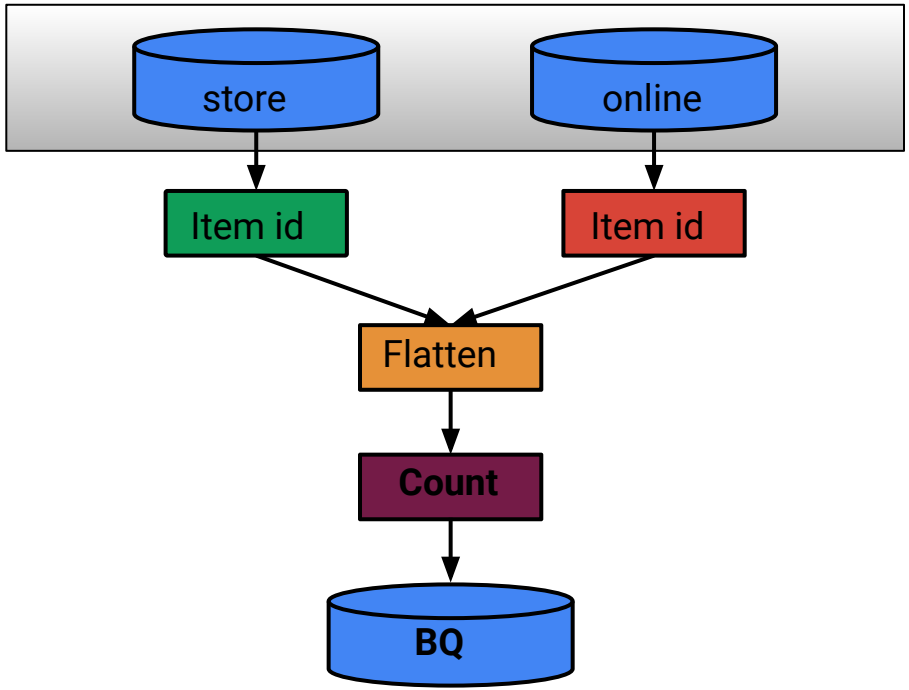
IO Connectors

Goals

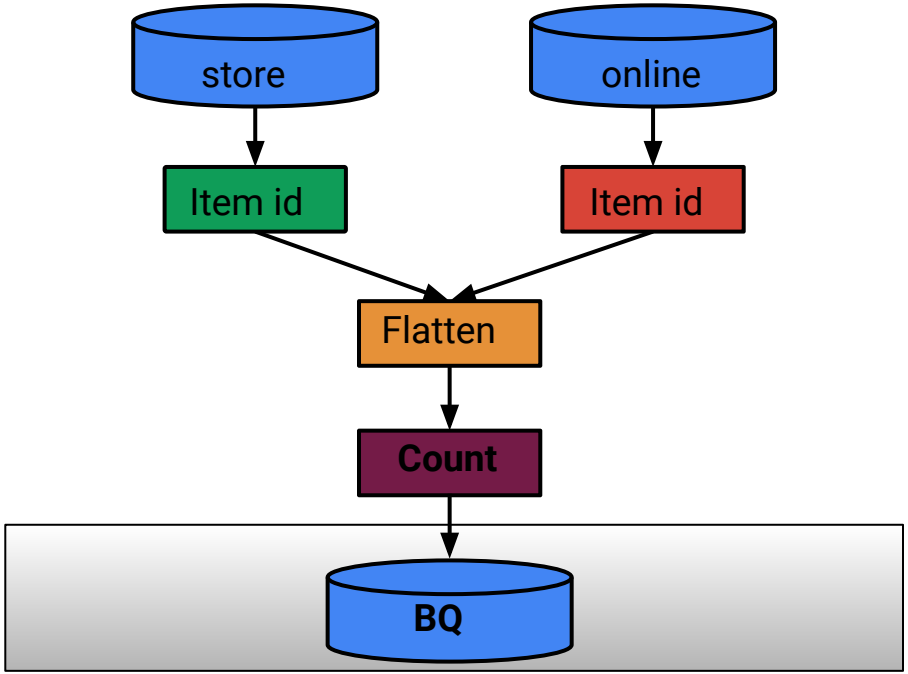
- Understanding Apache Beams Pipeline I/O through Sources and Sinks
- Pipeline I/O via Sources and Sinks.
- Overview of Bounded and Unbounded models
- Real-world patterns:
  - Files, BigQuery, and Kafka.
- Value of Splittable DoFns for writing I/O.

Reading Input Data

Source



Writing output Data



Sink

Beam Provided I/O transforms - Java

Name	Description	Javadoc
FileIO	General-purpose transforms for working with files: listing files (matching), reading and writing.	org.apache.beam.sdk.io.FileIO
AvroIO	PTransforms for reading from and writing to Avro files.	org.apache.beam.sdk.io.AvroIO
TextIO	PTransforms for reading and writing text files.	org.apache.beam.sdk.io.TextIO
TFRecordIO	PTransforms for reading and writing TensorFlow TFRecord files.	org.apache.beam.sdk.io.TFRecordIO
XmlIO	Transforms for reading and writing XML files using JAXB mappers.	org.apache.beam.sdk.io.xml.XmlIO
TikaIO	Transforms for parsing arbitrary files using Apache Tika.	org.apache.beam.sdk.io.tika.TikaIO
ParquetIO (guide)	IO for reading from and writing to Parquet files.	org.apache.beam.sdk.io.parquet.ParquetIO
ThriftIO	PTransforms for reading and writing files containing Thrift-encoded data.	org.apache.beam.sdk.io.thrift.ThriftIO

