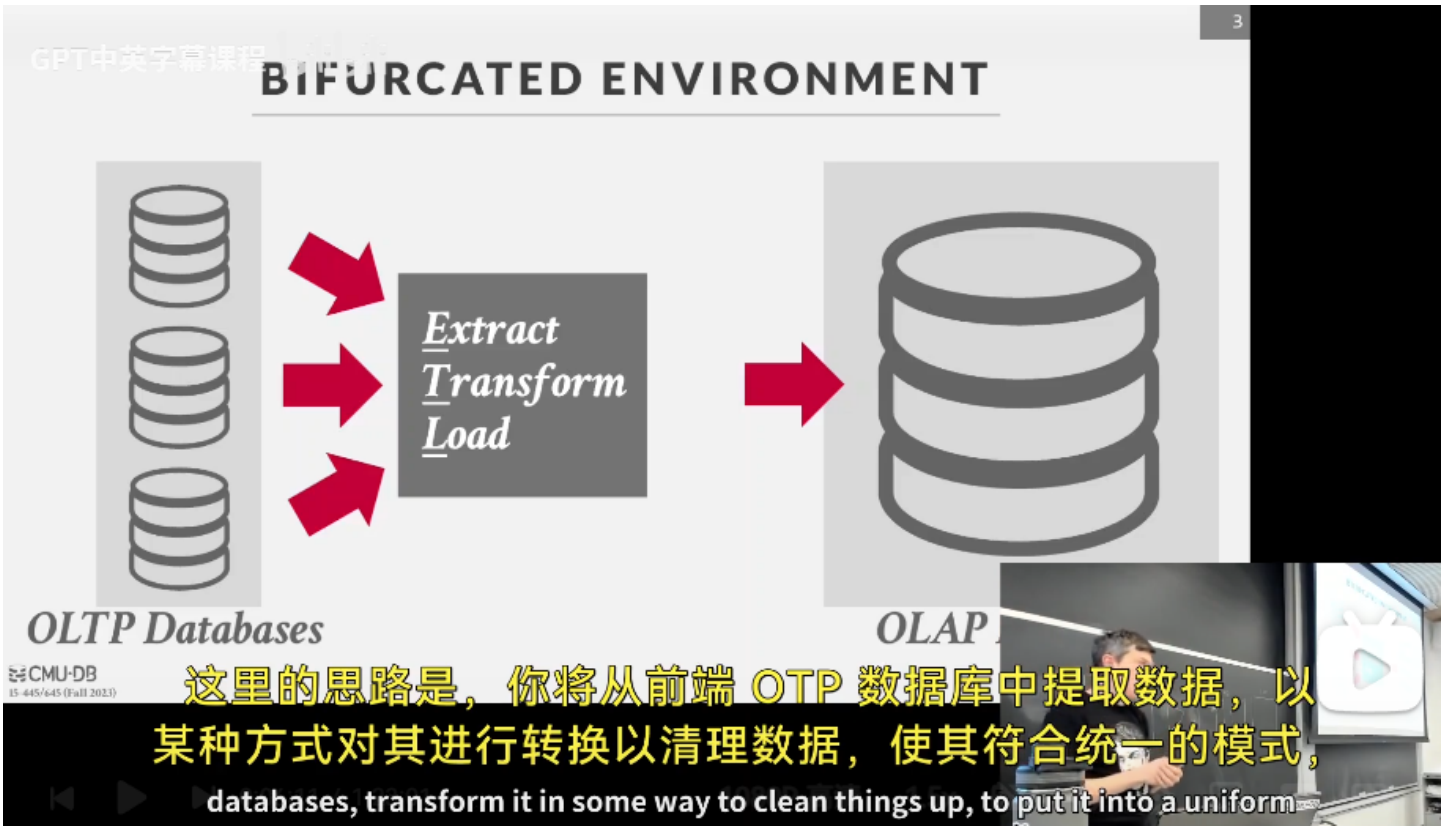
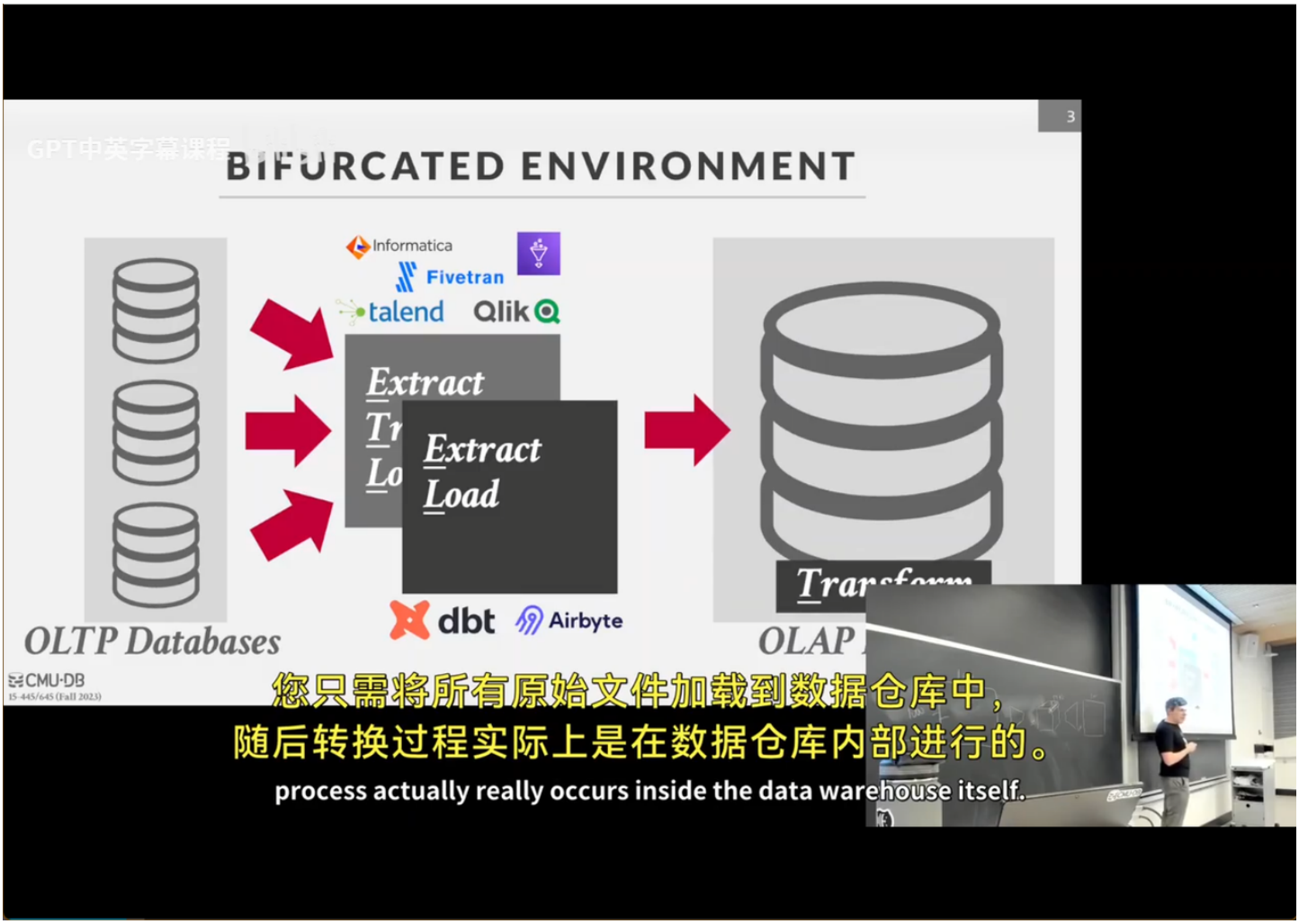
分布式OLAP DB

