A Developer's View Into Spark's Memory Model

Wenchen Fan

2017-6-7

databricks

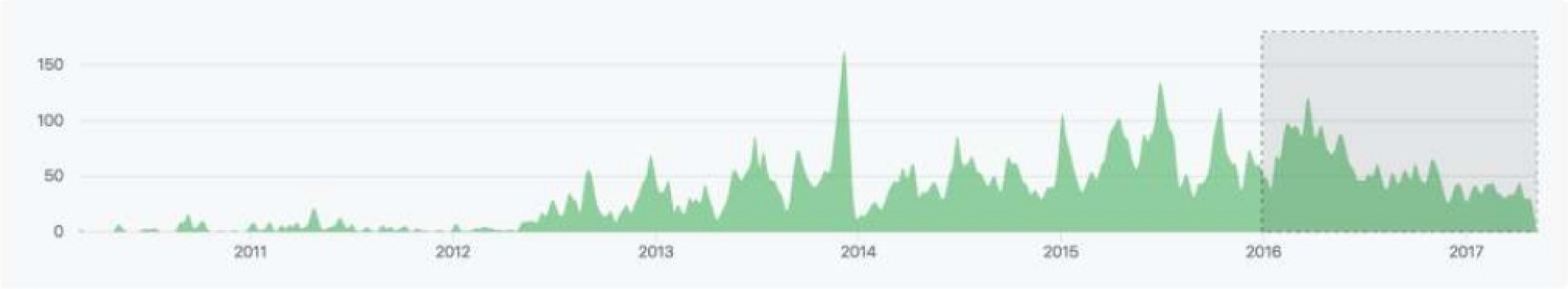
About Me

- Software Engineer @databricks
- Apache Spark Committer
- One of the most active Spark contributors

Jan 27, 2016 – Jun 6, 2017

Contributions to master, excluding merge commits

Contributions: **Commits** ▾



rxin #1

267 commits / 74,167 ++ / 83,132 --

cloud-fan #2

238 commits / 2[illegible],939 ++ / 19,586 --

About Databricks

TEAM

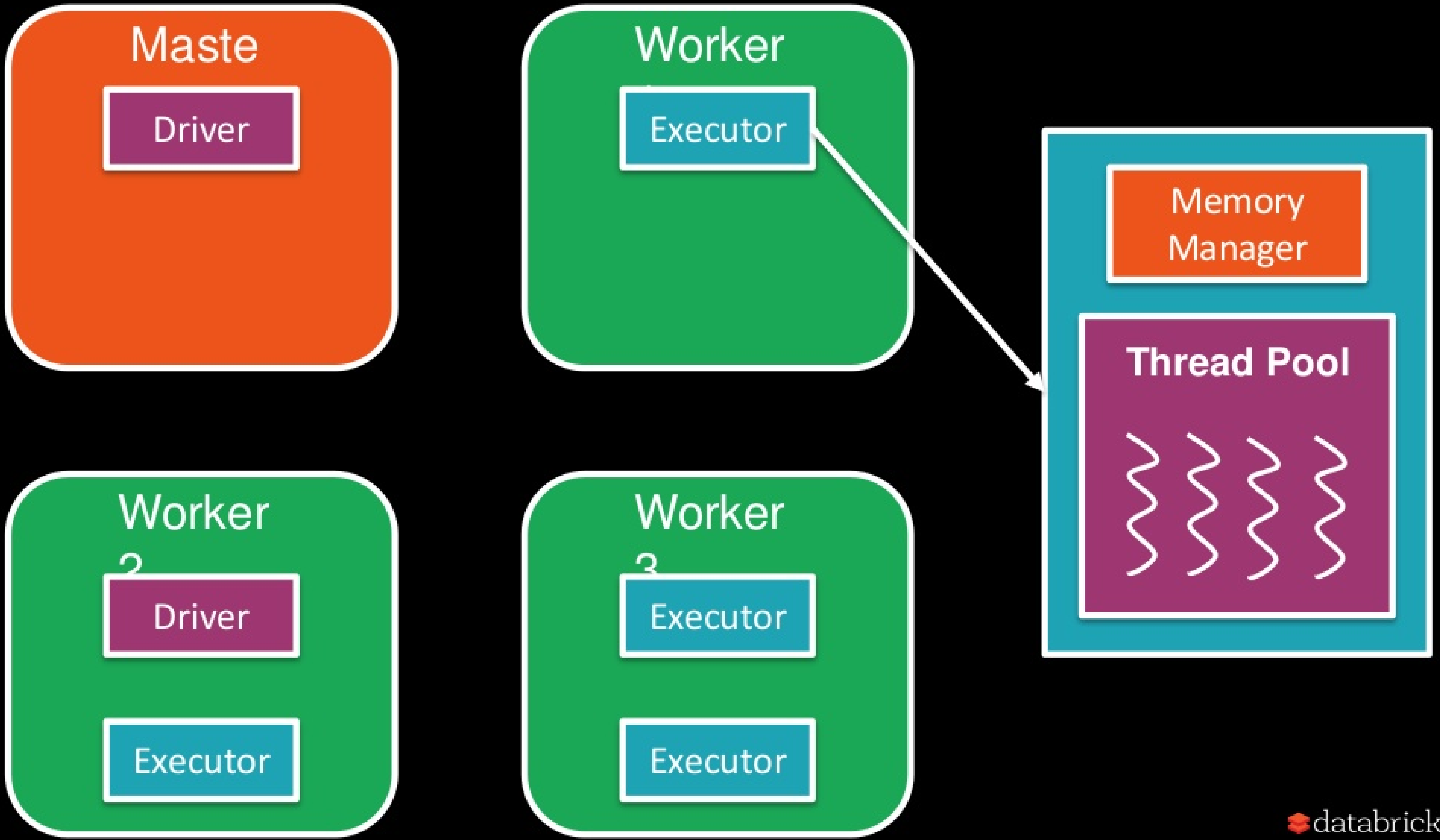
Started Spark project (now Apache Spark) at UC Berkeley in 2009

