Declarative Data Processing With Java in Apache Flink

Vasia Kalavri- kalavri@kth.se
PhD student @KTH
Apache Flink PMC

Overview

- Introduction to Apache Flink
- The DataSet API
- Runtime Execution
- Data Exchange
- Memory Management

Apache Flink

introduction

What is Apache Flink?

- Distributed data processing on a dataflow *streaming* processing engine
- Java, Scala & Python APIs, SQL-like DSL (Table API)
- Batch and Streaming Analytics
- Runs locally, on your cluster, on YARN
- Performs well even when memory runs out

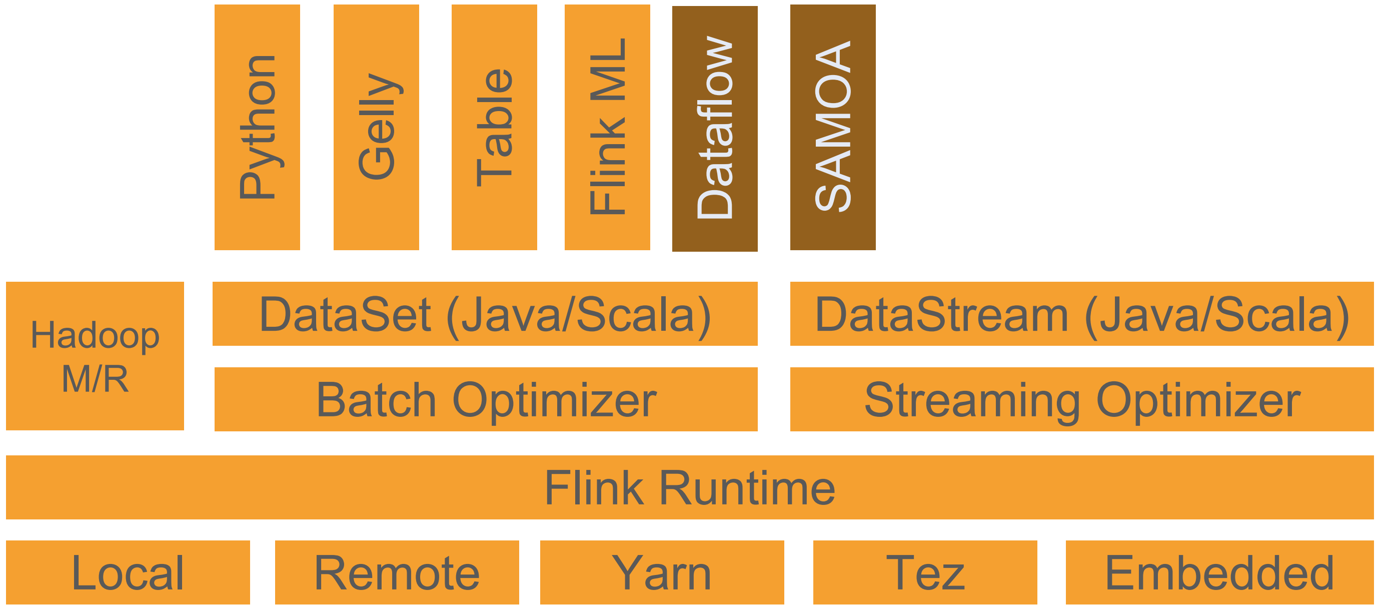
What can you do with Flink?

- Text processing
- Information retrieval
- Web search
- Graph processing
- Machine learning
- Social network analysis
- Large-scale relational queries
- Business intelligence
- …

Flink in the Analytics Ecosystem

Applications	Hive, Cascading, Giraph, Mahout, Pig, Crunch
Data processing engines	MapReduce, Flink, Spark, Storm, Tez
App and resource management	Yarn, Mesos
Storage, streams	HDFS, HBase, Kafka, …

The Flink Stack



Available Transformations

- map, flatMap
- filter
- reduce, reduceGroup
- join
- coGroup
- aggregate
- cross
- project
- distinct
- union
- iterate
- iterateDelta
- …

The DataSet API

declarative data processing with Java

Declarative Programming

Express the *logic* of computation, without describing the *control flow*.

- Describe **what** we want the result to be and let the system decide **how** to produce it
- SQL, functional programming languages, logic programming languages, HTML
- Shorter programs, easier to follow program logic, let the system deal with optimization

Declarative Data Processing

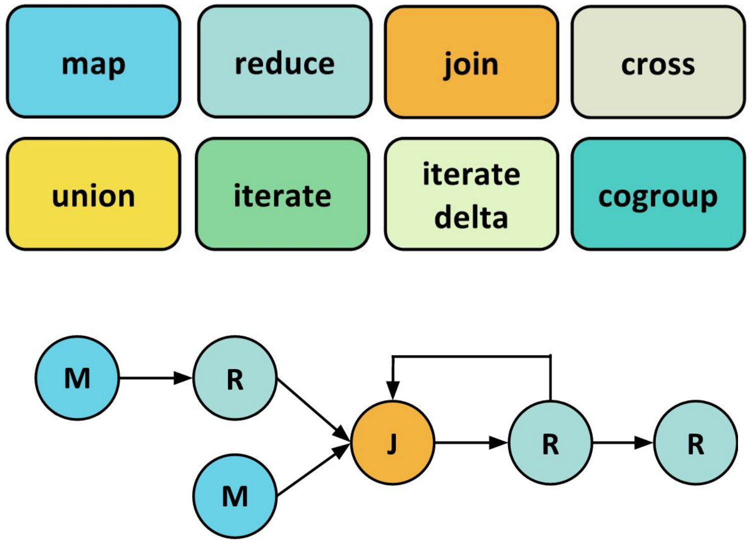
- Users of data processing tools are not necessarily programmers
- Focus on *what* you compute, not *how* you compute it

```
Foreach Person p in A
  if p.age > 18
  add p to B
return B
```

vs.

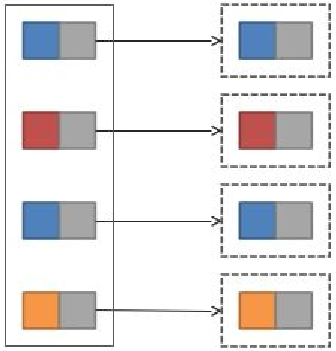
```
B = A.filter(“age” > 18)
```

Operators and Data Flows



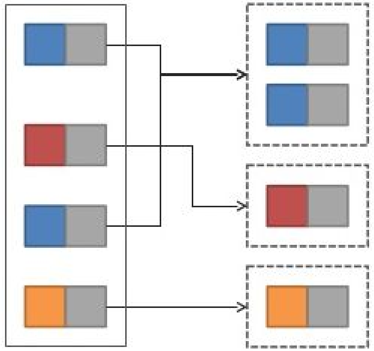
- Operators represent common data analysis tasks
- They can form advanced data-flow graphs, including loops
- A Flink program is translated into a data-flow graph
- API constructs are mapped to runtime operators

The Map Operator



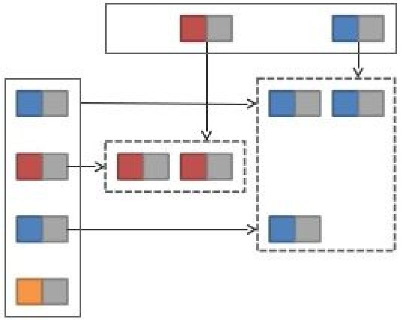
- Transforms each record individually.
- The operation may return zero, one, or many records.

The Reduce Operator



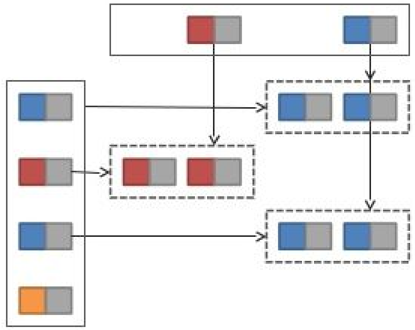
- Groups the records on one or more fields and transforms each group.
- Aggregations that combine the records in the group into a single record.

The CoGroup Operator



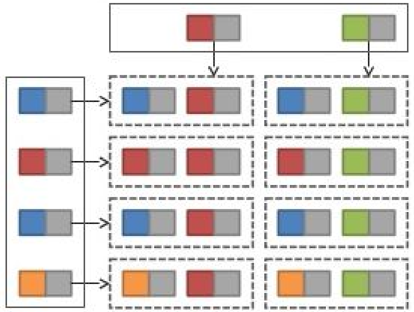
- Two-dimensional variant of the reduce operation. Groups each input on one or more fields and then joins the groups.
- The transformation function is called per pair of groups.

The Join Operator



- Joins two data sets on one or more fields.
- The transformation function gets each pair of joining records.

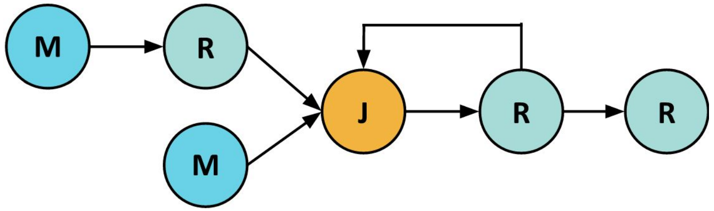
The Cross Operator



- Builds the cartesian product (cross product) of two inputs.
- The transformation function gets all pairs of records in the product.

