Introduction to Apache Flink™

Maximilian Michels
mxm@apache.org
@stadtlegende

Ufuk Celebi
uce@apache.org
@iamuce

EIT ICT Summer School 2015

dataArtisans

About Us

- Ufuk Celebi <uce@apache.org>
- Maximilian Michels <mxm@apache.org>

- Apache Flink Committers

- Working full-time on Apache Flink at dataArtisans

Structure

Goal: Get you started with Apache Flink

- Overview
- Internals
- APIs
- Exercise
- Demo
- Advanced features

Exercises

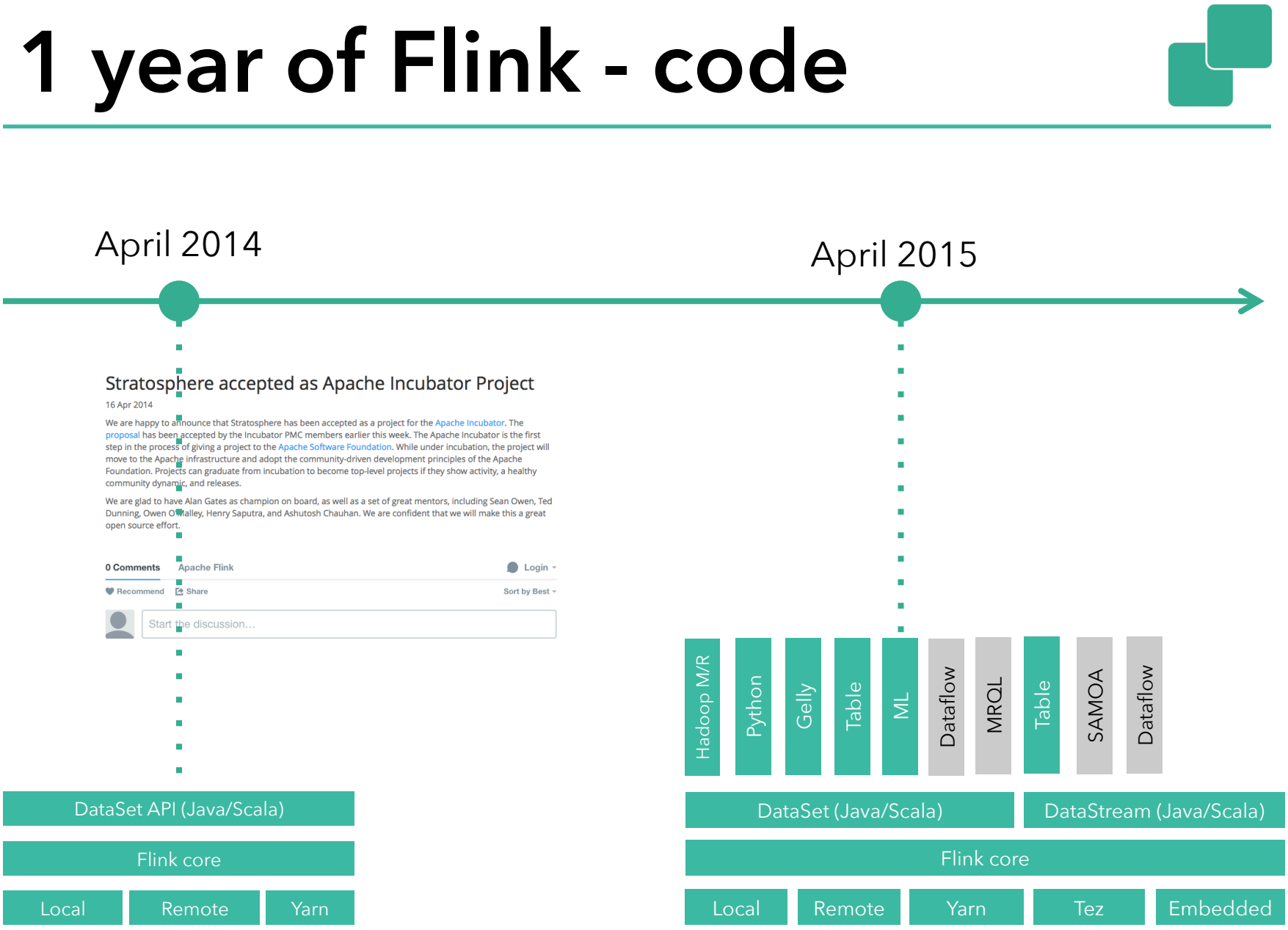
- We will ask you to do an exercise using Flink

- Please check out the website for this session:

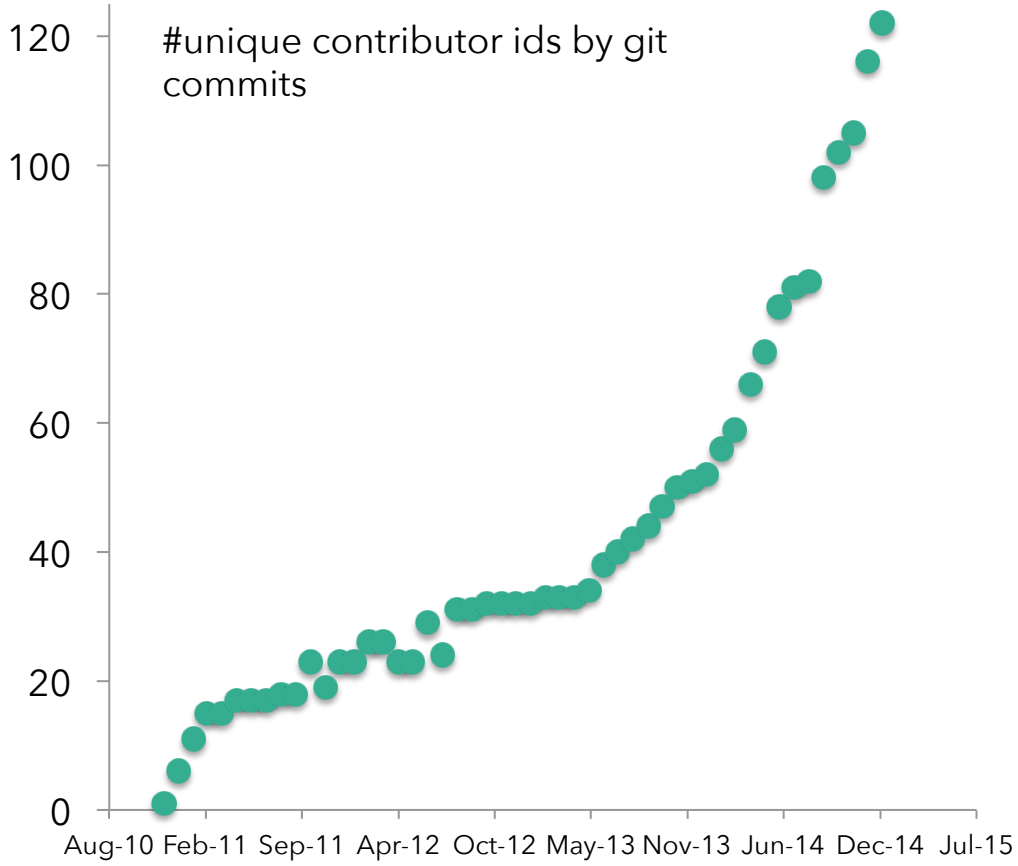
http://dataArtisans.github.io/eit-summerschool-15

- The website also contains a solution which we will present later on but, first, try to solve the exercise.

1 year of Flink - code



Flink Community



In top 5 of Apache's big data projects after one year in the Apache Software Foundation

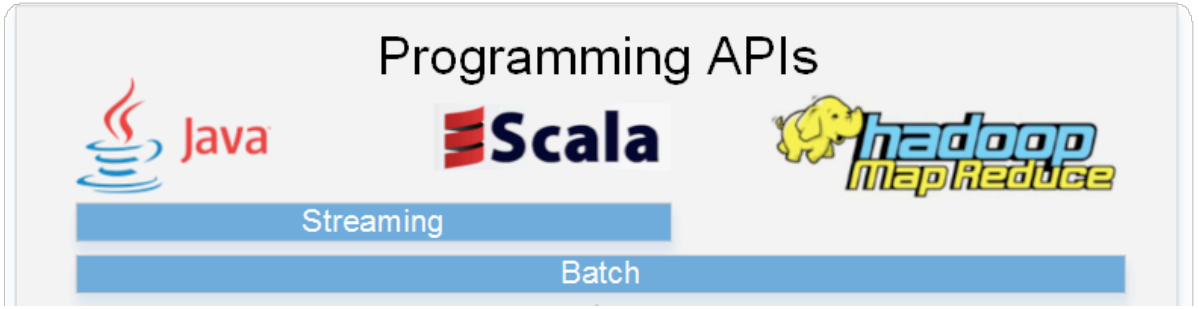
The Apache Way

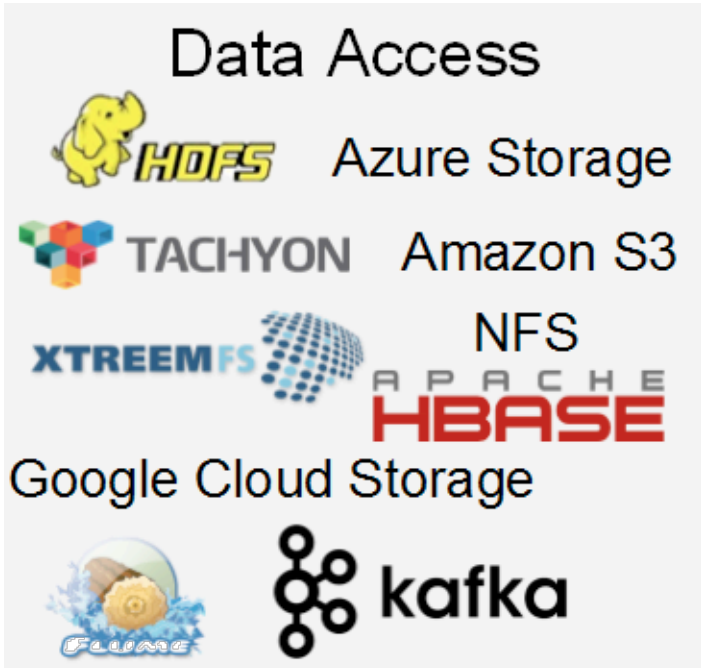
- Independent, non-profit organization
- Community-driven open source software development approach
- Consensus-based decision making
- Public communication and open to new contributors

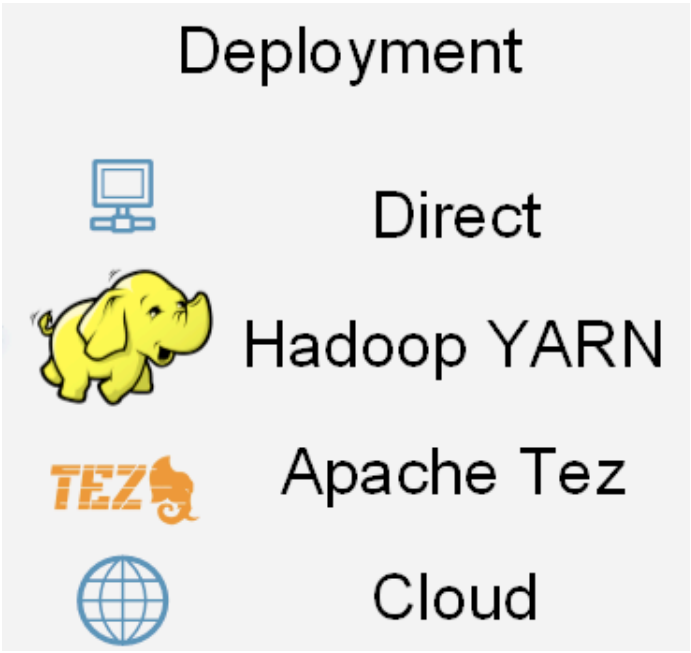
The Apache Software Foundation

Community-led development since 1999.

