Big Data Stream Processing

Tilmann Rabl

Berlin Big Data Center

www.dima.tu-berlin.de | bbdc.berlin | rabl@tu-berlin.de

Agenda

Introduction to Streams

- Use cases
- Stream Processing 101

Stream Processing Systems

- Ingredients of a stream processing system
- Some examples
- More details on Storm, Spark, Flink
- Maybe a demo (!)

Stream Processing Optimizations (if we have time)

- How to optimize

With slides from Data Artisans, Volker Markl, Asterios Katsifodimos, Jonas Traub

Big Fast Data

- Data is growing and can be evaluated
  - Tweets, social networks (statuses, check-ins, shared content), blogs, click streams, various logs, …
  - *Facebook: > 845M active users, > 8B messages/day*
  - *Twitter: > 140M active users, > 340M tweets/day*
- Everyone is interested!

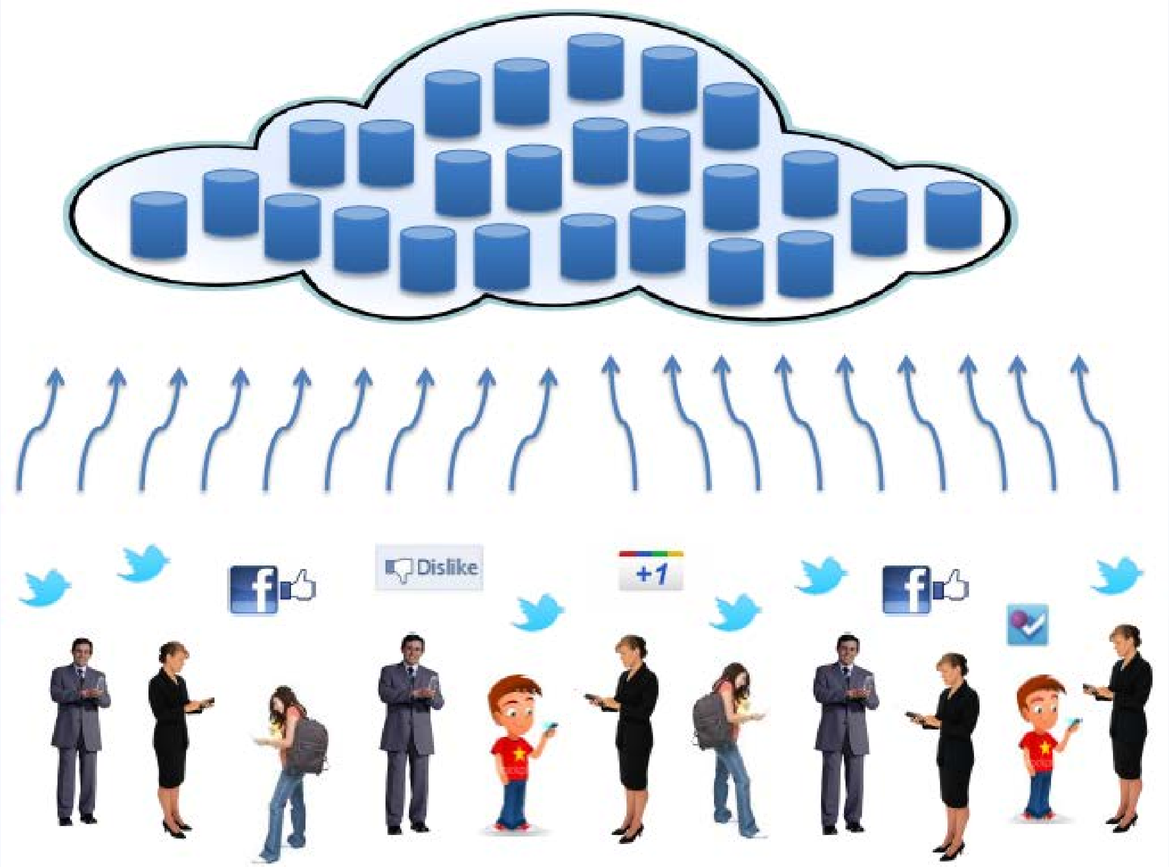


Image: Michael Carey

But there is so much more…

- Autonomous Driving
  - Requires rich navigation info
  - Rich data sensor readings
  - 1GB data per minute per car (all sensors)[1]

- Traffic Monitoring
  - High event rates: millions events / sec
  - High query rates: thousands queries / sec
  - Queries: filtering, notifications, analytical

Source: http://theroadtochangeindia.wordpress.com/2011/01/13/better-roads/

- Pre-processing of sensor data
  - CERN experiments generate ~1PB of measurements per second.
  - Unfeasible to store or process directly, fast preprocessing is a must.

[1]Cobb: http://www.hybridcars.com/tech-experts-put-the-brakes-on-autonomous-cars/

Why is this hard?

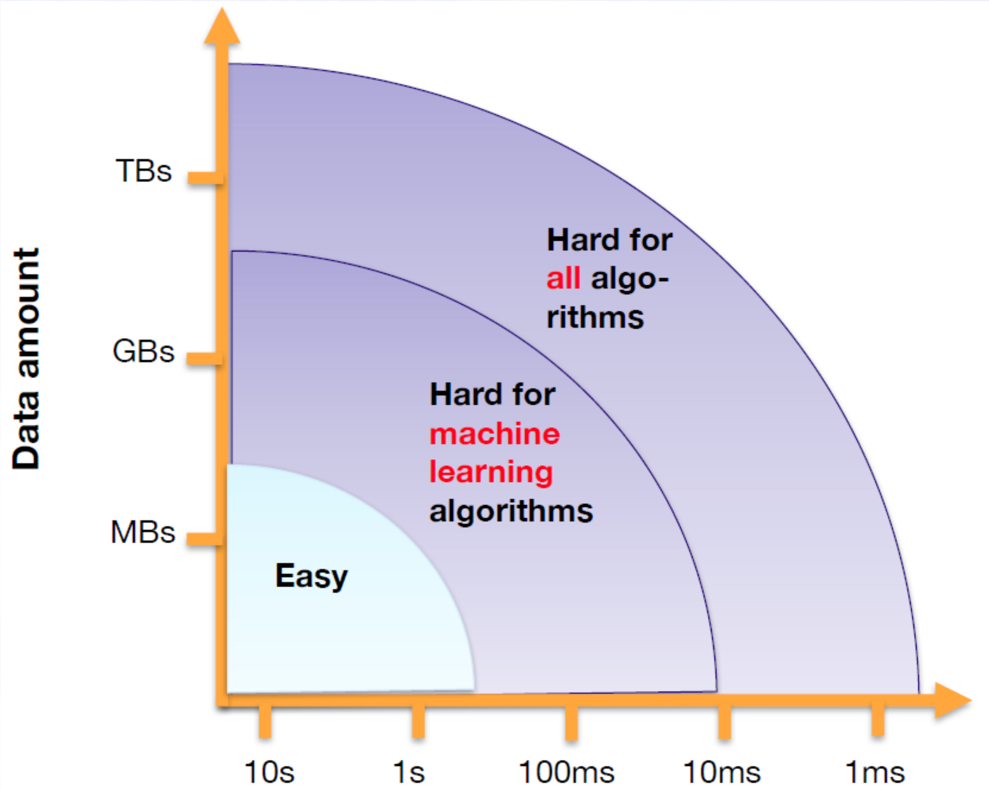


Image: Peter Pietzuch

Tension between *performance* and *algorithmic expressiveness*

Stream Processing 101

With some Flink Examples

Based on the Data Flow Model

What is a Stream?

- Unbounded data
  - Conceptually infinite, ever growing set of ***data items / events***
  - Practically continuous stream of data, which needs to be processed / analyzed

- Push model
  - Data production and procession is controlled by the source
  - Publish / subscribe model

- Concept of time
  - Often need to reason about ***when*** data is produced and when processed data should be output
  - Time agnostic, processing time, ingestion time, event time

This part is largely based on Tyler Akidau's great blog on streaming - https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101

Stream Models

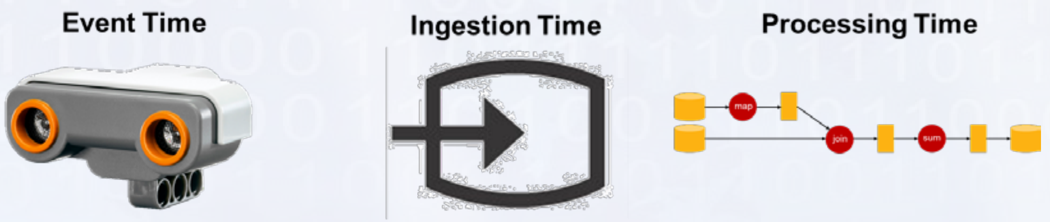
$S = s_i, s_{i+1}, \ldots$ $\quad s_i$ = <data item, *timestamp*>

- Turnstile
  - Elements can come and go
  - Underlying model is a vector of elements (domain)
  - s_i is an update (increment or decrement) to a vector element
  - Traditional database model
  - Flexible model for algorithms

- Cash register
  - Similar to turnstile, but elements cannot leave

- Time series
  - s_i is is a new vector entry
  - Vector is increasing
  - This is what all big stream processing engines use

Event Time



- Event time
  - Data item production time
- Ingestion time
  - System time when data item is received
- Processing time
  - System time when data item is processed

- Typically, these do not match!
- In practice, streams are unordered!

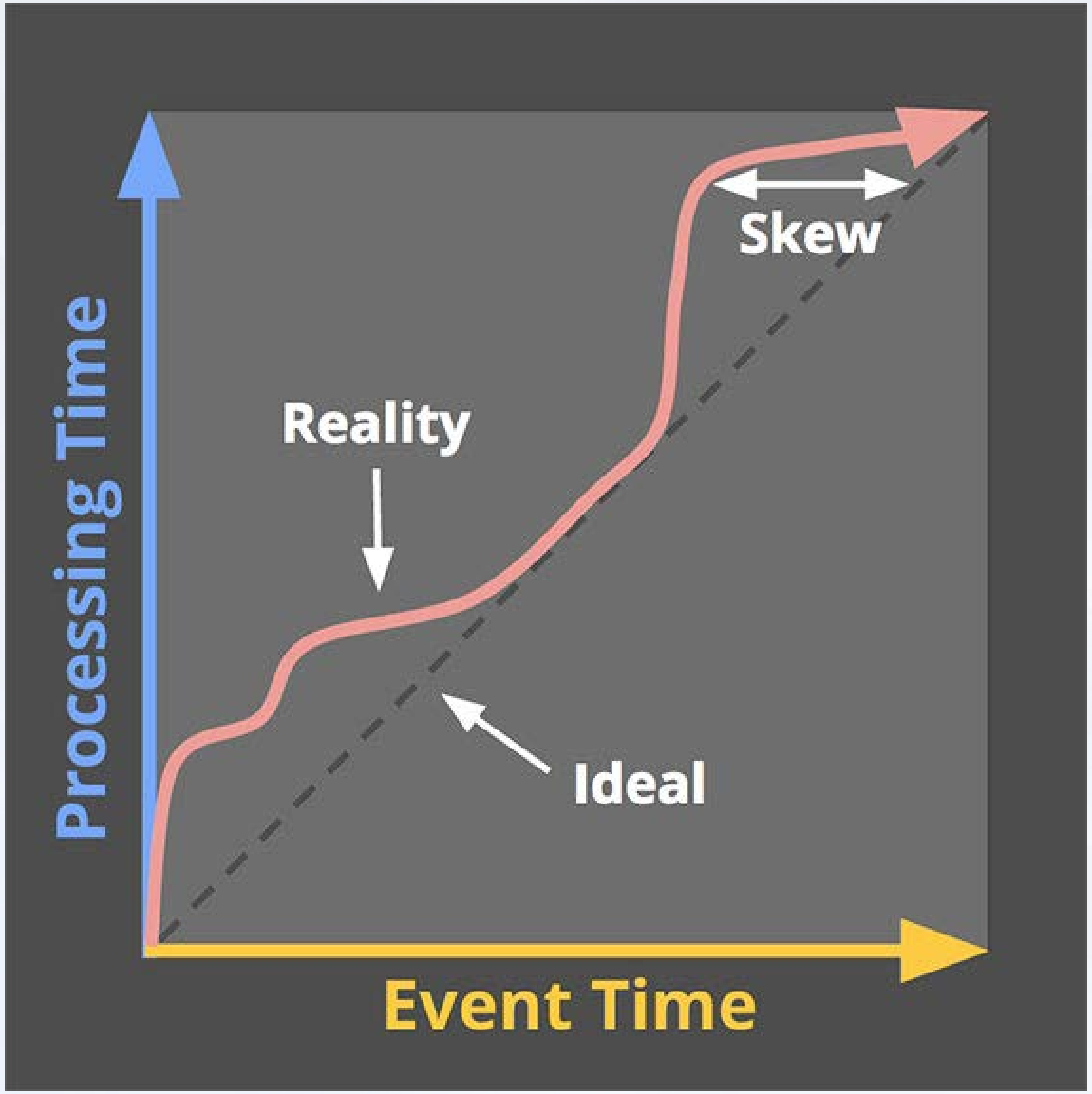


Image: Tyler Akidau

Time Agnostic Processing

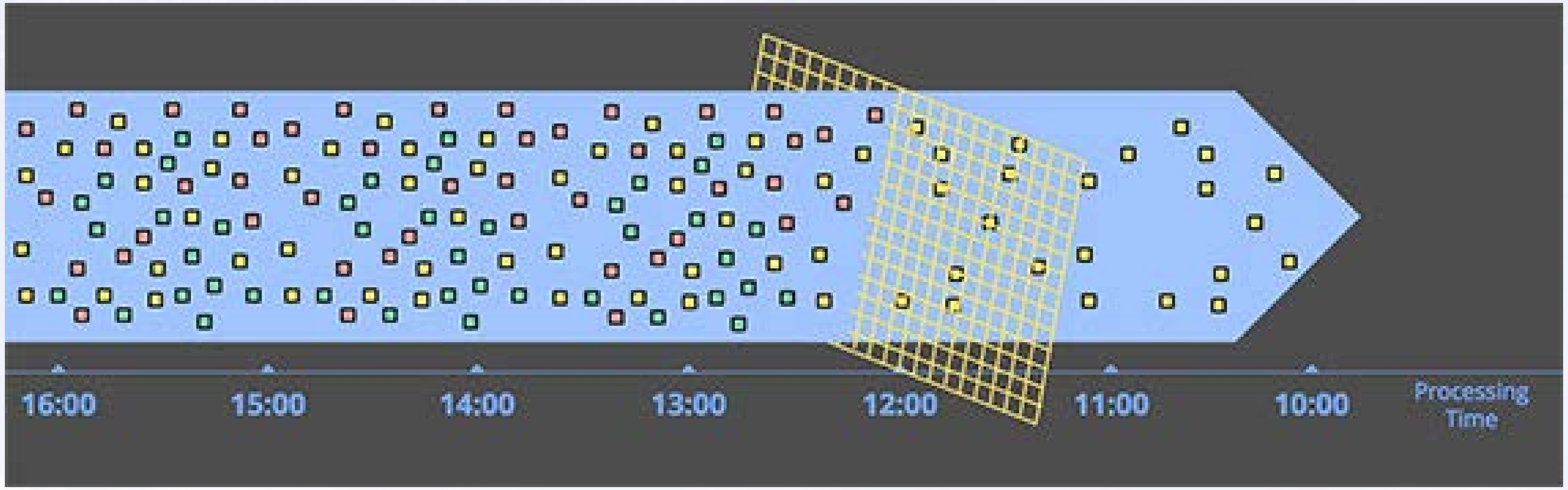


Image: Tyler Akidau

- Filtering
  - Stateless
  - Can be done per data item
  - Implementations: hash table or bloom filter

Time Agnostic Processing II

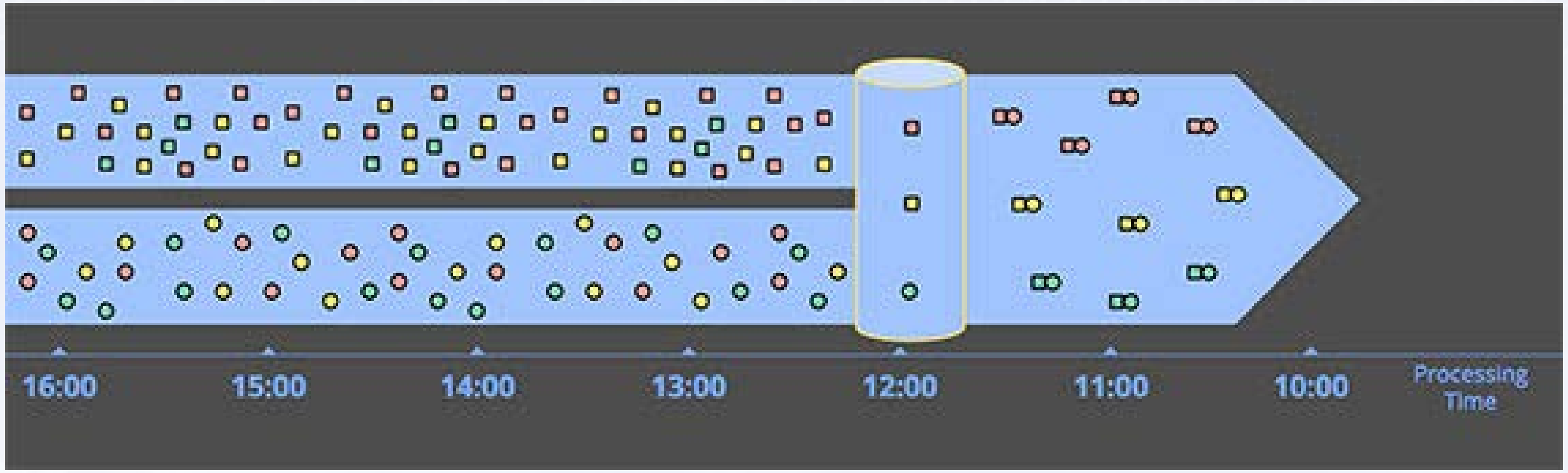


Image: Tyler Akidau

- Inner join
  - Only current elements
  - Stateful
  - E.g., hash join
- What about other joins (e.g., outer join)?

