# Distributed Data Analytics

## Types of Systems

<table>
<thead>
<tr>
<th>Services (online systems)</th>
<th>OLTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Accept requests and send responses</td>
<td></td>
</tr>
<tr>
<td>- Performance measure: response time and availability</td>
<td></td>
</tr>
<tr>
<td>- Expected runtime: milliseconds to seconds</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Batch processing systems (offline systems)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Take (large amounts of) data; run (complex) jobs; produce some output</td>
<td></td>
</tr>
<tr>
<td>- Performance measure: throughput (i.e., data per time)</td>
<td></td>
</tr>
<tr>
<td>- Expected runtime: minutes to days</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stream processing systems (near-real-time systems)</th>
<th>OLAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Consume volatile inputs; operate stream jobs; produce some output</td>
<td></td>
</tr>
<tr>
<td>- Performance measure: throughput and precision</td>
<td></td>
</tr>
<tr>
<td>- Expected runtime: near-real-time (i.e., as data arrives)</td>
<td></td>
</tr>
</tbody>
</table>
Distributed Data Analytics

Types of Systems

Batch processing systems (offline systems)

Stream processing systems (near-real-time systems)
Distributed Data Analytics

Types of Systems

Batch processing systems (offline systems)

Stream processing systems (near-real-time systems)
Distributed Data Analytics
Use Cases for Streaming Data

Sensor Processing
- Continuous and endless readings by nature

Process Monitoring
- Side effects of processes that are continuously observed

Location Tracking
- Continuous location updates of certain devices

Log Analysis
- Digital footprints of applications that grow continuously

User Interaction
- Continuous and oftentimes bursty click- and call-events

Market and Climate Prediction
- Changing stock market prices and weather characteristics

...
Spark Streaming (Recap)

Batched Stream Processing
- Reasons:
  - Incremental processing: start processing data that is still being written to
  - Latency reduction: pipeline data to maximizing resource utilization

Stream processing reads the data exactly once and still guarantees fault-tolerance through check pointing and write ahead logs (WAL)
Overview
Stream Processing

Transmitting Event Streams

Databases and Streams

Processing Streams

Distributed Data Analytics
Stream Processing
Thorsten Papenbrock
Slide 7
Transmitting Event Streams

Streams

Data Stream
- Any data that is incrementally made available over time
- Examples:
  - Unix stdin and stdout
  - Filesystem APIs (e.g. Java’s FileInputStream)
  - Online media delivery (audio/video streaming)
- Creation from ...
  - static data: files or databases (read records line-wise)
  - dynamic data: sensor readings, service calls, transmitted data, logs, ...

Event
- = an immutable record in a stream (often with timestamp)
- “Something that happened”
- Encoded in Json, XML, CSV, ... maybe in binary format

Any format that allows incremental appends
Transmitting Event Streams

Batch vs. Stream

**Batches**
- Write once, read often

**Streams**
- Send once, receive once
- maybe multiple receivers

Distributed Data Analytics
Stream Processing

Thorsten Papenbrock
Slide 9
Transmitting Event Streams
Event Transmission

Dataflow Through Databases
- Process 1
- Process 2
- Process 2 needs to poll the database for updates
  - bad performance
  - slow event propagation

Dataflow Through Services
- Process 1
- Process 2
- Working speed of process 2 determines stream speed
  - maybe bad performance
  - ok-ish event propagation

Message-Passing Dataflow
- Process 1
- Process 2
- Asynchronous messaging and notification about new events
  - good performance
  - fast event propagation
Communication
- Objects send messages to other objects via queues

Message
- Container for data (= events)
  - Often carries metadata (sender, receiver, timestamp, ...)

