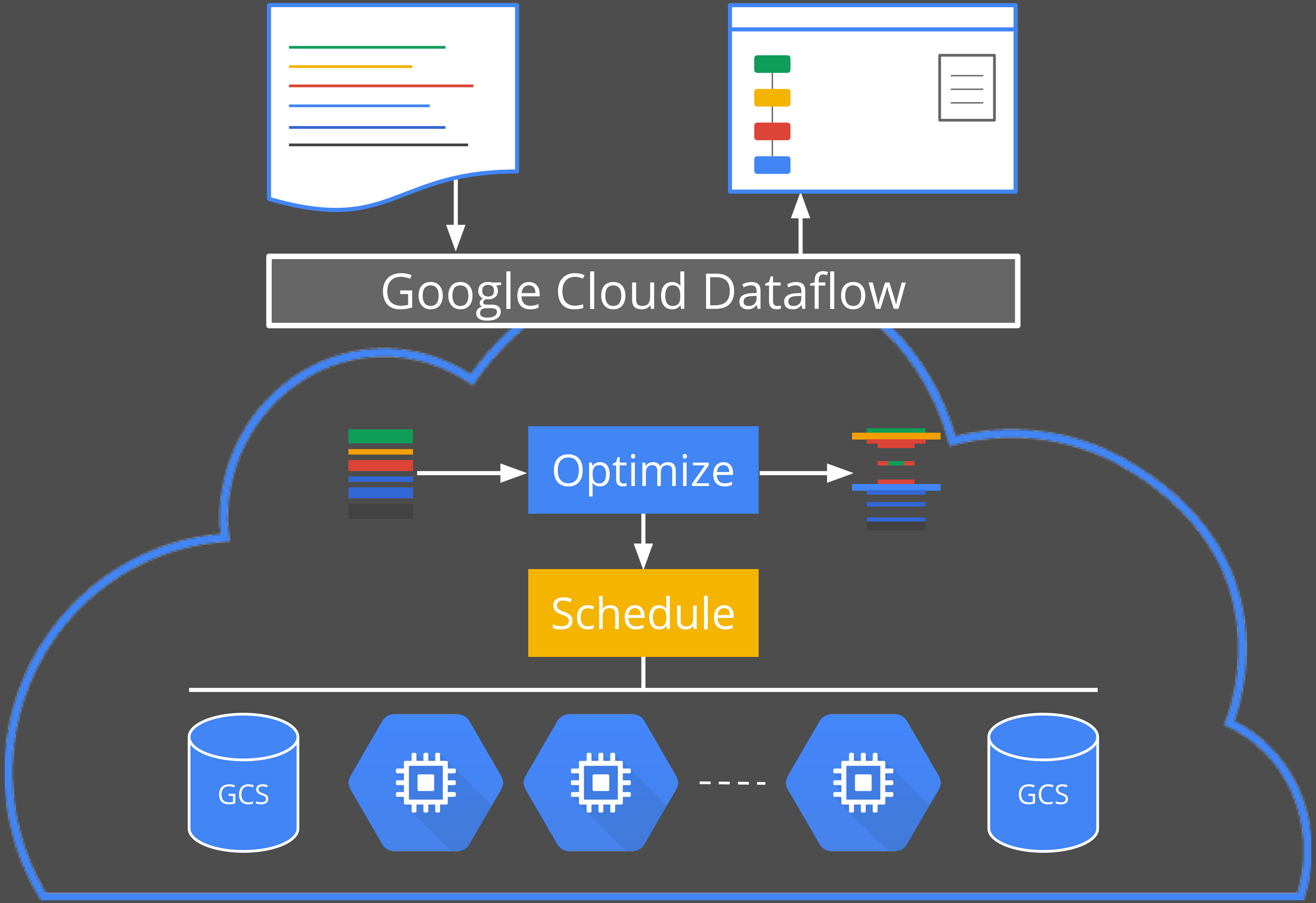
Google Docs version of slides (with animations) available at: **http://goo.gl/eX5kxa**

Have Your Cake & Eat It Too

Further Dispelling the Myths of the Lambda Architecture

Tyler Akidau
Staff Software Engineer

MillWheel	- Stream Processing System
Streaming Flume	- High-level API
Cloud Dataflow	- Data Processing Service


Google Cloud Dataflow
Optimize
Schedule
GCS
GCS

MillWheel	-	Slava Chernyak, Josh Haberman, Reuven Lax, Daniel Mills, Paul Nordstrom, Sam McVeety, Sam Whittle, and more...
Streaming Flume	-	Robert Bradshaw, Daniel Mills, and more...
Cloud Dataflow	-	Robert Bradshaw, Craig Chambers, Reuven Lax, Daniel Mills, Frances Perry, and more...

Cloud Dataflow is unreleased.

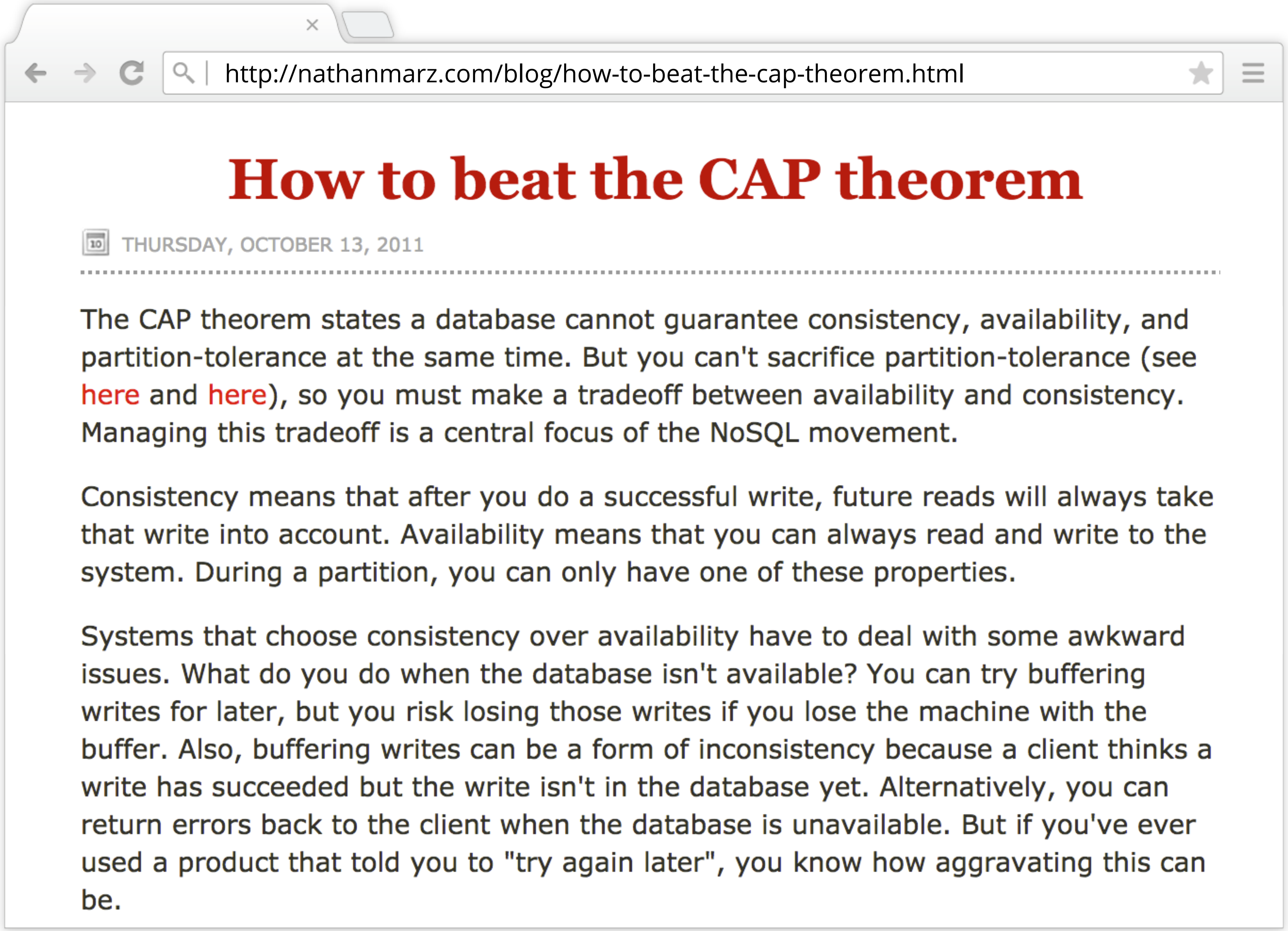
Things may change.

