Best Practices

Ajay Kumar Yadav

Strategic Cloud Engineer

Google Cloud

Agenda

Course Intro

Beam Concepts Review

Windows, Watermarks, and Triggers

Sources and Sinks

Schemas

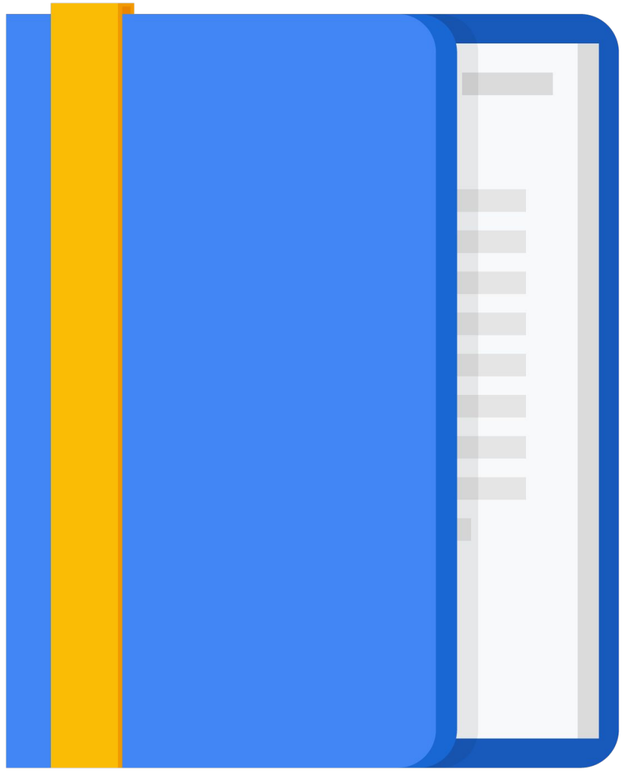
State and Timers

Best Practices

SQL and DataFrames

Beam Notebooks

Summary



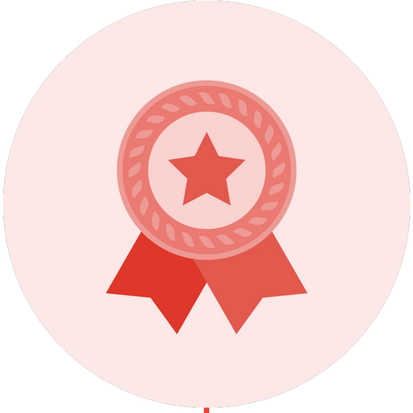
Best Practices

Agenda

- Schemas
- Handling erroneous data
- Error handling
- AutoValue code generator
- JSON data handling
- Utilize DoFn lifecycle
- Pipeline optimizations

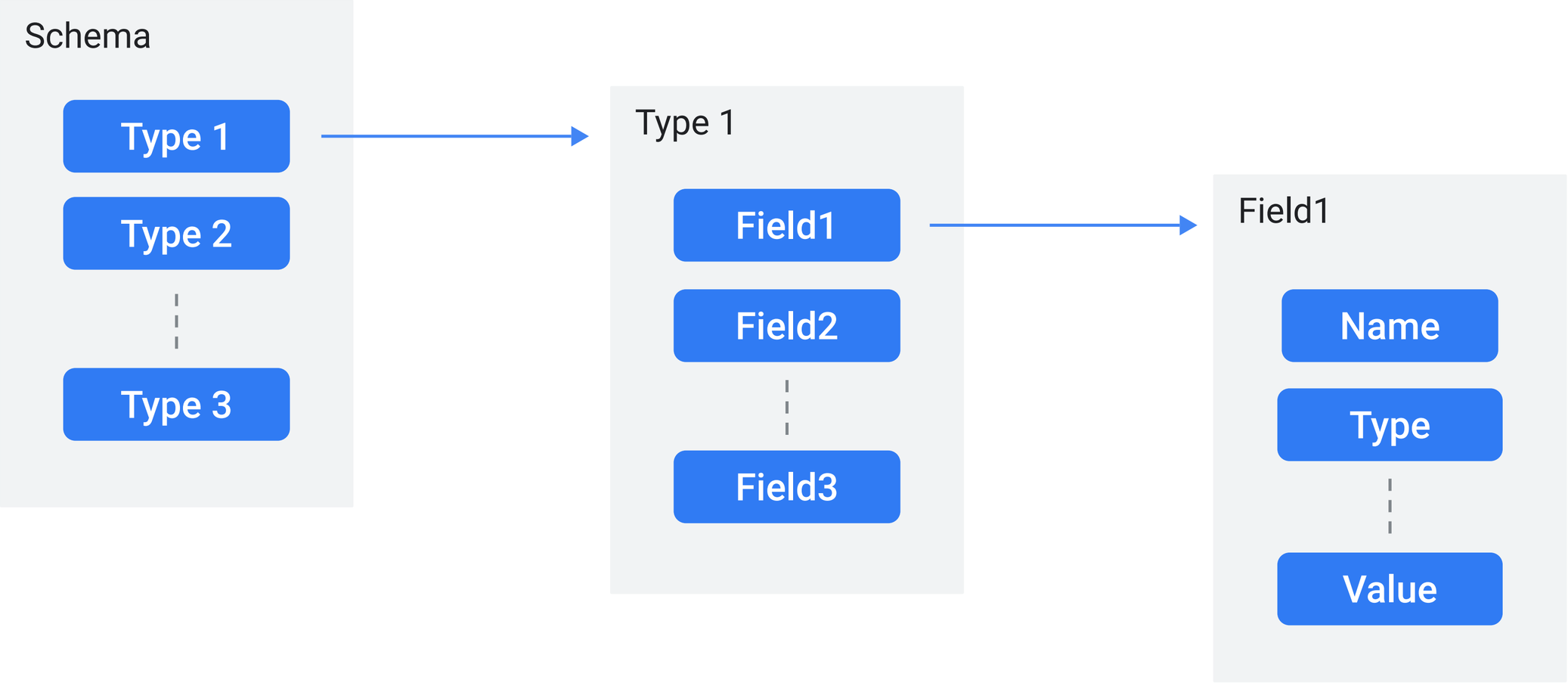
Best Practices

Agenda



- Schemas
- Handling erroneous data
- Error handling
- AutoValue code generator
- JSON data handling
- Utilize DoFn lifecycle
- Pipeline optimizations

