Deep Dive into Project Tungsten: Bringing Spark Closer to Bare Metal

Josh Rosen (@jshrsn)

June 16, 2015

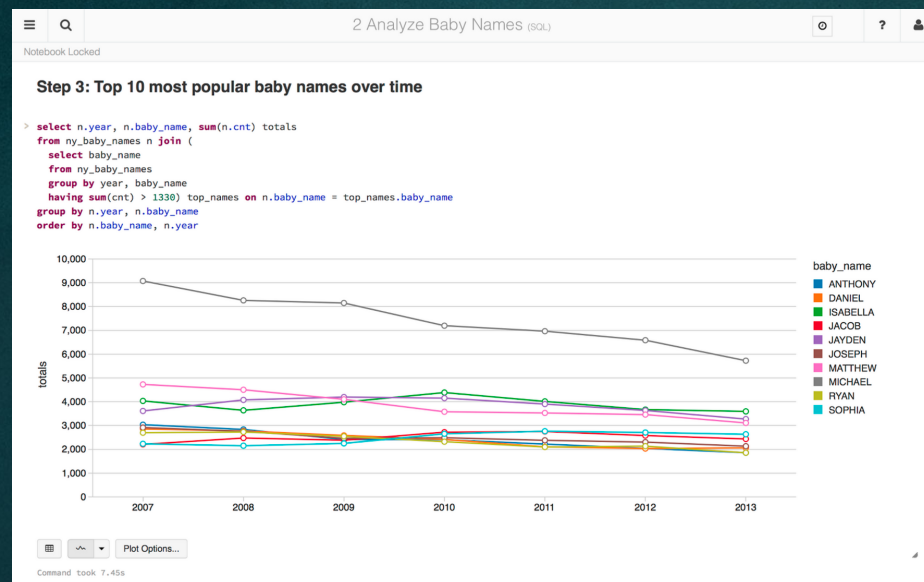
databricks™

About Databricks

Founded by creators of Spark and remains largest contributor

Offers a hosted service:

- Spark on EC2
- Notebooks
- Plot visualizations
- Cluster management
- Scheduled jobs



Goals of Project Tungsten

Substantially improve the **memory and CPU** efficiency of Spark applications .

Push performance closer to the limits of modern hardware.

In this talk

- Motivation: why we're focusing on compute instead of IO
- How Tungsten optimizes memory + CPU
- Case study: aggregation
- Case study: record sorting
- Performance results
- Roadmap + next steps

Many big data workloads are now compute bound

NSDI'15:

Making Sense of Performance in Data Analytics Frameworks

Kay Ousterhout[*], Ryan Rasti[*†◇], Sylvia Ratnasamy[*], Scott Shenker[*†], Byung-Gon Chun[‡]
[*]UC Berkeley, [†]ICSI, [◇]VMware, [‡]Seoul National University

- "Network optimizations can only reduce job completion time by a median of at most 2%."
- "Optimizing or eliminating disk accesses can only reduce job completion time by a median of at most 19%."
- We've observed similar characteristics in many Databricks Cloud customer workloads.

Why is CPU the new bottleneck?

- Hardware has improved:
  - Increasingly large aggregate IO bandwidth, such as 10Gbps links in networks
  - High bandwidth SSD's or striped HDD arrays for storage
- Spark's IO has been optimized:
  - many workloads now avoid significant disk IO by pruning input data that is not needed in a given job
  - new shuffle and network layer implementations
- Data formats have improved:
  - Parquet, binary data formats
- Serialization and hashing are CPU-bound bottlenecks

How Tungsten improves CPU & memory efficiency

- **Memory Management and Binary Processing**: leverage 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**: exploit modern compilers and CPUs; allow efficient operation directly on binary data

Generality has a cost, so we should use semantics and schema to exploit specificity instead

The overheads of Java objects

“abcd”

