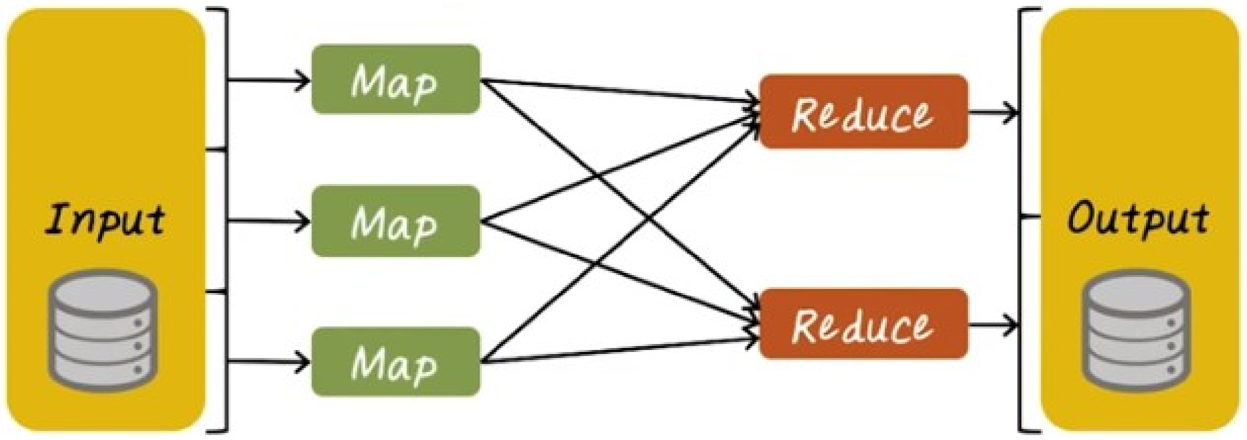
1. Previously, we have talked about MapReduce, a distributed fault tolerance system, for processing large data set. However, MapReduce is not efficient for supporting iterative workload as many machine learning algorithm require. Today, we'll introduce Spark, another big data system that provides better performance, by using distributed memory across many machines. We'll talk about the key concept behind Spark. Namely, Resilient Distributed Dataset or RDD. We'll explain how Spark can better support iterative algorithms. We also provide some example of house care applications using Spark.

ENVIRONMENT

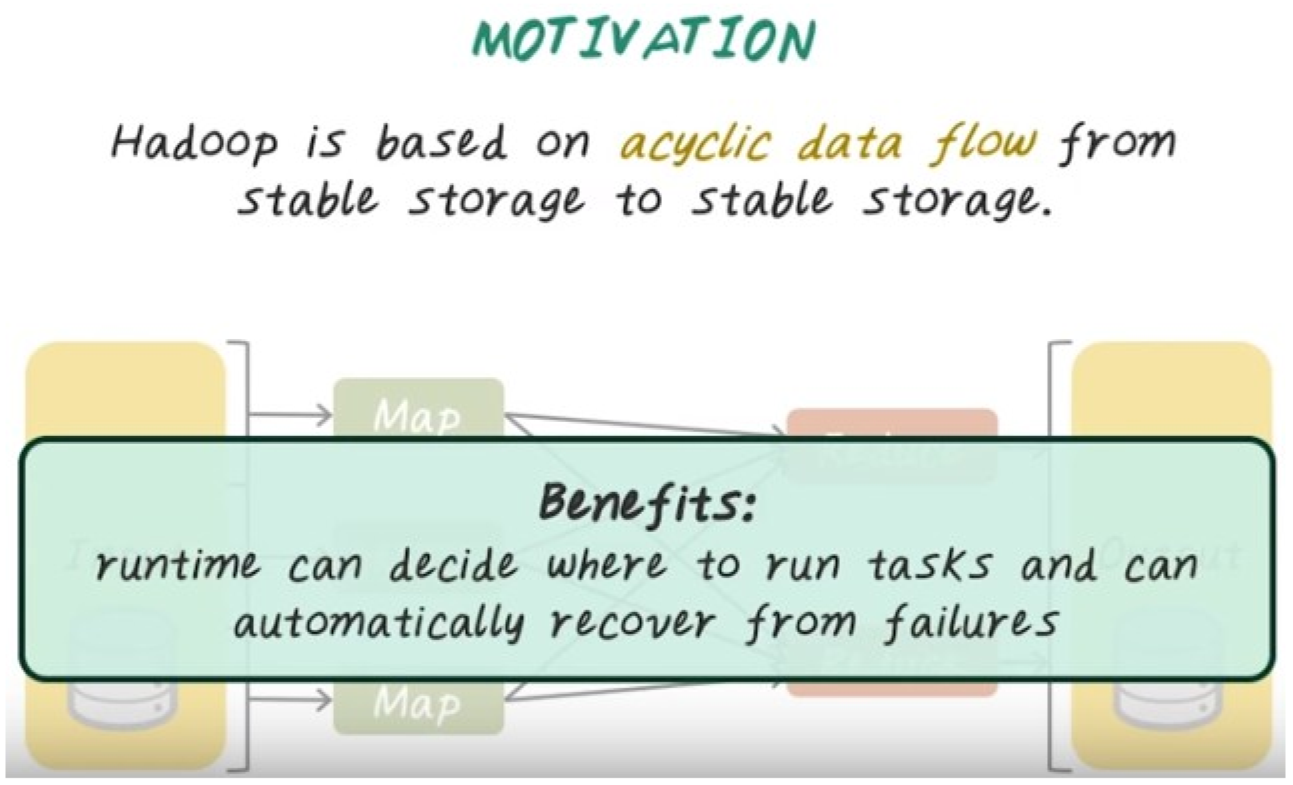
2. Before we introduce the big data system Spark, let's first remind ourself of the computing environment we're using here. For big data analytics, we usually need to perform all the analytics in a data center look like this. Many racks of servers that are interconnected through Internet. A lot of time we access this environment through cloud computing services such as, Amazon Web Services, Google Cloud Platform and Microsoft Azure. With this environment in mind, next we'll see why we need to design another big data system like Spark

MOTIVATION

Hadoop is based on acyclic data flow from stable storage to stable storage.



3. In the previous lecture we have introduced Hadoop and MapReduce. Hadoop is a big data system that operates on acyclic data flow graphs from stable storage to stable storage. So the input and output of Hadoop jobs are from the stable storage system. For example, hard disks in the distributed environment. The computing jobs follows the MapReduce paradigm, which forms acyclic data flow graph that look like this. And all the input and output from each Map and Reduce function are in this stable storage system.

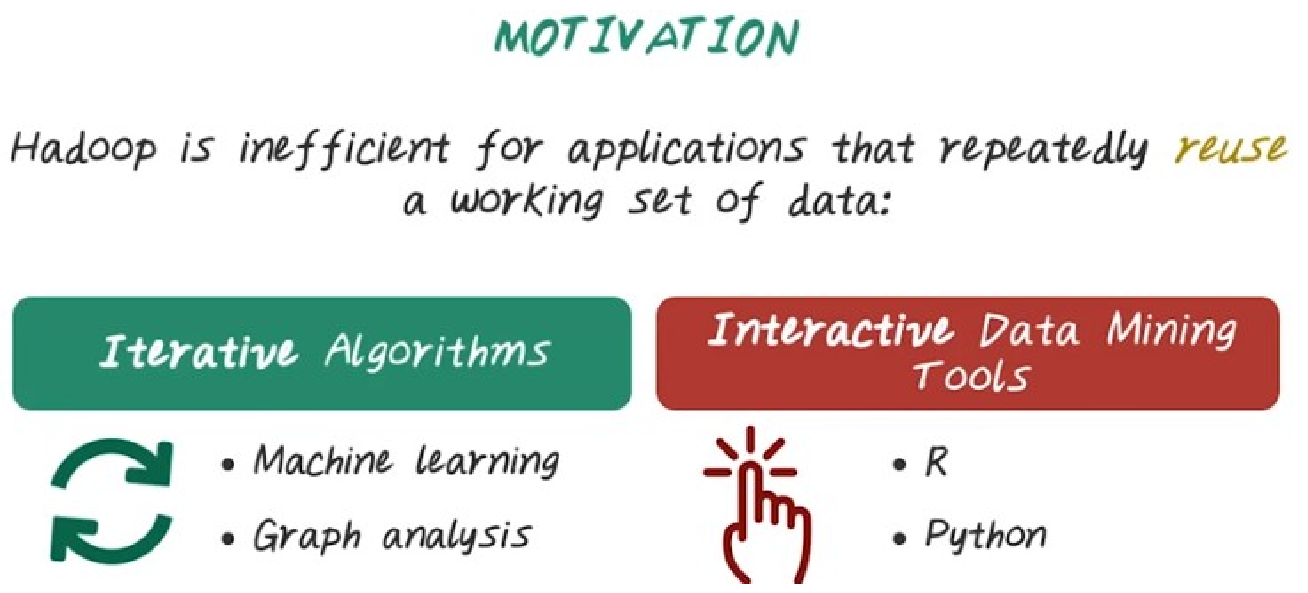


MOTIVATION

Hadoop is based on acyclic data flow from stable storage to stable storage.

Benefits:
runtime can decide where to run tasks and can automatically recover from failures

So Hadoop system is based on acyclic data flow graph and stable storage to ensure fault tolerance (fault tolerance 的意思應該就是 recover from failure). So the benefit of such a design is that at one time we can decide where to run different tasks. So the map and reduce function can be run on different machine in the distributed environment. And we can automatically recover from failures because the input and output of those MapReduce functions are stored in a stable storage. For example, disk.

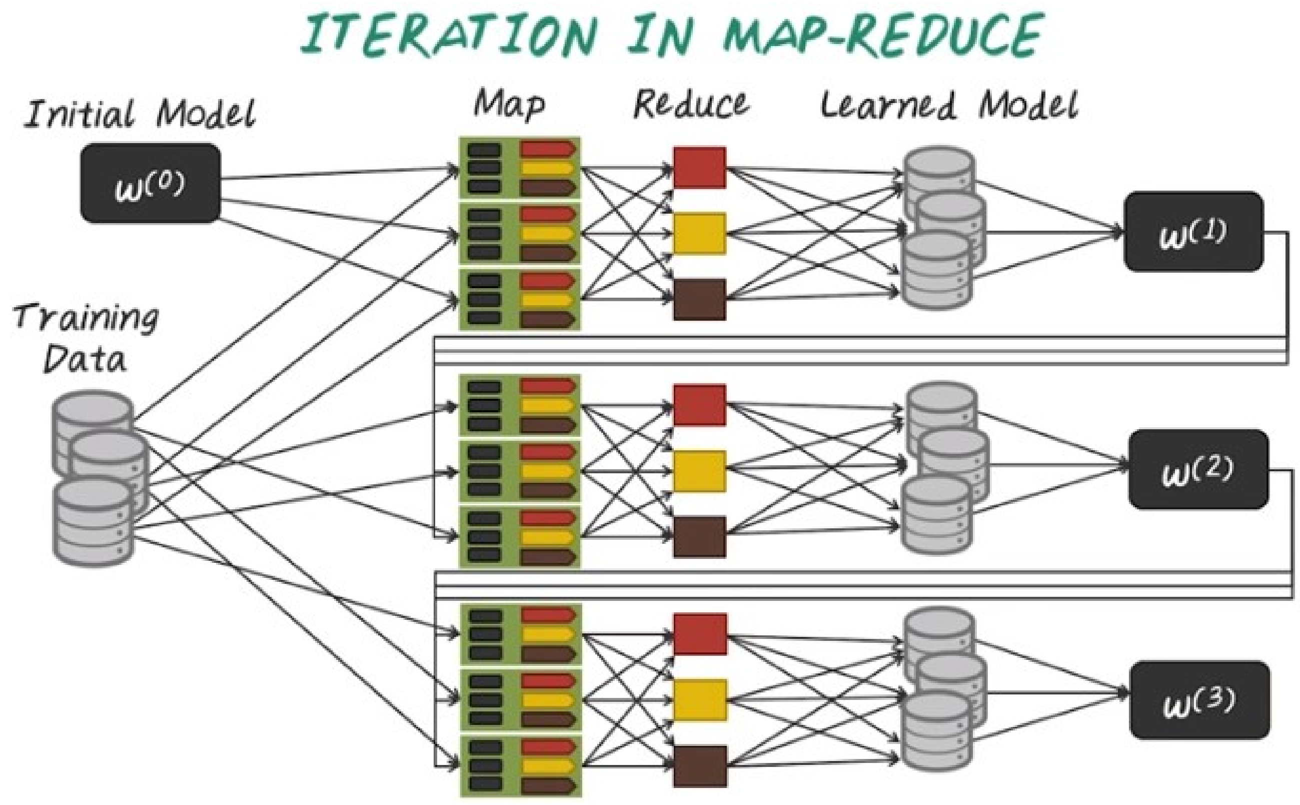


MOTIVATION

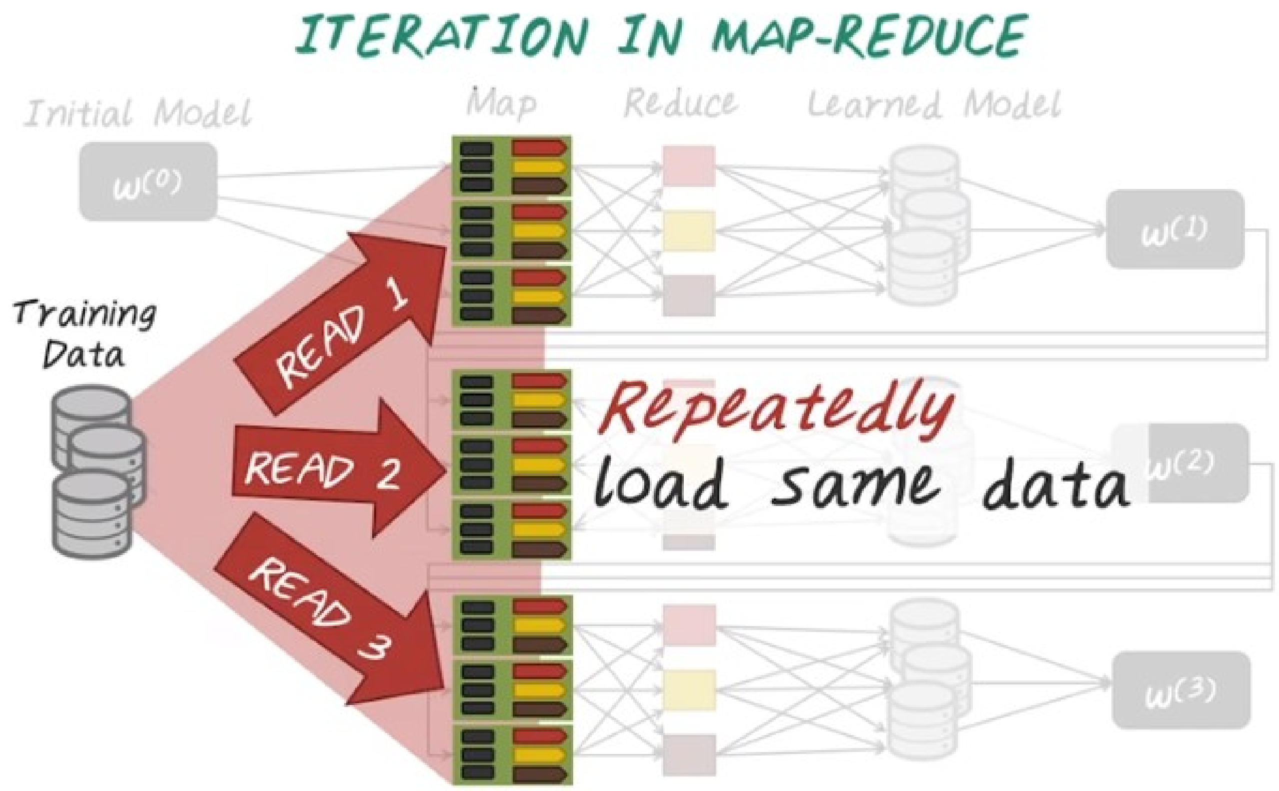
Hadoop is inefficient for applications that repeatedly reuse a working set of data:

Iterative Algorithms	Interactive Data Mining Tools
• Machine learning	• R
• Graph analysis	• Python

