Evaluation in Spark: Unlike Scala Collections!

Big Data Analysis with Scala and Spark

Heather Miller

Why is Spark Good for Data Science?

Why is Spark Good for Data Science?

Let's start by recapping some major themes from previous sessions:

- We learned the difference between transformations and actions.
  - **Transformations:** Deferred/lazy
  - **Actions:** Eager, kick off staged transformations.
- We learned that latency makes a big difference; too much latency wastes the time of the data analyst.
  - **In-memory computation:** Significantly lower latencies (several orders of magnitude!)

Why is Spark Good for Data Science?

Let’s start by recapping some major themes from previous sessions:

- We learned the difference between transformations and actions.
  - **Transformations:** Deferred/lazy
  - **Actions:** Eager, kick off staged transformations.
- We learned that latency makes a big difference; too much latency wastes the time of the data analyst.
  - **In-memory computation:** Significantly lower latencies (several orders of magnitude!)

Why do you think Spark is good for data science?

Why is Spark Good for Data Science?

Let's start by recapping some major themes from previous sessions:

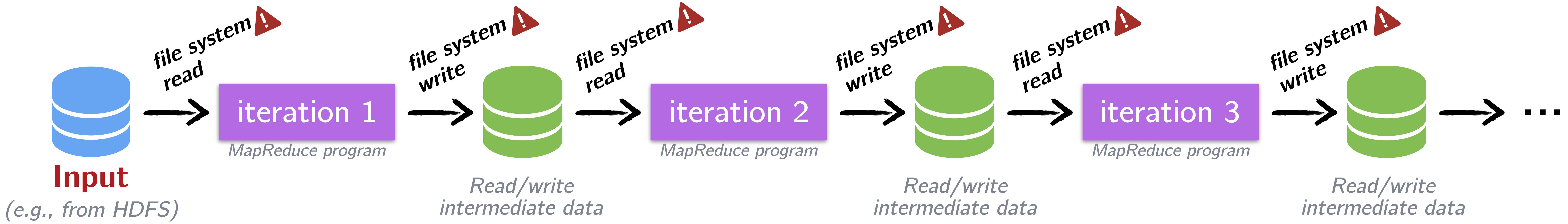
- We learned the difference between transformations and actions.
  - **Transformations:** Deferred/lazy
  - **Actions:** Eager, kick off staged transformations.
- We learned that latency makes a big difference; too much latency wastes the time of the data analyst.
  - **In-memory computation:** Significantly lower latencies (several orders of magnitude!)

Why do you think Spark is good for data science?

Hint: Most data science problems involve iteration.

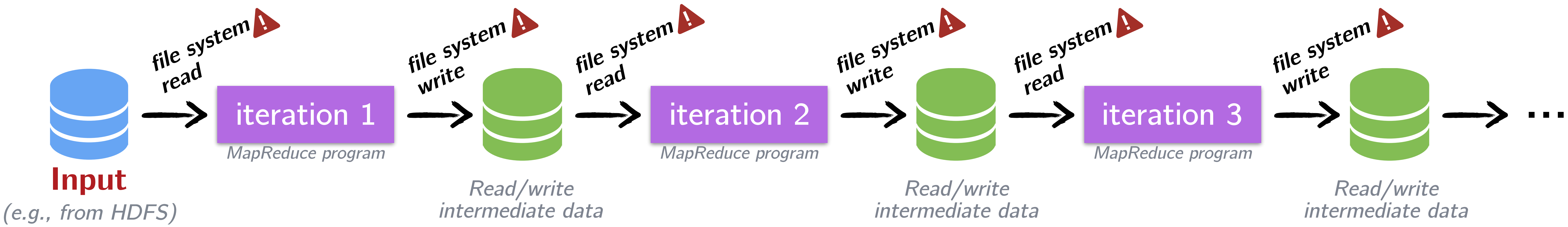
Iteration and Big Data Processing

Iteration in Hadoop:



Iteration and Big Data Processing

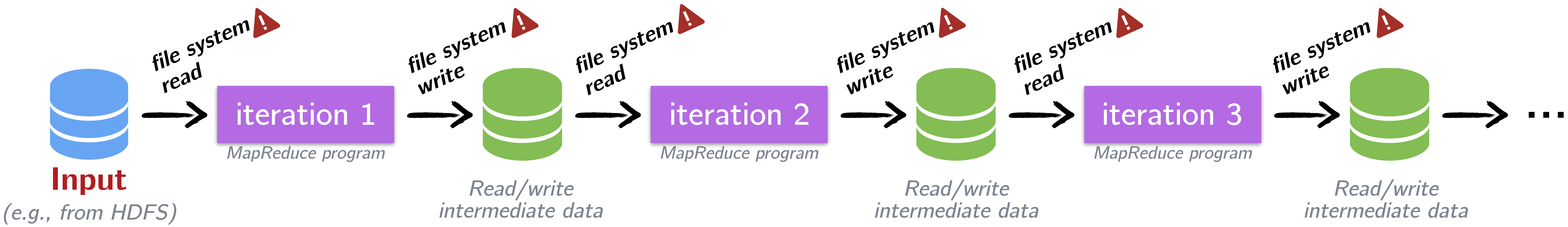
Iteration in Hadoop:



