Carnegie Mellon University

Database Systems

Final Review & Systems Potpourri

15-445/645 SPRING 2025 » PROF. JIGNESH PATEL

ADMINISTRIVIA

Final Exam is on Monday, April 28, 2025, from 5:30pm- 8:30pm.

→ Early exam will not be offered. Do not make travel plans.
→ Material: Lecture 12 – Lecture 24.
→ You can use the full 3 hours, though the exam is meant to be done in ~2 hours.

Last day to submit P4 (with late days and penalty) is April 30 @ 11:59 pm

Course Evals: Would like your feedback.

→ https://cmu.smartevals.com
→ https://www.ugrad.cs.cmu.edu/ta/S25/feedback/

OFFICE HOURS

Jignesh:

→ Thursday April 24th @ noon-2:00pm (GHC 9103)

All other TAs will have their office hours up to and including Saturday April 26th

FINAL EXAM

Where: Scaife Hall 105 and Scaife Hall 234.

When: Monday, April 28, 2025, 5:30pm- 8:30pm.

Come to Scaife Hall 105 first.
Then, look at your seating assignment, which may assign you to Scaife Hall 234.

https://15445.courses.cs.cmu.edu/spring2025/final-guide.html

FINAL EXAM

What to bring:

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

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

→ Strong Strict 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

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
Cloud Architectures

TOPICS NOT ON EXAM!

Flash Talks

Seminar Talks

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

GOOGLE SPANNER

Google's geo-replicated DBMS (>2011)

Schematized, semi-relational data model.

Decentralized shared-disk architecture.

Log-structured on-disk storage.

Concurrency Control:

→ Strict 2PL + MVCC + Multi-Paxos + 2PC
→ **Externally consistent** global write-transactions with synchronous replication.
→ Lock-free read-only transactions.

SPANNER: CONCURRENCY CONTROL

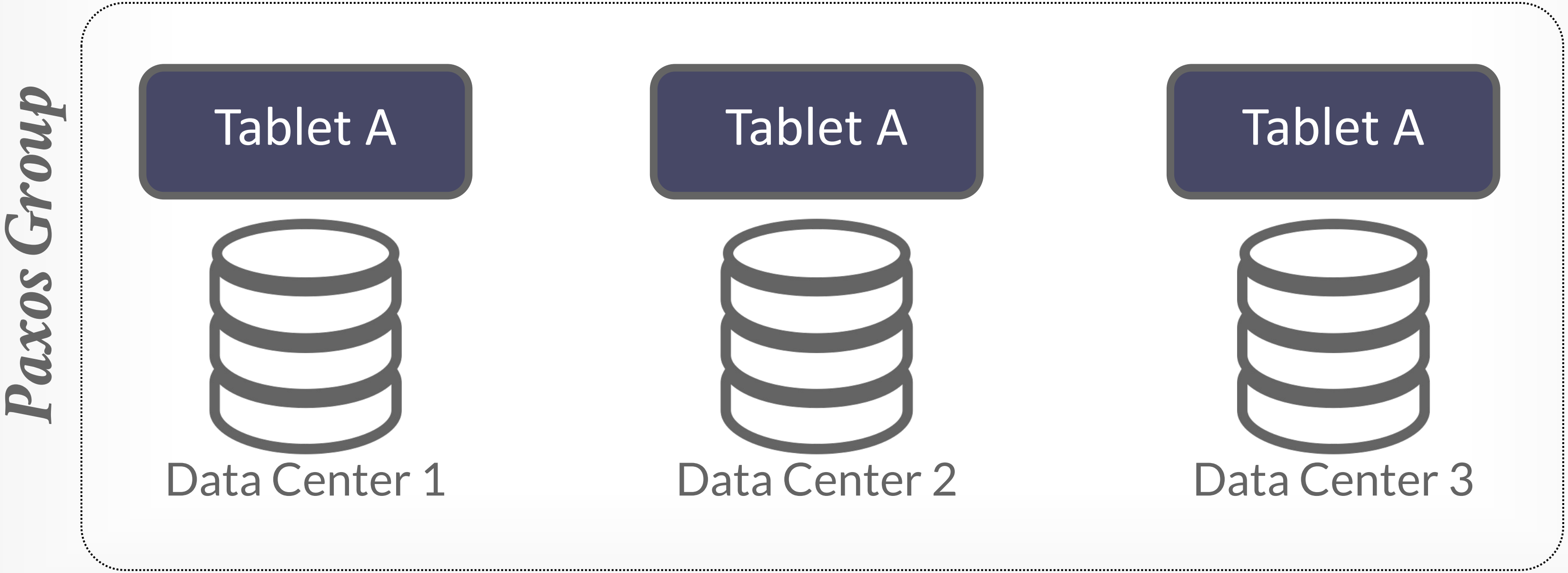
MVCC + Strict 2PL with Wound-Wait Deadlock Prevention

DBMS ensures ordering through globally unique timestamps generated from atomic clocks and GPS devices.

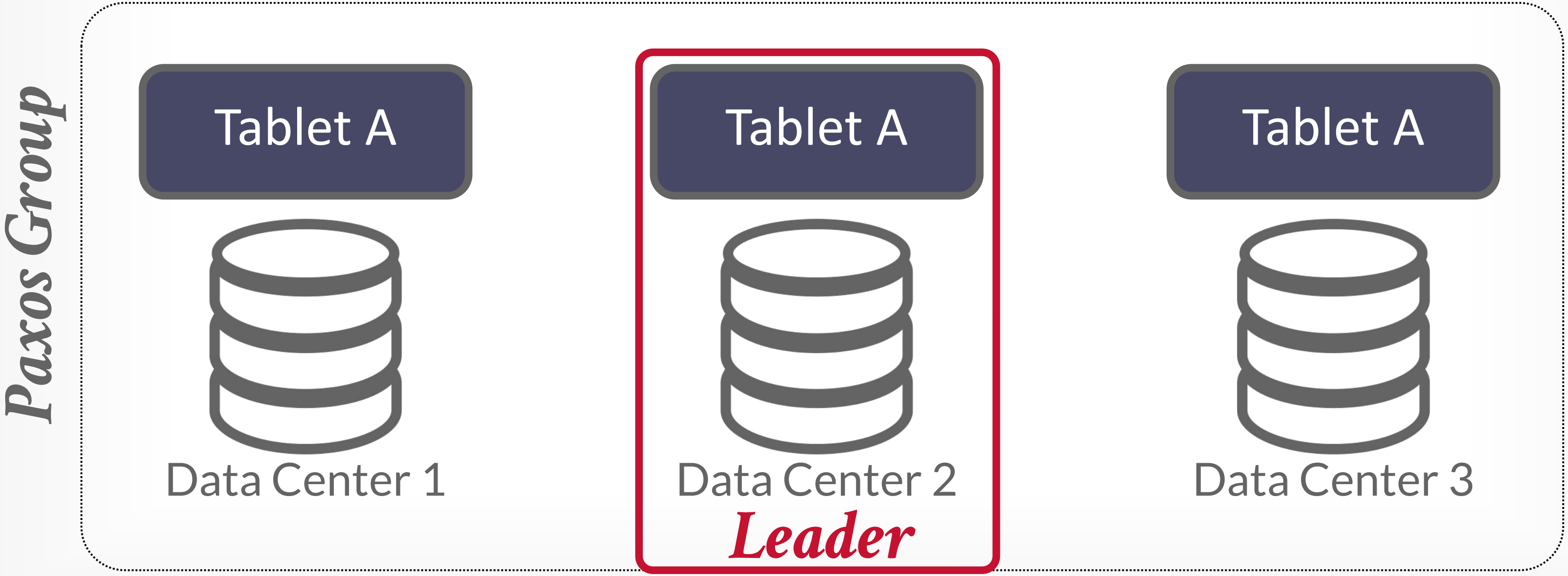
Buffer writes in the client, and these are sent to the server at commit time.

Database is broken up into tablets (partitions):
→ Use Paxos to elect leader in tablet group.
→ Use 2PC for txns that span tablets.

