We now move on to discuss another distributed analytics engine that is growing in popularity and adoption.
Studies reveal that while MapReduce provides a good fit for a wide array of large-scale problems, it is ill-suited for distributed graph processing and other applications which are iterative in nature. These applications pose challenges due to the problem scale (normally billions of vertices and trillions of edges), poor locality, little data and work per vertex, and changing problem scale over the course of execution (when the graph rises or shrinks over time). Problems such as web graph analytics, social network processing, disease outbreak path identification, transportation route calculation, and newspaper article similarity, among others, involve distributed processing of large-scale graphs.
This module covers the Apache Spark framework. Spark is an open source cluster computing framework developed at the UC Berkeley AMPLab. It uses in-memory primitives that allow it to perform over 100x faster than traditional MapReduce for some applications.
Spark was optimized towards three classes of parallel distributed applications:

- Iterative: such as most machine learning tasks that iterate over a training data set until convergence is met
- Interactive: where low-latency results are expected after rapid querying
- Streaming application: where input data is continuously arriving from one or a few streams leading to an update of the stored state

Several libraries have grown around Spark, allowing fast execution of SQL-like queries, Machine Learning and Graph Computation applications. In this module, we cover the core Spark framework and some parts of the Spark ecosystem. We start with an overview of Spark's architecture, discuss the lifecycle of Resilient Distributed Datasets (RDDs) which provide a distributed abstraction to express in-memory computations, delve into fault

tolerance and recovery and finally look at several libraries for different types computation on Spark.

Motivation

MapReduce and similar data-parallel frameworks fail to express many types of distributed applications efficiently. Spark [1] targets a specific subset of these applications- those that reuse a working set of data across multiple rounds of computation. These applications fall in one of three categories:

- Iterative jobs: Many algorithms (for e.g. most Machine Learning) fall in this category. Although MapReduce can express such computation, each job requires that the data be reloaded from disk, leading to a significant performance penalty.
- Interactive jobs: Hadoop allows ad hoc queries to be performed on data using tools like Pig and Hive, which allow users to run MapReduce jobs using simple SQL interfaces. Unfortunately, these incur a very high latency because each job requires loading the entire dataset from disk. A faster querying platform is desirable.
- Streaming jobs: Models that require periodic updates, such as incremental processing systems that periodically renew the stored parameters based on new inputs.

Generally speaking, MapReduce lacks a data sharing abstraction for leveraging distributed memory. Such an abstraction would allow many applications to have concurrent data access to memory across the cluster. It also suffers from many performance problems due to its inefficient use of resources (e.g. poor memory utilization by spilling to disk after each job).
One of the frameworks attempting to address these issues is Spark. Spark relies on a special abstraction called **resilient distributed datasets** (RDDs) [2] to support iterative, interactive

and streaming applications. Before discussing RDDs, we first present a high-level overview of the Spark framework.

An Overview of Spark

The goal of any distributed programming framework is to support the execution of a parallel computation across multiple nodes in a performant manner. Consider an iterative application that runs a machine learning algorithm on a large graph. Spark would store this graph as a **Resilient Distributed Dataset (RDD)** (Figure 5.32). The Spark Client would store the details of the program to be executed and map it to Spark-specific operations for a cluster, which comprises of many workers. There is a cluster manager that converts these operations into tasks and executes them on the worker nodes. Any cluster requires applications to be scheduled well to maximise the utilization and improve performance. Spark allows different policies to be used to schedule tasks on the cluster depending upon factors such as the priority, duration, and resources required by each task.

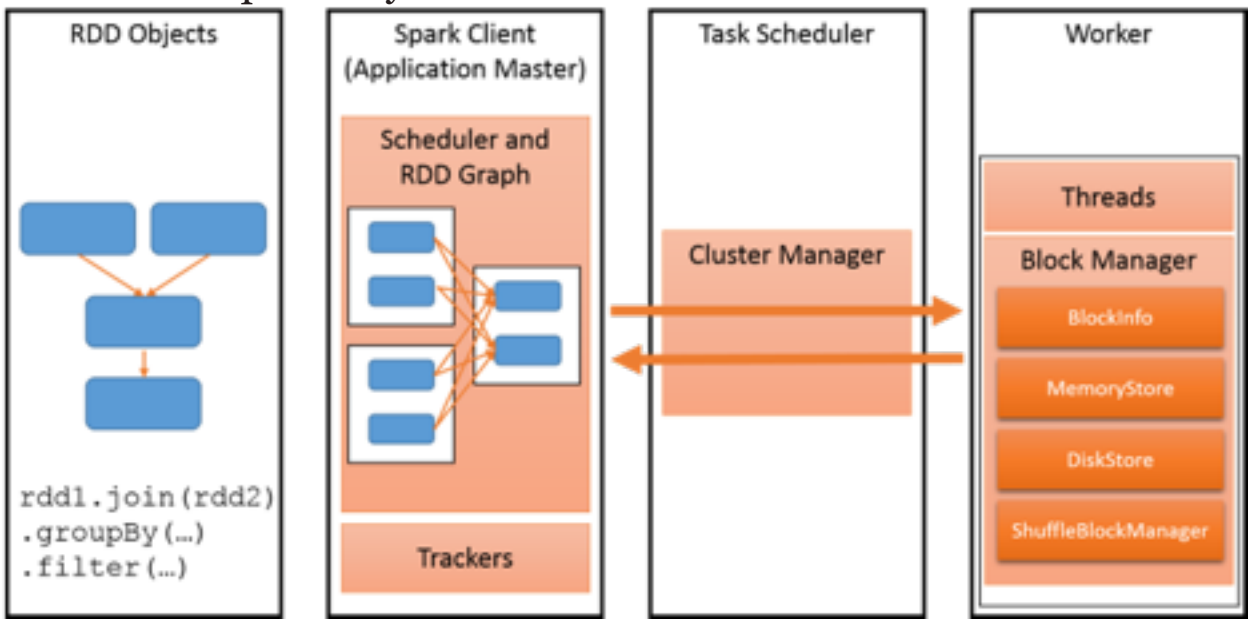

Figure 5.32 shows the most important parts of the Spark framework.

Spark is implemented in about 14,000 lines of Scala, a statically typed high-level programming language for the Java VM.

Spark relies on resilient distributed datasets (RDDs), a distributed memory abstraction to support fault-tolerant, in-memory computations on large datasets. Programmers invoke operations on RDDs by passing closures (functions) to workers, which are copied to and executed at these workers. We will explore each part of this system in detail.

Spark application developers write a driver program to connect to a cluster of workers. The driver defines one or more RDDs and invokes actions on them. The driver also tracks the RDDs' lineage, which records the history of how this RDD is generated as a Directed Acyclic Graph (DAG). The workers are long-lived processes (running for the entire lifetime of an application) that can store RDD partitions in RAM across operations.

