Data Analysis with Stratosphere

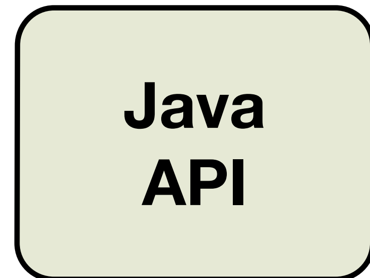
Ufuk Celebi
u.celebi@fu-berlin.de
@iam**uce**

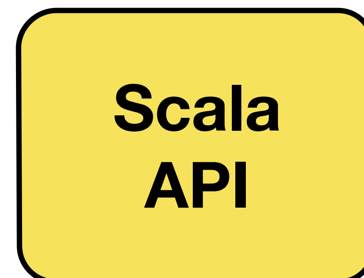
December 12th 2013

Future Cloud Action Line Workshop
EIT ICT Labs Helsinki

Overview

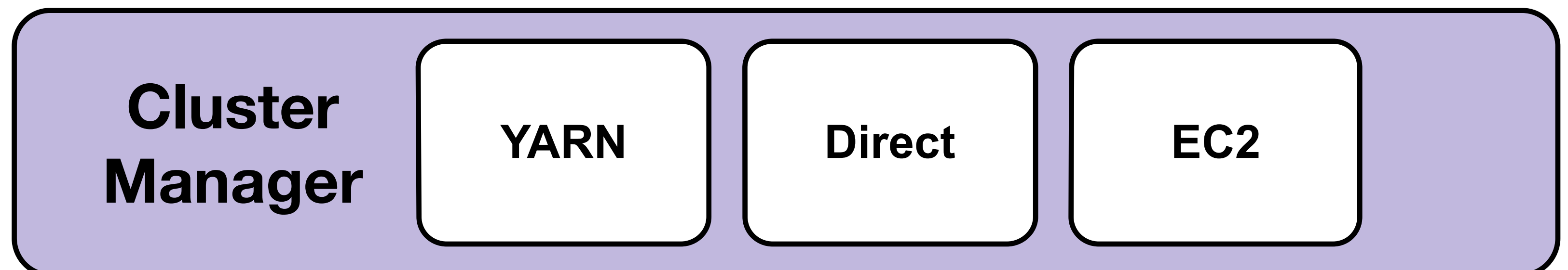
User writes *program*

