Carnegie Mellon University

Database Systems

Final Review & Systems Potpourri

15-445/645 FALL 2024 » PROF. ANDY PAVLO

ADMINISTRIVIA

Project #4 is due Sunday Dec 8th @ 11:59pm

Homework #6 is due Monday Dec 9th @ 11:59pm

Final Project Submission Deadline: Monday Dec 16th @ 11:59am

SPRING 2025

Jignesh is recruiting impressionable TAs for 15-445/645 in Spring 2025.

→ All BusTub projects will remain in C++.
→ **If you want to work on fixing BusTub over the winter break for money, please let us know.**

Sign up here:

https://www.ugrad.cs.cmu.edu/ta/S25

COURSE EVALS

Your feedback is strongly needed:
→ https://cmu.smartevals.com
→ https://www.ugrad.cs.cmu.edu/ta/F24/feedback/

Things that we want feedback on:
→ Homework Assignments
→ Projects
→ Reading Materials
→ Lectures

OFFICE HOURS

Andy:

→ Wednesday Dec 11th @ 3:30-4:30pm (GHC 9019)
→ Thursday Dec 12th @ 3:00-4:00pm (GHC 9019)
→ Or email me for an appt

Will:

→ Wednesday Dec 11th @ 10:30-11:30am (GHC 5th Floor Commons)

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

FINAL EXAM

Who: You

What: Final Exam

Where: Baker Hall A51

When: Friday Dec 13th @ 8:30-11:30am

Why: https://youtu.be/8tuoIO4CxOw

Email instructors if you need special accommodations.

https://15445.courses.cs.cmu.edu/fall2024/final-guide.html

FINAL EXAM

Everyone should come to BH A51.

You will then be assigned a random location in either A51 or A53.

There will be TAs stationed in each room to give you the exam and to handle questions.

Andy will bounce around the rooms during the exam time.

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

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
Partitioning Schemes
Two-Phase Commit

TOPICS NOT ON EXAM!

Flash Talks

Seminar Talks

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

Andy's legal troubles

CMU 15-721 (Spring 2024)

SPEED RUN

15721.courses.cs.cmu.edu/spring2024

SEQUENTIAL SCAN: OPTIMIZATIONS

- *Lecture #5* 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
- Code Specialization / Compilation

SELECTION SCANS

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

Source: Bogdan Raducanu

SELECTION SCANS

Scalar (Branching)

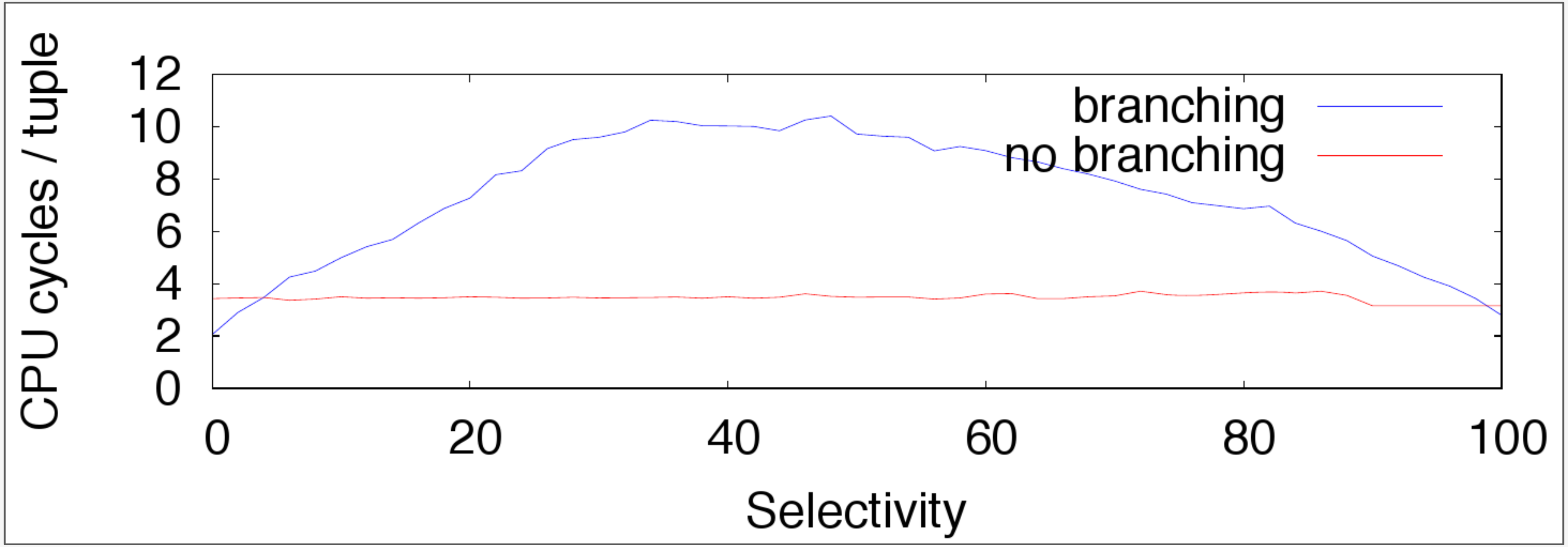
```
i = 0
for t in table:
  key = t.key
  if (key>low) && (key<high):
    copy(t, output[i])
    i = i + 1
```

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

VECTORIZED 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
```

VECTORIZED 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
```

VECTORIZED 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"
```

基于字符串键的查询.

VECTORIZED 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"
```

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	💩

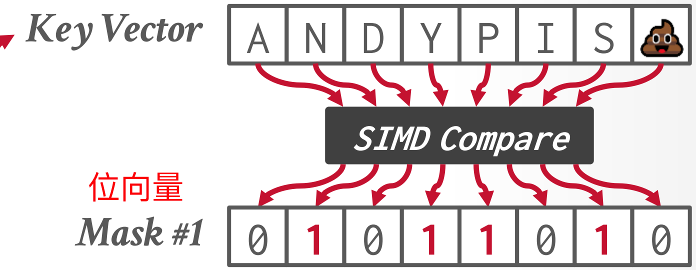
VECTORIZED SELECTION SCANS

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	💩



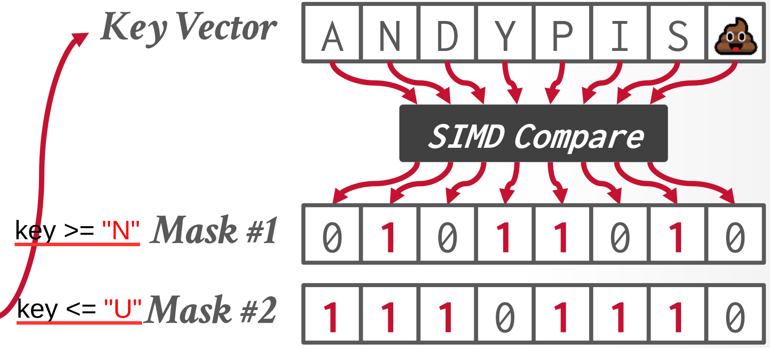
VECTORIZED SELECTION SCANS

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	💩



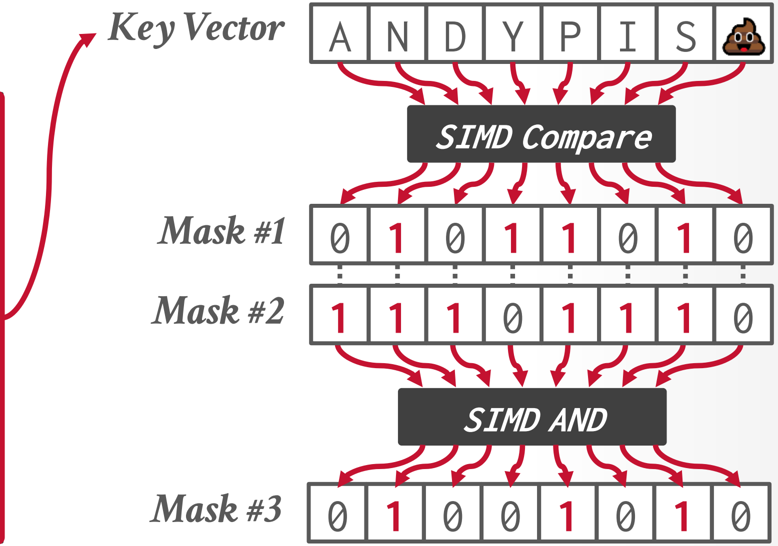
VECTORIZED SELECTION SCANS

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	💩



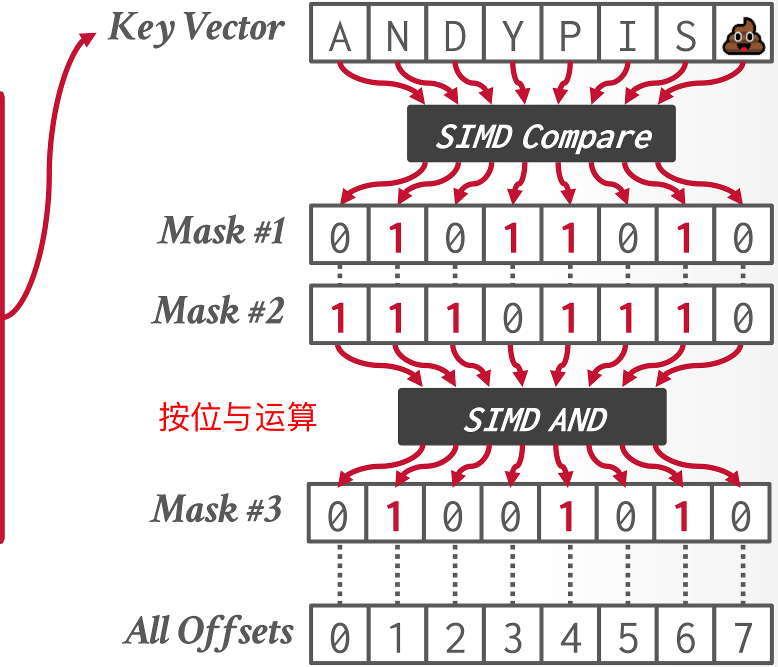
VECTORIZED SELECTION SCANS

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	💩



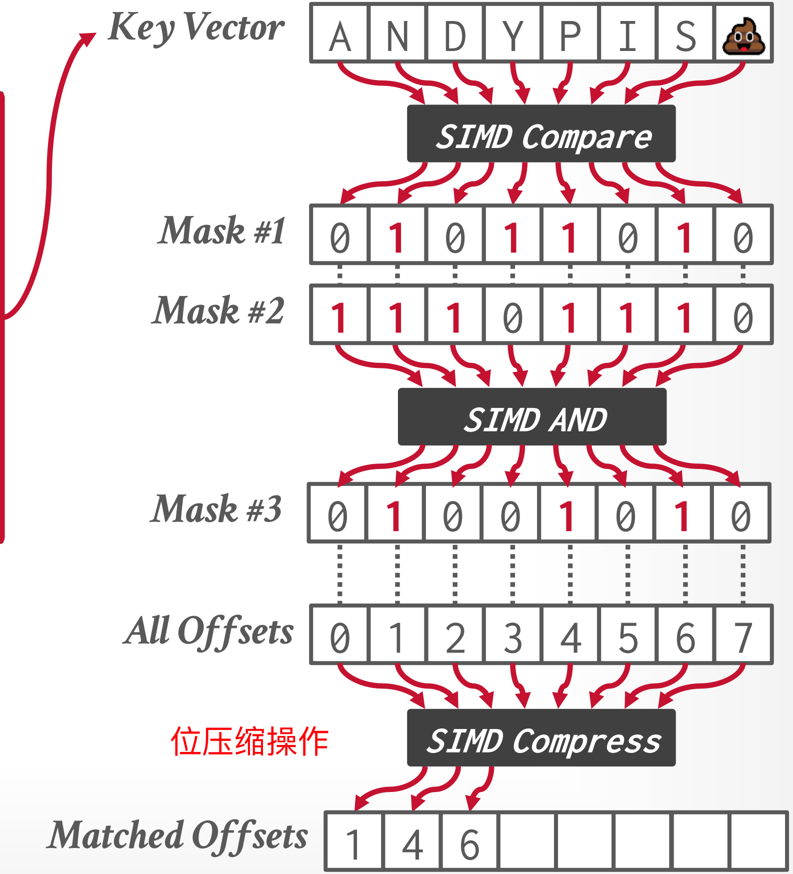
VECTORIZED SELECTION SCANS

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	💩



位 "1" 的元素与元素下标运算压缩得到有效的数组下标，
表示满足扫描过滤后的数据.