Construct the dataflow graph in a Flink program

- Each program has an input operator (source) and an output operator (sink)
- Each operator has
  - one (map, reduce) or two input operators (join, cogroup, cross)
  - one output operator
- Connect the operators and return the resulting plan



```
public Plan getPlan(String... args) {
String dataInput = args[1];
     String output    = args[2];

     FileDataSource source = new FileDataSource(new TextInputFormat(), dataInput);

     MapOperator mapper = MapOperator.builder(new TokenizeLine())
          .input(source)
          .build();

     ReduceOperator reducer = ReduceOperator.builder(Words.class, String.class, 0)
          .input(mapper)
          .build();

     FileDataSink out = new FileDataSink(new CsvOutputFormat(), output, reducer);
     CsvOutputFormat.configureRecordFormat(out);

     Plan plan = new Plan(out, "WordCount Example");
     plan.setDefaultParallelism(numSubTasks);
     return plan;
}
```

```java
public Plan getPlan(String... args) {
String dataInput = args[1];
     String output     = args[2];

     FileDataSource source = new FileDataSource(new TextInputFormat(), dataInput);

     MapOperator mapper = MapOperator.builder(new TokenizeLine())
          .input(source)
          .build();

     ReduceOperator reducer = ReduceOperator.builder(Words.class, String.class, 0)
          .input(mapper)
          .build();

     FileDataSink out = new FileDataSink(new CsvOutputFormat(), output, reducer);
     CsvOutputFormat.configureRecordFormat(out);

     Plan plan = new Plan(out, "WordCount Example");
     plan.setDefaultParallelism(numSubTasks);
     return plan;
}
```

Operator Abstraction

- Not the right one for a declarative API
- Focuses on *how* the dataflow graph is constructed
- Distracts the user from the program logic
- Hard to follow, long programs

Instead, we need an abstraction that *describes the results*

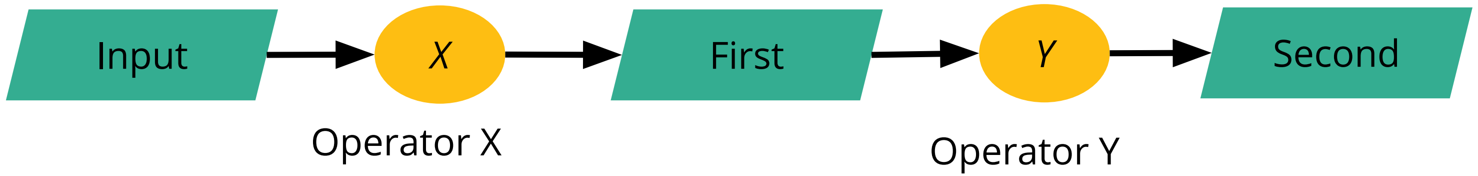
DataSet Abstraction

Think of it as a collection of data elements that can be produced/recovered in several ways:

> … like a Java collection
>
> … like an RDD
>
> … perhaps it is never fully materialized (because the program does not need it to)
>
> … implicitly updated in an iteration

→ this should be *transparent* to the user

Series of Transformations



```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
DataSet<String> input = env.readTextFile(input);

DataSet<String> first = input.filter (str -> str.contains(“Apache Flink“));
DataSet<String> second = first.filter (str -> str.length() > 40);

second.print()
env.execute();
```

Declarative Word Count

Java

```
DataSet<String> text = env.readTextFile(input);
DataSet<Tuple2<String, Integer>> result = text
         .flatMap(new Tokenizer())
         .groupBy(0)
         .sum(1);
```

Scala

```
val input = env.readTextFile(input);
val words = input flatMap { line => line.split("\\W+")}
                  map { word => (word, 1)}
val counts = words groupBy(0) sum(1)
```

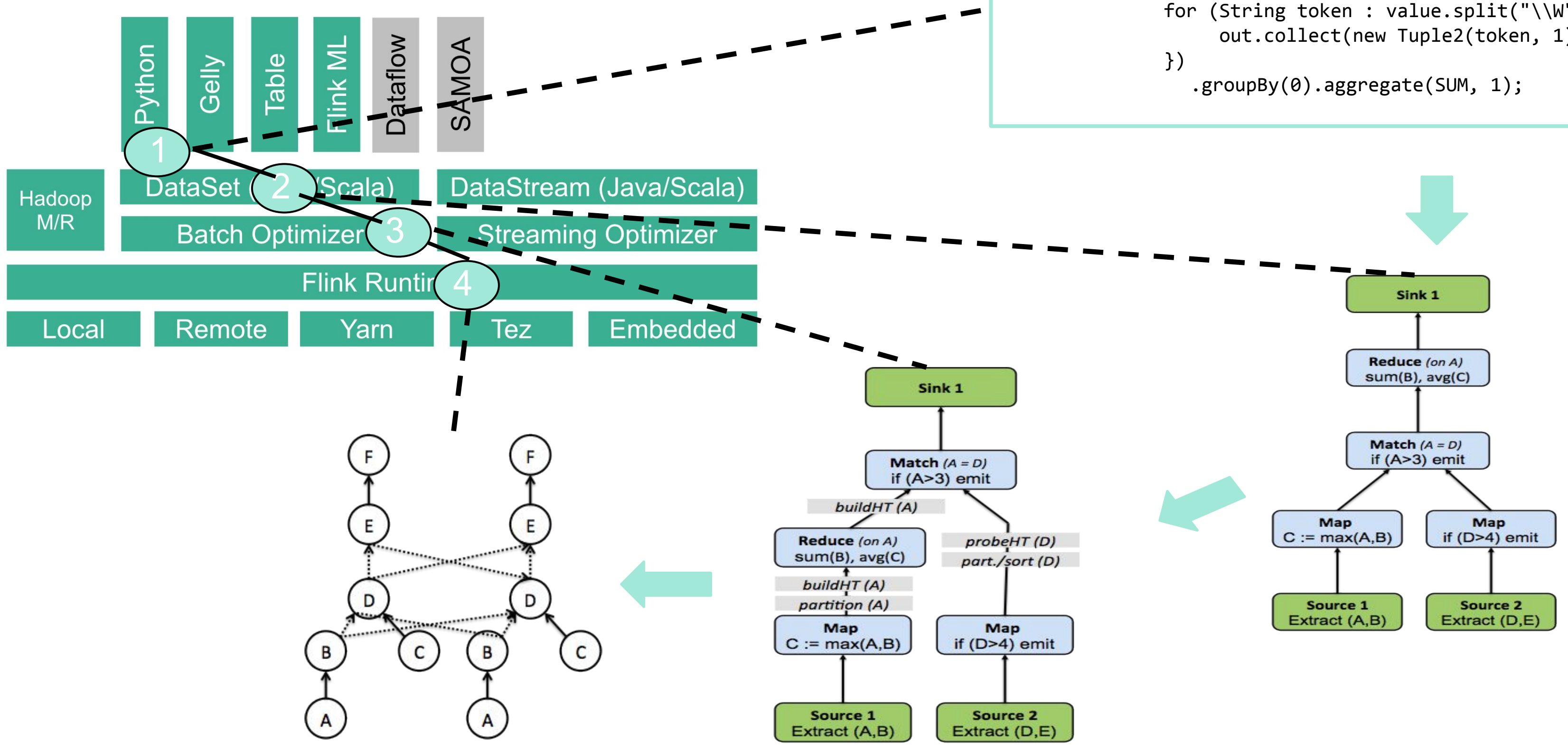
Runtime Execution

What happens when you submit a Flink program?

Program Life-Cycle

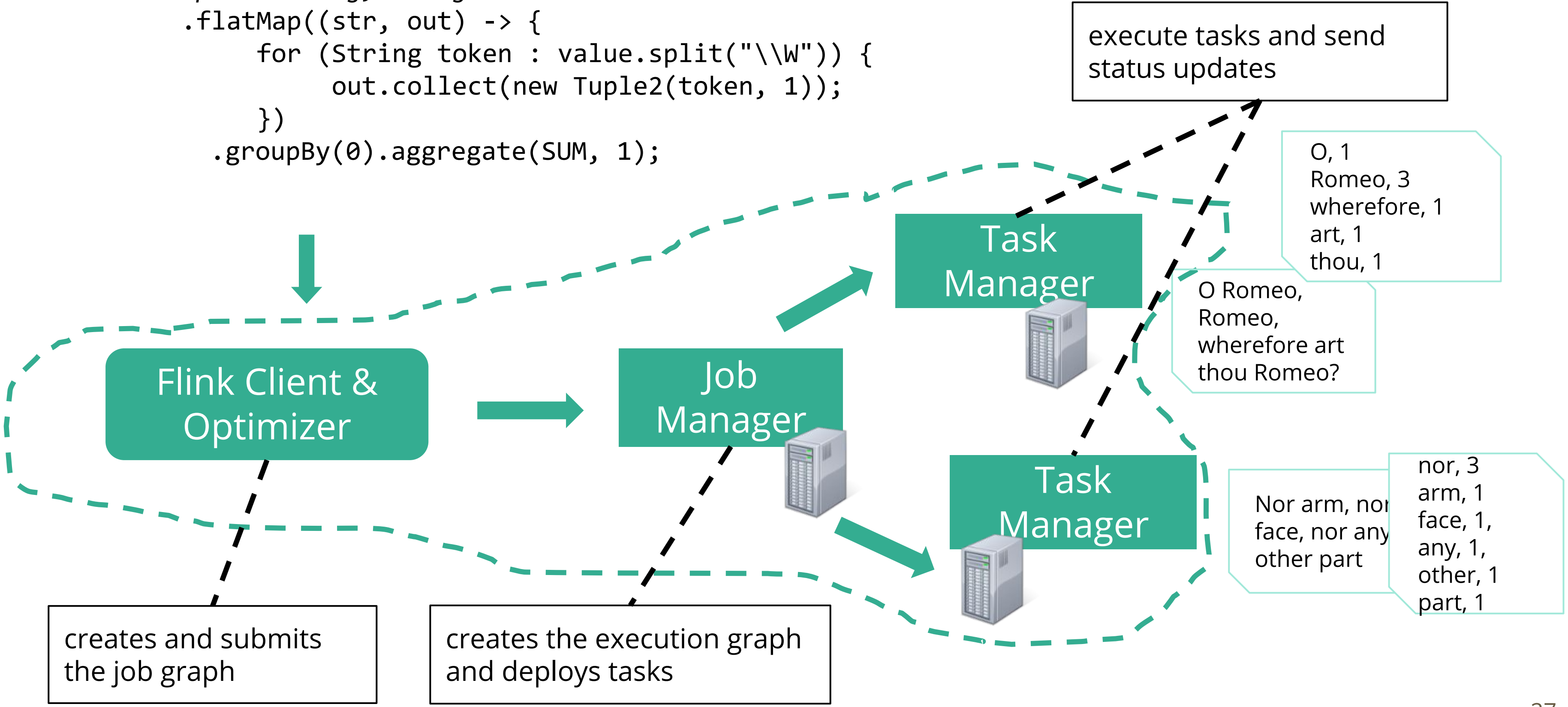
```
DataSet<String> text = env.readTextFile(input);

DataSet<Tuple2<String, Integer>> result = text
         .flatMap((str, out) -> {
              for (String token : value.split("\\W")) {
                   out.collect(new Tuple2(token, 1));
              })
               .groupBy(0).aggregate(SUM, 1);
```



