



# SQL & DataFrames

David Sabater

Outbound Product Manager,  
Google Cloud



---

# Agenda

Course Intro

Beam Concepts Review

Windows, Watermarks, and Triggers

Sources and Sinks

Schemas

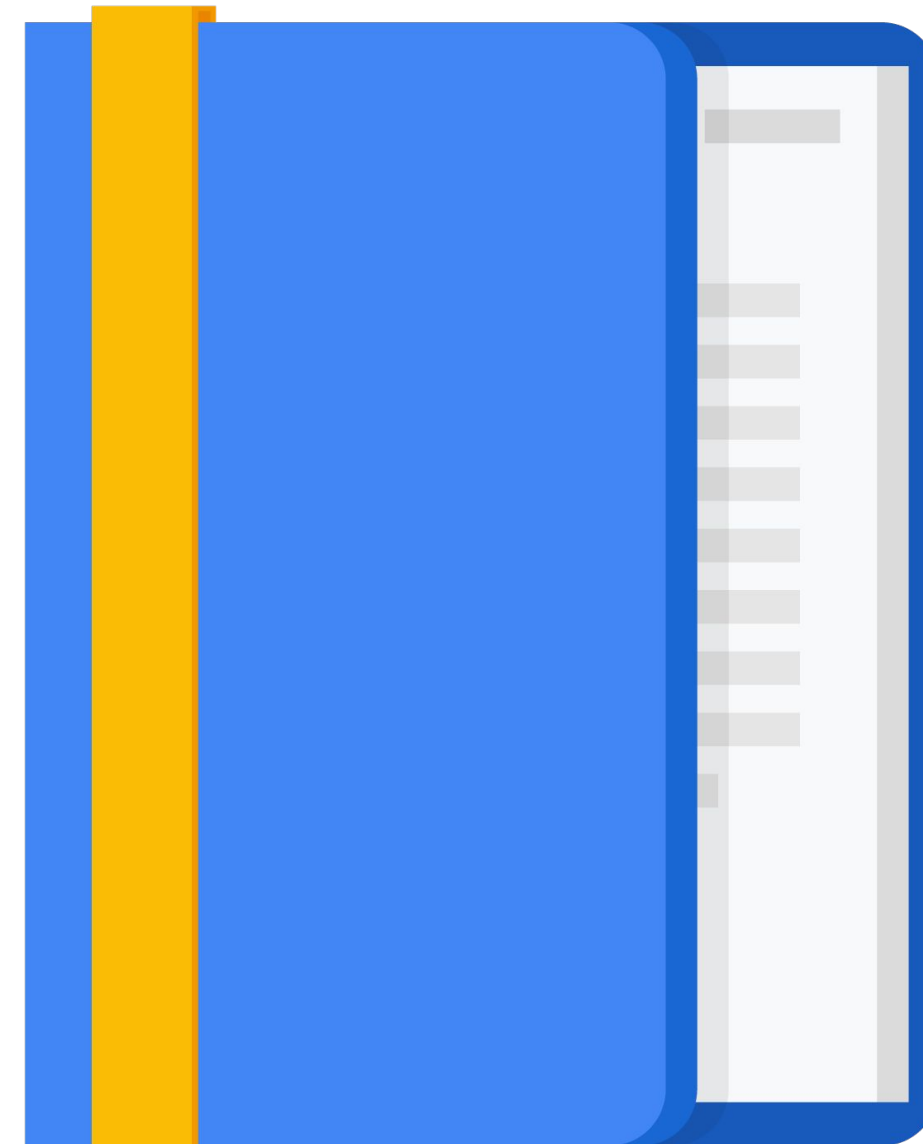
State and Timers

Best Practices

**SQL and DataFrames**

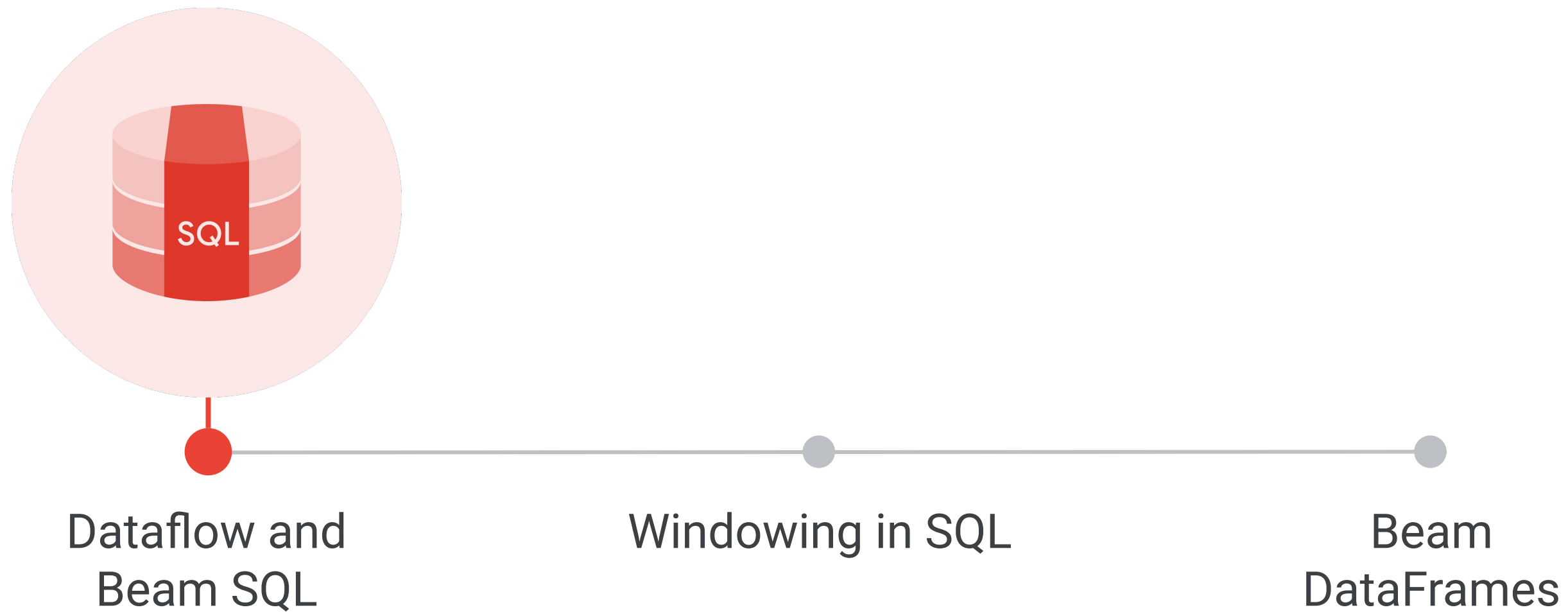
Beam Notebooks

Summary



# SQL and DataFrames

## Agenda



---

## Schemas+SQL to the rescue!

By understanding the structure of a pipeline's records,  
we can provide much more concise APIs for data processing.

---

# What is SQL?

- SQL is a domain-specific language used in:
  - RDBMS
  - Stream processing

```
SELECT SUM(foo) AS baz, end_of_window
FROM my_topic WHERE something_is_true(bizzle)
GROUP BY TUMBLING(timestamp, 1 HOUR)
HAVING baz > my_magic_number LIMIT 3;
```

---

# What is SQL?

- SQL is a domain-specific language used in:
  - RDBMS
  - Stream processing
- Relational algebra:
  - Projection
  - Filter
  - Aggregation

```
SELECT SUM(foo) AS baz, end_of_window
FROM my_topic WHERE something_is_true(bizzle)
GROUP BY TUMBLING(timestamp, 1 HOUR)
HAVING baz > my_magic_number LIMIT 3;
```

---

# What is SQL?

- SQL is a domain-specific language used in:
  - RDBMS
  - Stream processing
- Relational algebra:
  - Projection
  - Filter
  - Aggregation
- Syntax to operate on nested structures.

```
SELECT SUM(foo) AS baz, end_of_window
FROM my_topic WHERE something_is_true(bizzle)
GROUP BY TUMBLING(timestamp, 1 HOUR)
HAVING baz > my_magic_number LIMIT 3;
```

# A join in Java

```
package org.apache.beam.examples.cookbook;

import com.google.api.services.bigquery.model.TableRow;
...

public class JoinExamples {

    /** Join two collections, using country code as the key. */
    static PCollection<String> joinEvents(
        PCollection<TableRow> eventsTable, PCollection<TableRow> countryCodes) throws
    Exception {

        final TupleTag<String> eventInfoTag = new TupleTag<>();
        final TupleTag<String> countryInfoTag = new TupleTag<>();

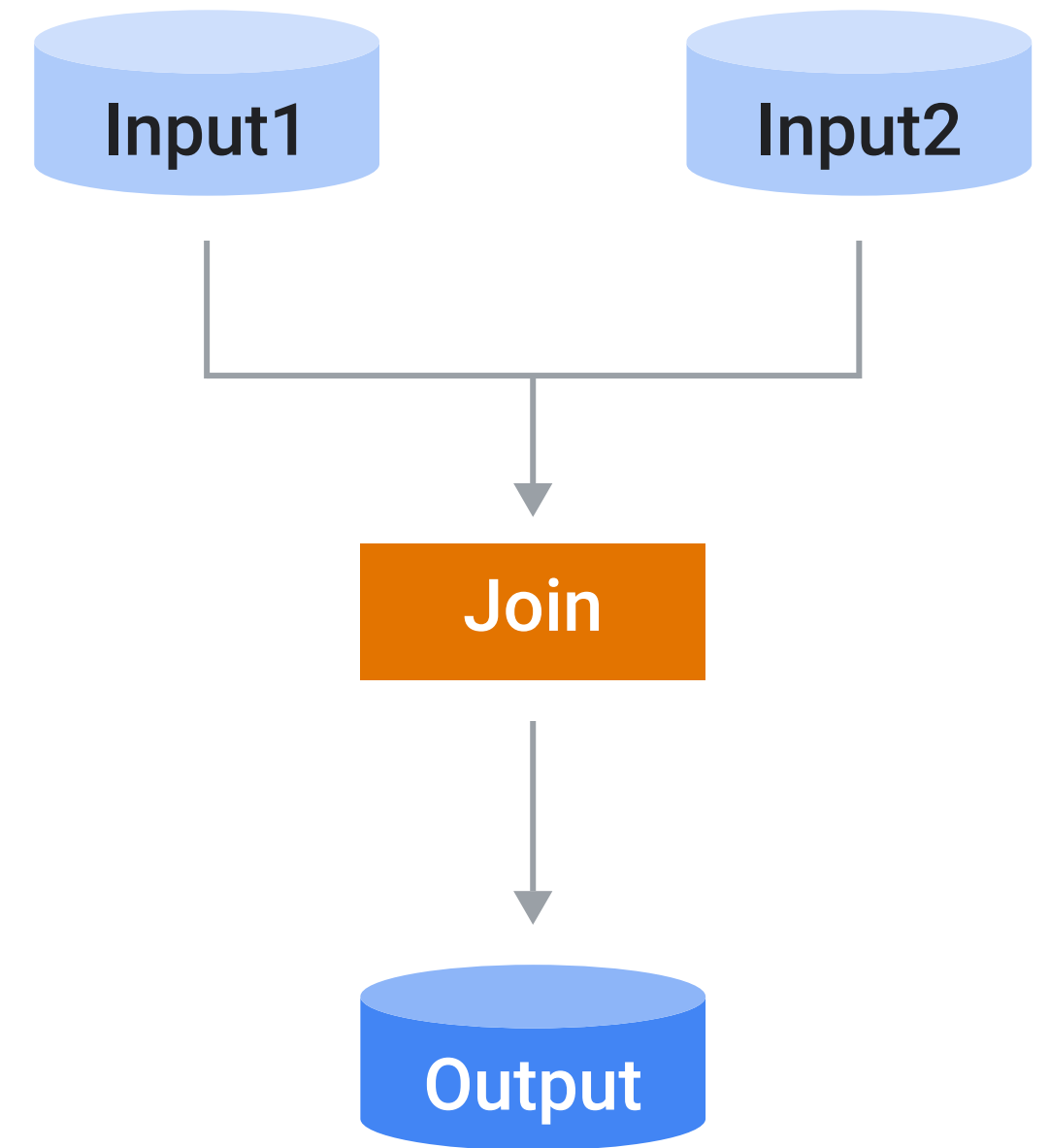
        // transform both input collections to tuple collections, where the keys are
        country
        // codes in both cases.
        PCollection<KV<String, String>> eventInfo = eventsTable.apply(ParDo.of(new
        ExtractEventDataFn()));
        PCollection<KV<String, String>> countryInfo = countryCodes.apply(ParDo.of(new
        ExtractCountryInfoFn()));

        // country code 'key' -> CGBKR (<event info>, <country name>)
        PCollection<KV<String, CoGbkResult>> kvpCollection =
            KeyedPCollectionTuple.of(eventInfoTag, eventInfo)
                .and(countryInfoTag, countryInfo)
                .apply(CoGroupByKey.create());

        // Process the CoGbkResult elements generated by the CoGroupByKey transform.
        // country code 'key' -> string of <event info>, <country name>
        PCollection<KV<String, String>> finalResultCollection =
            kvpCollection.apply("Process",
                ParDo.of(
                    new DoFn<KV<String, CoGbkResult>, KV<String, String>>() {
                        @ProcessElement
                        public void processElement(ProcessContext c) {
                            KV<String, CoGbkResult> e = c.element();
                            String countryCode = e.getKey();
                            String countryName = "none";
                            countryName = e.getValue().getOnly(countryInfoTag);
                            for (String eventInfo : c.element().getValue().getAll(eventInfoTag)) {
                                // Generate a string that combines information from both collection
                                values
                                c.output(KV.of(countryCode, "Country name: " + countryName + ", Event
                                info: " + eventInfo));
                            }
                        }
                    }
                ));
    }
}
```

```
// write to GCS
PCollection<String> formattedResults =
    finalResultCollection.apply("Format",
        ParDo.of(
            new DoFn<KV<String, String>, String>() {
                @ProcessElement
                public void processElement(ProcessContext c) {
                    String outputstring = "Country code: " + c.element().getKey() + ", " +
                    c.element().getValue();
                    c.output(outputstring);
                }
            }
        ));
return formattedResults;
}

...
public static void main(String[] args) throws Exception {
    Options options =
        PipelineOptionsFactory.fromArgs(args).withValidation().as(Options.class);
    Pipeline p = Pipeline.create(options);
    // the following two 'applies' create multiple inputs to our pipeline, one for
    each
    // of our two input sources.
    PCollection<TableRow> eventsTable =
        p.apply(BigQueryIO.readTableRows().from(GDELT_EVENTS_TABLE));
    PCollection<TableRow> countryCodes =
        p.apply(BigQueryIO.readTableRows().from(COUNTRY_CODES));
    PCollection<String> formattedResults = joinEvents(eventsTable, countryCodes);
    formattedResults.apply(TextIO.write().to(options.getOutput()));
    p.run().waitUntilFinish();
}
}
```

