UNIVERSITY OF WATERLOO

Batch Processing II
(v1.00)

Week 4: September 25, 2025

Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo

These slides are available at `https://lintool.github.io/cs451-2025f/`

Key Questions

In what ways does Spark improve over MapReduce?

How is distributed group by implemented efficiently at scale?

How do commutative and associative operations
contribute to efficient distributed execution?

How does partitioning contribute to efficient distributed execution?

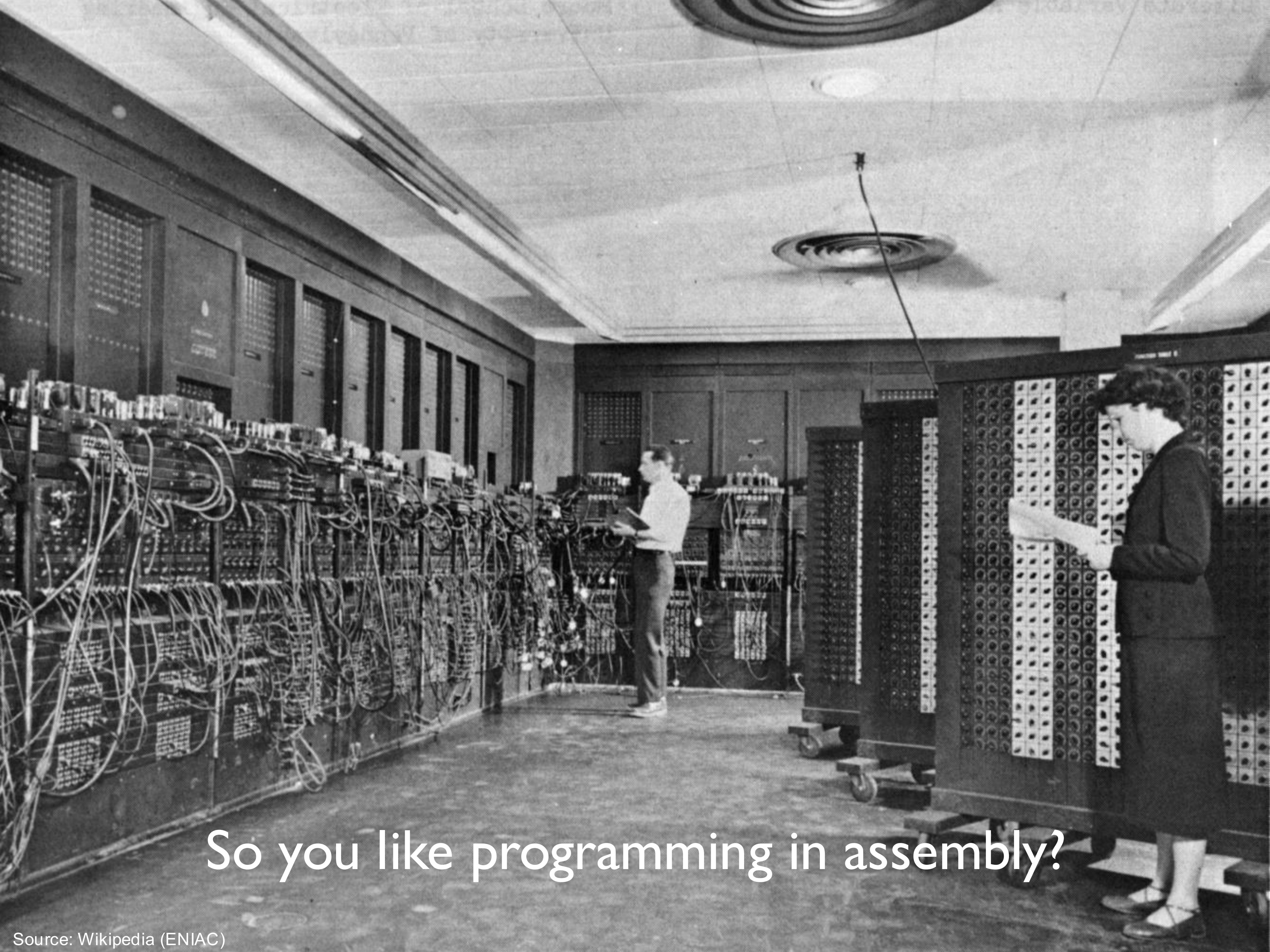
How do these concepts come together in efficient joins at scale?

The datacenter *is* the computer!

What's the instruction set?

map / reduce

Source: Google



Source: Wikipedia (ENIAC)

The datacenter *is* the computer!

What's the instruction set?

```
map / filter / flatMap ...
groupByKey / reduceByKey / join / cogroup ...
```

Source: Google

That's it.

MapReduce

```
results = records.map(...)
                 .reduce(...)

results2 = results1.map(...)
                   .reduce(...)
```

Spark

```
results = rdd.foo(...)
             .bar(...)
             .baz(...)
```

More? Why do you care?

The essence of abstraction is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context.

– computer scientist John V. Guttag

1. All abstractions are *leaky*
2. Important to develop intuitions
3. What do you want to be?
4. Curiosity

You don't have to be an engineer to be be a racing driver,
but you do have to have mechanical sympathy

– Formula One driver Jackie Stewart

Spark Stack

DataFrames / DataSets

RDDs

Physical Operators

Who lives where?