Agenda

1. Lambda vs Streaming
2. Strong Consistency
3. Reasoning About Time

1

Lambda vs Streaming



http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html

How to beat the CAP theorem

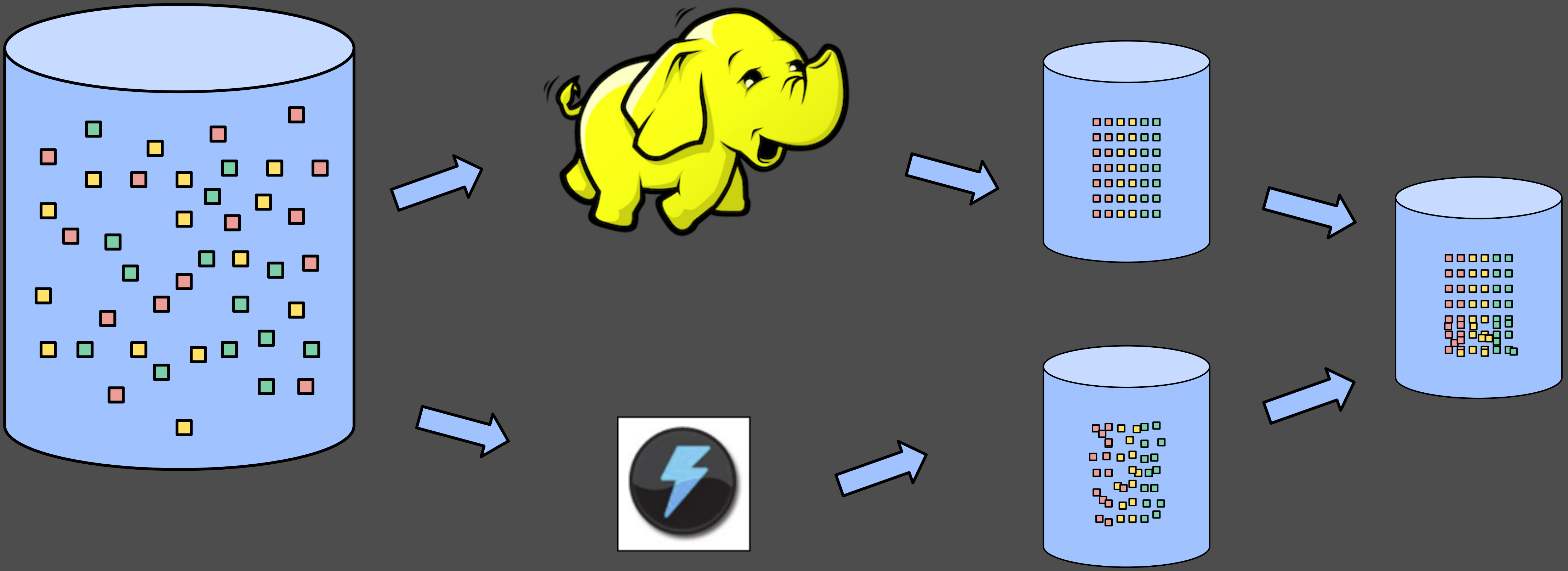
THURSDAY, OCTOBER 13, 2011

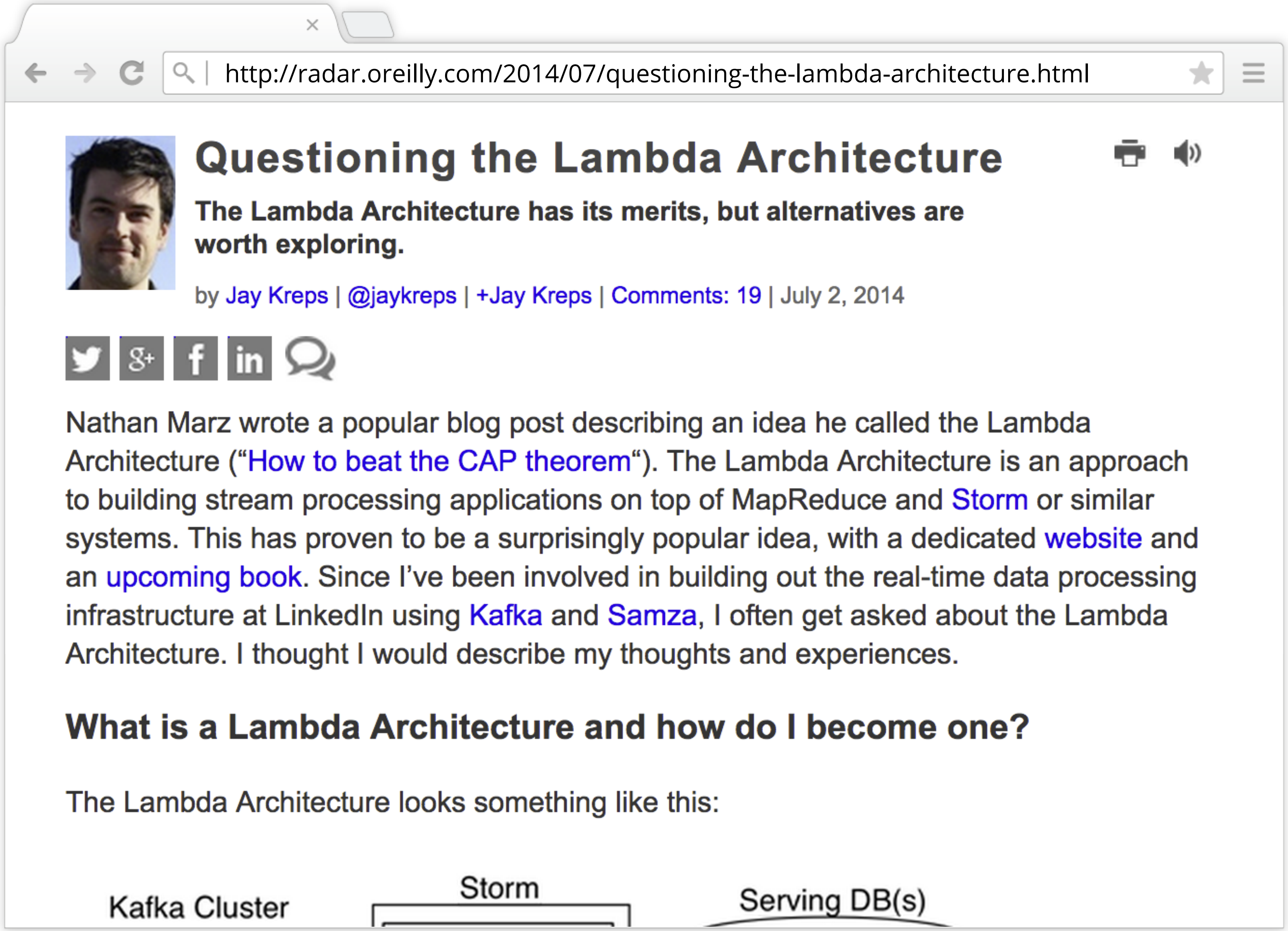
The CAP theorem states a database cannot guarantee consistency, availability, and partition-tolerance at the same time. But you can't sacrifice partition-tolerance (see here and here), so you must make a tradeoff between availability and consistency. Managing this tradeoff is a central focus of the NoSQL movement.

Consistency means that after you do a successful write, future reads will always take that write into account. Availability means that you can always read and write to the system. During a partition, you can only have one of these properties.

Systems that choose consistency over availability have to deal with some awkward issues. What do you do when the database isn't available? You can try buffering writes for later, but you risk losing those writes if you lose the machine with the buffer. Also, buffering writes can be a form of inconsistency because a client thinks a write has succeeded but the write isn't in the database yet. Alternatively, you can return errors back to the client when the database is unavailable. But if you've ever used a product that told you to "try again later", you know how aggravating this can be.

The Lambda Architecture





http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html

Questioning the Lambda Architecture

The Lambda Architecture has its merits, but alternatives are worth exploring.

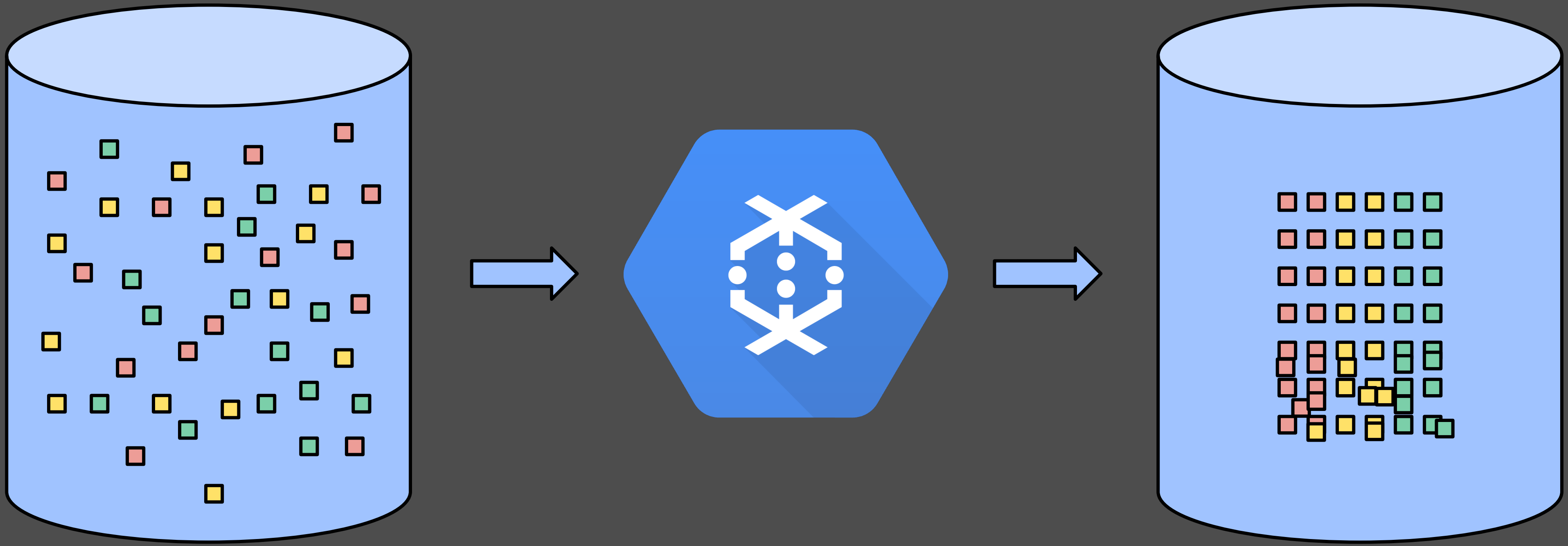
by Jay Kreps | @jaykreps | +Jay Kreps | Comments: 19 | July 2, 2014

Nathan Marz wrote a popular blog post describing an idea he called the Lambda Architecture ("How to beat the CAP theorem"). The Lambda Architecture is an approach to building stream processing applications on top of MapReduce and Storm or similar systems. This has proven to be a surprisingly popular idea, with a dedicated website and an upcoming book. Since I've been involved in building out the real-time data processing infrastructure at LinkedIn using Kafka and Samza, I often get asked about the Lambda Architecture. I thought I would describe my thoughts and experiences.

What is a Lambda Architecture and how do I become one?

The Lambda Architecture looks something like this:

The Evolution of Streaming



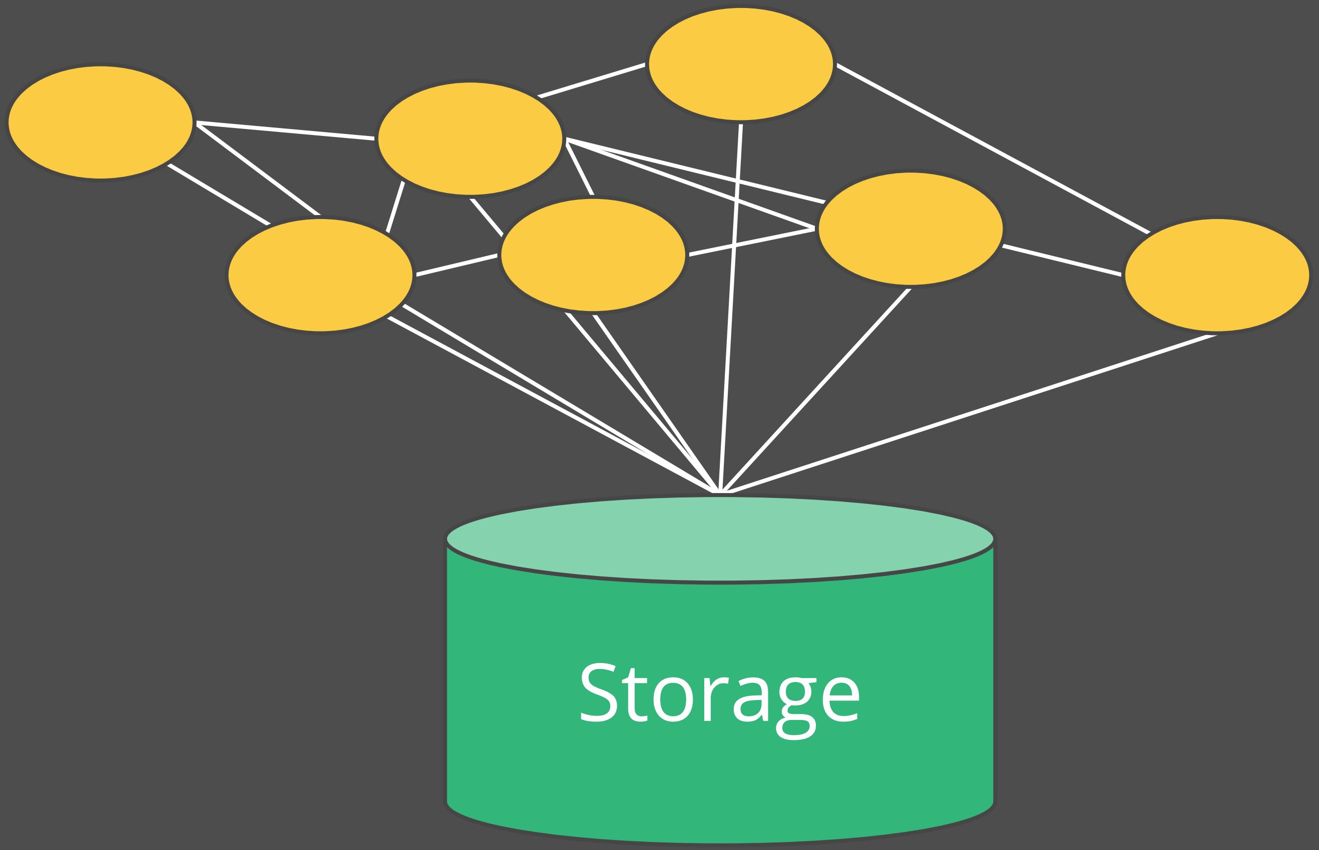
What does it take?