- **Native:** 4 bytes with UTF-8 encoding
- **Java:** 48 bytes

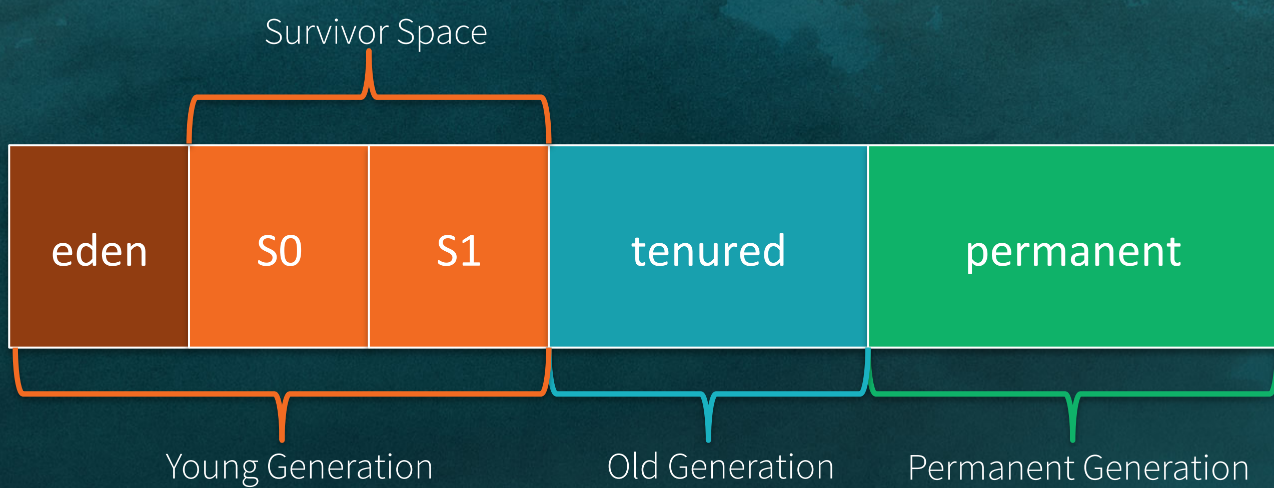
```
java.lang.String object internals:
OFFSET  SIZE   TYPE DESCRIPTION                    VALUE
     0     4        (object header)                ...
     4     4        (object header)                ...
     8     4        (object header)                ...
    12     4 char[] String.value                   []
    16     4    int String.hash                    0
    20     4    int String.hash32                  0
Instance size: 24 bytes (reported by Instrumentation API)
```

12 byte object header

20 bytes of overhead + 8 bytes for chars

8 byte hashcode

Garbage collection challenges



- Many big data workloads create objects in ways that are unfriendly to regular Java GC.
- Guest blog on GC tuning: tinyurl.com/db-gc-tuning

sun.misc.Unsafe

- JVM internal API for directly manipulating memory without safety checks (hence “unsafe”)
- We use this API to build data structures in both on- and off-heap memory

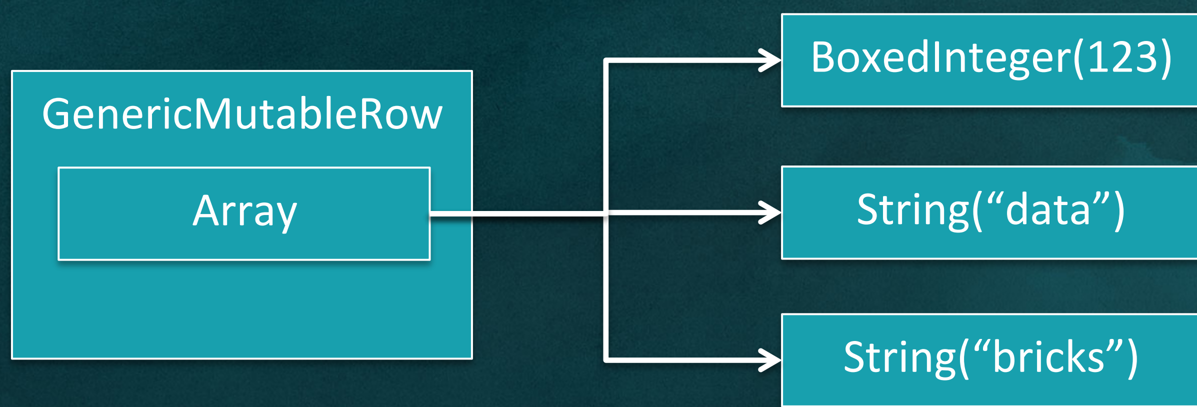
Flat data structures

Data structures with pointers

Complex examples

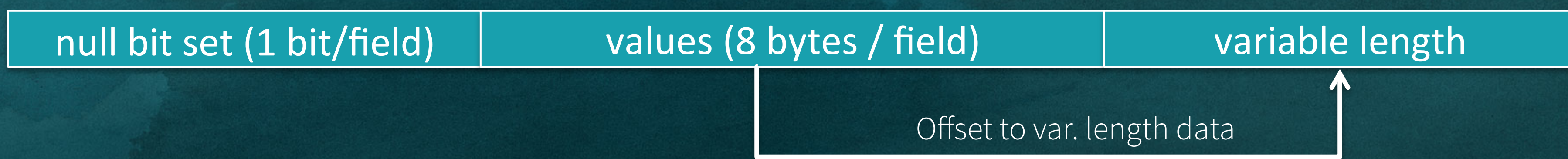
Java object-based row representation

3 fields of type (int, string, string)
with value (123, “data”, “bricks”)



