What We Will Cover

- Map Reduce
  - Grandfather of most current approaches
- Apache Spark
  - Current leader in big data (OLAP-style) frameworks
  - Supports many query/analysis models, including a light version of SQL
  - Used for Assignment 9
- MongoDB
  - Perhaps the most popular NoSQL system, uses a ”document” (JSON) data model
  - Focus primarily on OLTP
  - Doesn’t really support joins (some limited ability today) – have to do that in the app
- How to “Parallelize” Operations
  - Useful to understand how Spark and other systems actually work
  - Often times you have to build these in the application layer
  - The original MapReduce framework
    - Led to development of much work on large-scale data analysis (OLAP-style)
    - Basically a way to execute a group-by at scale on non-relational data
- Hadoop Distributed File System (briefly)
  - A key infrastructure piece, with no real alternative
  - Basic file system interface, with replication and redundancy built in for failures
- Quick overview of other NoSQL data models

MapReduce Framework

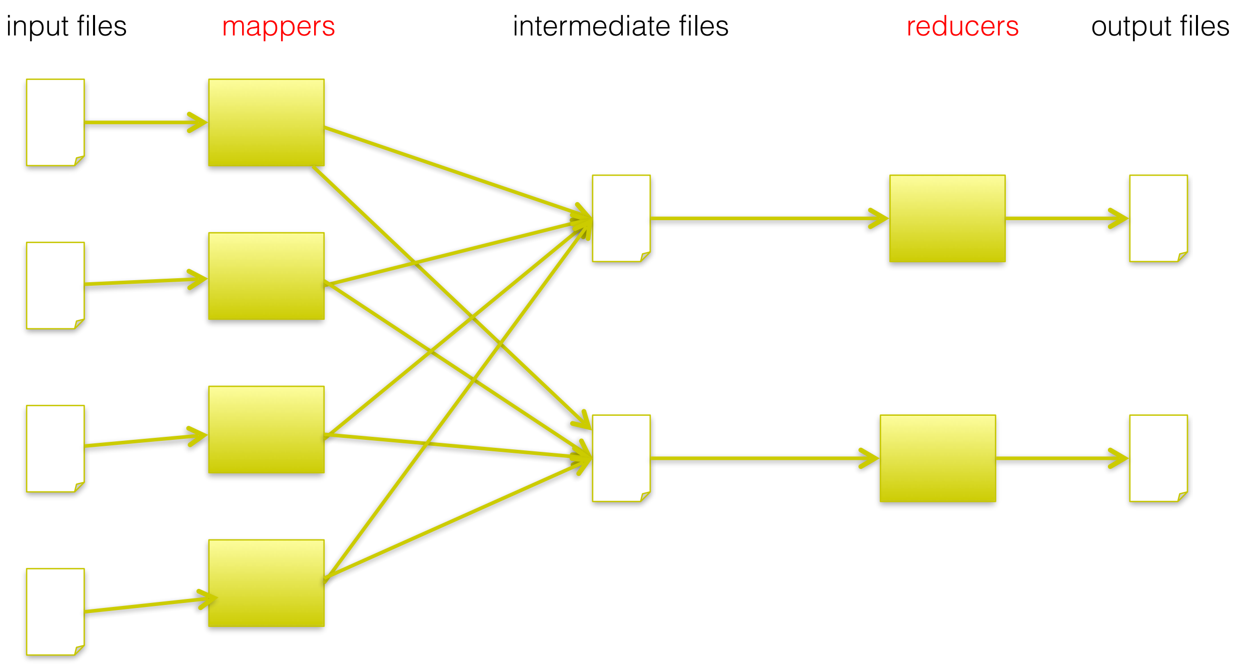
- Provides a fairly restricted, but still powerful abstraction for programming

- Programmers write a pipeline of functions, called *map* or *reduce*
  - map programs
    - inputs: a list of “records” (record defined arbitrarily – could be images, genomes etc…)
    - output: for each record, produce a set of “(key, value)” pairs

  - reduce programs
    - input: a list of “(key, {values})” grouped together from the mapper
    - output: whatever

  - Both can do arbitrary computations on the input data as long as the basic structure is followed