Approximate Processing

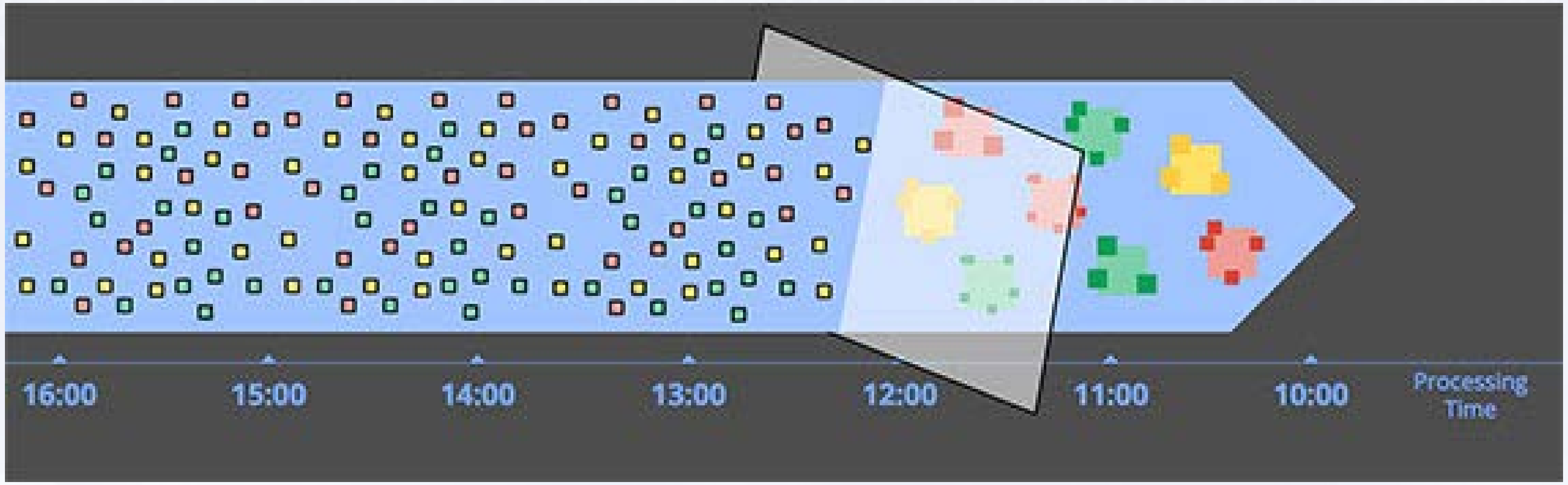


Image: Tyler Akidau

- Streaming k-means, sketches
  - Low overhead
  - Notion of time
- Not covered in this talk

Windows

- Fixed
  - Also tumbling
- Sliding
  - Also hopping
- Session
  - Based on activity

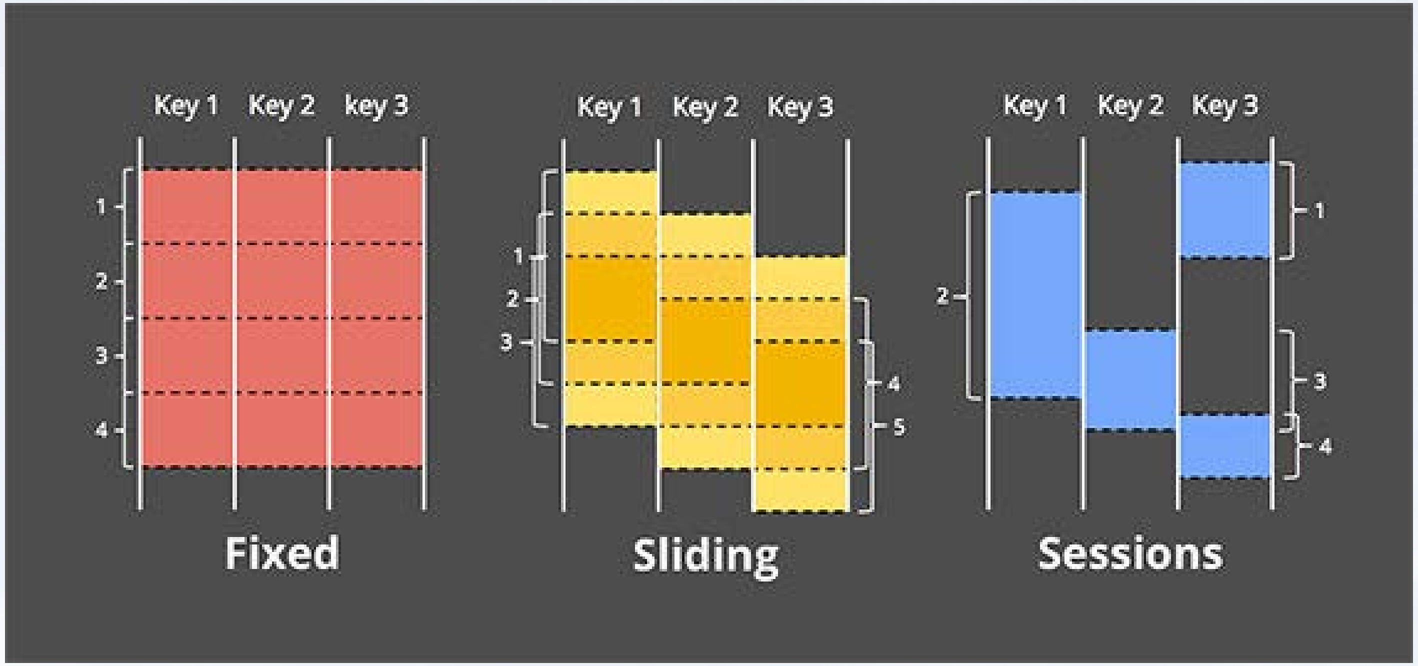


Image: Tyler Akidau

- Triggered by
  - Event time, processing time, count, *watermark*
- Eviction policy
  - Window width / size

Processing Time Windows

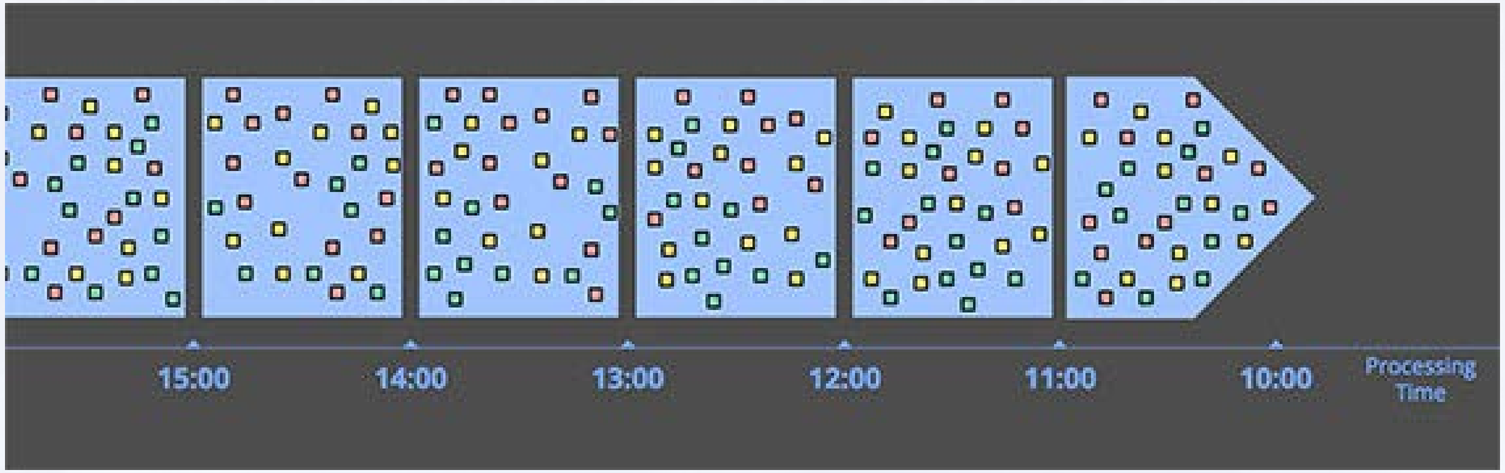
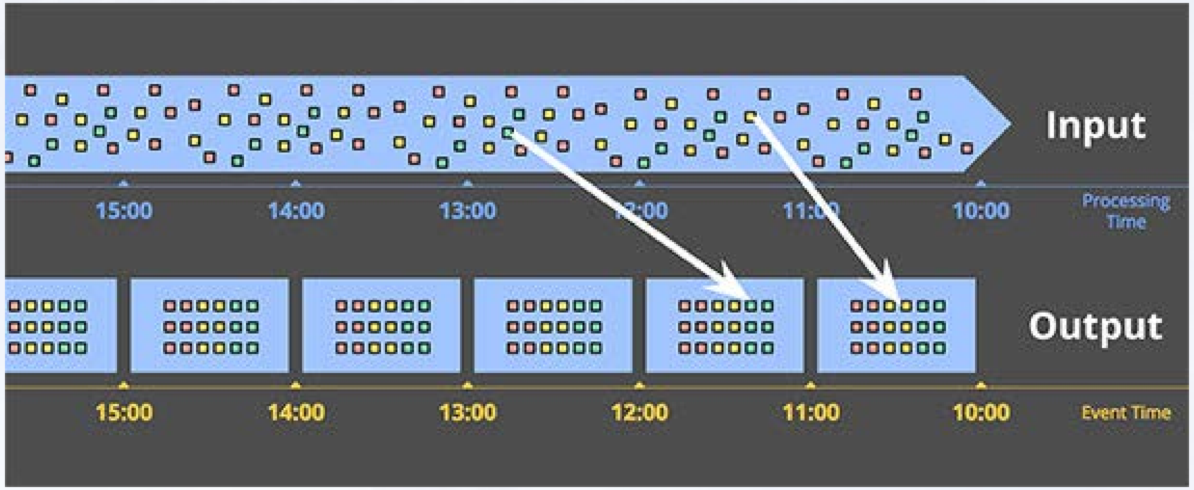
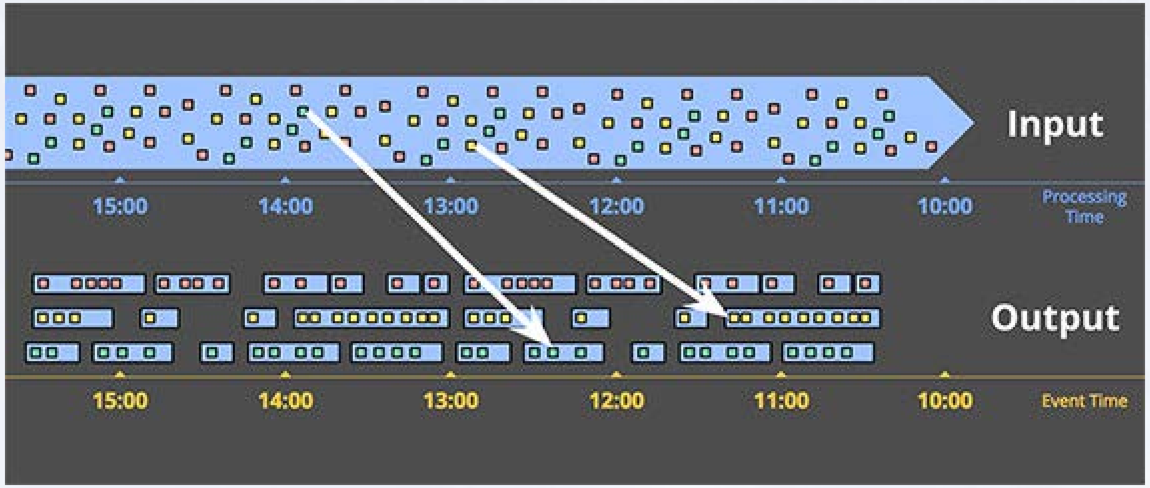


Image: Tyler Akidau

- System waits for x time units
  - System decides on stream partitioning
  - Simple, easy to implement
  - Ignores any time information in the stream -> any aggregation can be arbitrary
- Similar: Counting Windows

Event Time Windows



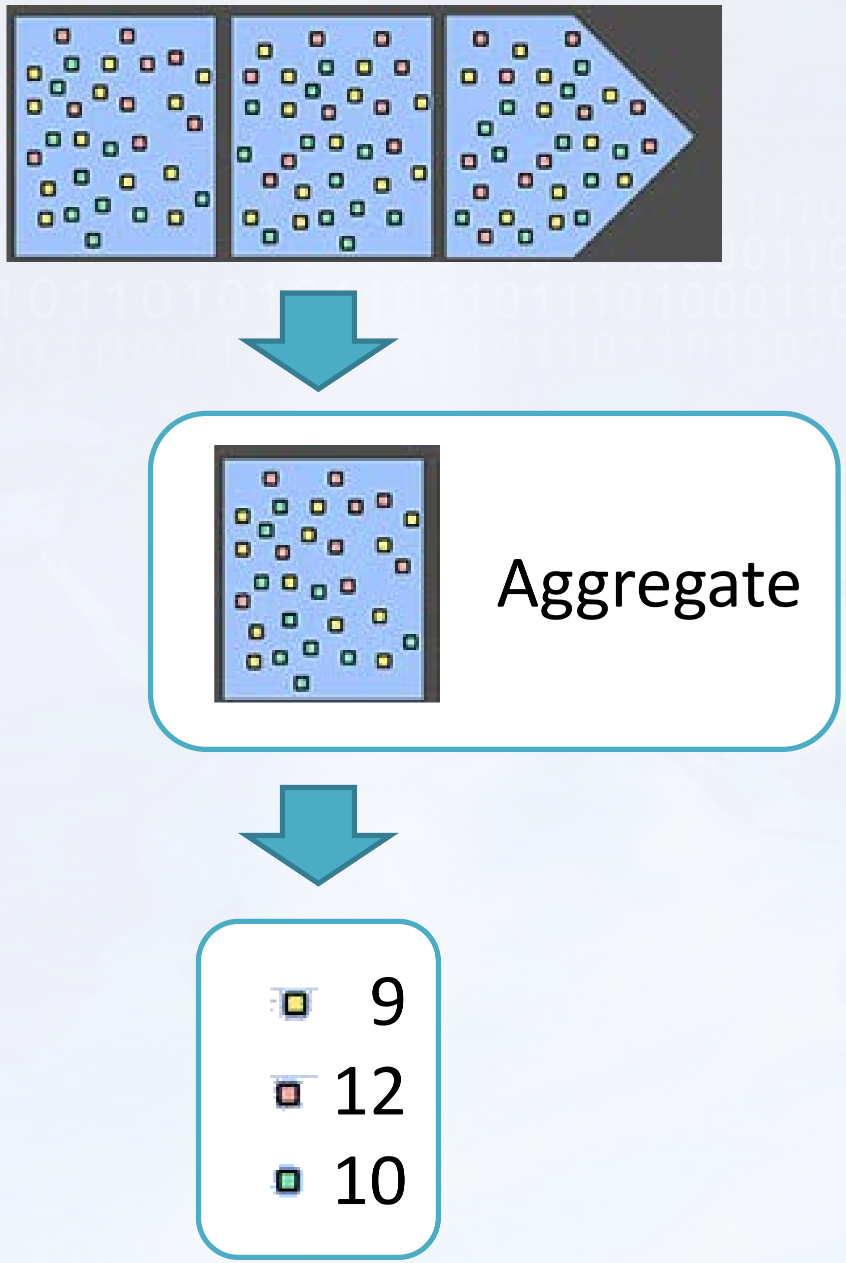


Images: Tyler Akidau

- Windows based on the time information in stream
  - Adheres to stream semantic
  - Correct calculations
  - Buffering required, potentially unordered (more on this later)

Basic Stream Operators

- Windowed Aggregation
  - E.g., average speed
  - Sum of URL accesses
  - Daily highscore
- Windowed Join
  - Correlated observations in timeframe
  - E.g., temperature in time



Flink's Windowing

- Windows can be any combination of (multiple) triggers & evictions
  - Arbitrary tumbling, sliding, session, etc. windows can be constructed.

