Carnegie Mellon University

Database Systems

Distributed OLAP Databases

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

ADMINISTRIVIA

DBMS Potpourri Lecture on Wednesday Dec 4th

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

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

Final Exam is on Friday Dec 13th @ 8:30am
→ Early exam will not be offered.
→ Study guide will be released tomorrow.

UPCOMING DATABASE TALKS

OpenDAL / DataBend (DB Seminar)

→ Monday Dec 2nd @ 4:30pm

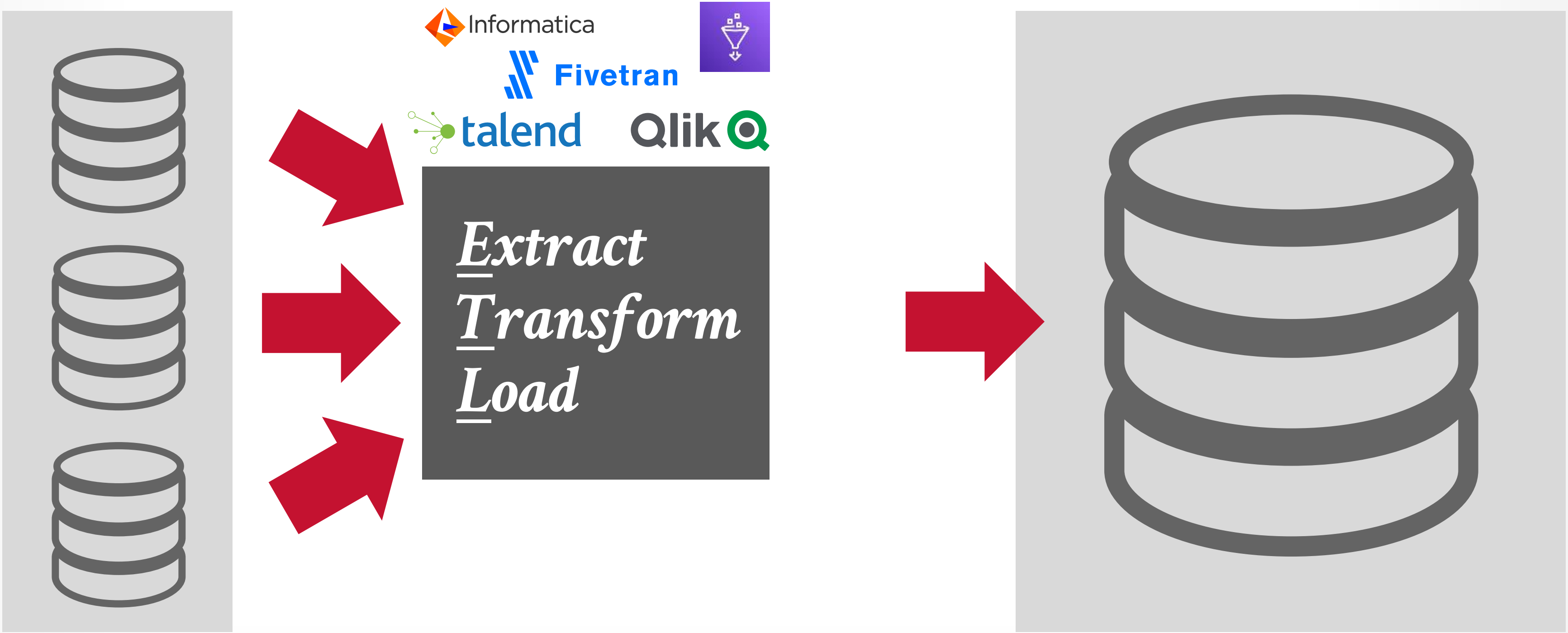
→ Zoom

GreptimeDB (DB Seminar)

→ Monday Dec 9th @ 4:30pm

→ Zoom

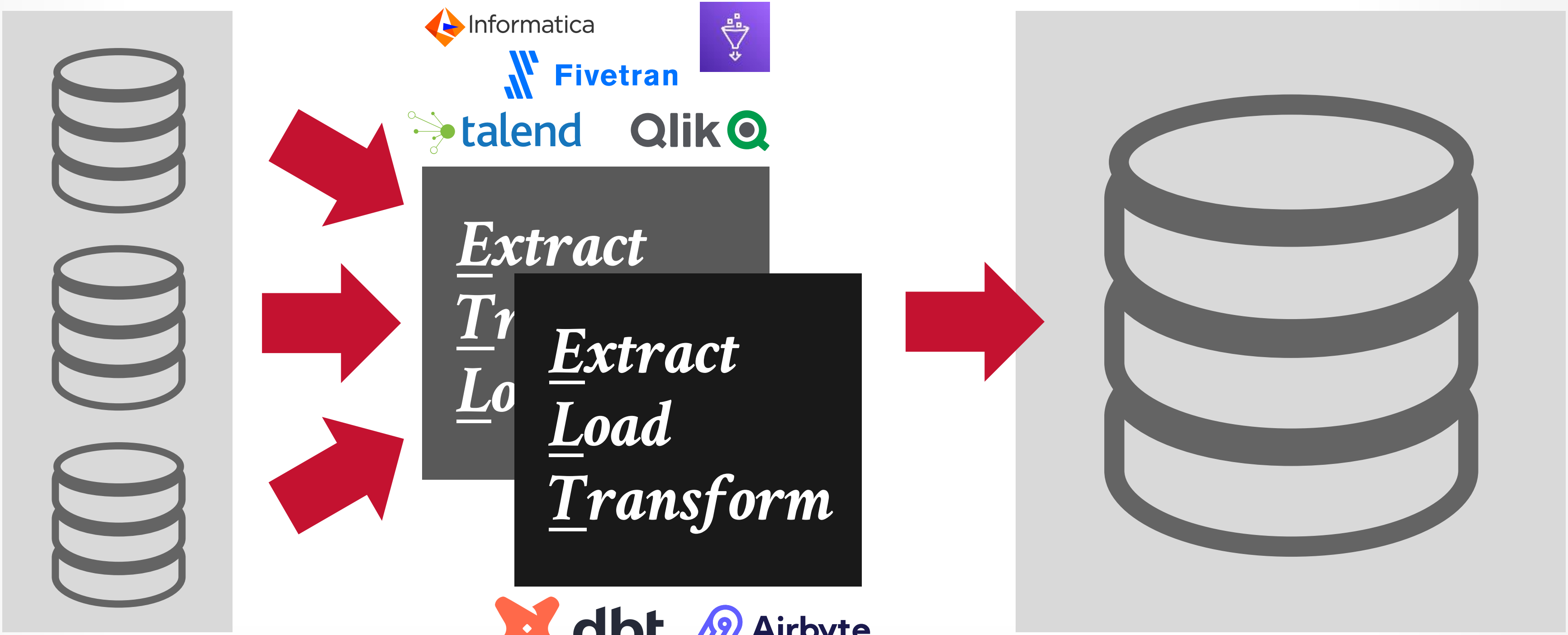
BIFURCATED ENVIRONMENT



OLTP Databases

OLAP Database

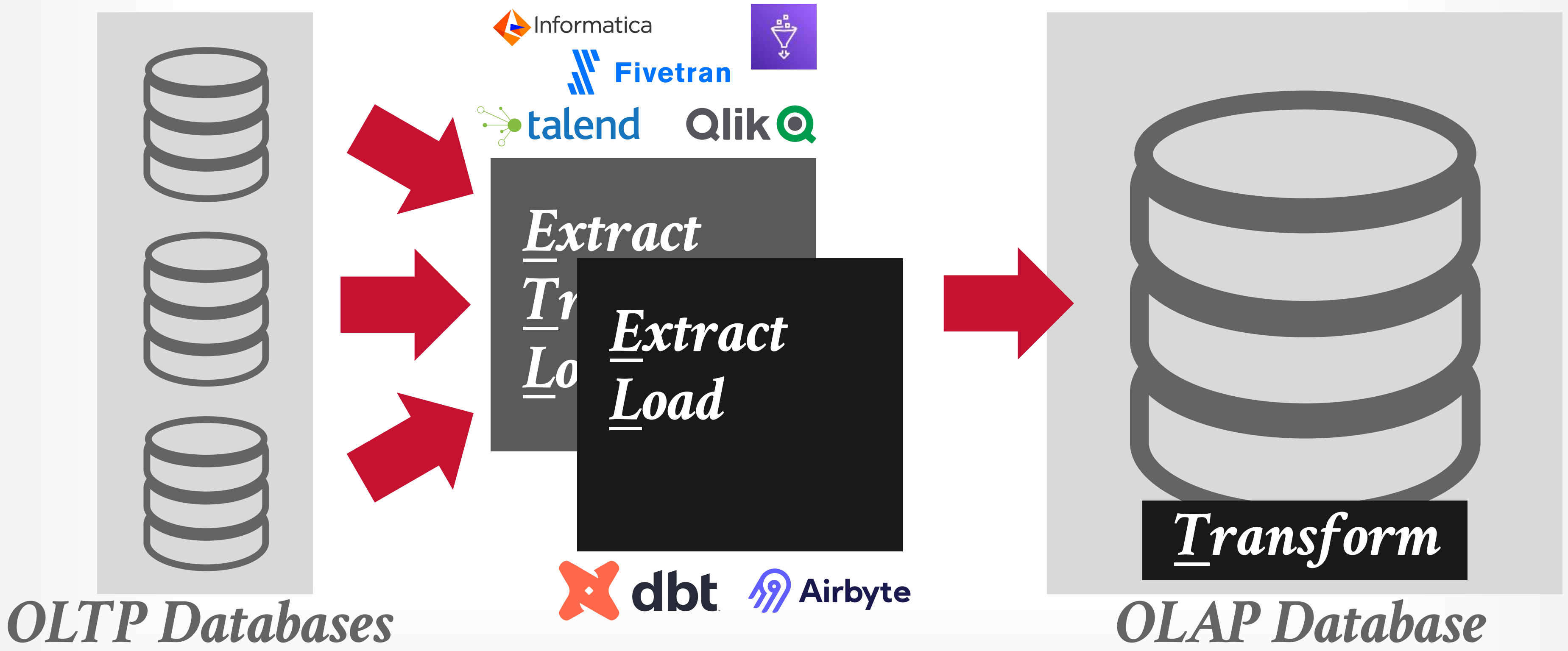
BIFURCATED ENVIRONMENT



OLTP Databases

OLAP Database

BIFURCATED ENVIRONMENT

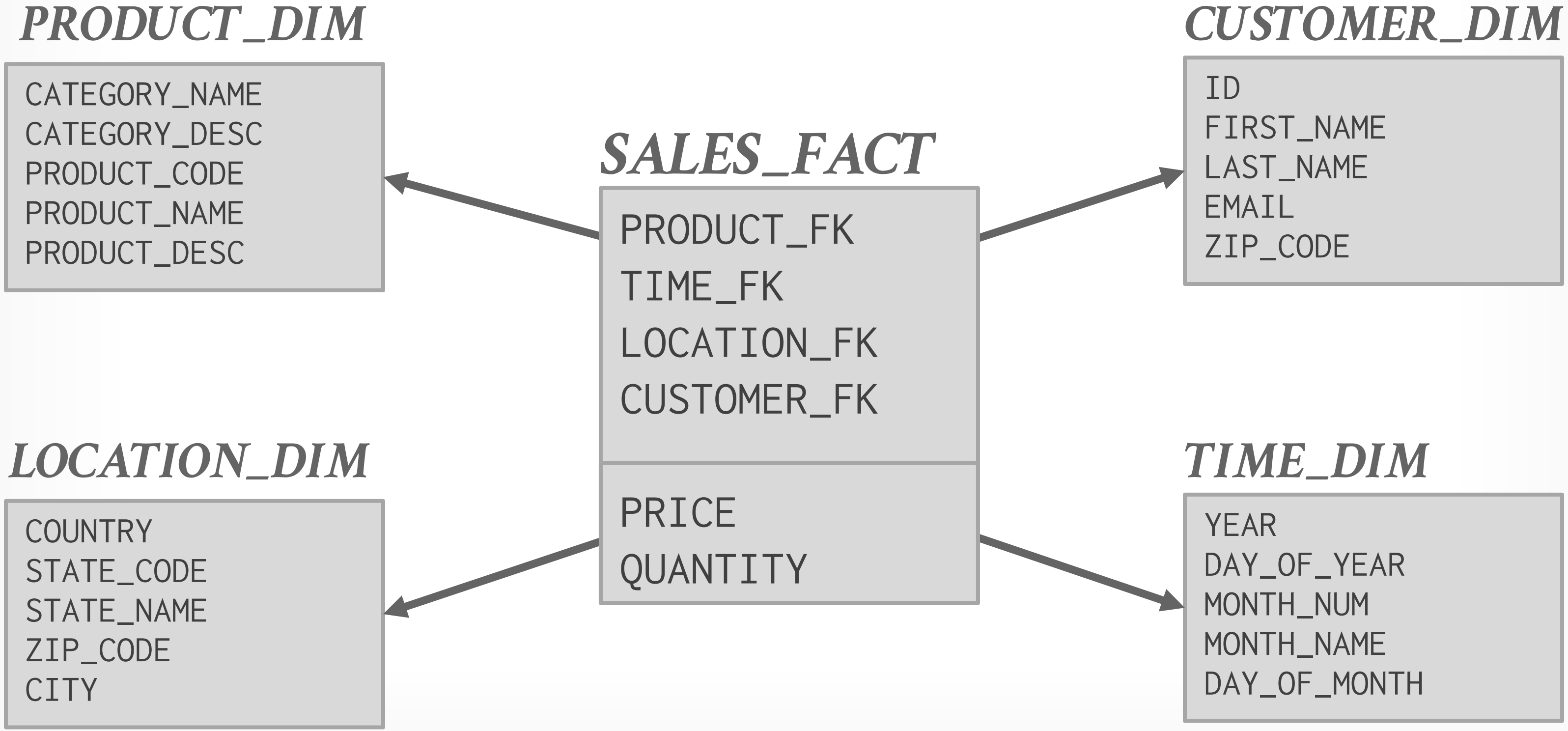


DECISION SUPPORT SYSTEMS

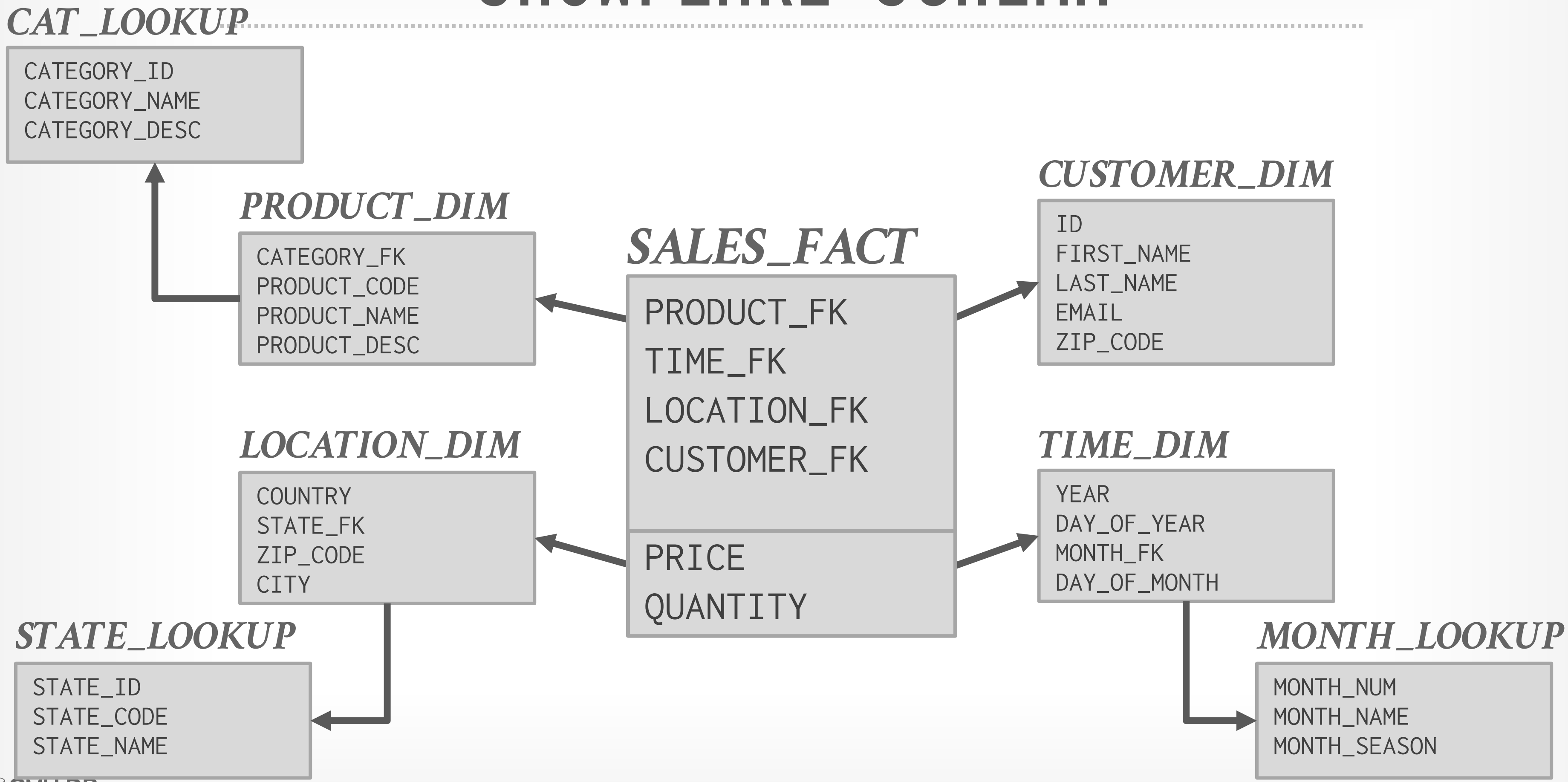
Applications that serve the management, operations, and planning levels of an organization to help people make decisions about future issues and problems by analyzing historical data.