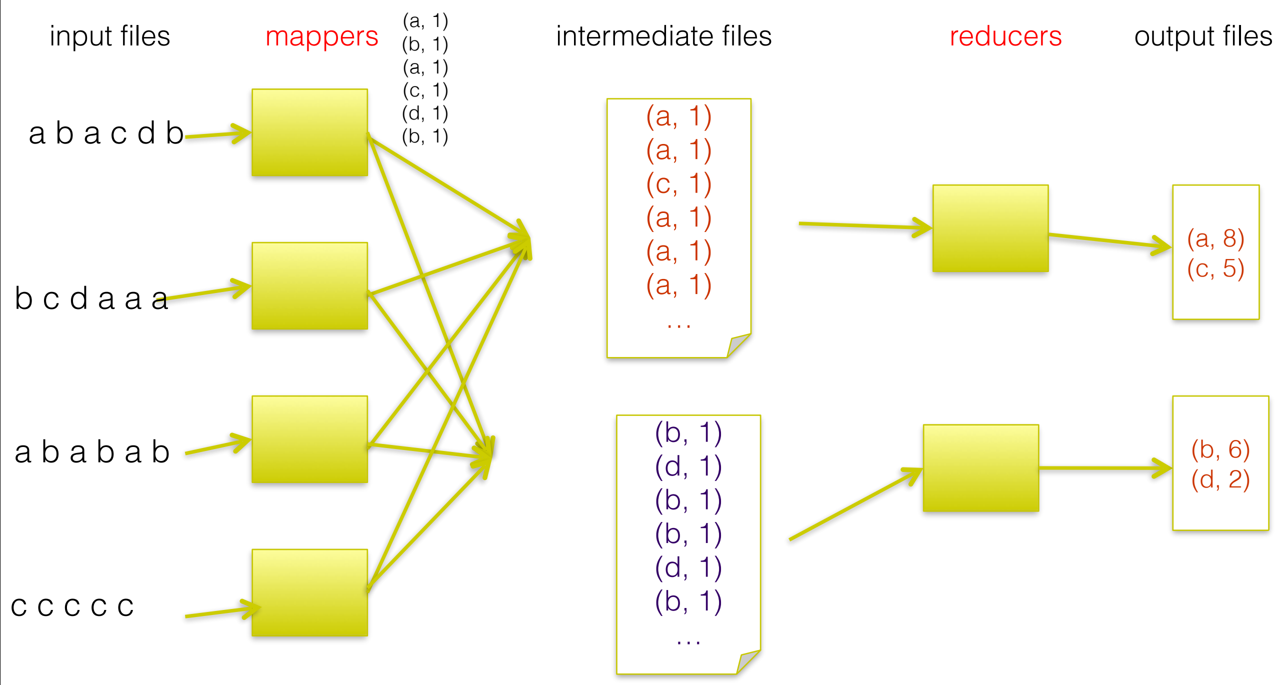
MapReduce Framework



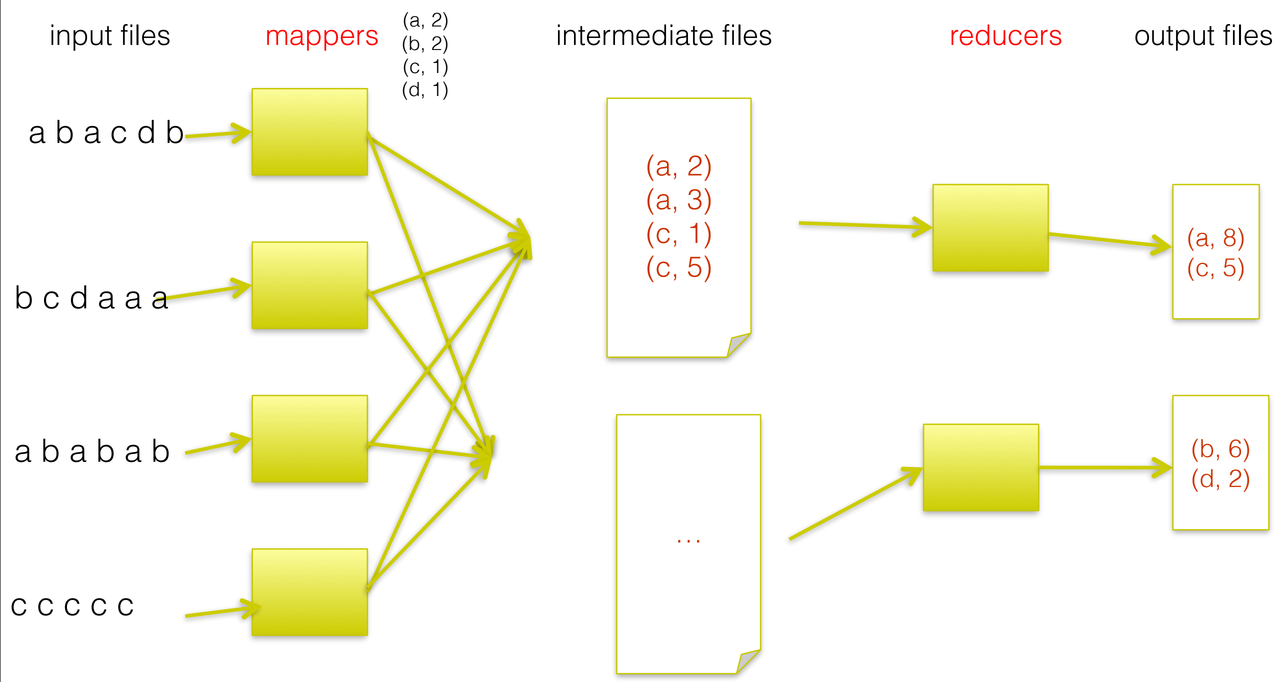
Word Count Example

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

MapReduce Framework: Word Count



More Efficient Word Count



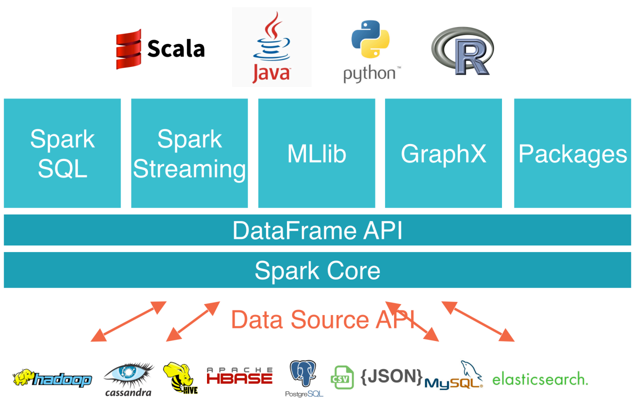
"mapper-side" combiner

Apache Spark

- Map-Reduce on steroids
- Book Chapters
  - 10.4 (7TH EDITION) covers this topic, but Spark programming guide is a better resource
  - Assignment will refer to the programming guide