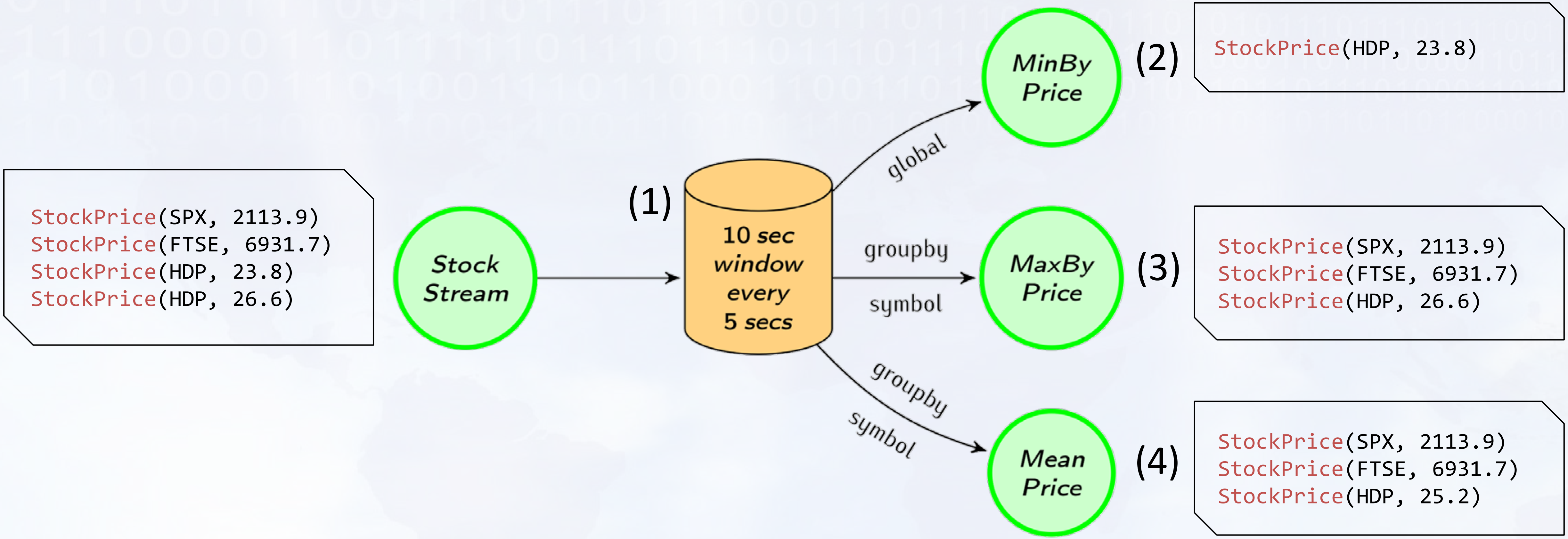
- Common triggers/evictions part of the API
  - Time (processing vs. event time), Count

- Even more flexibility: *define your own UDF* trigger/eviction

- Examples:

```
dataStream.windowAll(TumblingEventTimeWindows.of(Time.seconds(5)));
dataStream.keyBy(0).window(TumblingEventTimeWindows.of(Time.seconds(5)));
```

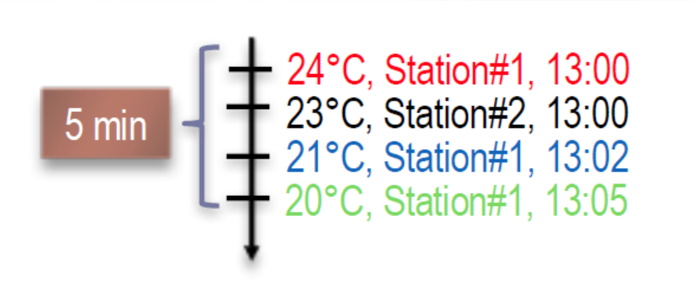
Example Analysis: Windowed Aggregation




```
(1) val windowedStream = stockStream.window(Time.of(10, SECONDS)).every(Time.of(5, SECONDS))
(2) val lowest = windowedStream.minBy("price")
(3) val maxByStock = windowedStream.groupBy("symbol").maxBy("price")
(4) val rollingMean = windowedStream.groupBy("symbol").mapWindow(mean _)
```

Complex Event Processing

- Detecting patterns in a stream
- Complex event = sequence of events
- Defined using logical and temporal conditions
  - Logical: data values and combinations
  - Temporal: within a given period of time

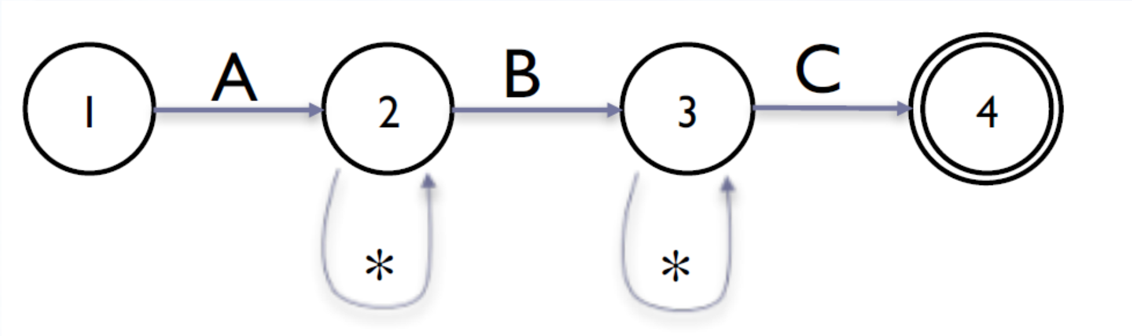



```
SEQ(A, B, C) WITH
  A.Temp > 23°C &&
  B.Station = A.Station && B.Temp < A.Temp &&
  C.Station = A.Station && A.Temp-C.Temp > 3
```

Slide by Kai-Uwe Sattler

Complex Event Processing Contd.

- Composite events constructed e.g. by
  - SEQ, AND, OR, NEG, ...
  - SEQ(e1, e2) → (e1, t1) ∧ (e2, t2)∧t1 ≤ t2 ∧ e1,e2 ε 𝕎
- Implemented by constructing a NFA
  - Example: SEQ(A, B, C)



Slide by Kai-Uwe Sattler

Stream Processing Systems

What makes a system a stream processing system?

8 Requirements of Big Streaming

- Keep the data moving
  - Streaming architecture
- Declarative access
  - E.g. StreamSQL, CQL
- Handle imperfections
  - Late, missing, unordered items
- Predictable outcomes
  - Consistency, event time
- Integrate stored and streaming data
  - Hybrid stream and batch
- Data safety and availability
  - Fault tolerance, durable state
- Automatic partitioning and scaling
  - Distributed processing
- Instantaneous processing and response

The 8 Requirements of Real-Time Stream Processing – Stonebraker et al. 2005

8 Requirements of Big Streaming

- **Keep the data moving**
  - Streaming architecture
- Declarative access
  - E.g. StreamSQL, CQL
- **Handle imperfections**
  - Late, missing, unordered items
- Predictable outcomes
  - Consistency, event time
- Integrate stored and streaming data
  - Hybrid stream and batch
- **Data safety and availability**
  - Fault tolerance, durable state
- **Automatic partitioning and scaling**
  - Distributed processing
- Instantaneous processing and response

The 8 Requirements of Real-Time Stream Processing – Stonebraker et al. 2005

Big Data Processing

- Databases can process very large data since forever (see VLDB)
  - Why not use those?
- Big data is not (fully) structured
  - No good for database ☹
- We want to learn more from data than just
  - Select, project, join
- First solution: MapReduce

Map Reduce

- Framework / programming model by Google
  - Presented 2004 at OSDI'04
- Inspired by map and reduce functions in functional languages / MPI
  - Second order functions
- Simple parallelization model for shared nothing architectures (“commodity hardware”)
- Apache Hadoop
  - Open-source implementation
  - Initiated at Yahoo

Map: Computation

For each input create list of output values

Example:

For each word in a sentence emit a k/v pair indicating one occurrence of the word

(key, “hello world”) -> (“hello”,”1”), (“world”,”1”)

Signature

```
map (key, value) -> list(key’, value’)
```

Reduce: Aggregation

Combine all intermediate values for one key

Example:

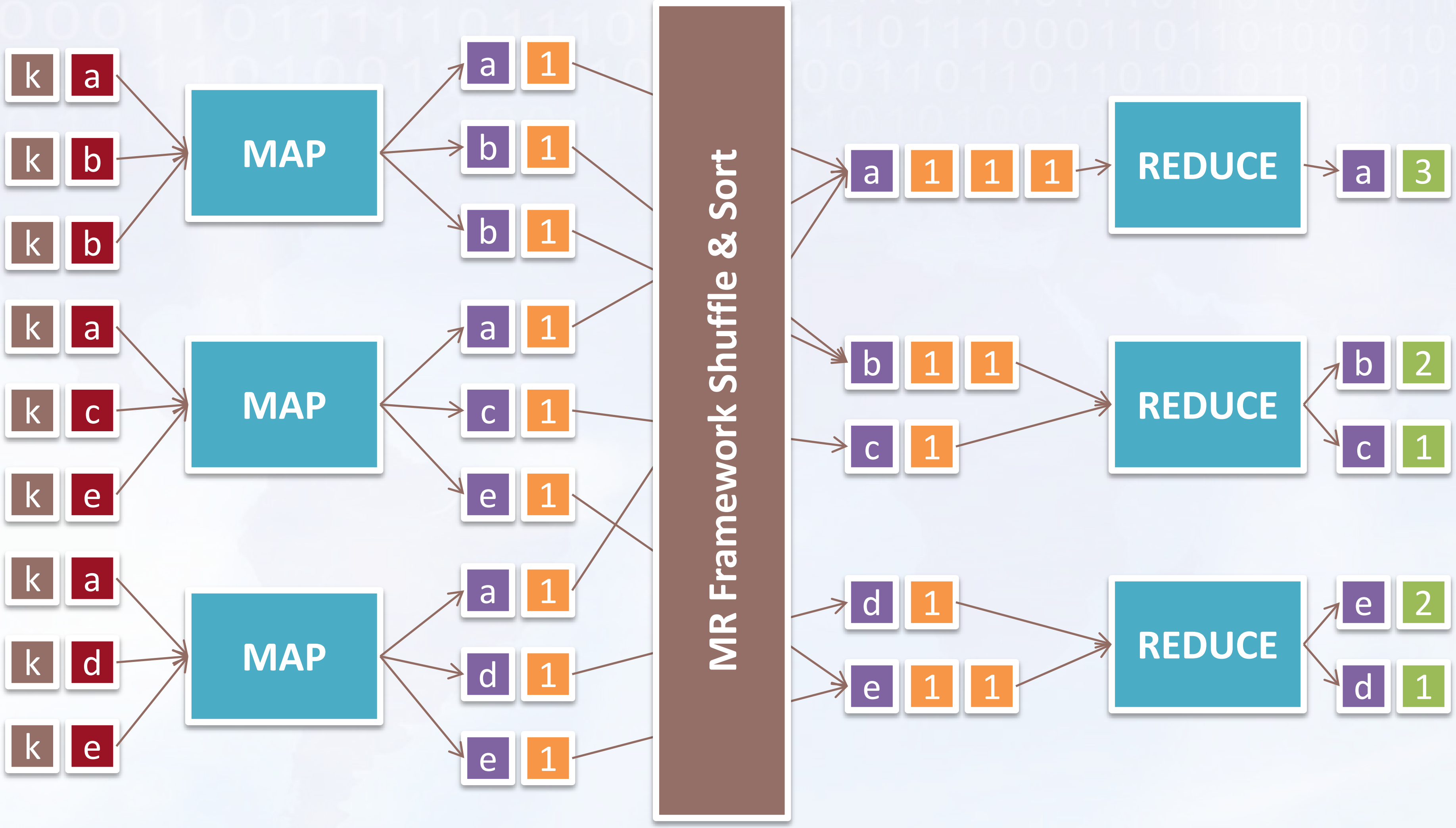
Sum up all values for the same key

(“Hello”,(“1”, “1”, “1”, “1”)) -> (“Hello”,(“4”))

Signature

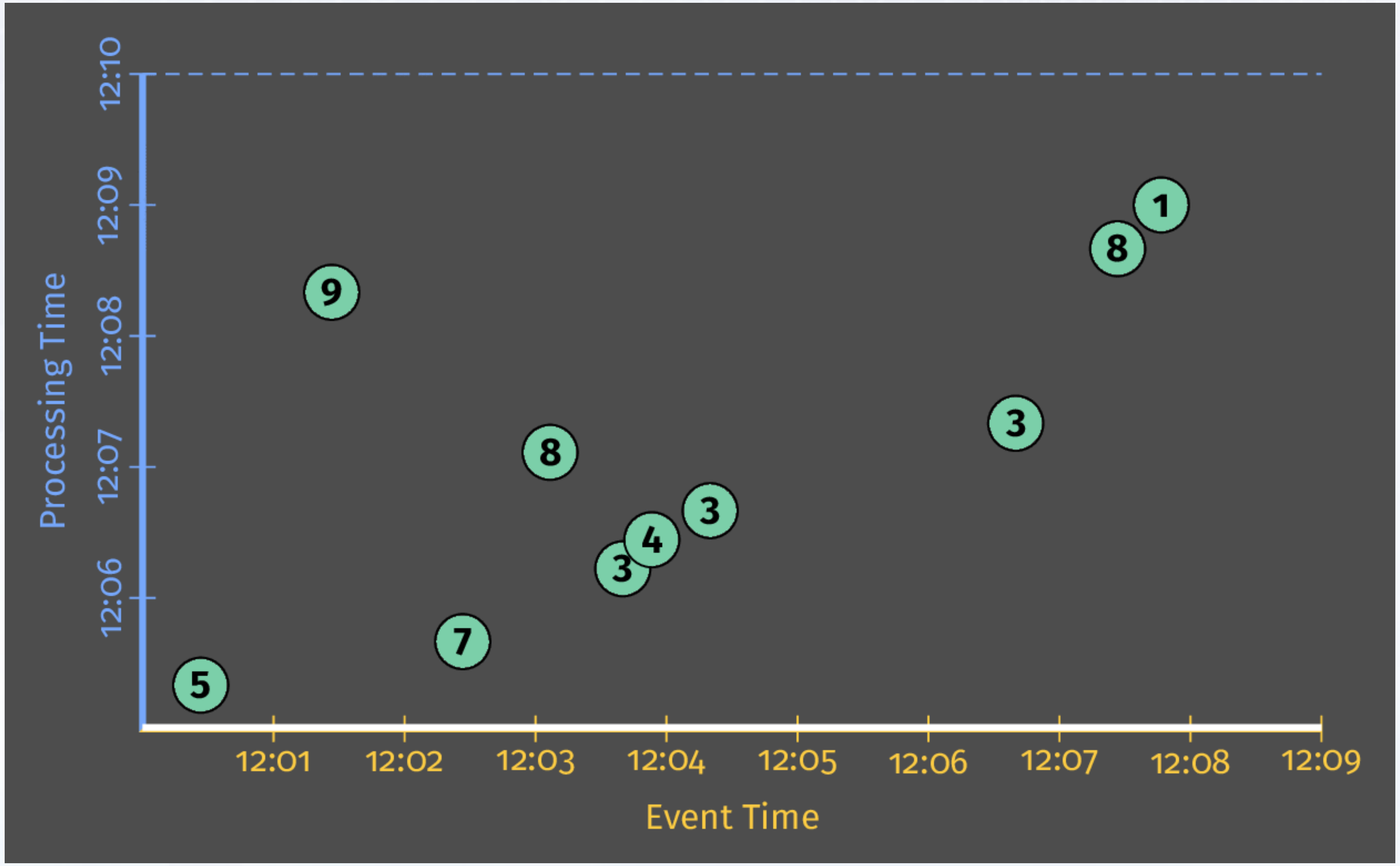
```
reduce (key, list(value)) -> list(value’)
```

MR Data Flow

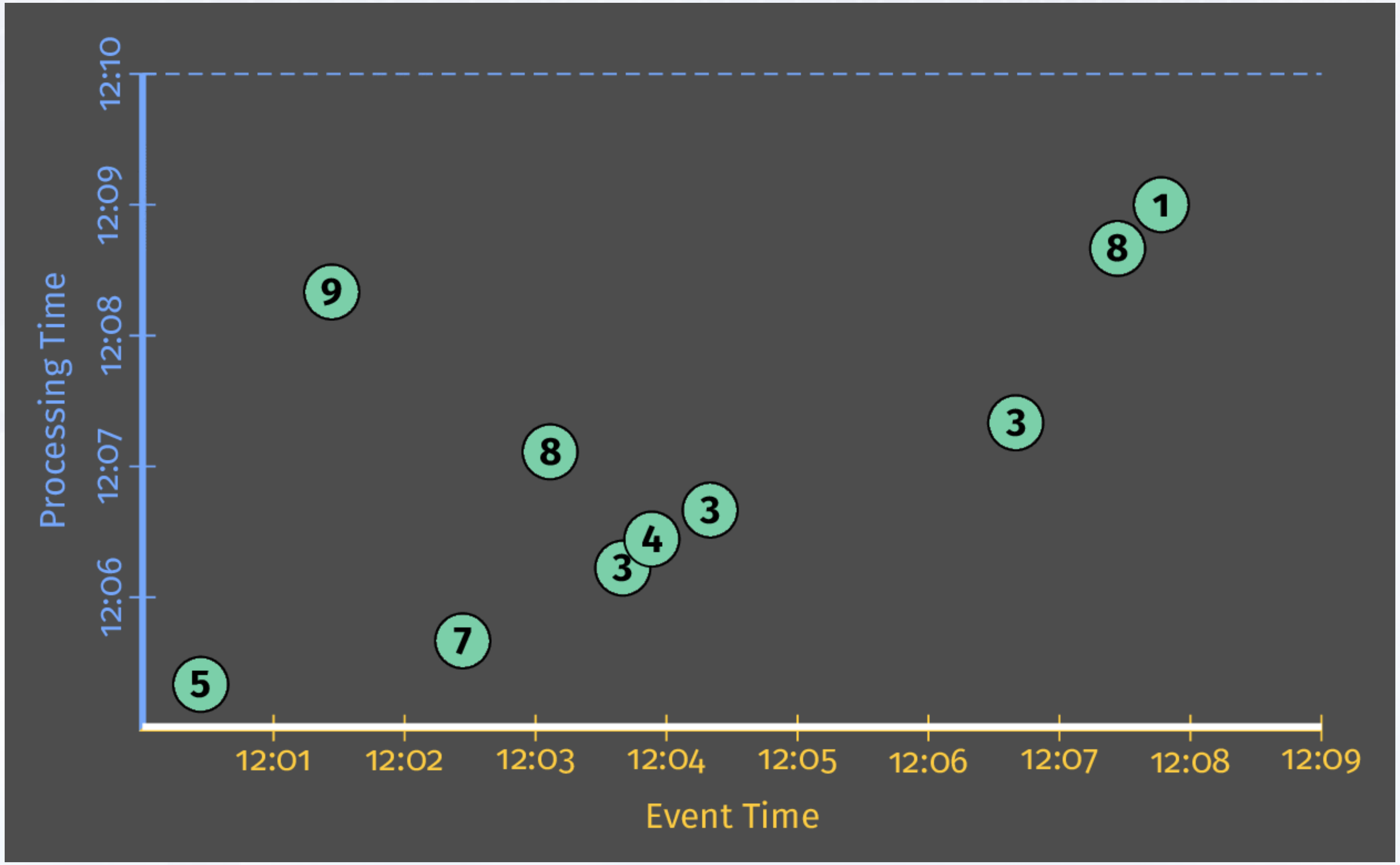


k a
k b
k b
k a
k c
k e
k a
k d
k e
MAP
MAP
MAP
a 1
b 1
b 1
a 1
c 1
e 1
a 1
d 1
e 1
MR Framework Shuffle & Sort
a 1 1 1
b 1 1
c 1
d 1
e 1 1
REDUCE
REDUCE
REDUCE
a 3
b 2
c 1
e 2
d 1

