Database Systems
CSE 414

Lecture 26: Spark

Announcements

- HW8 due next Fri
- Extra office hours today: Rajiv @ 6pm in CSE 220
- No lecture Monday (holiday)
- Guest lecture Wednesday
  – Kris Hildrum from Google will be here
  – she works on technologies related to Spark etc.
  – whatever she talks about will be on the final

Spark

- Open source system from Berkeley
- Distributed processing over HDFS
- Differences from MapReduce:
  - Multiple steps, including iterations
  - Stores intermediate results in main memory
  - Supports SQL
- Details: http://spark.apache.org/examples.html

Spark Interface

- Spark supports a Scala interface
- Scala = ext of Java with functions/closures
  - will show Scala/Spark examples shortly…

- Spark also supports a SQL interface
- It compiles SQL into Scala
- For HW8: 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 recompute the lost partition of the RDD

Programming in Spark

- A Spark/Scala program consists of:
  - Transformations (map, reduce, join…). Lazy
  - Actions (count, reduce, save...). Eager
- 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

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

- Start with “ERROR”
- Contain the string “sqlite”

```
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”

```
lines = spark.textFile(“hdfs://logfile.log”);

errors = lines.filter(_.startsWith(“ERROR”));

sqlerrors = errors.filter(_.contains(“sqlite”));

sqlerrors.collect()
```

Transformation:
Not executed yet…

Action:
triggers execution
of entire program

MapReduce Again…

Steps in Spark resemble MapReduce:

- col.filter(p) applies in parallel the predicate p to all elements x of the partitioned collection, and returns those x where p(x) = true
- col.map(f) applies in parallel the function f to all elements x of the partitioned collection, and returns a new partitioned collection

Scala Primer

- Functions with one argument:

```
_.contains("sqlite")
_ > 6
```

- Functions with more arguments

```
(x => x.contains("sqlite"))
(x => x > 6)
((x,y) => x+3*y)
```

- Closures (functions with variable references):

```
var x = 5;   rdd.filter(_ > x)
var s = "sqlite";  rdd.filter(x => x.contains(s))
```

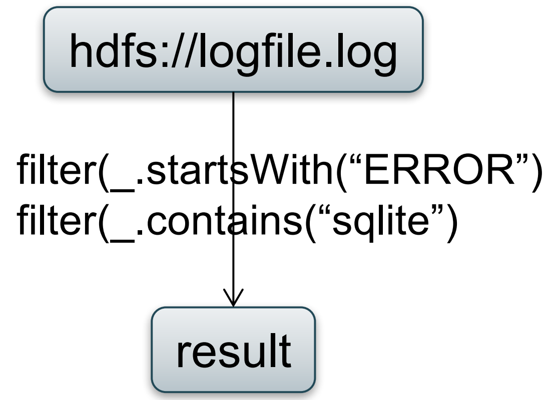
Persistence

```
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

Persistence

RDD:

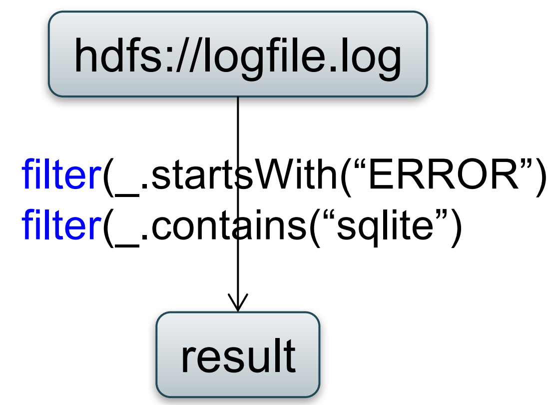



```
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()
```