MISSON

Make Big Data Simple

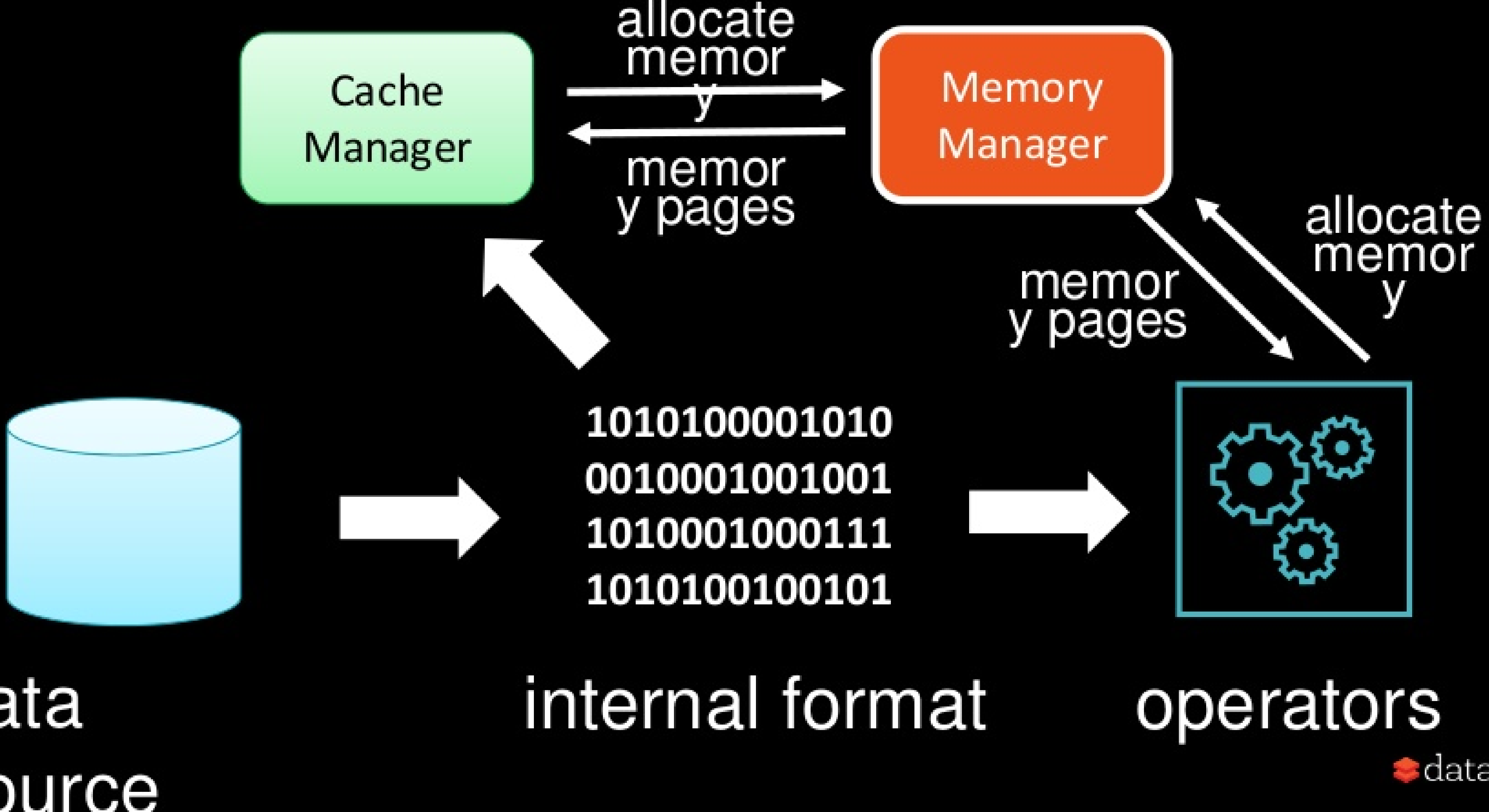
PRODUC
T

Unified Analytics Platform

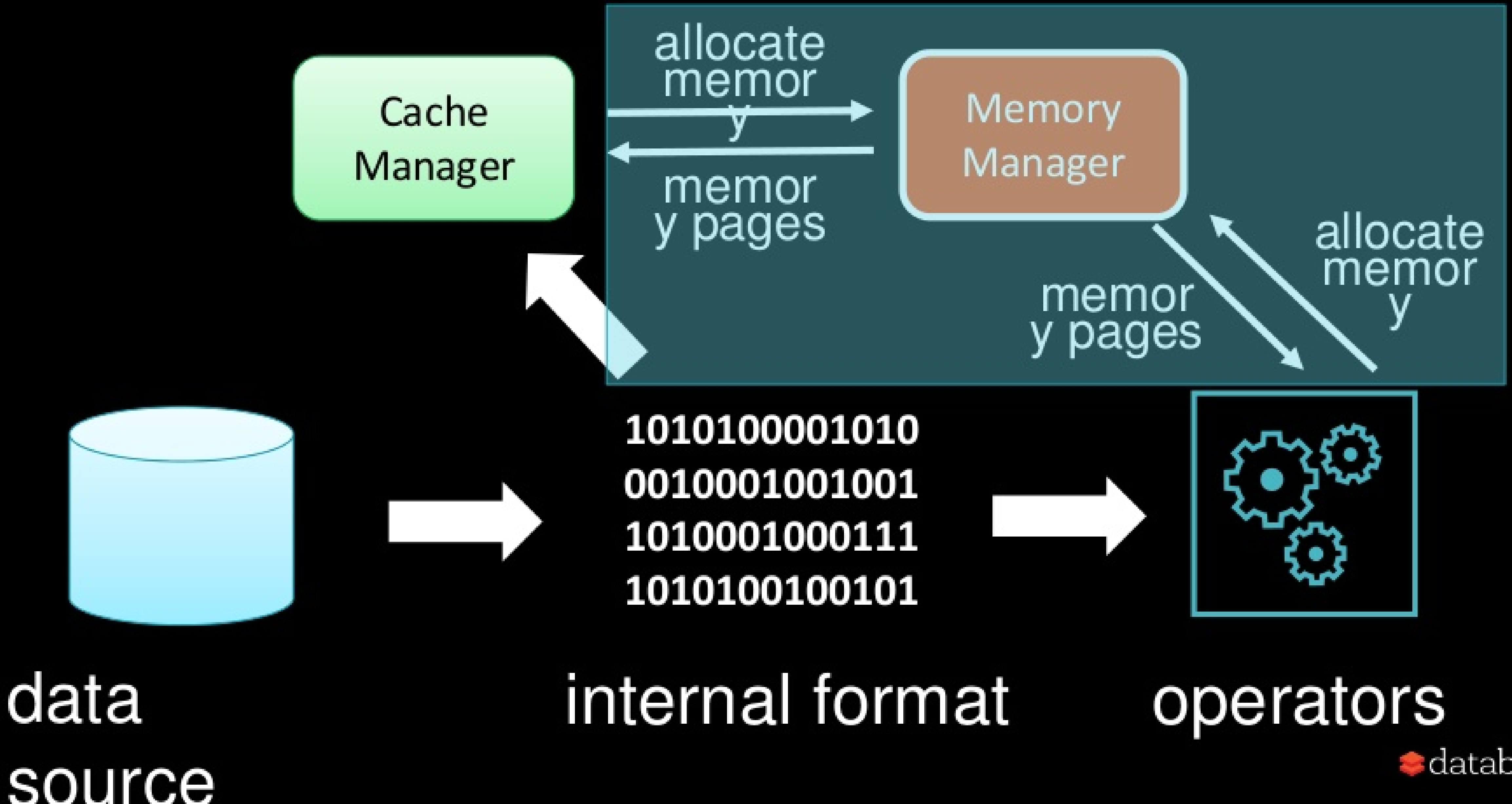


Maste
Driver
Worker
Executor
Memory Manager
Thread Pool
Worker
2
Driver
Executor
Worker
3
Executor
Executor
databricks

Memory Model inside Executor



Memory Model inside Executor



databricks

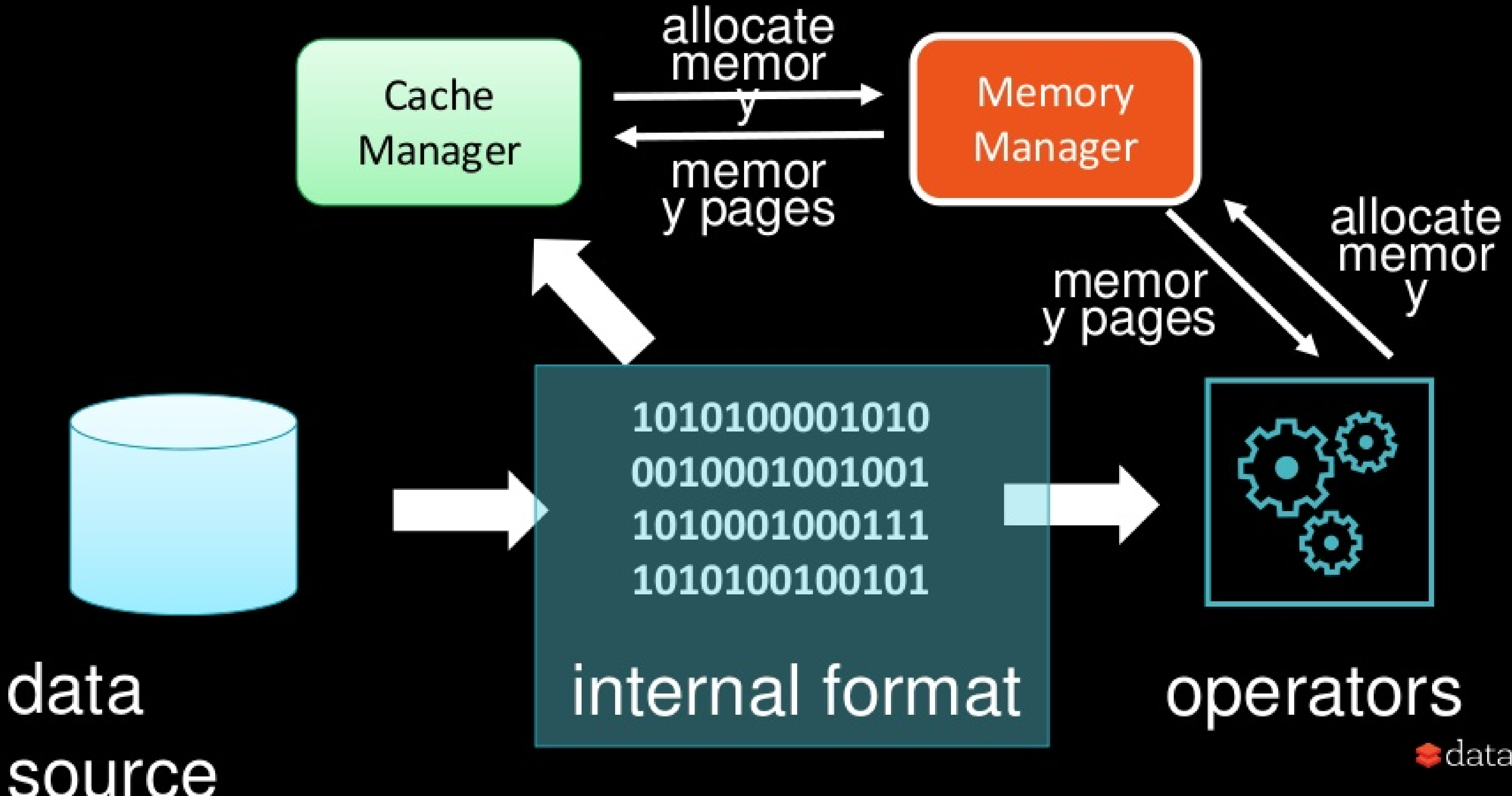
Memory Allocation

- Allocation happens in page granularity.
- Off-heap supported!
- Page is not fixed-size, but has a lower and upper bound.
- No pooling, pages are freed once there is no data on it.

Why var-length page and no pooling?

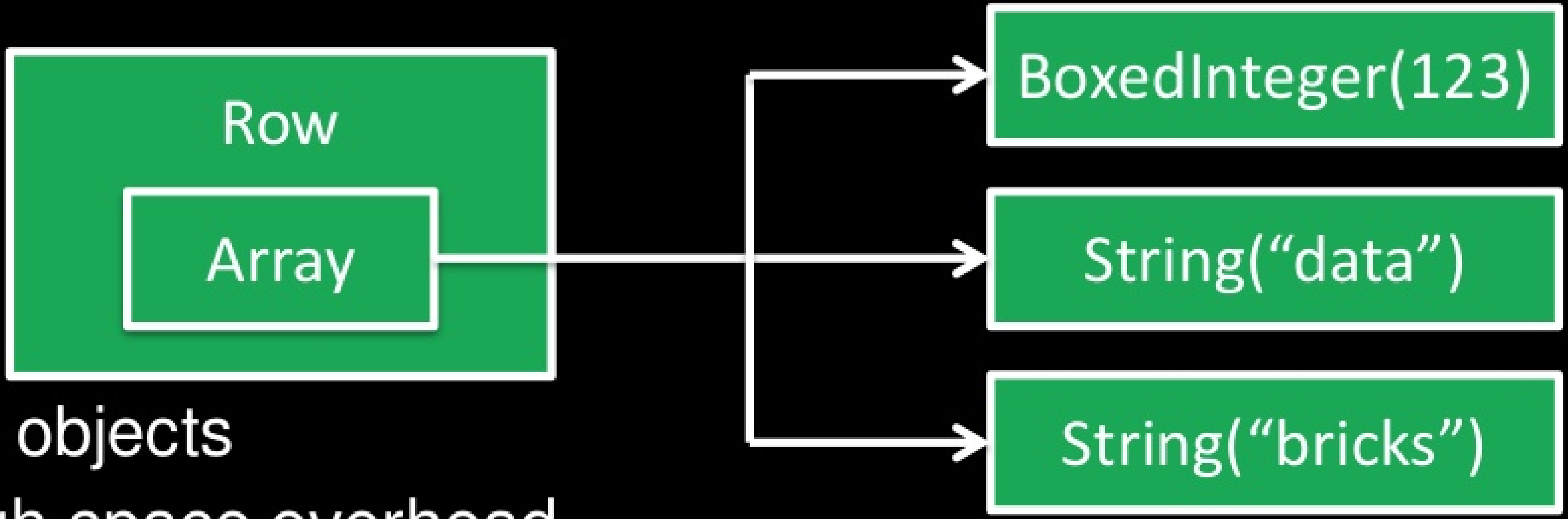
- Pros:
  - simplify the implementation. (no single record will across pages)
  - free memory immediately so that the OS can use them for file buffer, etc.
- Cons:
  - can not handle super big single record. (very rare in reality)
  - fragmentation for records bigger than page size lower bound. (the lower bound is several mega bytes, so it’s also rare)
  - overhead in allocation. (most malloc algorithms should work well)

Memory Model inside Executor



Java Objects Based Row Format

(123, “data”, “bricks”)



- 5+ objects
- high space overhead
- slow value accessing
- expensive hashCode()

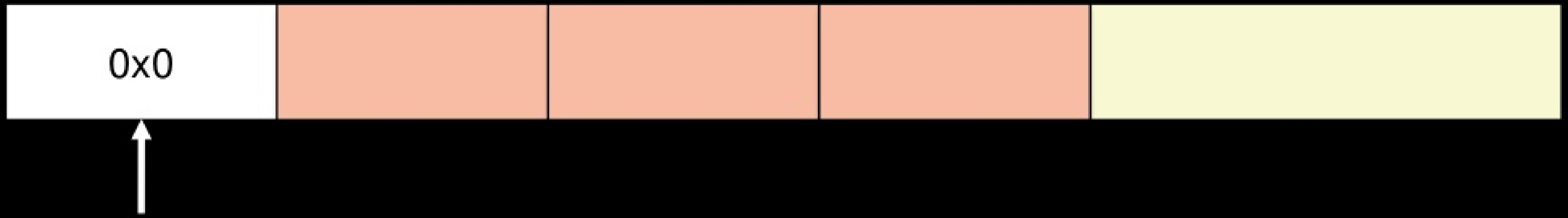
databricks

Data objects? No!

- It is hard to monitor and control the memory usage when we have a lot of objects.
- Garbage collection will be the killer.
- High serialization cost when transfer data inside cluster.

