PDD

Lecture 3: Spark SQL

Prepared by Jacek Sroka

SparkSQL

- Big data/Spark applications
  - Algorithms for Big data
  - **Data analytics**
  - Stream processing
    - Also SQL over streams
  - Machine learning
    - MLlib
  - Graphs
    - GraphFrames
- Is Spark RDD cool?
- What are the problems?

SparkSQL

- Big data/Spark applications
  - Algorithms for Big data
  - **Data analytics**
  - Stream processing
    - Also SQL over streams
  - Machine learning
    - MLlib
  - Graphs
    - GraphFrames

- Is Spark RDD cool?
- What are the problems?
- Data analytics to cumbersome for non-programmers (programmers?)

```
my_rdd.map(lambda x: (x.dept, [x.age, 1])) \
   .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
   .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
   .collect()
```

SparkSQL

- Data analytics to cumbersome for non-programmers (programmers?)

```
my_rdd.map(lambda x: (x.dept, [x.age, 1])) \
   .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
   .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
   .collect()
```

- What abstraction/language would be more appropriate?

SparkSQL

- Data analytics to cumbersome for non-programmers (programmers?)

```
my_rdd.map(lambda x: (x.dept, [x.age, 1])) \
   .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
   .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
   .collect()
```

- What abstraction/language would be more appropriate?

```
SELECT detp, AVG(age) FROM my_rdd GROUP BY dept
```

```
my_rdd.groupBy(”dept”).agg(avg(”age”))
```

- Can the framework add some optimisations?

SparkSQL

- RDDs have no schema and we run arbitrary code in our lambdas
  - Spark does not understand what is inside and cannot optimize for it
  - Schema allows for columnar caching
  - Schema supports declarative programming
  - Additional optimizations possible for **declarative** programs (queries)
  - Additional low level optimizations possible if we understand what is in items and lambdas (similar as with Just-In-Time compiling)
- Spark + RDD = User, User, User, ….
- SQL + Schema = (Name, Age, Sex), (Name, Age, Sex), (Name, Age, Sex) ….

By the way: JIT Just-In-Time Compiler

- method inlining
- dead code elimination
- compilation to native code
- lock coarsening (grupowanie blokad)
- loop unrolling
- lock elision (eliminacja blokad)
- type sharpening

Spark interfaces

- Resilient Distributed Dataset (RDD)
  - Apache Spark's first abstraction
  - Interface to a sequence of data objects that consist of one or more types that are located across a collection of machines (a cluster)
  - "Lowest level" API available (provides lots of control)

- DataFrame
  - Conceptually equivalent to a table in a relational database or a DataFrame in R/Python
  - Can be constructed from structured data files, tables in Hive, external databases, or existing RDDs

- Dataset
  - Added type safety to DataFrames (now integrated)
  - Only in statically typed languages

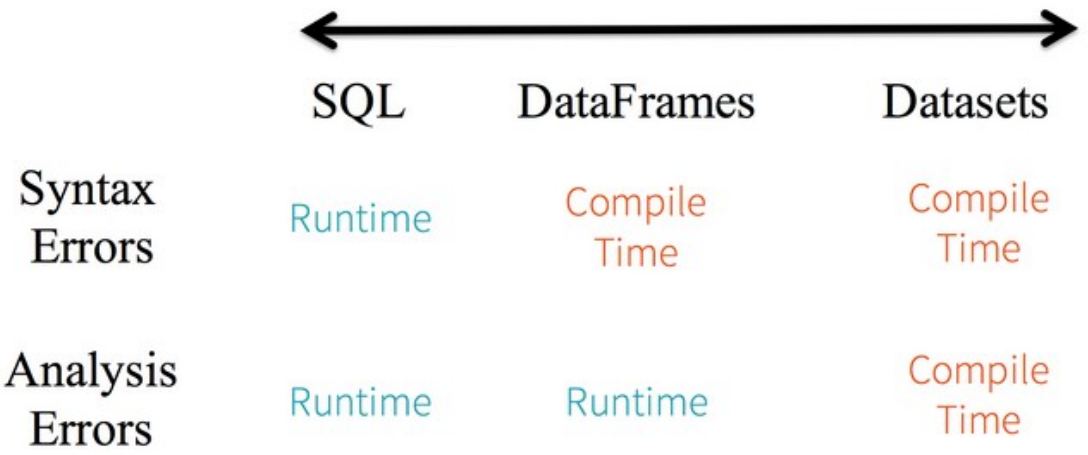
- DataFrames and Datasets contain optimizations (Tungsten+Catalyst)
  - It is good to understand RDDs as the underlying infrastructure that allows Spark to run so fast and provides data lineage

