St. Louis Clojure

Powderkeg

Christopher Mark Gore

cgore.com

Tuesday, March 21, AD 2017

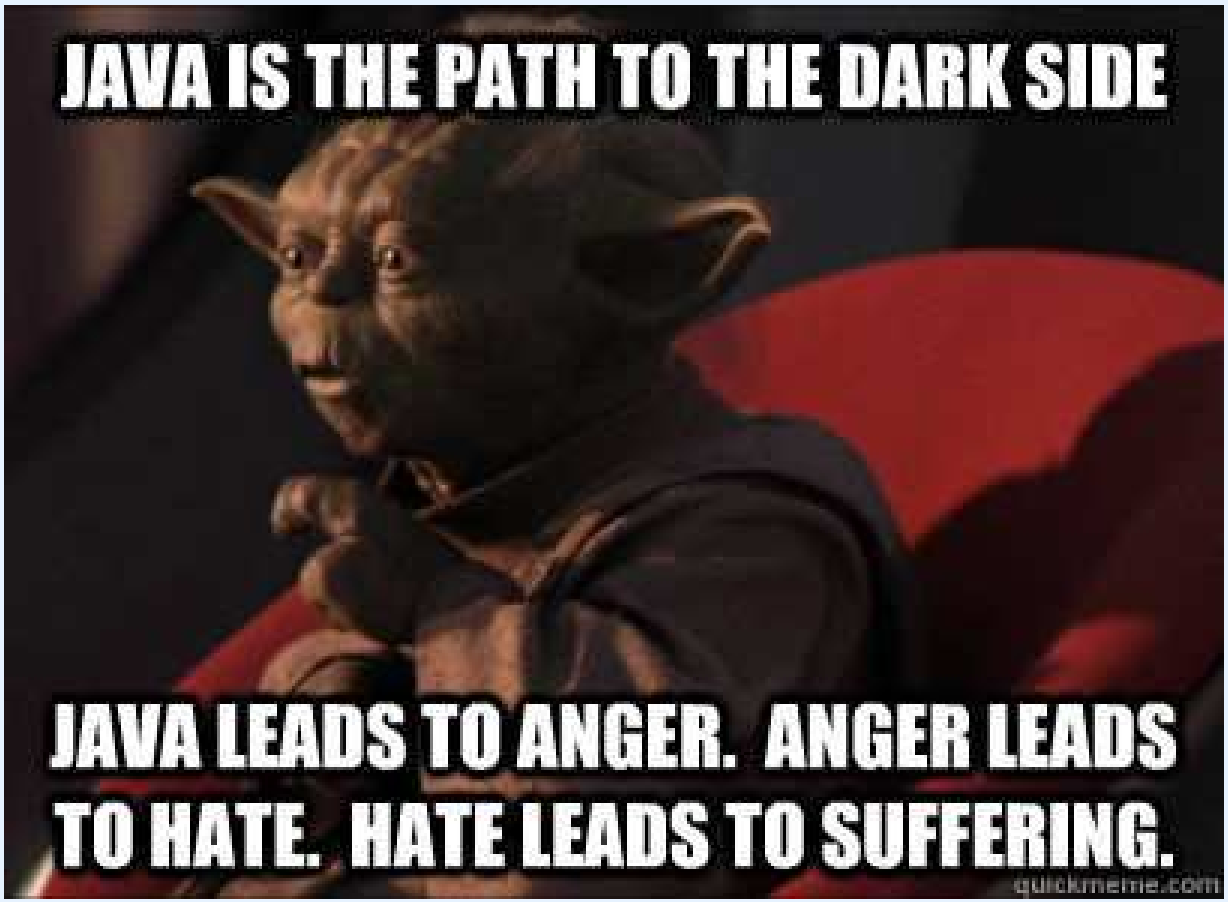
We write Clojure at The Climate Corporation, and we're hiring! Come work with us!

Especially now that Bayer is buying us!

