Handling Edge Cases in a Modern Data Architecture

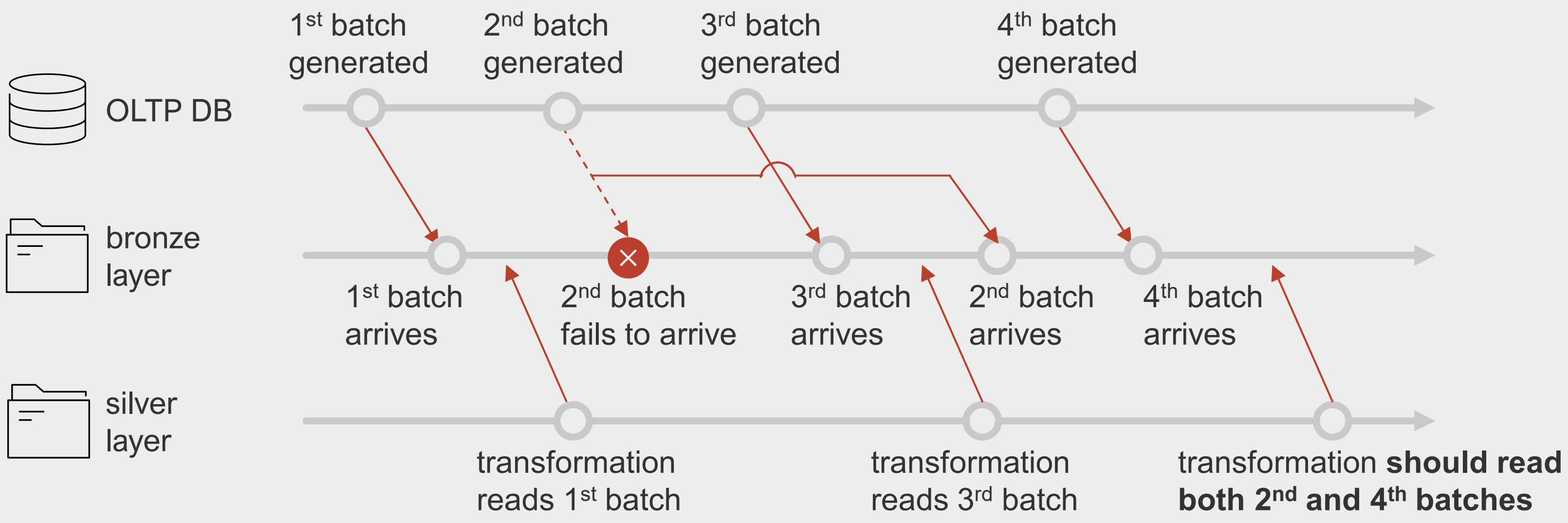
by Nicola Orecchini, 19/07/2025

When you design the architecture of a data platform, you may start from requirements that cover 99.9% of use-cases.

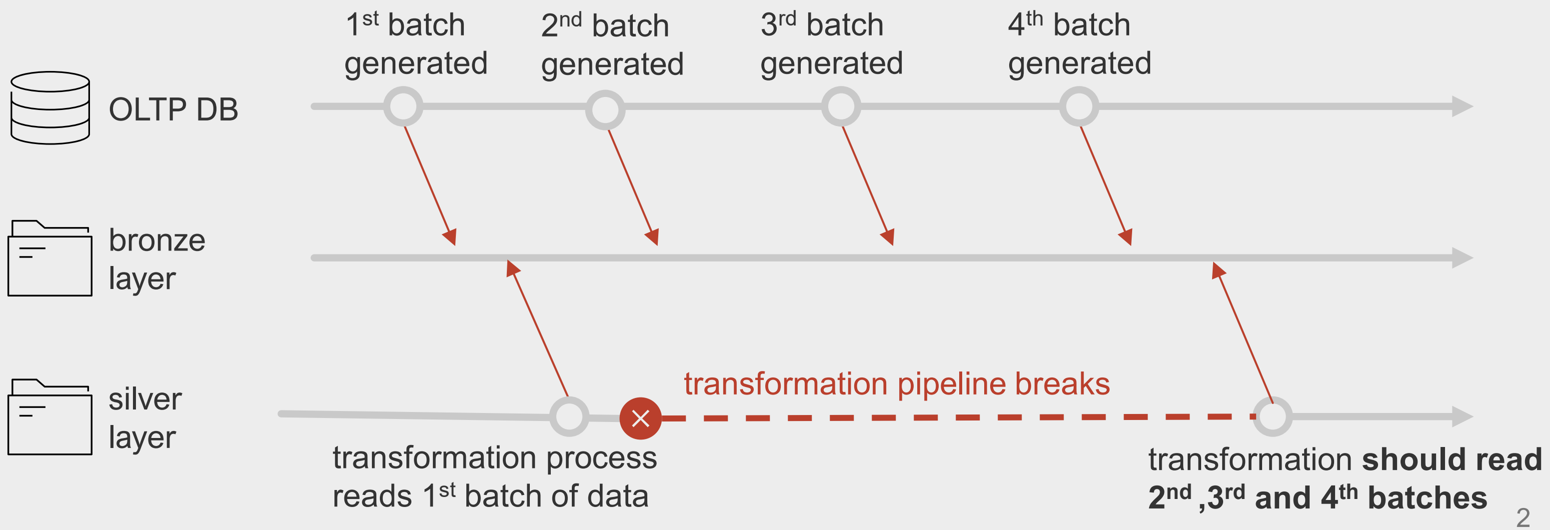
That 0.01% of cases though, ***edge or corner cases***, must still be considered and carefully evaluated, as they could lead to impacts on architectural decisions.

Today, we cover **two edge cases** that we dealt with when designing a data platform architecture

1 Late arriving data



2 Accumulated data



Agenda

1. A Data Platform architecure
2. Edge Cases
   - Late Arriving Data
   - Accumulated Data
3. Takeaways

Annex

Storage Layer

ETL Pipeline

Late Arriving Data Example

Accumulated Data Example

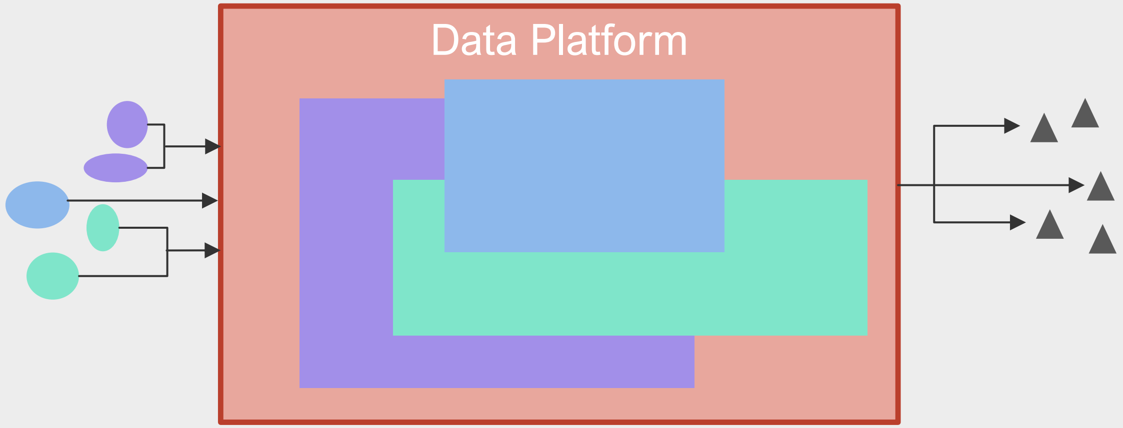
Sources

Agenda

Two architecture principles have shaped our Cloud Data Platform architececure

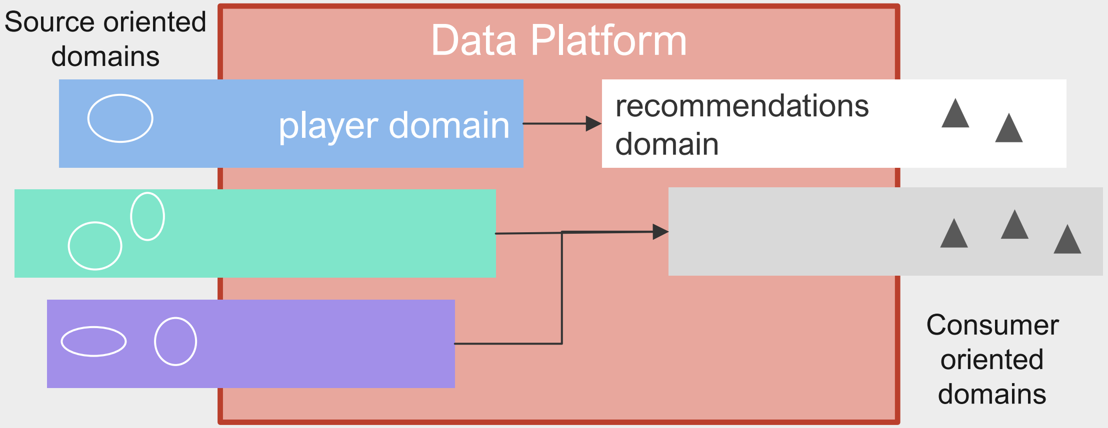
1. Data mesh
2. Medallion Architecture

Data Mesh’s goal is to decentralize the monolithic data platform architecture by creating a domain-oriented data platform