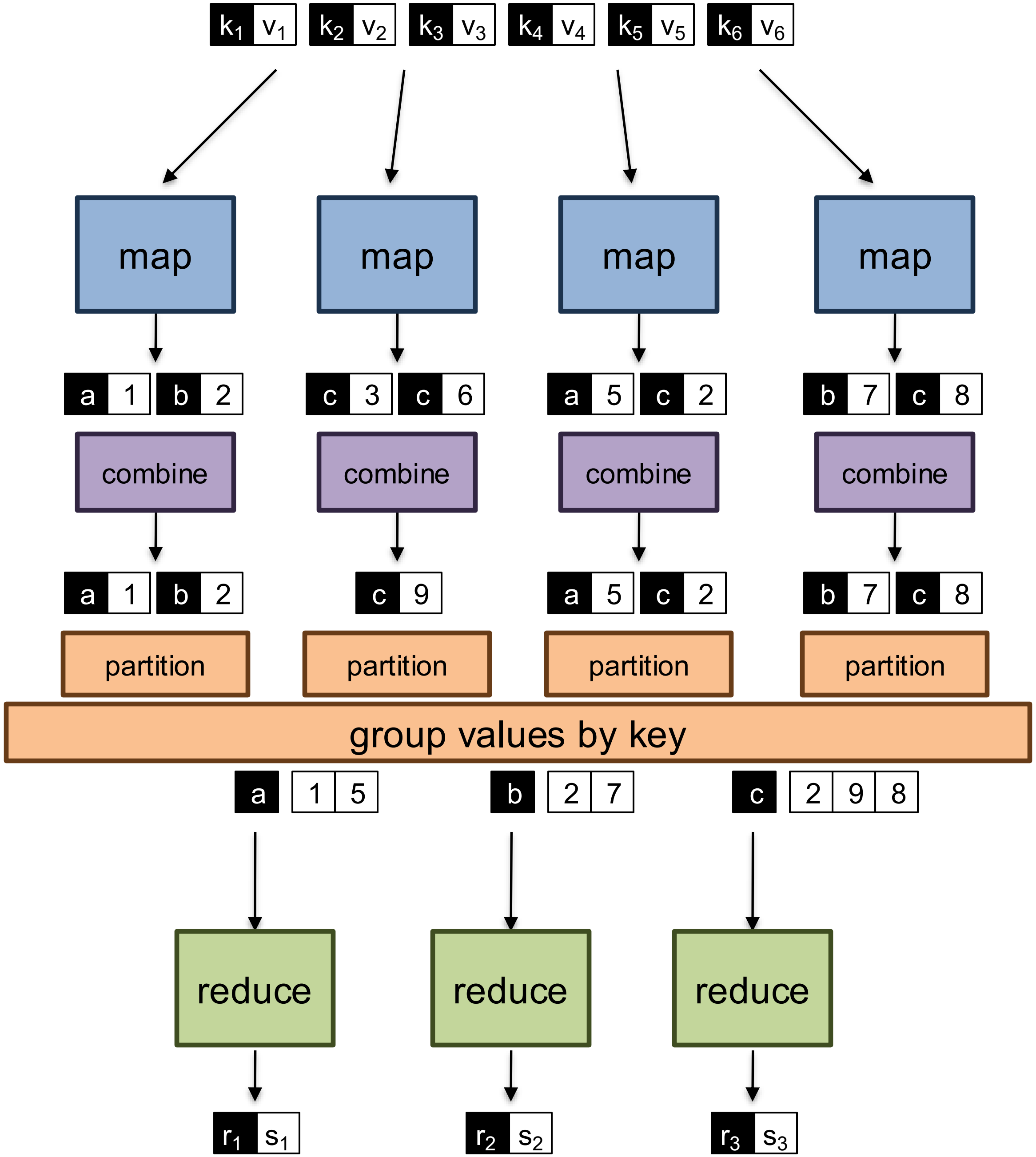
Technical Deep Dives

Implementation of distributed group by

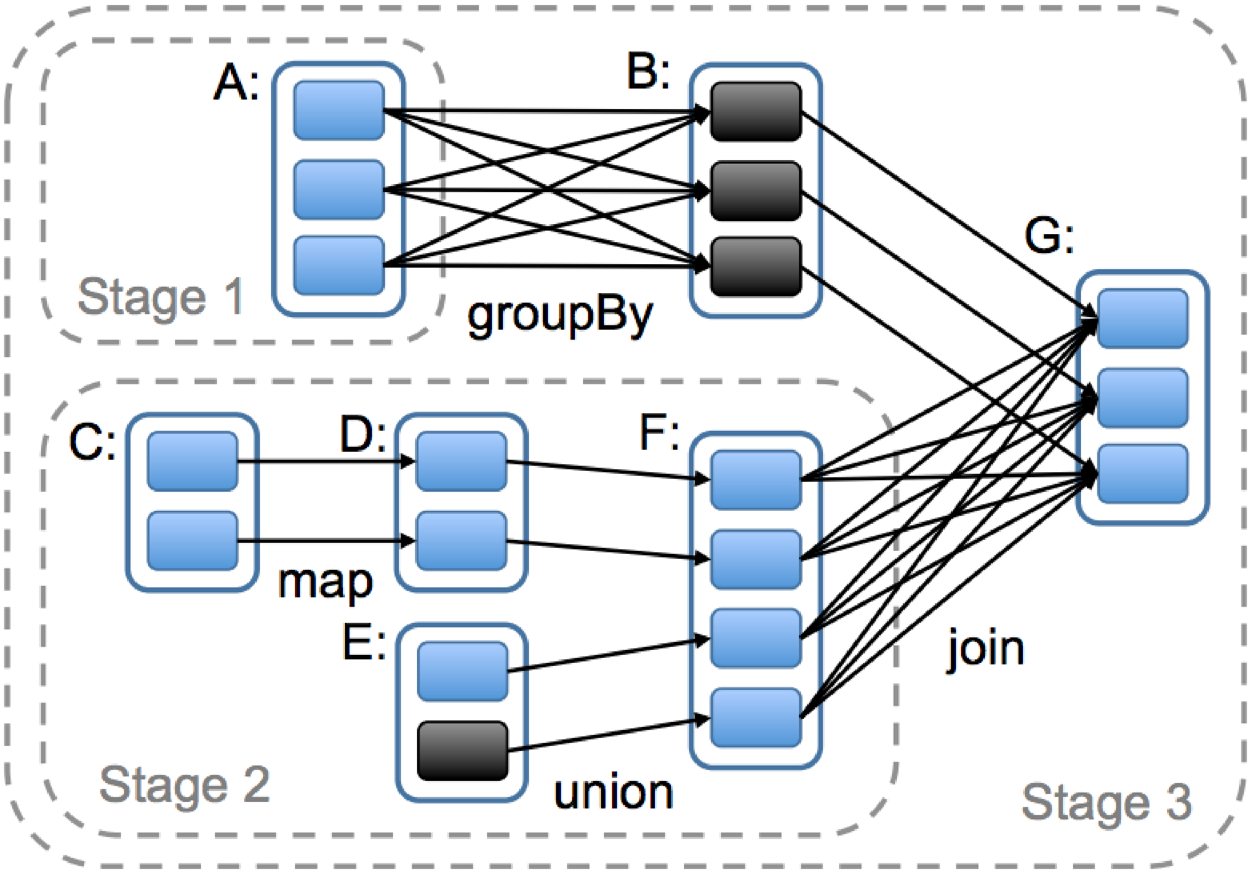
… implications for algorithm design

The importance of partitioning

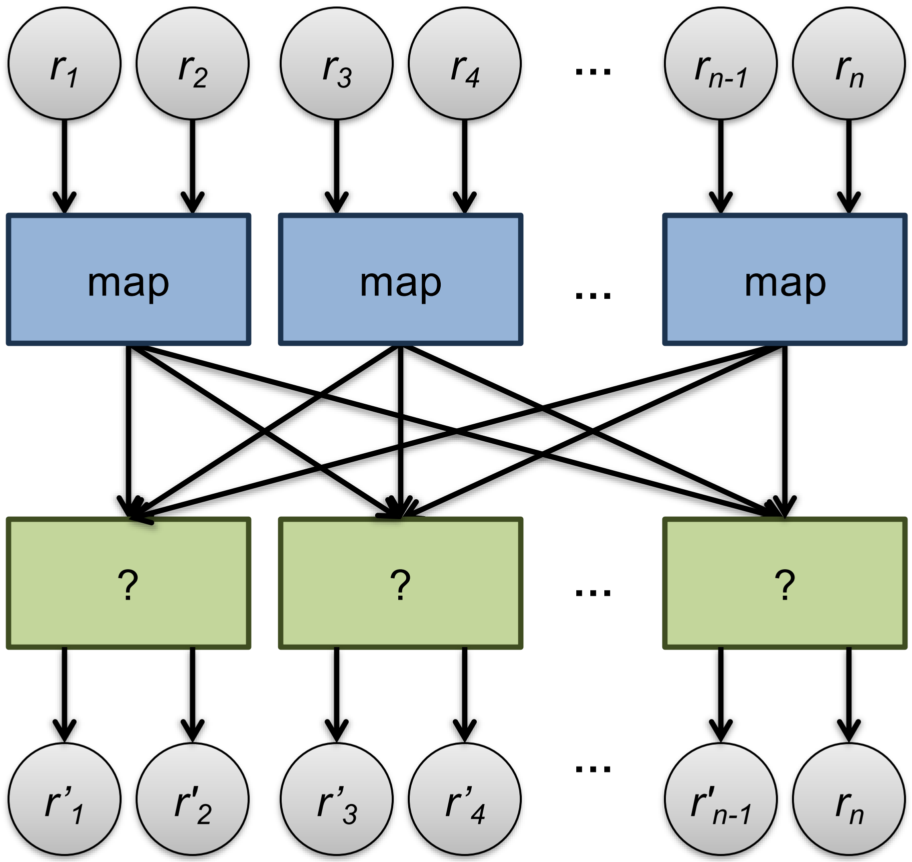
Case study in relational joins


k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
group values by key
a 1 5
b 2 7
c 2 9 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

Spark Execution Plan



We need communication



How is this implemented?

Why do you care?

1. All abstractions are *leaky*
2. Important to develop intuitions
3. What do you want to be?
4. Curiosity

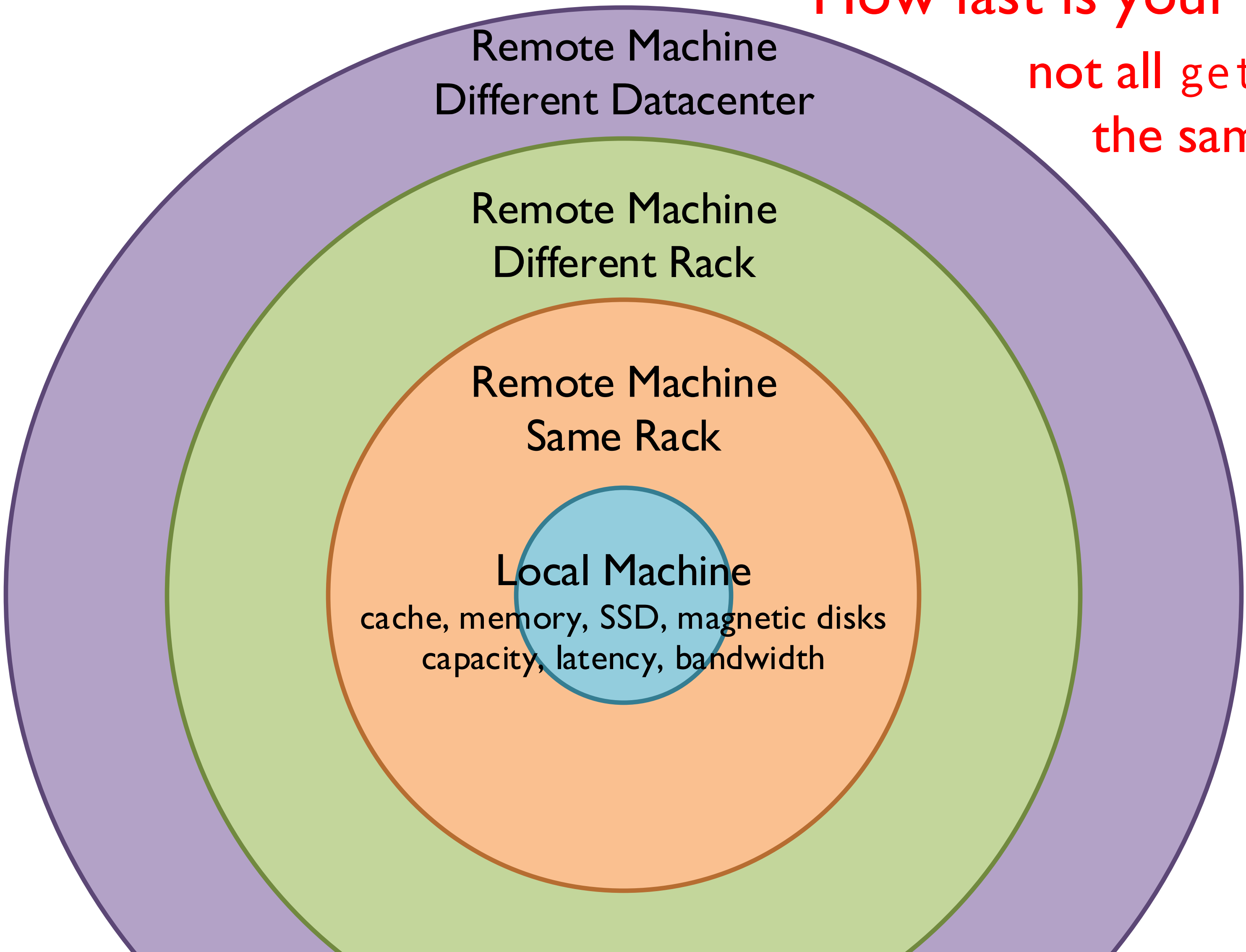
All abstractions are *leaky*

example: `get(x)`

Storage Hierarchy

How fast is your get?

not all `gets` are the same!