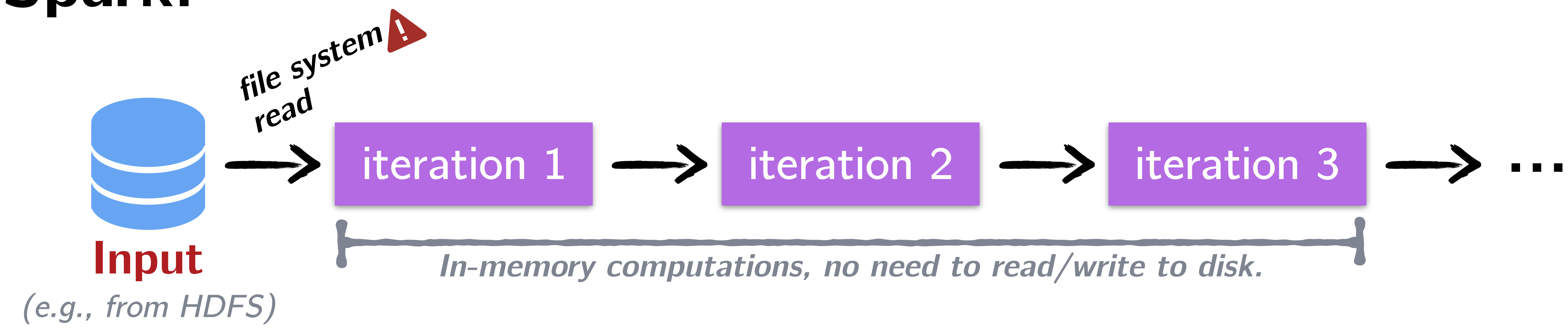
>90% of time in IO that Spark can avoid.

Iteration and Big Data Processing

Iteration in Hadoop:

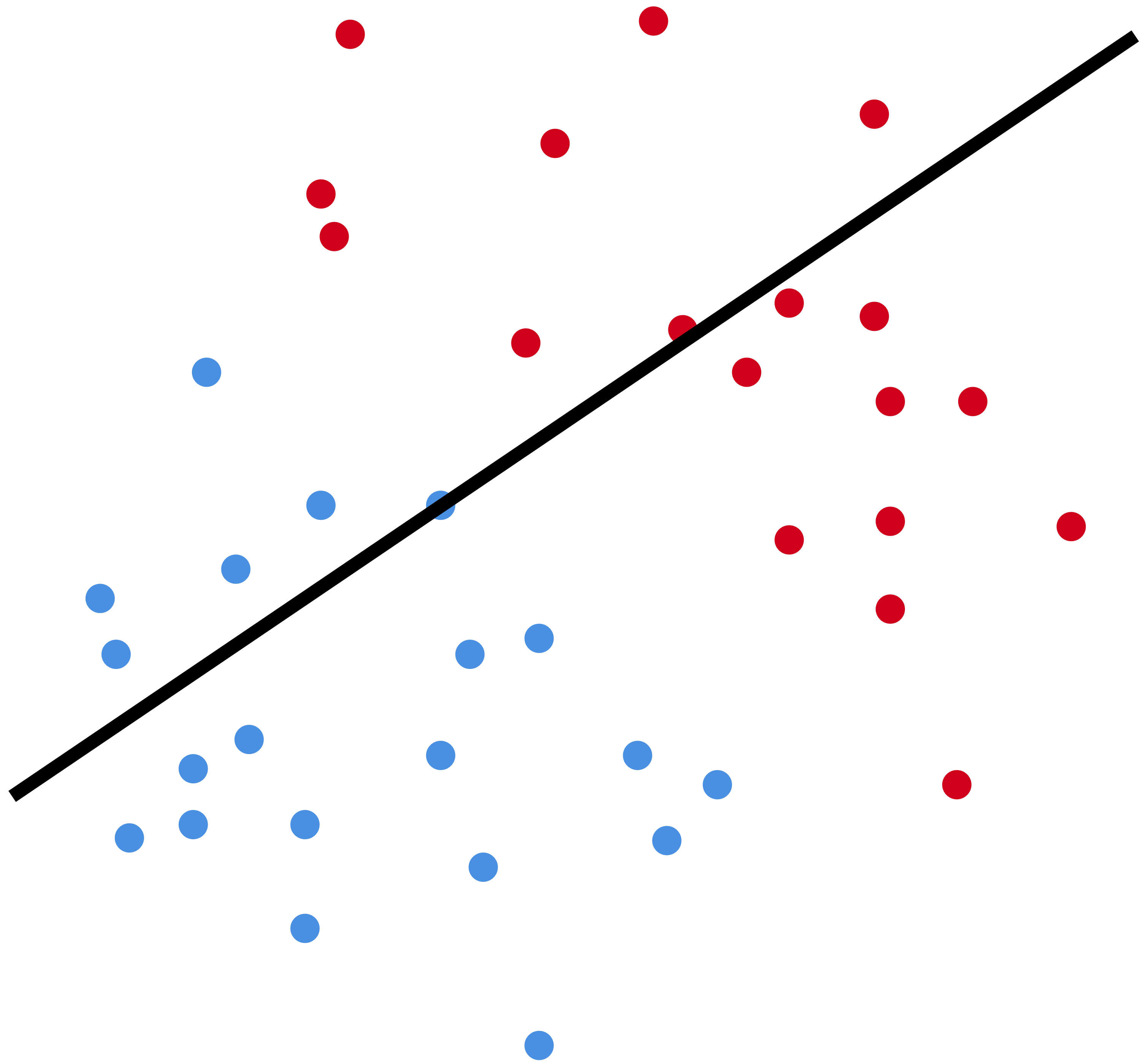


Iteration in Spark:



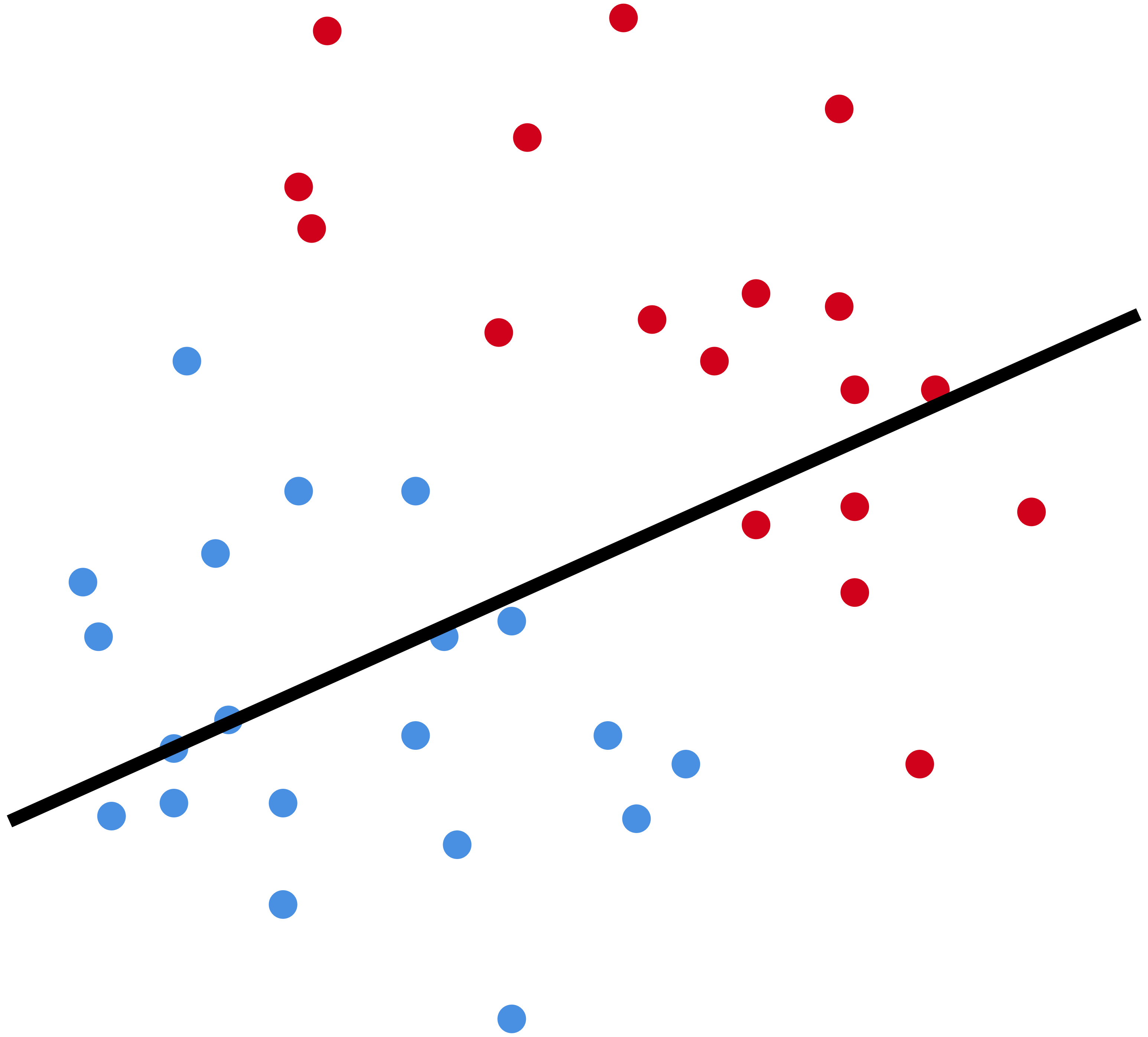
Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.