Monolithic architecture – Push & ingest model

- each source operational system (managed by a domain team) feeds data organized by domains into the platform, not caring where these data will be consumed
- the centrally owned data platform ingests the data
- after ingestion, the concept of domain is lost, and one single platform team is responsible of providing data from the platform to consumers
- the architectural quantum is represented by a stage of the pipeline (e.g., ingest/process/serve)



Data mesh architecture – Serve & pull model

- each domain owns, hosts and serves their datasets for access by any team downstream
- domains are subdivided into source-oriented and consumer-oriented: the latter will take input from source-oriented domains’ datasets
- the physical location where the datasets actually reside could still be centralized (e.g., Amazon S3 buckets), but datasets ownership remains within the domain generating them
- the architectural quantum is represented by a domain

Legend

data sources | data domains | data consumers | data flow – each arrow means a different team owns it | different colors indicate different business domains

Sources: [1], [2]

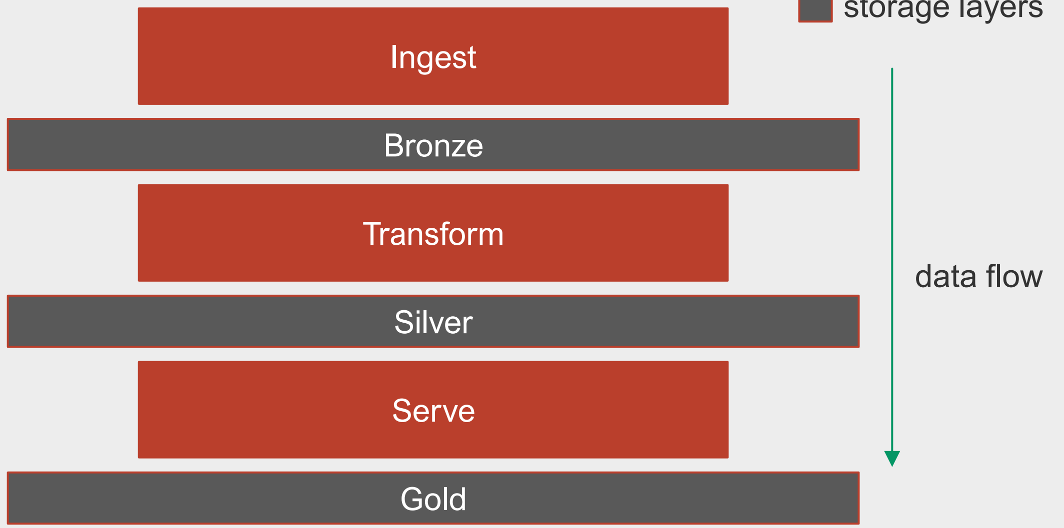
The medallion architecture is a way to decompose the old ‘monolithic ETL script’ model in a layered, modular approach

Traditional ETL pipeline

```sql
--Extract and transform sales data for reporting
WITH
raw_sales AS (SELECT…
FROM …
WHERE …),
enriched_sales AS (SELECT…
FROM …
LEFT JOIN … ON …
WHERE …),
--Final select (used for reporting)
SELECT…
FROM aggregated_sales a
LEFT JOIN region_mapping r
ON …
WHERE…
ORDER BY …;
```

- A single **monolithic** SQL script or notebook
- Filled with **nested subqueries** and **views**
- Hard to debug, test, or scale
- Everything runs in a **tight sequence**, one giant transformation pipeline
- **Difficult to reuse** or reason about **intermediate steps**

Medallion Architecture



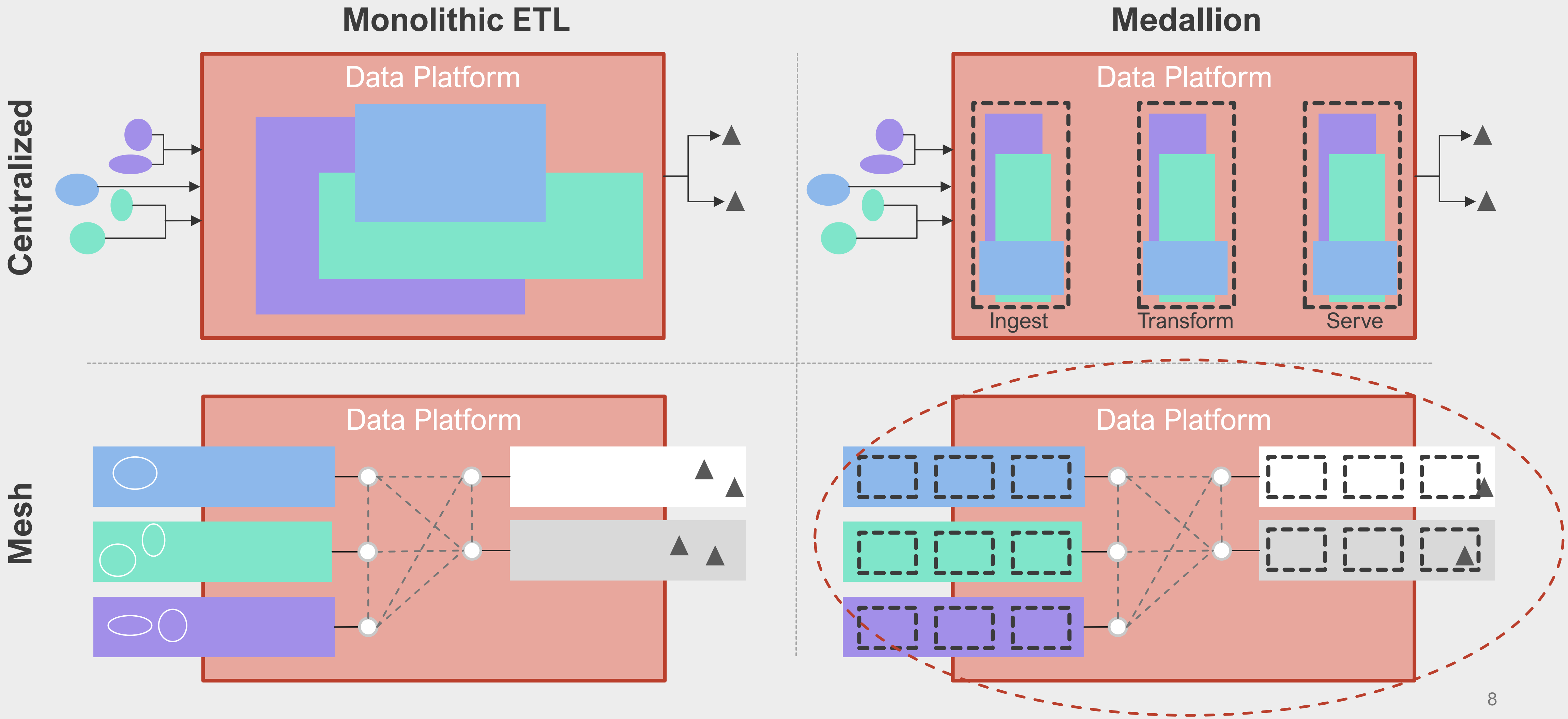
ETL is broken down into **3 standalone, idempotent processing jobs**:

1. **Ingest** → writes to bronze/
2. **Transform** → reads from bronze/, writes to silver/
3. **Serve** → reads from silver/, writes to gold/

