

Apache Spark SQL Optimization Tungsten Optimizer

Surendra Panpaliya

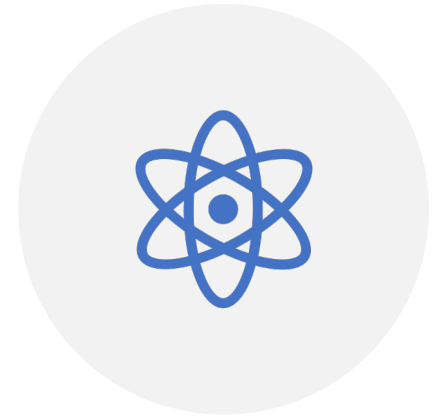
Agenda



TUNGSTEN OPTIMIZER



TUNGSTEN OVERVIEW



TUNGSTEN'S BINARY
FORMAT

Goal

The goal of Project Tungsten is

to improve Spark execution

by optimizing Spark jobs

for CPU and memory efficiency

(as opposed to network and disk I/O)

Scope

Tungsten focuses on

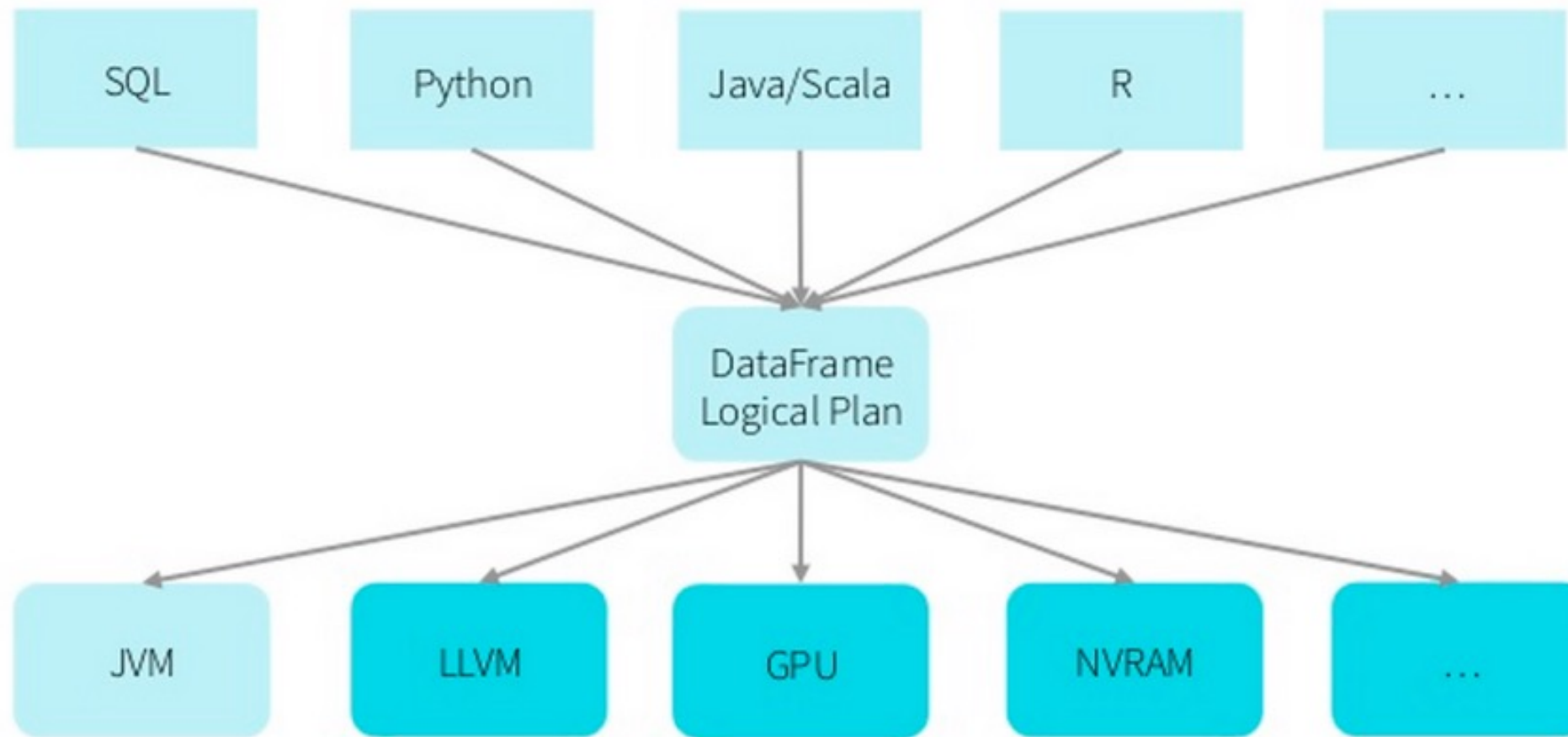
the hardware architecture

of the platform Spark runs on,

including but not limited

to JVM, LLVM, GPU, NVRAM, etc.

language
frontend



Tungsten
backend

Optimization Features



Off-Heap Memory Management



using binary in-memory data
representation



aka Tungsten row format and



managing memory explicitly

Optimization Features

Cache Locality which is about

cache-aware computations with

cache-aware layout for high cache hit rates,

Whole-Stage Code Generation
(aka *CodeGen*).