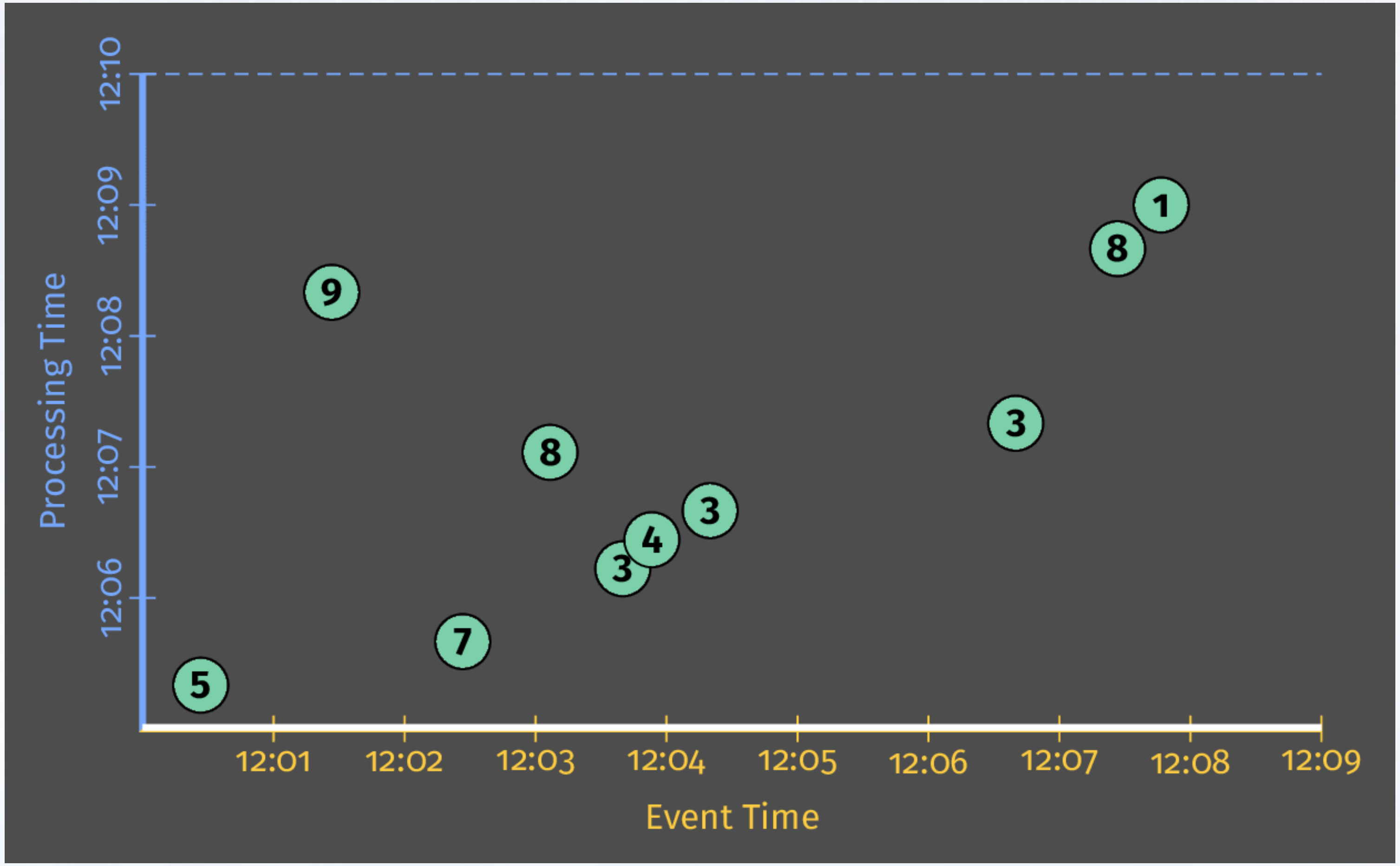
MR / Batch Processing



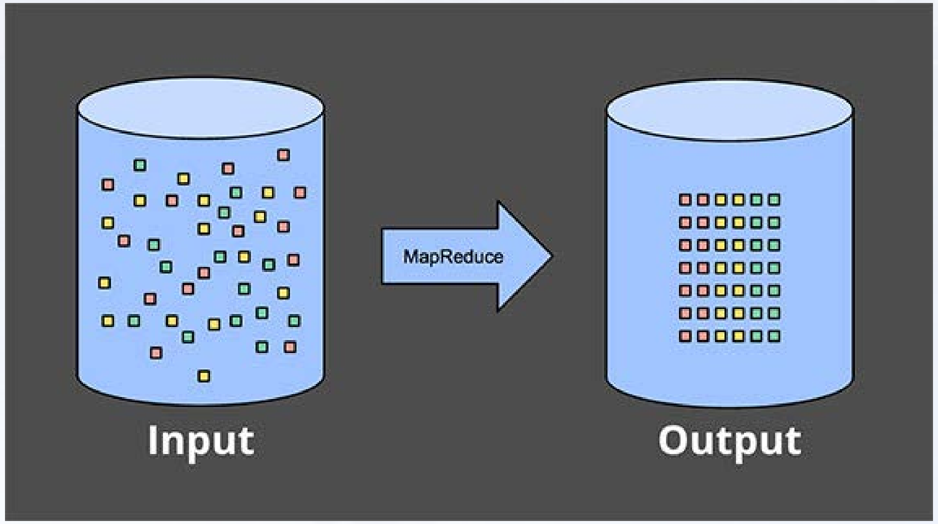
MR / Batch Processing

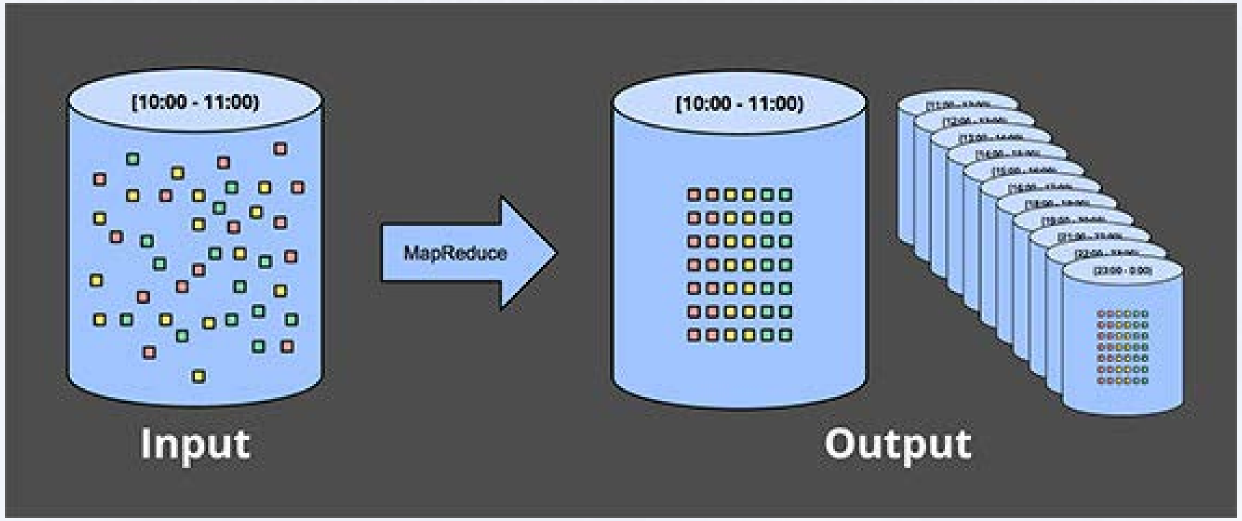


MR / Batch Window Processing



MR Discussion



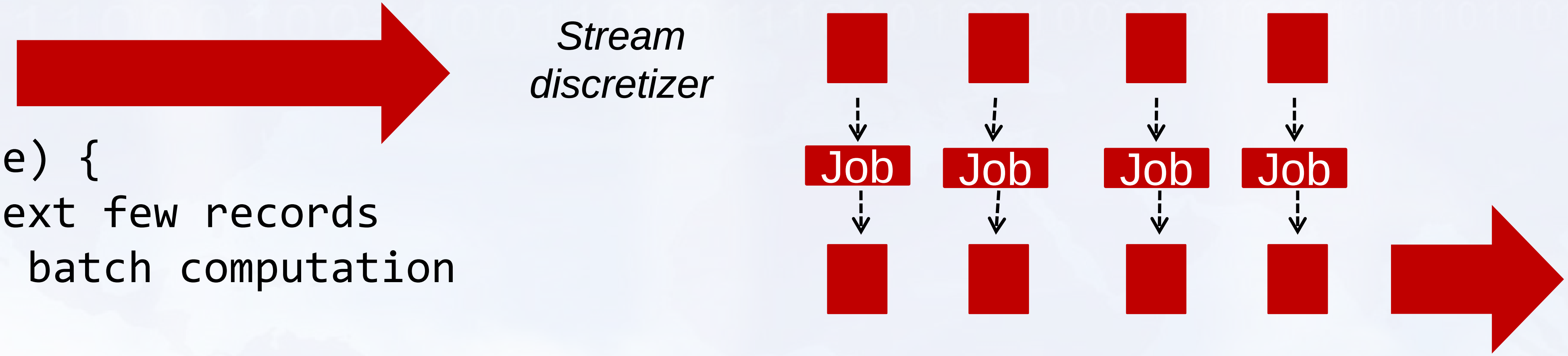


Images: Tyler Akidau

- Great for large amounts of static data
- For streams: only for large windows
- *Data is not moving!*
- *High latency, low efficiency*

How to keep data moving?

Discretized Streams (mini-batch)




```
while (true) {
  // get next few records
  // issue batch computation
}
```

Native streaming


```
while (true) {
   // process next record
}
```

Discussion of Mini-Batch

- Easy to implement
- Easy consistency and fault-tolerance
- Hard to do event time and sessions

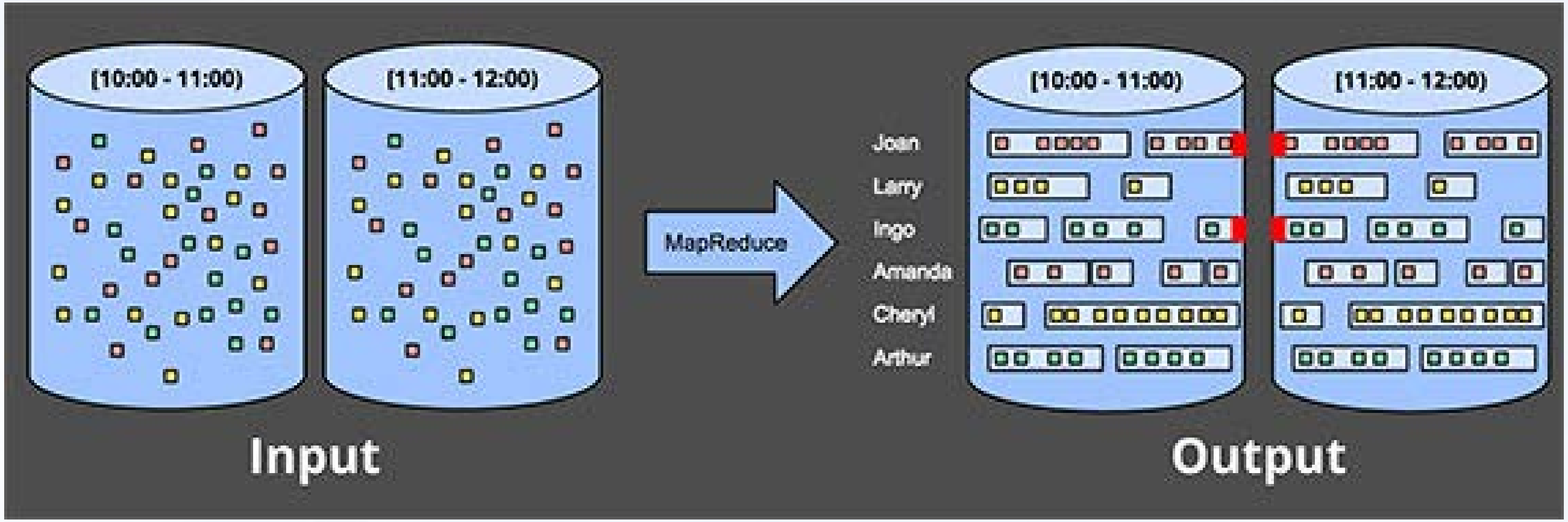
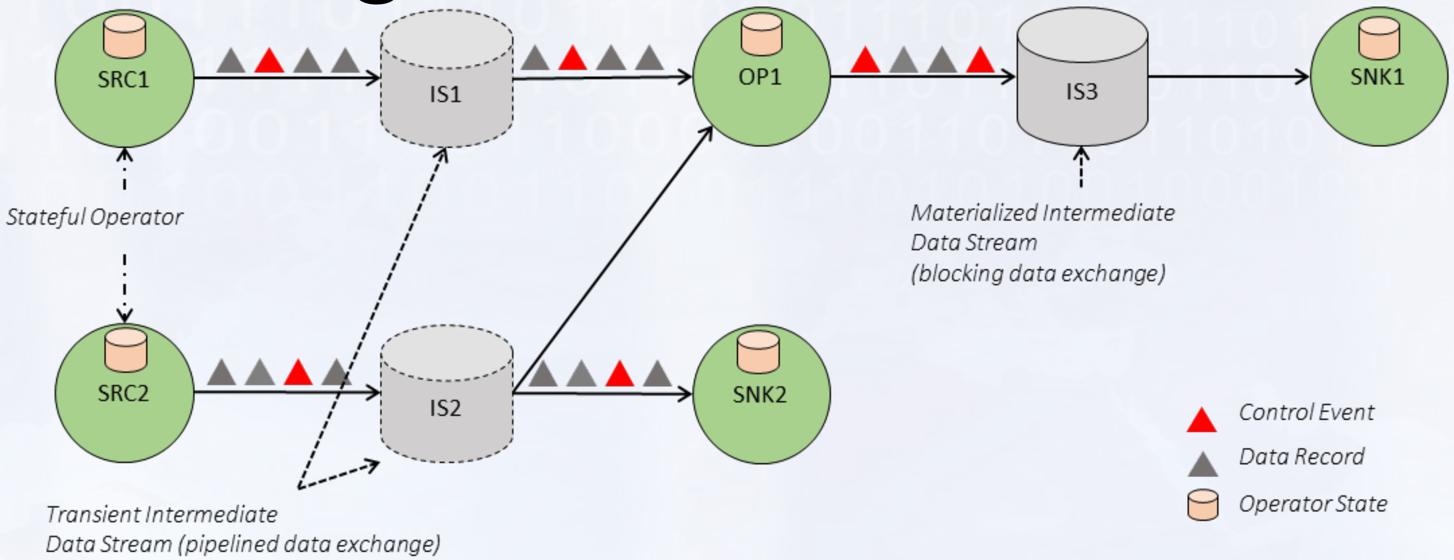


Image: Tyler Akidau

True Streaming Architecture



- Program = DAG* of operators and intermediate streams
- Operator = computation + state
- Intermediate streams = logical stream of records

- Stream transformations
  - Basic transformations: *Map, Reduce, Filter, Aggregations*…
  - Binary stream transformations: *CoMap, CoReduce*…
  - Windowing semantics: *Policy based flexible windowing (Time, Count, Delta…)*
  - Temporal binary stream operators: *Joins, Crosses*…
  - Native support for iterations

Handle Imperfections – Watermarks

- Data items arrive early, on-time, or late
- Solution: Watermarks
  - Perfect or heuristic measure on when window is complete

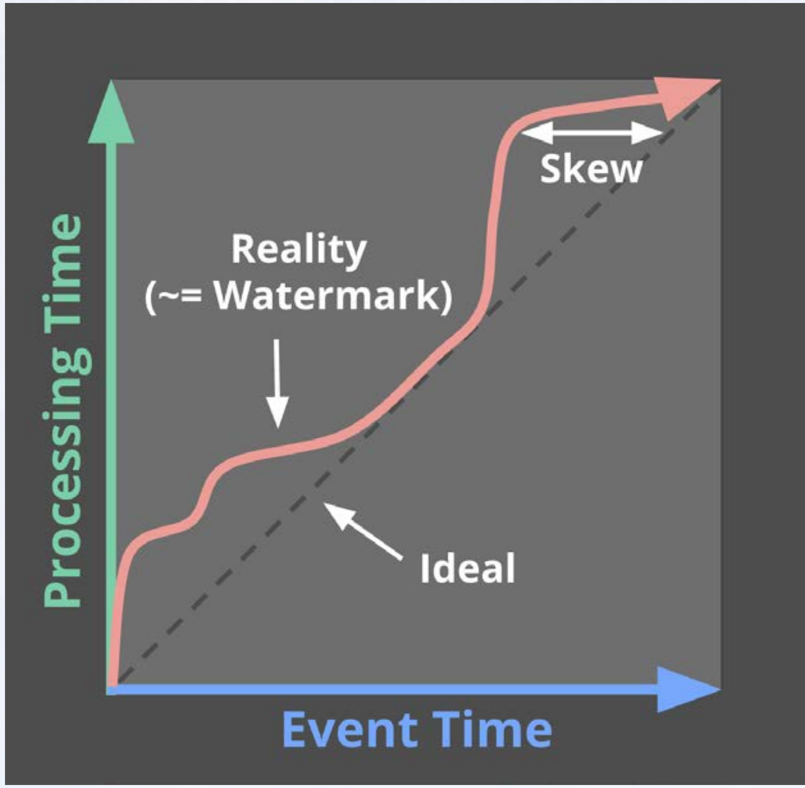


Image: Tyler Akidau

Handle Imperfections – Watermarks

- Data items arrive early, on-time, or late
- Solution: Watermarks
  - Perfect or heuristic measure on when window is complete