Schemas


Schema
Type 1
Type 2
Type 3
Type 1
Field1
Field2
Field3
Field1
Name
Type
Value

Java

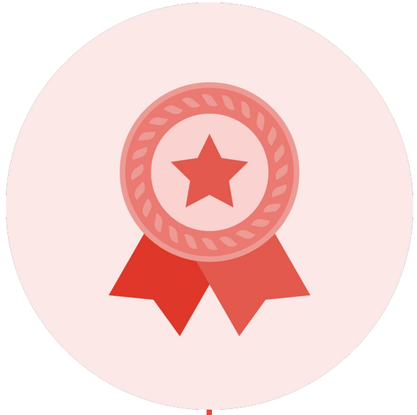
```
@DefaultSchema(JavaBeanSchema.class)
public class Purchase {
  public String getUserId();  // Returns the id of the user who made the purchase.
  public long getItemId();  // Returns the identifier of the item that was purchased.
  public ShippingAddress getShippingAddress();  // Returns the shipping address, a nested type.
  public long getCostCents();  // Returns the cost of the item.
  public List<Transaction> getTransactions();  // Returns the transactions that paid for this purchase
(returns a list, since the purchase might be spread out over multiple credit cards).
  @SchemaCreate
  public Purchase(String userId, long itemId, ShippingAddress shippingAddress, long costCents,
                  List<Transaction> transactions) {
      ... }}
```

Python

```
class Purchase(typing.NamedTuple):
  user_id: str  # The id of the user who made the purchase.
  item_id: int  # The identifier of the item that was purchased.
  shipping_address: ShippingAddress  # The shipping address, a nested type.
  cost_cents: int  # The cost of the item
  transactions: typing.Sequence[Transaction]
```

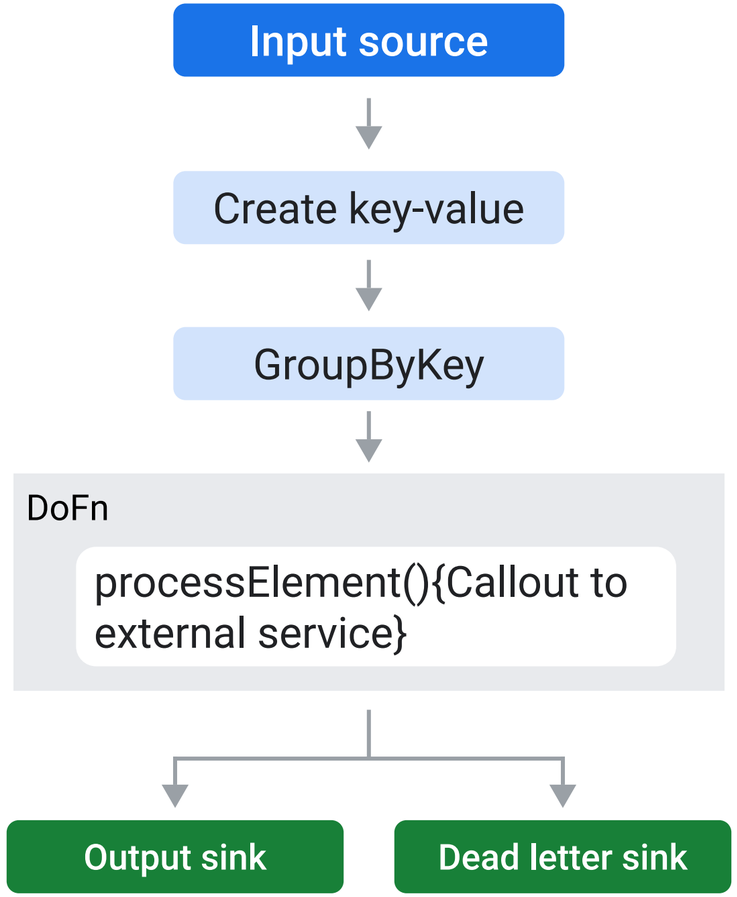
Best Practices

Agenda



- Schemas
- Handling erroneous data
- Error handling
- AutoValue code generator
- JSON data handling
- Utilize DoFn lifecycle
- Pipeline optimizations

Handling erroneous data



Handling unprocessable data

```
final TupleTag successTag;
final TupleTag deadLetterTag;
PCollection input = /* ... */;

PCollectionTuple outputTuple = input.apply(ParDo.of(new DoFn(){
    @Override
    void processElement(ProcessContext ctxt) {
        try {
            c.output(process(c.element));
        } catch(MyException ex) {
            // Optional Logging at debug level
            c.sideOutPut(deadLetterTag, c.element);
        }
    }
})).writeOutPutTags(successTag, TupleTagList.of(deadLetterTag));

// Write dead letter elements to separate sink
outputTuple.get(deadLetterTag).apply(BigQuery.write(...));

// Process the successful element differently.
PCollection success = outputTuple.get(successTag);
```

Write erroneous records to dead-letter sink.

Next section is “handling un-processable or erroneous records in a pipeline”.

Best Practices

Agenda

- Schemas
- Handling unprocess-able data
- Error handling
- AutoValue code generator
- JSON data handling
- Utilize DoFn lifecycle
- Pipeline optimizations

Error handling

- Errors and exceptions are **part of** any data processing pipeline.

Error handling

- Errors and exceptions are **part of** any data processing pipeline.
- Within the DoFn, always use a **try-catch block** around activities like parsing data.

