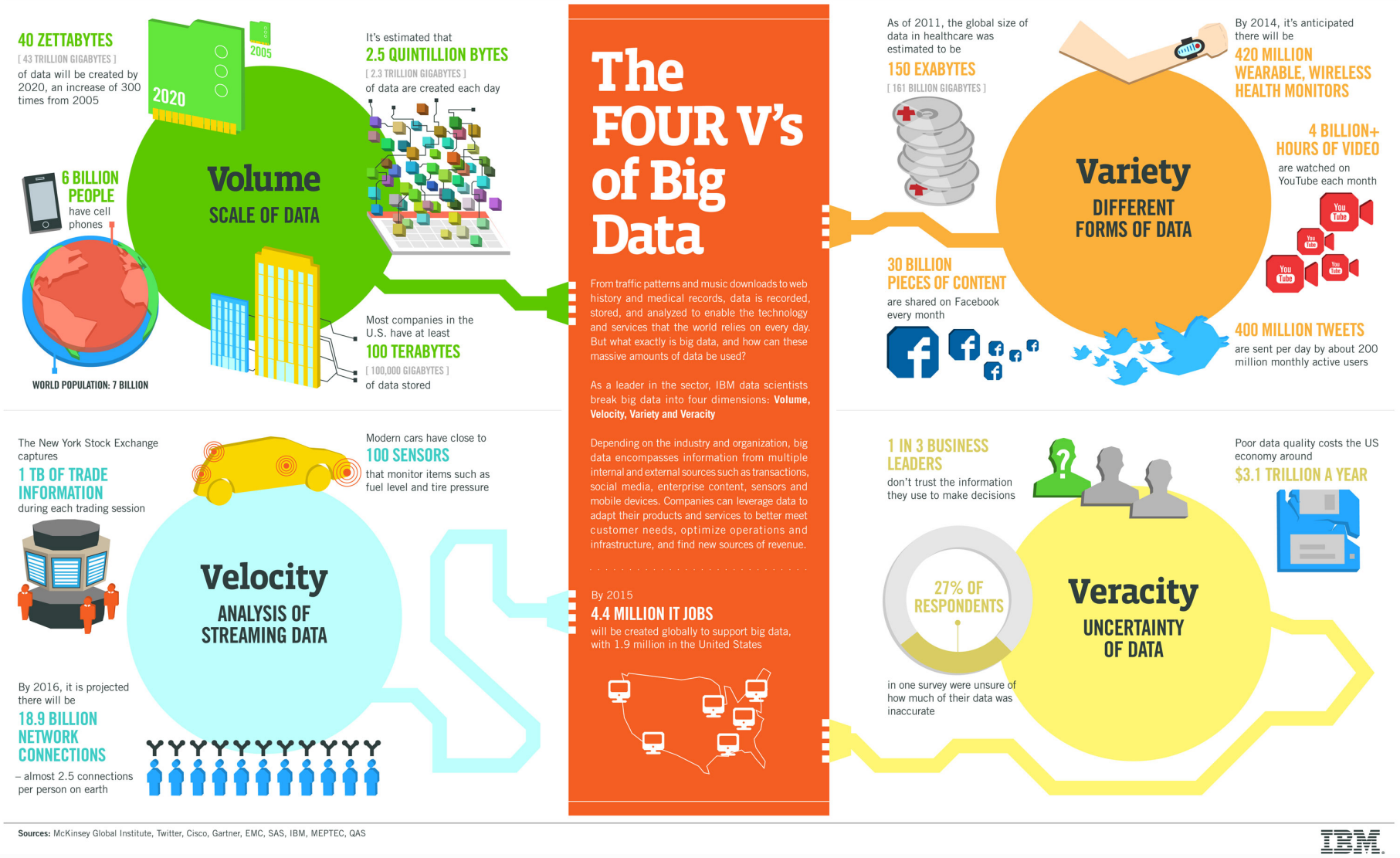
INFO-H515: BIG DATA: DISTRIBUTED MANAGEMENT

Lecture 3: Distributed Stream Processing

Dimitris Sacharidis

REMEMBER THE V'S?

THE CHALLENGES OF BIG DATA



Source: http://www.ibmbigdatahub.com/sites/default/files/infographic_file/

THE NEED TO DEAL WITH VELOCITY

Definition

Velocity refers to the speed at which data is being produced, captured, or refreshed.

THE NEED TO DEAL WITH VELOCITY

Definition

Velocity refers to the speed at which data is being produced, captured, or refreshed.

Data is not just Big, it is also Fast ...

- Facebook users upload $>$ 900 million photos/day (= 652,000 photos/minute)
- Twitter processes 500 million tweets/day (= 350,000 tweets/minute)
- The NYSE generates 1 TB of data per trading session (= 2.4 GB/min)

THE NEED TO DEAL WITH VELOCITY

Definition

Velocity refers to the speed at which data is being produced, captured, or refreshed.

Data is not just Big, it is also Fast ...

- Facebook users upload > 900 million photos/day (= 652,000 photos/minute)
- Twitter processes 500 million tweets/day (= 350,000 tweets/minute)
- The NYSE generates 1 TB of data per trading session (= 2.4 GB/min)

... and requires (near) real-time analysis

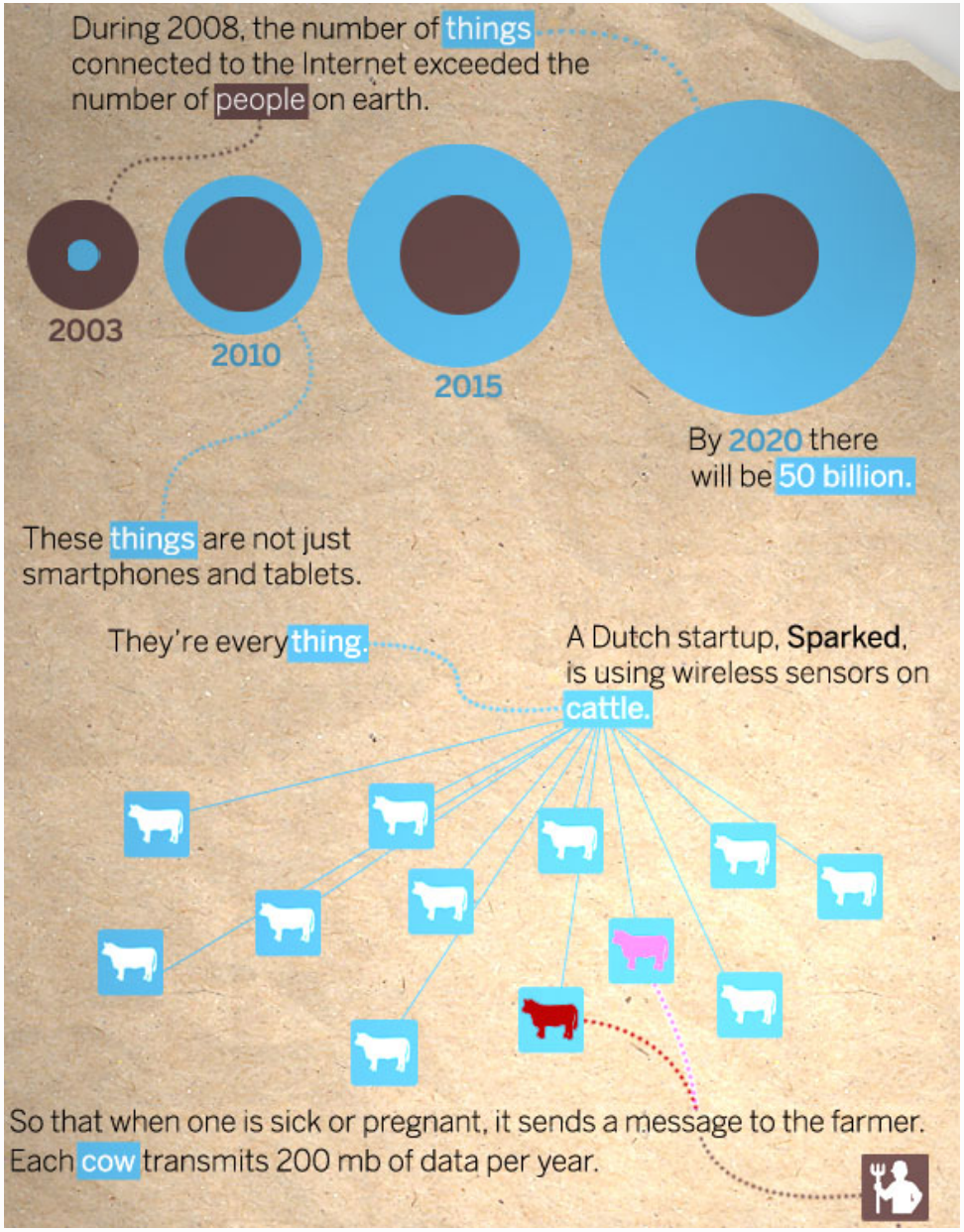
- In electronic trading, a 1 millisecond advantage can be worth $100 million to a major brokerage firm.
- In retail, out-of-stock detection is preferable in seconds or minutes rather than days or weeks.

THE INTERNET OF THINGS

There are more devices connected to the internet than people on earth.

Sensors are/will be everywhere.

How can systems ingest this continuous data stream?



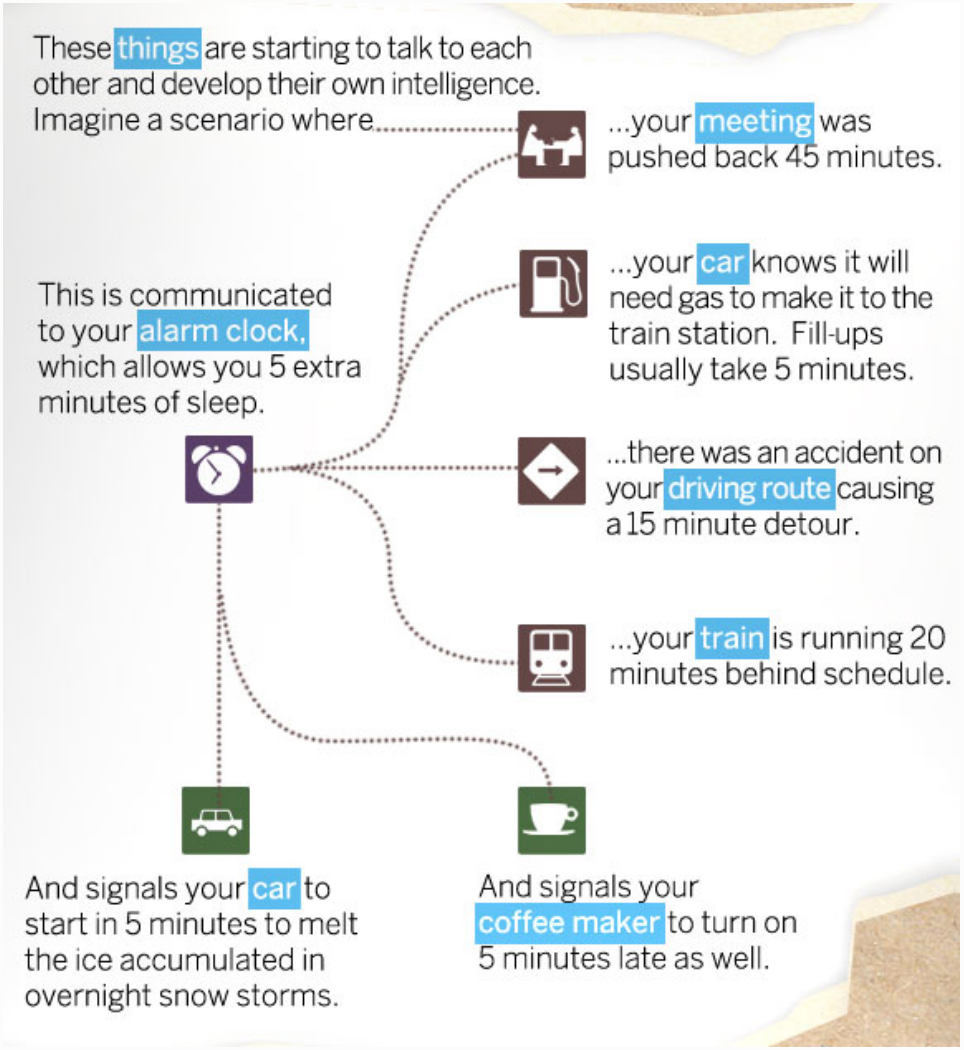
Source: `https://blogs.cisco.com/diversity/the-internet-of-things-infographic`

THE INTERNET OF THINGS

There are more devices connected to the internet than people on earth.

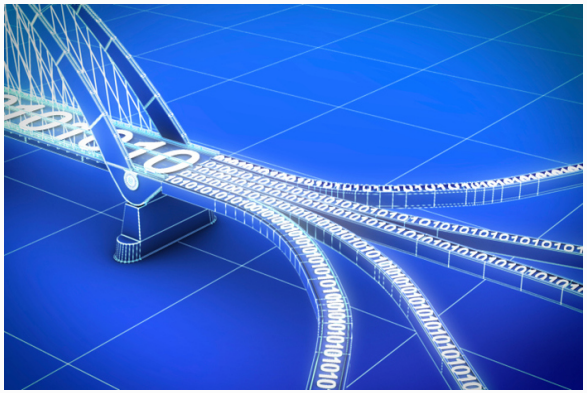
Sensors are/will be everywhere.

How can systems ingest this continuous data stream?



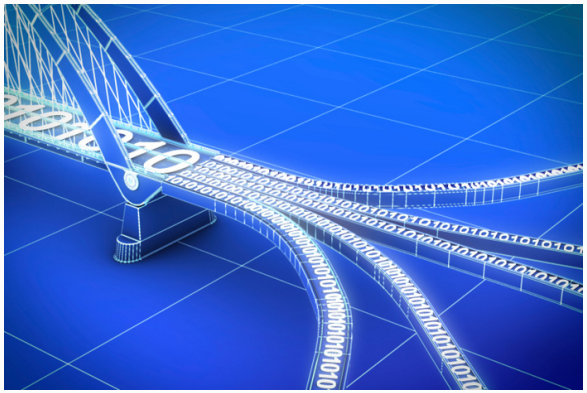
Source: `https://blogs.cisco.com/diversity/the-internet-of-things-infographic`

LECTURE OUTLINE



Even if an application does not require real-time analysis, use of data streaming is widespread because of the λ-architecture.

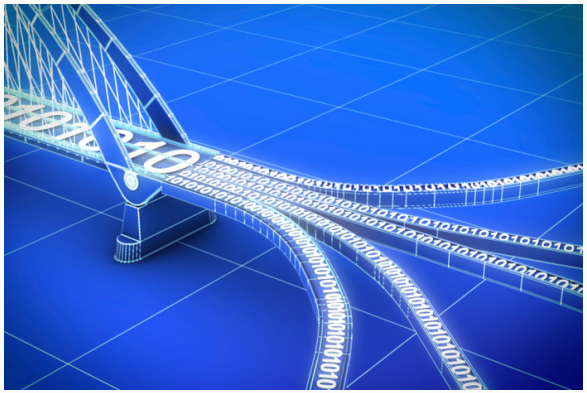
LECTURE OUTLINE



Even if an application does not require real-time analysis, use of data streaming is widespread because of the λ-architecture.

Big Data Software Frameworks that allow analysis of high-velocity data are called data streaming frameworks. We will discuss two, with fundamentally different architectures:

LECTURE OUTLINE



Even if an application does not require real-time analysis, use of data streaming is widespread because of the λ-architecture.

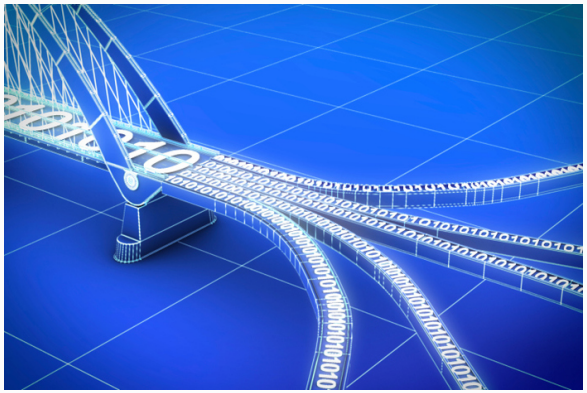
Big Data Software Frameworks that allow analysis of high-velocity data are called data streaming frameworks. We will discuss two, with fundamentally different architectures:

APACHE STORM™

Twitter Storm/Heron

Tuple-at-a-time

LECTURE OUTLINE



Even if an application does not require real-time analysis, use of data streaming is widespread because of the λ-architecture.

Big Data Software Frameworks that allow analysis of high-velocity data are called data streaming frameworks. We will discuss two, with fundamentally different architectures:

APACHE STORM™

Twitter Storm/Heron

Tuple-at-a-time

Spark Streaming

Spark Streaming

Mini-batching

LECTURE OUTLINE

THE λ-ARCHITECTURE

A data system is a system that manages the storage and querying of data with a lifetime measured in years encompassing every version of the application to ever exist, every hardware failure, and every human mistake every made.

—Nathan Marz

A data system is a system that manages the storage and querying of data with a lifetime measured in years encompassing every version of the application to ever exist, every hardware failure, and every human mistake every made.

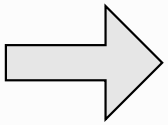
—Nathan Marz

What is hence a good way to architect a (big) data system?

THE PROBLEM OF MUTABLE DATA (1/2)

Traditionally, we mutate the database to store only most recent data.

Person	Location
Sally	Philadelphia
Bob	Chicago



Person	Location
Sally	New York
Bob	Chicago

Sally moves to New York.

THE PROBLEM OF MUTABLE DATA (2/2)

Mutable data is corruptable.

- Bugs will be deployed to production software over the lifetime of a data system.
- Humans can err, therefore operational mistakes will be made at some point.
- You must design to safeguard against data corruption.

THE PROBLEM OF MUTABLE DATA (2/2)

Mutable data is corruptable.

- Bugs will be deployed to production software over the lifetime of a data system.
- Humans can err, therefore operational mistakes will be made at some point.
- You must design to safeguard against data corruption.

Examples of errors.

- Deploy a software bug that increments counters by two instead of one
- Accidentally delete data from a database
- Incorrectly modify a data item (Sally did not move to New York, Bob did!)

THE PROBLEM OF MUTABLE DATA (2/2)

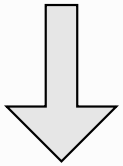
Mutable data is corruptable.

- Bugs will be deployed to production software over the lifetime of a data system.
- Humans can err, therefore operational mistakes will be made at some point.
- You must design to safeguard against data corruption.

Key observation: as long as an error does not lose or corrupt good data, we can fix what went wrong.

IMMUTABILITY

Person	Location
Sally	Philadelphia
Bob	Chicago



Person	Location
Sally	New York
Bob	Chicago

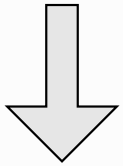
Sally moves to New York

Mutable data is corruptable. Therefore, keep all (master) data immutable.

- An immutable data system captures a historical record of events
- Each event happens at a particular time, and remains true forever.
- You can only append new events; never delete or modify existing event records.
- Filter on the timestamp to see what is true at a particular moment in time.

IMMUTABILITY

Person	Location	Time
Sally	Philadelphia	1318358
Bob	Chicago	1237921



Person	Location	Time
Sally	Philadelphia	1318358
Bob	Chicago	1237921
Sally	New York	1338812

Sally moves to New York

Mutable data is corruptable. Therefore, keep all (master) data immutable.

- An immutable data system captures a historical record of events
- Each event happens at a particular time, and remains true forever.
- You can only append new events; never delete or modify existing event records.
- Filter on the timestamp to see what is true at a particular moment in time.

Immutability makes sense:

Immutability makes sense:

A data system answers is supposed to answer questions based on data that was acquired in the past. So why modify acquired data?

Immutability makes sense:

A data system answers is supposed to answer questions based on data that was acquired in the past. So why modify acquired data?

Big Data technologies allow to store all raw data so that it can be used later to gain new insights and create new data-driven products, which we haven't yet thought of!

DESIRED PROPERTIES OF A BIG DATA SYSTEM

- Robust and fault tolerance (under both machine and human failures)
- Scalable: maintain performance under increasing load by adding resources.
- Answer pre-defined queries with low latency
- Support ad-hoc querying
- Low-latency updates
- Extensible/general

QUERY = FUNCTION(ALL DATA)

Location Information Database

- How many people live in a particular location?
- Where does Sally Live?
- What is the most popular location in summer?

Web Analytics Database

- How many pageviews on September 2nd?
- How many unique visitors over time?

QUERY = FUNCTION(ALL DATA)

Location Information Database

- How many people live in a particular location?
- Where does Sally Live?
- What is the most popular location in summer?

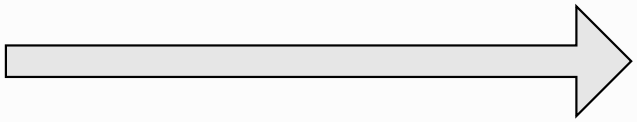
Web Analytics Database

- How many pageviews on September 2nd?
- How many unique visitors over time?

- A query hence transforms our immutable raw data into useful information.
- Problem: All data = petabyte scale. How do we get small latency?

TOWARDS THE λ-ARCHITECTURE

All Immutable Data



Query (Application)

User	URL	Time
192.168.2.1	foo.com/blog	1318358
192.168.4.8	foo.com/blog	1318458
192.168.4.8	foo.com/about	1318898
192.250.6.9	oof.com/item	1318898
192.250.6.9	foo.com/blog	1818123
...	...	...

How many unique people visited this domain each hour for the past three days?

Problem: All data = petabyte scale. How do we get small latency?

- Computing query answers on the fly has high latency (hours!)

TOWARDS THE λ-ARCHITECTURE

User	URL	Time
192.168.2.1	foo.com/blog	1318358
192.168.4.8	foo.com/blog	1318458
192.168.4.8	foo.com/about	1318898
192.250.6.9	oof.com/item	1318898
192.250.6.9	foo.com/blog	1818123
...	...	...

URL	Hour	#Unique
foo.com/blog	1	203
foo.com/blog	2	402
foo.com/blog	3	130
foo.com/blog	4	239
foo.com/blog	5	391
...	...	...

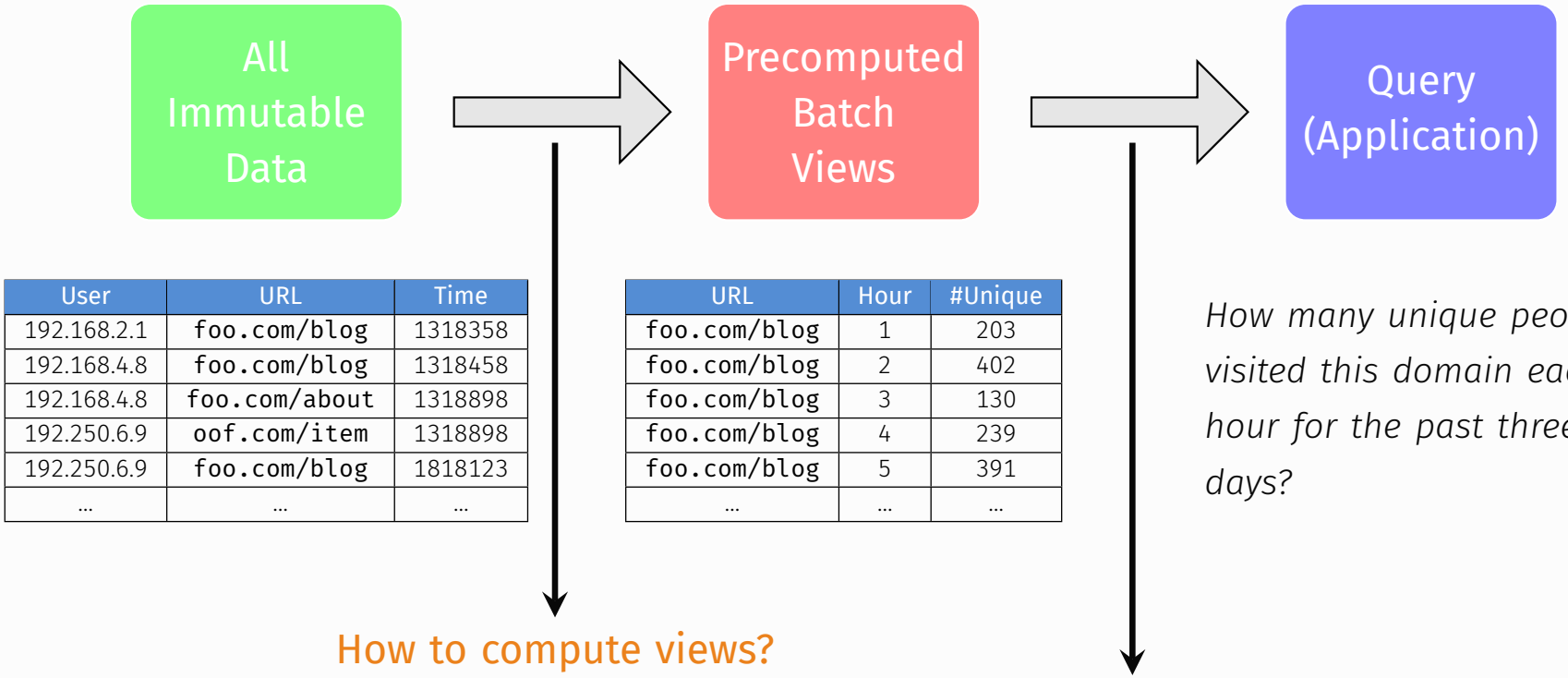
How many unique people visited this domain each hour for the past three days?

