Lecture 21 **11/27/2017**

Next Lecture: Quiz review & project meetings

Streaming & Apache Kafka

What problem does Kafka solve?

Provides a way to deliver updates about changes in state from one service to another via a persistent queue mechanism.

Why do I want this?

In real systems, events happen asynchronously, and services are loosely coupled. Need a way to notify one part of the system that something has happened. E.g.,:

- a user clicked on an ad, so an advertiser needs to be charged, and an ad click table needs to be updated and ...

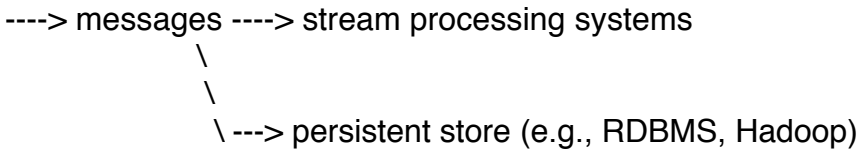
- some long running task completes and its result is ready to be consumed.

Why can't a database be used for this?

- It can be (and database vendors provide various notification services about events)

- But you don't always want to go into the database and back out -- that could add latency, and some of your services may not speak SQL

"Lambda architecture"



Don't other services do this?

Yes -- message queues or transactional queues have been used in big systems for a long time.

So what's the model?

(Show sample code slides

Producers "publish" data on particular *topics*.

Messages for a given topic are split into partitions (either round robin or using hashing).

Brokers store the messages for each partition in log files. (Show architecture slide)

Each message is identified by its offset into the log (Show log structure slide)

Multiple brokers serve a single topic -- partitions can be replicated across brokers.

Consumers read topics from brokers.

Brokers are stateless -- consumers are responsible for keeping track of what they what they have read -- enabled by use of offsets!

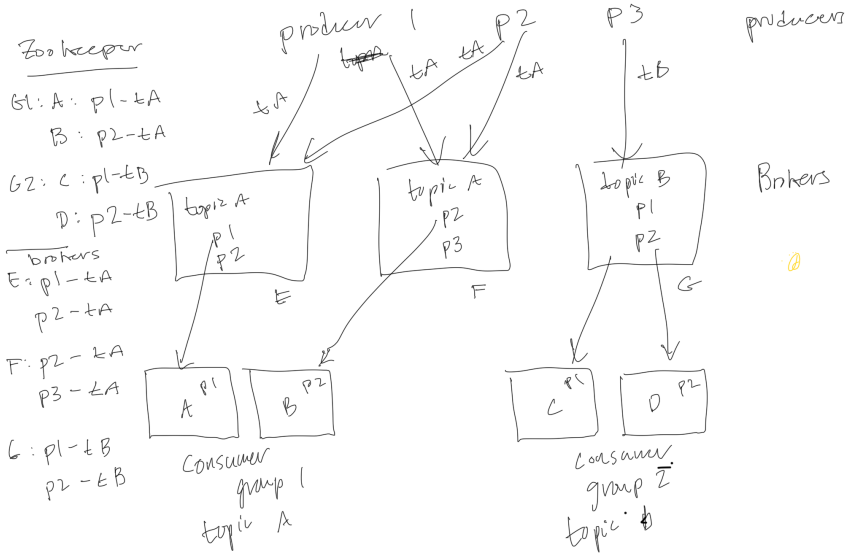
Brokers expire data older than some configurable time.

Consumer Coordination

Can have "consumer groups" -- multiple consumers nodes consuming a stream, where each message is processed by one node. Kafka takes care of spreading messages out to the different consumers in a group. Rather than spreading on a per message basis, consumers read from different partitions.

Uses Zookeeper -- a Chubby clone -- to keep track of the most recent message read in each partition, as well as the consumers in each group and the partitions and topics on each broker.

Diagram:



So what's different about Kafka vs a traditional message queue?

- not transactional -- at least once delivery
- consumer pull -- ability to access historical messages, stateless brokers
- designed to scale to very large numbers of messages & high rates via scaling

Kafka by itself doesn't provide any kind of high level language for operating on these streams of updates -- it just delivers messages.

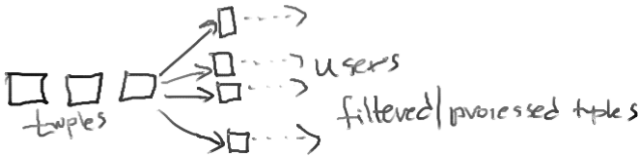
Instead we may want some kind of streaming query language (which is what Kafka Streams provides) that allows operations on these streams.

