Spark

- Open source system from Berkeley
- Distributed processing over HDFS
- Differences from MapReduce:
  - Multiple steps, including a fixed number of iterations
  - E.g., running Spark SQL
  - Stores intermediate results in main memory
  - Supports SQL
- Details: [http://spark.apache.org/examples.html](http://spark.apache.org/examples.html)

Spark Interface

- Spark supports a Scala interface
- Scala = ext of Java with lambda functions/closures
  - will show Scala/Spark examples shortly…
- Spark also supports a SQL interface
- It compiles SQL into Scala
- For HW6: you only need the SQL interface!

RDD

- RDD = Resilient Distributed Datasets
  - A distributed relation, together with its lineage
  - Lineage = expression that says how that relation was computed = a relational algebra plan
- Spark stores intermediate results as RDD
- If a server crashes, its RDD in main memory is lost. However, the driver (=master node) knows the lineage, and will simply re-compute the lost partition of the RDD

Programming in Spark

- A Spark/Scala program consists of:
  - Transformations (map, reduceByKey, join…). Lazy
    - Construct a new RDD from a previous one
    - Compute the new RDD at the first time it is used in an action
  - Actions (count, reduce, save…). Eager
    - Compute a result based on an RDD, and either return it to the driver program or save it to an external storage system
- RDD[T] = an RDD collection of type T
  - Partitioned, recoverable (through lineage), not nested
- Seq[T] = a Scala sequence
  - Local to a server, may be nested

Example

```scala
Given a large log file hdfs://logfile.log, retrieve all lines that:
- Start with "ERROR"
- Contain the string "sqlite"

```scala
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()
```
### Example

Given a large log file hdfs://logfile.log, retrieve all lines that:

- Start with "ERROR"
- Contain the string "sqlite"

```scala
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR")).
sqlerrors = errors.filter(_.contains("sqlite")).
sqlerrors.collect()
```

### MapReduce Again...

Steps in Spark resemble MapReduce:

- `rdd.filter(p)` applies in parallel the predicate `p` to all elements `x` of the partitioned collection / RDD, and returns those `x` where `p(x) = true`
  
  - E.g., `rdd = {1, 2, 3, 3}. rdd.filter(x => x != 1)` has result `{2, 3, 3}`

- `rdd.map(f)` applies in parallel the function `f` to all elements `x` of the partitioned collection / RDD, and returns a new partitioned collection
  
  - E.g., `rdd = {1, 2, 3, 3}. rdd.map(x => x + 1)` has result `{2, 3, 4, 4}`

### Scala Primer

- Functions with one argument:
  ```scala
  _.contains("sqlite")
  _ > 6
  ```

- Functions with more arguments:
  ```scala
  (x => x.contains("sqlite"))
  (x => x > 6)
  ((x, y) => x+3*y)
  ```

- Closures (functions using one or more variables declared outside the function):
  ```scala
  var x = 5;  rdd.filter(_ > x)
  var s = "sqlite";  rdd.filter(x => x.contains(s))
  ```

### Persistence

- By default, an RDD is re-computed each time an action is run on it.
  - `persist()` can choose to store an RDD’s content in memory or on disk, so the content can be reused in multiple actions.