Strong Consistency

Tools for Reasoning About Time

2 Strong Consistency

Consistent Storage

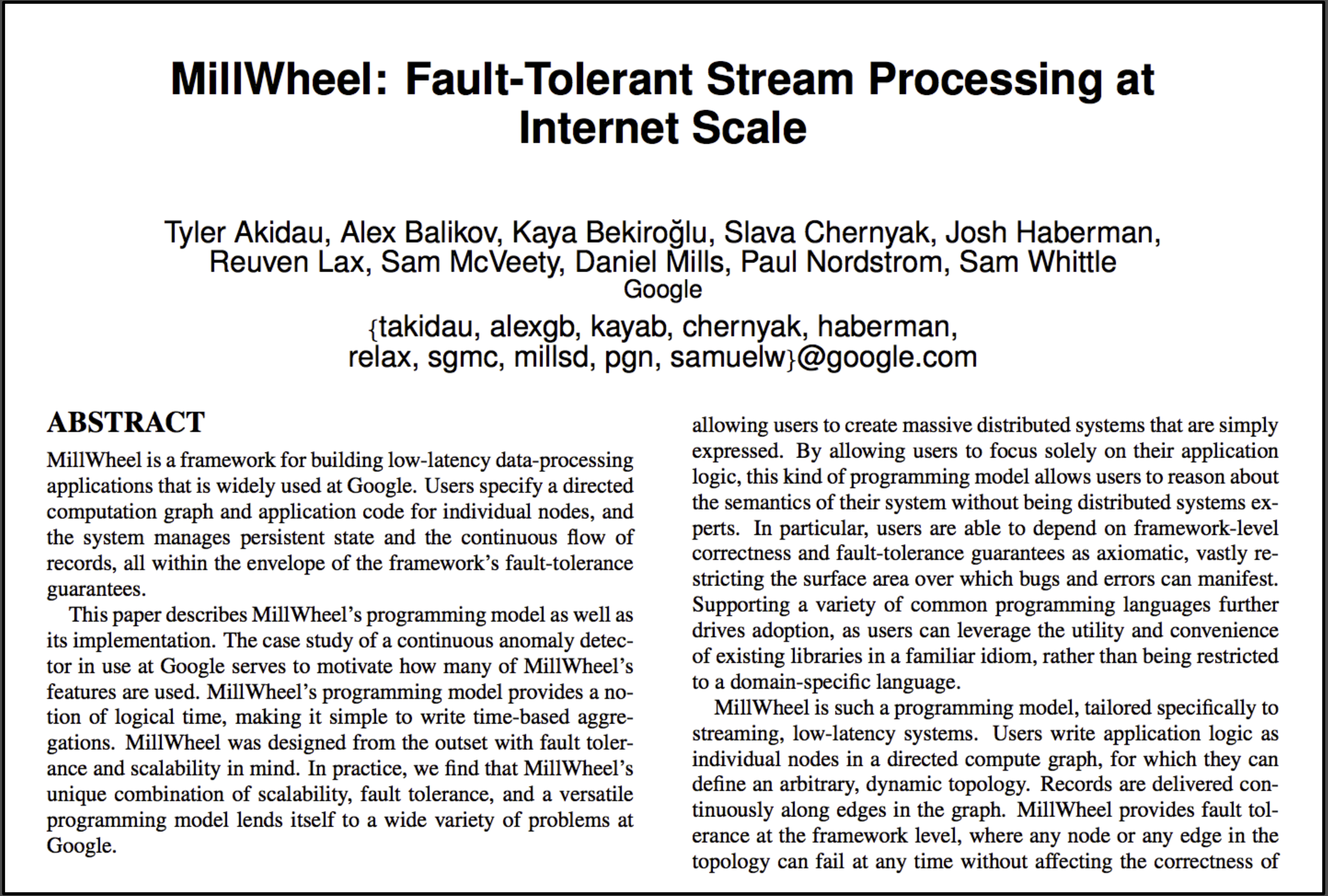


Why consistency is important

- Mostly correct is not good enough
- Required for exactly-once processing
- Required for repeatable results
- Cannot replace batch without it

How?

- Sequencers (e.g. BigTable)
- Leases (e.g. Spanner)
- Federation of storage silos (e.g. Samza, Dataflow)
- RDDs (e.g. Spark)



MillWheel: Fault-Tolerant Stream Processing at Internet Scale

Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, Sam Whittle
Google
{takidau, alexgb, kayab, chernyak, haberman, relax, sgmc, millsd, pgn, samuelw}@google.com

ABSTRACT

MillWheel is a framework for building low-latency data-processing applications that is widely used at Google. Users specify a directed computation graph and application code for individual nodes, and the system manages persistent state and the continuous flow of records, all within the envelope of the framework's fault-tolerance guarantees.

This paper describes MillWheel's programming model as well as its implementation. The case study of a continuous anomaly detector in use at Google serves to motivate how many of MillWheel's features are used. MillWheel's programming model provides a notion of logical time, making it simple to write time-based aggregations. MillWheel was designed from the outset with fault tolerance and scalability in mind. In practice, we find that MillWheel's unique combination of scalability, fault tolerance, and a versatile programming model lends itself to a wide variety of problems at Google.

allowing users to create massive distributed systems that are simply expressed. By allowing users to focus solely on their application logic, this kind of programming model allows users to reason about the semantics of their system without being distributed systems experts. In particular, users are able to depend on framework-level correctness and fault-tolerance guarantees as axiomatic, vastly restricting the surface area over which bugs and errors can manifest. Supporting a variety of common programming languages further drives adoption, as users can leverage the utility and convenience of existing libraries in a familiar idiom, rather than being restricted to a domain-specific language.

MillWheel is such a programming model, tailored specifically to streaming, low-latency systems. Users write application logic as individual nodes in a directed compute graph, for which they can define an arbitrary, dynamic topology. Records are delivered continuously along edges in the graph. MillWheel provides fault tolerance at the framework level, where any node or any edge in the topology can fail at any time without affecting the correctness of