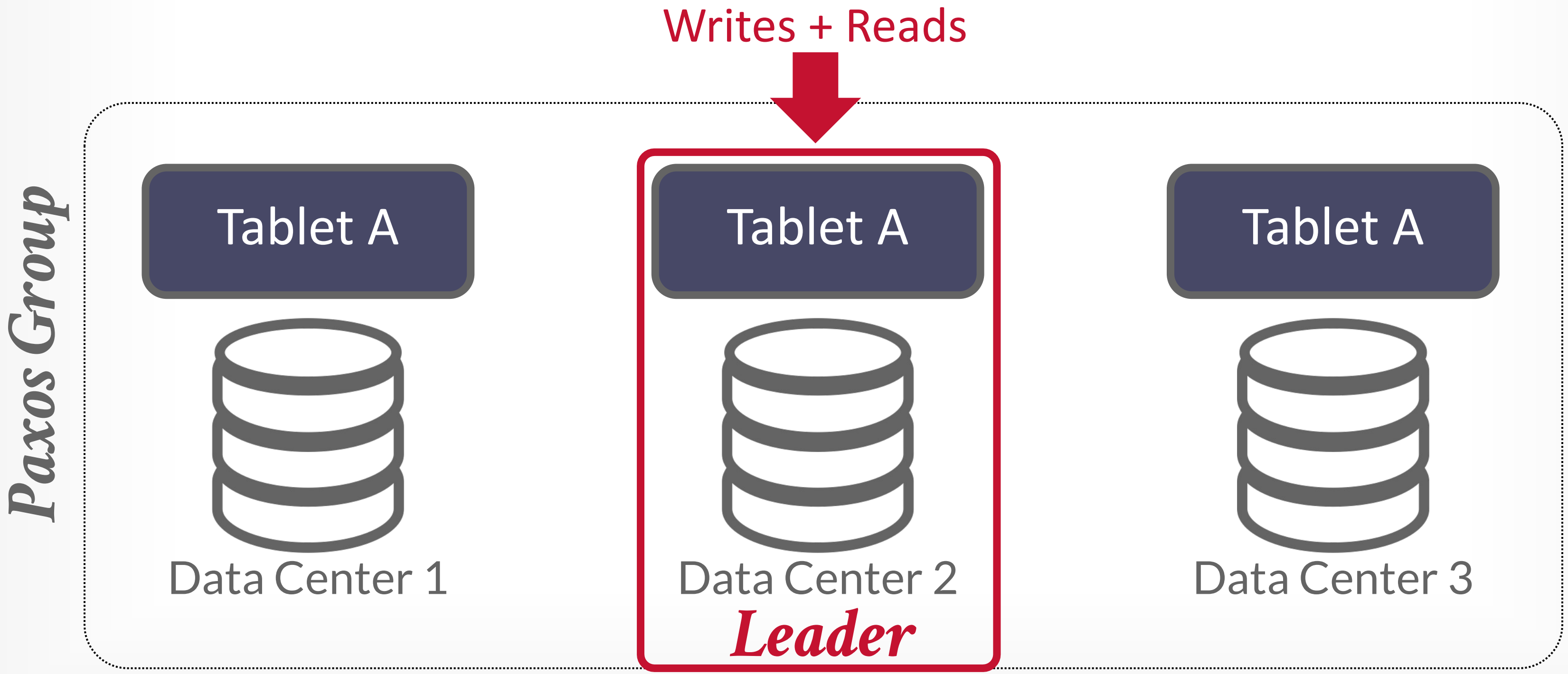
SPANNER TABLETS



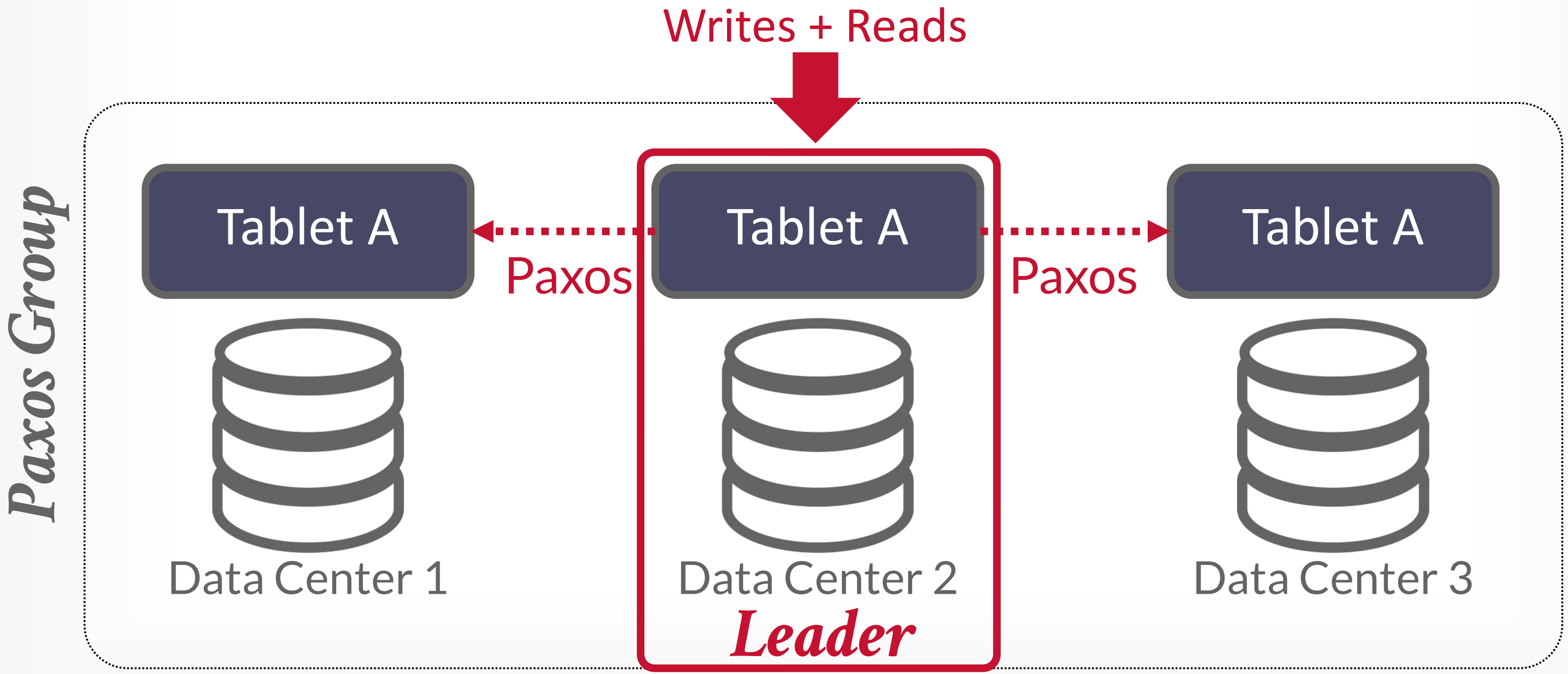
SPANNER TABLETS



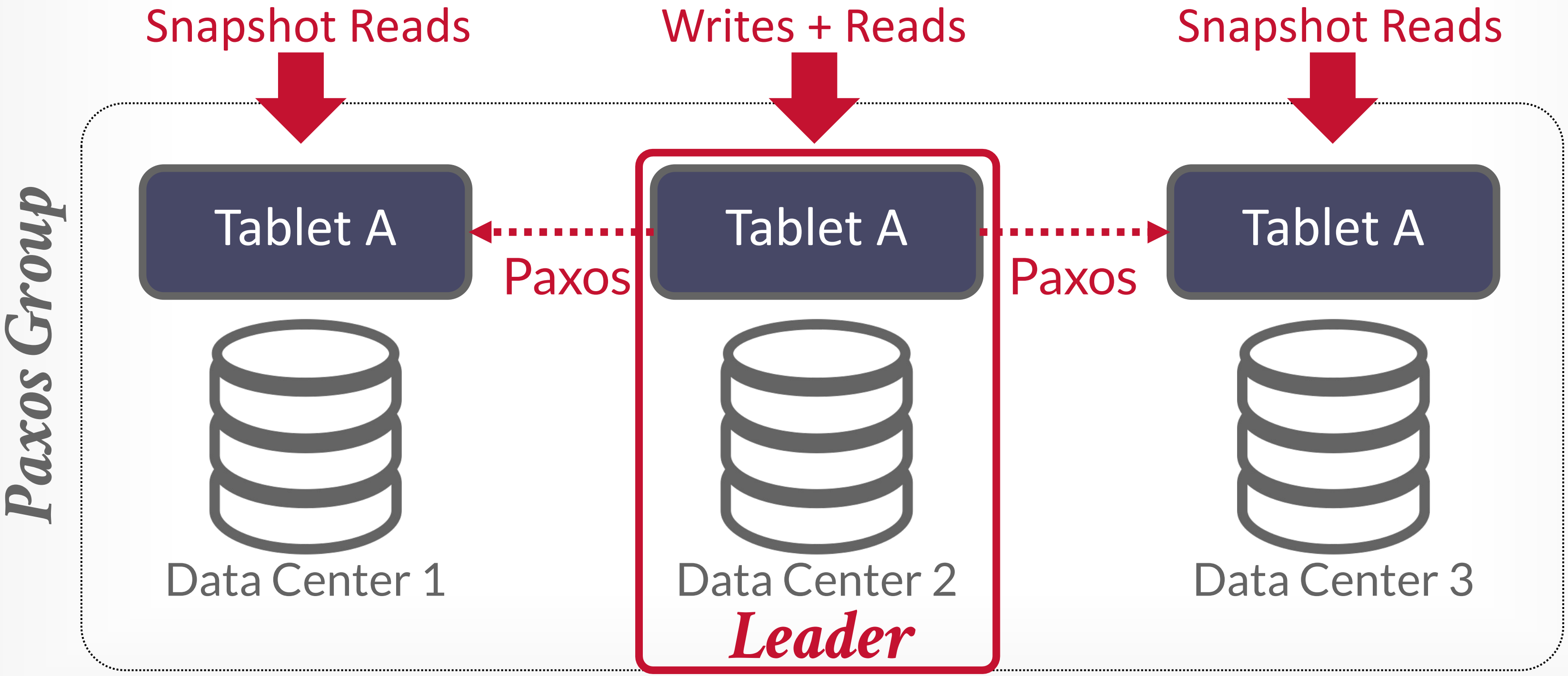
SPANNER TABLETS



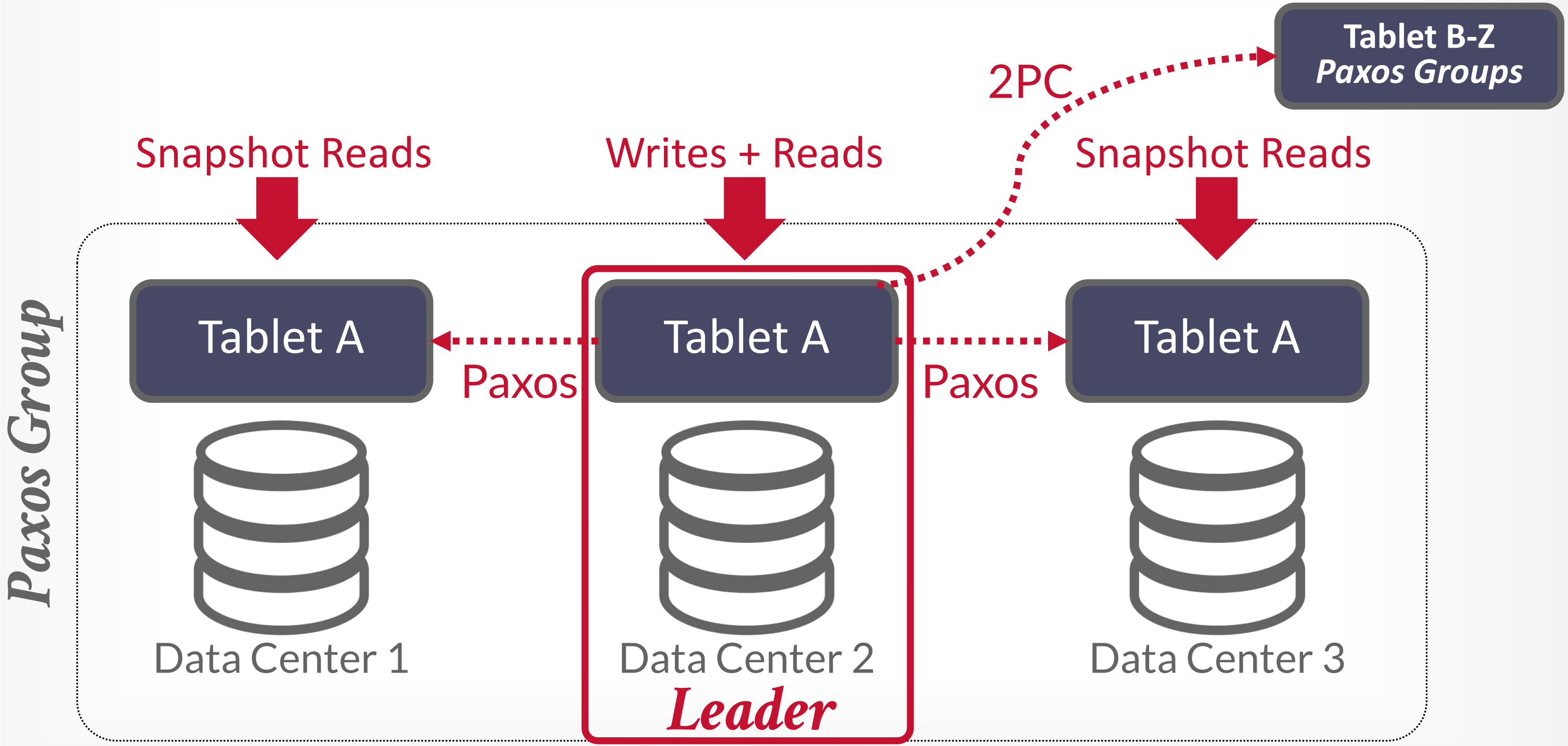
SPANNER TABLETS



SPANNER TABLETS



SPANNER TABLETS



SPANNER: TRANSACTION ORDERING

DBMS orders transactions based on physical "wall-clock" time.

→ This is necessary to guarantee strict serializability.

→ If T_1 finishes before T_2, then T_2 should see the result of T_1.

Each Paxos group decides in what order transactions should be committed according to the timestamps.

→ If T_1 commits at $\texttt{time}_1$ and T_2 starts at $\texttt{time}_2$ > $\texttt{time}_1$, then T_1's timestamp should be less than T_2's.

SPANNER TRUETIME

The DBMS maintains a global wall-clock time across all data centers with bounded uncertainty.

Timestamps are intervals, not single values

SPANNER TRUETIME

The DBMS maintains a global wall-clock time across all data centers with bounded uncertainty.

Timestamps are intervals, not single values

SPANNER: TRUETIME

Each data center has GPS and atomic clocks

→ These two provide fine-grained clock synchronization down to a few milliseconds.

→ Every 30 seconds, there is a maximum 7 ms difference.

Multiple sync daemons per data center

→ GPS and atomic clocks can fail in various conditions.

→ Sync daemons talk to each other within a data center as well as across data centers.

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

BIGQUERY: OVERVIEW

Shared-Disk / Disaggregated Storage

Vectorized Query Processing

$\boxed{\text{Shuffle-based Distributed Query Execution}}$

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

Hash Joins Only

Heuristic Optimizer + Adaptive Optimizations

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:

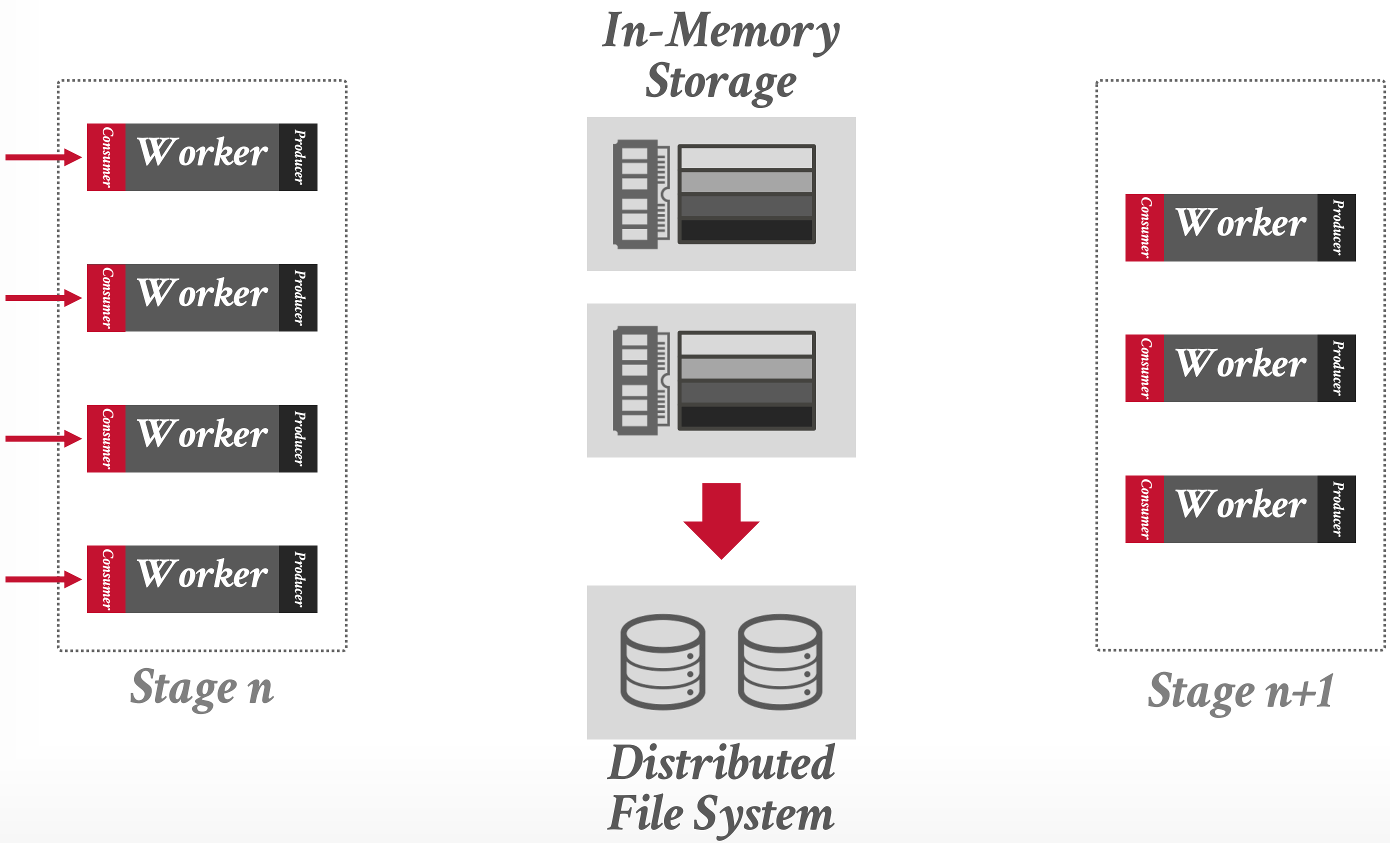
→ If a worker does not produce a task's results within a deadline, the coordinator speculatively executes a redundant task.

Dynamic Resource Allocation:

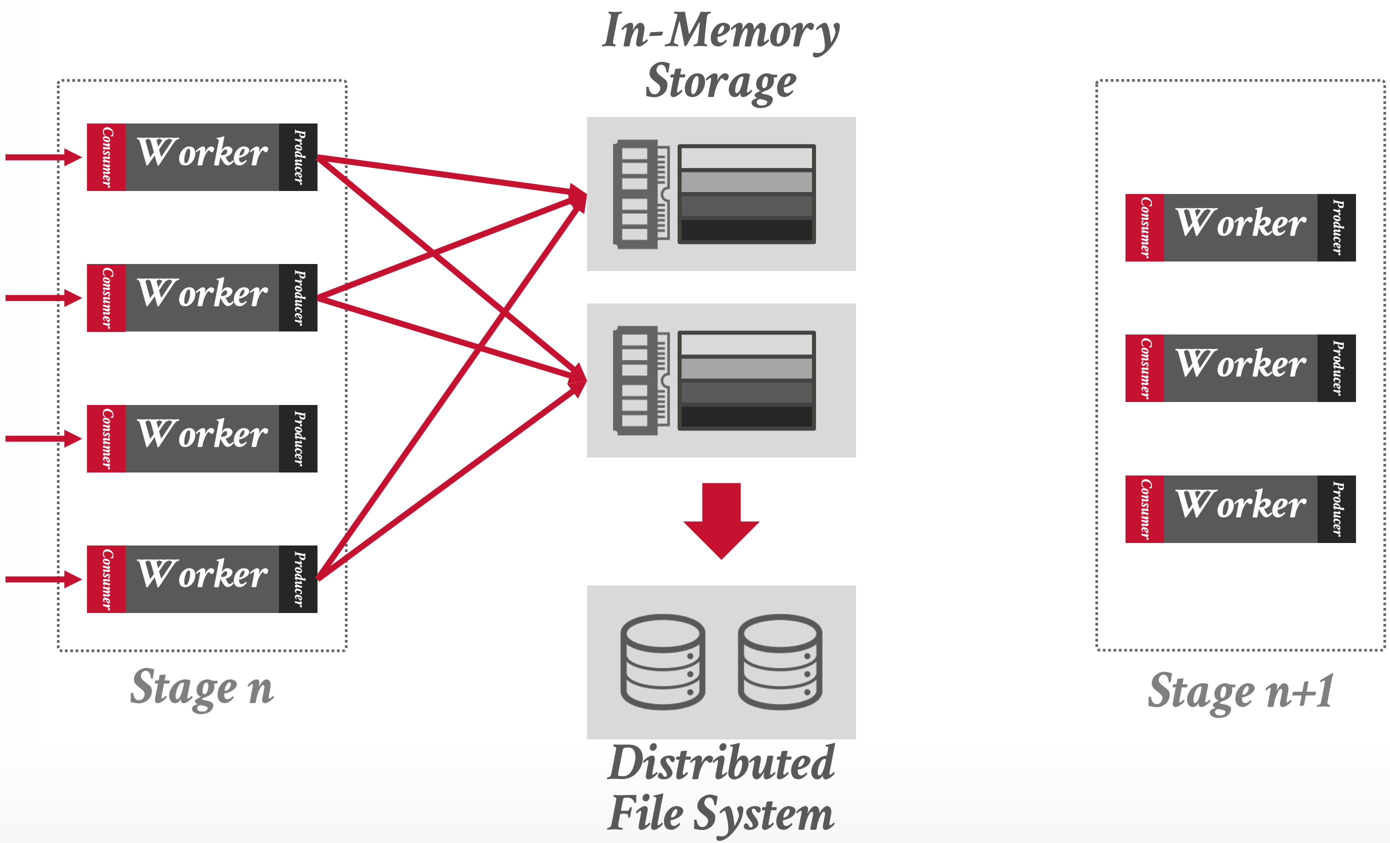
→ Scale up / down the number of workers for the next stage depending size of a stage's output.