Message queue
- Data structure (queue or list) assigned to communicating object(s)
  - Enqueues messages in order of arrival
  - Buffers incoming messages for being processed
  - Notifies subscribers if new messages are available
What if the stream producer is faster than the stream consumer(s)?

a) **Drop messages**
   - Delete messages that cannot be accepted
   - Ok for use cases where timeliness is more important than completeness (e.g., for processing of sensor readings)

b) **Buffer messages**
   - Store messages in a cache until resources are available
   - Ok to capture load spikes and if there is no constant overload that fills up buffers permanently (e.g., for user activity event streams)

c) **Apply backpressure**
   - Block the sender until resources are available
   - Ok if the sender can be blocked and if the stream is not generated from outside (e.g., for reading a file as a stream from disk)
What if nodes crash or temporarily go offline?

a) Fault ignorance
   - Failed messages are lost
     - Ensures optimal throughput and latency

b) Fault tolerance
   - Failed messages are recovered from checkpoints (disk or replicas)
     - Ensures messaging reliability

More on fault tolerance later!
Message Broker

- Also called message queue or message-oriented middleware
- Part of the message-passing framework that delivers messages from their sender to the receiver(s)
- Maintains queues that sender can post messages to
- Notifies subscribers on new messages
- Resolves sender and receiver addresses
- Applies binary encoding when necessary
- Define the ...
  - message congestion strategy
  - messaging fault strategy

If it blocks and persists, then it is a database, right?
Transmitting Event Streams
Message Brokers vs. Databases

**Message Broker**
- Short lived messages
  - Delete messages once successfully transmitted
- Small working set
  - If the number of pending messages increases, the performance drops (disk!)
- Subscription-based retrieval
  - Deliver messages to all subscribers of a queue
- Push client communication
  - Knows clients and initiates communications

**Database**
- Long-term persisted records
  - Store records until explicitly deleted
- Large working set
  - If the number of records increases, the performance is hardly affected
- Query-based retrieval
  - Read records upon client query using indexes
- Pull client communication
  - Clients are unknown and initiate communications
Routing

- Producer send messages to queues
- Message Broker notifies **one or many** consumers about such deliveries
- Routing strategies:
  
  a) **One-to-one** messages (Load Balancing)
     - Messages are routed to one subscriber
     - For data parallelism

  b) **One-to-many** messages (Fan-out)
     - Messages are routed to all subscribers
     - For task parallelism
Fault tolerance

- **Acknowledgement:**
  - Consumer send an acknowledgement to the Message Broker when they completed a message
  - Message Broker removes any completed message from its queues

- **Redelivery:**
  - If acknowledgement fails to appear, the Message Broker redelivers it (perhaps to a different consumer)
Fault tolerance

m3 failed at consumer 2 and is **redelivered** to consumer 3

m3 is preserved but stream at consumer 3 is now **out-of-order**!
**Persist**
- Keep all queue content (until reaching size or time limit)
- No need to track consumers
- Let consumers go back in time
  - Database-like
- Log-based Message Broker
  (e.g. Kafka, Kinesis or DistributedLog)

**Forget**
- Remove processed queue content (immediately after acknowledgement)
- Track consumers to forget old content
- The past is past
  - Volatile, light-weight
- JMS\(^1\) or AMQP\(^2\) Message Brokers
  (e.g. RabbitMQ, ActiveMQ or HornetQ)

\(^1\) Java Message Service (JMS) 2.0 Specification
\(^2\) Advanced Message Queuing Protocol (AMQP) Specification
Transmitting Event Streams
Message Brokers: Persist or Forget

https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka

Transmitting Event Streams
Message Brokers: Persist or Forget

Persist

- Keep all queue content (until reaching size or time limit)
- No need to track consumers
- Let consumers go back in time
  - Database-like
- Log-based Message Broker
  (e.g. Kafka, Kinesis or DistributedLog)

Forget

- Remove processed queue content (immediately after acknowledgement)
- Track consumers to forget old content
- The past is past
  - Volatile, light-weight
- JMS\textsuperscript{1} or AMQP\textsuperscript{2} Message Brokers
  (e.g. RabbitMQ, ActiveMQ or HornetQ)

\textsuperscript{1}Java Message Service (JMS) 2.0 Specification
\textsuperscript{2}Advanced Message Queuing Protocol (AMQP) Specification
Log-based Message Broker

- Message Broker that persist queues as logs on disk (distributed, replicated)
- Recall ...
  - LSM-Trees with B-Trees and SSTables (Chapter 3)

- Leader-based replication (Chapter 5)
Transmitting Event Streams
Partitioned Logs

Log-based Massage Broker

Send message by appending to log

Receive message by reading log sequentially; when reaching the end, wait for notification

Only one-to-many messaging!

 sequence offsets to ensure ordering

= Stream B

partitioning (and replication)

Distributed Data Analytics
Stream Processing

ThorstenPapenbrock
Slide 23
Transmitting Event Streams
Partitioned Logs

Log-based Message Broker

Storing a history for events **costs memory**

**Example:**
- 6 TB of disk capacity (= log size)
- 150 MB/s write throughput
- 11 h until an event is forgotten (at maximum event throughput!)