Beam Provided I/O transforms - Java

Name	Description	Javadoc
KinesisIO	PTransforms for reading from and writing to Kinesis streams.	org.apache.beam.sdk.io.aws2.kinesis.KinesisIO (recommended) org.apache.beam.sdk.io.kinesis.KinesisIO
AmqpIO	AMQP 1.0 protocol using the Apache QPid Proton-J library	org.apache.beam.sdk.io.amqp.AmqpIO
KafkaIO	Read and Write PTransforms for Apache Kafka.	org.apache.beam.sdk.io.kafka.KafkaIO
PubSubIO	Read and Write PTransforms for Google Cloud Pub/Sub streams.	org.apache.beam.sdk.io.gcp.pubsub.PubsubIO
JmsIO	An unbounded source for JMS destinations (queues or topics).	org.apache.beam.sdk.io.jms.JmsIO
MqttIO	An unbounded source for MQTT broker.	org.apache.beam.sdk.io.mqtt.MqttIO
RabbitMqIO	A IO to publish or consume messages with a RabbitMQ broker.	org.apache.beam.sdk.io.rabbitmq.RabbitMqIO
SqsIO	An unbounded source for Amazon Simple Queue Service (SQS).	org.apache.beam.sdk.io.aws2.sqs.SqsIO (recommended) org.apache.beam.sdk.io.aws.sqs.SqsIO
SnsIO	PTransforms for writing to Amazon Simple Notification Service (SNS).	org.apache.beam.sdk.io.aws2.sns.SnsIO (recommended) org.apache.beam.sdk.io.aws.sns.SnsIO org.apache.beam.sdk.io.aws2.sns.SnsIO
Pub/Sub Lite	I/O transforms for reading from Google Pub/Sub Lite.	org.apache.beam.sdk.io.pubsublite.PubSubLiteIO

Beam Provided I/O transforms - Java

Name	Description	Javadoc
CassandraIO	An IO to read from Apache Cassandra.	org.apache.beam.sdk.io.cassandra.CassandraIO
HadoopFormatIO (guide)	Allows for reading data from any source or writing data to any sink which implements Hadoop InputFormat or OutputFormat.	org.apache.beam.sdk.io.hadoop.format.HadoopFormatIO
HBaseIO	A bounded source and sink for HBase.	org.apache.beam.sdk.io.hbase.HBaseIO
HCatalogIO (guide)	HCatalog source supports reading of HCatRecord from a HCatalog-managed source, for example Hive.	org.apache.beam.sdk.io.hcatalog.HCatalogIO
KuduIO	A bounded source and sink for Kudu.	org.apache.beam.sdk.io.kudu
SolrIO	Transforms for reading and writing data from/to Solr.	org.apache.beam.sdk.io.solr.SolrIO
ElasticsearchIO	Transforms for reading and writing data from/to Elasticsearch.	org.apache.beam.sdk.io.elasticsearch.ElasticsearchIO
BigQueryIO (guide)	Read from and write to Google Cloud BigQuery.	org.apache.beam.sdk.io.gcp.bigquery.BigQueryIO
BigTableIO	Read from (only for Java SDK) and write to Google Cloud Bigtable.	org.apache.beam.sdk.io.gcp.bigtable.BigtableIO
DatastoreIO	Read from and write to Google Cloud Datastore.	org.apache.beam.sdk.io.gcp.datastore.DatastoreIO
SnowflakeIO (guide)	Experimental Transforms for reading from and writing to Snowflake.	org.apache.beam.sdk.io.snowflake.SnowflakeIO
SpannerIO	Experimental Transforms for reading from and writing to Google Cloud Spanner.	org.apache.beam.sdk.io.gcp.spanner.SpannerIO
JdbcIO	IO to read and write data on JDBC.	org.apache.beam.sdk.io.jdbc.JdbcIO
MongoDbIO	IO to read and write data on MongoDB.	org.apache.beam.sdk.io.mongodb.MongoDbIO
MongoDbGridFSIO	IO to read and write data on MongoDB GridFS.	org.apache.beam.sdk.io.mongodb.MongoDbGridFSIO
RedisIO	An IO to manipulate a Redis key/value database.	org.apache.beam.sdk.io.redis.RedisIO
DynamoDBIO	Read from and write to Amazon DynamoDB.	org.apache.beam.sdk.io.aws2.dynamodb.DynamoDBIO (recommended) org.apache.beam.sdk.io.aws.dynamodb.DynamoDBIO
ClickHouseIO	Transform for writing to ClickHouse.	org.apache.beam.sdk.io.clickhouse.ClickHouseIO
InfluxDB	IO to read and write from InfluxDB.	org.apache.beam.sdk.io.influxdb.InfluxDbIO
Firestore IO	FirestoreIO provides an API for reading from and writing to Google Cloud Firestore.	org.apache.beam.sdk.io.gcp.healthcare.FirestoreIO

Beam Provided I/O transforms - Python

Name	Description	pydoc
FileIO	General-purpose transforms for working with files: listing files (matching), reading and writing.	apache_beam.io.FileIO
AvroIO	PTransforms for reading from and writing to Avro files.	apache_beam.io.avroio
TextIO	PTransforms for reading and writing text files.	apache_beam.io.textio
TFRecordIO	PTransforms for reading and writing TensorFlow TFRecord files.	apache_beam.io.tfrecordio
ParquetIO (guide)	IO for reading from and writing to Parquet files.	apache_beam.io.parquetio
S3IO	A source for reading from and writing to Amazon S3.	apache_beam.io.aws.s3io
GcsIO	A source for reading from and writing to Google Cloud Storage.	apache_beam.io.gcp.gcsio

Name	Description	pydoc
KafkaIO	Read and Write PTransforms for Apache Kafka.	apache_beam.io.external.kafka
PubSubIO	Read and Write PTransforms for Google Cloud Pub/Sub streams.	apache_beam.io.gcp.pubsub apache_beam.io.external.gcp.pubsub

Beam Provided I/O transforms - Python

Name	Description	pydoc
BigQueryIO (guide)	Read from and write to Google Cloud BigQuery.	apache_beam.io.gcp.bigquery
BigTableIO	Read from (only for Java SDK) and write to Google Cloud Bigtable.	apache_beam.io.gcp.bigtableio module
DatastoreIO	Read from and write to Google Cloud Datastore.	apache_beam.io.gcp.datastore.v1new.datastoreio
SnowflakeIO (guide)	Experimental Transforms for reading from and writing to Snowflake.	apache_beam.io.snowflake
MongoDbIO	IO to read and write data on MongoDB.	apache_beam.io.mongodbio

Beam Provided I/O Transforms - Go - Experimental

Name	Description	Godoc
AvroIO	PTransforms for reading from and writing to Avro files.	github.com/apache/beam/sdks/go/pkg/beam/io/avroio
TextIO	PTransforms for reading and writing text files.	github.com/apache/beam/sdks/go/pkg/beam/io/textio

Name	Description	Godoc
GcsFileSystem	`FileSystem` implementation for Google Cloud Storage.	github.com/apache/beam/sdks/go/pkg/beam/io/filesystem/gcs
LocalFileSystem	`FileSystem` implementation for accessing files on disk.	github.com/apache/beam/sdks/go/pkg/beam/io/filesystem/local
In-memory	`FileSystem` implementation in memory; useful for testing.	github.com/apache/beam/sdks/go/pkg/beam/io/filesystem/memfs

