CSCI E-88A Introduction to Functional and Stream Processing for Big Data Systems

Harvard University Extension, Spring 2020
Marina Popova, Edward Sumitra

Lecture 08 - Advanced Stream Processing

Agenda

- Admin: next Lab
  - review of the Mid-Term results
  - review advanced watermark concepts
- Reminder: Mid-Term Quiz is Due Mon, 03/30, noon, EST!
- parallelization in Streams - deeper dive
- stateful vs stateless stream operations
- advanced stream processing concepts
  - batch vs stream
  - window operations and concepts
  - triggers
  - watermarks

Parallelism in Streams - Recap

Lets see how execution of pipelines changes when running in parallel mode:

```
public static void executeAsNotParallel(int listSize){
    logger.info("executeAsNotParallel():");
    List<String> testList = StreamGeneration.getTestList(listSize);
    Stream<String> streamOfItems = testList.stream()
            .map(item -> mapItem(item))
            ;
    logger.info("streamOfItems.isParallel(): {}", streamOfItems.isParallel());
    streamOfItems.forEach(item -> handleItem(item));
}
```

```
> Task :test
cscie88a.streams.BasicParallelOperationsTest > testExecuteAsNotParallel() STARTED
21:56:04.231 [Test worker] INFO cscie88a.streams.BasicParallelOperations - number of available Cores: 12
21:56:04.235 [Test worker] INFO cscie88a.streams.BasicParallelOperations - executeAsNotParallel():
21:56:04.240 [Test worker] INFO cscie88a.streams.BasicParallelOperations - streamOfItems.isParallel(): false
21:56:04.240 [Test worker] INFO cscie88a.streams.BasicParallelOperations - in mapItem: item_0
21:56:04.250 [Test worker] INFO cscie88a.streams.BasicParallelOperations - I'm handling item: item_0_mapped
21:56:04.268 [Test worker] INFO cscie88a.streams.BasicParallelOperations - in mapItem: item_1
21:56:04.277 [Test worker] INFO cscie88a.streams.BasicParallelOperations - I'm handling item: item_1_mapped
21:56:04.293 [Test worker] INFO cscie88a.streams.BasicParallelOperations - in mapItem: item_2
21:56:04.303 [Test worker] INFO cscie88a.streams.BasicParallelOperations - I'm handling item: item_2_mapped
```

Parallelism in Streams - recap

```java
public static void executeAsParallelAfterMap(int listSize){
    logger.info("executeAsParallelAfterMap():");
    List<String> testList = StreamGeneration.getTestList(listSize);
    Stream<String> streamOfItems = testList.stream()
            .map(item -> mapItem(item))
            .parallel();
    logger.info("streamOfItems.isParallel(): {}", streamOfItems.isParallel());
    streamOfItems.forEach(item -> handleItem(item));
}
```

```
> Task :test
cscie88a.streams.BasicParallelOperationsTest > testExecuteAsParallelAfterMap() STARTED
21:58:18.653 [Test worker] INFO cscie88a.streams.BasicParallelOperations - number of available Cores: 12
21:58:18.657 [Test worker] INFO cscie88a.streams.BasicParallelOperations - executeAsParallelAfterMap():
21:58:18.661 [Test worker] INFO cscie88a.streams.BasicParallelOperations - streamOfItems.isParallel(): true
21:58:18.664 [Test worker] INFO cscie88a.streams.BasicParallelOperations - in mapItem: item_12
21:58:18.665 [ForkJoinPool.commonPool-worker-27] INFO cscie88a.streams.BasicParallelOperations - in mapItem: item_1
21:58:18.665 [ForkJoinPool.commonPool-worker-31] INFO cscie88a.streams.BasicParallelOperations - in mapItem: item_1
21:58:18.666 [ForkJoinPool.commonPool-worker-19] INFO cscie88a.streams.BasicParallelOperations - in mapItem: item_1
21:58:18.666 [ForkJoinPool.commonPool-worker-23] INFO cscie88a.streams.BasicParallelOperations - in mapItem: item_1
21:58:18.666 [ForkJoinPool.commonPool-worker-13] INFO cscie88a.streams.BasicParallelOperations - in mapItem: item_1
21:58:18.666 [ForkJoinPool.commonPool-worker-5] INFO cscie88a.streams.BasicParallelOperations - in mapItem: item_6
```

Parallelism in Streams - recap

What did we notice?

- streams are marked as parallel via the parallel() method
- parallel pipeline is executed in multiple threads!
- the threads are coming from a thread pool (default or custom)

Who creates those threads and thread pools?
NOT YOU!!

```java
public static void executeWithCustomPool(int listSize) {
    logger.info("executeWithCustomPool():");
    ForkJoinPool myThreadPool = new ForkJoinPool(3);
    List<String> testList = StreamGeneration.getTestList(listSize);

    myThreadPool.submit(() -> {
        Stream<String> streamOfItems = testList.stream()
                .parallel()
                .map(item -> mapItem(item))
                ;
        logger.info("streamOfItems.isParallel(): {}", streamOfItems.isParallel());
        streamOfItems.forEach(item -> handleItem(item));
    }).join();
}
```

```
> Task :test
cscie88a.streams.BasicParallelOperationsTest > testExecuteWithCustomPool() STARTED
22:17:39.200 [Test worker] INFO cscie88a.streams.BasicParallelOperations - number of available Cores: 12
22:17:39.204 [Test worker] INFO cscie88a.streams.BasicParallelOperations - executeWithCustomPool():
22:17:39.211 [ForkJoinPool-1-worker-3] INFO cscie88a.streams.BasicParallelOperations - streamOfItems.isParallel(): true
22:17:39.213 [ForkJoinPool-1-worker-3] INFO cscie88a.streams.BasicParallelOperations - in mapItem: item_12
22:17:39.213 [ForkJoinPool-1-worker-7] INFO cscie88a.streams.BasicParallelOperations - in mapItem: item_17
22:17:39.213 [ForkJoinPool-1-worker-5] INFO cscie88a.streams.BasicParallelOperations - in mapItem: item_6
22:17:39.223 [ForkJoinPool-1-worker-3] INFO cscie88a.streams.BasicParallelOperations - I'm handling item: item_12_mapped
22:17:39.223 [ForkJoinPool-1-worker-7] INFO cscie88a.streams.BasicParallelOperations - I'm handling item: item_17_mapped
22:17:39.223 [ForkJoinPool-1-worker-5] INFO cscie88a.streams.BasicParallelOperations - I'm handling item: item_6_mapped
22:17:39.239 [ForkJoinPool-1-worker-5] INFO cscie88a.streams.BasicParallelOperations - in mapItem: item_5
```

Parallelization in Java: Basics

Remember this?

"Java 8 introduces a concept of a Stream that allows the programmer to process data descriptively and rely on a multi-core architecture without the need to write any special code."

Where the "special code" is the code that starts and runs operations in parallel, using Threads and Thread Pools, and is done by the Streams library internally!

Lets look under the hood of this internal thread management by Streams ...

Thread Pools in Java

- There are many different ThreadPools in Java
- the most important for the MapReduce-like / parallel pipelines is the **ForkJoinPool** pool
- this pool is used by the Stream internally
- it is in java.util.concurrent package since Java 7
- ForkJoinPool is a special kind of a more generic **ExecutorService** in Java, that enables thread pool creation, operation and management; in essence it does:
    - creates pools of threads
    - accepts tasks
    - assigns them to a pool of threads
    - pool of threads will execute the tasks

let's see how it works ...

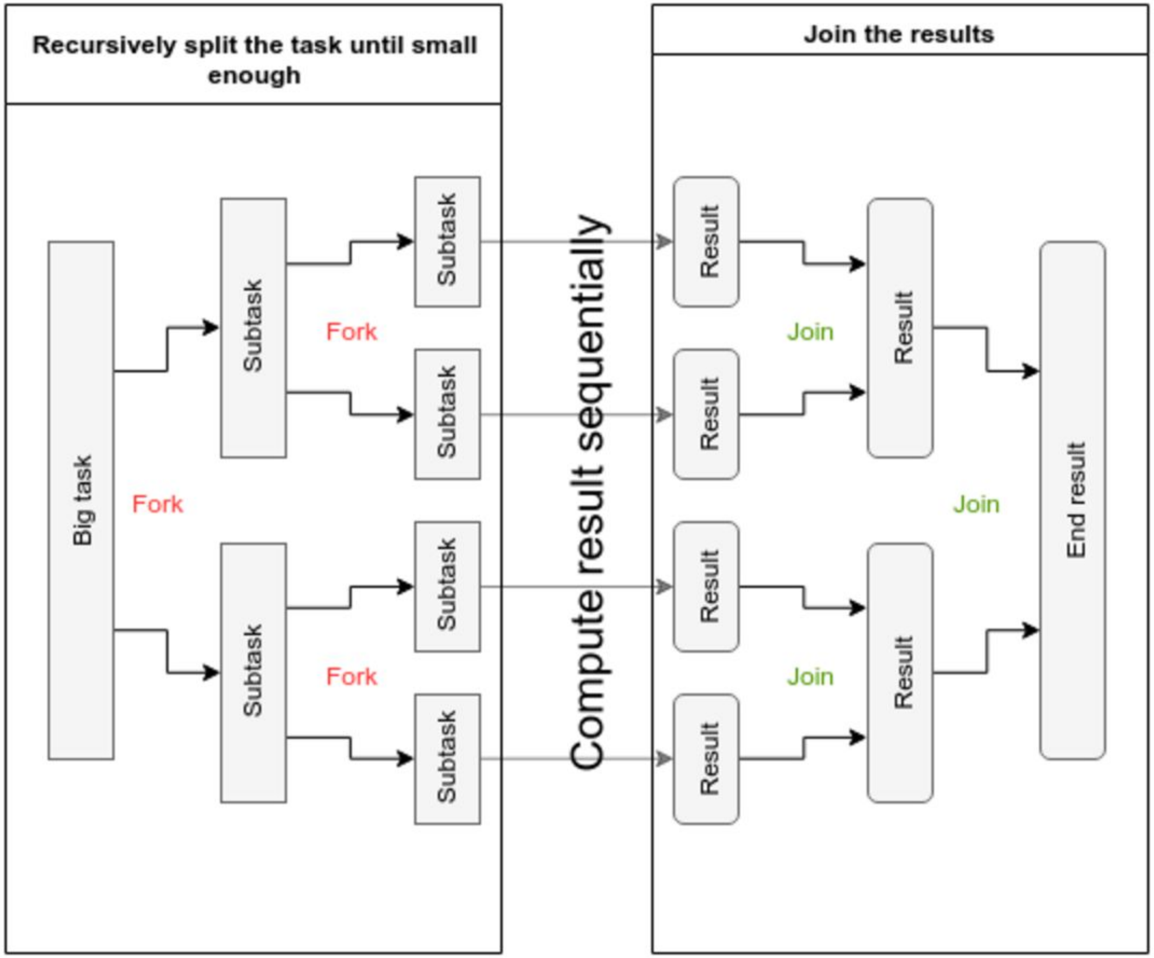
Thread Pools in Java

Ref: http://blog.indrek.io/articles/the-fork-slash-join-framework-in-java/

- ForkJoinTask is the base class for all tasks that can be assigned to the ForkJoinPool
- It is a special kind of task that knows how to:
  - split itself into sub-tasks if needed (fork)
  - wait (join) until all subtasks are finished
  - merge the results of sub-tasks

The critical question is: how do tasks know how to split/fork themselves?

the magic is in the Spliterators !



@Marina Popova, Edward Sumitra

Spliterators - where the magic happens!