The **`SparkContext`** object can connect to several types of cluster managers that handle the scheduling of applications and tasks (Figure 5.33). The cluster manager isolates multiple Spark programs from each other- each application has its own driver and runs on isolated executors coordinated by the cluster manager. Currently, Spark supports applications written in Scala, Java and Python.

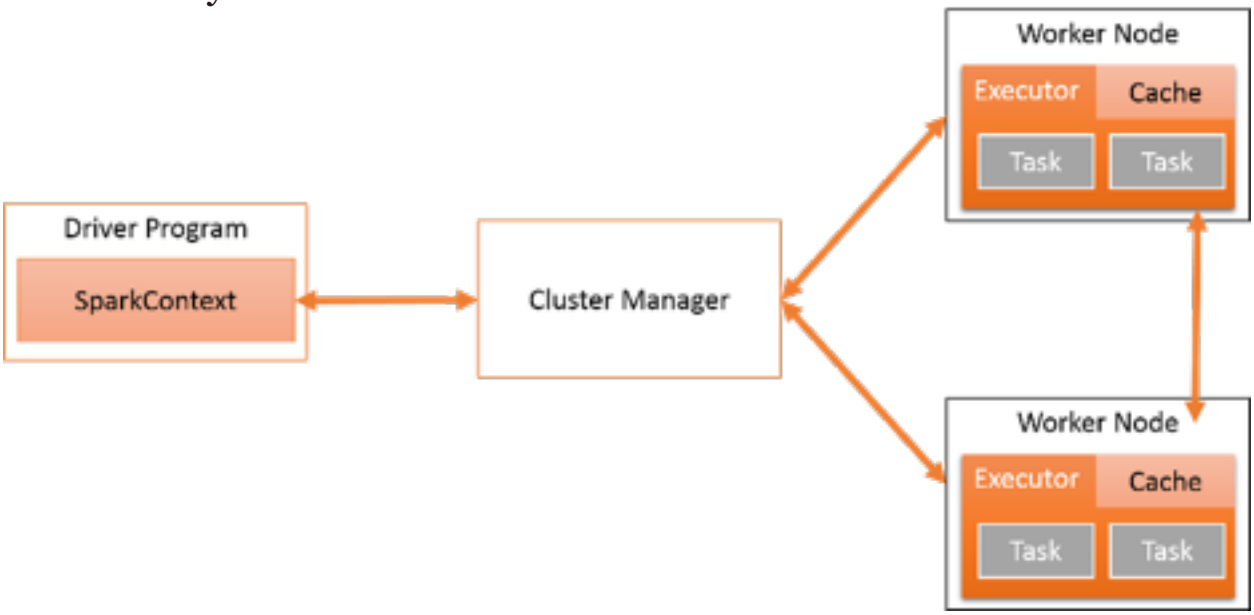

Figure 5.33: Spark Architecture

Each Spark application runs as an independent set of processes on a distributed cluster. The driver is the process that runs the main() function of the application and creates a **`SparkContext`** object. Spark applications are coordinated by the **`SparkContext`** object. The **`SparkContext`** in turn connects to a Cluster Manager, which allocates resources across all applications on the cluster. The **`SparkContext`** object also contains a number of implicit conversions and parameters for use with various Spark features. The system currently supports three cluster managers:

- **Standalone** – a simple cluster manager, included with Spark, makes it easy to setup a cluster. A group of applications submitted in standalone mode will run in first-in-first-out (FIFO) order and each application will try to use all available nodes.
- **Apache Mesos** [3] – a general cluster manager that can also run Hadoop MapReduce and other distributed applications. Mesos allows static and dynamic sharing of CPU cores between applications.
- **Hadoop YARN** [4] – covered earlier. Spark supports two deployment modes in YARN. In yarn-cluster mode, the Spark driver runs inside an application master process, which is managed by YARN. Thus the client's only role is to initiate the application. In yarn-client mode, the driver runs in the client process and the Application Master (AM) is only used to request resources from YARN. When run on YARN, Spark supports Dynamic Resource Allocation, which returns the idle resources allocated to an application back to the global pool.

Once the **`SparkContext`** connects to the **`ClusterManager`**, Spark acquires executors on the worker nodes, which are the actual processes that run computation and store data. After executor is acquired, the Java/Python/Scala code is sent to the executor and run as tasks. Notice that each application has its own executor

processes, which run tasks in multiple threads. The executor exists for the entire application lifecycle.
An advantage of this approach is that applications are isolated from each other. Scheduling decisions are made by individual drivers independent of other applications. Also, executors for different applications are isolated as each one runs in a separate JVM. The disadvantage is that it is more difficult to share data between applications.
The driver is the process where the `main()` method of the program runs. It has two main roles:

1. Converting the user program into tasks: At a high-level, a Spark program implicitly creates a logical directed acyclic graph (DAG) of operations. The driver converts this into a physical execution plan. Several optimizations are performed at this point, converting the execution graph into a set of stages, where each stage encompasses multiple tasks.
2. Scheduling tasks on executors: Once a physical execution plan is created, the driver schedules the running of individual tasks on executors. The driver maintains a global view of all executors.

Executors are worker processes that are created when the Spark application starts and they run until it ends. Executors run the tasks scheduled by the driver and return the results. Each executor consists of a `BlockManager` which provides in-memory storage for caching RDDs.
The core of Spark is the execution and storage of Resilient Distributed Datasets, which is explained next.

A Note on the Spark Shell

Apart from writing programs, Spark also provides an interactive shell. This provides an easy way to explore Spark's APIs and allows a tool for interactive analysis of large datasets. The shell supports both Python and Scala (but not Java).

Spark Resilient Distributed Datasets

Spark relies on a special abstraction called resilient distributed datasets (RDDs) [1] . RDDs are in-memory read-only objects partitioned across the cluster. They let users control persistence and partitioning settings to optimize data placement and manipulate this data using a rich set of operators. An RDD is partitioned across machines either based on a range (partitioning of consecutive records) or the hash of a key in each record. Each partitioning method is optimal for a particular use case (hash partitioning speeds up joins by providing locality to records from different datasets that share keys; range partitions speed up access to a small filtered subset of the data).
Despite not needing to exist on physical storage, RDDs can be fault-tolerant. However, they do not need to be replicated; rather they have a notion of lineage by which they "remember" the set of operations that were executed to construct them, allowing them to be rebuilt if they lose data. A handle to an RDD contains enough information to recompute it from a version of the data stored on-disk.
All work in Spark is expressed either as creating new RDDs, transforming existing RDDs or running operations on RDDs. One important thing to realize about RDDs is that they are **lazily computed** and **ephemeral**. Lazy computation is an optimization whereby many transformations are pipelined, and the RDD is computed only when it is first used with an "action". Ephemeral means that RDDs may be materialized (computed and loaded in memory) when used in a parallel application but that they are subsequently discarded from memory.
Aspect
RDDs
Distributed Shared Memory
Reads

Coarse or Fine-Grained
Fine-Grained
Writes
Coarse-Grained
Fine-Grained
Consistency
Trivial (Immutable)
Up to Application / Runtime
Fault Recovery
Fine-Grained and low-overhead using lineage
Requires checkpoints and program rollback
Straggler Mitigation
Possible using backup tasks
Difficult
Work Placement
Automatic based on data locality
Up to Application (runtimes aim for transparency)
Behavior if not enough RAM
Similar to existing data flow systems
Poor performance (Swapping)

The RDD abstraction is a type of distributed shared collection system, similar to traditional Distributed Shared Memory (DSM) systems [2] . It may be interesting to compare the two abstractions.

