# [639] Data Ingestion Considerations

Meenakshi Syamkumar

# Learning Objectives

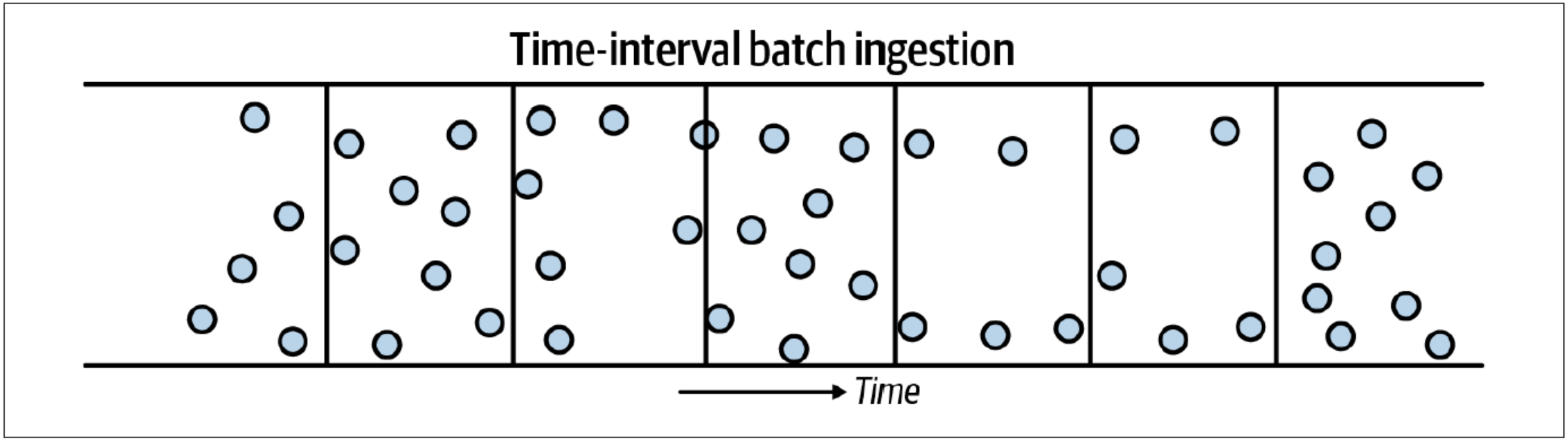
| | |
|---|---|
| Understand | the key considerations that govern batch ingestion design |
| Understand | the key considerations that govern stream / event-driven ingestion design |

# Batch Ingestion

- Processing data in bulk
  - Example: Airbyte ingestion of our survey Google sheet data
- Data is ingested by taking a subset of the data from a source system based on:
  - Time interval
  - Size of accumulated data