Efficient Binary Format

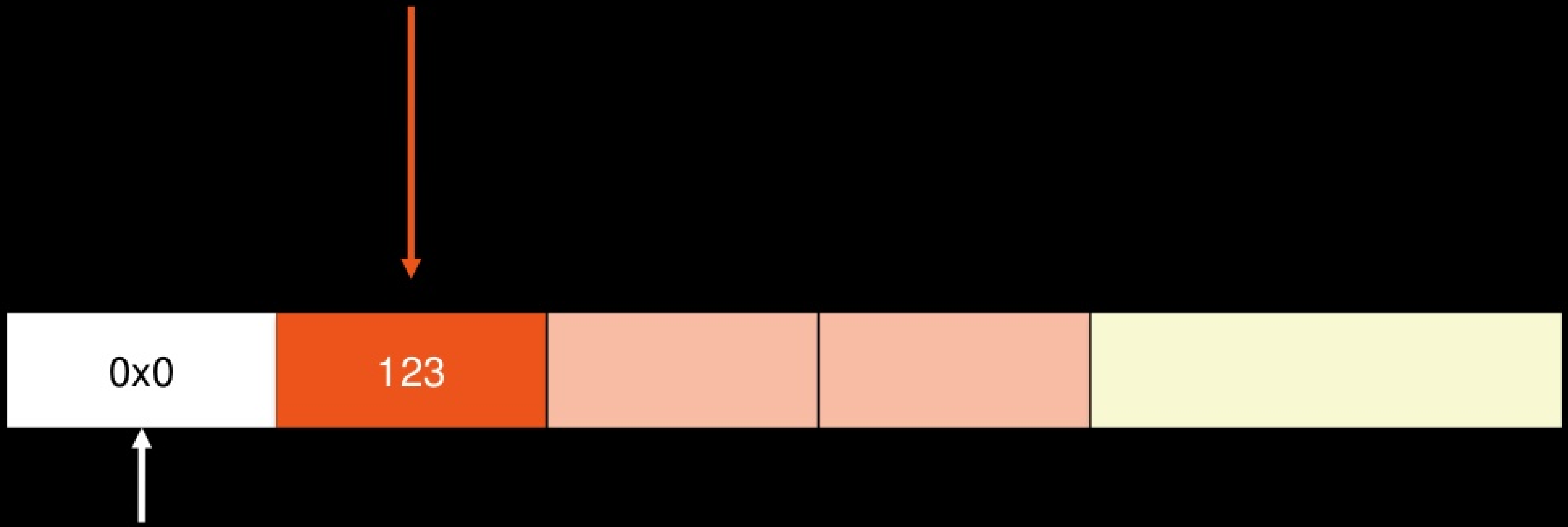
(123, “data”, “bricks”)



null tracking

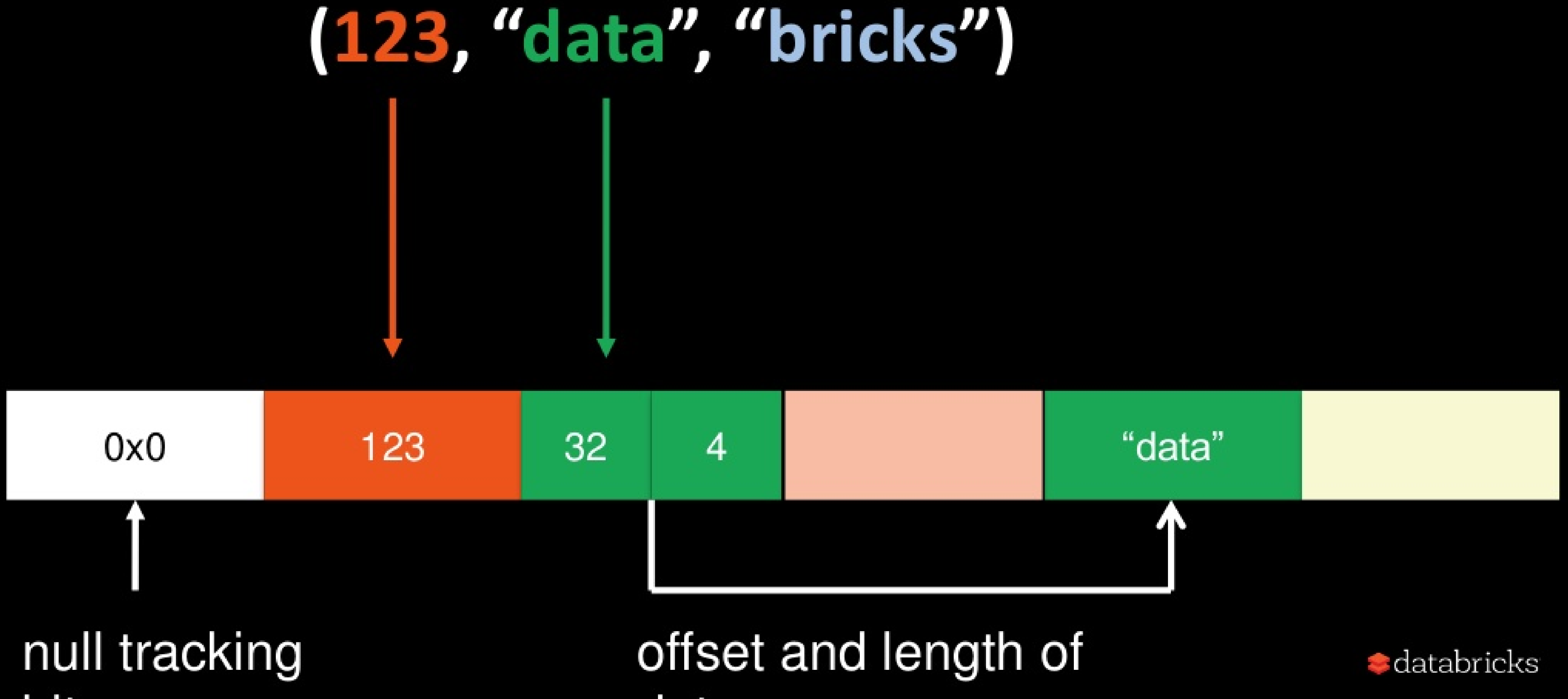
Efficient Binary Format

(123, “data”, “bricks”)



