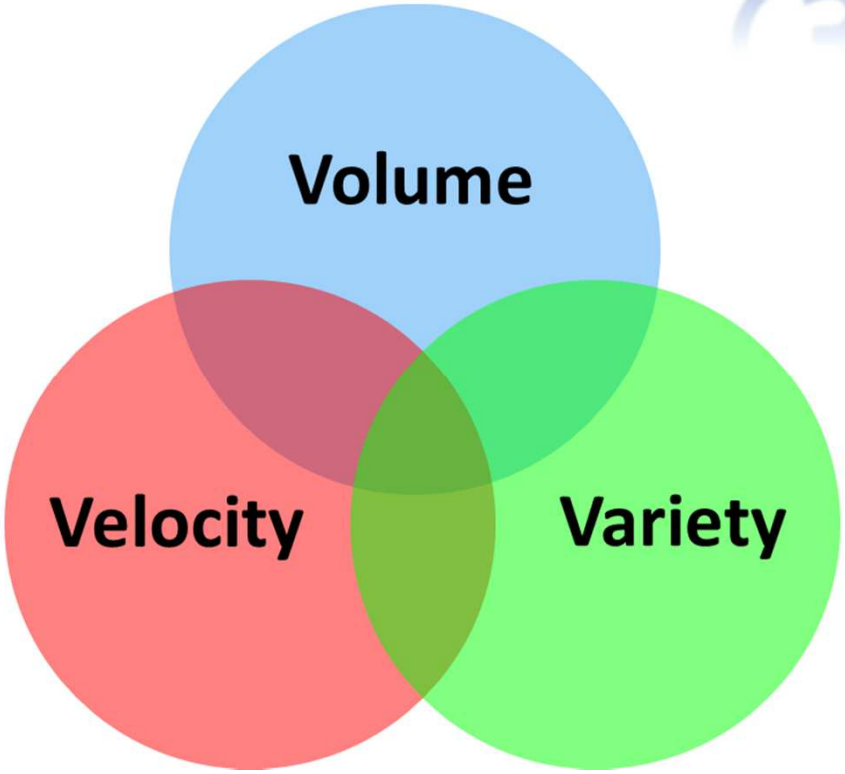
大数据管理系统与大规模数据分析

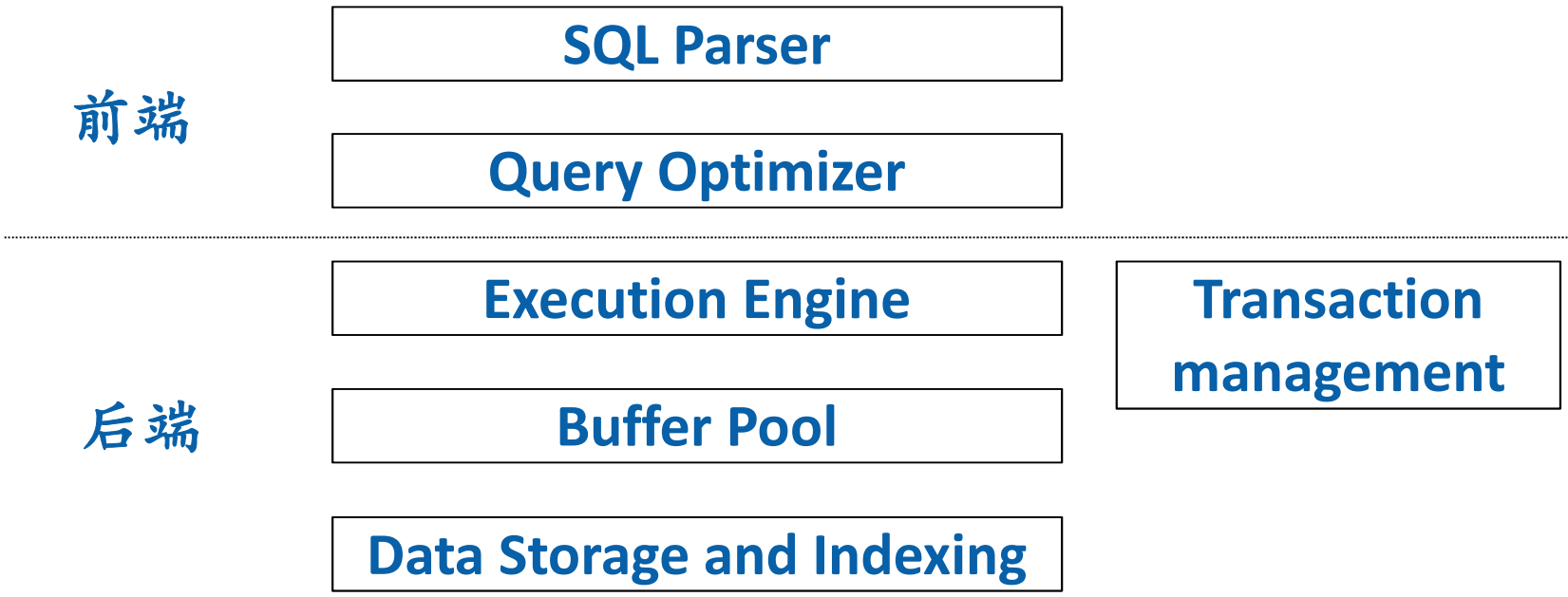
关系型数据管理系统（3）



陈世敏

中科院计算所
计算机体系结构
国家重点实验室

RDBMS的系统架构(单机)



Outline

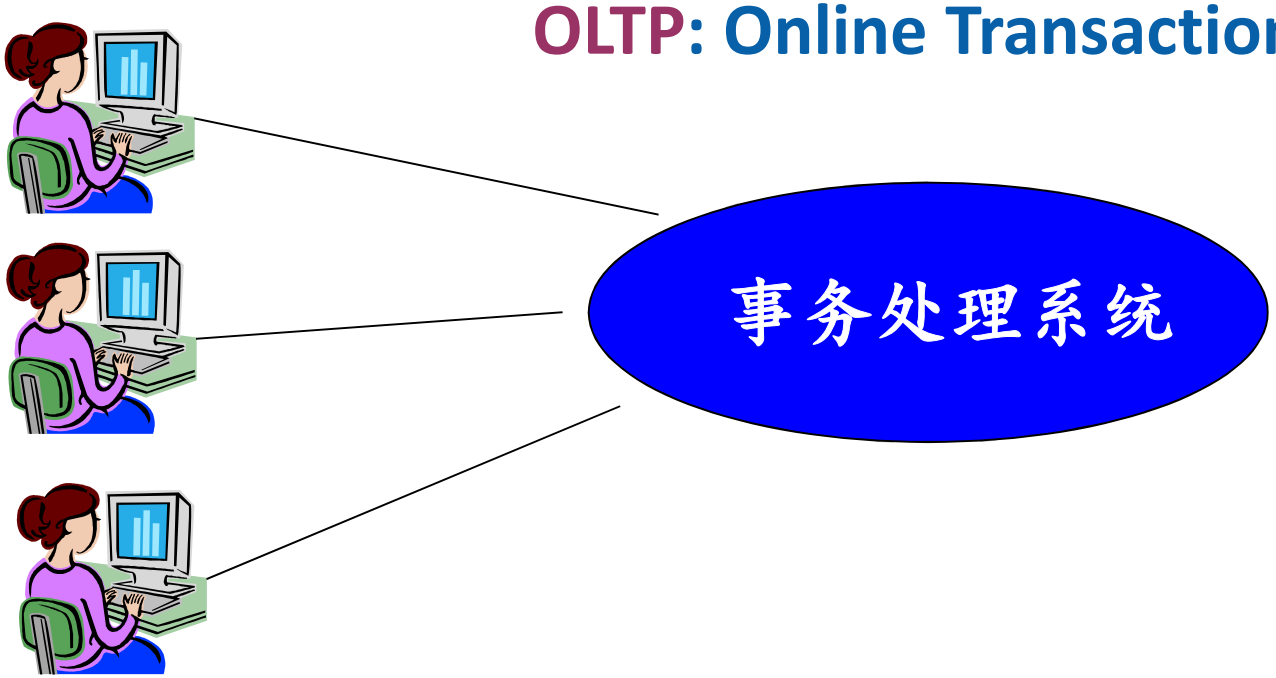
- 事务处理
- 数据仓库
- 分布式数据库

Outline

- 事务处理
  - ACID
  - Concurrency Control (并发控制)
  - Crash Recovery (崩溃恢复)
- 数据仓库
- 分布式数据库

事务处理 (Transaction Processing)

OLTP: Online Transaction Processing



- 典型例子：银行业务，订票，购物等
- 大量并发用户，少量随机读写操作

什么叫事务？（Transaction）

- 一个事务可能包含多个操作
  - select
  - insert/delete/update
  - 等
- 事务中的所有操作满足ACID性质

事务的表现形式

- 没有特殊设置
  - 那么每个SQL语句被认为是一个事务
- 使用特殊的语句
  - 开始transaction
  - 成功结束transaction
  - 异常结束transaction

Transaction

成功的事务

```
begin transaction;
……
commit transaction;
```

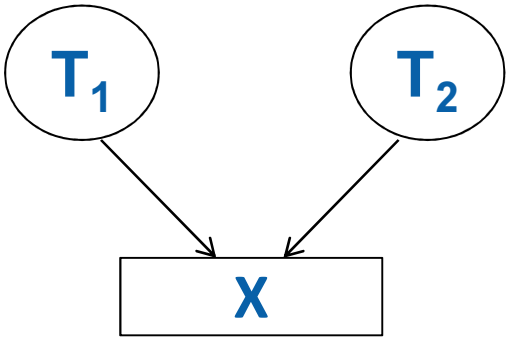
可以用rollback回卷事务

```
begin transaction;
……
rollback transaction;
```

ACID：DBMS保证事务的ACID性质

- Atomicity（原子性）
  - all or nothing
  - 要么完全执行，要么完全没有执行
- Consistency（一致性）
  - 从一个正确状态转换到另一个正确状态
    （正确指：constraints, triggers等）
- Isolation（隔离性）
  - 每个事务与其它并发事务互不影响
- Durability（持久性）
  - Transaction commit后，结果持久有效，crash也不消失

同一个数据元素被并发访问

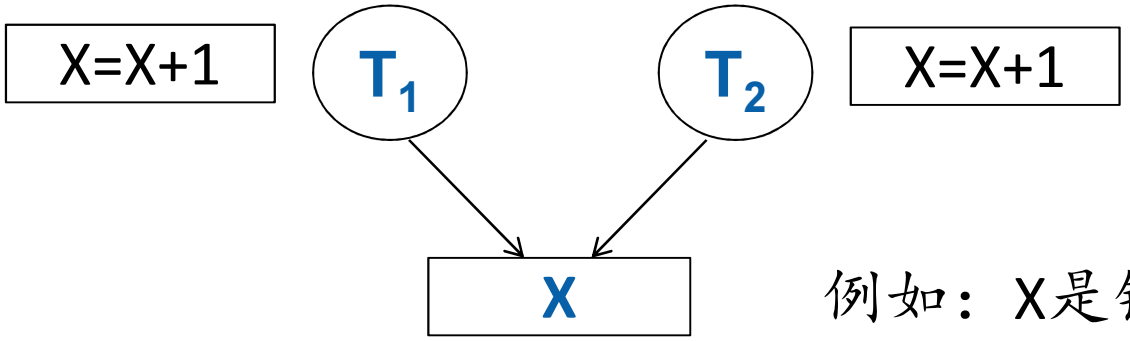


例如：X是银行账户余额

- 会有什么问题？

数据竞争(Data Race)

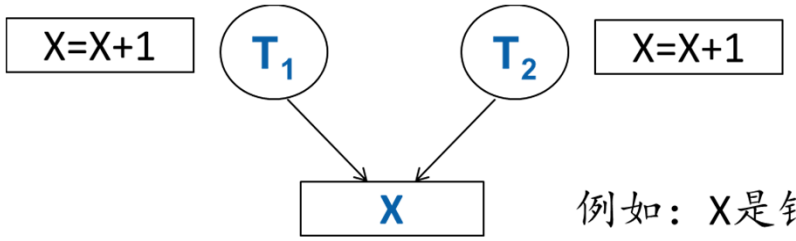
- 当两个并发访问都是写，或者一个读一个写时



例如：X是银行账户余额

- 场景：同一个账户两笔转帐并发发生会怎样？
  - 初始：X=100
  - 最终X=?

Schedule
(调度/执行顺序)



例如：X是银行账户余额

初始： X=100

T1	T2
Read(X)	
	Read(X)
	Write(X)
Write(X)	

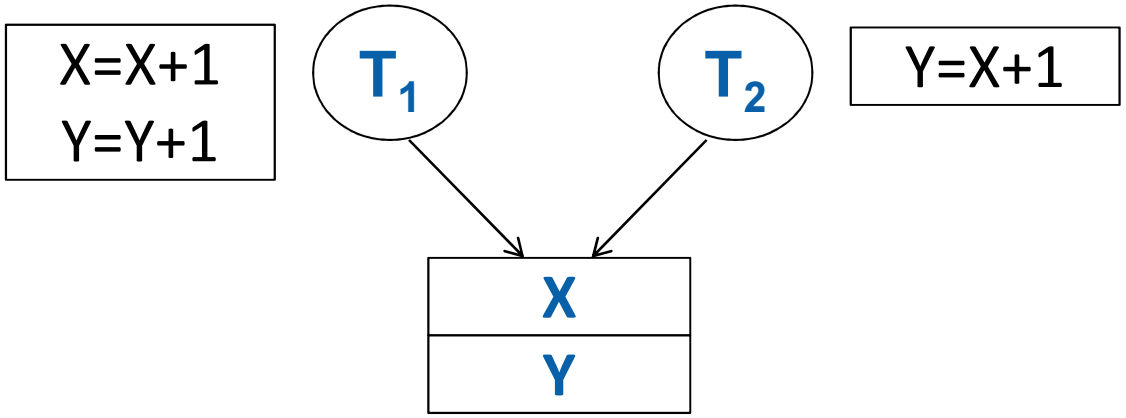
X=101

T2的write被覆盖了

T1	T2
Read(X)	
Write(X)	
	Read(X)
	Write(X)

X=102

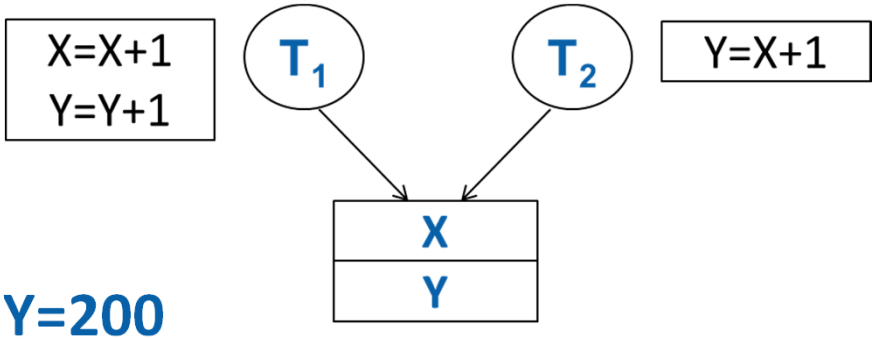
更复杂的情况



- 两个Transactions并发访问多个共享的数据元素
- 实际上，真实情况更加复杂

Schedule

(调度/执行顺序)



初始： X=100, Y=200

T1	T2
Read(X)	
	X=100
	Read(X)
	Write(Y)
	Y=101
Write(X)	
Read(Y)	
Write(Y)	

Y=102

T1	T2
Read(X)	
Write(X)	
	X=101
	Read(X)
	Write(Y)
	Y=102
Read(Y)	
Write(Y)	

Y=103

T1	T2
Read(X)	
	X=100
	Read(X)
Write(X)	
Read(Y)	
Write(Y)	
	Write(Y)

Y=101

……

正确性问题

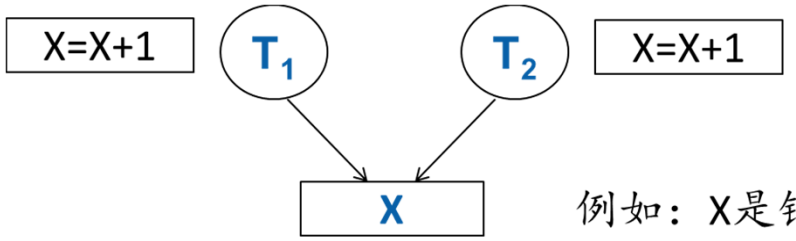
- 提出解决方案前，我们必须提问
- 如何判断一组Transactions正确执行？

- 存在一个顺序，按照这个顺序依次**串行执行**这些Transactions，得到的结果与**并行执行**相同

Serializable(可串行化)

- 判断一组并行Transactions是否正确执行的标准

Serializable?