Star Schema vs. Snowflake Schema

STAR SCHEMA


PRODUCT_DIM
CATEGORY_NAME
CATEGORY_DESC
PRODUCT_CODE
PRODUCT_NAME
PRODUCT_DESC
CUSTOMER_DIM
ID
FIRST_NAME
LAST_NAME
EMAIL
ZIP_CODE
SALES_FACT
PRODUCT_FK
TIME_FK
LOCATION_FK
CUSTOMER_FK
PRICE
QUANTITY
LOCATION_DIM
COUNTRY
STATE_CODE
STATE_NAME
ZIP_CODE
CITY
TIME_DIM
YEAR
DAY_OF_YEAR
MONTH_NUM
MONTH_NAME
DAY_OF_MONTH

SNOWFLAKE SCHEMA



STAR VS. SNOWFLAKE SCHEMA

Issue #1: Normalization

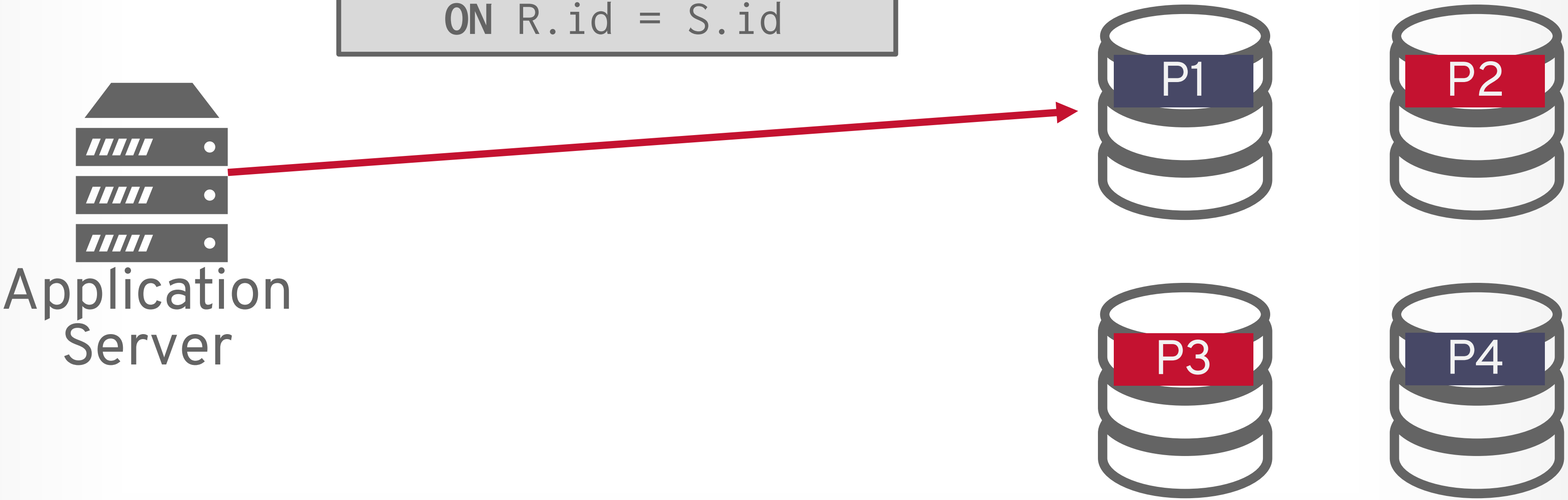
→ Snowflake schemas take up less storage space.
→ Denormalized data models may incur integrity and consistency violations.

Issue #2: Query Complexity

→ Snowflake schemas require more joins to get the data needed for a query.
→ Queries on star schemas will (usually) be faster.

PROBLEM SETUP

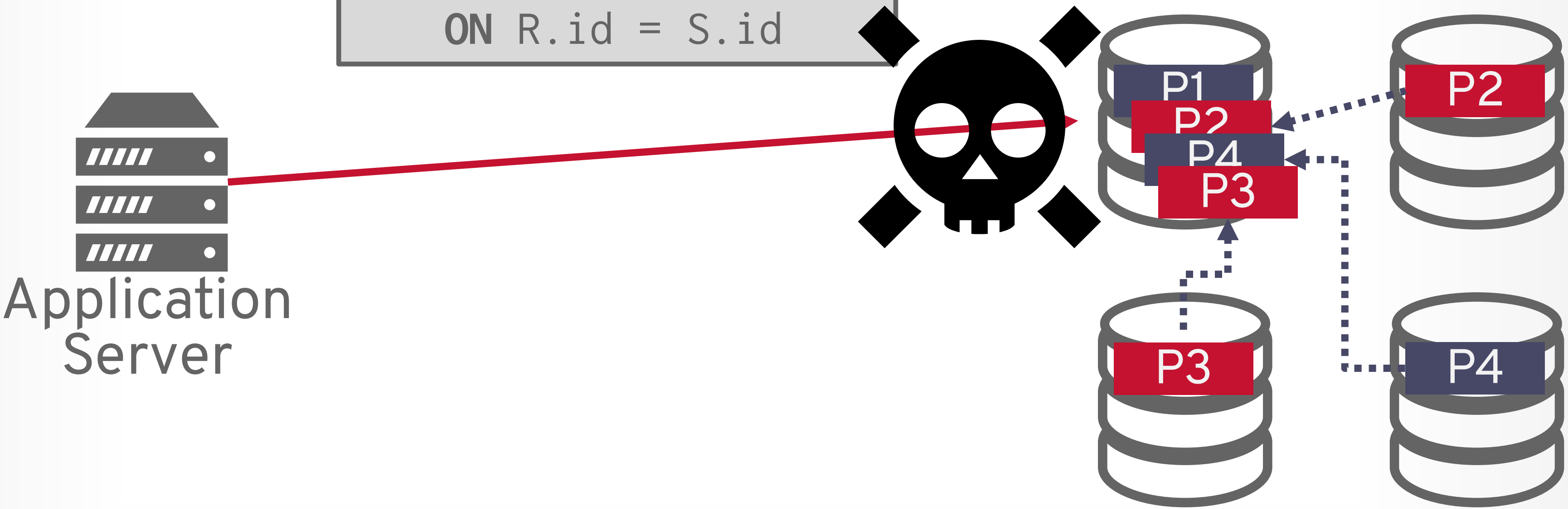
```
SELECT * FROM R JOIN S
ON R.id = S.id
```



PROBLEM SETUP

避免将数据全部传输到
一个节点进行 JOIN 运算

```
SELECT * FROM R JOIN S
ON R.id = S.id
```



TODAY'S AGENDA

Execution Models

Query Planning

Distributed Join Algorithms

Cloud Systems

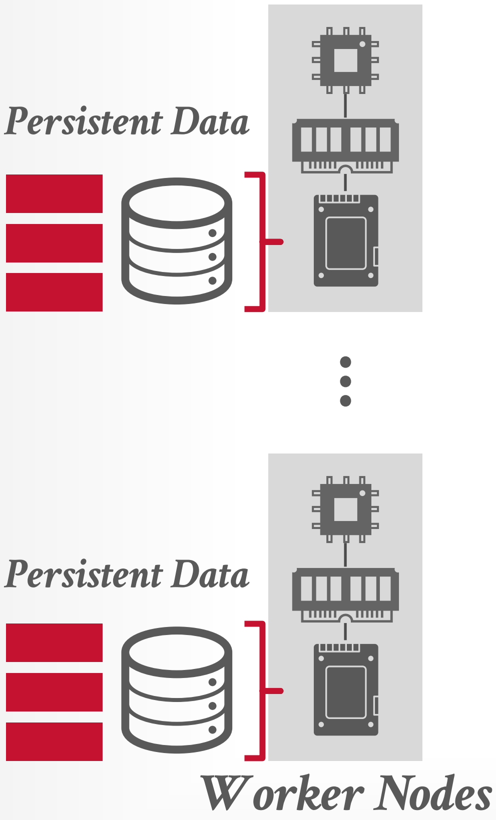
DISTRIBUTED QUERY EXECUTION

Executing an OLAP query in a distributed DBMS is roughly the same as on a single-node DBMS.
→ Query plan is a DAG of physical operators.

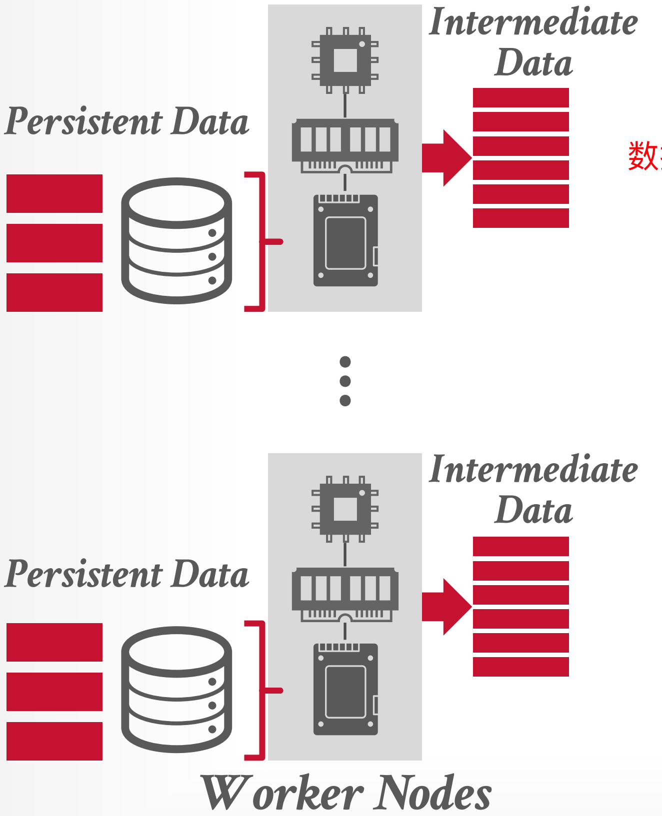
For each operator, the DBMS considers where input is coming from and where to send output.
→ Table Scans
→ Joins
→ Aggregations
→ Sorting

根据 pipeline 分而治之

DISTRIBUTED QUERY EXECUTION

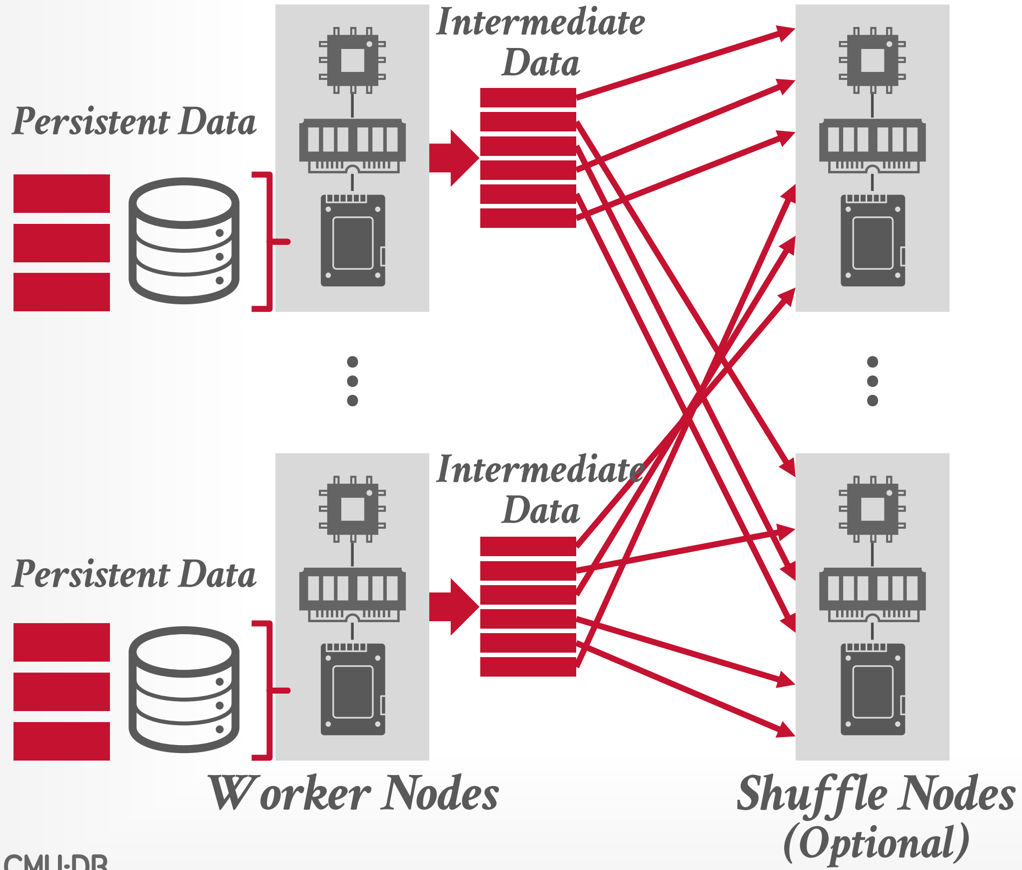


DISTRIBUTED QUERY EXECUTION

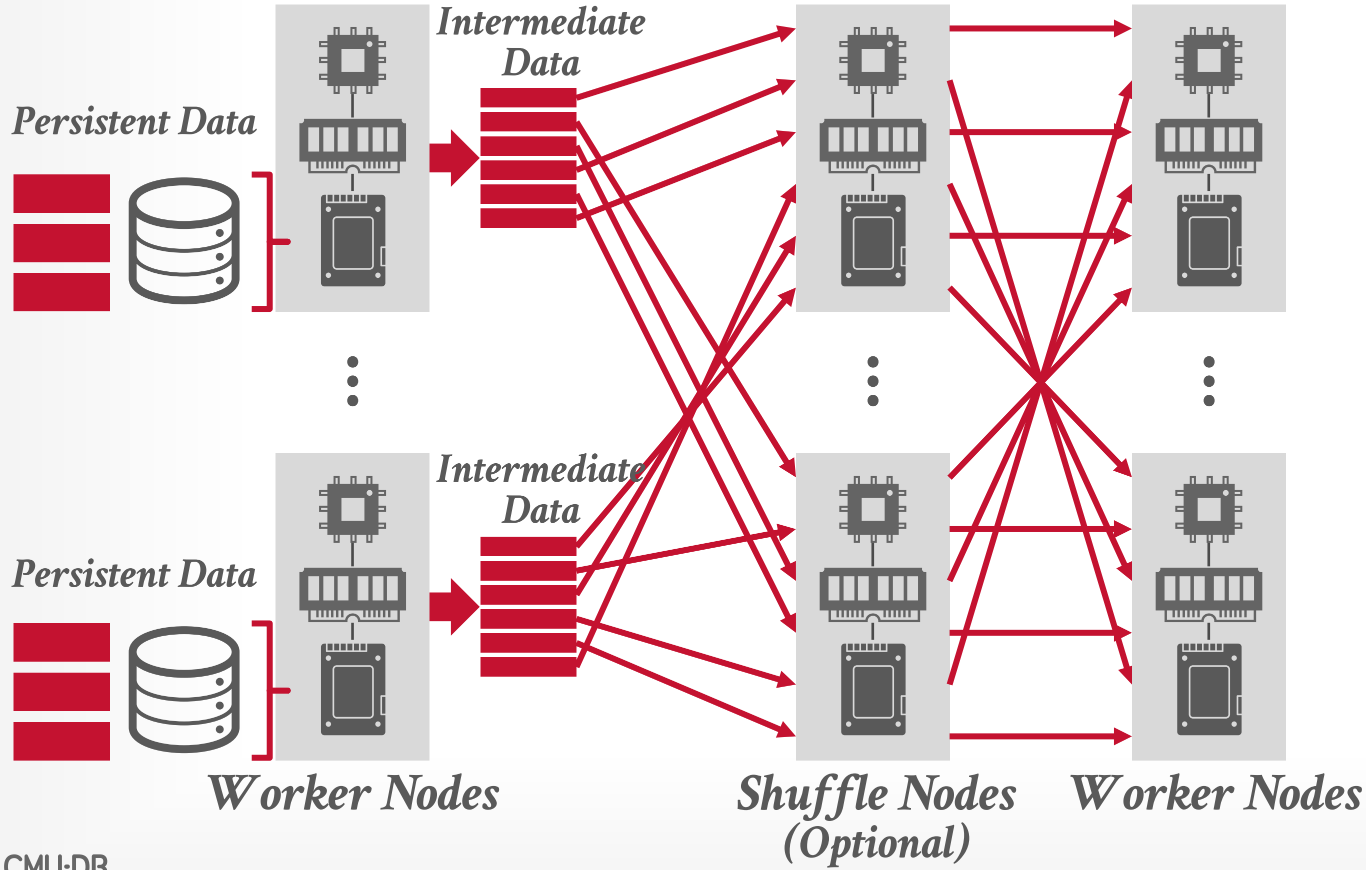


数据在不同算子间传递.