Unlike DSMs, which allow read/writes to individual memory locations, Spark only allows coarse-grained transformations of RDDs. Also, RDDs enable a low overhead recovery mechanism using lineage, unlike the coordinated checkpointing needed by DSM systems [3] . Similar to MapReduce, stragglers can be mitigated using speculative execution of slow tasks.

Next, let's discuss the lifecycle of an RDD.

Creating RDDs

By default, in the Spark programming model, an RDD is represented as a Scala object (though it could also be a Python or Java object). It can be constructed in several ways:

1. From a file on a distributed file system like HDFS
2. By parallelizing a collection/array and distributing it to many nodes
3. By transforming an existing RDD (we will discuss transform operations below)
4. By changing the persistence level of an existing RDD using one of two actions:
    - **cache**: this is a hint to the framework to keep the RDD in-memory after the first computation to ensure reuse
    - **save**: writes the data to a distributed filesystem like HDFS

Examples of creating new RDDs (these commands can be tried out in the Spark Scala shell):

1. We load a text file (**`server.logs`**) into a string RDD using the Python **`textFile()`** method.

```
1
>>> log_lines_RDD = sc.textFile("server.logs")

```

5. We parallelize an existing RDD:

```
1
>>> greeting_lines_RDD = sc.parallelize(["hello",
"world"])

```

Operations on RDDs

Once created, an RDD supports two types of operations (see Figure 5.34):

1. **Transformations**: Operations that create new RDDs from existing ones
2. **Actions**: Computations on an RDD that return a single object to the driver

As mentioned earlier, Spark transformations are lazy by default, i.e. they are not computed immediately, rather they are batched and executed only when an action is executed. The execution of an action causes all RDDs in the lineage to be materialized. However, once the computation is completed, an RDD will persist only if explicitly required to by the program.

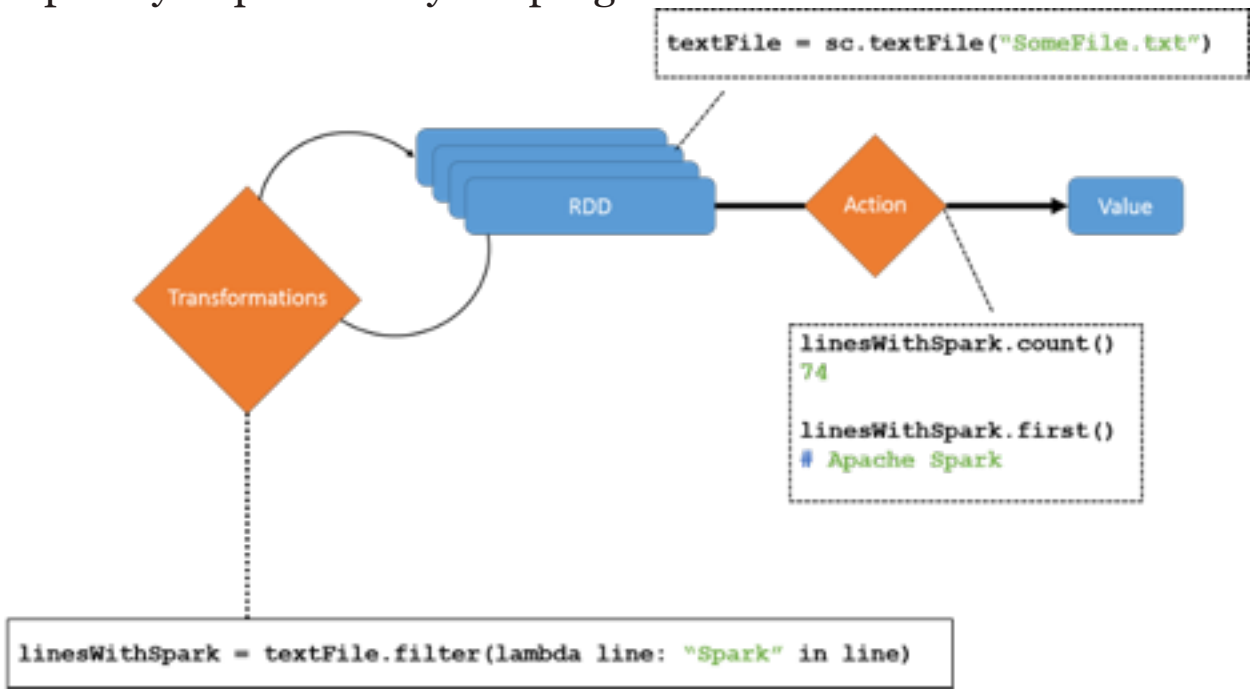

Figure 5.34 : Operations on RDDs

Here are some examples of transformations and actions:

Transformation

Meaning

Action
Meaning
`map(func)`
Return a new RDD formed by passing each element of the source through function func.
`reduce(func)`
Aggregate the elements of the dataset using function func (which takes two arguments and returns one). The function should be associative so that it can be computed correctly in parallel.
`filter(func)`
Return a new RDD by selecting those elements of the source on which func returns true.
`collect()`
Return all the elements of the RDD as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
`join(otherDataset, [numTasks])`
When called on two RDDs of type `(K, V)` and `(K, W)` respectively, returns a RDD of `(K, (V, W))` pairs with all pairs of elements for each key.
`count()`
Return the number of elements in the RDD.
We will explore these in more detail in the next page where we look at some sample programs. For now, let us look at a few simple examples to understand the basic operations allowed by an RDD.

Transformations

```
1
>>> log_lines_RDD = sc.textFile("server.logs")

2
>>> xss_RDD =  log_lines_RDD.filter(lambda x:
"%3C%73%63%72%69%70%74%3E" in x)
```

```
7
8  3
9  >>> sqli_RDD =  log_lines_RDD.filter(lambda x:
   "bobby_tables" in x)
10
11 4
12 >>> owasp_attacks_RDD = xss_RDD.union(sqli_RDD)
13
```

The Python code above filters **`log_lines_RDD`** to find attacks against a web server by searching for lines matching a particular signature string.
`filter()` does not modify the original RDD, rather we now have 2 new RDDs. The **`log_lines_RDD`** can still be used in the future. **`union()`** is a transformation that acts on 2 RDDs to form a combined owasp_attacks_RDD .

