Parallel Computing

Prof. Marco Bertini

Data parallelism: Lambda architecture

Big data and parallelism

- Big Data differs from traditional data processing through its use of parallelism — only by bringing multiple computing resources together we can process terabytes of data.

- In this lecture we are going to analyze the Lambda Architecture.
  This architecture, originally proposed by Nathan Merz combines the large-scale batch-processing strengths of MapReduce with the real-time responsiveness of stream processing

- The goal is to create scalable, responsive, and fault-tolerant solutions to Big Data problems.

Big data and parallelism

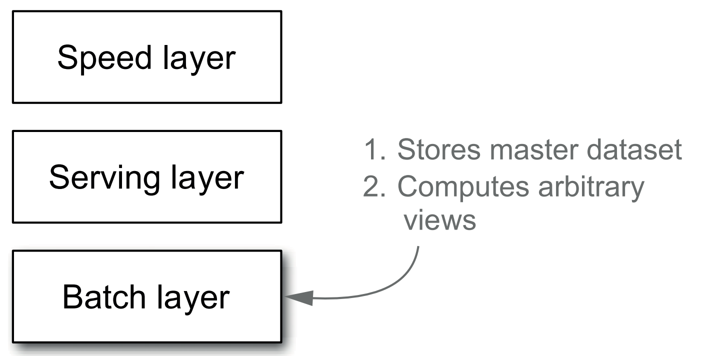
- The Lambda Architecture leverages data parallelism.

- It does so on a huge scale, distributing both data and computation over clusters of tens or hundreds of machines.

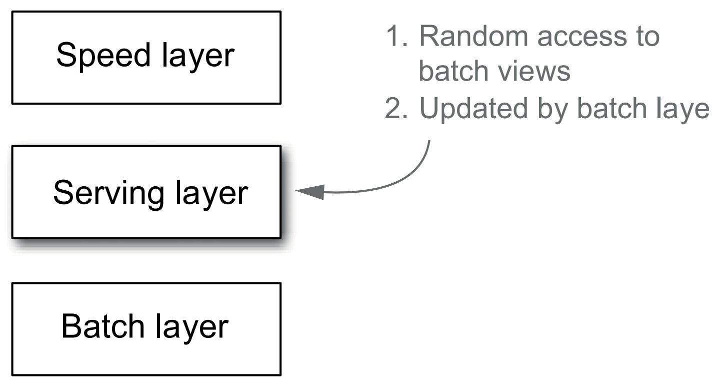
  - Not only does this provide enough horsepower to make previously intractable problems tractable, but it also allows us to create systems that are fault tolerant against both hardware failure and human error.

Lambda architecture: layers view

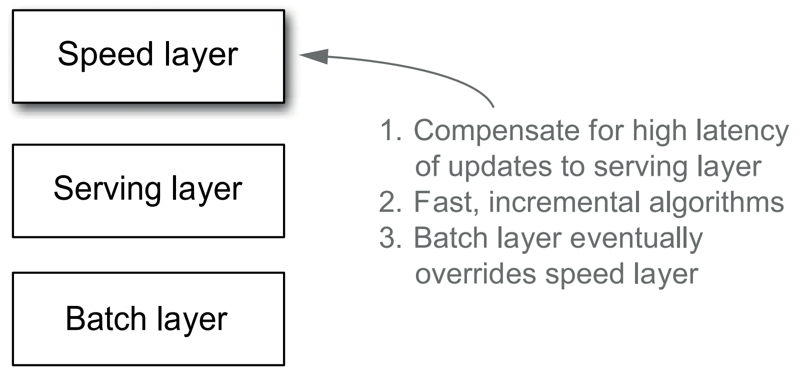
- Each layer satisfies a subset of the properties and builds upon the functionality provided by the layers beneath it.



- The batch layer needs to be able to do two things: store an immutable, constantly growing master dataset, and compute arbitrary functions on that dataset.



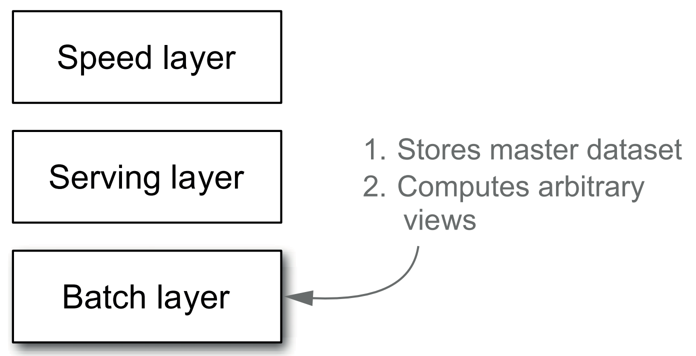
- The serving layer is a specialized distributed database that loads in a batch view and makes it possible to do random reads on it. A serving layer database supports batch updates and random reads. Most notably, it doesn't need to support random writes. (e.g. ElephantDB)


Speed layer
Serving layer
Batch layer
1. Compensate for high latency of updates to serving layer
2. Fast, incremental algorithms
3. Batch layer eventually overrides speed layer

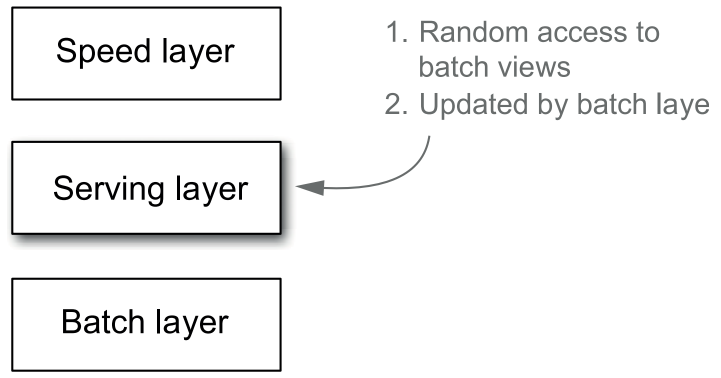
- The serving layer updates whenever the batch layer finishes precomputing a batch view. The speed layer only looks at recent data, whereas the batch layer looks at all the data at once.
  It updates the realtime views as it receives new data instead of recomputing the views from scratch like the batch layer does.

The Lambda Architecture in full is summarized by these three equations:
batch view = function(all data)
realtime view = function(realtime view, new data)
query = function(batch view, realtime view)

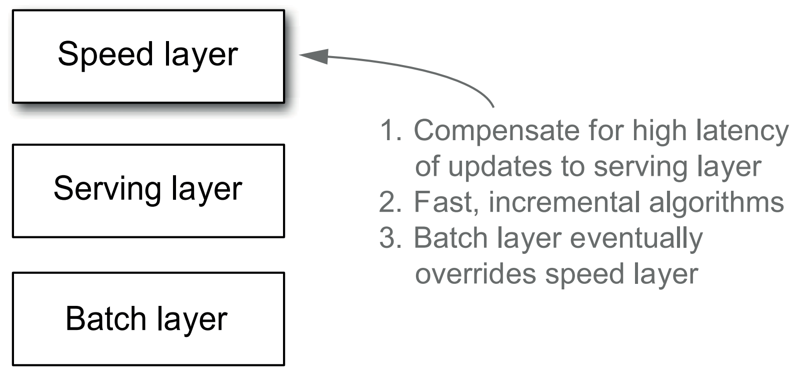
- Each layer satisfies a subset of the properties and builds upon the functionality provided by the layers beneath it.



- The batch layer needs to be able to do two things: store an immutable, constantly growing master dataset, and compute arbitrary functions on that dataset.

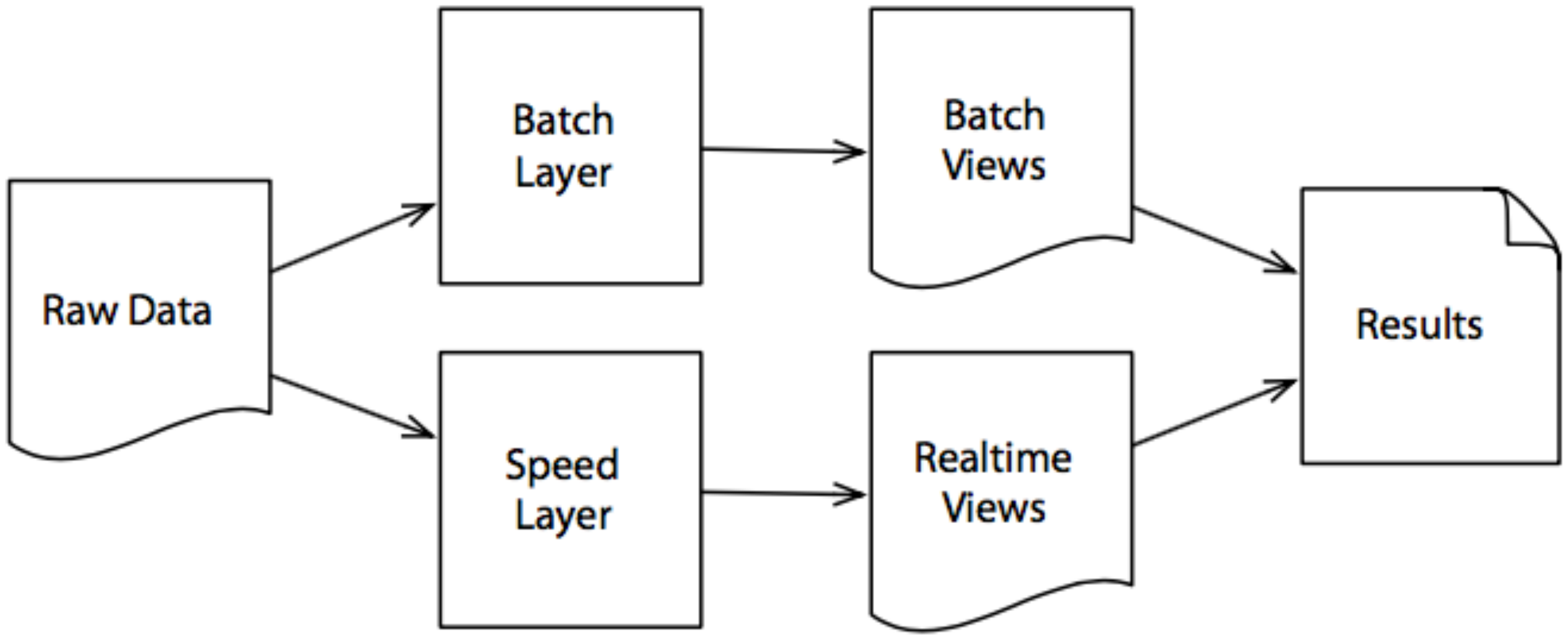


- The serving layer is a specialized distributed database that loads in a batch view and makes it possible to do random reads on it. A serving layer database supports batch updates and random reads. Most notably, it doesn't need to support random writes. (e.g. ElephantDB)


Speed layer
Serving layer
Batch layer
1. Compensate for high latency of updates to serving layer
2. Fast, incremental algorithms
3. Batch layer eventually overrides speed layer

- The serving layer updates whenever the batch layer finishes precomputing a batch view. The speed layer only looks at recent data, whereas the batch layer looks at all the data at once.
  It updates the realtime views as it receives new data instead of recomputing the views from scratch like the batch layer does.