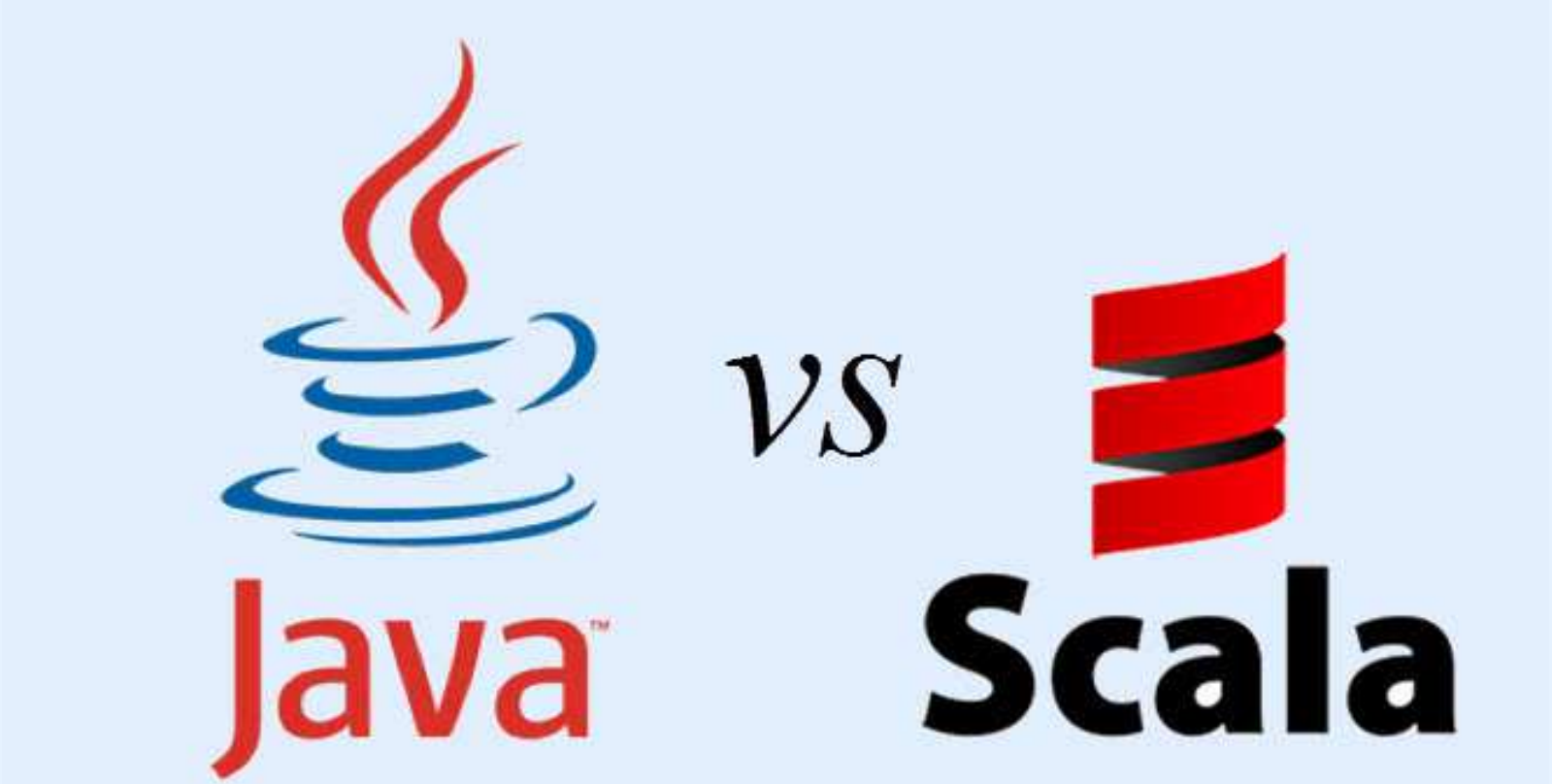
It's a pretty cool place to work, we've even got a giant globe to play with.

Clojure is a lisp.