Problem: All data = petabyte scale. How do we get small latency?

- Computing query answers on the fly has high latency (hours!)

Solution: Precompute views from which the answers to pre-defined queries can be read/computed with low latency.

TOWARDS THE λ-ARCHITECTURE



User	URL	Time
192.168.2.1	foo.com/blog	1318358
192.168.4.8	foo.com/blog	1318458
192.168.4.8	foo.com/about	1318898
192.250.6.9	oof.com/item	1318898
192.250.6.9	foo.com/blog	1818123
...	...	...

URL	Hour	#Unique
foo.com/blog	1	203
foo.com/blog	2	402
foo.com/blog	3	130
foo.com/blog	4	239
foo.com/blog	5	391
...	...	...

How many unique people visited this domain each hour for the past three days?

TOWARDS THE λ-ARCHITECTURE

User	URL	Time
192.168.2.1	foo.com/blog	1318358
192.168.4.8	foo.com/blog	1318458
192.168.4.8	foo.com/about	1318898
192.250.6.9	oof.com/item	1318898
192.250.6.9	foo.com/blog	1818123
...	...	...

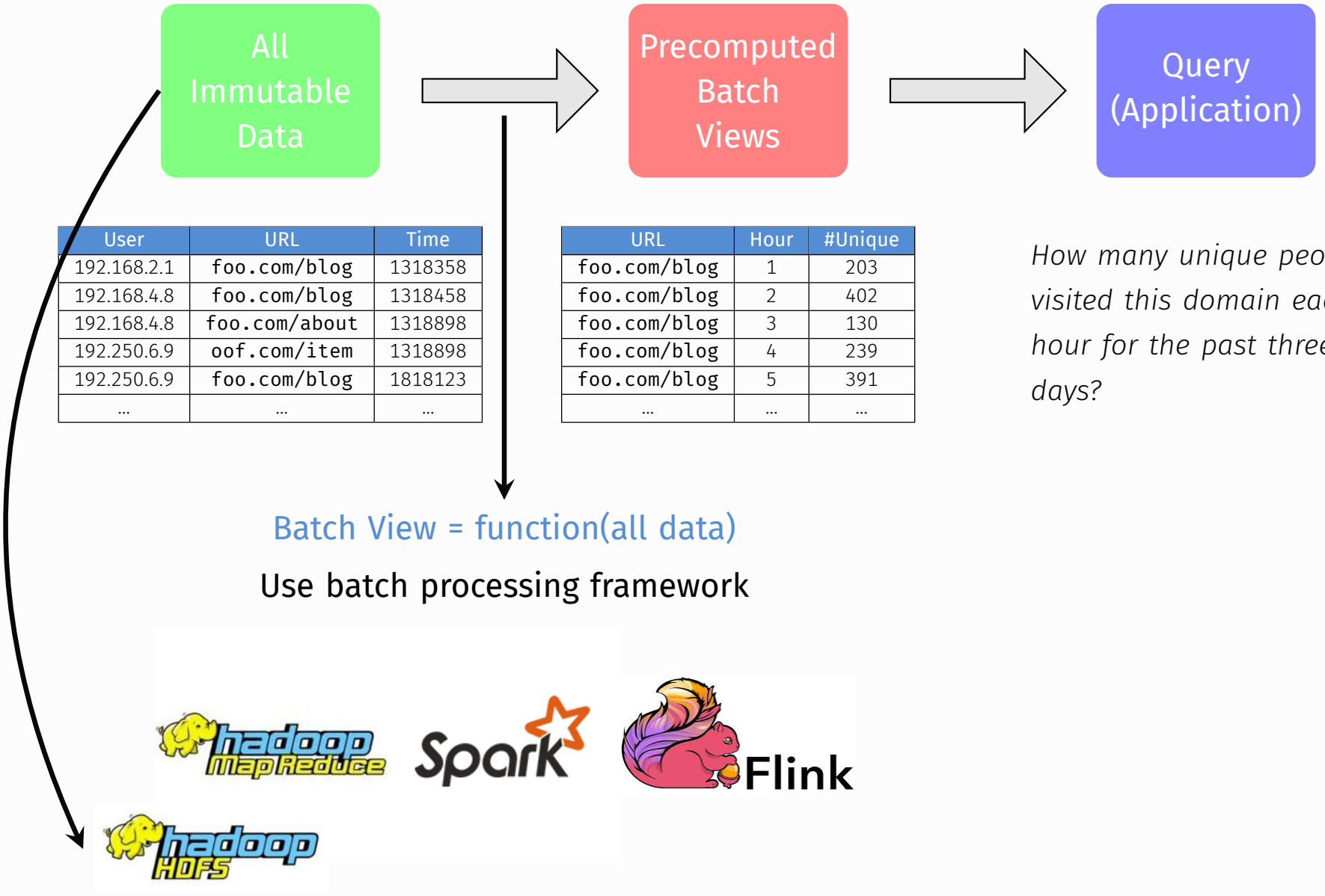
URL	Hour	#Unique
foo.com/blog	1	203
foo.com/blog	2	402
foo.com/blog	3	130
foo.com/blog	4	239
foo.com/blog	5	391
...	...	...

How many unique people visited this domain each hour for the past three days?

Batch View = function(all data)

Use batch processing framework

TOWARDS THE λ-ARCHITECTURE



User	URL	Time
192.168.2.1	foo.com/blog	1318358
192.168.4.8	foo.com/blog	1318458
192.168.4.8	foo.com/about	1318898
192.250.6.9	oof.com/item	1318898
192.250.6.9	foo.com/blog	1818123
...	...	...

URL	Hour	#Unique
foo.com/blog	1	203
foo.com/blog	2	402
foo.com/blog	3	130
foo.com/blog	4	239
foo.com/blog	5	391
...	...	...

How many unique people visited this domain each hour for the past three days?

TOWARDS THE λ-ARCHITECTURE

User	URL	Time
192.168.2.1	foo.com/blog	1318358
192.168.4.8	foo.com/blog	1318458
192.168.4.8	foo.com/about	1318898
192.250.6.9	oof.com/item	1318898
192.250.6.9	foo.com/blog	1818123
...	...	...

URL	Hour	#Unique
foo.com/blog	1	203
foo.com/blog	2	402
foo.com/blog	3	130
foo.com/blog	4	239
foo.com/blog	5	391
...	...	...

How many unique people visited this domain each hour for the past three days?

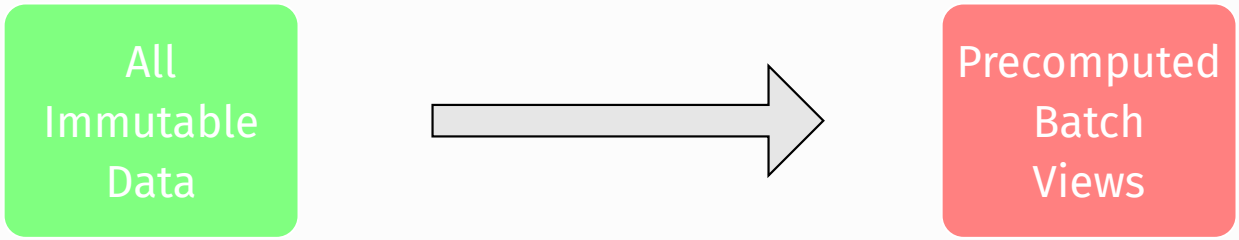
Query = fast function(batch view)

Use a batch view database that:

- Is highly available
- Scalable
- Supports batch-writes (from batch framework)
- Supports fast random (indexed) reads (low-latency querying)
- Note: random writes not necessary!

ARE WE THERE YET?

Not quite: The raw data changes all the time. How do we keep the batch views up to date?

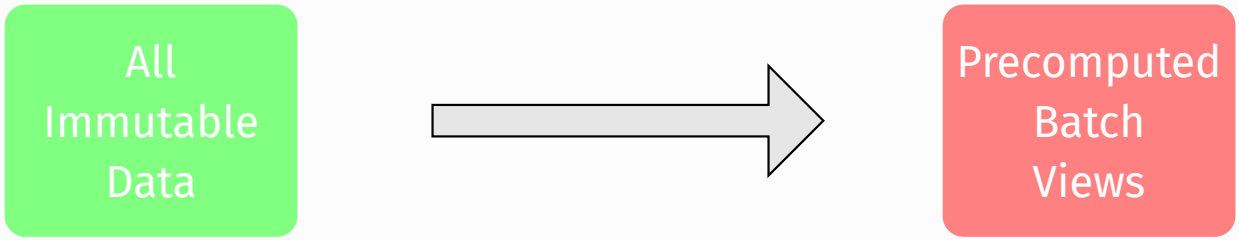


User	URL	Time
192.168.2.1	foo.com/blog	1318358
192.168.4.8	foo.com/blog	1318458
192.168.4.8	foo.com/about	1318898
192.250.6.9	oof.com/item	1318898
192.250.6.9	foo.com/blog	1818123

URL	Hour	#Unique
foo.com/blog	1	203
foo.com/blog	2	402
foo.com/blog	3	130
foo.com/blog	4	239
foo.com/blog	5	391

ARE WE THERE YET?

Not quite: The raw data changes all the time. How do we keep the batch views up to date?



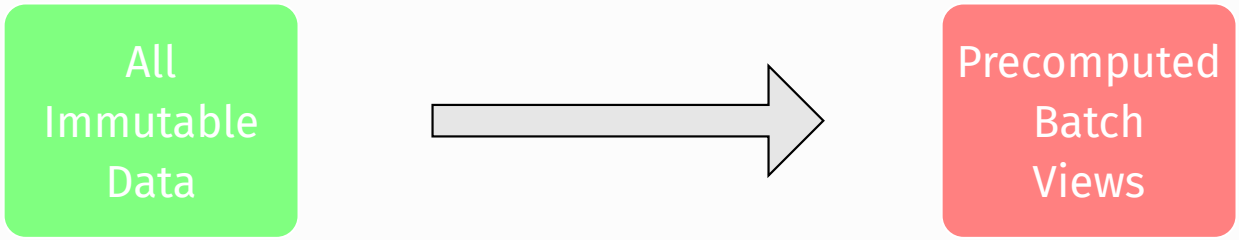
User	URL	Time
192.168.2.1	foo.com/blog	1318358
192.168.4.8	foo.com/blog	1318458
192.168.4.8	foo.com/about	1318898
192.250.6.9	oof.com/item	1318898
192.250.6.9	foo.com/blog	1818123
152.214.1.2	goo.com/idx	2010911
182.228.1.2	foo.com/blog	2012918
192.168.4.8	foo.com/about	2318898
192.250.6.9	oof.com/item	2318898
...	...	...

URL	Hour	#Unique
foo.com/blog	1	203
foo.com/blog	2	402
foo.com/blog	3	130
foo.com/blog	4	239
foo.com/blog	5	391

ARE WE THERE YET?

Not quite: The raw data changes all the time. How do we keep the batch views up to date?

Answer: Try and incrementalize the batch computation as much as possible.



User	URL	Time
192.168.2.1	foo.com/blog	1318358
192.168.4.8	foo.com/blog	1318458
192.168.4.8	foo.com/about	1318898
192.250.6.9	oof.com/item	1318898
192.250.6.9	foo.com/blog	1818123
152.214.1.2	goo.com/idx	2010911
182.228.1.2	foo.com/blog	2012918
192.168.4.8	foo.com/about	2318898
192.250.6.9	oof.com/item	2318898
...	...	...

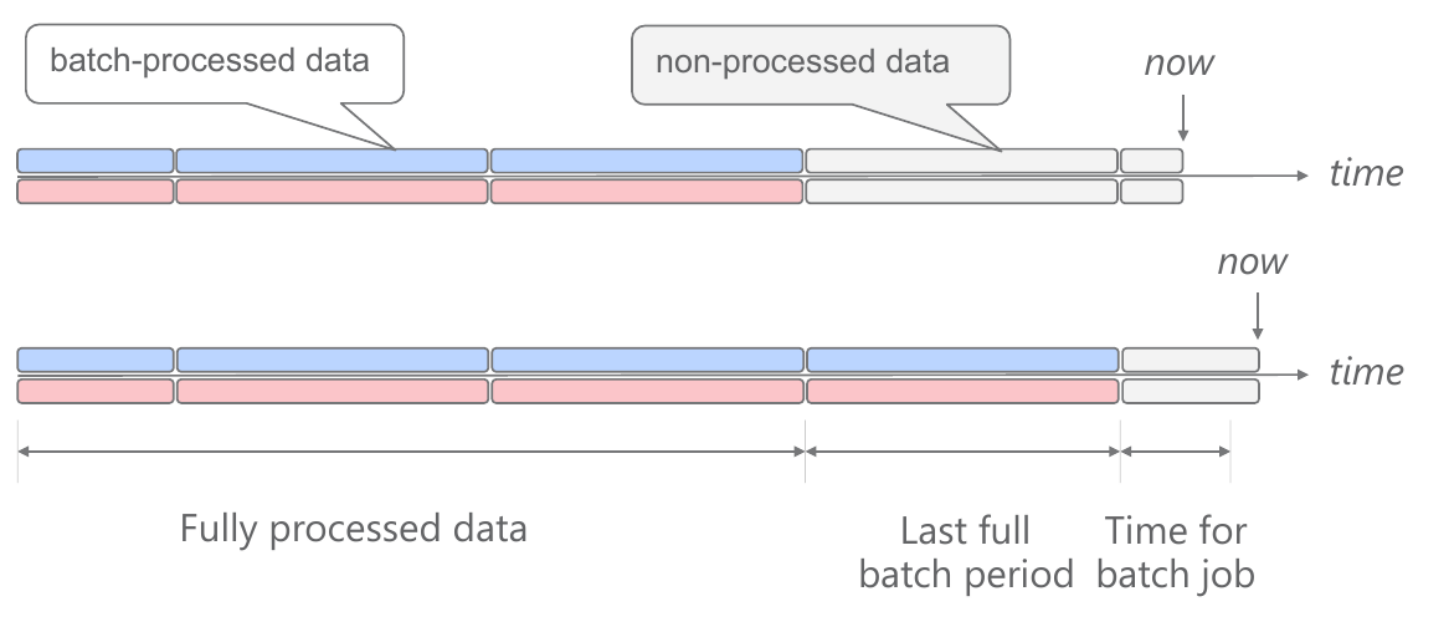
URL	Hour	#Unique
foo.com/blog	1	203
foo.com/blog	2	402
foo.com/blog	3	130
foo.com/blog	4	239
foo.com/blog	5	391
foo.com/blog	6	203
foo.com/blog	7	402
foo.com/blog	8	130
foo.com/blog	9	239
...	...	...

ARE WE THERE YET?

Still not quite: Even incremental batch computation can be relatively slow.

ARE WE THERE YET?

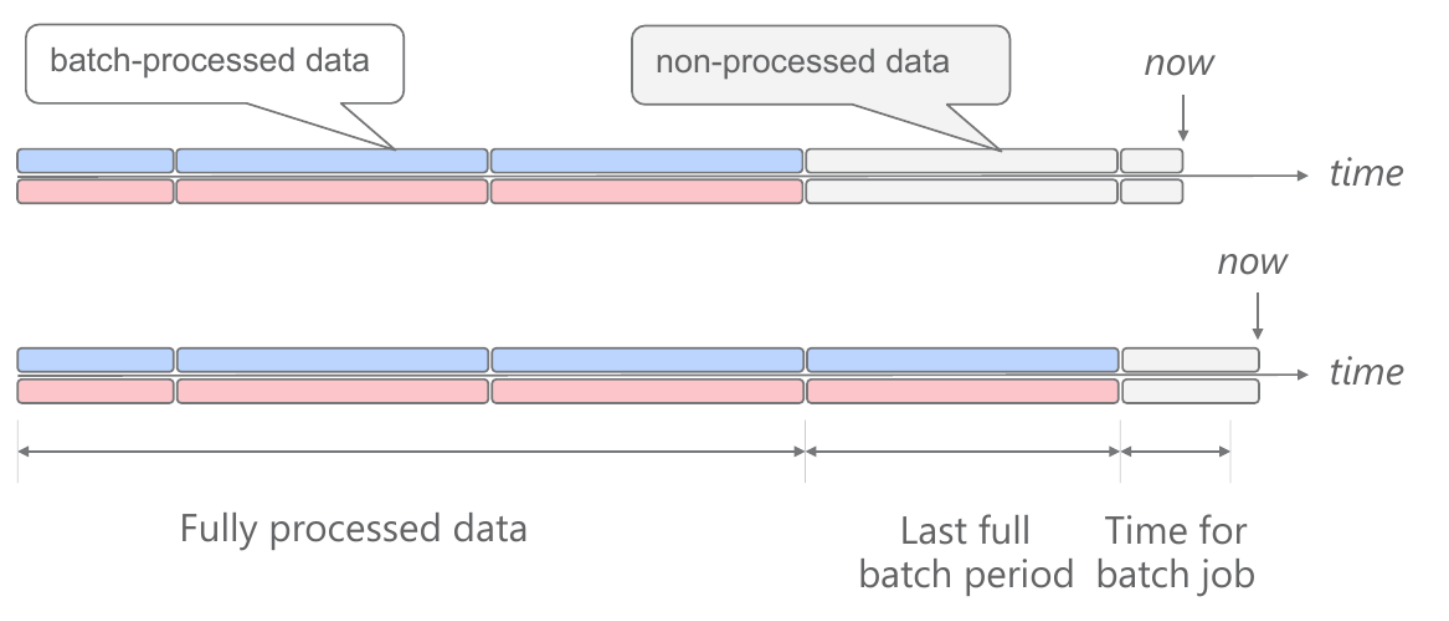
Still not quite: Even incremental batch computation can be relatively slow.



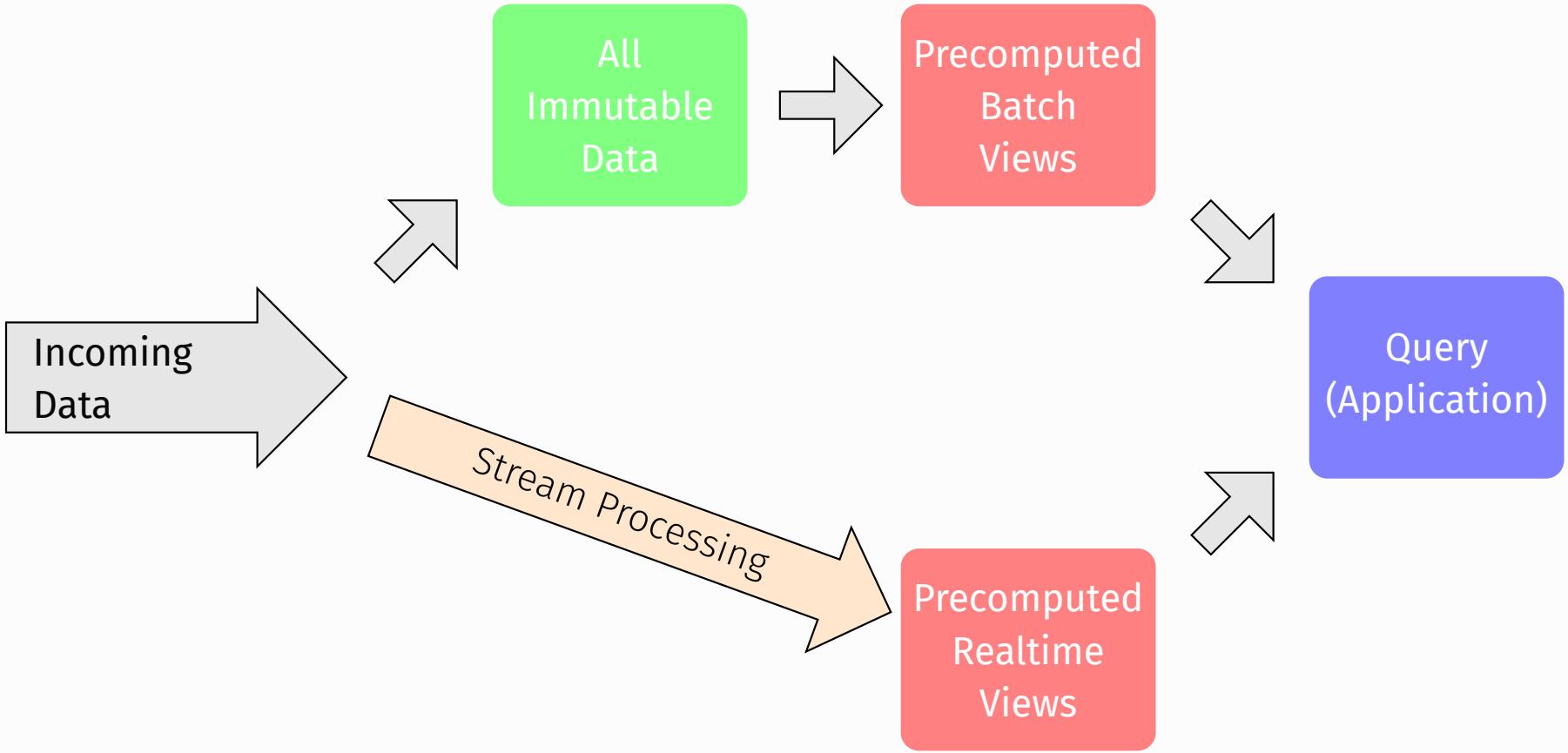
ARE WE THERE YET?

Still not quite: Even incremental batch computation can be relatively slow.