All abstractions are *leaky*

example: `get(x)`

You shouldn't need to care about locality, but you kinda need to…

Here's another one…

Seek vs. Scans

Consider a 1 TB database with 100 byte records

We want to update 1 percent of the records

Scenario 1: Mutate each record

Each update takes ~30 ms (seek, read, write)

10^8 updates = ~35 days

Scenario 2: Rewrite all records

Assume 100 MB/s throughput

Time = 5.6 hours(!)

Lesson? Random access is expensive!

Source: Ted Dunning, on Hadoop mailing list

What's the point?

At scale, sorting is the most efficient implementation of distributed group by!

Sort/Shuffle/Merge MapReduce

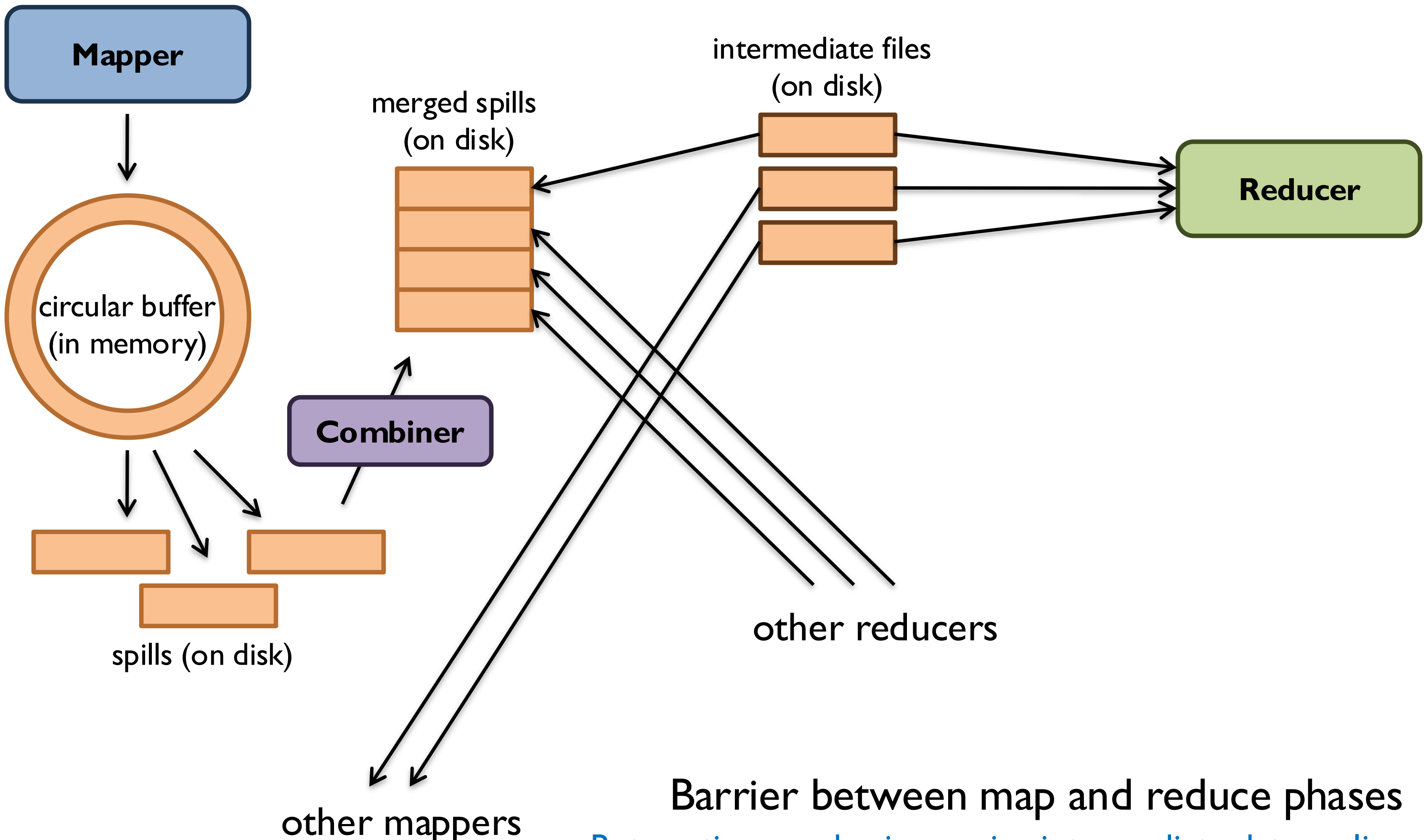
Map side

Map outputs are buffered in memory in a circular buffer
When buffer reaches threshold, contents are "spilled" to disk
Spills are merged into a single, partitioned file (sorted within each partition)
Combiner runs during the merges

Reduce side

Map outputs are copied over to the reducer machines
"Sort" is a multi-pass merge of map outputs (happens in memory and on disk)
Final merge pass goes directly into reducer