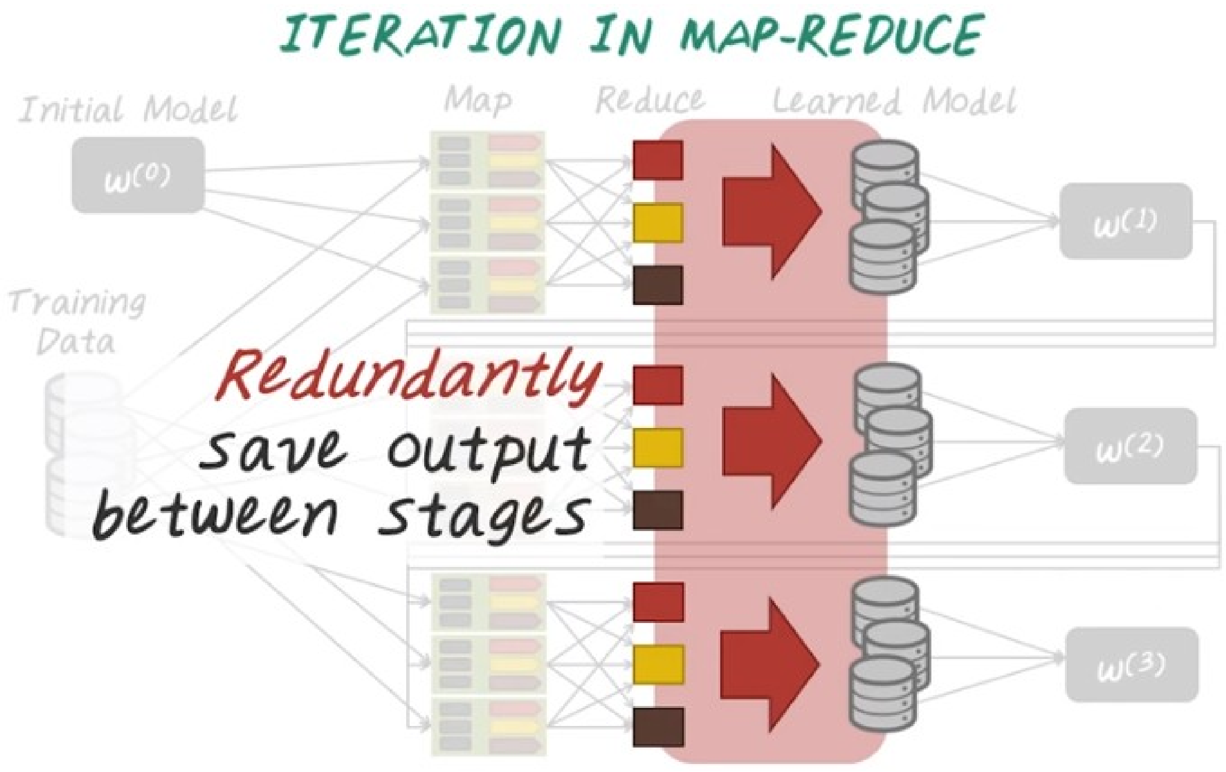
So now we know Hadoop can work with big data using MapReduce computation. So what are the limitations of Hadoop? Hadoop often does not perform well when the workload involves cycles. More precisely, Hadoop is inefficient for application that repeatedly reuse a working set of data. So what are those application that require such workload? There are some major applications such as iterative algorithms and interactive data mining tools that fall into this category. Iterative algorithm contains many machine learning algorithms, such as clustering, classification. They often times need to be repeatedly computed on the same data set. Then we have graph analysis. Many graph algorithms, such as page rank computation, spectral clustering, they also fall into this iterative algorithm category. Also, more and more data mining practice need interactive response. This days, data scientists using interactive tools such as R and Python to explore data, form hypotheses, and validate those hypotheses and repeat on the same data set. It will be great to provide more efficient big data platform that can support all this.



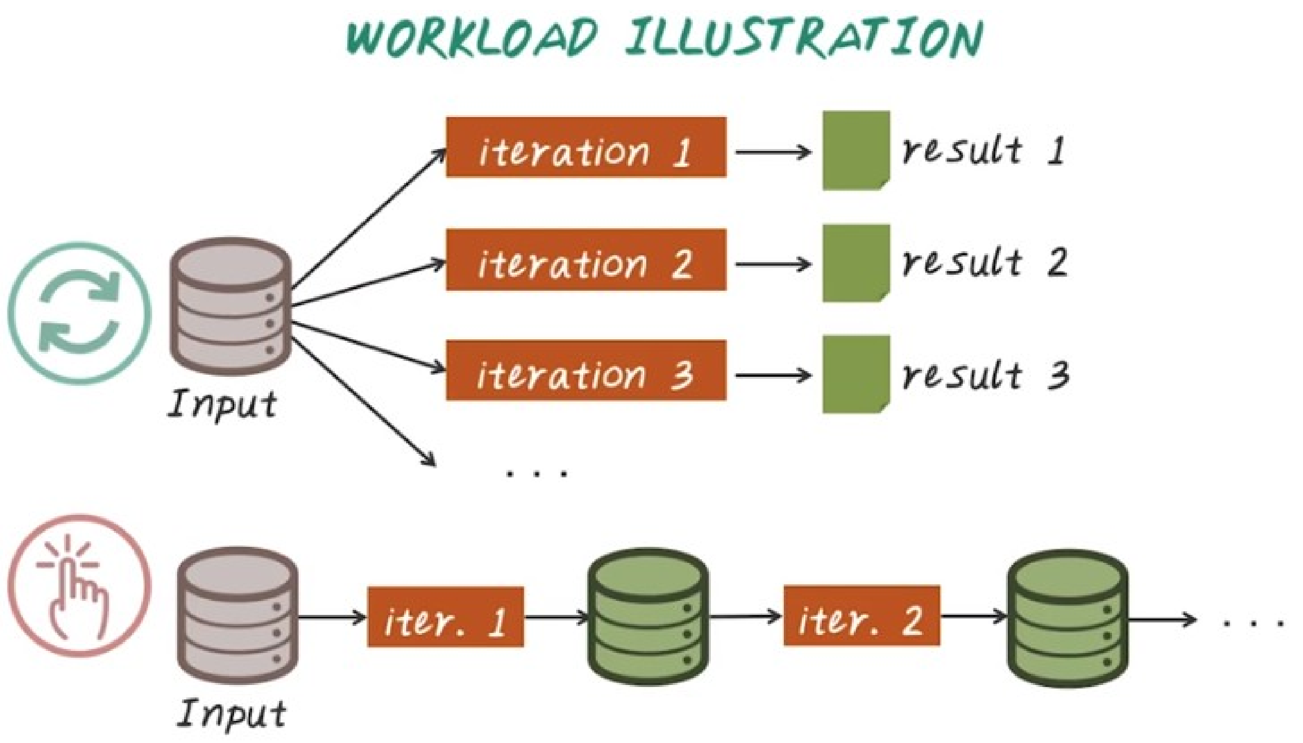
4. Next, let's illustrate the inefficiency of Hadoop using one concrete example. Say we want to a machine learning model on this training set. And the model is specified by the model parameter w. Typically, a machine learning algorithm start with some initial model. Then they run the algorithm for one iteration. For example, this can be implemented as the map-reduce job. And the resulting model, W (1) is also going to be stored on the disk. Then we repeat this process with this new model parameter against the same training data to get the next iteration of model. And this process repeated many times until convergence.



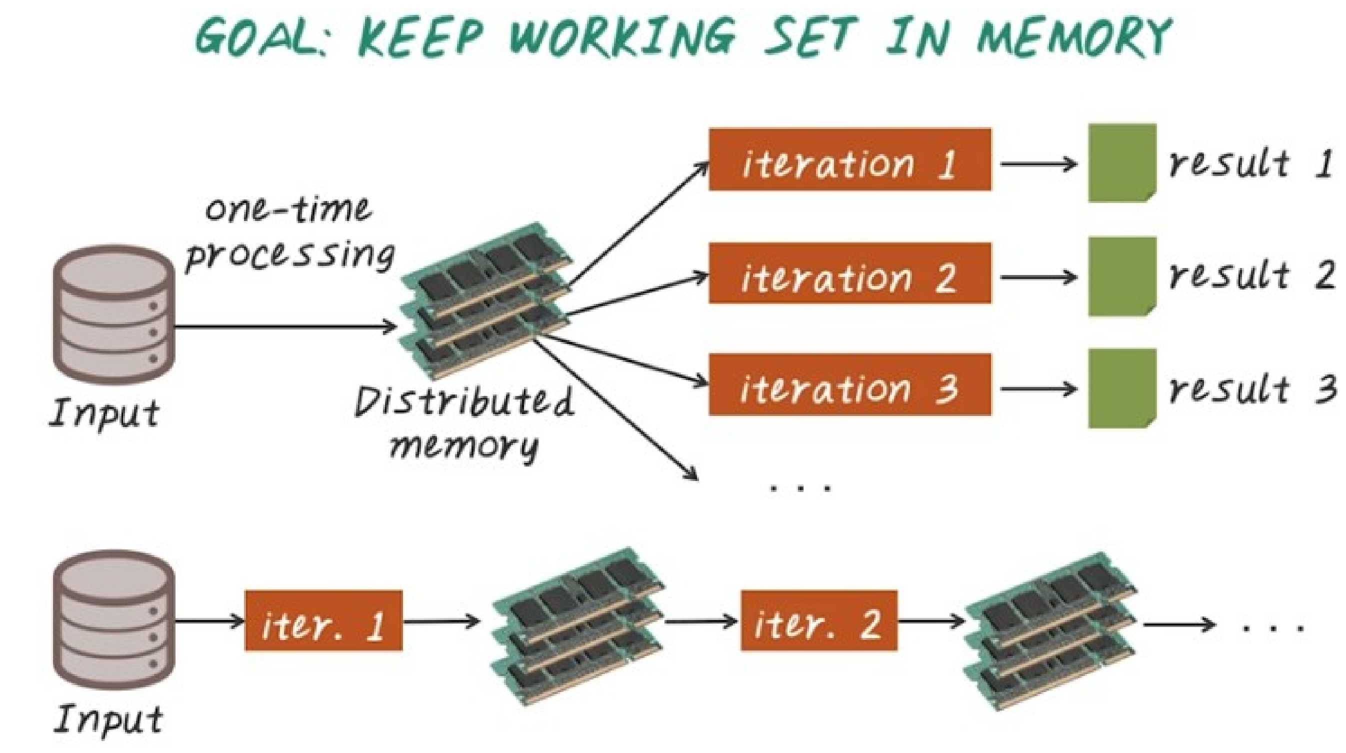
If you look at this computation pattern, you quickly realize that we repeatedly load the same data from disk to memory in order to perform the computation.



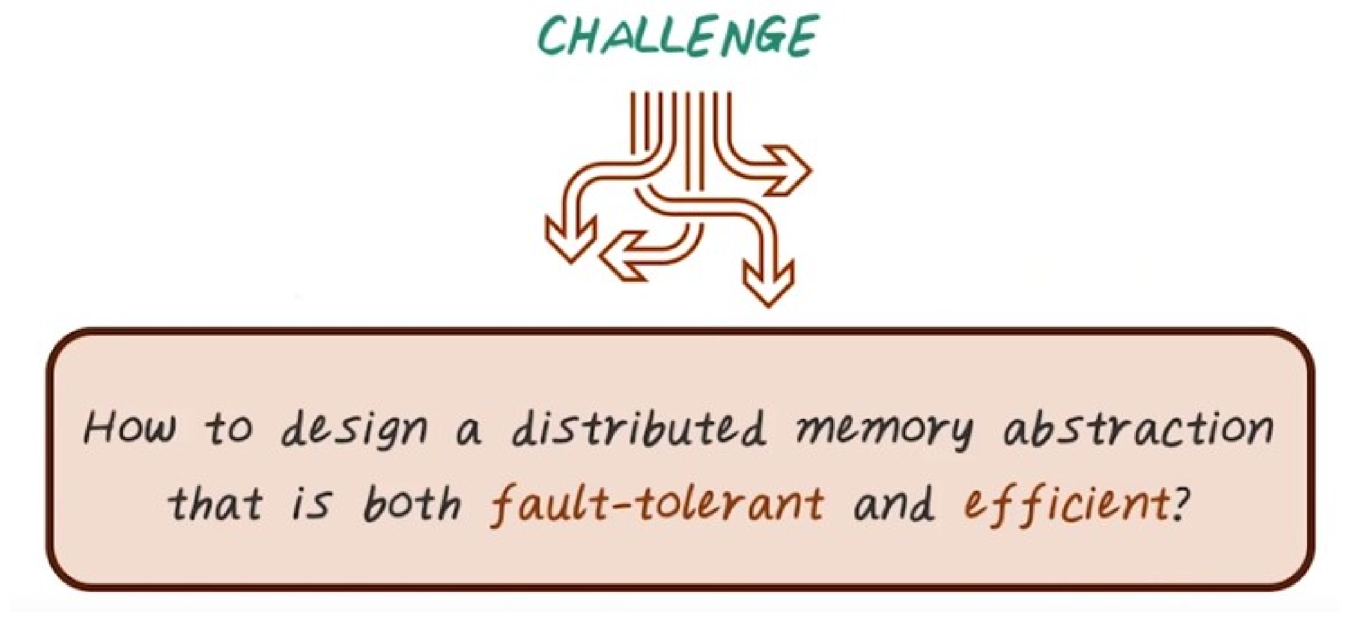
At the same time, we have to write the result, in this case, are those model parameter repeatedly to disk from iteration to iteration. And this repeated reading and writing to disk are the inefficiencies of Hadoop.



5. Next, let's illustrate these two types of workload, iterative algorithm and interactive computation. For iterative algorithm, we apply the same computation logic on the same input data set again and again to generate different iteration of result set. And those result are often corresponding to the model parameters. Interactive computation is slightly different. In this case, we iteratively perform some computation. Across the iteration, those computation can be very different. For example, we start with some raw data, then perform some data cleaning, get a cleaner data, then perform modeling algorithms to get the model. And the requirement is, we want this iteration to iteration to be fast so that we can perform this work in a interactive manner.



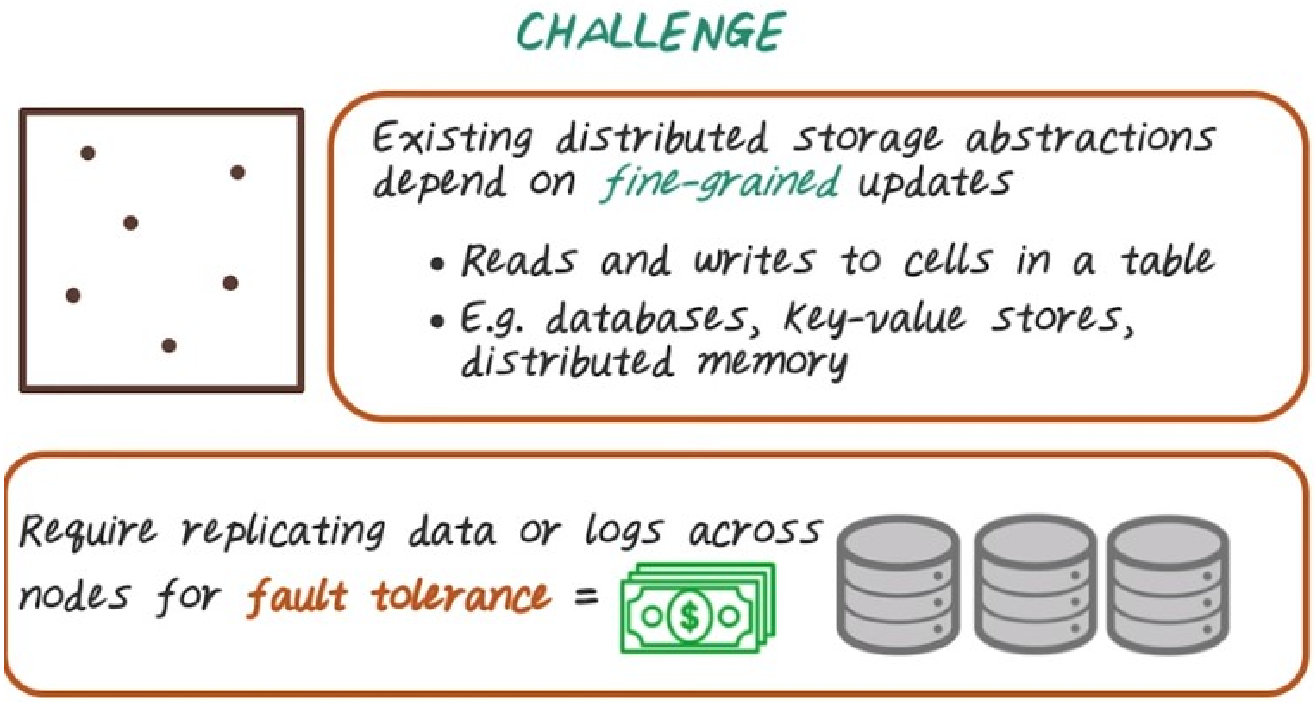
So the key objective for supporting iterative algorithms and interactive computation is to keep the working set in memory so that we can perform all those operations fast. So here we illustrate that ideas using this diagram. For iterative algorithm, what we want to do here is load the entire data set into a distributed memory across many machines. So that when we perform all of this iterative computation, all the data are in memory so we don't have to read and write to disk. For the interactive computation, the idea is also very similar. We want to keep the intermediate result all in memory so that the next iteration can perform immediately once the first iteration is done. There is no read and write to disk anymore.



