Intro to Database Systems (15-445/645)

23 Distributed OLAP Databases

Carnegie Mellon University

FALL 2022

Andy Pavlo

ADMINISTRIVIA

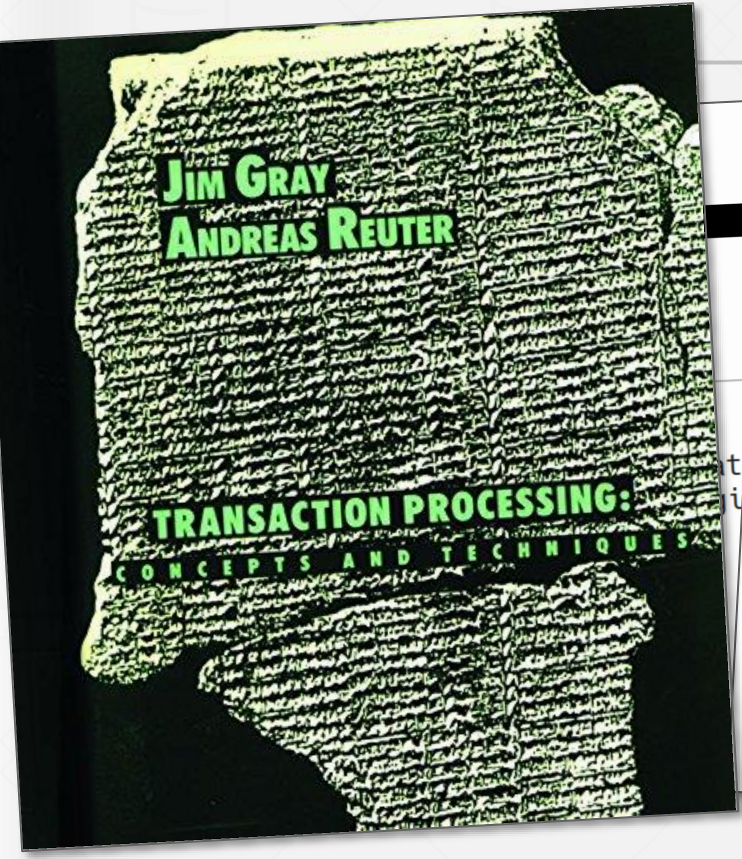
Homework #5 is due **Sunday Dec 4th @ 11:59pm**

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

Upcoming Special Lectures:
→ **Virtual Snowflake Lecture** (Tuesday Dec 6th)
→ **Live Call-in Q&A Lecture** (Thursday Dec 8th)

Final Exam is **Friday Dec 16th @ 1:00pm.**

EMAILS



history of 2PC

From: Bob Devine [redacted]
To: Andy Pavlo <pavlo@cs.cmu.edu>
Date: 11/24/22 12:51 PM

Hi Prof Pavlo,

In a recent video of your DB class, you mentioned that the origin of 2PC was fuzzy.

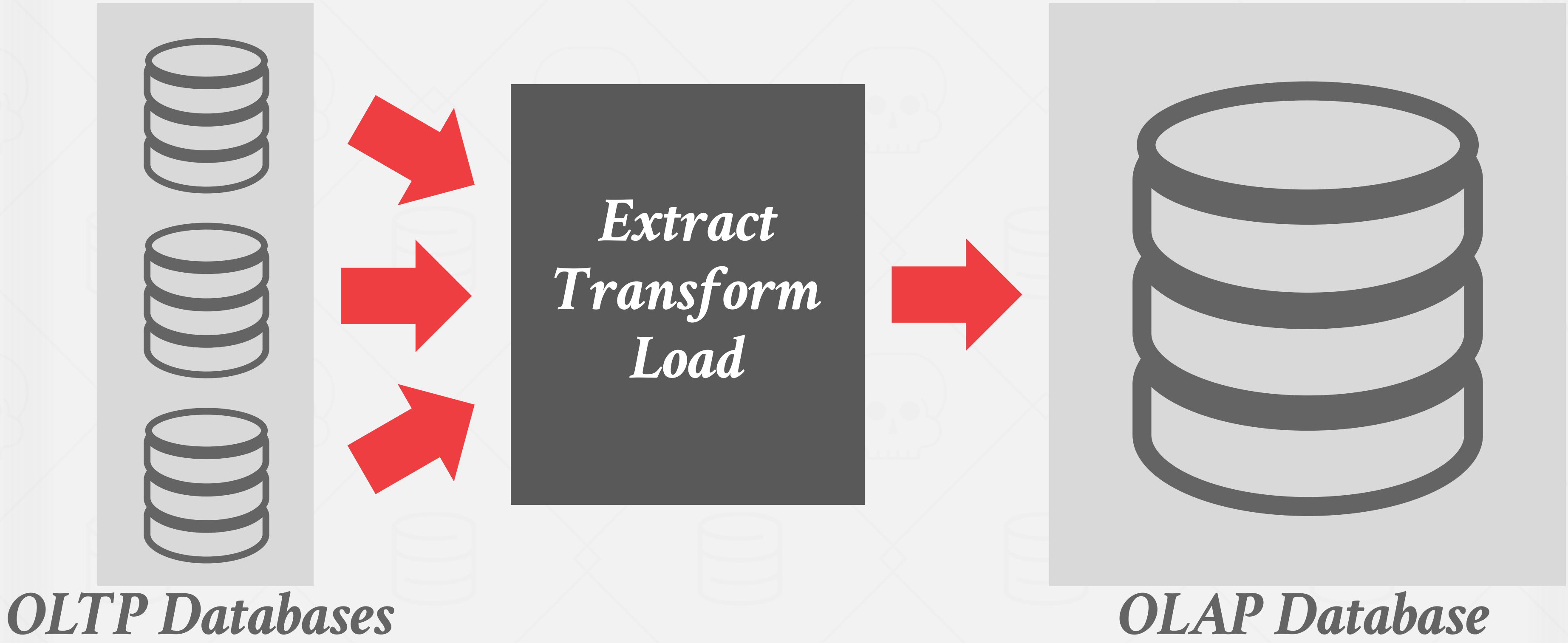
I had worked with Jim Gray in the 1990s. I vaguely remembered his answer to one of my idle questions about 2PC during a lunchtime chat.

Here's his description (from his book with Andreas Reuter, p.575):

> The two-phase commit protocol is just contract law applied to computers, so it is difficult to claim any one individual invented it. The first known instance of its use in distributed systems is credited to Nico Garzado in implementing the Italian social security system in the early 1970s. By the mid-1970s, it had been fairly well analyzed and had been named.

