Radically modular data ingestion APIs in Apache Beam

Eugene Kirpichov <kirpichov@google.com>
Staff Software Engineer
Plan

01 Intro to Beam
   Unified, portable data processing

02 IO — APIs for data ingestion
   What's the big deal

03 Composable IO
   IO as data processing

04 Splittable DoFn
   Missing piece for composable sources

05 Recap
   If you remember two things
01 Intro to Beam

Unified, portable data processing
(2004) MapReduce
SELECT + GROUPBY

(2008) FlumeJava
High-level API

(2013) Millwheel
Deterministic streaming

(2014) Dataflow
Batch/streaming agnostic, Portable across languages & runners

(2016) Apache Beam
Open, Community-driven, Vendor-independent
Batch vs. streaming is moot

— Beam

(Batch is nearly always part of higher-level streaming)
Choose your language...

Java  Python  Other languages

Beam Model

...and your runtime.

[Logos for APEX, Pig, Spark, Dremel, and others]
Beam PTransforms

- **DoFn**
- **ParDo**
  *(good old FlatMap)*
- **GroupByKey**
- **Composite**
User code

Libraries of PTransforms, IO

SDK (per language)

Runner
Pipeline p = Pipeline.create(options);

PCollection<String> lines = p.apply(
    TextIO.read().from("gs://.../*"));

PCollection<KV<String, Long>> wordCounts = lines
    .apply(FlatMapElements.via(word → word.split("\W+")))
    .apply(Count.perElement());

wordCounts
    .apply(MapElements.via(
        count → count.getKey() + "\": " + count.getValue())
    .apply(TextIO.write().to("gs://.../..."));

p.run();
02 IO - APIs for data ingestion

What's the big deal
Beam IO

Files
- Text/Avro/XML/...
  - HDFS, S3, GCS

Kafka
Kinesis
AMQP
Pubsub
JMS

Hadoop
MQTT
JDBC
MongoDb
Redis
Cassandra
HBase

Hive
Solr
Elasticsearch
BigQuery
BigTable
Datastore
Spanner
IO is essential

Most pipelines move data from X to Y

ETL: Extract, Transform, Load
IO is **messy**

Cozy, pure programming model
IO is *messy*

**Cozy, pure programming model**
## IO is **messy**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read via CSV dump</td>
<td></td>
</tr>
<tr>
<td>Clean up temp files</td>
<td>Read tons of small files</td>
</tr>
<tr>
<td>Preserve filenames</td>
<td>Skip headers</td>
</tr>
<tr>
<td>Route to different tables</td>
<td>Write A, then write B</td>
</tr>
<tr>
<td>Decompress ZIP</td>
<td>Write to A, then read B</td>
</tr>
<tr>
<td>Dead-letter failed records</td>
<td>Read multiple tables in tx</td>
</tr>
<tr>
<td>Read multiple tables in tx</td>
<td>Stream new files</td>
</tr>
<tr>
<td>Quotas &amp; size limits</td>
<td>Rate limiting / throttling</td>
</tr>
</tbody>
</table>

...
IO is **unified**

(batch/streaming agnostic)

**Classic batch**
- Read files
- Write files

**Classic streaming**
- Read Kafka
- Stream to Kafka

**Reality**
- Read files + watch new files
- Stream files
- Read Kafka from start + tail
IO is **unified**

*(batch/streaming agnostic)*

Evolves changes

**Keeps output = f(input)**

changes Evolves

IO is **unforgiving**

**Correctness**
- Any bug = data corruption
- Fault tolerance
- Exactly-once reads/writes
- Error handling

**Performance**
- Unexpected scale
- Throughput, latency, memory, parallelism
IO is a chance to do better

Nobody writes a paper about their IO API.
(MapReduce paper — 3 paragraphs; Spark, Flink, Beam: 0)

Requirements too diverse
  to support everything out of the box
APIs too rigid
  to let users do it themselves

I made a bigdata programming model

Cool, how does data get in and out?

Brb
IO is essential, but messy and unforgiving.

It begs for good abstractions.
03 Composable IO

IO as data processing
Traditionally: ad-hoc API, at pipeline boundary

"Source"  "Transform"  "Sink"

InputFormat / Receiver / SourceFunction / ...

OutputFormat / Sink / SinkFunction / ...

Configuration:
Filepattern
Query string
Topic name
...

Configuration:
Directory
Table name
Topic name
...

A  B
Traditionally: ad-hoc API, at pipeline boundary

"Source"

My filenames come on a Kafka topic.

I have a table per client + table of clients

"Transform"

Narrow APIs are not hackable

I want to know which records failed to write

I want to kick off another transform after writing

"Sink"
IO is just another data processing task

Globs → Parse files → Records → Parameters → Execute queries → Rows

Rows → Import to database → Invalid rows

Import statistics
IO is just another data processing task
Composability
(aka hackability)

Unified batch/streaming

Transparent fault tolerance

Scalability
(read 1M files = process 1M elements)

Monitoring, debugging

Orchestration
(do X, then read / write, then do Y)

Future features

The rest of the programming model has been getting this for free all along.

Join the party.
IO in Beam: just transforms

```java
JdbcIO.<T>read().from(query)
```
BigQueryIO.write():
(write to files, call import API)
Dynamic routing
Cleanup
Sharding to fit under API limits
...

