732A54/TDDE31 Big Data Analytics

Lecture 9: Machine Learning with Spark

Jose M. Peña
IDA, Linköping University, Sweden

Contents

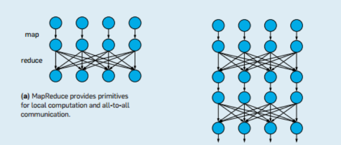
Literature

- Main sources
  - Zaharia, M. et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, 15-28, 2012.
  - Meng, X. et al. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research*, 17(34):1-7, 2016.
- Additional sources
  - Zaharia, M. et al. Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM*, 59(11):56-65, 2016.
  - Spark programming guide available at `https://spark.apache.org/docs/latest/rdd-programming-guide.html`
  - MLlib manual available at `http://spark.apache.org/docs/latest/ml-guide.html`
  - Slides for 732A99/TDDE01 Machine Learning.

Spark Framework

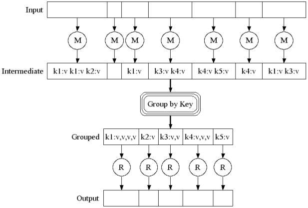
- Recall from the previous lecture that MapReduce can emulate any distributed computation, since this can be divided into a sequence of MapReduce calls.

Figure 10. Emulating an arbitrary distributed computation with MapReduce.



(a) MapReduce provides primitives for local computation and all-to-all communication.

(b) By chaining these steps together, we can emulate any distributed computation. The main costs for this emulation are the latency of the rounds and the overhead of passing state across steps.



Spark Framework

- Recall from the previous lecture that MapReduce can emulate any distributed computation, since this can be divided into a sequence of MapReduce calls.

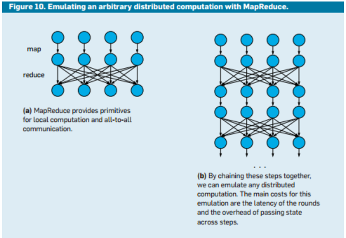
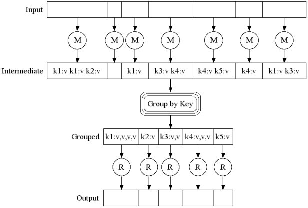


Figure 10. Emulating an arbitrary distributed computation with MapReduce.

(a) MapReduce provides primitives for local computation and all-to-all communication.

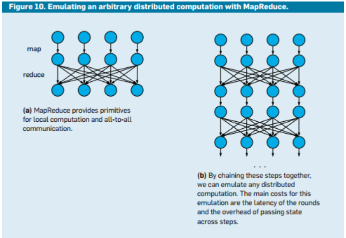
(b) By chaining these steps together, we can emulate any distributed computation. The main costs for this emulation are the latency of the rounds and the overhead of passing state across steps.

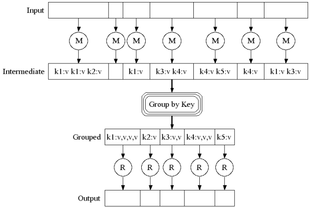


- However, the emulation may be **inefficient** since the message exchange relies on external storage, e.g. disk.

Spark Framework

- Recall from the previous lecture that MapReduce can emulate any distributed computation, since this can be divided into a sequence of MapReduce calls.





- However, the emulation may be **inefficient** since the message exchange relies on external storage, e.g. disk.
- This is a problem for iterative machine learning algorithms. **Even worse**: Each iteration (i.e., MapReduce call) loads the data anew from disk.

Spark Framework

- Recall from the previous lecture that MapReduce can emulate any distributed computation, since this can be divided into a sequence of MapReduce calls.

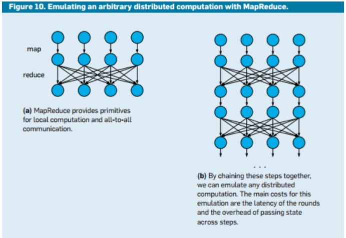
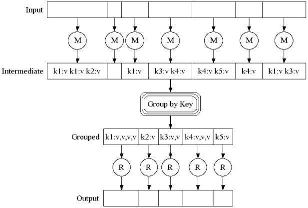


Figure 10. Emulating an arbitrary distributed computation with MapReduce.

(a) MapReduce provides primitives for local computation and all-to-all communication.