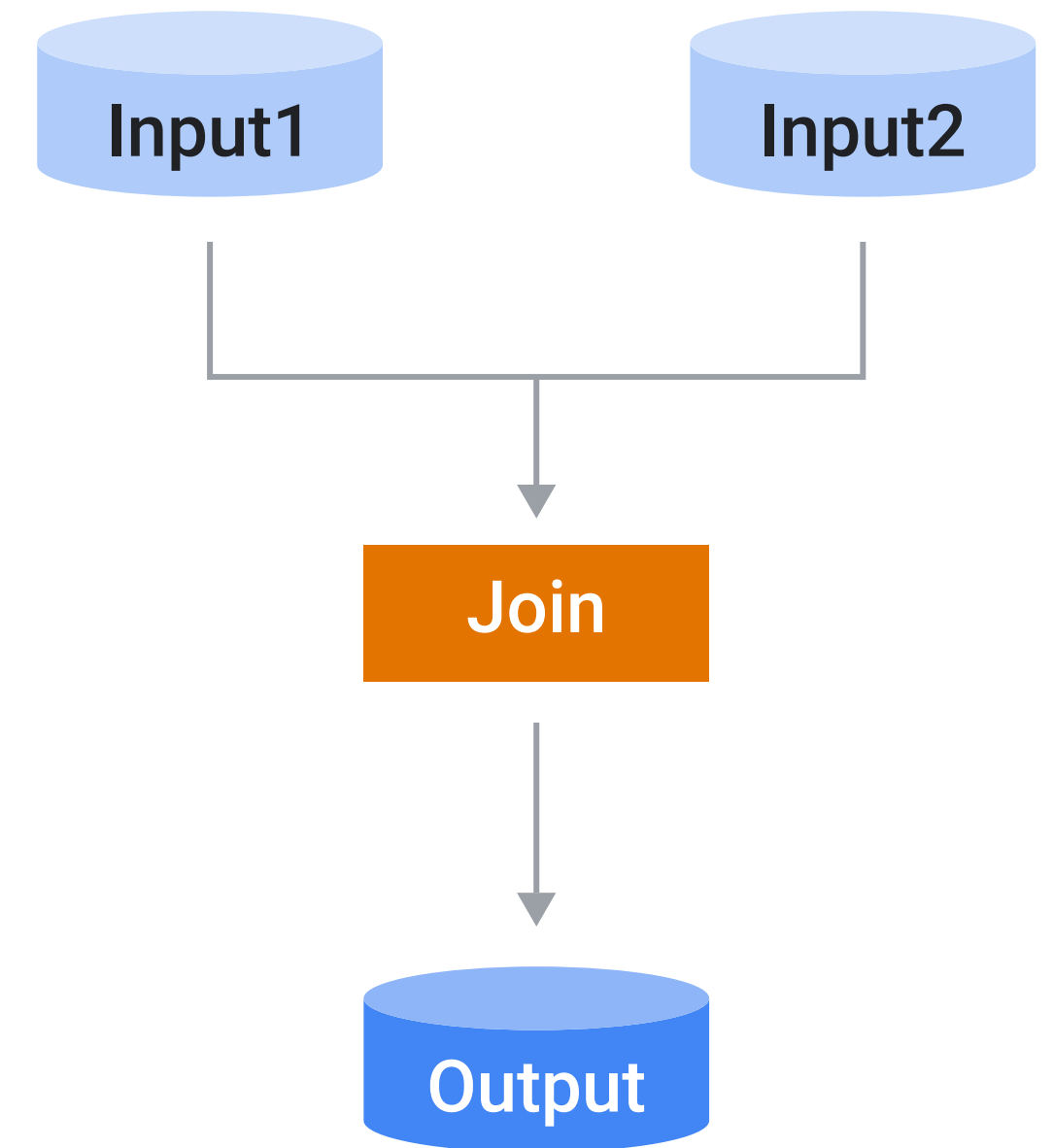




# A join in Scala (using Scio)

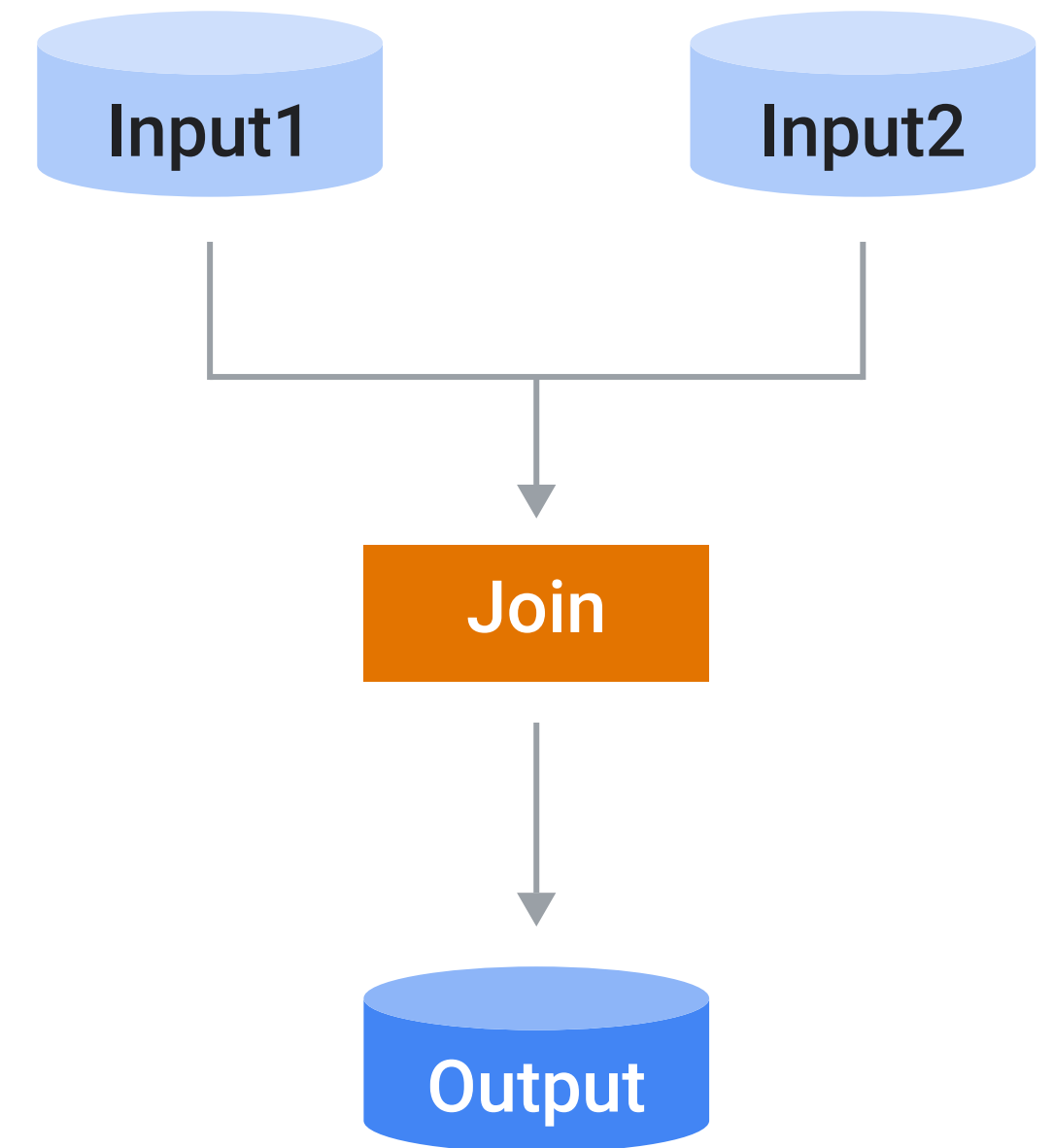
```
import com.spotify.scio._

object Join {
  def readInput(sc: ScioContext, path: String) = {
    val KeyLen = 10
    sc.textFile(path)
      .map((x: String) => (x.substring(0, KeyLen),
x.substring(KeyLen)))
  }
  def main(cmdlineArgs: Array[String]): Unit = {
    val (sc, args) = ContextAndArgs(cmdlineArgs)
    val left = readInput(sc, args("input1"))
    val right = readInput(sc, args("input2"))
    left.leftOuterJoin(right).saveAsTextFile(args("output"))
    sc.close()
  }
}
```



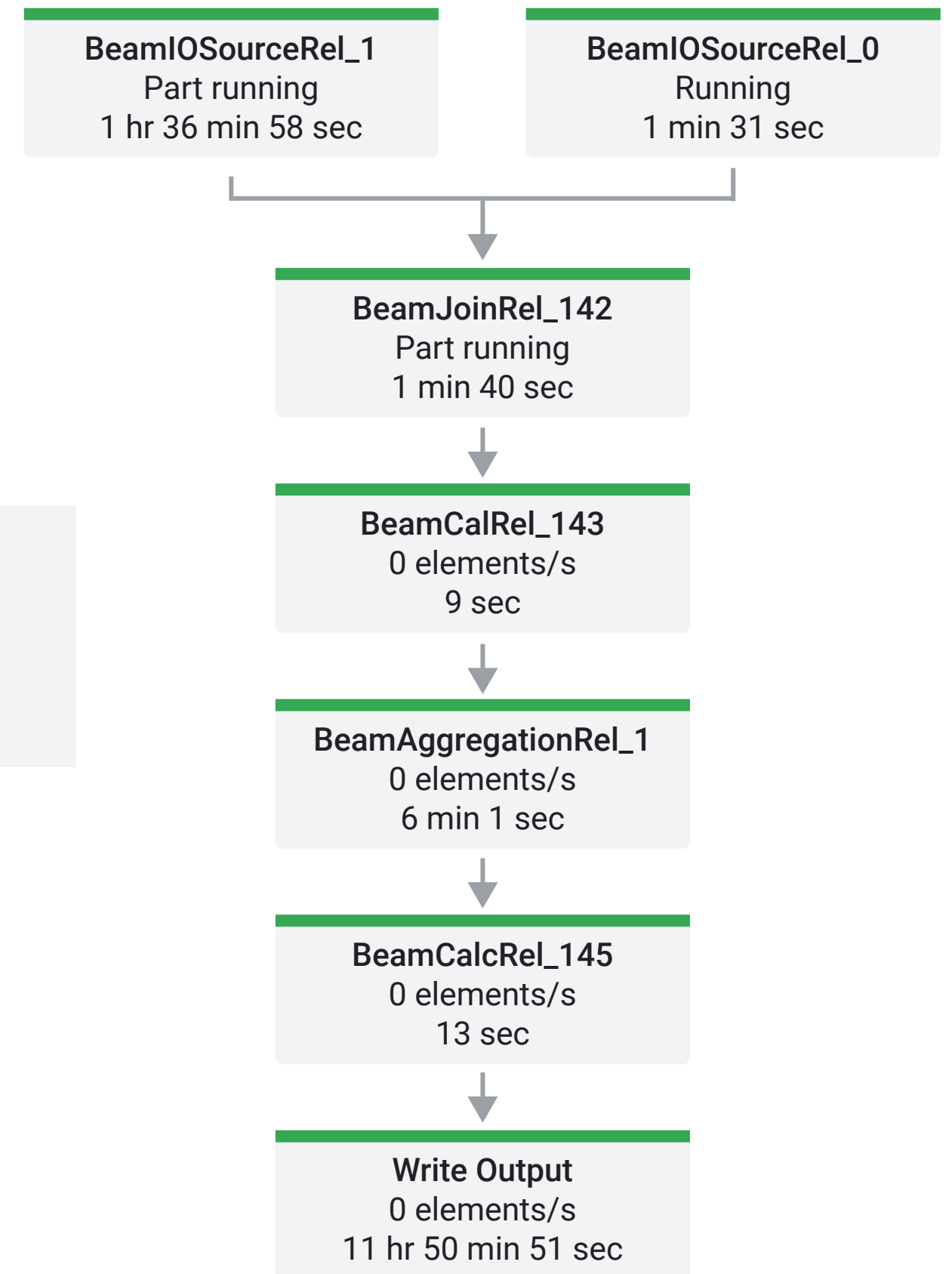
# Why SQL?

```
SELECT input1.*, input2.*  
FROM input1 LEFT OUTER JOIN input2  
    ON input1.Id = input2.Id
```



# SQL—Translated to PTransform

```
SELECT input1.*, input2.*  
FROM input1 LEFT OUTER JOIN input2  
  ON input1.Id = input2.Id
```



