# Deep Dive: How Spark Uses Memory

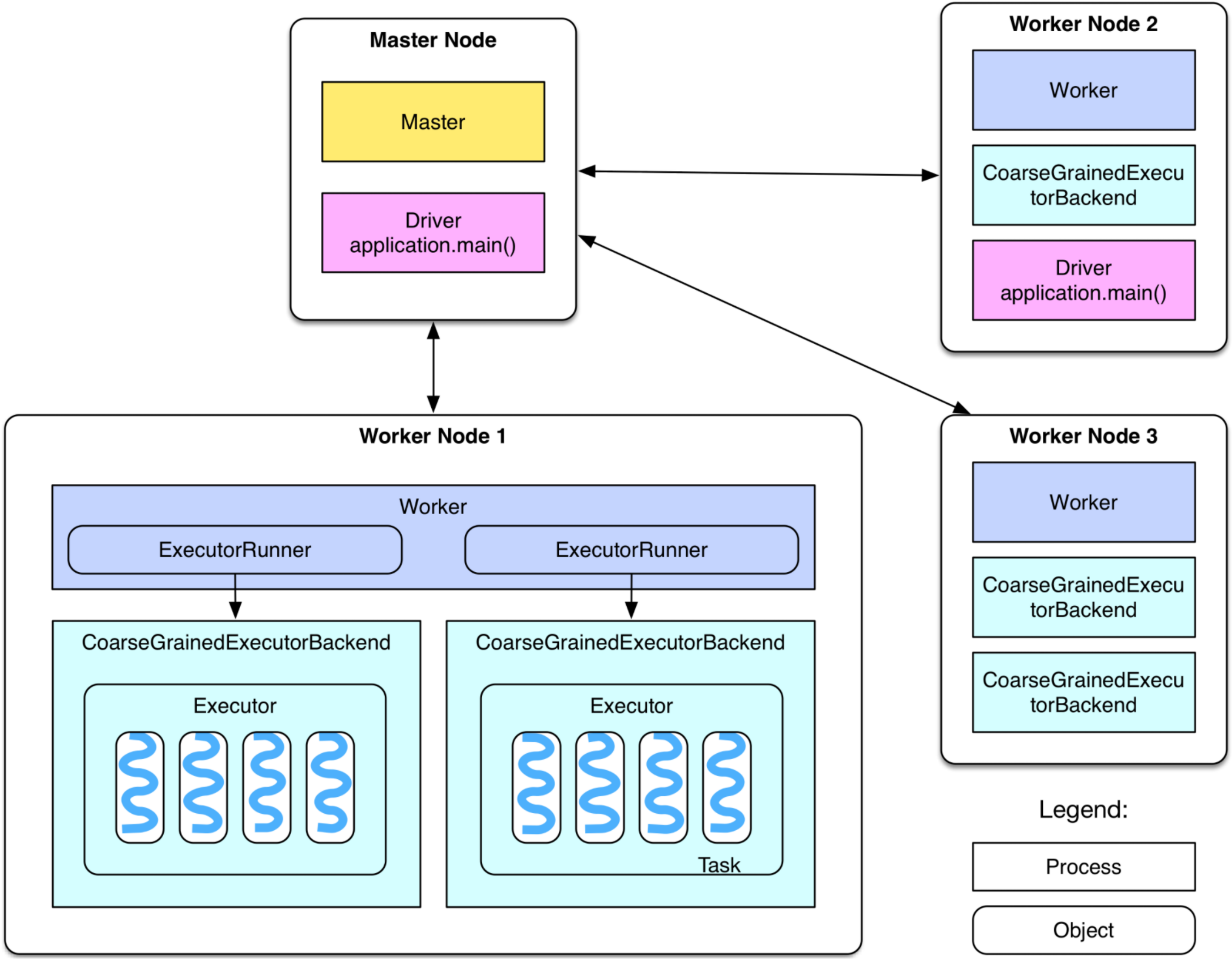
Wenchen Fan

2017-5-19

databricks

# Agenda

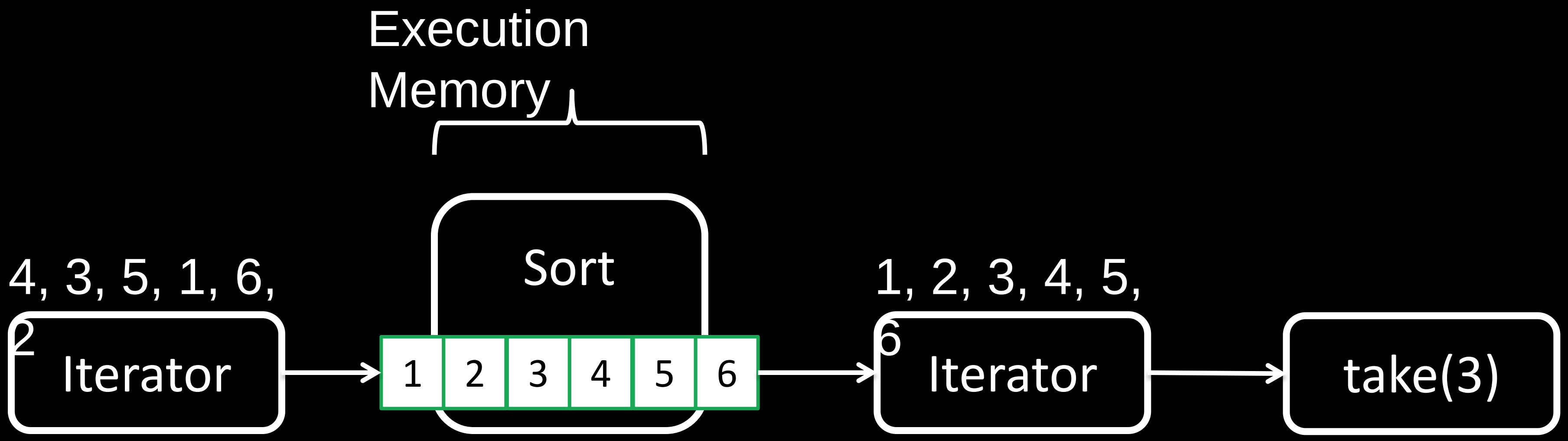
- Memory Usage Overview
- Memory Contention
- Tungsten Memory Format
- Cache-aware Computation
- Future Plans



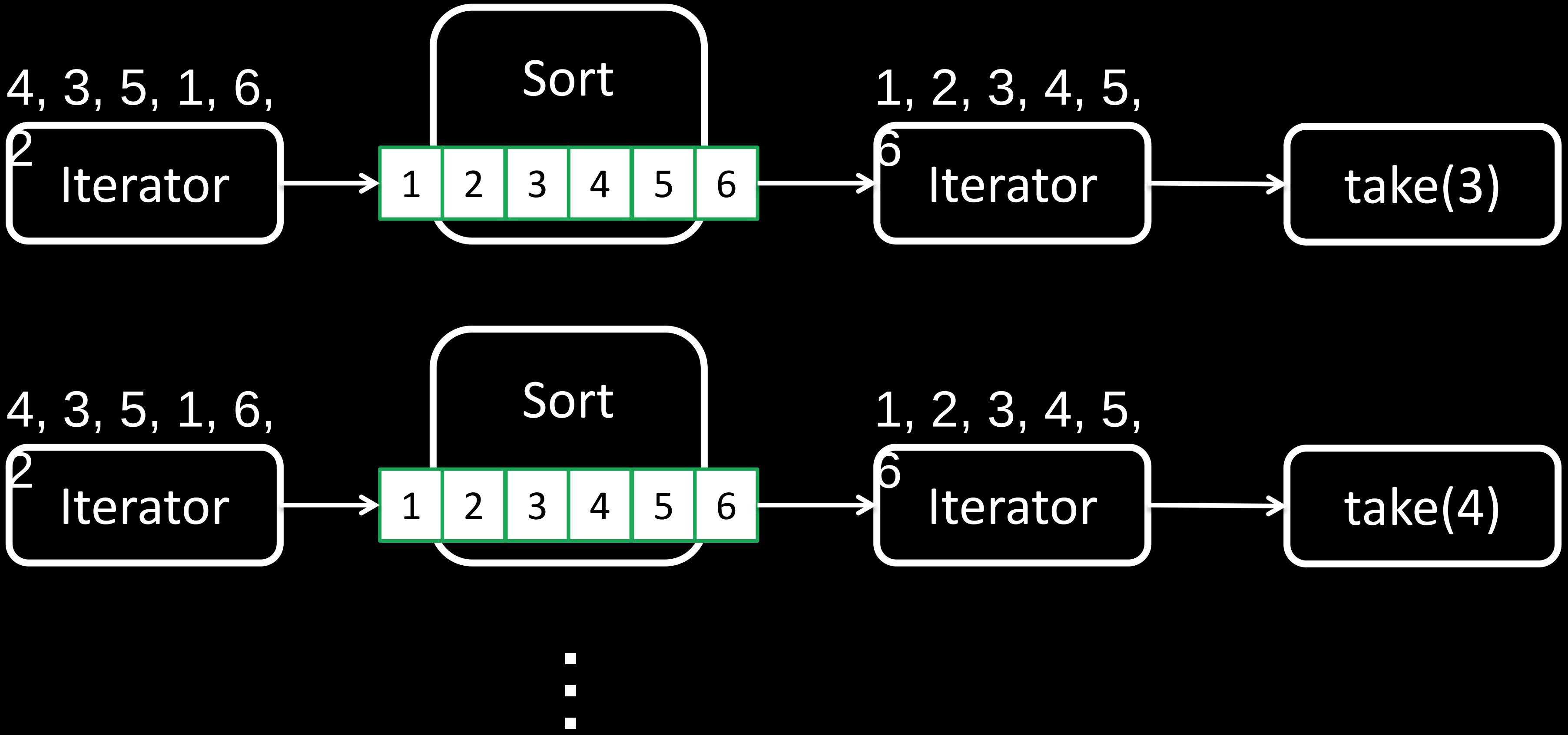
Master Node
Master
Driver
application.main()
Worker Node 2
Worker
CoarseGrainedExecutorBackend
Driver
application.main()
Worker Node 1
Worker
ExecutorRunner
ExecutorRunner
CoarseGrainedExecutorBackend
Executor
CoarseGrainedExecutorBackend
Executor
Task
Worker Node 3
Worker
CoarseGrainedExecutorBackend
CoarseGrainedExecutorBackend
Legend:
Process
Object

# Where Spark Uses Memory

- **storage**: memory used to cache data that will be used later. (controlled by memory manager)
- **execution**: memory used for computation in shuffles, joins, sorts and aggregations. (controlled by memory manager)
- **others**: user data structure, internal metadata, objects created by UDF, etc.

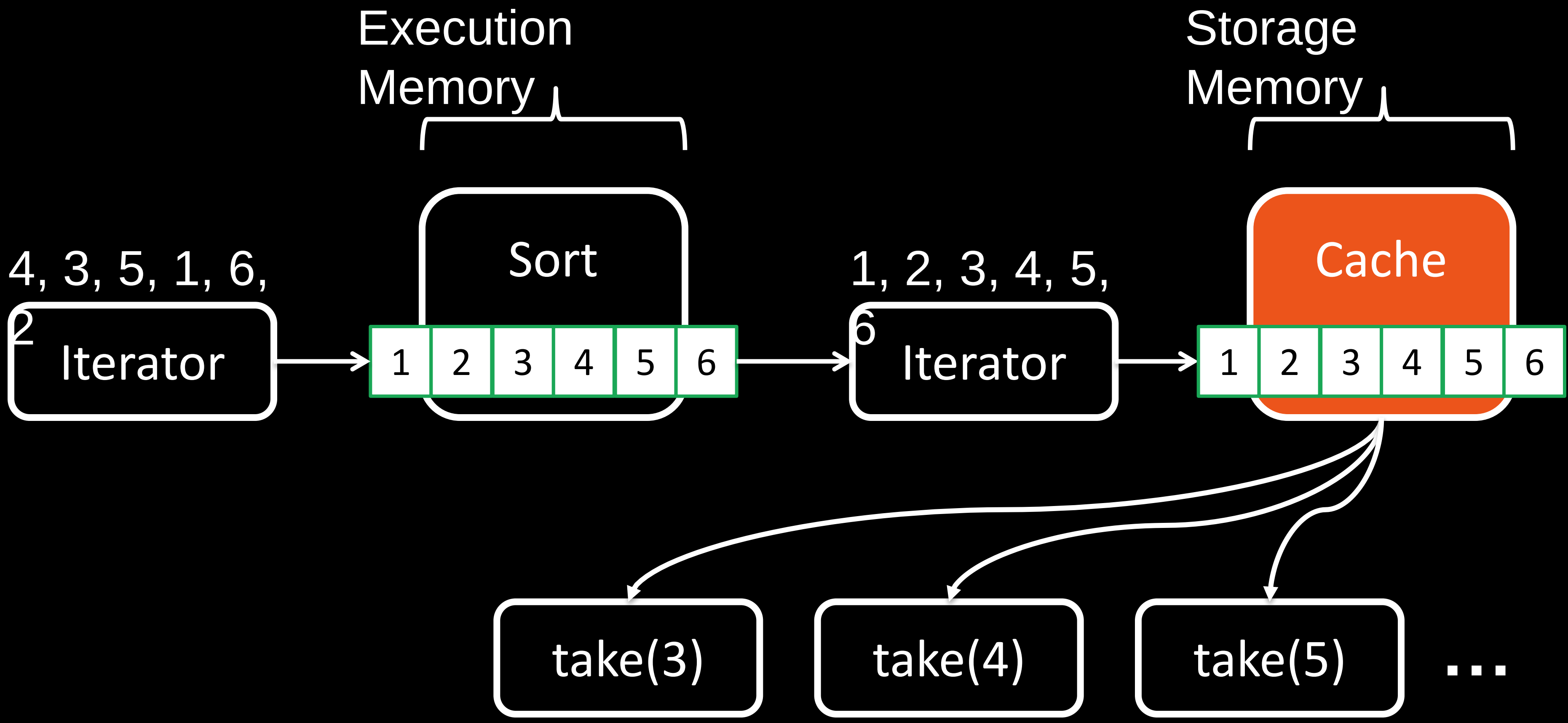



What if I want the sorted values again?



