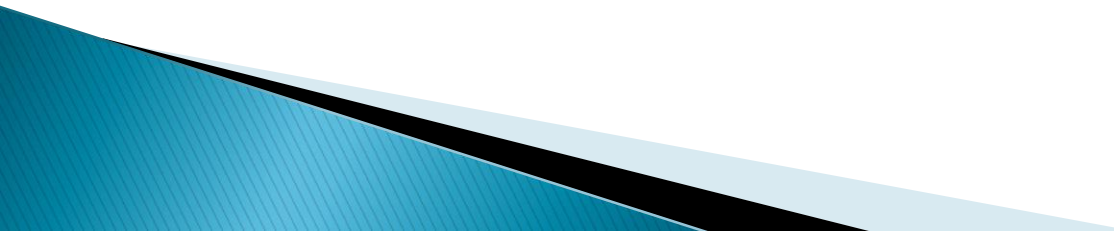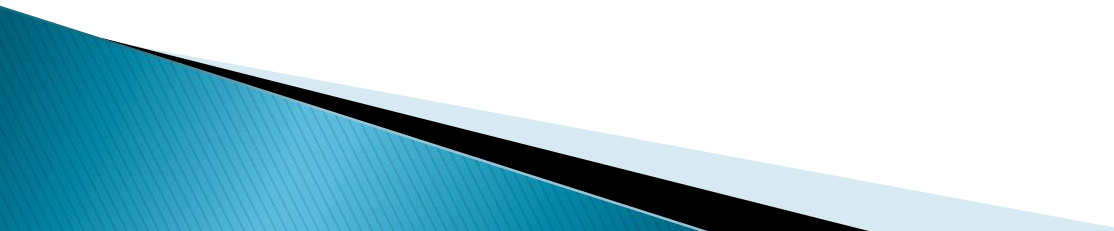# Stream Processing

Rajesh Pasham
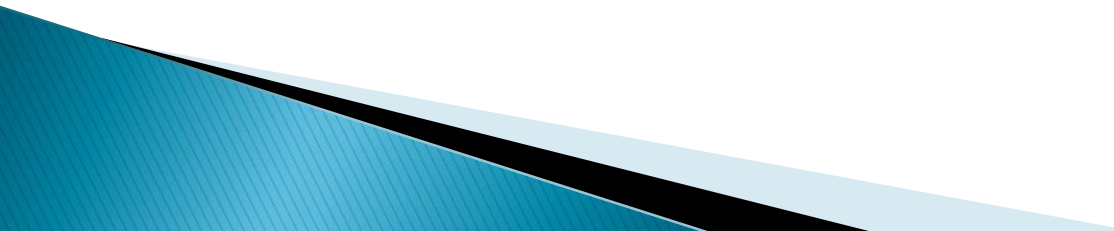
Stream Processing

- Kafka's reliable stream delivery capabilities make it a perfect source of data for stream-processing systems.
- Starting from version 0.10.0, Kafka does more than provide a reliable source of data streams to every popular stream-processing framework.
- Now Kafka includes a powerful stream-processing library as part of its collection of client libraries.
- This allows developers to consume, process, and produce events in their own apps, without relying on an external processing framework.
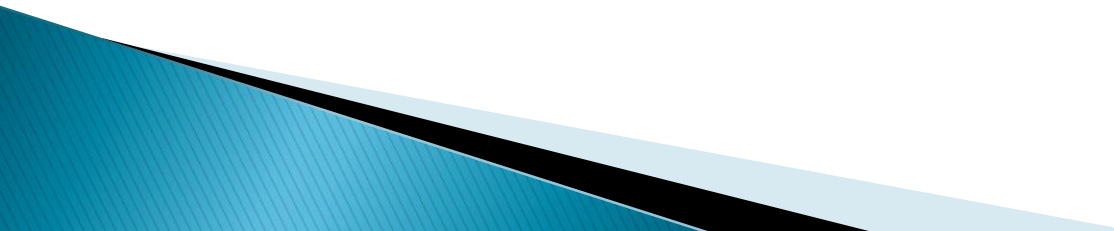
What Is Stream Processing?

- What is a data stream (also called an *event stream* or *streaming data*)?

- First and foremost, a *data stream* is an abstraction representing an unbounded dataset.
- *Unbounded* means infinite and ever growing.
- The dataset is unbounded because over time, new records keep arriving.
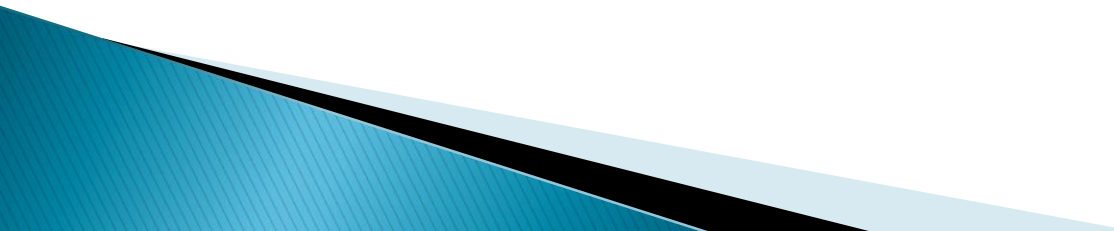- This definition is used by Google, Amazon, and pretty much everyone else.

What Is Stream Processing?

- This simple model (a stream of events) can be used to represent pretty much every business activity we care to analyze.
- We can look at a stream of credit card transactions, stock trades, package deliveries, network events going through a switch, events reported by sensors in manufacturing equipment, emails sent, moves in a game, etc.
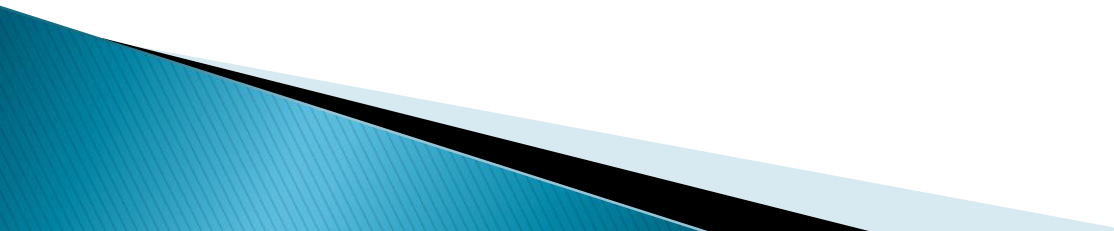- The list of examples is endless because pretty much everything can be seen as a sequence of events.

What Is Stream Processing?

▸ There are few other attributes of event streams model, in addition to their unbounded nature:

▸ *Event streams are ordered*
  ◦ There is an inherent notion of which events occur before or after other events.
  ◦ This is clearest when looking at financial events.
  ◦ A sequence in which I first put money in my account and later spend the money is very different from a sequence at which I first spend the money and later cover my debt by depositing money back.
  ◦ The latter will incur overdraft charges while the former will not.

What Is Stream Processing?

- There are few other attributes of event streams model, in addition to their unbounded nature:

- *Immutable data records*
  - Events, once occured, can never be modified.
  - A financial transaction that is cancelled does not disapear.
  - Instead, an additional event is written to the stream, recording a cancellation of previous transaction.
  - When a customer returns merchandise to a shop, we don't delete the fact that the merchandise was sold to him earlier, rather we record the return as an additional event.
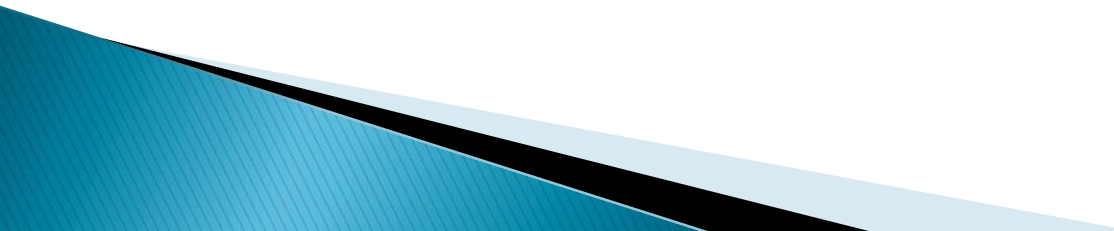
What Is Stream Processing?

- There are few other attributes of event streams model, in addition to their unbounded nature:

- *Event streams are replayable*
  - This is a desirable property.
  - While it is easy to imagine nonreplayable streams (TCP packets streaming through a socket are generally nonreplayable), for most business applications, it is critical to be able to replay a raw stream of events that occured months (and sometimes years) earlier.
  - This is required in order to correct errors, try new methods of analysis, or perform audits.
  - Kafka allows capturing and replaying a stream of events.

What Is Stream Processing?

- Stream processing refers to the ongoing processing of one or more event streams.
- Stream processing is a programming paradigm - just like requestresponse and batch processing.
- *Request-response*
    - This is the lowest latency paradigm, with response times ranging from submilliseconds to a few milliseconds, usually with the expectation that response times will be highly consistent.
    - The mode of processing is usually blocking - an app sends a request and waits for the processing system to respond.
    - In the database world, this paradigm is known as *online transaction processing* (OLTP).
    - Point-ofsale systems, credit card processing, and time-tracking systems typically work in this paradigm.

What Is Stream Processing?
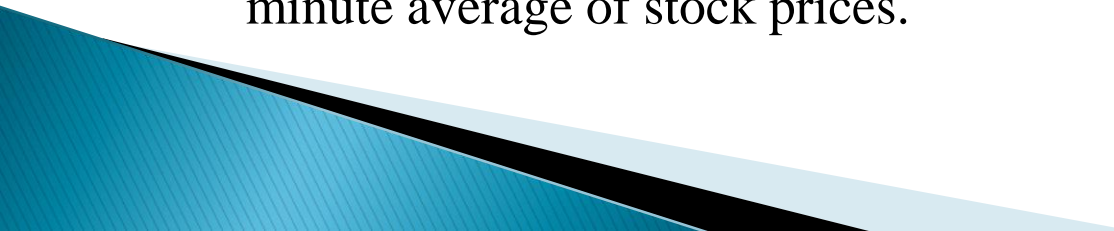
- *Batch processing*
  - This is the high-latency/high-throughput option.
  - The processing system wakes up at set times - every day at 2:00 A.M., every hour on the hour, etc.
  - It reads all required input (either all data available since last execution, all data from beginning of month, etc.), writes all required output, and goes away until the next time it is scheduled to run.
  - Processing times range from minutes to hours and users expect to read stale data when they are looking at results.
  - In the database world, these are the data warehouse and business intelligence systems - data is loaded in huge batches once a day, reports are generated, and users look at the same reports until the next data load occurs.

