Batch & Stream Processing

Tejas Parikh (t.parikh@northeastern.edu)

CSYE 6225

Northeastern University

Introduction

- Data systems are complex.
- No one database to satisfy different needs to access and process data.
- Systems that store and process data can be grouped into two broad categories:
  - System of record
  - Derived data systems
- The distinction between system of record and derived data system depends not on the tool, but on how you use it in your application.

Systems of record

- A system of record, also known as *source of truth*, holds the authoritative version of your data.
- When new data comes in it is first written here.
- Each fact is represented exactly once.
- If there is any discrepancy between another system and the system of record, then the value in the system of record is (by definition) the correct one.

Derived data systems

- Data in a derived system is the result of taking some existing data from another system and transforming or processing it in some way.
- If you lose derived data, you can recreate it from the original source.
- A classic example is a cache: data can be served from the cache if present, but if the cache doesn’t contain what you need, you can fall back to the underlying database.
- Derived data is *redundant* and is commonly *denormalized.*

Types of Systems

- Services (online systems)
- Batch processing systems (offline systems)
- Stream processing systems (near-real-time systems)

Services – Online Systems

- A service waits for a request or instruction from a client to arrive.
- When one is received, the service tries to handle it as quickly as possible and sends a response back.
- Response time is usually the primary measure of performance of a service, and availability is often very important (if the client can’t reach the service, the user will probably get an error message).

Batch processing systems - offline systems

- A batch processing system takes a large amount of input data, runs a job to process it, and produces some output data.
- Jobs often take a while (from a few minutes to several days), so there normally isn’t a user waiting for the job to finish.
- Instead, batch jobs are often scheduled to run periodically (for example, once a day).
- The primary performance measure of a batch job is usually throughput i.e. the time it takes to crunch through an input dataset of a certain size.

Stream processing systems - near-real-time systems

- Stream processing is somewhere between online and offline/batch processing (so it is sometimes called near-real-time or nearline processing).
- Like a batch processing system, a stream processor consumes inputs and produces outputs.
- However, a stream job operates on events shortly after they happen, whereas a batch job operates on a fixed set of input data.
- This difference allows stream processing systems to have lower latency than the equivalent batch systems.

Batch Processing

Batch processing

- Batch processing is an important building block in our quest to build reliable, scalable, and maintainable applications.
- MapReduce, a batch processing algorithm published in 2004, was called “the algorithm that makes Google so massively scalable”.
- It was subsequently implemented in various open source data systems, including Hadoop, CouchDB, and MongoDB.

Problems Solved By Distributed Batch Processing

- The two main problems that distributed batch processing frameworks need to solve are:
- Partitioning
  - In MapReduce, mappers are partitioned according to input file blocks.
  - The output of mappers is repartitioned, sorted, and merged into a configurable number of reducer partitions.
  - The purpose of this process is to bring all the related data—e.g., all the records with the same key—together in the same place.
- Fault tolerance
  - MapReduce frequently writes to disk, which makes it easy to recover from an individual failed task without restarting the entire job but slows down execution in the failure-free case.
  - Dataflow engines perform less materialization of intermediate state and keep more in memory, which means that they need to recompute more data if a node fails.
  - Deterministic operators reduce the amount of data that needs to be recomputed.

Consider Batch Processing when

- Real-time transfers and results are not crucial
- Large volumes of data need to be processed
- Data is accessed in batches as opposed to in streams
- Complex algorithms must have access to the entire batch
- Tables in relational databases need to be joined
- The work is repetitive

Use Cases

- DevOps /SRE - Consolidate logs, metrics, etc. from multiple regions (data centers) into central location for analysis.
- Machine Learning
- Payroll
- Billing
- Bank Transactions
- Etc.

Stream Processing

Bounded vs. Unbounded Data

- The input for batch processing jobs is bounded—i.e., of a known and finite size—so the batch process knows when it has finished reading its input.
- However, a lot of data is unbounded because it arrives gradually over time.
- Batch processors must artificially divide the data into chunks of fixed duration: for example, processing a day's worth of data at the end of every day, or processing an hour's worth of data at the end of every hour.

Stream, Event, & Event Streams

- A *stream* refers to data that is incrementally made available over time.
- An *event* may be encoded as a text string, or JSON, or perhaps in some binary form.
- In streaming terminology, an *event* is generated once by a *producer* (also known as a publisher or sender), and then potentially processed by multiple *consumers* (subscribers or recipients).
- In a filesystem, a filename identifies a set of related records; in a streaming system, related *events* are usually grouped together into a *topic* or *stream*.

Pull (poll) vs. Push (notify)

- In principle, a file or database is sufficient to connect producers and consumers.
  - A producer writes every event that it generates to the datastore, and each consumer periodically polls the datastore to check for events that have appeared since it last ran.
  - This is essentially what a batch process does when it processes a day's worth of data at the end of every day.
- However, when moving toward continual processing with low delays, polling becomes expensive if the datastore is not designed for this kind of usage.
- The more often you poll, the lower the percentage of requests that return new events, and thus the higher the overheads become.
- Instead, it is better for consumers to be *notified* when new events appear.

