Parallel Data Processing

(as seen from the Stratosphere)

Alexander Alexandrov
alexander.alexandrov@tu-berlin.de

With material from V. Markl, F. Hueske, S. Ewen, K. Tzoumas

Outline

- Query Processing Fundamentals
  - Relational Algebra
  - Query Optimization Basics
- Parallel Query Processing
  - Pipelined vs. Data Parallelism
  - Data partitioning
- Stratosphere / Fling – Big Data meets Parallel Databases
  - Architecture
  - Programming API
  - Iterations

Query Processing Fundamentals

• • •

Relational Algebra & Relational Calculus

Query Optimization

Database Engines 101

Database engines provide high-level abstractions for efficient data management

- Storage
  - "Store {some data} as {X}."
- Querying
  - "Retrieve {something} from $\{X_1\}$, …, $\{X_n\}$ where {some condition} applies."
- Transactions
  - Ensures a minimal set of operational abstractions (ACID)
  - Required mostly in scenarios with concurrent read/write access. Not discussed here.

Query Processing Pipeline

SQL Query

Parsing

Logical Plan

Optimization

Physical Plan

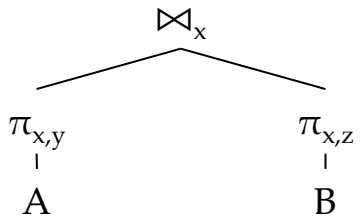
Execution

Result

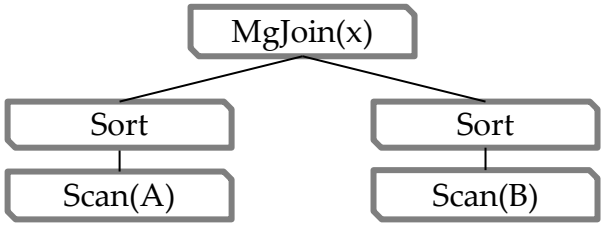
$\{(y,z) \mid (x,y) \in A, (x,z) \in B\}$

```
SELECT   A.y, B.z
FROM     A, B
WHERE    A.x = B.x;
```

declarative representation



(abstract)
functional representation



(concrete)
procedural representation

y	z
15	42
17	23

Data Model

- Databases traditionally rely on a **Relational Data Model**
  - Originally proposed by Codd in 1970

- Data represented as a set of **relations**
  - $R_1, R_2, ..., R_n$ example: Students, Professors, Lectures

- A relation is a subset over a **product** of **attribute domains**
  - $R \subseteq A_1 \times \ldots \times A_n$ example: Person $\subseteq$ Name × Gender × Birthday
  - A_i normally restricted to scalar types, some extensions (NF^2) allow relation nesting

- **Normalization** can be applied to improve model quality

Query Languages

- Two **formal languages** allow querying a relational model
  - **Calculus** and **Algebra**
  - Both are **equally powerful**
  - Again, many extensions exist (for example for querying NF^2 models)

- **Relational Calculus**
  - Query syntax similar to predicate logic, set comprehension syntax
  - **Variable**-oriented view example: $\{ (y, z) \mid (x, y) \in A, (x, z) \in B \}$

- **Relational Algebra**
  - Query syntax similar to functional language expressions
  - **Function**-oriented view example: $\pi_{y,z} (\bowtie_x (\pi_{x,y} (A), \pi_{x,y}(B)))$

Relational Algebra

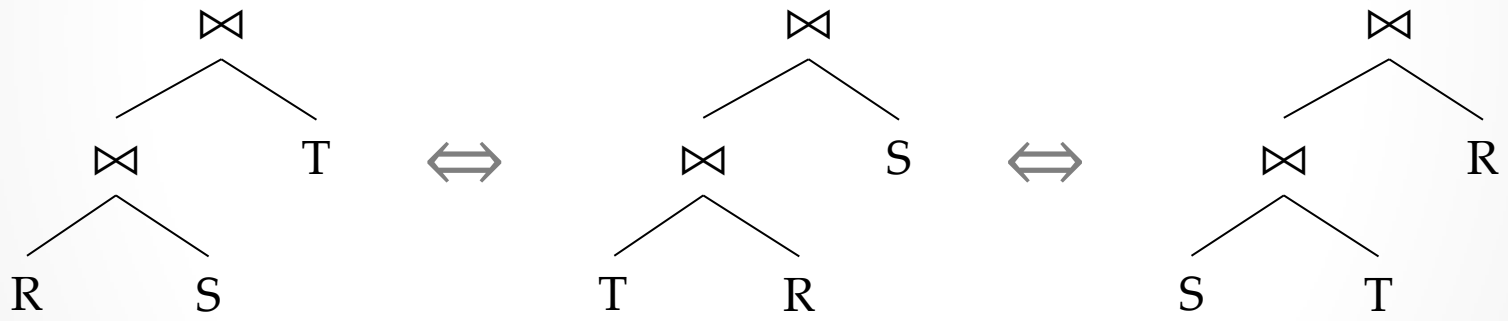
Operator	Semantics	Description
R, S, T, ...	Bag / Set	a collection of elements
$\pi_{\mathbf{A}}(R)$	$\{ \pi_{\mathbf{A}}(x) \mid x \in R \}$	projects a set of attributes **A**
$\sigma_{\mathbf{p}}(R)$	$\{ x \mid x \in S, p(x) \}$	filters elements where *p* is *false*
$R \times S$	$\{ (x, y) \mid x \in R, y \in S \}$	all pairs (Cartesian product)
$R \bowtie_{\theta} S$	$\{ (x, y) \mid x \in R, y \in S, \theta(x, y) \}$	theta-join, cross + filter
$R \bowtie_{\mathbf{A}} S$	$\{ (x, y) \mid x \in R, t \in S, \pi_{\mathbf{A}}(x) = \pi_{\mathbf{A}}(y) \}$	equi-join , *s*, *t* select the join key
$\gamma_{\mathbf{A}}(R)$	$\{ G_x = \{ y \mid y \in R, \pi_{\mathbf{A}}(y) = \pi_{\mathbf{A}}(x) \} \mid x \in R \}$	group by **A**
$\gamma_{\mathbf{A}, a}(R)$	$\{ \{ \pi_{\mathbf{A}}(y), agg(G_x) \} \mid x \in R \}$	group and compute aggregate *agg*
$S \cup T$ / $S \uplus T$		bag / set union
$S - T$ / $S \dot{-} T$		bag / set difference

Query Optimization

- Parsed query is represent as a relational algebra expression
  - (abstract) logical plan with well-defined semantics
- The optimizer translates it into a (concrete) execution plan
- Multiple degrees of freedom during the compilation process
  - Algebraic rewrites
  - Algorithm selection

Algebraic Rewrites

- Makes use of equivalences of relational algebra operators
  - Join operator is associative
  - Selections & projections can may commute joins, sometimes aggregates



join commutativity

Algorithm Selection

- Logical operators can be realized in different ways
  - Different time / space requirements depending on the concrete algorithm choice
  - Applicability depends on certain “physical properties” of the inputs (e.g. sorting)

S\T	1	3	1	4
7	(7,1)	(7,3)	(7,1)	(7,4)
4	(4,1)	(4,3)	(4,1)	**(4,4)**
1	**(1,1)**	(1,3)	**(1,1)**	(1,4)
4	(4,1)	(4,3)	(4,1)	**(4,4)**
3	(3,1)	**(3,3)**	(3,1)	(3,4)
2	(2,1)	(2,3)	(2,1)	(2,4)

Nested-Loops Join

S\T	1	1	3	4
1	**(1,1)**	**(1,1)**	(1,3)	(1,4)
2	(2,1)	(2,1)	(2,3)	(2,4)
3	(3,1)	(3,1)	**(3,3)**	(3,4)
4	(4,1)	(4,1)	(4,3)	**(4,4)**
4	(4,1)	(4,1)	(4,3)	**(4,4)**
7	(7,1)	(7,1)	(7,3)	(7,4)

Sort-Merge Join
(S, T sorted on join key)

Optimizer Search Space

- Orthogonal degrees of freedom ⇒ exponential search space
  - Can be restricted using heuristics, branch & bound pruning
- Finding the optimal execution plan is non-trivial
  - Decision can be cost-based or rule-based
- Algorithm costs are data-dependent
  - Typically dominated by a bottleneck resource (I/O or network)
  - Information required to calculate them can be approximated using data statistics

Parallel Query Processing

...

Pipelined vs. Data Parallelism

Data Partitioning

Parallel DB Architectures

- Shared Memory
  - Several CPUs share a single memory space and (multiple) disks
  - Communication over a single common bus

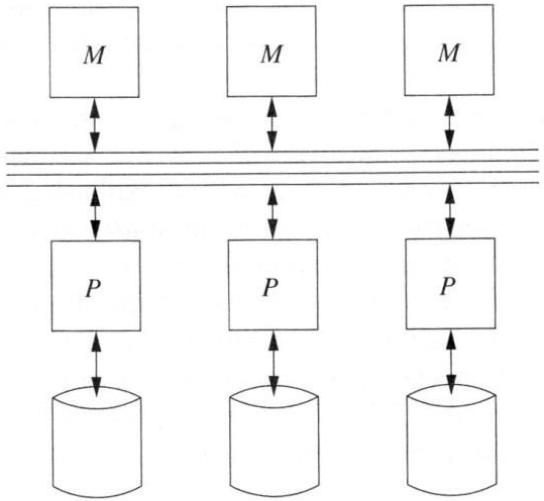


Figure 20.1: A shared-memory machine

Source:
Garcia-Molina et al., „Database Systems – The Complete Book. Second Edition“

Parallel DB Architectures

- Shared Disk
  - Several nodes with multiple CPUs, each node has its private memory
  - Single attached disk (array): Often NAS, SAN, etc…

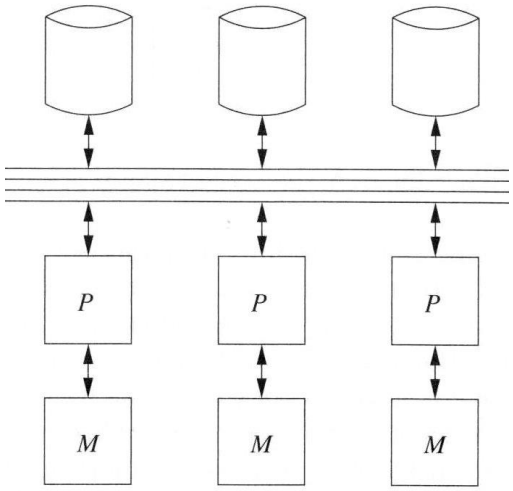


Figure 20.2: A shared-disk machine

Source:
Garcia-Molina et al., „Database Systems – The Complete Book. Second Edition“

Parallel DB Architectures

- Shared Nothing
  - Each node has it own set of CPUs, memory and disks attached
  - Data needs to be partitioned over the nodes
  - Data is exchanged through direct node-to-node communication

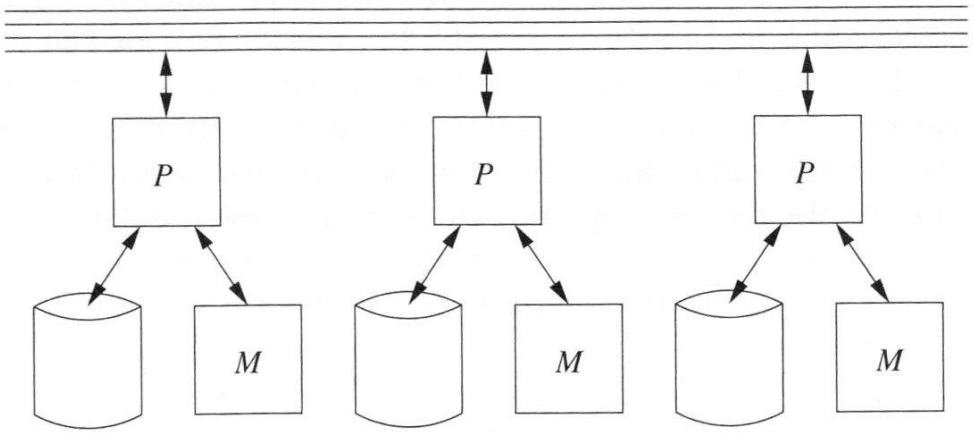


Figure 20.3: A shared-nothing machine

Source:
Garcia-Molina et al., „Database Systems – The Complete Book. Second Edition“

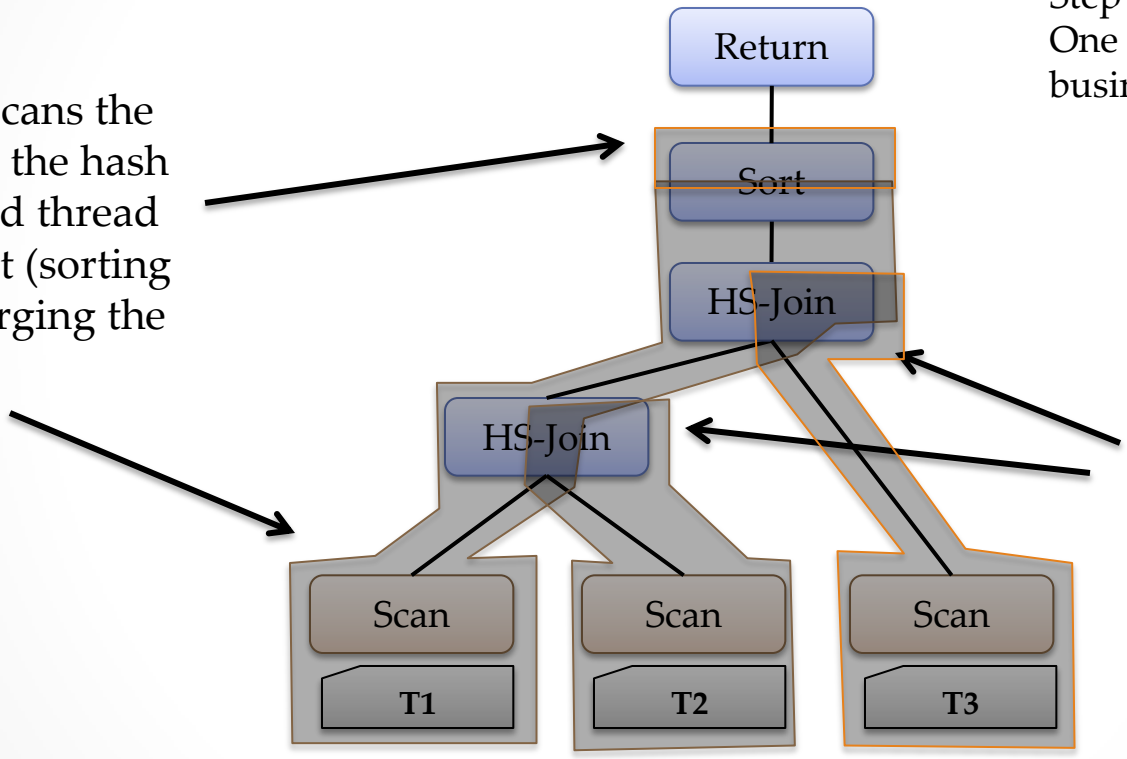
Query Parallelism

- **Inter-Query Parallelism** (concurrent queries)
  - Necessary for efficient resource utilization: While one query waits (e.g. for I/O), another one executes.
  - Requires concurrency control (locking mechanisms) to guarantee transactional properties like isolation.