\- Bob Devine

BIFURCATED ENVIRONMENT

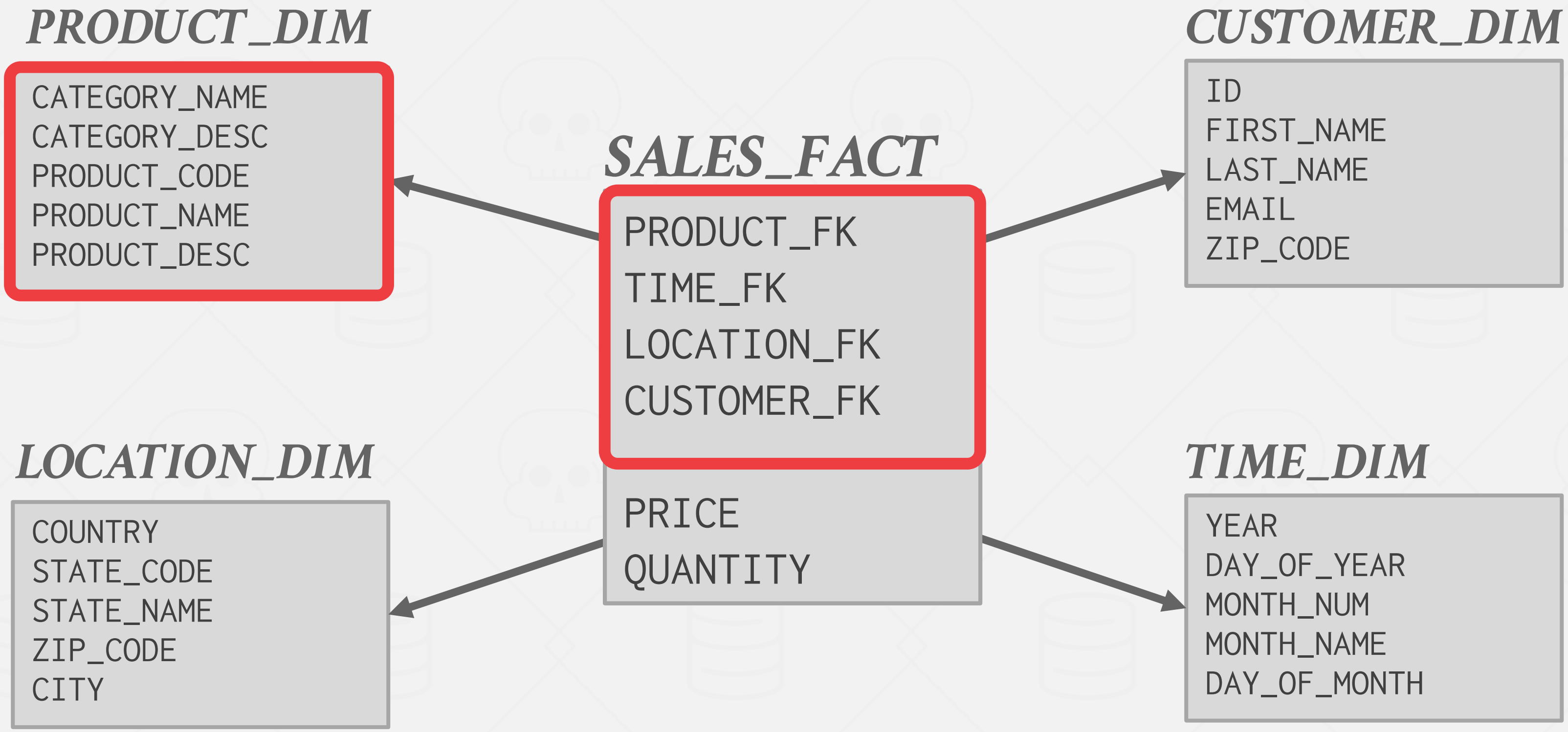


DECISION SUPPORT SYSTEMS

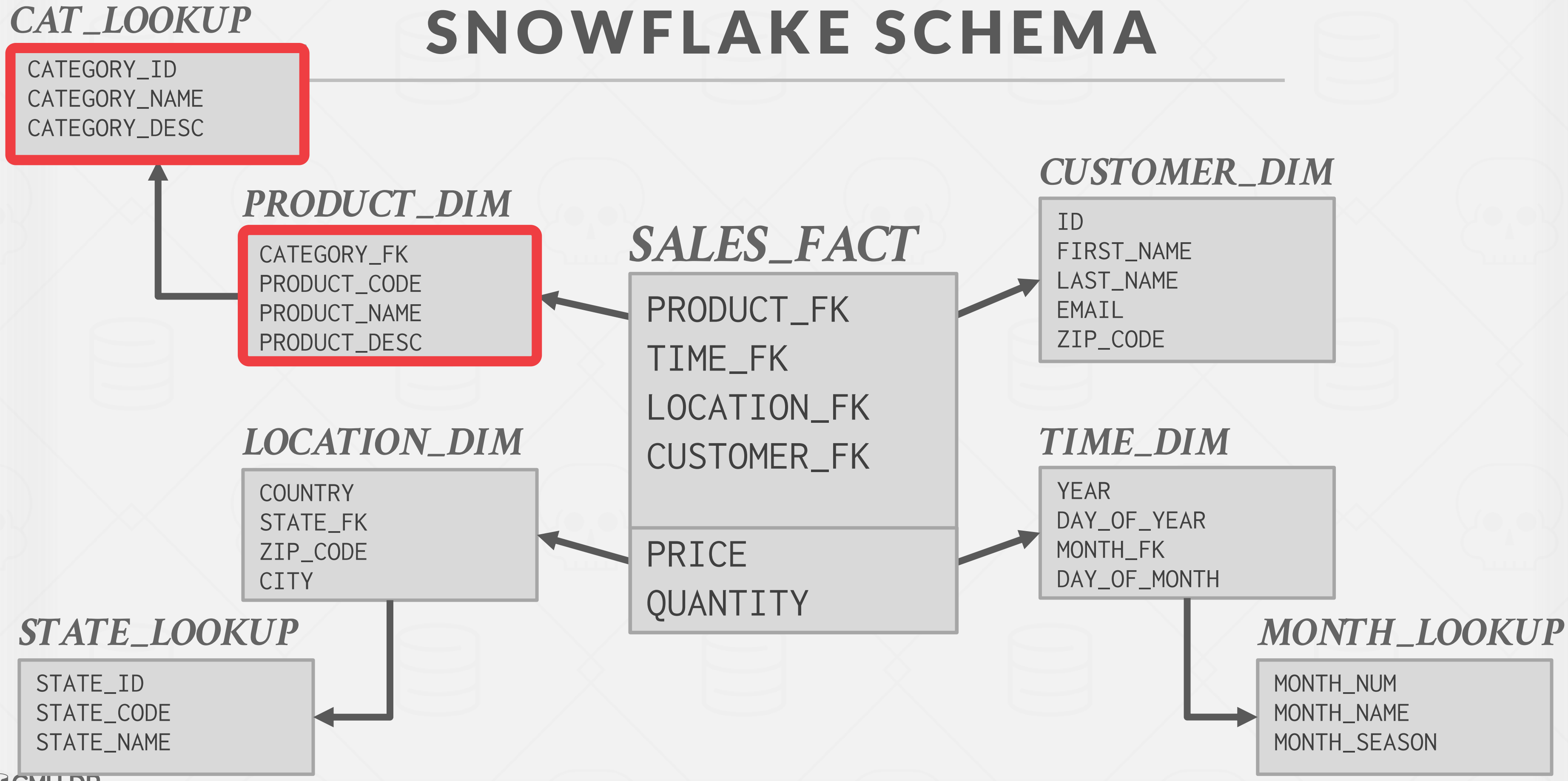
Applications that serve the management, operations, and planning levels of an organization to help people make decisions about future issues and problems by analyzing historical data.

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

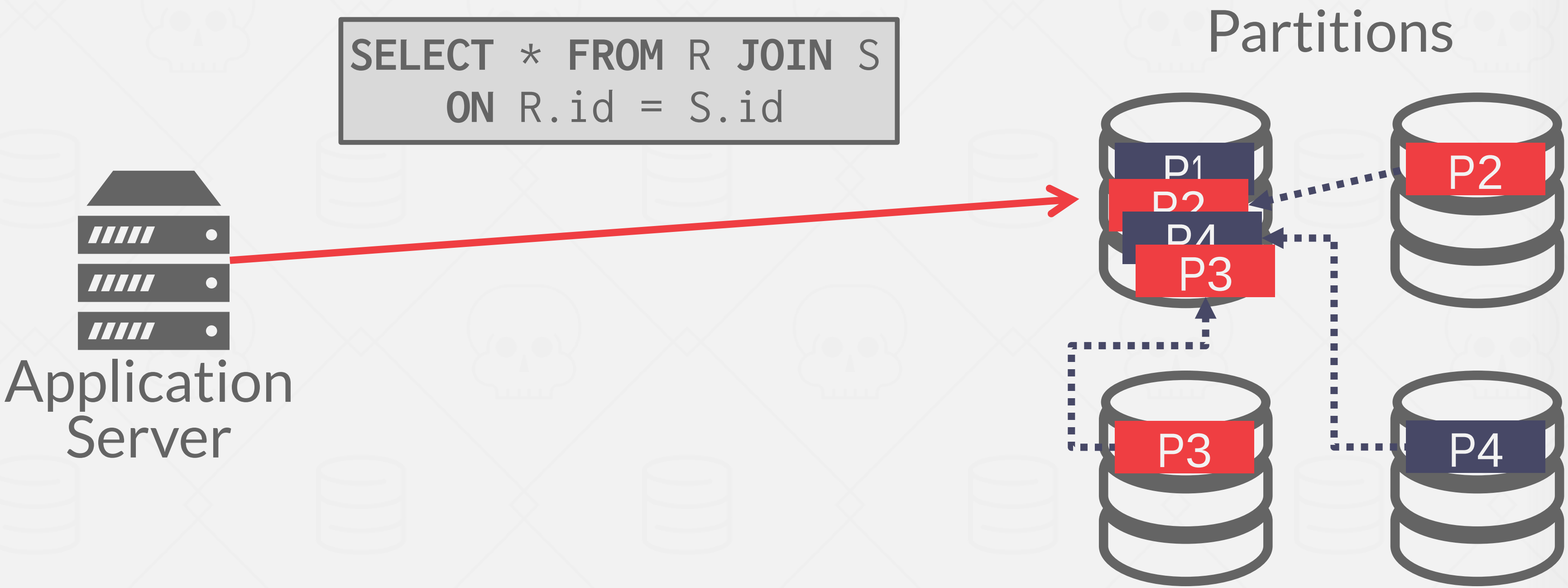
→ Snowflake schemas take up less storage space.
→ Denormalized data models may incur integrity and consistency violations.