例如：X是银行账户余额

初始：X=100

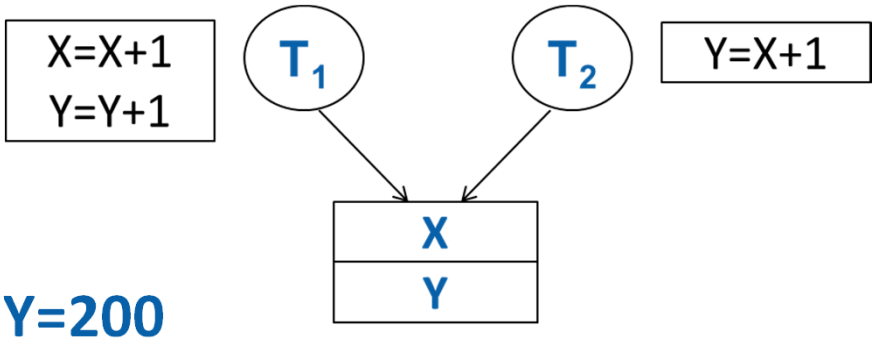
T1	T2
Read(X)	
	Read(X)
	Write(X)
Write(X)	

X=101

T1	T2
Read(X)	
Write(X)	
	Read(X)
	Write(X)

X=102

Serializable?



初始： X=100, Y=200

T1	T2
Read(X)	
	Read(X)
	Write(Y)
Write(X)	
Read(Y)	
Write(Y)	

Y=102

T1	T2
Read(X)	
Write(X)	
	Read(X)
	Write(Y)
Read(Y)	
Write(Y)	

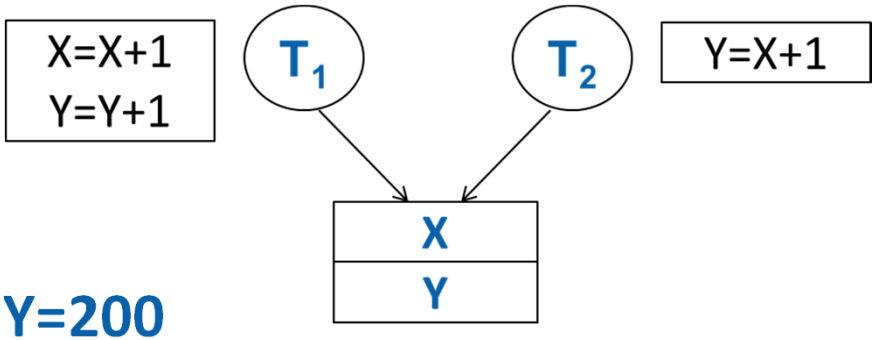
Y=103

T1	T2
Read(X)	
	Read(X)
Write(X)	
Read(Y)	
Write(Y)	
	Write(Y)

Y=101

……

Serializable?



初始： X=100, Y=200

T1	T2
Read(X)	
	Read(X)
	Write(Y)
Write(X)	
Read(Y)	
Write(Y)	

Y=102

=

T1	T2
	Read(X)
	Write(Y)
Read(X)	
Write(X)	
Read(Y)	
Write(Y)	

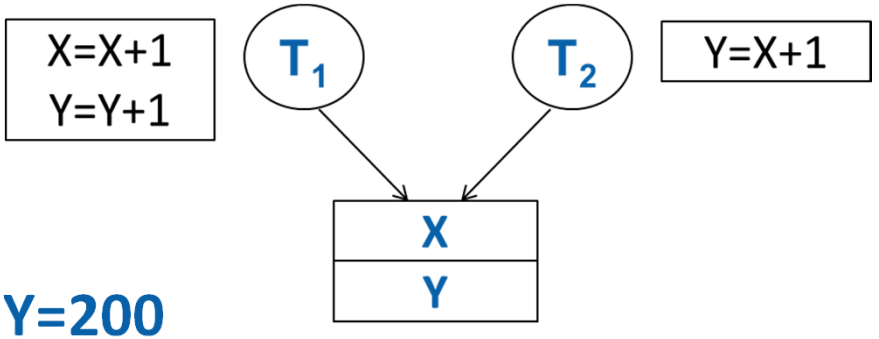
数据冲突引起的问题

- Read uncommitted data (读脏数据) (写读)
  - 在T2 commit之前，T1读了T2已经修改了的数据
- Unrepeatable reads(不可重复读) (读写)
  - 在T2 commit之前，T1写了T2已经读的数据
  - 如果T2再次读同一个数据，那么将发现不同的值
- Overwrite uncommitted data (更新丢失) (写写)
  - 在T2 commit之前，T1重写了T2已经修改了的数据

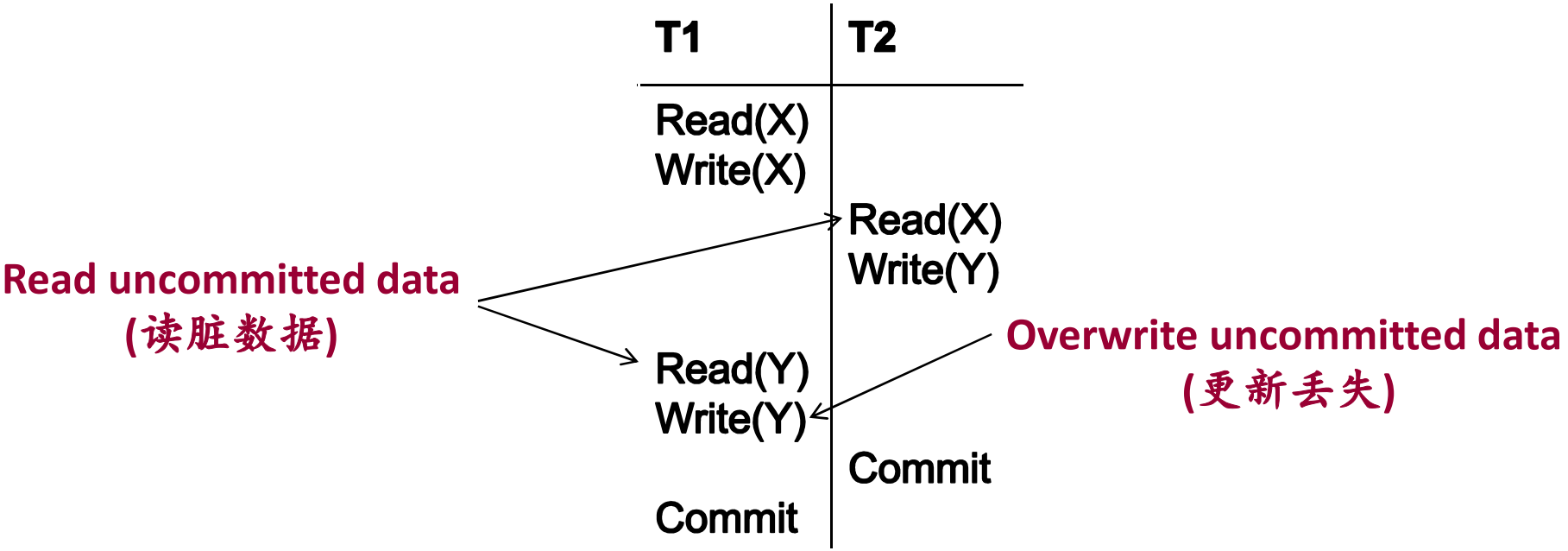
Isolation Level

	Read uncommitted data （写读）	Unrepeatable Read （读写）	Overwrite uncommitted Data （写写）
Serializable	no	no	no
Repeatable Read	no	no	possible
Read committed	no	possible	possible
Read uncommitted	possible	possible	possible

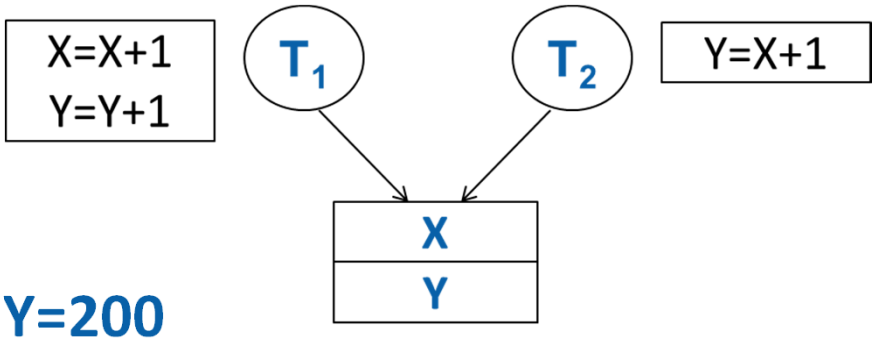
问题举例



初始： X=100, Y=200



问题举例



初始： X=100, Y=200

T1	T2
Read(X)	
	Read(X)
Write(X)	
Read(Y)	
Write(Y)	
	Write(Y)
	commit
commit	

Unrepeatable reads
(不可重复读)

Overwrite uncommitted data
(更新丢失)

两大类解决方案

- Pessimistic (悲观)
  - 假设：数据竞争可能经常出现
  - 防止：采用某种机制确保数据竞争不会出现
    - 如果一个Transaction T_1可能和正在运行的其它Transaction有冲突，那么就让这个T_1等待，一直等到有冲突的其它所有Transaction都完成为止，才开始执行。
- Optimistic (乐观)
  - 假设：数据竞争很少见
  - 检查：
    - 允许所有Transaction都直接执行
    - 但是Transaction不直接修改数据，而是把修改保留起来
    - 当Transaction结束时，检查这些修改是否有数据竞争
      - 没有竞争，成功结束，真正修改数据
      - 有竞争，丢弃结果，重新计算

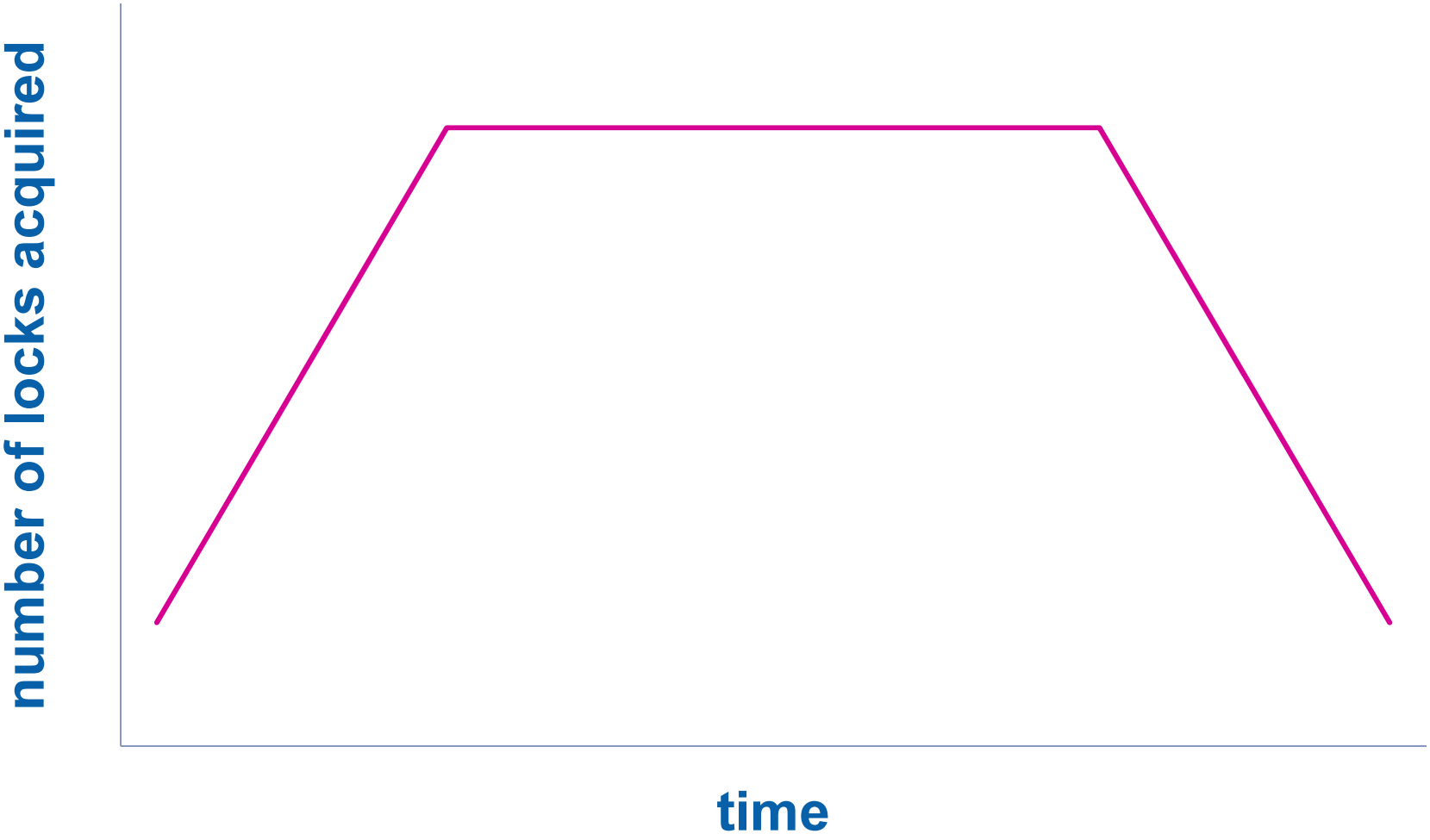
Pessimistic: 加锁

- 使用加锁协议来实现
- 对于每个事务中的SQL语句，数据库系统自动检测其中的读、写的数据
- 对事务中的读写数据进行加锁
- 通常采用两阶段加锁（2 Phase Locking）