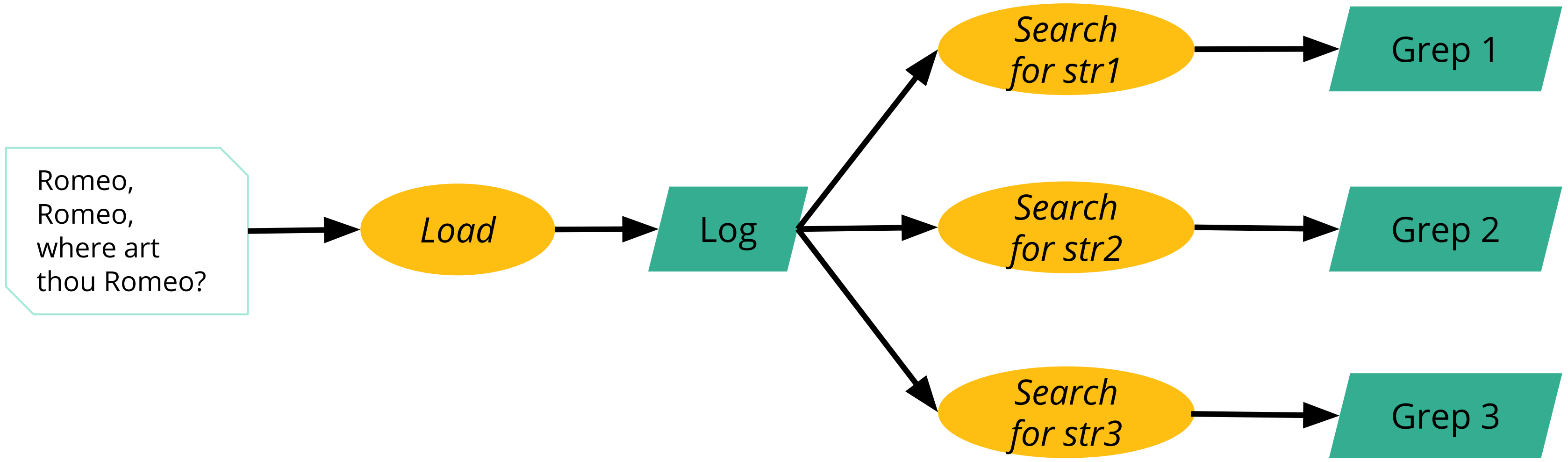
```
DataSet<String> text = env.readTextFile(input);

DataSet<Tuple2<String, Integer>> result = text
         .flatMap((str, out) -> {
              for (String token : value.split("\\W")) {
                   out.collect(new Tuple2(token, 1));
              })
           .groupBy(0).aggregate(SUM, 1);
```

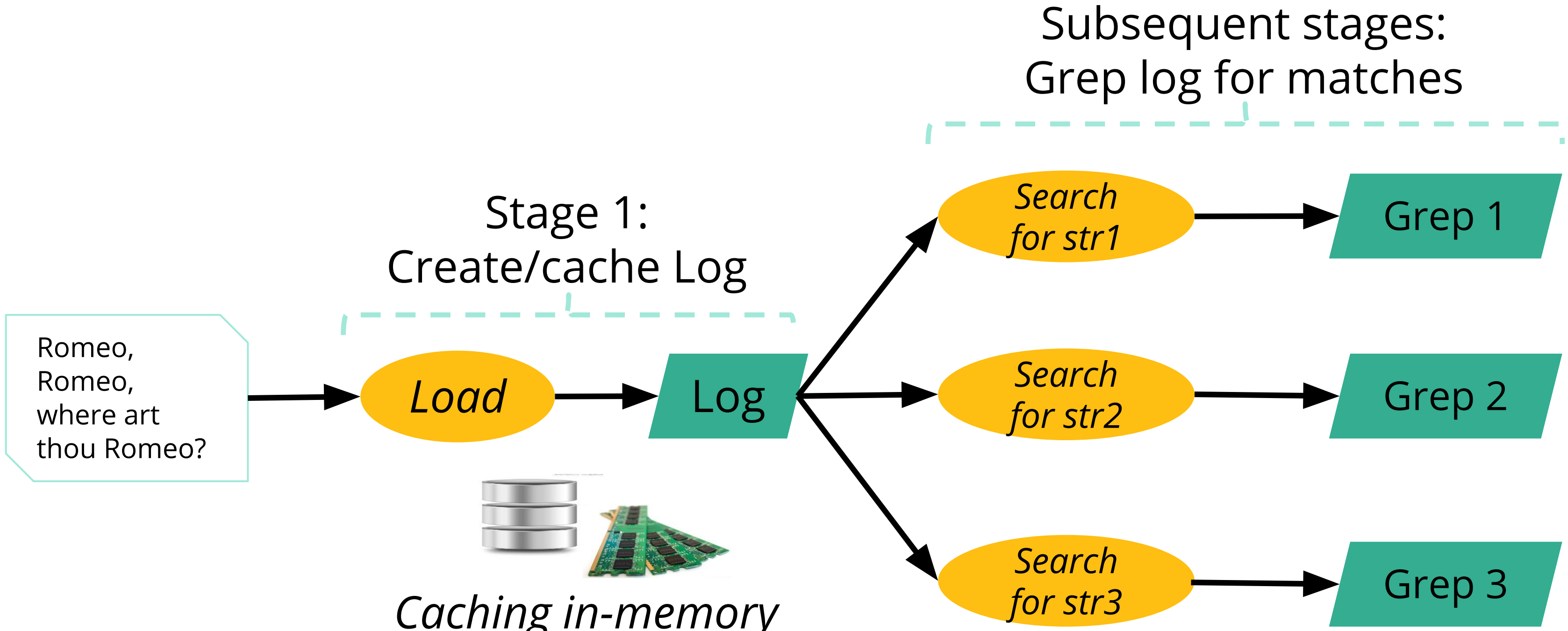


execute tasks and send status updates
O, 1
Romeo, 3
wherefore, 1
art, 1
thou, 1
Task Manager
O Romeo, Romeo, wherefore art thou Romeo?
Flink Client & Optimizer
Job Manager
Task Manager
Nor arm, nor face, nor any other part
nor, 3
arm, 1
face, 1,
any, 1,
other, 1
part, 1
creates and submits the job graph
creates the execution graph and deploys tasks