**No one-to-one scheduling:**
Max parallelism bound by number of partitions in a topic!

Events with high processing costs **block** all subsequent events

**Distributed Data Analytics**
Stream Processing

Thorsten Papenbrock
Slide 24
Transmitting Event Streams
Message Brokers: Persist or Forget

Persist

- Keep all queue content (until reaching size or time limit)
- No need to track consumers
- Let consumers go back in time
  - Database-like
- Log-based Message Broker (e.g. Kafka, Kinesis or DistributedLog)

Forget

- Remove processed queue content (immediately after acknowledgement)
- Track consumers to forget old content
- The past is past
  - Volatile, light-weight
- JMS\(^1\) or AMQP\(^2\) Message Brokers (e.g. RabbitMQ, ActiveMQ or HornetQ)

Use if **throughput** matters, event processing costs are similar and the **order of messages** is important

Use if **one-to-one scheduling** is needed, **event processing costs differ** and the **order of messages** is insignificant

\(^{1}\)Java Message Service (JMS) 2.0 Specification
\(^{2}\)Advanced Message Queuing Protocol (AMQP) Specification
Transmitting Event Streams
Message Brokers: Persist or Forget

- **Persist**
  - Keep all queue content (until reaching size or time limit)
  - No need to track consumers
  - Let consumers go back in time
    - Database-like
  - Log-based Message Broker (e.g. Kafka, Kinesis or DistributedLog)

- **Forget**
  - Java Message Service (JMS) 2.0 Specification
  - Advanced Message Queuing Protocol (AMQP) Specification

Use if **throughput** matters, event processing costs are similar and the order of messages is important

Wait **throughput**?
Yes, because ...
- dumping events to storage instead of routing them to consumers is faster
- broker does not need to track acknowledgements for every event (only consumers track their queue offset)
- broker can utilize batching and pipelining internally
Overview
Stream Processing

Transmitting Event Streams

Databases and Streams

Processing Streams

Distributed Data Analytics
Stream Processing
ThorstenPapenbrock
Slide 27
Databases and Streams

Data Storage – Keeping Systems in Sync

Producer

Events

Volatile write/delete instructions

OLAP System

OLTP System

Search Index

Caches

Slide 28

Distributed Data Analytics
Stream Processing

ThorstenPapenbrock
Write conflict:

Database and search index are inconsistent, because they don’t share a common leader (that implements e.g. 2PC or MVCC)
Stream Processing

Distributed Data Analytics
Stream Processing

Enables:
- Global ordering of events (→ eventual consistency)
- Fault-safe event delivery
- Backpressure on high load
Databases and Streams
Data Storage – Keeping Systems in Sync

Producer

Events

Persisting Message Broker

OLAP System

OLTP System

Search Index

Caches

Distributed Data Analytics
Stream Processing
ThorstenPapenbrock
Slide 31
Data Change Event Streams

- If events are change operations (writes/deletes) to individual objects (records) it suffices to store only the most recent log entry for each object to rebuild a database

- Log Compaction:
  - Periodically removes outdated log entries from the log
  - Lets the log grow linearly with the data

Message Broker → Database

- If the broker knows what the events mean (e.g. key-value mappings) it can apply log compaction
  - Event log does not outgrow the maximum buffer size
  - Message broker becomes a database

- Implemented by e.g. Apache Kafka
Message Broker as a Database

- Advantages:
  - Data Provenance/Auditability:
    - The line of events describes the history of every value
    - Allows to follow a value back in time (e.g. the balance history of a bank account)
    - Fraud protection, temporal analytics, data recovery, ...
  - Command Query Responsibility Segregation (CQRS):
    - Events describe what happened (= facts) not their implications
    - Allows consumers to read/interpret events differently (= different views)
    - Multi-tenant systems, system evolution, data analytics, ...