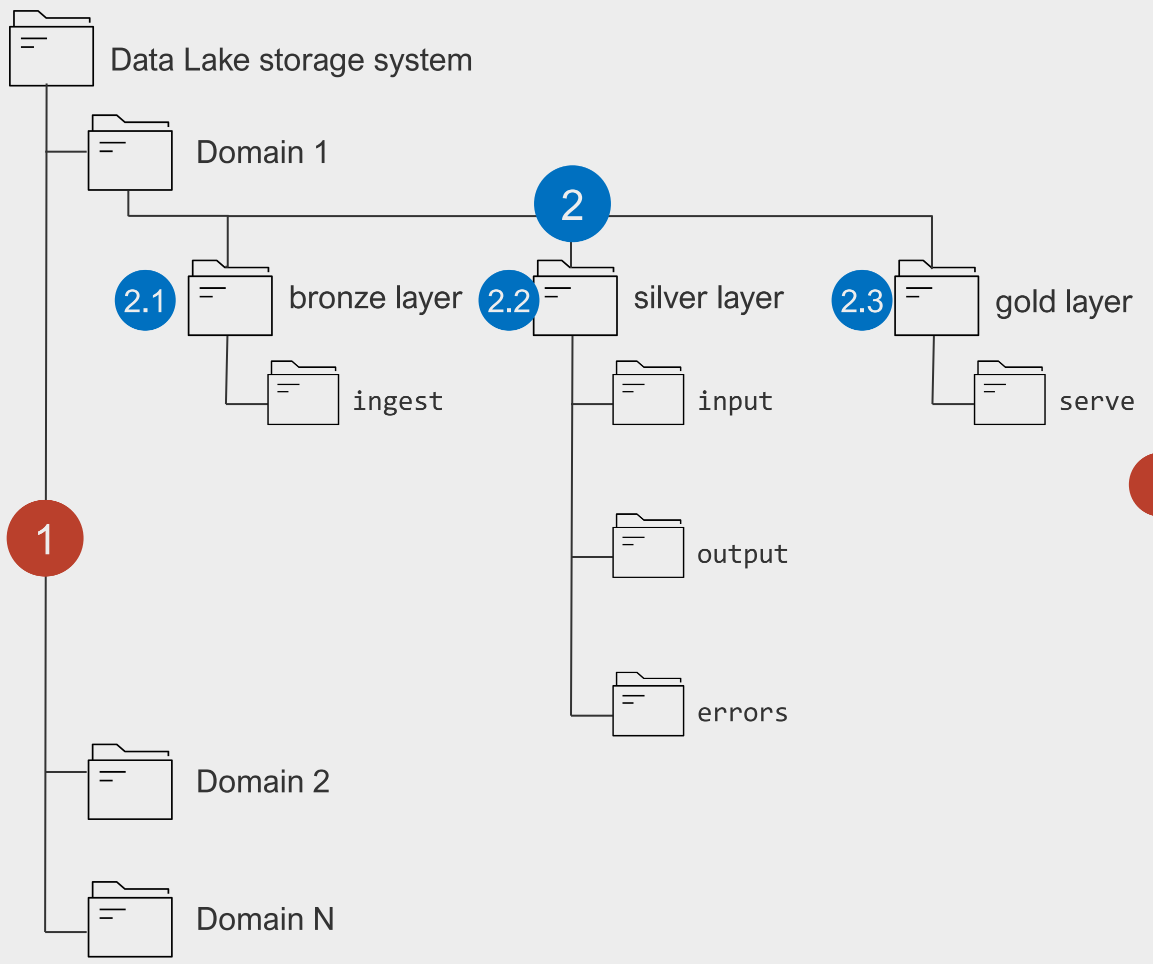
Each stage is:

- **Modular** (can be tested and deployed separately)
- **Idempotent** (can be rerun without side effects)
- **Folder-based** (uses file/table boundaries like bronze/, silver/)
- **Decoupled** (can run independently or be orchestrated in parallel/asynchronously)

Our architecture puts together Mesh decentralization with Medallion-like Modular ETL



To set up this architecture, we start by laying out the data lake storage structure



1 Data mesh

At the highest level, the storage layer is sub-divided into **folders** for **each domain**. Each folder contains all datasets relevant to a specific data product, be it **source-oriented** (e.g., daily sales) or **consumer oriented** (e.g., sales forecast)

2 Medallion Architecture

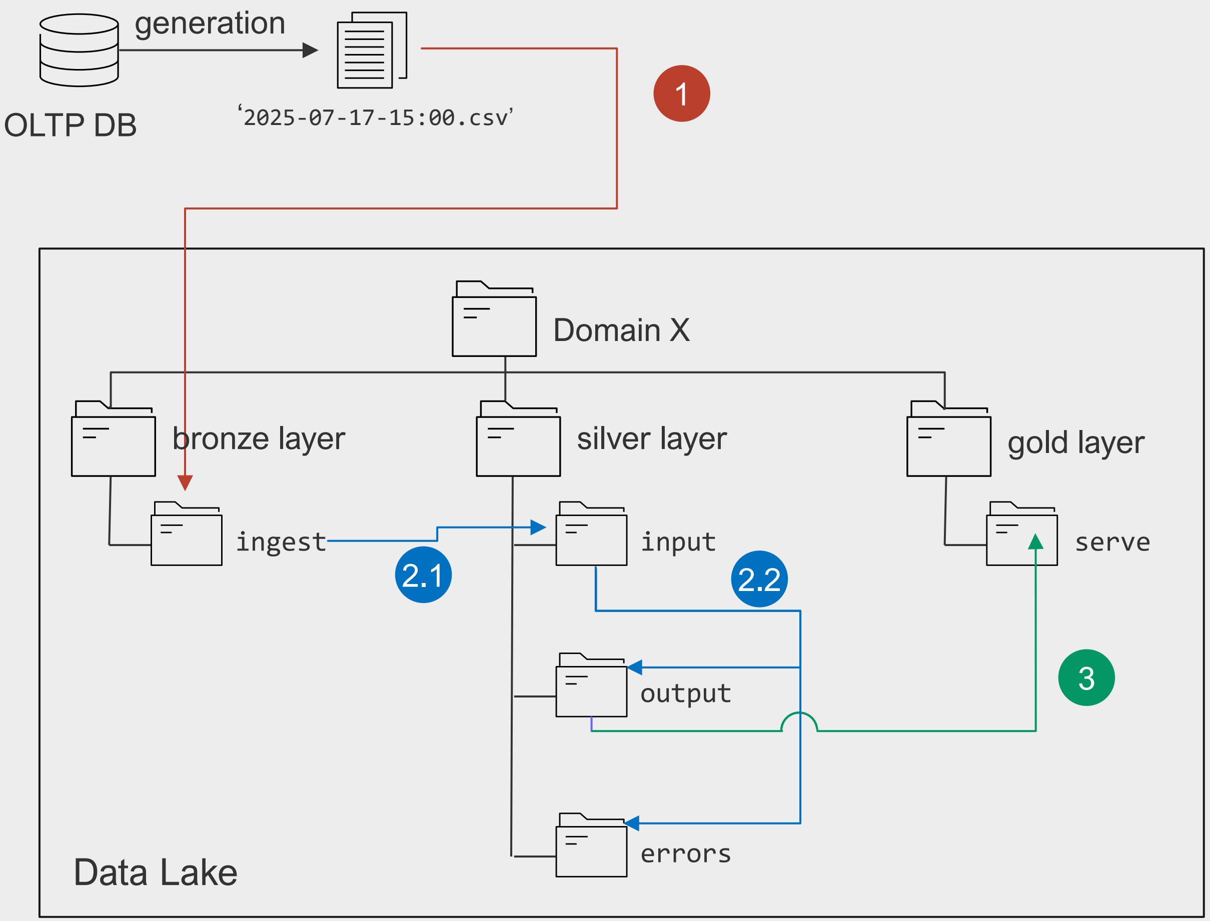
Each domain is then sub-divided into 3 folders:

2.1 **bronze layer**: contains a copy of the data as received from the data source, without applying any transformation to them

2.2 **silver layer**:
- **input**: contains the data as from bronze layer, but converted in Delta Lake format for easy manipulation
- **output**: contains the cleaned data obtained after applying specific transformations / data quality checks
- **errors**: contains records that didn't pass the quality checks

2.3 **gold layer**: contains the data ready to be served to other domains, and is **partitioned** to **optimize** business users' queries

Then, we design the data flow through an ETL pipeline decomposed into 3 main stages



1 **Ingest**
A process (e.g., a Azure Data Factory Pipeline, a Databricks notebook) uploads data without applying any changes and in the **original format**

2 **Transform**
The transform stage is responsible of applying transformations to data to make them compliant with defined quality standards

2.1 A process (e.g., a Databricks notebook) copies data from the *ingest* folder into the *input* folder, converting them into *delta lake* format

2.2 A process (e.g., a Databricks notebook) verifies data quality (e.g., schema, formats, specific business rules, etc.), and outputs valid records in the *output folder*, whereas the others go in the *errors* folder

3 **Serve**
The serve stage is responsible of enriching the cleaned data (e.g., normalizing, adding more columns through joins, etc.) and of updating the existing *serve* table through specific writing algorithms (e.g., upsert, SCD, etc.)