SQL 2003 as Spark interface

- Spark can run all of 99 TPC-DS queries (analytical queries benchmark)
- Standard compliant parser with good messages
- Subqueries (correlated and uncorrelated)
- Approximate aggregate stats

Spark interfaces

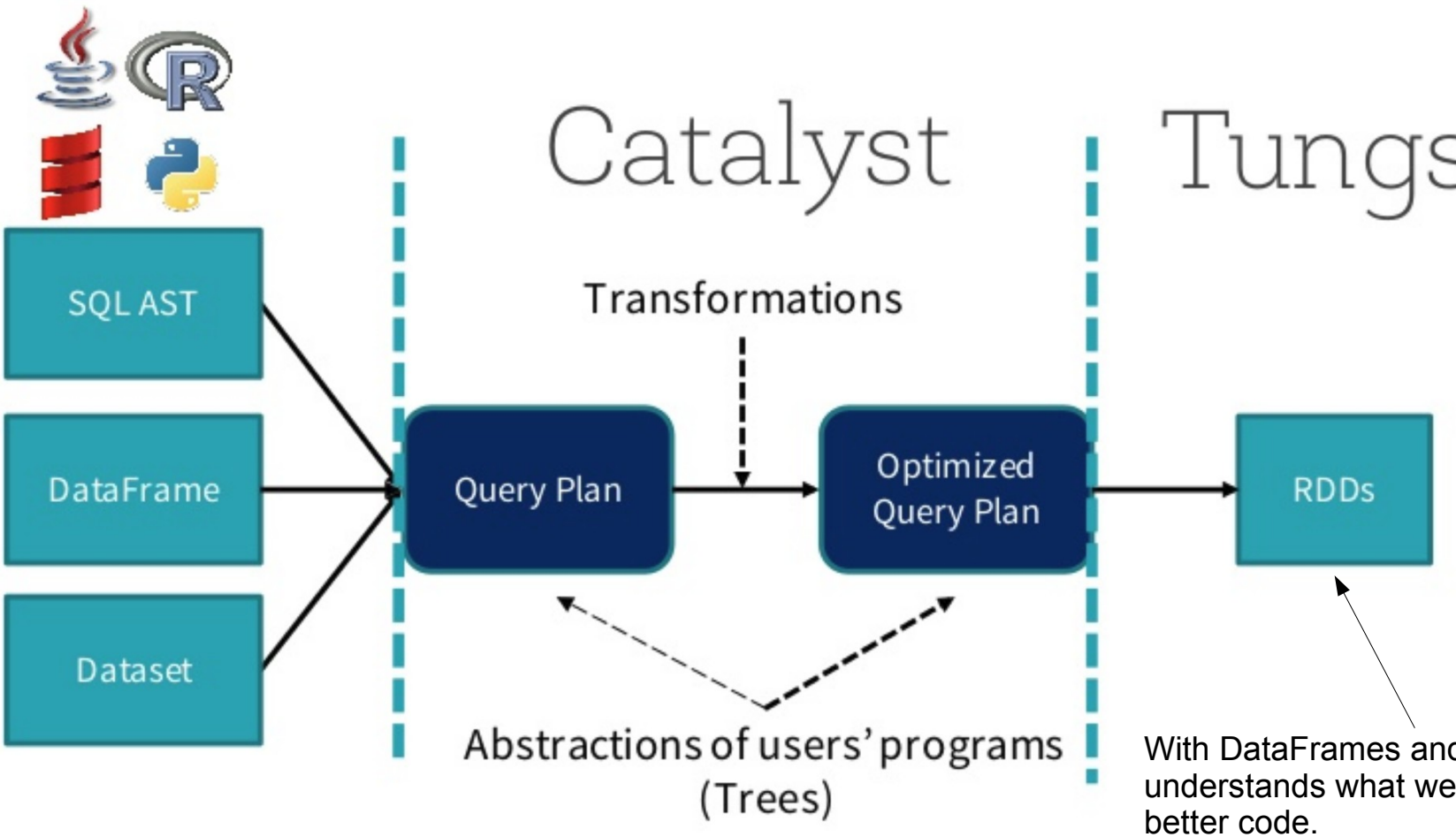
- Scala
  - Dataset[T]
  - DataFrame (alias for Dataset[Row])
- Java
  - Dataset[T]
- Python
  - DataFrame
- R
  - DataFrame