Lambda architecture



- The primary building blocks are the batch layer and the speed layer.
- The batch layer uses batch-oriented technologies like MapReduce to precompute batch views from historical data. This is effective, but latency is high.
- The speed layer uses low-latency techniques like stream processing to create real-time views from new data as it arrives.
- The two types of views are then combined to create query results.

Batch layer

Raw data

- We can divide information into two categories:
  - **raw data**,
  - **derived information**.
- Consider a page on Wikipedia — pages are constantly being updated and improved, so if I view a particular page today, I may well see something different from what I saw yesterday.
  - But pages aren't the raw data from which Wikipedia is constructed — a single page is the result of combining many edits by many different contributors. These **edits** are the **raw data** from which **pages** are **derived**.
  - Pages change day by day but edits do not. Edits are **immutable**.

Raw data and immutability

- The fundamental basis of the Lambda Architecture is that raw data is immutable.
  - In some cases we'll need to add a timestamp to make some data immutable, e.g. a timestamp associated to the address of a person to record immutably that at time X somebody lived at address A and at time Y the same person lived at address B.

UNIVERSITÀ DEGLI STUDI FIRENZE

Immutable data

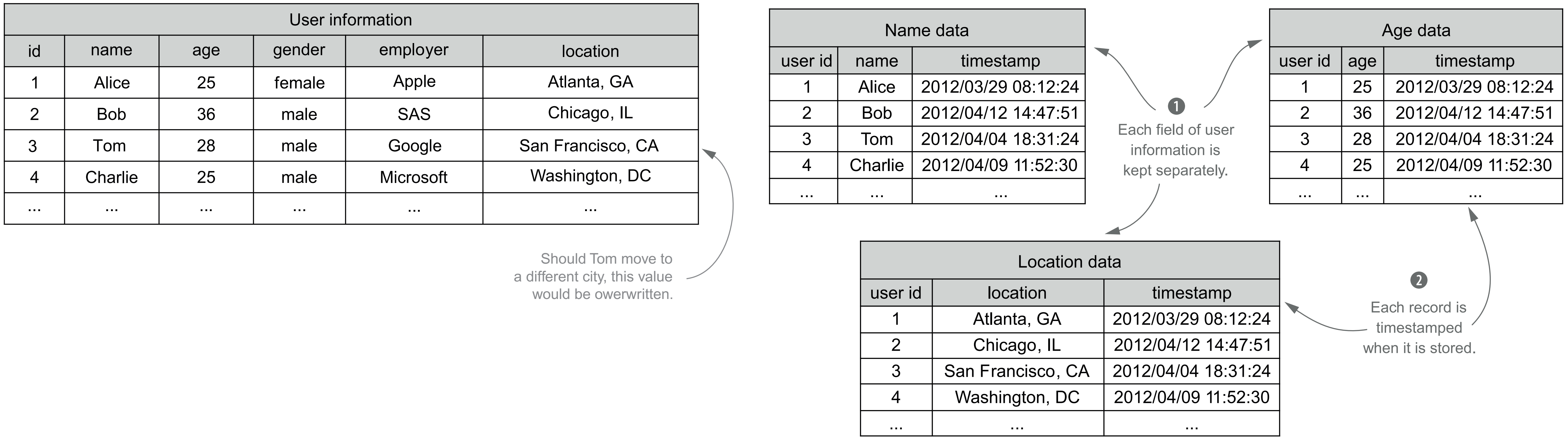
- Storing immutable data is easy: just append new data when it becomes available.

- Multiple threads can access immutable data in parallel without any concern of interfering with each other. We can take copies of it and operate on those copies, **without worrying** about them becoming out- of-date.

  - Distribution of immutable data across a cluster immediately becomes much easier.

- We can compute batch views from raw data to simply select the views that we need. This is exactly the task of the batch layer of Lambda Architecture.

Mutable vs. Immutable database: example



User information					
id	name	age	gender	employer	location
1	Alice	25	female	Apple	Atlanta, GA
2	Bob	36	male	SAS	Chicago, IL
3	Tom	28	male	Google	San Francisco, CA
4	Charlie	25	male	Microsoft	Washington, DC
...	...	...	...	...	...

Should Tom move to a different city, this value would be owerwritten.

Name data		
user id	name	timestamp
1	Alice	2012/03/29 08:12:24
2	Bob	2012/04/12 14:47:51
3	Tom	2012/04/04 18:31:24
4	Charlie	2012/04/09 11:52:30
...	...	...

Age data		
user id	age	timestamp
1	25	2012/03/29 08:12:24
2	36	2012/04/12 14:47:51
3	28	2012/04/04 18:31:24
4	25	2012/04/09 11:52:30
...	...	...

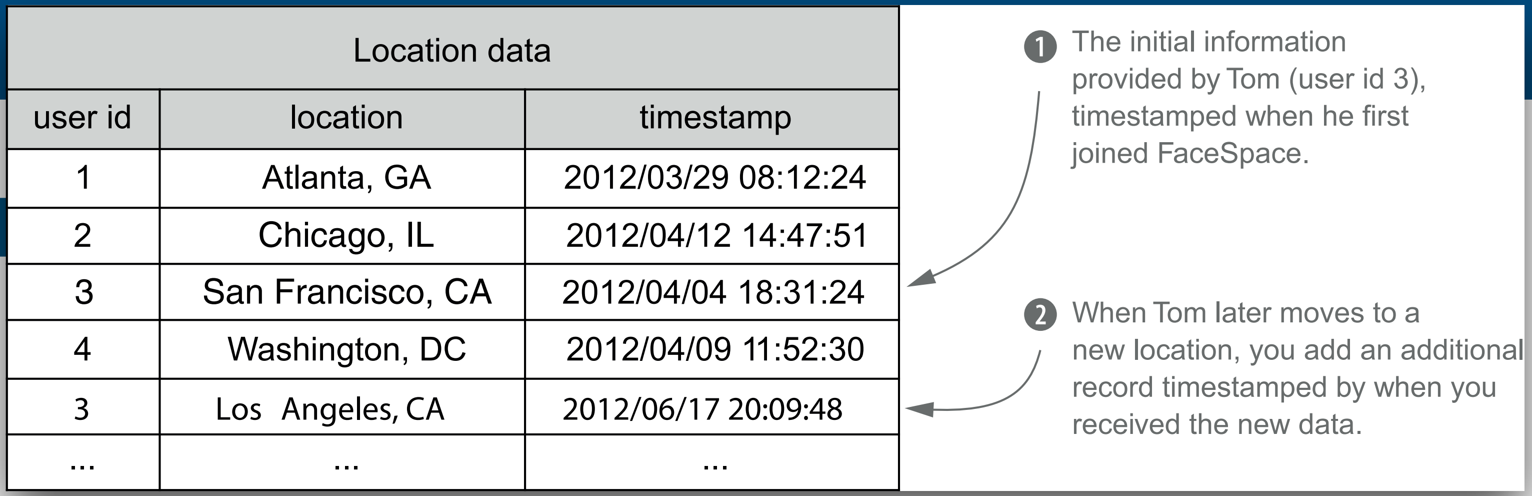
1 Each field of user information is kept separately.

Location data		
user id	location	timestamp
1	Atlanta, GA	2012/03/29 08:12:24
2	Chicago, IL	2012/04/12 14:47:51
3	San Francisco, CA	2012/04/04 18:31:24
4	Washington, DC	2012/04/09 11:52:30
...	...	...

2 Each record is timestamped when it is stored.

- Mutable schema:
  When details change—e.g., Tom moves to Los Angeles—previous values are overwritten and lost.
- Immutable schema
  Each field is tracked in a separate table, and each row has a timestamp for when it's known to be true.