Ref: https://www.ibm.com/developerworks/library/j-java-streams-3-brian-goetz/index.html

- The splitting of the input stream is done by a **Spliterator**
- Spliterator combines two behaviors:
    - accessing the elements of the source (iterating)
    - and possibly decomposing the input source for parallel execution (splitting)

- to split the source, so that two threads can work separately on different sections of the input, Spliterator provides a ***trySplit()*** method:

    Spliterator<T> trySplit();

- The Collection implementations in the JDK have all been furnished with high-quality Spliterator implementations. Some sources have better implementations than others: an ArrayList with more than one element can always be split cleanly and evenly; a LinkedList always splits poorly; and hash-based and tree-based sets can generally be split reasonably well

- The internal implementation of Spliterator is using the default ForkJoinPool as a thread pool for the tasks
- The actual number of parallel tasks is determined by the number of available Cores

Stream library - benefits

If you do not use Streams and FP, but use imperative style of programming:

- You have to control how tasks are divided into sub-tasks yourself
- You have to be careful with state mutation
- Cannot use blocking operations (like I/O)
- you have to understand Java Memory Model and its implications for the multi-threaded access

And this is why using Streams and Stream pipeline operations is so important:
they provide a different way to process data in parallel without writing a low-level thread code

Ref: https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

In-depth explanation of stream parallelization in Java:
https://developer.ibm.com/articles/j-java-streams-2-brian-goetz/

Parallelism in Streams - Reduction recap

Mutable vs Immutable reduction

reduce

```
T reduce(T identity,
         BinaryOperator<T> accumulator)
```

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);
```

reduce

```
<U> U reduce(U identity,
             BiFunction<U,? super T,U> accumulator,
             BinaryOperator<U> combiner)
```

reduce vs collect - recap

reduce() vs. collect(): immutable vs. mutable reduction!

https://stackoverflow.com/questions/24308146/why-is-a-combiner-needed-for-reduce-method-that-converts-type-in-java-8

```java
/**
 * reduce() vs collect()
 * Immutable reduction - via reduce() with 2 args
 * @param listSize
 */
public static void concatStringsAsReduce(int listSize){
    logger.info("concatStringsAsReduce():");
    List<String> testList = StreamGeneration.getTestList(listSize);
    String joinedString = testList.stream()
            .parallel()
            .reduce("",
                    (s1, s2) -> s1 + " " + s2);
    logger.info("Final joinedString = {}", joinedString);
}
```

```java
/**
 * reduce() vs collect()
 * Mutable reduction - via collect()
 * @param listSize
 */
public static void concatStringAsCollect(int listSize){
    logger.info("concatStringAsCollect():");
    List<String> testList = StreamGeneration.getTestList(listSize);
    StringBuilder stringBuilderContainer = testList.stream()
            .parallel()
            .collect(StringBuilder::new,
                    (sb, s1) -> sb.append(" ").append(s1),
                    (sb1, sb2) -> sb1.append(sb2.toString())
                    );
    String joinedString = stringBuilderContainer.toString();
    logger.info("Final joinedString = {}", joinedString);
}
```

Parallelism in Streams - Requirements

are ALL operations really executed in parallel ?

In order for stream operations to be truly parallelizable - these operations have to conform to some rules:

Most of the stream operations take a Lambda expression or function as an argument,

To be parallelizable, these Lambda functions have to be:

- **non-interfering** (they do not modify the stream source)
- **stateless** (their result should not depend on any state that might change during execution of the stream pipeline)
- **associative** (order of operations should not matter) (`a op b op c op d == (a op b) op (c op d)`)

Lets review them in more details ...
These requirements take slightly different shape for immutable vs. mutable reductions (we will see later)

Non-interference

non-interfering operations:

insuring non-interference means ensuring that the data source is *not modified at all* during the execution of the stream pipeline

Example: this operation is interfering as it can modify the data source:

```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.stream().forEach(x -> list.remove(x));
```

Important note:

- structural modification are NOT allowed
- not-structural (element-level) are OK

Stateless Behavior

Operation (Lambda) is stateless:

- if its result **does not depend on any state that might change** during execution of the stream pipeline

Without this requirement behavior of a parallel pipeline is **non-deterministic** - *which is a disaster waiting to happen …*

Example of a stateful operation:

```
Set<Integer> seen =
Collections.synchronizedSet(new HashSet<>());

stream.parallel().map(
      e -> {
            if (seen.add(e)) return 0;
            else return e;
            }
)...
```

Non-interference

associative == order of operations should not matter

a op b op c op d == (a op b) op (c op d))

Good Ref (from Brian Goetz): https://developer.ibm.com/articles/j-java-streams-2-brian-goetz/

((a b) c) = (a (b c))

- associativity basically means that grouping doesn't matter
- If the binary operator is associative, the reduction can be safely performed in any order:
  - in a sequential execution, the natural order of execution is from left to right
  - in a parallel execution, the data is partitioned into segments, each segment reduced separately, and results are combined
  - associativity ensures that these two approaches yield the same answer
- this is easier to see if the definition of associativity is expanded to four terms: ***(((a b) c) d) = ((a b) (c d))***
  - the left side corresponds to a typical sequential computation
  - the right side corresponds to a partitioned execution that would be typical of a parallel execution where the input sequence is broken into parts, the parts reduced in parallel, and the partial results combined with

Examples:

(x,y) -> Math.max(x,y)

(x,y) -> x+y

Immutable Reduction - Parallelization requirements

For immutable reductions:

Parameters:

- **identity** - the identity value for the combiner function
- **accumulator** - an **associative, non-interfering, stateless** function for incorporating an additional element into a result
- **combiner** - an **associative, non-interfering, stateless** function for combining two values, which must be compatible with the accumulator function

reduce

```
<U> U reduce(U identity,
             BiFunction<U,? super T,U> accumulator,
             BinaryOperator<U> combiner)
```

Important requirement: **Identity value:**
it is a value such that **x op identity = x.**

Examples:

- *String concatenated = strings.stream().reduce(*
  ***""**, String::concat);*
- *int sum = Stream.of(ints).reduce(**0**, (x,y) -> x+y);*
- *reduce(**true**, (a,b) -> a&&b)*
- *reduce(**false**, (a,b) -> a||b)*

Mutable Reduction - Parallelism

For mutable reductions:

Parameters:

- **supplier** - a function that creates a new result container. For a parallel execution, this function may be called multiple times and must return a fresh value each time
- **accumulator** - an **associative, non-interfering, stateless** function for incorporating an additional element into a result
- **combiner** - an **associative, non-interfering, stateless** function for combining two values, which must be compatible with the accumulator function

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);
```

The core requirements explained:

- the collector functions must satisfy an **identity** and an associativity constraints
- the **identity constraint** means that:
  - for any partially accumulated result, combining it with an empty result container must produce an equivalent result
- the **associativity constraint** means that:
  - splitting the computation in any way must produce an equivalent result.

Collectors - higher level wrappers

The class **Collectors** provides implementations of many common mutable reductions (Collectors) via static methods

Example of collecting Strings into an Array via generic collect() method vs a specialized one:

```
List<String> strings = testList.stream()
        .collect(
                () -> new ArrayList<>(),
                (c, e) -> c.add(e),
                (c1, c2) -> c1.addAll(c2)
        );
```

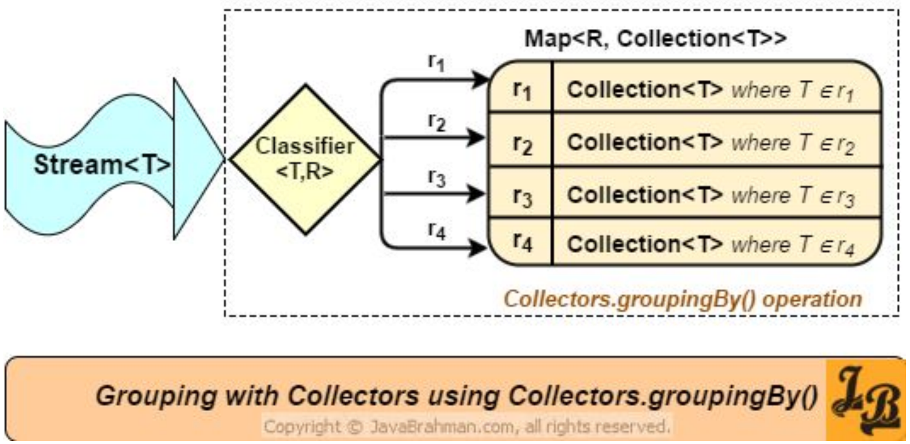
```
List<String> strings = testList.stream()
        .collect(
                Collectors.toList()
        );
```

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);
```

Collectors.groupingBy()

A very useful utility Collector, similar to SQL's groupBy operator
Ref: https://www.javabrahman.com/java-8/java-8-grouping-with-collectors-groupingby-method-tutorial-with-examples/

public static <T,K> Collector<T,?,Map<K,List<T>>>
groupingBy(Function<? super T,? extends K> classifier)



In simpler terms:

- input to groupingBy() is a classifier Function that takes an element (of type T) and returns a classification (value of type K or R on the picture) that this element belongs to
- output of the groupingBy() is a Collector , which, in turn, returns a Map such that:
- keys are all possible values from the classification function (type K) and values are the original stream elements that "fall" into this group

groupingBy() has three overloaded forms - next:

don't forget, in the end - it is the same collect() operation!

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);
```

Collectors.groupingBy()

groupingBy() has three overloaded forms, we'll consider the simples (one arg) and the most generic (three args) versions only

simplest form (one arg):

- one arg: classifier function

default values:

- downstream collector of results is the Collectors.toList() collector
- result map is a HashMap created via HashMap::new

```
public static void groupByRegularOneArg(int listSize){
    logger.info("groupByRegularOneArg():");
    List<String> testList = getListOfRandomStrings(listSize);
    Map<Integer, List<String>> stringsByLength = testList.stream()
            .parallel()
            .collect(
                    Collectors.groupingBy(e -> e.length())
            );
    stringsByLength.forEach(StateOperations::printMapEntry);
}
```

3 arg form arguments:

- classifier function
- constructor for the resulting Map (how the result Map should be created)
- downstream collector - specifies how the grouped elements should be collected

```
public static void groupByRegularThreeArg(int listSize){
    logger.info("groupByRegularThreeArg():");
    List<String> testList = getListOfRandomStrings(listSize);
    Map<Integer, List<String>> stringsByLength = testList.stream()
            .parallel()
            .collect(
                    Collectors.groupingBy(
                            e -> e.length(),
                            HashMap::new,
                            Collectors.toList())
            );
```

Collectors.groupingBy() vs groupingByConcurrent()

the groupBy() operation may not perform well in parallel pipelines - to address this, concurrent versions of groupBy() were added for each overloaded method:

```java
public static void groupByConcurrentOneArg(int listSize){
    logger.info("groupByConcurrentOneArg():");
    List<String> testList = getListOfRandomStrings(listSize);
    Map<Integer, List<String>> stringsByLength = testList.stream()
            .parallel()
            .collect(
                    Collectors.groupingByConcurrent(e -> e.length())
            );
    stringsByLength.forEach(StateOperations::printMapEntry);
}
```

```java
public static void groupByConcurrentThreeArg(int listSize){
    logger.info("groupByConcurrentThreeArg():");
    List<String> testList = getListOfTestStrings();
    Map<Integer, List<String>> stringsByLength = testList.stream()
            .parallel()
            .collect(
                    Collectors.groupingByConcurrent(
                            e -> e.length(),
                            ConcurrentHashMap::new,
                            Collectors.toList())
            );
    stringsByLength.forEach(StateOperations::printMapEntry);
```

Collectors.groupingBy() vs groupingByConcurrent()

- Streams framework splits the data source into chunks (fork phase)
- next, (join phase), the sub-tasks have to process their chunks of data - and this is where the difference between two methods shows up:

non-concurrent groupingBy	concurrent groupingBy
thread-safe? yes	thread-safe? yes
each chunk is processed by creating a local container (Map in this case) using the Collector's supplier (see earlier); accumulations are done into this local Map	only one ConcurrentMap (container) is created ; all threads/sub-tasks accumulate results into this same Map
partial results (from each sub-task/chunk) have to be merged by merging Maps	no merge is required !
many distinct keys? merge step is expensive!	many distinct keys? concurrent updates to the Map are cheap!
many duplicate keys? values for the same key have to be merged	many duplicate keys? contention on the same key may degrade performance

Performance might differ based on the specifics of the workload in the pipeline!