- Key topics:
  - A Resilient Distributed Dataset (RDD)
  - Operations on RDDs

Spark

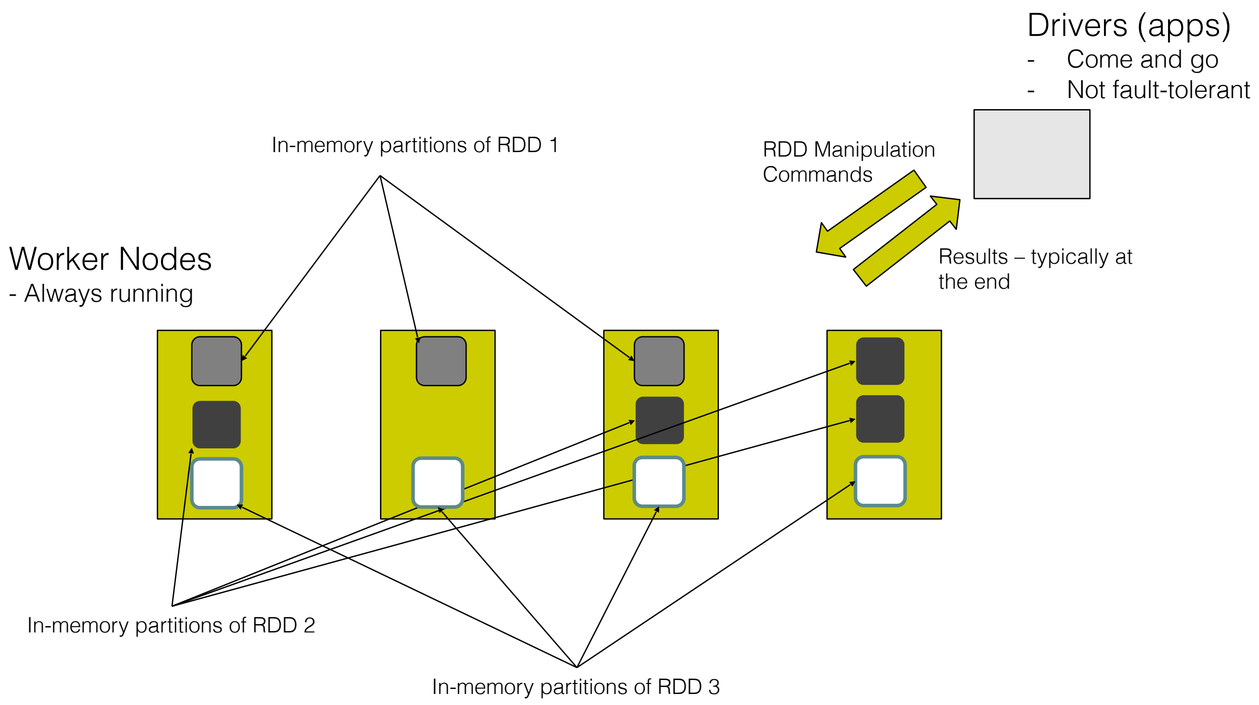
- Open-source, distributed cluster computing framework
- Much better performance than Hadoop MapReduce through in-memory caching and pipelining
- Originally provided a low-level RDD-centric API, but today, most of the use is through the “Dataframes” (i.e., relations) API
  - Dataframes support relational operations like Joins, Aggregates, etc.