Catalyst and Tungsten

- Catalyst
  - query plans → optimized query plans
- Tungsten
  - generate bytecode realising query plans
    - possible for DataFrame/DataSet API (no black boxes)
  - technical but results in 10x optimisations

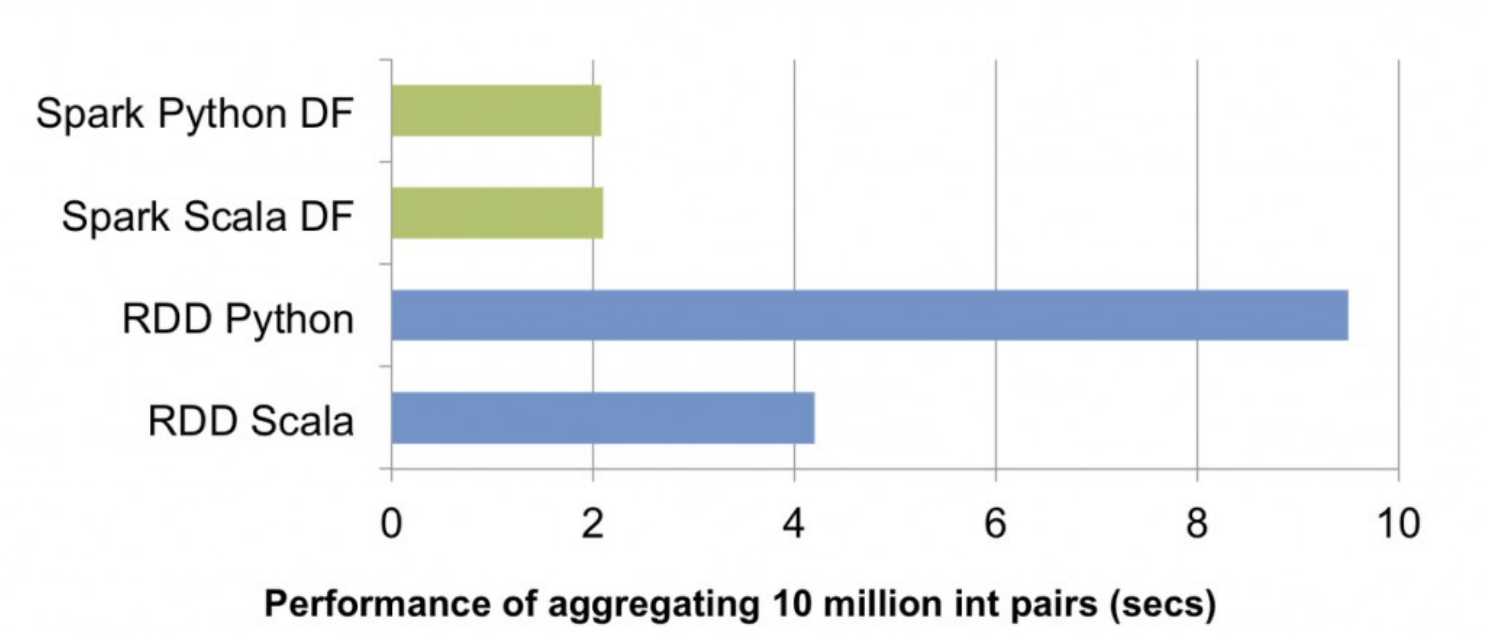
Catalyst and Tungsten



With DataFrames and Datasets Spark understands what we do and generates better code.