Database community spent many years on streaming -- proposing extensions to SQL that provide this kind of functionality.

What is the idea in a streaming query system?

Query posed, runs until user cancels it. Continuously outputs results.

Model:



Examples:

Monitoring applications -- stock prices, sensors, cheap airline tickets, traffic delays
Queries whose answers change over time!

How are streaming queries different from standard relational queries?

- Data arrives (asynchronously) from external sources, rather than being pre-loaded into tables on disk.

- Don't control when results arrive. Results may arrive in bursts, may saturate QP, etc.

What operations can I do on streams?

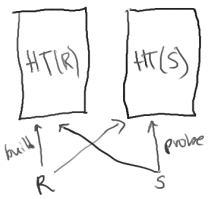
Basically what you can do on tables -- filter, map, join (merge), aggregate, sort

Since results never end, need some way to decide when to output tuples from blocking operators.

- What's a blocking operator? Filters aren't blocking, clearly.

- Which operators are blocking?
    - Sort
    - Aggregate
    - Join?

- Is join really blocking? No -- Show Parallel Pipelined Join



- So it's more than blocking. What else might we worry about with join?

Need to decide when we can purge operator state!

Solution for both blocking ops and join purge: windows

- Answers are typically notifying the user of some interesting event or occurrence, or maintaining some kind of running tabulation -- somewhat different from database queries

What notion from traditional databases is this a lot like?

Triggers!

How is this different from triggers?

Not that different. One way to look at a stream query processor is as an efficient trigger processor, with special support for maintaining state across trigger invocations and for purging state on window boundaries.

How does this relate to what Kafka (or pub/sub / messaging services) provides?

Similar mechanism for delivering asynchronous updates, except in stream processing systems we get to transform the results we receive via queries in addition to simply retrieving results.

These ideas have begun to appear in commercial databases -- e.g., Oracle*Streams, plus several startup companies (StreamBase, etc.) Increasingly new open source systems, like Kafka Streams, are starting to appear.

Kafka streams:

Idea is to provide a streaming API on top of Kafka.

Actually two APIs: a "Stream DSL" and a "processor" API. Processor API makes it possible to define a pipeline of operators.

Show examples -- processor.java + topology.java & dsl.java

(Code examples are from http://docs.confluent.io/3.1.1/streams/developer-guide.html#windowing-a-stream)

Kafka provides two APIs KStream and KTable, both of which support the same types of operations. A stream is just a immutable sequence of updates to keys, whereas a table is a materialized collection of (key, record) pairs.

Kafka allows me to convert a stream into a table by taking a stream, and "compacting" them into the most recent value of each key in the table.

Show table-stream-duality.java

Can also merge (join) a stream into a table, or aggregate a stream to produce a table of running aggregate values.

E.g., (k1,7), (k2,9), (k3,11), (k2,4) ==>

k1	7
k2	4
k3	11

Show join.java / dsl.java

Similarly I can view a table as a stream of updates.

This duality of tables / streams was first noted in the Stanford CQL project.

What operations can we do on a stream in Kafka Streams

Map
Filter
GroupBy, Reduce, Count, ...
Join

How are windows defined? What types of window semantics are supported?

Stream: 1, 3, 7, 5, 11, 12, 15, …
Sliding: [1, 3, 7], [3,7,5], [7,5,11], …
Tumbling (non-overlapping): [1,3,7], [5,11,12],...

Hopping by 2: [1,3,7], [7,5,11],[11,12,15],…

Windows can also be defined over a time interval, and tumble or hop by a particular time (show windowing.java)

Out of order semantics vis a vis timestamp ordering

One challenge that early streaming databases, e.g., the Aurora system, struggled with was how to deal with out of order values.

Why are out of order values a problem?

In distributed systems, can never know that you have seen all updates in a given time range, since some server may have been offline, network may have delayed, etc.

So if we are using time-based windows, how do we know when we can output a given window of data?

In Aurora, the system allowed users to define "slack" that provided a degree of tolerance to out-of-orderness by buffering up to N tuples are reordering them before outputting to downstream operators. But this required the user to pick N, and is inherently brittle.

Instead, Kafka provides an elegant model where past outputs -- e.g. aggregate values -- which are just tables -- can be updated when out of order tuples arise. Applications can decide how long they will wait for out of order stuff.

Example -- look at windowing.java -- this table contains values for each window, the contents for older windows will update when out of order records arrive!

Pretty elegant.

How is Kafka Streams like (or not like) other open source streaming products, like Spark Streaming?

Paper does a nice job of laying out why Spark Streaming is a lot more mechanism that you might want just to do stream processing. Show diagram