Example: grep



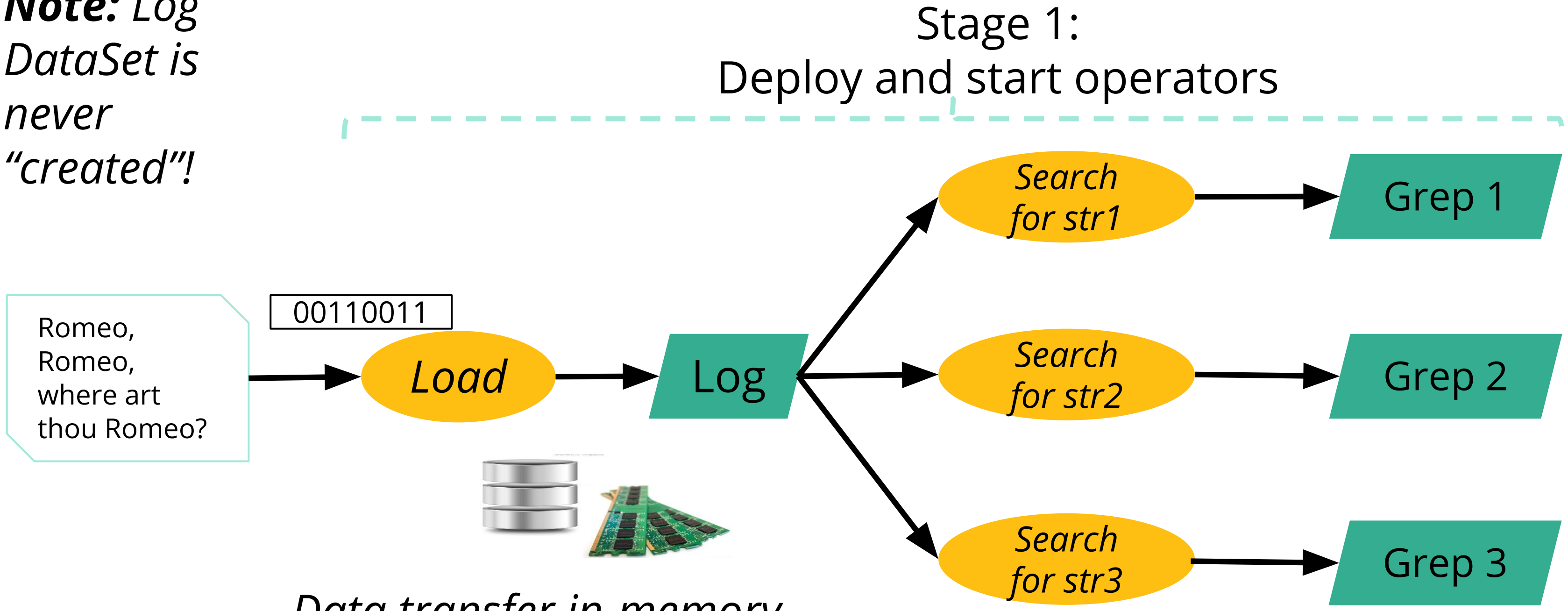
Staged (batch) execution



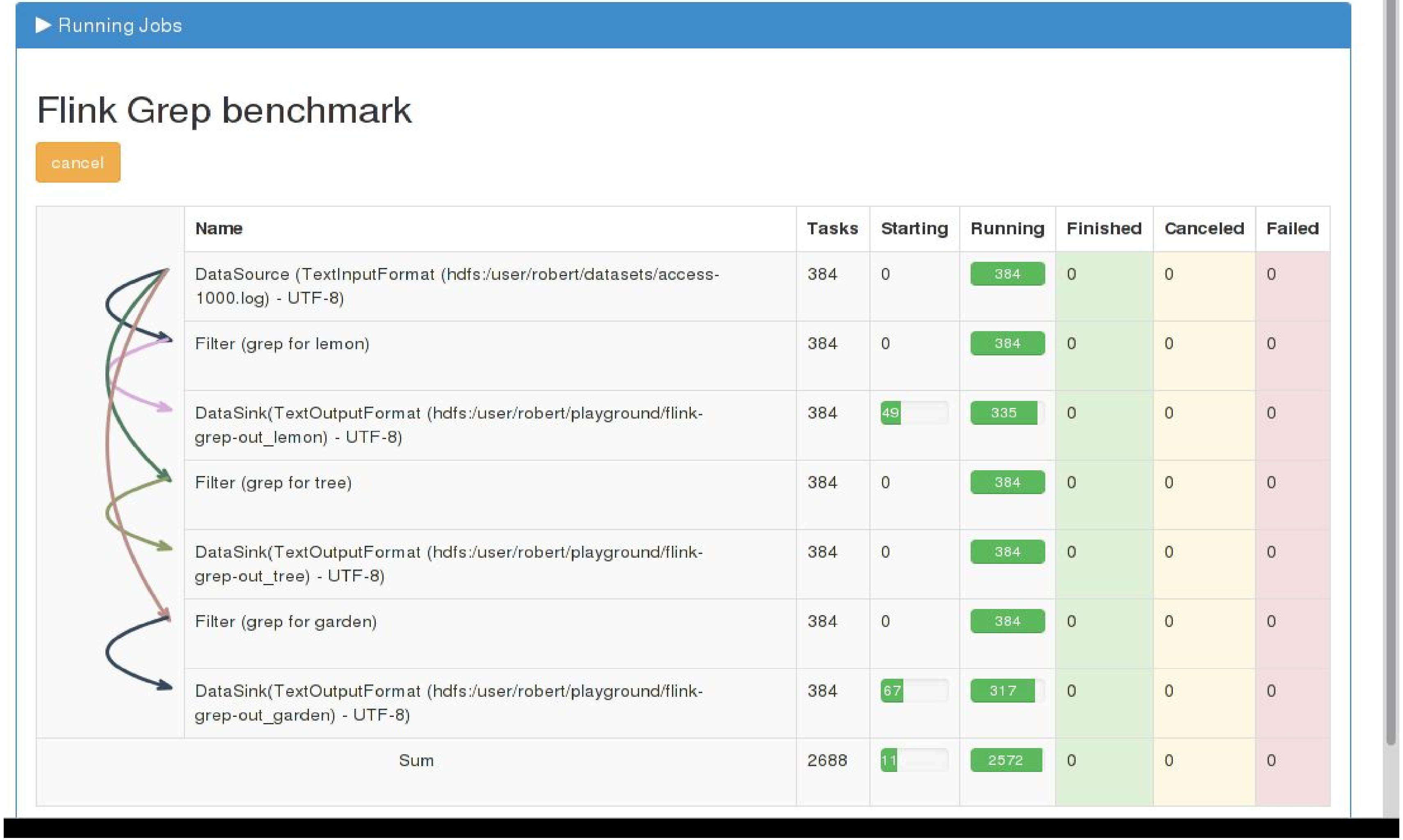
Caching in-memory and disk if needed

Pipelined execution

Note: *Log DataSet is never "created"!*



Data transfer in-memory and disk if needed



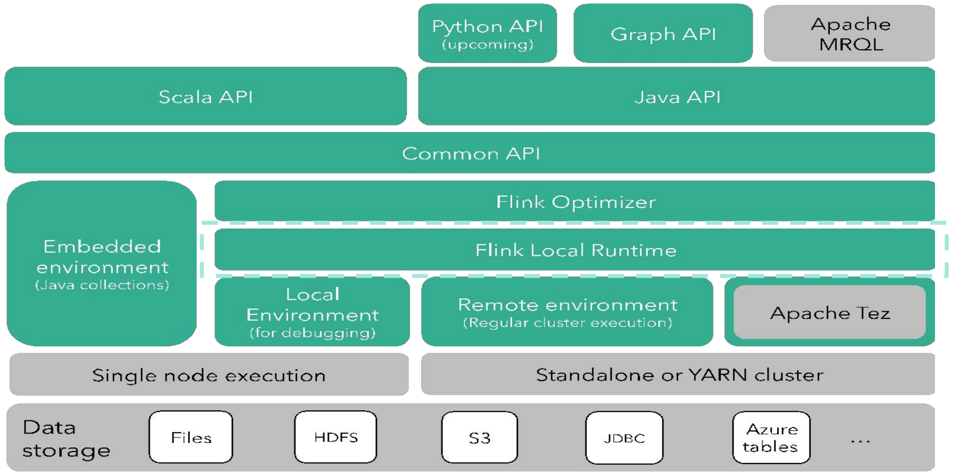
▶ Running Jobs

Flink Grep benchmark

cancel

Name	Tasks	Starting	Running	Finished	Canceled	Failed
DataSource (TextInputFormat (hdfs:/user/robert/datasets/access-1000.log) - UTF-8)	384	0	384	0	0	0
Filter (grep for lemon)	384	0	384	0	0	0
DataSink(TextOutputFormat (hdfs:/user/robert/playground/flink-grep-out_lemon) - UTF-8)	384	49	335	0	0	0
Filter (grep for tree)	384	0	384	0	0	0
DataSink(TextOutputFormat (hdfs:/user/robert/playground/flink-grep-out_tree) - UTF-8)	384	0	384	0	0	0
Filter (grep for garden)	384	0	384	0	0	0
DataSink(TextOutputFormat (hdfs:/user/robert/playground/flink-grep-out_garden) - UTF-8)	384	67	317	0	0	0
Sum	2688	11	2572	0	0	0

Flink Local Runtime

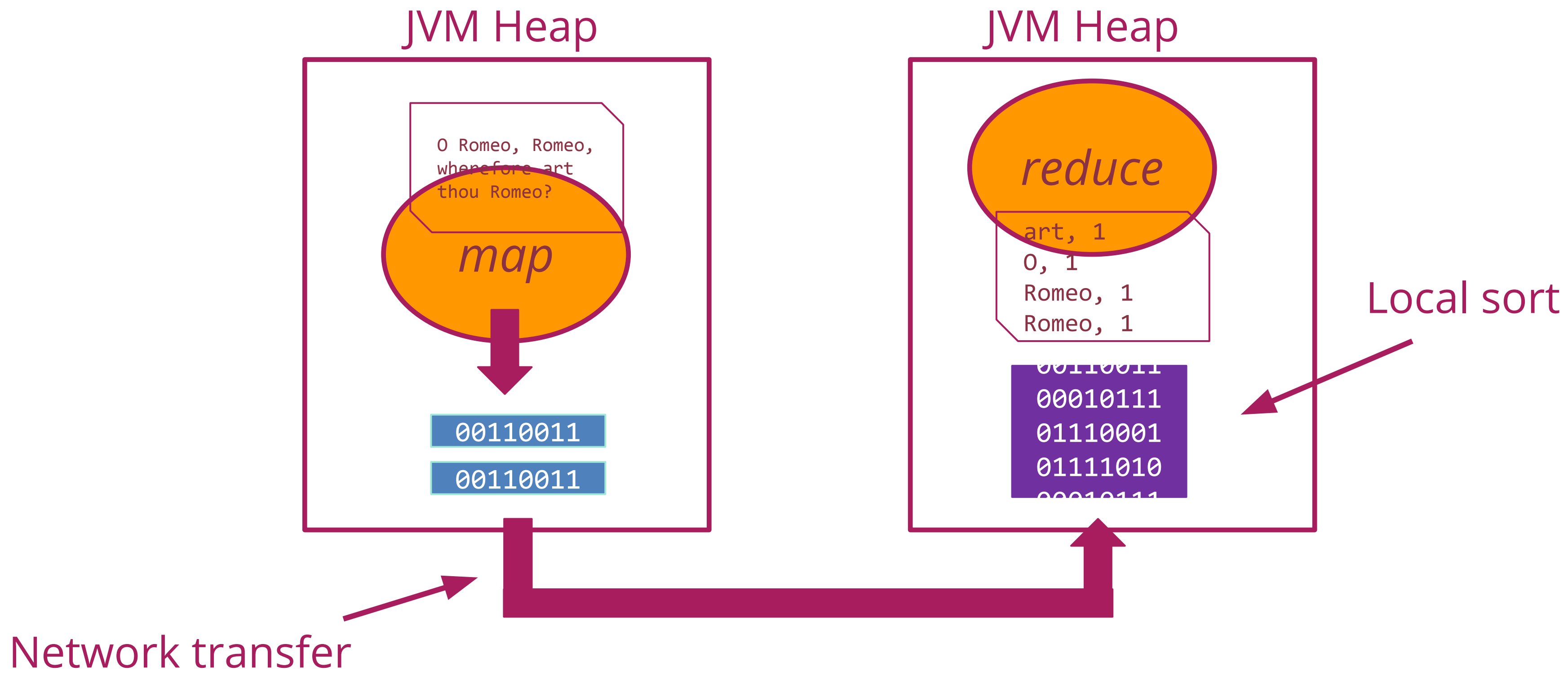


- *Local* runtime, not the distributed execution engine
- Aka: what happens inside every parallel task