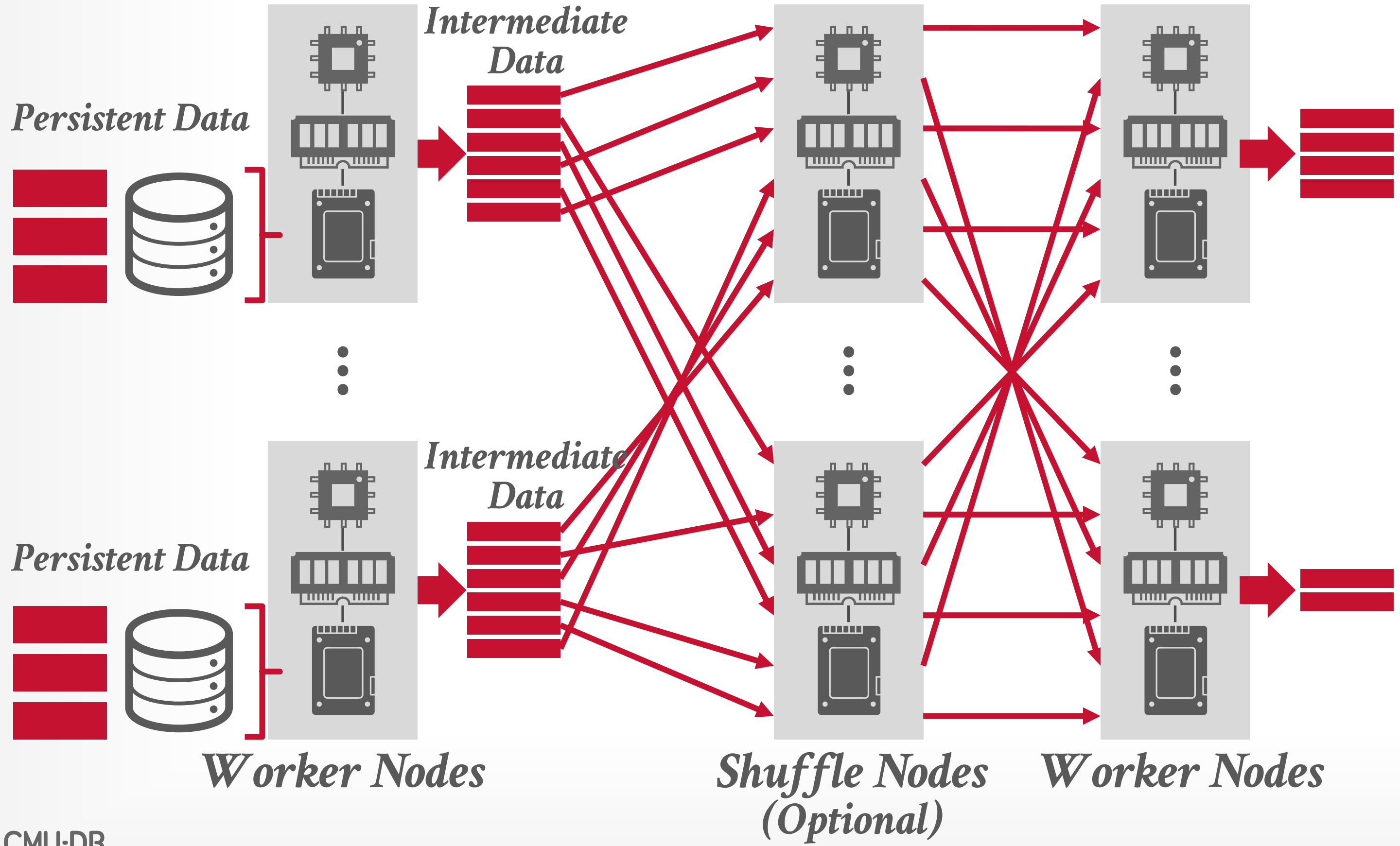
DISTRIBUTED QUERY EXECUTION



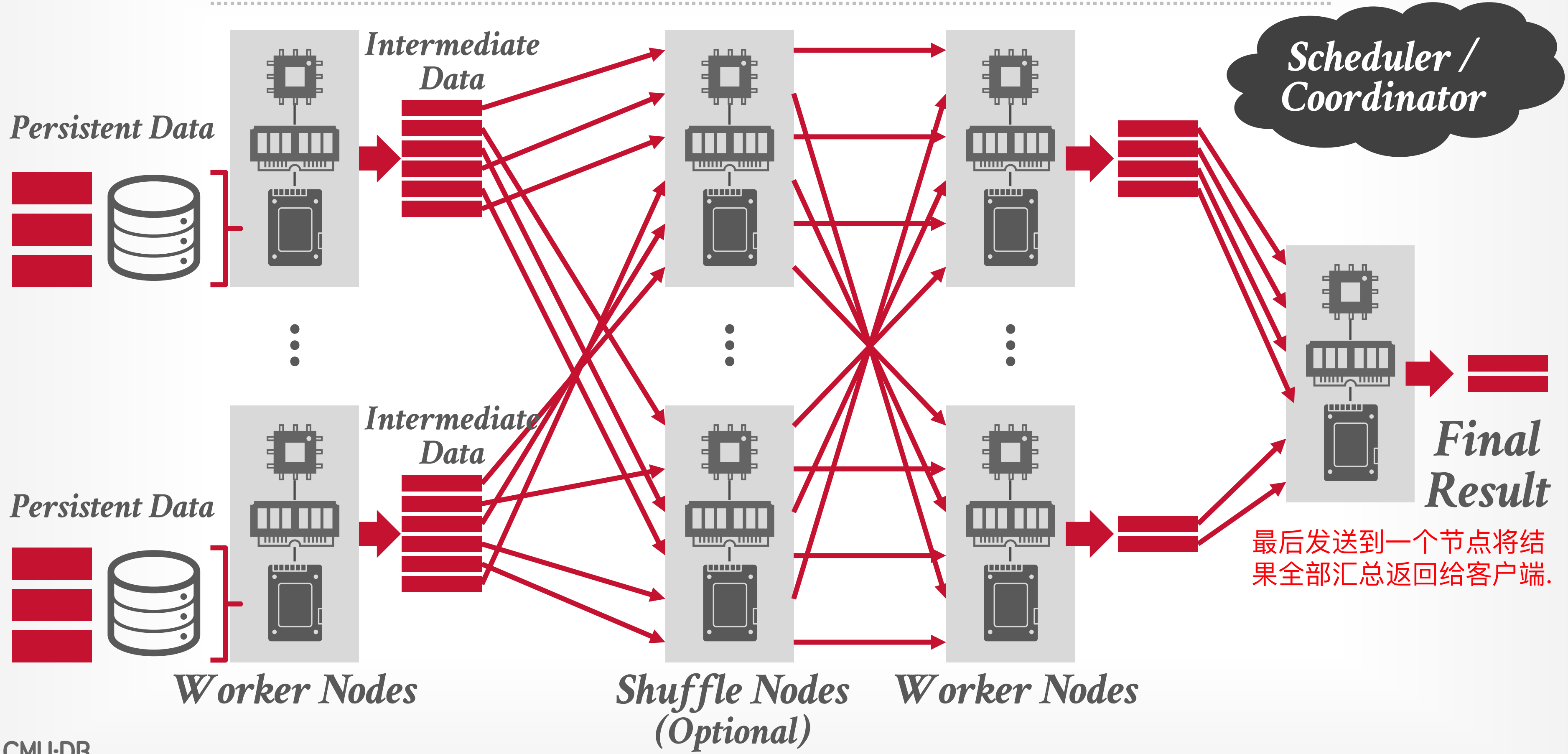
DISTRIBUTED QUERY EXECUTION



DISTRIBUTED QUERY EXECUTION



DISTRIBUTED QUERY EXECUTION



DATA CATEGORIES

Persistent Data:

→ The "source of record" for the database (e.g., tables).

→ Modern systems assume that these data files are immutable but can support updates by rewriting them.

Intermediate Data:

→ Short-lived artifacts produced by query operators during execution and then consumed by other operators.

→ The amount of intermediate data that a query generates has little to no correlation to amount of persistent data that it reads or the execution time.

DISTRIBUTED SYSTEM ARCHITECTURE

A distributed DBMS's system architecture specifies the location of the database's data files. This affects how nodes coordinate with each other and where they retrieve/store objects in the database.

Two approaches (not mutually exclusive):
→ **Push Query to Data**
→ **Pull Data to Query**

PUSH VS. PULL

Approach #1: Push Query to Data

→ Send the query (or a portion of it) to the node that contains the data.

→ Perform as much filtering and processing as possible where data resides before transmitting over network.

Approach #2: Pull Data to Query

→ Bring the data to the node that is executing a query that needs it for processing.

→ This is necessary when there is no compute resources available where database files are located.

PUSH VS. PULL

Approa
→ Send th
contai
→ Perfor
data re

Approa
→ Bring
needs it for processing.
→ This is necessary when there is no compute resources available where database files are located.

Filtering and retrieving data using Amazon S3 Select

amazon

PDF | RSS

With Amazon S3 Select, you can use simple structured query language (SQL) statements to filter the contents of an Amazon S3 object and retrieve just the subset of data that you need. By using Amazon S3 Select to filter this data, you can reduce the amount of data that Amazon S3 transfers, which reduces the cost and latency to retrieve this data.

Amazon S3 Select works on objects stored in CSV, JSON, or Apache Parquet format. It also works with objects that are compressed with GZIP or BZIP2 (for CSV and JSON objects only), and server-side encrypted objects. You can specify the format of the results as either CSV or JSON, and you can determine how the records in the result are delimited.

You pass SQL expressions to Amazon S3 in the request. Amazon S3 Select supports a subset of SQL. For more information about the SQL elements that are supported by Amazon S3 Select, see SQL reference for Amazon S3 Select.

You can perform SQL queries using AWS SDKs, the SELECT Object Content REST API, the AWS Command Line Interface (AWS CLI), or the Amazon S3 console. The Amazon S3 console limits the amount of data returned to 40 MB. To retrieve more data, use the AWS CLI or the API.

PUSH vs. PULL

Approac…

…mpute resources
…ed.

Filtering and retrieving data using Amazon S3 Select

amazon

PDF | RSS

With Amazon S3 Select, yo… …uery language (SQL) statements to filter the contents of an …at you need. By using Amazon S3 Select to filter this data, you can …ich reduces the cost and latency to retrieve this data.

…or Apache Parquet format. It also works with objects that are … only), and server-side encrypted objects. You can specify the …etermine how the records in the result are delimited.

…azon S3 Select supports a subset of SQL. For more information … Select, see SQL reference for Amazon S3 Select.

…Object Content REST API, the AWS Command Line Interface …le limits the amount of data returned to 40 MB. To retrieve

Query Blob Contents

Microsoft

Feedback

Article • 07/20/2021 • 10 minutes to read • 3 contributors

The `Query Blob Contents` API applies a simple Structured Query Language (SQL) statement on a blob's contents and returns only the queried subset of the data. You can also call `Query Blob Contents` to query the contents of a version or snapshot.

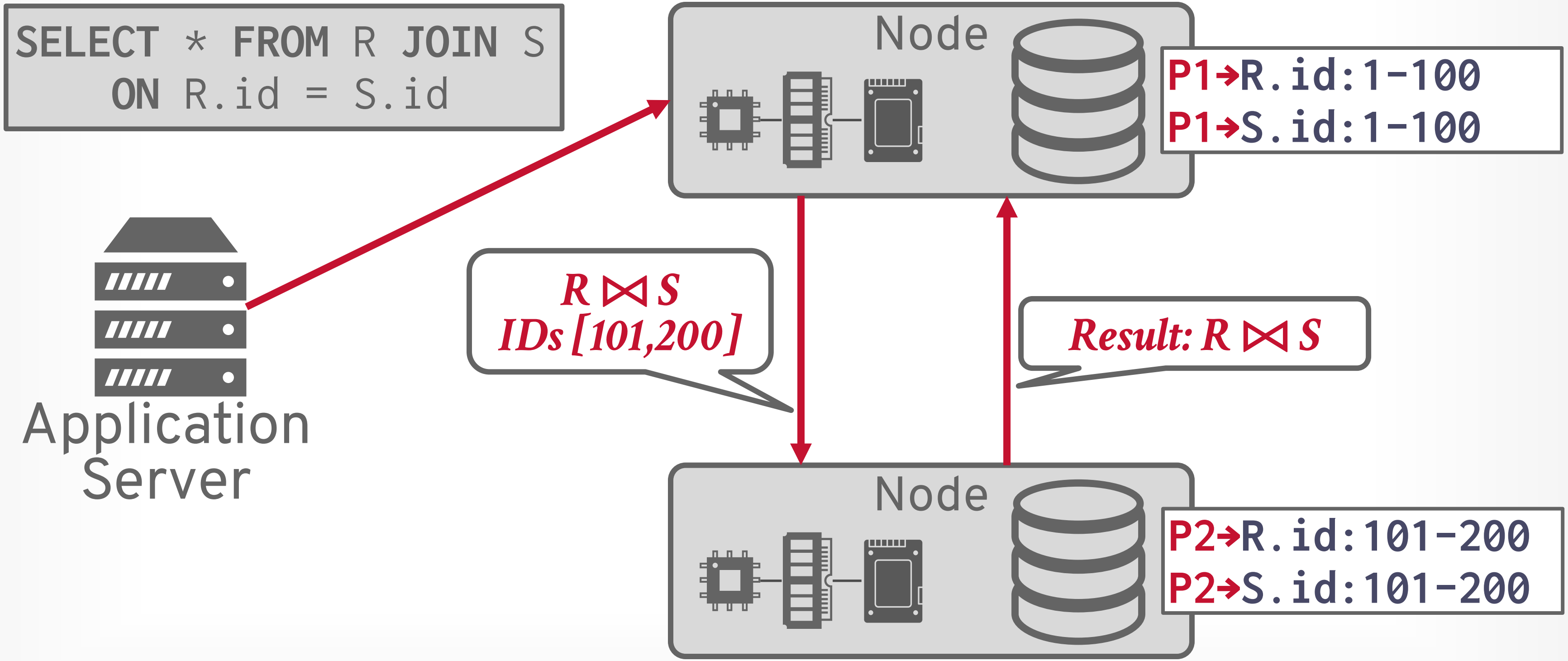
Request

The `Query Blob Contents` request may be constructed as follows. HTTPS is recommended. Replace *myaccount* with the name of your storage account:

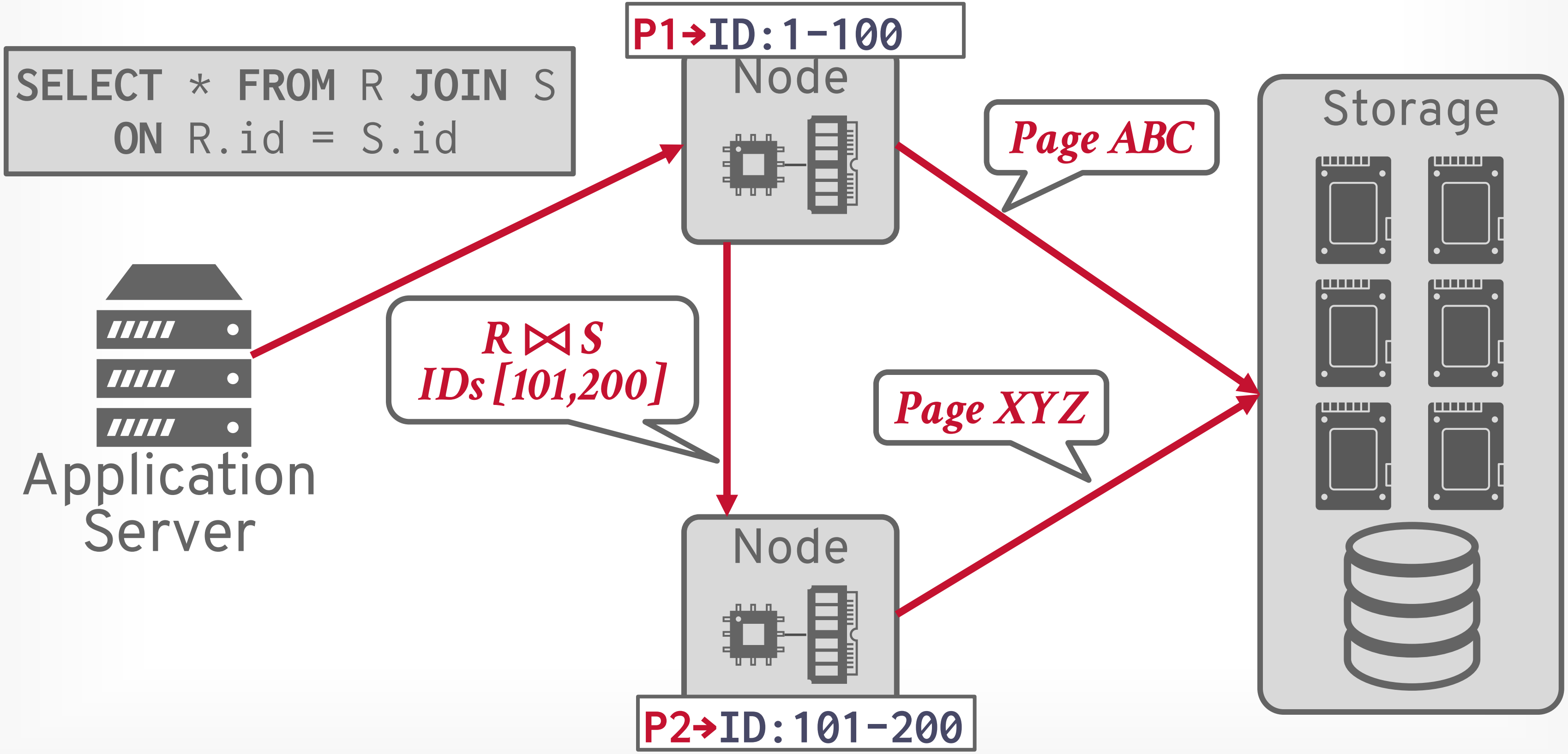
POST Method Request URI	HTTP Version
`https://myaccount.blob.core.windows.net/mycontainer/myblob?comp=query`	HTTP/1.0
`https://myaccount.blob.core.windows.net/mycontainer/myblob?comp=query&snapshot=<DateTime>`	HTTP/1.1
`https://myaccount.blob.core.windows.net/mycontainer/myblob?comp=query&versionid=<DateTime>`	

PUSH QUERY TO DATA

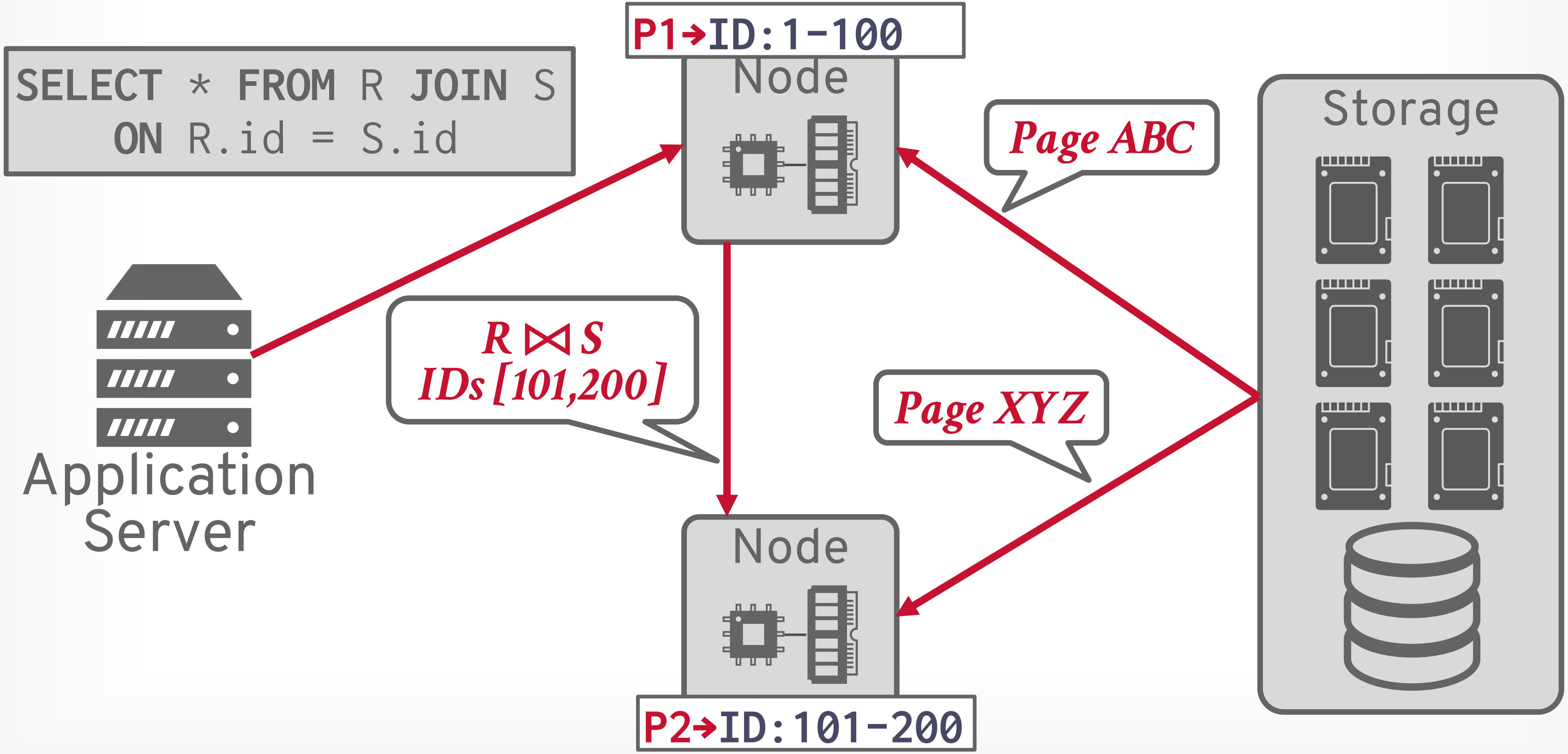
```
SELECT * FROM R JOIN S
ON R.id = S.id
```



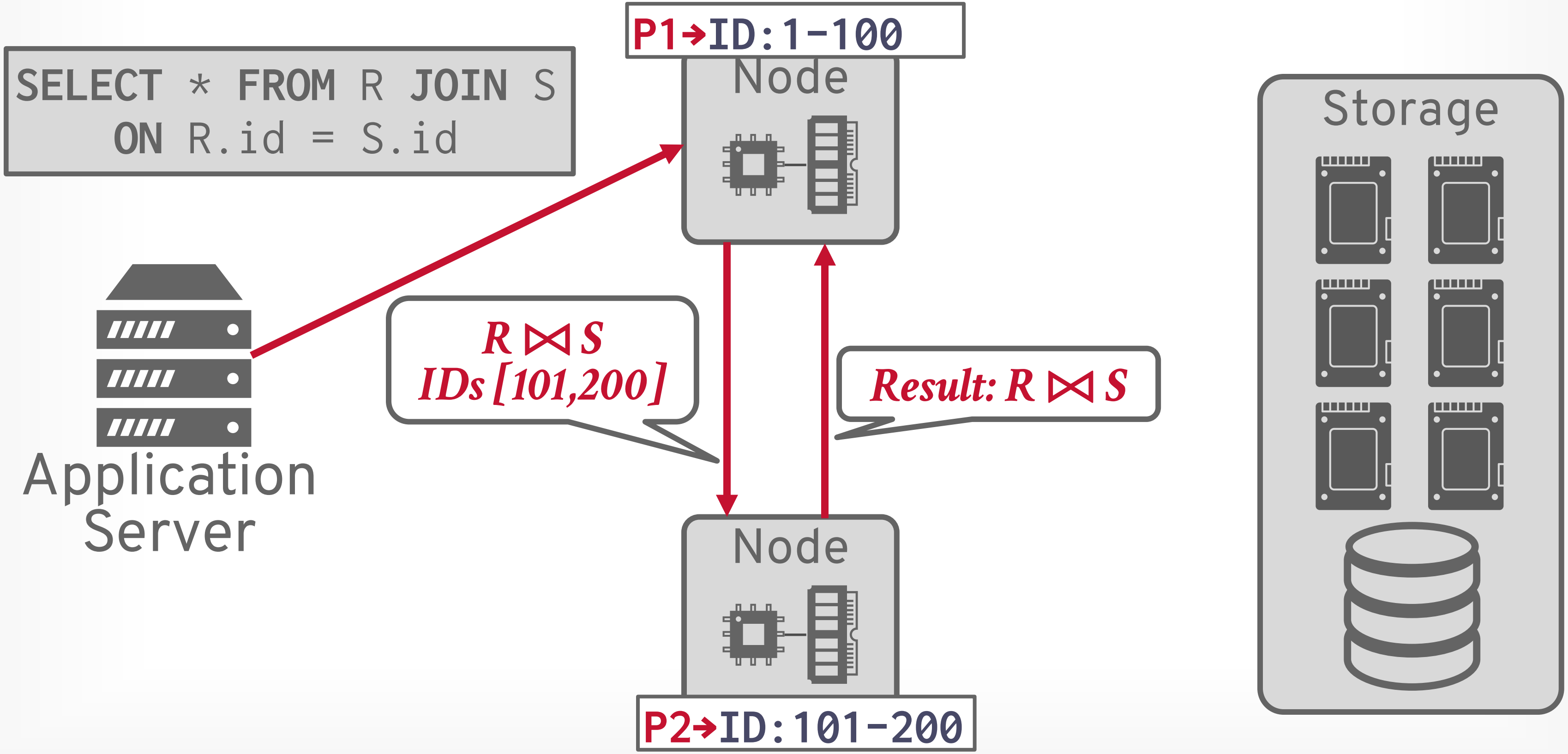
PULL DATA TO QUERY



PULL DATA TO QUERY



PULL DATA TO QUERY



OBSERVATION

The data that a node receives from remote sources are cached in the buffer pool.