Name	Description	Godoc
PubSubIO	Read and Write PTransforms for Google Cloud Pub/Sub streams.	github.com/apache/beam/sdks/go/pkg/beam/io/pubsubio

Name	Description	Godoc
BigQueryIO (guide)	Read from and write to Google Cloud BigQuery.	github.com/apache/beam/sdks/go/pkg/beam/io/bigqueryio
DatabaseIO	Package databaseio provides transformations and utilities to interact with a generic database / SQL API.	github.com/apache/beam/sdks/go/pkg/beam/io/databaseio

Beam Provided I/O Transforms - Go via XLang

- BigQuery
- Debezium
- JDBC
- Kafka
- Schema
- https://pkg.go.dev/github.com/apache/beam/sdks/v2/go/pkg/beam/io/xlang for sources

Updated list of Apache Beam's IO Connectors

http://s.apache.org/beam-ios

(https://beam.apache.org/documentation/io/built-in/)

Custom Sources and Sinks

Community has contributed many of the non GCP IO's

- Customers can build and ideally contribute new IO's
- Not always contributed back to Apache Beam, can be on GitHub standalone;
  - Several technology partners have followed this model;
  - They may have issues around the Apache licence
  - May have other commercial reasons to not contribute back

Inside Beam's IO Connectors

IOs are the mostly same as any other ParDo

- Reading is a PTransform<PBegin, PCollection<T>>
- Writing is a PTransform<PCollection<T>, PDone> *
- Implementation is the same process of wrapping whatever API the IO connects to into DoFns and ParDos.

```
class Input<T> extends PTransform<PBegin, PCollection<T>> {

  @Override
  public PCollection<T> expand(PBegin input) {
    return input.apply( name: "Start", Create.of(SomeApi.getElements()));
  }
}
```

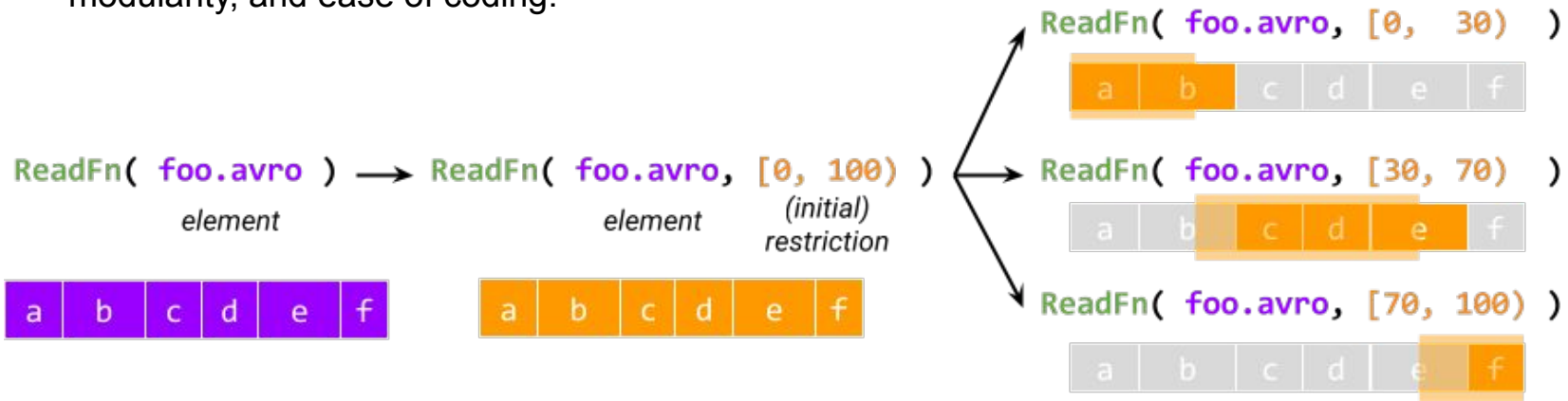
But...

- Big inputs and outputs that need to be parallelized
- Error handling for external api errors to prevent data loss
- Integration challenges
- All means they are trickier to build than most transforms.

Splittable DoFn

More Powerful IO connectors

- A generalization of DoFn that gives it the core capabilities of Source while retaining DoFn's syntax, flexibility, modularity, and ease of coding.



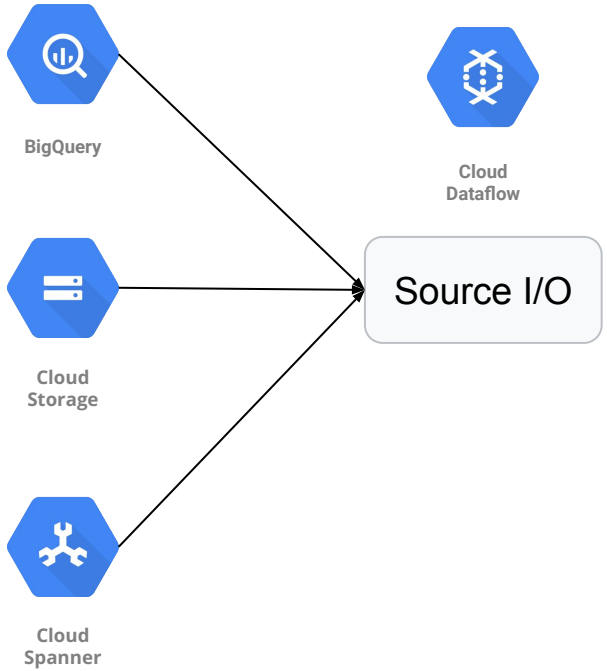
Standards for IOs

- API Standard https://s.apache.org/beam-io-api-standard
- Standards for scalability and testing are in the works