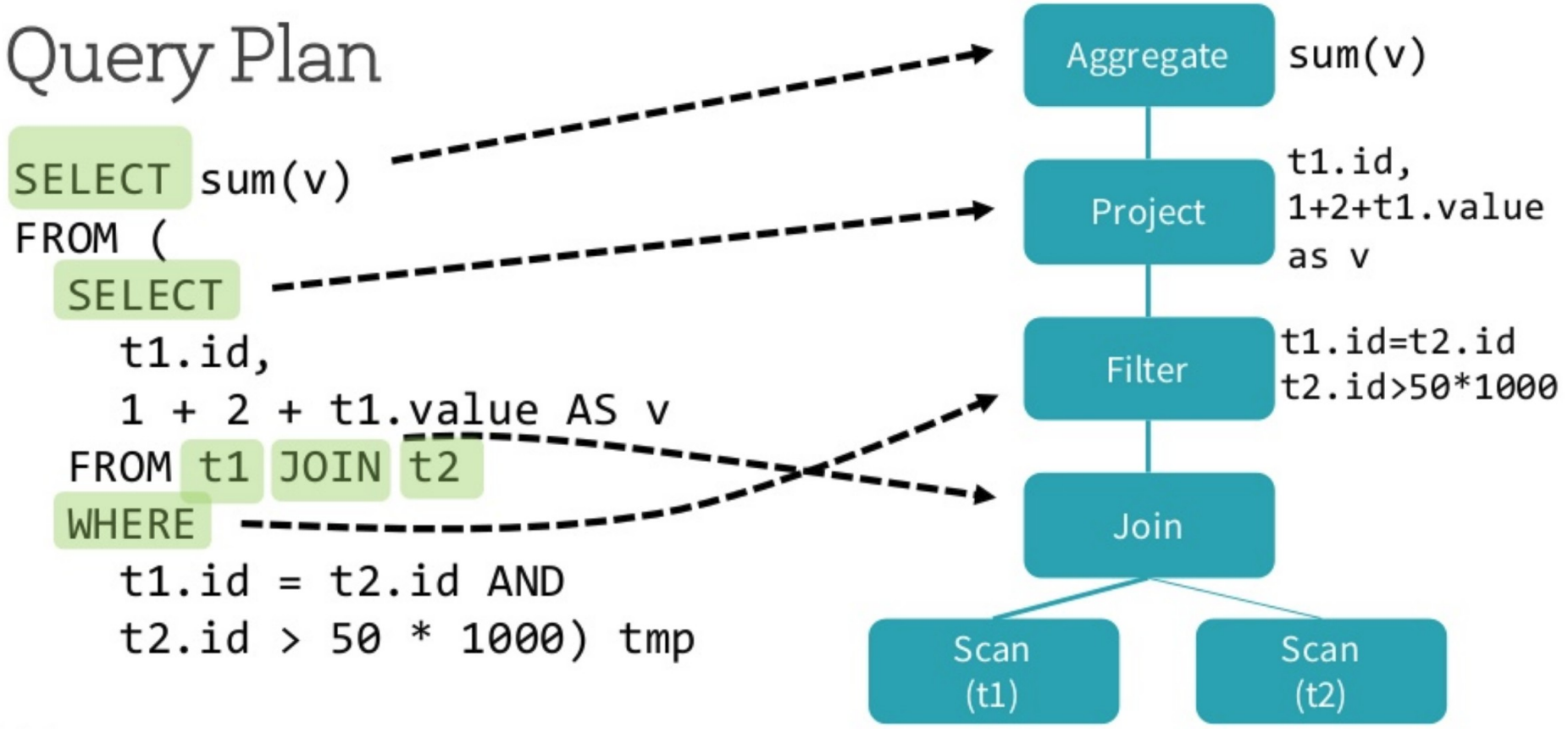
With RDD it is just some lambda!