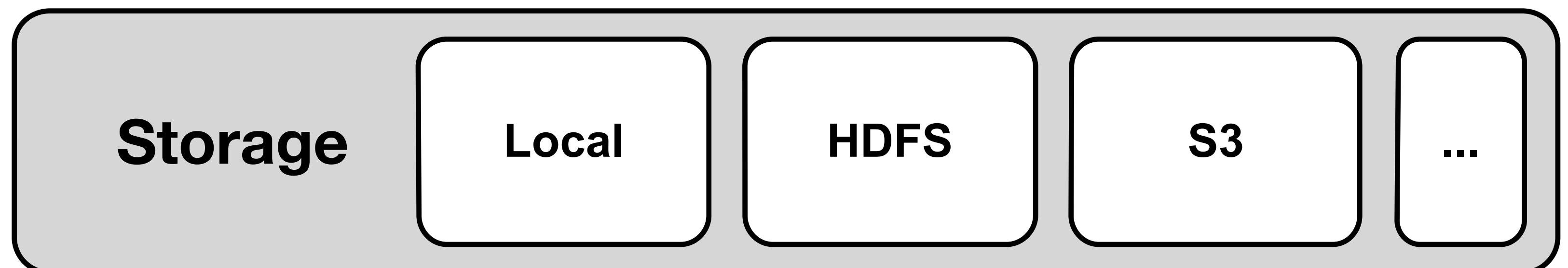




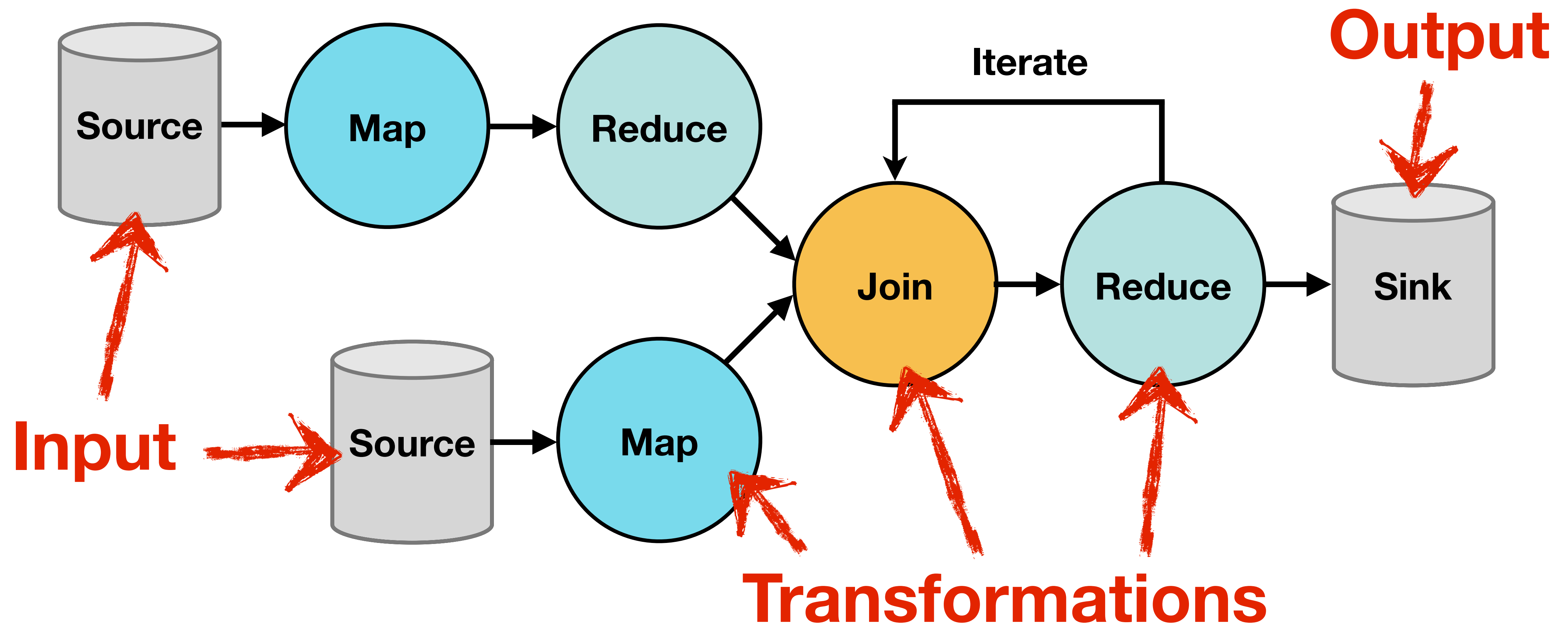
Focus today

Stratosphere *distributes*, *parallelizes* and optimizes execution



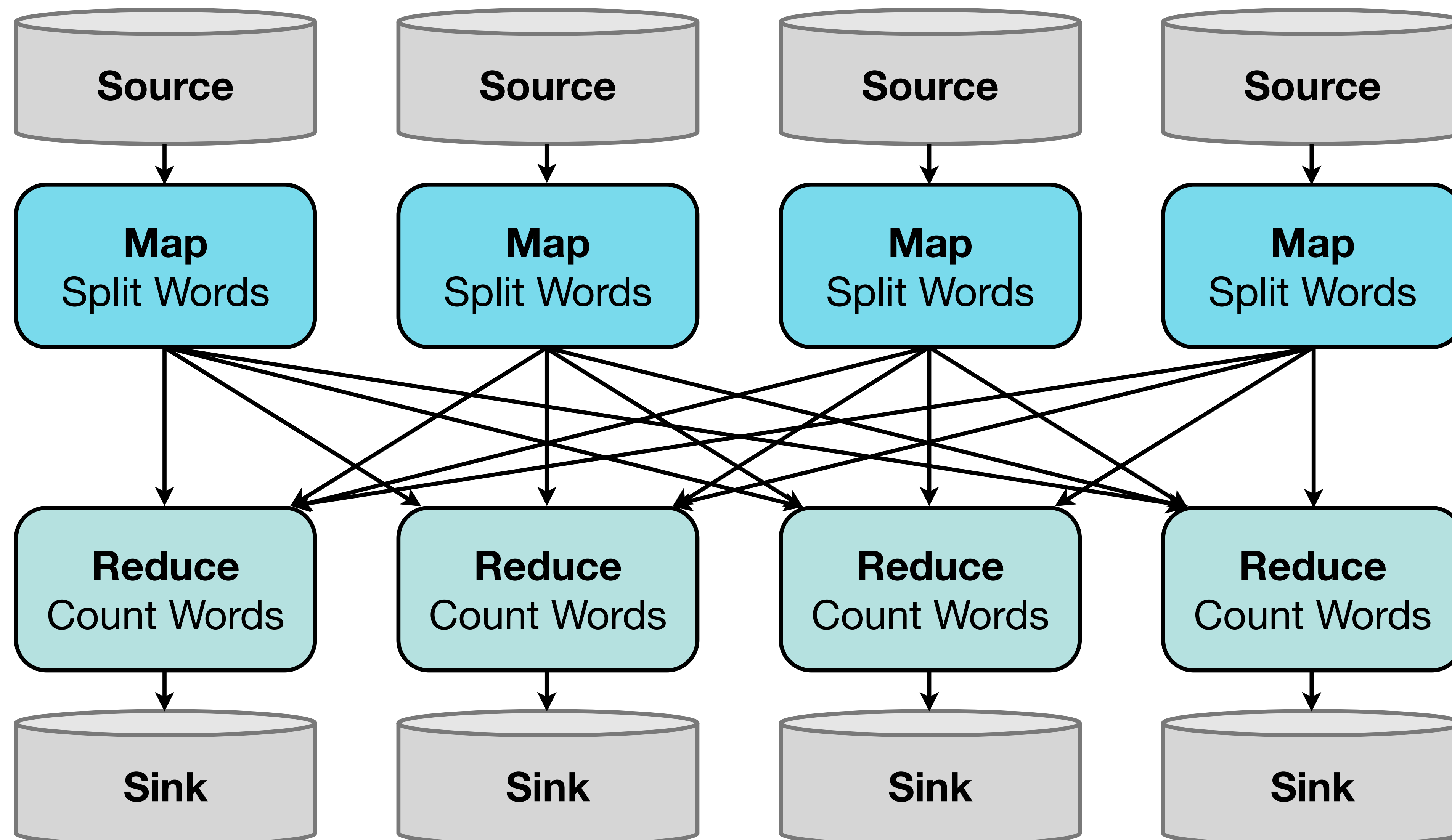


Data Flows



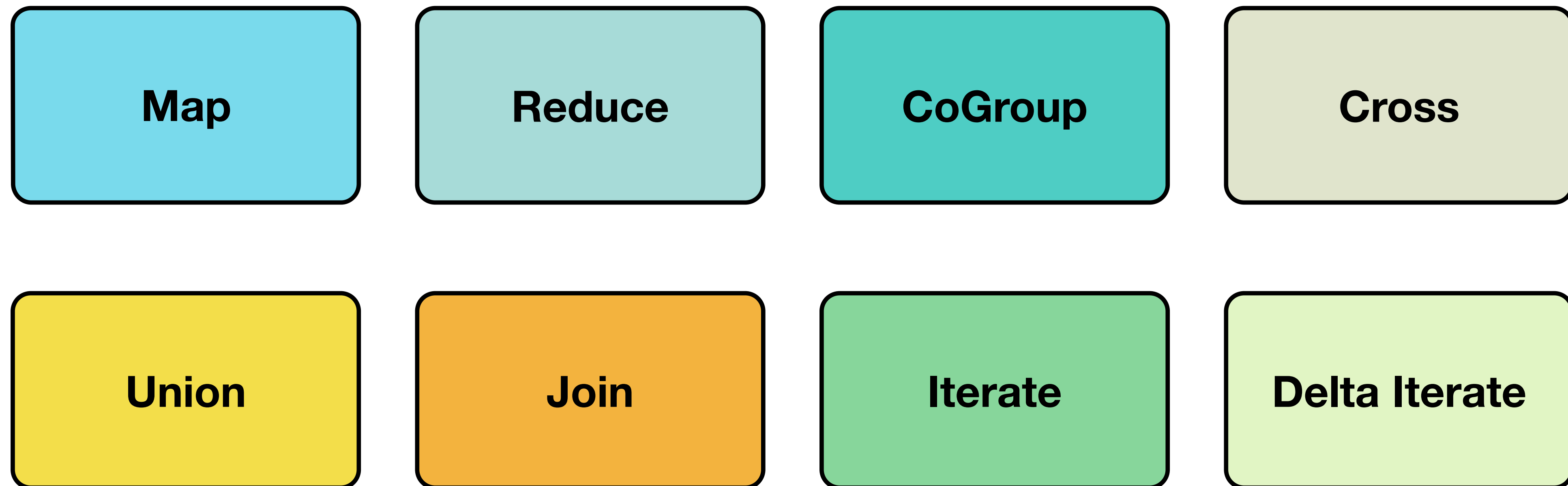
Programs are expressed as **data flows** from **sources** to **sinks**.