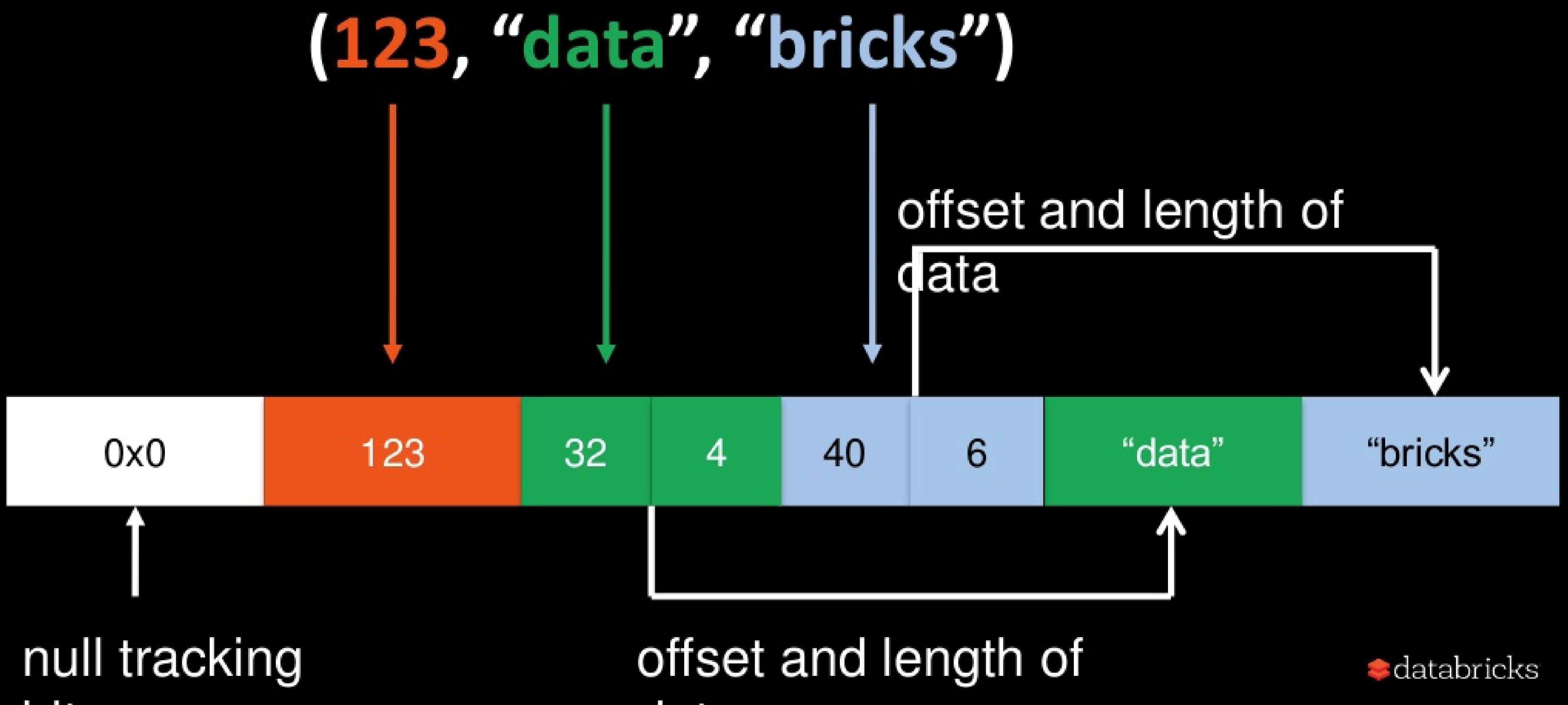
null tracking

Efficient Binary Format



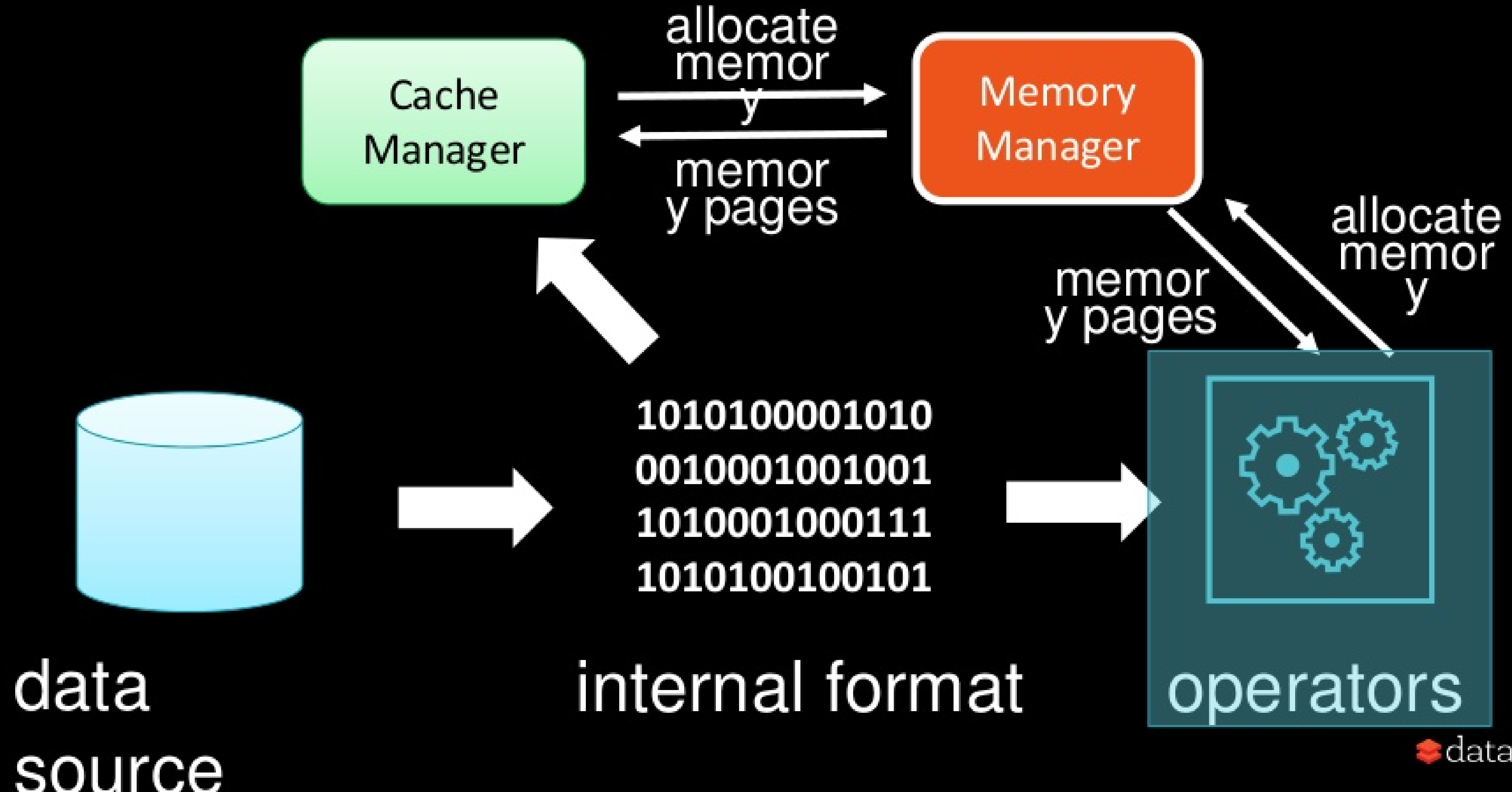
databricks

Efficient Binary Format



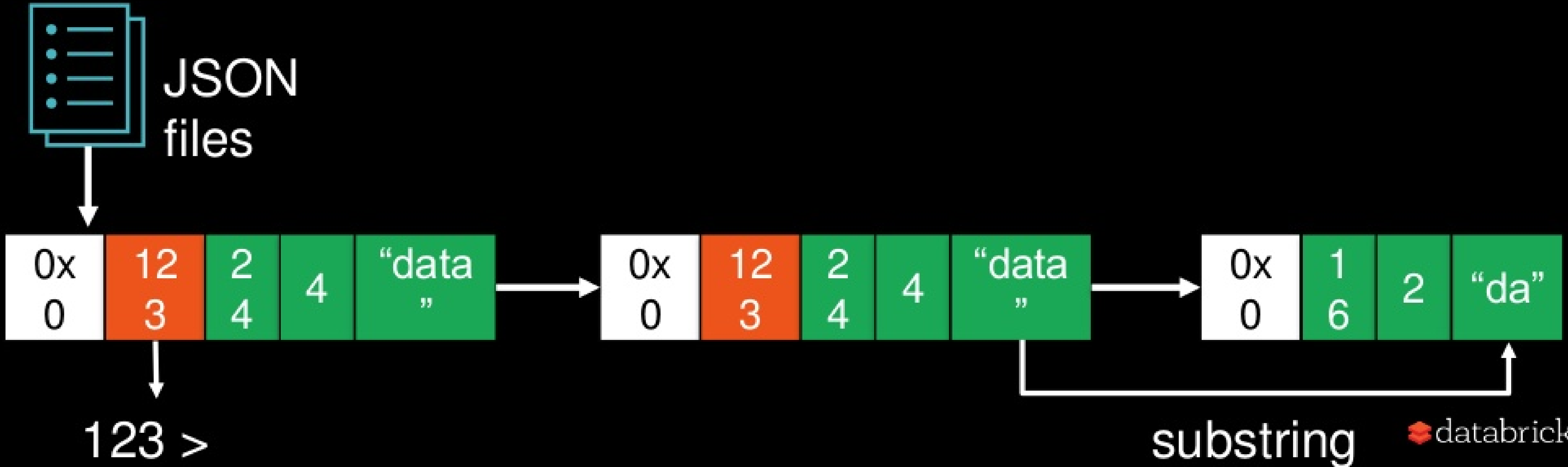
databricks

Memory Model inside Executor



Operate On Binary

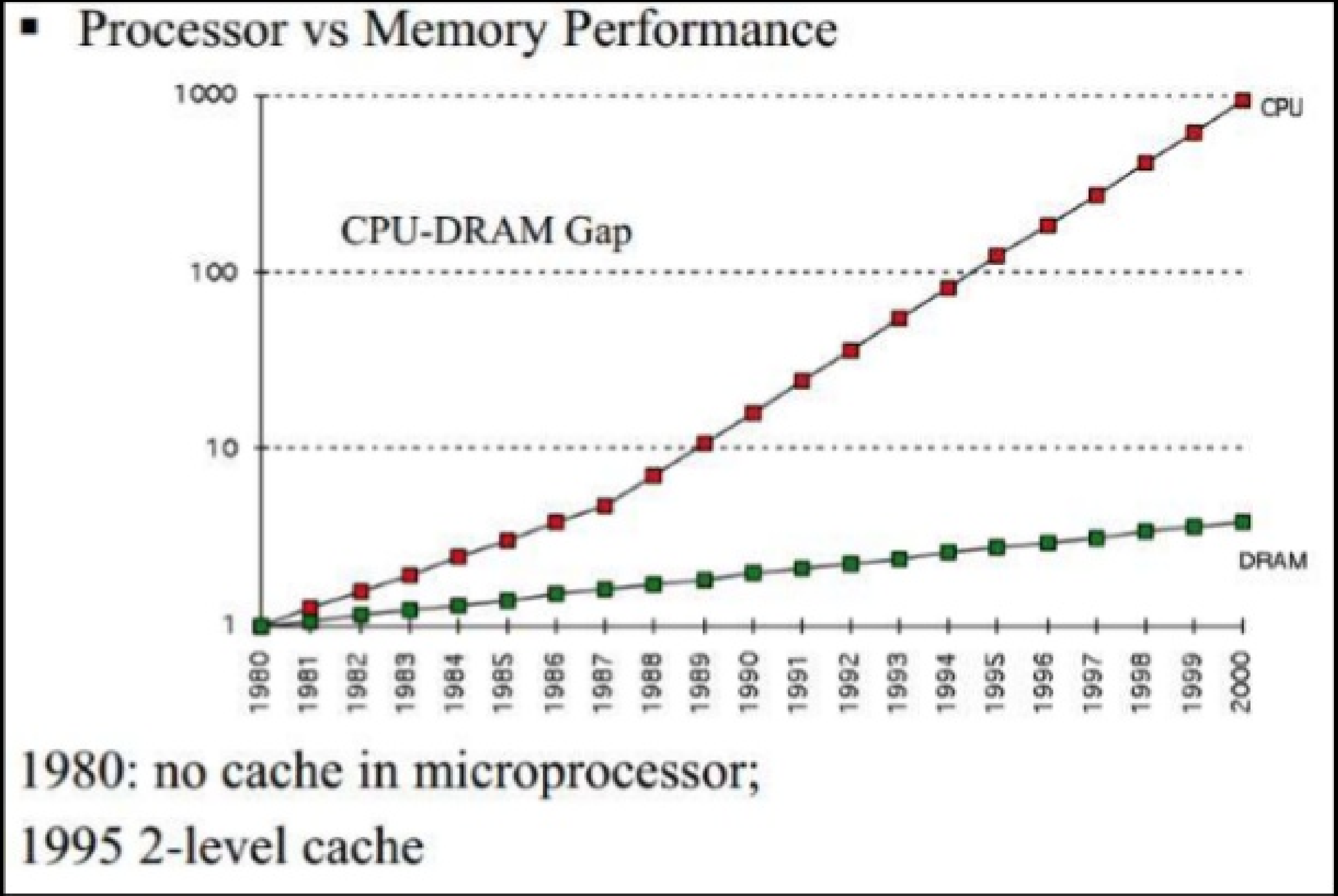
```
spark.read.schema("i int, j string").json("/tmp/x.json")
  .filter($"i" > 0)
  .select($"j".substr(0, 2))
```



databricks

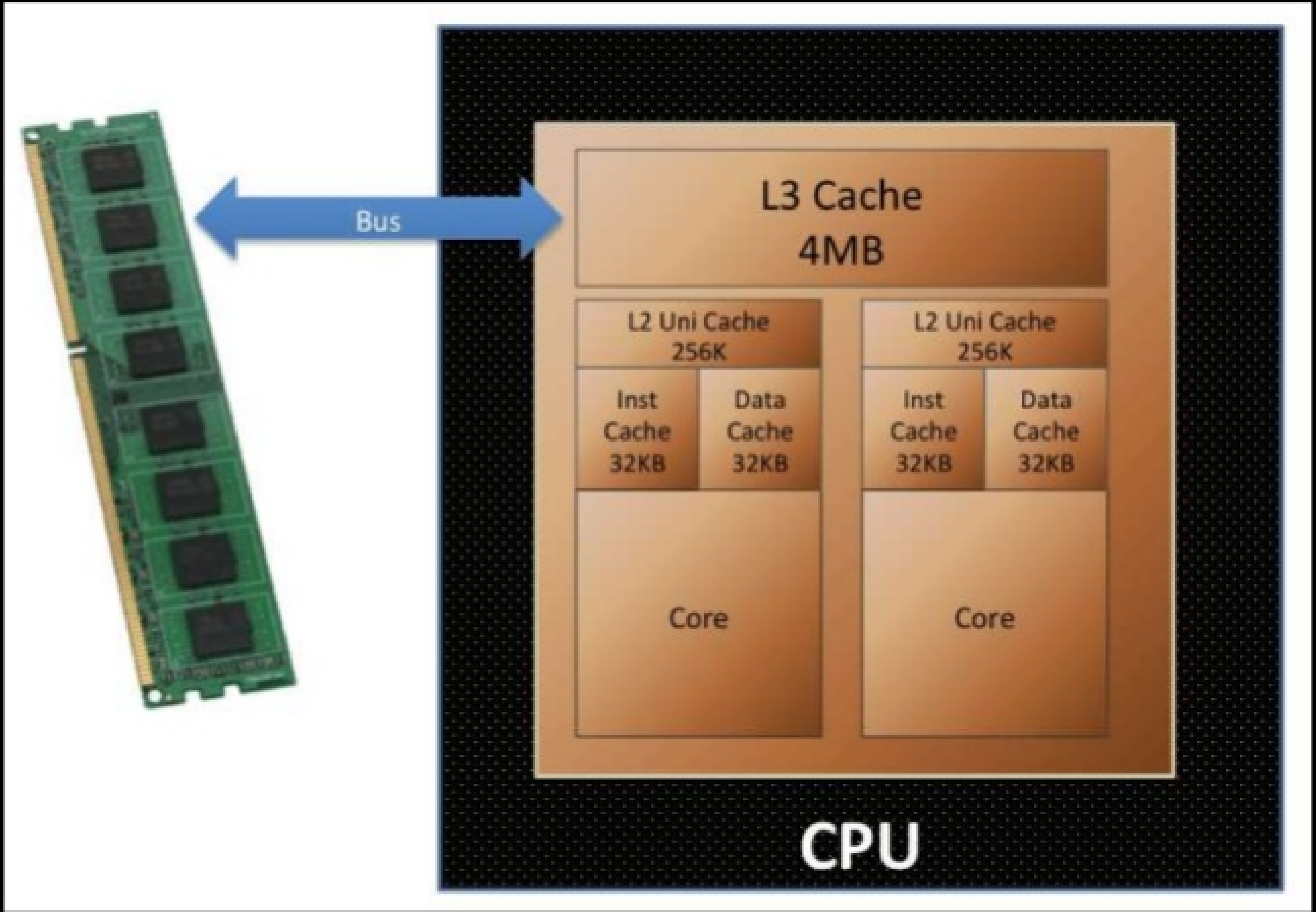
How to process binary data more efficiently?