Optimizations



Source: Databricks

Catalyst



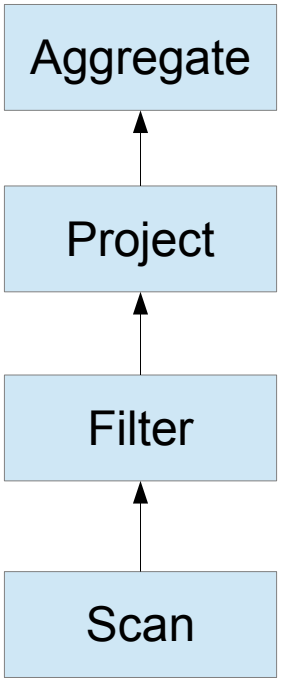
Source: Databricks

Mix of:

- Constant folding
- Column pruning
- Predicate pushdown

The Past: Volcano Iterator Model

```
select count(*) from sales
where item_sk = 1000
```

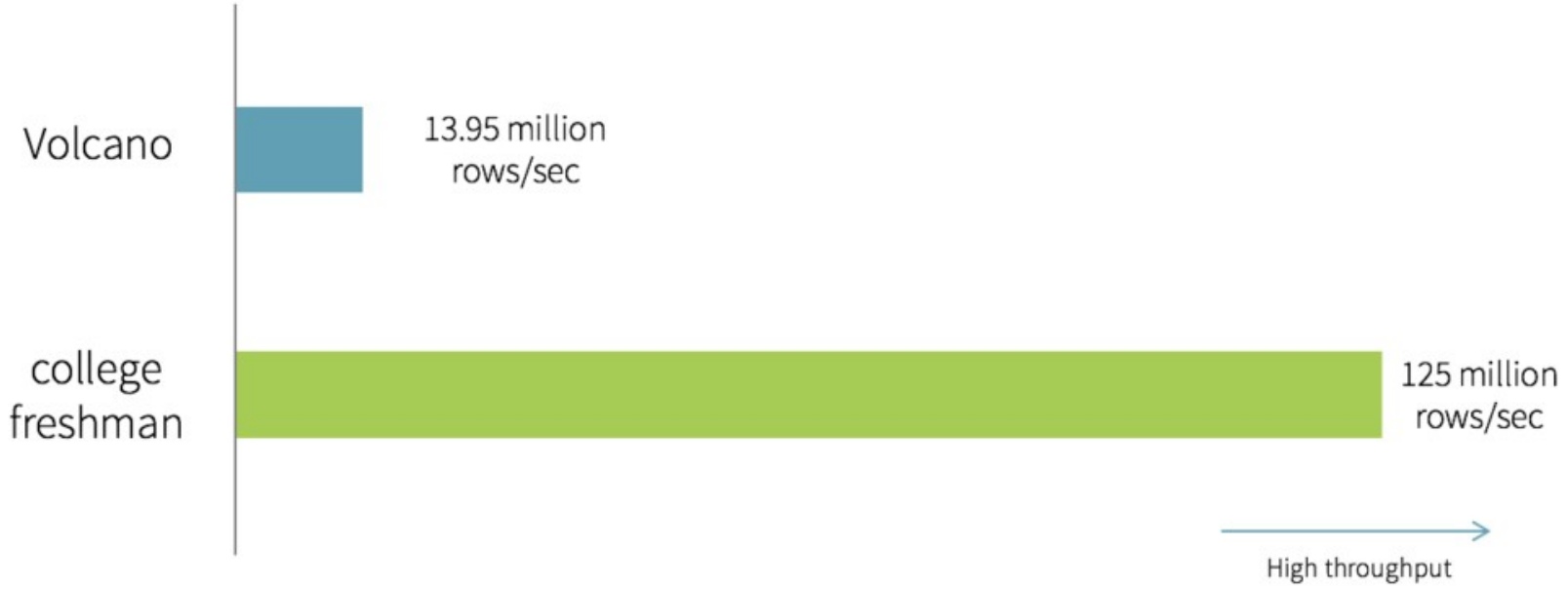



```
class Filter(child: Operator, predicate: (Row => Boolean)) extends Operator {
  def next(): Row = {
    var current = child.next()
    while (current != null && !predicate(current)) {
      current = child.next()
    }
    return current
  }
}
```