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
```



TODAY'S AGENDA

Execution Models
Query Planning
Distributed Join Algorithms
Cloud Systems

PUSH VS. PULL

Approach #1: Push Query to Data

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

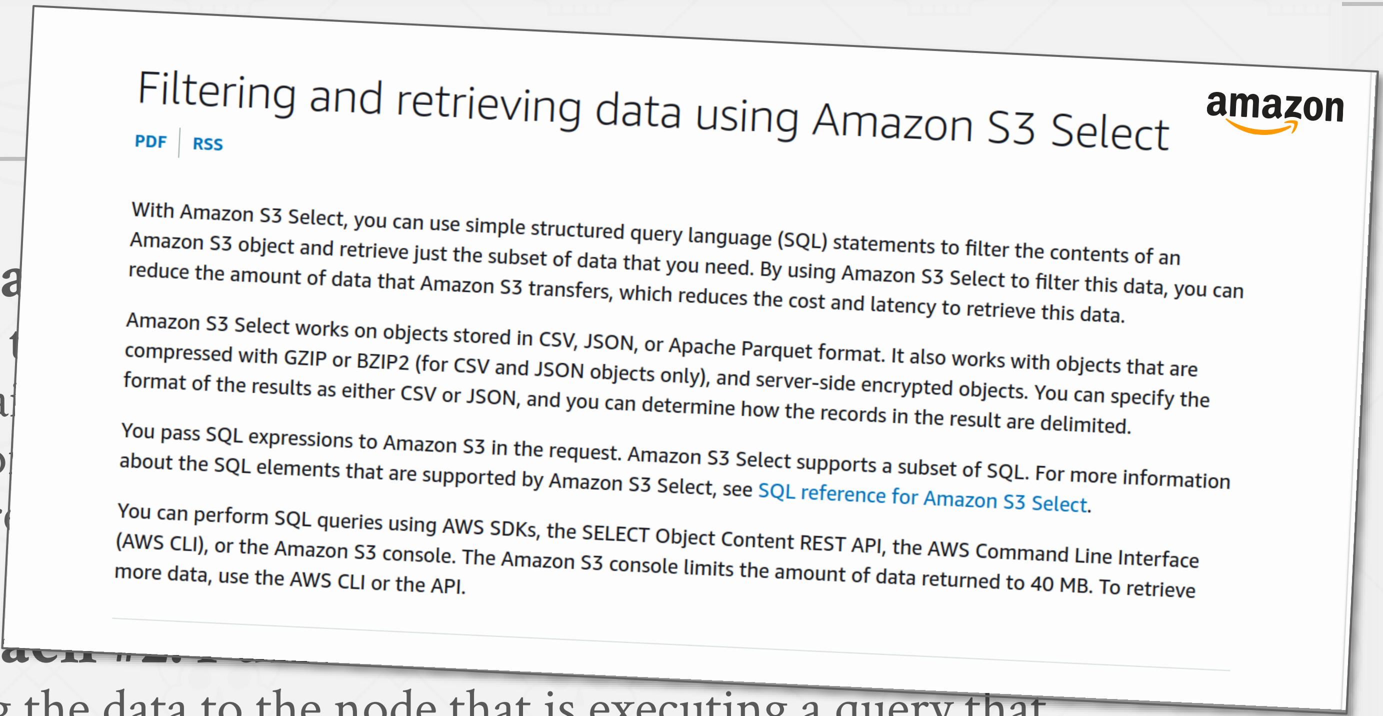
→ Perform as much filtering and processing as possible where data resides before transmitting over network.

Approach #2: Pull Data to Query

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

Approa

→ Send t
conta
→ Perfo
where



Filtering and retrieving data using Amazon S3 Select

PDF | RSS

amazon

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.

Approach #2: P

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

Filtering and retrieving data using Amazon S3 Select

amazon

PDF | RSS

With Amazon S3 Select, you

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.

r 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

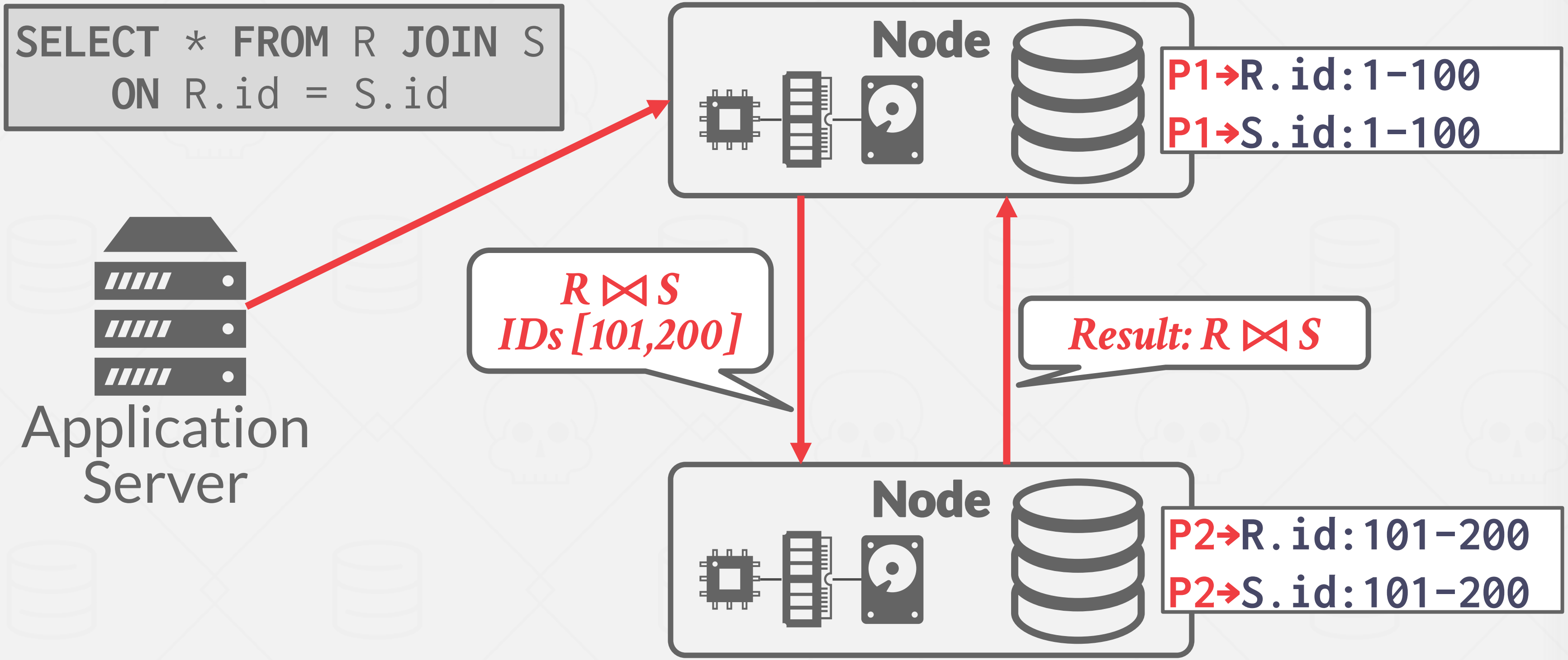
The `Query Blob Contents` request may be constructed as follows. HTTPS is recommended. Replace *myaccount* with the name of your storage account:

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>`	

cuting a query that

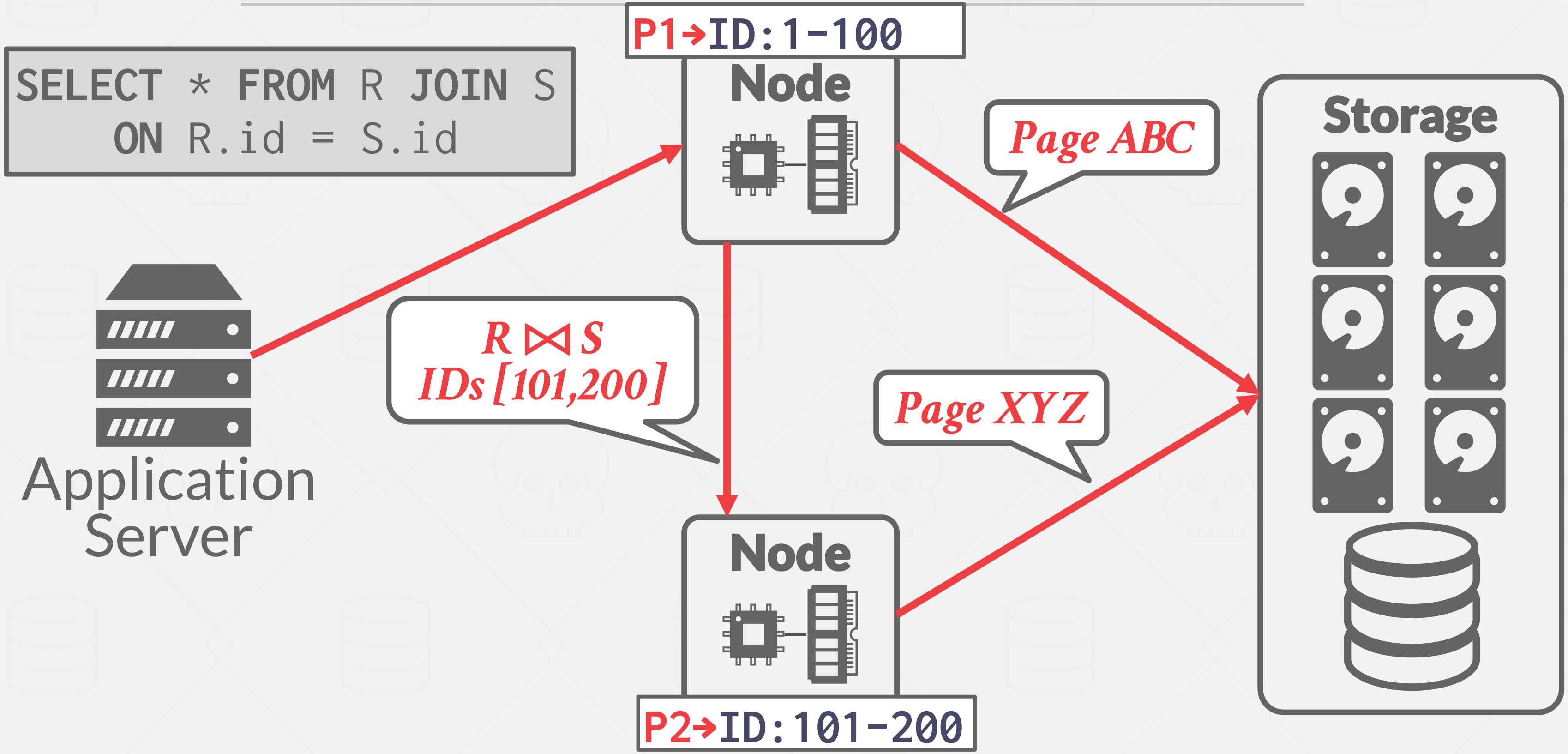
PUSH QUERY TO DATA

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

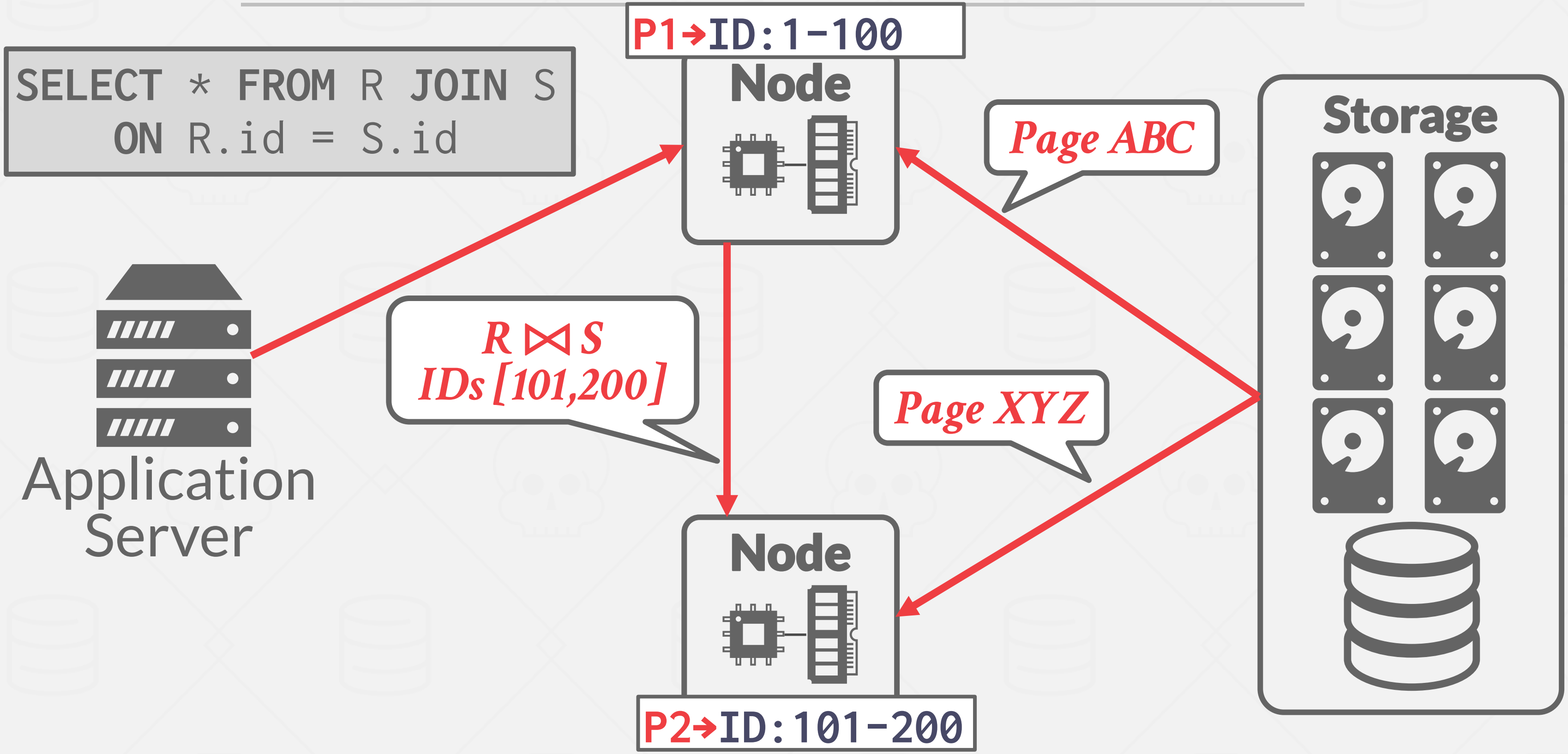


PULL DATA TO QUERY

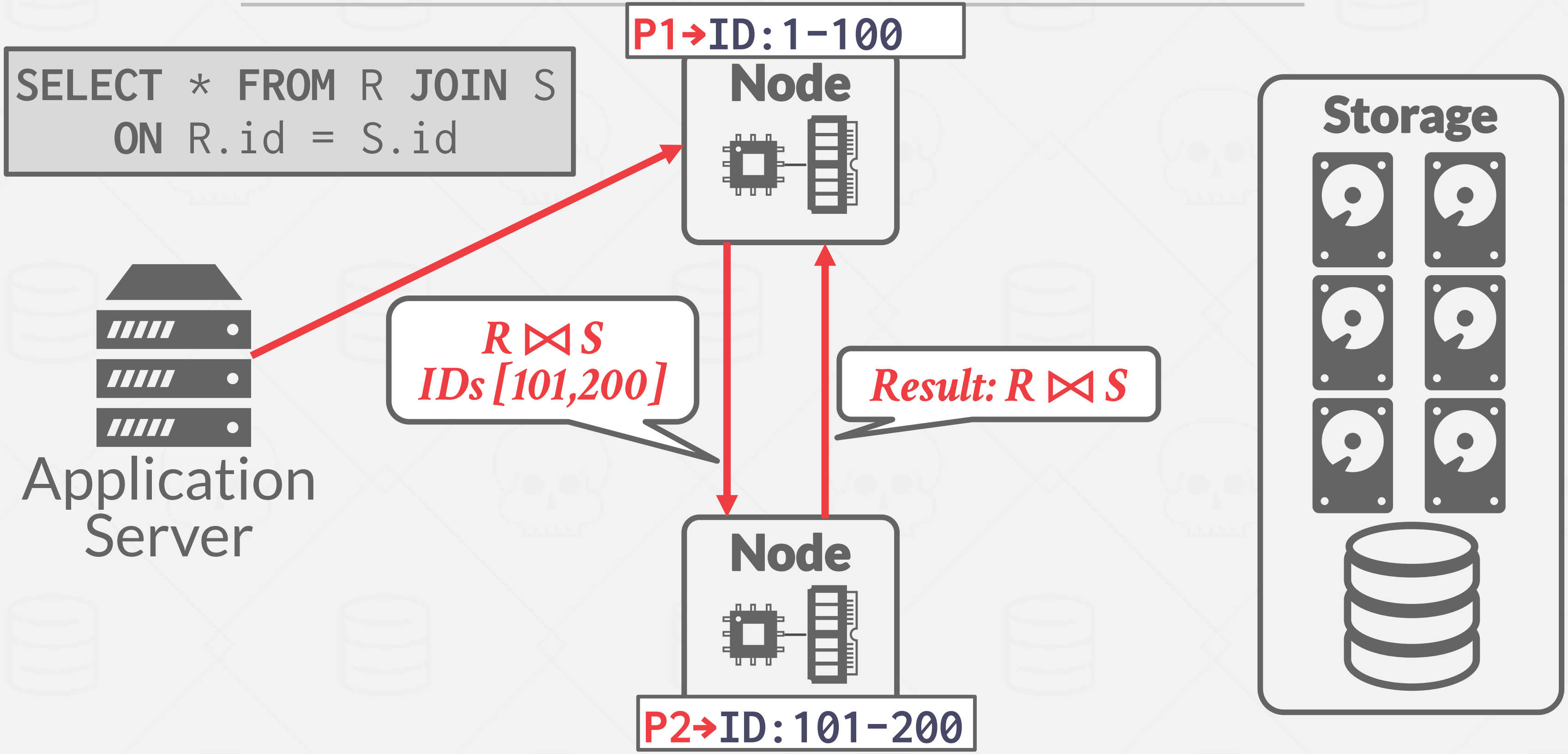
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



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

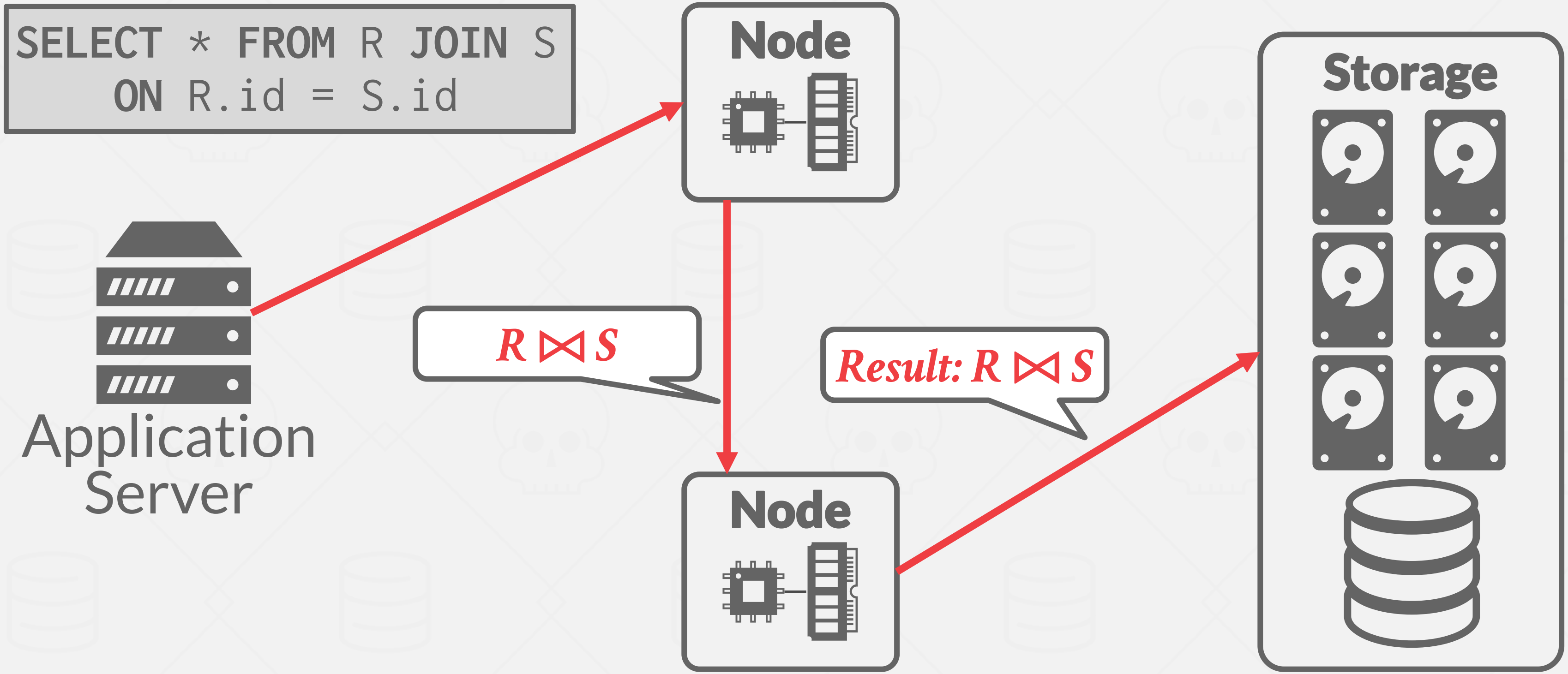
Most shared-nothing distributed OLAP DBMSs are designed to assume that nodes do not fail during query execution.

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

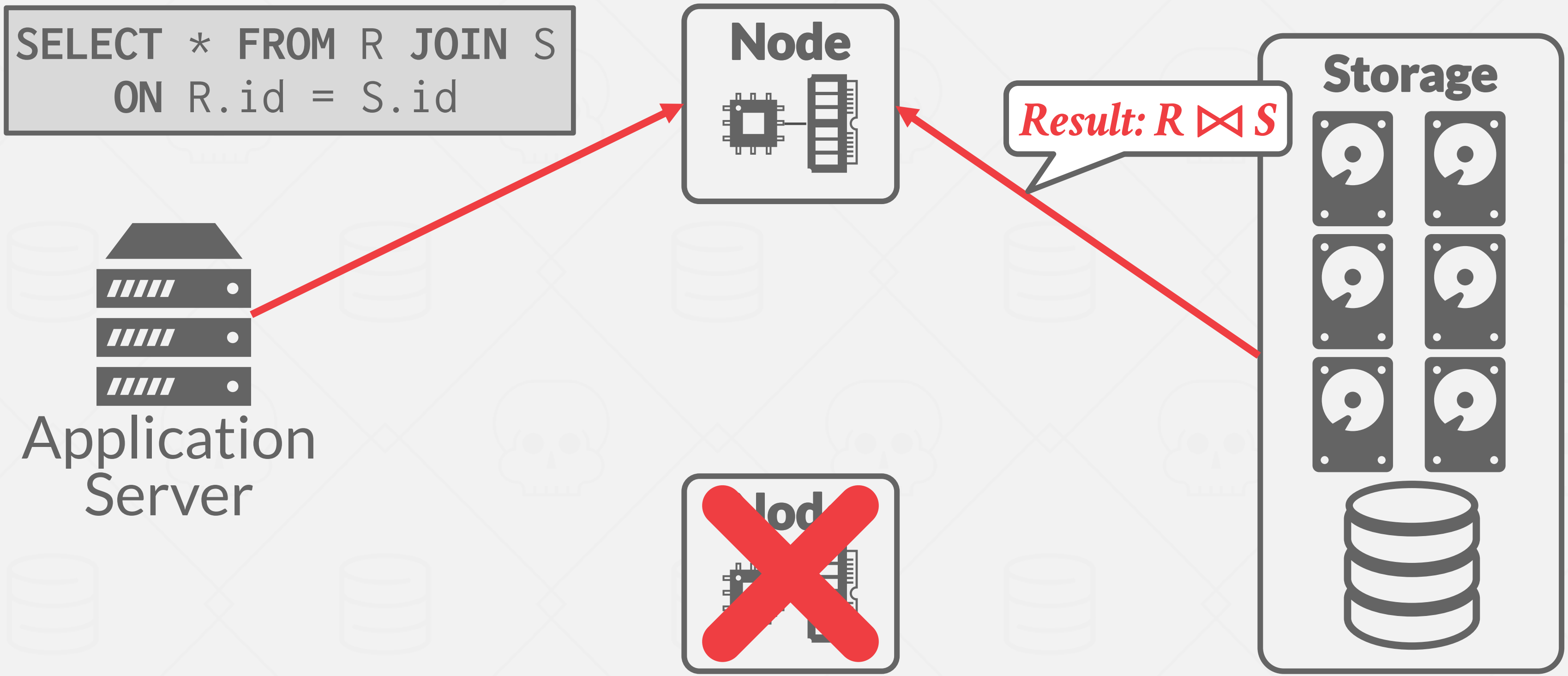
The DBMS could take a snapshot of the intermediate results for a query during execution to allow it to recover if nodes fail.

QUERY FAULT TOLERANCE

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



QUERY FAULT TOLERANCE



QUERY PLANNING

All the optimizations that we talked about before are still applicable in a distributed environment.

→ Predicate Pushdown
→ Early Projections
→ Optimal Join Orderings

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

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 that I knows about that uses this approach.

N FRAGMENTS

Union the output of each join to produce the final result.

