Sources and Sinks

Wei Hsia

Customer Engineer, Google Cloud

Agenda

Course Intro

Beam Concepts Review

Windows, Watermarks, and Triggers

Sources and Sinks

Schemas

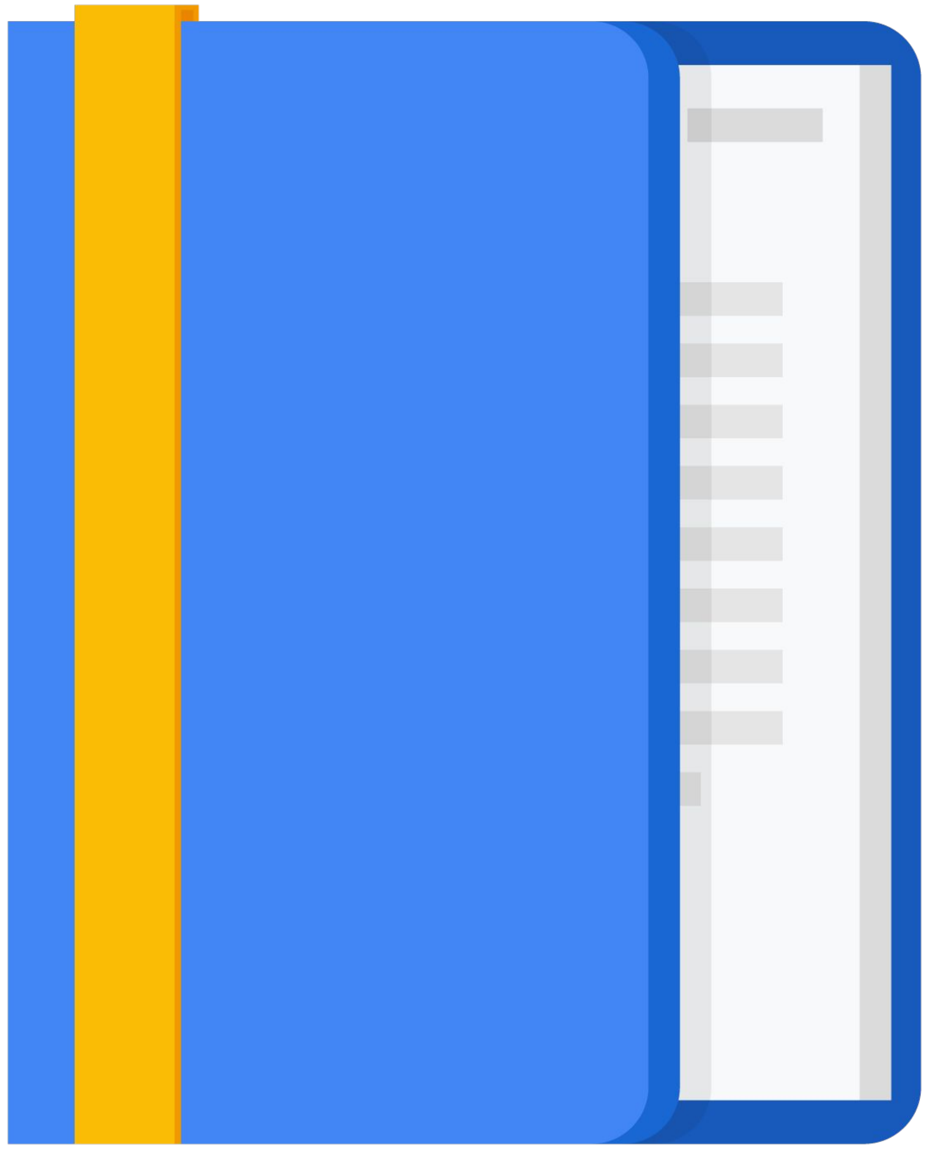
State and Timer

Best Practices

SQL and DataFrames

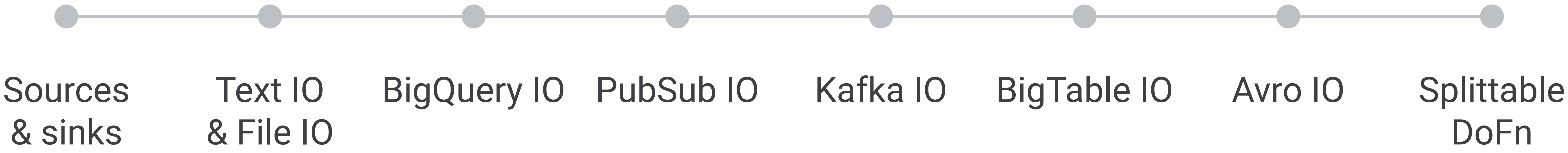
Beam Notebooks

Summary



Sources and Sinks

Agenda



Sources & sinks

Text IO & File IO

BigQuery IO

PubSub IO

Kafka IO

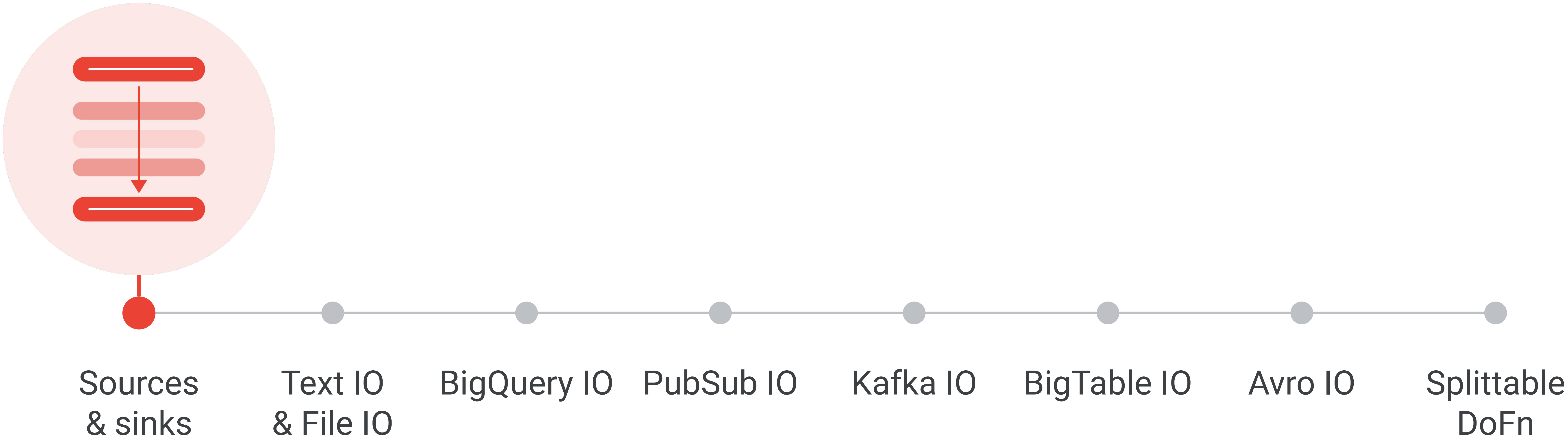
BigTable IO

Avro IO

Splittable DoFn

Sources and Sinks

Agenda



Sources & sinks

Text IO & File IO

BigQuery IO

PubSub IO

