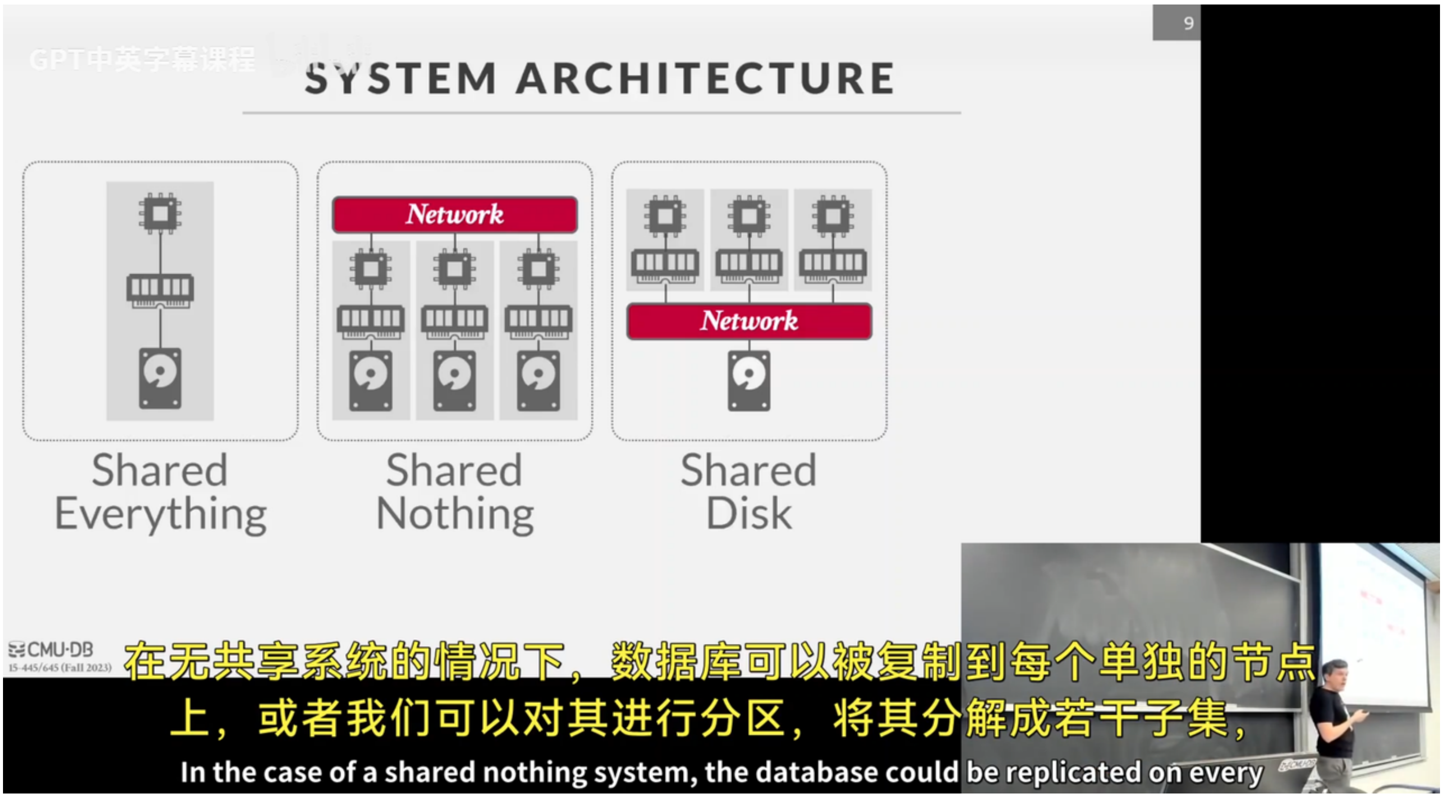
分布式数据库系统intro

系统架构



9

SYSTEM ARCHITECTURE

Shared Everything

Shared Nothing

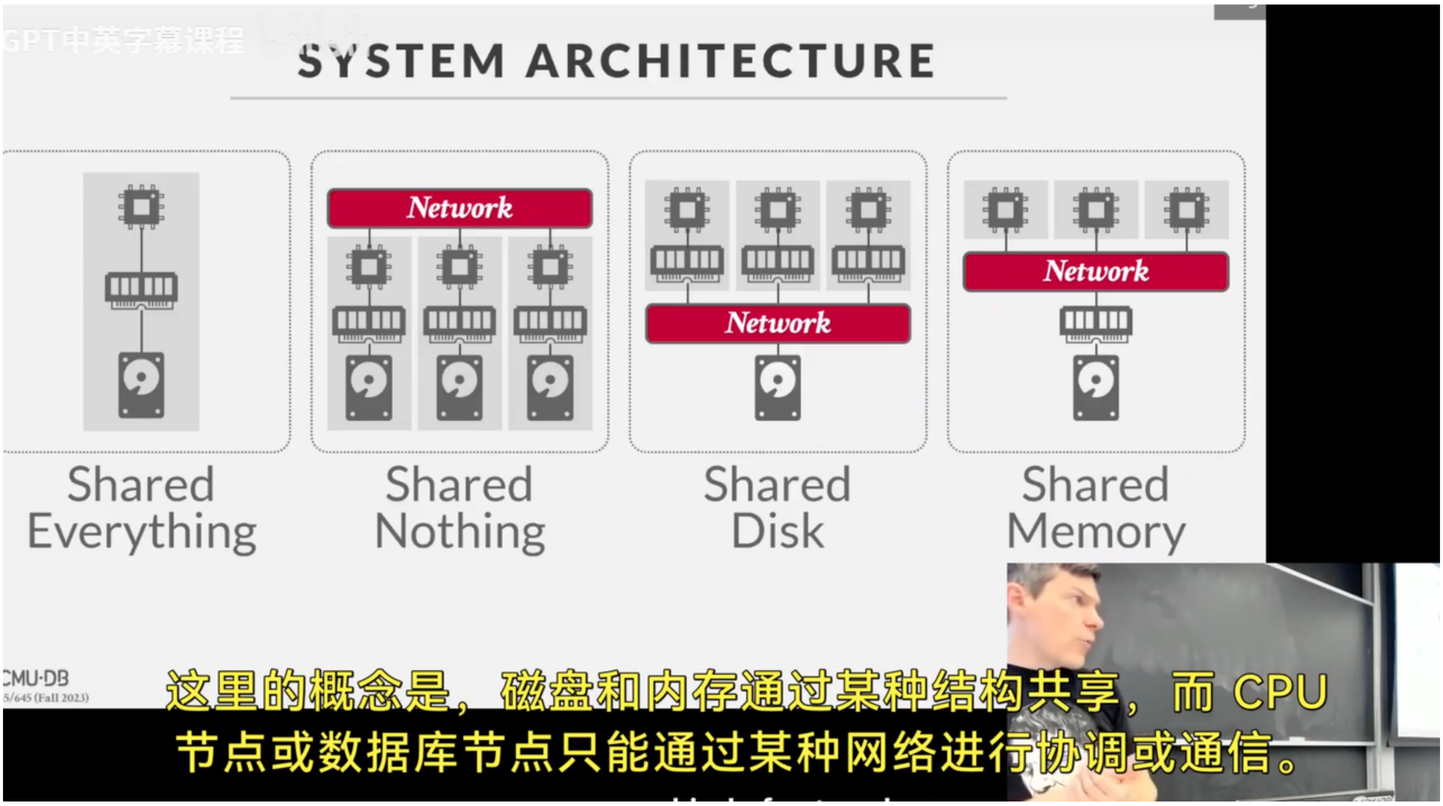
Shared Disk

CMU·DB
15-445/645 (Fall 2023)

在无共享系统的情况下，数据库可以被复制到每个单独的节点上，或者我们可以对其进行分区，将其分解成若干子集，

In the case of a shared nothing system, the database could be replicated on every

- Shared memory通常用于高计算领域，构建一个单一的非聚合内存池；所有节点都可以通过该内存池协调



SYSTEM ARCHITECTURE

Shared Everything

Shared Nothing

Shared Disk

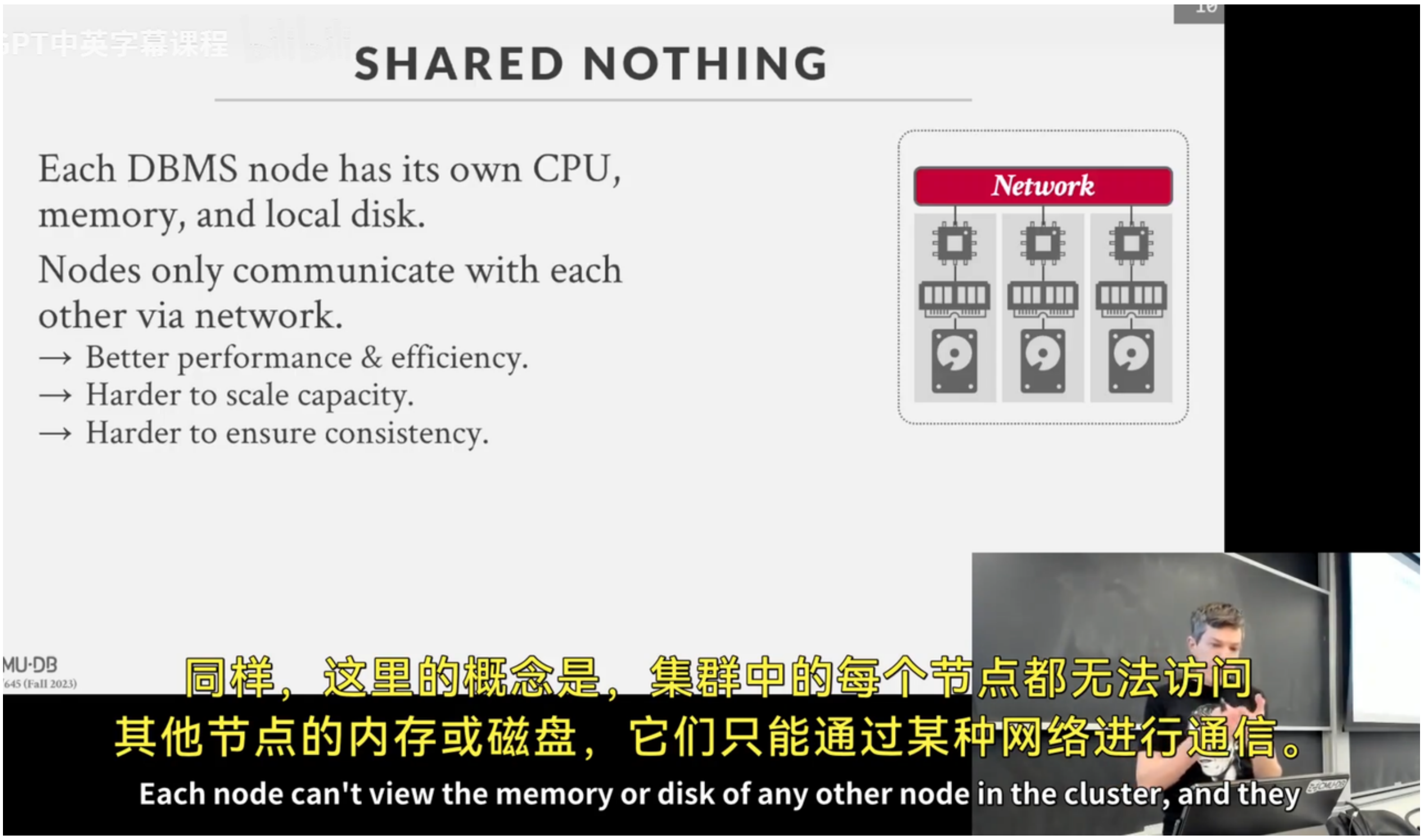
Shared Memory

CMU-DB
5/645 (Fall 2023)

这里的概念是，磁盘和内存通过某种结构共享，而 CPU
节点或数据库节点只能通过某种网络进行协调或通信。

Shared notinng

> 将数据库分片或者分区到不同节点；可以分发查询，通过Exchage操作符来把并行执行的结果进行汇总



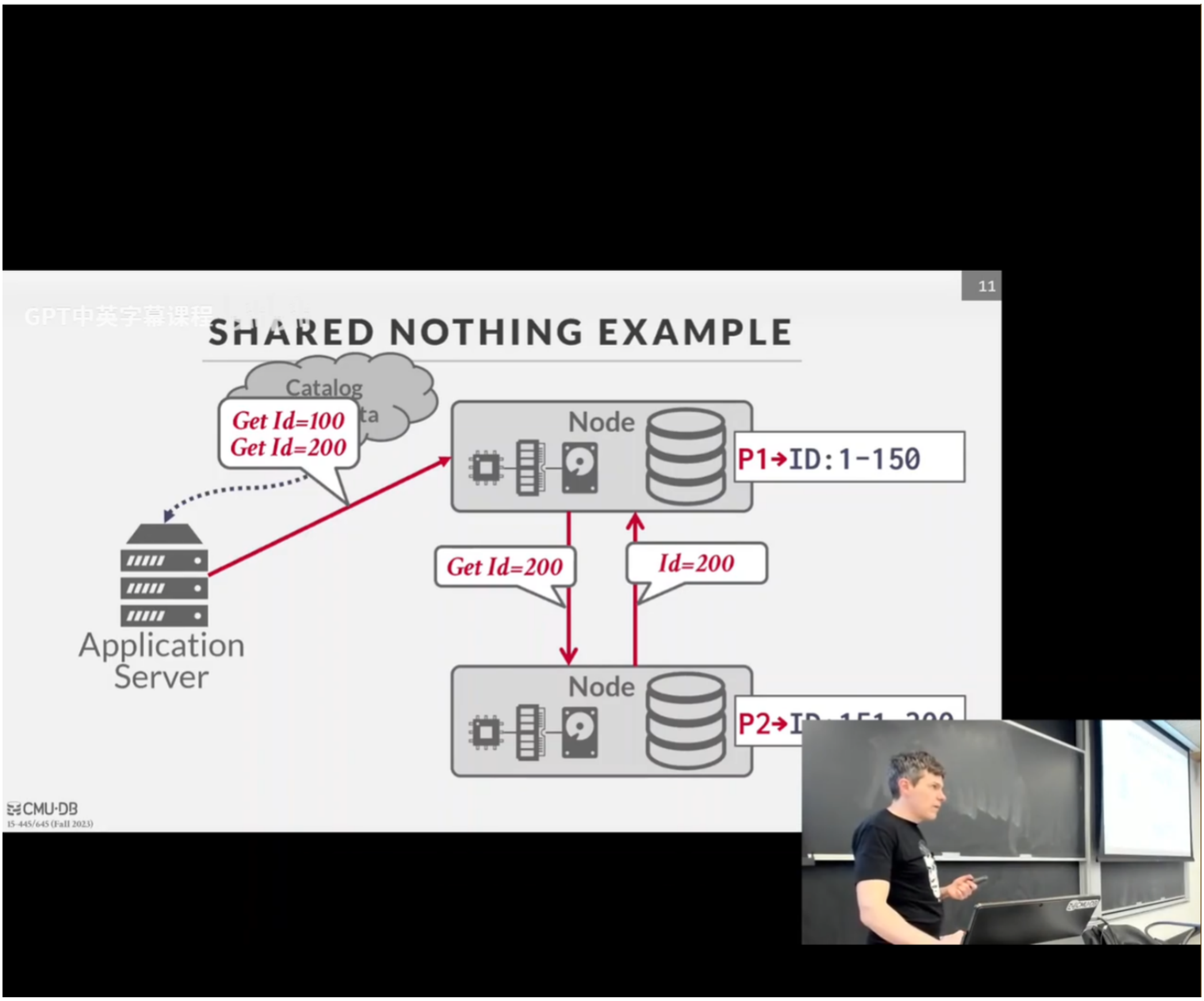
SHARED NOTHING

Each DBMS node has its own CPU, memory, and local disk.

Nodes only communicate with each other via network.
→ Better performance & efficiency.
→ Harder to scale capacity.
→ Harder to ensure consistency.

MU-DB
645 (Fall 2023)

同样，这里的概念是，集群中的每个节点都无法访问
其他节点的内存或磁盘，它们只能通过某种网络进行通信。
Each node can't view the memory or disk of any other node in the cluster, and they



SHARED NOTHING EXAMPLE

Catalog
ta

Get Id=100
Get Id=200

Application Server

Node

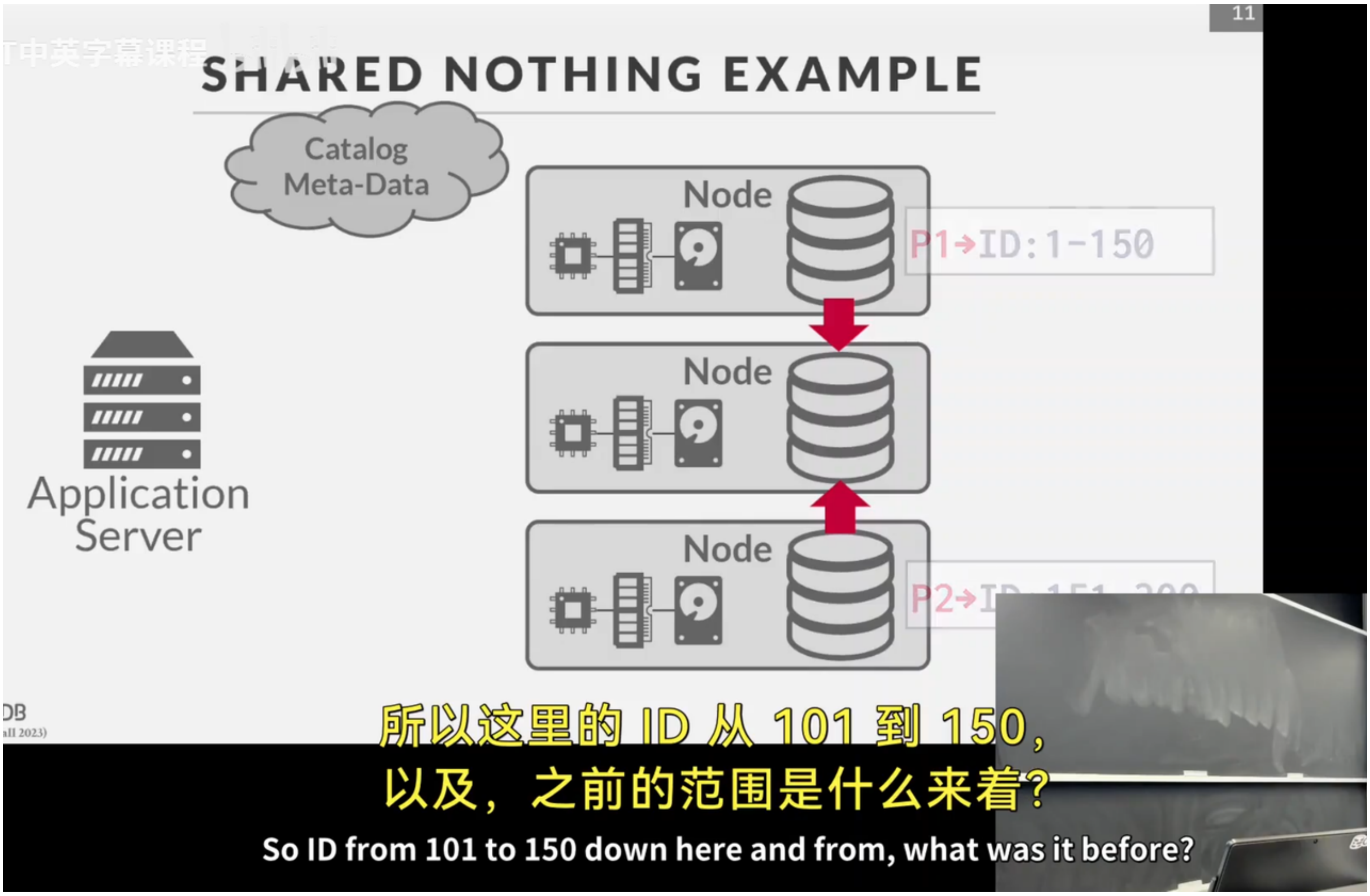
P1→ID:1-150

Get Id=200

Id=200

Node

P2→I



11

SHARED NOTHING EXAMPLE

Catalog
Meta-Data

Node

Node

Node

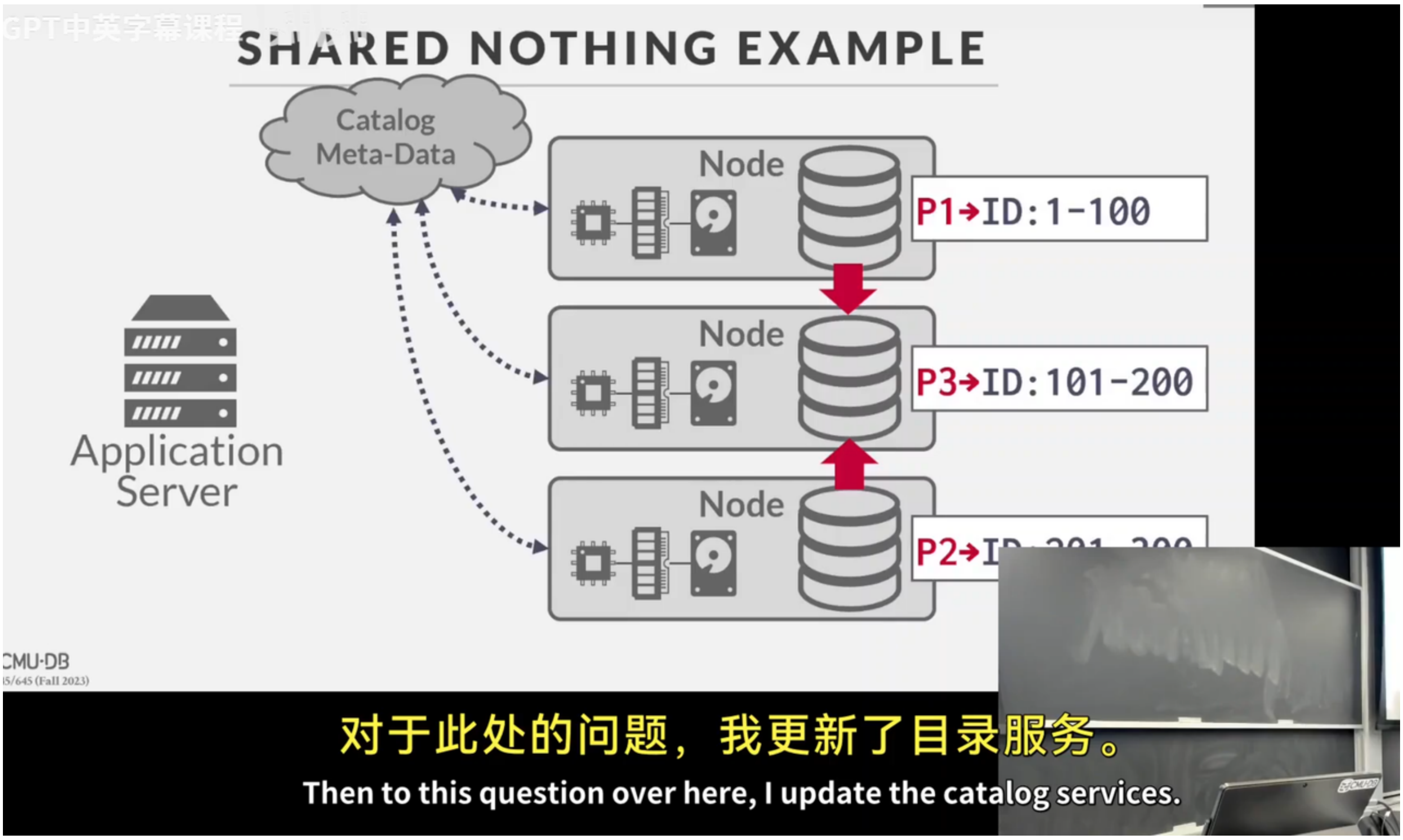
P1→ID:1-150

P2→I

Application
Server

所以这里的 ID 从 101 到 150，
以及，之前的范围是什么来着？
So ID from 101 to 150 down here and from, what was it before?

> MongoDB早期没有按照事务安全的方式移动数据；没有保证节点数据和元 数据的一致性



SHARED NOTHING EXAMPLE

Catalog
Meta-Data

Node

Node

Node

P1→ID:1-100

P3→ID:101-200

P2→I

Application
Server

CMU·DB

对于此处的问题，我更新了目录服务。
Then to this question over here, I update the catalog services.