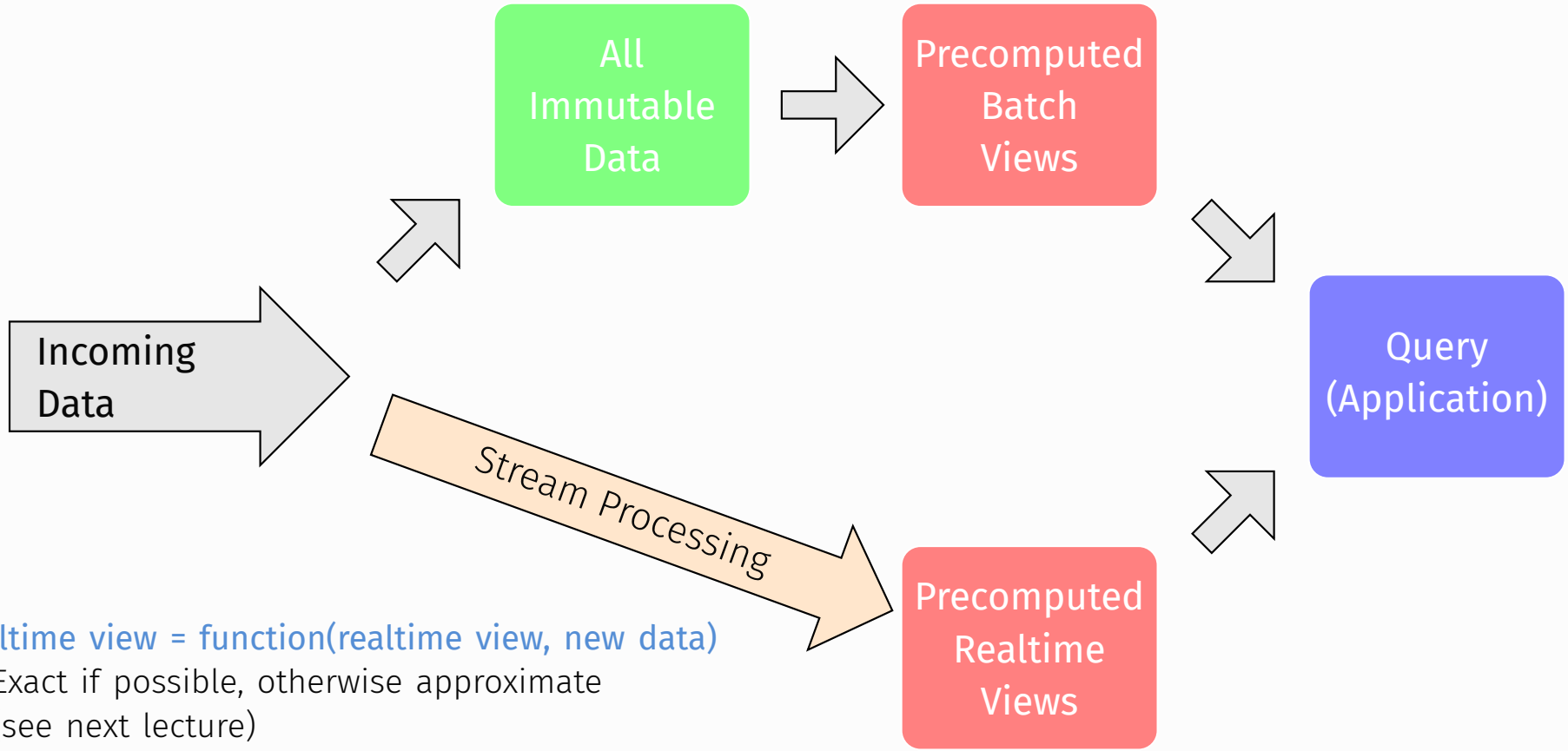
Solution: Treat “fresh” data separately: real-time stream processing.



ADDING REAL-TIME STREAM PROCESSING



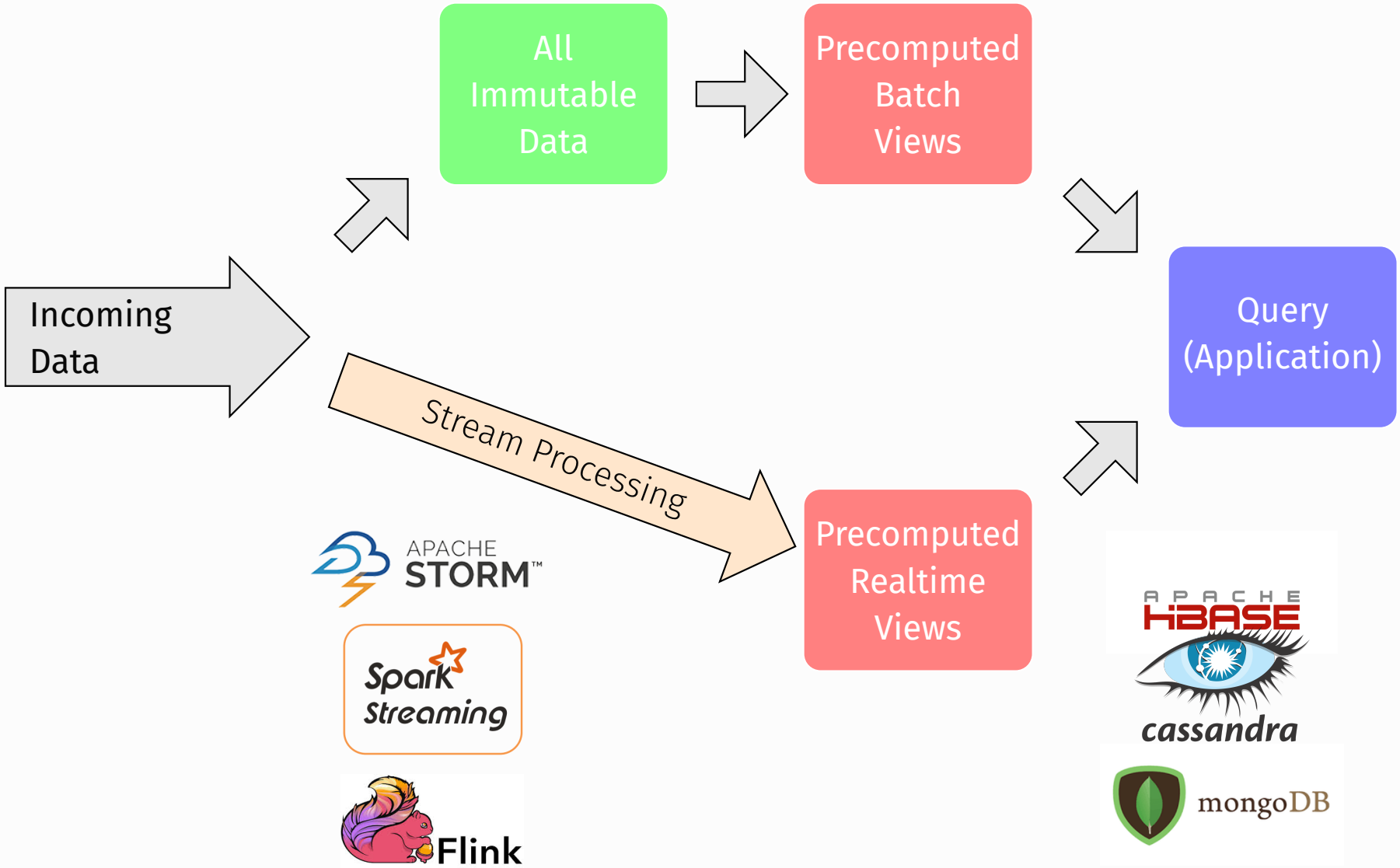
ADDING REAL-TIME STREAM PROCESSING



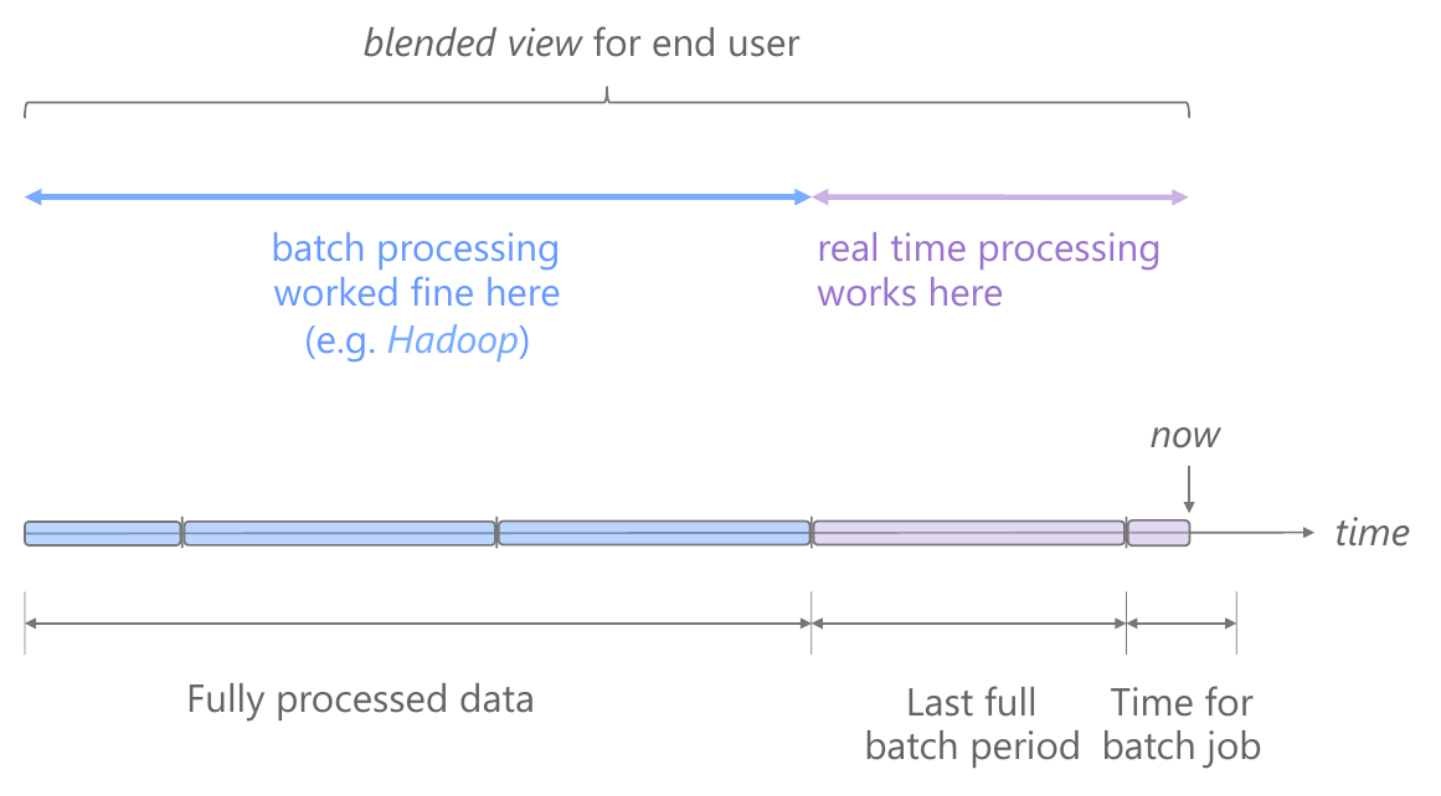
Realtime view = function(realtime view, new data)

- Exact if possible, otherwise approximate (see next lecture)
- Approximate realtime-views become exact batch view over time.
- Realtime view database must support random writes (= more complex).

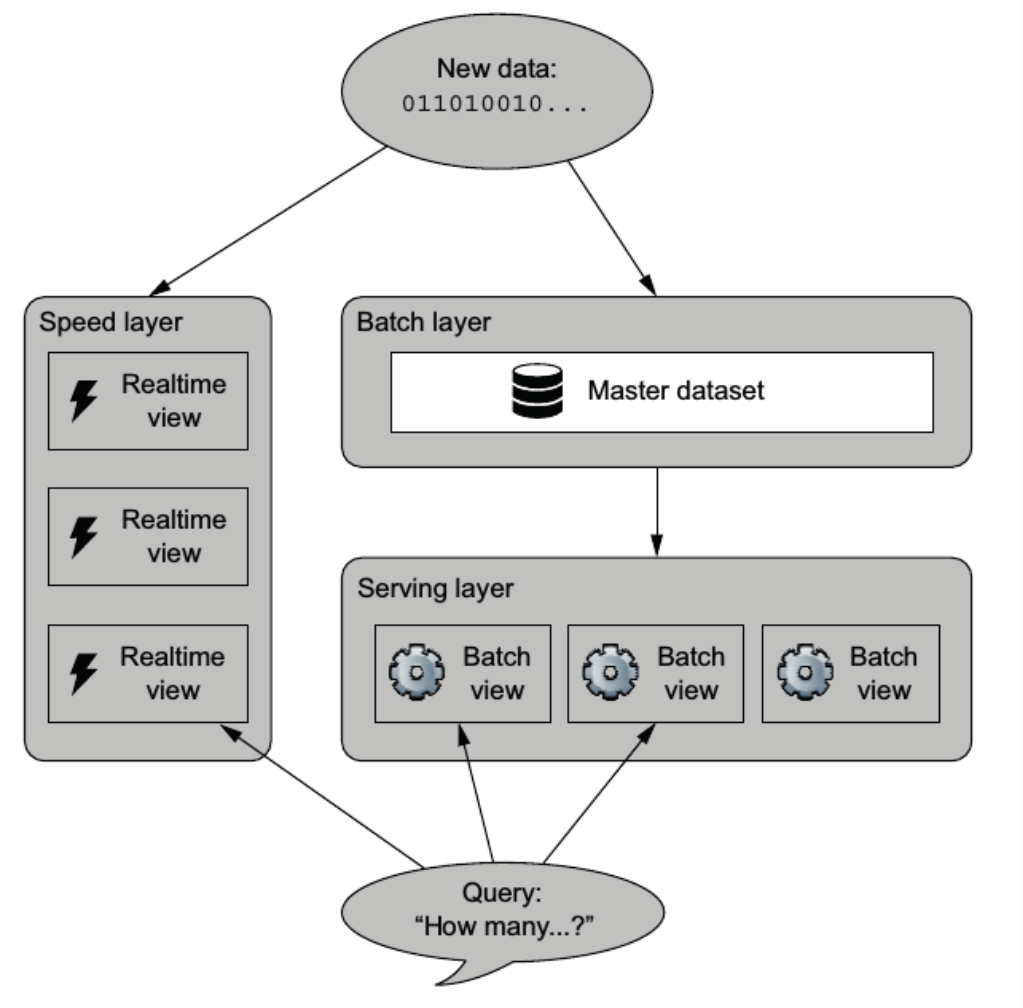
ADDING REAL-TIME STREAM PROCESSING



ADDING REAL-TIME STREAM PROCESSING



IN SUMMARY: THE λ-ARCHITECTURE



The λ-architecture is a big data architecture that distinguishes between three layers.

Figure Source: Big Data Book

IN SUMMARY: THE λ-ARCHITECTURE

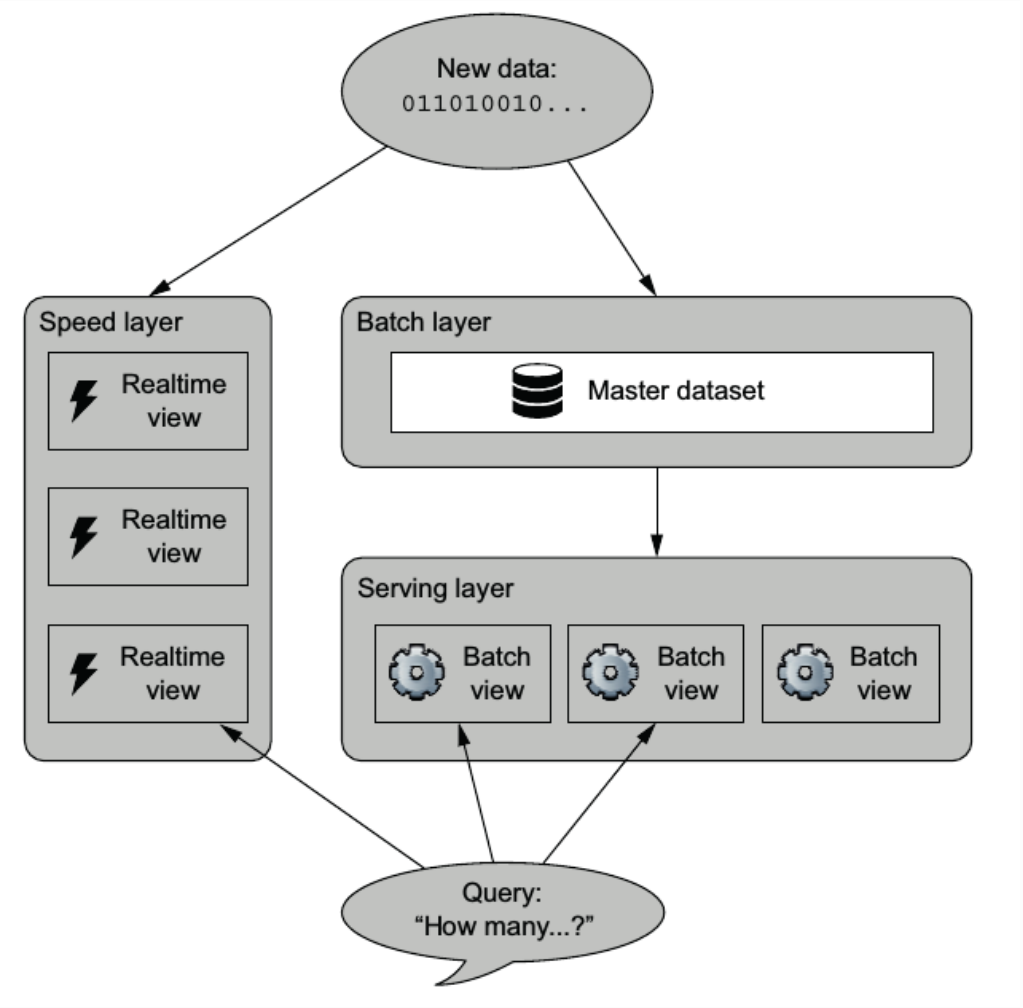


Figure Source: Big Data Book

The λ-architecture is a big data architecture that distinguishes between three layers.

The Batch Layer

- Keeps an immutable, historical log of all data also known as the data lake.
- Computes batch views from this master data.
- Allows ad-hoc (but high-latency) querying.

IN SUMMARY: THE λ-ARCHITECTURE

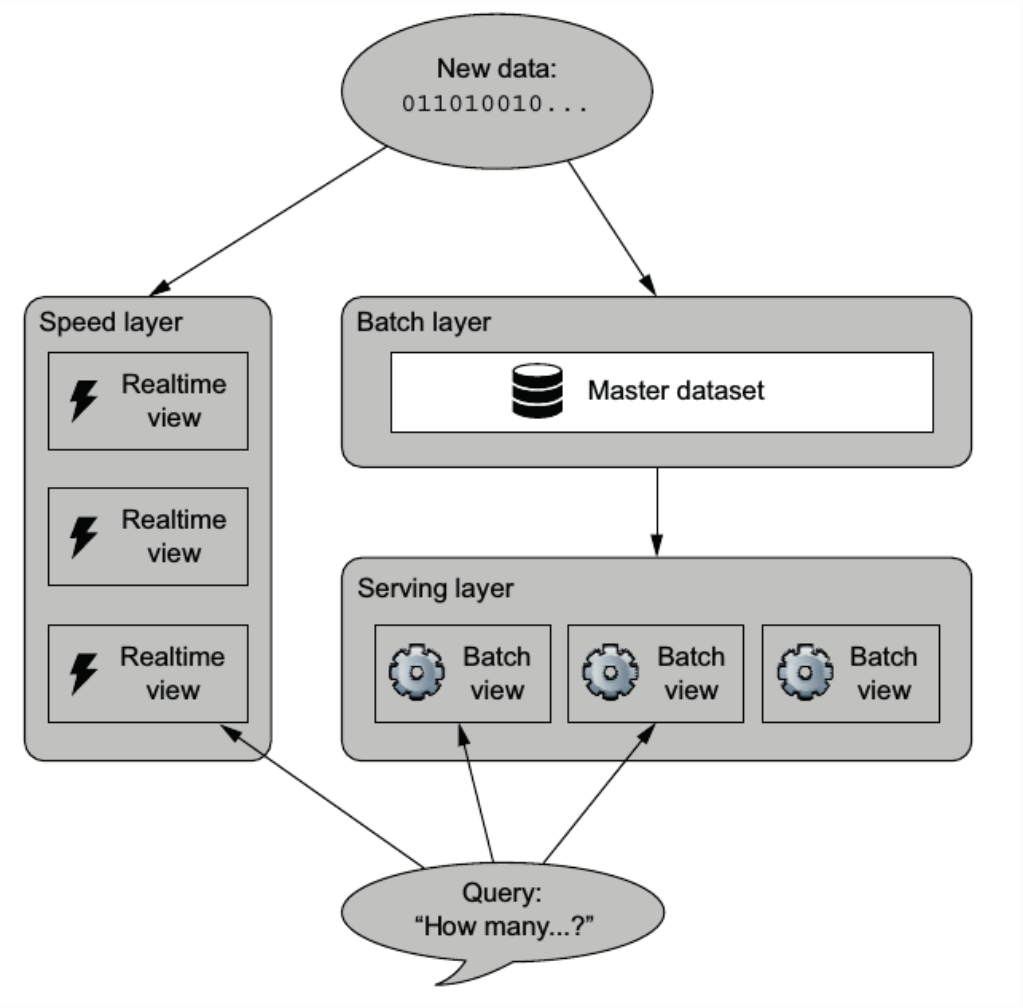


Figure Source: Big Data Book

The λ-architecture is a big data architecture that distinguishes between three layers.

The Serving Layer

- Stores the batch views.
- Uses these views to respond to pre-defined queries.

IN SUMMARY: THE λ-ARCHITECTURE

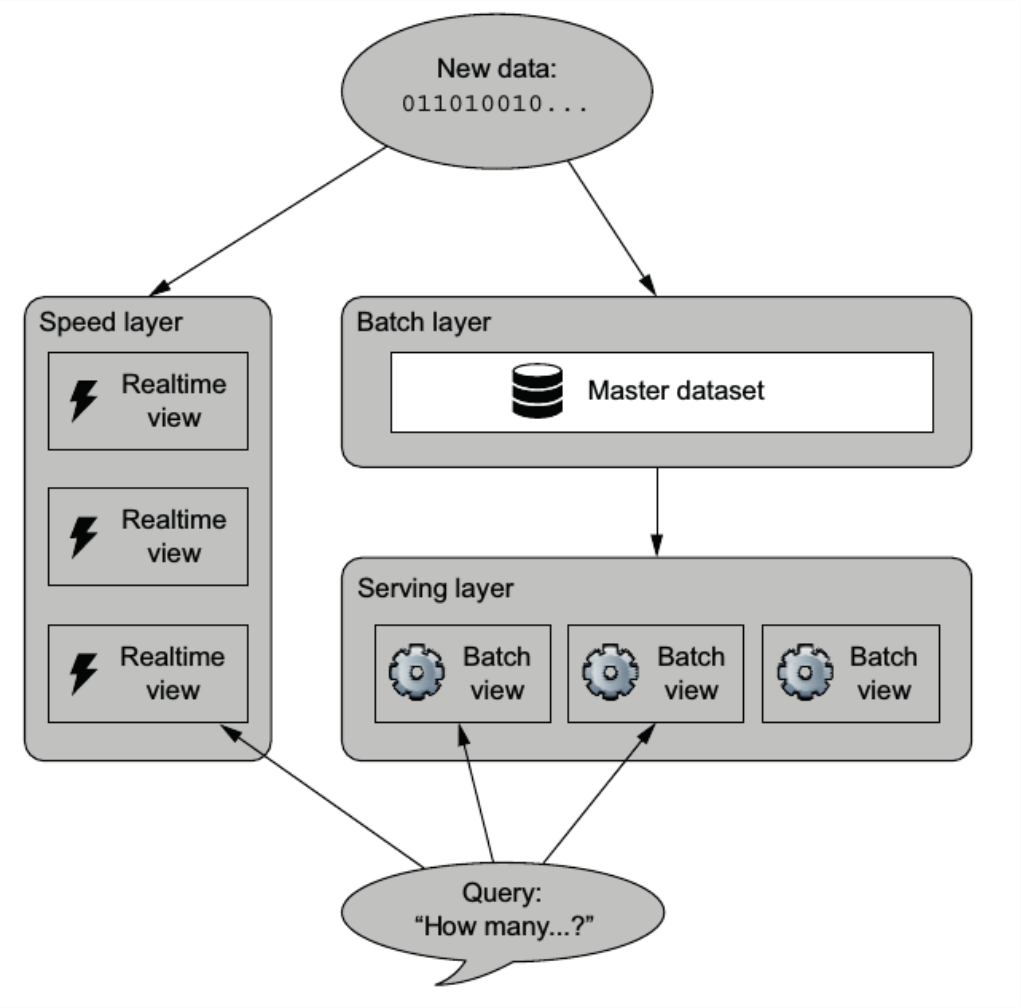


Figure Source: Big Data Book

The λ-architecture is a big data architecture that distinguishes between three layers.

