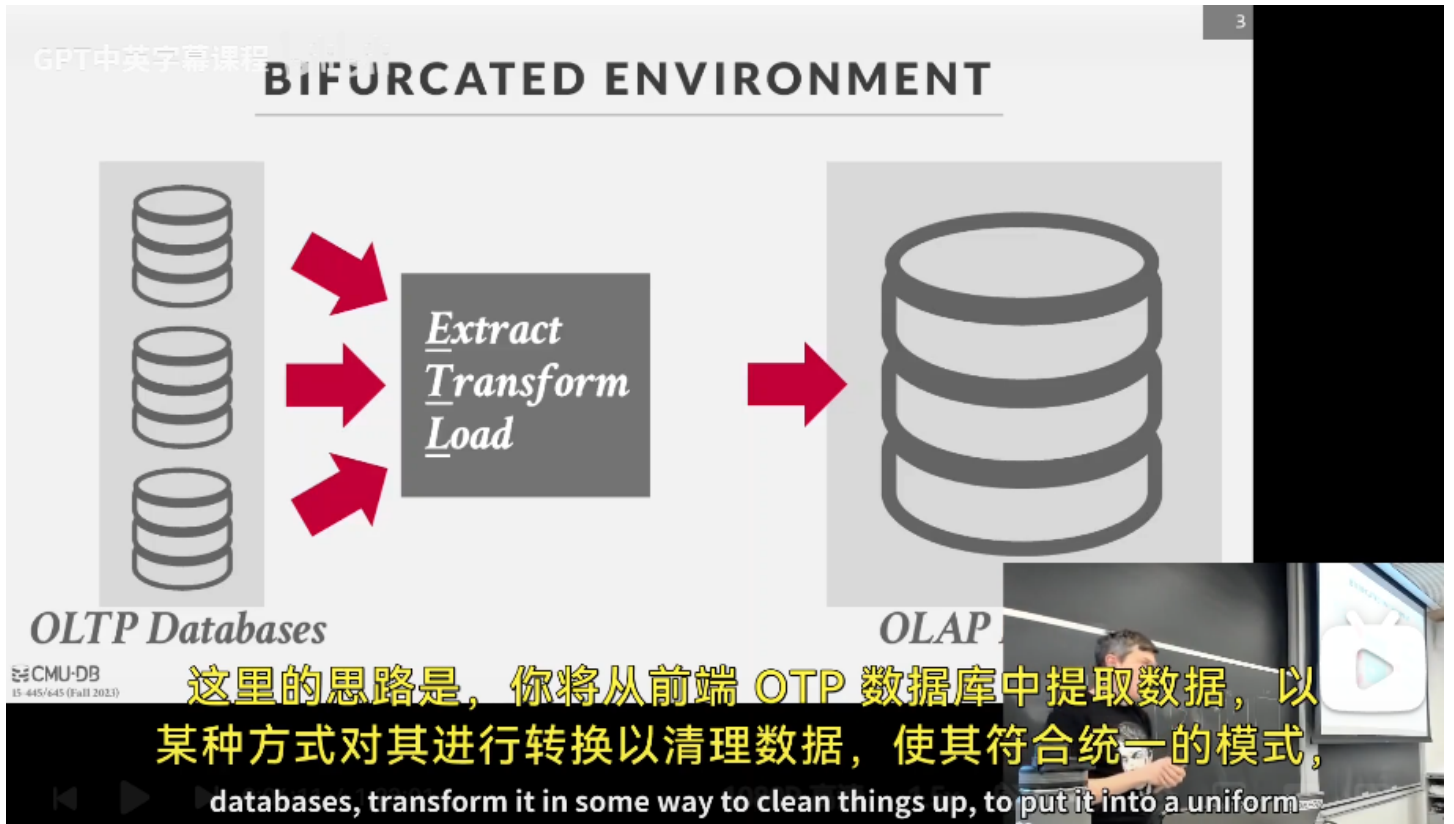


# 分布式OLAP DB

## 背景介绍



## BIFURCATED ENVIRONMENT



您只需将所有原始文件加载到数据仓库中，  
随后转换过程实际上是在数据仓库内部进行的。

process actually really occurs inside the data warehouse itself.