databricks

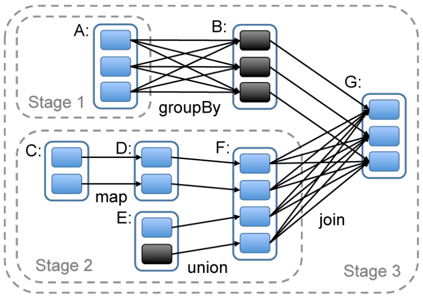
Resilient Distributed Dataset (RDD)

- RDD = Collection of records stored across multiple machines in-memory


Drivers (apps)
- Come and go
- Not fault-tolerant
In-memory partitions of RDD 1
RDD Manipulation Commands
Results – typically at the end
Worker Nodes
- Always running
In-memory partitions of RDD 2
In-memory partitions of RDD 3

Spark

- Why "Resilient"?
  - Can survive the failure of a worker node
  - Spark maintains a "lineage graph" of how each RDD partition was created
  - If a worker node fails, the partitions are recreated from its inputs
  - Only a small set of well-defined operations are permitted on the RDDs
    - But the operations usually take in arbitrary "map" and "reduce" functions



- Fault tolerance for the "driver" is trickier
  - Drivers have arbitrary logic (cf., the programs you are writing)
  - In some cases (e.g., Spark Streaming), you can do fault tolerance
  - But in general, driver failure requires a restart

Example Spark Program

Initialize RDD by reading the textFile and partitioning. If textFile stored on HDFS, it is already partitioned – just read each partition as a separate RDD partition

Split each line into words, creating an RDD of words

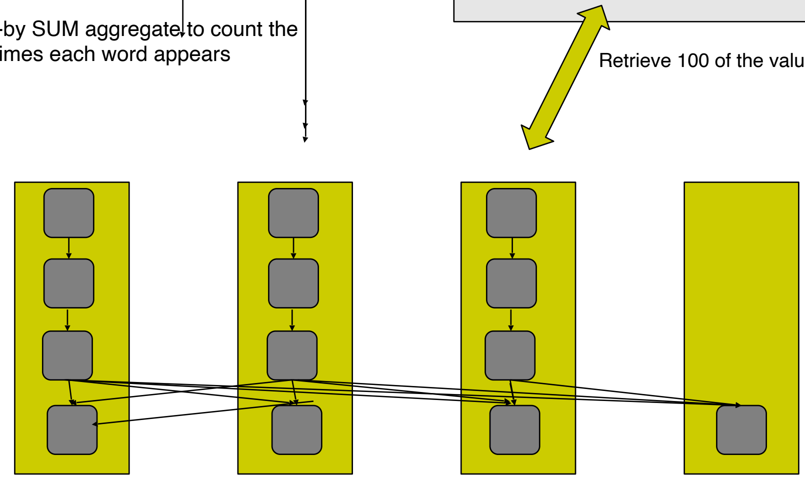
For each word, output (word, 1), creating a new RDD

Do a group-by SUM aggregate to count the number of times each word appears

Driver

```
from pyspark import SparkContext

sc = SparkContext("local", "Simple App")

textFile = sc.textFile("README.md")

counts = textFile
            .flatMap(lambda line: line.split(" "))
            .map(lambda word: (word, 1))
            .reduceByKey(lambda a, b: a + b)

print(counts.take(100))
```

Retrieve 100 of the values in the final RDD



Spark

- Operations often take in a "function" as input
- Using the inline "lambda" functionality

```
flatMap(lambda line: line.split(" "))
```

- Or a more explicit function declaration

```
def split(line):
      return line.split(" ")

flatMap(split)
```

- Similarly "reduce" functions essentially tell Spark how to do pairwise aggregation

```
reduceByKey(lambda a, b: a + b)
```

  - Spark will apply this to the dataset pair of values at a time
  - Difficult to do something like "median"

Spark: Map

x1, x2, … can be anything, including documents, images, text files, tuples, dicts, etc.

