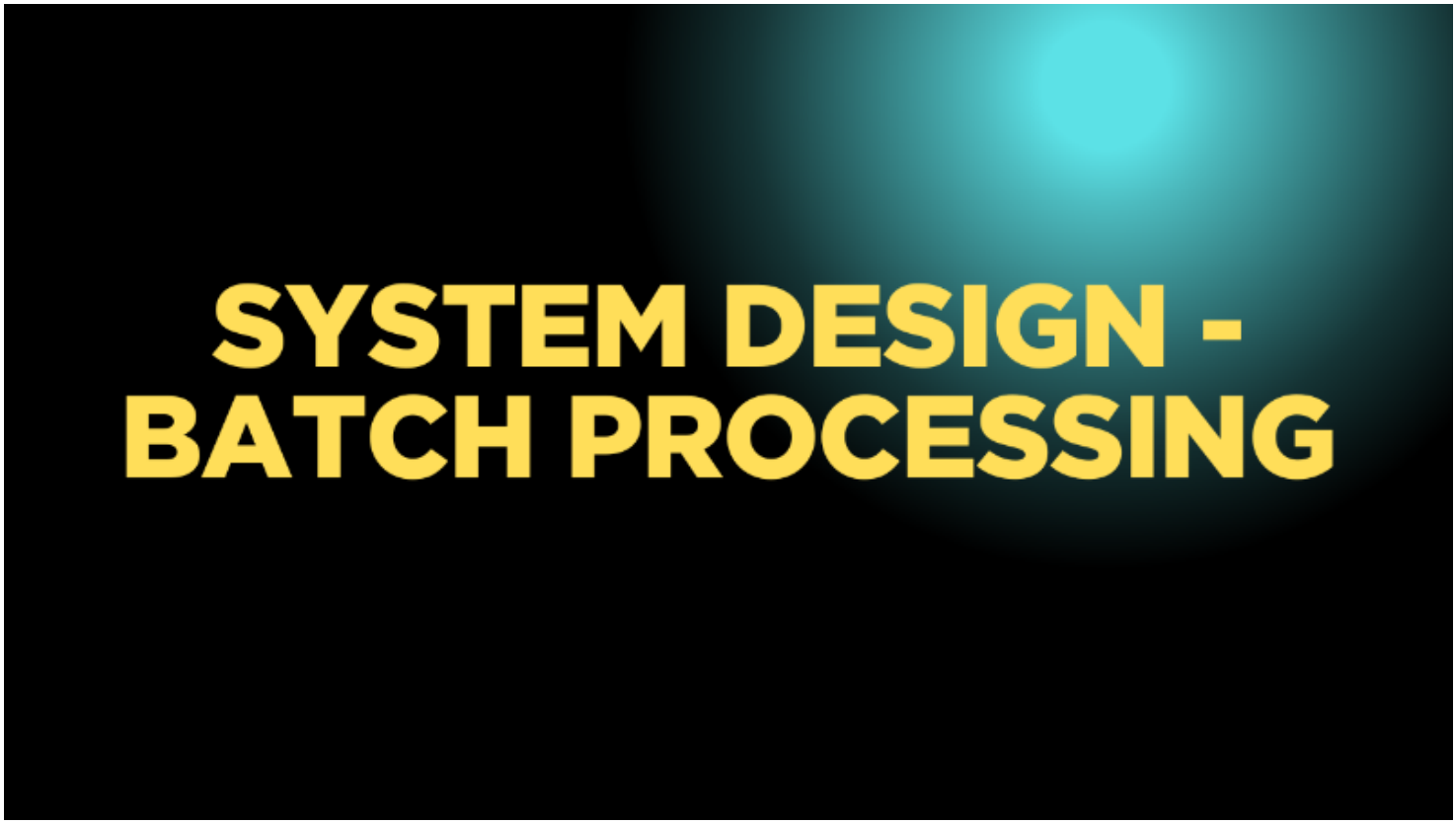

SYSTEM DESIGN -
BATCH PROCESSING

System Design Question and Solution: Batch

One common type of data engineering system design question is the batch process pipeline, with any system design question there are many viable approaches, below is just one example. Take a few minutes to walk through your own solution first, before diving in.