Messaging Systems

- A common approach for notifying consumers about new events is to use a messaging system: a producer sends a message containing the event, which is then pushed to consumers.
- A messaging system allows multiple producer nodes to send messages to the same topic and allows multiple consumer nodes to receive messages in a topic.

Publish/Subscribe Model Considerations

- What happens if the producers send messages faster than the consumers can process them?
  - The system can:
    - drop messages,
    - buffer messages in a queue
    - apply backpressure (also known as flow control; i.e., blocking the producer from sending more messages
- What happens if nodes crash or temporarily go offline—are any messages lost?
  - Durability may require some combination of writing to disk and/or replication.
  - If you can afford to sometimes lose messages, you can probably get higher throughput and lower latency on the same hardware.

Direct Messaging From Producers To Consumers

- A number of messaging systems use direct network communication between producers and consumers without going via intermediary nodes.
  - UDP multicast is widely used in the financial industry for streams such as stock market feeds, where low latency is important.
  - Brokerless messaging libraries such as ZeroMQ and nanomsg take a similar approach, implementing publish/subscribe messaging over TCP or IP multicast.
  - StatsD use unreliable UDP messaging for collecting metrics from all machines on the network and monitoring them.
- Although these direct messaging systems work well in the situations for which they are designed, they generally require the application code to be aware of the possibility of message loss.
- The faults they can tolerate are quite limited: even if the protocols detect and retransmit packets that are lost in the network, they generally assume that producers and consumers are constantly online.
- If a consumer is offline, it may miss messages that were sent while it is unreachable. Some protocols allow the producer to retry failed message deliveries, but this approach may break down if the producer crashes, losing the buffer of messages that it was supposed to retry.

Message Brokers

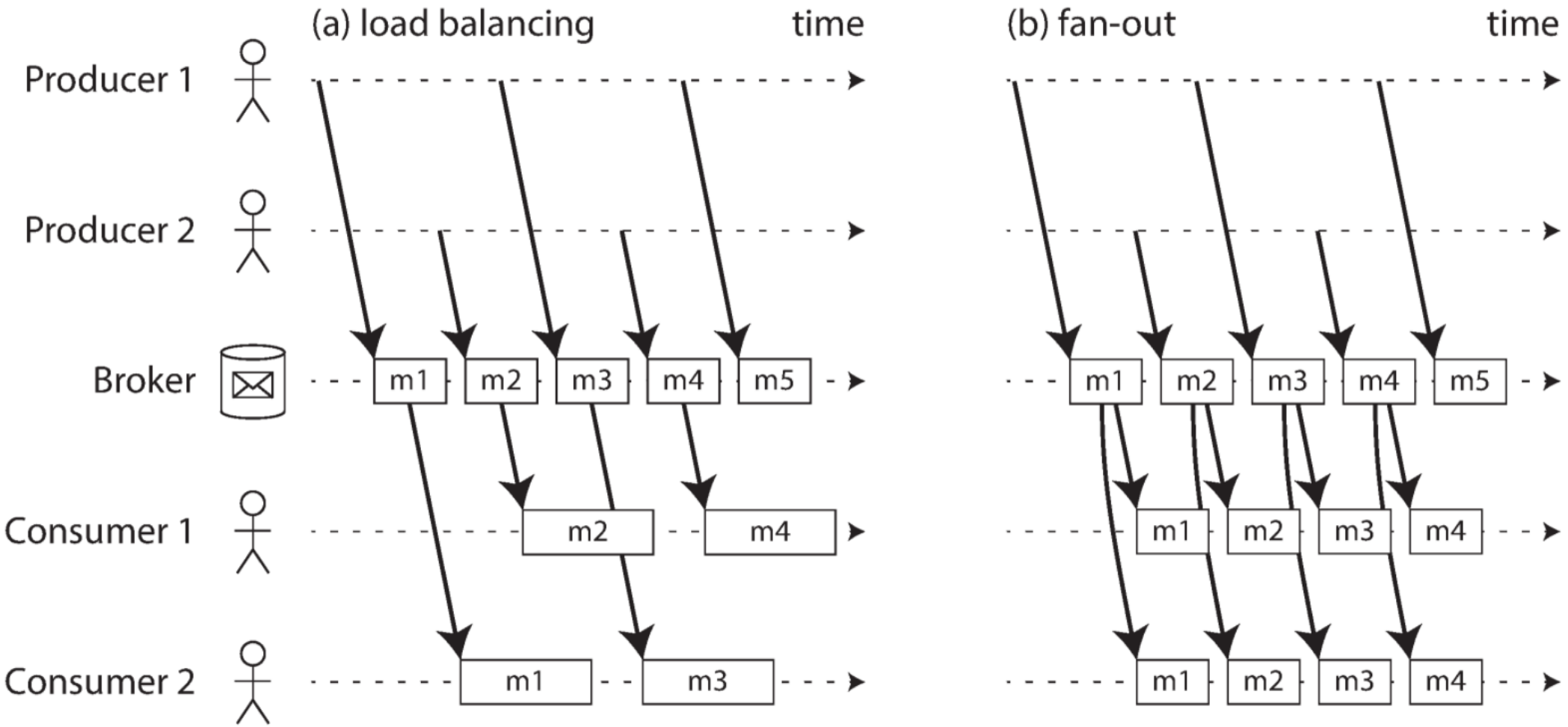
- Send messages via a message broker (also known as a message queue), which is essentially a kind of database that is optimized for handling message streams.
- It runs as a server, with producers and consumers connecting to it as clients.
- Producers write messages to the broker, and consumers receive them by reading them from the broker.
- By centralizing the data in the broker, these systems can more easily tolerate clients that come and go (connect, disconnect, and crash), and the question of durability is moved to the broker instead.