Ref: Holger's answer in this SO post:
https://stackoverflow.com/questions/41041698/why-should-i-use-concurrent-characteristic-in-parallel-stream-with-collect

More on Concurrent Reduction

From Java Docs: https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

- Collector that supports concurrent reduction is marked with the Collector.Characteristics.CONCURRENT characteristic
- however, if multiple threads are depositing results concurrently into a shared container, **the order in which results are deposited is non-deterministic**
- consequently, a concurrent reduction is only possible if ordering is not important for the stream being processed

The Stream.collect(Collector) implementation will only perform a concurrent reduction if

- the stream is parallel;
- the collector has the Collector.Characteristics.CONCURRENT characteristic, and
- either the stream is unordered, or the collector has the Collector.Characteristics.UNORDERED characteristic.

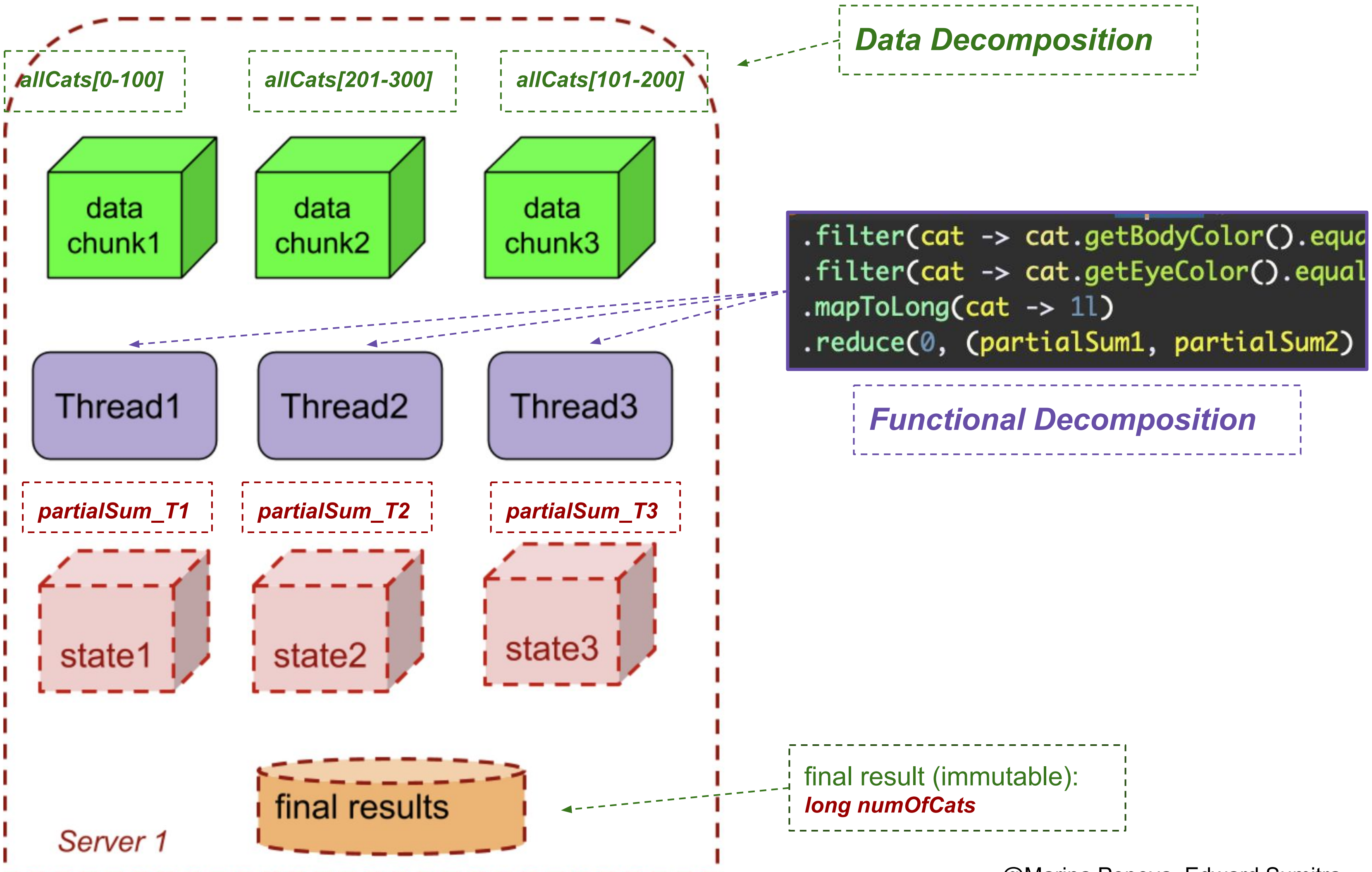
You can ensure the stream is unordered by using the BaseStream.unordered() method. For example:

```
Map<Buyer, List<Transaction>> salesByBuyer
    = txns.parallelStream()
          .unordered()
          .collect(groupingByConcurrent(Transaction::getBuyer));
```

Now you should really understand this ...

Imperative way - will NOT work here!

Functional way:



@Marina Popova, Edward Sumitra

Next: Moving to Advanced Stream Processing Concepts

Fundamental Concepts:

- Batch vs Stream processing
- event vs processing time
- windowing concepts
- triggers
- watermarks
- accumulators

Resources

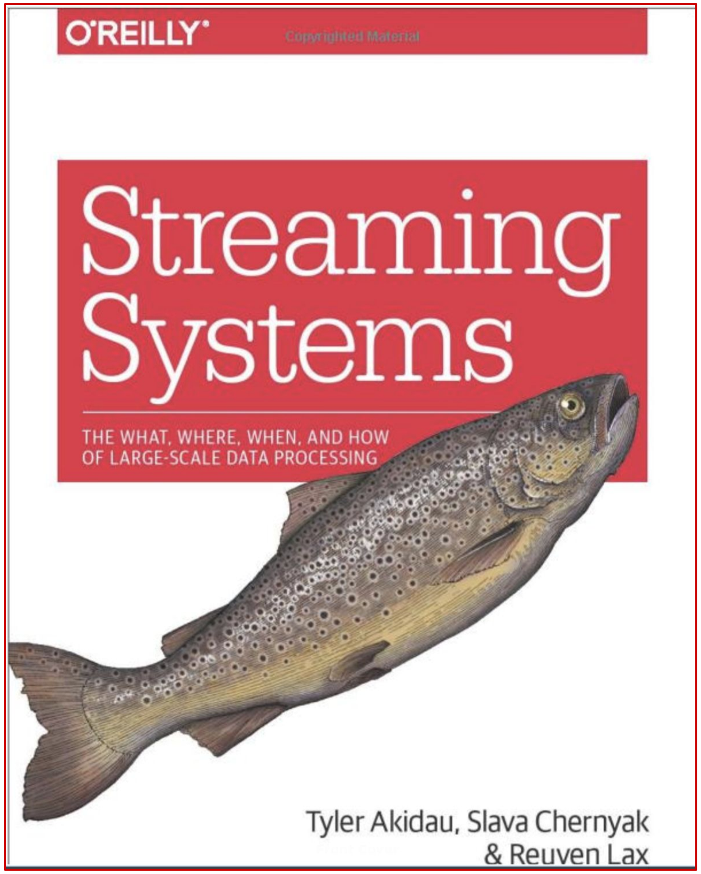
"Streaming Systems"
by Tyler Akidau, Slava Chernyak, Reuven Lax

Amazon:
https://www.amazon.com/Streaming-Systems-Where-Large-Scale-Processing/dp/1491983876

available through your Harvard account at
https://learning-oreilly-com.ezp-prod1.hul.harvard.edu/library/view/streaming-systems/9781491983867/

Free 5 day trial that allows you to download this book in PDF:
http://www.ebookdownl.com/book.php?i=11&g=book&b=496481&n=

Animated images:
http://www.streamingbook.net/fig



Stream vs Batch Processing - Generalization

The main differences between and characteristics of static and stream data and processing:

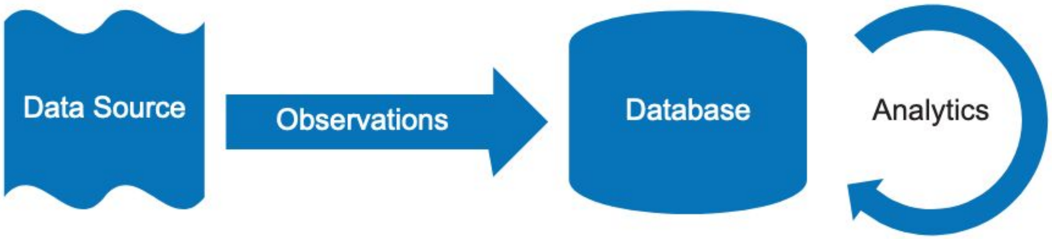
- nature of the date (bounded or unbounded)
- where and when the data is stored
- when is data analyzed
- how is data analyzed? - query models

Static Data and Batch Processing

- data is bounded - has a limit

This type of data is often called "at-rest" data

- data is stored in the database (data source) before it is processed
- processing (analytics) is done on the stored data later on, by querying the data store



- how is data analyzed?

Static Dataset Queries: query is submitted and results are returned to the client/app

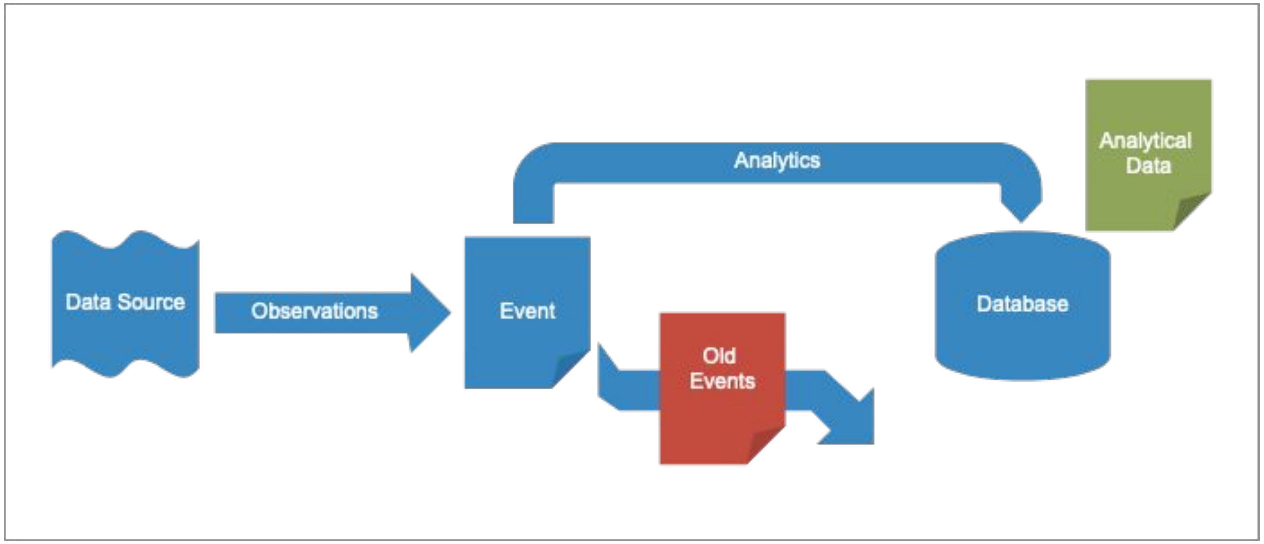
Good summary and visualization:
https://documentation.sas.com/?docsetId=esptex&docsetTarget=streaming-static.htm&docsetVersion=4.3&locale=en

Streaming data and Stream Processing

- data is unbounded - has NO limit

This type of data is often called "in-flight" or "real-time" data

- data is analyzed before it is stored
- it is often the results of the processing that are stored into the further data storage
- the events themselves are either discarded or shipped to a historical storage



- how is data analyzed?

