SYSTEM DESIGN - STREAM PROCESSING

System Design Question and Solution: Streaming

One common type of data engineering system design question is the streaming data 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 near real time data ingestion pipeline to ingest and model iOS app engagement for use in rear real time dashboards.

Understand the data requirements and establish scope

Data Characteristics:

- What types of engagement metrics are we tracking (e.g., clicks, swipes, session length)?
- How is the timestamp information captured and stored for each event?

Data Sources:

- Are the data coming directly from the iOS app or are there intermediary systems like analytics tools involved?
- Are there any third-party integrations or SDKs that the app uses which might produce engagement data?

Volume and Velocity:

- What's the estimated number of events per second/minute/hour?
- Do we expect any bursts in data, such as during promotions or peak user activity hours?

Transformation and Processing:

- Do we need to enrich the data with additional information, perhaps from other systems?
- Are there specific transformations required, like aggregations or filtering?
- Are there any real-time analytics or algorithms to be applied on the data?

Destination and Usage:

- What are the downstream systems or databases where this data needs to be stored?
- Who are the consumers of this data? Is it only for the dashboard, or are there other systems/users?

Historical Data:

- Do we need to migrate any historical app engagement data into this new pipeline?
- How far back does the historical data go, and what's its volume?

SLAs and Latency:

- What’s the maximum acceptable latency from the time an event occurs in the app to when it appears in the dashboard?
- Are there any service level agreements (SLAs) in place regarding data availability and freshness?

Data Security and Compliance:

- Are there any sensitive data points being captured (e.g., personally identifiable information)?
- Are there any compliance standards we need to adhere to, such as GDPR or CCPA?

Scalability and Future Growth:

- What’s the projected growth rate for the app’s user base in the next year or two?
- Are there plans to introduce more metrics or event types in the future?
- Do we foresee any architectural changes to the app or the analytics tools in use?

Propose high-level data flow and get buy-in

The data flow initiates with the iOS app producing engagement events. These events are then sent to a collection layer, potentially comprising an API Gateway or directly to Kafka Producers. These producers publish data to Kafka topics. Depending on the event type or source, different topics might be used. Kafka’s distributed commit log ensures that data is ingested in real-time and provides durability through replication across multiple Kafka brokers.

Once inside the Kafka ecosystem, the events can be consumed by Kafka Streams or Kafka Consumers for processing. Kafka Streams allows for stateful and stateless operations, enabling tasks such as filtering, aggregation, or joining data streams. For more complex processing, the data can be consumed by other stream processing frameworks like Apache Flink. These processors enrich, aggregate, and transform the raw engagement events into a format suitable for analytics and visualization. After processing, the resultant data can be published back into another Kafka topic, segregating raw and processed data.

Post-processing, the transformed data can be consumed by a data sink or storage system, like Amazon S3 or a real-time database like Apache Druid. This can be easily done using Kafka Connect, which provides connectors to various storage systems, ensuring seamless data flow. These storage systems serve as the data backend for real-time dashboards.

This high-level Kafka-based pipeline ensures a robust, scalable, and near real-time data flow, ideal for modeling iOS app engagement for real-time dashboards.

Dive deep into the data pipeline design

Data Extraction: The iOS app serves as the primary producer, emitting engagement events directly to Kafka topics. As user interactions occur within the app — such as an app installation, a session start or end, or a notification view — the app captures these events and sends them to dedicated Kafka topics. Specifically, “install”, “session”, and “view_notification” topics might be used to segregate events by type. The use of distinct topics allows for easy categorization, consumption, and parallel processing, catering to potentially different data handling needs for each event type.

Data Storage: After the initial ingestion into Kafka topics, the raw data remains stored in Kafka’s distributed log system, ensuring durability, fault-tolerance, and scalability. Once processed by the Flink jobs, the transformed and modeled data is then synced to Amazon S3 using Kafka Connect with S3 connectors. S3, being a highly durable storage solution, is used to store this data in the Parquet format. Parquet, a columnar storage format, offers advantages such as data compression and schema evolution, making it ideal for analytical querying and serving as the backend storage for dashboards.

