Carnegie Mellon University

DATABASE SYSTEMS

Advanced DB Speed-Run

LECTURE #24 » 15-445/645 FALL 2025 » PROF. ANDY PAVLO

ADMINISTRIVIA

Project #4 is due Sunday Dec 7th @ 11:59pm
→ Recitation Slides + Video (@300)
→ Office Hours Saturday Dec 6th @ 3:00-5:00pm (GHC 5201)

Homework #6 is due Sunday Dec 7th @ 11:59pm

Final Exam is on Thursday Dec 11th @ 1:00pm
→ Do not make travel plans before this date!

We are recruiting TAs for the next semester
→ Apply at: https://www.ugrad.cs.cmu.edu/ta/S26/

OFFICE HOURS

Andy:

→ Wednesday Dec 10th @ 10:30-12:00pm (GHC 9019)
→ Wednesday Dec 10th @ 4:00-5:00pm (GHC 9019)

All other TAs will have their office hours up to and including Saturday Dec 7th

FINAL EXAM

Where: McConomy Auditorium (University Center)

When: Thursday Dec 11th @ 1:00-4:00pm

What to bring:

→ CMU ID
→ Pencil + Eraser (!!!)
→ Calculator (cellphone is okay)
→ One 8.5x11" page of handwritten notes (double-sided)

https://15445.courses.cs.cmu.edu/fall2025/final-guide.html

STUFF BEFORE MID-TERM

SQL
Buffer Pool Management
Data Structures (Hash Tables, B+Trees)
Storage Models
Query Processing Models
Inter-Query Parallelism
Basic Understanding of BusTub Internals

JOIN ALGORITHMS

Join Algorithms
→ Naïve Nested Loops
→ Block Nested Loops
→ Index Nested Loops
→ Sort-Merge
→ Hash Join: Simple, Partitioned, Hybrid Hash
→ Optimization using Bloom Filters
→ Cost functions

QUERY EXECUTION

Execution Models
- → Iterator
- → Materialized
- → Vector / Batch

Plan Processing: Push vs. Pull

Access Methods
- → Sequential Scan and various optimization
- → Index Scan, including multi-index scan
- → Issues with update queries

Expression Evaluation

QUERY EXECUTION

Process Model

Parallel Execution

→ Inter Query Parallelism

→ Intra Query Parallelism: Intra-Operator: horizontal, vertical, and bushy
Parallel hash join, Exchange operator

→ Intra Query Parallelism: Inter-Operator, aka. pipelined parallelism

IO Parallelism

QUERY OPTIMIZATION

Heuristics

→ Predicate Pushdown
→ Projection Pushdown
→ Nested Sub-Queries: Rewrite and Decompose

Statistics

→ Cardinality Estimation
→ Histograms

Cost-based search

→ Bottom-up vs. Top-Down

TRANSACTIONS

ACID

Conflict Serializability:
→ How to check for correctness?
→ How to check for equivalence?

View Serializability
→ Difference with conflict serializability

Isolation Levels / Anomalies

TRANSACTIONS

Two-Phase Locking

→ **Strict 2PL:** Txn holds **X** locks until it commits or aborts. May release **S** locks earlier, during the shrinking phase.

→ **Strong Strict 2PL:** Txn holds all locks (**S** and **X**) until it commits or aborts. *Also called "Rigorous 2PL".*

Cascading Aborts Problem

Deadlock Detection & Prevention

Multiple Granularity Locking

→ Intention Locks

→ Understanding performance trade-offs

→ Lock Escalation (i.e., when is it allowed)

TRANSACTIONS

Optimistic Concurrency Control
→ Read Phase
→ Validation Phase (Backwards vs. Forwards)
→ Write Phase

Multi-Version Concurrency Control
→ Version Storage / Ordering
→ Garbage Collection
→ Index Maintenance

MVCC WITH 2PL

Schedule

TIME

TS(T_1)=1

TS(T_2)=2

	T_2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(A)
	W(A)
R(A)	
COMMIT	
	COMMIT

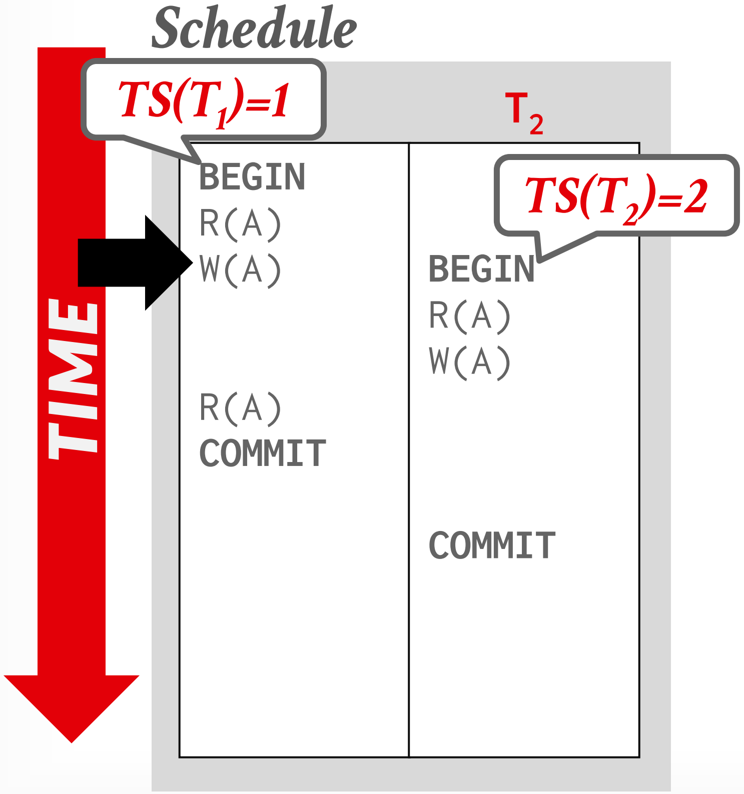
Database

	begin-ts	end-ts	value
A_0	0	-	123

Txn Status Table

txnid	timestamp	status
T_1	1	Active

MVCC WITH 2PL



Database

	begin-ts	end-ts	value
A_0	0	-	123
A_1	1	-	456

Txn Status Table

txnid	timestamp	status
T_1	1	Active

MVCC WITH 2PL

Schedule

TIME

$TS(T_1)=1$

$TS(T_2)=2$

T_1	T_2
BEGIN	
R(A)	
W(A)	BEGIN
	R(A)
	W(A)
R(A)	
COMMIT	
	COMMIT

Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Txn Status Table

txnid	timestamp	status
T_1	1	Active

MVCC WITH 2PL

Schedule

TIME

$TS(T_1)=1$

	T_2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(A)
	W(A)
R(A)	
COMMIT	
	COMMIT

$TS(T_2)=2$

T_2 reads version A_0 because T_1 has not committed yet.

Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

atus Table

	timestamp	status
T_1	1	Active
T_2	2	Active

MVCC WITH 2PL

Schedule

TIME

$TS(T_1)=1$	T_2
BEGIN	
R(A)	
W(A)	
	BEGIN ($TS(T_2)=2$)
	R(A)
	W(A)
R(A)	
COMMIT	
	COMMIT

T_2 stalls until T_1 commits to acquire write lock on A.

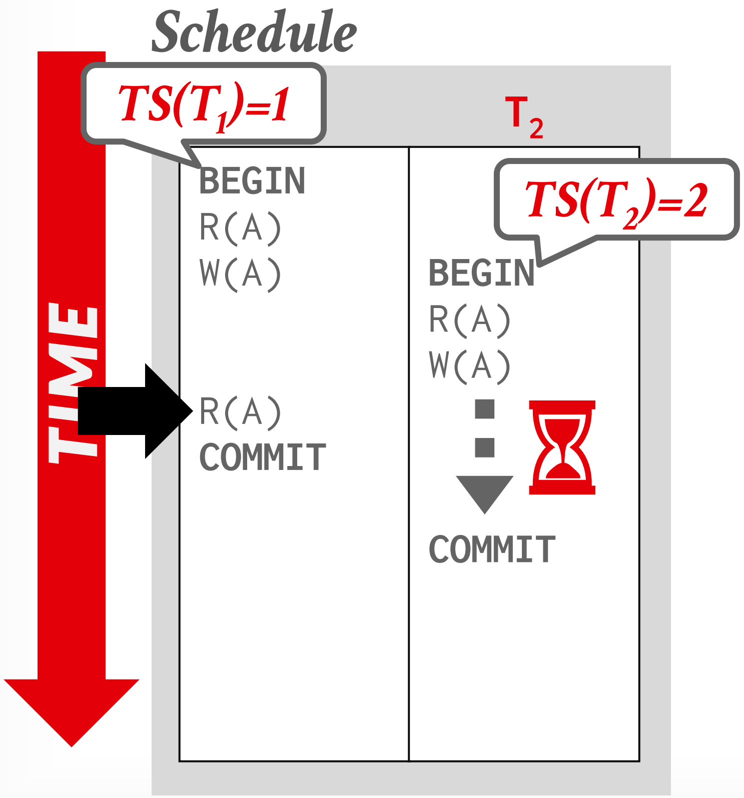
Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

…atus Table

	…imestamp	status
T_1	1	Active
T_2	2	Active

MVCC WITH 2PL



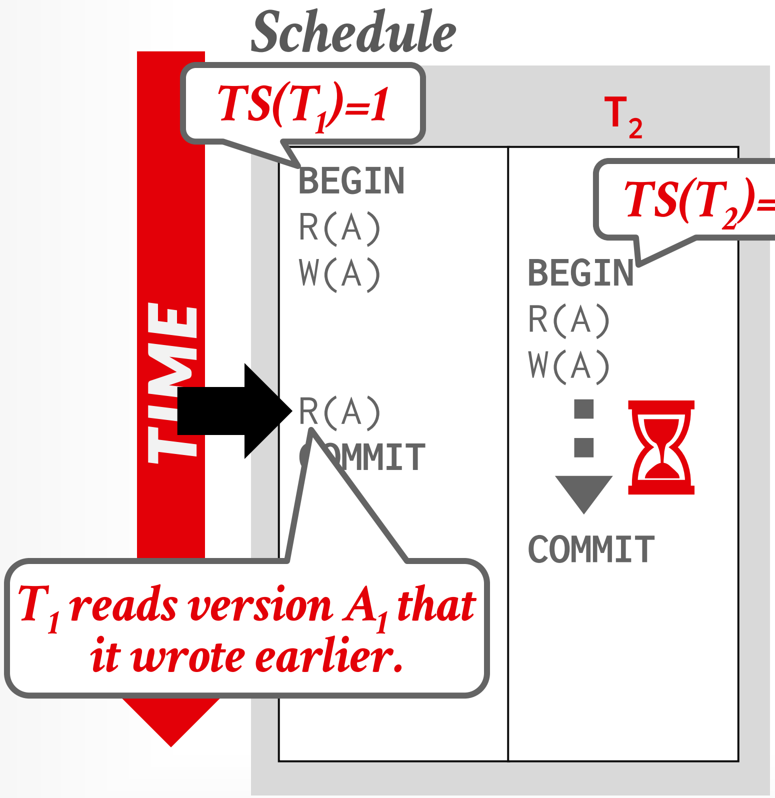
Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Txn Status Table

txnid	timestamp	status
T_1	1	Active
T_2	2	Active

MVCC WITH 2PL



Database

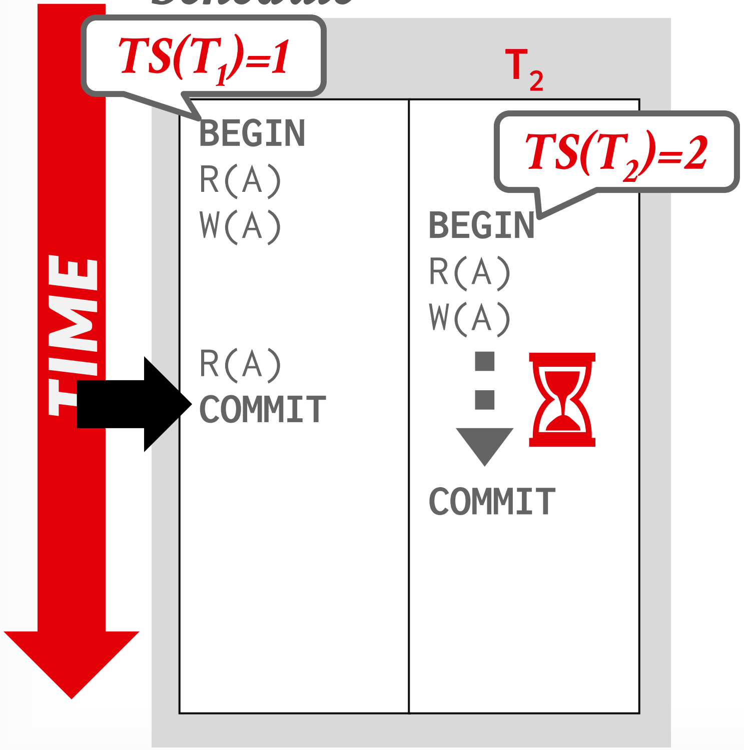
	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Txn Status Table

txnid	timestamp	status
T_1	1	Active
T_2	2	Active

MVCC WITH 2PL

Schedule



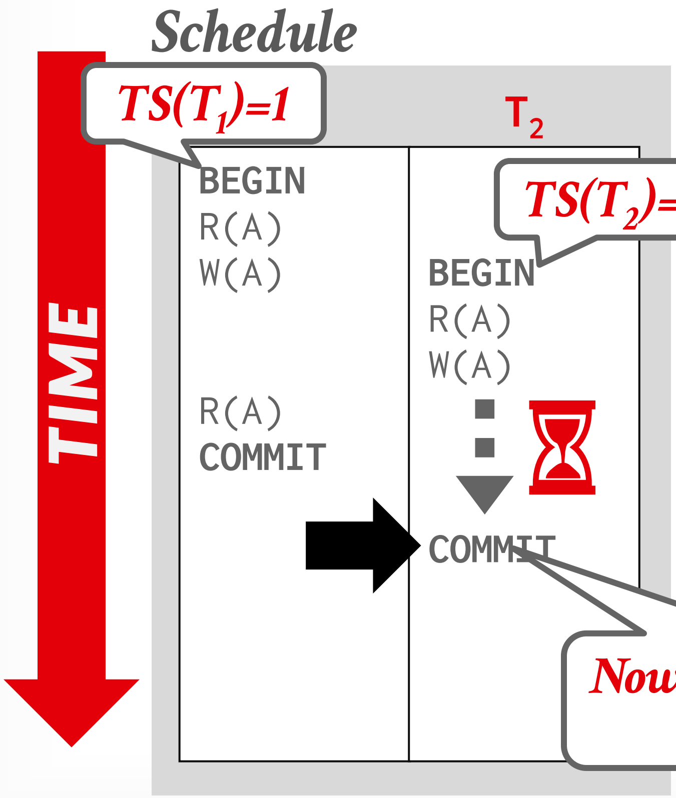
Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	-	456

Txn Status Table

txnid	timestamp	status
T_1	1	Committed
T_2	2	Active

MVCC WITH 2PL



Now T_2 can create the new version.

Database

	begin-ts	end-ts	value
A_0	0	1	123
A_1	1	2	456
A_2	2	-	789

Txn Status Table

txnid	timestamp	status
		Committed
		Active

CRASH RECOVERY

Buffer Pool Policies:
→ STEAL vs. NO-STEAL
→ FORCE vs. NO-FORCE

Shadow Paging

Write-Ahead Logging
→ How it relates to buffer pool management
→ Logging Schemes (Physical vs. Logical)

CRASH RECOVERY

Checkpoints
- → Non-Fuzzy vs. Fuzzy

ARIES Recovery
- → Dirty Page Table (DPT)
- → Active Transaction Table (ATT)
- → Analyze, Redo, Undo phases
- → Log Sequence Numbers
- → CLRs