- **Intra-Query Parallelism** (within a single query)
  - **I/O Parallelism**: Concurrent reading from multiple disks
  - **Data Parallelism**: Multiple threads work on the same operator
  - **Pipeline Parallelism**: Multiple pipelined parts of the plan run in parallel

Pipeline Parallelism

Step 2:
One thread scans the table, probes the hash tables. Second thread starts the sort (sorting sub-lists, merging the first lists)



Step 3:
One thread, return result, business as usual…

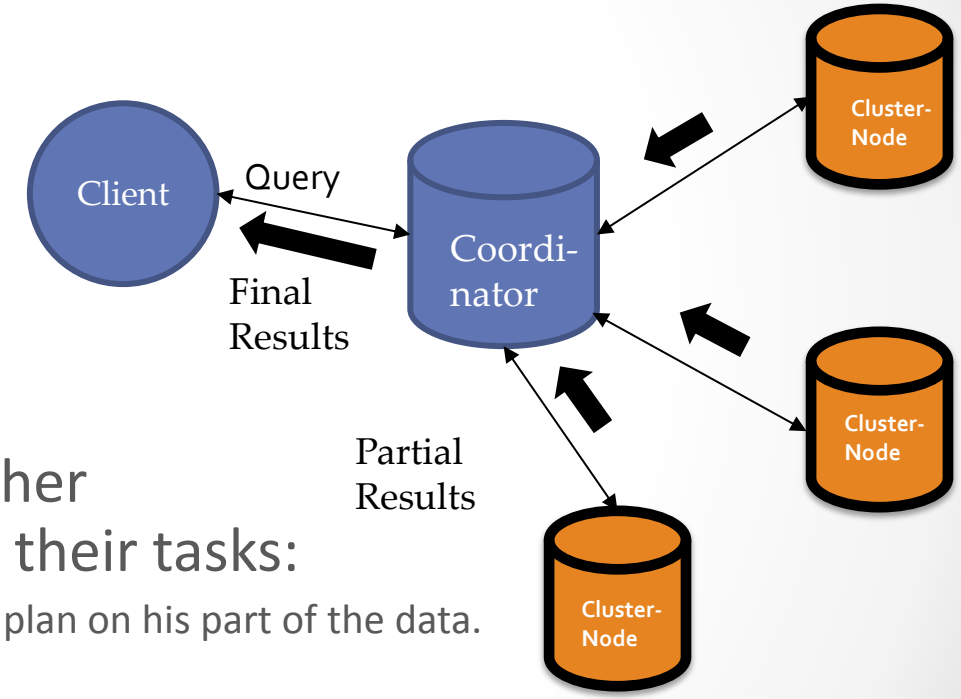
Step 1:
Two threads scan one base table each and build the hash tables for the joins.

Data Parallelism

- Data is divided into several sub-sets
  - Most operations don't need a complete view of the data!
    - E.g. $\sigma_p(\cdot)$ looks only at a single tuple at a time.
  - Subsets can be processed independently and hence in parallel.
- Degree of Parallelism as high as the number of possible subsets
  - For $\sigma_p(\cdot)$ as high as the number of tuples
- Some operations possibly need a view of larger portions of the data:
  - E.g. Grouping/Aggregation operation needs all tuples with the same grouping key.
  - Are they all in the same set? Can we guarantee that?
  - Different operators need different sets!

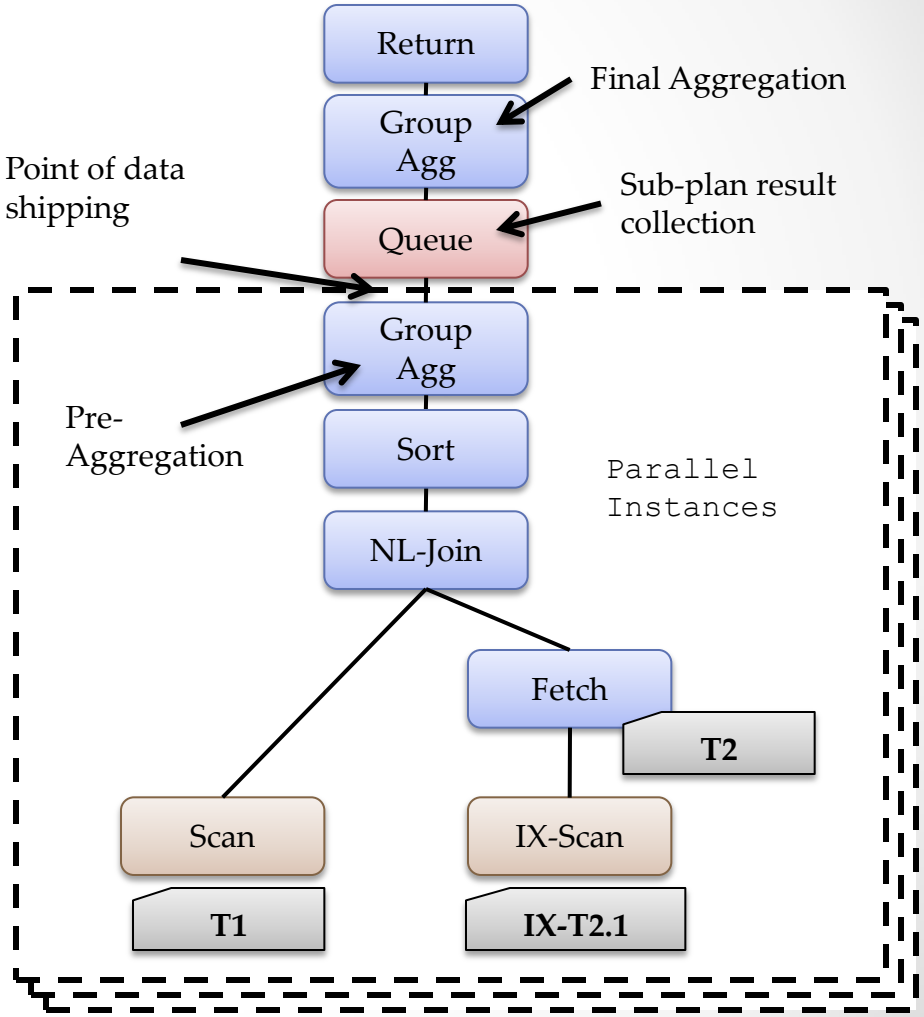
Data Parallelism Workflow

- Client send a SQL query to one of the cluster nodes:
  - Node becomes the "coordinator".
- Coordinator compiles the query:
  - Parsing, Checking, Optimization.
  - Parallelization.
- Sends partial plans to the other cluster nodes that describes their tasks:
  - Coordinator also executes the partial plan on his part of the data.
- Collects partial results and finalizes them (see next slide)



Data Parallelism Example

- For shared-nothing & shared-disk systems
  - Multiple instances of a sub-plan are executed on different computers.
  - The instances operate on different splits or partitions of the data.
  - At some points, results from the sub-plans are collected.
  - For more complex queries, results are not collected but re-distributed, for further parallel processing.



Data Partitioning

- Partitioning the data means creating multiple disjoint sub-sets
  - Example: Sales data, every year gets its own partition.
- For shared-nothing, data must be partitioned across nodes:
  - If it were replicated, it would effectively become a shared-disk with the local disks acting like a cache (must be kept coherent).
- Partitioning with certain characteristics has more advantages:
  - Some queries can be limited to operate on certain sets only, if it is provable that all relevant data (passing the predicates) is in that partition.
  - Partitions can be simply dropped as a whole (data is rolled out) when it is no longer needed (e.g. discard old sales).

Data Partitioning

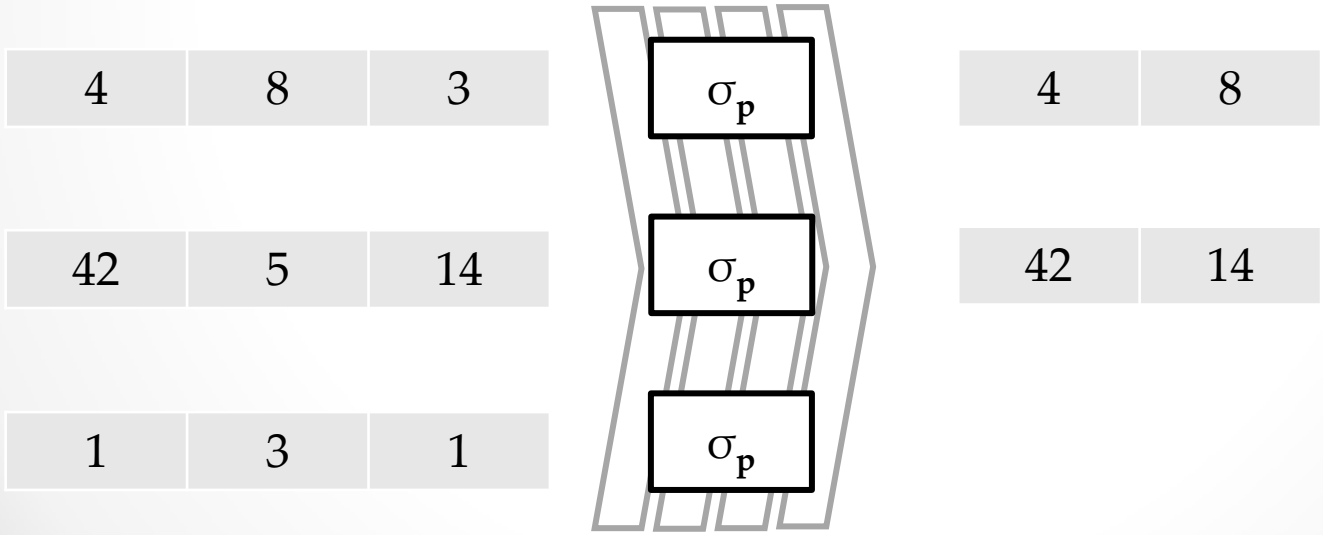
- **Round robin**: Each set gets a tuple in a round, all sets have guaranteed equal amount of tuples, no apparent relationship between tuples in one set.

- **Hash Partitioned**: Define a set of partitioning columns. Generate a hash value over those columns to decide the target set. All tuples with equal values in the partitioning columns are in the same set.

- **Range Partitioned**: Define a set of partitioning columns and split the domain of those columns into ranges. The range determines the target set. All tuples on one set are in the same range.

Parallel Selection

- Each node performs the selection on its existing local partition.
  - Selection needs no context.
  - Data can be partitioned in a arbitrary way.
  - Partial results *union*-ed afterwards.



Parallel Aggregation

- Re-partition dataset on grouping column set
  - Tuples with the same grouping values will end up on the same machine
  - Apply local grouping/aggregation algorithm to each partition in parallel

- Examples:
  - $sum(S.c) = \sum_{i=1}^{p} sum(S[i,h].c)$
  - $min(S.c) = min\big(min(S[1,h].c), \dots, min(S[p,h].c)\big)$
  - $avg(S.c) = \sum_{i=1}^{p} sum(S[i,h].c) / \sum_{i=1}^{p} count(S[i,h].c)$

- Not possible if the aggregation function requires sorting
  - E.g. Median, Quantiles

Parallel Equi-Joins

- A special class of joins suited for parallelization are Equi-Joins.
  - Only look at tuple pairs that share the same join key
  - Partition relations R and S using the same partitioning scheme over the join key
  - All values of R and S with the same join key end up at the same node
  - **All joins can be performed locally**

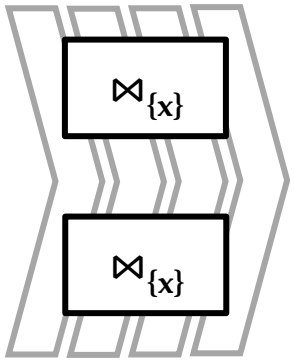
- Multiple partitioning strategies possible:
  - Co-Located Join
  - Directed Join
  - Re-Partitioning Join

Co-Located Join

- Data is already partitioned on the join key
  - No re-partioning needed
  - Local joins work “out of the box”

2	8	10
2	4	4

5	3	1
3	5	9

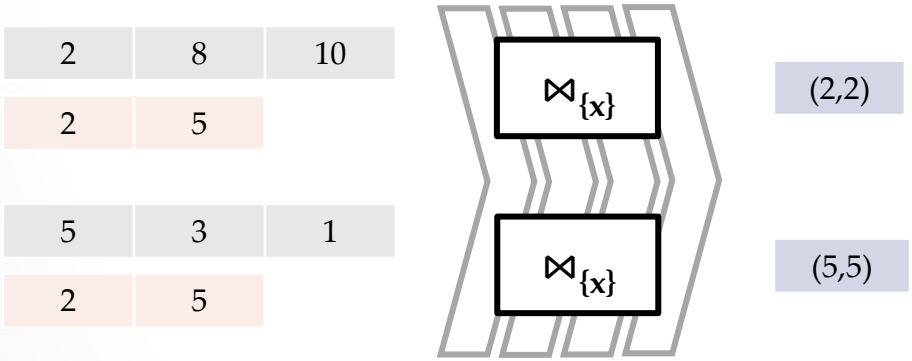


(2,2)

(5,5) (3,3)

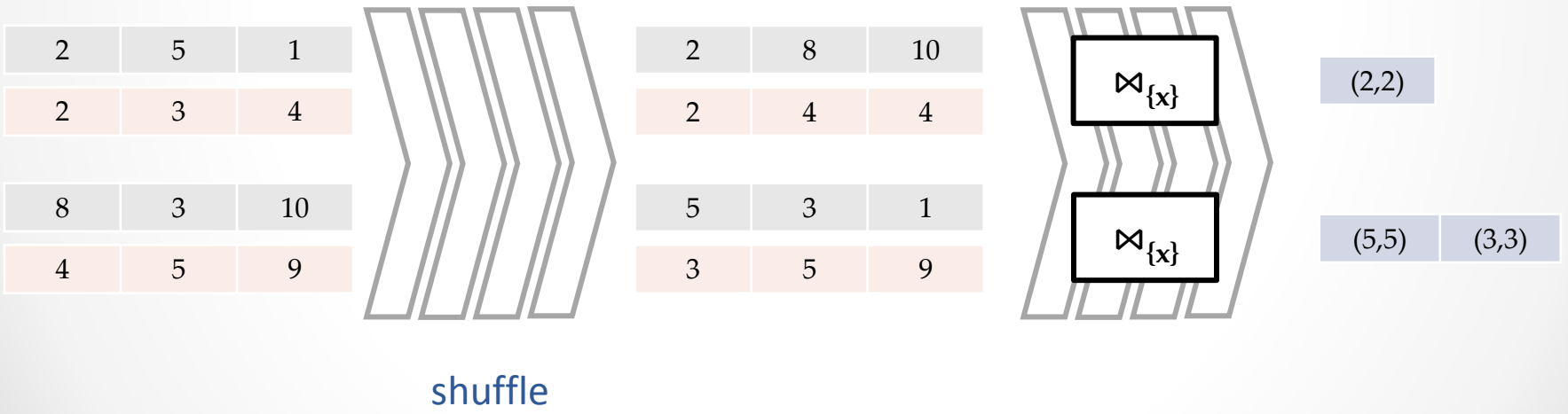
Directed Join

- One side is fully replicated on each node
  - No re-partioning for the other side needed
  - Works best if one side is much smaller than the other



Re-Partitioning Join

- Both-sides re-partitioned on the join key
  - Fallback strategy for inputs with similar size



Stratosphere

Next-Gen Data Analytics Platform

Project status

- Research project started in 2009 by TU Berlin, HU Berlin, HPI
- Now a growing open source project with first industrial adoption

- Apache Incubator accepted, moving soon
- v0.4 - stable & documented, v0.5 – beta status

Introducing Stratosphere

General Purpose Data Analytics Platform.