Kafka IO

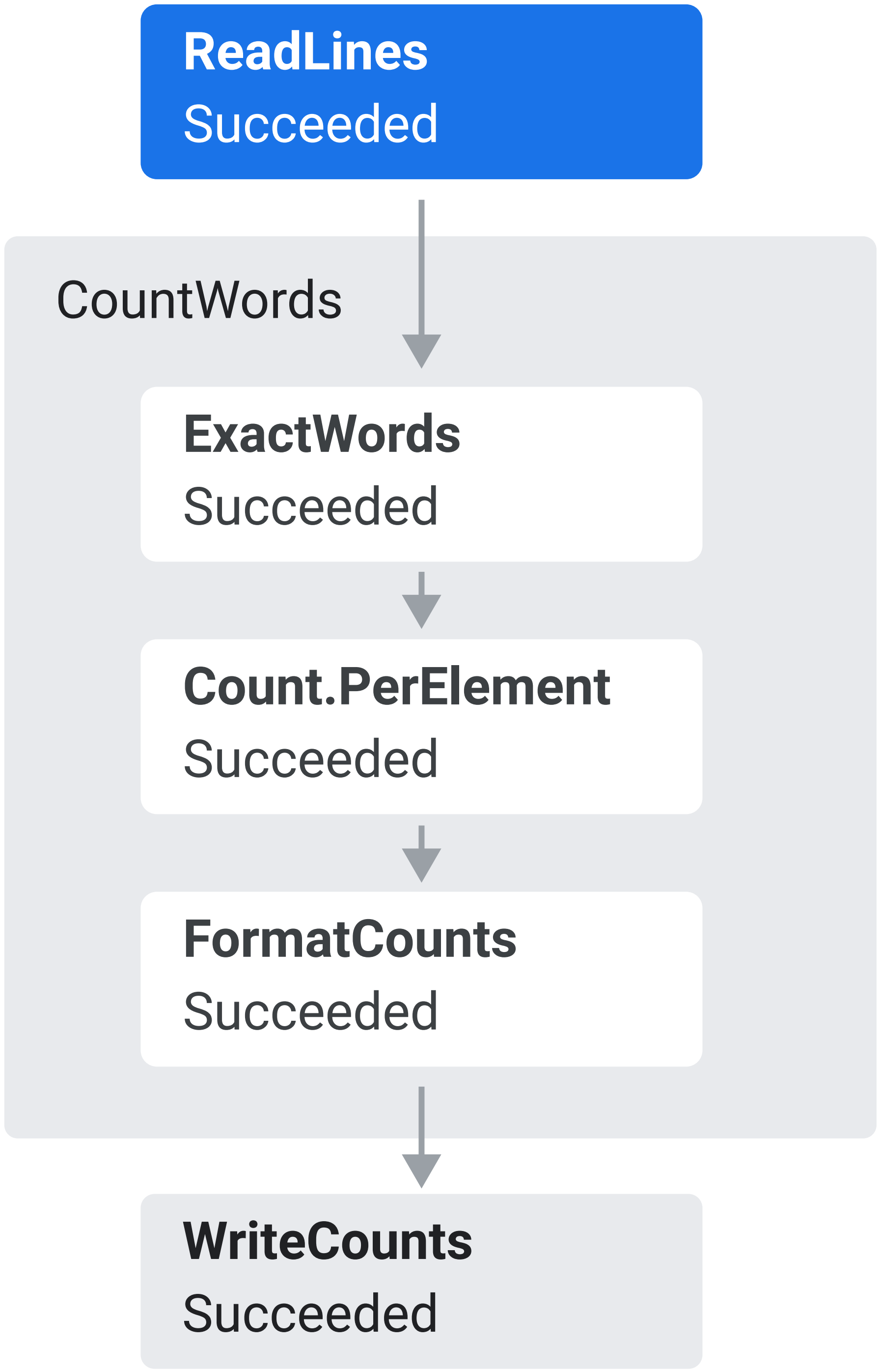
BigTable IO

Avro IO

Splittable DoFn

Reading input data

Source



ReadLines
Succeeded

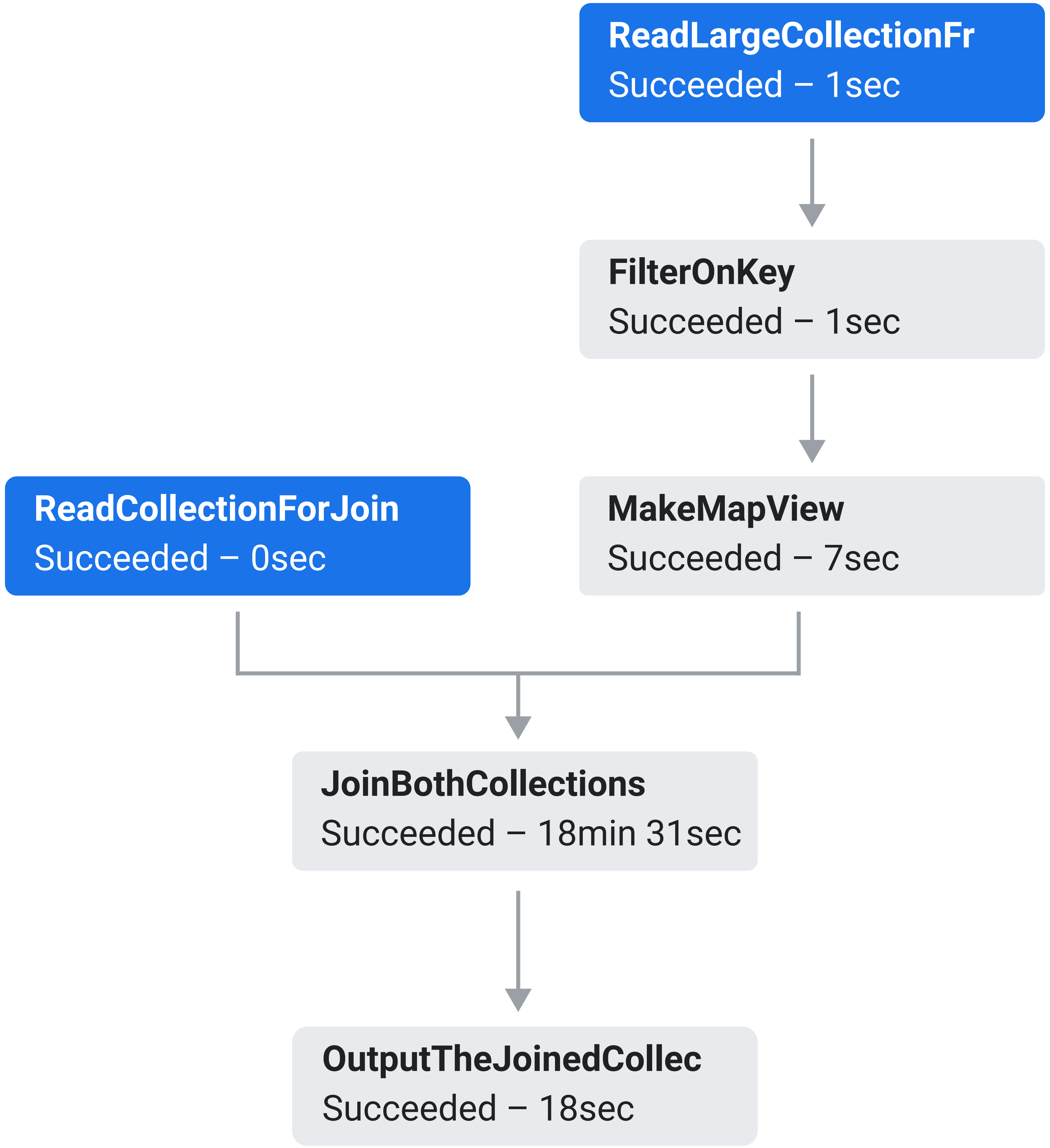
CountWords

ExactWords
Succeeded

Count.PerElement
Succeeded

FormatCounts
Succeeded

WriteCounts
Succeeded



ReadLargeCollectionFr
Succeeded – 1sec

FilterOnKey
Succeeded – 1sec

ReadCollectionForJoin
Succeeded – 0sec

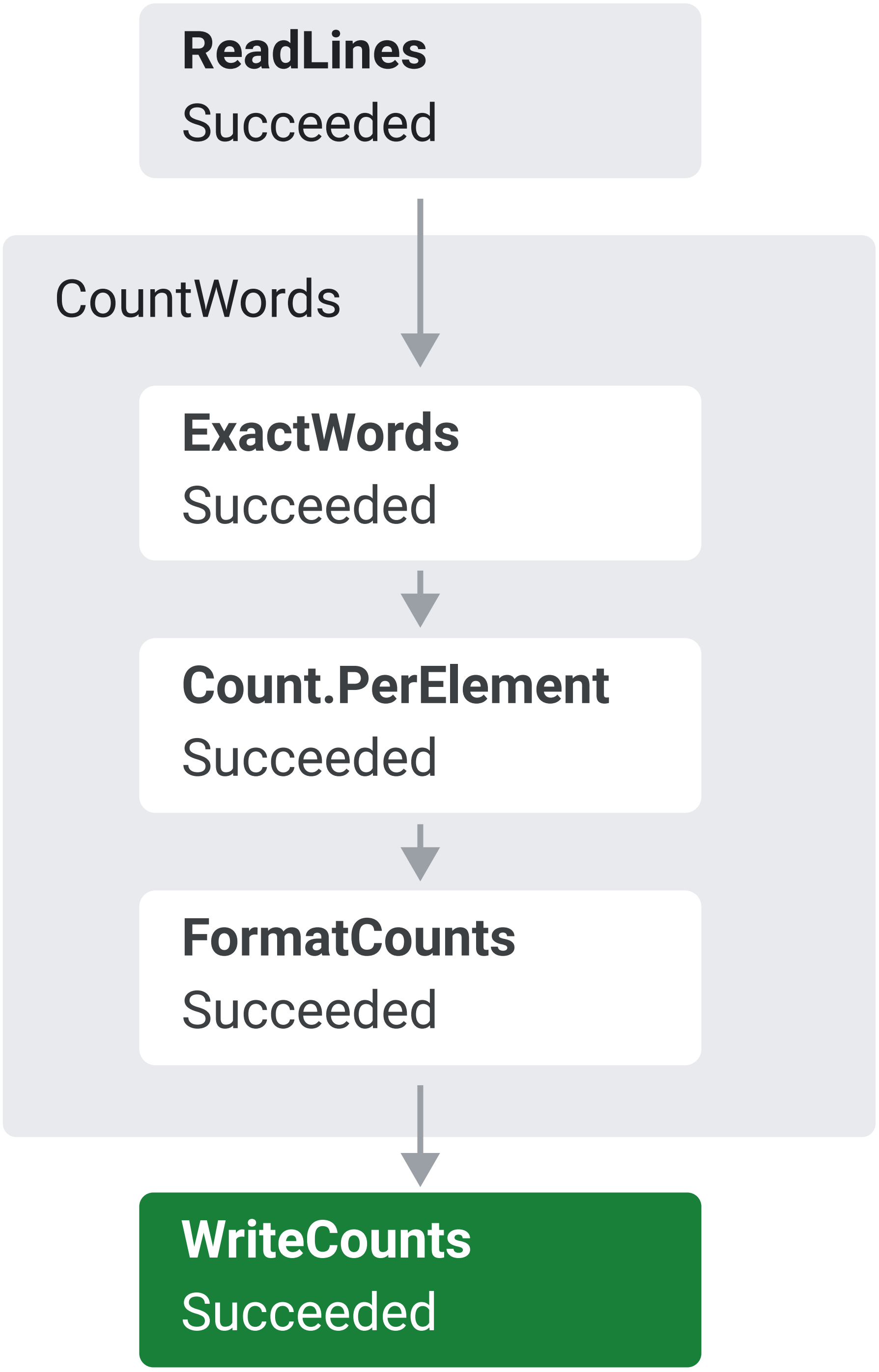
MakeMapView
Succeeded – 7sec

JoinBothCollections
Succeeded – 18min 31sec

OutputTheJoinedCollec
Succeeded – 18sec

Writing output data

Sink



ReadLines