Shared disk

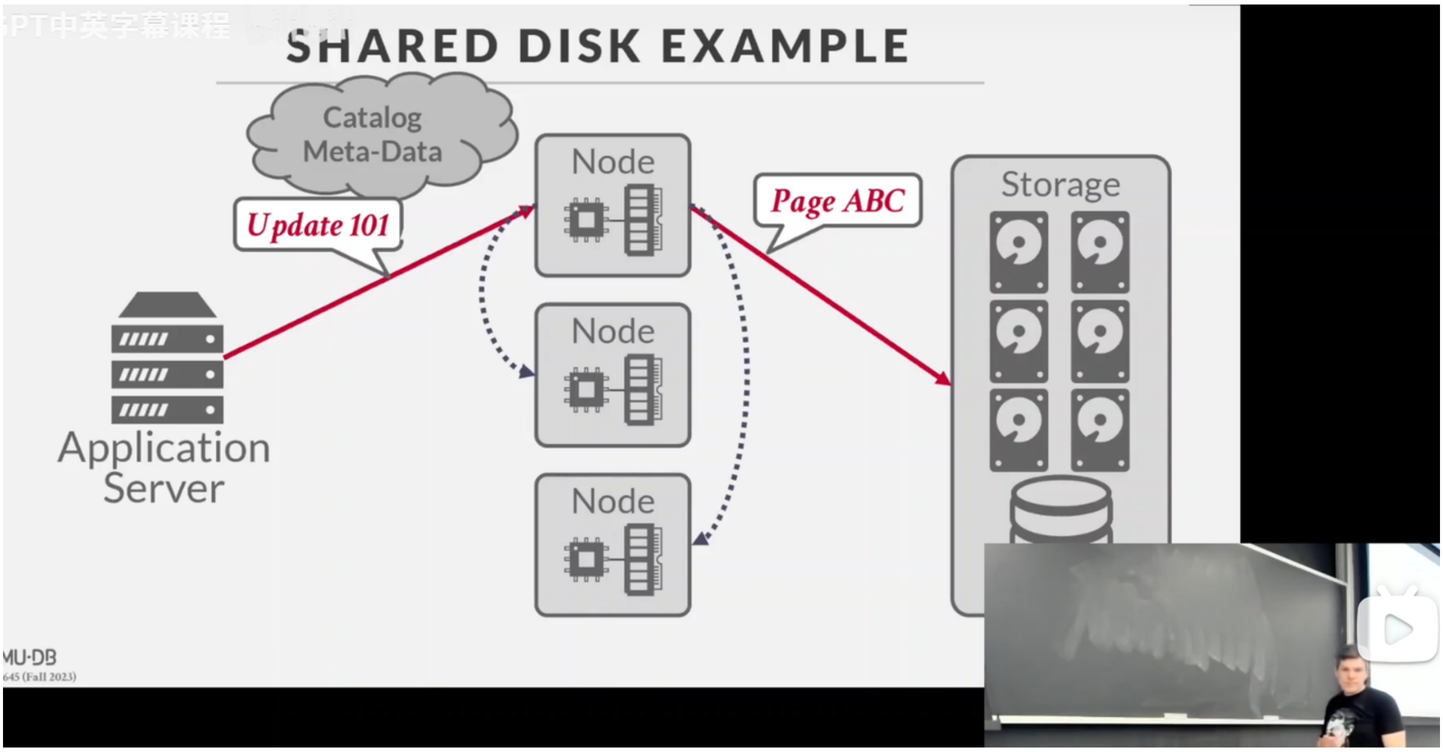
SHARED DISK

Nodes access a single logical disk via an interconnect, but each have their own private memories.

- → Scale execution layer independently from the storage layer.
- → Nodes can still use direct attached storage as a slower/larger cache.
- → This architecture facilitates **data lakes** and **serverless** systems.

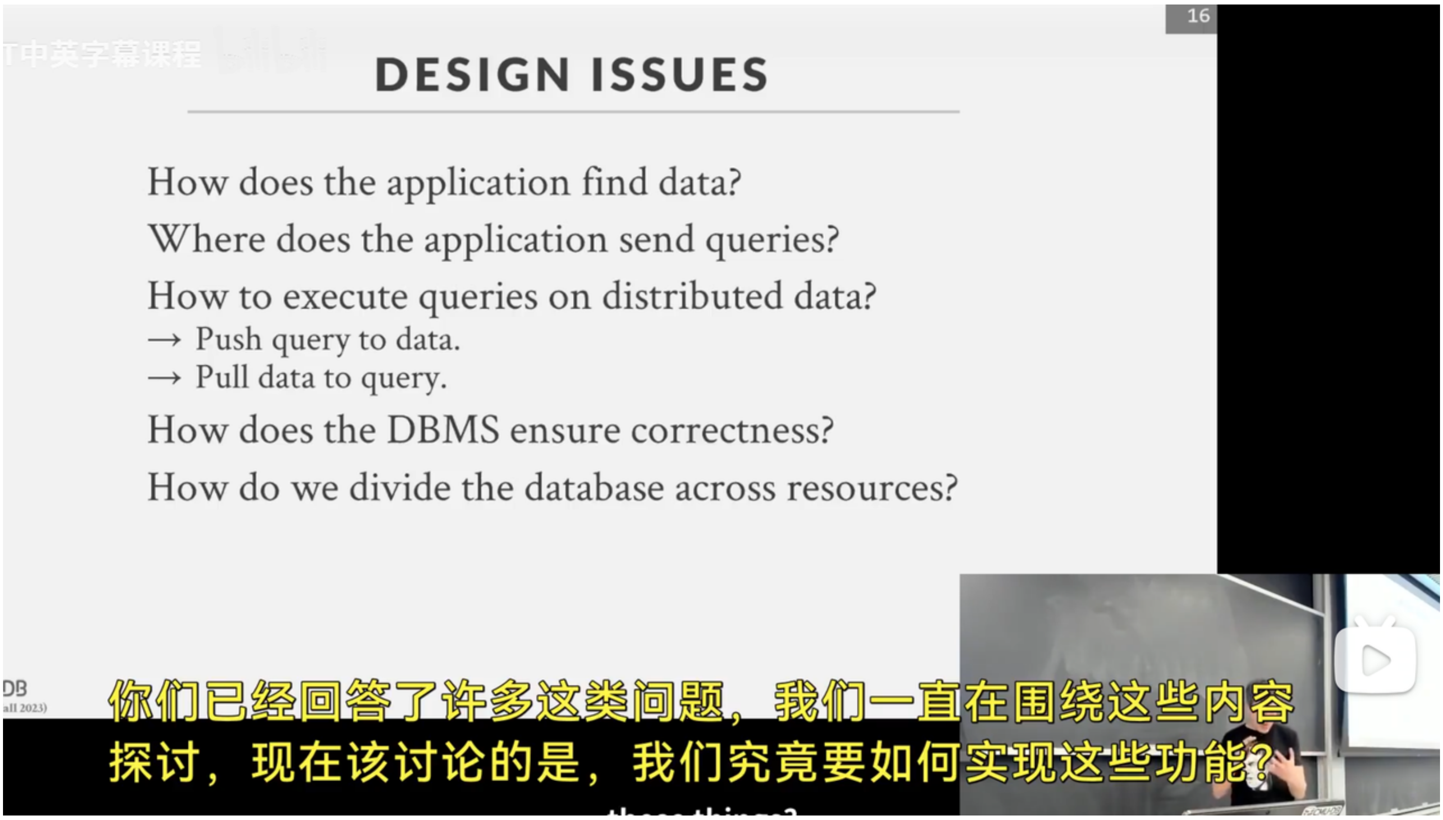
采用这种方法的显著优势在于，你现在可以独立地扩展数据系统的两个部分。

计算节点横向扩展很方便，计算节点可以随时关闭，不会影响db



SHARED DISK EXAMPLE

问题



DESIGN ISSUES

How does the application find data?

Where does the application send queries?

How to execute queries on distributed data?
→ Push query to data.
→ Pull data to query.

How does the DBMS ensure correctness?

How do we divide the database across resources?

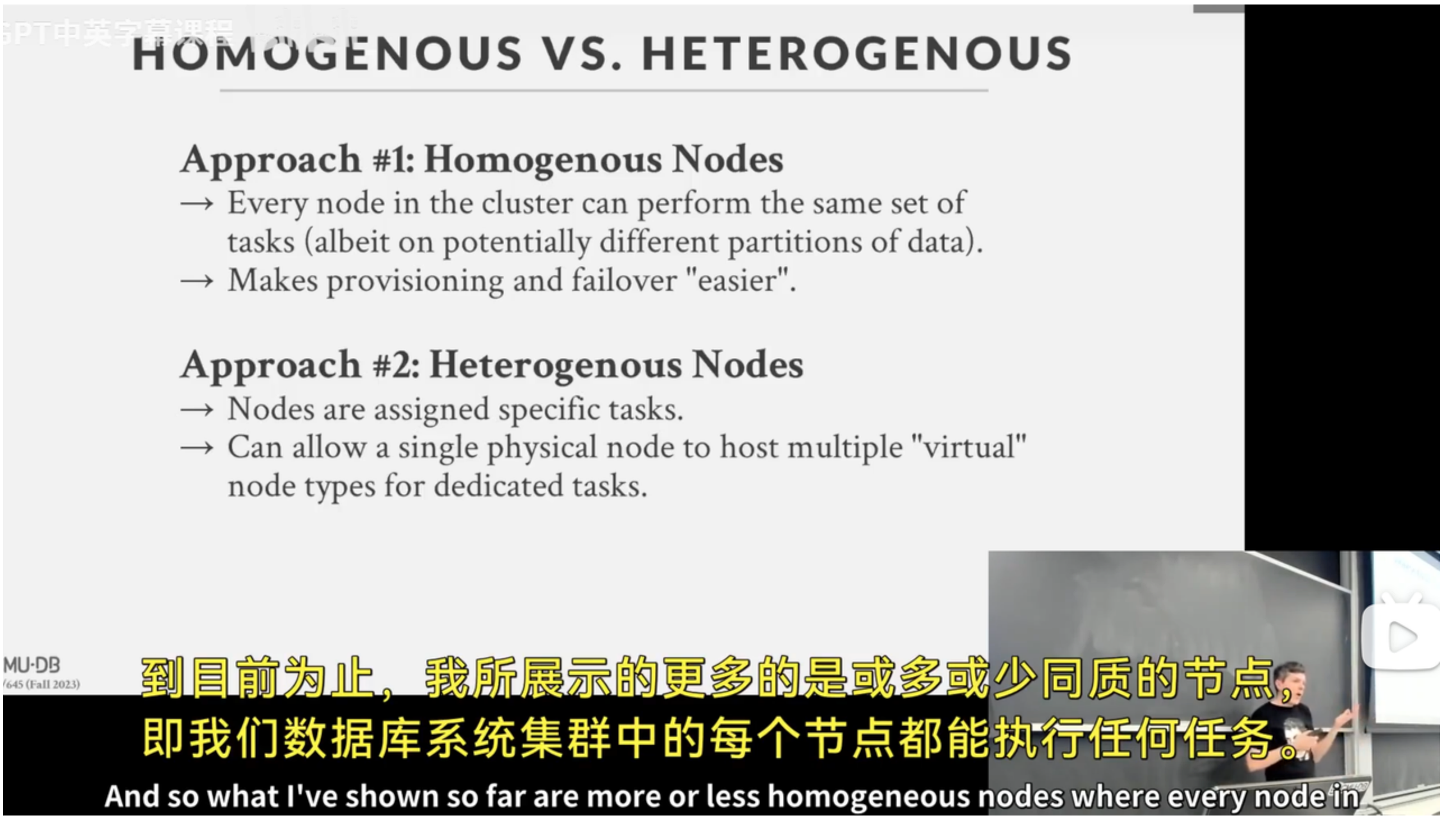
你们已经回答了许多这类问题，我们一直在围绕这些内容探讨，现在该讨论的是，我们究竟要如何实现这些功能？

these things?

同构节点和异构节点

> 同构节点：集群中每个节点都可以承担相同类型的任务；如果某个节点宕机，重新启动一个新节点可以与剩余节点无缝衔接

> 异构节点：每个节点被分配给不同的任务；可以让一个物理节点拥有多个虚拟节点（一致性哈希）



HOMOGENOUS VS. HETEROGENOUS

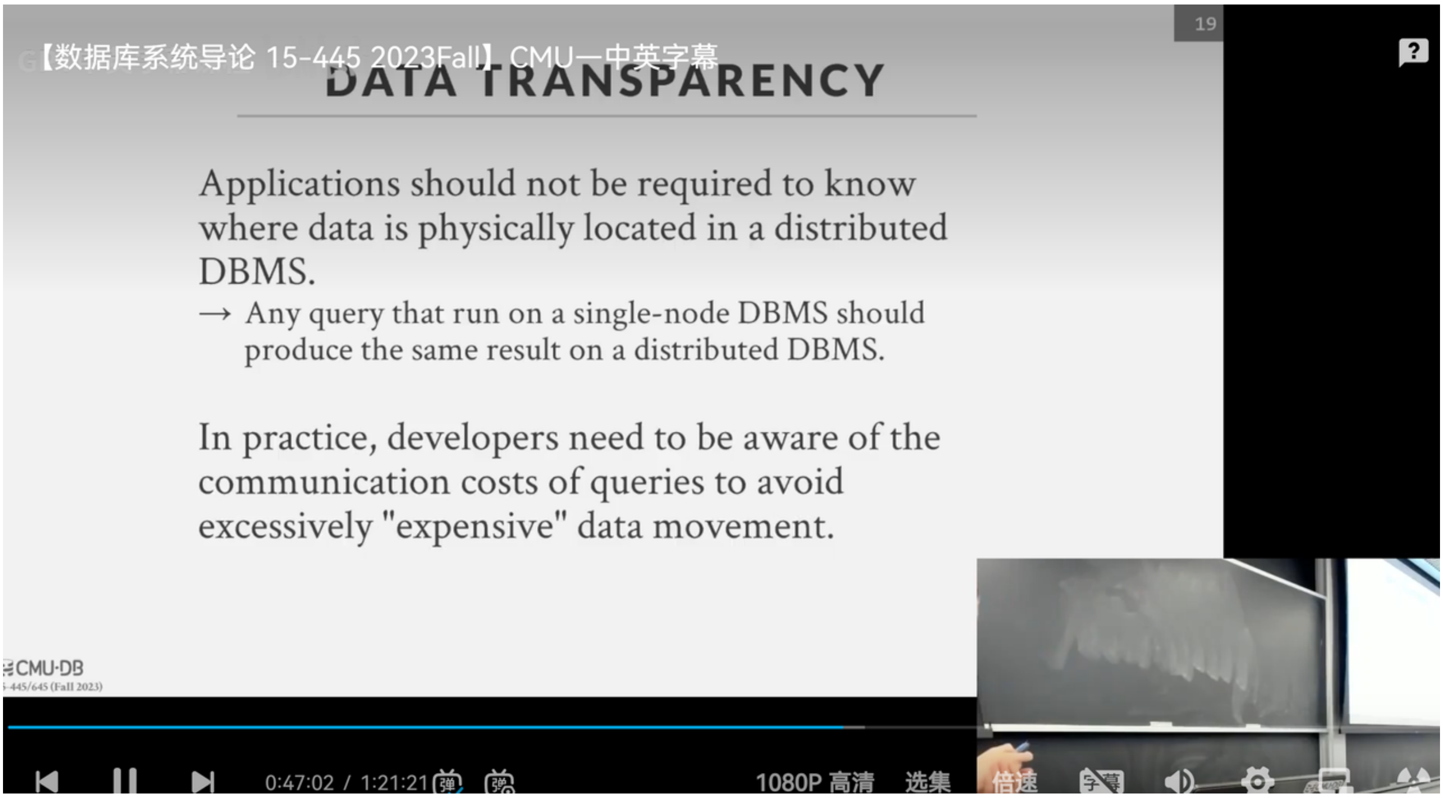
Approach #1: Homogenous Nodes
→ Every node in the cluster can perform the same set of tasks (albeit on potentially different partitions of data).
→ Makes provisioning and failover "easier".

Approach #2: Heterogenous Nodes
→ Nodes are assigned specific tasks.
→ Can allow a single physical node to host multiple "virtual" node types for dedicated tasks.

到目前为止，我所展示的更多的是或多或少同质的节点，即我们数据库系统集群中的每个节点都能执行任何任务。

And so what I've shown so far are more or less homogeneous nodes where every node in

> 我们应该尽可能避免查询间的数据移动开销



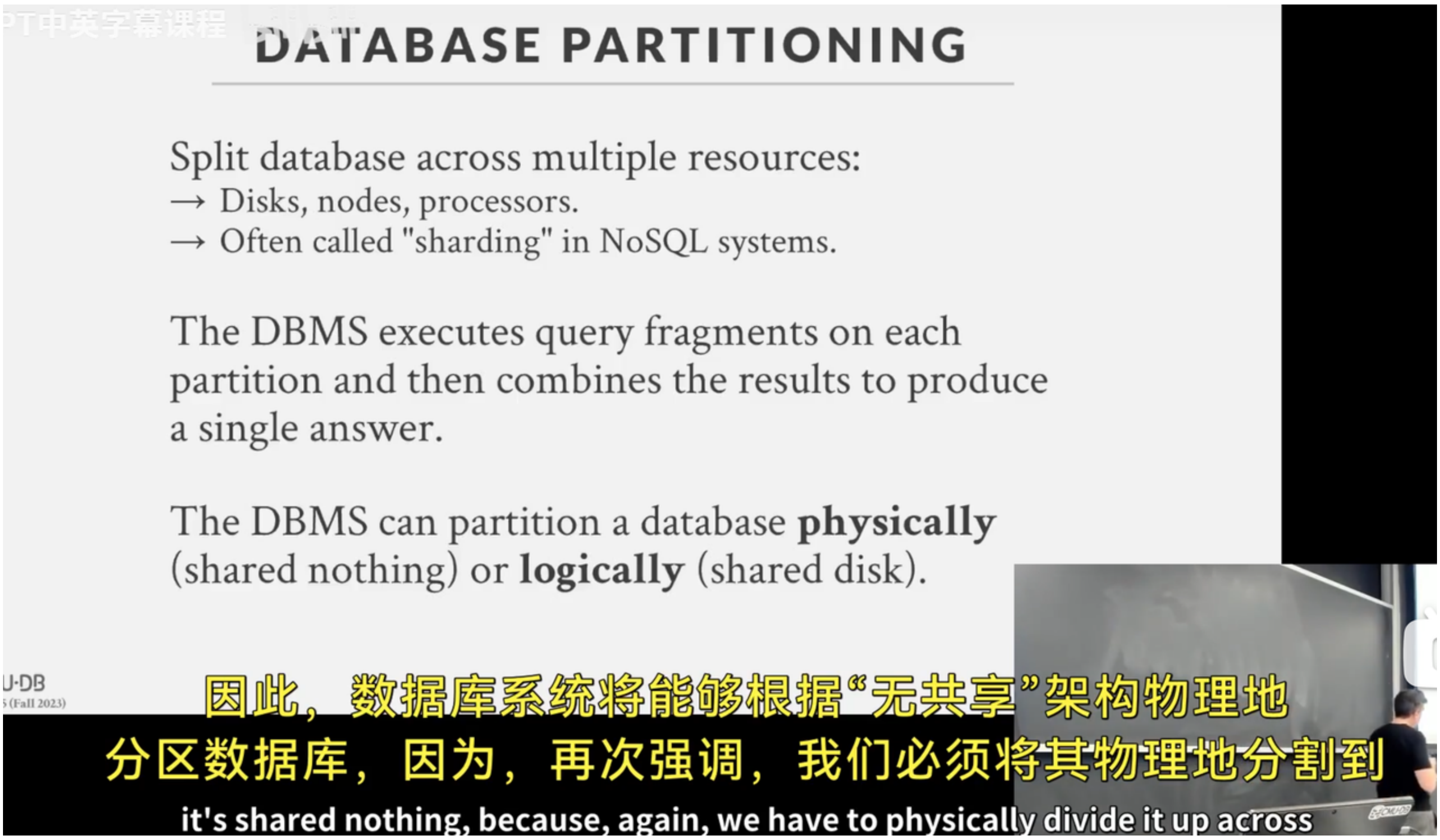
DATA TRANSPARENCY

Applications should not be required to know where data is physically located in a distributed DBMS.

→ Any query that run on a single-node DBMS should produce the same result on a distributed DBMS.

In practice, developers need to be aware of the communication costs of queries to avoid excessively "expensive" data movement.

数据分区(Partitioning/Sharding)



DATABASE PARTITIONING

Split database across multiple resources:

→ Disks, nodes, processors.

→ Often called "sharding" in NoSQL systems.

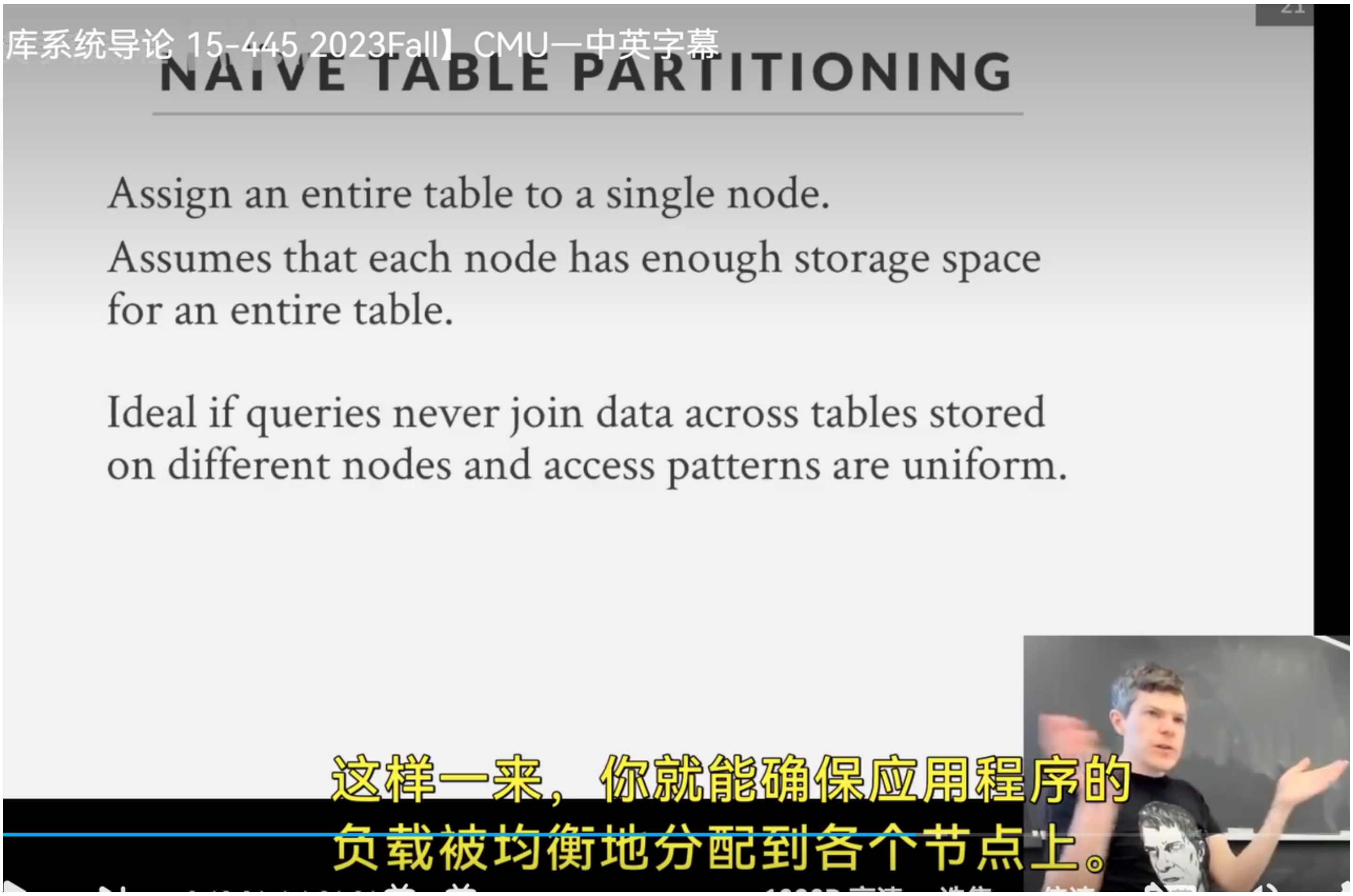
The DBMS executes query fragments on each partition and then combines the results to produce a single answer.

The DBMS can partition a database **physically** (shared nothing) or **logically** (shared disk).