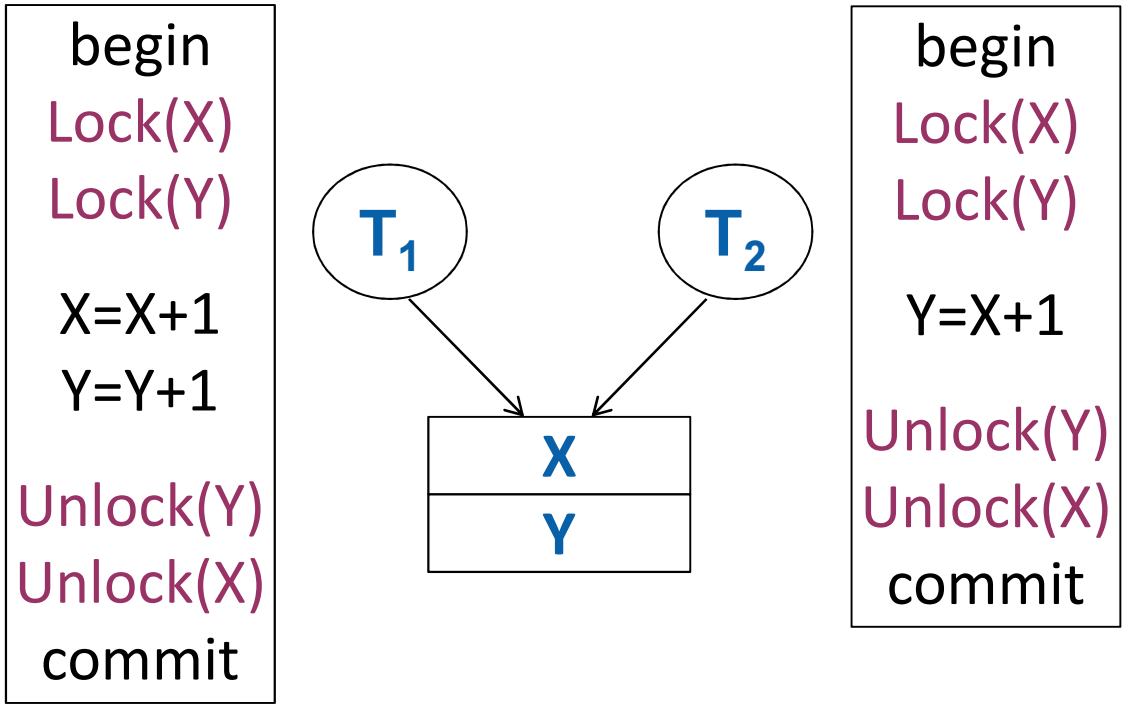
2 Phase Locking

- Pessimistic concurrency control
- 对每个访问的数据都要加锁后才能访问
- 算法如下
  - 在Transaction开始时，对每个需要访问的数据加锁
    - 如果不能加锁，就等待，直到加锁成功
  - 执行Transaction的内容
  - 在Transaction commit前，集中进行解锁
  - Commit
- 有一个集中的加锁阶段和一个集中的解锁阶段
  - 由此得名

2PL的执行过程

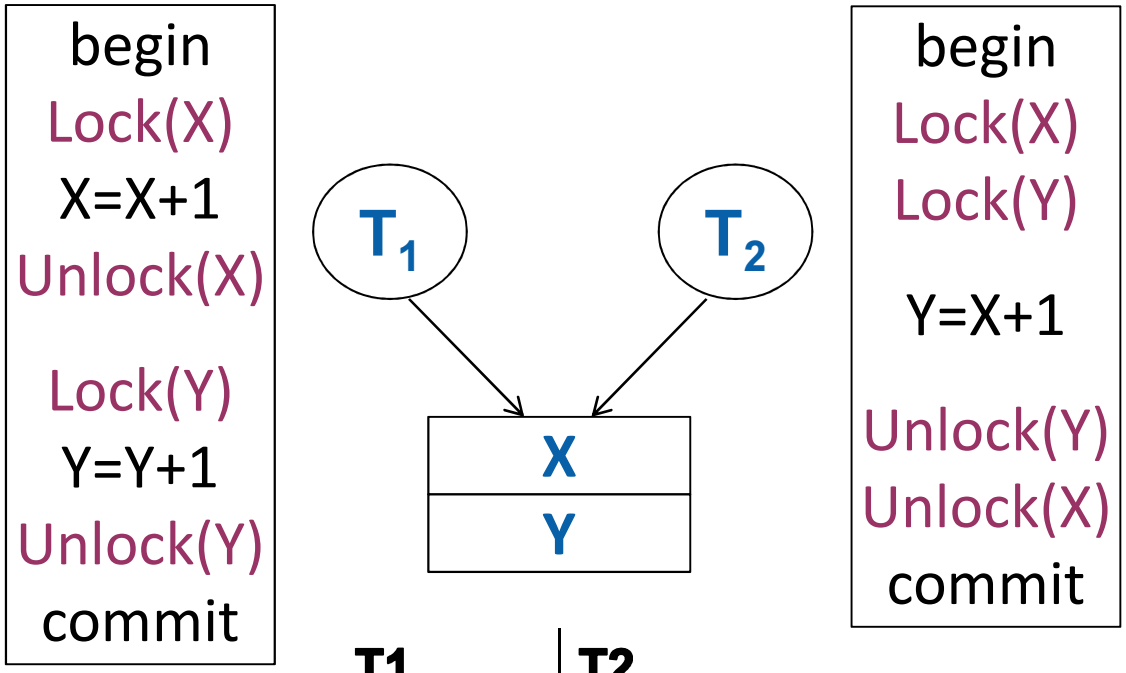


举例



- 这样一来，两个transactions不会同时执行

为什么一定要2-phase?



T1	T2
Read(X)	
Write(X)	
	Read(X)
	Write(Y)
Read(Y)	
Write(Y)	

- 有问题吗?

实现细节1：读写的锁是不同的

- Shared lock(S)：保护读操作
- Exclusive lock(X)：保护写操作

Lock Compatibility Matrix

	Shared Lock(S)	Exclusive Lock(X)
Shared Lock(S)	√	X
Exclusive Lock(X)	X	X

实现细节2： Lock Granularity

- 锁的粒度是不同的
  - Table?
  - Record?
  - Index?
  - Leaf node?
- Intent locks
  - IS(a)：将对a下面更细粒度的数据元素进行读
  - IX(a)：将对a下面更细粒度的数据元素进行写
- 为了得到S,IS: 所有祖先必须为IS或IX
- 为了得到X,IX: 所有祖先必须为IX

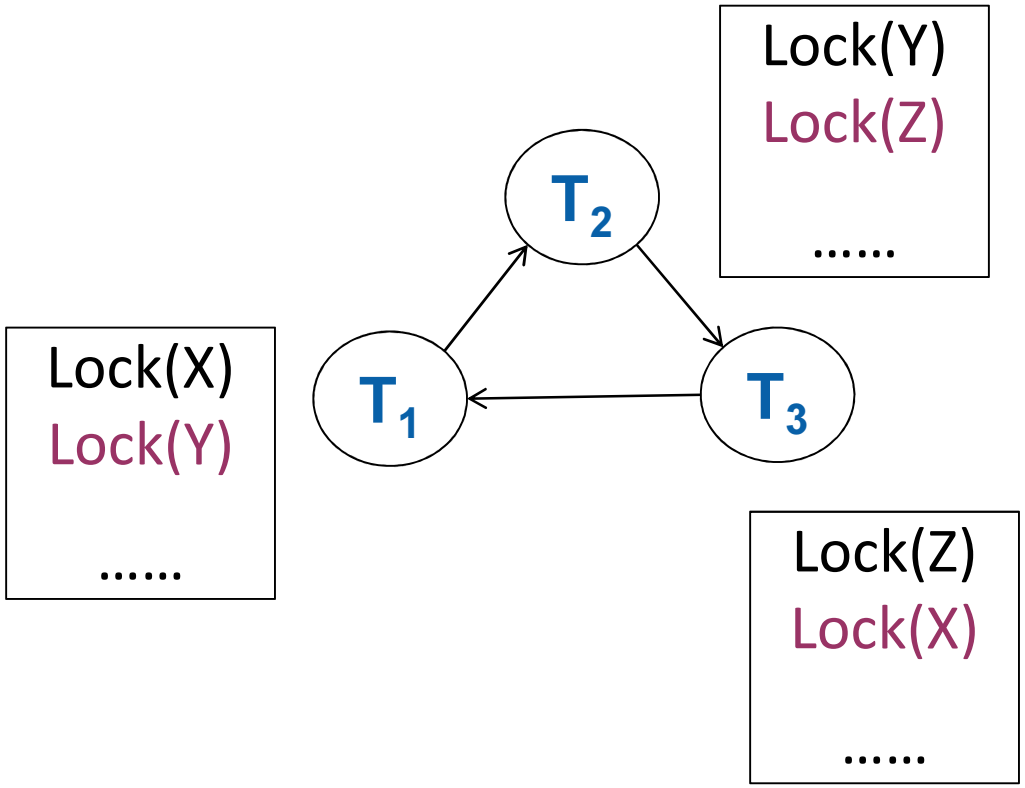
实现细节2： Lock Granularity

Lock Compatibility Matrix

	IS (intent shared)	IX (intent exclusive)	S (shared)	X (exclusive)
IS	√	√	√	X
IX	√	√	X	X
S	√	X	√	X
X	X	X	X	X

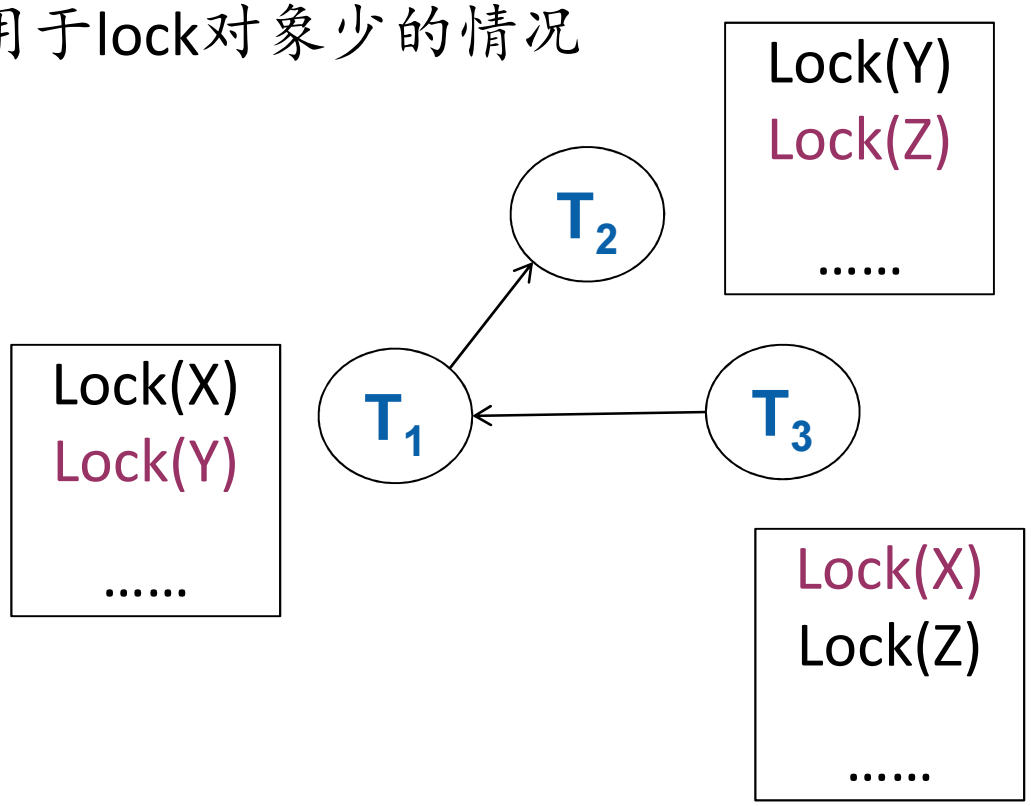
实现细节3: deadlock

- 什么情况下会出现死锁？
  - 最重要的条件：circular wait 循环等待（~~wait in a loop~~?）



如何解决deadlock问题？

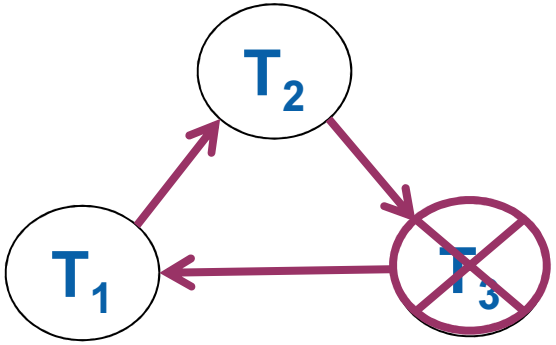
- 死锁避免
  - 规定lock对象的顺序
  - 按照顺序请求lock
  - 适用于lock对象少的情况



X, Y, Z顺序

如何解决deadlock问题？

- 数据库的lock对象很多，不适合死锁避免
- 死锁检测
  - 周期地对长期等待的Transactions检查是否有circular wait
  - 如果有，那么就选择环上其中一个Transaction abort



乐观的并发控制：不采用加锁

- 事务执行分为三个阶段
  - 读：事务开始执行，读数据到私有工作区，并在私有工作区上完成事务的处理请求，完成修改操作
  - 验证：如果事务决定提交，检查事务是否与其它事务冲突
    - 如果存在冲突，那么终止事务，清空私有工作区
    - 重试事务
  - 写：验证通过，没有发现冲突，那么把私有工作区的修改复制到数据库公共数据中