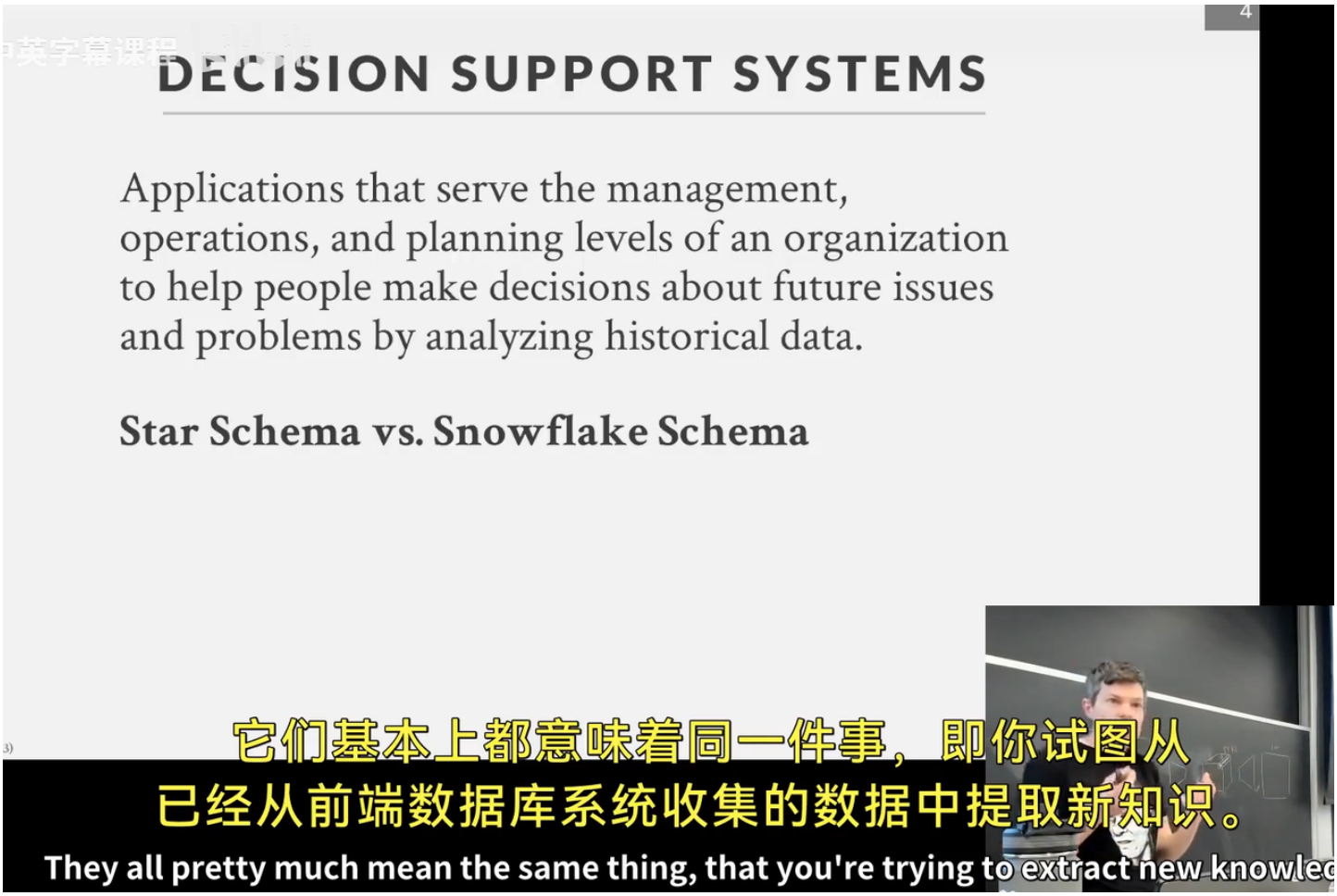
背景介绍





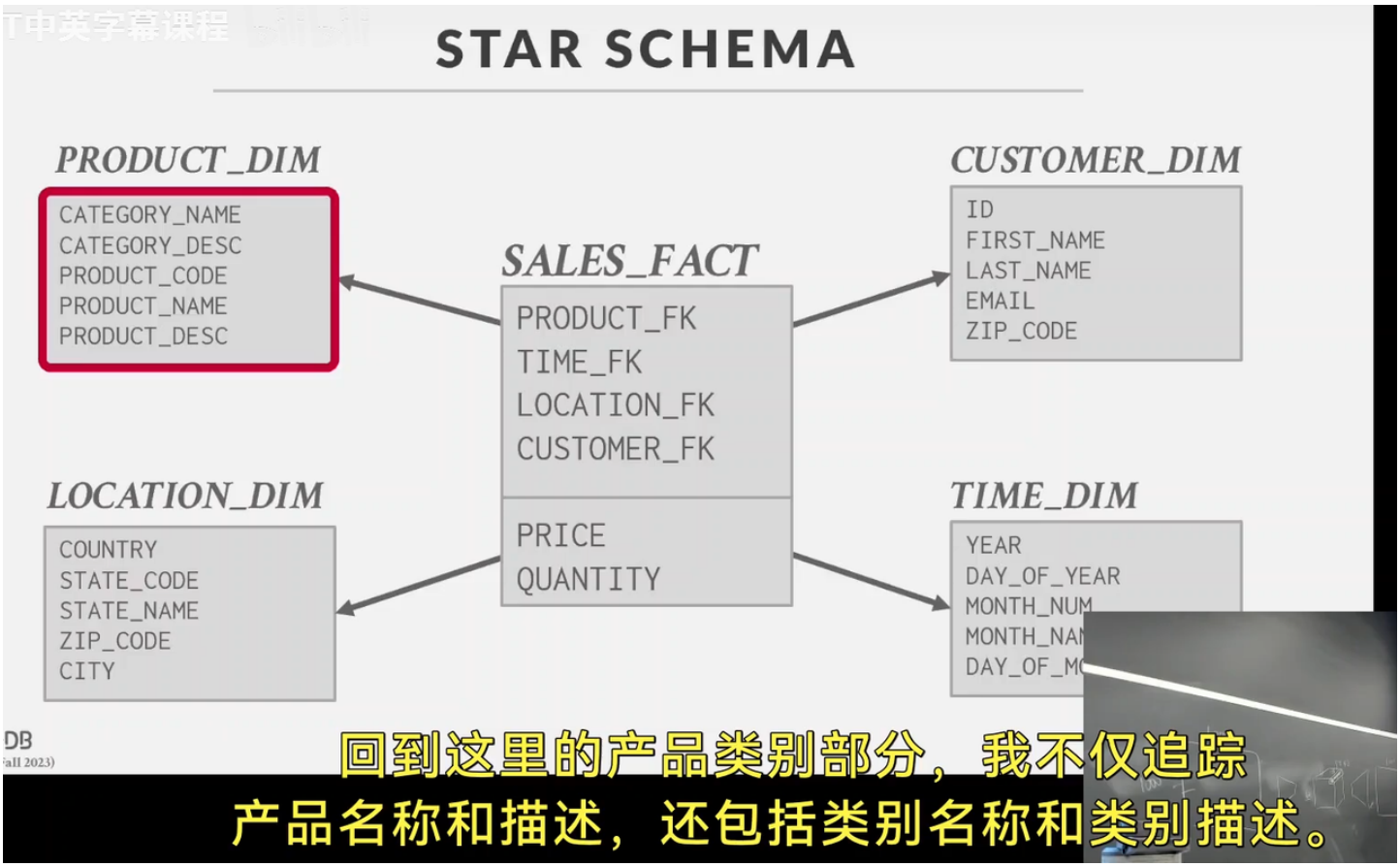
应用程序在前端运行着OLTP DB，需要将其整合到自己的大型OLAP系统中；需要经过抽取转换，因为前端应用程序编写的规范不一样。