http://research.google.com/pubs/pub41378.html

3 Reasoning About Time

Event Time vs Stream Time

Batch vs Streaming

Approaches

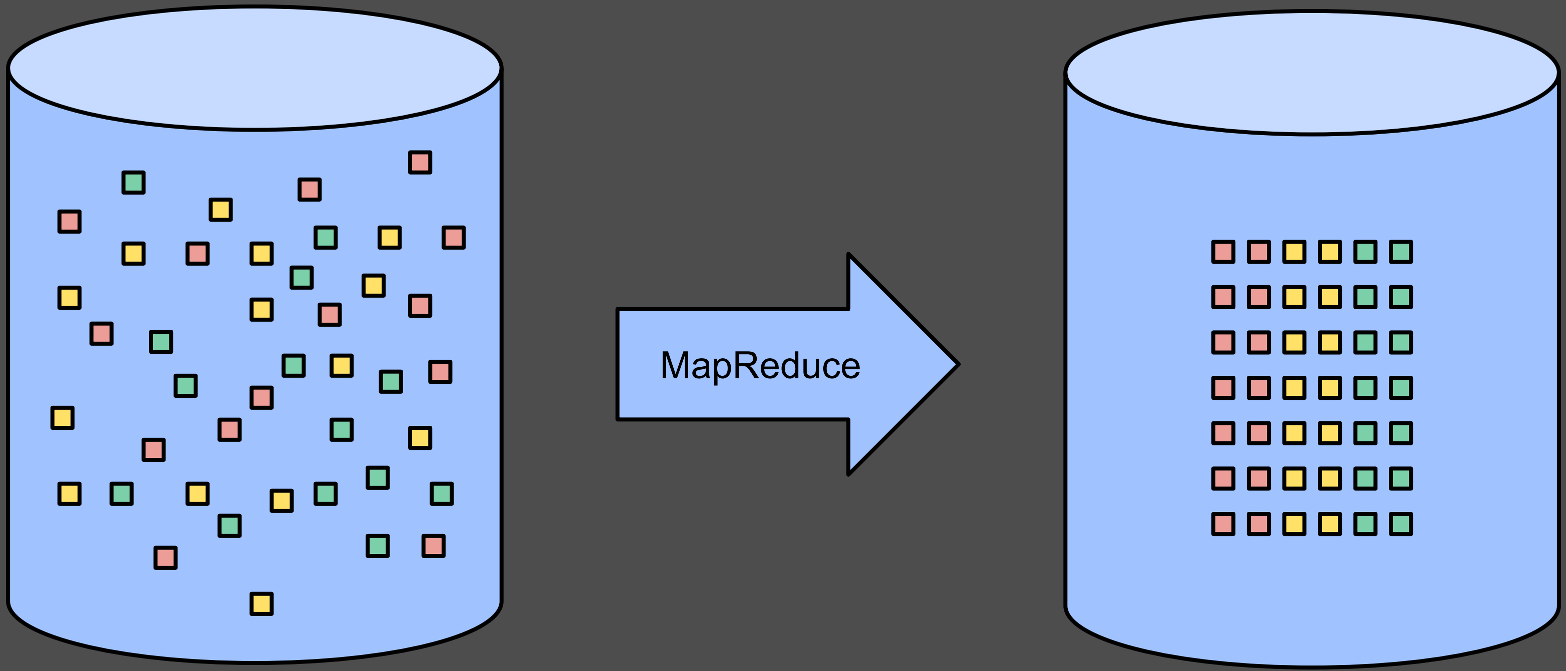
Dataflow API

Event Time - When Events Happened

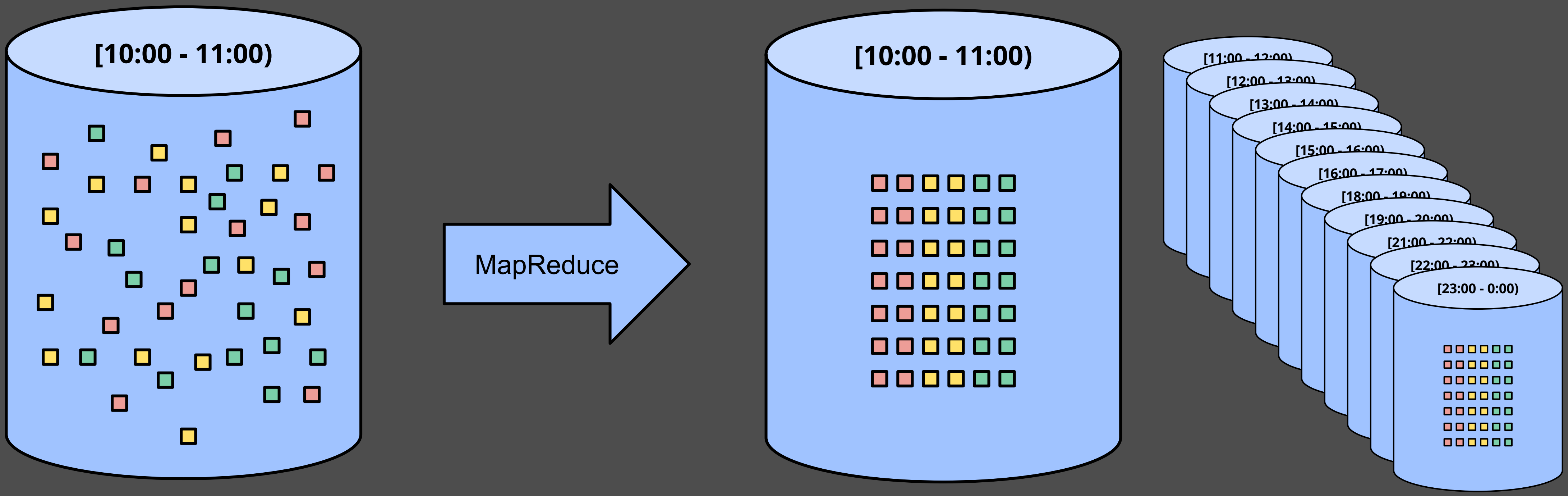
Stream Time - When Events Are Processed

Batch vs Streaming

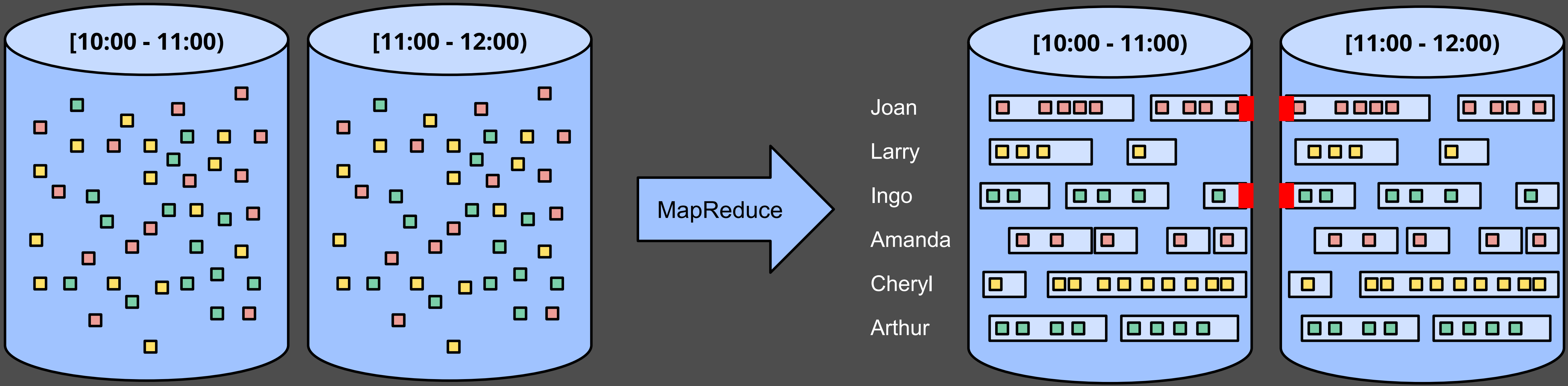
Batch



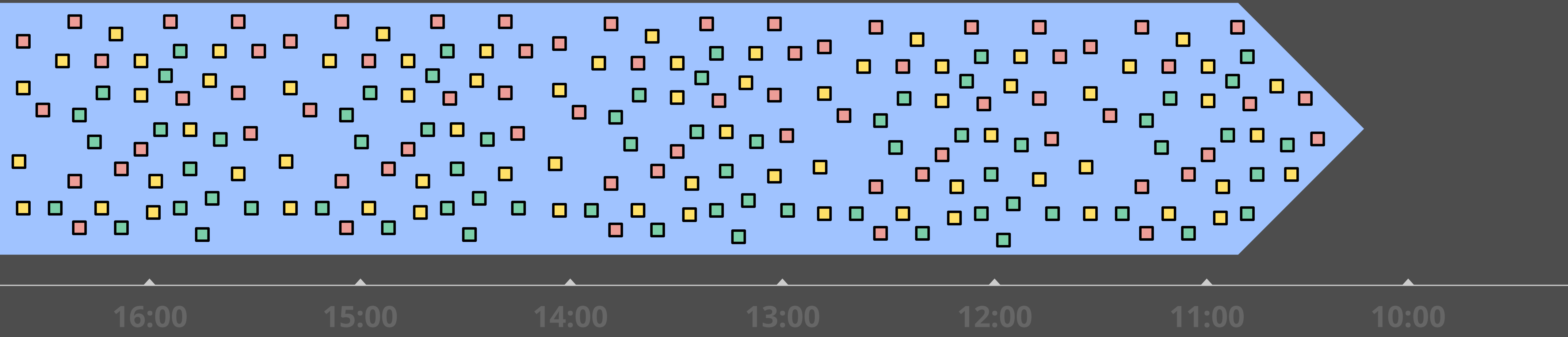
Batch: Fixed Windows



Batch: User Sessions



Streaming



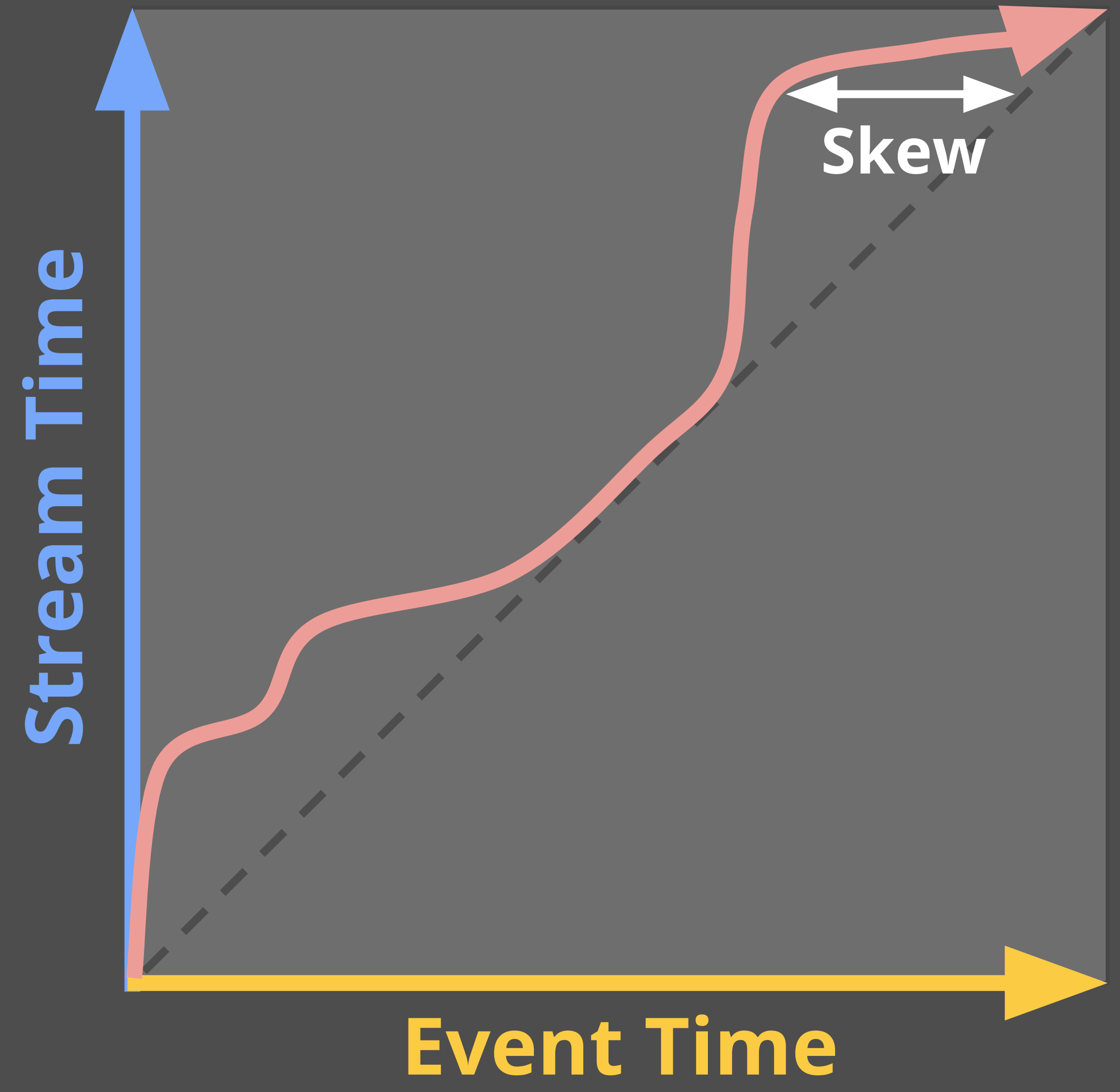
Confounding characteristics of data streams