- Disadvantages:
  - Non-standing reads are slow (need to scan and interpret the entire event history)
  - Deleting data means declaring it deleted (actually deleting data is hard)
Overview
Stream Processing

Transmitting Event Streams

Databases and Streams

Processing Streams

Distributed Data Analytics
Stream Processing
Thorsten Papenbrock
Goal

- Query and analyze streaming data in real-time (i.e. as data passes by)

Challenges

- Limited memory resources (but endlessly large volumes of data)
  - Only a fixed-size window of the stream is accessible at a time
- Old data is permanently gone (and not accessible any more)
  - Only one-pass algorithms can be used
- Endlessness contradicts certain operations
  - E.g. sorting makes no sense, i.e., no sort-merge-joins or -groupings
- Input cannot be re-read or back-traced
  - Fault tolerance must be ensured differently
### Windows
- A continuous segment of the stream usually implemented as a buffer
  - New events oust the oldest events from the window
- Events within the window can be accessed arbitrarily often
- Bounded in size usually using a time interval or a maximum number of events

While sliding over the events, successive windows may or may not overlap.

At the heart of processing infinite streams, as they let us make clear statements for concrete intervals.
Standing queries

- Persisted queries that are served with volatile event data (reversed DBMS principle)
- Produce a streaming output of “complex events”
- Apply event checking, pattern matching, correlation analysis, aggregation, ...
- Operate on windows
Complex Event Processing (CEP)

- “Check a stream for patterns; whenever something special happens, raise a flag”
- Similar to pattern matching with regular expressions (often SQL-dialects)
- Implementations: Esper, IBM InfoSphere, Apama, TIBICO StreamBase, SQLstream

Stream Analytics

- “Transform or aggregate a stream; continuously output current results”
- Often use statistical metrics and probabilistic algorithms:
  - Bloom filters (set membership)
  - HyperLogLog (cardinality estimation)
  - HDHistogram, t-digest, decay (percentile approximation)
- Implementations: Storm, Flink, Spark Streaming, Concord, Samza, Kafka Streams, Google Cloud Dataflow, Azure Stream Analytics

Approximation is often used for optimization, but Stream Processing is not inherently approximate!
Maintaining Materialized Views

- "Serve materialized views with up-to-date data from a stream"
- Views are also caches, search indexes, data warehouses, and any derived data system
- Implementations: Samza, Kafka Streams (but also works with Flink, Spark, and co.)

Search on Streams

- "Search for events in the stream; emit any event that matches the query"
- Similar to CEP but the standing queries are indexed, less complex, and more in number
- Implementations: Elasticsearch

Message Passing

- "Use the stream for event communication; actors/processes consume and produce events"
- Requires non-blocking one-to-many communication
- Implementations: Any message broker; RPC systems with one-to-many support (e.g. Storm)
Processing Streams

Windows

**Tumbling Windows**
- Fixed-length, non-overlapping windows
  - New window starts when previous window ended (e.g. successive intervals of 3 seconds/events)

**Hopping Windows**
- Fixed-length, overlapping windows with fix steps
  - Defined by window length and hop width (e.g. intervals of 3 seconds starting every 2 seconds)

**Sliding Windows**
- Fixed-length, overlapping windows with event dependent steps
  - Either new events oust old events or events stay for a certain amount of time

**Session Windows**
- Arbitrary-length, overlapping windows
  - Fix start- and end-event (e.g. user logs in; user logs out or session times out)
Processing Streams

Windows

Non-Keyed Windows

- Partition a stream into another stream of buckets
- For parallel processing, buckets need to be replicated
  - Not supported by all streaming frameworks

Keyed Windows

- Partition a stream into multiple other streams of buckets (one per key value)
- Output streams can naturally be processed in parallel without replication
  - Default stream parallelization technique

Also called partitioned windows
Continuous Query Language

- Developed at Stanford University: [http://www-db.stanford.edu/stream](http://www-db.stanford.edu/stream)
- Used to define standing queries for windows of a stream

```
SELECT count(*)
FROM Requests R [RANGE 1 Day PRECEDING]
WHERE R.domain = 'stanford.edu'
```

"Count the number of requests to stanford.edu for the last 1 day."

```
SELECT count(*)
FROM Requests R [PARTITION BY R.client_id
ROWS 10 PRECEDING
WHERE R.domain = 'stanford.edu']
WHERE R.url LIKE 'http://cs.stanford.edu/%%'
```

"From the last 10 requests of a user to standord.edu, count all her calls to cs."