4, 3, 5, 1, 6, 2
Iterator
Sort
1 2 3 4 5 6
1, 2, 3, 4, 5, 6
Iterator
take(3)
4, 3, 5, 1, 6, 2
Iterator
Sort
1 2 3 4 5 6
1, 2, 3, 4, 5, 6
Iterator
take(4)


databricks


Execution Memory
Storage Memory
4, 3, 5, 1, 6, 2
Iterator
Sort
1 2 3 4 5 6
1, 2, 3, 4, 5, 6
Iterator
Cache
1 2 3 4 5 6
take(3)
take(4)
take(5)
...
databricks

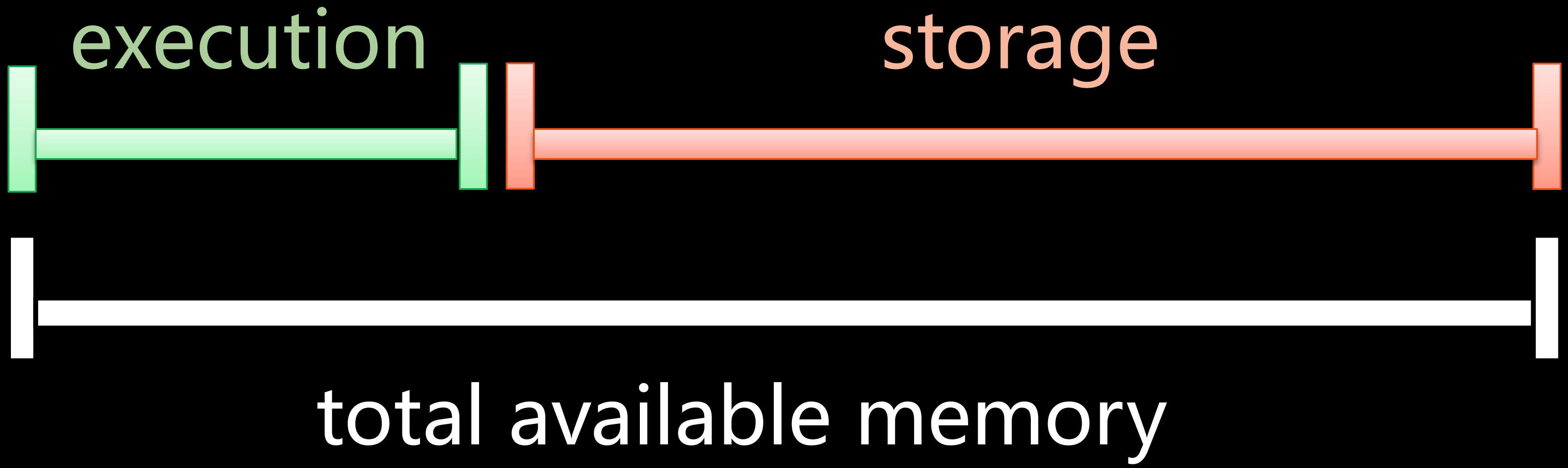
# Memory Contention

- How to arbitrate memory between execution and storage?
- How to arbitrate memory across tasks running in parallel?
- How to arbitrate memory across operators running within the same task?

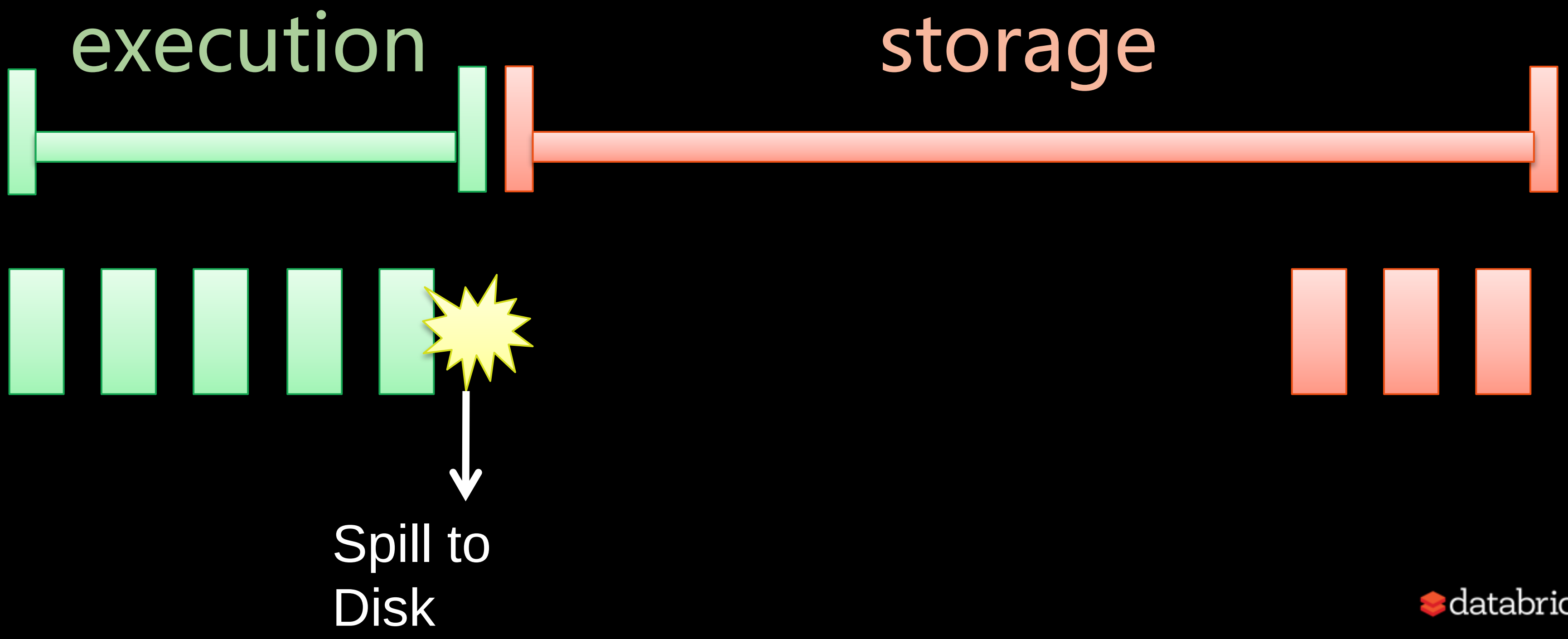
# Challenge #1

How to arbitrate memory between execution and storage?

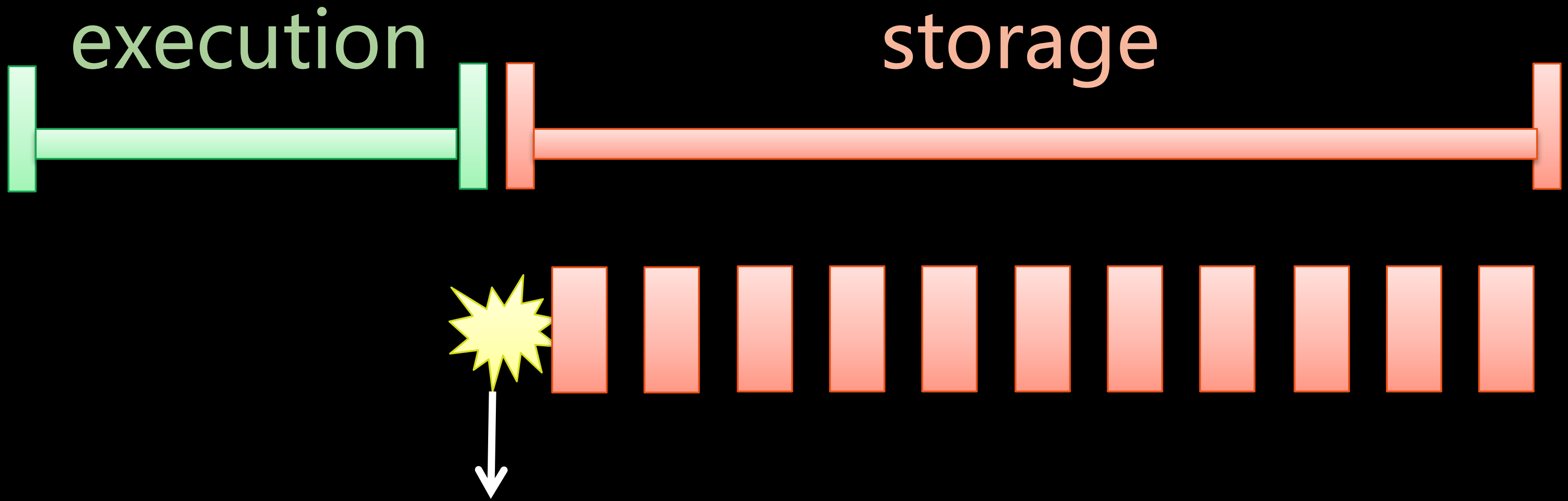
# Easy, static assignment!



# Easy, static assignment!



# Easy, static assignment!



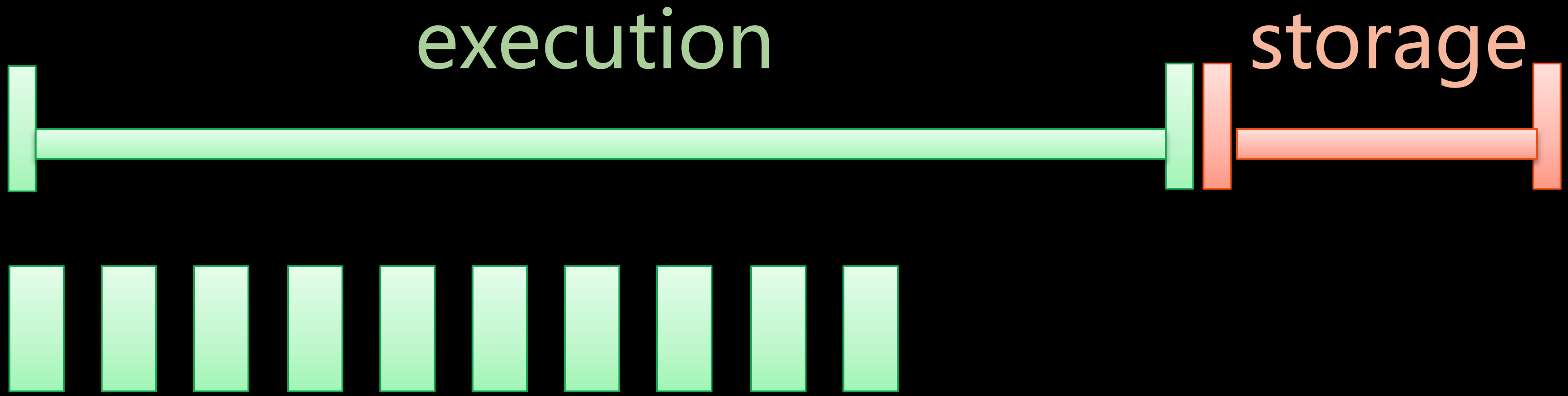

Evict LRU blocks to disk

Inefficient memory usage leads to bad performance

# Easy, static assignment!

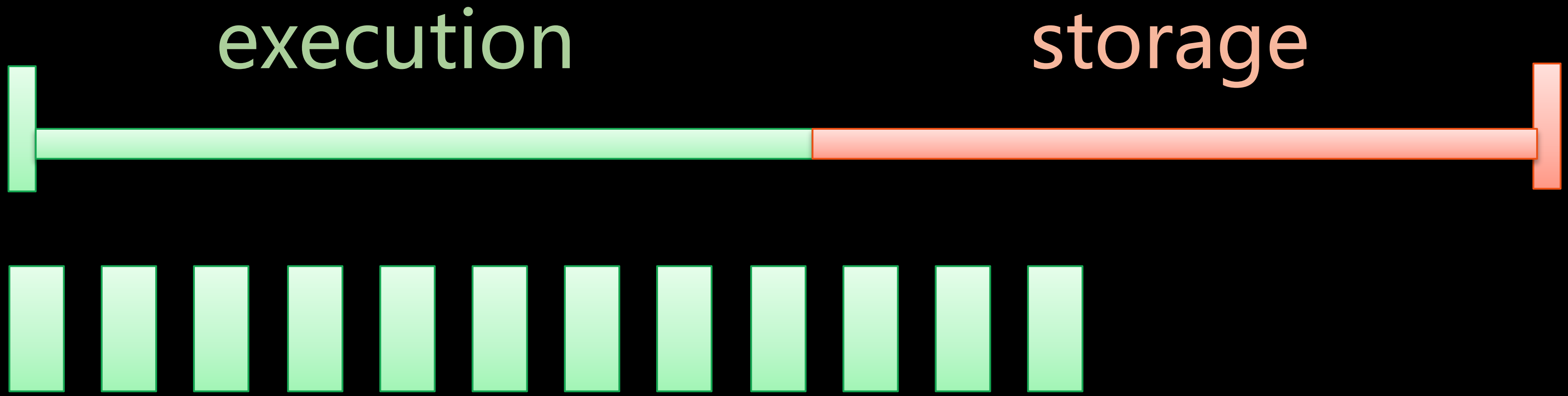

Execution can only use a fraction of the memory, even when there is no storage!