Database Technology

MapReduce-style Technology

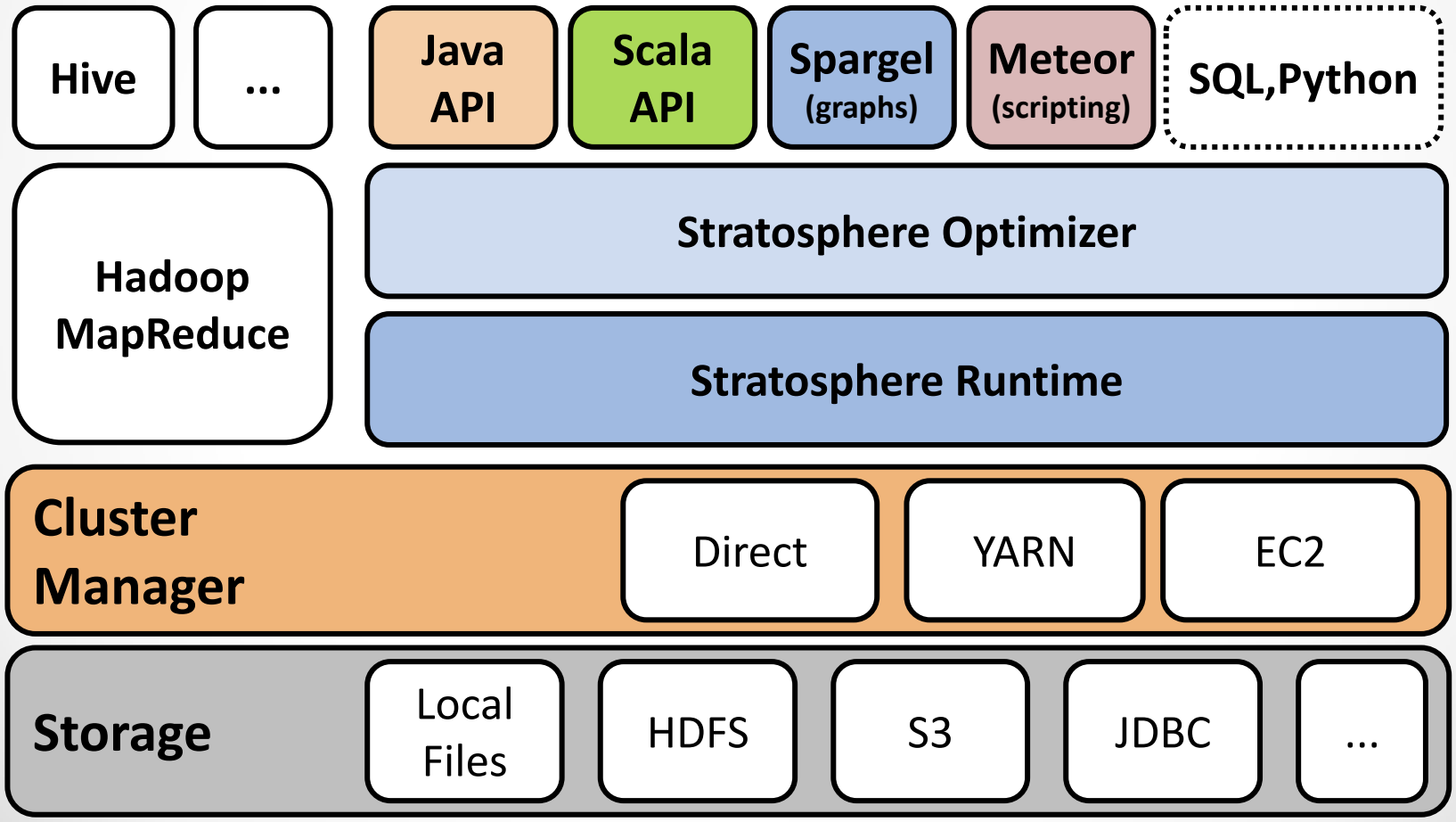
- Declarativity for SQL
- Optimizer
- Efficient Runtime

Stratosphere

- Iterations
- Advanced Dataflows
- Declarativity

- Scalability
- User-defined functions (UDFs)
- Complex data types
- Schema on read

Stratosphere Stack



Key Features

Easy to use developer APIs

- Java, Scala, Graphs, Nested Data (Python & SQL under development)
- Flexible data pipelines

Automatic Optimization

- Join algorithms
- Operator chaining
- Reusing partitioning/sorting

High Performance Runtime

- Complex DAGs of operators
- In memory & out-of-core
- Data streamed between operations

Native Iterations

- Embedded in the APIs
- Data streaming / in-memory
- Delta iterations speed up many programs by orders of mag.

Concise & rich APIs

Word Count in Stratosphere, Java API

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

DataSet<Tuple2<String, Integer>> result = text
                                        .flatMap(new Splitter())
                                        .groupBy(0).aggregate(SUM, 1);


// map function implementation
class Splitter extends FlatMap<String, Tuple2<String, Integer>> {