应用程序在前端运行着OLTP DB，需要将其整合到自己的大型OLAP系统中；需要经过抽取转换，因为前端应用程序编写的规范不一样。

这种方式通常作为 决策支持系统，利用前端数据库收集的数据提取新知识

# 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

它们基本上都意味着同一件事，即你试图从已经从前端数据库系统收集的数据中提取新知识。

They all pretty much mean the same thing, that you're trying to extract new knowledge

构建OLAP数据库的两种技术

Star schema (subset of snowflake)

# 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

回到这里的产品类别部分，我不仅追踪产品名称和描述，还包括类别名称和类别描述。

去规范化：无需执行连接操作就能直接获取所有所需的信息；坏处是一旦更新了某个元组的名称，就必须保证更新所有相关的元组。

星型结构是去规范化的

## Snowflake schema

# SNOWFLAKE SCHEMA

CATEGORY_ID
CATEGORY_NAME
CATEGORY_DESC

## PRODUCT\_DIM

CATEGORY_FK
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_FK
ZIP_CODE
CITY

## TIME\_DIM

YEAR
DAY_OF_YEAR
MONTH_FK
DAY_OF_MONTH

## STATE\_LOOKUP

STATE_ID
STATE_CODE
STATE_NAME

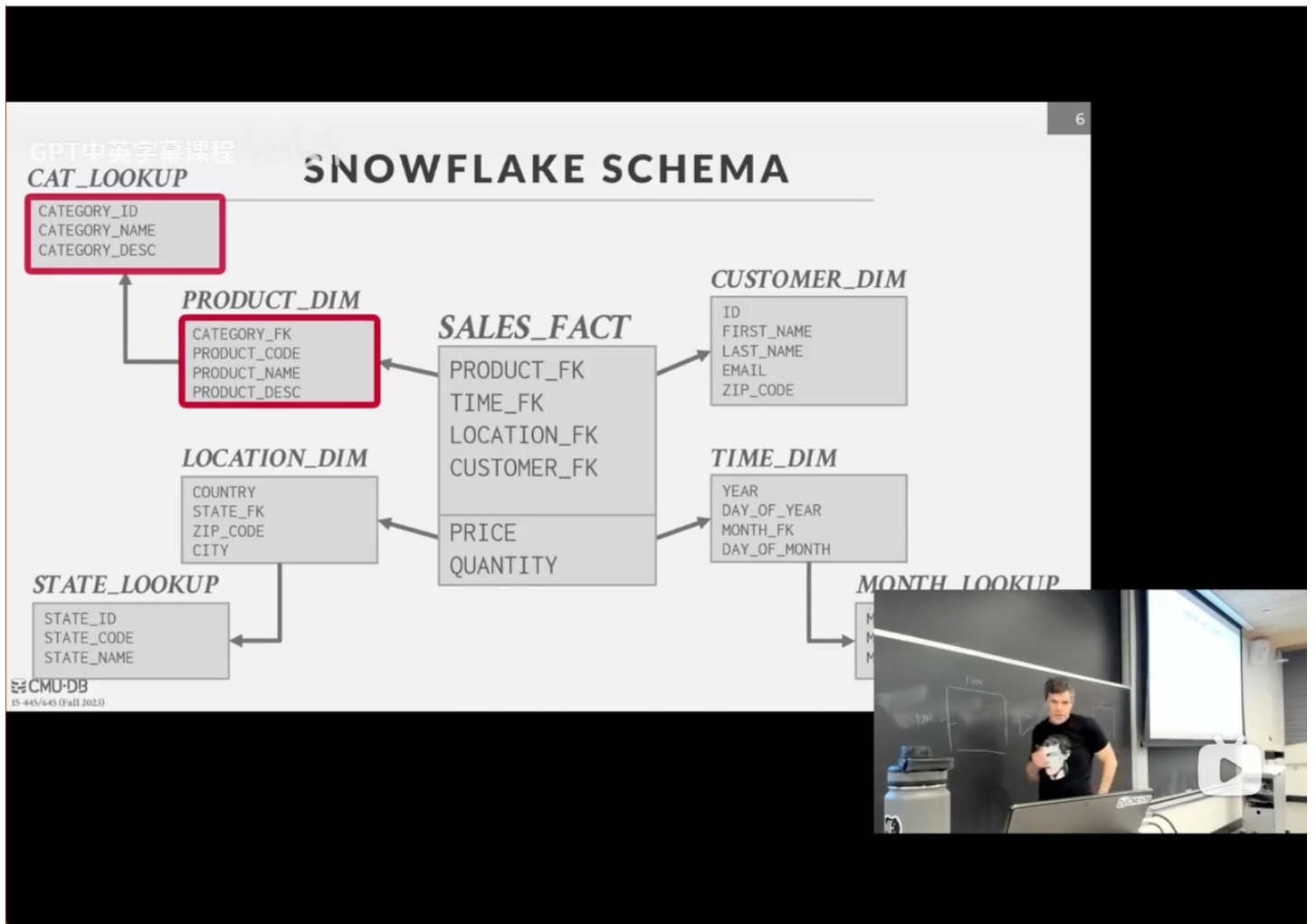
## MONTH\_LOOKUP

MONTH_ID
MONTH_CODE