Understanding CPU Cache



Memory is becoming slower and slower than CPU.

Understanding CPU Cache

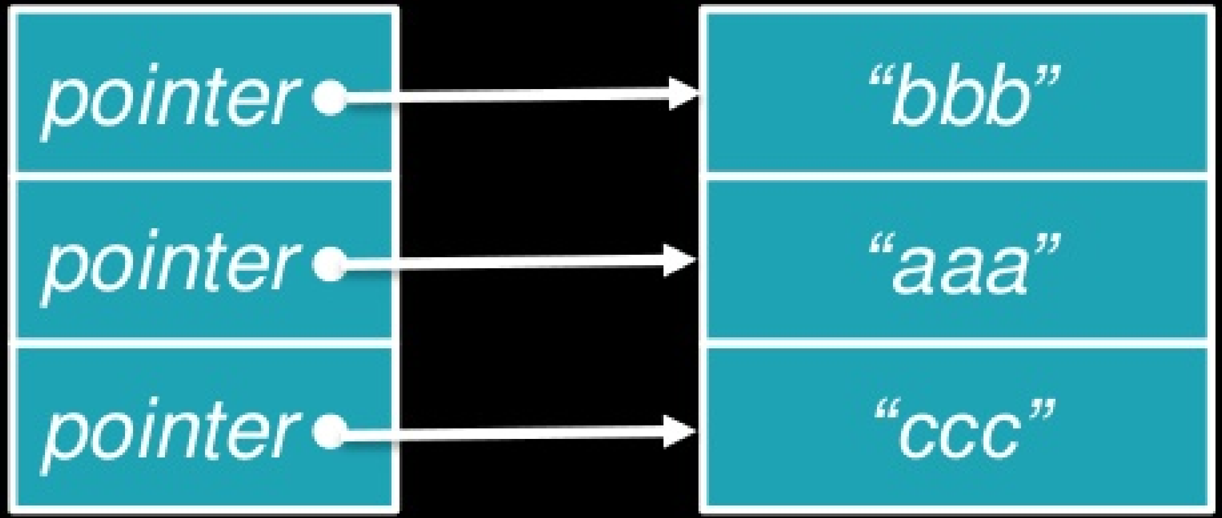


Pre-fetch frequently accessed data into CPU cache.

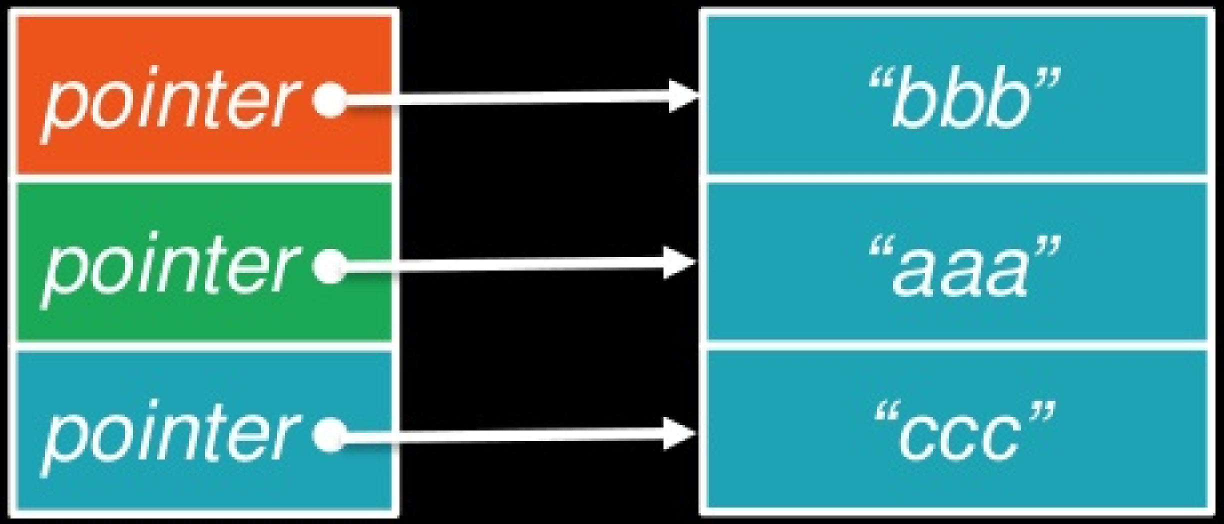
The most 2 important algorithms in big data are ...

Sort and **Hash**!