    public void flatMap(String value, Collector out){
        for (String token : value.split("\\W")) {
            out.collect(new Tuple2<String, Integer>(token, 1));
        }
    }
}
```

Can use regular POJOs!

Concise & rich APIs

Word Count in Stratosphere

Scala API

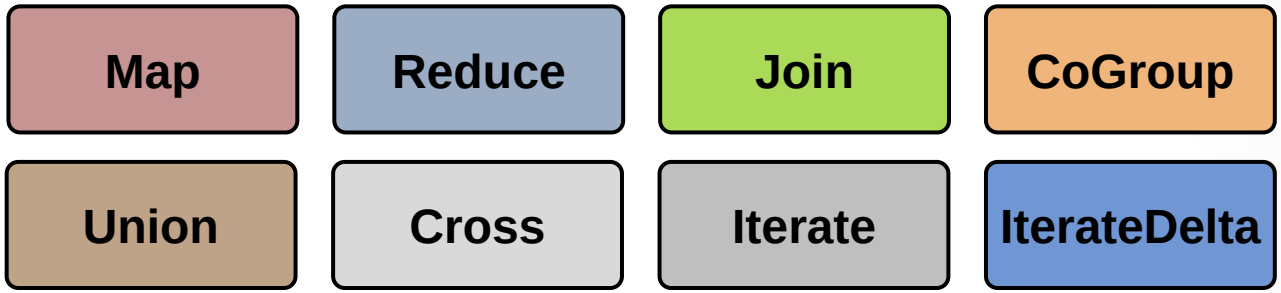
```
val input = TextFile(textInput)
val words = input flatMap { line => line.split("\\W+") }
val counts = words groupBy { word => word } count()
```

Stratosphere APIs

...

Operator Model

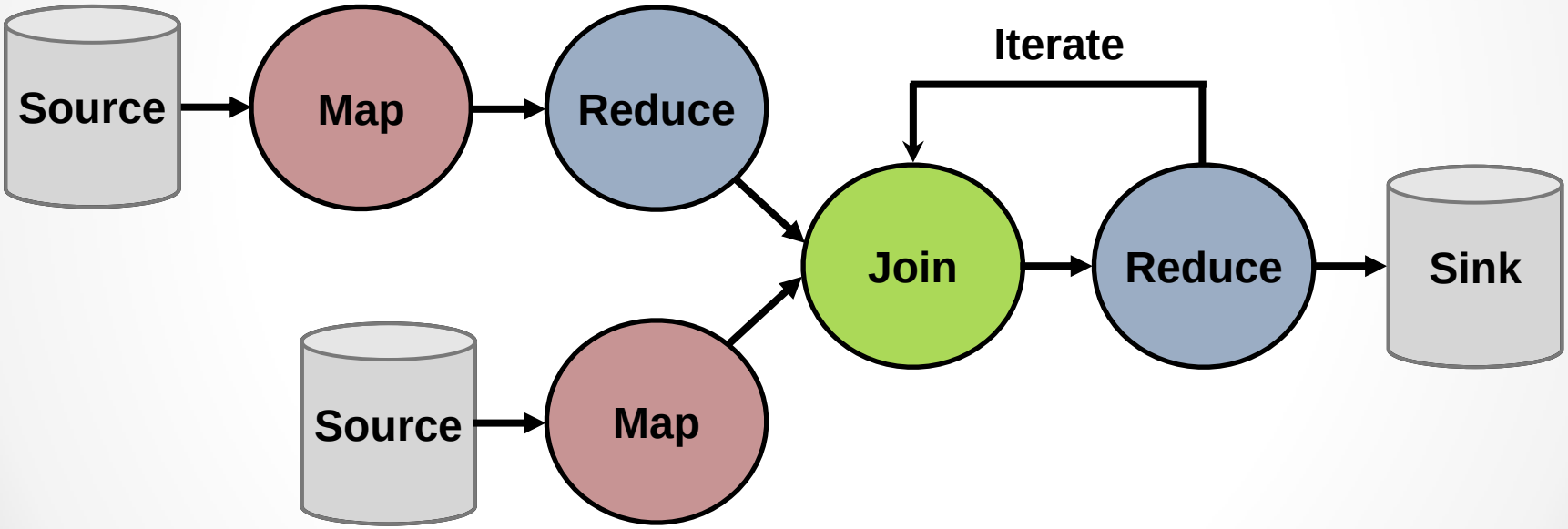
Basic Operators



Derived Operators

- Filter, FlatMap, Project
- Aggregate, Distinct
- Outer-Join, Semi-Join, Anti-Join
- Vertex-Centric Graphs computation (Pregel style)
- ....

Flexible Data Pipelines

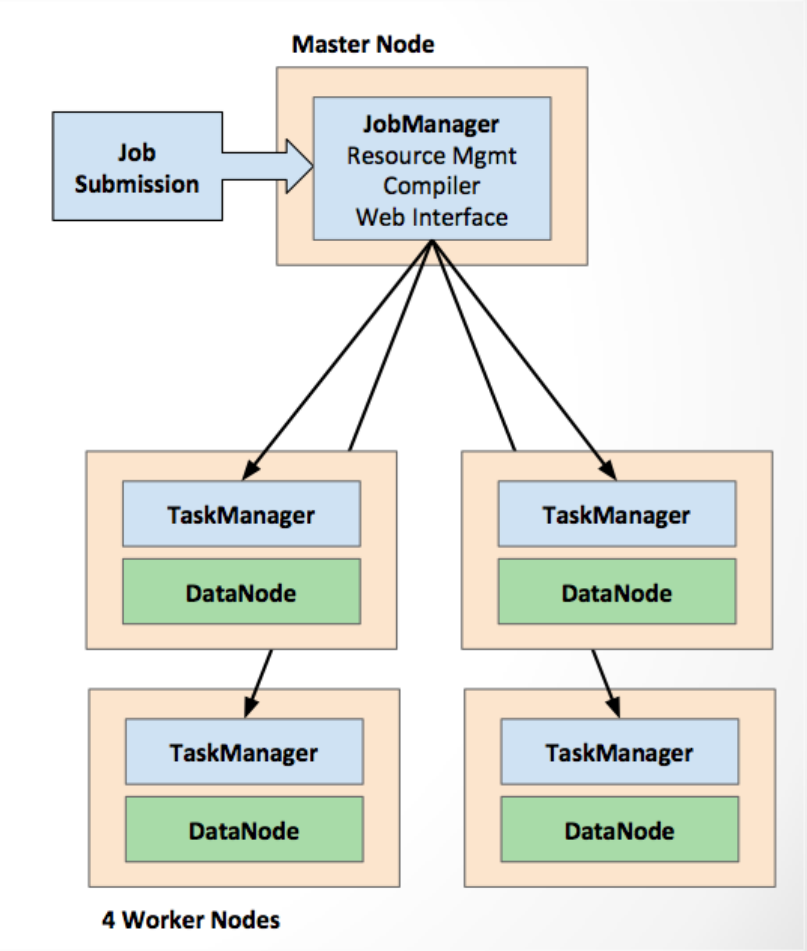


Stratosphere Runtime

...

Distributed Runtime

- Master (Job Manager) handles job submission, scheduling, and metadata
- Workers (Task Managers) execute operations
- Data can be streamed between nodes
- All operators start in-memory and gradually go out-of-core

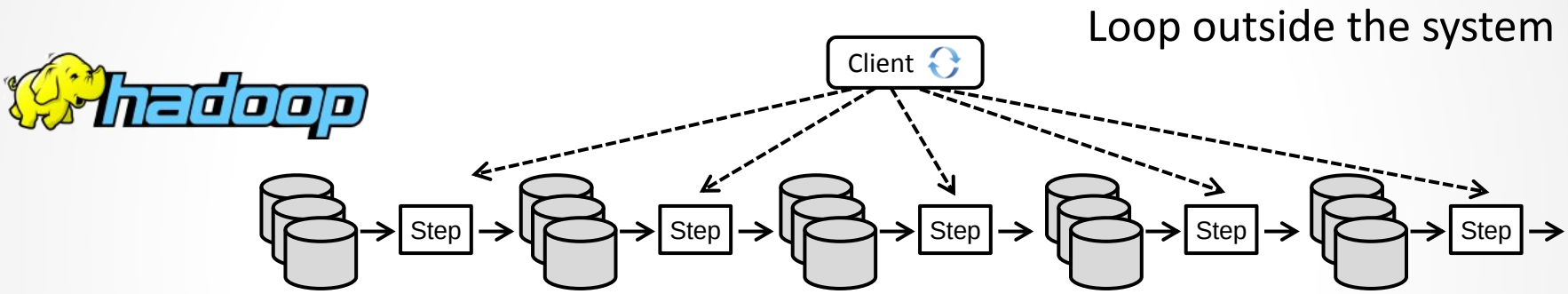


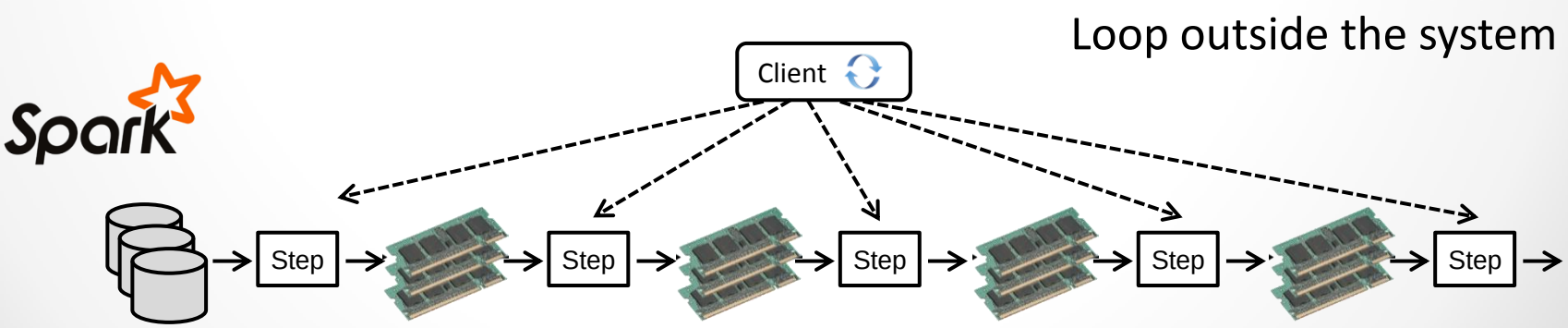
Iterative Programs

Iterative Algorithms

- Algorithms that need iterations
  - Clustering (K-Means, Canopy, …)
  - Gradient descent (e.g., Logistic Regression, Matrix Factorization)
  - Graph Algorithms (e.g., PageRank, Line-Rank, components, paths, reachability, centrality)
  - Graph communities / dense sub-components
  - Inference (belief propagation)
  - …
- Loop makes multiple passes over the data

Iterations in other systems

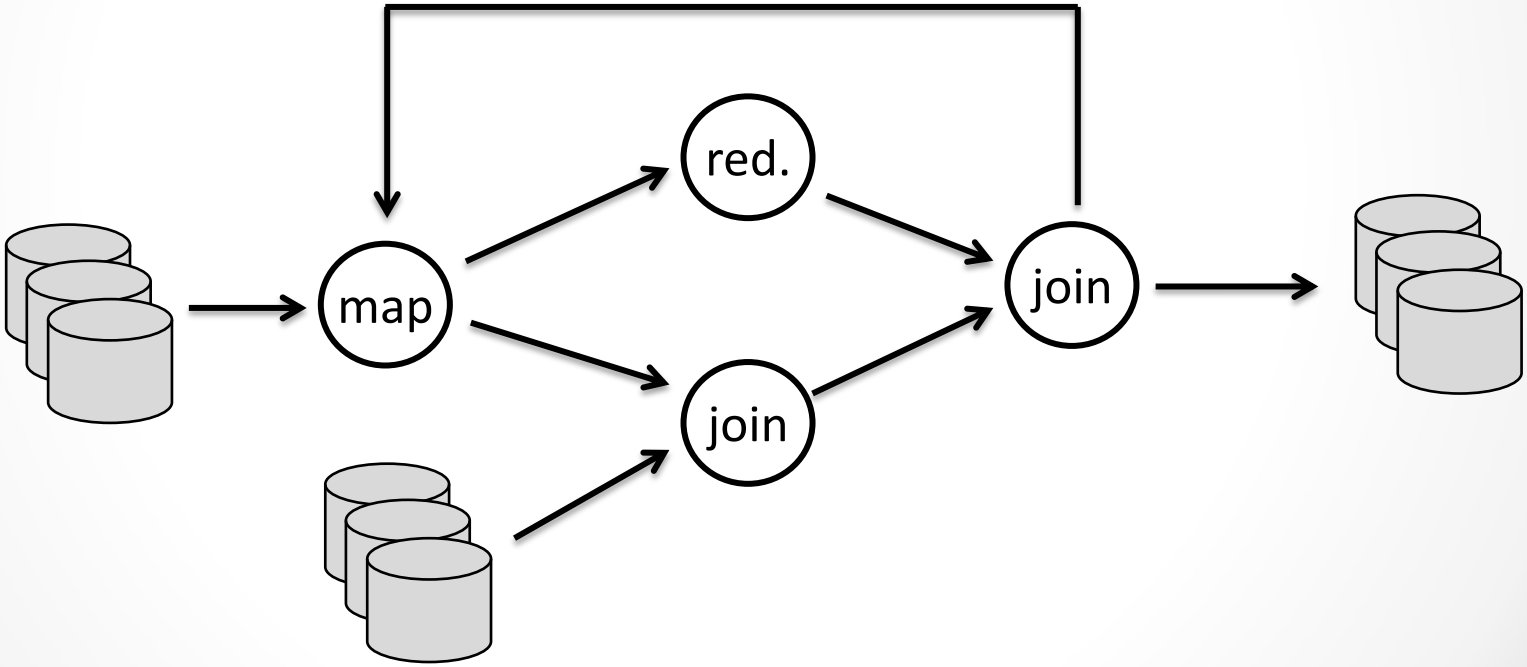




Iterations in Stratosphere

Big Data looks tiny from Stratosphere

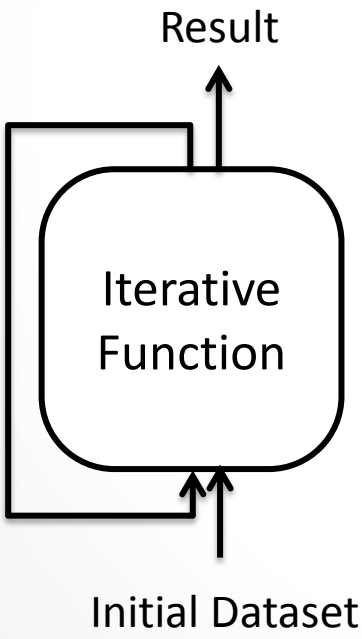
Streaming dataflow with feedback



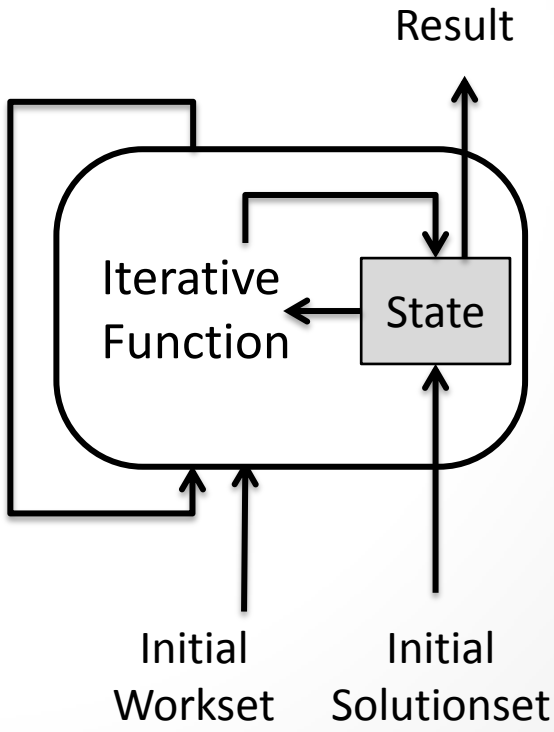
System is iteration-aware, performs automatic optimization

Stratosphere offers two types of iterations

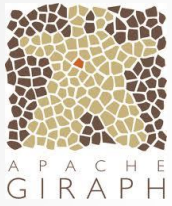
Bulk Iterations



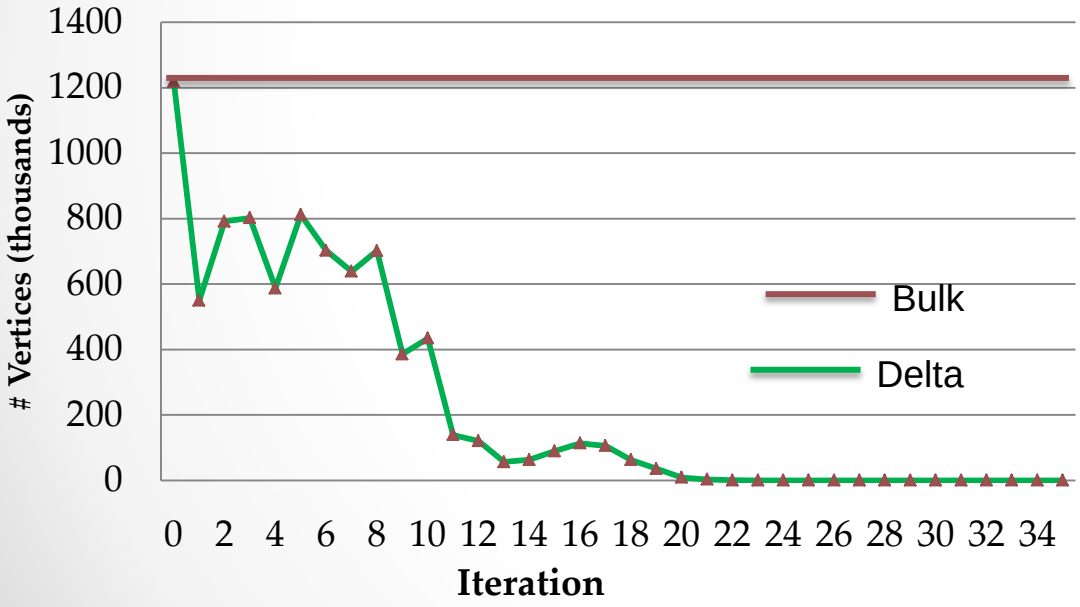
Deltalterations
(aka. Workset Iterations)



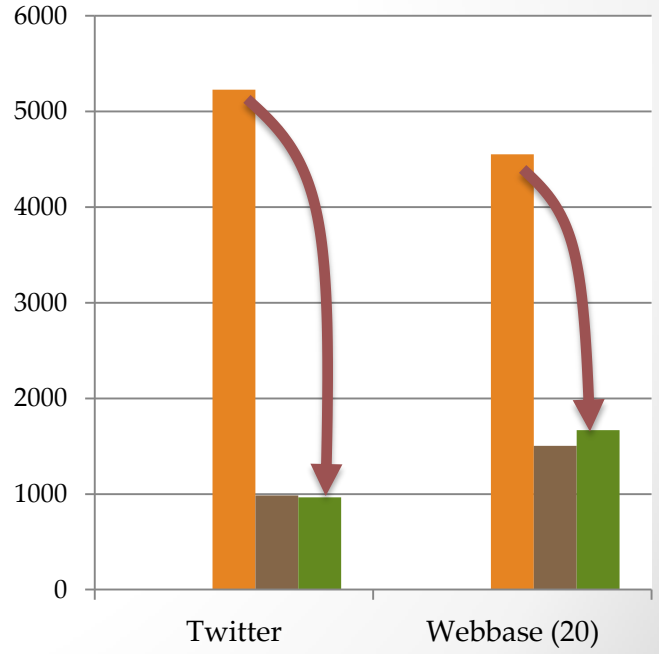
Delta Iterations speed up certain problems by a lot



Cover typical use cases of Pregel-like systems with comparable performance in a generic platform and developer API.

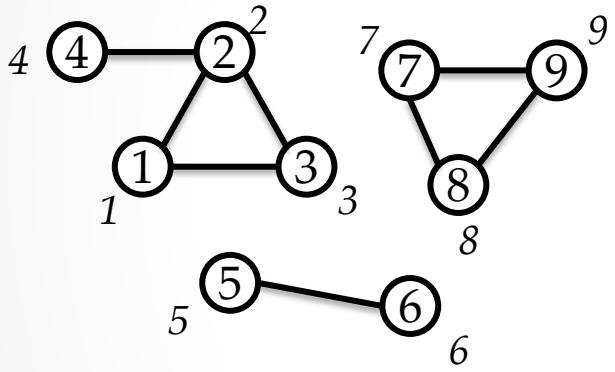


Computations performed in each iteration for connected communities of a social graph



Runtime (secs)

Workset Algorithm Illustrated



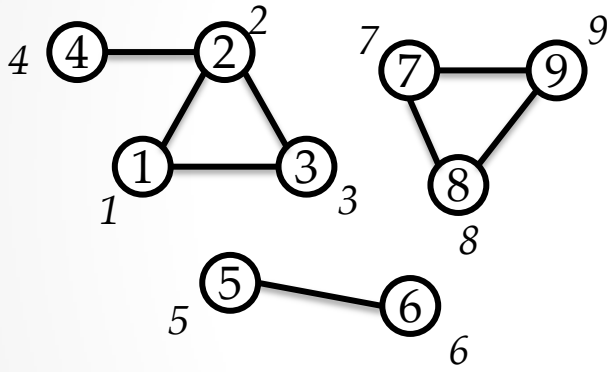
Algorithm: Find connected components of a graph.

Start: Vertices have IDs that respresent the component they belong to. Initially, every vertex has its own id (is its own component).

Step: Each vertex tells its neighbors its component id. Vertices take the min-ID of all candidates from their neighbors. A vertex that did not adopt a new ID needs not participate in the next step, as it has nothing new to tell its neighbors.

Workset Algorithm Illustrated

Solution Set



Solution Set Delta

Workset

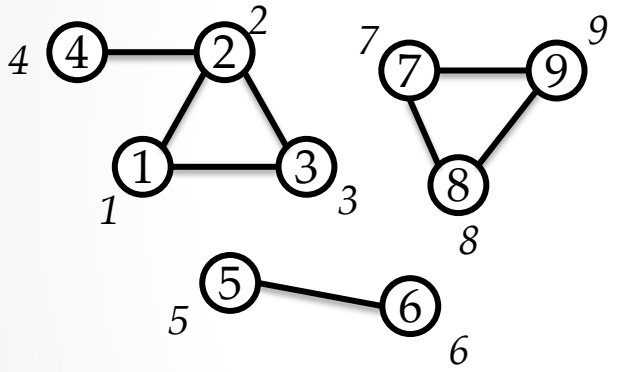
1 (2,2)	3 (1,1)	8 (7,7)
(3,3)	(2,2)	(9,9)
2 (1,1)	4 (2,2)	9 (7,7)
(3,3)		(8,8)
(4,4)	5 (6,6)	
	6 (5,5)	

Messages sent to neighbors:

1 (4, 3) means that vertex 1 receives a candidate id of 3 from vertex 4

Workset Algorithm Illustrated

Solution Set



Workset

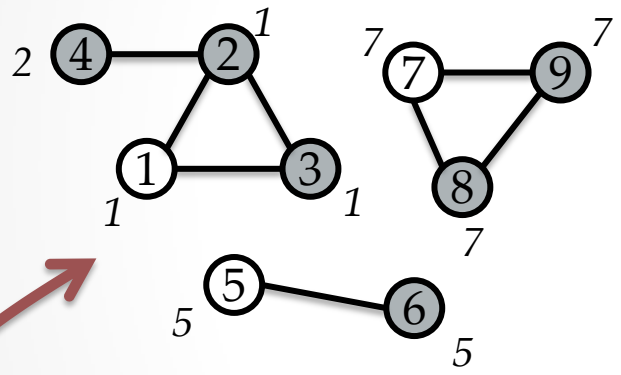
1 (2,2)	3 (1,1)	8 (7,7)
(3,3)	(2,2)	(9,9)
2 (1,1)	4 (2,2)	9 (7,7)
(3,3)		(8,8)
(4,4)	5 (6,6)	
	6 (5,5)	

Solution Set Delta

(2,1)	(6, 5)
(3,1)	(8,7)
(4,2)	(9,7)

Workset Algorithm Illustrated

Solution Set



Workset

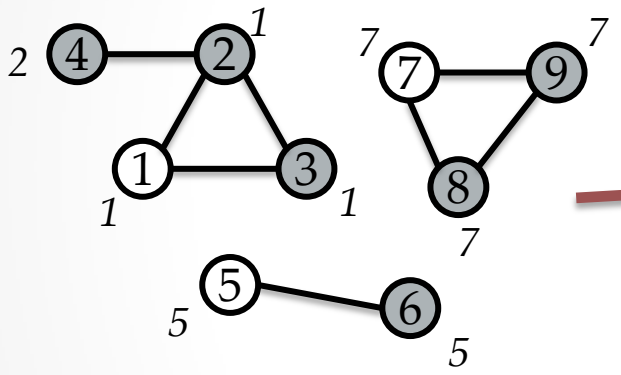
1 (2,2)	3 (1,1)	8 (7,7)
(3,3)	(2,2)	(9,9)
2 (1,1)	4 (2,2)	9 (7,7)
(3,3)		(8,8)
(4,4)	5 (6,6)	
	6 (5,5)	

Solution Set Delta

(2,1)	(6, 5)
(3,1)	(8,7)
(4,2)	(9,7)

Workset Algorithm Illustrated

Solution Set



Workset

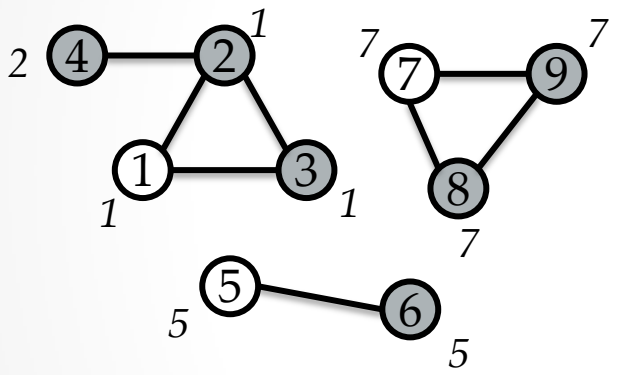
1 (2,1)	3 (2,1)	7 (8,7)
(3,1)		(9,7)
	4 (2,1)	
2 (3,1)		8 (9,7)
(4,2)	5 (6,5)	
		9 (8,7)

Solution Set Delta

(2,1)	(6, 5)
(3,1)	(8,7)
(4,2)	(9,7)

Workset Algorithm Illustrated

Solution Set



Workset

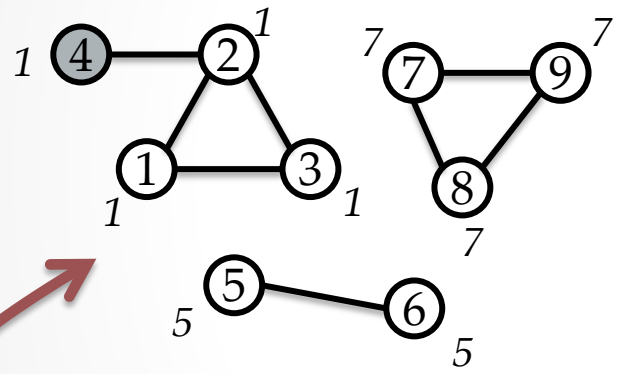
1 (2,1)	3 (2,1)	7 (8,7)
(3,1)		(9,7)
	4 (2,1)	
2 (3,1)		8 (9,7)
(4,2)	5 (6,5)	
		9 (8,7)

Solution Set Delta

(4,1)

Workset Algorithm Illustrated

Solution Set



Workset

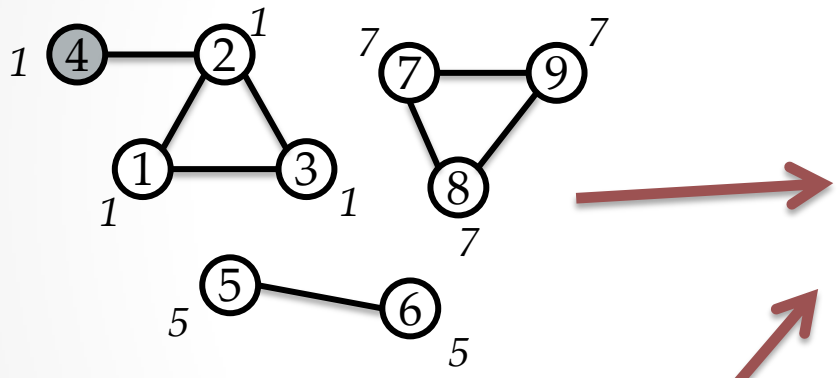
1 (2,1)	3 (2,1)	7 (8,7)
(3,1)		(9,7)
	4 (2,1)	
2 (3,1)		8 (9,7)
(4,2)	5 (6,5)	
		9 (8,7)

Solution Set Delta

(4,1)

Workset Algorithm Illustrated

Solution Set



Workset

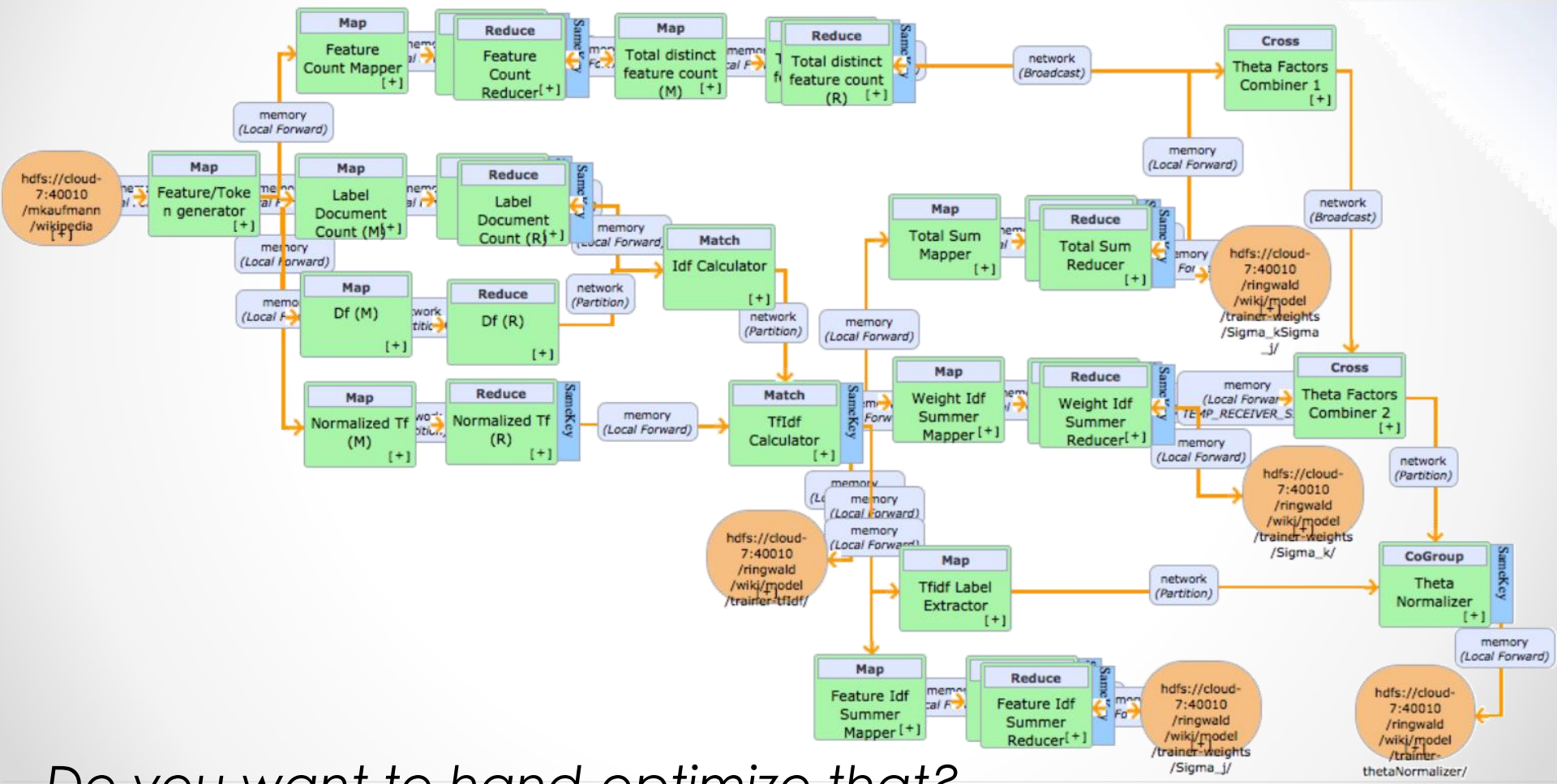
2 (4,1)

Solution Set Delta

(4,1)

Program Optimization …

Why Program Optimization ?

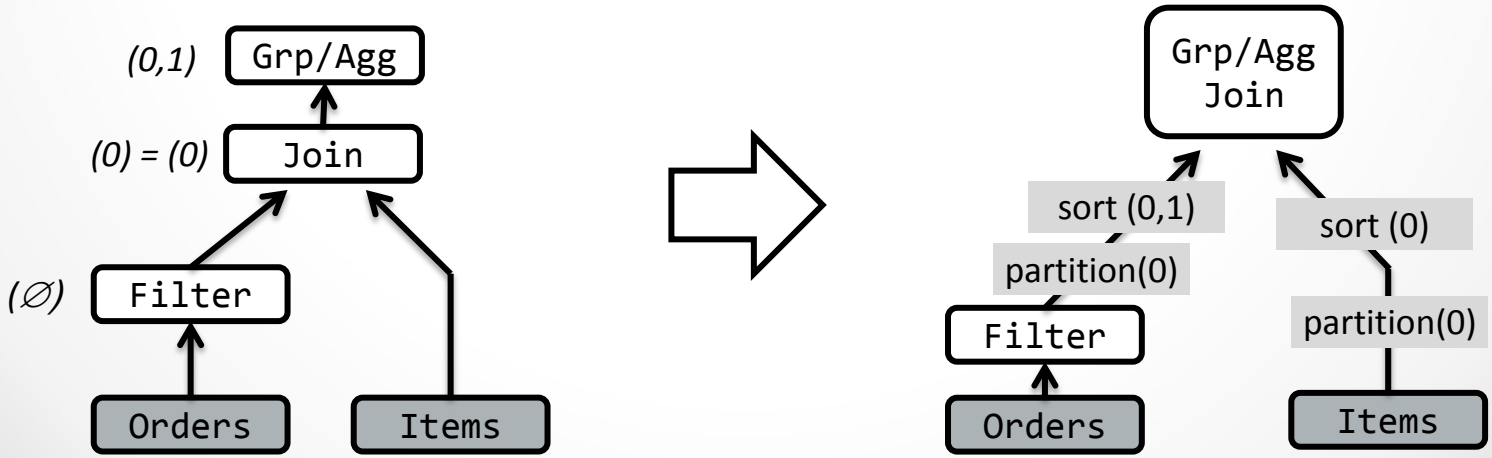


Do you want to hand-optimize that?

Optimization