Integration into the Big Data Stack



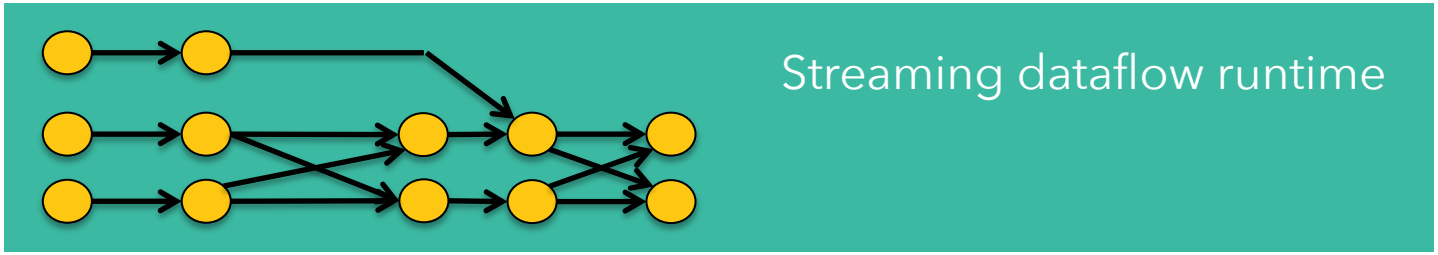




Apache Flink

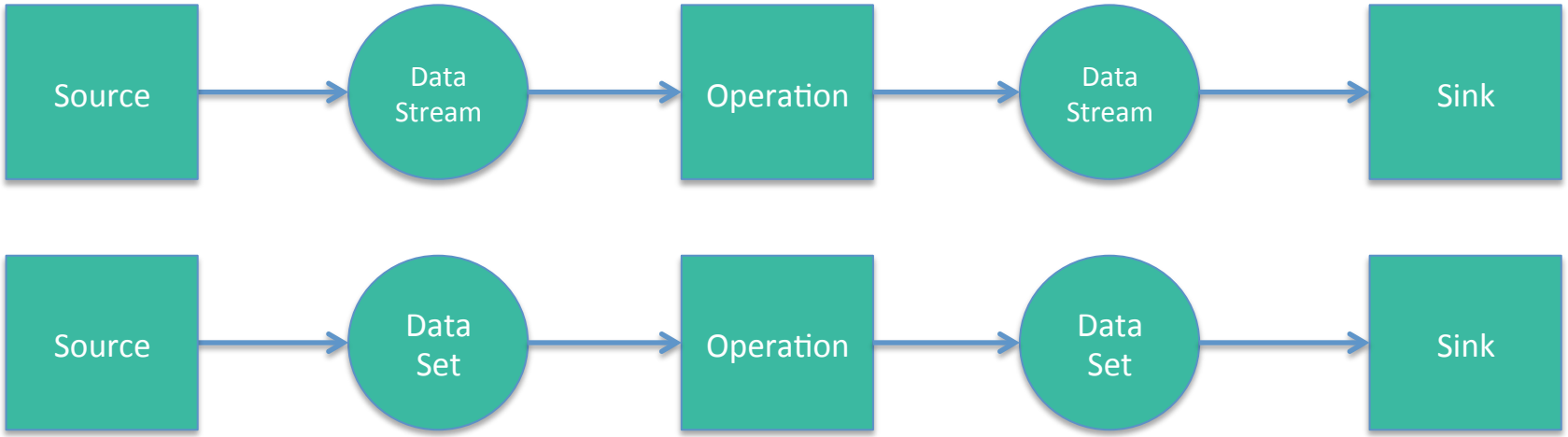
A stream processor

with many applications



What can I do with Flink?

Basic API Concept

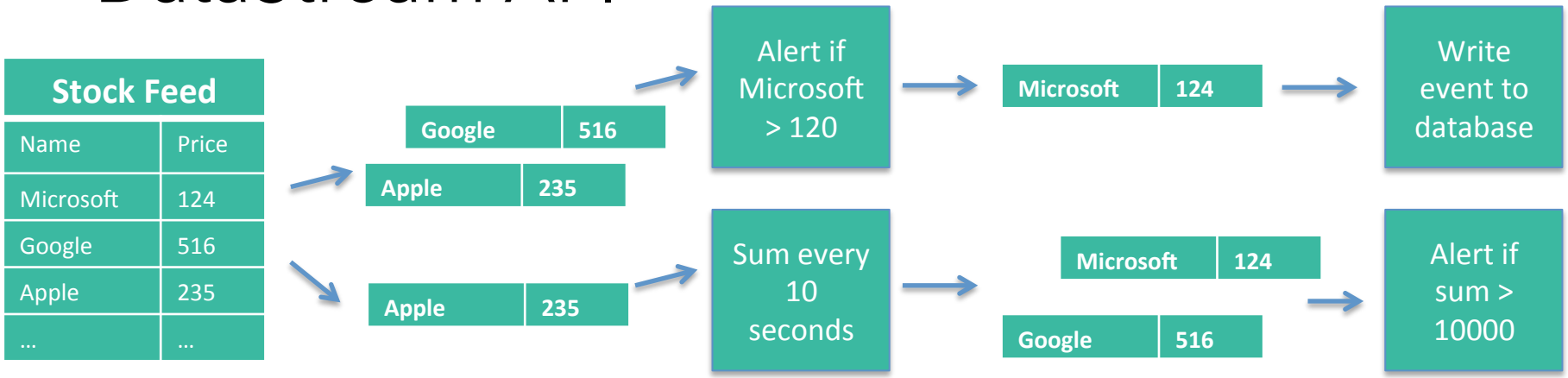


How do I write a Flink program?

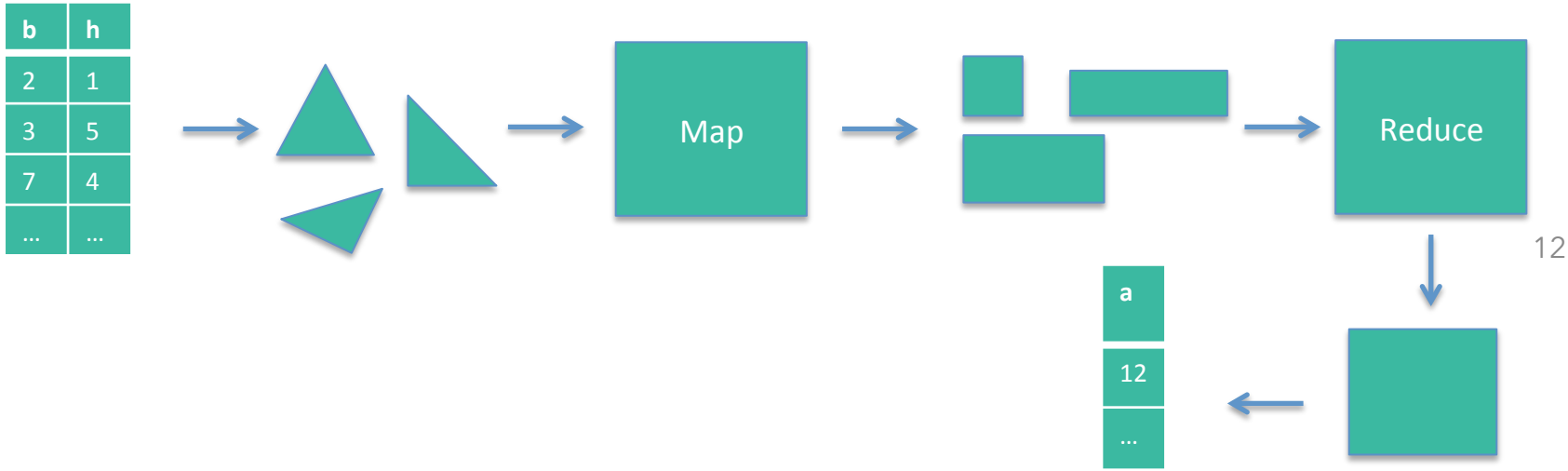
1. Bootstrap sources
2. Apply operations
3. Output to source