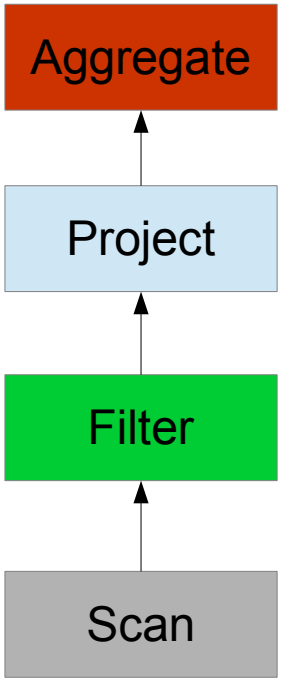
College freshman code

```
var count = 0

for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1
  }
}
```



Source: Databricks




```
long count = 0;
for (item in store_sales) {
    if (item.id == 1000) {
        count += 1;
    }
}
```

```
// * whole-stage code generation is enabled
spark.range(1000).filter("id > 100").selectExpr("sum(id)").explain()

== Physical Plan ==
*Aggregate(functions=[sum(id#201L)])
+- Exchange SinglePartition, None
   +- *Aggregate(functions=[sum(id#201L)])
      +- *Filter (id#201L > 100)
         +- *Range 0, 1, 3, 1000, [id#201L]
```

Reasons

- No virtual function dispatches
- Intermediate data in memory vs CPU registers
- Loop unrolling and SIMD

Idea is inspired by Thomas Neumann's seminal VLDB 2011 paper on:
Efficiently Compiling Efficient Query Plans for Modern Hardware

General rule: Each problem in CS can be solved by introducing an intermediary or if this an efficiency problem by removing an intermediary!

JVM objects and GC

- Eliminate overhead of JVM objects and GC becomes non-negligible

- “abcd” would take 4 bytes to store using UTF-8 encoding
- In Java
  - UTF-16 encoding
  - Object header
  - Array char[] with additional overhead
  - Hash code

- Most Spark operations operate directly against binary data rather than Java objects
  - C-style programming
  - Hash table that operates directly against binary data with memory explicitly managed by Spark. Compared with the standard Java HashMap, it has much less indirection overhead and is invisible to the garbage collector.

Project Tungsten

- Goals
  - **Memory Management and Binary Processing**: leveraging application semantics to manage memory explicitly and eliminate the overhead of JVM object model and **garbage collection**
  - **Cache-aware computation**: algorithms and data structures to exploit memory hierarchy
  - **Code generation**: using code generation to exploit modern compilers and CPUs
  - **No virtual function dispatches**: this reduces multiple CPU calls which can have a profound impact on performance when dispatching billions of times.
  - **Intermediate data in memory vs CPU registers**: Tungsten Phase 2 places intermediate data into CPU registers. This is an order of magnitudes reduction in the number of cycles to obtain data from the CPU registers instead of from memory
  - **Loop unrolling and SIMD**: Optimize Apache Spark's execution engine to take advantage of modern compilers and CPUs' ability to efficiently compile and execute simple for loops (as opposed to complex function call graphs).
- See
  - https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html
  - https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html
  - https://databricks.com/session/deep-dive-into-project-tungsten-bringing-spark-closer-to-bare-metal

Arrow and Parquet

Even small improvements matter in large scale

- On a big scale optimizing 0.5% of 3% use cases is a good job (Google)
- Disk and network I/O Hurt

Action	Computer time	Human scale time
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 µs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet SF to NYC	40 ms	4 years
Internet SF to UK	81 ms	8 years
Internet SF to Australia	183 ms	19 years

Based on data from: Systems Performance: Enterprise and the Cloud

Serialization

- Java Serialization, Kryo
- Google Protocol Buffers
  - Many languages
  - Limited schema evolution
- Apache Avro
  - Many languages
  - Schema evolution
  - Tuned for Hadoop (the same authors)
  - Optimized for Big data (compression, comparators)
- Apache Parquet (open standard in many open source projects)
  - All that +
  - Columnar
  - At least as fast, often faster (https://blog.cloudera.com/blog/2016/04/benchmarking-apache-parquet-the-allstate-experience/)
- Apache Arrow (open standard in many open source projects)
  - Columnar format for in memory applications