Data Flows at Runtime

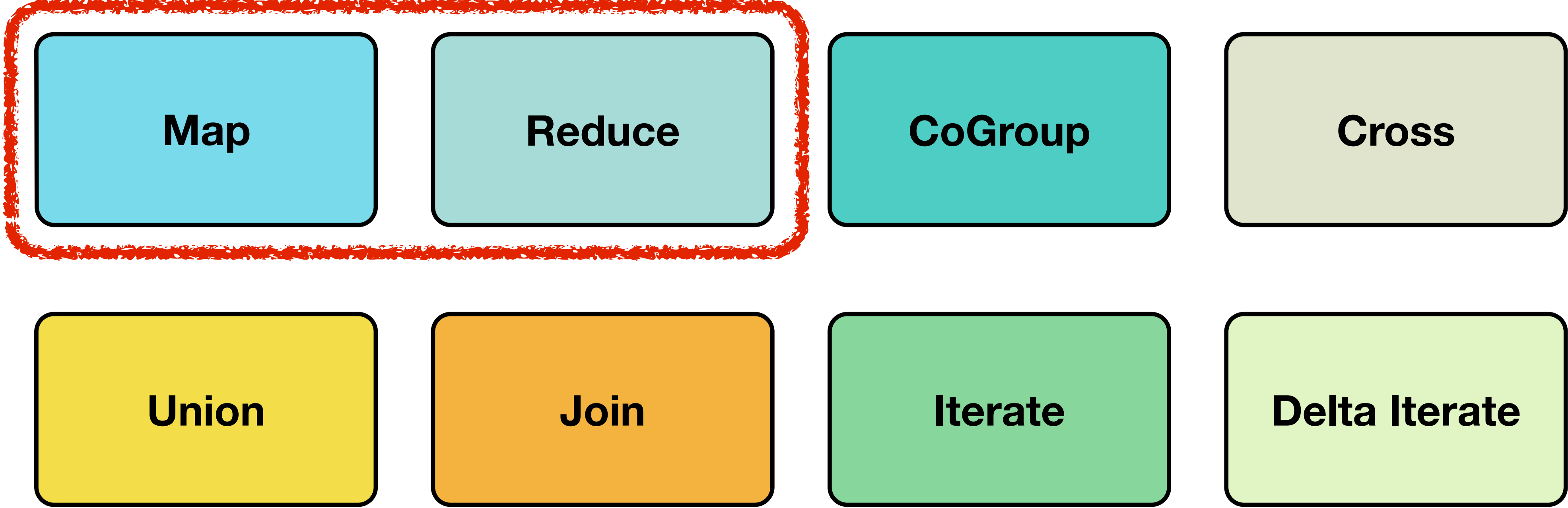


Stratosphere distributes, parallelizes, and optimizes **execution**.

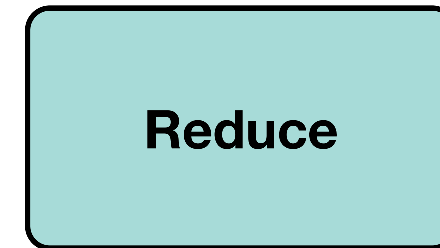
Operators



Operators run **user defined functions** (UDFs) and **describe** how data is handed to it.

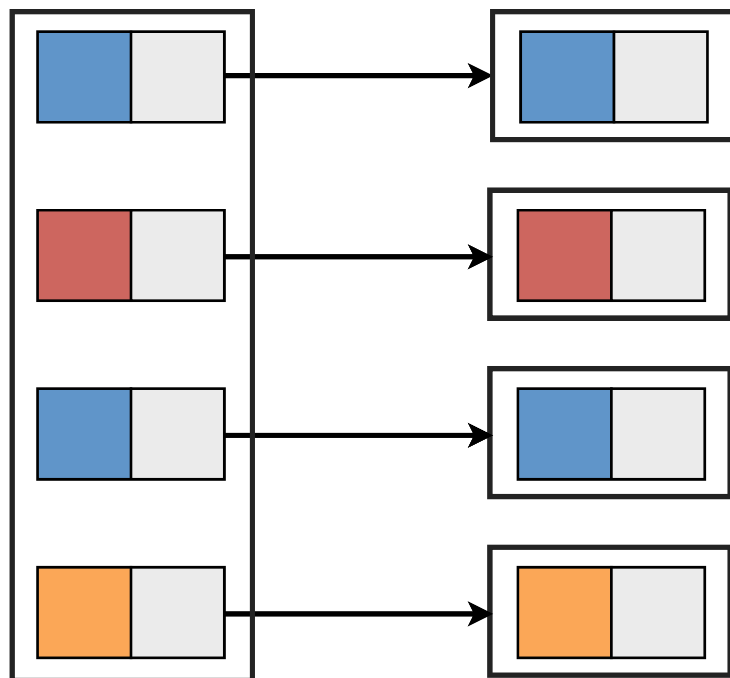

Map
Reduce
CoGroup
Cross
Union
Join
Iterate
Delta Iterate

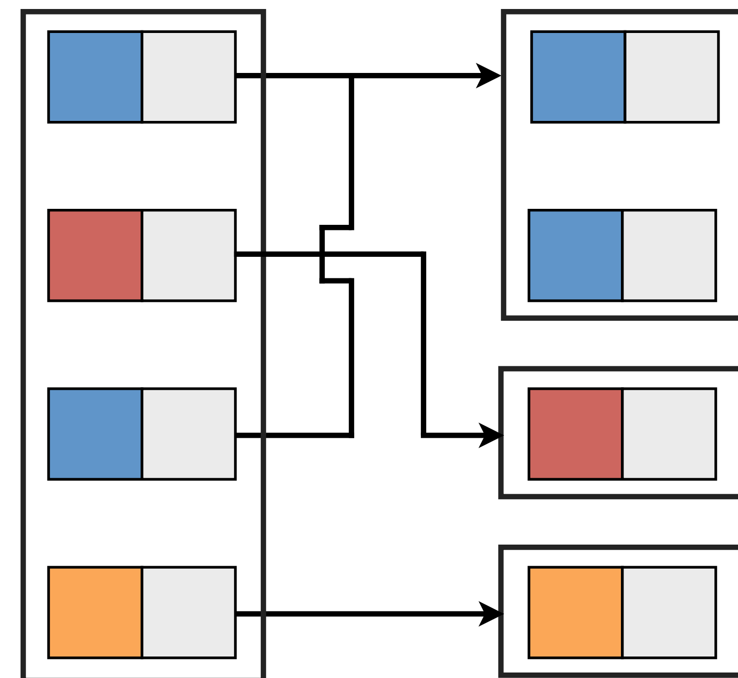
Map & Reduce



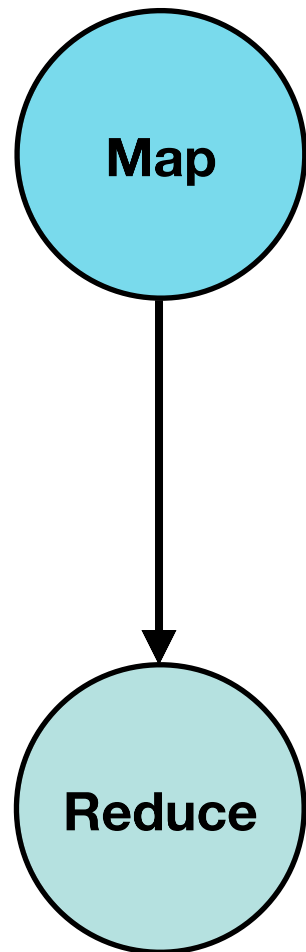
User code receives **one record at a time.**