The Speed Layer

- Computes realtime views on streaming new data.
- Uses these views to augment the batch views for responding to pre-defined queries.

IN SUMMARY: THE λ-ARCHITECTURE

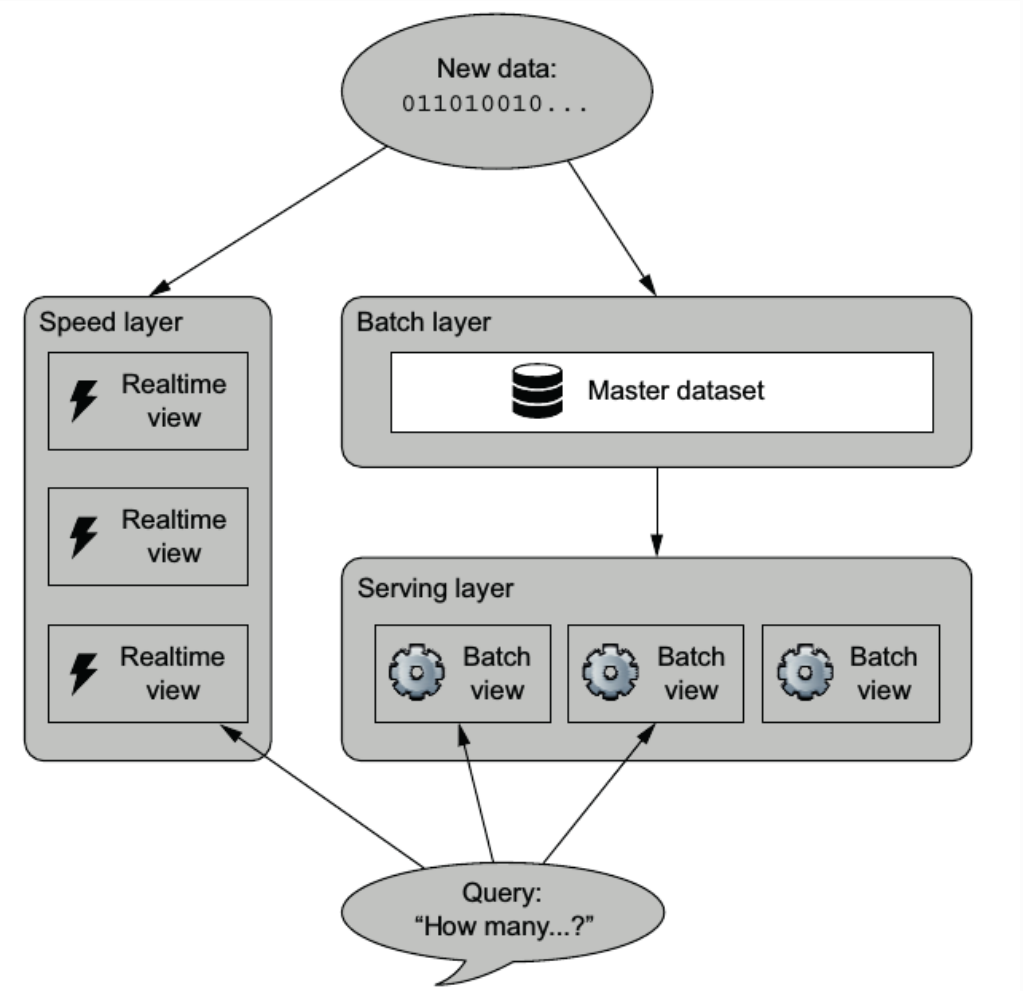


Figure Source: Big Data Book

The λ-architecture is a big data architecture that distinguishes between three layers.

The λ-architecture is a meta-architecture: concrete frameworks for each layer can be chosen on a case-by-case basis.

THE OFTEN-FORGOTTEN LAYERS

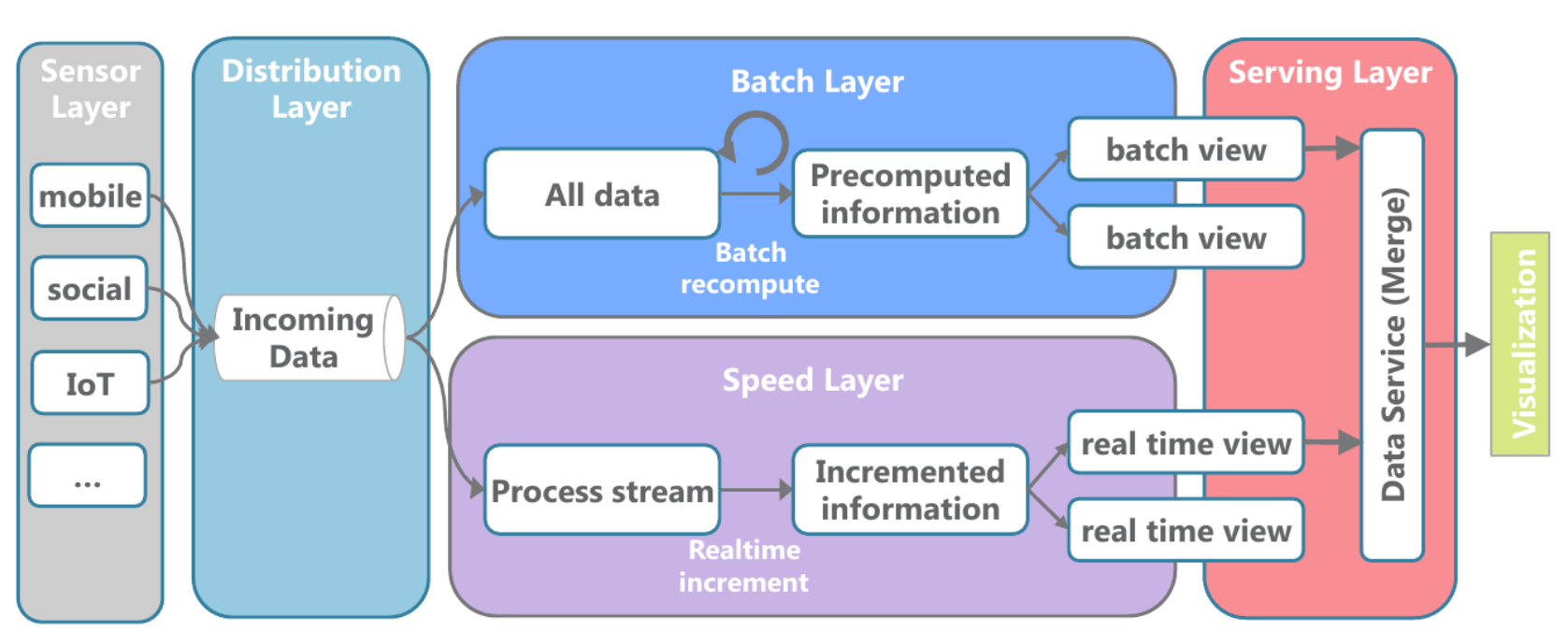


Figure Source: `https://www.slideshare.net/gschmutz/big-data-and-fast-data-lambda-architecture-in-action`

THE OFTEN-FORGOTTEN LAYERS

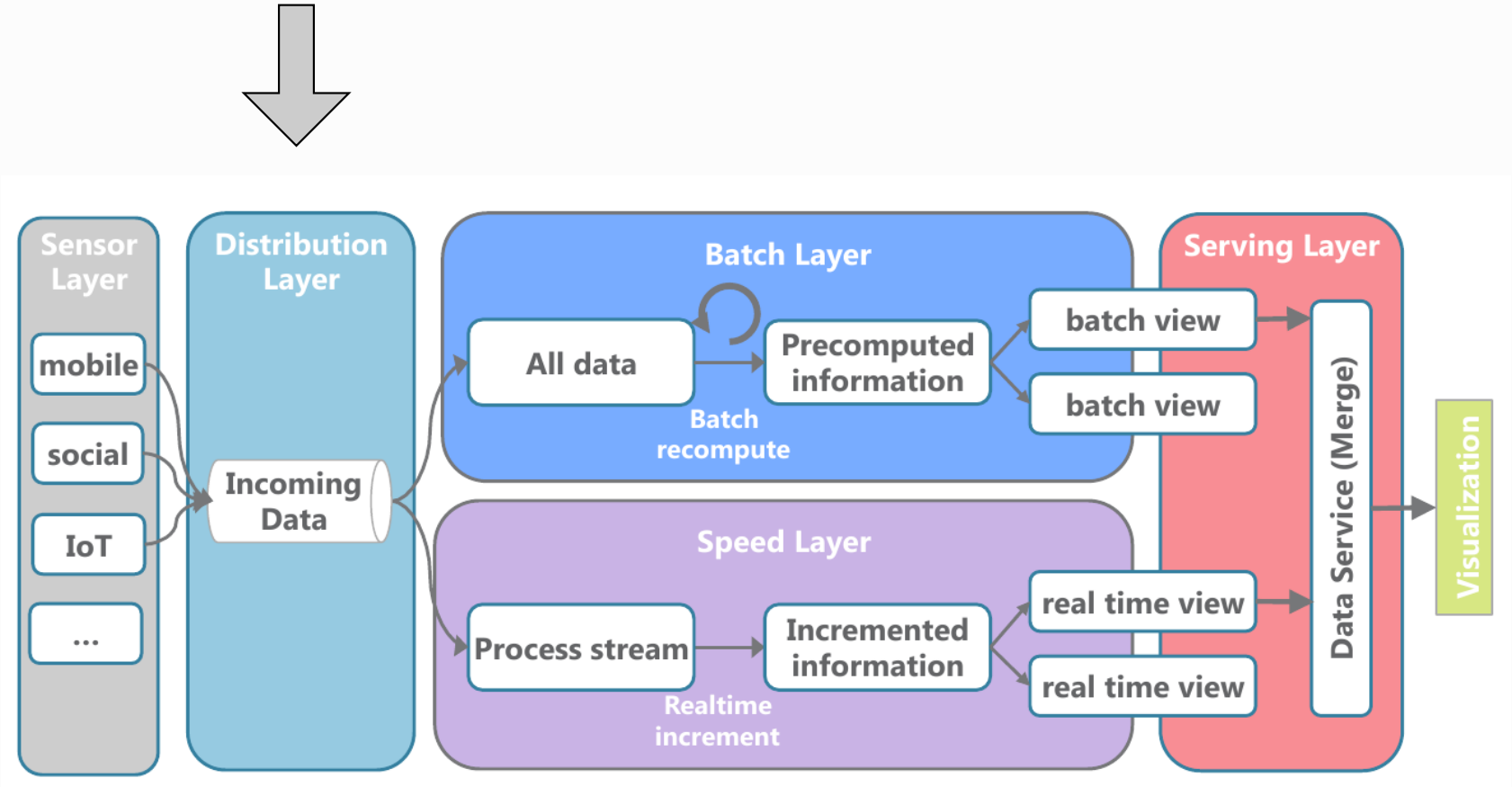
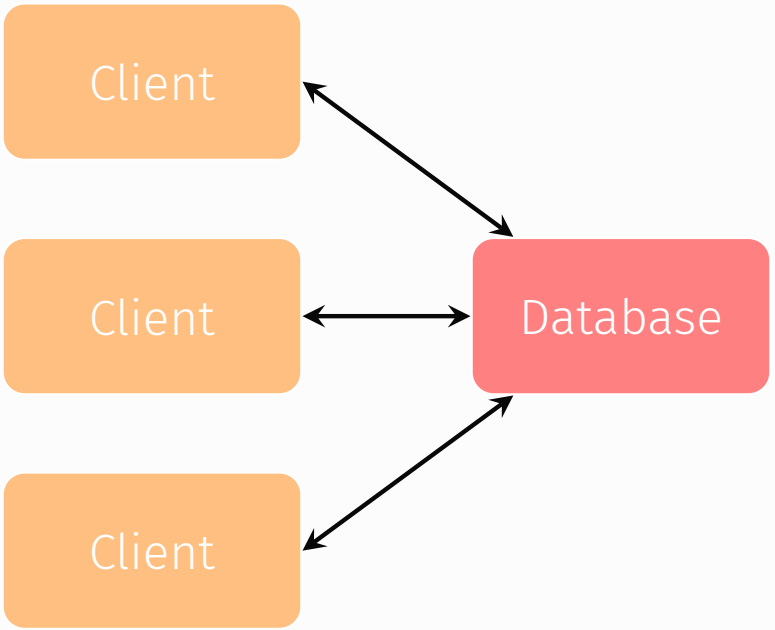


Figure Source: `https://www.slideshare.net/gschmutz/big-data-and-fast-data-lambda-architecture-in-action`

DISTRIBUTION LAYER: MESSAGE QUEUES

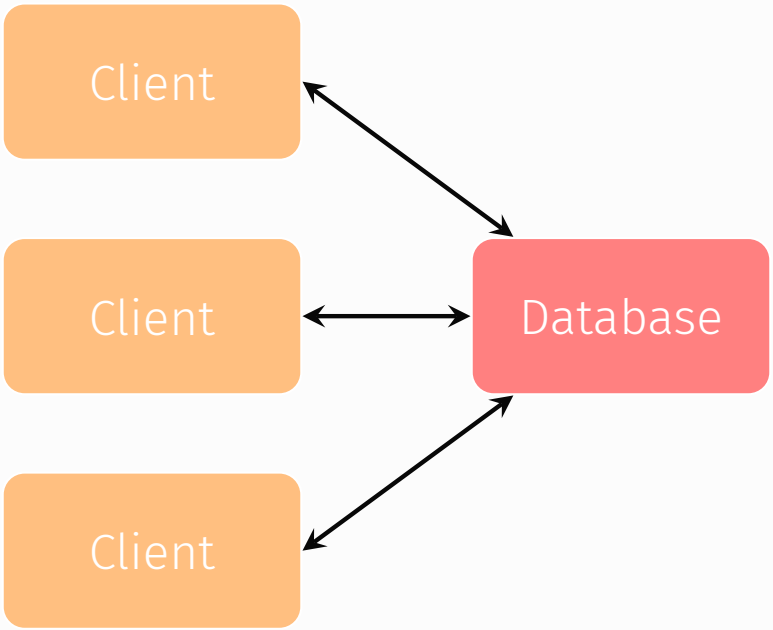
TWO WAYS OF PROCESSING UPDATES (1/2)



Synchronous

- Direct connection client ↔ database
- Client waits until update complete.

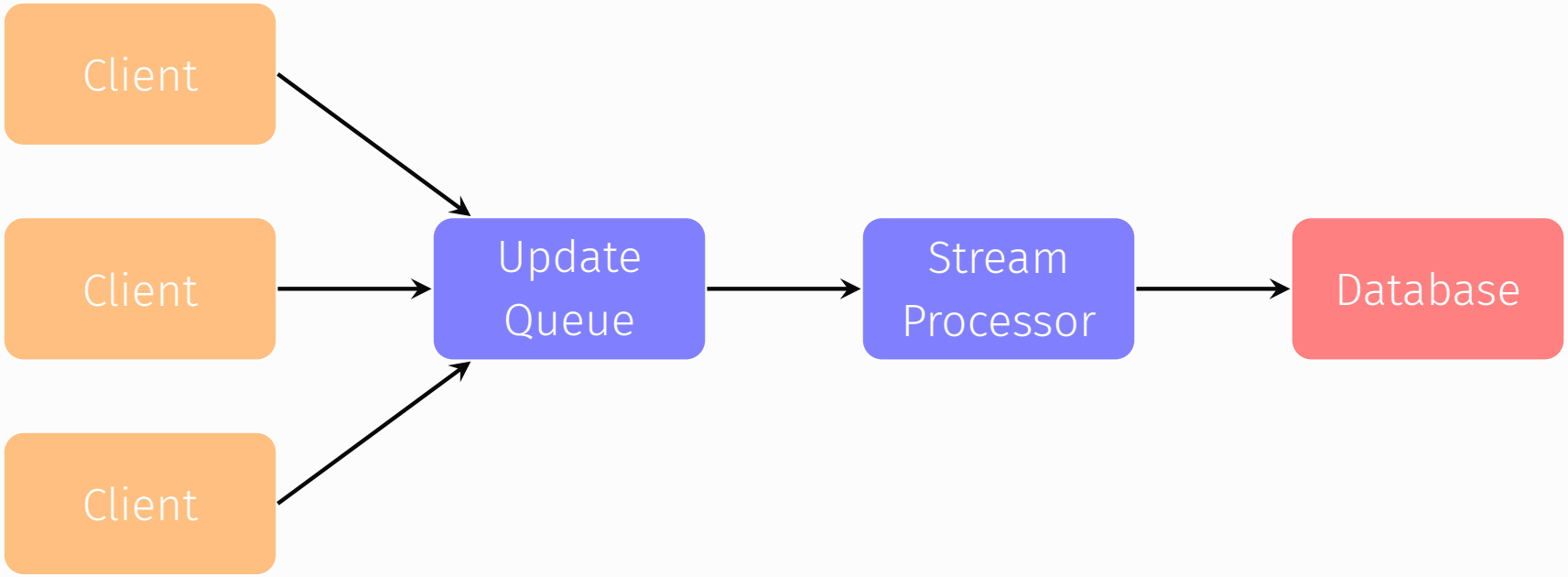
TWO WAYS OF PROCESSING UPDATES (1/2)



Synchronous

- Direct connection client ↔ database
- Client waits until update complete.
- 🙂 Client knows when update is done
- ☹ Spikes can overload the database

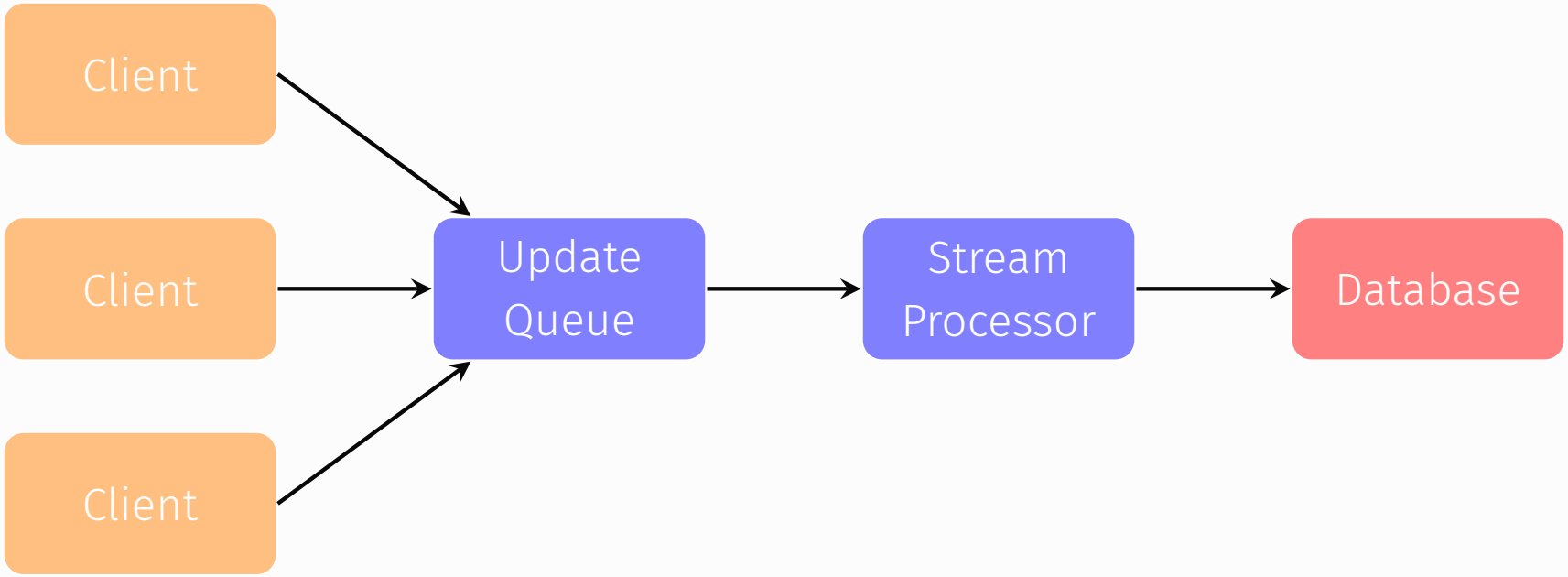
TWO WAYS OF PROCESSING UPDATES (2/2)



Asynchronous

- Client submits update to a queue, and continues work without waiting.
- Updates are processed by stream processor when possible.

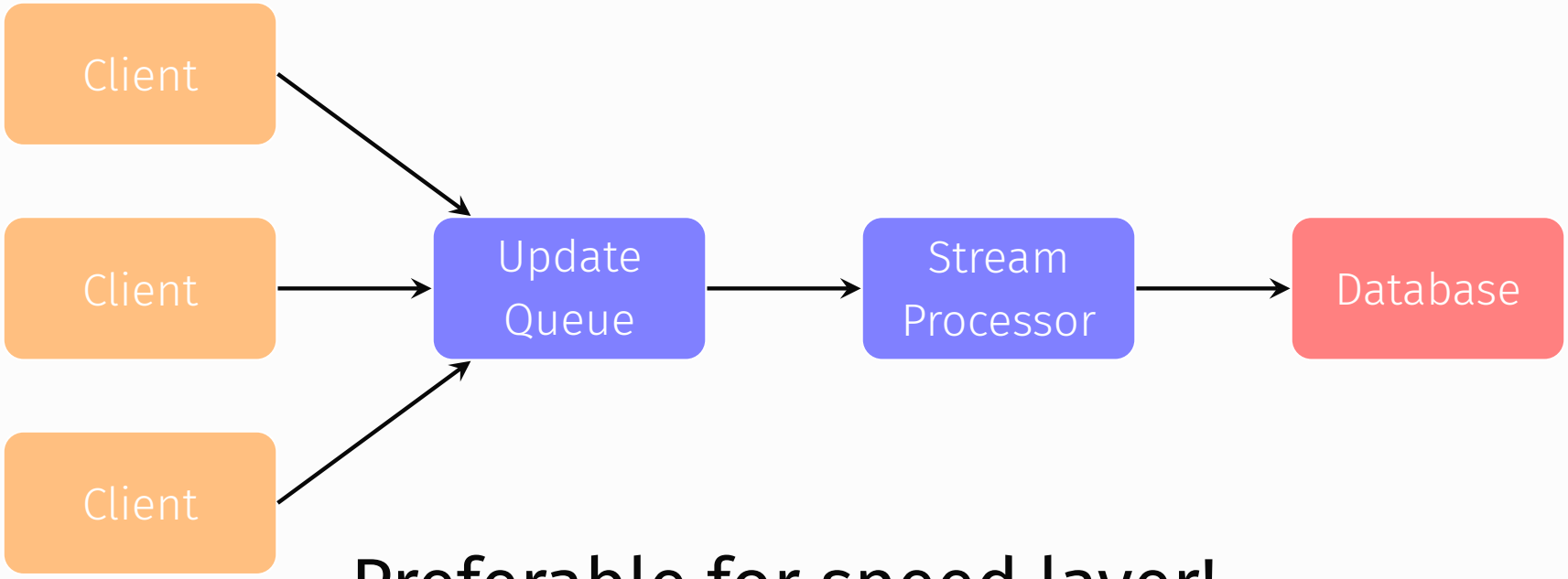
TWO WAYS OF PROCESSING UPDATES (2/2)



Asynchronous

- Client submits update to a queue, and continues work without waiting.
- Updates are processed by stream processor when possible.
- 🙂 Load spikes are easily handled
- 🙁 Special mechanism required to acknowledge update is processed

TWO WAYS OF PROCESSING UPDATES (2/2)

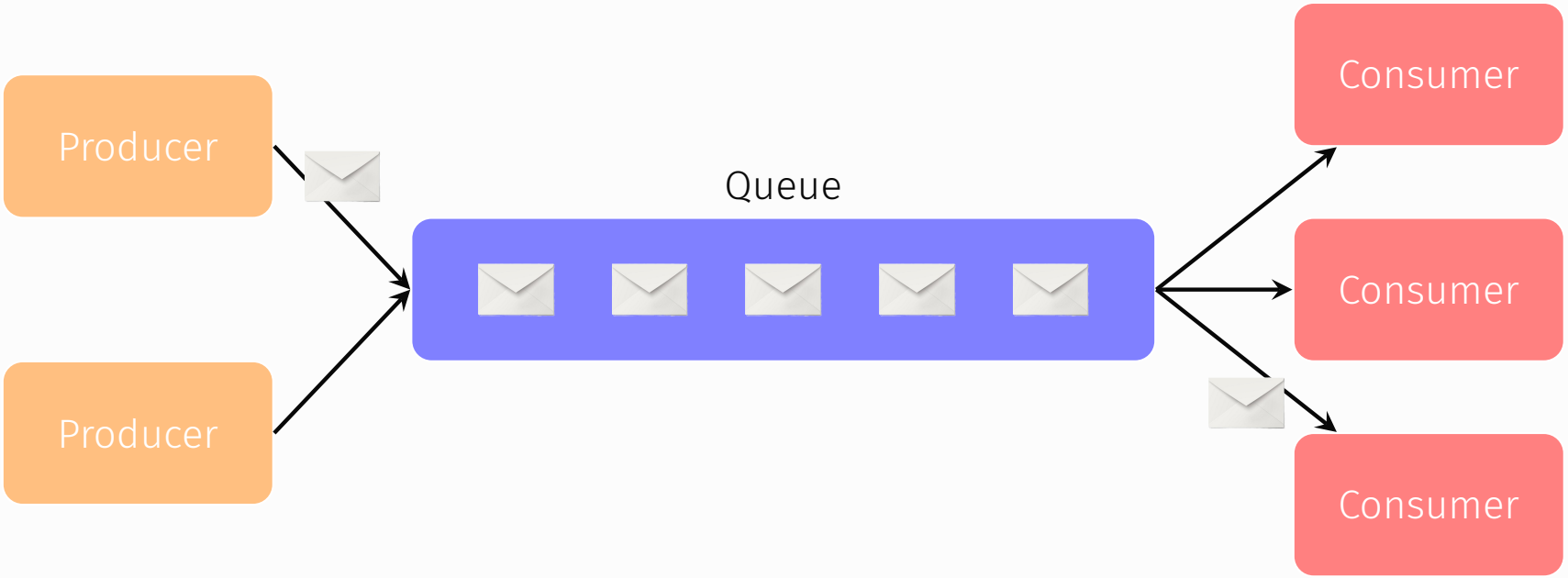


Preferable for speed layer!