Design a batch data ingestion pipeline to ingest and model ecommerce sales data for use in reporting dashboards.

Understand the data requirements and establish scope

Data Characteristics:

- What's the format of the incoming data? (e.g., JSON, CSV, XML)
- Is the data structured, semi-structured, or unstructured?

Data Sources:

- From where will the data be sourced? Can we assume its a OLTP database such as a postgres?
- What are the entities we care about, ex orders, customers, etc?
- How many data sources are involved, and are they all consistent in terms of format and structure?

Volume and Velocity:

- How much data are we expecting to ingest daily or in each batch?
- How often will the data be ingested? (e.g., once a day, once an hour)

Transformation and Processing:

- What kind of transformations or computations are required on the data before it's ready for reporting?
- Are there any specific data quality checks or cleansing procedures that need to be applied?

Destination and Usage:

- Where will the ingested data be stored? Can we assume the raw data will be stored in a data lake, and the modeled data in a data warehouse?
- How will the reporting dashboards access this data? What tools or platforms are used for reporting?

Historical Data:

- Is there historical sales data that needs to be ingested initially?
- If so, how much historical data is there?

SLAs and Latency:

- What is the acceptable latency from when data is produced to when it's available in the reporting dashboards?
- Are there any Service Level Agreements (SLAs) in place regarding data freshness or uptime?

Data Security and Compliance:

- Are there any specific security protocols or compliances (like GDPR or PCI DSS) that need to be adhered to during data ingestion and storage?
- Do we need to consider data encryption, masking, or anonymization?

Scalability and Future Growth:

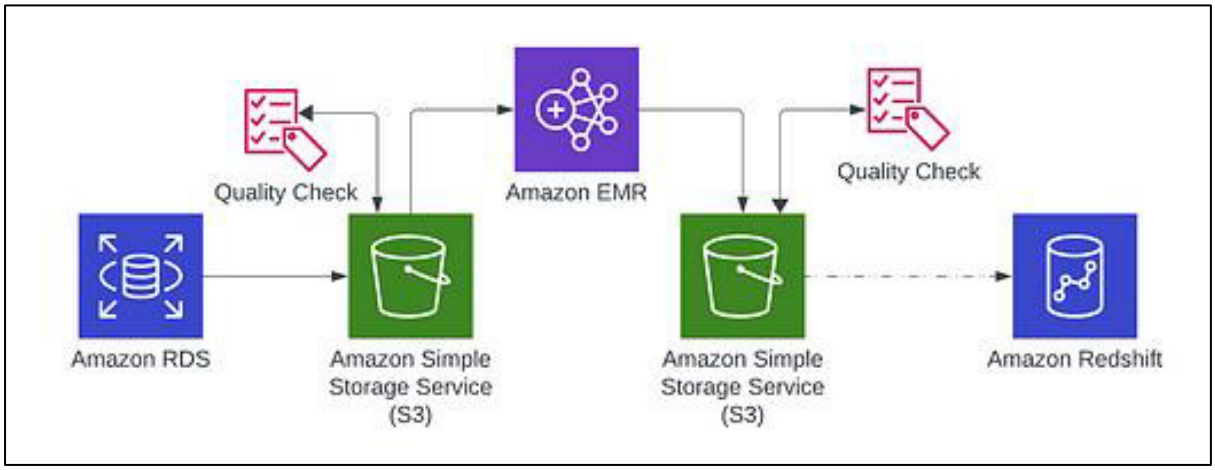
- Do we expect a significant increase in the data volume in the foreseeable future?
- How should the system be designed to handle this growth?

By asking these questions, you can clarify the requirements, constraints, and objectives for the ingestion pipeline, allowing them to provide a more accurate and efficient design.

Propose high-level data flow and get buy-in

One could approach this problem by creating a data pipeline is orchestrated using Airflow. Initially, data can be extracted from the PostgreSQL database and is subsequently loaded into an AWS S3 bucket in the CSV format. This raw data serves as input for a subsequent Spark job, which reads the CSV files, performs necessary transformations and models the data, and then outputs the processed data in the more efficient Parquet format to a separate S3 bucket.