DISTRIBUTED DATABASES

System Architectures

Replication Schemes

Partitioning Schemes

Two-Phase Commit

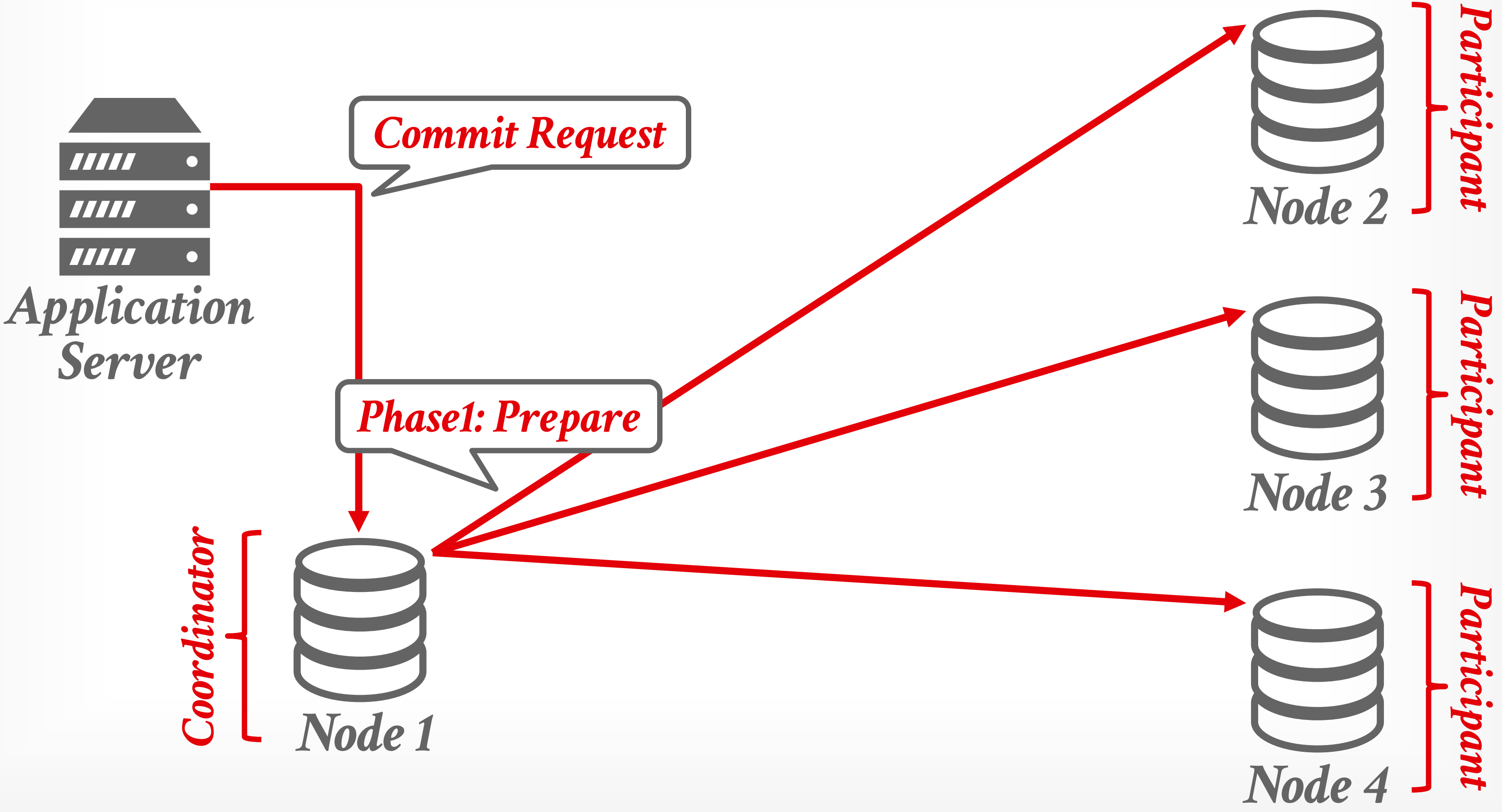
Paxos

Distributed Query Execution

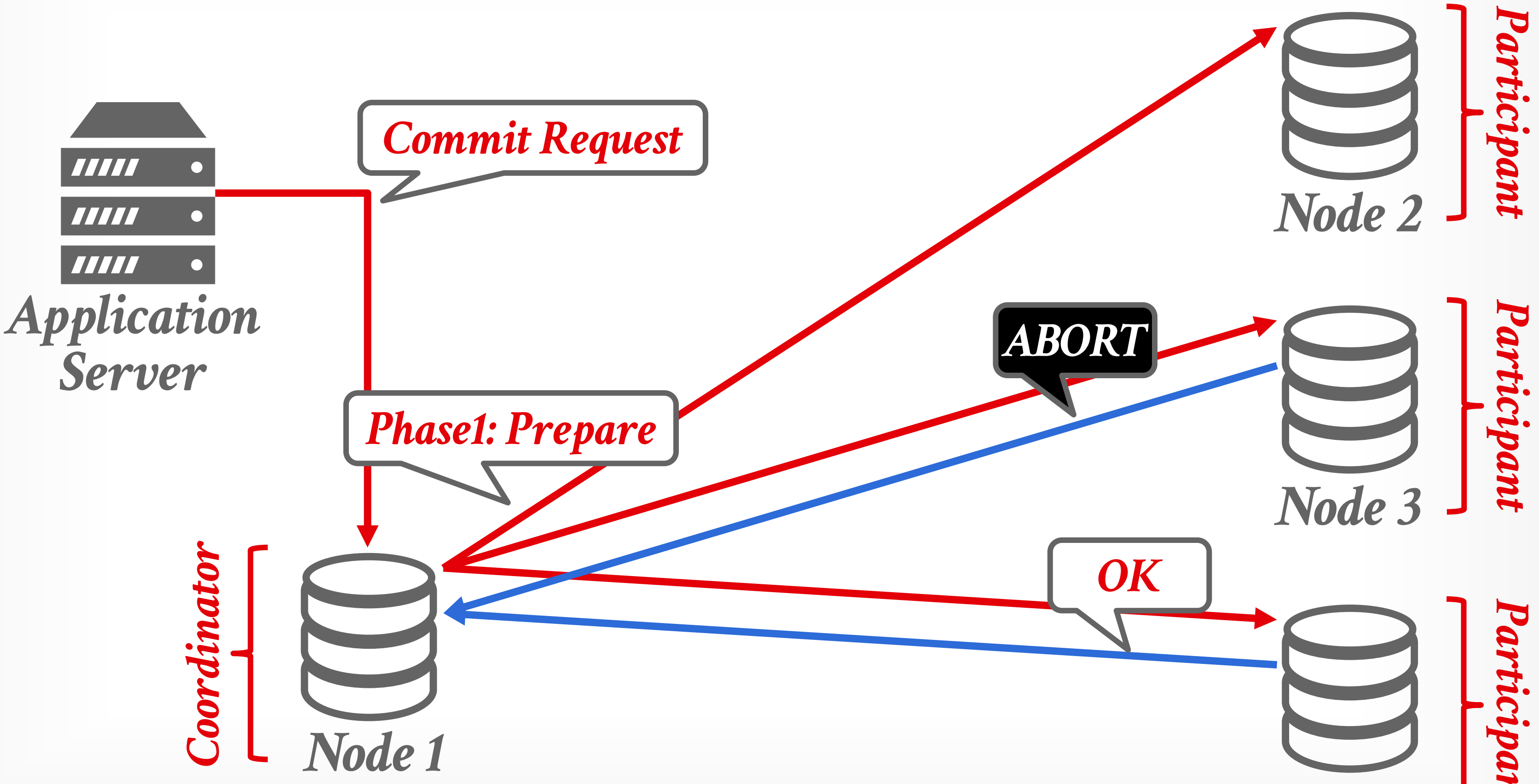
Distributed Join Algorithms

Semi-Join Optimization

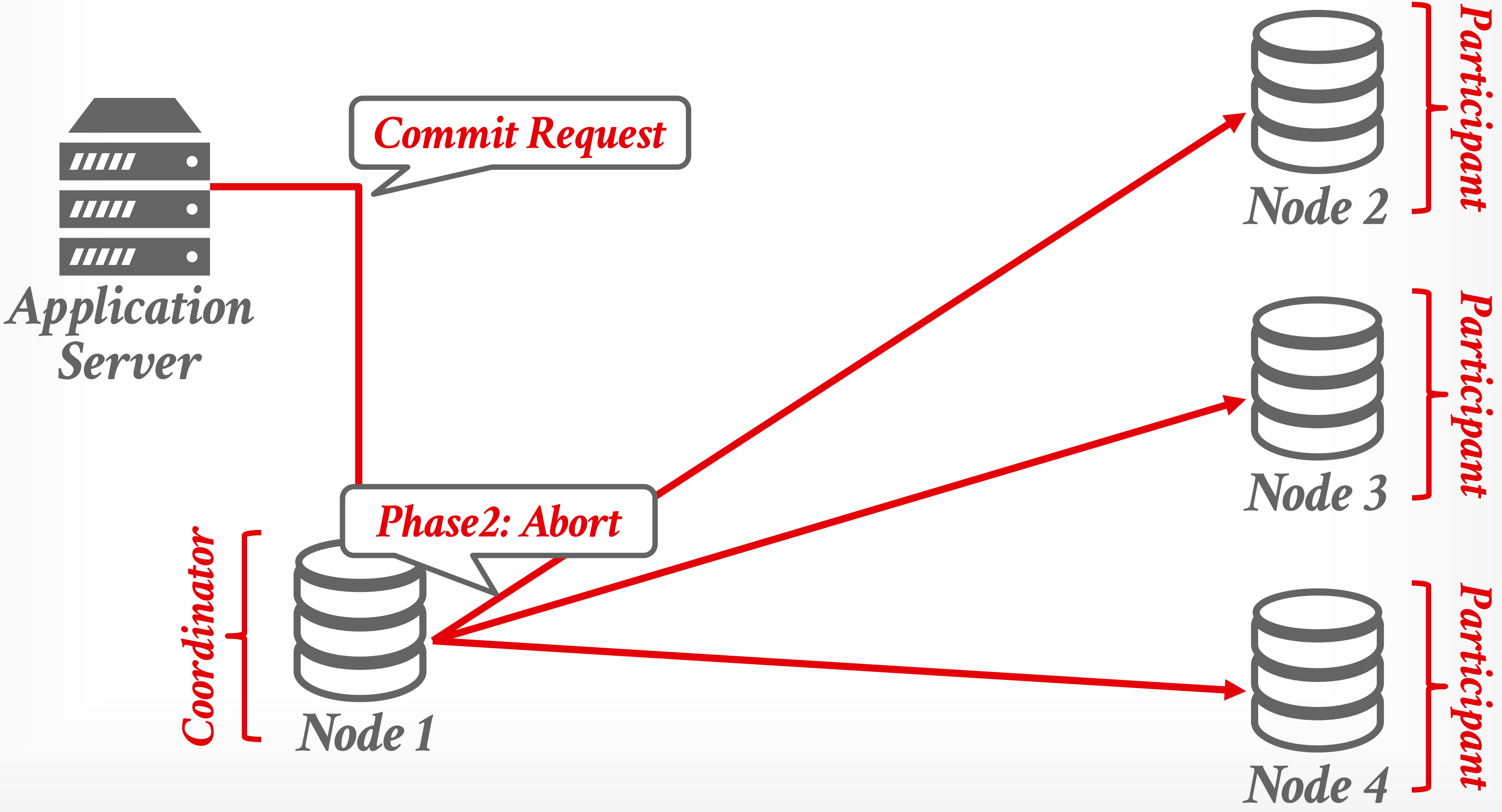
TWO-PHASE COMMIT (ABORT)



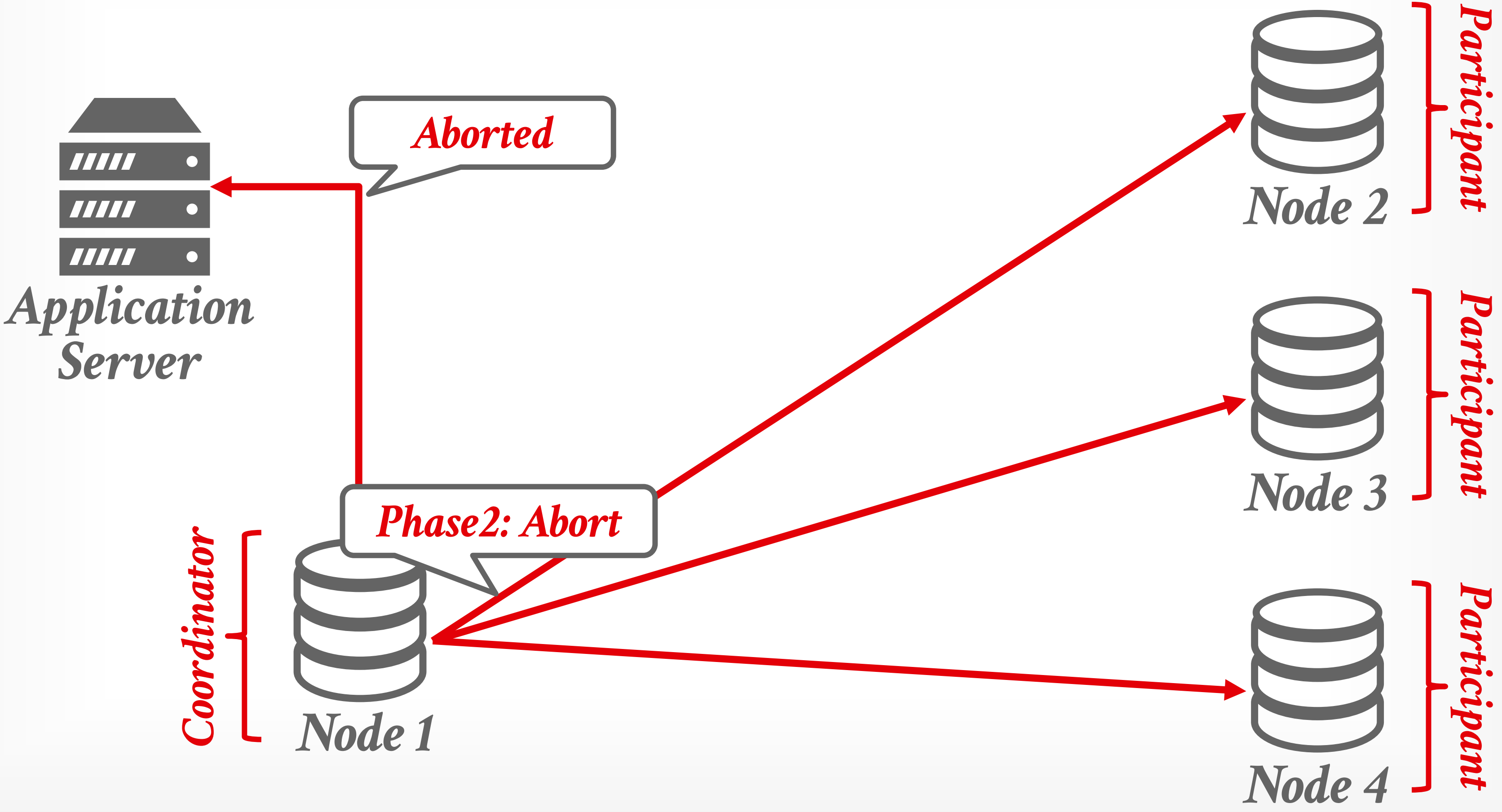
TWO-PHASE COMMIT (ABORT)



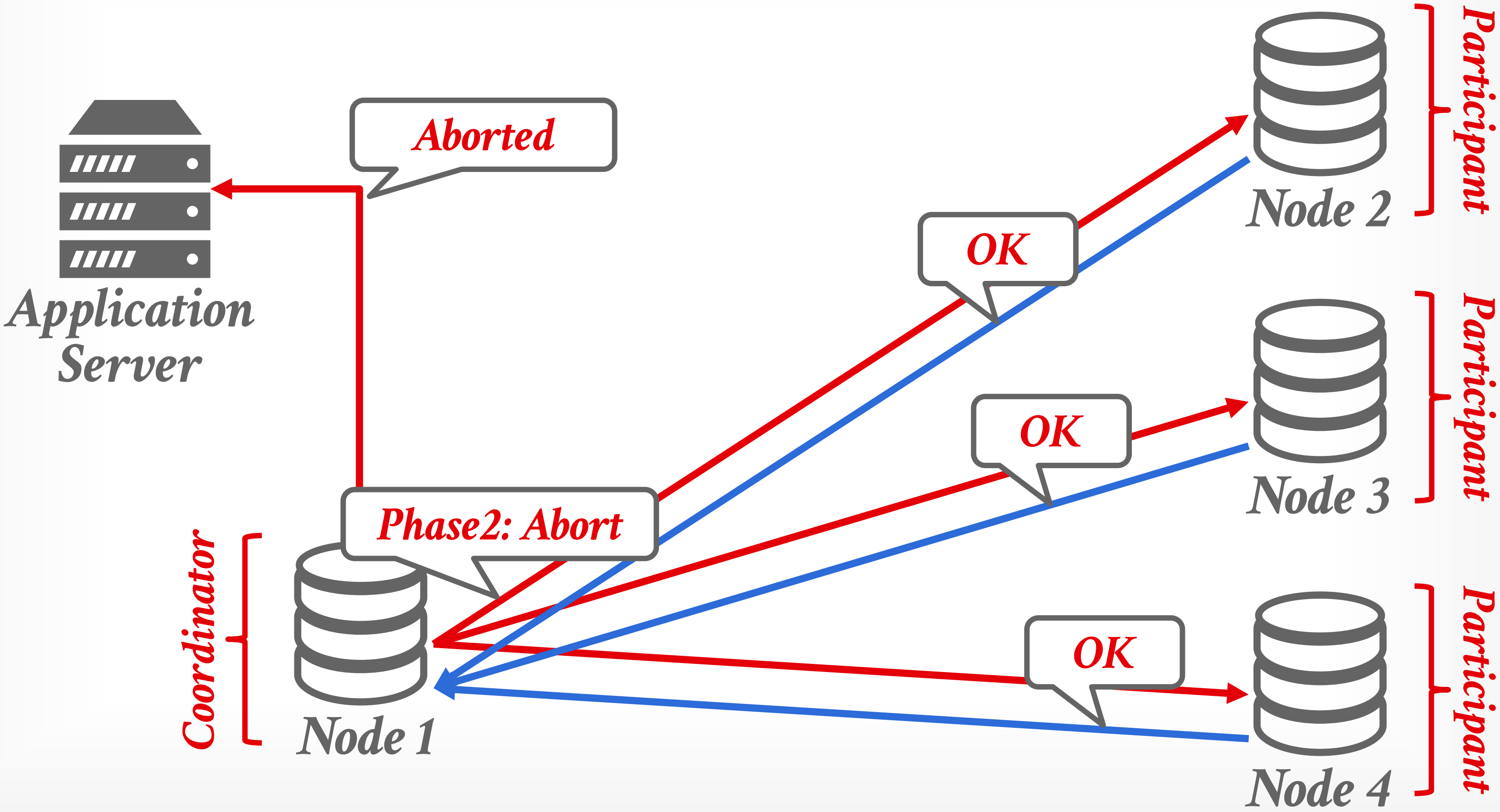
TWO-PHASE COMMIT (ABORT)



TWO-PHASE COMMIT (ABORT)



TWO-PHASE COMMIT (ABORT)



TOPICS NOT ON EXAM!

Flash Talks

Seminar Talks

Details of specific database systems (e.g., Postgres)

CMU 15-721 (Spring 2024)

SPEED RUN

https://15721.courses.cs.cmu.edu/spring2024

SEQUENTIAL SCAN: OPTIMIZATIONS

Lecture #05 Data Encoding / Compression

Lecture #06 Prefetching / Scan Sharing / Buffer Bypass

Lecture #14 Task Parallelization / Multi-threading

Lecture #08 Clustering / Sorting

Lecture #12 Late Materialization

Materialized Views / Result Caching

Lecture #13 Data Skipping

Lecture #14 Data Parallelization / Vectorization

$\boxed{\text{Code Specialization / Compilation}}$

SELECTION SCANS

```
SELECT * FROM table
 WHERE key > $(low)
   AND key < $(high)
```

Source: Bogdan Raducanu

SELECTION SCANS

Scalar (Branching)

```
i = 0
for t in table:
  key = t.key
  if (key>low) && (key<high):
    copy(t, output[i])
    i = i + 1
```

Source: Bogdan Raducanu

SELECTION SCANS

Scalar (Branching)

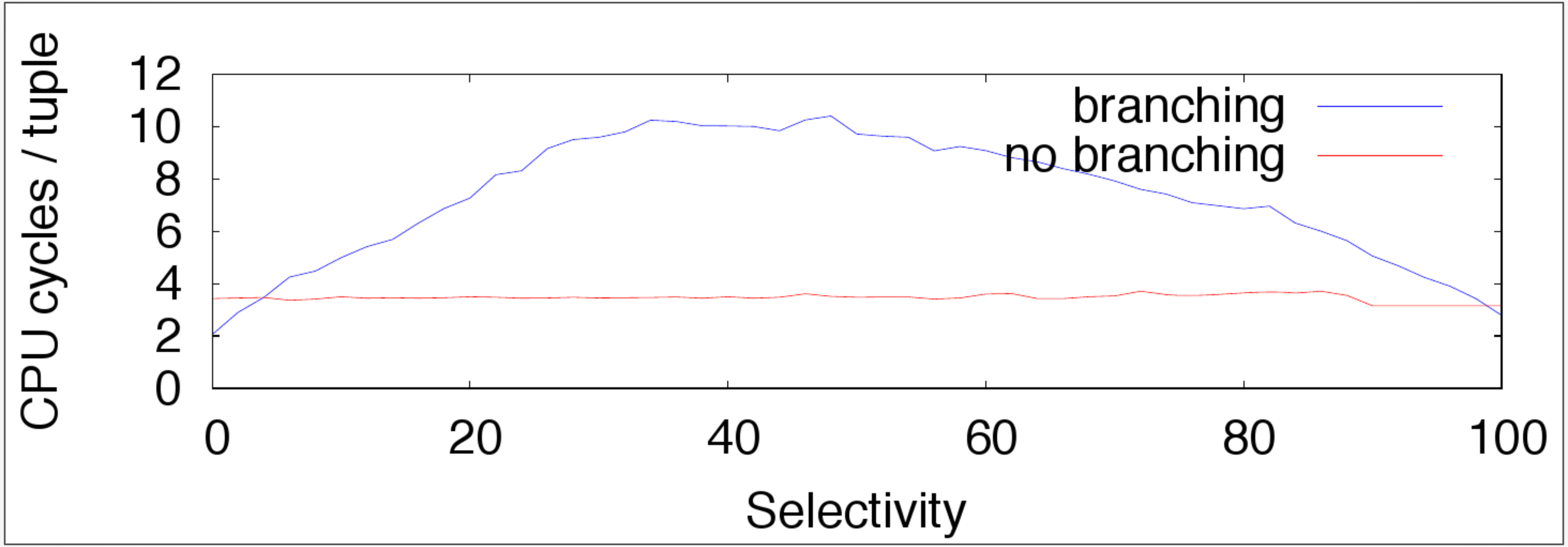
```
i = 0
for t in table:
  key = t.key
  if (key>low) && (key<high):
    copy(t, output[i])
    i = i + 1
```

Scalar (Branchless)

```
i = 0
for t in table:
  copy(t, output[i])
  key = t.key
  delta = (key>low ? 1 : 0) &
          ⮱(key<high ? 1 : 0)
  i = i + delta
```

Source: Bogdan Raducanu

SELECTION SCANS



Source: Bogdan Raducanu

SIMD SELECTION SCANS

Scalar (Branchless)

```
i = 0
for t in table:
  copy(t, output[i])
  key = t.key
  m = (key≥low ? 1 : 0) &
     ⮑(key≤high ? 1 : 0)
  i = i + m
```

```
SELECT * FROM table
 WHERE key >= $low AND key <= $high
```

SIMD SELECTION SCANS

Vectorized

```
i = 0
for vₜ in table:
  simdLoad(vₜ.key, vₖ)
  vₘ = (vₖ≥low ? 1 : 0) &
      ⮱(vₖ≤high ? 1 : 0)
  simdStore(vₜ, vₘ, output[i])
  i = i + |vₘ≠false|
```

```
SELECT * FROM table
 WHERE key >= $low AND key <= $high
```

SIMD SELECTION SCANS

Vectorized

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
       ⮱(vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

```
SELECT * FROM table
 WHERE key >= $low AND key <= $high
```

SIMD SELECTION SCANS

Vectorized

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
       ↳(vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

```
SELECT * FROM table
 WHERE key >= $low AND key <= $high
```

SIMD SELECTION SCANS

Vectorized

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
        ⮱(vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

```
SELECT * FROM table
 WHERE key >= $low AND key <= $high
```

SIMD SELECTION SCANS

Vectorized

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
       ⮱(vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

```
SELECT * FROM table
 WHERE key >= $low AND key <= $high
```

SIMD SELECTION SCANS

Vectorized

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
      ⮑(vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'U'
```

SIMD SELECTION SCANS

Vectorized

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
       ⮑(vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

tid	key
100	A
101	N
102	D
103	Y
104	P
105	I
106	S
107	💩

```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'U'
```

SIMD SELECTION SCANS

Vectorized

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
       ⮱(vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

tid	key
100	A
101	N
102	D
103	Y
104	P
105	I
106	S
107	💩

Key Vector

A	N	D	Y	P	I	S	💩

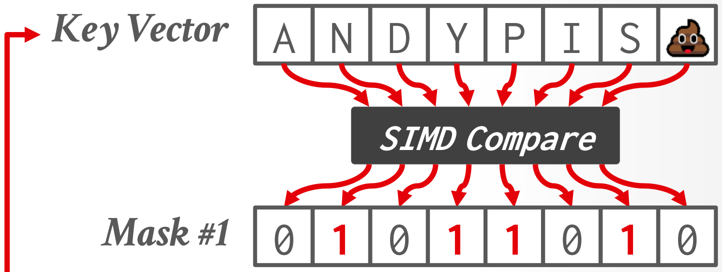
```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'U'
```

SIMD SELECTION SCANS

Vectorized

```
i = 0
for v_t in table:
  simdLoad(v_t.key, v_k)
  v_m = (v_k≥low ? 1 : 0) &
       ↳(v_k≤high ? 1 : 0)
  simdStore(v_t, v_m, output[i])
  i = i + |v_m≠false|
```

tid	key
100	A
101	N
102	D
103	Y
104	P
105	I
106	S
107	💩




```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'U'
```

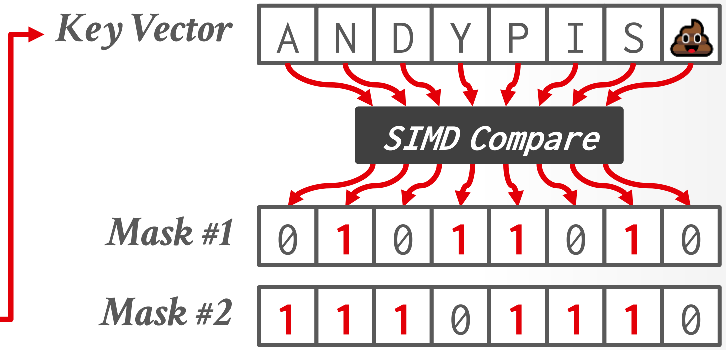
SIMD SELECTION SCANS

Vectorized

```
i = 0
for v_t in table:
   simdLoad(v_t.key, v_k)
   v_m = (v_k≥low ? 1 : 0) &
        ⮱(v_k≤high ? 1 : 0)
   simdStore(v_t, v_m, output[i])
   i = i + |v_m≠false|
```

```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'U'
```

tid	key
100	A
101	N
102	D
103	Y
104	P
105	I
106	S
107	💩



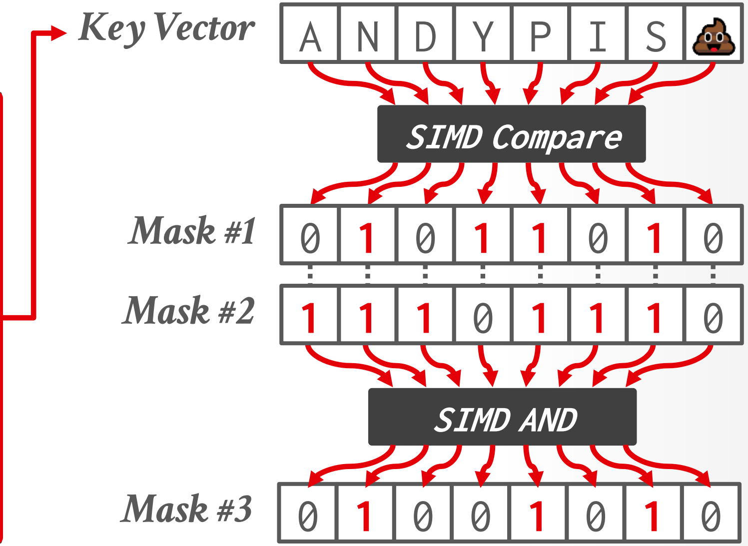
SIMD SELECTION SCANS

Vectorized

```
i = 0
for v_t in table:
  simdLoad(v_t.key, v_k)
  v_m = (v_k≥low ? 1 : 0) &
      ⮱(v_k≤high ? 1 : 0)
  simdStore(v_t, v_m, output[i])
  i = i + |v_m≠false|
```

```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'U'
```

tid	key
100	A
101	N
102	D
103	Y
104	P
105	I
106	S
107	💩



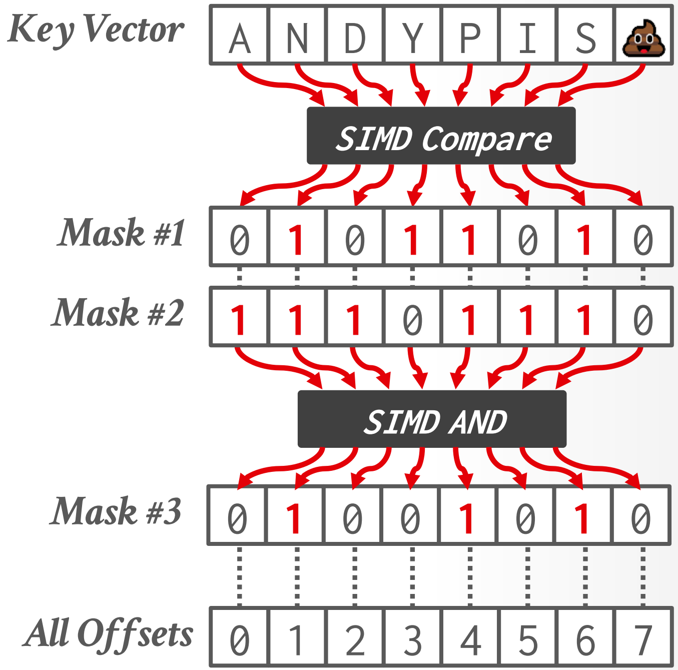
SIMD SELECTION SCANS

Vectorized