MONTH_NAME

现在，若采用雪花型模式，我便能将产品类别信息进行规范化处理，从而获得一个独立的类别查询表。

my products so that now I have a separate category lookup table.

获取指定产品的类别只需要执行一次连接操作即可



Star VS. 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.

但在星型模式中，其优势在于运行速度可能会快很多，因为我不需要进行大量的连接操作。

potentially, because I don't have to do a bunch of joins.

我将会出现数据冗余，因为我已经将我的表进行了扁平化处理，或者将多个表合并成了单个表。

But I'm going to have this duplication of data because I've sort of flattened my

雪花模式将需要更多的连接操作。

The snowflake schemas are going to require more joins.

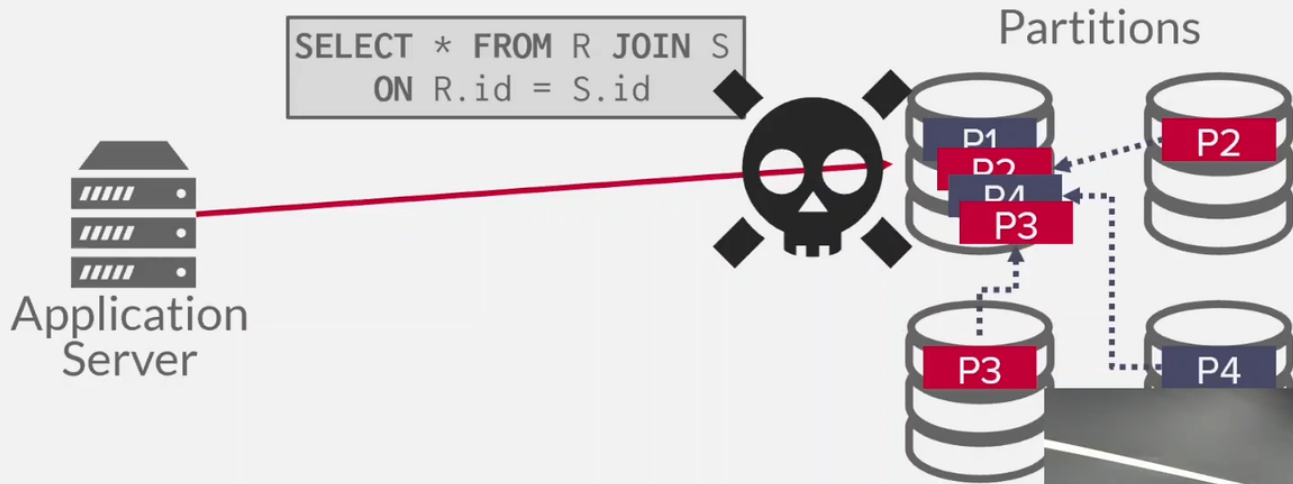
但同样，我拥有数据隔离的优势，或者我正在减少其副本的数量。

But again, I have the advantage that I have that isolation of the data, or I'm

现实世界中使用star的很少了，一般使用snowflake

Problem setup

# PROBLEM SETUP



DB  
Fall 2023

显然，我们希望以一种方式实现跨多个节点的查询，确保不会出现假阳性或假阴性的情况。

nodes in such a way that we don't have any false positives or false negatives.