Agenda

1. A Data Platform architecure
2. Edge Cases
   - Late Arriving Data
   - Accumulated Data
3. Takeaways

Annex

Storage Layer

ETL Pipeline

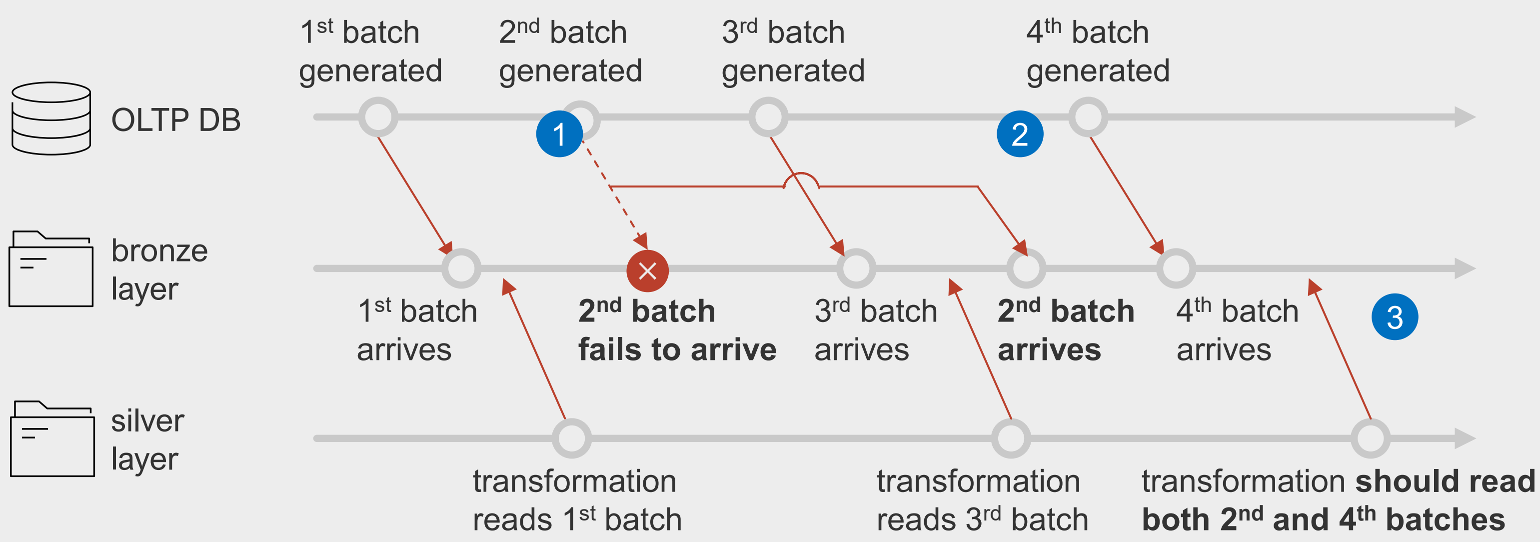
Late Arriving Data Example

Accumulated Data Example

Sources

Agenda

Source systems are subject to network latency & instability



Suppose:

- the **source system** (OLTP) is expected to **send batches of data each day** into the **bronze layer**
- the **transformation stage** of the **pipeline** is expected to **pull data** from the bronze layer each day, limiting to only "new data", i.e. data that weren't there at the last run, and putting them in the **silver layer**

1. Due to network latency or instability, the 2nd batch of data doesn't manage to arrive at the bronze layer at the expected time (i.e., before the 3rd batch arrives)
2. When the issue is solved in the source system (possibly manually), the 2nd batch of data arrives at the bronze layer
3. Next time the **transformation pipeline** runs, we expect it to pull both the 2nd and the 4th batches of data

The data batches sequence of arrival is altered

This isn't necessarily a problem, but in some situations it can be: if the transformation pipeline is designed to **run once a day** and only **read new data from the previous day**, it may involve some line of code such as:

```
SELECT * FROM bronze_layer
WHERE date_created >= SELECT(
    MAX(date_created)
    FROM silver_layer)
```

Generally, **`date_created`** is a field that contains the timestamp when the batch was created in the source system.

After the 4th batch of data has arrived, the line `MAX(date_created) FROM silver_layer` will return the timestamp at which the last batch (the 3rd) was created in the source system, which is a date **later** than that of the 2nd batch's `date_created`:

- 2nd batch `date_created: t`
- 3rd batch `date_created: t+1`

This can cause the 2nd batch of data to be filtered out and thus not to be read by the transformation pipeline

Solving this problem is quite easy…

This isn't necessarily a problem, but in some situations it can be: if the transformation pipeline is designed to **run once a day** and only **read new data from the previous day**, it may involve some line of code such as:

```
SELECT * FROM bronze_layer
WHERE date_created >= SELECT(
    MAX(date_created)
    FROM silver_layer)
```

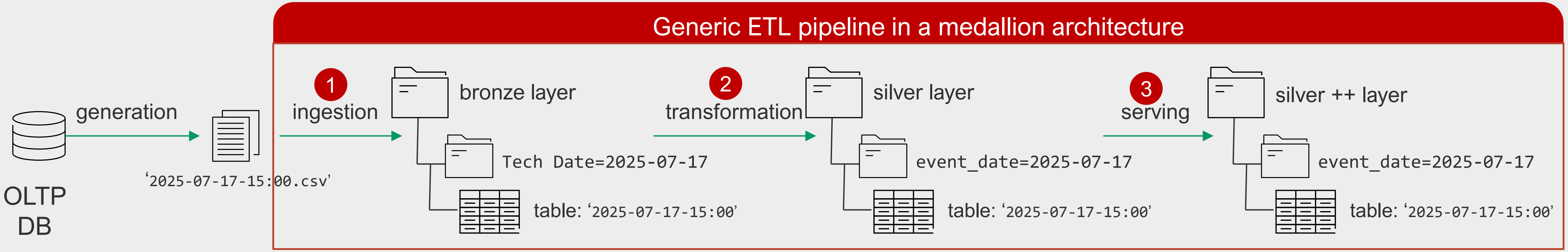
Generally, **`date_created`** is a field that contains the timestamp when the batch was created in the source system.

After the 4^{th} batch of data has arrived, the line `MAX(date_created) FROM silver_layer` will return the timestamp at which the last batch (the 3^{rd}) was created in the source system, which is a date **later** than that of the 2^{nd} batch's `date_created`:

- 2^{nd} batch `date_created: t`
- 3^{rd} batch `date_created: t+1`

This can cause the 2nd batch of data to be filtered out and thus not to be read by the transformation pipeline

…but the solution has an impact on the architecture: data must be partitioned by *Tech Date*



1

```sql
SELECT * FROM
landing_layer
WHERE date > '2025-07-01'
```

2

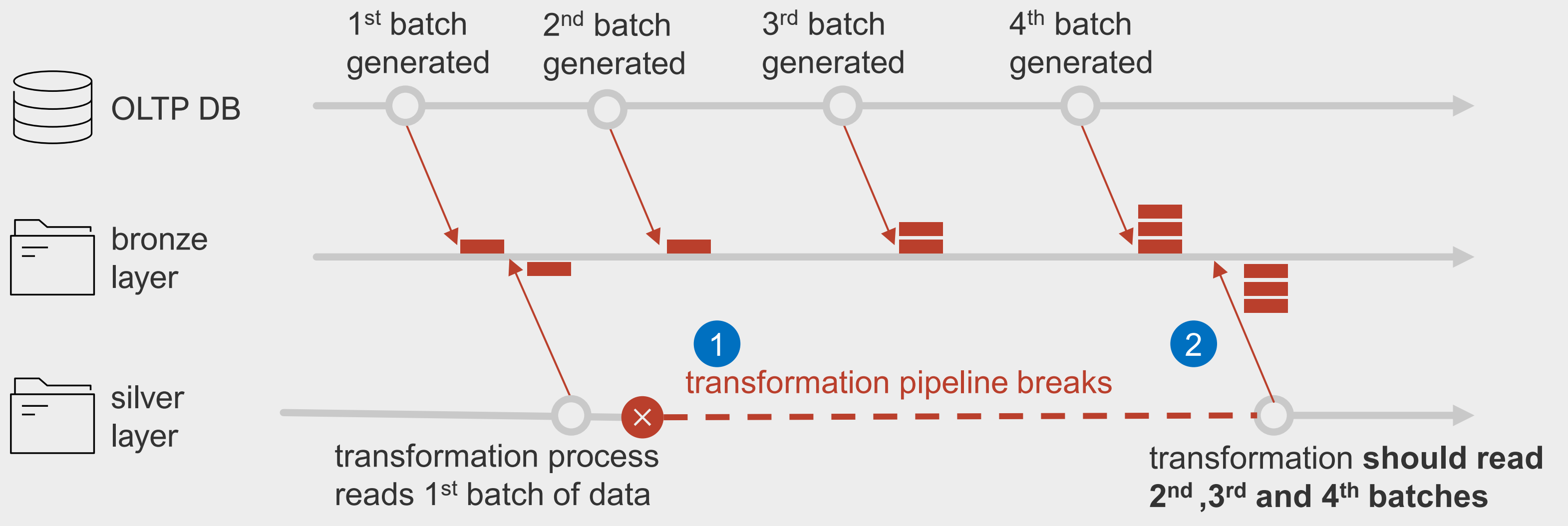
```sql
SELECT * FROM
bronze_layer
WHERE date > '2025-07-01'
```