# Easy, static assignment!



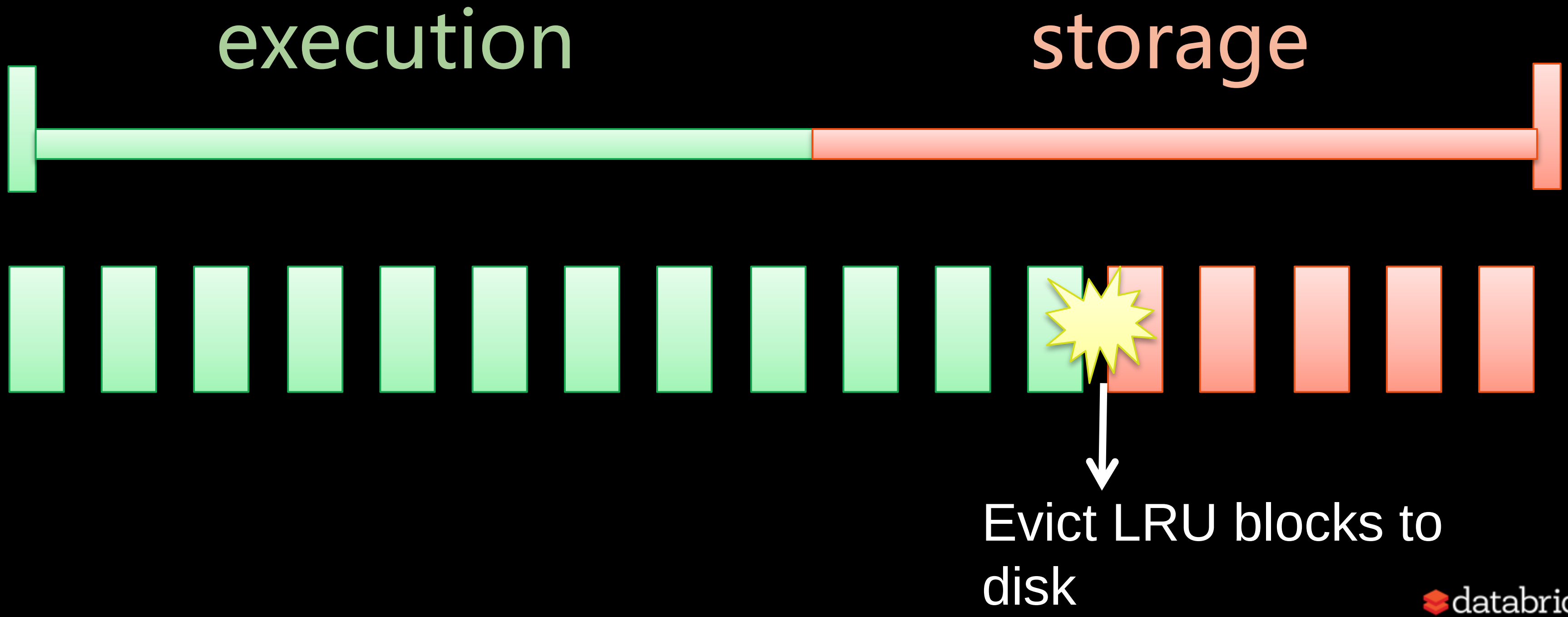

Efficient use of memory required user tuning

# Unified Memory Management

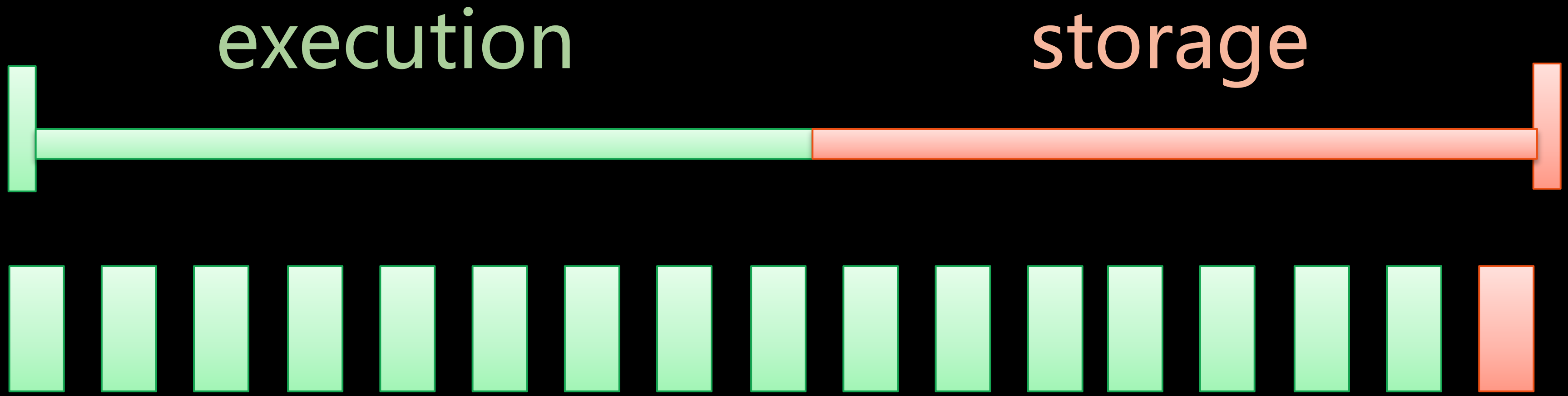



What happens if there is already storage?

# Unified Memory Management



# Unified Memory Management




databricks

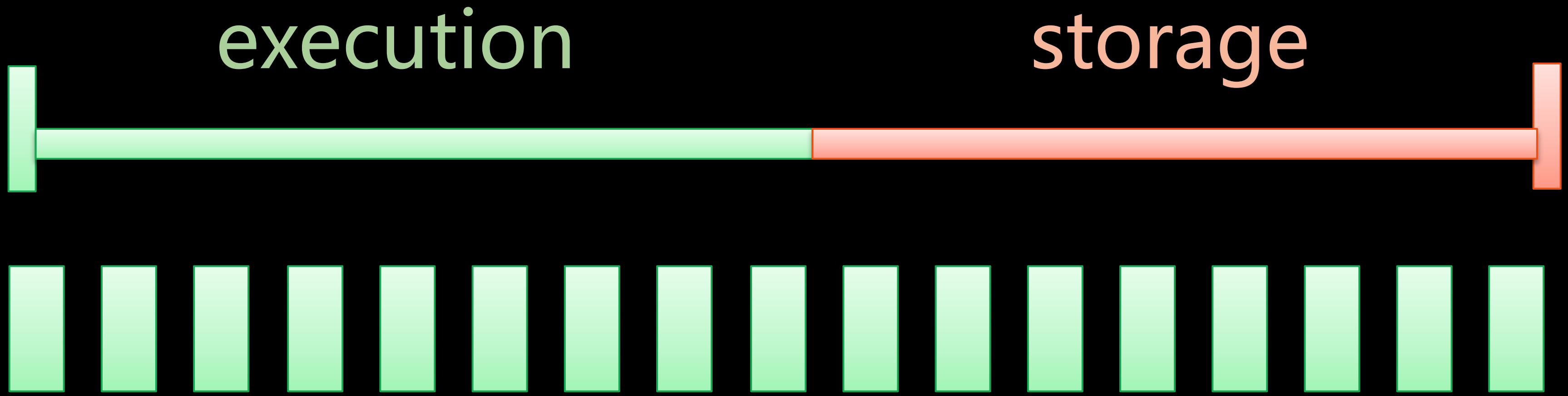
# Design Considerations

- Why evict storage, not execution?
  - Spilled execution data will always be read back from disk, where as cached data may not.

- What if the application relies on cache?

databricks

# Unified Memory Management




This is bad!

databricks

# Design Considerations

- Why evict storage, not execution?
  - Spilled execution data will always be read back from disk, where as cached data may not.

- What if the application relies on cache?
  - allow users to specify a minimum unevictable amount of cached data(not a reservation!)

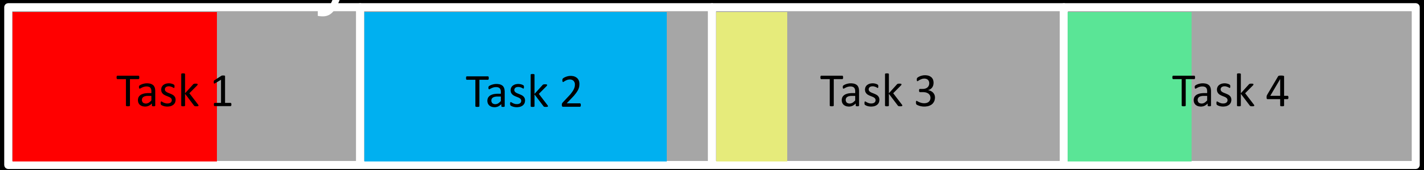
# Challenge #2

How to arbitrate memory across tasks running in parallel?

# Easy, static assignment!

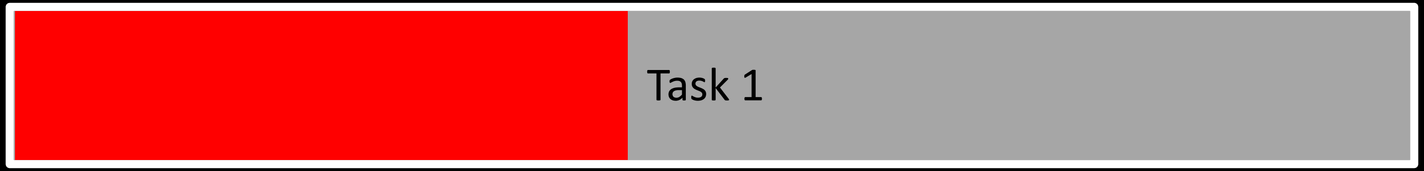
Worker machine has 4 cores

Each task gets ¼ of the total memory



# Dynamic Assignment

The share of each task depends on the number of actively running tasks



# Dynamic Assignment

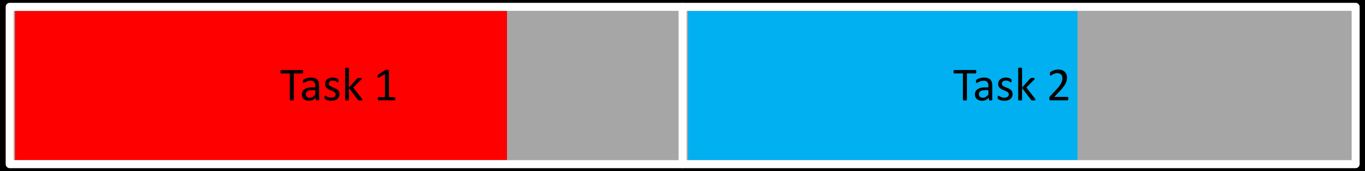
The share of each task depends on the number of actively running tasks