Succeeded

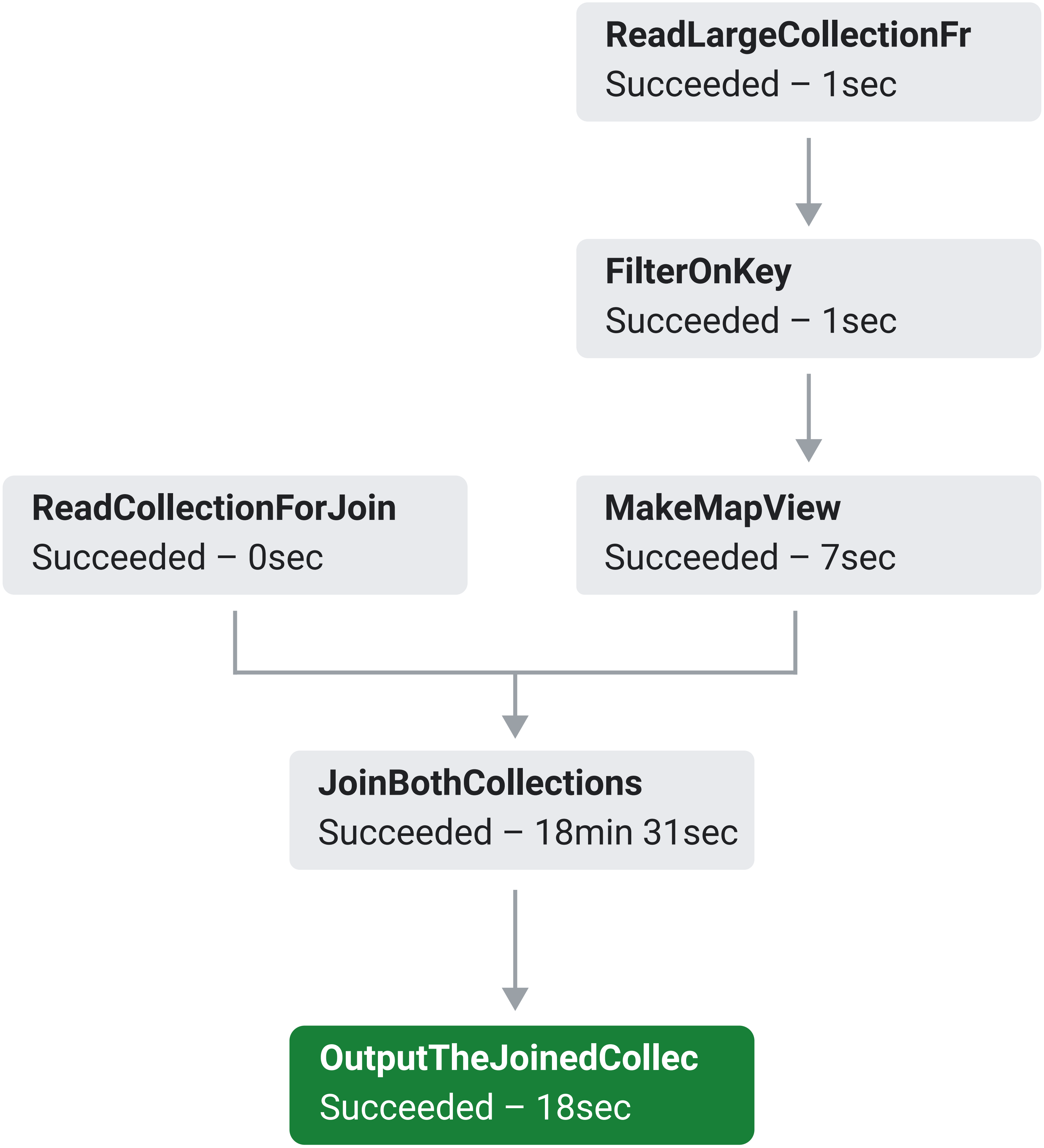
CountWords

ExactWords
Succeeded

Count.PerElement
Succeeded

FormatCounts
Succeeded

WriteCounts
Succeeded



ReadLargeCollectionFr
Succeeded – 1sec

FilterOnKey
Succeeded – 1sec

ReadCollectionForJoin
Succeeded – 0sec

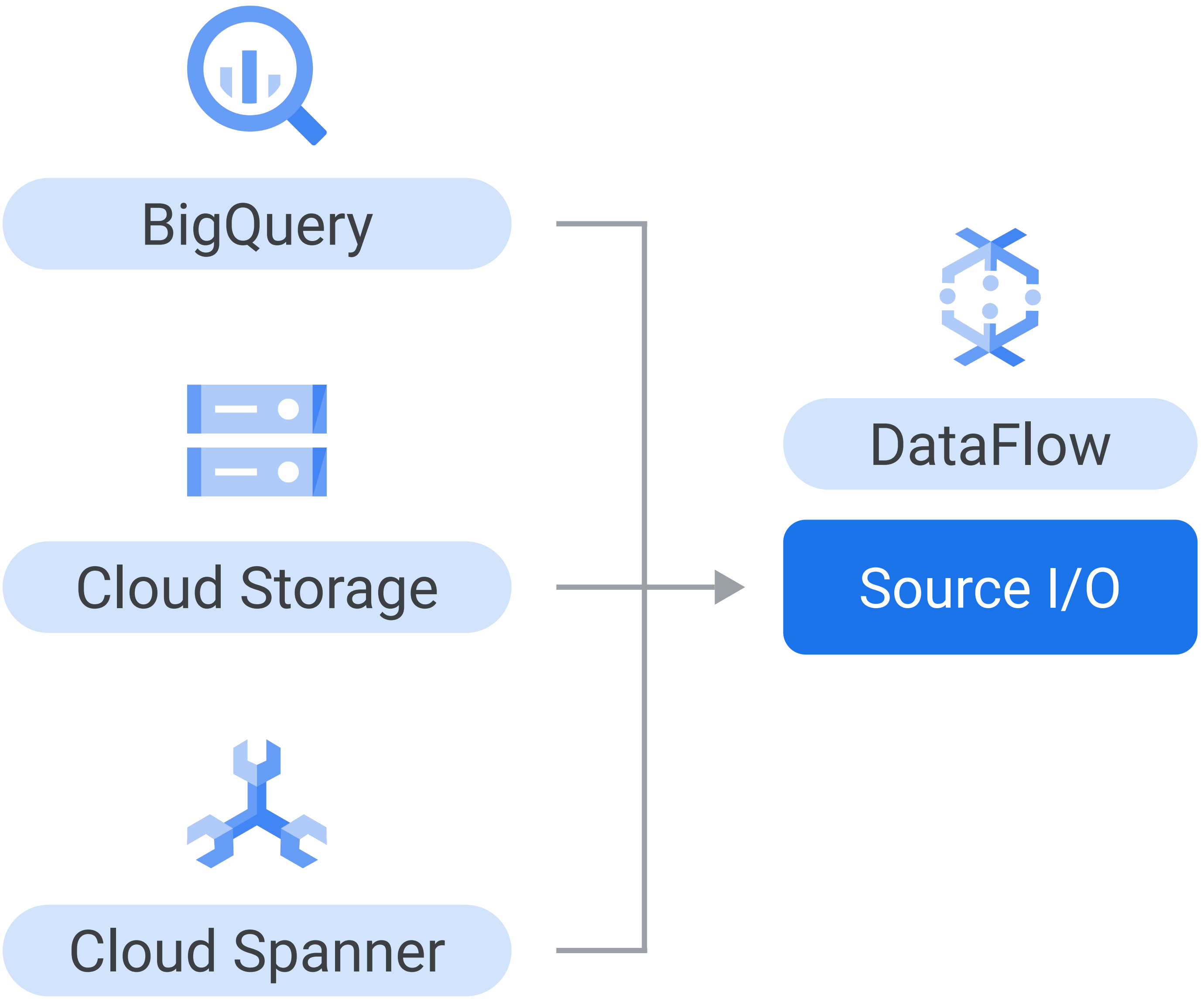
MakeMapView
Succeeded – 7sec

JoinBothCollections
Succeeded – 18min 31sec

OutputTheJoinedCollec
Succeeded – 18sec

Bounded sources

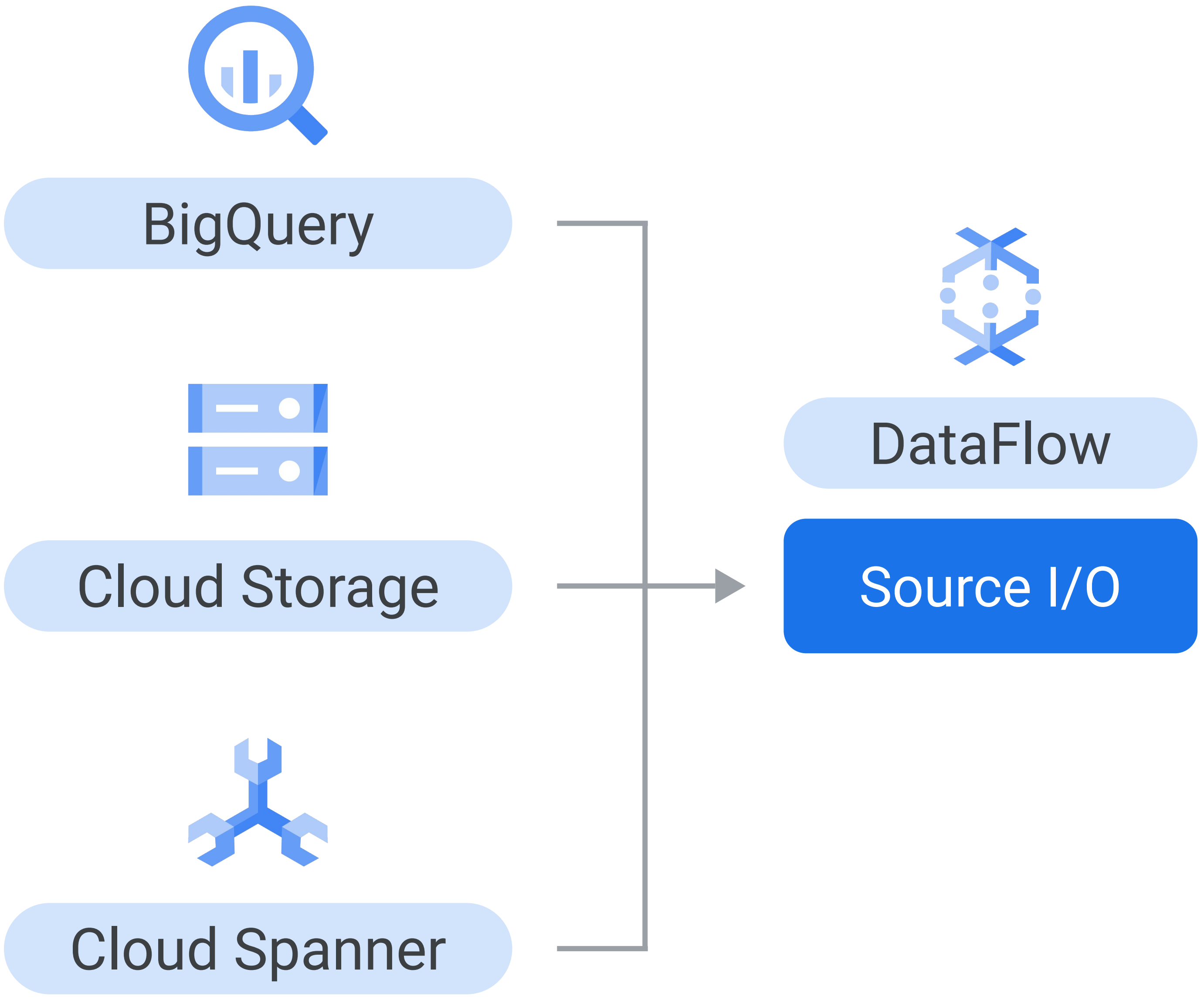
Sources that read a finite amount of input.



Bounded sources

Sources that read a finite amount of input.

