DATA+AI SUMMIT 2022

ÀLaSpark

Gousto's Recipe for Building Scalable PySpark Pipelines

Daniel Baron & Elena Martina

Data Engineering @ Gousto

ORGANIZED BY databricks

ÀLaSpark

Gousto's Recipe for Building Scalable PySpark Pipelines

Data Engineering @ Gousto

Daniel Baron
Data Engineer

Elena Martina
Data Engineer

gousto

Today's **Agenda**

In this session we'll cover:

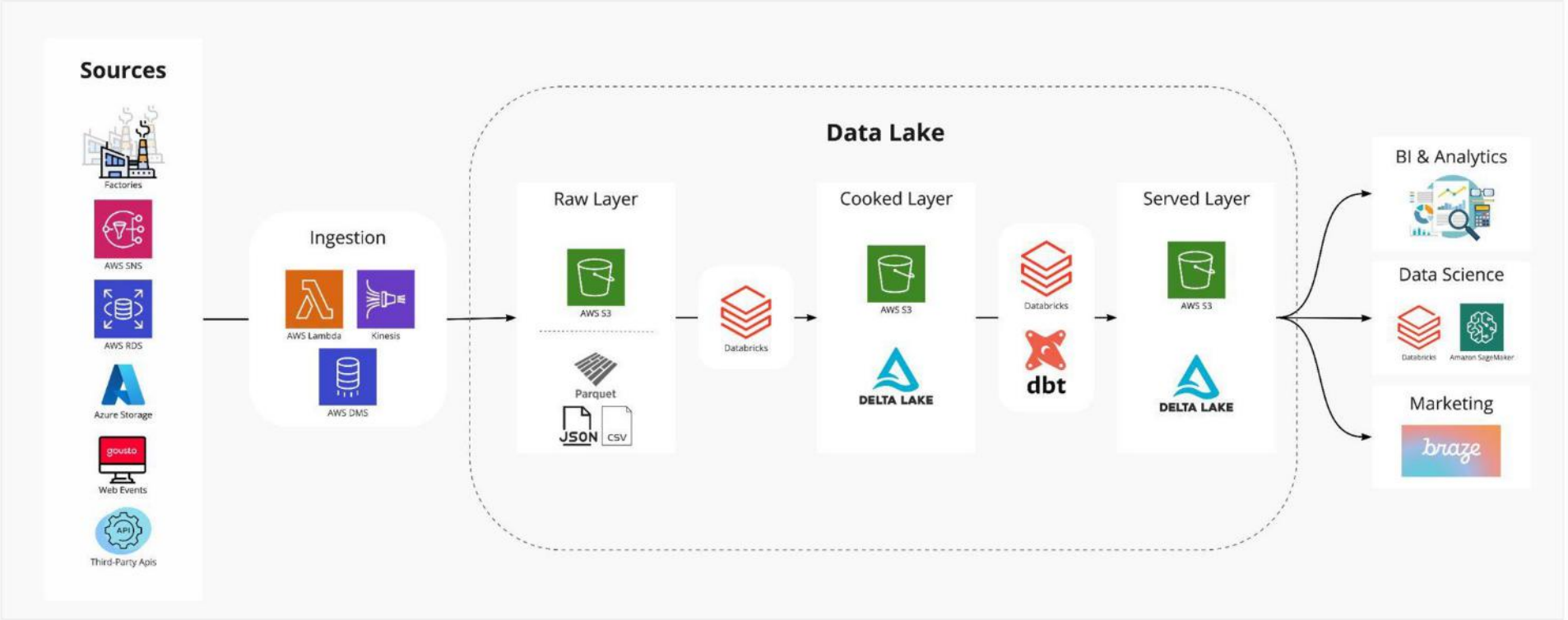
- A bit about Gousto and our Data Architecture
- Common issues with Spark pipelines
- What is ÀLaSpark?
- Live Demo
- ÀLaSpark building blocks: Deep Dive
- What's Next
- Q&A

About Gousto

“A data company that loves food!”