Bounded Data Sources

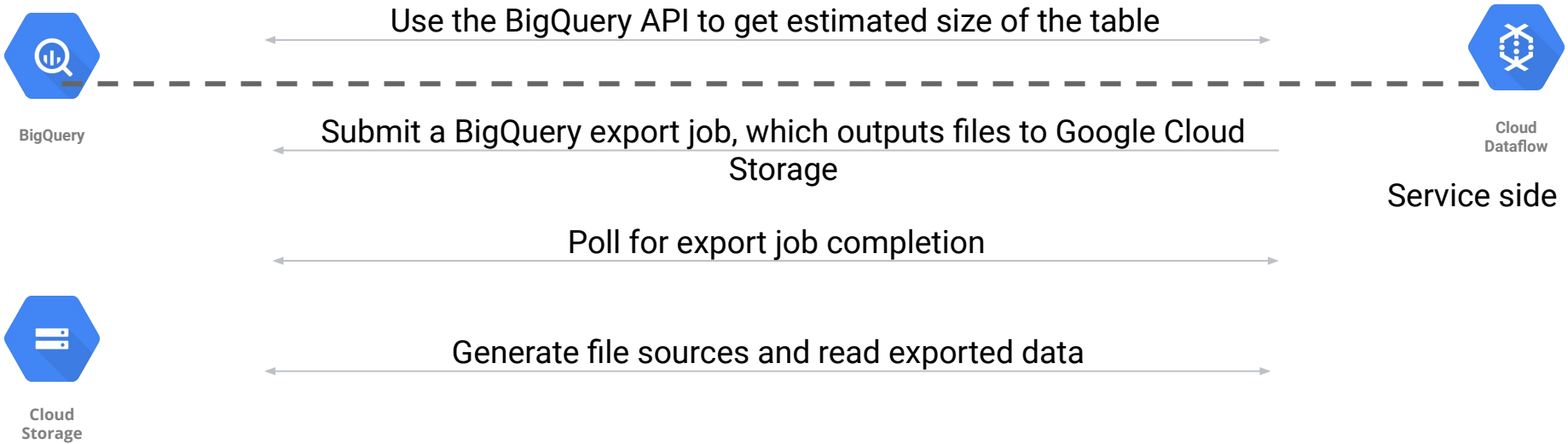
We may read a finite, ‘bounded’ amount of input. This would usually run in a batch pipeline - and in these cases, the main considerations are:

- Splitting the work of reading into smaller chunks, known as bundles.
- Provide estimates of progress to the service and number of bytes remaining to be processed.
- Track if the units of work (bundles) can be broken down into smaller chunks for dynamic work rebalancing.
- If work can be broken down, carry out the split operation.



Concrete Example - BigQueryIO Table read in Mode EXPORT

SDF initial splits: SPLIT operation:



Concrete Example - BigQueryIO Table read in Mode EXPORT

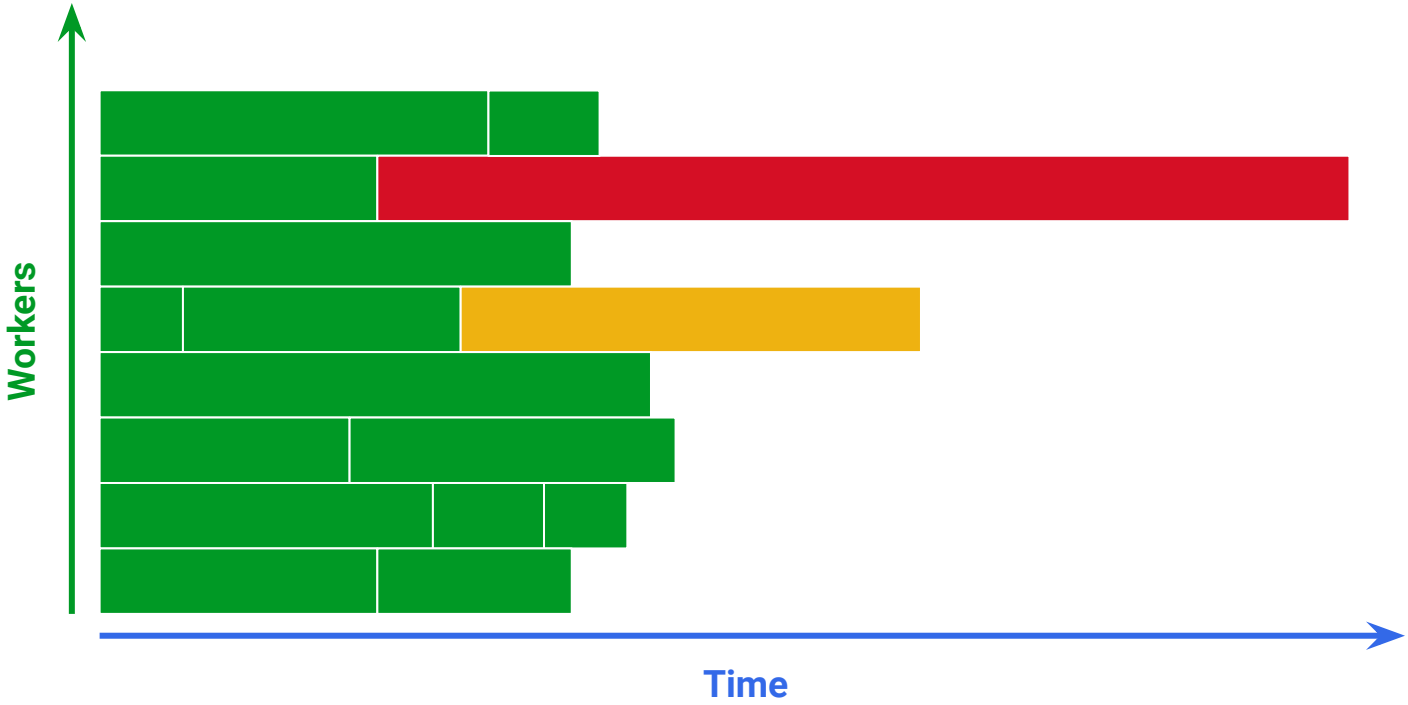
Cloud Storage

Start reading files

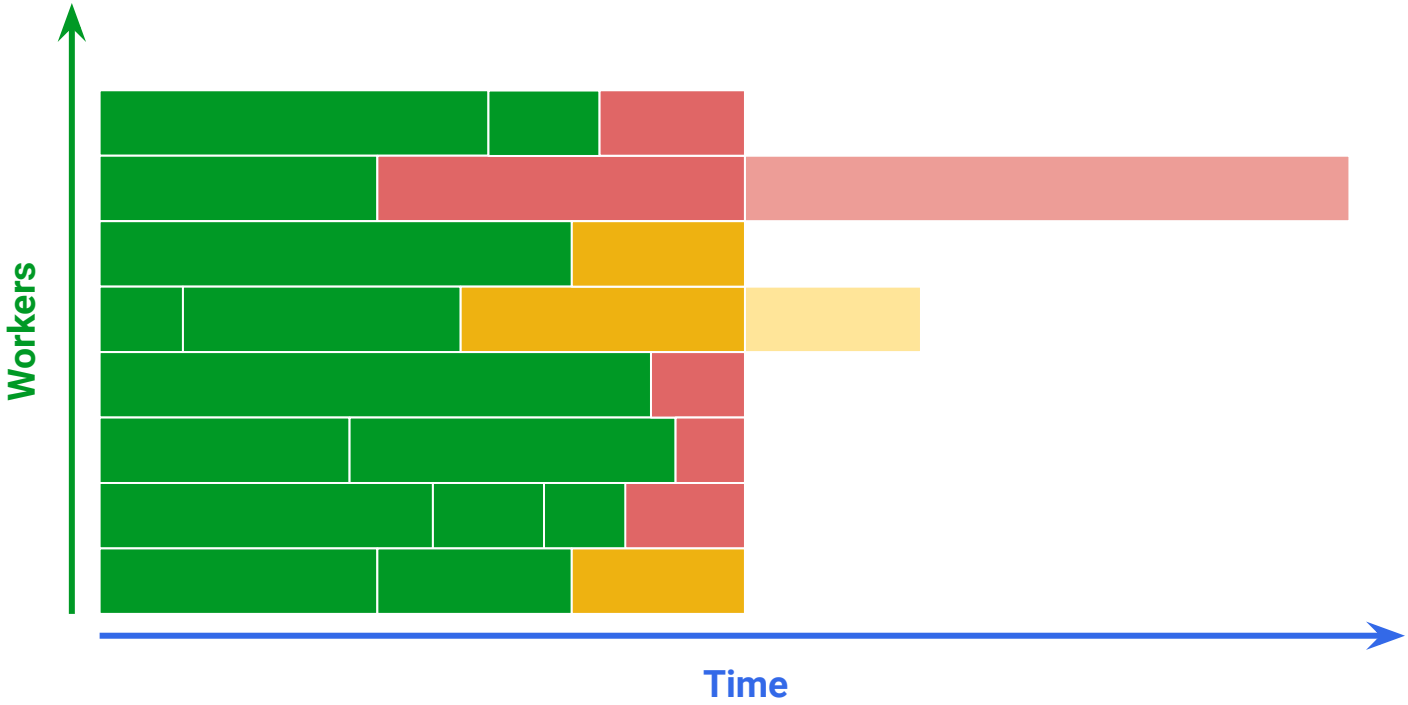
trySplit() as the output files from BigQuery have readable offset allow for **DYNAMIC SPLITTING** of work into smaller bundles

Cloud Dataflow

Dynamic Splitting? - Detecting stragglers



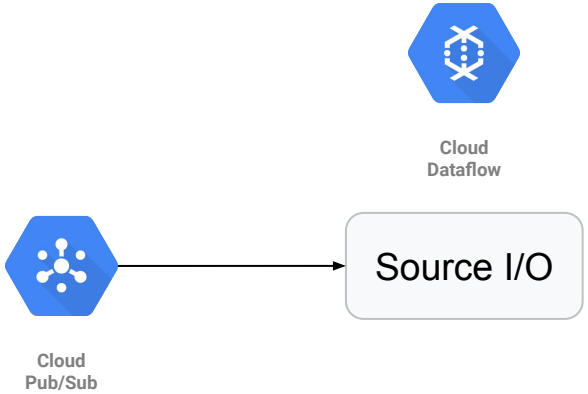
Dynamic Splitting? - Mitigating stragglers



Streaming Data Sources

We may be reading an unbounded amount of input (e.g. streaming sources). Important considerations when implementing a Beam transform that reads streaming data are:

- **Checkpointing**, allowing to save progress, recover from failures and restart without re-reading data.
- **Watermarks**, providing info to the service on what point in time the data is complete.
- **Deduping**: The option to make use of Record ids from the unbounded source -- many streaming sources do not guarantee that a given record will be sent only once. Record ids allow deduping of such elements.



Data Sinks

Sinks are often ‘normal’ PTransforms that write data to end systems.

Both Java and Python however provide an abstraction for FileBasedSinks, which we will explore in the next section.

We will also explore the existing implementation of BigQueryIO.write to gain an understand of the techniques used to solve for issues like fault-tolerance.

I/O Connectors in Detail

In this section we will take a in-depth look at several heavily used connectors .

- FileIO
  - MatchAll, continuous file processing, dynamic destinations
- BigQueryIO
  - Partition Tables (Ingestion time, Column based), Sharded Table
  - Streaming performance considerations
- KafkaIO
  - Partition Assignment & Checkpointing,
  - Advanced Kafka Configurations,
  - Event Time & Watermarks

FileIO

Useful FileIO Patterns

- Processing files as they arrive
- Using file metadata within a DoFn as elements are processed
- Writing files to different locations based on data properties at run time

Processing files as they arrive

You can continuously read files or trigger stream and processing pipelines when a file arrives.

Continuous read mode

You can use FileIO or TextIO to continuously read the source for new files.
Use the FileIO class to continuously watch a single file pattern. The following example matches a file pattern repeatedly every 30 seconds, continuously returns new matched files as an unbounded PCollection<Metadata>, and stops if no new files appear for one hour:

Processing files as they arrive

```
// This produces PCollection<MatchResult.Metadata>
p.apply(
    FileIO.match()
        .filepattern("...")
        .continuously(
            Duration.standardSeconds(30),
            afterTimeSinceNewOutput(Duration.standardHours(1))));
```

Processing files as they arrive

Some runners may retain file lists during updates, but file lists don’t persist when you restart a pipeline. You can save file lists by:

- Storing processed filenames in an external file and deduplicating the lists at the next transform
- Adding timestamps to filenames, writing a glob pattern to pull in only new files, and matching the pattern when the pipeline restarts

Processing files as they arrive

Stream processing triggered from external source

A streaming pipeline can process data from an unbounded source. For example, to trigger stream processing with Google Cloud Pub/Sub:

1. Use an external process to detect when new files arrive.
2. Send a Google Cloud Pub/Sub message with a URI to the file.
3. Access the URI from a DoFn that follows the Google Cloud Pub/Sub source.
4. Process the file.

Accessing filenames

Use the FileIO class to read filenames in a pipeline job. FileIO returns a PCollection<ReadableFile> object, and the ReadableFile instance contains the filename.

To access filenames:

1. Create a ReadableFile instance with FileIO. FileIO returns a PCollection<ReadableFile> object. The ReadableFile class contains the filename.
2. Call the readFullyAsUTF8String() method to read the file into memory and return the filename as a String object. If memory is limited, you can use utility classes like Beams FileSystems class to work directly with the file.