---

# Explaining components

Input

**BigQuery UI**

Input

**Analytical queries over historical data**

Input

**Data analyst**

---

# Explaining components

Input

Input

Input

BigQuery UI

Analytical queries over historical data

Data analyst

**Dataflow SQL UI**

**Analytical queries over real-time data**

**Data analyst**

---

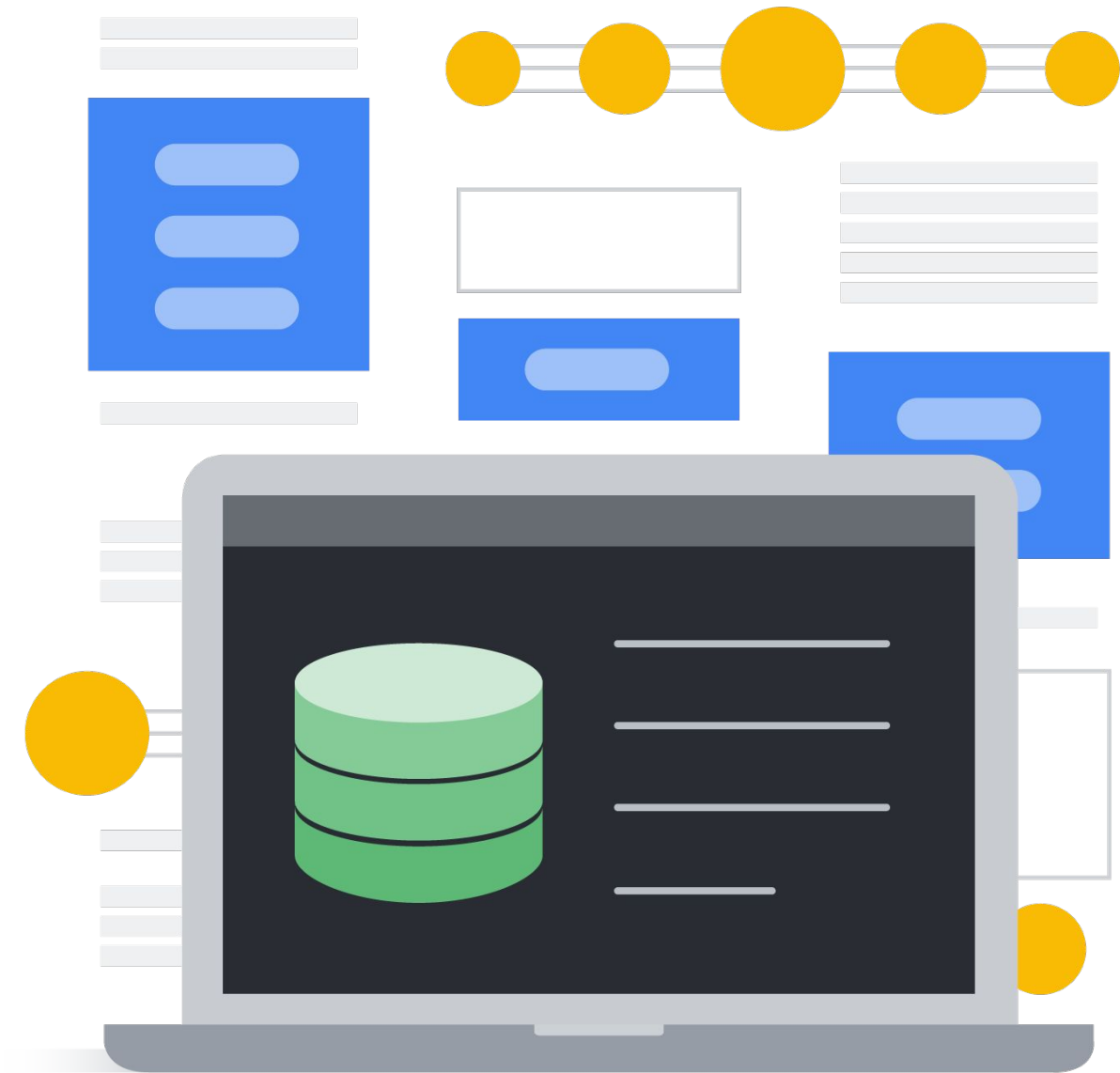
# Explaining components

Input	Input	Input
BigQuery UI	Analytical queries over historical data	Data analyst
Dataflow SQL UI	Analytical queries over real-time data	Data analyst
Beam SQL	Integrating SQL within a Beam pipeline	Data engineer

---

# Beam SQL

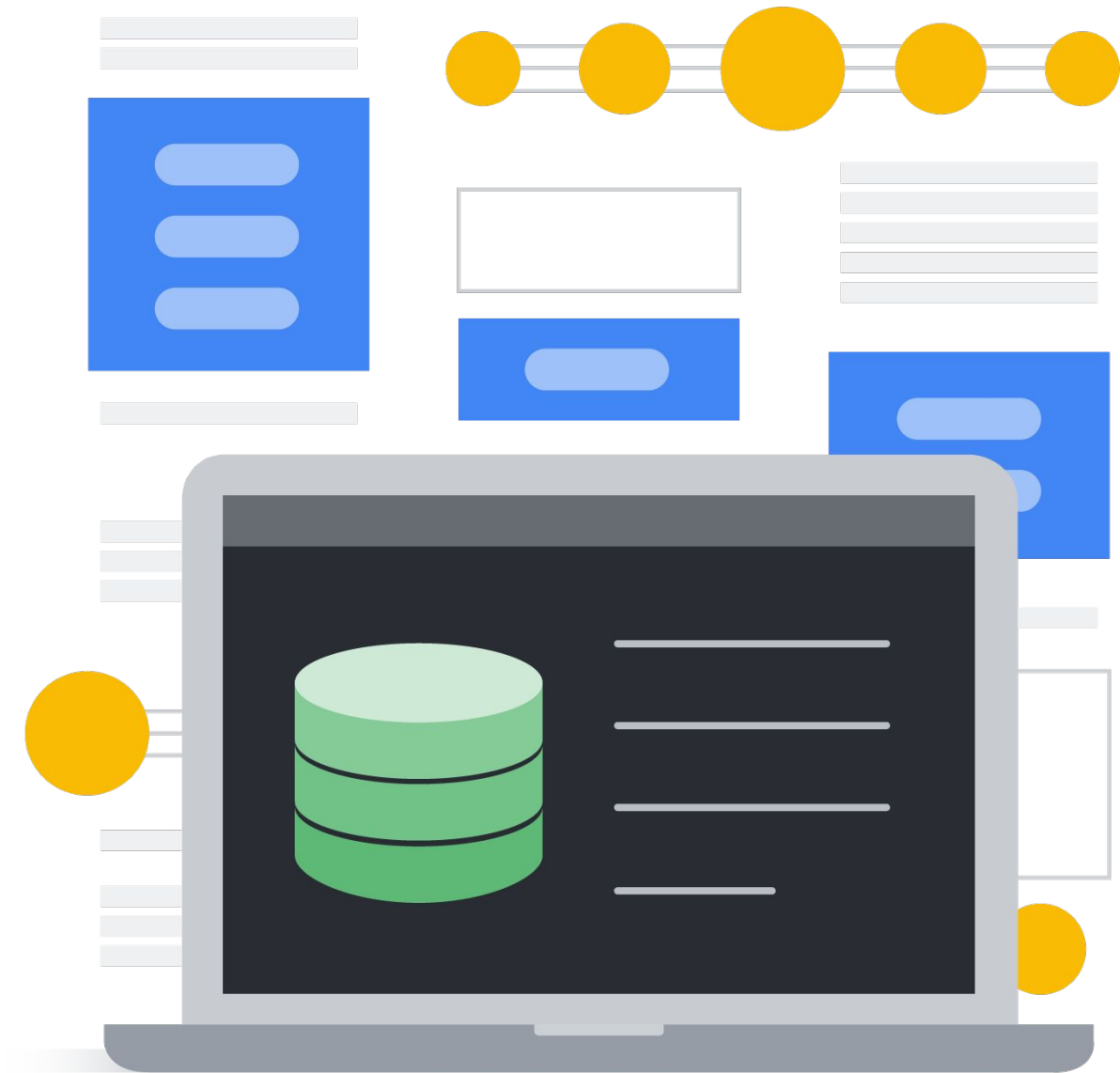
- Works with stream and batch inputs.



---

# Beam SQL

- Works with stream and batch inputs.
- Can be embedded in an existing pipeline using SqlTransform, which can be mixed with PTransforms.

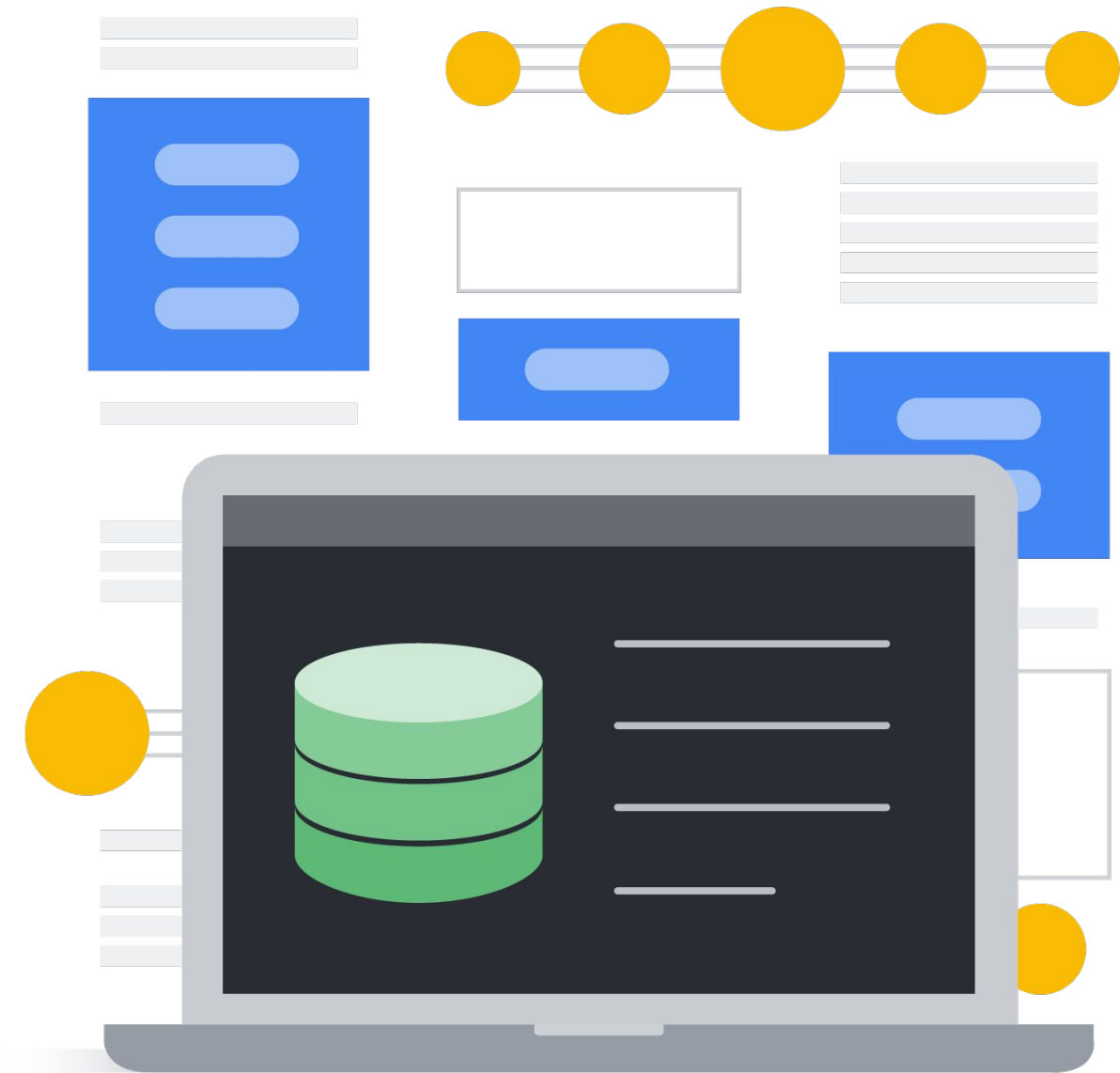




---

# Beam SQL

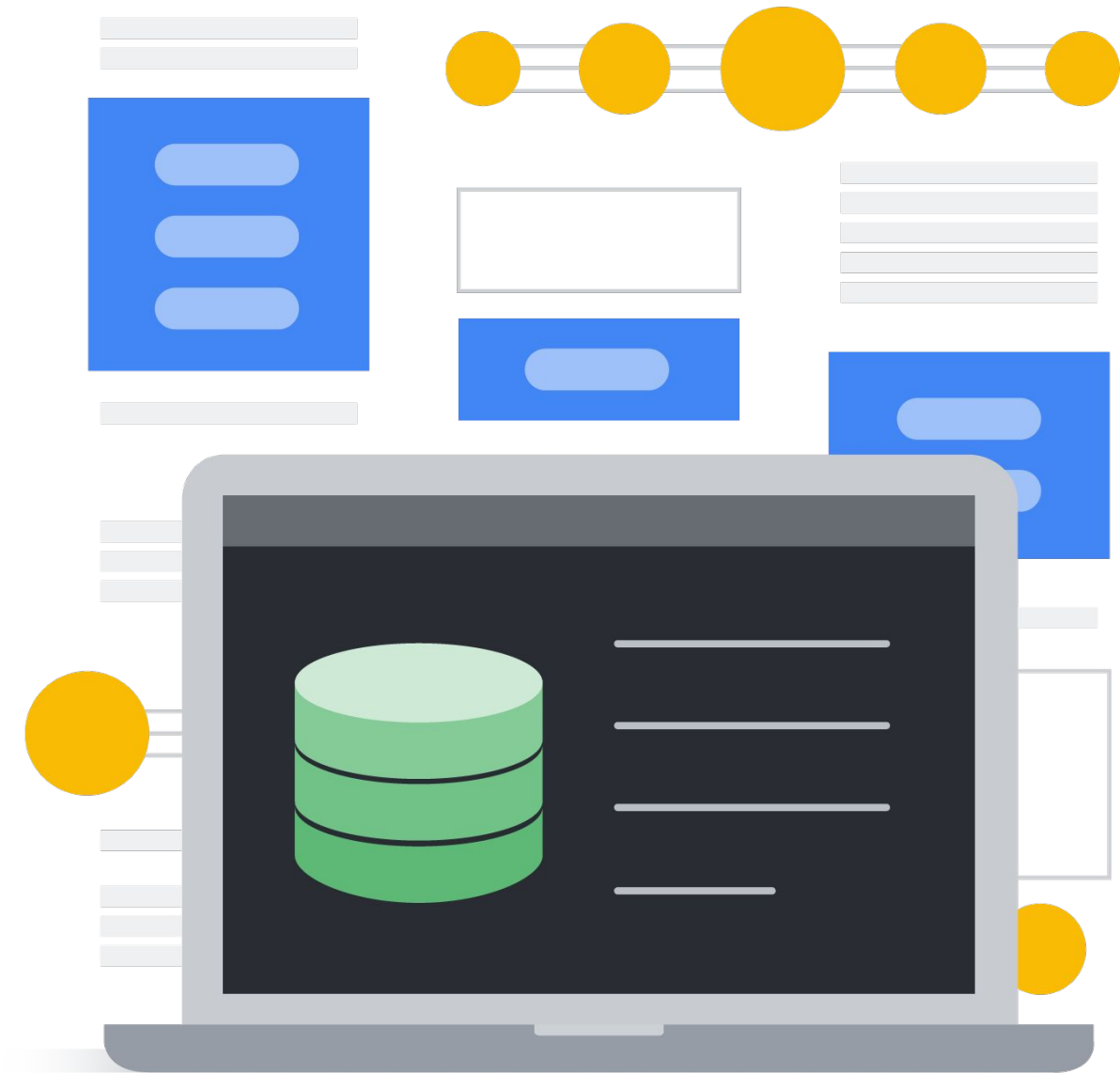
- Works with stream and batch inputs.
- Can be embedded in an existing pipeline using SqlTransform, which can be mixed with PTransforms.
- Supports UDFs in Java.