Google Big Query

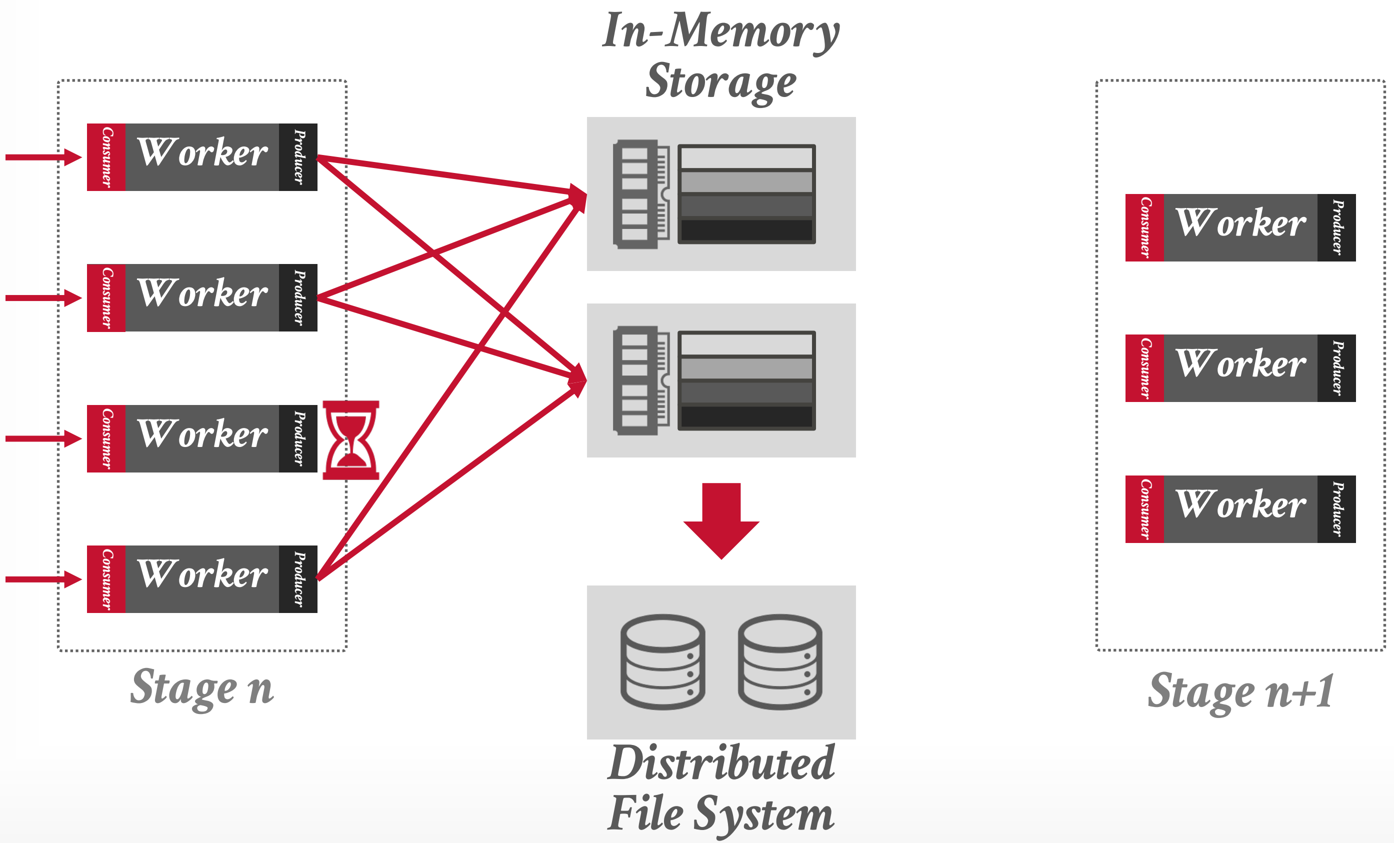
BIGQUERY: IN-MEMORY SHUFFLE



BIGQUERY: IN-MEMORY SHUFFLE

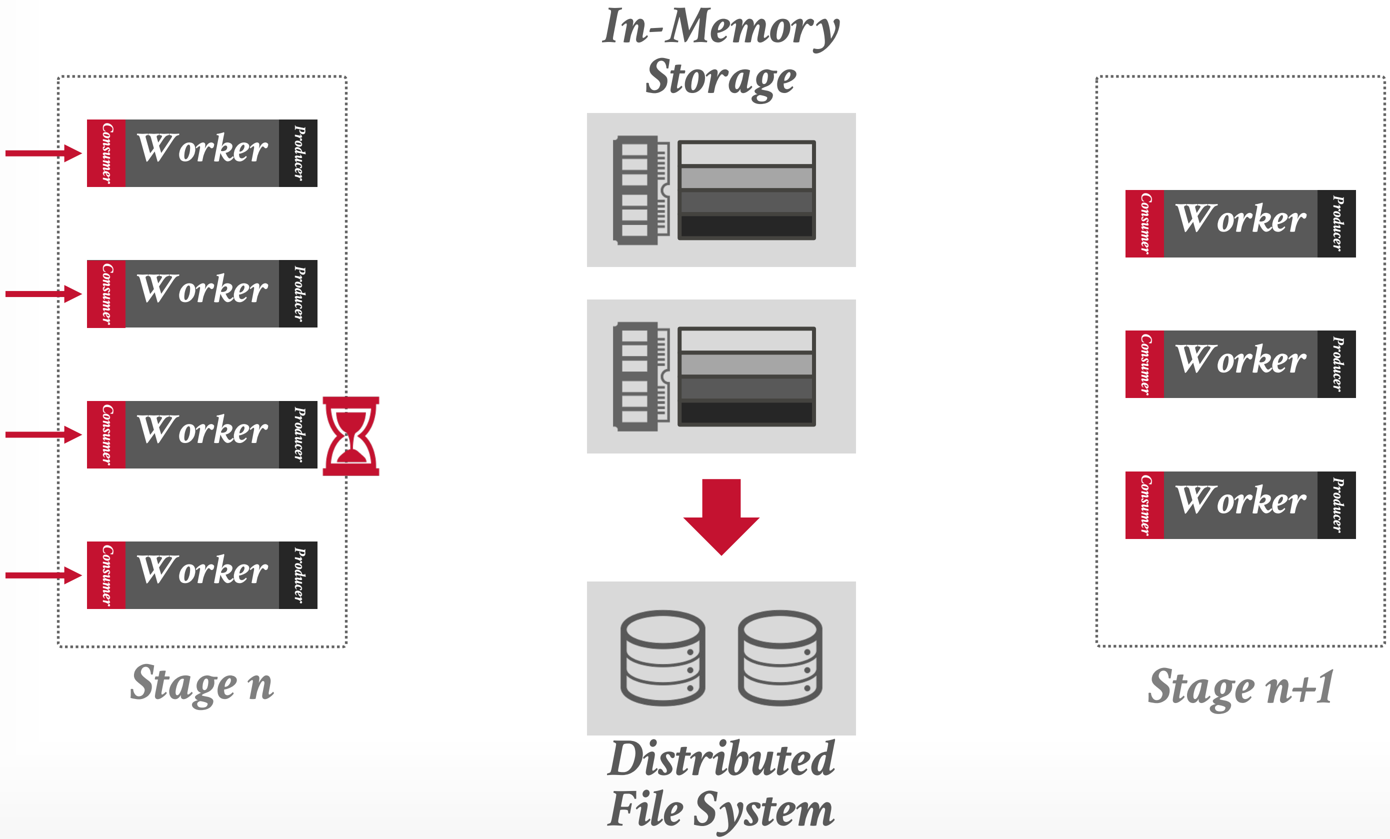


BIGQUERY: IN-MEMORY SHUFFLE



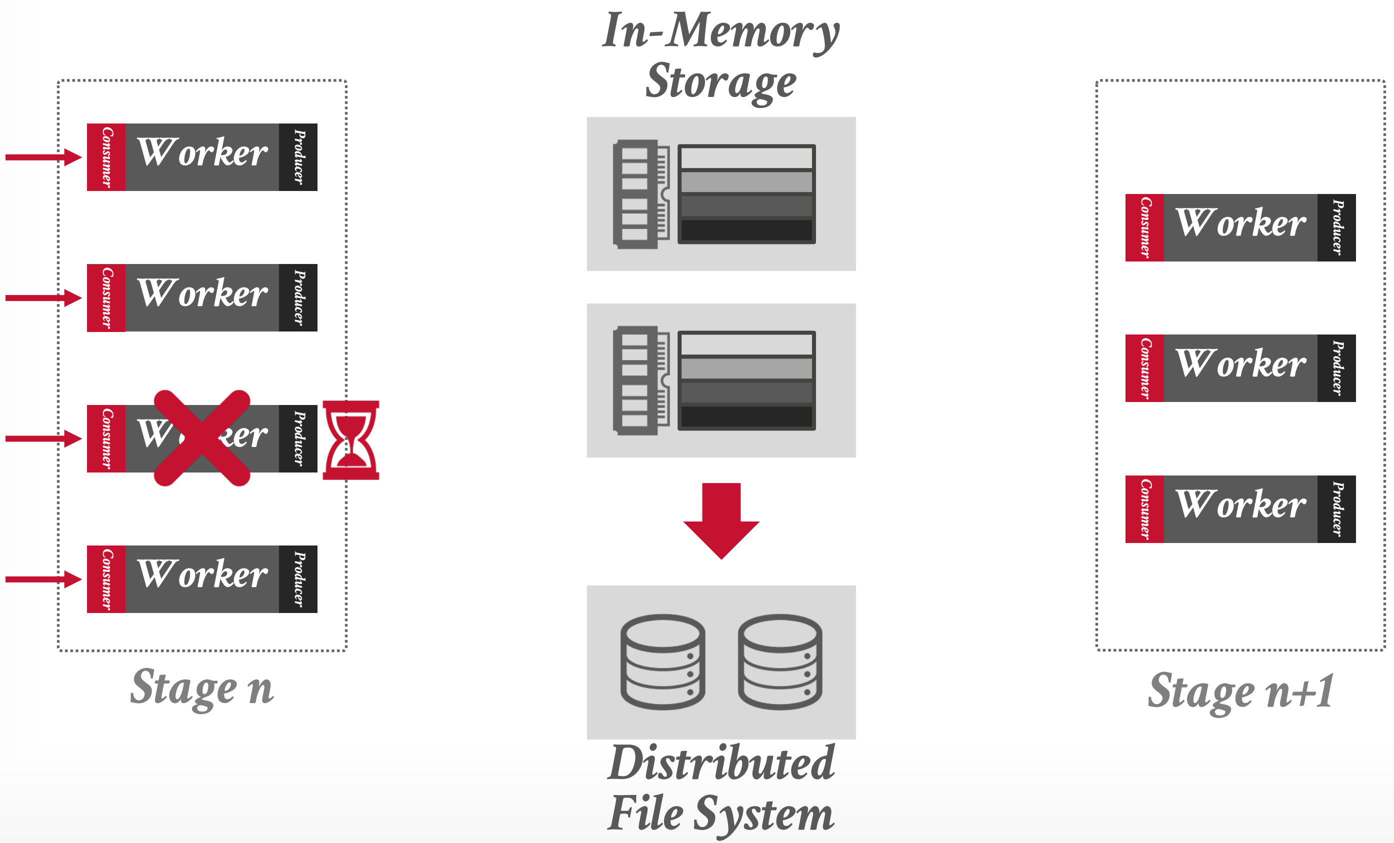
Google Big Query

BIGQUERY: IN-MEMORY SHUFFLE

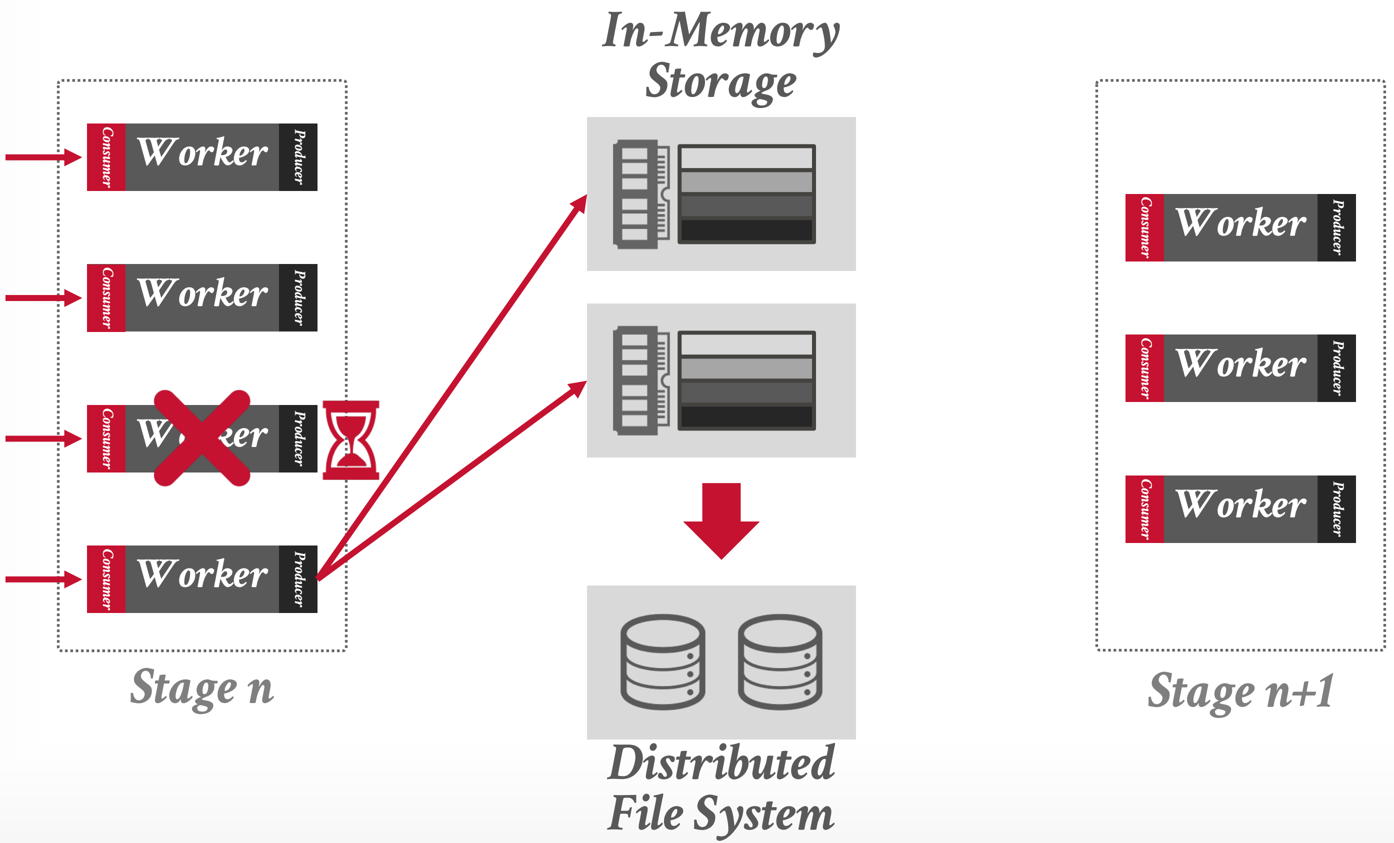


Google
Big Query

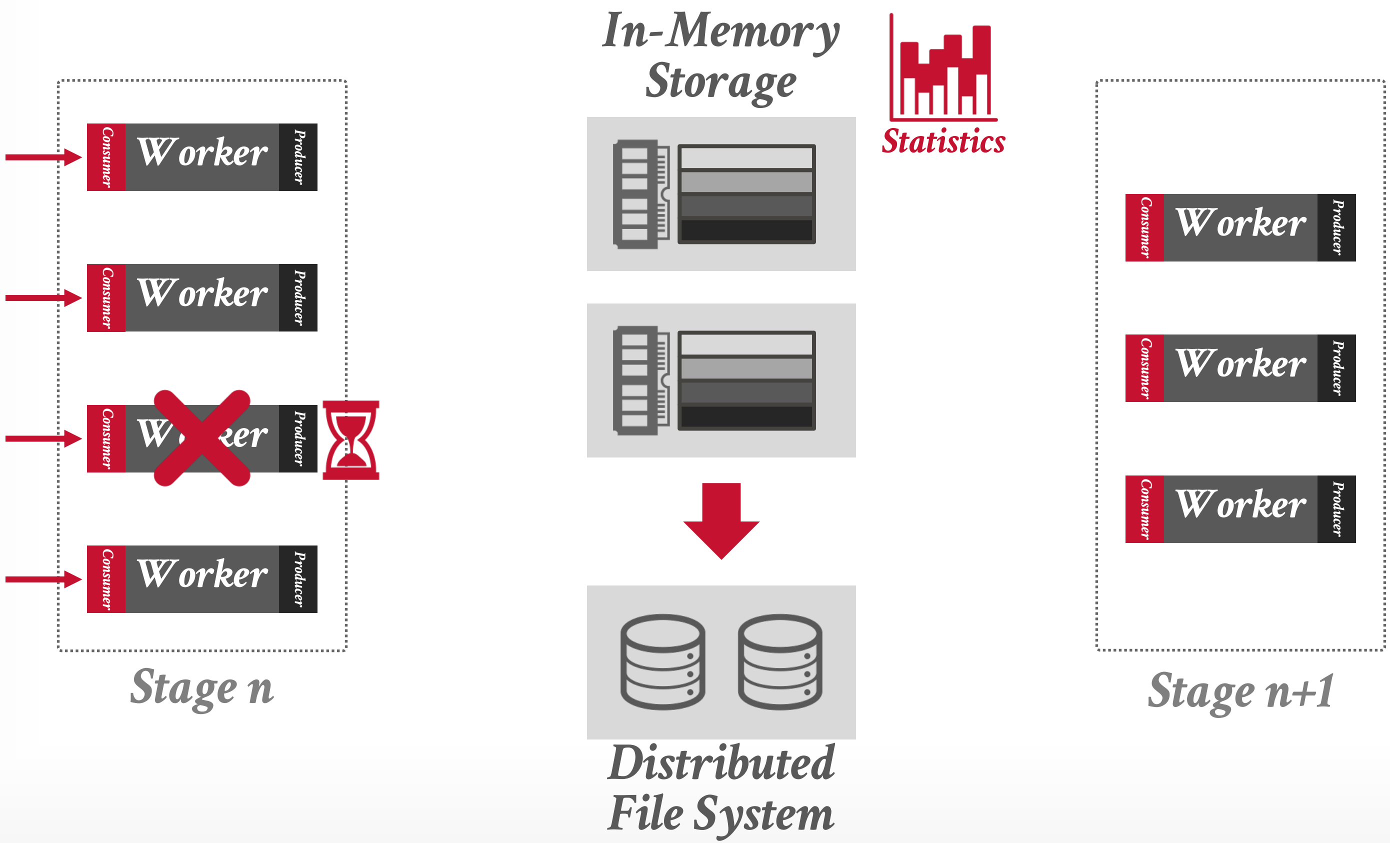
BIGQUERY: IN-MEMORY SHUFFLE



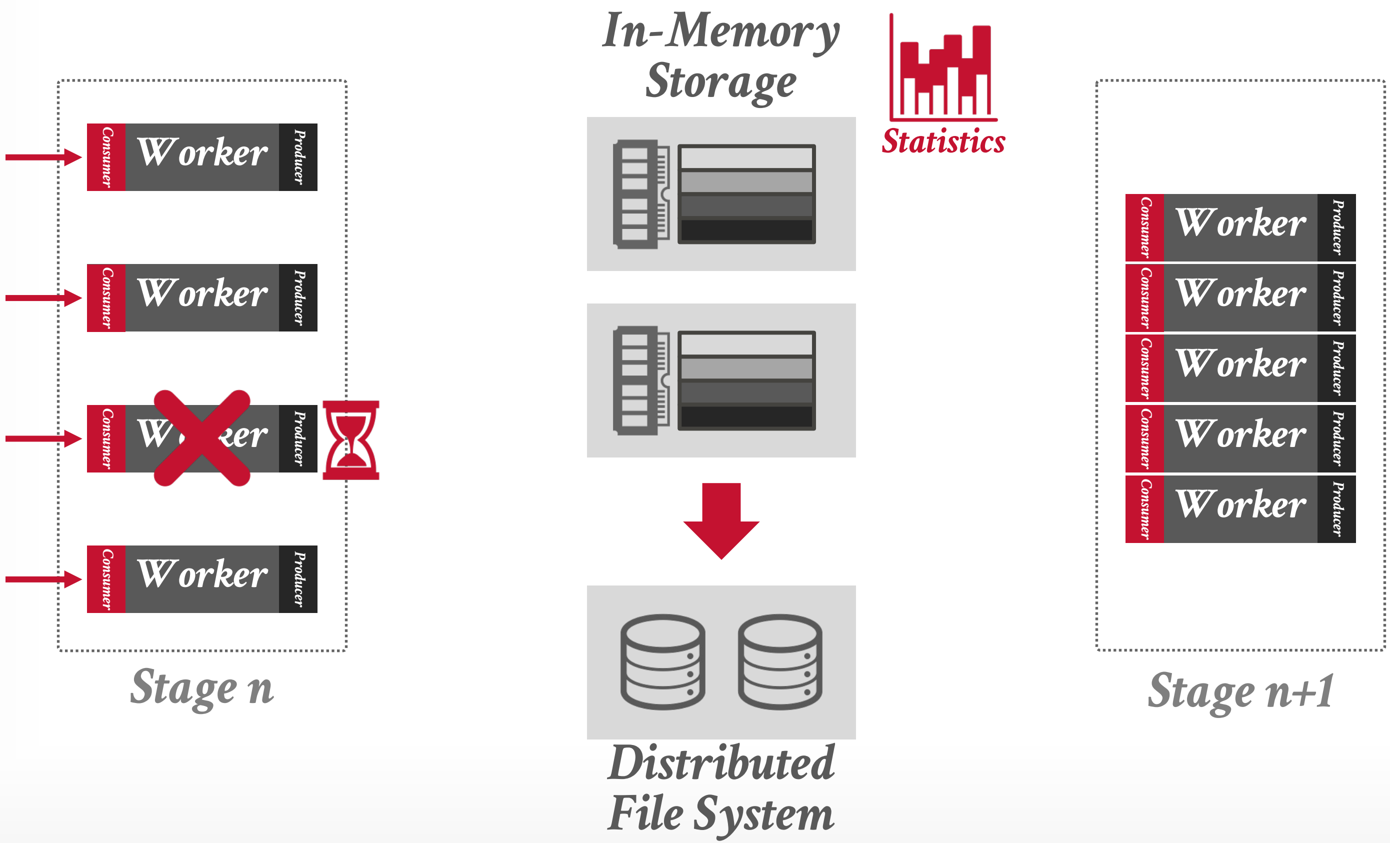
BIGQUERY: IN-MEMORY SHUFFLE



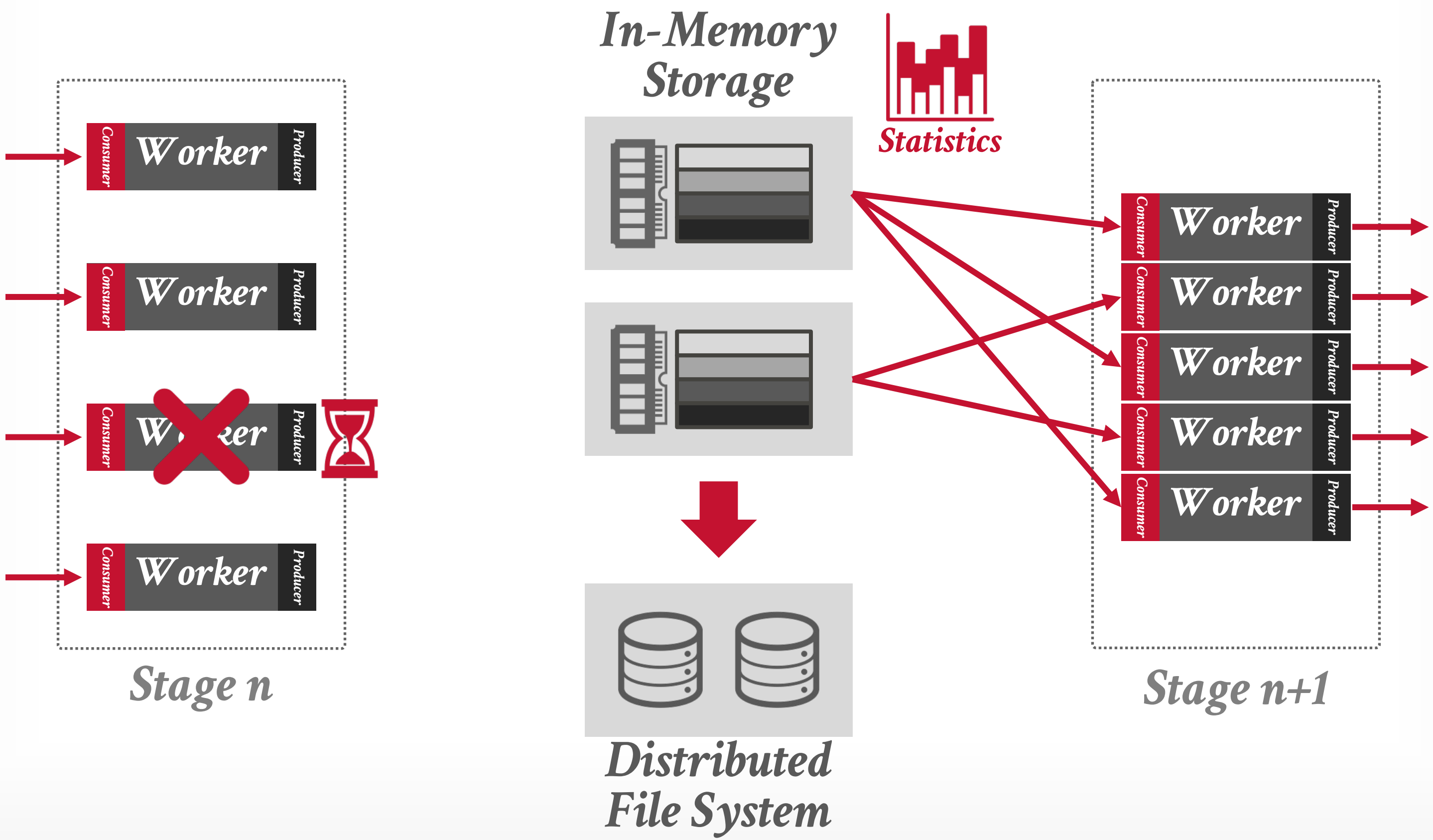
BIGQUERY: IN-MEMORY SHUFFLE



BIGQUERY: IN-MEMORY SHUFFLE



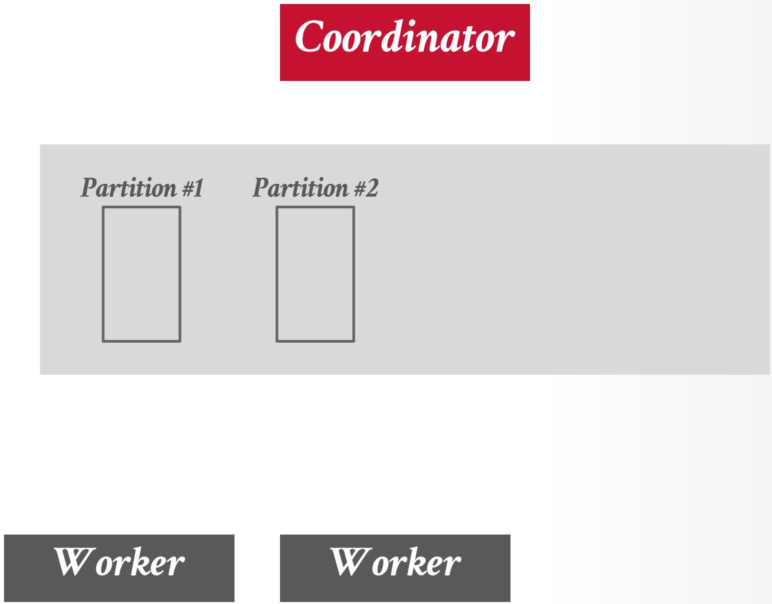
