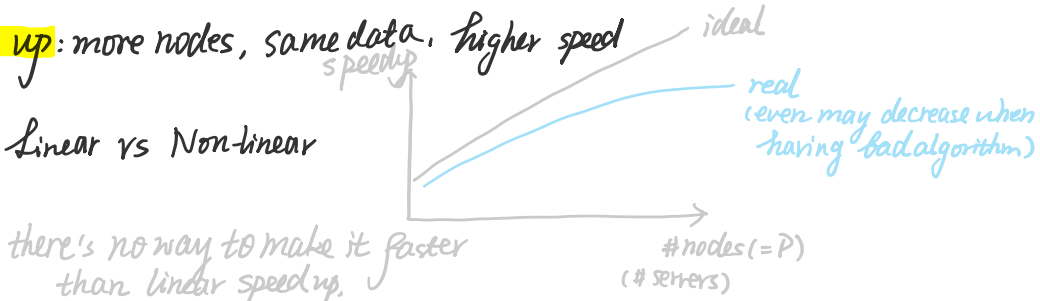
Spark

Wednesday, May 15, 2019 1:30 PM

- Why parallism:
  - Multi-cores
  - big data too large to fit in main memory.
    - distributed query processing on 100x-1000x servers
- Performance Matrix for DBMS
  - Nodes: processors, computers
  - speed up: more nodes, same data, higher speed
    - Linear vs Non-linear



  - scale up: more nodes, same speed, more data
    - Linear vs Non-linear scaleup

  - Reason for sub-linear:

- start up cost — cost of an starting operation on many nodes
- interference — contention for resources between nodes
- skew — slowest node becomes the bottleneck

- **Spark** — A case study of the Map Resource Programing paradigm.
  - Distributed processing over HDFS.
  - Supports interfaces in Java, Scala, Python.
  - Difference from MapReduce
    - Multiple steps including iterations
    - Store intermediate result in main memory
    - Closer to relational algebra

  - A spark program
    - **Transformation** (map, join, reduce) **Lazy**
    - **Actions** (count, reduce, save) **Eager**
      - Lazy: operator tree is constructed in memory
      - Eager: operators are executed immediately
  - Collections:
    - **RDD<T>** = an RDD collection of type T
      - not nested, distributed
      - executed in parallel
      - Recoverable via lineage

- Seq<T> = a sequence
  - local to one server, may be nested
  - done sequentially

e.g., `s = SparkSession.builder()...getOrCreate();`

```
sqlerrors = s.read().textFile("hdfs://logfile.log")
            .filter(l -> l.startsWith("ERROR"))
            .filter(l -> l.contains("sqlite"))
            .collect();
```

transformation

action

↑ Call chaining style

write the functions as a chain

- Anonymous functions (Lambda Expressions)

```
errors = lines.filter(l -> l.startsWith("Error"));
```

⇕

```
class FilterFn implements Function<Row, Boolean>{
    Boolean call(Row r)
    { return l.startsWith("Error"); }
}
errors = lines.filter(new FilterFn());
```

- Fault Tolerance

When a job is executed on many servers, the probability of failure is high.

$P(fail) = \sum P_i(fail)$

is the sum of P of all servers.

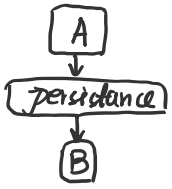
- soln:
  - Parallel database system — restart the whole thing (Expensive)
  - MapReduce — write everything to disk, redo. (Slow)

- Spark redo only what's needed. (efficient)

- **RDD = Resilient Distributed Dataset**
  - Distributed, immutable (never move/change), and records 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 recompute the lost partition of the RDD.

- **Persistence** : a kind of immediate materialization

```
A = ...
A.persit();   ← create a new RDD
B = A. ...
```



Transformations:	
map(f : T -> U):	RDD<T> -> RDD<U> (perform f on each T.)
flatMap(f: T -> Seq(U)):	RDD<T> -> RDD<U> (perform f on T, get a sequence, then flat it)
filter(f:T->Bool):	RDD<T> -> RDD<T>
groupByKey():	RDD<(K,V)> -> RDD<(K,Seq[V])>
reduceByKey(F:(V,V)-> V):	RDD<(K,V)> -> RDD<(K,V)>
union():	(RDD<T>,RDD<T>) -> RDD<T>
join():	(RDD<(K,V)>,RDD<(K,W)>) -> RDD<(K,(V,W))>
cogroup():	(RDD<(K,V)>,RDD<(K,W)>)-> RDD<(K,(Seq<V>,Seq<W>))>
crossProduct():	(RDD<T>,RDD<U>) -> RDD<(T,U)>

Actions:	
count():	RDD<T> -> Long
collect():	RDD<T> -> Seq<T>

reduce(f:(T,T)->T):	RDD<T> -> T
save(path:String):	Outputs RDD to a storage system e.g., HDFS

- Three Java-Spark APIs
  - RDD JavaRDD<T>
    - basically untyped
    - distributed, main memory
  - Data frames DataSet<Row>
    - Row: a record, dynamically typed.
    - Distributed, main memory, or external (e.g. SQL)
  - Datasets Dataset<Person>
    - <Person>: user defined type
    - distributed, main memory (not external)
- Data frames
  - like RDD, also immutable distributed collection of data.
  - Organized into named columns rather than individual objects (like a relation)
  - Similar API as RDDs with additional methods.
- Dataset

- Similar to DataFrames, except that the element must be typed.
- Can detect errors during compilation time

Datasets API: Sample Methods

- Functional API
  - `agg(Column expr, Column... exprs)`
    Aggregates on the entire Dataset without groups.
  - `groupBy(String col1, String... cols)`
    Groups the Dataset using the specified columns, so that we can run aggregation on them.
  - `join(Dataset<?> right)`
    Join with another DataFrame.
  - `orderBy(Column... sortExprs)`
    Returns a new Dataset sorted by the given expressions.
  - `select(Column... cols)`
    Selects a set of column based expressions.
- "SQL" API
  - `SparkSession.sql("select * from R");`