User code receives **group of records** with **same key.**





Counting Words



map(Hello, World!) → Hello 1 World 1

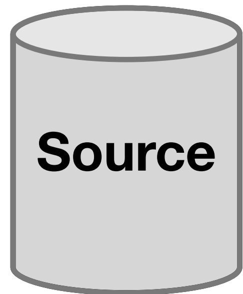
map(Hello, Helsinki!) → Hello 1 Helsinki 1

reduce(Hello 1 1) → Hello 2

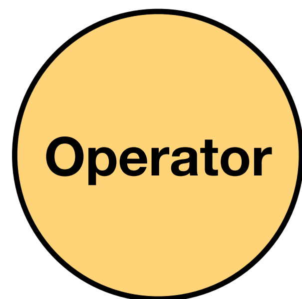
reduce(World 1) → World 1

reduce(Helsinki 1) → Helsinki 1

Stratosphere Program Skeleton



Input in internal **`DataSet`** representation



Transformations on the **`DataSet`**

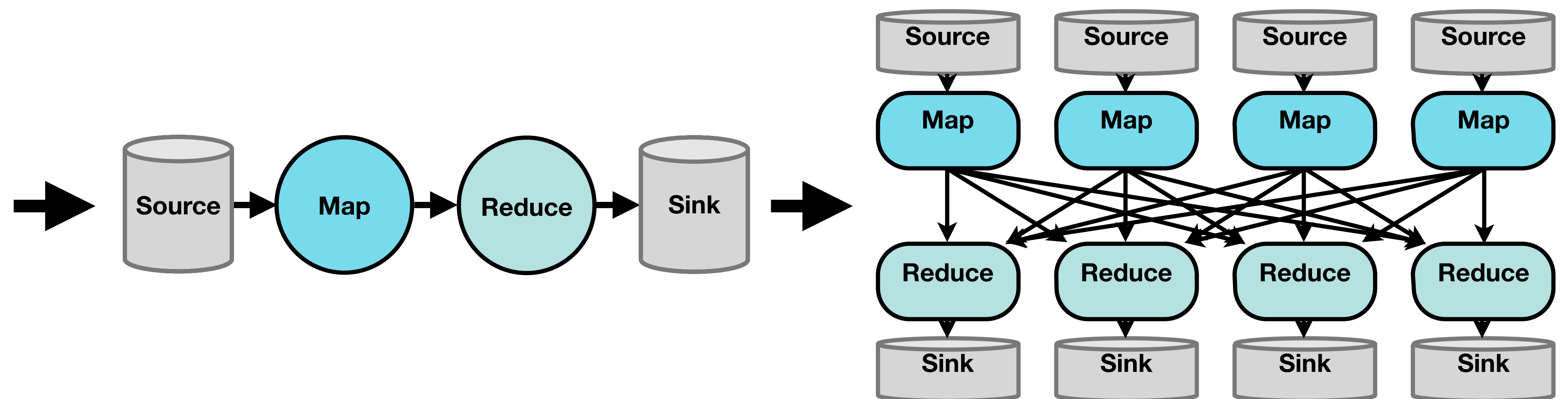
Output results in **`DataSink`**

The Almighty DataSet

Operators are methods on **DataSet**[A].

Applying operators to **DataSet** objects creates a **data flow** that can be executed.

```
val input =
TextFile(textInput)

val words = input.flatMap
{ line => line.split(" ") }

val counts = words
      .groupBy { word => word
}
      .count()

val output =
counts.write("file://",
CsvOutputFormat())

val plan = new
ScalaPlan(Seq(output), "Word
Count")

execute(plan)
```



An Important Difference

```
val input : List[String] = ...

val mapped = input.map { s => (s, 1) }
```

Executed **immediately**.

```
val input: DataSet[String] = ...

val mapped = input.map { s => (s, 1) }

val result = mapped.write(“file://”, ...)

val plan = new Plan(result)

execute(plan)
```

Executed when **data flow** is **executed**.

Creating Data Sources

```
// type: DataSet[String]
val input = TextFile(“hdfs://”)
```

```
// type: DataSet[(Int, String)]
val input = DataSource(“file://”, CsvInputFormat[(Int, String)]())
```

```
// type: DataSet[(Int, Int)]
def parseInput(): (Int, Int) = {...}

val input = DataSource(“file://”, DelimitedInputFormat)(parseInput)
```

Usable data types:

- Primitive types
- Tuples
- Case classes
- Custom data types (implementing **`Value`** interface)

Map Operator

```
val input: DataSet[(Int, String)] = ...
```

```
val mapped = input.map { (a, b) => (a + 1, b) }
```

```
(1, “foo”)                    (2, “foo”)
(2, “bar”)  ——map——>          (3, “bar”)
```

```
val filtered = input.filter { (a, b) => a > 1 }
```

```
(1, “foo”)
(2, “bar”)  ——filter——>       (2, “bar”)
```

```
val flapMapped = input.flatMap { (a, b) => b.split(“ “) }
```

```
                                   “foo”
(1, “foo bar”)                     “bar”
(2, “bar”)       ——flatMap——>      “bar”
```

Reduce Operator

```
val input: DataSet[(Int, String, Int)] = ...
```

```
val reduced = input
    .groupBy { (id, word, num) => word }
    .reduce { (w1, w2) => (w1._1 + w2._1, w1._2, w1._3 + w2._3) }
```

```
(1, “green”, 1)                              (10, “green”, 2)
(7, “blue”, 1)   ——groupBy, reduce——>        (10, “blue”, 2)
(6, “brown”, 1)                              (6, “brown”, 1)
(3, “blue”, 1)
(9, “green”,1 )
```

```
val input: DataSet[(Int, String)] = ...
```

```
val groupReduced = input
    .groupBy { (id, word) => word }
    .reduceGroup { _.minBy { (id, word) => id } }
```

```
(1, “green”)                                 (1, “green”)
(7, “blue”)   ——groupBy, reduceGroup——>      (3, “blue”)
(6, “brown”)                                 (6, “brown”)
(3, “blue”)
(9, “green”)
```

Creating Data Sinks

```
val counts: DataSet[(String, Int)] = ...
```

```
val sink = counts.write(“file://”, CsvOutputFormat())
```

```
def formatOutput(a: (String, Int)): String = {
    a._1 + “ occurs “ + a._2 + “ times.”
}

val sink = counts.write(“file://”, DelimitedOutputFormat(formatOutput))
```

Local and Remote Execution

```
val plan = new ScalaPlan(Seq(output), "Word Count")
```

```
val local = new LocalExecutor()

local.start()
local.executePlan(plan)
local.stop()
```

```
val remote = new RemoteExecutor("localhost", 6123, "some.jar")

remote.executePlan(plan)
```

Counting Words in Stratosphere

```scala
val input = TextFile(textInput)

val words = input.flatMap { line => line.split(" ") }

val counts = words
  .groupBy { word => word }
  .count()

val output = counts.write("file://", CsvOutputFormat())

val plan = new ScalaPlan(Seq(output), "Word Count")

execute(plan)
```