Post transformation, an AWS Glue Crawler can be deployed to catalog the transformed data stored in the S3 bucket. This cataloging action facilitates the integration of the processed data with Redshift Spectrum, allowing for seamless querying of the Parquet data on S3 directly from Redshift. If low latency query speed is important the data from the Spectrum table could be used to populate a native Redshift table. Throughout this entire process, Airflow ensures that each task is executed in sequence, with appropriate error handling, monitoring, and alerting mechanisms in place, and has room to scale to include data quality checks, additional sources, and more. This design offers a robust and scalable solution for ingesting and modeling e-commerce sales data, making it primed for reporting dashboards.



Dive deep into the data pipeline design

Data Extraction:

- *Source*: The source of our data is a PostgreSQL database, which might contain tables and records associated with e-commerce transactions, such as customer information, product details, sales transactions, and more.
- *Method*: Utilizing Airflow's PostgresOperator, a SQL query is executed to select the relevant data. This could be a full extraction or incremental extraction based on a certain criterion like a timestamp. It's crucial to optimize the extraction process to ensure minimal load on the source system and efficient data retrieval.
- *Output*: The retrieved data from the PostgreSQL database is serialized into a CSV format. This format is chosen because of its ubiquity and ease of use, making it compatible with a vast array of tools and platforms.

Data Storage:

- *Destination*: The CSV data is uploaded to an AWS S3 bucket, a highly durable and scalable object storage service.
- *Method*: If we are hosting our Postgres database on AWS RDS we can use the `aws_s3.query_export_to_s3` function to unload directly to S3, else we can use the S3Hook in Airflow, which pushes the CSV files to a designated S3 bucket, ideally partitioned date.
- *Advantages*: S3 provides durability, quick retrieval, and easy integration with many AWS services. Storing data in this intermediate form ensures it's readily accessible for subsequent transformation steps

  .

Data Transformation:

- *Tool*: Apache Spark, a distributed computing system known for processing large datasets, is employed for the transformation process.
- *Method*: The Spark job ingests the CSV data from the S3 bucket and initiates the transformation. This may include cleaning data (handling missing values, outliers), deriving new columns, aggregating data points, and optimizing the schema for analytical operations. The output of the transformation stage will still align with the input format, we will model the data next. In spark this could easily be a separate EMR step.
- *Output*: Once the transformation is complete, Spark writes the data back to a different S3 bucket in Parquet format. Parquet is a columnar storage format optimized for performance and is especially beneficial for analytical queries due to its compression and columnar read capabilities.

Data Modeling:

- *Approach*: At this stage, we focus on structuring the data in a way that's optimal for the end use-case: reporting dashboards. Depending on the nature of the reports, data might be modeled into fact and dimension tables, adhering to principles like the star schema. Or if our dashboarding tools performed better with less joins we could leverage the one big table schema.
- *Tool*: The same Spark job can be extended to perform this modeling. By defining appropriate data frames and leveraging Spark's SQL capabilities, the data can be reshaped and indexed as per the requirements.
- *Integration with Redshift Spectrum:* Once the data is modeled and stored in Parquet format, an AWS Glue Crawler catalogs this data. This catalog is then used by Redshift Spectrum, allowing users to execute Redshift SQL queries directly on the S3 Parquet data, thereby extending the querying capabilities of their existing Amazon Redshift data warehouses to unbounded datasets. If our use case requires faster query speeds, we could use the Glue table to push the data into a native Redshift table.

Quality Checks After Data Extraction:

- *Expectation Suites for Source Data:* Before you even start extracting the data from your PostgreSQL database, you can design an "expectation suite" tailored for the source data. This suite can consist of expectations like ensuring specific columns are present, validating data types, checking for null values, and more.
- *Validation After Extraction:* Once the data is extracted and serialized into CSV format, you can use Great Expectations to validate the dataset against the pre-defined expectation suite. This ensures that the data being pulled into the pipeline meets the initial criteria.
- *Alerting and Remediation:* If the extracted data fails any expectations, Airflow can be set up to send notifications. This alerting mechanism provides immediate feedback, and based on its severity, you can either halt the pipeline for manual intervention or proceed with caution.