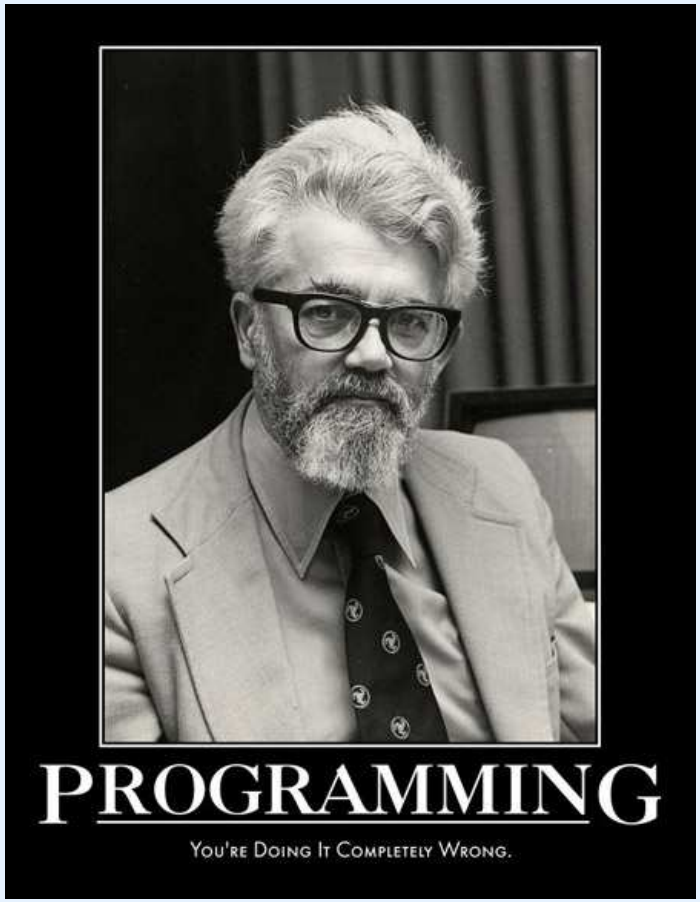
Clojure is a lisp on the JVM.



Scala is also on the JVM.



But Scala isn't a lisp.



Apache Spark is really cool, but it's in Scala.

Apache Spark is an open source cluster computing framework.

- Based on the RDD, resilient distributed dataset.
- An RDD is basically a distributed read-only multiset.
- RDDs allow for the power of MapReduce but with a lot more flexibility.
- RDDs can be treated as shared memory.
- This allows for iterative algorithms, not just map and then reduce operations.

What's a Clojurian to do?

- Use Scala? Nope.
- Here at Climate, we made `clj-spark` well before I came to work here, which was a good start.
- This eventually became Flambo, which is pretty good, but doesn't exactly feel like Clojure.

Let's make another library!

- Igor Ges and Christophe Grand introduced a new library called *Powderkeg* to work with Apache Spark in Clojure.
- It looks like normal Clojure code, thanks to transducers!
- But . . . it's still really early alpha.

It looks almost like normal Clojure.

```
1  ;; 'normal' Clojure
2  (into [] (map #(* % %))
3                [1 2 3 4 5])
4
5  ;; Flambo
6  (-> (f/parallelize sc [1 2 3 4 5])
7      (f/map (f/fn [x] (* x x)))
8      f/collect)
9
10 ;; Powderkeg
11 (into [] (keg/rdd [1 2 3 4 5]
12                   (map #(* % %))))
```

So what exactly do you do with a Spark cluster?

- Make a big list of stuff, and RDD.
- Map on that RDD …
- Filter down that RDD …
- Reduce on that RDD …
- …