InputRDD: [x1, x2, …., x_n]

map(lambda x: x + 1)

```
def fn(x):
      return x+1
map(fn)
```

outputRDD: [x1+1, x2+1, …., x_n+1]

InputRDD: [x1, x2, …, x_n]

map(fn)

outputRDD: [fn(x1), fn(x2), …, fn(x_n)]

Spark: flatMap

InputRDD: [(a1, b1), (a2, b2), ….]

flatMap(lambda x: [x[0], x[1]])

outputRDD: [a1, b1, a2, b2, …]

InputRDD: [‘the little brown fox…’, …]

flatMap(lambda x: x.split())

outputRDD: [‘the’, ‘little’, ‘brown’, …]

Spark: groupByKey

InputRDD: [(a1, b1), (a2, b2), (a1, b3), (a1, b4), (a2, b5)…]

InputRDD must be a collection of 2-tuples
Usually called (Key, Value) pairs
Value can be anything (e.g., dicts, tuples, bytes)

groupByKey()

outputRDD: [(a1, [b1, b3, b4, …]), (a2, [b2, b5,…]), …]

Spark: reduceByKey

InputRDD: [(a1, b1), (a2, b2), (a1, b3), (a1, b4), (a2, b5)…]

InputRDD must be a collection of 2-tuples
Usually called (Key, Value) pairs

reduceByKey(func)

```
def func(V1, V2):
        return V3
```

All of V1, V2, and V3 same type

outputRDD: [(a1, …func(func(b1, b3), b4)…),
(a2, …func(func(b2, b5), …)…),]

”func” executed in parallel in a pairwise fashion

Spark: join

InputRDD1: [(a1, b1), (a2, b2), (a1, b3), (a1, b4), (a2, b5)…]
InputRDD2: [(a1, c1), (a2, c2), (a1, c3), (a1, c4), (a2, c5)…]

InputRDD1 and InputRDD2 both must be a collection of 2-tuples

↓

inputRDD1.join(inputRDD2)

↓

```
outputRDD:  [    (a1, (b1, c1)),
                 (a1, (b1, c3)),
                 (a1, (b1, c4)),
            ….]
```

Spark: cogroup

InputRDD1: [(a1, b1), (a2, b2), (a1, b3), (a1, b4), (a2, b5)…]
InputRDD2: [(a1, c1), (a2, c2), (a1, c3), (a1, c4), (a2, c5)…]

InputRDD1 and InputRDD2 both must be a collection of 2-tuples

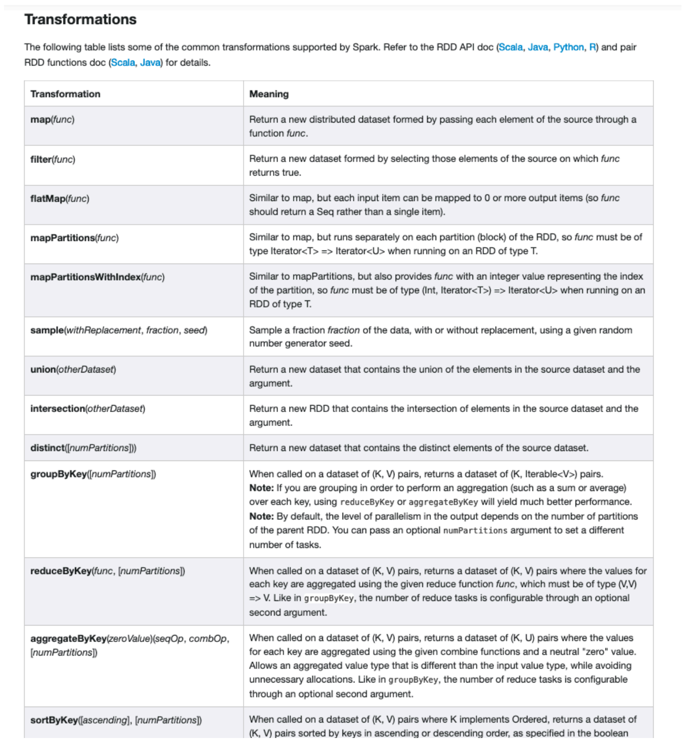
↓

inputRDD1.cogroup(inputRDD2)