Accessing filenames

```
p.apply(FileIO.match().filepattern("hdfs://path/to/*.gz"))
    // The withCompression method is optional. By default, the Beam SDK detects
compression from the filename.
    .apply(FileIO.readMatches().withCompression(Compression.GZIP))
    .apply(
        ParDo.of(
            new DoFn<FileIO.ReadableFile, String>() {
              @ProcessElement
              public void process(@Element FileIO.ReadableFile file) {
                // We can now access the file and its metadata.
                LOG.info("File Metadata resourceId is {} ",
file.getMetadata().resourceId());
              }
            }));
```

Writing to dynamic file destinations

Write Dynamic

If the elements in the input collection can be partitioned into groups that should be treated differently, FileIO.Write supports different treatment per group ("destination"). It can use different file naming strategies for different groups, and can differently configure the FileIO.Sink, e.g. write different elements to Avro files in different directories with different schemas.

Writing CSV files to different directories

```
enum TransactionType {
   DEPOSIT, WITHDRAWAL, TRANSFER,
   ...
   List<String> getFieldNames();
   List<String> getAllFields(BankTransaction tx);
 }

 PCollection<BankTransaction> transactions = ...;
 transactions.apply(FileIO.<TransactionType, Transaction>writeDynamic()
     .by(Transaction::getType)
     .via(tx -> tx.getType().toFields(tx),  // Convert the data to be written to CSVSink
          type -> new CSVSink(type.getFieldNames()))
     .to(".../path/to/")
     .withNaming(type -> defaultNaming(type + "-transactions", ".csv"));
```

BigQueryIO

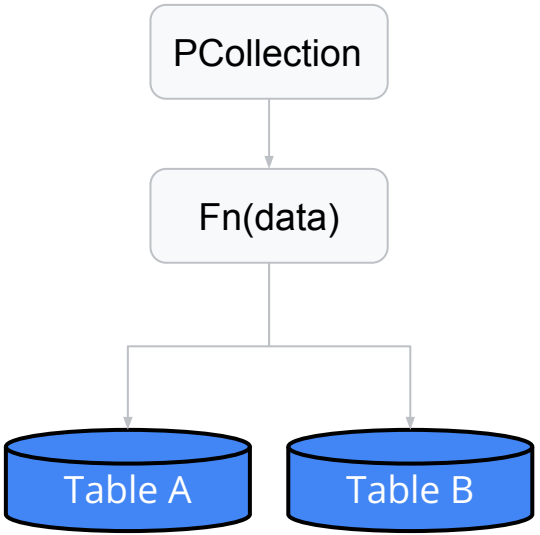
Useful BigQueryIO Patterns

- Writing data to different tables based on data properties at run time
- Writing to partitioned tables within BigQuery
- Optimization Considerations when streaming data to BigQuery

BigQuery: Dynamic Destinations

You can write to dynamic output destinations

Examine the record, and decide which table to route it to.



BigQuery: Dynamic Destinations

```
pc.apply(BigQueryIO.<Purchase>write(tableSpec)
     .useBeamSchema()
    .to((ValueInSingleWindow<Purchase> purchase) -> {
         return new TableDestination("project:dataset-" +
            purchase.getValue().getUser() + ":purchases", "");
     });
```

BigQuery writing to partition tables

1. Time Sharded Tables
2. Ingest Time Partitioned tables
3. Time Column partitioned tables

Detail BigQueryIO - Time Sharded Tables

As an alternative to partitioned tables, you can shard tables using a time-based naming approach such as `[PREFIX]_YYYYMMDD`. This is referred to as creating date-sharded tables. Using either standard SQL or legacy SQL, you can specify a query with a `UNION` operator to limit the tables scanned by the query.

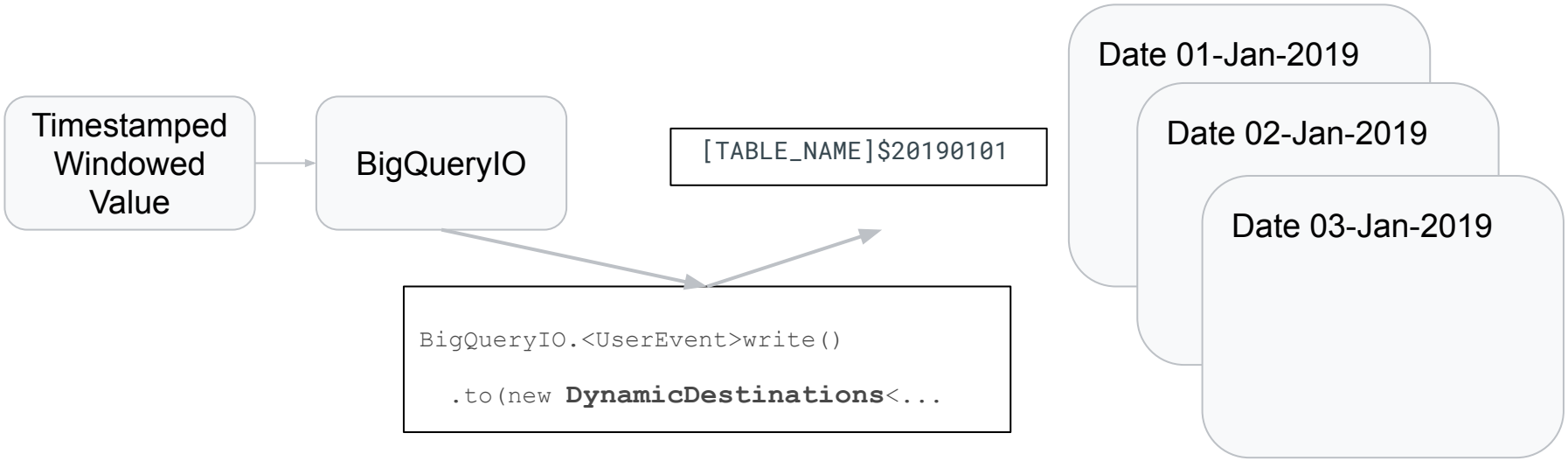
- There are performance overheads for using these types of tables, where possible use either of the previous options. If there is a need to use a sharded table, the same pattern described before, using DynamicDestinations within BigQueryIO can be used to route data to the correct table.

BigQuery - Ingest Time Partitioned Tables

When you create a table partitioned by ingestion time, BigQuery automatically loads data into daily, date-based partitions that reflect the data's ingestion or arrival date. Pseudo column and suffix identifiers allow you to restate (replace) and redirect data to partitions for a specific day.