HIQUE: HOLISTIC CODE GENERATION

将查询编译为机器代码.

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)
```

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.

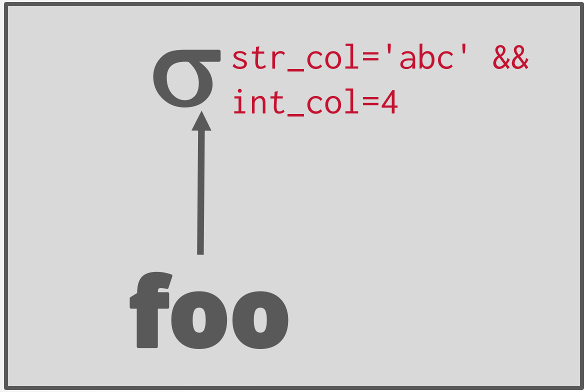
→ Function calls are amortized over multiple tuples.

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

MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

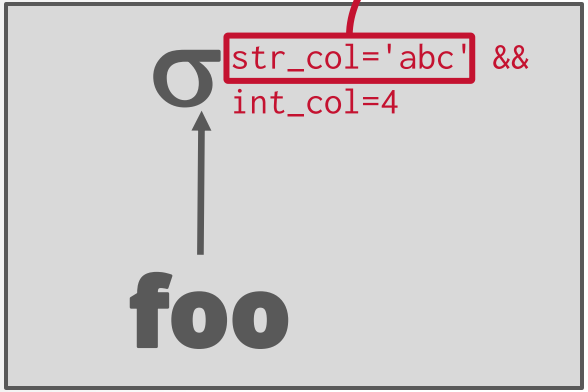
VECTORWISE: PRECOMPILED PRIMITIVES

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



VECTORWISE: PRECOMPILED PRIMITIVES

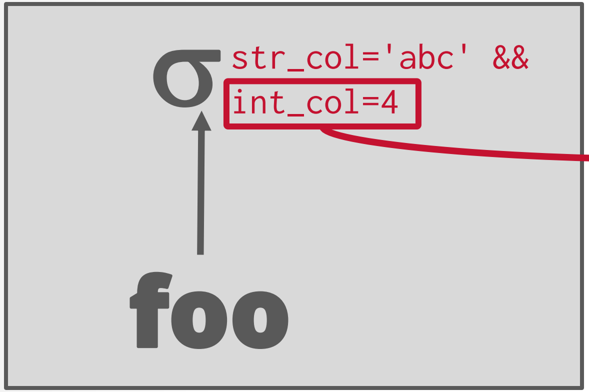
```
SELECT * FROM foo
WHERE str_col = 'abc'
  AND int_col = 4;
```