```
ROM 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
```

id:1-100

id:101-200

id:201-300

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

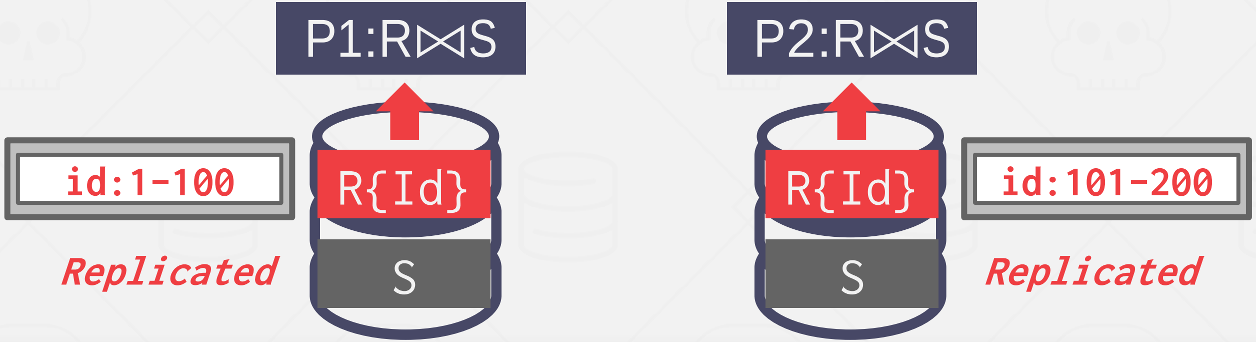
To join tables **R** and **S**, the DBMS needs to get the proper tuples on the same node.

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

SCENARIO #1

One table is replicated at every node. Each node joins its local data in parallel and then sends their results to a coordinating node.

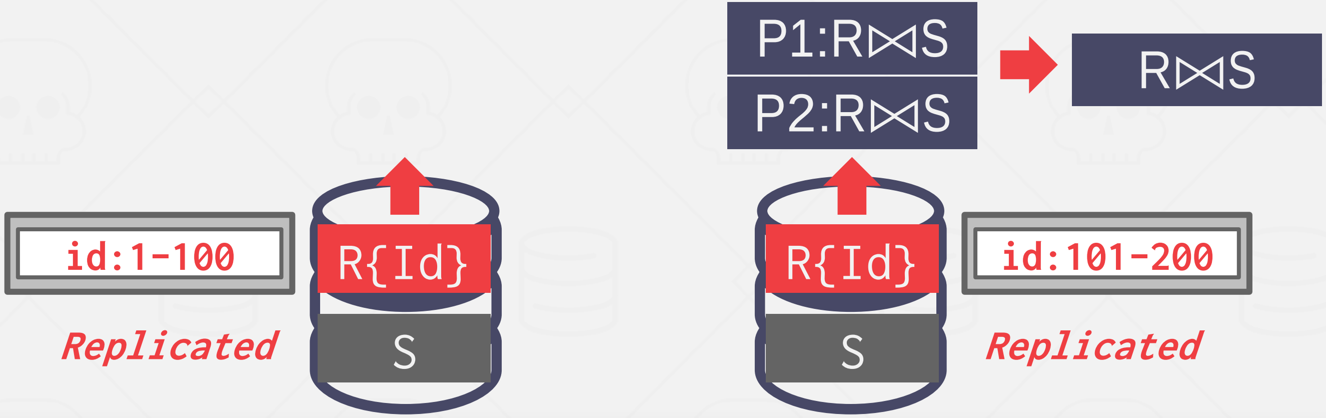
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



SCENARIO #1

One table is replicated at every node. Each node joins its local data in parallel and then sends their results to a coordinating node.

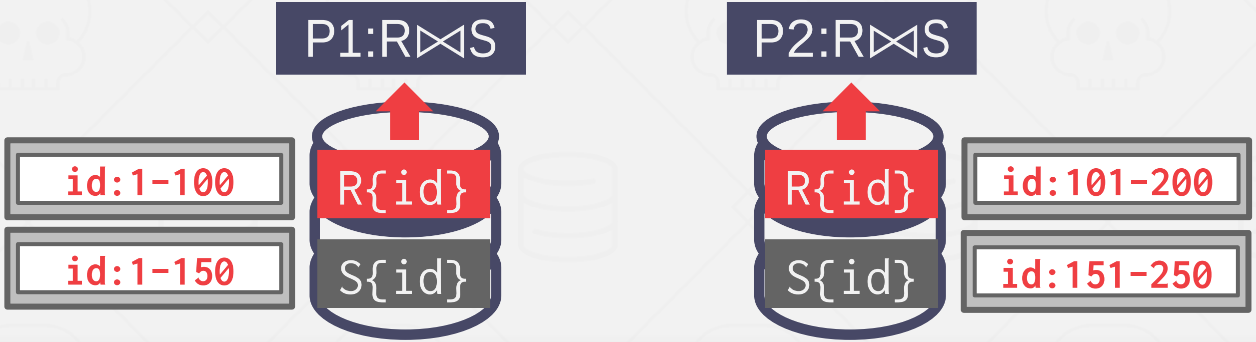
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



SCENARIO #2

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

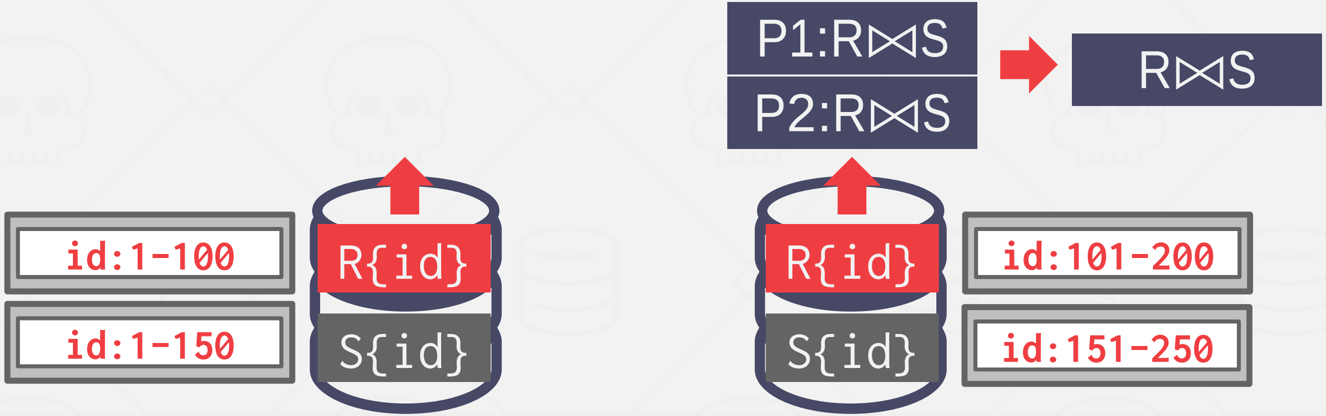
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



SCENARIO #2

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

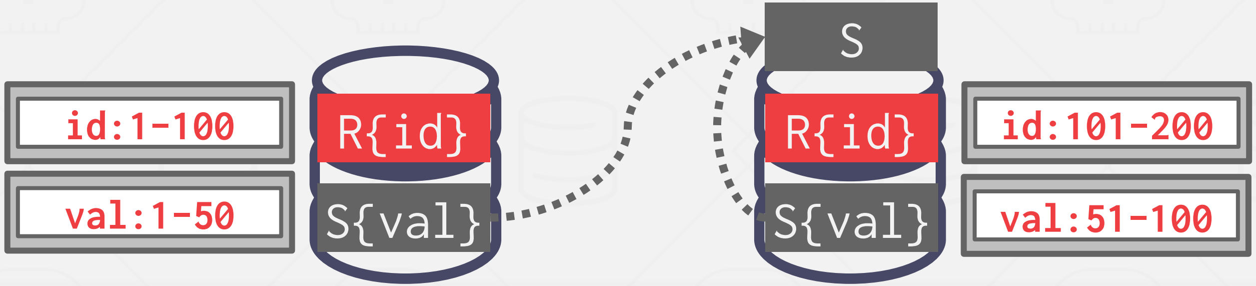
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



SCENARIO #3

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

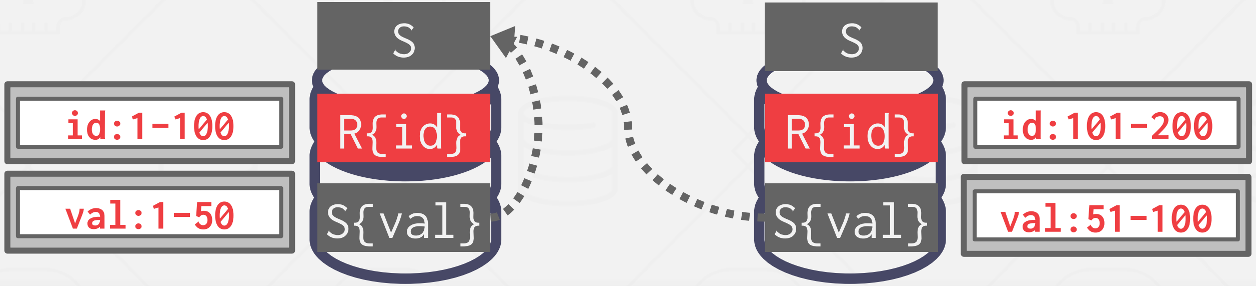
```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #3