What Is Stream Processing?

▸ *Stream processing*
  ◦ This is a contentious and nonblocking option.
  ◦ Filling the gap between the request-response world where we wait for events that take two milliseconds to process and the batch processing world where data is processed once a day and takes eight hours to complete.
  ◦ Most business processes don't require an immediate response within milliseconds but can't wait for the next day either.
  ◦ Most business processes happen continuously, and as long as the business reports are updated continuously and the line of business apps can continuously respond, the processing can proceed without anyone waiting for a specific response within milliseconds.
  ◦ Business processes like alerting on suspicious credit transactions or network activity, adjusting prices in real-time based on supply and demand, or tracking deliveries of packages are all natural fit for continuous but nonblocking processing.

Stream-Processing Concepts

- Stream processing is very similar to any type of data processing
- However, there are some key concepts that are unique to stream processing
- Let's take a look at a few of those concepts.

- Time
  ◦ Time is probably the most important concept in stream processing and often the most confusing.
  ◦ Most stream applications perform operations on time windows.
  ◦ For example, our stream application might calculate a moving five-minute average of stock prices.

Stream-Processing Concepts

- Stream-processing systems typically refer to the following notions of time:
  - *Event time*
  - *Log append time*
  - *Processing time*

Stream-Processing Concepts

- *Event time*
    - This is the time the events we are tracking occurred and the record was created-the time a measurement was taken, an item at was sold at a shop, a user viewed a page on our website, etc.
    - In versions 0.10.0 and later, Kafka automatically adds the current time to producer records at the time they are created.
    - If this does not match your application's notion of *event time*, such as in cases where the Kafka record is created based on a database record some time after the event occurred, you should add the event time as a field in the record itself.
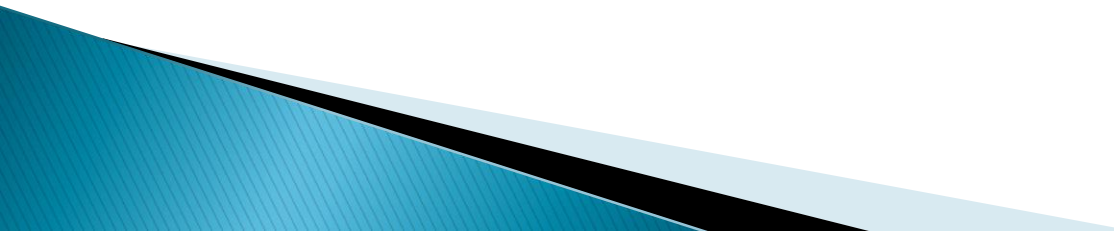   - Event time is usually the time that matters most when processing stream data.

Stream-Processing Concepts

- *Log append time*
  - This is the time the event arrived to the Kafka broker and was stored there.
  - In versions 0.10.0 and higher, Kafka brokers will automatically add this time to records they receive if Kafka is configured to do so or if the records arrive from older producers and contain no timestamps.
  - This notion of time is typically less relevant for stream processing, since we are usually interested in the times the events occurred.
  - In cases where the real event time was not recorded, log append time can still be used consistently because it does not change after the record was created.

Stream-Processing Concepts

▸ *Processing time*

- This is the time at which a stream-processing application received the event in order to perform some calculation.
- This time can be milliseconds, hours, or days after the event occurred.
- This notion of time assigns different timestamps to the same event depending on exactly when each stream processing application happened to read the event.
- It can even differ for two threads in the same application!
- Therefore, this notion of time is highly unreliable and best avoided.

Stream-Processing Concepts

‣ State
- ◦ As long as you only need to process each event individually, stream processing is a very simple activity.
- ◦ Stream processing becomes really interesting when you have operations that involve multiple events: counting the number of events by type, moving averages, joining two streams to create an enriched stream of information, etc.
- ◦ In those cases, it is not enough to look at each event by itself; you need to keep track of more information - how many events of each type did we see this hour, all events that require joining, sums, averages, etc.
- ◦ We call the information that is stored between events a *state*.
- ◦ It is often tempting to store the state in variables that are local to the streamprocessing app, such as a simple hash-table to store moving counts.

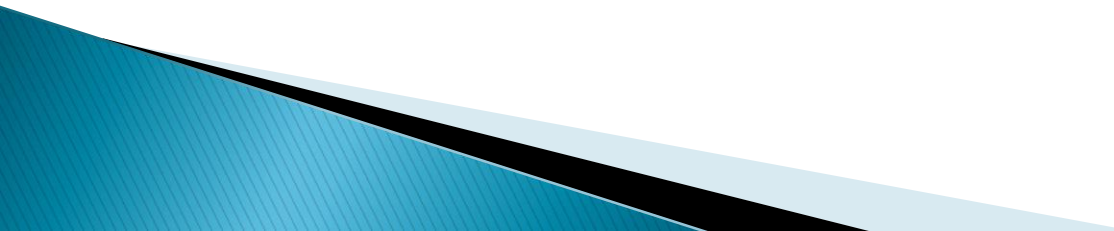# Stream-Processing Concepts

- State
    - However, this is not a reliable approach for managing state in stream processing because when the stream-processing application is stopped, the state is lost, which changes the results.

- Stream processing refers to several types of state:
    - *Local or internal state*
    - *External state*

Stream-Processing Concepts

Local or internal state

- State that is accessible only by a specific instance of the stream-processing application.
- This state is usually maintained and managed with an embedded, inmemory database running within the application.
- The advantage of local state is that it is extremely fast.
- The disadvantage is that you are limited to the amount of memory available.
- As a result, many of the design patterns in stream processing focus on ways to partition the data into substreams that can be processed using a limited amount of local state.
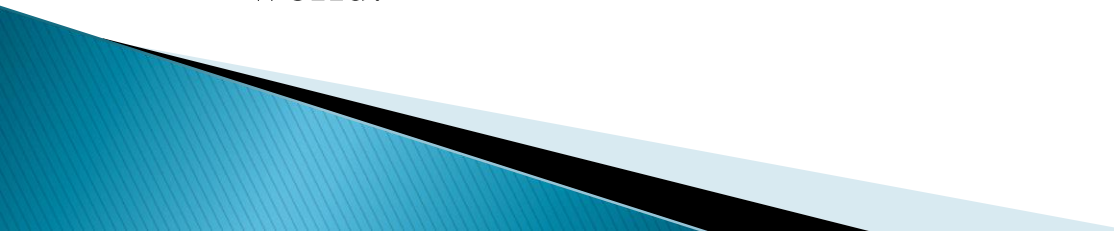
Stream-Processing Concepts

External state

- State that is maintained in an external datastore, often a NoSQL system like Cassandra.
- The advantages of an external state are its virtually unlimited size and the fact that it can be accessed from multiple instances of the application or even from different applications.
- The downside is the extra latency and complexity introduced with an additional system.
- Most stream-processing apps try to avoid having to deal with an external store, or at least limit the latency overhead by caching information in the local state and communicating with the external store as rarely as possible.
- This usually introduces challenges with maintaining consistency between the internal and external state.
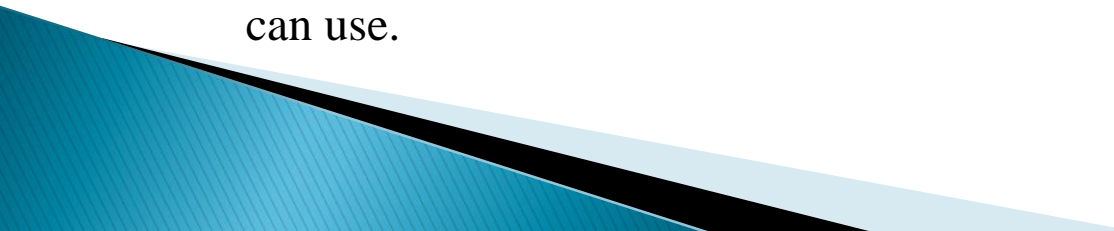
Stream-Processing Concepts

Stream-Table Duality

- ◦ We are all familiar with database tables.
- ◦ A table is a collection of records, each identified by its primary key and containing a set of attributes as defined by a schema.
- ◦ Table records are mutable.
- ◦ Unlike tables, streams contain a history of changes.
- ◦ Streams are a string of events wherein each event caused a change.
- ◦ A table contains a current state of the world, which is the result of many changes.
- ◦ streams and tables are two sides of the same coin—the world always changes, and sometimes we are interested in the events that caused those changes, whereas other times we are interested in the current state of the world.

Stream-Processing Concepts

Stream-Table Duality

- In order to convert a table to a stream, we need to capture the changes that modify the table.
- Take all those insert, update, and delete events and store them in a stream.
- Most databases offer change data capture (CDC) solutions for capturing these changes and there are many Kafka connectors that can pipe those changes into Kafka where they will be available for stream processing.
- In order to convert a stream to a table, we need to apply all the changes that the stream contains.
- This is also called *materializing* the stream.
- We create a table, either in memory, in an internal state store, or in an external database, and start going over all the events in the stream from beginning to end, changing the state as we go.
- When we finish, we have a table representing a state at a specific time that we can use.

Stream-Processing Concepts

Stream-Table Duality

▶ Suppose we have a store selling shoes. A stream representation of our retail activity can be a stream of events:
  ◦ "Shipment arrived with red, blue, and green shoes"
  ◦ "Blue shoes sold"
  ◦ "Red shoes sold"
  ◦ "Blue shoes returned"
  ◦ "Green shoes sold"

▶ If we want to know what our inventory contains right now or how much money we made until now, we need to materialize the view.

▶ If we want to know how busy the store is, we can look at the entire stream and see that there were five transactions.

▶ We may also want to investigate why the blue shoes were returned.