```
case class Order(id: Int, priority: Int, ...)
case class Item(id: Int, price: double, )
case class PricedOrder(id, priority, price)
```

```
val orders = DataSource(...)
val items = DataSource(...)

val filtered = orders filter { ... }

val prio = filtered join items where { _.id } isEqualTo { _.id }
                    map {(o,li) => PricedOrder(o.id, o.priority, li.price)}

val sales = prio groupBy {p => (p.id, p.priority)} aggregate ({_.price},SUM)
```



Using Stratosphere

...

Its easy to get started…

trying it out…

Quickstart projects set up a program skeleton, including embedded local execution/debugging environment…

running a local pseudo-cluster

```
$ wget https://.../stratosphere-0.4.tgz
$ tar xzf stratosphere-*.tgz
$ stratosphere/bin/start-local.sh
```

Also available as a Debian package

For the experts…

If you have YARN, deploy a full stratosphere setup in 3 commands:

```
wget http://stratosphere-bin.s3-website-us-east-1
             .amazonaws.com/stratosphere-dist-0.5-SNAPSHOT-yarn.tar.gz
tar xvzf stratosphere-dist-0.5-SNAPSHOT-yarn.tar.gz
./stratosphere-yarn-0.5-SNAPSHOT/bin/yarn-session.sh -n 4 -jm 1024 -tm 3000
```

Also works on Amazon Elastic MapReduce ;-)

Download

Download the ready to run binary package. Choose the Stratosphere distribution that **matches your Hadoop version**. If you are unsure which version to choose or you just want to run locally, pick the package for Hadoop 1.2.

Hadoop 1.2 | Hadoop 2 (YARN)

Download Stratosphere for Hadoop 1.2

Start

You are almost done.

1. **Go to the download directory**,
2. **Unpack the downloaded archive**, and
3. **Start Stratosphere**.