Quality Checks After Data Modeling:

- *Expectation Suites for Modeled Data:* Modeled data, especially when being prepped for analytics and dashboarding, has its own set of criteria. You'd want to ensure, for instance, that aggregations are computed correctly, foreign keys are consistent, or certain metrics don't exceed predefined thresholds.
- *Validation After Modeling:* Once the data transformation and modeling with Spark is completed, and the data is written back in Parquet format, you can validate this modeled data against its specific expectation suite using Great Expectations.
- *Feedback Loop:* If the modeled data doesn't meet the expectations, it's essential to understand why. The discrepancies could arise due to changes in source data patterns, issues within the transformation logic, or even external factors affecting the data. Having this validation step can act as a safeguard, ensuring only quality data makes it to the end-users.

Deep Dive into Database Design

A deep dive into a data model, especially one using a star schema, involves understanding the core fact tables and their surrounding dimension tables. In our context, given the entities of orders and customers, here's what the data model might look like:

Fact Table (Orders Fact Table):

- This table would capture transactional data, with each row typically representing an order.
- Attributes: OrderID (unique identifier for each order), CustomerID (foreign key to the Customers dimension), ProductID, DateID (foreign key to a Date dimension), Quantity, TotalPrice, and any other transaction-specific measures.
- Measures within the fact table, such as Quantity or TotalPrice, are the data points around which analytics will revolve, e.g., total sales, average order value, etc.

Dimension Tables:

- Customers Dimension: This table provides descriptive attributes about the customers.
- Attributes: CustomerID (primary key), CustomerName, CustomerEmail, DateOfBirth, RegistrationDate, Address, City, State, PostalCode, etc.
- This dimension provides context to the orders. For instance, it can help answer questions like "Which city has the highest number of customers?" or "What's the average order value for customers from a particular state?".
- Date Dimension: A commonly used dimension in star schemas, allowing for time-based analysis.
- Attributes: DateID (primary key), Day, Month, Year, Weekday, IsWeekend, IsHoliday, Quarter, etc.
- It supports queries like "What were the total sales in Q1 of the year?" or "How many orders were placed on weekends?".
- Product Dimension (assuming there are multiple products involved in orders): Descriptive details about the products.
- Attributes: ProductID, ProductName, Category, Manufacturer, UnitPrice, etc.
- Others: Depending on the business context and requirements, other dimensions like PaymentMethod, ShipmentDetails, or Warehouse might be relevant.

Relationships: In the star schema:

- The Fact table will have foreign keys pointing to the primary keys of the dimension tables.
- Joins between the fact table and the dimension tables are usually straightforward, designed for high-performance querying.

Queries & Analysis:

- The star schema is optimized for querying, so users can easily write queries that join the fact table with multiple dimensions.
- For instance: “Find the total sales by product category for customers from New York in January 2023” would involve joining the Orders fact table with the Customers, Date, and Product dimensions.

A deep dive would not only describe this schema structure but would also involve:

- Cardinality between tables, ensuring there aren’t any unexpected many-to-many relationships.
- Grain of the fact table: ensuring clarity about what each row represents.
- Data Lifecycle: Understand how frequently the data is loaded or refreshed in these tables, the retention policies, etc.
- Performance Considerations: Indexing strategies on frequently queried columns, materialized views for pre-aggregated data, and potential partitioning strategies for large fact tables.
- Data Quality Checks: Validations to ensure there are no orphan records in the fact table (i.e., records that don’t have corresponding entries in the dimension tables) and ensuring data consistency across tables.