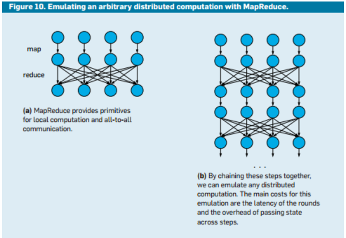
(b) By chaining these steps together, we can emulate any distributed computation. The main costs for this emulation are the latency of the rounds and the overhead of passing state across steps.

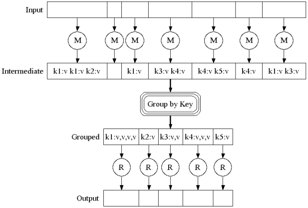


- However, the emulation may be **inefficient** since the message exchange relies on external storage, e.g. disk.
- This is a problem for iterative machine learning algorithms. **Even worse**: Each iteration (i.e., MapReduce call) loads the data anew from disk.
- Apache Spark is a framework to process large amounts of data by parallelizing computations across a cluster of nodes.

Spark Framework

- Recall from the previous lecture that MapReduce can emulate any distributed computation, since this can be divided into a sequence of MapReduce calls.

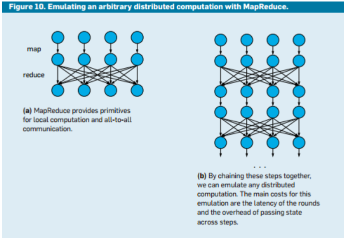


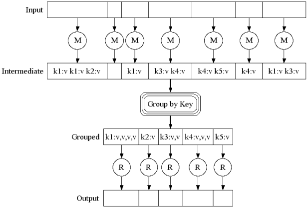


- However, the emulation may be **inefficient** since the message exchange relies on external storage, e.g. disk.
- This is a problem for iterative machine learning algorithms. **Even worse**: Each iteration (i.e., MapReduce call) loads the data anew from disk.
- Apache Spark is a framework to process large amounts of data by parallelizing computations across a cluster of nodes.
- It builds on MapReduce's ability to emulate any distributed computation but it makes it more **efficiently** by emulating in-memory data sharing across MapReduce calls.

Spark Framework

- Recall from the previous lecture that MapReduce can emulate any distributed computation, since this can be divided into a sequence of MapReduce calls.





- However, the emulation may be **inefficient** since the message exchange relies on external storage, e.g. disk.
- This is a problem for iterative machine learning algorithms. **Even worse**: Each iteration (i.e., MapReduce call) loads the data anew from disk.
- Apache Spark is a framework to process large amounts of data by parallelizing computations across a cluster of nodes.
- It builds on MapReduce's ability to emulate any distributed computation but it makes it more **efficiently** by emulating in-memory data sharing across MapReduce calls.
- It includes MLlib, a library for machine learning that uses linear algebra libraries on each node.

Spark Framework

- Data sharing is achieved via resilient distributed datasets (RDDs).

Spark Framework

- Data sharing is achieved via resilient distributed datasets (RDDs).
- RDD is a read-only, partitioned collection of records that can only be **defined** through transformations applied to external storage or to other RDDs.

Spark Framework

- Data sharing is achieved via resilient distributed datasets (RDDs).
- RDD is a read-only, partitioned collection of records that can only be **defined** through transformations applied to external storage or to other RDDs.

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] $\Rightarrow$ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] $\Rightarrow$ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) $\Rightarrow$ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) $\Rightarrow$ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
Actions	$count()$	:	RDD[T] $\Rightarrow$ Long
	$collect()$	:	RDD[T] $\Rightarrow$ Seq[T]
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] $\Rightarrow$ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] $\Rightarrow$ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Spark Framework

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$sample(fraction : \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$union()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
	$join()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$cogroup()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$crossProduct()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
Actions	$count()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Long}$
	$collect()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Seq[T]}$
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{T}$
	$lookup(k : \mathrm{K})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{Seq[V]}$ (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

- Note that some transformations and actions do not require RDDs of (key, value) pairs, i.e. so-called pair RDDs.
- Note also that the transformations and actions for non-pair RDDs work on pair RDDs too.

Spark Framework

- Data sharing is achieved via resilient distributed datasets (RDDs).
- RDD is a read-only, partitioned collection of records that can only be **defined** through transformations applied to external storage or to other RDDs.

Transformations	*map*(f : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(f : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(f : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(f : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(p : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(f : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(k : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Spark Framework

- Data sharing is achieved via resilient distributed datasets (RDDs).
- RDD is a read-only, partitioned collection of records that can only be **defined** through transformations applied to external storage or to other RDDs.