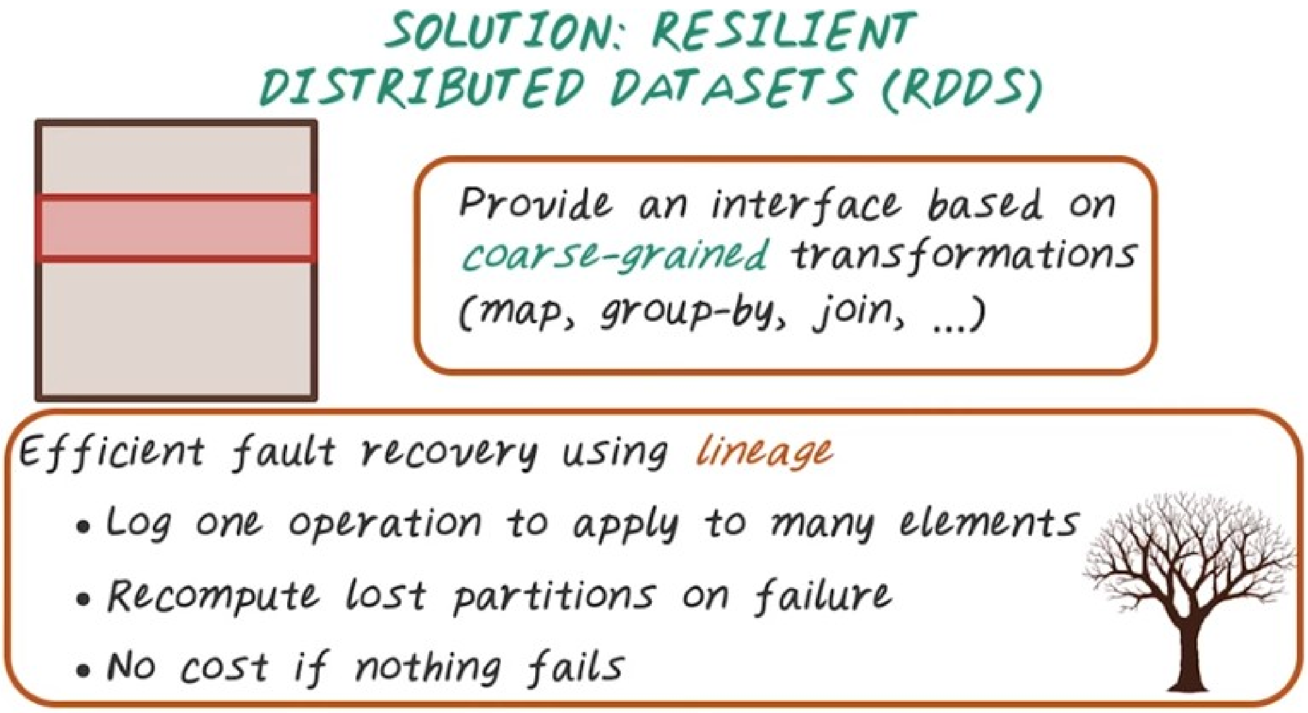
CHALLENGE

How to design a distributed memory abstraction that is both fault-tolerant and efficient?

6. So what's the challenge about keeping everything in memory? The key challenge is how do we design a distributed memory abstraction that is both fault-tolerant and efficient. Hadoop guarantee fault-tolerance by using disk so that they can keep data and intermediate result in a stable storage. But, memory is not a stable storage. It's, by definition, efficient, but it's not fault-tolerant. To guarantee both fault-tolerance and efficiency is the challenge for such a system.


CHALLENGE
Existing distributed storage abstractions depend on fine-grained updates
• Reads and writes to cells in a table
• E.g. databases, key-value stores, distributed memory
Require replicating data or logs across nodes for fault tolerance =

Next, let's look at some options here. A lot of existing distributed storage systems depends on fine-grained updates. For example, given the table look like this, we can re-write from any place in this table. So those dots indicate a place we want to read and write a specific cell from that table. Example of such abstraction include database systems, key-value stores, and distributed memory. So, the way to make this type of storage abstraction fault-tolerance is we have to replicate data like what we have done in the Hadoop setting where we replicate data several times and store on different machines. Or we have to keep track what operation has happened using logs. But if we have this very fine-grained updates any place on this table, then keeping track of what has been changed or replicating all those data become very expensive.


SOLUTION: RESILIENT DISTRIBUTED DATASETS (RDDS)

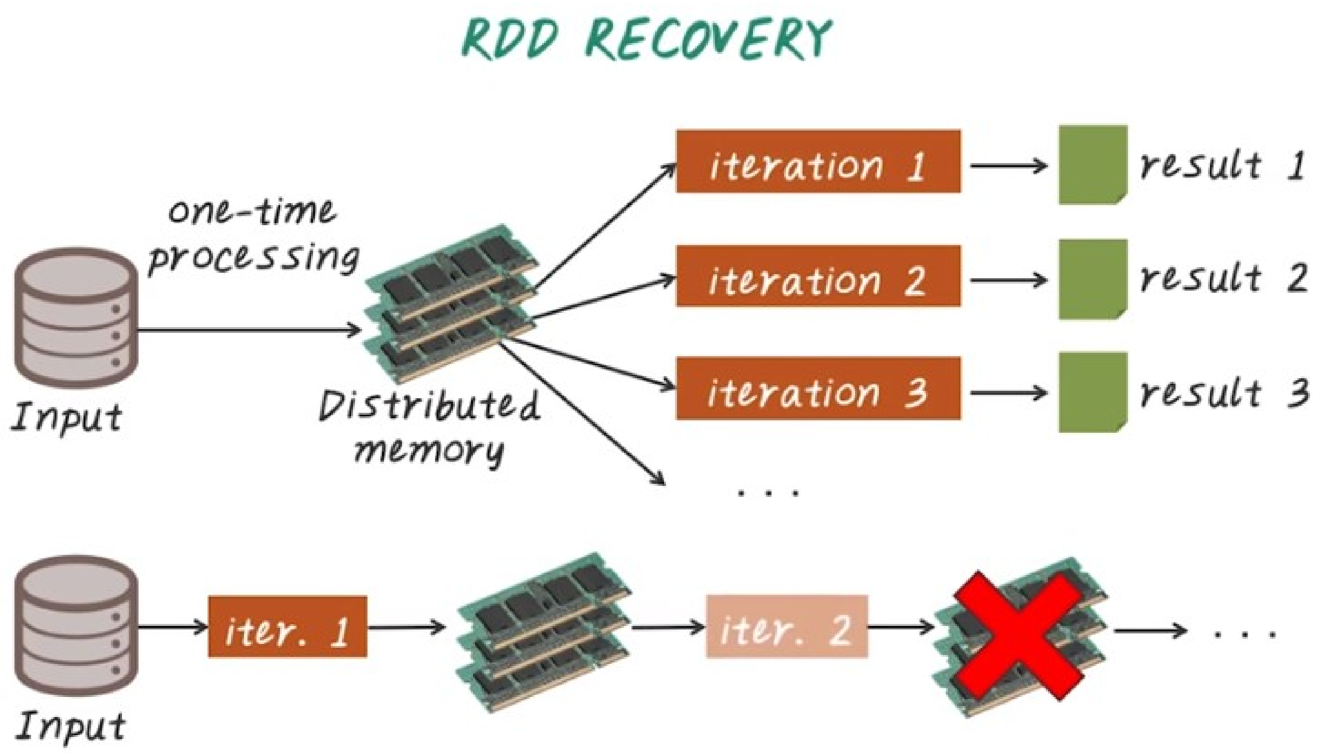
Provide an interface based on *coarse-grained* transformations (map, group-by, join, ...)

Efficient fault recovery using *lineage*

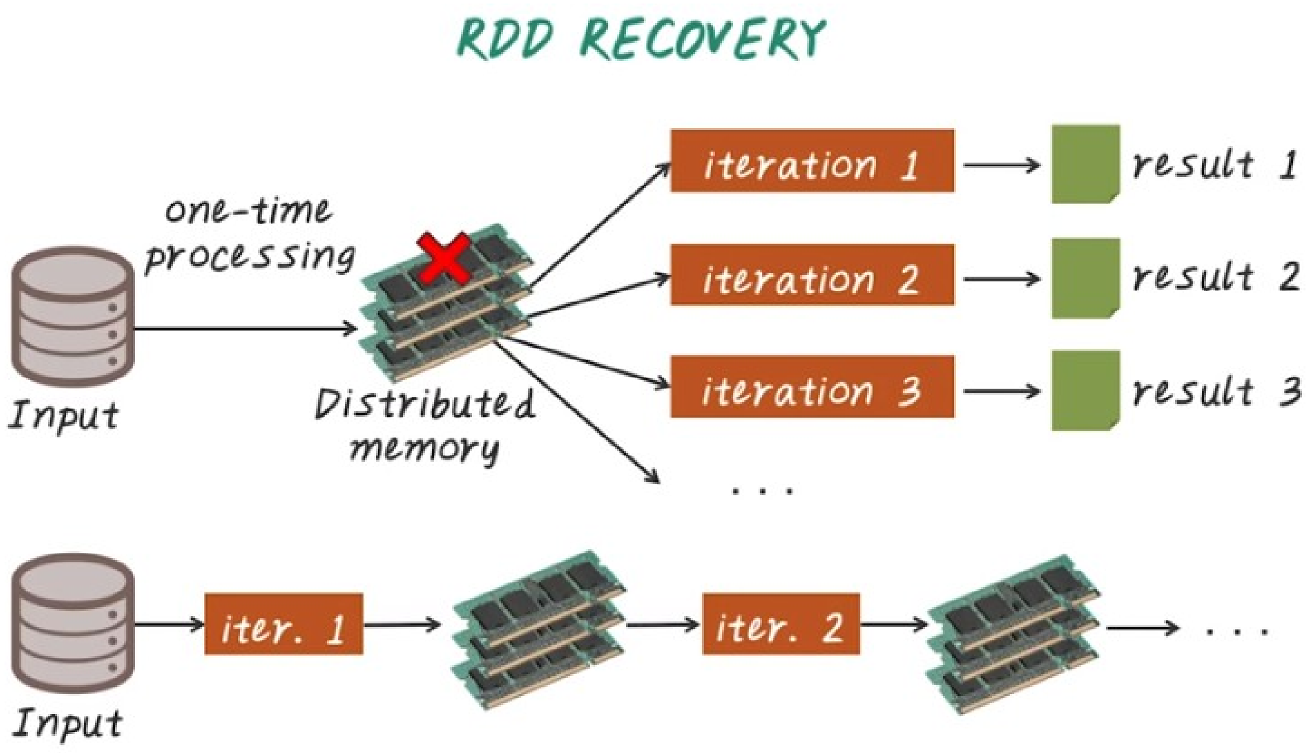
- Log one operation to apply to many elements
- Recompute lost partitions on failure
- No cost if nothing fails

resilient: 有弹性的

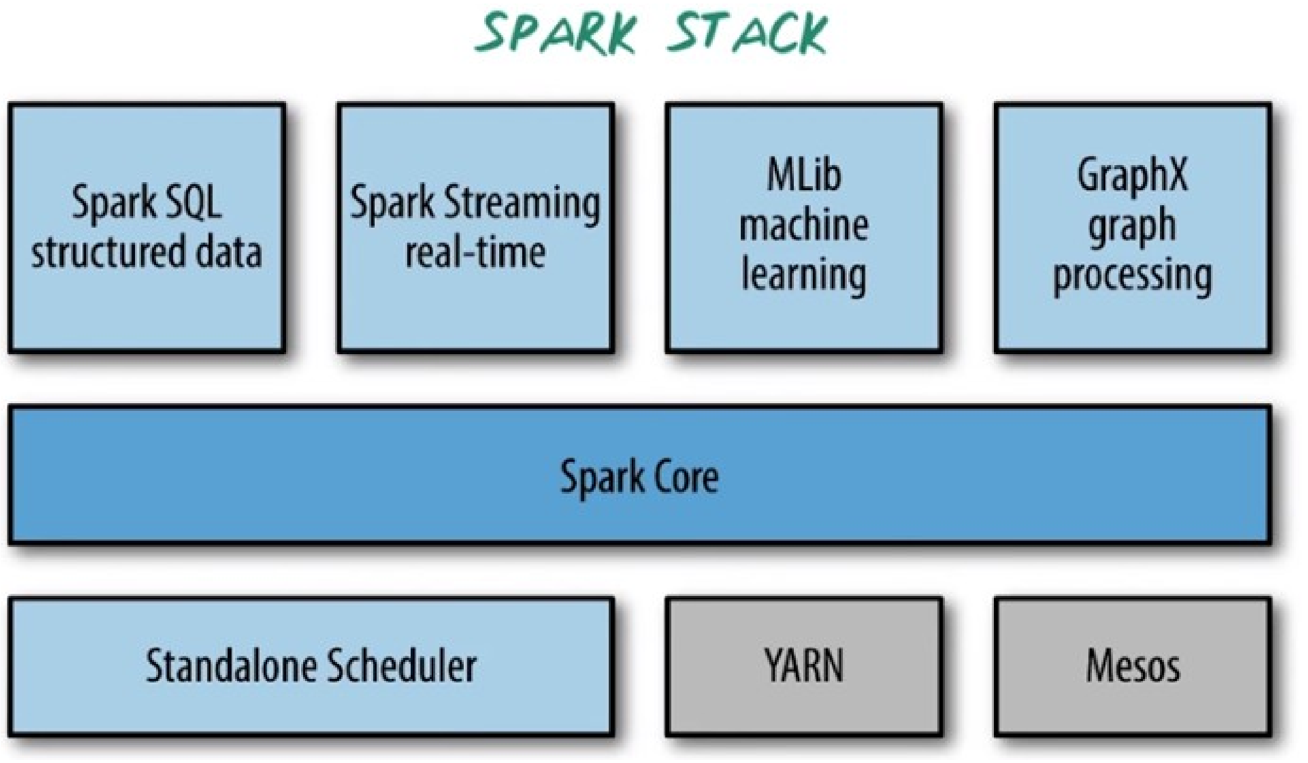
7. To solve this problem, spark decides to strike a balance between granularity of the computation and the efficiency for enabling fall tolerance. In this case, RDD provide an interface based on coarse-grained transformations of the entire data set. For example, a map function can be performed on every record in this dataset. Similarly for group-by or join or filter. So this operation are course-grained operations because it's not focusing on a specific part of the dataset. They are being applied on the entire dataset. If we only have to keep track of the course-grained transformations, all operations can be efficiently tracked. In this case, efficient fault recovery can be done using lineage(親緣關系). So lineage here looks like a family tree. It keeps track of all the transformation across different RDD. It may start with the root RDD and some transformation being applied to those RDD and derived RDD are generated. So, if we only support coarse-grained transformation, we can log those operations that apply to many elements in this RDD. And we can re compute the failure happened because we have the operation being logged. And more importantly, since everything is in memory, there's no cause if nothing fails. So this is very efficient mechanism to enable fault tolerance.