执行连接操作，让P2\P3\P4把数据都发送给P1，直接执行join，但是这样会变成一个单节点的情况，失去了分布式的优势，这样是不对的

## Agenda

# TODAY'S AGENDA

- Execution Models
- Query Planning
- Distributed Join Algorithms
- Cloud Systems

我们将探讨分布式数据库中执行模型  
的多种可能性，以实现重叠操作。

We're going to talk about the execution models you could have for a distributed

分布式环境下要么是排序hash合并连接，要么是hash连接；大多数分布式数据库都会采用hash连接，因为在多数情况下，数据会被hash分区

## 分布式查询执行

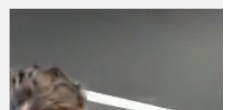
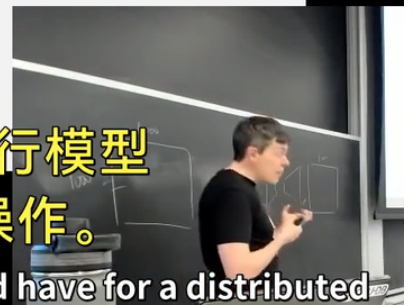
# DISTRIBUTED QUERY EXECUTION

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





# 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):

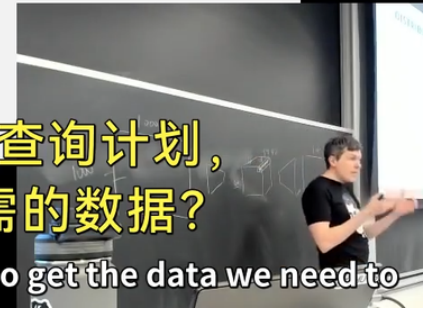
- **Push Query to Data**
- **Pull Data to Query**

那么接下来的问题是，对于给定的查询计划，  
我们如何获取执行这些操作符所需的数据？

Next question is, for a given query plan, how are we going to get the data we need to