Stream-Processing Concepts
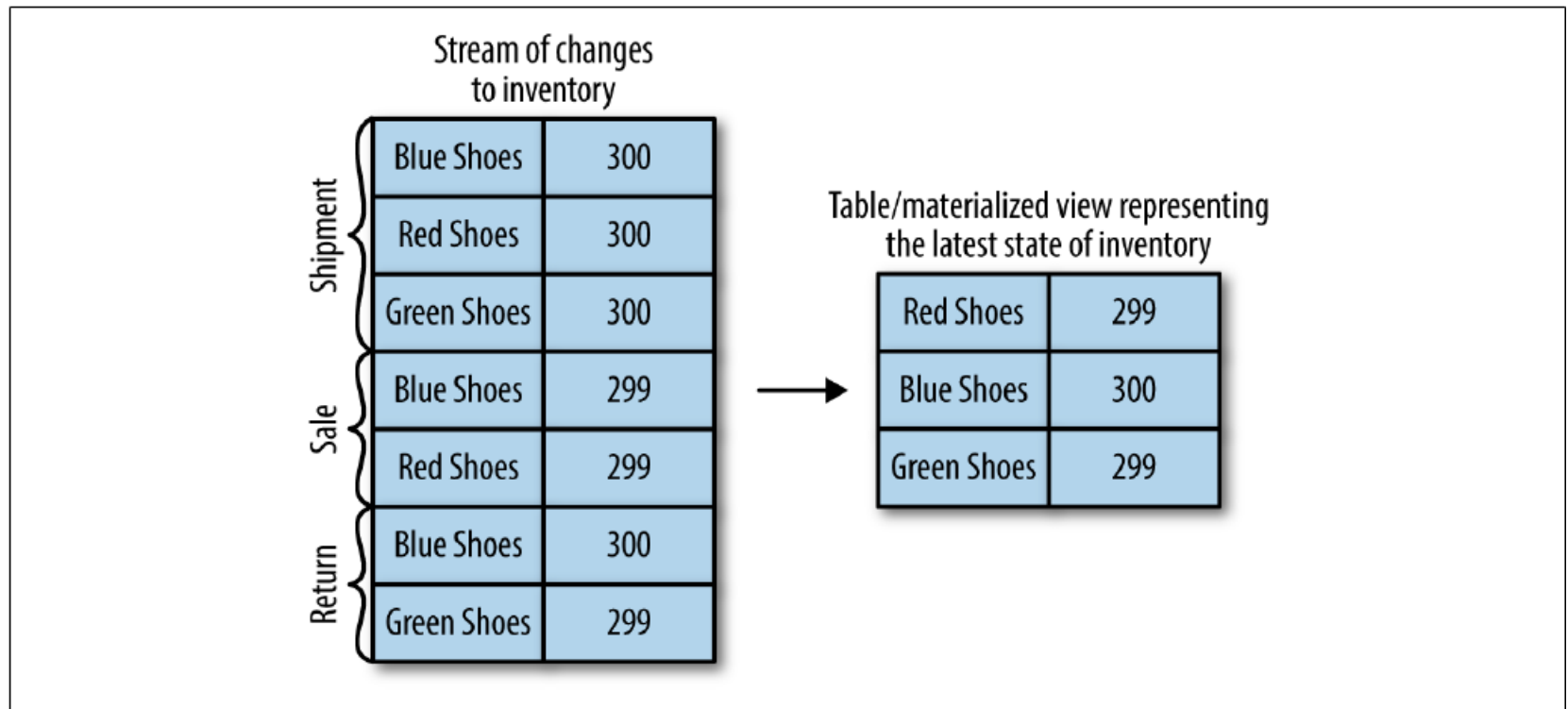
Stream-Table Duality

Figure 11-1. Materializing inventory changes

Stream-Processing Concepts

Time Windows

- Most operations on streams are windowed operations-operating on slices of time:
  - ◦ moving averages, top products sold this week, 99th percentile load on the system, etc.
- Join operations on two streams are also windowed - we join events that occurred at the same slice of time.
- we want to know:
  - ◦ Size of the window:
  - ◦ How often the window moves (*advance interval*):
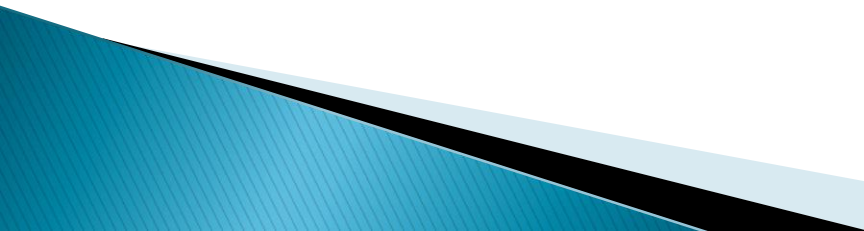 - ◦ How long the window remains updatable:
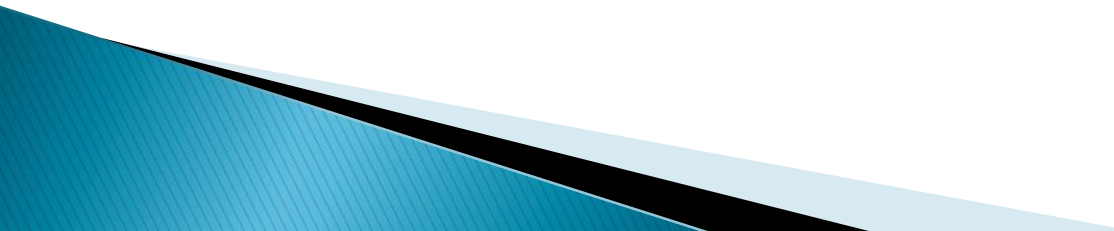
Stream-Processing Concepts

Time Windows

- ▶ Size of the window:
  - ◦ do we want to calculate the average of all events in every fiveminute window? Every 15-minute window? Or the entire day? Larger windows are smoother but they lag more - if price increases, it will take longer to notice than with a smaller window.
- ▶ How often the window moves (*advance interval*):
  - ◦ Five-minute averages can update every minute, second, or every time there is a new event.
  - ◦ When the *advance interval* is equal to the window size, this is sometimes called a *tumbling window*.
  - ◦ When the window moves on every record, this is sometimes called a *sliding window*.

Stream-Processing Concepts

Time Windows

▸ How long the window remains updatable:

◦ Our five-minute moving average calculated the average for 00:00-00:05 window.

◦ Now an hour later, we are getting a few more results with their *event time* showing 00:02.

◦ Ideally, we'll be able to define a certain time period during which events will get added to their respective time-slice.

◦ For example, if the events were up to four hours late, we should recalculate the results and update.

◦ If events arrive later than that, we can ignore them.

Stream-Processing Design Patterns

- Every stream-processing system is different—from the basic combination of a consumer, processing logic, and producer to involved clusters like Spark Streaming with its machine learning libraries, and much in between.
- But there are some basic design patterns, which are known solutions to common requirements of stream-processing architectures.
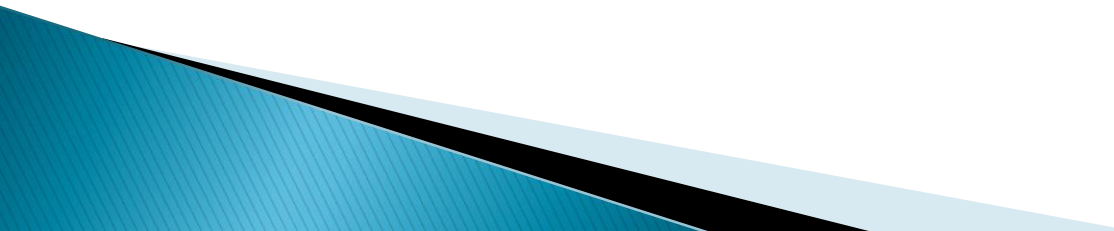- We'll review a few of those well-known patterns and show how they are used with a few examples.

Stream-Processing Design Patterns

Single-Event Processing

- The most basic pattern of stream processing is the processing of each event in isolation.

- This is also known as a map/filter pattern because it is commonly used to filter unnecessary events from the stream or transform each event.

- In this pattern, the stream-processing app consumes events from the stream, modifies each event, and then produces the events to another stream.

- An example is an app that reads log messages from a stream and writes ERROR events into a high-priority stream and the rest of the events into a low-priority stream.
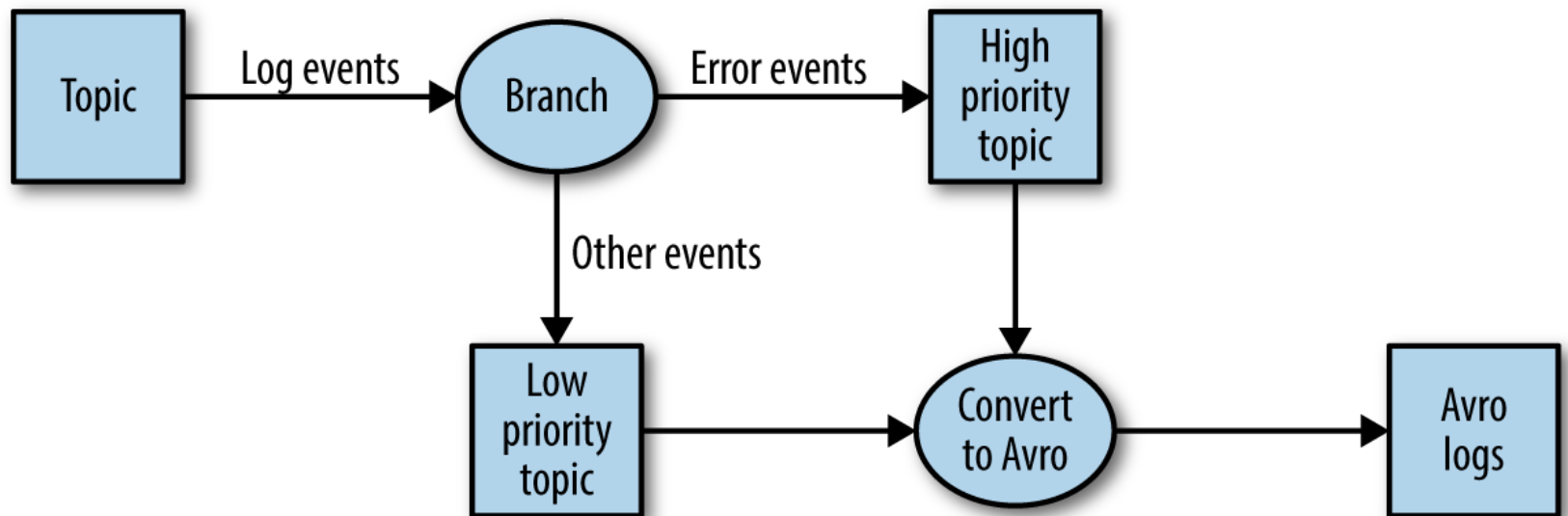
Stream-Processing Design Patterns

Single-Event Processing

▸ Another example is an application that reads events from a stream and modifies them from JSON to Avro.