```
i = 0
for v_t in table:
  simdLoad(v_t.key, v_k)
  v_m = (v_k≥low ? 1 : 0) &
        ↳(v_k≤high ? 1 : 0)
  simdStore(v_t, v_m, output[i])
  i = i + |v_m≠false|
```

```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'U'
```

Offset	tid	key
0	100	A
1	101	N
2	102	D
3	103	Y
4	104	P
5	105	I
6	106	S
7	107	💩



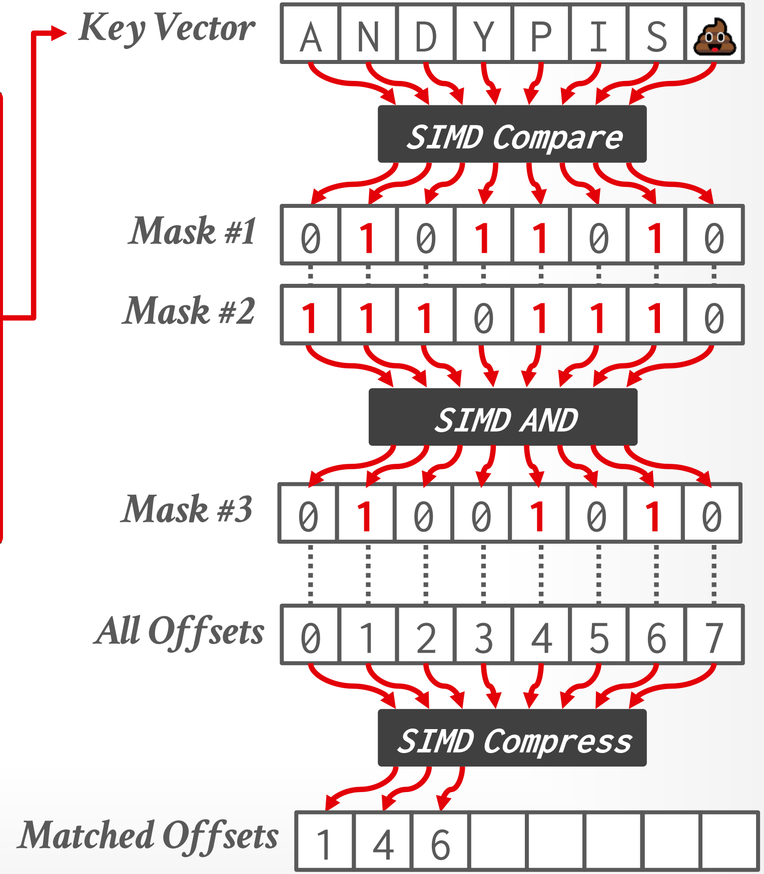
SIMD SELECTION SCANS

Vectorized

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
       ⮑(vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

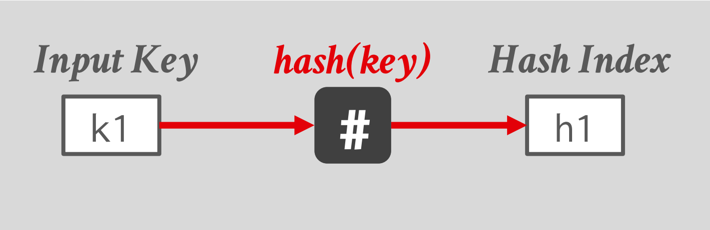
```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'U'
```

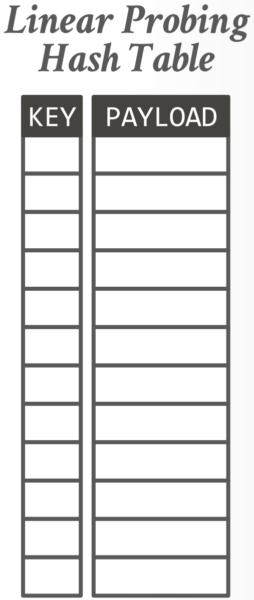
Offset	tid	key
0	100	A
1	101	N
2	102	D
3	103	Y
4	104	P
5	105	I
6	106	S
7	107	💩



SIMD HASH TABLE PROBING

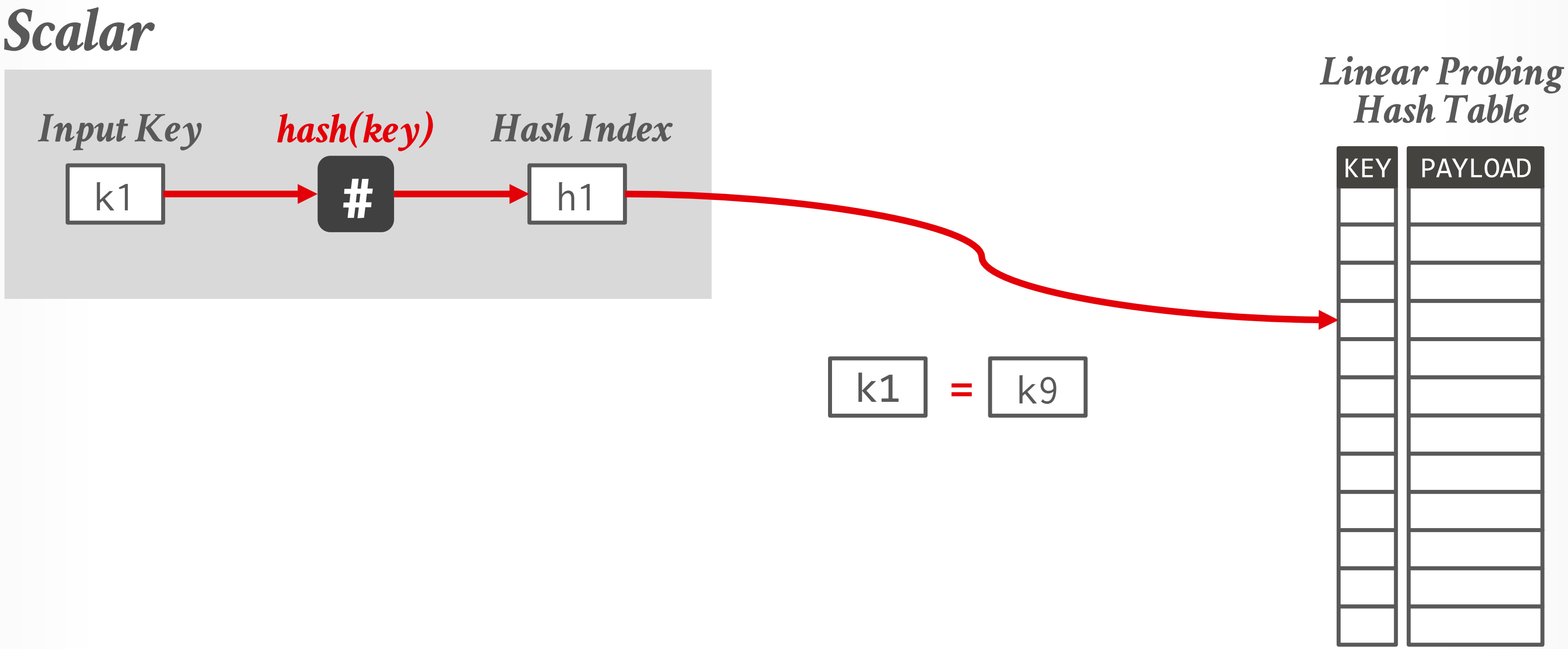
Scalar





MAKE THE MOST OUT OF YOUR SIMD INVESTMENTS: COUNTER CONTROL FLOW DIVERGENCE IN COMPILED QUERY PIPELINES
VLDB JOURNAL 2020

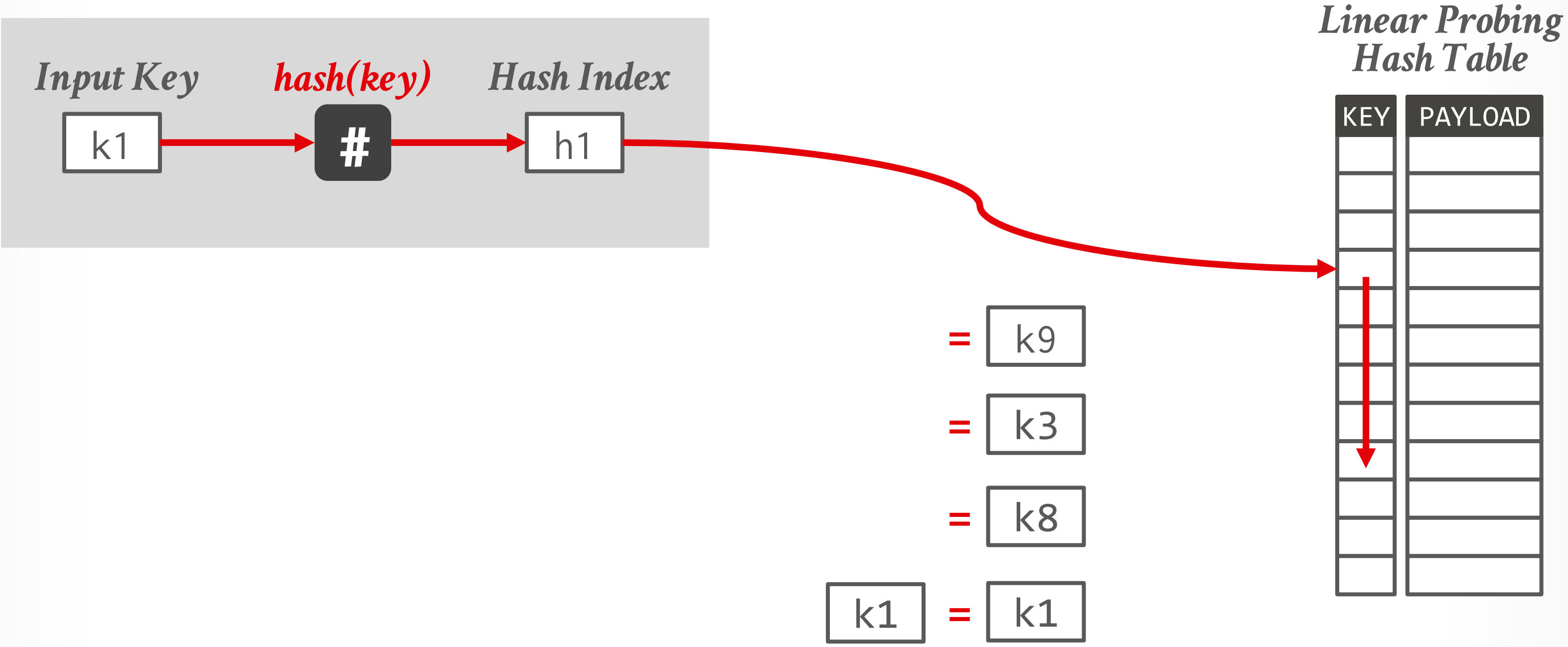
SIMD HASH TABLE PROBING



MAKE THE MOST OUT OF YOUR SIMD INVESTMENTS: COUNTER CONTROL FLOW DIVERGENCE IN COMPILED QUERY PIPELINES
VLDB JOURNAL 2020

SIMD HASH TABLE PROBING

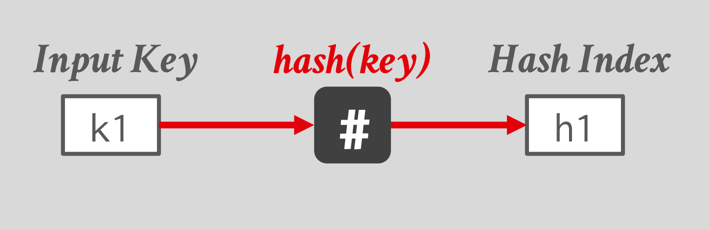
Scalar



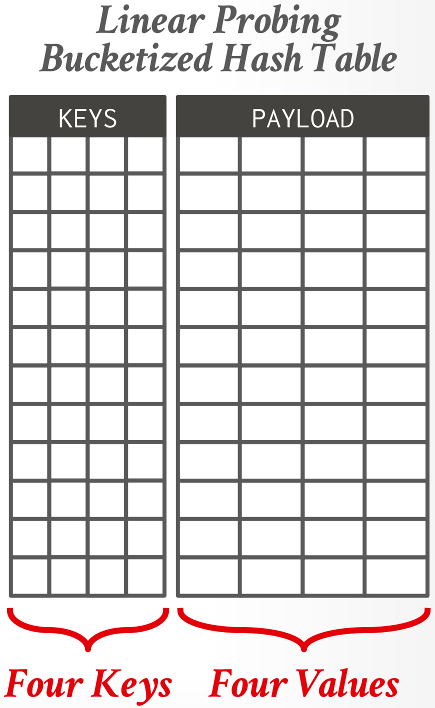
MAKE THE MOST OUT OF YOUR SIMD INVESTMENTS: COUNTER CONTROL FLOW DIVERGENCE IN COMPILED QUERY PIPELINES
VLDB JOURNAL 2020

SIMD HASH TABLE PROBING

Scalar



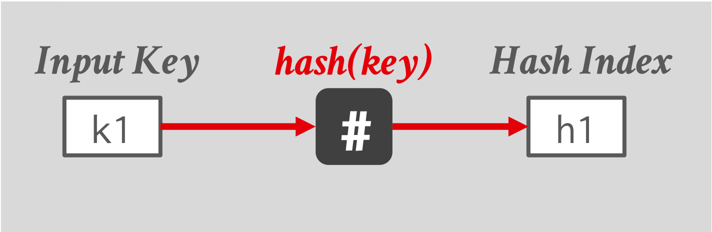
Vectorized (Horizontal)



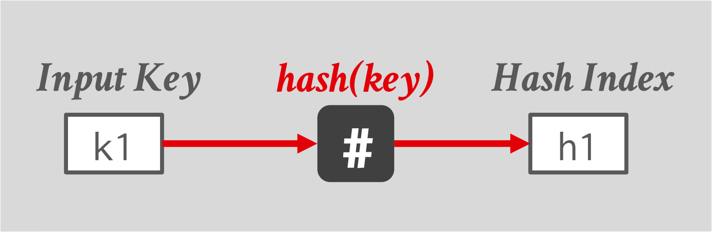
MAKE THE MOST OUT OF YOUR SIMD INVESTMENTS: COUNTER CONTROL FLOW DIVERGENCE IN COMPILED QUERY PIPELINES
VLDB JOURNAL 2020

SIMD HASH TABLE PROBING

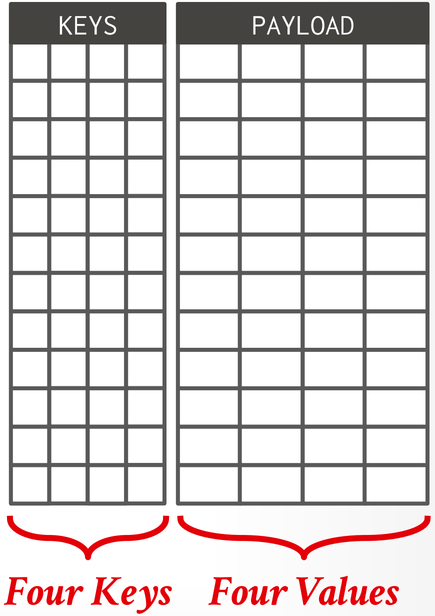
Scalar



Vectorized (Horizontal)

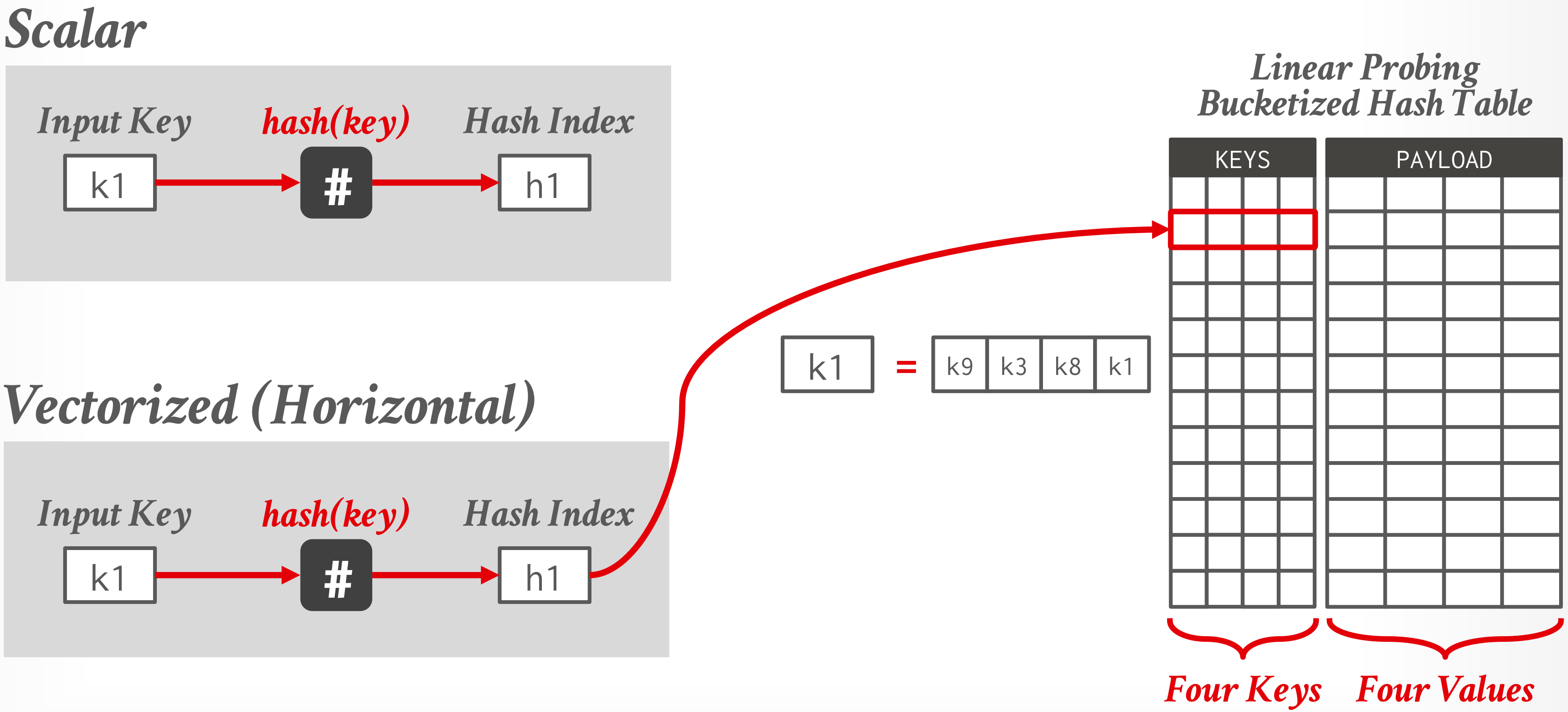


Linear Probing
Bucketized Hash Table



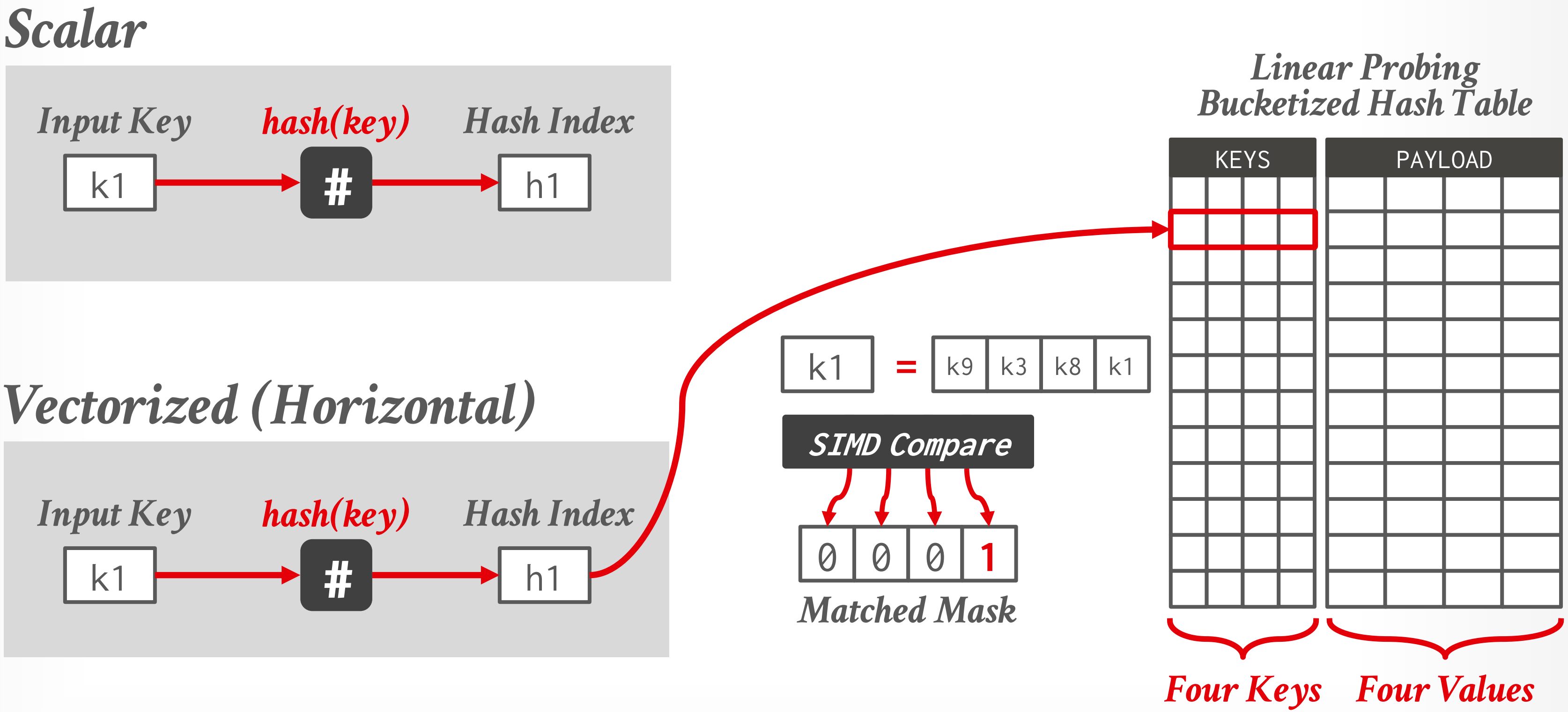
MAKE THE MOST OUT OF YOUR SIMD INVESTMENTS: COUNTER CONTROL FLOW DIVERGENCE IN COMPILED QUERY PIPELINES
VLDB JOURNAL 2020

SIMD HASH TABLE PROBING



MAKE THE MOST OUT OF YOUR SIMD INVESTMENTS: COUNTER CONTROL FLOW DIVERGENCE IN COMPILED QUERY PIPELINES
VLDB JOURNAL 2020

SIMD HASH TABLE PROBING



MAKE THE MOST OUT OF YOUR SIMD INVESTMENTS: COUNTER CONTROL FLOW DIVERGENCE IN COMPILED QUERY PIPELINES
VLDB JOURNAL 2020

FILTER REPRESENTATION

`WHERE col0 IS NULL OR col1 LIKE 'b%'`

Approach #1: Selection Vectors

- → Dense sorted list of tuple identifiers that indicate which tuples in a batch are valid.
- → Pre-allocate selection vector as the max-size of the input vector.

col0: int32

data	null?
55	0
66	0
77	0
-	1
88	0

col1: varchar

data	null?
aa	0
bb	0
-	1
cc	0
bbb	0

Selection Vector

offset
1
3
4

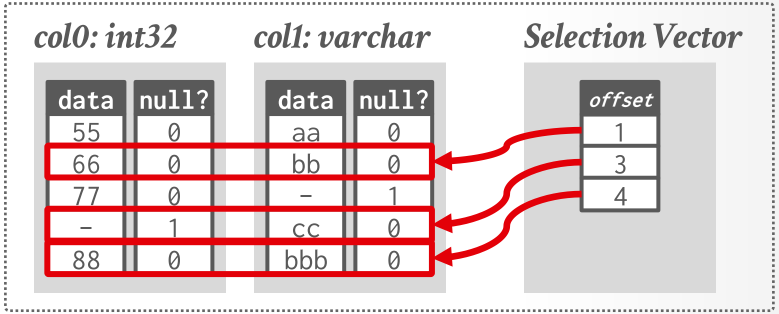
FILTER REPRESENTATION IN VECTORIZED QUERY EXECUTION
DAMON 2021

FILTER REPRESENTATION