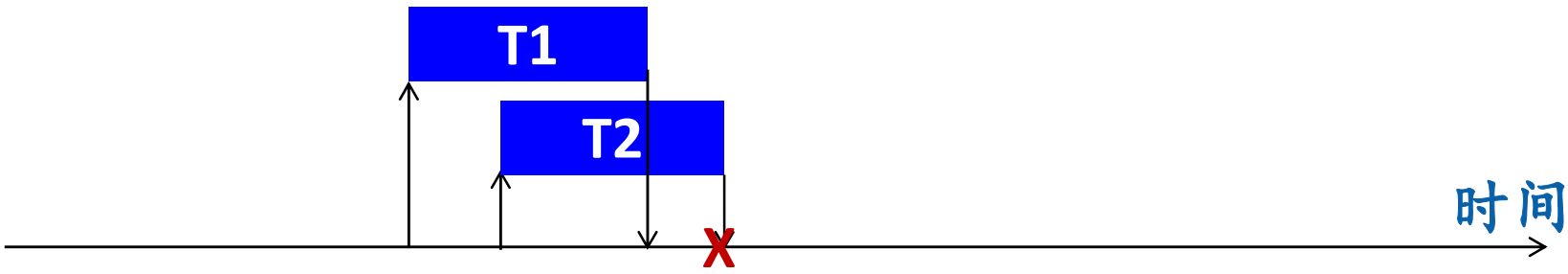
- 优点：当冲突很少时，没有加锁的开销
- 缺点：当冲突很多时，可能不断地重试，浪费大量资源，甚至无法前进

多种乐观并发控制方案

- 具体的读、验证、写的机制不同
- 有多种方案，我们这里介绍
  - Time-stamp ordering基于时间戳的并发控制
  - Snapshot Isolation和MVCC

另一种并发控制方法:Snapshot Isolation

- 一种Optimistic concurrency control
- Snapshot: 一个时点的数据库数据状态
- Transaction
  - 在起始时点的snapshot
  - 读：这个snapshot的数据
  - 写：先临时保存起来，在commit时检查有无冲突，有冲突就abort
    - First writer wins
- Multiversion concurrency control是Snapshot Isolation一个实现



另一种并发控制方法: Snapshot Isolation

- 在某些情况下, Snapshot Isolation不是Serializable的

初始：x=10; y=10

要求：x+y>=0
两个账户总和不能透支

T1

```
if (x + y >= 20) {
    x = x – 20;
}
```

T2

```
if (x + y >= 20) {
    y = y – 20;
}
```

结果：x= -10; y= -10

在Snapshot Isolation下，T1与T2可以同时正确执行。
注意：虽然在每个事物中，x+y>0，但总结果却不是了

Durability (持久性) 如何实现？

- Transaction commit后，结果持久有效，crash不消失

- 想法一
  - 在transaction commit时，把所有的修改都写回硬盘
  - 只有当写硬盘完成后，才commit

- 有什么问题？
  - 正确性问题：如果写多个page，中间掉电，怎么办？
    Atomicity被破坏了！
  - 性能问题：随机写硬盘，等待写完成

解决方案：WAL (Write Ahead Logging)

- 什么是Logging
- 什么是Write-Ahead
- 怎样保证Durability
- 怎么实现Write-Ahead Logging
- Crash Recovery

经典算法：ARIES

什么是Transactional Logging(事务日志)

- 事务日志记录(Transactional Log Record)
  - 记录一个写操作的全部信息

- 例如：记录的修改操作的日志记录

(LSN, tID, opID, pageID, slotID, columnID, old value, new value)

  - LSN: Log sequence number，是一个不断递增的整数，唯一代表一个记录；每产生一个日志记录，LSN加1
  - tID: transaction ID
  - opID: 写操作类型
  - pageID, slotID, columnID：定位到具体一个页的一个记录的一个列
  - old value, new value：旧值和新值

什么是Transactional Logging(事务日志)

- 对Transaction中每个写操作产生一个事务日志记录
- Transaction commit会产生一个commit日志记录
  - (LSN, tID, commit)
- Transaction abort会产生一个abort日志记录
  - (LSN, tID, abort)
- **日志记录被追加(append)到日志文件末尾**
  - 日志文件是一个append-only的文件
  - 文件中日志按照LSN顺序添加

什么是Write-Ahead Logging?

- Write-Ahead
  - Logging 总是先于实际的操作
  - Logging 相当于意向，先记录意向，然后再实际操作

- 写操作
  - 先Logging
  - 然后执行写操作
- Commit
  - 先记录commit 日志记录
  - 然后commit

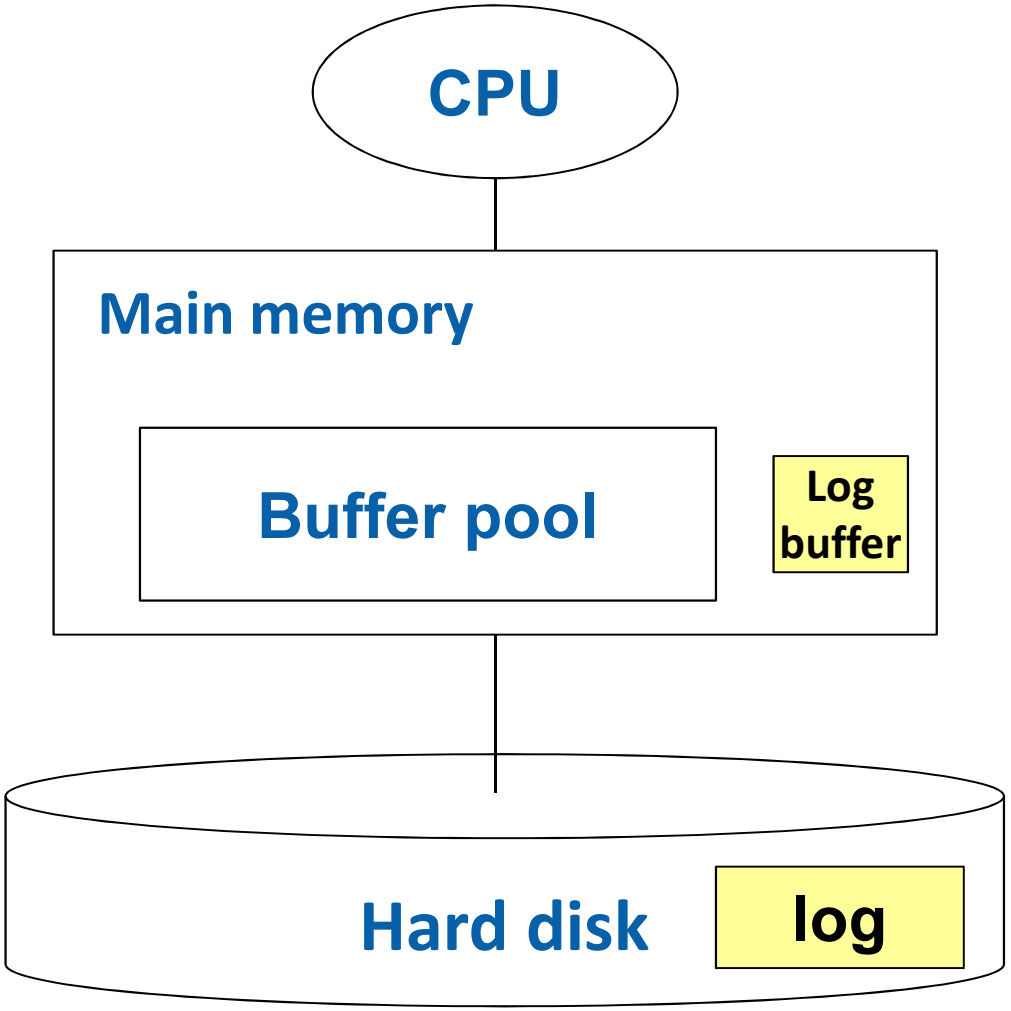
WAL怎样保证Durability？

- 条件：日志是Durable的
- 当出现掉电时，可以根据日志发现所有写操作
  - 总是先记录意向，然后实际操作
  - 所以只有存在日志记录，相应的操作才有可能发生
- 对于一个Transaction，寻找它的commit日志记录
  - 如果找到，那么这个transaction 已经commit了
  - 如果没找到，那么这个transaction 没有完成
- 已Commit
  - 根据日志记录，确保所有的写操作都完成了
- 没有commit
  - 根据日志记录，对每个写操作检查和恢复原值

如何保证日志Durable?

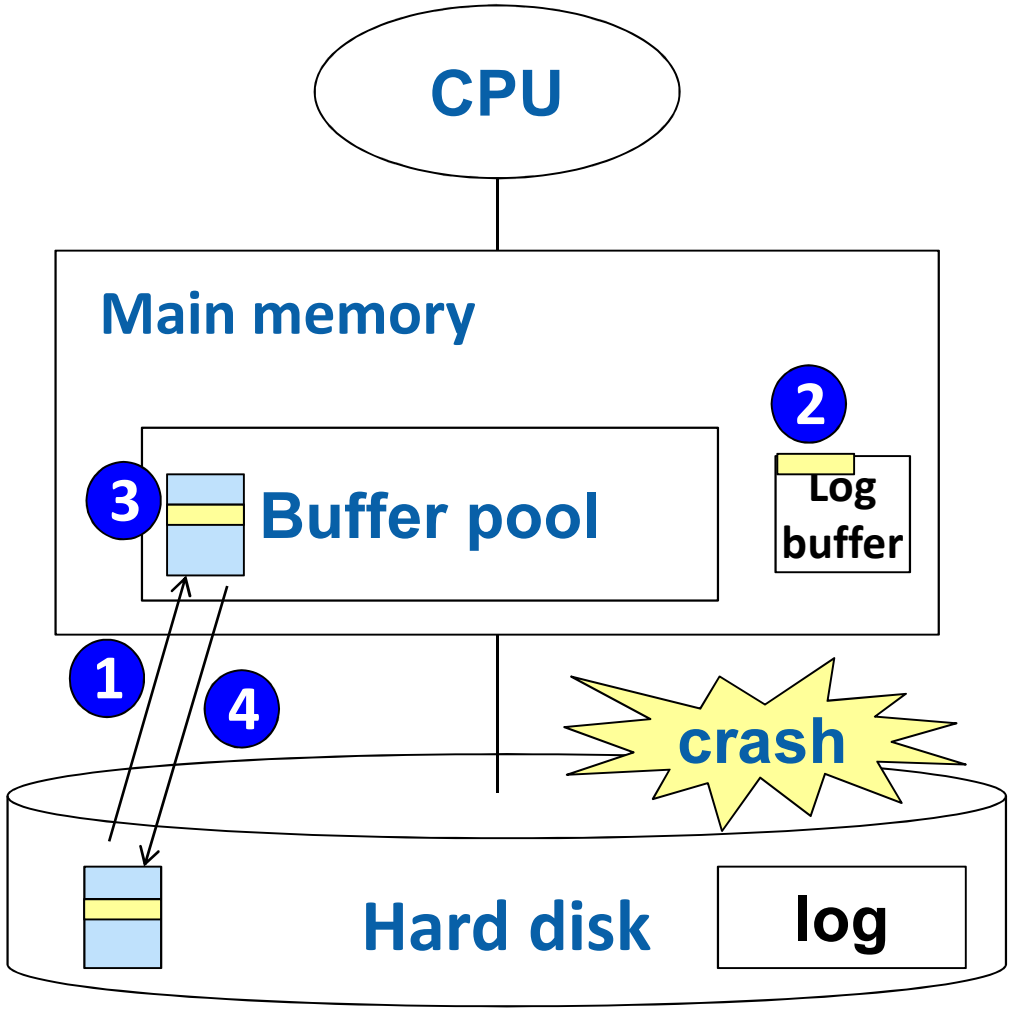
- 简单方法：写日志记录时保证日志记录Durable
  - write + flush
  - 必须执行一个写操作，然后用flush保证写操作确实写到硬盘上了，并且等待flush结束
  - 这个过程通常需要~10ms
- 简单计算
  - 假设每个transaction需要修改10个记录
  - 那么上述写日志就需要~100ms
  - 硬盘同时只能执行一个写操作
  - 所以系统的throughput为10 transaction/second
    - 如果每个transaction修改100个记录，会怎么样？
- 太慢了！

实现： WAL (Write Ahead Logging)



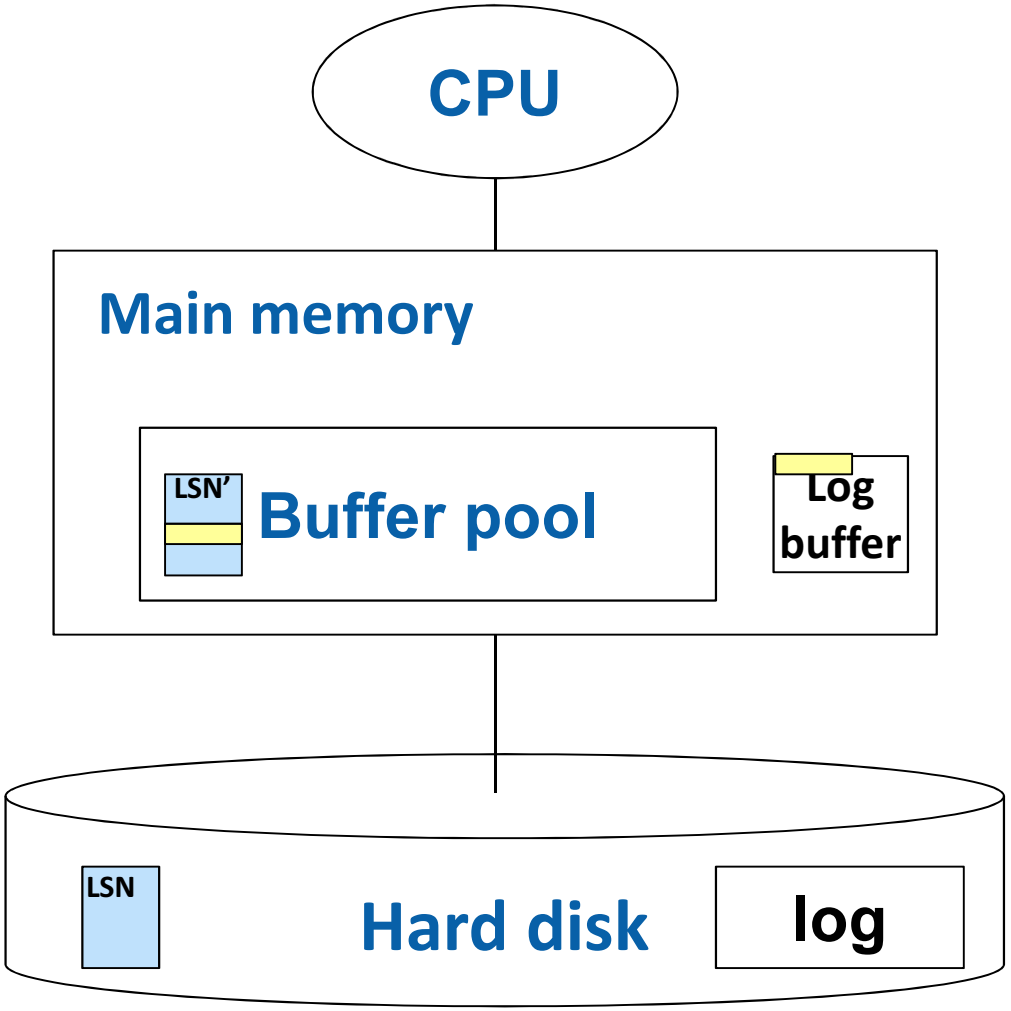
- **Log**: 硬盘上日志文件
- **Log buffer**：在内存中分配一个缓冲区