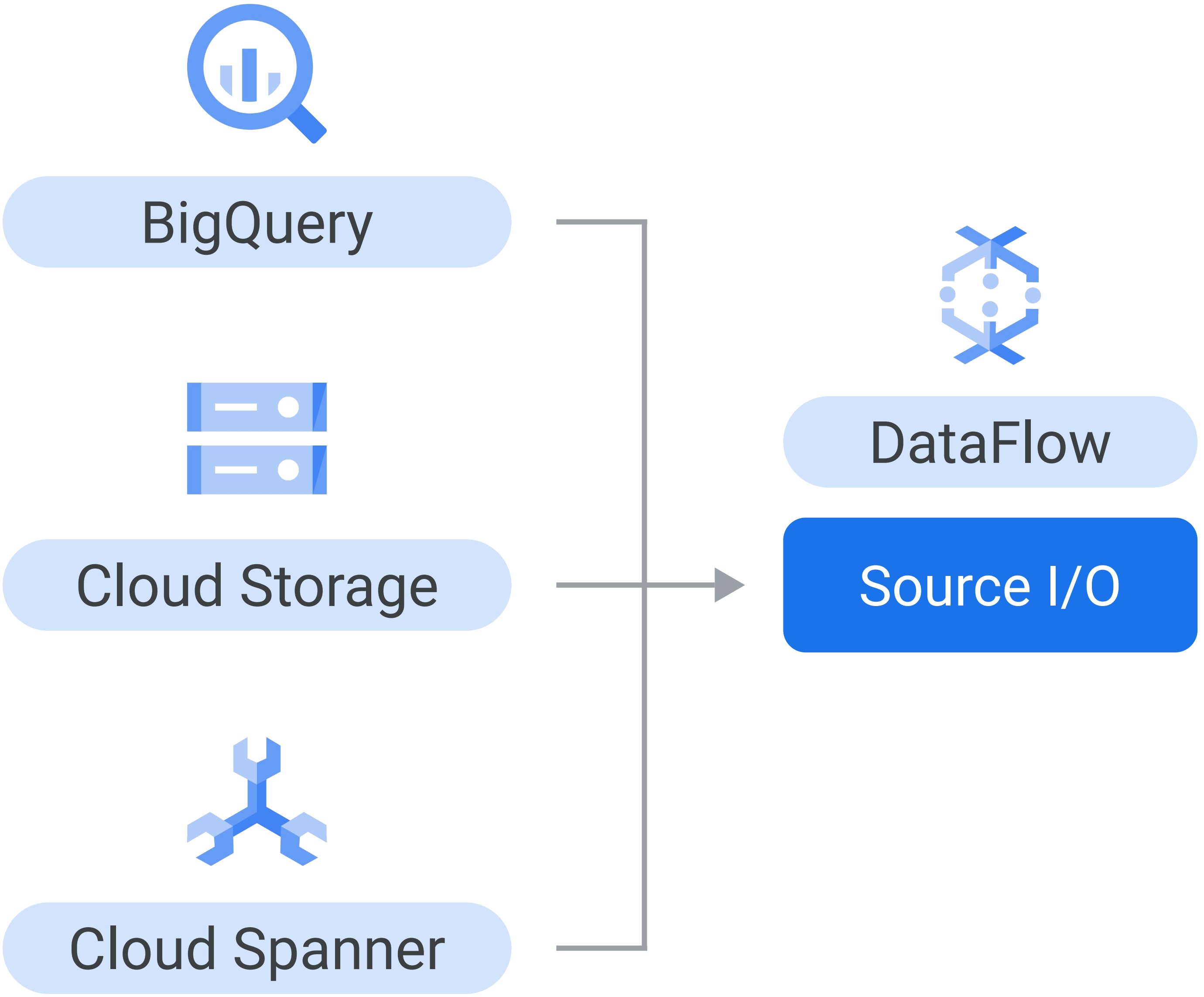
- Split the work of reading into smaller chunks, known as **bundles**.



Bounded sources

Sources that read a finite amount of input.

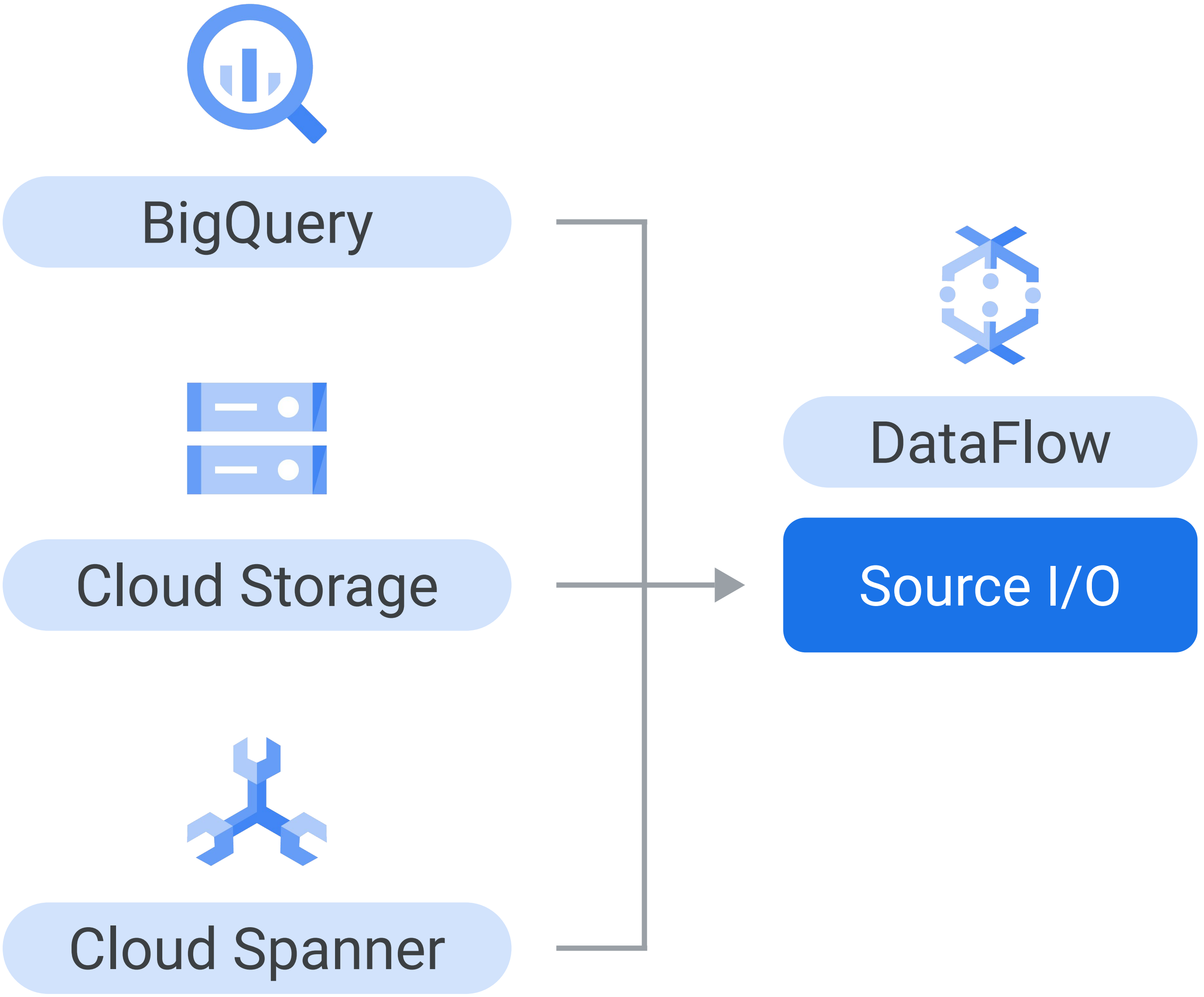
- Split the work of reading into smaller chunks, known as **bundles**.
- Provide **estimates** of progress to the service and number of bytes to be processed.



Bounded sources

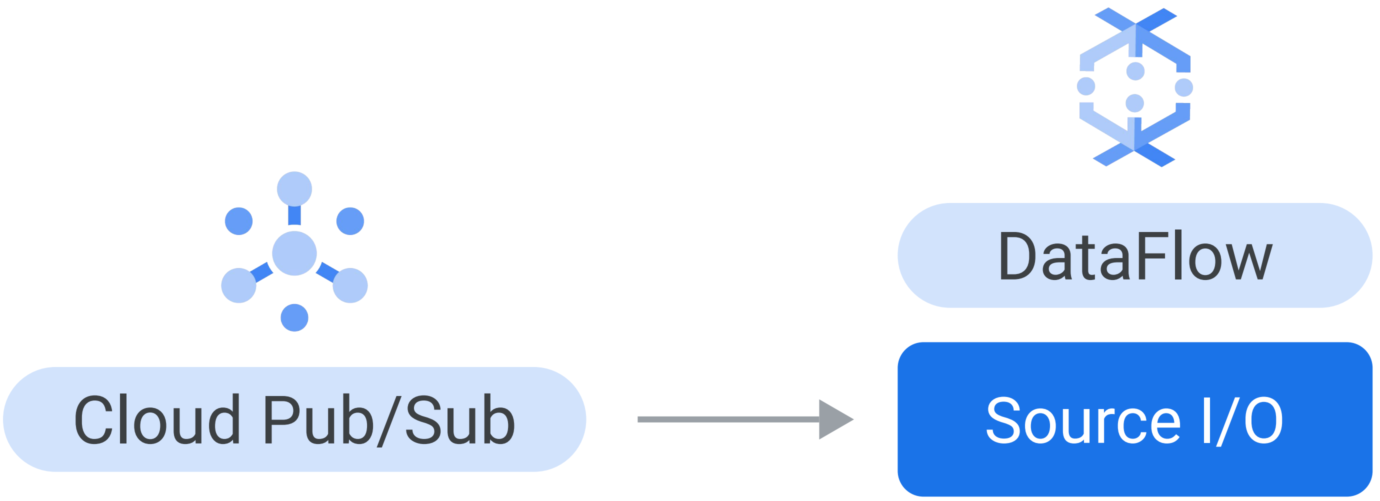
Sources that read a finite amount of input.

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



Unbounded sources

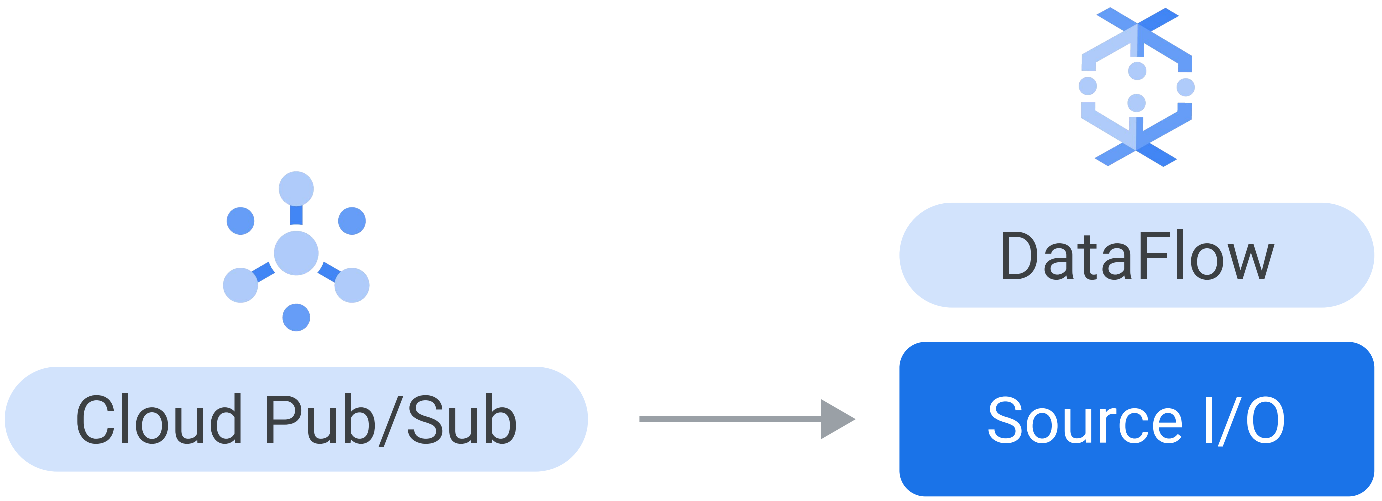
Source that reads an unbounded amount of input (e.g., streaming).



Unbounded sources

Source that reads an unbounded amount of input (e.g., streaming).