→ This allows the DBMS to support intermediate results that are large than the amount of memory available.

→ Ephemeral pages are not persisted after a restart.

What happens to a long-running OLAP query if a node crashes during execution?

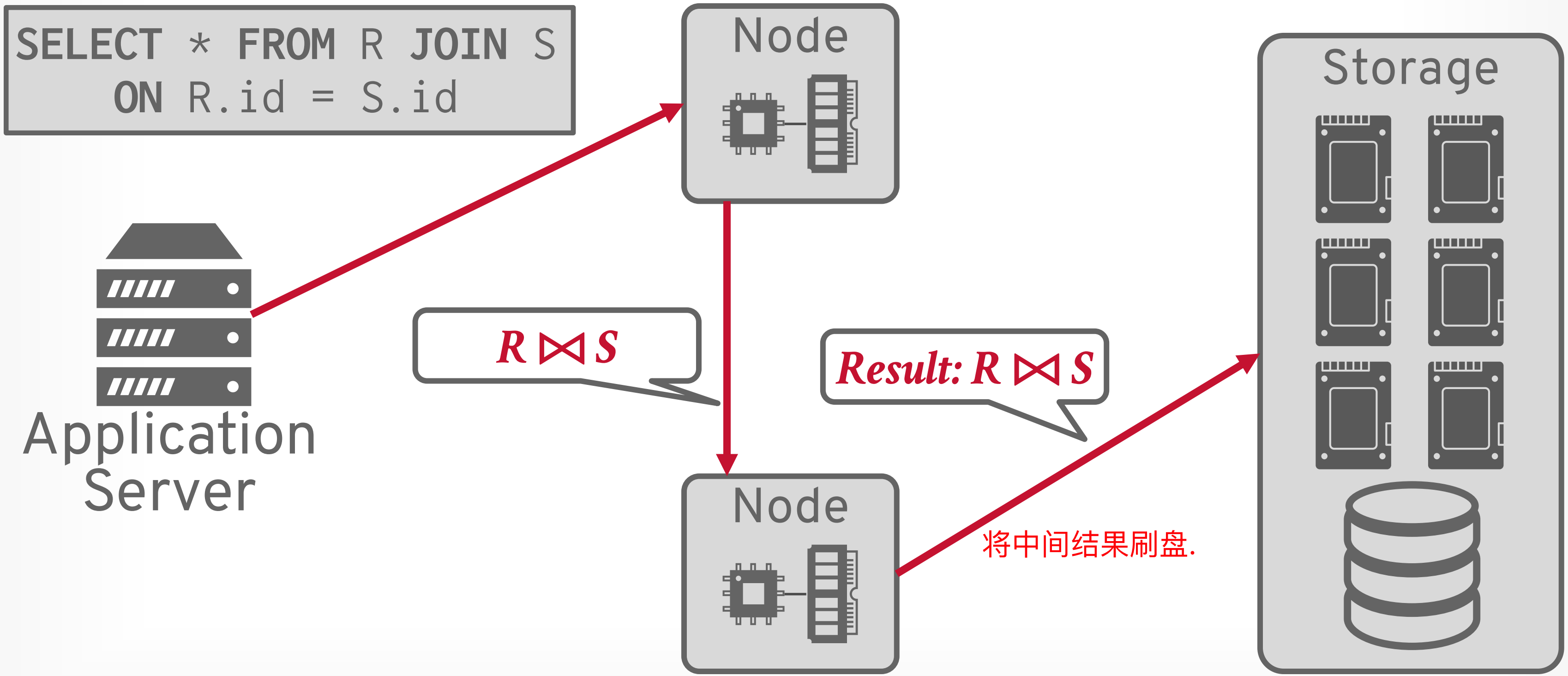
QUERY FAULT TOLERANCE

Most shared-nothing distributed OLAP DBMSs are designed to assume that nodes do not fail during query execution.

→ If one node fails during query execution, then the whole query fails.

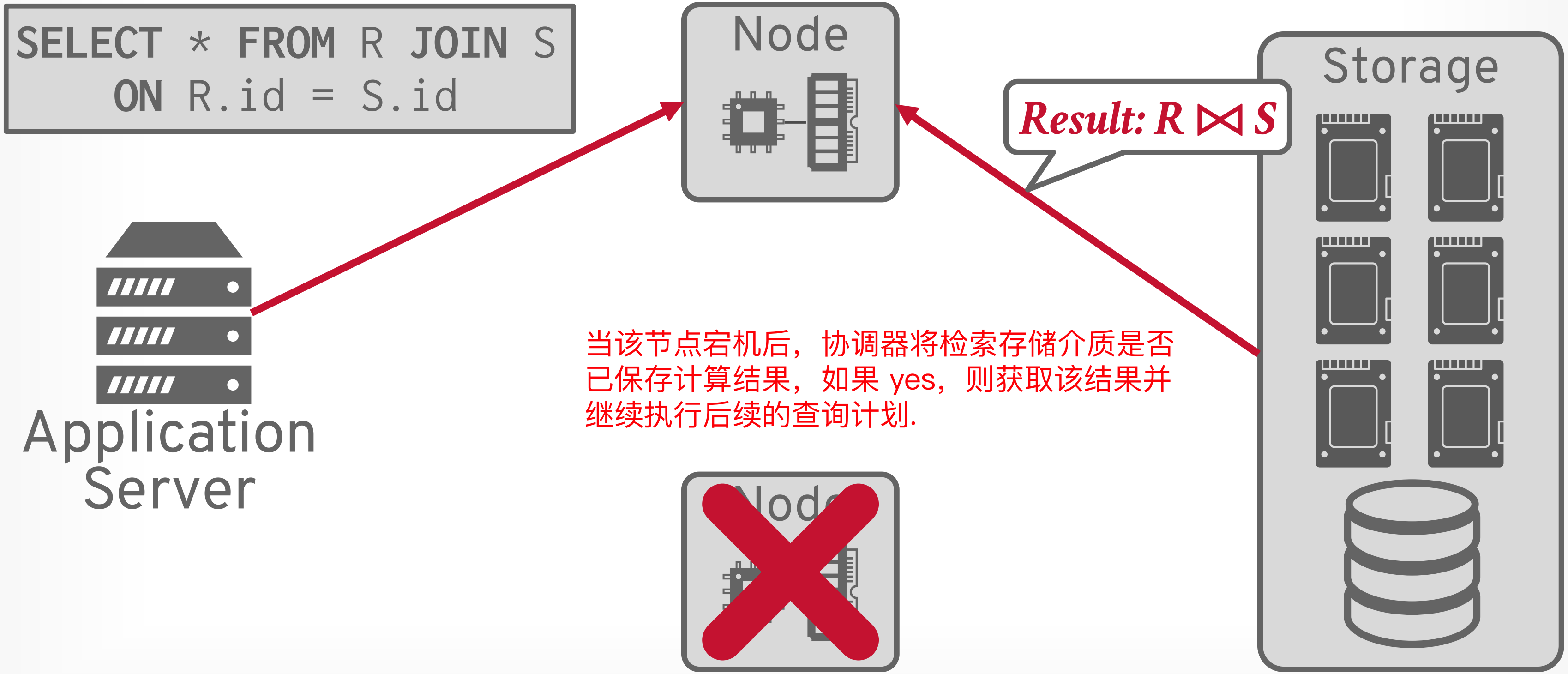
The DBMS could take a snapshot of the intermediate results for a query during execution to allow it to recover if nodes fail. 为查询的中间结果提供故障容错机制.

QUERY FAULT TOLERANCE



QUERY FAULT TOLERANCE

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



当该节点宕机后，协调器将检索存储介质是否已保存计算结果，如果 yes，则获取该结果并继续执行后续的查询计划.

QUERY PLANNING

All the optimizations that we talked about before are still applicable in a distributed environment.
- → Predicate Pushdown
- → Projection Pushdown
- → Optimal Join Orderings

Distributed query optimization is even harder because it must consider the physical location of data and network transfer costs. 成本估算模型.

QUERY PLAN FRAGMENTS

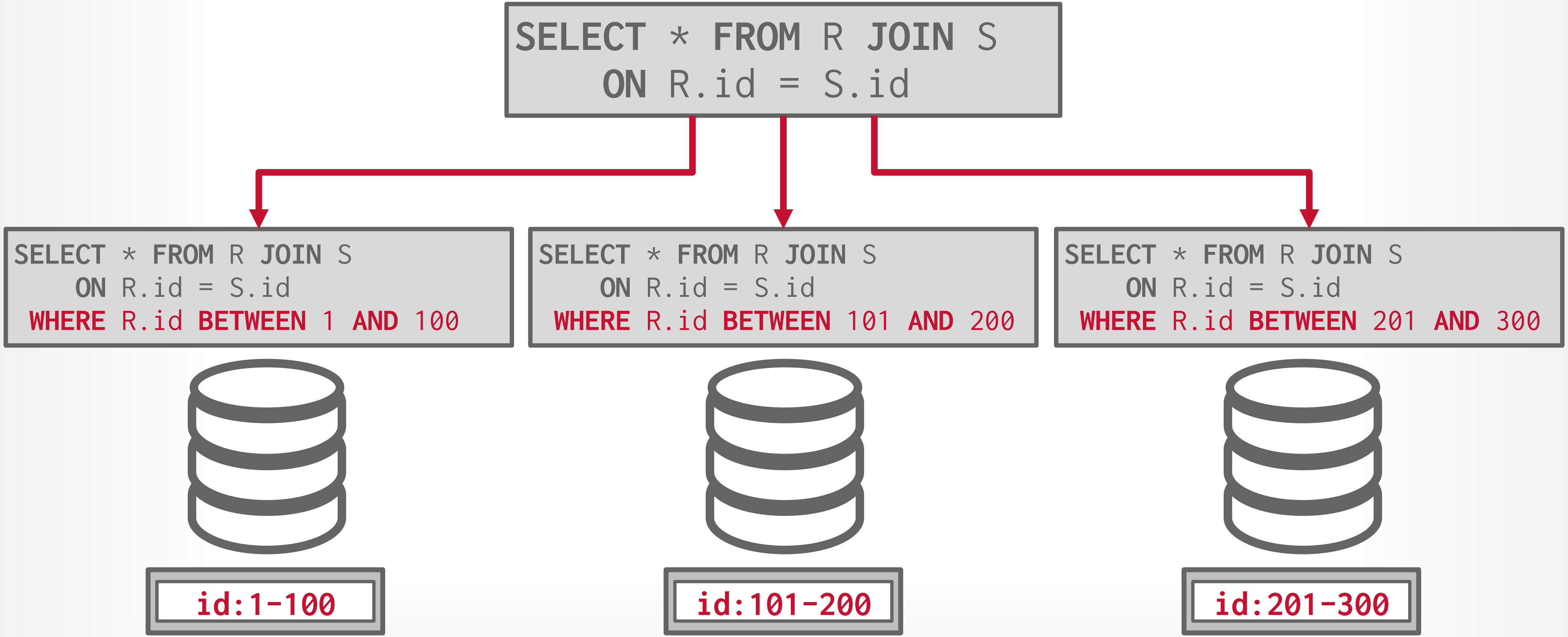
Approach #1: Physical Operators

→ Generate a single query plan and then break it up into partition-specific fragments.
→ Most systems implement this approach.

Approach #2: SQL

→ Rewrite original query into partition-specific queries.
→ Allows for local optimization at each node.
→ SingleStore + Vitess are the only systems we know that use this approach.

QUERY PLAN FRAGMENTS



QUERY PLAN FRAGMENTS

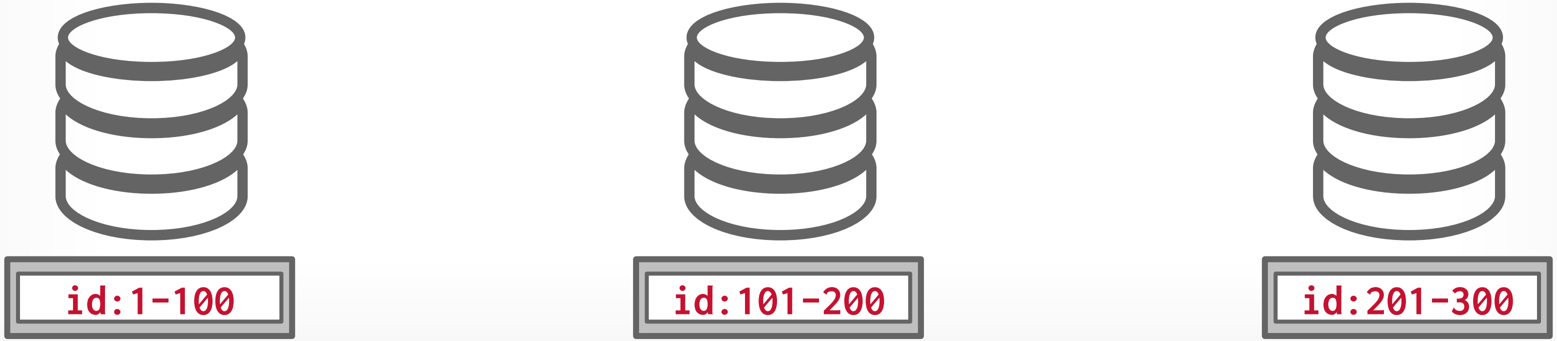
Union the output of each join to produce final result.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

```
SELECT * FROM R JOIN S
    ON R.id = S.id
 WHERE R.id BETWEEN 1 AND 100
```

```
SELECT * FROM R JOIN S
    ON R.id = S.id
 WHERE R.id BETWEEN 101 AND 200
```

```
SELECT * FROM R JOIN S
    ON R.id = S.id
 WHERE R.id BETWEEN 201 AND 300
```



OBSERVATION

The efficiency of a distributed join depends on the target tables' partitioning schemes.

One approach is to put entire tables on a single node and then perform the join.

- → You lose the parallelism of a distributed DBMS.
- → Costly data transfer over the network.

DISTRIBUTED JOIN ALGORITHMS

To join tables **R** and **S**, the DBMS needs to get the proper tuples on the same node.

Once the data is at the node, the DBMS then executes the same join algorithms that we discussed earlier in the semester.

→ Need to produce the correct answer as if all the data is located in a single node system.

SCENARIO #1

The entire copy of one data set is replicated at every node.
→ Think of it as a small dimension table.

Each node joins its local data in parallel and then sends their results to a coordinating node.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```

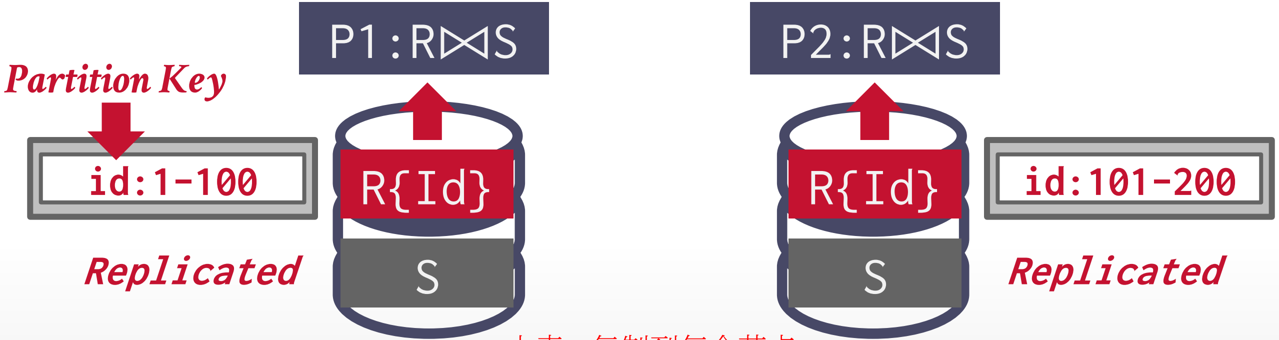
表 R 根据 JOIN 键将数据分区到不同节点.

SCENARIO #1

The entire copy of one data set is replicated at every node.
→ Think of it as a small dimension table.

Each node joins its local data in parallel and then sends their results to a coordinating node. 在每个节点上表 R 分别和 S 进行 JOIN 计算输出结果.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



小表 S 复制到每个节点.

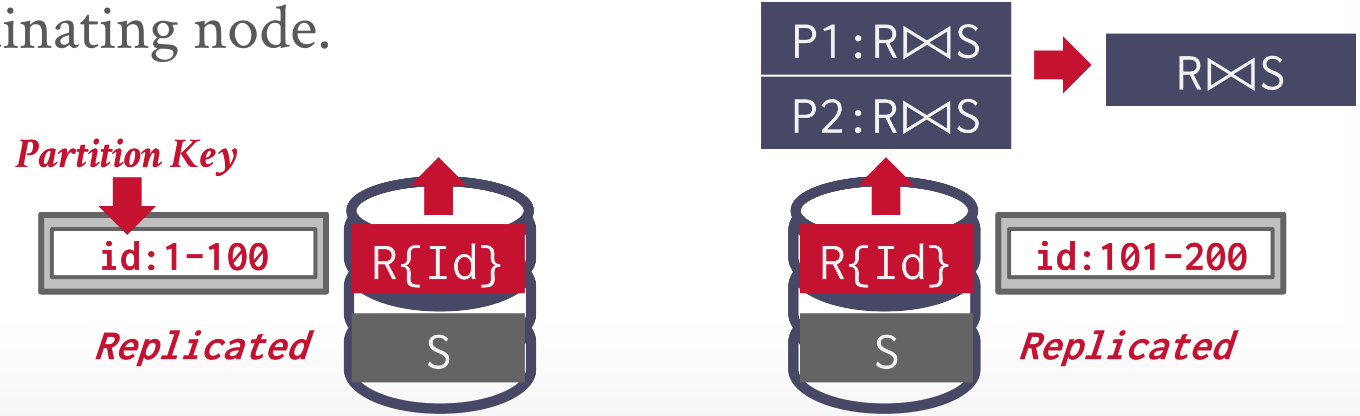
SCENARIO #1

The entire copy of one data set is replicated at every node.