3

```sql
SELECT * FROM
silver_layer
WHERE date > '2025-07-01'
```

Agenda

If a step of our pipeline breaks while all the preceding continue to work, data will accumulate



Suppose:
- the **source system** (OLTP) is expected to **send batches of data each day** into the **bronze layer**
- the **transformation stage** of the **pipeline** is expected to **pull data** from the bronze layer each day, limiting to only “new data”, i.e. data that weren’t there at the last run, and putting them in the **silver layer**

1. Due to a failure of the transformation pipeline (e.g.,a bug), the 2nd and 3rd batch of data aren’t processed, and accumulate in the bronze layer
2. At the 4th batch, the pipeline is back again, and is able to read data from the bronze layer. As a normal behaviour, we would expect that all accumulated batches are read and processed

The data batches sequence of arrival is altered

This isn't necessarily a problem, but in some situations it can be: if the transformation pipeline is designed to **run once a day** and only **read new data from the previous day**, it may involve some line of code such as:

```
SELECT * FROM bronze_layer
WHERE date_created >= SELECT(
    MAX(date_created)
    FROM silver_layer)
```

Generally, **`date_created`** is a field that contains the timestamp when the batch was created in the source system.

After the 4th batch of data has arrived, the line `MAX(date_created) FROM silver_layer` will return the timestamp at which the last batch (the 3rd) was created in the source system, which is a date **later** than that of the 2nd batch's `date_created`:

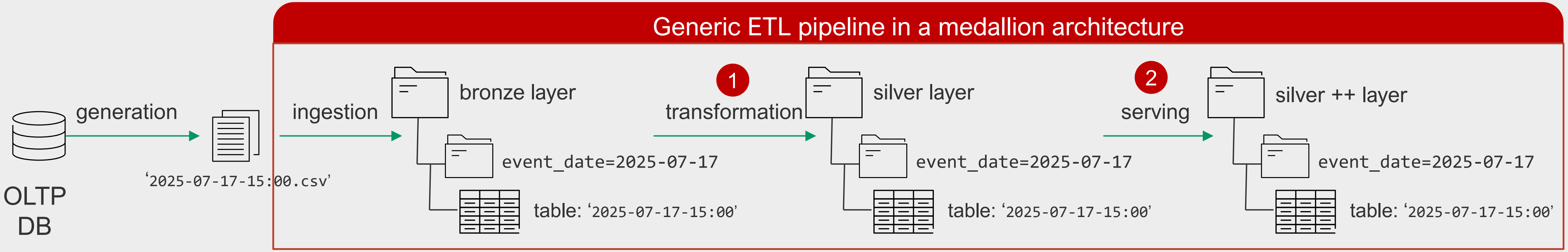
- 2nd batch `date_created: t`
- 3rd batch `date_created: t+1`

This can cause the 2nd batch of data to be filtered out and thus not to be read by the transformation pipeline

In presence of accumulated data, the pipeline must behave as if there were no interruption: process each «unit» singularly

processing units

In our pipeline, this problem can affect all steps, as they all are independent

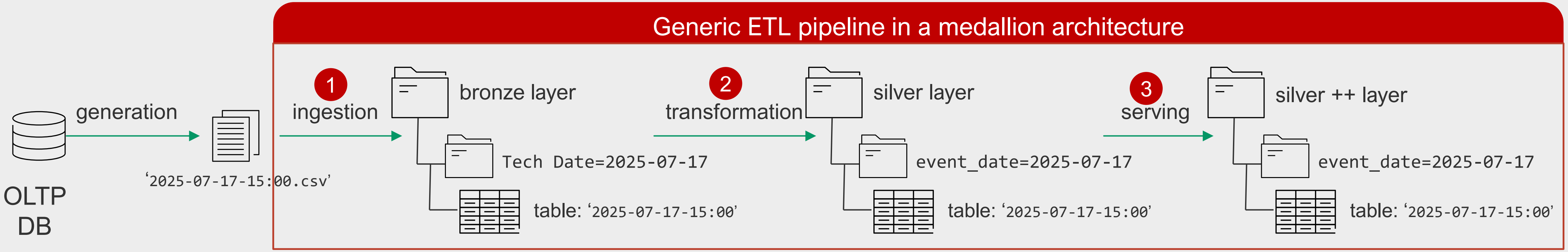


1 Transformation

2 Serving

Read input → Apply specific logics → Write output

This has an impact on the architecture: data must be partitioned by *Tech Date*



1

```
SELECT * FROM
landing_layer
WHERE date > '2025-07-01'
```

2

```
SELECT * FROM
bronze_layer
WHERE date > '2025-07-01'
```

3

```
SELECT * FROM
silver_layer
WHERE date > '2025-07-01'
```

Agenda

n takeaways for architecture design

The field on which you filter impacts how you partition

The field on which you filter impacts how you handle late arriving data

End. Thank you

nicola.orecchini@gmail.com

Rookie Data Engineer

Agenda

1. A Data Platform architecure
2. Edge Cases
   - Late Arriving Data
   - Accumulated Data
3. Takeaways

Annex

Storage Layer

ETL Pipeline

Late Arriving Data Example

Accumulated Data Example

Sources

Agenda

When you have a large dataset stored on disk as a single Parquet file, filtering it can be costly

```sql
SELECT *
FROM orders
WHERE 2025-07-18 <= date <= 2025-07-19
```

orders

Order ID	Item	Date
1001	T-shirt	2025-07-15
1002	Coffee Mug	2025-07-16
1003	Laptop Case	2025-07-16
1004	Notebook	2025-07-17
1005	Water Bottle	2025-07-18
1006	Headphones	2025-07-19
1007	Phone Stand	2025-07-20

To execute this query, the Spark engine has to read the entire orders table, as it doesn't know where records with the requested *date* are

Source: https://delta.io/blog/liquid-clustering/

To solve this challenge, you need a data layout, a way to organize your data in the storage.The traditional data layout is *partitioning*[1]

Order ID	Item	Date
1001	T-shirt	2025-07-15
1002	Coffee Mug	2025-07-16
1003	Laptop Case	2025-07-16
1004	Notebook	2025-07-17
1005	Water Bottle	2025-07-18
1006	Headphones	2025-07-19
1007	Phone Stand	2025-07-20

You split your dataset into smaller chunks based on the values in a specific column (e.g., *date*)

Order ID	Item	Date
1001	T-shirt	2025-07-15

Order ID	Item	Date
1002	Coffee Mug	2025-07-16
1003	Laptop Case	2025-07-16

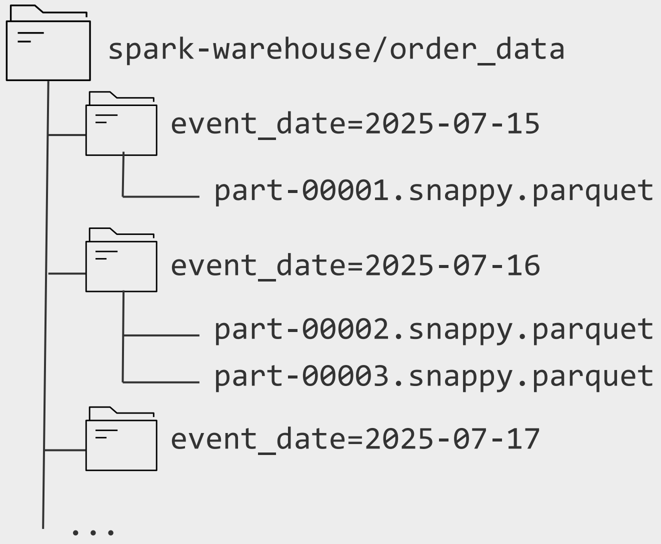
Order ID	Item	Date
1004	Notebook	2025-07-17