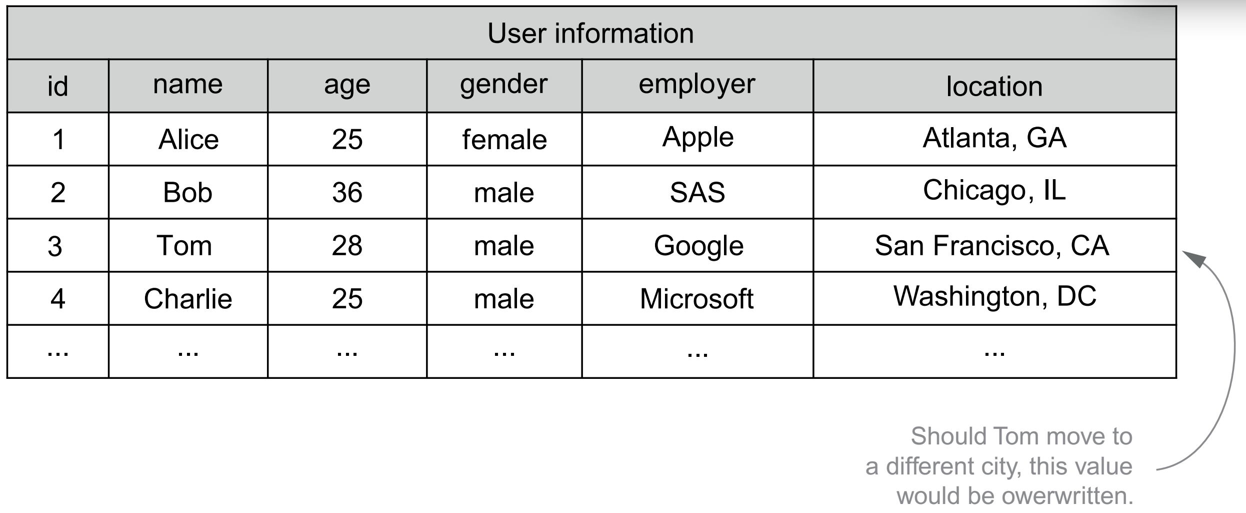
Mutable vs. Immu



Location data		
user id	location	timestamp
1	Atlanta, GA	2012/03/29 08:12:24
2	Chicago, IL	2012/04/12 14:47:51
3	San Francisco, CA	2012/04/04 18:31:24
4	Washington, DC	2012/04/09 11:52:30
3	Los Angeles, CA	2012/06/17 20:09:48
…	…	…

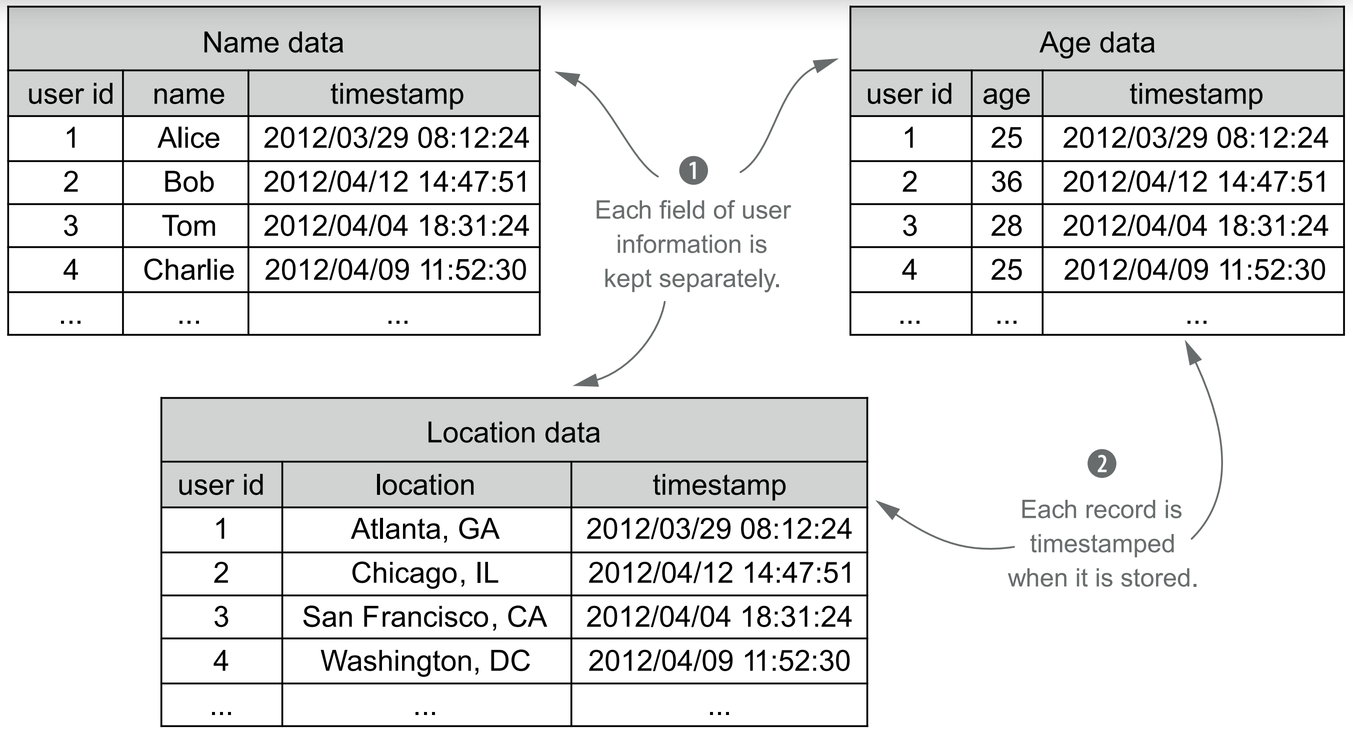
1 The initial information provided by Tom (user id 3), timestamped when he first joined FaceSpace.

2 When Tom later moves to a new location, you add an additional record timestamped by when you received the new data.



User information					
id	name	age	gender	employer	location
1	Alice	25	female	Apple	Atlanta, GA
2	Bob	36	male	SAS	Chicago, IL
3	Tom	28	male	Google	San Francisco, CA
4	Charlie	25	male	Microsoft	Washington, DC
...	...	...	...	...	...

Should Tom move to a different city, this value would be owerwritten.



Name data		
user id	name	timestamp
1	Alice	2012/03/29 08:12:24
2	Bob	2012/04/12 14:47:51
3	Tom	2012/04/04 18:31:24
4	Charlie	2012/04/09 11:52:30
...	...	...

Age data		
user id	age	timestamp
1	25	2012/03/29 08:12:24
2	36	2012/04/12 14:47:51
3	28	2012/04/04 18:31:24
4	25	2012/04/09 11:52:30
...	...	...

Location data		
user id	location	timestamp
1	Atlanta, GA	2012/03/29 08:12:24
2	Chicago, IL	2012/04/12 14:47:51
3	San Francisco, CA	2012/04/04 18:31:24
4	Washington, DC	2012/04/09 11:52:30
...	...	...

1 Each field of user information is kept separately.

2 Each record is timestamped when it is stored.

- Mutable schema:
  When details change—e.g., Tom moves to Los Angeles—previous values are overwritten and lost.
- Immutable schema
  Each field is tracked in a separate table, and each row has a timestamp for when it's known to be true.

Example of batch views

- Get a snapshot of contributors to Wikipedia and compute who is the most active.
  - We get the raw data (e.g. Wikipedia metadata dumps) and compute a batch view with the batch layer.
  - As said before Wikipedia edits are immutable: perfect for some batch processing…

Batch view

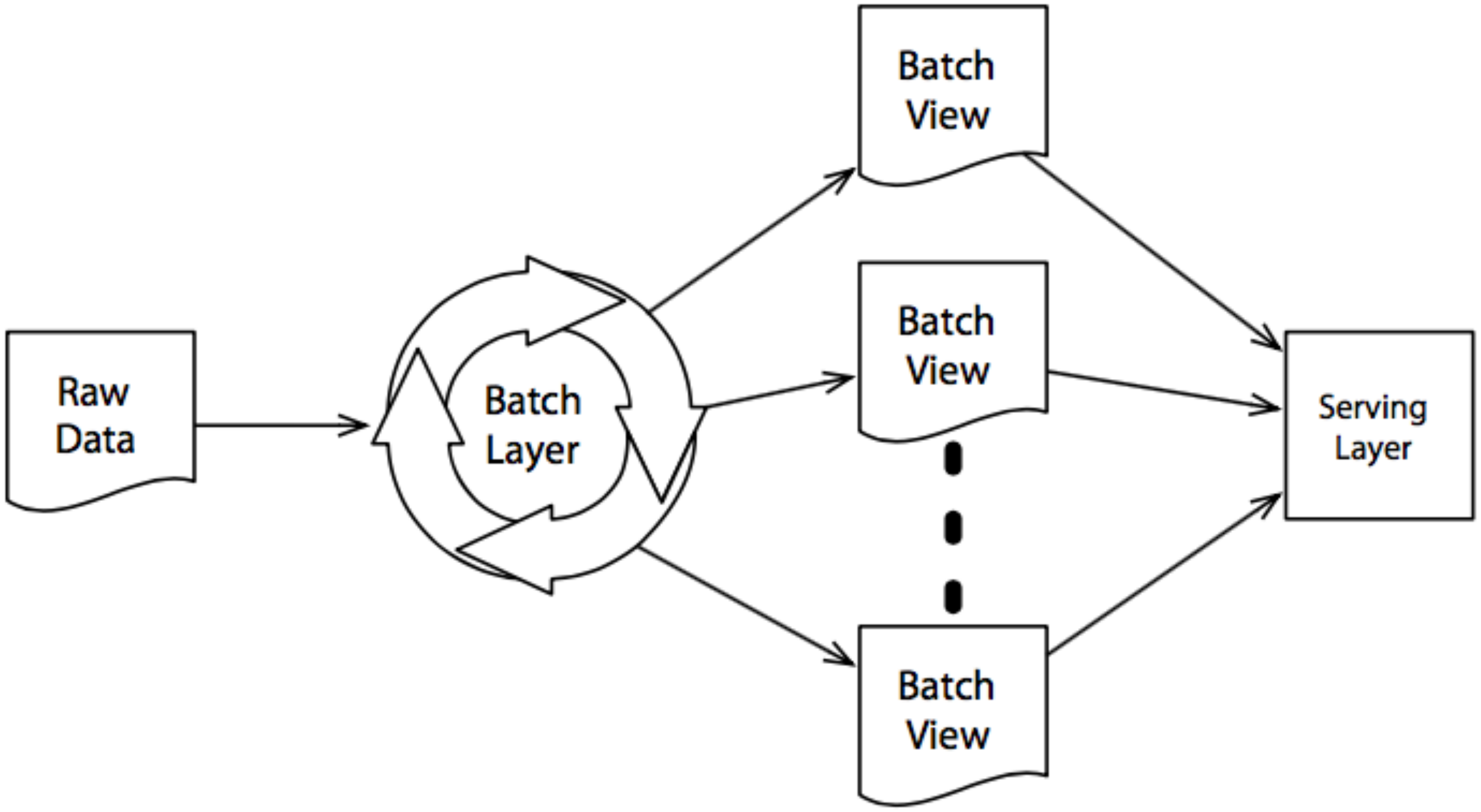
- Ideally we recompile a batch view from scratch but in some cases we may need to implement an incremental approach.