→ Think of it as a small dimension table.

Each node joins its local data in parallel and then sends their results to a coordinating node.

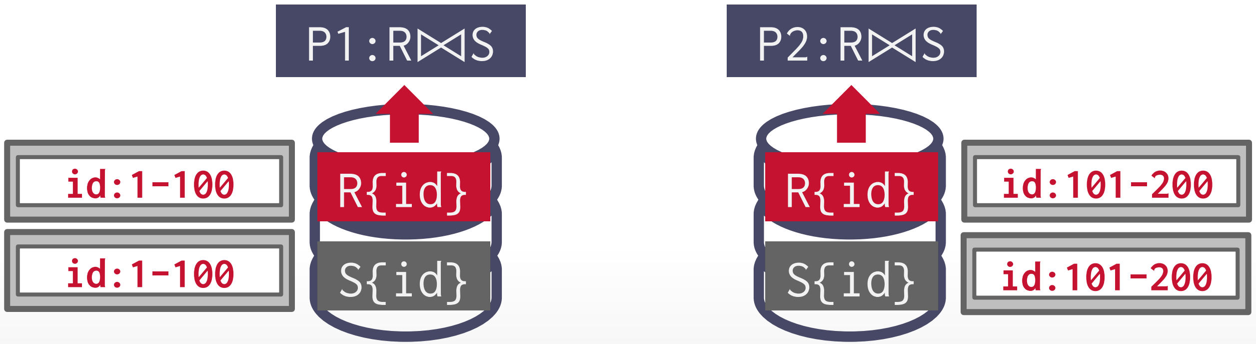
```
SELECT * FROM R JOIN S
ON R.id = S.id
```



SCENARIO #2

Both data sets are partitioned on the join attribute. Each node performs the join on local data and then sends to a coordinator node for coalescing.

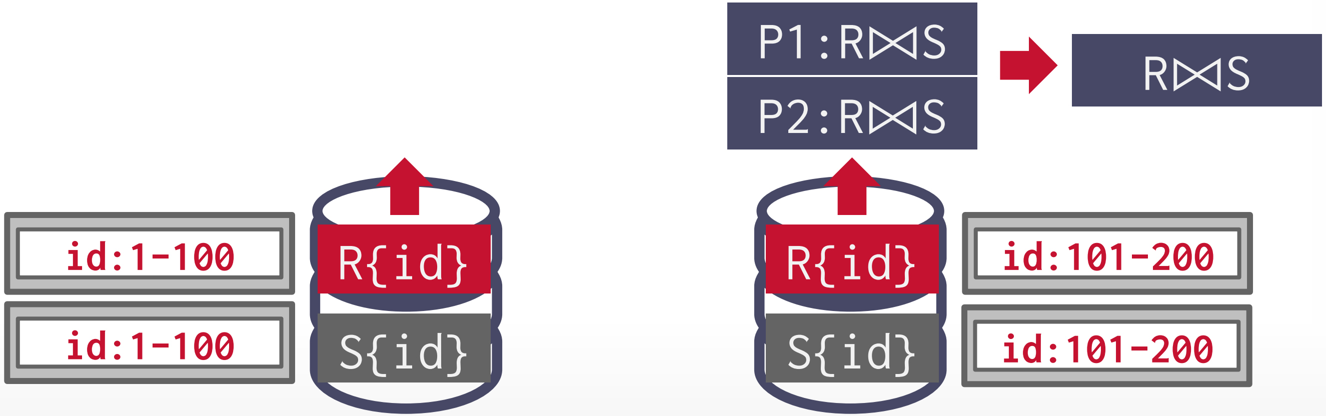
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



SCENARIO #2

Both data sets are partitioned on the join attribute. Each node performs the join on local data and then sends to a coordinator node for coalescing.

```
SELECT * FROM R JOIN S
   ON R.id = S.id
```

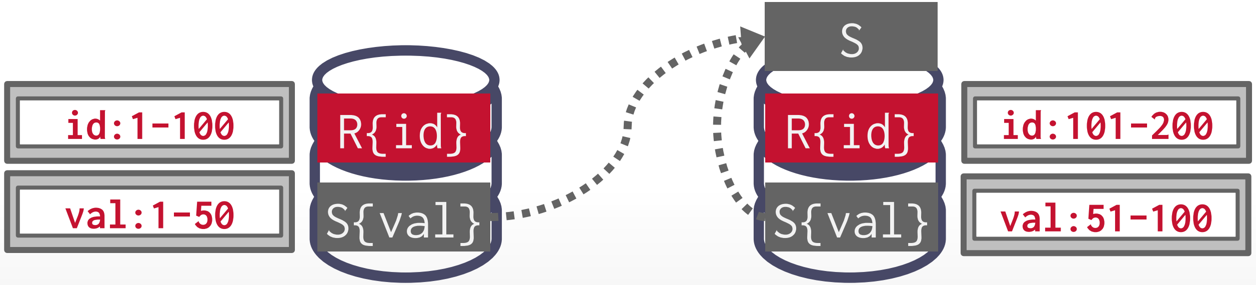


SCENARIO #3 – BROADCAST JOIN

Both data sets are partitioned on different keys. If one of the data sets is small, then the DBMS "broadcasts" that data to all nodes. 两个表按不同的 key 进行数据分区.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

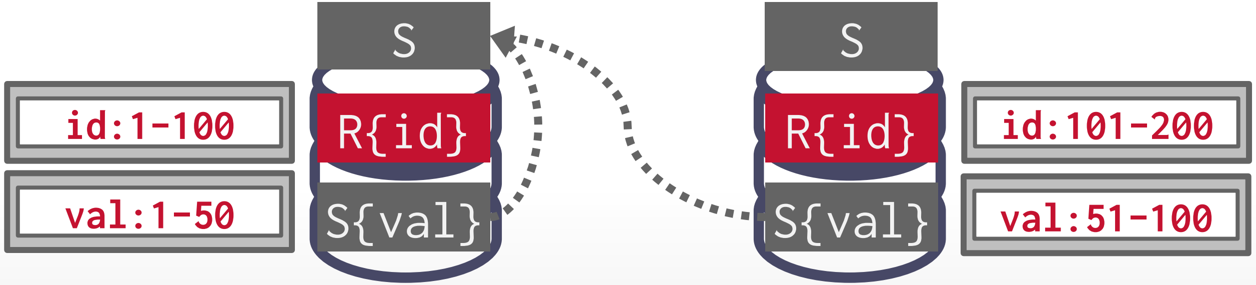
例如，小表 S 按非连接键 val 列进行数据分区. 将一个节点上 S 的部分数据广播到另一个节点，使得这个节点拥有表 S 的全部数据，然后再将 S 的全部数据复制到其他节点进行 JOIN 运算.



SCENARIO #3 – BROADCAST JOIN

Both data sets are partitioned on different keys. If one of the data sets is small, then the DBMS "broadcasts" that data to all nodes.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #3 – BROADCAST JOIN

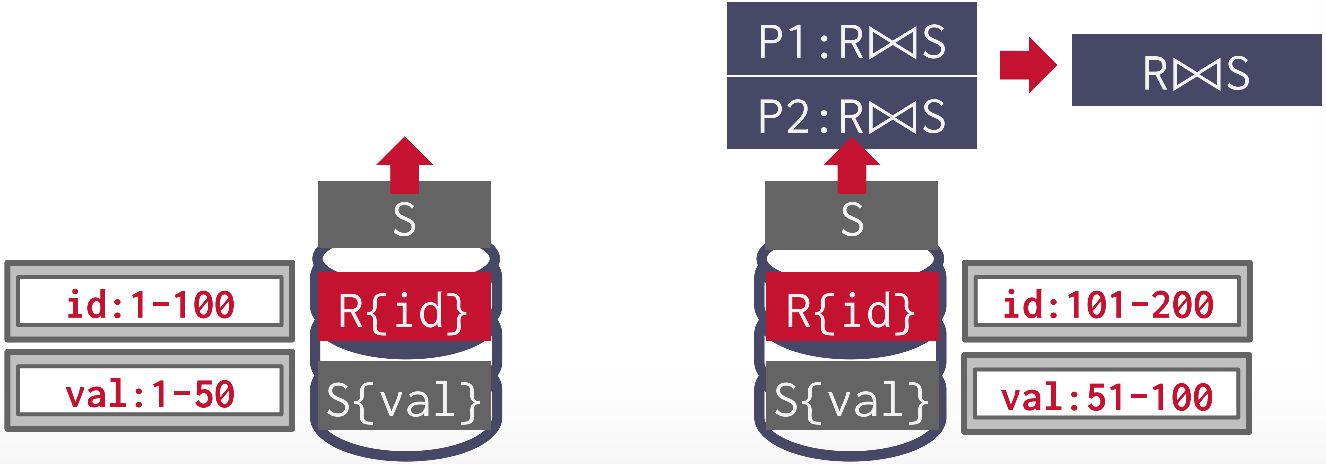
Both data sets are partitioned on different keys. If one of the data sets is small, then the DBMS "broadcasts" that data to all nodes.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

SCENARIO #3 – BROADCAST JOIN

Both data sets are partitioned on different keys. If one of the data sets is small, then the DBMS "broadcasts" that data to all nodes.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #4 – SHUFFLE JOIN

Both data sets are not partitioned on the join key. The DBMS copies/re-partitions the data on-the-fly across nodes.

→ The repartitioned data copy is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

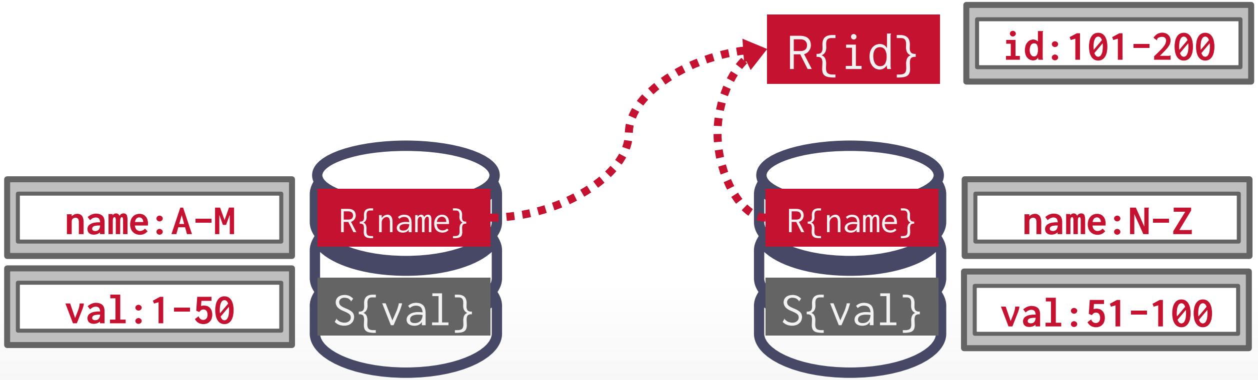
SCENARIO #4 – SHUFFLE JOIN

Both data sets are not partitioned on the join key. The DBMS copies/re-partitions the data on-the-fly across nodes.

→ The repartitioned data copy is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```

SHUFFLE：表 R 首先根据所有数据的 id 列进行 hash 操作，然后进行范围分区，从而将表 R 数据重新分区.

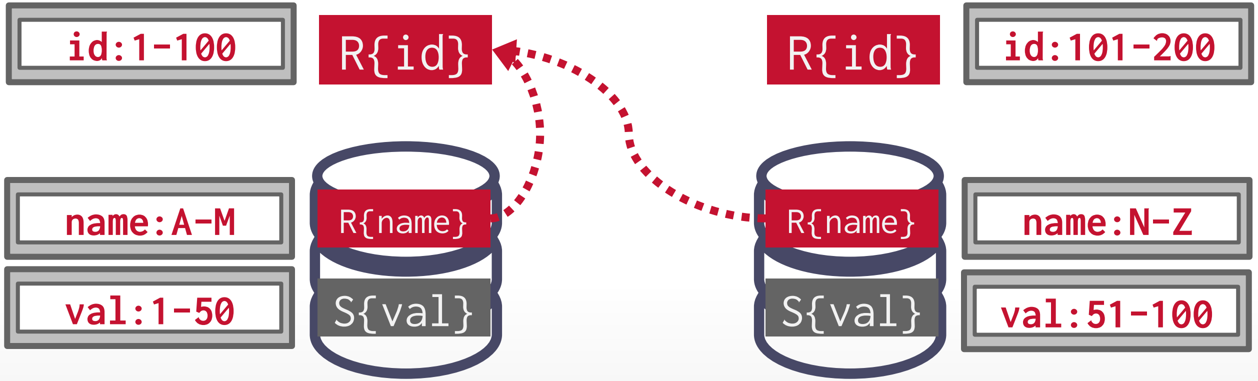


SCENARIO #4 – SHUFFLE JOIN

Both data sets are <u>not</u> partitioned on the join key. The DBMS copies/re-partitions the data on-the-fly across nodes.

→ The repartitioned data copy is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```

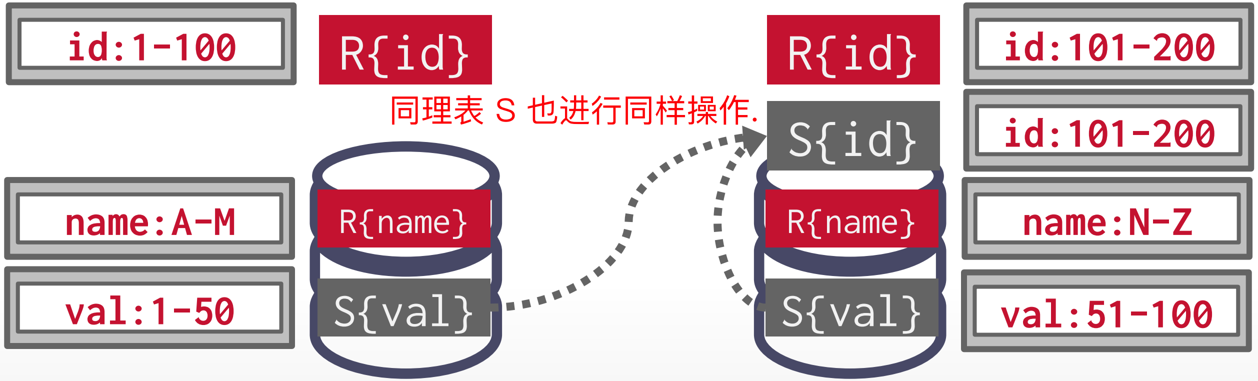


SCENARIO #4 – SHUFFLE JOIN

Both data sets are not partitioned on the join key. The DBMS copies/re-partitions the data on-the-fly across nodes.

→ The repartitioned data copy is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

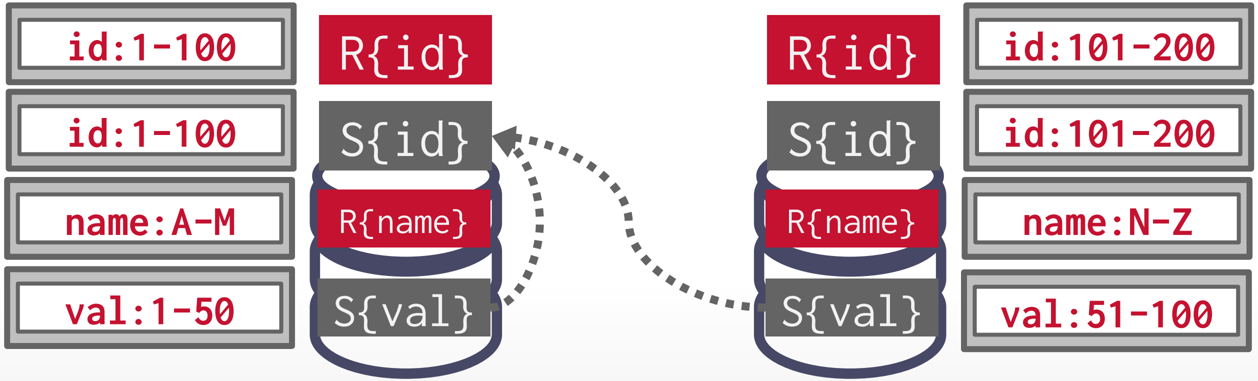


SCENARIO #4 – SHUFFLE JOIN

Both data sets are not partitioned on the join key. The DBMS copies/re-partitions the data on-the-fly across nodes.

→ The repartitioned data copy is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

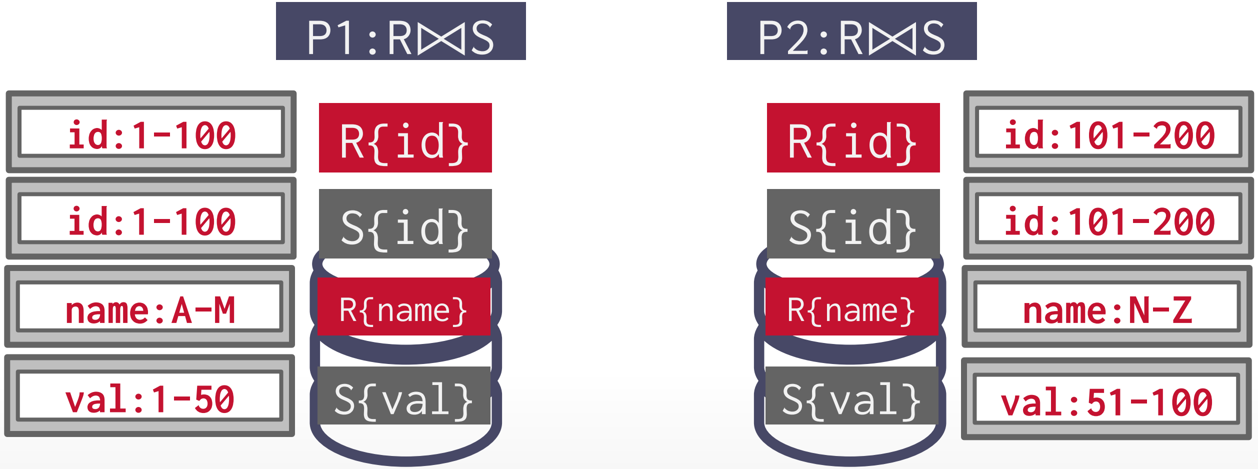


SCENARIO #4 – SHUFFLE JOIN

Both data sets are not partitioned on the join key. The DBMS copies/re-partitions the data on-the-fly across nodes.

→ The repartitioned data copy is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```

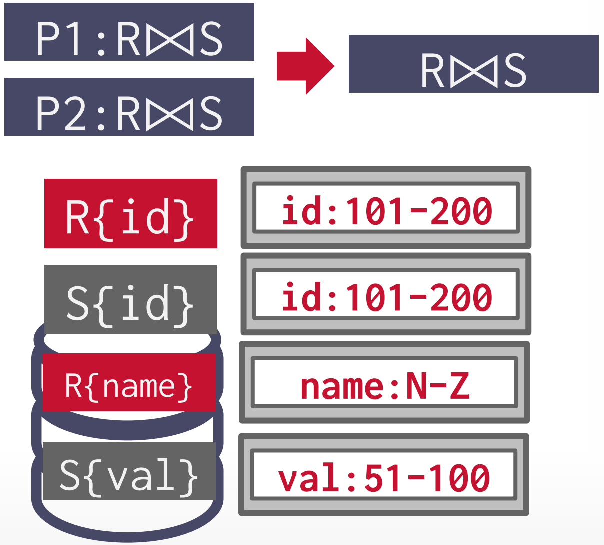


SCENARIO #4 – SHUFFLE JOIN

Both data sets are not partitioned on the join key. The DBMS copies/re-partitions the data on-the-fly across nodes.