BIGQUERY: IN-MEMORY SHUFFLE



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.

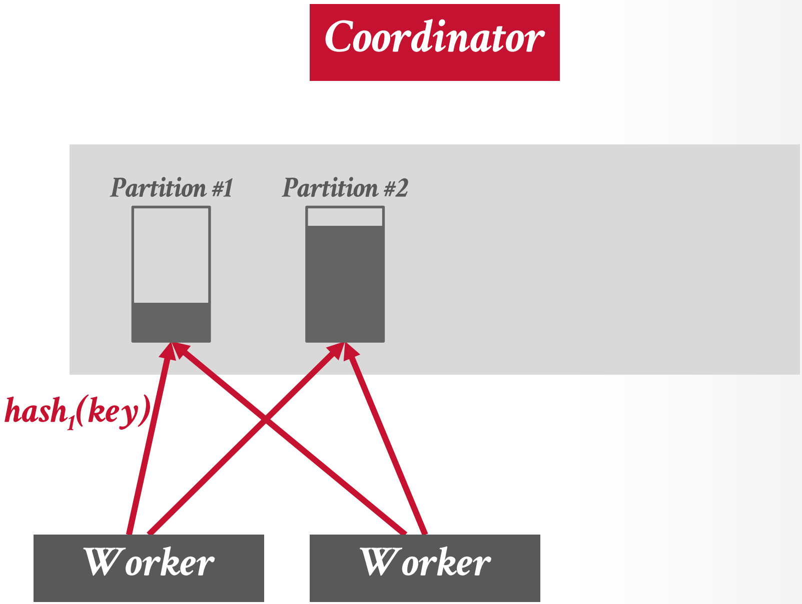


Source: H.Ahmadi + A.Surna

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.



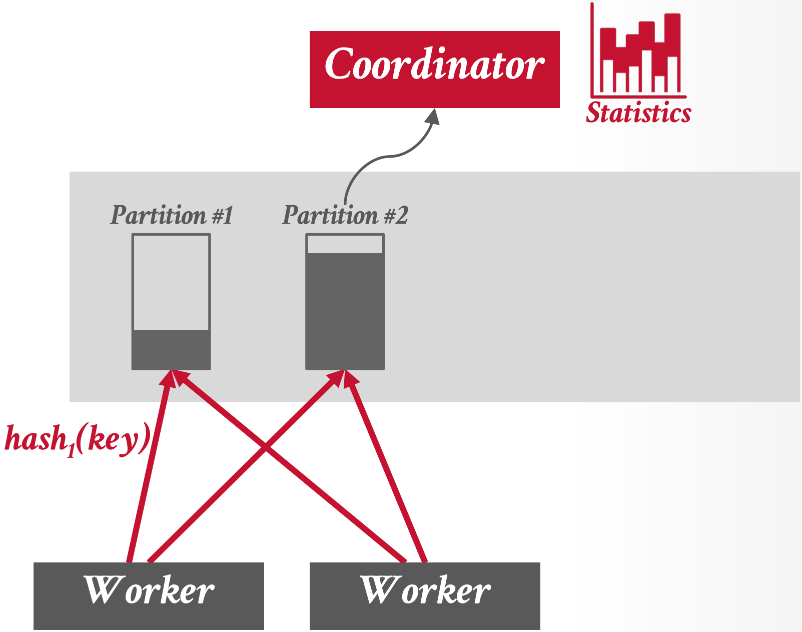
Source: H.Ahmadi + A.Surna

Google Big Query

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.



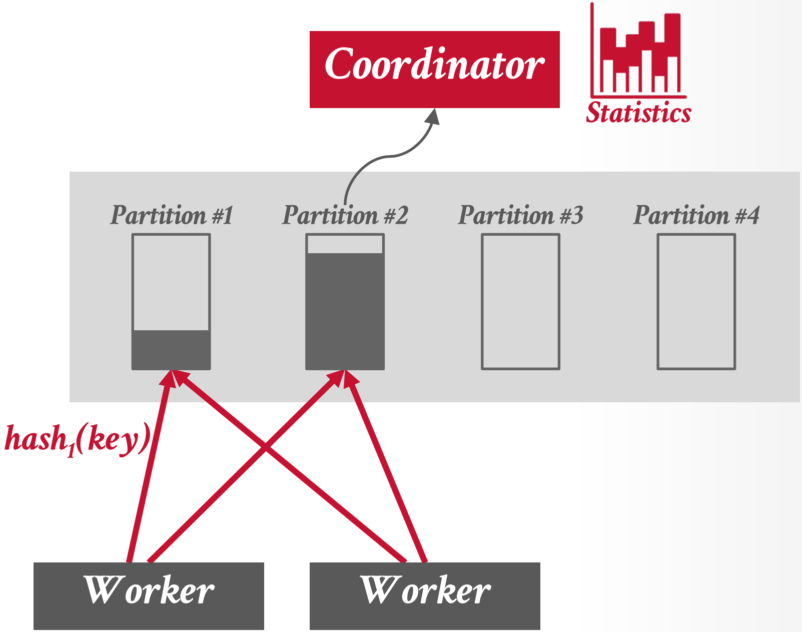
Source: H.Ahmadi + A.Surna

Google Big Query

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.



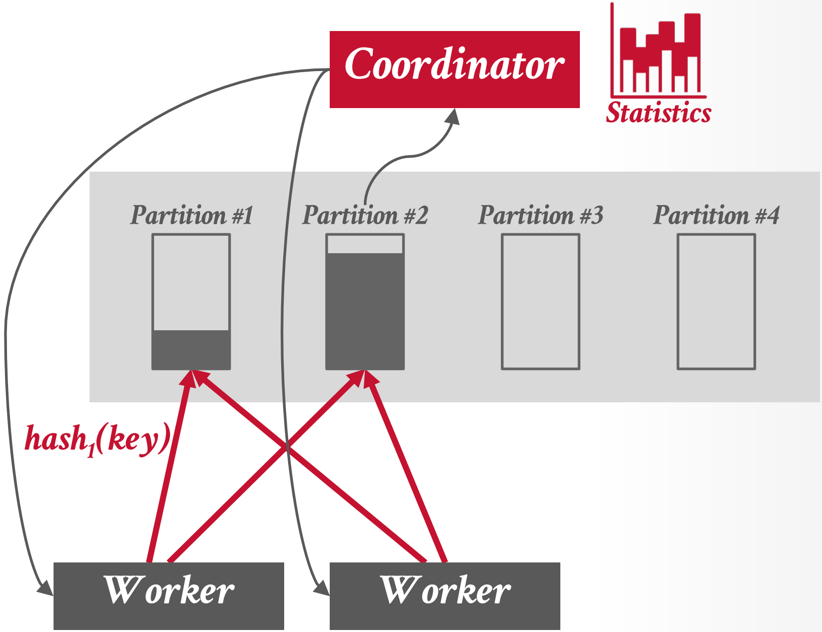
Source: H.Ahmadi + A.Surna

Google Big Query

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.