Map

Reduce

CoGroup

Cross

Union

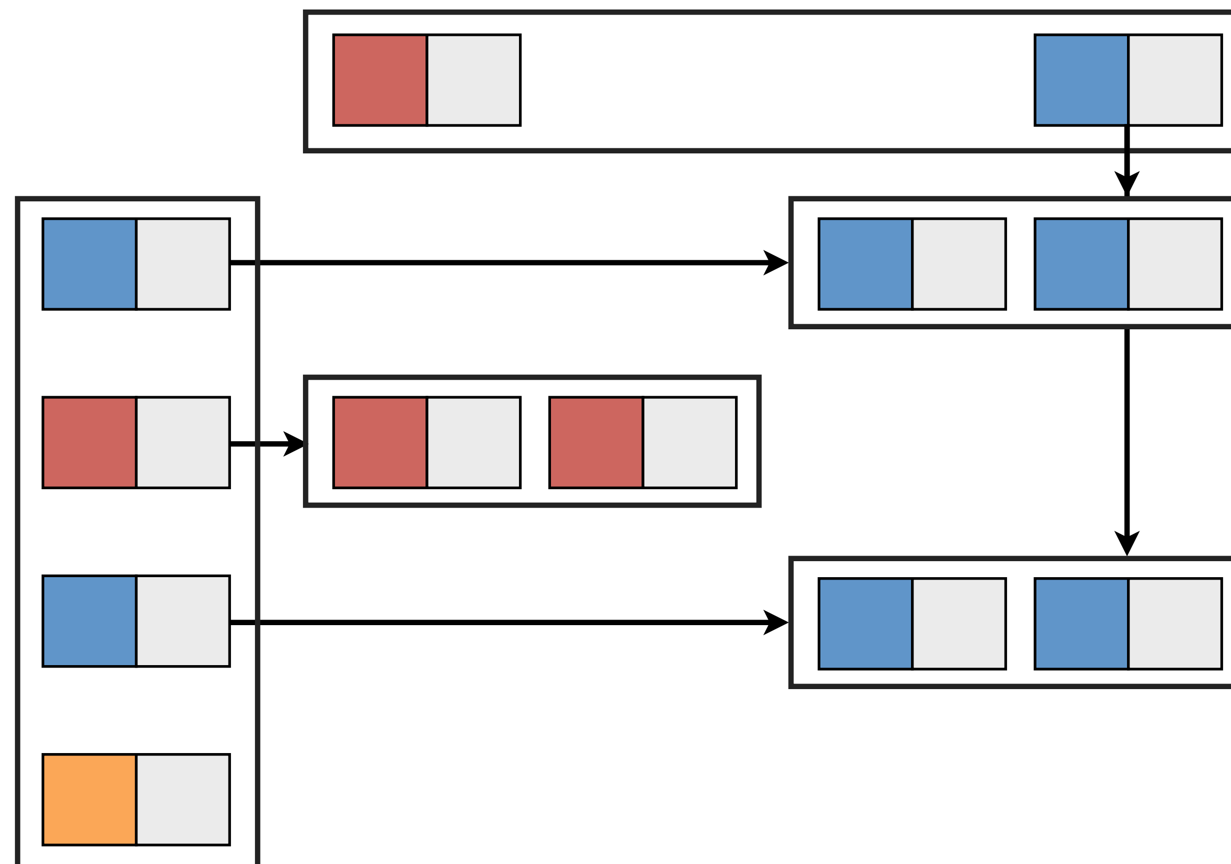
Join

Iterate

Delta Iterate

Join

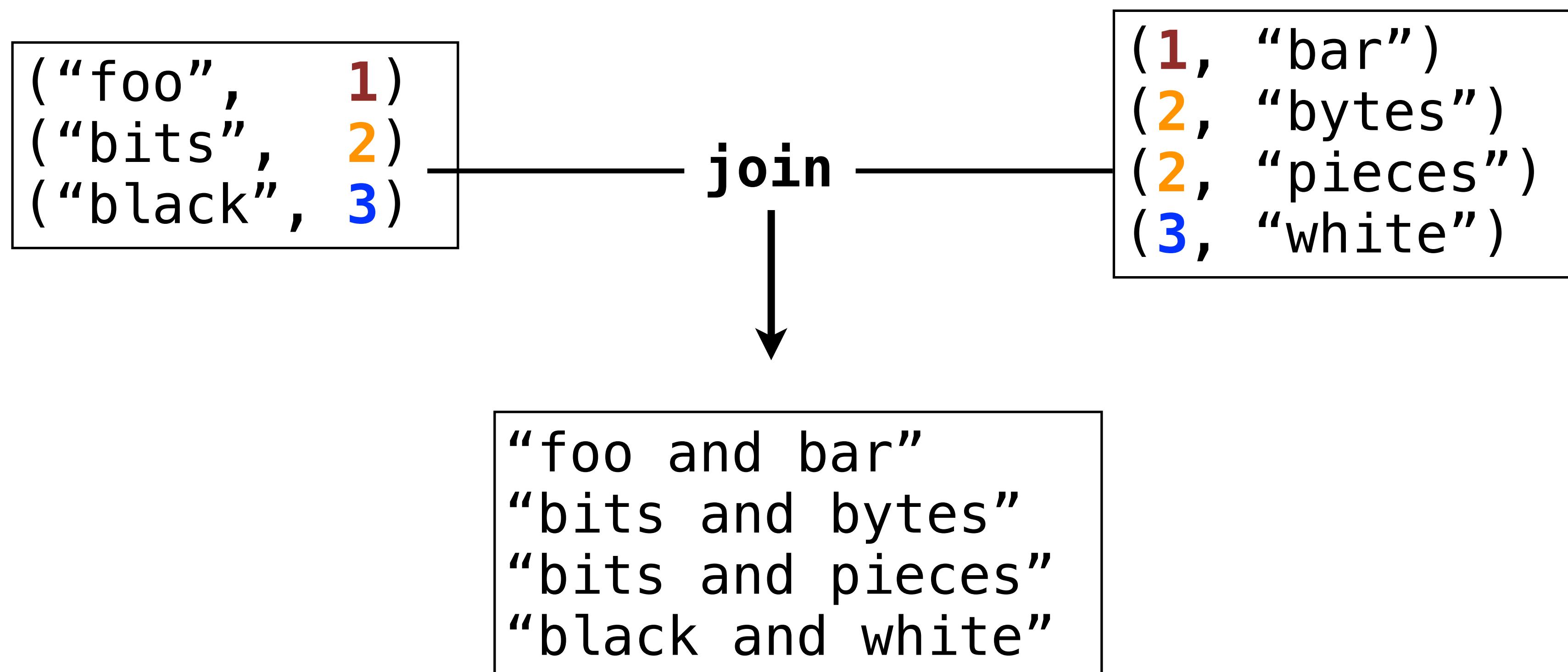
User code receives
records with same key from **two inputs**.



Join Operator

```
val counts: DataSet[(String, Int)] = ...
val names:  DataSet[(Int, String)] = ...
```

```
val join = counts
    .join(names)
    .where { x => x._2 } // key field of left input (counts)
    .isEqualTo { x => x._1 } // key field of right input (names)
    .map { (left, right) => left._1 + “and” + right._2 }
```