---

# Beam SQL

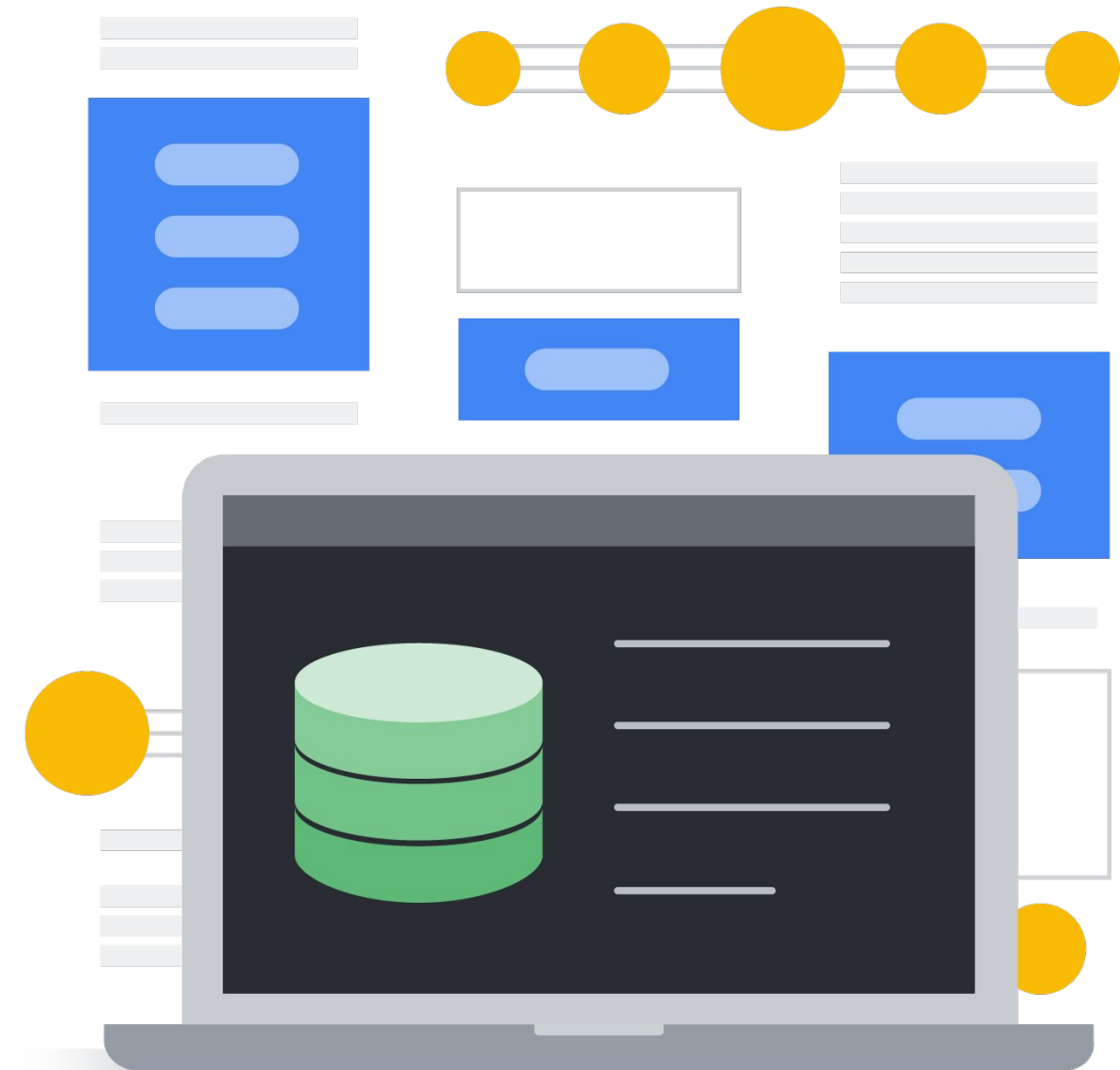
- Works with stream and batch inputs.
- Can be embedded in an existing pipeline using SqlTransform, which can be mixed with PTransforms.
- Supports UDFs in Java.
- Supports multiple dialects
  - Beam Calcite SQL
  - Google ZetaSQL



---

# Beam SQL

- Works with stream and batch inputs.
- Can be embedded in an existing pipeline using SqlTransform, which can be mixed with PTransforms.
- Supports UDFs in Java.
- Supports multiple dialects
  - Beam Calcite SQL
  - Google ZetaSQL
- Integrates with Schemas
  - Uses Row
- Stream aggregations support windows.

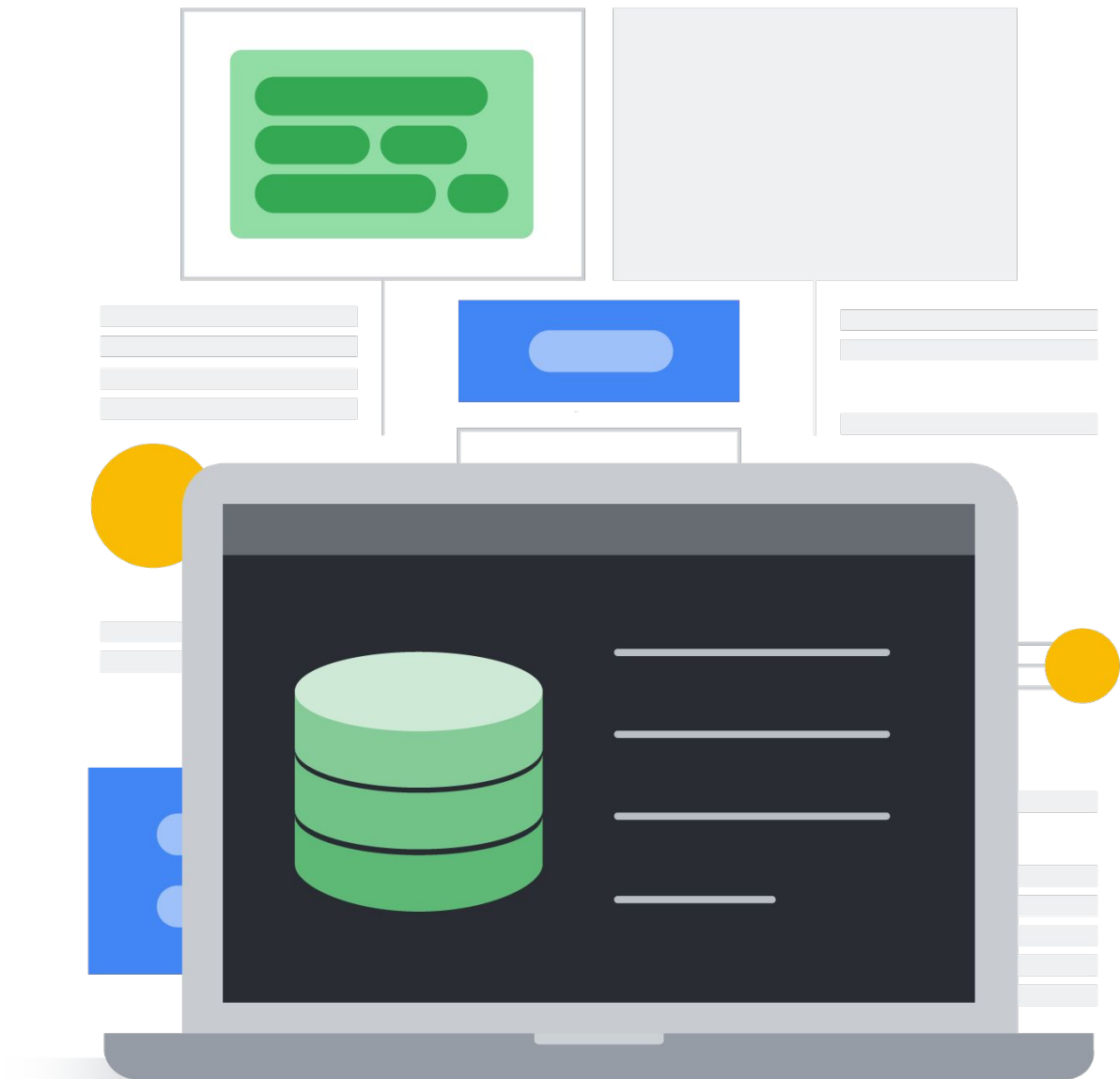


---

# Beam SQL dialects

## Apache Calcite

- Provides compatibility with other OSS SQL dialects (e.g. Flink SQL)
  - Copy-paste queries may require changes to table names, array indexing
- More mature implementation
  - Supports Java UDFs



---

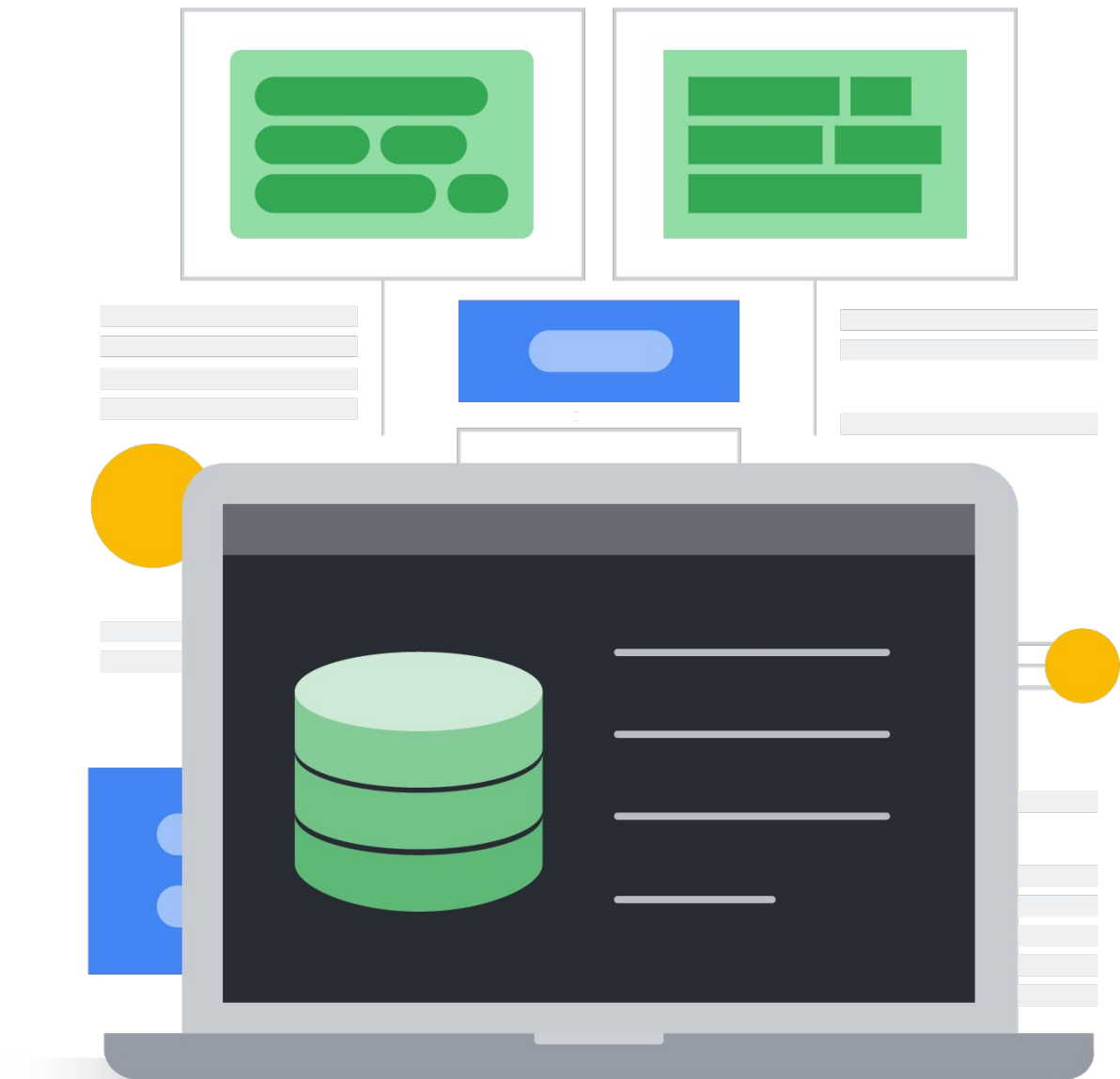
# Beam SQL dialects

## Apache Calcite

- Provides compatibility with other OSS SQL dialects (e.g. Flink SQL)
  - Copy-paste queries may require changes to table names, array indexing
- More mature implementation
  - Supports Java UDFs

## ZetaSQL

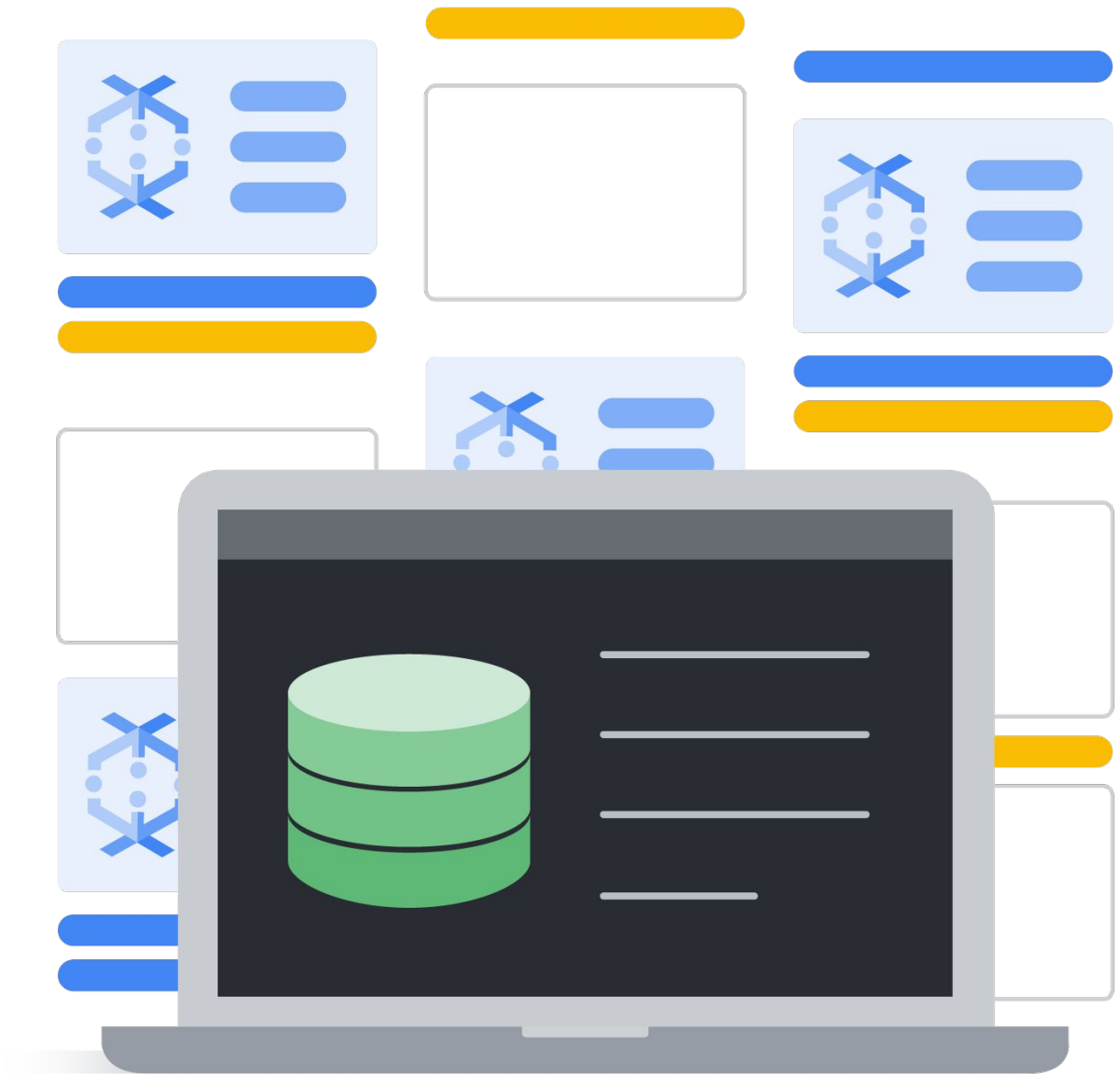
- Provides BigQuery compatibility
  - Copy-paste queries may require changes to table names