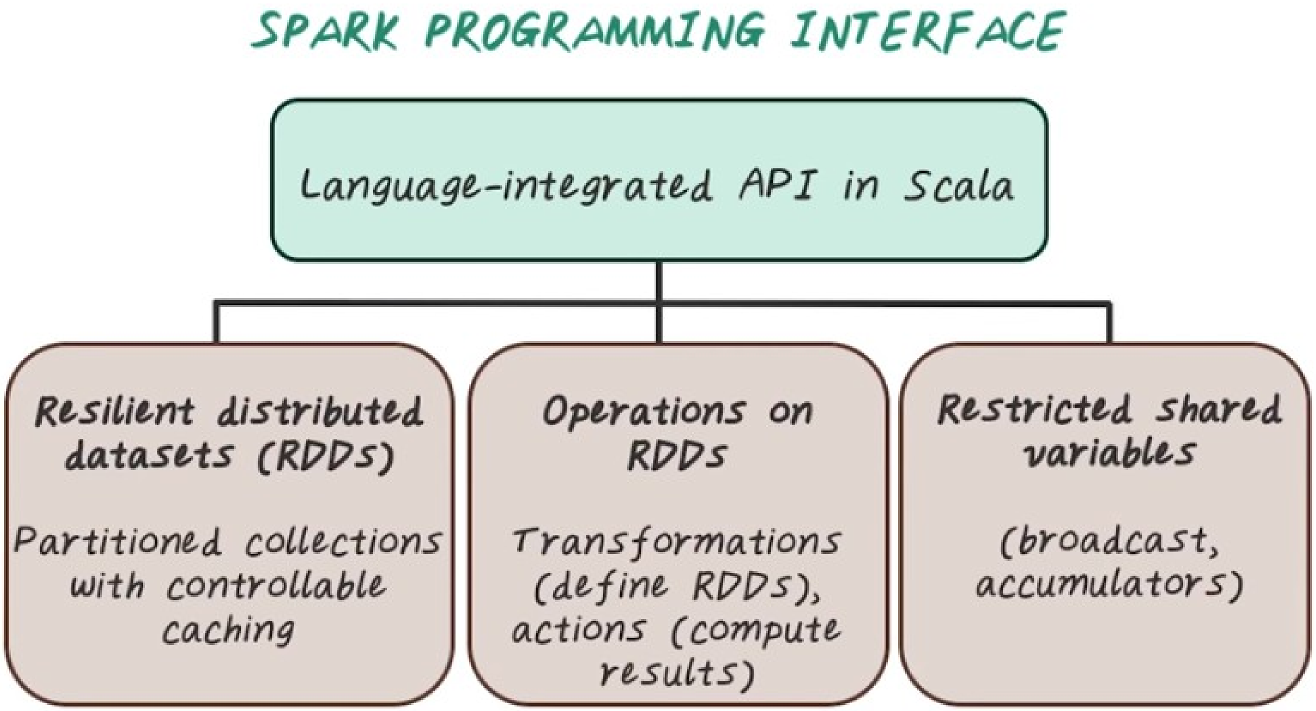
8. Next, let's illustrate how RDD can recover from failure. Again, we have these two different scenarios. This is for iterative algorithms and this is for interactive computation. It's quite obvious for the interactive computation. For example, the result from the second iteration failed. In this case, we only need to repeat the second iteration to regenerate the same result. If both results are failed in memory, we can repeat iteration one and two to recompute the result.



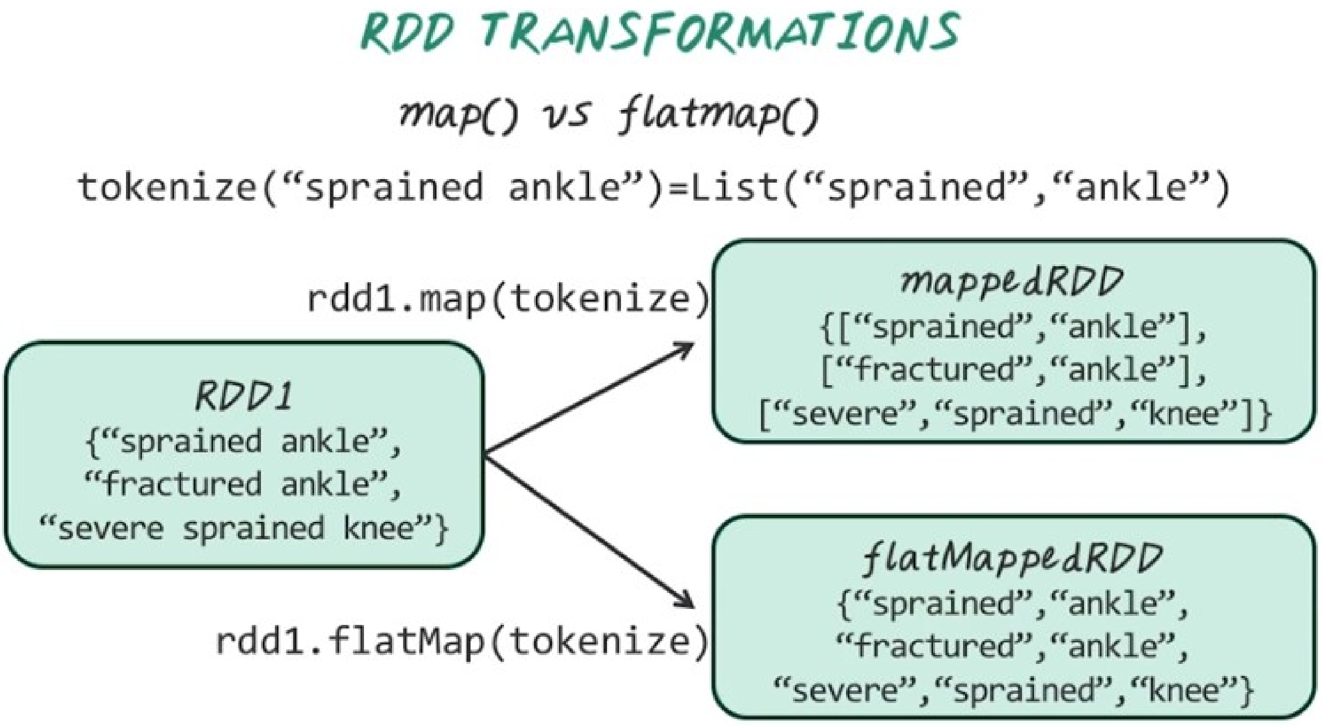
For iterative algorithms, if only part of data has failed in memory, we can load the corresponding chunk from the stable storage to refresh that memory. Also those results from each iteration are stored in memory. So if part of that is filled, like the latest one (指到 result 3 的), then we only need to repeat that from the previous one (指到 result 2 的) to regenerate the latest one. And this is the key idea behind spark, so spark is really a big data systems built on top of RDD.



9. Software stack for Spark already become quite rich. The Spark core contains the basic functionality of Spark including components for scheduling, memory management, fault recovery, and interacting with storage systems and more. And Spark Core is a home API for RDD and this RDD, as we just illustrated, is the main programming abstraction. And RDD are represented as a collection of data points distributed across many computer nodes, and can be manipulated in parallel. Spark Core provides many APIs for building and manipulating this RDD. Spark SQL is Spark package for working with structure data. It allows querying data via SQL like syntax. Spark streaming is a spark component that enable processing real-time data such as weblogs. Spark also has a machine learning library called MLib. MLib provide many machine learning algorithms including classification, regression, clustering, collaborative futuring, and so on. All this machine learning algorithm in MLib are designed to scale to a cluster of computers. GraphX is a graph processing engine for Spark. It can manipulate large graph such as social networks of friends, citation network of papers, publications, and patient and disease graph as well as all those medical oncologies. GraphX provide various operators on graphs, such as extracting sub graphs and map vertices. It also provide a library of common graph algorithms such as pagerank. Under the hood, Spark is designed to efficiently scale up from one machine to many machines. To achieve this flexibility, Spark can run over a variety of cluster managers such as Standalone Scheduler on a single machine, and Hadoop YARN, and Apache Mesos.

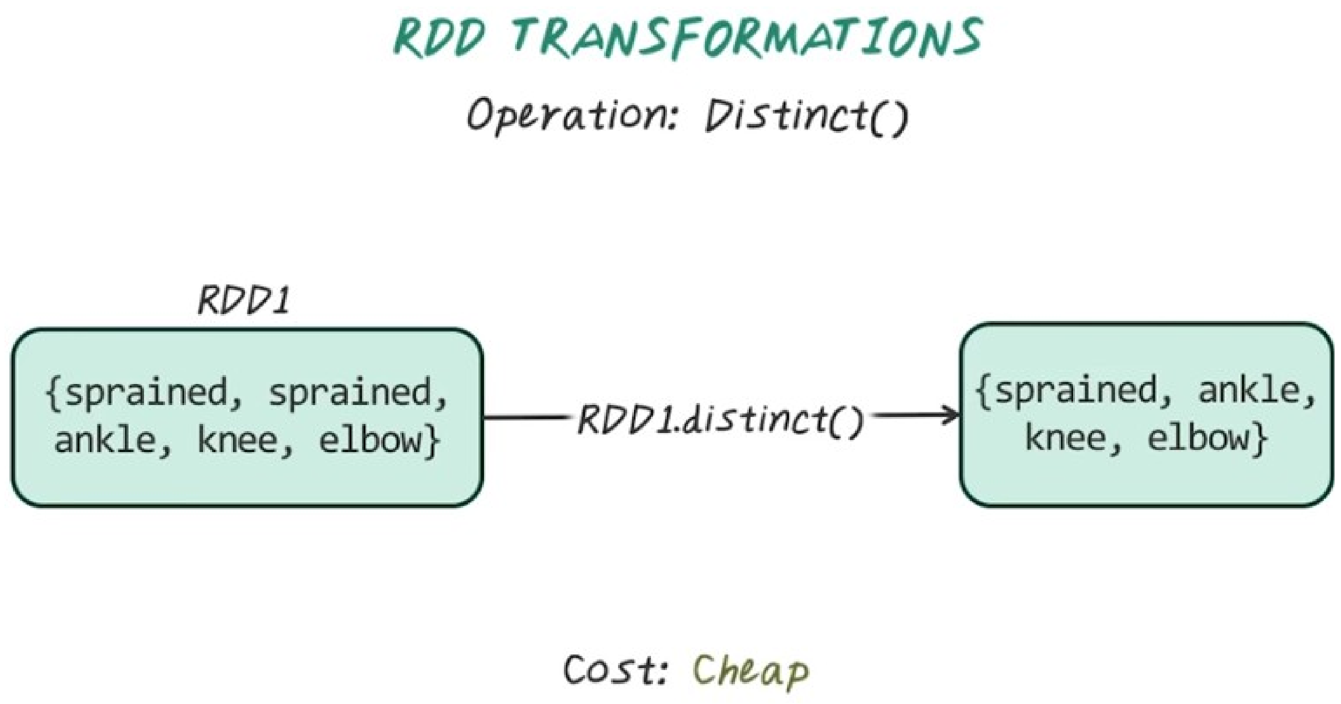

SPARK PROGRAMMING INTERFACE
Language-integrated API in Scala
Resilient distributed datasets (RDDs)
Partitioned collections with controllable caching
Operations on RDDs
Transformations (define RDDs), actions (compute results)
Restricted shared variables
(broadcast, accumulators)

10. Next, let's talk about programming interface for Spark. The core interface is written in Scala. It has several other different languages being supported such as Python and Java. There are three different types of API. There's one set of API is about creating and manipulating RDD. Then we have these different operations on RDD, such as transformation of one RDD to another, an action that has to operate on an RDD in order to compute some results. Spark also provide a way to share global variables across machines through this restricted share variable mechanism, such as broadcast and accumulator.

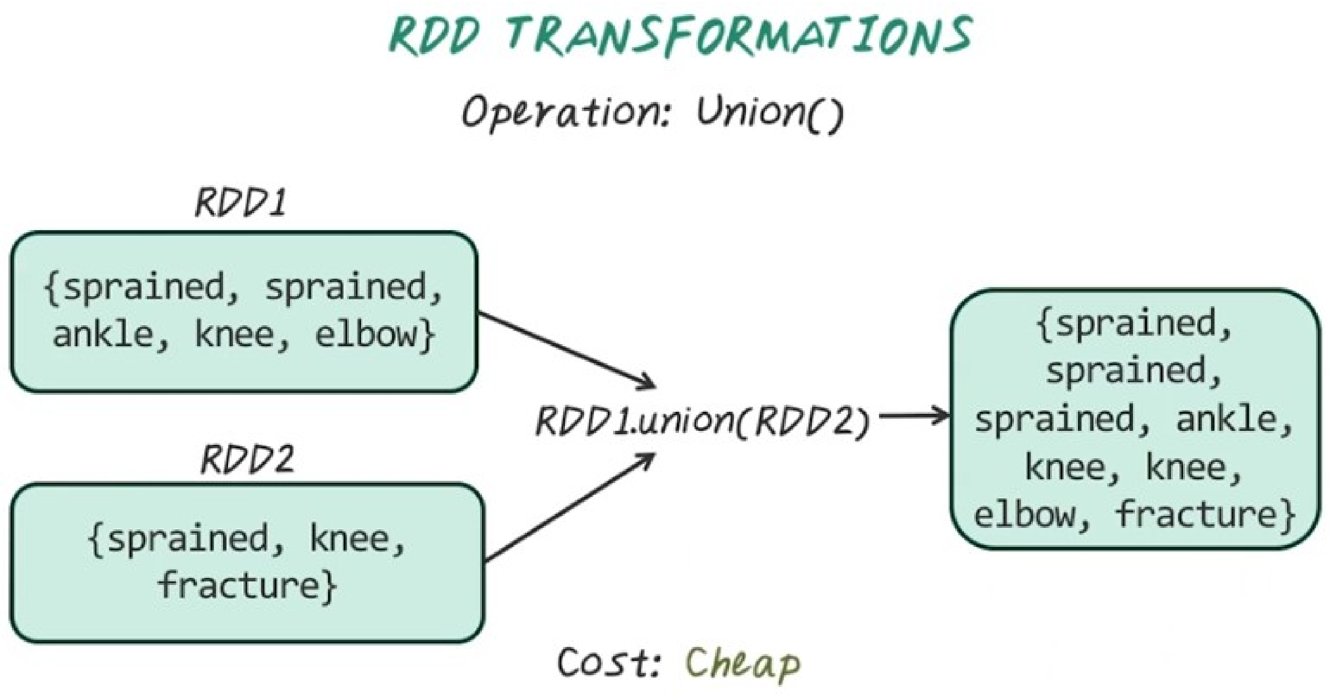


注意以上主要是 tokenize 這個函數的功能, map 只是去执行這個函數, 更詳細的見後面 RDD transformation 表.

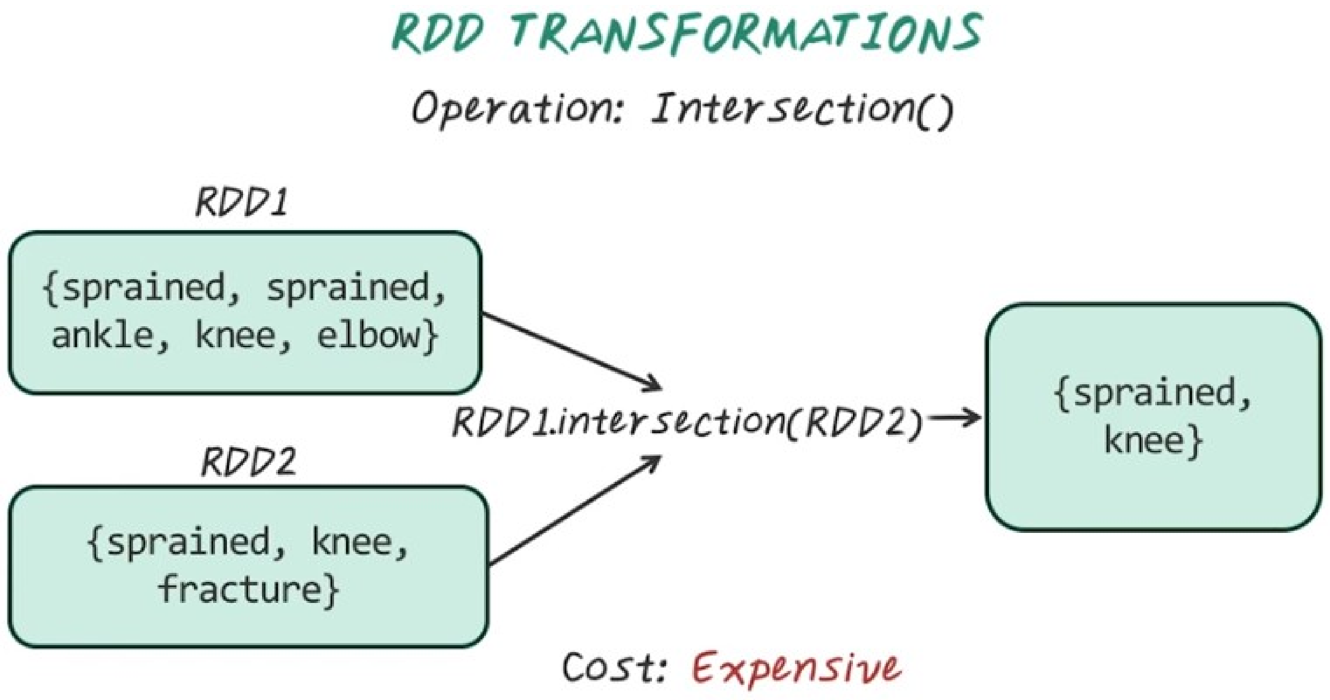
11. Next, let's look at some example of RDD transformation. Here's example illustrate map versus flatmap, given an RDD look like this we have three different type of symptoms, sprained ankles, fractured ankle and severe sprained knee. Then we want to apply this map function with this particular operation tokenize. For example when we tokenize this strain we'll get the list of words. For example the input is "sprained ankle" then output is two words, "sprained" and "ankle". And that's a list so if we applied this with the map function we'll have a set of list one for each element here. For example, sprained ankle ill become a list of two word, sprained and ankle. And fractured ankle will be another list, fractured and ankle. And the severe sprained knee becomes a list of three words severe, sprained, knee. So the result of map function become a list of lists. In many cases, we don't want this two-level structure. We want one level list with all those words in the same list. So in that case we can apply flatMap. The result of flatMap is a list of all those individual words. So essentially we're flattening the RDD from map function into this one level list or another way to say that is, we concatenate all those lists together. Make them a single list.