Stream & Batch Processing

- DataStream API



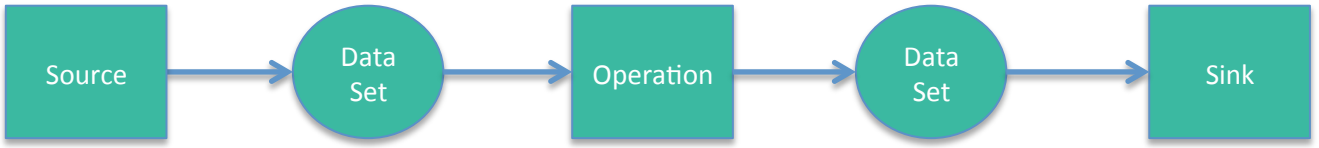
- DataSet API

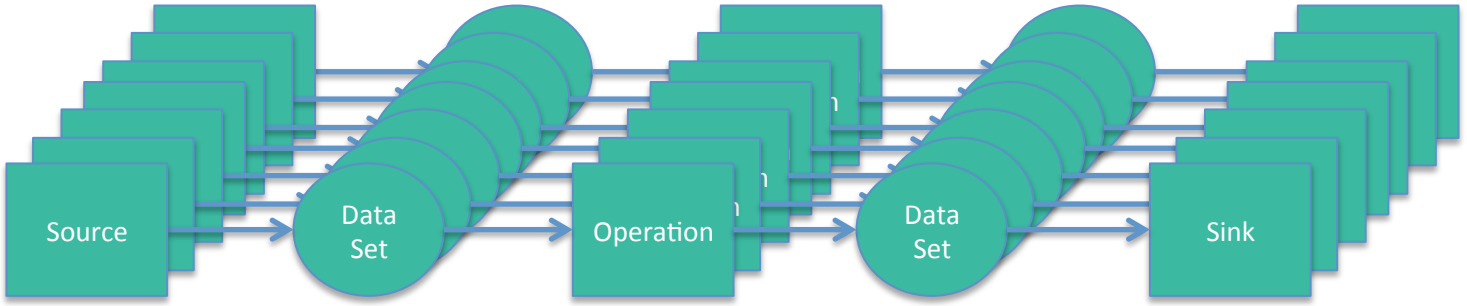


Streaming & Batch

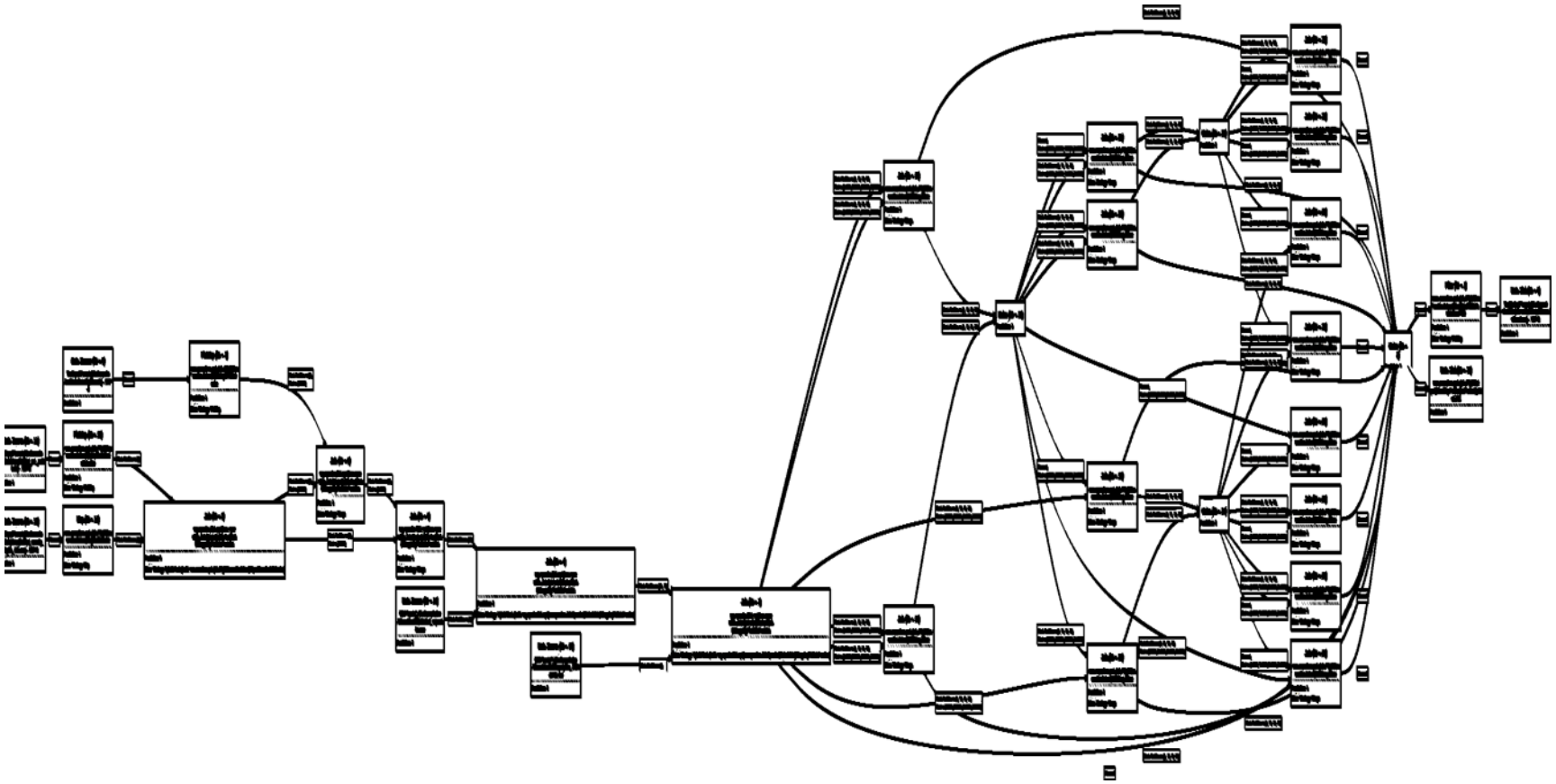
	Streaming	Batch
Input	infinite	finite
Data transfer	pipelined	blocking or pipelined
Latency	low	high

Scaling out


Source
Data Set
Operation
Data Set
Sink


Source
Data Set
Operation
Data Set
Sink

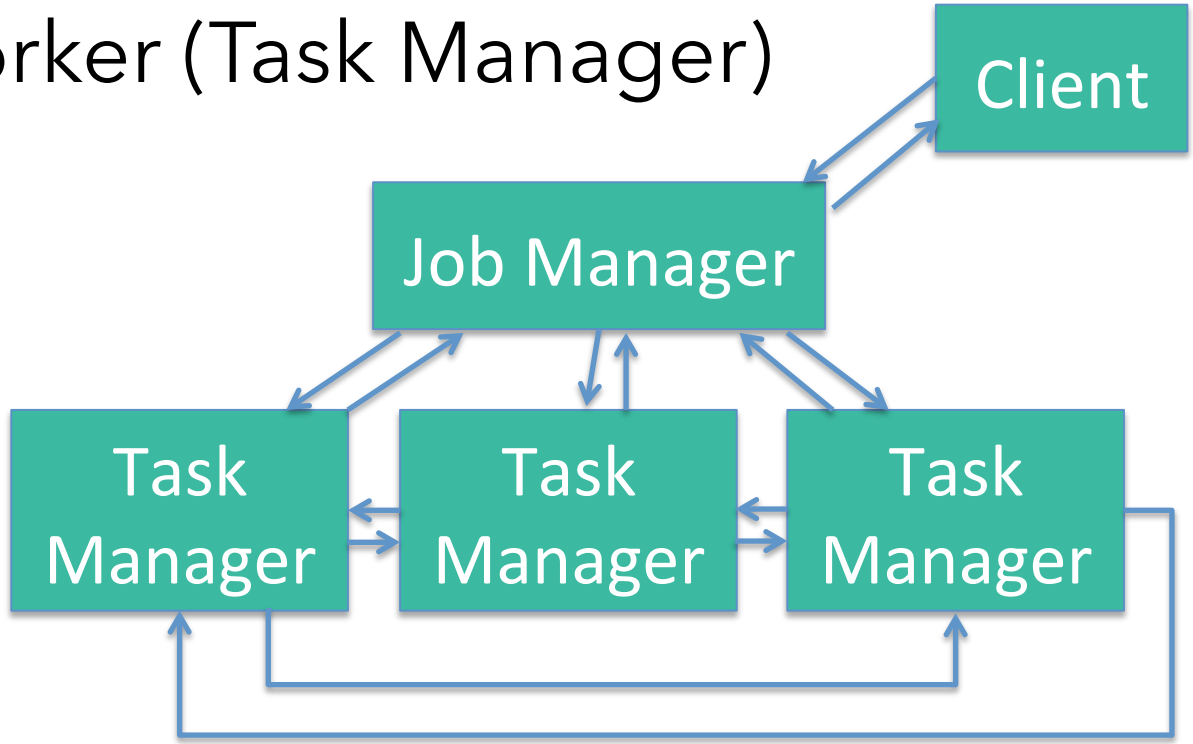
Scaling up



Flink's Architecture

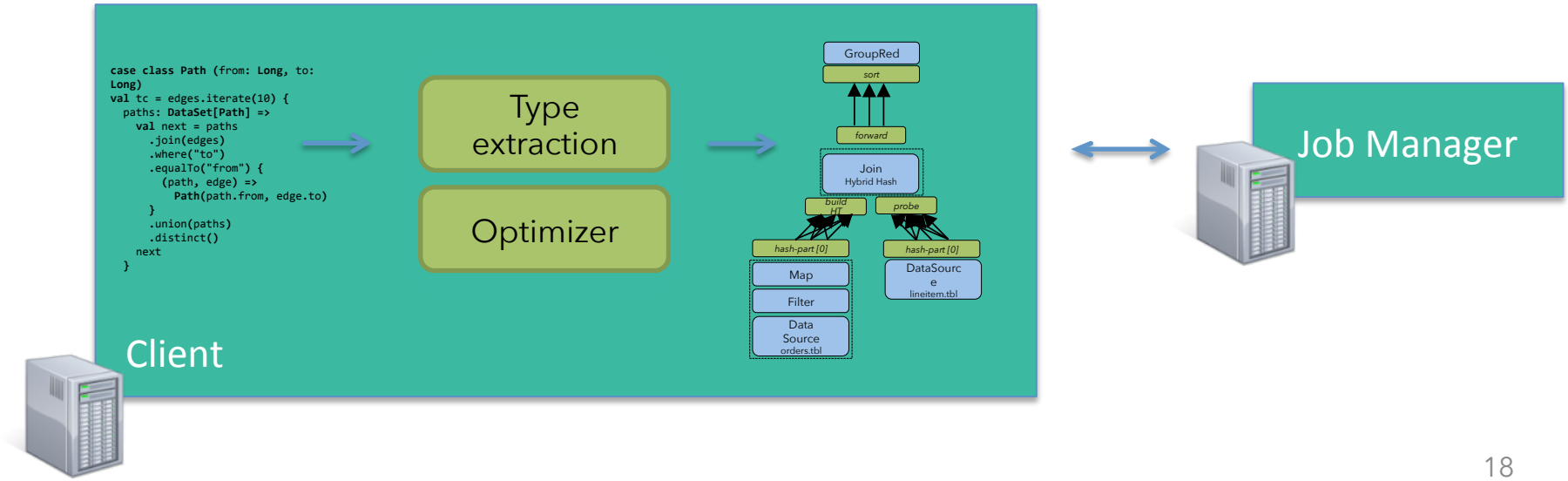
Architecture Overview

- Client
- Master (Job Manager)
- Worker (Task Manager)