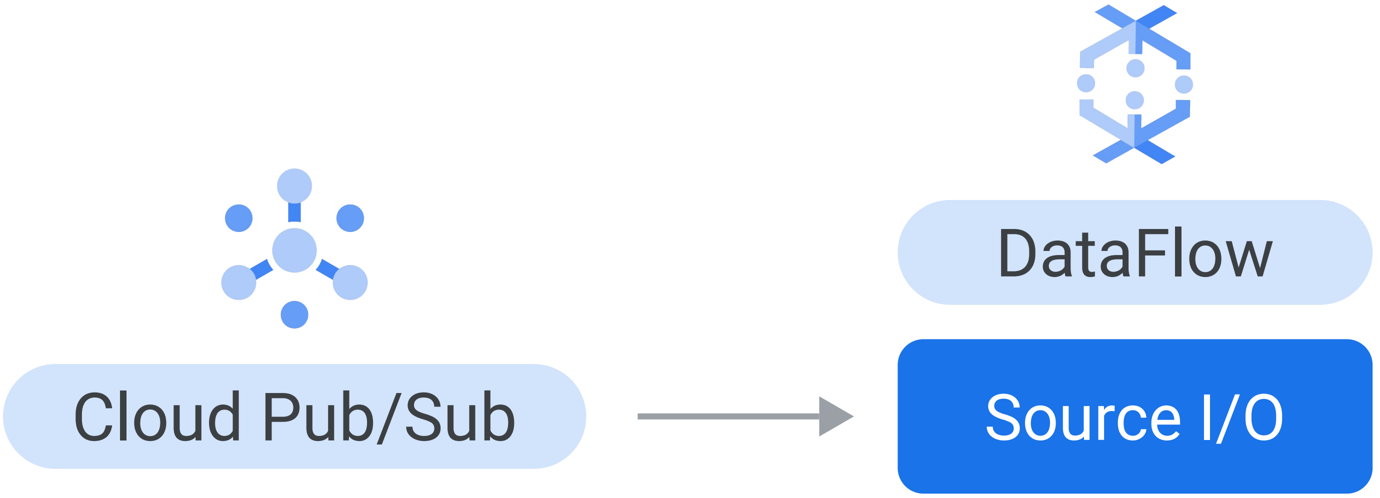
- Allowing for the source to not re-read the same data by using **checkpoints**.



Unbounded sources

Source that reads an unbounded amount of input (e.g., streaming).

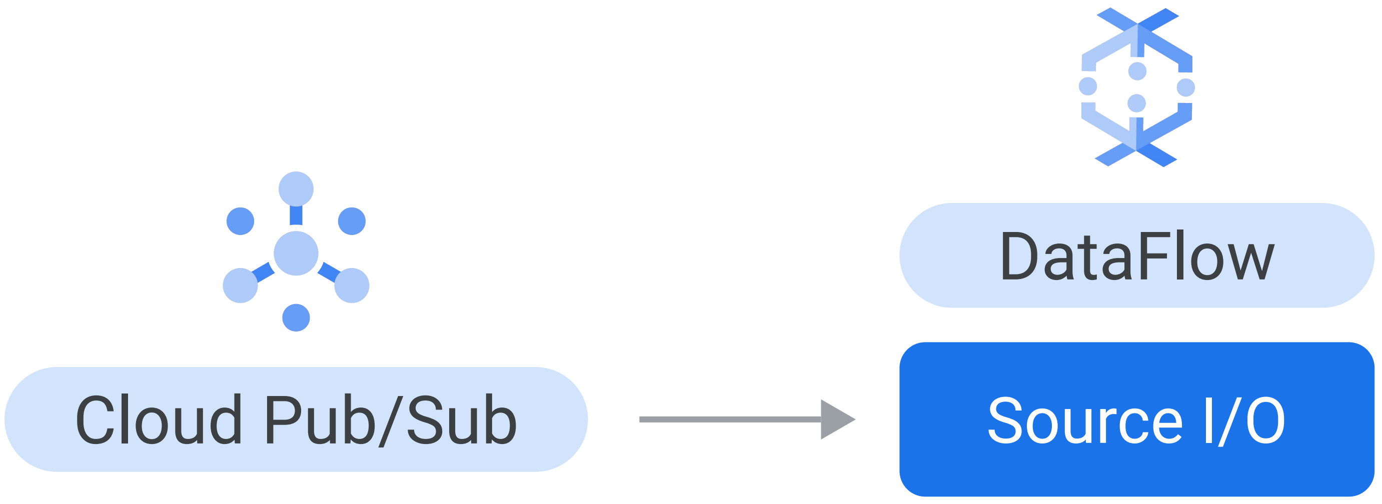
- Allowing for the source to not re-read the same data by using **checkpoints**.
- Providing data to the service on what point in time the data is complete by using **watermarks**.



Unbounded sources

Source that reads an unbounded amount of input (e.g., streaming).

- Allowing for the source to not re-read the same data by using **checkpoints**.
- Providing data to the service on what point in time the data is complete by using **watermarks**.
- Deduping the data with the option to make use of **Record IDs** from the unbounded source.



Cloud Pub/Sub

DataFlow

Source I/O

Data sinks

Sinks are often “normal” PTransforms that write data to end systems.

Java

```java
@AutoValue
  public abstract static class Write<T> extends
PTransform<PCollection<T>,WriteResult> {
```

Python

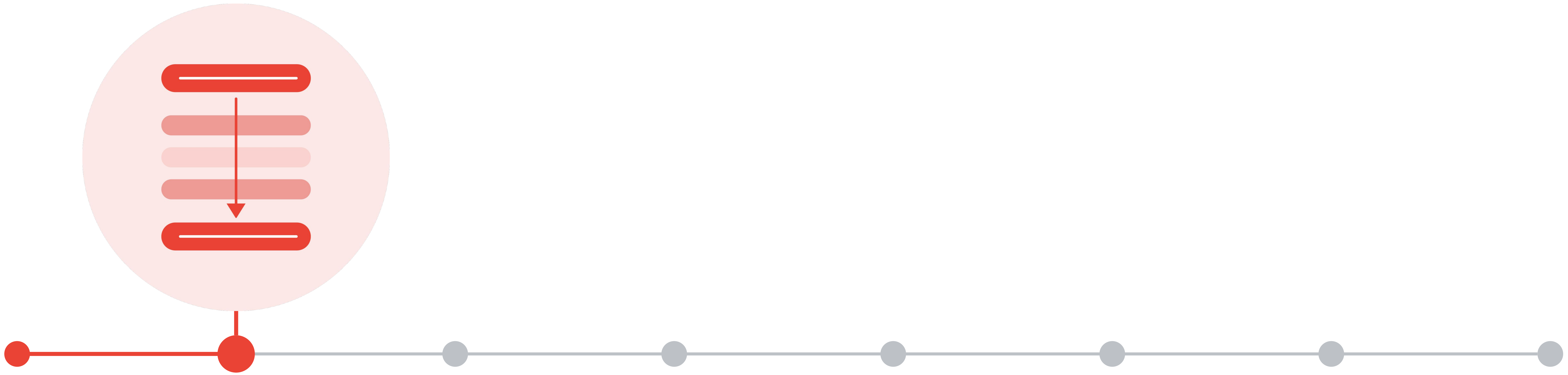
```python
class WriteToPubSub(PTransform):
```

Updated list of Apache Beam’s IO connectors

s.apache.org/beam-io

Sources and Sinks

Agenda



Sources & sinks

Text IO & File IO

BigQuery IO

PubSub IO

Kafka IO

BigTable IO

Avro IO

Splittable DoFn

Text IO reading

Java

```
Pipeline
   .apply(
     "Read from source",
      TextIO
         .read()
         .from(options
         .getInputFilePattern()))
```

Read method

Text IO reading

Python

```
pcoll = (pipeline
   | 'Create' >> Create([file_name])
   | 'ReadAll' >> ReadAllFromText())

pcoll2 = pipeline | 'Read' >> ReadFromText(file_name)
```

Read method

File IO reading with filenames

Java

```
p.apply(
  FileIO
  .match()
  .filepattern("hdfs://path/to/*.gz"))
.apply(
  FileIO
  .readMatches().withCompression(Compression.GZIP))
.apply(
  ParDo.of(
    new DoFn<FileIO.ReadableFile, String>() {
    @ProcessElement
    public void process(
      @Element FileIO.ReadableFile file) {
        LOG.info("File Metadata resourceId is {} ",
          file.getMetadata().resourceId());
      }
    }));
```

Match file pattern

Access file metadata

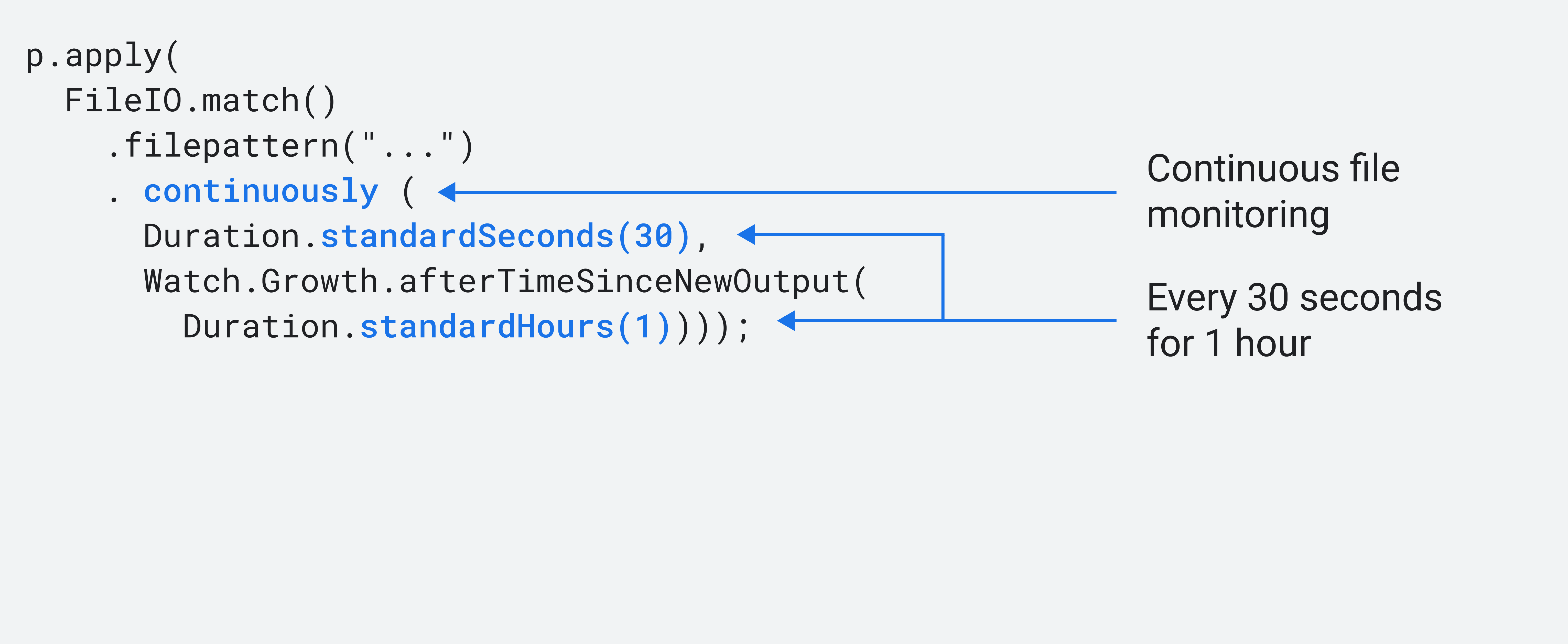
File IO reading with filenames

Python

```
with beam.Pipeline() as p:
  readable_files = (
    p
    |  fileio.MatchFiles (' hdfs://path/to/*.txt ')  ←  Match file pattern
    | fileio.ReadMatches()
    | beam.Reshuffle())
  files_and_contents = (
    readable_files
    | beam.Map(lambda x: ( x.metadata.path ,  ←——————  Access file metadata
       x.read_utf8()))
```