Advantages of a standard

- No need to convert from one representation to the other (cross-system communication)
  - Saves lots of CPU
- Projects can share functionality (e.g Parquet-to-Arrow reader)
  - projects: Spark, Pandas, Drill, Impala
  - serialization to: Parquet, Cassandra, Kudu, HBase

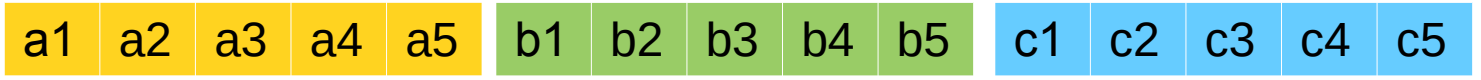
Table layout

Logical representation

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

Row layout

Column layout



Main benefits

- Can encode many values of the same type together
  - Smarter encodings
  - More efficient compression (more homogeneous values)
- Can access only the columns you need
  - VERY important for AI
  - Integrated with Catalyst
    - Some predicates can be evaluated without reading any data

Parquet in practice

- Less data = limited network transfer + limited disk usage + quicker queries = savings

Dataset	Size on Amazon S3	Query time	Scanned data	Cost
CSV files	1 TB	236 seconds	1.15 TB	$5.75
Apache Parquet format	130 GB	6.78 seconds	2.51 GB	$0.01
Savings/Speedup	87% less	34x faster	99% less	99.7% savings

(results from https://www.databricks.com/glossary/what-is-parquet)

Arrow vs Parquet differences

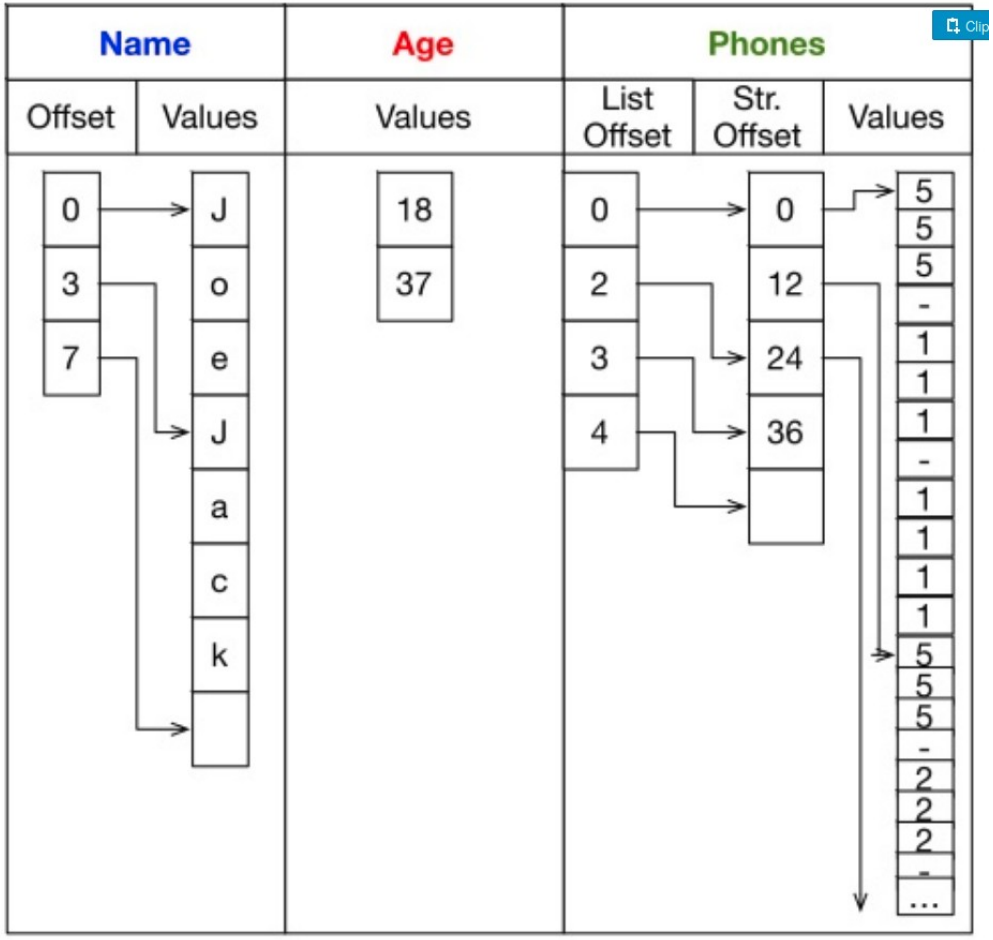
- On disc:
  - Write once, read many times
  - Priority to reduce I/O
  - Streaming access
  - Quick serialization, easy to store in HDFS chunks
- In memory:
  - Read once, use once
  - Priority to reduce CPU and increase cache locality
  - Streaming and random access
  - No serialization needed, can be sent over network as is