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

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



SCENARIO #3

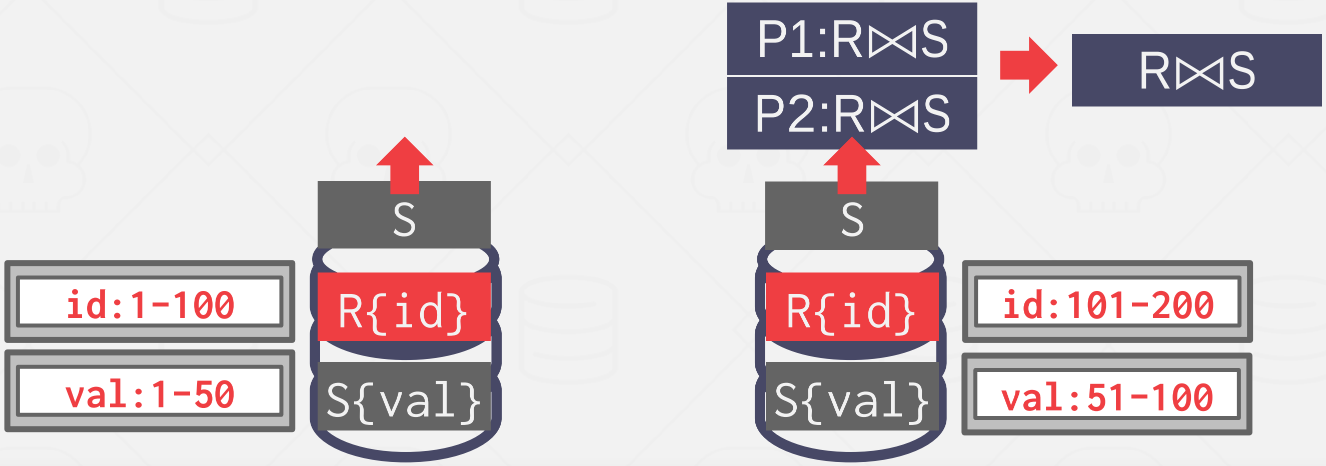
Both tables are partitioned on different keys. If one of the tables is small, then the DBMS "broadcasts" that table to all nodes.