- Batch views can be stored in a database, e.g. to serve them through a serving layer of the Lambda Architecture.

    - But the DB can be simplified since there is no need of random writes: the batch view is updated when the batch layer is executed.

    - There are specific databases like ElephantDB, that is a database that specializes in exporting key/value data from Hadoop, that creates an indexed key/value dataset that is stored on a distributed filesystem.

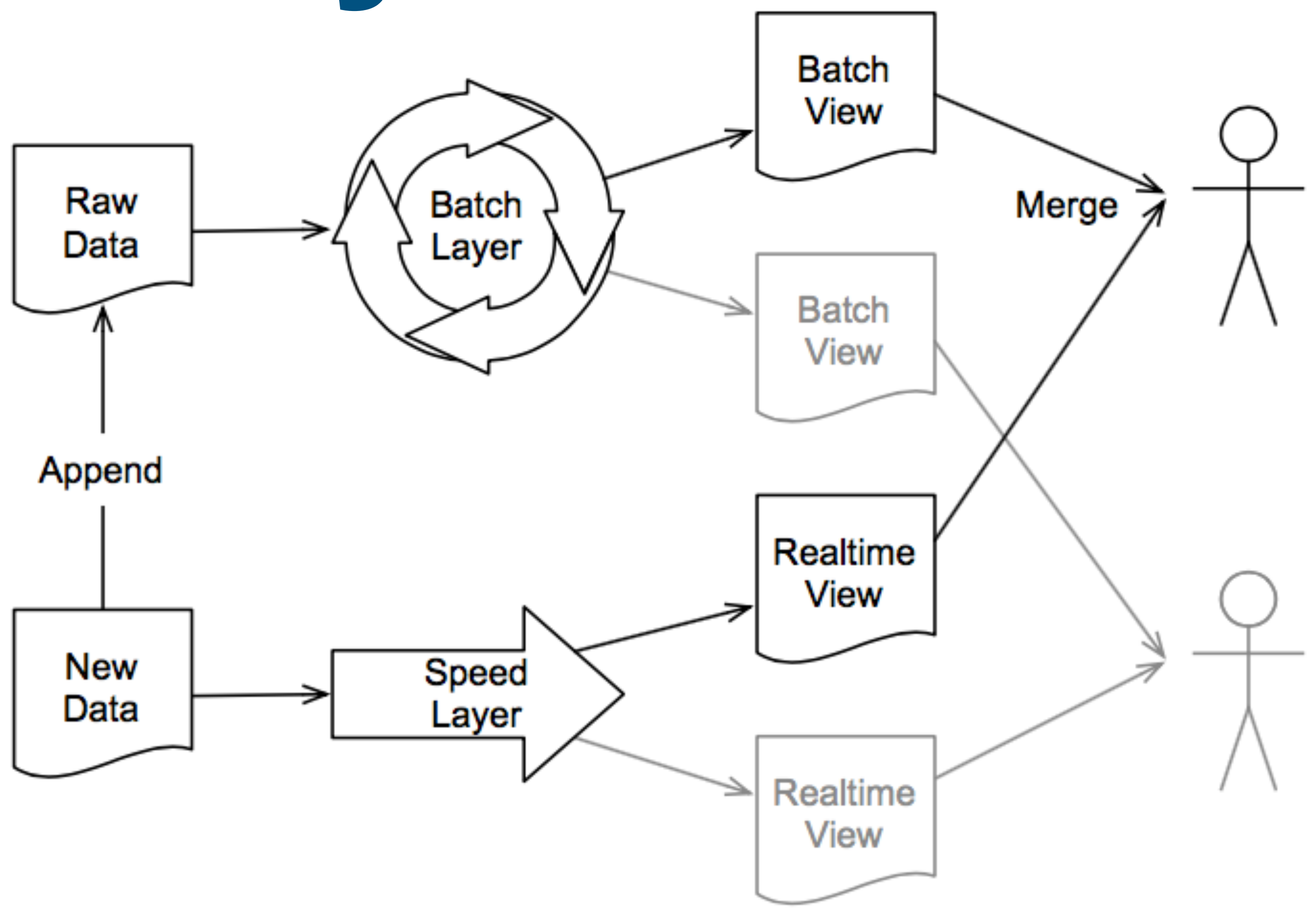
Batch view process



- The batch layer runs in an infinite loop, regenerating batch views from our raw data. Each time a batch run completes, the serving layer updates its database.
- Because it only ever operates on immutable raw data, the batch layer can easily exploit parallelism.
- The main problem is **latency**: if the batch layer takes an hour to run, then our batch views will always be at least an hour out-of-date.

Speed layer

Speed layer: motivations



- The scope of the speed layer is to reduce the problem of latency. We can't use batch-oriented approach for this.

- As new data arrives, we both append it to the raw data that the batch layer works on and send it to the speed layer. The speed layer generates real-time views, which are combined with batch views to create fully up-to-date answers to queries.

- Real-time views contain only information derived from the data that arrived since the batch views were last generated and are discarded when the data they were built from is processed by the batch layer.

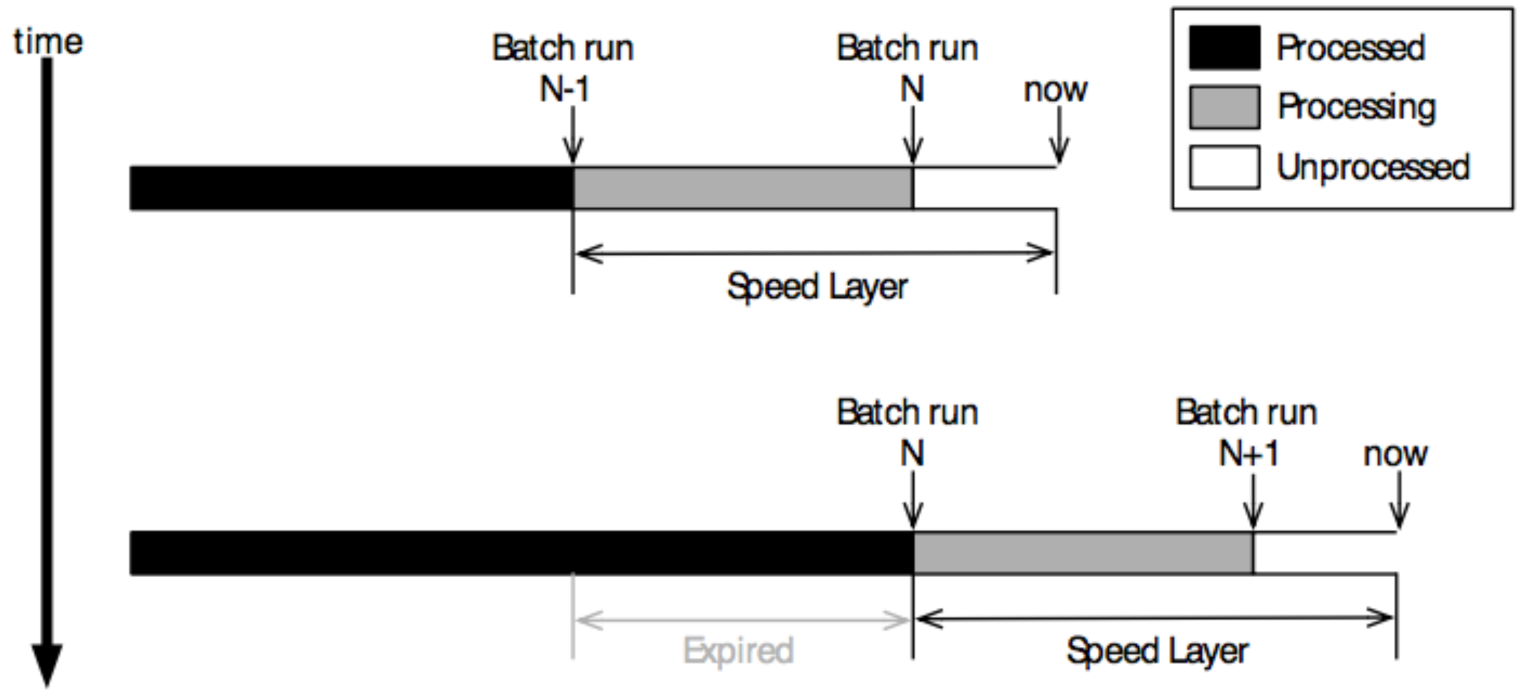
Difficulties

- We can not rely on immutability of data.
- Need to follow an incremental approach at processing data.
- Typically need to handle data from traditional databases (with random writes, locks, transactions, etc.)
  - All these issues must be handled to manage the most recent data. Once the batch layer catches up all this data can be expired from speed layer.