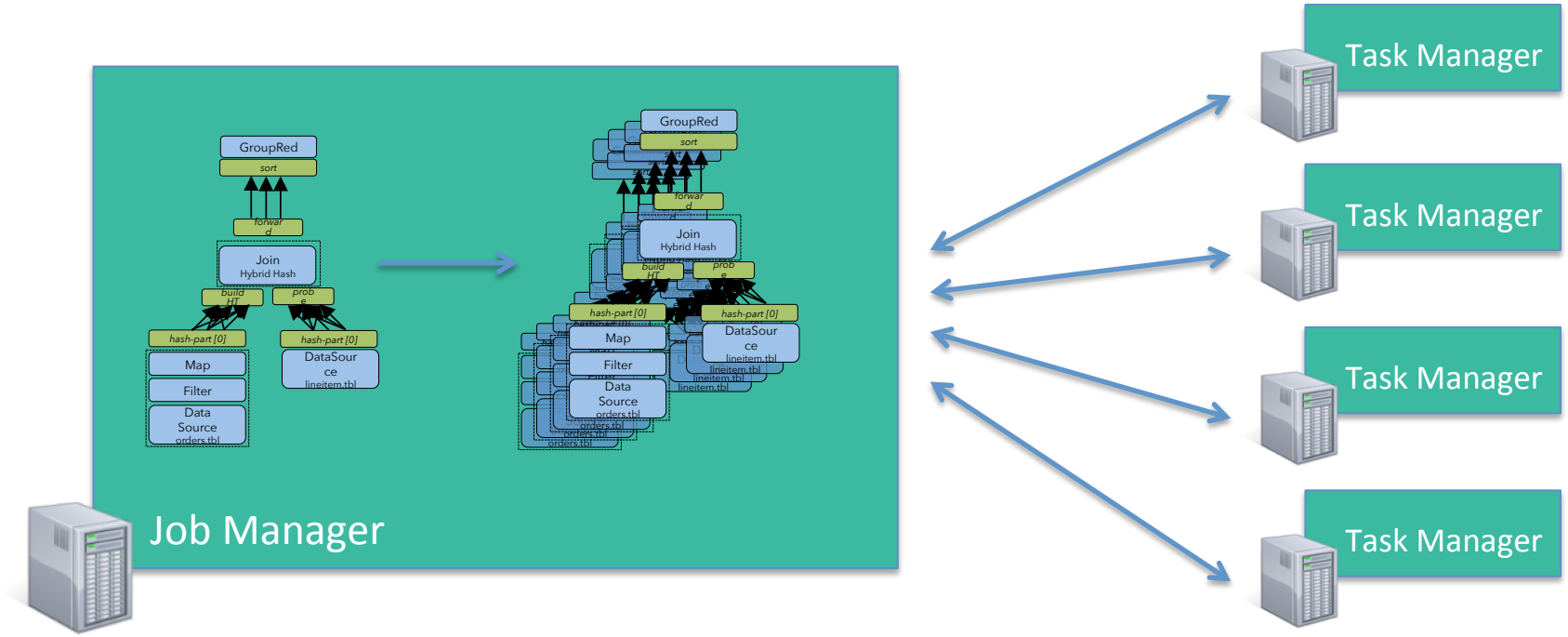
Client

- Optimize
- Construct job graph
- Pass job graph to job manager
- Retrieve job results



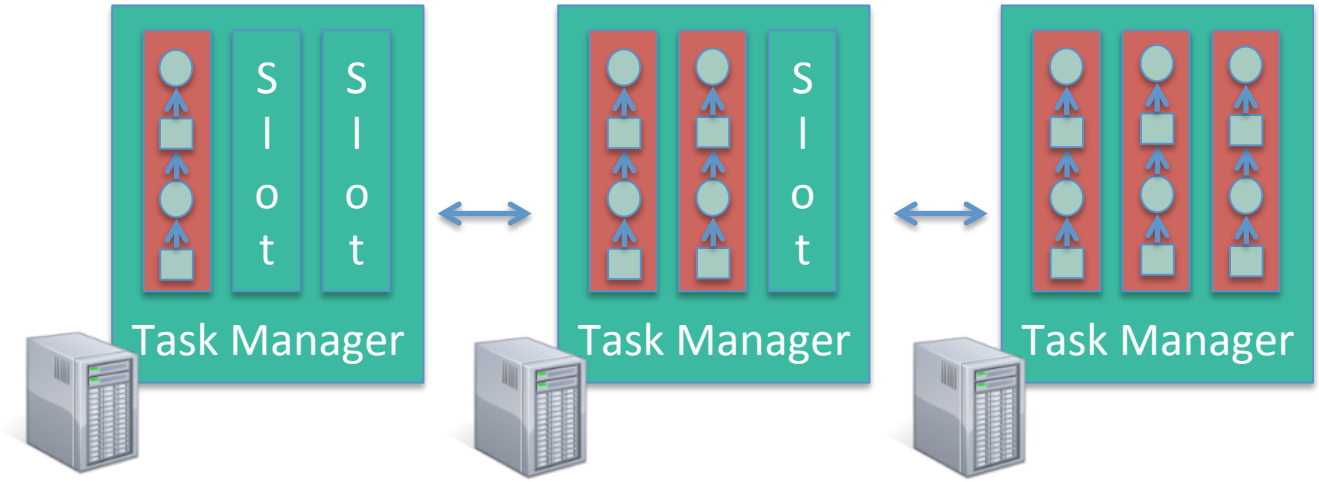
Job Manager

- **Parallelization**: Create Execution Graph
- **Scheduling**: Assign tasks to task managers
- **State tracking**: Supervise the execution



Task Manager

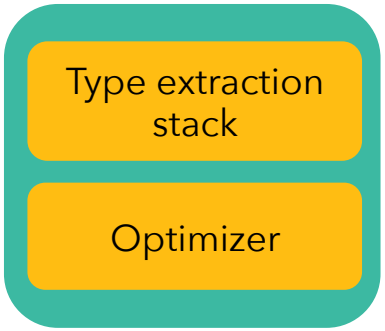
- Operations are split up into **tasks** depending on the specified parallelism
- Each parallel instance of an operation runs in a separate **task slot**
- The scheduler may run several tasks from different operators in one task slot



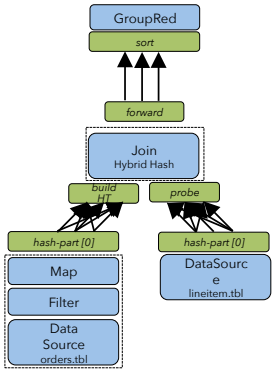
From Program to Execution

```
case class Path (from: Long, to: Long)
val tc = edges.iterate(10) {
  paths: DataSet[Path] =>
    val next = paths
      .join(edges)
      .where("to")
      .equalTo("from") {
        (path, edge) =>
          Path(path.from, edge.to)
      }
      .union(paths)
      .distinct()
    next
  }
```

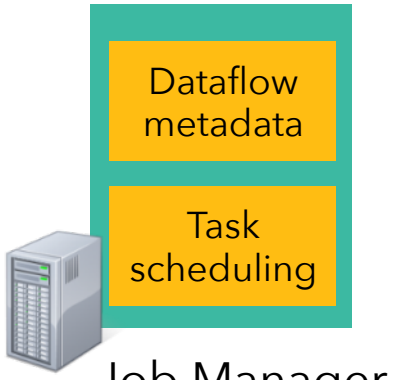
Program



Pre-flight (Client)