---

# Dataflow SQL

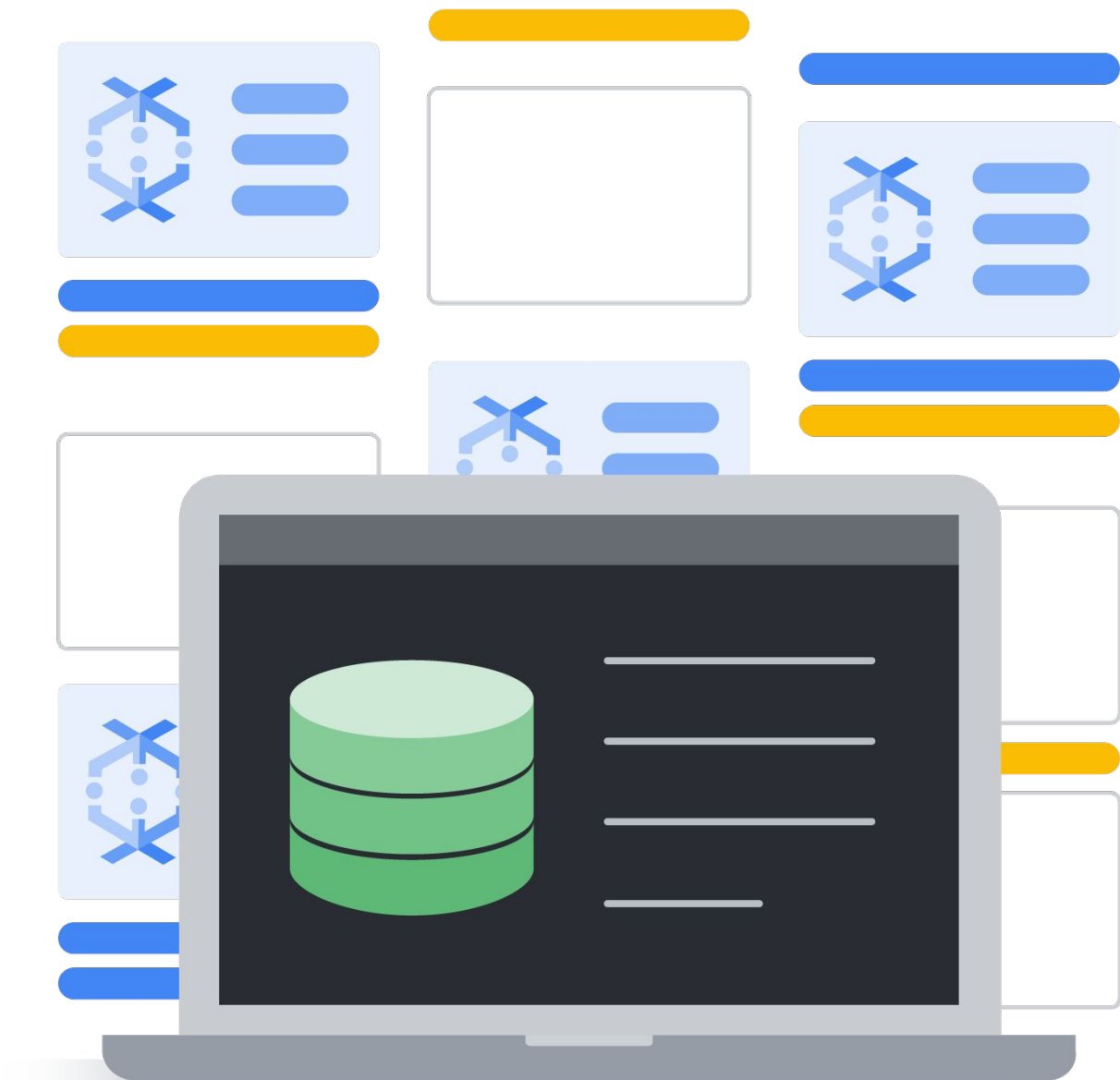
- It's a Beam ZetaSQL SqlTransform in a Dataflow Flex Template!



---

# Dataflow SQL

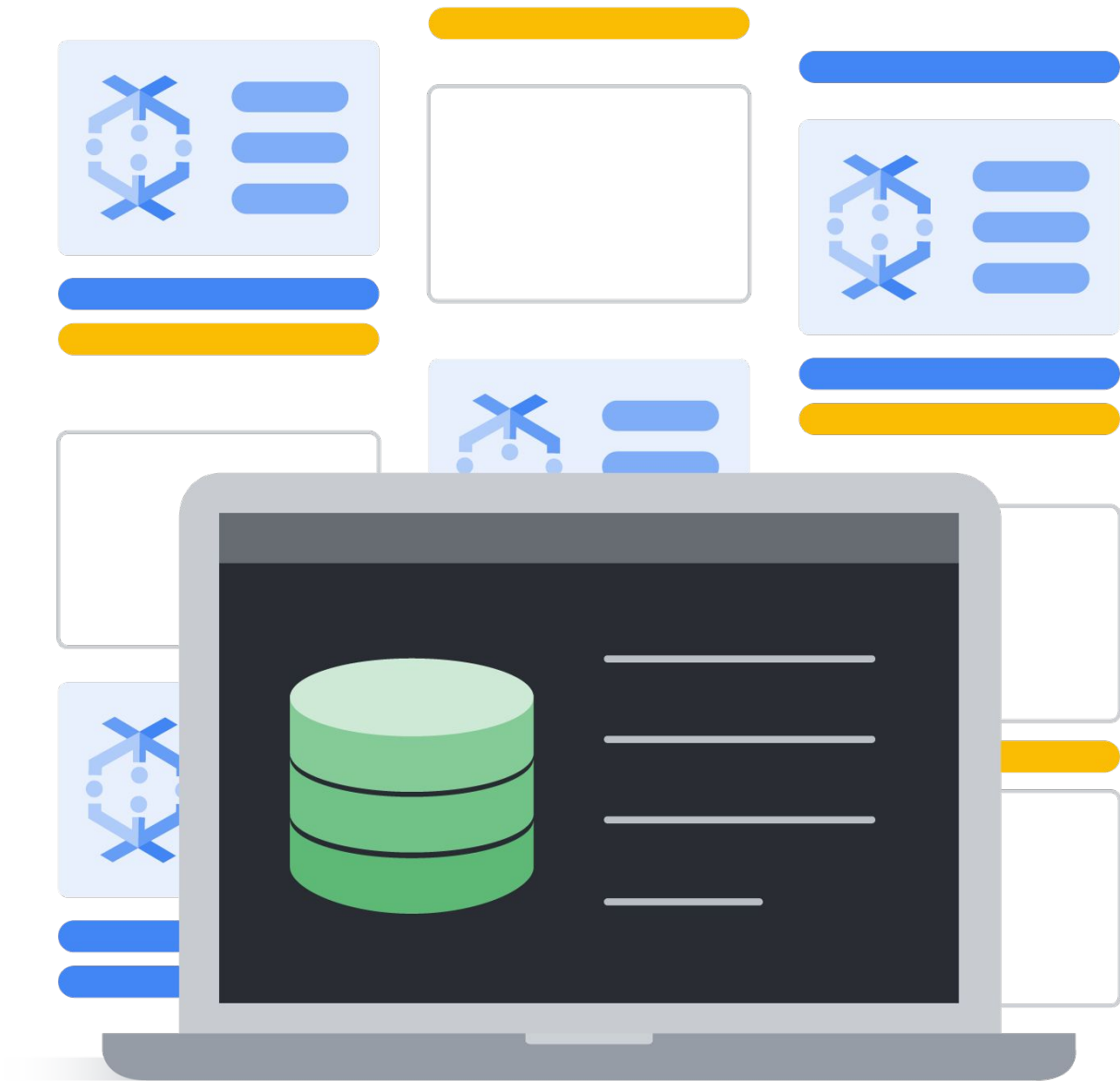
- It's a Beam ZetaSQL SqlTransform in a Dataflow Flex Template!
- Write Dataflow SQL queries in the BigQuery UI or gcloud CLI.



---

# Dataflow SQL

- It's a Beam ZetaSQL SqlTransform in a Dataflow Flex Template!
- Write Dataflow SQL queries in the BigQuery UI or gcloud CLI
- Uses ZetaSQL, the same dialect as BigQuery Standard SQL.

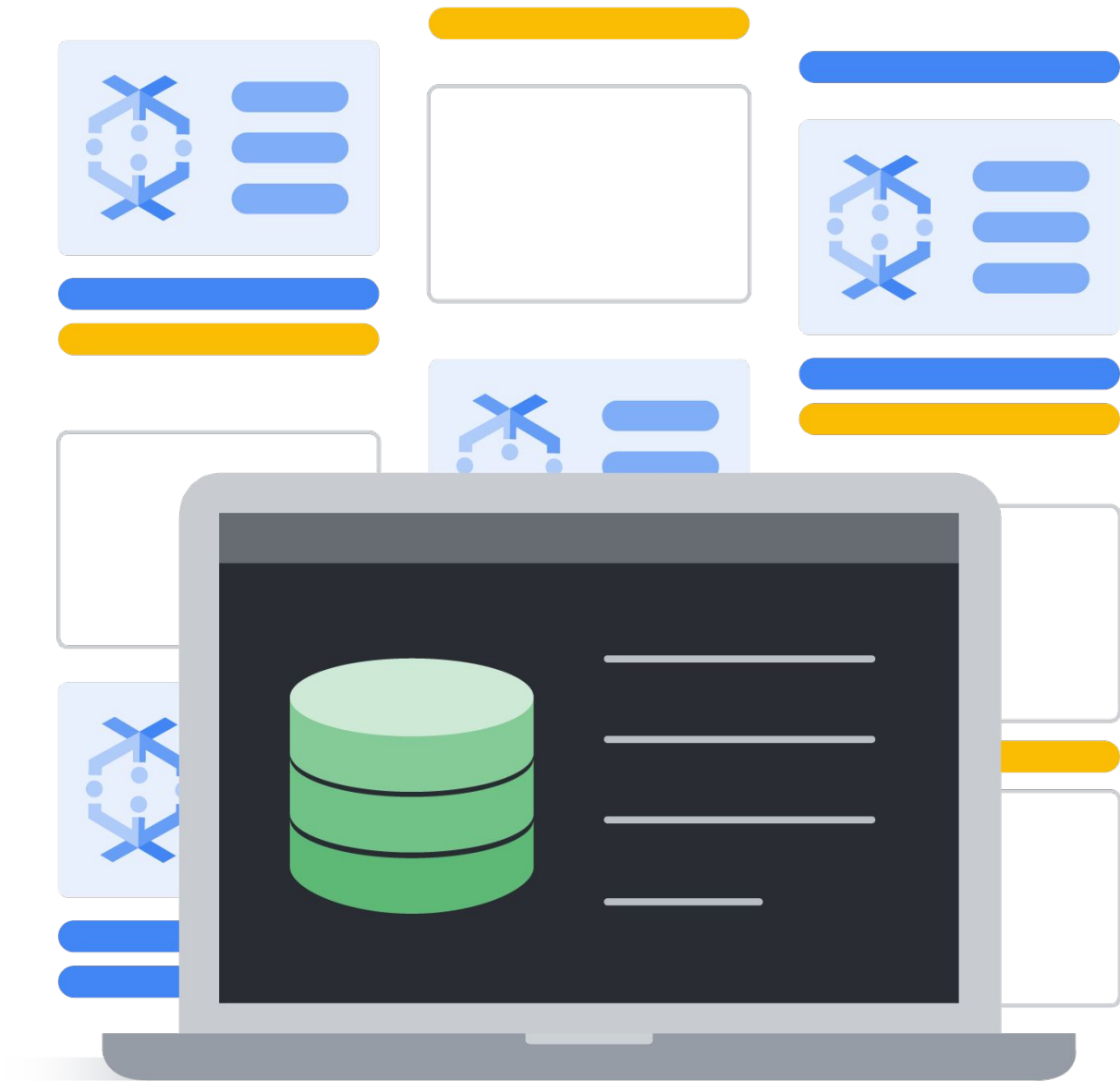




---

# Dataflow SQL

- It's a Beam ZetaSQL SqlTransform in a Dataflow Flex Template!
- Write Dataflow SQL queries in the BigQuery UI or gcloud CLI.
- Uses ZetaSQL, the same dialect as BigQuery Standard SQL.
- Optional engine for long running batch jobs.