问题2：我计算得到的中间结果存储在哪？

- 当前节点，等别人来获取；直接发送到目标节点
- 写入共享磁盘



# 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.

【数据库系统导论 15-445 2023Fall】CMU—中英字幕

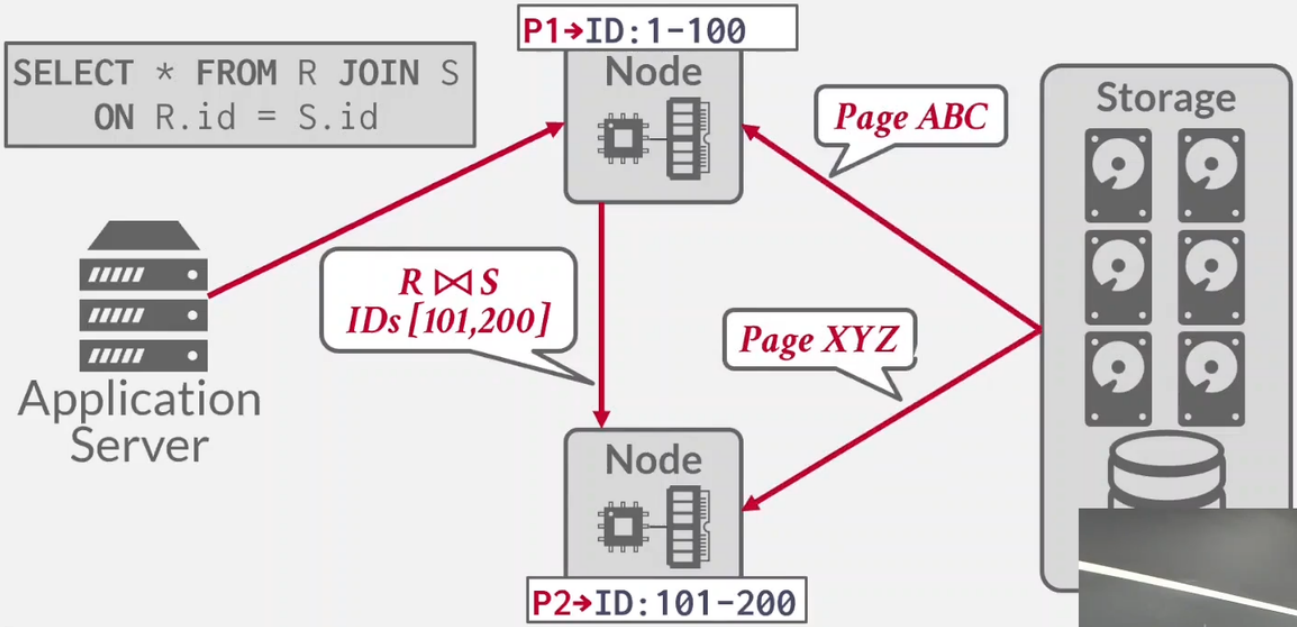
### PUSH QUERY TO DATA

The diagram shows an Application Server sending a query to a top Node. The query is: `SELECT * FROM R JOIN S ON R.id = S.id`. The top Node pushes a query fragment to a bottom Node. The fragment is: `R ⋈ S IDs [101,200]`. The top Node's local storage contains data for `P1 → R.id: 1-100` and `P1 → S.id: 1-100`. The bottom Node's local storage contains data for `P2 → R.id: 101-200` and `P2 → S.`.

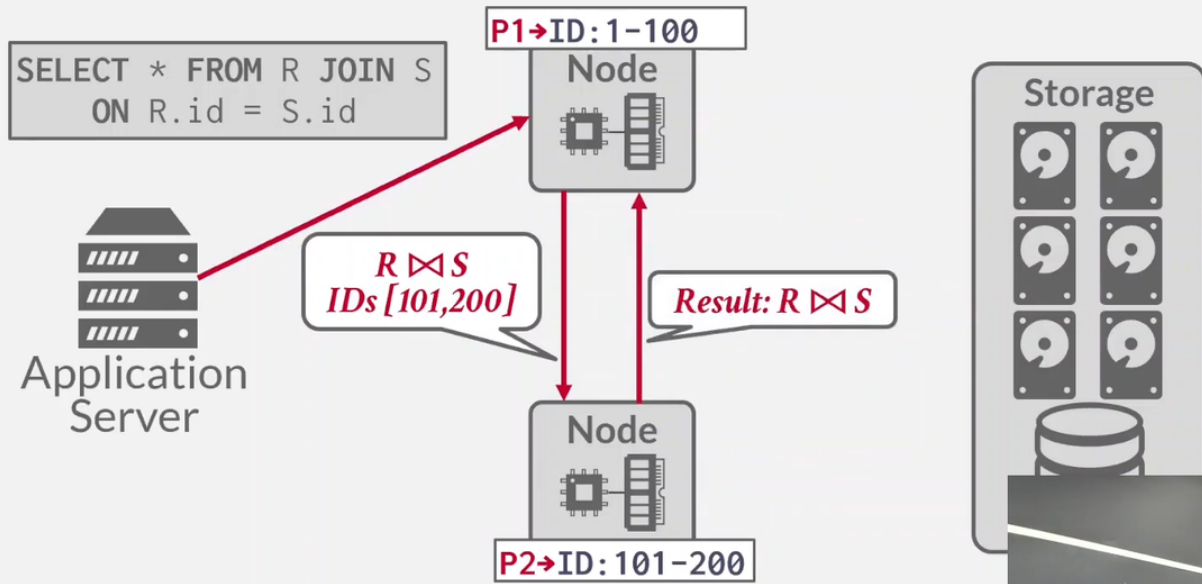
因此，不是顶层节点告诉底层节点：“把你所有的数据都发给我”，而是将查询计划片段下发。

So instead of the top node telling the bottom node, send me all the data you have,