实现：WAL (Write Ahead Logging)



- 日志写在log buffer中
- 当commit时write+flush log buffer
- 这样性能好了，但有什么问题？
  - Dirty page可能被写回硬盘！
  - 掉电后，硬盘上数据已经修改，但是log没有记录！

实现： WAL (Write Ahead Logging)



- 日志写在log buffer中
- 当commit时write+flush log buffer
- 解决方法：
  - Page header记录本page 最新写的LSN
  - Buffer pool在替代写回一个dirty page时，必须保证page LSN之前的所有日志已经flush过了
- **保证：日志记录一定是先于修改后的数据出现在硬盘上**

Checkpoint（检查点）

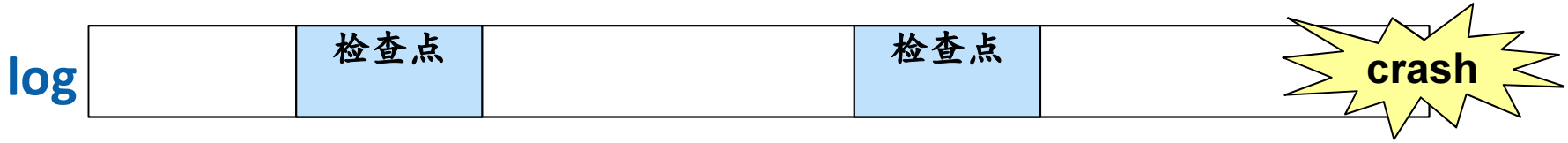
- 为什么要用checkpoint?
  - 为了使崩溃恢复的时间可控
  - 如果没有checkpoint，可能需要读整个日志，redo/undo很多工作
- 定期执行checkpoint
- checkpoint的内容
  - 当前活动的事务表：包括事务的最新日志的LSN
  - 当前脏页表：每个页最早的尚未写回硬盘的LSN

Log Truncation

- Log file不能无限地增长
- 什么情况下一个日志记录不需要了？
  - 对应的transaction完成了
  - 对应的写操作已经在硬盘上了
- 如果LSN之前的所有日志记录都不需要了，那么就可以删除LSN之前的Log – log truncation
  - Hot page 问题：一个page经常被更新，总是在buffer pool中，很长时间也不写回硬盘，而硬盘上对应的page很长时间没有更新，使得log truncation 难以进行
  - 定期地把长时间缓存在buffer pool中的dirty page写回（例如，在检查点时做）
  - LSN= min(脏页最早的尚未写回硬盘的LSN)
    - 这个LSN之前的所有日志都可以丢弃

Crash Recovery

- 系统定期把当前活跃的Transaction信息(tID, earliest LSN)记录在log中



- Crash后重新启动
- ARIES算法
  - 分析阶段
  - redo阶段
  - undo阶段

崩溃恢复：分析阶段

- 找到最后一个检查点
  - 检查点的位置记录在硬盘上一个特定文件中
  - 读这个文件，可以得知最后一个检查点的位置
- 找到日志崩溃点
  - 如果是掉电等故障，必须找到日志的崩溃点
  - 当日志是循环写时，需要从检查点扫描日志，检查每个日志页的校验码，发现校验码出错的位置，或者LSN变小的位置
- 确定崩溃时的活跃事务和脏页
  - 最后一个检查点时的活跃事务表和脏页表
  - 正向扫描日志，遇到commit, rollback, begin更新事务表
    - 同时记录每个活动事务的最新LSN
  - 遇到写更新脏页表
    - 同时记录每个页的最早尚未写回硬盘的LSN

崩溃恢复：Redo阶段

- 目标：把系统恢复到崩溃前瞬间的状态
- 找到所有脏页的最早的LSN
- 从这个LSN向日志尾正向读日志
  - Redo每个日志修改记录
- 对于一个日志记录
  - 如果其涉及的页不在脏页表中，那么跳过
  - 如果数据页的LSN>=日志的LSN，那么跳过
    - 数据页已经包含了这个修改
  - 其它情况，修改数据页

崩溃恢复：Undo阶段

- 目标：清除未提交的事务的修改
- 对于所有在崩溃时活跃的事务
  - 找到这个事务最新的LSN
  - 通过反向链表，读这个事务的所有日志记录
- undo所有未提交事务的修改
  - Undo时，比较数据页的LSN和日志的LSN
  - if (数据页LSN>=日志LSN) 时，才进行undo

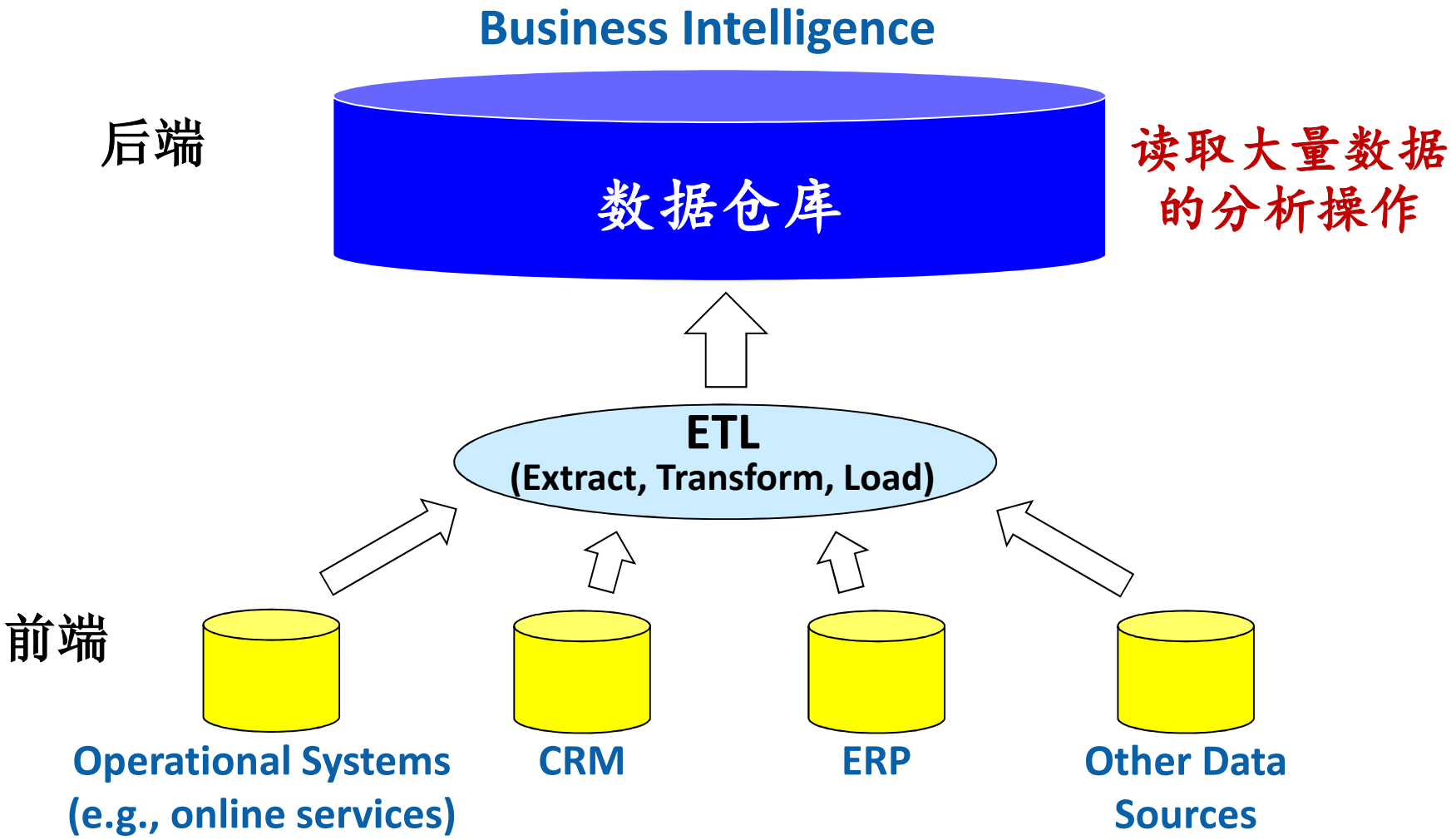
介质故障的恢复

- 如果硬盘坏了，那么日志可能也损坏了
  - 无法正常恢复
- 硬件的方法：RAID
- 如果整个RAID坏了，怎么办？
- 需要定期replicate备份数据库
  - 备份数据库数据
  - 更频繁地备份事务日志
  - 那么就可以根据数据和日志恢复数据库状态
  - 例如：双机系统

Outline

- 事务处理
- 数据仓库
  - OLAP
  - 行式与列式数据库
- 分布式数据库

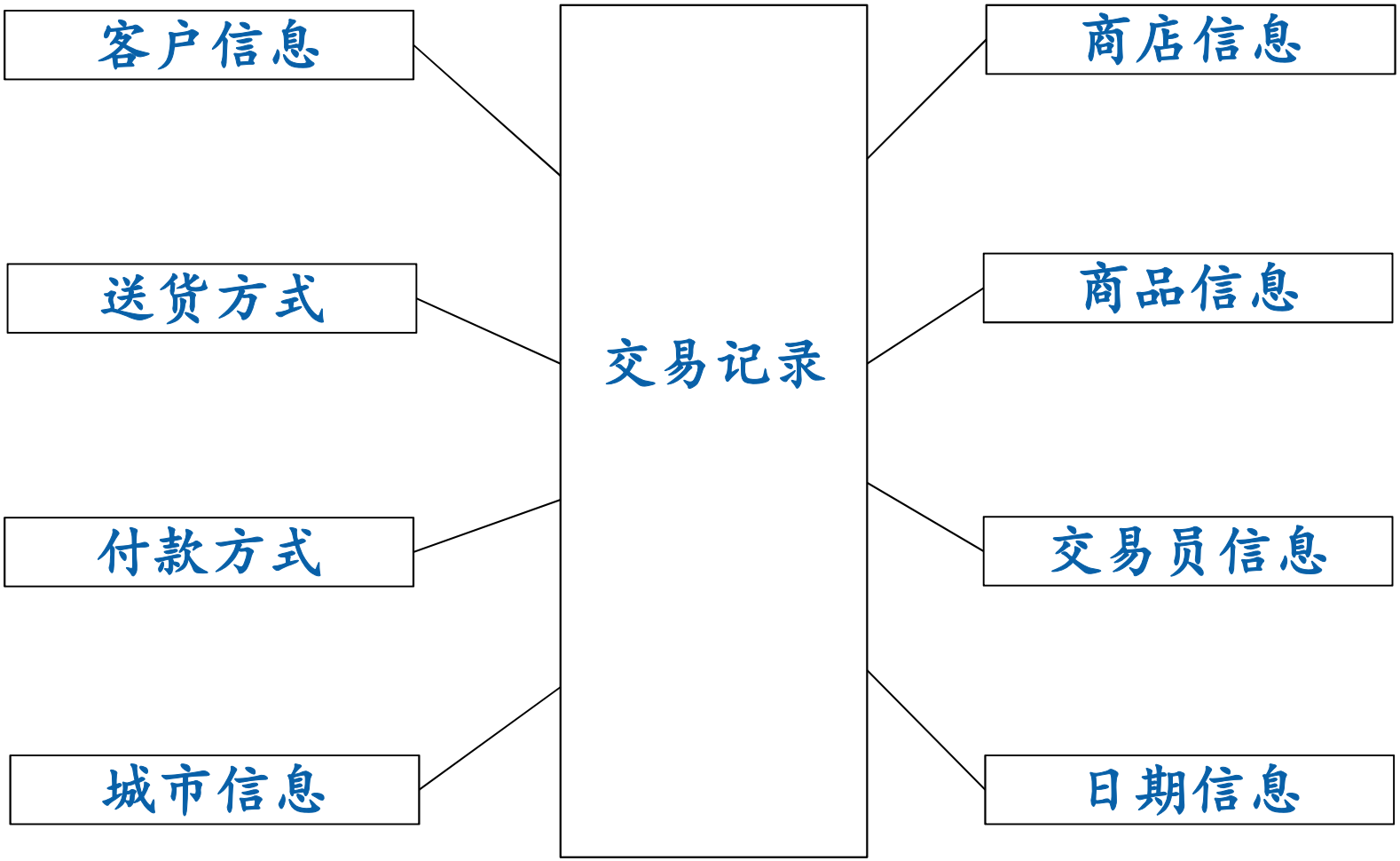
数据仓库 (Data Warehouse)



数据仓库 vs. 事务处理

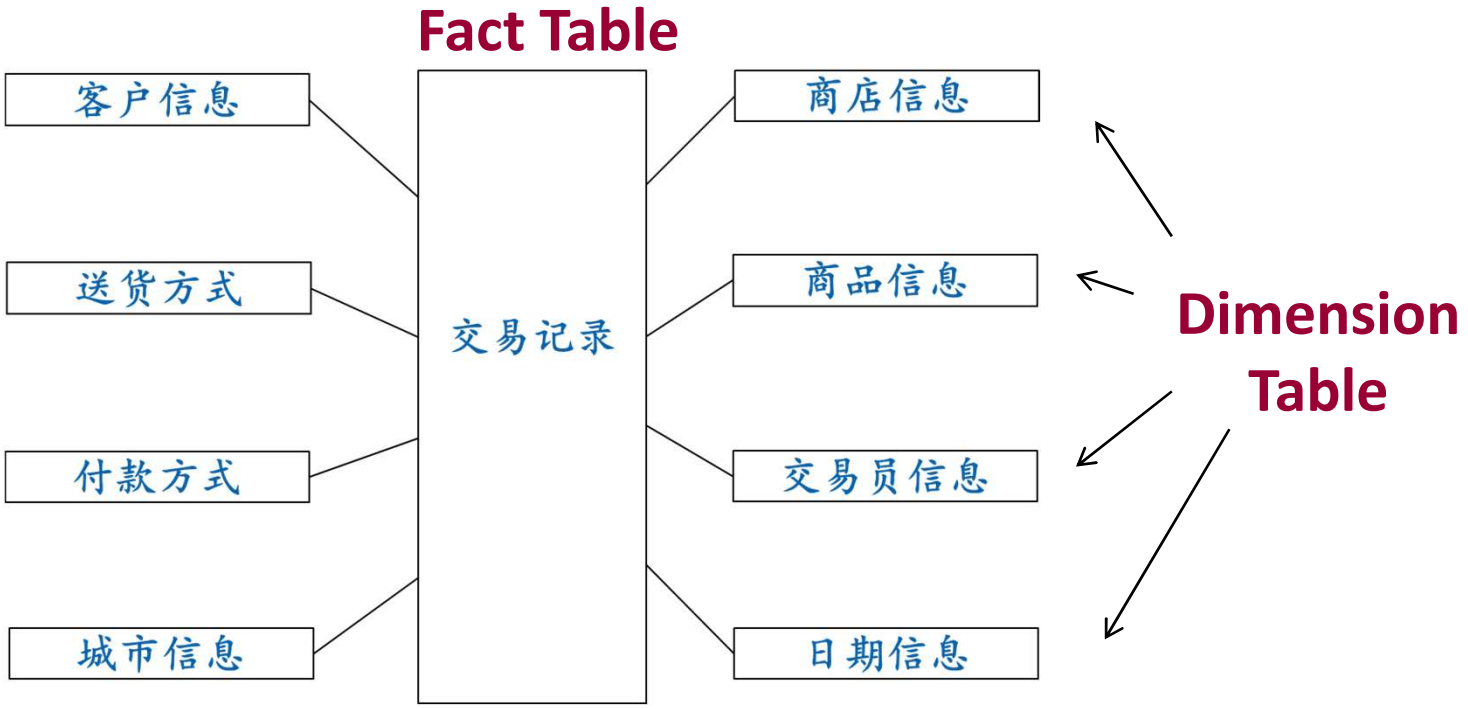
- 数据仓库
  - 少数数据分析操作
  - 每个操作访问大量的数据
  - 分析操作以读为主
- 事务处理
  - 大量的并发transactions
  - 每个transaction访问很少的数据
  - 读写