RDD TRANSFORMATIONS
Operation: Distinct()
RDD1
{sprained, sprained, ankle, knee, elbow}
RDD1.distinct()
{sprained, ankle, knee, elbow}
Cost: Cheap

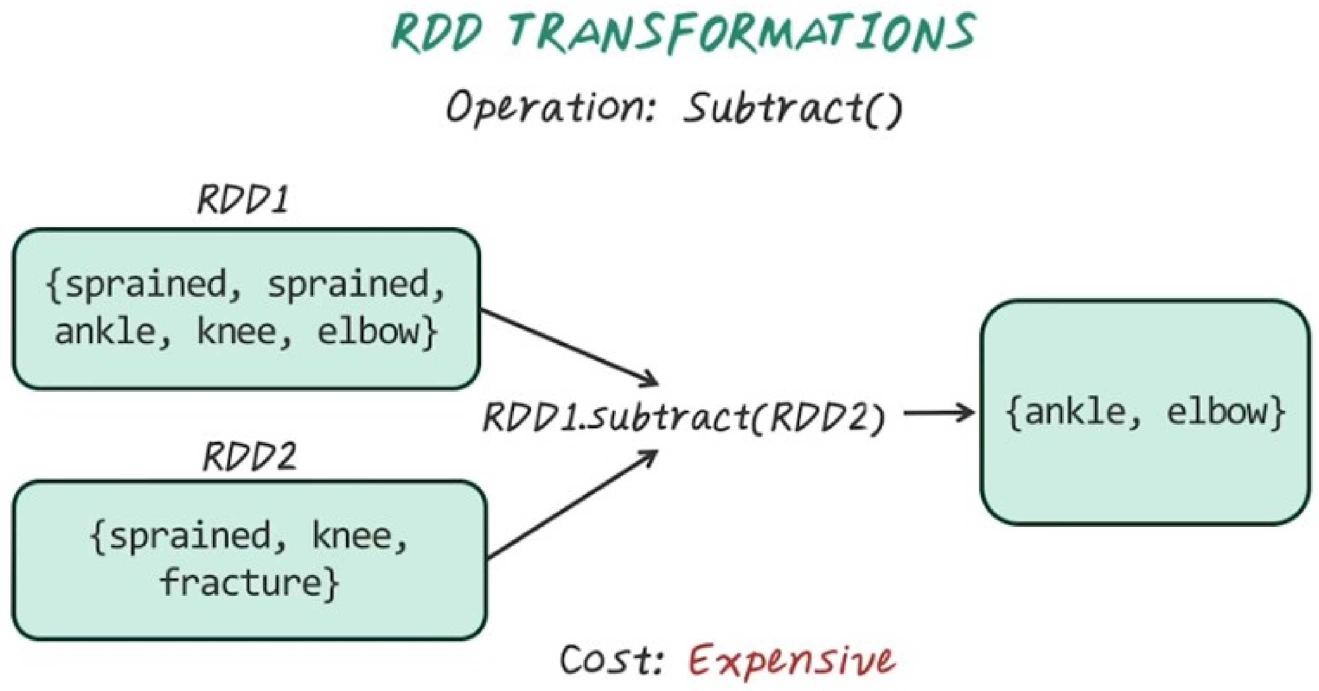
Now some more examples of RDD transformation. Now, given an RDD of words, if we apply distinct transformation, the result of that will be the distinct word in the original RDD. So sprained only show up once in the resulting RDD. The result on this operation is relatively cheap.


RDD TRANSFORMATIONS
Operation: Union()
RDD1
{sprained, sprained, ankle, knee, elbow}
RDD2
{sprained, knee, fracture}
RDD1.union(RDD2)
{sprained, sprained, sprained, ankle, knee, knee, elbow, fracture}
Cost: Cheap

Another commonly used operation is union. So union works with two RDDs. So now we have this RDD1, with the set of words. RDD2 with another set of words. Now, if we apply union, so RDD1.union(RDD2) So this will give us a resulting RDD that merged this two RDD together. In this operation is also relatively cheap, because it only involves [INAUDIBLE] these two already together. Notice that this is really a simple concatenate of two RDD together. The redundancy across this RDD still remains in this resulting RDD.


RDD TRANSFORMATIONS
Operation: Intersection()
RDD1
{sprained, sprained, ankle, knee, elbow}
RDD2
{sprained, knee, fracture}
RDD1.intersection(RDD2)
{sprained, knee}
Cost: Expensive

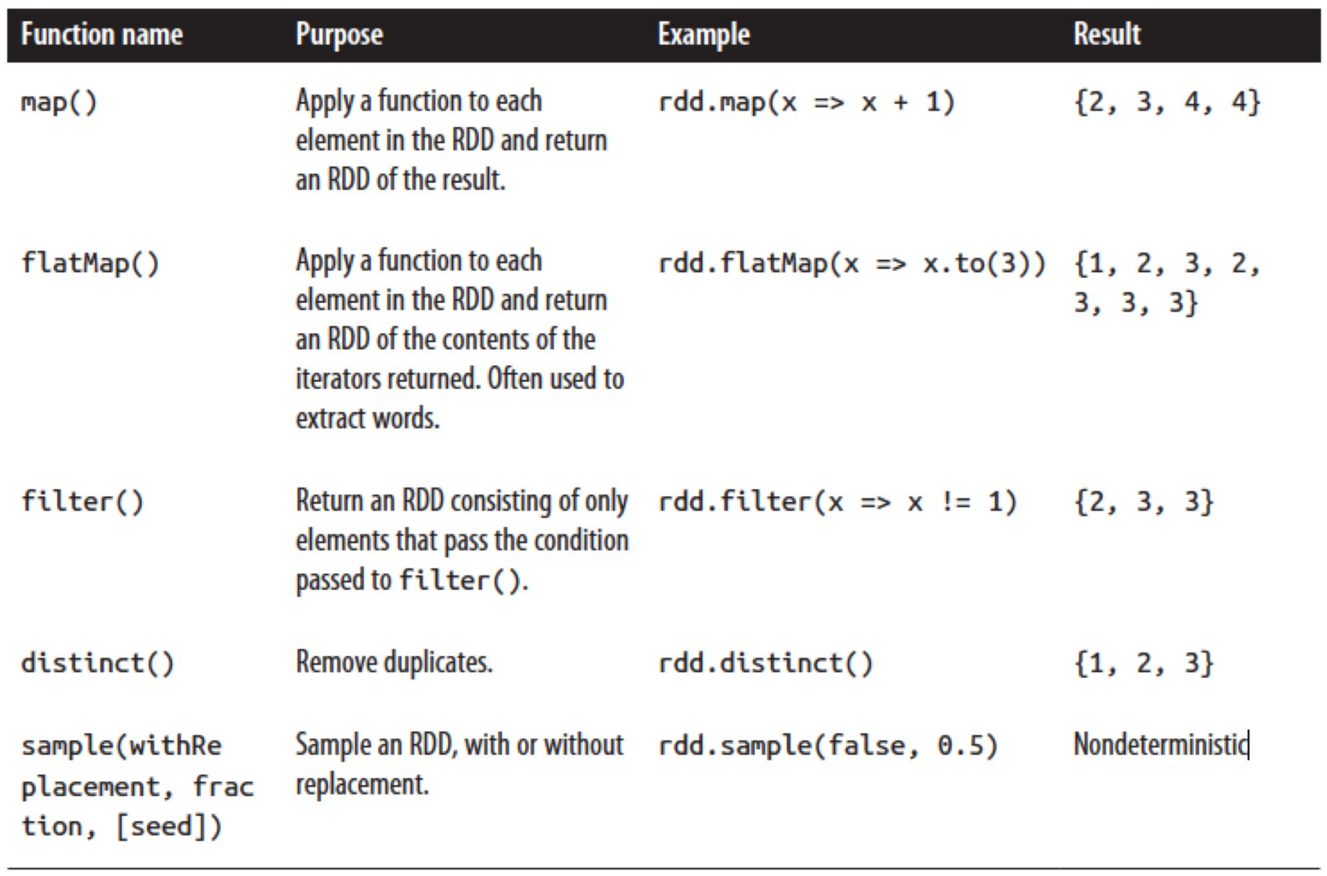
Other popular operations, such as intersection, is also supported. Now given these two RDDs, we want to find intersection between these two, and that will be the sprain and knee because they are in both RDD. This operation is more expensive because it involve sorting, refining out distinct values and finding out overlapping across two RDD. So this is more expensive to do.



Another transformation such as subtract. That's also a set operation. Given two RDDs, we want to remove the elements in RDD2 from RDD1. So RDD1.subtract(RDD2). So this gives us the resulting RDD, which is the remaining element in RDD1 that are not in RDD2. This operation can be expensive because we have to find the distinct elements in both RDDs, then perform a set difference operation.

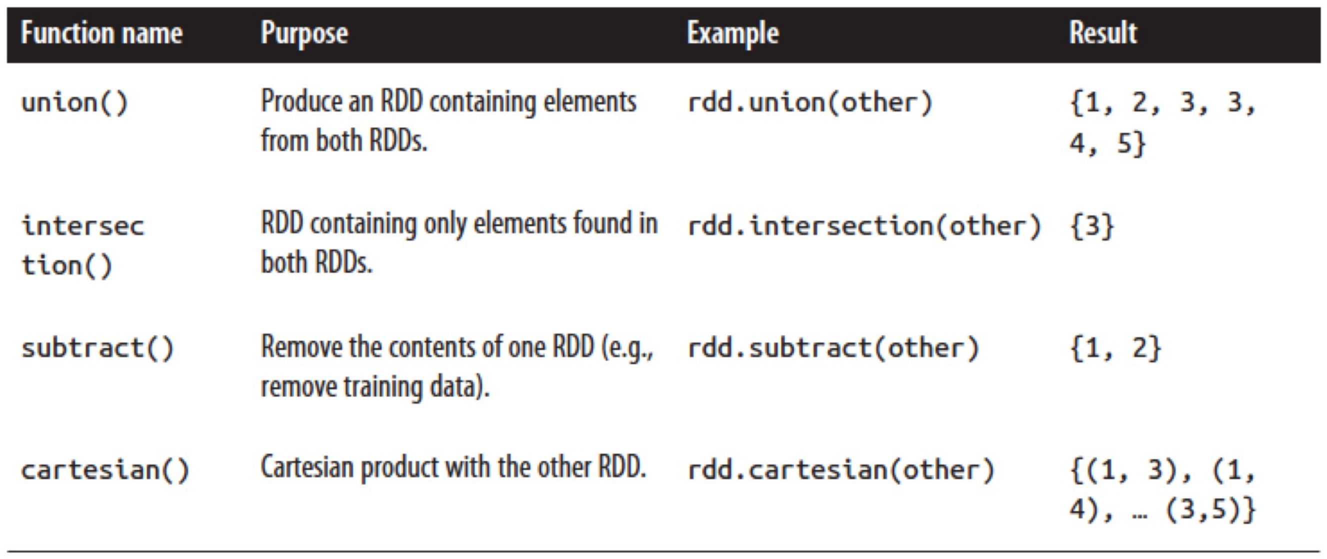
以下表是視頻中提供的, 該段只有這些表, 沒有說話.

Table 3-2. Basic RDD transformations on an RDD containing {1, 2, 3, 3}



Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

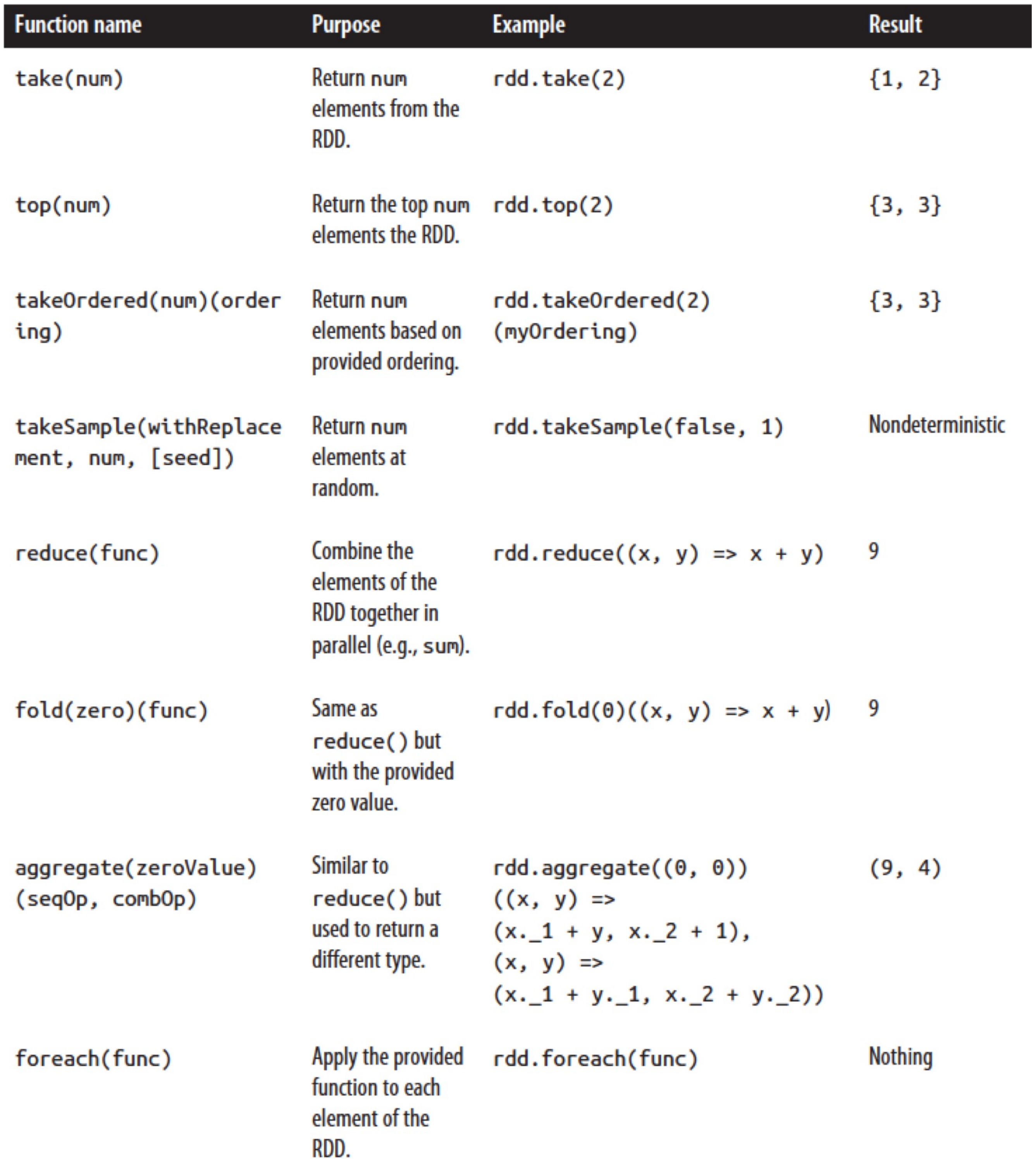
Table 3-3. Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}



Function name	Purpose	Example	Result
`union()`	Produce an RDD containing elements from both RDDs.	`rdd.union(other)`	`{1, 2, 3, 3, 4, 5}`
`intersection()`	RDD containing only elements found in both RDDs.	`rdd.intersection(other)`	`{3}`
`subtract()`	Remove the contents of one RDD (e.g., remove training data).	`rdd.subtract(other)`	`{1, 2}`
`cartesian()`	Cartesian product with the other RDD.	`rdd.cartesian(other)`	`{(1, 3), (1, 4), … (3,5)}`