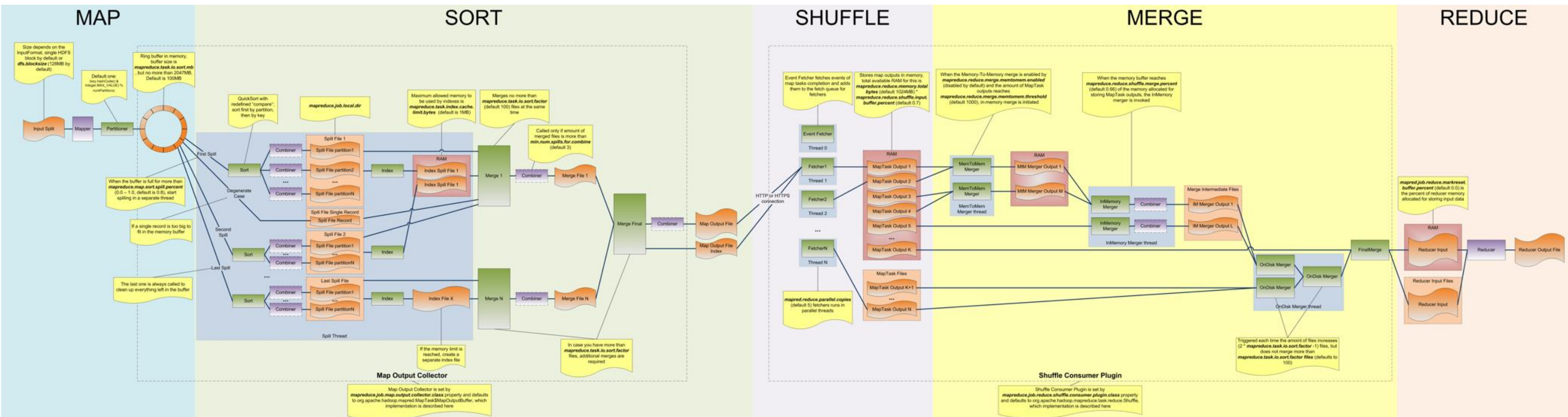
Sort/Shuffle/Merge MapReduce



Barrier between map and reduce phases

But runtime can begin copying intermediate data earlier

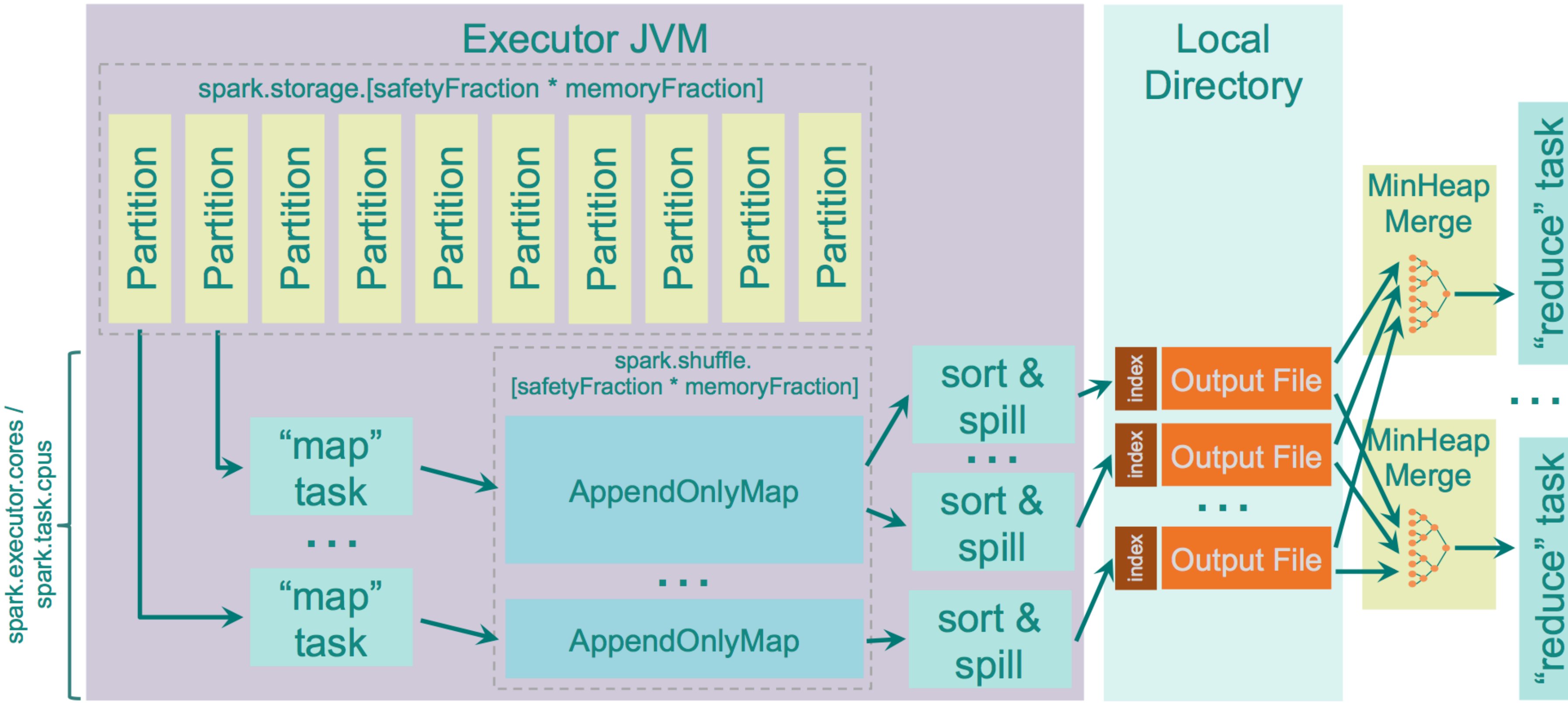
Sort/Shuffle/Merge MapReduce



Source: https://0x0fff.com/hadoop-mapreduce-comprehensive-description/

Spark Shuffle Implementations

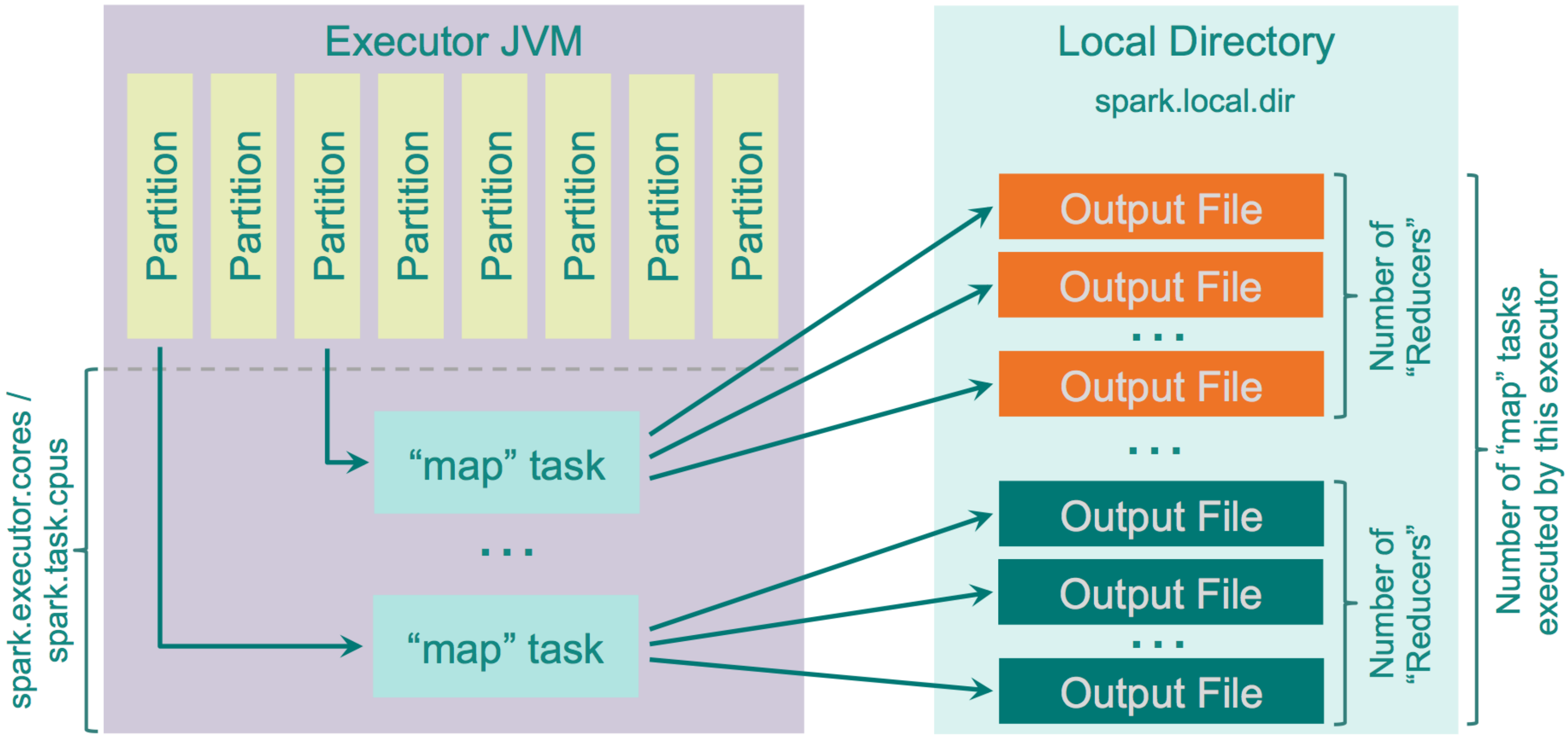
Sort shuffle



Source: http://0x0fff.com/spark-architecture-shuffle/

Spark Shuffle Implementations

Hash shuffle



Source: http://0x0fff.com/spark-architecture-shuffle/

What’s the point?

Technical Deep Dives

Implementation of distributed group by

… implications for algorithm design

The importance of partitioning

Case study in relational joins

Super powers:
Associativity and Commutativity!

The Power of Associativity

You can put parentheses wherever you want!

$$\left(v_1 \oplus v_2 \oplus v_3 \right) \oplus \left(v_4 \oplus v_5 \oplus v_6 \oplus v_7 \right) \oplus \left(v_8 \oplus v_9 \right)$$

$$\left(v_1 \oplus v_2 \right) \oplus \left(v_3 \oplus v_4 \oplus v_5 \right) \oplus \left(v_6 \oplus v_7 \oplus v_8 \oplus v_9 \right)$$

$$\left(v_1 \oplus v_2 \oplus \left(v_3 \oplus v_4 \oplus v_5 \right) \right) \oplus \left(v_6 \oplus v_7 \oplus v_8 \oplus v_9 \right)$$

Credit to Oscar Boykin for the idea behind these slides

The Power of Commutativity

You can swap order of operands however you want!

$$\left[v_1 \oplus v_2 \oplus v_3 \right] \oplus \left[v_4 \oplus v_5 \oplus v_6 \oplus v_7 \right] \oplus \left[v_8 \oplus v_9 \right]$$

$$\left[v_4 \oplus v_5 \oplus v_6 \oplus v_7 \right] \oplus \left[v_1 \oplus v_2 \oplus v_3 \right] \oplus \left[v_8 \oplus v_9 \right]$$

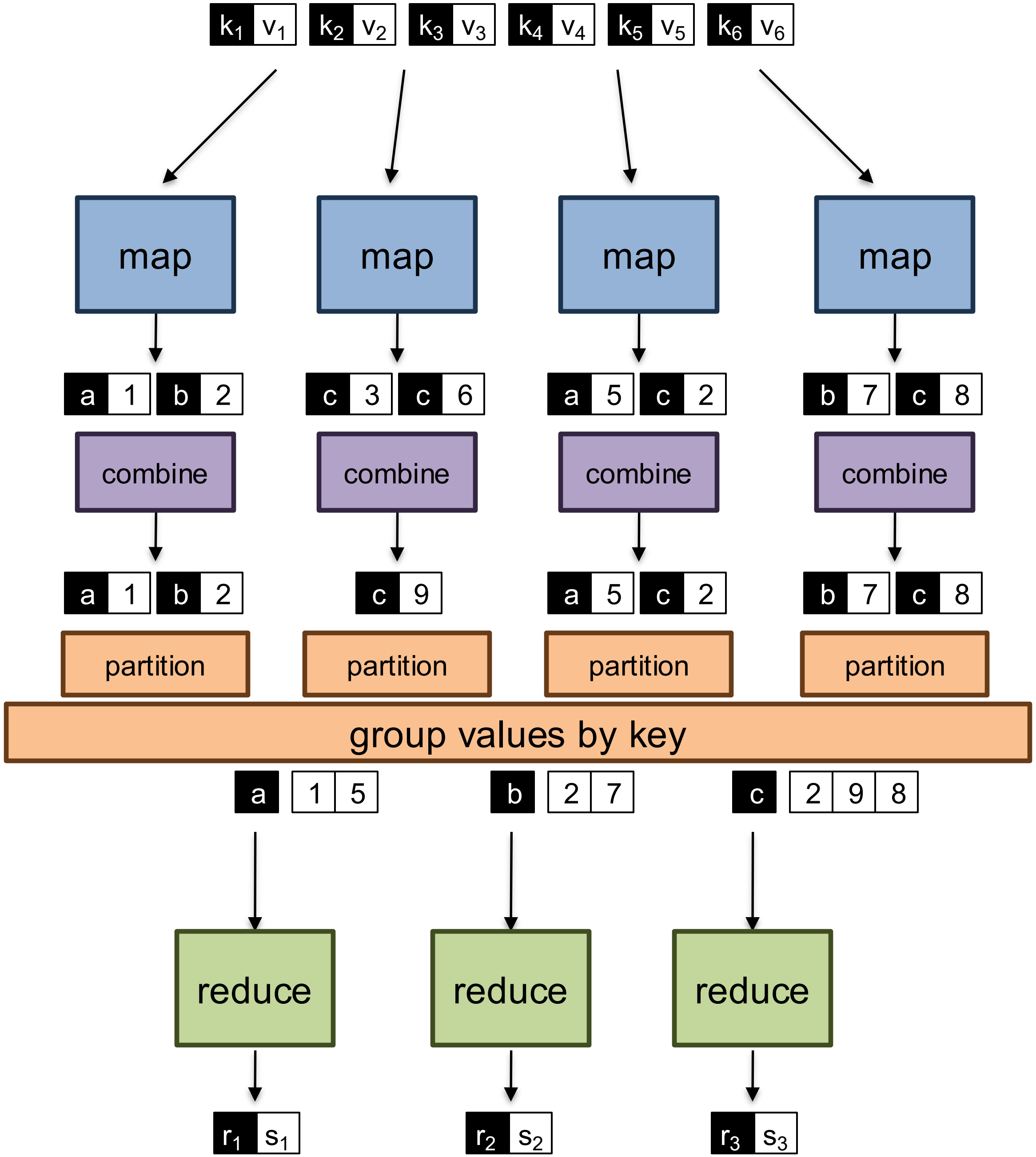
$$\left[v_8 \oplus v_9 \right] \oplus \left[v_4 \oplus v_5 \oplus v_6 \oplus v_7 \right] \oplus \left[v_1 \oplus v_2 \oplus v_3 \right]$$

Implications for distributed processing?