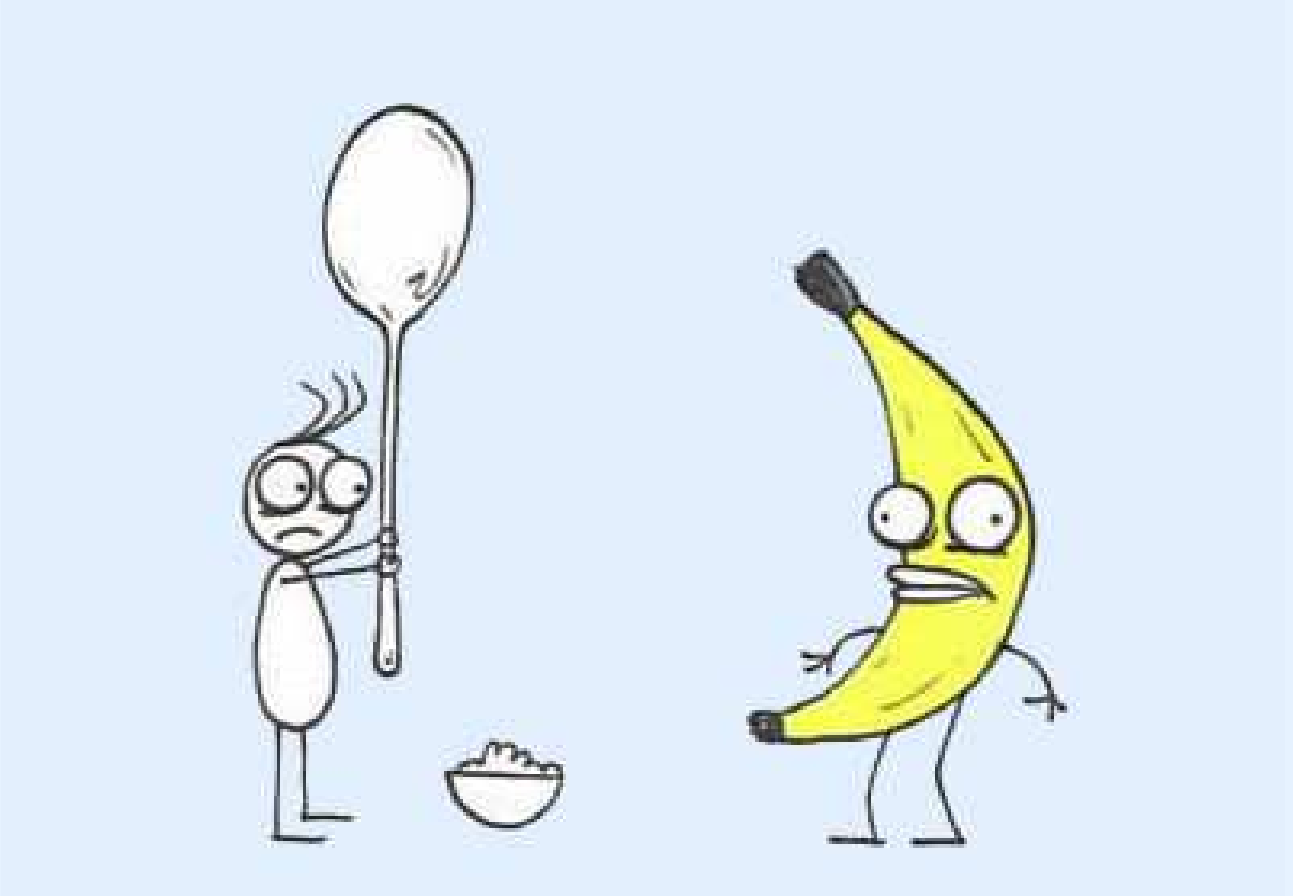
Until you have the final result of your computation. But, all this mapping, reducing, and filtering has occurred on multiple machines, not just one machine.

Why we care at Climate
(besides it just being cool.)