Queues

- Message brokers generally support unbounded queueing.
- A consequence of queueing is also that consumers are generally asynchronous: when a producer sends a message, it normally only waits for the broker to confirm that it has buffered the message and does not wait for the message to be processed by consumers.
- The delivery to consumers will happen at some undetermined future point in time—often within a fraction of a second, but sometimes significantly later if there is a queue backlog.

Multiple Consumers Patterns

- Load balancing
  - Each message is delivered to one of the consumers, so the consumers can share the work of processing the messages in the topic.
  - The broker may assign messages to consumers arbitrarily.
  - This pattern is useful when the messages are expensive to process, and so you want to be able to add consumers to parallelize the processing.
- Fan-out
  - Each message is delivered to all of the consumers.
  - Fan-out allows several independent consumers to each “tune in” to the same broadcast of messages, without affecting each other—the streaming equivalent of having several different batch jobs that read the same input file.
- The two patterns can be combined: for example, two separate groups of consumers may each subscribe to a topic, such that each group collectively receives all messages, but within each group only one of the nodes receives each message.



(a) Load balancing: sharing the work of consuming a topic among consumers; (b) fan-out: delivering each message to multiple consumers.

Acknowledgments And Redelivery

- Consumers may crash at any time, so it could happen that a broker delivers a message to a consumer, but the consumer never processes it, or only partially processes it before crashing.
- In order to ensure that the message is not lost, message brokers use acknowledgments: a client must explicitly tell the broker when it has finished processing a message so that the broker can remove it from the queue.
- If the connection to a client is closed or times out without the broker receiving an acknowledgment, it assumes that the message was not processed, and therefore it delivers the message again to another consumer.

Consumer Offsets

- Consuming a partition sequentially makes it easy to tell which messages have been processed: all messages with an offset less than a consumer's current offset have already been processed, and all messages with a greater offset have not yet been seen.
- Thus, the broker does not need to track acknowledgments for every single message—it only needs to periodically record the consumer offsets.
- The reduced bookkeeping overhead and the opportunities for batching and pipelining in this approach help increase the throughput of log-based systems.
- If a consumer node fails, another node in the consumer group is assigned the failed consumer's partitions, and it starts consuming messages at the last recorded offset. If the consumer had processed subsequent messages but not yet recorded their offset, those messages will be processed a second time upon restart.

Processing Streams

- The one crucial difference to batch jobs is that a stream never ends.

Uses of Stream Processing

- Fraud detection systems need to determine if the usage patterns of a credit card have unexpectedly changed, and block the card if it is likely to have been stolen.
- Trading systems need to examine price changes in a financial market and execute trades according to specified rules.
- Manufacturing systems need to monitor the status of machines in a factory, and quickly identify the problem if there is a malfunction.
- Military and intelligence systems need to track the activities of a potential aggressor, and raise the alarm if there are signs of an attack.

Reasoning About Time

- Stream processors often need to deal with time, especially when used for analytics purposes, which frequently use time windows such as “the average over the last five minutes.”
- It might seem that the meaning of “the last five minutes” should be unambiguous and clear, but unfortunately the notion is surprisingly tricky.

Event Time Versus Processing Time

- There are many reasons why processing may be delayed: queueing, network faults, a performance issue leading to contention in the message broker or processor, a restart of the stream consumer, or reprocessing of past events while recovering from a fault or after fixing a bug in the code.
- Message delays can also lead to unpredictable ordering of messages.
- Confusing event time and processing time leads to bad data.

Whose Clock Are You Using, Anyway?

- Assigning timestamps to events is even more difficult when events can be buffered at several points in the system.
- For example, consider a mobile app that reports events for usage metrics to a server. The app may be used while the device is offline, in which case it will buffer events locally on the device and send them to a server when an internet connection is next available (which may be hours or even days later). To any consumers of this stream, the events will appear as extremely delayed stragglers. In this context, the timestamp on the events should really be the time at which the user interaction occurred, according to the mobile device's local clock.

Recommended Reading

Designing Data-Intensive Applications by Martin Kleppmann

Publisher: O'Reilly Media, Inc.

Release Date: March 2017

ISBN: 9781449373320