Star Schema：数据仓库中常见



Star Schema：数据仓库中常见

- 一个很大的fact table，多个dimension table
- Primary key – foreign key



常见的query形式

select …
from *fact*, *dim1*, *dim2*, …, *dimk*
where (*dim1.a op val1*) and
(*dim2.b op val2*) and
……
(*dimk.z op valk*) and
join key constraints
group by …
having …

在dimension表上施加约束条件，
对fact表进行统计分析

OLAP

- Online Analytical Processing（联机分析处理）
- 数据仓库通常是OLAP的基础
  - OLAP是在数据仓库的基础上实现的
- OLAP的基本数据模型是多维矩阵
  - 例如，在多个dimension上进行group by操作
  - 得到的多维矩阵的每项代表一个分组，
    每项的值是Fact表上对于这个分组的聚集统计值
- 称作：Data Cube（数据立方）

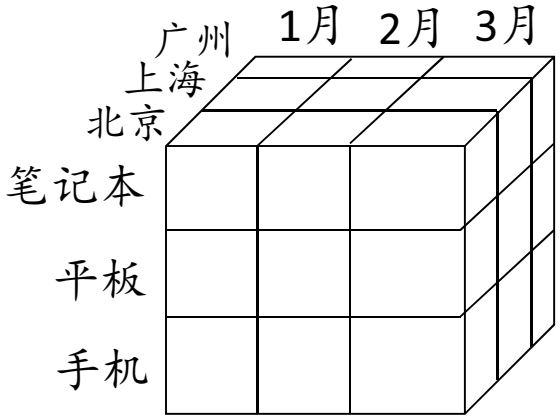
Data Cube(数据立方)

- 例如，二维的数据立方，记录分组的统计数据

时间

商品		1月	2月	3月
	笔记本	1000	1500	1600
	平板	2000	2500	3000
	手机	3000	3100	3200

- 例如，三维的数据立方



Data Cube(数据立方)

- 多维的数据表示
  - 适合对趋势的分析
  - 可以从宏观到微观，从微观到宏观
- 常用操作
  - Roll up / drill down
  - Slice, dice

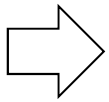
Data Cube(数据立方)：rollup（上卷）

- 例如，二维的数据立方

Rollup 时间维度

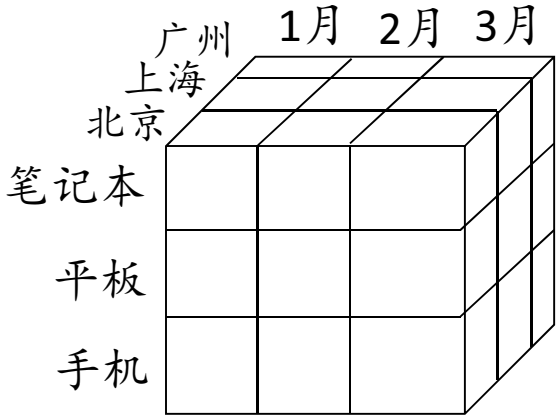
时间

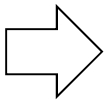
商品		1月	2月	3月
	笔记本	1000	1500	1600
	平板	2000	2500	3000
	手机	3000	3100	3200

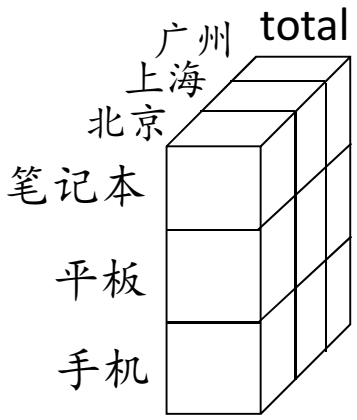


商品	total
笔记本	4100
平板	7500
手机	9300

- 例如，三维的数据立方







Data Cube(数据立方)：drill down(下钻)

- 例如，二维的数据立方

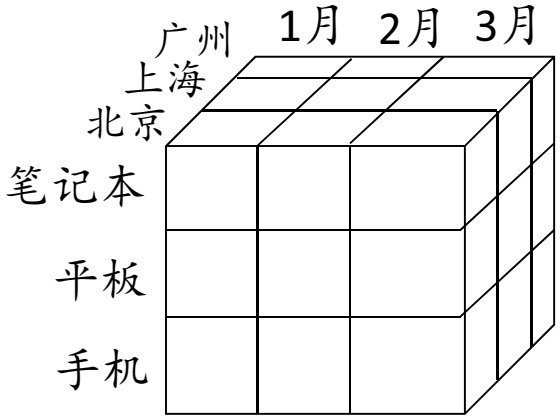
时间维度

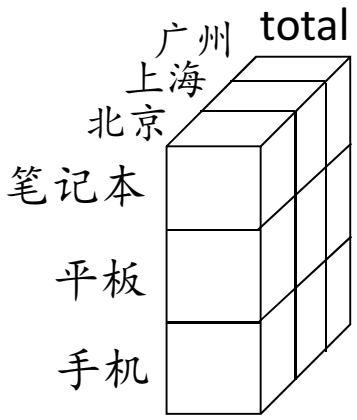
时间

商品	1月	2月	3月
笔记本	1000	1500	1600
平板	2000	2500	3000
手机	3000	3100	3200

商品	total
笔记本	4100
平板	7500
手机	9300

- 例如，三维的数据立方





概念层级

- 在一个维度上有可能可以定义层级
  - 时间：年-月-日
  - 地点：国家-省-市
  - 商品：品种-具体型号-不同厂家的同类产品
  - 等等
- 前面的例子
  - Roll up：在某维上求和，降维
  - Drill down：把某维的和分解，增维
- 还可以对概念层级操作
  - Roll up：在某维上，从细粒度到粗粒度
  - Drill down：在某维上，从粗粒度到细粒度

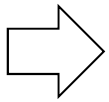
Data Cube(数据立方)：slice(切片)

在某维上选一个值

- 例如，二维的数据立方

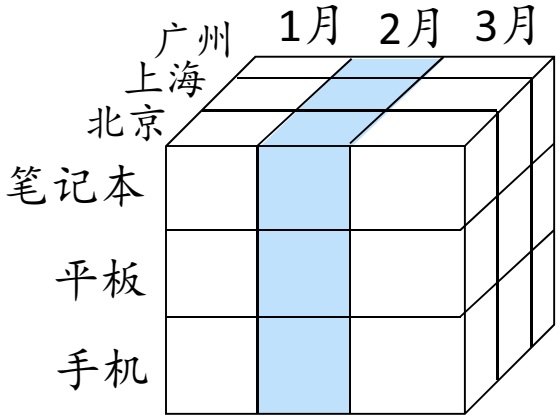
时间

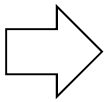
商品	1月	2月	3月
笔记本	1000	1500	1600
平板	2000	2500	3000
手机	3000	3100	3200

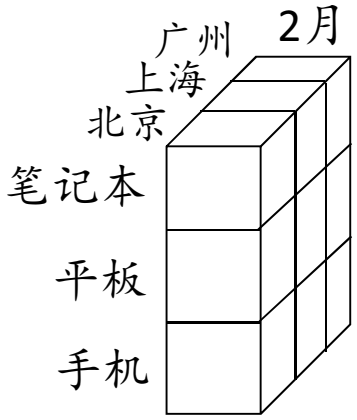


商品	2月
笔记本	1500
平板	2500
手机	3100

- 例如，三维的数据立方







Data Cube(数据立方)：dice（切块）

在多维上选多个值

- 例如，二维的数据立方

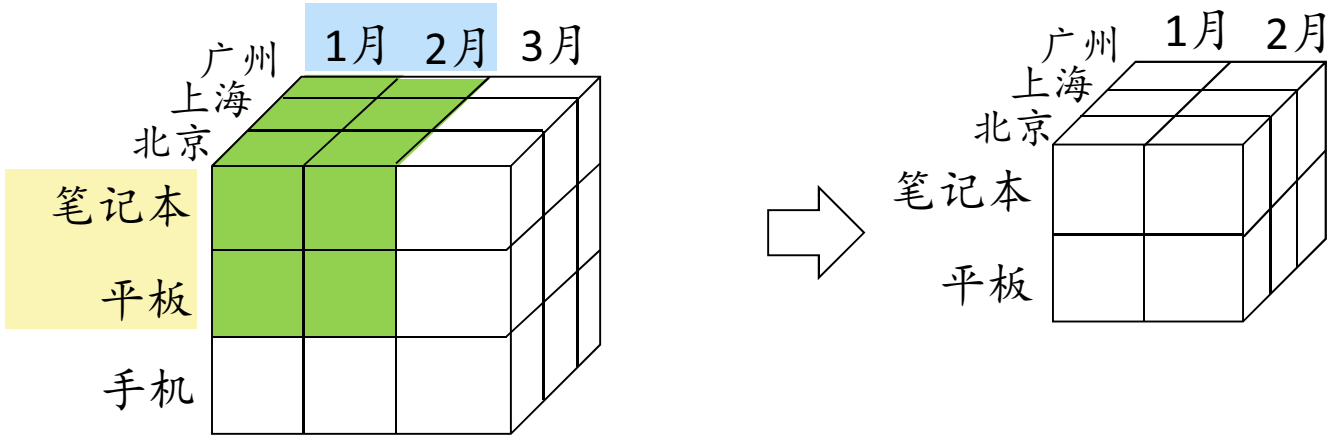
时间

商品		1月	2月	3月
	笔记本	1000	1500	1600
	平板	2000	2500	3000
	手机	3000	3100	3200

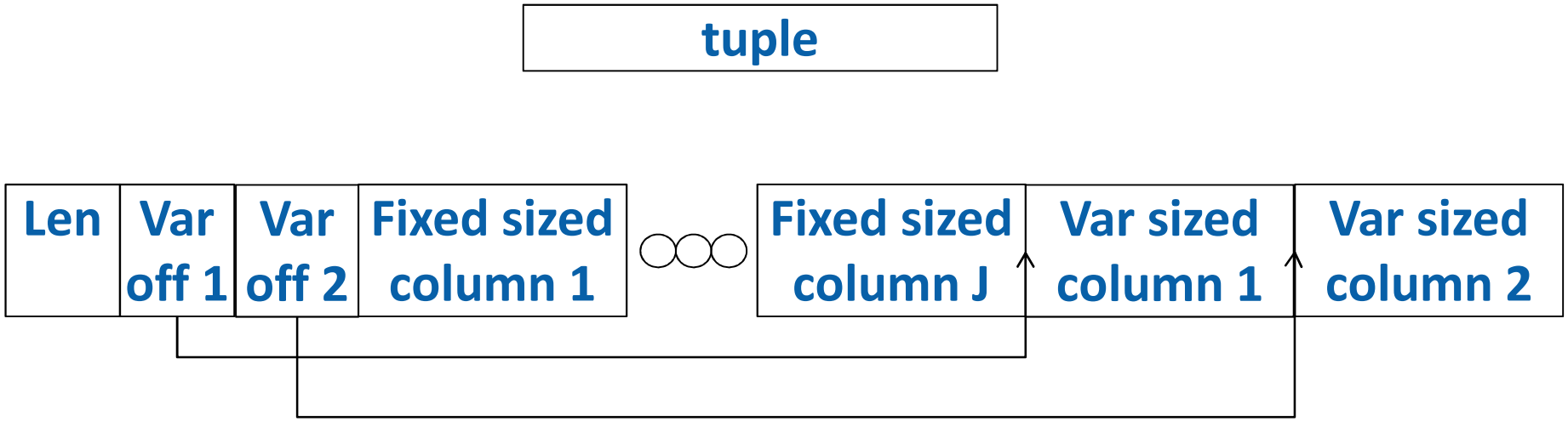
时间

商品		1月	2月
	笔记本	1000	1500
	平板	2000	2500

- 例如，三维的数据立方



行式数据存储



- 每个记录中把所有的列相邻地存放
- 优点
  - 多个列的值，可以一次I/O都得到
  - **适合于OLTP**，同时需要读写同一个记录的多个列的值
- 对于数据分析操作有什么问题？ **只使用少数列**

列式数据存储

- 每个列产生一个文件，存储所有记录中该列的值

ID	Name	Birthday	Gender	Major	Year	GPA
131234	张飞	1995/1/1	男	计算机	2013	85
145678	貂蝉	1996/3/3	女	经管	2014	90
129012	孙权	1994/5/5	男	法律	2012	80
121101	关羽	1994/6/6	男	计算机	2012	90
142233	赵云	1996/7/7	男	计算机	2014	95

存储为7个列文件

ID
131234
145678
129012
121101
142233

Name
张飞
貂蝉
孙权
关羽
赵云

Birthday
1995/1/1
1996/3/3
1994/5/5
1994/6/6
1996/7/7

Gender
男
女
男
男
男

Major
计算机
经管
法律
计算机
计算机

Year
2013
2014
2012
2012
2014

GPA
85
90
80
90
95

为什么要用列式存储？

- 数据仓库的分析查询
  - 大部分情况只涉及一个表的少数几列
  - 会读一大部分记录
- 在这种情况下，行式存储需要读很多无用的数据
- 采用列式存储可以降低读的数据量

ID	Name	Birthday	Gender	Major	Year	GPA
131234	张飞	1995/1/1	男	计算机	2013	85
145678	貂蝉	1996/3/3	女	经管	2014	90
129012	孙权	1994/5/5	男	法律	2012	80
121101	关羽	1994/6/6	男	计算机	2012	90
142233	赵云	1996/7/7	男	计算机	2014	95