```
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

```
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)
DuckDB (2019)
TabDB (2019)

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

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.

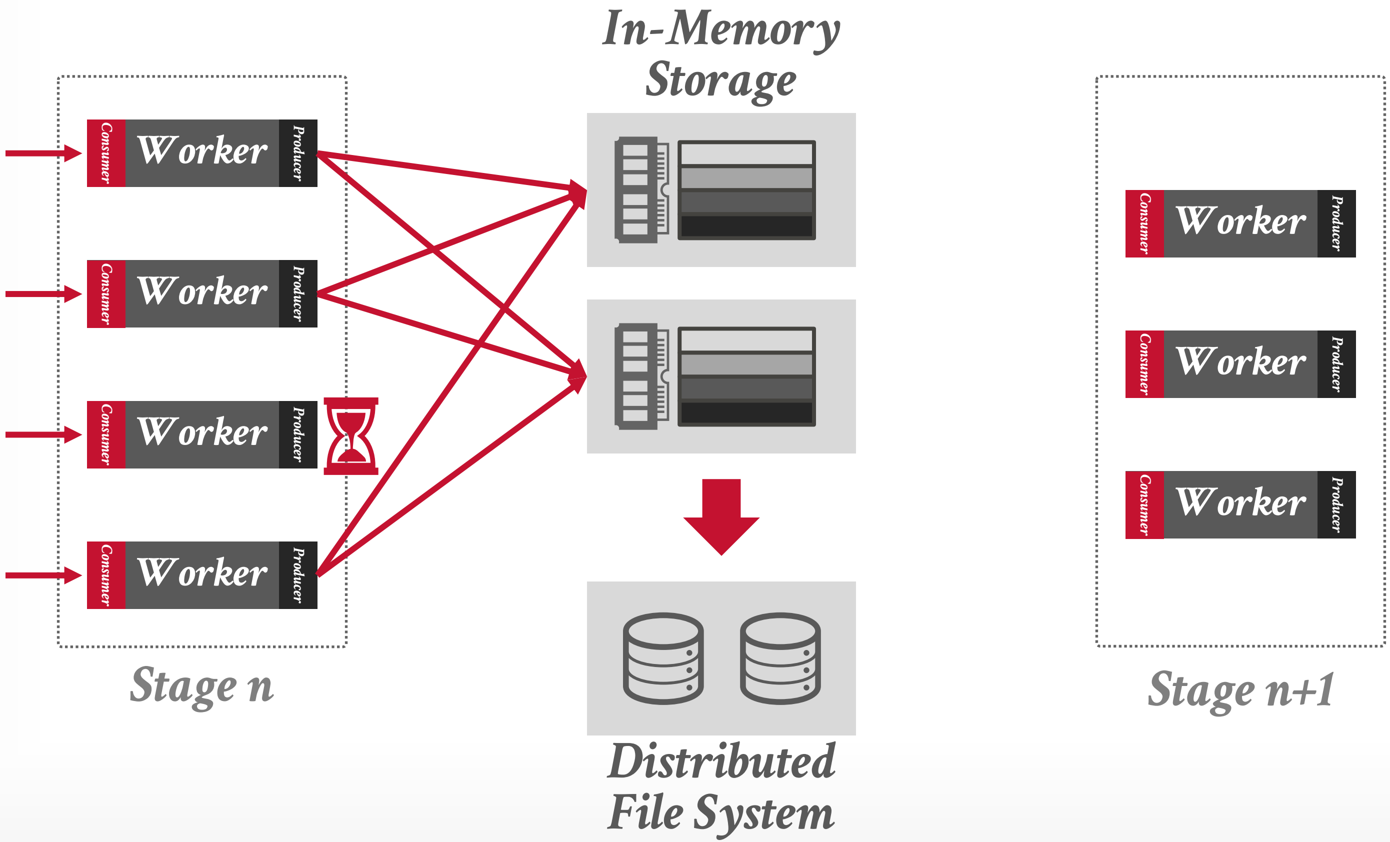
Fault Tolerance / Straggler Avoidance:

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

Dynamic Resource Allocation:

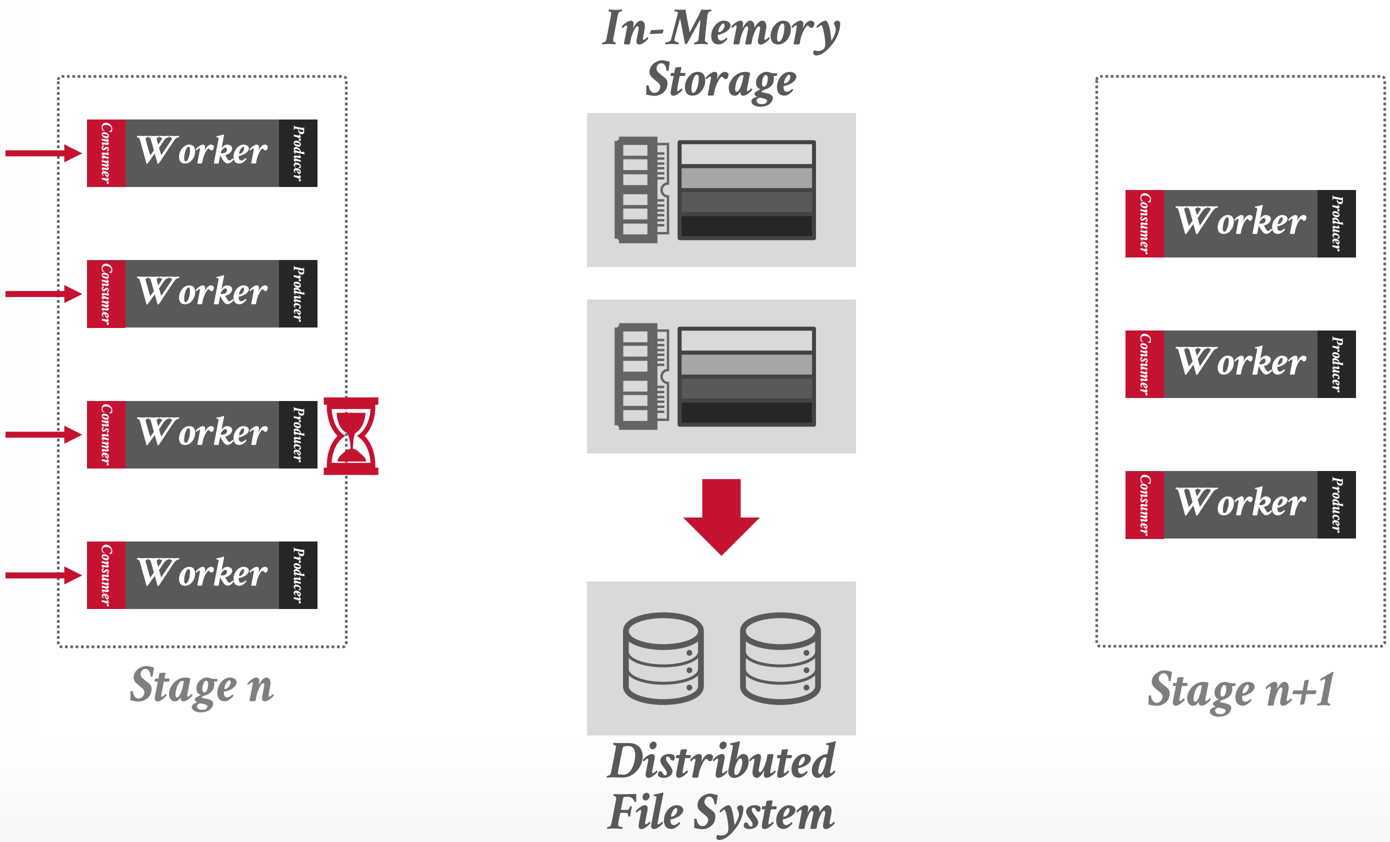
→ Scale up / down the number of workers for the next stage depending size of a stage's output.

BIGQUERY: IN-MEMORY SHUFFLE

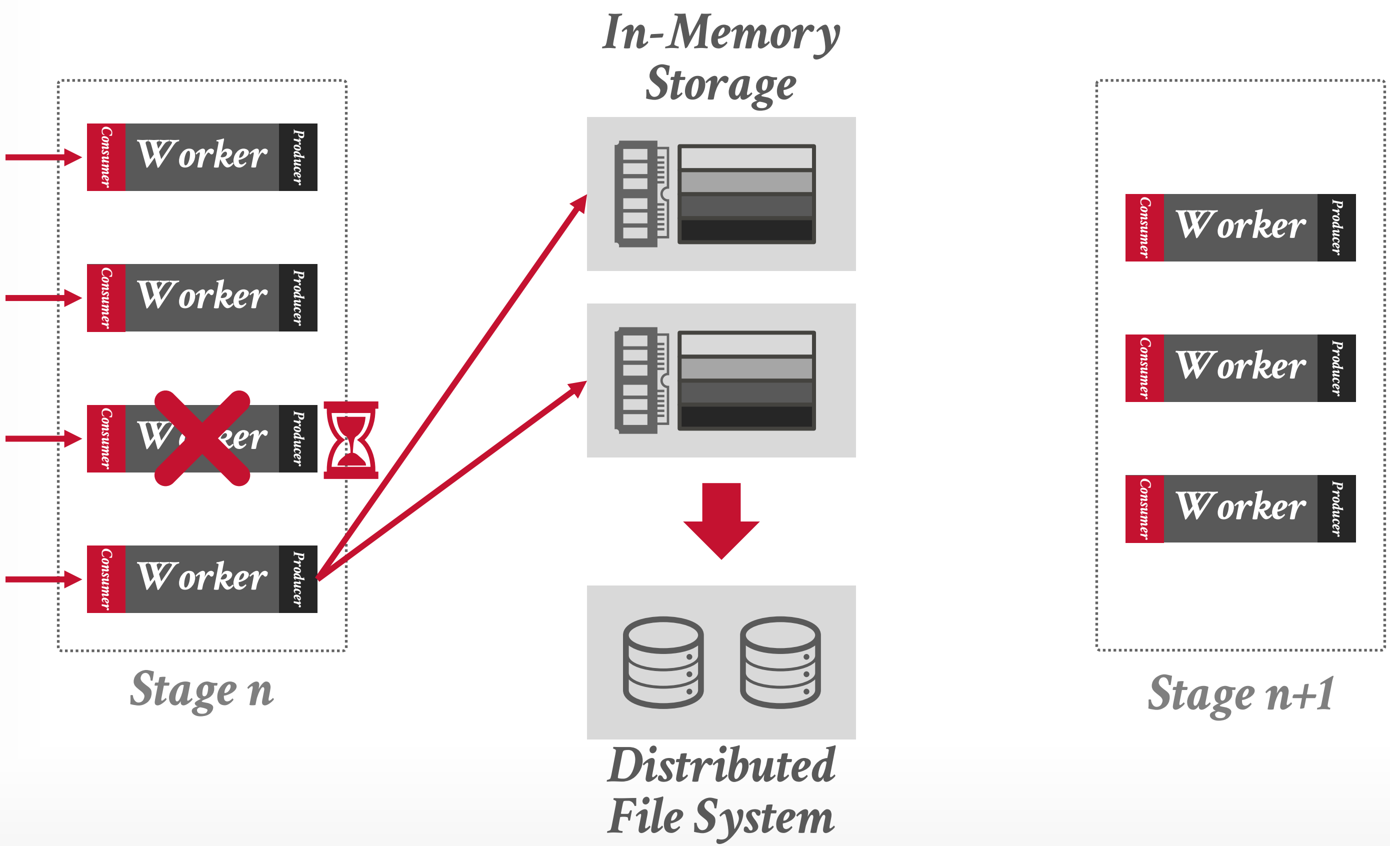


Google Big Query

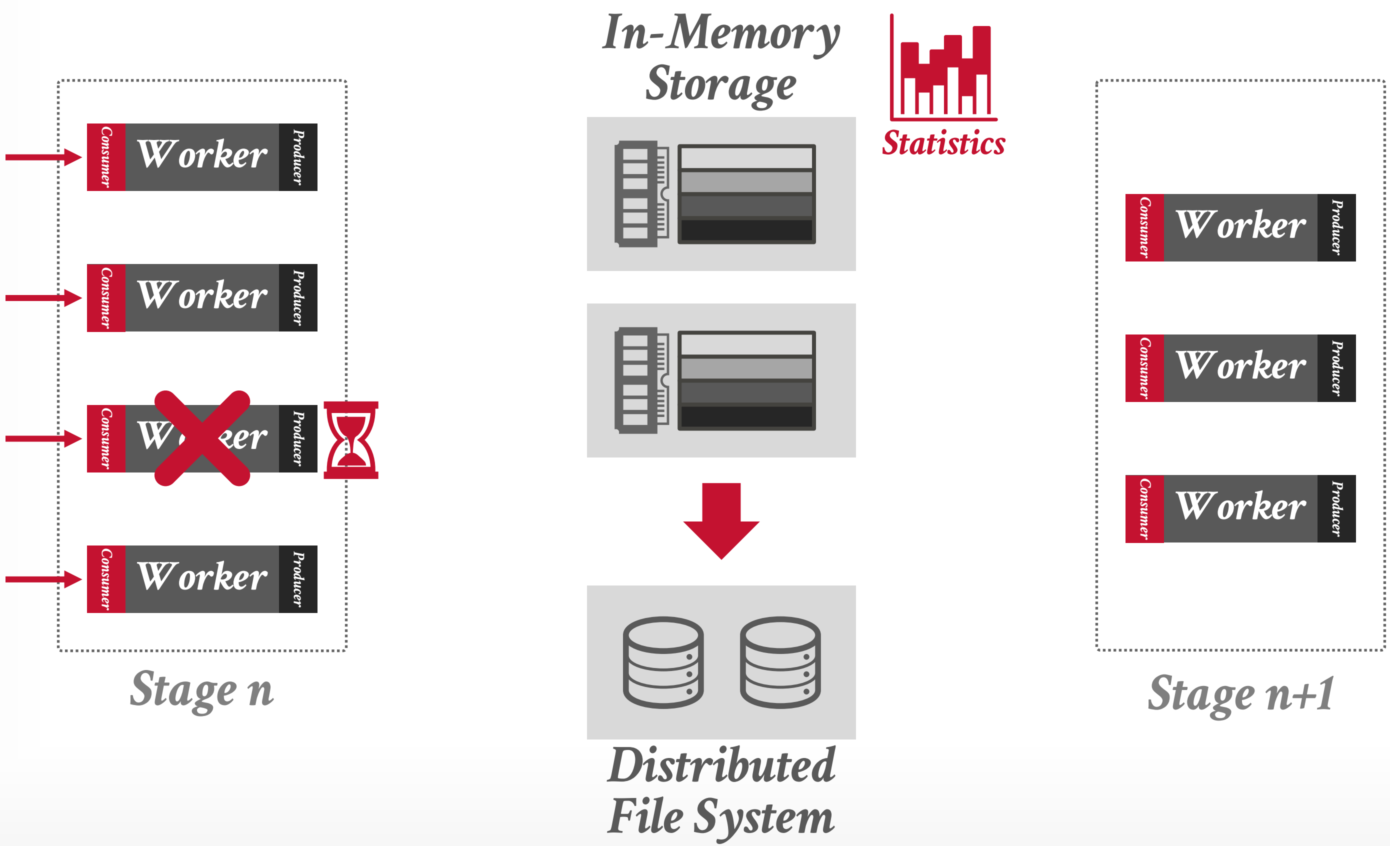
BIGQUERY: IN-MEMORY SHUFFLE



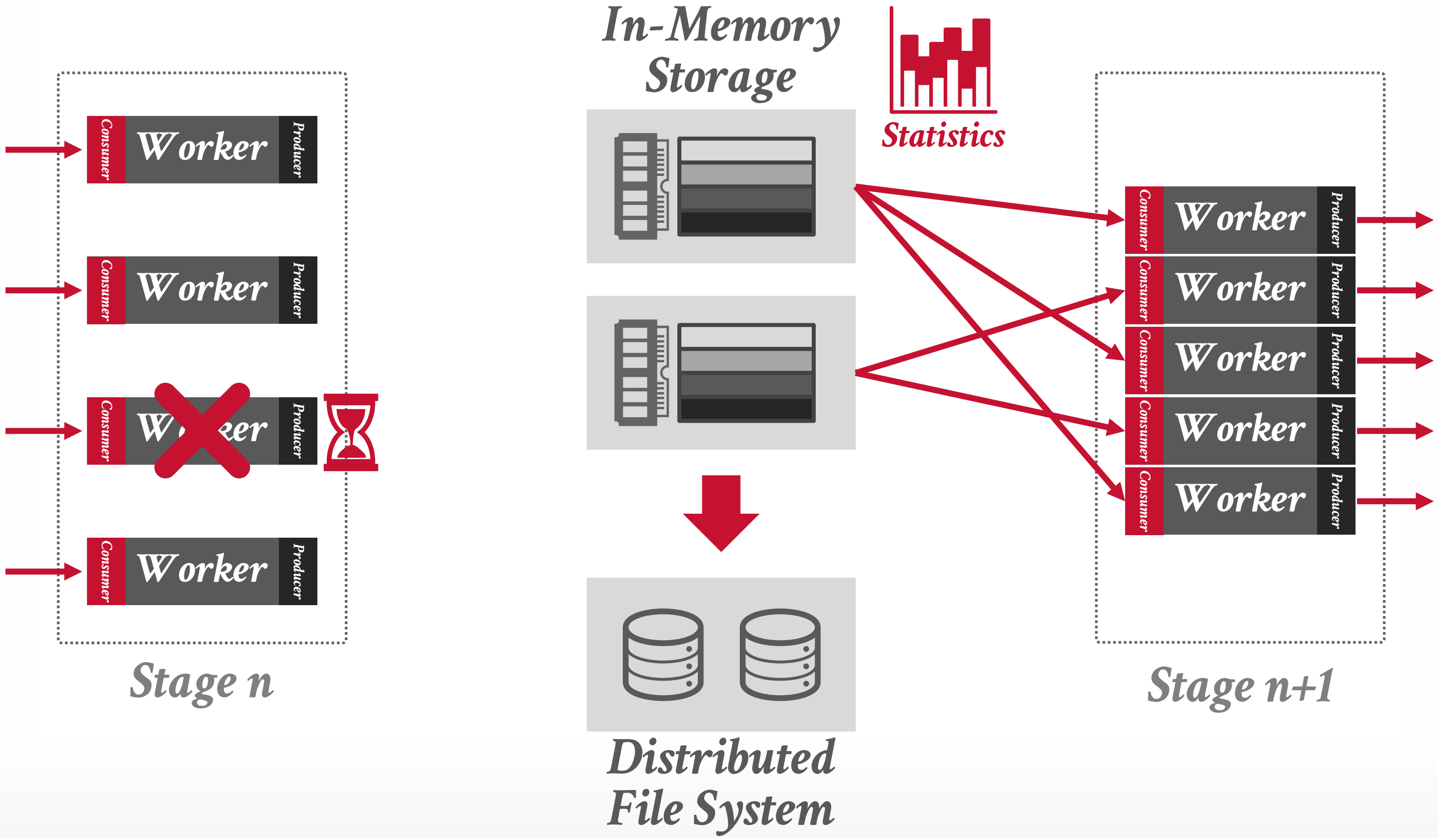
BIGQUERY: IN-MEMORY SHUFFLE



BIGQUERY: IN-MEMORY SHUFFLE



BIGQUERY: IN-MEMORY SHUFFLE

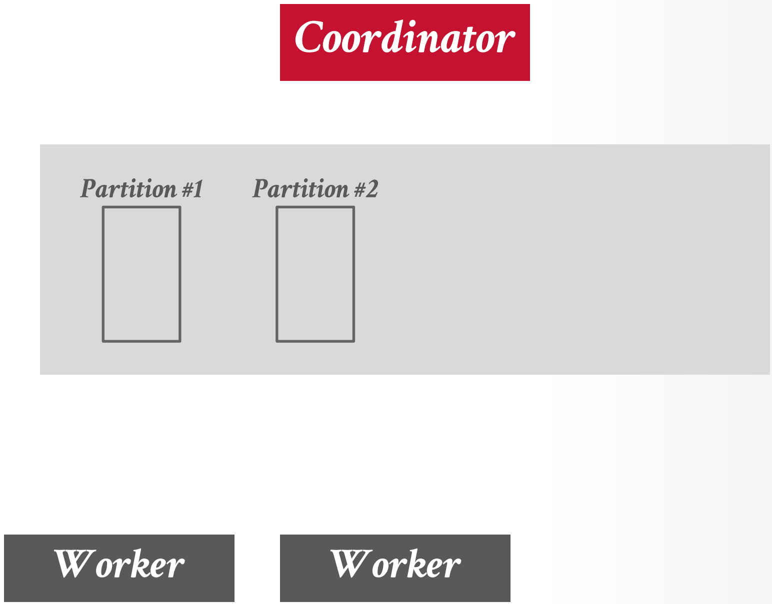


Google Big Query

