# What's in Apache Spark 2.0.0?

- Dataset evolution:
  - SparkSession (entry to Dataset/DataFrame - not SQLContext(sc))
  - Unify DataFrame and Dataset API (DF is now DS[Row])
- Off-heap caching: Overcome GC limits, compressed object pointers (compressed oops - up to 32GB Java heap - 32 bits/4 bytes, 64 bits/8 bytes if over)

# Project Tungsten - Closer to bare metal

- Memory management and binary processing
- Code generation: Don't create object, compare binary

# Tungsten Binary Format

- Spark 1.5
  - Java serialized object: JVM GC, 2 bytes UTF-16 encoding, header, hash code
  - C-style memory access - sun.misc.Unsafe
    - allocateMemory, freeMemory, getAddress
  - Spark manages memory

# Catalyst Optimizer

- represent query as tree/manipulate
- rule-based and cost-based optimization
- analysis, logical optimization, physical planning, and code generation to compile parts of queries to Java bytecode
- Literal(value: Int), Attribute(name: String)
- Add(left: TreeNode, right: TreeNode)
- Rule: for example tree.transform (add(lit1,lit2) = lit(1+2)
- Analysis: look up column names/types/tables from Catalog
- Logical Optimization: rule-based with constant folding, predicate pushdown, projection pruning, null propagation, Boolean expression simplification, and other rules
- Physical plans - cost model, code gen- expressions (+ predicate pushdown)

# Project Tungsten 2.0 - reduce CPU bottlenecks

- Virtual function calls
- Use CPU registers instead of cache/memory

# Simple aggregate query with filter

- count how many items have the sk 1000
- Note: all following graphs from Agarwal, Liu, and Xin (2016)

# Pre-2.0 Apache Spark: Volcano Iterator Model

- Graefe, 1994 paper "Volcano" - iterator, virtual function call
- "elegantly compose arbitrary combinations of operators"
- but virtual function calls (Operators)
- Heap memory for function calls

# Handwritten Code

- "explicit", customized, no combinations

# Handwritten vs. Volcano

- handwritten - much faster, not composable

# Whole-Stage Code Generation Benefits

- No virtual function dispatches
  - multiple CPU instructions - slower (over big data)
- Intermediate data in CPU registers
  - vs. function call stack in heap/memory
  - JVM JIT puts intermediate data into CPU registers
- Loop unrolling and SIMD
  - JIT compiler can unroll
  - pipelining, prefetching
  - Single instruction, multiple data (SIMD) - process multiple rows at one time
- Thomas Neumann's VLDB 2011 paper "Efficiently Compiling Efficient Query Plans for Modern Hardware."

# Whole-Stage Code Generation Example

- Have operators generate efficient code at runtime
- Before: single expression only (e.g. $a + 1$), now whole query plan

# See Whole-Stage Code Generation with explain()

- spark - new SparkSession entry point
- range(end) - 0 to end (exclusive) with id column

# Vectorization

- Use if unable to do whole-stage codegen
  - Complex code
  - Third party code
  - infeasible to generate code to fuse the entire query into a single function
- Each "next" call runs operator on batched column value
- Still in memory not registers
- New Vectorized Parquet reader

Agarwal, Sameer, Davies Liu, and Reynold S. Xin. 2016. "Apache Spark as a Compiler." `https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-seco` `html`.