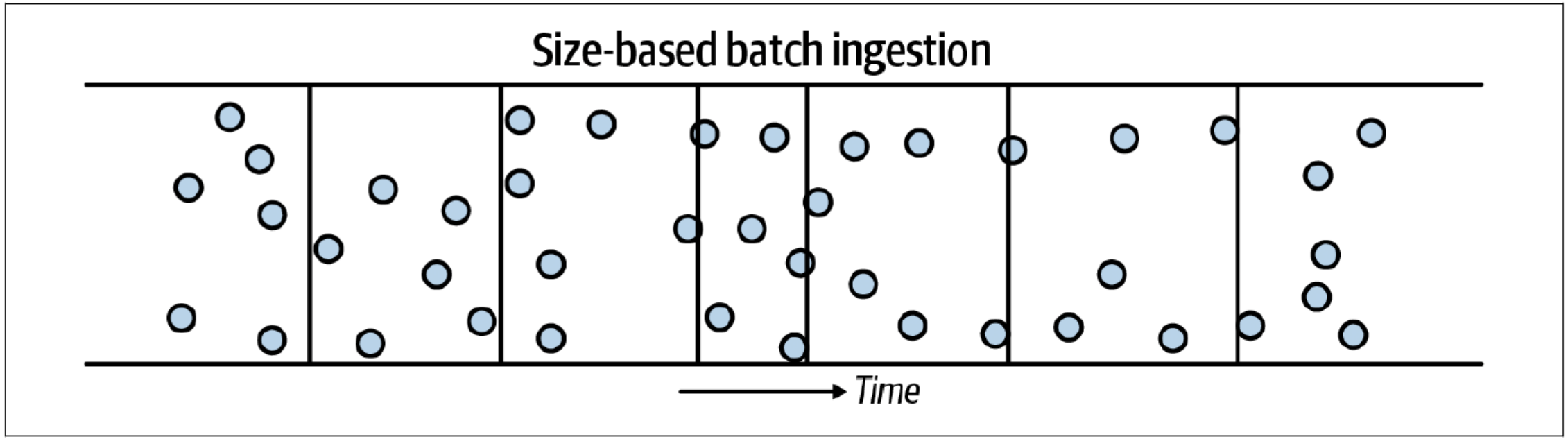
# Time-interval batch ingestion

- Use case scenario: widespread in traditional business ETL for data warehousing
- Typical pattern: process data once a day, overnight during off-hours, to provide daily reporting



# Size-based batch ingestion

- Use case scenario: when data is moved from a streaming-based system into object storage
- Data must be cut into discrete blocks for future processing in a data lake.
- Example criteria (to break data into objects): size in bytes of the total number of events



# Batch Ingestion Considerations

# Full Snapshots or Differential Extraction

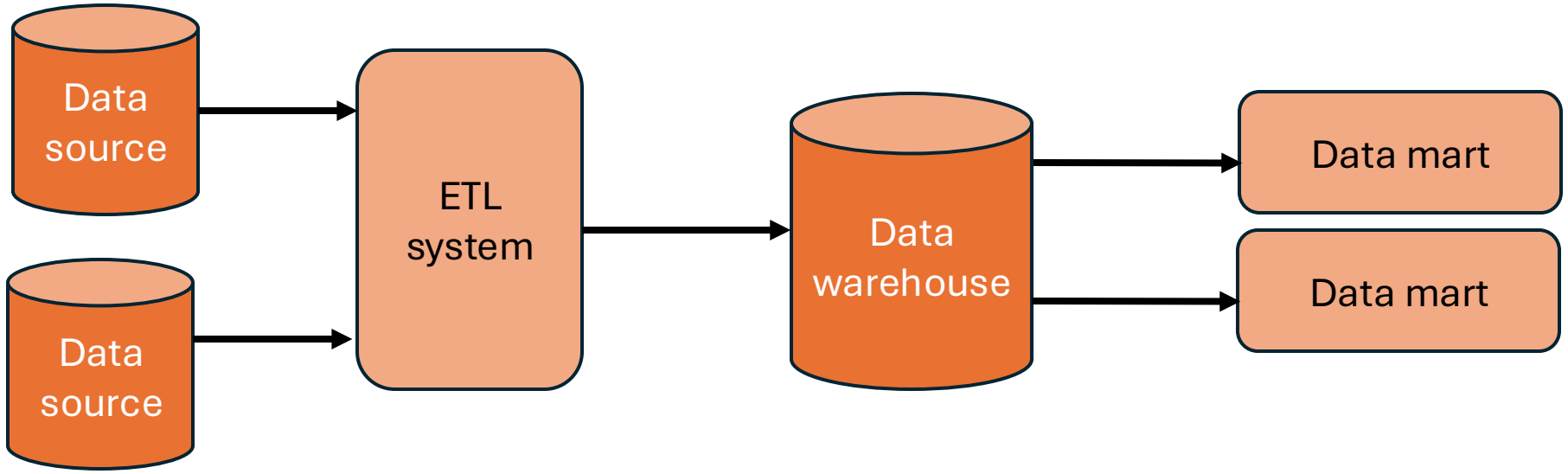
- Full snapshots: entire current state of the source system on each update read
  - Pros: simple to implement → extremely common
  - Cons: high storage requirement, incurs additional network traffic
  - Airbyte uses full snapshots for our survey Google Sheet source connector
- Differential Extraction: only pull updates and changes since the last read from the source system
  - Pros: minimizes network traffic, minimizes target storage usage
  - Cons: challenging to implement