Error handling

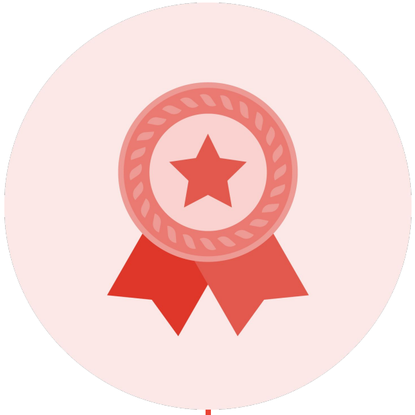
- Errors and exceptions are **part of** any data processing pipeline.
- Within the DoFn, always use a **try-catch block** around activities like parsing data.
- In the exception block, send the erroneous records to a separate sink, instead of just logging the issue.

Error handling

- Errors and exceptions are **part of** any data processing pipeline.
- Within the DoFn, always use a **try-catch block** around activities like parsing data.
- In the exception block, send the erroneous records to a separate sink, instead of just logging the issue.
- Use tuple tags to access multiple outputs from the PCollection.

Best Practices

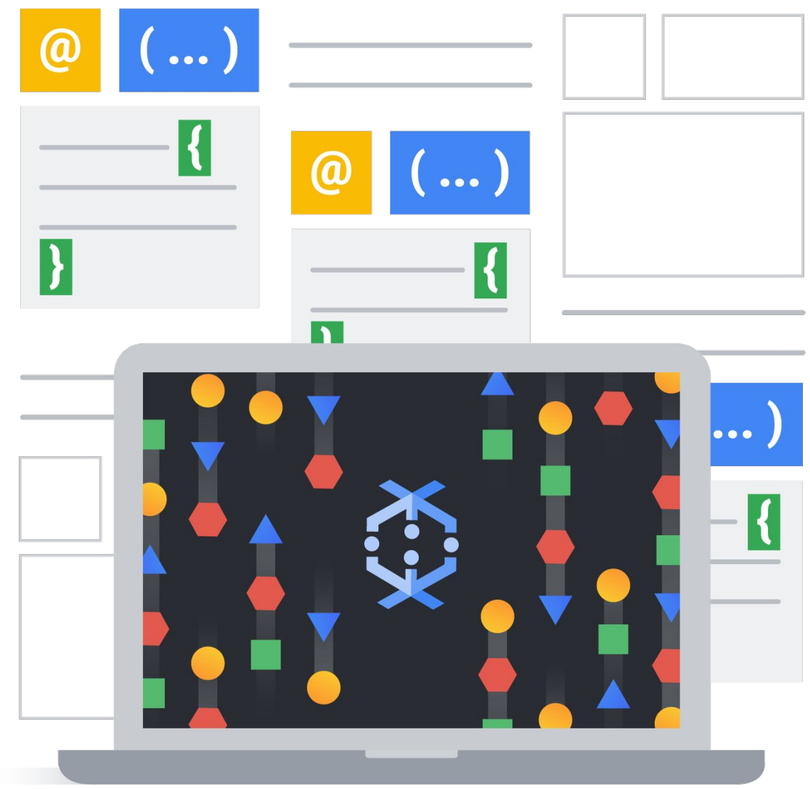
Agenda



- Schemas
- Handling erroneous data
- Error handling
- AutoValue code generator
- JSON data handling
- Utilize DoFn lifecycle
- Pipeline optimizations

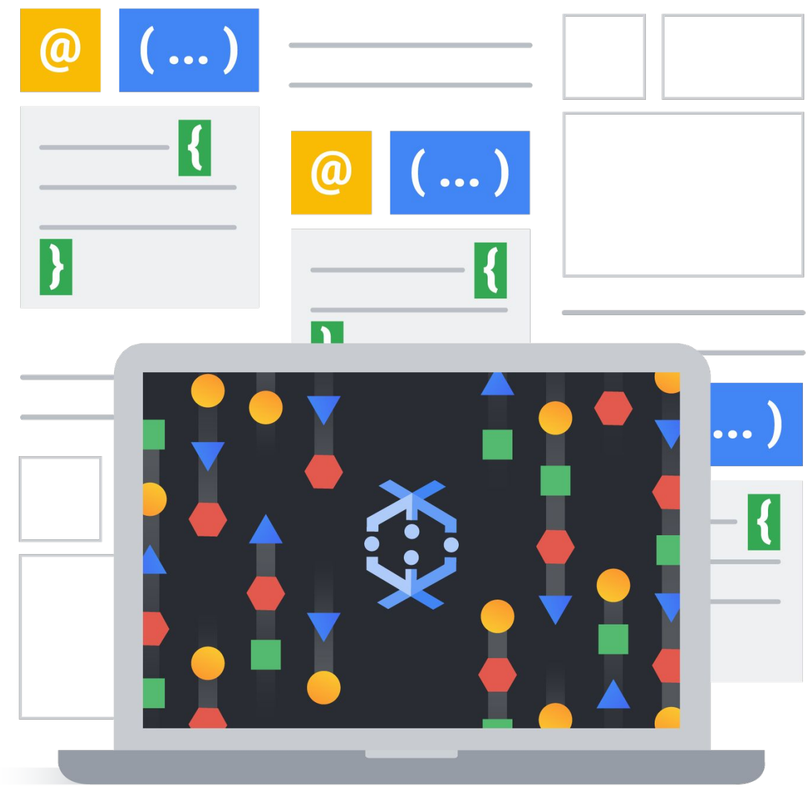
AutoValue code generator

- Although Apache Beam schemas are overall the best way to represent objects in a pipeline, there are still places where a POJO is needed while developing pipelines in Java.