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] ⇒ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] ⇒ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] ⇒ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	$count()$	:	RDD[T] ⇒ Long
	$collect()$	:	RDD[T] ⇒ Seq[T]
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] ⇒ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

- The sequence of transformations that defines a RDD is called its lineage. It is used to rebuild it in case of failure, i.e. there is no data replication unlike in MapReduce.

Spark Framework

- Data sharing is achieved via resilient distributed datasets (RDDs).
- RDD is a read-only, partitioned collection of records that can only be **defined** through transformations applied to external storage or to other RDDs.

Transformations	*map*(f : T ⇒ U)	: RDD[T] ⇒ RDD[U]
	filter(f : T ⇒ Bool)	: RDD[T] ⇒ RDD[T]
	flatMap(f : T ⇒ Seq[U])	: RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	: RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	: RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f : (V, V) ⇒ V)	: RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	: (RDD[T], RDD[T]) ⇒ RDD[T]
	join()	: (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	: (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	: (RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(f : V ⇒ W)	: RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K])	: RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(p : Partitioner[K])	: RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	: RDD[T] ⇒ Long
	collect()	: RDD[T] ⇒ Seq[T]
	reduce(f : (T, T) ⇒ T)	: RDD[T] ⇒ T
	lookup(k : K)	: RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	: Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

- The sequence of transformations that defines a RDD is called its lineage. It is used to rebuild it in case of failure, i.e. there is no data replication unlike in MapReduce.
- RDDs are **created** only when an action is executed. Why ? E.g., [read + filter] more memory efficient than [read, filter].

Spark Framework

- Data sharing is achieved via resilient distributed datasets (RDDs).
- RDD is a read-only, partitioned collection of records that can only be **defined** through transformations applied to external storage or to other RDDs.

Transformations	*map*(f : T ⇒ U)	: RDD[T] ⇒ RDD[U]
	filter(f : T ⇒ Bool)	: RDD[T] ⇒ RDD[T]
	flatMap(f : T ⇒ Seq[U])	: RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	: RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	: RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f : (V, V) ⇒ V)	: RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	: (RDD[T], RDD[T]) ⇒ RDD[T]
	join()	: (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	: (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	: (RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(f : V ⇒ W)	: RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K])	: RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(p : Partitioner[K])	: RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	: RDD[T] ⇒ Long
	collect()	: RDD[T] ⇒ Seq[T]
	reduce(f : (T, T) ⇒ T)	: RDD[T] ⇒ T
	lookup(k : K)	: RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	: Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

- The sequence of transformations that defines a RDD is called its lineage. It is used to rebuild it in case of failure, i.e. there is no data replication unlike in MapReduce.
- RDDs are **created** only when an action is executed. Why ? E.g., [read + filter] more memory efficient than [read, filter].
- Actually, RDDs are **created each time an action is executed, unless the user persist them** in memory and/or disk.

Spark Framework

- Data sharing is achieved via resilient distributed datasets (RDDs).
- RDD is a read-only, partitioned collection of records that can only be **defined** through transformations applied to external storage or to other RDDs.

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	: RDD[T] $\Rightarrow$ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	: RDD[T] $\Rightarrow$ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	: RDD[T] $\Rightarrow$ RDD[U]
	$sample(fraction : \mathrm{Float})$	: RDD[T] $\Rightarrow$ RDD[T] (Deterministic sampling)
	$groupByKey()$	: RDD[(K, V)] $\Rightarrow$ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	: RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$union()$	: (RDD[T], RDD[T]) $\Rightarrow$ RDD[T]
	$join()$	: (RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (V, W))]
	$cogroup()$	: (RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	: (RDD[T], RDD[U]) $\Rightarrow$ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	: RDD[(K, V)] $\Rightarrow$ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	: RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	: RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
Actions	$count()$	: RDD[T] $\Rightarrow$ Long
	$collect()$	: RDD[T] $\Rightarrow$ Seq[T]
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	: RDD[T] $\Rightarrow$ T
	$lookup(k : \mathrm{K})$	: RDD[(K, V)] $\Rightarrow$ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	: Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

- The sequence of transformations that defines a RDD is called its lineage. It is used to rebuild it in case of failure, i.e. there is no data replication unlike in MapReduce.
- RDDs are **created** only when an action is executed. Why ? E.g., [read + filter] more memory efficient than [read, filter].
- Actually, RDDs are **created each time an action is executed, unless the user persist them** in memory and/or disk.
- Actions write to disk or return values to the master/driver.