```
WHERE col0 IS NULL OR col1 LIKE 'b%'
```

Approach #1: Selection Vectors

- → Dense sorted list of tuple identifiers that indicate which tuples in a batch are valid.
- → Pre-allocate selection vector as the max-size of the input vector.



FILTER REPRESENTATION IN VECTORIZED QUERY EXECUTION
DAMON 2021

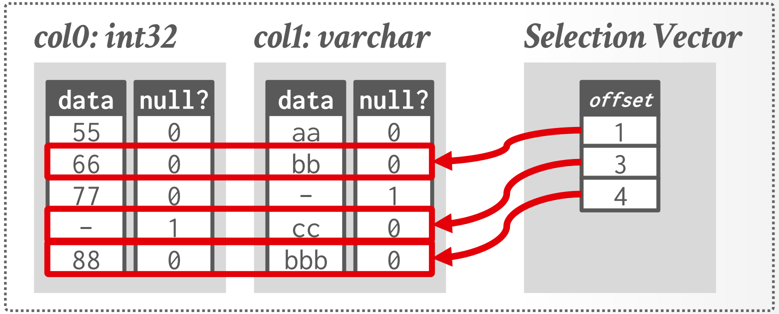
FILTER REPRESENTATION

```
WHERE col0 IS NULL OR col1 LIKE 'b%'
```

Approach #1: Selection Vectors

→ Dense sorted list of tuple identifiers that indicate which tuples in a batch are valid.

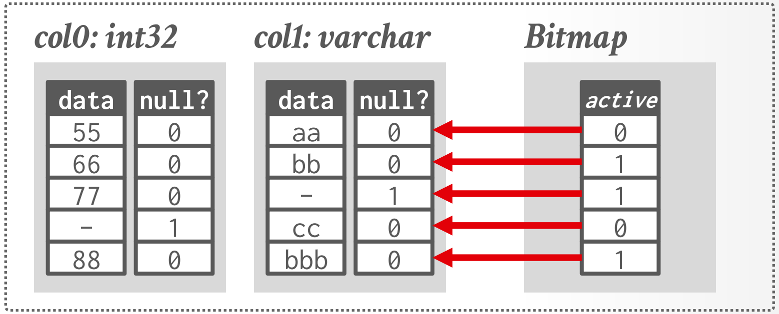
→ Pre-allocate selection vector as the max-size of the input vector.



Approach #2: Bitmaps

→ Positionally-aligned bitmap that indicates whether a tuple is valid at an offset.

→ Some SIMD instructions natively use these bitmaps as input masks.



FILTER REPRESENTATION IN VECTORIZED QUERY EXECUTION
DAMON 2021

HIQUE: HOLISTIC CODE GENERATION

For a given query plan, create a C/C++ program that implements that query's execution.

→ Bake in all the predicates and type conversions.

Use an off-shelf compiler to convert the code into a shared object, link it to the DBMS process, and then invoke the exec function.

GENERATING CODE FOR HOLISTIC QUERY EVALUATION
ICDE 2010

HIQUE: OPERATOR TEMPLATES

Interpreted Plan

```
for t in range(table.num_tuples):
  tuple = get_tuple(table, t)
  if eval(predicate, tuple, params):
    emit(tuple)
```

1. *Get schema in catalog for table.*
2. *Calculate offset based on tuple size.*
3. *Return pointer to tuple.*

HIQUE: OPERATOR TEMPLATES

Interpreted Plan

```
for t in range(table.num_tuples):
  tuple = get_tuple(table, t)
  if eval(predicate, tuple, params):
    emit(tuple)
```

1. *Get schema in catalog for table.*
2. *Calculate offset based on tuple size.*
3. *Return pointer to tuple.*

1. *Traverse predicate tree and pull values up.*
2. *If tuple value, calculate the offset of the target attribute.*
3. *Perform casting as needed for comparison operators.*
4. *Return true / false.*

HIQUE: OPERATOR TEMPLATES

Interpreted Plan

```
for t in range(table.num_tuples):
  tuple = get_tuple(table, t)
  if eval(predicate, tuple, params):
    emit(tuple)
```

1. *Get schema in catalog for table.*
2. *Calculate offset based on tuple size.*
3. *Return pointer to tuple.*

1. *Traverse predicate tree and pull values up.*
2. *If tuple value, calculate the offset of the target attribute.*
3. *Perform casting as needed for comparison operators.*
4. *Return true / false.*

Templated Plan

```
tuple_size = ###
predicate_offset = ###
parameter_value = ###

for t in range(table.num_tuples):
    tuple = table.data + t * tuple_size
    val = (tuple+predicate_offset)
    if (val == parameter_value + 1):
      emit(tuple)
```

HIQUE: OPERATOR TEMPLATES

Interpreted Plan

```
for t in range(table.num_tuples):
  tuple = get_tuple(table, t)
  if eval(predicate, tuple, params):
    emit(tuple)
```

1. *Get schema in catalog for table.*
2. *Calculate offset based on tuple size.*
3. *Return pointer to tuple.*

1. *Traverse predicate tree and pull values up.*
2. *If tuple value, calculate the offset of the target attribute.*
3. *Perform casting as needed for comparison operators.*
4. *Return true / false.*

Templated Plan

```
tuple_size = ###
predicate_offset = ###
parameter_value = ###

for t in range(table.num_tuples):
    tuple = table.data + t * tuple_size
    val = (tuple+predicate_offset)
    if (val == parameter_value + 1):
      emit(tuple)
```

HIQUE: OPERATOR TEMPLATES

Interpreted Plan

```
for t in range(table.num_tuples):
  tuple = get_tuple(table, t)
  if eval(predicate, tuple, params):
    emit(tuple)
```

1. *Get schema in catalog for table.*
2. *Calculate offset based on tuple size.*
3. *Return pointer to tuple.*

1. *Traverse predicate tree and pull values up.*
2. *If tuple value, calculate the offset of the target attribute.*
3. *Perform casting as needed for comparison operators.*
4. *Return true / false.*

Templated Plan

```
tuple_size = ###
predicate_offset = ###
parameter_value = ###

for t in range(table.num_tuples):
    tuple = table.data + t * tuple_size
    val = (tuple+predicate_offset)
    if (val == parameter_value + 1):
      emit(tuple)
```

VECTORWISE: PRECOMPILED PRIMITIVES

Pre-compiles thousands of "primitives" that perform basic operations on typed data.

→ Using simple kernels for each primitive means that they are easier to vectorize.

The DBMS then executes a query plan that invokes these primitives at runtime.

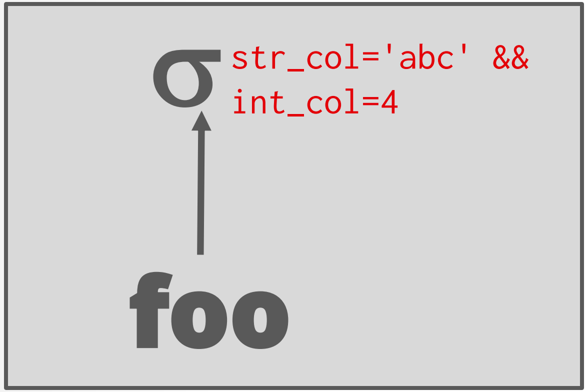
→ Function calls are amortized over multiple tuples.

→ The output of a primitive are the offsets of tuples that

MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

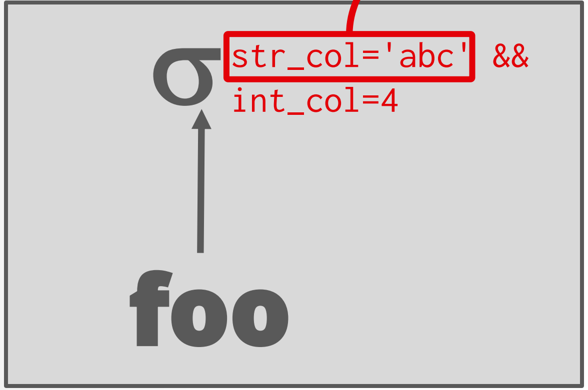
VECTORWISE: PRECOMPILED PRIMITIVES


```sql
SELECT * FROM foo
WHERE str_col = 'abc'
  AND int_col = 4;
```



VECTORWISE: PRECOMPILED PRIMITIVES

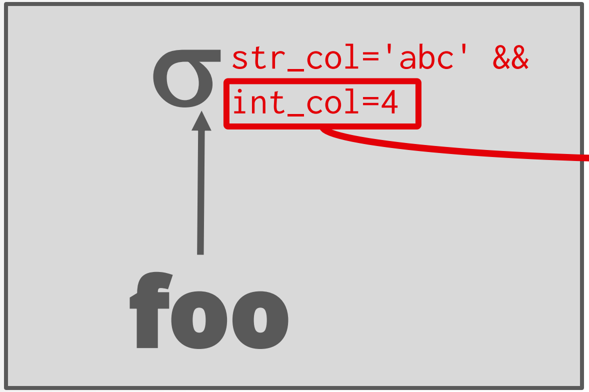
```sql
SELECT * FROM foo
WHERE str_col = 'abc'
  AND int_col = 4;
```




```
vec<offset> sel_eq_str(vec<string> col, string val) {
  vec<offset> positions;
  for (offset i = 0; i < col.size(); i++)
    if (col[i] == val) positions.append(i);
  return (positions);
}
```

VECTORWISE: PRECOMPILED PRIMITIVES

```sql
SELECT * FROM foo
WHERE str_col = 'abc'
  AND int_col = 4;
```

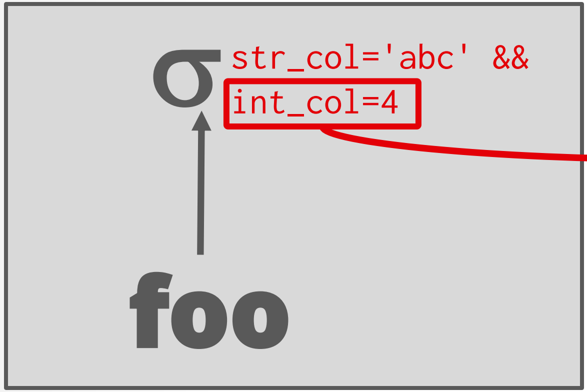



```
vec<offset> sel_eq_str(vec<string> col, string val) {
  vec<offset> positions;
  for (offset i = 0; i < col.size(); i++)
    if (col[i] == val) positions.append(i);
  return (positions);
}
```

```
vec<offset> sel_eq_int(vec<int> col, int val,
                       vec<offset> positions) {
  vec<offset> res;
  for (offset i : positions)
    if (col[i] == val) res.append(i);
  return (res);
}
```

VECTORWISE: PRECOMPILED PRIMITIVES

```
SELECT * FROM foo
WHERE str_col = 'abc'
  AND int_col = 4;
```




```
vec<offset> sel_eq_str(vec<string> col, string val) {
  vec<offset> positions;
  for (offset i = 0; i < col.size(); i++)
    if (col[i] == val) positions.append(i);
  return (positions);
}
```