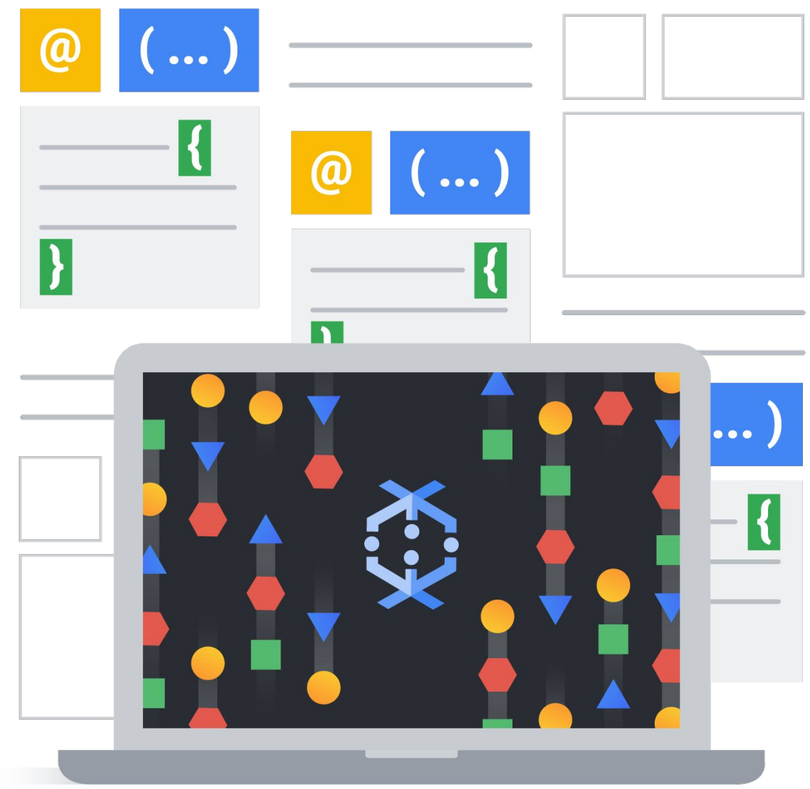
AutoValue code generator

- Although Apache Beam schemas are overall the best way to represent objects in a pipeline, there are still places where a POJO is needed while developing pipelines in Java.
- Use the AutoValue class builder to generate POJOs when not using Beam schemas.



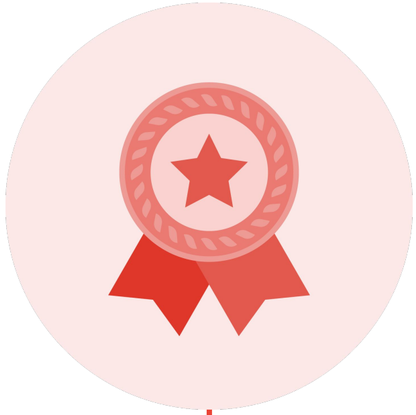
AutoValue code generator

- Although Apache Beam schemas are overall the best way to represent objects in a pipeline, there are still places where a POJO is needed while developing pipelines in Java.
- Use the AutoValue class builder to generate POJOs when not using Beam schemas.
- AutoValue can also be used in concert with Apache Beam if you add an @DefaultSchema(AutoValueSchema.class) annotation.



Best Practices

Agenda



- Schemas
- Handling erroneous data
- Error handling
- AutoValue code generator
- JSON data handling
- Utilize DoFn lifecycle
- Pipeline optimizations

Handling JSON data

```
PCollection<String> json = ...

PCollection<MyUserType>  = json
  .apply("Parse JSON to Beam Rows", JsonToRow.withSchema(expectedSchema))
  .apply("Convert to a user type with a compatible schema registered",
Convert.to(MyUserType.class))
```

Handling JSON data

```
PCollection<String> json = ...

PCollection<MyUserType>  = json
  .apply("Parse JSON to Beam Rows", JsonToRow.withSchema(expectedSchema))
  .apply("Convert to a user type with a compatible schema registered",
Convert.to(MyUserType.class))
```

- Use the Apache Beam built-in transform **JsonToRow** to convert JSON strings to POJOs.

Handling JSON data

```
PCollection<String> json = ...

PCollection<MyUserType>  = json
  .apply("Parse JSON to Beam Rows", JsonToRow.withSchema(expectedSchema))
  .apply("Convert to a user type with a compatible schema registered",
Convert.to(MyUserType.class))
```

- Use the Apache Beam built-in transform **JsonToRow** to convert JSON strings to POJOs.
- To convert JSON strings to POJOs using **AutoValue**, register a schema for using @DefaultSchema(AutoValueSchema.class) annotation, then use the Convert utility.

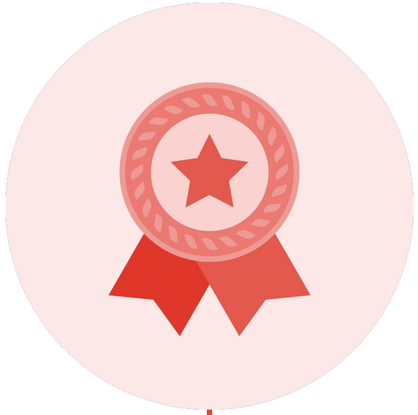
Handling JSON data

```
PCollection<String> json = ...

PCollection<MyUserType>  = json
  .apply("Parse JSON to Beam Rows", JsonToRow.withSchema(expectedSchema))
  .apply("Convert to a user type with a compatible schema registered",
Convert.to(MyUserType.class))
```

- Use the Apache Beam built-in transform **JsonToRow** to convert JSON strings to POJOs.
- To convert JSON strings to POJOs using **AutoValue**, register a schema for using @DefaultSchema(AutoValueSchema.class) annotation, then use the Convert utility
- Use Deadletter pattern for processing unsuccessful messages