# PULL DATA TO QUERY

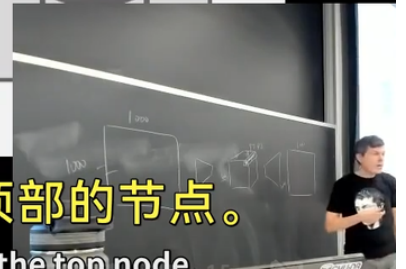


# PULL DATA TO QUERY



然后，底部的节点将其结果传递给顶部的节点。

And then the bottom guy sends his result up to the top node



## 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?

- 从远端获取到的数据被缓存在bufferpool中

### Query Fault Tolerance

| 为中间结果增加快照

## QUERY FAULT TOLERANCE

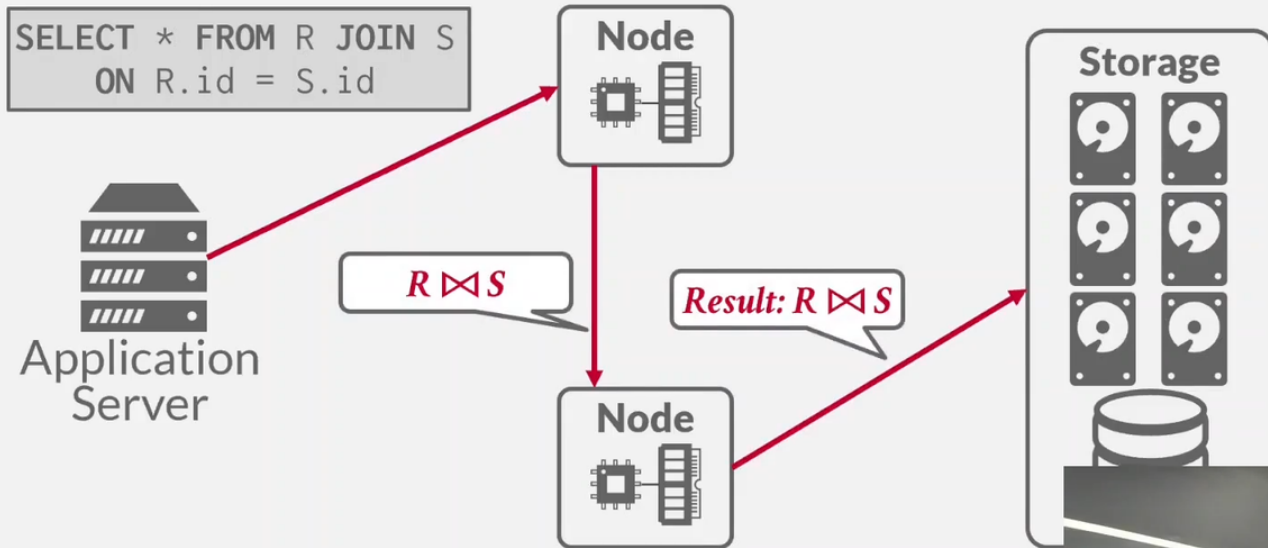
---

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

DB  
Fall 2023)

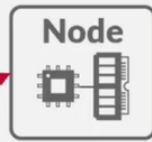
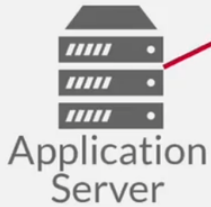
然后，与其立即将结果直接返回给那里的节点，  
我打算将其写入我的对象存储或写入我的共享磁盘。

I'm going to write it to my object store or write it to my shared disk.

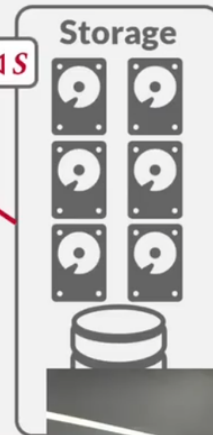
- hadoop会采用这种检查点机制，将计算的结果放入磁盘，避免在发生故障时不得不重启整个查询；但是map reduce的检查点机制拷贝的内容过多，影响效率；现在基本不用
- Presto/Trino是针对共享磁盘存储上执行分析查询更高效替代方案

## QUERY FAULT TOLERANCE

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



Result:  $R \bowtie S$



现在，如果这个节点崩溃了，另一个节点可以直接检索那个结果，并从断点处继续执行。

then the other node can just retrieve that result and pick up where it left off.

## 查询计划

在计划开始执行之前，运行一些测试获取参数（网络延时、节点间传输数据的时延）；并将这些参数作为代价模型的参数

## 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.

因此，像 DB2 所做的那样进行某种微基准测试是正确的做法。

So doing some kind of micro benchmarking like DB2 does is the right way to go.