```
vec<offset> sel_eq_int(vec<int> col, int val,
                       vec<offset> positions) {
  vec<offset> res;
  for (offset i : positions)
    if (col[i] == val) res.append(i);
  return (res);
}
```

SYSTEMS

Google BigQuery (2011)
Snowflake (2013)
Amazon Redshift (2014)
Yellowbrick (2014)
Databricks Photon (2022)
ClickHouse (2016)
⚡**DB Flash Talk: RelationalAI**

Google Big Query

GOOGLE BIGQUERY (2011)

Originally developed as "Dremel" in 2006 as a side-project for analyzing data artifacts generated from other tools.

→ The "interactive" goal means that they want to support ad hoc queries on **in-situ** data files.

→ Did not support joins in the first version.

Rewritten in the late 2010s to shared-disk architecture built on top of GFS.

Released as public commercial product (BigQuery) in 2012.

BIGQUERY: OVERVIEW

Shared-Disk / Disaggregated Storage

Vectorized Query Processing

Shuffle-based Distributed Query Execution

Columnar Storage
→ Zone Maps / Filters
→ Dictionary + RLE Compression
→ Only Allows "Search" Inverted Indexes

Hash Joins Only

Heuristic Optimizer + Adaptive Optimizations

DREMEL: A DECADE OF INTERACTIVE SQL ANALYSIS AT WEB SCALE
VLDB 2020

BIGQUERY: IN-MEMORY SHUFFLE

The shuffle phases represent checkpoints in a query's lifecycle where that the coordinator makes sure that all tasks are completed.

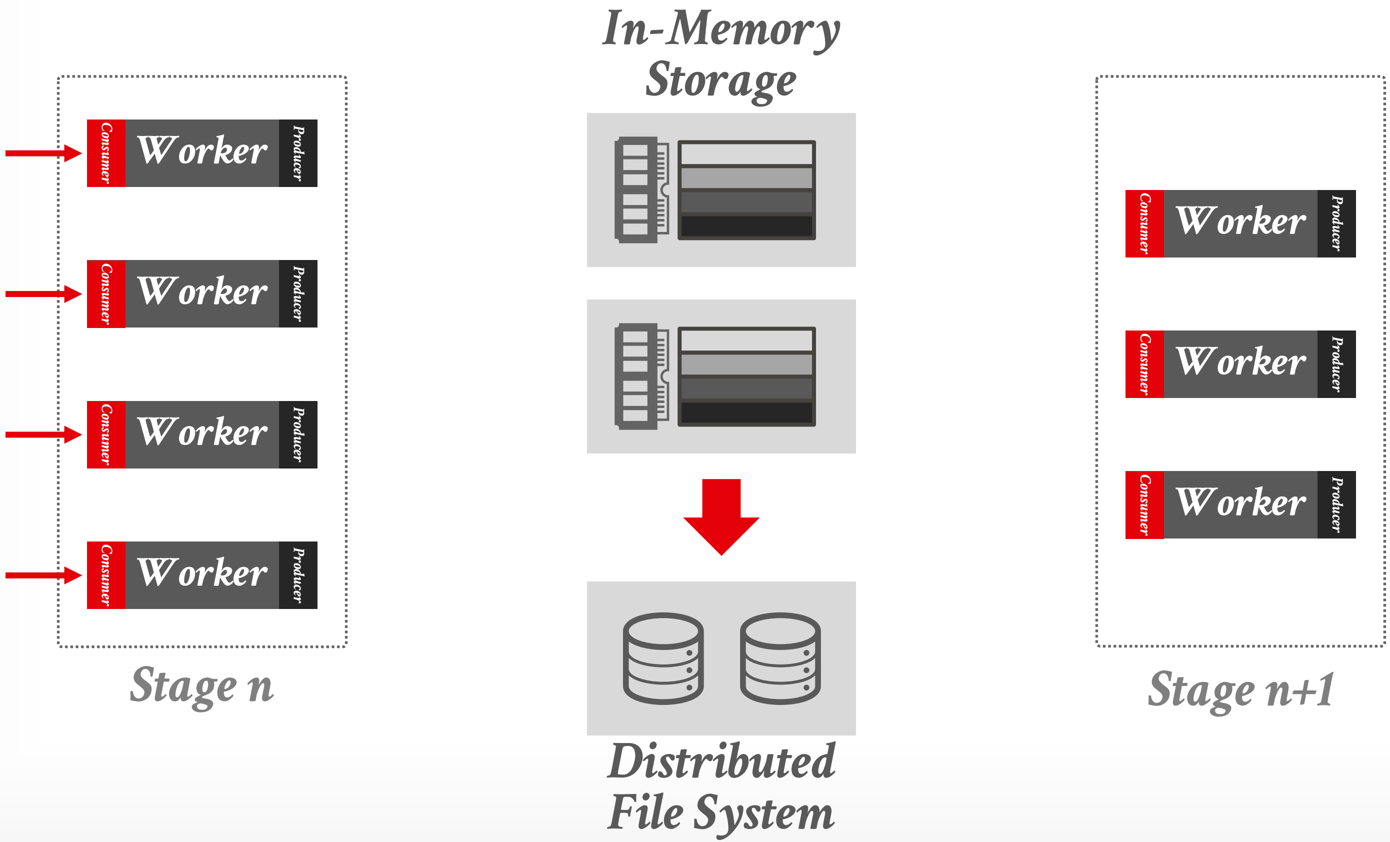
Fault Tolerance / Straggler Avoidance:

→ If a worker does not produce a task's results within a deadline, the coordinator speculatively executes a redundant task.

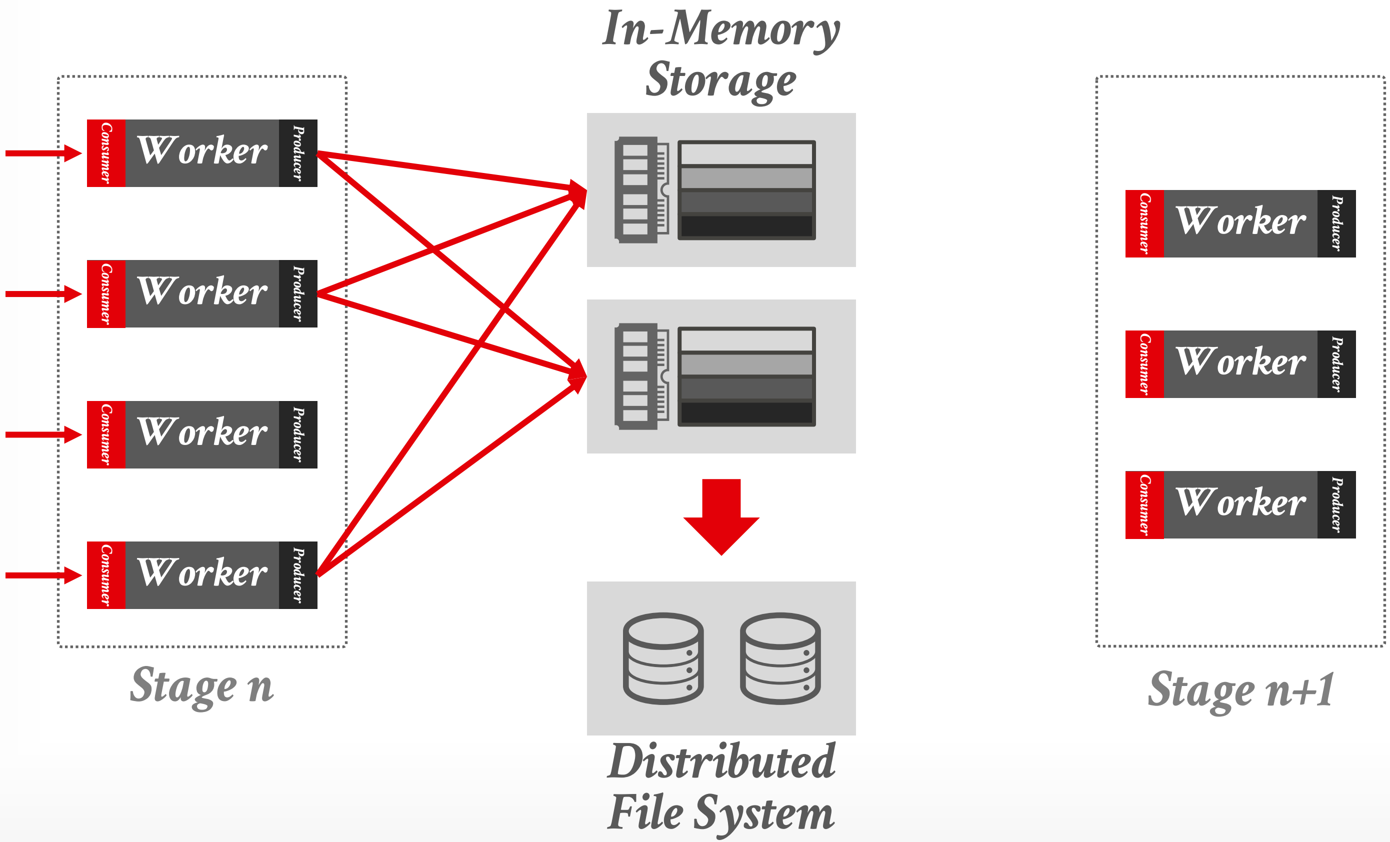
Dynamic Resource Allocation:

→ Scale up / down the number of workers for the next stage depending size of a stage's output.

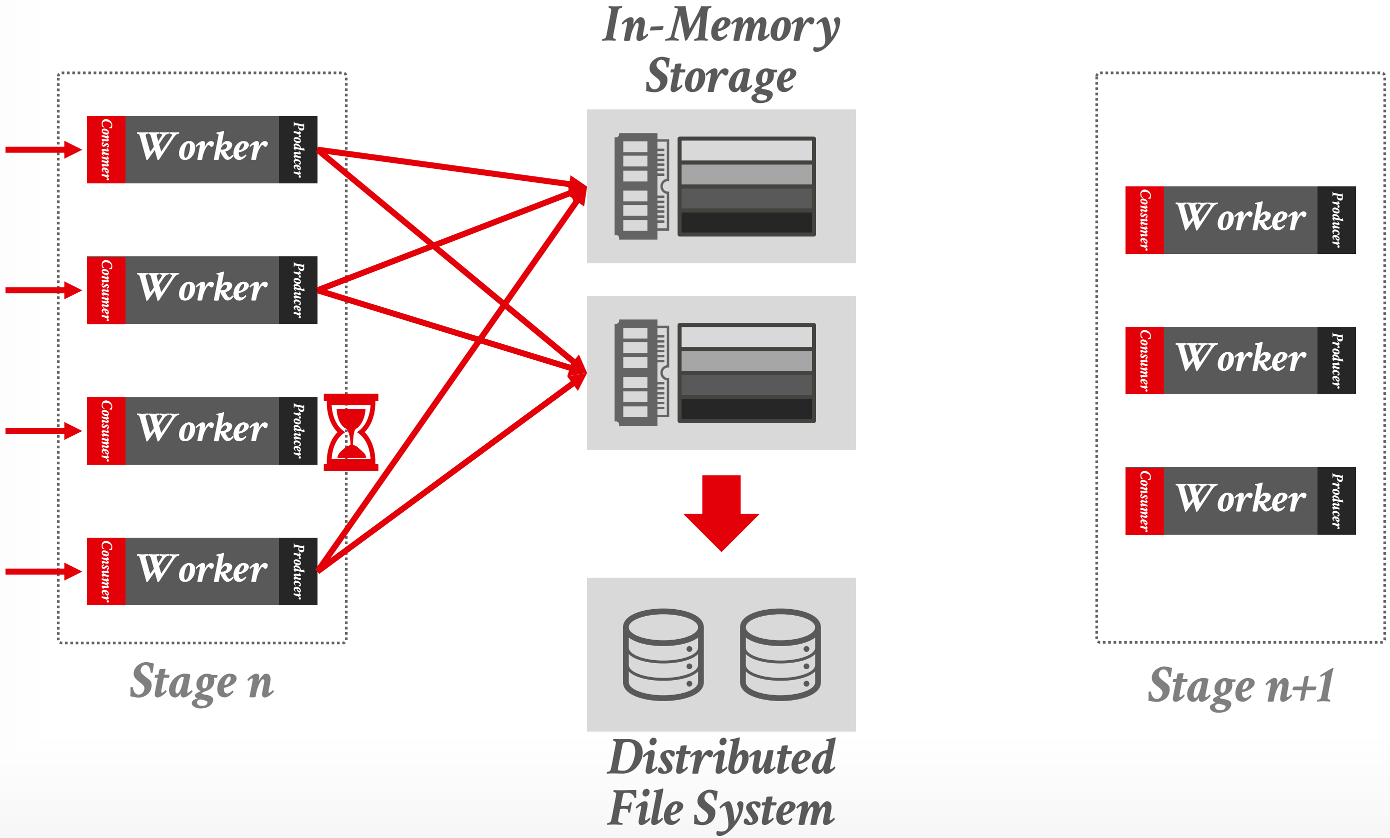
BIGQUERY: IN-MEMORY SHUFFLE



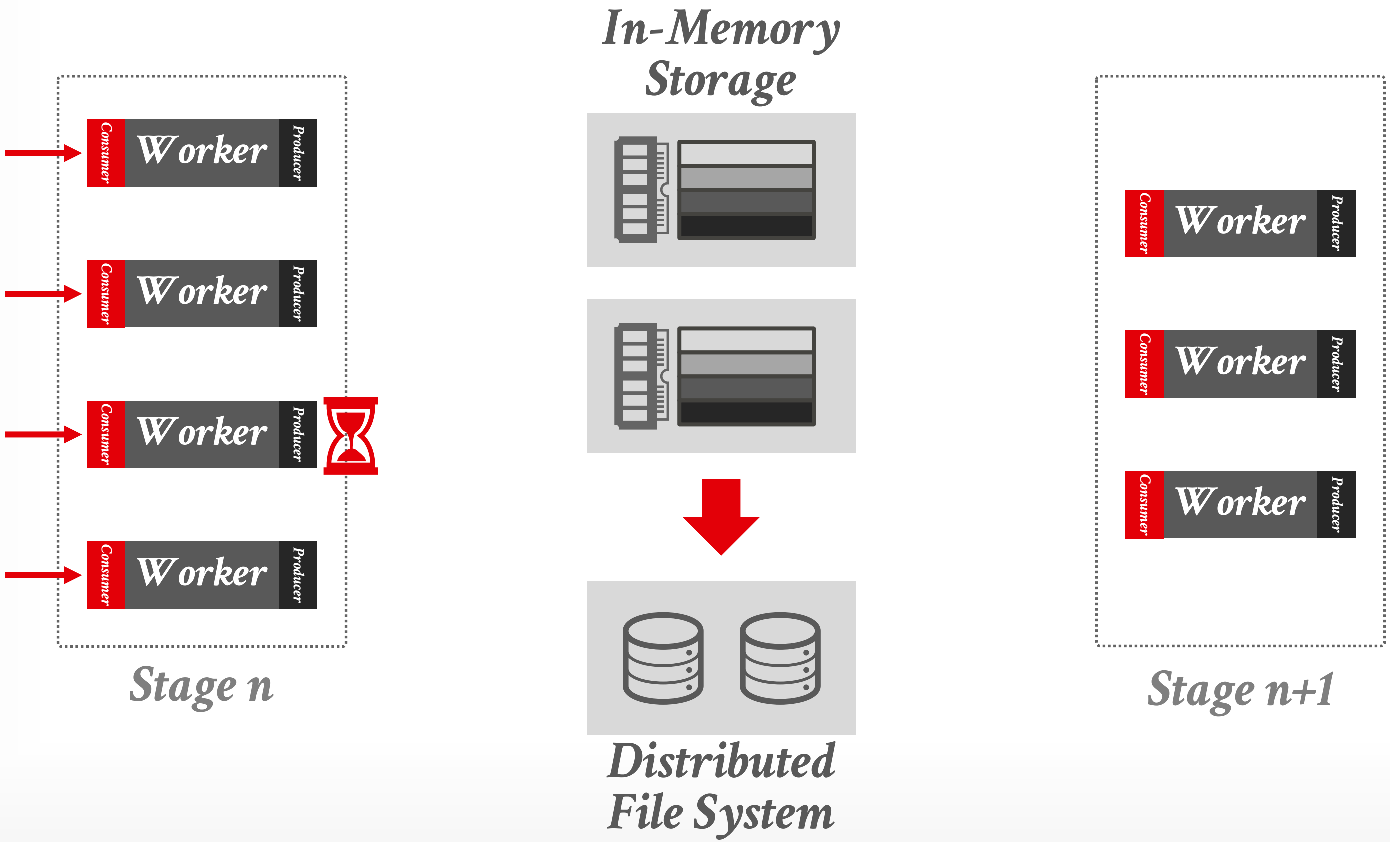
BIGQUERY: IN-MEMORY SHUFFLE



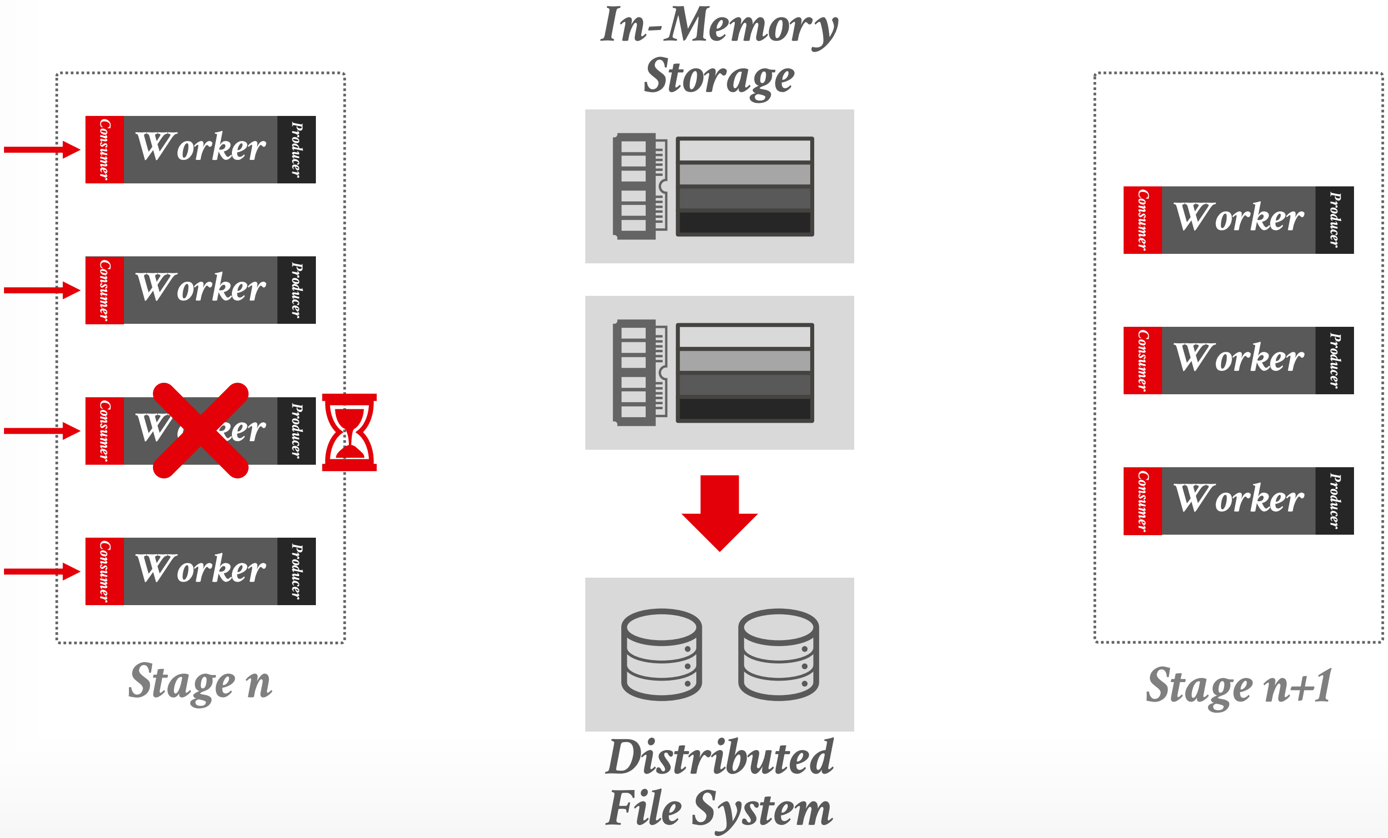
BIGQUERY: IN-MEMORY SHUFFLE



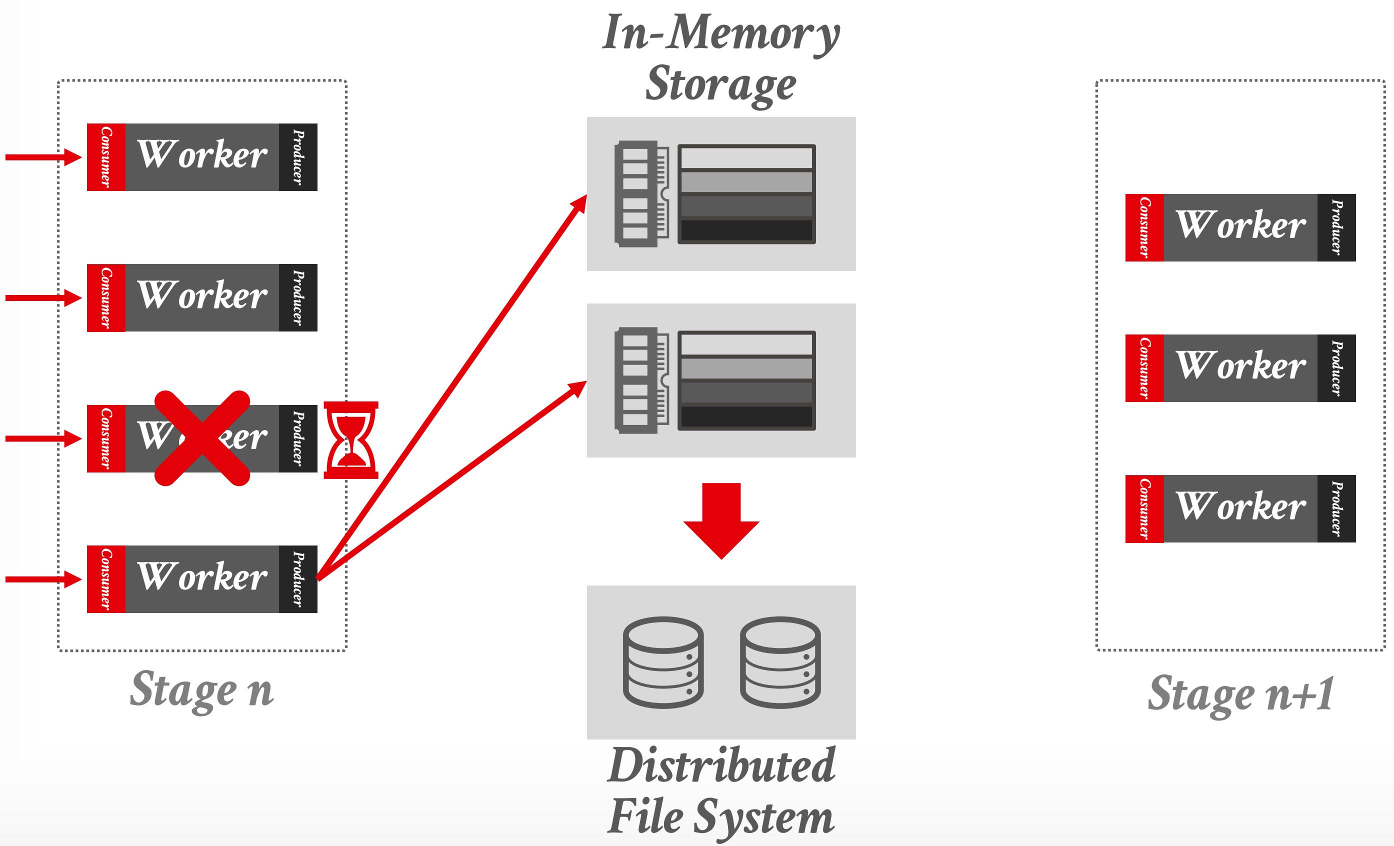
BIGQUERY: IN-MEMORY SHUFFLE



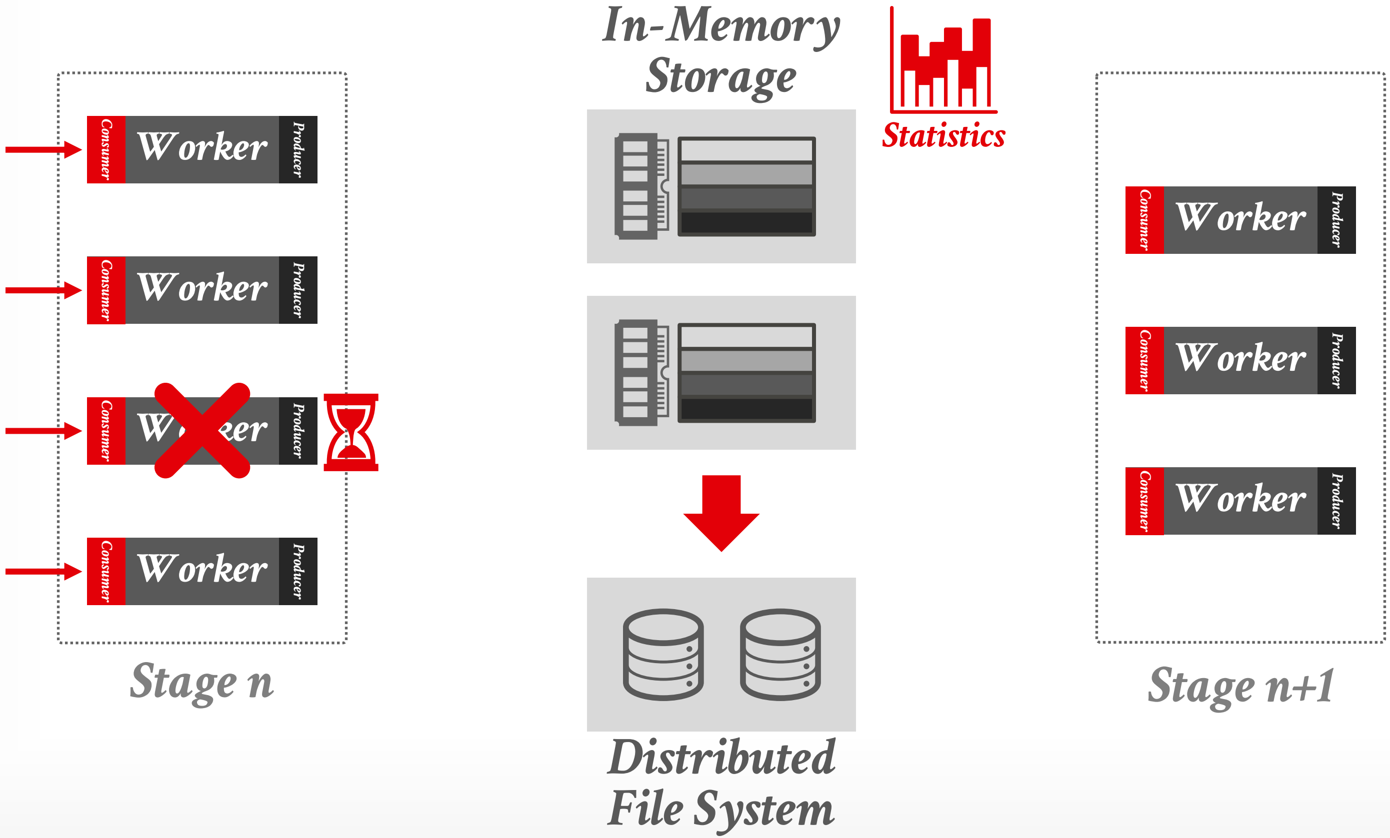
BIGQUERY: IN-MEMORY SHUFFLE



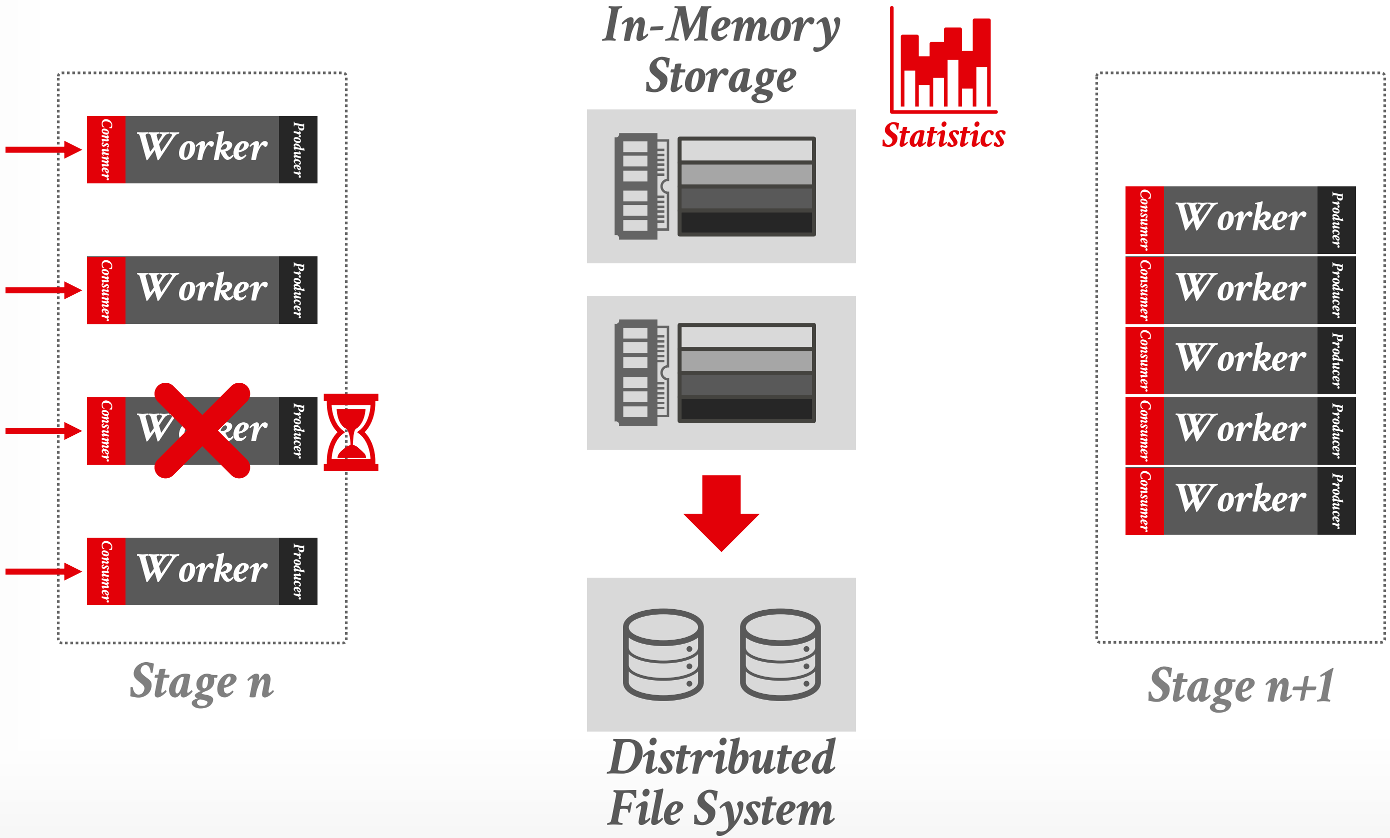
BIGQUERY: IN-MEMORY SHUFFLE



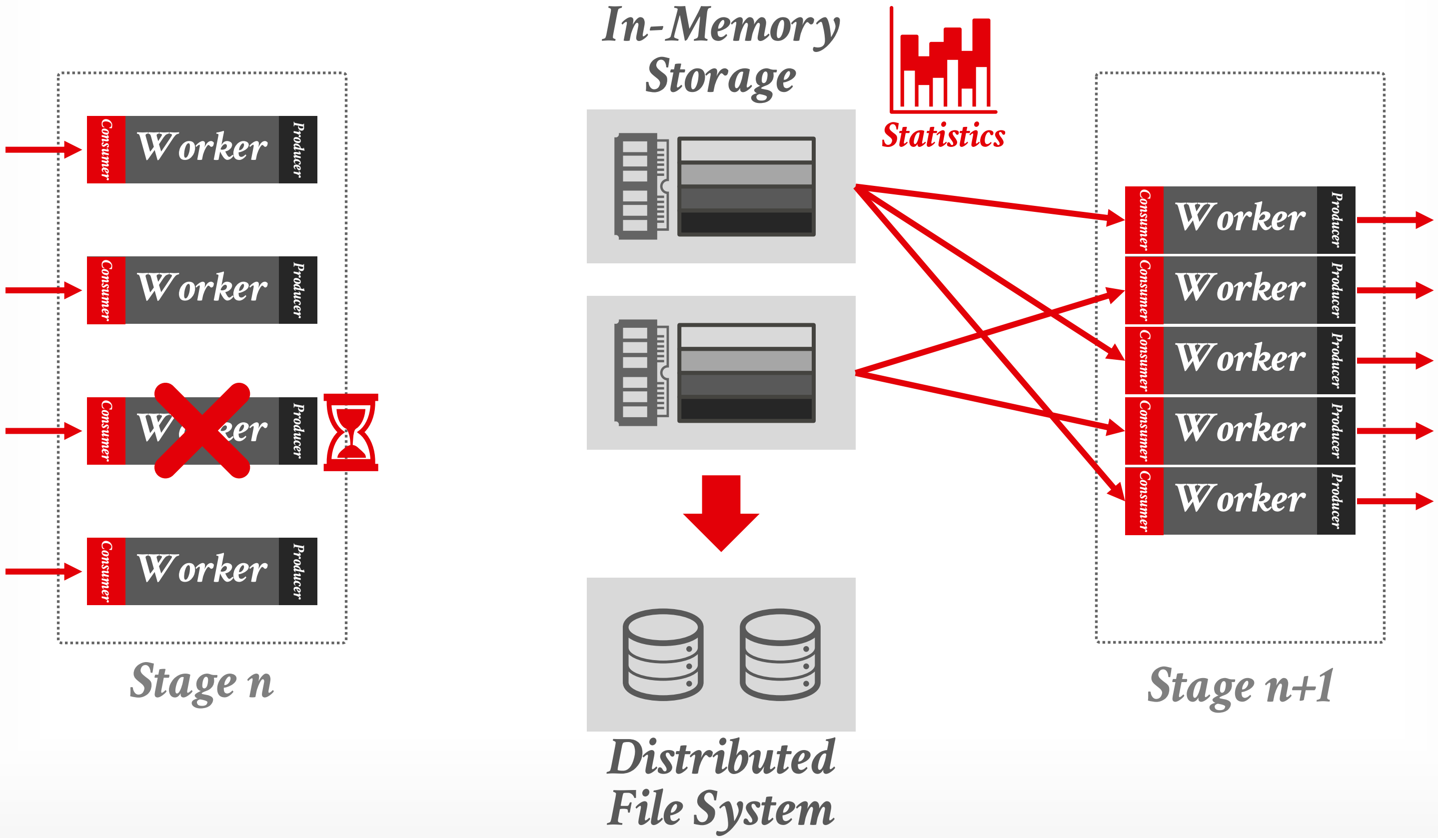
BIGQUERY: IN-MEMORY SHUFFLE



BIGQUERY: IN-MEMORY SHUFFLE



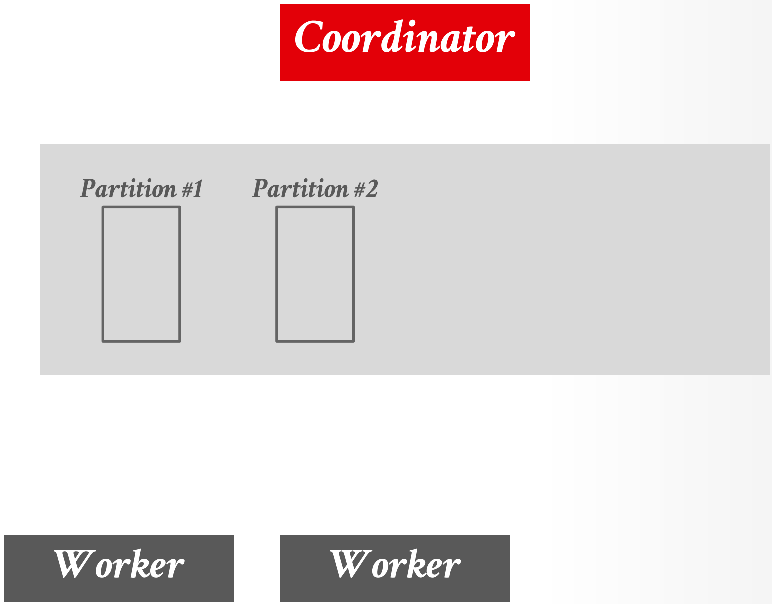
BIGQUERY: IN-MEMORY SHUFFLE



BIGQUERY: DYNAMIC REPARTITIONING

BigQuery dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.

DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.

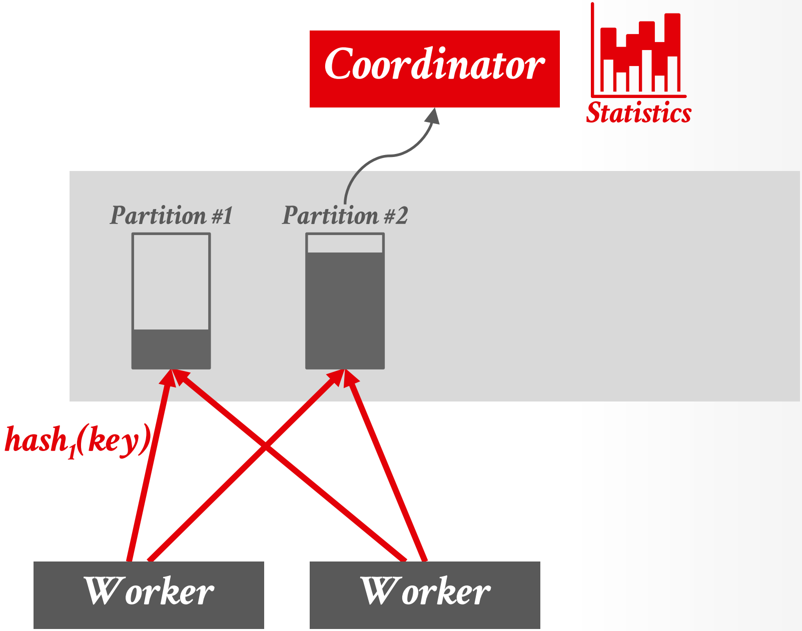


Source: H.Ahmadi + A.Surna

BIGQUERY: DYNAMIC REPARTITIONING

BigQuery dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.

DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.

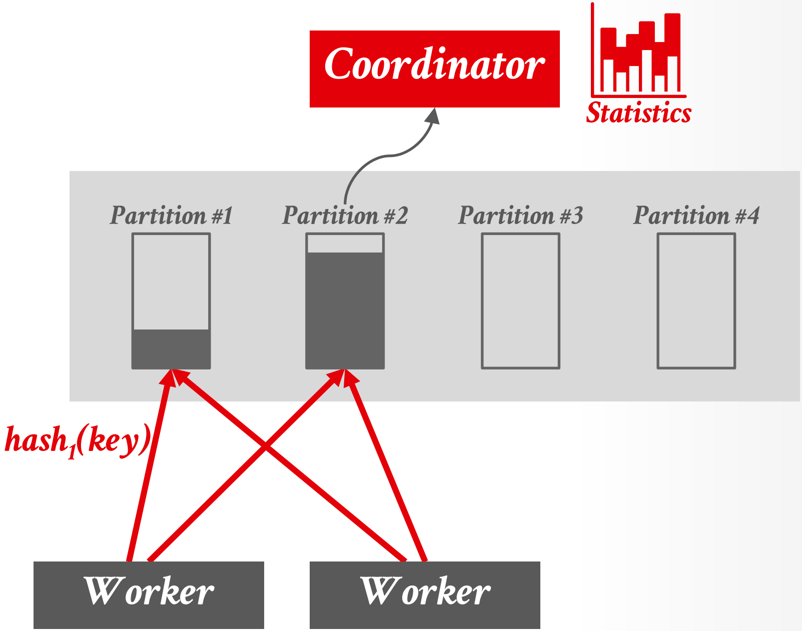


Source: H.Ahmadi + A.Surna

BIGQUERY: DYNAMIC REPARTITIONING

BigQuery dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.

DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.

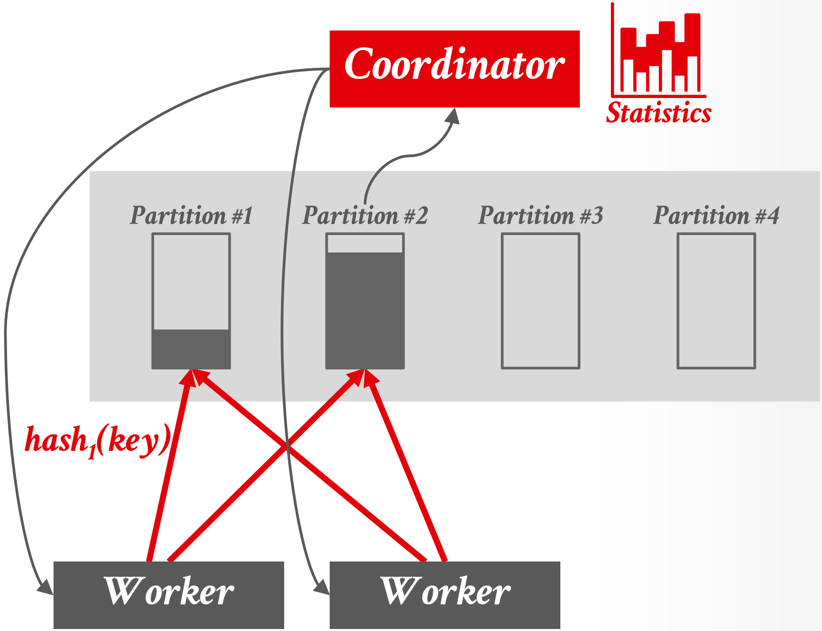


Source: H.Ahmadi + A.Surna

BIGQUERY: DYNAMIC REPARTITIONING

BigQuery dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.

DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.

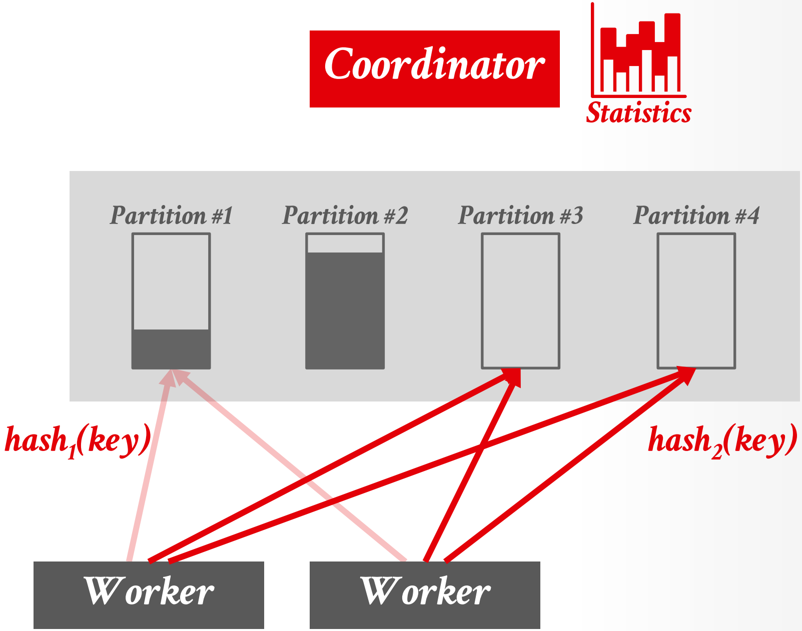


Source: H.Ahmadi + A.Surna

BIGQUERY: DYNAMIC REPARTITIONING

BigQuery dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.

DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.

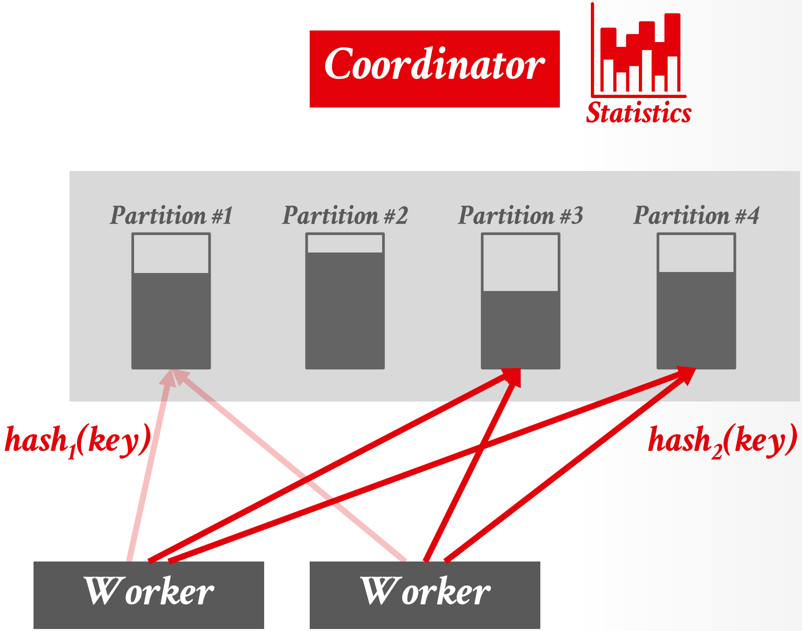


Source: H.Ahmadi + A.Surna

BIGQUERY: DYNAMIC REPARTITIONING

BigQuery dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.

DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.

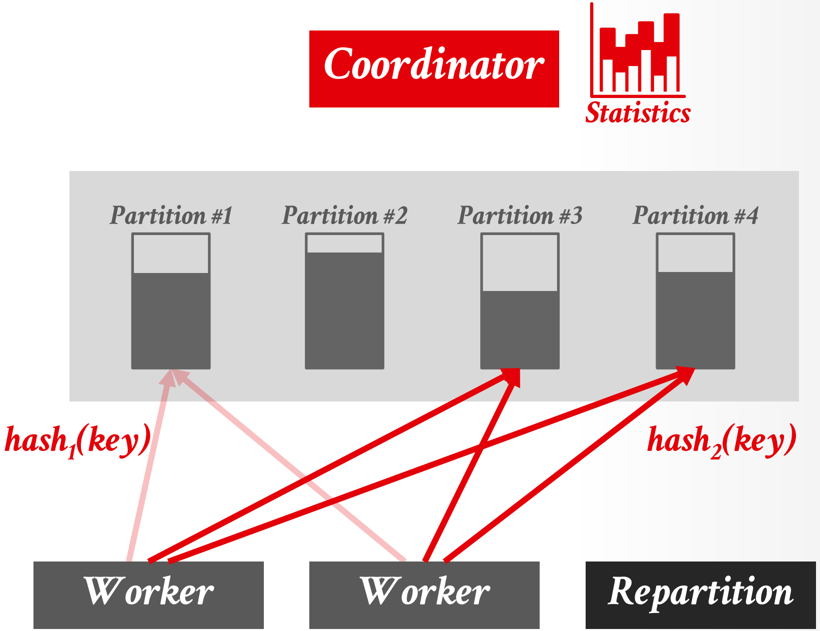


Source: H.Ahmadi + A.Surna

BIGQUERY: DYNAMIC REPARTITIONING

BigQuery dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.

DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.

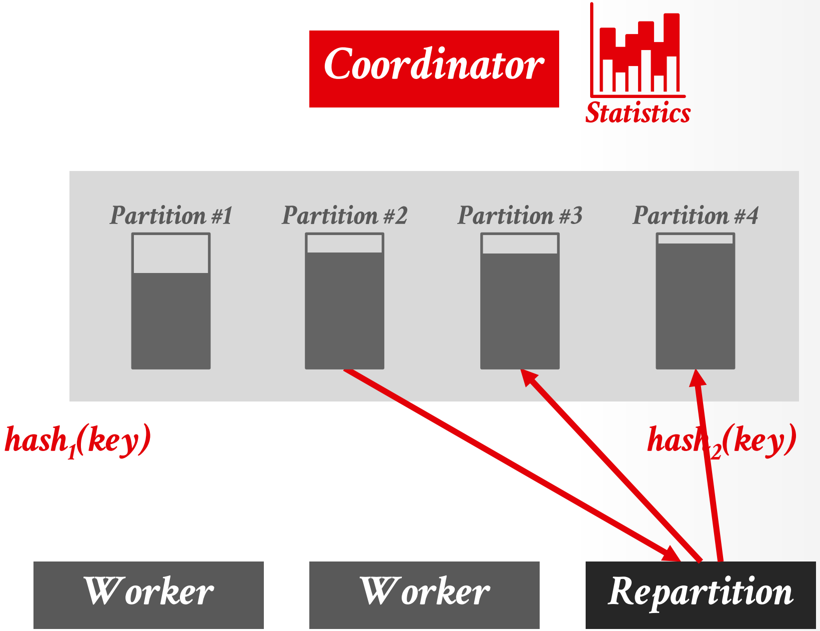


Source: H.Ahmadi + A.Surna

BIGQUERY: DYNAMIC REPARTITIONING

BigQuery dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.

DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.

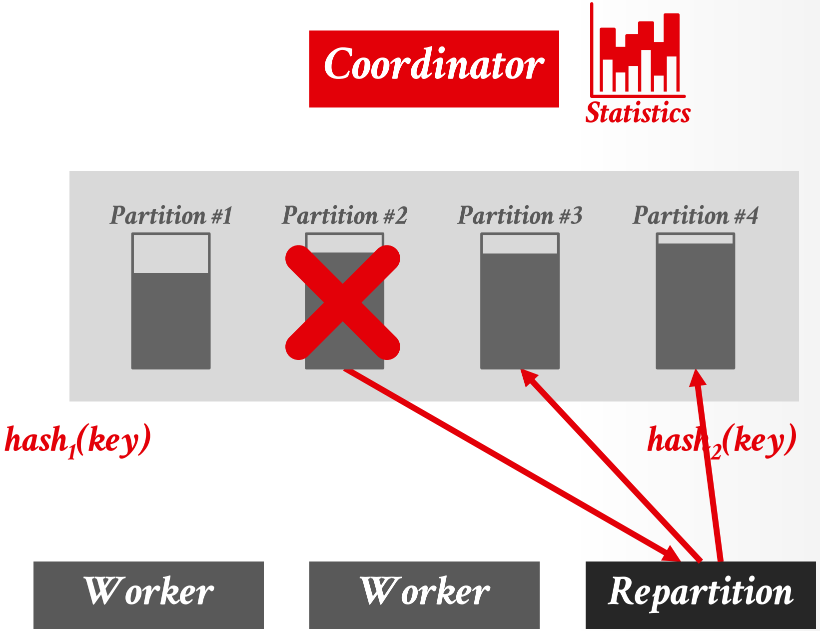


Source: H.Ahmadi + A.Surna

BIGQUERY: DYNAMIC REPARTITIONING

BigQuery dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.

DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.

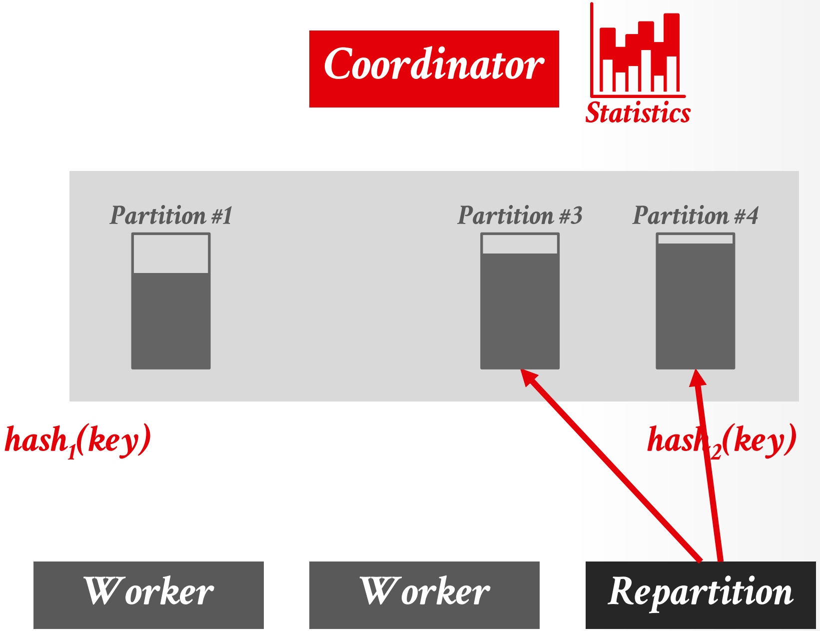


Source: H.Ahmadi + A.Surna

BIGQUERY: DYNAMIC REPARTITIONING

BigQuery dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.

DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.



Source: H.Ahmadi + A.Surna

snowflake

SNOWFLAKE (2013)

Managed OLAP DBMS written in C++.

→ Shared-disk architecture with aggressive compute-side local caching.

→ Written from scratch. Did not borrow components from existing systems.

→ Custom SQL dialect and client-server network protocols.

The OG cloud-native data warehouse.

THE SNOWFLAKE ELASTIC DATA WAREHOUSE