# Dynamic Assignment

Now another task comes along, the first task have to spill to free up memory

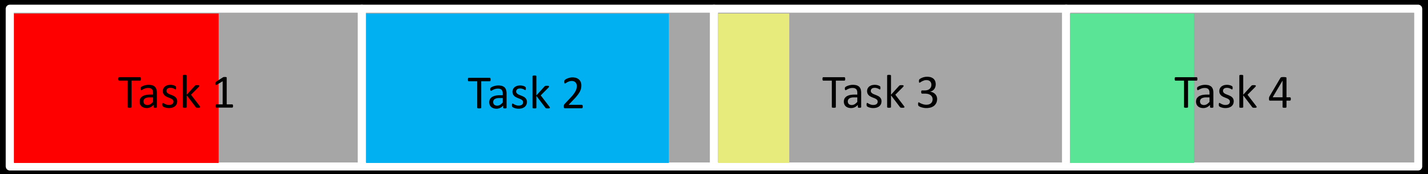
# Dynamic Assignment

Each task is now assigned ½ of the total memory



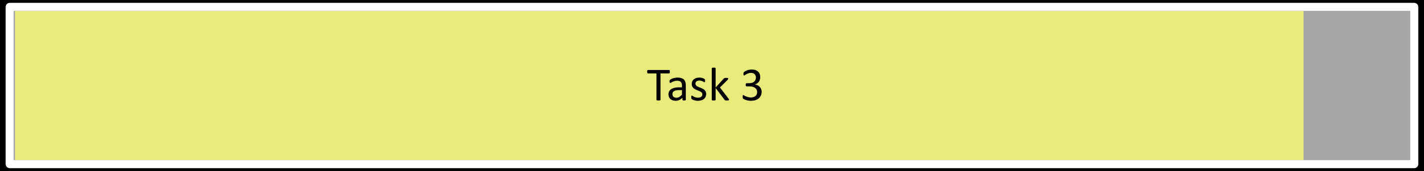
# Dynamic Assignment

Each task is now assigned ¼ of the total memory



# Dynamic Assignment

Last remaining task gets all the memory



# Static vs Dynamic Assignment

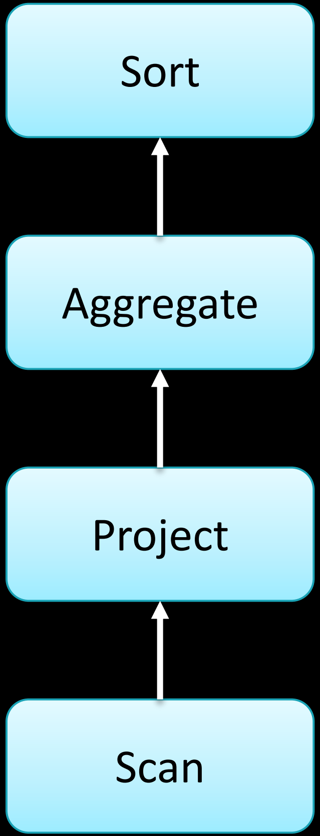
- Both are fair and starvation free
- Static Assignment is simpler
- Dynamic assignment handles stragglers better

# Challenge #3

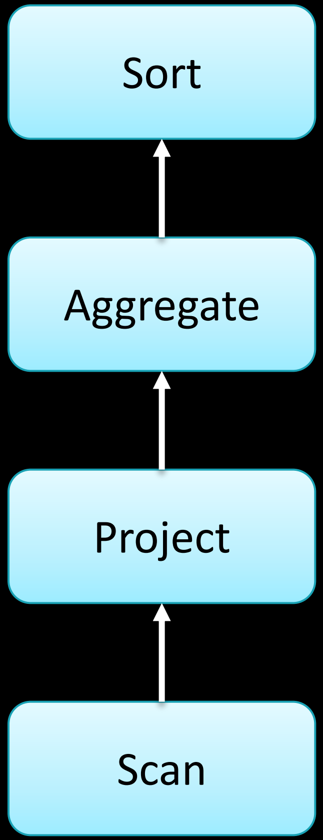
How to arbitrate memory across operators running within the same task?

```
SELECT age, AVG(height)
    FROM students
    GROUP BY age
    ORDER BY AVG(height)
```

```
students.groupBy("age")
    .avg("height")
    .orderBy("avg(height)")
    .collect()
```



The task has 6 pages of memory




databricks

Map { // age → (total, count)

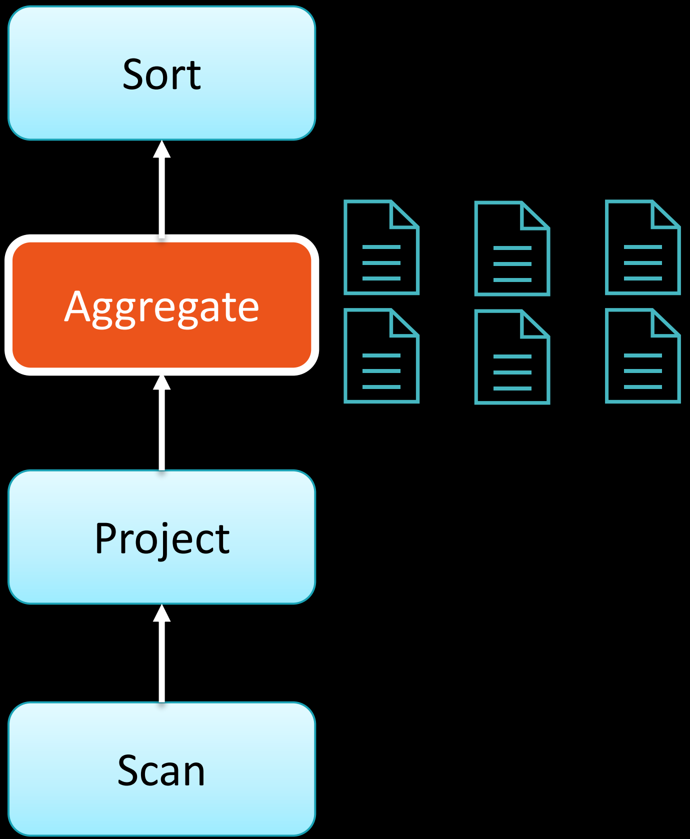
20 → (483, 3)

21 → (935, 5)

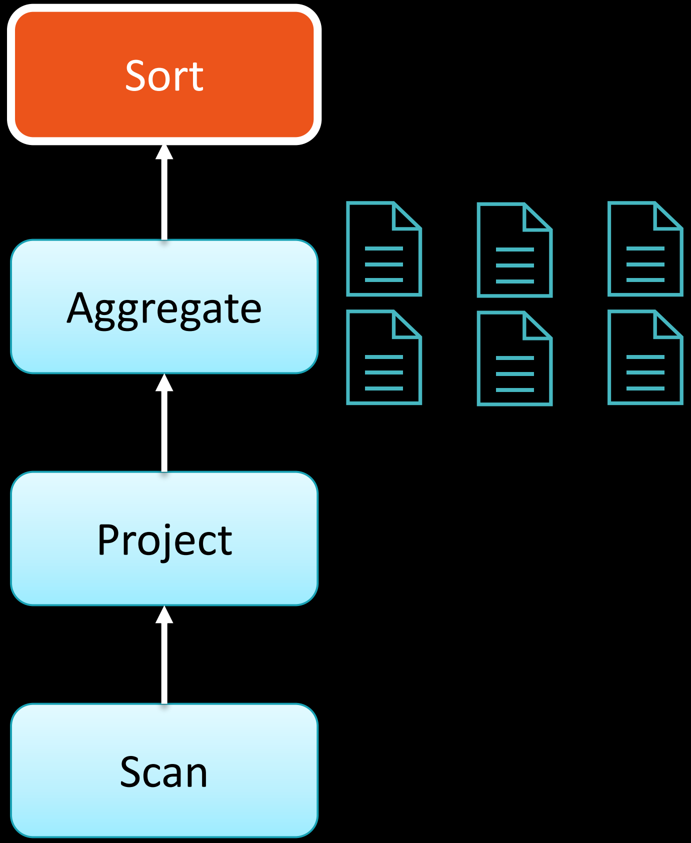
22 → (172, 1)

…

}



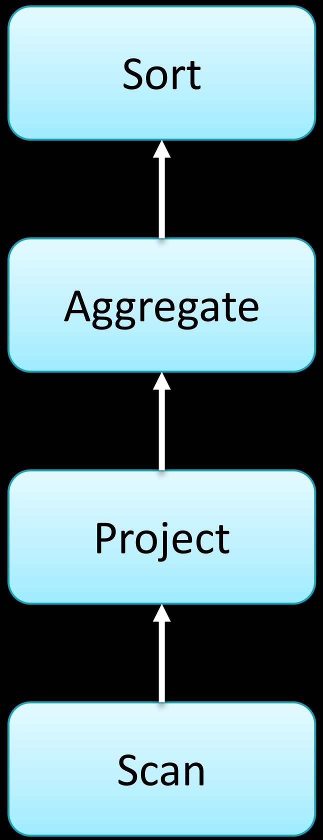
All 6 pages were used by **Aggregate**, leaving no memory for **Sort**!




databricks

# Solution #1

Reserve a page
for each operator

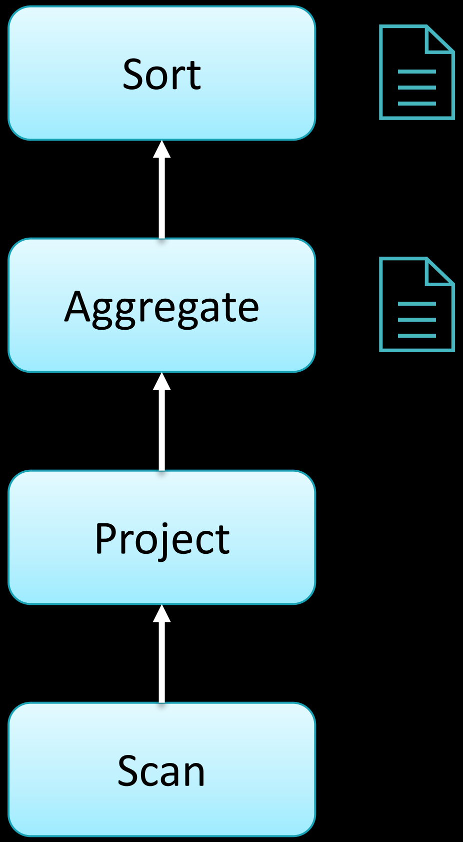


# Solution #1

Reserve a page
for each operator

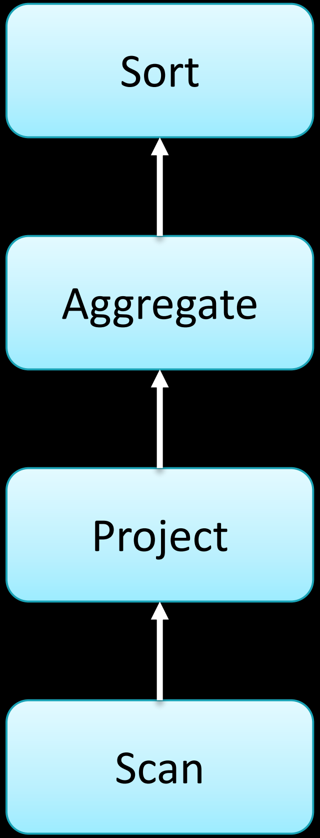
Starvation free, but still not fair…

What if there were more operators?