Runtime Operators

- Sorting and hashing data
  - Necessary for grouping, aggregation, reduce, join, cogroup, delta iterations
- Flink contains tailored implementations of hybrid hashing and external sorting in Java
  - Scale well with both abundant and restricted memory sizes

Internal Data Representation



Internal Data Representation

Java objects

- Easier to program
- Can suffer from GC overhead
- Hard to de-stage data to disk, may suffer from “out of memory exceptions”

Raw bytes

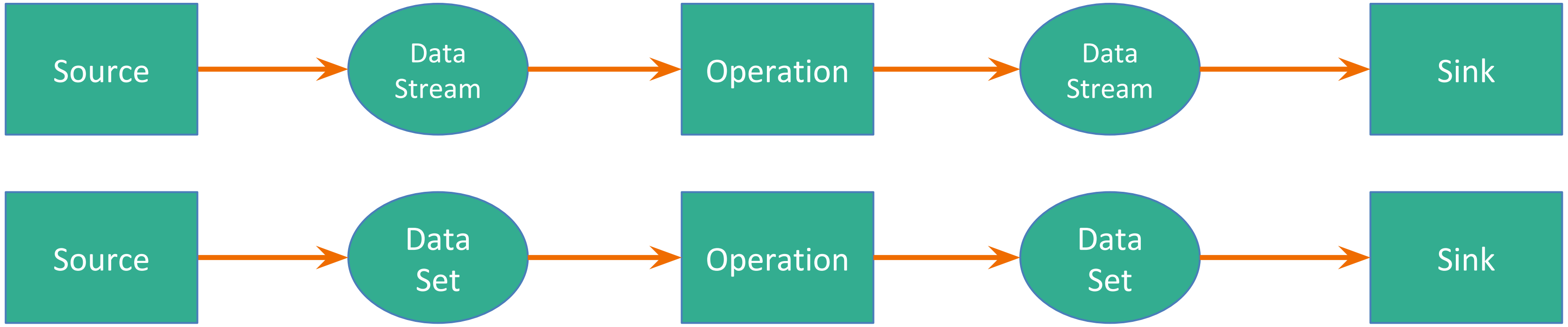
- Harder to program (customer serialization stack, more involved runtime operators)
- Solves most of memory and GC problems
- Overhead from object (de) serialization

Flink follows the raw byte approach

Distributed Execution

- Pipelined
  - Same engine for Flink and Flink streaming
- Pluggable
  - Local runtime can be executed on other engines
  - E.g., Java collections and Apache Tez
- Coordination built in Akka

Basic API Concept

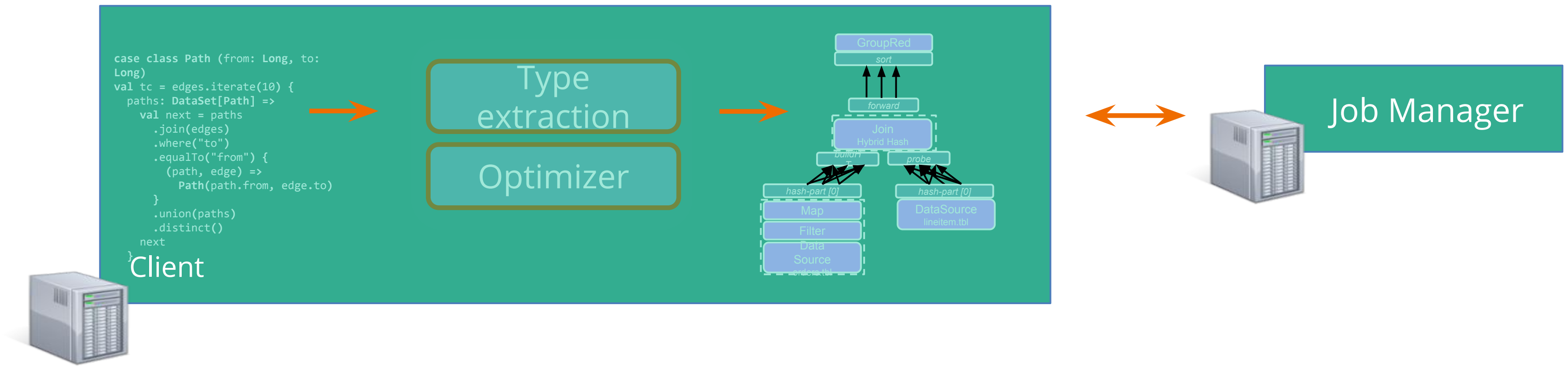


How do I write a Flink program?

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

Client

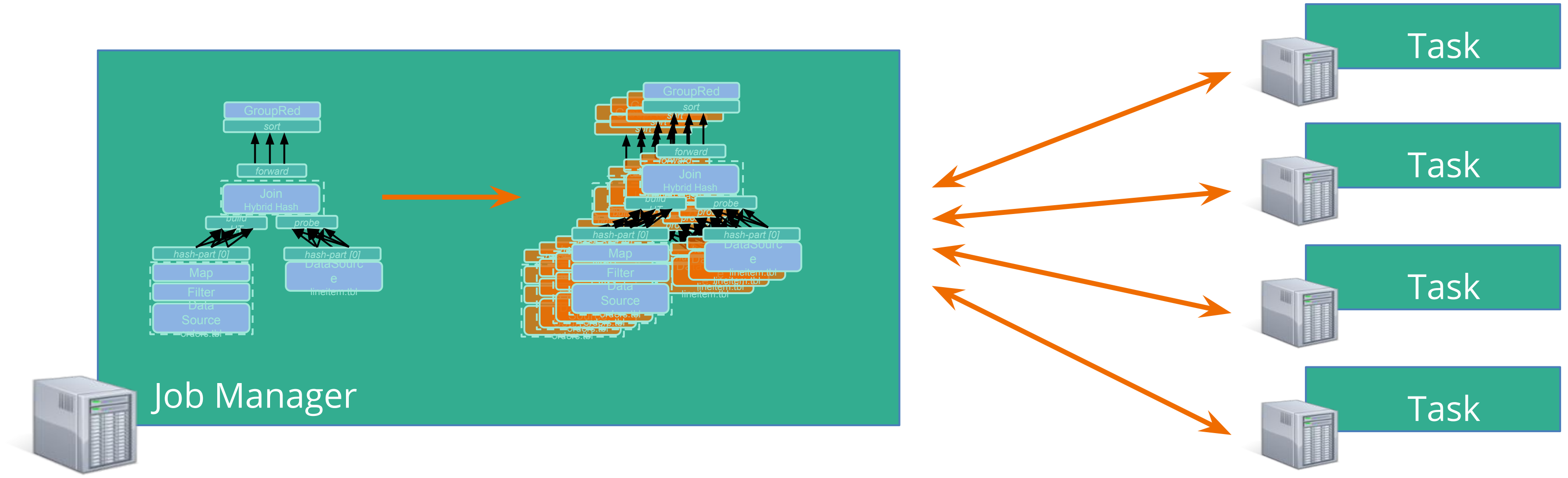
- Optimize
- Construct job graph
- Pass job graph to job manager
- Retrieve job results



```
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
  }
```