因此，数据库系统将能够根据“无共享”架构物理地分区数据库，因为，再次强调，我们必须将其物理地分割到

it's shared nothing, because, again, we have to physically divide it up across

Navie table partitioning



库系统导论 15-445 2023Fall】CMU—中英字幕

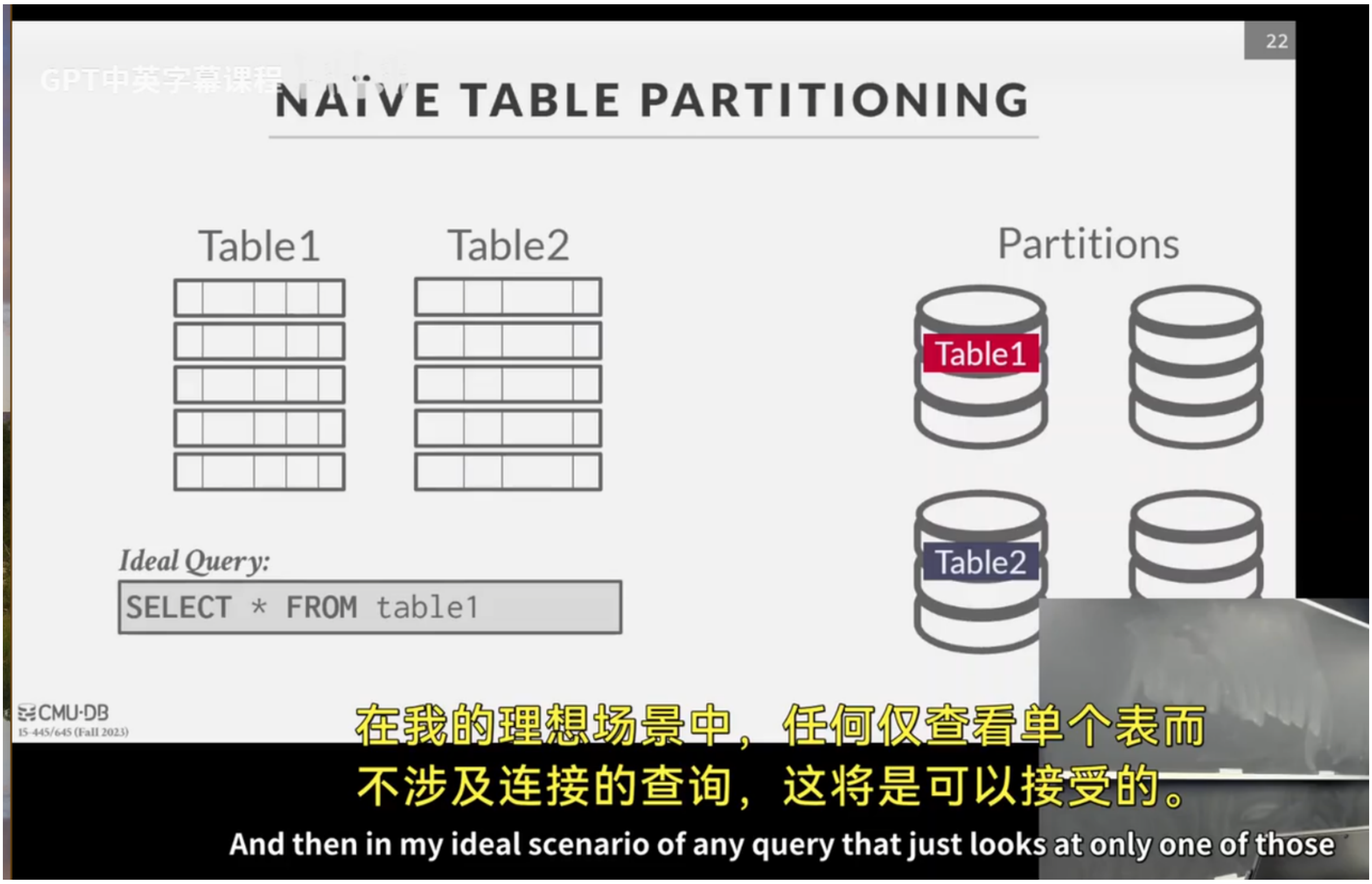
NAÏVE TABLE PARTITIONING

Assign an entire table to a single node.

Assumes that each node has enough storage space for an entire table.

Ideal if queries never join data across tables stored on different nodes and access patterns are uniform.

这样一来，你就能确保应用程序的
负载被均衡地分配到各个节点上。



NAÏVE TABLE PARTITIONING

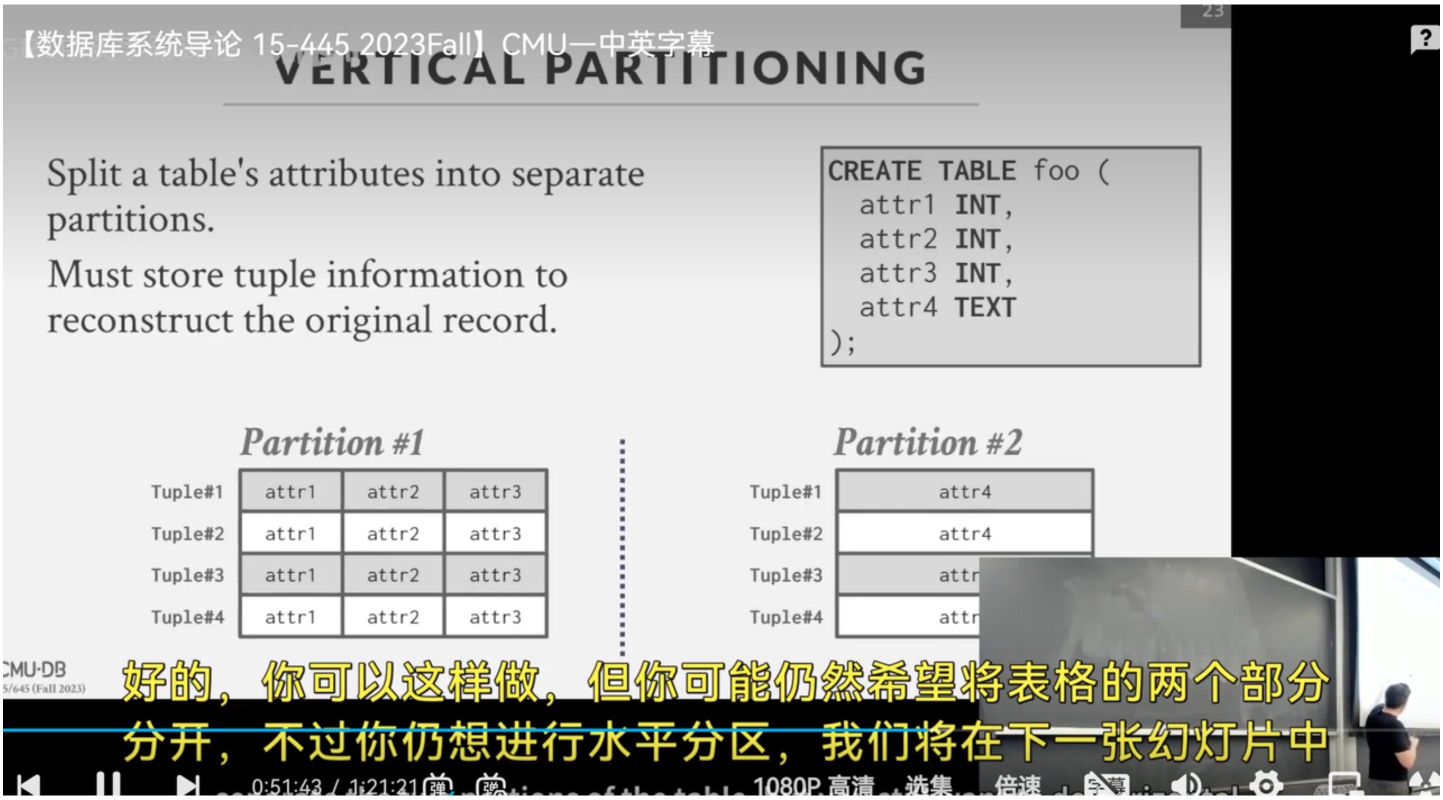
Ideal Query:

```
SELECT * FROM table1
```

在我的理想场景中，任何仅查看单个表而
不涉及连接的查询，这将是可以接受的。
And then in my ideal scenario of any query that just looks at only one of those

Vertical Partitioning

> 类似列存储，将变长的不常访问的属性单独分区



VERTICAL PARTITIONING

Split a table's attributes into separate partitions.

Must store tuple information to reconstruct the original record.

```
CREATE TABLE foo (
  attr1 INT,
  attr2 INT,
  attr3 INT,
  attr4 TEXT
);
```

Partition #1

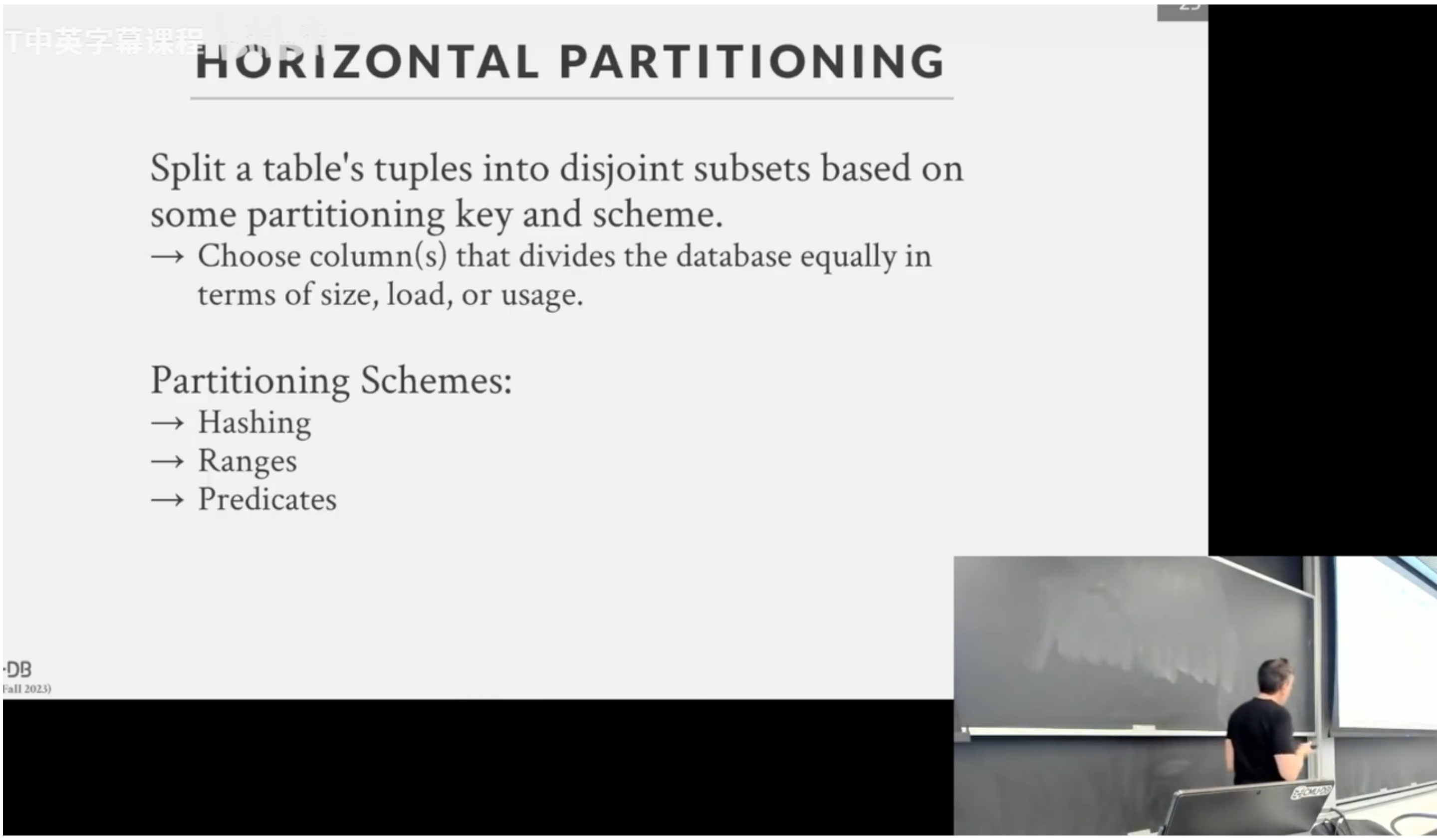
Tuple#1	attr1	attr2	attr3
Tuple#2	attr1	attr2	attr3
Tuple#3	attr1	attr2	attr3
Tuple#4	attr1	attr2	attr3

Partition #2

Tuple#1	attr4
Tuple#2	attr4
Tuple#3	attr
Tuple#4	attr

好的，你可以这样做，但你可能仍然希望将表格的两个部分
分开，不过你仍想进行水平分区，我们将在下一张幻灯片中

Horizontal Partitioning

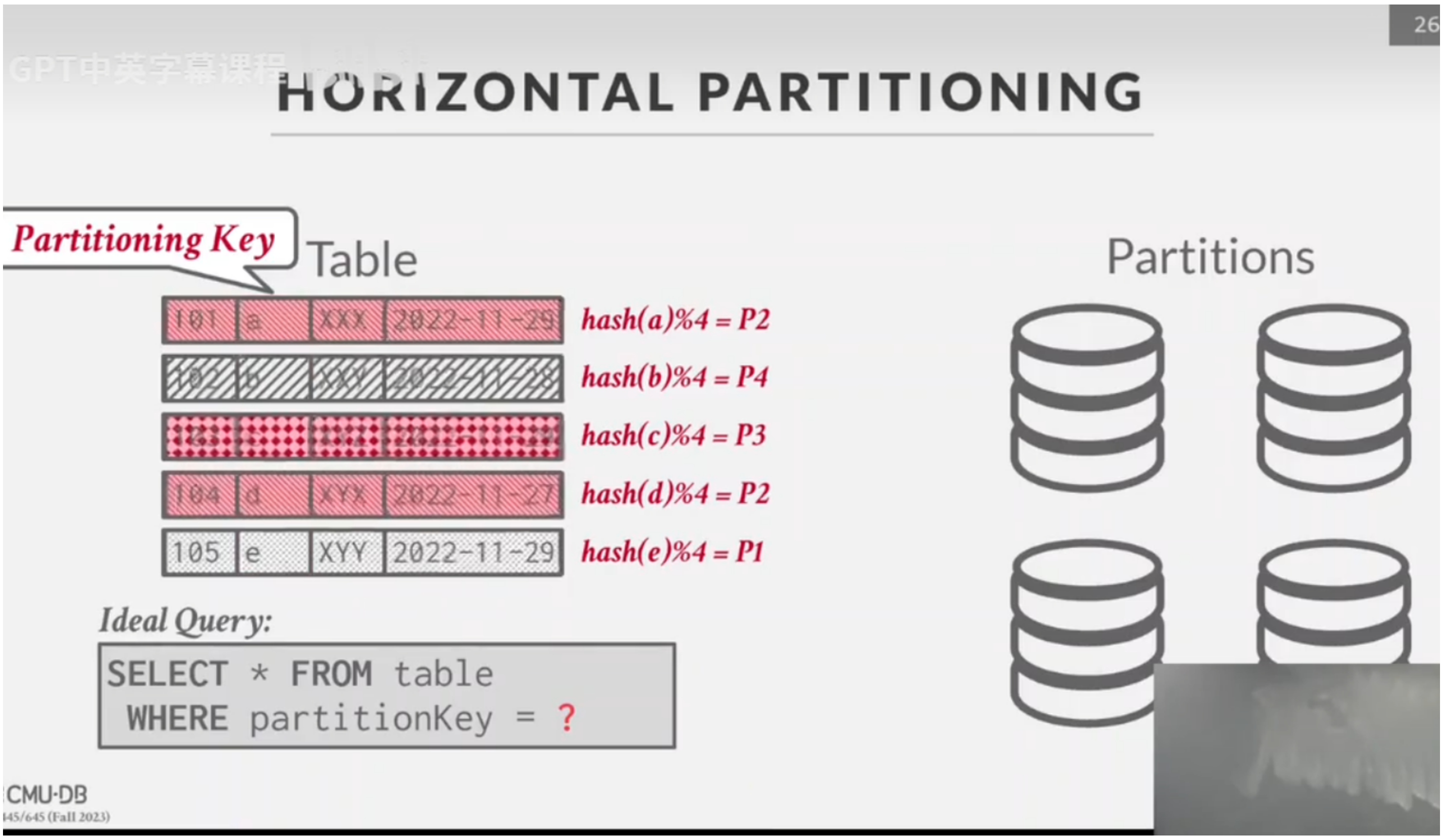


HORIZONTAL PARTITIONING

Split a table's tuples into disjoint subsets based on some partitioning key and scheme.
→ Choose column(s) that divides the database equally in terms of size, load, or usage.

Partitioning Schemes:
→ Hashing
→ Ranges
→ Predicates

> 选择某一列作为分区的依据



HORIZONTAL PARTITIONING

Partitioning Key

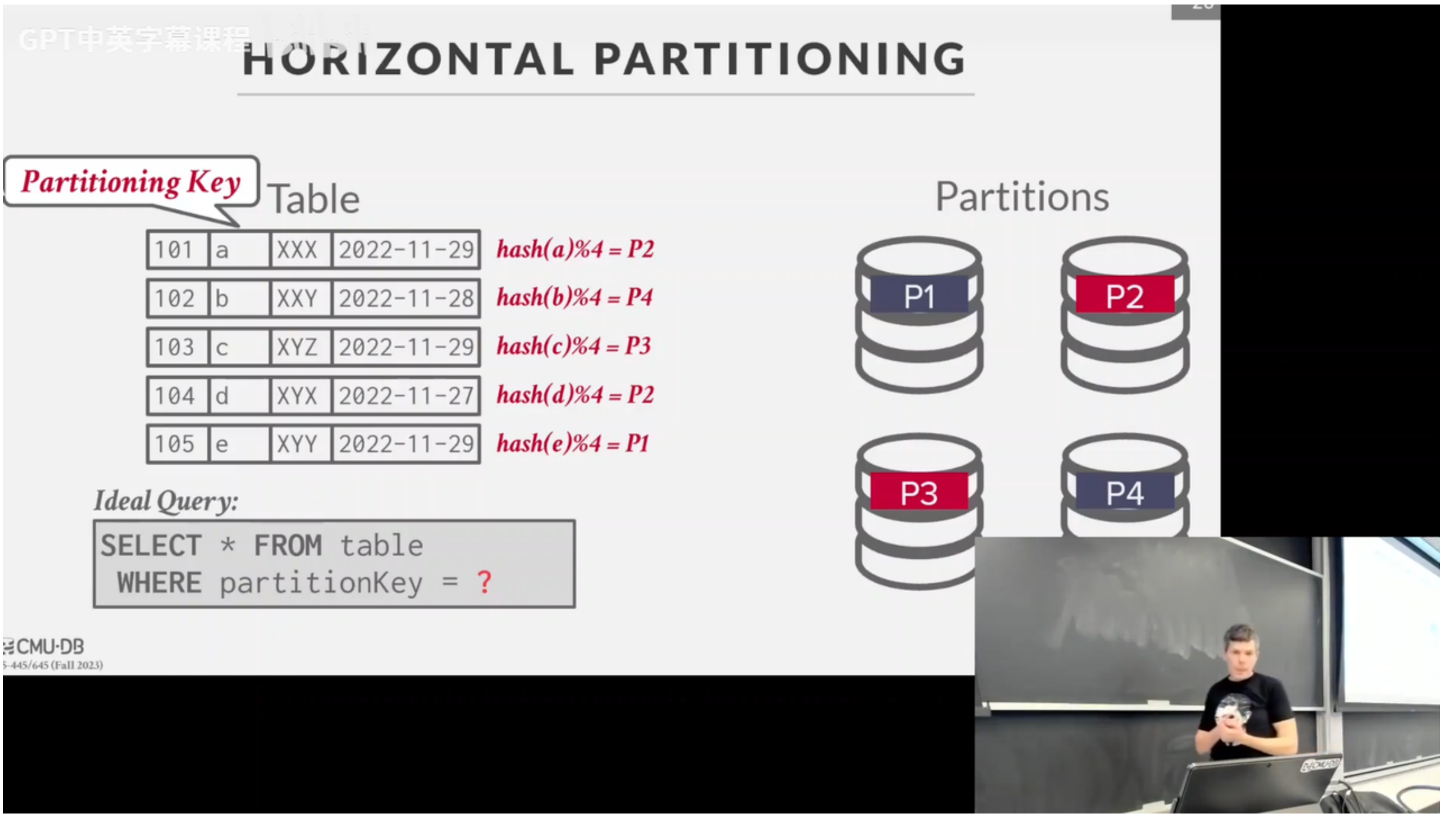
Table

101	a	XXX	2022-11-29	*hash(a)%4 = P2*
102	b	XXY	2022-11-28	*hash(b)%4 = P4*
[illegible]	[illegible]	[illegible]	[illegible]	*hash(c)%4 = P3*
104	d	XYX	2022-11-27	*hash(d)%4 = P2*
105	e	XYY	2022-11-29	*hash(e)%4 = P1*

Partitions

Ideal Query:

```
SELECT * FROM table
 WHERE partitionKey = ?
```



HORIZONTAL PARTITIONING

Partitioning Key

Table

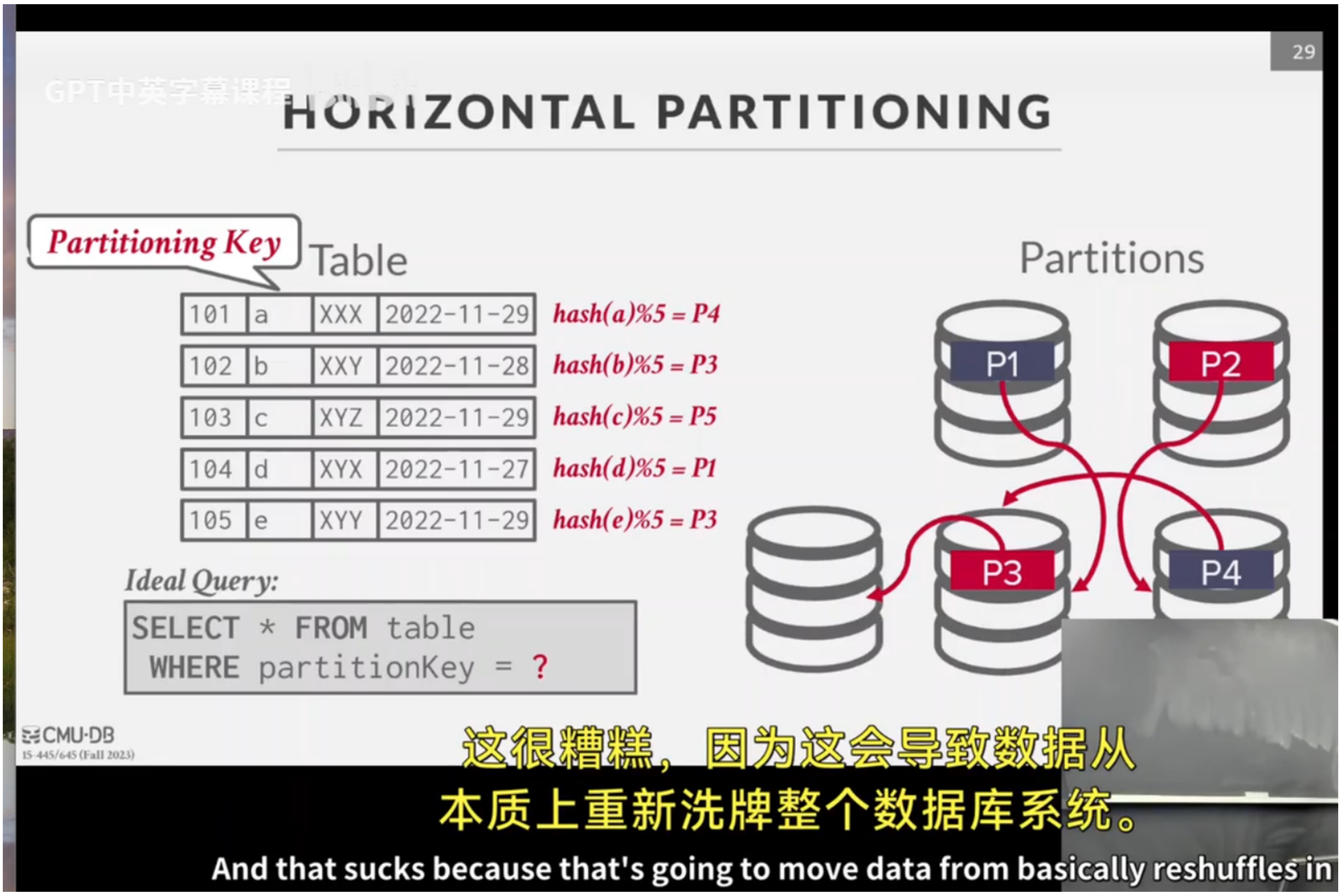
101	a	XXX	2022-11-29	*hash(a)%4 = P2*
102	b	XXY	2022-11-28	*hash(b)%4 = P4*
103	c	XYZ	2022-11-29	*hash(c)%4 = P3*
104	d	XYX	2022-11-27	*hash(d)%4 = P2*
105	e	XYY	2022-11-29	*hash(e)%4 = P1*