Unordered

Unbounded

Of Varying Event Time Skew

Event Time Skew

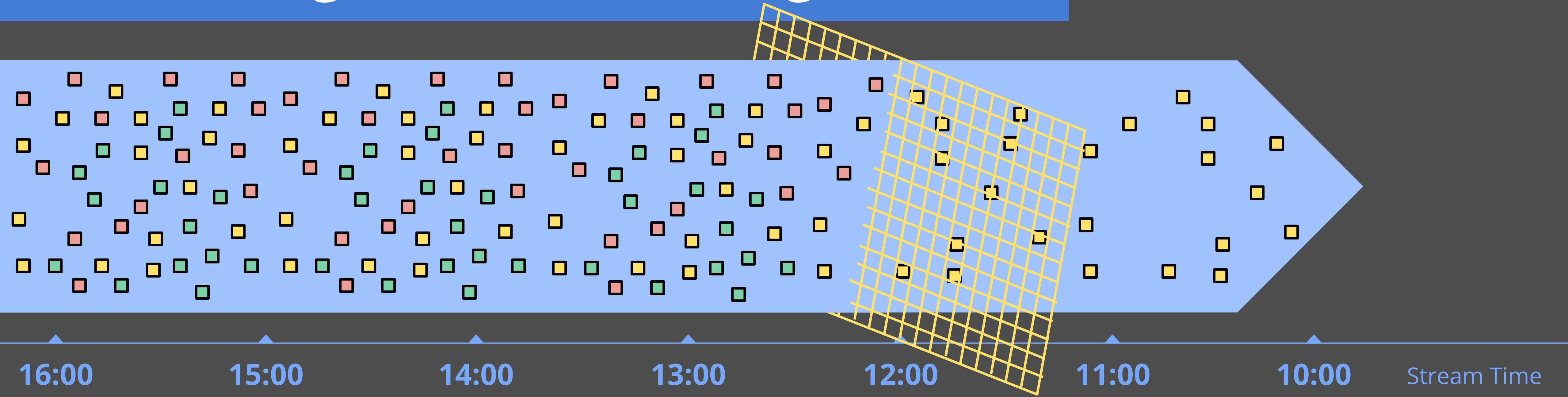


Approaches

Approaches to reasoning about time

1. Time-Agnostic Processing
2. Approximation
3. Stream Time Windowing
4. Event Time Windowing

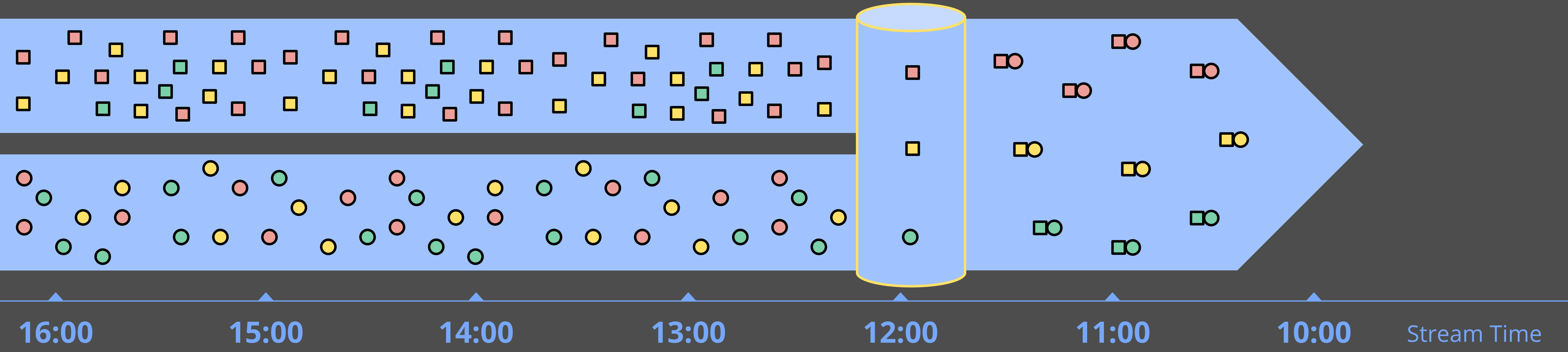
1. Time-Agnostic Processing - Filters



Example Input: Web server traffic logs
Example Output: All traffic from specific domains

Pros: Straightforward
Efficient
Cons: Limited utility

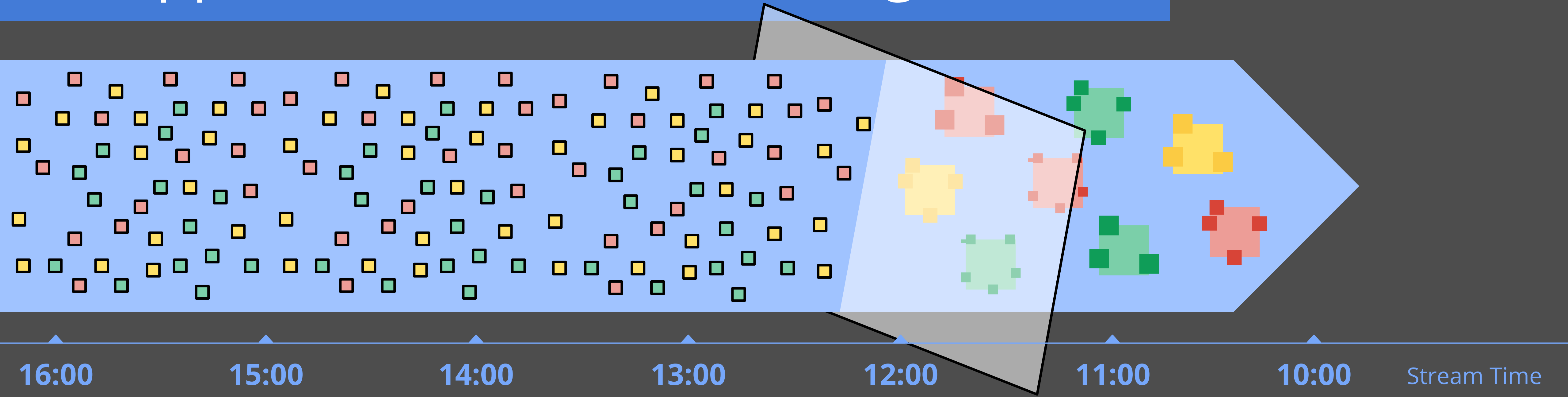
1. Time-Agnostic Processing - Hash Join



Example Input: Query & Click traffic
Example Output: Joined stream of Query + Click pairs

Pros: Straightforward
Efficient
Cons: Limited utility

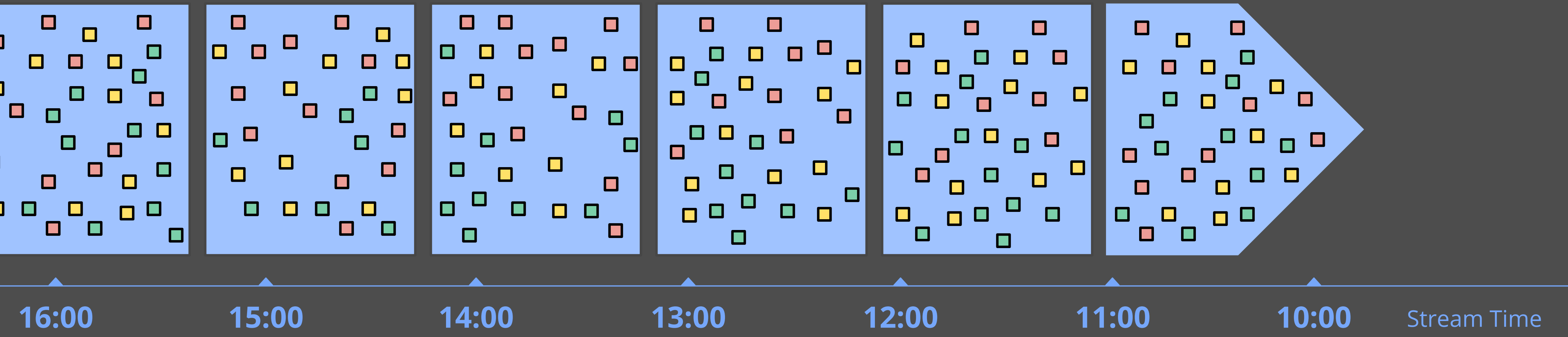
2. Approximation via Online Algorithms



Example Input: Twitter hashtags
Example Output: Approximate top N hashtags per prefix

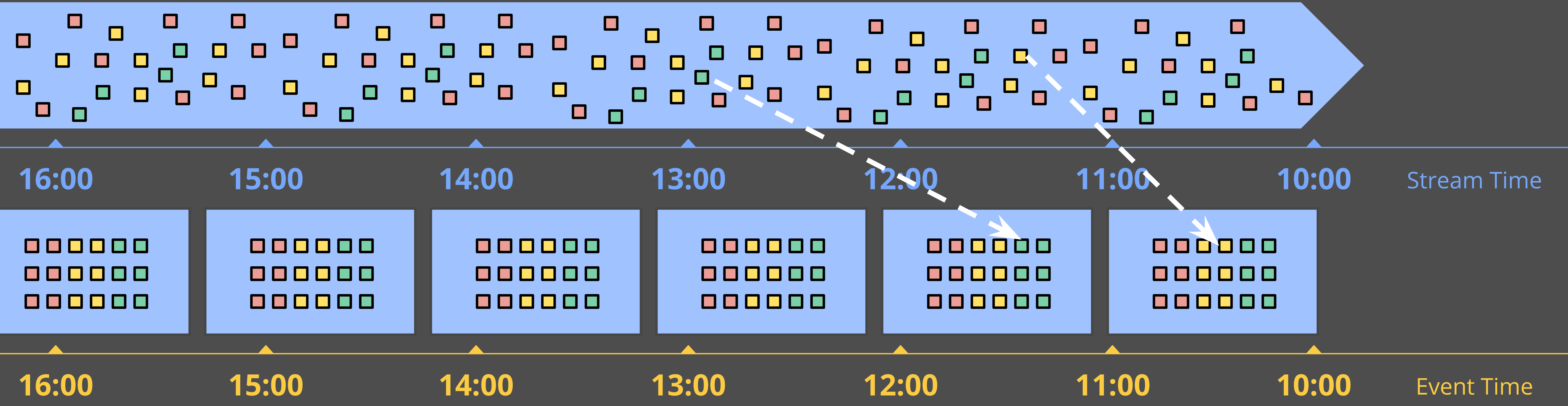
Pros: Efficient
Cons: Inexact
Complicated Algorithms