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.

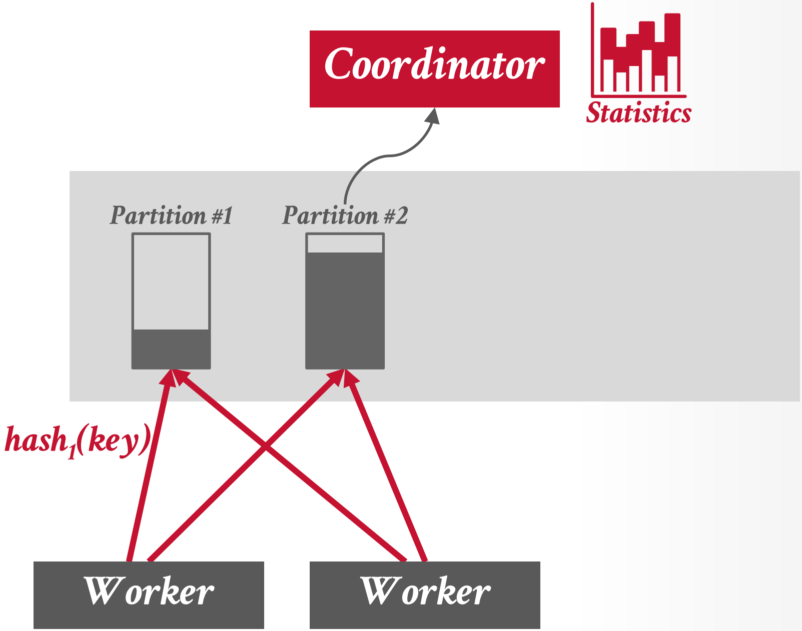


Source: H.Ahmadi + A.Surna

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.



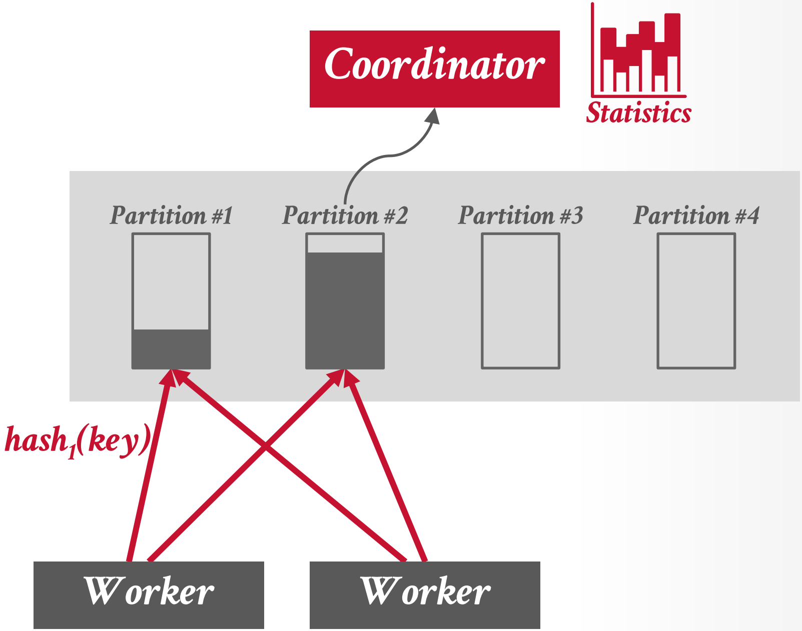
Source: H.Ahmadi + A.Surna

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.

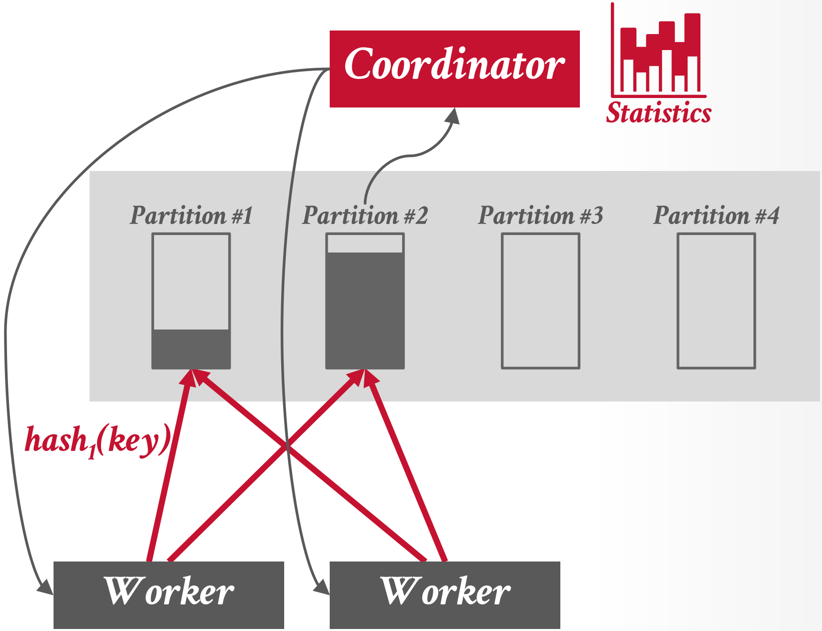


Source: H.Ahmadi + A.Surna

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.

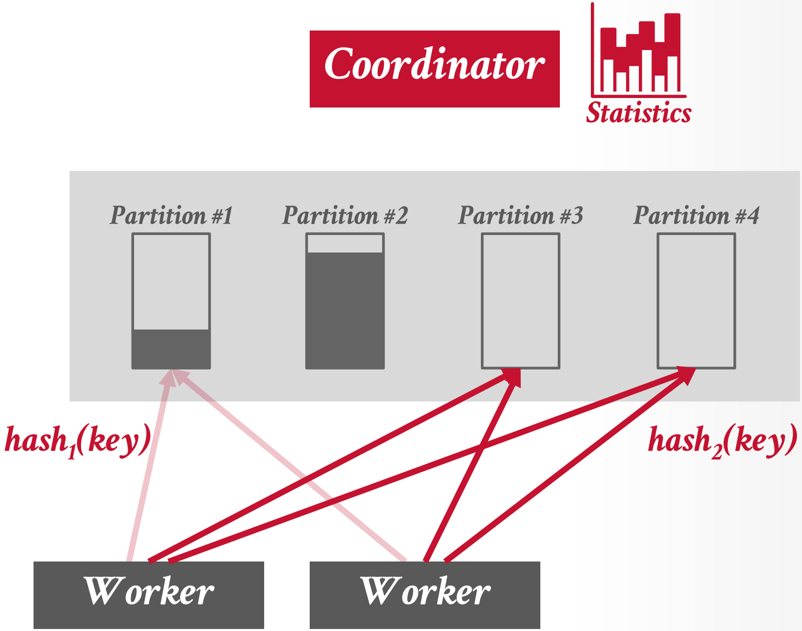


Source: H.Ahmadi + A.Surna

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.

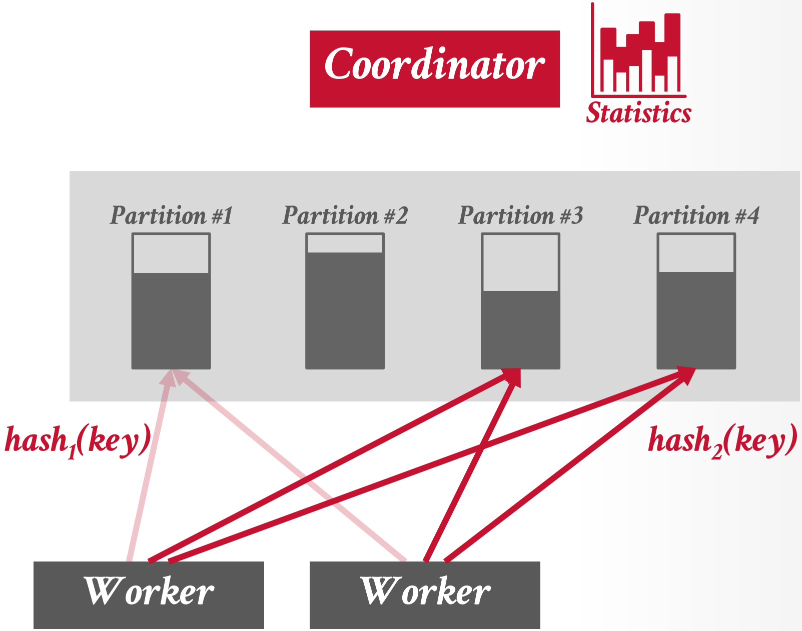


Source: H.Ahmadi + A.Surna

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.

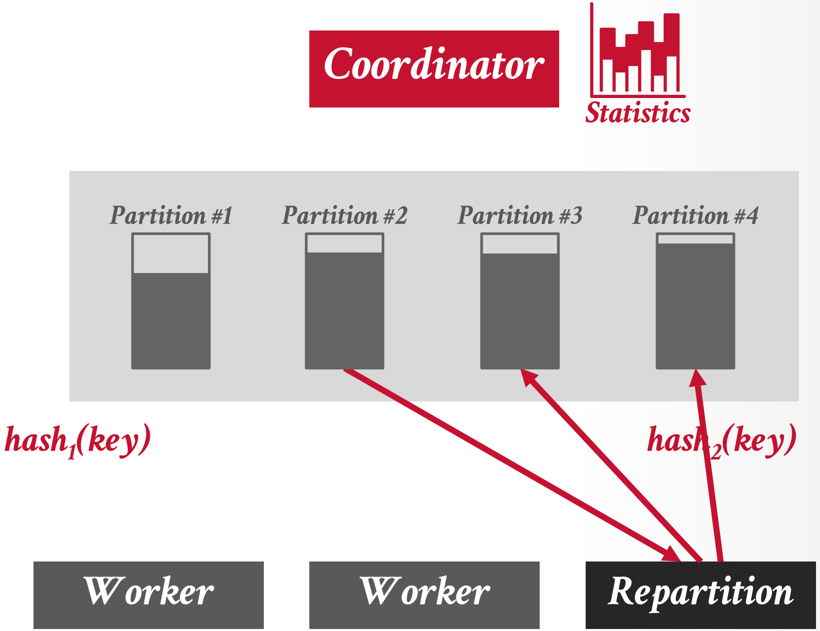


Source: H.Ahmadi + A.Surna

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.



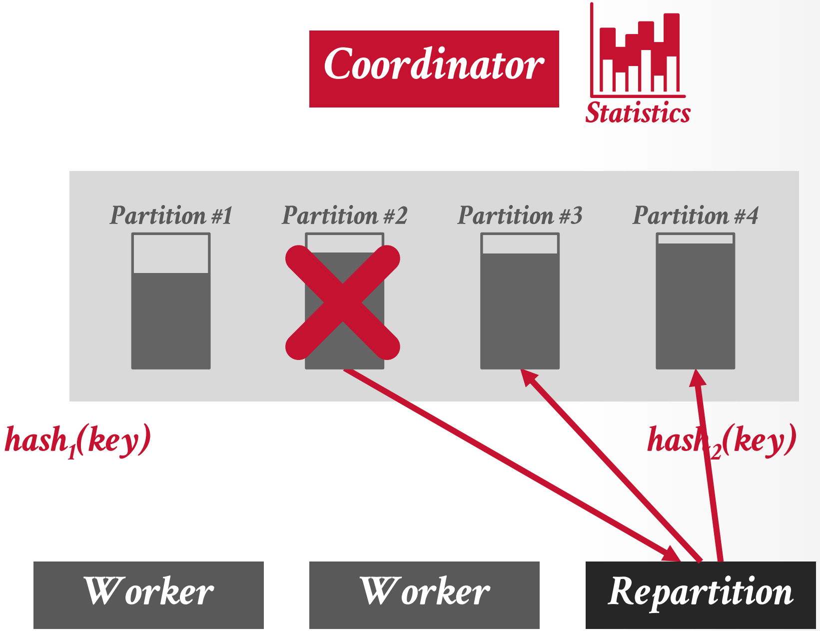
Source: H.Ahmadi + A.Surna

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.