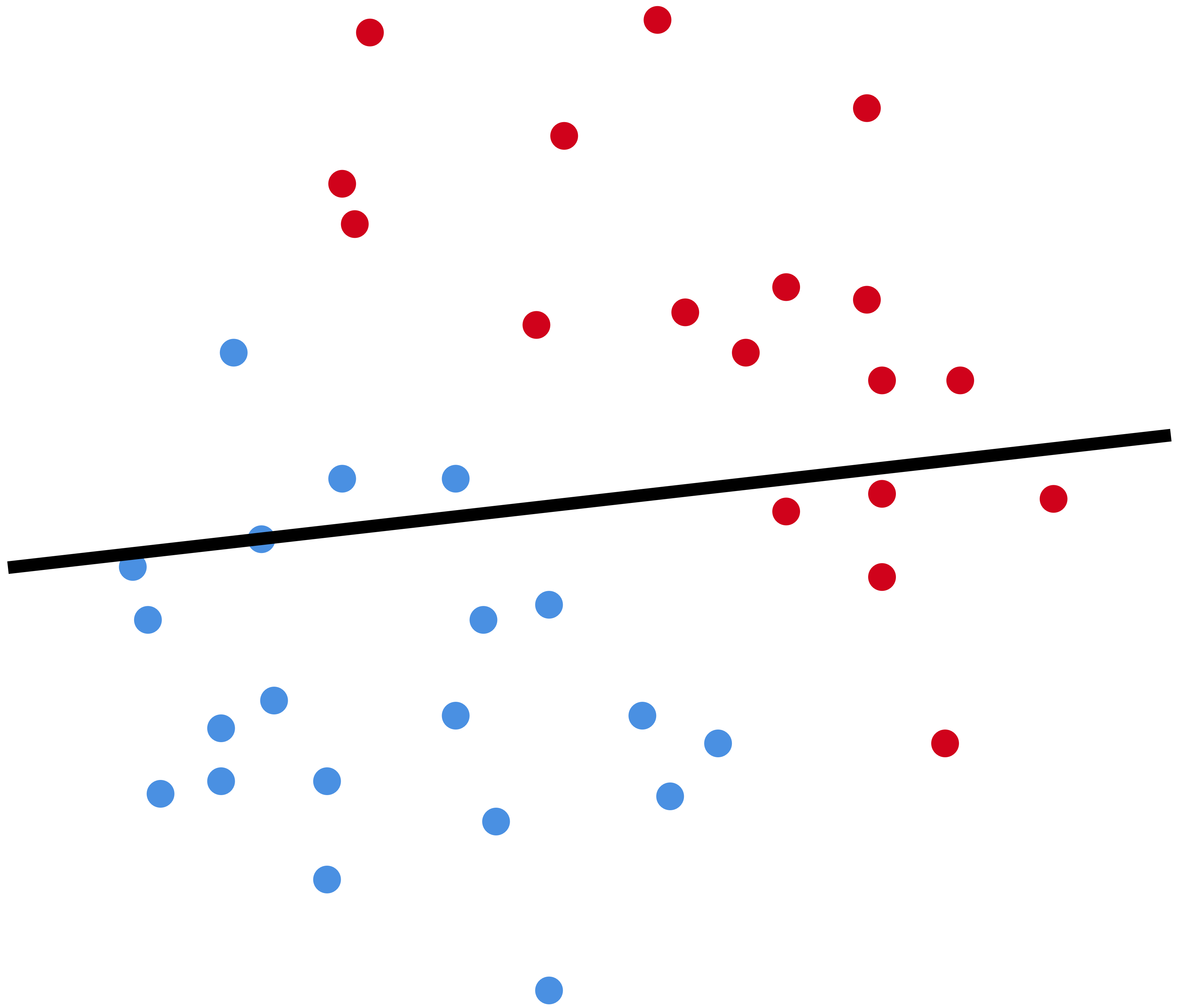
Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.



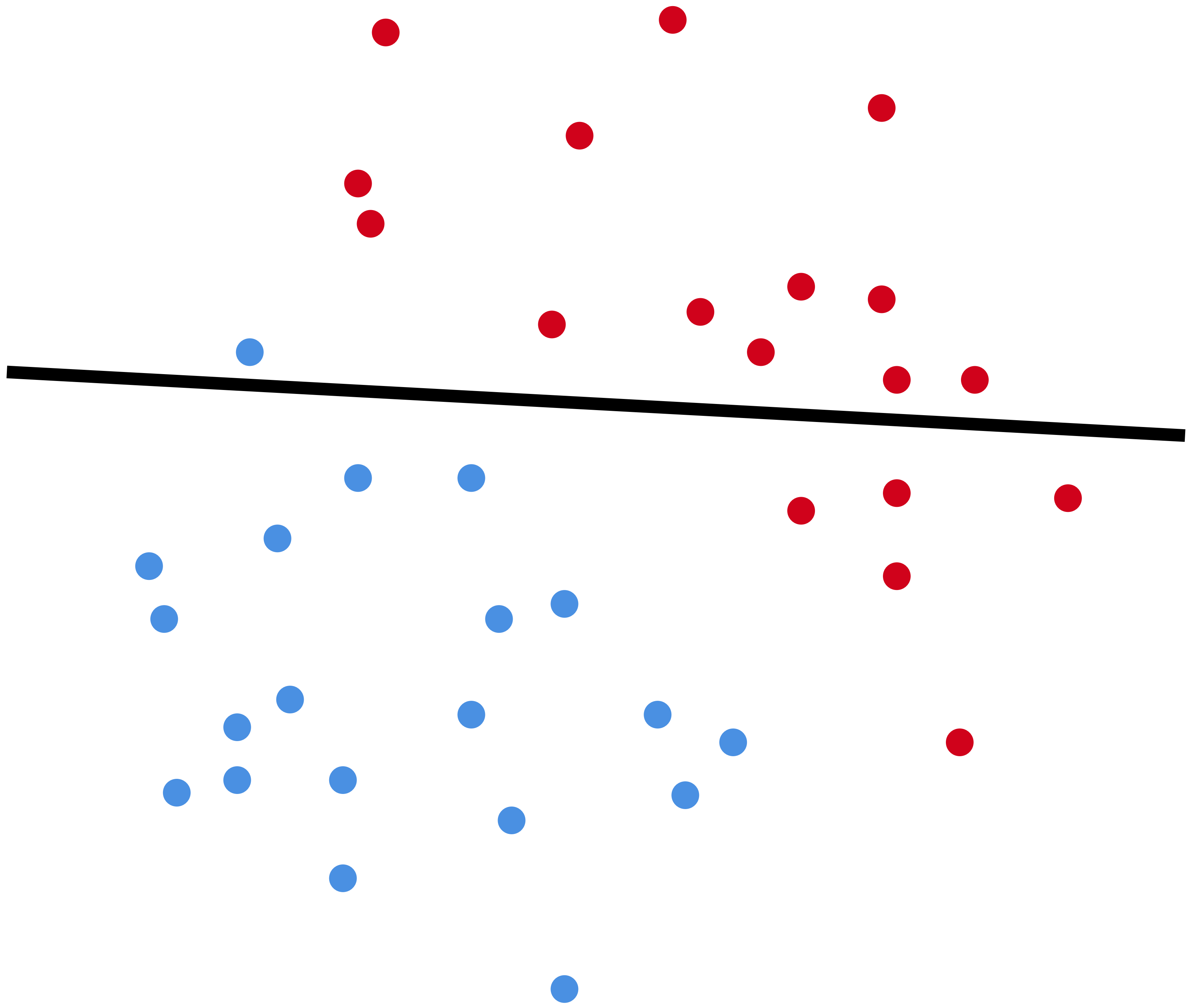
Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.