SIGMOD 2016

snowflake

SNOWFLAKE: OVERVIEW

Cloud-native OLAP DBMS written in C++

Shared-Disk / Disaggregated Storage

Push-based Vectorized Query Processing

Precompiled Operator Primitives

Separate Table Data from Meta-Data

No Buffer Pool

PAX Columnar Storage

SNOWFLAKE: QUERY PROCESSING

Snowflake is a push-based vectorized engine that uses precompiled primitives for operator kernels.

→ Pre-compile variants using C++ templates for different vector data types.

→ Only uses codegen (via LLVM) for tuple serialization/deserialization between workers.

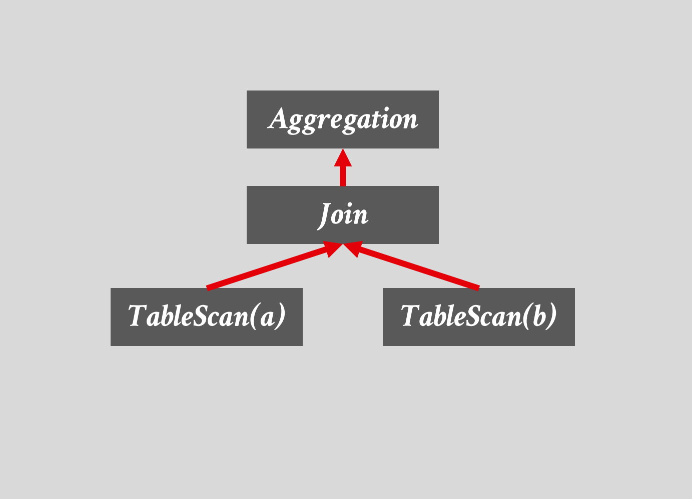
Does not support partial query retries

→ If a worker fails, then the entire query has to restart.

SNOWFLAKE: ADAPTIVE OPTIMIZATION

After determining join ordering, Snowflake's optimizer identifies aggregation operators to push down into the plan below joins.

The optimizer adds the downstream aggregations but then the DBMS only enables them at runtime according to statistics observed during execution.

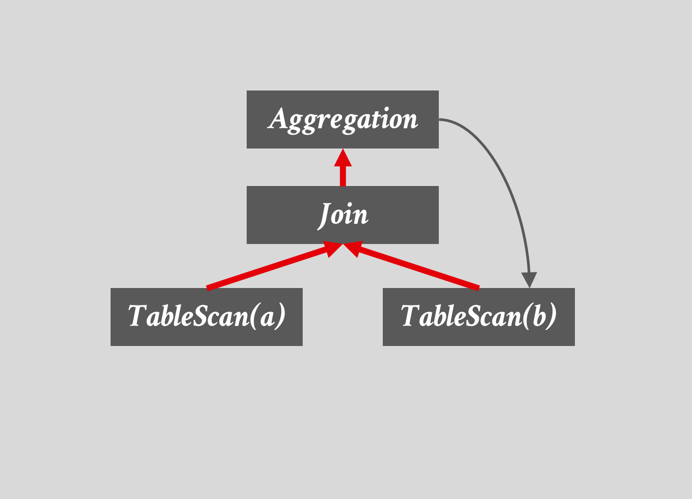


Source: Bowei Chen

SNOWFLAKE: ADAPTIVE OPTIMIZATION

After determining join ordering, Snowflake's optimizer identifies aggregation operators to push down into the plan below joins.

The optimizer adds the downstream aggregations but then the DBMS only enables them at runtime according to statistics observed during execution.

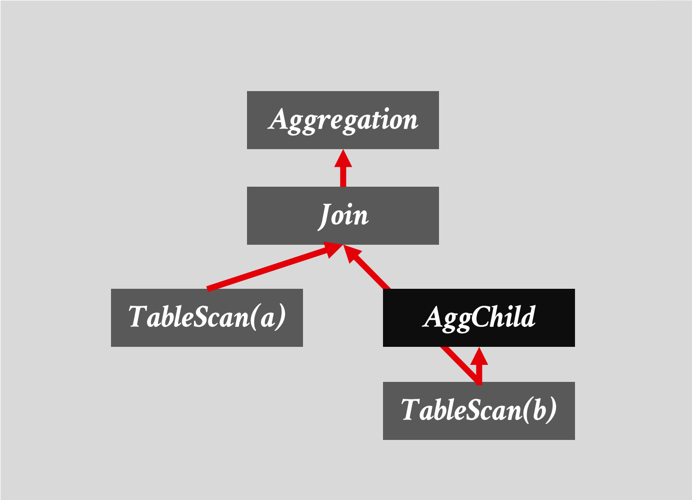


Source: Bowei Chen

SNOWFLAKE: ADAPTIVE OPTIMIZATION

After determining join ordering, Snowflake's optimizer identifies aggregation operators to push down into the plan below joins.

The optimizer adds the downstream aggregations but then the DBMS only enables them at runtime according to statistics observed during execution.

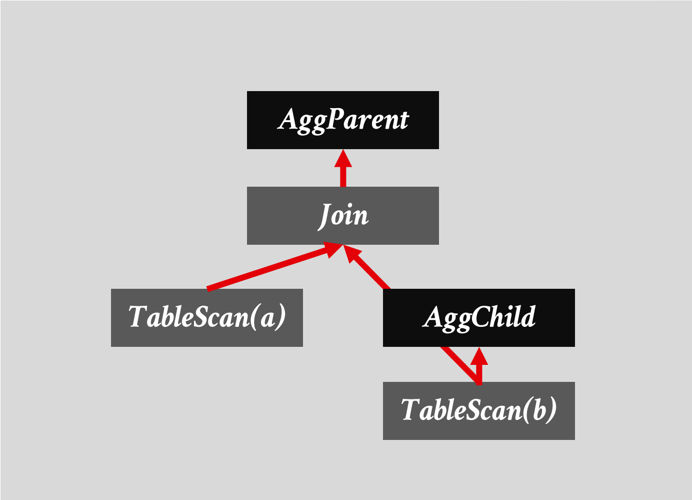


Source: Bowei Chen

SNOWFLAKE: ADAPTIVE OPTIMIZATION

After determining join ordering, Snowflake's optimizer identifies aggregation operators to push down into the plan below joins.

The optimizer adds the downstream aggregations but then the DBMS only enables them at runtime according to statistics observed during execution.



Source: Bowei Chen

SNOWFLAKE: A

After determining join orderi
Snowflake's optimizer identif
aggregation operators to pusl
into the plan below joins.

The optimizer adds the dow
aggregations but then the DI
enables them at runtime acc
statistics observed during ex

Medium Search Write

Aggregation Placement — An Adaptive Query Optimization for Snowflake

Bowei Chen · Follow
Published in Snowflake · 8 min read · Aug 10, 2023

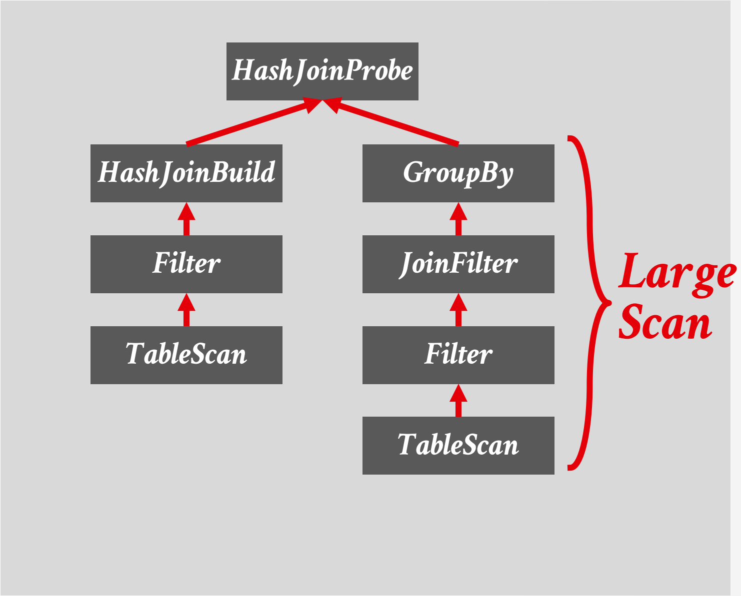
Snowflake's Data Cloud is backed by a data platform designed from the ground up to leverage cloud computing technology. The platform is delivered as a fully managed service, providing a user-friendly experience to run complex analytical workloads easily and efficiently without the burden of managing on-premise infrastructure. Snowflake's architecture separates the compute layer from the storage layer. Compute workloads on the same dataset can scale independently and run in isolation without interfering with each other, and compute resources could be allocated and scaled on demand within seconds. The cloud-native architecture makes Snowflake a powerful platform for data warehousing, data engineering, data science, and many other types of applications. More about Snowflake architecture can be found in Key Concepts & Architecture documentation and the Snowflake Elastic Data Warehouse research paper.

Source: Bowei Chen

Snowflake: Flexible Compute

If a query plan fragment will process a large amount of data, then the DBMS can temporarily deploy additional worker nodes to accelerate its performance.

Flexible compute worker nodes write results to storage as if it was a table.



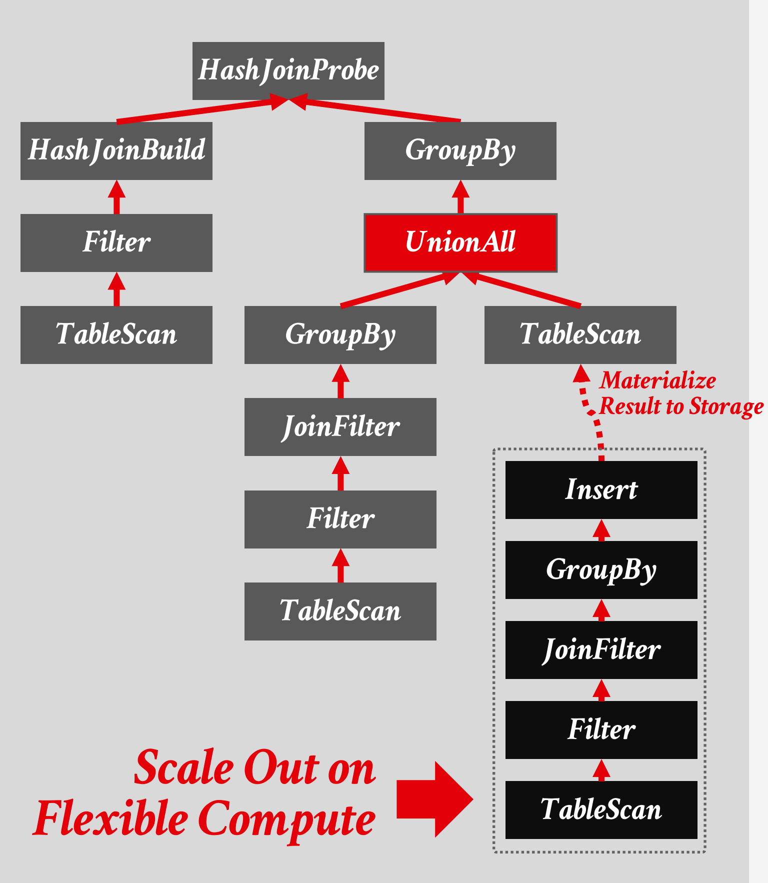
Source: Libo Wang

snowflake

SNOWFLAKE: FLEXIBLE COMPUTE

If a query plan fragment will process a large amount of data, then the DBMS can temporarily deploy additional worker nodes to accelerate its performance.

Flexible compute worker nodes write results to storage as if it was a table.



Source: Libo Wang

amazon
REDSHIFT

AMAZON REDSHIFT (2014)

Amazon's flagship OLAP DBaaS.

→ Based on ParAccel's original shared-nothing architecture.
→ Switched to support disaggregated storage (S3) in 2017.
→ Added serverless deployments in 2022.

Redshift is a more traditional data warehouse compared to BigQuery/Spark where it wants to control all the data.

Overarching design goal is to remove as much administration + configuration choices from users.

AMAZON REDSHIFT RE-INVENTED
SIGMOD 2022

REDSHIFT: OVERVIEW

Shared-Disk / Disaggregated Storage
Push-based Vectorized Query Processing
Transpilation Query Codegen (C++)
Precompiled Primitives
Compute-side Caching
PAX Columnar Storage
Sort-Merge + Hash Joins
Hardware Acceleration (AQUA)
Stratified Query Optimizer