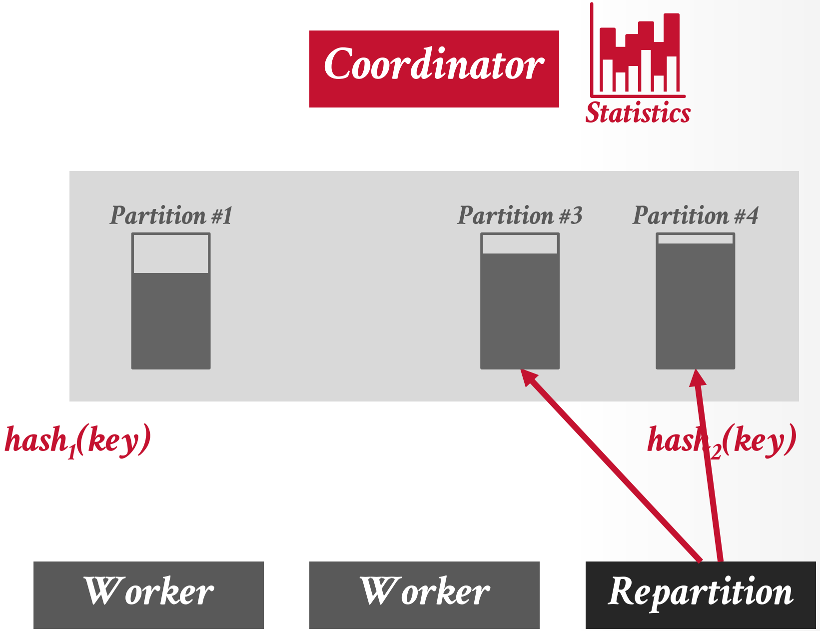


Source: H.Ahmadi + A.Surna

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

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: 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.

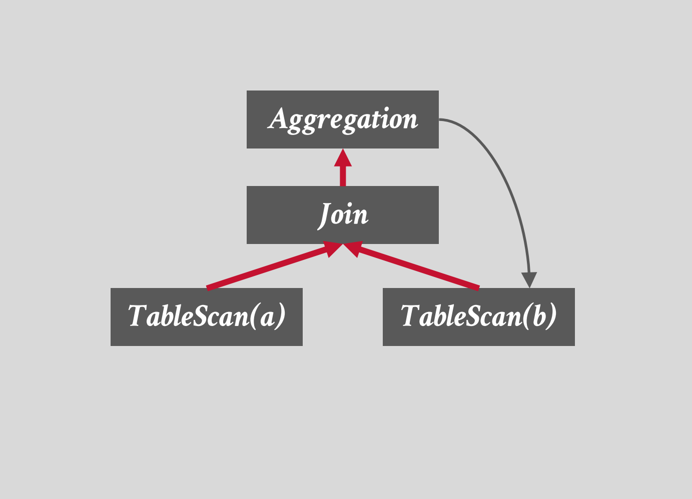
Does not support partial query retries

→ 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.

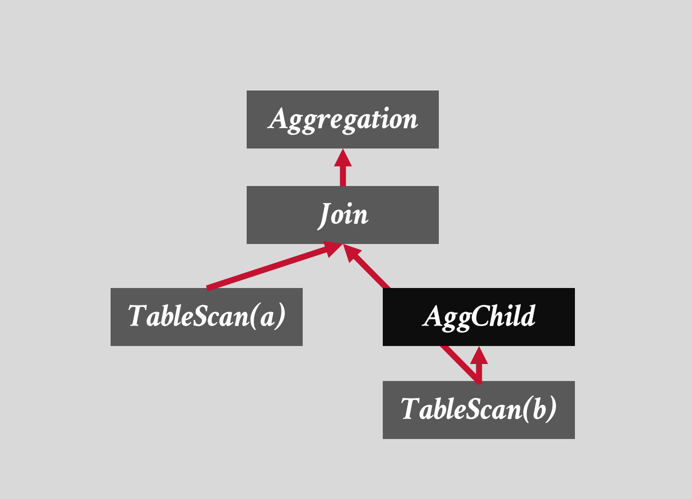


Source: Bowei Chen

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.

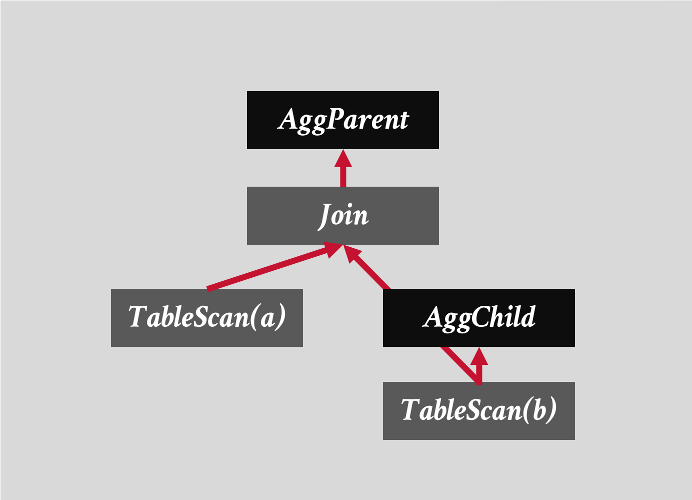


Source: Bowei Chen

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.

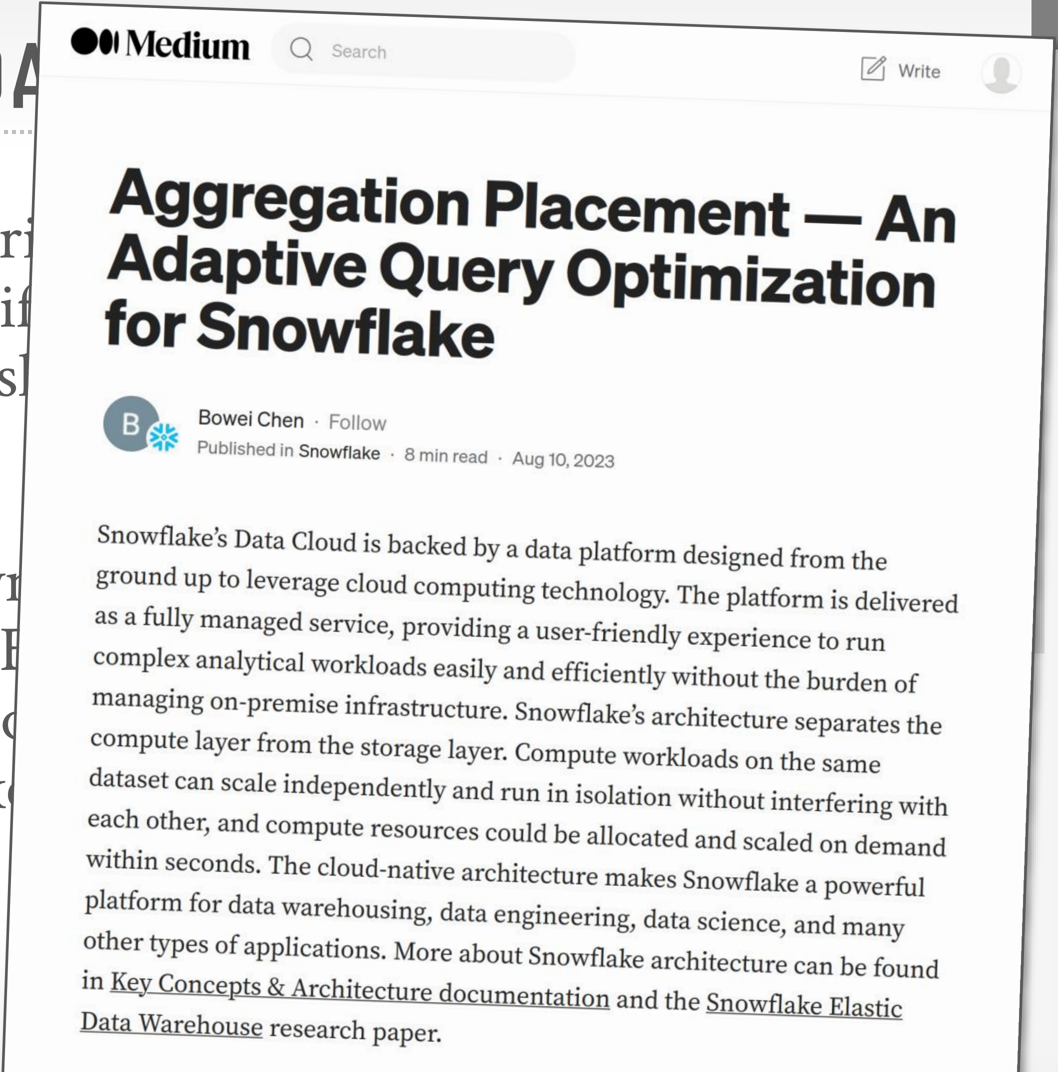


Source: Bowei Chen

SNOWFLAKE: ADA

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

The optimizer adds the dowr
aggregations but then the DE
enables them at runtime acco
statistics observed during exe