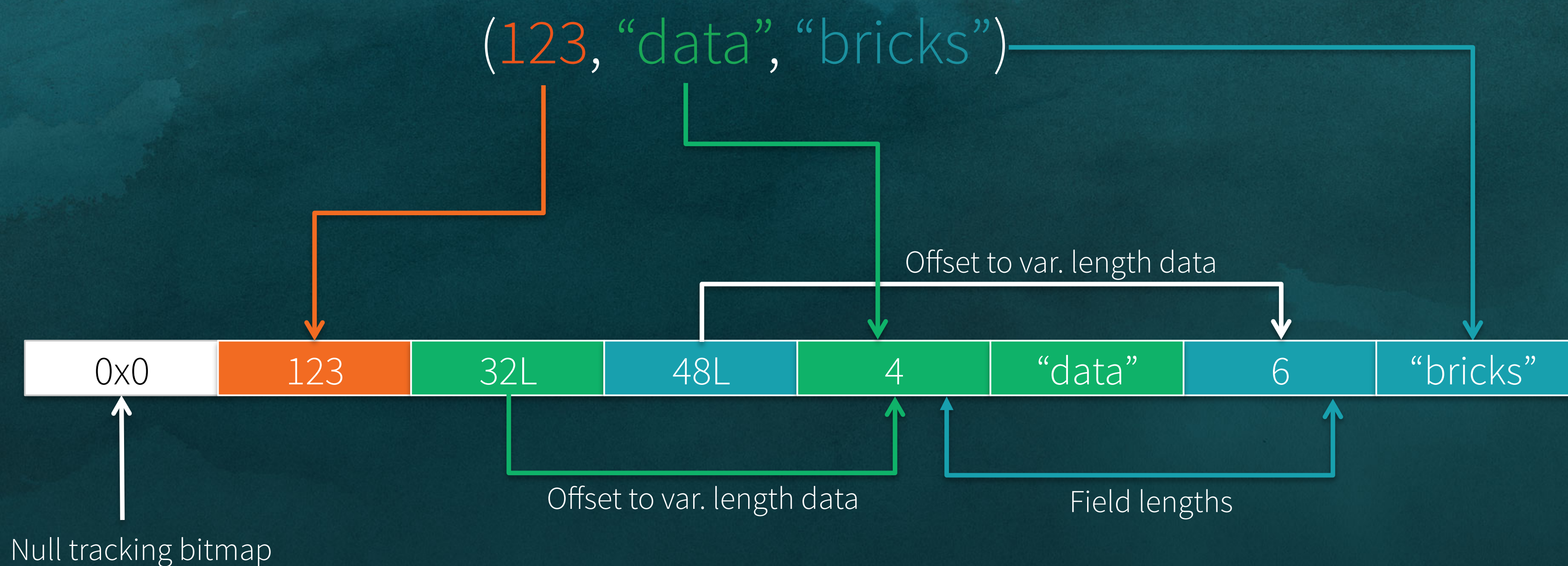
5+ objects; high space overhead; expensive hashCode()

Tungsten's UnsafeRow format



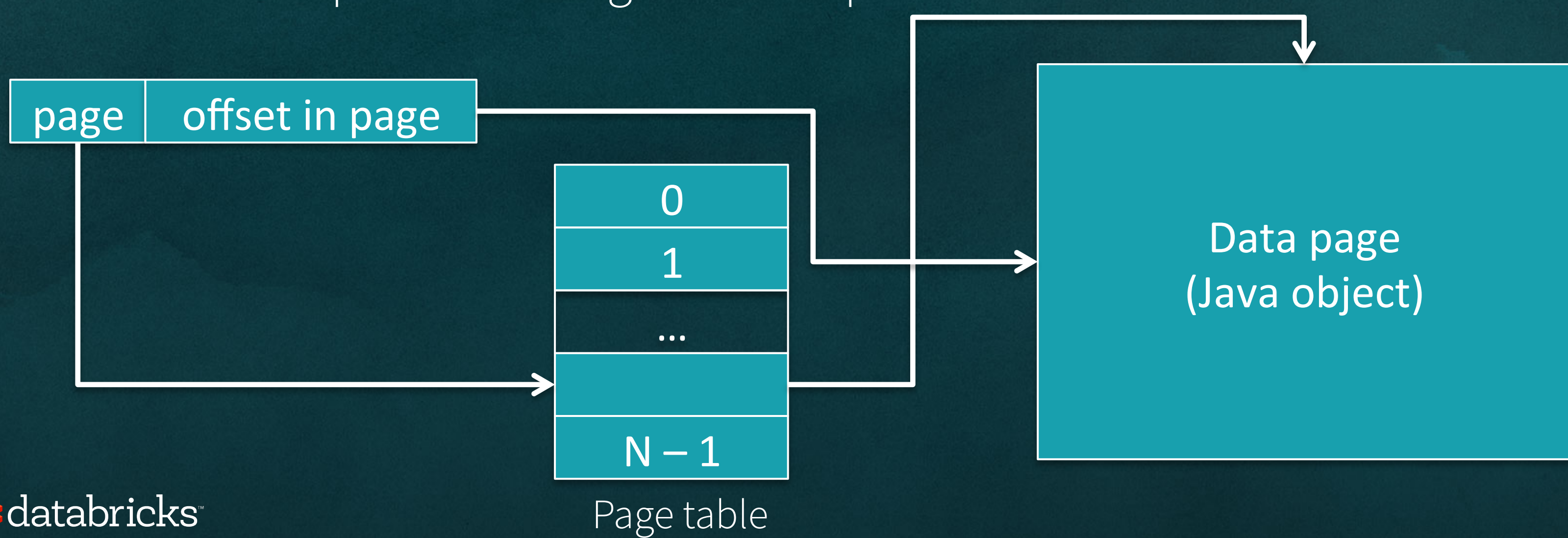
- Bit set for tracking null values
- Every column appears in the fixed-length values region:
  - Small values are inlined
  - For variable-length values (strings), we store a relative offset into the variable-length data section
- Rows are always 8-byte word aligned (size is multiple of 8 bytes)
- Equality comparison and hashing can be performed on raw bytes without requiring additional interpretation