```
$ cd ~/Downloads              # Go to download directory
$ tar xzf stratosphere-*.tgz  # Unpack the downloaded archive
$ cd stratosphere
$ bin/start-local.sh          # Start Stratosphere
```

Check the **JobManager's web frontend** at http://localhost:8081 and make sure everything is up and running.

Run Example

Run the **Word Count example** to see Stratosphere at work.

1. **Download test data:**

   ```
   $ wget -O hamlet.txt http://www.gutenberg.org/cache/epub/1787/pg1787.txt
   ```

   You now have a text file called *hamlet.txt* in your working directory.

2. **Start the example program**:

   ```
   $ bin/stratosphere run \
       --jarfile ./examples/stratosphere-java-examples-0.4-WordCount.jar \
       --arguments 1 file://`pwd`/hamlet.txt file://`pwd`/wordcount-result.txt
   ```

   You will find a file called **wordcount-result.txt** in your current directory.

Cluster Setup

Quick Start: Stratosphere K-Means Example

This guide will demonstrate Stratosphere's features by example. You will see how you can leverage Stratosphere's Iteration-feature to find clusters in a dataset using K-Means clustering. On the way, you will see the compiler, the status interface and the result of the algorithm.

Generate Input Data

Stratosphere contains a data generator for K-Means.

```
# Download Stratosphere (Development version)
wget http://stratosphere-bin.s3-website-us-east-1.amazonaws.com/stratosphere-0.5-SNAPSHOT.tgz
tar xzf stratosphere-0.5-SNAPSHOT.tgz
cd stratosphere
mkdir kmeans
cd kmeans
# run data generator
java -cp ../examples/stratosphere-java-examples-0.5-SNAPSHOT-KMeansIterative.jar eu.stratosphere.example.j
ava.record.kmeans.KMeansSampleDataGenerator 500 10 0.08
```

The generator has the following arguments:

```
KMeansDataGenerator <numberOfDataPoints> <numberOfClusterCenters> [<relative stddev>] [<centroid range>]
[<seed>]
```

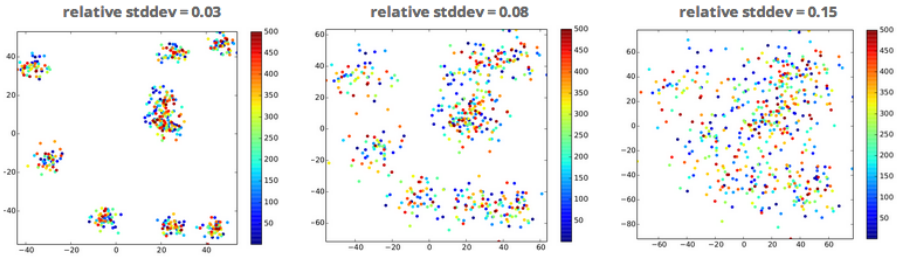
The *relative standard deviation* is an interesting tuning parameter: it determines the closeness of the points to the centers. The `kmeans/` directory should now contain two files: `centers` and `points`.

Review Input Data

Use the `plotPoints.py` tool to review the result of the data generator. Download Python Script

```
python2.7 plotPoints.py points input
```

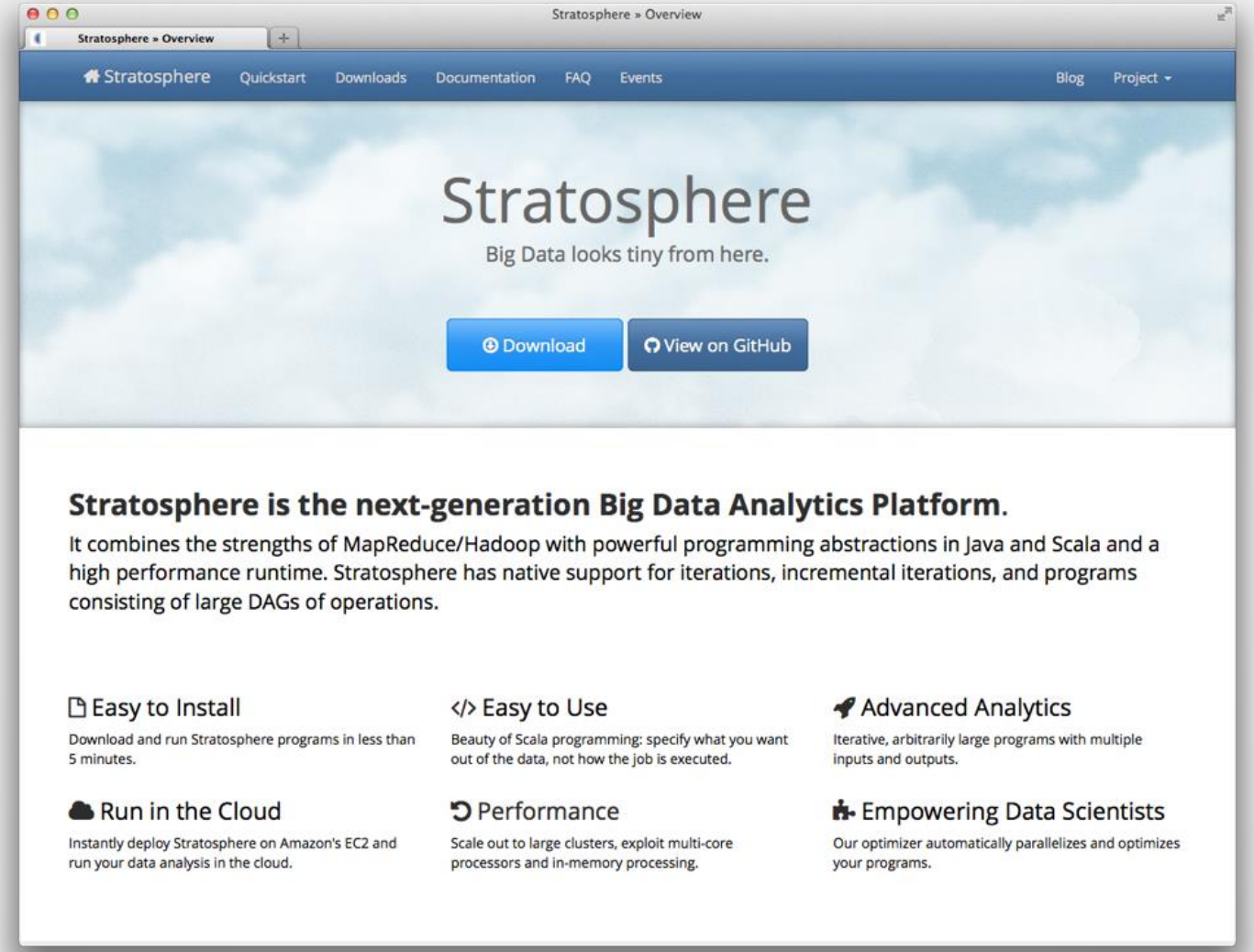
Note: You might have to install matplotlib (`python-matplotlib` package on Ubuntu) to use the Python script. The following overview presents the impact of the different standard deviations on the input data.



Run Clustering

We are using the generated input data to run the clustering using a Stratosphere job.

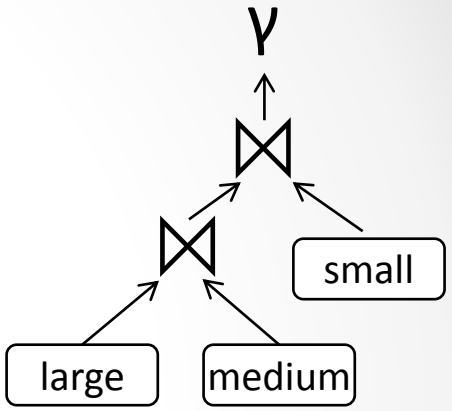
```
# go to the Stratosphere-root directory
cd stratosphere
# start Stratosphere (use ./bin/start-cluster.sh if you're on a cluster)
./bin/start-local.sh
```



www.stratosphere.eu

Backup Slides

Joins in Stratosphere




```
DataSet<Tuple...> large = env.readCsv(...);
DataSet<Tuple...> medium = env.readCsv(...);
DataSet<Tuple...> small = env.readCsv(...);

DataSet<Tuple...> joined1 = large.join(medium).where(3).equals(1)
                                 .with(new JoinFunction() { ... });

DataSet<Tuple...> joined2 = small.join(joined1).where(0).equals(2)
                                 .with(new JoinFunction() { ... });

DataSet<Tuple...> result = joined2.groupBy(3).aggregate(MAX, 2);
```

Built-in strategies include *partitioned* join and *replicated* join with local *sort-merge* or *hybrid-hash* algorithms.

Automatic Optimization

```
DataSet<Tuple...> large = env.readCsv(...);
DataSet<Tuple...> medium = env.readCsv(...);
DataSet<Tuple...> small = env.readCsv(...);

DataSet<Tuple...> joined1 = large.join(medium).where(3).equals(1)
                                 .with(new JoinFunction() { ... });

DataSet<Tuple...> joined2 = small.join(joined1).where(0).equals(2)
                                 .with(new JoinFunction() { ... });

DataSet<Tuple...> result = joined2.groupBy(3).aggregate(MAX, 2);
```

Possible execution

1) Partitioned hash-join

2) Broadcast hash-join

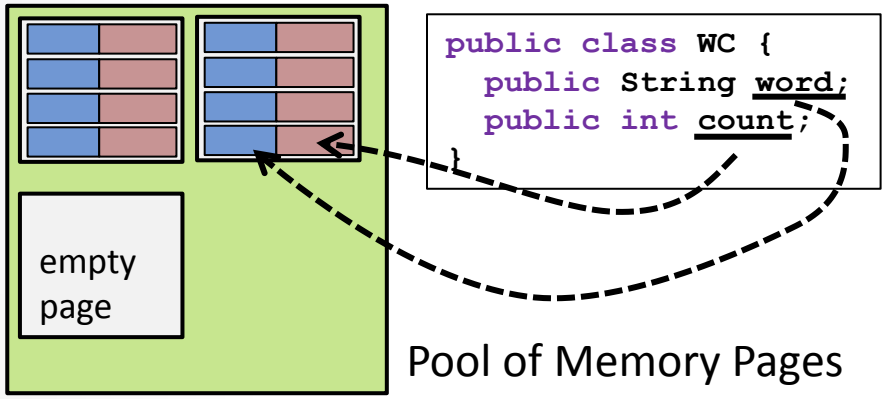
3) Grouping /Aggregation reuses the partitioning from step (1) → No shuffle!!!

Partitioned ≈ Reduce-side
Broadcast ≈ Map-side

Runtime Architecture *(comparison)*

Big Data looks tiny from Stratosphere

Spark



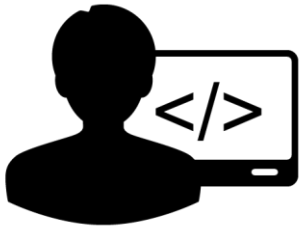
Distributed Collection

List[WC]

- Works on pages of bytes
- Maps objects transparently to these pages
- Full control over memory, out-of-core enabled
- Algorithms work on binary representation
- Address individual fields (not deserialize whole object)

- Collections of objects
- General-purpose serializer (Java / Kryo)
- Limited control over memory & less efficient spilling
- Deserialize all or nothing

What is Automatic Optimization



Hash vs. Sort
Partition vs. Broadcast
Caching
Reusing partition/sort

Execution Plan A

Execution Plan B

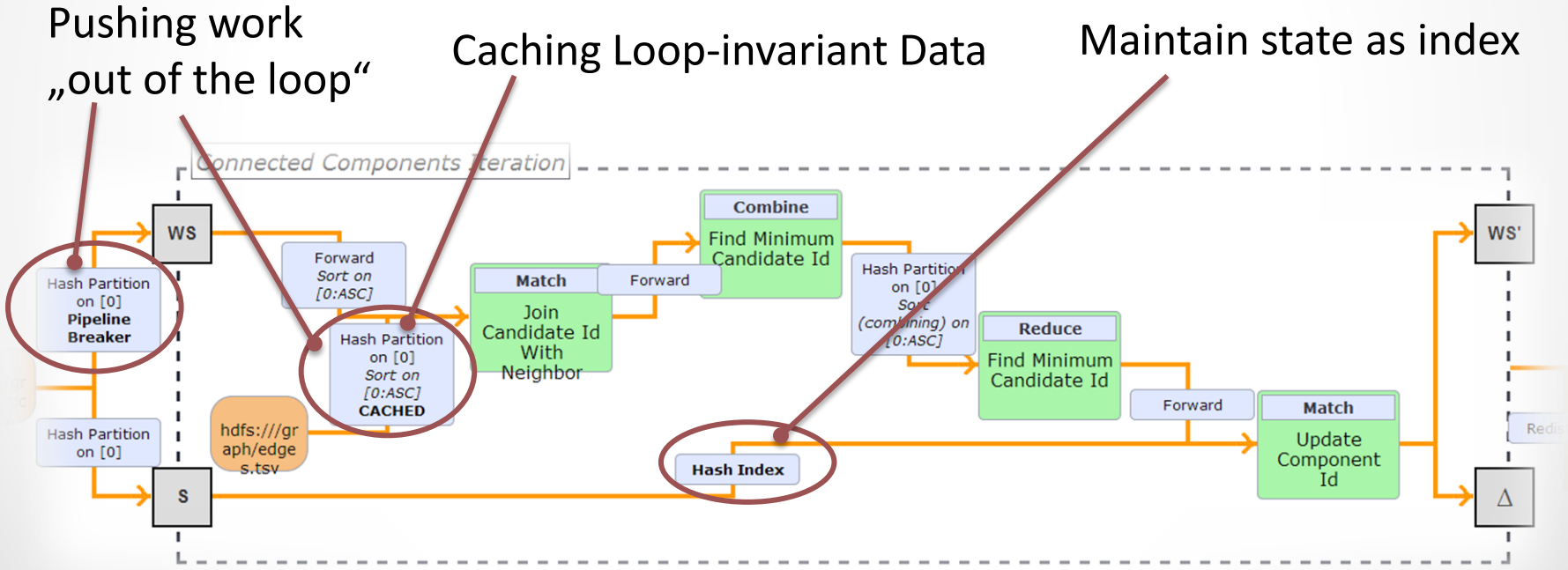
Execution Plan C

Run on a sample on the laptop

Run on large files on the cluster

Run a month later after the data evolved

Automatic Optimization for Iterative Programs



Unifies various kinds of Computations

```
ExecutionEnvironment env = getExecutionEnvironment();

DataSet<Long> vertexIds = ...
DataSet<Tuple2<Long, Long>> edges = ...