Asynchronous

- Client submits up[illegible] tinues work without waiting.
- Updates are proce[illegible] by [illegible] possible.
- 🙂 Load spikes are easily [illegible]
- 🙁 Special mechanism required to acknowledge [illegible] is processed

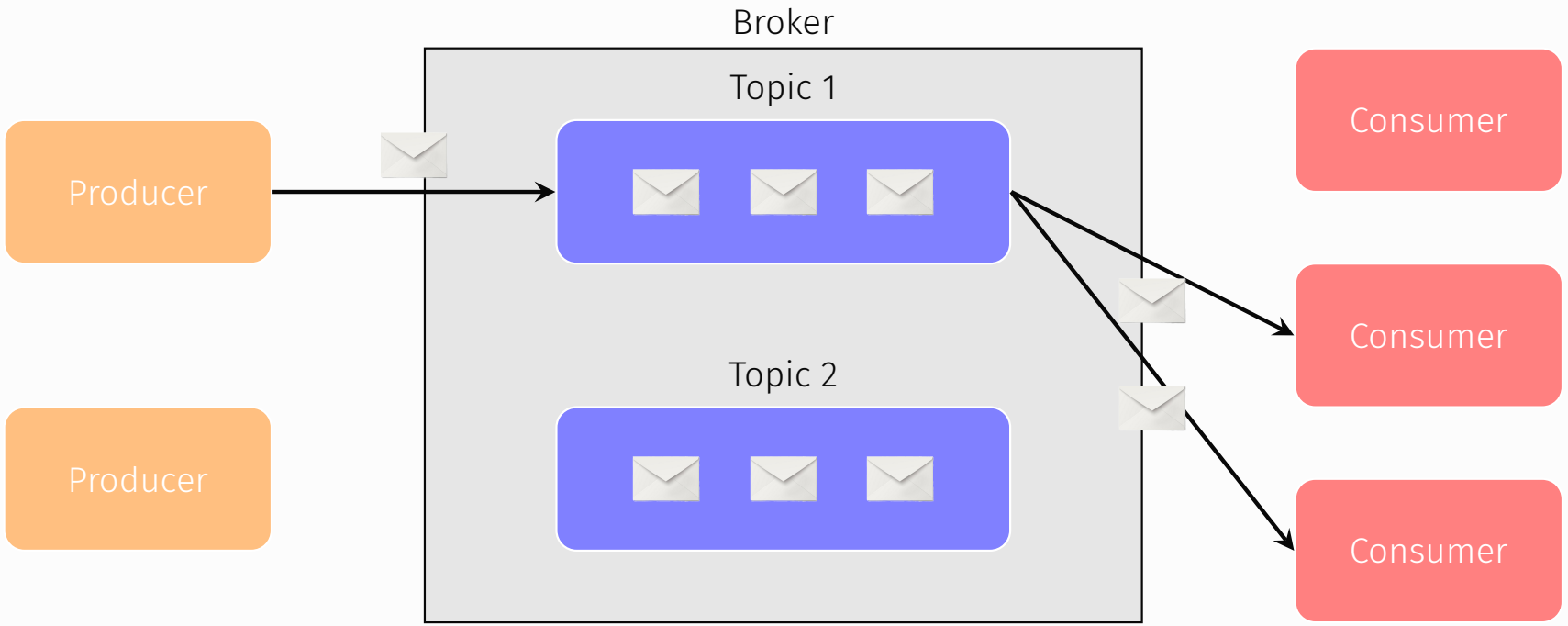
TWO TYPES OF MESSAGE QUEUES



Single-consumption

- Also known as point-to-point queue, message bus, message queue
- Producers (senders) push messages to queue.
- Consumers (receivers) pop messages from queue.
- Parallellism through (round-robin) distribution over consumers
- ☹ Every message is consumed by only one consumer! By default, a message is deleted from the queue when consumer fetches message (→ fault-tolerance problem).

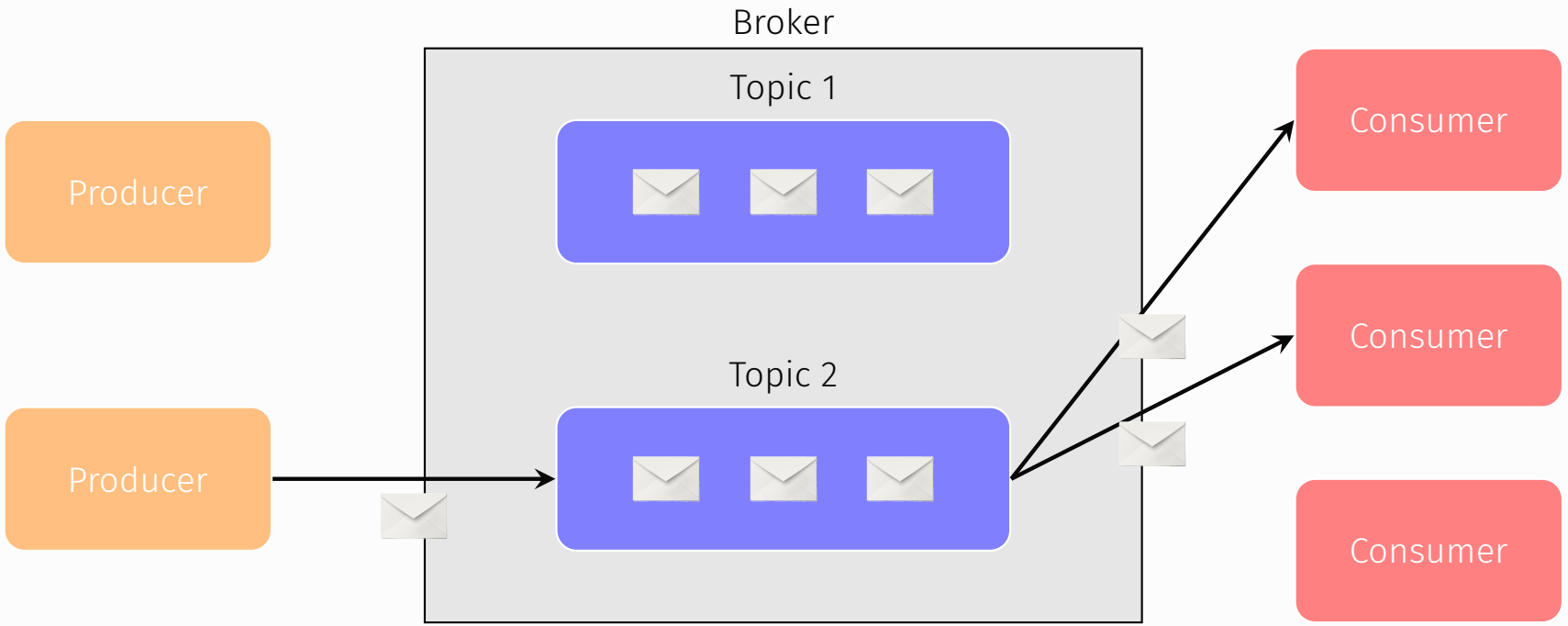
TWO TYPES OF MESSAGE QUEUES



Publish-subscribe messaging

- The queue is organized into topics
- A broker is responsible for queue maintenance and message delivery.
- Producers (publishers, senders) push messages to topics on queue.
- Consumers (subscribers, receivers) receive messages from the topics that they are subscribed to. The broker decides when old messages can be deleted.
- 🙂 Every message is processed by all consumers subscribed to the message topic!
- 😕 Scalability (Parallellism)?

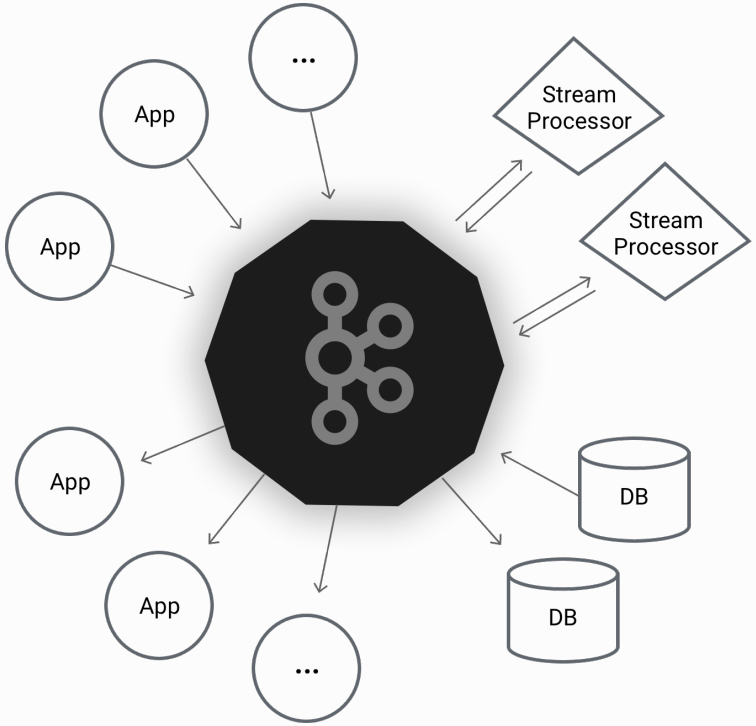
TWO TYPES OF MESSAGE QUEUES



Publish-subscribe messaging

- The queue is organized into topics
- A broker is responsible for queue maintenance and message delivery.
- Producers (publishers, senders) push messages to topics on queue.
- Consumers (subscribers, receivers) receive messages from the topics that they are subscribed to. The broker decides when old messages can be deleted.
- 🙂 Every message is processed by all consumers subscribed to the message topic!
- 😕 Scalability (Parallellism)?

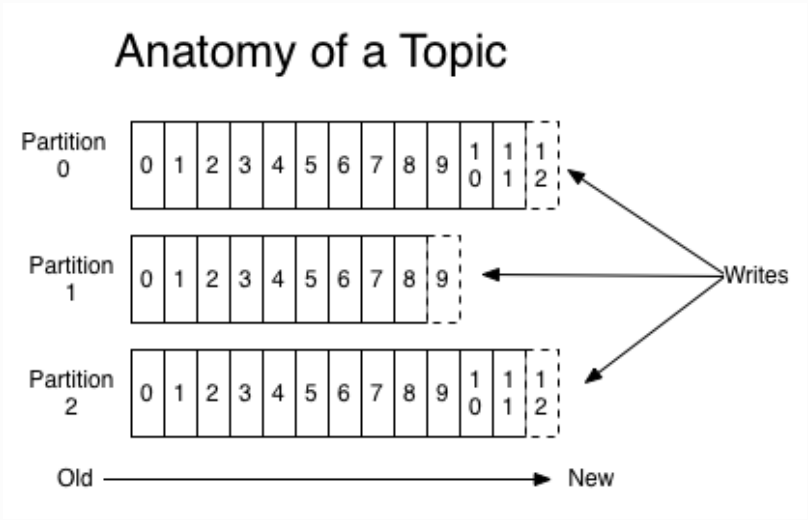
APACHE KAFKA



Apache Kafka is a distributed messaging / streaming platform based on partitioned publish-subscribe.

- Topics are partitioned.
- Partitions are distributed for scalability and replicated for fault-tolerance.

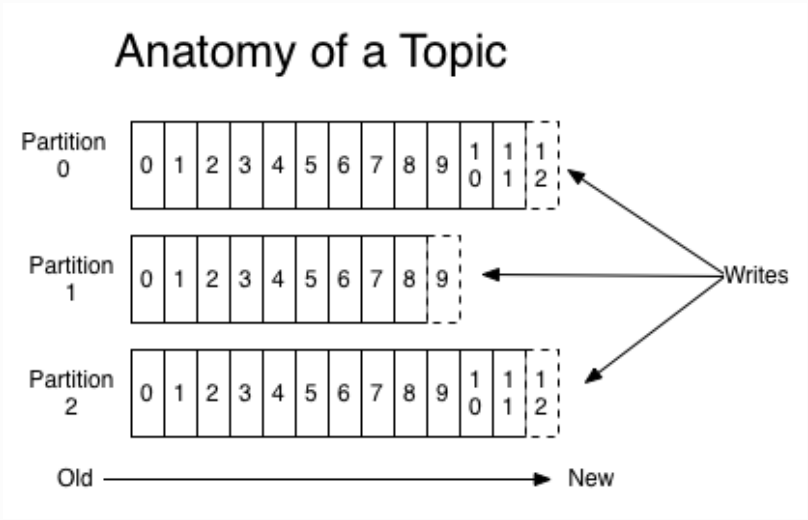
APACHE KAFKA



Apache Kafka is a distributed messaging / streaming platform based on partitioned publish-subscribe.

- Topics are partitioned.
- Partitions are distributed for scalability and replicated for fault-tolerance.

APACHE KAFKA



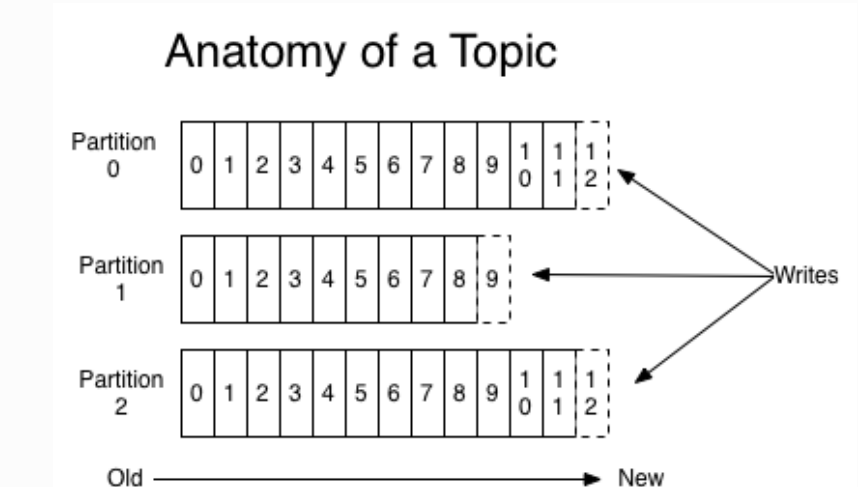
Apache Kafka is a distributed messaging / streaming platform based on partitioned publish-subscribe.

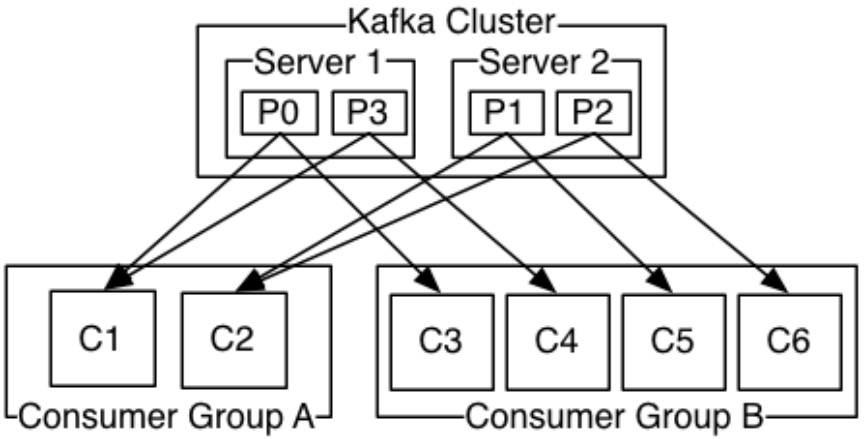
- Topics are partitioned.
- Partitions are distributed for scalability and replicated for fault-tolerance.

Distributed production:

- Producers can choose to push a message on topic *X* to a particular partition of *X* (load-balances production)

APACHE KAFKA





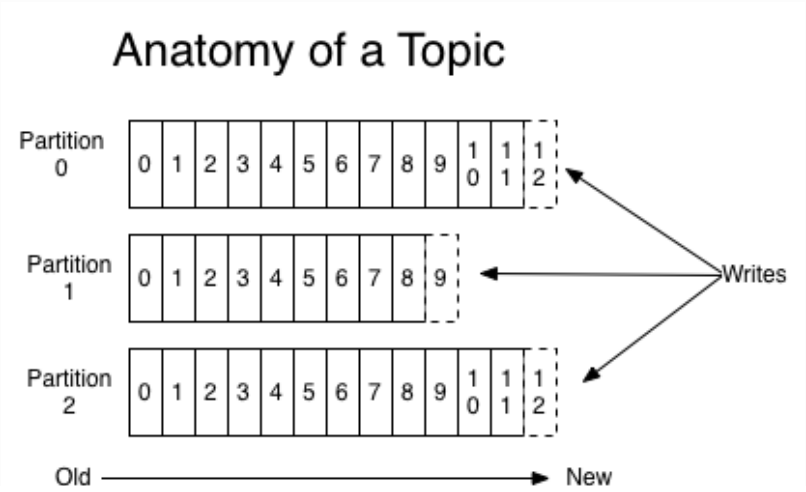
Apache Kafka is a distributed messaging / streaming platform based on partitioned publish-subscribe.

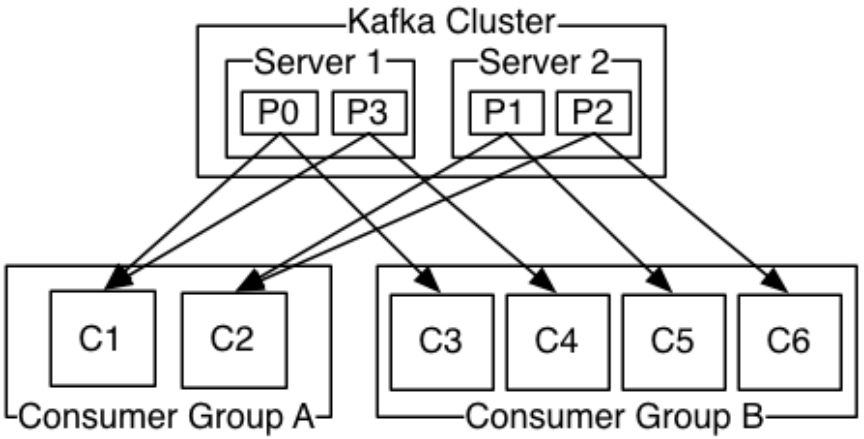
- Topics are partitioned.
- Partitions are distributed for scalability and replicated for fault-tolerance.

Hybrid consumption model:

- Consumers are collected into consumer groups
- Every message to topic *X* is processed by each consumer group subscribed to *X* (publish subscribe)
- However, each message is handled by only one consumer inside a consumer group (point-to-point), which allows load-balancing consumption

APACHE KAFKA





Apache Kafka is a distributed messaging / streaming platform based on partitioned publish-subscribe.

- Topics are partitioned.
- Partitions are distributed for scalability and replicated for fault-tolerance.

Some specifics:

- Kafka consumption is pull-based; consumers can choose to retrieve messages in batch
- All messages are stored on disk and retained for a configurable period (consumers can hence replay messages if need be)
- So you can also think of it as a distributed filesystem (for temporary files).

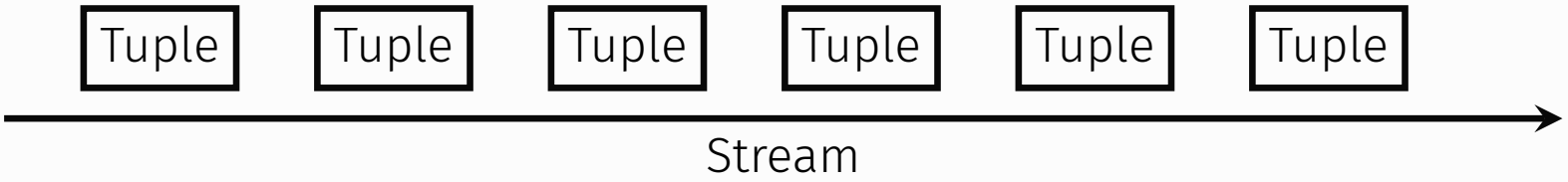
SPEED LAYER: TUPLE-AT-A-TIME PROCESSING

APACHE STORM/TWITTER HERON

Apache Storm

- Distributed and Fault-Tolerant real-time computation
- Developed at Backtype/Twitter, open sourced in 2011
- Has been superseded by Heron at Twitter since 2014 (open-sourced by Apache).
- Heron has the same programming model, but different implementation.