Best Practices

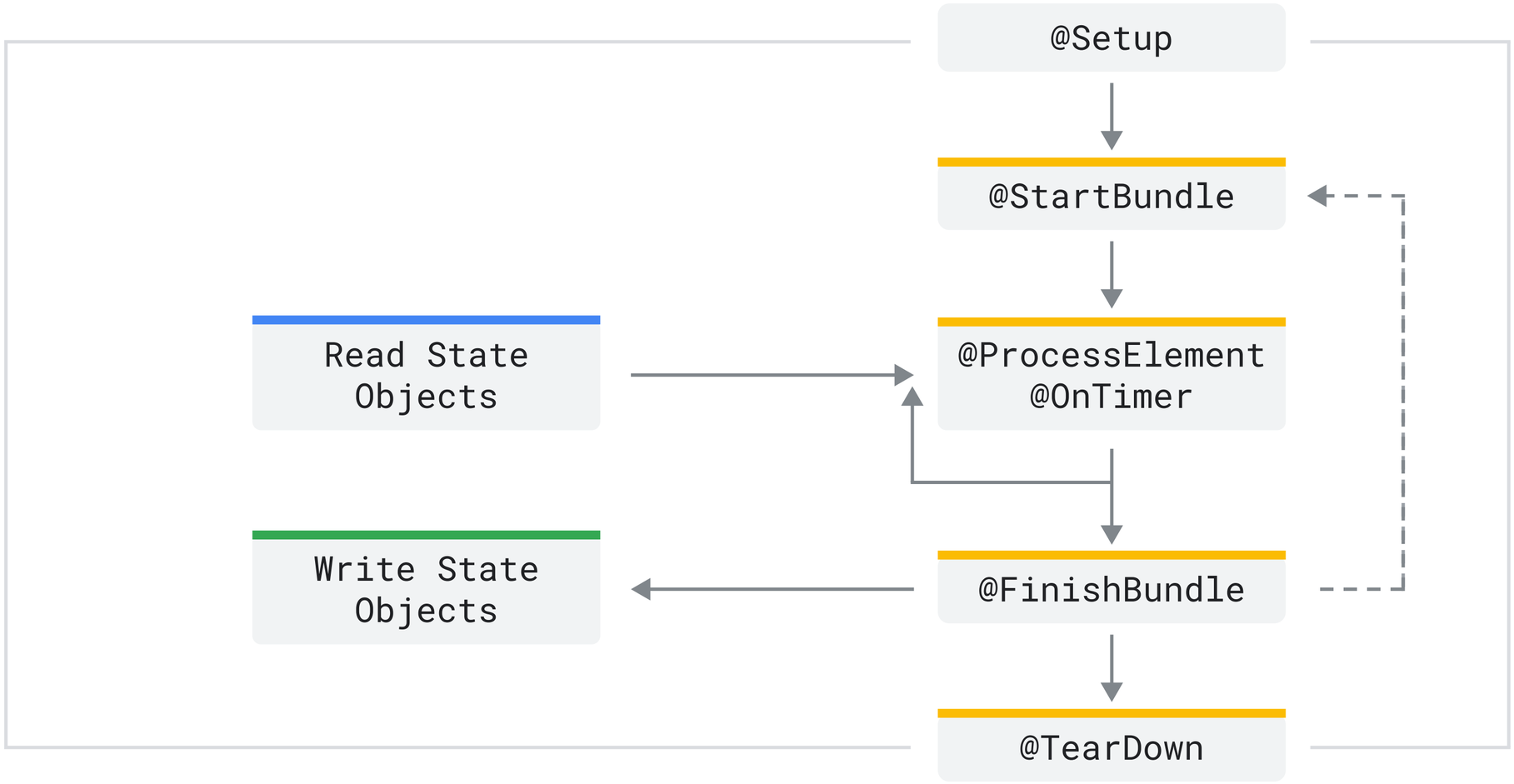
Agenda



- Schemas
- Handling erroneous data
- Error handling
- AutoValue code generator
- JSON data handling
- Utilize DoFn lifecycle
- Pipeline optimizations

DoFn for micro batching

DoFn Lifecycle



DoFn for micro batching

Java

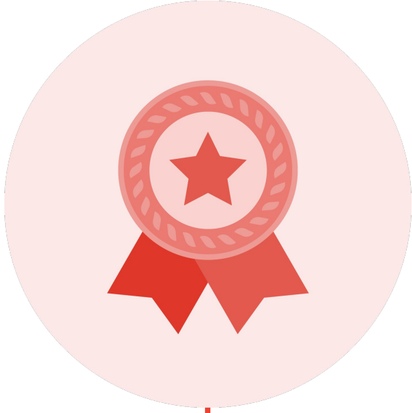
```
public class External extends DoFn {
@Override
public void startBundle(){
  Instantiate your external service client (Static if threadsafe)
}}

@Override
public void processElement(){
  Call out to external service
}}

@Override
public void finishBundle(){
  Shutdown your external service client if needed
}}
```

Python

```
class MyDoFn(beam.DoFn):
    def setup(self):
        pass
    def start_bundle(self):
        pass
    def process(self, element):
        pass
    def finish_bundle(self):
        pass
    def teardown(self):
        pass
```