DataSet<Tuple2<Long, Long>> vertices = vertexIds.map(new
IdAssigner());

DataSet<Tuple2<Long, Long>> result = vertices .runOperation(
        VertexCentricIteration.withPlainEdges(
                edges, new CCUpdater(), new CCMessager(), 100));

result.print();
env.execute("Connected Components");
```

Pregel/Giraph-style Graph Computation

A Sample Bulk Iteration

```
// read inputs
val pages = DataSource(verticesPath, CsvInputFormat[Long]())
val edges = DataSource(edgesPath, CsvInputFormat[Edge]())

// assign initial rank
val pagesWithRank = pages map { p => PageWithRank(p, initialRank) }

// the iterative compüutation
def computeRank(ranks: DataSet[PageWithRank]) = {

    // send rank to neighbors
    val ranksForNeighbors = ranks join edges
        where { _.pageId } isEqualTo { _.from }
        map { (p, e) => (e.to, p.rank * e.transitionProbability) }

    // gather ranks per vertex and apply page rank formula
    ranksForNeighbors .groupBy { case (node, rank) => node }
                      .reduce { (a, b) => (a._1, a._2 + b._2) }
                      .map {case (node, rank) => PageWithRank(node, rank * dampening + randomJump) }
}

// invoke iteratively
val finalRanks = pagesWithRank.iterate(numIterations, computeRank)
val output = finalRanks.write(outputPath, CsvOutputFormat())
```

A Sample Delta Iteration

Connected Components of a Graph

```
def step = (s: DataSet[Vertex], ws: DataSet[Vertex]) => {

  val min = ws groupBy {_.id} reduceGroup { x => x.minBy { _.component } }

  val delta = s join minNeighbor where { _.id } isEqualTo { _.id }
              flatMap { (c,o) => if (c.component < o.component)
                                 Some(c) else None }

  val nextWs = delta join edges where {v => v.id} isEqualTo {e => e.from}
                     map { (v, e) => Vertex(e.to, v.component) }

  (delta, nextWs)
}

val components = vertices.iterateWithWorkset(initialWorkset, {_.id}, step)
```

A Sample Delta Iteration

Connected Components of a Graph

Define Step function

```
def step = (s: DataSet[Vertex], ws: DataSet[Vertex]) => {

  val min = ws groupBy {_.id} reduceGroup { x => x.minBy { _.component } }

  val delta = s join minNeighbor where { _.id } isEqualTo { _.id }
              flatMap { (c,o) => if (c.component < o.component)
                                 Some(c) else None }

  val nextWs = delta join edges where {v => v.id} isEqualTo {e => e.from}
                     map { (v, e) => Vertex(e.to, v.component) }

  (delta, nextWs)
}

val components = vertices.iterateWithWorkset(initialWorkset, {_.id}, step)
```

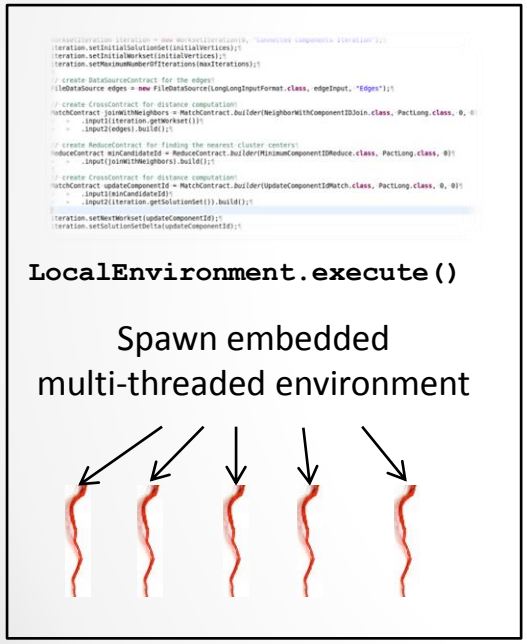
Return Delta and next Workset

Invoke Iteration

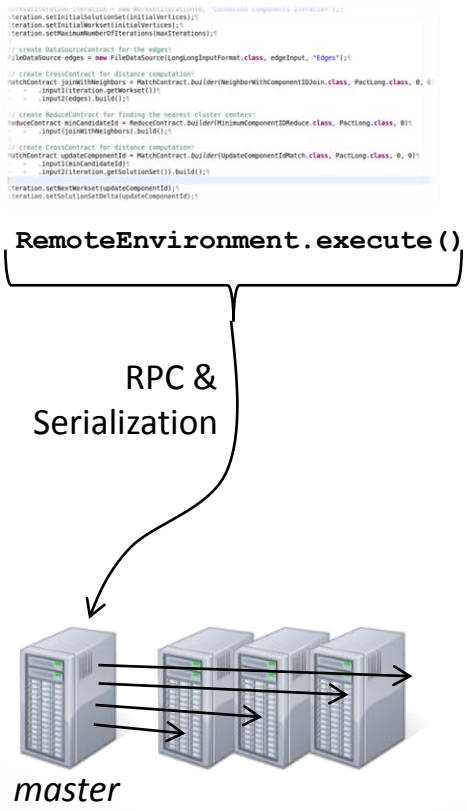
72

Running Programs

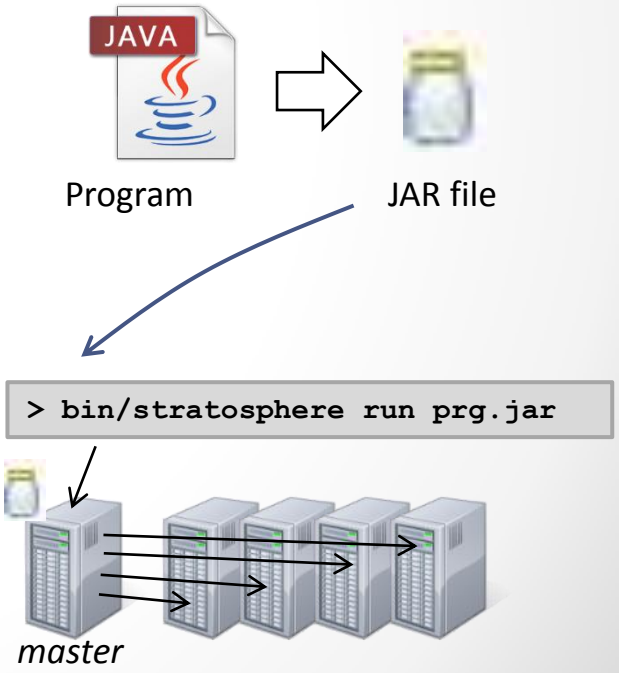
Local Environment



Remote Environment



Packaged Programs



Overview

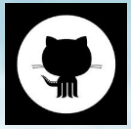
			
Paradigm	MapReduce	Iterative Data Flows	Distributed Collections (RDD)
Data Model	Writable Key/Value pairs	Java/Scala type system	Java/Scala types as key/value pairs
Runtime	Batch Parallel Sort	Streaming in-memory & out of core	Batch processing in memory
Compilation/ Optimization	none	holistic planning for data exchange, sort/hash, caching, ...	none

Data Model Stratosphere vs. Spark

Arbitrary Java Objects	Arbitrary Java Objects
Tuples as first class citizens	Key/value pairs as first class citizens
Joins / Grouping via field references *(tuple position, selector-function)* *(coming: field name)*	Joins / Grouping via Key/value pairs

"Big Data looks tiny from Stratosphere"

stratosphere.eu



github.com/stratosphere/stratosphere

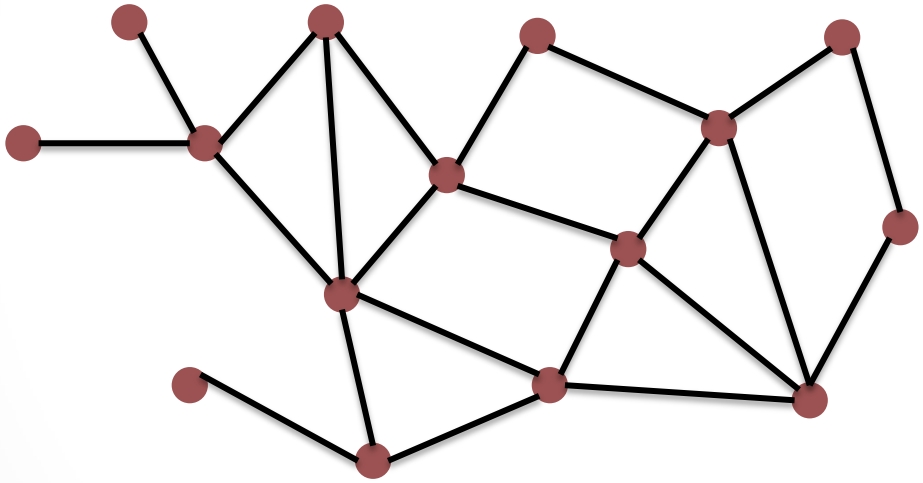
@stratosphere_eu

Simple and self contained Programming/Testing

```
ExecutionEnvironment env = getExecutionEnvironment();

DataSet<String> text = env.fromElements("To be", "or not to be",
        "or to be still", "and certainly not to be not at all", …);

DataSet<Tuple2<String, Integer>> result = text
                                .flatMap(new Tokenizer())
                                .groupBy(0).aggregate(SUM, 1);

List<Tuple2<String, Integer>> list = new ArrayList<>();
result.output(new CollectingOutput(list));
env.execute();

// validate the list contents
```

Scala API by Example

- Graph Triangles (Friend-of-a-Friend problem)
  - Recommending friends, finding important connections



- 1) Enumerate candidate triads
- 2) Close as triangles

Scala API by Example

```
case class Edge(from: Int, to: Int)
case class Triangle(apex: Int, base1: Int, base1: Int)

val vertices = DataSource("hdfs:///...", CsvFormat[Edge])

val byDegree = vertices map { projectToLowerDegree }

val byID = byDegree map { (x) => if (x.from < x.to) x
                                 else Edge(x.to, x.from) }

val triads = byDegree groupBy { _.from } reduceGroup { buildTriads }

val triangles = triads join byID
                       where { t => (t.base1, t.base2) }
                       isEqualTo { e => (e.from, e.to) }
                       map { (triangle, edge) => triangle }
```

Scala API by Example

Custom Data Types

In-situ data source

```
case class Edge(from: Int, to: Int)
case class Triangle(apex: Int, base1: Int, base1: Int)

val vertices = DataSource("hdfs:///...", CsvFormat[Edge])

val byDegree = vertices map { projectToLowerDegree }

val byID = byDegree map { (x) => if (x.from < x.to) x
                                 else Edge(x.to, x.from) }

val triads = byDegree groupBy { _.from } reduceGroup { buildTriads }

val triangles = triads join byID
                       where { t => (t.base1, t.base2) }
                       isEqualTo { e => (e.from, e.to) }
                       map { (triangle, edge) => triangle }
```

Scala API by Example

Non-relational library function

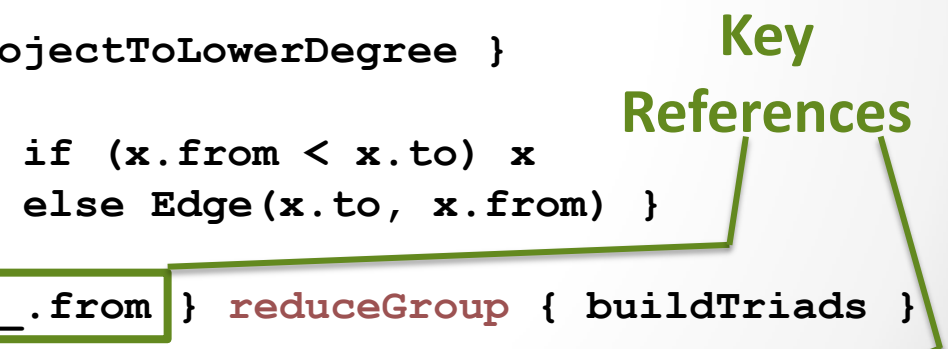
Non-relational function

```
case class Edge(from: Int, to: Int)
case class Triangle(apex: Int, base1: Int, base2: Int)

val vertices = DataSource("hdfs:///...", CsvFormat[Edge])

val byDegree = vertices map { projectToLowerDegree }

val byID = byDegree map { (x) => if (x.from < x.to) x
                                 else Edge(x.to, x.from) }

val triads = byDegree groupBy { _.from } reduceGroup { buildTriads }

val triangles = triads join byID
                where { t => (t.base1, t.base2) }
                isEqualTo { e => (e.from, e.to) }
                map { (triangle, edge) => triangle }
```

Relational Join

Scala API by Example

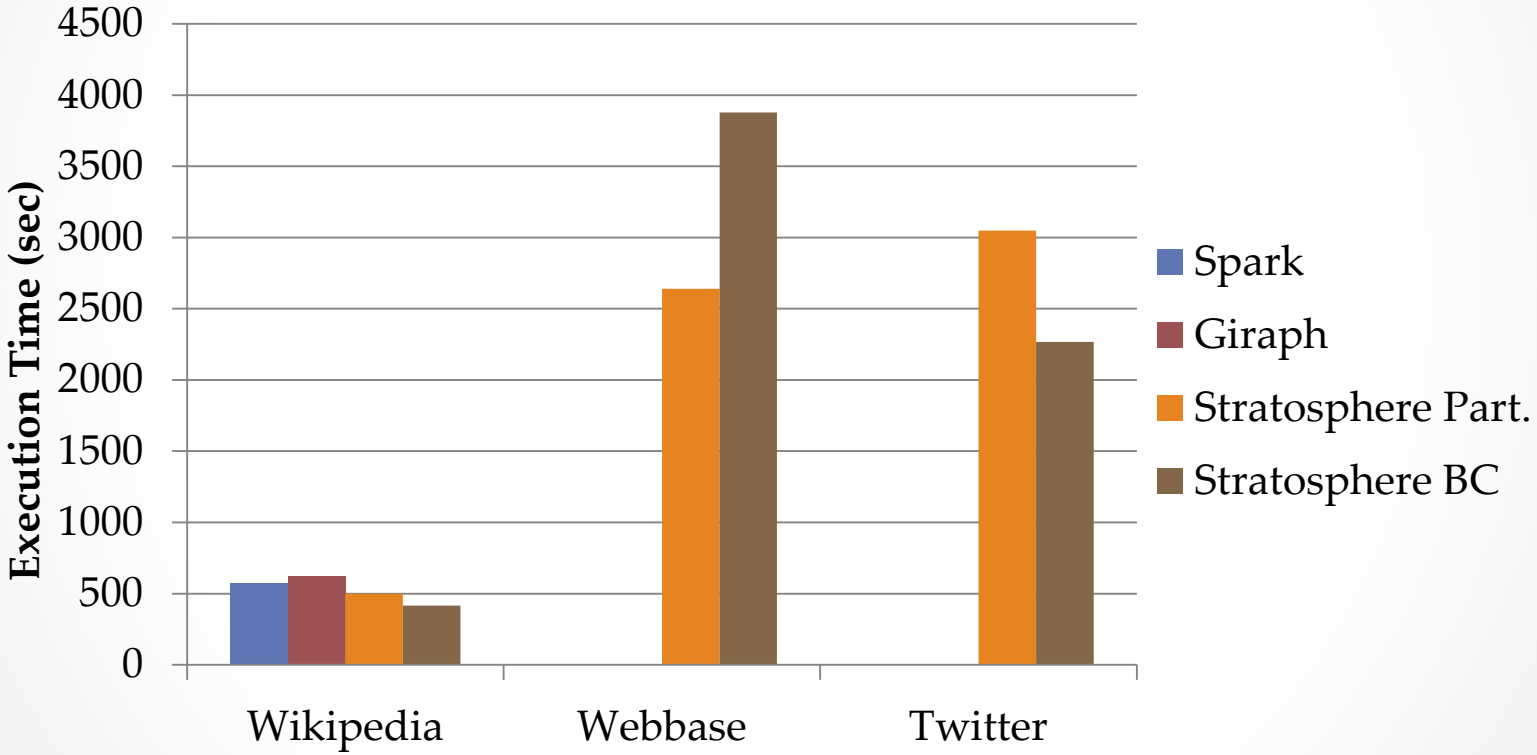
```
case class Edge(from: Int, to: Int)
case class Triangle(apex: Int, base1: Int, base2: Int)

val vertices = DataSource("hdfs:///...", CsvFormat[Edge])

val byDegree = vertices map { projectToLowerDegree }

val byID = byDegree map { (x) => if (x.from < x.to) x
                                 else Edge(x.to, x.from) }

val triads = byDegree groupBy { _.from } reduceGroup { buildTriads }

val triangles = triads join byID
                     where { t => (t.base1, t.base2) }
                     isEqualTo { e => (e.from, e.to) }
                     map { (triangle, edge) => triangle }
```



Iterative Program (Java)