Continuous Queries: query is registered by the application when it starts, results are computed continuously, and returned to the client all the time (periodically) for as long as the application runs (or wants the results)

It is a "one-pass" processing model - you get to touch the data only once; no iterations are possible

Event Time vs Processing Time

Static/Batch systems: any timestamps associated with events as they are stored in are always event times; thus all time-based aggregations are 100% correct from the time perspective

Streaming systems: this is no longer true….

- **Event time:** when an event took place
- **Processing (Stream) time**: when the event entered the streaming system

They can be different for many reasons:

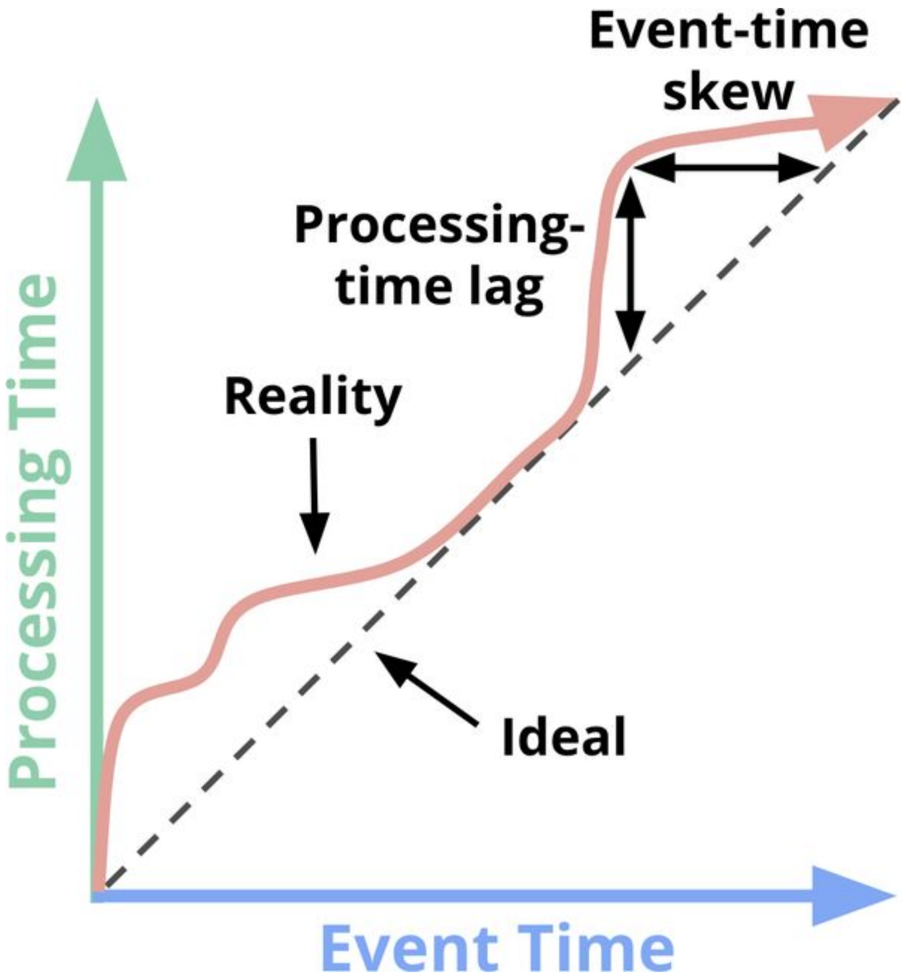
- Delay in the collection tier due to network slowdown or bursts of incoming data
- Out-of-order/old events due to failures and reprocessing in the collection or messaging tiers
- Delays in the actual processing of the event due to slowness of the streaming tier - under increased load

Event Time vs Processing Time

This difference is called **"time skew"** and it has direct impact on some of the streaming computations

Some frameworks support one more type of "time":

Ingestion time: a hybrid of processing and event time. It assigns wall clock timestamps to records as soon as they arrive in the system (at the source) and continues processing with event time semantics based on the attached timestamps



Windowing Operations

Why do we need "windows" ? When processing static data - you may (*) be able to process it all at once

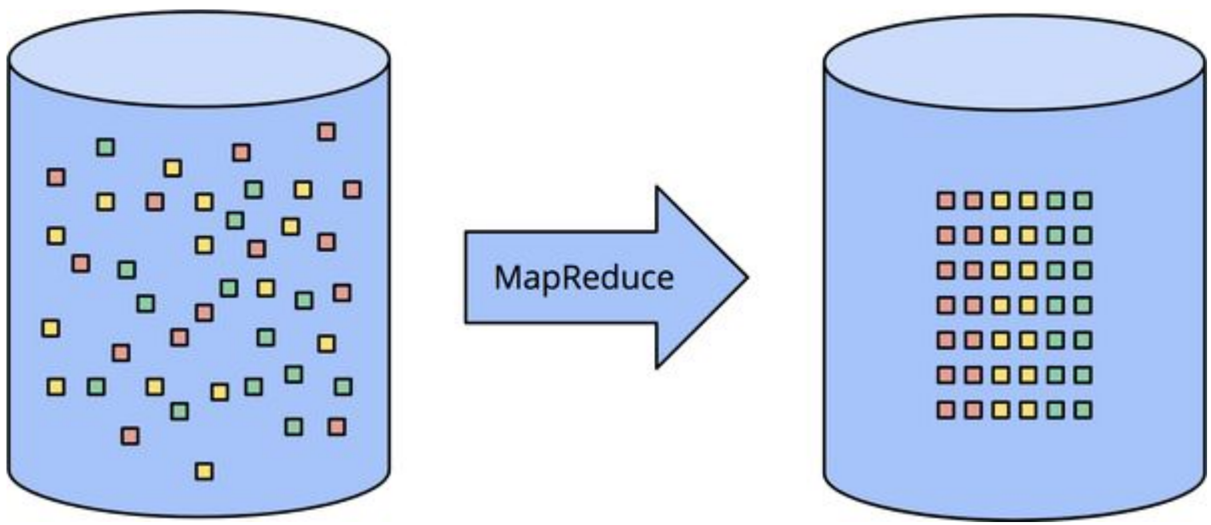


Figure 1-2. Bounded data processing with a classic batch engine. A finite pool of unstructured data on the left is run through a data processing engine, resulting in corresponding structured data on the right.

**** NOTE: for real BIG data that is not the case, and the same windowing techniques can be used - only they are called data splitting/chunking or…. data decomposition !***

Windowing Operations

For streaming data - there is no natural "boundaries" for data, so we have to invent some:

"window" - a base unit for windowing operations in streaming systems

- It is needed since there is no other natural boundaries for data - streaming data is endless
- Generally speaking, a window defines a *finite set of elements* on an unbounded stream.
- This set can be based on time, element counts, a combination of counts and time, or some custom logic to assign elements to windows

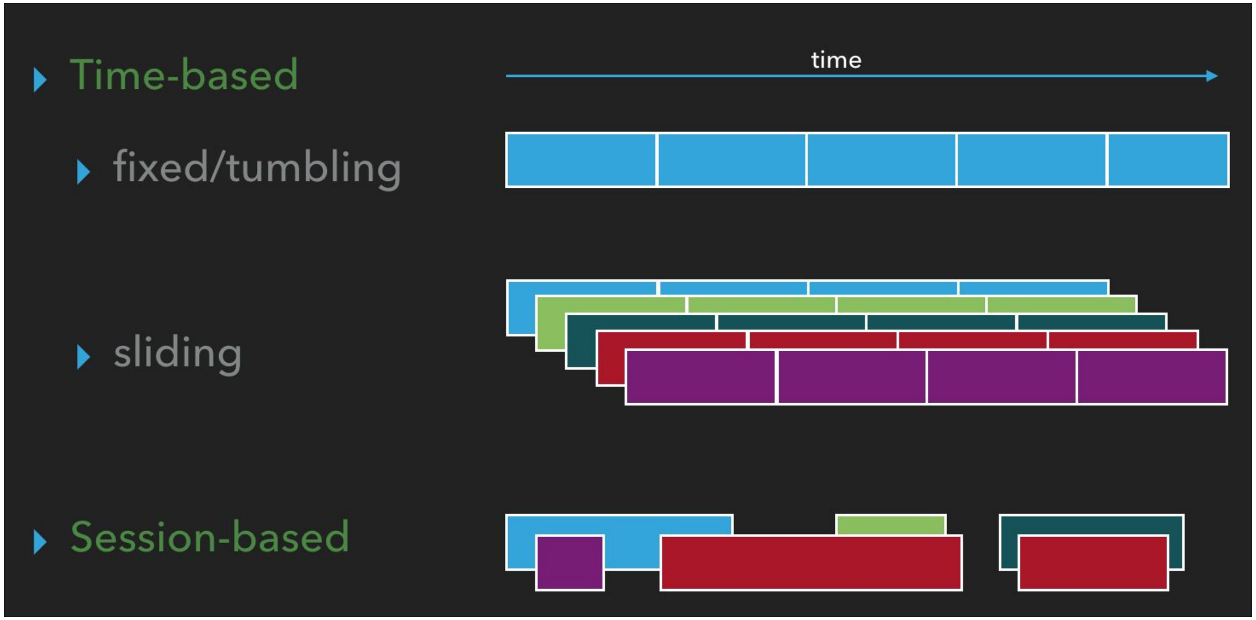
Attributes common for all windowing techniques - which we will discuss later

- **Trigger policy**: defines the rules that dictate when it is time to process all data in the window - window is "complete"
- **Eviction policy**: defines the rules to decide if a specific event should be evicted from the window

Trigger and Eviction policies will be later generalized into more comprehensive Allowed Lateness and State management concepts, but for now - they give you an idea of challenges that have to be solved for stream processing

Stream Processing: Windowing Operations

Types of windows:
Ref: https://softwaremill.com/windowing-in-big-data-streams-spark-flink-kafka-akka/



Stream Processing: Windowing Operations

Nice explanation of different windowing techniques: https://flink.apache.org/news/2015/12/04/Introducing-windows.html