Expiring data

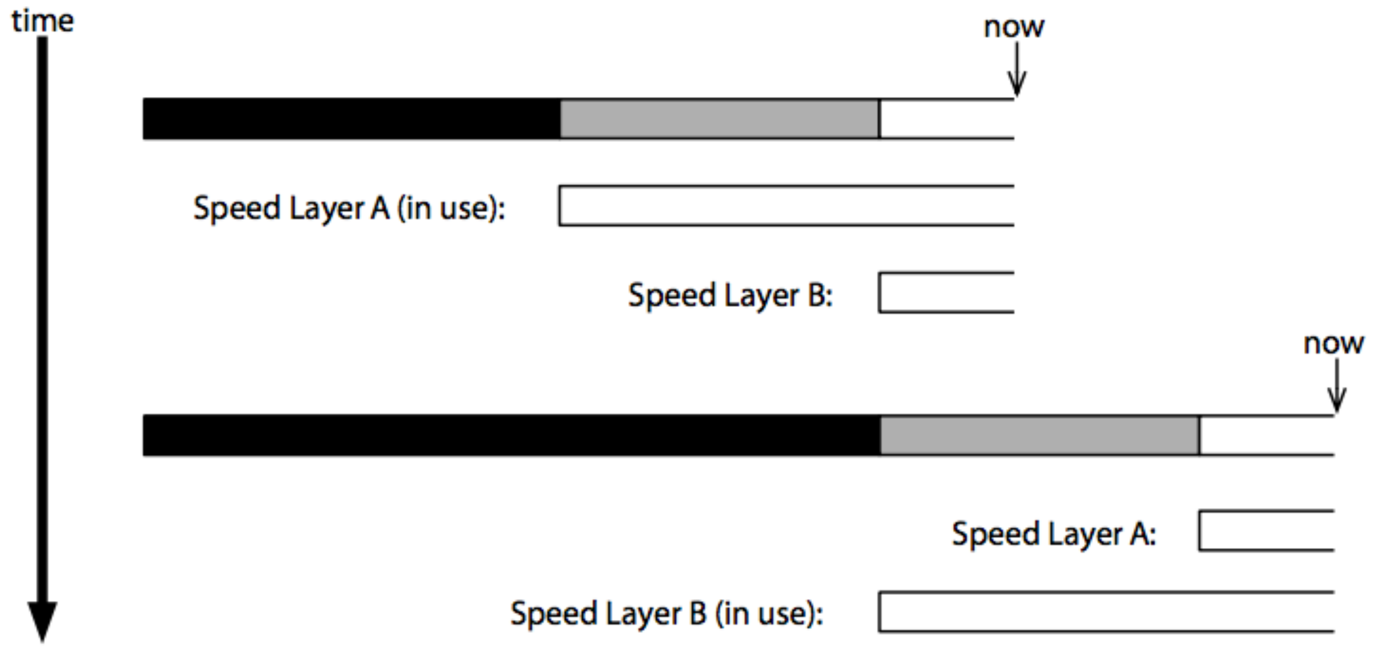
- Imagine that batch run *N-1* has just completed and batch run *N* is just about to start.
  If each takes two hours to run, that means that our batch views will be two hours out-of-date.

- The speed layer therefore needs to serve requests for those two hours' worth of data **plus** any data that arrives before batch run *N* completes, for a total of four hours' worth.



Ping pong schema

- When batch run N does complete, we then need to expire the data that represents the oldest two hours but still retain the most recent two hours' worth.
- A simple solution can be to run two copies of the speed layer in parallel and ping-pong between them:
  - Whenever a batch run completes and new data becomes available in the batch views, we switch from the speed layer that's currently serving queries to its counterpart with more recent data.
  - The now-idle speed layer then clears its database and starts building a new set of views from scratch, starting at the point where the new batch run started.



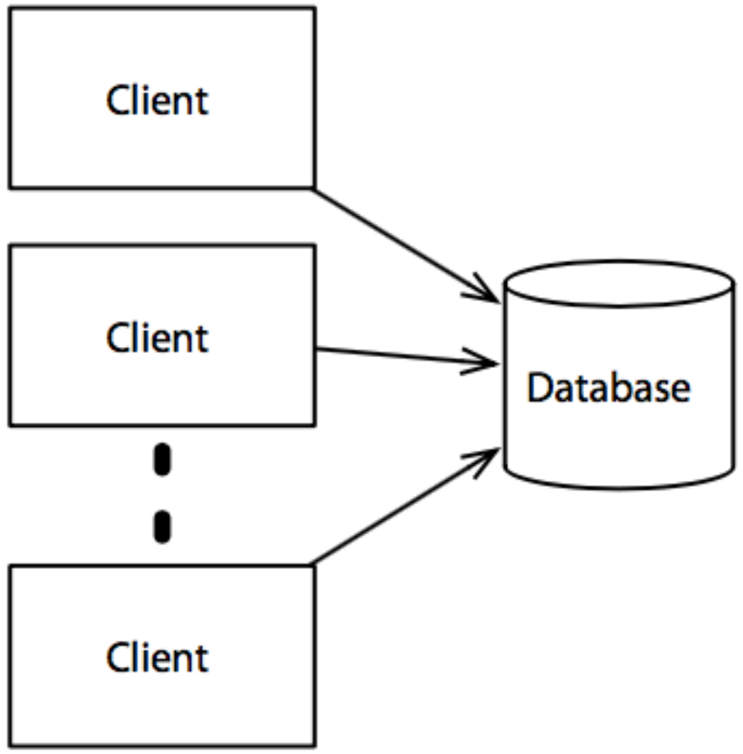
Ping pong schema

- Pros:
  - No need to identify which data to delete from the speed layer’s database;
  - Performance and reliability: each iteration of the speed layer starts from a clean database
- Cons:
  - Need to maintain two copies of speed layer’s data and double occupation of computational resources.

Synchronous approach

- In this approach, clients communicate directly with the database and block while it's processing each update.

- … but blocking leads to loss of performance if new data is added vey fast



Asynchronous approach

- In this approach clients add updates to a queue (e.g. implemented with Apache Kafka) as they arrive and without blocking. A stream processor then handles these updates in turn and performs the database update.

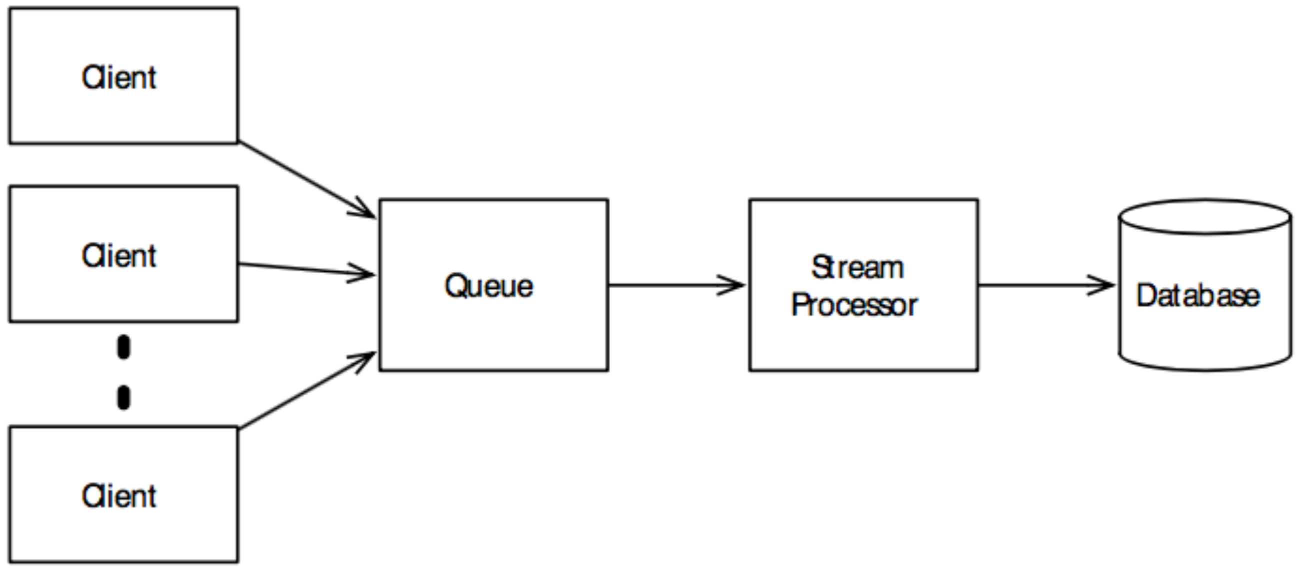
- Using a queue decouples clients from database updates:

  - it is **more complex** to coordinate updates with other actions;

  - no blocking by clients, leading to **greater throughput**;

  - **no timeouts or dropped updates**: what can not be processed fast enough (e.g. during a spike) is just added to the queue;

  - natural exploitation of parallelism.



Apache Storm

Apache Storm

- Apache Storm is a free and open source distributed realtime computation system.

- Storm aims to do for real-time processing what Hadoop has for batch processing — to make it easy to distribute computation across multiple machines in order to improve both performance and fault tolerance.