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

Persistence

RDD:




```
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

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

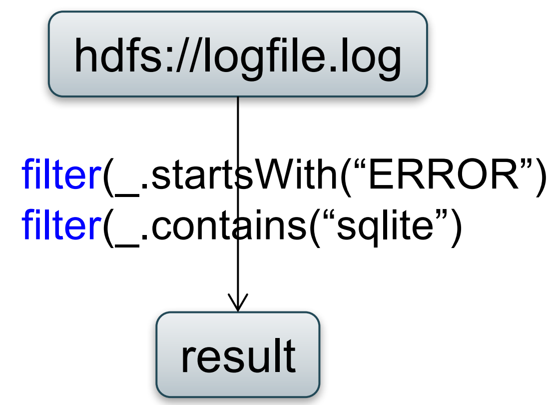
New RDD

Spark can recompute the result from errors

Persistence

RDD:

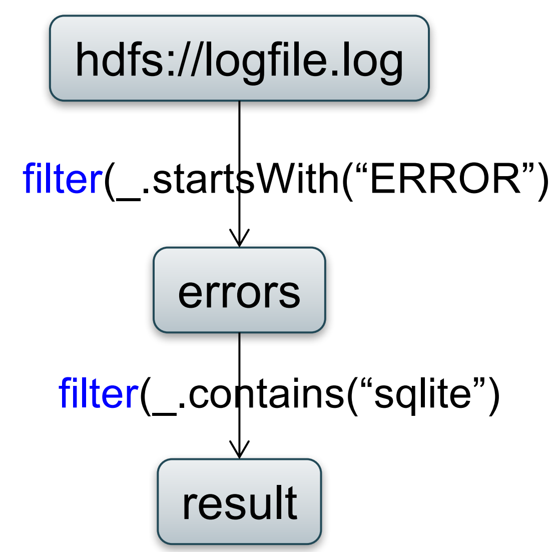
```
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()
```



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

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

New RDD



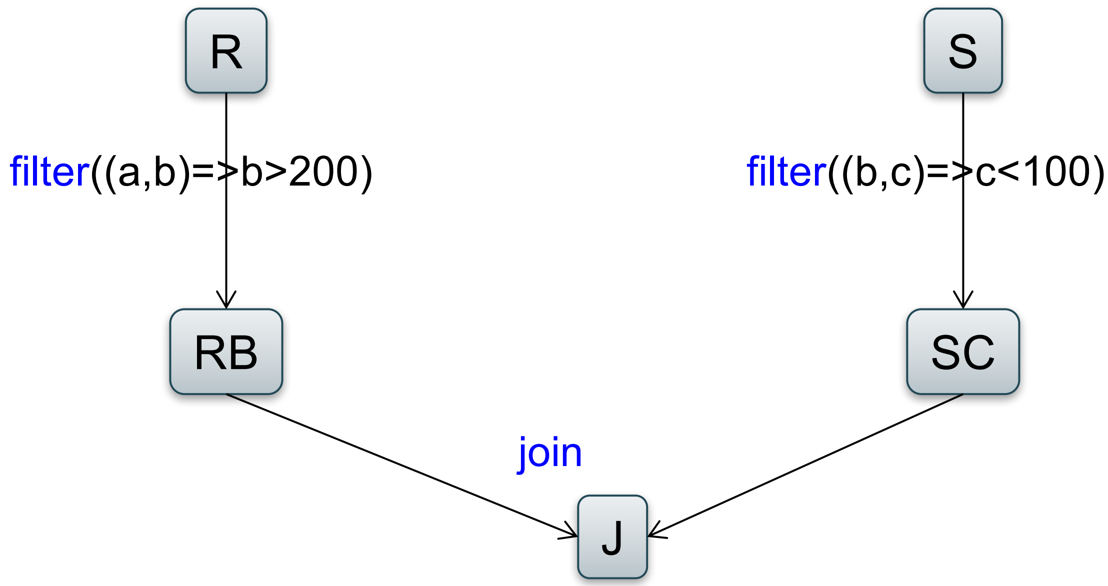
Spark can recompute the result from errors

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

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

Example