Spark Framework

- Example in Scala to find error lines in a log file:

```
1.lines=spark.textFile("hdfs://...")
2.errors=lines.filter(_.startsWith("ERROR"))
3.errors.persist() //Store in memory
4.errors.count() //Materialize
5.errors.filter(_.contains("HDFS")).map(_.split('\t')(3)).collect()
```

Spark Framework

- Example in Scala to find error lines in a log file:

```
1.lines=spark.textFile("hdfs://...")
2.errors=lines.filter(_.startsWith("ERROR"))
3.errors.persist() //Store in memory
4.errors.count() //Materialize
5.errors.filter(_.contains("HDFS")).map(_.split('\t')(3)).collect()
```

- Note that:
  - Line 3 indicates to store the error lines **in memory**. Note `persist()` = `persist(MEMORY_ONLY)` = `cache()` $\neq$ `persist(MEMORY_AND_DISK)` $\neq \ldots$
  - However, this does not happen until line 4, when the RDDs are computed.

Spark Framework

- Example in Scala to find error lines in a log file:

```
1.lines=spark.textFile("hdfs://...")
2.errors=lines.filter(_.startsWith("ERROR"))
3.errors.persist() //Store in memory
4.errors.count() //Materialize
5.errors.filter(_.contains("HDFS")).map(_.split('\t')(3)).collect()
```

- Note that:
  - Line 3 indicates to store the error lines **in memory**. Note `persist()` = `persist(MEMORY_ONLY)` = `cache()` $\neq$ `persist(MEMORY_AND_DISK)` $\neq \ldots$
  - However, this does not happen until line 4, when the RDDs are computed.
  - The rest of the RDDs (e.g., `lines`) are discarded after being used.

Spark Framework

- Example in Scala to find error lines in a log file:

```
1.lines=spark.textFile("hdfs://...")
2.errors=lines.filter(_.startsWith("ERROR"))
3.errors.persist() //Store in memory
4.errors.count() //Materialize
5.errors.filter(_.contains("HDFS")).map(_.split('\t')(3)).collect()
```

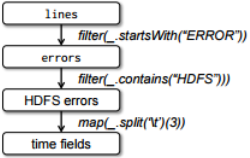
- Note that:
  - Line 3 indicates to store the error lines **in memory**. Note `persist()` = `persist(MEMORY_ONLY)` = `cache()` $\neq$ `persist(MEMORY_AND_DISK)` $\neq \ldots$
  - However, this does not happen until line 4, when the RDDs are computed.
  - The rest of the RDDs (e.g., `lines`) are discarded after being used.
  - Line 5 does not access disk because the data are **in memory**.

Spark Framework

- Example in Scala to find error lines in a log file:

```
1.lines=spark.textFile("hdfs://...")
2.errors=lines.filter(_.startsWith("ERROR"))
3.errors.persist() //Store in memory
4.errors.count() //Materialize
5.errors.filter(_.contains("HDFS")).map(_.split('\t')(3)).collect()
```

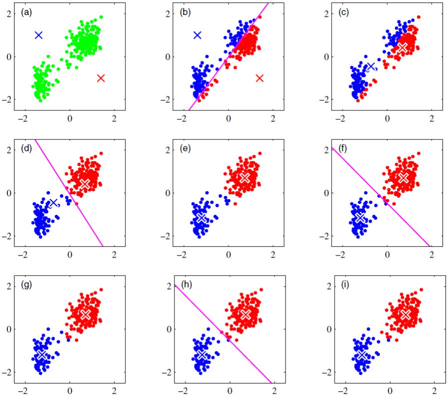
- Note that:
  - Line 3 indicates to store the error lines **in memory**. Note `persist()` = `persist(MEMORY_ONLY)` = `cache()` ≠ `persist(MEMORY_AND_DISK)` ≠ ...
  - However, this does not happen until line 4, when the RDDs are computed.
  - The rest of the RDDs (e.g., `lines`) are discarded after being used.
  - Line 5 does not access disk because the data are **in memory**.
  - If any partition of the in-memory data has gone lost, it can be rebuilt with the help of the lineage graph.