Naive Sort

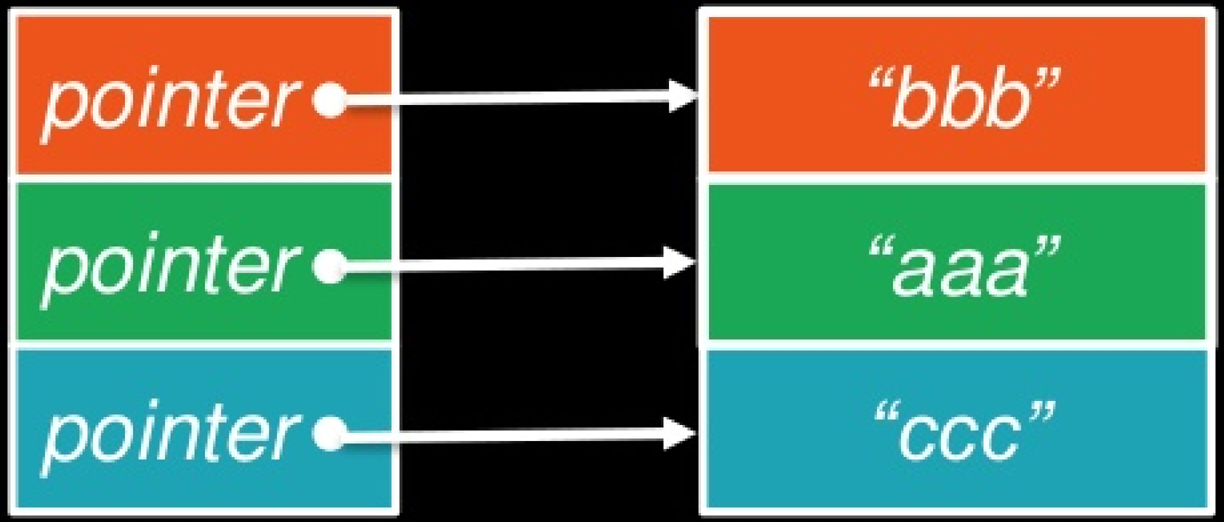


Naive Sort



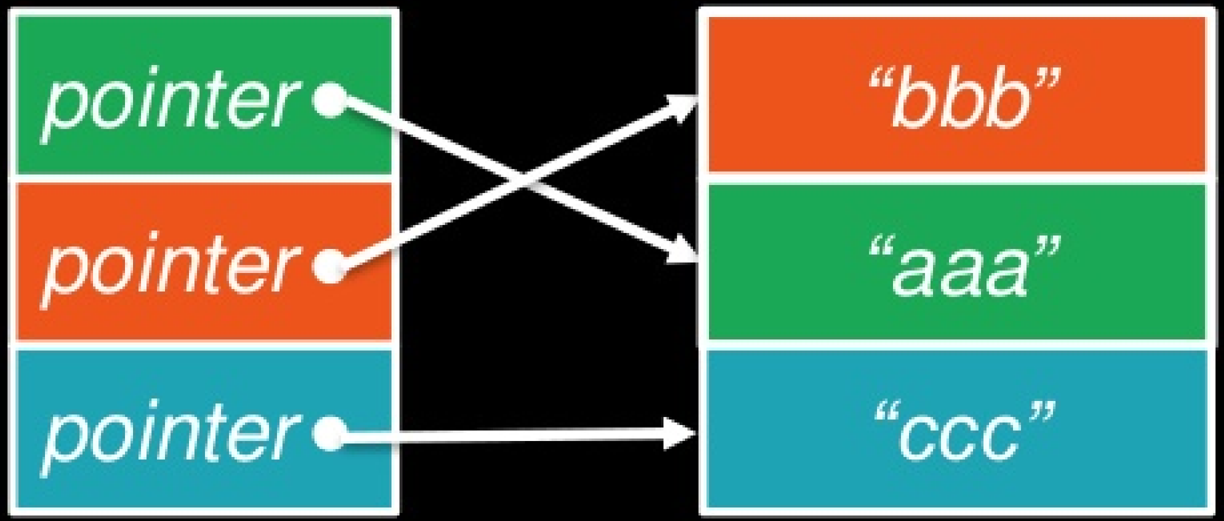
databricks

Naive Sort



databricks

Naive Sort

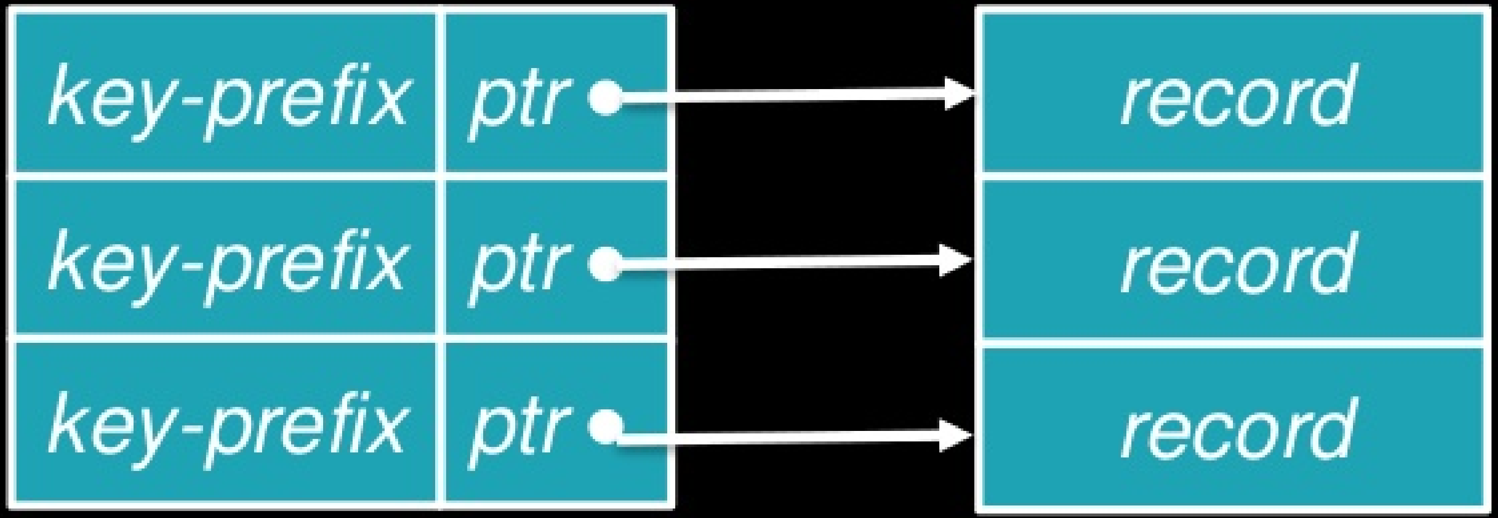


databricks

Naive Sort

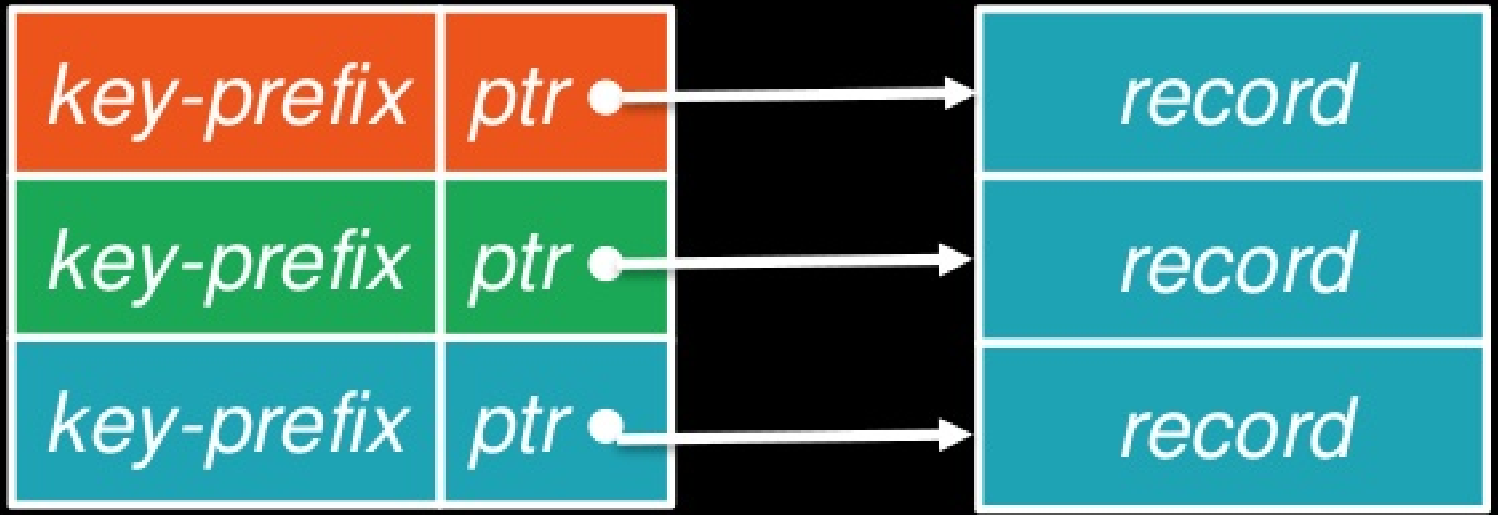
Each comparison needs to access 2 different memory regions, which makes it hard for CPU cache to pre-fetch data, poor cache locality!

Cache-aware Sort



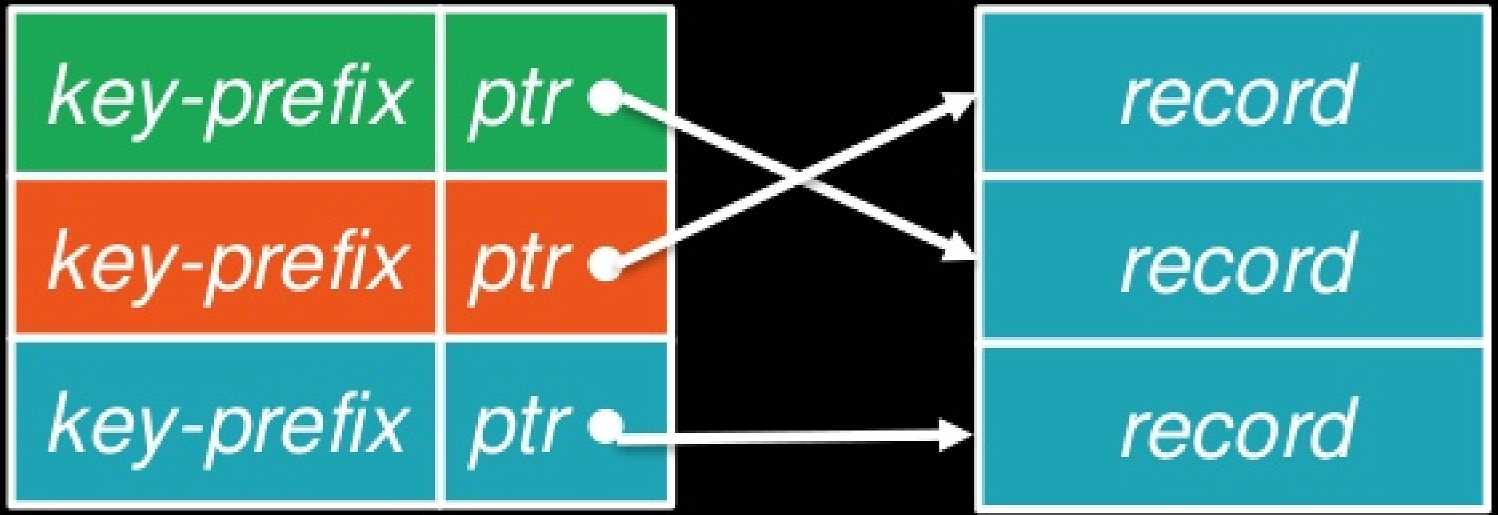
databricks

Cache-aware Sort



databricks

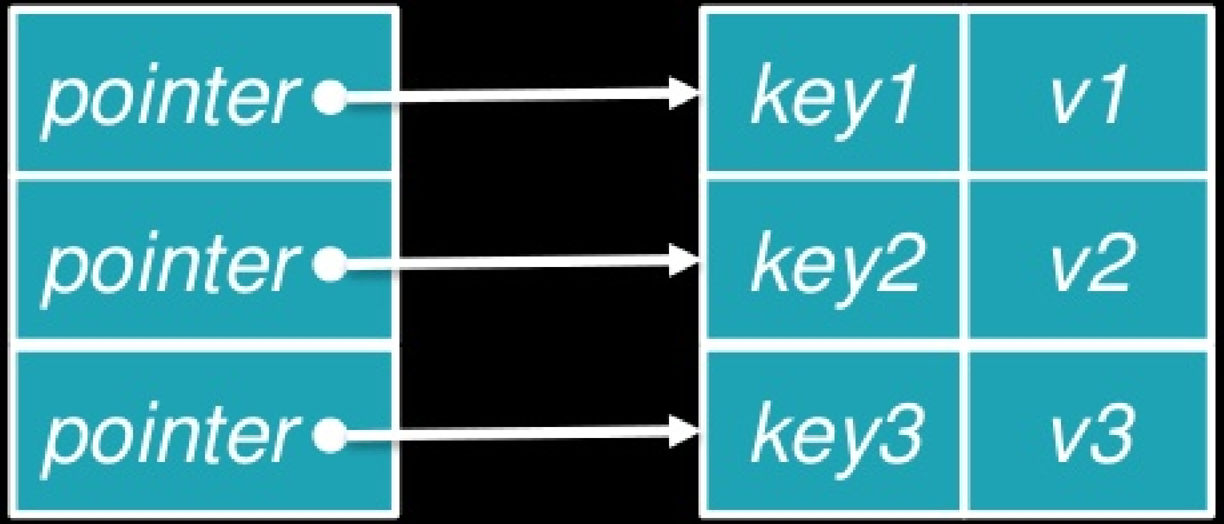
Cache-aware Sort



Cache-aware Sort

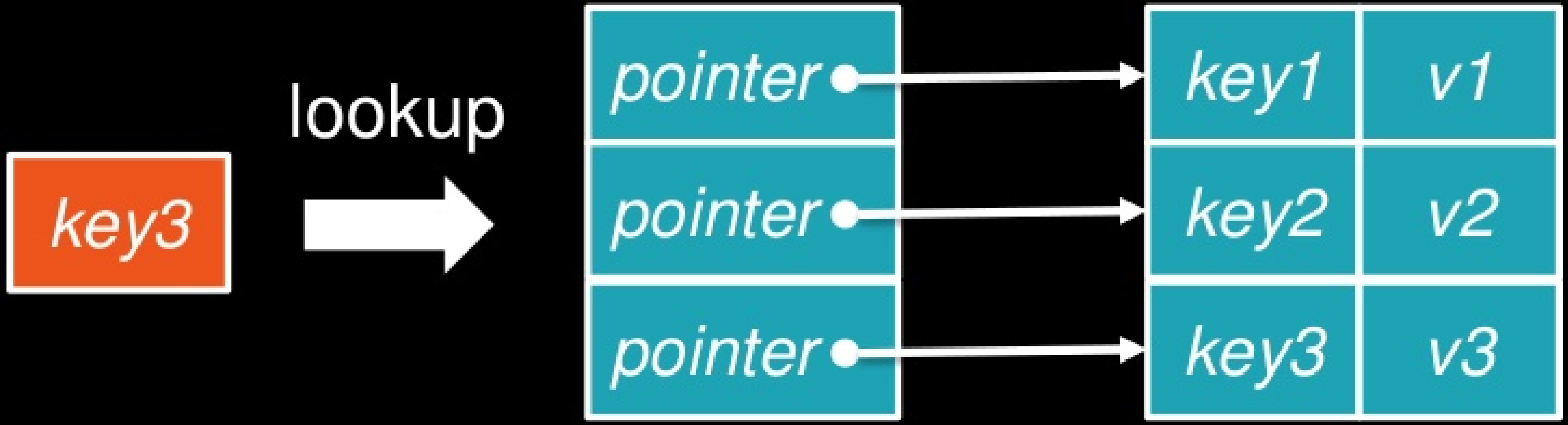
Most of the time, just go through the key-prefixes in a linear fashion, good cache locality!

