Backend ingestion into Data Warehouse

Problem statement

There is a backend that serves as the check-out component of a ticket sales platform that is built along the principles of micro-services-based architecture. We would like to retrieve the data produced by the operation of this service and to make it usable for analysis.

Conclusion

A system to ingest the check-out event has been proposed that is optimized for scalability and high-performance. It consists of 4 main components:

- Ingestion from API into a NoSQL storage solution, done via dockerized Python on Airflow, in daily or even hourly batches assuming very large quantities of data; Kafka is also a high-performance option with many advantages, but it's over-kill when batch-processing is sufficient
- Storage into DynamoDB for ultra-high performance, hassle-free key-value store that can ingest large quantities of data without any issues and keep it for further processing. Data replicated into S3 and stored in the Glacier tiers for many years, such that additional fields can be retrieved from events that have already passed without having to pay large storage costs
- Processing the raw events from the NoSQL store into a tabular format also done via batches, using dockerized Python, where the payload of the events is processed, relevant fields retrieved, organized, and stored according to entities that emerge from applying domain-driven design, such as tickets, customers, attractions.
- Data Warehouse where the processed data is ready for analysis, and is organized according to a model that is a hybrid between Kimball and Data Vault 2.0

The solution could be simplified e.g. by using S3 to store the raw data at the cost of performance.

Considerations

This system is composed of a few parts:

- one that retrieves the events from the backend API and stores them (ingestion)
- the storage (ingestion storage)
- the mechanism for processing the raw events data and uploading them to the next step (data processing)
- the final part, the data warehouse (data warehouse)

The selection of all these parts depends heavily on the size of the data and performance requirements, as well as monetary constraints. Since the job application mentioned the desire for 100x scaling, I will assume performance matters more than keeping costs low

Ingestion

Since the architecture seems to be micro-services, I assume that there is already an API that is used to communicate events.

Most often these events will be in the form of nested JSONs and the company already has a method of communication between the micro-services set up. We can assume that each event will have some standard keys, like “event_name”, “event_id”, “event_timestamp” and a variable “payload” which will be its own JSON within the main JSON, and which will be specific to the event – for example “ticket_id” and “sale_timestamp” and “sale_price” for an event that goes into the tickets micro-service, or “transaction_id”, “user_payment_details” and “amount_to_be_charged” for an event that feeds into the payments microservice. I will therefore assume that the events follow a generic format:

```
{
  "event_id": 123456,
  "event_name": "ticket_sale",
  "event_timestamp": "2025-03-27Z20:48:37",
  "payload": {
    "key_1": "value_1",
    "key_2": "value_2"
  }
}
```

Depending on the downstream requirements, we could ingest these events via streaming as soon as they happen:

- Kafka is a high-performance, heavy-duty option for this; RabbitMQ is also used sometimes
- AWS SQS for a simpler one if the messages are small enough;
- AWS lambda is also an option assuming there are not that many events to be processed – if we process thousands of events per hour it might become expensive; this solution would also require an intermediary that triggers the lambda function when an event is created. This could be done with AWS EventBridge, which as far as I know is the only AWS service that can take as an input a third party event bus.

However, the task mentions that this data is meant for analytical purposes, in which case batch-processing makes more sense due to being cheaper, easier to implement and maintain, easier to recover from in case of failure.

The frequency of the batches depends on business requirements as well as size, but normally in analytics data of d-1 is good enough (so e.g. on the 26th of April 2025, you can process all the data up to and including the full day of 25th of April 2025), so I will assume daily batches. If the data is too much, these can be broken down to hourly level and each hour processed in parallel for extra speed. We have multiple options for running these daily batches:

- Airflow is the best option, paired with e.g. a dockerized Python script that reads the data from the API using the date inputs (managed by Airflow) to retrieve the correct batch of data (e.g. between `timestamp_interval_start` 2025-04-19Z17:00:00 and `timestamp_interval_end` 2025-04-19Z18:00:00) or even a pre-made API ingestion operator where we simply need to pass the API details and that handles the ingestion for us. These might exist for popular APIs, but especially for something developed in-house, Python probably makes more sense. The advantages of using Airflow for this include seamless batch-management, strong idempotency capabilities, but also that it can work with any data size and can be made very fast depending on the resources made available.
- We could also use AWS Lambda assuming the size of the data is not too large (this could be overcome by simply making the batches more frequent), which can be scheduled very conveniently in the AWS UI.