Example: traffic sensor that counts every 15 seconds the number of vehicles passing a certain location. The resulting stream could look like:

sensor → , 9, 6, 8, 4, 7, 3, 8, 4, 2, 1, 3, 2, → out

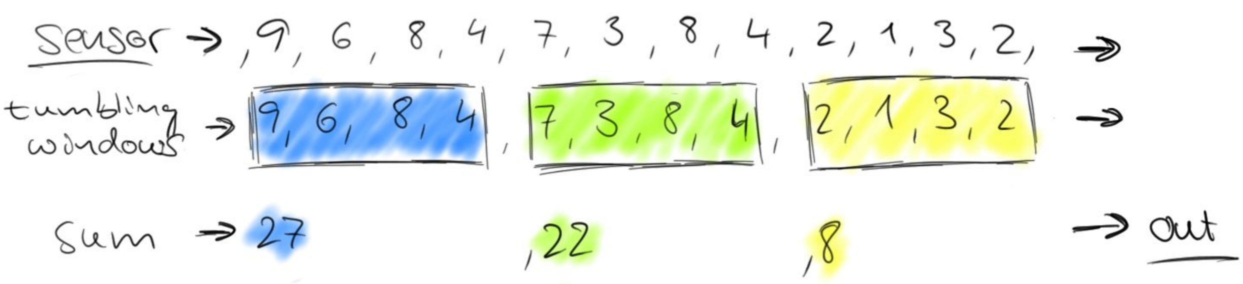
Without using any windows, we can only ask "Ad-hoc" queries like: get a rolling sum of all vehicles passed **so far**:
By computing rolling sums, we return for each input event an updated sum record. This would yield a new stream of partial sums:

sensor → , 9, 6, 8, 4, 7, 3, 8, 4, 2, 1, 3, 2, →

rolling sum → , 57, 48, 42, 34, 30, 23, 20, 12, 8, 6, 5, 2, → out

Tumbling Windows

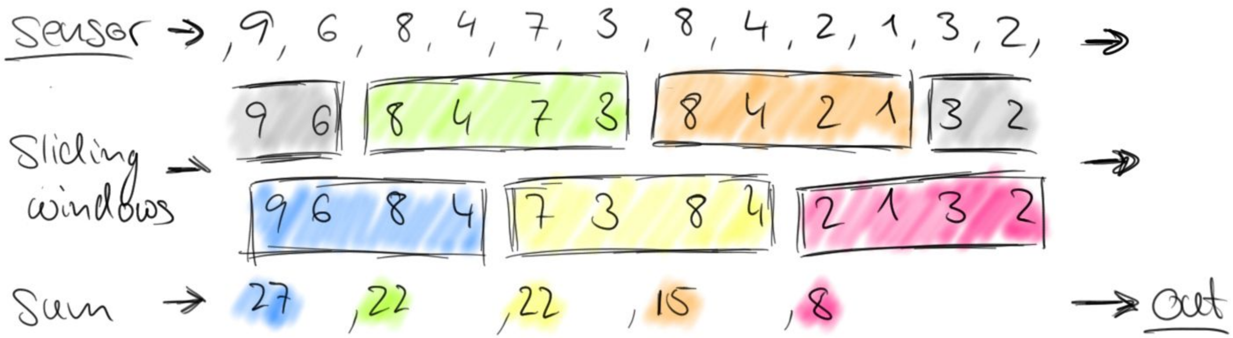
In the above example, some important information such as variation over time is lost.
Hence, we might want to rephrase our question and ask for the number of cars that pass the location **every minute**.
This requires us to group the elements of the stream into finite sets, each set corresponding to 60 seconds.
This operation is called a *tumbling windows* operation:



Main characteristics of Tumbling Windows

- Tumbling windows discretize a stream into **non-overlapping windows**.
- Eviction policy: window is full (60 sec in the example)
- Trigger policy can be of many types:
  - time-based (length of the window)
  - count-based
- The above example is the *time-based* tumbling window, with the trigger policy = 60 sec

Sliding Windows



Trigger and Eviction policies are based on time:
Eviction: defined by the window length - duration of time that data is retained and available for processing

Trigger: is defined by the sliding interval: each time sliding interval is reached - our code will be notified to start processing the window

Big difference with the Tumbling windows:
The same event can fall into multiple windows!

Using the example above, we may want to compute smoothed aggregates. For example, we can compute every thirty seconds the number of cars passed in the last minute

Session windows

- Can have various sizes and are defined basing on data, which should carry some session identifiers;
- Sessions are typically defined as periods of activity (e.g., for a specific user) terminated by a gap of inactivity.

There can be other variations of the windowing techniques, such as **hopping windows and snapshot windows**, but they are less commonly used.

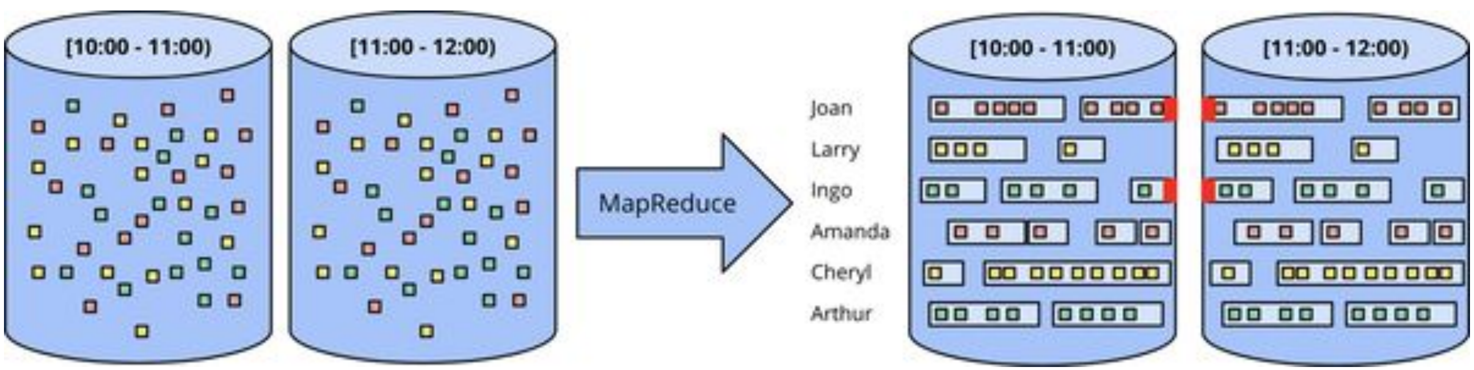


Figure 1-4. Unbounded data processing into sessions via ad hoc fixed windows with a classic batch engine. An unbounded dataset is collected up front into finite, fixed-size windows of bounded data that are then subdivided into dynamic session windows via successive runs a of classic batch engine.

Stream Processing: Windowing Operations

All windowing techniques can work in both time domains:
Event-based and Processing-based

The simplest form is Processing-time based:

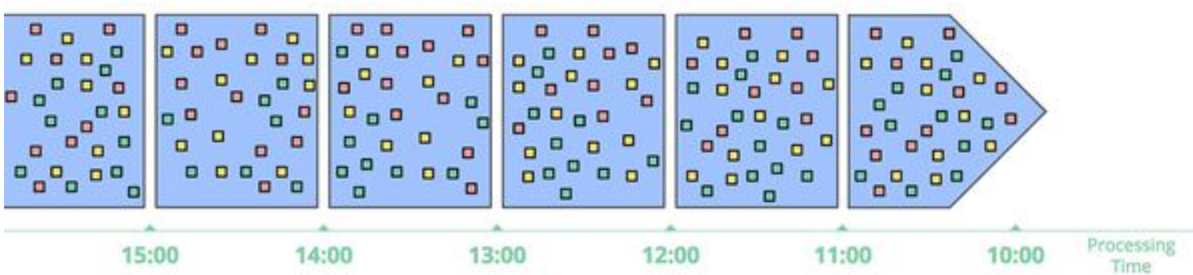


Figure 1-9. Windowing into fixed windows by processing time. Data are collected into windows based on the order they arrive in the pipeline.

Stream Processing: Windowing Operations

In reality - events are never recieved in real time, immediately

Event-time based windows and processing:

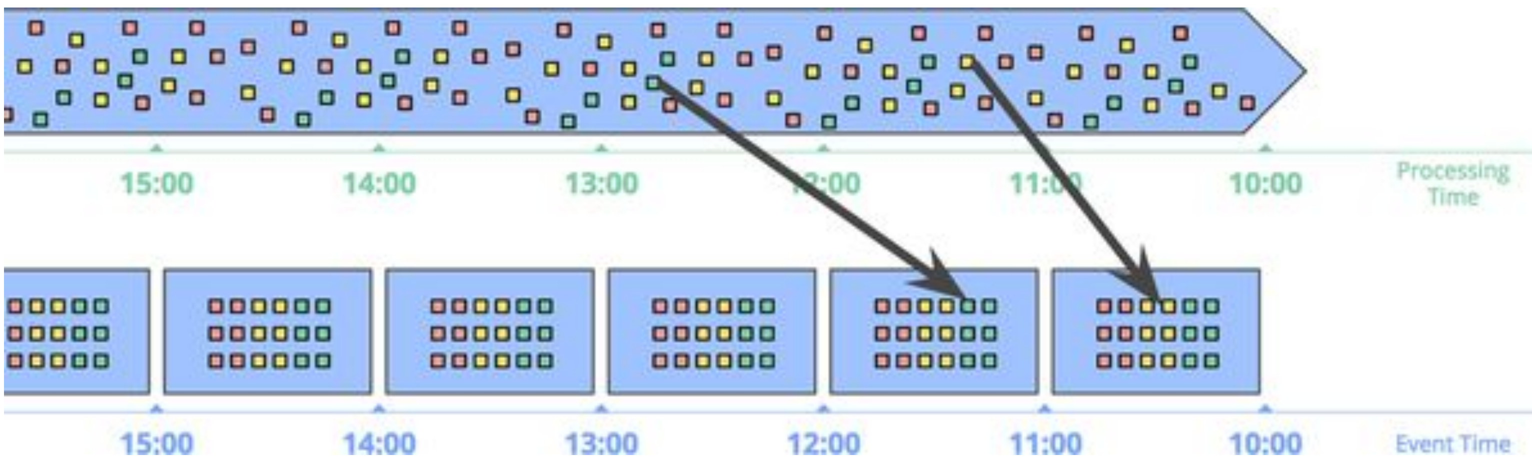


Figure 1-10. Windowing into fixed windows by event time. Data are collected into windows based on the times at which they occurred. The black arrows call out example data that arrived in processing-time windows that differed from the event-time windows to which they belonged.

Stream Processing: Windowing Operations

Event-time based windows and processing with session-based windows

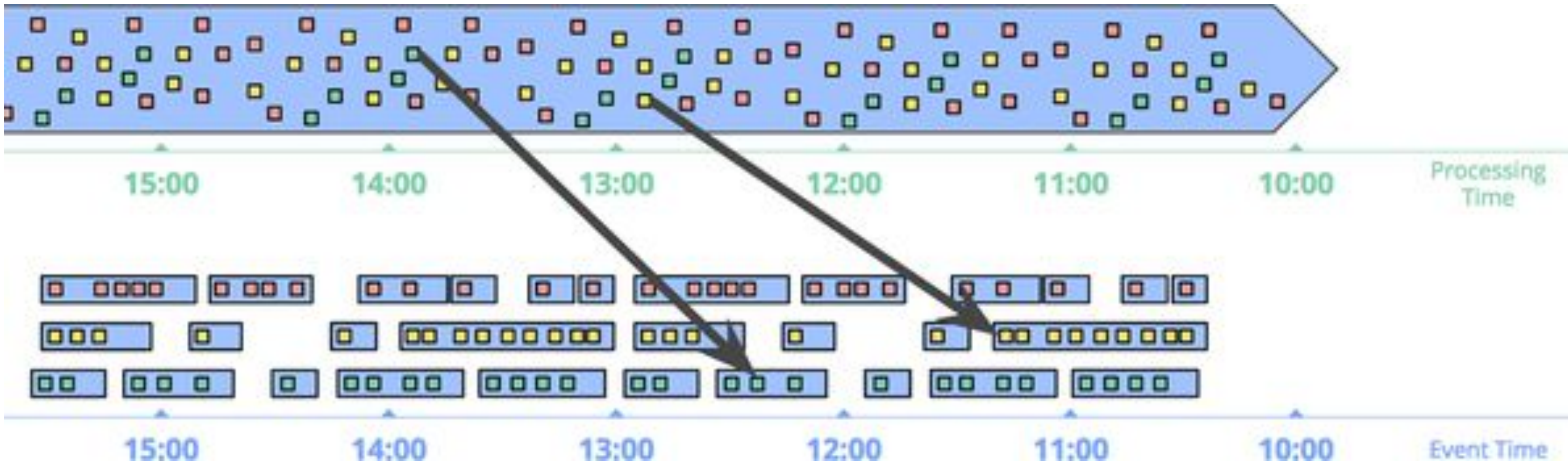


Figure 1-11. Windowing into session windows by event time. Data are collected into session windows capturing bursts of activity based on the times that the corresponding events occurred. The black arrows again call out the temporal shuffle necessary to put the data into their correct event-time locations.