Order ID	Item	Date
1005	Water Bottle	2025-07-18

Order ID	Item	Date
1007	Phone Stand	2025-07-20

Then, you physically organize the dataset in **folders on disk**

- each chunk gets its own folder



Now, queries on the *date* column will run faster, because of **partition pruning**: Spark will read only files in relevant folders

Pros:

✓ **reduce volume of data read by queries** (partition pruning), but only if you know exactly which queries will run frequently on the dataset

✓ **optimize disk or cloud I/O**

Cons:

× **not flexible**: you need to decide which column to partition on

× if you want to **change partition column**, you have to rewrite the entire dataset

1. The word *partitioning*, in Spark, can be used with 2 different meanings. One is the one we're describing, the other refers to distributed computing. See Annex for a summary slide on this ambiguity

Source: https://delta.io/blog/liquid-clustering/

If you use partitioning, be sure to do it on a column that then you use in queries

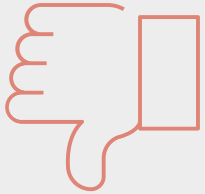
```
SELECT *
FROM sales
WHERE 2025-07-18 <= event_date <= 2025-07-19
```

full dataset partitioned by sale_rep_name

sale_rep_name='Jack Lemon'

sale_rep_name='Johnson Righeira'

sale_rep_name='Manu Chao'



To execute the above query, Spark would need to go through all files

```
SELECT *
FROM sales
WHERE 2025-07-18 <= event_date <= 2025-07-19
```

full dataset partitioned by event_date

event_date=2025-07-17

event_date=2025-07-18

event_date=2025-07-19

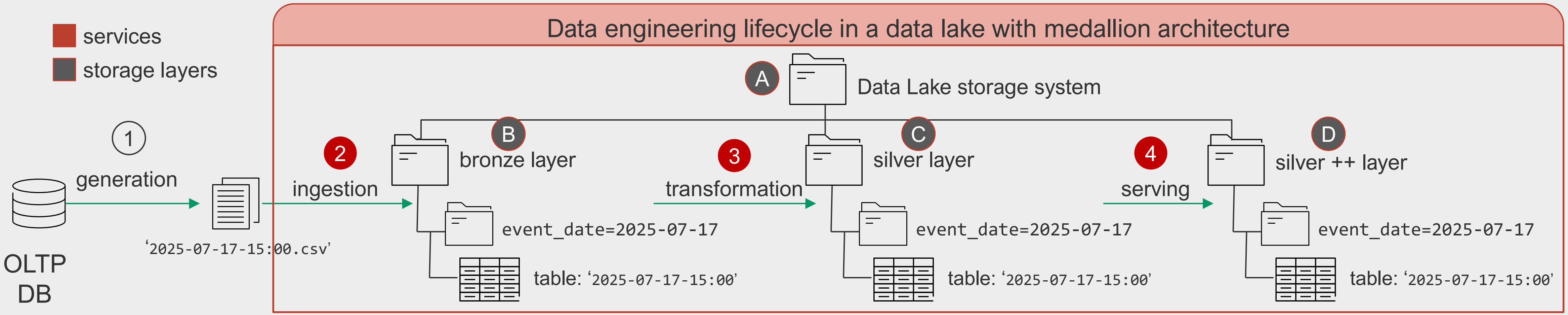
Partition pruning: Spark just looks into the relevant folders

Choose the key to partition data as the key that will be most frequently used in queries

An alternative to partitioning is Liquid Clustering

Agenda

During the design, some key considerations have shaped the architecture of the data lake



A

Data layout

- What are the **most frequent queries** that will be performed on the data?
- How to **organize data on the disk** to guarantee maximum **efficiency** of such **queries**? Partitioning? Liquid clustering?
- Do we store the complete history of data in each layer?

2

Data source type

- How will the pipeline **receive data from** the **source** systems? Will the pipeline receive snapshots of data, delta data or events? How often?
- Is it possible that the source could produce **late arriving data?**

2 3 4

Pipeline characteristics

- What are minimum **SLA** the **pipeline** must **guarantee**? (e.g., idempotency, …)
- How does **each step read input from the previous**? **Icrementally** (i.e., only new data), or **fully** (i.e., all data)? It may seem a dumb question, but if you read incrementally then you could have challenges with **late arriving data**

4

Writing algorithms

- How will we have to **write the incoming data**? Overwrite the existing dataset? Maintain history?

In any ETL pipeline, there is a point where you have to combine new incoming data into an existing dataset

new incoming data

Order ID	Item	Date
1001	T-shirt	2025-07-15
1002	Coffee Mug	2025-07-16
1003	Laptop Case	2025-07-16
1004	Notebook	2025-07-17
1005	Water Bottle	2025-07-18
1006	Headphones	2025-07-19
1007	Phone Stand	2025-07-20

existing dataset

Order ID	Item	Date
1001	T-shirt	2025-07-15
1002	Coffee Mug	2025-07-16
1003	Laptop Case	2025-07-16
1004	Notebook	2025-07-17
1005	Water Bottle	2025-07-18

- new data
- old data
- inactive data

There are multiple ways of doing it: Full vs Incremental

Full

Discard the current destination table and create a new one from the entire new incoming data

Incremental

Insert only a subset of incoming data into the destination table, while leaving the rest untouched.
There are many possibilities, and we report 3 of the most used

Full refresh

`new incoming`

ID	Value	Date
1	F	02-07-25
2	G	02-07-25
3	H	02-07-25
4	I	02-07-25
5	L	02-07-25

`destination`

ID	Value	Date
1	A	01-07-25
2	B	01-07-25
3	C	01-07-25
4	D	01-07-25
5	E	01-07-25

`updated destination`

ID	Value	Date
1	F	02-07-25
2	G	02-07-25
3	H	02-07-25
4	I	02-07-25
5	L	02-07-25

Overwrites the entire dataset:
1. discard all records in `destination`
2. insert in `destination` all records coming from `new incoming`

Warning: rebuilding the whole table can take time and cost more money. However, if the table is not large the operation can be still affordable (a few million rows or less)

Append

`new incoming`

ID	Value	Date
1	A	01-07-25
2	B	01-07-25
3	C	01-07-25
4	D	02-07-25
5	E	02-07-25
6	F	03-07-25

`destination`

ID	Value	Date
1	A	01-07-25
2	B	01-07-25
3	C	01-07-25
4	D	02-07-25

`updated destination`

ID	Value	Date
1	A	01-07-25
2	B	01-07-25
3	C	01-07-25
4	D	02-07-25
4	D	02-07-25
5	E	02-07-25
6	F	03-07-25

Insert all or some of the new incoming records into the destination table:
1. apply any filters on `updates` to **get only new records***
2. insert records from step 1 into `destination`

Warning: depending on the filters applied in step 1, `destination could` have duplicates (e.g., id=4 in the example is duplicated, because in step 1 the filter was something like `where date >= 02-07-25`)

Upsert

`new incoming`

ID	Value1	Value2
2	new	new
3	new	new
99	x	y

`destination`

ID	Value1	Value2
1	old	old
2	old	old
3	old	old

`updated destination`

ID	Value1	Value2
1	old	old
2	new	new
3	new	new
99	x	y

Solves the problem of duplicate records of Append. If the unique key already exists in the destination table, updates the record; if the records don't exist, inserts them:
1. apply any filters on updates to **get only new & updated records***
2. get updated records ids: ids that are both in `new incoming` and in `destination`
3. get new record ids: ids of step 1 – ids of step 2
4. update records from step 2 and insert records from step 3

Slowly Changing Dimension

`new incoming`

ID	Key	Start
2	X	2025
3	Y	2025
99	Z	2025

`destination`

ID	Key	Start	End	Active
1	A	2020	2999	Y
2	B	2020	2999	Y
3	C	2020	2999	Y

`updated destination`

ID	Key	Start	End	Active
1	A	2020	2999	Y
2	B	2020	2024	N
3	C	2020	2024	N
2	X	2025	2999	Y
3	Y	2025	2999	Y
99	Z	2025	2999	Y

A mix of Append and Upsert. Here, the goal is to maintain the history. new records. The process goes on similar to Upsert, with the difference that, at step 4,
1. **new records***: rows are inserted and marked as "active"
2. changed records: old version is maintained and marked as "inactive; new version is inserted and marled as "actrive"

* see next slides to understand what we mean by "new records"
Source: https://medium.com/refined-and-refactored/dbt-incremental-choosing-the-right-strategy-p1-6113d51898ec

Here are some tested patterns you can use for each scenario

Full

```
--Insert Overwrite Pattern
INSERT OVERWRITE TABLE
vendite_silver
SELECT *
FROM vendite_bronze
```