Partitions

Ideal Query:

```
SELECT * FROM table
 WHERE partitionKey = ?
```

> hash分区新增节点会导致数据重新分配，消耗大量资源
>
> 范围分区在这种情况下有优势



HORIZONTAL PARTITIONING

Partitioning Key

Table

101	a	XXX	2022-11-29	*hash(a)%5 = P4*
102	b	XXY	2022-11-28	*hash(b)%5 = P3*
103	c	XYZ	2022-11-29	*hash(c)%5 = P5*
104	d	XYX	2022-11-27	*hash(d)%5 = P1*
105	e	XYY	2022-11-29	*hash(e)%5 = P3*

Ideal Query:

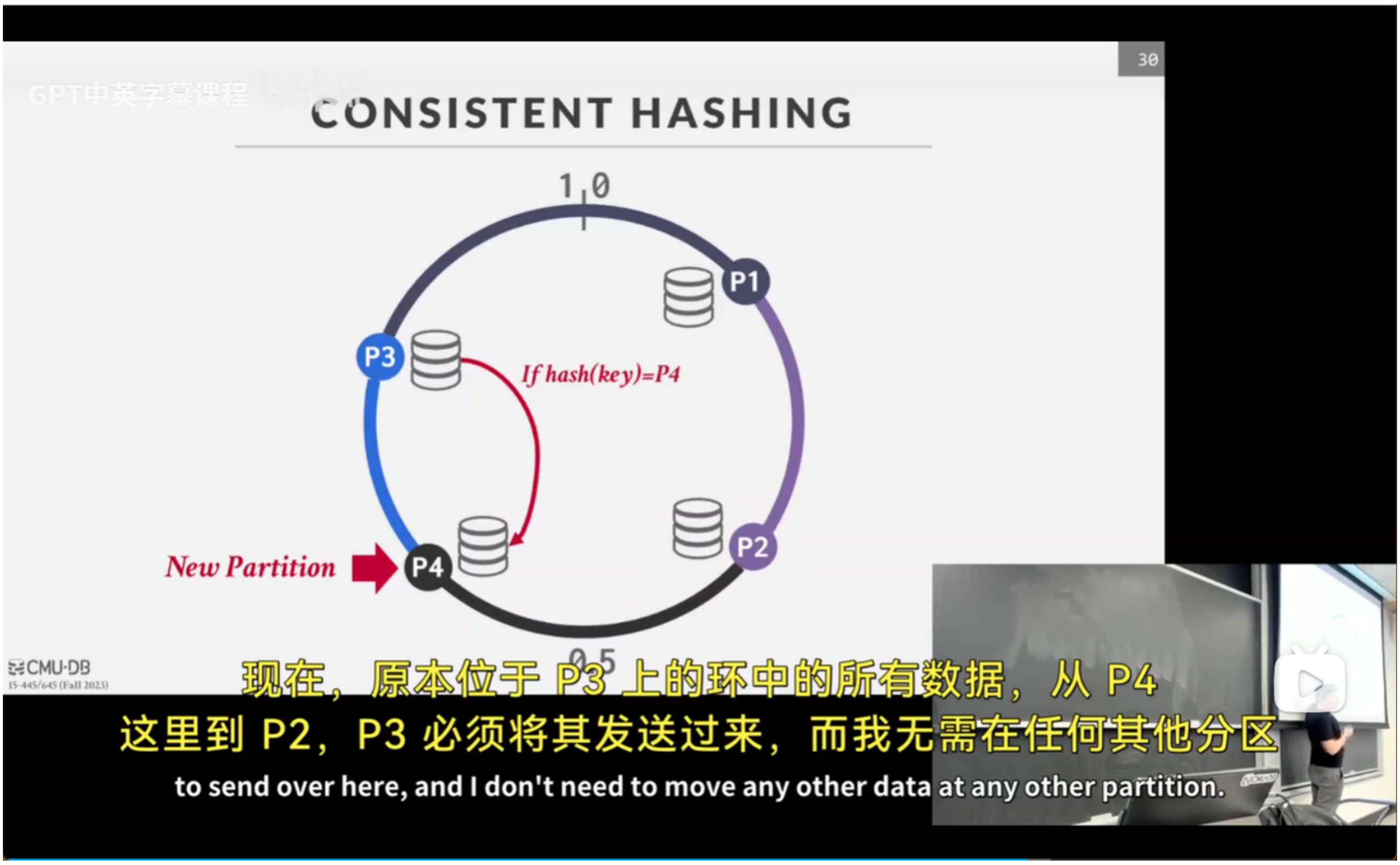
```
SELECT * FROM table
 WHERE partitionKey = ?
```

CMU·DB
15-445/645 (Fall 2023)

这很糟糕，因为这会导致数据从
本质上重新洗牌整个数据库系统。
And that sucks because that's going to move data from basically reshuffles in

Consistent hashing

> 当新增节点时，我们只需要将临近节点上的数据分配给新节点即可

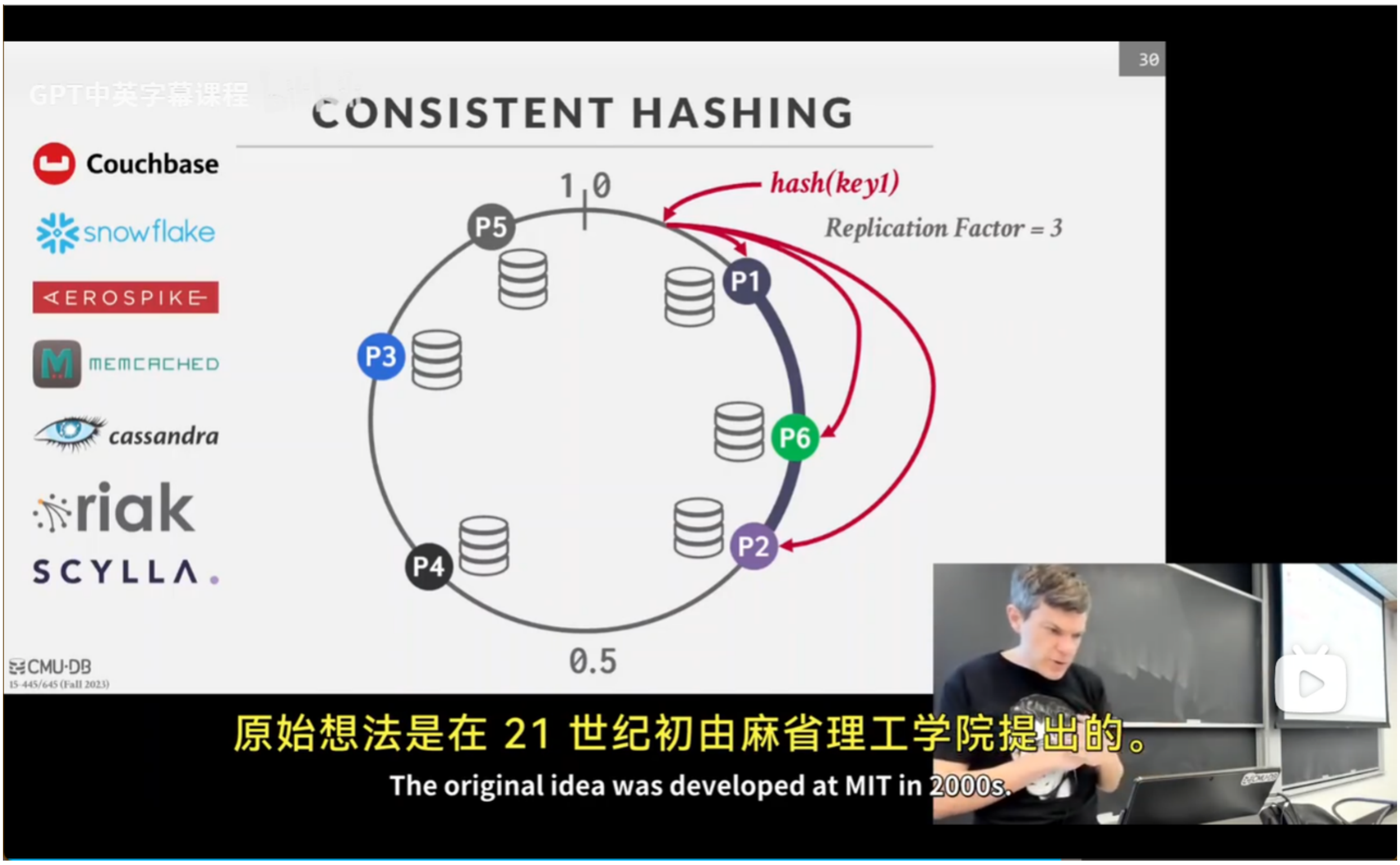


CONSISTENT HASHING

CMU·DB
15-445/645 (Fall 2023)

现在，原本位于 P3 上的环中的所有数据，从 P4
这里到 P2，P3 必须将其发送过来，而我无需在任何其他分区
to send over here, and I don't need to move any other data at any other partition.

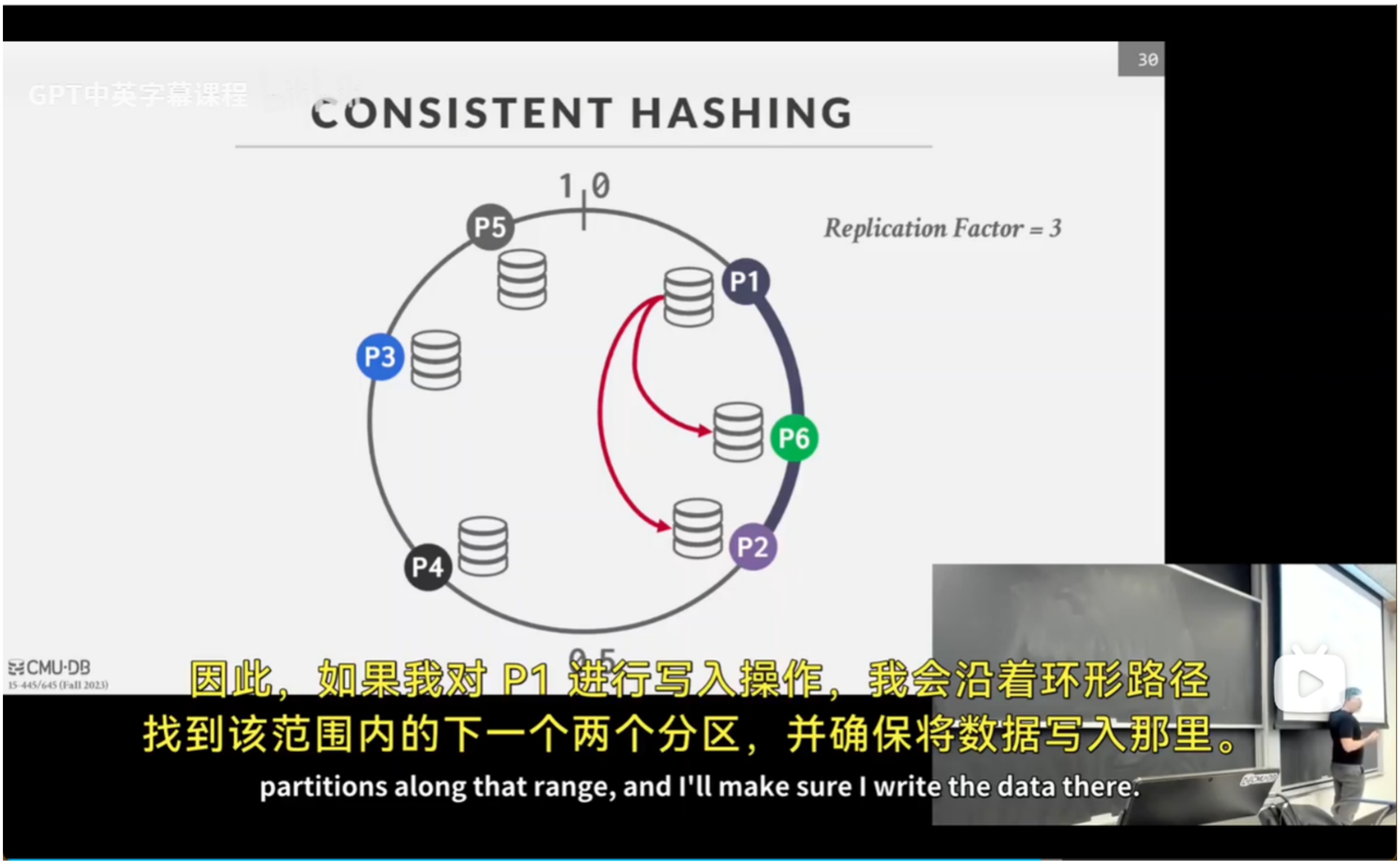
> 这里的replication实际上是在环上将数据复制到相邻的接下来的replication factor - 1个节点上


CONSISTENT HASHING

Replication Factor = 3

原始想法是在 21 世纪初由麻省理工学院提出的。

The original idea was developed at MIT in 2000s.


CONSISTENT HASHING

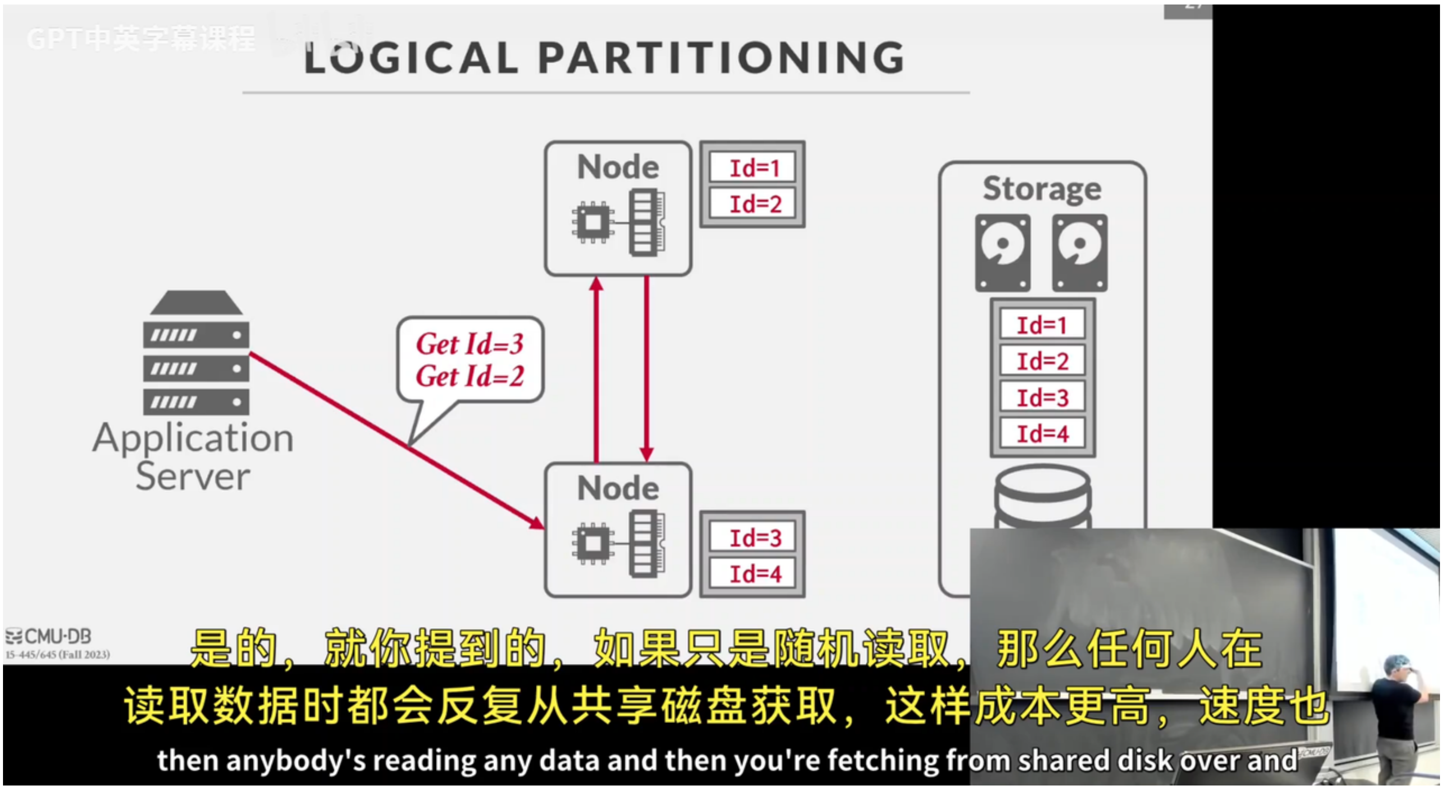
Replication Factor = 3

因此，如果我对 P1 进行写入操作，我会沿着环形路径
找到该范围内的下一个两个分区，并确保将数据写入那里。

partitions along that range, and I'll make sure I write the data there.

Logical Partitioning

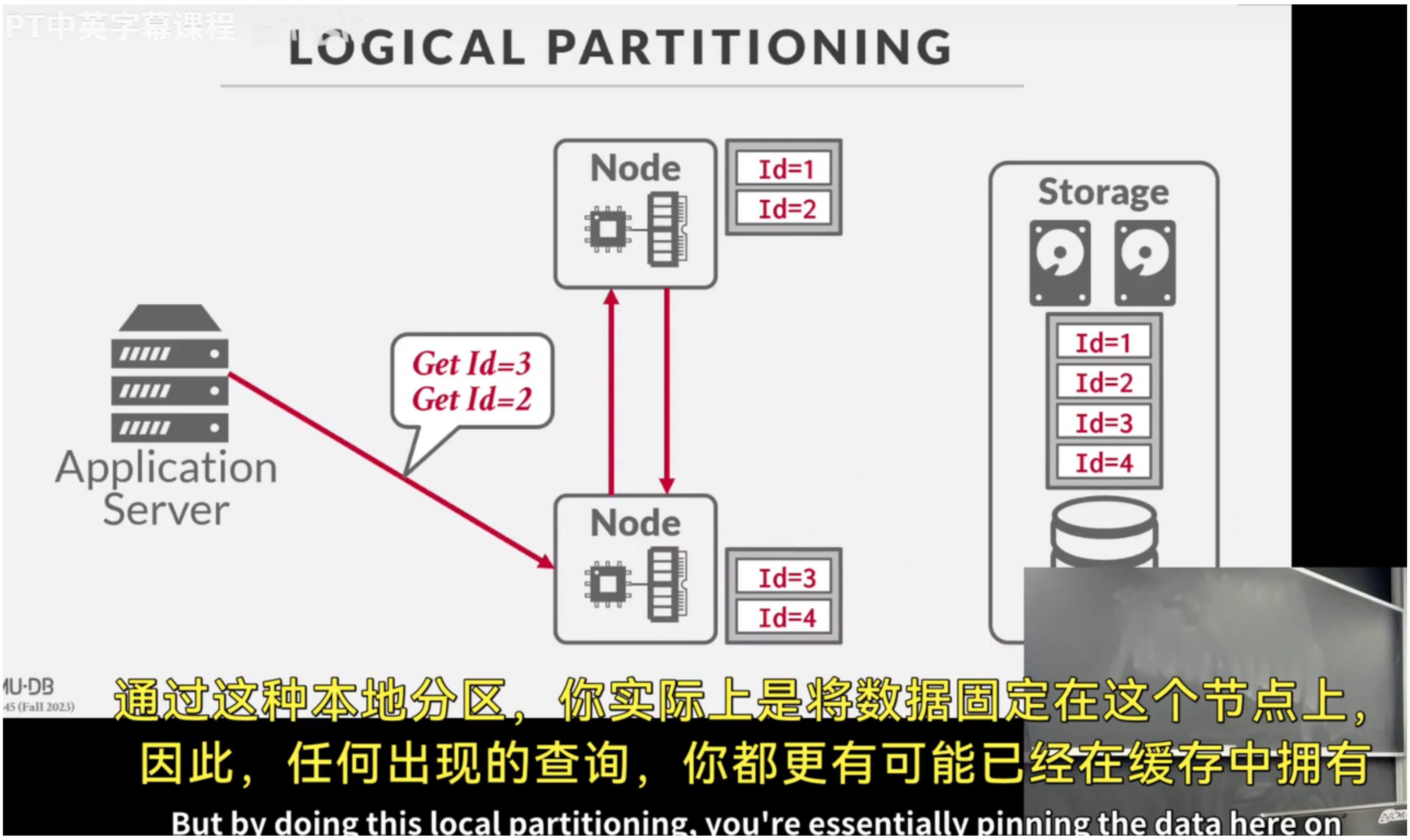
> 在逻辑上分区，而不是在物理上分区，share disk架构



LOGICAL PARTITIONING

是的，就你提到的，如果只是随机读取，那么任何人在读取数据时都会反复从共享磁盘获取，这样成本更高，速度也

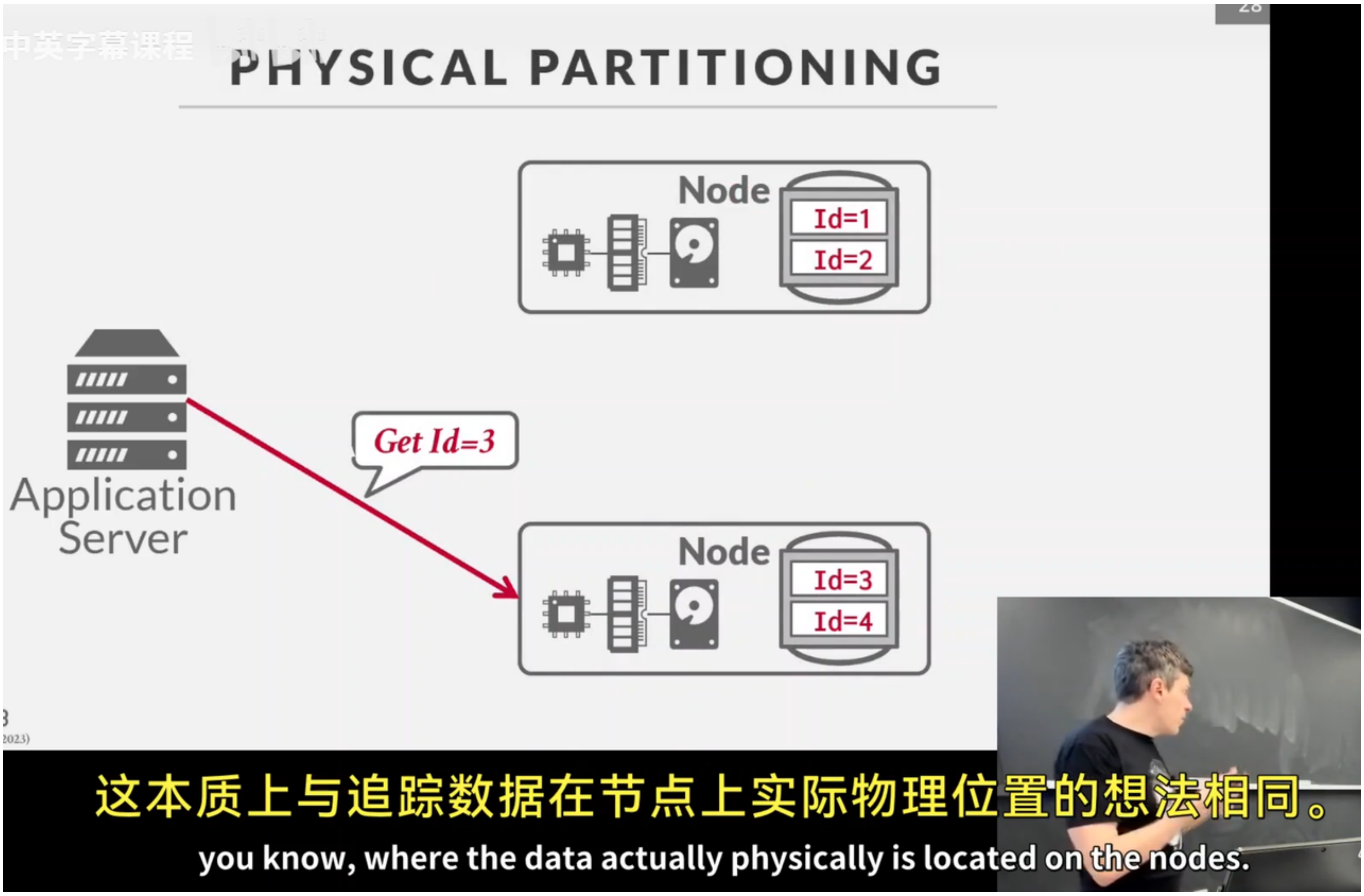
then anybody's reading any data and then you're fetching from shared disk over and