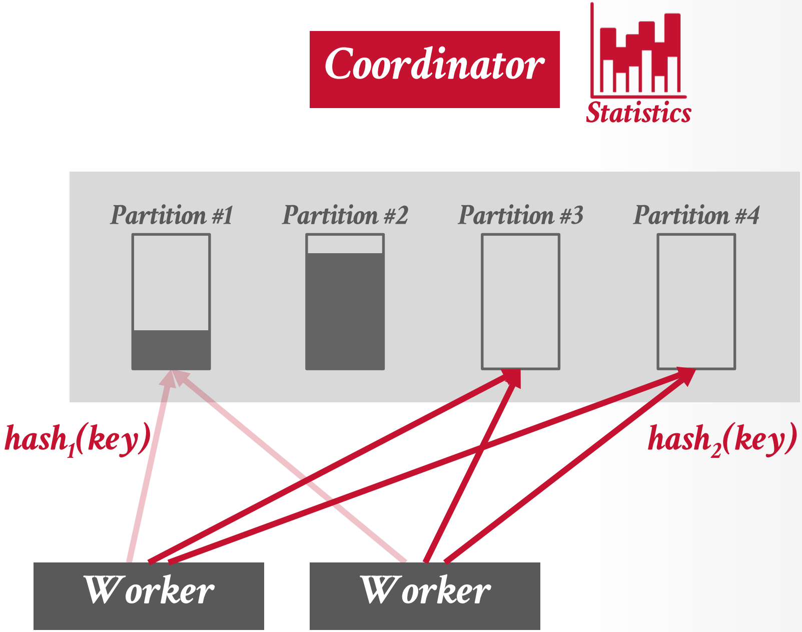


Source: H.Ahmadi + A.Surna

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.



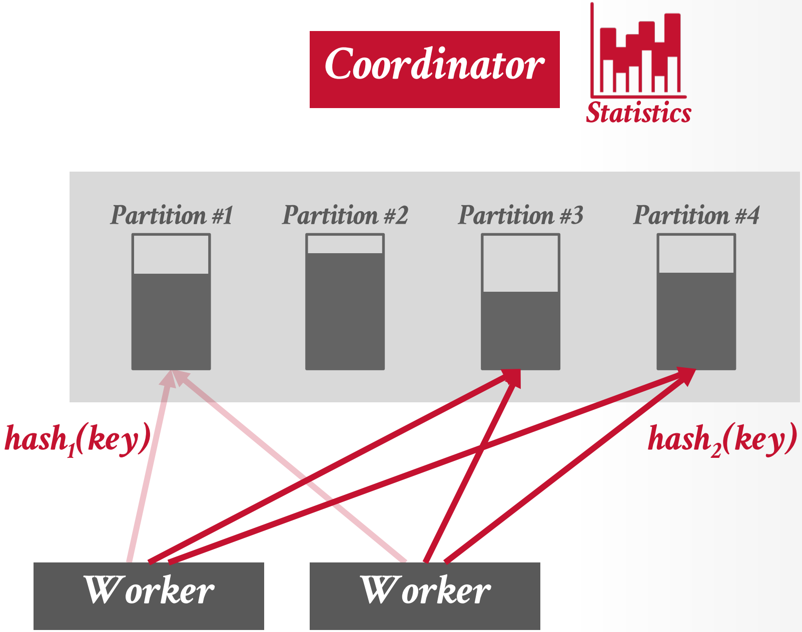
Source: H.Ahmadi + A.Surna

Google Big Query

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.



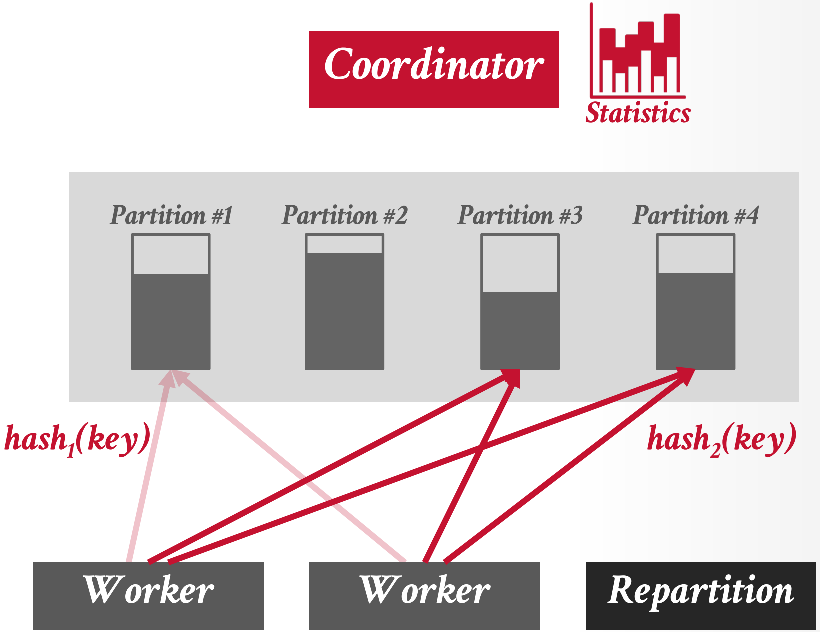
Source: H.Ahmadi + A.Surna

Google Big Query

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.

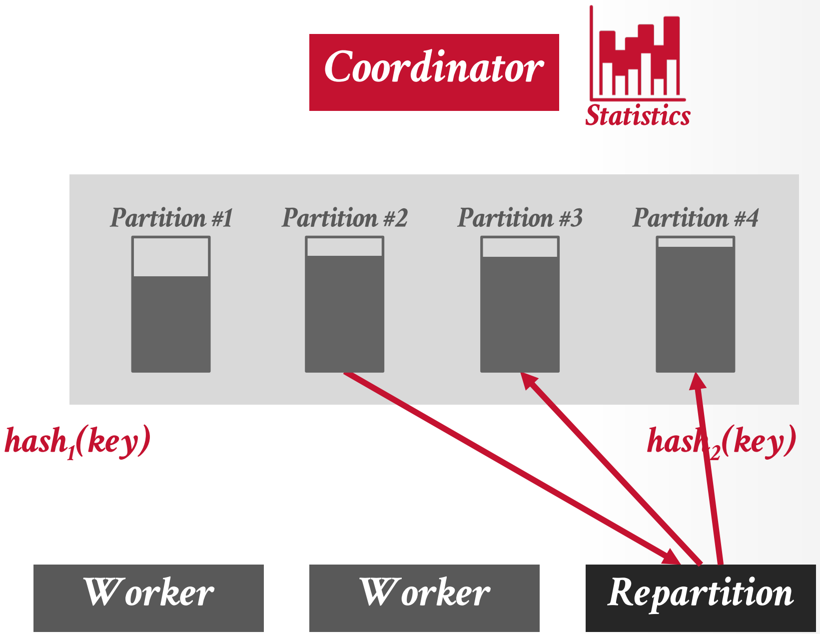


Source: H.Ahmadi + A.Surna

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.

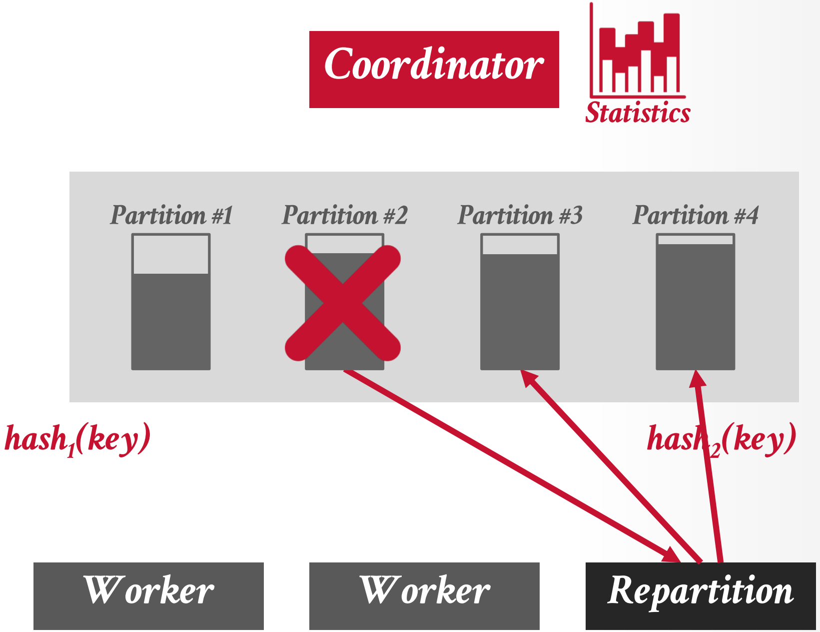


Source: H.Ahmadi + A.Surna

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.

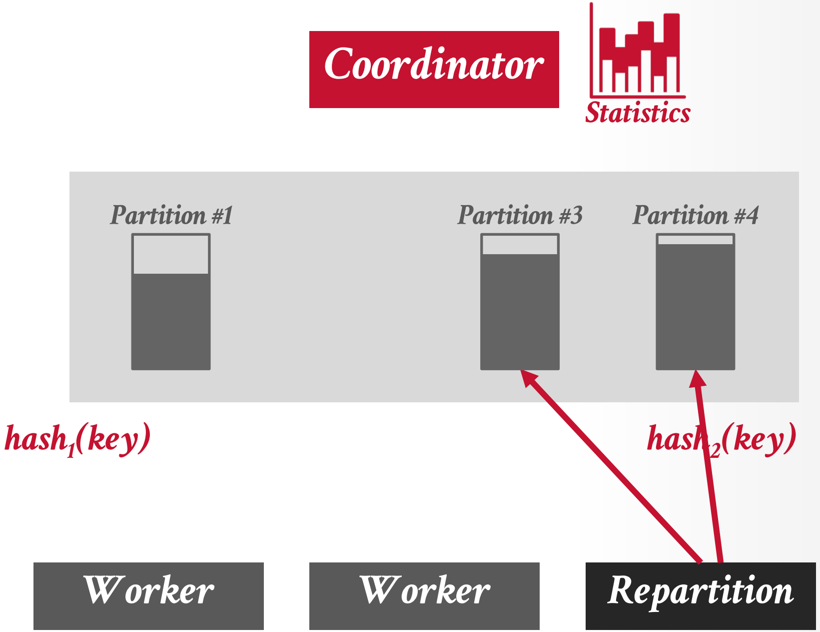


Source: H.Ahmadi + A.Surna

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 (2

Managed OLAP DBMS written i

→ Shared-disk architecture with aggres local caching.

→ Written from scratch. Did not borr existing systems.

→ Custom SQL dialect and client-serv

The OG cloud-native data warel

THE SNOWFLAKE ELASTIC DATA WAREHOUSE
SIGMOD 2016

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.

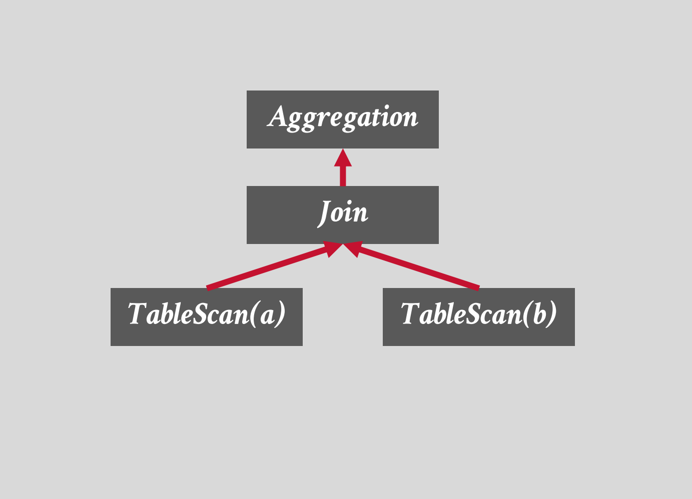
Does not support partial query retries

→ 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.

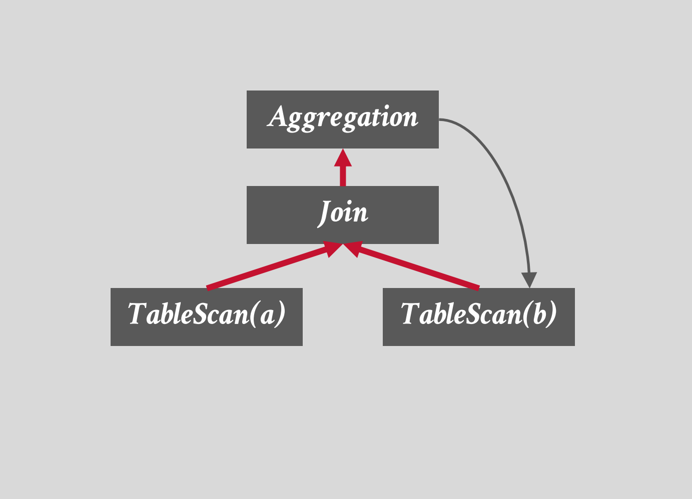


Source: Bowei Chen

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.

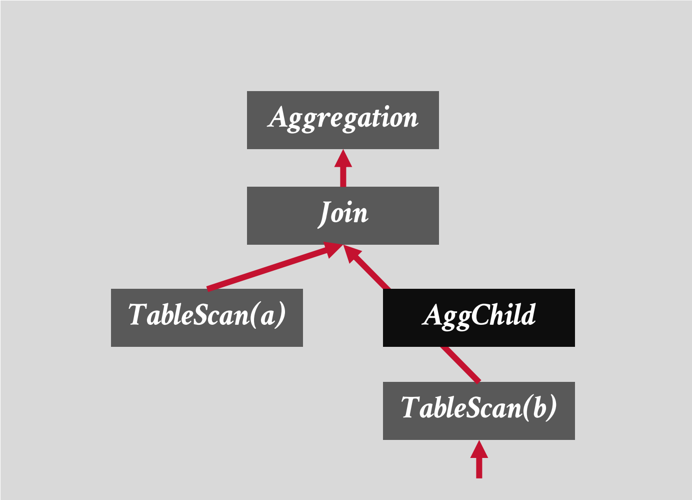


Source: Bowei Chen

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.

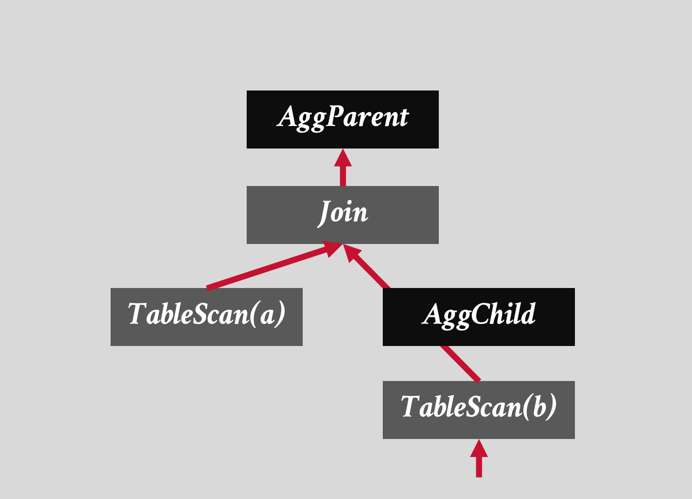


Source: Bowei Chen

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: AD

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

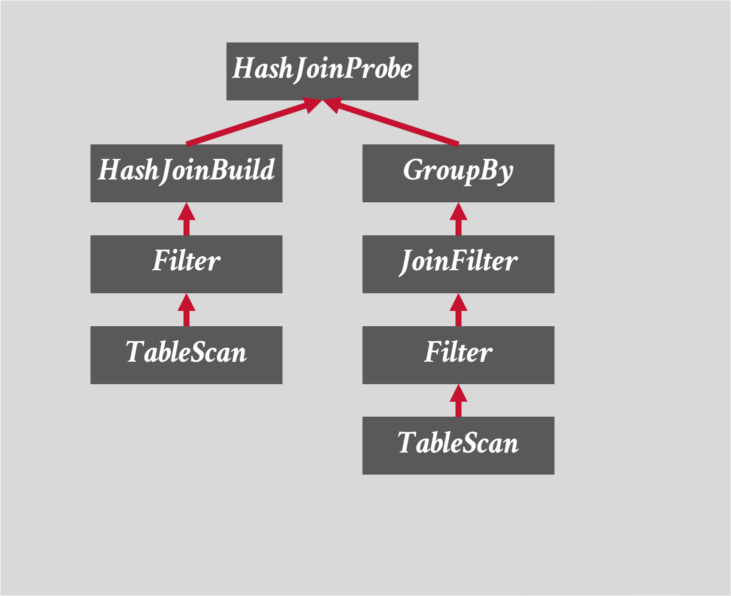
Snowflake's Data Cloud is backed by a data platform designed from the ground up to leverage cloud computing technology. The platform is delivered as a fully managed service, providing a user-friendly experience to run complex analytical workloads easily and efficiently without the burden of managing 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.

