Pizza Restaurant System Design (AWS)

1. Requirements and Goals of the System

Requirements

- Handle large write volume: Billions of write events per day.
- Handle large read/query volume: Millions of merchants wish to gain insight into their business. Read/Query patterns are time-series related metrics.
- Provide metrics to customers with at most one hour delay.
- Run with minimum downtime.
- Have the ability to reprocess historical data in case of bugs in the processing logic.

Goals

- A customer can order pizza either online or in-store
- Online orders still need to be fulfilled by a store
- An item cannot be ordered if there is not enough required inventory
- Low inventory will trigger raw material ordering from dedicated suppliers
- A sale only completes if the order can be fulfilled
- After a sale is made, the transactions marked “delivery” will be sent to delivery partners, and the inventory will be updated accordingly
- Data from the operational sales database will be forwarded to an analytical database for further analytics by other teams on hourly basis

2. Capacity Estimation and Constraints

- hundreds of GB data per month

- 2000 stores (also assuming their business runs 8 hours per day)

- Assuming the billions of write events do not distribute evenly throughout the 2000 stores because some stores may be more rural than the others, some stores will have more frequent read/write than the others

- Assuming the 20:80 rule, say 80% of 1 billion writes per day come from 20% of the stores, then that’ll be
    - 1 billion x 80% = 800 million writes per day come from 2000 x 20% = 400 stores

- Each write event from a sale consists of the following fields (total = 58 bytes):

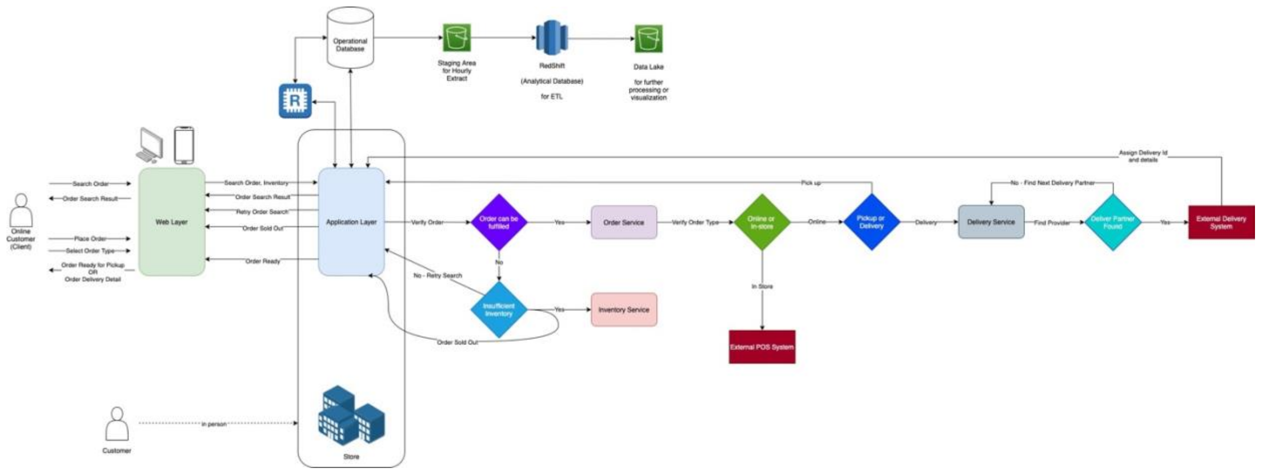
    - store_id (4 bytes)
    - order_id (4 bytes)
    - product_id (4 bytes)
    - transaction_ts (10 bytes)
    - quantity (4 bytes)
    - unit_price (4 bytes)
    - total_transaction_amt (4 bytes)
    - delivery_id (4 bytes)
    - delivery_type (20 bytes)

- If we have 800 million writes per day from the busiest stores, that would be at least 45 GB of storage per day (1350 GB per month), so over a year, that'll be 17 TB of data

- Using the golden 20:80 rule again and in favor of handling trying times like a COVID pandemic, we'll assume 80% of the writes are from online and 20% from in-store. That translates into 640 mil from online, and 360 mil from in-store. Let's see how much bandwidth is needed for either case:
    - 640 million per day / 8 hour * 60 minute * 60 sec = 222K writes per sec
    - 360 million per day / 8 hour * 60 minute * 60 sec = 125K writes per sec but can only be synced every 15min, so 6.5 GB per 15min

- The online transactions would be flowing in real time whereas the transactions made by POS can be synced every 15 minutes in batches