```
R = spark.textFile("R.csv").map(parseRecord).persist()
S = spark.textFile("S.csv").map(parseRecord).persist()
RB = R.filter((a,b) => b > 200).persist()
SC = S.filter((a,c) => c < 100).persist()
J = RB.join(SC).persist()
J.count();
```



Programming in Spark

- A Spark/Scala program consists of:
  - Transformations (map, reduce, join…). Lazy
  - Actions (count, reduce, save...). Eager
- 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

Transformations:	
map(f : T => U):	RDD[T] => RDD[U]
flatMap(f: T => Seq[U]):	RDD[T] => RDD[U]
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

MapReduce ~> Spark

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

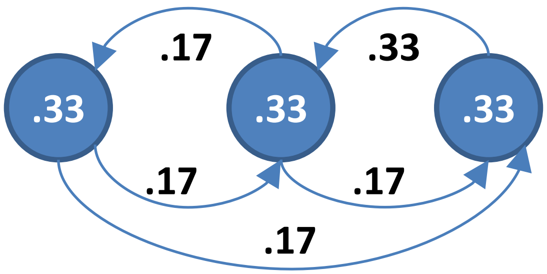
SQL ~> Spark

- You know enough to execute SQL on Spark!
- Idea: (1) SQL to RA + (2) RA on Spark
  - σ = filter
  - π = map
  - γ = groupByKey
  - $\times$ = crossProduct
  - $\bowtie$ = join
- Spark SQL does small optimizations to RA
- Also chooses btw broadcast and parallel joins

PageRank

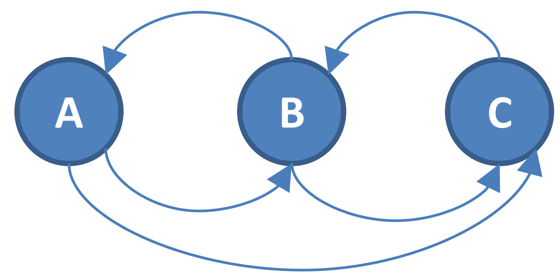
- Page Rank is an algorithm that assigns to each page a score such that pages have higher scores if more pages with high scores link to them
- Page Rank was introduced by Google, and, essentially, defined Google

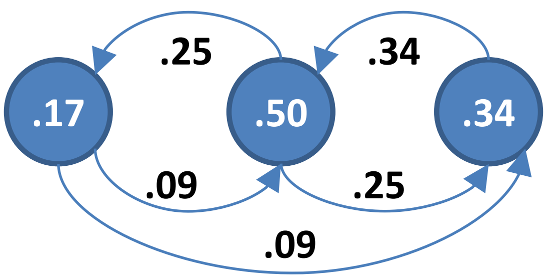
PageRank toy example



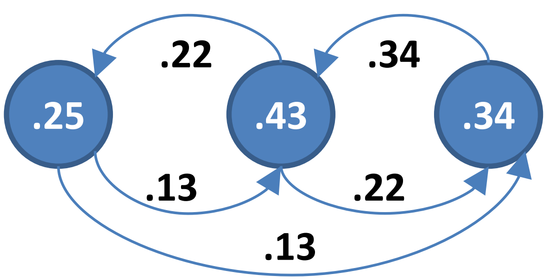
Superstep 0

Input graph





Superstep 1



Superstep 2

http://www.slideshare.net/sscdotopen/large-scale/20

PageRank

```
for i = 1 to n:
  r[i] = 1/n

repeat
  for j = 1 to n: contribs[j] = 0
  for i = 1 to n:
    k = links[i].length()
    for j in links[i]:
       contribs[j] += r[i] / k
   for i = 1 to n: r[i] = contribs[i]
until convergence
/* usually 10-20 iterations */
```

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

r[i] = prob. that we are at node i

PageRank

```
for i = 1 to n:
   r[i] = 1/n

repeat
   for j = 1 to n: contribs[j] = 0
   for i = 1 to n:
      k = links[i].length()
      for j in links[i]:
         contribs[j] += r[i] / k
    for i = 1 to n: r[i] = contribs[i]
until convergence
/* usually 10-20 iterations */
```

Random walk interpretation:

Start at a random node i
At each step, randomly choose an outgoing link and follow it.

Improvement: with small prob. a restart at a random node.

r[i] = a/N + (1-a)*contribs[i]

where a $\in$(0,1) is the restart probability

links: RDD[url:string, links:SEQ[string]]
ranks: RDD[url:string, rank:fload]

PageRank

```
for i = 1 to n:
   r[i] = 1/n

repeat
   for j = 1 to n: contribs[j] = 0
   for i = 1 to n:
      k = links[i].length()
      for j in links[i]:
          contribs[j] += r[i] / k
    for i = 1 to n: r[i] = a/N + (1-a)*contribs[i]
until convergence
/* usually 10-20 iterations */
```