STORM CONCEPTS



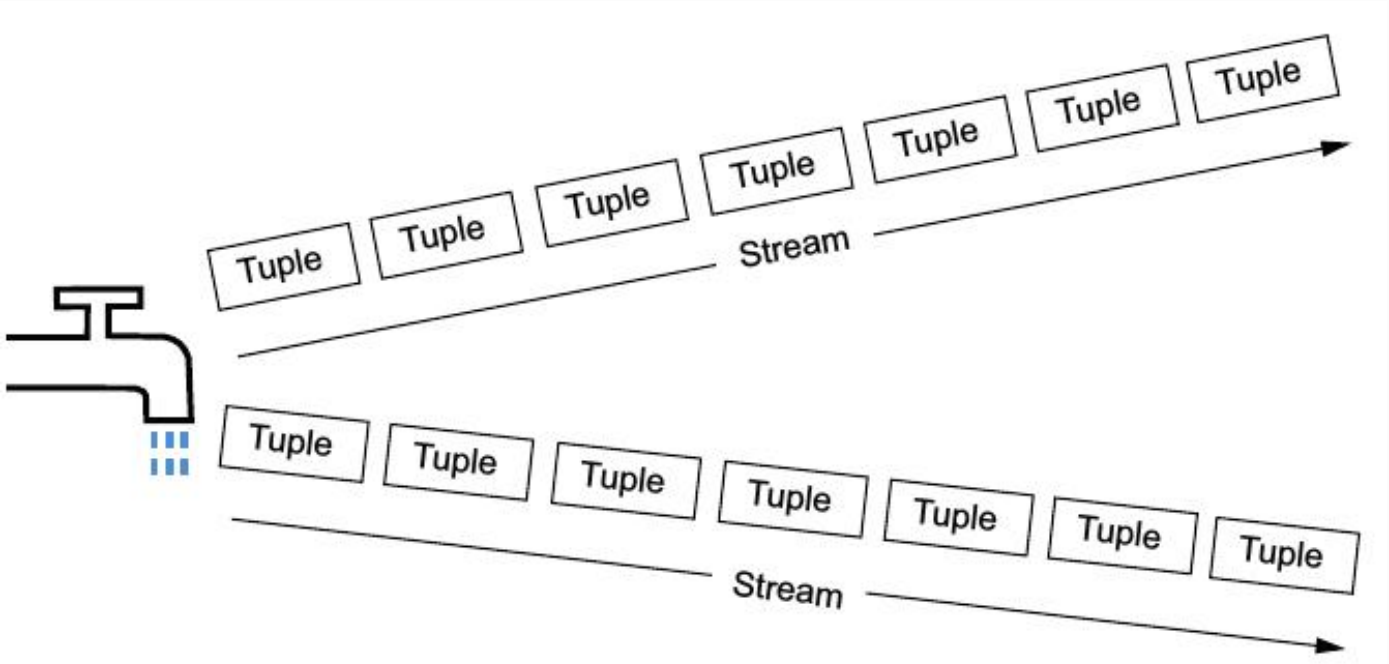
Tuple

- Core Unit of Data
- Immutable set of key/value pairs.

Stream

- Unbounded sequence of tuples

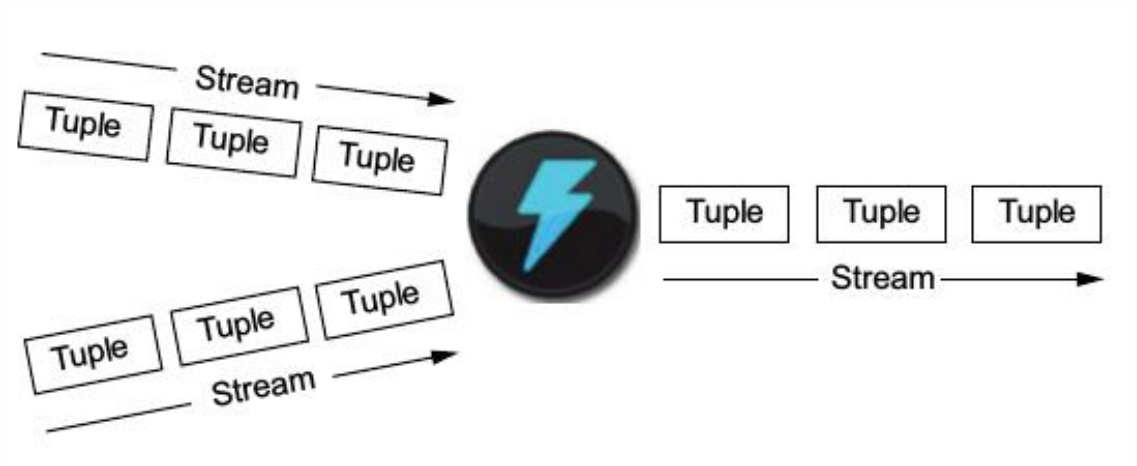
STORM CONCEPTS



Spout = data source

- Emits streams.
- Example: a Kafka spout on a particular topic.

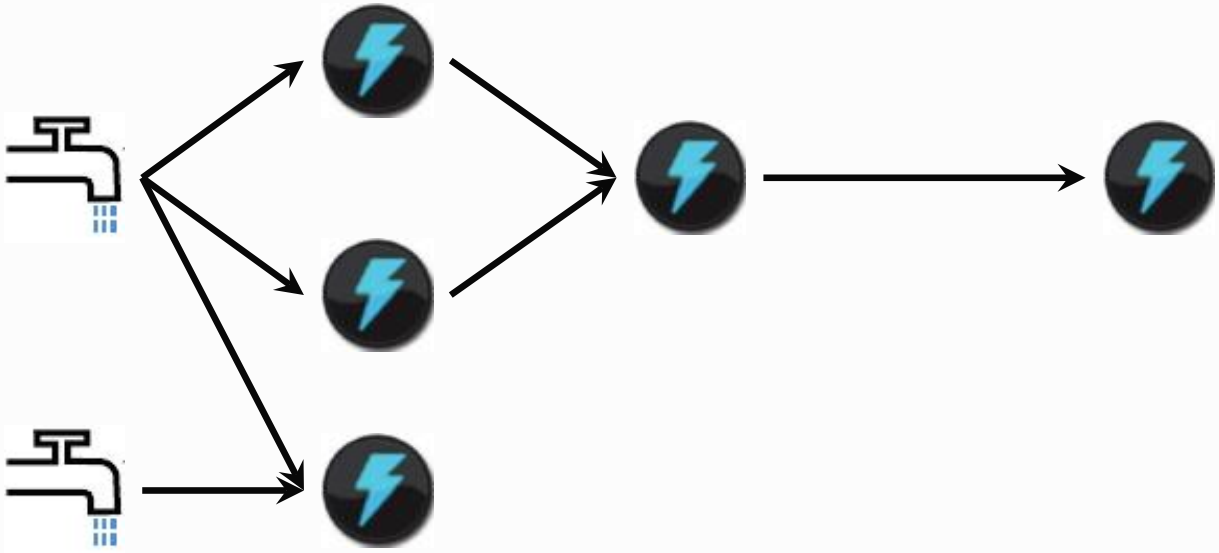
STORM CONCEPTS



Bolt = Core function of streaming computation

- Receive tuples and process them, e.g.:
  - Write to a data store
  - Lookup a tuple in a data store
  - Perform arbitrary computation
  - (Usually, but not necessarily) Emit additional streams

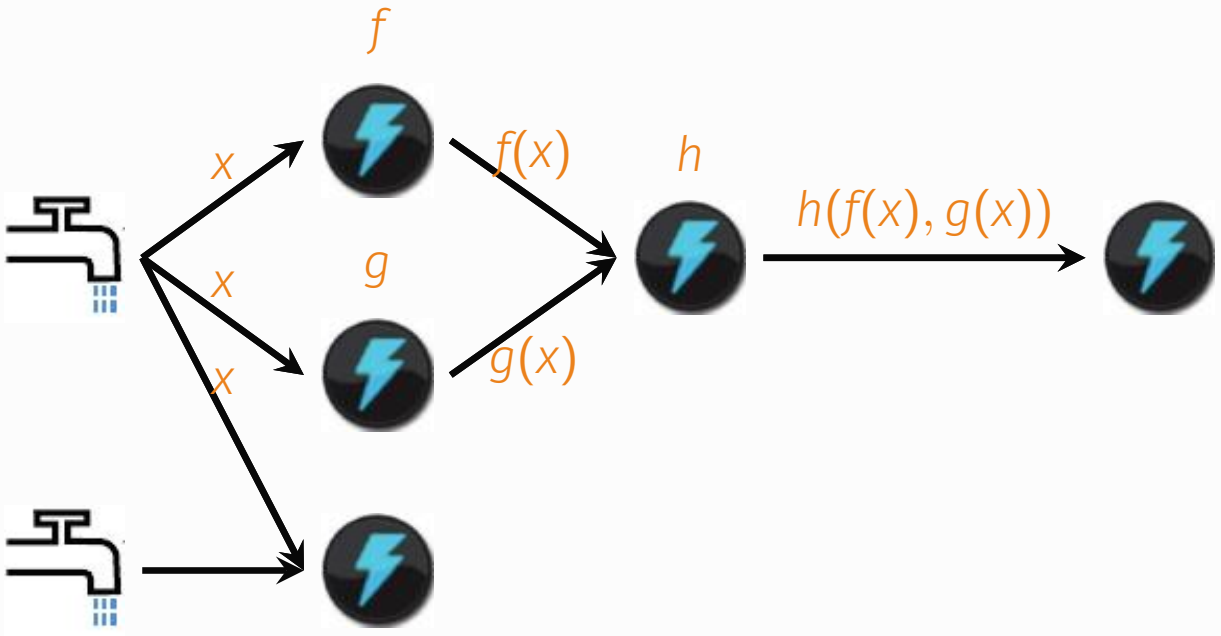
STORM CONCEPTS



Topology = DAG of Spouts, Bolts, and Streams

- Data Flow representation of computation.
- Tuples are pushed through the DAG (streaming computation) starting from spouts.

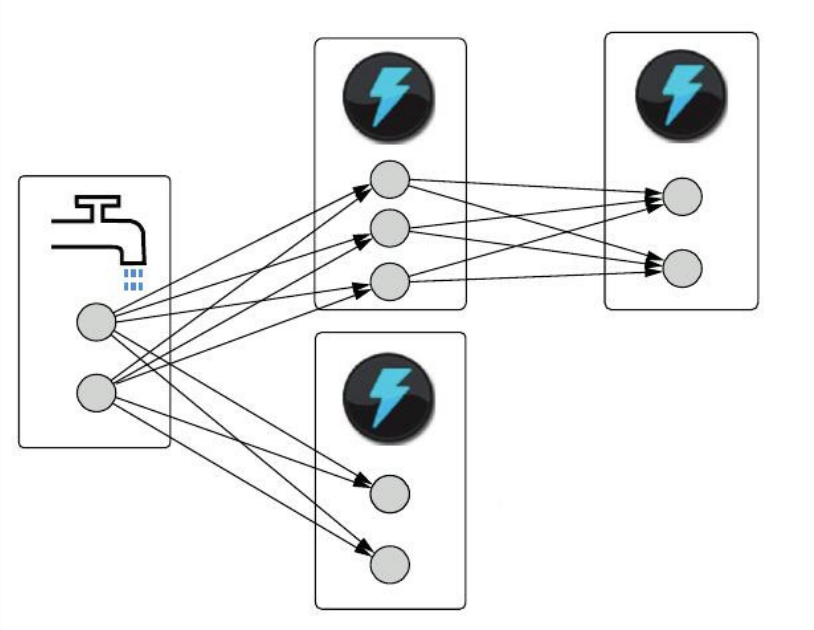
STORM CONCEPTS



Topology = DAG of Spouts, Bolts, and Streams

- Data Flow representation of computation.
- Tuples are pushed through the DAG (streaming computation) starting from spouts.

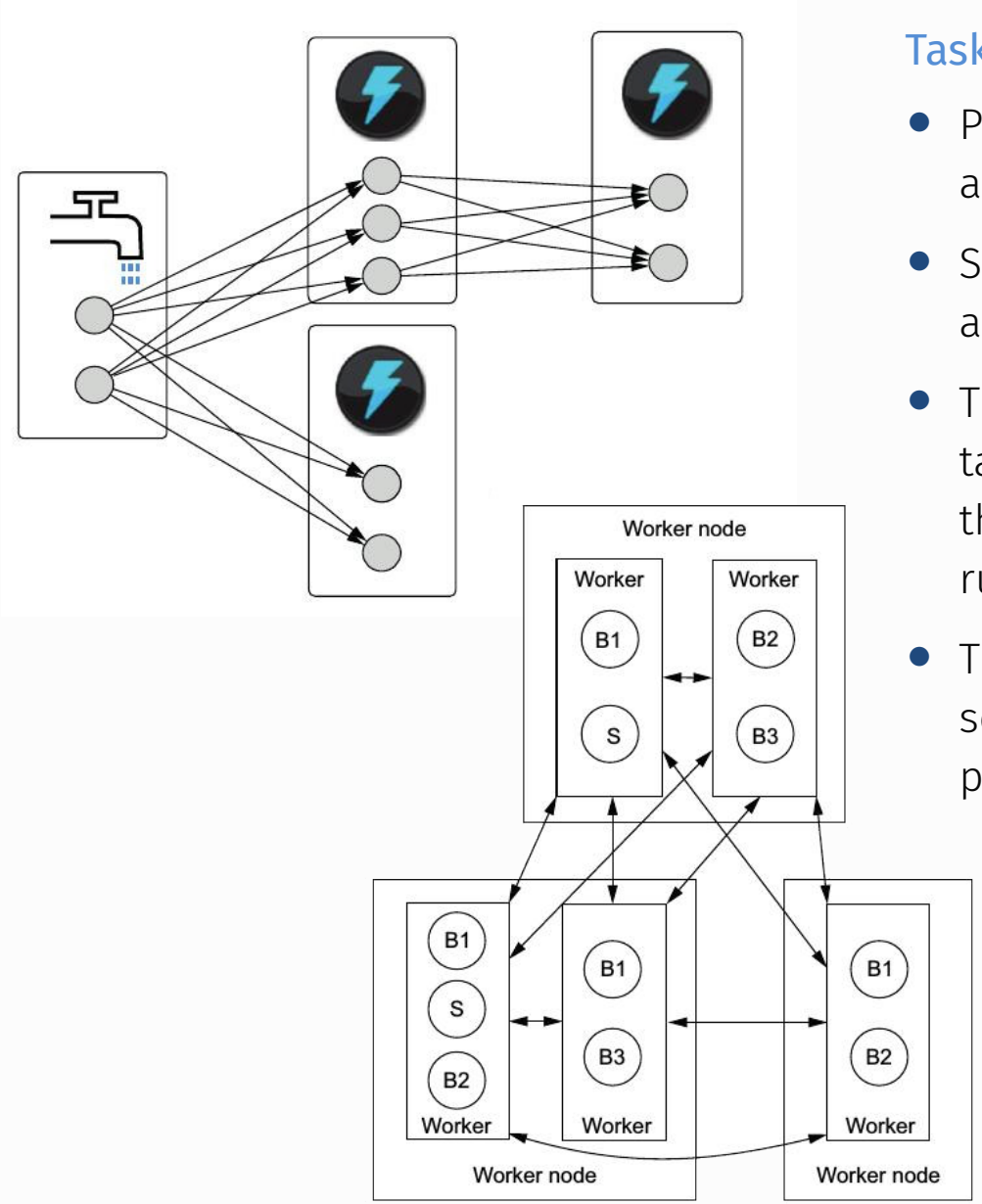
STORM CONCEPTS



Tasks

- Parallel/Distributed execution of Spouts and Bolts.
- Spouts and Bolts execute as many tasks across the cluster.
- The programmer defines the number of tasks for each Spout/Bolt when defining the topology (can be re-configured at runtime).

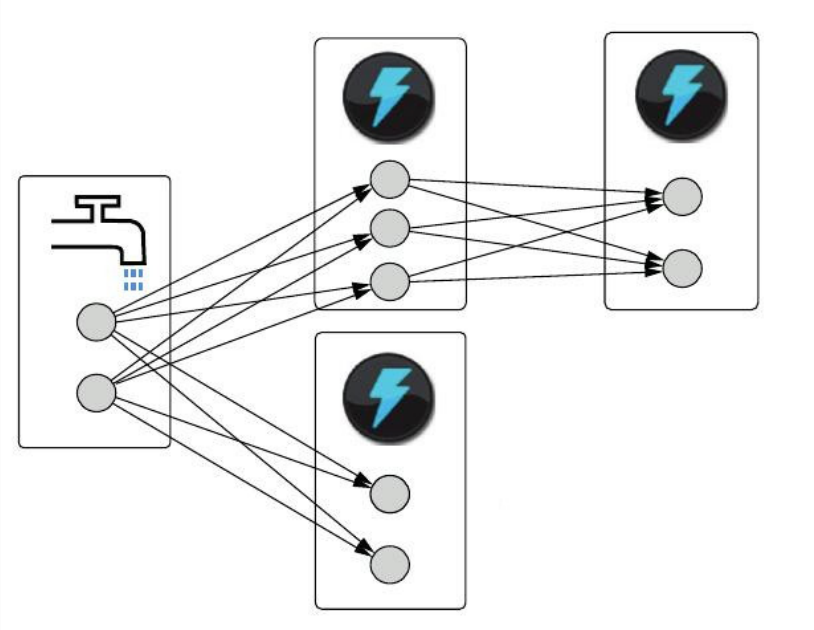
STORM CONCEPTS



Tasks

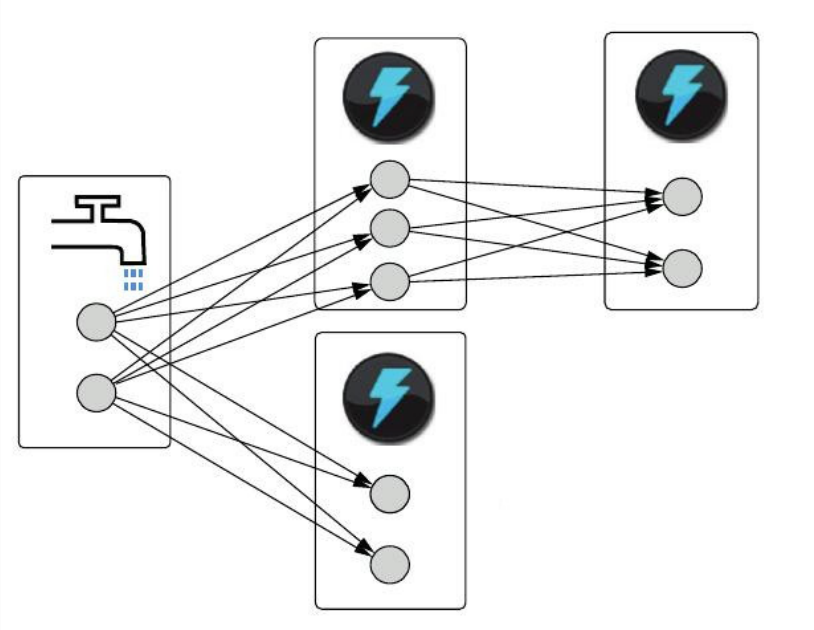
- Parallel/Distributed execution of Spouts and Bolts.
- Spouts and Bolts execute as many tasks across the cluster.
- The programmer defines the number of tasks for each Spout/Bolt when defining the topology (can be re-configured at runtime).
- The Storm Scheduler is responsible for scheduling these tasks across the physical machines in the cluster.

STORM CONCEPTS



Question: When a tuple is emitted, what task does it go to?

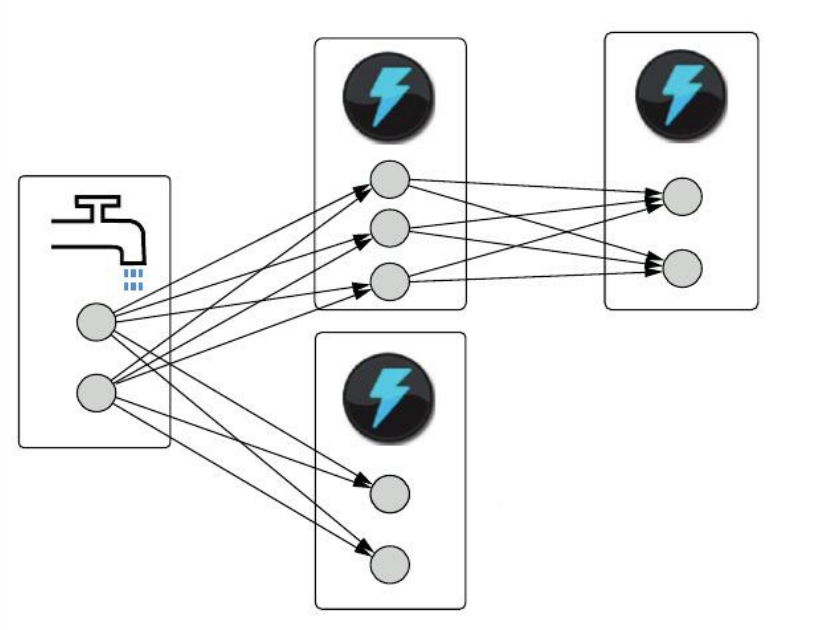
STORM CONCEPTS

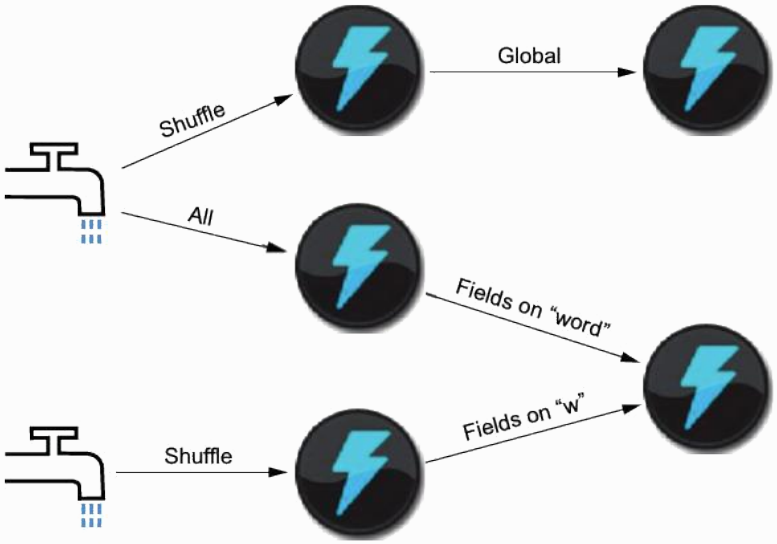


Question: When a tuple is emitted, what task does it go to?

Answer: Depends on the grouping specified by the topology programmer.

STORM CONCEPTS





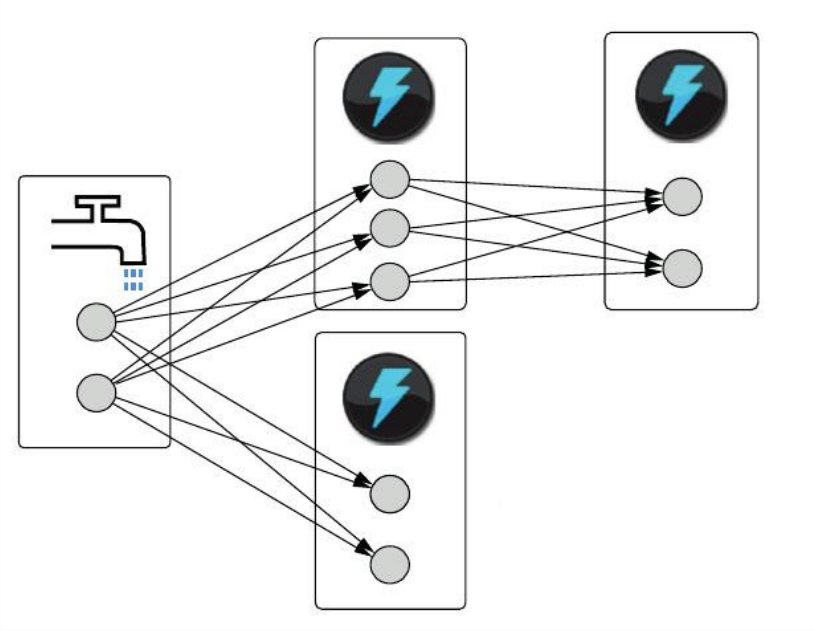
Question: When a tuple is emitted, what task does it go to?