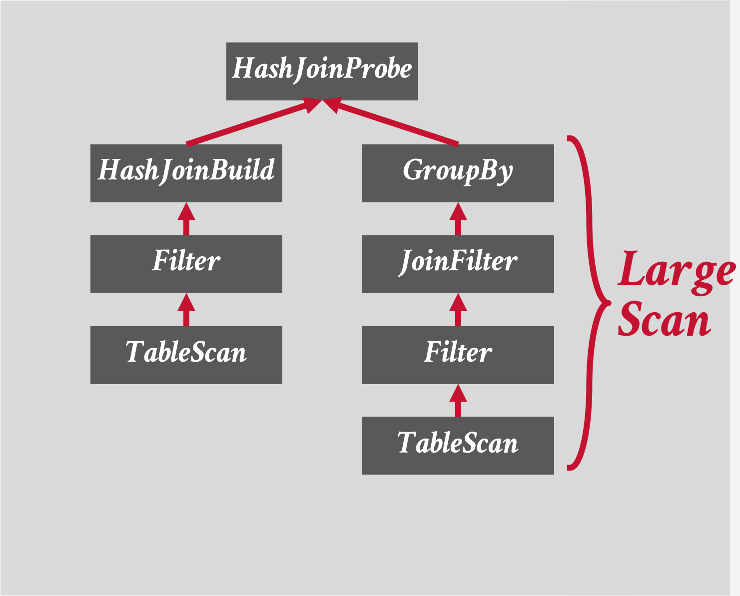


Source: Libo Wang

SNOWFLAKE: 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.

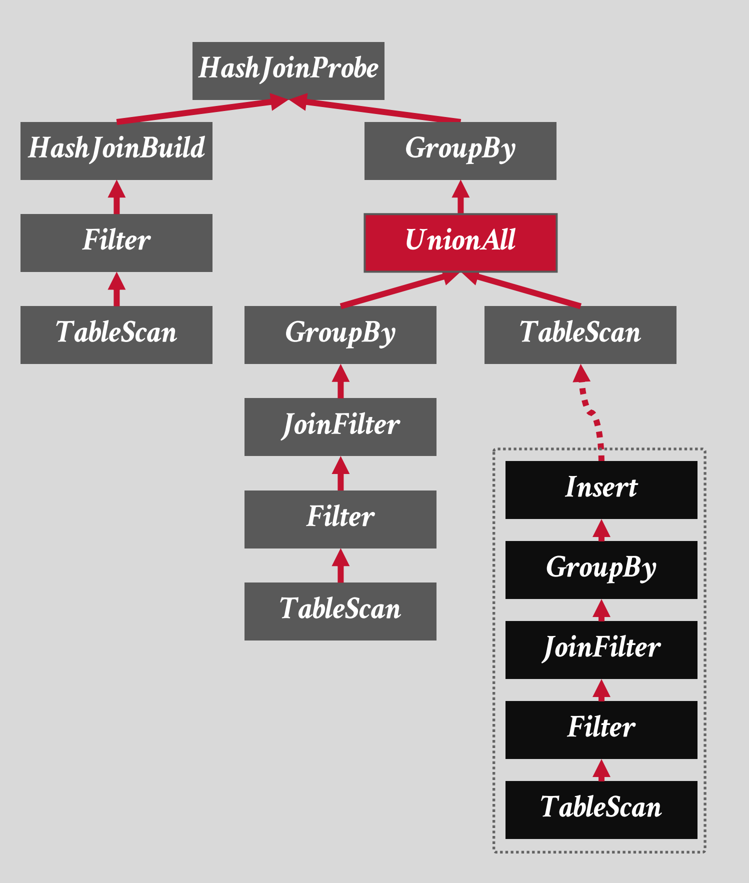


Source: Libo Wang

SNOWFLAKE: 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.

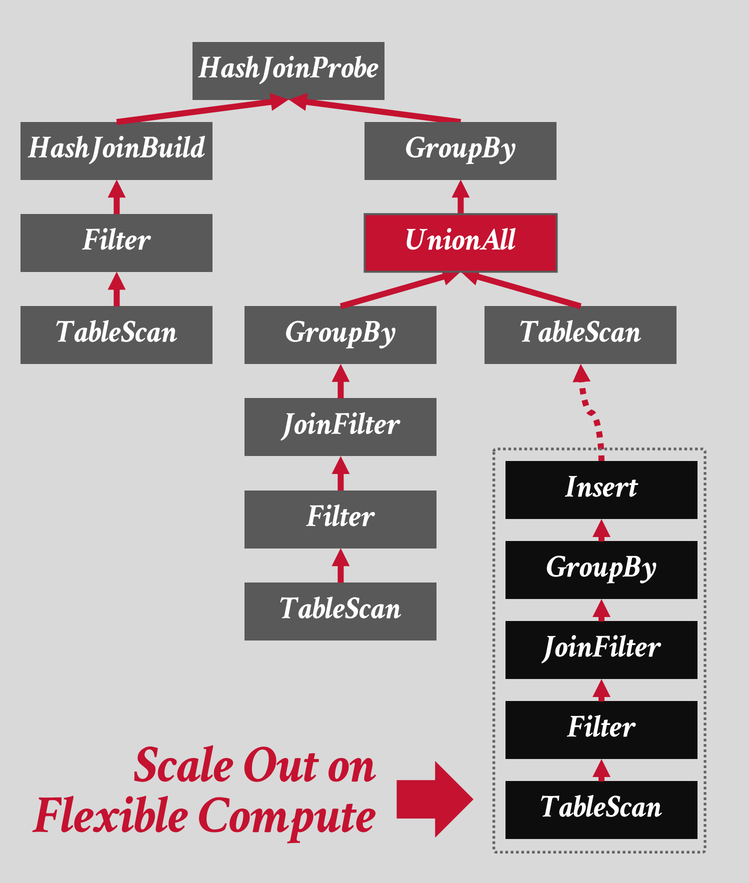


Source: Libo Wang

SNOWFLAKE: 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.

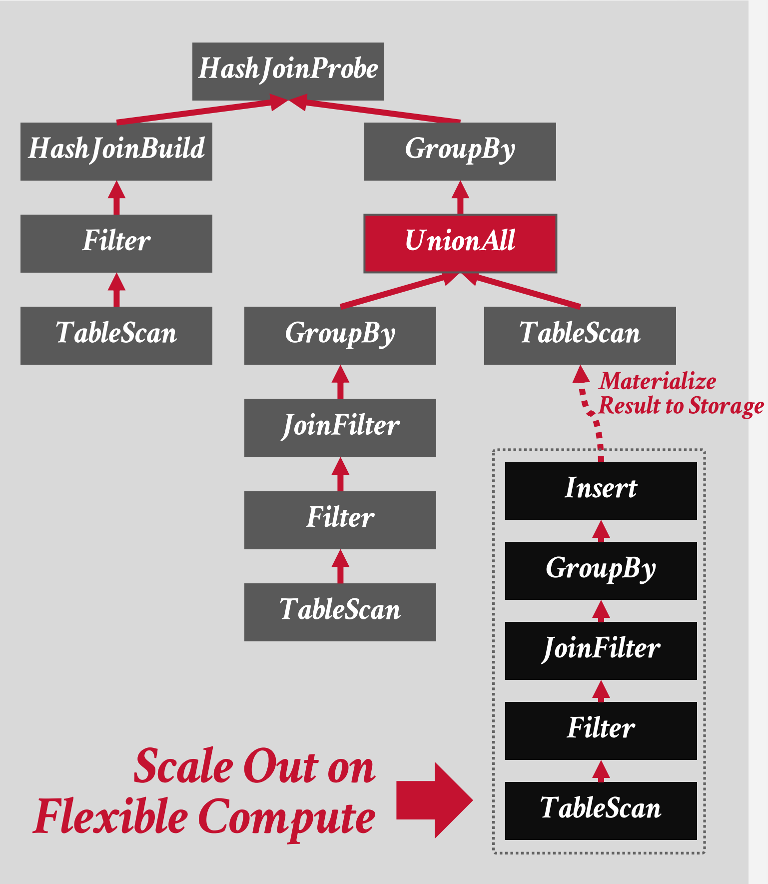


Source: Libo Wang

SNOWFLAKE: 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: 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.

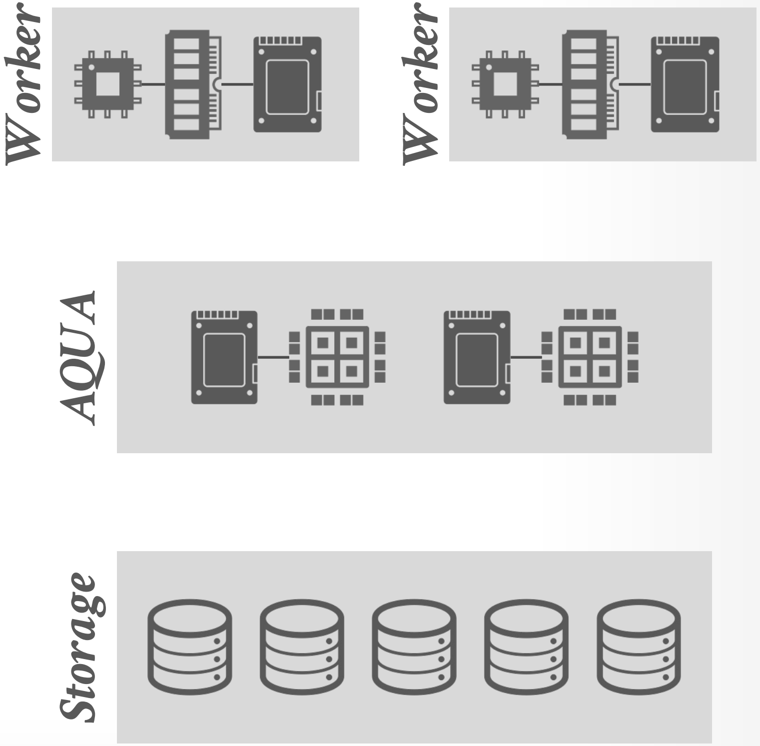
Background workers proactively recompile plans when new version of DBMS is released.

REDSHIFT: HARDWARE ACCELERATION

AWS introduced the **AQUA** (Advanced Query Accelerator) for Redshift (Spectrum?) in 2021.

Separate compute/cache nodes that use FPGAs to evaluate predicates.

AQUA was phased out and replaced with Nitro cards on compute nodes

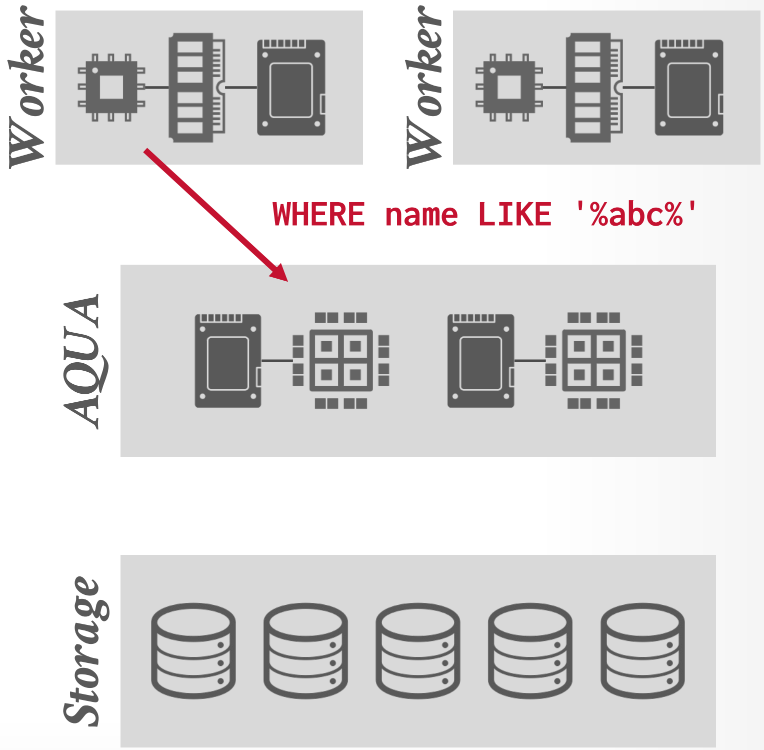


REDSHIFT: HARDWARE ACCELERATION

AWS introduced the **AQUA** (Advanced Query Accelerator) for Redshift (Spectrum?) in 2021.

Separate compute/cache nodes that use FPGAs to evaluate predicates.

AQUA was phased out and replaced with Nitro cards on compute nodes

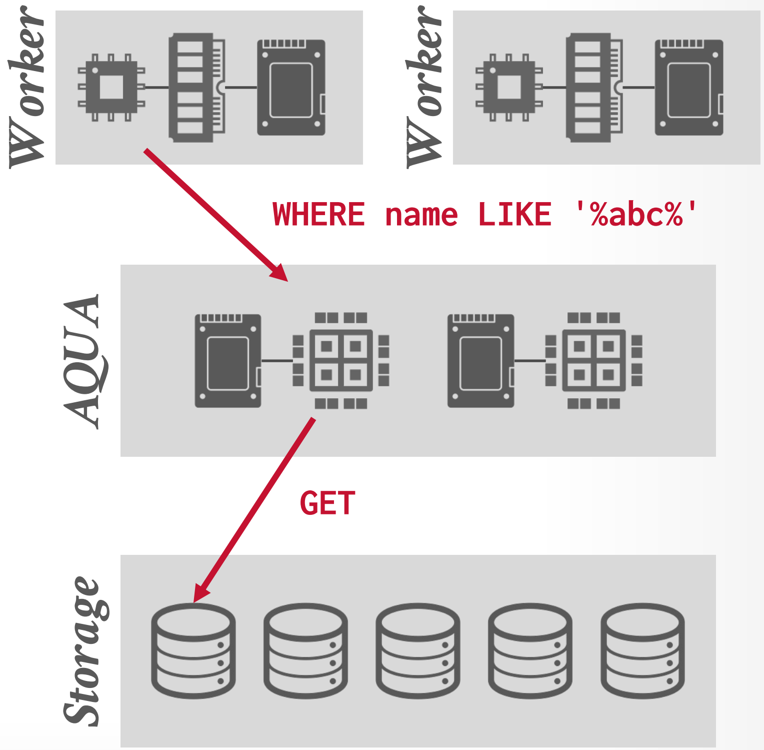


REDSHIFT: HARDWARE ACCELERATION

AWS introduced the **AQUA** (Advanced Query Accelerator) for Redshift (Spectrum?) in 2021.

Separate compute/cache nodes that use FPGAs to evaluate predicates.

AQUA was phased out and replaced with Nitro cards on compute nodes



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 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: 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.