▸ Such applications need to maintain state within the application because each event can be handled independently.

▸ This means that recovering from app failures or loadbalancing is incredibly easy as there is no need to recover state; you can simply hand off the events to another instance of the app to process.

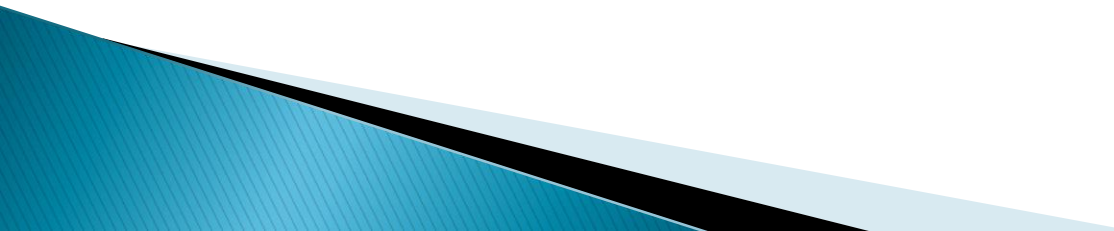# Stream-Processing Design Patterns

Single-Event Processing

▸ This pattern can be easily handled with a simple producer and consumer, as seen in Figure.
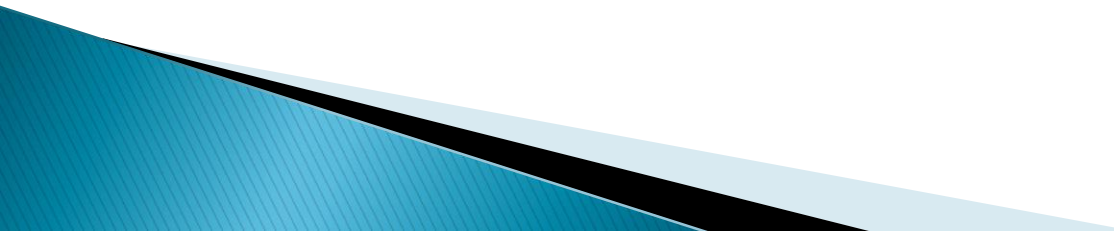

Single-event processing topology

Stream-Processing Design Patterns

Processing with Local State

- Most stream-processing applications are concerned with aggregating information, especially time-window aggregation.
- An example of this is finding the minimum and maximum stock prices for each day of trading and calculating a moving average.
- These aggregations require maintaining a *state* for the stream.
- In our example, in order to calculate the minimum and average price each day, we need to store the minimum and maximum values we've seen up until the current time and compare each new value in the stream to the stored minimum and maximum.
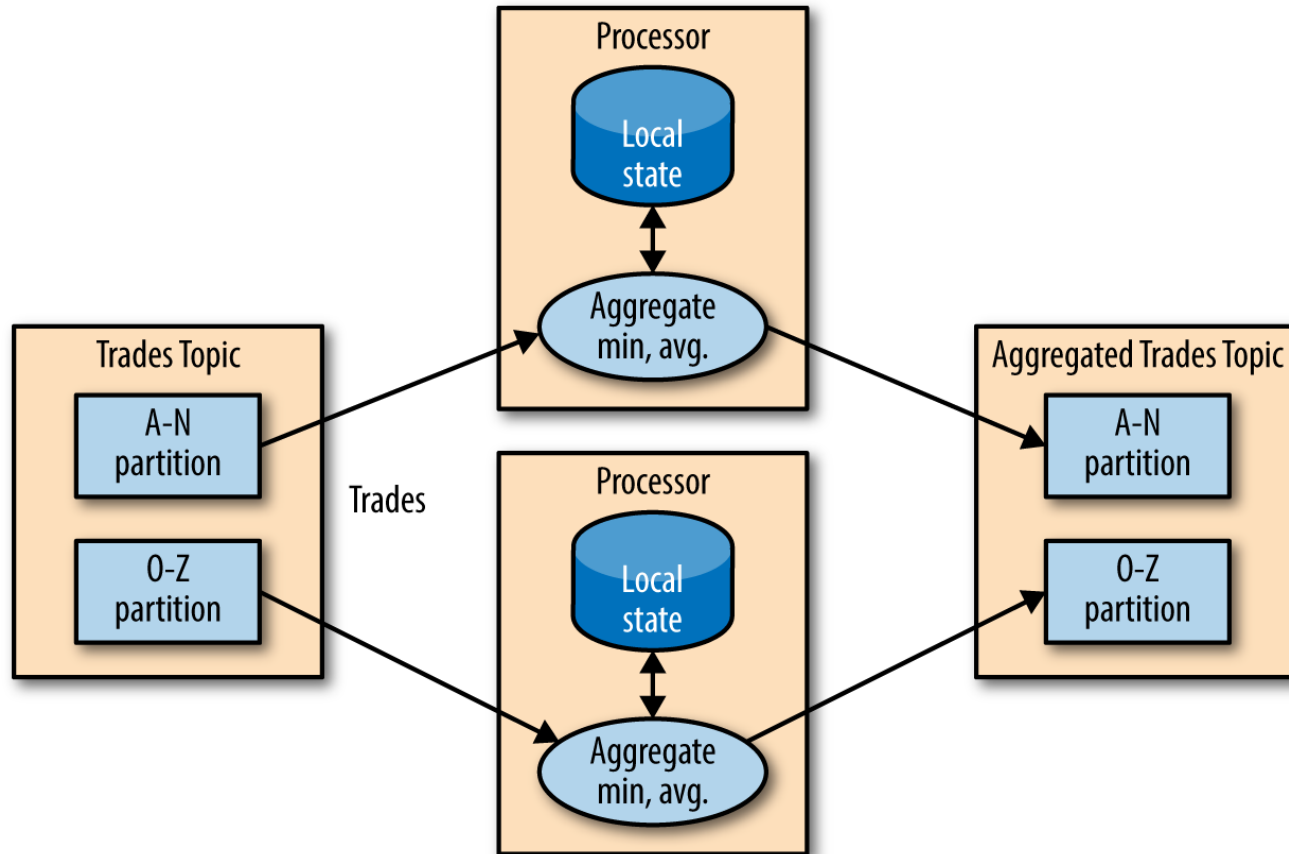
Stream-Processing Design Patterns

Processing with Local State

- All these can be done using *local* state (rather than a shared state) because each operation in our example is a *group by* aggregate.

- That is, we perform the aggregation per stock symbol, not on the entire stock market in general.

- We use a Kafka partitioner to make sure that all events with the same stock symbol are written to the same partition.

- Then, each instance of the application will get all the events from the partitions that are assigned to it (this is a Kafka consumer guarantee).

- This means that each instance of the application can maintain state for the subset of stock symbols that are written to the partitions that are assigned to it.

Stream-Processing Design Patterns

Processing with Local State

Topology for event processing with local state

Stream-Processing Design Patterns

Processing with Local State

▸ Stream-processing applications become significantly more complicated when the application has local state and there are several issues a stream-processing application must address:

◦ *Memory usage*

• The local state must fit into the memory available to the application instance.

◦ *Persistence*

• We need to make sure the state is not lost when an application instance shuts down, and that the state can be recovered when the instance starts again or is replaced by a different instance.

• This is something that Kafka Streams handles very well—local state is stored in-memory using embedded RocksDB, which also persists the data to disk for quick recovery after restarts.

Stream-Processing Design Patterns

Processing with Local State

- *Rebalancing*
  - Partitions sometimes get reassigned to a different consumer.
  - When this happens, the instance that loses the partition must store the last good state, and the instance that receives the partition must know to recover the correct state.