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



构建OLAP数据库的两种技术

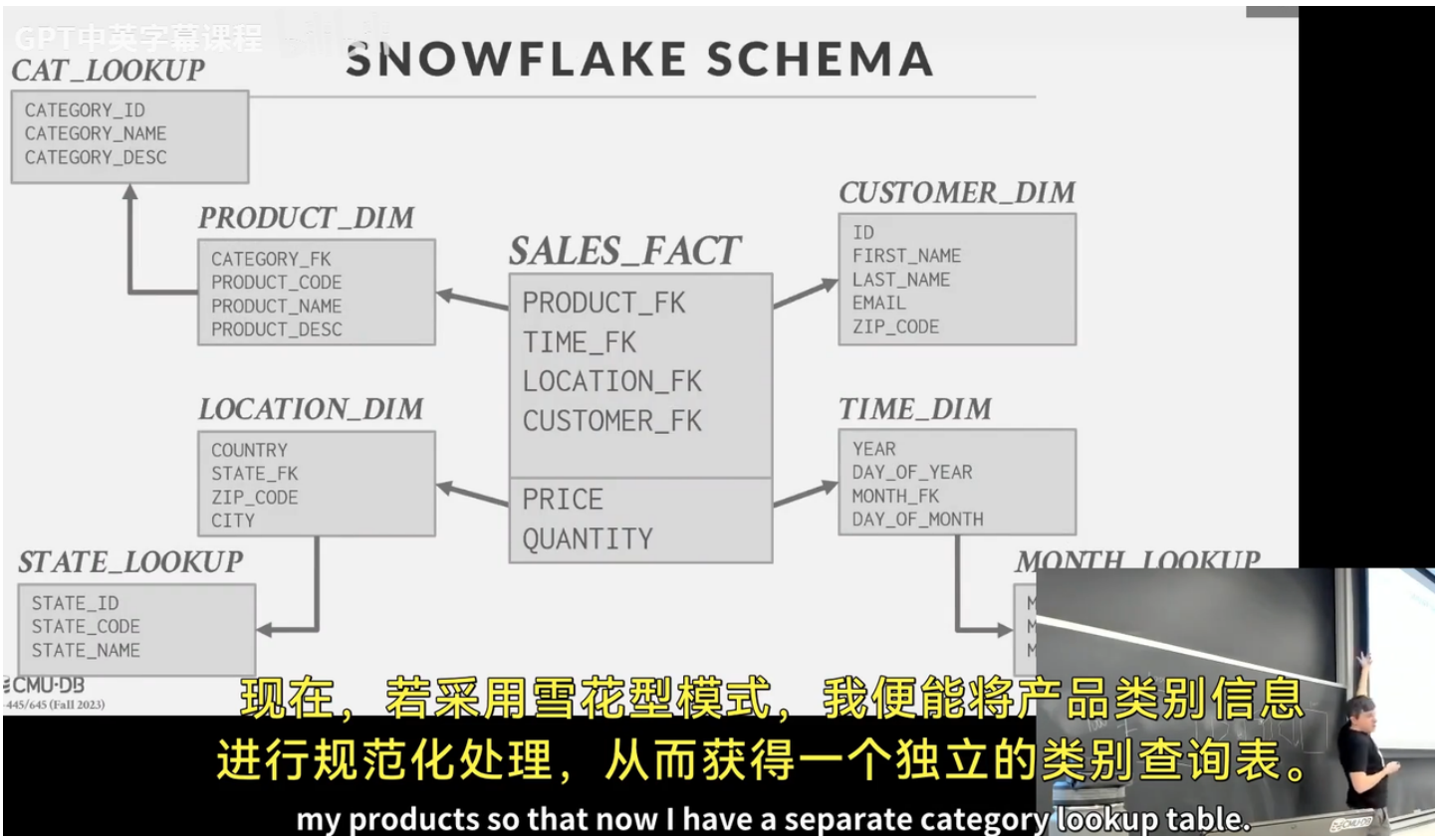
Star schema（subset of snowflake）



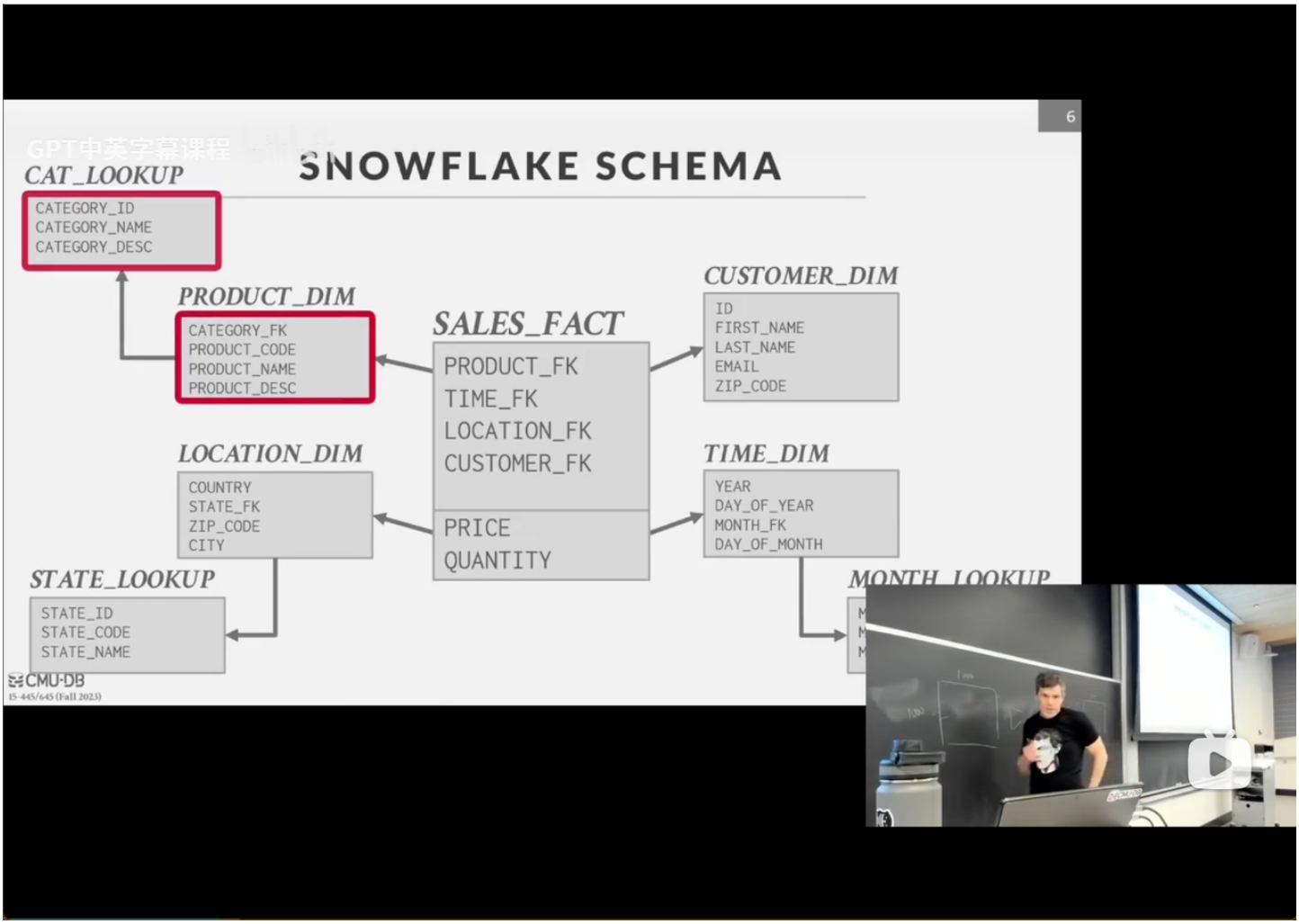
去规范化：无需执行连接操作就能直接获取所有所需的信息；坏处是一旦更新了某个元组的名称，就必须保证更新所有相关的元组。

