IO.Write(...));

pipeline.run();
```
Unified unbounded & bounded PCollections

pipeline.apply(IO.Read(...)).apply(MapElements.of(new ParseClicksAndAssignUser()));

Apache Kafka, ActiveMQ, tailing filesystem, ...
- A live, roughly in-order stream of messages, *unbounded PCollections*.
- KafkaIO.read().fromTopic(“pageviews”)

HDFS, Google Cloud Storage, yesterday’s Kafka log, ...
- Archival data, often readable in any order, *bounded PCollections*.
- TextIO.read().from(“hdfs://apache.org/pageviews/*”)
Windowing and triggers

```java
PCollection<KV<User, Long>> userSessions =
    clickstream.apply(
        Window.into(Sessions.withGapDuration(Minutes(3)))
            .triggering(
                AtWatermark()
                    .withEarlyFirings(AtPeriod(Minutes(1))))
    )
```

Event time

3:00 3:05 3:10 3:15 3:20 3:25
Windowing and triggers

```java
PCollection<KV<User, Long>> userSessions =
    clickstream.apply(Window.into(Sessions.withGapDuration(Minutes(3)))
        .triggering(
            AtWatermark()
                .withEarlyFirings(AtPeriod(Minutes(1)))))
```

One session, 3:04-3:25
Windowing and triggers

```java
PCollection<KV<User, Long>> userSessions =
    clickstream.apply(Windows.into(Sessions.withGapDuration(Minutes(3))))
        .triggering(
            AtWatermark()
                .withEarlyFirings(AtPeriod(Minutes(1))))
```
Windowing and triggers

PCollection<KV<User, Long>> userSessions =
  clickstream.apply(Window.into(Sessions.withGapDuration(Minutes(3))))
  .triggering(
    AtWatermark()
    .withEarlyFirings(AtPeriod(Minutes(1))))

(EARLY) 1 session, 3:04–3:10
Windowing and triggers

```java
PCollection<KV<User, Long>> userSessions =
    clickstream.apply(Windows.into(Sessions.withGapDuration(Minutes(3)))
        .triggering(
            AtWatermark()
            .withEarlyFirings(AtPeriod(Minutes(1))))
```

(\textit{EARLY}) 2 sessions, 3:04–3:10 \& 3:15–3:20

(\textit{EARLY}) 1 session, 3:04–3:10
Windowing and triggers

```java
PCollection<KV<User, Long>> userSessions =
clickstream.apply(Window.into(Sessions.withGapDuration(Minutes(3)))
   .triggering(
       AtWatermark()
       .withEarlyFirings(AtPeriod(Minutes(1))))
```

Processing time

Event time

1 session, 3:04–3:25

( EARLY ) 2 sessions, 3:04–3:10 & 3:15–3:20

( EARLY ) 1 session, 3:04–3:10
Windowing and triggers

```java
PCollection<KV<User, Long>> userSessions =
    clickstream.apply(Windows.into(Sessions.withGapDuration(Minutes(3))).
                        triggering(AtWatermark().
                                   withEarlyFirings(AtPeriod(Minutes(1))))));
```
Writing output, and bundling