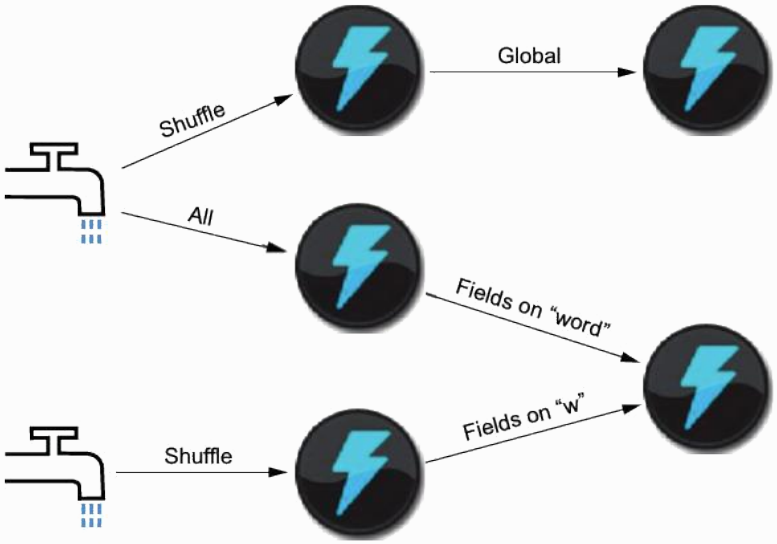
Answer: Depends on the grouping specified by the topology programmer.

Stream grouping

- Shuffle grouping: pick task in round-robin fashion

STORM CONCEPTS





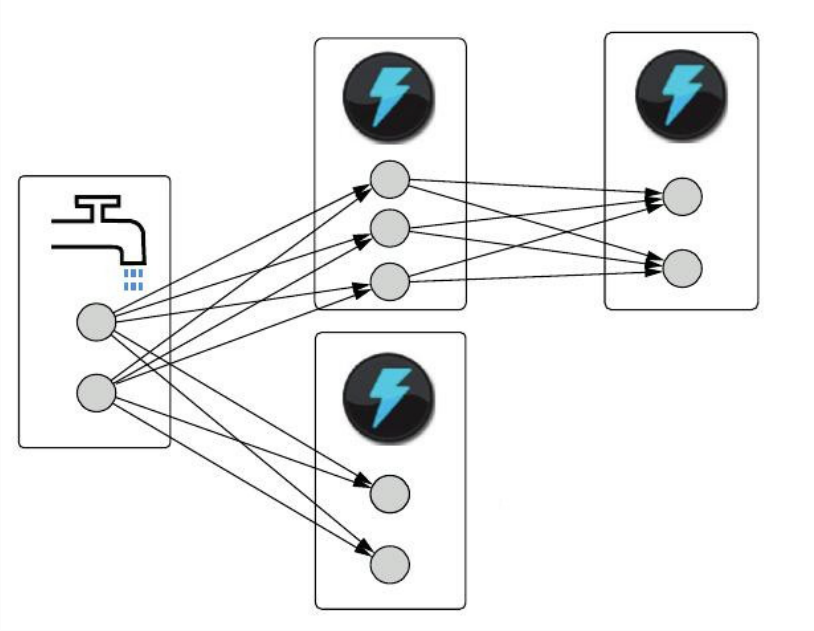
Question: When a tuple is emitted, what task does it go to?

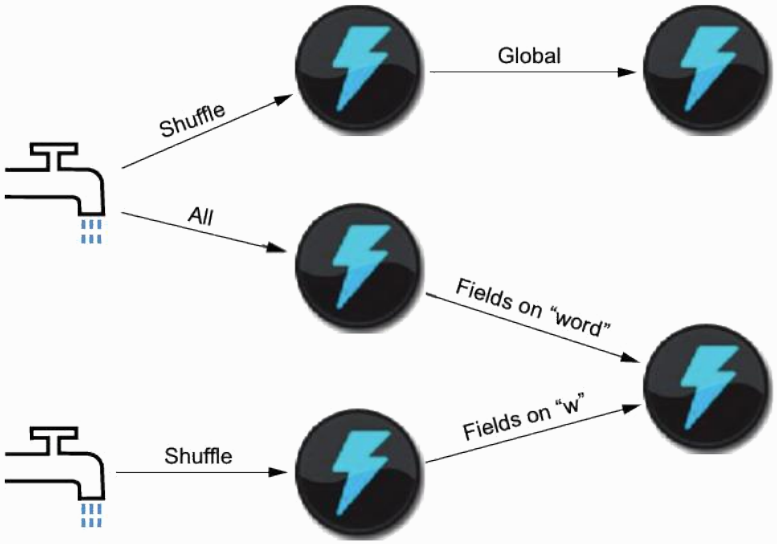
Answer: Depends on the grouping specified by the topology programmer.

Stream grouping

- Shuffle grouping: pick task in round-robin fashion
- Field grouping: consistent hashing on a subset of tuple fields

STORM CONCEPTS





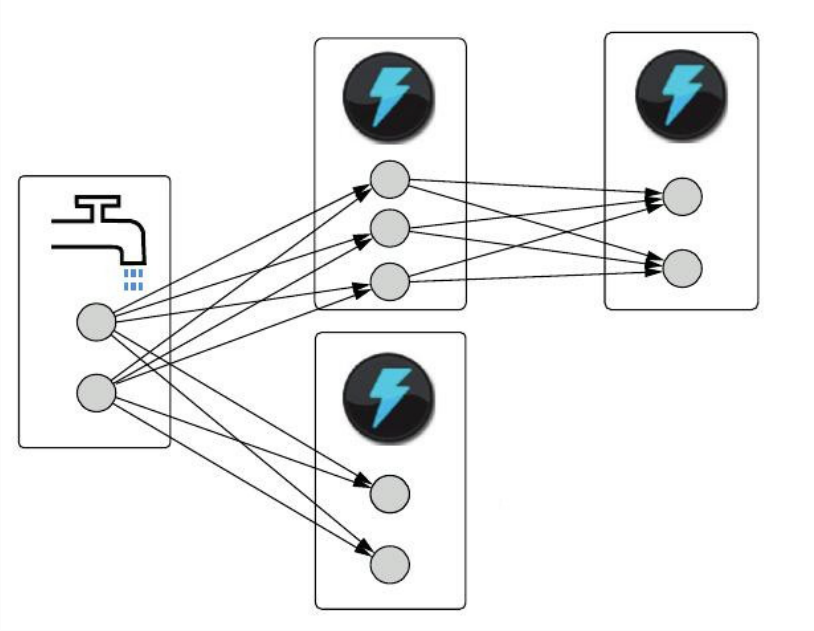
Question: When a tuple is emitted, what task does it go to?

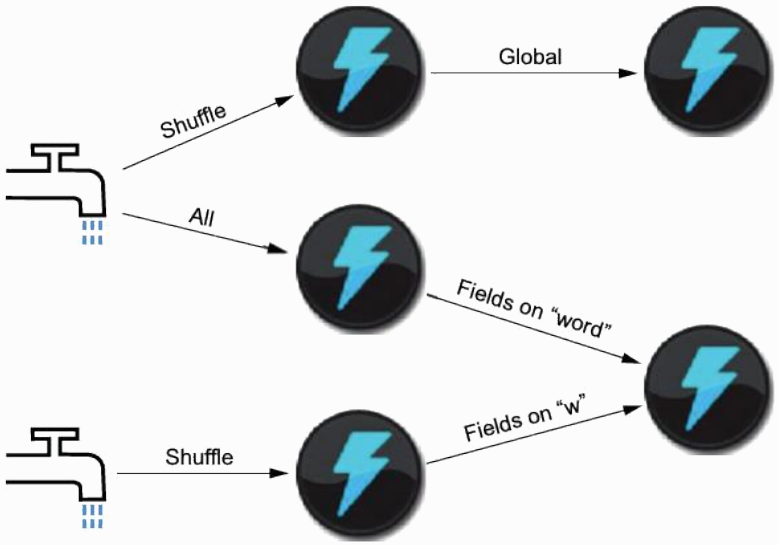
Answer: Depends on the grouping specified by the topology programmer.

Stream grouping

- Shuffle grouping: pick task in round-robin fashion
- Field grouping: consistent hashing on a subset of tuple fields
- All grouping: send to all tasks

STORM CONCEPTS





Question: When a tuple is emitted, what task does it go to?

Answer: Depends on the grouping specified by the topology programmer.

Stream grouping

- Shuffle grouping: pick task in round-robin fashion
- Field grouping: consistent hashing on a subset of tuple fields
- All grouping: send to all tasks
- Global grouping: pick task with lowest id.

STORM API

Spout API

```
public interface ISpout extends Serializable {
        void open(Map conf,
                  TopologyContext context,
                  SpoutOutputCollector collector);

        void close();

        void activate();

        void deactivate();

        void nextTuple();

        void ack(Object msgId);

        void fail(Object msgId)
}
```

STORM API

Spout API

```
public interface ISpout extends Serializable {
        void open(Map conf,
                  TopologyContext context,
                  SpoutOutputCollector collector);

        void close();

        void activate();

        void deactivate();

        void nextTuple();

        void ack(Object msgId);

        void fail(Object msgId)
}
```

Lifecycle API

STORM API

Spout API

```
public interface ISpout extends Serializable {
        void open(Map conf,
                  TopologyContext context,
                  SpoutOutputCollector collector);

        void close();

        void activate();

        void deactivate();

        void nextTuple();

        void ack(Object msgId);

        void fail(Object msgId)
}
```

Core API

STORM API

Spout API

```
public interface ISpout extends Serializable {
        void open(Map conf,
                  TopologyContext context,
                  SpoutOutputCollector collector);

        void close();

        void activate();

        void deactivate();

        void nextTuple();

        void ack(Object msgId);

        void fail(Object msgId)
}
```

Reliability API

STORM API

Bolt API

```
public interface IBolt extends Serializable {
        void prepare(Map stormConf,
                     TopologyContext context,
                     OutputCollector collector);

        void cleanup();

        void execute(Tuple input);
}

public interface IBasicBolt extends IComponent {
        void prepare(Map stormConf, TopologyContext context)
        void cleanup()

        void execute(Tuple input, BasicOutputCollector collector)
}
```

STORM API

Bolt API

```
public interface IBolt extends Serializable {
        void prepare(Map stormConf,
                     TopologyContext context,
                     OutputCollector collector);

        void cleanup();

        void execute(Tuple input);
}

public interface IBasicBolt extends IComponent {
        void prepare(Map stormConf, TopologyContext context)
        void cleanup()

        void execute(Tuple input, BasicOutputCollector collector)
}
```

Lifecycle API

STORM API

Bolt API

```
public interface IBolt extends Serializable {
        void prepare(Map stormConf,
                     TopologyContext context,
                     OutputCollector collector);

        void cleanup();

        void execute(Tuple input);
}

public interface IBasicBolt extends IComponent {
        void prepare(Map stormConf, TopologyContext context)
        void cleanup()

        void execute(Tuple input, BasicOutputCollector collector)
}
```

Core API

STORM API

Bolt Output API

```
public interface IOutputCollector extends IErrorReporter {
        List<Integer> emit(String streamId,
                           Collection<Tuple> anchors,
                           List<Object> tuple);

        void emitDirect(int taskId,
                        String streamId,
                        Collection<Tuple> anchors,
                        List<Object> tuple);

        void ack(Tuple input);

        void fail(Tuple input);
}
```

STORM API

Bolt Output API

```
public interface IOutputCollector extends IErrorReporter {
        List<Integer> emit(String streamId,
                           Collection<Tuple> anchors,
                           List<Object> tuple);

        void emitDirect(int taskId,
                        String streamId,
                        Collection<Tuple> anchors,
                        List<Object> tuple);

        void ack(Tuple input);

        void fail(Tuple input);
}
```

Core API

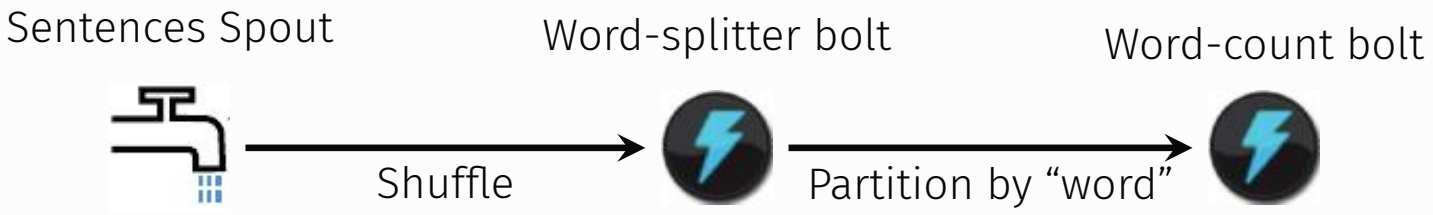
STORM API

Bolt Output API

```
public interface IOutputCollector extends IErrorReporter {
        List<Integer> emit(String streamId,
                           Collection<Tuple> anchors,
                           List<Object> tuple);

        void emitDirect(int taskId,
                        String streamId,
                        Collection<Tuple> anchors,
                        List<Object> tuple);

        void ack(Tuple input);

        void fail(Tuple input);
}
```

Reliability API

STORM: WORD COUNT EXAMPLE (JAVA)




```
/** ... some contents removed ...*/

TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("spout1", new KafkaSpout(...) );

builder.setBolt("splitterBolt", new SplitSentenceBolt(), 8)
       .shuffleGrouping("spout1");

builder.setBolt("countBolt", new WordCountBolt(), 12)
       .fieldsGrouping("splitterBolt", new Fields("word"));
```

STORM: WORD COUNT EXAMPLE (CONT)

```
public class SplitSentenceBolt implements IRichBolt {

    private OutputCollector _collector;

    public void prepare(Map stormConf, TopologyContext context,
                 OutputCollector collector) {
        _collector = collector;
    }

    public void execute(Tuple tuple) {
        String sentence = tuple.getString(0);
        for (String word: sentence.split("␣")) {
            _collector.emit(tuple, new Values(word));
        }
        _collector.ack(tuple);
     }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("word"));
    }

}
```

STORM: WORD COUNT EXAMPLE (CONT)

```
public class WordCountBolt extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();

    public void execute(Tuple tuple, BasicOutputCollector collector) {
      String word = tuple.getString(0);
      Integer count = counts.get(word);
      if (count == null)
        count = 0;
      count++;
      counts.put(word, count);
      collector.emit(new Values(word, count));
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("word", "count"));
    }
}
```

GUARANTEEING MESSAGE PROCESSING

```
public class SplitSentenceBolt implements IRichBolt {

    private OutputCollector _collector;

    public void prepare(Map stormConf, TopologyContext context,
                 OutputCollector collector) {
        _collector = collector;
    }

    public void execute(Tuple tuple) {
        String sentence = tuple.getString(0);
        for (String word: sentence.split("␣")) {
            _collector.emit(tuple, new Values(word));
        }
        _collector.ack(tuple);
     }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("word"));
    }

}
```

GUARANTEEING MESSAGE PROCESSING

```
public class SplitSentenceBolt implements IRichBolt {

    private OutputCollector _collector;

    public void prepare(Map stormConf, TopologyContext context,
                 OutputCollector collector) {
        _collector = collector;
    }

    public void execute(Tuple tuple) {
        String sentence = tuple.getString(0);
        for (String word: sentence.split(" ")) {
            _collector.emit(tuple, new Values(word));
        }
        _collector.ack(tuple);
     }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("word"));
    }

}
```

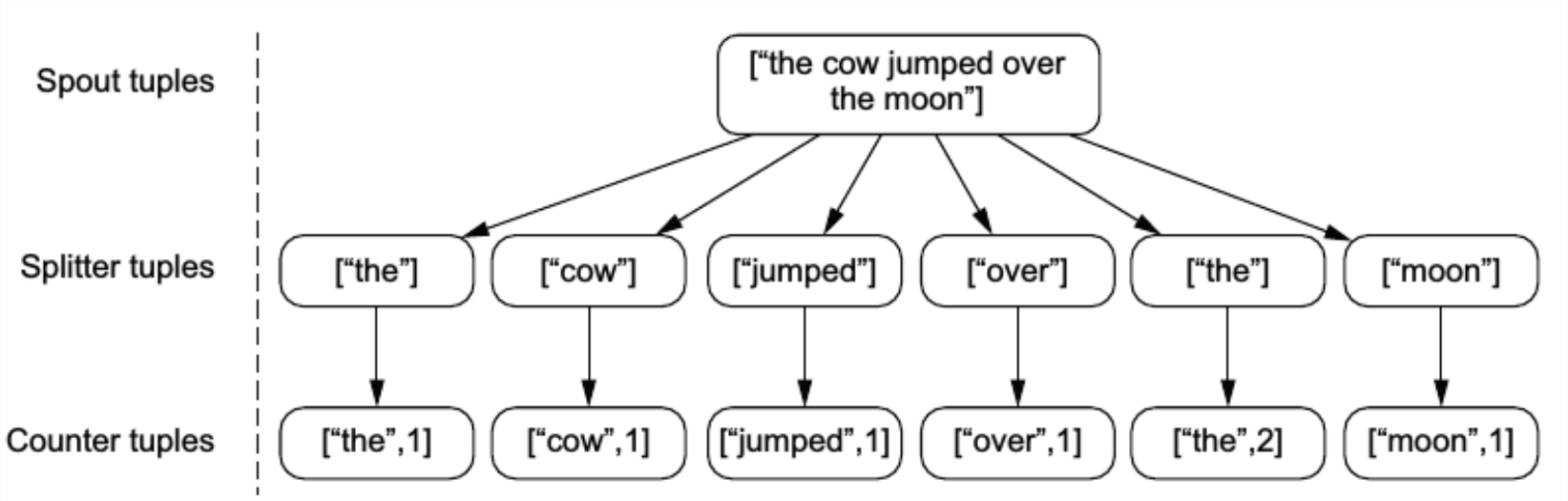
"Anchors" tuple to be a parent of word.

GUARANTEEING MESSAGE PROCESSING

```
public class SplitSentenceBolt implements IRichBolt {

    private OutputCollector _collector;

    public void prepare(Map stormConf, TopologyContext context,
                 OutputCollector collector) {
        _collector = collector;
    }

    public void execute(Tuple tuple) {
        String sentence = tuple.getString(0);
        for (String word: sentence.split("␣")) {
            _collector.emit(tuple, new Values(word));
        }
        _collector.ack(tuple);
     }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("word"));
    }

}
```

Confirm `tuple` correctly processed

GUARANTEEING MESSAGE PROCESSING



- A spout tuple is not fully processed until all tuples that "descend" from it have been completely processed.
- The descendant-relationship (encoded by `emit`) is recorded in a tuple tree.
- If the tuple tree is not completed within a specified timeout, the spout tuple is replayed. This is known as at least once processing semantics.
- Question: Is replaying a tuple always "safe"?

CONCLUSION

Apache Storm/Heron

- Distributed and Fault-Tolerant real-time computation
- Tuple-at-a-time processing model

Noteworthy:

- Twitter used a domain specific language (DSL) called Summingbird to specify computations in a declarative manner, and compile this both to M/R and Storm/Heron Topologies.

SPEED LAYER: MINI-BATCHING

SPARK STREAMING

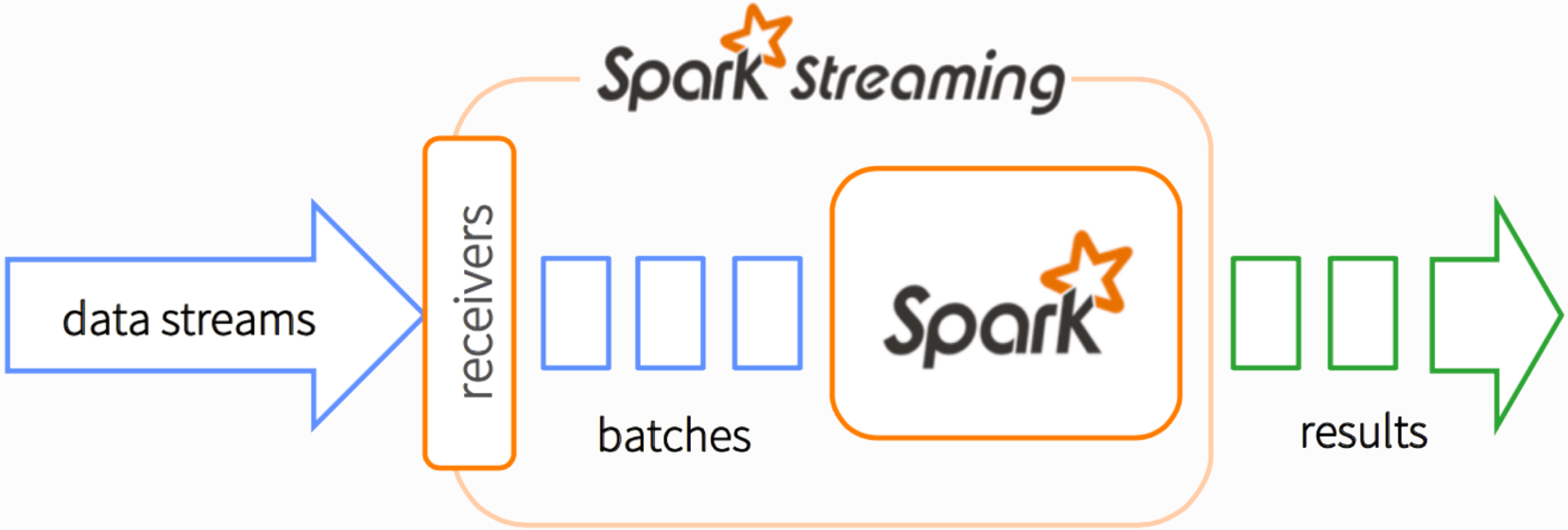
Spark Streaming

- Distributed and Fault-Tolerant (near) real-time computation
- Developed as a separate library on top of Spark in 2015
- Re-uses Spark concepts and programming model (RDDs, functional transformations, failure recovery through re-execution).

Key difference with Storm:

- Processes stream tuples in (mini)-batches.
- Has exactly-once semantics.

MINI-BATCHING IN SPARK STREAMING



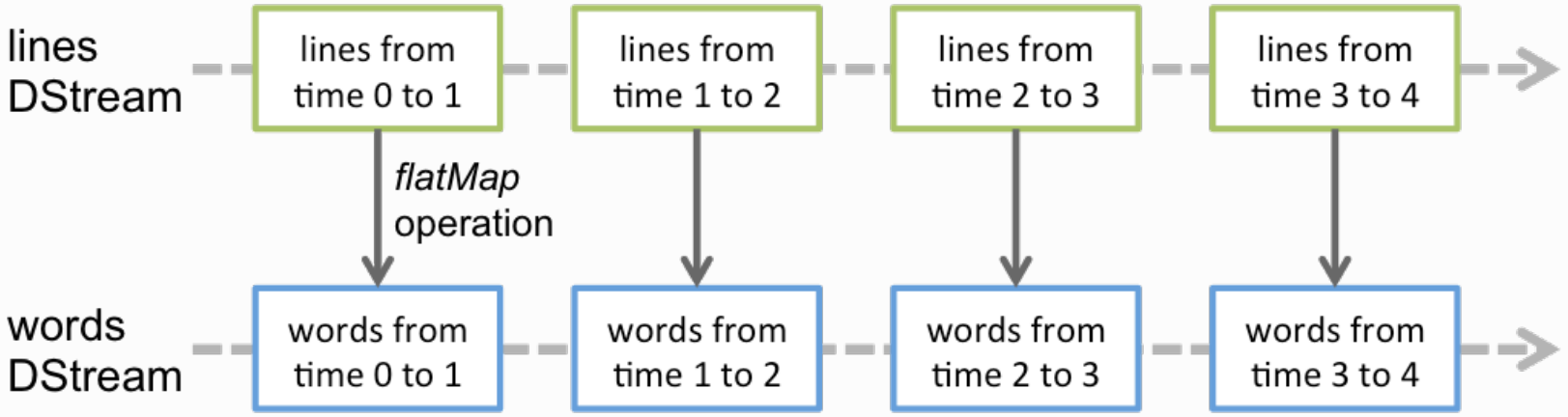
Mini-batching

- Spark streaming partitions the input stream into disjoint time intervals
- The data in each time interval becomes a (mini-batch) RDD; the stream of mini-batch RDDs is called a Discretized Stream (DStream)
- Mini-batch RDDs are normal RDDs. Therefore, normal spark operators can be used to transform/act on mini-batch RDDs (possibly jointly with normal RDDs)
- Each transformed mini-batch RDD can then be saved to HDFS, or stored in a DB, or communicated to a message queue, ...