# File-based Export and Ingestion

- Data is often moved between databases and systems using files
- Push-based ingestion pattern
  - Data export and preparation work is done on the source system side
- Advantages over direct database connections
  - Undesirable to allow direct access to the backend system (security concerns)
  - Export process run on the data-source side → complete control over data preprocessing and export
  - Many ways to push data to target systems
    - Object storage
    - Secure file transfer protocol (SFTP)
    - Electronic data interchange (EDI)
    - Secure copy (SCP)

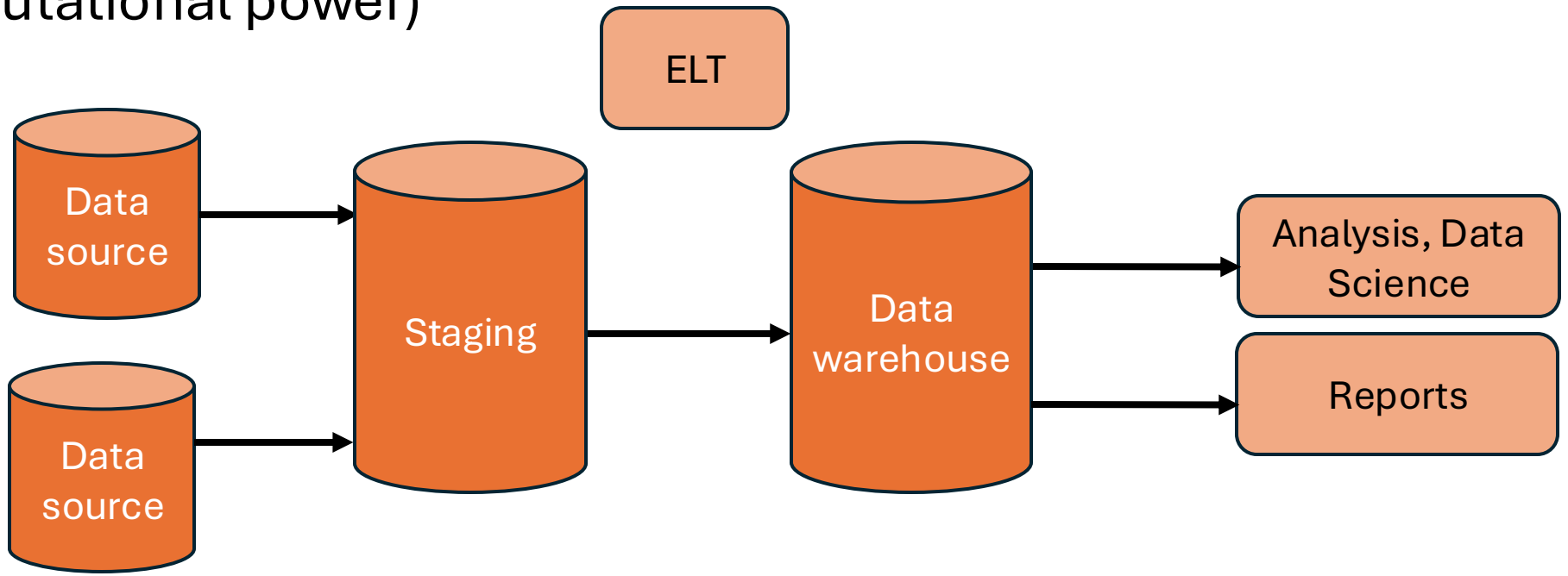
# Extract-Transform-Load (ETL)