Map

Reduce

CoGroup

Cross

Union

Join

Iterate

Delta Iterate

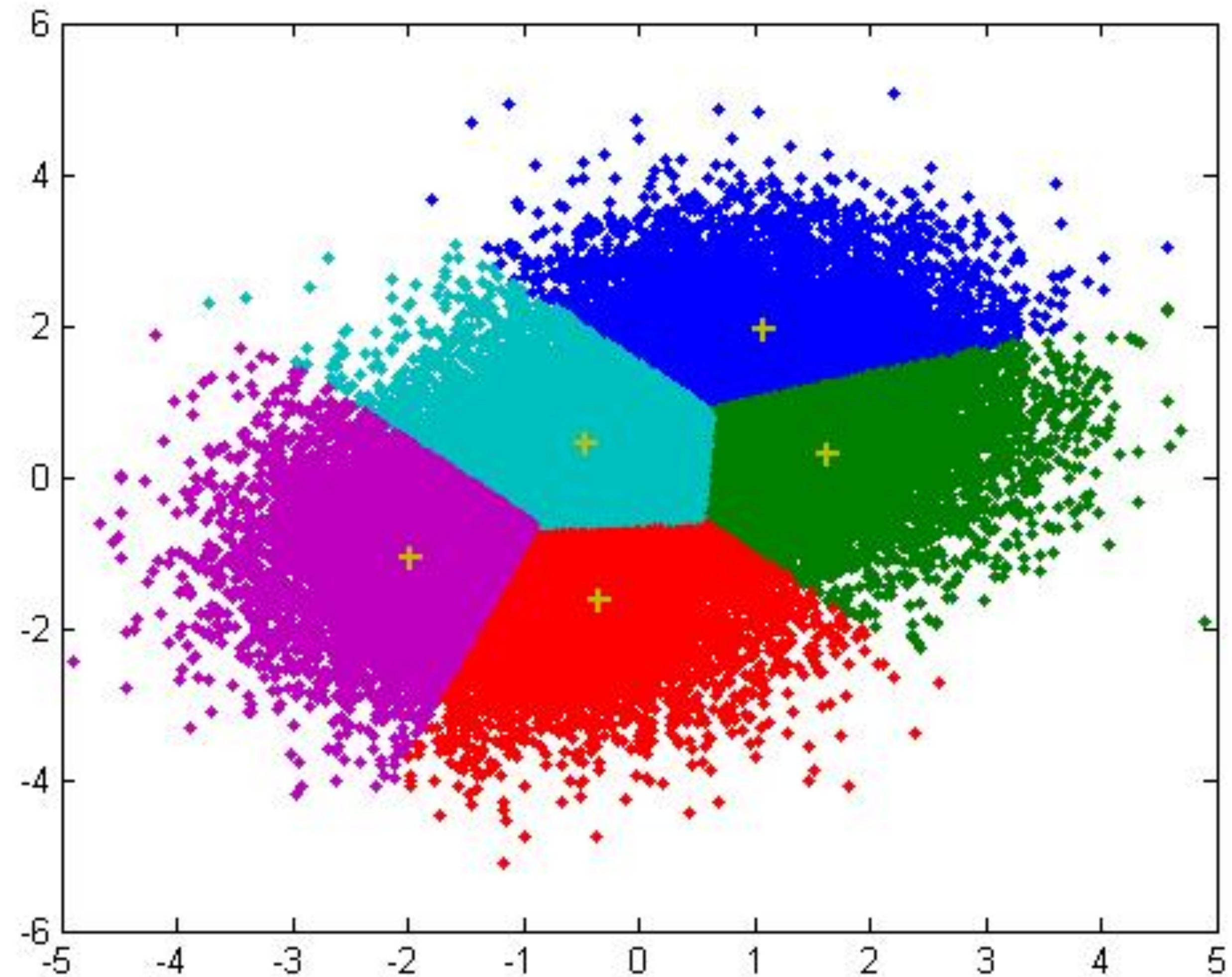
Iterations

Many Algorithms **loop over the data.**

- **Machine Learning**: iteratively refine model
- **Graph processing**: propagate information hop by hop

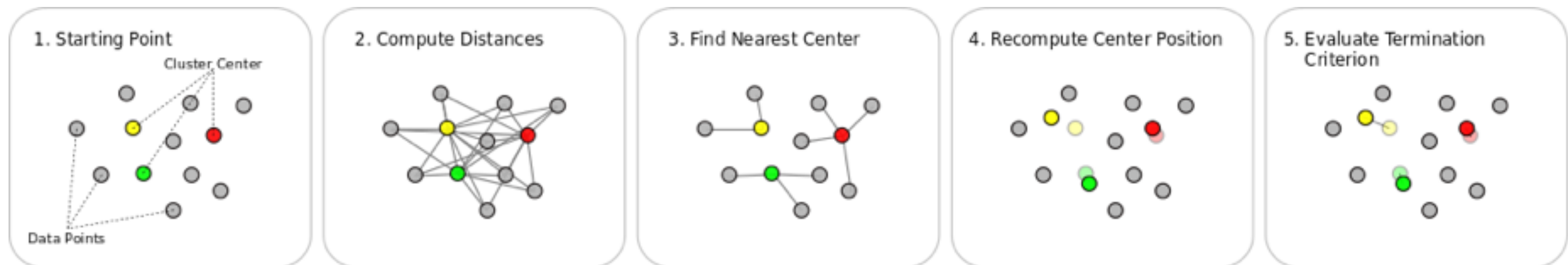
k-means clustering

Partition N vectors into k clusters minimizing within-cluster sum of squares distance.

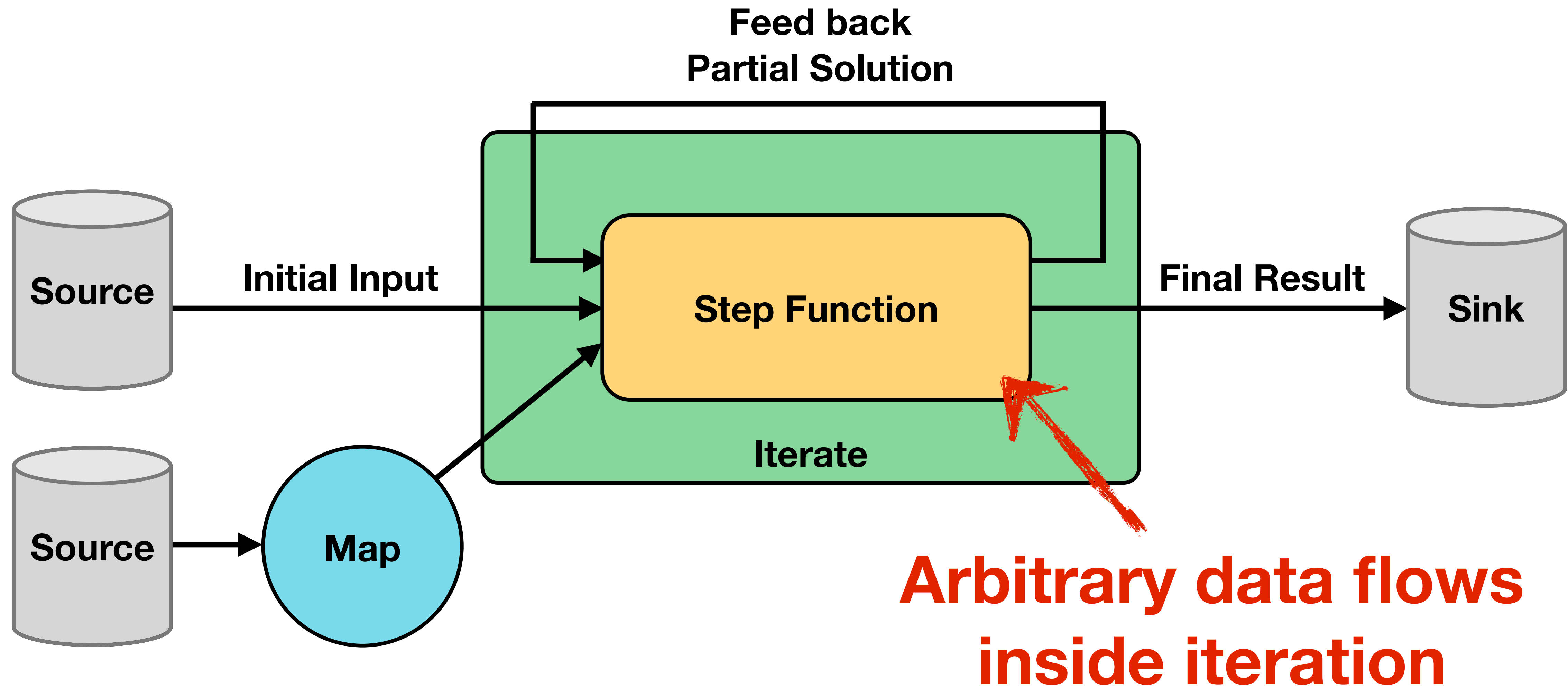


Simple and powerful clustering algorithm

Iterate:



Iterate



Iterations run arbitrary **data flows** and **iteratively update** the ***partial solution***.