```scala
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR")).
sqlerrors = errors.filter(_.contains("sqlite")).
sqlerrors.collect()
```

- If any server fails before the end, then Spark must restart
  ```scala
  errors.persist()
  sqlerrors = errors.filter(_.contains("sqlite")).
sqlerrors.collect()
  ```

- Spark can re-compute the result from errors
  ```scala
  lines = spark.textFile("hdfs://logfile.log");
  errors = lines.filter(_.startsWith("ERROR")).
sqlerrors = errors.filter(_.contains("sqlite")).
sqlerrors.collect()
  ```

- By default, an RDD is re-computed each time an action is run on it. `persist()` can choose to store an RDD’s content in memory or on disk, so the content can be reused in multiple actions.
Persistence

If any server fails before the end, then Spark must restart.

Spark can re-compute the result from errors.

R(A, B) S(A, C)

Example

SELECT count(*) FROM R, S WHERE R.B > 200 and S.C < 100 and R.A = S.A

Example Transformations

For RDDs containing key/value pairs
E.g., rdd = [(1, 2), (3, 4), (3, 6)], rdd2 = [(3, 9)]

- groupByKey()
  - Group values with the same key
  - rdd.groupByKey() has result [(1, 2), (3, 6)]

- reduceByKey()
  - Combine values with the same key
  - rdd.reduceByKey((x, y) => x + y) has result [(1, 2), (3, 10)]
Example Transformations – Cont.

For RDDs containing key/value pairs
E.g., rdd = {(1, 2), (3, 4), (3, 6)}, rdd2 = {(3, 9)}

- **mapValues()**
  - Apply a function to each value of a key/value pair without changing the key
  - rdd.mapValues(x => x + 1) has result {(1, 3), (3, 5), (3, 7)}

- **cogroup()**
  - Group data from both RDDs sharing the same key
  - rdd.group(rdd2) has result {(1, ([2], [])), (3, ([4, 6], [9]))}

Example Actions

E.g., rdd = {1, 2, 3, 3}

- **count()**
  - Number of elements in the RDD
  - rdd.count() has result 4

- **reduce()**
  - Combine the elements of the RDD together in parallel
  - rdd.reduce((x, y) => x + y) has result 9

MapReduce ~> Spark

- input into an RDD
- map phase becomes .flatMap
- shuffle & sort becomes .groupByKey
- reduce becomes another .flatMap
- save output to HDFS

SQL ~> Spark

- You know enough to execute SQL on Spark!
- Idea: (1) SQL to RA + (2) RA on Spark
  - σ = filter
  - π = map
  - γ = groupByKey
  - Χ = cartesian
  - ¥ = join
- Spark SQL does small optimizations to RA
- Also chooses between broadcast and parallel joins

PageRank

- PageRank is an algorithm that assigns to each page a score, such that pages have higher scores if more pages with high scores link to them
- PageRank was introduced by Google, and essentially defined Google

Purpose of PageRank

- Compute $\rho(d)$, the prior probability of the document $d$ for retrieval purpose
- Not all Web pages are equally important
  - E.g., pages on popular Web sites tend to be more important
- Give weights to Web pages based on how often they are hyperlinked by other Web pages
  - Hyperlink = citation
  - More citations $\Rightarrow$ more important
Model behind PageRank: Random Walk

• Imagine a Web surfer doing a random walk on the Web
  – Start at a random page
  – At each step, go out of the current page along one of the links on the page
  – Each link is chosen with equal probability
• In the steady state, each page has a long-term visit rate
  – Called the page’s PageRank
  – It does not matter where the surfer starts
• PageRank = long-term visit rate = steady state probability

Random Walk – Cont.

• A Markov chain consists of \( N \) states + an \( N \times N \) transition probability matrix \( P \)
  • state = page
• At each step, the Web surfer is on exactly one page, say page \( i \)
  • For \( 1 \leq i, j \leq N \), the matrix entry \( P_{ij} \) is the probability of moving from page \( i \) to page \( j \) in the next step
  • For every \( i \), \( \sum_{j=1}^{N} P_{ij} = 1 \)

Example Web Graph

Transition Probability Matrix
**Ergodicity Theorem**

- Theorem in stochastic processes: Web-graph + teleporting has a steady-state probability distribution
  - Each page in the Web-graph + teleporting has a PageRank

- Steady state probability vector \( \mathbf{\pi} = (\pi_1, \pi_2, \ldots, \pi_N) \)
  - \( \pi_i \) is the long-term visit rate (or PageRank) of page \( i \)

**Probability Vector**

- At a specific step, a probability (row) vector \( \mathbf{X} = (x_1, \ldots, x_N) \) tells us where the random walk is at
  - The random walk is on page \( i \) with probability \( x_i \)
  - \( \sum_{i=1}^{N} x_i = 1 \)
- Example:
  - \( (0.1 \ 0.2 \ 0.3 \ 0.15 \ 0.25) \)

**Change in Probability Vector**

- If the probability vector in the current step is \( \mathbf{X} = (x_1, \ldots, x_N) \), the probability vector in the next step is \( \mathbf{X} \mathbf{P} \)
  - In the next step, the random walk is on page \( j \) with probability \( \sum_{i=1}^{N} x_i \cdot P_{ij} \)

**Compute the Steady State Probability Vector**

- Suppose the distribution has reached the steady state \( \mathbf{\pi} = (\pi_1, \pi_2, \ldots, \pi_N) \) in the current step
- The distribution in the next step is \( \mathbf{\pi} \mathbf{P} \), which should also be in steady state
- So \( \mathbf{\pi} = \mathbf{\pi} \mathbf{P} \)
- Solving this matrix equation gives us \( \mathbf{\pi} \)
  - \( \mathbf{\pi} \) is the principal left eigenvector for \( \mathbf{P} \)
  - i.e., the left eigenvector with the largest eigenvalue

**Another Way of Writing \( \mathbf{\pi} = \mathbf{\pi} \mathbf{P} \)**

- Assume no dead end for now
- Suppose pages \( T_1, \ldots, T_m \) have links to page \( A \)
- \( C(T_i) \): the number of links going out of page \( T_i \)

\[
\text{PageRank}(A) = \frac{r}{N} + \left(1 - r\right)\left[ \frac{\text{PageRank}(T_1)}{C(T_1)} + \cdots + \frac{\text{PageRank}(T_m)}{C(T_m)} \right]
\]
One Way of Computing the PageRank

- Start with any distribution $X$
- E.g., uniform distribution
- After one step, we get $XP$
- After two steps, we get $XP^2$
- After $k$ steps, we get $XP^k$
- Algorithm: multiply $X$ by increasing powers of $P$ until convergence
- This is called the power method

PageRank

Random walk interpretation:

Start at a random node $i$
At each step, randomly choose an outgoing link and follow it.
Repeat for a very long time

$x[i] = \text{prob. that we are at node } i$

Google Dataflow

- Similar to Spark/Scala
- Allows you to lazily build pipelines and then execute them
- Much simpler than multi-job MapReduce

Summary

- Parallel databases
  - Pre-defined relational operators
  - Optimization
  - Transactions
- MapReduce
  - User-defined map and reduce functions
  - Must manually implement/optimize relational operators
  - No updates/transactions
- Spark
  - Pre-defined relational operators
  - Must manually optimize
  - No updates/transactions
Summary cont.

- All of these technologies use **dataflow engines**:
  - Google Dataflow (on top of MapReduce)
  - Spark (on top of Hadoop)
  - AsterixDB (on top of Hyracks)

- Spark & AsterixDB map SQL to a dataflow pipeline
  - SQL $\rightarrow$ RA $\rightarrow$ dataflow operators (group, join, map)
  - could do the same thing for Google Dataflow

- None of these systems optimize RA very well (as of 2015)
  - Spark has no indexes
  - AsterixDB has indexes, but no statistics

- Future work should improve that