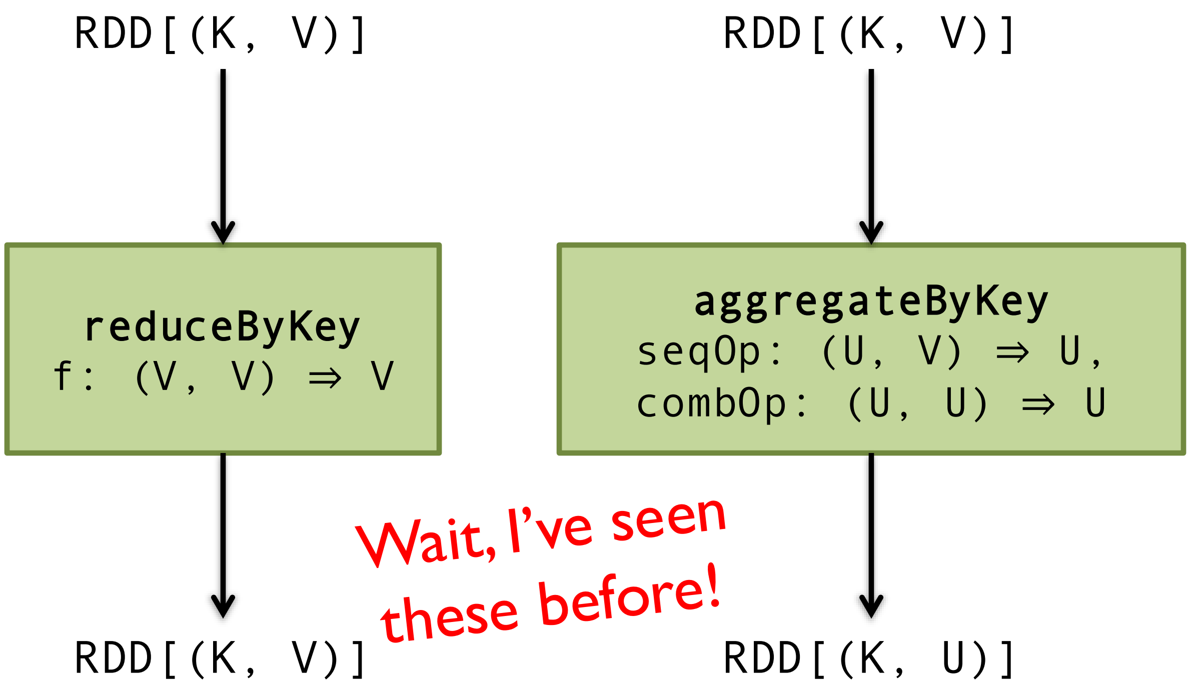
You don’t know when the tasks begin
You don’t know when the tasks end
You don’t know when the tasks interrupt each other
You don’t know when intermediate data arrive
…

It’s okay!

When can reducers be combiners?



Back to these…



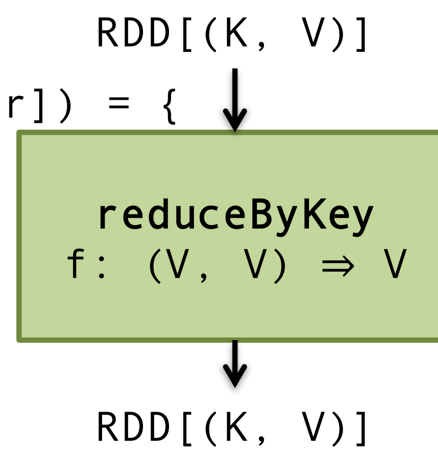
Computing the Mean: Version 1

```
class Mapper {
  def map(key: String, value: Int) = {
    emit(key, value)
  }
}

class Reducer {
  def reduce(key: String, values: Iterable[Int]) {
    for (value <- values) {
      sum += value
      cnt += 1
    }
    emit(key, sum/cnt)
  }
}
```

Computing the Mean: Version 3

```
class Mapper {
  def map(key: String, value: Int) =
    context.write(key, (value, 1))
}
class Combiner {
  def reduce(key: String, values: Iterable[Pair]) = {
    for ((s, c) <- values) {
      sum += s
      cnt += c
    }
    emit(key, (sum, cnt))
  }
}
class Reducer {
  def reduce(key: String, values: Iterable[Pair]) = {
    for ((s, c) <- values) {
      sum += s
      cnt += c
    }
    emit(key, sum/cnt)
  }
}
```



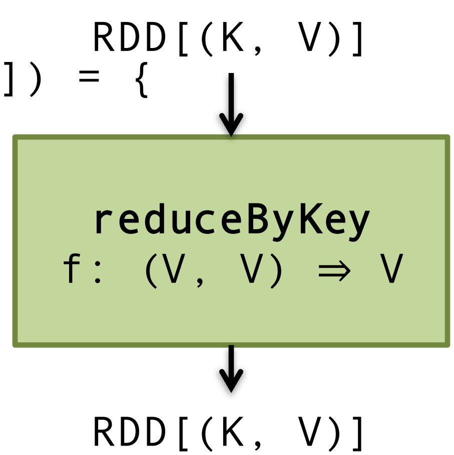
Co-occurrence Matrix: Stripes

```
class Mapper {
  def map(key: Long, value: String) = {
    for (u <- tokenize(value)) {
      val map = new Map()
      for (v <- neighbors(u)) {
        map(v) += 1
      }
      emit(u, map)
    }
  }
}
```

a → { b: 1, c: 2, d: 5, e: 3, f: 2 }

a → { b: 1, d: 5, e: 3 }
\+ a → { b: 1, c: 2, d: 2, f: 2 }
a → { b: 2, c: 2, d: 7, e: 3, f: 2 }

```
class Reducer {
  def reduce(key: String, values: Iterable[Map]) = {
    val map = new Map()
    for (value <- values) {
      map += value
    }
    emit(key, map)
  }
}
```



Computing the Mean: Version 2

```
class Mapper {
  def map(key: String, value: Int) =
    context.write(key, value)
}
class Combiner {
  def reduce(key: String, values: Iterable[Int]) = {
    for (value <- values) {
      sum += value
      cnt += 1
    }
    emit(key, (sum, cnt))
  }
}
class Reducer {
  def reduce(key: String, values: Iterable[Pair]) = {
    for ((s, c) <- values) {
      sum += s
      cnt += c
    }
    emit(key, sum/cnt)
  }
}
```

RDD[(K, V)]

aggregateByKey
seqOp: (U, V) ⇒ U,
combOp: (U, U) ⇒ U

RDD[(K, U)]

Super powers:

Associativity and Commutativity!

(But what's the fallback option?)

Technical Deep Dives

Implementation of distributed group by

… implications for algorithm design

The importance of partitioning

Case study in relational joins

Why does partitioning matter?

Ideal Scaling: Why not?

Communication is unavoidable

Workers need to share intermediate results…
Which requires communication across machines…
Which requires synchronization…
Which kills performance.

Let's of discussion here!

Skew creates idle workers

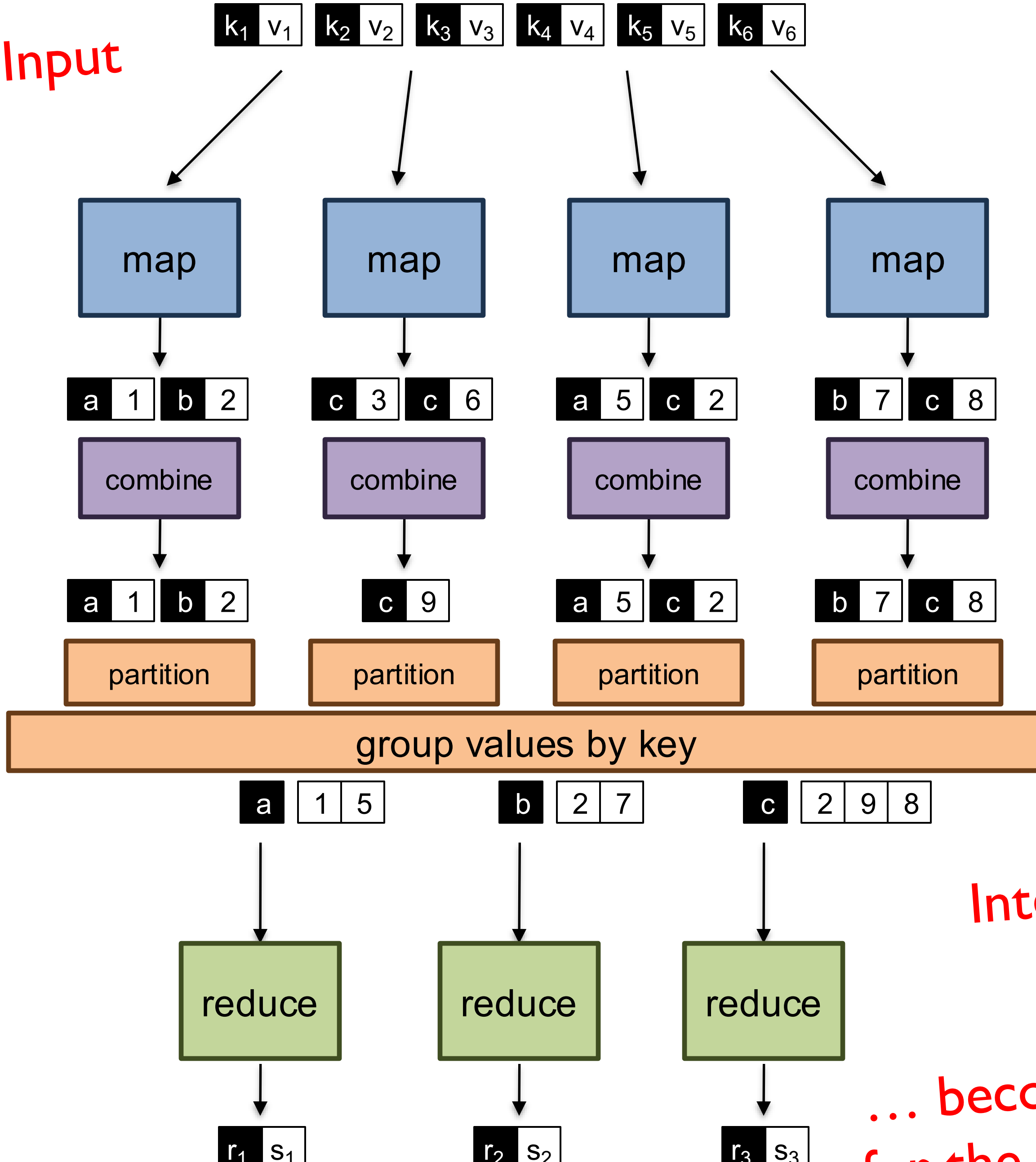
Tasks are never divided perfectly evenly…
And even if they are, processing times can be unpredictable…
Which leads to idle workers.