File IO processing files as they arrive

Java



```
p.apply(
  FileIO.match()
    .filepattern("...")
    . continuously (
      Duration.standardSeconds(30),
      Watch.Growth.afterTimeSinceNewOutput(
        Duration.standardHours(1))));
```

Continuous file monitoring

Every 30 seconds for 1 hour

File IO processing files as they arrive

Python

```
with beam.Pipeline() as p:
  readable_files = (
      p
      | beam.io.ReadFromPubSub(...)
      ... #<Parse PubSub Message and Yield
Filename>
  )

  files_and_contents = (
      readable_files
      | ReadAllFromText())
```

Trigger with message queue

Used parsed filename to read

Contextual Text IO reading

Java

```
PCollection<Row> records =
  p.apply(ContextualTextIO.read().from("..."));

PCollection<Row> records2 =
  p.apply(ContextualTextIO.read()
    .from("/local/path/to/files/*.csv")
    .withHasMultilineCSVRecords(true));

PCollection<Row> records3 =
  p.apply(ContextualTextIO.read()
    .from("/local/path/to/files/*")
    .watchForNewFiles(
      Duration.standardMinutes(1),
      afterTimeSinceNewOutput(
        Duration.standardHours(1))));
```

Text IO writing

Java

```
csv.apply(
  "Write to storage",
  TextIO
  .write()
  .to(Options
      .getTextWritePrefix())
      .withSuffix(".csv"));
```

Write method

Text IO writing

Python

```
transformed_data
| 'write' >> WriteToText(
  known_args.output, coder=JsonCoder()))
```

Write method

Text IO writing with dynamic destinations

Java

```
PCollection<BankTransaction> transactions = ...;

transactions.apply(FileIO.<TransactionType,
 Transaction>writeDynamic()
 .by(Transaction::getTypeName)
 .via(tx -> tx.getTypeName().toFields(tx),
    type -> new CSVSink(type.getFieldNames()))
 .to(".../path/to/")
 .withNaming(type -> defaultNaming(
   type + "-transactions", ".csv"));
```

Dynamic destination

Generate dynamic filename

Text IO writing with dynamic destinations

Python

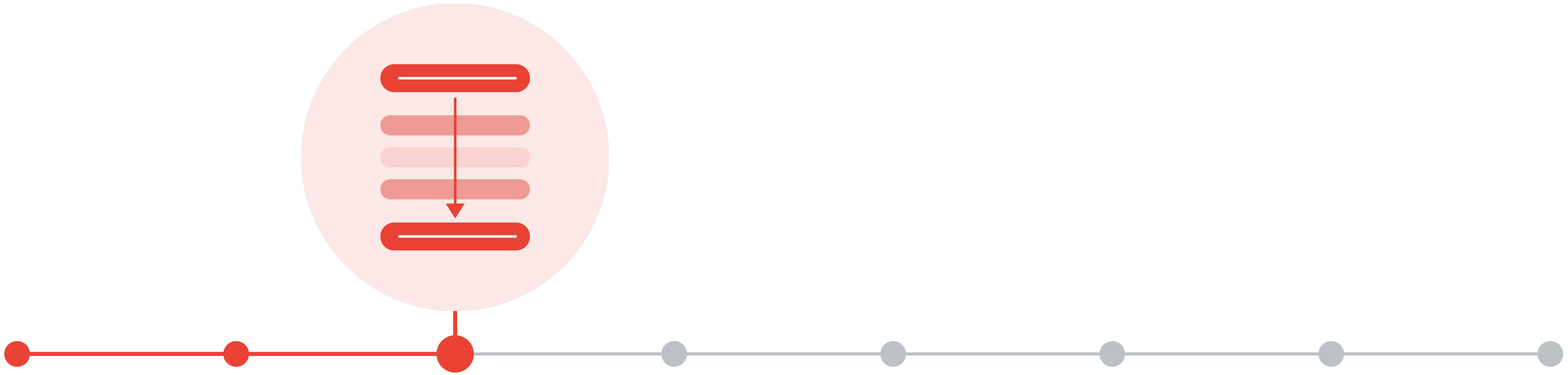
```
(my_pcollection
| beam.io.fileio.WriteToFiles(
      path='/my/file/path',
      destination=lambda record: 'avro'
        if record['type'] == 'A' else 'csv',
      sink=lambda dest: AvroSink()
        if dest == 'avro' else CsvSink(),
          file_naming=beam.io.fileio
          .destination_prefix_naming()))
```

Dynamic destination

Write to dynamic sink

Sources and Sinks

Agenda



Sources & sinks

Text IO & File IO

BigQuery IO

PubSub IO

Kafka IO

BigTable IO

Avro IO

Splittable DoFn

BigQuery IO reading with query

Java

```
PCollection<Double> maxTemperatures =
  p.apply(
    BigQueryIO.read(
      (SchemaAndRecord elem) -> (Double)
         elem.getRecord()
        .get("max_temperature"))
    .fromQuery(
      "SELECT max_temperature FROM
   `clouddataflow-readonly.samples.weather_stations`")
    .usingStandardSql()
    .withCoder(DoubleCoder.of()));
```

Map results

Source using query

BigQuery IO reading with query

Python

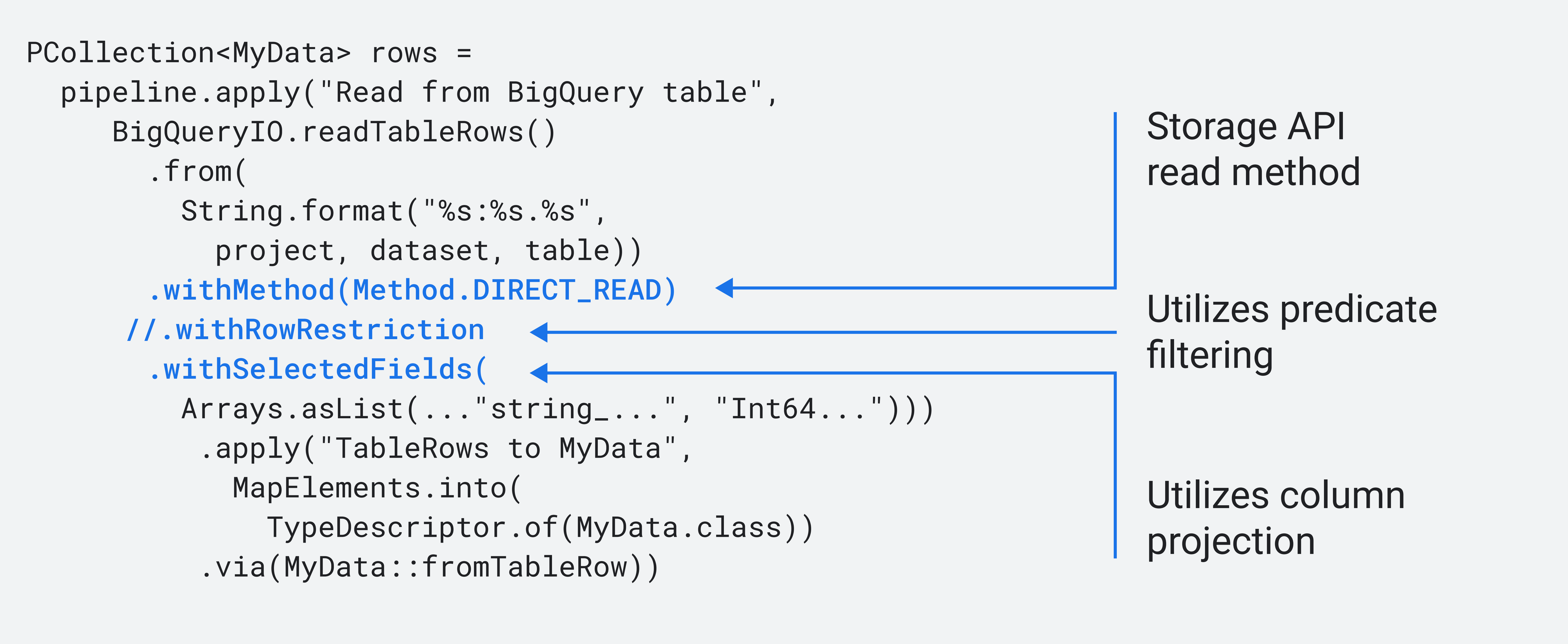
```
max_temperatures = (
    p
    | 'QueryTableStdSQL' >> beam.io.ReadFromBigQuery(
        query='SELECT max_temperature FROM '\
'`clouddataflow-readonly.samples.weather_stations`',
        use_standard_sql=True)
    | beam.Map(lambda elem: elem['max_temperature']))
```

← Map results

← Source using query

BigQuery IO reading with BigQuery Storage API

Java



```
PCollection<MyData> rows =
  pipeline.apply("Read from BigQuery table",
     BigQueryIO.readTableRows()
       .from(
         String.format("%s:%s.%s",
           project, dataset, table))
       .withMethod(Method.DIRECT_READ)
      //.withRowRestriction
       .withSelectedFields(
         Arrays.asList(..."string_...", "Int64...")))
          .apply("TableRows to MyData",
            MapElements.into(
              TypeDescriptor.of(MyData.class))
          .via(MyData::fromTableRow))
```