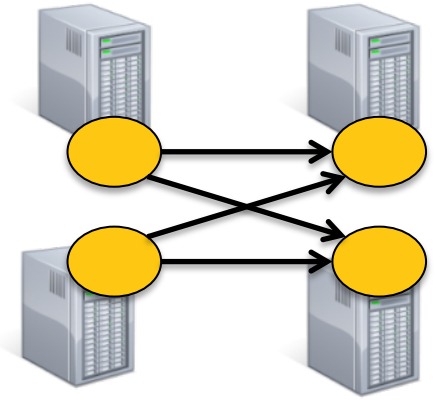
Dataflow Graph



Job Manager

deploy operators

track intermediate results

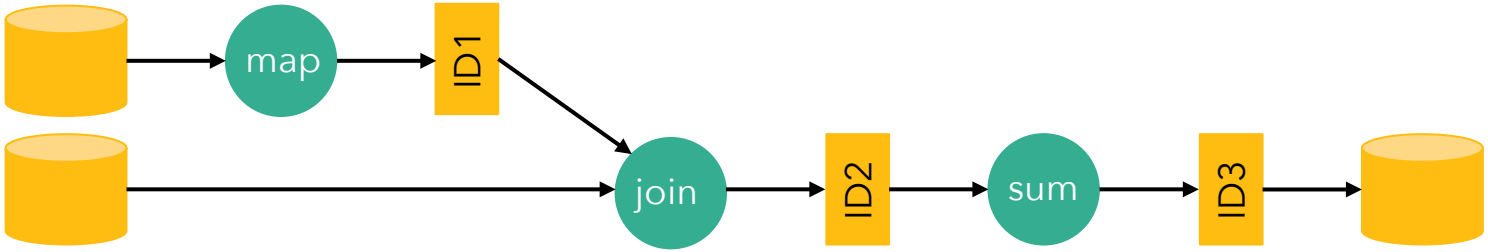


Task Managers

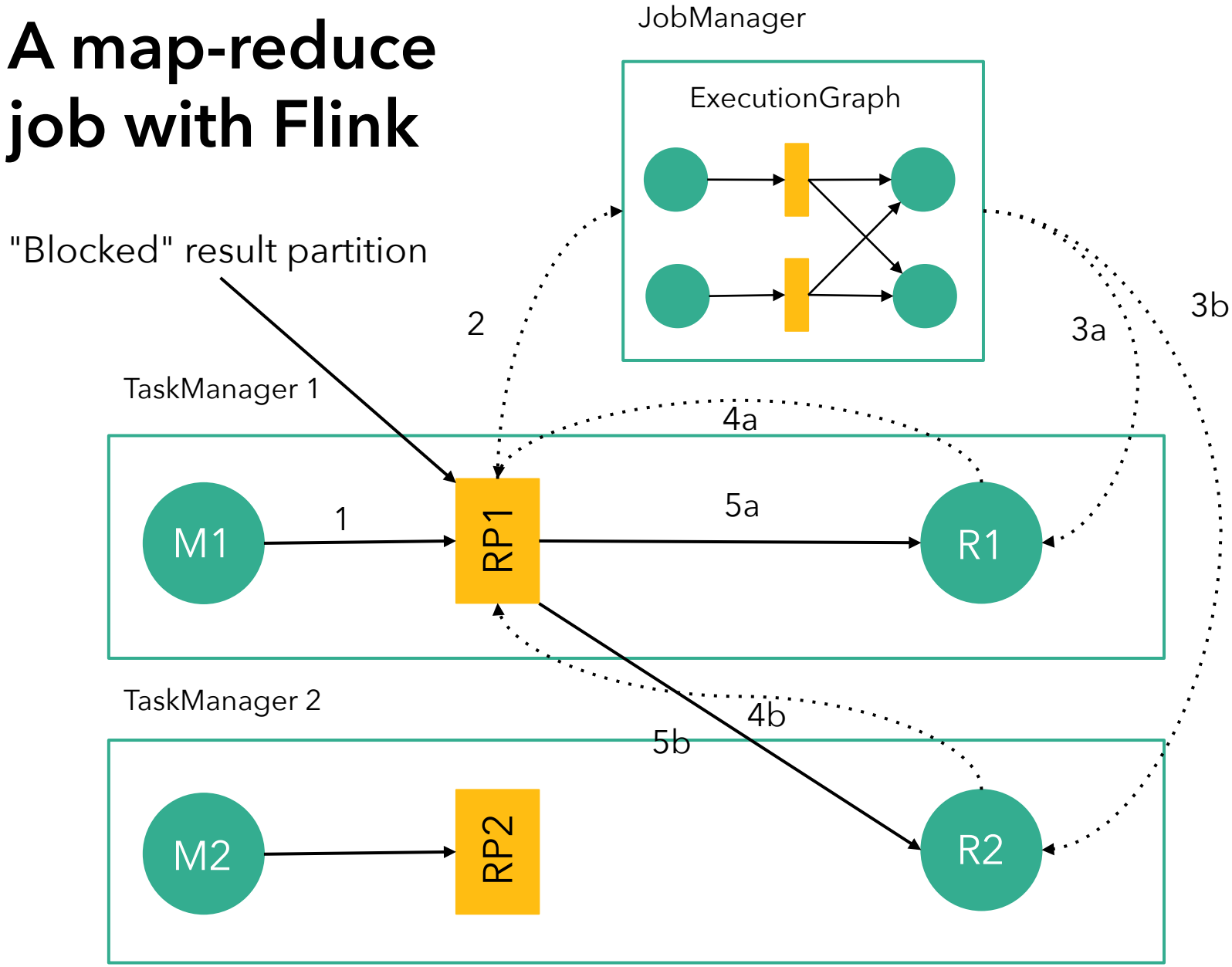
Flink's execution model

Flink execution model

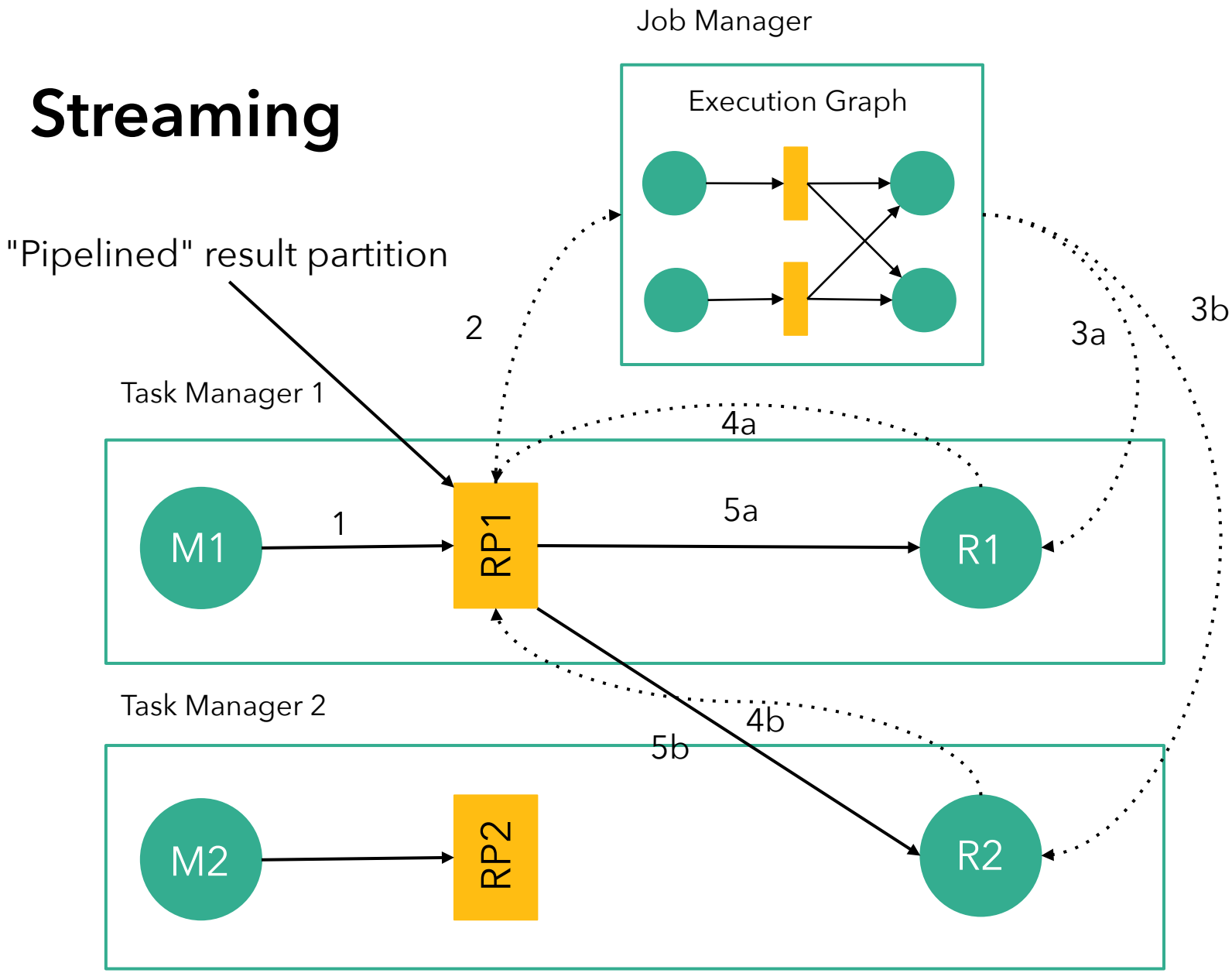
- A program is a graph (DAG) of operators
- Operators = computation + state
- Operators produce intermediate results = logical streams of records
- Other operators can consume those



A map-reduce job with Flink

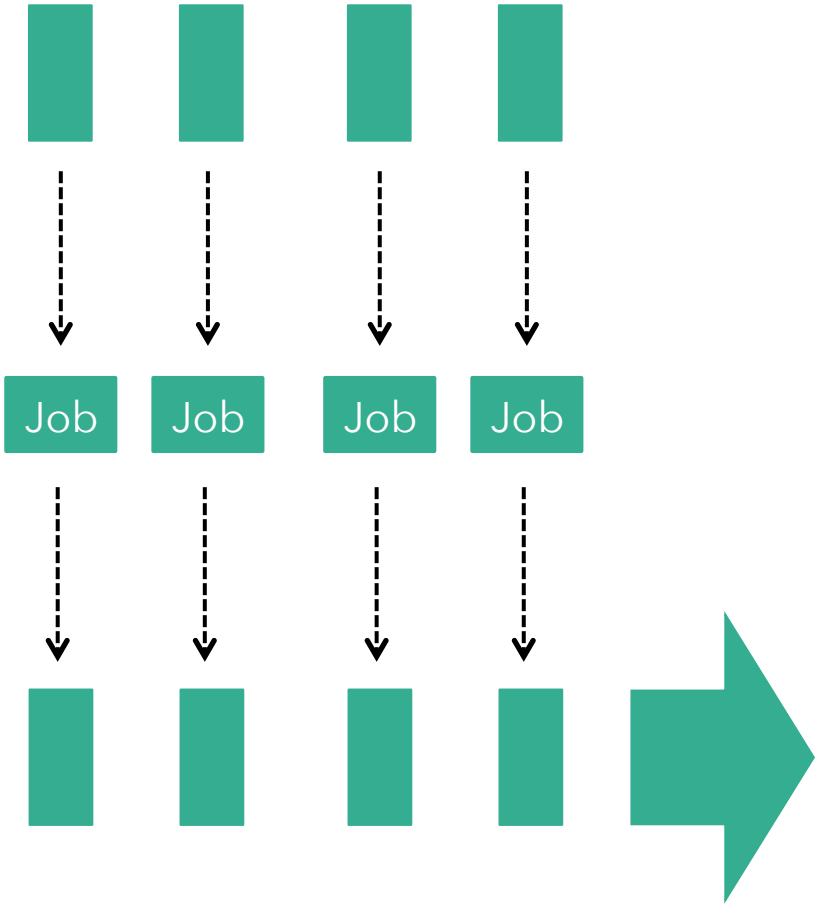


Streaming


Job Manager
Execution Graph
"Pipelined" result partition
2
3a
3b
Task Manager 1
4a
M1
1
RP1
5a
R1
Task Manager 2
4b
5b
M2
RP2
R2