Example of an UnsafeRow

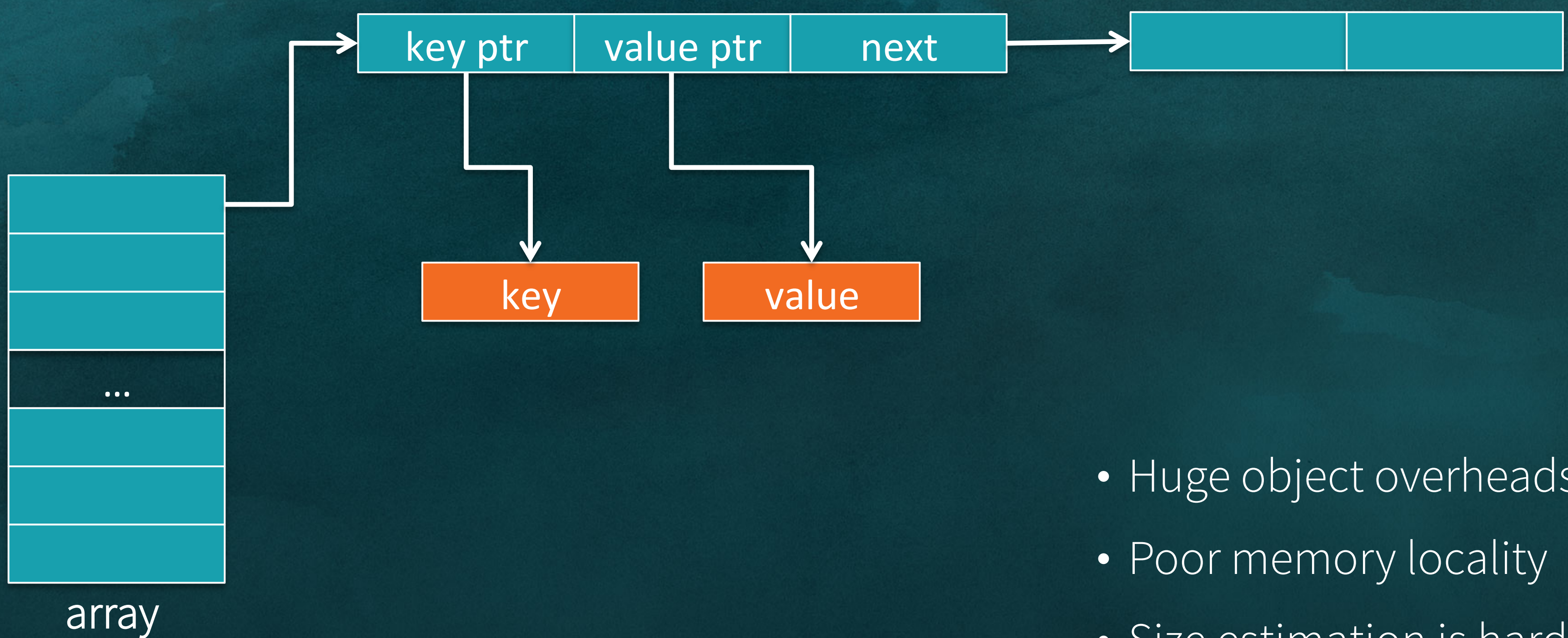


How we encode memory addresses

- Off heap: addresses are raw memory pointers.
- On heap: addresses are base object + offset pairs.
- We use our own "page table" abstraction to enable more compact encoding of on-heap addresses:



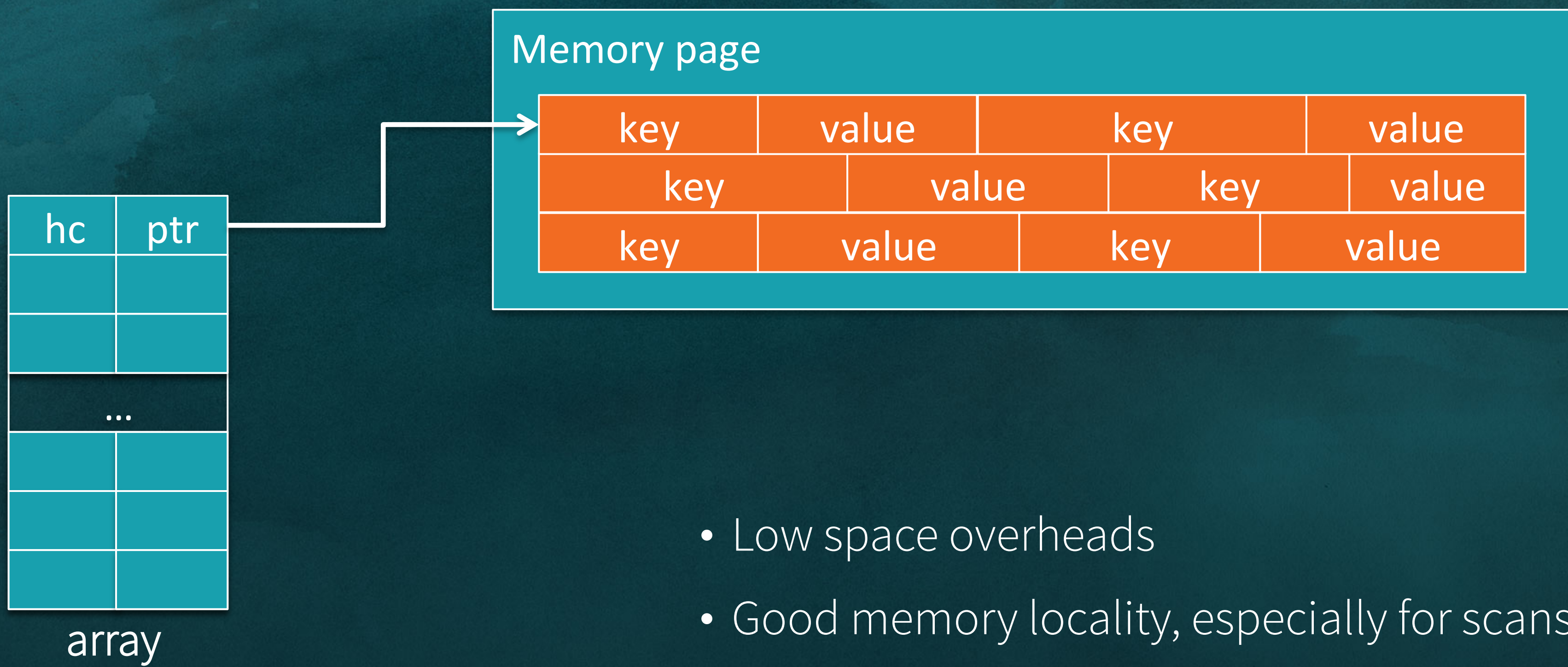
java.util.HashMap



- Huge object overheads
- Poor memory locality
- Size estimation is hard

databricks

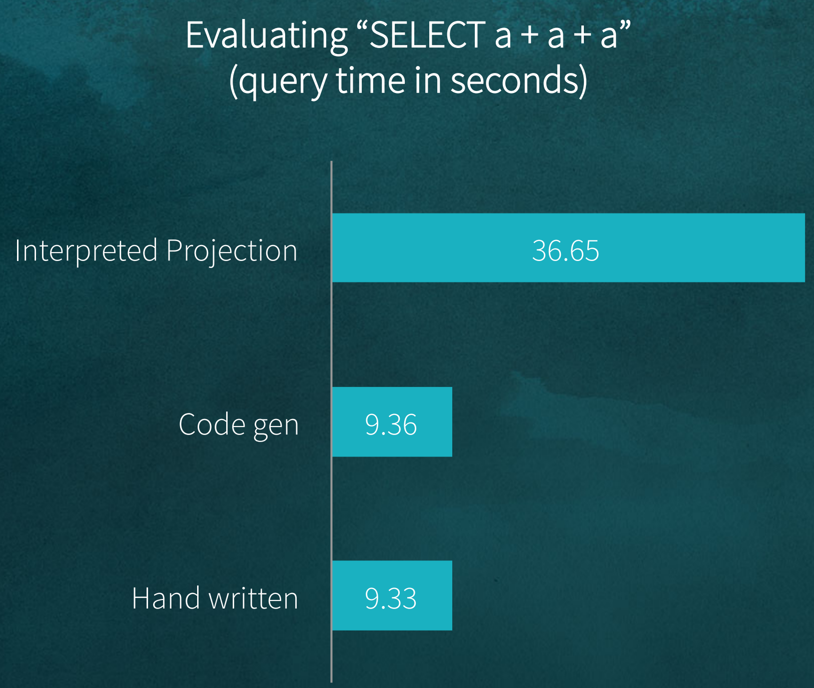
Tungsten’s BytesToBytesMap



- Low space overheads
- Good memory locality, especially for scans

Code generation

- Generic evaluation of expression logic is very expensive on the JVM
  - Virtual function calls
  - Branches based on expression type
  - Object creation due to primitive boxing