3. Windowing by Stream Time



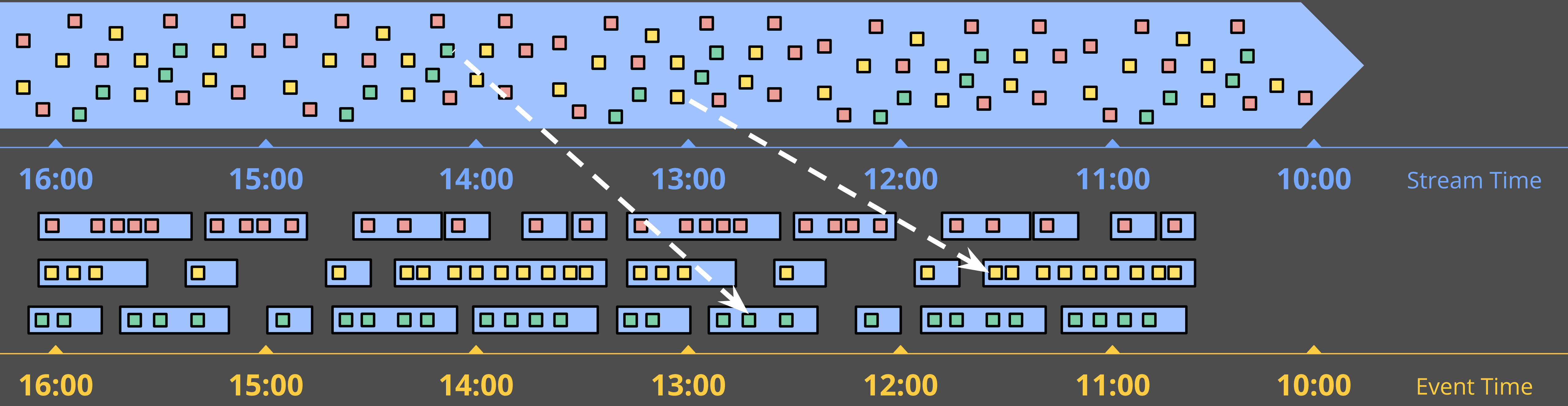
Example Input:	Web server request traffic
Example Output:	Per-minute rate of received requests
Pros:	Straightforward
	Results reflect contents of stream
Cons:	Results don’t reflect events as they happened
	If approximating event time, usefulness varies

4. Windowing by Event Time - Fixed Windows



Example Input:	Twitter hashtags
Example Output:	Top N hashtags by prefix per hour.
Pros:	Reflects events as they occurred
Cons:	More complicated buffering
	Completeness issues

4. Windowing by Event Time - Sessions



Example Input:	User activity stream
Example Output:	Per-session group of activities
Pros:	Reflects events as they occurred
Cons:	More complicated buffering
	Completeness issues

Dataflow API

What are you computing?

Where in event time?

When in stream time?

What = Aggregation API

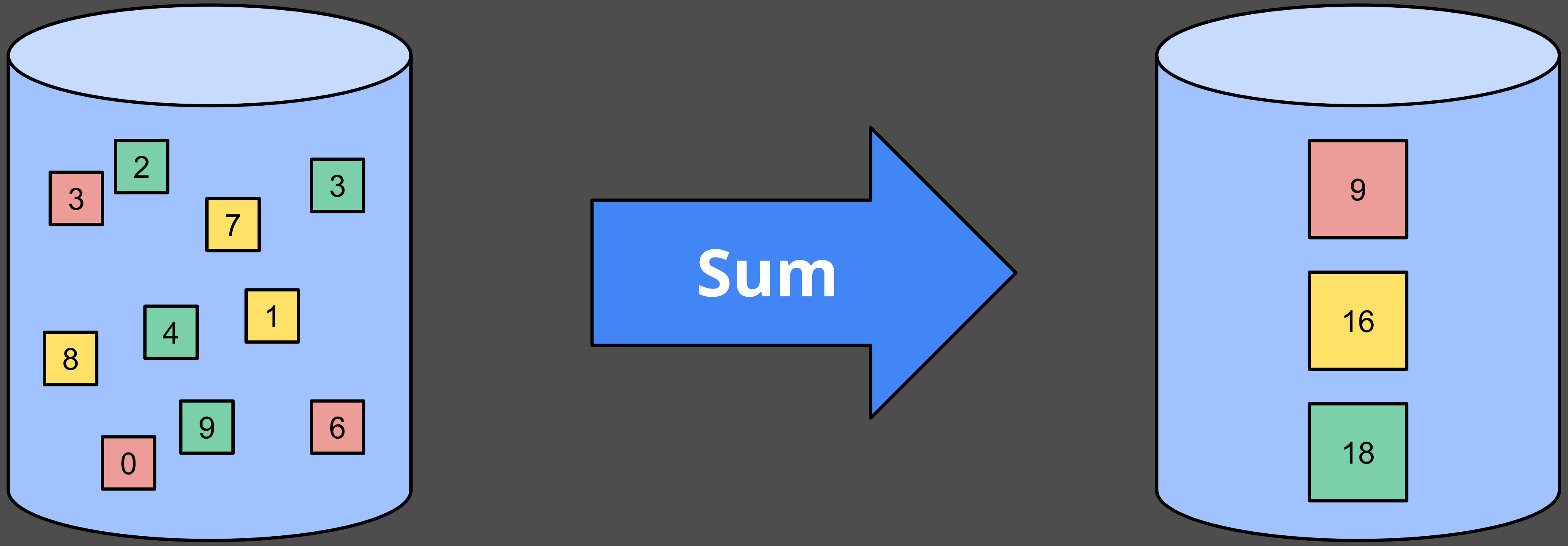
Where = Windowing API

When = Watermarks + Triggers API

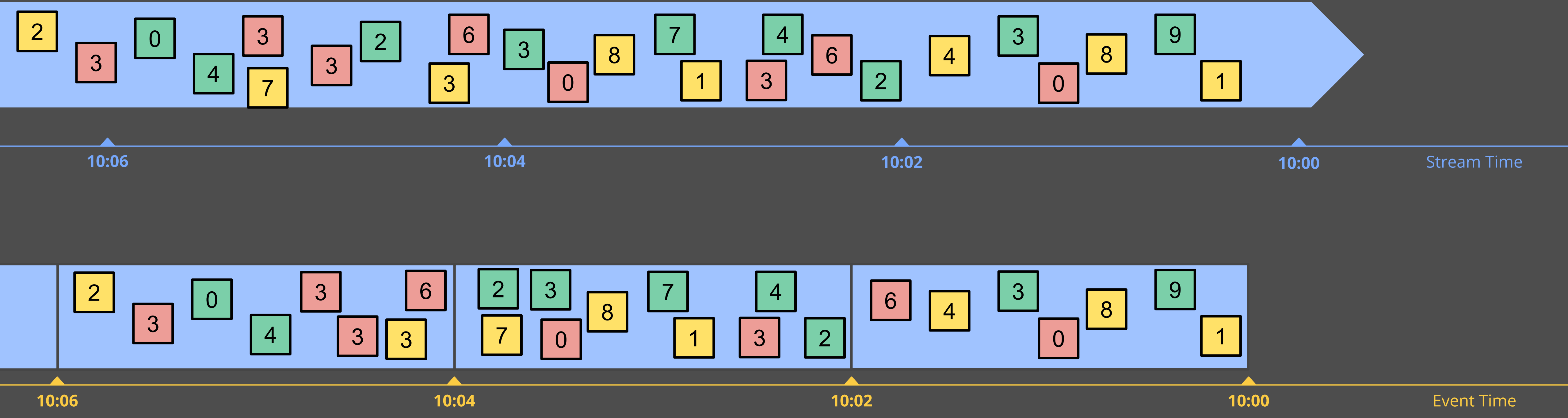
Aggregation API

```
PCollection<KV<String, Double>> sums = Pipeline
   .begin()
   .read(“userRequests”)
   .apply(new Sum());
```

Aggregation API



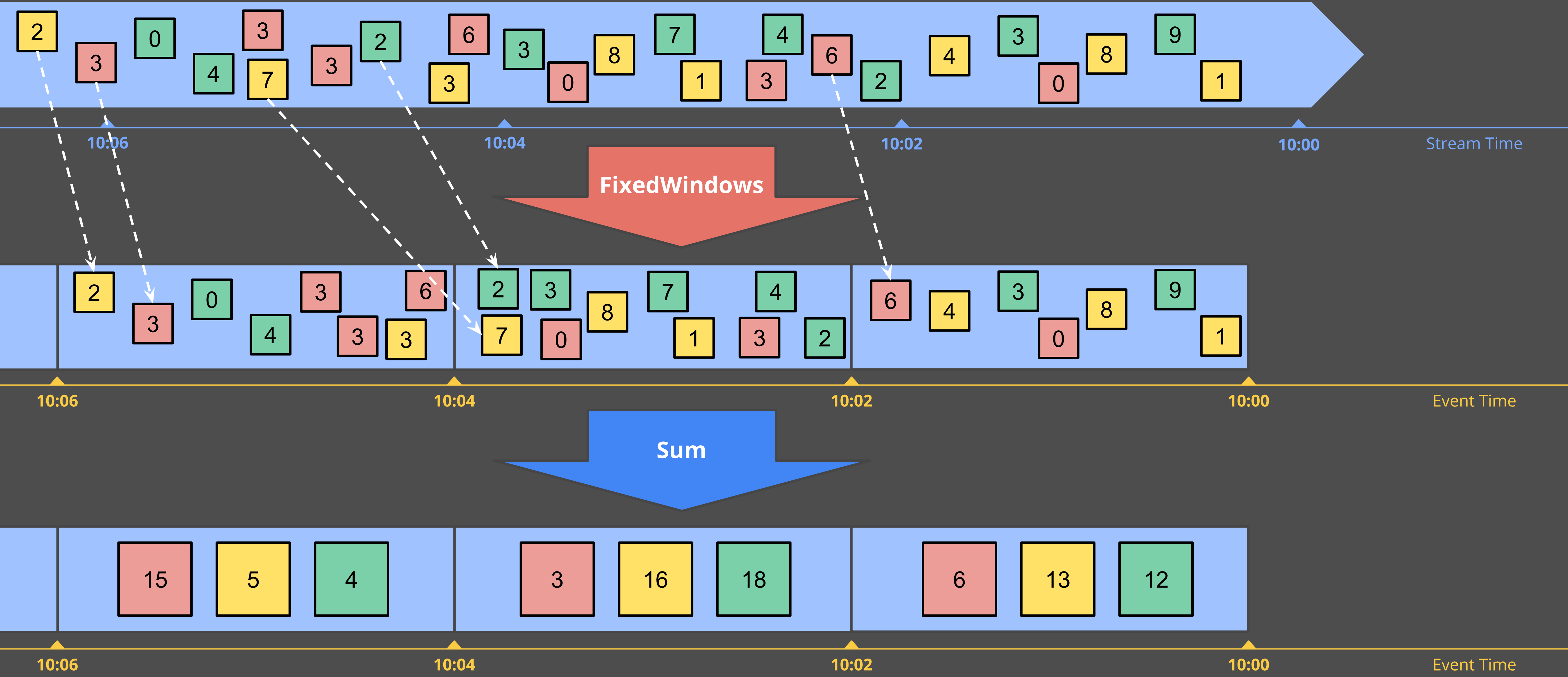
Streaming Mode



Windowing API

```
PCollection<KV<String, Long>> sums = Pipeline
   .begin()
   .read(“userRequests”)
   .apply(Window.into(new FixedWindows(2, MINUTE)));
   .apply(new Sum());
```