Non-native streaming

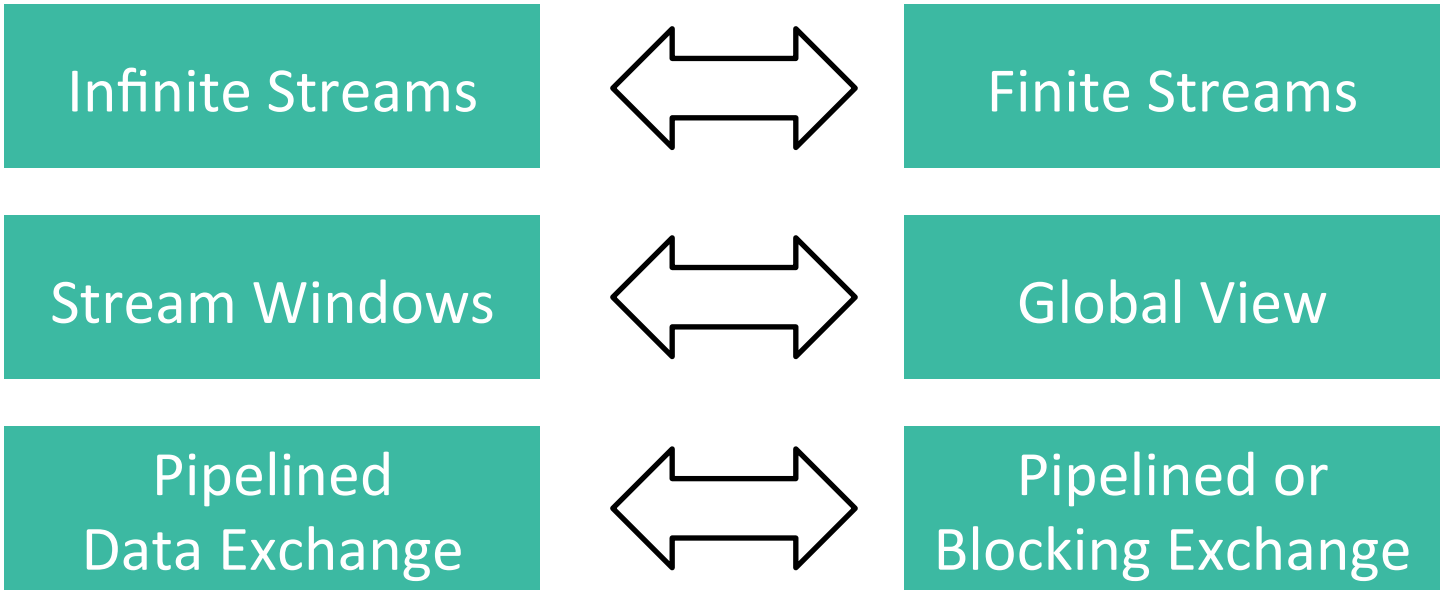
stream discretizer



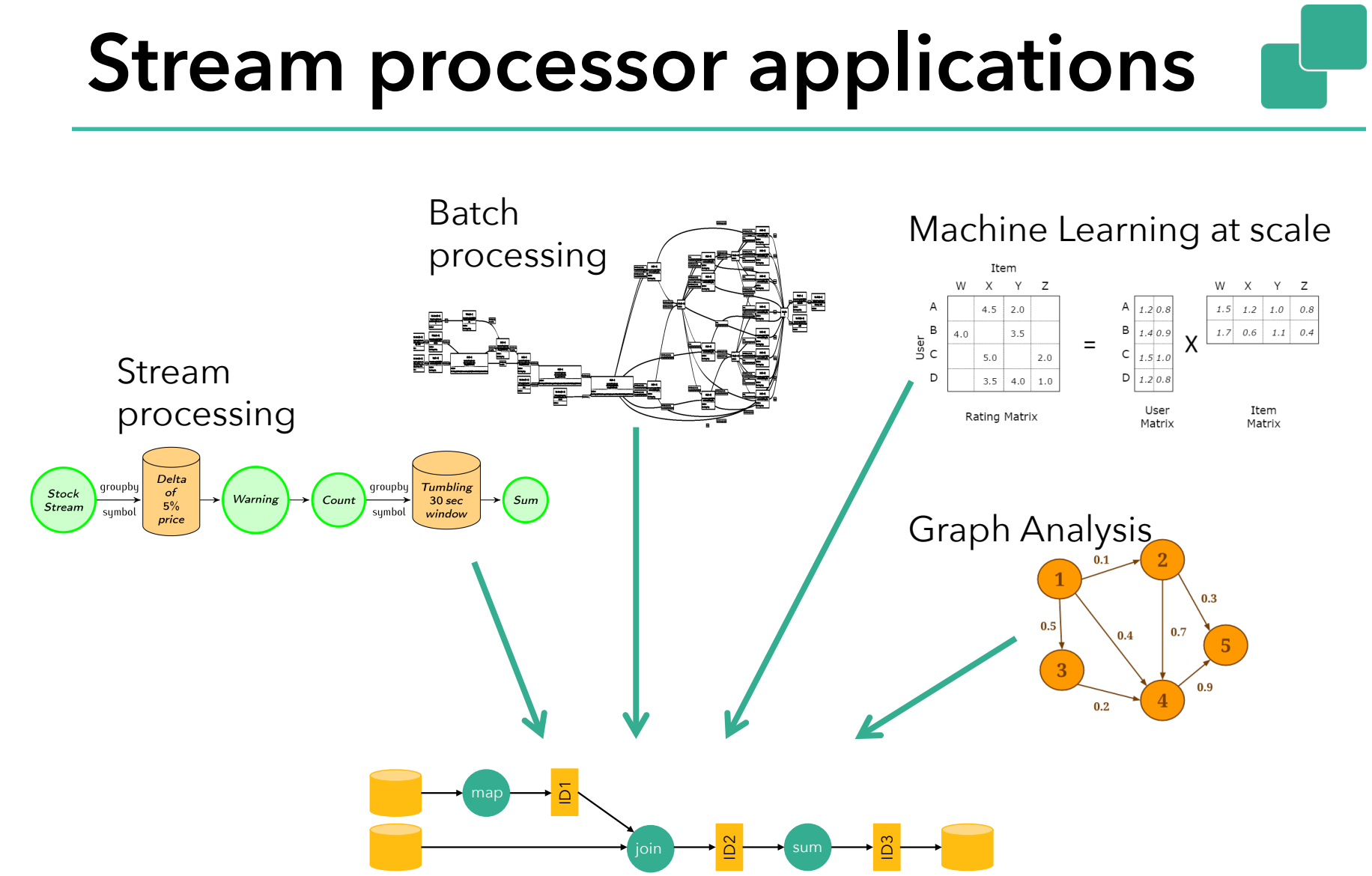

```
while (true) {
  // get next few records
  // issue batch job
}
```

Batch on Streaming

- Batch programs are a special kind of streaming program



Stream processor applications



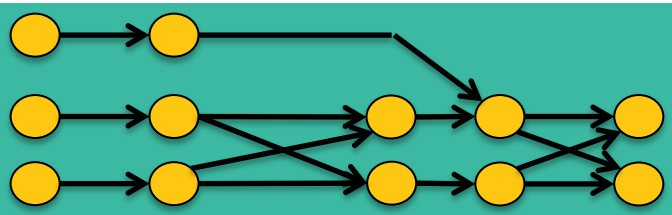
Demo

Introduction to the DataSet API

Flink's APIs

DataStream (Java/Scala)

DataSet (Java/Scala)



Streaming dataflow runtime

API Preview

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

DataSet API (batch):

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

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

DataStream API (streaming):

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

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

WordCount: main method

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

Execution Environment

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

Data Sources

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

Data types

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

Transformations

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

User functions

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

DataSinks

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

Execute!

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final ExecutionEnvironment env =
        ExecutionEnvironment.getExecutionEnvironment();

    // get input data either from file or use example data
    DataSet<String> inputText = env.readTextFile(args[0]);

    DataSet<Tuple2<String, Integer>> counts =
            // split up the lines in tuples containing: (word,1)
            inputText.flatMap(new Tokenizer())
            // group by the tuple field "0"
            .groupBy(0)
            //sum up tuple field "1"
            .reduceGroup(new SumWords());

    // emit result
    counts.writeAsCsv(args[1], "\n", " ");
    // execute program
    env.execute("WordCount Example");
}
```

WordCount: Map

```
public static class Tokenizer
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out) {
        // normalize and split the line
        String[] tokens = value.toLowerCase().split("\\W+");

        // emit the pairs
        for (String token : tokens) {
            if (token.length() > 0) {
                out.collect(
                   new Tuple2<String, Integer>(token, 1));
            }
        }
    }
}
```

WordCount: Map: Interface

```
public static class Tokenizer
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out) {
        // normalize and split the line
        String[] tokens = value.toLowerCase().split("\\W+");

        // emit the pairs
        for (String token : tokens) {
            if (token.length() > 0) {
                out.collect(
                   new Tuple2<String, Integer>(token, 1));
            }
        }
    }
}
```

WordCount: Map: Types

```
public static class Tokenizer
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out) {
        // normalize and split the line
        String[] tokens = value.toLowerCase().split("\\W+");

        // emit the pairs
        for (String token : tokens) {
            if (token.length() > 0) {
                out.collect(
                    new Tuple2<String, Integer>(token, 1));
            }
        }
    }
}
```

WordCount: Map: Collector

```
public static class Tokenizer
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out) {
        // normalize and split the line
        String[] tokens = value.toLowerCase().split("\\W+");

        // emit the pairs
        for (String token : tokens) {
            if (token.length() > 0) {
                out.collect(
                   new Tuple2<String, Integer>(token, 1));
            }
        }
    }
}
```

WordCount: Reduce

```
public static class SumWords implements
 GroupReduceFunction<Tuple2<String, Integer>,
                     Tuple2<String, Integer>> {

    @Override
    public void reduce(Iterable<Tuple2<String, Integer>> values,
                       Collector<Tuple2<String, Integer>> out) {
          int count = 0;
          String word = null;
          for (Tuple2<String, Integer> tuple : values) {
              word = tuple.f0;
              count++;
          }
          out.collect(new Tuple2<String, Integer>(word, count));
    }

}
```

WordCount: Reduce: Interface

```
public static class SumWords implements
 GroupReduceFunction<Tuple2<String, Integer>,
                     Tuple2<String, Integer>> {

    @Override
    public void reduce(Iterable<Tuple2<String, Integer>> values,
                       Collector<Tuple2<String, Integer>> out) {
          int count = 0;
          String word = null;
          for (Tuple2<String, Integer> tuple : values) {
              word = tuple.f0;
              count++;
          }
          out.collect(new Tuple2<String, Integer>(word, count));
    }

}
```

WordCount: Reduce: Types

```
public static class SumWords implements
 GroupReduceFunction<Tuple2<String, Integer>,
                     Tuple2<String, Integer>> {

    @Override
    public void reduce(Iterable<Tuple2<String, Integer>> values,
                       Collector<Tuple2<String, Integer>> out) {
          int count = 0;
          String word = null;
          for (Tuple2<String, Integer> tuple : values) {
              word = tuple.f0;
              count++;
          }
          out.collect(new Tuple2<String, Integer>(word, count));
    }

}
```

WordCount: Reduce: Collector

```
public static class SumWords implements
 GroupReduceFunction<Tuple2<String, Integer>,
                     Tuple2<String, Integer>> {