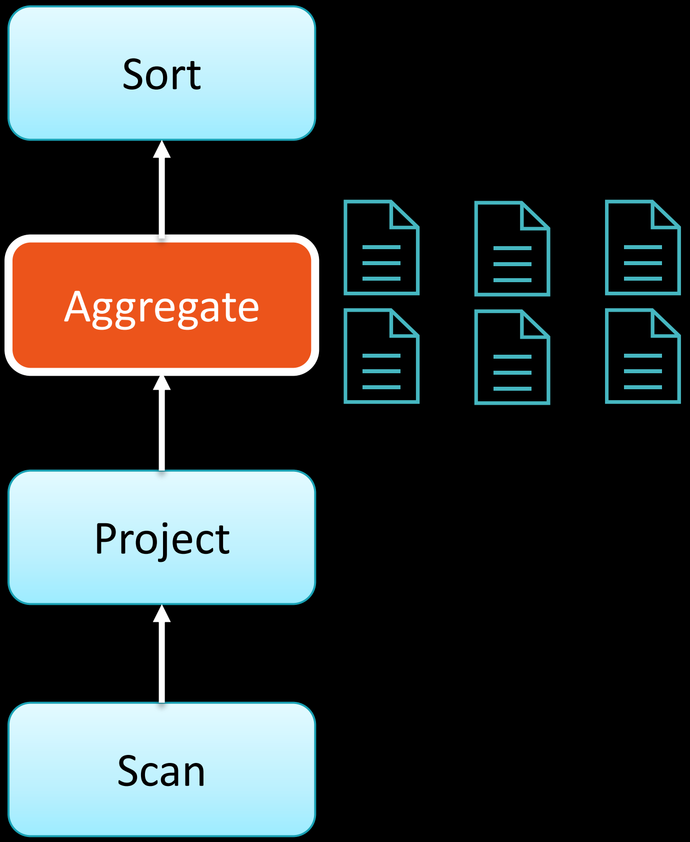
# Solution #2

Cooperative spilling



## Solution #2

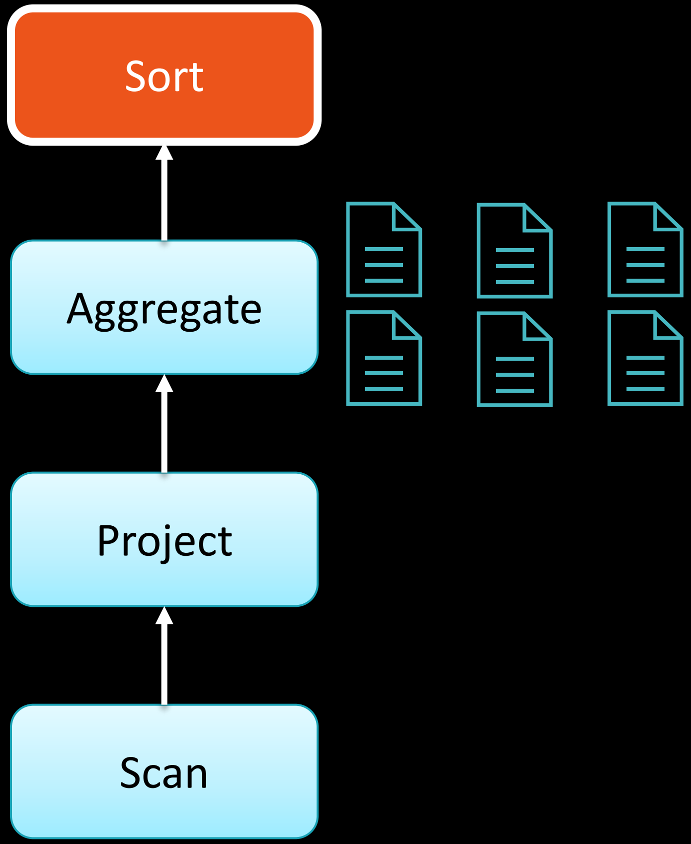
Cooperative spilling



# Solution #2

Cooperative spilling

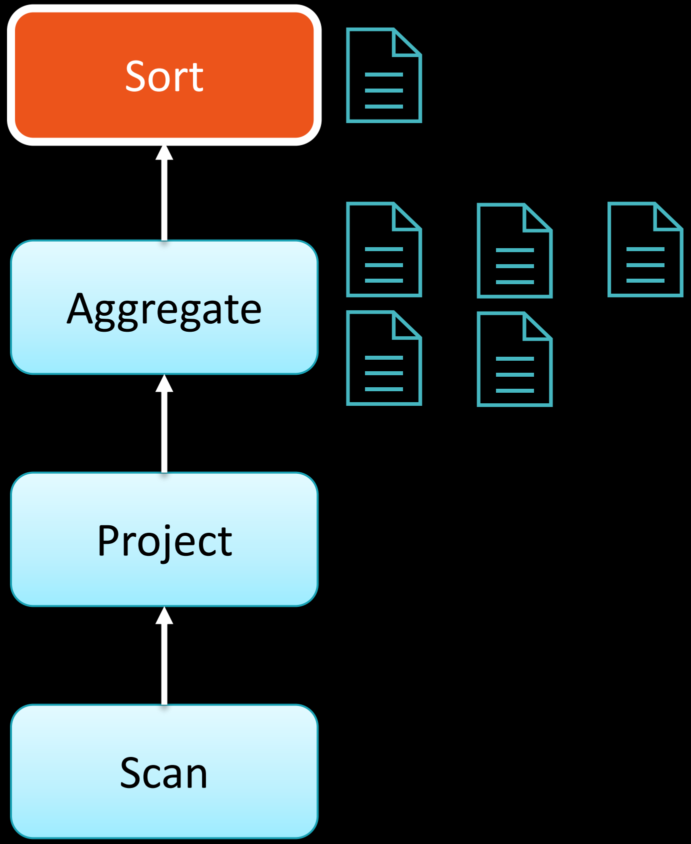
**Sort** forces **Aggregate** to spill a page to free memory



# Solution #2

Cooperative spilling

**Sort** needs more memory so it forces **Aggregate** to spill another page(and so on)

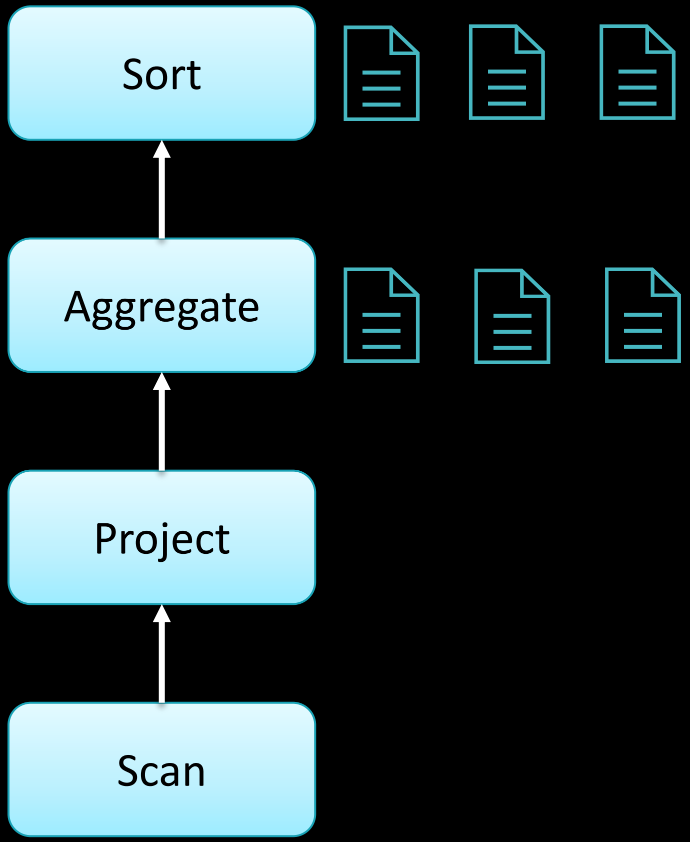


# Solution #2

Cooperative spilling

**Sort** finishes with 3 pages

**Aggregate** does not have to spill its remaining pages



# Recap: three source of contention

How to arbitrate memory …

- between execution and storage?
- across tasks running in parallel?
- across operators running with the same task?

*Instead of statically reserving memory in advance, deal with memory contention when it raises by forcing members to spill*

# How Spark keep data in memory

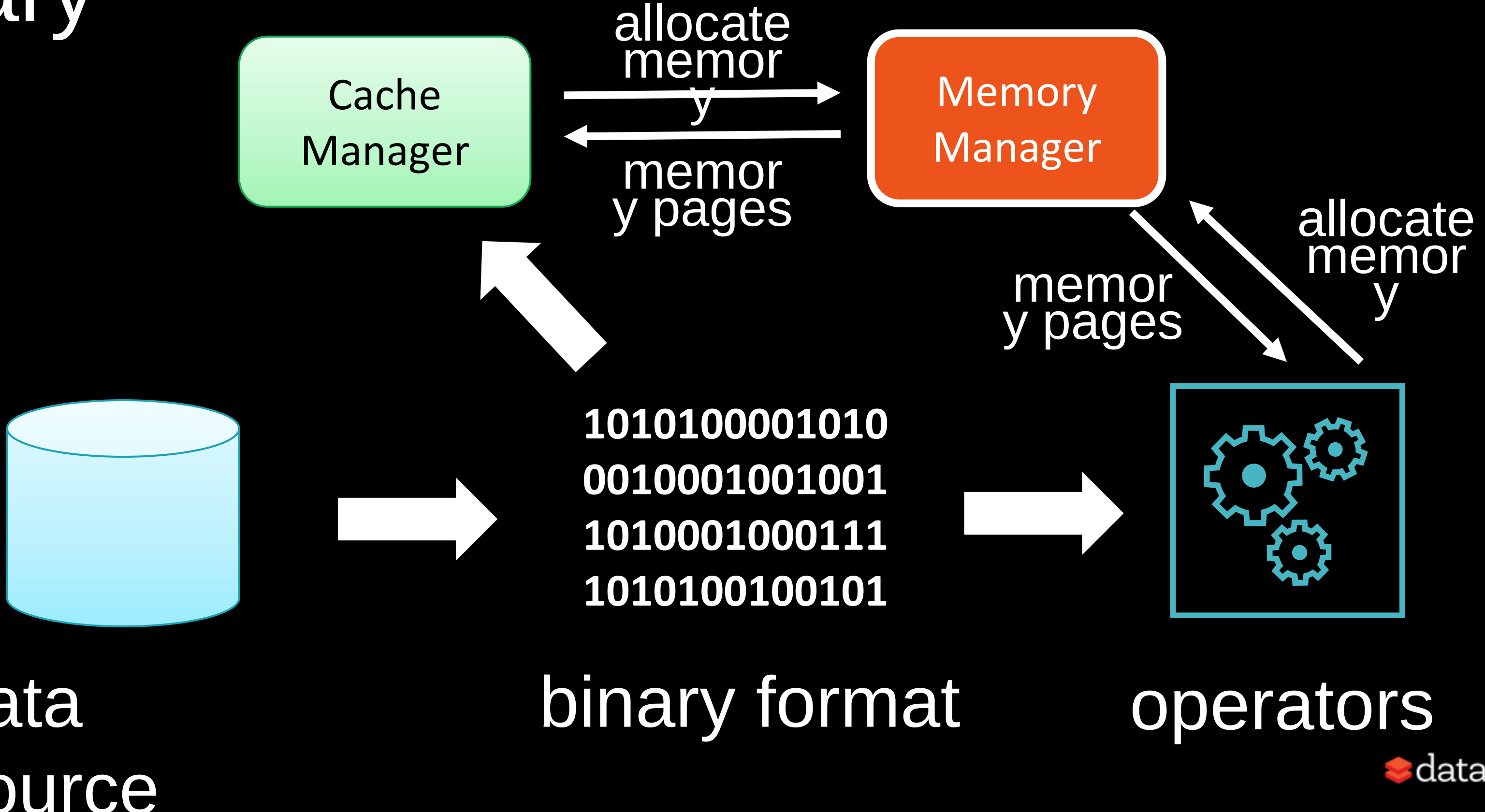
- Put data as objects on t[illegible] operate on these objects.

- Data caching is si[illegible] us[illegible] list to k[illegible] data objects.

# Data objects? No!

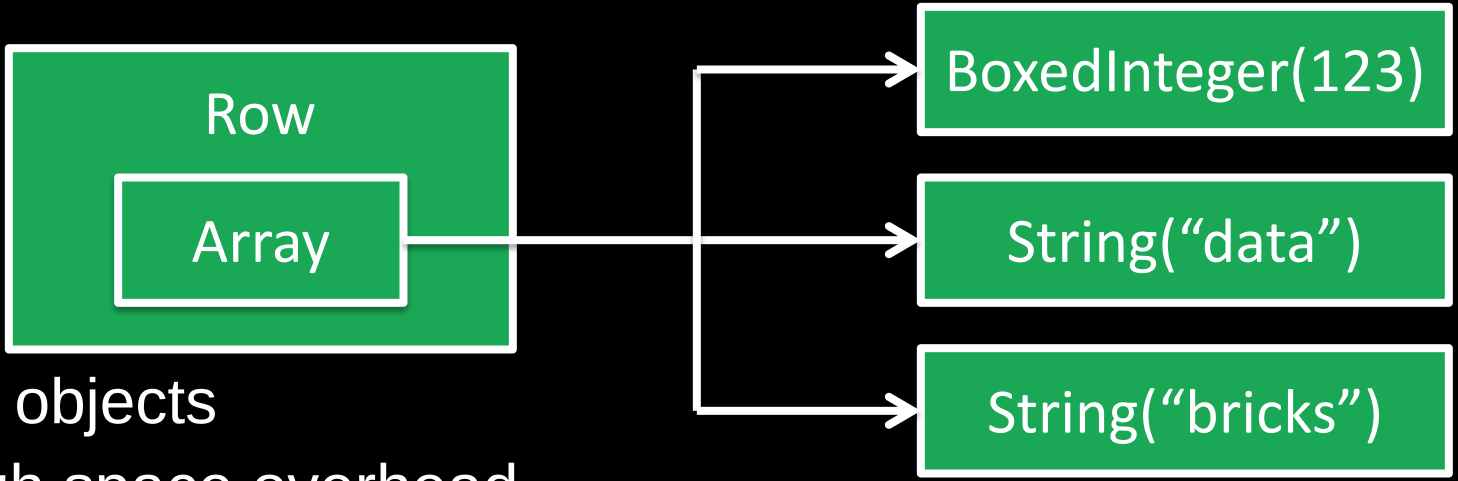
- It is hard to monitor and control the memory usage when we have a lot of objects.
- Garbage collection will be the killer.
- Java objects has notable space overhead.
- High serialization cost when transfer data inside cluster.