Stream-Processing Design Patterns

Multiphase Processing/Repartitioning

- Local state is great if you need a *group by* type of aggregate.
- But what if you need a result that uses all available information? For example, suppose we want to publish the top 10 stocks each day.
- Obviously, nothing we do locally on each application instance is enough because all the top 10 stocks could be in partitions assigned to other instances.
- What we need is a two-phase approach.
- First, we calculate the daily gain/loss for each stock symbol.
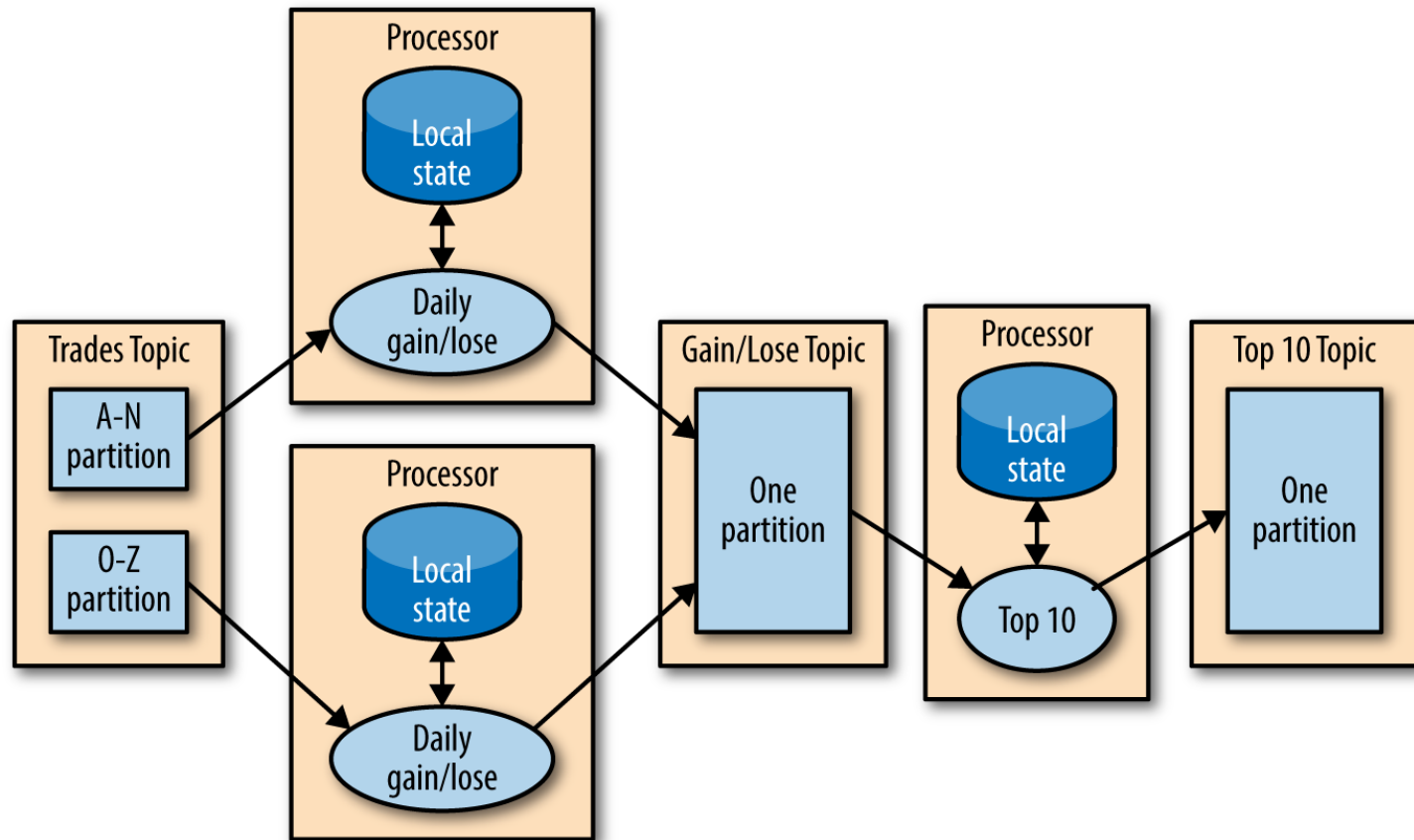- We can do this on each instance with a local state.

Stream-Processing Design Patterns

Multiphase Processing/Repartitioning

- Then we write the results to a new topic with a single partition.
- This partition will be read by a single application instance that can then find the top 10 stocks for the day.
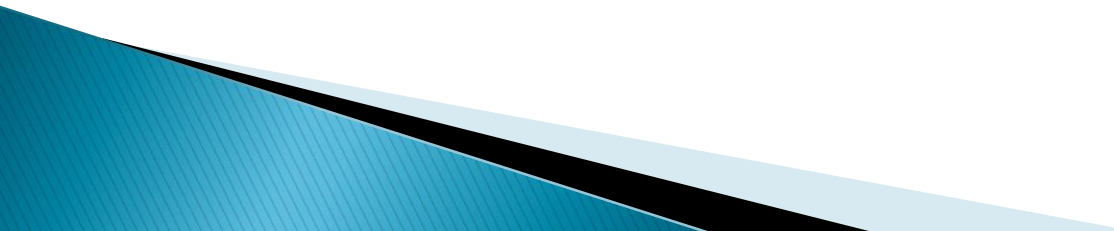
Stream-Processing Design Patterns

Multiphase Processing/Repartitioning

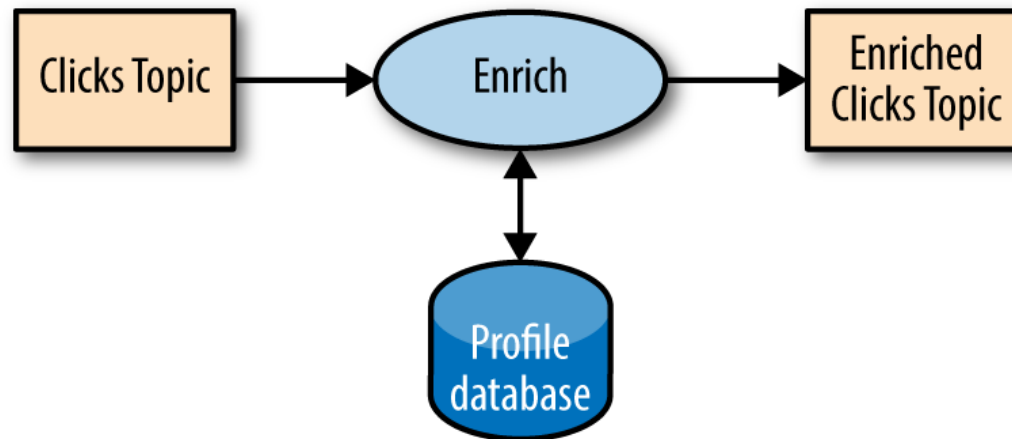*Topology that includes both local state and repartitioning steps*

Stream-Processing Design Patterns

Processing with External Lookup: Stream-Table Join

- Sometimes stream processing requires integration with data external to the stream-validating transactions against a set of rules stored in a database, or enriching clickstream information with data about the users who clicked.

- The obvious idea on how to perform an external lookup for data enrichment is something like this: for every click event in the stream, look up the user in the profile database and write an event that includes the original click plus the user age and gender to another topic.
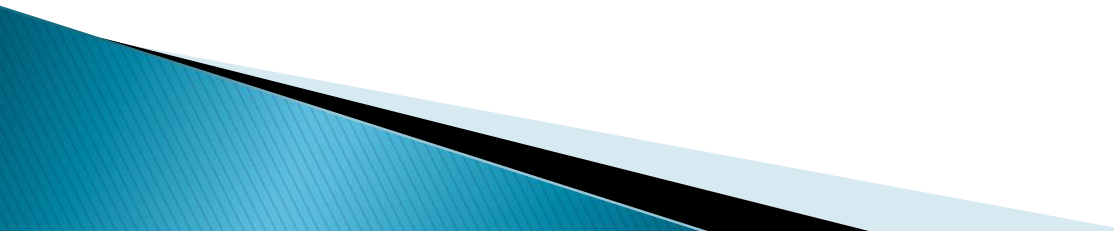
Stream-Processing Design Patterns

Processing with External Lookup: Stream-Table Join

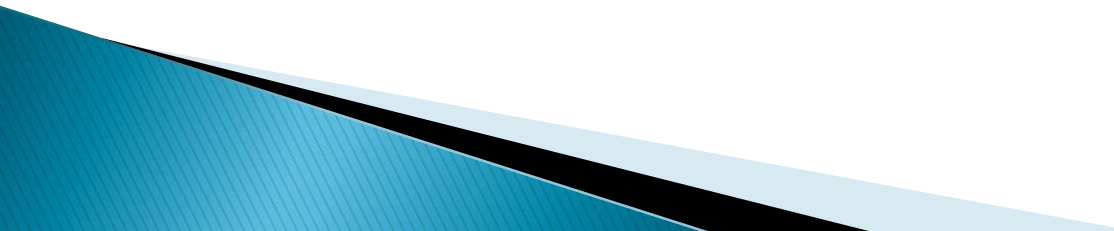*Stream processing that includes an external data source*

Stream-Processing Design Patterns

Processing with External Lookup: Stream-Table Join

- The problem with this obvious idea is that an external lookup adds significant latency to the processing of every record-usually between 5-15 milliseconds.

- In many cases, this is not feasible.

- Often the additional load this places on the external datastore is also not acceptable-stream-processing systems can often handle 100K-500K events per second, but the database can only handle perhaps 10K events per second at reasonable performance.

- We want a solution that scales better.

Stream-Processing Design Patterns

Processing with External Lookup: Stream-Table Join

- In order to get good performance and scale, we need to cache the information from the database in our stream-processing application.
- Managing this cache can be challenging though-how do we prevent the information in the cache from getting stale?
- If we refresh events too often, we are still hammering the database and the cache isn't helping much.
- If we wait too long to get new events, we are doing stream processing with stale information.
- But if we can capture all the changes that happen to the database table in a stream of events, we can have our stream-processing job listen to this stream and update the cache based on database change events.
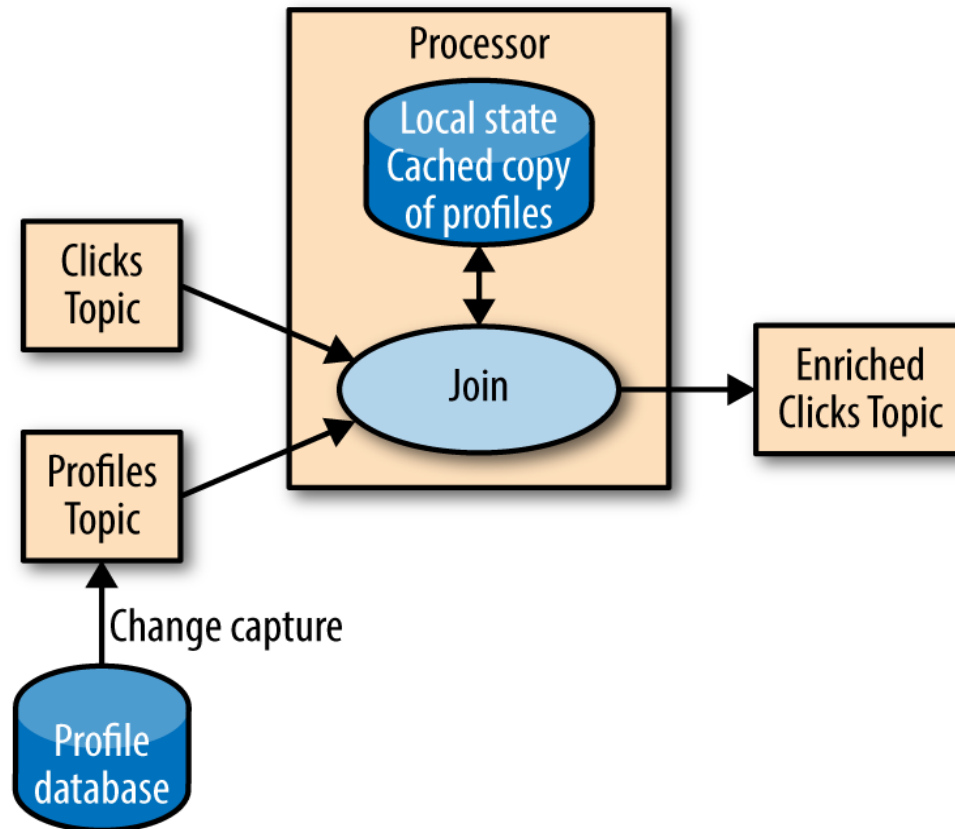
Stream-Processing Design Patterns

Processing with External Lookup: Stream-Table Join

- Capturing changes to the database as events in a stream is known as CDC, and if you use Kafka Connect you will find multiple connectors capable of performing CDC and converting database tables to a stream of change events.