星型结构是去规范化的

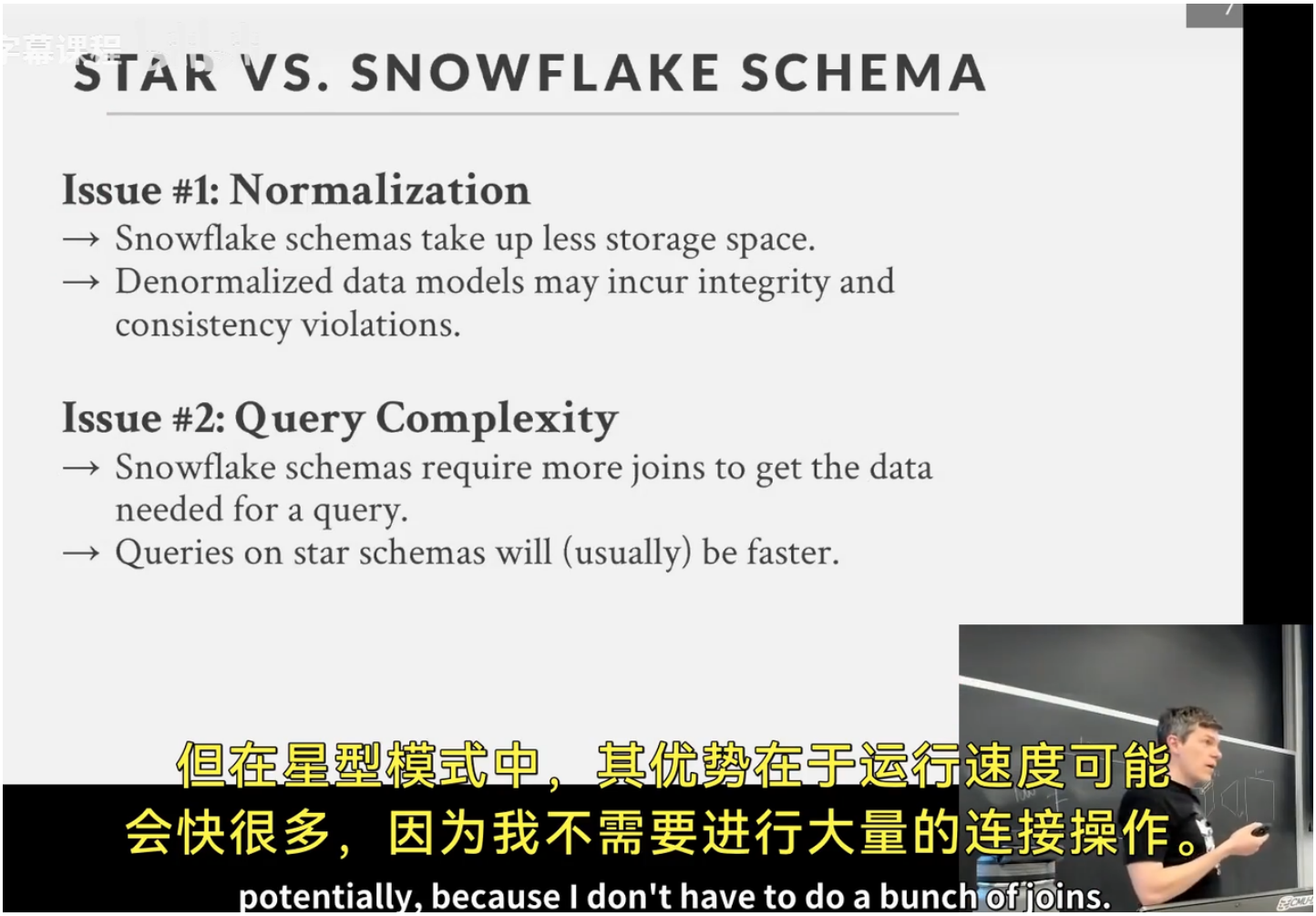
Snowflake schema

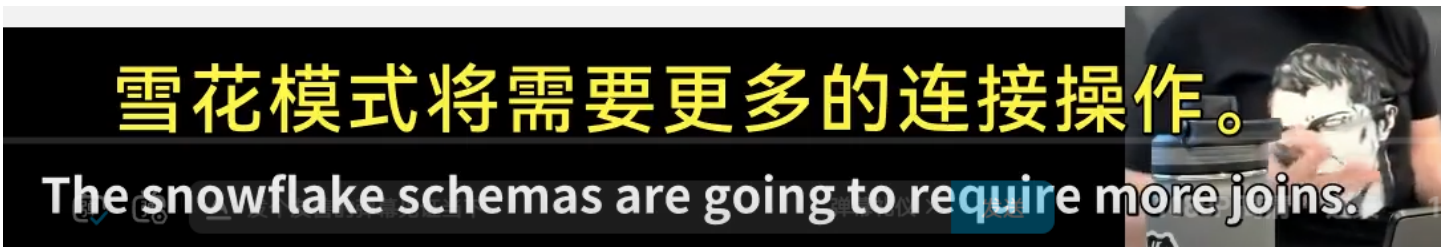


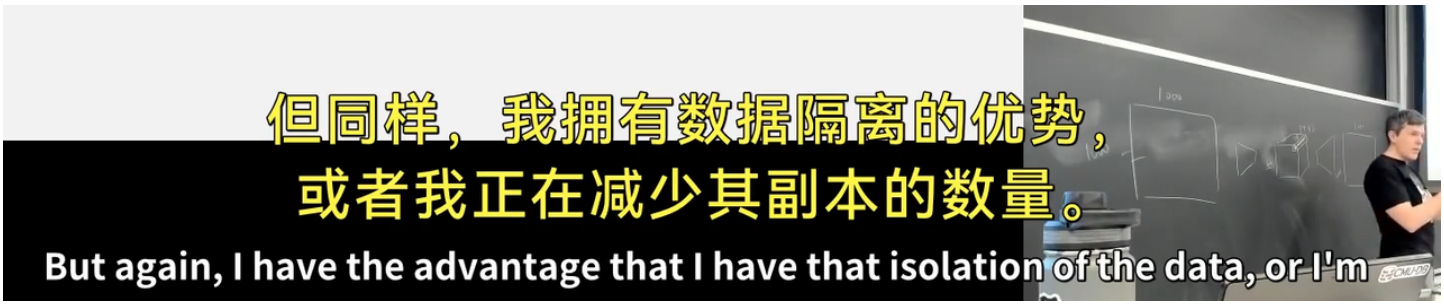
获取指定产品的类别只需要执行一次连接操作即可



Star VS. Snowflake schema

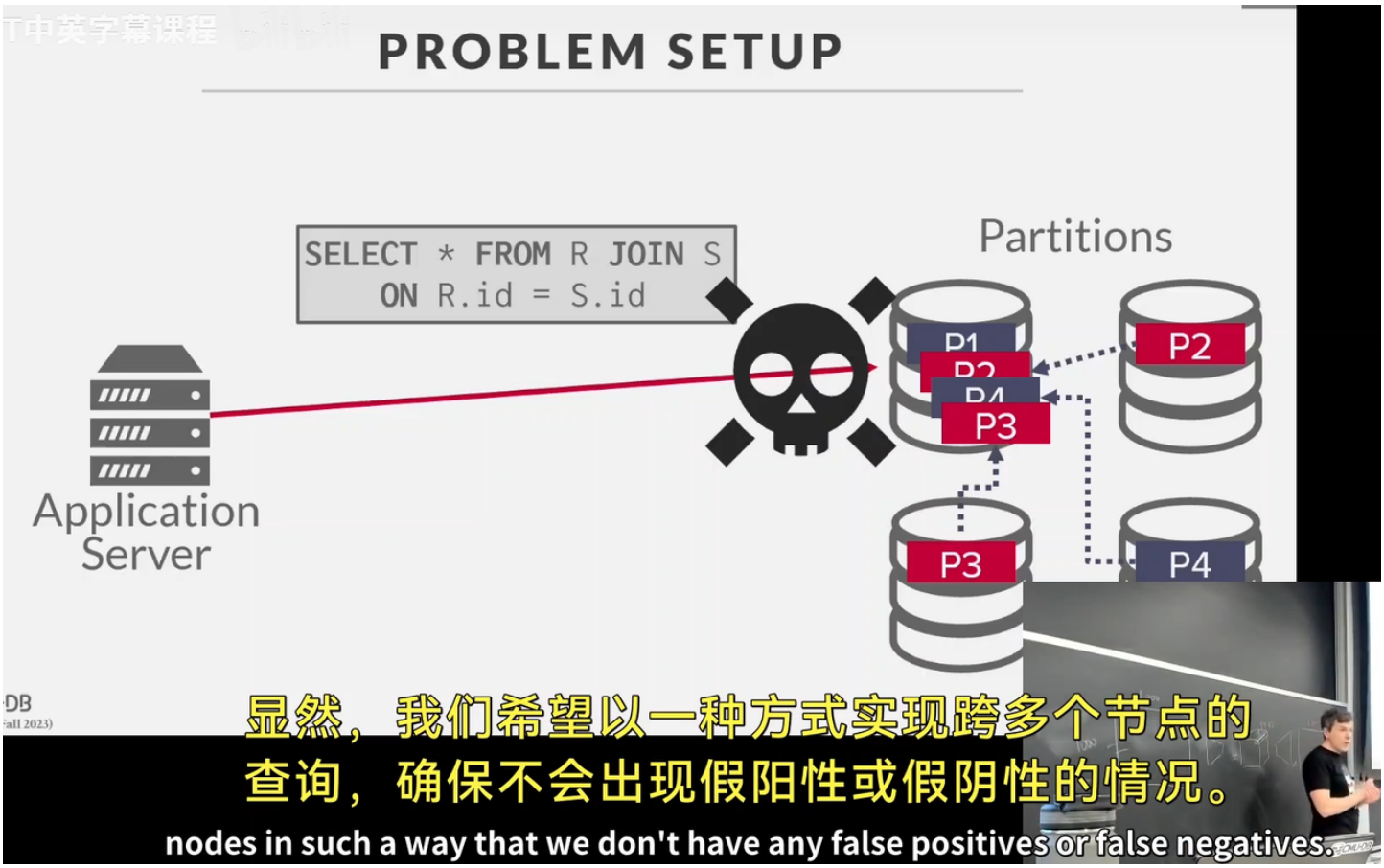






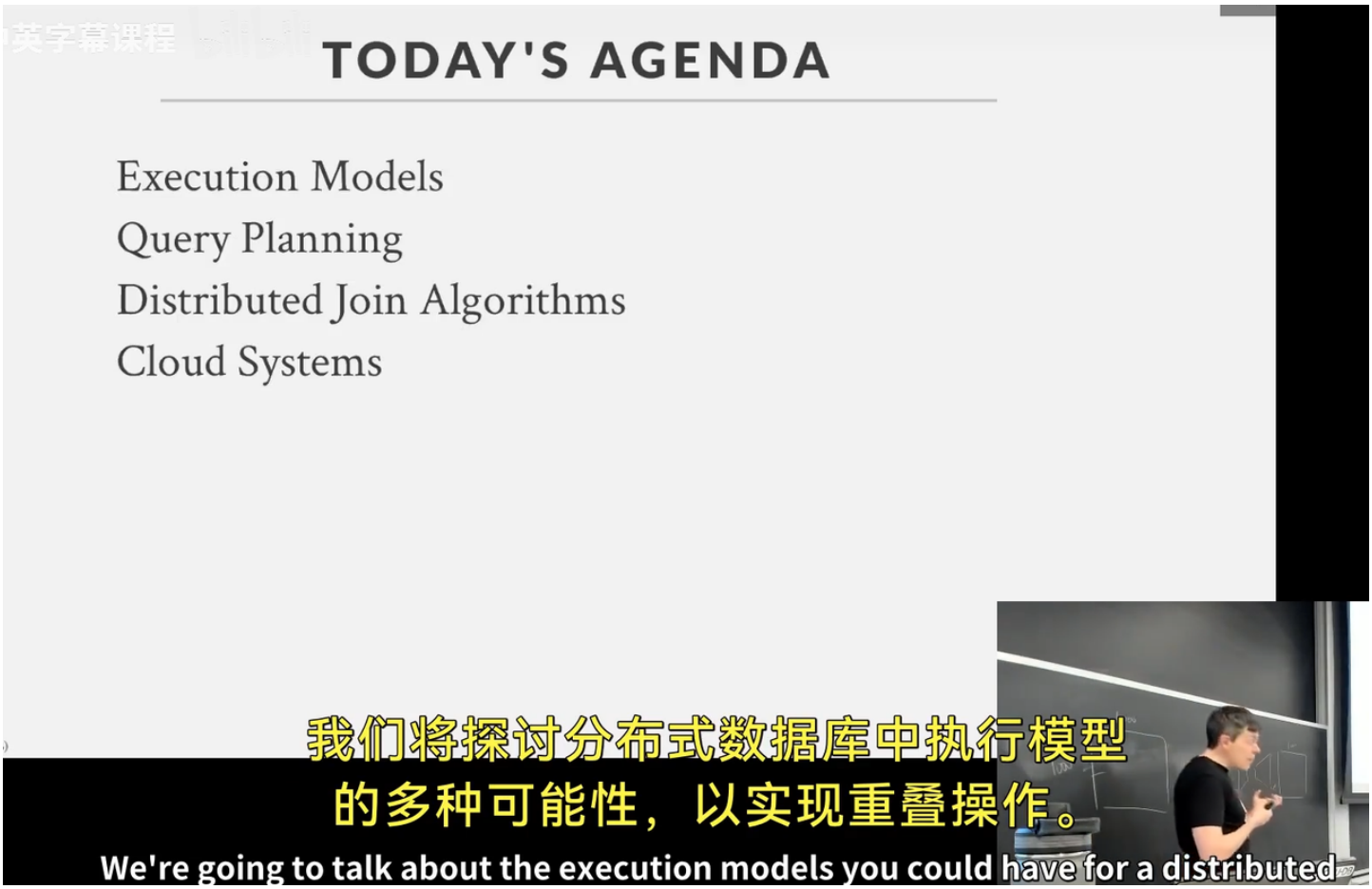
现实世界中使用star的很少了，一般使用snowflake

Problem setup



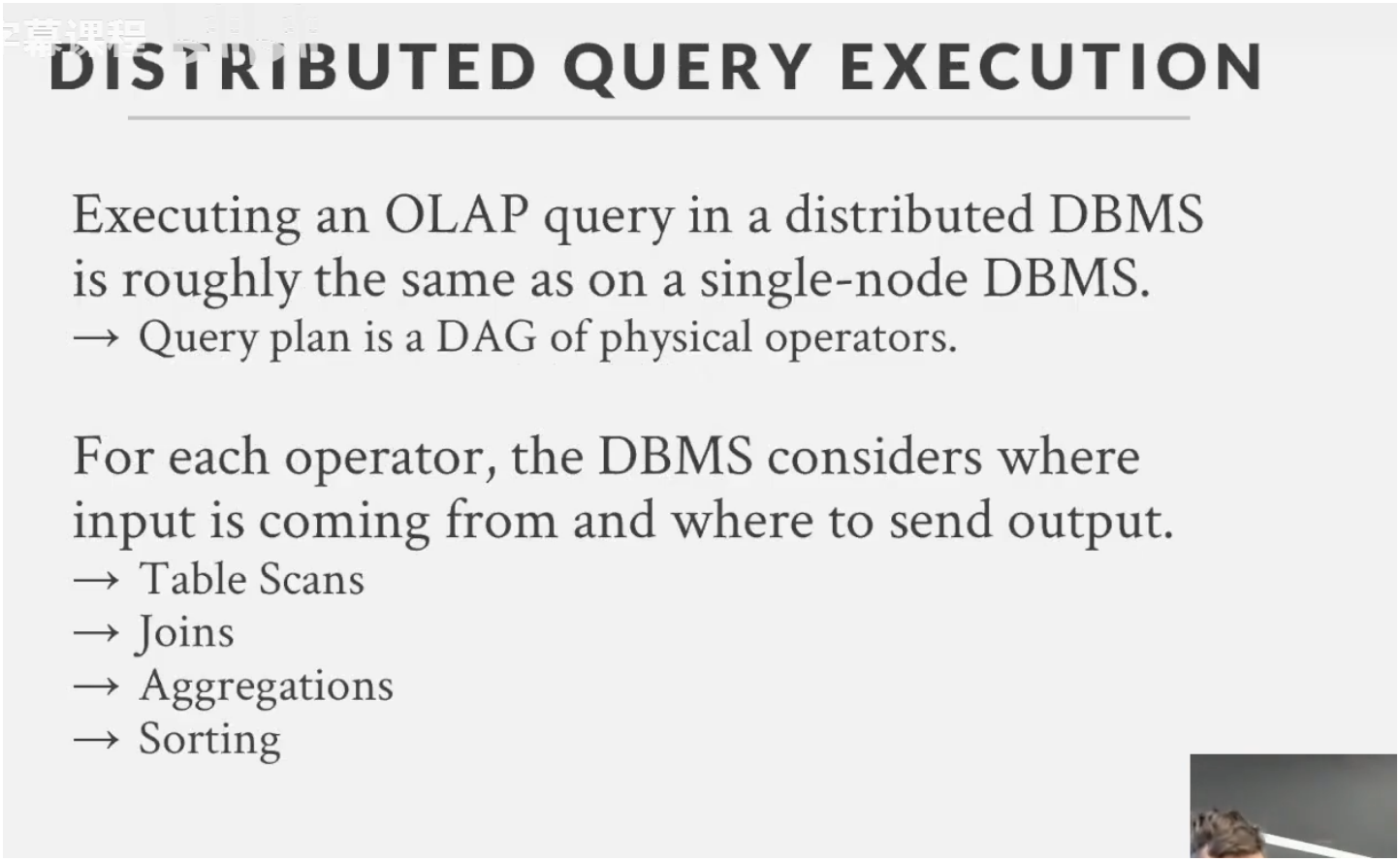
执行连接操作，让P2\P3\P4把数据都发送给P1，直接执行join，但是这样会变成一个单节点的情况，失去了分布式的优势，这样是不对的