3. High Level Design



- **Web layer**: interface to users to make requests

- **Application layer**: handles responses to the requests after the Order Service finishes verifying the order

- **Order Service**: verifies whether an order can be fulfilled by checking against the Aurora operational database whether the products exist and by inquiring the Inventory Service for sufficient inventory. If an order can be fulfilled, it labels the order type to be "online" or "in-store". If in-store, send to the external POS system, and Lambda function to be triggered to ingest these POS transactions back to Aurora DB every 15min. If online, send through Kinesis Data Stream which is being consumed by a Delivery Service. If the Online order is labelled to be "delivery", the Delivery Service will find a delivery partner and assign a Delivery id to the transaction. Lambda functions will be consuming transactions from Kinesis Data Stream to insert them into Aurora DB.

- **Inventory Service**: verifies whether there is a sufficient inventory to fulfill an order. If there is sufficient inventory, confirm with Order Service to proceed to payment, and update the Aurora DB with the new inventory count per raw material via Lambda. If there is insufficient inventory, feedback to Application Layer to retry another order. (Improvements: we could set a threshold for the inventory count so that when it is under, the associated supplier will be contacted and put in more orders for raw materials).

- **Delivery Service**: handles finding a delivery partner for the given order. If a delivery partner is found, it will forward the order event to the 3rd party provider and update the Aurora DB's order_fct table with the new delivery Id via Lambda. If no partner found, it will keep retrying for the next best delivery partner. (Improvements: there is a risk that the "next best" delivery partner is not realistically nearby, so there should be a logic to deny "delivery" option if the "next best" delivery partner would have charged too much delivery fee or must come from a beyond-threshold distance).

4. Database Design

- Since these writes are transactional where a sale can only be made once a product confirms existence and sufficient inventory, a relational database such as RDS or AWS Aurora will be used. We will go with AWS Aurora because it is the best supported in AWS to handle 600K of reads per sec and 200K off writes per sec within just one instance. It is designed for high-velocity data, scales fast, has fast disaster recovery (minimum downtime), and even allows analytical workload via parallel query.

- The parallel query function would allow merchants who wish to see the most recent records quickly.

- We’re assuming POS is an external system processing in-store transactions after our Order Service confirms the order can be fulfilled. Because the POS batch job, 6.5 GB per 15 min, is still well under the AWS Lambda memory allocation limit (10 GB), we can utilize it to run batch jobs from the POS system into our Aurora operational database upon a 15min Event Trigger.

- Between Aurora and the Application Layer, the in-memory database, ElasticCache Redis will offload some hot product, hot inventory or hot delivery partner read requests.

- Online real-time transactions can be fed through Kinesis Data Streams, and Lambda functions will poll these data to send to Aurora DB.

5. Data Indexing and Sharding

- The store_id will be the partition key in Kinesis for efficient processing of data. For the 400 busier stores, we would have to split shards in Kinesis for their online transactions to prevent ProvisionedThroughputExceeded Error. Using store_id is bound to have the hot partition problem, but because we can do resharding, it’s better to manage these busier stores’ data together than using random partitioning for even distribution at the cost of having to fetch the same store’s data across different shards.

- During busier times of the year, we would have to split more shards for the already-busier stores. Conversely, if it’s calmer times of the year, we can merge those shards to scale in the processing (we’re paying for per shard after all!) (Improvements: an additional administrator application will be deployed to monitor and manage the resharding).

- Since the access pattern is time-series based, the primary index should be the transaction_ts and because we would most likely analyze by stores whether it’s the merchants or the analytical employees, secondary index can be the store_id.

6. Analytical Workload

- Even though Aurora’s parallel query function is very enabling, it is not intended to specialize in OLAP workload. For this reason, we will have an ETL job behind the Aurora DB to pull raw operational data to a staging area in S3 on hourly basis. AWS RedShift will pick up the raw data from the staging area, implement transformation logic, and load them to S3 Data Lake for additional analytical workload.

- In the event of error in ETL processing logic, raw data still exist in the S3 staging area to be re-processed and re-loaded into AWS RedShift and Data Lake.

7. Security

- Individual IAM roles will be created for each AWS service to access resources. Stores might decide to get their own accounts to manage their own bill payments, so cross-account access might need to be set up using Trust Policy as well.

- Wherever Lambda needs to interact with external API (ex, external POS and delivery systems), it will go through NAT Gateway to gain internet access.