ThorstenPapenbrock
Storm

- A free and open source distributed real-time computation system (stream processor)
- Competes with Apache Flink in stream processing speed
- Creates a directed acyclic graph (DAG) of “spout” and “bolt” vertices
  - Spout = streaming data source
  - Bolt = data transformation operator

- Designed for:
  - real-time analytics
  - online machine learning
  - continuous computation
  - distributed RPC
  - ETL

- Guarantees:
  - scalability
  - fault-tolerance
  - exactly-once processing of data
  - ease to set up and operate
A source that streams some text lines

```java
public class RandomSentenceSpout extends BaseRichSpout {
  SpoutOutputCollector _collector;
  Random _rand;

  @Override
  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
    _collector = collector;
    _rand = new Random();
  }

  @Override
  public void nextTuple() {
    Util.sleep(100);
    String[] sentences = new String[]{"the cow jumped over the moon", "an apple a day keeps the doctor away", "four score and seven years ago", "snow white and the seven dwarfs", "i am at two with nature");
    String sentence = sentences[_rand.nextInt(sentences.length)];
    _collector.emit(new Values(sentence));
  }

  @Override
  public void ack(Object id) {
  }

  @Override
  public void fail(Object id) {
  }

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

Text to be streamed

Output format
Storm implements UDFs

### Example

**RandomSentenceSpout**
```java
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;
}
```

**SplitSentence**
```java
public static class SplitSentence extends BaseBasicBolt {
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }

    @Override
    public Map<String, Object> getComponentConfiguration() {
        return null;
    }

    public void execute(Tuple tuple, BasicOutputCollector basicOutputCollector) {
        String sentence = tuple.getStringByField("sentence");
        String words[] = sentence.split(" ");
        for (String w : words) {
            basicOutputCollector.emit(new Values(w));
        }
    }

    @Override
    public void fail(Object id) {
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```
Another flatMap() implementation

Streaming output: emit every update
Parallelism hint for souts/bolts

Define the grouping for the input of each bolt:
- shuffle: assign randomly
- field: assign by field value

Execute on cluster

Execute locally

Runs until explicitly stopped

More on Apache Storm @ http://storm.apache.org/

Define the grouping for the input of each bolt:
- shuffle: assign randomly
- field: assign by field value

Execute on cluster

Execute locally

Runs until explicitly stopped

More on Apache Storm @ http://storm.apache.org/
Spark Streaming (Recap)

```scala
val articles = spark.read.text("/mnt/data/articles/*.csv")

val words = articles.as[String].flatMap(_.split(" "))
val urls = words.filter(_.startsWith("http"))
val occurrences = urls.groupBy("value").count()

occurrences.show()
```

Streaming input sources:
- Files: text, csv, json, parquet
- Kafka: Apache Kafka message broker
- Socket: UTF8 text data from a socket
- Rate: Generated data for testing

Streaming output sinks:
- Files: "parquet", "orc", "json", "csv", etc.
- Kafka: "kafka" pointing to a Kafka topic
- Foreach: .foreach(...)
- Console: "console"
- Memory: "memory" with .queryName("...")

"complete" write the entire result for every result update
"append" append new results; old results should not change
"update" output only changed results
val env = StreamExecutionEnvironment.getExecutionEnvironment
val text = env.socketTextStream("localhost", 4242, "n")

val windowCounts = text
  .flatMap{ w => w.split("\s") }
  .map{ w => WordWithCount(w, 1) }
  .keyBy("word")
  .timeWindow(Time.seconds(5), Time.seconds(1))
  .sum("count")

windowCounts.print().setParallelism(1)
env.execute("Socket Window WordCount")

case class WordWithCount(word: String, count: Long)
Processing Streams

Event Time vs. Processing Time

Event Time
- Time when the event is emitted by the producer (when it occurred)

Processing Time
- Time when the event is perceived by the stream processor (when it arrives)

Unpredictable Time Lag
- Events might be delayed due to ...
  - congestion, queuing, faults, ...
- Events might be out-of-order due to ...
  - message loss and resend, alternative routing, ...
- Event time might be measured differently due to ...
  - multiple clocks in distributed systems, clock skew and correction, ...