Actions

```
14
15 1
16 >>> log_lines_RDD = sc.textFile("server.logs")
17
18 2
19 >>> xss_RDD =  log_lines_RDD.filter(lambda x:
   "%3C%73%63%72%69%70%74%3E" in x)
20
21 3
22 >>> sqli_RDD =  log_lines_RDD.filter(lambda x:
   "bobby_tables" in x)
23
24 4
25 >>> owasp_attacks_RDD = xss_RDD.union(sqli_RDD)
```

Here, the Python code counts the number of attacks in our combined **`owasp_attacks_RDD`**. An action forces materialization. In many **`examplesRDD.count()`** is often used to force materialization during local testing of Spark code.
Clearly, RDDs are best suited for batch operations where the same operations are applied to all elements of the dataset. Applications requiring asynchronous, fine-grained updates would need more specialized systems. Since they are immutable, the overhead of computing new RDDs for each additional input item is very high. Hence, even when dealing with real-time data inputs, Spark often batches the changes over short periods of time.

27 **Persisting RDDs**RDDs may be persisted in four ways [4] :

a **In-memory as deserialized objects**: A deserialized object expresses a data structure as a set of bytes. Storing the raw deserialized RDD objects in-memory has the highest performance, since the framework can access the elements natively in code. However, there is a some overhead, since apart from the data, the metadata of the object is also stored. For e.g., Java objects are fast to access, but consume 2-5x more space than the raw data that they encapsulate.

b **In-memory as serialized data**:
By serializing the RDD, the associated data is stored in a well-defined format. This method is slower than storing deserialized objects, but is more memory-

efficient than storing object graphs.

c **On-disk storage**: This helps store really large RDDs that do not fit in-memory but shouldn't be continuously recomputed.

d **Off-heap storage**: Off-heap storage is provided by a special memory-centric storage system called Tachyon [4] , which enables cluster-wide reliable data sharing at memory-speed.

In all four cases, the RDD is stored in partitions across the workers. Each partition is an atomic piece of the dataset. When running out of memory because of a high rate of incoming data, in-memory RDD partitions are generally dealt with using an LRU eviction policy. However, if the least-recently used partition belongs to the same RDD as the arriving partition, it is not evicted to prevent thrashing.

RDDs are recomputed by default for every action executed on them. In order to reuse an RDD in multiple actions, the RDD.`persist()` method can be called. RDDs can be persisted in several places as outlined in the following table:

Level
Space used
CPU time
In memory
On disk
Comments

`MEMORY_ONLY`

High
Low
Y
N

`MEMORY_ONLY_SER`

Low
High
Y
N

`MEMORY_AND_DISK`

High
Medium
Some
Some
Spills to disk if there is too much data to fit in memory.

`MEMORY_AND_DISK_SER`

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

`DISK_ONLY`

Low
High
N
Y

Spark also allows data to be persisted into a cluster-wide in-memory cache. When data is accessed frequently, for e.g. a small "hot" dataset or when running an iterative algorithm, this technique can be extremely useful to improve performance. RDDs also have an **`unpersist()`** method to remove it from the persistent store.

Of course, to achieve all these properties using RDDs, we also must choose a suitable representation to store them. Any representation must be able to track lineage across a wide range

of transformations, which users can combine in arbitrary ways. Spark uses a simple graph-based representation for RDDs. Each RDD is accessed through a common interface which exposes five features: a set of partitions (atomic pieces of the dataset), a set of dependencies (on parent RDDs), a function for computing the dataset based on its parent(s), metadata about the partitioner, and a list of preferred nodes where each partition can be accessed faster due to locality.

Operation
Meaning
`partitions()`
Return a list of Partition objects
`preferredLocations(p)`
List nodes where partition p can be accessed faster due to data locality
`dependencies()`
Return a list of dependencies
`iterator(p, parentIters)`
Compute the elements of partition p given iterators for its parent partitions
`partitioner()`
Return metadata specifying whether the RDD is hash/range partitioned

Consider that our input dataset consists of files on HDFS. By default, **`partitions()`** returns a list for each HDFS block encompassed by the file. Each Partition object in this list is represented by the block's offset. Since this is an HDFS system, **`preferredLocations(p)`** returns the list of nodes storing a local copy of the block. The **`iterator(p, parentIters)`** simply reads the block.

At this stage, we have looked at RDDs and some of the basic operations on them. In the next pages, we will cover more detailed examples to understand how RDDs help Spark achieve very high

performance for many types of applications and also how they enable fault tolerance and recovery.

Fault Tolerance

On large, distributed clusters running long jobs on commodity hardware, it is very important to have a fault-tolerant framework in case of any errors. Unlike Hadoop which emphasises quick fault-recovery by replicating data stored on HDFS and straggler jobs, Spark relies on the abstract concept of a lineage to recover from errors. A lineage is a directed acyclic graph that defines the operations required to create an RDD [1] .
Let us return to the program that we saw on the previous page (describing RDD transformation):

```
1
>>> log_lines_RDD = sc.textFile("server.logs")
2
>>> xss_RDD =  log_lines_RDD.filter(lambda x: "%3C
%73%63%72%69%70%74%3E" in x)
3
>>> sqli_RDD =  log_lines_RDD.filter(lambda x:
"bobby_tables" in x)
4
>>> owasp_attacks_RDD = xss_RDD.union(sqli_RDD)
```

As RDDs are derived from each other using transformations, Spark keeps track of the dependencies using a lineage graph. This allows the RDDs to be computed lazily (as defined earlier) and provides fault recovery information to the framework. The lineage for the code above is shown here (Figure 5.35).