Storage API read method

Utilizes predicate filtering

Utilizes column projection

BigQuery IO writing with dynamic destinations

Java

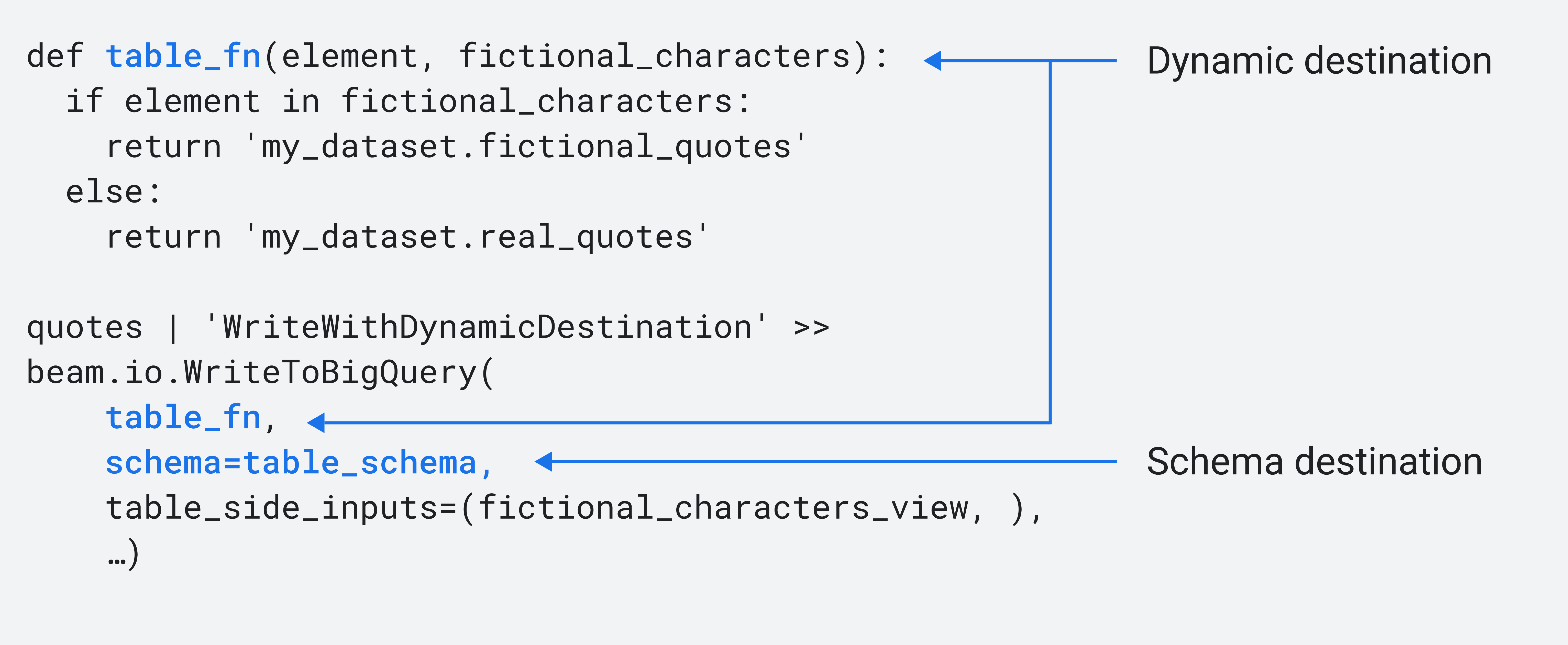
```
pc.apply(BigQueryIO.<Purchase>write(tableSpec)
   .useBeamSchema()
   .to((ValueInSingleWindow<Purchase> purchase) -> {
        return new TableDestination(
"project:dataset-" +
           purchase.getValue().getUser() +
          ":purchases", "");
    });
```

Schema definition

Dynamic destination

BigQuery IO writing with dynamic destinations

Python



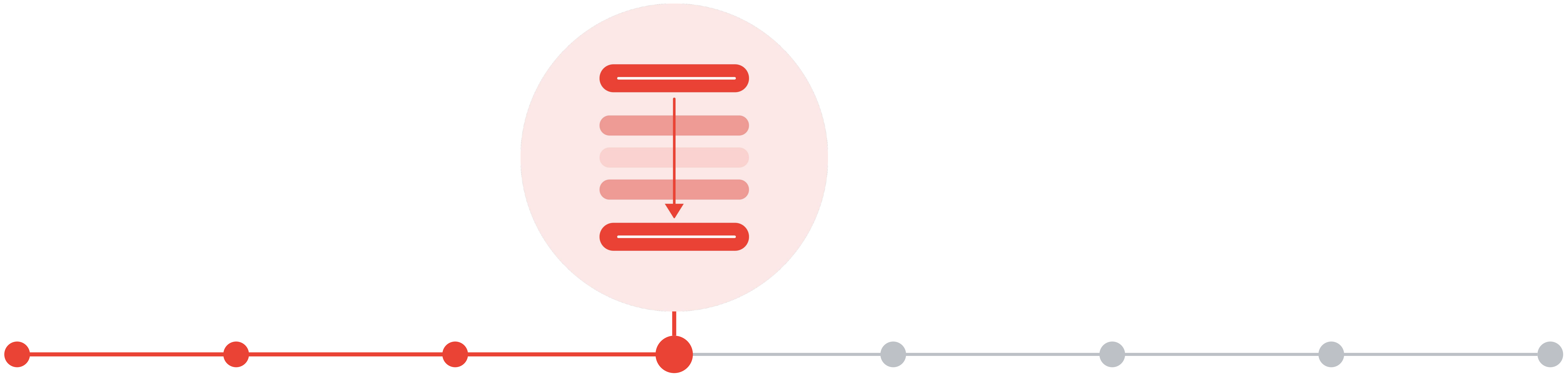
```
def table_fn(element, fictional_characters):
  if element in fictional_characters:
    return 'my_dataset.fictional_quotes'
  else:
    return 'my_dataset.real_quotes'

quotes | 'WriteWithDynamicDestination' >>
beam.io.WriteToBigQuery(
    table_fn,
    schema=table_schema,
    table_side_inputs=(fictional_characters_view, ),
    …)
```

Dynamic destination

Schema destination

Sources and Sinks

Agenda



Sources & sinks

Text IO & File IO

BigQuery IO

PubSub IO

Kafka IO

BigTable IO

Avro IO

Splittable DoFn

PubSub IO reading

Java

```
pipeline
  .apply("Read PubSub Messages",
    PubsubIO
    .readStrings()
    .fromTopic(options.getInputTopic()))
  .apply(
    Window.into(
      FixedWindows.of(
        Duration.standardMinutes(
          options.getWindowSize()))));
```

Windowing using message timestamps

PubSub IO reading

Java

```
class GroupWindowsIntoBatches(beam.PTransform):
…
        >> beam.WindowInto(
          window.FixedWindows(self.window_size))

pipeline
    | "Read PubSub Messages"
        >> beam.io.ReadFromPubSub(topic=input_topic)
    | "Window into"
        >> GroupWindowsIntoBatches(window_size)
```

Windowing using message timestamps

PubSub IO reading

Java

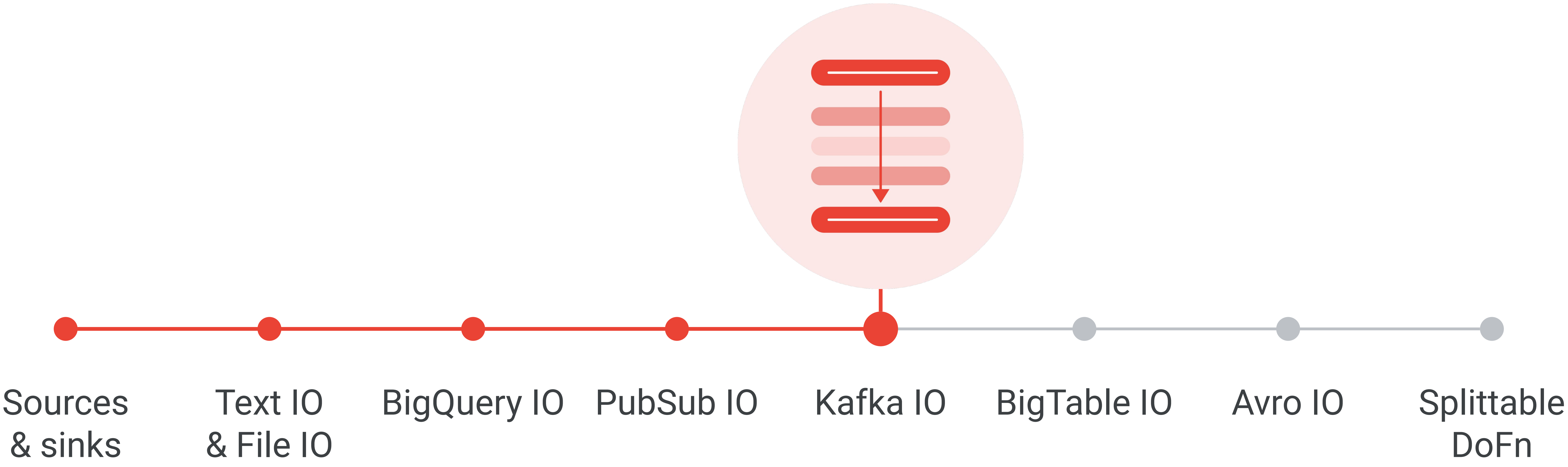
```
appliedUdf
  .get(KafkaPubsubConstants.UDF_OUT)
  .apply("getSuccessUDFOutElements",
    MapElements.into(stringTypeDescriptor).via(FailsafeElement::getPayload))
  .setCoder(NullableCoder.of(StringUtf8Coder.of()))
  .apply("writeSuccessMessages",
    PubsubIO.writeStrings().to(options.getOutputTopic()));

if (options.getOutputDeadLetterTopic() != null) {
  appliedUdf.get(KafkaPubsubConstants.UDF_DEADLETTER_OUT)
    .apply("getFailedMessages",
      MapElements.into(
        TypeDescriptors.kvs(stringTypeDescriptor, stringTypeDescriptor))
      .via(FailsafeElement::getOriginalPayload))
    .apply("extractMessageValues",
      MapElements.into(stringTypeDescriptor).via(KV<String,String>::getValue))
        .setCoder(NullableCoder.of(StringUtf8Coder.of()))
    .apply("writeFailureMessages",
      PubsubIO.writeStrings().to(options.getOutputDeadLetterTopic()));
}
```

Write message that passes UDF criteria

Dead-letter message that fails UDF criteria

Sources and Sinks

Agenda



Kafka IO reading

Java

```
PCollection<KV<String, String>> records =
pipeline
  .apply("Read From Kafka",
    KafkaIO.<String, String>read()
      .withConsumerConfigUpdates(ImmutableMap.of(
         ConsumerConfig
           .AUTO_OFFSET_RESET_CONFIG, "earliest"))
  .withBootstrapServers(options.getBootstrapServers())
            .withTopics(<...list...>)
            .withKeyDeserializerAndCoder(...))
            .withValueDeserializerAndCoder(...)
            .withoutMetadata())
```