PHOTON: EXPRESSION FUSION

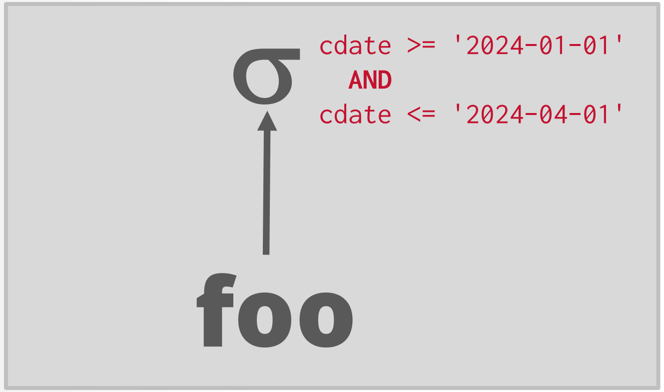
```
SELECT * FROM foo
 WHERE cdate BETWEEN '2024-01-01' AND '2024-04-01';
```

PHOTON: EXPRESSION FUSION


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

PHOTON: EXPRESSION FUSION

```
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';
```

σ cdate >= '2024-01-01'
AND
cdate <= '2024-04-01'

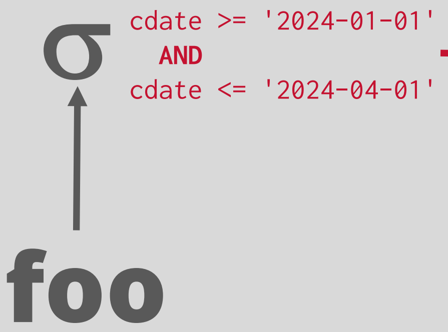
foo

```
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);
}
```

```
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';
```




```cpp
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';
```

σ cdate >= '2024-01-01'
AND
cdate <= '2024-04-01'

foo

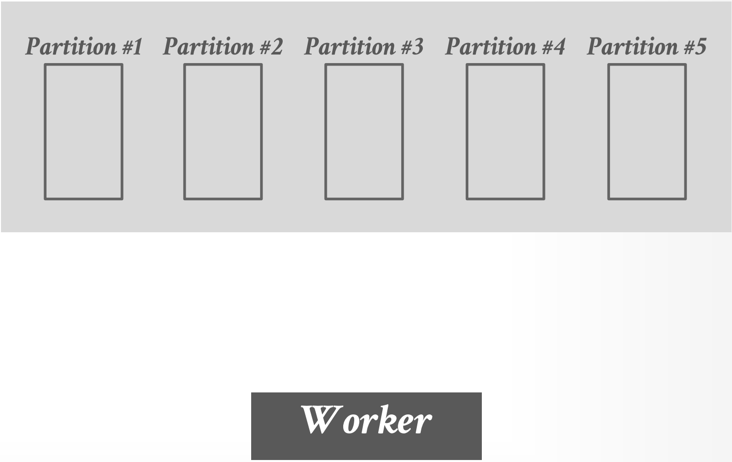
```cpp
vec<offset> sel_between_dates(vec<date> batch,
                              date low, date high) {
  vec<offset> positions;
  for (offset i = 0; i < batch.size(); i++)
    if (batch[i] >= low && batch[i] <= high)
      positions.append(i);
  return (positions);
}
```

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.



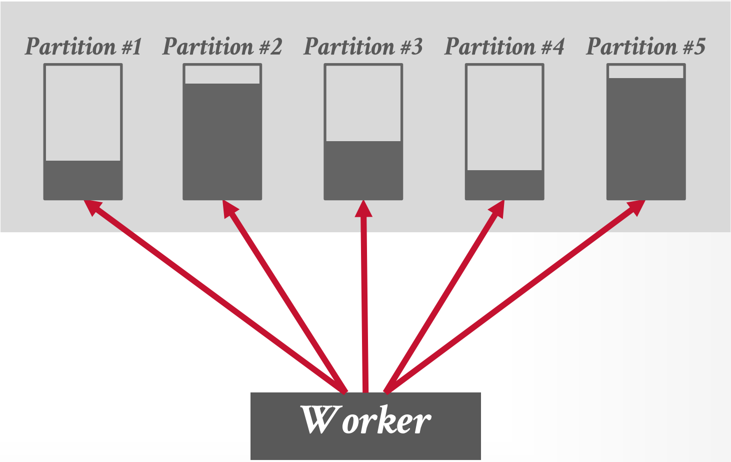
Source: Maryann Xue

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

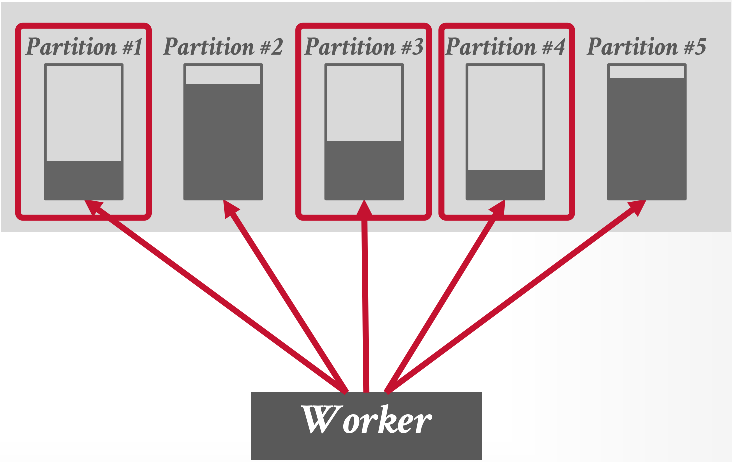
databricks

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.



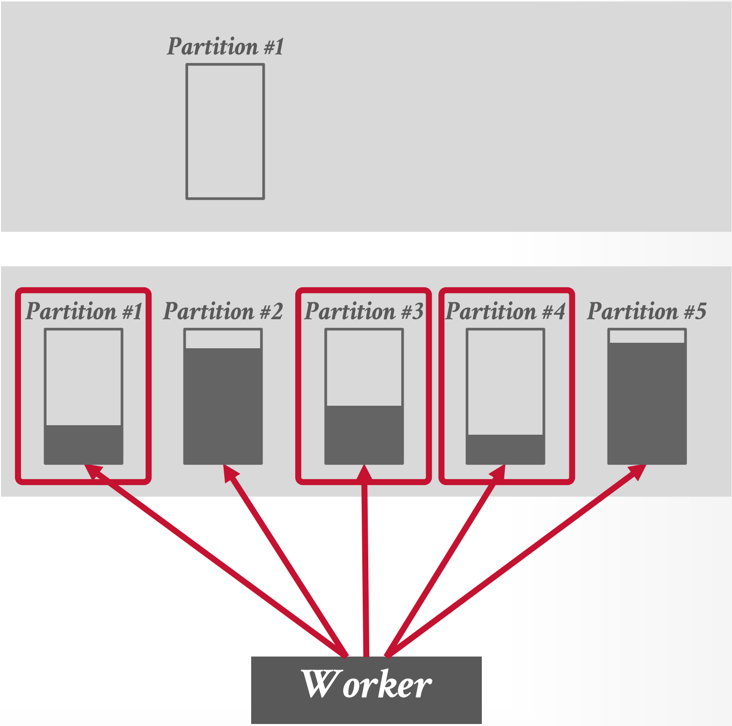
Source: Maryann Xue

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.

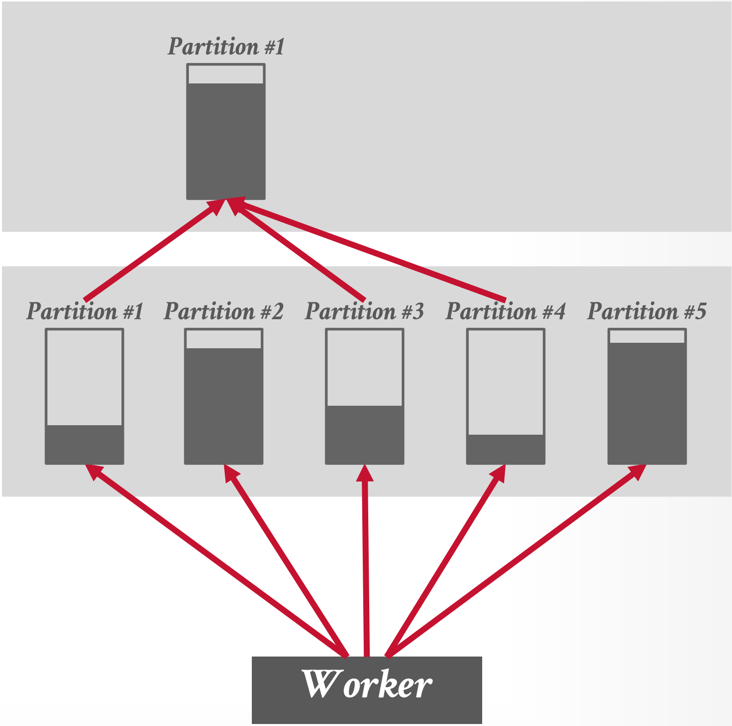


Source: Maryann Xue

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

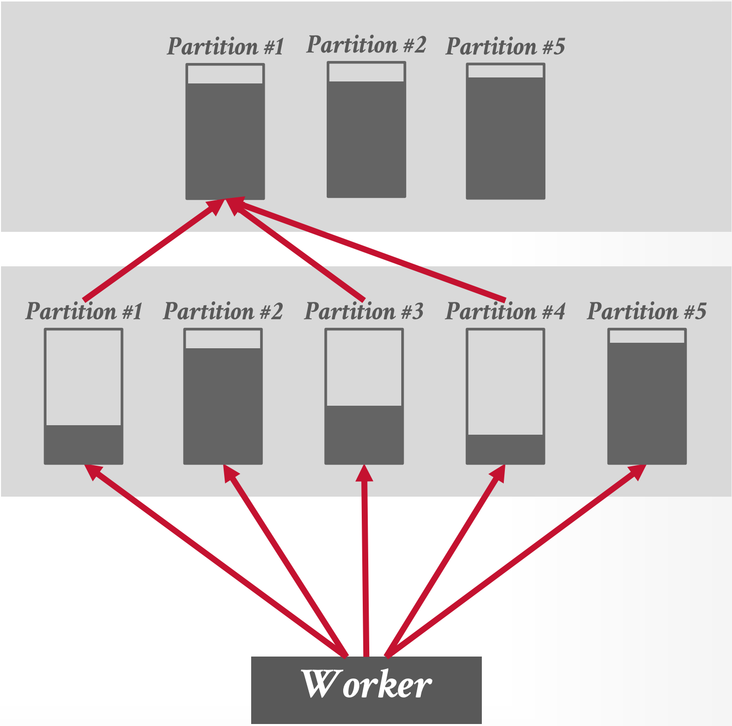
databricks

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

DuckDB

DuckDB

DUCKDB (2019)

Multi-threaded embedded (in-process, serverless) DBMS that executes SQL over disparate data files.

→ PostgreSQL-like dialect with quality-of-life enhancements.
→ ***"SQLite for Analytics"***

Provides zero-copy access to query results via Arrow to client code running in same process.

The core DBMS is nearly all custom C++ code with little to no third-party dependencies.

→ Relies on extensions ecosystem to expand capabilities.

DUCKDB (2019)

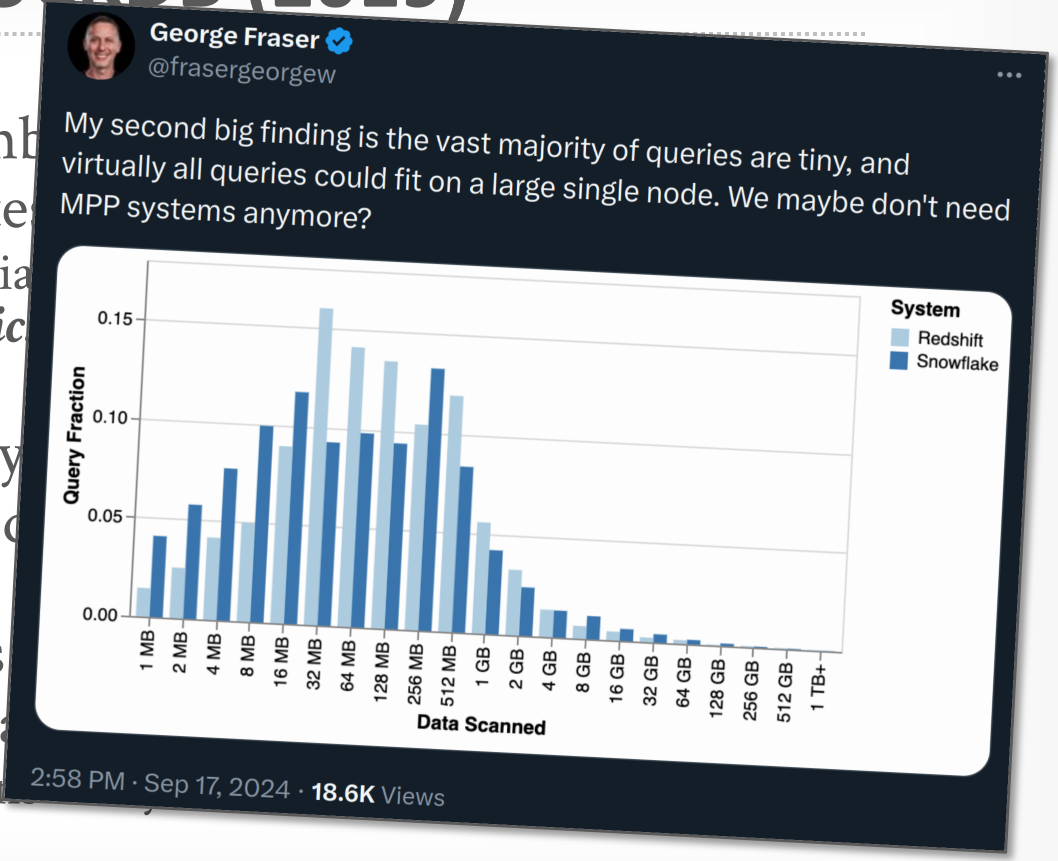
Multi-threaded emb
DBMS that execute

→ PostgreSQL-like dia
→ *"SQLite for Analytic*

Provides zero-copy
Arrow to client coo

The core DBMS is
little to no third-pa

→ Relies on extensio



DUCKDB: OVERVIEW

Shared-Everything

Push-based Vectorized Query Processing

Precompiled Primitives

Multi-Version Concurrency Control

Morsel Parallelism + Scheduling

PAX Columnar Storage

Sort-Merge + Hash Joins

Stratified Query Optimizer

DUCKDB: OVERVIEW

Shared-Everything

Push-based Vectorized Query Processing

Precompiled Primitives

Multi-Version Concurrency Control

Morsel Parallelism + Scheduling

PAX Columnar Storage

Sort-Merge + Hash Joins

Stratified Query Optimizer

DUCKDB: PUSH-BASED PROCESSING

System originally used pull-based vectorized query processing but found it unwieldly to expand to support more complex parallelism.
→ Cannot invoke multiple pipelines simultaneously.

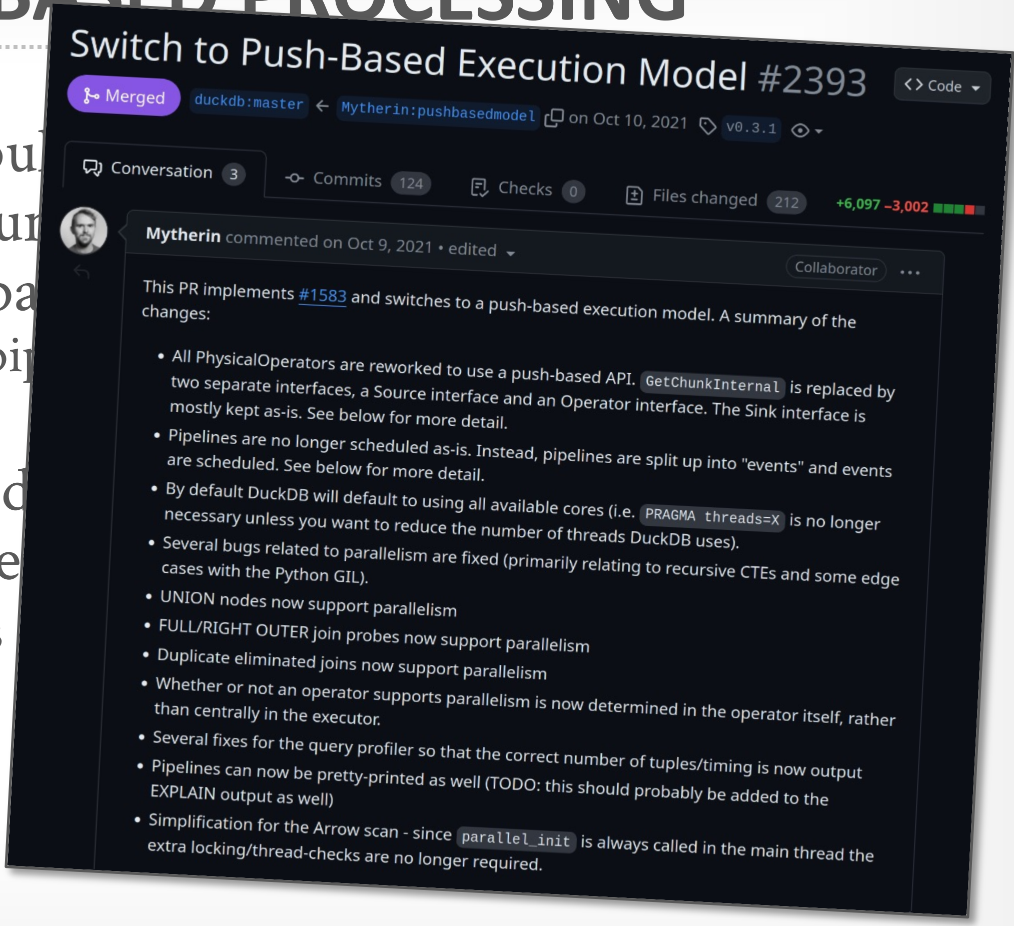
Switched to a push-based query processing model in 2021. Each operator determines whether it will execute in parallel on its own instead of a centralized executor.