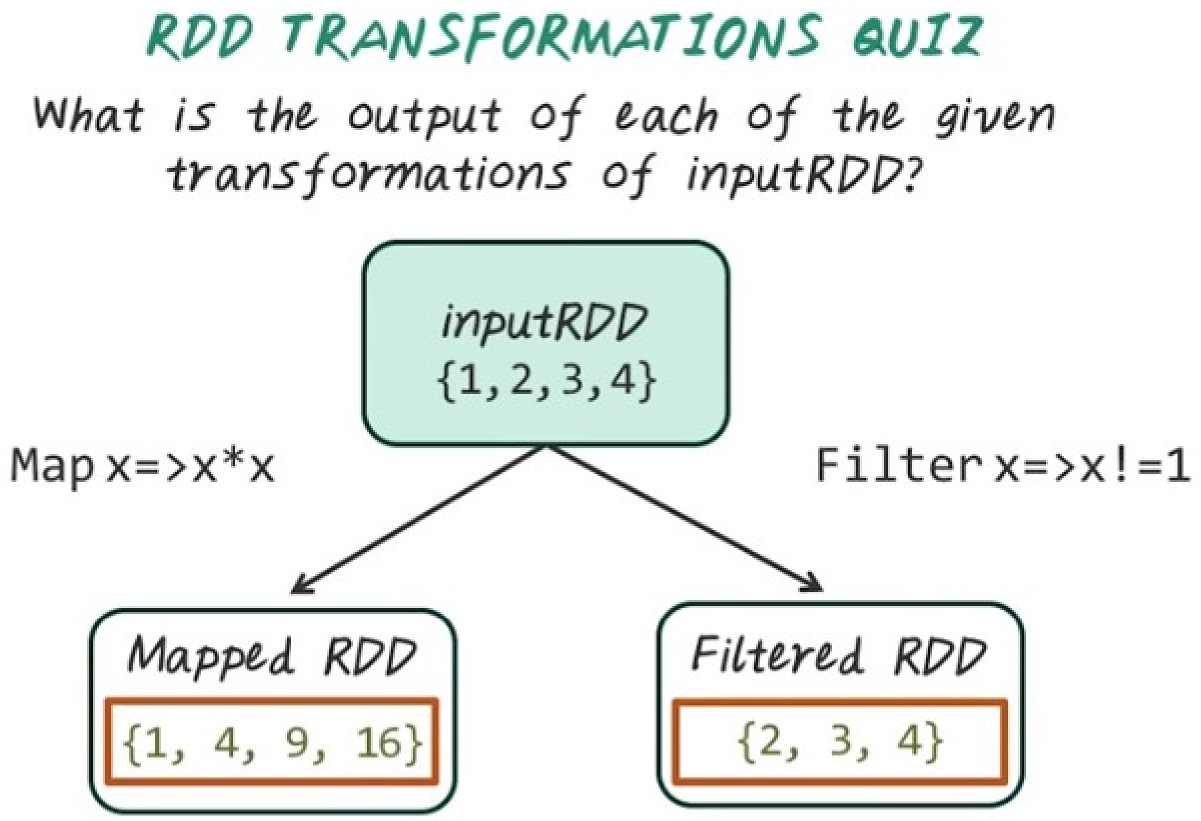
Table 3-4. Basic actions on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	`4`
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`

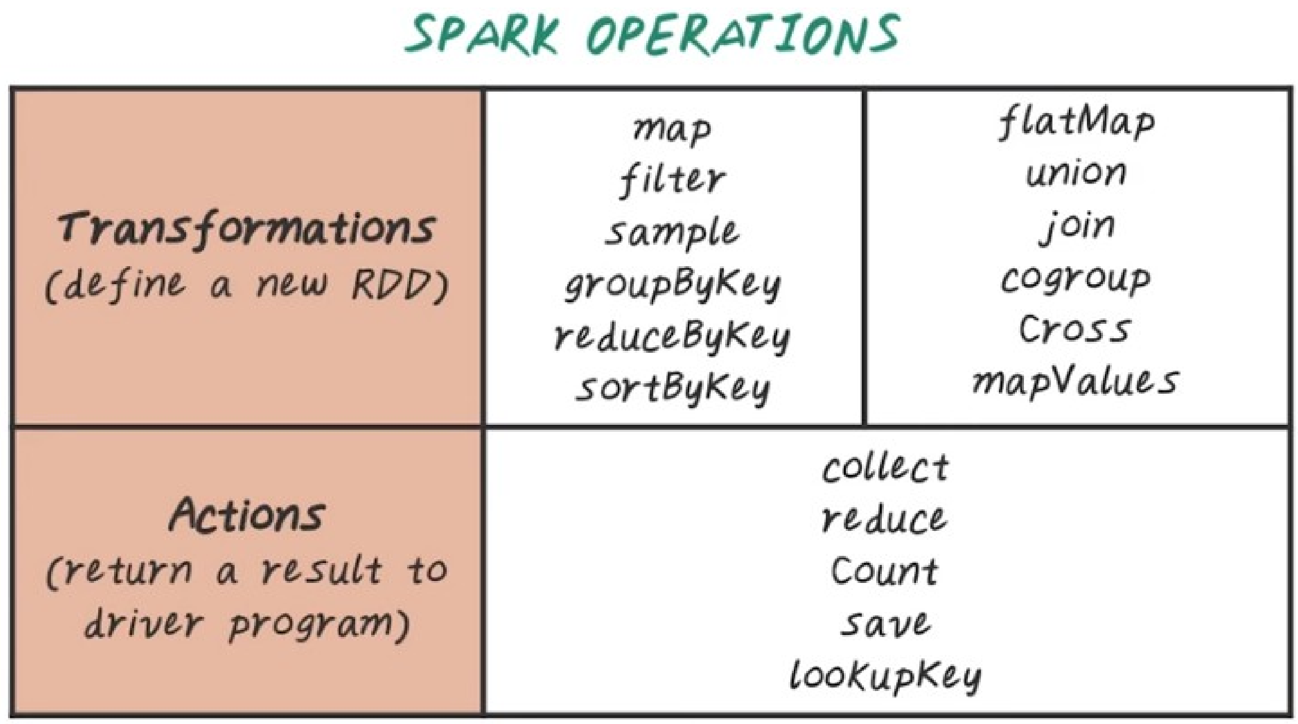


Function name	Purpose	Example	Result
`take(num)`	Return `num` elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top `num` elements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return `num` elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
`takeSample(withReplacement, num, [seed])`	Return `num` elements at random.	`rdd.takeSample(false, 1)`	Nondeterministic
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., `sum`).	`rdd.reduce((x, y) => x + y)`	9
`fold(zero)(func)`	Same as `reduce()` but with the provided zero value.	`rdd.fold(0)((x, y) => x + y)`	9
`aggregate(zeroValue)(seqOp, combOp)`	Similar to `reduce()` but used to return a different type.	`rdd.aggregate((0, 0))((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))`	`(9, 4)`
`foreach(func)`	Apply the provided function to each element of the RDD.	`rdd.foreach(func)`	Nothing

12. Now let's do a crisp on RDD transformation. So what is the output of each given transformation for this input RDD. This is the map function applying to x and return x times x. Then will have this filter function apply to x which was x equal to one. So write your result in this red boxes.



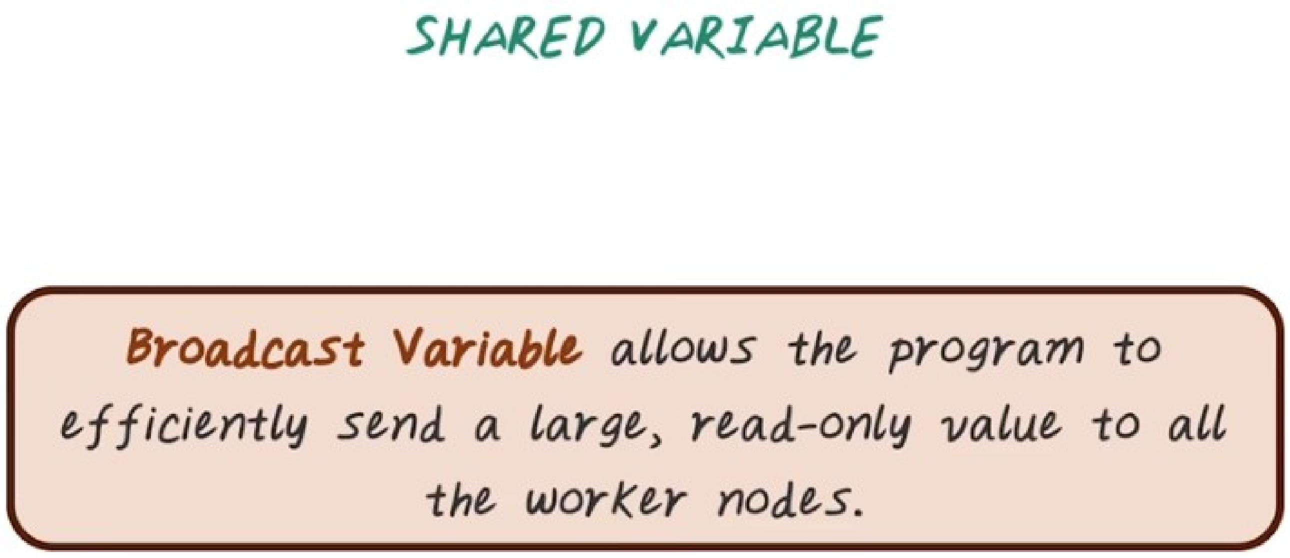
13. When we apply mapped function, this is the result. When we apply filtered function, in this case, we only return the value when they are not equal to one. We have value 2, 3 and 4.



SPARK OPERATIONS

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup Cross mapValues
Actions (return a result to driver program)	collect reduce Count save lookupKey	

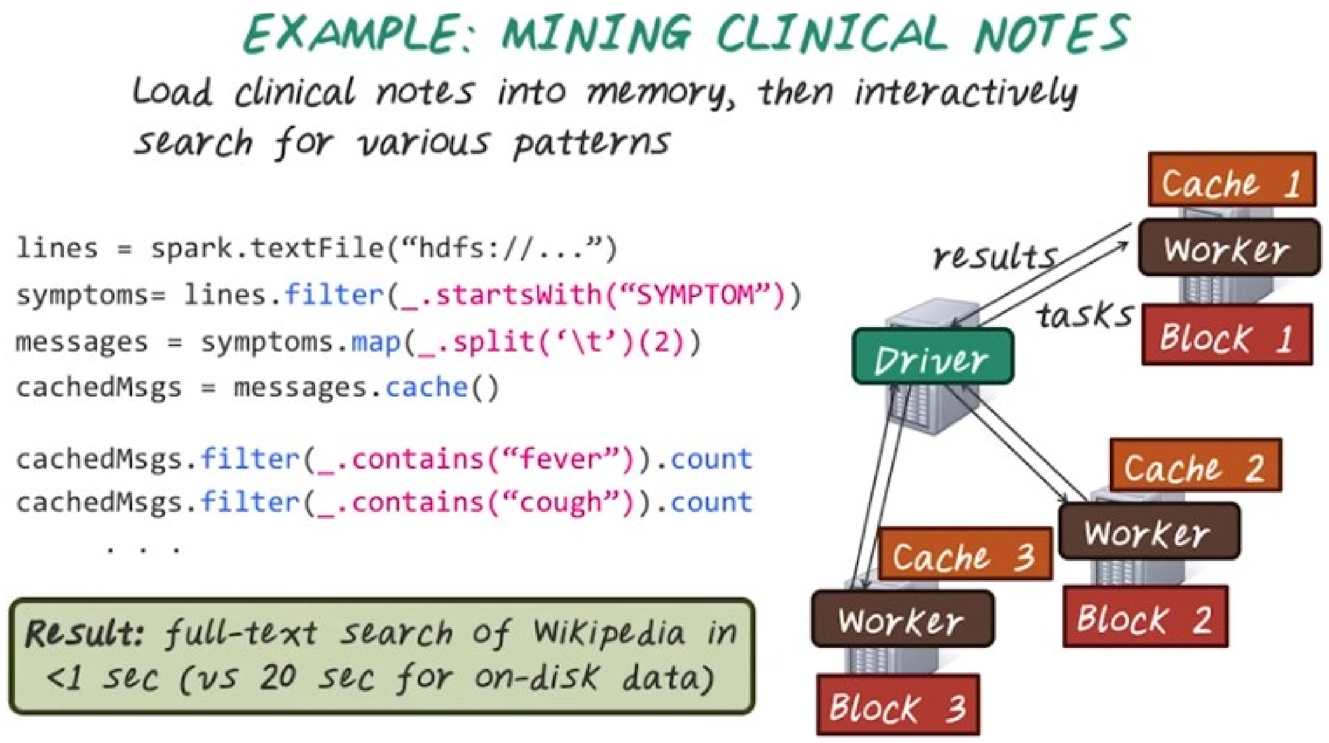
14. Spark provides many operations that categorize into these two groups, transformations, which define a new RDD from an input RDD, and actions, that return a result to the driver program. For example, for transformations, we talked about map, filter, and there are many more, such as sample, groupByKey, reduceByKey, sortByKey, flatMap, union, join, cogroup, Cross, mapValues. So it's a very rich set up of transformations, way beyond what MapReduce provide us in Hadoop. Same for Actions. Action is like the reduce phase in MapReduce. So it can collect the values from RDD. We can perform a reduce function, we can count how many records, we can save it somewhere else, we can look up by key to find a subset. These are example set of transformation and actions, but there are many more operations for Spark, and more example will be in the instructor notes.



SHARED VARIABLE

Broadcast Variable allows the program to efficiently send a large, read-only value to all the worker nodes.

15. Another important concept in Spark is this shared variable. The way to create a shared variable is to use this broadcast variable that allows us to efficiently send a large, read-only values to all the worker nodes.



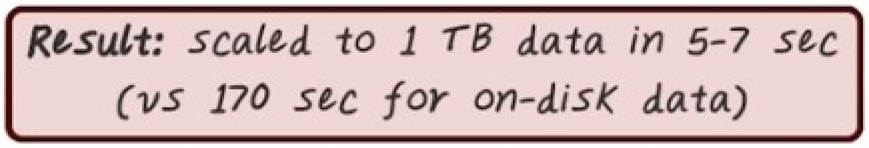
EXAMPLE: MINING CLINICAL NOTES

Load clinical notes into memory, then interactively search for various patterns

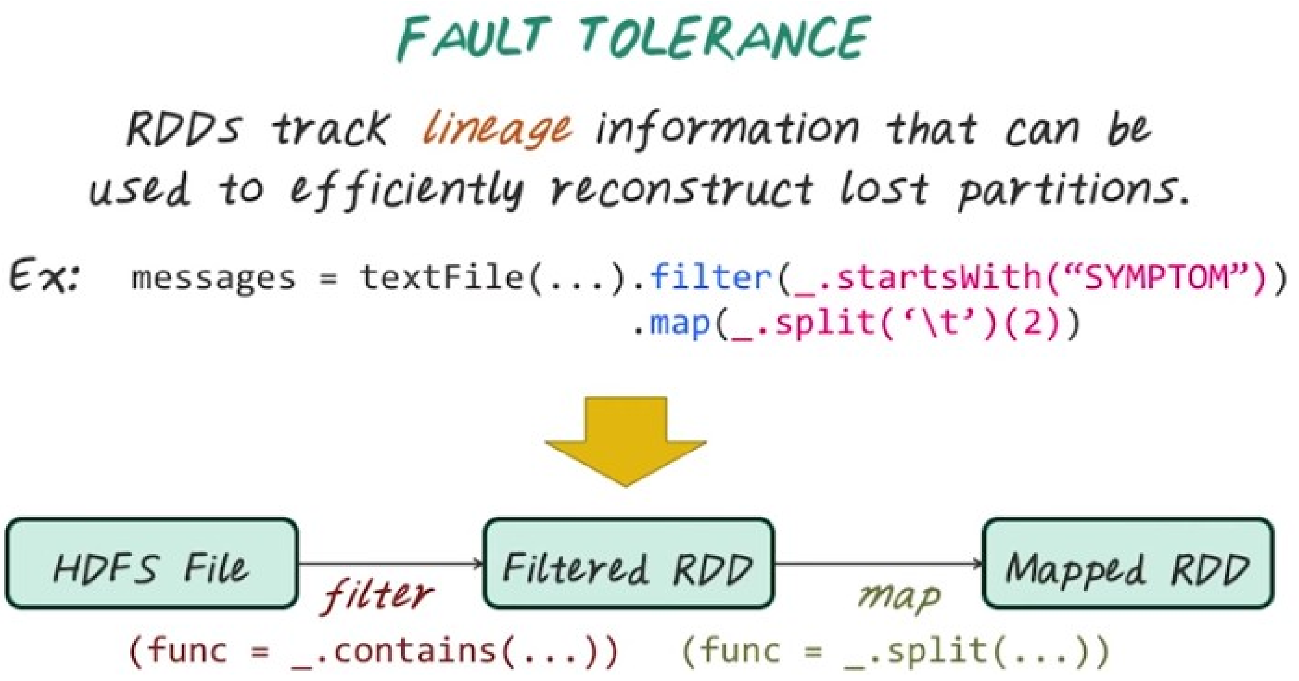
```
lines = spark.textFile("hdfs://...")
symptoms= lines.filter(_.startsWith("SYMPTOM"))
messages = symptoms.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("fever")).count
cachedMsgs.filter(_.contains("cough")).count
      . . .
```