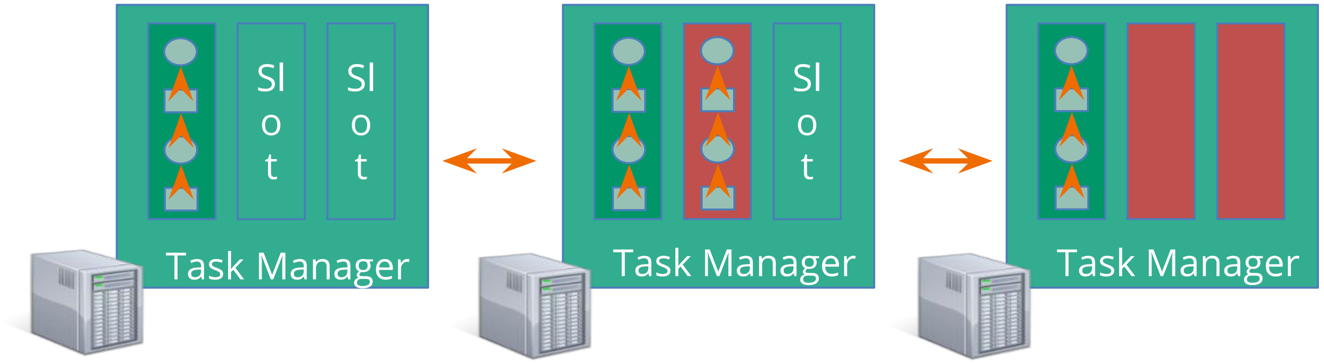
Job Manager

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



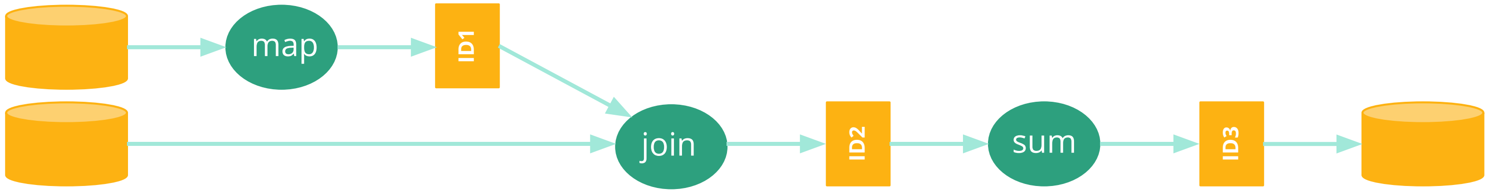
Task Manager

- 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

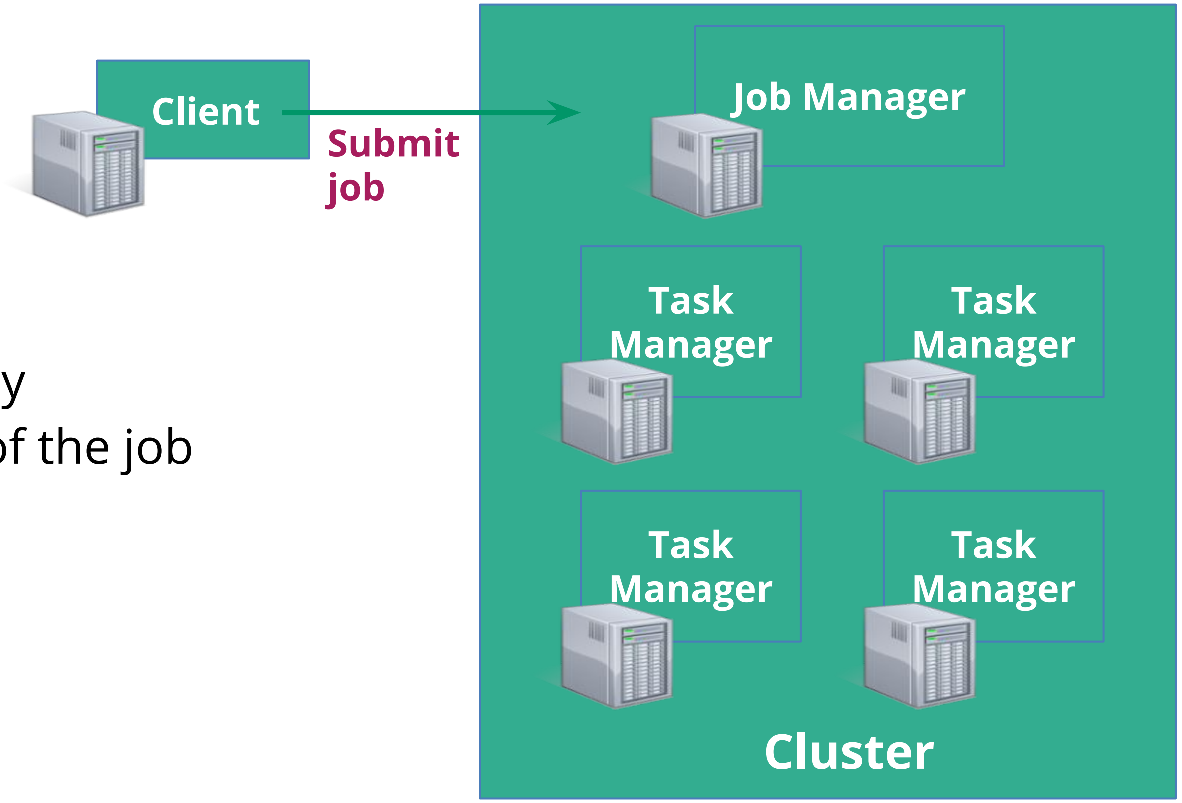


Execution Model

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



Remote Execution



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

Data Exchange

Flink’s Network Stack

Data Exchange

Design principles:

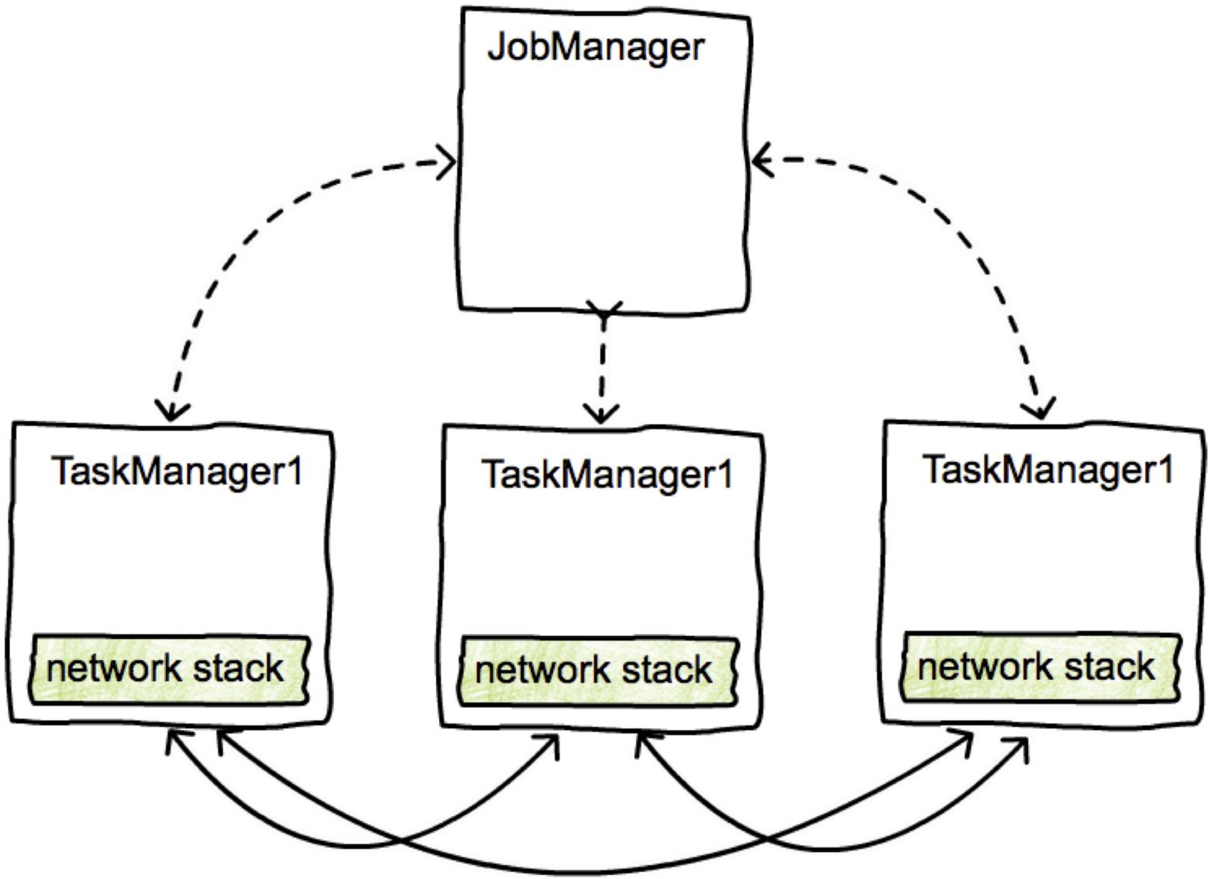
1. The control flow for data exchange (i.e., the message passing in order to initiate the exchange) is **receiver-initiated**, much like the original MapReduce.
2. The data flow for data exchange, i.e., the actual transfer of data over the wire is abstracted by the notion of an **IntermediateResult**, and is pluggable. This means that the system can support both *streaming* data transfer and *batch* data transfer with the same implementation.

Data Exchange Components

JobManager: responsible for scheduling tasks, recovery, and coordination, and holds the big picture of a job via the ExecutionGraph data structure.