Pretty complex, but arbitrarily powerful
Composability ⇒ Modularity

What can be composed, can be decomposed.

Image credit: Wikimedia
Read text file globs

Glob ➔ Expand globs ➔ Filename ➔ Read as text file (tail -f) ➔ String

Read Kafka topic

Topic ➔ List partitions ➔ Topic, partition ➔ Watch new results ➔ Read partition ➔ Message
**Read DB via CSV**

- **Invoke dump**
- **Glob**
- **Parse CSV**

**Write db via CSV**

- **Format CSV**
- **Write to text files**
- **Invoke import**
- **Done signal**
Consistent import into 2 databases
What can be composed, can be decomposed.
What this means for you

Library authors
- Ignore native IO APIs if possible
- Unify batch & streaming
- Decompose ruthlessly

Users
- Ignore native IO APIs if possible
- Assemble what you need from powerful primitives
04 Splittable DoFn

Missing piece for composable sources
Typical IO transforms

Read

Write

Split → Read each

ParDo

{ REST call }
Read Kafka topic

List partitions

Watch new results

Infinite output per input

Topic

Message

Read text file globs

Expand globs

Watch new results

No parallelism within file*

Filename

String

Glob

Read as text file

*No Shard Left Behind: Straggler-free data processing in Cloud Dataflow
What ParDo can't do

Per-element work is **indivisible black box**

⇒ can't be **infinite**

⇒ can't be **parallelized further**
Splittable DoFn (SDF): Partial work via restrictions

Element: what work

Restriction: what part of the work

Design: s.apache.org/splittable-do-fn
### Example restrictions

<table>
<thead>
<tr>
<th>Element</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading splittable files</td>
<td>filename, start offset, end offset</td>
</tr>
<tr>
<td>Reading Bigtable</td>
<td>table, filter, columns, start key, end key</td>
</tr>
<tr>
<td>Reading Kafka</td>
<td>topic, partition, start offset, end offset</td>
</tr>
</tbody>
</table>
Splitting restriction
ReadFn( foo.avro, [0, 100) )

ReadFn( foo.avro, [0, 30) )

ReadFn( foo.avro, [30, 70) )

ReadFn( foo.avro, [70, 100) )
Unbounded work per element

ReadKafkaFn( some-topic, [100, inf) )
Anatomy of an SDF

How to process 1 element?  
Read a text file: \(\text{(String filename)} \rightarrow \text{records}\)

How to do it in parts?  
Reading byte sub-ranges

How to describe 1 part? (restriction)  
\{long start, long end\}

How to do this part of this element?  
\[
f = \text{open(element)}; \\
f.\text{seek(start)}; \\
\text{while(f.tell() < end) \{ yield f.readLine(); \}}
\]
Dynamic splitting of restrictions
(basically work stealing)

Restriction

process(e, r)

Split!

Runner

Primary (keeps running)

Residual (can start in parallel)
class ReadAvroFn extends DoFn<Filename, AvroRecord> {
    
    void processElement(ProcessContext c, OffsetRange range) {

        try (AvroReader r = Avro.open(c.element())) {
            for (r.seek(range.start()); r.currentBlockOffset() < range.end(); r.readNextBlock()) {
                for (AvroRecord record : r.currentBlock()) {
                    c.output(record);
                }
            }
        }
    }
}
class ReadAvroFn extends DoFn<Filename, AvroRecord> {
    void processElement(ProcessContext c, OffsetRange range) {
        try (AvroReader r = Avro.open(c.element())) {
            for (r.seek(range.start());
                 r.currentBlockOffset() < range.end();
                 r.readNextBlock()) {
                for (AvroRecord record : r.currentBlock()) {
                    c.output(record);
                }
            }
        }
    }
}
Concurrent splitting

Runner: Avoid returning something already done

process() call: Avoid doing something already returned

Idea: Claiming

Contract: process() claims work before doing it
Split off only unclaimed work
Restriction trackers, Blocks and Positions

```
ReadFn( foo.avro, [30, 70] )
```

RestrictionTracker
{ restriction, what part is claimed }

Block
Unit of claiming (indivisible portion of work within restriction)

Position
Identifies a block within restriction
void processElement(ProcessContext c, OffsetRangeTracker tracker) {
    try (AvroReader r = Avro.open(c.element()) {
        for (r.seek(tracker.start());
            tracker.tryClaim(r.currentBlockOffset());
            r.readNextBlock();
        for (AvroRecord record : r.currentBlock()) {
            c.output(record);
        }
    }
    true ⇒ safe to process (won't be split off)
    false ⇒ stop — hit end of restriction
Role in Beam APIs

**Fundamental building block** for splittable work *(primarily, reading data)*

Unbounded *(checkpoints)*

Dynamically splittable

**Enables library authors** to create higher-level building blocks

Matching globs, reading files, reading topics, …
05 Recap

If you remember two things
Recap: Context

Batch vs. streaming is moot *(including IO)*

IO is essential, messy, unforgiving

Traditionally: special APIs, neglected, inflexible
Begs for better abstractions
If you remember two things

Composable IO = data processing
  Full power of programming model
  Boycott native APIs
  Composable = decomposable
    *(smaller building blocks)*

Splittable DoFn
  **Element:** what work
  **Restriction:** what part of work.
  Enables composable IO for sources