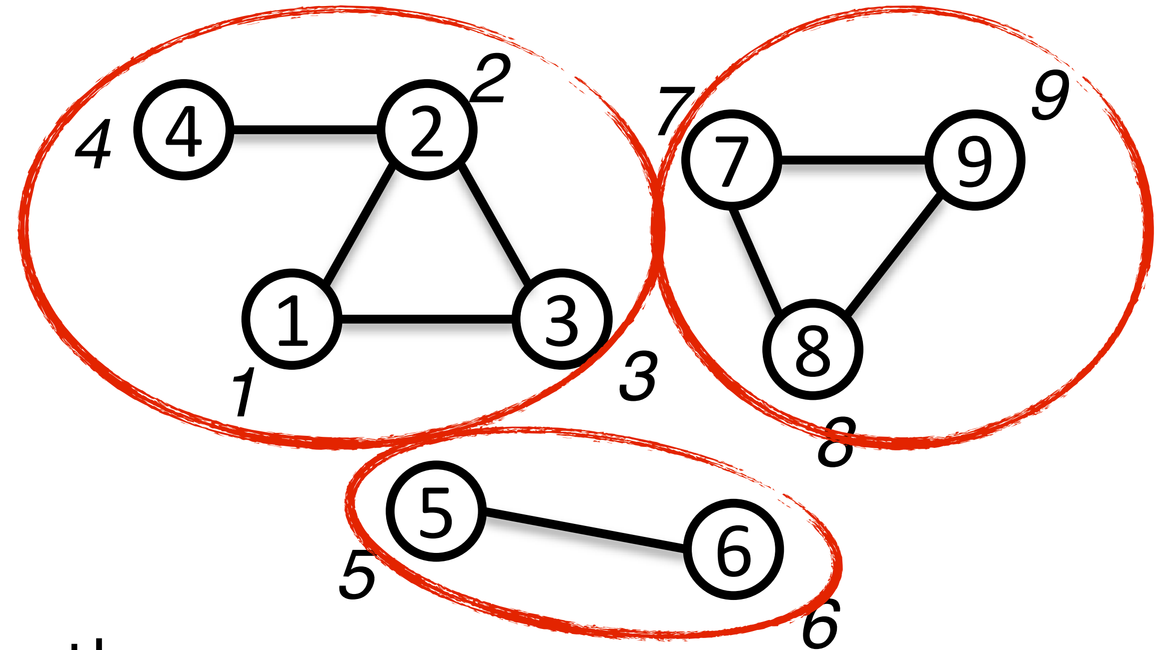
Iterate Operator

```
val input: DataSet[(Int, Int)] = ...

def step(partial: DataSet[(Int, Int)]) = {
    // possible to combine arbitrary operators here
    val nextPartial = partial.map { (a, b) => (a, b + 1) }

    nextPartial
}

val numIterations = 10;
val iteration = input.iterate(numIterations, step)
```

```
(1, 1)                          (1, 11)
(2, 2) ——iterate(10)——→         (2, 12)
```

Weakly connected components

Component 1 Component 2

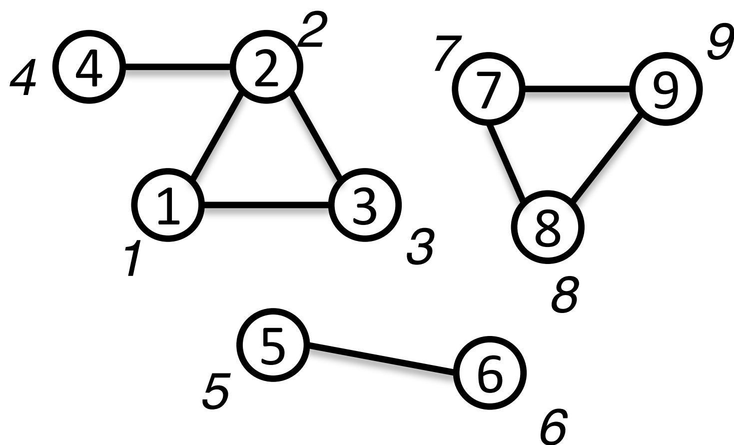


Component 3

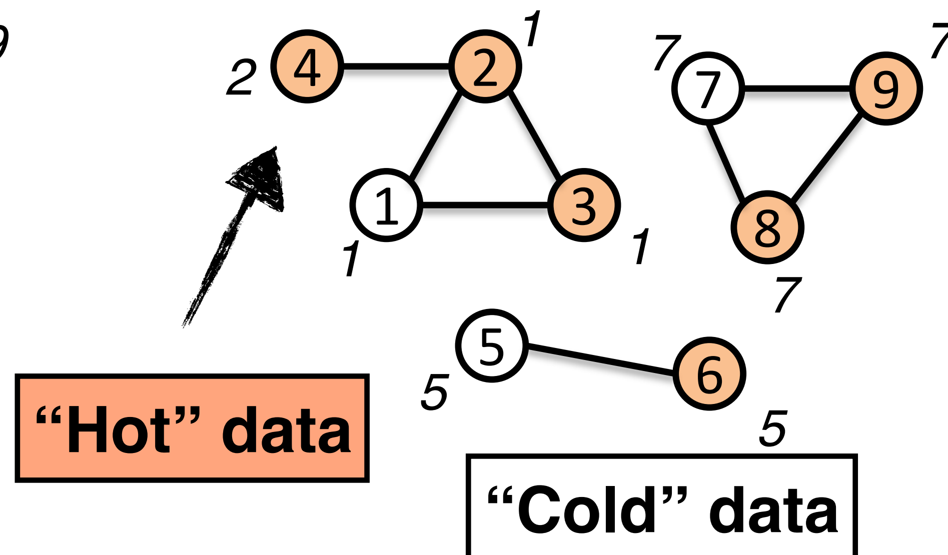
Given an undirected graph, find maximal subgraphs in which a path exists from each node to each node