SPARK STREAMING WORD COUNT EXAMPLE (SCALA)

```scala
// Create a local StreamingContext with two working thread and batch
// interval of 1 second.
// The master requires 2 cores to prevent from a starvation scenario.
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))

// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)

// Split each line into words
val words = lines.flatMap(_.split("␣"))

// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

// Print the first ten elements of each RDD generated in this DStream
//  to the console
wordCounts.print()

ssc.start()             // Start the computation
ssc.awaitTermination()  // Wait for the computation to terminate
```

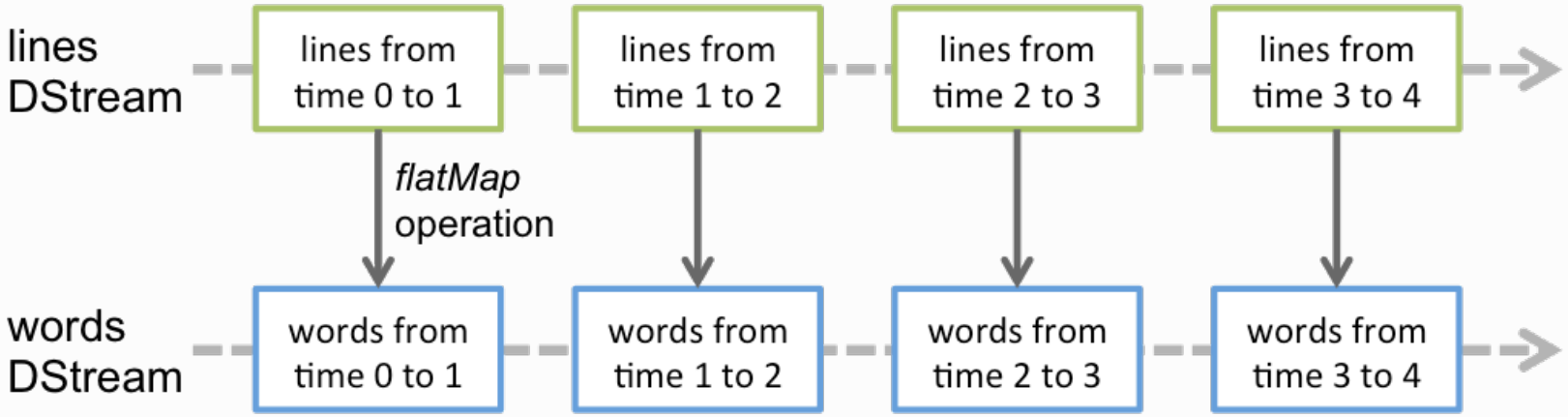
SPARK STREAMING CONCEPTS



DStream

- Stream of mini-batch RDDs (interval width is configurable)
- Operations on the stream (`flatMap, filter, …`) are performed on each mini-batch RDD in the DStream separately.
- The result is a new DStream.

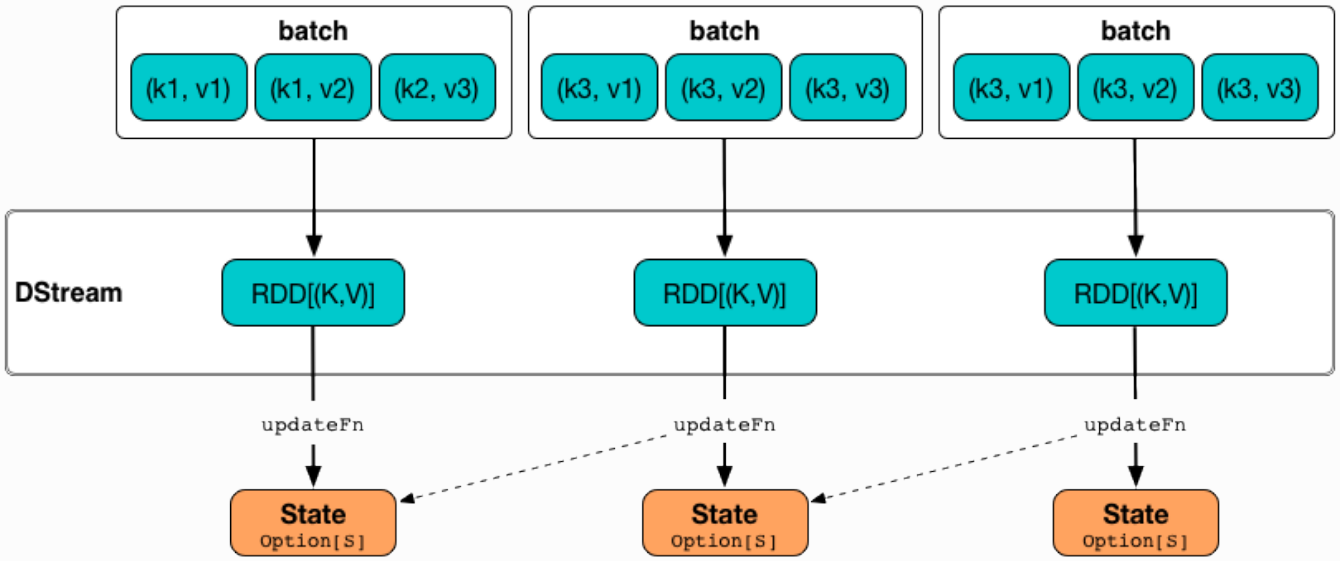
SPARK STREAMING CONCEPTS



Operations on DStreams

- `print`: print the first 10 items of each mini-batch RDD in the DStream (debugging)
- `saveAsTextFiles`: save each mini-batch RDD as a separate text file
- `foreachRDD(func)`: apply arbitrary function `func` to each RDD in the stream.
- ...

MAINTAINING STATE




```
updateStateByKey(updateFn)
updateFn: (newValues, oldState) => newState
```

Question: Assume that we want to maintain the count of every word seen so far, and update this count whenever new data arrives. How do we do this ?

Answer: Use `updateStateByKey` or `mapWithState` (both on pair DStreams)

UPDATESTATEBYKEY EXAMPLE (SCALA)

```
...

// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)

// Split each line into words
val words = lines.flatMap(_.split("␣"))

// Count each word in each batch
val pairs = words.map(word => (word, 1))

// create a DStream in which each RDD has the total count for every word
// ever seen
val globalCount = pairs.updateStateByKey( (vals, totalCount) => {
     totalCount.match {
       //Was there already some state for this key? If so, update
       case Some(total) => vals.sum + total
       //Otherwise, create the state
       case None => vals.sum
     }
})
```

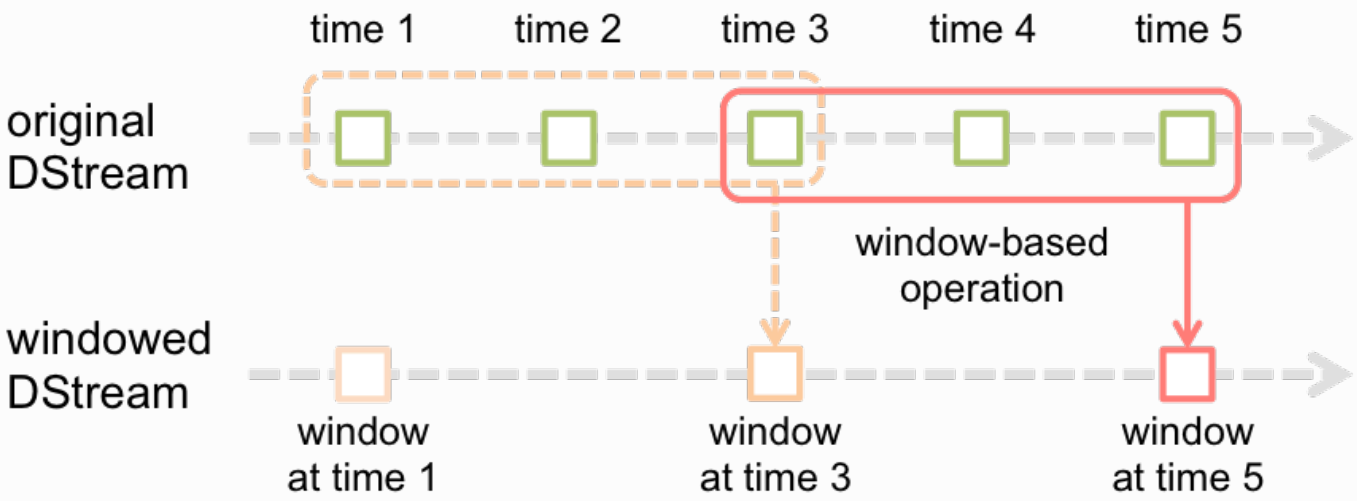
MAPWITHSTATE EXAMPLE (SCALA)

```scala
// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)

// Split each line into words
val words = lines.flatMap(_.split("␣"))

// Count each word in each batch
val pairs = words.map(word => (word, 1))


// Specify the state-update function for mapWithState
// word: the key for which we need to update the state;
val mappingFunc = (word: String, one: Option[Int], state: State[Int]) => {
   val sum = one.getOrElse(0) + state.getOption.getOrElse(0)
   val output = (word, sum)
   state.update(sum)  //updates the count for this key (word)
   output  //we need to output the new total for this word
}

// create a DStream in which each RDD has the total count for every word
// ever seen. StateSpec allows fine-grained tuning (# of partitions, ...)
val globalCount = pairs.mapWithState(StateSpec.function(mappingFunc))
```

WINDOWING

Question: Assume that we want to maintain a running count of every word seen over the past 10 minutes, and update this running count whenever new data arrives. How do we do this ?

WINDOWING

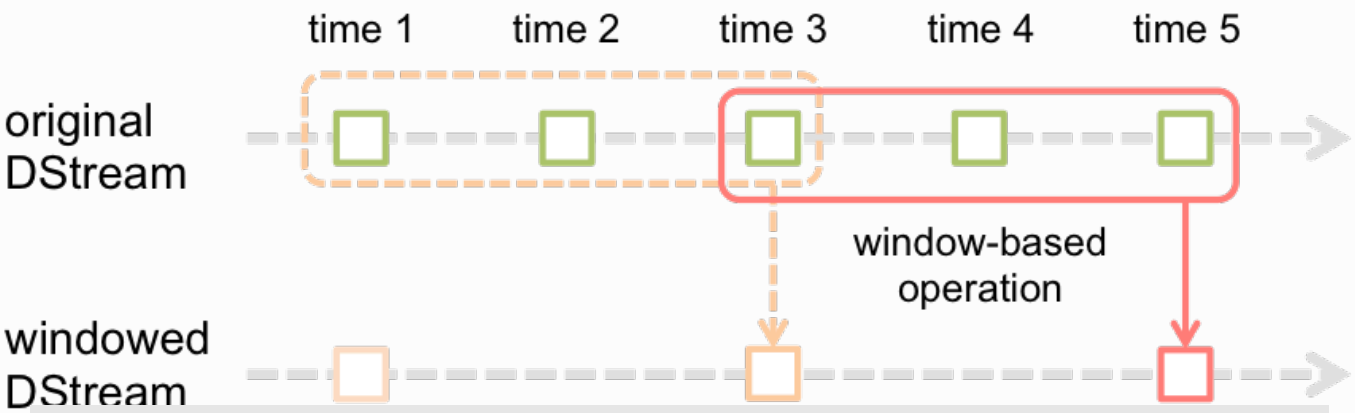


Question: Assume that we want to maintain a running count of every word seen over the past 10 minutes, and update this running count whenever new data arrives. How do we do this ?

Answer: Use windowing operations:

- Every window has a length (3 in figure) and sliding interval (2 in the figure), both must be multiples of the batch interval
- Window length = the duration of the window
- Sliding interval = the interval at which the window operation is performed

WINDOWING



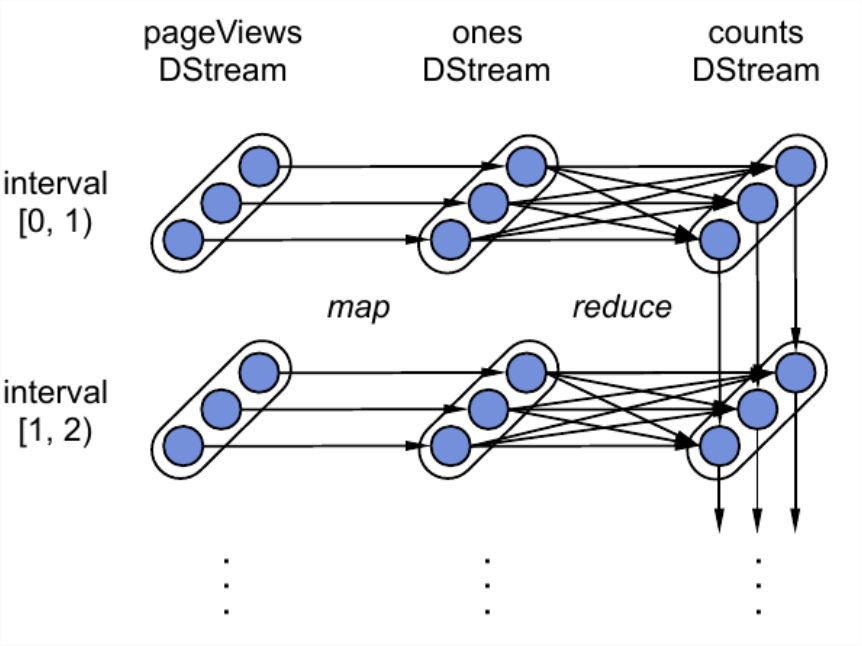

```
// Reduce last 30 seconds of data, every 10 seconds
val windowedWordCounts = pairs.reduceByKeyAndWindow(
    (a:Int,b:Int) => (a + b), Seconds(30), Seconds(10))
```

Question: Assume that we want to maintain a running count of every word seen over the past 10 minutes, and update this running count whenever new data arrives. How do we do this ?

Answer: Use windowing operations:

- Every window has a length (3 in figure) and sliding interval (2 in the figure), both must be multiples of the batch interval
- Window length = the duration of the window
- Sliding interval = the interval at which the window operation is performed

FAULT-TOLERANCE SEMANTICS

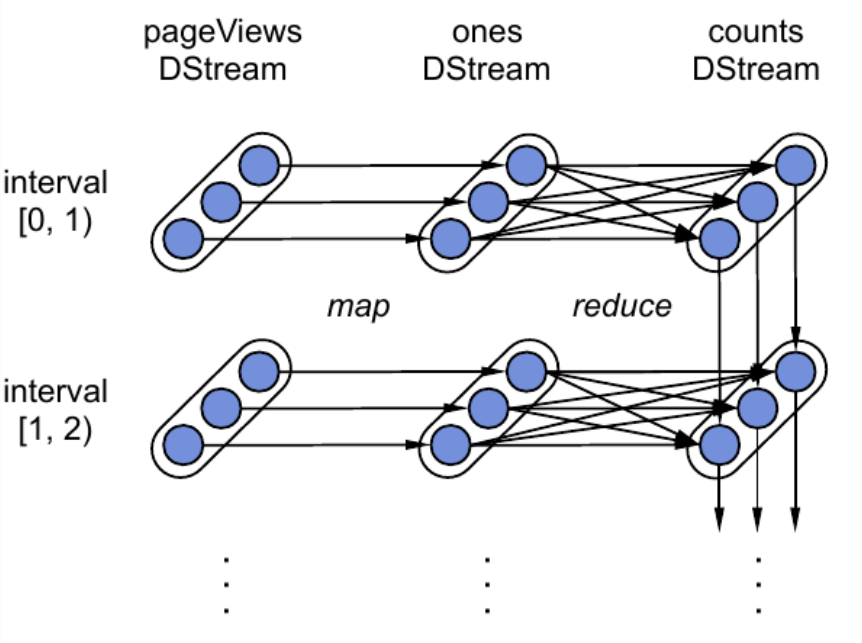


Example lineage graph. Each oval is an RDD and each blue circle an RDD partition. Edges indicate lineage dependency.

Remember

- RDDs are immutable and deterministically re-computable given the RDD's lineage graph (graph of transformations)
- Likewise, if we track lineage of operations on mini-batch RDDs in a DStream, then we can re-compute any such RDD when needed.

FAULT-TOLERANCE SEMANTICS

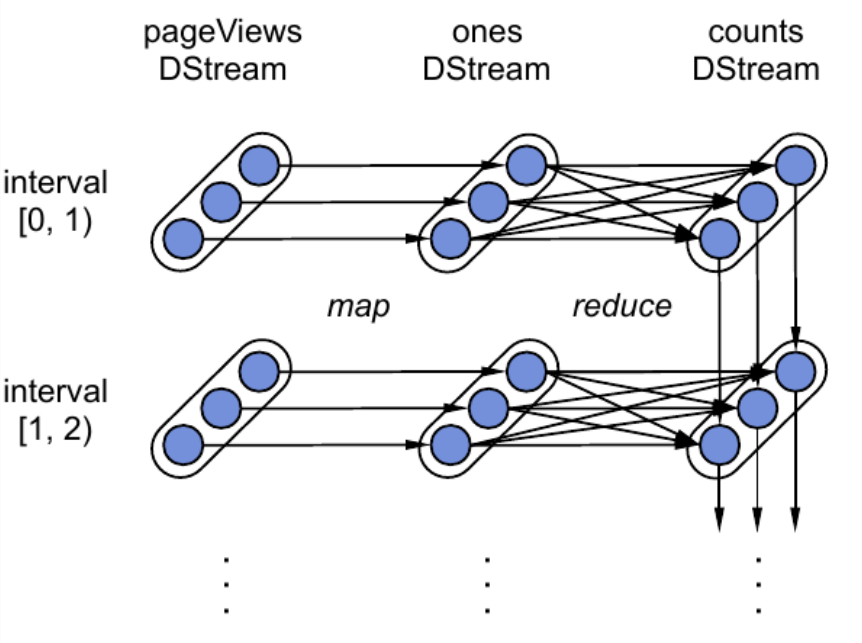


Example lineage graph. Each oval is an RDD and each blue circle an RDD partition. Edges indicate lineage dependency.

Some caveats:

- Lineage graph may become very big, especially when state is maintained.
- Add checkpointing to allow truncating the lineage graph
- Recomputation means that input data must be available. To ensure this, spark replicates input stream data.
- Enable write ahead logging to ensure no data loss.

FAULT-TOLERANCE SEMANTICS

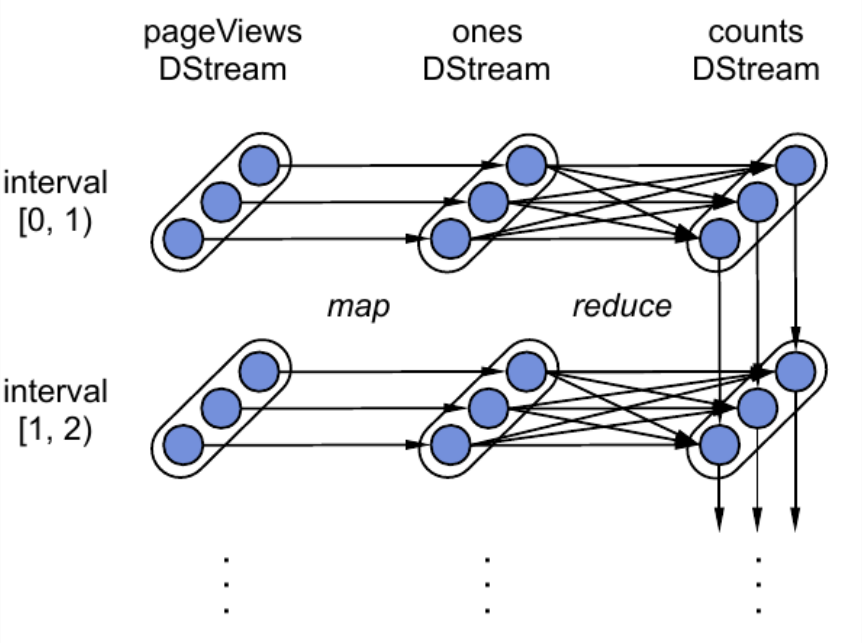


Example lineage graph. Each oval is an RDD and each blue circle an RDD partition. Edges indicate lineage dependency.

Possible processing guarantees:

- At most once: Each record will be either processed once or not processed at all.
- At least once: Each record will be processed one or more times. There may be duplicates.
- Exactly once: Each record will be processed exactly once - no data will be lost and no data will be processed multiple times.

FAULT-TOLERANCE SEMANTICS



Example lineage graph. Each oval is an RDD and each blue circle an RDD partition. Edges indicate lineage dependency.

Spark Streaming provides Exactly-Once semantics

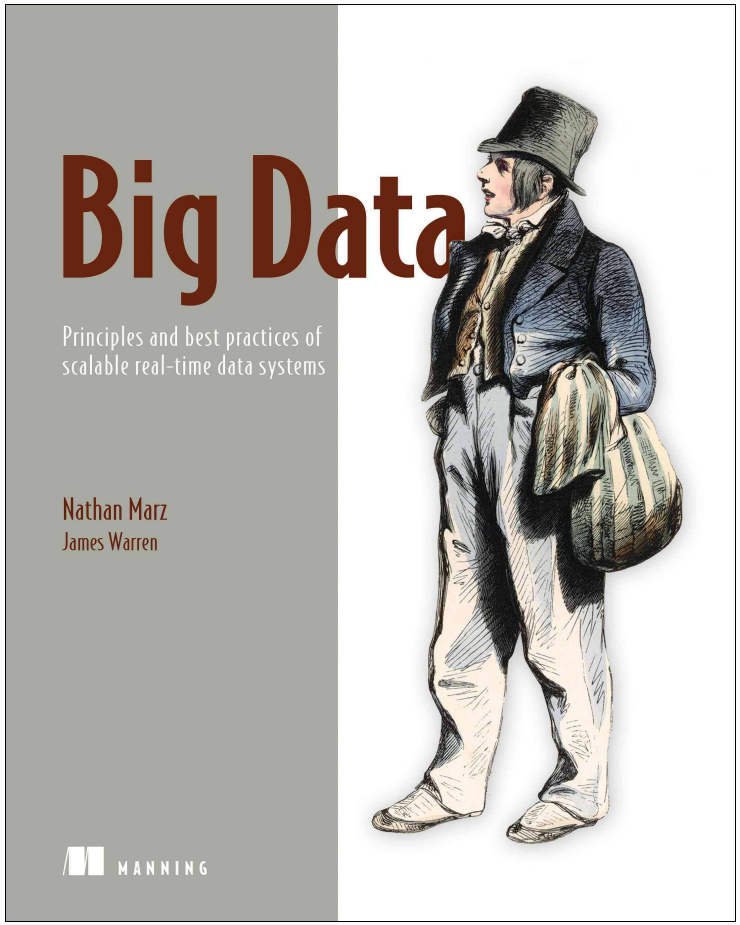
- But this is only for data transformations
- If you want end-to-end exactly-once semantics, your data source and your data sink should also have this semantics. (Which is the case e.g. with Kafka).

REFERENCES

- N. Marz, J. Warren. *Big Data: Principles and best practices of scalable realtime data systems* Manning Publications, April 2015.
- O. Boykin, S. Ritchi, I. O'Connel, and Jimmy Lin. *Summingbird: A Framework for Integrating Batch and Online MapReduce Computations.* Proceedings of VLDB, 2014.
- S. Kulkarni, et al. *Twitter Heron: Stream Processing at Scale.* SIGMOD Conference, 2015.
- M. Zaharia et al. *Discretized Streams: Fault-Tolerant Streaming Computation at Scale.* SOSP Conference, 2013.

Questions?

ACKNOWLEDGEMENTS



Based on content created by Stijn Vansummeren and on material taken from the book “Big Data: Principles and best practices of scalable realtime data systems” by Nathan Marz and James Warren April 2015, Manning Publications, April 2015.