```
// SPARK
val links = spark.textFile(..).map(..).persist()
var ranks = // RDD of (URL, 1/n) pairs
for (k <- 1 to ITERATIONS) {
  // Build RDD of (targetURL, float) pairs
  // with contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url,  (links,rank)) =>
       links.map(dest => (dest, rank/links.size))
   }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
       .mapValues(sum => a/n + (1-a)*sum)
}
```

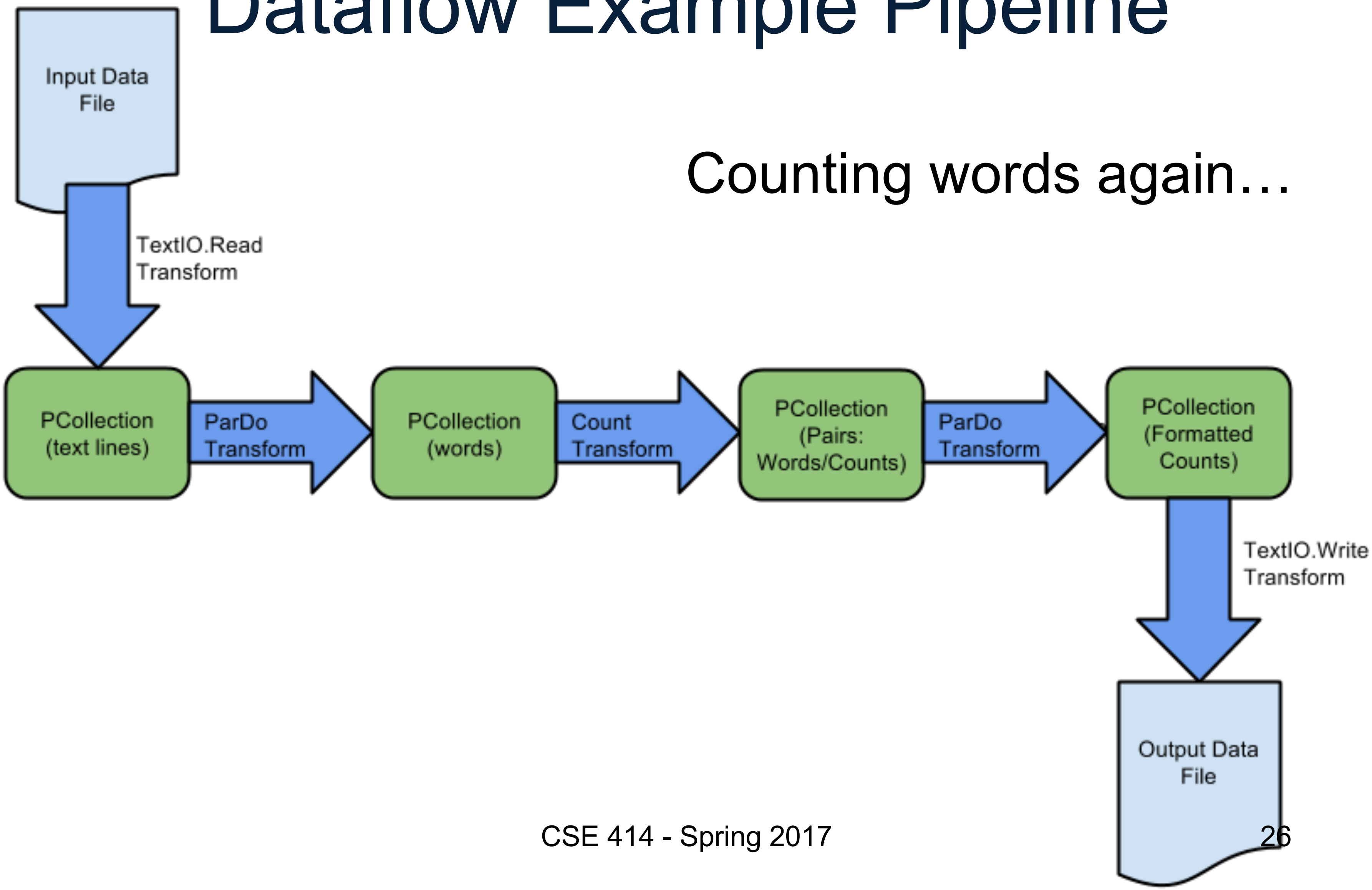
Google Dataflow

- Similar to Spark/Scala
- Allows you to lazily build pipelines and then execute them

- Much simpler than multi-job MapReduce

Dataflow Example Pipeline

Counting words again…



Dataflow Example Code

Read lines into PCollection

```
Pipeline p = Pipeline.create(options);

p.apply(TextIO.Read.from(
    "gs://dataflow-samples/shakespeare/kinglear.txt"))

  .apply(ParDo.named("ExtractWords").of(new DoFn<String, String>() {
     @Override
     public void processElement(ProcessContext c) {
       for (String word : c.element().split("[^a-zA-Z']+")) {
         if (!word.isEmpty()) {
           c.output(word);
         }
       }
     }
  }))
```

map line to bag of words

Dataflow Example Code cont.

```
  .apply(Count.<String>perElement())

  .apply(MapElements.via(new SimpleFunction<KV<String, Long>, String>() {
     @Override
     public String apply(KV<String, Long> element) {
       return element.getKey() + ": " + element.getValue();
     }
  }))

  .apply(TextIO.Write.to("gs://my-bucket/counts.txt"));

p.run();
```

built-in routine to count occurrances

(“foo”, 3) ~> “foo: 3”

Write results into GFS

execute now

Summary

- Parallel databases
  - Predefined relational operators
  - Optimization
  - Transactions
- MapReduce
  - User-defined map and reduce functions
  - Must implement/optimize manually relational ops
  - No updates/transactions
- Spark
  - Predefined relational operators
  - Must optimize manually
  - 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 ~> RA ~> 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