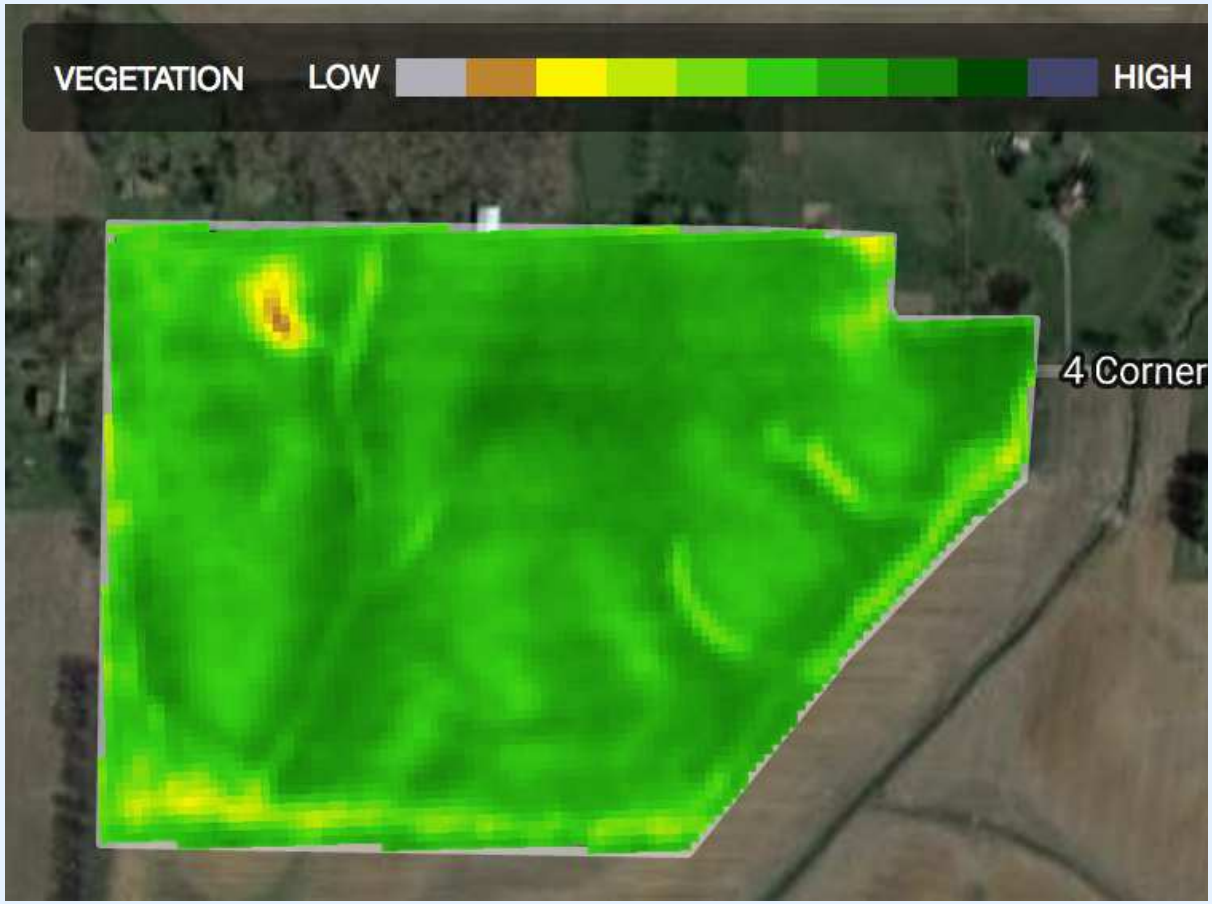
- Most of our imagery generation is quite amenable to this sort of parallellization.
- We currently have to run on somewhat large (and expensive) instances because we currently operate on a per-field basis.
- The instance type required for a 100-acre field is a lot cheaper than what we need to calculate on a 3,000-acre field.
- But if we can span across multiple instances per-field, then we can use smaller and cheaper instances, just more of them.

Why *you* care at *YourMegaCorp, Inc.* *(besides it just being cool.)*

Actually, there's various limits to the sizes of things, so *biggish* data.