↓

```
outputRDD:  [   (a1, ([b1, b3, b4, …], [c1, c3, c4, …]),
                (a2, ([b2, b5, …], [c2, c5, …]), …
            ]
```

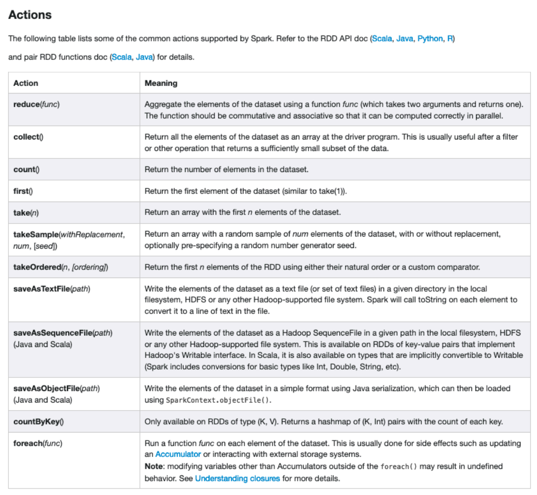
RDD Operations



Transformations

The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R) and pair RDD functions doc (Scala, Java) for details.

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numPartitions*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks.
reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean



Actions

The following table lists some of the common actions supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R)

and pair RDD functions doc (Scala, Java) for details.

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset 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.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the `foreach()` may result in undefined behavior. See Understanding closures for more details.

Dataframes Example

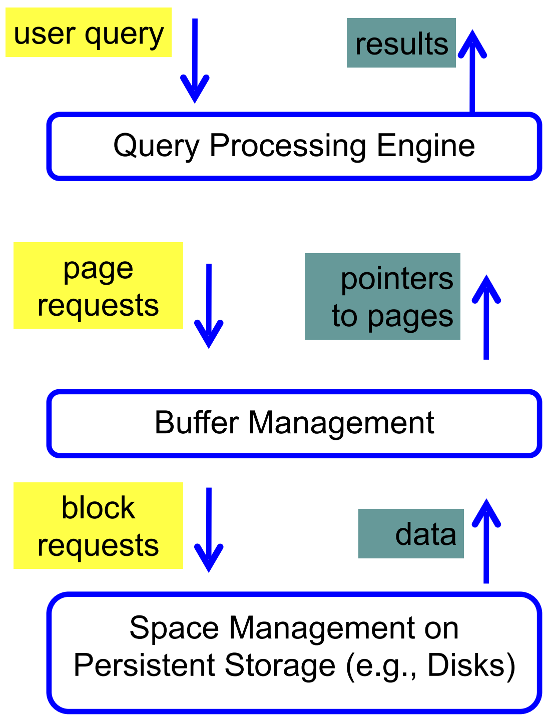
```
def basic_df_example(spark):
    # $example on:create_df$
    # spark is an existing SparkSession
    df = spark.read.json("examples/src/main/resources/people.json")
    # Displays the content of the DataFrame to stdout
    df.show()
    # +----+-------+
    # | age|   name|
    # +----+-------+
    # |null|Michael|
    # |  30|   Andy|
    # |  19| Justin|
    # +----+-------+
    # $example off:create_df$


    # $example on:untyped_ops$
    # spark, df are from the previous example
    # Print the schema in a tree format
    df.printSchema()
    # root
    # |-- age: long (nullable = true)
    # |-- name: string (nullable = true)


    # Select only the "name" column
    df.select("name").show()
    # +-------+
    # |   name|
    # +-------+
    # |Michael|
    # |   Andy|
    # | Justin|
    # +-------+


    # Select everybody, but increment the age by 1
    df.select(df['name'], df['age'] + 1).show()
    # +-------+---------+
    # |   name|(age + 1)|
    # +-------+---------+
    # |Michael|     null|
    # |   Andy|       31|
    # | Justin|       20|
    # +-------+---------+
```

```
    # Select people older than 21
    df.filter(df['age'] > 21).show()
    # +---+----+
    # |age|name|
    # +---+----+
    # | 30|Andy|
    # +---+----+


    # Count people by age
    df.groupBy("age").count().show()
    # +----+-----+
    # | age|count|
    # +----+-----+
    # |  19|    1|
    # |null|    1|
    # |  30|    1|
    # +----+-----+
    # $example off:untyped_ops$

    sqlDF = spark.sql("SELECT * FROM people")
    sqlDF.show()
    # +----+-------+
    # | age|   name|
    # +----+-------+
    # |null|Michael|
    # |  30|   Andy|
    # |  19| Justin|
    # +----+-------+
    # $example off:run_sql$


    # $example on:global_temp_view$
    # Register the DataFrame as a global temporary view
    df.createGlobalTempView("people")


    # Global temporary view is tied to a system preserved database
`global_temp`
    spark.sql("SELECT * FROM global_temp.people").show()
    # +----+-------+
    # | age|   name|
    # +----+-------+
    # |null|Michael|
    # |  30|   Andy|
    # |  19| Justin|
    # +----+-------+
```