Design Improvements



Tungsten includes specialized



in-memory data structures



tuned for the type of operations required by Spark,



improved code generation, and



a specialized wire protocol.

Design Improvements

Tungsten's
representation is

substantially
smaller than
objects serialized

using Java or
even Kryo
serializers.

Design Improvements

As Tungsten does not depend on Java objects,
both on-heap and off-heap allocations are supported.

Not only is the format more compact,
serialization times can be substantially
faster than with native serialization.

Design Improvements



Tungsten no longer depends



on working with Java objects,



you can use either on-heap (in the JVM) or



off-heap storage.

Design Improvements



If you use off-heap storage,



it is important to leave enough room



in your containers for the off-heap allocations



- which you can get an approximate idea



for from the web ui.

Design Improvements



Tungsten's data structures



created closely in mind with



the kind of processing



for which they are used.

Design Improvements

The classic example

sorting, a common and expensive operation.

The on-wire representation is implemented

so that sorting can be done

without having

to deserialize the data again.

Design Improvements



Avoiding the memory and GC overhead



of regular Java objects,



Tungsten is able to process larger data sets



than the same hand-written aggregations.

Benefits



The following Spark jobs will benefit from Tungsten:



Dataframes: Java, Scala, Python, R



SparkSQL queries



Some RDD API programs via general serialization and



compression optimizations

Tungsten Execution Backend (Project Tungsten)

The goal of **Project Tungsten** is

to improve Spark execution

by optimizing Spark jobs

for **CPU and memory efficiency**

(as opposed to network and

disk I/O which are considered fast enough).

Tungsten Execution Backend (Project Tungsten)

Tungsten focuses

on the hardware architecture

of the platform Spark runs on,

including but not limited

to JVM, LLVM, GPU, NVRAM, etc.

Tungsten Execution Backend (Project Tungsten)

It does so by offering the following optimization features:

Off-Heap Memory Management

using binary in-memory data representation
aka

Tungsten row format and

managing memory explicitly,

Tungsten Execution Backend (Project Tungsten)

Cache Locality

which is about cache-aware computations

with cache-aware layout for

high cache hit rates,

Whole-Stage Code Generation

(*CodeGen*).

Tungsten Execution Backend (Project Tungsten)

Important

Project Tungsten uses `sun.misc.unsafe` API

for direct memory access

to bypass the JVM in order

to avoid garbage collection.

Tungsten Execution Backend

```
scala> val intsMM = 1 to math.pow(10, 6).toInt  
val intsMM: scala.collection.immutable.Range.Inclusive = Range 1 to 1000000
```

```
scala> sc.parallelize(intsMM).cache.count  
val res1: Long = 1000000
```

Tungsten Execution Backend

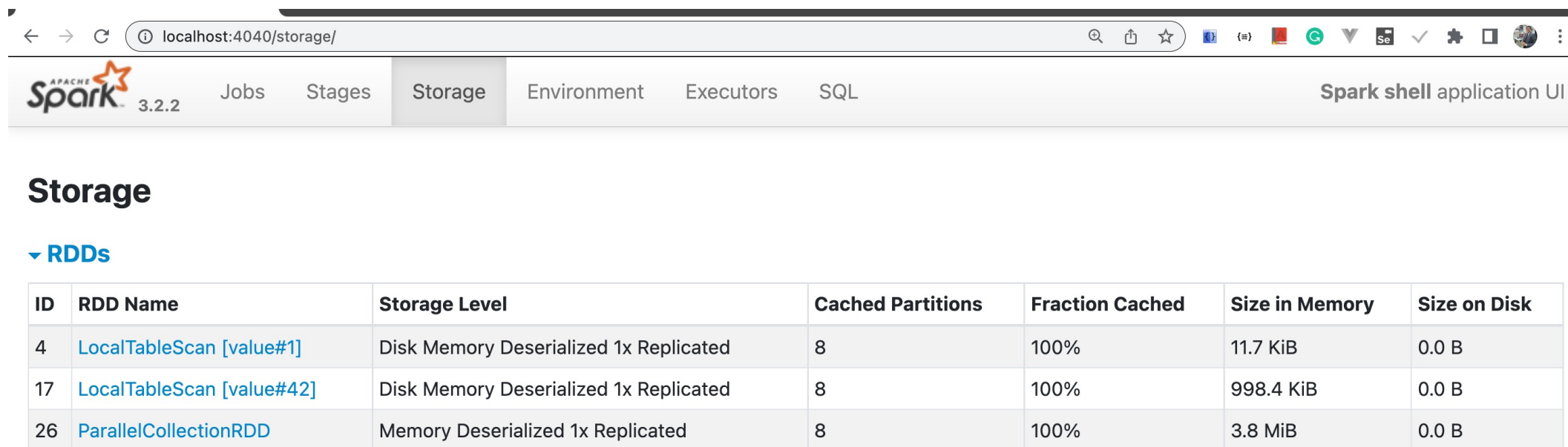
```
scala> intsMM.toDF.cache.count
```

```
22/08/09 23:26:13 WARN TaskSetManager: Stage 5 contains a task of very large  
size (4035 KiB). The maximum recommended task size is 1000 KiB.
```

```
val res2: Long =  
1000000
```

```
scala> sc.parallelize(intsMM).cache.count  
val res3: Long = 1000000
```