Aggregation Placement — An Adaptive Query Optimization for Snowflake

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

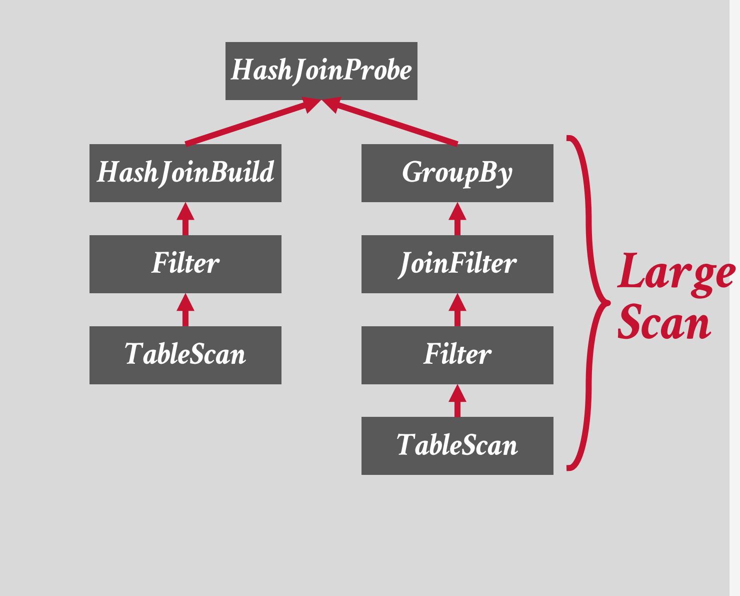
Snowflake's Data Cloud is backed by a data platform designed from the ground up to leverage cloud computing technology. The platform is delivered as a fully managed service, providing a user-friendly experience to run complex analytical workloads easily and efficiently without the burden of managing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

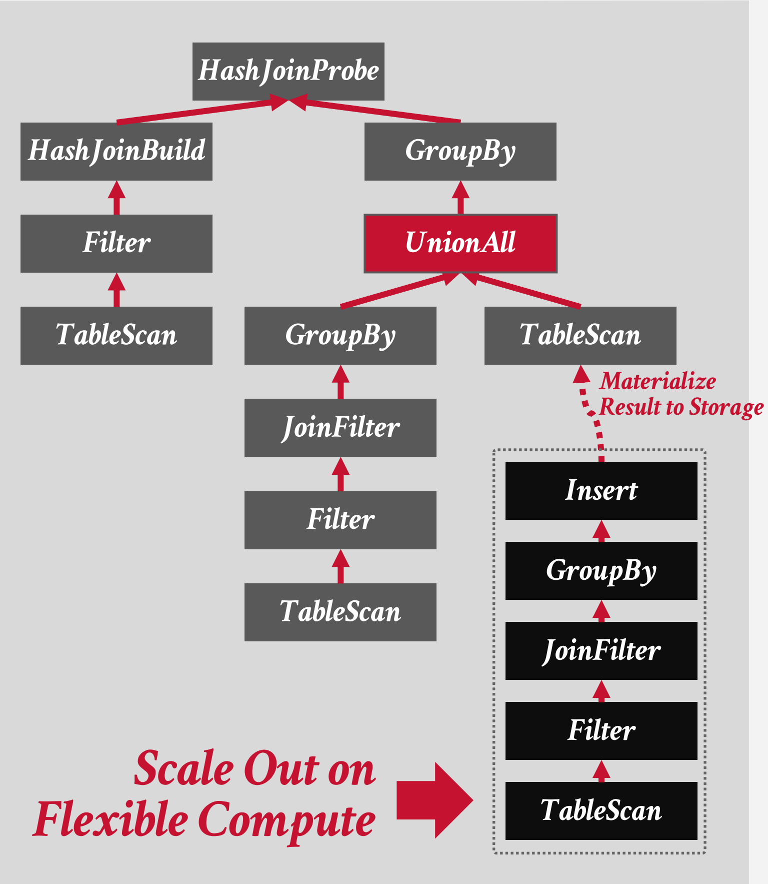


Source: Libo Wang

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

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.

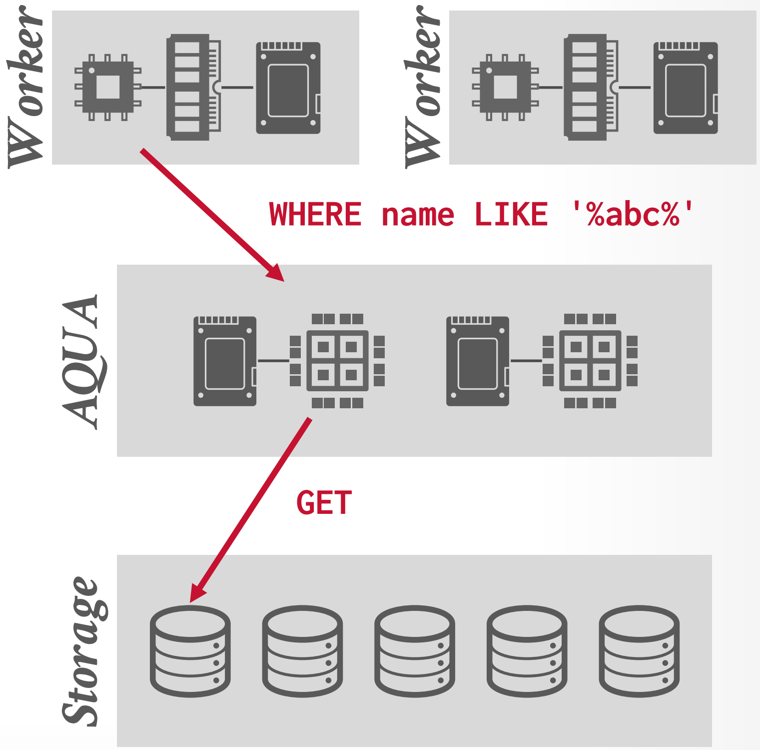
Background workers proactively recompile plans when new version of DBMS is released.

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



Yellowbrick

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