Machine Learning with Spark: *K*-Means

- Consider data clustering (a.k.a. unsupervised learning) via the K-means algorithm.

```
Assign each point to a cluster (a.k.a. subpopulation) at random
Compute the cluster centroids as the averages of the points assigned to each cluster
Repeat until the centroids do not change
    Assign each point to the cluster with the closest centroid
    Update the cluster centroids as the averages of the points assigned to each cluster
```



Machine Learning with Spark: K-Means

- K-Means in Python (`data` has been **persisted** when read from file):

```python
def closestPoint(p, centers):
    bestIndex = 0
    closest = float("+inf")
    for i in range(len(centers)):
        tempDist = np.sum((p - centers[i]) ** 2)
        if tempDist < closest:
            closest = tempDist
            bestIndex = i
    return bestIndex
kPoints = data.takeSample(False, K, 1)
tempDist = 1.0

while tempDist > convergeDist:
    closest = data.map(
        lambda p: (closestPoint(p, kPoints), (p, 1)))
    pointStats = closest.reduceByKey(
        lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
    newPoints = pointStats.map(
        lambda st: (st[0], st[1][0] / st[1][1])).collect()

    tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)

    for (iK, p) in newPoints:
        kPoints[iK] = p

print("Final centers: " + str(kPoints))
```

Machine Learning with Spark: Logistic Regression

- Consider a binary classification problem, i.e. $t \in \{-1, +1\}$. Then,

$$p(t = +1|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|t = +1)p(t = +1)}{p(\boldsymbol{x}|t = +1)p(t = +1) + p(\boldsymbol{x}|t = -1)p(t = -1)} = \sigma(s(\boldsymbol{x}))$$

where $s(\boldsymbol{x}) = \log \frac{p(\boldsymbol{x}|t=+1)p(t=+1)}{p(\boldsymbol{x}|t=-1)p(t=-1)}$, and $\sigma(a) = \frac{1}{1+exp(-a)}$ is called logistic sigmoid function.

Machine Learning with Spark: Logistic Regression

- Consider a binary classification problem, i.e. $t \in \{-1, +1\}$. Then,

$$p(t = +1|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|t = +1)p(t = +1)}{p(\boldsymbol{x}|t = +1)p(t = +1) + p(\boldsymbol{x}|t = -1)p(t = -1)} = \sigma(s(\boldsymbol{x}))$$

  where $s(\boldsymbol{x}) = \log \frac{p(\boldsymbol{x}|t=+1)p(t=+1)}{p(\boldsymbol{x}|t=-1)p(t=-1)}$, and $\sigma(a) = \frac{1}{1+exp(-a)}$ is called logistic sigmoid function.
- We assume that $p(\boldsymbol{x}|t)$ is a member of the exponential family with equal scale parameter (e.g. Gaussian with equal covariance matrix, multinomial), which implies that $s(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$. The model $y(\boldsymbol{x}) = p(t = +1|\boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{x})$ is called logistic regression.

Machine Learning with Spark: Logistic Regression

- Consider a binary classification problem, i.e. $t \in \{-1, +1\}$. Then,

$$p(t = +1|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|t = +1)p(t = +1)}{p(\boldsymbol{x}|t = +1)p(t = +1) + p(\boldsymbol{x}|t = -1)p(t = -1)} = \sigma(s(\boldsymbol{x}))$$

  where $s(\boldsymbol{x}) = \log \frac{p(\boldsymbol{x}|t=+1)p(t=+1)}{p(\boldsymbol{x}|t=-1)p(t=-1)}$, and $\sigma(a) = \frac{1}{1+exp(-a)}$ is called logistic sigmoid function.
- We assume that $p(\boldsymbol{x}|t)$ is a member of the exponential family with equal scale parameter (e.g. Gaussian with equal covariance matrix, multinomial), which implies that $s(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$. The model $y(\boldsymbol{x}) = p(t = +1|\boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{x})$ is called logistic regression.
- Note that

$$p(t = +1|\boldsymbol{x}) = \sigma(s(\boldsymbol{x})) = \frac{1}{1 + \exp(-\boldsymbol{w}^T\boldsymbol{x})}$$

$$p(t = -1|\boldsymbol{x}) = 1 - p(t = +1|\boldsymbol{x}) = 1 - \sigma(s(\boldsymbol{x})) = \sigma(-s(\boldsymbol{x})) = \frac{1}{1 + \exp(\boldsymbol{w}^T\boldsymbol{x})}$$

Machine Learning with Spark: Logistic Regression

- Consider a binary classification problem, i.e. $t \in \{-1, +1\}$. Then,

$$p(t = +1|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|t = +1)p(t = +1)}{p(\boldsymbol{x}|t = +1)p(t = +1) + p(\boldsymbol{x}|t = -1)p(t = -1)} = \sigma(s(\boldsymbol{x}))$$