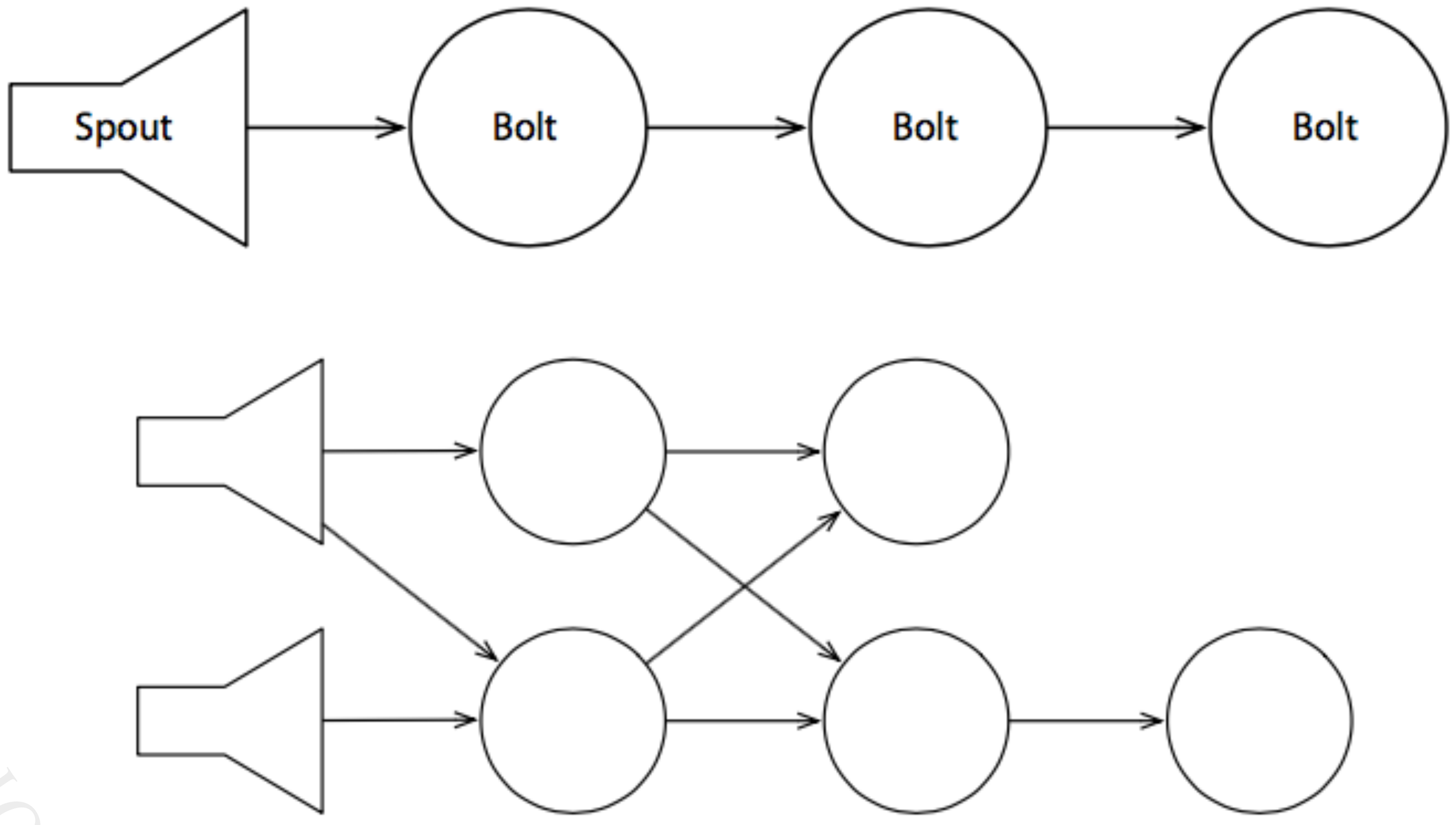
- Storm has two modes of operation: **local mode** and **remote mode**. In local mode, you can develop and test topologies completely in process on your local machine. In remote mode, you submit topologies for execution on a cluster of machines.

Spout, bolt and topology

- A Storm system processes streams of named tuples. A stream is an unbounded sequence of tuples.
  - By default, tuples can contain integers, longs, shorts, bytes, strings, doubles, floats, booleans, and byte arrays. You can also define your own serializers so that custom types can be used natively within tuples.
- Tuples are created by **spouts** and processed by **bolts**, which can create tuples in turn. Spouts and bolts are connected by streams to form a topology.
  - Note: a spout is a tube, pipe, or hole out of which a liquid flows
- The logic for a realtime application is packaged into a Storm topology. A Storm topology is analogous to a MapReduce job.

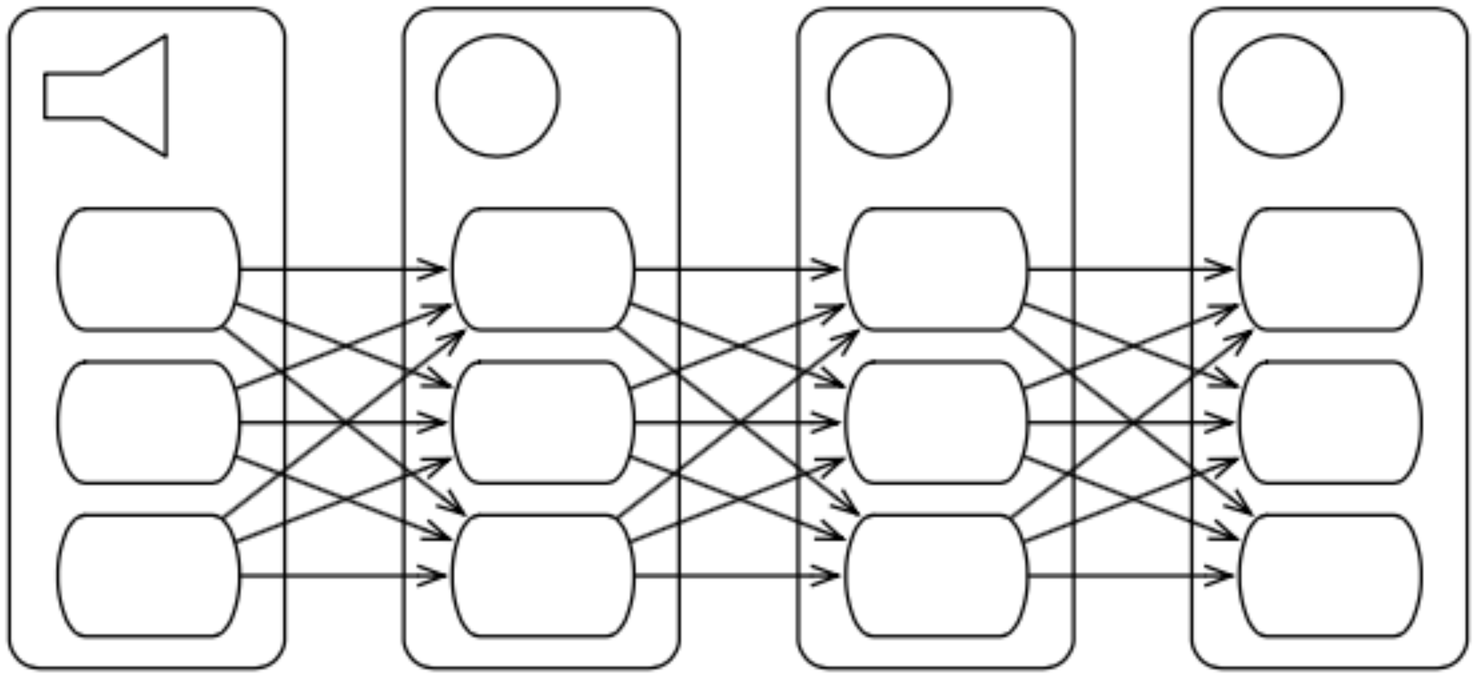
Topology

- Can be even DAGs: bolts can consume multiple streams, and a single stream can be consumed by multiple bolts.



Workers and parallelism

- Not only do spouts and bolts run in parallel with each other, but they are also internally parallel — each is implemented as a set of workers.
- The workers of each node of the pipeline can send tuples to any of the workers in their downstream node.
  - Workers are distributed — if we're running on a four-node cluster, for example, then our spout's workers might be on nodes 1, 2, and 3; the first bolt's workers might be on nodes 2 and 4 (e.g. two on node 2, one on node 4); and so on.
  - We just need to specify our topology, and the Storm runtime allocates workers to nodes and makes sure that tuples are routed appropriately.



Workers and fault tolerance

- A large part of the reason for distributing a single spout or bolt’s workers across multiple machines is fault tolerance.
  If one of the machines in our cluster fails, our topology can continue to operate by routing tuples to the machines that are still operating.

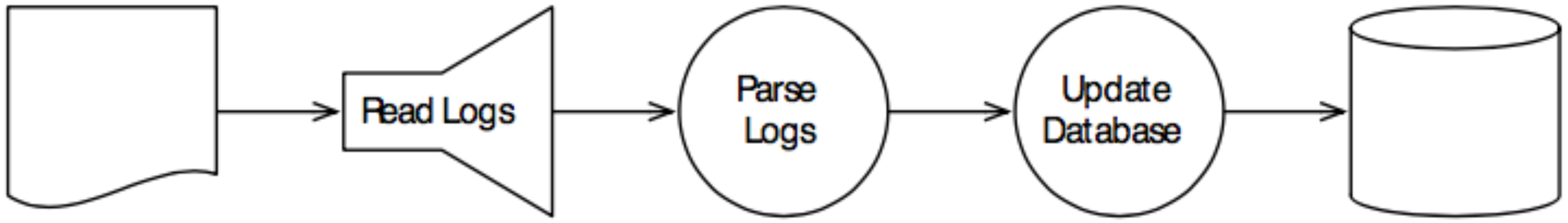
- Storm keeps track of the dependencies between tuples.
  If a particular tuple’s processing isn’t completed, Storm fails and retries the spout tuple(s) upon which it depends.

- This means that, by default, Storm provides an “**at least once**” processing guarantee. Applications need to be aware of the fact that tuples might be retried and continue to function correctly if they are.

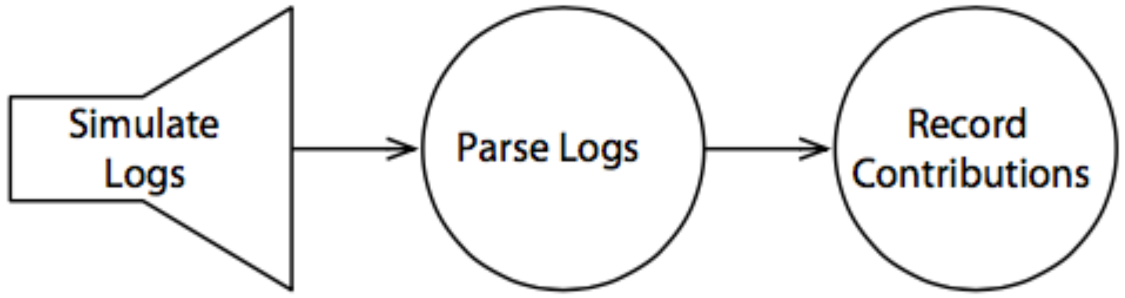
- The Trident API of Storm provides also an “**exactly once**” semantics for processing.