Data Transformation: Apache Flink, integrated with Kafka Consumers, fetches the raw events from the Kafka topics. For session-related tasks, Flink handles “sessionization” by grouping sequences of related user activity events, determining session starts and ends based on timestamps or user inactivity gaps. This allows the identification and computation of metrics like session duration, user paths, and bounce rates. Apart from session-related transformations, Flink can perform event enrichment, filtering, and aggregation as needed.

Data Modeling: Within the Flink jobs, after basic transformations, the data can be further modeled to create user engagement metrics. These might include daily active users, average session lengths, notification click-through rates, etc. Such metrics can be created by aggregating event data over specific windows (e.g. hourly) and then outputting these aggregated results. The resultant modeled data is then published to another Kafka topic or directly sent to S3, depending on the integration approach.

Quality Checks on Raw Data: Before processing, it’s vital to ensure the integrity of the raw data. As the iOS app produces events, validation checks can be implemented to ensure mandatory fields like user_id or timestamp are present. Kafka itself can be configured with log compaction to maintain only the latest event for a specific key, ensuring outdated or duplicate events are minimized. Monitoring tools like Kafka’s built-in Consumer Lag Monitoring or external solutions like Confluent Control Center can track if all produced messages are being consumed, indicating potential data loss.

Quality Checks on Processed Data: Post-transformation, the data's quality must again be validated. Within the Flink job, anomaly detection algorithms can highlight unexpected spikes or drops in metrics. Consistency checks, comparing the sum of sessionized durations against raw events, can ensure no sessions are missed or duplicated. After syncing to S3, tools like Apache Deequ or AWS Glue can be used for further data quality assessments, verifying metrics like uniqueness, completeness, and compliance against predefined benchmarks.

The design of your data model in S3, particularly if you're storing in Parquet format (a columnar storage file format optimized for analytics), would hinge on the analytics and querying patterns you anticipate, the nature of the data, and the tools you intend to use for querying and analysis. Here's a potential data model structure:

Bucket Structure:

- A dedicated S3 bucket for the analytics data.
- Use S3 prefixes (which function similarly to folders) to logically separate different types of data, such as raw/, processed/, and aggregated/.

Partitioning:

- For optimized querying and cost savings, partition the data. This is particularly useful if you anticipate filtering data by specific dimensions during analysis.
- Common partitions include temporal dimensions such as year, month, day, and hour. This helps in quicker data retrieval and reduces the amount of data scanned during queries.
- Example structure: processed/app_events/year=2023/month=09/day=24/hour=14/

Data Files:

- Use Parquet format for the stored files. It's compressed by default and provides efficient columnar access, making it ideal for analytic queries.
- Ensure that the Parquet files are of optimal size, often between 64MB and 1GB, to utilize distributed querying engines efficiently.

Schema Design:

- Common fields might include: event_id, user_id, session_id, event_type (e.g., install, session, view_notification), timestamp, device_info, geo_location, and any other metrics or dimensions pertinent to your analytics.
- With Parquet, you can evolve the schema over time (add optional fields) without much hassle.

Metadata & Data Cataloging:

- Consider integrating with a data cataloging tool like AWS Glue. This helps in discovering, querying, and managing the dataset in S3.
- Storing metadata, like the creation time of the file, source, or any transformations applied, can be beneficial for auditing and tracking data lineage.

Data Retention and Lifecycle Policies:

- Implement S3 Lifecycle policies to transition older data to cheaper storage classes (like S3 Infrequent Access) or archive/delete it after a certain period.

Consistency and Data Lake Hygiene:

- Ensure that data is written to S3 in an atomic manner. It's a common practice to first write data to a temporary location and then move it to the final location to avoid any inconsistencies during reads.
- Regularly clean up any small files or incomplete partitions to maintain the health of the data lake.