- This allows you to keep your own private copy of the table, and you will be notified whenever there is a database change event so you can update your own copy accordingly.

- Then, when you get click events, you can look up the user_id at your local cache and enrich the event.

- And because you are using a local cache, this scales a lot better and will not affect the database and other apps using it.

- We refer to this as a *stream-table join* because one of the streams represents changes to a locally cached table.
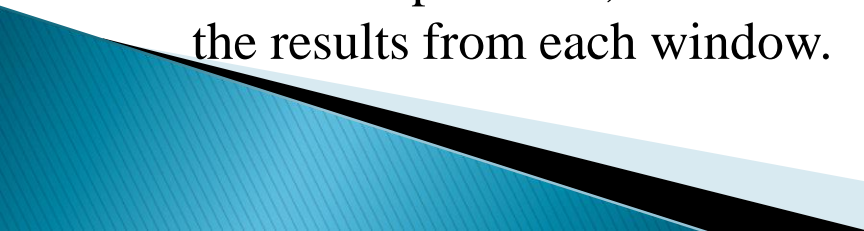
Stream-Processing Design Patterns

Processing with External Lookup: Stream-Table Join

Topology joining a table and a stream of events, removing the need to involve an external data source in stream processing
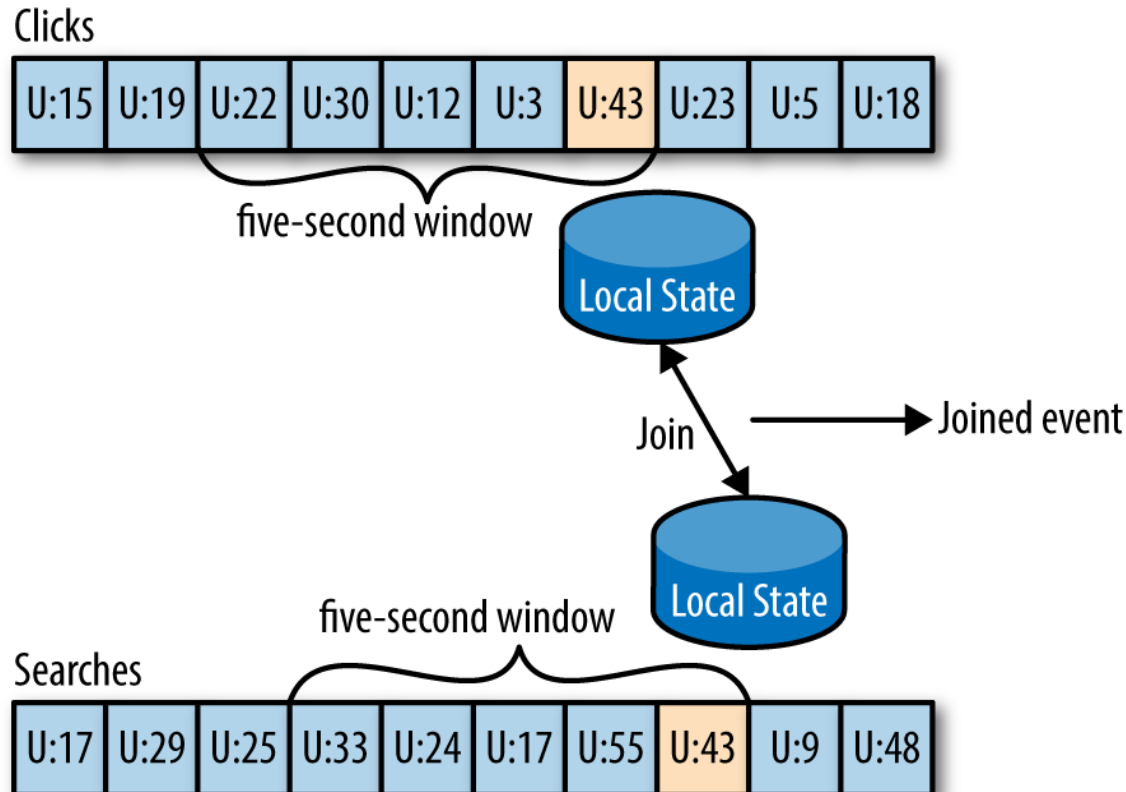
Stream-Processing Design Patterns

Streaming Join

- Sometimes you want to join two real event streams rather than a stream with a table.

- For example, let's say that we have one stream with search queries that people entered into our website and another stream with clicks, which include clicks on search results.

- We want to match search queries with the results they clicked on so that we will know which result is most popular for which query.

- Obviously we want to match results based on the search term but only match them within a certain time-window.

- We assume the result is clicked seconds after the query was entered into our search engine.

- So we keep a small, few-seconds-long window on each stream and match the results from each window.

Stream-Processing Design Patterns

Streaming Join

Joining two streams of events; these joins always involve a moving time window

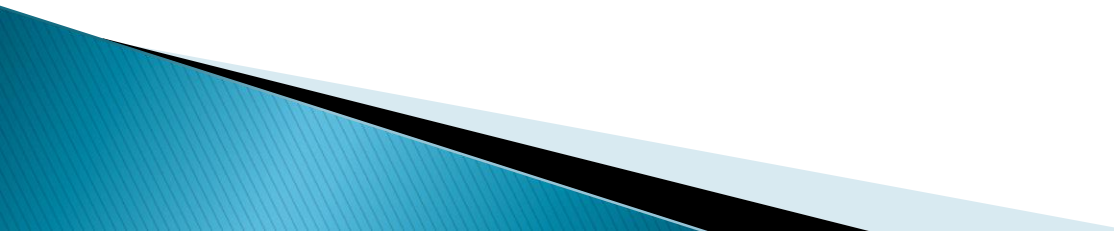# Stream-Processing Design Patterns

Streaming Join

- The way this works in Kafka Streams is that both streams, queries and clicks, are partitioned on the same keys, which are also the join keys.

- This way, all the click events from user_id:42 end up in partition 5 of the clicks topic, and all the search events for user_id:42 end up in partition 5 of the search topic.

- Kafka Streams then makes sure that partition 5 of both topics is assigned to the same task.

- So this task sees all the relevant events for user_id:42.

- It maintains the join-window for both topics in its embedded RocksDB cache, and this is how it can perform the join.

Stream-Processing Design Patterns

Out-of-Sequence Events

- Handling events that arrive at the stream at the wrong time is a challenge not just in stream processing but also in traditional ETL systems.

- Out-of-sequence events happen quite frequently and expectedly in IoT (Internet of Things) scenarios.

- For example, a mobile device loses WiFi signal for a few hours and sends a few hours' worth of events when it reconnects.

- This also happens when monitoring network equipment (a faulty switch doesn't send diagnostics signals until it is repaired) or manufacturing (network connectivity in plants is notoriously unreliable, especially in developing countries).
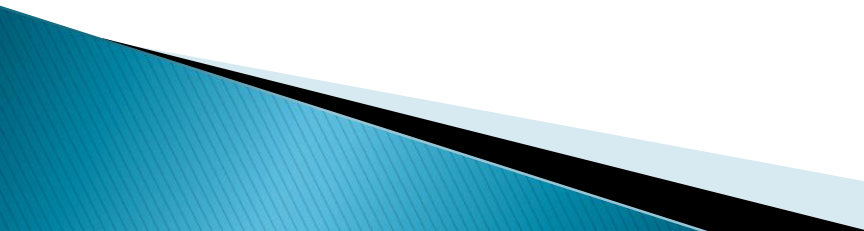
Stream-Processing Design Patterns

Out-of-Sequence Events

Out of sequence events

Stream-Processing Design Patterns

Out-of-Sequence Events

- Our streams applications need to be able to handle those scenarios.
- This typically means the application has to do the following:
  - Recognize that an event is out of sequence—this requires that the application examine the event time and discover that it is older than the current time.
  - Define a time period during which it will attempt to reconcile out-of-sequence events.
  - Perhaps a three-hour delay should be reconciled and events over three weeks old can be thrown away.
  - Have an in-band capability to reconcile this event.

Stream-Processing Design Patterns

Out-of-Sequence Events

- This typically means the application has to do the following:
  - Be able to update results.
- Several stream-processing frameworks, including Google's Dataflow and Kafka Streams, have built-in support for the notion of event time independent of the processing time and the ability to handle events with event times that are older or newer than the current processing time.
- This is typically done by maintaining multiple aggregation windows available for update in the local state and giving developers the ability to configure how long to keep those window aggregates available for updates.
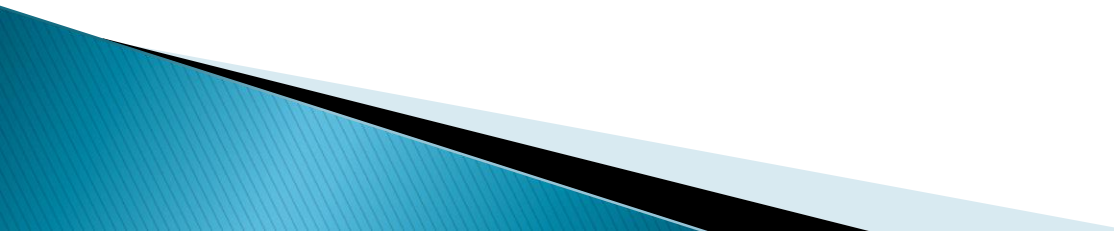
Stream-Processing Design Patterns

Out-of-Sequence Events

- The longer the aggregation windows are kept available for updates, the more memory is required to maintain the local state.

- The Kafka's Streams API always writes aggregation results to result topics.