- Extract: getting data, metadata, schema changes from a source system (push / pull / poll)
- Transform: cleaning and standardizing data, organizing and imposing business logic in a highly modeled form
- Load: pushes data into data warehouse target database system



# Extract-Load-Transform (ELT)

- Load: raw data gets moved directly from production system into a staging area in the data warehouse
- Data warehouses can handle transformations in-house (massive computational power)



# Inserts, Updates, and Batch Size

- Batch-oriented systems often perform poorly when users attempt to perform many small-batch operations
- Critical factors:
  - understand the appropriate update patterns for the data store
    - individual row-based are common in transactional databases
    - bad pattern for columnar databases due to creation of many small, suboptimal files
  - understand the limits and characteristics of your tool
    - for example, certain technologies are purpose-built for high insert rate

# Data Migration

- Most data systems perform best when data is moved in bulk rather than individual rows or events
- File object storage is an excellent intermediate stage for data transfer
- Challenge: movement of data pipeline connections from the old system to the new one

# Stream Ingestion Considerations

# Schema registries

- In streaming data, every message has a schema, and these schemas may evolve between producers and consumers
- A schema registry is a metadata repository used to maintain schema and data type integrity
- Can also track schema versions and history
- Describes the data model for messages, allowing consistent serialization and deserialization between producers and consumers

# Schema Evolution

- Fields may be added or removed
- Value types might change (ex: string to integer)
- Schema evolutions  can have unintended impacts on your data pipelines and destinations
  - Ex: an IoT device gets a firmware update that adds a new field to the event it transmits
- Solutions:
  - Use schema registries to version your schema changes
  - Error-handling using dead-letter queues
  - Regularly communicated with upstream stakeholders about potential schema changes and proactively address instead of reactively handling breaking changes

# Late-arriving data

- Event data might arrive late
  - A group of events might occur around the same time frame, but some might arrive later than others (late ingestion times)
  - Ex: an IoT device messages coming in late due to Internet issues
- Critical consideration: do not assume ingestion or process time is the same as the event time
- Solution: set a cut-off timer after which late-arriving data will not longer be processed

# Message delivery semantics

- Exactly once
  - message gets delivered exactly once and after acknowledgement won't be delivered again. Ex: Apache Kafka
  - prioritized when duplicates must be avoided with complex tracking mechanism
- At-least once
  - message gets delivered at least once, but it may be delivered more than once (duplicates)
  - prioritized when duplicates can be handled
- At-most once
  - message get delivered at most once, meaning it either arrives or not at all
  - low latency prioritized over data completeness -> data loss is acceptable

# Ordering, Multiple Delivery, Replay, and Message Size

- Streaming platforms are generally built on distributed systems
- Problem: messages may be delivered out of order and more than once
- Message replay
  - Allows readers to request a range of messages from the history, allowing rewind to a particular point in time
  - Useful when you need to re-ingest data for a specific time range
- Message size
  - Ensure that the streaming framework can handle the maximum expected message size

# Time to Live (TTL)

- How long will you preserve your event record?
- Key parameter: Maximum message retention time aka time to live (TTL):
  - Configuration for long long you want events to live before they are acknowledged and ingested
- Any acknowledged event that is not ingested after its TTL expires automatically disappears.
  - Helpful to reduce backpressure and unnecessary event volume in your event-ingestion pipeline
- An extremely short TTL (milliseconds or seconds) might cause most messages to disappear before processing
- A very long TTL (several weeks or months) will create a backlog of many unprocessed messages, resulting in long wait times

# Error Handling and Dead-Letter Queues

- Sometimes events aren't successfully ingested
  - gets sent to a non-existent topic or message queue
  - size is too large
  - expired past its TTL
- Events cannot be ingested need to be rerouted to a dead-letter queue