All patterns guarantee idempotency

Upsert

```
--Merge Pattern
MERGE INTO target USING
updates
ON target.id = updates.id
WHEN MATCHED THEN UPDATE
WHEN NOT MATCHED THEN INSERT
```

```
--Delete-Write Pattern
DELETE FROM target
WHERE last_updated = '2025-07-
17'
INSERT INTO target
SELECT * FROM updates
WHERE last_updated >= '2025-
07-17'
```

Slowly Changing Dimension

```
--SCD Type 2 Pattern
MERGE INTO dim_clienti AS
target
USING updates
ON target.cod_fisc =
updates.cod_fisc AND
target.fine_validità = '2999-
12-31'

WHEN MATCHED AND
target.indirizzo <>
updates.indirizzo THEN
    UPDATE SET fine_validità =
current_date()

WHEN NOT MATCHED THEN
    INSERT (cod_fis, ind,
iniz_val, fine_val)
    VALUES (...)
```

When you use Upsert or SCD, you need to define what «new records» are

In the example patterns, we just put a dummy date for the sake of simplicity. In reality, you need to calculate this date from your existing dataset

There are multiple ways:

- selecting the max date of your existing dataset
- selecting mthe max date of the new incoming dataset
- selecting the timestamp at which the pipeline is running

Selecting the best way depends of course on the business logic you want to accomplish, as well as what guarantees you want to give your pipeline. But we will discuss the latter later on

```
--Delete-Write Pattern

DELETE FROM target
WHERE last_updated = SELECT(
    MAX(last_updated)
    FROM target)

INSERT INTO target
SELECT * FROM updates
WHERE last_updated >= SELECT(
    MAX(last_updated)
    FROM target)
```

When you use Upsert or SCD, a deduplication algorithm must be implemented

Both Upsert and SCD algorithms work with an assumption:

If the value of an existing record has changed, then there must be 1 and only 1 new version of it to replace it

Why? Because if there were 2 or more new versions, which one should the algorithm use to update the record?

new incoming

ID	Value	Start
2	X	2025
2	Y	2025
99	Z	2025

There are two records with ID 2. Which one do we need to use to replace the old value for ID 2?

destination

ID	Value	Start	End	Active
1	A	2020	2999	Y
2	B	2020	2999	Y
3	C	2020	2999	Y

So, if for whatever reason you find 2 or more “new versions” of an old record, you must set up a **deduplication algorithm** to make sure there’s exactly 1 new version to feed into the upsert/SCD algorithm.

For example, the deduplication logic could be: take the most recent record, and, in case of tie, choose one randomly

Selecting the best writing algorithm: 4 factors to consider

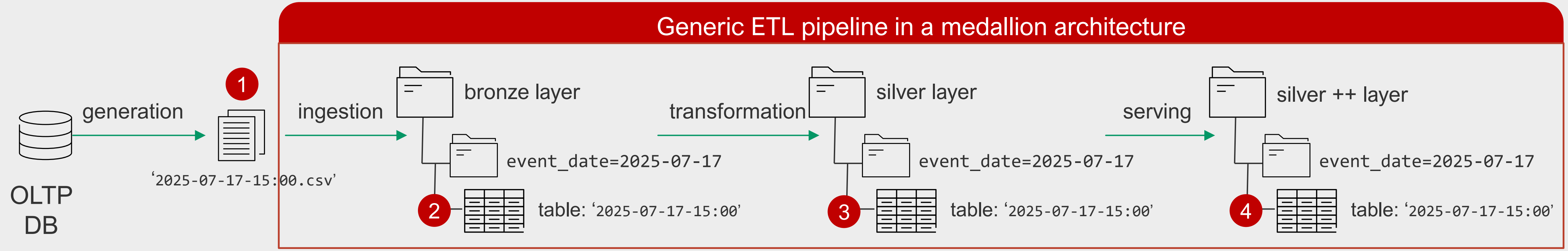
Factor	Possibilities	
Incoming dataset type	**Snapshot** You receive each time the whole dataset	**Delta** You receive each time only records generated from the last ingestion
Incoming dataset size	**Small** Few million rows	**Large** Millions/billions rows
Transformations performed on data	**Simple** Schema/format checks, nulls, deduplication	**Computationally expensive** Regex expressions, UDFs
Business requirements	**Keep history**	**Discard history**

Suitable writing algorithms*

- Full refresh
- Append
- Upsert
- SCD

*generally speaking. Choice of the proper algorithm always needs to be evaluated taking into account the broader context

In our pipeline, writing algorithms are implemented on the serve phase



1

Order ID	Item	Generation Time
1001	T-shirt	2025-07-15
1002	Coffee Mug	2025-07-16
1003	Laptop Case	2025-07-16
1004	Notebook	2025-07-17
1005	NULL	2025-07-17

2

Order ID	Item	Generation Time
1001	T-shirt	2025-07-15
1002	Coffee Mug	2025-07-16
1003	Laptop	2025-07-16
1004	Notebook	2025-07-17
1005	NULL	2025-07-17

3

Order ID	Item	Generation Time
1001	T-shirt	2025-07-15
1002	Coffee Mug	2025-07-16
1003	LaptopCase	2025-07-16
1004	Notebook	2025-07-17

4

Order ID	Item	Generation Time	Processing Time

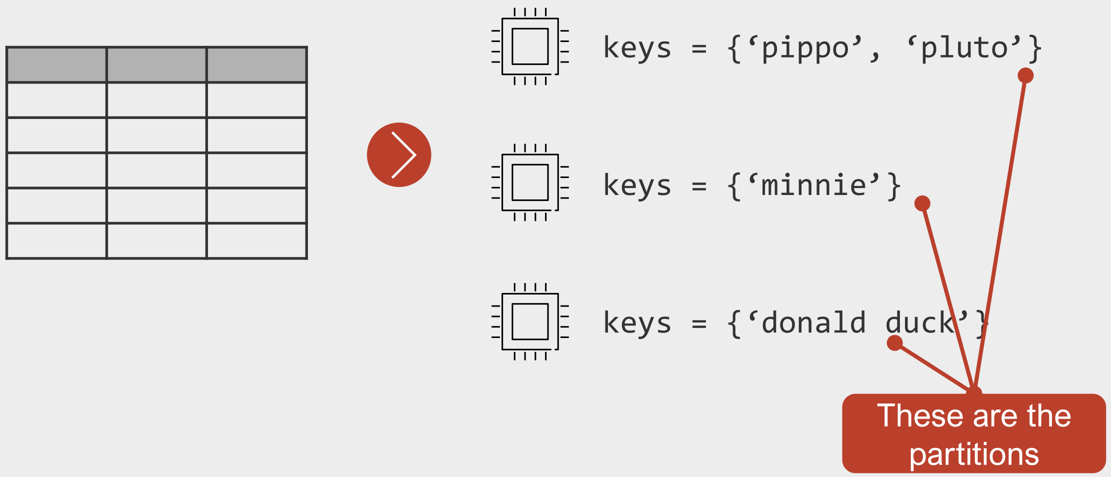
new data old data

Notice that, in the Spark world, the term "Partition" is used with 2 very different meanings

today we focus on this meaning: from now on, we'll use only «partition» to refer to it

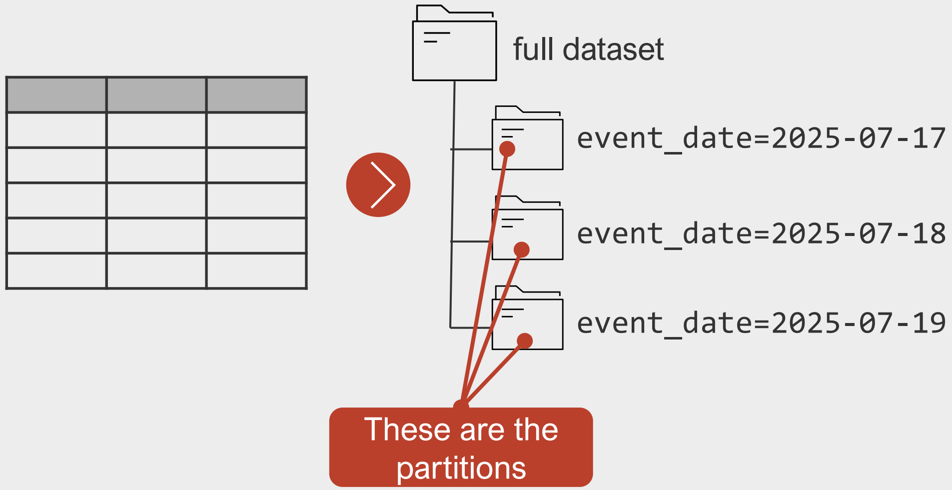
Partition (**logical**, on RAM memory)