Naive Hash Map



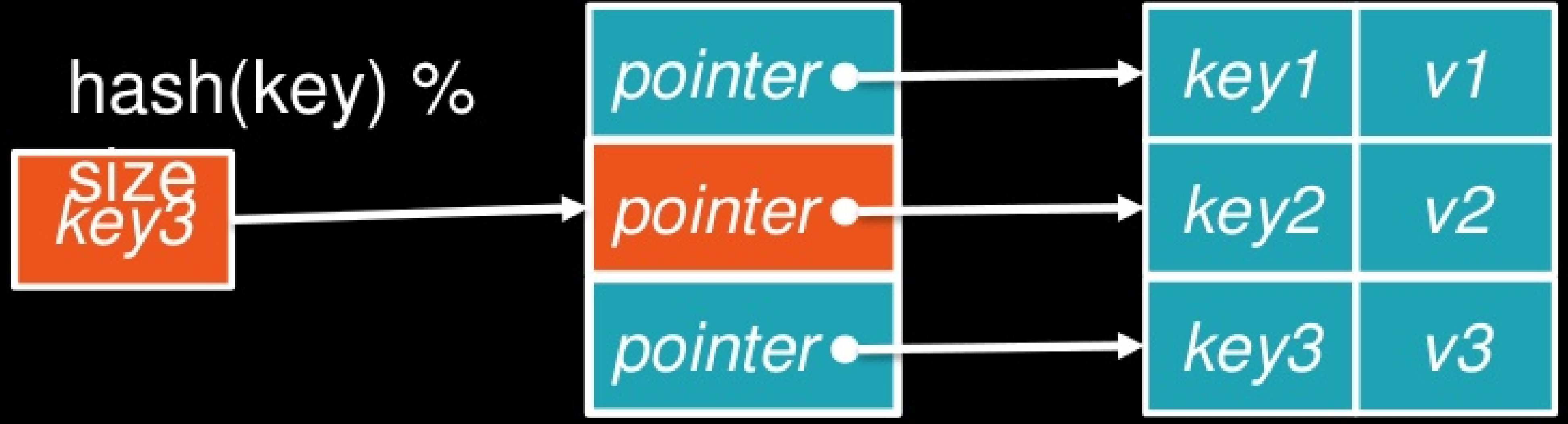
databricks

Naive Hash Map

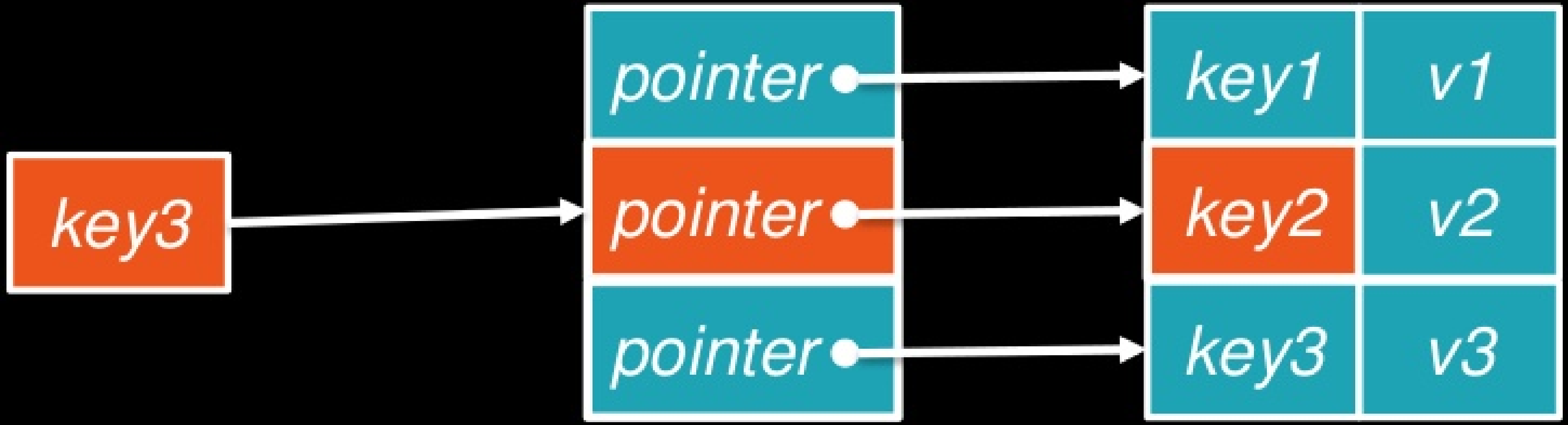


databricks

Naive Hash Map

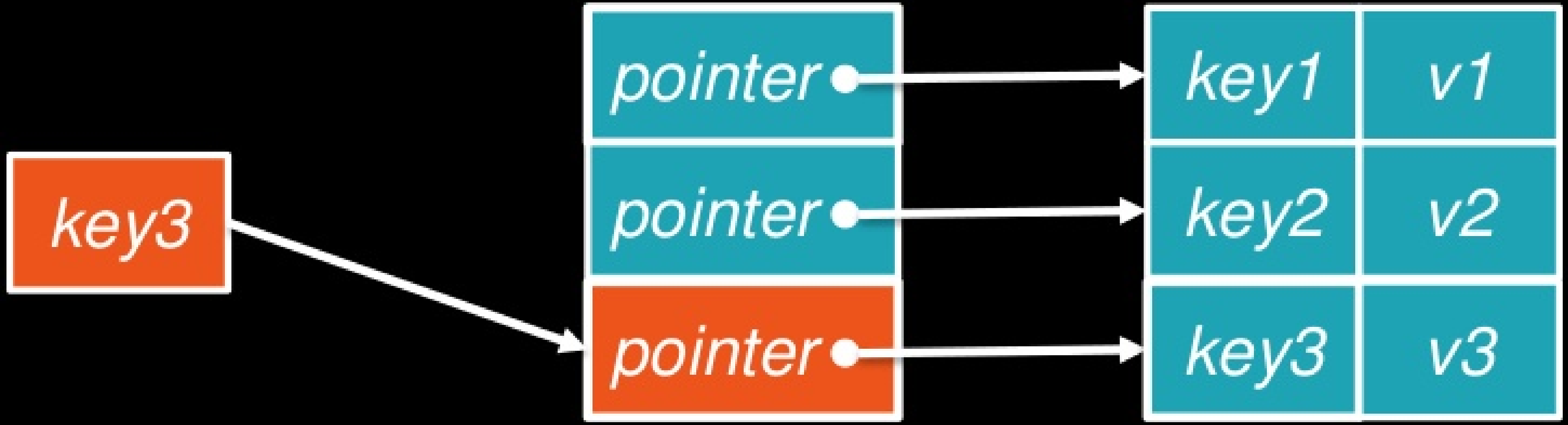


Naive Hash Map



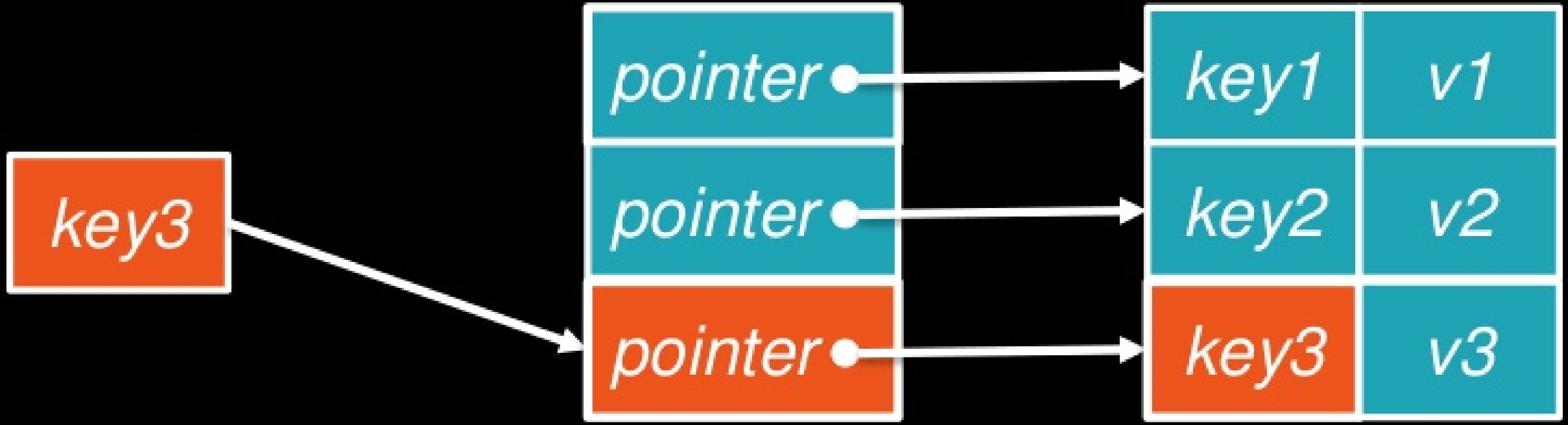
compare these 2
keys

Naive Hash Map



quadratic
probing

Naive Hash Map

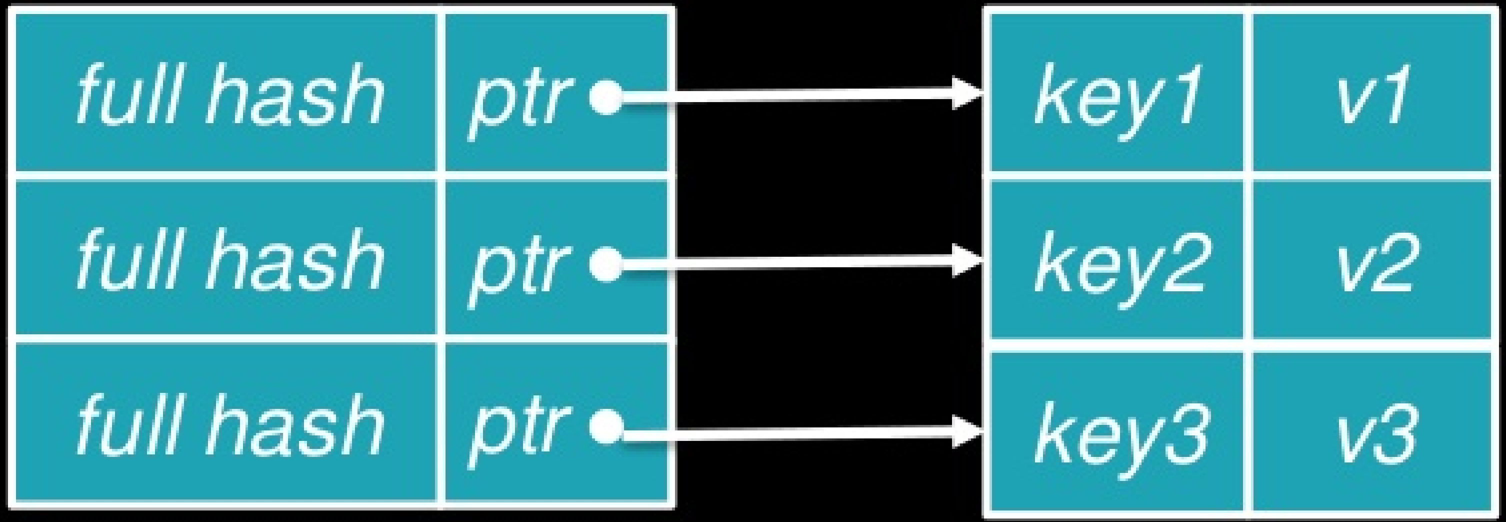


compare these 2 keys

Naive Hash Map

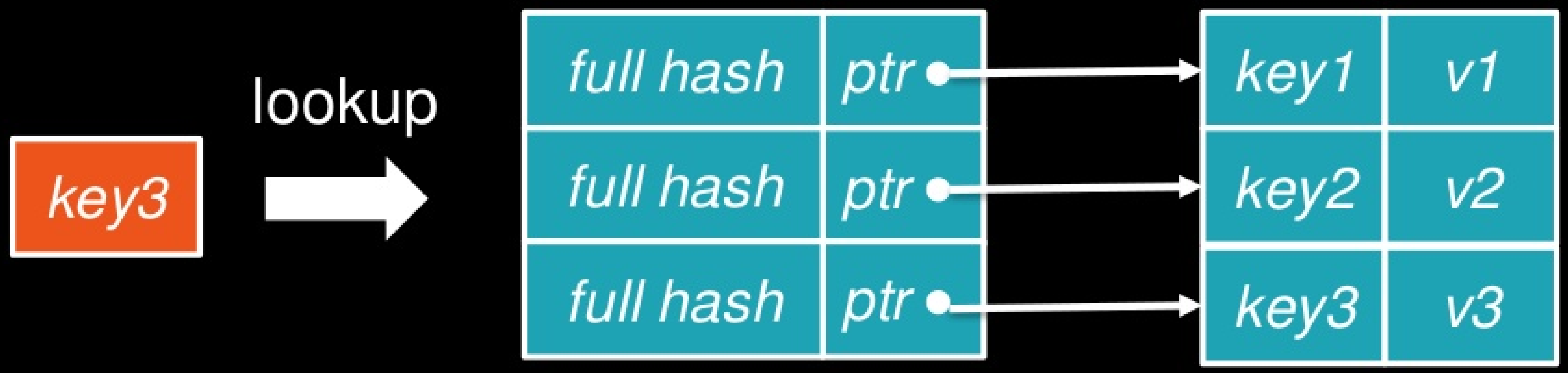
Each lookup needs many pointer dereferences and key comparison when hash collision happens, and jumps between 2 memory regions, bad cache locality!