```java
userSessions.apply(MapElements.of(new FormatSessionsForOutput()))
  .apply(IO.Write(...));
```

Writing is the dual of reading — format and then output.

Fault-tolerant side-effects: exploit sink semantics get effectively once delivery:

- **deterministic** operations,
- **idempotent** operations (create, delete, set),
- transactions / unique operation IDs,
- or within-pipeline state.
Two example runs of this pipeline

Streaming job consuming Kafka stream
- Uses 10 workers.
- Pipeline lag of a few seconds.
- With a 2 million users over 1 day.
- A total of ~4.7M early + final sessions.
- 240 worker-hours

Batch job consuming HDFS archive
- Uses 200 workers.
- Runs for 30 minutes.
- Same input.
- A total of ~2.1M final sessions.
- 100 worker-hours

With Apache Beam, the same pipeline works for both — just switch I/O.
Apache Beam is a unified programming model designed to provide efficient and portable data processing pipelines.
Beam abstractions empower runners

Efficiency at runner’s discretion

“Read from this source, splitting it 1000 ways”

• user decides, via trial and error at small scale
  and hopes that it works

“Read from this source”

• and the runner decides
Beam abstractions empower runners

Efficiency at runner’s discretion

“Read from this source, splitting it 1000 ways”
• user decides, via trial and error at small scale and hopes that it works

“Read from this source”
• and the runner decides

APIs:
• long getEstimatedSize()
• List<Source> split(size)
Beam abstractions empower runners

Efficiency at runner’s discretion

“Read from this source, splitting it 1000 ways”

• user decides, via trial and error at small scale and hopes that it works

“Read from this source”

• and the runner decides

APIs:

• long `getEstimatedSize()`
• `List<Source> split(size)`
Beam abstractions empower runners

Efficiency at runner’s discretion

“Read from this source, splitting it 1000 ways”
• user decides, via trial and error at small scale and hopes that it works

“Read from this source”
• and the runner decides

APIs:
• long getEstimatedSize()
• List<Source> split(size)
Beam abstractions empower runners

Efficiency at runner’s discretion

“Read from this source, splitting it 1000 ways”
  • user decides, via trial and error at small scale and hopes that it works

“Read from this source”
  • and the runner decides

APIs:
  • `long getEstimatedSize()`
  • `List<Source> split(size)`

hdfs://logs/*

50 TiB
Beam abstractions empower runners

Efficiency at runner’s discretion

“Read from this source, splitting it 1000 ways”
  • user decides, via trial and error at small scale and hopes that it works

“Read from this source”
  • and the runner decides

APIs:
  • long getEstimatedSize()
  • List<Source> split(size)
Beam abstractions empower runners

Efficiency at runner’s discretion

“Read from this source, splitting it 1000 ways”
• user decides, via trial and error at small scale and hopes that it works

“Read from this source”
• and the runner decides

APIs:
• long `getEstimatedSize()`
• `List<Source> split(size)`
Beam abstractions empower runners

Efficiency at runner’s discretion

“Read from this source, splitting it 1000 ways”

• user decides, via trial and error at small scale and hopes that it works

“Read from this source”

• and the runner decides

APIs:

• long `getEstimatedSize()`
• List<Source> `split(size)`

Cluster utilization?

Quota? Bandwidth?

Reservations?

Bottleneck? Throughput?
Bundling and runner efficiency

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

APIs (ParDo/DoFn):
- `startBundle()`
- `processElement()` \(n\) times
- `finishBundle()`
Bundling and runner efficiency

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

APIs (ParDo/DoFn):
- `startBundle()`
- `processElement() n times`
- `finishBundle()`
Bundling and runner efficiency

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

APIs (ParDo/DoFn):
• `startBundle()`
• `processElement()` \( n \) times
• `finishBundle()`

**Streaming** runner: small bundles, low-latency pipelining across stages, overhead of frequent commits.
Bundling and runner efficiency

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

APIs (ParDo/DoFn):
• \texttt{startBundle()}
• \texttt{processElement()} \textit{n times}
• \texttt{finishBundle()}

\textbf{Streaming} runner: small bundles, low-latency pipelining across stages, overhead of frequent commits.

\textbf{Classic batch} runner: large bundles, fewer large commits, more efficient, long synchronous stages.
Bundling and runner efficiency

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

**APIs (ParDo/DoFn):**

- `startBundle()`
- `processElement()` \( n \) times
- `finishBundle()`

**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 strike a different balance.
Runner-controlled triggering

Beam triggers are flow control, not instructions.
• “it is okay to produce data” not “produce data now”.
• Runners decide when to produce data, and can make local choices for efficiency.

Streaming clickstream analysis: runner may optimize for latency and freshness.
• Small bundles and frequent triggering → more files and more (speculative) records.

Batch clickstream analysis: runner may optimize for throughput and efficiency.
• Large bundles and no early triggering → fewer large files and no early records.
Pipeline workload varies

Streaming pipeline’s input varies

Batch pipelines go through stages
Perils of fixed decisions

Over-provisioned / worst case

Under-provisioned / average case
Ideal case
The Straggler Problem

Work is unevenly distributed across tasks.

Reasons:
  • Underlying data.
  • Processing.
  • Runtime effects.

Effects are cumulative per stage.
Standard workarounds for stragglers

- Split files into equal sizes?
- Preemptively over-split?
- Detect slow workers and re-execute?
- Sample extensively and then split?
Standard workarounds for stragglers

Split files into equal sizes?

Preemptively over-split?

Detect slow workers and re-execute?

Sample extensively and then split?

All of these have major costs, none is a complete solution.
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.
Beam readers enable dynamic adaptation

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

APIs for how much work is pending:
- Bounded: `double getFractionConsumed()`
- Unbounded: `long getBacklogBytes()`

Work-stealing:
- Bounded: `Source splitAtFraction(double)`
  `int getParallelismRemaining()`
Dynamic work rebalancing

- Done work
- Active work
- Predicted completion

Tasks

Time

Now

Average
Dynamic work rebalancing

- **Done work**
- **Active work**
- **Predicted completion**

Tasks

Time

Now
Dynamic work rebalancing

- **Done work**
- **Active work**
- **Predicted completion**
Dynamic work rebalancing

Tasks

Now

Done work
Active work
Predicted completion
Dynamic Work Rebalancing: a real example

Beam pipeline on the Google Cloud Dataflow runner

2-stage pipeline, split “evenly” but uneven in practice
Dynamic Work Rebalancing: a real example
Beam pipeline on the Google Cloud Dataflow runner

2-stage pipeline, split “evenly” but uneven in practice

Same pipeline dynamic work rebalancing enabled
Dynamic Work Rebalancing: a real example

Beam pipeline on the Google Cloud Dataflow runner

2-stage pipeline, split “evenly” but uneven in practice

Same pipeline dynamic work rebalancing enabled

Savings
Dynamic Work Rebalancing + Autoscaling

Beam pipeline on the Google Cloud Dataflow runner
Dynamic Work Rebalancing + Autoscaling
Beam pipeline on the Google Cloud Dataflow runner

Initially allocate ~80 workers based on size
Dynamic Work Rebalancing + Autoscaling

Beam pipeline on the Google Cloud Dataflow runner

Initially allocate ~80 workers based on size

Multiple rounds of upsizing enabled by dynamic splitting
Dynamic Work Rebalancing + Autoscaling

Beam pipeline on the Google Cloud Dataflow runner

Initially allocate ~80 workers based on size

Multiple rounds of upsizing enabled by dynamic splitting

Upscale to 1000 workers
* tasks stay well-balanced
* without oversplitting initially
Dynamic Work Rebalancing + Autoscaling

Beam pipeline on the Google Cloud Dataflow runner

Initially allocate ~80 workers based on size

Multiple rounds of upsizing enabled by dynamic splitting

Upscale to 1000 workers
* tasks stay well-balanced
* without oversplitting initially

Long-running tasks aborted without causing stragglers
Apache Beam is a unified programming model designed to provide efficient and portable data processing pipelines.
Write a pipeline once, run it anywhere
Unified model for portable execution