  - Memory consumption by boxed primitive objects
- Generating custom bytecode can eliminate these overheads



databricks

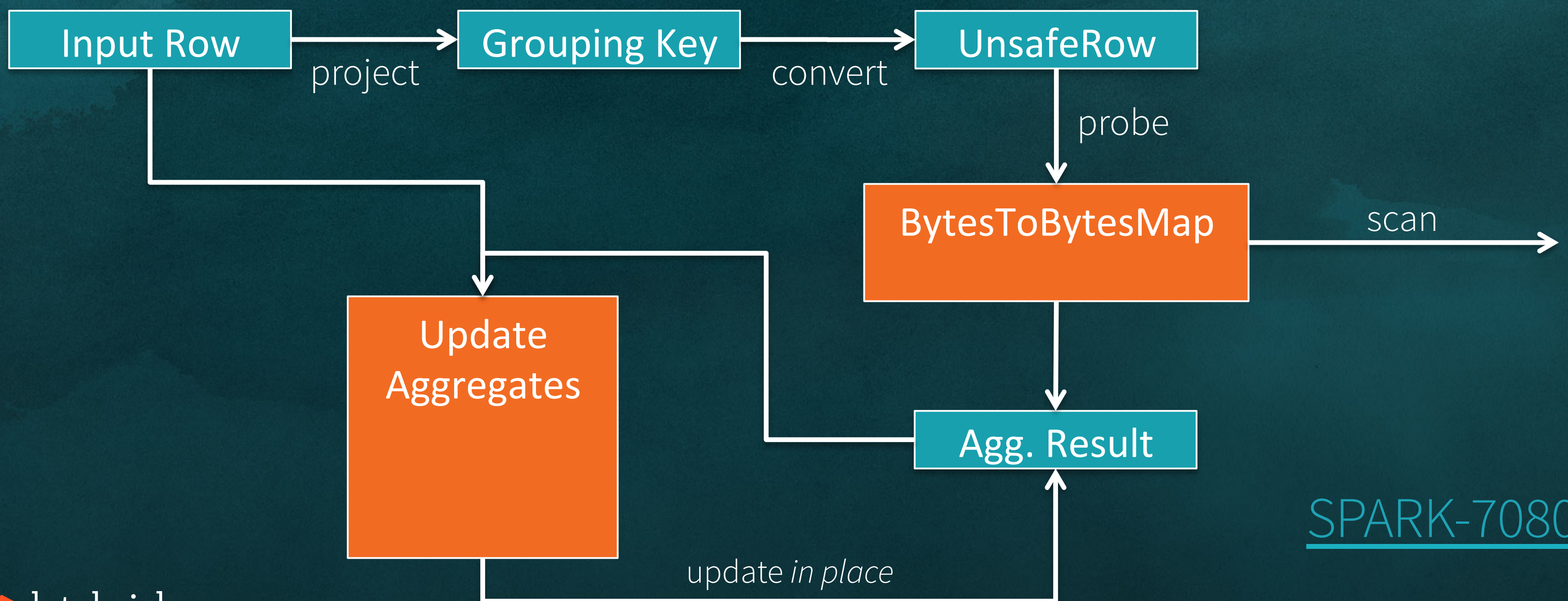
Code generation

- Project Tungsten uses the Janino compiler to reduce code generation time.
- Spark 1.5 will greatly expand the number of expressions that support code generation:
  - SPARK-8159

Example: aggregation optimizations in DataFrames and Spark SQL

```
df.groupBy("department").agg(max("age"), sum("expense"))
```