DuckDB

DUCKDB: PUSH-BASED PROCESSING

System originally used pul
processing but found it ur
support more complex pa
→ Cannot invoke multiple pi

Switched to a push-based
2021. Each operator dete
execute in parallel on its
centralized executor.


Switch to Push-Based Execution Model #2393

Code

Merged duckdb:master ← Mytherin:pushbasedmodel on Oct 10, 2021 v0.3.1

Conversation 3 Commits 124 Checks 0 Files changed 212 +6,097 −3,002

Mytherin commented on Oct 9, 2021 • edited

Collaborator

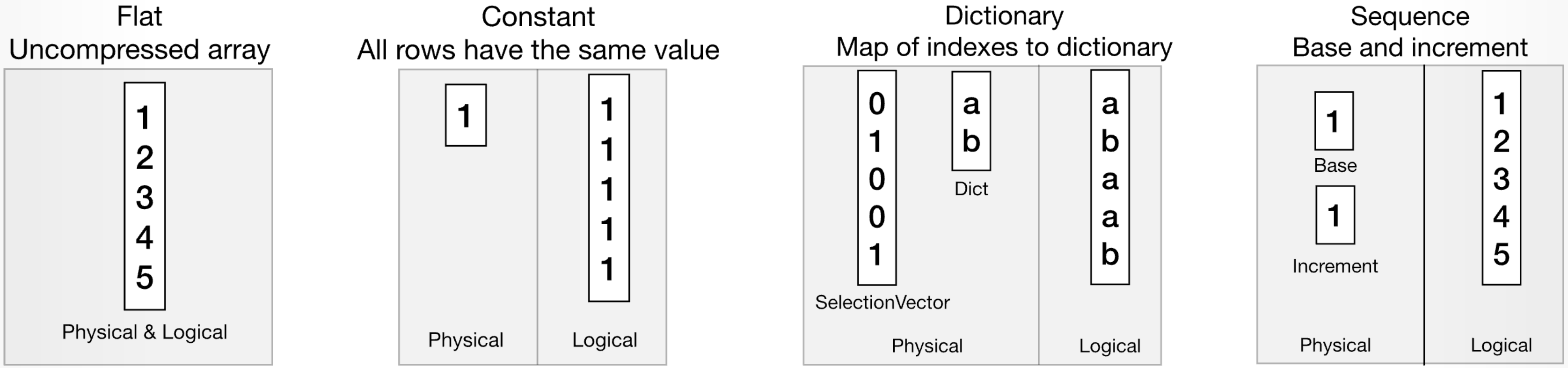
This PR implements #1583 and switches to a push-based execution model. A summary of the changes:

- All PhysicalOperators are reworked to use a push-based API. `GetChunkInternal` is replaced by two separate interfaces, a Source interface and an Operator interface. The Sink interface is mostly kept as-is. See below for more detail.
- Pipelines are no longer scheduled as-is. Instead, pipelines are split up into "events" and events are scheduled. See below for more detail.
- By default DuckDB will default to using all available cores (i.e. `PRAGMA threads=X` is no longer necessary unless you want to reduce the number of threads DuckDB uses).
- Several bugs related to parallelism are fixed (primarily relating to recursive CTEs and some edge cases with the Python GIL).
- UNION nodes now support parallelism
- FULL/RIGHT OUTER join probes now support parallelism
- Duplicate eliminated joins now support parallelism
- Whether or not an operator supports parallelism is now determined in the operator itself, rather than centrally in the executor.
- Several fixes for the query profiler so that the correct number of tuples/timing is now output
- Pipelines can now be pretty-printed as well (TODO: this should probably be added to the EXPLAIN output as well)
- Simplification for the Arrow scan - since `parallel_init` is always called in the main thread the extra locking/thread-checks are no longer required.

DUCKDB: VECTORS

Custom internal vector layout for intermediate results that is compatible with Velox.

Supports multiple vector types:



Source: Mark Raasveldt

DUCKDB: VECTORS

DuckDB uses a unified format to process all vector types without needing to decompress them first.

→ Reduce # of specialized primitives per vector type

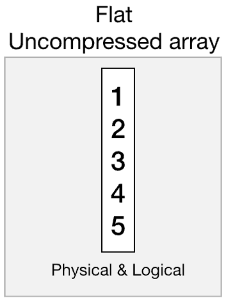
Source: Mark Raasveldt

DuckDB

DUCKDB: VECTORS

DuckDB uses a unified format to process all vector types without needing to decompress them first.

→ Reduce # of specialized primitives per vector type

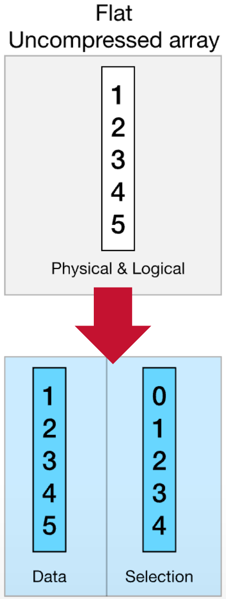


Source: Mark Raasveldt

DUCKDB: VECTORS

DuckDB uses a unified format to process all vector types without needing to decompress them first.

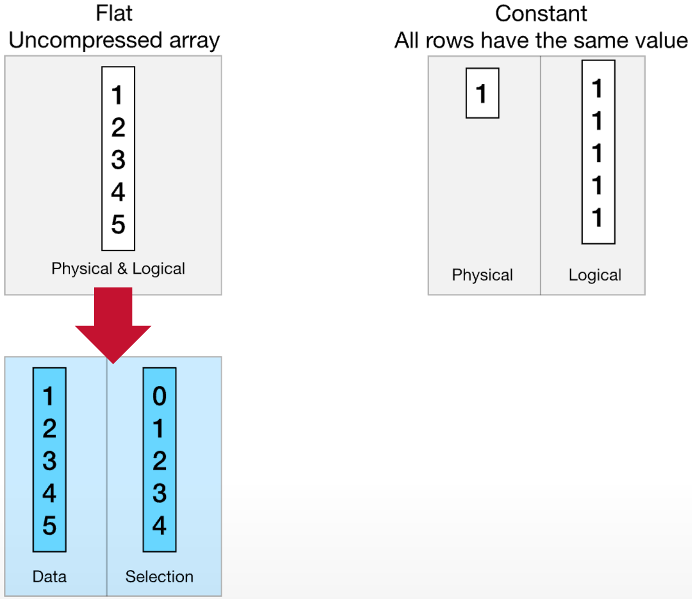
→ Reduce # of specialized primitives per vector type



Source: Mark Raasveldt

DUCKDB: VECTORS

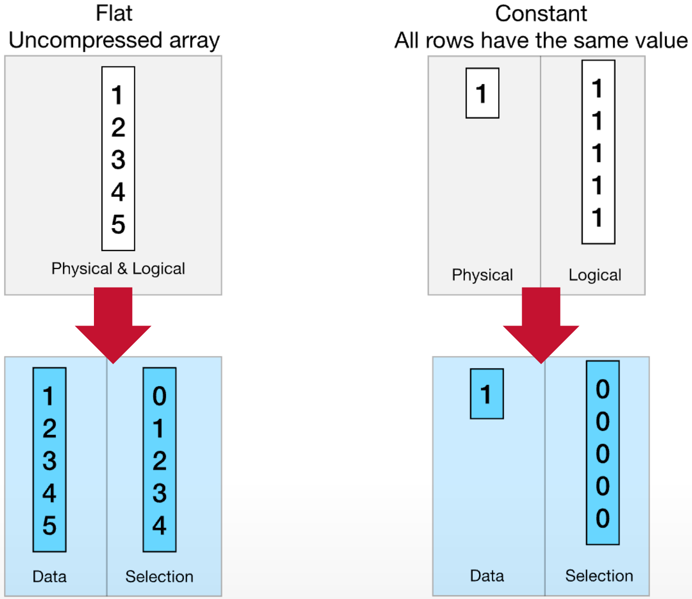
DuckDB uses a unified format to process all vector types without needing to decompress them first.
→ Reduce # of specialized primitives per vector type



Source: Mark Raasveldt

DUCKDB: VECTORS

DuckDB uses a unified format to process all vector types without needing to decompress them first.
→ Reduce # of specialized primitives per vector type

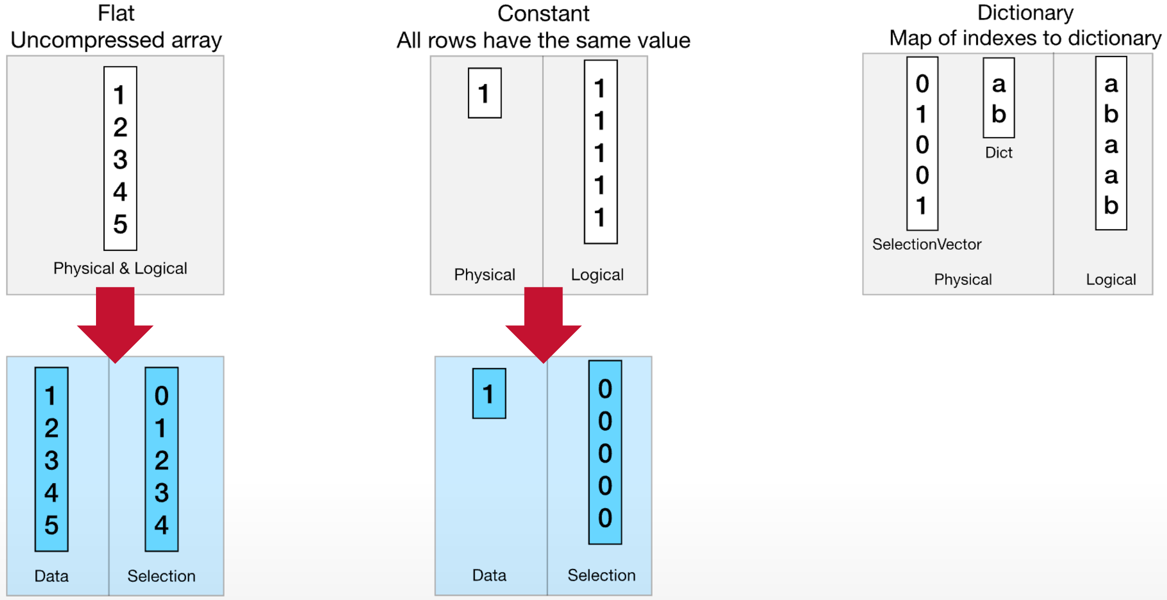


Source: Mark Raasveldt

DUCKDB: VECTORS

DuckDB uses a unified format to process all vector types without needing to decompress them first.

→ Reduce # of specialized primitives per vector type

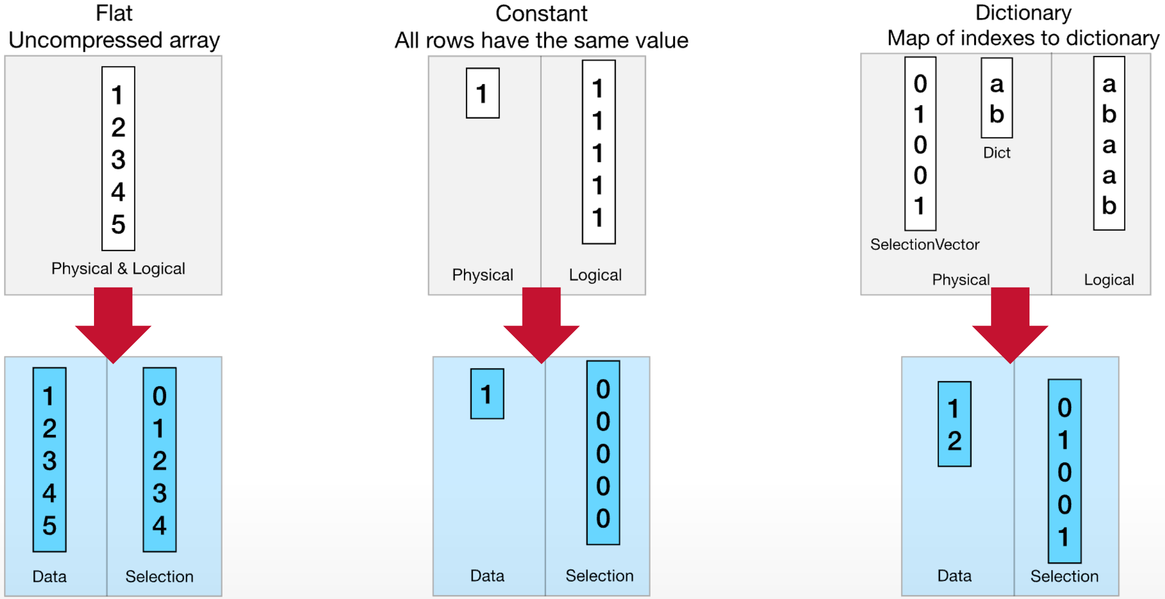


Source: Mark Raasveldt

DuckDB

DUCKDB: VECTORS

DuckDB uses a unified format to process all vector types without needing to decompress them first.
→ Reduce # of specialized primitives per vector type

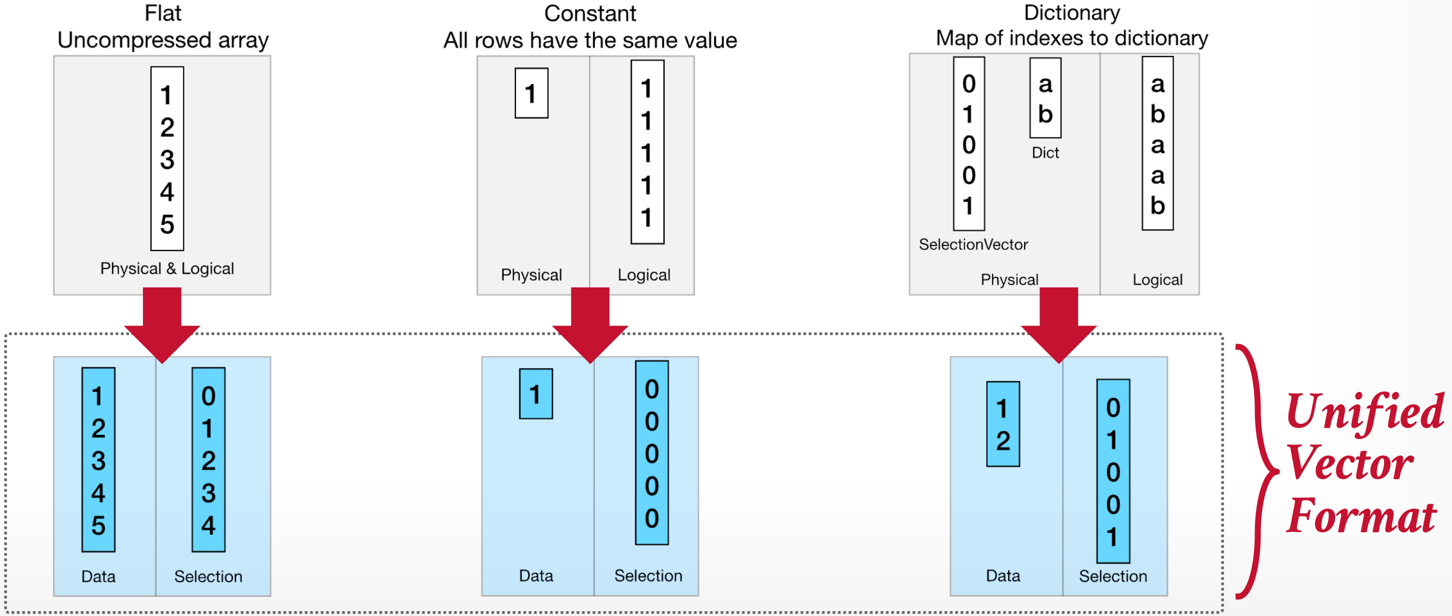


Source: Mark Raasveldt

DUCKDB: VECTORS

DuckDB uses a unified format to process all vector types without needing to decompress them first.

→ Reduce # of specialized primitives per vector type



Source: Mark Raasveldt