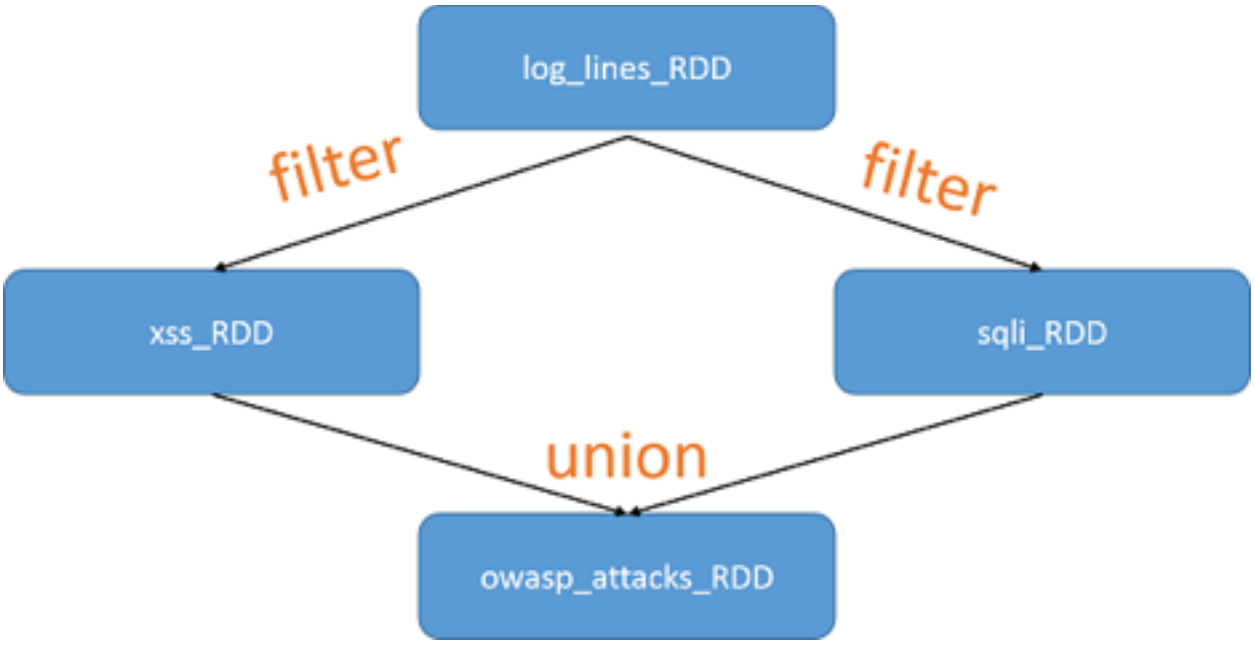

Figure 5.35: RDD lineage graph

Since RDDs are immutable, these graphs are extremely easy to define. Please note that at no point in the graph above has the RDD been materialized and computation been actually carried out. That occurs at the next line of code, which is the first action in the program:

```
1
>>> print "Number of attacks:" +
owasp_lines_RDD.count()
```

Since none of the RDDs were persisted, only the **`owasp_lines_RDD`** will be stored in memory at this point. By achieving fault-tolerance using in-memory data structures, Spark focuses on reducing the overhead of fault tolerance due to deplicated writes on HDFS. By storing RDDs in memory and recomputing lost partitions, Spark can avoid the high-cost of Hadoop fault-tolerance technique including replication and the consequent disk IO. On the other hand, Hadoop has much faster fault recovery by simply switching to one of the other replicas.

Dependencies

One of the interesting challenges faced by the creators of Spark was to have a suitable way to represent dependencies between RDDs. They classified dependencies into two types:

1. narrow dependencies, where each partition of the parent RDD is used by at most one partition of the child RDD
2. wide dependencies, where multiple partitions of the child RDD use each partition of the parent RDD

Consider the **`map(func)`** transformation, which returns a new distributed dataset formed by passing each element of the source through the function **`func`**. Here the child has a narrow dependency on the parent, since each parent RDD partition returns a single child partition. Basically, when **`map(func)`** is called on an RDD, the transformation returns a **`MappedRDD`** object. This has the same partitions and **`preferredLocations`** as the original (parent) RDD, but additionally applies **`func`** to the parent's record in the iterator method.

On the other hand, consider a **`join()`** transformation, which when called on RDDs of type **`(K, V)`** and **`(K, W)`**, returns a RDD of **`(K, (V, W))`** pairs with all pairs of elements for each key. If the parent RDDs are partitioned with the same range/hash partitioner), then this can be represented as two narrow dependencies. If neither RDD has a defined partition, then this is a wide dependency. Finally, if one parent has a partitioner and one does not, then it is categorized as a mixed dependency. In all three cases, the output RDD has a partitioner, this may be inherited from the parents, or a default (hash) partitioner.

Both of the dependencies above are shown in Figure 5.36:

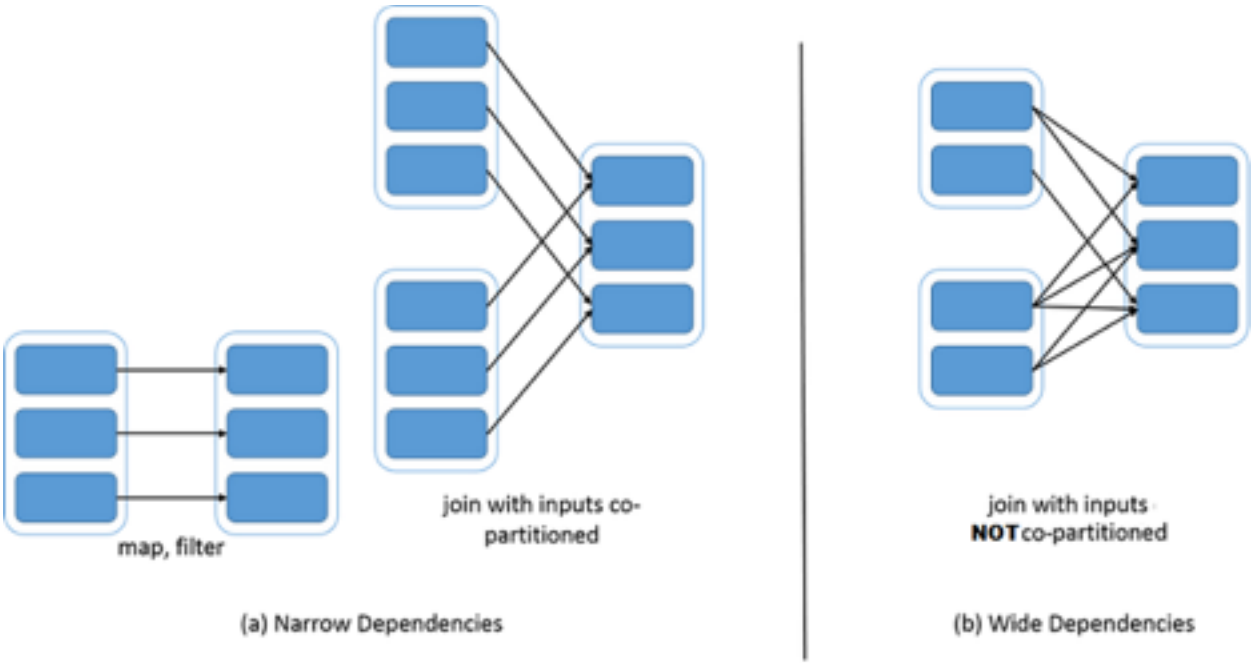

Figure 5.36: Narrow and wide dependencies in Spark

Checkpointing

In case a lineage chain grows extremely long or has very wide dependencies, it may be infeasible to re-run it on a failure. A solution is to checkpoint such RDDs. Unlike Distributed Shared Memory systems, however, the entire application state does not need to be checkpointed. Additionally, the immutable nature of RDDs makes it easy to do this as a background process without significantly impacting the performance of the running application or requiring a distributed snapshot scheme to deal with consistency.