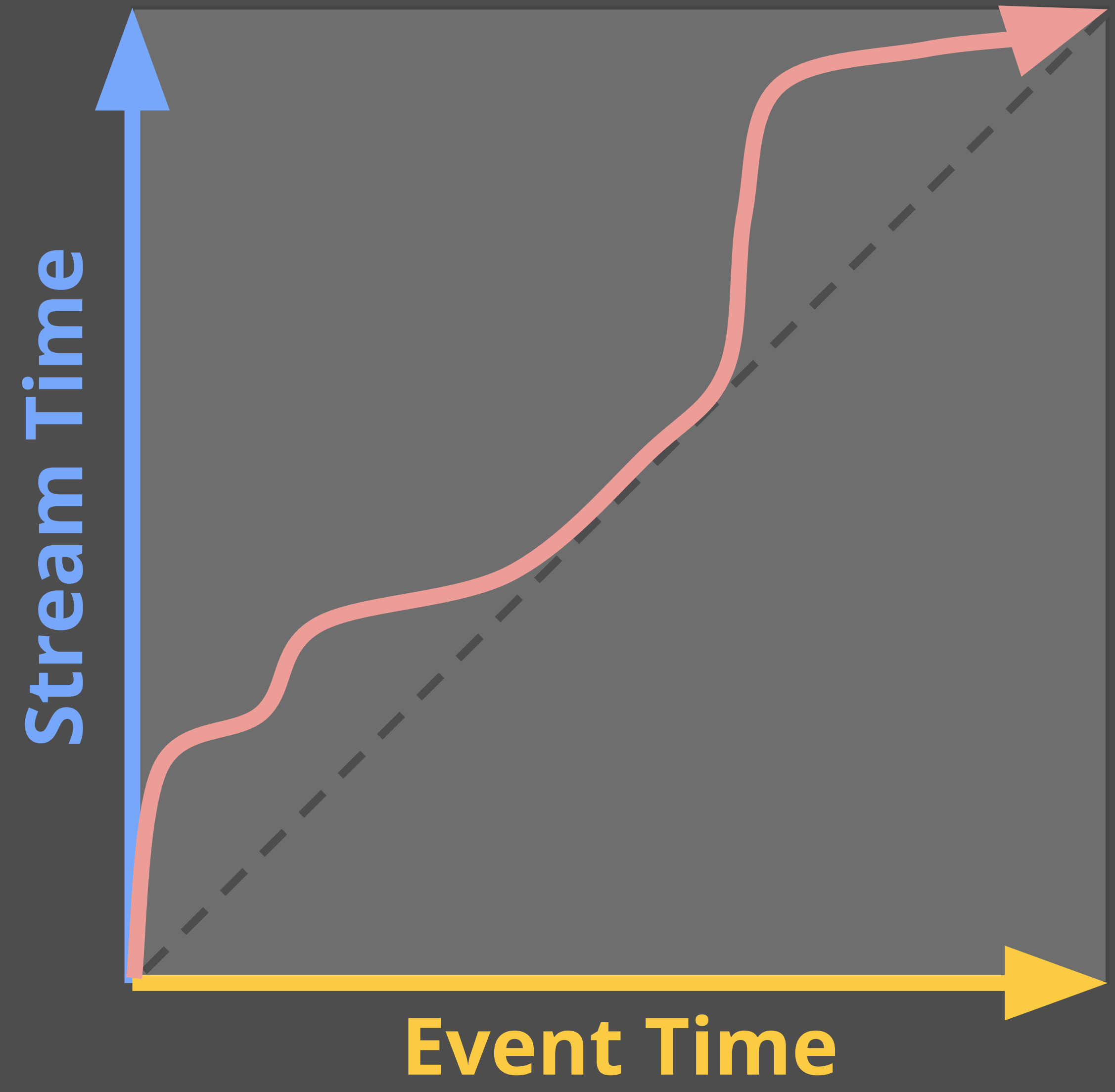
Windowing API



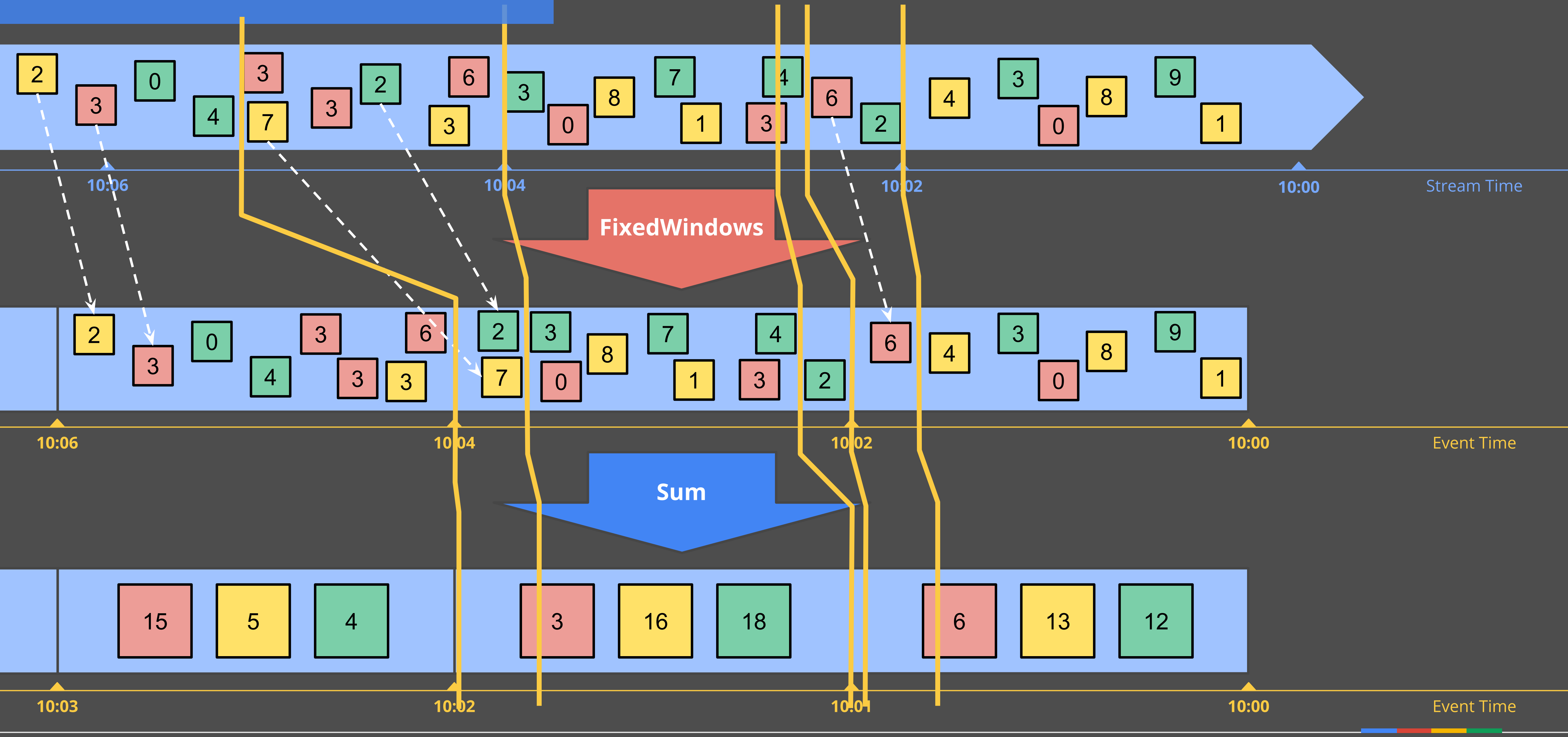
Watermarks

- f(S) -> E
- S = a point in stream time (i.e. now)
- E = the point in event time up to which input data is complete as of S

Event Time Skew



Watermarks


2
3
0
4
3
7
3
2
6
3
3
0
8
7
1
4
3
6
2
4
3
0
8
9
1
10:06
10:04
10:02
10:00
Stream Time
FixedWindows
2
3
0
4
3
3
6
3
2
7
3
0
8
7
1
4
3
2
6
4
3
0
8
9
1
10:06
10:04
10:02
10:00
Event Time
Sum
15
5
4
3
16
18
6
13
12
10:03
10:02
10:01
10:00
Event Time

Watermark Caveats

Too slow = more latency

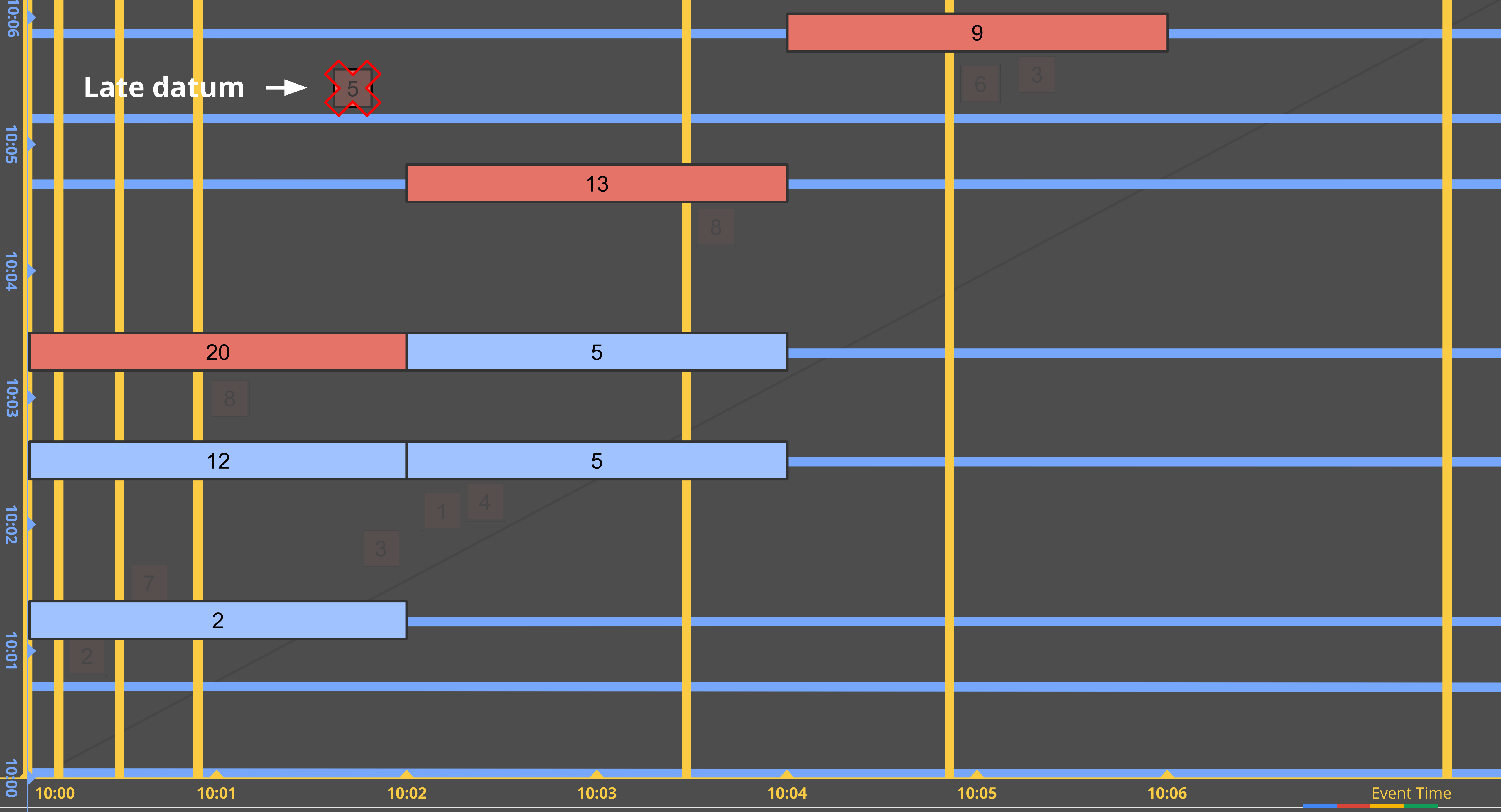
Too fast = late data

Triggers

When in stream time to emit?