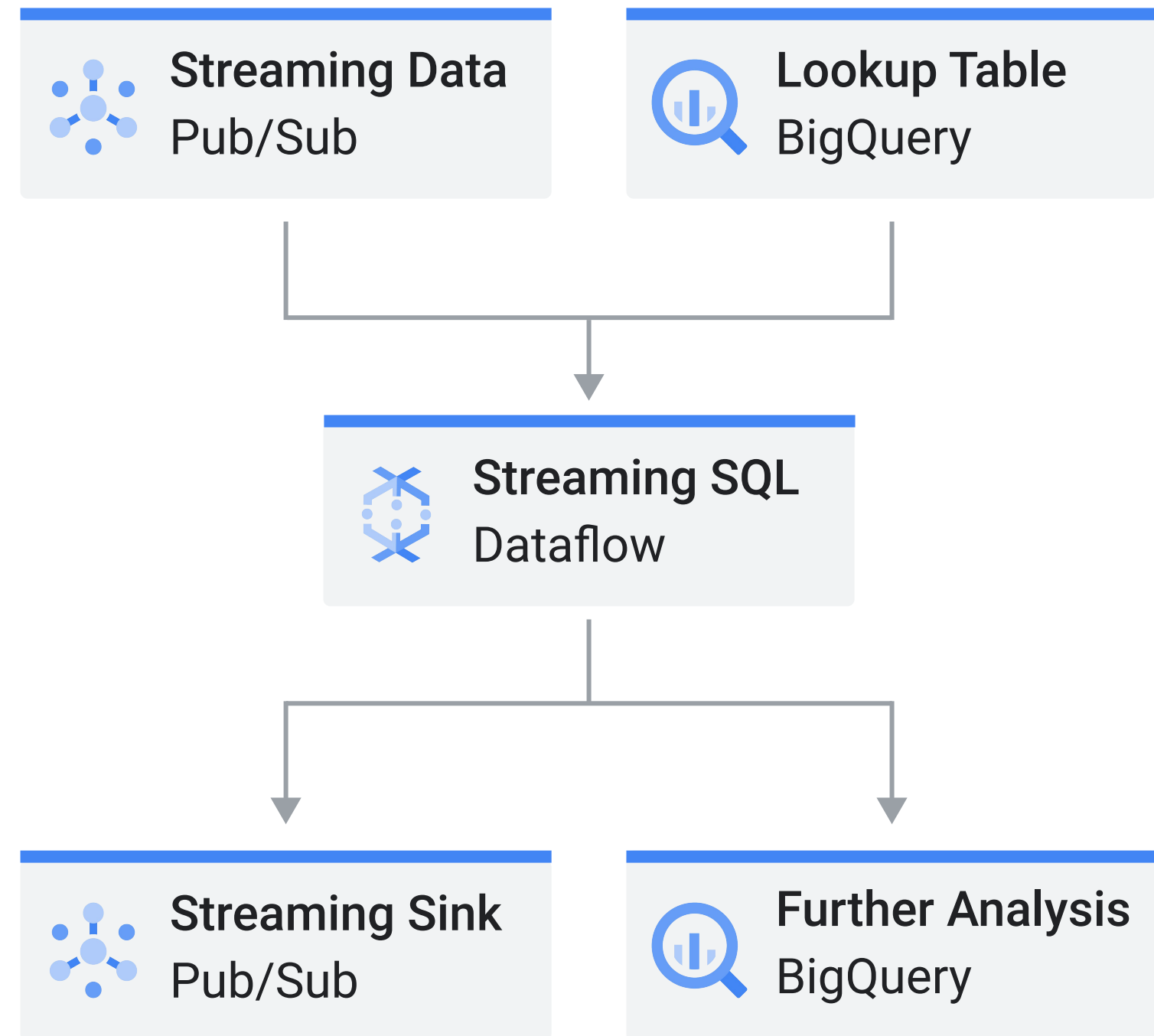


---

# Dataflow SQL

Streaming pipelines made easy

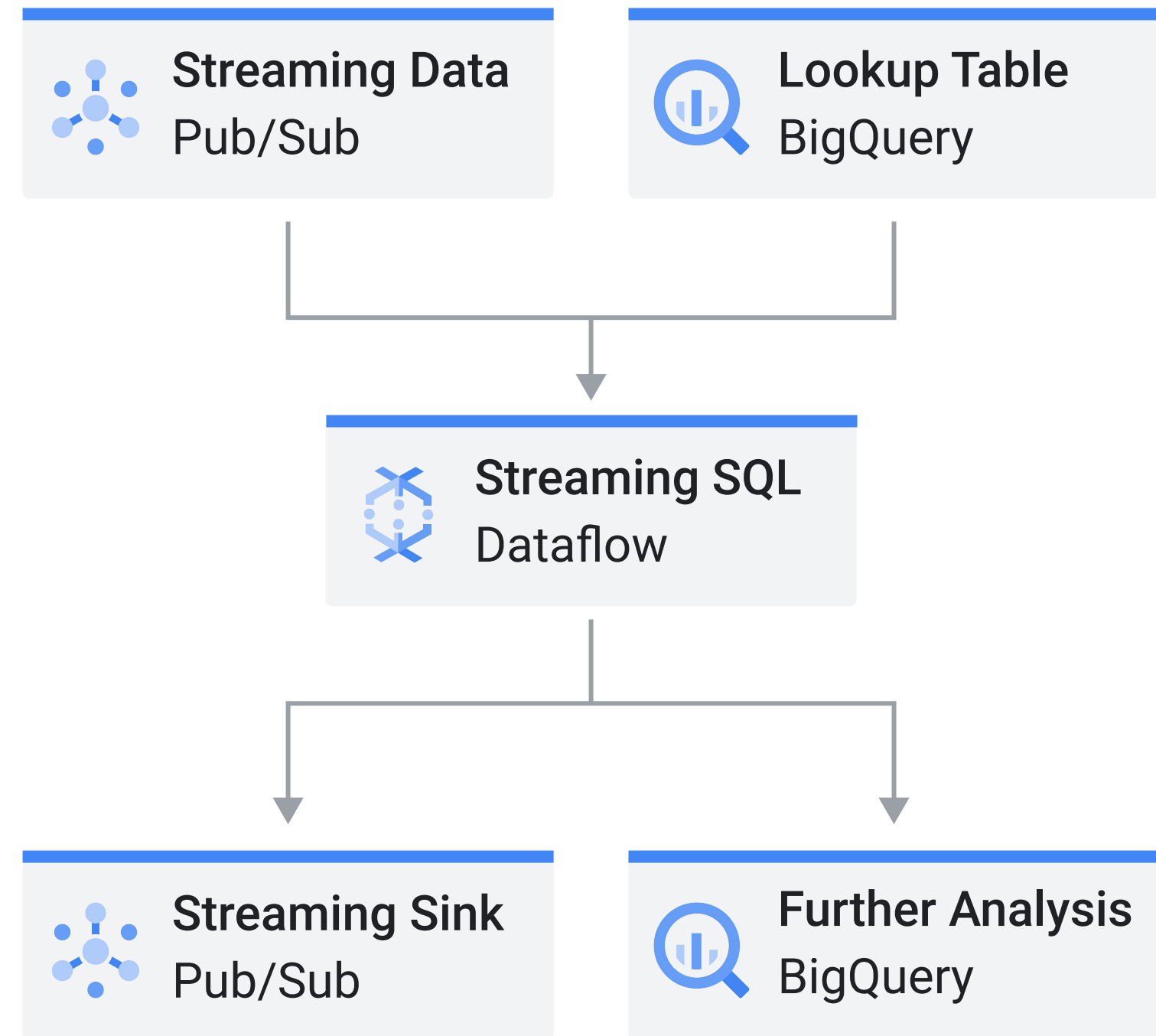
- Target of Dataflow SQL!



# Dataflow SQL

Streaming pipelines made easy

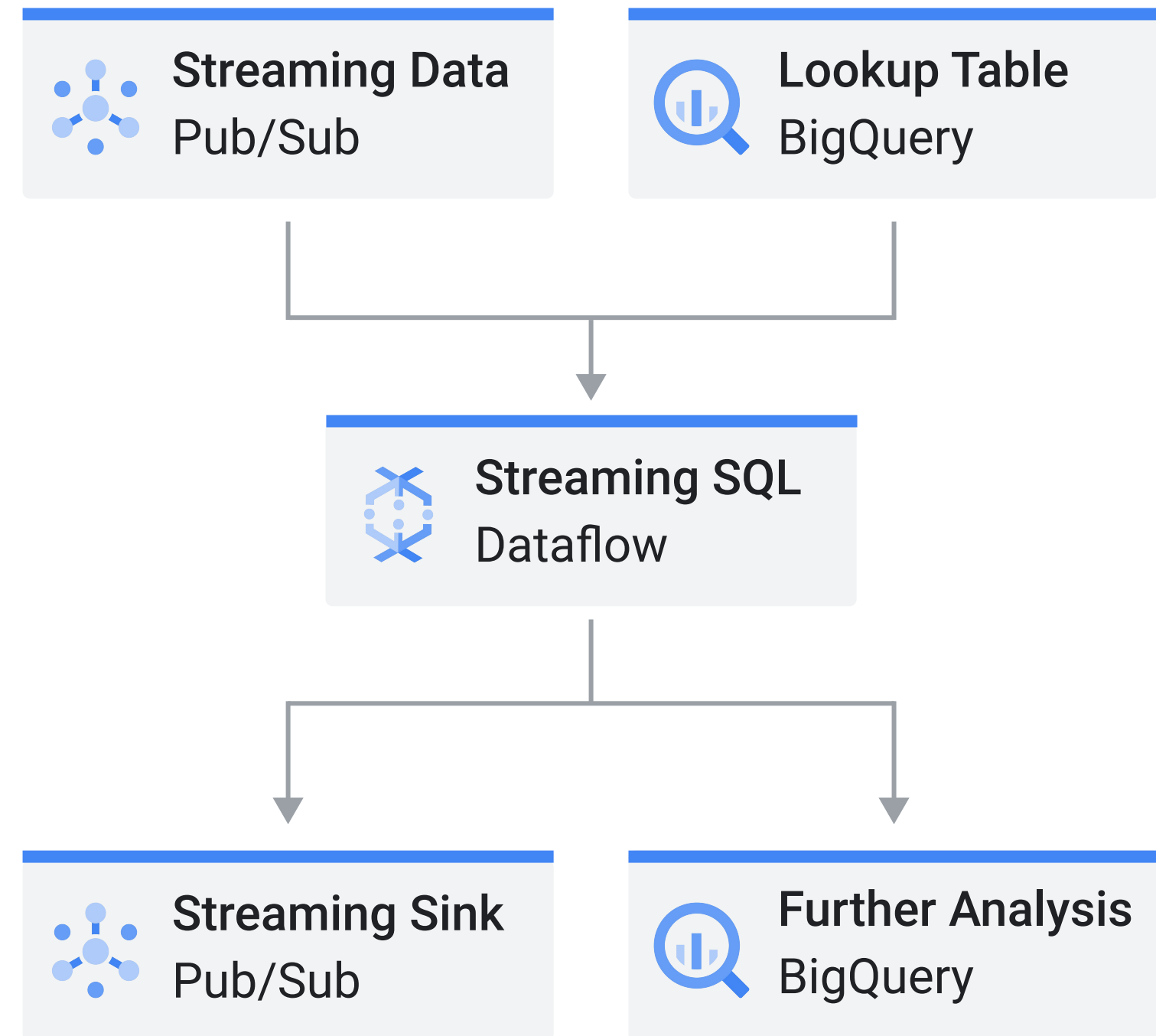
- Target of Dataflow SQL!
- An example use case would:
  - Select from PubSub
  - Join with batch data
  - Aggregate over Window
  - Publish to BigQuery or PubSub



# Dataflow SQL

Streaming pipelines made easy

- Target of Dataflow SQL!
- An example use case would:
  - Select from PubSub
  - Join with batch data
  - Aggregate over Window
  - Publish to BigQuery or PubSub
- Open framework with more connectors coming like Kafka and Bigtable.



# Dataflow SQL UI

The screenshot displays the Google Cloud Platform Dataflow SQL UI interface. The top navigation bar includes the Google Cloud Platform logo, the 'Dataflow cloud-workflows' dropdown menu, a search bar, and various utility icons (chat, help, notifications, and user profile). Below the navigation bar, the interface is divided into a left sidebar and a main content area.

**Left Sidebar:**

- BigQuery** (with a magnifying glass icon) and links to **FEATURES & INFO** and **SHORTCUTS**.
- Query history**
- Saved queries**
- Job history**
- Transfers**
- Scheduled queries**
- BI Engine**
- Resources** with a **+ ADD DATA** button and a search bar labeled 'Search for your tables and datasets'.
- google.com:clouddfe** (selected)
- Cloud Dataflow sources**

**Main Content Area:**


- Query editor** (with **HIDE EDITOR** and **FULL SCREEN** buttons). It contains a SQL query:

```
1 SELECT tr.payload.*, sr.sales_region
2 FROM pubsub.topic.`google.com:clouddfe`.transactions as tr
3 INNER JOIN bigquery.table.`google.com:clouddfe`.apilloud_test.dataflow_sql_dataset AS sr
4 ON tr.payload.state = sr.state_code
```
- Cloud Dataflow engine alpha** section with a **+ Create Cloud Dataflow job** button and a **More** dropdown menu.
- google.com:clouddfe** project name with **CREATE DATASET** and **PIN PROJECT** buttons.
- Datasets and tables available** section with the text: 'Use the Resources tree to view your data, or create a new dataset using the controls above'.

# Dataflow SQL UI

### Create Cloud Dataflow job

**Job name**  
Must be unique among running jobs. Use lowercase letters, numbers, and hyphens (-).

**Regional endpoint**   
Choose where to deploy Cloud Dataflow workers and store metadata for the job.

**Destination**

<b>Project name</b>	<b>Dataset name</b>
<input type="text" value="Dataflow cloud-workflows"/>	<input type="text" value="apilloud_test"/>

**Table name**  
The job will create the specified destination table if needed, or the table must be empty if it already exists.

# Dataflow SQL UI

Google Cloud Platform

davidsabater-petproject

Search products and resources

dfsqli-31a8d5a...

CLONE

STOP

CREATE SNAPSHOT

SHARE

MAX. TIME

JOB GRAPH

JOB METRICS

Run SQL Query

BeamIOSourceRel\_0

Running

0 sec

1 stage

BeamIOSourceRel\_1

Running

4 sec

1 stage

BeamSideInp...JoinRel\_32

Running

0 sec

3 stages

BeamZetaSqlCalcRel\_33

Running

0 sec

1 stage

Write to bi...table2021`

Running

0 sec

2 stages

Job info

Job name

dfsqli-31a8d5a4-177b582f4b7

Job ID

2021-02-18\_06\_21\_22-15922696997235364879

Job type

Streaming

Job status

Running

SDK version

Apache Beam SDK for Java 2.29.0-SNAPSHOT

Job region

us-central1

Worker location

us-central1-a

Current workers

1

Latest worker status

Worker pool started.