Stream Processing: Windowing Operations

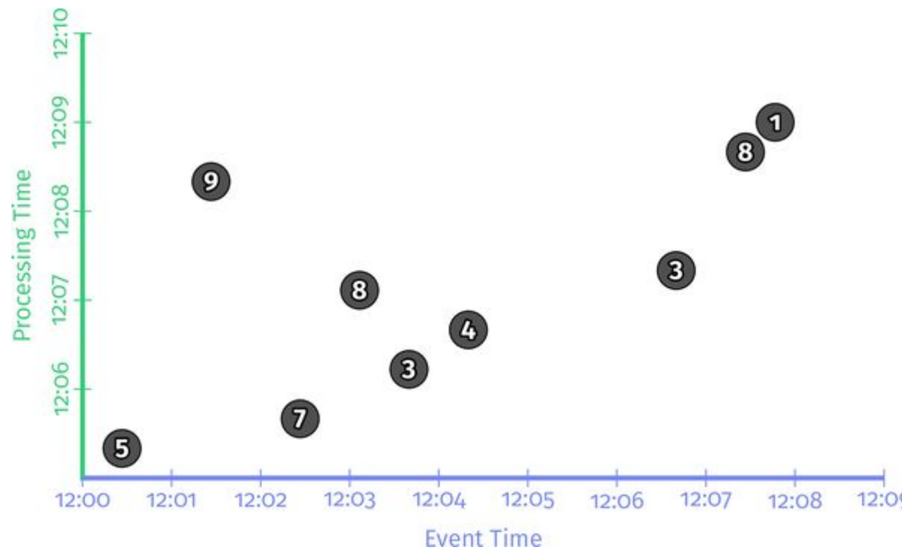
Challenges with event-time-based window operations:

- Buffering
  - in order to put events into correct windows - we have to keep "incomplete" windows for a longer time
  - more resources are used: disk or RAM
  - the raw events also may have to be buffered - for non-idempotent aggregation operations
- Completeness
  - how do you know when a window is "complete"?
  - you don't , really
  - but can guess based on some business knowledge of the data
  - or use probabilistic algorithms to estimate (heuristics)
  - that's what watermarks are for ….

Stream Processing: Example Data

We will use the test data from the "streaming Systems" examples:

Name	Team	Score	EventTime	ProcTime
Julie	TeamX	5	12:00:26	12:05:19
Frank	TeamX	9	12:01:26	12:08:19
Ed	TeamX	7	12:02:26	12:05:39
Julie	TeamX	8	12:03:06	12:07:06
Amy	TeamX	3	12:03:39	12:06:13
Fred	TeamX	4	12:04:19	12:06:39
Naomi	TeamX	3	12:06:39	12:07:19
Becky	TeamX	8	12:07:26	12:08:39
Naomi	TeamX	1	12:07:46	12:09:00

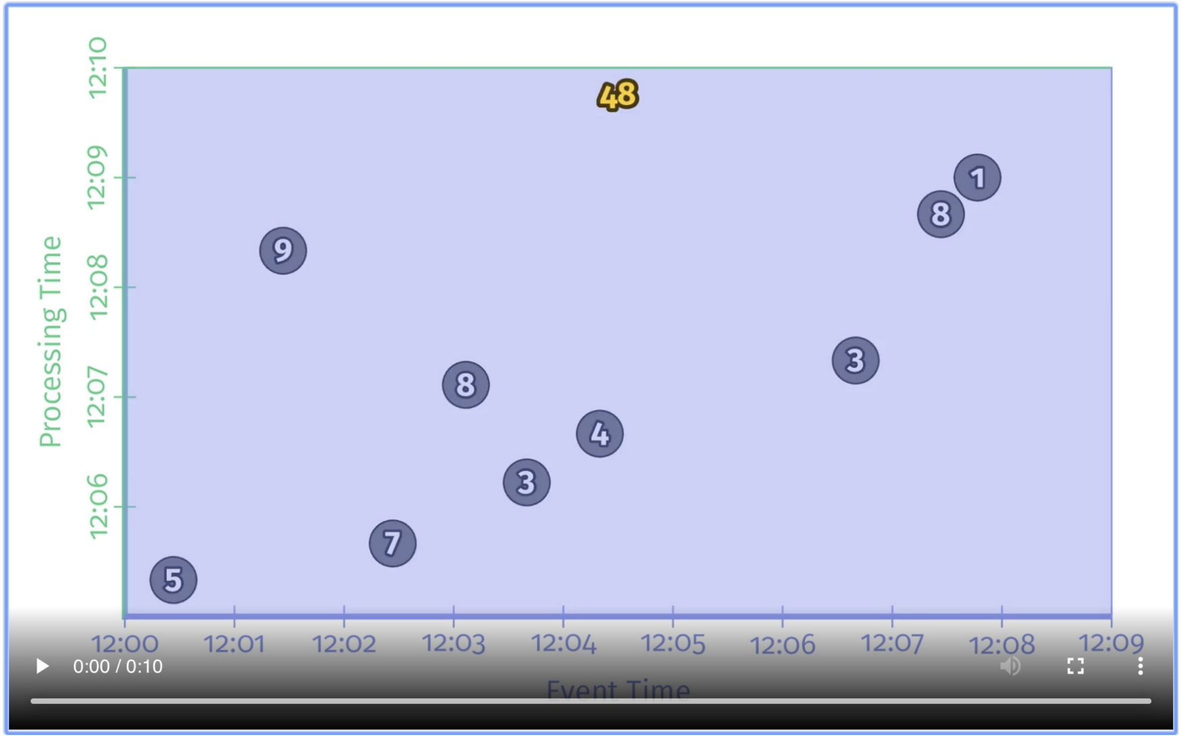


What are we calculating? Total sum of scores by team
Key: team name
Value: sum of scores of all players on that team

@Marina Popova, Edward Sumitra

Stream Processing: Example

Data processing in Batch mode:
https://learning-oreilly-com.ezp-prod1.hul.harvard.edu/library/view/streaming-systems/9781491983867/assets/stsy_0203.mp4



Stream Processing: Example

Data processing in Windowed Batch mode:
https://learning-oreilly-com.ezp-prod1.hul.harvard.edu/library/view/streaming-systems/9781491983867/assets/stsy_0205.mp4

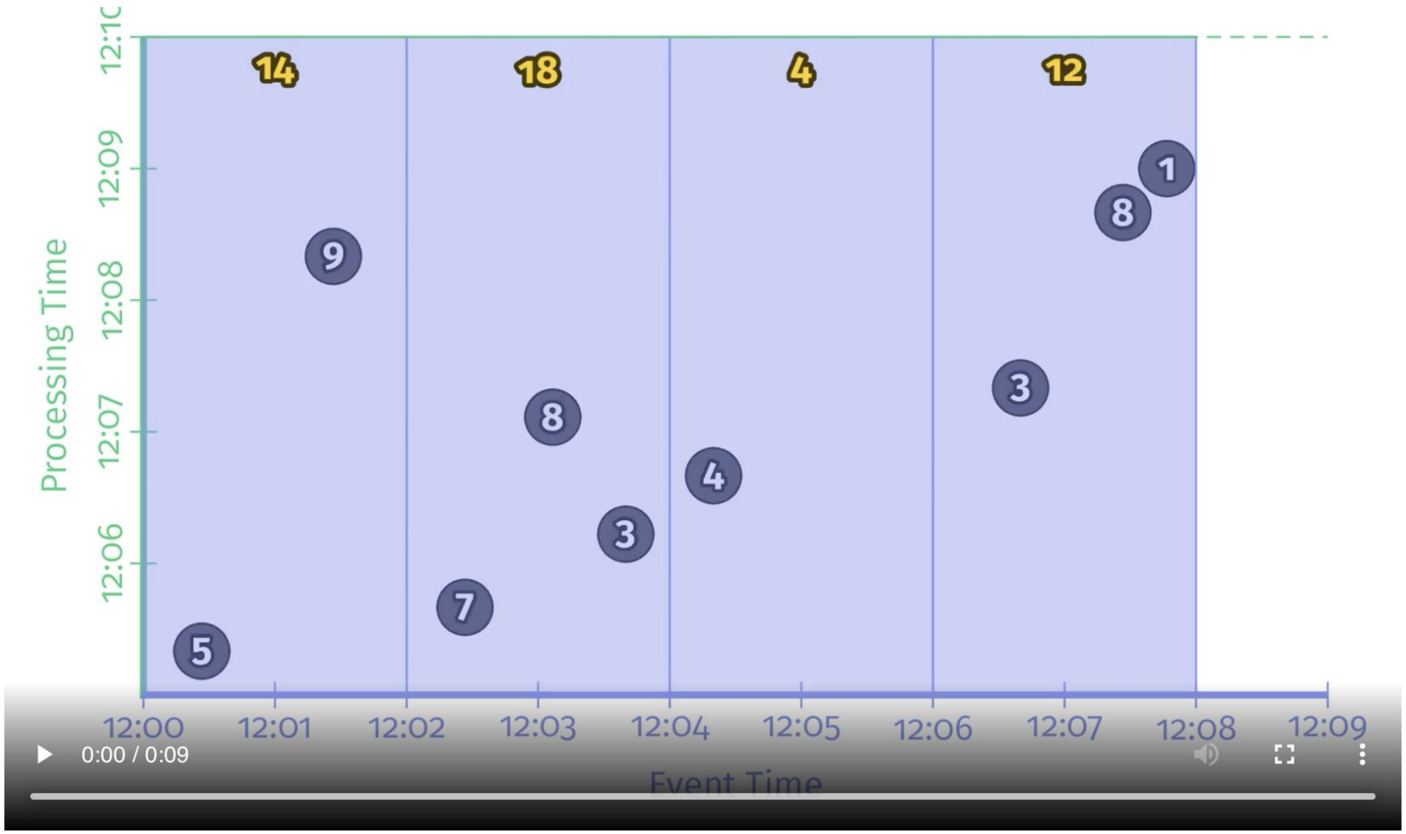


Figure 2-5. Windowed summation on a batch engine

@Marina Popova, Edward Sumitra

Triggers - Deep dive

Trigger is a policy that:

- defines the rules that dictate when it is time to process all data in the window - window is "complete"
- provide the answer to the question: “*When* in **processing** time are results materialized?”

each specific output for a window is referred to as a *pane* of the window.