列式存储的压缩

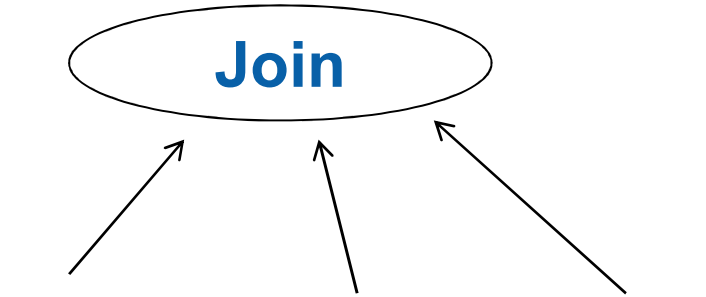
- 每个文件存储相同数据类型的值
- 数据更容易被压缩
- 比行式存储有更高的压缩比

Year
2013
2014
2012
2012
2014

可以有多种简单的方法压缩

列式存储的问题

- 如果用到了一个表的多个列
- 多列需要拼装在一起，付出拼装代价



Name
张飞
貂蝉
孙权
关羽
赵云

Major
计算机
经管
法律
计算机
计算机

Year
2013
2014
2012
2012
2014

解决这一问题

- 思路1
  - 数据存储两份，一份行式，一份列式（Fractured Mirrors）
- 思路2
  - 如果多个列总是一起使用
  - 那么就把它们存在一起
  - Vertica数据库的 ‘projection’

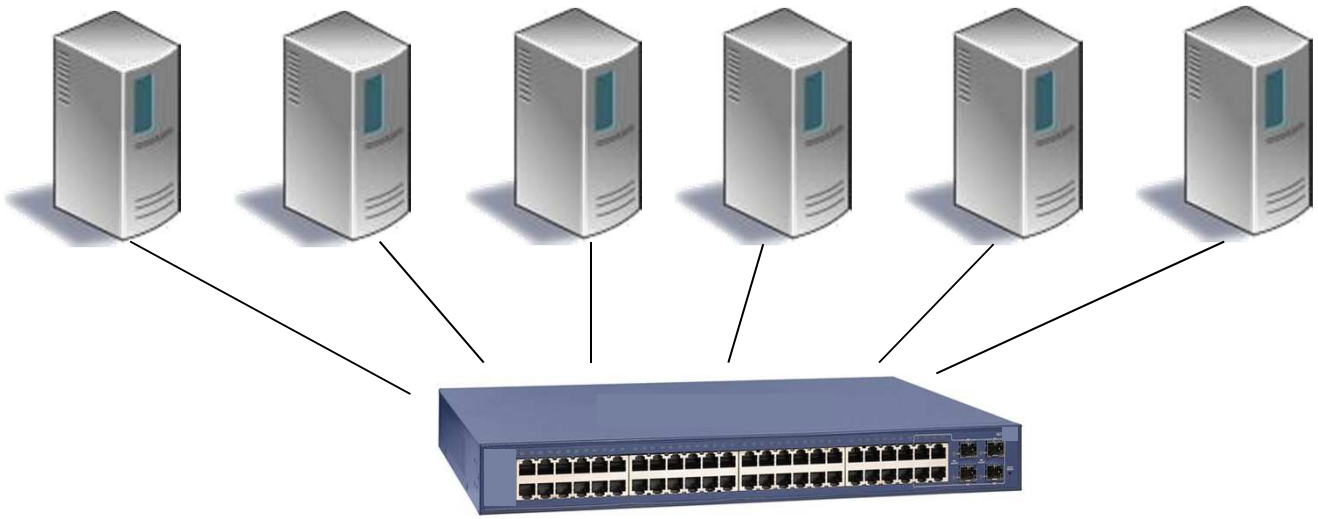
Outline

- 事务处理
- 数据仓库
- 分布式数据库
  - 系统架构
  - 分布式查询处理
  - 分布式事务处理

三种架构

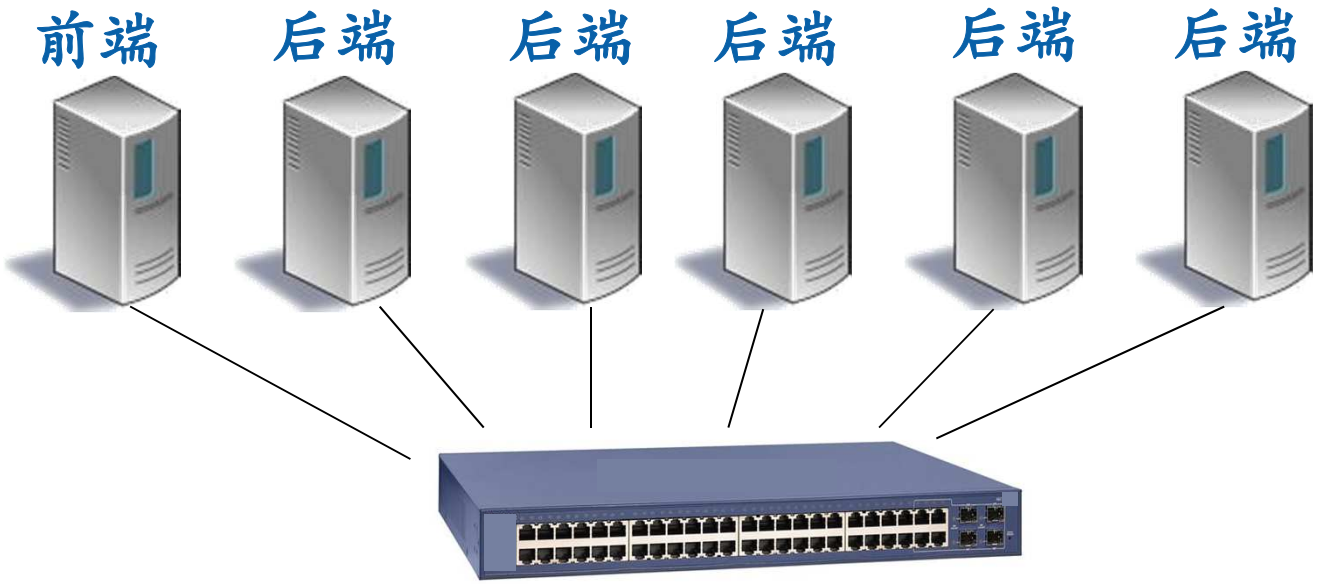
- Shared memory
  - 多芯片、多核
  - 或Distributed shared memory
- Shared disk
  - 多机连接相同的数据存储设备
- Shared nothing
  - 普通意义上的机群系统
  - 由以太网连接多台服务器

Shared Nothing



- 系统架构
- 关键技术

系统架构

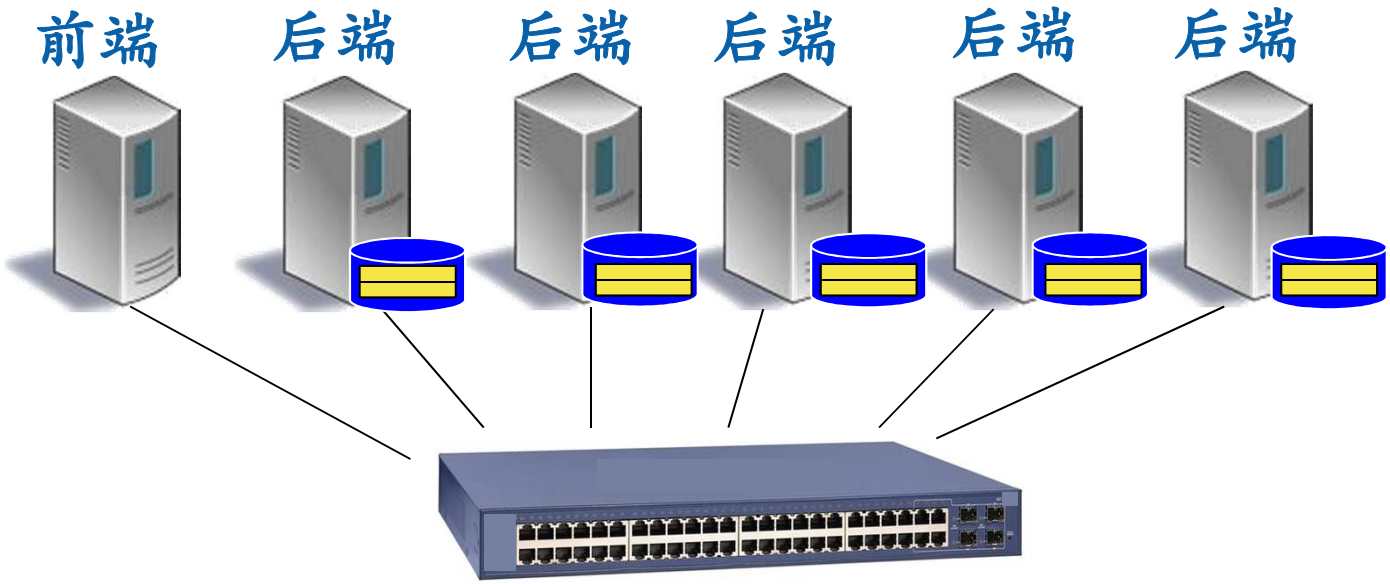


- 一个coordinator运行前端产生并行的query plan
- 每台worker服务器上都有后端
- Coordinator协调worker服务器执行

关键技术

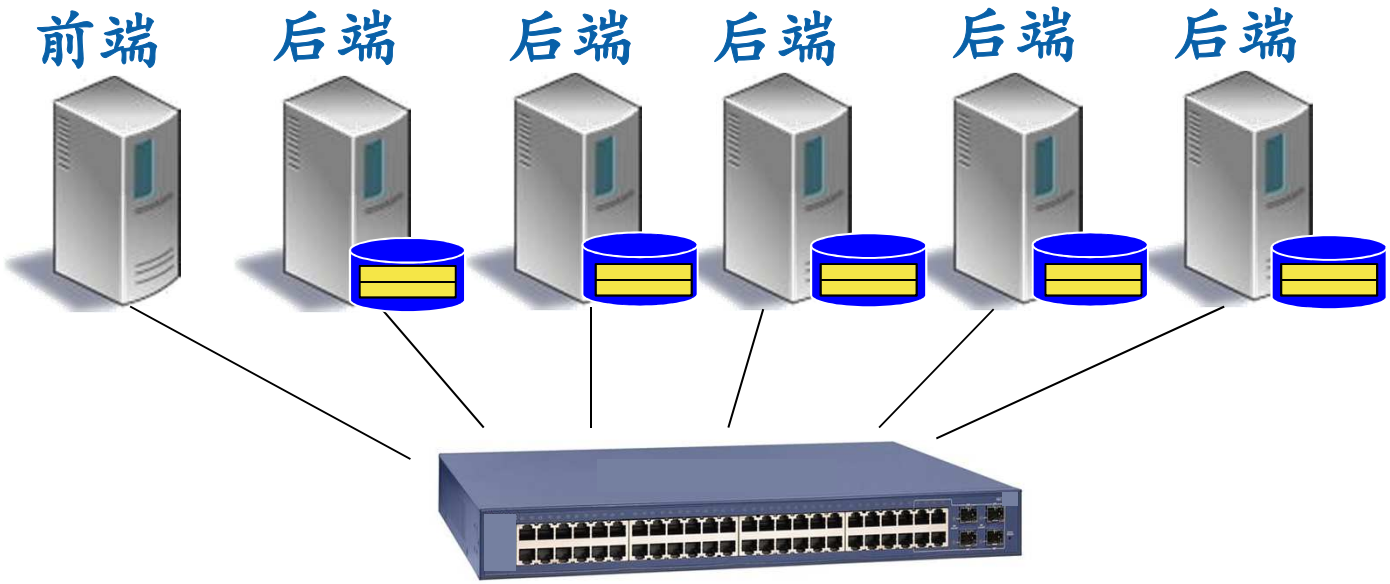
- Partitioning（划分）
  - 把数据分布在多台服务器上
  - 通常采用Horizontal partitioning
    - 把不同的记录分布在不同的服务器上
- Replication（备份）
  - 为了提高可靠性
  - 对性能的影响
    - 读？可能提高并行性
    - 写？额外代价

Horizontal Partitioning



- Hash partitioning
  - 类似GRACE: machine ID = hash(key) % MachineNumber
- Range partitioning
  - 每台服务器负责一个key的区间，所有区间都不重叠

并行执行

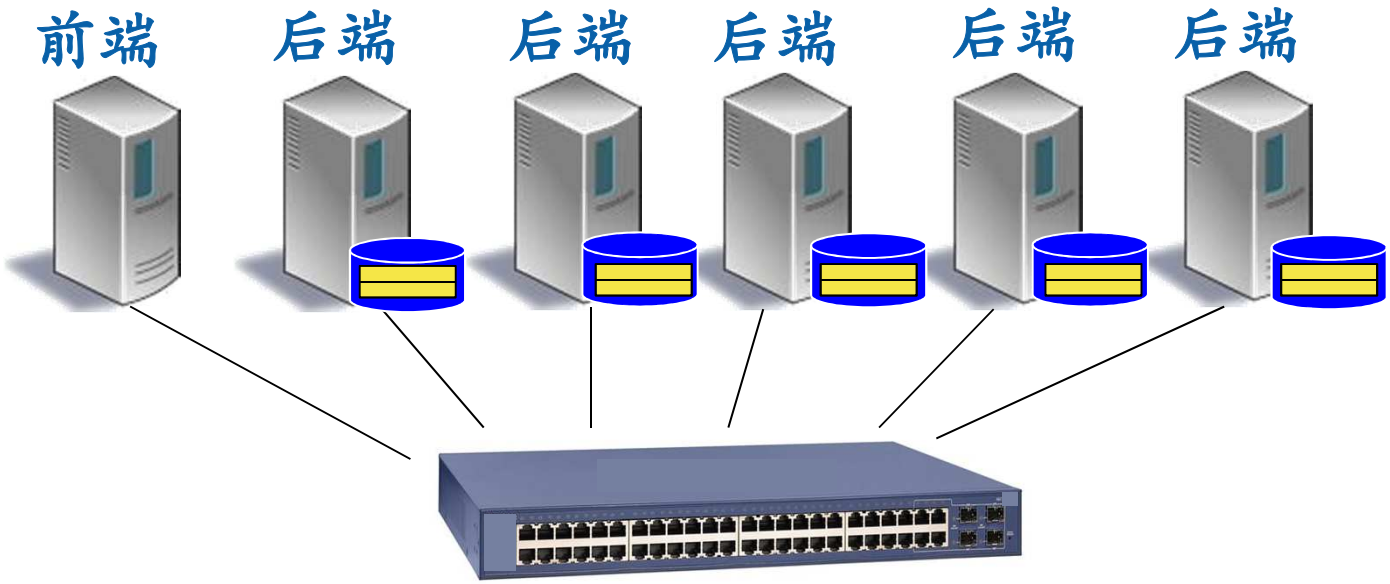


- Filter √
- Project √

☞ Join 可以并行执行吗？

Join

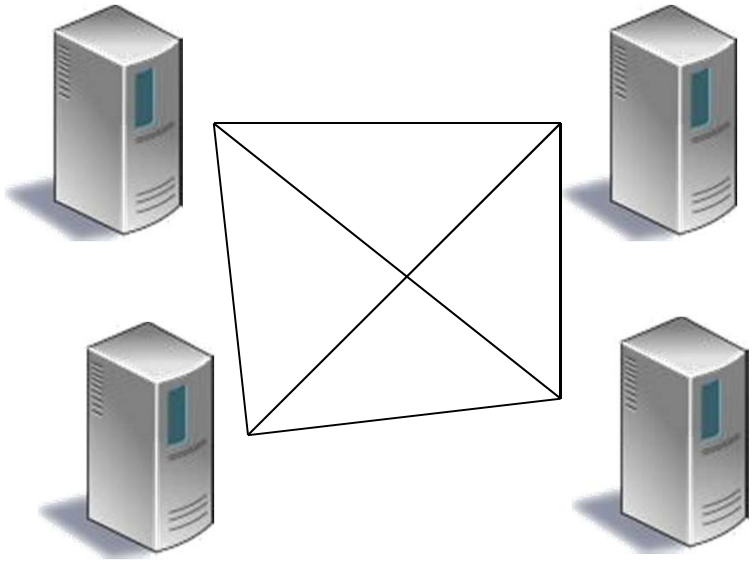
$R \bowtie_{R.a = S.b} S$



- 如果partition key就是join key，那么类似GRACE可以并行执行，每台机器单机join
- 其它情况？

partition key不是join key？ $R \bowtie_{R.a = S.b} S$

- 在Join key上进行分布式partitioning
  - 这一步类似GRACE中的I/O partitioning
  - 使同一个划分放在同一台机器上
  - 需要大量的数据传输
- 然后再join