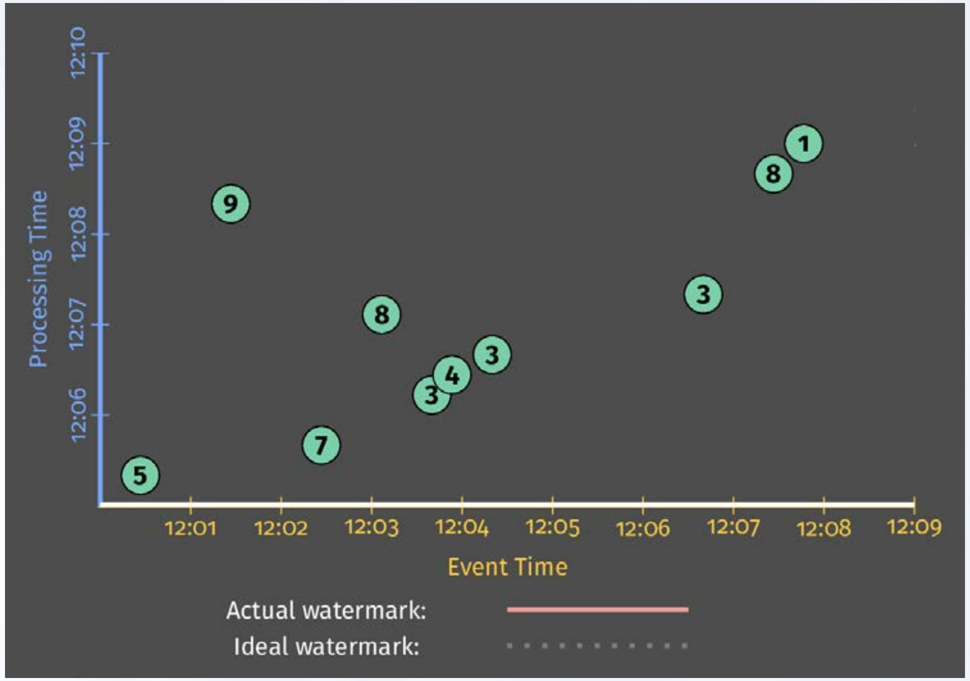


Image: Tyler Akidau

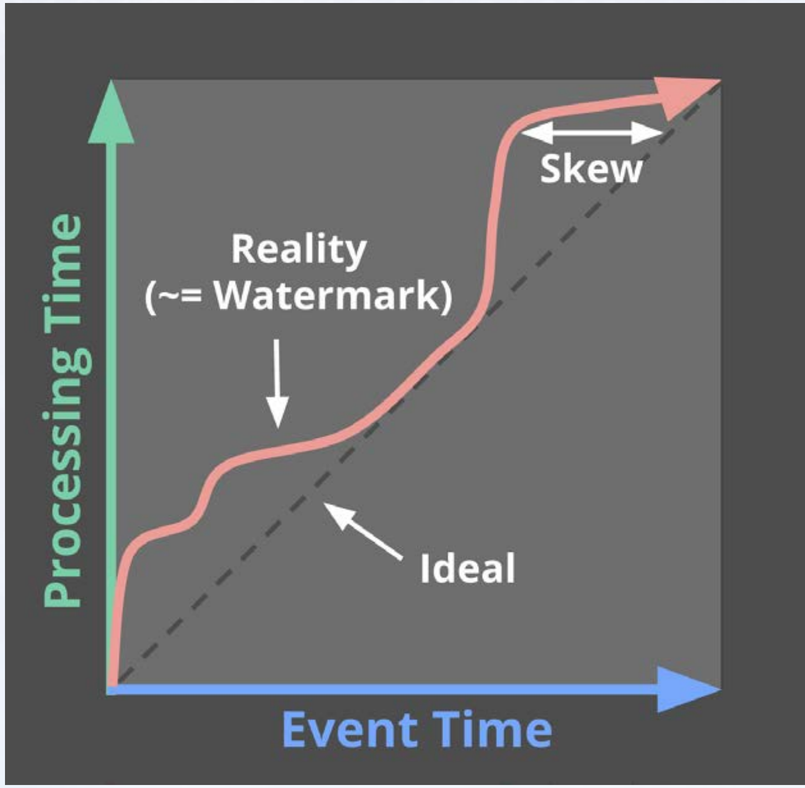


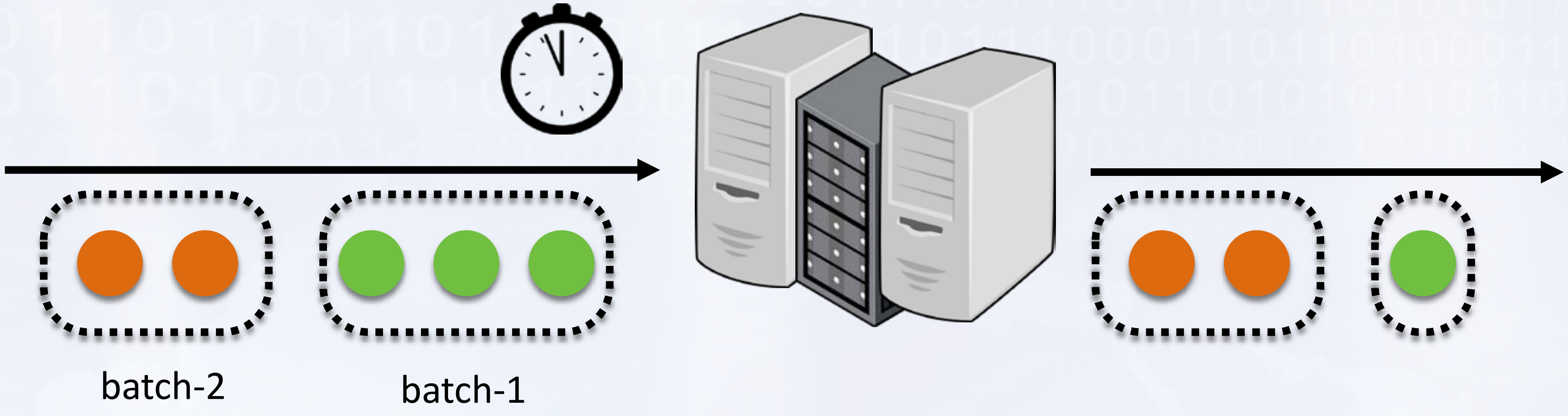
Image: Tyler Akidau

Data Safety and Availability

- Ensure that operators see all events
  - “At least once”
  - Solved by replaying a stream from a checkpoint
  - No good for correct results
- Ensure that operators do not perform duplicate updates to their state
  - “Exactly once”
  - Several solutions
- Ensure the job can survive failure

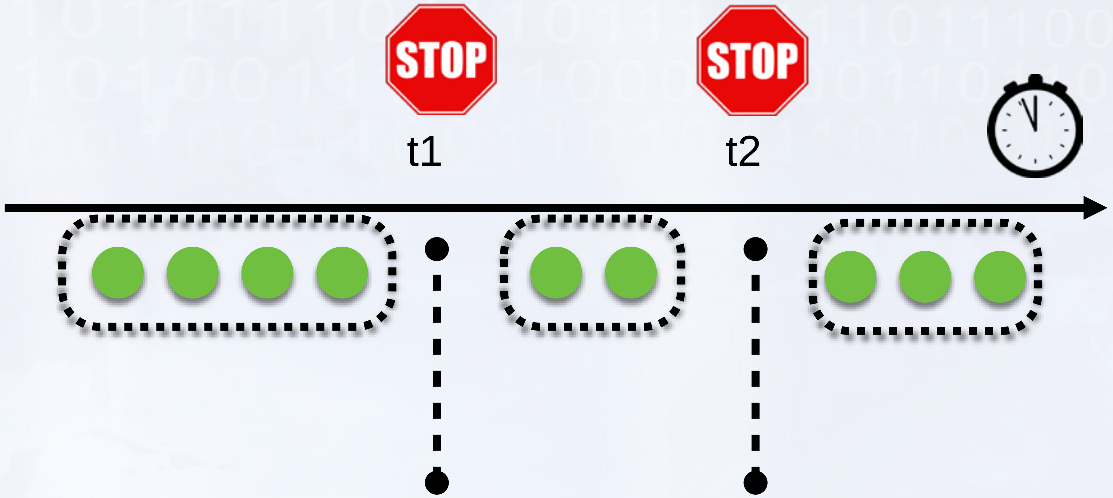
37

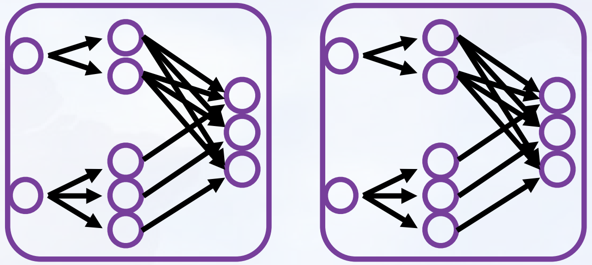
Lessons Learned from Batch



- If a batch computation fails, simply repeat computation as a transaction
- Transaction rate is **constant**
- Can we apply these principles to a true streaming execution?

Taking Snapshots – the naïve way



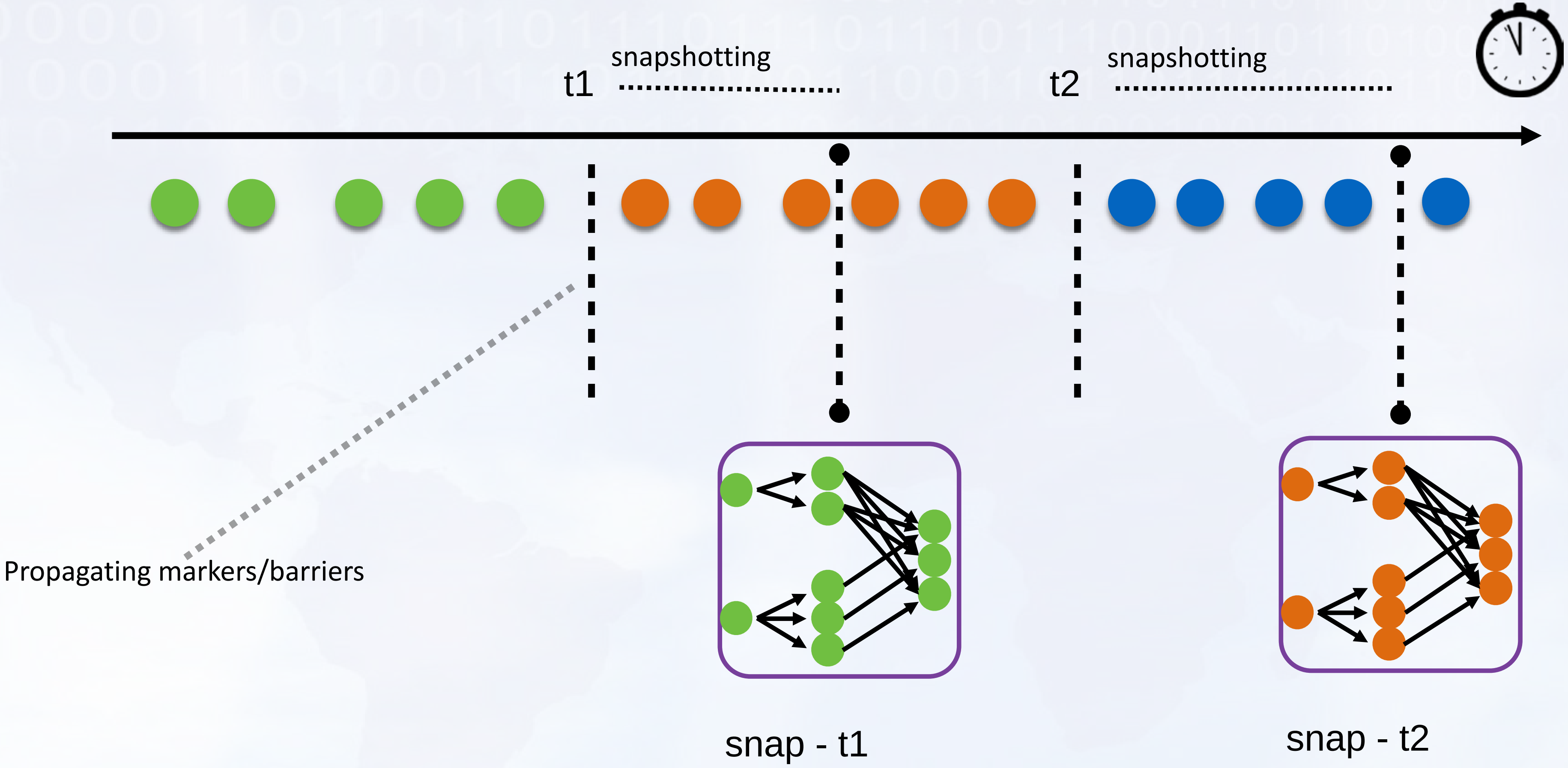


execution snapshots

Initial approach (e.g., Naiad)

- Pause execution on t1,t2,..
- Collect state
- Restore execution

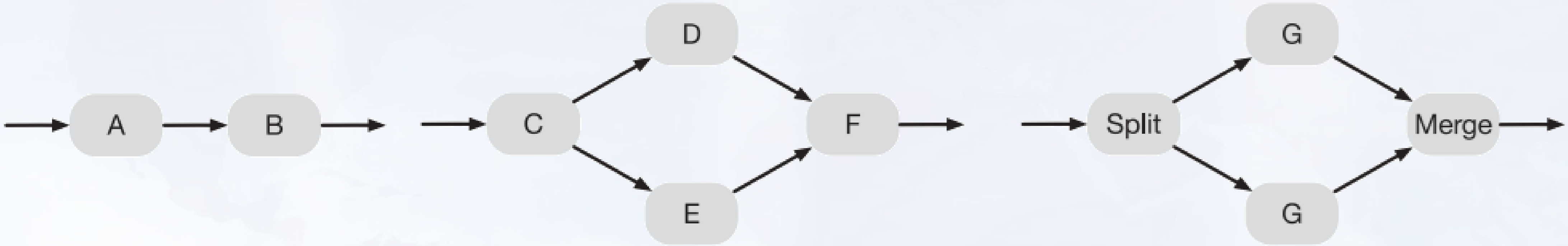
Asynchronous Snapshots in Flink



[Carbone et. al. 2015] "Lightweight Asynchronous Snapshots for Distributed Dataflows", Tech. Report. http://arxiv.org/abs/1506.08603

Automatic partitioning and scaling

- 3 Types of Parallelization

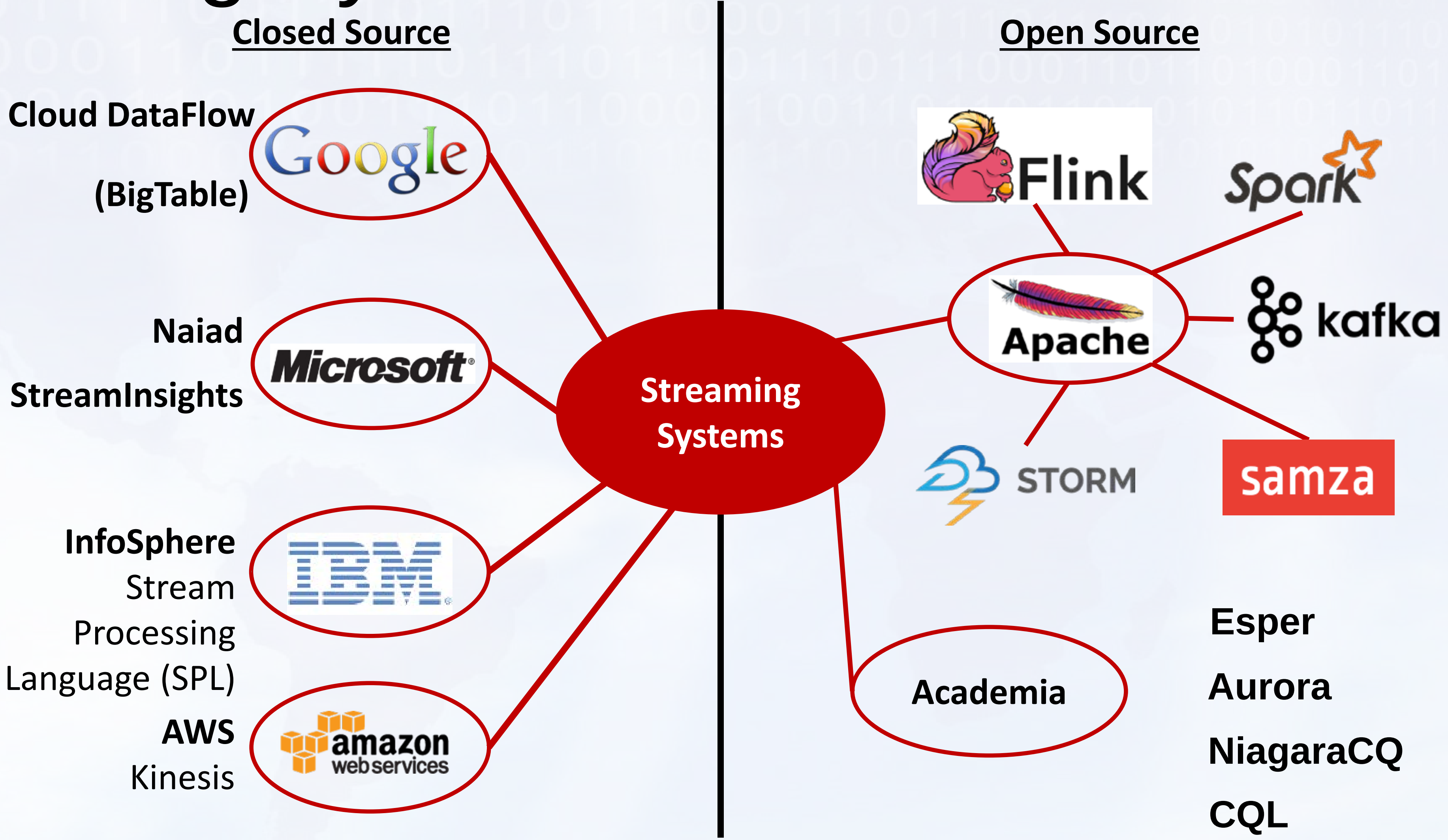


(a) Pipeline-parallel A || B. (b) Task-parallel D || E. (c) Data-parallel G || G.