Our Data Lake **Architecture**

Quick Overview of our Stack



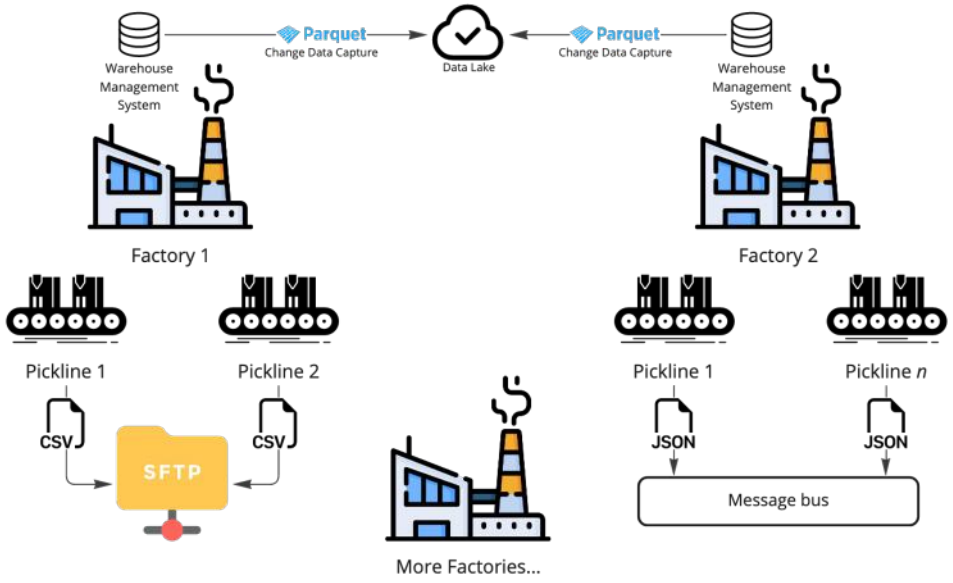
gousto

Example Use Case:
Factory Data

Example Use Case - **Factory Data**

Ingesting Factory Event Data into a Data Lake

- Several non standard factory systems
- Warehouse management systems + pickline systems
- RDBMS with CDC, SFTP and messaging systems
- Files stored in Parquet, CSV & JSON
- Data required in near real time (15 minute latency)
- More factories planned for the future



Example Use Case - **Factory Data**

What this Normally Looks Like...

- Duplicated code:
  - Bespoke implementation per pipeline
  - Bespoke metastore interactions
- Complex Spark configuration
- Horrible ‘Spark-Spaghetti’ code
- Lack of unit testing
- No naming standards
- Incoherent logging, monitoring and alerting
- Difficult to read

Example Use Case - **Factory Data**

How can we do better?

- Abstract repetitive configuration
- Create reusable components for repeated transformations
- Create a generic interface for reading and writing data
- Standardise naming conventions in the data lake & metastore
- Implement standard logging

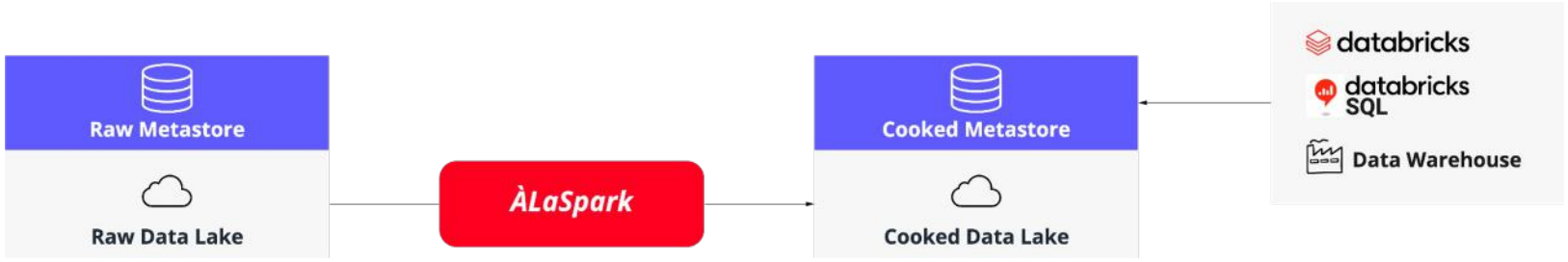
gousto

What is ÀLaSpark?