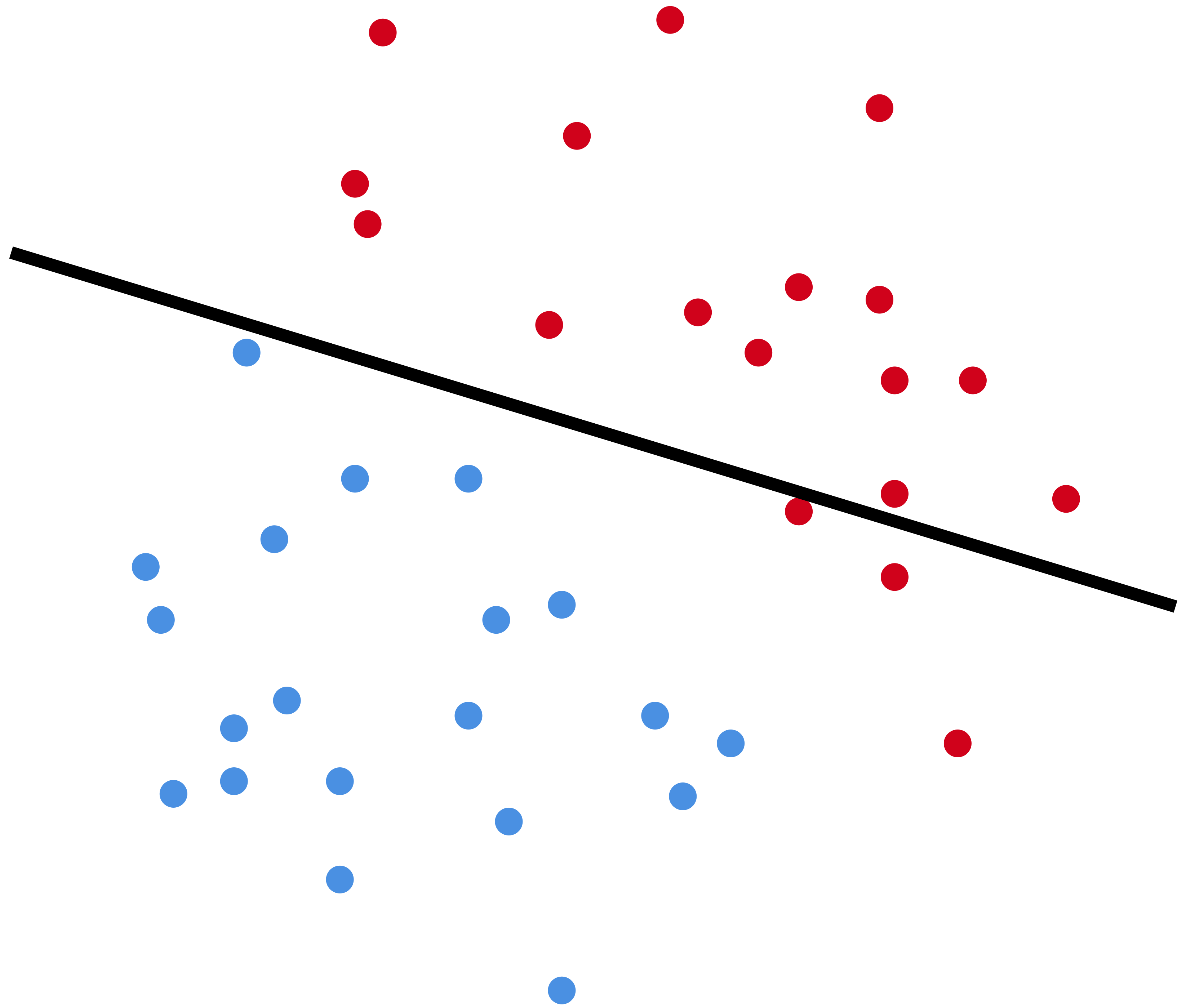
Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.