# Keep data as binary and operate on binary



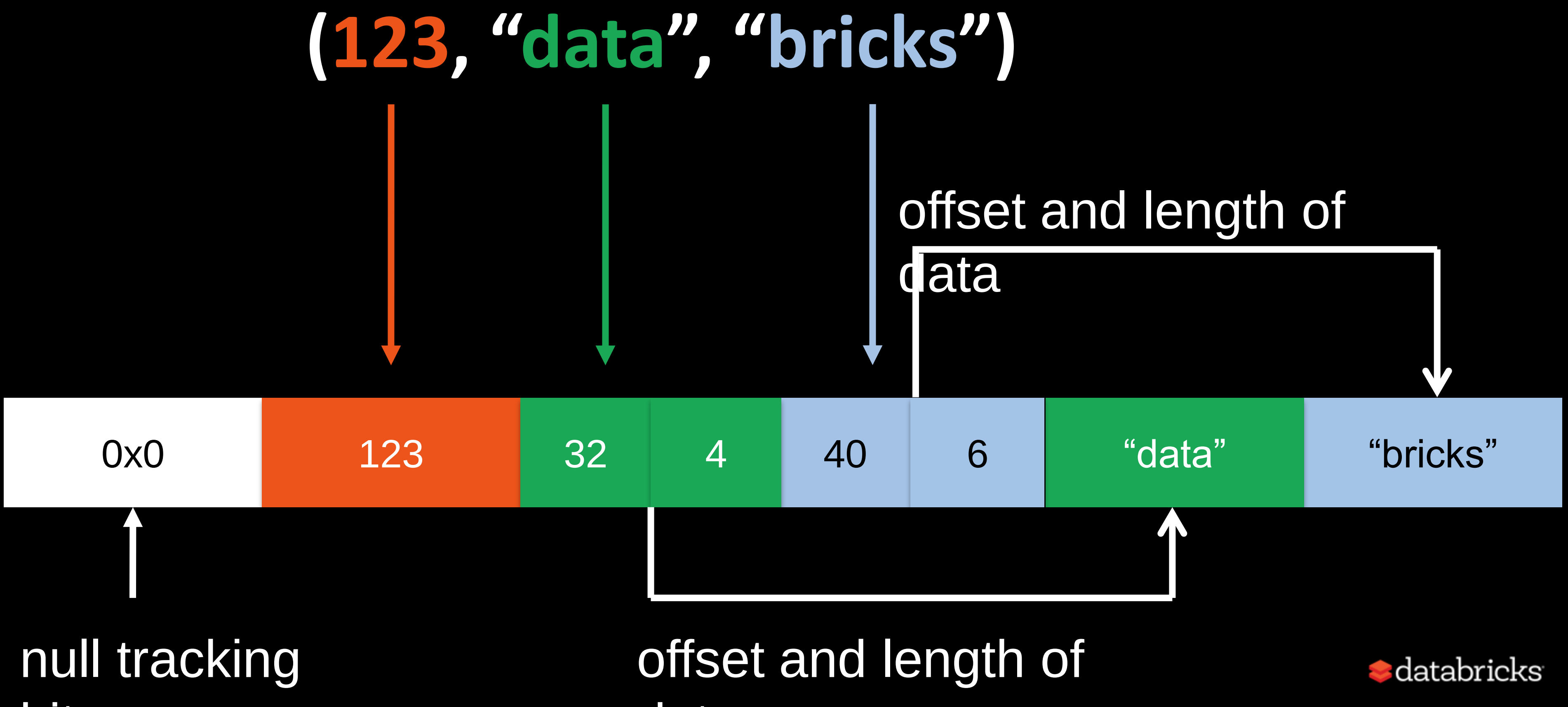

databricks

# Java Objects Based Row Format



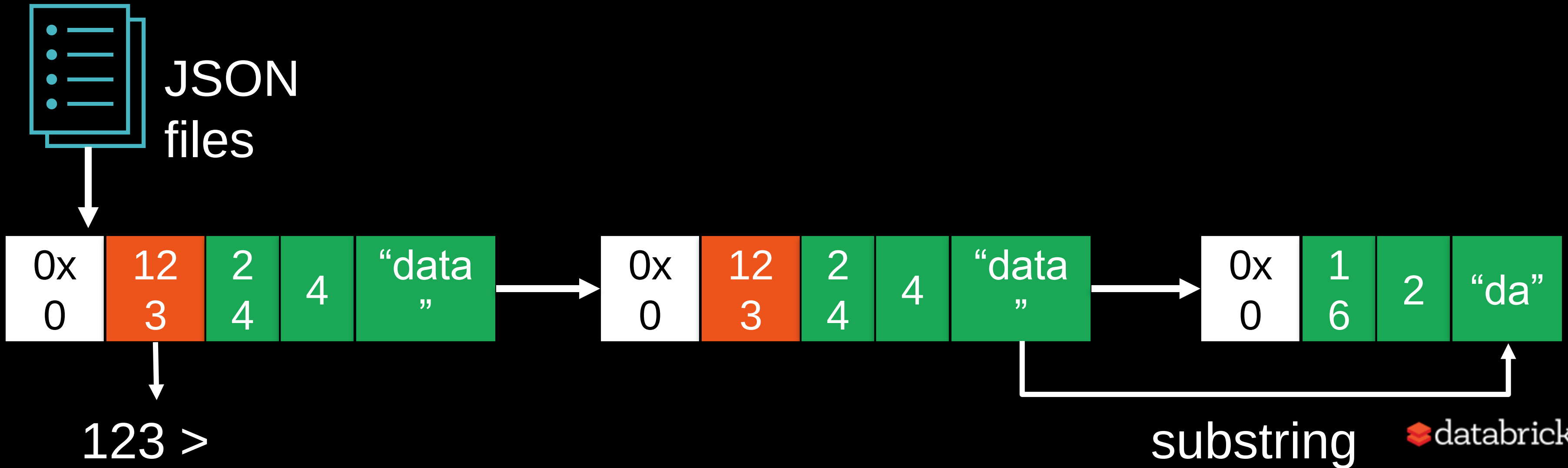

- 5+ objects
- high space overhead
- slow value accessing
- expensive hashCode()

# Efficient Binary Format



# Efficient Binary Format

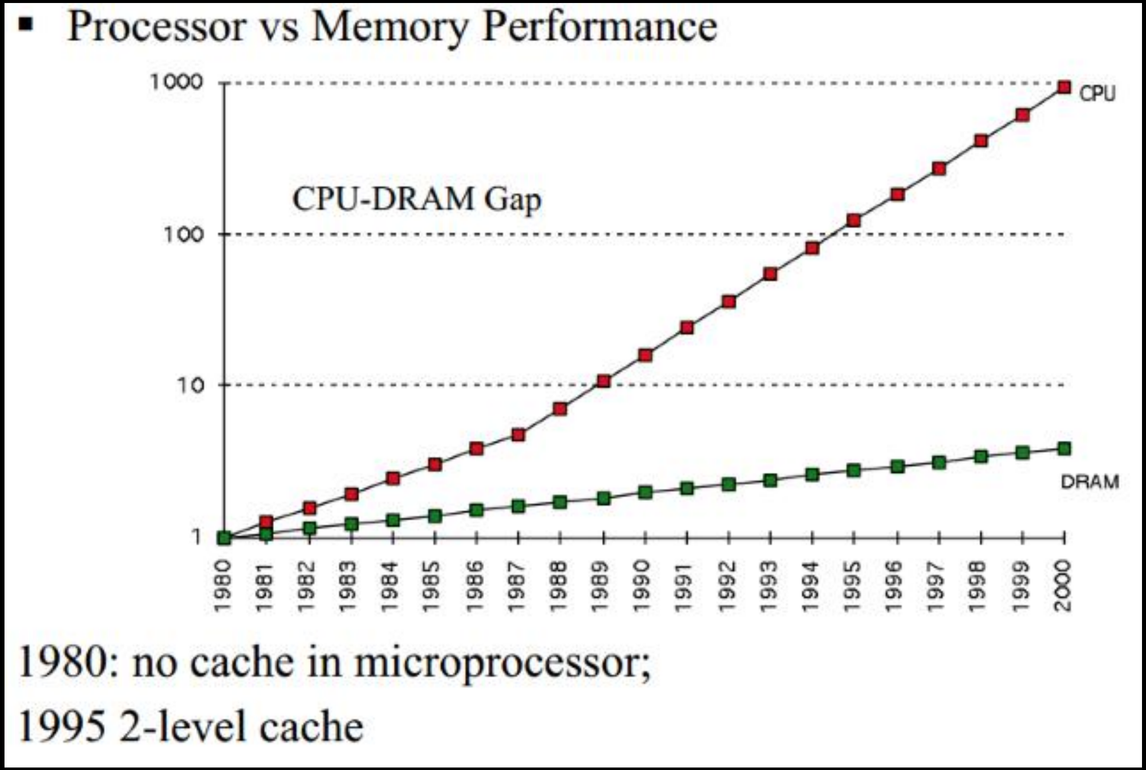
```
spark.read.schema("i int, j string").json("/tmp/x.json")
  .filter($"i" > 0)
  .select($"j".substr(0, 2))
```




databricks

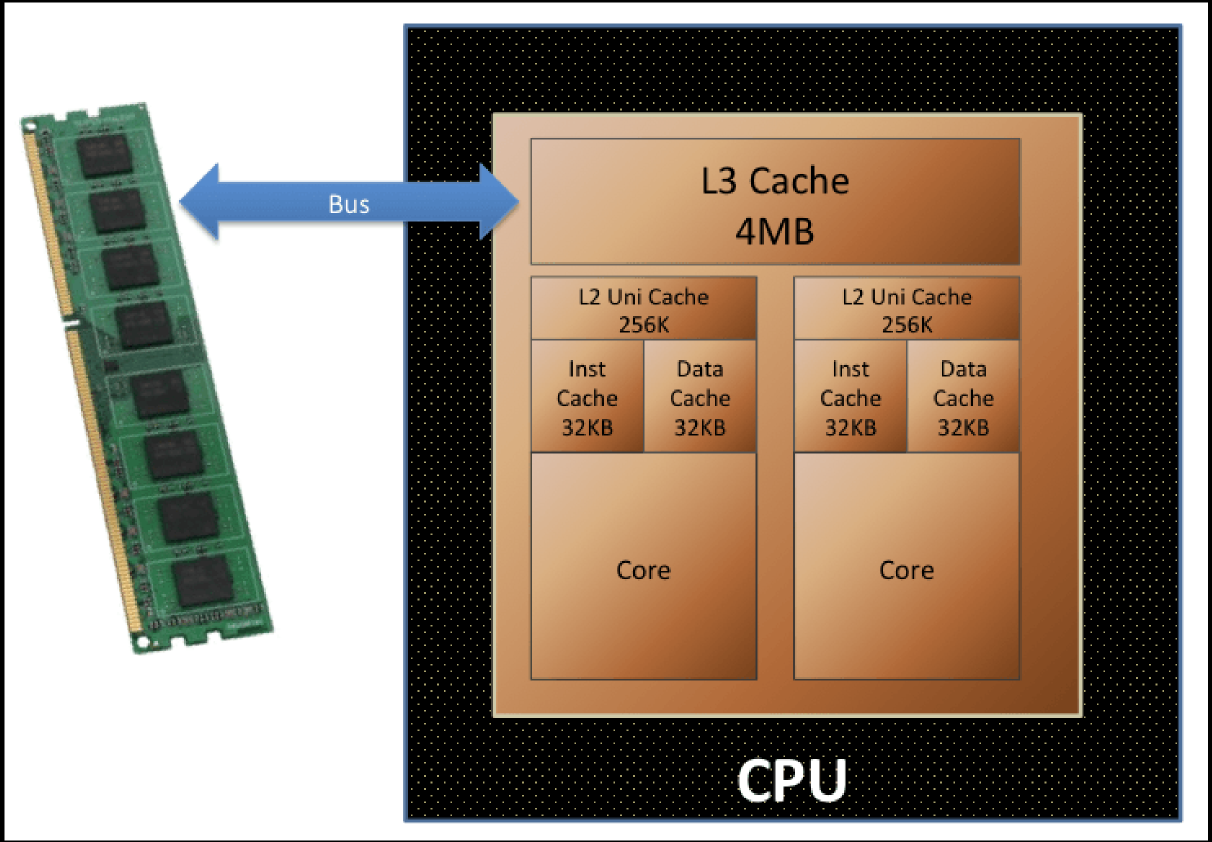
# How to process binary data more efficient?

# Understanding CPU Cache




Memory is becoming slower and slower than CPU, we should keep the frequently accessed data in CPU cache.

# Understanding CPU Cache




Pre-fetch data into CPU cache, with cache line boundary.

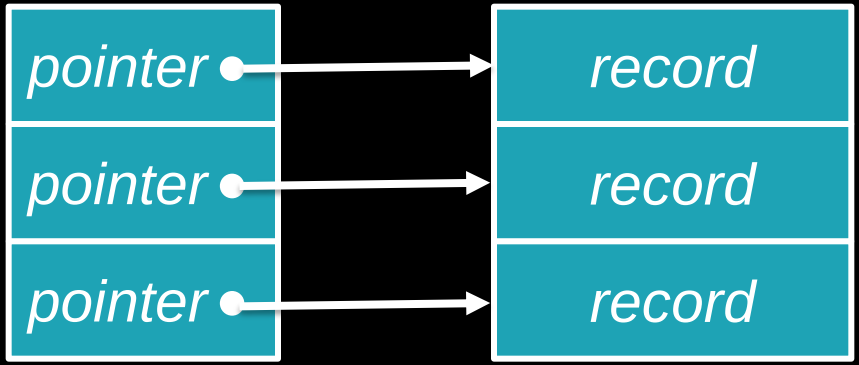
databricks

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

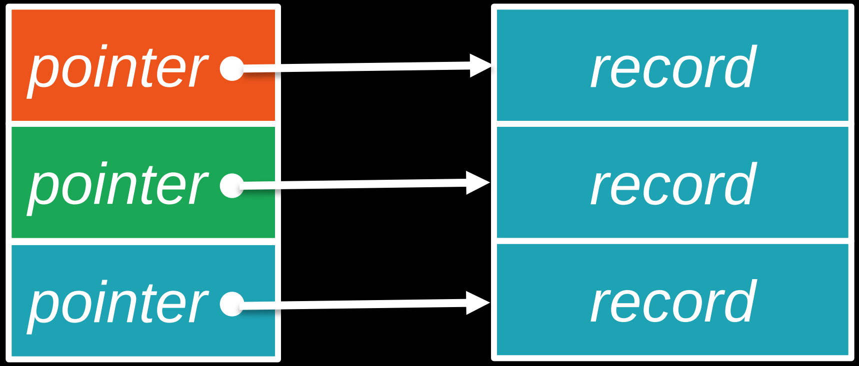
**Sort** and **Hash**!