Best Practices

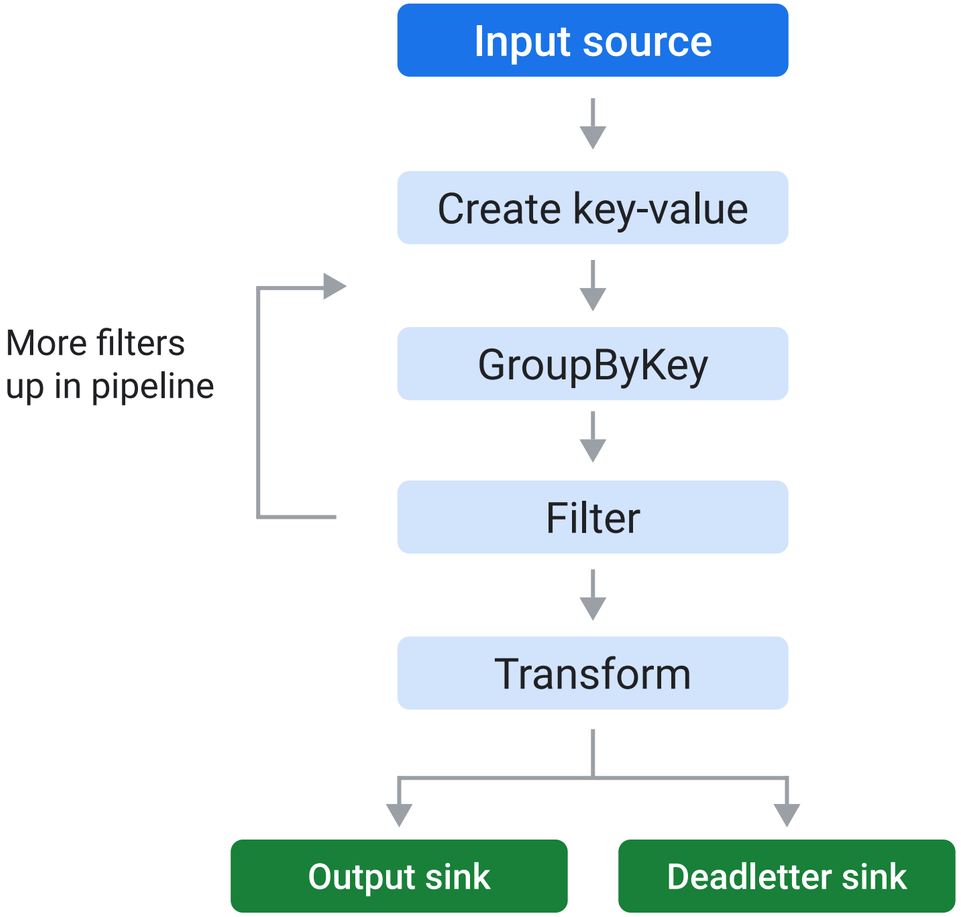
Agenda



- Schemas
- Handling erroneous data
- Error handling
- AutoValue code generator
- JSON data handling
- Utilize DoFn lifecycle
- Pipeline optimizations

Pipeline optimizations

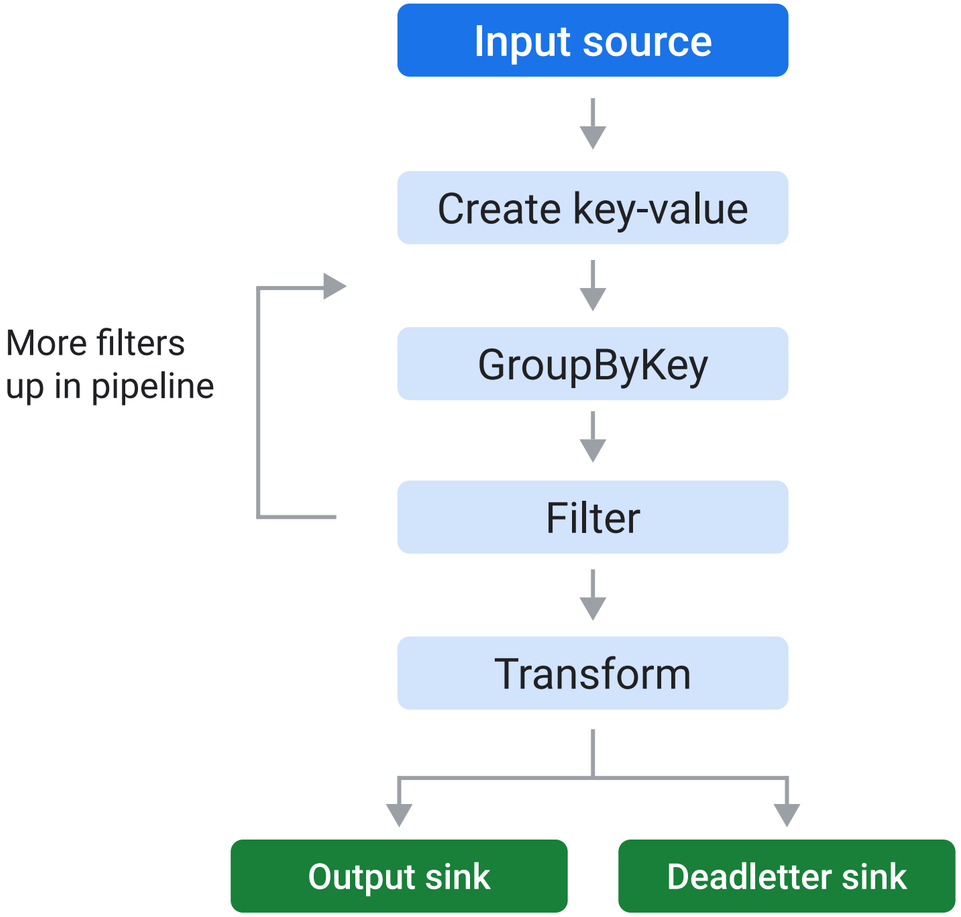
Filter data early



Pipeline optimizations

Filter data early

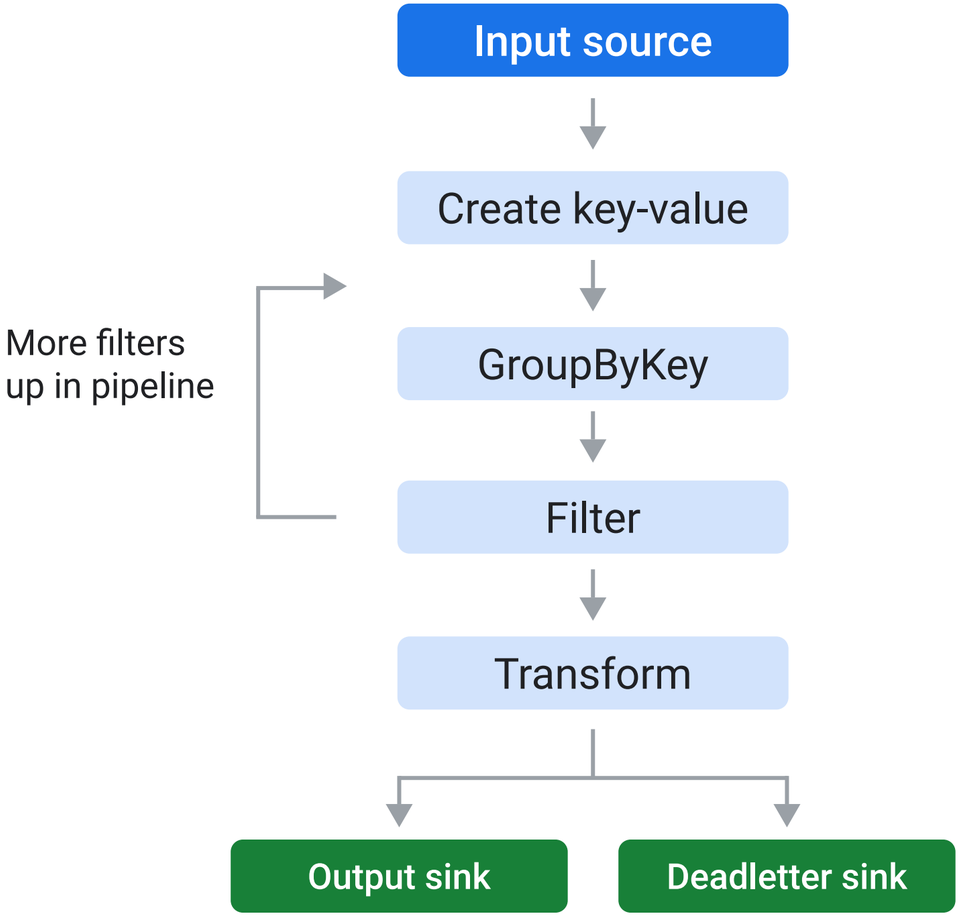
1. Filter data early in pipeline whenever possible.