Arrow representation

(Delta-length byte array)

Columnar data

```
persons = [{
    name: 'Joe',
    age: 18,
    phones: [
        '555-111-1111',
        '555-222-2222'
    ]
}, {
    name: 'Jack',
    age: 37,
    phones: [ '555-333-3333' ]
}]
```



Parquet

- As easy to use and set up as files
- Fast simple database type queries
- Can store Big data in a cluster
  - Tuned to be stored in HDFS chunks

Types

- BOOLEAN: 1 bit boolean
- INT32: 32 bit signed ints
- INT64: 64 bit signed ints
- INT96: 96 bit signed ints
- FLOAT: IEEE 32-bit floating point values
- DOUBLE: IEEE 64-bit floating point values
- BYTE_ARRAY: arbitrarily long byte arrays.
- strings are stored as byte arrays (binary) with a UTF8 annotation

- delta-length byte array (Arrow idea)
- dictionary encoded (if few unique values used)
- bit packing (UTF idea)
- delta Strings (incremental encoding or front compression: for each element in a sequence of strings, store the prefix length of the previous entry plus the suffix)

Dremel model

- Very simple schema similar to Protocol buffers
- Minimalistic
  - represents nesting using groups of fields and
  - repetition using repeated fields
  - no need for any other complex types like Maps, List or Sets
- The root of the schema is a group of fields called a message
- Each field has three attributes: a repetition, a type and a name
  - type of a field is either a group or a primitive type (e.g., int, float, boolean, string) and the repetition can be one of the three following cases: required, optional, repeated
- Example for address book

```
message AddressBook {
  required string owner;
  repeated string ownerPhoneNumbers;
  repeated group contacts {
    required string name;
    optional string phoneNumber;
  }
}
```

File format

- **Block** (HDFS block): This means a block in HDFS and the meaning is unchanged for describing this file format. The file format is designed to work well on top of HDFS.
- **File**: A HDFS file that must include the metadata for the file. It does not need to actually contain the data.
- **Row group**: A logical horizontal partitioning of the data into rows. There is no physical structure that is guaranteed for a row group. A row group consists of a column chunk for each column in the dataset.
- **Column chunk**: A chunk of the data for a particular column. These live in a particular row group and is guaranteed to be contiguous in the file.
- **Page**: Column chunks are divided up into pages. A page is conceptually an indivisible unit (in terms of compression and encoding). There can be multiple page types which is interleaved in a column chunk.
- Hierarchically:
  - A file consists of one or more row groups.
  - A row group contains exactly one column chunk per column.
  - Column chunks contain one or more pages

- Row group: A group of rows in columnar format.
  - Max size buffered in memory while writing.
  - One (or more) per split while reading
  - Roughly: 50MB < row group < 1 GB
- Column chunk: The data for one column in a row group.
  - Column chunks can be read independently for efficient scans.
- Page: Unit of access in a column chunk.
  - Should be big enough for a compression to be efficient
  - Minimum size to read to access a single record 8 KB < page < 1MB