By structuring your S3 data model effectively, you ensure that the data is easily accessible, queryable, and managed efficiently over its lifecycle. It provides a foundation for robust and scalable analytics operations.

Deep Dive into Kafka Setup

To ensure our Kafka system is easily scalable, especially since we anticipate the need to horizontally scale producers or consumers, we must consider configurations around topic partitioning and event keying.

Topic Partitioning:

- *Number of Partitions:* Start with a higher number of partitions than required. This allows adding more consumers in the future. Each partition can be read by only one consumer instance of a consumer group at a time, so having more partitions gives you the flexibility to increase the number of consumer instances as needed.
- *Partitioning Strategy:* While Kafka automatically distributes messages across partitions, for certain use-cases, you may want to ensure certain messages land in the same partition (e.g., all events of a specific user). This requires choosing an appropriate key (see event keying below).
- *Repartitioning:* Increasing the number of partitions for existing topics is possible but requires careful planning as it can impact existing consumers and the order of messages. Also, reducing the number of partitions is not directly supported in Kafka.

Event Keying:

- *Key Selection:* If the order of events is crucial (e.g., ensuring all events for a user are processed in order), use a consistent key like user_id. All messages with the same key will go to the same partition.
- C*ustom Partitioning:* For more advanced use-cases, Kafka allows you to implement a custom partitioner. This lets you define your logic for assigning messages to partitions. We will use the user_id so that all instances of a users events are in the same partitions, allowing for easier scaling.

Horizontal Scaling:

- *Brokers:* Add more brokers to the Kafka cluster as your throughput needs grow. This distributes the load and storage requirements. Newly created partitions can be assigned to these new brokers.
- *Producers:* Since producers are stateless, scaling them horizontally is straightforward. Just deploy more producer instances. Make sure they have proper load balancing if they're being fed data from another upstream system.
- *Consumers:* Use Kafka's consumer groups. As you add more consumer instances to a consumer group, Kafka will automatically rebalance the partitions among consumers. This makes it easy to scale out your consumers. But remember, in a consumer group, the number of consumer instances that can consume from a topic simultaneously is bound by the number of partitions that topic has.

Monitoring and Tuning:

- Keep an eye on key metrics like CPU, memory usage, and disk I/O of your Kafka brokers. Tools like the Confluent Control Center, Grafana with the JMX Exporter, or other monitoring solutions can help.
- Regularly review topic partition sizes, under-replicated partitions, and consumer group lag. If you notice a growing lag or skewed partition sizes, consider re-evaluating your partitioning and keying strategies.

Replication:

- Ensure you have replication set up for your topics. This not only provides data durability but also allows consumers to read from follower replicas, distributing the read load across multiple brokers.

By ensuring thoughtful configuration of partitions, judicious event keying, and regular monitoring, you can make your Kafka system resilient, scalable, and prepared for growth.

Deep Dive into Database Design

One way to model this data is the “one big table” schema design, the design philosophy typically shifts towards a wider table with each event type or metric represented as its own column. Such a schema design often simplifies querying because you only need to query one table, and it can also optimize storage and query performance for certain access patterns.

Here’s what a DDL might look like for a “One Big Table” approach for your daily aggregates:

```
CREATE EXTERNAL TABLE daily_aggregates_with_dimensions (
year INT,
month INT,
day INT,
hour INT,
device_type STRING,
app_version STRING,
country STRING, - You can break down location further if needed (e.g., region, city)
total_installs INT,
total_sessions INT,
total_views INT,
... - Any other metrics or dimensions you might want to add in the future

)


ROW FORMAT SERDE 'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
LOCATION 's3://your-bucket-name/aggregated/daily_aggregates_with_dimensions/';
```

With this design:

- For each day, you’ll have multiple rows corresponding to the unique combinations of device_type, app_version, and country. Each row will provide the aggregated metrics for that specific combination.
- If you wish to see how many installs happened on a specific day, from a particular country, on a certain device type, and for a specific app version, you would find a single row matching these criteria.