- Therefore, the BigQueryIO defaults will result any loading to be done based on processing time.
- It is possible to load data using ingest time partitioned tables through the suffix identifiers, with the following constraints
    - Using partition suffixes, you can stream to partitions within the last 31 days in the past and 16 days in the future relative to the current date, based on current UTC time.

BigQuery - Ingest Time Partitioned Tables



BigQuery - Time Column Partitioned Tables

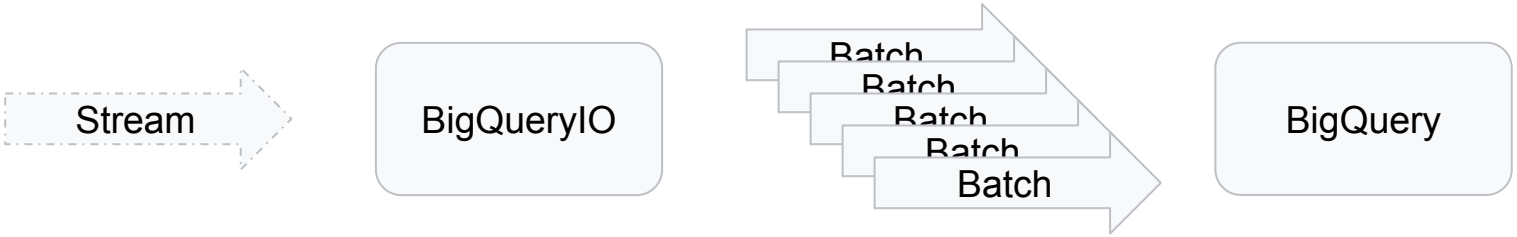
BigQuery also allows partitioned tables. Partitioned tables allow you to bind the partitioning scheme to a specific `TIMESTAMP` or `DATE` column. Data written to a partitioned table is automatically delivered to the appropriate partition based on the date value (expressed in UTC) in the partitioning column.

- A great option with no special considerations outside of the creation of timestamp column.
- Streaming limitations: You can stream data between 1 year in the past and 6 months in the future.

Detail BigQueryIO - Streaming load options

Mode 1 - Stream - > Stream

Mode 2 - Stream - > Batch



BigQuery: Streaming Writes

Two ways of writing to BigQuery in streaming:

Streaming Inserts - withMethod(STREAMING_INSERTS)

- Optimized for low latency.
- 100K QPS quota per table. Talk to BigQuery team if you need more.
- Pay for every insert
- Might sometimes duplicate data
    - If a Dataflow worker that submit rows to BQ fails, in the retry we'll be using same insert IDs for corresponding rows. In theory it's possible for users to see duplicate data even with insert IDs (BQ guarantee is time based) but most users find this to be a rare occurrence.

NOTE: New BigQuery streaming engine, which does not make use of insertId as outlined here: https://cloud.google.com/bigquery/streaming-data-into-bigquery#dataconsistency will not have these quotes.

BigQuery: File Loads

Two ways of writing to BigQuery in streaming:

Batch Loads - withMethod(BATCH_LOADS)

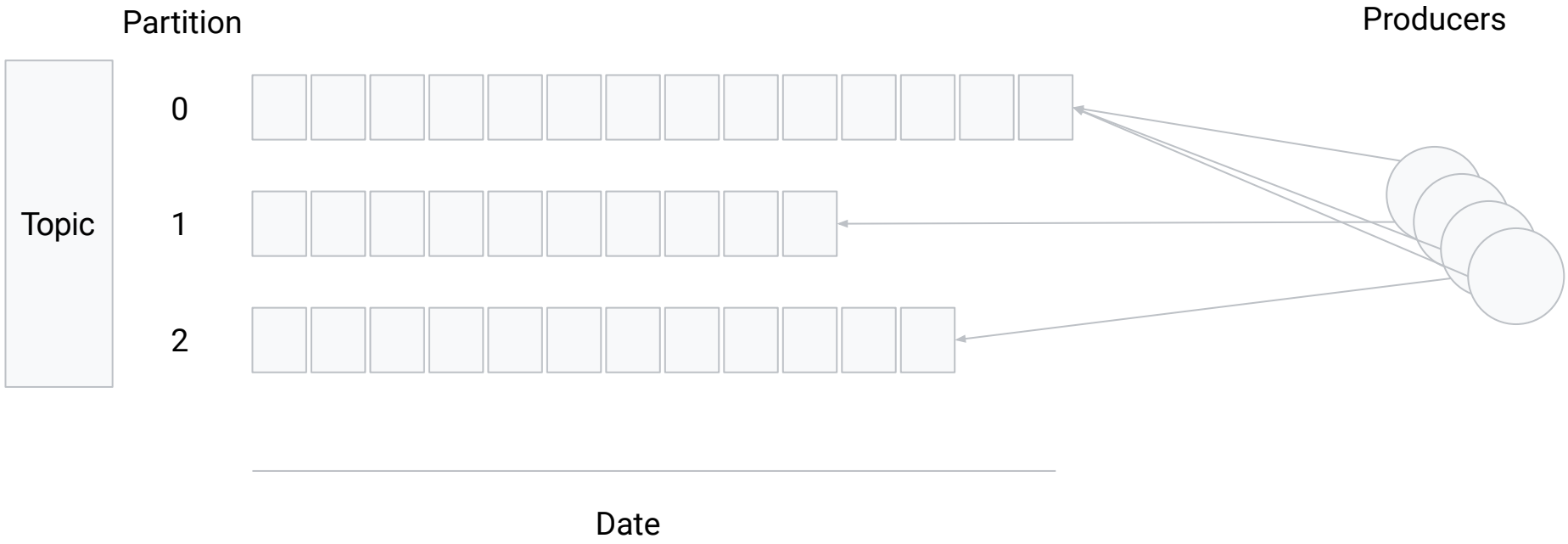
- Write output to files and periodically load into BigQuery
- BQ Batch load are Free! (Dataflow & GCS costs still incurred)
- BigQuery has daily quotas, so don’t load too often (every 10 minutes is usually safe)
- No duplicates
- Requires more tuning to get right, for example