- Big streaming systems should support all three

Big Data Streaming Systems

Streaming Systems Overview



Closed Source/Commercial Systems

Cloud DataFlow:
- Unified primitives for batch and stream processing
- Runs in Google's cloud only
- Open Source SDK (programs can run on other systems)
- Check out the Apache Beam Project! (http://beam.apache.org/)

BigTable:
- Not a real streaming solution
- Allows to feed streams as source into a google DB
- Data can be immediately queried

Naiad:
- Goals of Naiad:
  - High throughput (typical for batch processors)
  - Low latency (known from single system stream processors)
- Is able to process iterative data flows
- Can discretize windows only based on time

StreamInsights:
- Available through Microsoft's cloud
- Windows based on count-, time- and punctuation/snapshot
- Optimized for .NET framework applications

InfoSphere: Stream Processing Language (SPL)
- Well specified in several publications
- Can be deployed in customer clusters
- Own SQL-like query language enables many optimization means
- window discretization based on trigger- and eviction policies

Open Source Systems by Apache (1/2)

kafka

- Reliable handling of huge numbers of concurrent reads and writes
- Can be used as data-source / data-sink for Storm, Samza, Flink, Spark and many more systems
- Fault tolerant: Messages are persisted on disk and replicated within the cluster. Messages (reads and writes) can be repeated

STORM

- True streaming over distributed dataflow
- Low level API: Programmers have to specify the logic of each vertex in the flow graph
- Full understanding and hard coding of all used operators is required
- Enables very high throughput (single purpose programs with small overhead)

samza

- True streaming built on top of Apache Kafka and Hadoop YARN
- State is first class citizen
- Low level API

Open Source Systems by Apache (2/2)

Spark implements a batch execution engine

- The execution of a job graph is done in stages
- Operator outputs are materialized in memory (or disk) until the consuming operator is ready to consume the materialized data

Spark uses Discretized Streams (D-Streams)

- Streams are interpreted as a series of deterministic batch-processing jobs
- Micro batches have a fixed granularity
- All windows defined in queries must be multiples of this granularity

Flinks runtime is a native streaming engine

- Based on Nephele/PACTs
- Queries are compiled to a program in the form of an operator DAG
- Operator DAGs are compiled to job graphs
- Job graphs are generic streaming programs

Flink implements "true streaming"

- The whole job graph is deployed concurrently in the cluster
- Operators are long-running: Continuously consume input and produce output
- Output tuples are immediately forwarded to succeeding operators and are available for further processing (enables pipeline parallelism)

Further open source systems

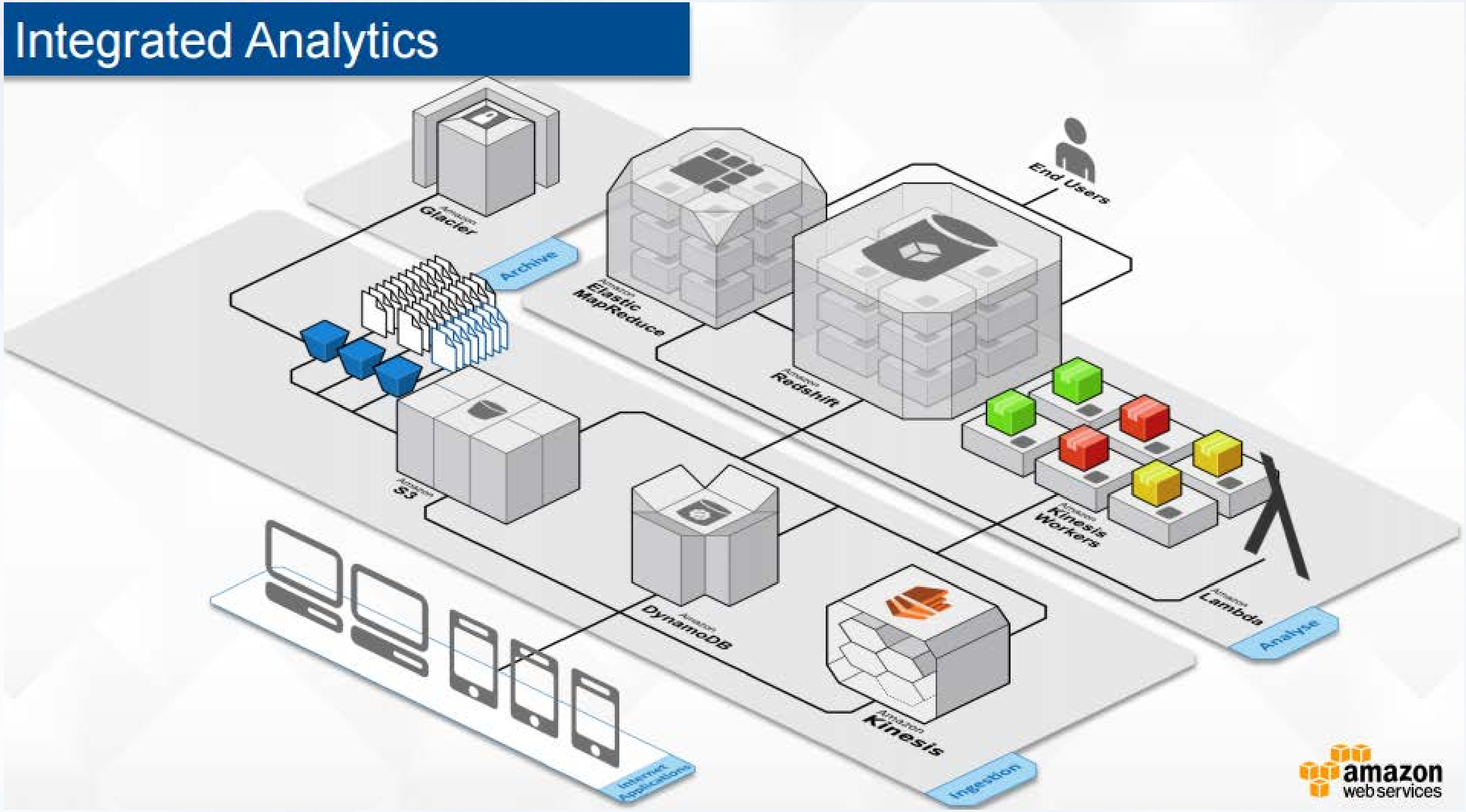
Esper
- Open source Complex Event Processing (CEP) engine
- Tightly coupled to Java: **Executable on J2EE application servers**
- Describing events in Plain Old Java Objects (POJOs)
- Time-based or count-based windows

Aurora
- First design and implementation that parallelizes stream computation including rich operation and windowing semantics
- Windows are always specified as ranges on some measure
- Was continued in Borealis Project

NiagaraCQ
- Focuses more on scalability than on the flexibility
- Provides various optimizations techniques to share common computation within and across queries
- Only time-based windows are possible

CQL
- Continuous query language
- Implemented by the STREAM DSMS at Stanford
- Captures a wide range of streaming application in an SQL-like query language

Cloud-Based Streaming Systems (example)



Storm, Spark Streaming, and Flink

Big Data Analytics Ecosystem

Applications & Languages

Hive | Cascading | Giraph

Mahout | Pig | Crunch

Data processing engines

MapReduce | Flink

Spark | Storm | Tez

App and resource management

Yarn | Mesos

Storage, streams

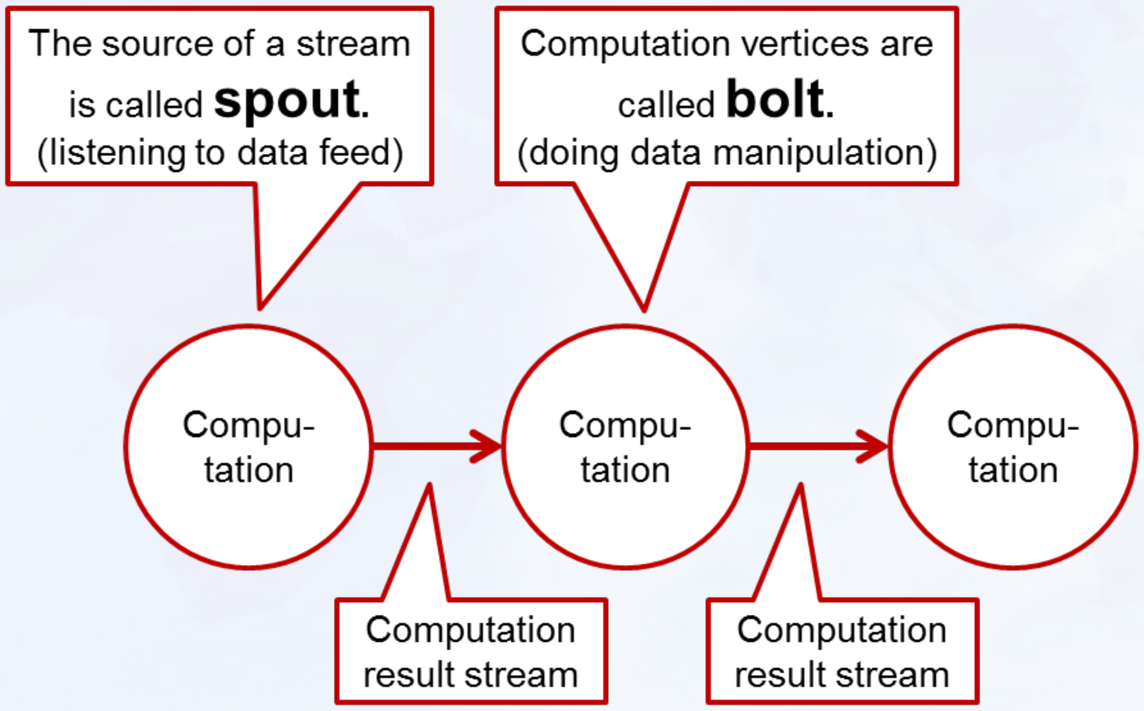
HDFS | HBase | Kafka | …

Apache Storm

Scalable Stream Processing Platform by Twitter

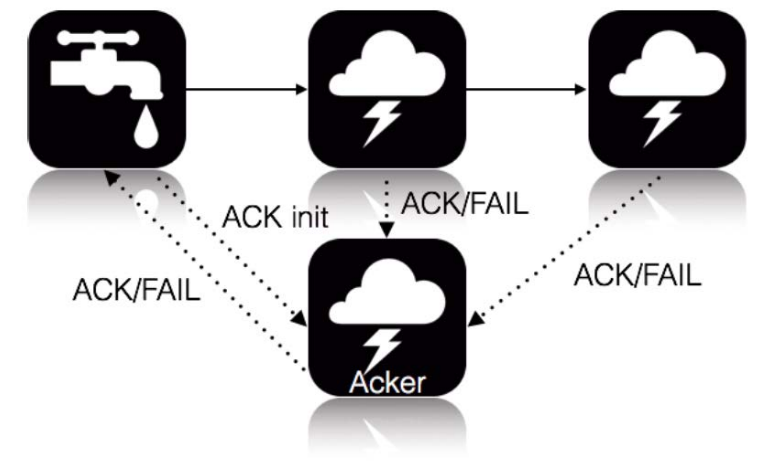
- Tuple wise computation
- Programs are represented in a **topology** graph
  - **vertices** are computations / data transformations
  - **edges** represent data **streams** between the computation nodes
  - streams consist of an unbounded sequence of data-items/tuples
- Low-level stream processing engine

Topology:



Storm‘s Fault Tolerance

- At least once guarantee via acknowledgments



- Acker logs progress of each tuple emitted by a spout

Storm Bolt Example

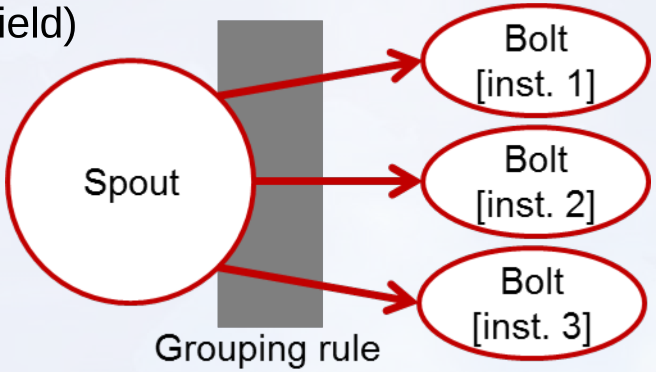
```
public class DoubleAndTripleBolt extends BaseRichBolt {
    private OutputCollectorBase _collector;

    @Override
    public void prepare(Map conf, TopologyContext context, OutputCollectorBase collector) {
        _collector = collector;
    }

    @Override
    public void execute(Tuple input) {
        int val = input.getInteger(0);
        _collector.emit(input, new Values(val*2, val*3));
        _collector.ack(input);
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("double", "triple"));
    }
}
```


Source: https://storm.apache.org/documentation/Tutorial.html

Building a Storm Topology

1) Use the TopologyBuilder class to connect spouts and bolts:
```
builder.setSpout("name",new MySpout());
builder.setBolt("name",new MyBolt());
```

2) Additionally, specify groupings to allow parallelization (shuffle, all, global, field)
```
builder.shuffleGrouping("BoltName");
```



3) Create topology using the factory method
```
StormTopology st=builder.createTopology();
```

4) Use LocalCluster class to test the topology
```
LocalCluster cluster=new LocalCluster();
cluster.submitTopology("name",new Config(),st);
```


Source: Allen et al., Storm Applied: Strategies for Real-Time Event Processing

Storm – Trident

- High-level abstraction built on top of Storm core:
  - operators like filter, join, groupBy, ...
- Stream-oriented API + UDFs
- Stateful, incremental processing
- Micro-Batch oriented (ordered & partitionable)
- Exactly-once semantics
- Trident topology compiled into spouts and bolt

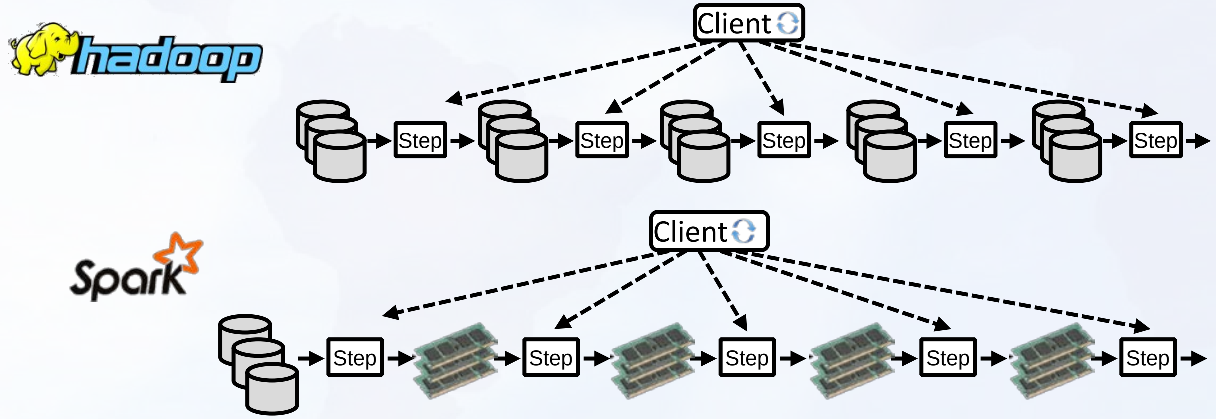
Storm – Heron

- New real-time streaming system based on Storm
- Introduced June 2015 by Twitter (SIGMOD)
- Fully compatible with Storm API
- Container-based implementation
- Back pressure mechanism
- Easy debugging of heron topologies through UI
- better performance than Storm (latency + throughput)

- No exactly once guarantee

Apache Spark

- In memory abstraction for big data processing
  - Resilient Distributed Data Sets
  - Fault-tolerance through lineage
  - Richt APIs for all kind of processing