→ The repartitioned data copy is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```



SEMI-JOIN OPTIMIZATION

Before pulling data from another node, send a **semi-join filter** to reduce data movement.

→ Perform a join on the bare minimum data needed to avoid unnecessary transfers.
→ Could use an approximate filter (Bloom Join).

```
SELECT Fact.price, Dim.*
  FROM Fact JOIN Dim
    ON Fact.id = Dim.id
 WHERE Dim.zip = 15213
```

SEMI-JOIN OPTIMIZATION

Before pulling data from another node, send a **semi-join filter** to reduce data movement. 减少数据量的移动.

→ Perform a join on the bare minimum data needed to avoid unnecessary transfers.

→ Could use an approximate filter (Bloom Join).

```
SELECT Fact.price, Dim.*
  FROM Fact JOIN Dim
    ON Fact.id = Dim.id
 WHERE Dim.zip = 15213
```

首先对表进行过滤和投影（仅连接键），获得它的最小形式的数据.

$$Dim_{semi} = \Pi_{id} (\sigma_{zip = 15213} Dim)$$

SEMI-JOIN OPTIMIZATION

Before pulling data from another node, send a **semi-join filter** to reduce data movement.

→ Perform a join on the bare minimum data needed to avoid unnecessary transfers.

→ Could use an approximate filter (Bloom Join).

```
SELECT Fact.price, Dim.*
  FROM Fact JOIN Dim
    ON Fact.id = Dim.id
 WHERE Dim.zip = 15213
```

$$\text{Dim}_{\text{semi}} = \Pi_{\text{id}}\,(\sigma_{\text{zip} = 15213}\ \text{Dim})$$

将过滤后的 Dim 表数据发送到表 Fact 位于的节点进行 JOIN 计算.

$$\text{F-small} = \text{Fact} \bowtie \text{Dim}_{\text{semi}}$$

SEMI-JOIN OPTIMIZATION

Before pulling data from another node, send a **semi-join filter** to reduce data movement.

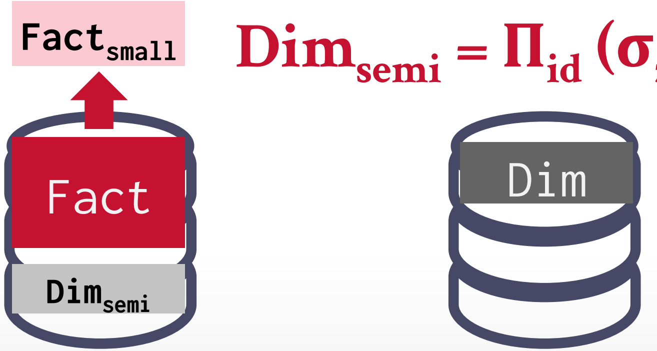
→ Perform a join on the bare minimum data needed to avoid unnecessary transfers.

→ Could use an approximate filter (Bloom Join).

```
SELECT Fact.price, Dim.*
  FROM Fact JOIN Dim
    ON Fact.id = Dim.id
 WHERE Dim.zip = 15213
```

$$Dim_{semi} = \Pi_{id}(\sigma_{zip = 15213} Dim)$$

$$F\text{-}small = Fact \bowtie Dim_{semi}$$



SEMI-JOIN OPTIMIZATION

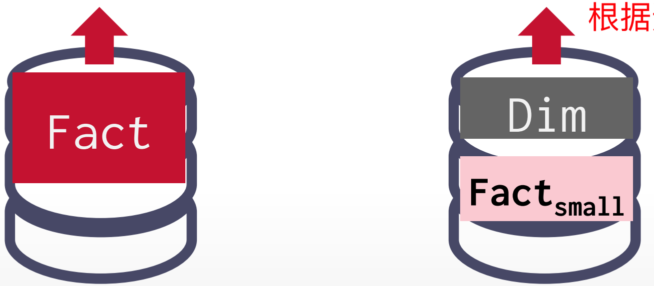
Before pulling data from another node, send a **<u>semi-join filter</u>** to reduce data movement.

→ Perform a join on the bare minimum data needed to avoid unnecessary transfers.
→ Could use an approximate filter (Bloom Join).

```
SELECT Fact.price, Dim.*
  FROM Fact JOIN Dim
    ON Fact.id = Dim.id
 WHERE Dim.zip = 15213
```

$$\textbf{Result} = \Pi_{price}(\textbf{Dim} \bowtie \textbf{Fact}_{small})$$

根据过滤后的表 Fact 数据再次进行 JOIN 计算.



OBSERVATION

Direct communication between compute nodes means the DBMS knows which nodes will participate in query execution ahead of time.

But data skew can cause imbalances…

A better approach is to dynamically adjust compute resources on the fly as a query executes.

SHUFFLE PHASE

Redistribute of intermediate data across nodes between query plan pipelines. pipeline breaker.

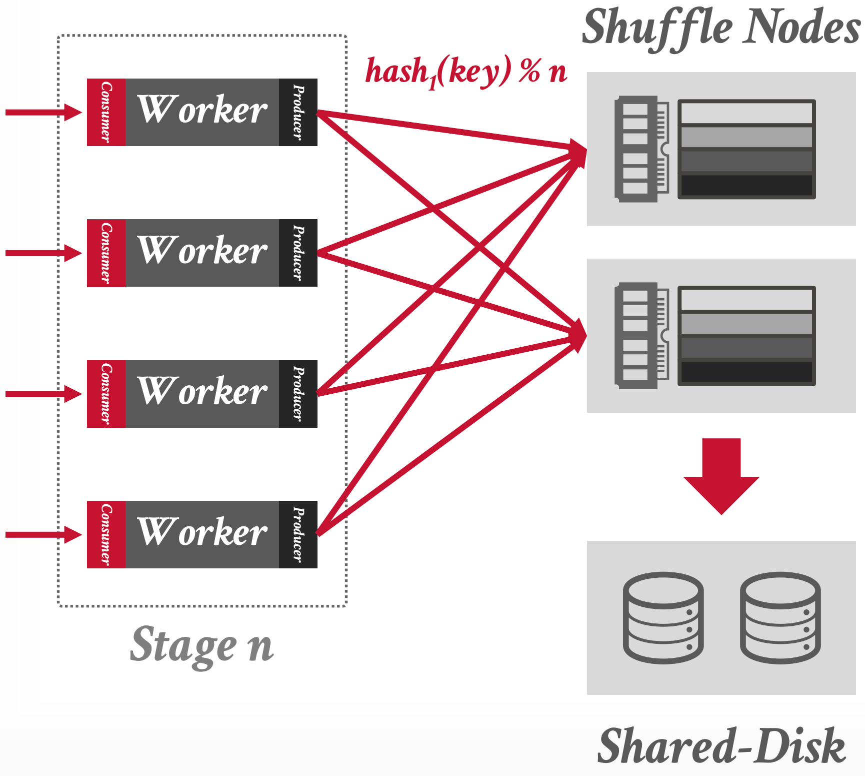
→ Can repartition / rebalance data based on observed data characteristics.

使用专门的硬件进行 SHUFFLE.

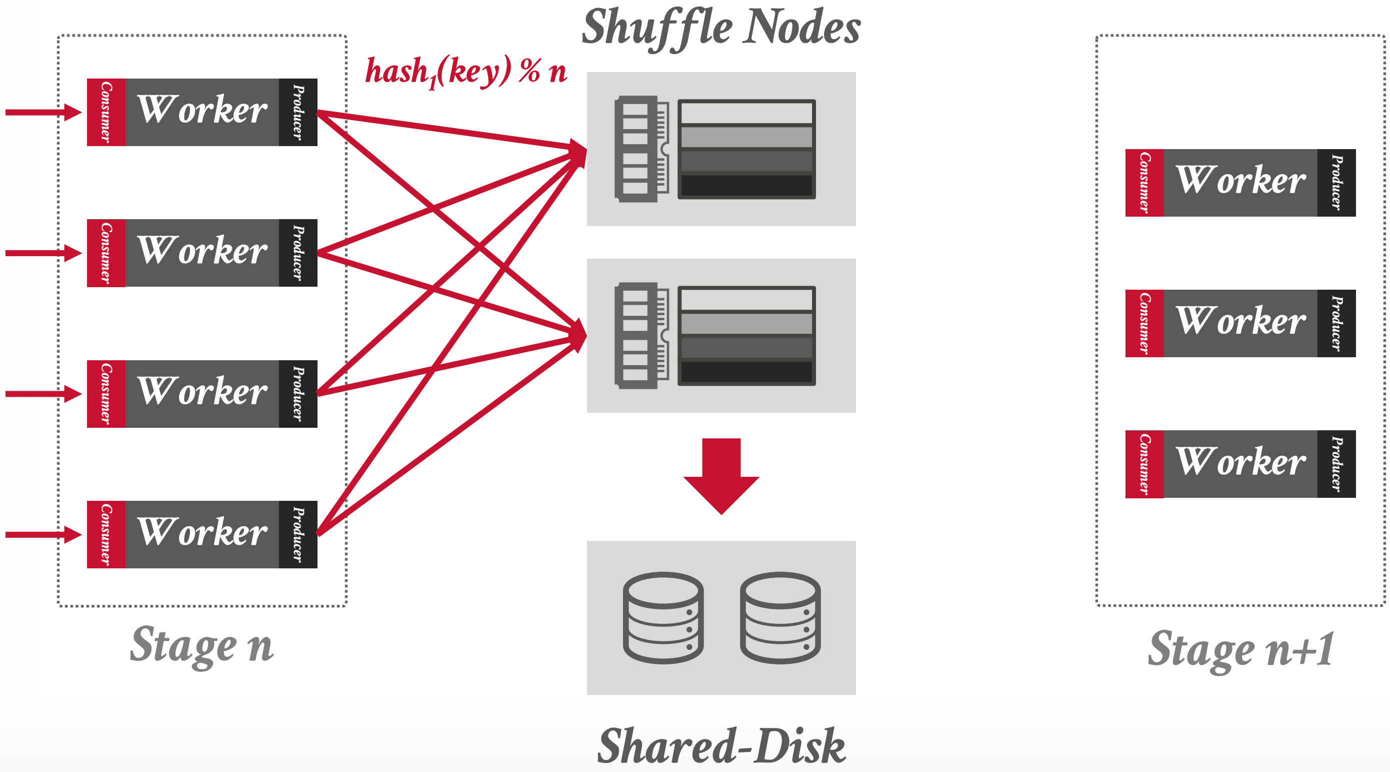
Some DBMSs support standalone fault-tolerant shuffle services.

→ Example: You can replace Spark's built-in in-memory shuffle implementation or replace it with a separate service.

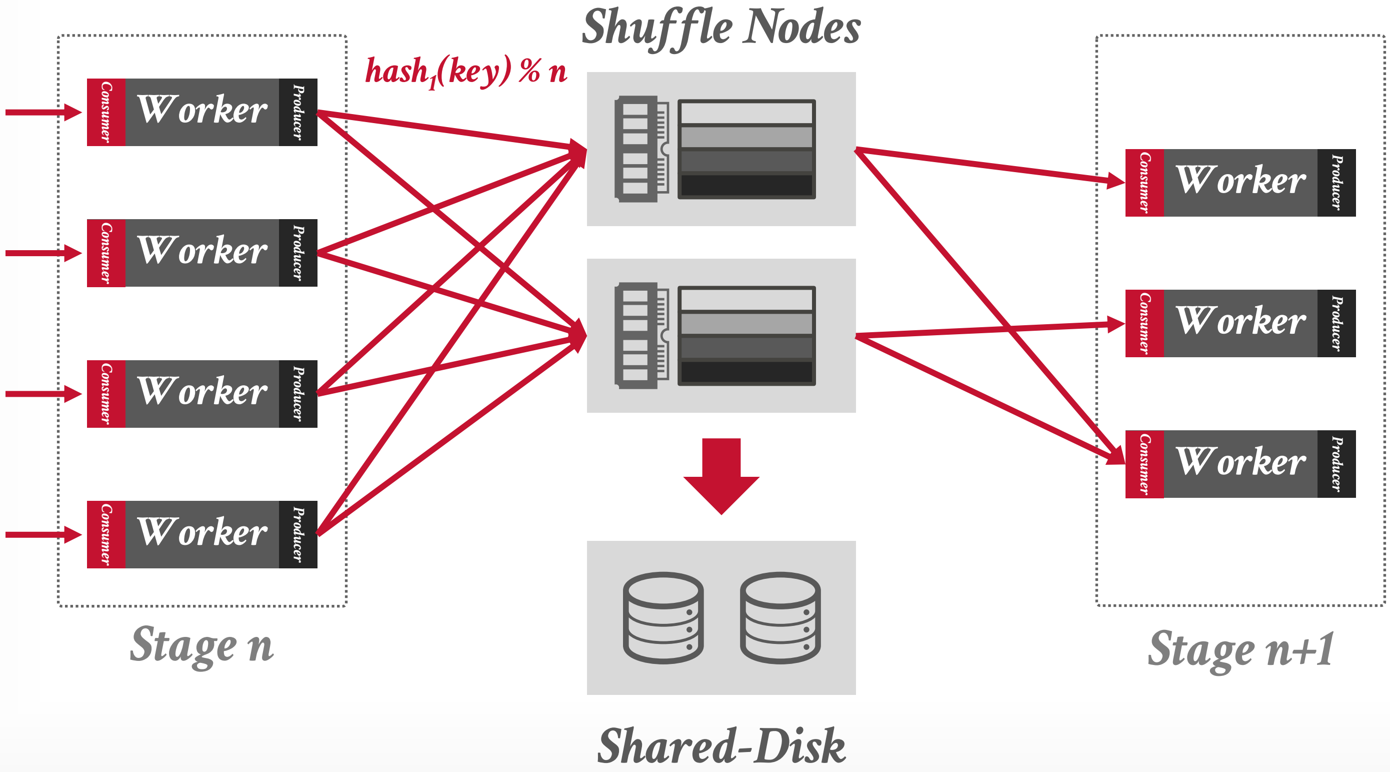
SHUFFLE PHASE



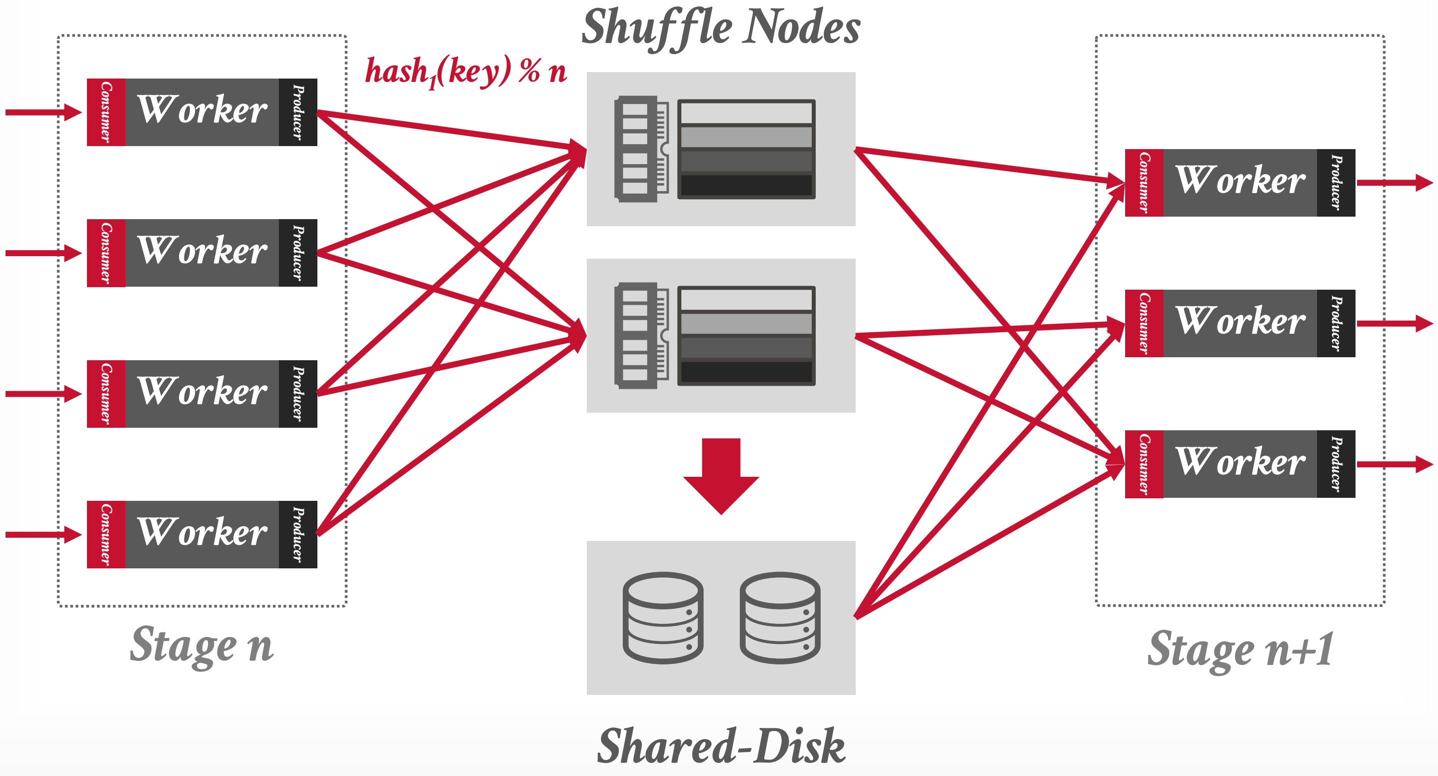
SHUFFLE PHASE



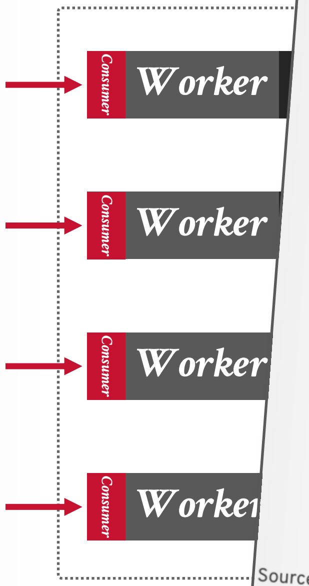
SHUFFLE PHASE



SHUFFLE PHASE



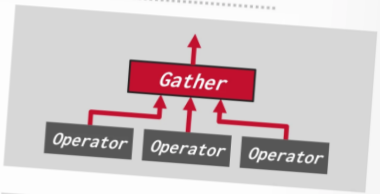
SHUFFLE PHASE



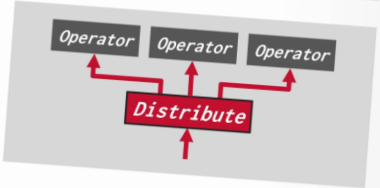
Stage

EXCHANGE OPERATOR

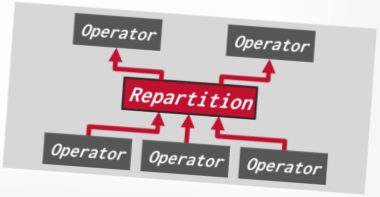
Exchange Type #1 – Gather
→ Combine the results from multiple workers into a single output stream.