REDSHIFT: COMPILATION SERVICE

Separate nodes to compile query plans using GCC and aggressive caching.

→ DBMS checks whether a compiled version of each templated fragment already exists in customer's local cache.

→ If fragment does not exist in the local cache, then it checks a global cache for the **entire** fleet of Redshift customers.

Background workers proactively recompile plans when new version of DBMS is released.

Yellowbrick

YELLOWBRICK (2014)

OLAP DBMS written on C++ and derived from a hardfork of PostgreSQL v9.5.

→ Uses PostgreSQL's front-end (networking, parser, catalog) to handle incoming SQL requests.
→ They hate the OS as much as I do.

Originally started as an on-prem appliance with FPGA acceleration. Switched to DBaaS in 2021.

Cloud-version uses Kubernetes for all components.

YELLOWBRICK: AN ELASTIC DATA WAREHOUSE ON KUBERNETES
CIDR 2024

YELLOWBRICK

Shared-Disk / Disaggregated Storage

Push-based Vectorized Query Processing

Transpilation Query Codegen (C++)

Compute-side Caching

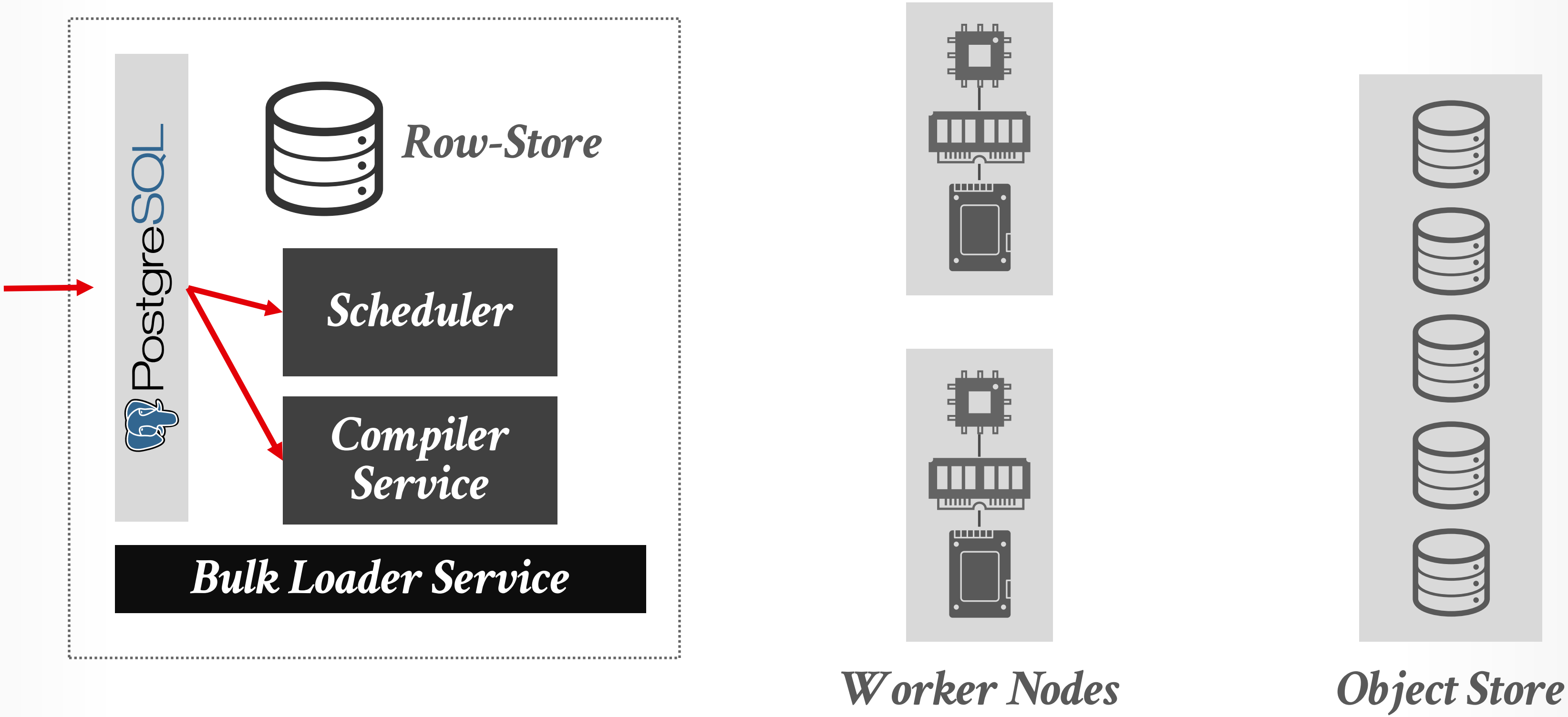
Separate Row + PAX Columnar Storage

Sort-Merge + Hash Joins

PostgreSQL Query Optimizer++

Insane Systems Engineering

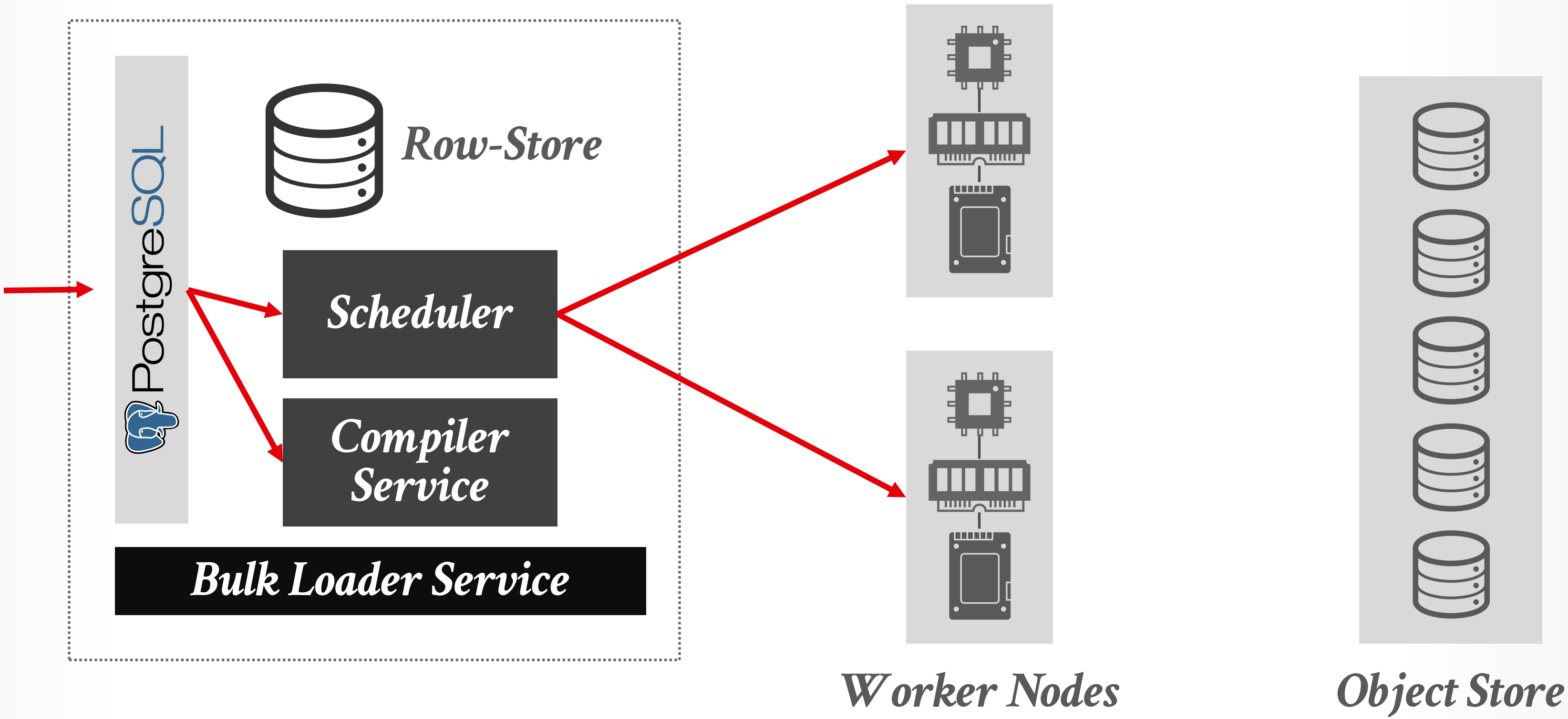
YELLOWBRICK: ARCHITECTURE



Source: Mark Cusack

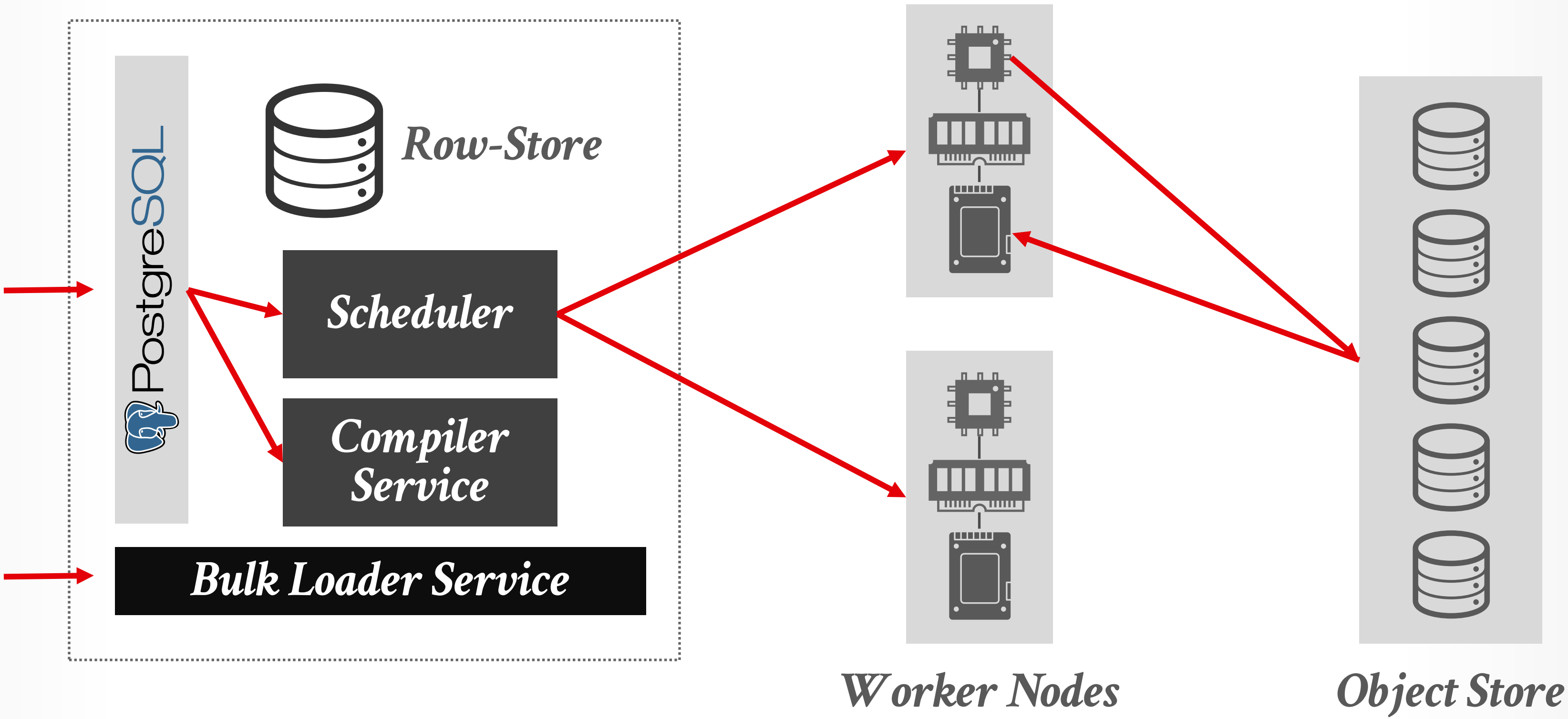
Yellowbrick

YELLOWBRICK: ARCHITECTURE



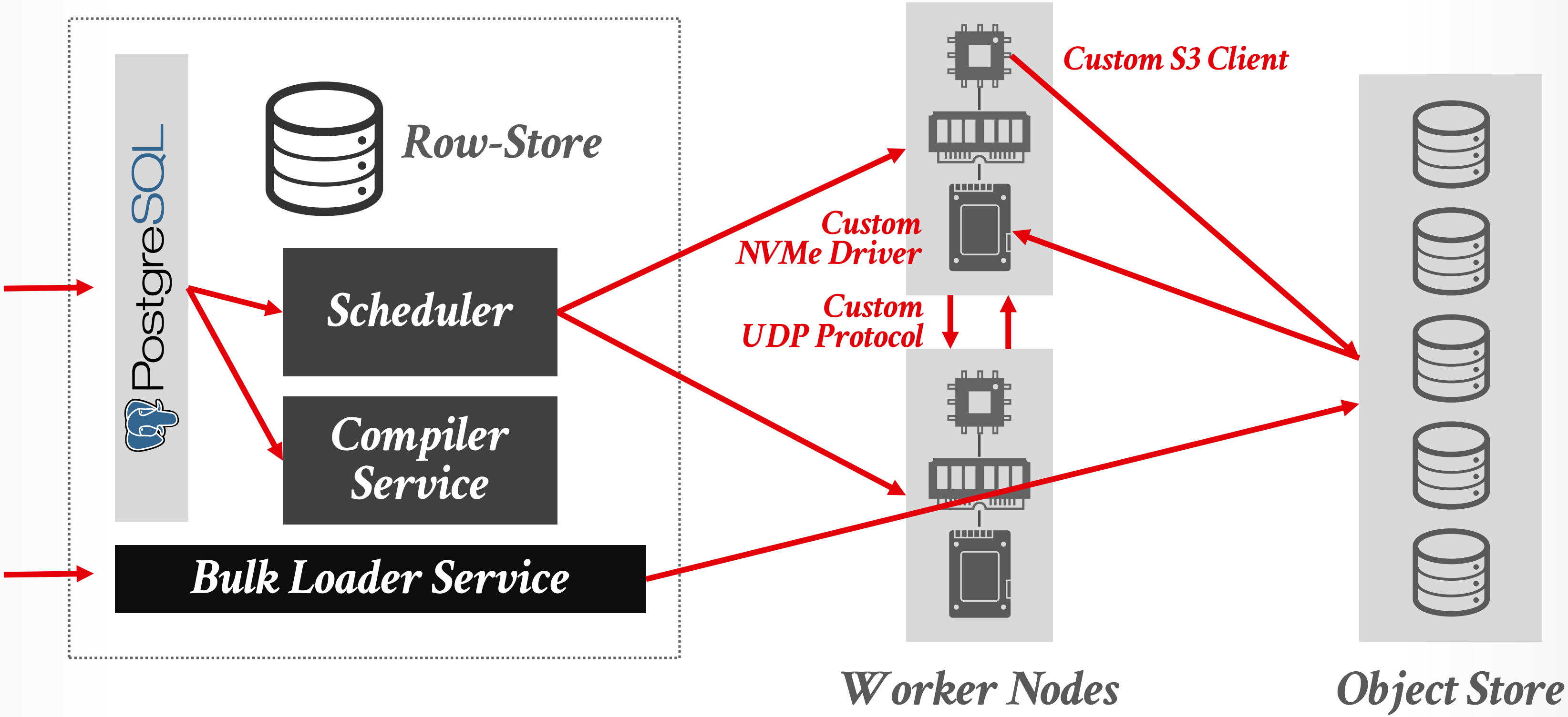
Source: Mark Cusack

YELLOWBRICK: ARCHITECTURE



Source: Mark Cusack

YELLOWBRICK: ARCHITECTURE



Source: Mark Cusack

YELLOWBRICK: QUERY EXECUTION

Pushed-based vectorized query processing that supports both row- and columnar-oriented data with early materialization.

→ Introduces transpose operators to convert data back and forth between row and columnar formats.

Holistic query compilation via source-to-source transpilation.

Yellowbrick's architecture goal is for workers to always process data residing in the CPU's L3 cache and not memory.

Yellowbrick: Memory Allocator

Custom NUMA-aware, latch-free allocator that gets all the memory needed upfront at start-up

→ Using **mmap** with **mlock** with huge pages.
→ Allocations are grouped by query to avoid fragmentation.
→ Claims their allocator is 100x faster than libc **malloc**.

Each worker also has a buffer pool manager that uses MySQL-style approximate LRU-K to store cached data files.

YELLOWBRICK: DEVICE DRIVERS

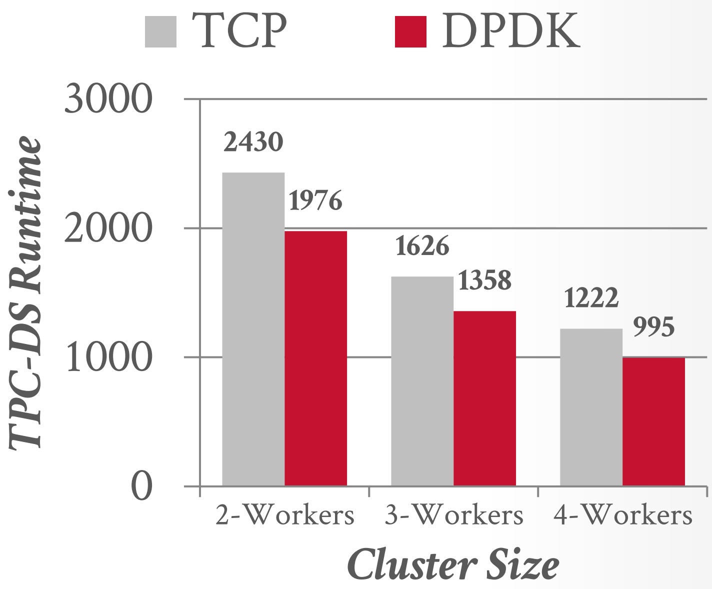
Custom NVMe / NIC drivers that run in user-space to avoid memory copy overheads.

→ Falls back to Linux drivers if necessary.

Custom reliable UDP network protocol with kernel-bypass (DPDK) for internal communication.

→ Each CPU has its own receive/transmit queues that it polls asynchronously.

→ Only sends data to a "partner" CPU at other workers.



databricks

databricks

DATABRICKS PHOTON (2022)

Single-threaded C++ execution engine embedded into **Databricks Runtime** (DBR) via JNI.

→ Overrides existing engine when appropriate.
→ Support both Spark's earlier SQL engine and Spark's DataFrame API.
→ Seamlessly handle impedance mismatch between row-oriented DBR and column-oriented Photon.

Accelerate execution of query plans over "raw / uncurated" files in a data lake.

PHOTON: A FAST QUERY ENGINE FOR LAKEHOUSE SYSTEMS
SIGMOD 2022

databricks

DATABRICKS PHOTON (2022)

Photon: A Fast Query Engine for Lakehouse Systems

Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman van Hovell, Maryann Xue, Reynold Xin, Matei Zaharia
photon-paper-authors@databricks.com
Databricks Inc.

ABSTRACT

Many organizations are shifting to a data management paradigm called the "Lakehouse," which implements the functionality of structured data warehouses on top of unstructured data lakes. This

from SQL to machine learning. Traditionally, for the most demanding SQL workloads, enterprises have also moved a curated subset of their data into data warehouses to get high performance, governance and concurrency. However, this two-tier architecture is

PHOTON: A FAST QUERY ENGINE FOR LAKEHOUSE SYSTEMS
SIGMOD 2022

PHOTON: OVERVIEW

Shared-Disk / Disaggregated Storage

Pull-based Vectorized Query Processing

Precompiled Primitives + Expression Fusion

Shuffle-based Distributed Query Execution

Sort-Merge + Hash Joins

Unified Query Optimizer + Adaptive Optimizations

PHOTON: VECTORIZED PROCESSING

Photon is a pull-based vectorized engine that uses precompiled **operator kernels** (primitives).

→ Converts physical plan into a list of pointers to functions that perform low-level operations on column batches.

Databricks: It is easier to build/maintain a vectorized engine than a JIT engine.

→ Engineers spend more time creating specialized codepaths to get closer to JIT performance.

→ With codegen, engineers write tooling and observability hooks instead of writing the engine.

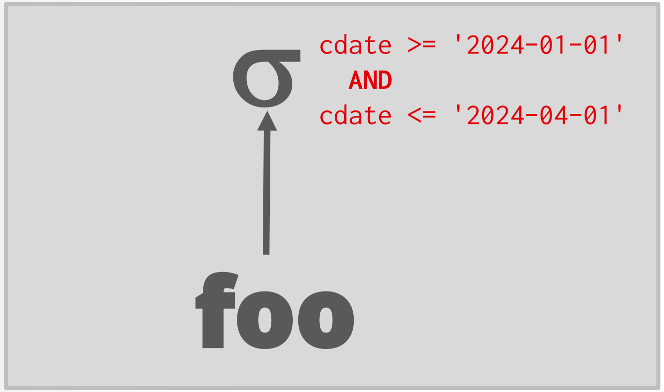
databricks

PHOTON: EXPRESSION FUSION