Pictures of corn … *from space.*



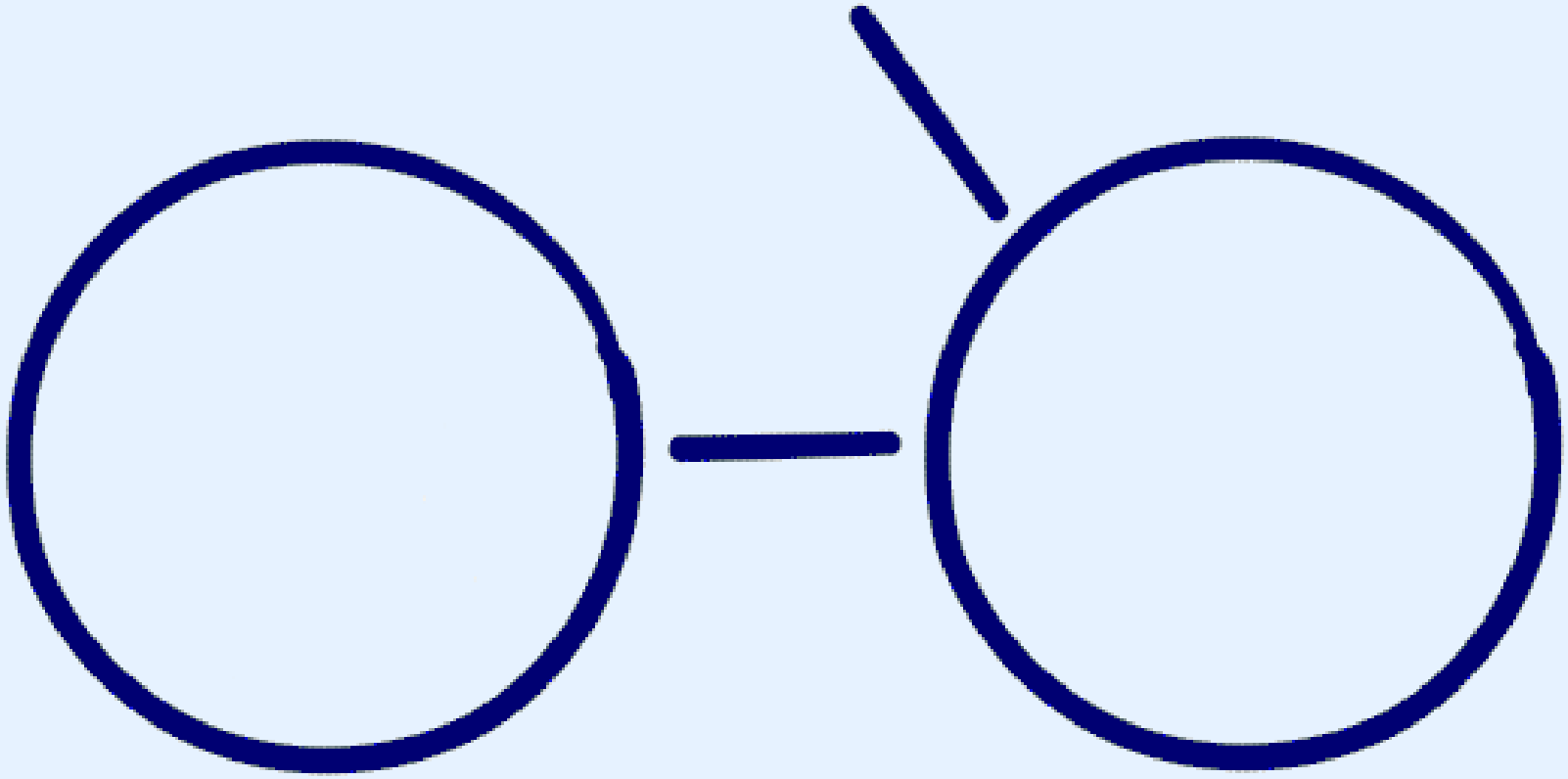
Spark SQL – there's work in progress to move this over.