This matters also!

Why does partitioning matter?

Better partitioning remediates skew!

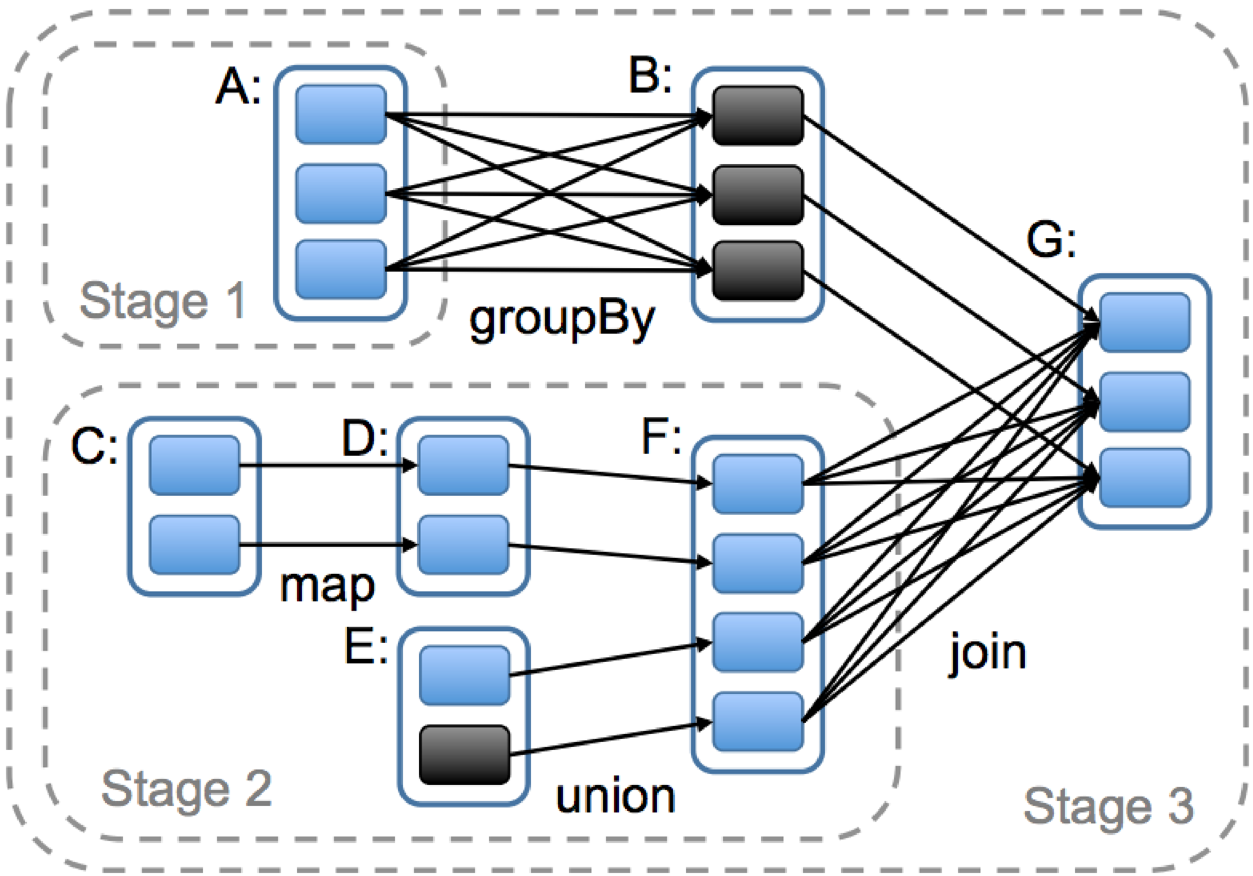
Skew in Input



Skew in Intermediates

… becomes input for the next phase

Spark Execution Plan



Partition challenges

Machines are heterogenous

Equal partitioning of inputs is hard…

Equal partitioning of intermediates is even harder…

Hash partitioning

Implementation?

Pairs: Pseudo-Code

```
class Mapper {
  def map(key: Long, value: String) = {
    for (u <- tokenize(value)) {
      for (v <- neighbors(u)) {
        emit((u, v), 1)
      }
    }
  }
}

class Reducer {
  def reduce(key: Pair, values: Iterable[Int]) = {
    for (value <- values) {
      sum += value
    }
    emit(key, sum)
  }
}
```

Pairs: Pseudo-Code

One more thing…

```
class Partitioner {
  def getPartition(key: Pair, value: Int, numTasks: Int): Int = {
    return key.left % numTasks
  }
}
```

Hash partitioning

Implementation?

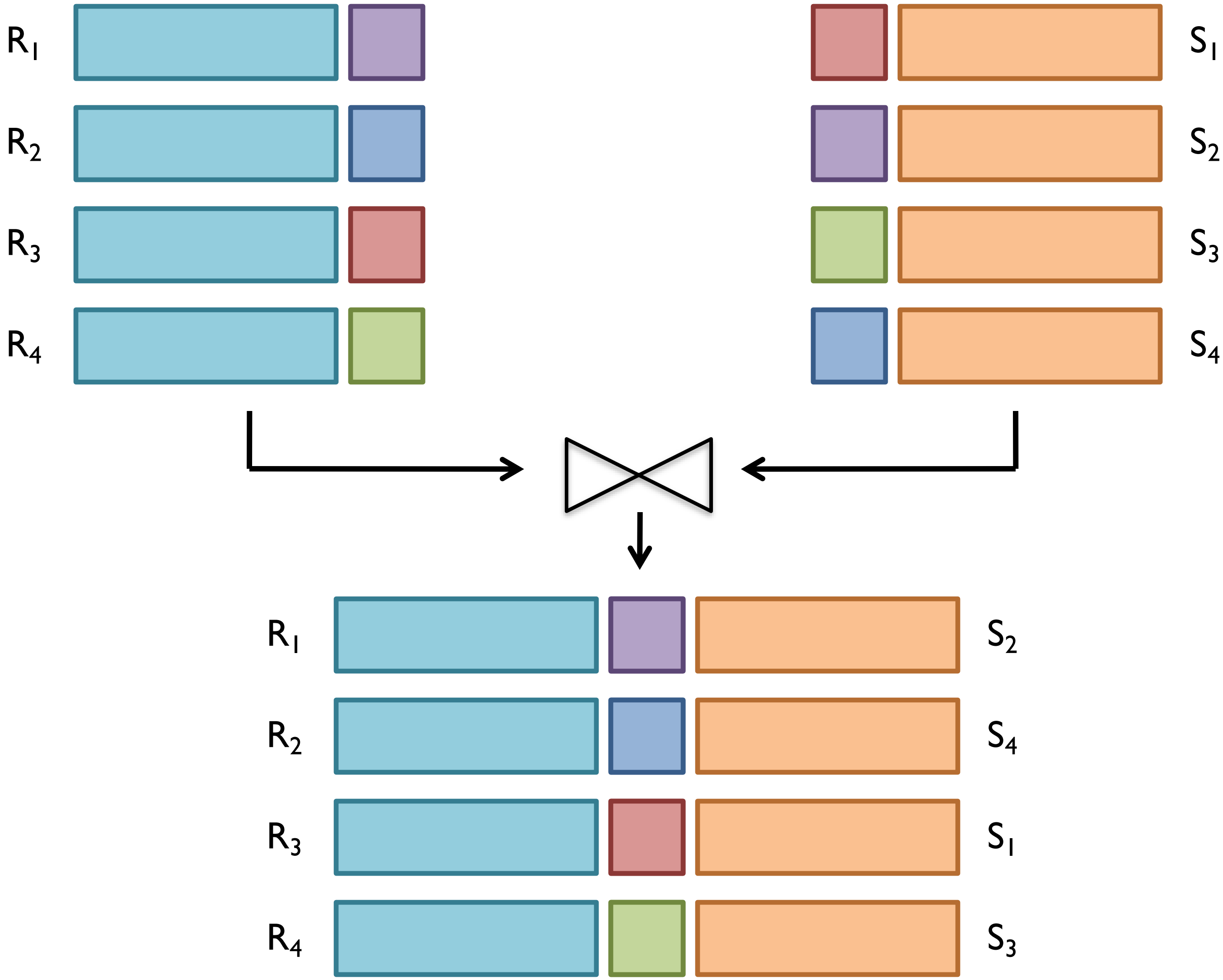
Range partitioning

Implementation?