```java
PCollection<KV<User, Click>> clickstream =
    pipeline.apply(IO.Read(...))
    .apply(MapElements.of(new ParseClicksAndAssignUser()));

PCollection<KV<User, Long>> userSessions =
    clickstream.apply(Window.into(Sessions.withGapDuration(Minutes(3)))
    .triggering(AtWatermark()
      .withEarlyFirings(AtPeriod(Minutes(1))))
    .apply(Count.perKey());

userSessions.apply(MapElements.of(new FormatSessionsForOutput()))
  .apply(IO.Write(...));

pipeline.run();
```
Get involved with Beam

If you analyze data: Try it out!

If you have a data storage or messaging system: Write a connector!

If you have Big Data APIs: Write a Beam SDK, DSL, or library!

If you have a distributed processing backend:
• Write a Beam Runner!
  (& join Apache Apex, Flink, Spark, Gearpump, and Google Cloud Dataflow)
Apache Beam at this conference

Using Apache Beam for Batch, Streaming, and Everything in Between
- Dan Halperin @ 10:15 am

Apache Beam: Integrating the Big Data Ecosystem Up, Down, and Sideways
- Davor Bonaci, and Jean-Baptiste Onofré @ 11:15 am

Concrete Big Data Use Cases Implemented with Apache Beam
- Jean-Baptiste Onofré @ 12:15 pm

Nexmark, a Unified Framework to Evaluate Big Data Processing Systems
- Ismaël Mejía, and Etienne Chauchot @ 2:30 pm
Apache Beam at this conference

Apache Beam Birds of a Feather
• Wednesday, 6:30 pm - 7:30 pm

Apache Beam Hacking Time
• Time: all-day Thursday
• 2nd floor collaboration area
• (depending on interest)
Learn more

Apache Beam website:
  • https://beam.apache.org/
  • documentation, mailing lists, downloads, etc.

Read our blog posts!
  • Streaming 101 & 102: oreilly.com/ideas/the-world-beyond-batch-streaming-101
  • No shard left behind: Dynamic work rebalancing in Google Cloud Dataflow
  • Apache Beam blog: http://beam.apache.org/blog/

Follow @ApacheBeam on Twitter