Exchange Type #2 – Distribute
→ Split a single input stream into multiple output streams.



Exchange Type #3 – Repartition
→ Shuffle multiple input streams across multiple output streams.
→ Some DBMSs always perform this step after every pipeline (e.g., Dremel/BigQuery).



Source: Craig Freedman

Shared-Disk

CLOUD SYSTEMS

Vendors provide ***database-as-a-service*** (DBaaS) offerings that are managed DBMS environments.

Newer systems are starting to blur the lines between shared-nothing and shared-disk.

→ Example: You can do simple filtering on Amazon S3 before copying data to compute nodes.

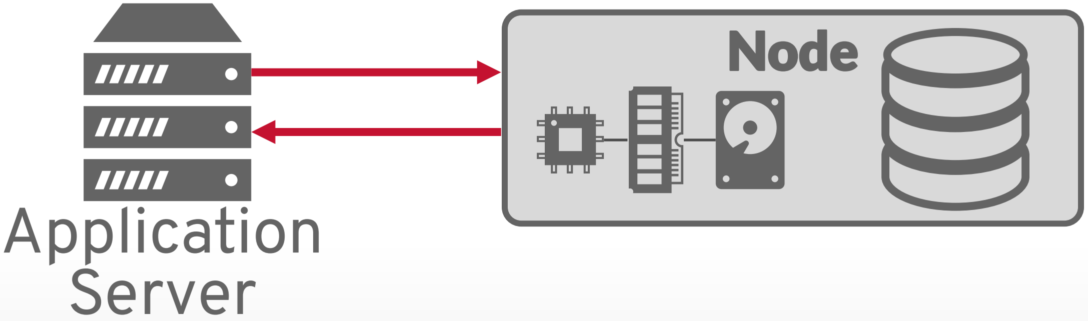
CLOUD SYSTEMS

Approach #1: Managed DBMSs
- → No significant modification to the DBMS to be "aware" that it is running in a cloud environment.
- → Examples: Most vendors

Approach #2: Cloud-Native DBMS
- → System designed explicitly to run in a cloud environment.
- → Usually based on a shared-disk architecture.
- → Examples: Snowflake, Google BigQuery

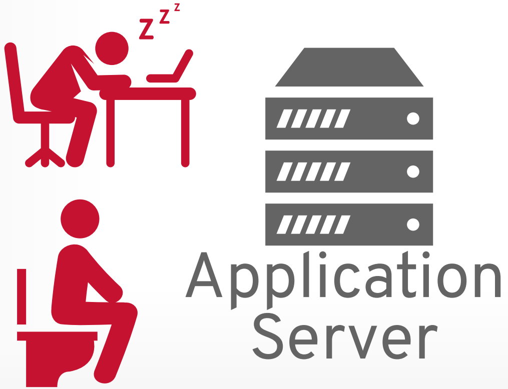
SERVERLESS DATABASES

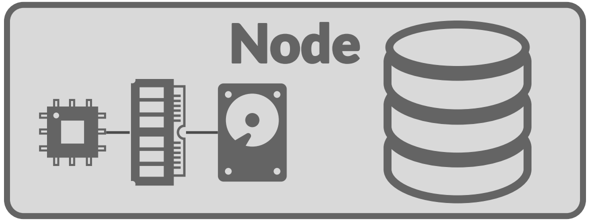
Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



SERVERLESS DATABASES

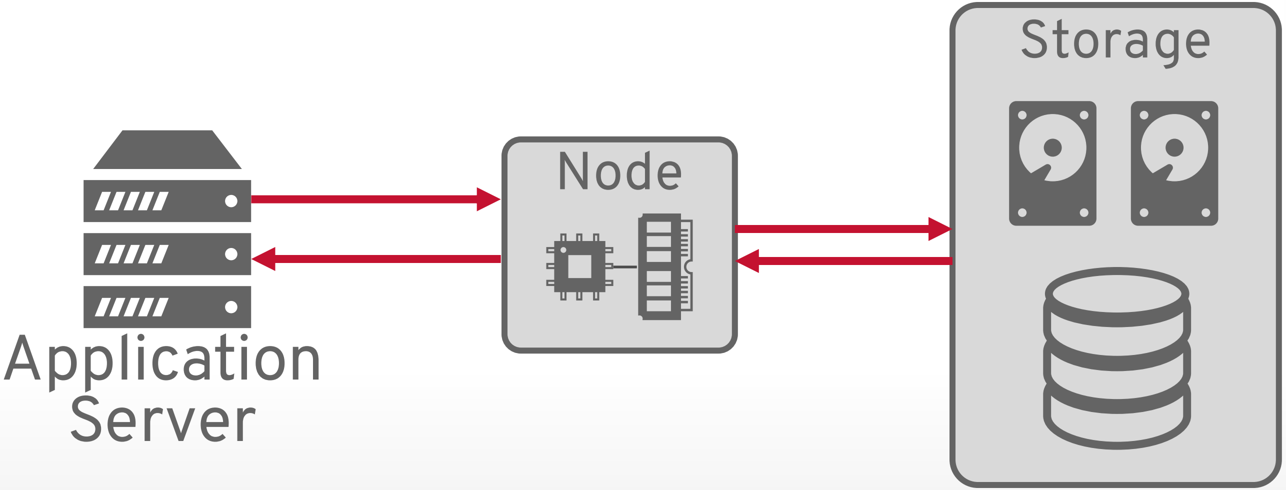
Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.





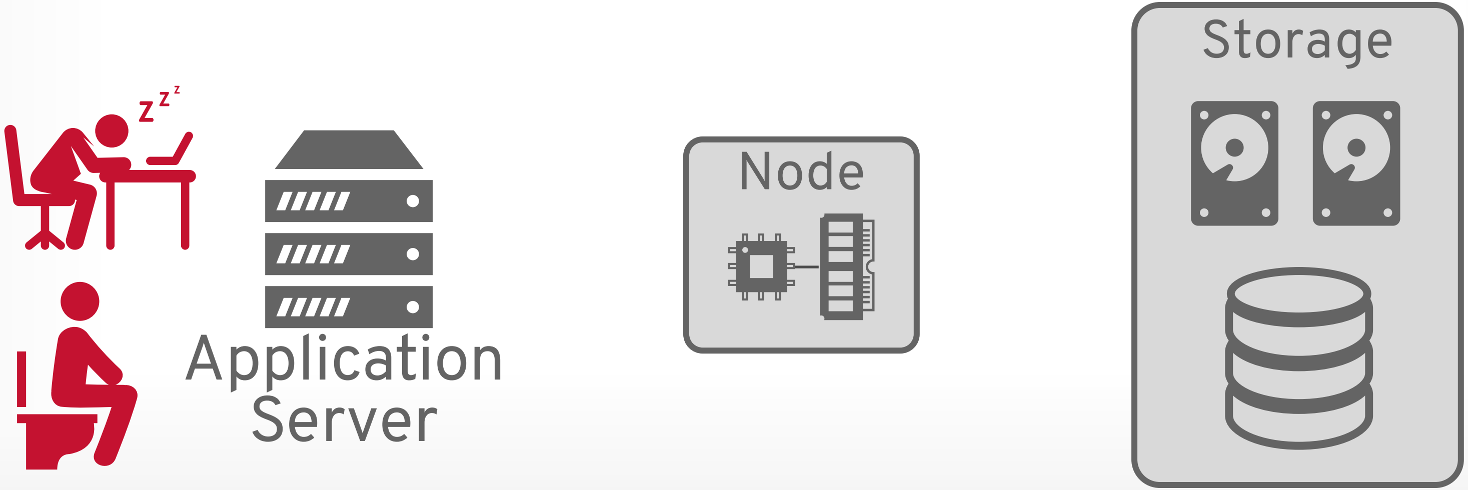
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



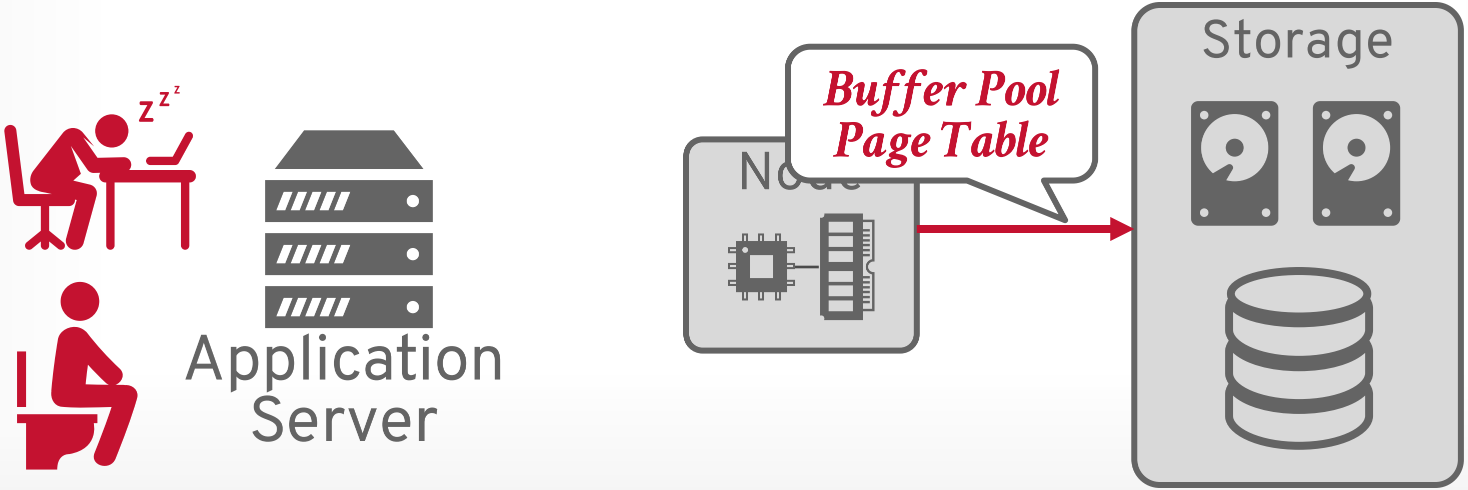
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



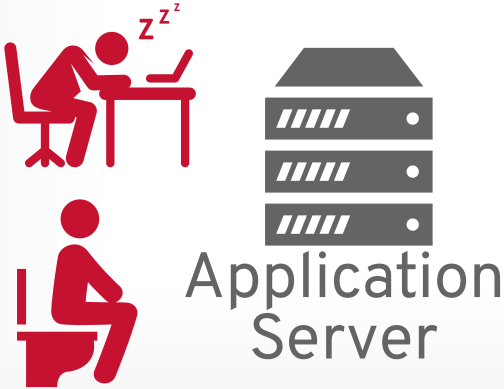
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



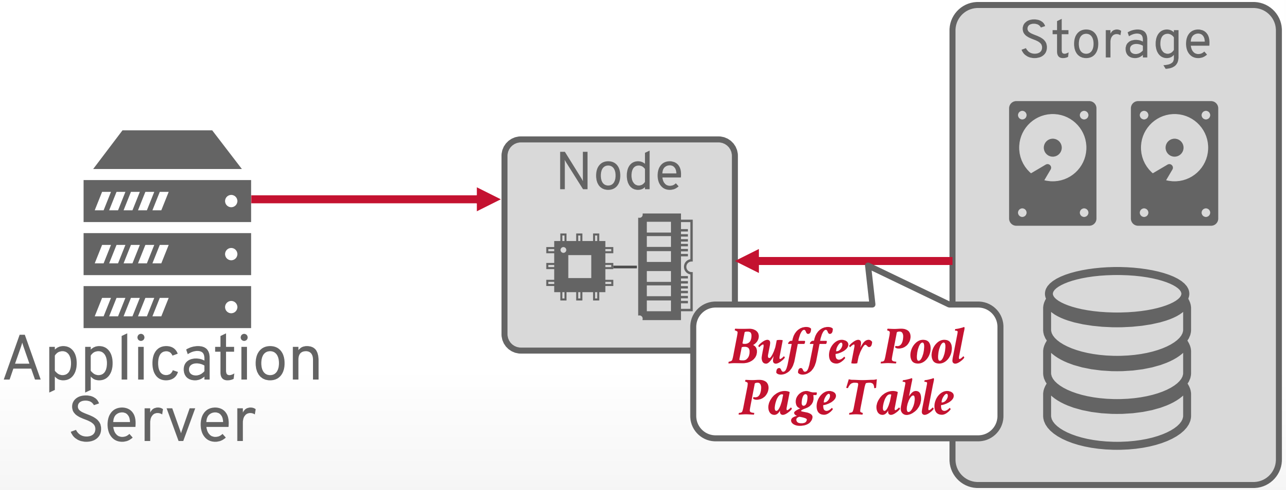
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



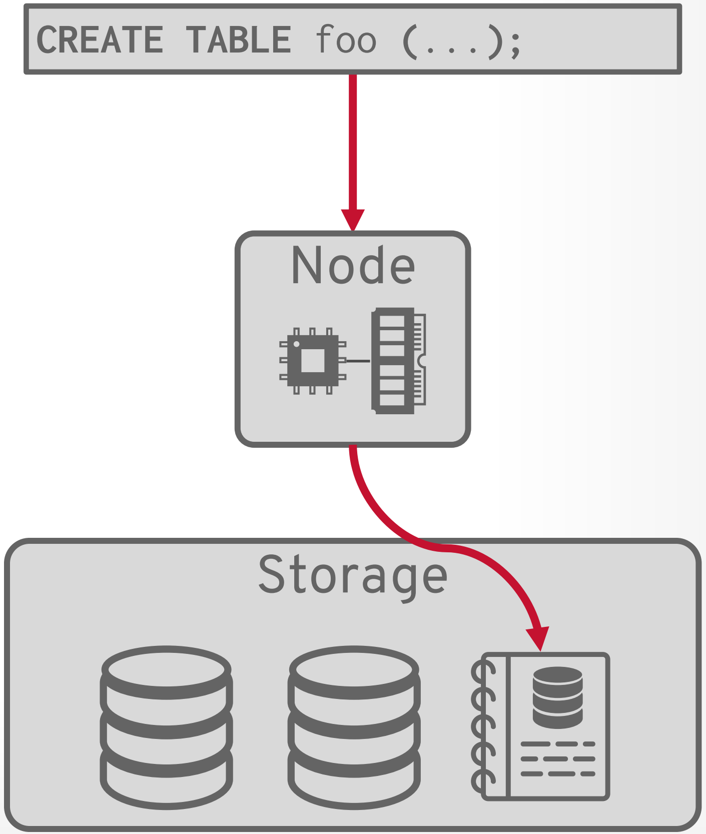
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



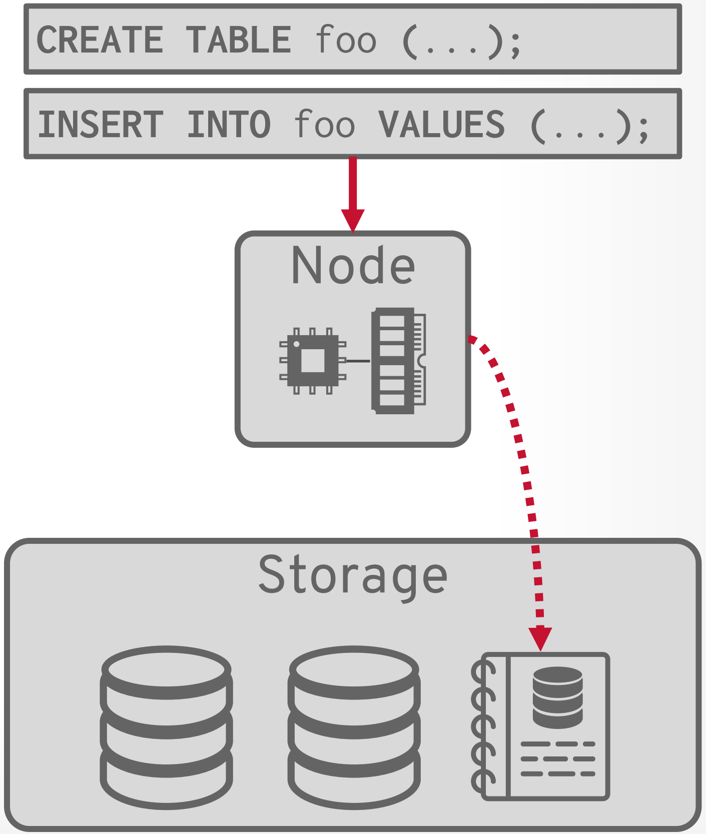
DATA LAKES

Repository for storing large amounts of structured, semi-structured, and unstructured data without having to define a schema or ingest the data into proprietary internal formats.



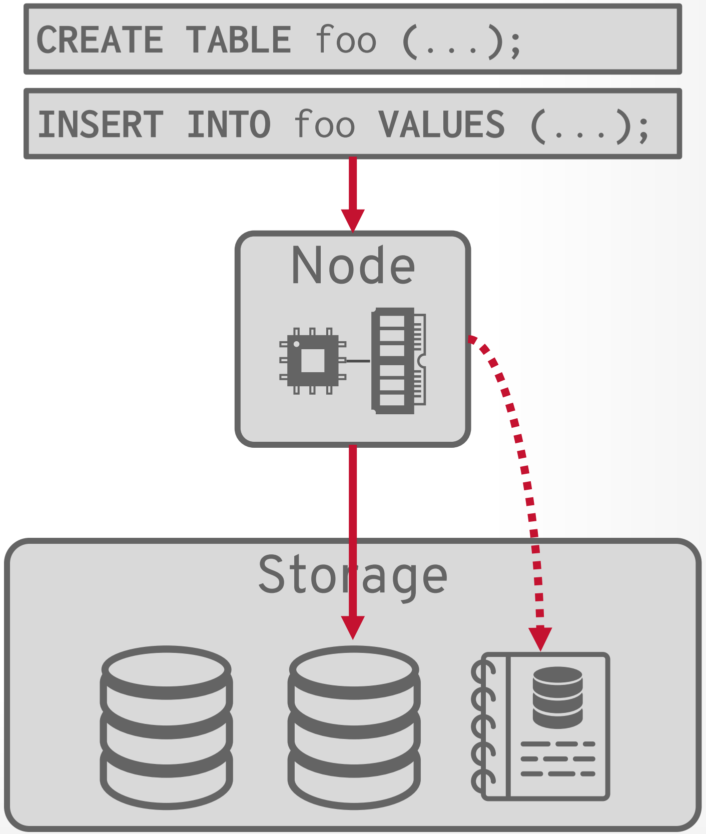
DATA LAKES

Repository for storing large amounts of structured, semi-structured, and unstructured data without having to define a schema or ingest the data into proprietary internal formats.