  where $s(\boldsymbol{x}) = \log \frac{p(\boldsymbol{x}|t=+1)p(t=+1)}{p(\boldsymbol{x}|t=-1)p(t=-1)}$, and $\sigma(a) = \frac{1}{1+exp(-a)}$ is called logistic sigmoid function.
- We assume that $p(\boldsymbol{x}|t)$ is a member of the exponential family with equal scale parameter (e.g. Gaussian with equal covariance matrix, multinomial), which implies that $s(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$. The model $y(\boldsymbol{x}) = p(t = +1|\boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{x})$ is called logistic regression.
- Note that

$$p(t = +1|\boldsymbol{x}) = \sigma(s(\boldsymbol{x})) = \frac{1}{1 + \exp(-\boldsymbol{w}^T\boldsymbol{x})}$$

$$p(t = -1|\boldsymbol{x}) = 1 - p(t = +1|\boldsymbol{x}) = 1 - \sigma(s(\boldsymbol{x})) = \sigma(-s(\boldsymbol{x})) = \frac{1}{1 + \exp(\boldsymbol{w}^T\boldsymbol{x})}$$

  and thus

$$p(t_n|\boldsymbol{x}_n) = \frac{1}{1 + \exp(-t_n\boldsymbol{w}^T\boldsymbol{x}_n)}.$$

Machine Learning with Spark: Logistic Regression

- Consider a binary classification problem, i.e. $t \in \{-1, +1\}$. Then,

$$p(t = +1|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|t = +1)p(t = +1)}{p(\boldsymbol{x}|t = +1)p(t = +1) + p(\boldsymbol{x}|t = -1)p(t = -1)} = \sigma(s(\boldsymbol{x}))$$

  where $s(\boldsymbol{x}) = \log \frac{p(\boldsymbol{x}|t=+1)p(t=+1)}{p(\boldsymbol{x}|t=-1)p(t=-1)}$, and $\sigma(a) = \frac{1}{1+exp(-a)}$ is called logistic sigmoid function.
- We assume that $p(\boldsymbol{x}|t)$ is a member of the exponential family with equal scale parameter (e.g. Gaussian with equal covariance matrix, multinomial), which implies that $s(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$. The model $y(\boldsymbol{x}) = p(t = +1|\boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{x})$ is called logistic regression.
- Note that

$$p(t = +1|\boldsymbol{x}) = \sigma(s(\boldsymbol{x})) = \frac{1}{1 + \exp(-\boldsymbol{w}^T\boldsymbol{x})}$$

$$p(t = -1|\boldsymbol{x}) = 1 - p(t = +1|\boldsymbol{x}) = 1 - \sigma(s(\boldsymbol{x})) = \sigma(-s(\boldsymbol{x})) = \frac{1}{1 + \exp(\boldsymbol{w}^T\boldsymbol{x})}$$

  and thus

$$p(t_n|\boldsymbol{x}_n) = \frac{1}{1 + \exp(-t_n\boldsymbol{w}^T\boldsymbol{x}_n)}.$$

- We determine the parameters $\boldsymbol{w}$ by minimizing the negative log-likelihood:

$$L(\boldsymbol{w}) = -\sum_n \log p(t_n|\boldsymbol{x}_n) = \sum_n \log(1 + \exp(-t_n\boldsymbol{w}^T\boldsymbol{x}_n))$$

Machine Learning with Spark: Logistic Regression

- Consider a binary classification problem, i.e. $t \in \{-1, +1\}$. Then,

$$p(t=+1|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|t=+1)p(t=+1)}{p(\boldsymbol{x}|t=+1)p(t=+1) + p(\boldsymbol{x}|t=-1)p(t=-1)} = \sigma(s(\boldsymbol{x}))$$

  where $s(\boldsymbol{x}) = \log \frac{p(\boldsymbol{x}|t=+1)p(t=+1)}{p(\boldsymbol{x}|t=-1)p(t=-1)}$, and $\sigma(a) = \frac{1}{1+exp(-a)}$ is called logistic sigmoid function.
- We assume that $p(\boldsymbol{x}|t)$ is a member of the exponential family with equal scale parameter (e.g. Gaussian with equal covariance matrix, multinomial), which implies that $s(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$. The model $y(\boldsymbol{x}) = p(t=+1|\boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{x})$ is called logistic regression.
- Note that

$$p(t=+1|\boldsymbol{x}) = \sigma(s(\boldsymbol{x})) = \frac{1}{1+\exp(-\boldsymbol{w}^T\boldsymbol{x})}$$

$$p(t=-1|\boldsymbol{x}) = 1 - p(t=+1|\boldsymbol{x}) = 1 - \sigma(s(\boldsymbol{x})) = \sigma(-s(\boldsymbol{x})) = \frac{1}{1+\exp(\boldsymbol{w}^T\boldsymbol{x})}$$

  and thus

$$p(t_n|\boldsymbol{x}_n) = \frac{1}{1+\exp(-t_n\boldsymbol{w}^T\boldsymbol{x}_n)}.$$