LOGICAL PARTITIONING

通过这种本地分区，你实际上是将数据固定在这个节点上，因此，任何出现的查询，你都更有可能已经在缓存中拥有

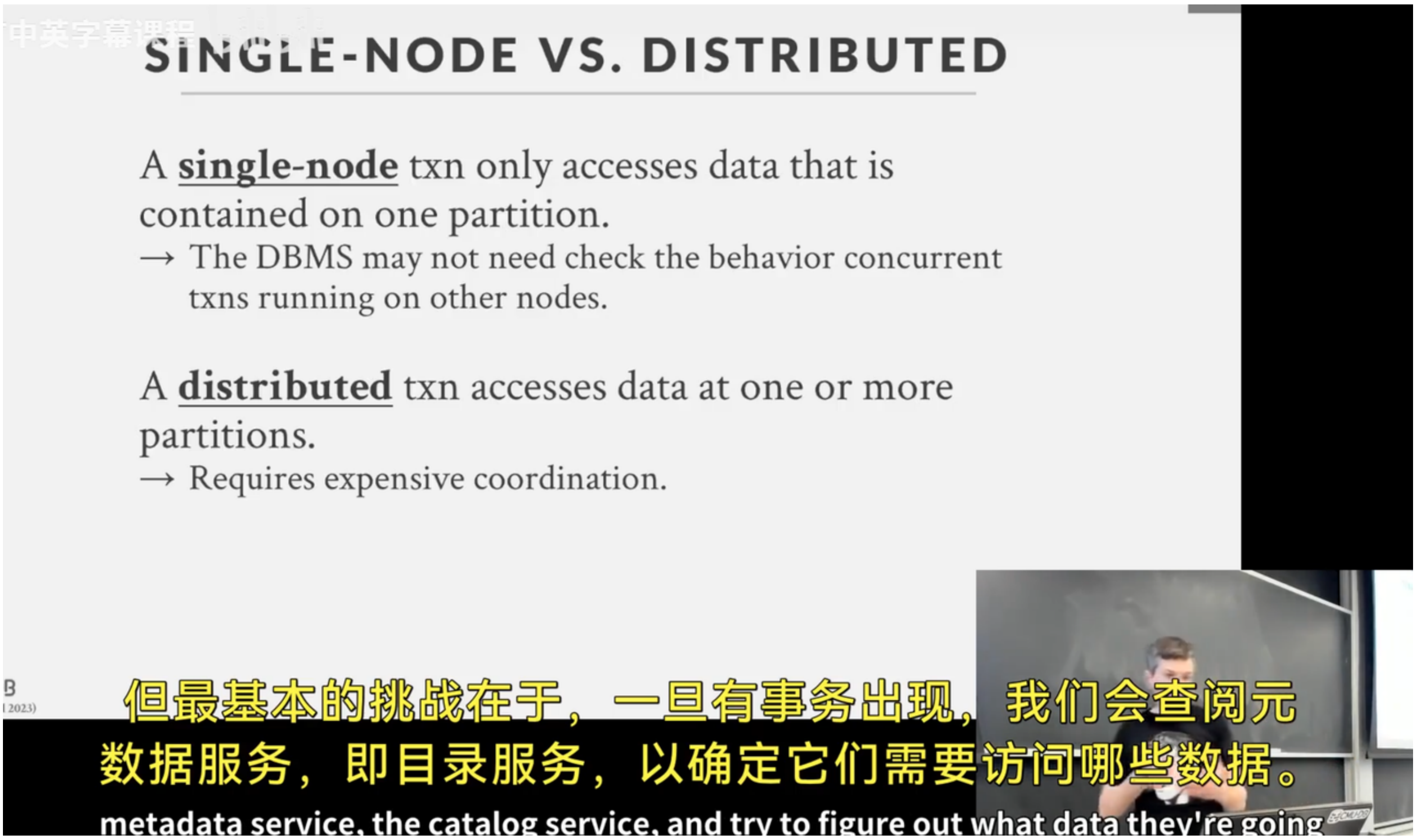
But by doing this local partitioning, you're essentially pinning the data here on



PHYSICAL PARTITIONING

这本质上与追踪数据在节点上实际物理位置的想法相同。

you know, where the data actually physically is located on the nodes.



SINGLE-NODE VS. DISTRIBUTED

A **single-node** txn only accesses data that is contained on one partition.

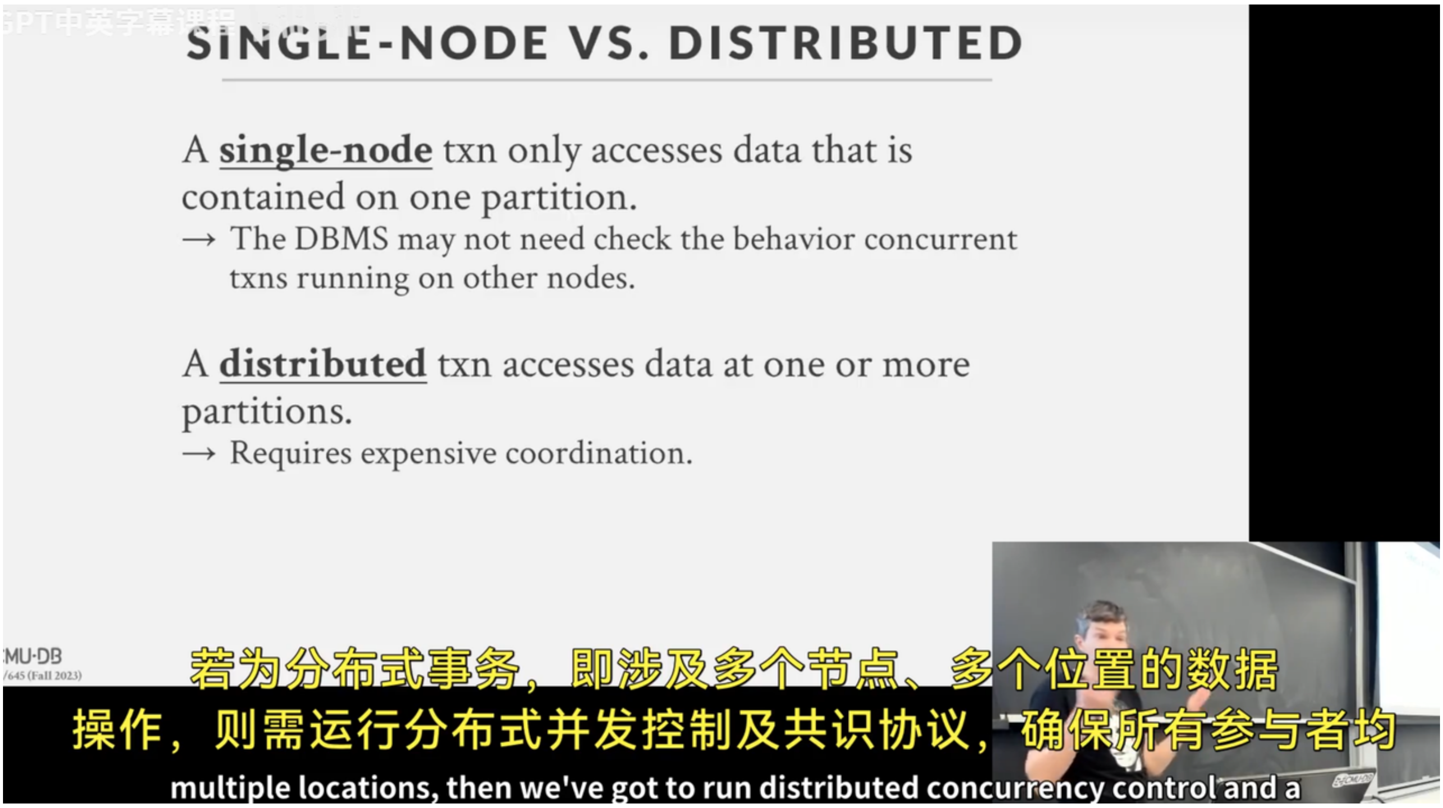
→ The DBMS may not need check the behavior concurrent txns running on other nodes.

A **distributed** txn accesses data at one or more partitions.

→ Requires expensive coordination.

但最基本的挑战在于，一旦有事务出现，我们会查阅元数据服务，即目录服务，以确定它们需要访问哪些数据。

metadata service, the catalog service, and try to figure out what data they're going



SINGLE-NODE VS. DISTRIBUTED

A **single-node** txn only accesses data that is contained on one partition.
→ The DBMS may not need check the behavior concurrent txns running on other nodes.

A **distributed** txn accesses data at one or more partitions.
→ Requires expensive coordination.

若为分布式事务，即涉及多个节点、多个位置的数据操作，则需运行分布式并发控制及共识协议，确保所有参与者均

multiple locations, then we've got to run distributed concurrency control and a

TRANSACTION COORDINATION

If our DBMS supports multi-operation and distributed txns, we need a way to coordinate their execution in the system.

Two different approaches:
→ **Centralized**: Global "traffic cop".
→ **Decentralized**: Nodes organize themselves.

Most distributed DBMSs use a hybrid approach where they periodically elect some node to be a temporary coordinator.

好的，如果我们希望在不同节点上支持多种操作，那么我们同样需要某种方式来协调该事务的执行。

need some way to, again, coordinate the execution of that transaction.

> 去中心化，但是为了控制并发效率，选举出leader节点来充当协调者

TRANSACTION COORDINATION

If our DBMS supports multi-operation and distributed txns, we need a way to coordinate their execution in the system.

Two different approaches:
→ **Centralized**: Global "traffic cop".
→ **Decentralized**: Nodes organize themselves.

Most distributed DBMSs use a hybrid approach where they periodically elect some node to be a temporary coordinator.

大多数分布式数据库采用的是一种混合方法，这种方法是去
中心化的，意味着没有一台专门的机器或节点来充当交通指挥官

Most distributed databases are going to use a hybrid approach where it's going to be

TRANSACTION COORDINATION

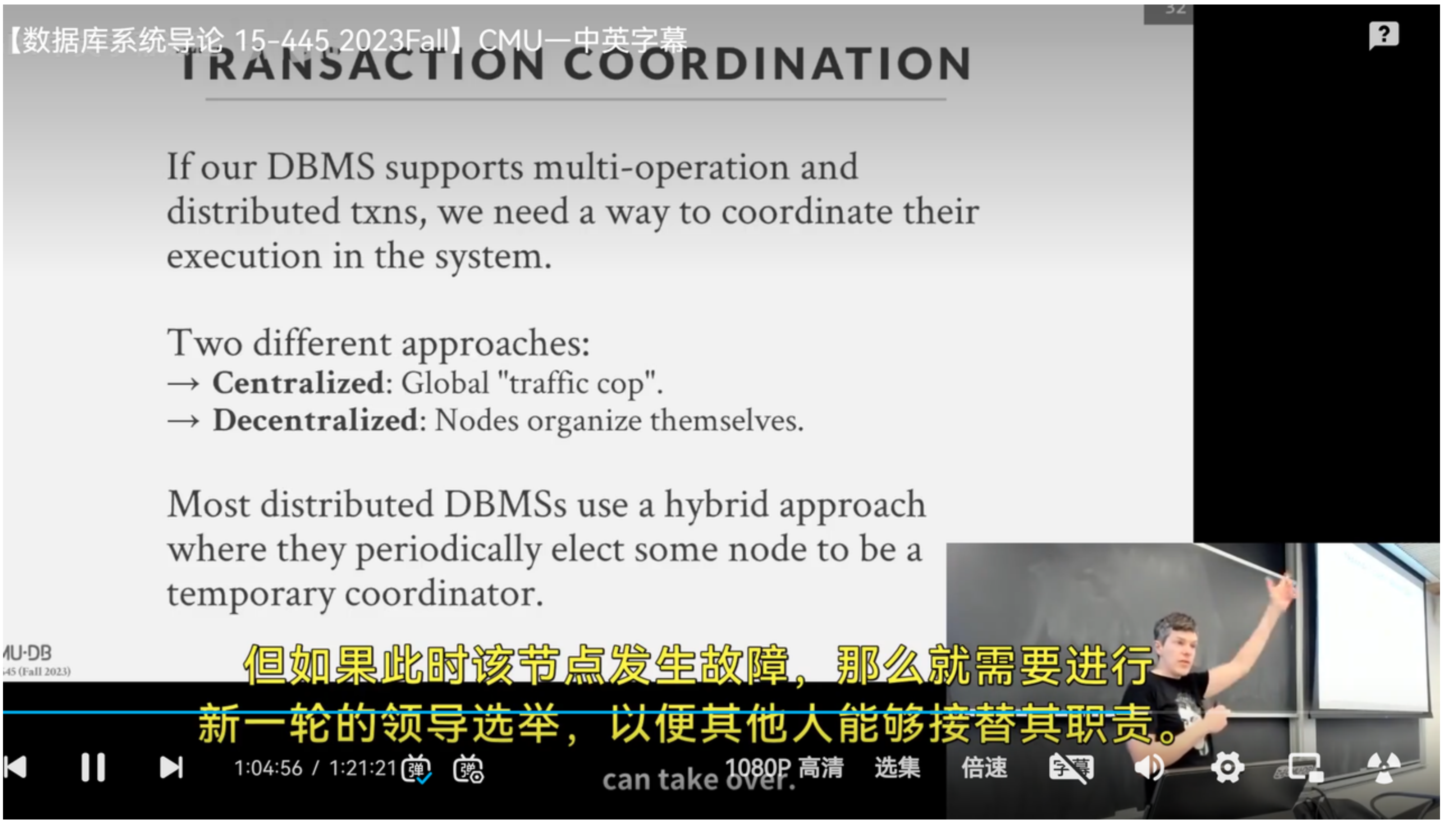
If our DBMS supports multi-operation and distributed txns, we need a way to coordinate their execution in the system.

Two different approaches:
→ **Centralized**: Global "traffic cop".
→ **Decentralized**: Nodes organize themselves.

Most distributed DBMSs use a hybrid approach where they periodically elect some node to be a temporary coordinator.

的角色。但由于去中心化的并发控制效率较低，因此会选举
出一个领导者，暂时担任交通指挥官，即协调者，来决定

cop, but since it's slow to do decentralized concurrency control, they're gonna elect



TRANSACTION COORDINATION

If our DBMS supports multi-operation and distributed txns, we need a way to coordinate their execution in the system.

Two different approaches:
→ **Centralized**: Global "traffic cop".
→ **Decentralized**: Nodes organize themselves.

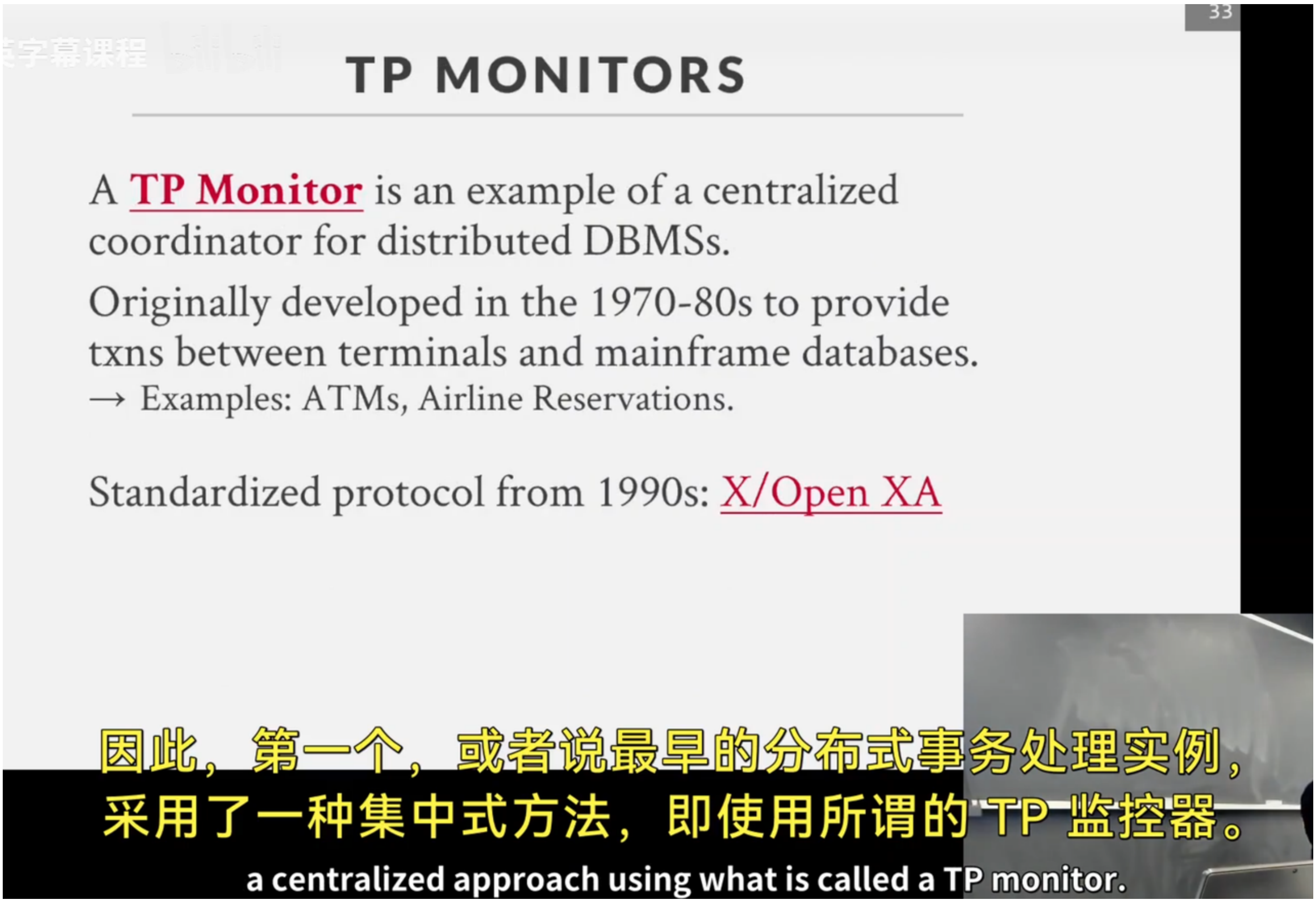
Most distributed DBMSs use a hybrid approach where they periodically elect some node to be a temporary coordinator.

但如果此时该节点发生故障，那么就需要进行新一轮的领导选举，以便其他人能够接替其职责。

can take over.

分布式事务处理方案

集中式



TP MONITORS

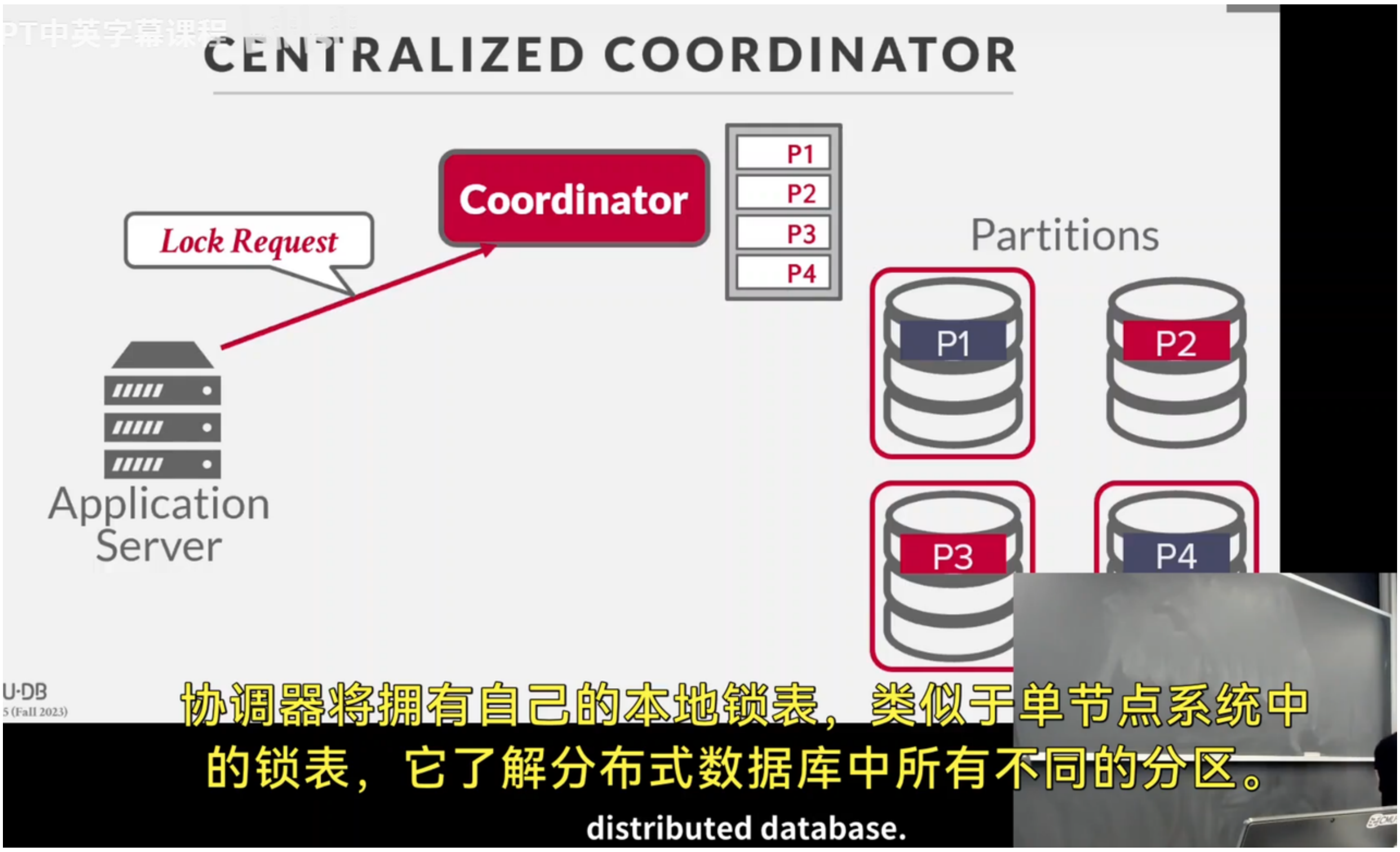
A **TP Monitor** is an example of a centralized coordinator for distributed DBMSs.

Originally developed in the 1970-80s to provide txns between terminals and mainframe databases.
→ Examples: ATMs, Airline Reservations.