Pipeline optimizations

Filter data early

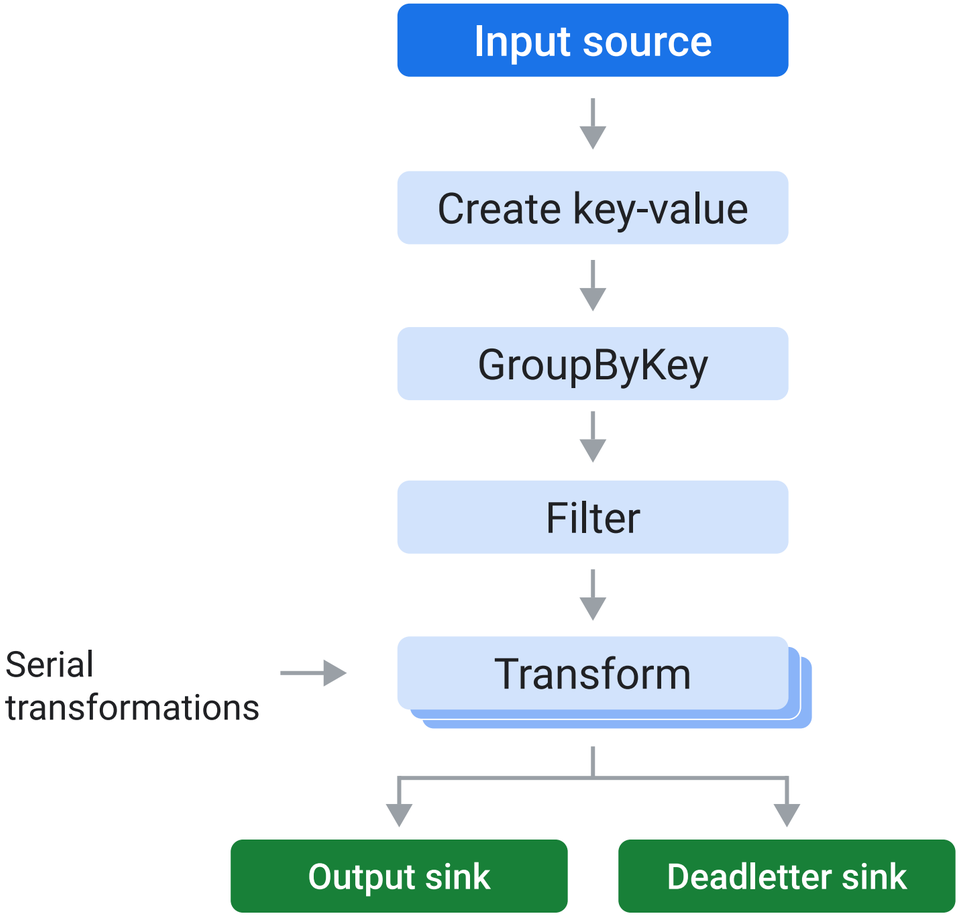
1. Filter data early in pipeline whenever possible.
2. Move any steps that reduce data volume up in your pipeline.