Technical Deep Dives

Implementation of distributed group by
… implications for algorithm design
The importance of partitioning
Case study in relational joins

Relational Joins



(More precisely, an inner join)

Spark Stack

DataFrames / DataSets

RDDs

Physical Operators

What level are we operating in?

Join Algorithms in MapReduce / Spark

Reduce-side join

aka repartition join

aka shuffle join

Map-side join

aka sort-merge join

Hash join

aka broadcast join

aka replicated join

Reduce-side Join

aka repartition join, shuffle join

Intuition: group by join key

Map over both datasets

Emit tuple as value with join key as the intermediate key

Execution framework brings together tuples sharing the same key

Perform join in reducer

Reduce-side Join

aka repartition join, shuffle join

Intuition: group by join key

~~Map over both datasets~~ Union two RDDs

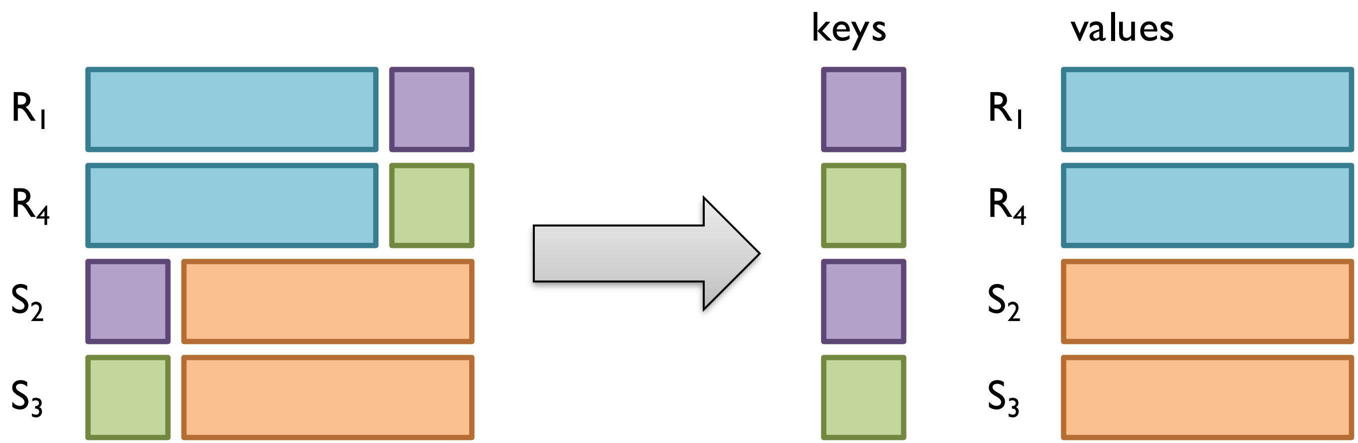
Emit tuple as value with join key as the intermediate key

Execution framework brings together tuples sharing the same key

Perform join in reducer

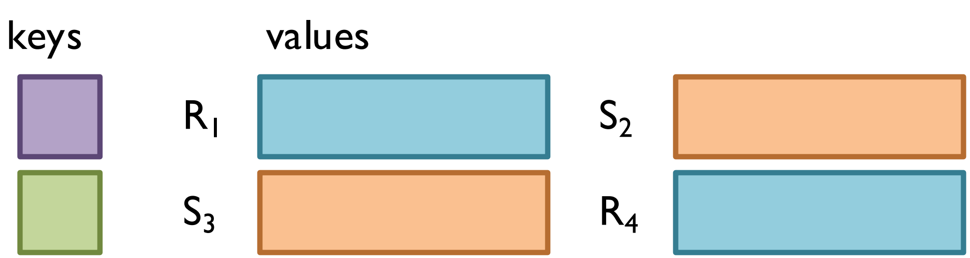
Reduce-side Join

Map



Remember to "tag" the tuple as being from R or S…

Reduce



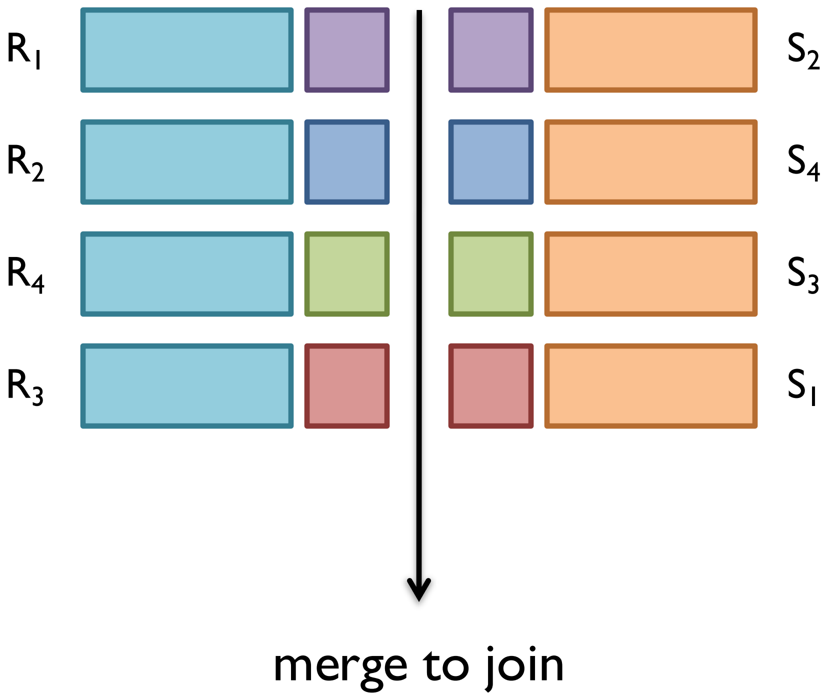
Note: no guarantee if R is going to come first or S

More precisely, an inner join: What about outer joins?

Map-side Join

aka sort-merge join

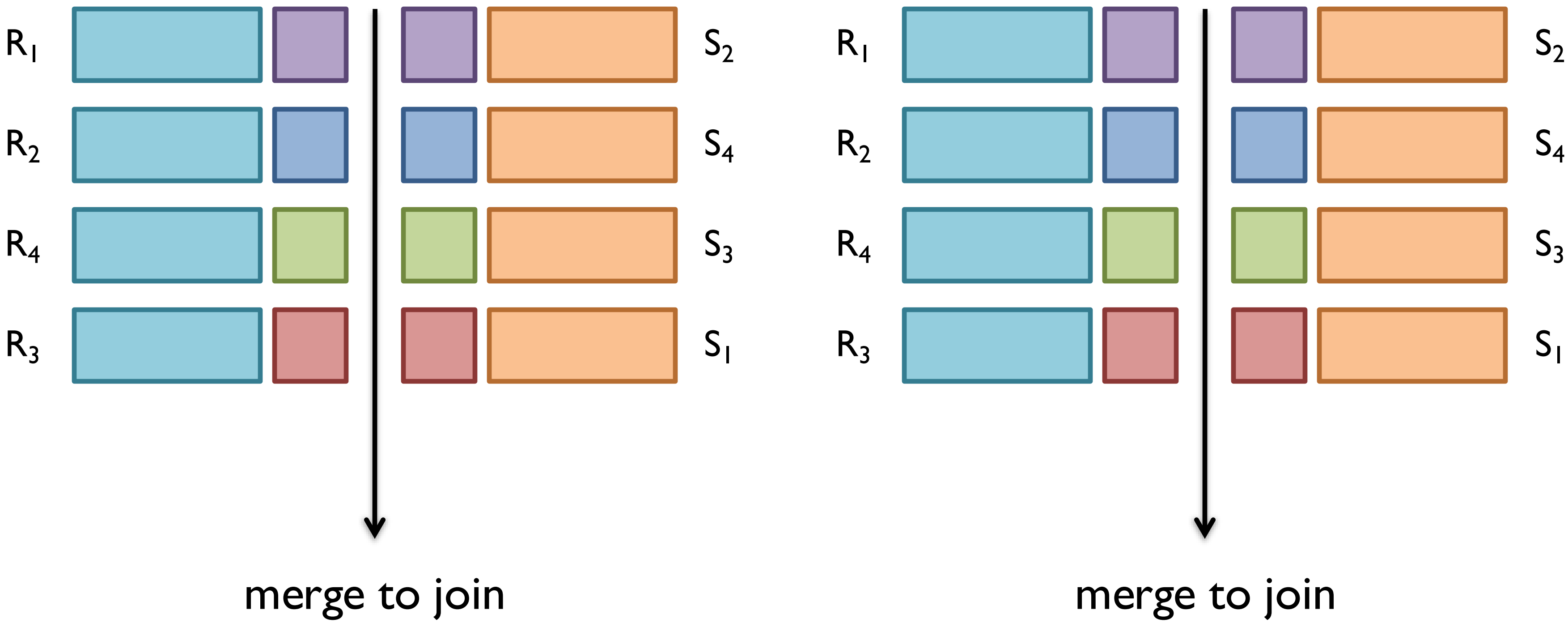
Assume two datasets are sorted by the join key:



Map-side Join

aka sort-merge join

Assume two datasets are sorted by the join key:



How can we parallelize this? Co-partitioning

Map-side Join

aka sort-merge join

Works if…

Two datasets are co-partitioned

Sorted by join key

MapReduce implementation:

Map over one dataset, read from other corresponding partition

Co-partitioned, sorted datasets: realistic to expect?