Standardized protocol from 1990s: X/Open XA

因此，第一个，或者说最早的分布式事务处理实例，采用了一种集中式方法，即使用所谓的 TP 监控器。

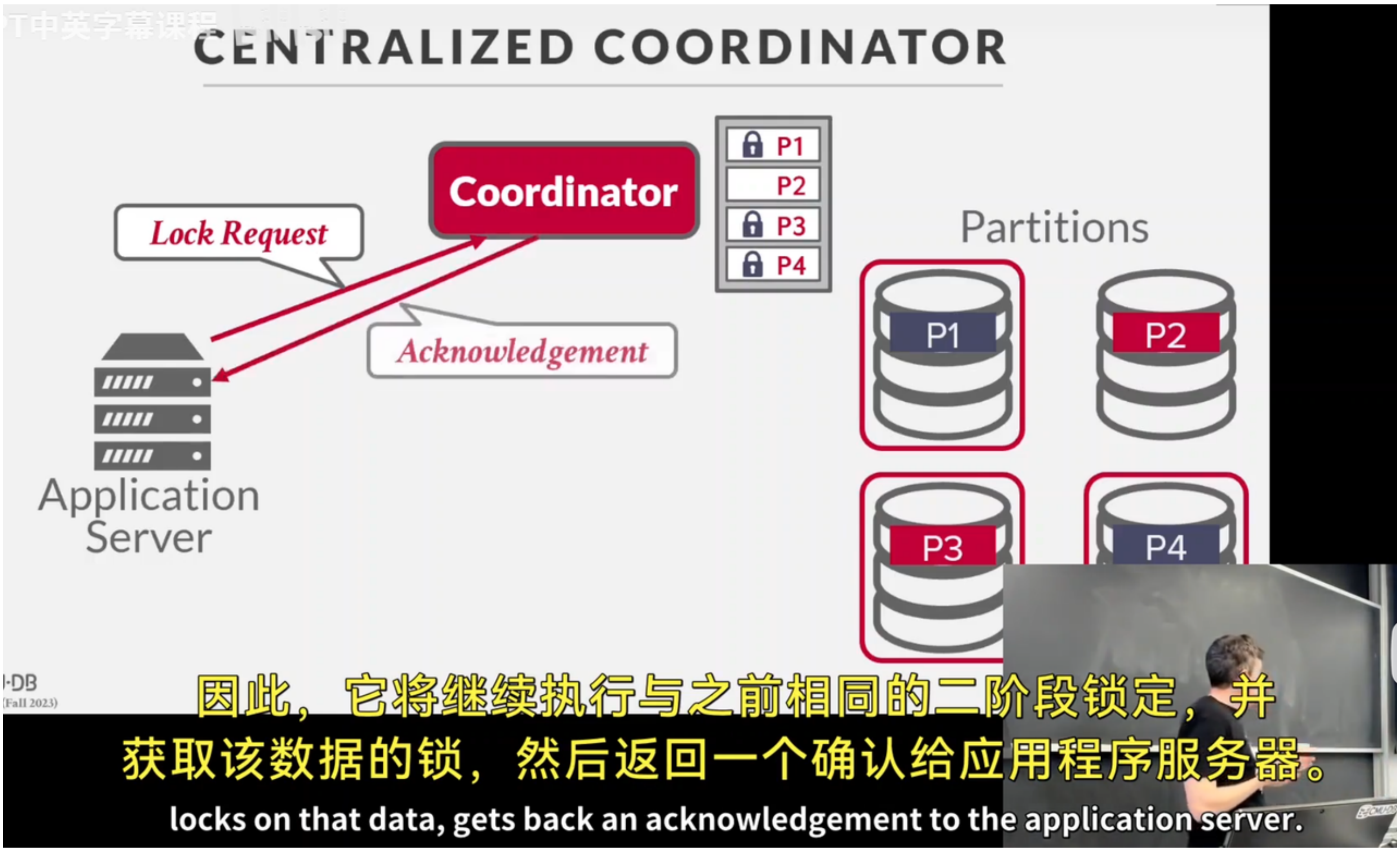
a centralized approach using what is called a TP monitor.



CENTRALIZED COORDINATOR

协调器将拥有自己的本地锁表，类似于单节点系统中的锁表，它了解分布式数据库中所有不同的分区。

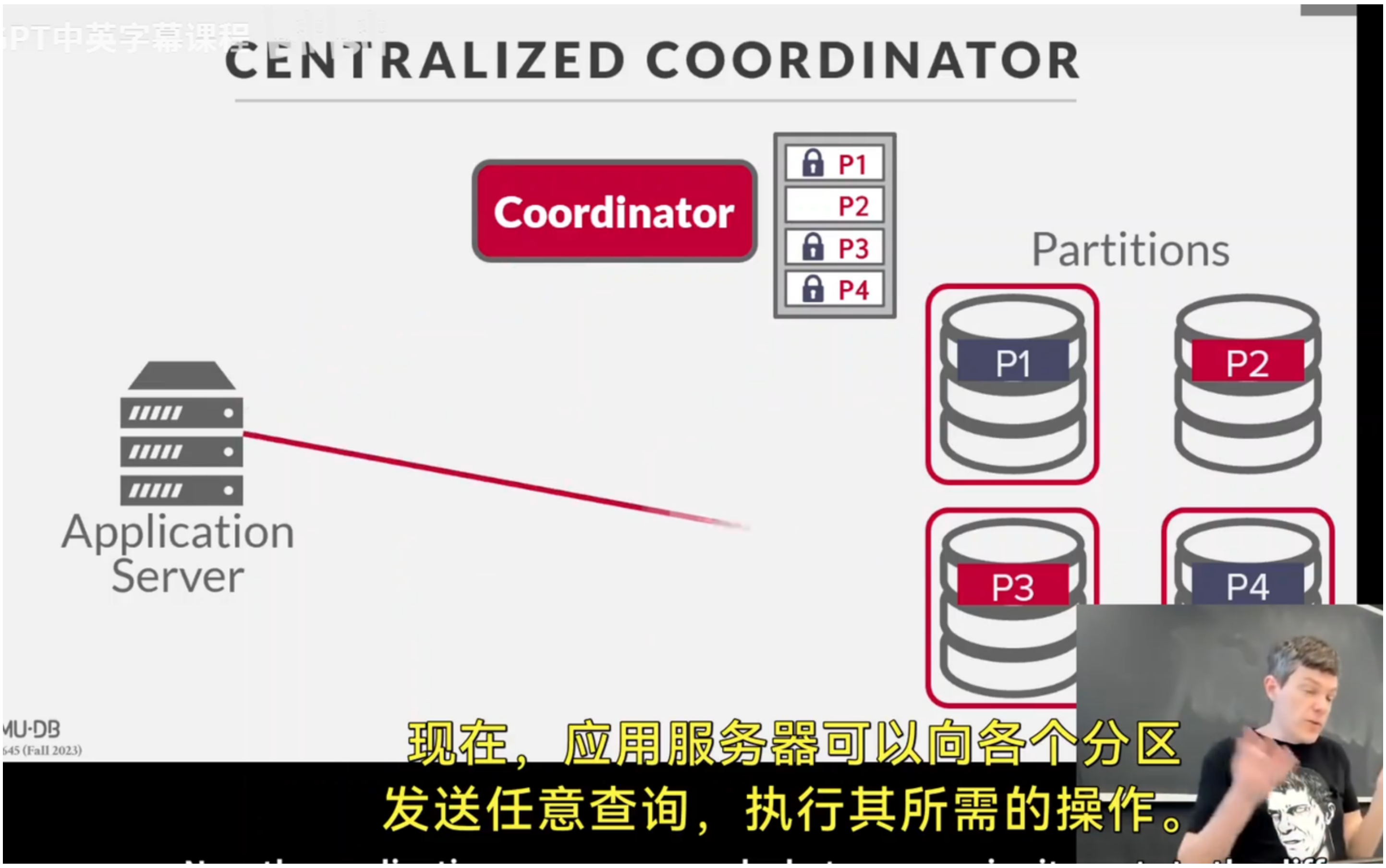
distributed database.



CENTRALIZED COORDINATOR

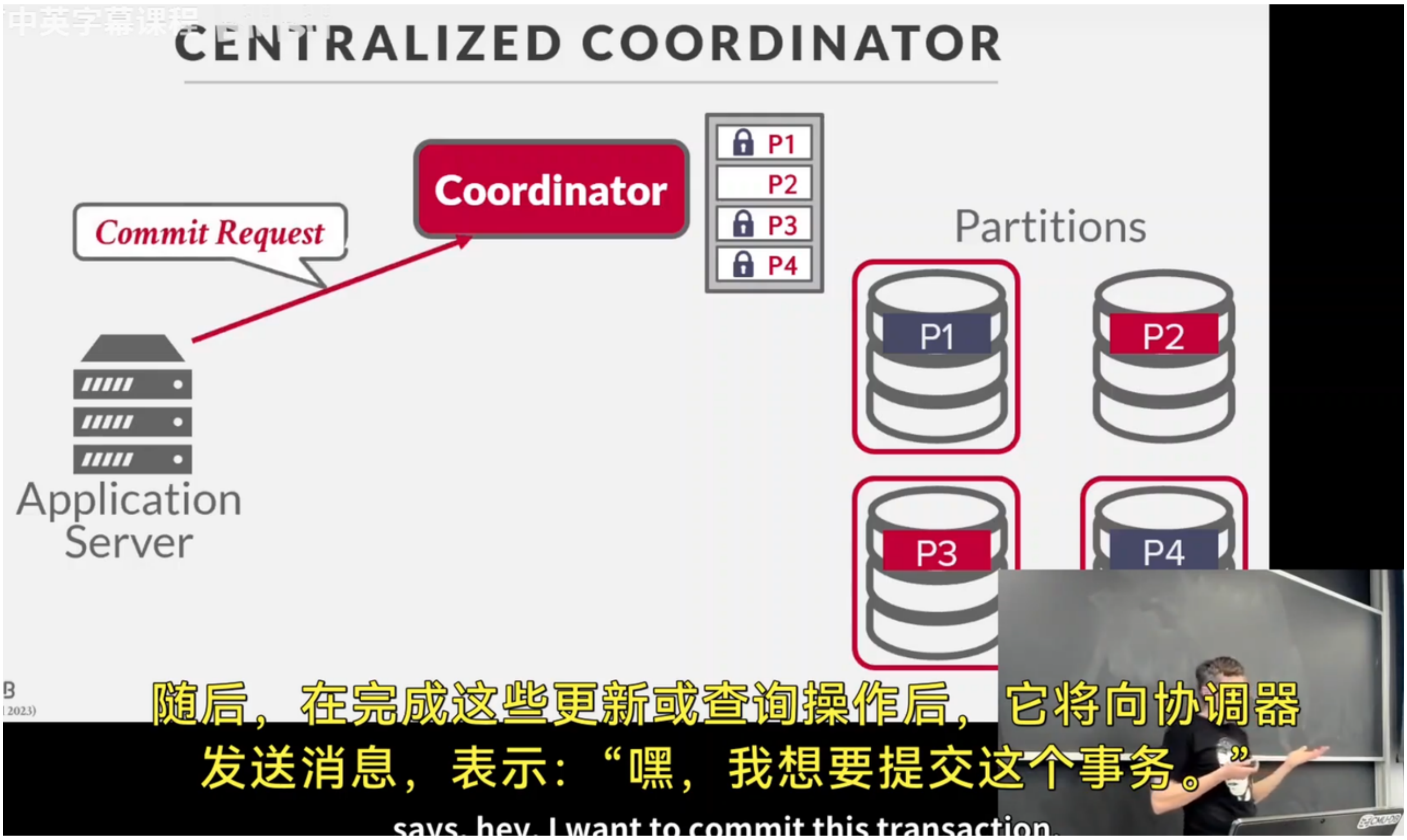
因此，它将继续执行与之前相同的二阶段锁定，并获取该数据的锁，然后返回一个确认给应用程序服务器。

locks on that data, gets back an acknowledgement to the application server.



CENTRALIZED COORDINATOR

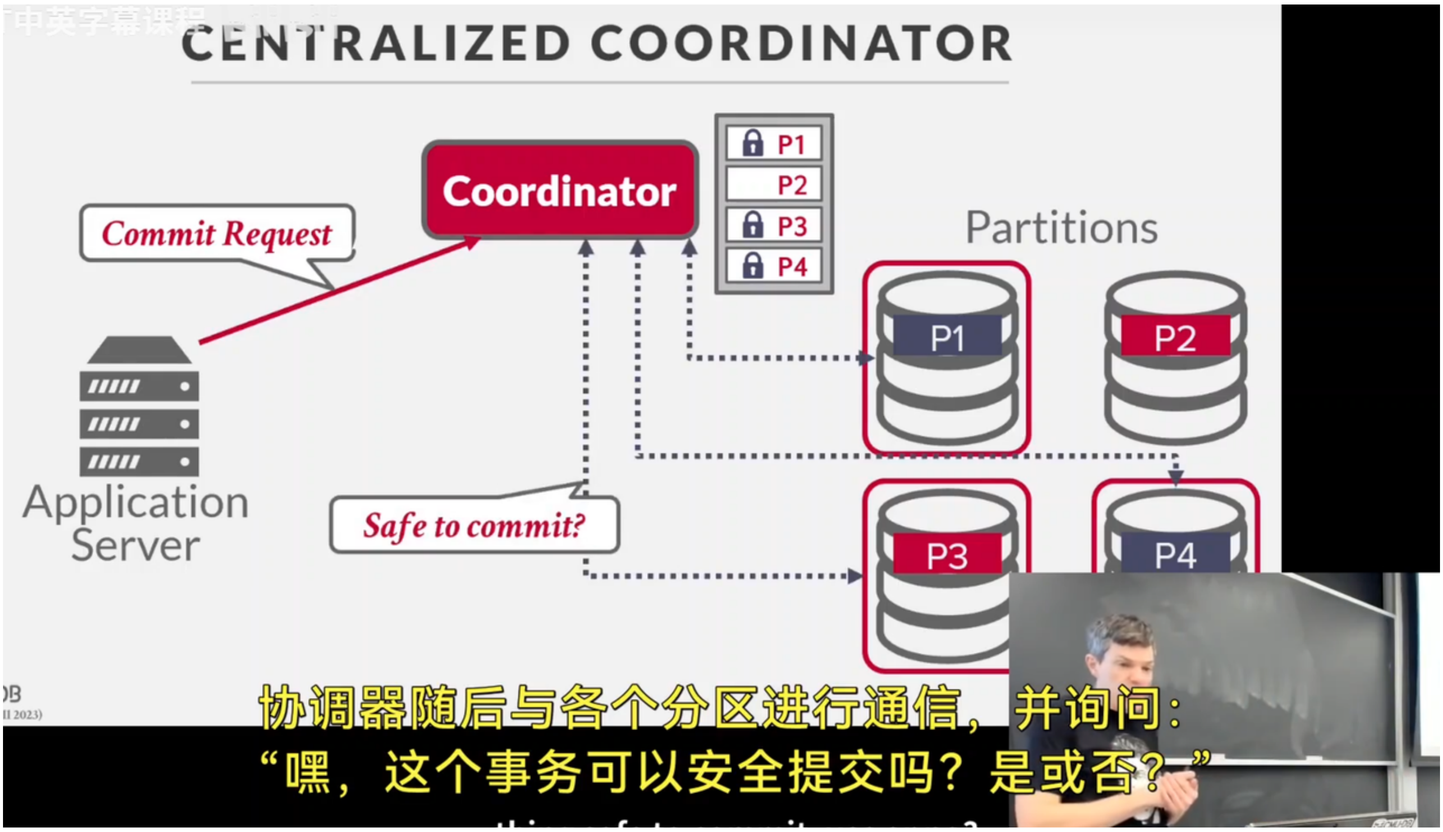
现在，应用服务器可以向各个分区
发送任意查询，执行其所需的操作。



CENTRALIZED COORDINATOR

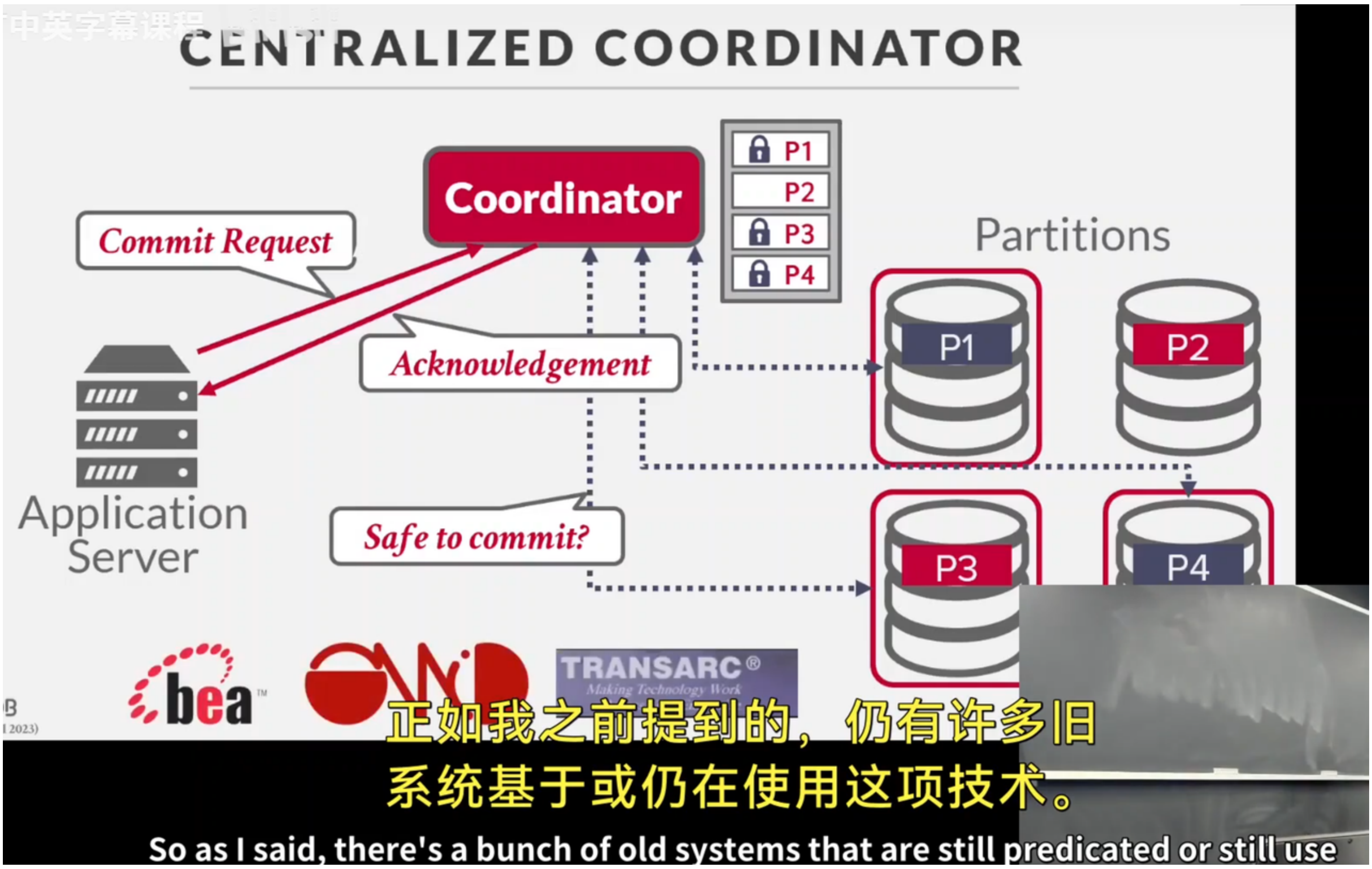
随后，在完成这些更新或查询操作后，它将向协调器
发送消息，表示："嘿，我想要提交这个事务。"

says, hey, I want to commit this transaction.



CENTRALIZED COORDINATOR

协调器随后与各个分区进行通信，并询问：
“嘿，这个事务可以安全提交吗？是或否？”

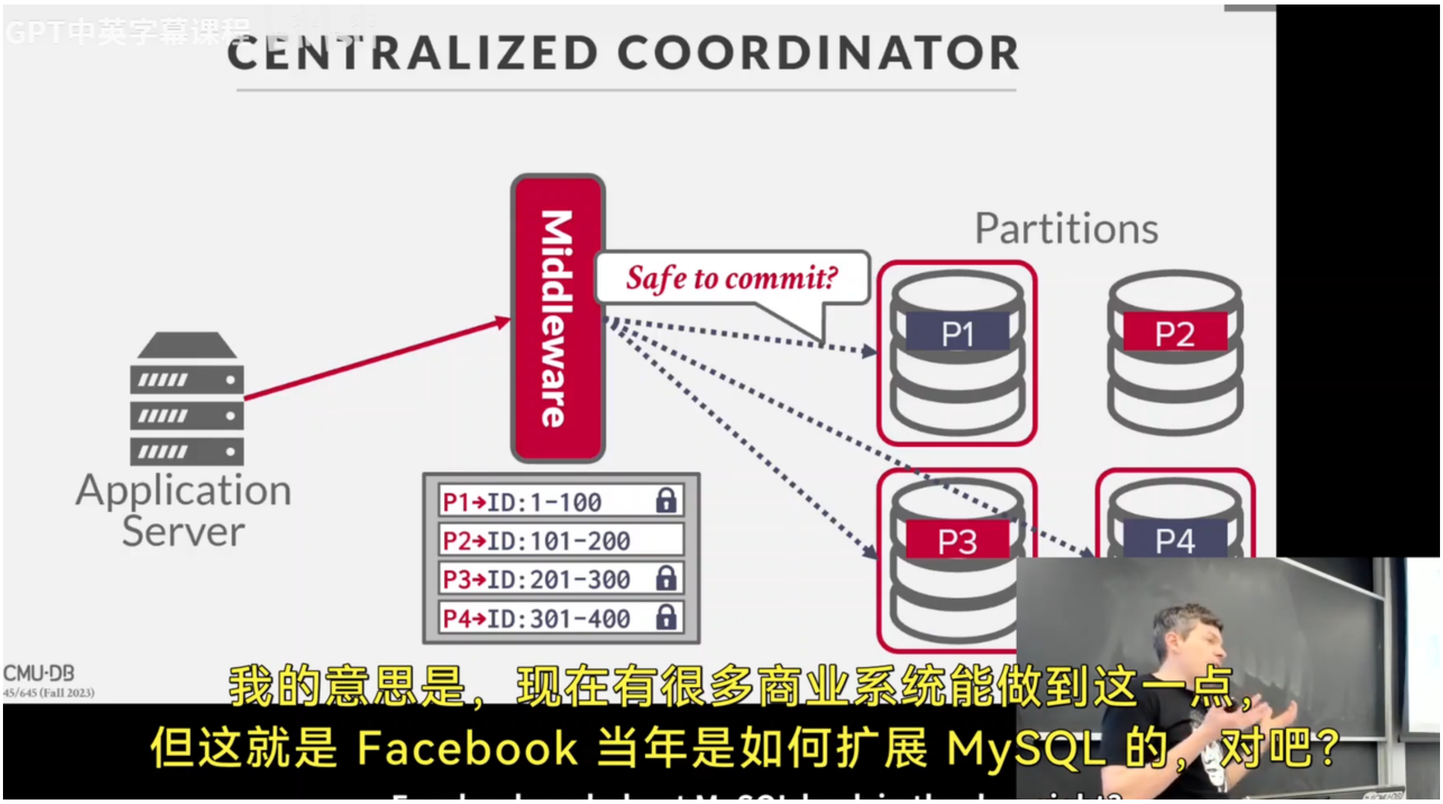


CENTRALIZED COORDINATOR

正如我之前提到的，仍有许多旧
系统基于或仍在使用这项技术。

So as I said, there's a bunch of old systems that are still predicated or still use