Benefits:

- Granular Analysis: With added dimensions, you can get insights at a more detailed level, helping answer questions like, “Which app version has the most views from iOS devices in the USA?”
- Optimized Storage: The columnar nature of Parquet still offers efficient storage, especially with repeating values in dimension columns.
- Segmented Analysis: It becomes easier to analyze trends within segments, such as how a specific app version is performing across different countries or on various device types.

Considerations:

- Increased Data Volume: With the added dimensions, the number of rows in the table will increase substantially. Ensure your query engine and storage are optimized to handle this volume.
- Complexity in Aggregation: The ETL or processing logic becomes a bit more complex since you need to aggregate data across multiple dimensions.
- Performance: Be judicious in selecting the granularity of your dimensions. For instance, adding very granular location details can exponentially increase the number of rows. Use higher-level aggregates when possible or filter down based on the most relevant dimensions for your analysis.
-

Hierarchical Aggregation:

- During the data processing, we will calculate aggregates at different hierarchy levels and store them together in the same table. For example, we might have one row for a specific device_type on a specific day, and another row for 'ALL_DEVICES' for that same day. This method allows for quick querying but increases storage requirements and adds complexity to the ETL process to ensure accurate calculations.

After creating this table, we would also need to periodically run partition discovery commands (like MSCK REPAIR TABLE app_events;) to ensure that new partitions are recognized by the system. We could schedule this to run every hour using AWS Lambda.

Wrap up and consider scalability

Scalability and Fault Tolerance:

Kafka Clusters: Deploy a multi-broker Kafka cluster. Distributing data across multiple brokers ensures that data remains available even if a few nodes fail.

- Partitioning: Ensure that Kafka topics are partitioned appropriately. Having multiple partitions allows for concurrent processing by multiple consumers. Choose a suitable partition key based on your use-case to evenly distribute the data.
- Replication: Use replication in Kafka to make your data fault-tolerant. If one broker fails, replicas ensure that data is not lost.

Flink Cluster: Run Flink in a cluster mode, which can distribute the computation workload.

- Checkpoints: Enable checkpointing in Flink to periodically save the state of your application. In case of failures, Flink can resume from the latest checkpoint.
- State Backends: For stateful processing, use a robust state backend like RocksDB. It’s suitable for holding large states efficiently.

Scalable Storage: If your data eventually lands in a storage system like S3, ensure that the storage structure (like directory layout for partitioning) is scalable and optimized for query performance.

Monitoring and Alerting:

Built-in Metrics:

- Kafka: Monitor key metrics like broker status, number of messages, lag, throughput, and error rates.
- Flink: Monitor checkpoints, backpressure, throughput, JVM metrics, and task failures.

Integration with Monitoring Systems: Both Kafka and Flink can be integrated with popular monitoring systems like Prometheus, Grafana, and others.

Logging: Ensure comprehensive logging in your processing logic. Centralize logs using platforms like ELK (Elasticsearch, Logstash, Kibana) or Splunk for easy access and analysis.

Alerting: Set up alerting thresholds for critical metrics. For instance, if the Kafka consumer lag grows beyond a certain point or if there’s a failure in the Flink job, alerts should be sent out. Tools like Alertmanager (with Prometheus) or native capabilities in Splunk can be utilized.

Documentation:

Architecture Diagram: Provide a high-level architecture diagram illustrating the flow of data from the iOS app through the entire streaming pipeline.

Data Schema: Document the schema of the event data, including any transformations or enrichments applied during processing.

Operational Guide: Include details about:

- Starting, stopping, and scaling services.
- Handling failures and recovery procedures.
- Backup and restoration processes.

*Monitoring Guide: P*rovide documentation on the monitoring setup. Highlight the key metrics to watch, their ideal ranges, and potential troubleshooting steps if anomalies are detected.

Evolution: If there are expected changes or future enhancements to the system, document them. This can be useful for onboarding new team members and ensuring continuity in understanding the system's roadmap.