Topic selection

Kafka IO reading

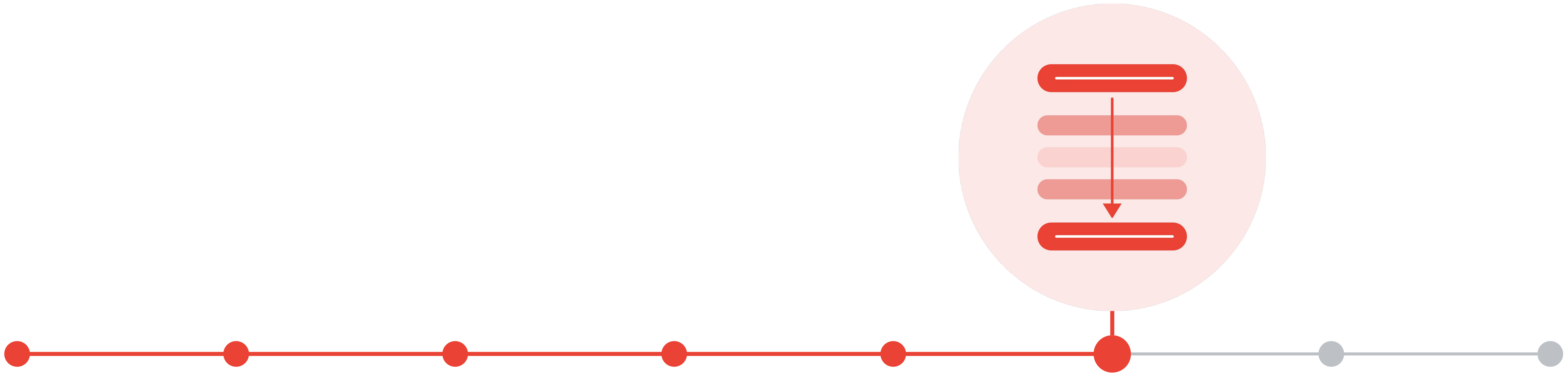
Python

```
pipeline
  | ReadFromKafka(
      consumer_config={
        'bootstrap.servers': bootstrap_servers},
        topics=[topic])
```

Cross-language transforms

Sources and Sinks

Agenda



Sources & sinks

Text IO & File IO

BigQuery IO

PubSub IO

Kafka IO

BigTable IO

Avro IO

Splittable DoFn

BigTable IO reading with row filters

Java

```
p.apply("filtered read",
     BigtableIO.read()
         .withProjectId(projectId)
         .withInstanceId(instanceId)
         .withTableId("table")
         .withRowFilter(filter));
```

Row filtered read

BigTable IO reading with prefix scan

Java

```java
ByteKeyRange keyRange = ...;
 p.apply("read",
     BigtableIO.read()
         .withProjectId(projectId)
         .withInstanceId(instanceId)
         .withTableId("table")
         .withKeyRange(keyRange));
```

Prefix scan

BigTable IO writing with additional actions

Java

```
PCollection<KV<..., Iterable<Mutation>>> data = ...;

PCollection<BigtableWriteResult> writeResults =
  data.apply("write",BigtableIO.write()
    .withProjectId("project")
    .withInstanceId("instance")
    .withTableId("table"))
    .withWriteResults();

PCollection<...> moreData = ...;

moreData
  .apply("wait for writes", Wait.on(writeResults))
  .apply("do something", ParDo.of(...))
```

← Additional actions triggered after write completion

Sources and Sinks

Agenda

Sources & sinks

Text IO & File IO

BigQuery IO

PubSub IO

Kafka IO

BigTable IO

Avro IO

Splittable DoFn

Avro IO reading with known schema

Java

```
PCollection<AvroAutoGenClass> records =
  p.apply(AvroIO.read(AvroAutoGenClass.class)
    .from("gs:..*.avro"));
```

Read Avro schema

```
Schema schema = new Schema.Parser()
  .parse(new File("schema.avsc"));
PCollection<GenericRecord> records =
  p.apply(AvroIO.readGenericRecords(schema)
    .from("gs:...-*.avro"));
```

Read Avro schema

Avro IO reading multiple files

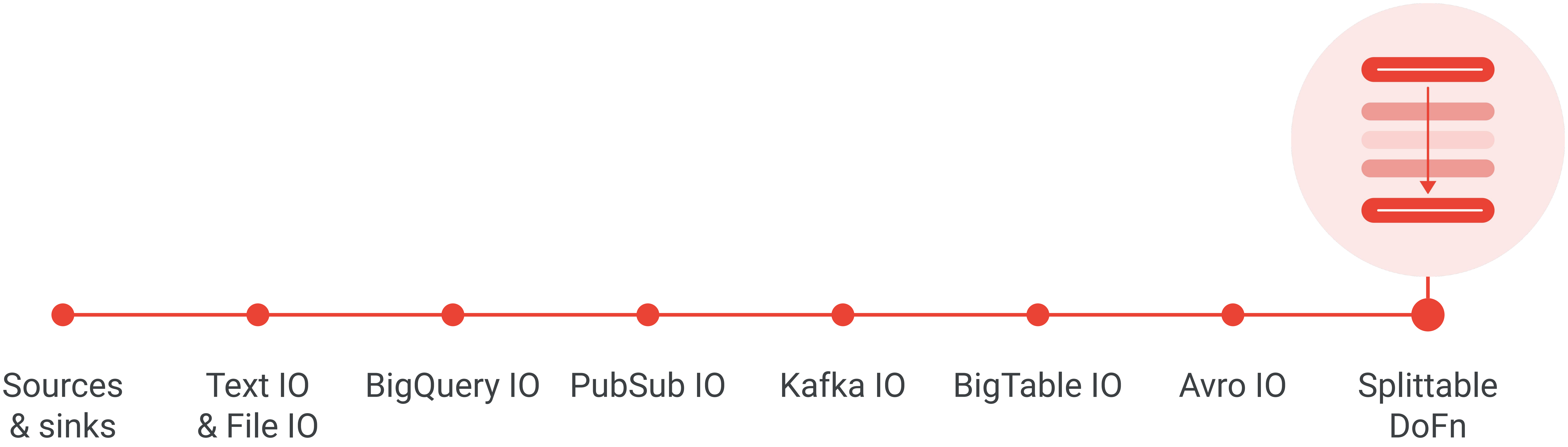
Python

```
with beam.Pipeline() as p:
  records = p | 'Read' >> beam.io.ReadFromAvro(
    '/mypath/myavrofiles*')
```

Read Avro schema

Sources and Sinks

Agenda



Sources & sinks

Text IO & File IO

BigQuery IO

PubSub IO

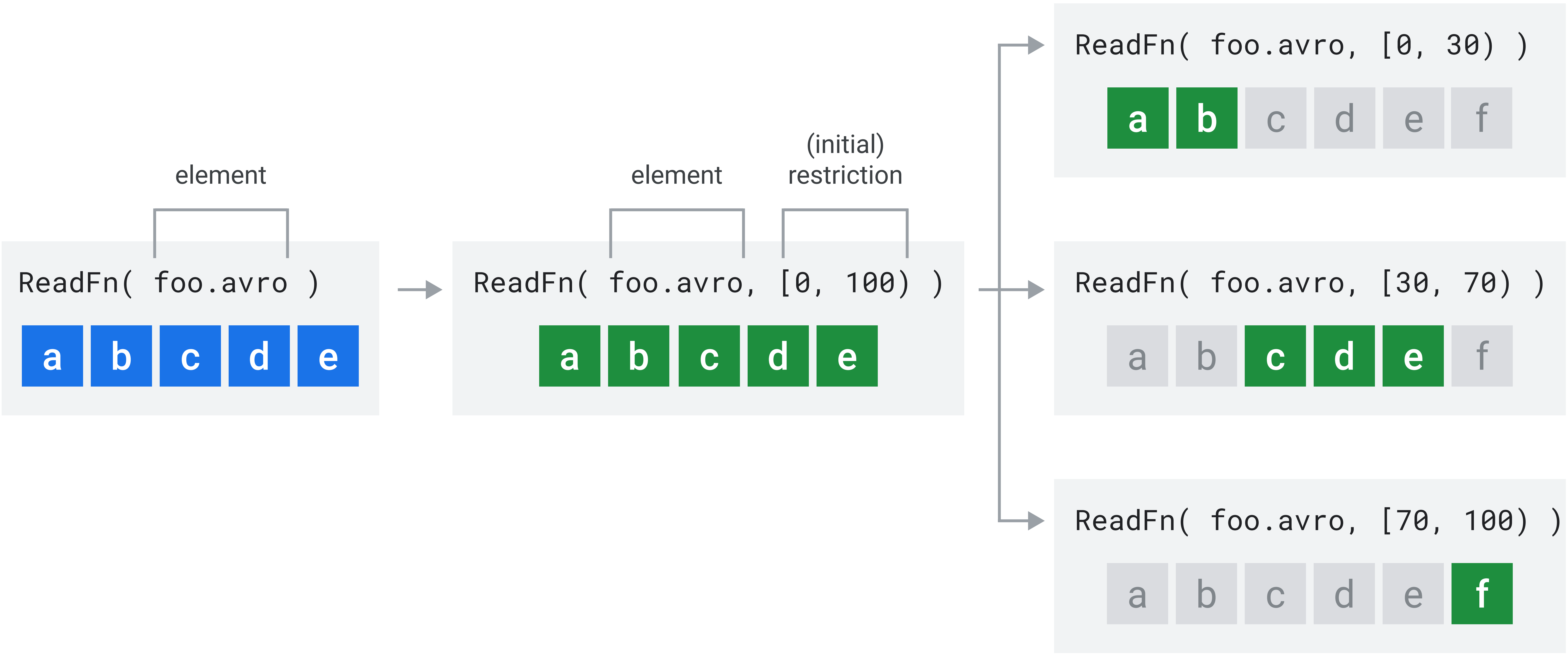
Kafka IO

BigTable IO

Avro IO

Splittable DoFn

Splittable DoFN


element
ReadFn(foo.avro)
a b c d e
element
(initial) restriction
ReadFn(foo.avro, [0, 100))
a b c d e
ReadFn(foo.avro, [0, 30))
a b c d e f
ReadFn(foo.avro, [30, 70))
a b c d e f
ReadFn(foo.avro, [70, 100))
a b c d e f

Splittable DoFn custom source

Java

```
@BoundedPerElement
private static class FileToWordsFn extends DoFn<String,
Integer> {
  @GetInitialRestriction
  public OffsetRange getInitialRestriction(
    @Element String fileName) throws IOException {
      return new OffsetRange(0,
        new File(fileName).length());
  }

  @ProcessElement
  public void processElement(
      @Element String fileName,
      RestrictionTracker<OffsetRange, Long> tracker,
      OutputReceiver<Integer> outputReceiver){...}
```

Tracking subset of restriction completed

Splittable DoFn custom source

Python

```python
class FileToWordsRestrictionProvider(
  beam.io.RestrictionProvider):
  def initial_restriction(self, file_name):
    return OffsetRange(0,
os.stat(file_name).st_size)

  def create_tracker(self, restriction):
    return beam.io.restriction_trackers
      .OffsetRestrictionTracker()

class FileToWordsFn(beam.DoFn):
  def process(...)
```

Initial restriction

Tracking subset of restriction completed

Dataflow best practices

Leverage templates

Use the existing templates or use the open source code as a basis for your code to accelerate your pipeline development.

- Python Beam examples
- Java Dataflow templates