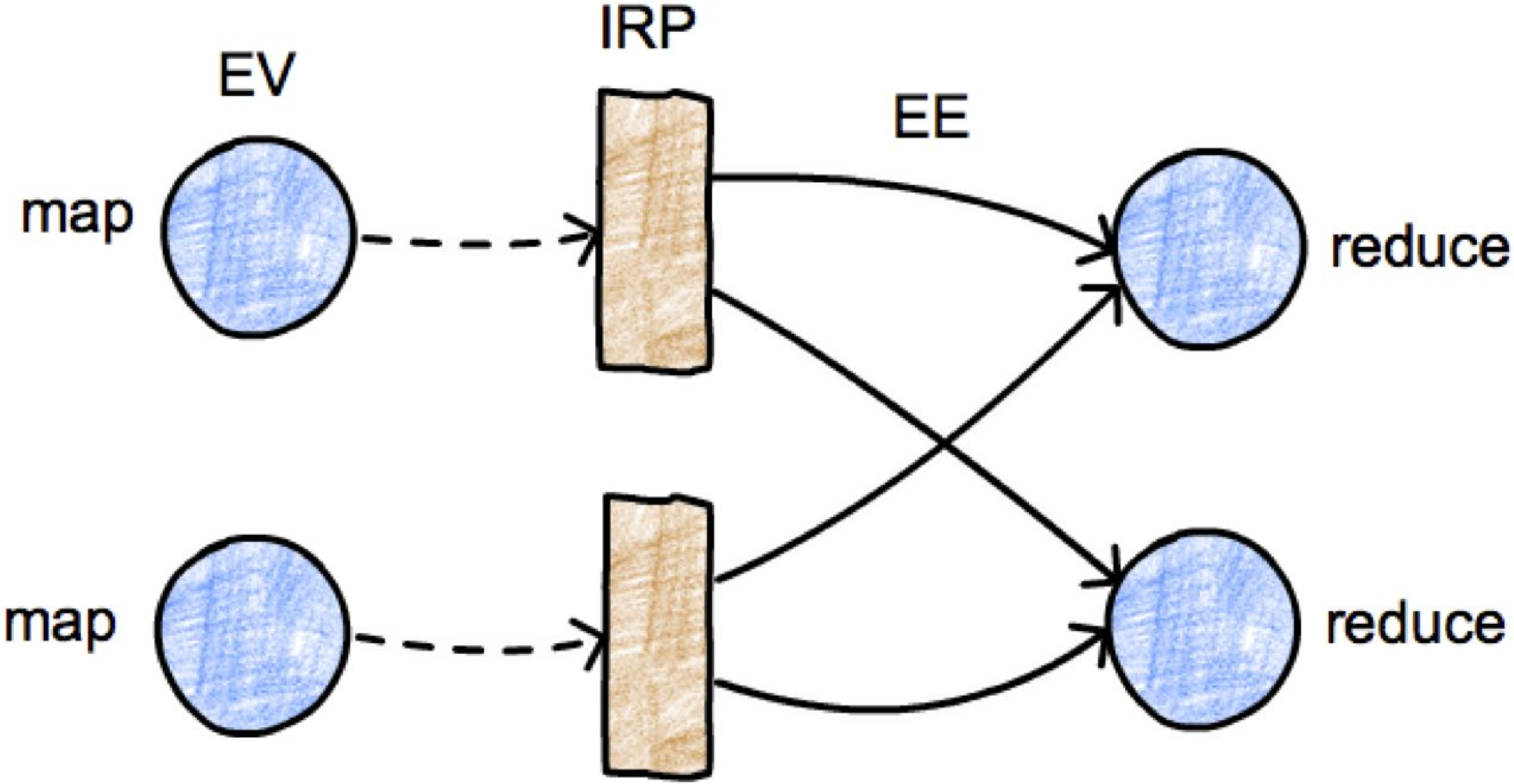
TaskManagers: executes many tasks concurrently in threads. Each TM also contains one **CommunicationManager** (CM - shared between tasks), and one **MemoryManager** (MM - also shared between tasks). TMs can exchange data with each other via standing TCP connections, which are created when needed.

Note that in Flink, it is TaskManagers, not tasks, that exchange data over the network, i.e., *data exchange between tasks that live in the same TM is multiplexed over one network connection*.


JobManager
TaskManager1
TaskManager1
TaskManager1
network stack
network stack
network stack

Execution Graph

- The execution graph is a data structure that contains the “ground truth” about the job computation.
- It consists of vertices (**ExecutionVertex**) that represent computation tasks, and intermediate results (**IntermediateResultPartition**), that represent data produced by tasks.
- Vertices are linked to the intermediate results they consume via **ExecutionEdges** (EE).


IRP
EV
EE
map
reduce
map
reduce

Runtime Data Structures (1)

- **ResultPartition** (RP): a chunk of data that a **BufferWriter** writes to, i.e., a chunk of data produced by a single task. A RP is a collection of **Result Subpartitions** (RSs). This is to distinguish between data that is destined to different receivers.
- **ResultSubpartition** (RS): one partition of the data that is created by an operator, together with the logic for forwarding this data to the receiving operator.
  - The implementation of a RS determines the actual data transfer logic
  - Pluggable mechanism that allows the system to support a variety of data transfers.
  - e.g. the **PipelinedSubpartition** supports streaming data exchange. The **SpillableSubpartition** is blocking and supports batch data exchange.

Runtime Data Structures (2)

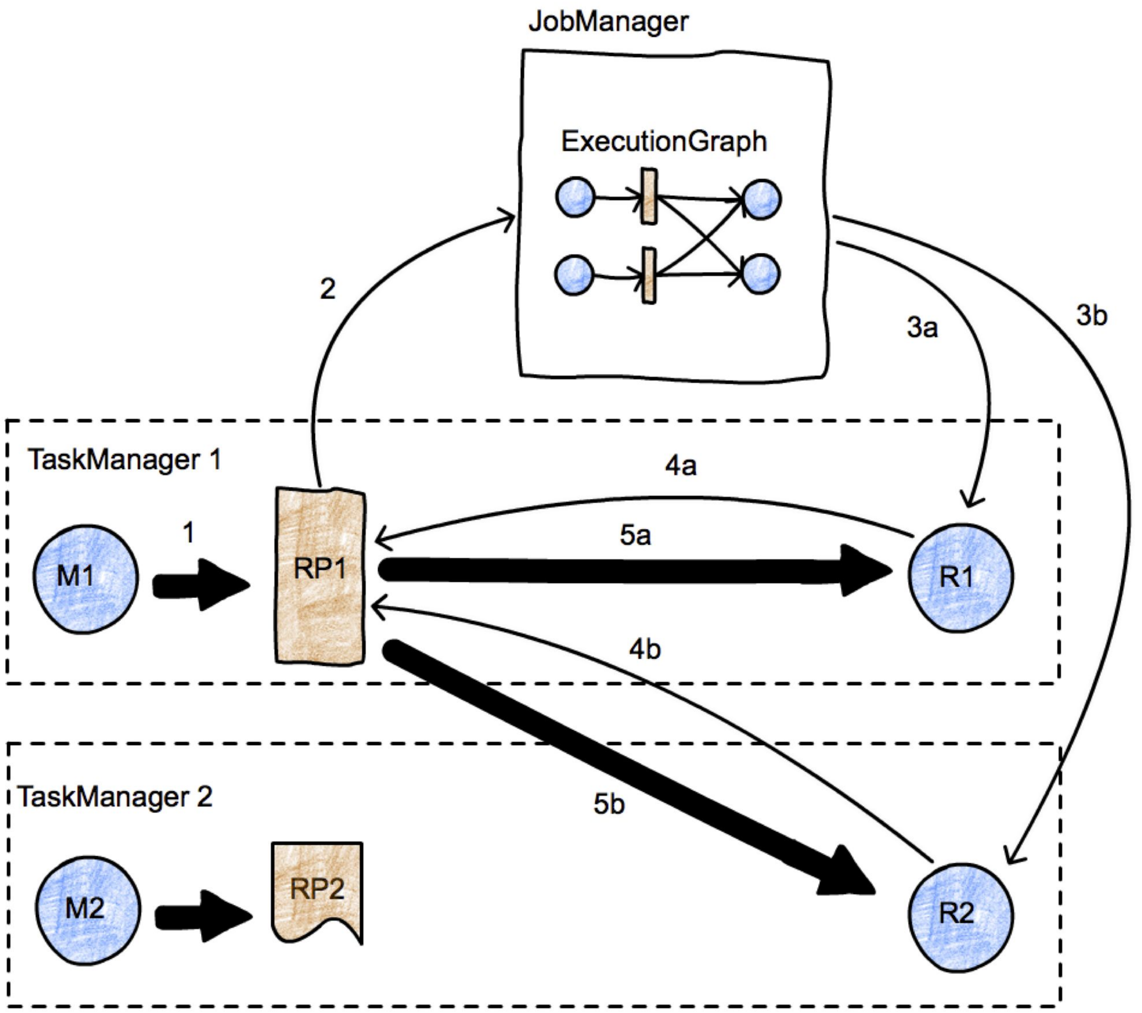
- **InputGate**: The logical equivalent of the RP at the receiving side. It is responsible for collecting buffers of data and handing them upstream.
- **InputChannel**: The logical equivalent of the RS at the receiving side. It is responsible for collecting buffers of data for a specific partition.
- **Buffer**: Used by the network stack to buffer records for network transfer.
- **(De)/Serializers**: reliably convert typed records into raw byte buffers and vice versa, handling records that span multiple buffers, etc.

Control Flow (1)

1. Operators produce a ResultPartitions (RP)
2. When a RP becomes available for consumption, it informs the JobManager.
3. The JobManager notifies the intended receivers of this partition that the partition is ready.
4. If the receivers have not been scheduled yet, this will actually trigger the deployment of the tasks.
5. The receivers request data from the RPs. This initiates the data transfer between the tasks, either locally, or passing through the network stack of the TaskManagers.

Control Flow (2)

- If a RP fully produces itself (and is perhaps written to a file) before informing the JM, the data exchange corresponds to a **batch** exchange.
- If the RP1 informs the JM as soon as its first record is produced, we have a **streaming** data exchange.


JobManager
ExecutionGraph
2
3a
3b
TaskManager 1
4a
1
5a
M1
RP1
R1
4b
TaskManager 2
5b
M2
RP2
R2

Buffer Transfer (1)

- Records produced by an Operator are passed to a **RecordWriter** object.
- RecordWriters contain serializers (**RecordSerializer** objects), one per consumer task that will possibly consume these records.
    - For example, in a shuffle or broadcast, there will be as many serializers as the number of consumer tasks.
- A **ChannelSelector** selects one or more serializers to place the record to.
    - For example, if records are broadcast, they will be placed in every serializer. If records are hash-partitioned, the ChannelSelector will evaluate the hash value on the record and select the appropriate serializer.