> 更常见的是下图使用一些中间件方法



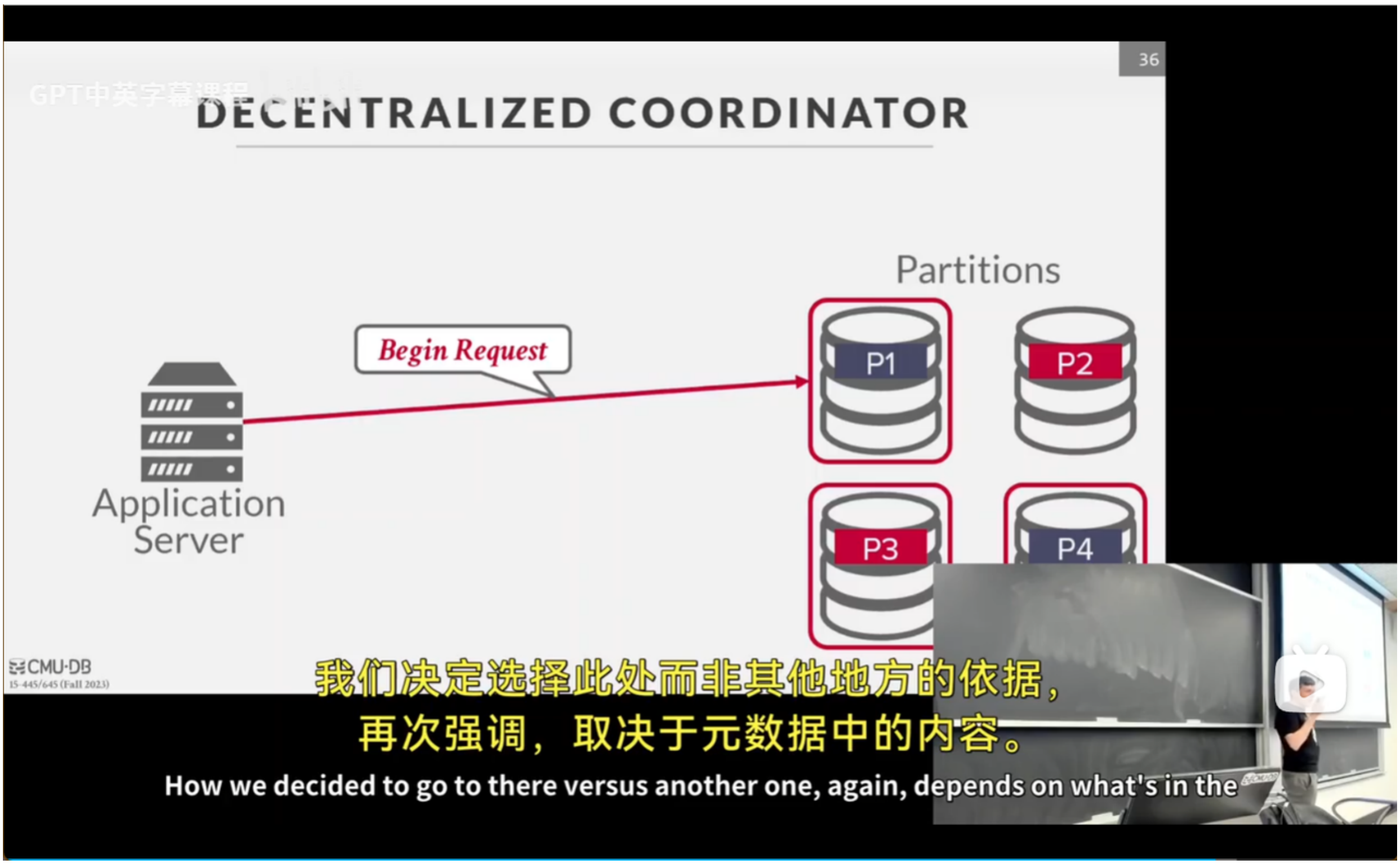
CENTRALIZED COORDINATOR

P1→ID:1-100 🔒
P2→ID:101-200
P3→ID:201-300 🔒
P4→ID:301-400 🔒

我的意思是，现在有很多商业系统能做到这一点，

但这就是 Facebook 当年是如何扩展 MySQL 的，对吧？

去中心化

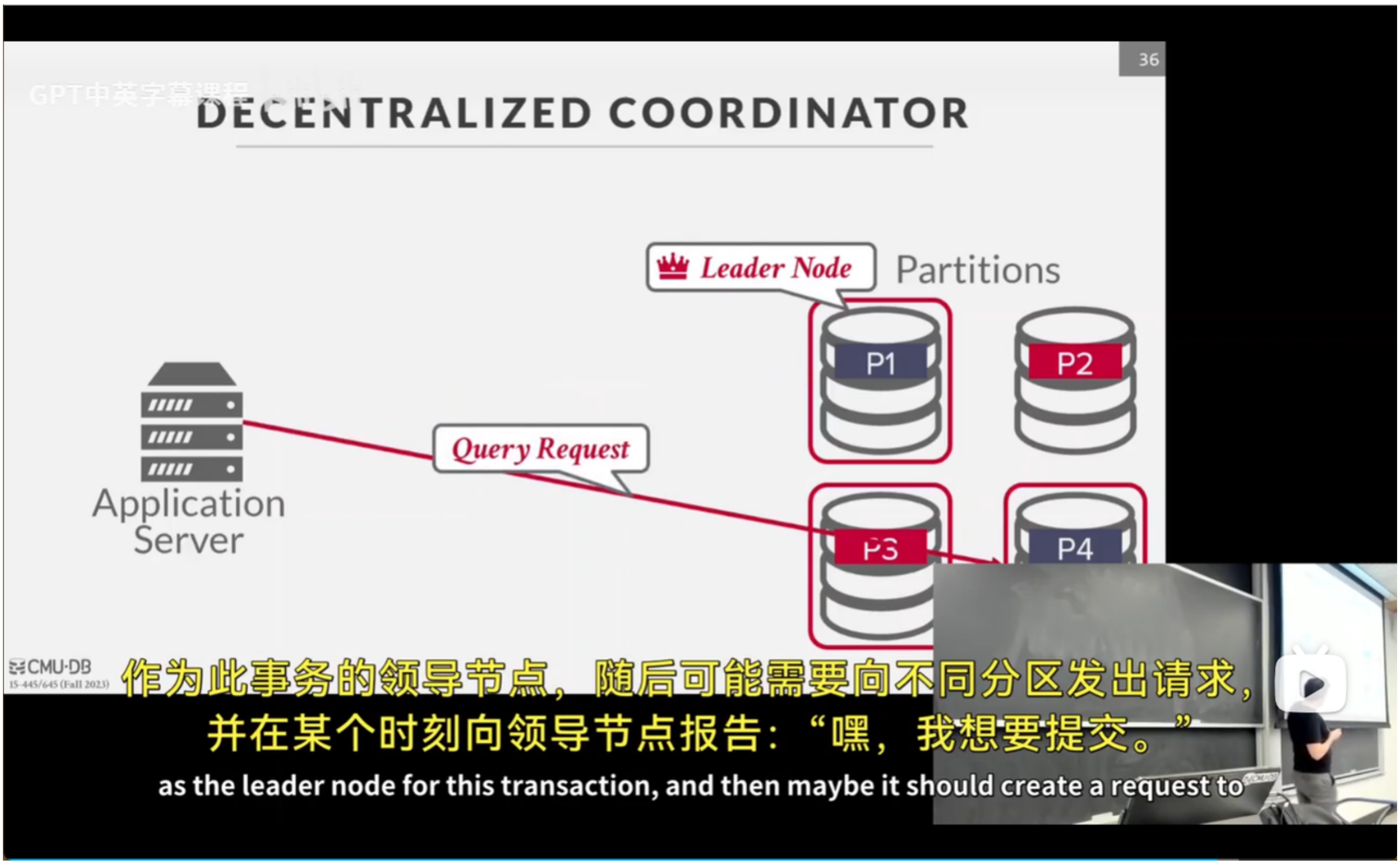


DECENTRALIZED COORDINATOR

我们决定选择此处而非其他地方的依据，

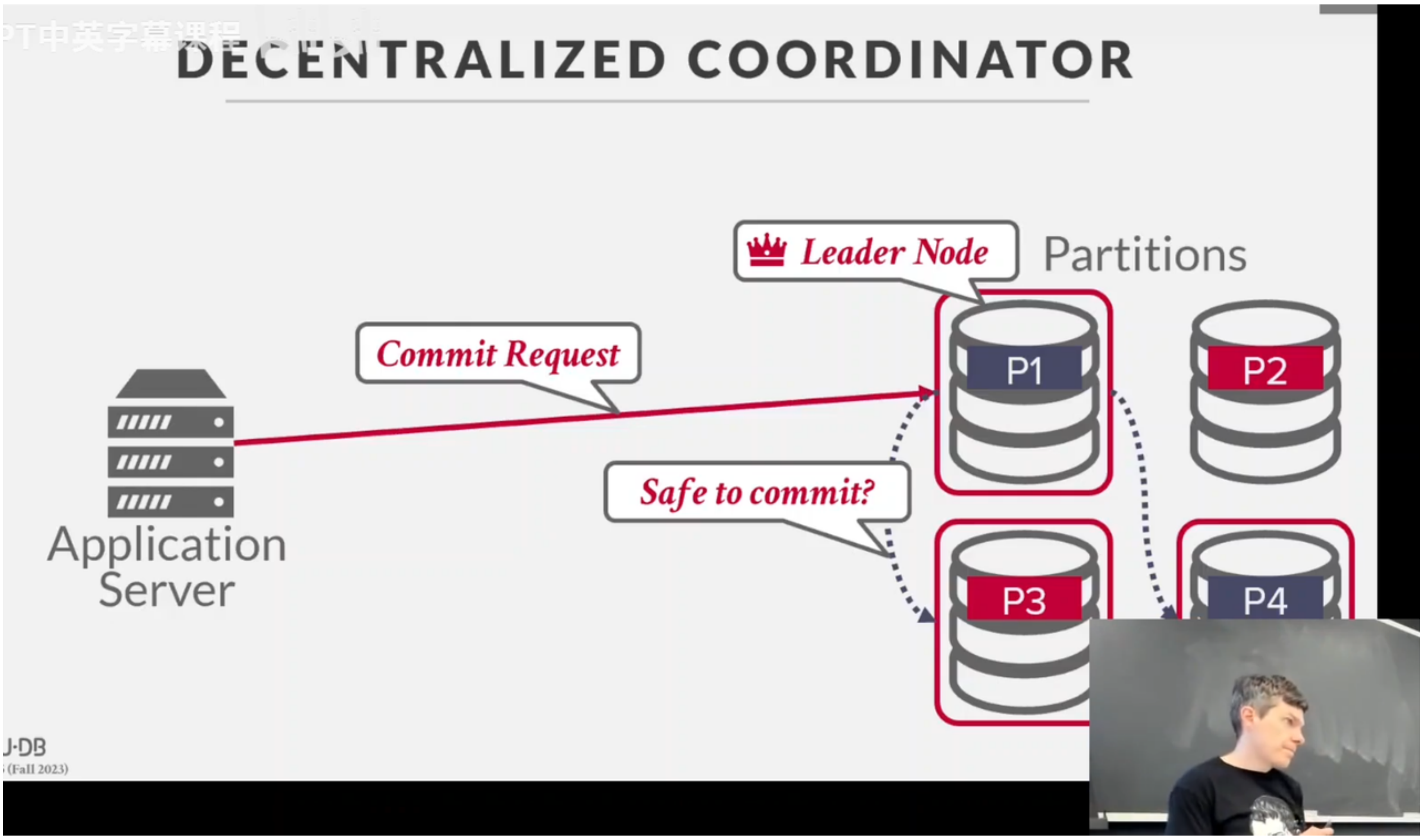
再次强调，取决于元数据中的内容。

How we decided to go to there versus another one, again, depends on what's in the

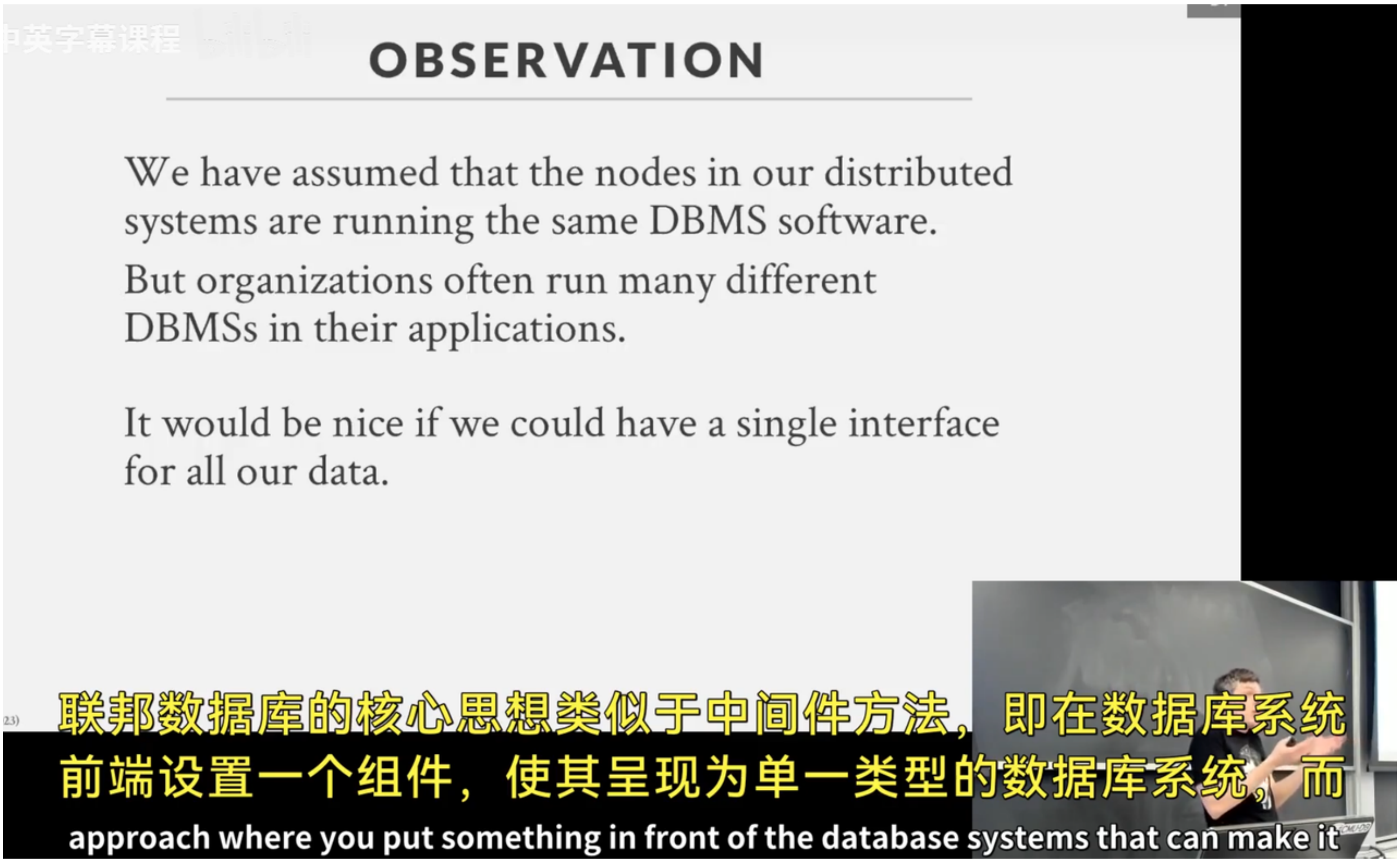


DECENTRALIZED COORDINATOR

作为此事务的领导节点，随后可能需要向不同分区发出请求，
并在某个时刻向领导节点报告：“嘿，我想要提交。”
as the leader node for this transaction, and then maybe it should create a request to



DECENTRALIZED COORDINATOR



OBSERVATION

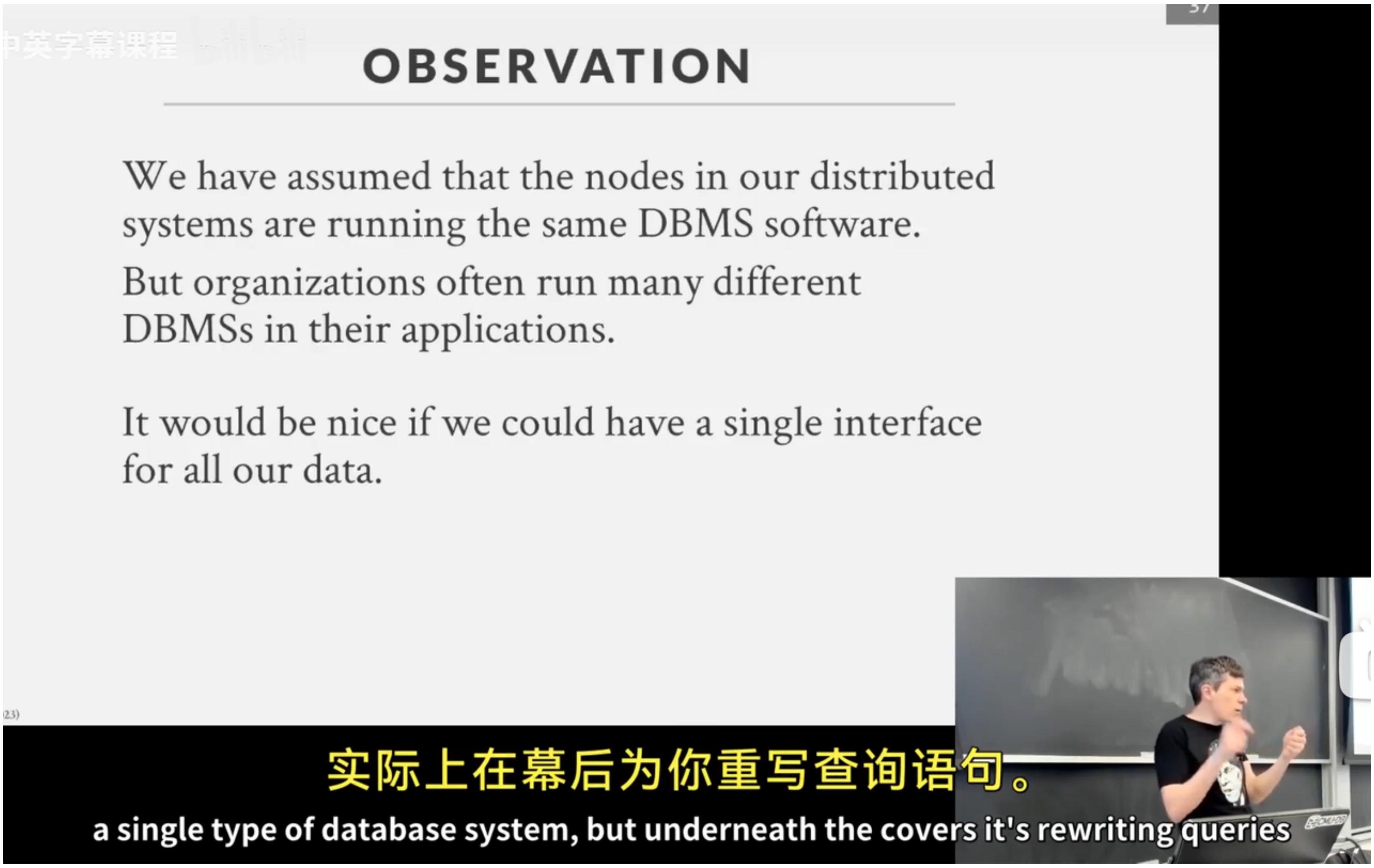
We have assumed that the nodes in our distributed systems are running the same DBMS software.

But organizations often run many different DBMSs in their applications.

It would be nice if we could have a single interface for all our data.

联邦数据库的核心思想类似于中间件方法，即在数据库系统前端设置一个组件，使其呈现为单一类型的数据库系统，而

approach where you put something in front of the database systems that can make it



OBSERVATION

We have assumed that the nodes in our distributed systems are running the same DBMS software.

But organizations often run many different DBMSs in their applications.

It would be nice if we could have a single interface for all our data.

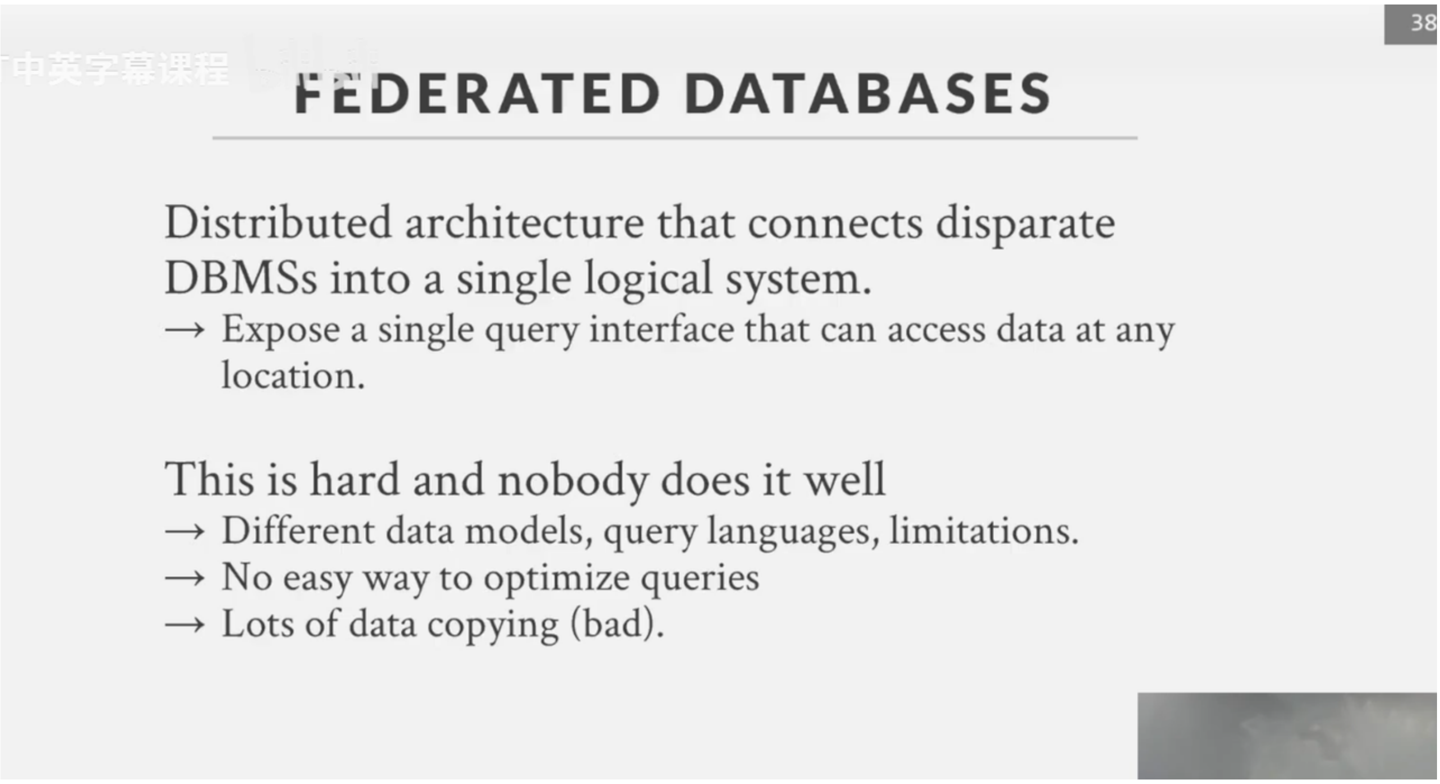
实际上在幕后为你重写查询语句。

a single type of database system, but underneath the covers it's rewriting queries

联邦数据库

> 使用了不同的数据库，有不同的数据模型、查询语言和限制；
>
> 必须将大量的数据拉向中间件

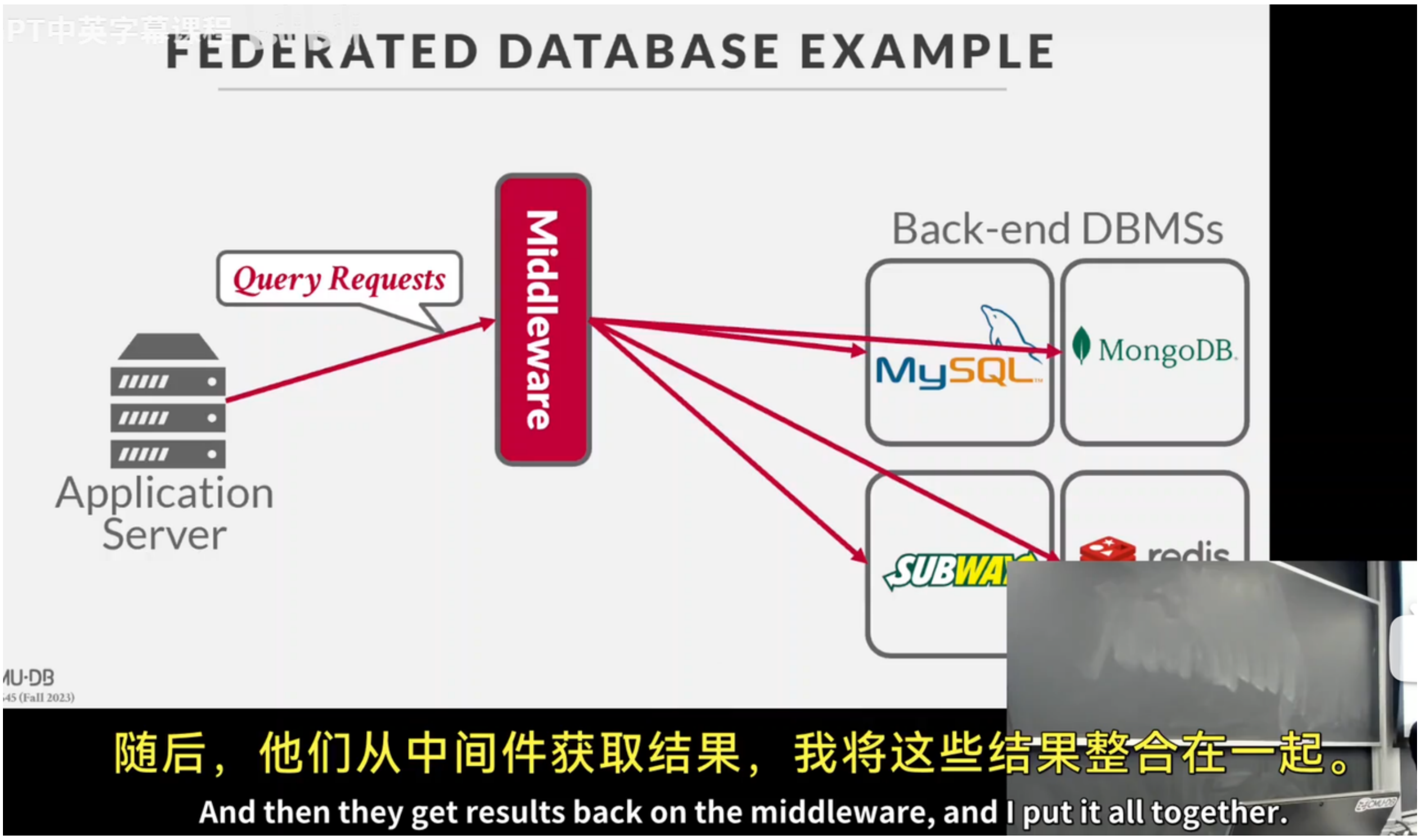
无法容易的优化查询



FEDERATED DATABASES

Distributed architecture that connects disparate DBMSs into a single logical system.
→ Expose a single query interface that can access data at any location.