https://github.com/HCADatalab/powderkeg/tree/sql

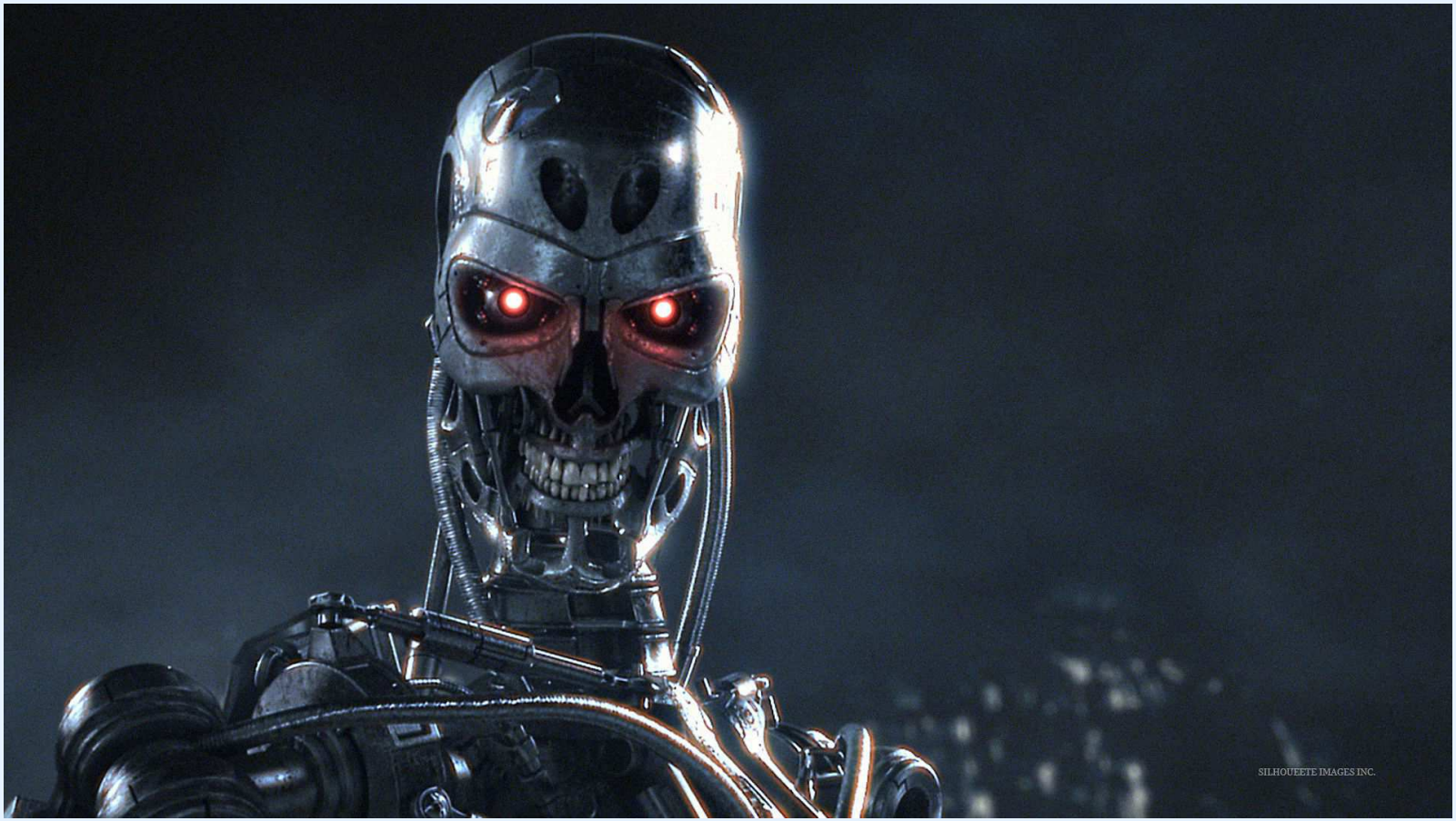
```
1  // Select customer name column
2  dfCustomers.select("name").show()
3  // Select customer name and city columns
4  dfCustomers.select("name", "city").show()
5  // Select a customer by id
6  dfCustomers.filter(
7    dfCustomers("customer_id").equalTo(500)
8  ).show()
9  // Count the customers by zip code
10 dfCustomers.groupBy("zip_code").count().show()
```

Spark Streaming – eventually, I don't think there's any work on that yet.

You can implement a batch-oriented lambda architecture in this, which is something that I need for Thinking Bicycle.