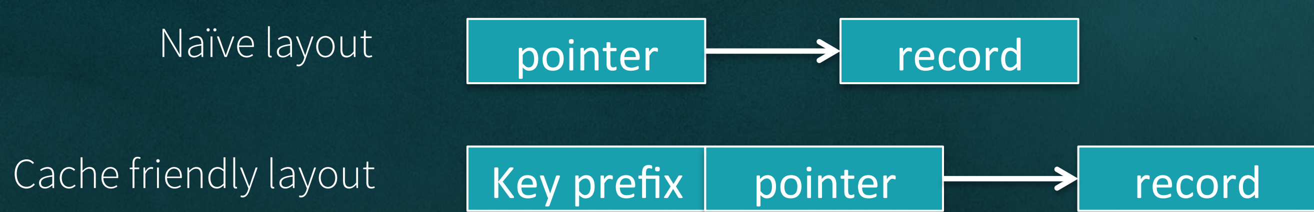
Example: aggregation optimizations in DataFrames and Spark SQL



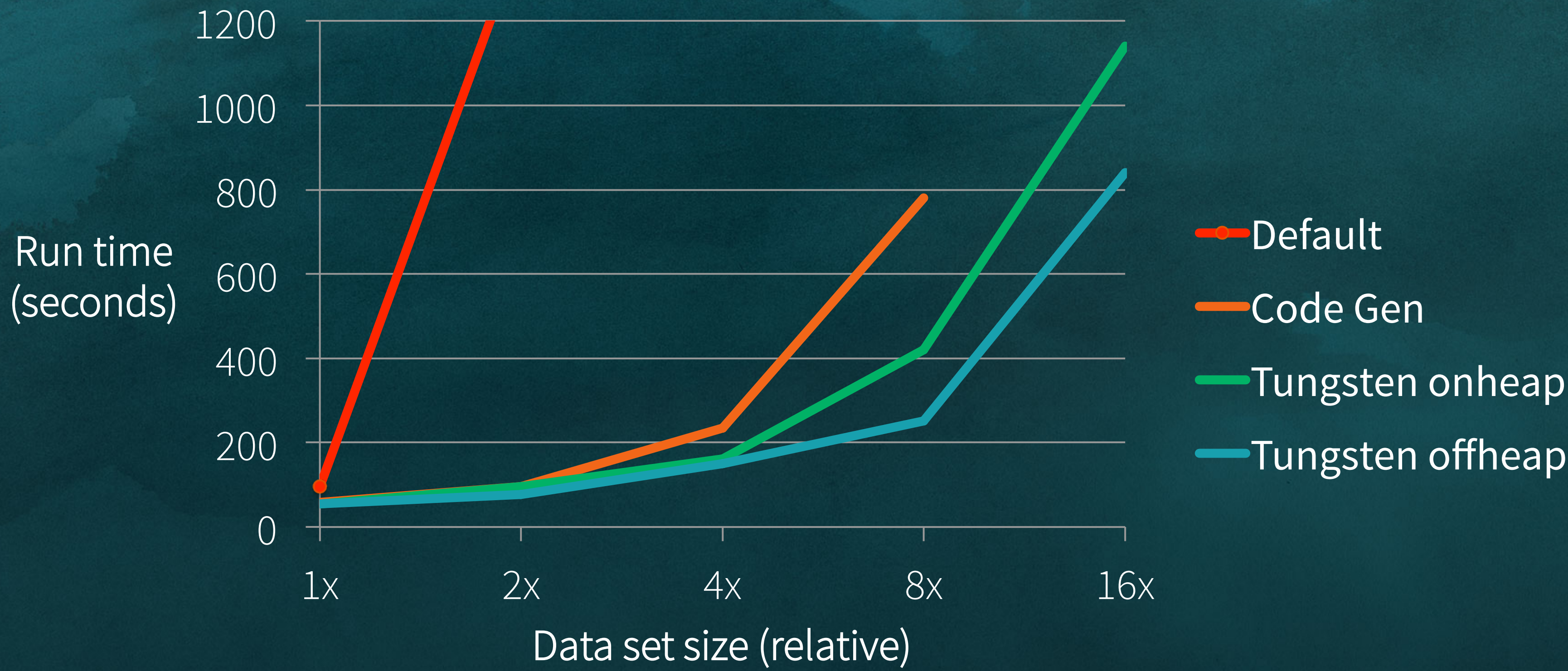
SPARK-7080

Optimized record sorting in Spark SQL + DataFrames (SPARK-7082)

- AlphaSort-style prefix sort:
  - Store prefixes of sort keys inside the sort pointer array
  - During sort, compare prefixes to short-circuit and avoid full record comparisons
- Use this to build external sort-merge join to support joins larger than memory