It might make sense to run the event extraction at night, when there is limited customer traffic, such that the API does not get overloaded. Other approaches could be, e.g. to make a replica of the micro-service's database and to read the data from here, or even read directly from the database if peak traffic is avoided.

This step could already do some very basic sorting of the events based e.g. on the event name, and this could help with keeping all the events for one target micro-service grouped. That way, the subsequent processing steps would be more efficient.

In any case, the ingestion script would simply retrieve from the check-out API all the events that fall within a certain batch defined by a given period of time (e.g. everything that happened between 17:00:00 and 18:00:00 on a given day) and write them into the ingestion storage, potentially partitioned by date and/or topic.

Ingestion storage

This is the component into which the ingestion script would write the data after retrieval from the API. This step could be skipped if keeping low-cost was more important, but it makes sense if we want high-performance, because the most suitable storage system is a NoSQL database like AWS DynamoDB that is optimized to work with data structured into key-value stores, like JSON objects, which is what events usually are (and what we've assumed is our case). AWS

DynamoDB is a perfect option because it can scale really well and has been designed for ultra-high reliability, availability, speed, etc., with the downside being costs. Assuming we've included some sorting in our ingestion script, we can assume that the data is now split by categories, which will make the subsequent step quicker and more efficient.

Choosing a key-value store would work well for processing this data relatively quickly, and is therefore perfect for feeding into the next step. However, there are cheaper ways to store this data long term, which might be needed as per the task details/requirements. In case we'd like to retrieve even more data from a given event much later on, it is cheaper to store in something like AWS S3, especially in one of the glacier tiers. Those are extremely cheap for long-term storage, with the caveat that retrieving the data may take multiple days. If this delay in retrieving the data is acceptable, then this would be the cheapest option for designing the system for back-fills at a later date.

Depending on the data size and processing-speed requirements, it could also be worth storing the raw events into a simple object storage like AWS S3, into a datalake sort of setting. Since this option would involve storage that is all-purpose, it would be less efficient for processing the data as key-value objects, but if there are not many events, it is acceptable. Some advantages of this solution are decreased costs, ease of organizing the storage, ease of sending the data into the Glacier tier for long-term storage. Unless we have large amounts of data and/or a requirement for high-performance, I would prefer the datalake as an alternative to DynamoDB or another key-value store that's optimized for heavy-duty processing.

Another advantage of storing the data in something like AWS S3 or GCP GCS is that it could be connected to a serverless Data Warehouse like BigQuery or Redshift or maybe even Snowflake directly (I know for sure BigQuery has this capability) and then queried seamlessly. Since these DWH solutions have great functions for working with arrays or other forms of nested data, there may be no need to process the data even further; or, it could be easier to process the data by reading from the DWH and writing back into it.

Nevertheless, since the theme is high-performance and scalability, the chosen option will be AWS DynamoDB instead of either datalake solution.

Data processing

There are multiple options for processing the raw events into data for analysis, and they depend on the size of the data as well as the storage method chosen; however, the most important element in this step is how to organize the data for storage and analysis, and it's where we need to discuss the data model. This should above all be determined through the principles of domain-driven design. Since we are selling tickets to attractions to customers, then some of our key entities will be customer, ticket, attraction, and then also relevant might be transactions

(e.g. if we want to analyze the behaviour of customers at check-out for further optimization), so we will create tables around these entities.