KafkaIO

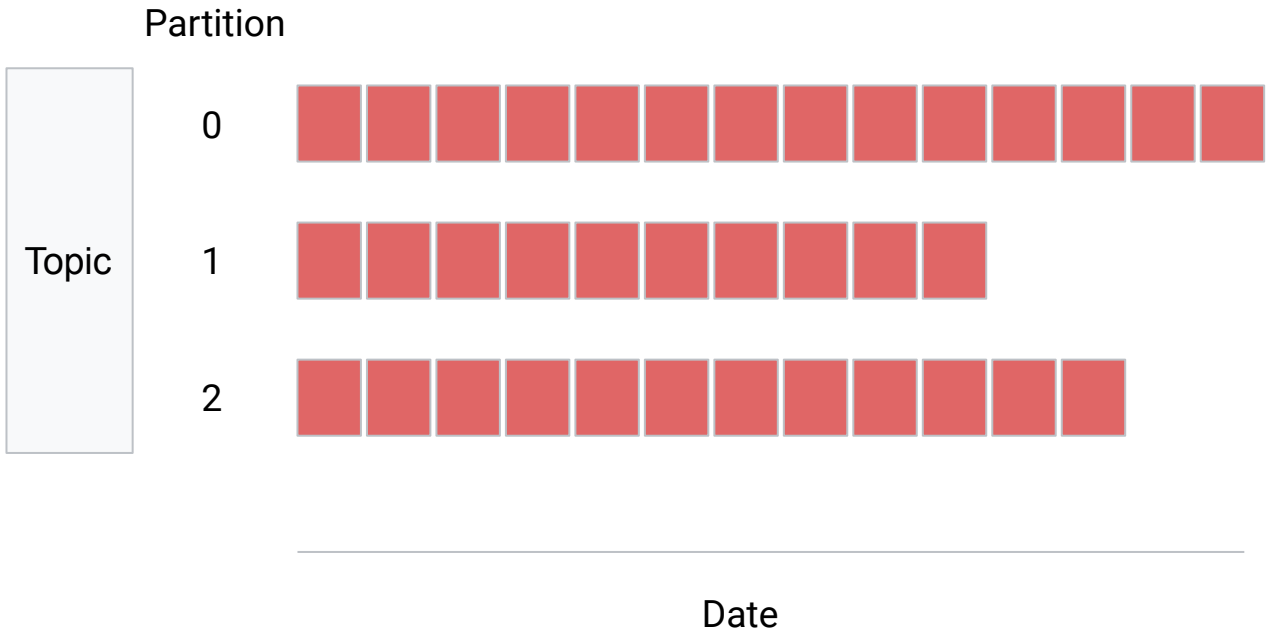
Useful KafkaIO Patterns

- Working with Kafka Topics and partitions
- Enabling advanced configurations with KafkaIO

Topics and Partitions

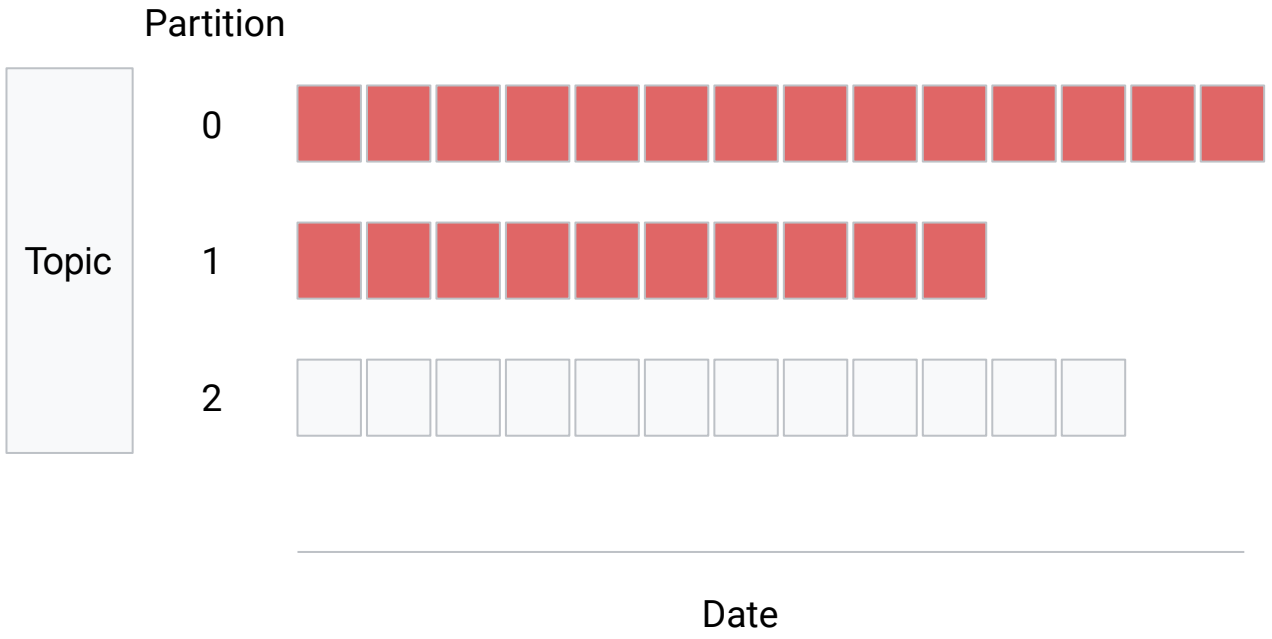


Topics and Partitions



KafkaIO
withTopics reads from one or more topics

Topics and Partitions



KafkaIO

withTopicPartitions(0,1) reads from one or more partitions within a topic

KafkaIO - Advanced Configuration

KafkaIO allows setting most of the properties in ConsumerConfig for source or in ProducerConfig for sink. E.g.

- If you would like to enable Kafka's internal offset auto commit (for external monitoring or other purposes), you can set "group.id", "enable.auto.commit", etc.
- If you would like Beam to handle the committing, you can enable commitOffsetsInFinalize instead, ensuring data is loaded into your runner before committing back to Kafka

KafkaIO - Advanced configuration

Timestamp
By default, record timestamp (event time) is set to processing time in KafkaIO reader and source watermark is current wall time.

If a topic has Kafka server-side ingestion timestamp enabled ('LogAppendTime'), it can be enabled with KafkaIO.Read.withLogAppendTime().

A custom timestamp policy can be provided by implementing TimestampPolicyFactory.

Upcoming features and announcements

Upcoming

- FileIO.watchForNewFiles option to watch for updated files, as well as new files
- Null values on Cross Language IOs - Done!
- BQ Storage API (Python Read / Java Read & Write) - Done!
- JDBC Support - Done!
- Dynamic resharding - Done!