Start time

18 February 2021 at 14:21:24 GMT+0

Elapsed time

10 min 24 sec

Encryption type

Google-managed key

Resource metrics

Current vCPUs

2

Total vCPU time

0.261 vCPU hr

Current memory

7.5 GB

Total memory time

0.979 GB hr

Current HDD PD

30 GB

Total HDD PD time

3.917 GB hr

Current SSD PD

0 B

Total SSD PD time

0 GB hr

Total streaming data processed

336.42 KB

---

# Dataflow SQL CLI

```
$ gcloud dataflow sql query 'SELECT SUM(foo) AS baz, end_of_window  
FROM my_topic WHERE something_is_true(bizzle)  
GROUP BY TUMBLING(timestamp, 1 HOUR)'
```



---

# Using SQL statements within existing pipelines

```
//run a simple query, and register the output as a table in BeamSql;  
String sql1 = "select MY_FUNC(c1), c2 from PCOLLECTION";  
PCollection<Row> outputTableA = inputTableA.apply(  
    SqlTransform  
        .query(sql1)  
        .addUdf("MY_FUNC", MY_FUNC.class, "FUNC");
```

---

# Dataflow SQL template

```
/** A simplification of the Dataflow SQL template. */
Pipeline pipeline = Pipeline.create(options);

DataCatalogTableProvider tableProvider =
    DataCatalogTableProvider.create(options);

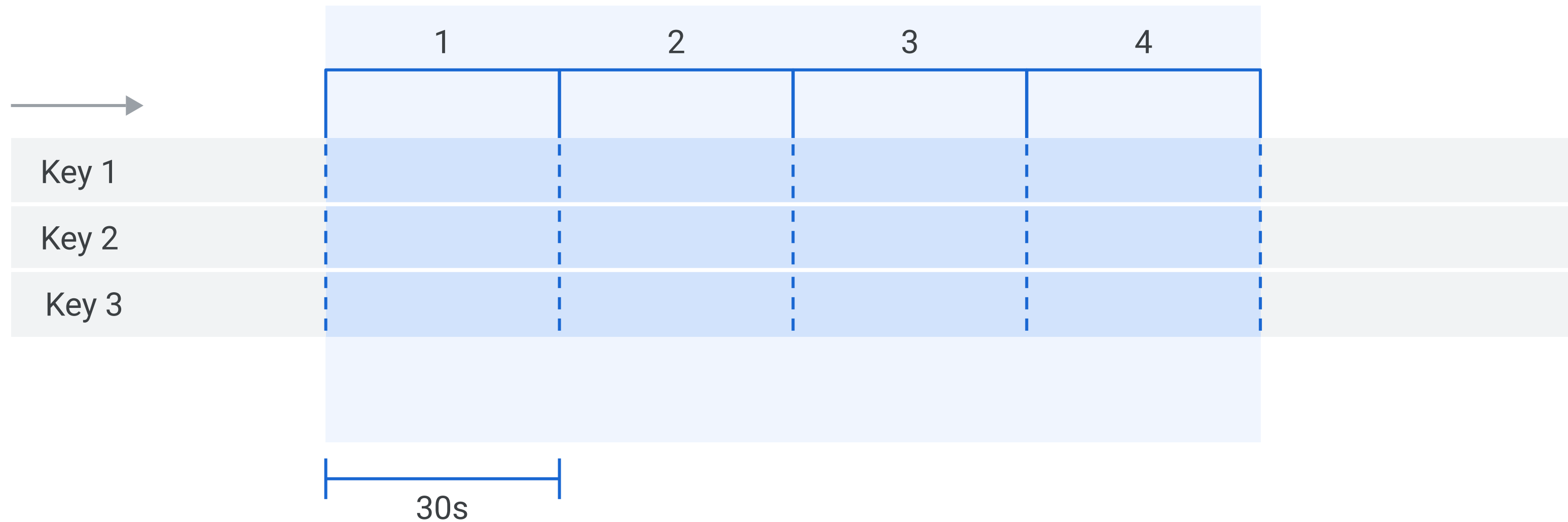
SqlTransform queryResult =
    p.apply(SqlTransform.query(options.getQueryString())
        .withDefaultTableProvider(tableProvider));
for (Output output : options.getOutputs()) {
    queryResult.apply(createSink(output, tableProvider));
}
pipeline.run();
```

# Dataflow SQL and DataFrames

## Agenda



# Windows—Tumbling windows (fixed windows)

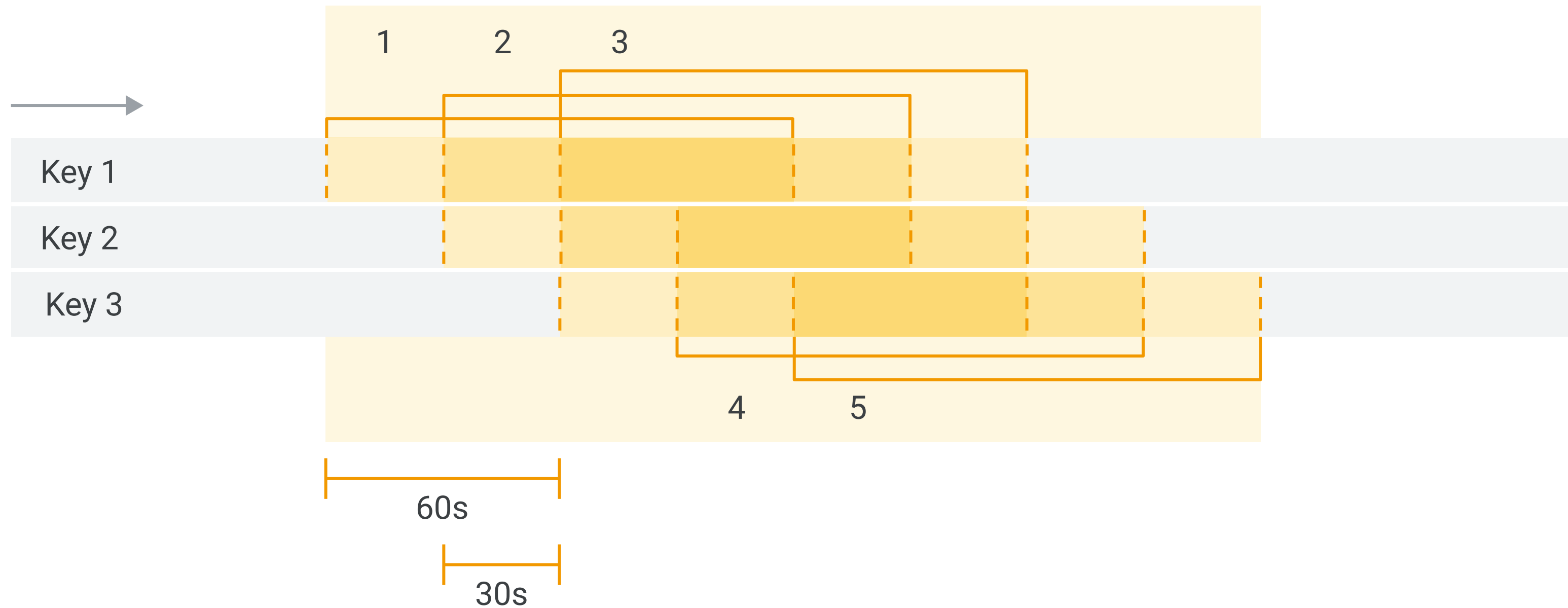


---

## Use case: Total purchases per product

```
SELECT productId,  
TUMBLE_START("INTERVAL 10 SECOND") as period_start, COUNT(transactionId)  
AS num_purchases  
FROM pubsub.topic.`instant-insights`.`retaildemo-online-purchases-json`  
AS pr  
GROUP BY productId,  
        TUMBLE(pr.event_timestamp, "INTERVAL 10 SECOND")
```

# Windows—Hopping windows (sliding windows)

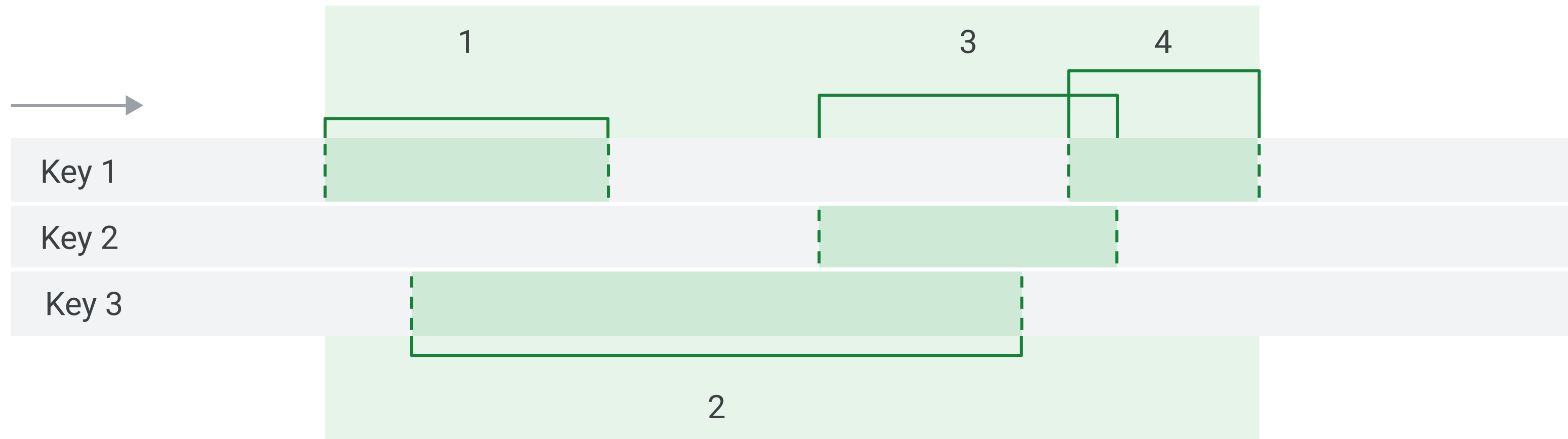


---

## Use case: Total purchases per product

```
SELECT productId,  
       HOP_START("INTERVAL 10 SECOND",  
                 "INTERVAL 30 SECOND") as period_start,  
       HOP_END("INTERVAL 10 SECOND",  
               "INTERVAL 30 SECOND") as period_end,  
       COUNT(transactionId) AS num_purchases  
FROM pubsub.topic.`instant-insights`.`retaildemo-online-purchases-json`  
AS pr  
GROUP BY productId,  
       HOP(pr.event_timestamp,  
           "INTERVAL 10 SECOND",  
           "INTERVAL 30 SECOND")
```

# Windows—Session windows





---

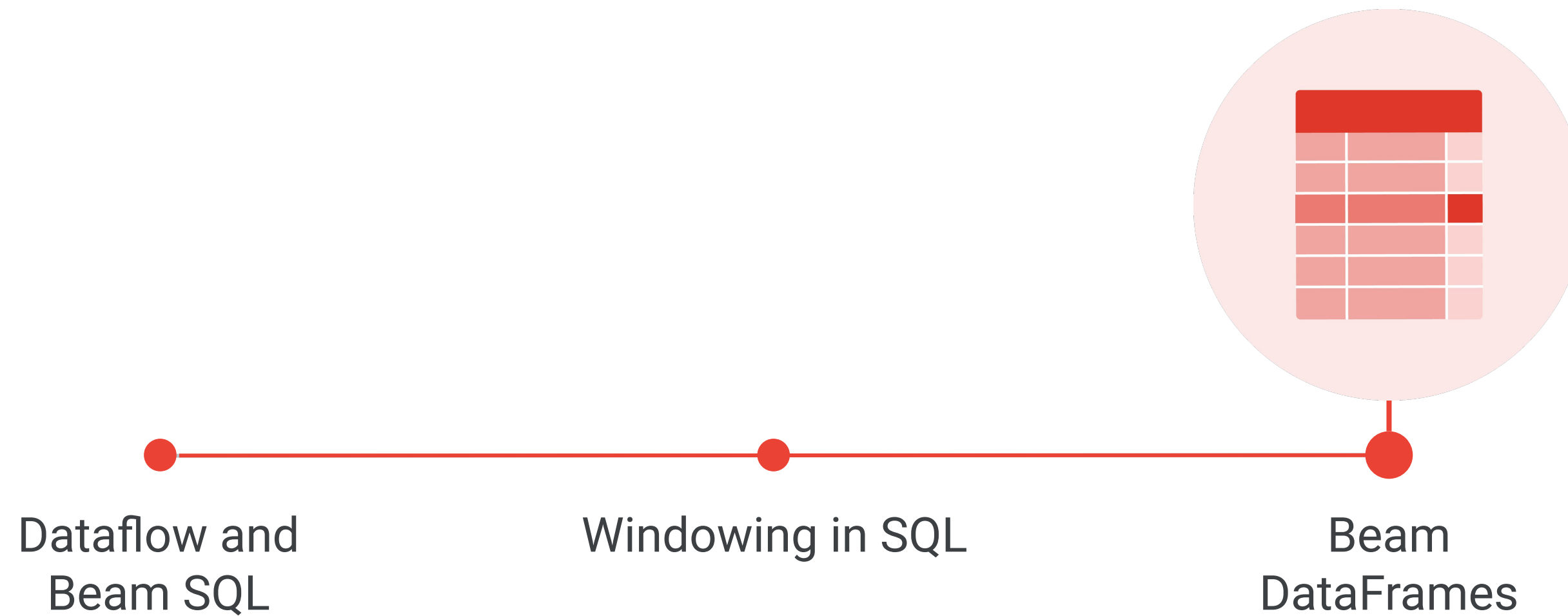
## Use case: Transactions per user session

```
SELECT userId,  
       SESSION_START("INTERVAL 10 MINUTE") AS interval_start,  
       SESSION_END("INTERVAL 10 MINUTE") AS interval_end ,  
       COUNT(transactionId) AS num_transactions  
FROM pubsub.topic.`instant-insights`.`retaildemo-online-purchases-json`  
AS pr  
GROUP BY userId,  
       SESSION(pr.event_timestamp, "INTERVAL 10 MINUTE")
```

---

# Dataflow SQL and DataFrames

## Agenda



# Beam DataFrames overview

- A more Pythonic expressive API compatible with Pandas DataFrames.

recipe	fruit	quantity	unit_cost	→	recipe	is_berry	total_quantity	total_price
pie	strawberry	3	\$1.50		pie	True	7	\$16.00
pie	raspberry	1	\$3.50		muffin	False	3	\$6.00
pie	blackberry	1	\$4.00		muffin	True	2	\$4.00
pie	blueberry	2	\$2.00					
muffin	banana	3	\$2.00					
muffin	blueberry	2	\$2.00					

# Beam DataFrames overview

- A more Pythonic expressive API compatible with Pandas DataFrames.
- Parallel processing with Beam model.

recipe	fruit	quantity	unit_cost	→	recipe	is_berry	total_quantity	total_price
pie	strawberry	3	\$1.50		pie	True	7	\$16.00
pie	raspberry	1	\$3.50		muffin	False	3	\$6.00
pie	blackberry	1	\$4.00		muffin	True	2	\$4.00
pie	blueberry	2	\$2.00					
muffin	banana	3	\$2.00					
muffin	blueberry	2	\$2.00					