Summary

- Spark is a popular and widely used framework for large-scale computing
- Simple programming interface
    - You don't need to typically worry about the parallelization
    - That's handled by Spark transparently
    - In practice, may need to fiddle with number of partitions etc.
- Managed services supported by several vendors including Databricks (started by the authors of Spark), Cloudera, etc.
- Many other concepts that we did not discuss
    - Shared accumulator and broadcast variables
    - Support for Machine Learning, Graph Analytics, Streaming, and other use cases
- Alternatives include: Apache Tez, Flink, and several others

Query Processing/Storage



- Given a query, decide how to "execute" it
- Specify sequence of pages to be brought in memory
- Operate upon the tuples to produce results

- Bringing pages from disk to memory
- Managing the limited memory

- Storage hierarchy
- How are relations mapped to files?
- How are tuples mapped to disk blocks?

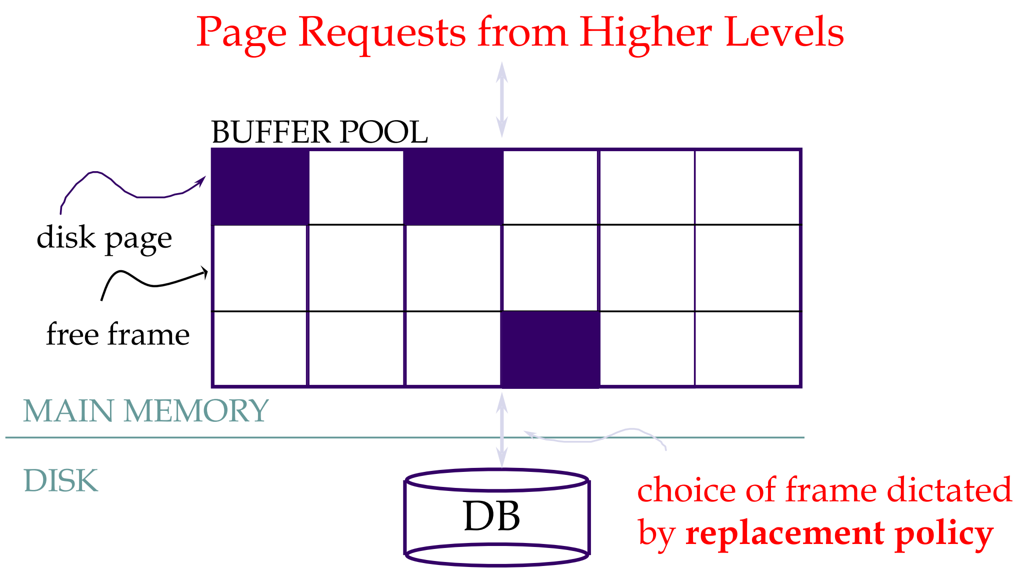
Outline

- Storage hierarchy
- Disks
- RAID
- Spark
- Buffer manager
- File Organization
- Etc….

Buffer Manager

- When the QP wants a block, it asks the “buffer manager”
  - The block must be in memory to operate upon
- Buffer manager:
  - If block already in memory: return a pointer to it
  - If not:
    - Evict a current page
      - Either write it to temporary storage,
      - or write it back to its original location,
      - or just throw it away (if it was read from disk, and not modified)
    - and make a request to the storage subsystem to fetch it

Buffer Manager


Page Requests from Higher Levels
BUFFER POOL
disk page
free frame
MAIN MEMORY
DISK
DB
choice of frame dictated by **replacement policy**

Buffer Manager

- Similar to *virtual memory manager*
- Buffer replacement policies
  - Which page to evict ?
  - LRU: Least Recently Used
    - Throw out the page that was not used in a long time
  - MRU: Most Recently Used
    - The opposite
    - If data set too big for cache, keep older pages as they might be accessed again before recent pages.
  - Clock ?
    - An efficient implementation of LRU

Buffer Manager Requirements

- *Pinning* a block
    - Not allowed to evict
- *Force-output (force-write)*
    - Force the contents of a block to be written to disk
- *Order the writes*
    - *This* block must be written to disk before *that* block

Critical for fault tolerant guarantees

- Otherwise database has no control over what is on disk

Outline

- Storage hierarchy
- Disks
- RAID
- Buffer Manager
- File Organization
- Etc….