## QUERY PLAN FRAGMENTS

### Approach #1: Physical Operators

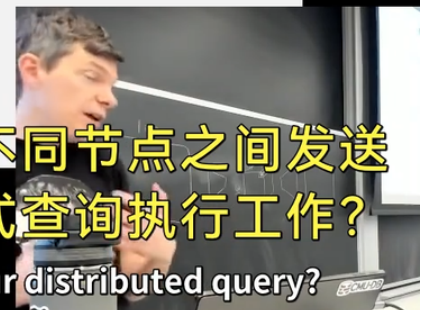
- Generate a single query plan and then break it up into partition-specific fragments.
- Most systems implement this approach.

### 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.

好的，现在的问题是，我们实际上要在不同节点之间发送什么信息，以指示它们代表我们的分布式查询执行工作？

different nodes to tell them to do work on behalf of our distributed query?

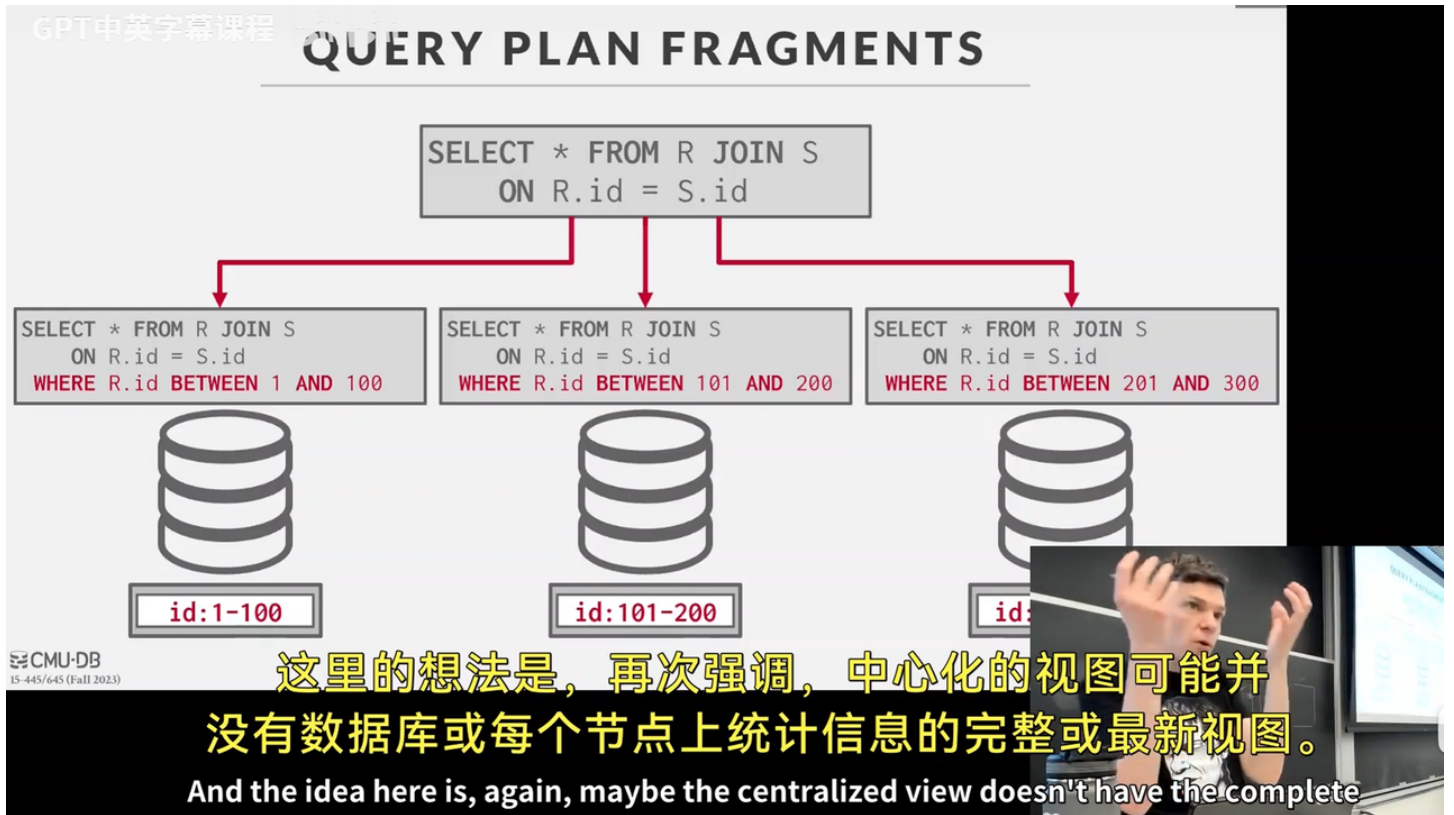


各位，这是我希望你们完成的任务”，  
然后他们可以各自做出自己的局部决策。

you to do, but then they can each make their own local decision

## 分发重写后的SQL

- 需要将物理计划转为sql，再由优化器优化后生成物理计划；每个节点自己去解析优化并生成物理计划





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

FROM R JOIN S

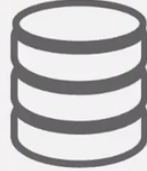
ON R.id = S.id

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



id:1-100

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



id:101-200

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



id:

然后，同样地，会有某种集中式的协调器，它知道如何合并结果并在最后将事物重新组合起来。

And then, again, there'll be some kind of centralized coordinator that knows how to

## 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 avoid false negatives due to missing tuples when running local join on each node.

我想说的是，我在这里将要讨论的内容，无论是共享磁盘还是无共享架构，都是一样的。

And I would say everything I'm going to talk about here is, again, the same for

GPT中英字幕课程