    @Override
    public void reduce(Iterable<Tuple2<String, Integer>> values,
                       Collector<Tuple2<String, Integer>> out) {
          int count = 0;
          String word = null;
          for (Tuple2<String, Integer> tuple : values) {
              word = tuple.f0;
              count++;
          }
          out.collect(new Tuple2<String, Integer>(word, count));
    }

}
```

DataSet API Concepts

Data Types

- Basic Java Types
  - String, Long, Integer, Boolean,…
  - Arrays

- Composite Types
  - Tuple
  - PoJo (Java objects)
  - Custom type

Tuples

- The easiest, lightweight, and generic way of encapsulating data in Flink
- Tuple1 up to Tuple25

```
Tuple3<String, String, Integer> person =
        new Tuple3<>("Max", "Mustermann", 42);

// zero based index!
String firstName = person.f0;
String secondName = person.f1;
Integer age = person.f2;
```

Transformations: Map

```
DataSet<Integer> integers = env.fromElements(1, 2, 3, 4);

// Regular Map - Takes one element and produces one element
DataSet<Integer> doubleIntegers =
     integers.map(new MapFunction<Integer, Integer>() {
     @Override
          public Integer map(Integer value) {
            return value * 2;
          }
     });

doubleIntegers.print();
> 2, 4, 6, 8


// Flat Map - Takes one element and produces zero, one, or more elements.
DataSet<Integer> doubleIntegers2 =

     integers.flatMap(new FlatMapFunction<Integer, Integer>() {
          @Override
          public void flatMap(Integer value, Collector<Integer> out) {
               out.collect(value * 2);
          }
     });

doubleIntegers2.print();
> 2, 4, 6, 8
```

Transformations: Filter

```
// The DataSet
DataSet<Integer> integers = env.fromElements(1, 2, 3, 4);

DataSet<Integer> filtered =

    integers.filter(new FilterFunction<Integer>() {
        @Override
        public boolean filter(Integer value) {
            return value != 3;
        }
    });

integers.print();
> 1, 2, 4
```

Groupings and Reduce

- DataSets can be split into groups
- Groups are defined using a common key

Name	Age
Stephan	18
Fabian	23
Julia	27
Romeo	27
Anna	18

```
// (name, age) of employees
DataSet<Tuple2<String, Integer>> employees = …

// group by second field (age)
DataSet<Integer, Integer> grouped = employees.groupBy(1)
  // return a list of age groups with its counts
  .reduceGroup(new CountSameAge());
```

AgeGroup	Count
18	2
23	1
27	2

GroupReduce

```
public static class CountSameAge implements GroupReduceFunction
<Tuple2<String, Integer>, Tuple2<Integer, Integer>> {

    @Override
    public void reduce(Iterable<Tuple2<String, Integer>> values,
                       Collector<Tuple2<Integer, Integer>> out) {

        Integer ageGroup = 0;
        Integer countsInGroup = 0;

        for (Tuple2<String, Integer> person : values) {
            ageGroup = person.f1;
            countsInGroup++;
        }

        out.collect(new Tuple2<Integer, Integer>
                                            (ageGroup, countsInGroup));
    }
}
```

Joining two DataSets

Authors		
Id	Name	email
1	Fabian	fabian@..
2	Julia	julia@...
3	Max	max@...
4	Romeo	romeo@.

Posts		
Title	Content	Author id
…	…	2
..	..	4
..	..	4
..	..	1
..	..	2

```
// authors (id, name, email)
DataSet<Tuple3<Integer, String, String>> authors = ..;
// posts (title, content, author_id)
DataSet<Tuple3<String, String, Integer>> posts = ..;

DataSet<Tuple2<
            Tuple3<Integer, String, String>,
            Tuple3<String, String, Integer>
        >> archive = authors.join(posts).where(0).equalTo(2);
```

Joining two DataSets

```
// authors (id, name, email)
DataSet<Tuple3<Integer, String, String>> authors = ..;
// posts (title, content, author_id)
DataSet<Tuple3<String, String, Integer>> posts = ..;

DataSet<Tuple2<
            Tuple3<Integer, String, String>,
            Tuple3<String, String, Integer>
        >> archive = authors.join(posts).where(0).equalTo(2);
```

Archive					
Id	Name	email	Title	Content	Author id
1	Fabian	fabian@..	..	..	1
2	Julia	julia@...	..	..	2
2	Julia	julia@...	..	..	2
3	Romeo	romeo@...	..	..	4
4	Romeo	romeo@.	..	..	4

Join with join function

```
// authors (id, name, email)
DataSet<Tuple3<Integer, String, String>> authors = ..;
// posts (title, content, author_id)
DataSet<Tuple3<String, String, Integer>> posts = ..;

// (title, author name)
DataSet<Tuple2<String, String>> archive =
    authors.join(posts).where(0).equalTo(2)
    .with(new PostsByUser());

public static class PostsByUser implements
  JoinFunction<Tuple3<Integer, String, String>,
               Tuple3<String, String, Integer>,
               Tuple2<String, String>> {
    @Override
    public Tuple2<String, String> join(
        Tuple3<Integer, String, String> left,
        Tuple3<String, String, Integer> right) {
            return new Tuple2<String, String>(left.f1, right.f0);
    }
}
```

Archive	
Name	Title
Fabian	..
Julia	..
Julia	..
Romeo	..
Romeo	..

Data Sources / Data Sinks

Data Sources

Text

- `readTextFile("/path/to/file")`

CSV

- `readCsvFile("/path/to/file")`

Collection

- `fromCollection(collection)`
- `fromElements(1,2,3,4,5)`

Data Sources: Collections

```
ExecutionEnvironment env =
    ExecutionEnvironment.getExecutionEnvironment();

// read from elements
DataSet<String> names = env.fromElements("Some", "Example",
"Strings");

// read from Java collection
List<String> list = new ArrayList<String>();
list.add("Some");
list.add("Example");
list.add("Strings");

DataSet<String> names = env.fromCollection(list);
```

Data Sources: File-Based

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();


// read text file from local or distributed file system
DataSet<String> localLines =
    env.readTextFile(”/path/to/my/textfile");


// read a CSV file with three fields
DataSet<Tuple3<Integer, String, Double>> csvInput =
                env.readCsvFile(“/the/CSV/file")
               .types(Integer.class, String.class, Double.class);


// read a CSV file with five fields, taking only two of them
DataSet<Tuple2<String, Double>> csvInput =
               env.readCsvFile(“/the/CSV/file")
               // take the first and the fourth field
               .includeFields("10010")
               .types(String.class, Double.class);
```

Data Sinks

Text

- `writeAsText("/path/to/file")`
- `writeAsFormattedText("/path/to/file", formatFunction)`

CSV

- `writeAsCsv("/path/to/file")`

Return data to the Client

- `Print()`
- `Collect()`
- `Count()`

Data Sinks (lazy)

- Lazily executed when `env.execute()` is called

```
DataSet<…> result;

// write DataSet to a file on the local file system
result.writeAsText(“/path/to/file");

// write DataSet to a file and overwrite the file if it exists
result.writeAsText("/path/to/file”,FileSystem.WriteMode.OVERWRITE);

// tuples as lines with pipe as the separator "a|b|c”
result.writeAsCsv("/path/to/file", "\n", "|");

// this wites values as strings using a user-defined TextFormatter
object
result.writeAsFormattedText("/path/to/file",
        new TextFormatter<Tuple2<Integer, Integer>>() {
            public String format (Tuple2<Integer, Integer> value) {
                return value.f1 + " - " + value.f0;
            }
        });
```

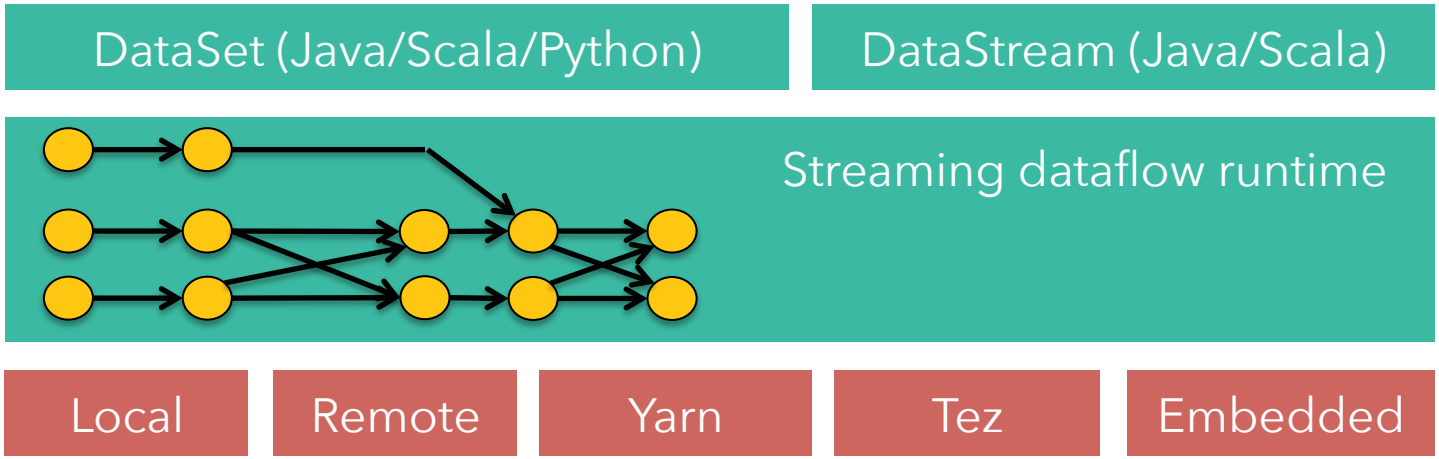
Data Sinks (eager)

- Eagerly executed

```
DataSet<Tuple2<String, Integer> result;


// print
result.print();

// count
int numberOfElements = result.count();

// collect
List<Tuple2<String, Integer> materializedResults = result.collect();
```