With proper setup, Spark automatically optimizes!

Hash Join

aka broadcast join, replicated join

Basic idea:

Load one dataset into memory in a hashmap, keyed by join key
Read the other dataset, probe for join key

Works if…

R << S and R fits into memory

Implementation:

Distribute R to all machines (e.g., broadcast in Spark)
Map over S, each mapper loads R in memory and builds the hashmap
For every tuple in S, probe join key in R

Hash Join Variants

Co-partitioned variant:

R and S co-partitioned (but not sorted)?

Only need to build hashmap on the corresponding partition

Striped variant:

R too big to fit into memory?

Divide R into R_1, R_2, R_3, … s.t. each R_n fits into memory

Perform hash join: $\forall n$, $R_n \bowtie S$

Take the union of all join results

Use a global key-value store:

Load R into memcached (or Redis)

Probe global key-value store for join key

Which join to use?

hash join > map-side join > reduce-side join

Limitations of each?

Hash join: memory

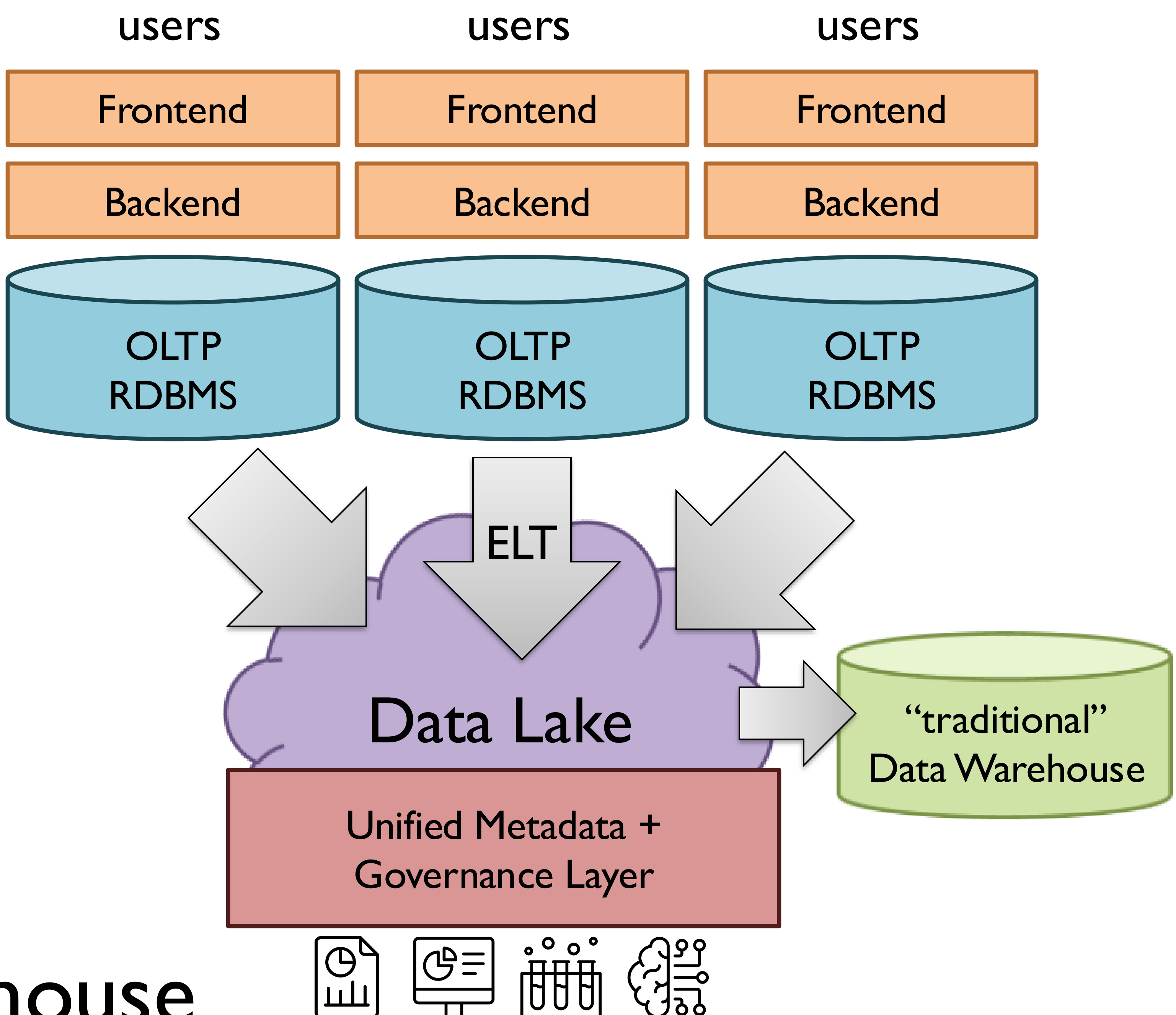
Map-side join: sort order and partitioning

Reduce-side join: general purpose

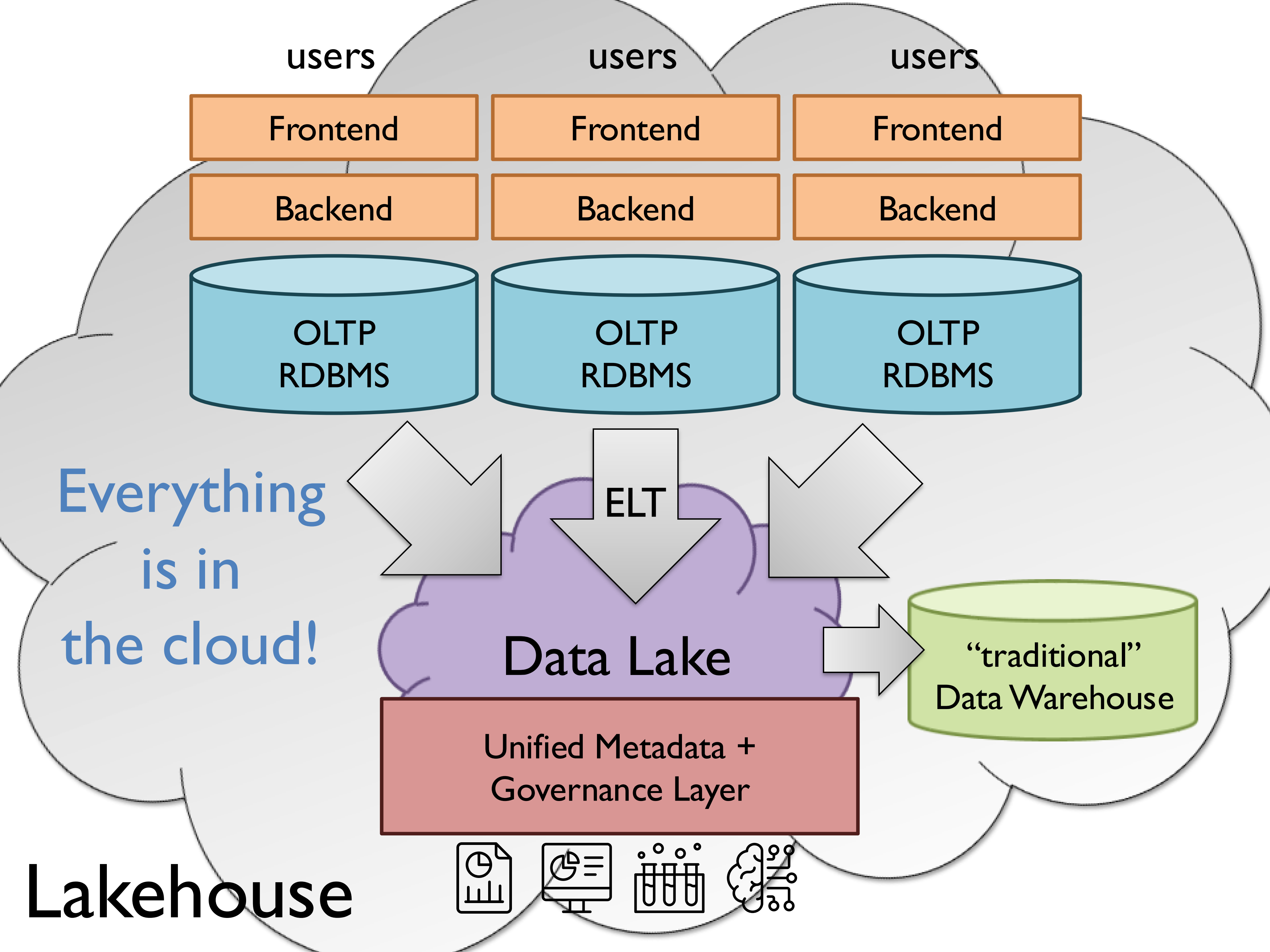
Who decides?

Technical Deep Dives

Implementation of distributed group by
… implications for algorithm design
The importance of partitioning
Case study in relational joins


users
users
users
Frontend
Frontend
Frontend
Backend
Backend
Backend
OLTP RDBMS
OLTP RDBMS
OLTP RDBMS
ELT
Data Lake
"traditional" Data Warehouse
Unified Metadata + Governance Layer

Lakehouse


users
users
users
Frontend
Frontend
Frontend
Backend
Backend
Backend
OLTP RDBMS
OLTP RDBMS
OLTP RDBMS
Everything is in the cloud!
ELT
Data Lake
"traditional" Data Warehouse
Unified Metadata + Governance Layer
Lakehouse

In the cloud, does any of this matter?

The cloud is just another abstraction!

Pros

You don't have to worry about it.

You don't need to know what's going on. ✓

Cons

You can't worry about it (even if you wanted to).

You don't know what's going on (even if you wanted to). ✗

Spark Stack

DataFrames / DataSets

RDDs

Physical Operators

What level are we operating in?

Next week…

Source: Wikipedia