There are a few types of triggers but most of them fall under the following two categories that are most frequently used:

- **repeated update triggers**
  - periodically generate updated panes for a window as its contents evolve
  - updates can be materialized under many different conditions:
    - with every new record
    - after some processing-time delay, such as once a minute
    - once a desired count is reached
    - etc.
- **completeness-based triggers**
  - updates to a window are only materialized once this window is believed to be complete
  - for batch processing - it is when all events are processed
  - for streaming data - completeness of windows has to be guessed, and is often defined using watermarks

Repeated Update Triggers

Repeated update trigger: per-record updates:
https://learning-oreilly-com.ezp-prod1.hul.harvard.edu/library/view/streaming-systems/9781491983867/assets/stsy_0206.mp4

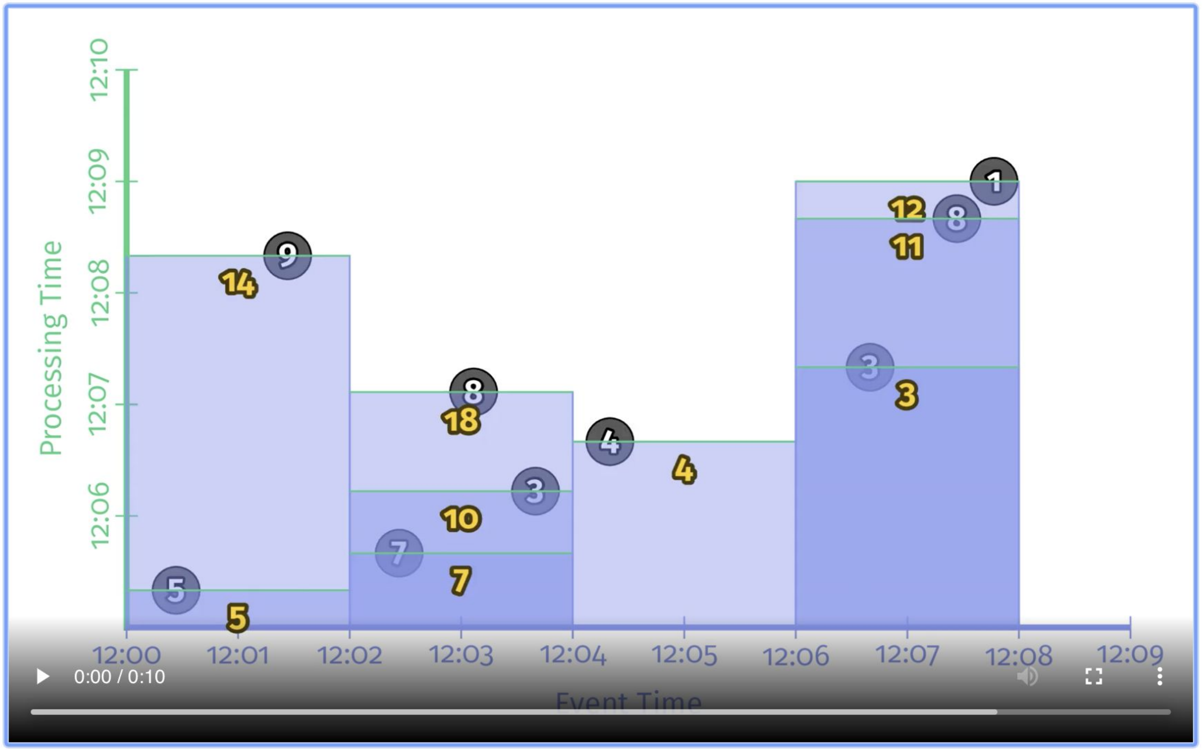


Figure 2-6. Per-record triggering on a streaming engine

Pros:
- no delay in getting the results materialized

Cons:
- very chatty

Repeated Update Triggers

Repeated update trigger: processing time delay [aligned] updates:
https://learning-oreilly-com.ezp-prod1.hul.harvard.edu/library/view/streaming-systems/9781491983867/assets/stsy_0207.mp4

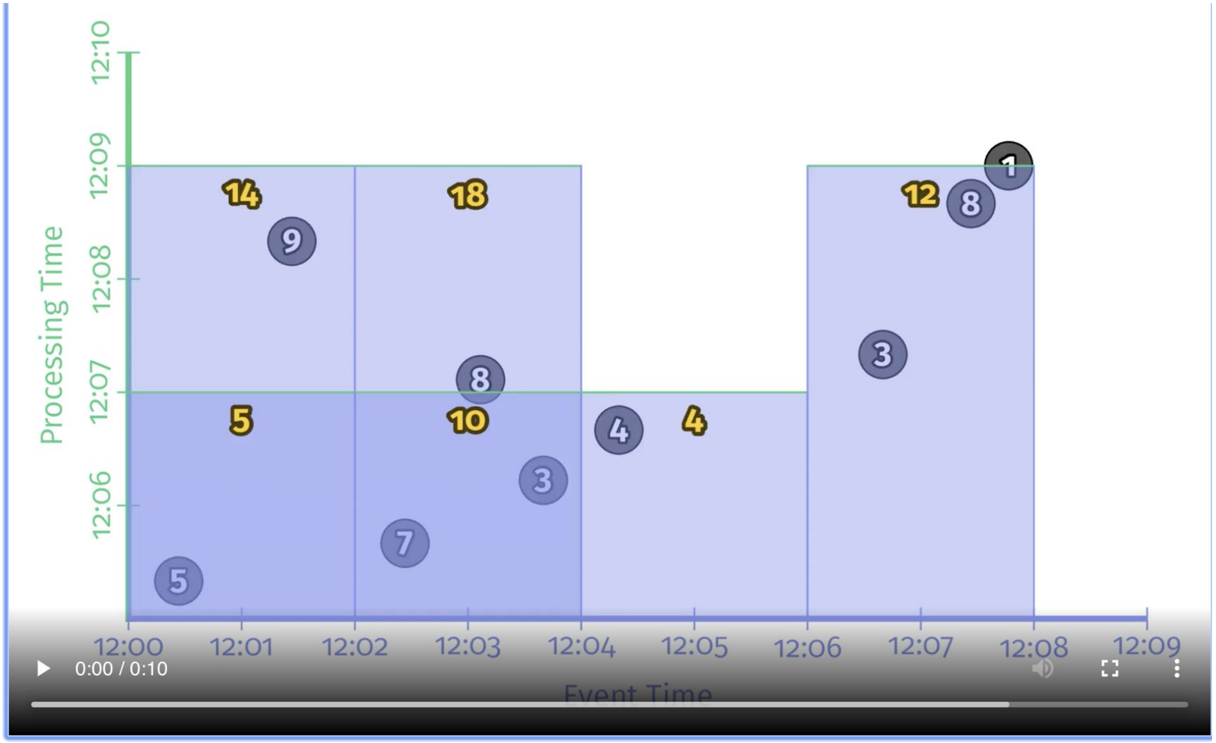


Figure 2-7. Two-minute aligned delay triggers (i.e., microbatching)

Can be of two types:

- aligned
  - delay is fixed for all keys/windows
- un-aligned
  - delay is relative to the data observed in the window

Aligned Delay Pros: predictability

Cons: bursty

Repeated Update Triggers

Repeated update trigger: processing time delay [unaligned] updates:
https://learning-oreilly-com.ezp-prod1.hul.harvard.edu/library/view/streaming-systems/9781491983867/assets/stsy_0208.mp4

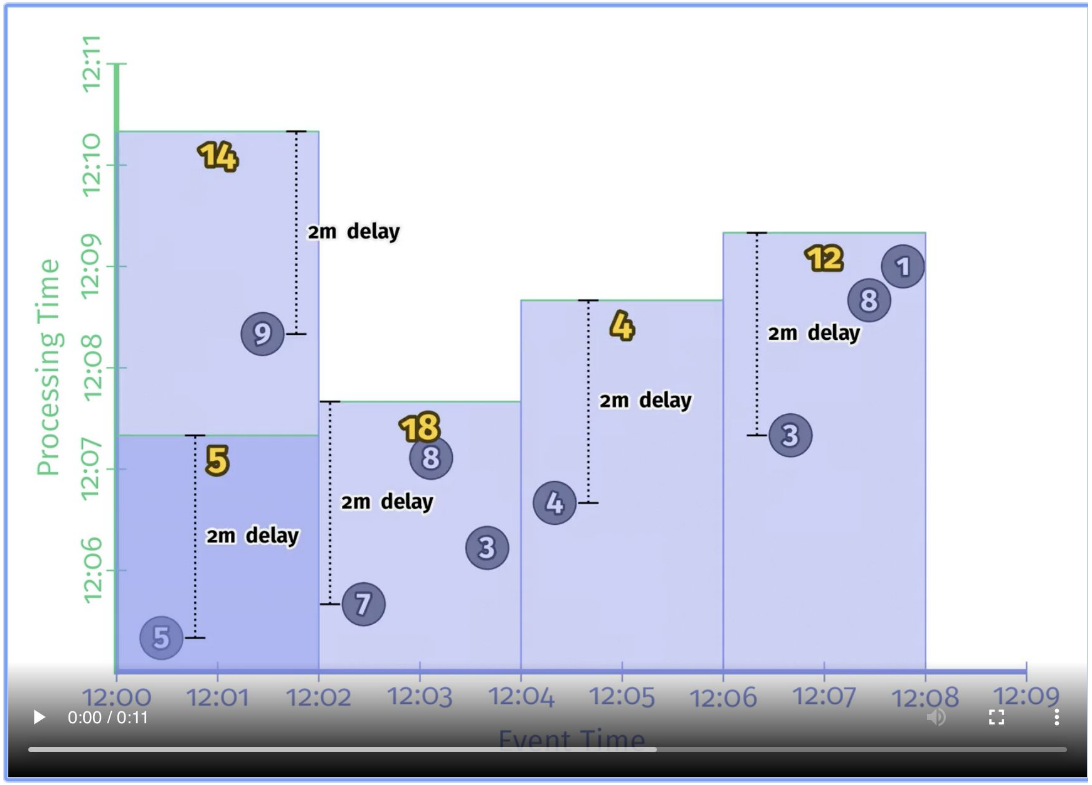


Figure 2-8. Two-minute unaligned delay triggers

Unaligned Delay Pros:

- smoothed materialization of results

Cons:

- less predictable
- can have longer "inactivity" gaps

@Marina Popova, Edward Sumitra

Completeness Triggers and Watermarks

Repeated triggers are great for periodic updates of the results OR
for use cases when event times do not matter and processing time is enough

If event time is important (almost always) - we need to know when ALL events for a specific window have arrived to be 100% sure the window is "complete"

Not really possible for most real-time systems ! :)

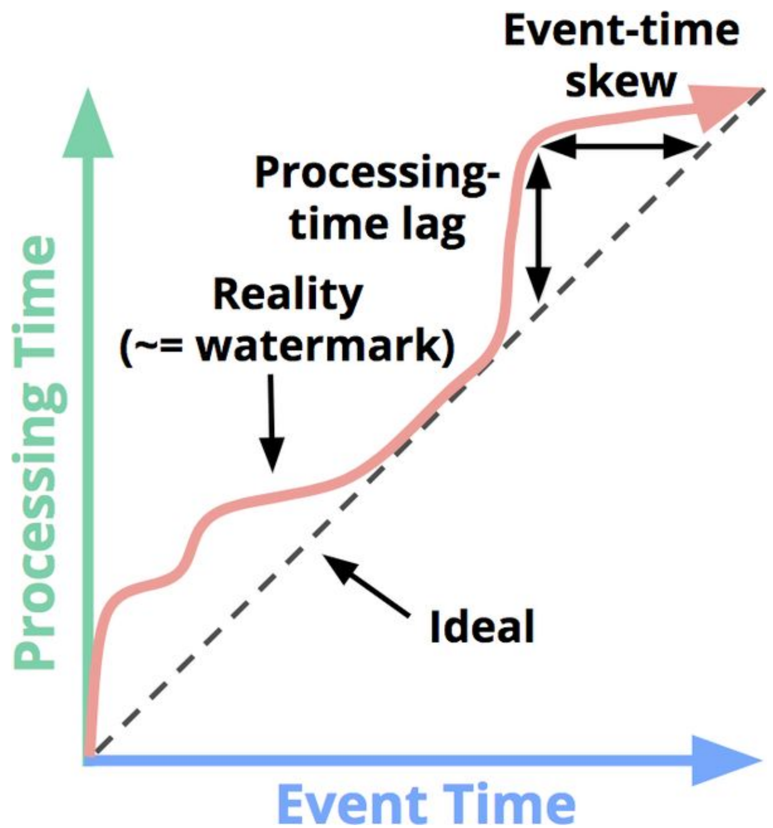
enter completeness triggers and watermarks ...