Loop outside the system, in driver program

Iterative program looks like many independent jobs

Spark Job

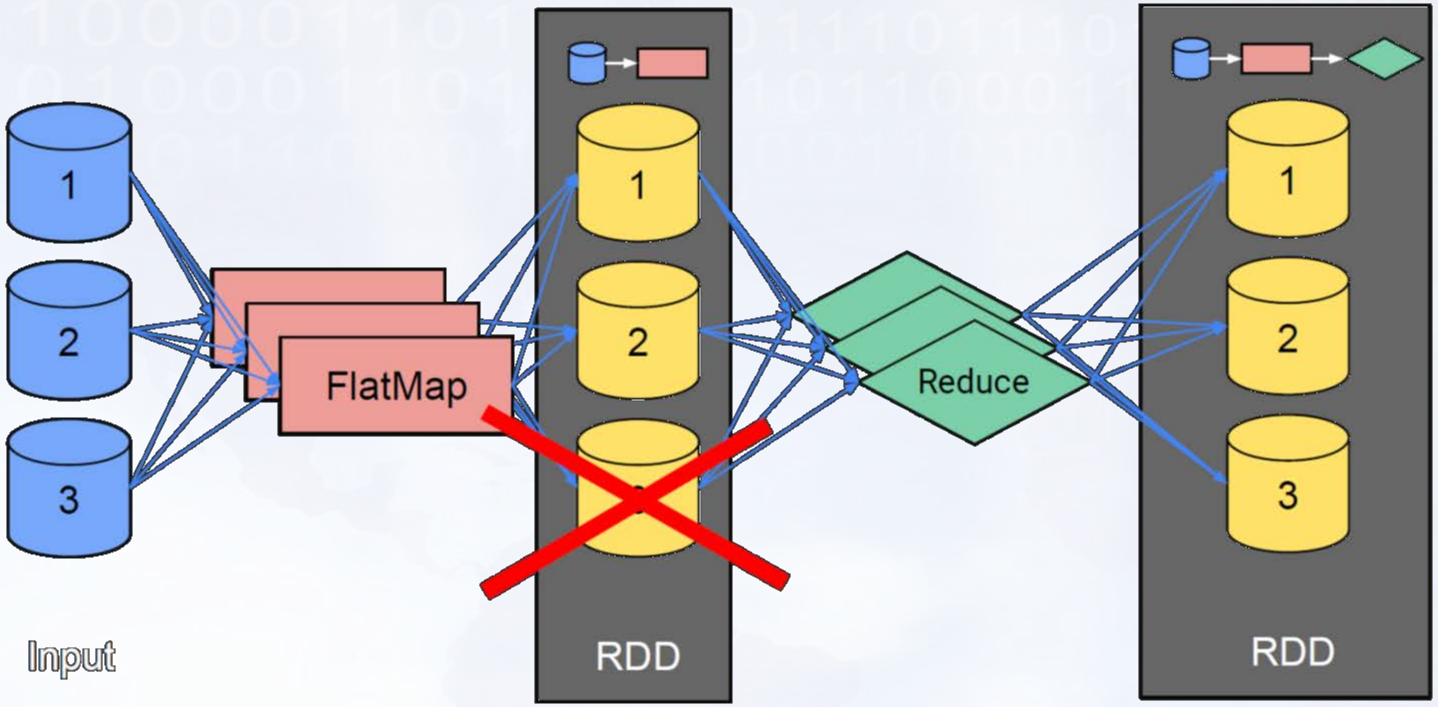
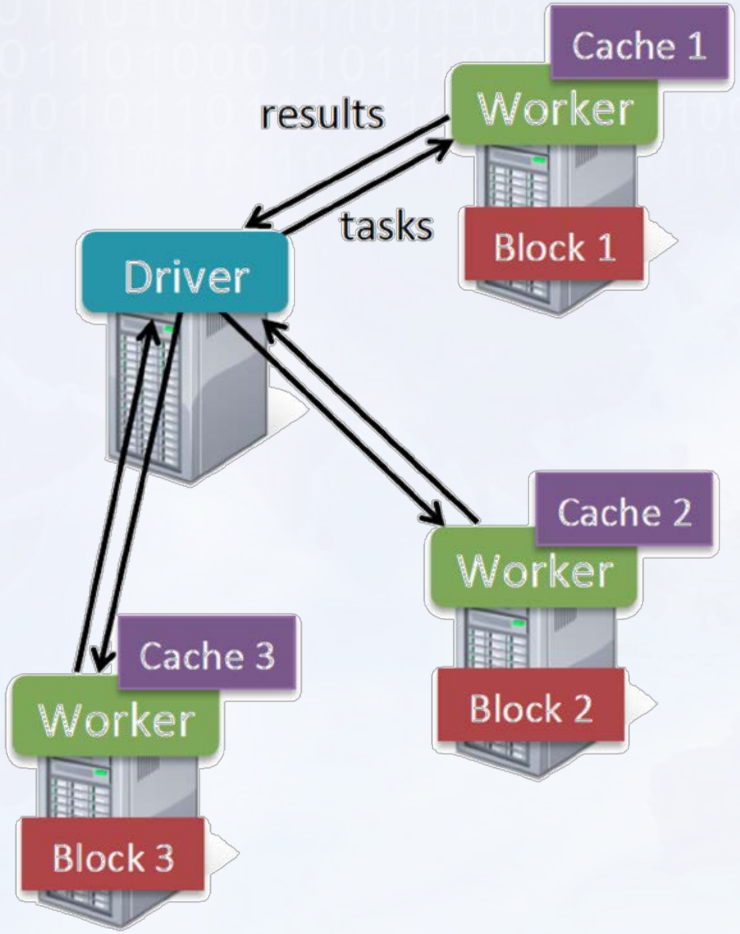


Image: Tyler Akidau



- Similar to MR, but much faster

Spark Streaming

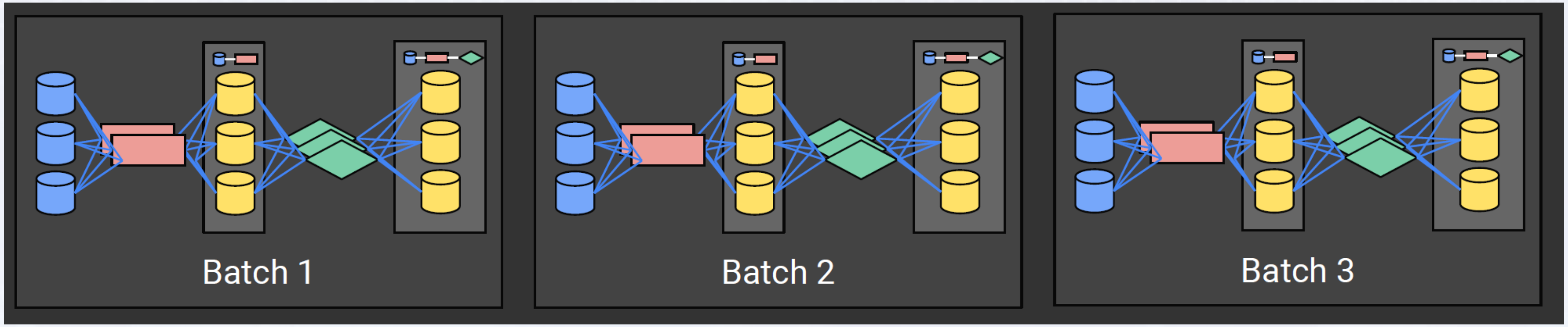


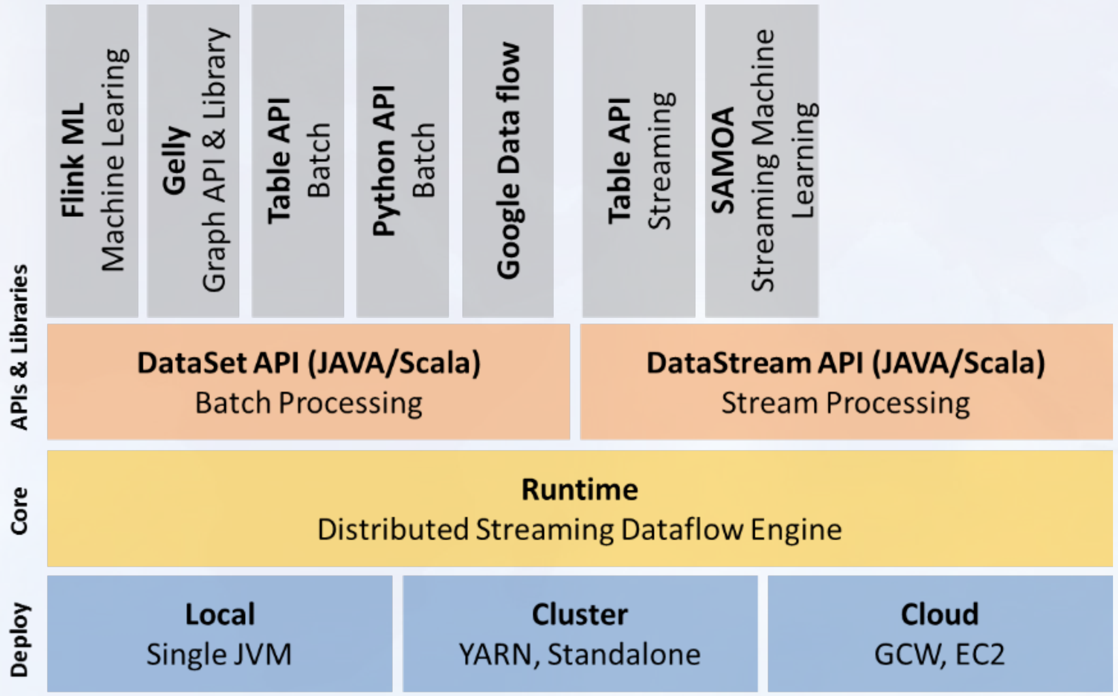
Image: Tyler Akidau

- Key abstraction: discretized streams (DStream)
  - micro-batch = series of RDDs
  - Stream computation = series of deterministic batch computation at a given time interval
- API very similar to Spark core (Java, Scala, Python)
  - (stateless) transformations on DStreams: map, filter, reduce, repartition, cogrop, ...
  - Stateful operators: time-based window operations, incremental aggregation, time-skewed joins
- Exactly-once semantics using checkpoints (asynchronous replication of state RDDs)
- No event time windows

Apache Flink

Apache Flink is an open source platform for scalable batch and stream data processing.

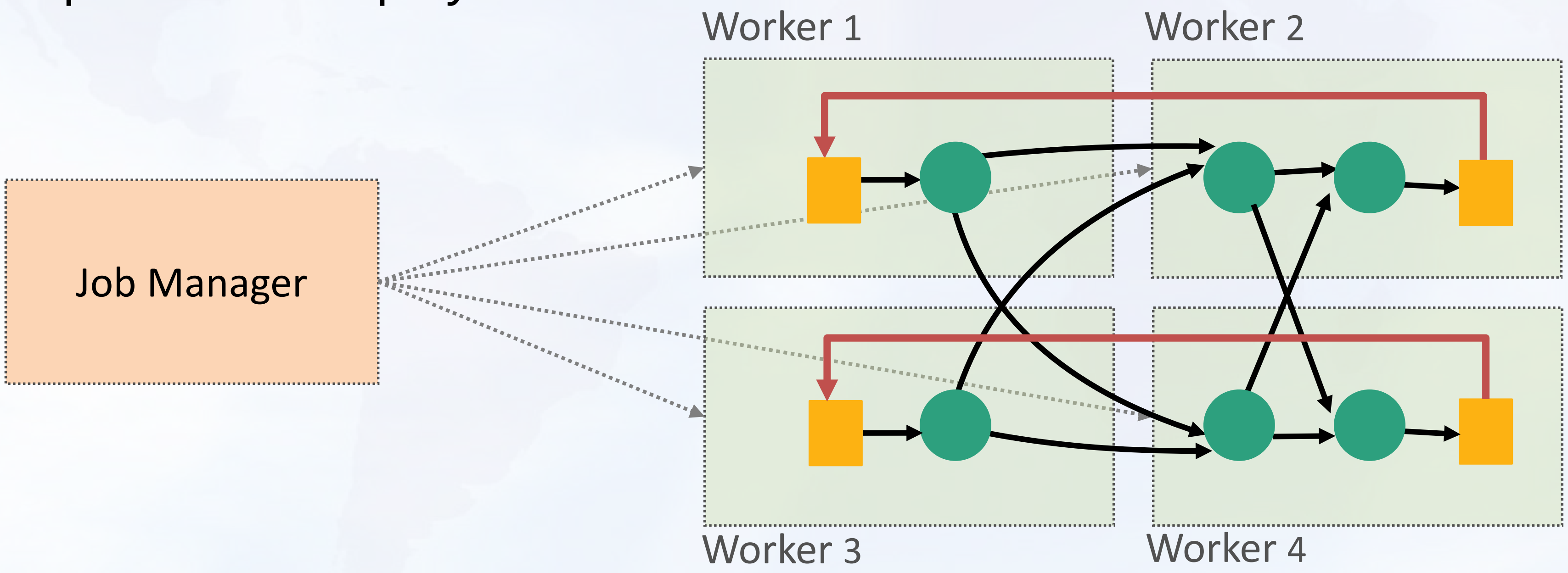
- The core of Flink is a distributed streaming dataflow engine.
  - Executing dataflows in parallel on clusters
  - Providing a reliable foundation for various workloads
- **DataSet** and **DataStream** programming abstractions are the foundation for user programs and higher layers



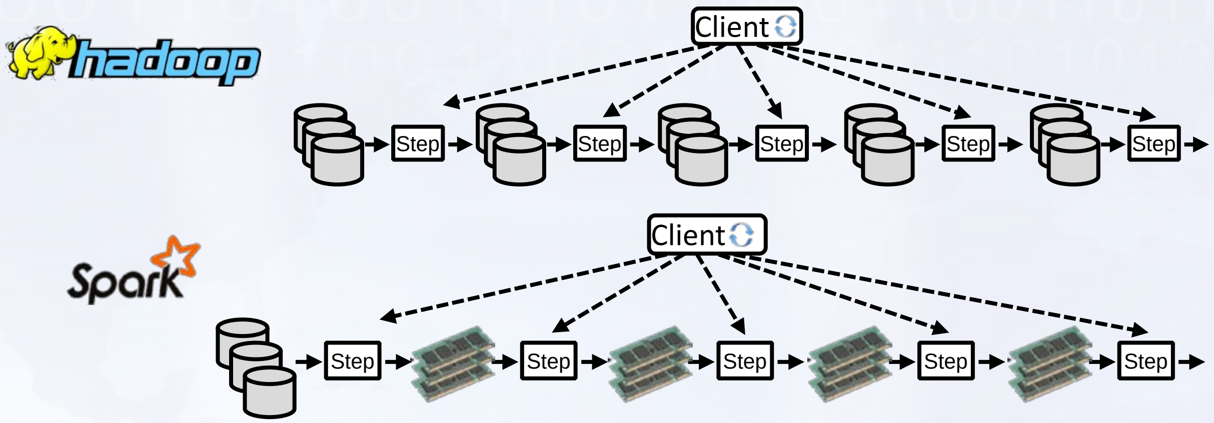
http://flink.apache.org

Architecture

- Hybrid MapReduce and MPP database runtime
- Pipelined/Streaming engine
  - Complete DAG deployed

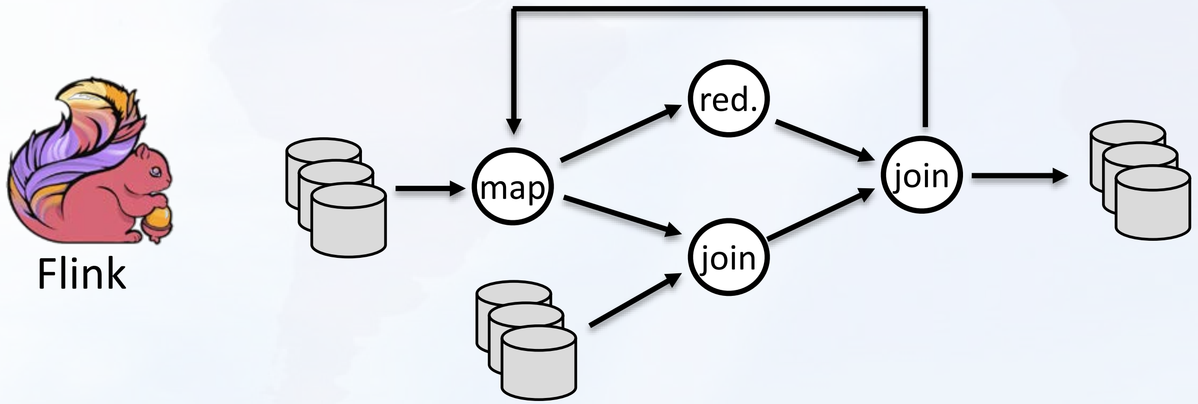


Built-in vs. driver-based looping



Loop outside the system, in driver program

Iterative program looks like many independent jobs



Dataflows with feedback edges

System is iteration-aware, can optimize the job

Sneak peak: Two of Flink's APIs

```
case class Word (word: String, frequency: Int)
```

DataSet API (batch):

```
val lines: DataSet[String] = env.readTextFile(...)

lines.flatMap {line => line.split(" ")
                              .map(word => Word(word,1))}
     .groupBy("word").sum("frequency")
     .print()
```

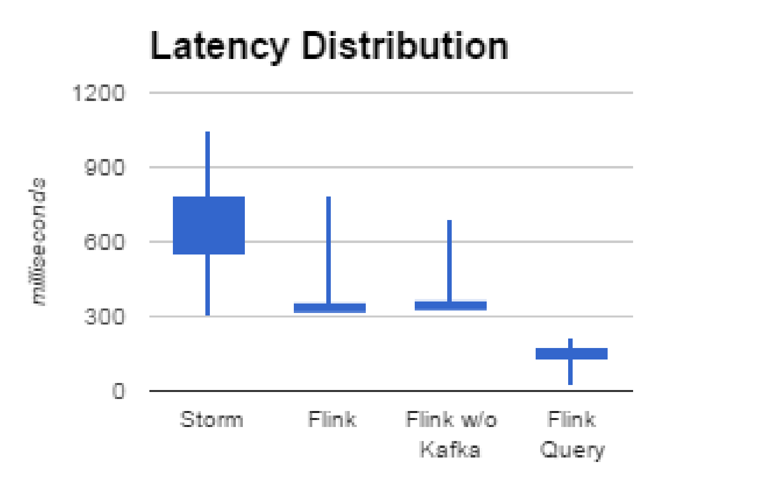
DataStream API (streaming):

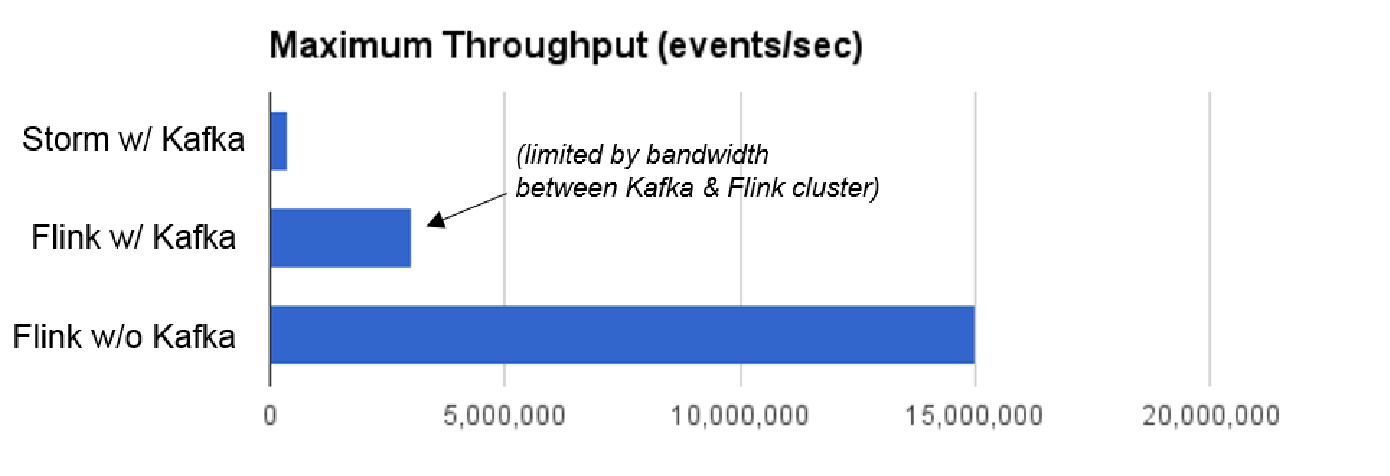
```
val lines: DataStream[String] = env.fromSocketStream(...)

lines.flatMap {line => line.split(" ")
                              .map(word => Word(word,1))}
    .keyBy("word")
    .window(Time.of(5,SECONDS)).every(Time.of(1,SECONDS))
    .sum("frequency")
    .print()
```

Some Benchmark Results

Initially performed by Yahoo! Engineering, Dec 16, 2015,

[..]Storm 0.10.0, 0.11.0-SNAPSHOT and Flink 0.10.1 show sub- second latencies at relatively high throughputs[..]. Spark streaming 1.5.1 supports high throughputs, but at a relatively higher latency.





http://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at
https://data-artisans.com/extending-the-yahoo-streaming-benchmark/

Processing Time vs Event Time

DEMO - STREAMING

„Inspired“ by
https://github.com/dataArtisans/oscon

Stream Optimizations

Based on Hirzel et. al: A Catalog of Stream Processing Optimizations, ACM Comp. Surveys. 46(4), 2014.