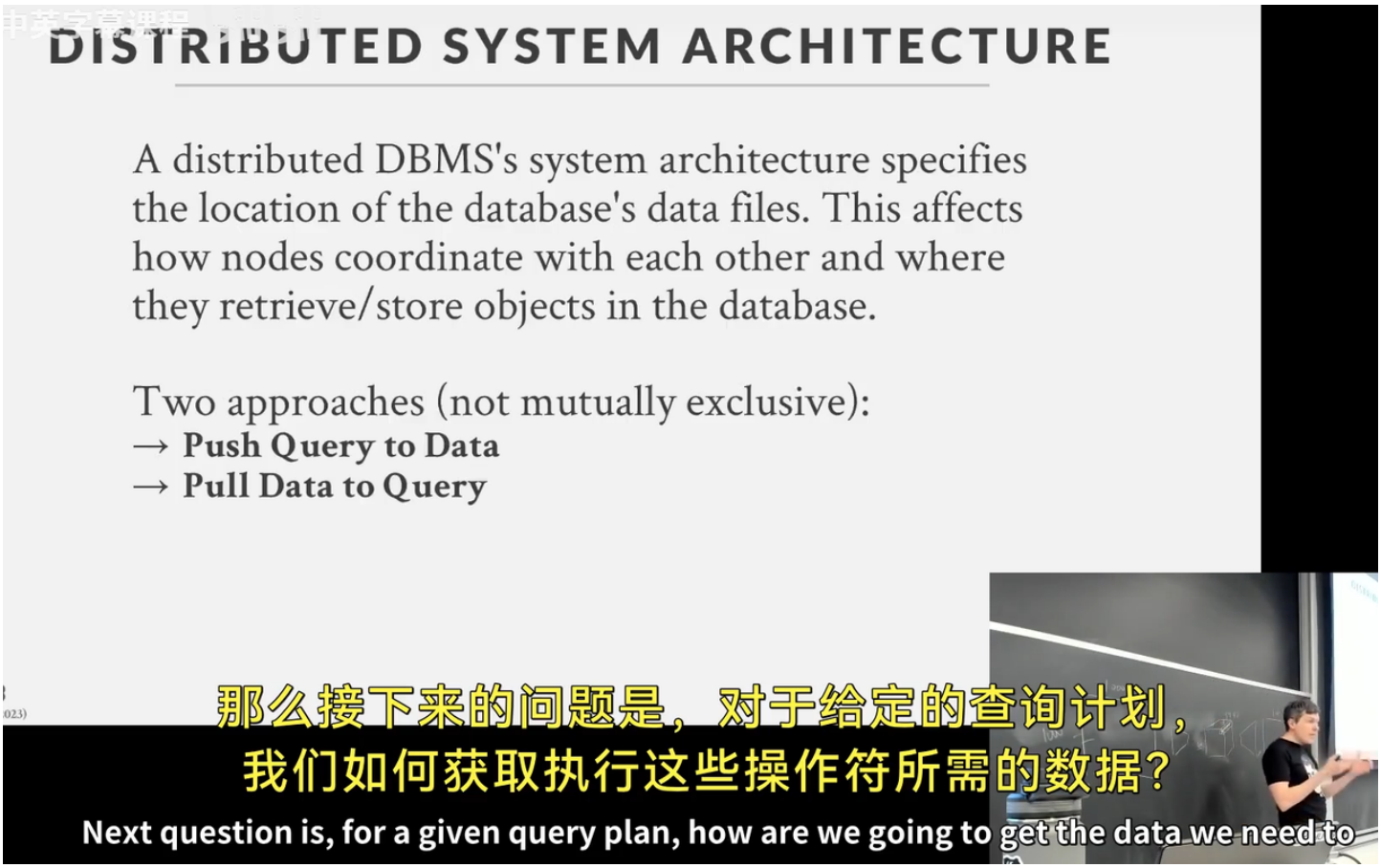
Agenda



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





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.

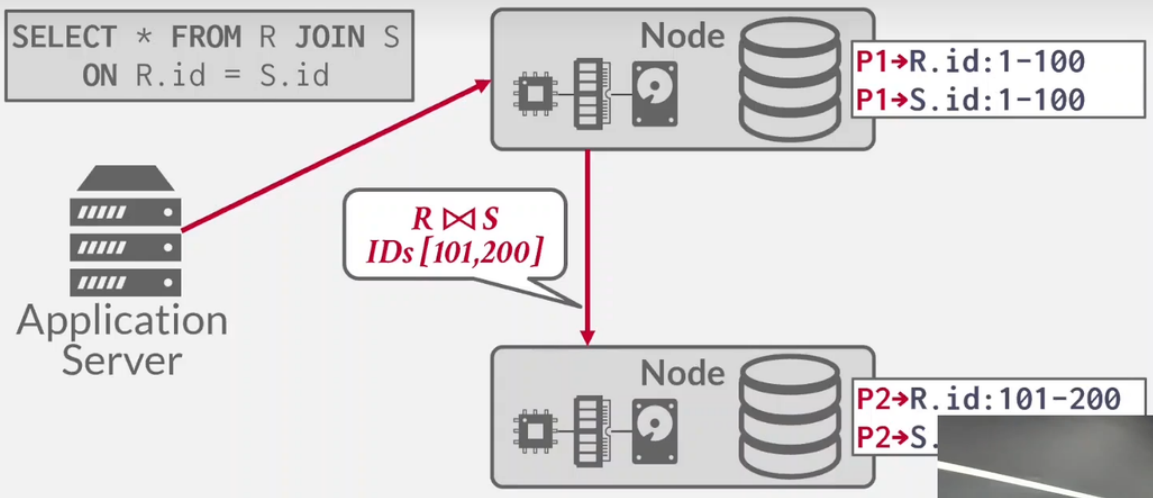
→ Perform as much filtering and processing as possible where data resides before transmitting over network.

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.

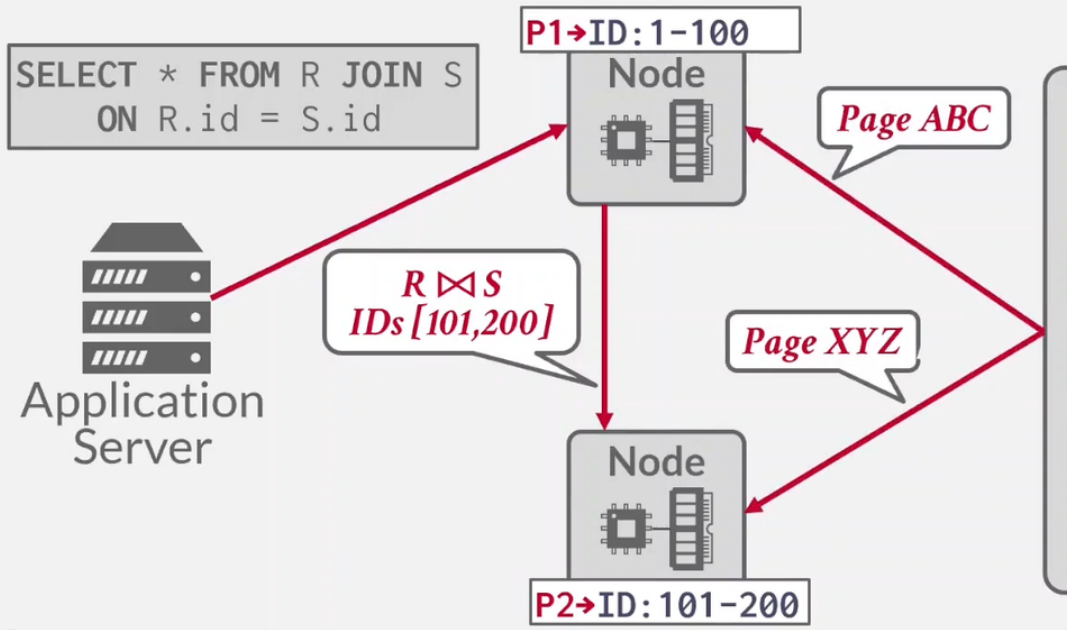
PUSH QUERY TO DATA



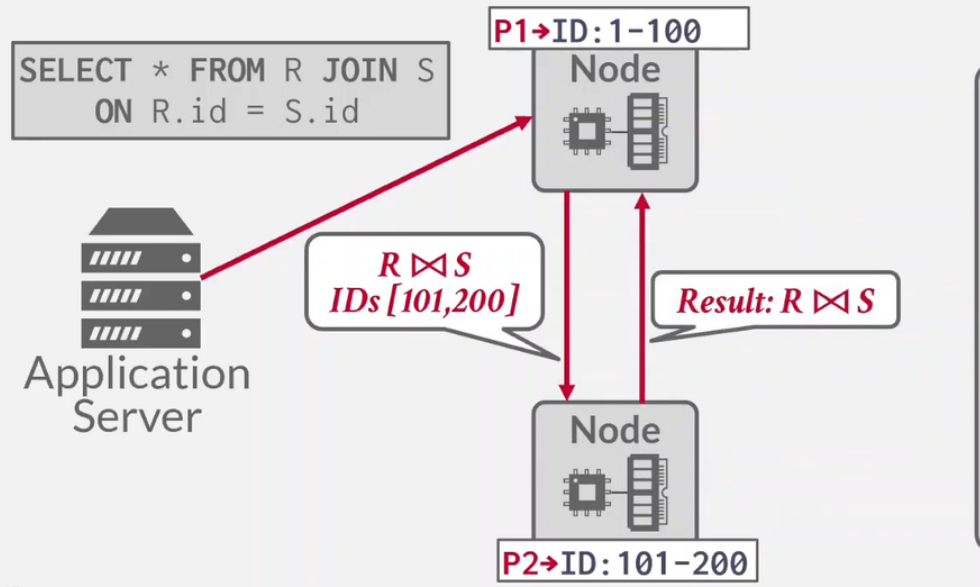
因此，不是顶层节点告诉底层节点：“把你所有的数据都发给我”，而是将查询计划片段下发。