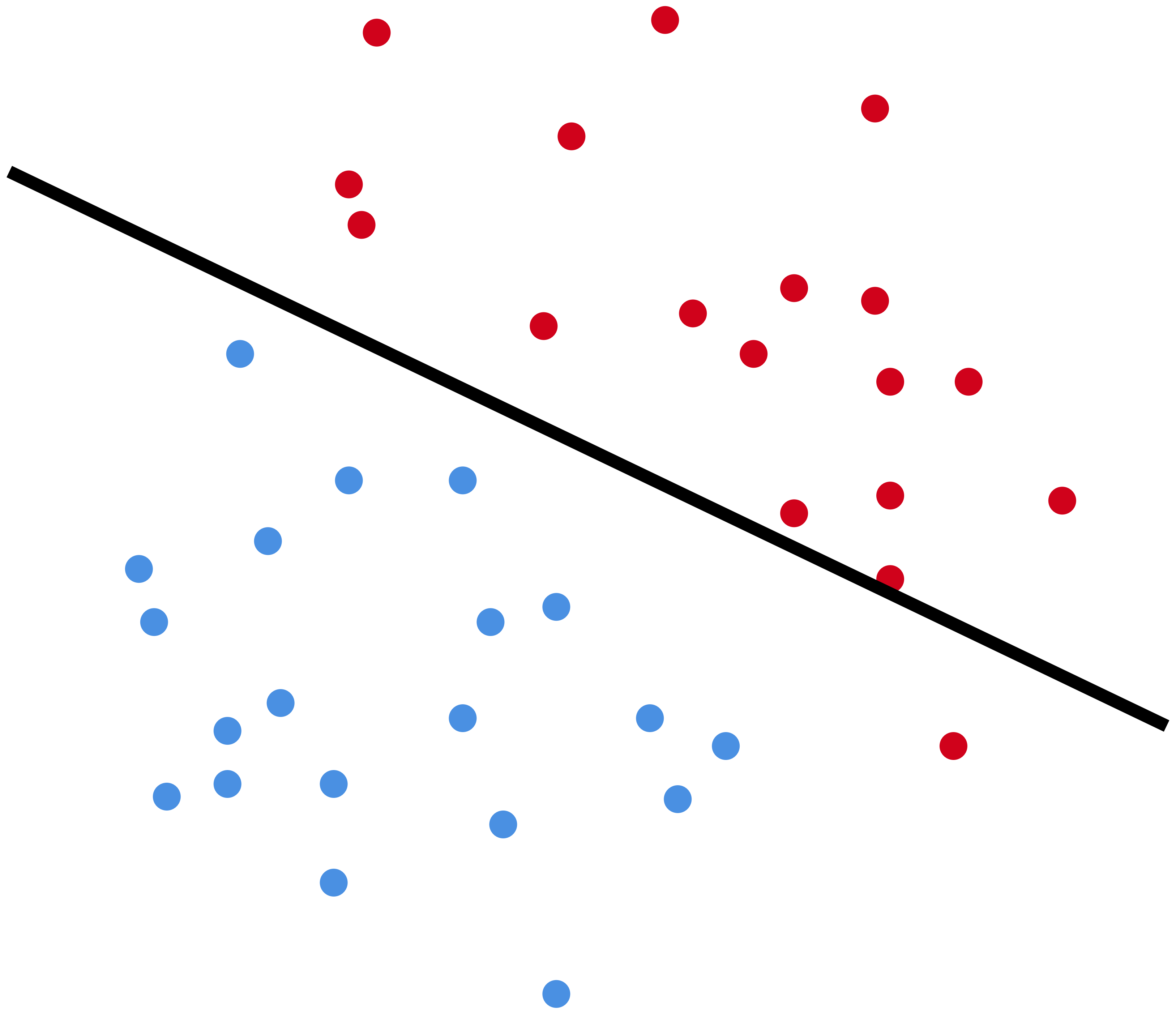
Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.