Pipeline optimization guidelines

Apply data transformations serially

1. Apply data transformations serially to let Dataflow optimize DAG.

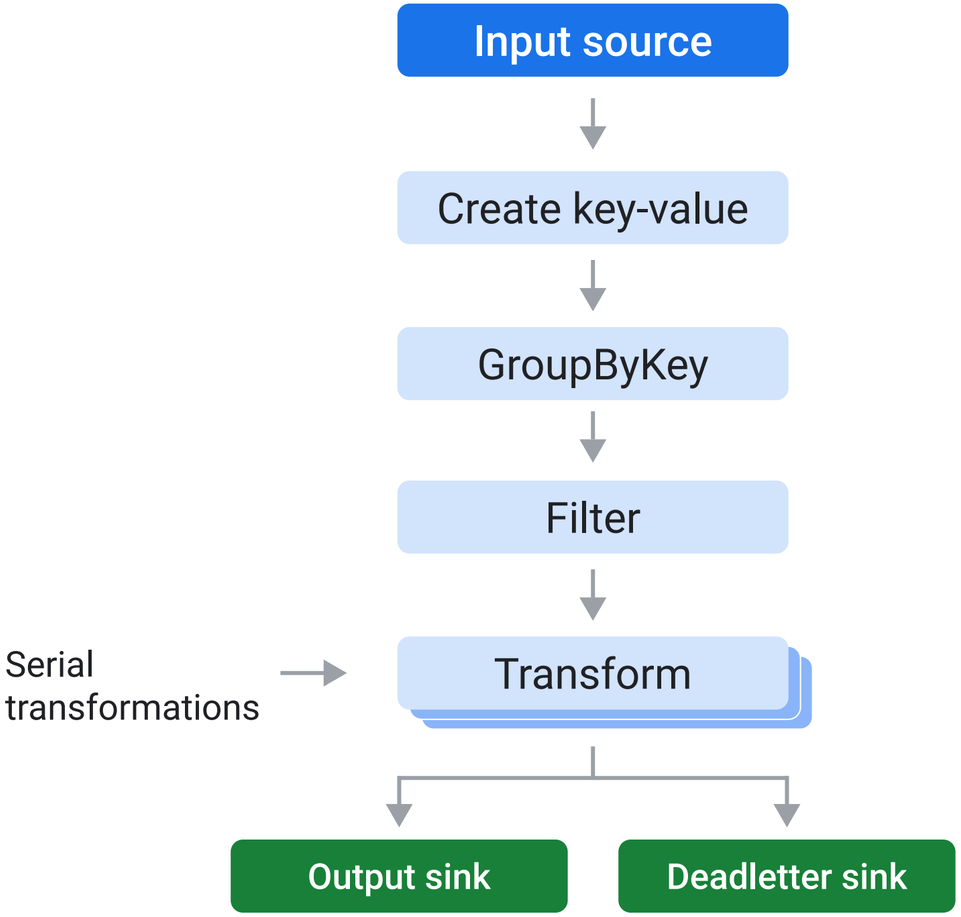


Pipeline optimization guidelines

Apply data transform serially

1. Apply data transformations serially to let Dataflow optimize DAG.

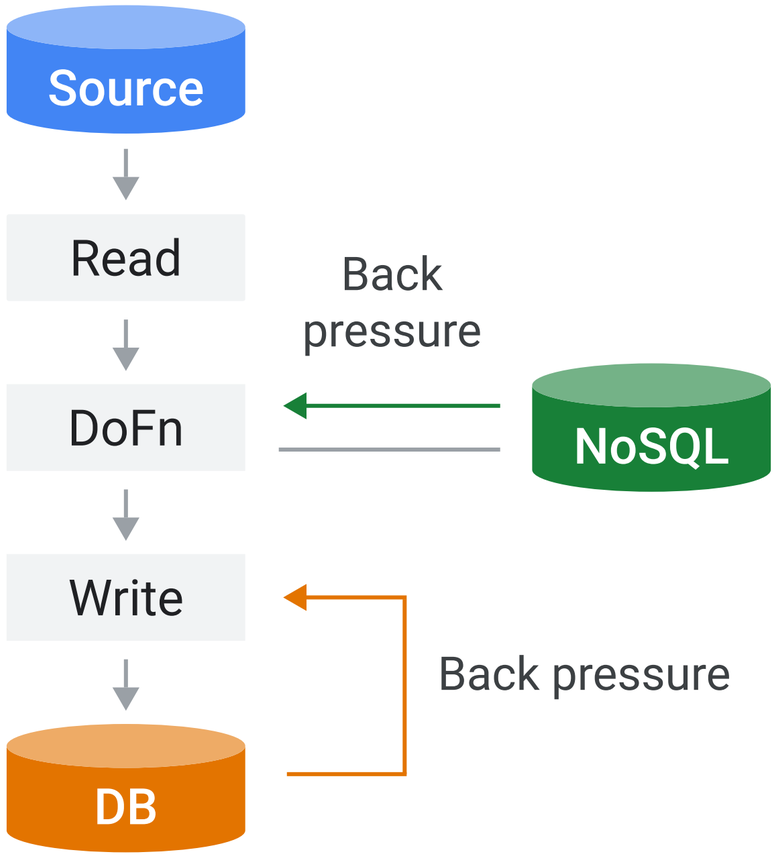
2. Transforms applied serially are good candidates for graph optimization because multiple steps can be fused in a single stage for execution in the same worker node.



Pipeline optimization guidelines

Handle back pressure from external systems

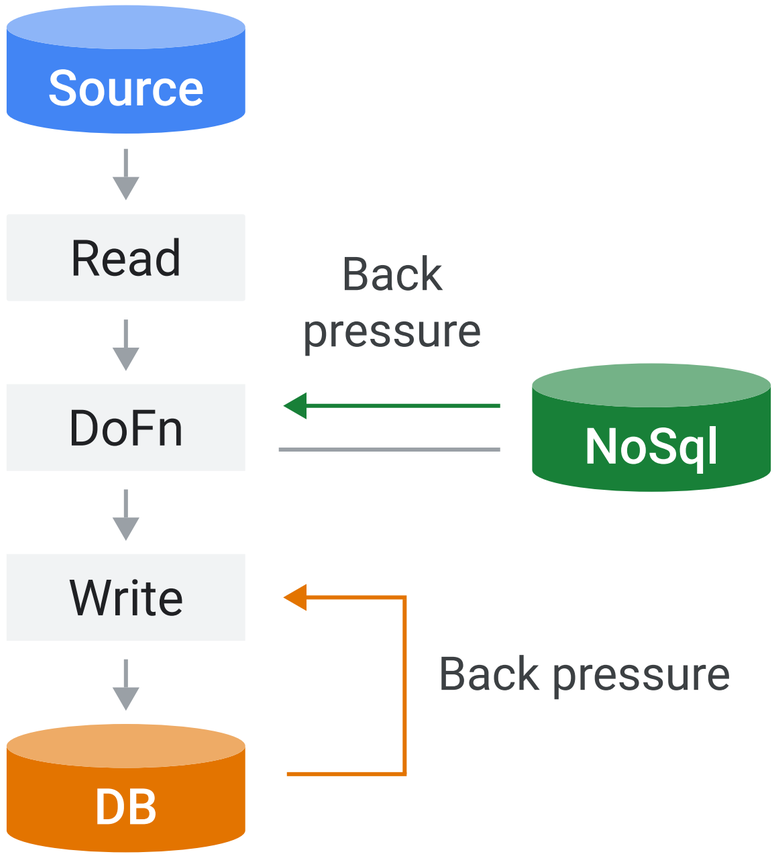
1. While working with external systems, look out for back pressure.



Pipeline optimization guidelines

Handle back pressure from external systems

1. While working with external systems, look out for back pressure.
2. Ensure external system are configured to handle peak volume.



Pipeline optimization guidelines

Handle back pressure from external systems

1. While working with external systems, look out for back pressure.
2. Ensure external system are configured to handle peak volume.
3. Enable autoscaling to downscale if workers are underutilized.