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

SCENARIO #3

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

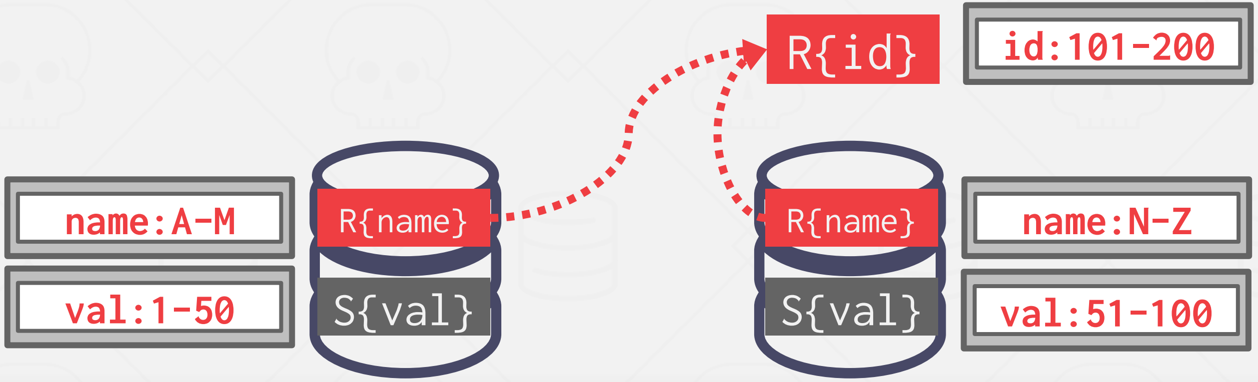
```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by "shuffling" them across nodes.

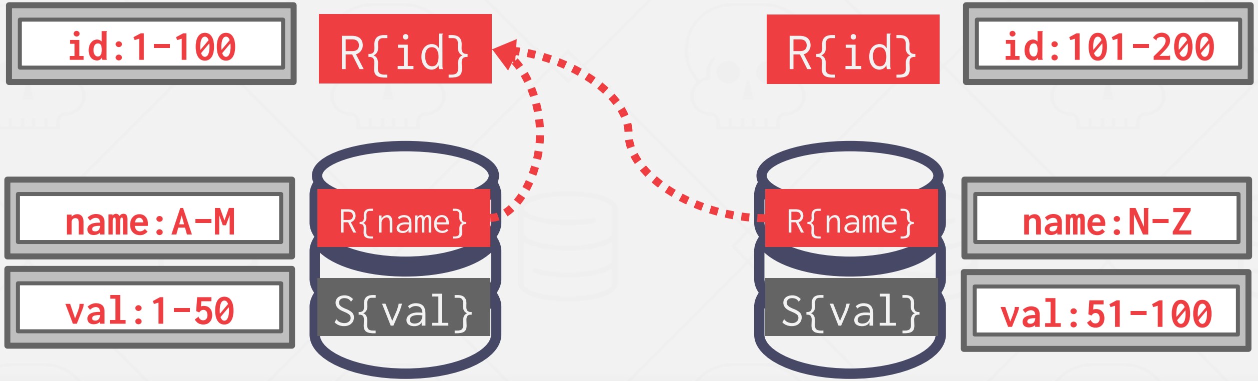
```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by "shuffling" them across nodes.

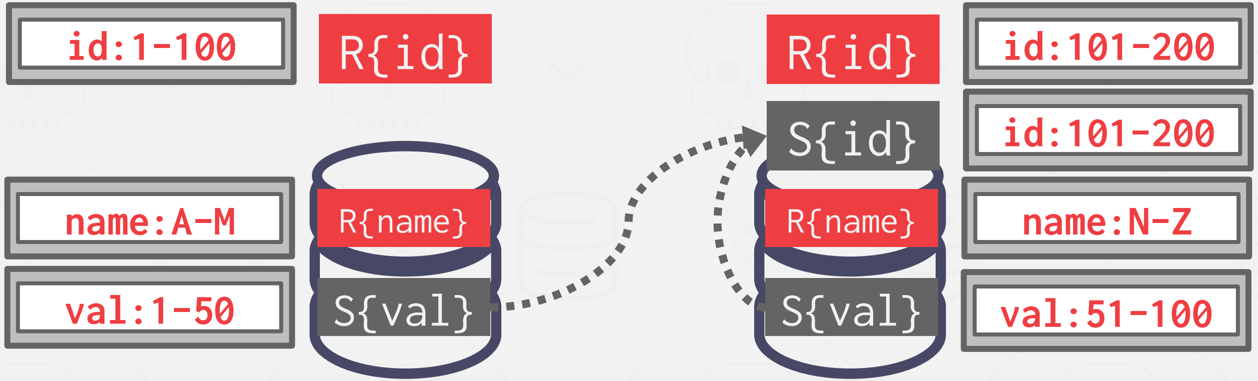
```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by "shuffling" them across nodes.

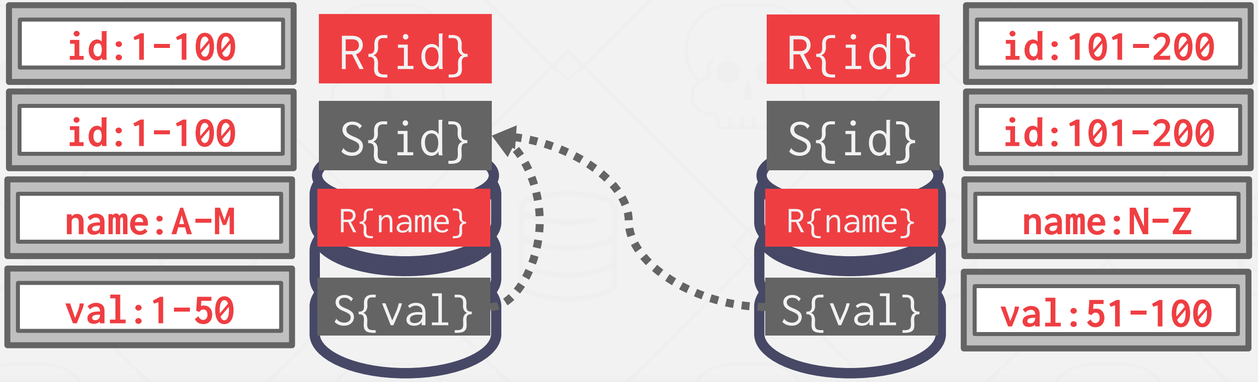
```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by "shuffling" them across nodes.

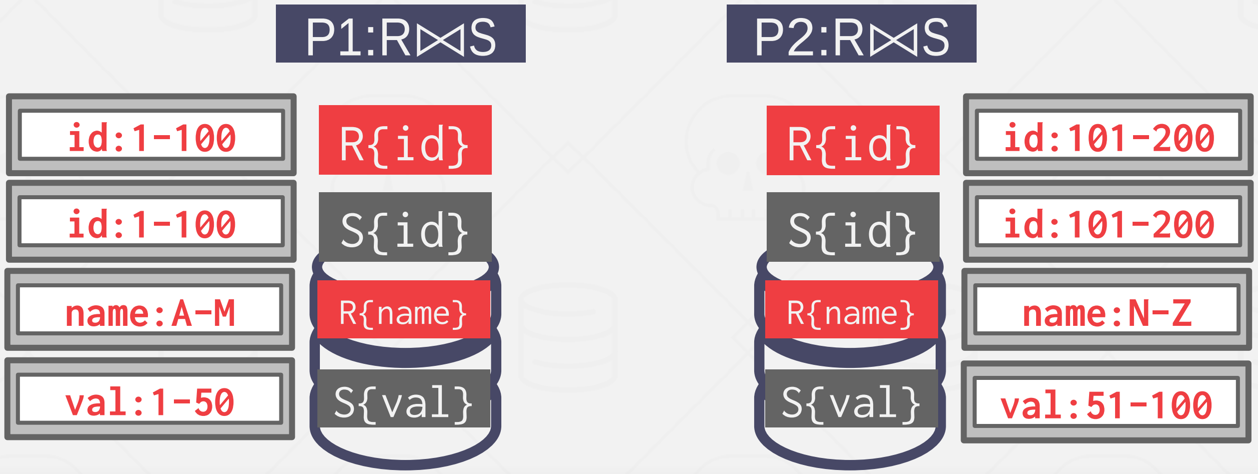
```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by "shuffling" them across nodes.

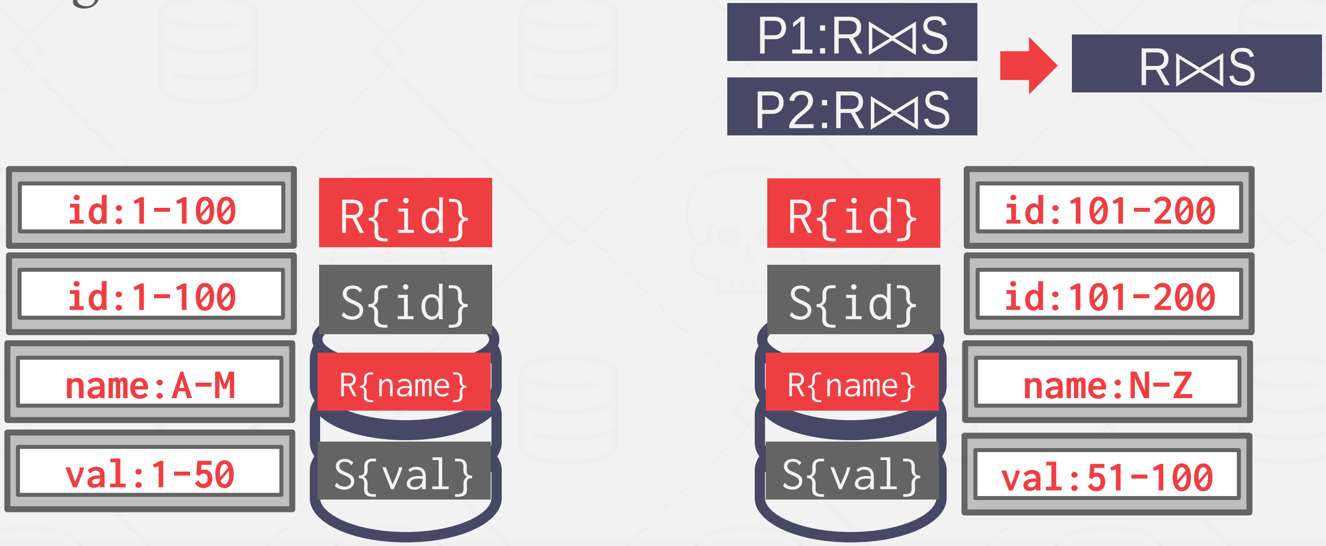
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



SCENARIO #4

Both tables are <u>not</u> partitioned on the join key. The DBMS copies the tables by "shuffling" them across nodes.

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



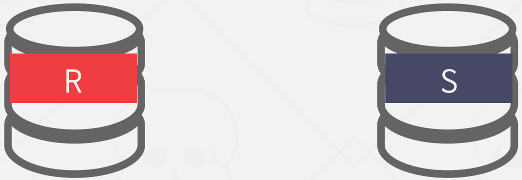
SEMI-JOIN

Join type where the result only contains columns from the left table.

Distributed DBMSs use semi-join to minimize the amount of data sent during joins.
→ This is like a projection pushdown.

Some DBMSs support **SEMI JOIN** SQL syntax. Otherwise you fake it with **EXISTS**.