- A very popular approach is Kimball, in which we would create fact_ticket, dim_ticket, dim_customer, dim_attraction, fact_transaction, dim_transaction, where the name of the entity would determine the way data is stored, e.g. in fact_ticket, each row would have a unique id for the individual ticket sold. These denormalized datasets would then be joined by analysts to create more complex calculations
- Another good option is data vault 2.0, which would use the same entities, but it would split the data into many more tables that would be less driven by logic and more by developer convenience. Basically, for each entity we would keep a "spine" with all the unique, non-null ids, and then in satellite tables we'd store the characteristics associated with these facts. The advantage of this approach is that any number of features can be added in one given sprint, and further data added without changing what was previously there, allowing for only-forwards development. The disadvantage of this approach is that it would require a lot of joins, which can be expensive and slow, and that it would require extra work to make data that is suitable for analysts to work with.
- The final option would be one-big-table per entity, which works well in columnar stores like BigQuery, and whose advantages include the fact that expensive join operations would be avoided. This is also a good option for working with nested arrays of data, like the payload of each individual event, and could be nice e.g. if we wanted to store all the tickets of one customer in the same row in the dim_customer table.

My preferred option here is a hybrid between Kimball and data-vault 2.0, because Kimball is very intuitive and leads to very clear relationships within the data that make it easy to figure out how to manage nulls, how to join, etc., and data-vault 2.0 offers great flexibility to the developers to build forwards without having to change what's already there.

Next comes the question of how to process the raw data into the desired format:

- Since we are storing it in DynamoDB in the main scenario, we can retrieve it from here with a Python script that also works in batches (daily or hourly) which simply reads the events and retrieves pre-determined information from the payload part of the message, depending also on e.g. some rules in the event titles. For example, if the event name contains "ticket", then from the message payload we need to retrieve the customer_id, ticket_id, sales_price, attraction_id, sales_timestamp, day_of_visit, while from messages that contain "trigger_payment" in the title we retrieve the transaction_id, transaction_amount, payment_processing_system. The events could be processed one-by-one, and sorted based on the title, and then having the payload be processed according to the rules. If the events are pre-sorted in the storage of the ingestion layer,

we can even do the processing in parallel, which would be faster – this would be extra desirable if speed and high-performance are a requirement

- If the events are not very large, they could also be processed as the messages of a queue. For example, a very lightweight build for this entire task could be to use AWS lambda to read the event from the back-end and then already retrieve the relevant, analytics-oriented data from it and write it directly to a data warehouse. Processing the data event-by-event would be very nice and easy to implement, but not very-scalable. Assuming the events are relatively small, they could also be processed with Kafka from the backend straight into clean analysis data into a DWH.
- A setup like dbt + BigQuery could also be used to do the processing: since BQ has great functions for working with arrays and nested fields, we could simply query the data from there by unnesting, and writing the results of these queries as clean data back into BQ.
- Various methods for parallel processing also exist, that would e.g. use partitions by event_name or event_id to split the events and crawl the payload data faster

The processing batches could also be run with Airflow, which has some very nice functionalities for back-filling data. This would be very helpful for e.g. reprocessing 2 years' worth of data over the weekend in case we've been asked to retrieve more fields from events that have already happened.

Indifferent of the option chosen above, the main idea of this step is to take the "un-structured" data of the raw events and to organize it into structured, relational tables using a data model, which is to be loaded into a data warehouse that is different from the previous storage method precisely because this one is optimized for relational data.

The Data Warehouse

The data warehouse is simply the place where the processed, analysis-ready data "lives" and awaits discovery from the analysts. Many serverless solutions exist these days, such as BigQuery, Redshift, and arguably Snowflake too. These products are nice because they are ready to use out of the box, but cheaper solutions likely exist, such as more classical databases like MySQL, PostGRES etc.; one solution I liked a lot from a previous job was a data lake-house set-up, where the data was stored in AWS S3, converted into tables via Glue, and then queried via a third-party query engine, in that case Photon. This set-up is nice because it's very flexible, allows storing all the data in one place, and the individual components can scale as needed; the query engine can also be changed without having to move the data, and separate query engines can even be provided to different teams depending on their use-cases. Nonetheless, the convenience of using an out-of-the-box solution like BigQuery or Snowflake is often worth the extra costs, especially if no extra flexibility is required and the solution is only aimed at analysts.