Tungsten Execution Backend (Project Tungsten)



The screenshot shows the Apache Spark web UI at localhost:4040/storage/. The 'Storage' tab is selected, showing a table of RDDs. The table has columns for ID, RDD Name, Storage Level, Cached Partitions, Fraction Cached, Size in Memory, and Size on Disk. Three RDDs are listed: ID 4 (LocalTableScan [value#1]), ID 17 (LocalTableScan [value#42]), and ID 26 (ParallelCollectionRDD). All three are stored in memory and have 8 cached partitions.

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
4	LocalTableScan [value#1]	Disk Memory Deserialized 1x Replicated	8	100%	11.7 KiB	0.0 B
17	LocalTableScan [value#42]	Disk Memory Deserialized 1x Replicated	8	100%	998.4 KiB	0.0 B
26	ParallelCollectionRDD	Memory Deserialized 1x Replicated	8	100%	3.8 MiB	0.0 B

RDD vs DataFrame Size in Memory in web UI—Thank you, Tungsten!

Off-Heap Memory Management

- Project Tungsten aims at substantially reducing the usage of JVM objects (and therefore JVM garbage collection) by introducing its own off-heap binary memory management. Instead of working with Java objects, Tungsten uses `sun.misc.Unsafe` to manipulate raw memory.

Off-Heap Memory Management

- Tungsten uses
- the compact storage format called
- [UnsafeRow](#) for data representation
- that further reduces memory footprint.

Off-Heap Memory Management

- Datasets have known [schema](#),
- Tungsten properly and in a
- more compact and efficient way lays out
- the objects on its own.

Off-Heap Memory Management

- That brings benefits similar
- to using extensions written in
- low-level and hardware-aware
- languages like C or assembler.

Cache Locality

- Tungsten uses algorithms and
- **cache-aware data structures** that
- exploit the physical machine caches at
- different levels - L1, L2, L3.

Whole-Stage Java Code Generation

- Tungsten does code generation at compile time and
- generates JVM bytecode to access
- Tungsten-managed memory structures
- that gives a very fast access.
- It uses the [Janino compiler](#) — a super-small,
- super-fast Java compiler.

Whole-Stage Java Code Generation

- *CodeGen* is a physical query optimization
- in Spark SQL that
- fuses multiple physical operators
- together into a single Java function.

Whole-Stage Java Code Generation

- Whole-Stage Java Code Generation
- improves the execution performance of
- a query by collapsing a query tree
- into a single optimized function
- that eliminates virtual function calls and
- leverages CPU registers
- for intermediate data.

Whole-Stage Java Code Generation

- Whole-Stage Code Generation is controlled by [spark.sql.codegen.wholeStage](#)
- Spark internal property.
- Whole-Stage Code Generation
- is enabled by default.
- `import org.apache.spark.sql.internal.SQLConf.WHOLESTAGE_CODEGEN_ENABLED`
- `scala> spark.conf.get(WHOLESTAGE_CODEGEN_ENABLED)`
- `res0: String = true`

Whole-Stage Java Code Generation

- Use [SQLConf.wholeStageEnabled](#) method to access the current value.
- scala> spark.sessionState.conf.wholeStageEnabled
- res1: Boolean = true

Whole-Stage Java Code Generation

- Use [SQLConf.wholeStageEnabled](#) method to access the current value.
- scala> spark.sessionState.conf.wholeStageEnabled
- res1: Boolean = true