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

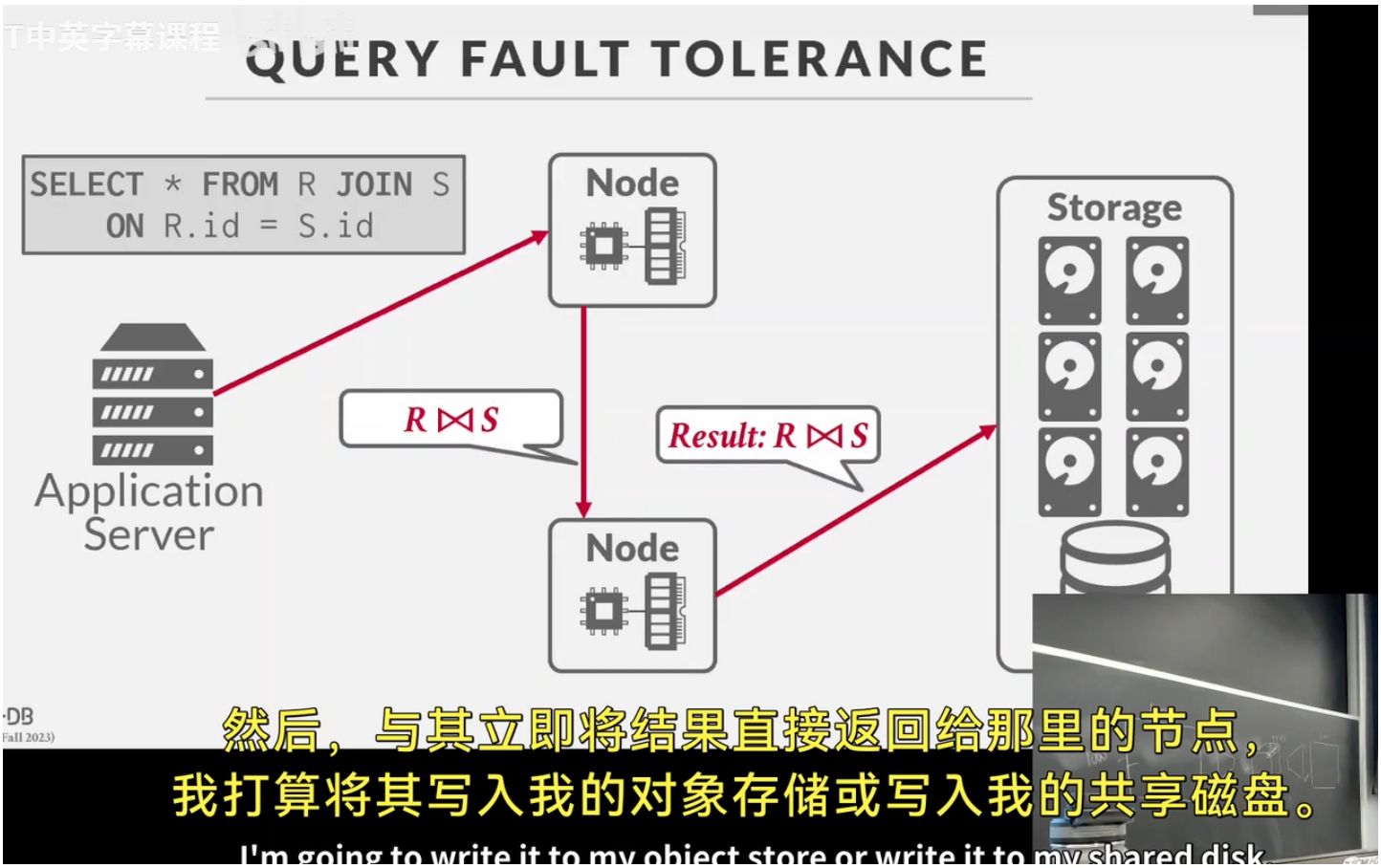
> 为中间结果增加快照

QUERY FAULT TOLERANCE

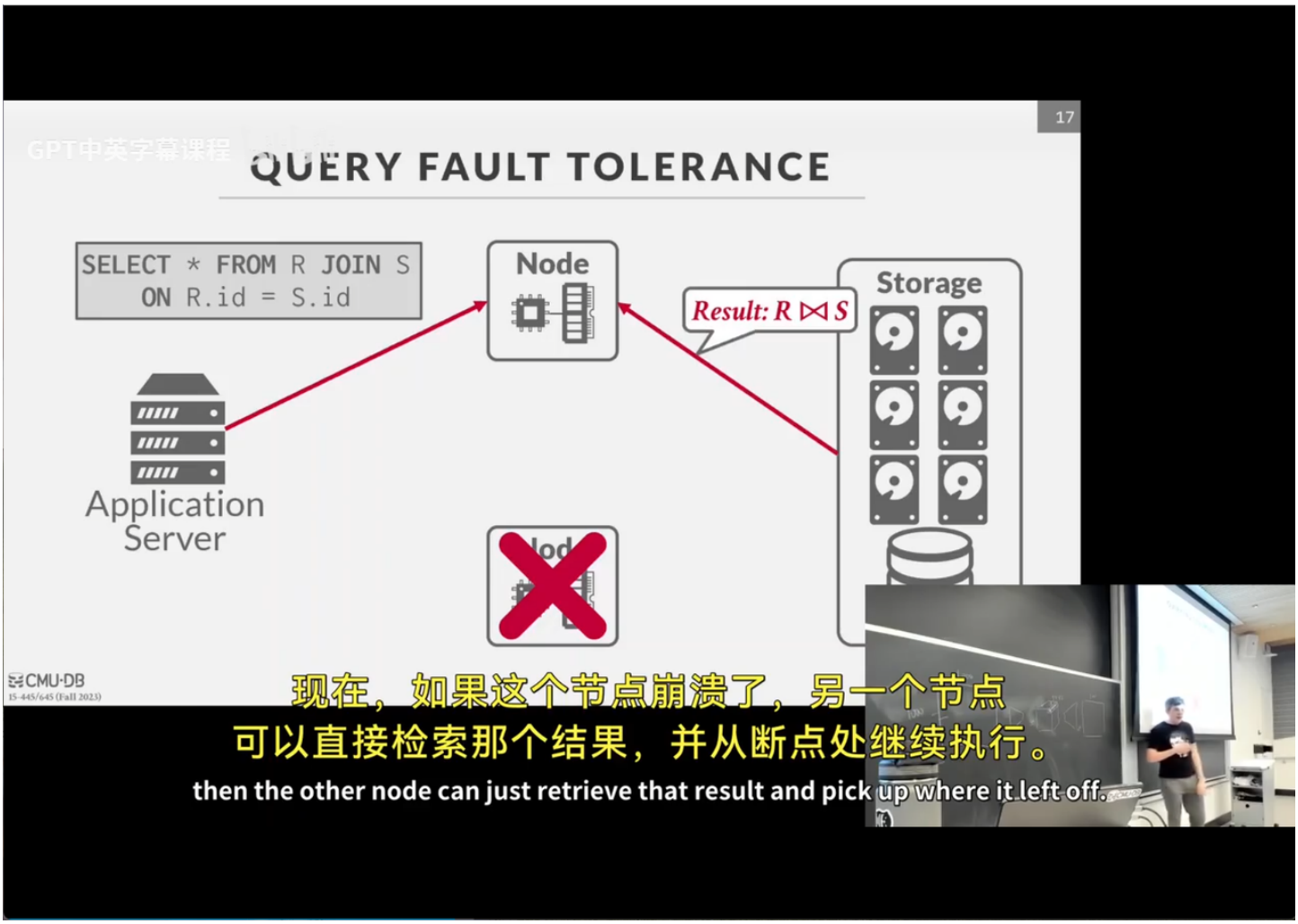
Most shared-nothing distributed OLAP DBMSs are designed to assume that nodes do not fail during query execution.

→ 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.



- 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



查询计划

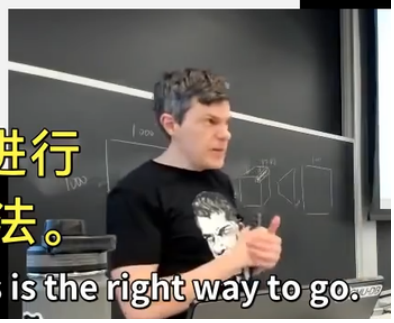
> 在计划开始执行之前，运行一些测试获取参数（网络延时、节点间传输数据的时延）；并将这些参数作为代价模型的参数

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.

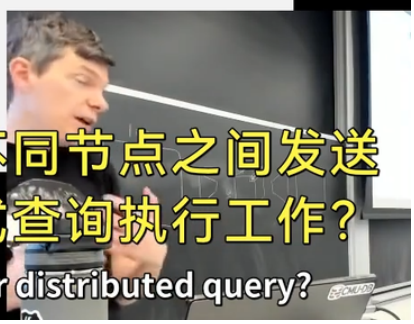
QUERY PLAN FRAGMENTS

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.