Example: Storm topology

- Let us suppose we want to integrate a speed layer that counts the daily contributions to Wikipedia. We need a topology similar to:



- Note: actually a real topology is like the following since we do not have access to Wikipedia real time data:



Create a Spout

```
public class RandomContributorSpout extends BaseRichSpout {
  private static final Random rand = new Random();
  private static final DateTimeFormatter isoFormat = ISODateTimeFormat.dateTimeNoMillis();

  private SpoutOutputCollector collector;
  private int contributionId = 10000;

  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
    this.collector = collector;
  }

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

  public void nextTuple() {
    Utils.sleep(rand.nextInt(100));
    ++contributionId;
    String line = isoFormat.print(DateTime.now()) + " " + contributionId + " " +
      rand.nextInt(10000) + " " + "dummyusername";
    collector.emit(new Values(line));
  }

}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Create a Spout

create a spout by deriving from `BaseRichSpout`

```
public class RandomContributorSpout extends BaseRichSpout {
  private static final Random rand = new Random();
  private static final DateTimeFormatter isoFormat = ISODateTimeFormat.dateTimeNoMillis();

  private SpoutOutputCollector collector;
  private int contributionId = 10000;

  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
    this.collector = collector;
  }

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

  public void nextTuple() {
    Utils.sleep(rand.nextInt(100));
    ++contributionId;
    String line = isoFormat.print(DateTime.now()) + " " + contributionId + " " +
      rand.nextInt(10000) + " " + "dummyusername";
    collector.emit(new Values(line));
  }

}
```

Create a Spout

```
public class RandomContributorSpout extends BaseRichSpout {
  private static final Random rand = new Random();
  private static final DateTimeFormatter isoFormat = ISODateTimeFormat.dateTimeNoMillis();

  private SpoutOutputCollector collector;
  private int contributionId = 10000;

  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
    this.collector = collector;
  }

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

  public void nextTuple() {
    Utils.sleep(rand.nextInt(100));
    ++contributionId;
    String line = isoFormat.print(DateTime.now()) + " " + contributionId + " " +
      rand.nextInt(10000) + " " + "dummyusername";
    collector.emit(new Values(line));
  }

}
```

Create a Spout

```
public class RandomContributorSpout extends BaseRichSpout {
  private static final Random rand = new Random();
  private static final Da

  private SpoutOutputColl
  private int contributio

  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
    this.collector = collector;
  }

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

  public void nextTuple() {
    Utils.sleep(rand.nextInt(100));
    ++contributionId;
    String line = isoFormat.print(DateTime.now()) + " " + contributionId + " " +
      rand.nextInt(10000) + " " + "dummyusername";
    collector.emit(new Values(line));
  }

}
```

Storm calls **`open()`** method during initialization.
we simply keep a record of the `SpoutOutputCollector`,
which is where we'll send our output.

Create a Spout

```java
public class RandomContributorSpout extends BaseRichSpout {
  private static final Random rand = new Random();
  private static final DateTimeFormatter isoFormat = ISODateTimeFormat.dateTimeNoMillis();

  private SpoutOutputCollector collector;
  private int contributionId = 10000;

  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
    this.collector = collector;
  }

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

  public void nextTuple() {
    Utils.sleep(rand.nextInt(100));
    ++contributionId;
    String line = isoFormat.print(DateTime.now()) + " " + contributionId + " " +
      rand.nextInt(10000) + " " + "dummyusername";
    collector.emit(new Values(line));
  }

}
```

Create a Spout

```
public class RandomContributorSpout extends BaseRichSpout {
  private static final Random rand = new Random();
  private static final DateTimeFormatter isoFormat = ISODateTimeFormat.dateTimeNoMillis();

  private SpoutOutputCollector collector;
  private int contributionId = 10000;

                                                                          collector) {
```

Storm also calls **`declareOutputFields()`** method during initialization to find out how the tuples generated by this spout are structured. in this case, the tuples have a single field called line.

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

  public void nextTuple() {
    Utils.sleep(rand.nextInt(100));
    ++contributionId;
    String line = isoFormat.print(DateTime.now()) + " " + contributionId + " " +
      rand.nextInt(10000) + " " + "dummyusername";
    collector.emit(new Values(line));
  }

}
```

Create a Spout

```
public class RandomContributorSpout extends BaseRichSpout {
  private static final Random rand = new Random();
  private static final DateTimeFormatter isoFormat = ISODateTimeFormat.dateTimeNoMillis();

  private SpoutOutputCollector collector;
  private int contributionId = 10000;

  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
    this.collector = collector;
  }

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

  public void nextTuple() {
    Utils.sleep(rand.nextInt(100));
    ++contributionId;
    String line = isoFormat.print(DateTime.now()) + " " + contributionId + " " +
      rand.nextInt(10000) + " " + "dummyusername";
    collector.emit(new Values(line));
  }

}
```

Create a Spout

```
public class RandomContributorSpout extends BaseRichSpout {
  private static final Random rand = new Random();
  private static final DateTimeFormatter isoFormat = ISODateTimeFormat.dateTimeNoMillis();

  private SpoutOutputCollector collector;
  private int contributionId = 10000;

  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
    this.collector = collector;
  }

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

  public void nextTuple() {
    Utils.sleep(rand.nextInt(1
    ++contributionId;
    String line = isoFormat.pr
      rand.nextInt(10000) + "
    collector.emit(new Values(line));
  }

}
```

The method that does most of the work is `nextTuple()`.
It uses the collector to emit the created tuples.

Create a Bolt

```
class ContributionParser extends BaseBasicBolt {

  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("timestamp",
                                "id",
                                "contributorId",
                                "username"
                          )
                     );
  }

  public void execute(Tuple tuple, BasicOutputCollector collector) {
    Contribution contribution = new Contribution(tuple.getString(0));
    collector.emit( new Values(contribution.timestamp,
                               contribution.id,
                               contribution.contributorId,
                               contribution.username
                         )
                  );

  }

}
```

Create a Bolt

we're creating a bolt by deriving from BaseBasicBolt

```
class ContributionParser extends BaseBasicBolt {

  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("timestamp",
                                "id",
                                "contributorId",
                                "username"
                          )
                    );
  }

  public void execute(Tuple tuple, BasicOutputCollector collector) {
    Contribution contribution = new Contribution(tuple.getString(0));
    collector.emit( new Values(contribution.timestamp,
                               contribution.id,
                               contribution.contributorId,
                               contribution.username
                         )
                 );

  }

}
```

Create a Bolt

```
class ContributionParser extends BaseBasicBolt {

  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("timestamp",
                                "id",
                                "contributorId",
                                "username"
                          )
                     );
  }

  public void execute(Tuple tuple, BasicOutputCollector collector) {
    Contribution contribution = new Contribution(tuple.getString(0));
    collector.emit( new Values(contribution.timestamp,
                               contribution.id,
                               contribution.contributorId,
                               contribution.username
                         )
                  );

  }

}
```

Create a Bolt

we implement `declareOutputFields()` to let Storm know how our output tuples are structured — in this case they have four fields.

```
public void declareOutputFields(OutputFieldsDeclarer declarer) {
  declarer.declare(new Fields("timestamp",
                              "id",
                              "contributorId",
                              "username"
                        )
                  );
}

public void execute(Tuple tuple, BasicOutputCollector collector) {
  Contribution contribution = new Contribution(tuple.getString(0));
  collector.emit( new Values(contribution.timestamp,
                             contribution.id,
                             contribution.contributorId,
                             contribution.username
                       )
               );

}

}
```

Create a Bolt

```
class ContributionParser extends BaseBasicBolt {

  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("timestamp",
                                "id",
                                "contributorId",
                                "username"
                          )
                     );
  }

  public void execute(Tuple tuple, BasicOutputCollector collector) {
    Contribution contribution = new Contribution(tuple.getString(0));
    collector.emit( new Values(contribution.timestamp,
                               contribution.id,
                               contribution.contributorId,
                               contribution.username
                         )
                  );

  }

}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Create a Bolt

```
class ContributionParser extends BaseBasicBolt {

  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("timestamp",
                                "id",
                                "contributorId",
                                "username"
                          )
                   );
  }

  public void execute(Tuple tuple, BasicOutputCollector collector) {
    Contribution contribution = new Contribution(tuple.getString(0));
    collector.emit( new Values(contribution.timestamp,
                               contribution.id,
                               contribution.contributorId,
                               contribution.username
                         )
                );

  }
```

The method that does most of the work is `execute()`. In this case, it uses `Contributor` (should be the same of the batch layer) to parse the log line into its components and then calls `contributor.emit()` to output the tuple.

Create another Bolt

```
class ContributionRecord extends BaseBasicBolt {

  private static final HashMap<Integer, HashSet<Long>> timestamps =
                 new HashMap<Integer, HashSet<Long>>();



  public void declareOutputFields(OutputFieldsDeclarer declarer) {
  }

  public void execute(Tuple tuple, BasicOutputCollector collector) {
    addTimestamp(tuple.getInteger(2), tuple.getLong(0));
  }

  private void addTimestamp(int contributorId, long timestamp) {
    HashSet<Long> contributorTimestamps = timestamps.get(contributorId);
    if (contributorTimestamps == null) {
      contributorTimestamps = new HashSet<Long>();
      timestamps.put(contributorId, contributorTimestamps);
    }
    contributorTimestamps.add(timestamp);
  }

}
```

Create another Bolt

This final Bolt simply maintains an in-memory database in the hash table.
It could also connect to a better DB to stop the results.