File Organization

- How are the relations mapped to the disk blocks ?
  - Use a standard file system ?
    - High-end systems have their own OS/file systems
    - OS interferes more than helps in many cases
  - Mapping of relations to file ?
    - One-to-one ?
    - Advantages in storing multiple relations clustered together
  - A *file* is essentially a *collection of disk blocks*
    - How are the tuples mapped to the disk blocks ?
    - How are they stored within each block

File Organization

- Goals:
  - Allow insertion/deletions of tuples/records
  - Fetch a particular record (specified by record id)
  - Find all tuples that match a condition (say SSN = 123) ?
- Simplest case
  - Each relation is mapped to a file
  - A file contains a sequence of records
  - Each record corresponds to a logical tuple
- Next:
  - How are tuples/records stored within a block ?

Fixed Length Records

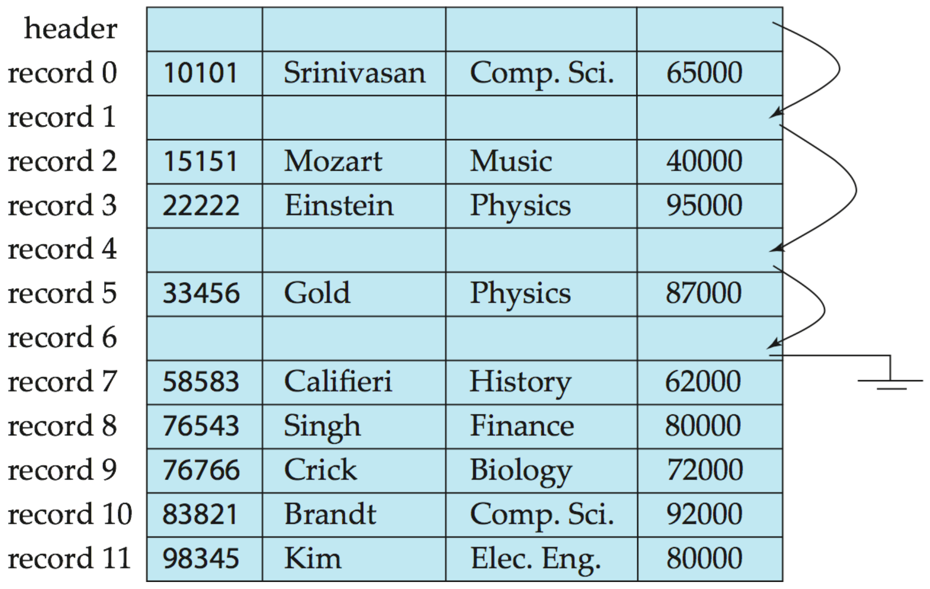
- n = number of bytes per record
- Store record *i* at position:
    - n * (i – 1)
- Records may cross blocks
    - Not desirable
    - Stagger so that that doesn't happen
- Inserting a tuple ?
    - Depends on the policy used
    - One option: Simply append at the end of the file. Problems?
- Deletions ?
    - Option 1: Rearrange
    - Option 2: Keep a *free list* and use for next insert

The above assumes records not ordered.

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Fixed Length Records

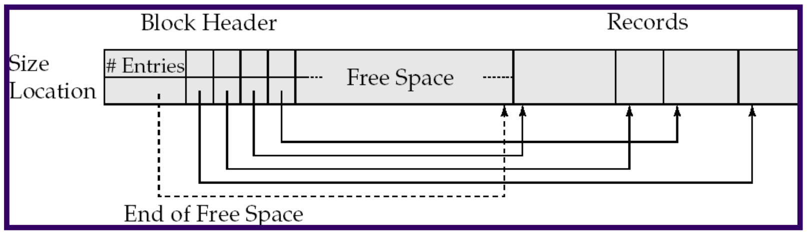
- Deleting: using "free lists"



header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Variable-length Records

Slotted page structure



- *Indirection:*
    - The records may move inside the page, but the outside world is oblivious to it
    - Why ?
        - The headers are used as an indirection mechanism
        - "Record ID 1000 is the 5th entry in page X"