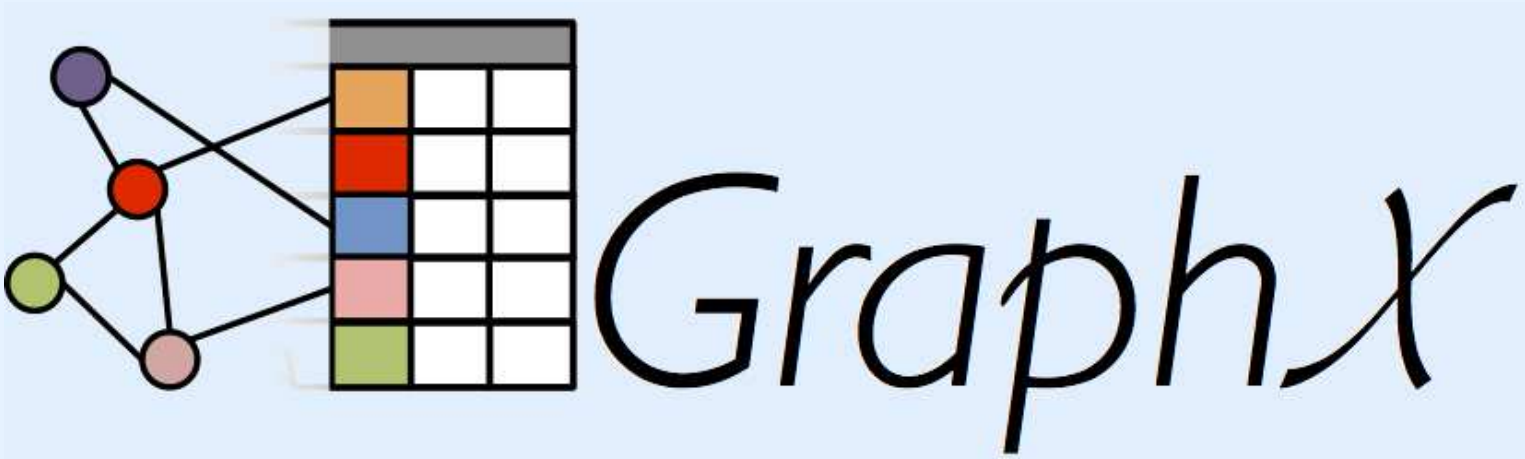


MLlib: Machine learning is cool, and there's a library for that on top of Spark. I really want it, so I'll probably make it.



MLlib's SVMs *(support vector machines)* more specifically. Because clouds suck when you've only got civilian satellites.

GraphX, a graph processing framework on top of Apache Spark, also looks pretty interesting and useful.



RDDs have *actions* which return values (1/2).

reduce aggregate the elements with some function *(think summation and the + function)*

collect return a normal array

count how many items in the RDD

first the first item in the RDD

take return a normal array of n items

takeSample random sampling of n elements

takeOrdered first n elements based on an ordering

RDDs have *actions* which return values (2/2).

saveAsTextFile write out a simple text file

saveAsSequenceFile write out a Hadoop SequenceFile

saveAsObjectFile write out a simple serialized file

countByKey for (k, v) RDDs, return (k, n) of counts

foreach run a function over the entire RDD for side effects

RDDs have *transformations* which return new RDDs (1/3).

map new RDD by mapping f

filter new RDD filtered for f is true

flatMap non-bijective map (non-1:1)

mapPartitions like map, but running separately on each partition (block) of the RDD

mapPartitionsWithIndex like *mapPartitions*, but also with an integer index of the partition

sample random sample of the RDD

RDDs have *transformations* which return new RDDs (2/3).

union union of two RDDs

intersection intersection of two RDDs

distinct the distinct (unique) elements as a new RDD

groupByKey (k, v) to $(k, \{v_1, v_2, \ldots\})$

reduceByKey like reduce, but on a per-key basis

aggregateByKey per-key merge of the RDD

sortByKey an RDD sorted by the keys

RDDs have *transformations* which return new RDDs (3/3).

join (k, v) and (k, w) become $(k, (v, w))$

cogroup groups two RDDs together on their shared keys

cartesian this is the reason why some people shouldn't be allowed near databases

pipe pipe through a shell command

coalesce reduce the number of partitions (blocks)

repartition set the number of partitions (blocks)

repartionAndSortWithinPartitions more efficient than a repartition and then sort

Questions?