## 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
```



CMU-DB  
5-445/645 (Fall 2023)

这就是最佳情况，因为在计算连接时我没有进行任何数据传输，然后显然我需要发送结果，但这取决于选择性或我试图

So this is the best case scenario because I did no data transfer in order to compute

- 一张表在每个节点上被复制：图中显示了一个名为 **S** 的表，它在每个节点上都有相同的副本。
- 数据分片：另一张表 **R** 被分成不同的范围（例如，**id:1-100** 和 **id:101-200**）并分布在不同的节点上。
- 并行处理：每个节点（如 **P1** 和 **P2**）会在本地执行连接操作，即执行 `SELECT * FROM R JOIN S ON R.id = S.id`，把分片的 **R** 与本地副本的 **S** 进行连接。

- **结果合并：**各个节点会把连接操作的结果发送给一个协调节点，该节点将所有结果进行汇总，以提供最终的查询结果。
- 这种架构方式利用了数据复制和分片来并行处理查询操作。由于 S 表被复制到了每个节点上，每个节点只需处理自己的 R 分片数据，这样可以减少节点间的数据传输并提升查询速度。

PT中英字幕课程

## 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
```

MU-DB  
645 (Fall 2023)

但如果这两个表是按照你试图连接的相同属性进行分区的。

But if the two tables are partitioned on the same attributes as you're trying to join

MU-DB  
Fall 2023

对于任意一组查询和任意一组可用于分区的属性，确定表的最佳分区方案并非易事。

## 假阴性

因为数据在不同节点导致的该匹配上的节点没有匹配成功

### 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
```



DB Fall 2023

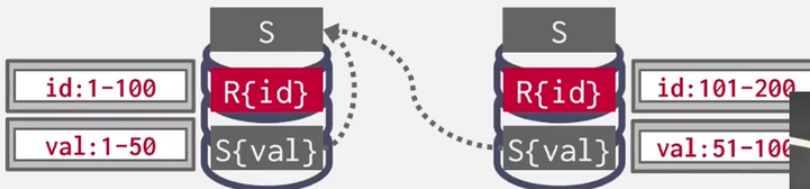
在这种情况下，R 表是基于 ID 进行分区的，但现在我的 S 表是基于另一个属性——值进行分区的。

因此，如果我只是在这里对本地数据进行连接操作，同样，可能会出现假阴性结果，因为如果存在某个 id 等于 1

### 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
```



CMU-DB Fall 2023

可能会出现这样的情况，有人选择了，我希望基于某个值进行分区，因为我的大多数查询都需要根据该值进行连接操作或

It may be the case that someone picked, I want to partition on value because most of