- We determine the parameters $\boldsymbol{w}$ by minimizing the negative log-likelihood:

$$L(\boldsymbol{w}) = -\sum_n \log p(t_n|\boldsymbol{x}_n) = \sum_n \log(1+\exp(-t_n\boldsymbol{w}^T\boldsymbol{x}_n))$$

  whose gradient is $-\sum_n t_n(1 - 1/(1+\exp(-t_n\boldsymbol{w}^T\boldsymbol{x}_n)))\boldsymbol{x}_n$.

Machine Learning with Spark: Logistic Regression

- Logistic regression in Scala (note the use of **persist**, map and reduce):

```
val points = spark.textFile(...)
                 .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

Machine Learning with Spark: Logistic Regression

- Logistic regression in Scala (note the use of **persist**, map and reduce):

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

- Logistic regression in Python (`points` was **persisted** when read from file):

```
# Initialize w to a random value
w = 2 * np.random.ranf(size=D) - 1
print("Initial w: " + str(w))

# Compute logistic regression gradient for a matrix of data points
def gradient(matrix, w):
    Y = matrix[:, 0]    # point labels (first column of input file)
    X = matrix[:, 1:]   # point coordinates
    # For each point (x, y), compute gradient function, then sum these up
    return ((1.0 / (1.0 + np.exp(-Y * X.dot(w))) - 1.0) * Y * X.T).sum(1)

def add(x, y):
    x += y
    return x

for i in range(iterations):
    print("On iteration %i" % (i + 1))
    w -= points.map(lambda m: gradient(m, w)).reduce(add)

print("Final w: " + str(w))
```

Machine Learning with Spark: MLlib

- Many machine learning methods are already implemented in MLlib, i.e. the user does not need to specify the transformations and actions.
- Logistic regression in Python:

```
lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)
lrModel = lr.fit(training)
```

- SVMs in Python:

```
model = SVMWithSGD.train(parsedData, iterations=100)
```

- NNs in Python:

```
layers = [4, 5, 4, 3]
trainer = MultilayerPerceptronClassifier(maxIter=100, layers=layers,
blockSize=128, seed=1234)
model = trainer.fit(train)
```

- MMs in Python:

```
gmm = GaussianMixture().setK(2)
model = gmm.fit(dataset)
```

- K-Means in Python:

```
kmeans = KMeans().setK(2).setSeed(1)
model = kmeans.fit(dataset)
```

Machine Learning with Spark: Experiments

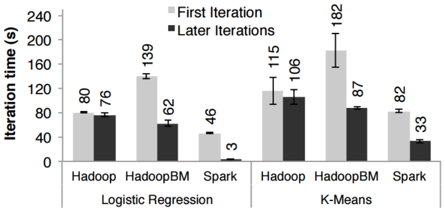
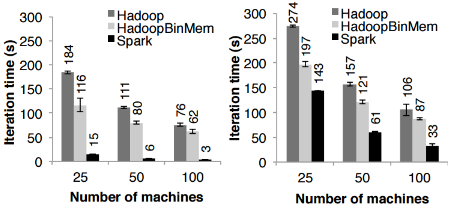


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.



(a) Logistic Regression (b) K-Means

Figure 8: Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.

Lab with Spark

- Implement a kernel model to predict the hourly temperatures for a date and place in Sweden. To do so, you are provided with the files `stations.csv` and `temps.csv`. These files contain information about weather stations and temperature measurements for the stations at different days and times. The data have been kindly provided by the Swedish Meteorological and Hydrological Institute (SMHI) and processed by Zlatan Dragisic.
- You are asked to provide a temperature forecast for a date and place in Sweden. The forecast should consist of the predicted temperatures from 4 am to 24 pm in an interval of 2 hours. Use a kernel that is the sum of three Gaussian kernels:
  - The first to account for the distance from a station to the point of interest.
  - The second to account for the distance between the day a temperature measurement was made and the day of interest.
  - The third to account for the distance between the hour of the day a temperature measurement was made and the hour of interest.
- Repeat the exercise about multiplying instead of summing the three kernels above.

Lab with Spark

- Consider regressing an unidimensional continuous random variable Y on a D-dimensional continuous random variable $\boldsymbol{X}$.

Lab with Spark

- Consider regressing an unidimensional continuous random variable Y on a D-dimensional continuous random variable $\boldsymbol{X}$.
- The best regression function under the squared error loss function is $y^*(\boldsymbol{x}) = \mathbb{E}_Y[y|\boldsymbol{x}]$.

Lab with Spark

- Consider regressing an unidimensional continuous random variable Y on a D-dimensional continuous random variable $\boldsymbol{X}$.
- The best regression function under the squared error loss function is $y^*(\boldsymbol{x}) = \mathbb{E}_Y[y|\boldsymbol{x}]$.
- Since $\boldsymbol{x}$ may not appear in the finite training set $\{(\boldsymbol{x}_n, y_n)\}$ available, then we output a weighted average over all the training points. That is

$$y(\boldsymbol{x}) = \frac{\sum_n k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right) y_n}{\sum_n k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right)}$$

Lab with Spark

- Consider regressing an unidimensional continuous random variable Y on a D-dimensional continuous random variable $\boldsymbol{X}$.
- The best regression function under the squared error loss function is $y^*(\boldsymbol{x}) = \mathbb{E}_Y[y|\boldsymbol{x}]$.
- Since $\boldsymbol{x}$ may not appear in the finite training set $\{(\boldsymbol{x}_n, y_n)\}$ available, then we output a weighted average over all the training points. That is

$$y(\boldsymbol{x}) = \frac{\sum_n k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right) y_n}{\sum_n k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right)}$$

  where $k : \mathbb{R}^D \to \mathbb{R}$ is a kernel function, which is usually non-negative and monotone decreasing along rays starting from the origin. The parameter h is called smoothing factor or width.

Lab with Spark

- Consider regressing an unidimensional continuous random variable Y on a D-dimensional continuous random variable $\boldsymbol{X}$.
- The best regression function under the squared error loss function is $y^*(\boldsymbol{x}) = \mathbb{E}_Y[y|\boldsymbol{x}]$.
- Since $\boldsymbol{x}$ may not appear in the finite training set $\{(\boldsymbol{x}_n, y_n)\}$ available, then we output a weighted average over all the training points. That is

$$y(\boldsymbol{x}) = \frac{\sum_n k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right) y_n}{\sum_n k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right)}$$

where $k : \mathbb{R}^D \to \mathbb{R}$ is a kernel function, which is usually non-negative and monotone decreasing along rays starting from the origin. The parameter h is called smoothing factor or width.

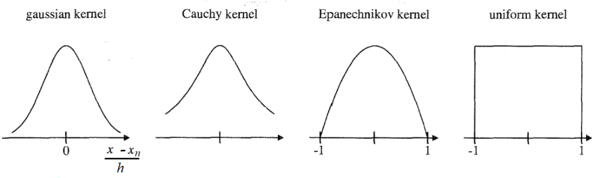
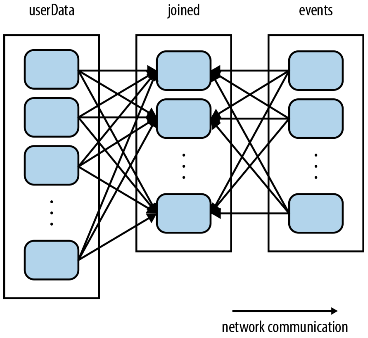


FIGURE 10.3. *Various kernels on $\mathcal{R}$.*

- Gaussian kernel: $k(u) = exp(-||u||^2)$ where $||\cdot||$ is the Euclidean norm.

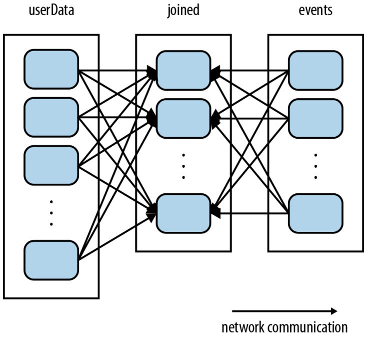
Lab with Spark

- Bear in mind that a join operation may trigger a shuffle operation, which is time and memory consuming.



Lab with Spark

- Bear in mind that a join operation may trigger a shuffle operation, which is time and memory consuming.



- Instead, broadcast one of the RDDs to join, if small. This sends a copy of the RDD to each node, and the join can be performed locally (or even skipped).

```
rdd = rdd.collectAsMap()
bc = sc.broadcast(rdd)
bc.value[i]
```

Summary

- Spark is a framework to process large datasets by parallelizing computations.
- It is particularly suitable for iterative distributed computations, since data can be store in memory.
- It includes MLlib, a machine learning library.