# Beam DataFrames overview

- A more Pythonic expressive API compatible with Pandas DataFrames.
- Parallel processing with Beam model.
- Think of Beam DataFrames as a domain-specific language (DSL) for Beam pipelines.

recipe	fruit	quantity	unit_cost	→	recipe	is_berry	total_quantity	total_price
pie	strawberry	3	\$1.50		pie	True	7	\$16.00
pie	raspberry	1	\$3.50		muffin	False	3	\$6.00
pie	blackberry	1	\$4.00		muffin	True	2	\$4.00
pie	blueberry	2	\$2.00					
muffin	banana	3	\$2.00					
muffin	blueberry	2	\$2.00					

# Beam DataFrames overview

- A more Pythonic expressive API compatible with Pandas DataFrames.
- Parallel processing with Beam model.
- Think of Beam DataFrames as a domain-specific language (DSL) for Beam pipelines.
- Provides access to familiar programming interfaces within a Beam pipeline.

recipe	fruit	quantity	unit_cost	→	recipe	is_berry	total_quantity	total_price
pie	strawberry	3	\$1.50		pie	True	7	\$16.00
pie	raspberry	1	\$3.50		muffin	False	3	\$6.00
pie	blackberry	1	\$4.00		muffin	True	2	\$4.00
pie	blueberry	2	\$2.00					
muffin	banana	3	\$2.00					
muffin	blueberry	2	\$2.00					

---

# Using DataFrames—GroupBy

```
pc = p | beam.Create(['strawberry', 'raspberry', 'blackberry',  
                     'blueberry', 'banana'])
```

```
pc | GroupBy(lambda name: name[0])
```



Arbitrary expression

---

# Using DataFrames—GroupBy

```
pc = p | beam.Create(['strawberry', 'raspberry', 'blackberry',  
                     'blueberry', 'banana'])
```

```
pc | GroupBy(lambda name: name[0])
```

```
('s', ['strawberry'])  
('r', ['raspberry'])  
('b', ['blackberry', 'blueberry', 'banana'])
```



---

# Using DataFrames—Transform

```
def my_function(df):  
    ...  
    return result
```

```
output = input | DataframeTransform(my_function)
```

---

# Using DataFrames—Transform

```
def my_function(df):  
    df['C'] = df.A + 2*df.B  
    result = df.groupby('C').sum().filter('A < 0')  
    return result
```

```
output = input | DataframeTransform(my_function)
```

Non-functional APIs  
are supported as well.

---

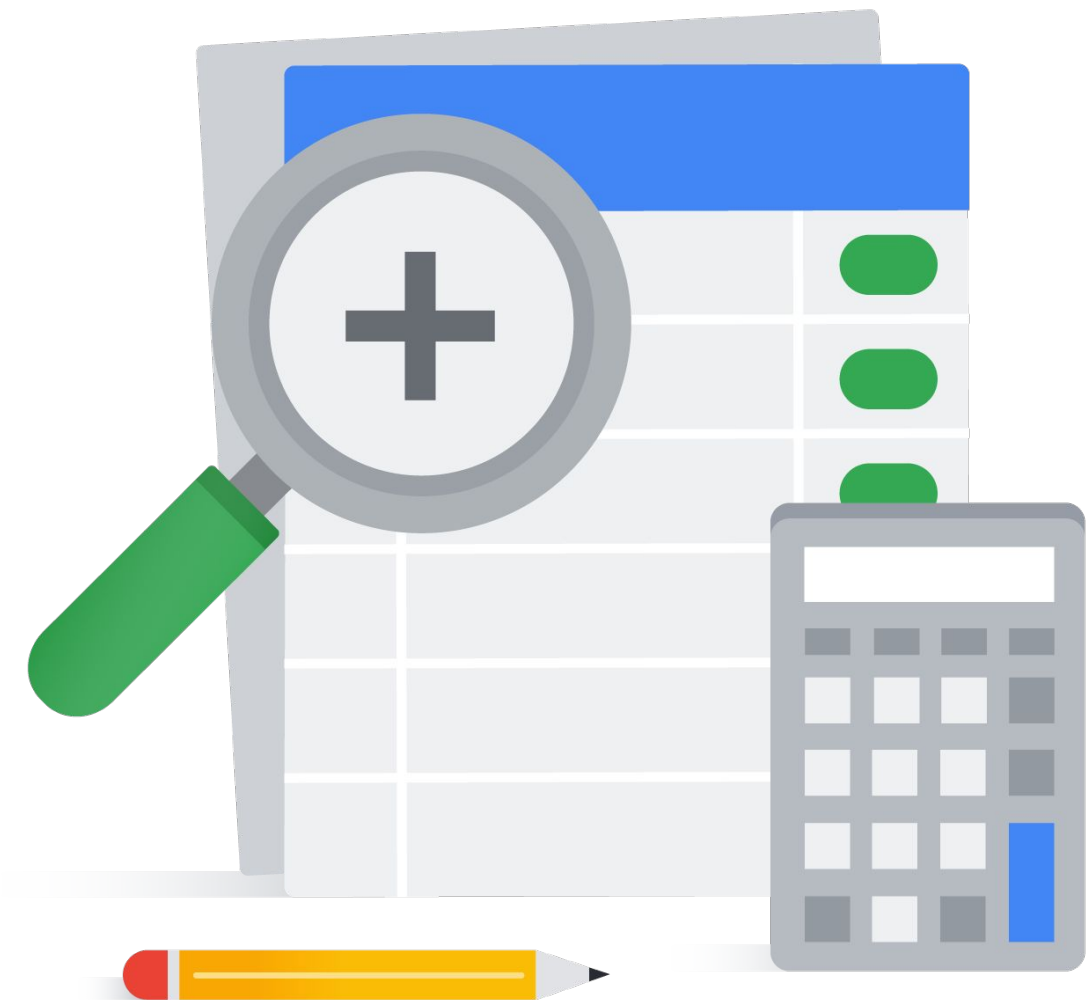
# DataFrame/PCollection conversion

```
with beam.Pipeline() as p:  
    pc1 = ...  
    pc2 = ...  
  
    df1 = to_dataframe(pc1)  
    df2 = to_dataframe(pc2)  
    ...  
    result = ...  
  
    result_pc = to_pcollection(result)  
  
    result_pc | beam.WriteToText(...)
```

---

# Differences from standard Pandas

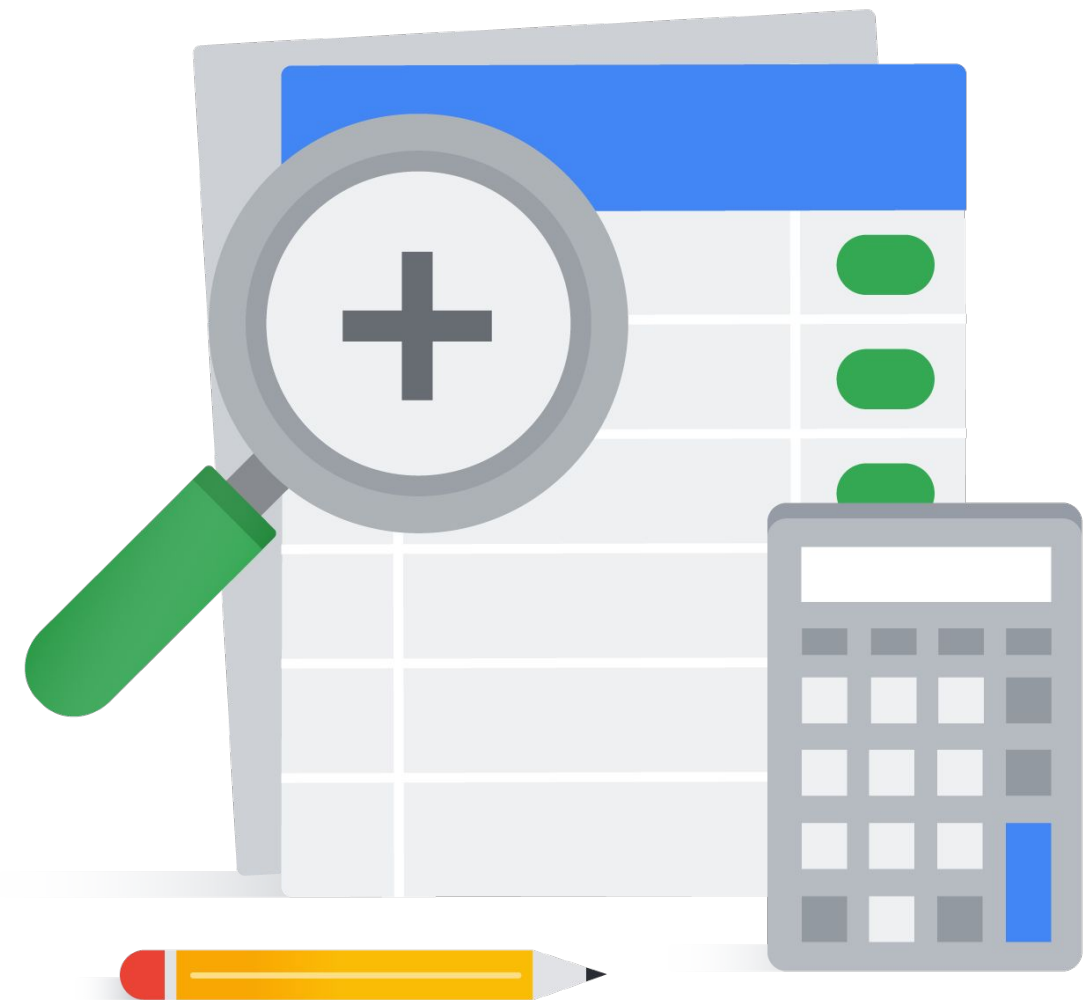
- Operations are deferred, and the result of a given operation may not be available for control flow or interactive visualizations. For example, you can compute a sum, but you can't branch on the result.



---

# Differences from standard Pandas

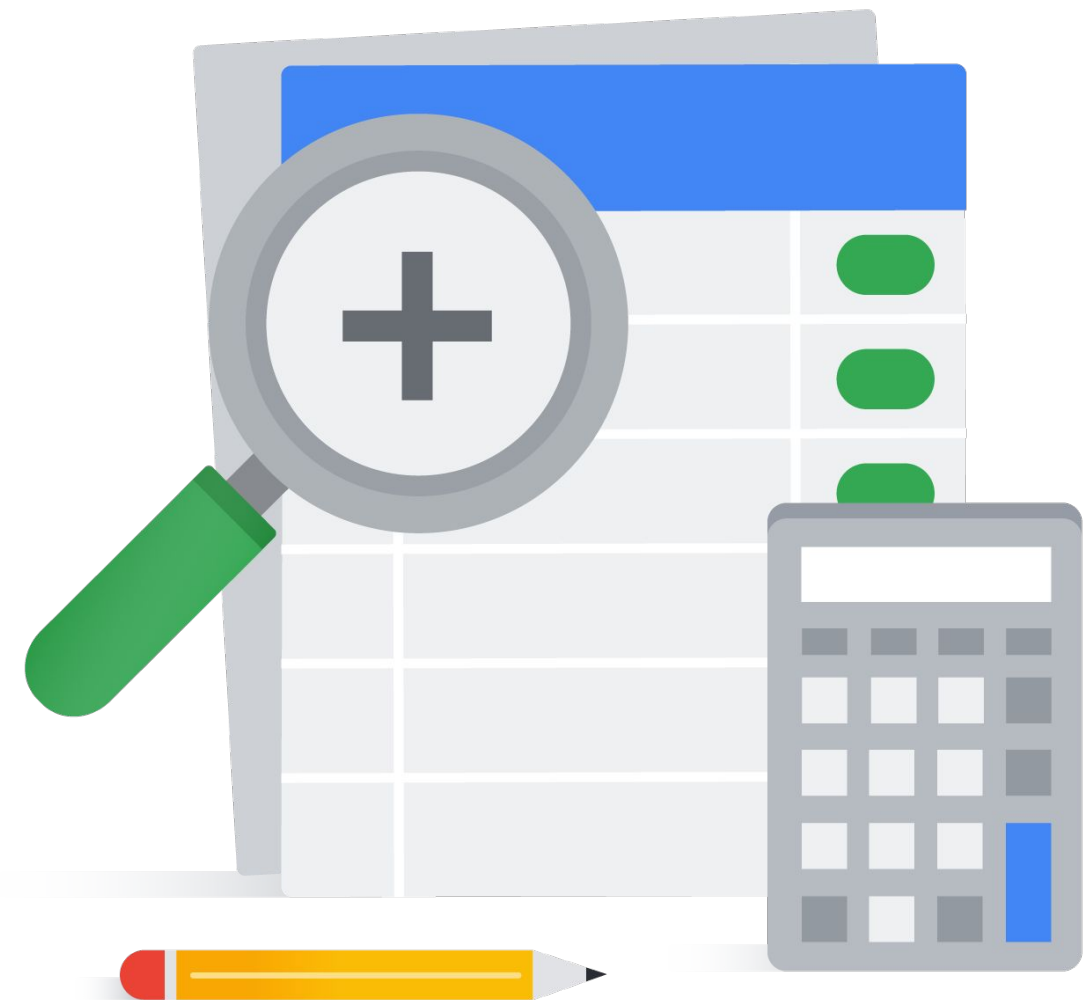
- Operations are deferred, and the result of a given operation may not be available for control flow or interactive visualizations. For example, you can compute a sum, but you can't branch on the result.
- Result columns must be computable without access to the data. For example, you can't use transpose.



---

# Differences from standard Pandas

- Operations are deferred, and the result of a given operation may not be available for control flow or interactive visualizations. For example, you can compute a sum, but you can't branch on the result.
- Result columns must be computable without access to the data. For example, you can't use transpose.
- PCollections in Beam are inherently unordered, so Pandas operations that are sensitive to the ordering of rows are unsupported. For example, head and tail are not supported.



---

## Use case: Count words

```
words = (  
    lines  
    | 'Split' >> beam.FlatMap(  
        lambda line: re.findall(r'[\w]+', line)).with_output_types(str)  
    # Map to Row objects to generate a schema suitable for conversion  
    # to a dataframe.  
    | 'ToRows' >> beam.Map(lambda word: beam.Row(word=word)))  
  
df = to_dataframe(words)  
df['count'] = 1  
counted = df.groupby('word').sum()  
counted.to_csv(known_args.output)
```

---

## Use case: Count words

```
# Deferred DataFrames can also be converted back to schema'd PCollections
counted_pc = to_pcollection(counted, include_indexes=True)

# Print out every word that occurred >50 times
_ = (
    counted_pc
    | beam.Filter(lambda row: row.count > 50)
    | beam.Map(lambda row: f'{row.word}: {row.count}')
    | beam.Map(print))
```