Checkpointing is done for tracking metadata (job configuration, job state) and saving some of the generated RDDs to reliable storage (HDFS).

Using the techniques of lineage and checkpointing, Spark achieves fault-tolerance and recovery.

Now that we have seen the architecture and the computation flow of Spark, it makes sense to look at a simple Spark program. Before doing that, still need to understand some of the common primitive RDD operations.
We start with some basic single-RDD transformations **(RDD1 = {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}`

Transformations may also have more than one RDD as the input. Here we look at transformations having two RDDs as the input **(RDD1 = {1,2,3}; RDD2 = {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)}`

Finally, we look at some actions, again on a single RDD = `{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)}`
`take(num)`	Return numelements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top numelements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return numelements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
`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)`

Now that we have studied some primitives, let us see how all these operations fit together to implement a simple program. We explore the implementation of an iterative PageRank algorithm on Spark.

The PageRank algorithm iteratively updates a rank for each URL by adding contributions from URLs that link to it. PageRank assumes that a web surfer starting on a random page has an 85%

chance to select a random link from their current page, and a 15% chance to jump to a random page anywhere on the Internet.
On each iteration, each URL contributes r/n to its neighbors, where r is its own rank, and n is the number of neighbors of that node. It then updates its rank to
is the sum of all received contributions and N is the total number of documents. **a** indicates the probability that a random surfer starting from a web page will stop clicking through. This is known as the damping factor (as mentioned above, studies have found it to have a probability of about 0.85).
Consider the following PageRank Scala program:
In this Spark implementation of PageRank, our input dataset consists of a text file of the format `(URL, rank)`. For each iteration, a join operation on `links` and `ranks` is used to aggregate the contribution for each URL. The `contribs` RDD represents the contribution sent by each URL. These are summed up over each key (using a reduce) and this value is then updated using the PageRank formula

. Once the `ranks` RDD is updated, the process repeats again for the number of iterations specified.

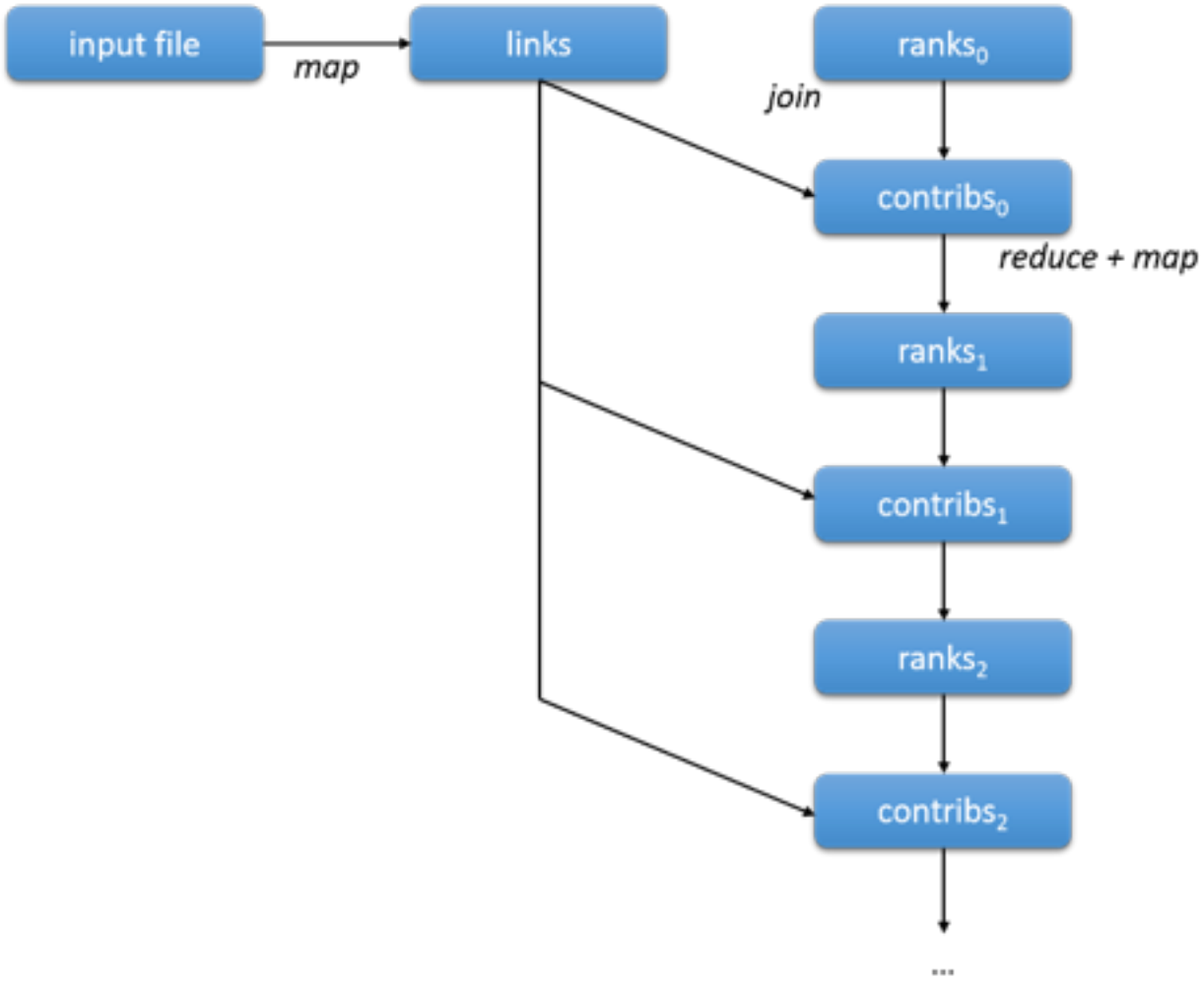

Figure 5.37: Lineage graph for the Spark PageRank example

As we mentioned on the previous page, the join operation can be optimized to reduce communication by partitioning the links and the ranks in the same way (for e.g. using a hash-partitioner to partition URLs across nodes). If each link partition and its corresponding rank partition are on the same node, we can eliminate cross-node communication entirely for the join.

Notice that in Figure 5.37 above, the lineage of the ranks RDD in each iteration keeps increasing. Hence, it may be important to adopta strategy to persist some of the versions of ranks for better fault recovery.

Decisions about partitioning, persistence and choosing the optimal set of operations to define a computation are some of the features that make developing optimal Spark programs

challenging. However, if implemented correctly, Spark jobs can have a huge speedup over traditional MapReduce jobs.

Spark has built-in libraries or modules that include Spark SQL for SQL and structured data processing, Spark Streaming, MLlib for machine learning and GraphX for graph processing. This basically presents an unified platform to perform ETL, MapReduce, and complex analytics.