```
class

  private static final HashMap<Integer, HashSet<Long>> timestamps =
                   new HashMap<Integer, HashSet<Long>>();


  public void declareOutputFields(OutputFieldsDeclarer declarer) {
  }

  public void execute(Tuple tuple, BasicOutputCollector collector) {
    addTimestamp(tuple.getInteger(2), tuple.getLong(0));
  }

  private void addTimestamp(int contributorId, long timestamp) {
    HashSet<Long> contributorTimestamps = timestamps.get(contributorId);
    if (contributorTimestamps == null) {
      contributorTimestamps = new HashSet<Long>();
      timestamps.put(contributorId, contributorTimestamps);
    }
    contributorTimestamps.add(timestamp);
  }

}
```

Create another Bolt

```
class ContributionRecord extends BaseBasicBolt {

  private static final HashMap<Integer, HashSet<Long>> timestamps =
                   new HashMap<Integer, HashSet<Long>>();



  public void declareOutputFields(OutputFieldsDeclarer declarer) {
  }

  public void execute(Tuple tuple, BasicOutputCollector collector) {
    addTimestamp(tuple.getInteger(2), tuple.getLong(0));
  }

  private void addTimestamp(int contributorId, long timestamp) {
    HashSet<Long> contributorTimestamps = timestamps.get(contributorId);
    if (contributorTimestamps == null) {
      contributorTimestamps = new HashSet<Long>();
      timestamps.put(contributorId, contributorTimestamps);
    }
    contributorTimestamps.add(timestamp);
  }

}
```

Create another Bolt

```
class ContributionRecord extends BaseBasicBolt {

  private static final HashMap<Integer, HashSet<Long>> timestamps =
                   new HashMap<Integer, HashSet<Long>>();
```

In this case we're not generating any output, so `declareOutputFields()` is empty.

```
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
  }

  public void execute(Tuple tuple, BasicOutputCollector collector) {
    addTimestamp(tuple.getInteger(2), tuple.getLong(0));
  }

  private void addTimestamp(int contributorId, long timestamp) {
    HashSet<Long> contributorTimestamps = timestamps.get(contributorId);
    if (contributorTimestamps == null) {
      contributorTimestamps = new HashSet<Long>();
      timestamps.put(contributorId, contributorTimestamps);
    }
    contributorTimestamps.add(timestamp);
  }

}
```

Create another Bolt

```
class ContributionRecord extends BaseBasicBolt {

  private static final HashMap<Integer, HashSet<Long>> timestamps =
                   new HashMap<Integer, HashSet<Long>>();



  public void declareOutputFields(OutputFieldsDeclarer declarer) {
  }

  public void execute(Tuple tuple, BasicOutputCollector collector) {
    addTimestamp(tuple.getInteger(2), tuple.getLong(0));
  }

  private void addTimestamp(int contributorId, long timestamp) {
    HashSet<Long> contributorTimestamps = timestamps.get(contributorId);
    if (contributorTimestamps == null) {
      contributorTimestamps = new HashSet<Long>();
      timestamps.put(contributorId, contributorTimestamps);
    }
    contributorTimestamps.add(timestamp);
  }

}
```

Create another Bolt

```
class ContributionRecord extends BaseBasicBolt {

  private static final HashMap<Integer, HashSet<Long>> timestamps =
                   new HashMap<Integer, HashSet<Long>>();
```

`execute()` method simply extracts the relevant fields from its input tuple and passes them to `addTimestamp()`, which simply stores them in the in-memory DB.

```
  public void execute(Tuple tuple, BasicOutputCollector collector) {
    addTimestamp(tuple.getInteger(2), tuple.getLong(0));
  }

  private void addTimestamp(int contributorId, long timestamp) {
    HashSet<Long> contributorTimestamps = timestamps.get(contributorId);
    if (contributorTimestamps == null) {
      contributorTimestamps = new HashSet<Long>();
      timestamps.put(contributorId, contributorTimestamps);
    }
    contributorTimestamps.add(timestamp);
  }

}
```

Create the Topology

```
public class WikiContributorsTopology {

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

    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("contribution_spout", new RandomContributorSpout(), 4);

    builder.setBolt("contribution_parser", new ContributionParser(), 4).
            shuffleGrouping(“contribution_spout");

    builder.setBolt("contribution_recorder", new ContributionRecord(), 4).
            fieldsGrouping("contribution_parser", new Fields("contributorId"));

    LocalCluster cluster = new LocalCluster();
    Config conf = new Config();
    cluster.submitTopology("wiki-contributors", conf, builder.createTopology());

    Thread.sleep(10000);

    cluster.shutdown();

  }

}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Create the Topology

```
public class WikiContributorsTopology {

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

    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("contribution_spout", new RandomContributorSpout(), 4);

    builder.setBolt("contribution_parser", new ContributionParser(), 4).
            shuffleGrouping("contribution_spout");

    builder.setBolt("contribution_recorder", new ContributionRecord(), 4).
            fieldsGrouping("contribution_parser", new Fields("contributorId"));

    LocalCluster cluster = new LocalCluster();
    Config conf = new Config();
    cluster.submitTopology("wiki-contributors", conf, builder.createTopology());

    Thread.sleep(10000);

    cluster.shutdown();

  }

}
```

Create a `TopologyBuilder`

Create the Topology

```
public class WikiContributorsTopology {

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

    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("contribution_spout", new RandomContributorSpout(), 4);

    builder.setBolt("contribution_parser", new ContributionParser(), 4).
            shuffleGrouping("contribution_spout");

    builder.setBolt("contribution_recorder", new ContributionRecord(), 4).
            fieldsGrouping("contribution_parser", new Fields("contributorId"));

    LocalCluster cluster = new LocalCluster();
    Config conf = new Config();
    cluster.submitTopology("wiki-contributors", conf, builder.createTopology());

    Thread.sleep(10000);

    cluster.shutdown();

  }

}
```

Create the Topology

```
public class WikiContributorsTopology {

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

    Topolog
```

Create and name the Spout. Suggest to use 4 workers.

```
    builder.setSpout("contribution_spout", new RandomContributorSpout(), 4);

    builder.setBolt("contribution_parser", new ContributionParser(), 4).
            shuffleGrouping(“contribution_spout");

    builder.setBolt("contribution_recorder", new ContributionRecord(), 4).
            fieldsGrouping("contribution_parser", new Fields("contributorId"));

    LocalCluster cluster = new LocalCluster();
    Config conf = new Config();
    cluster.submitTopology("wiki-contributors", conf, builder.createTopology());

    Thread.sleep(10000);

    cluster.shutdown();

  }

}
```

Create the Topology

```
public class WikiContributorsTopology {

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

    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("contribution_spout", new RandomContributorSpout(), 4);

    builder.setBolt("contribution_parser", new ContributionParser(), 4).
            shuffleGrouping(“contribution_spout");

    builder.setBolt("contribution_recorder", new ContributionRecord(), 4).
            fieldsGrouping("contribution_parser", new Fields("contributorId"));

    LocalCluster cluster = new LocalCluster();
    Config conf = new Config();
    cluster.submitTopology("wiki-contributors", conf, builder.createTopology());

    Thread.sleep(10000);

    cluster.shutdown();

  }

}
```

Create the Topology

```
public class WikiContributorsTopology {

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

    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("contribution_spout", new RandomContributorSpout(), 4);

    builder.setBolt("contribution_parser", new ContributionParser(), 4).
            shuffleGrouping(“contribution_spout");
```

Create and name a Bolt. Tell to get tuples from the named Spout.
Will receive a randomly chosen tuple from Spout.

```
                                                                       ).
                                                                      rId"));

    LocalCluster cluster = new LocalCluster();
    Config conf = new Config();
    cluster.submitTopology("wiki-contributors", conf, builder.createTopology());

    Thread.sleep(10000);

    cluster.shutdown();

  }

}
```

Create the Topology

```
public class WikiContributorsTopology {

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

    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("contribution_spout", new RandomContributorSpout(), 4);

    builder.setBolt("contribution_parser", new ContributionParser(), 4).
            shuffleGrouping(“contribution_spout");

    builder.setBolt("contribution_recorder", new ContributionRecord(), 4).
            fieldsGrouping("contribution_parser", new Fields("contributorId"));

    LocalCluster cluster = new LocalCluster();
    Config conf = new Config();
    cluster.submitTopology("wiki-contributors", conf, builder.createTopology());

    Thread.sleep(10000);

    cluster.shutdown();

  }

}
```

Create the Topology

```
public class WikiContributorsTopology {

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

    TopologyBuilder builder = new TopologyBuilder();
```

Create and name a Bolt. Tell to get tuples from the named Bolt.
Require to get tuples based on a specific value for a set of fields (one field in this case).
This guarantees that the same worker will get the timestamps of the same contributor.

```
    builder.setBolt("contribution_recorder", new ContributionRecord(), 4).
            fieldsGrouping("contribution_parser", new Fields("contributorId"));

    LocalCluster cluster = new LocalCluster();
    Config conf = new Config();
    cluster.submitTopology("wiki-contributors", conf, builder.createTopology());

    Thread.sleep(10000);

    cluster.shutdown();

  }

}
```

Create the Topology

```
public class WikiContributorsTopology {

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

    TopologyBuilder builder = new TopologyBuilder();
```

Create and name a Bolt. Tell to get tuples from the named Bolt.
Require to get tuples based on a specific value for a set of fields (one field in this case).
This guarantees that the same worker will get the timestamps of the same contributor.

```
    builder.setBolt("contribution_recorder", new ContributionRecord(), 4).
            fieldsGrouping("contribution_parser", new Fields("contributorId"));

    LocalCluster cluster = new LocalCluster();
    Config conf = new Config();
    cluster.submitTopology("wiki-contributors", conf, builder.createTopology());

    Thread.sleep(10000);

    cluster.shutdown();

  }
```

Create a Storm cluster, submit the topology and let it run for 10 seconds, then shut it down.

```
}
```

Books

- Seven Concurrency Models in Seven Weeks, Paul Butcher, Pragmatic Bookshelf - Chapt. 8
- Big Data, Nathan Marz, Manning - Chapt. 1, 2, 6