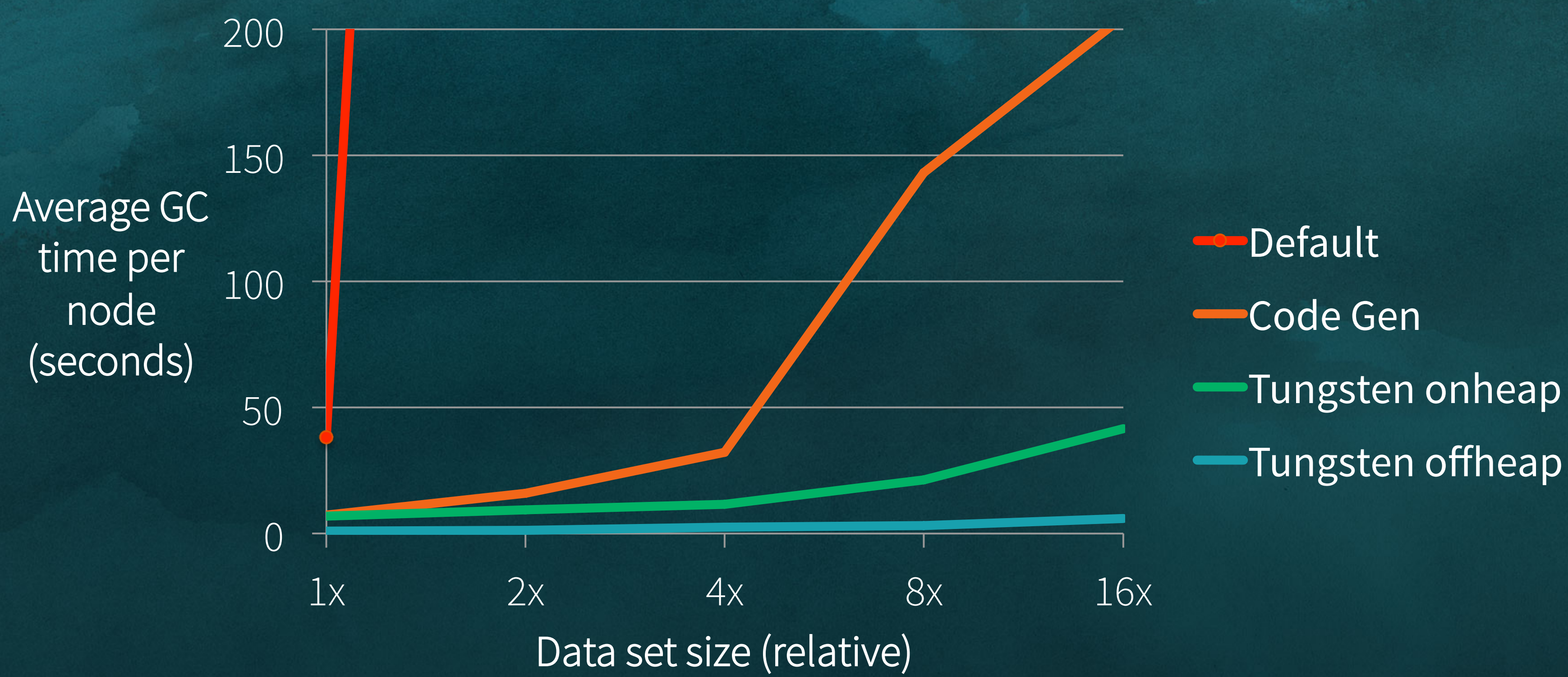


Initial performance results for agg. query



databricks

Initial performance results for agg. query



databricks

Project Tungsten Roadmap

Spark 1.4	Spark 1.5	Spark 1.6
• Binary processing for aggregation in Spark SQL / DataFrames • New Tungsten shuffle manager • Compression & serialization optimizations	• Optimized code generation • Optimized sorting in Spark SQL / DataFrames • End-to-end processing using binary data representations • External aggregation	• Vectorized / batched processing • ???

databricks

Which Spark jobs can benefit from Tungsten?

- DataFrames
  - Java
  - Scala
  - Python
  - R
- Spark SQL queries
- **Some Spark RDD API programs**, via general serialization + compression optimizations

```
logs.join(
  users,
  logs.userId == users.userId,
  "left_outer") \
.groupBy("userId").agg({"*": "count"})
```

databricks

How to enable all of Spark 1.4's Tungsten optimizations

Warning! These features are experimental in 1.4!

```
spark.sql.codegen = true
spark.sql.unsafe.enabled = true
spark.shuffle.manager = tungsten-sort
```

databricks

Thank you.

Follow our progress on JIRA: [SPARK-7075]

databricks™