- Those are usually compacted topics, which means that only the latest value for each key is preserved.

- In case the results of an aggregation window need to be updated as a result of a late event, Kafka Streams will simply write a new result for this aggregation window, which will overwrite the previous result.

Stream-Processing Design Patterns

Reprocessing

- The last important pattern is processing events.
- There are two variants of this pattern:
  - We have an improved version of our stream-processing application. We want to run the new version of the application on the same event stream as the old, produce a new stream of results that does not replace the first version, compare the results between the two versions, and at some point move clients to use the new results instead of the existing ones.

  - The existing stream-processing app is buggy. We fix the bug and we want to reprocess the event stream and recalculate our results
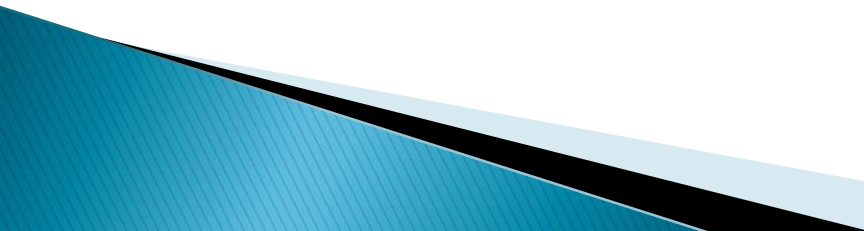
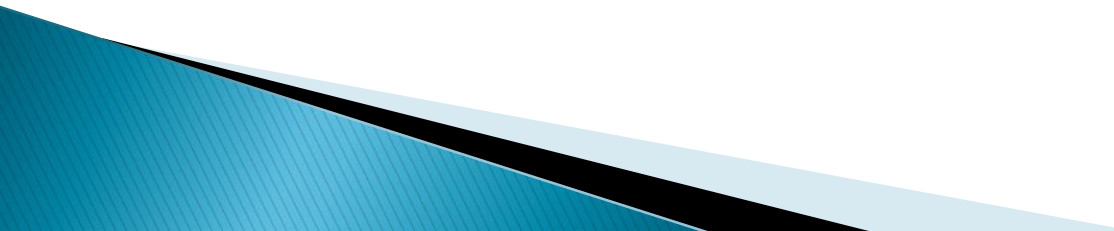# Stream-Processing Design Patterns

Reprocessing

▸ The first use case is made simple by the fact that Apache Kafka stores the event streams in their entirety for long periods of time in a scalable datastore.

▸ This means that having two versions of a stream processing-application writing two result streams only requires the following:

◦ Spinning up the new version of the application as a new consumer group.

◦ Configuring the new version to start processing from the first offset of the input topics (so it will get its own copy of all events in the input streams)
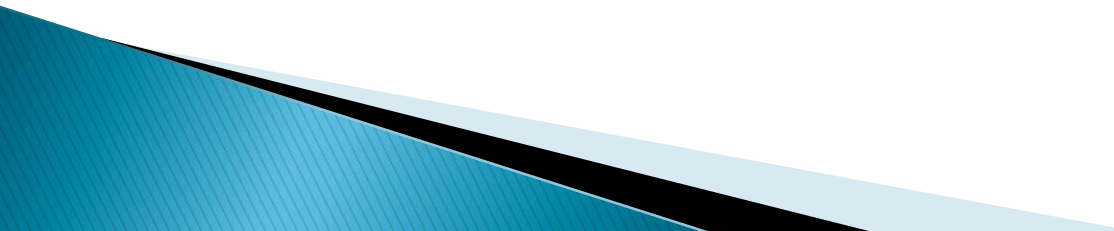
Stream-Processing Design Patterns

Reprocessing

- ◦ Letting the new application continue processing and switching the client applications to the new result stream when the new version of the processing job has caught up

- The second use case is more challenging-it requires "resetting" an existing app to start processing back at the beginning of the input streams, resetting the local state (so we won't mix results from the two versions of the app), and possibly cleaning the previous output stream.

- While Kafka Streams has a tool for resetting the state for a stream-processing app, best practice is to try to use the first method whenever sufficient capacity exists to run two copies of the app and generate two result streams.

- The first method is much safer—it allows switching back and forth between multiple versions and comparing results between versions, and doesn't risk losing critical data or introducing errors during the cleanup process.
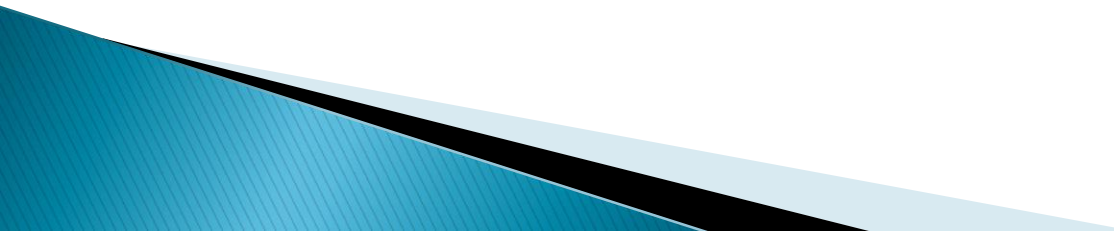
Kafka Streams by Example

- Apache Kafka has two streams APIs - a low-level Processor API and a high-level Streams DSL.
- We will use Kafka Streams DSL in our examples.
- The DSL allows you to define the stream-processing application by defining a chain of transformations to events in the streams.
- Transformations can be as simple as a filter or as complex as a stream-to-stream join.
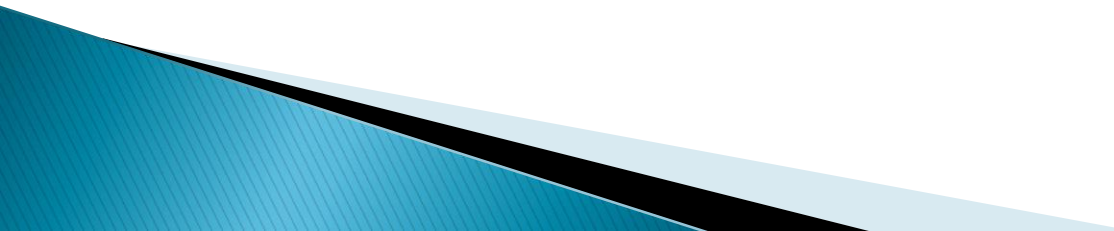- The lower level API allows you to create your own transformations, but as you'll see, this is rarely required.

Kafka Streams by Example

- An application that uses the DSL API always starts with using the StreamBuilder to create a processing *topology* - a directed graph (DAG) of transformations that are applied to the events in the streams.

- Then you create a KafkaStreams execution object from the topology.

- Starting the KafkaStreams object will start multiple threads, each applying the processing topology to events in the stream.

- The processing will conclude when you close the KafkaStreams object.

Kafka Streams by Example

- We'll look at few examples that use Kafka Streams to implement some of the design patterns we just discussed.
- A simple word count example will be used to demonstrate the map/filter pattern and simple aggregates.
- Then we'll move to an example where we calculate different statistics on stock market trades, which will allow us to demonstrate window aggregations.
- Finally we'll use ClickStream Enrichment as an example to demonstrate streaming joins.

Word Count

- Let's walk through an abbreviated word count example for Kafka Streams. You can find the full example on GitHub.

- The first thing you do when creating a stream-processing app is configure Kafka Streams.

- Kafka Streams has a large number of possible configurations, you can find them in the documentation.

- In addition, you can also configure the producer and consumer embedded in Kafka Streams by adding any producer or consumer config to the Properties object:

Word Count

```java
public class WordCountExample {

public static void main(String[] args) throws Exception{

Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG,"wordcount");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,"localhost:9092");
props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,Serdes.String().getClass().getName());
```

Word Count

```
props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());

KStreamBuilder builder = new KStreamBuilder();

KStream<String, String> source = builder.stream("wordcount-
input");

final Pattern pattern = Pattern.compile("\\W+");
```

Word Count

```
KStream counts = source.flatMapValues(value->
        Arrays.asList(pattern.split(value.toLowerCase())))
                .map((key, value) -> new KeyValue<Object,
                Object>(value, value))
                .filter((key, value) -> (!value.equals("the")))
                .groupByKey()
                .count("CountStore").mapValues(value->
                Long.toString(value)).toStream();

counts.to("wordcount-output");
```

Word Count

```
KafkaStreams streams = new KafkaStreams(builder, props);

streams.start();

// usually the stream application would be running
        forever,
// in this example we just let it run for some time and
        stop since the input data is finite.
Thread.sleep(5000L);

streams.close();
}
}
```
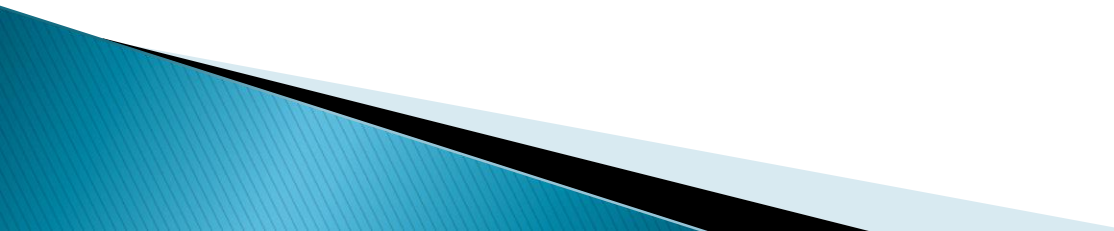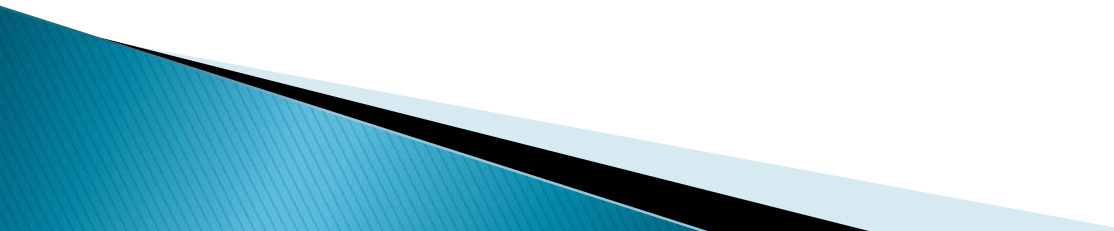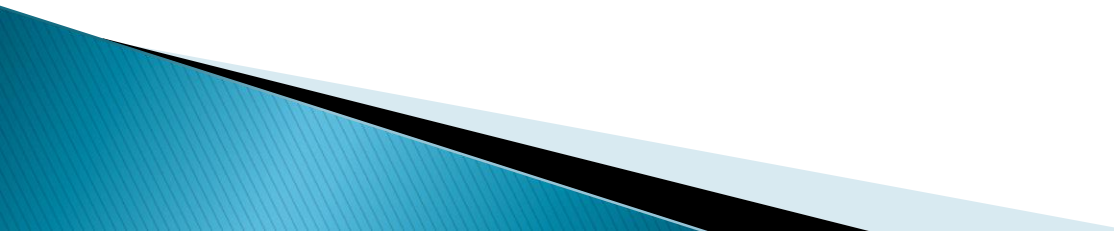
Word Count

- If your input topic contains multiple partitions, you can run multiple instances of the WordCount application (just run the app in several different terminal tabs) and you have your first Kafka Streams processing cluster.