```
SELECT R.id
  FROM R JOIN S
    ON R.id = S.id
 WHERE R.id IS NOT NULL
```



SEMI-JOIN

Join type where the result only contains columns from the left table.

Distributed DBMSs use semi-join to minimize the amount of data sent during joins.

→ This is like a projection pushdown.

Some DBMSs support **SEMI JOIN** SQL syntax. Otherwise you fake it with **EXISTS**.

```
SELECT R.id
  FROM R JOIN S
    ON R.id = S.id
 WHERE R.id IS NOT NULL
```

SEMI-JOIN

Join type where the result only contains columns from the left table.

Distributed DBMSs use semi-join to minimize the amount of data sent during joins.

→ This is like a projection pushdown.

```
SELECT R.id
  FROM R JOIN S
    ON R.id = S.id
 WHERE R.id IS NOT NULL
```

Some DBMSs support **SEMI JOIN** SQL syntax. Otherwise you fake it with **EXISTS**.

SEMI-JOIN

Join type where the result only contains columns from the left table.

Distributed DBMSs use semi-join to minimize the amount of data sent during joins.
→ This is like a projection pushdown.

Some DBMSs support **SEMI JOIN** SQL syntax. Otherwise you fake it with **EXISTS**.

```
SELECT R.id
  FROM R JOIN S
    ON R.id = S.id
 WHERE R.id IS NOT NULL
```


```
SELECT R.id FROM R
 WHERE EXISTS (
  SELECT 1 FROM S
   WHERE R.id = S.id)
```

SEMI-JOIN

Join type where the result only contains columns from the left table.

Distributed DBMSs use semi-join to minimize the amount of data sent during joins.

→ This is like a projection pushdown.

Some DBMSs support **SEMI JOIN** SQL syntax. Otherwise you fake it with **EXISTS**.

```
SELECT R.id
  FROM R JOIN S
    ON R.id = S.id
 WHERE R.id IS NOT NULL
```


```
SELECT R.id FROM R
 WHERE EXISTS (
  SELECT 1 FROM S
   WHERE R.id = S.id)
```

SEMI-JOIN

Join type where the result only contains columns from the left table.

Distributed DBMSs use semi-join to minimize the amount of data sent during joins.
→ This is like a projection pushdown.

Some DBMSs support **SEMI JOIN** SQL syntax. Otherwise you fake it with **EXISTS**.