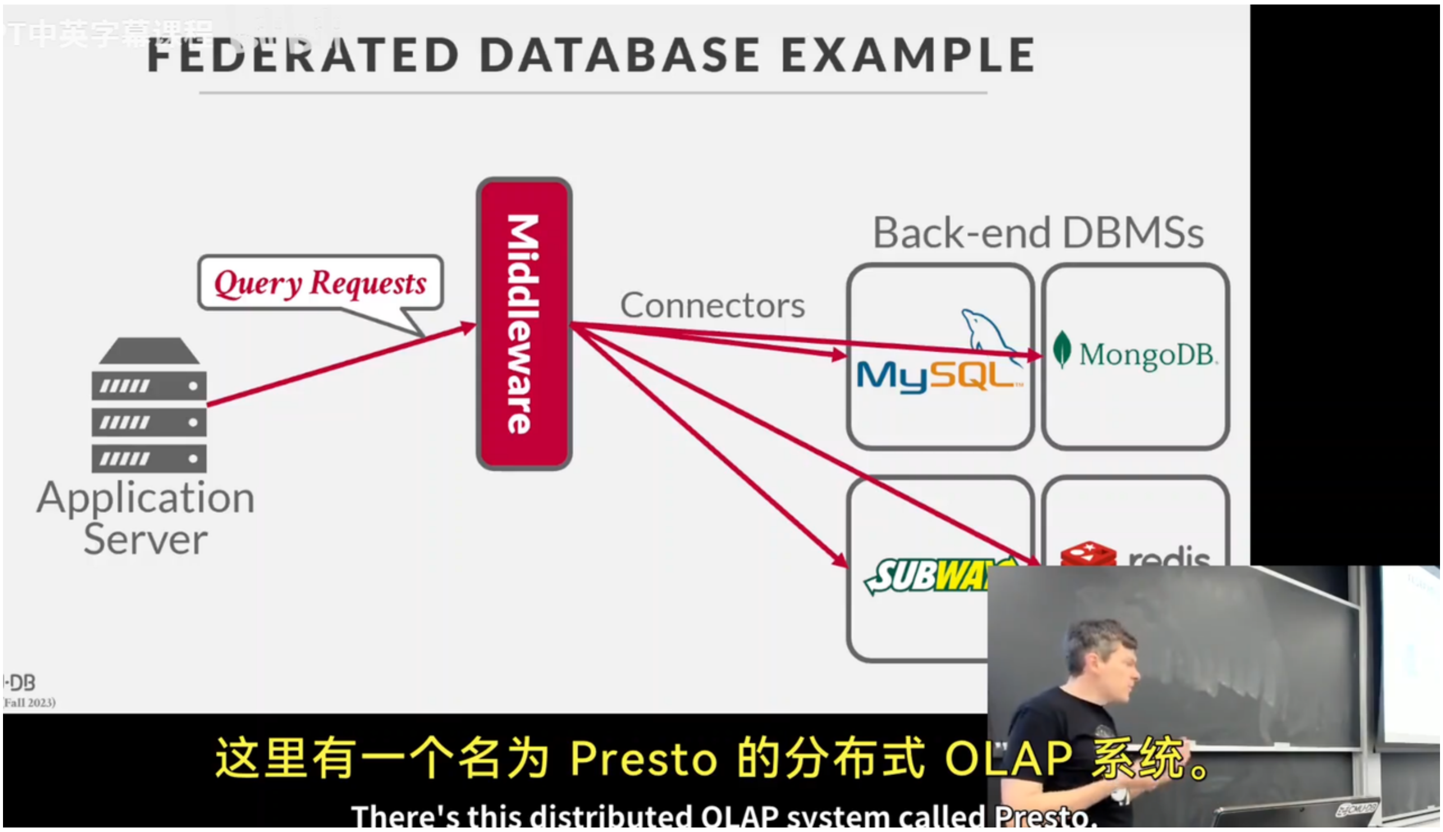
This is hard and nobody does it well
→ Different data models, query languages, limitations.
→ No easy way to optimize queries
→ Lots of data copying (bad).



FEDERATED DATABASE EXAMPLE

随后，他们从中间件获取结果，我将这些结果整合在一起。

And then they get results back on the middleware, and I put it all together.



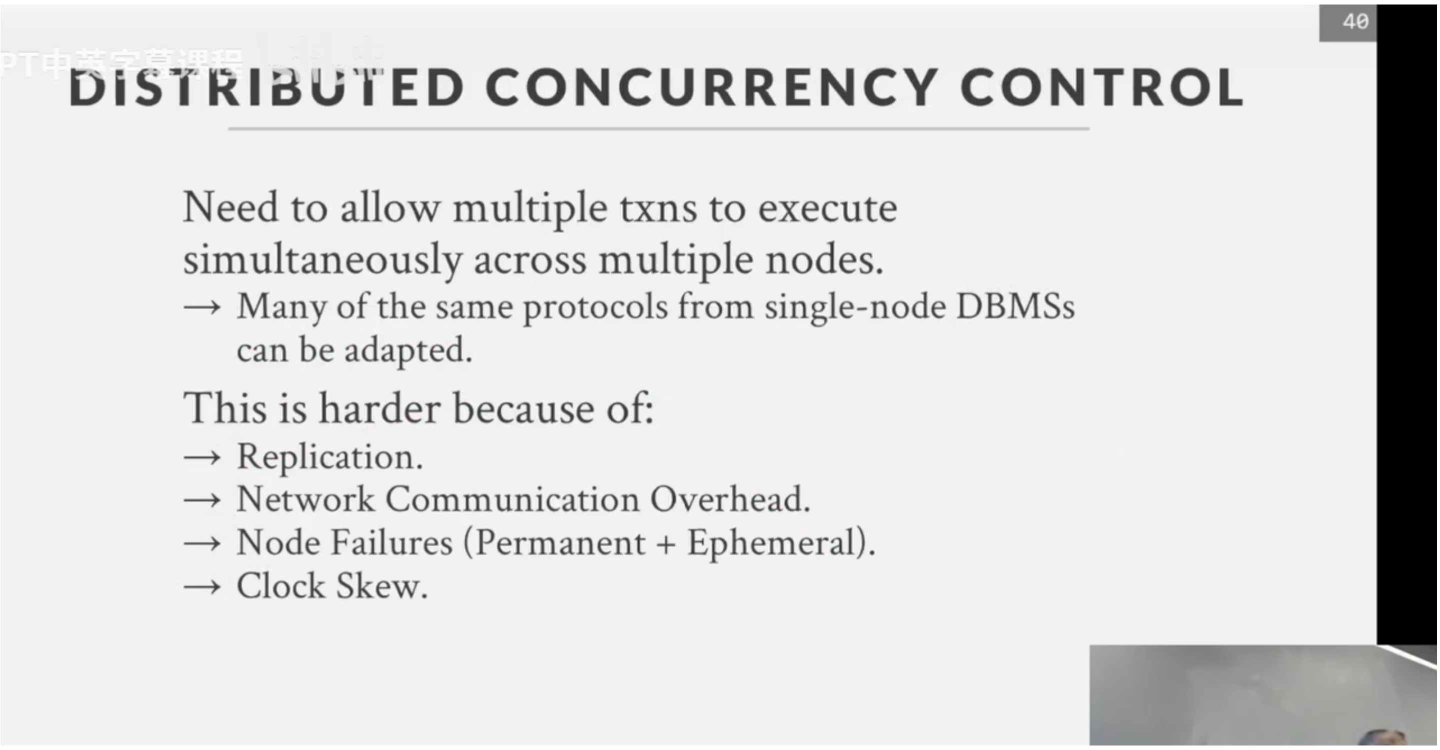
FEDERATED DATABASE EXAMPLE

这里有一个名为 Presto 的分布式 OLAP 系统。

There's this distributed OLAP system called Presto.

分布式并发控制

> 数据复制同步问题和时钟偏斜问题

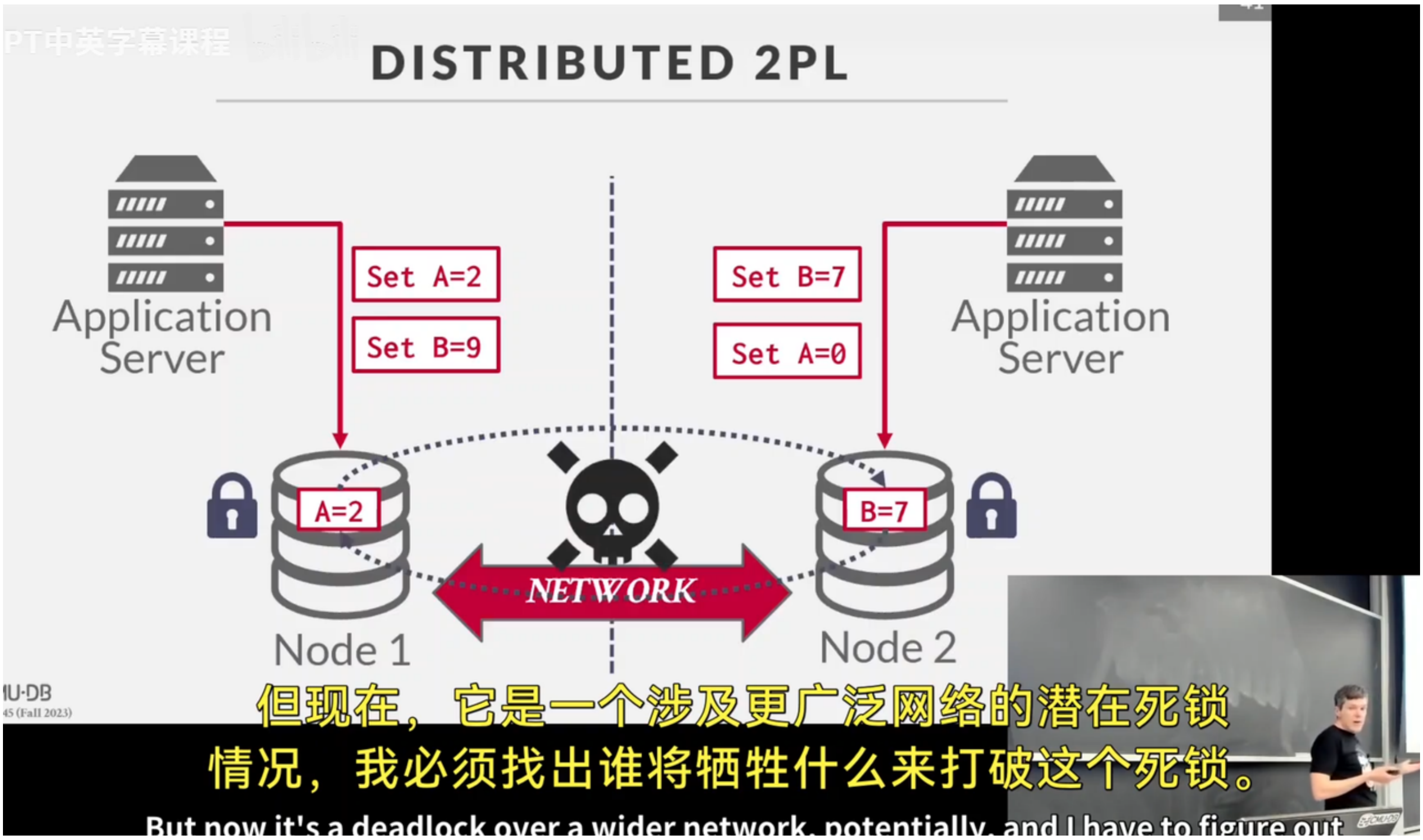


DISTRIBUTED CONCURRENCY CONTROL

Need to allow multiple txns to execute simultaneously across multiple nodes.
→ Many of the same protocols from single-node DBMSs can be adapted.

This is harder because of:
→ Replication.
→ Network Communication Overhead.
→ Node Failures (Permanent + Ephemeral).
→ Clock Skew.

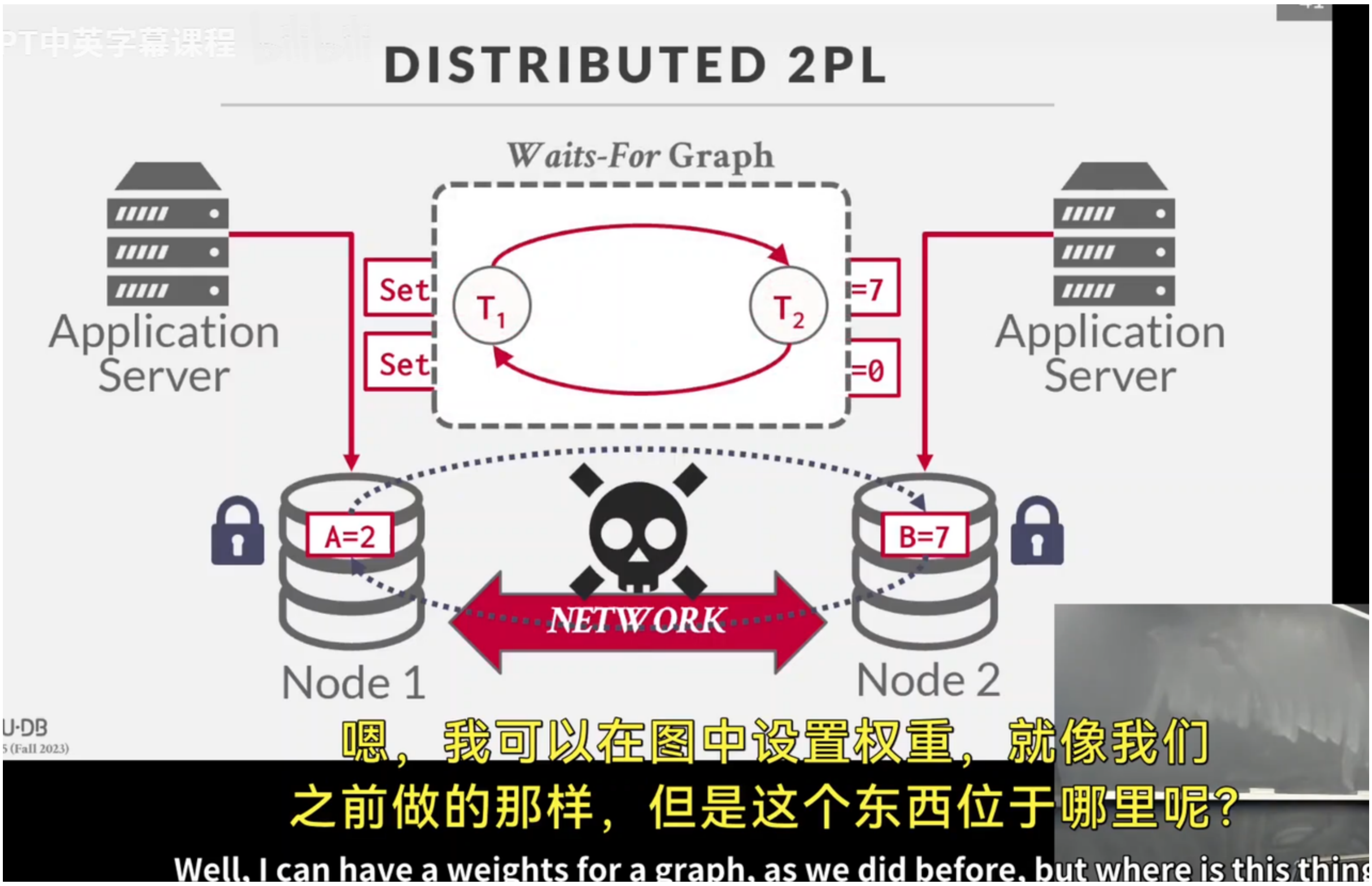
分布式2PL



DISTRIBUTED 2PL

但现在，它是一个涉及更广泛网络的潜在死锁情况，我必须找出谁将牺牲什么来打破这个死锁。

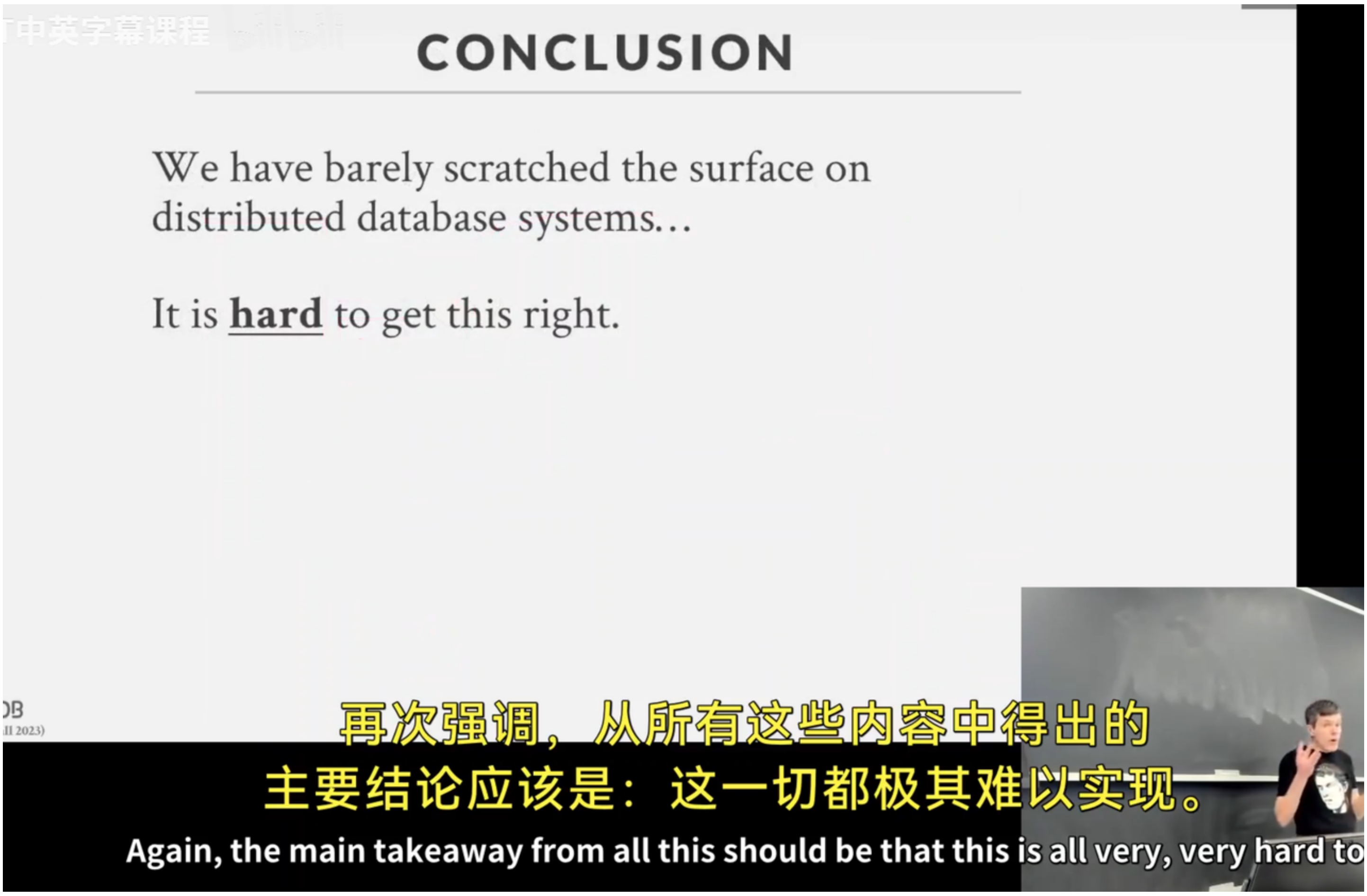
But now it's a deadlock over a wider network, potentially, and I have to figure out



DISTRIBUTED 2PL

嗯，我可以在图中设置权重，就像我们之前做的那样，但是这个东西位于哪里呢？

Well, I can have a weights for a graph, as we did before, but where is this thing

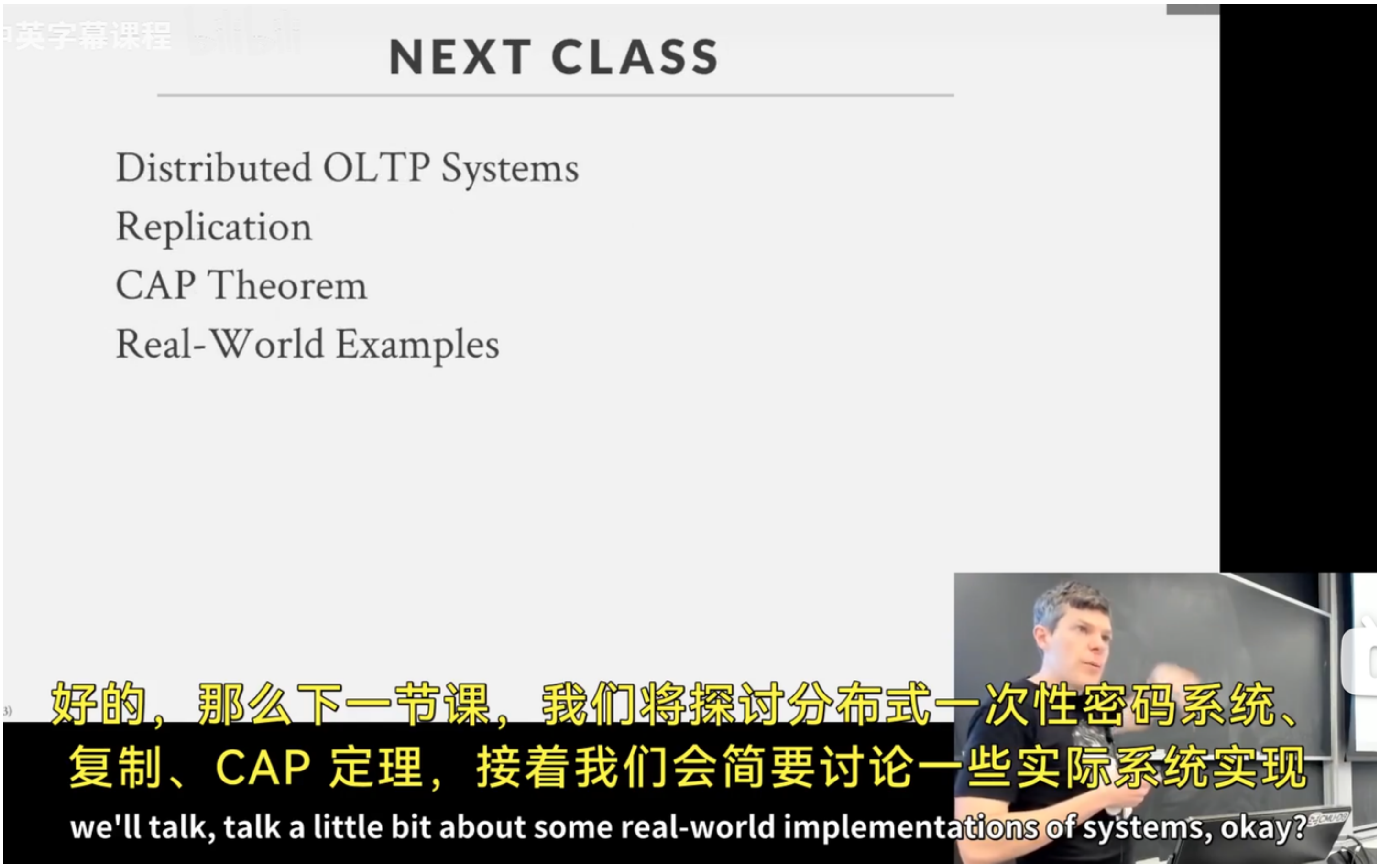


CONCLUSION

We have barely scratched the surface on distributed database systems…

It is **hard** to get this right.

再次强调，从所有这些内容中得出的
主要结论应该是：这一切都极其难以实现。
Again, the main takeaway from all this should be that this is all very, very hard to



NEXT CLASS

Distributed OLTP Systems
Replication
CAP Theorem
Real-World Examples

好的，那么下一节课，我们将探讨分布式一次性密码系统、
复制、CAP 定理，接着我们会简要讨论一些实际系统实现
we'll talk, talk a little bit about some real-world implementations of systems, okay?