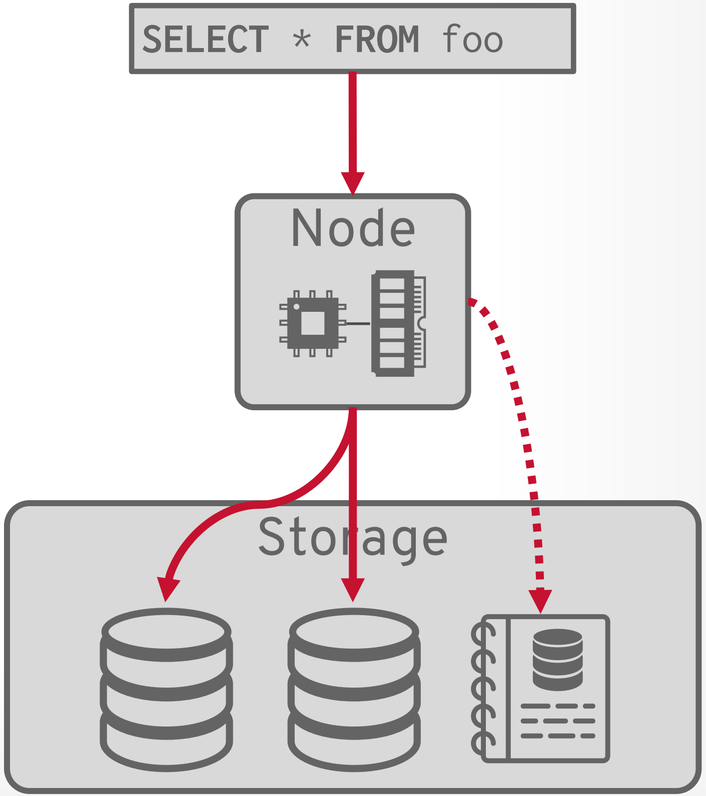
DATA LAKES

Repository for storing large amounts of structured, semi-structured, and unstructured data without having to define a schema or ingest the data into proprietary internal formats.



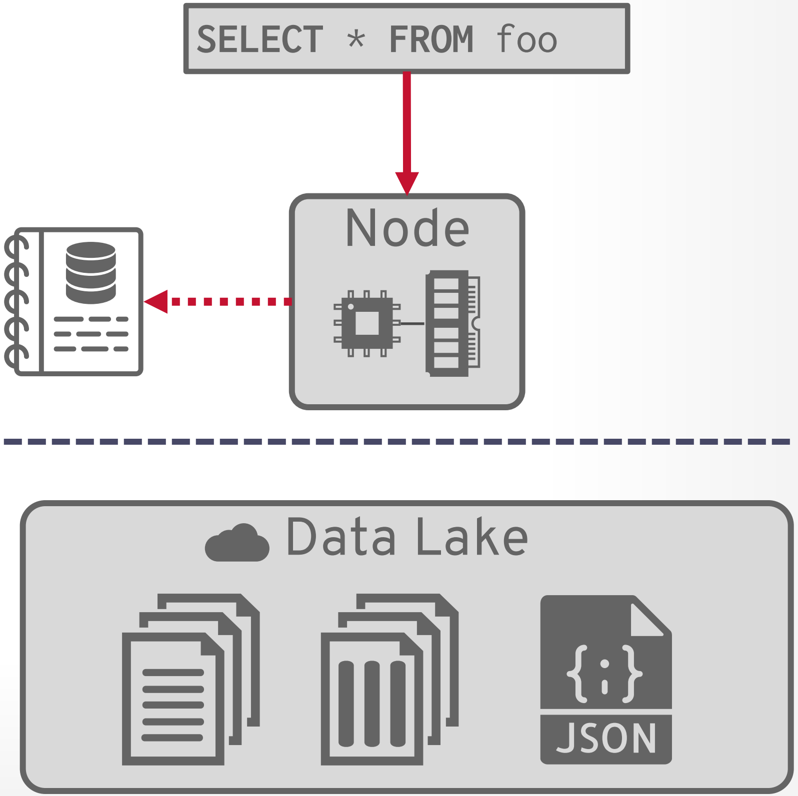
DATA LAKES

Repository for storing large amounts of structured, semi-structured, and unstructured data without having to define a schema or ingest the data into proprietary internal formats.



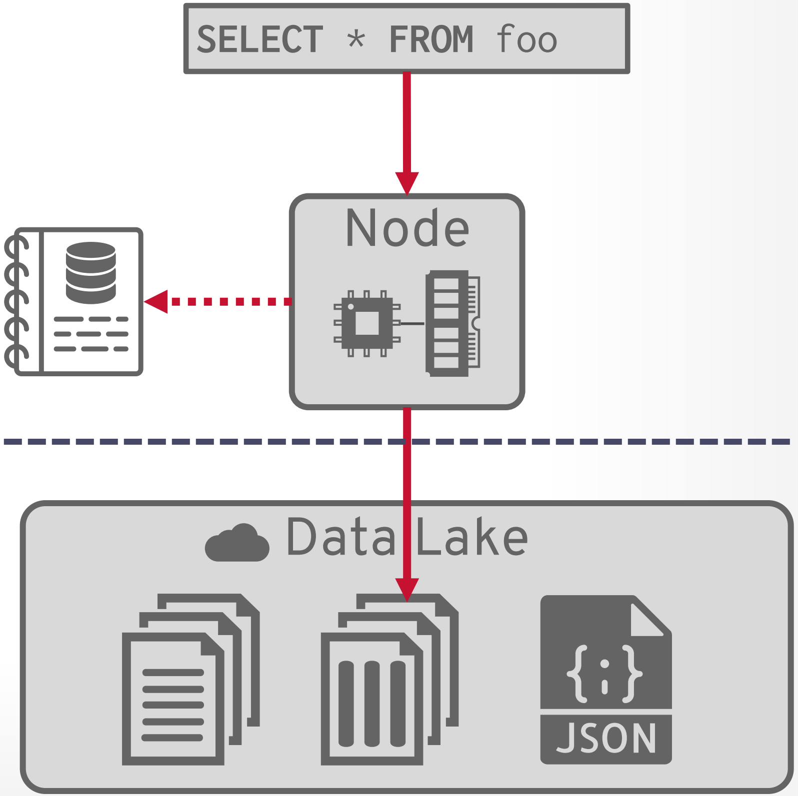
DATA LAKES

Repository for storing large amounts of structured, semi-structured, and unstructured data without having to define a schema or ingest the data into proprietary internal formats.



DATA LAKES

Repository for storing large amounts of structured, semi-structured, and unstructured data without having to define a schema or ingest the data into proprietary internal formats.



trino amazon REDSHIFT Google Big Query HIVE databricks snowflake presto

OLAP DBMS COMPONENTS

One recent trend of the last decade is the breakout of OLAP DBMS components into standalone services and libraries:

→ System Catalogs
→ Intermediate Representation
→ Query Optimizers
→ File Format / Access Libraries
→ Execution Engines / Fabrics

Lots of engineering challenges to make these components interoperable + performant.

SYSTEM CATALOGS

A DBMS tracks a database's schema (table, columns) and data files in its catalog.

→ If the DBMS is on the data ingestion path, then it can maintain the catalog incrementally.

→ If an external process adds data files, then it also needs to update the catalog so that the DBMS is aware of them.

Notable implementations:

→ HCatalog

→ Google Data Catalog

→ Amazon Glue Data Catalog

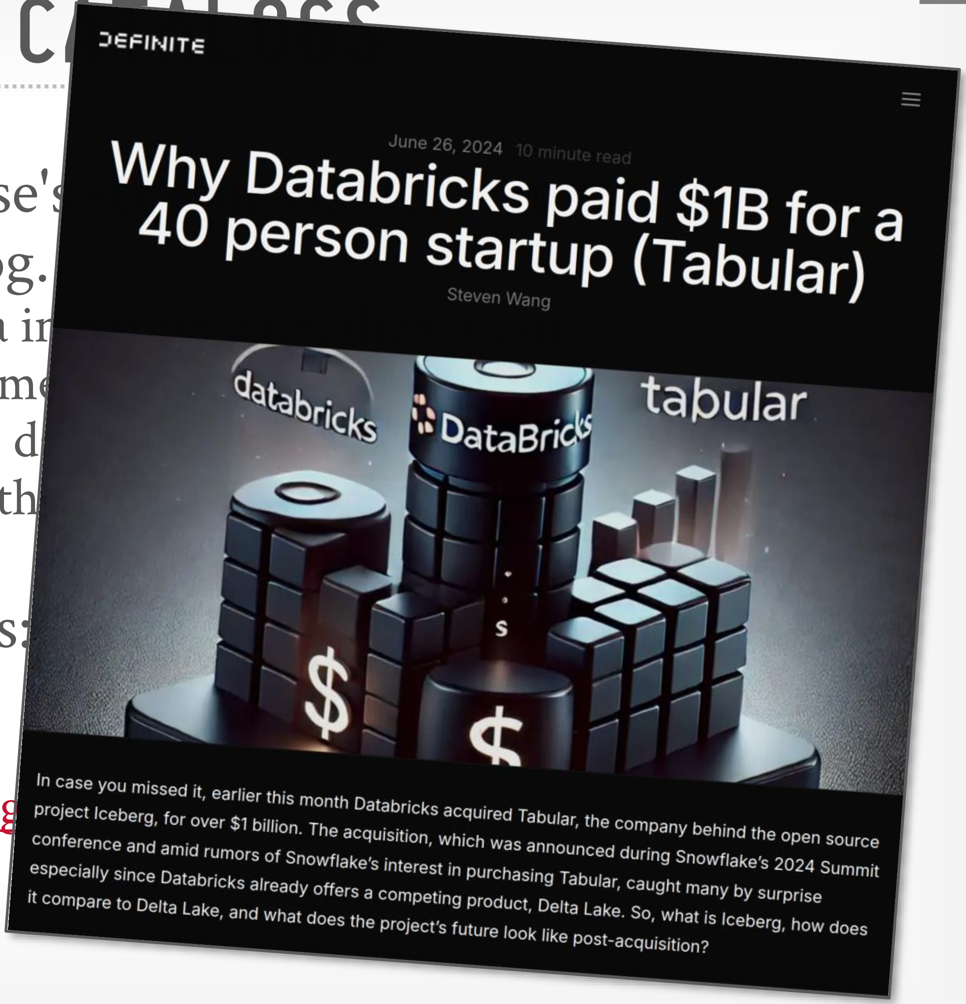
→ Databricks Unity

→ Apache Iceberg

SYSTEM CA

A DBMS tracks a database's
and data files in its catalog.
→ If the DBMS is on the data in
maintain the catalog increme
→ If an external process adds d
update the catalog so that th

Notable implementations:
→ HCatalog
→ Google Data Catalog
→ Amazon Glue Data Catalog
→ Databricks Unity
→ Apache Iceberg



DEFINITE

June 26, 2024 10 minute read

Why Databricks paid $1B for a 40 person startup (Tabular)

Steven Wang

In case you missed it, earlier this month Databricks acquired Tabular, the company behind the open source project Iceberg, for over $1 billion. The acquisition, which was announced during Snowflake's 2024 Summit conference and amid rumors of Snowflake's interest in purchasing Tabular, caught many by surprise especially since Databricks already offers a competing product, Delta Lake. So, what is Iceberg, how does it compare to Delta Lake, and what does the project's future look like post-acquisition?

QUERY OPTIMIZERS

Extendible search engine framework for heuristic- and cost-based query optimization.

→ DBMS provides transformation rules and cost estimates.
→ Framework returns either a logical or physical query plan.

Notable implementations:

→ Greenplum Orca
→ Apache Calcite

This is what 15-799 will cover next semester!

DATA FILE FORMATS

Most DBMSs use a proprietary on-disk binary file format for their databases.

→ Think of the BusTub page types…

The only way to share data between systems is to convert data into a common text-based format

→ Examples: CSV, JSON, XML

There are new open-source binary file formats that make it easier to access data across systems.

DATA FILE FORMATS

Apache Parquet

→ Compressed columnar storage from Cloudera/Twitter

Apache ORC

→ Compressed columnar storage from Apache Hive.

Apache CarbonData

→ Compressed columnar storage with indexes from Huawei.

Apache Iceberg

→ Flexible data format that supports schema evolution from Netflix.

HDF5

→ Multi-dimensional arrays for scientific workloads.

Apache Arrow

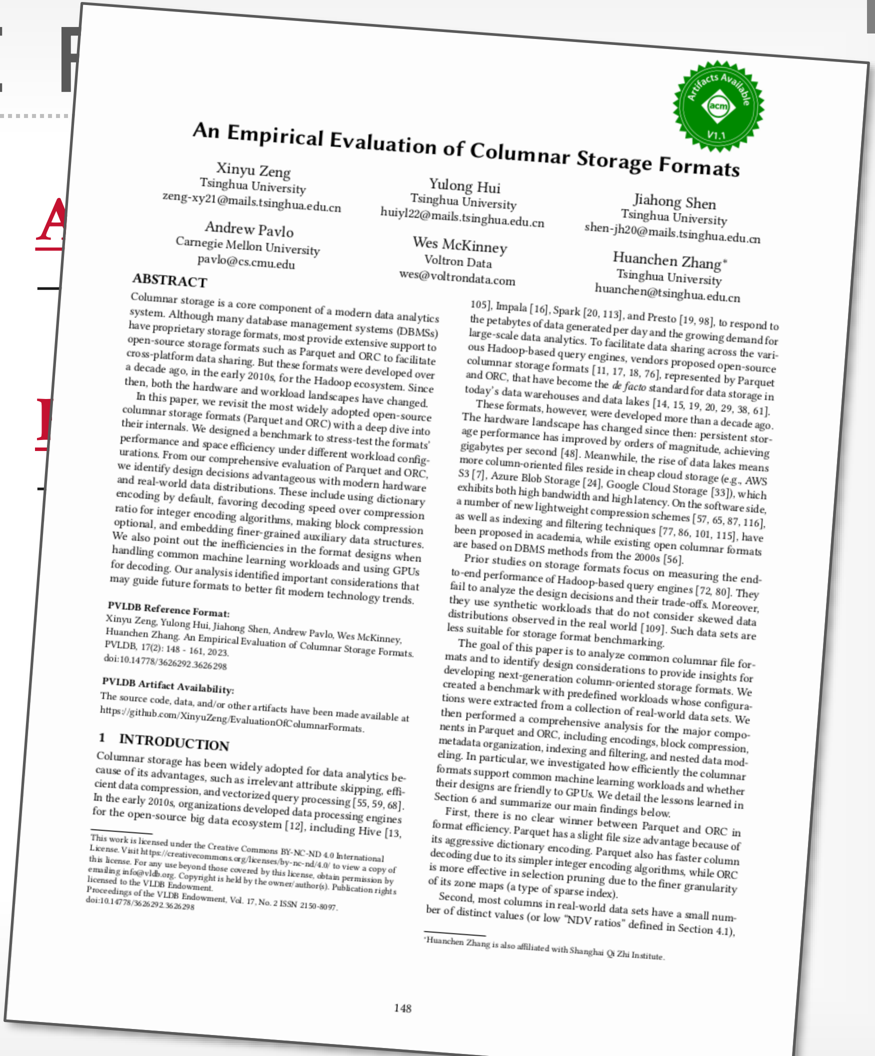
→ In-memory compressed columnar storage from Pandas/Dremio.

DATA FILE F

Apache Parquet
→ Compressed columnar storage from Cloudera/Twitter

Apache ORC
→ Compressed columnar storage from Apache Hive.

Apache CarbonData
→ Compressed columnar storage with indexes from Huawei.



An Empirical Evaluation of Columnar Storage Formats

Xinyu Zeng
Tsinghua University
zeng-xy21@mails.tsinghua.edu.cn

Yulong Hui
Tsinghua University
huiyl22@mails.tsinghua.edu.cn

Jiahong Shen
Tsinghua University
shen-jh20@mails.tsinghua.edu.cn

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Wes McKinney
Voltron Data
wes@voltrondata.com

Huanchen Zhang*
Tsinghua University
huanchen@tsinghua.edu.cn

ABSTRACT

Columnar storage is a core component of a modern data analytics system. Although many database management systems (DBMSs) have proprietary storage formats, most provide extensive support to open-source storage formats such as Parquet and ORC to facilitate cross-platform data sharing. But these formats were developed over a decade ago, in the early 2010s, for the Hadoop ecosystem. Since then, both the hardware and workload landscapes have changed.

In this paper, we revisit the most widely adopted open-source columnar storage formats (Parquet and ORC) with a deep dive into their internals. We designed a benchmark to stress-test the formats' performance and space efficiency under different workload configurations. From our comprehensive evaluation of Parquet and ORC, we identify design decisions advantageous with modern hardware and real-world data distributions. These include using dictionary encoding by default, favoring decoding speed over compression ratio for integer encoding algorithms, making block compression optional, and embedding finer-grained auxiliary data structures. We also point out the inefficiencies in the format designs when handling common machine learning workloads and using GPUs for decoding. Our analysis identified important considerations that may guide future formats to better fit modern technology trends.

PVLDB Reference Format:
Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, Huanchen Zhang. An Empirical Evaluation of Columnar Storage Formats. PVLDB, 17(2): 148 - 161, 2023.
doi:10.14778/3626292.3626298

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/XinyuZeng/EvaluationOfColumnarFormats.

1 INTRODUCTION

Columnar storage has been widely adopted for data analytics because of its advantages, such as irrelevant attribute skipping, efficient data compression, and vectorized query processing [55, 59, 68]. In the early 2010s, organizations developed data processing engines for the open-source big data ecosystem [12], including Hive [13, 105], Impala [16], Spark [20, 113], and Presto [19, 98], to respond to the petabytes of data generated per day and the growing demand for large-scale data analytics. To facilitate data sharing across the various Hadoop-based query engines, vendors proposed open-source columnar storage formats [11, 17, 18, 76], represented by Parquet and ORC, that have become the *de facto* standard for data storage in today's data warehouses and data lakes [14, 15, 19, 20, 29, 38, 61].

These formats, however, were developed more than a decade ago. The hardware landscape has changed since then: persistent storage performance has improved by orders of magnitude, achieving gigabytes per second [48]. Meanwhile, the rise of data lakes means more column-oriented files reside in cheap cloud storage (e.g., AWS S3 [7], Azure Blob Storage [24], Google Cloud Storage [33]), which exhibits both high bandwidth and high latency. On the software side, a number of new lightweight compression schemes [57, 65, 87, 116], as well as indexing and filtering techniques [77, 86, 101, 115], have been proposed in academia, while existing open columnar formats are based on DBMS methods from the 2000s [56].

Prior studies on storage formats focus on measuring the end-to-end performance of Hadoop-based query engines [72, 80]. They fail to analyze the design decisions and their trade-offs. Moreover, they use synthetic workloads that do not consider skewed data distributions observed in the real world [109]. Such data sets are less suitable for storage format benchmarking.

The goal of this paper is to analyze common columnar file formats and to identify design considerations to provide insights for developing next-generation column-oriented storage formats. We created a benchmark with predefined workloads whose configurations were extracted from a collection of real-world data sets. We then performed a comprehensive analysis for the major components in Parquet and ORC, including encodings, block compression, metadata organization, indexing and filtering, and nested data modeling. In particular, we investigated how efficiently the columnar formats support common machine learning workloads and whether their designs are friendly to GPUs. We detail the lessons learned in Section 6 and summarize our main findings below.

First, there is no clear winner between Parquet and ORC in format efficiency. Parquet has a slight file size advantage because of its aggressive dictionary encoding. Parquet also has faster column decoding due to its simpler integer encoding algorithms, while ORC is more effective in selection pruning due to the finer granularity of its zone maps (a type of sparse index).

Second, most columns in real-world data sets have a small number of distinct values (or low "NDV ratios" defined in Section 4.1),

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 2 ISSN 2150-8097.
doi:10.14778/3626292.3626298

*Huanchen Zhang is also affiliated with Shanghai Qi Zhi Institute.

148

EXECUTION ENGINES

Standalone libraries for executing vectorized query operators on columnar data.

→ Input is a DAG of physical operators.
→ Require external scheduling and orchestration.

Notable implementations:

→ Velox
→ DataFusion
→ Intel OAP

CONCLUSION

The cloud has made the distributed OLAP DBMS market flourish. Lots of vendors. Lots of money.

But more money, more data, more problems…

NEXT CLASS

Final Review

15-721 in a single lecture!