databricks

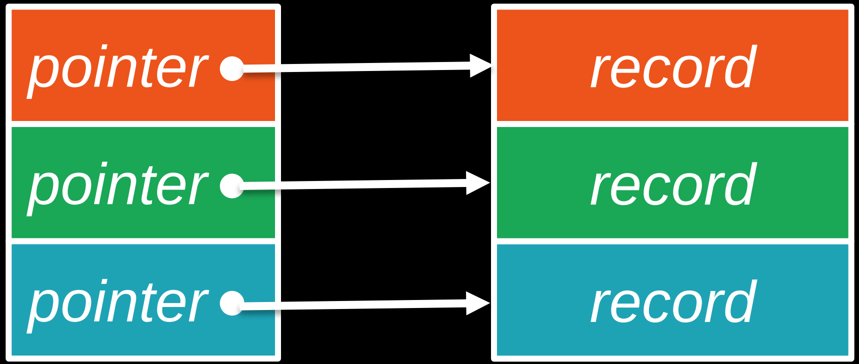
# Naive Sort



# Naive Sort

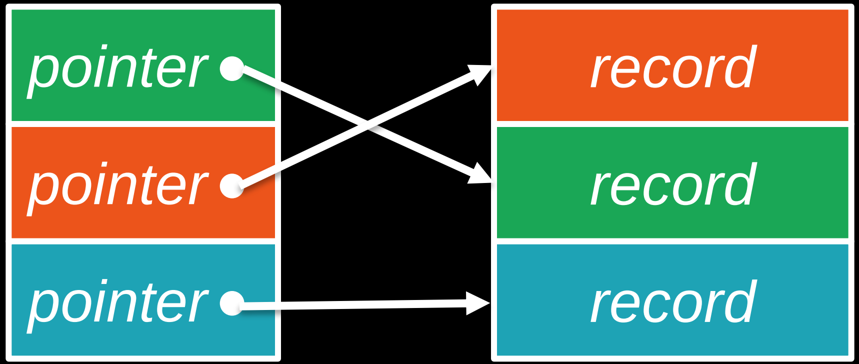


# Naive Sort




databricks

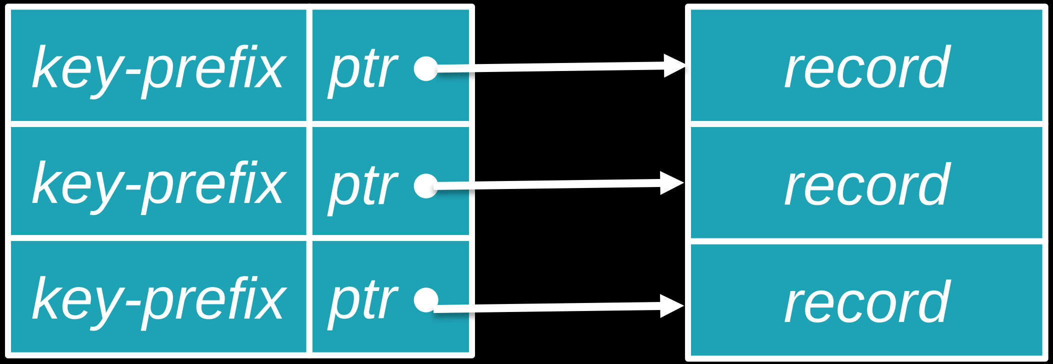
# Naive Sort



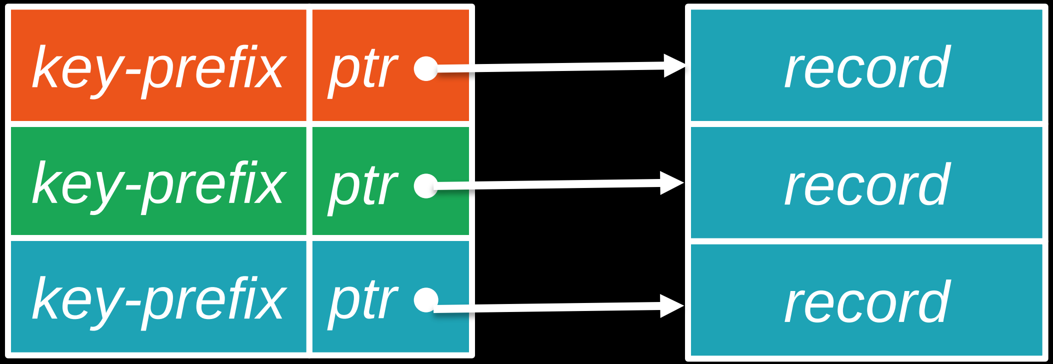
# Naive Sort

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

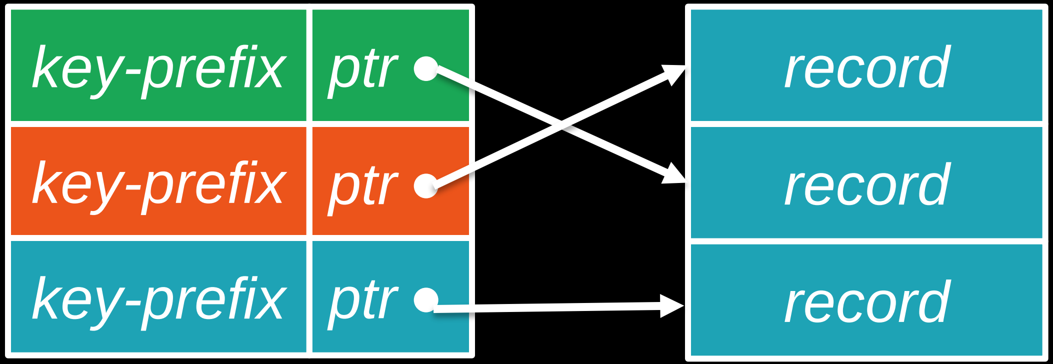


# Cache-aware Sort




databricks

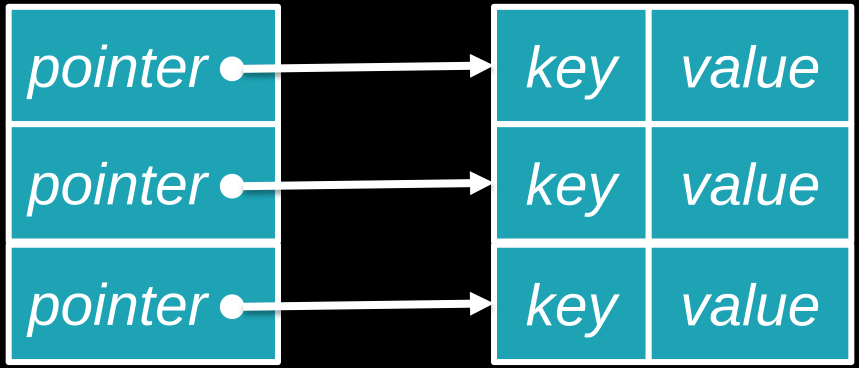
# Cache-aware Sort



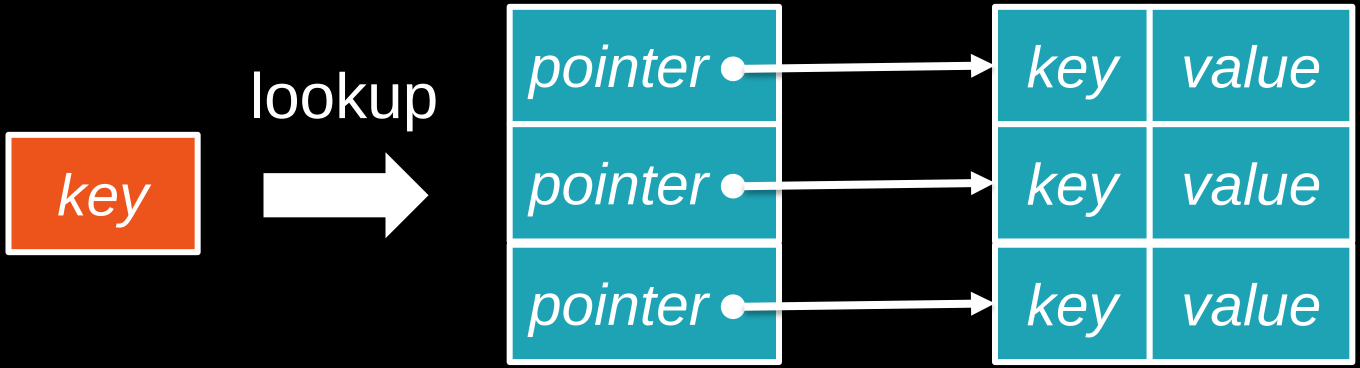
# Cache-aware Sort

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

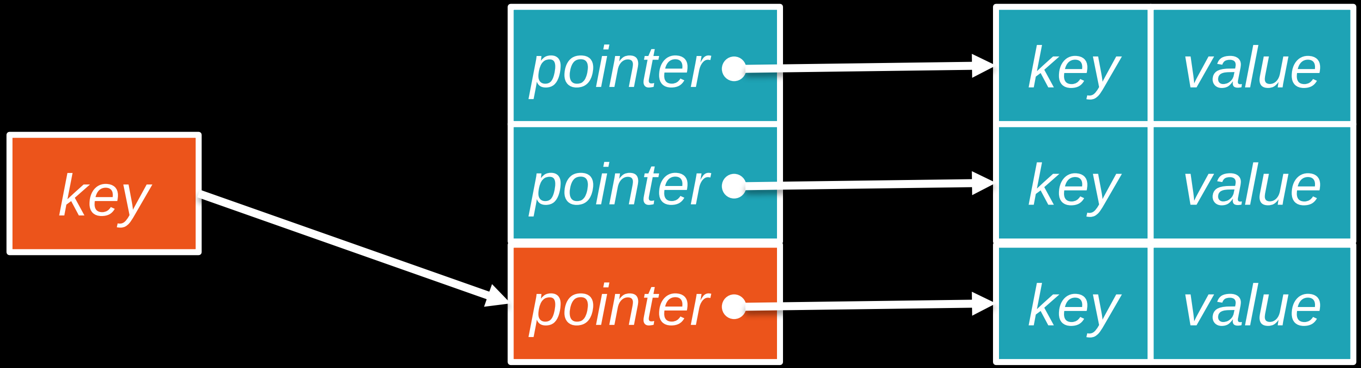
# Naive Hash Map



# Naive Hash Map

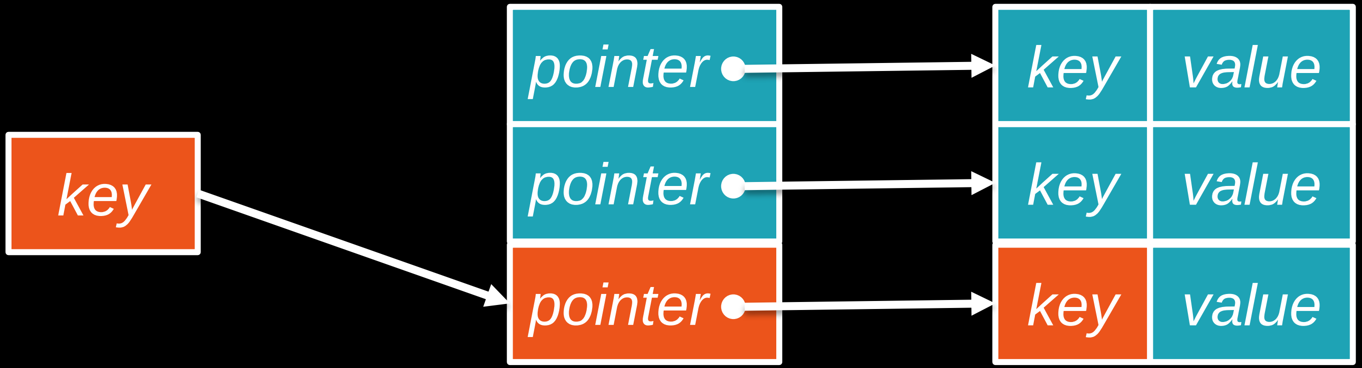


# Naive Hash Map



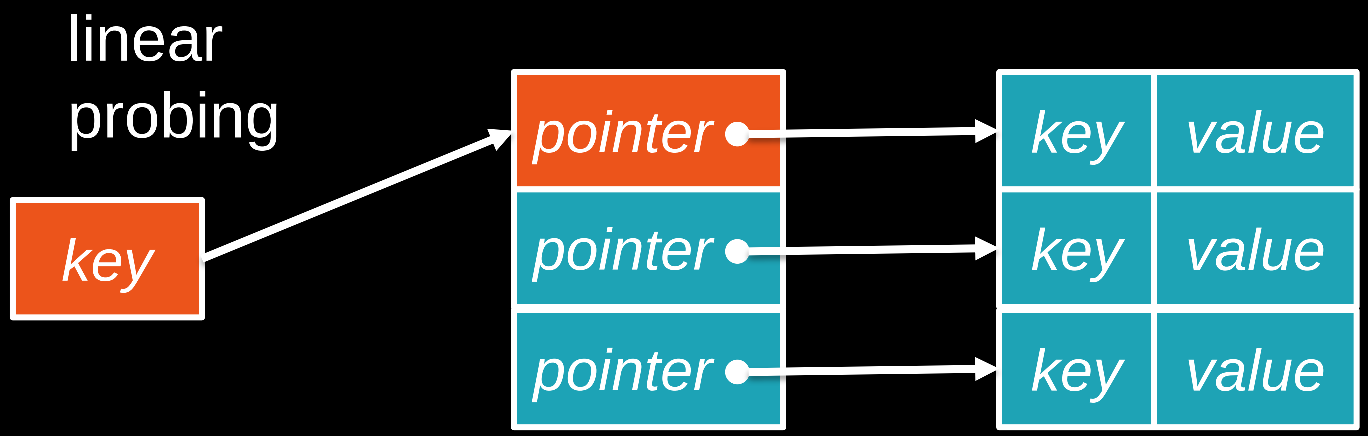

hash(key) % size

# Naive Hash Map

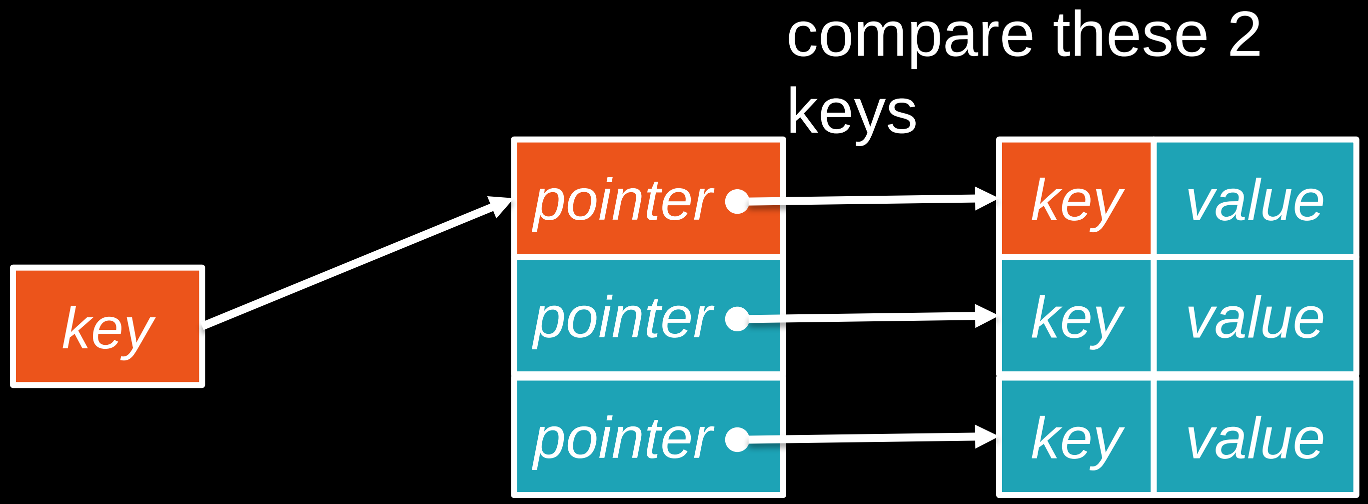



compare these 2 keys

# Naive Hash Map



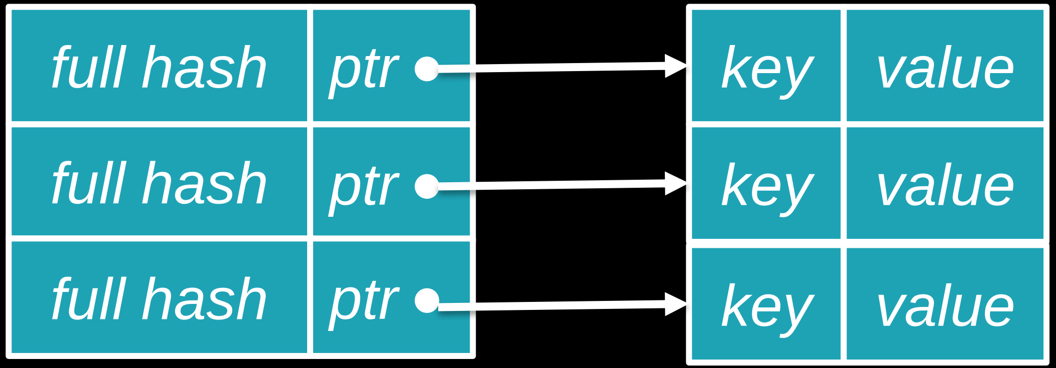