```
WorksetIteration iteration = new WorksetIteration(0, "Connected Components Iteration");
iteration.setInitialSolutionSet(initialVertices);
iteration.setInitialWorkset(initialVertices);
iteration.setMaximumNumberOfIterations(maxIterations);

// create DataSourceContract for the edges
FileDataSource edges = new FileDataSource(LongLongInputFormat.class, edgeInput, "Edges");

// create CrossContract for distance computation
MatchContract joinWithNeighbors = MatchContract.builder(NeighborWithComponentIDJoin.class, PactLong.class, 0, 0)
    .input1(iteration.getWorkset())
    .input2(edges).build();

// create ReduceContract for finding the nearest cluster centers
ReduceContract minCandidateId = ReduceContract.builder(MinimumComponentIDReduce.class, PactLong.class, 0)
    .input(joinWithNeighbors).build();

// create CrossContract for distance computation
MatchContract updateComponentId = MatchContract.builder(UpdateComponentIdMatch.class, PactLong.class, 0, 0)
    .input1(minCandidateId)
    .input2(iteration.getSolutionSet()).build();

iteration.setNextWorkset(updateComponentId);
iteration.setSolutionSetDelta(updateComponentId);
```

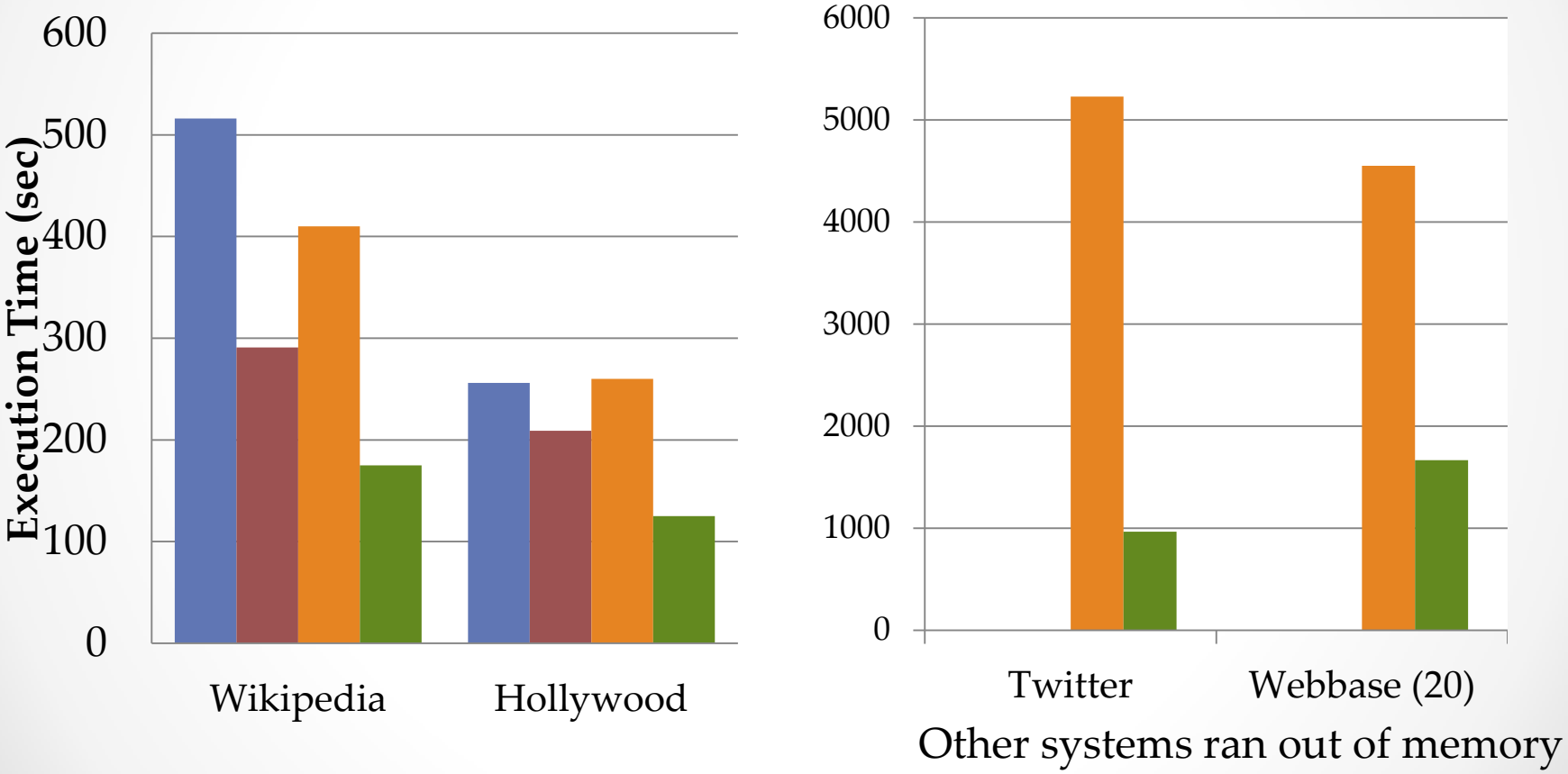
Bulk Iteration Performance



Other systems ran out of memory

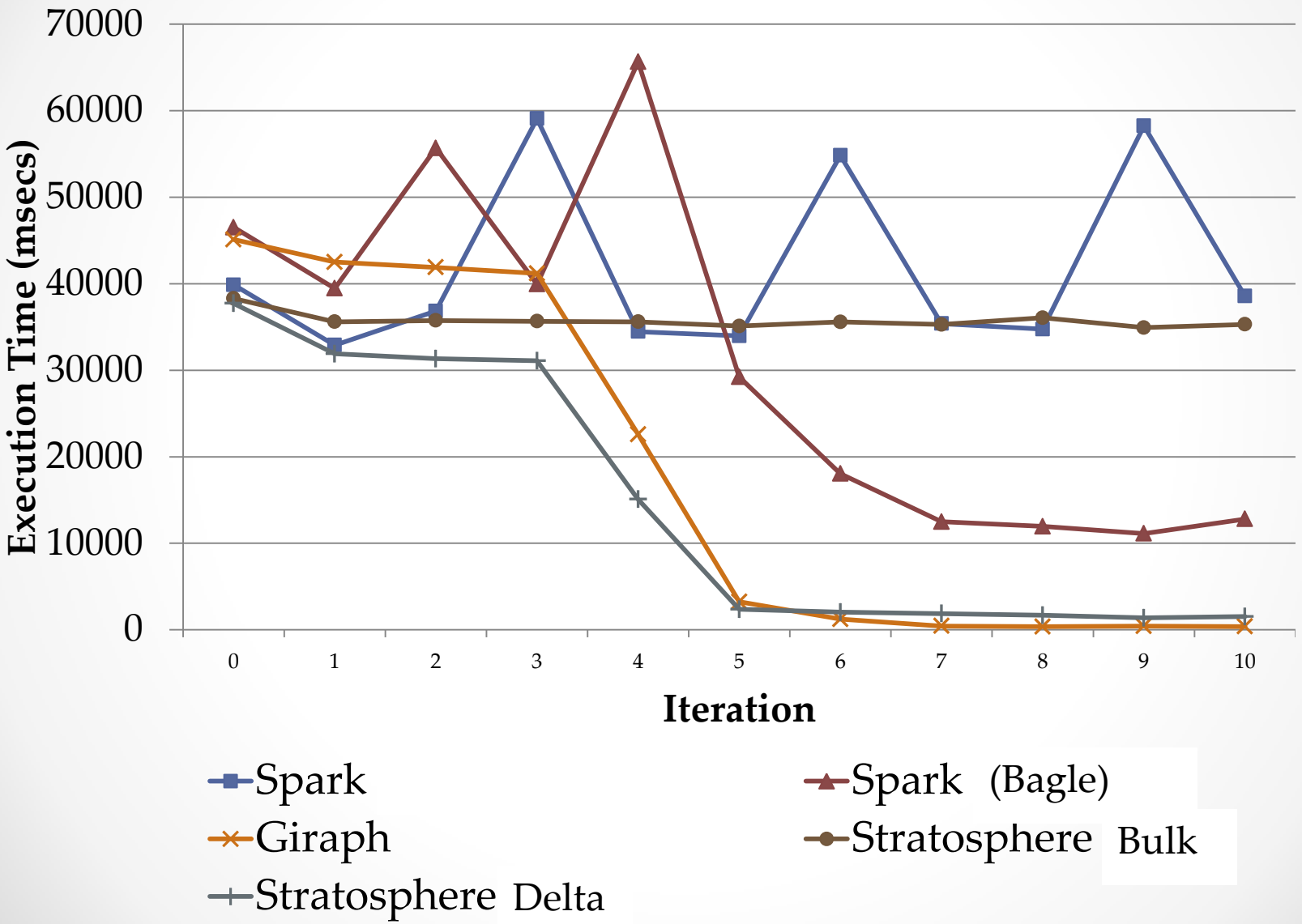
Cf. VLDB 2012 Paper "Spinning Fast Iterative Data Flows"

Delta Iteration Performance



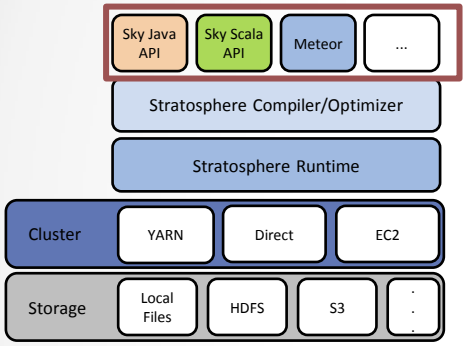
Cf. VLDB 2012 Paper "Spinning Fast Iterative Data Flows"

Per-Iteration Times



Cf. VLDB 2012 Paper "Spinning Fast Iterative Data Flows"

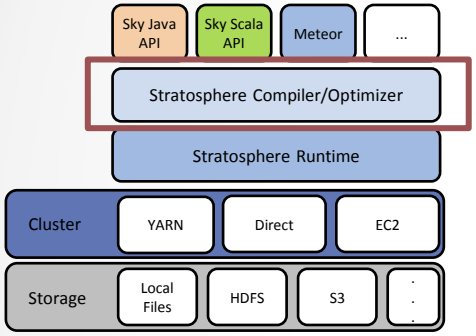
Architecture: Front-Ends



Stratosphere Front-Ends

- Multiple Font-Ends for different target audiences
- Supports operations from both shallow analytics (SQL, Hadoop MapReduce) and deep analytics (Machine Learning, Data Mining)
- Supports *custom user-defined operations* as required to support the diverse Big Data use cases
- API may be used to create connectors to other application tools (visualization, dashboards, …)
- Shared compiler/optimizer allows easy creation of new front-ends (unlike Hadoop eco-system).

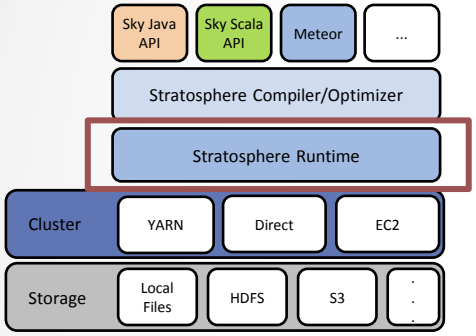
Architecture: Compiler



Stratosphere Compiler

- Inspired by Relational Database Optimizer (optimizes SQL, enables efficient complex queries)
- Extended to non-relational use cases through *code analysis* techniques
- Eliminates costly and time-intensive manual tuning of analysis tasks to the data, automatically adapts programs when the data characteristics change
- Extended to iterative algorithms, optimizes machine learning algorithms and subsumes specialized systems
- *Code generation* techniques efficiently support *custom operations* in the system

Architecture: Runtime



Stratosphere Runtime

- Hybrid between a *Parallel Database* and a *MapReduce engine.*
- Supports both database operations, and custom operation defined by users for specialized use cases

- Streaming Engine → Fast, low latency queries

- Support for *stateful multi-pass algorithms*
  → Very efficient for machine learning and graph analysis algorithms

- Heavily in-memory → Fast on modern computers

- Out-of-core capabilities → Scales beyond main memory

Algebra

Operator	Semantics	Description
S	Bag / Set	a collection of elements
S.filter(p)	$\{ x \mid x \in S, p(x) \}$	*p* evaluates to Boolean
S.map(f)	$\{ f(x) \mid x \in S \}$	*f* returns a single value
S.flatMap(f)	$\{ f(x) \mid x \in S \}$	*f* returns a bag / set of values
S cross T	$\{ (x, y) \mid x \in S, y \in Y \}$	all pairs (Cartesian product)
S join(θ) T	$\{ (x, y) \mid x \in S, y \in Y, \theta(x, y) \}$	theta-join, cross + filter
S join(s, t) T	$\{ (x, y) \mid x \in S, y \in Y, s(x) = t(y) \}$	equi-join , *s*, *t* select the join key

Algebra

Operator	Semantics	Description
S union T	$S \cup T$ / $S \uplus T$	bag / set union
S minus T	$S - T$ / $S \dot{-} T$	bag / set difference
S.groupBy(k)	$\{ \{ y \mid k(y) = k(x), y \in S \} \mid x \in S \}$	*k* selects grp. key, result is a set
S.fold(f,e)	(x::xs).fold(f,e) = f(x) :: xs.fold(f,e) ({}).fold(f,e) = e	spine transformer (str. recursion) mold for one pass aggregations