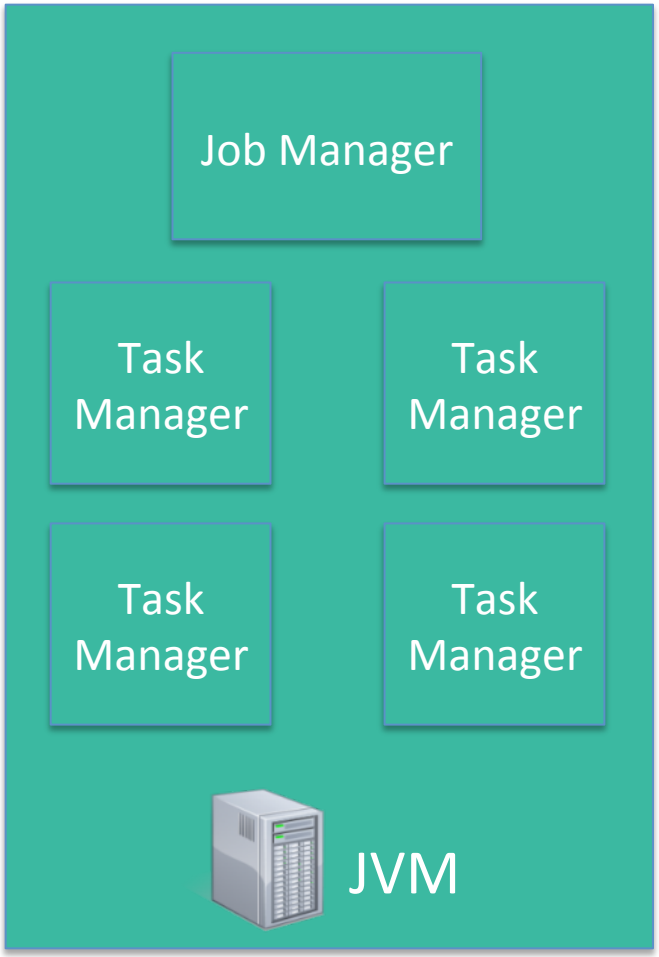
Execution Setups

Ways to Run a Flink Program



Local Execution

- Starts local Flink cluster
- All processes run in the same JVM
- Behaves just like a regular Cluster
- Very useful for developing and debugging

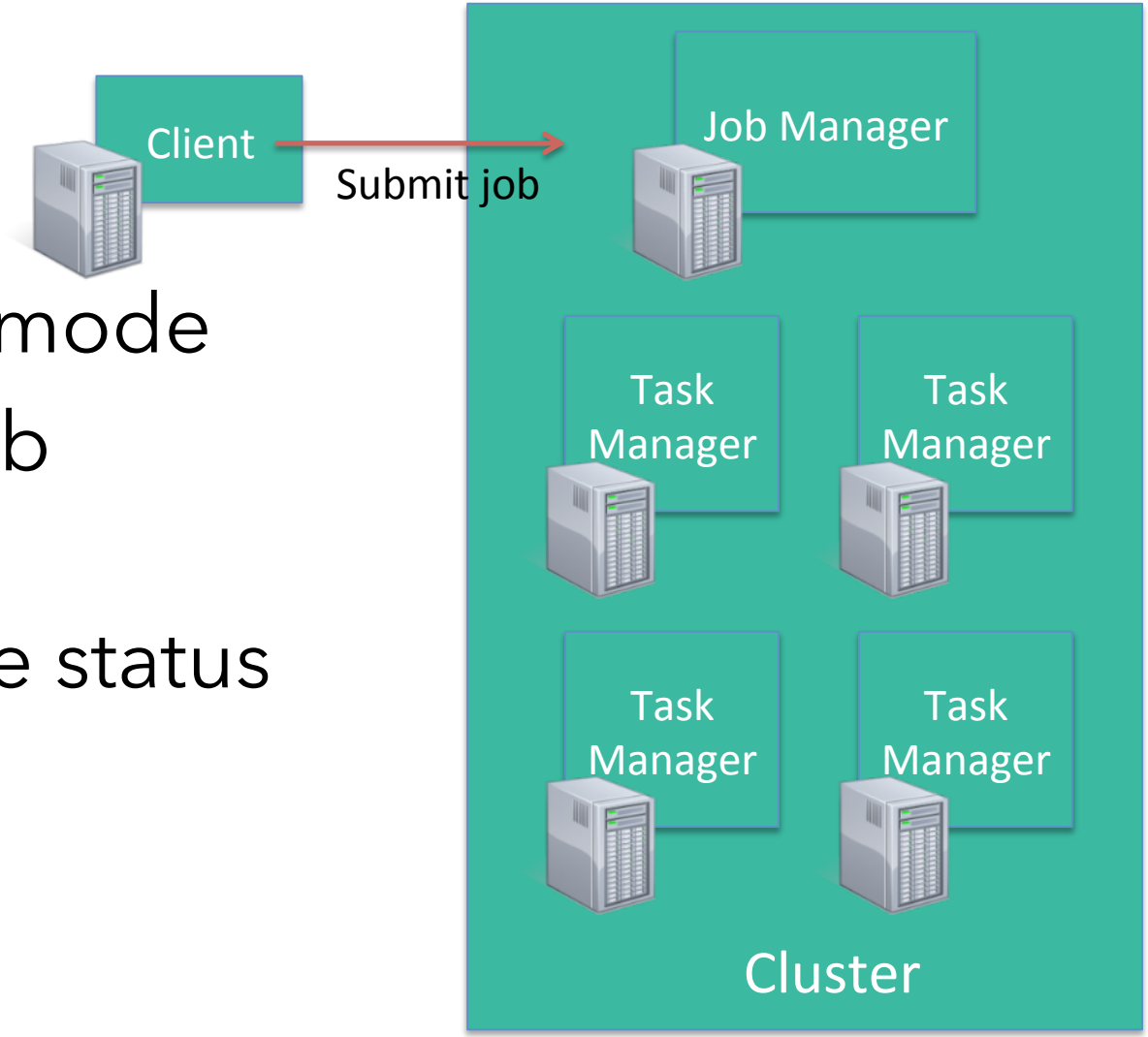


Embedded Execution

- Runs operators on simple Java collections
- Lower overhead
- Does not use memory management
- Useful for testing and debugging

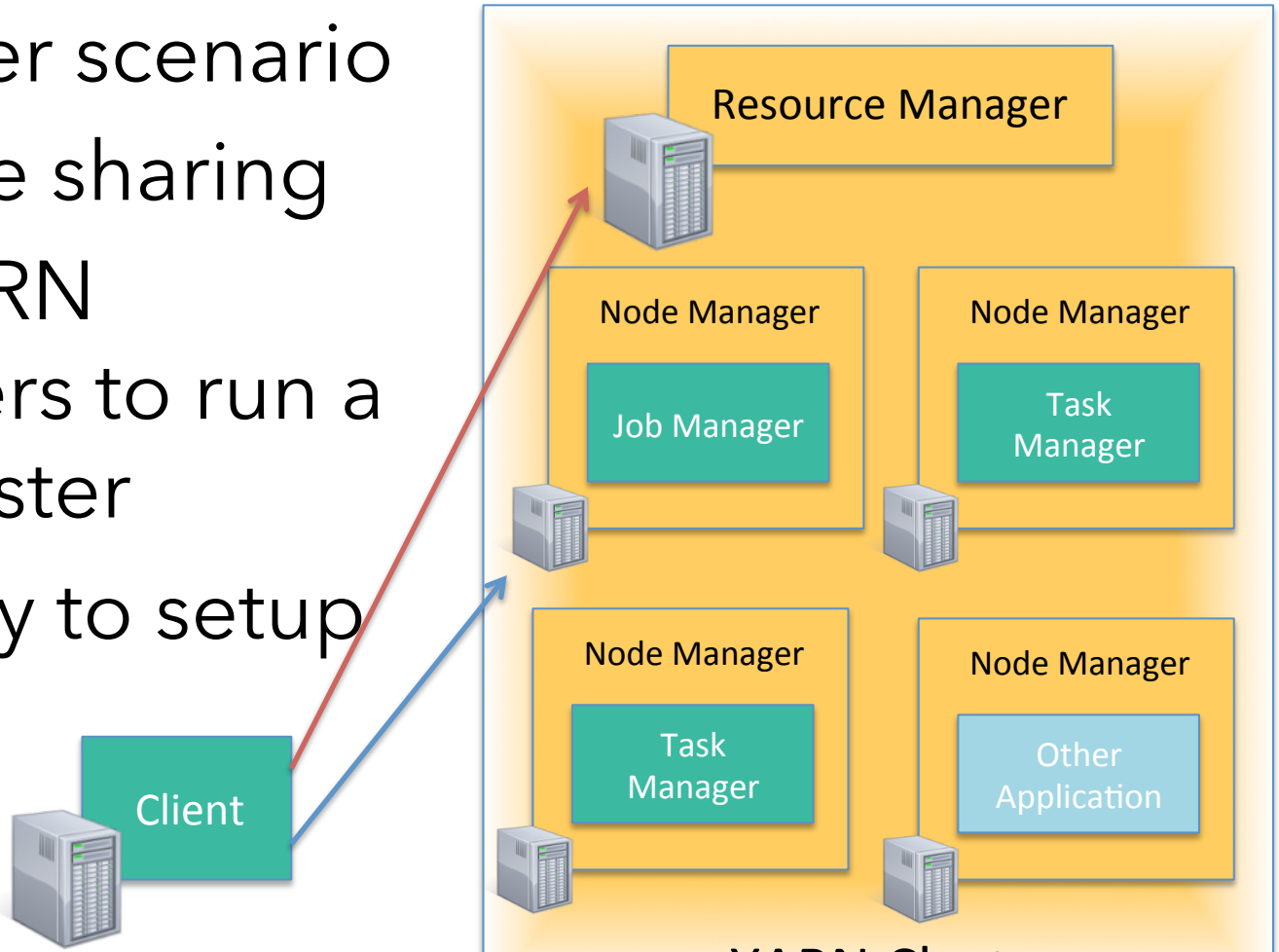
Remote Execution

- The cluster mode
- Submit a Job remotely
- Monitors the status of the job



YARN Execution

- Multi user scenario
- Resource sharing
- Uses YARN containers to run a Flink cluster
- Very easy to setup Flink



Execution

- Leverages Apache Tez's runtime
- Built on top of YARN
- Good YARN citizen
- Fast path to elastic deployments
- Slower than Flink

http://dataArtisans.github.io/eit-summerschool-15

Exercises and Hands-on

Closing

tl;dr: What was this about?

- The case for Flink
  - Low latency
  - High throughput
  - Fault-tolerant
  - Easy to use APIs, library ecosystem
  - Growing community
- A stream processor that is great for batch analytics as well

I ♥  , do you?

- Get involved and start a discussion on Flink's mailing list
- { user, dev }@flink.apache.org

- Subscribe to news@flink.apache.org
- Follow flink.apache.org/blog and @ApacheFlink on Twitter

Flink Forward

BERLIN 12/13 OCT 2015

flink-forward.org

October 12-13, 2015

Call for papers deadline:
August 14, 2015

Discount code: FlinkEITSummerSchool25

Thank you for listening!