What is ÀLaSpark?

A Package for Building Data Pipelines at Scale

- A framework for easily building and managing data pipelines
- Reusable, tested components
- Abstracted Spark configuration
- Standardised logging
- A self serve package for non-data engineers



What is ÀLaSpark?

ÀLaSpark Building Blocks

Readers

Read data from a variety of sources and formats, in batch or streaming.

- Applied to a source table object

Processors

Perform data transformations like:

- Add/Drop Partitions
- Filter/Select columns
- Data deduplication
- Mask/Hash PII columns
- Join datasets

Writers

Write to a variety of targets, in a variety of formats.

- Automatically handle metastore interaction
- Can write to one or more tables

What is ÀLaSpark?

Building a pipeline with ÀLaSpark

```
for table in self.source_database.get_tables():

        processed_table = (

        table.set_pipeline_name("raw_to_cooked")
             .read(DeltaStreamReader)
             .apply_to_table_processor(SomeProcessor)
             .apply_to_table_processor(SomeOtherProcessor)

    )
```

What is ÀLaSpark?

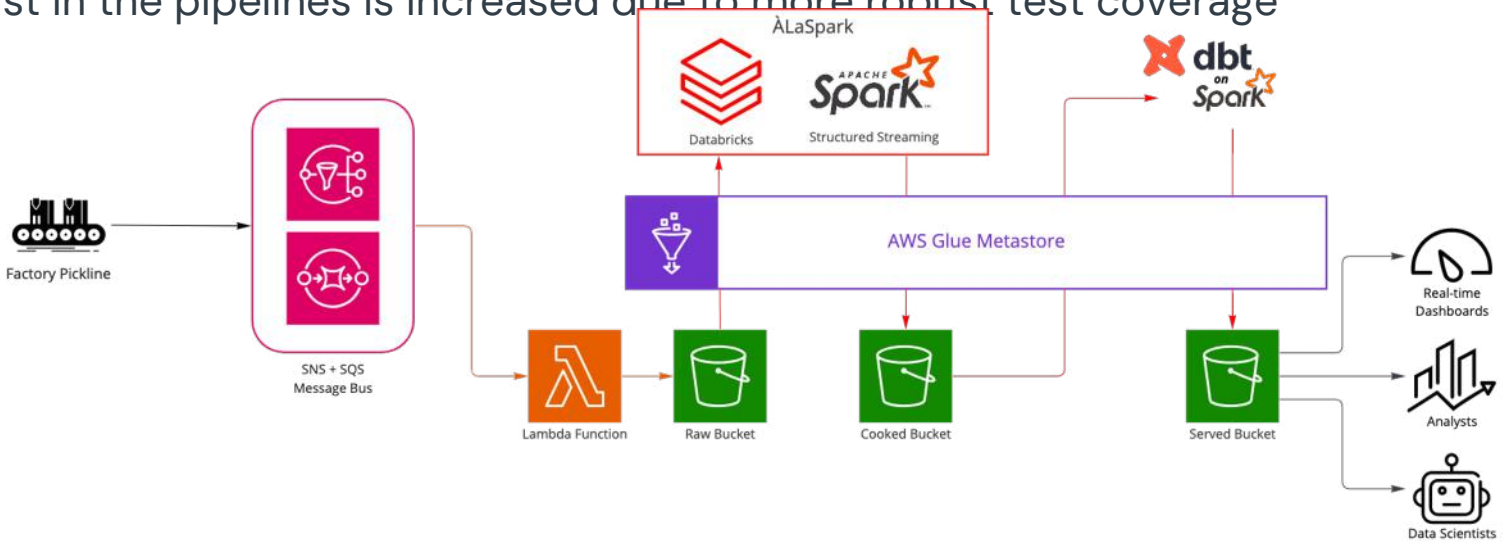
How to Build a Pipeline

```
        processed_table
    .write(
        DeltaStreamWriter,
        to_glue_database=self.target_database,
        output_mode=StreamingOutputMode.APPEND,
        merge_schema=MergeSchema.TRUE,
        trigger=SparkTrigger(
            MicrobatchProcessingTime(5, TimeUnit.MINUTES)
        )
    )
```