Overview

- 11 Optimizations (numbered from 2 to 12 ☺)

Section	Optimization	Graph	Semantics	Dynamic
2.	Operator reordering	changed	unchanged	(depends)
3.	Redundancy elimination	changed	unchanged	(depends)
4.	Operator separation	changed	unchanged	static
5.	Fusion	changed	unchanged	(depends)
6.	Fission	changed	(depends)	(depends)
7.	Placement	unchanged	unchanged	(depends)
8.	Load balancing	unchanged	unchanged	(depends)
9.	State sharing	unchanged	unchanged	static
10.	Batching	unchanged	unchanged	(depends)
11.	Algorithm selection	unchanged	(depends)	(depends)
12.	Load shedding	unchanged	changed	dynamic

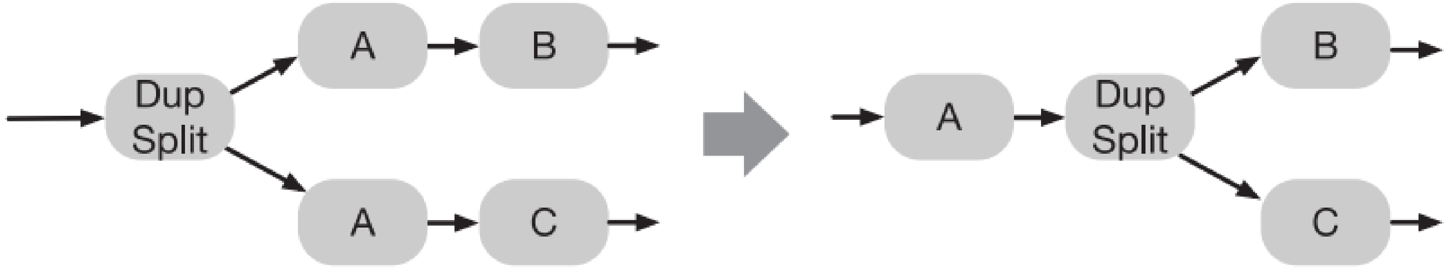
Reordering and Elimination

2. OPERATOR REORDERING (A.K.A. HOISTING, SINKING, ROTATION, PUSH-DOWN)

Move more selective operators upstream to filter data early.

3. REDUNDANCY ELIMINATION (A.K.A. SUBGRAPH SHARING, MULTIQUERY OPTIMIZATION)

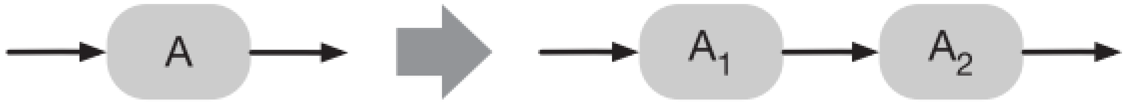
Eliminate redundant computations.



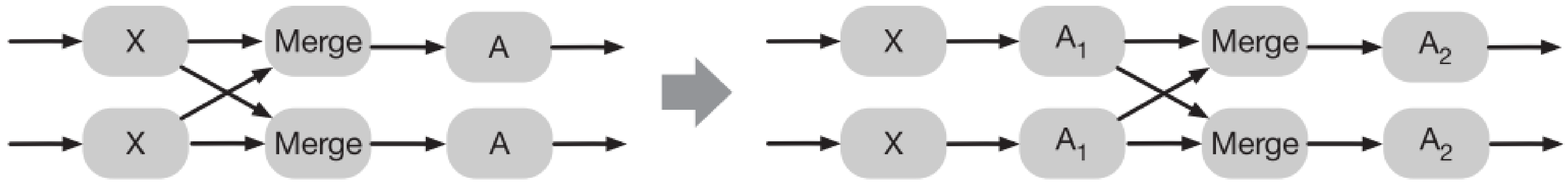
Operator Separation

4. OPERATOR SEPARATION (A.K.A. DECOUPLED SOFTWARE PIPELINING)

Separate operators into smaller computational steps.



Operator separation is profitable if it enables other optimizations such as operator reordering or fission, or if the resulting pipeline parallelism pays off when running on multiple cores.



Fusion

5. FUSION (A.K.A. SUPERBOX SCHEDULING)

Avoid the overhead of data serialization and transport.

Flink

In Apache Flink (and many other applications) we call this

chaining

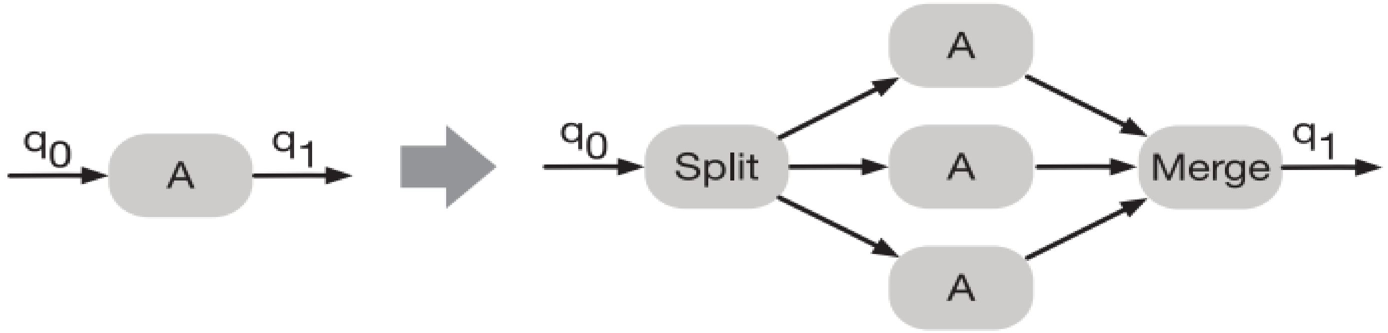
Goal: Reduce communication costs

Method: Shared memory among operators instead of network communication

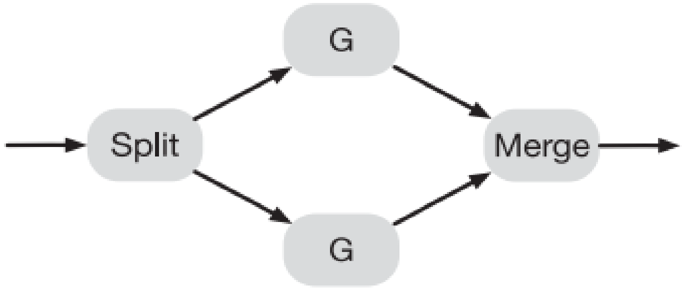
Fission

6. FISSION (A.K.A. PARTITIONING, DATA PARALLELISM, REPLICATION)

Parallelize computations.



Directly maps to data parallelism:

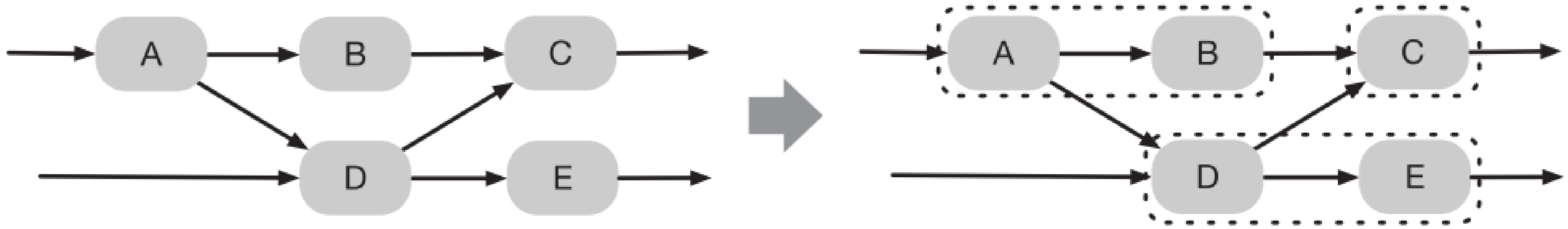


(c) Data-parallel G || G.

Placement

7. PLACEMENT (A.K.A. LAYOUT)

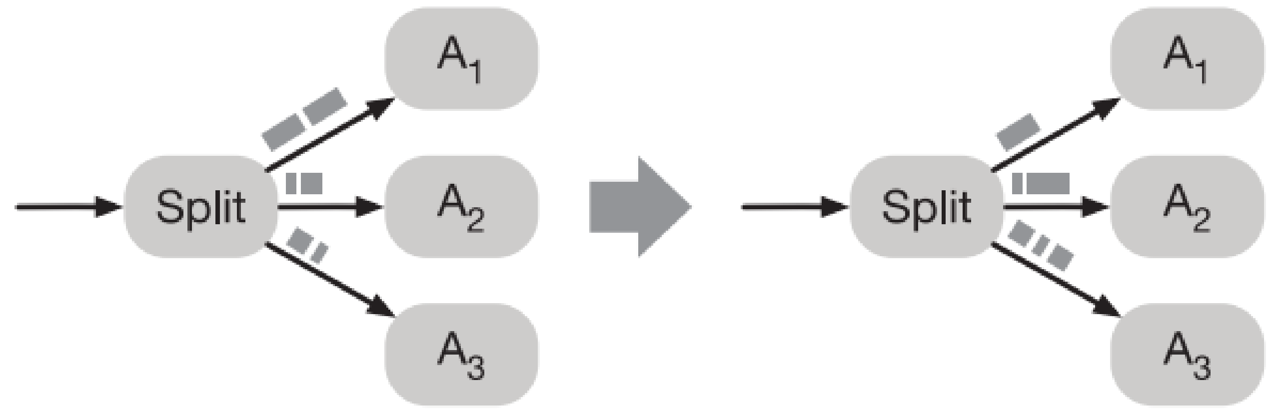
Assign operators to hosts and cores.



Flink — Assigning Operators to slots

MapReduce + HDFS — Co-locating Data and Computations

Load Balancing

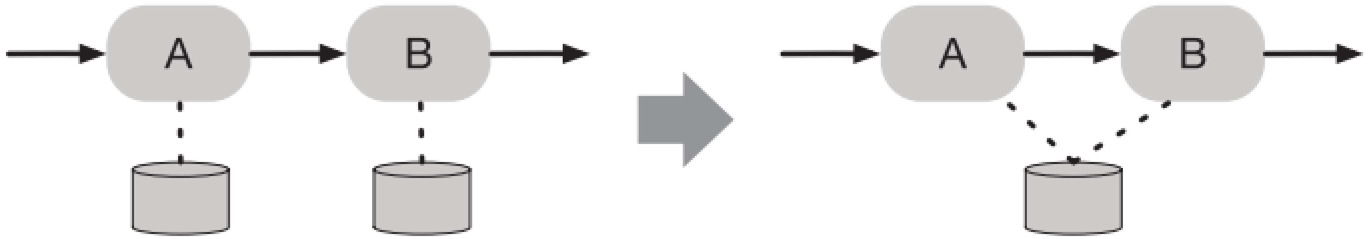

8. LOAD BALANCING

Distribute workload evenly across resources.

State Sharing

9. STATE SHARING (A.K.A. SYNOPSIS SHARING, DOUBLE-BUFFERING)

Optimize for space by avoiding unnecessary copies of data.



Distributed File Systems

A single storage layer for the whole cluster

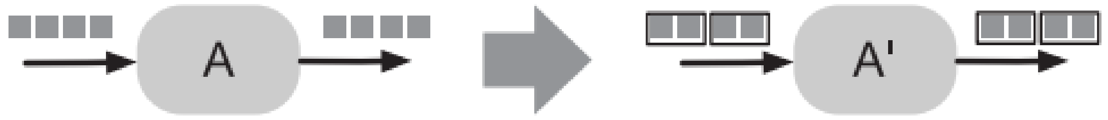
Chaining again…

Share memory among several operators instead of copying the data

Batching

10. BATCHING (A.K.A. TRAIN SCHEDULING, EXECUTION SCALING)

Process multiple data items in a single batch.



Flink — "Under the hood" batch wise network traffic (buffering)

Spark — **D-Streams***: All the stream processing is done in micro-batches

* Zaharia, Matei, et al. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In: *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association, 2012. S. 10-10. https://www.usenix.org/system/files/conference/hotcloud12/hotcloud12-final28.pdf

Algorithm Selection & Load Shedding

11. ALGORITHM SELECTION (A.K.A. TRANSLATION TO PHYSICAL QUERY PLAN)

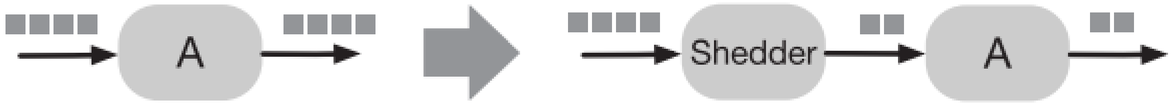
Use a faster algorithm for implementing an operator.

Flink

The optimizer selects the (hopefully) optimal join implementation

12. LOAD SHEDDING (A.K.A. ADMISSION CONTROL, GRACEFUL DEGRADATION)

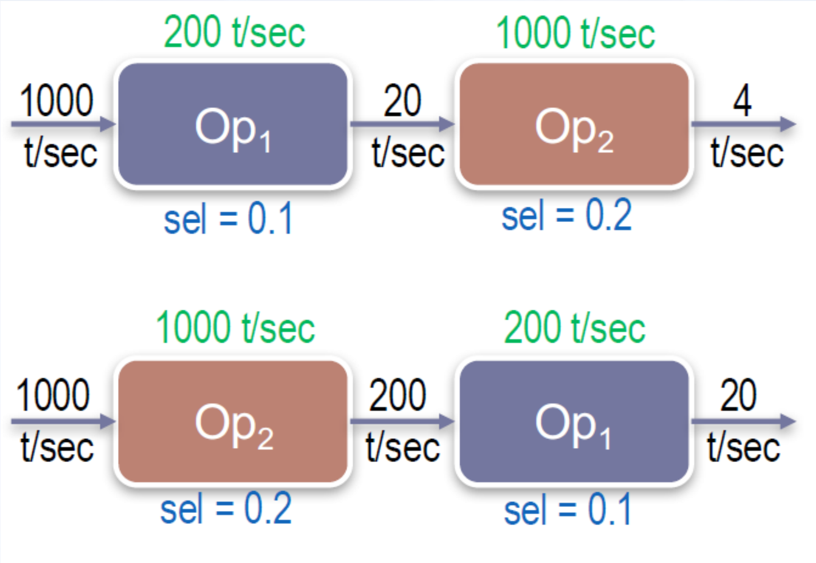
Degrade gracefully when overloaded.



Cost Model

- Traditional cost-based query optimization is based on cardinality estimation
  ➟ inadequate for unbounded streams
- Possible solution: rate-based cost estimation
  – (Viglas et al.: Rate-based query optimization for streaming information sources, SIGMOD 2002)

$$\textit{output rate} = \frac{\#\textit{outputs transmitted}}{\textit{time for transmission}}$$



- Challenges:
  – Fluctuating streams
  – Data-parallel processing

Slide by Kai-Uwe Sattler

Conclusion

Introduction to Streams

- Stream Processing 101
- How to do real streaming

Stream Processing Systems

- Ingredients of a stream processing system
- Storm, Spark, Flink
- Continuously evolving

Stream Processing Optimizations

- How to optimize

Thank You

Contact:

Tilmann Rabl
rabl@tu-berlin.de

We are hiring!

Further Reading

Historical papers on STREAM, Aurora, TelegraphCQ, Borealis, CQL, ...

- Papers and blogs on Storm, Heron, Flink, Spark Streaming, ...
- Alexandrov, Alexander, et al. The Stratosphere platform for big data analytics. *The VLDB Journal-The International Journal on Very Large Data Bases*, 2014, 23. Jg., Nr. 6, S. 939-964.
- Zaharia, Matei, et al. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In: *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association, 2012. S. 10-10.
- Murray, Derek G., et al. Naiad: a timely dataflow system. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013. S. 439-455.

Windows & Semantics

- Ghanem et al.: Incremental Evaluation of Sliding-Window Queries over Data Streams, TKDE 19(1), 2007
- Tucker et al.: Exploiting Punctuation Semantics in Continuous Data Streams, TKDE 15(3), 2003
- Krämer et al.: Sematics and Implementation of Continuous Sliding Window Queries over Data Streams, TODS 34(1), 2009
- Botan et al.: SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems, VLDB 2010

CEP:

- Wu et al.: High-Performance Complex Event Processing over Streams, SIGMOD 2006
- Schultz-Moeller et al.: Distributed Complex Event Processing with Query Rewriting, DEBS 2009

Fault Tolerance:

- Hwang et al.: High-availability algorithms for distributed stream processing, ICDE 2005
- Zaharia et al.: Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. HotCloud, 2012.
- Fernandez et al.: Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management, SIGMOD 2013

Partitioning & Optimization:

- Hirzel et al.: A Catalog of Stream Processing Optimizations, ACM Comp. Surveys 46(4), 2014.
- Gedik et al.: Elastic Scaling for Data Streams, TPDS 25(6), 2014.
- Viglas et al.: Rate-Based Query Optimization for Streaming Information Sources, SIGMOD 2002