```
SELECT * FROM foo
 WHERE cdate BETWEEN '2024-01-01' AND '2024-04-01';
```

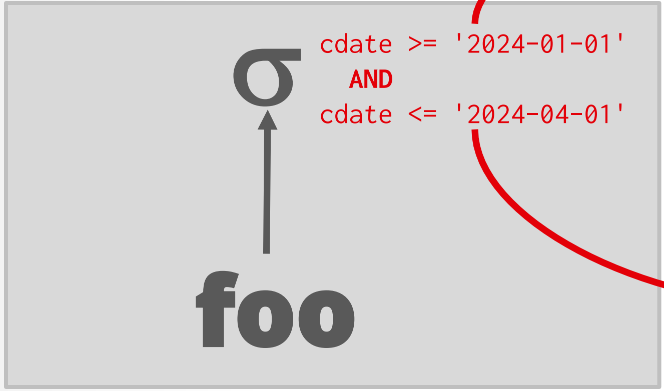
PHOTON: EXPRESSION FUSION


```sql
SELECT * FROM foo
 WHERE cdate >= '2024-01-01'
   AND cdate <= '2024-04-01';
```



PHOTON: EXPRESSION FUSION

```sql
SELECT * FROM foo
 WHERE cdate >= '2024-01-01'
   AND cdate <= '2024-04-01';
```

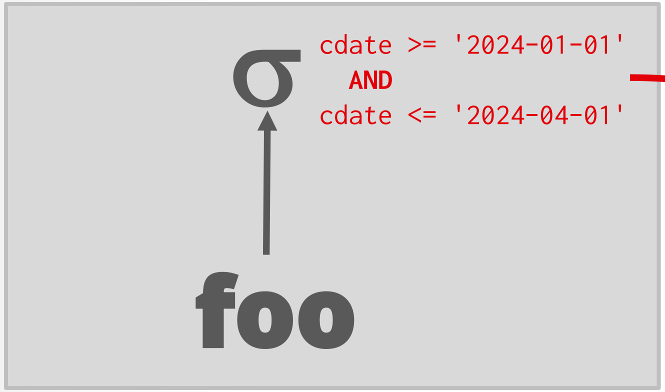



```cpp
vec<offset> sel_geq_date(vec<date> batch, date val) {
  vec<offset> positions;
  for (offset i = 0; i < batch.size(); i++)
    if (batch[i] >= val) positions.append(i);
  return (positions);
}
```

```cpp
vec<offset> sel_leq_date(vec<date> batch, date val) {
  vec<offset> positions;
  for (offset i = 0; i < batch.size(); i++)
    if (batch[i] <= val) positions.append(i);
  return (positions);
}
```

PHOTON: EXPRESSION FUSION

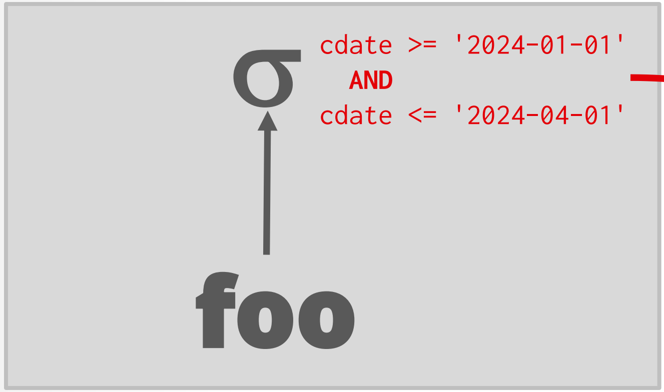

```sql
SELECT * FROM foo
 WHERE cdate >= '2024-01-01'
   AND cdate <= '2024-04-01';
```




```
vec<offset> sel_between_dates(vec<date> batch,
                              date low, date high) {
  vec<offset> positions;
  for (offset i = 0; i < batch.size(); i++)
    if (batch[i] >= low && batch[i] <= high)
      positions.append(i);
  return (positions);
}
```

PHOTON: EXPRESSION FUSION


```sql
SELECT * FROM foo
 WHERE cdate >= '2024-01-01'
   AND cdate <= '2024-04-01';
```



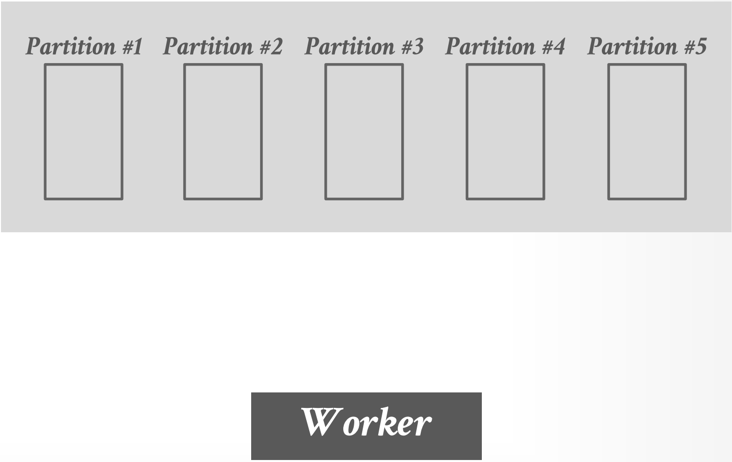

```
vec<offset> sel_between_dates(vec<date> batch,
                              date low, date high) {
  vec<offset> positions;
  for (offset i = 0; i < batch.size(); i++)
    if (batch[i] >= low && batch[i] <= high)
      positions.append(i);
  return (positions);
}
```

SPARK: PARTITION COALESCING

Spark (over-)allocates a large number of shuffle partitions for each stage.

→ Number needs to be large enough to avoid one partitioning from filling up too much.

After the shuffle completes, the DBMS then combines underutilized partitions using heuristics.

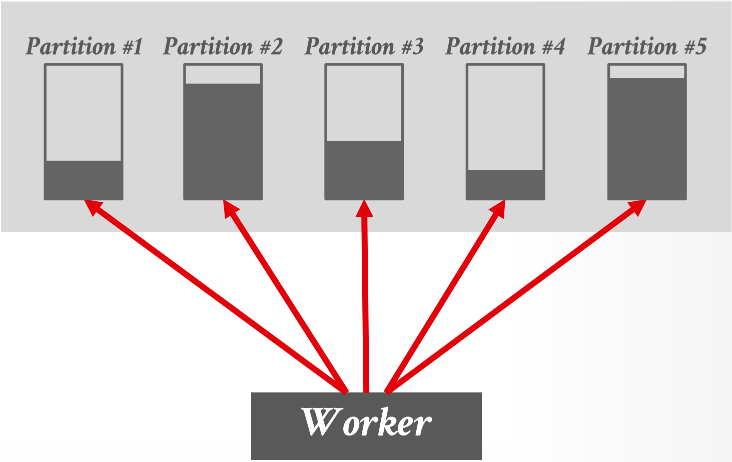


Source: Maryann Xue

Spark: Partition Coalescing

Spark (over-)allocates a large number of shuffle partitions for each stage.
→ Number needs to be large enough to avoid one partitioning from filling up too much.

After the shuffle completes, the DBMS then combines underutilized partitions using heuristics.



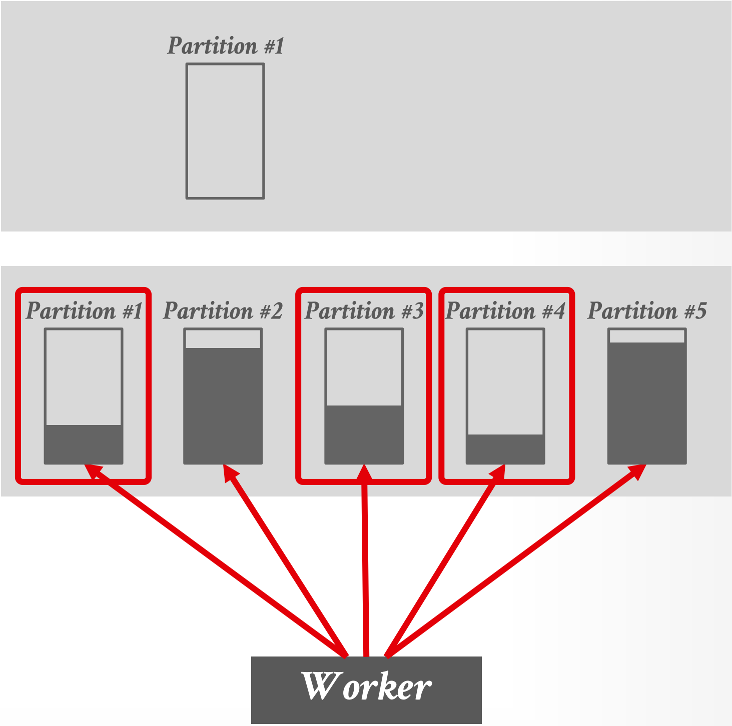
Source: Maryann Xue

Spark: Partition Coalescing

Spark (over-)allocates a large number of shuffle partitions for each stage.

→ Number needs to be large enough to avoid one partitioning from filling up too much.

After the shuffle completes, the DBMS then combines underutilized partitions using heuristics.

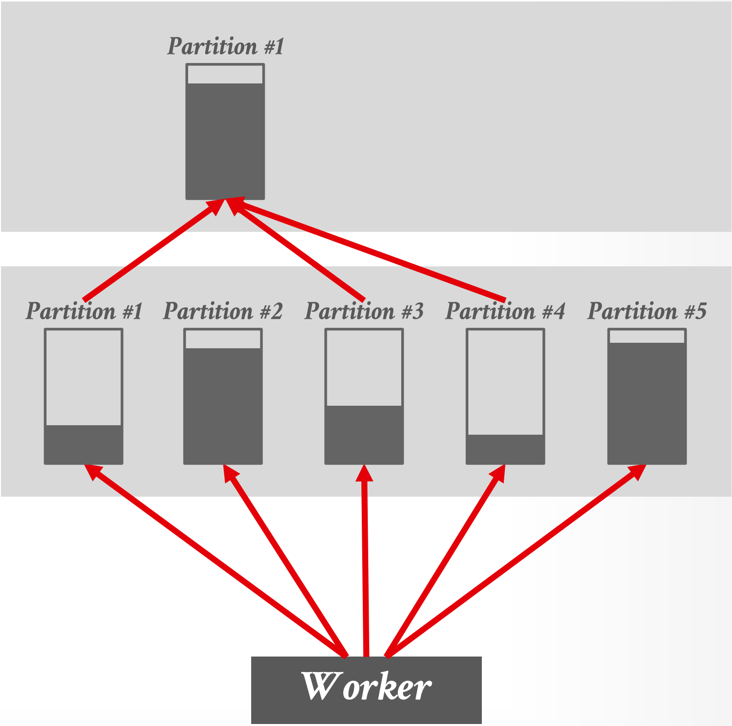


Source: Maryann Xue

SPARK: PARTITION COALESCING

Spark (over-)allocates a large number of shuffle partitions for each stage.
→ Number needs to be large enough to avoid one partitioning from filling up too much.

After the shuffle completes, the DBMS then combines underutilized partitions using heuristics.



Source: Maryann Xue

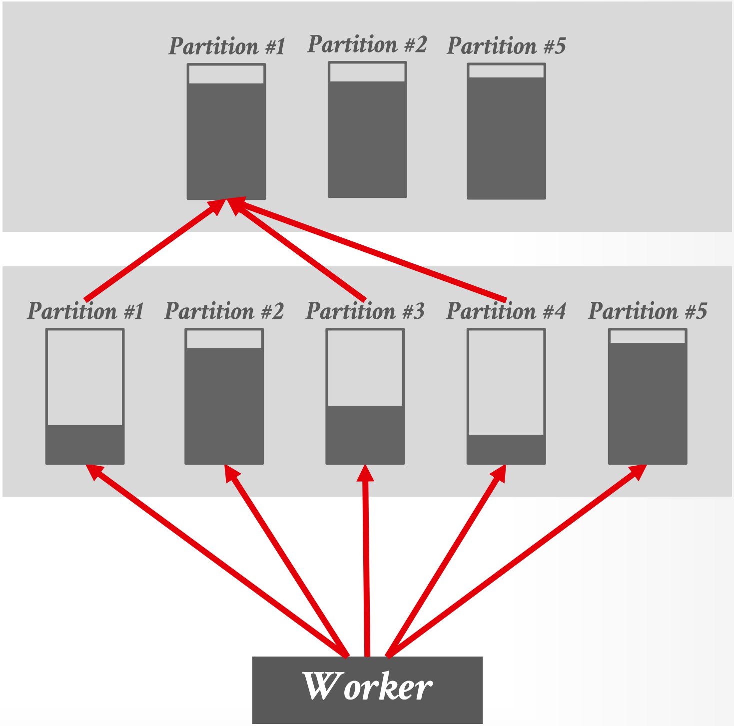
databricks

SPARK: PARTITION COALESCING

Spark (over-)allocates a large number of shuffle partitions for each stage.

→ Number needs to be large enough to avoid one partitioning from filling up too much.

After the shuffle completes, the DBMS then combines underutilized partitions using heuristics.



Source: Maryann Xue

ClickHouse

ClickHouse (2016)

C++ OLAP DBMS that supports different table engines
→ Default: MergeTree with SSTable-like immutable files

Shared-Nothing Architecture

Pull-Based Vectorized Query Processing

Operator-at-a-Time Execution

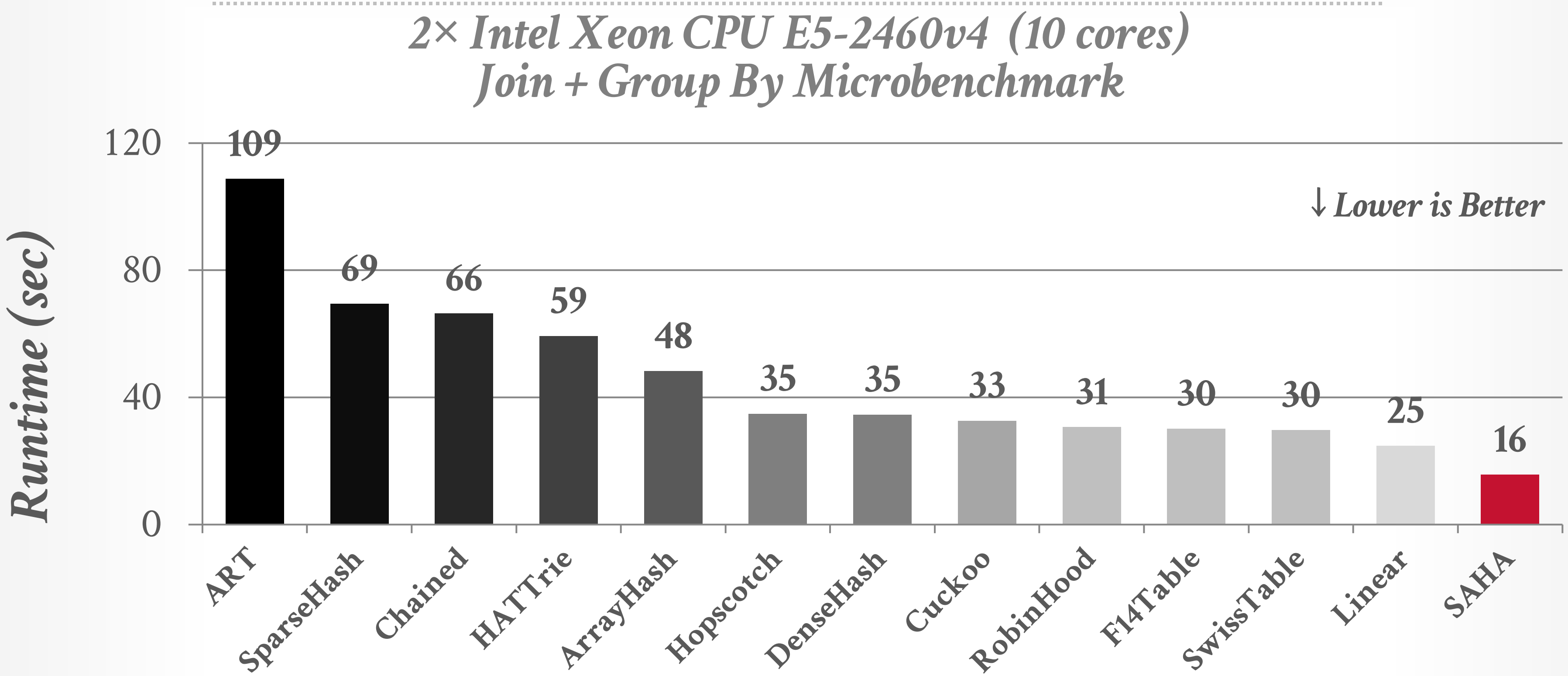
Compiled Expression Evaluator (LLVM)

Sort-Merge + Hash Joins

Heuristic Optimizer + Rule-Based Rewriting

CLICKHOUSE - LIGHTNING FAST ANALYTICS FOR EVERYONE
VLDB 2024

ClickHouse: String Hash Tables



SAHA: A STRING ADAPTIVE HASH TABLE FOR ANALYTICAL DATABASES
APPL. SCI. 2020

Concluding Remarks

Databases are awesome.

→ They cover all facets of computer science.

→ We have barely scratched the surface...

Going forth, you should now have a good understanding how these systems work.

This will allow you to make informed decisions throughout your entire career.

→ Avoid premature optimizations.