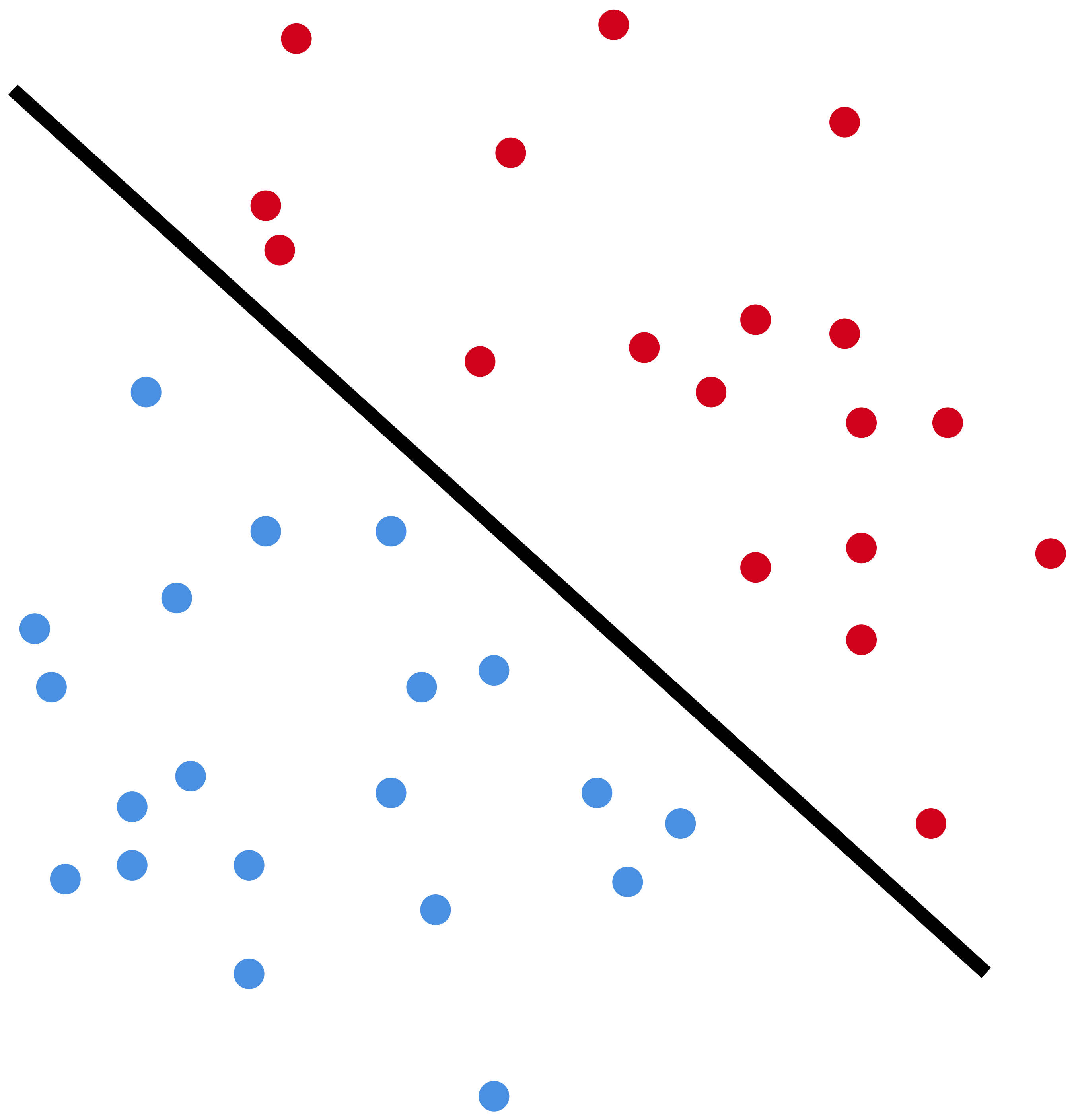
Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.



Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.



Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

Logistic regression can be implemented in Spark in a straightforward way:

case class Point(x: Double, y: Double)

```
val points = sc.textFile(...).map(parsePoint)
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)))- 1) * p.y * p.y
  }.reduce(_ + _)
  w -= alpha * gradient
}
```

What's going on in this code snippet?

Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.

```
val points = sc.textFile(...).map(parsePoint)
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)))- 1) * p.y * p.y
  }.reduce(_ + _)
  w -= alpha * gradient
}
```

`points` is being re-evaluated upon every iteration!
That's unnecessary! What can we do about this?

Caching and Persistence

By default, RDDs are recomputed each time you run an action on them. This can be expensive (in time) if you need to use a dataset more than once.

Spark allows you to control what is cached in memory.

Caching and Persistence

By default, RDDs are recomputed each time you run an action on them. This can be expensive (in time) if you need to use a dataset more than once.

Spark allows you to control what is cached in memory.

To tell Spark to cache an RDD in memory, simply call `persist()` or `cache()` on it.

Caching and Persistence

By default, RDDs are recomputed each time you run an action on them. This can be expensive (in time) if you need to use a dataset more than once.

Spark allows you to control what is cached in memory.

```
val lastYearsLogs: RDD[String] = ...
val logsWithErrors = lastYearsLogs.filter(_.contains(”ERROR”)).persist()
val firstLogsWithErrors = logsWithErrors.take(10)
```

Here, we *cache* `logsWithErrors` in memory.

After `firstLogsWithErrors` is computed, Spark will store the contents of `logsWithErrors` for faster access in future operations if we would like to reuse it.

Caching and Persistence

By default, RDDs are recomputed each time you run an action on them. This can be expensive (in time) if you need to use a dataset more than once.

Spark allows you to control what is cached in memory.

```scala
val lastYearsLogs: RDD[String] = ...
val logsWithErrors = lastYearsLogs.filter(_.contains(”ERROR”)).persist()
val firstLogsWithErrors = logsWithErrors.take(10)
val numErrors = logsWithErrors.count() // faster
```

Now, computing the `count` on `logsWithErrors` is much faster.