Result: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

Next, let's illustrate a Spark job using this example. Say we have a large volume of clinical nodes. We want to find certain patterns. So Spark job runs on a clusters. You have two sets of nodes. The driver which is like MapReduce master. And workers, which just like MapReduce slaves. The drivers coordinate the entire task and the workers perform all those individual computation. Here are some example of a Spark code. First step, we want to load this clinical node from hdfs. So we load all the lines in this clinical nodes. So that's our base RDD. Next we want to filter all those lines to find the line start with keyword SYMPTOM. This will become the transformed RDD. Then we want to split the symptom lines with deliminator tab. Then find the third element with this parameter 2. For example, in this case we'll assume the third element corresponding to the symptom name. So that's what we want. Since we know we're going to find various patterns on those symptom names, so we cache this result. So far everything runs on driver and no computation has really happened, until we reach an action. Here is an action. We want to filter all these symptom names to find the keyword, fever, and we want to count those lines. So we want to know how many times symptom fever occurs in this case. So this is an action and they will happen. When we reach this action line, the driver will send those tasks to individual workers to work on their own block (後面說了: the cache will be created on all those worker nodes). Then the worker will go through the steps to try to find fevers and count the number of times they occurred. And the result will be sent back to the driver node, and that's the total count for fever. At the same times, the cache will be created on all those worker nodes. So next time we don't have to do all the beginning operations, because the result are cached, so the next line is another action. Here we'll look for a different symptom, cough, and we'll want to count how many times they occur in the clinical nodes. In this case we send the task, again, to the workers, and they already have the cache. So they don't have to go through the raw data again. So what they need to do is continue from this cached line to compute the number of times cough occurred. And the result will be returned back to the driver and the process continued. So this is an illustration of how Spark job works. Spark can give quite amazing results because this in-memory operation. For example, full-text search on Wikipedia takes less than 1 second versus 20 seconds if it's read from disk.


Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

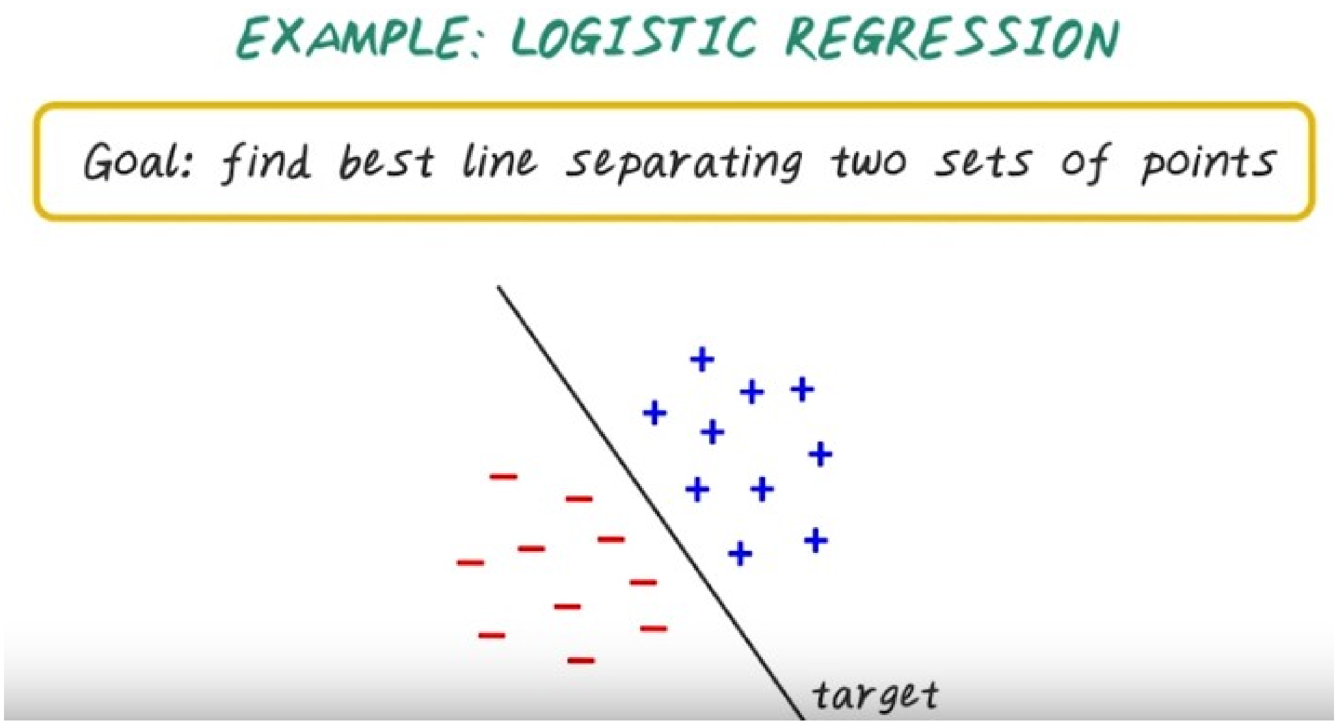
With Spark, we can scale to 1 TB data in 5-7 seconds versus 170 seconds just to read from disk. So the reason we can get this 5-7 second near real time response on a huge data set is because now we can read and processing data in parallel. And also cache the result in memory whenever needed.


FAULT TOLERANCE

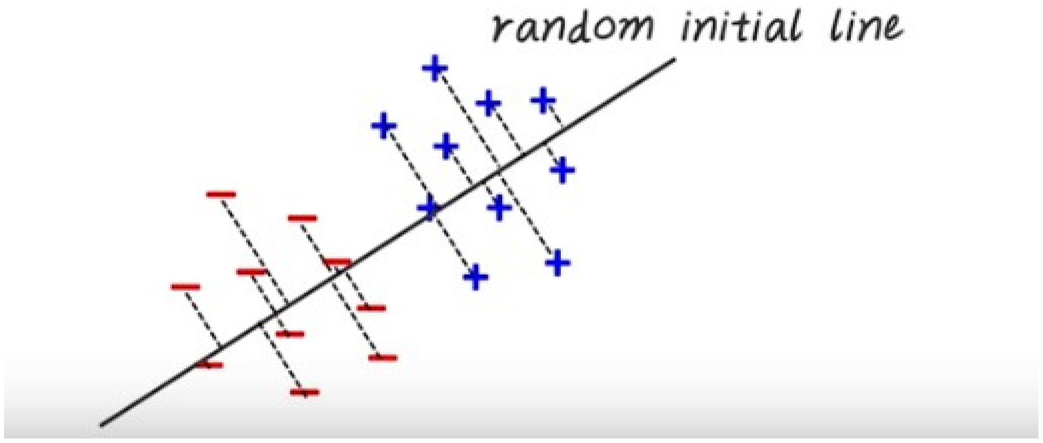
RDDs track *lineage* information that can be used to efficiently reconstruct lost partitions.

Ex:
```
messages = textFile(...).filter(_.startsWith("SYMPTOM"))
                        .map(_.split('\t')(2))
```

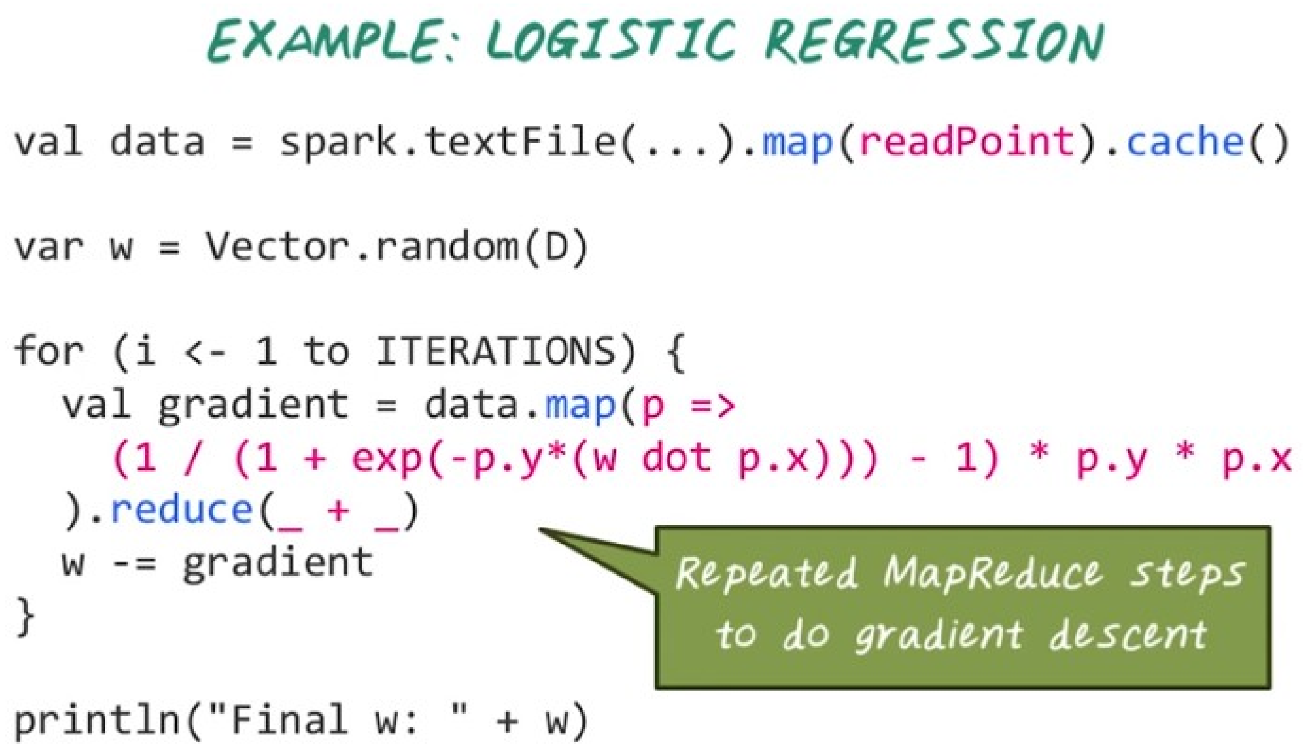
16. Now, let's illustrate how fault tolerance is enabled using RDD. So, RDD tracks that lineage information of all those different transformations, and they can recompute efficiently if lost partition happen. In the previous example, we have seen that this sequence of transformation help us to find the symptom names. So, we first filter to find the line contain symptom then split and extract the individual elements which is the symptom name. So, visually, what happens is we're first read a file from HDFS, perform filter operation, then we have this filtered RDD. Then, the filtered RDD will be the input to the map function to generate the mapped RDD. For example, if part of the result from the filtered RDD is lost, we can recompute very quickly from the previous step. By the way, all those transformation are tracked because they're the lineage information for us to reconstruct the RDDs we need if lost partition happen.


EXAMPLE: LOGISTIC REGRESSION
Goal: find best line separating two sets of points
target

17. With Spark, now we can efficiently support iterative algorithms such as machine learning algorithm like logistic regression. So giving logistic regression as a classification algorithm, trying to find the best line to separate these two sets of points. And this is the target we want to learn.



We start with this random initial line. First, we compute a gradient over all the data points, then we update this initial line by moving towards the gradient. Then we recompute the gradient again on all the data points, update the line, and do this update again, again, and again (就是那條線不停地轉, 直到轉到正確的方向), until it converges. That's how we get the final target line. Because the input and output are all capped in memory. This update can be done very quickly.