```
SELECT R.id
  FROM R JOIN S
    ON R.id = S.id
 WHERE R.id IS NOT NULL
```


```
SELECT R.id FROM R
 WHERE EXISTS (
  SELECT 1 FROM S
   WHERE R.id = S.id)
```

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

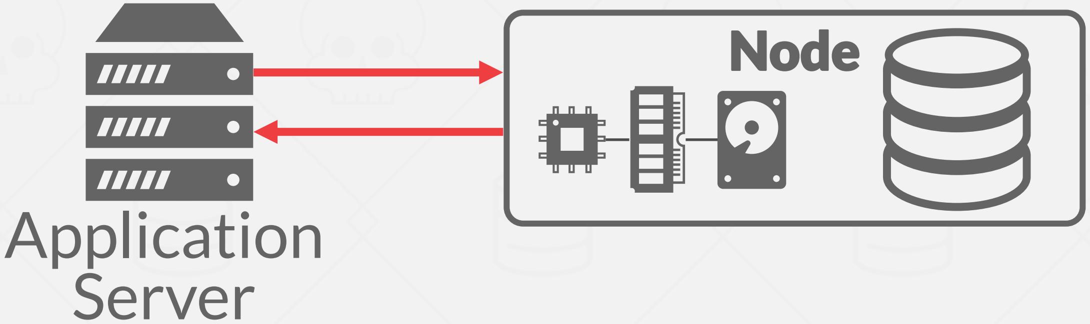
→ No significant modification to the DBMS to be "aware" that it is running in a cloud environment.
→ Examples: Most vendors

Approach #2: Cloud-Native DBMS

→ The system is designed explicitly to run in a cloud environment.
→ Usually based on a shared-disk architecture.
→ Examples: Snowflake, Google BigQuery, Amazon Redshift, Microsoft SQL Azure

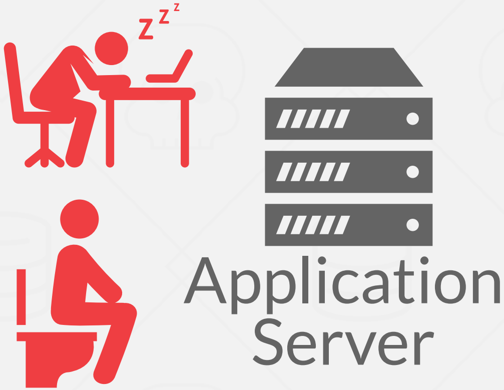
SERVERLESS DATABASES

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



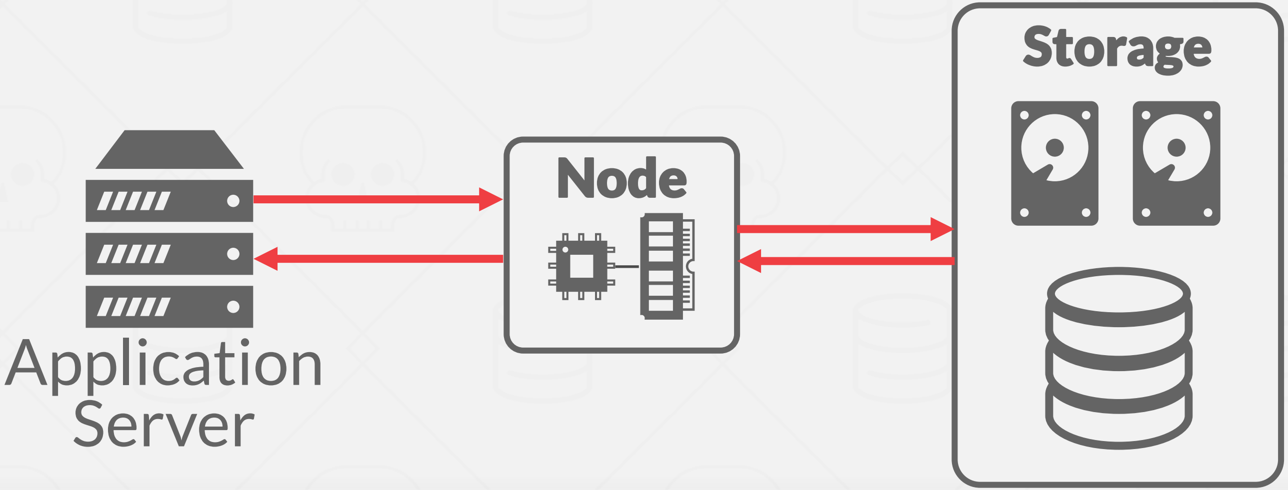
SERVERLESS DATABASES

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



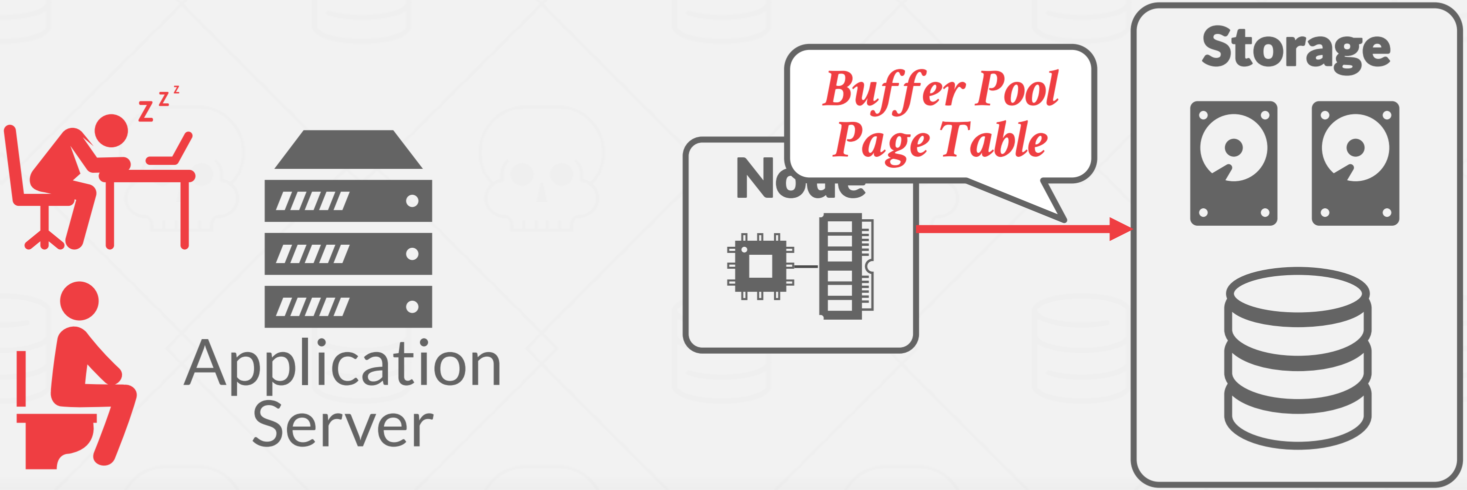
SERVERLESS DATABASES

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



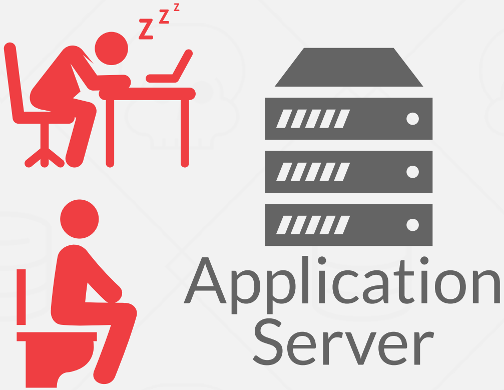
SERVERLESS DATABASES

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



SERVERLESS DATABASES

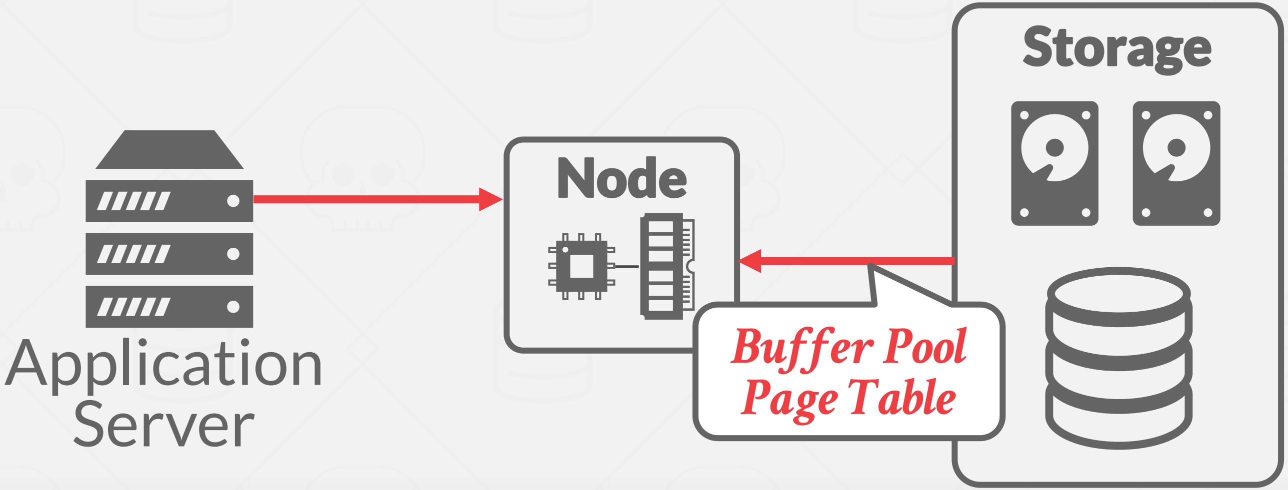
Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



SERVERLESS DATABASES

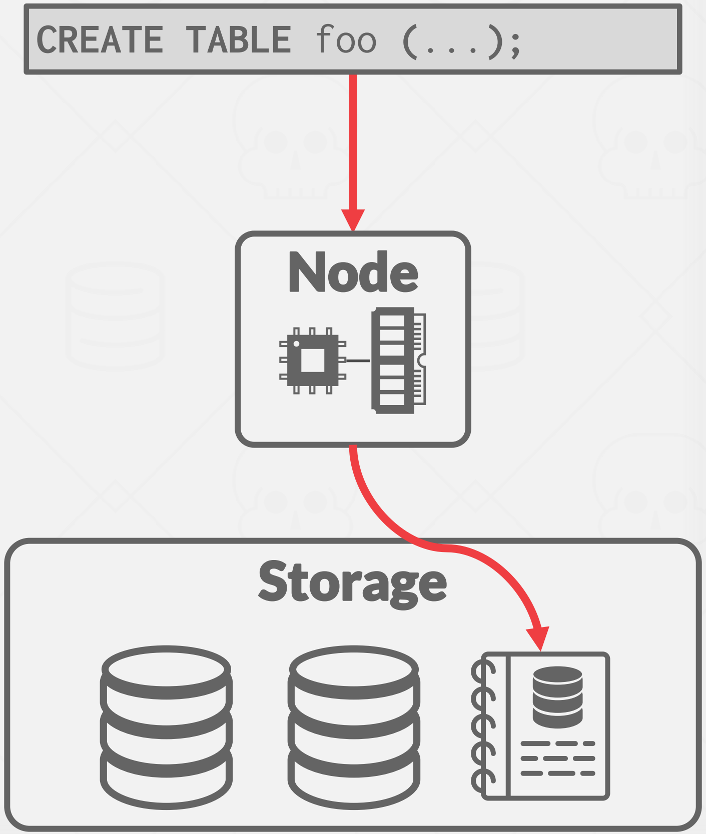
Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.

planetscale
CockroachDB
NEON
amazon
fauna
Microsoft SQL Azure



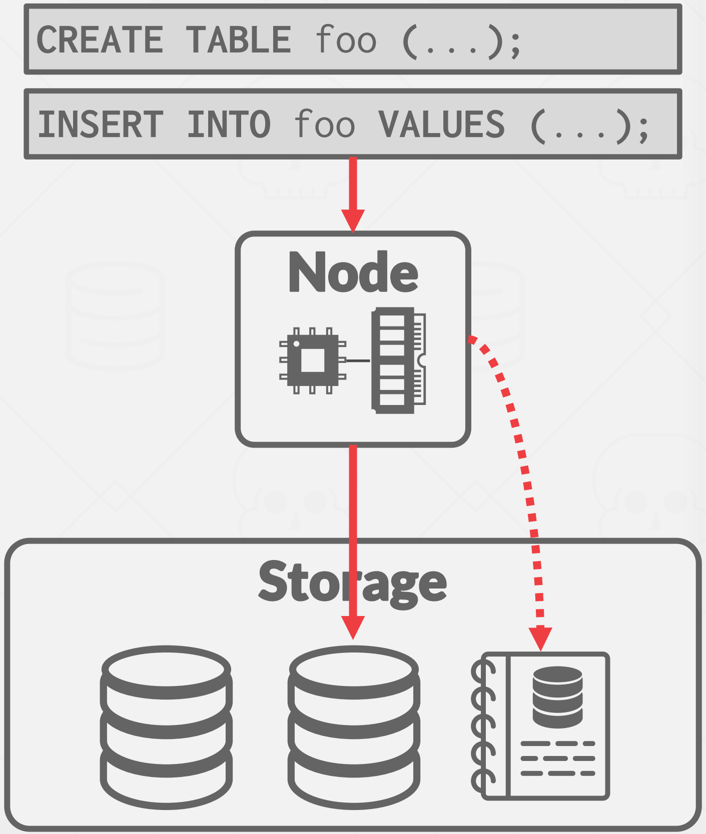
DATA LAKES

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.



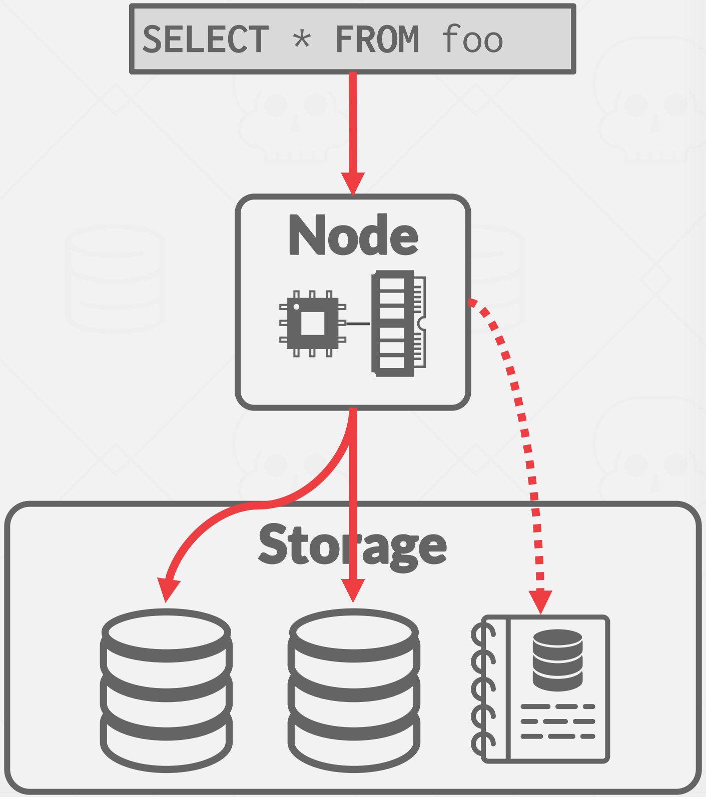
DATA LAKES

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.



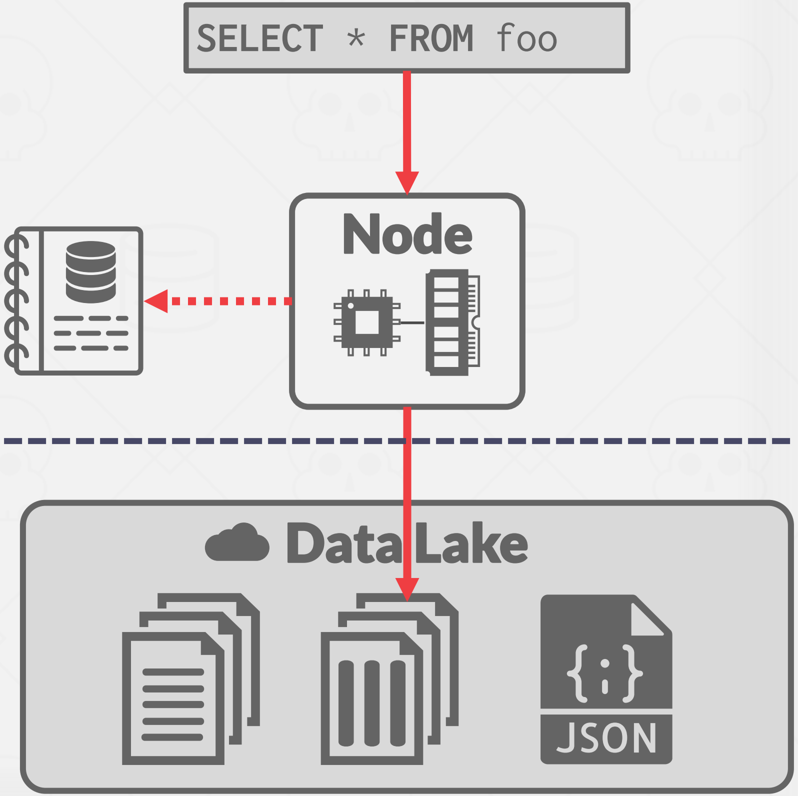
DATA LAKES

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

UNIVERSAL 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.

UNIVERSAL 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.

DISAGGREGATED COMPONENTS

System Catalogs
→ HCatalog, Google Data Catalog, Amazon Glue Data Catalog

Node Management
→ Kubernetes, Apache YARN, Cloud Vendor Tools

Query Optimizers
→ Greenplum Orca, Apache Calcite

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

Andy's potentially frivolous attempt to convince you to put as much application logic as you can into the DBMS but then you will go into the real world and find out that few people do these things.