# Naive Hash Map



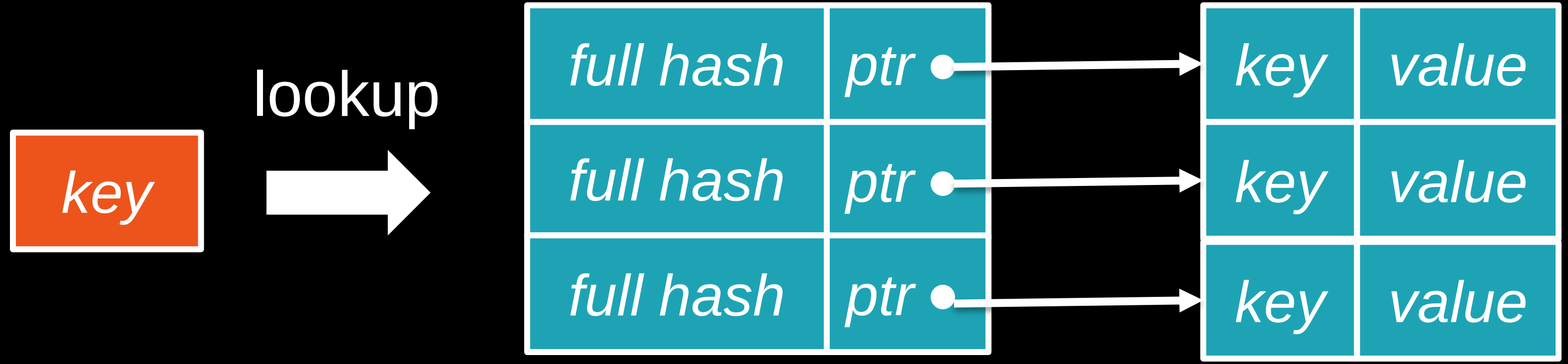
# Naive Hash Map

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

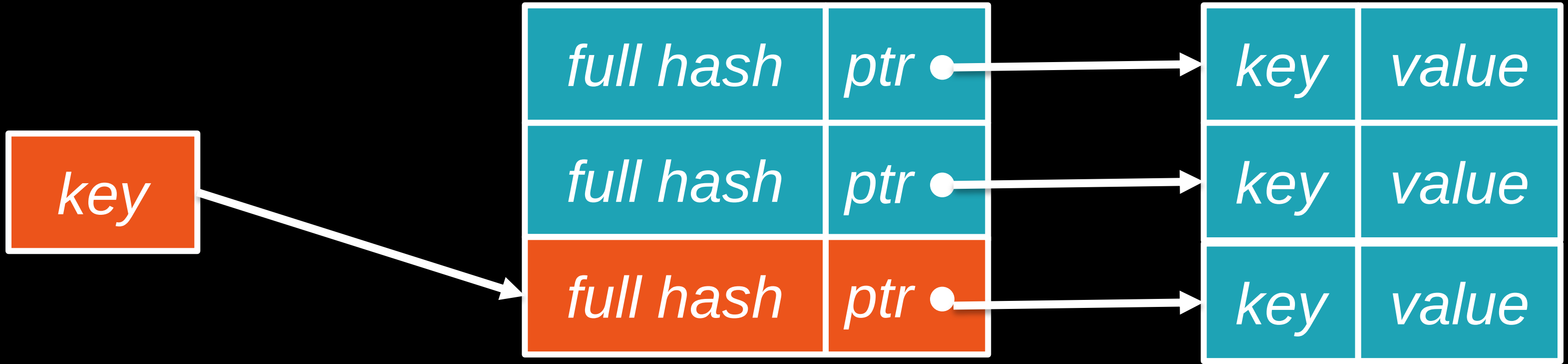
# Cache-aware Hash Map



# Cache-aware Hash Map



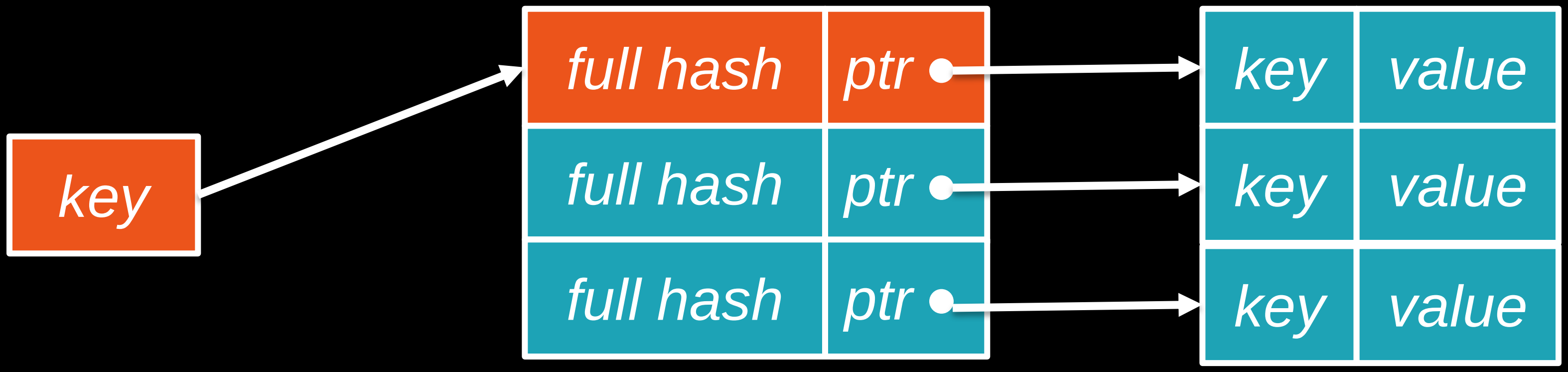
# Cache-aware Hash Map



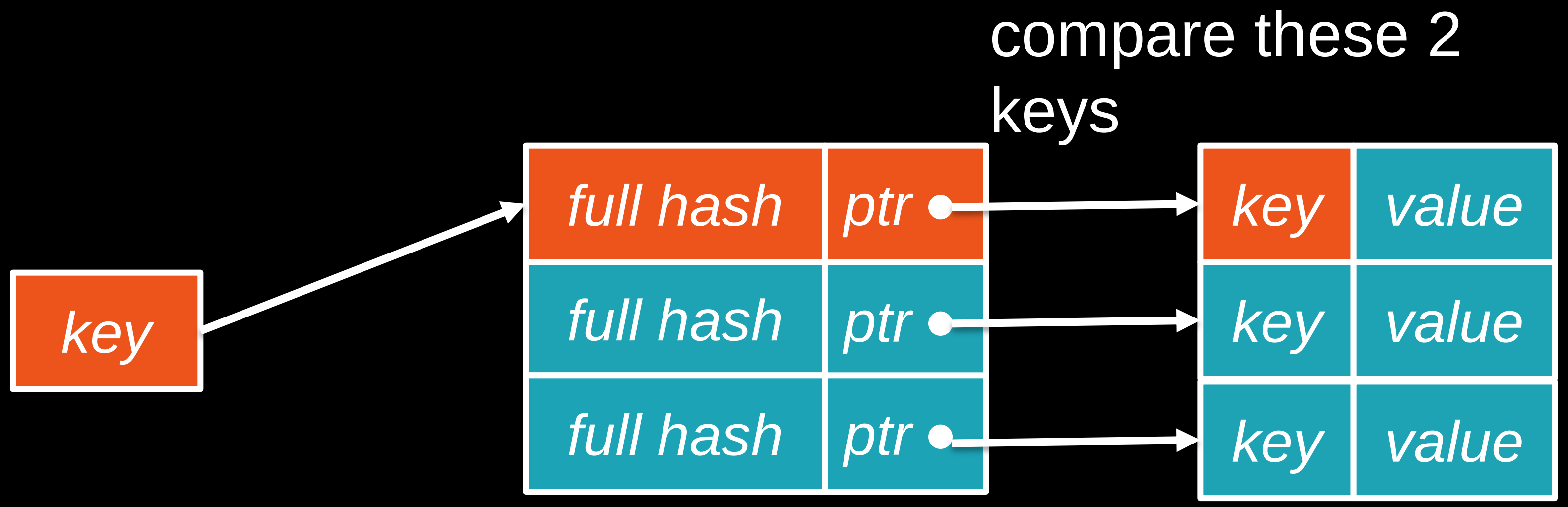

hash(key) % size, and
compare the full hash

# Cache-aware Hash Map

linear probing, and
compare the full hash



# Cache-aware Hash Map



# Cache-aware Hash Map

Each lookup mostly only needs one pointer dereference and key comparison(full hash collision is rare), and access data 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.*

# What’s next

- Standard binary format, may use Apache Arrow.
  - SPARK-19489
  - SPARK-13534

- Columnar execution engine.
  - SPARK-15687

# Thank You

# We Are Hiring!!!

Send your resume to
**wenchen@databricks.com**

Work at Hangzhou,
full time Spark developer!