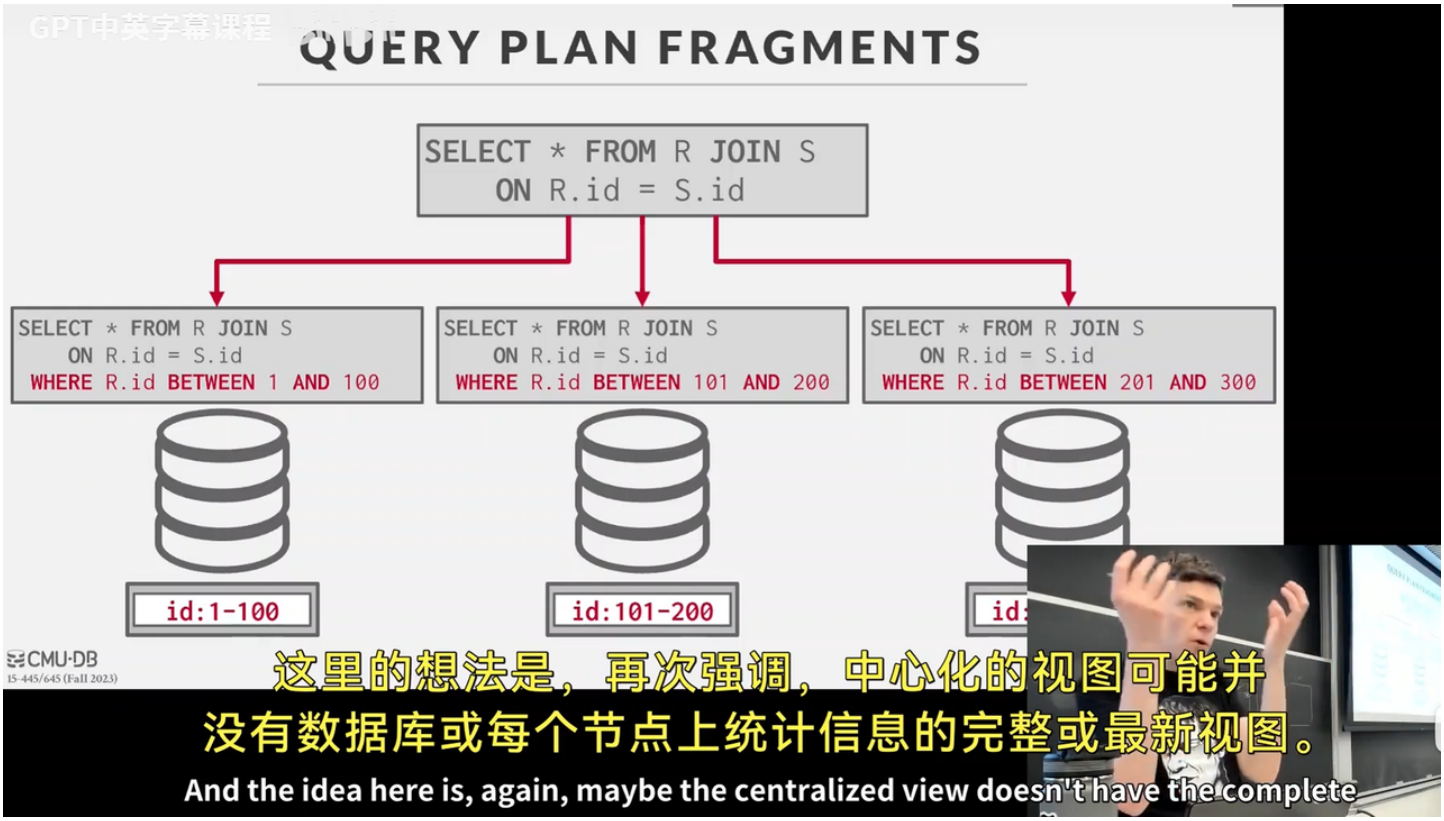


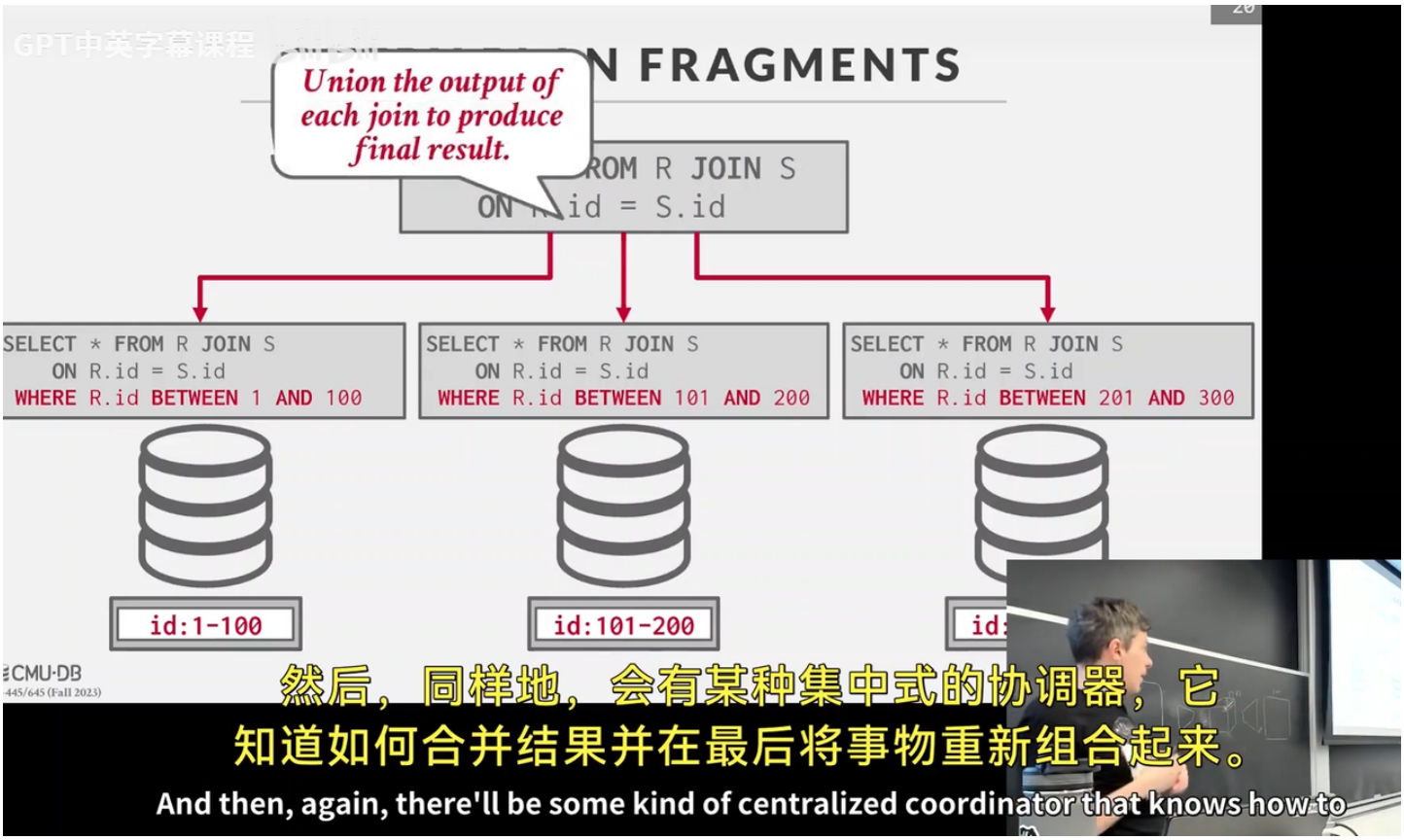
好的，现在的问题是，我们实际上要在不同节点之间发送什么信息，以指示它们代表我们的分布式查询执行工作？