Spark SQL

Apache Spark provides a module for structured data processing - Spark SQL. With Spark SQL, users have the ability to run SQL-style queries against Spark RDDs. There are two main advantages to using Spark SQL: 1) A wide user base of SQL programmers and developers can use Spark to run analytics jobs, and 2) This allows application developers to use Spark RDDs as a database backend, similar to MySQL or Hive.

Spark SQL provides a programming abstraction for its users in the form of **`DataFrames`** -- which is a distributed collection of data organized into columns. **`DataFrames`** also allows the integration of SQL commands into applications that use the MLlib library. This is explained a bit more in the MLlib section. The API for **`DataFrames`** is available in Java, Scala and Python. **`DataFrames`** may be constructed from tables in Hive, external databases, structured data files or RDDs.

Applications may run SQL queries programmatically (similar to MySQL) via the sql function in **`sqlContext`**. The result is returned as a **`DataFrame`**. Similarly, the schema for a table can also be specified programmatically. There are a variety of options for specifying data sources through the **`DataFrame`** interface and Spark SQL offers support for a variety of data sources (for

example, json, parquet). Like other operations on a table, data may also be loaded programmatically.

Spark Streaming

An extension of the core Spark API, Spark Streaming enables scalable, high-throughput, fault-tolerant stream processing of live data streams.

Figure 5.38: Spark Streaming

Spark streaming allows for a streaming data source to be connected to a Spark cluster (Figure 5.38). The spark streaming system will divide the streaming input into batches which can then be fed into the Spark engine to run the required analytics operations. One example is that a spark cluster can be connected to a twitter stream to filter out tweets that are relevant to a particular company or product, and a sentiment analysis can be run on the filtered tweets to provide real-time reports on product or brand sentiment.

The high-level abstraction provided by Spark streaming is the discretized stream or **`Dstream`**, which can be created from an input stream. **`Dstreams`** are represented internally in Spark as a sequence of RDDs. The APIs in Spark streaming allow for the creation of a stream processing pipeline in Java, Python or Scala. To build a streaming pipeline in Spark streaming, first a Dstream should be constructed from an input data source. The source can be as simple as a network socket or file stream, or can be a more complex system such as Kakfa, Flume, Kinesis or a Twitter feed. Once a **`Dstream`** is constructed, it can be sent to any number of streaming functions to be transformed. Some of the functions

include **`map()`**, **`reduce()`**, **`join()`**, **`count()`**, etc. Refer to the API for more details.

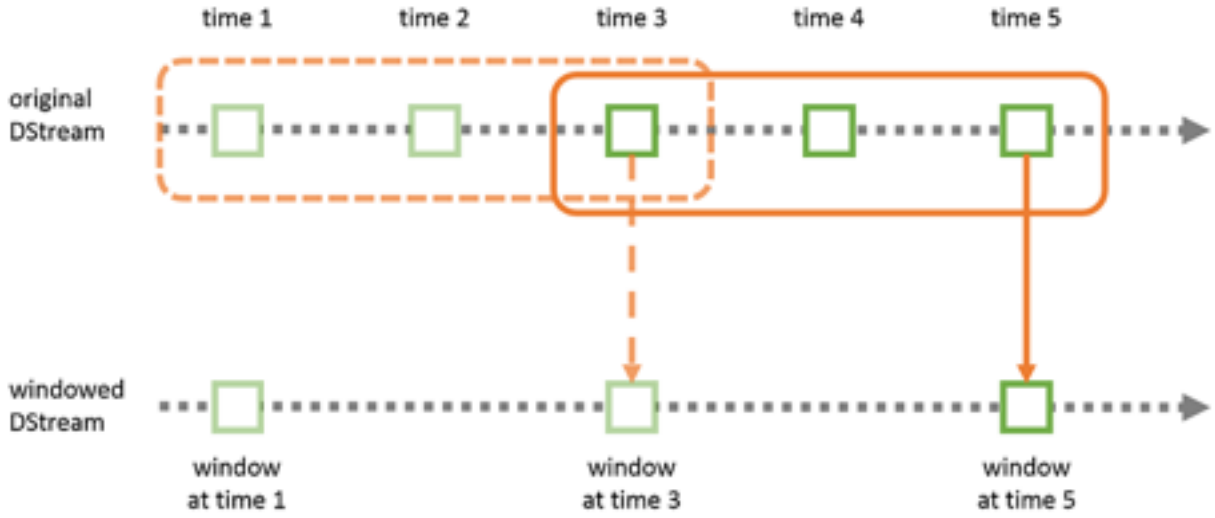

Figure 5.39 : Sliding Window Computation in Spark Streaming

In addition to simple transformations, Spark streaming provides the capability to perform windowed computations, i.e., transformations applied over a sliding window of data (Figure 5.39). In this case, the user can specify both the **window size** (the number of source RDDs to include in the transformation window) and the **sliding interval** (the number of source RDDs to slide across at the end of each transformation).
Once the source RDDs in a **`Dstream`** have been computed, Spark streaming provides various output options, including write to binary or text or Hadoop-compatible files, or save the RDDs for future processing.

MLlib

Spark provides a scalable machine learning library called MLlib. This fits into the APIs for Apache Spark and is available to use in Java, Scala and Python. Using MLlib with Python requires the NumPy module. The Hadoop file system (HDFS) and other Hadoop based data sources (such as HBase) may be used with this, hence making it easier for the modules in the libraries of

MLlib to be easily plugged into Hadoop workflows. MLlib consists of common machine learning algorithms including those for classification, regression, clustering, dimensionality reduction, transformation and extraction of features and collaborative filtering. There are also libraries for basic statistics. Additionally, there are a few optimization libraries such as stochastic gradient descent and BFGS.

The libraries in MLlib leverage iterative computations and hence are high-performant since Spark is adept at iterative computations. This is superior to MapReduce algorithms that sometimes use one-pass approximations. Furthermore, the newer versions of MLlib in Apache Spark include a new package - **`spark.ml`** which lets users combine multiple algorithm into a single learning pipeline that may be specified as a sequence of stages using a set of high level APIs.

The Spark ML API has different components that are important to understand. One of the most important considerations of any machine learning library is how to handle data, especially a variety of data types. Spark ML uses **`DataFrame`** from Spark SQL to support a variety of datatypes under a unified Dataset concept.

Another feature that simplifies handling data that is included in the Spark ML API is the concept of **`Transformers`**, by implementing a method - **`transform()`**, which helps in data transformation (for example, converting a feature vector to another via regularization). In other words, a **`Transformer`** is essentially a model that converts one **`DataFrame`** to another.

Supplementing this, of course, is a machine learning algorithm that fits and trains data. This is the concept of an **`Estimator`** which implements a method - **`fit()`**.