Stream processors (e.g. Flink) let you choose which time to use for windowing!

Recall lecture on “Distributed Systems”

Distributed Data Analytics
Stream Processing
ThorstenPapenbrock
Slide 50
Event Time
- Time when the event is emitted by the producer (when it occurred)

Processing Time
- Time when the event is perceived by the stream processor (when it arrives)
Solutions?

- Assign timestamps as early as possible:
  - producer > leader > time-synced worker > un-synced worker
- Assign multiple timestamps
  - creation-time, send-time, receive-time, forward-time, ...
- Solve time lag programmatically:
  - Exchange a fixed event frequency (e.g. frequency = 1 second)
  - Reasoning over events (e.g. order(X) > pay(X) > deliver(X))

Many events (e.g. sensor or log) carry timestamps naturally

Used to calculate the lag

filming order ≠ narrative order

Distributed Data Analytics
Stream Processing
Processing Streams
Completing a Window

Problem
- How does a stream worker know that all events for a certain window have arrived? (as events might be delayed → straggler events)

Solution
- Declare a window as completed if ...
  a) the first event for next window arrives or
  b) a timeout for this window has elapsed.
- Handle straggler events after completion of their window by ...
  a) ignoring them (maybe counting/reporting ignored stragglers) or
  b) publishing an update for their window or
  c) assigning them to the next window.
### Issues

- **Unbounded:**
  - Jobs cannot wait making their output visible until their stream finishes

- **Volatile:**
  - If a fault occurs, stream data cannot be re-read
Microbatching and Checkpointing

- **Microbatches** (see Spark):
  - Tumbling windows that are treated as batches (cached, checkpointed, ...)
  - Windows represent state that is written to disk and serves to recover from faults

- **Checkpoints** (see Flink):
  - Rolling checkpoints that are triggered periodically by barriers in the event stream
  - Operator state is written to disk and serves to recover from faults
  - Checkpoints are not tied to particular window sizes

- Both strategies ensure that every event is processed
  - No event is lost until it produced some output

- Still problematic:
  - Actions that recover from faults might produce *redundant outputs* to external event sinks (databases, message brokers, HDFS, ...)

Distributed Data Analytics

Stream Processing

ThorstenPapenbrock

Slide 55
Atomic Commit (revisited)

- Avoid **redundant outputs** using a commit protocol in conjunction with every event sink
- Commits are logged, which helps to check whether an output happened before
- Single event commits are cheaper than transaction commits
- Still a research area with only a few systems supporting it:
  - Google Cloud Dataflow, VoltDB, Kafka (in development)

Idempotence

- Avoid **redundant output effects** using only idempotent output operations
- Idempotent operation = operation that has the same effect regardless how often it is applied
- Examples (multiple calls always replace the existing data with itself):
  - Set key to value; Create file with name; Delete resource; Overwrite content with text
- Many non-idempotent operations can be made idempotent:
  - Add an offset/identifier to each output event that identifies redundancy
Stream-Stream Join
- Task: Join events in stream A with events in stream B
- Problem: Joins require all events of one side to be randomly accessible, but stream is endless
- Solution: Window Joins
  - One side of the join is kept in memory as a window (e.g. session window of logged-in users)
  - The other side of the join is probed against the events of that window (e.g. request events to an API)
  - Straggler events are dropped

Stream-Table Join
- Task: Join events in a stream with events in a database
- Problem: Database is too large for memory and too slow for stream checks
- Solution: Database Partitioning/Replication
  - Forward the stream to different partitions/replica that perform different parts of the join
1. Data Mining
2. Large-Scale File Systems and Map-Reduce
3. Finding Similar Items
4. Mining Data Streams
   - Sampling and Filtering
   - Counting and Aggregation
   - Estimation
   - Decaying Windows
5. Link Analysis
6. Frequent Itemsets
7. Clustering
8. Advertising on the Web
9. Recommendation Systems