- It refers to splitting a dataset into **chunks**, i.e. logical groupings, on the RAM
- Useful for:
  - **parallelizing** the **processing** of the dataset (each partition is processed by only 1 executor)
  - **distributing** work across the cluster to **reduce memory requirements** of each node (*horizontal scalability*)



Partition (**physical**, on disk)

- It refers to physically organizing data in **folders** (e.g., by creating groups of rows based on the value of a specific key)
- Useful for:
  - **reducing volume of data** read by queries (*partition pruning*)
  - **optimizing disk** or cloud **I/O**



Agenda

1. A Data Platform architecure
2. Edge Cases
   - Late Arriving Data
   - Accumulated Data
3. Takeaways

Annex

- Storage Layer
- ETL Pipeline
- Late Arriving Data Example
- Accumulated Data Example

Sources

Here's an example with dummy data

01/07

incoming

Order ID	Item	Date
1001	T-shirt	2025-07-01
1002	Coffee Mug	2025-07-01

silver layer

Order ID	Item	Date

Date to filter calculated as:
SELECT MAX(Date) FROM existing

upsert (simplified)

```
INSERT INTO existing
SELECT * FROM incoming
WHERE date > '2025-07-01'
```

silver layer updated

Order ID	Item	Date
1001	T-shirt	2025-07-01
1002	Coffee Mug	2025-07-01

02/07

silver layer

Order ID	Item	Date
1001	T-shirt	2025-07-01
1002	Coffee Mug	2025-07-01

```
INSERT INTO existing
SELECT * FROM incoming
WHERE date > '2025-07-01'
```

silver layer updated

Order ID	Item	Date
1001	T-shirt	2025-07-01
1002	Coffee Mug	2025-07-01

03/07

incoming

Order ID	Item	Date
1003	Laptop Case	2025-07-03
1004	Notebook	2025-07-03

silver layer

Order ID	Item	Date
1001	T-shirt	2025-07-01
1002	Coffee Mug	2025-07-01

```
INSERT INTO existing
SELECT * FROM incoming
WHERE date > '2025-07-01'
```

silver layer updated

Order ID	Item	Date
1001	T-shirt	2025-07-01
1002	Coffee Mug	2025-07-01
1003	Laptop Case	2025-07-03
1004	Notebook	2025-07-03

04/07

incoming

Order ID	Item	Date
1005	Kettlebell	2025-07-02
1006	Headphones	2025-07-02
1007	Phone Stand	2025-07-04

Records from 07-02 are late arriving data

silver layer

Order ID	Item	Date
1001	T-shirt	2025-07-01
1002	Coffee Mug	2025-07-01
1003	Laptop Case	2025-07-03
1004	Notebook	2025-07-03

```
INSERT INTO existing
SELECT * FROM incoming
WHERE date > '2025-07-03'
```

The most recent date in existing is 07-03, so the query filters incoming > 07-03

silver layer updated

Order ID	Item	Date
1001	T-shirt	2025-07-01
1002	Coffee Mug	2025-07-01
1003	Laptop Case	2025-07-03
1004	Notebook	2025-07-03
1007	Phone Stand	2025-07-04

Records from 07-02 are missing!

Solution: filter on *Tech Date*, the date the file arrives in pipeline

incoming | existing | upsert (simplified) | existing updated

01/07

incoming

ID	Item	Bus Date	Tech Date
1001		2025-07-01	2025-07-01
1002		2025-07-01	2025-07-01

existing

ID	Item	Bus Date	Tech Date

Date to filter calculated as:
SELECT MAX(Tech Date) FROM existing

upsert (simplified)

```
INSERT INTO existing
SELECT * FROM incoming
WHERE tech_date > '2025-
07-01'
```

existing updated

ID	Item	Bus Date	Tech Date
1001		2025-07-01	2025-07-01
1002		2025-07-01	2025-07-01

02/07

existing

ID	Item	Bus Date	Tech Date
1001		2025-07-01	2025-07-01
1002		2025-07-01	2025-07-01

upsert (simplified)

```
INSERT INTO existing
SELECT * FROM incoming
WHERE tech_date > '2025-
07-01'
```

existing updated

ID	Item	Bus Date	Tech Date
1001		2025-07-01	2025-07-01
1002		2025-07-01	2025-07-01

03/07

incoming

ID	Item	Bus Date	Tech Date
1003		2025-07-03	2025-07-03
1004		2025-07-03	2025-07-03

existing

ID	Item	Bus Date	Tech Date
1001		2025-07-01	2025-07-01
1002		2025-07-01	2025-07-01

upsert (simplified)

```
INSERT INTO existing
SELECT * FROM incoming
WHERE tech_date > '2025-
07-01'
```

existing updated

ID	Item	Bus Date	Tech Date
1001		2025-07-01	2025-07-01
1002		2025-07-01	2025-07-01
1003		2025-07-03	2025-07-03
1004		2025-07-03	2025-07-03

04/07

incoming

ID	Item	Bus Date	Tech Date
1005		2025-07-02	2025-07-04
1006		2025-07-02	2025-07-04
1007		2025-07-04	2025-07-04

Records from 07-02
are late arriving data

existing

ID	Item	Bus Date	Tech Date
1001		2025-07-01	2025-07-01
1002		2025-07-01	2025-07-01
1003		2025-07-03	2025-07-03
1004		2025-07-03	2025-07-03

upsert (simplified)

```
INSERT INTO existing
SELECT * FROM incoming
WHERE tech_date > '2025-
07-03'
```

The most recent Tech Date
in existing is 07-03, so the
query filters incoming with
Tech Date > 07-03

existing updated

ID	Item	Bus Date	Tech Date
1001		2025-07-01	2025-07-01
1002		2025-07-01	2025-07-01
1003		2025-07-03	2025-07-03
1004		2025-07-03	2025-07-03
1005		2025-07-02	2025-07-04
1006		2025-07-02	2025-07-04
1007		2025-07-04	2025-07-04

Records from 07-02 are now present

Agenda

Imagine a step supposed to run once a day breaks. How does the step behave when it comes up again?

01/07

incoming

Order ID	Item	Date
1001	T-shirt	2025-07-01
1002	Coffee Mug	2025-07-01

existing

Order ID	Item	Date

Date to filter calculated as:
SELECT MAX(Date) FROM existing

pipeline filter step

```
SELECT * FROM
previous_step
WHERE date > '2025-07-01'
```

existing updated

Order ID	Item	Date
1001	T-shirt	2025-07-01
1002	Coffee Mug	2025-07-01

02/07

incoming

Order ID	Item	Date
1005	Kettlebell	2025-07-02
1006	Headphones	2025-07-02

Pipeline stage is down

03/07

incoming

Order ID	Item	Date
1005	Kettlebell	2025-07-02
1006	Headphones	2025-07-02
1003	Laptop Case	2025-07-03
1004	Notebook	2025-07-03

Pipeline stage is down

04/07

incoming

Order ID	Item	Date
1005	Kettlebell	2025-07-02
1006	Headphones	2025-07-02
1003	Laptop Case	2025-07-03
1004	Notebook	2025-07-03
1007	Phone Stand	2025-07-04

existing

Order ID	Item	Date
1001	T-shirt	2025-07-01
1002	Coffee Mug	2025-07-01

pipeline filter step

```
SELECT * FROM
previous_step
WHERE date > '2025-07-01'
```

existing updated

ID	Item	Bus Date	Tech Date
1001		2025-07-01	2025-07-01
1002		2025-07-01	2025-07-01
1003		2025-07-03	2025-07-03
1004		2025-07-03	2025-07-03
1005		2025-07-02	2025-07-04
1006		2025-07-02	2025-07-04

When the step of the pipeline is up again, it will find accumulated data to process. **Can the step consider the data as one single batch and process them together?**

Agenda

1. A Data Platform architecure
2. Edge Cases
   - Late Arriving Data
   - Accumulated Data
3. Takeaways

Annex

Storage Layer

ETL Pipeline

Late Arriving Data Example

Accumulated Data Example

Sources

Sources

[1] Zhamak Dehghani, How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh (https://martinfowler.com/articles/data-monolith-to-mesh.html)

[2] James Serra, Deciphering Data Architectures: Choosing Between a Modern Data Warehouse, Data Fabric, Data Lakehouse, and Data Mesh (https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://sessionize.com/download/oiocfa~DmD7XzPK3YsAX4MzjCBWQo.pdf~Deciphering%2520Data%2520Architectures%2520-%2520Data%2520Bash.pdf&ved=2ahUKEwiNs5-TrfOOAxW1RfEDHWHJODUQFnoECCUQAQ&usg=AOvVaw375doSEsznSvZs6n8F5dz-)