Buffer Transfer (2)

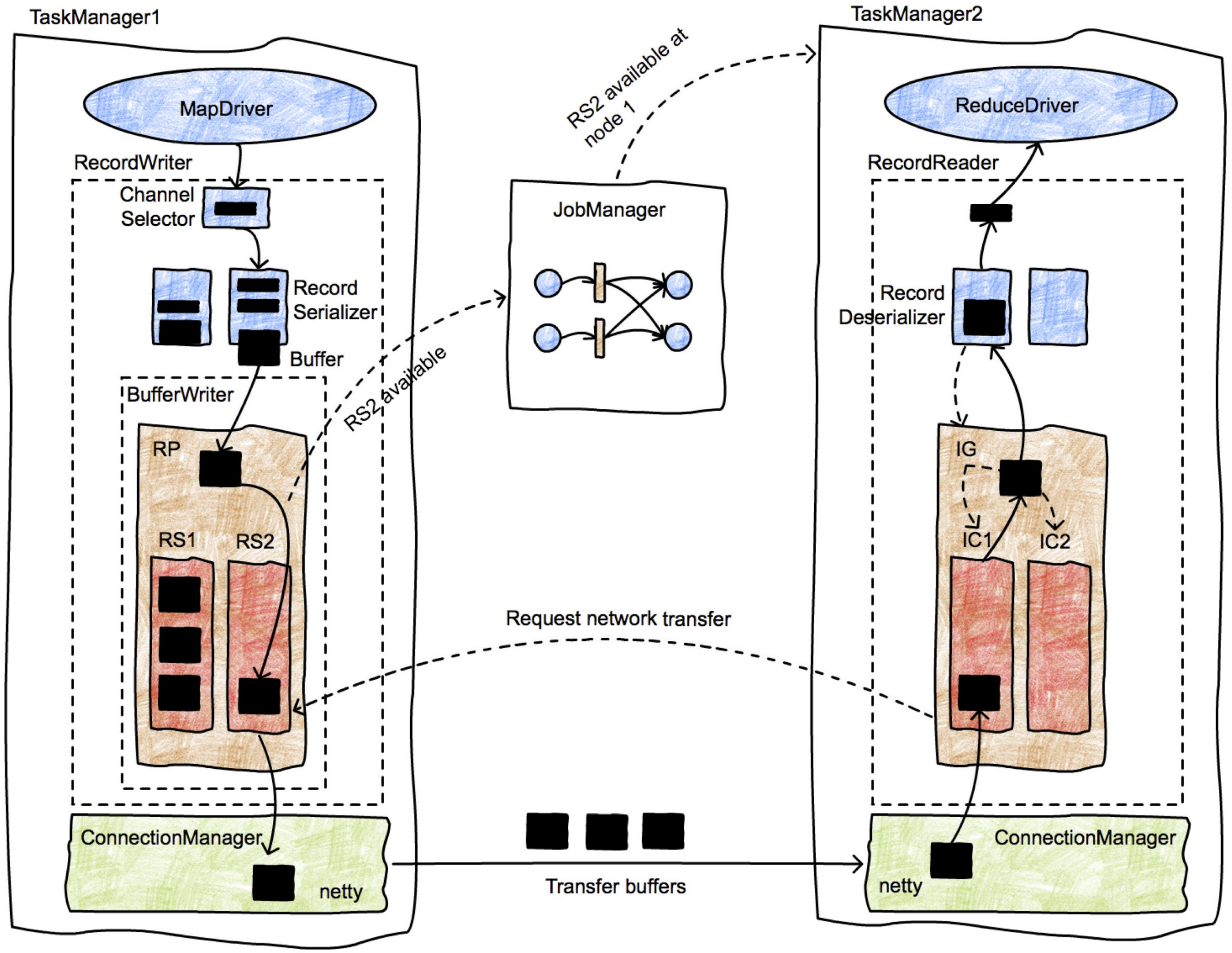
- The serializers serialize the records into their binary representation, and place them in *fixed-size* buffers (records can span multiple buffers).
- These buffers and handed over to a **BufferWriter** and written out to an ResultPartition (RP). The RP consists of several subpartitions (ResultSubpartitions - RSs) that collect buffers for specific consumers.

Buffer Transfer (3)

- The JobManager looks up the consumers of ResultPartitions and notifies the TaskManagers that a chunk of data is available.
- Messages to TMs are propagated down to the **InputChannel** that is supposed to receive this buffer, which in turn notifies ResultPartitions that a network transfer can be initiated.
- ResultPartitions hand over the buffer to the network stack of TMs, which in turn hand it over to **netty** for shipping.
- Network connections are long-running and exist between TaskManagers, not individual tasks.

Buffer Transfer (4)

- Once a buffer is received by a TM, it passes through a similar object hierarchy and finally ends up in a **RecordDeserializer** that produces typed records from buffers and hands them over to the receiving task.


TaskManager1
MapDriver
RecordWriter
Channel Selector
Record Serializer
Buffer
BufferWriter
RP
RS1
RS2
ConnectionManager
netty
RS2 available
JobManager
RS2 available at node 1
Request network transfer
Transfer buffers
TaskManager2
ReduceDriver
RecordReader
Record Deserializer
IG
IC1
IC2
ConnectionManager
netty

Memory Management

Performance in the Java world

Big data processing systems which use the JVM have to deal with several challenges:

- object storage overhead
- garbage collection
- `outOfMemoryErrors`

Solution: Take control of memory!

- Allocate large memory chunks and manage them manually
    - avoid heap fragmentation
    - better memory usage

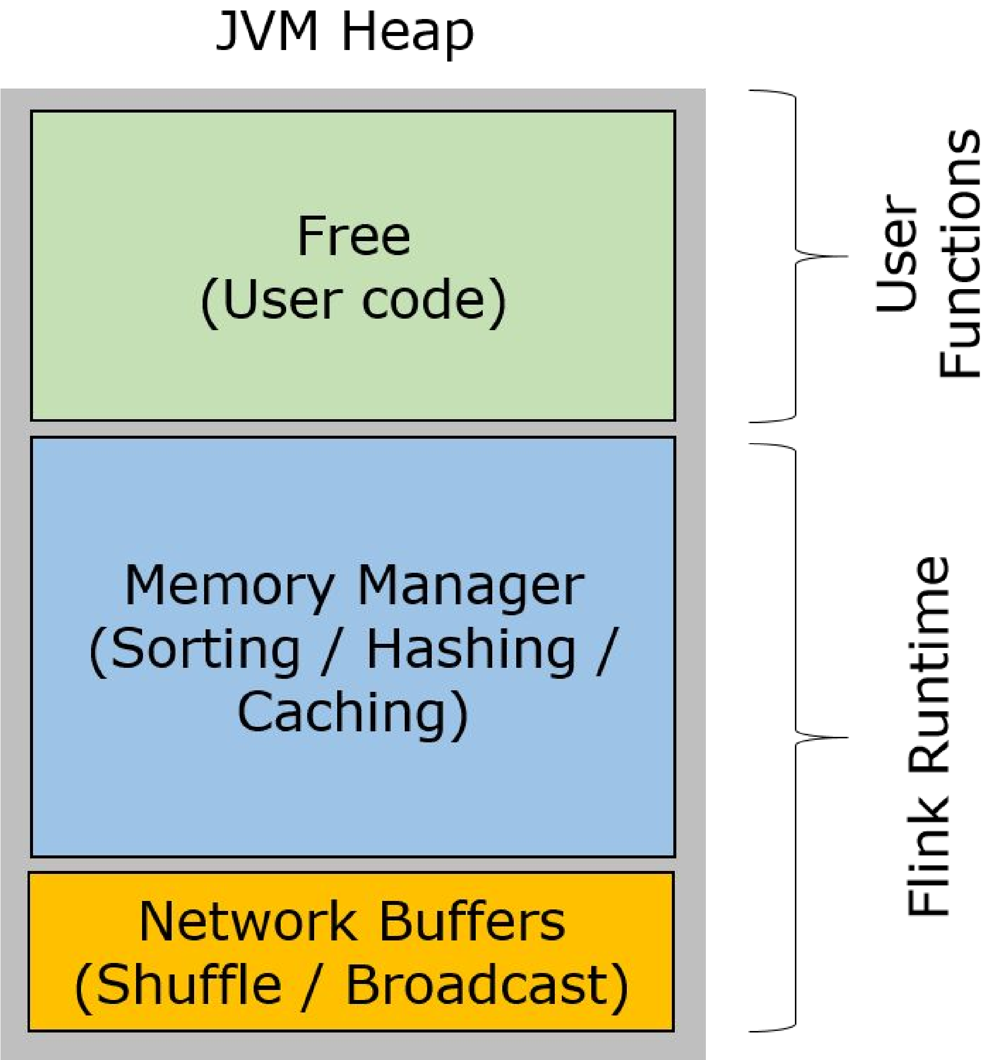
Memory management in Flink

Divide the Java heap into 3 regions:

1. **Network Buffers:** 32KB buffers for buffering records at startup
2. **Memory Manager Pool:** 32KB managed buffers offered as a pool to internal algorithms (Sort, Shuffle, Join)

$\Big\}$ ~70%

3. **Memory for User Code:** for short-lived instances of user code



Both Network buffers and the Memory Manager segments live throughout the entire life of a TaskManager

- Buffer sizes are configurable

Benefits of Memory Management

- Short-lived user code instances are quickly collected
- Internal algorithms allocate and release blocks of 32KB of memory, but the MemorySegment’s that these represent are never are never garbage-collected
- Algorithms persist to disk if their allocated segments are exceeded