Cache-aware Hash Map



databricks

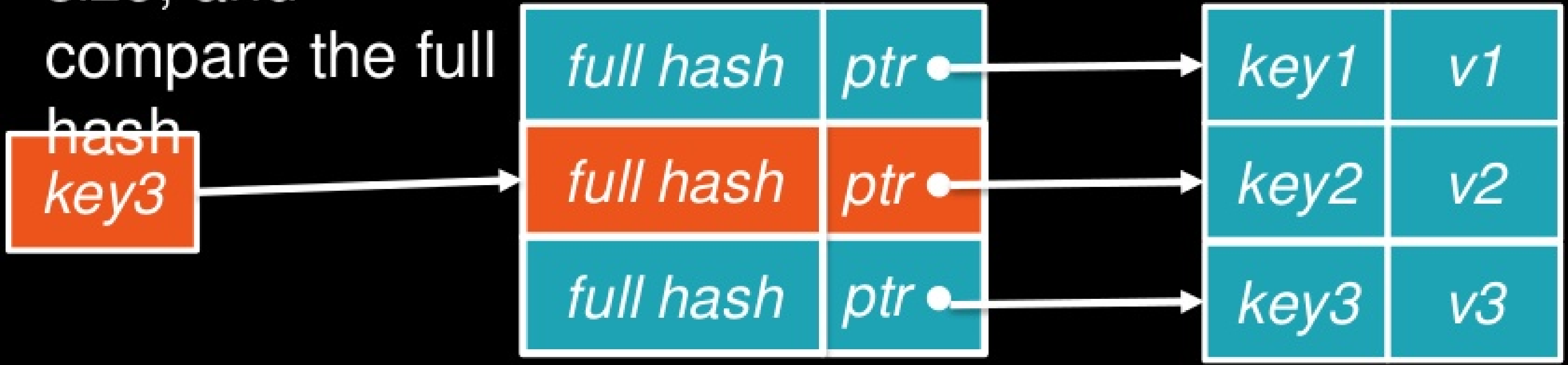
Cache-aware Hash Map



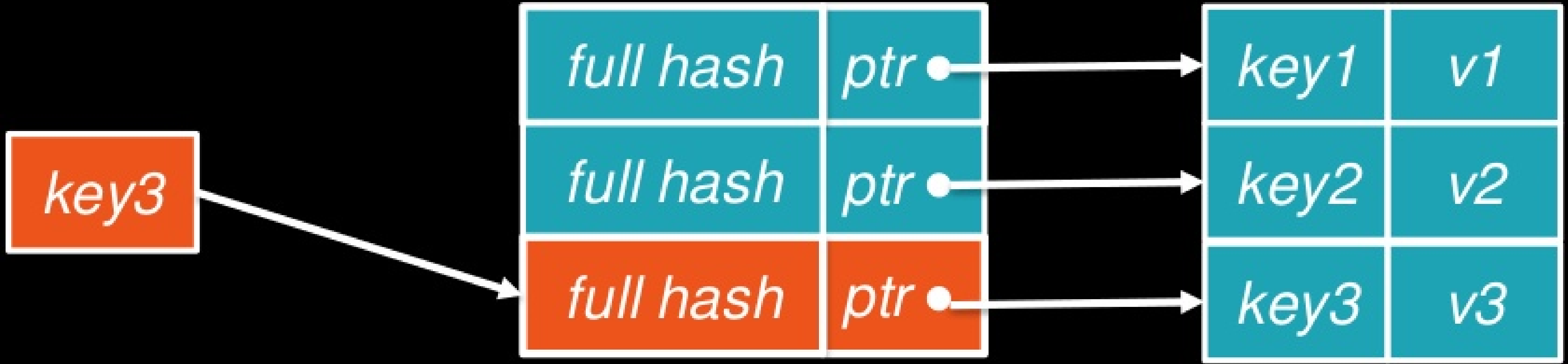
databricks

Cache-aware Hash Map

hash(key) % size, and compare the full hash

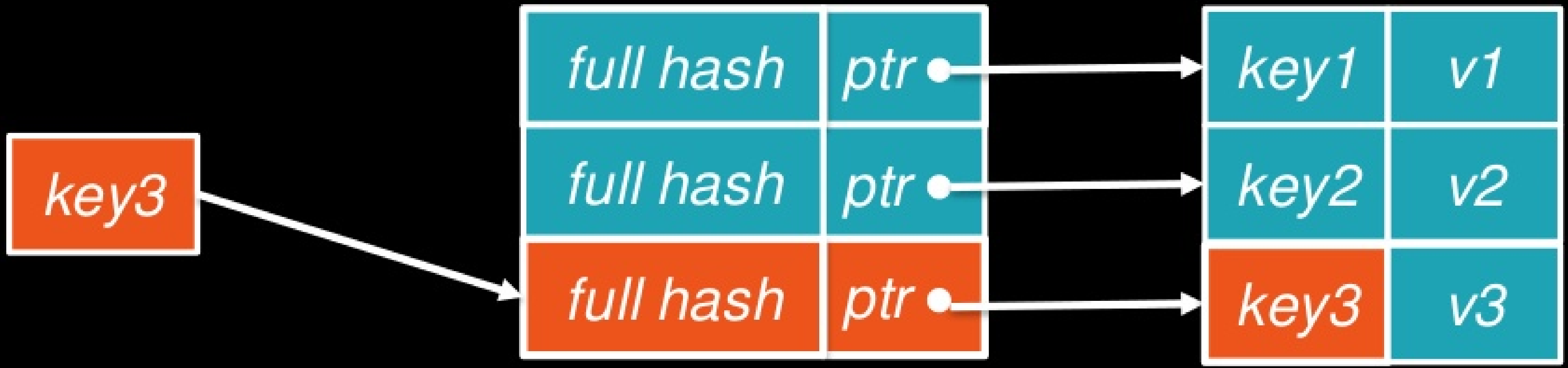


Cache-aware Hash Map



quadratic probing, and
compare the full hash

Cache-aware Hash Map



compare these 2 keys

Cache-aware Hash Map

Each lookup mostly only needs one pointer dereference and key comparison(full hash collision is rare), and access data mostly in a single memory region, better cache locality!

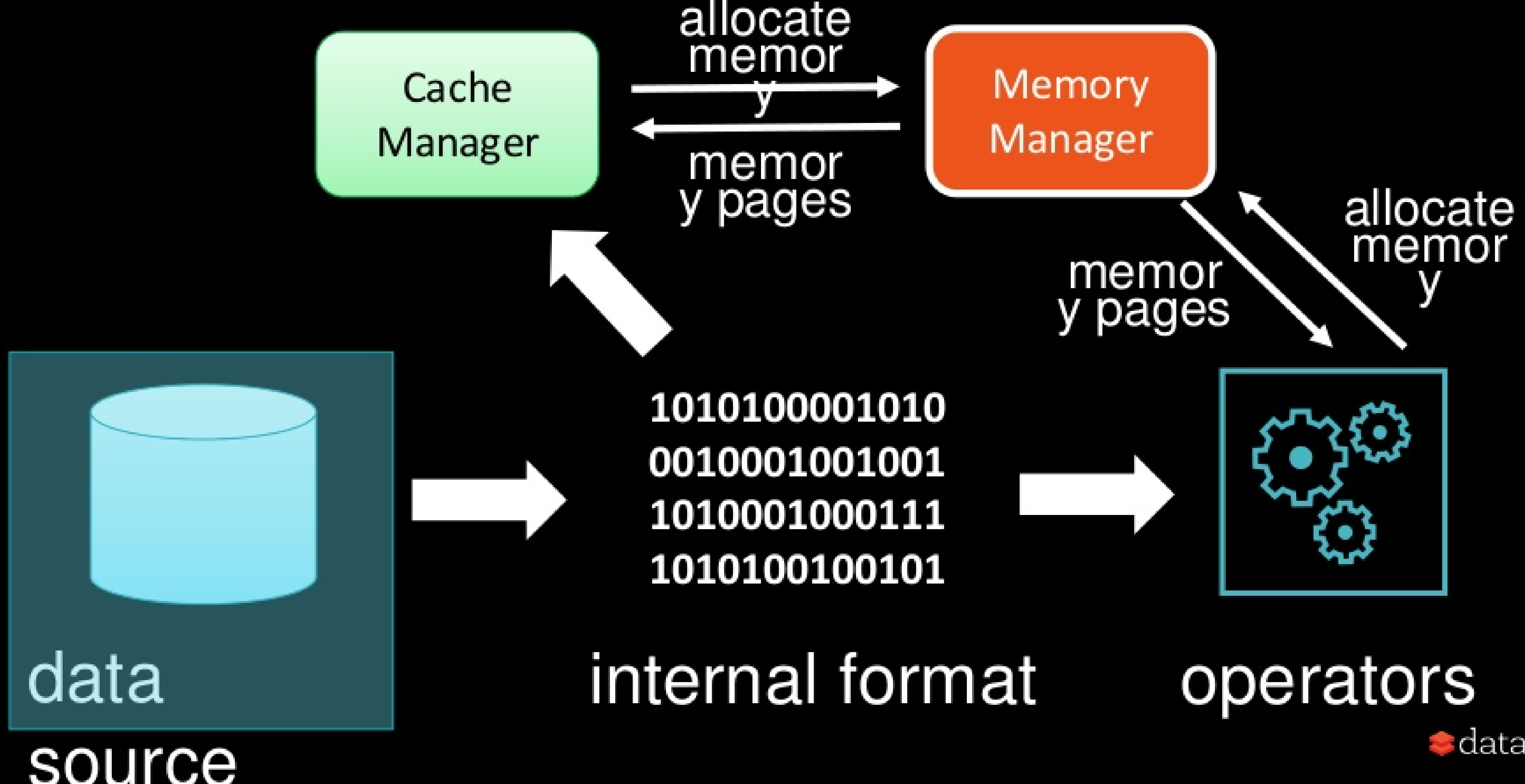
Recap: Cache-aware data structure

How to improve cache locality …

- store key-prefix with pointer.
- store key full hash with pointer.

Store extra information to try to keep the memory accessing in a single region.

Memory Model inside Executor



databricks

Future Work

- SPARK-19489: Stable serialization format for external & native code integration.
- SPARK-15689: Data source API v2
- SPARK-15687: Columnar execution engine.

Try Apache Spark in Databricks!

UNIFIED ANALYTICS PLATFORM

- Collaborative cloud environment
- Free version (community edition)

DATABRICKS RUNTIME 3.0

- Apache Spark - optimized for the cloud
- Caching and optimization layer - DBIO
- Enterprise security - DBES

Try for free today.
databricks.com

Thank You

databricks