Completeness Triggers and Watermarks

Watermarks define completeness of events relative to the event-processing time

A watermark=X specifies that we assume that all events before X have been observed.

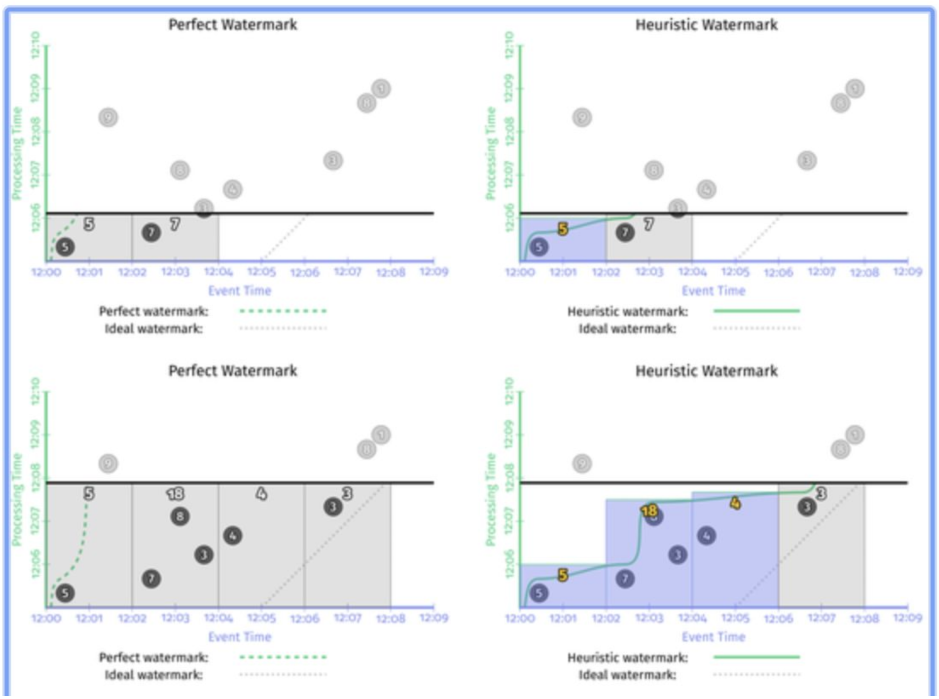
- Ideal watermark - if there is no skew
- Precise/Perfect watermark (red line)
  - we know exactly how old the events can be
- Heuristic watermarks
  - Probabilistic determination of completeness
  - Estimates based on all available information (data size, event number, etc.)



- watermarks form the foundation for the ***completeness triggers***
- windows are materialized (results are emitted) as the watermark passes the end of the window

Completeness Triggers and Watermarks

Perfect vs Heuristic watermarks:
https://learning-oreilly-com.ezp-prod1.hul.harvard.edu/library/view/streaming-systems/9781491983867/assets/stsy_0210.mp4



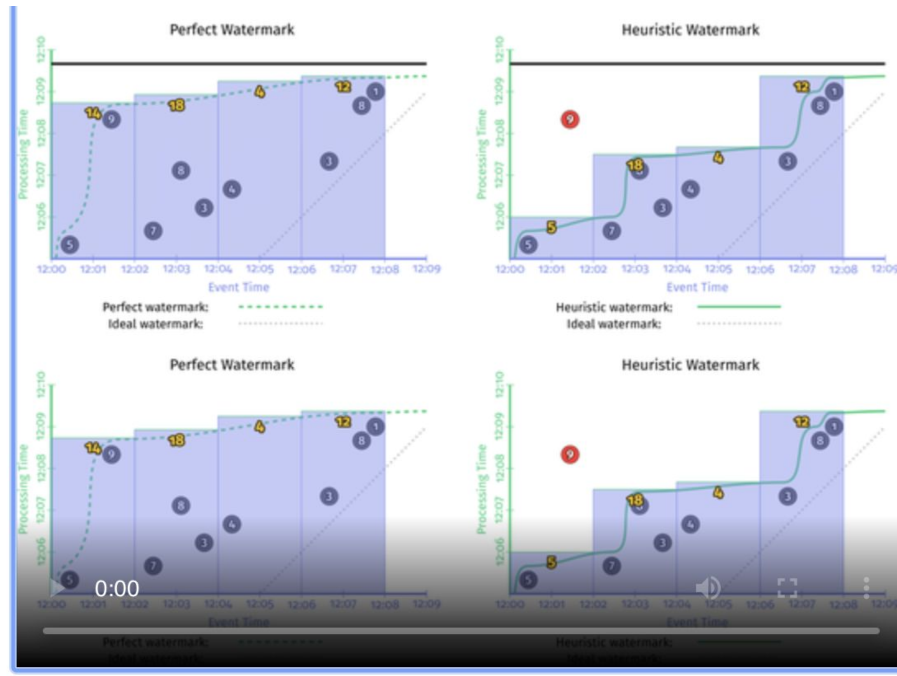


Figure 2-10. Windowed summation on a streaming engine with perfect (left) and heuristic (right) watermarks

Completeness Triggers and Watermarks

Issues with Watermarks so far:

- can be too slow:
  - if the watermark is very "generous" and waits for really old events - materialization of the results of all windows are delayed
  - see example with the Perfect watermark
- can be too fast:
  - if the watermark advances too fast, it can materialize windows before ALL old events came in
  - some old events will be missed
  - see example with the Heuristic watermark

Solution? Early/ On-Time/ Late triggers combination!

Early/On-Time/Late Triggers and Watermarks

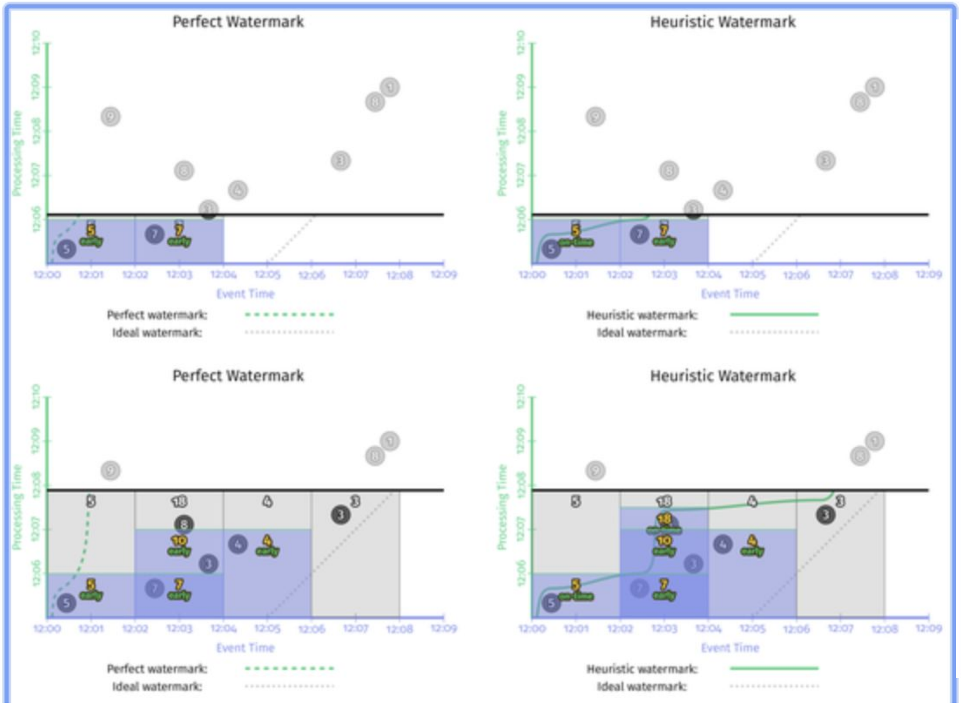
To address all the concerns , a combination of three triggers and watermarks can be used:

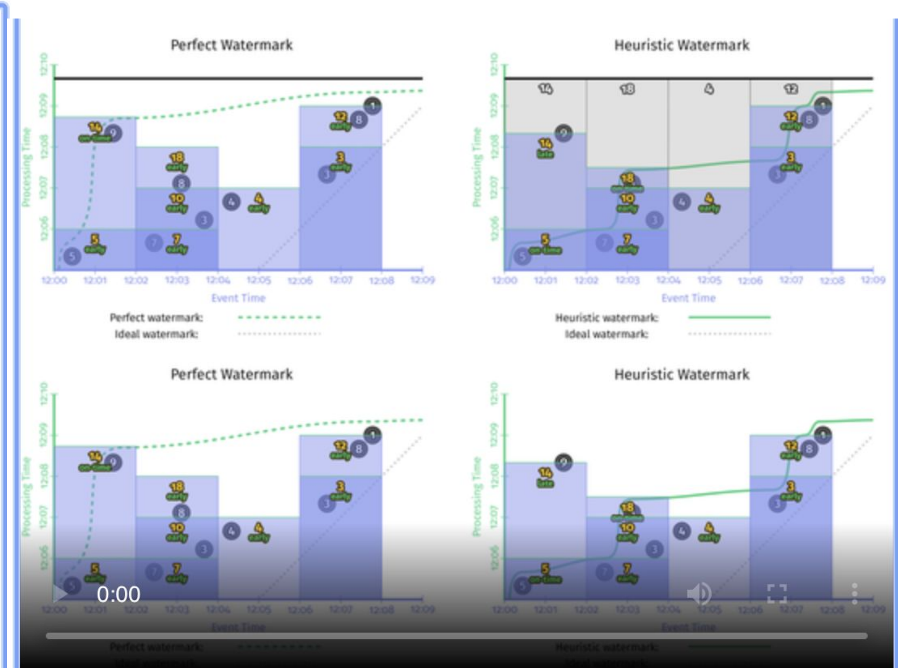
- Early triggers:
  - zero or more results of the periodic/repeated update triggers, until the watermark passes the end of the window
- On-Time trigger:
  - at most one on-time trigger when the watermark passes the end of the window
- Late Triggers:
  - zero or more results materialized when late events arrive
  - can be a repeated update trigger as well - wit a per-event or delay policy

Lets see how this plays in the example...

Early/On-Time/Late Triggers and Watermarks

https://learning-oreilly-com.ezp-prod1.hul.harvard.edu/library/view/streaming-systems/9781491983867/assets/stsy_0211.mp4





pipeline with a periodic processing-time trigger with an aligned delay of one minute for the early firings, and a per-record trigger for the late firings; windows are 2 min still

@Marina Popova, Edward Sumitra