ÀLaSpark Use Case - **Factory Data**

Building factory pipelines

- Newer factories can be added more easily with simple configuration
- No bespoke metastore interactions are required for new pipelines
- Trust in the pipelines is increased due to more robust test coverage



ÀLaSpark Use Case - **Factory Data**

Building factory pipelines

- We've since added more factory systems to the lake with ÀLaSpark
- This includes:
  - A greater variety of event data
  - Change data
  - CSV data

gousto

Live Demo

gousto

Live Demo

What we're going to build:

- A pipeline to process raw **web event data** from an S3 bucket
- Raw data written as **CSV** microbatches
- The data schema is available in a **metastore**
- Provision the data for downstream processing/querying
- Data should be **stored efficiently** with an appropriate **partitioning scheme**

Cmd 1

Setup Parameters

```
schema_name = "databricks-summit" table_name = "fake_web_events_csv"  raw_bucket ...
```

Show cell

Cmd 2

Create Raw Metastore Database & Table

```
spark.sql(f"CREATE DATABASE IF NOT EXISTS {raw_database_name} LOCATION '{raw_dat ...
```

Show cell

Cmd 3

```
%sql
SELECT * FROM glue_raw__databricks_summit.fake_web_events_csv
```

Cmd 4

Install Goustospark

```
%pip install git+ssh://git@github.com/Gousto/goustospark-package.git@develop
```

Cmd 5

Job Config

Python

```
import os

os.environ["DEPLOY_ENV"] = "artichokes"
os.environ["DEPLOY_REGION"] = "eu-west-1"

config = f"""
deploy_env = "{{{{ env.DEPLOY_ENV }}}}"
schema_name = "{schema_name}"
pipeline_name = "demo_pipeline"

source_path = "@jinja {raw_bucket}/dan-b/{{{{ this.SCHEMA_NAME }}}}"
target_path = "@jinja {processed_bucket}/{{{{ this.SCHEMA_NAME }}}}"
bad_records_path = "@jinja {bad_records_bucket}/dan-b/{{{{ this.SCHEMA_NAME }}}}/{{{{ this.PIPELINE_NAME }}}}"
```

gousto

ÀLaSpark
Deep Dive

ÀLaSpark **Deep Dive**

Readers & Writers

- ÀLaSpark provides simple to use **readers** and **writers**.
- Supports a variety of formats in both **stream** and **batch**
  - Formats such as
    - Delta
    - JSON
    - CSV
    - Parquet
- Supports reading from **queues**
- Integrated with Databricks **Auto Loader**
  - This allows us to be **stream first** in almost all scenarios

ÀLaSpark **Deep Dive**

Sample Reader

```python
class DeltaStreamReader(BaseStreamReader):
        def run(self, **kwargs) -> ProcessedTable:
                table_location = self.table.location()
                df = (
                        self.spark.readStream.format("delta")
                        .options(**self.read_options(**kwargs))
                        .load(table_location)
                )

                return self._get_processed_table(df=df)


reader = DeltaStreamReader(source_table, pipeline_name, spark, config)
```

ÀLaSpark **Deep Dive**

Sample Writer

```
class DeltaStreamWriter(BaseStreamWriter):

        @log()
        def run(self, **kwargs) -> WrittenTable:
                streaming_query = self._streaming_query(**kwargs)
                written_table =
self._get_written_table(streaming_query=streaming_query)

                if not written_table.delta_exists():
                        written_table.delta_create()
                written_table.drop_and_create_table()

                return written_table
```

ÀLaSpark **Deep Dive**

Sample Writer Streaming Query

```
1  def _streaming_query(
2          output_mode: StreamingOutputMode,
3      processed_table: ProcessedTable,
4      save_location: str,
5      **kwargs,
6  ) -> StreamingQuery:
7          return (
8                  processed_table.df
9          .writeStream
10         .format("delta")
11         .options(**self.options(**options))
12         .partitionBy(processed_table.partitions)
13         .queryName(query_name)
14         .outputMode(outputMode.value)
15         .start(save_location)
16         )
```