For example, **`NaiveBayes`** implements a multinomial naive Bayes classifier. The output is a **`NaiveBayesModel`** which can be used for prediction on test data. **`NaiveBayes`**, in this case is the

`Estimator` and the model learnt, **`NaiveBayesModel`**, may be used as a **`Transformer`**.

The workflow of an entire machine learning process - from taking in data, converting it to the required format, fitting it to a model - is represented in the form of a Pipeline in Spark ML, consisting of a series of **`PipelineStages`**. One can specify the sequence of operations in Spark ML for their ML job via a Pipeline.

GraphX

GraphX is used for graphs and graph parallel computation. GraphX is an extension of RDDs and although both have similar basic operations, GraphX extends Spark RDD by adding a new graph abstraction. The idea behind GraphX is to support some of the operations and techniques used by graph-specific frameworks such as Pregel and GraphLab (covered in the next Module). As a result, GraphX allows Spark to efficiently run graph-parallel computation such as PageRank and connected components at performance comparable to these graph-specific frameworks. GraphX includes abstractions to efficiently represent attributed, multi-relational graphs using Spark RDDs. One example is the property graph. In this kind of graph, the edges are labeled and both vertices and edges can have any number of key-value pairs associated with them (Figure 5.40). This type of graph is also directed. This type of graph basically helps support multiple parallel edges which, for example, can be used to signify multiple relationships between 2 vertices (for example, relationship between co-workers can also be that of friends and the vertices of the graph would be 2 people). In GraphX, a vertex is represented as a unique 64 bit long identifier - **`vertexID`**. There are no constraints on ordering of these identifiers. Edges of the graph also have corresponding source and vertex identifiers. The Graph class has members to access the vertices and edges of the graph.

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

The figure shows a property graph consisting of various collaborators of project Archon. As shown in the graph, the vertex property is the username of the individual and the occupation. The edges of the graph describe the relationship between the individuals. The graph shown below would have the signature as the following:

```
val userGraph: Graph[(String, String), String]
```

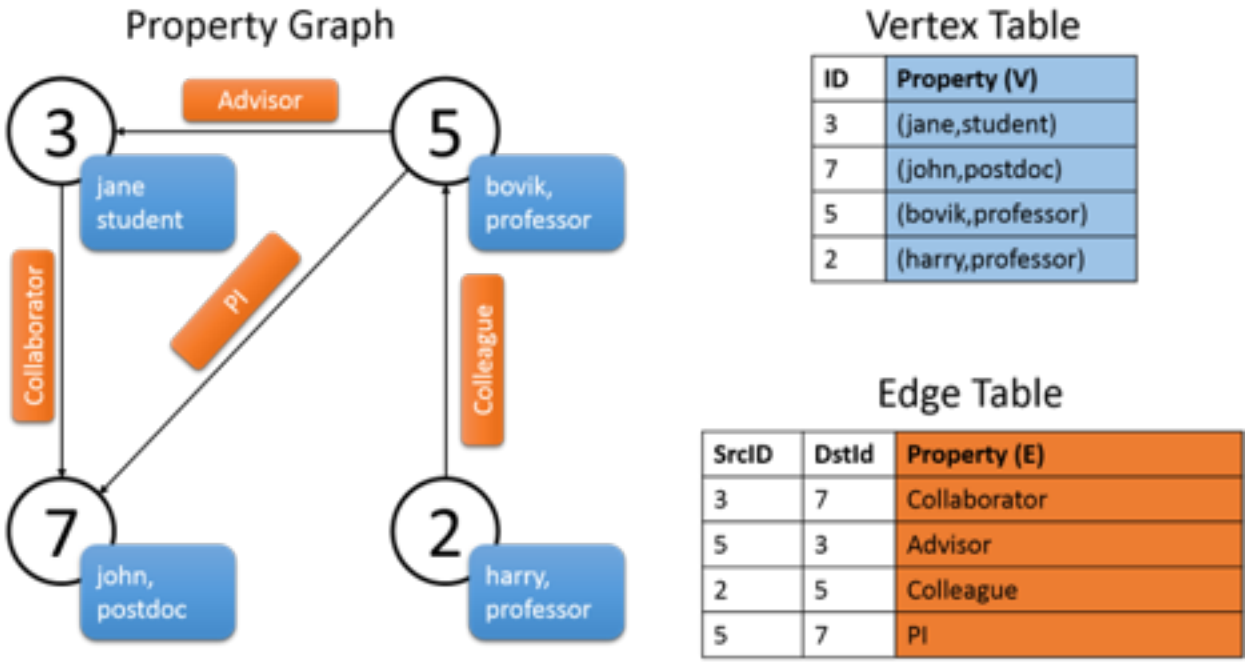

Figure 5.40 : A property graph in GraphX

A property graph in GraphX is parameterized over vertex (VD) and edge (ED) types. GraphX provides several ways of

constructing this graph, given a collection of vertices and edges in an RDD or a disk. Graphs can be generated from raw files, RDDs or using synthetic generators.
GraphX supports fundamental graph operations (for example, **`subgraph`**, **`joinVertices`**, and **`aggregrateMessages`**) as well as an optimized version of the Pregel API. Additionally, to simplify graph analytics for users, Spark GraphX also includes a variety of algorithms and graph builders. The algorithms can be accessed directly as methods and are included in a package called **`org.apache.spark.graphx.lib`**.
Spark GraphX also allows users to create functions and transform graphs based on new properties. GraphX is designed as an extensible framework. There are 2 classes that help do this. The **`Graph`** class consists of the core operators with certain optimizations needed to generate the graph. **`GraphOps`** is an extension of the **`Graph`** class that has more convenient operators using the core operators in **`Graph`**. Apache Spark separated the operators this way to provide for convenient future extensions. Future graph representations would have to provide the core operators listed in Graph and have the option to reuse the implementation in GraphOps. A list of operations in each of these is provided in the API documentation.

Distributed Analytics Engines for the Cloud: Spark

- MapReduce is ill-suited towards certain types of applications which are iterative in nature. This is due to the heavy I/O cost involved in reading inputs from DFS and writing them back to DFS for each iteration.

- Spark is an optimized, in-memory framework suited towards iterative, interactive and streaming applications.
- Spark relies on Resilient Distributed Datasets (RDD), a distributed memory abstraction to support fault-tolerant, in-memory computations on large clusters.
- Spark can either be run in standalone mode, or in a cluster using either Mesos or YARN resource managers
- RDDs are in-memory read-only (immutable) objects partitioned across the cluster.
- RDDs are fault tolerant by using a lineage tracking technique that keeps track of the sequence operations performed to transform on-disk data to its current form in memory.
- Dependencies in RDDs are classified as either narrow or wide dependencies.
- The Spark ecosystem includes Spark SQL, Spark Streaming, MLlib and GraphX.