This detailed review ensures that the star schema effectively supports the analytical needs of the business while maintaining performance and data integrity.

```
CREATE TABLE OrdersFact (
OrderID INT PRIMARY KEY,
CustomerID INT,
ProductID INT,
DateID INT,
Quantity INT,
TotalPrice DECIMAL(10,2),
FOREIGN KEY (CustomerID) REFERENCES CustomersDimension(CustomerID),
FOREIGN KEY (ProductID) REFERENCES ProductDimension(ProductID),
FOREIGN KEY (DateID) REFERENCES DateDimension(DateID)
);
```

```
CREATE TABLE CustomersDimension (
CustomerID INT PRIMARY KEY,
CustomerName VARCHAR(255),
CustomerEmail VARCHAR(255) UNIQUE,
DateOfBirth DATE,
RegistrationDate DATE,
Address VARCHAR(255),
City VARCHAR(100),
State VARCHAR(50),
PostalCode VARCHAR(20)
);
```

```
CREATE TABLE DateDimension (
DateID INT PRIMARY KEY,
Day INT,
Month INT,
Year INT,
Weekday VARCHAR(10),
IsWeekend BOOLEAN,
IsHoliday BOOLEAN,
Quarter INT
);


CREATE TABLE ProductDimension (
ProductID INT PRIMARY KEY,
ProductName VARCHAR(255),
Category VARCHAR(100),
Manufacturer VARCHAR(150),
UnitPrice DECIMAL(10,2)
);
```

Please note that the above are simplified schema definitions for demonstration purposes. In a real-world scenario, there would be other design considerations:

- Indexes would be added to frequently accessed columns to optimize query performance.
- Depending on the DBMS, constraints like CHECK constraints or additional UNIQUE constraints might be added.
- Some fields might be further normalized into other reference tables.
- The data types used, especially VARCHAR lengths, may vary based on exact data requirements.
- The use of sequences or auto-increment properties for primary key generation.

You would also likely have more attributes in each table, depending on the granularity and specificity of the data you're working with.

Wrap up and consider scalability

Scalability and Fault Tolerance:

The chosen technologies inherently support scaling. For instance:

- *Airflow*: Can be deployed with CeleryExecutors that distribute task execution across worker nodes, allowing for scaling out as the number of tasks increases.
- *Spark*: Designed for distributed data processing, it can efficiently scale out by adding more nodes to the Spark cluster.
- *Amazon S3:* Built to store and retrieve any amount of data at any time, it's inherently scalable.
- *Redshift Spectrum:* Uses the power of the Redshift cluster combined with Amazon S3, allowing querying large amounts of data in S3 directly.

Fault Tolerance:

- *Airflow*: With CeleryExecutors, even if one worker node fails, tasks can be rerouted to other nodes.
- *Spark*: If a node fails during a job, Spark can recover lost data from lineage information.
- *Amazon S3:* Highly durable with 99.999999999% (11 9's) of durability, ensuring data is safe.
- *Redshift Spectrum*: As it works on top of S3, it benefits from S3's high durability.

Monitoring and Alerting:

Monitoring:

- *Airflow*: Comes with a rich UI that shows DAG runs, task status, and logs. You can easily spot failures or performance bottlenecks.
- Tools like *Prometheus* can be integrated with Airflow, Spark, and other components to pull metrics.
- AWS services like S3 and Redshift offer native monitoring tools within the AWS Management Console.

Alerting:

- *Airflow*: You can configure alerting mechanisms that notify stakeholders via email or other channels when tasks fail, or even when they succeed.
- Integrations with platforms like PagerDuty or Slack can offer immediate notifications to relevant teams.
- AWS services provide CloudWatch Alarms which can alert based on various metrics or anomalies.

Documentation:

- *Pipeline Design and Flow:* All stages of the pipeline, from extraction to modeling, should be thoroughly documented. This includes data sources, transformation logic, the rationale for design decisions, and any known limitations or areas for improvement.
- *Metadata Management:* Tools like Apache Atlas can be used to provide a comprehensive view of data lineage, helping users understand data origins and transformations.
- *Code Documentation:* Ensure that all code, be it DAGs in Airflow or Spark jobs, is well-commented. This aids future maintainers and other data engineers who might work on the system.
- *User Documentation:* For users of the reporting dashboards, provide clear explanations of data sources, update frequency, key metrics definitions, and any other details necessary to interpret the data correctly.