ÀLaSpark **Deep Dive**

Sources & Sinks - the Metastore

- Goustospark provides a **table** implementation to use as a **source** and **sink**
- Most commonly this is backed by a **metastore** table (not always!)
- We can **read** from these tables and **write** to them
- **Adding** a new table simply means implementing **access** to **table metadata** (e.g. schema, partitions, etc.)
    - This can be done with Spark SQL (for Hive), Delta library, or some other custom method (e.g. schema registries)

ÀLaSpark **Deep Dive**

Processors

- We have two types of **processor**
  - ToTableProcessor (*1:1*)
  - ToManyTablesProcessor (*1:**)
- They apply some **transformation** to the underlying dataframe
  - Trivial transformations such as WithColumn, Filter, AddCurrentTimestamp
  - Complex transformations such as ExplodeColumns, DeduplicateMicrobatch
- Currently **over 35 different Processors**!

ÀLaSpark **Deep Dive**

Processors

Table-level Operations	Add Partition	Join Dataframe	Drop Duplicates	Add Watermark
	Drop Partition	Union Dataframe	Filter Rows	Select Columns
Column-level Operations	With Column	Add Current Timestamp	Add Source Filename	
	Drop Columns	Add Source Path	Alter Data Type	
PII Utilities	Mask Column	Null Column		
	Hash Column	Replace Character		
Array Utilities	Flatten Schema	Parse Json		
	Explode Column	Relationalize		

ÀLaSpark **Deep Dive**

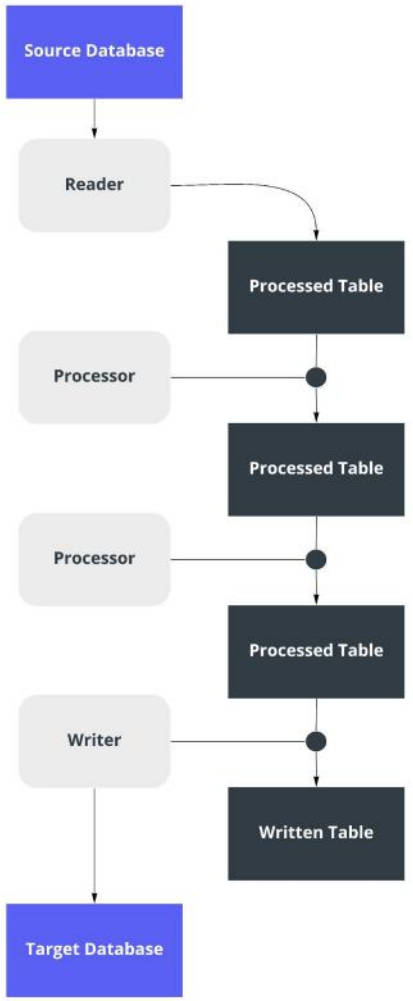
Processors - Select Columns Processor

```python
class SelectColumnsProcessor(BaseToTableProcessor):

        def run(self, columns_to_select: List[str], **kwargs):
                df = self.processed_table.df
                df = df.select(*columns_to_select)
                partitions = self.processed_table.partitions

                partitions = [
                        partition for partition in partitions if partition in
columns_to_select
                ]

                self.processed_table.partitions = partition
                self.processed_table.df = df

                return self.processed_table
```

ÀLaSpark **Deep Dive**

Processed Table

- A Processed Table is an **abstraction** of a table
  - It combines a table's **dataframe** and **metadata**
- Provides methods to apply processors & writers
- Tracks processors applied to some table for **lineage**



ÀLaSpark **Deep Dive**

Processed Table

```python
class ProcessedTable(BaseTable):

  def apply_to_table_processor(self, processor, **kwargs) -> ProcessedTable:
        processor = to_table_processor(self)
        return processor.run(**kwargs)


  def apply_to_many_table_processor(self, processor, **kwargs) -> ProcessedTable:
        processor = to_many_tables_processor(self)
        return processor.run(**kwargs)


  def write(self, writer, database, **kwargs) -> WrittenTable:
        return writer(self, database, self.spark, self.config).run(**kwargs)


```

What’s next on the menu?

What's next on the menu?

1. Open Sourcing **ÀLaSpark**!
2. Schema Registries
3. Greater Lineage support

And that’s a wrap

Any Questions?

Thank you