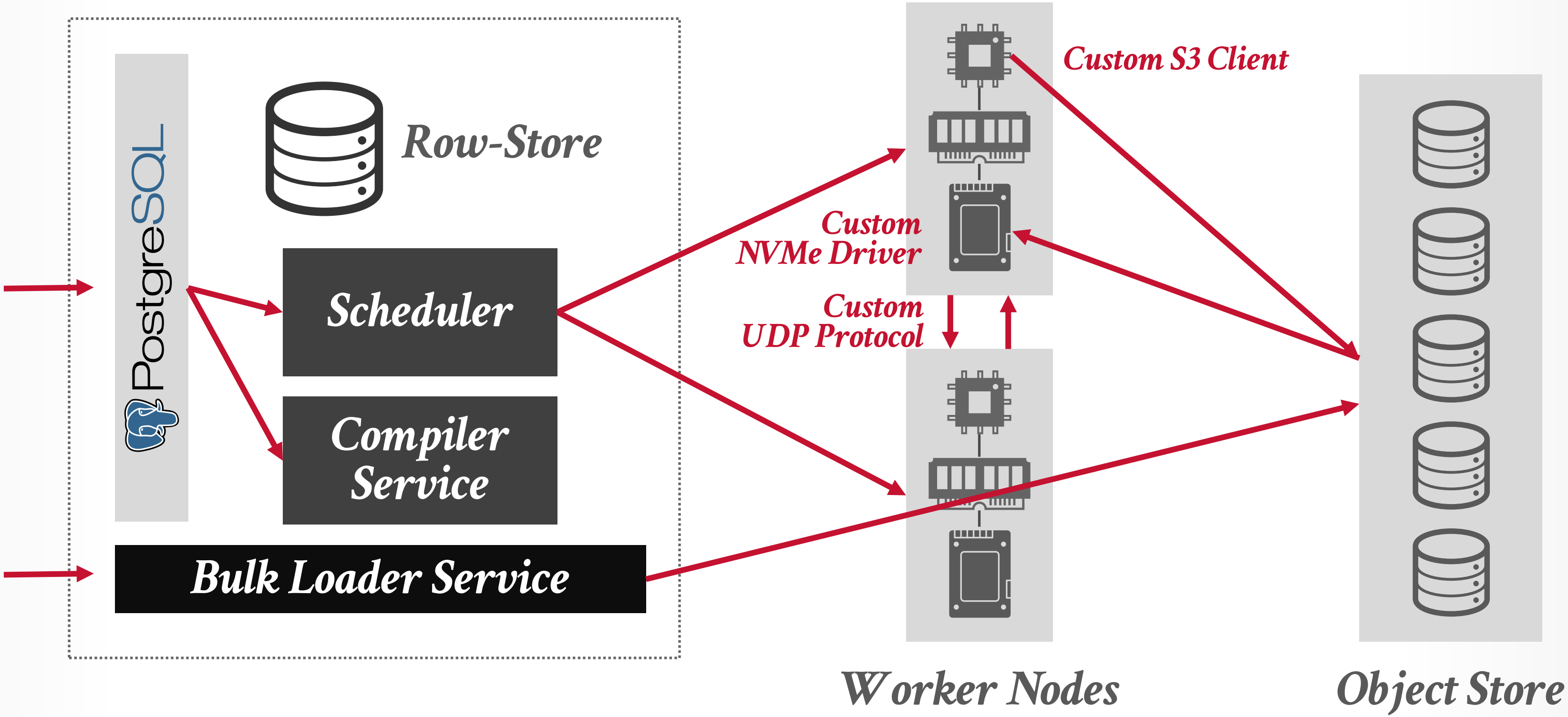
Separate Row + PAX Columnar Storage

Sort-Merge + Hash Joins

PostgreSQL Query Optimizer++

Insane Systems Engineering

YELLOWBRICK: ARCHITECTURE



Source: Mark Cusack

YELLOWBRICK: 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

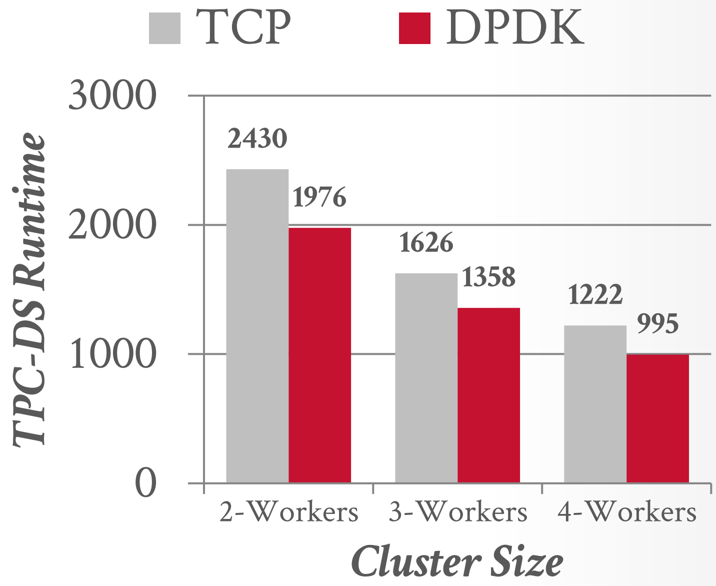
→ Using **`mmap`** with **`mlock`** with huge pages.
→ Allocations are grouped by query to avoid fragmentation.
→ Claims their allocator is 100x faster than libc **`malloc`**.

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 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: 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.

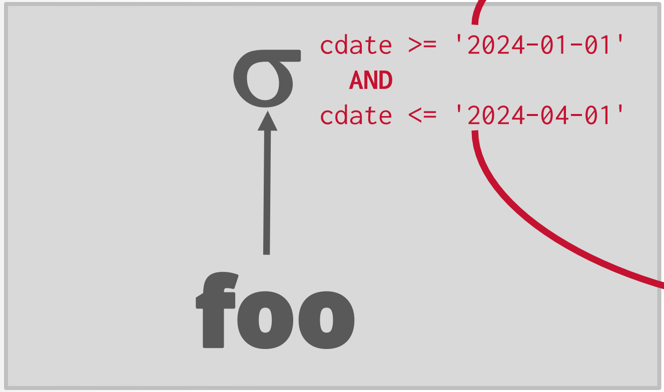
PHOTON: EXPRESSION FUSION


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

databricks

PHOTON: EXPRESSION FUSION

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

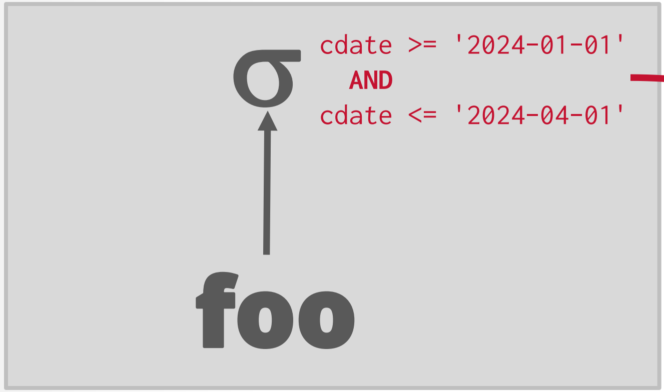