Back to Our Logistic Regression Example

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.

```
val points = sc.textFile(...).map(parsePoint).persist()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)))- 1) * p.y * p.y
  }.reduce(_ + _)
  w -= alpha * gradient
}
```

Now, `points` is evaluated once and and is cached in memory. It is then re-used on each iteration.

Caching and Persistence

There are many ways to configure how your data is persisted.

Possible to persist data set:

- in memory as regular Java objects
- on disk as regular Java objects
- in memory as serialized Java objects (more compact)
- on disk as serialized Java objects (more compact)
- both in memory and on disk (spill over to disk to avoid re-computation)

cache( )
Shorthand for using the default storage level, which is in memory only as regular Java objects.

`persist`
Persistence can be customized with this method. Pass the storage level you'd like as a parameter to `persist`.

Caching and Persistence

Storage levels. Other ways to control how Spark stores objects.

Level	*Space used*	*CPU time*	*In memory*	*On disk*
`MEMORY_ONLY`	High	Low	Y	N
`MEMORY_ONLY_SER`	Low	High	Y	N
`MEMORY_AND_DISK`*	High	Medium	Some	Some
`MEMORY_AND_DISK_SER`†	Low	High	Some	Some
`DISK_ONLY`	Low	High	N	Y

* Spills to disk if there is too much data to fit in memory

† Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.

Caching and Persistence

Storage levels. Other ways to control how Spark stores objects.

Level	*Space used*	*CPU time*	*In memory*	*On disk*
`MEMORY_ONLY`	**High**	**Low**	**Y**	**N**
`MEMORY_ONLY_SER`	Low	High	Y	N
`MEMORY_AND_DISK`[*]	High	Medium	Some	Some
`MEMORY_AND_DISK_SER`[†]	Low	High	Some	Some
`DISK_ONLY`	Low	High	N	Y

Default

[*] Spills to disk if there is too much data to fit in memory

[†] Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.

RDDs Look Like Collections, But Behave Totally Differently

Key takeaway:

Despite similar-looking API to Scala Collections,
the deferred semantics of Spark's RDDs are very unlike Scala Collections.

RDDs Look Like Collections, But Behave Totally Differently

Key takeaway:

Despite similar-looking API to Scala Collections,
the deferred semantics of Spark's RDDs are very unlike Scala Collections.

Due to:

- the lazy semantics of RDD transformation operations (`map`, `flatMap`, `filter`),
- and users' implicit reflex to assume collections are eagerly evaluated...

...One of the most common performance bottlenecks of newcomers to Spark arises from unknowingly re-evaluating several transformations when caching could be used.

RDDs Look Like Collections, But Behave Totally Differently

Key takeaway:

> Despite similar-looking API to Scala Collections,
> **the deferred semantics of Spark's RDDs are very unlike Scala Collections.**

Due to:

- the lazy semantics of RDD transformation operations (`map`, `flatMap`, `filter`),
- and users' implicit reflex to assume collections are eagerly evaluated...

...One of the most common performance bottlenecks of newcomers to Spark arises from unknowingly re-evaluating several transformations when caching could be used.

Don't make this mistake in your programming assignments.

Restating the Benefits of Laziness for Large-Scale Data

While many users struggle with the lazy semantics of RDDs at first, it's helpful to remember the ways in which these semantics are helpful in the face of large-scale distributed computing.

Example #1:

```
val lastYearsLogs: RDD[String] = ...
val firstLogsWithErrors = lastYearsLogs.filter(_.contains(”ERROR”)).take(10)
```

Restating the Benefits of Laziness for Large-Scale Data

While many users struggle with the lazy semantics of RDDs at first, it's helpful to remember the ways in which these semantics are helpful in the face of large-scale distributed computing.

Example #1:

```scala
val lastYearsLogs: RDD[String] = ...
val firstLogsWithErrors = lastYearsLogs.filter(_.contains(”ERROR”)).take(10)
```

The execution of `filter` is deferred until the `take` action is applied.

Spark leverages this by analyzing and optimizing the **chain of operations** before executing it.

Spark will not compute intermediate RDDs. Instead, as soon as 10 elements of the filtered RDD have been computed, `firstLogsWithErrors` is done. At this point Spark stops working, saving time and space computing elements of the unused result of `filter`.

Restating the Benefits of Laziness for Large-Scale Data

While many users struggle with the lazy semantics of RDDs at first, it’s helpful to remember the ways in which these semantics are helpful in the face of large-scale distributed computing.

Example #2:

```
val lastYearsLogs: RDD[String] = ...
val numErrors = lastYearsLogs.map(_.lowercase)
                             .filter(_.contains(”error”))
                             .count()
```

Restating the Benefits of Laziness for Large-Scale Data

While many users struggle with the lazy semantics of RDDs at first, it’s helpful to remember the ways in which these semantics are helpful in the face of large-scale distributed computing.

Example #2:

```
val lastYearsLogs: RDD[String] = ...
val numErrors = lastYearsLogs.map(_.lowercase)
                             .filter(_.contains(”error”))
                             .count()
```

Lazy evaluation of these transformations allows Spark to *stage* computations. That is, Spark can make important optimizations to the the **chain of operations** before execution.

For example, after calling `map` and `filter`, Spark knows that it can avoid doing multiple passes through the data. That is, Spark can traverse through the RDD once, computing the result of `map` and `filter` in this single pass, before returning the resulting count.