E.g., to find communities in social networks

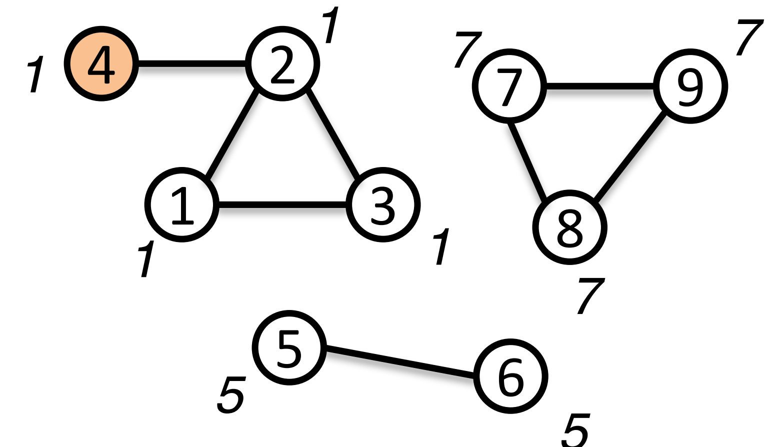
1: Assign initial labels



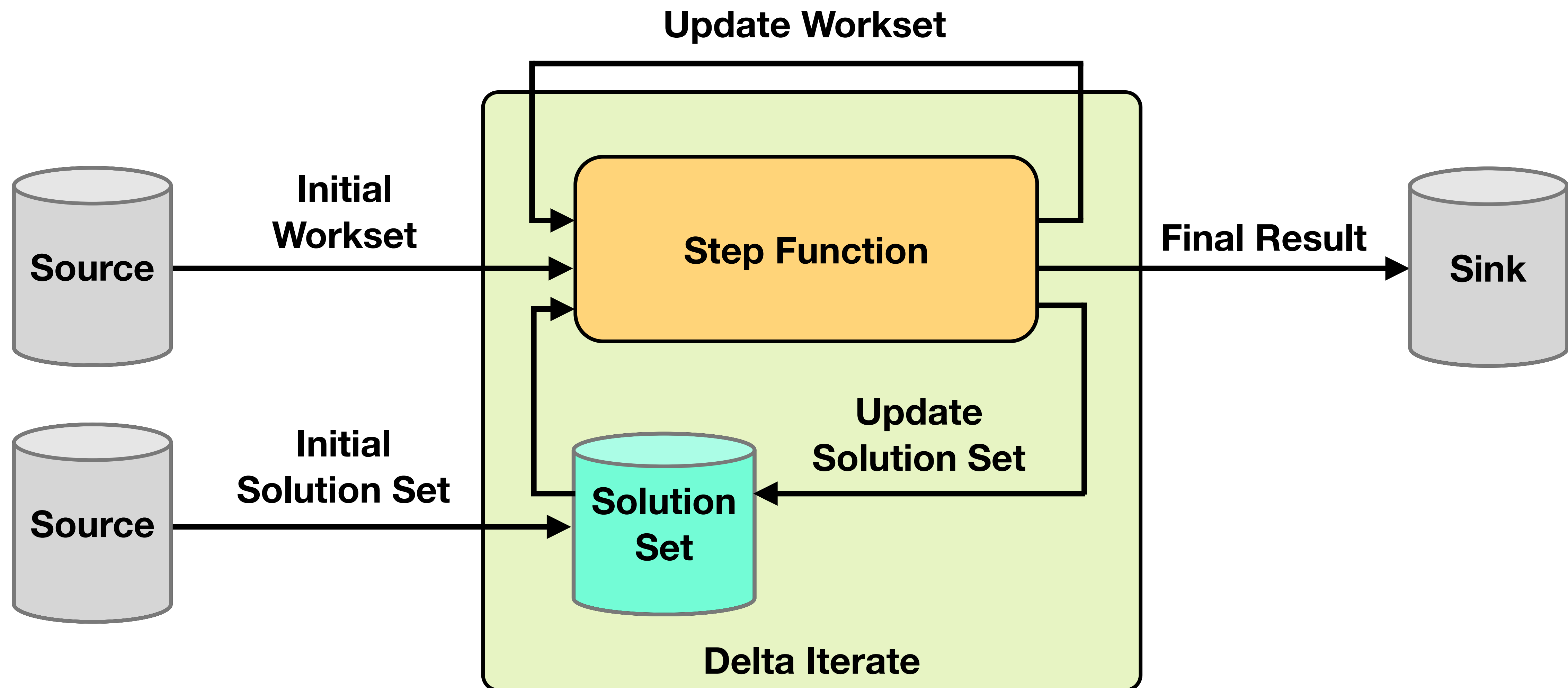
2: Get minimum label of neighbors



3: Iterate until convergence



Delta Iterate



Delta Iterations run arbitrary **data flows** and **iteratively update** the ***workset*** and ***solution set***.

Delta Iterate Operator

```
val input: DataSet[(Int, Int)] = ...
val initialWorkset: DataSet[(Int, Int)] = ...
val initialSolutionSet: DataSet[(Int, Int)] = ...
```

```
def step(ss: DataSet[(Int, Int)], ws: DataSet[(Int, Int)]) = {
    // possible to combine arbitrary operators here
    val delta = ...
    val nextWorkset = ...

    (delta, nextWorkset)
}

val maxIterations = 10;
val iteration = input.iterateWithWorkset(
        initialSolutionSet, solutionSetKey, step, maxIterations)
```

Backup Slides

Java API

The **Java API** provides the same set of **operators**.

Main differences to Scala:

- Data moves through operators as **`PactRecord`**s
- User code extends **operator stub classes**
- More **explicit** wiring of data flows

PactRecord

Data moves through operators as **PactRecord**s.

```
PactRecord record = new PactRecord();

PactInteger id = new PactInteger(1);
PactString name = new PactString("Stratosphere");

record.setField(0, id);
record.setField(1, name);

int i = record.getField(0, PactInteger.class).getValue();
```

Basic data types

PactInteger, PactDouble, PactString, PactBoolean.

Custom data types must implement **Value** interface.

Operator Stubs

```
public class Mapper extends MapStub {

    @Override
    public void map(PactRecord record,
                    Collector<PactRecord> collector) {

        // do stuff

        collector.collect(...);
    }
}
```

Operator Stubs

```
public class CountWords extends ReduceStub {

    @Override
    public void reduce(Iterator<PactRecord> records,
                       Collector<PactRecord> collector)
        throws Exception {

        while (records.hasNext()) {
            PactRecord current = records.next();
            // do stuff
        }

        collector.collect(...);
    }
}
```

Plan Wiring

```
FileDataSource source = new FileDataSource(TextInputFormat.class,
                                           dataInput);

MapContract mapper = MapContract.builder(TokenizeLine.class)
    .input(source)
    .name("Tokenize Lines")
    .build();

ReduceContract reducer = ReduceContract.builder(
    CountWords.class, PactString.class, 0)
    .input(mapper)
    .name("Count Words")
    .build();

FileDataSink out = new FileDataSink(RecordOutputFormat.class,
    output, reducer, "Word Counts");

RecordOutputFormat.configureRecordFormat(out)
    .recordDelimiter('\n')
    .fieldDelimiter(' ')
    .field(PactString.class, 0)
    .field(PactInteger.class, 1);
```

Map

Reduce

CoGroup

Cross

Union

Join

Iterate

Delta Iterate

CoGroup, Cross, Union

CoGroup: generalized reduce for **two inputs.**

Cross: cartesian product of **two inputs.**

Union: union of **two inputs.**