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

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





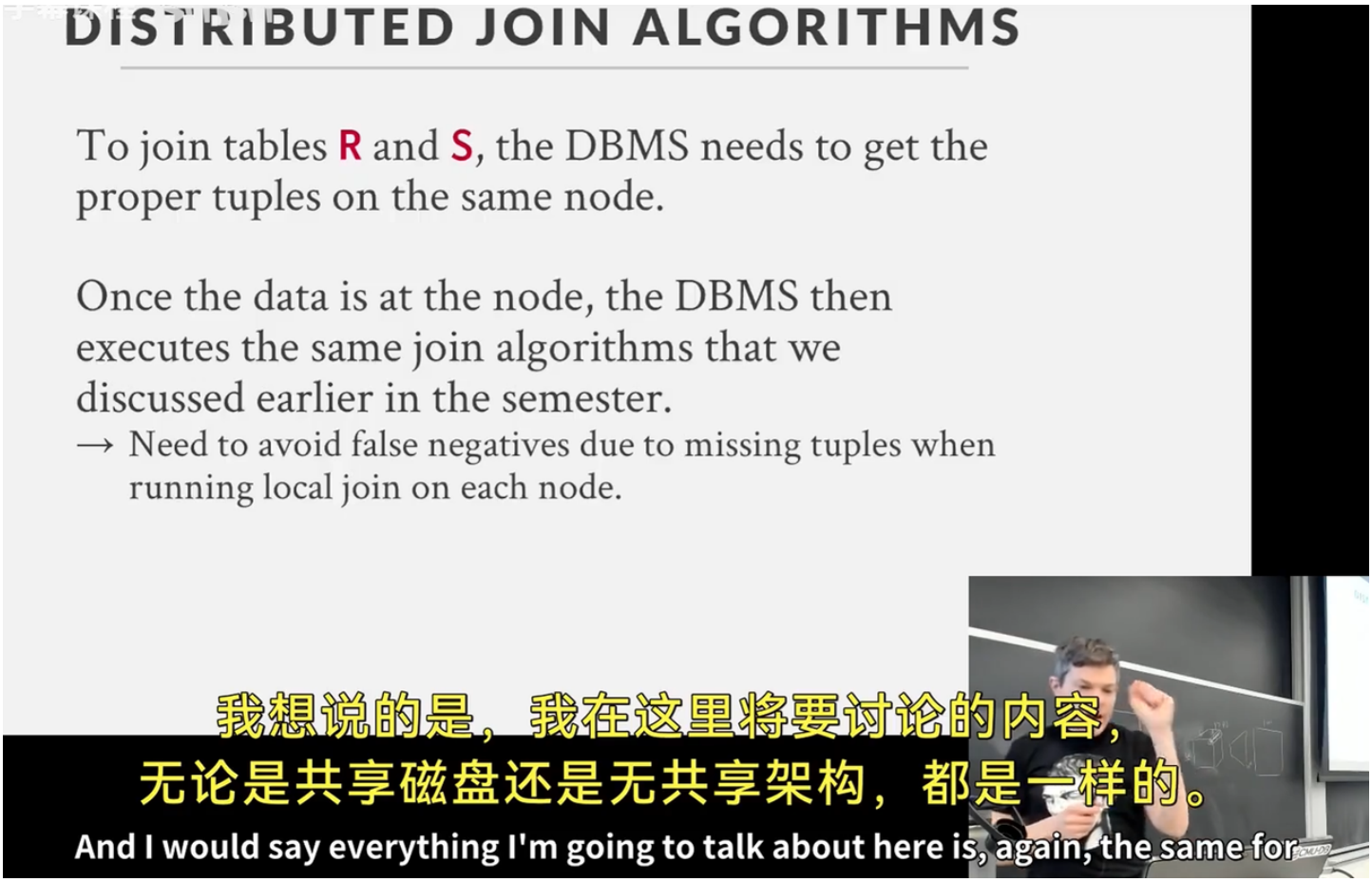
OBSERVATION

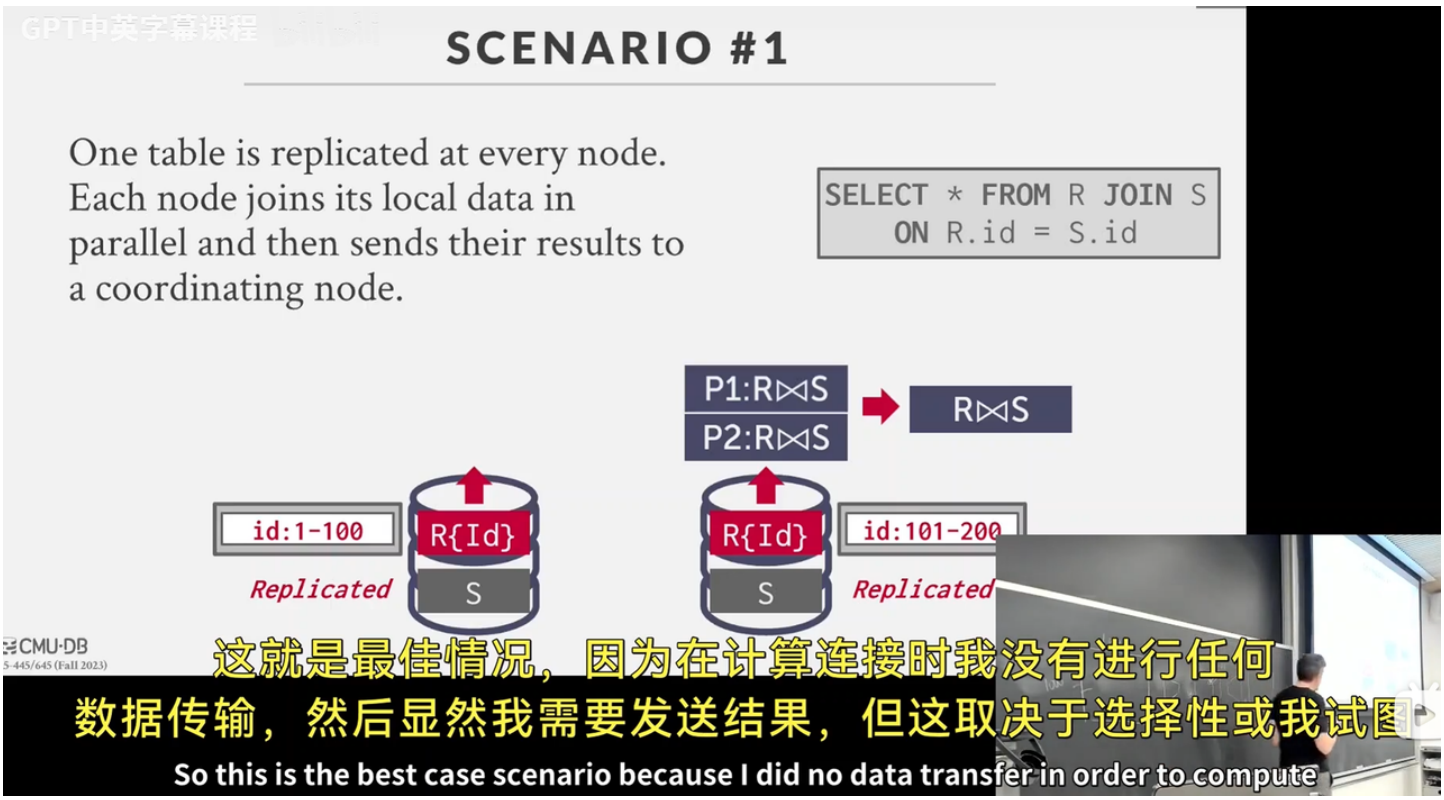
The efficiency of a distributed join depends on the target tables' partitioning schemes.

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

→ You lose the parallelism of a distributed DBMS.

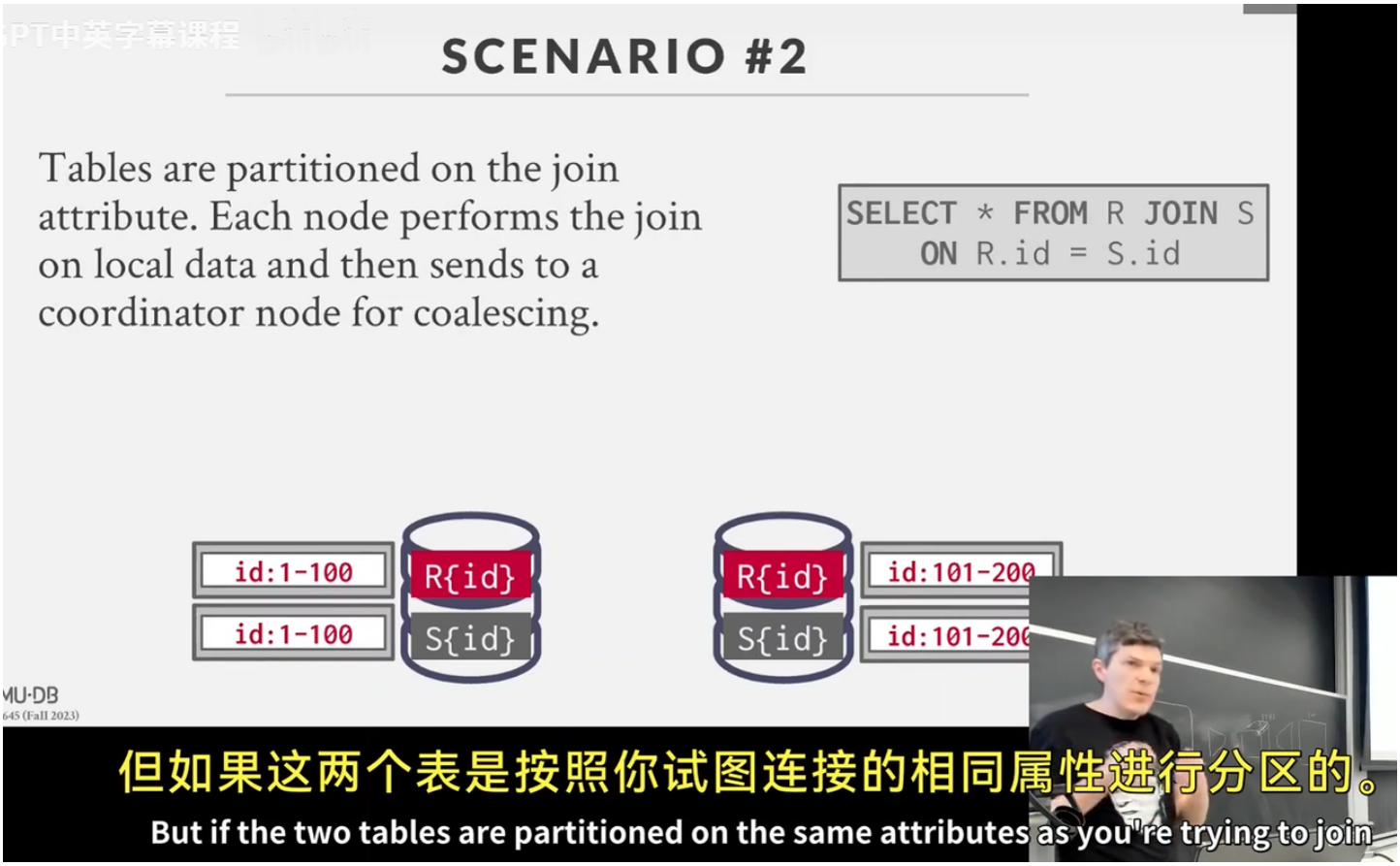
→ Costly data transfer over the network.

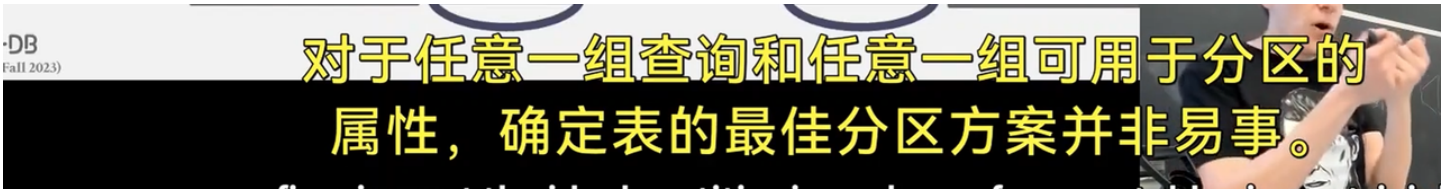




- **一张表在每个节点上被复制**：图中显示了一个名为 `S` 的表，它在每个节点上都有相同的副本。
- **数据分片**：另一张表 `R` 被分成不同的范围（例如，`id:1-100` 和 `id:101-200`）并分布在不同的节点上。
- **并行处理**：每个节点（如 P1 和 P2）会在本地执行连接操作，即执行 `SELECT * FROM R JOIN S ON R.id = S.id`，把分片的 `R` 与本地副本的 `S` 进行连接。

- **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。





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
```

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。

partitioned on value, some other attribute

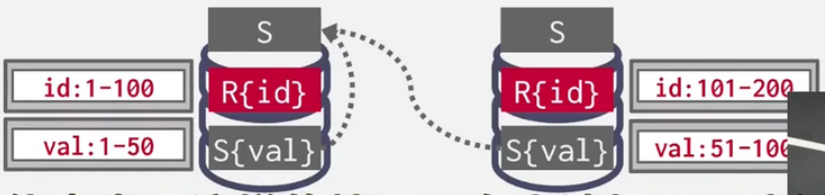
因此，如果我只是在这里对本地数据进行连接操作，同样，
可能会出现假阴性结果，因为如果存在某个 id 等于 1

GPT中英字幕课程

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
```



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