```
vec<offset> sel_geq_date(vec<date> batch, date val) {
  vec<offset> positions;
  for (offset i = 0; i < batch.size(); i++)
    if (batch[i] >= val) positions.append(i);
  return (positions);
}
```

```
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';
```




```
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);
}
```

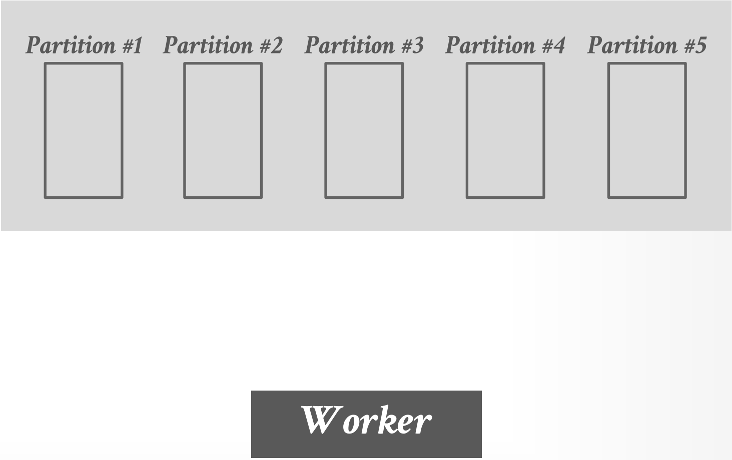
databricks

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

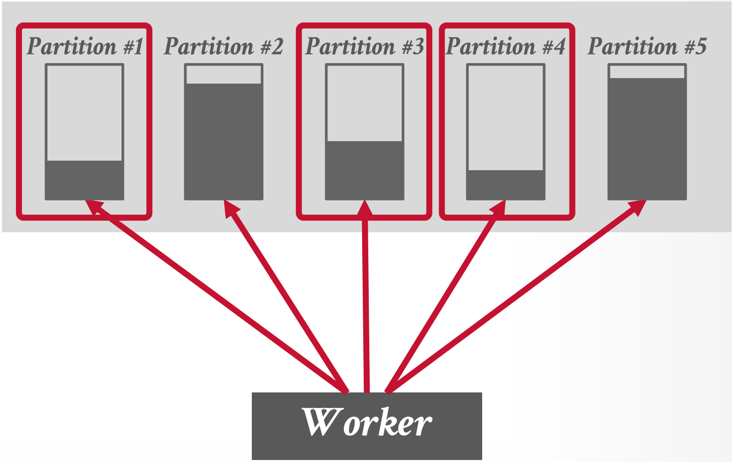
databricks

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.



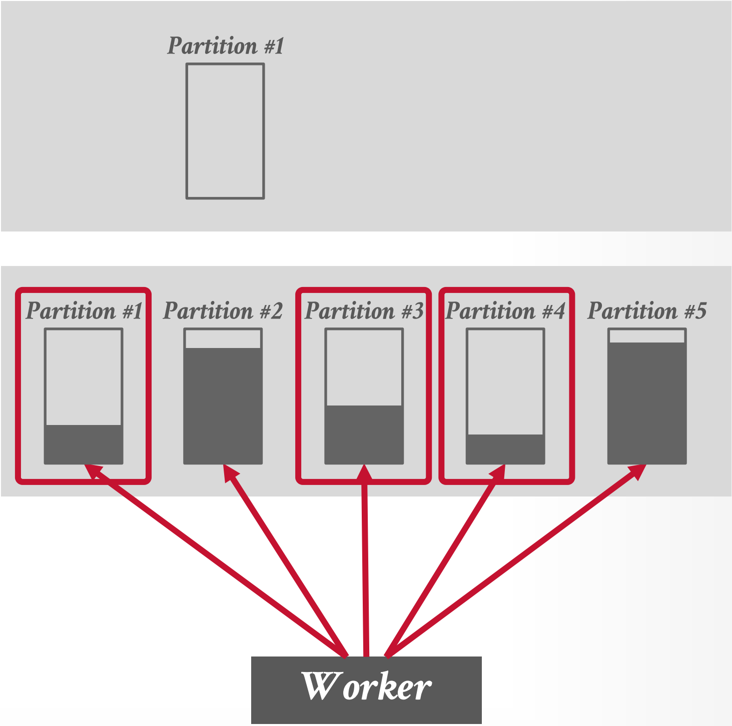
Source: Maryann Xue

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.



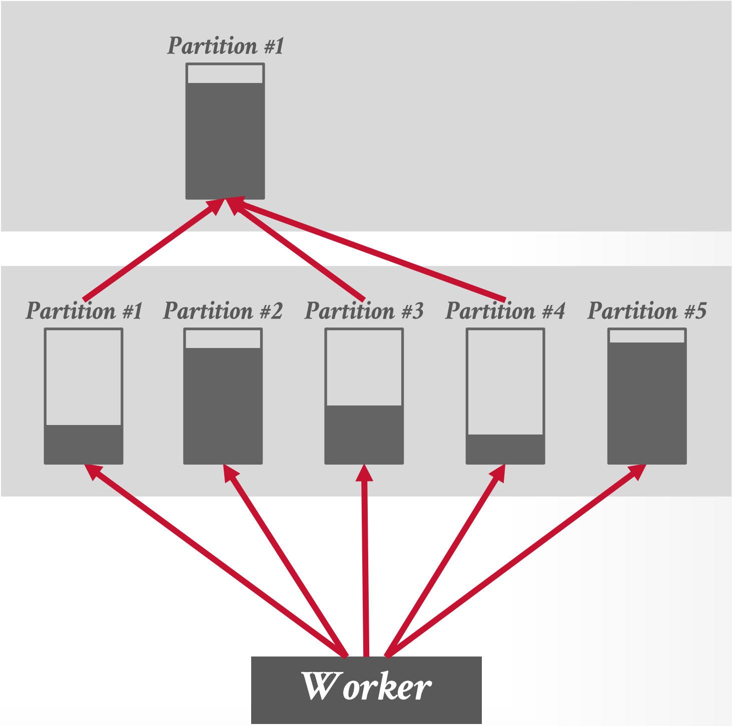
Source: Maryann Xue

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

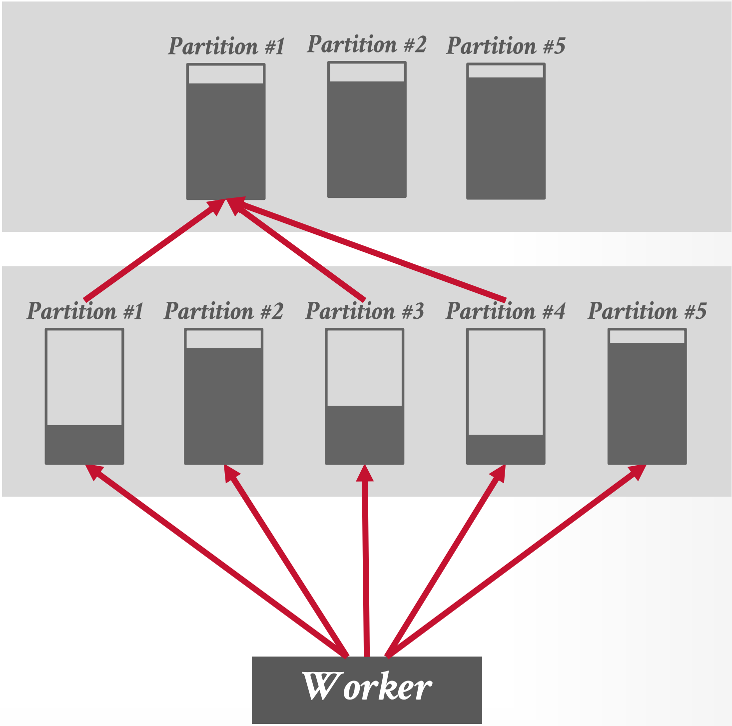
databricks

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 (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.

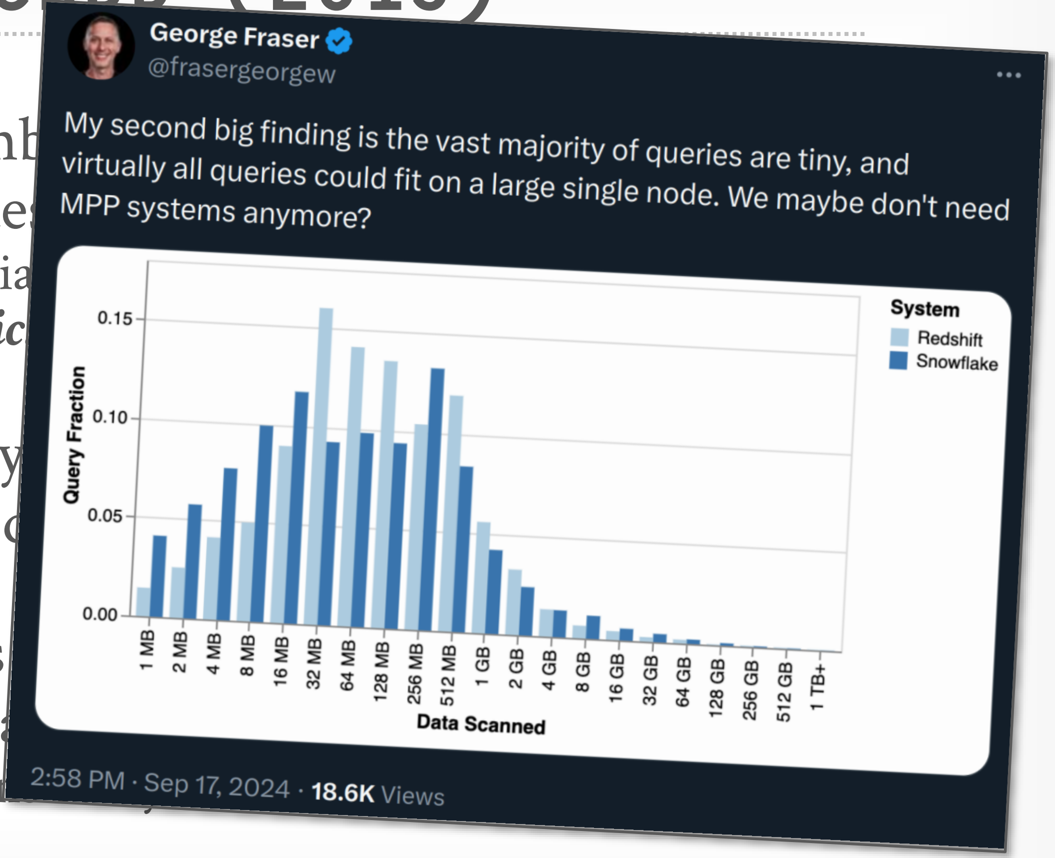
The core DBMS is nearly all custom C++ code with little to no third-party dependencies.
→ Relies on extensions ecosystem to expand capabilities.

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 extensior



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

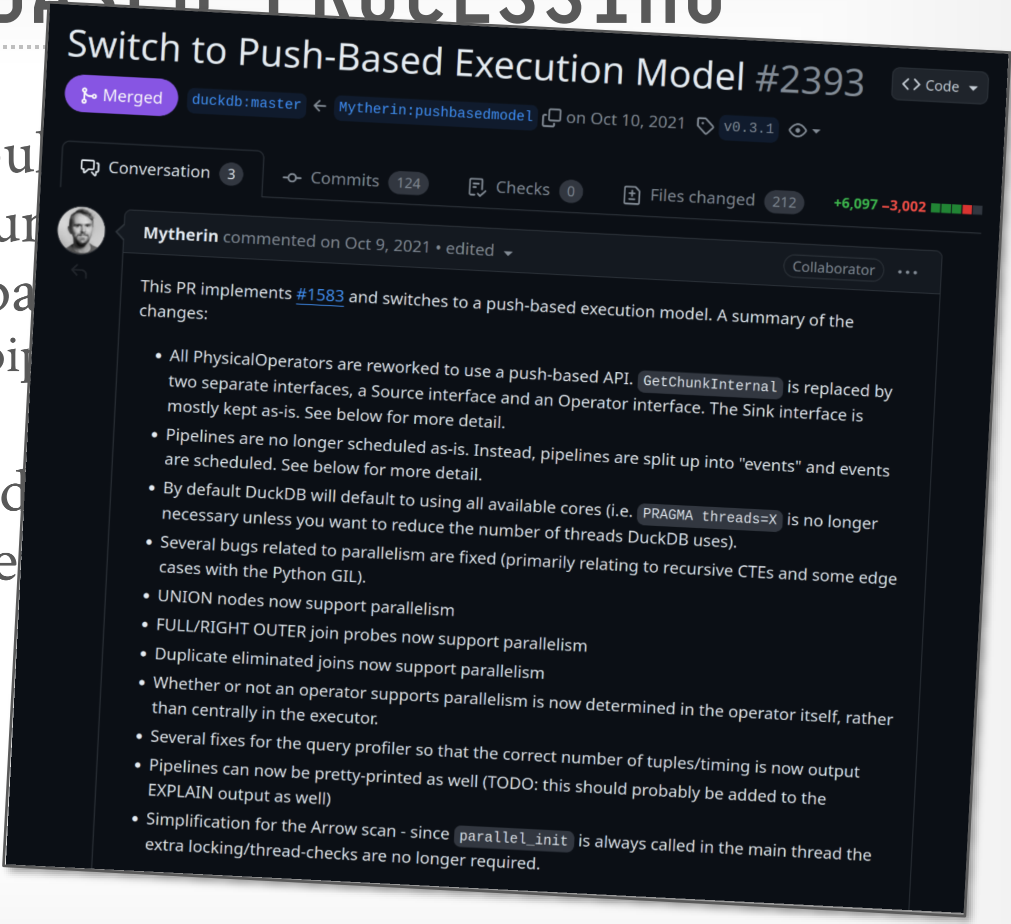
System originally used pull-based vectorized query processing but found it unwieldly to expand to support more complex parallelism.
→ Cannot invoke multiple pipelines simultaneously.

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: PUSH-BASED PROCESSING

System originally used pu
processing but found it u
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

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:

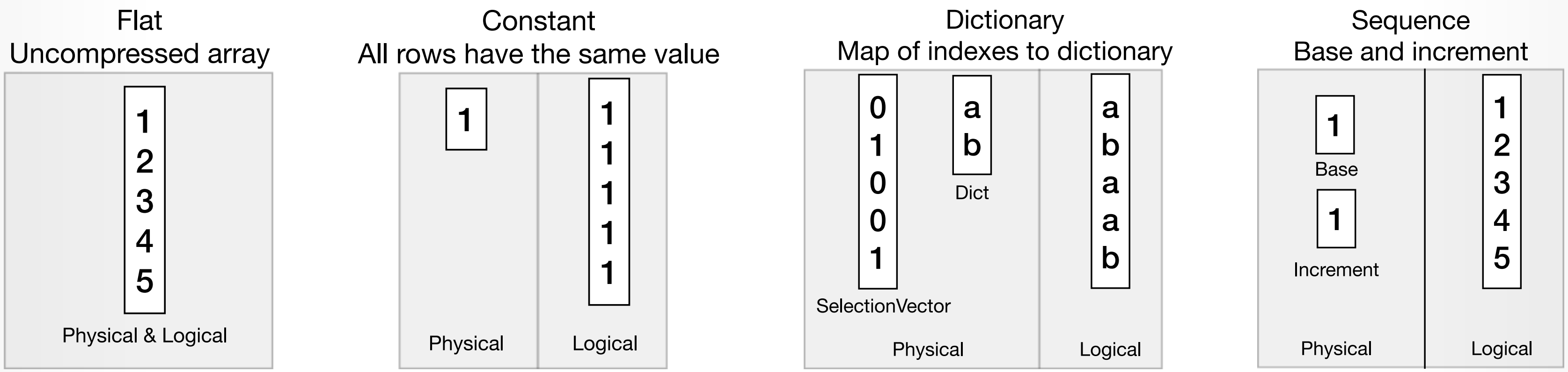
- All PhysicalOperators are reworked to use a push-based API. `GetChunkInternal` is replaced by two separate interfaces, a Source interface and an Operator interface. The Sink interface is mostly kept as-is. See below for more detail.
- Pipelines are no longer scheduled as-is. Instead, pipelines are 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

DUCKDB: VECTORS

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

Supports multiple vector types:

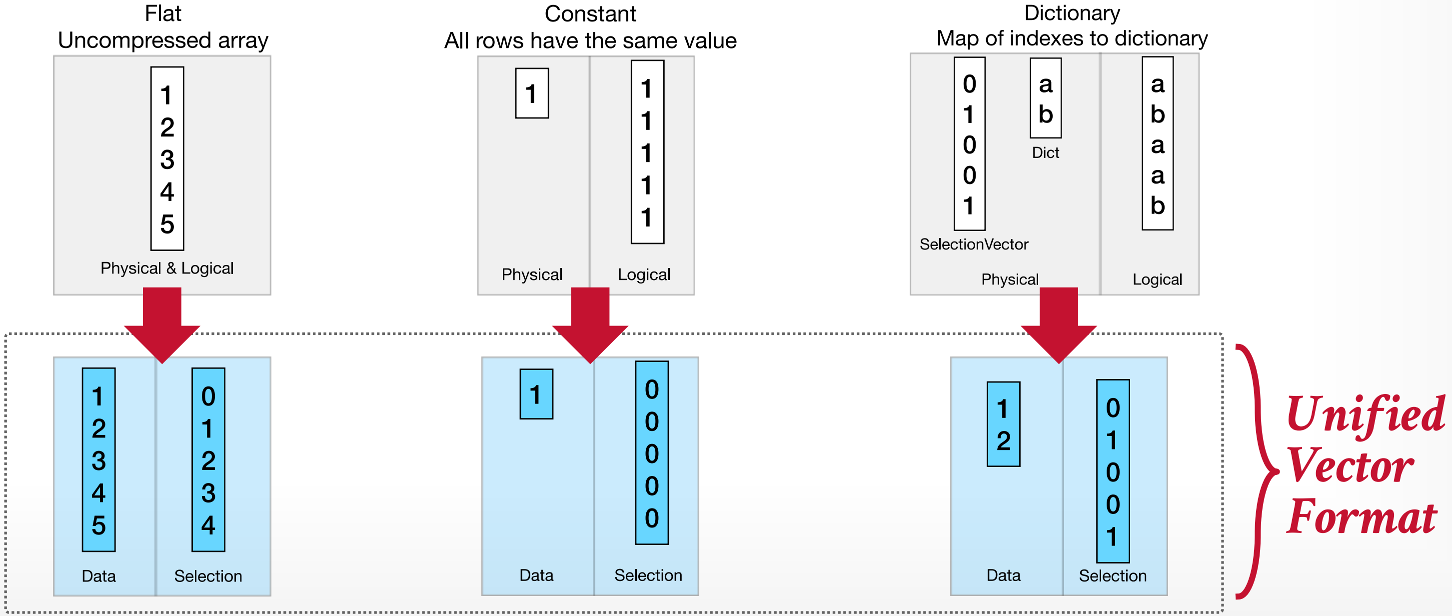


Source: Mark Raasveldt

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

TDB

TABDB (2019)

TabDB is a relational DBMS that stores data in your browser's tab title fields.

It uses Emscripten to convert SQLite's C code into JavaScript.

It then splits the SQLite database file into strings and stores them in your browser tabs.

https://tabdb.io/

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.