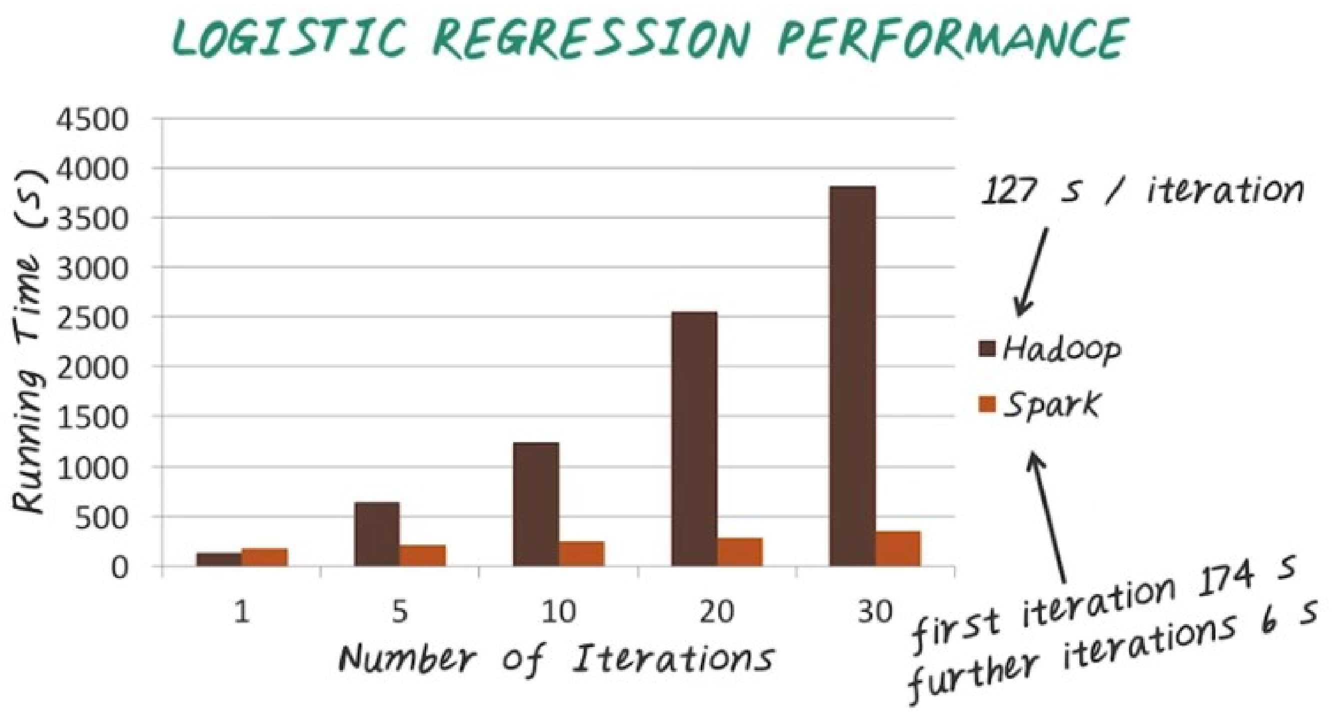


EXAMPLE: LOGISTIC REGRESSION

```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```

Repeated MapReduce steps to do gradient descent

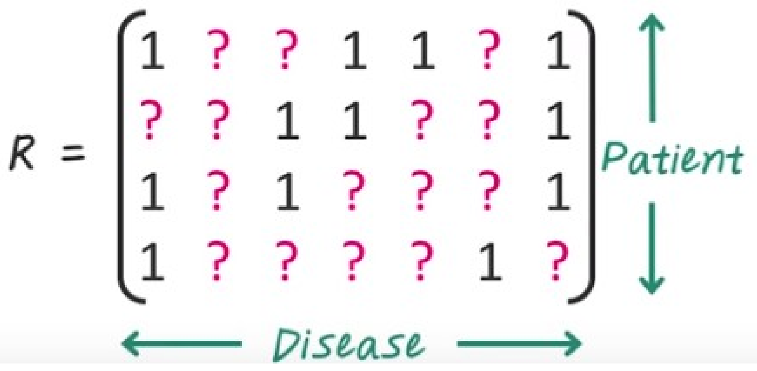
Here's example of spark code for writing logistic regression. We start by loading the data from disk into memory and cache that. Then we initialize the model parameters w. Then we perform the following iterations. Then we complete the gradient over the entire data set, which is really involve a map function over all the data points. And for each data points, we perform this calculation, then perform the reduce function. This is really to sum them up, that give us the gradient. Notice that with this one simple operation, we performed a MapReduce function to compute the gradient. Once we have the gradient, we can update the model parameters then repeat this process. Finally, we have the final parameters once it's done the all the iterations.



Here is a performance comparison between Hadoop and Spark for computing logistic regression. So X axis indicates the number of iterations from first iteration to 30th iteration. So Y axis is total running time. And in this case, the lower the better. Notice that every iteration of Hadoop takes about 127 second. Notice that for Spark the first iteration is even longer than Hadoop because the caching operation. Then the further iterations for Spark only take 6 seconds as opposed to over 100 seconds. That's why Spark's in total running time is much lower than Hadoop.

EXAMPLE: DISEASE RISK PREDICTION

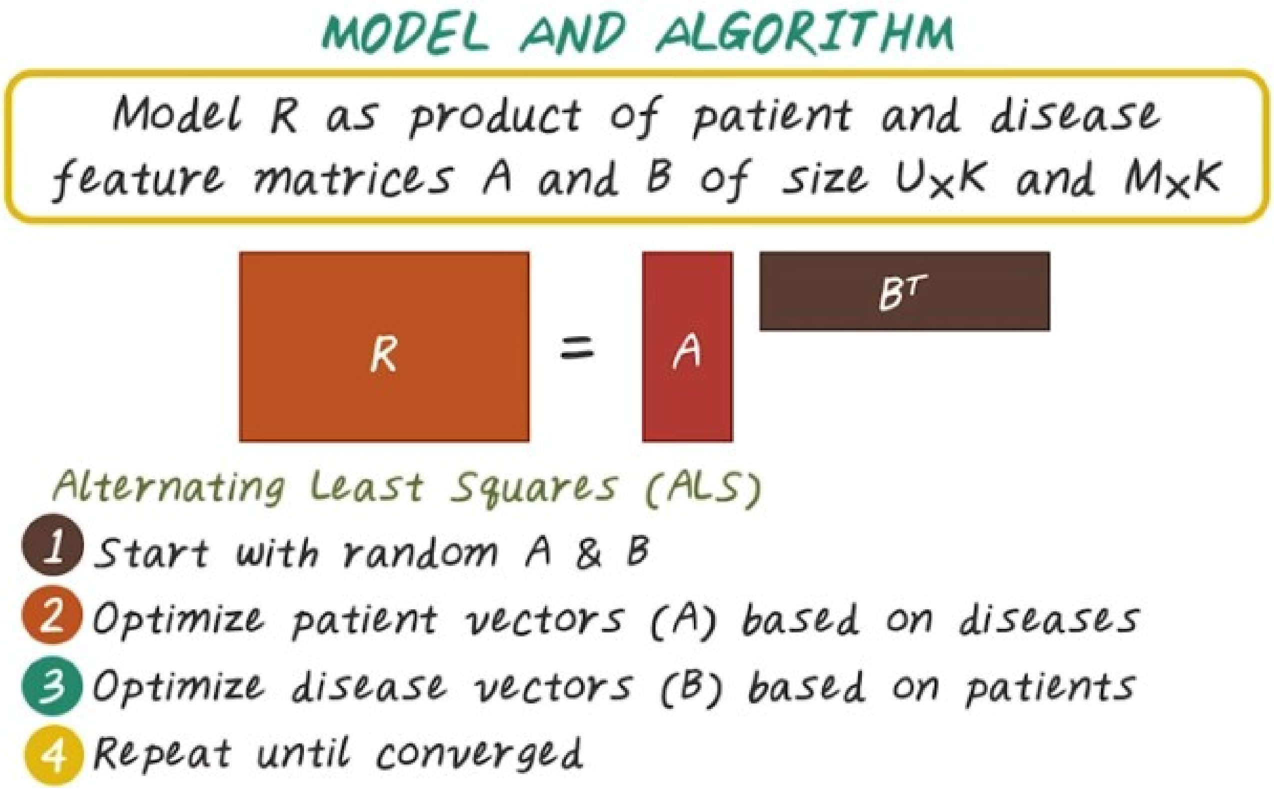
Goal: predict disease risk based on existing disease diagnosis



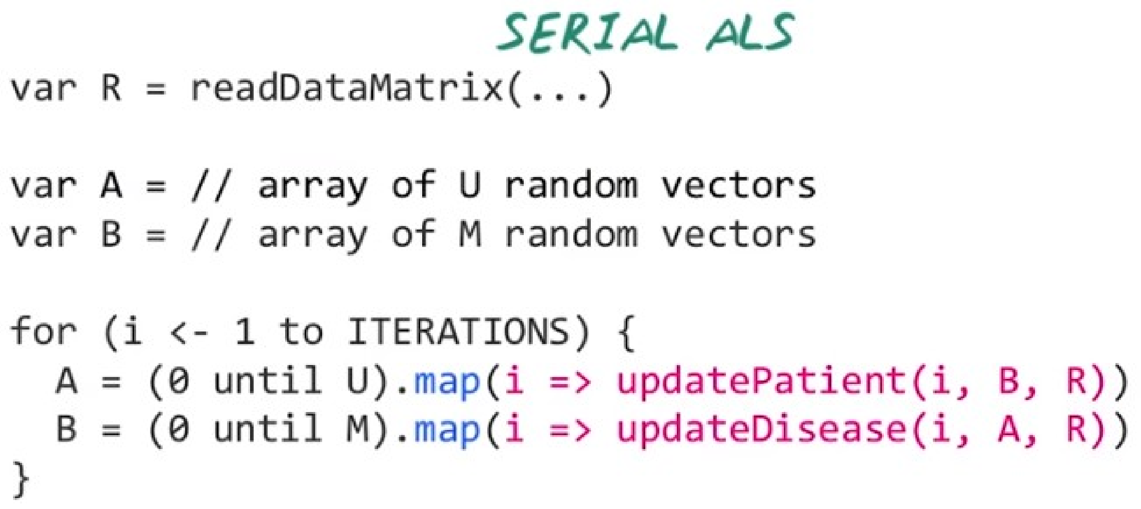
$$R = \begin{pmatrix} 1 & ? & ? & 1 & 1 & ? & 1 \\ ? & ? & 1 & 1 & ? & ? & 1 \\ 1 & ? & 1 & ? & ? & ? & 1 \\ 1 & ? & ? & ? & ? & 1 & ? \end{pmatrix}$$

18. Next, let's illustrate Spark with another health care example. Say we have matrix R which is the patient by disease. Every row corresponding to a patient, every column corresponding to a disease.

And the goal here is to predict the disease risk based on existing disease diagnosis. For example, we know for a patient what disease they already have, but for the ones they don't have, we want to assess the risk. So those are all those question marks.


MODEL AND ALGORITHM
Model R as product of patient and disease feature matrices A and B of size UxK and MxK
R
=
A
B^T
Alternating Least Squares (ALS)
1 Start with random A & B
2 Optimize patient vectors (A) based on diseases
3 Optimize disease vectors (B) based on patients
4 Repeat until converged

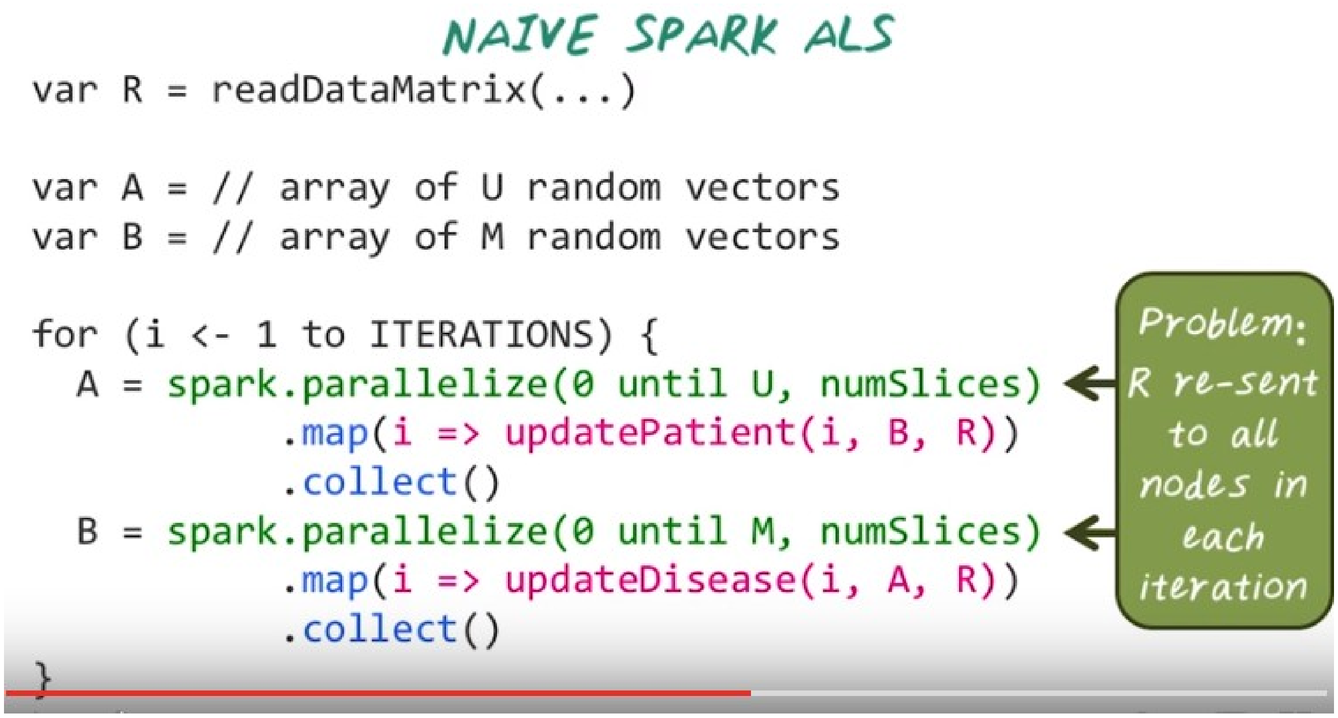
So how do we model that problem? This is really a collaborative filtering problems. We can model a large matrix R with many missing values with a product of two matrices, A and B. A corresponding to all the patient features, and the B corresponding to all the disease features. So the way to do that is through this alternating least squares operation, ALS. So we'll fix one of this matrix. For example we'll fix B then update A, then we can fix A update B. So this is called alternating least squares. So the algorithm goes as follows, we start with a random initialization of A and B. They'll start to optimize the patient feature vectors, A, based on the disease feature vector, B. Then we do that for the disease feature vectors based on the patient feature vectors. Then we repeat this process until convergence. So if we want to write this using Spark, it's actually quite easy to do.



SERIAL ALS

```
var R = readDataMatrix(...)

var A = // array of U random vectors
var B = // array of M random vectors

for (i <- 1 to ITERATIONS) {
  A = (0 until U).map(i => updatePatient(i, B, R))
  B = (0 until M).map(i => updateDisease(i, A, R))
}
```

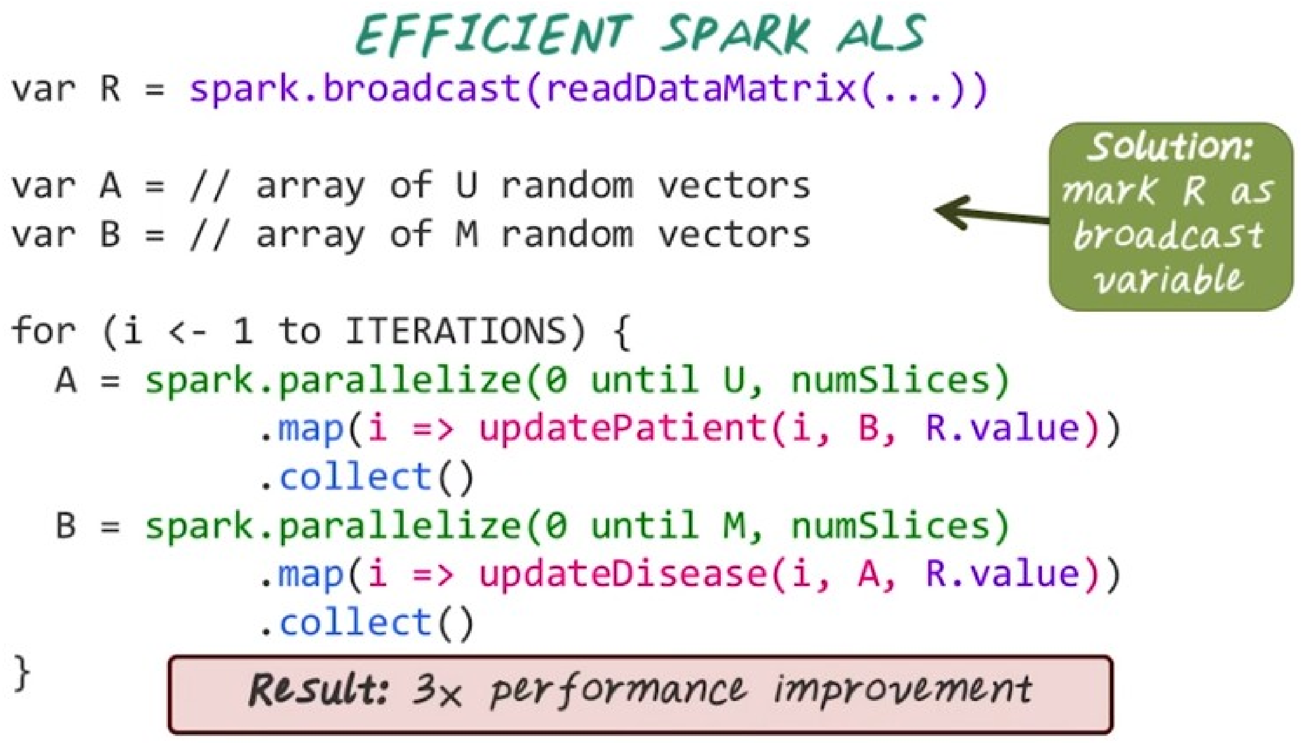
19. Next, let's talk about how do we do this using Spark. So, we can right this simple 0 implementation of alternating least square. First, we can read the data matrix R. Then, we can initialize A and B randomly. After that, we can start this iterative operation. Here we want to update A and B alternatively. For example we have U patients, for each patient I will update that corresponding patient based on the matrix B and input matrix R. Say we have a function for doing that. This is actually quite straight forward because it's just simple B square problem can be solved with a linear regression operation. That is just to update for patient I. We want to update A and B alternatively. So these are the range values. So for A we update each patient from 0 to U minus 1, and for B we update each disease from 0 to M minus 1. So, for each patient we want to update the patient based on the input matrix, R. And the disease matrix speed. So this really becomes really a regression problems. And we want to do this for every patient. So that's where this map function comes in. For each patient I, from zero to U minus 1 we apply this function. Similarly, we can do that for disease. For each disease, we update the disease based on input data matrix R, and patient matrix A. So next, let's see how we can do this in parallel.



NAIVE SPARK ALS

```
var R = readDataMatrix(...)

var A = // array of U random vectors
var B = // array of M random vectors

for (i <- 1 to ITERATIONS) {
  A = spark.parallelize(0 until U, numSlices)
           .map(i => updatePatient(i, B, R))
           .collect()
  B = spark.parallelize(0 until M, numSlices)
           .map(i => updateDisease(i, A, R))
           .collect()
}
```

20. Again, we read the input data, and we initialize A and B, then here, we can run all of those update for each patient in parallel. Then, collect those result to the driver node. So, spark.parallelize provide the way for us to perform this parallel computation. And similarly we can perform this parallelized operation on diseases. For each disease, we want to update that as well. And the problem here is the big data matrix R has to be sent to all the node in each iteration. This is very inefficient if we have a large data matrix.



EFFICIENT SPARK ALS

```
var R = spark.broadcast(readDataMatrix(...))

var A = // array of U random vectors
var B = // array of M random vectors

for (i <- 1 to ITERATIONS) {
  A = spark.parallelize(0 until U, numSlices)
           .map(i => updatePatient(i, B, R.value))
           .collect()
  B = spark.parallelize(0 until M, numSlices)
           .map(i => updateDisease(i, A, R.value))
           .collect()
}
```

Solution: mark R as broadcast variable
Result: 3x performance improvement

So this is where we use broadcast. Instead of reading data directly, we can use spark.broadcast to create this read only object. Then when we want to use this matrix, we can do R.value to access that particular variable. It has been shown that we can achieve three times performance improvement by simply doing broadcast.