- The instances of the WordCount application talk to each other and coordinate the work.

- With the Kafka's Streams API, you just start multiple instances of your app - and you have a cluster.

- The exact same app is running on your development machine and in production.
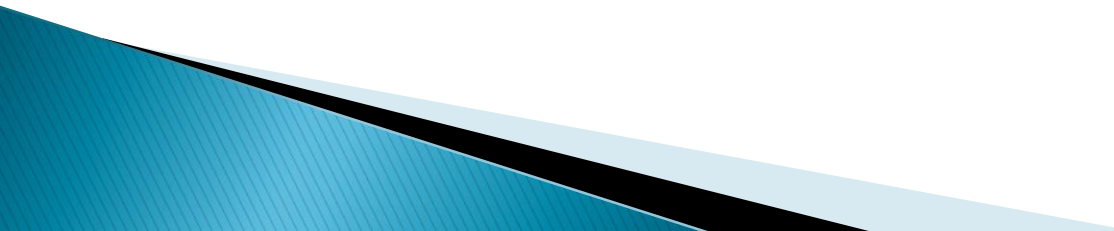
Stock Market Statistics

- We will read a stream of stock market trading events that include the stock ticker, ask price, and ask size.

- In stock market trades, *ask price* is what a seller is asking for whereas *bid price* is what the buyer is suggesting to pay.

- *Ask size* is the number of shares the seller is willing to sell at that price.

- For simplicity of the example, we'll ignore bids completely.

- We also won't include a timestamp in our data; instead, we'll rely on event time populated by our Kafka producer.
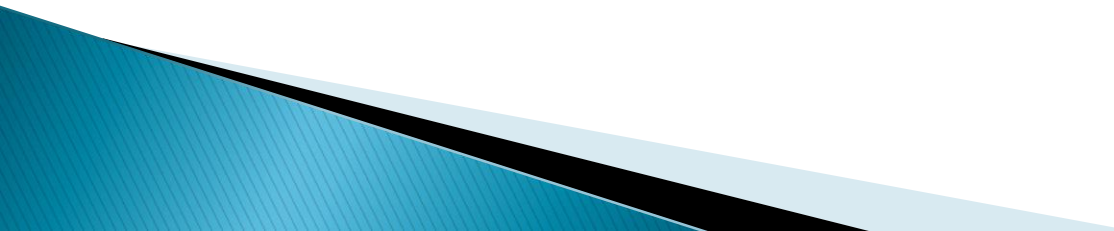
Stock Market Statistics

- We will then create output streams that contains a few windowed statistics:
    - Best (i.e., minimum) ask price for every five-second window
    - Number of trades for every five-second window
    - Average ask price for every five-second window

- All statistics will be updated every second.
- For simplicity, we'll assume our exchange only has 10 stock tickers trading in it.

Stock Market Statistics

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG,
"stockstat");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
Constants.BROKER);
props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());
props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
TradeSerde.class.getName());
```
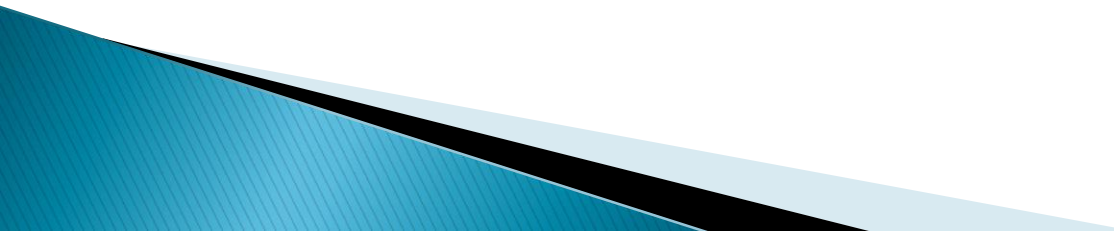
Stock Market Statistics

- In this example, the key is still a string, but the value is a Trade object that contains the ticker symbol, ask price, and ask size.

- In order to serialize and deserialize this object (and a few other objects we used in this small app), we used the Gson library from Google to generate a JSon serializer and deserializer from our Java object.

- Then created a small wrapper that created a Serde object from those.

Stock Market Statistics

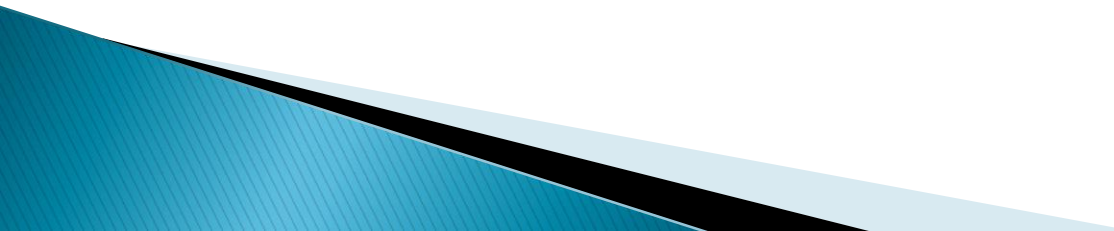Here is how we created the Serde:

```
static public final class TradeSerde extends
WrapperSerde<Trade> {
 public TradeSerde() {
  super(new JsonSerializer<Trade>(),
    new JsonDeserializer<Trade>(Trade.class));
  }
}
```

Stock Market Statistics

```
KStream<TickerWindow, TradeStats> stats = source.groupByKey()
        .aggregate(TradeStats::new,
                (k, v, tradestats) -> tradestats.add(v),
                TimeWindows.of(5000).advanceBy(1000),
                new TradeStatsSerde(),
                "trade-stats-store")
        .toStream((key, value) -> new TickerWindow(key.key(),
                key.window().start()))
        .mapValues((trade) -> trade.computeAvgPrice());

stats.to(new TickerWindowSerde(), new TradeStatsSerde(),
        "stockstats-output");
```

Stock Market Statistics

- This example shows how to perform windowed aggregation on a stream - probably the most popular use case of stream processing.

- One thing to notice is how little work was needed to maintain the local state of the aggregation - just provide a Serde and name the state store.

- Yet this application will scale to multiple instances and automatically recover from a failure of each instance by shifting processing of some partitions to one of the surviving instances.
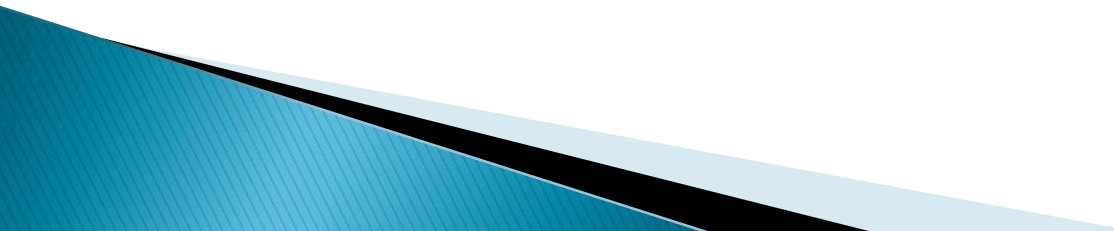
Click Stream Enrichment

- Will demonstrate streaming joins by enriching a stream of clicks on a website.

- We will generate a stream of simulated clicks, a stream of updates to a fictional profile database table, and a stream of web searches.

- We will then join all three streams to get a 360-view into each user activity.

- What did the users search for? What did they click as a result? Did they change their "interests" in their user profile? These kinds of joins provide a rich data collection for analytics.

- Product recommendations are often based on this kind of information - user searched for bikes, clicked on links for "Trek," and is interested in travel, so we can advertise bikes from Trek, helmets, and bike tours to exotic locations like Nebraska.

- Since configuring the app is similar to the previous examples, let's skip this part and take a look at the topology for joining multiple streams:

Click Stream Enrichment

```
KStream<Integer, PageView> views =
builder.stream(Serdes.Integer(),
new PageViewSerde(), Constants.PAGE_VIEW_TOPIC);

KStream<Integer, Search> searches =
builder.stream(Serdes.Integer(), new SearchSerde(),
Constants.SEARCH_TOPIC);

KTable<Integer, UserProfile> profiles =
builder.table(Serdes.Integer(), new ProfileSerde(),
Constants.USER_PROFILE_TOPIC, "profile-store");
```

Click Stream Enrichment

```
KStream<Integer, UserActivity> viewsWithProfile = views.leftJoin(profiles,
        (page, profile) -> new UserActivity(profile.getUserID(),
        profile.getUserName(), profile.getZipcode(),
        profile.getInterests(), "", page.getPage()));

KStream<Integer, UserActivity> userActivityKStream =
viewsWithProfile.leftJoin(searches,
        (userActivity, search) ->
        userActivity.updateSearch(search.getSearchTerms()),
        JoinWindows.of(1000), Serdes.Integer(),
        new UserActivitySerde(), new SearchSerde());
```

Click Stream Enrichment

- This example shows two different join patterns possible in stream processing.
- One joins a stream with a table to enrich all streaming events with information in the table.
- This is similar to joining a fact table with a dimension when running queries on a data warehouse.
- The second example joins two streams based on a time window.
- This operation is unique to stream processing.