Triggers API

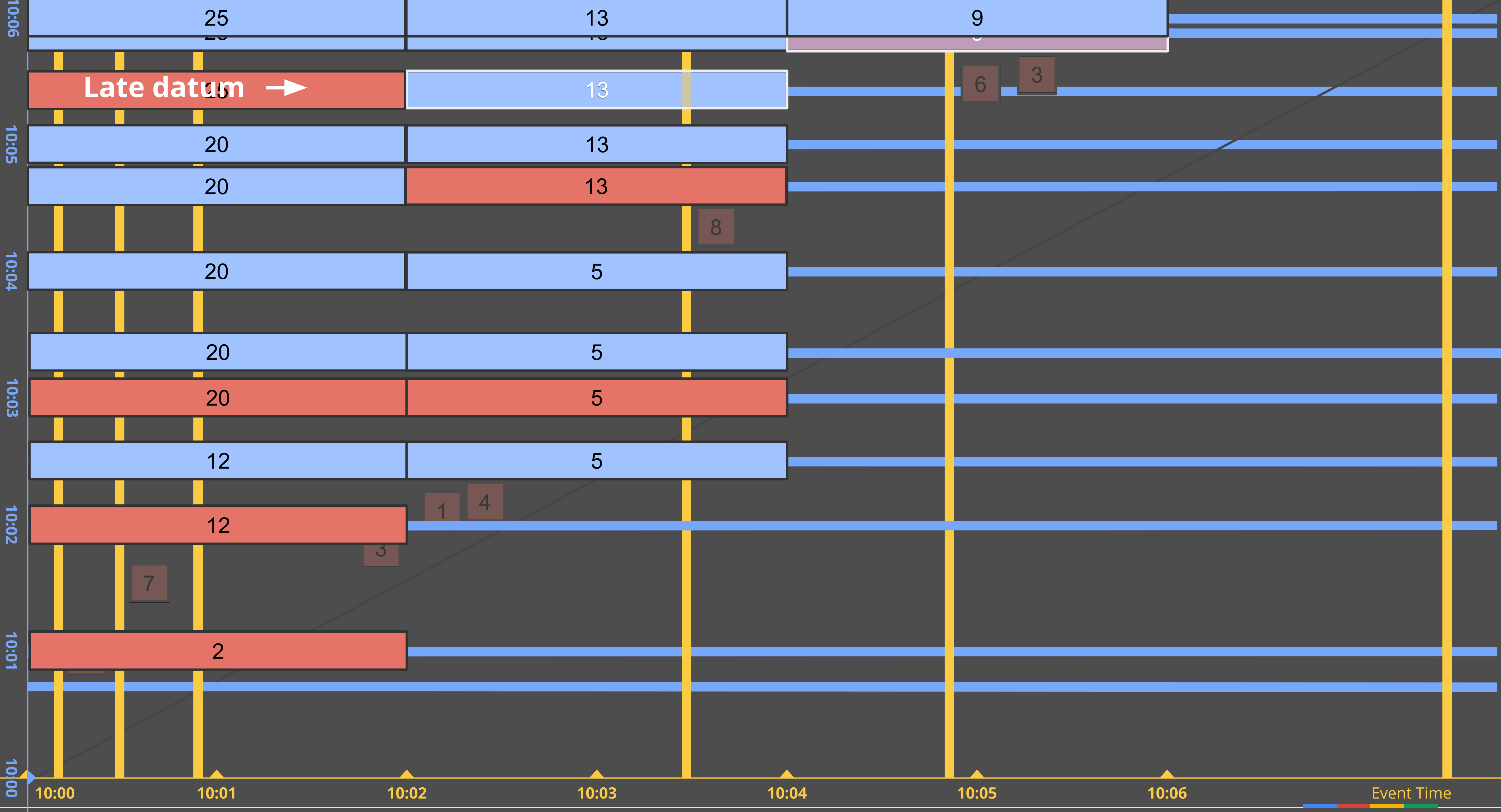
```
PCollection<KV<String, Long>> sums = Pipeline
   .begin()
   .read(“userRequests”)
   .apply(Window.into(new FixedWindows(2, MINUTES))
                  .trigger(new AtWatermark());
   .apply(new Sum());
```



Google Cloud Platform

A Better Strategy

1. Once per stream time minute
2. At watermark
3. Once per record for two weeks



Google Cloud Platform

Triggers API

```
PCollection<KV<String, Long>> sums = Pipeline
  .begin()
  .read(“userRequests”)
  .apply(Window.into(new FixedWindows(2, MINUTE))
               .trigger(new SequenceOf(
                 new RepeatUntil(
                   new AtPeriod(1, MINUTE),
                   new AtWatermark()),
                 new AtWatermark(),
                 new RepeatUntil(
                   new AfterCount(1),
                   new AfterDelay(
                     14, DAYS, TimeDomain.EVENT_TIME))));
  .apply(new Sum());
```

Lambda vs Streaming

Low-latency, approximate results

Complete, correct results as soon as possible

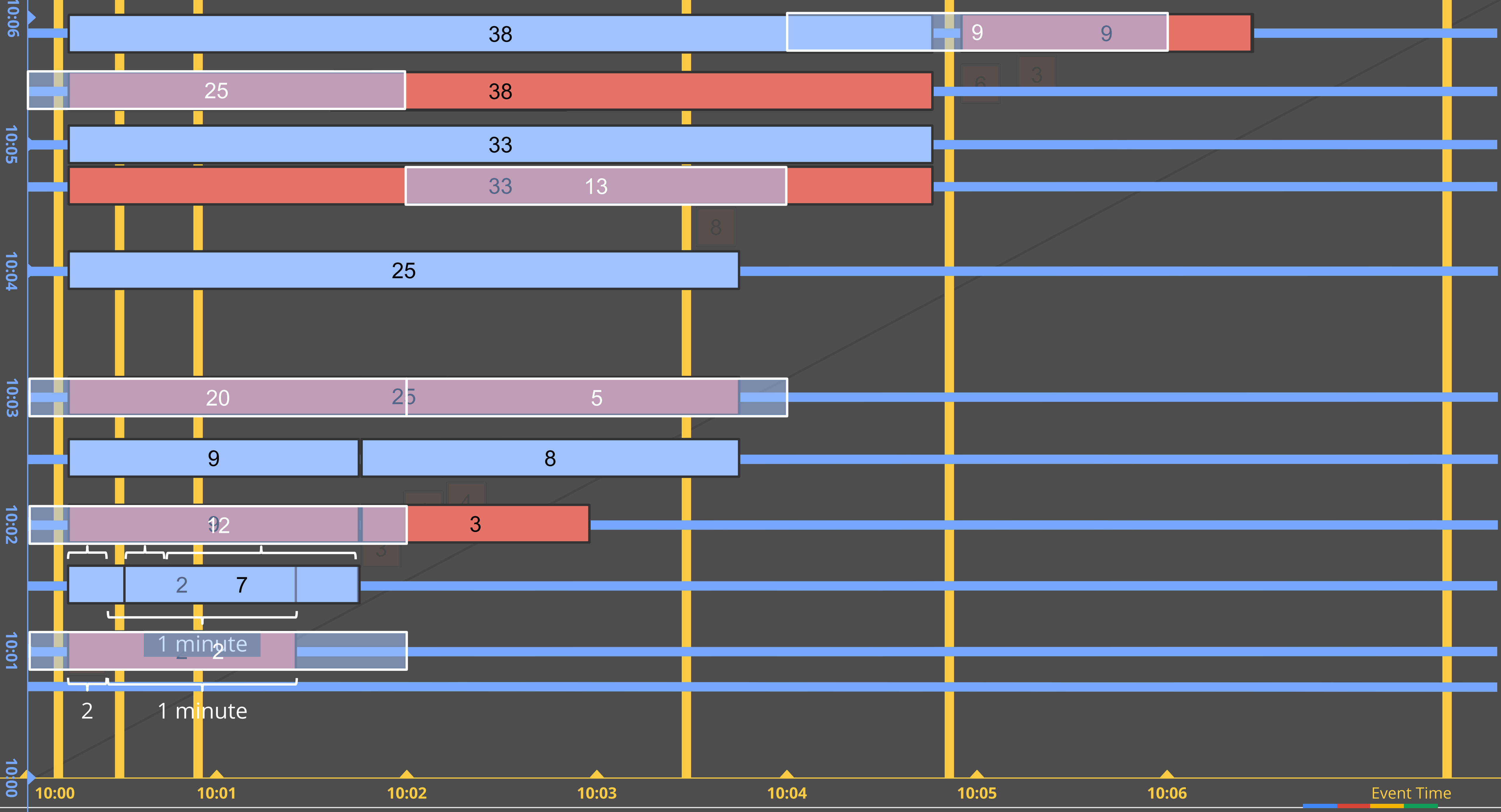
Ability to deal with changes upstream

One Last Thing...

What if I want sessions?

Triggers API

```
PCollection<KV<String, Long>> sums = Pipeline
  .begin()
  .read(“userRequests”)
  .apply(Window.into(new Sessions(1, MINUTE))
               .trigger(new SequenceOf(
                  new RepeatUntil(
                    new AtPeriod(1, MINUTE),
                    new AtWatermark()),
                  new AtWatermark(),
                  new RepeatUntil(
                    new AfterCount(1),
                    new AfterDelay(
                      14, DAYS, TimeDomain.EVENT_TIME))));
  .apply(new Sum());
```



Google Cloud Platform

Summary

Lambda is great

Streaming by itself is better :-)

Strong Consistency = Correctness

Streaming = Aggregation + Windowing + Triggers

Tools For Reasoning About Time = Power + Flexibility

Thank you!

Questions?

Questions about this talk: takidau@google.com (Tyler Akidau)
Questions about Cloud Dataflow: cloude@google.com (Eric Schmidt)