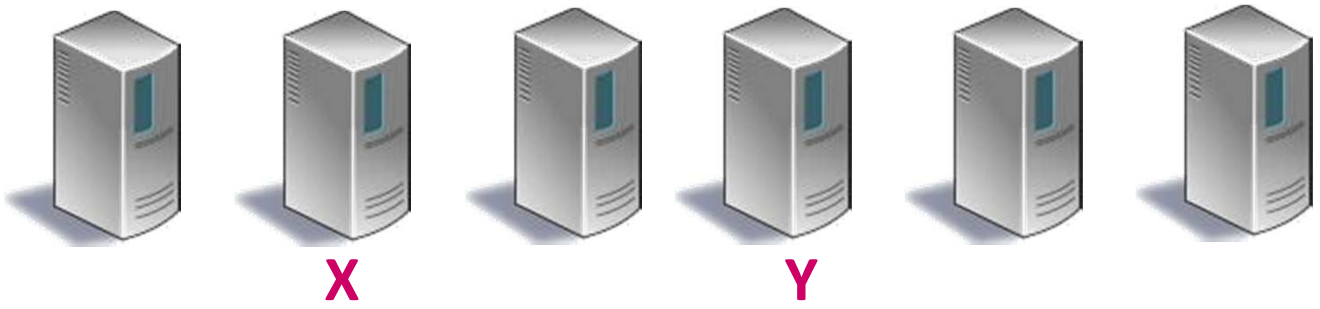
Semi Join

$R \ltimes_{R.a = S.b} S$

- 目标：减少数据传输开销
- 思路：把没有匹配的记录过滤掉、不发送
- 方法：用S.b过滤R
  - 提取S.b，把S.b发送到R所在的所有机器
  - 用S.b对R进行过滤，找出存在匹配的R记录
  - 在对R划分数据传输时，只需要针对这些存在匹配的记录
- 同样，可以用R.a对S进行过滤

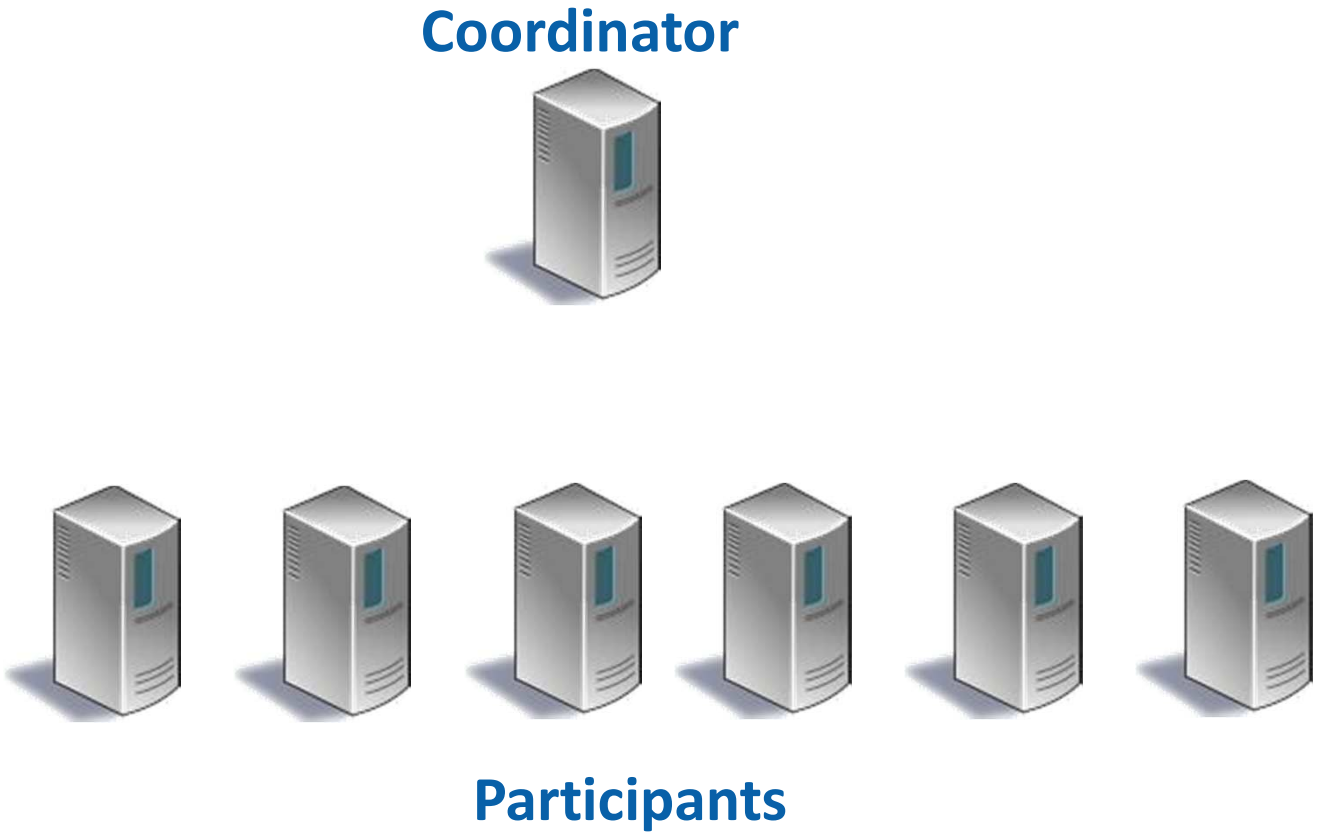
分布式事务

```
Begin Transaction;
X=X+1;
Y=Y+1;
Commit Transaction;
```



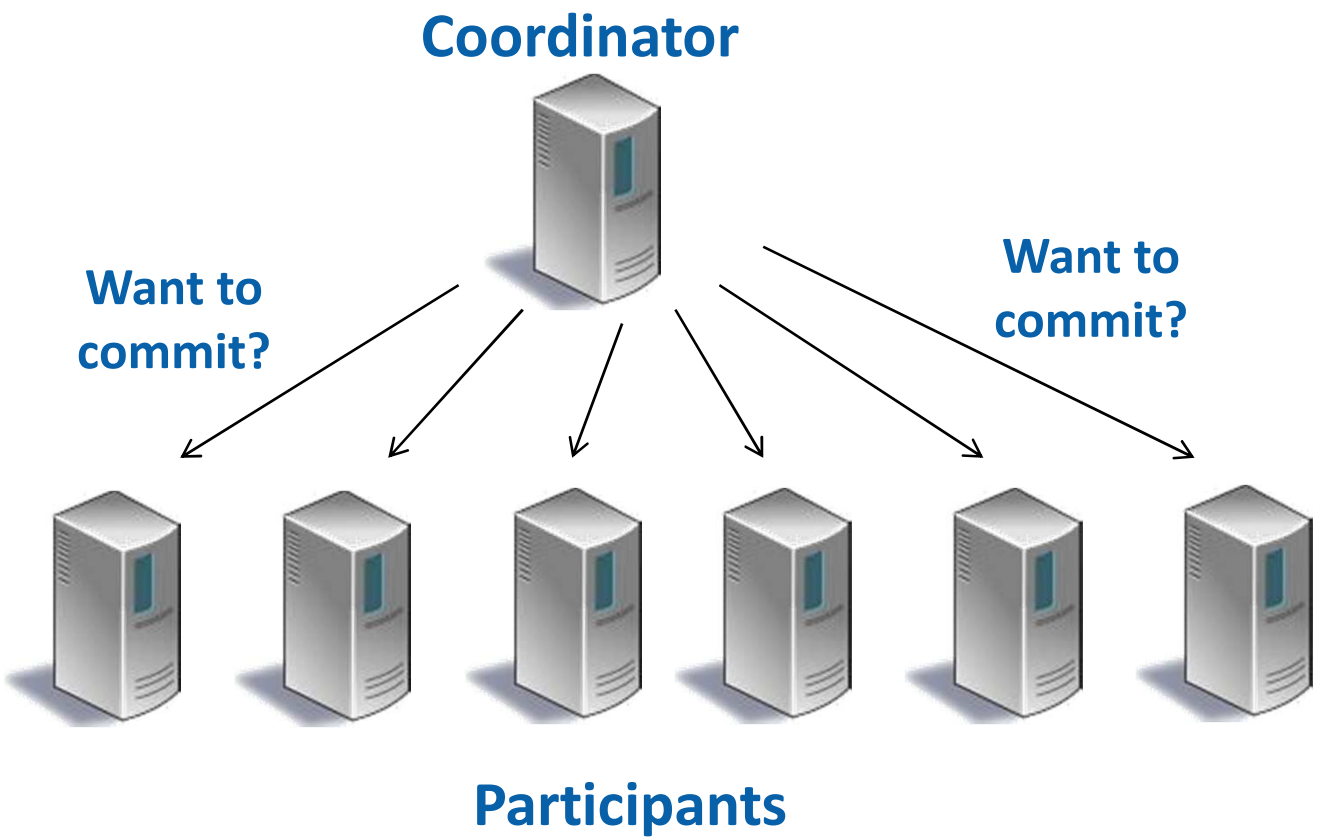
- 如果一个事务读写的数据分布在不同机器上
- 怎么支持?

2 Phase Commit



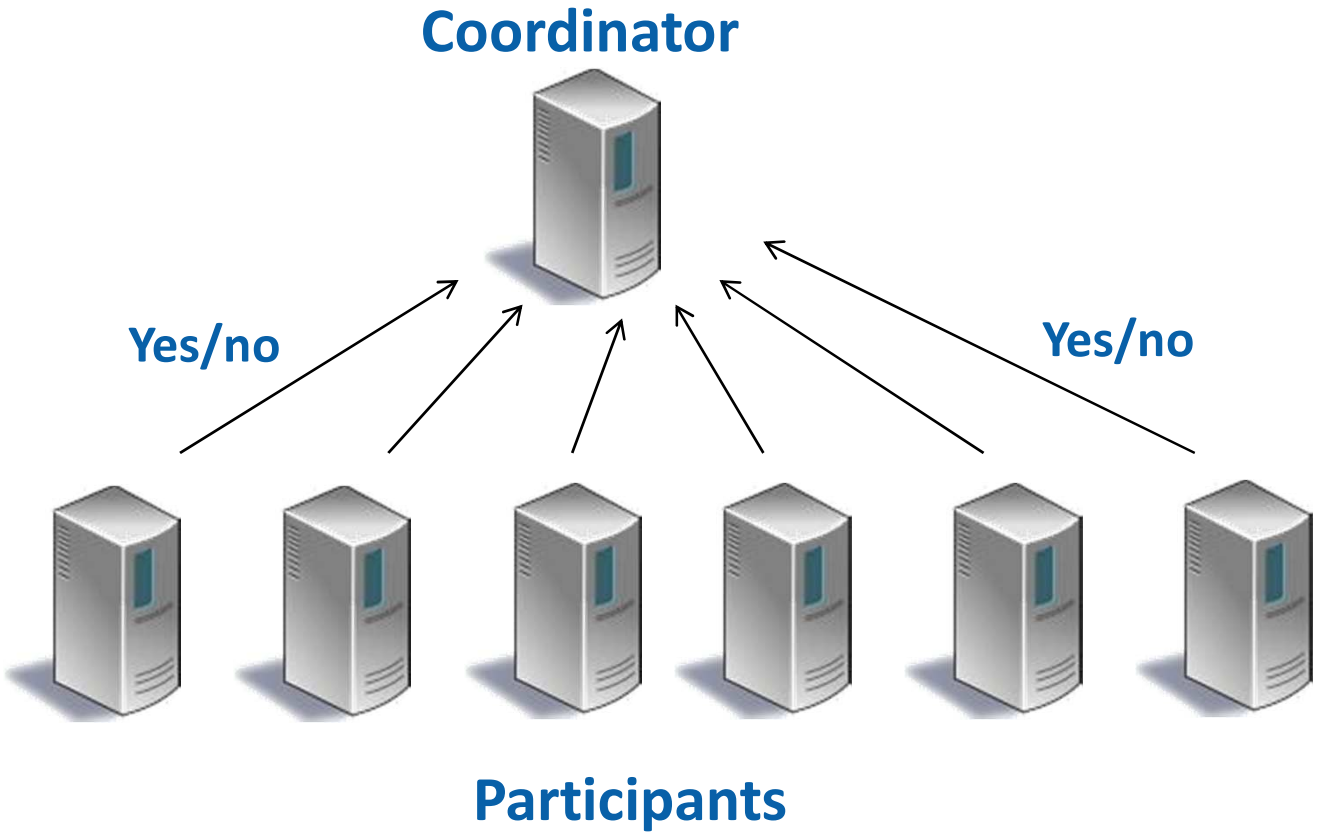
- Participant: 完成分布式事务的部分读写操作
- Coordinator: 协调分布式事务的进行

2 Phase Commit： phase 1 (voting)



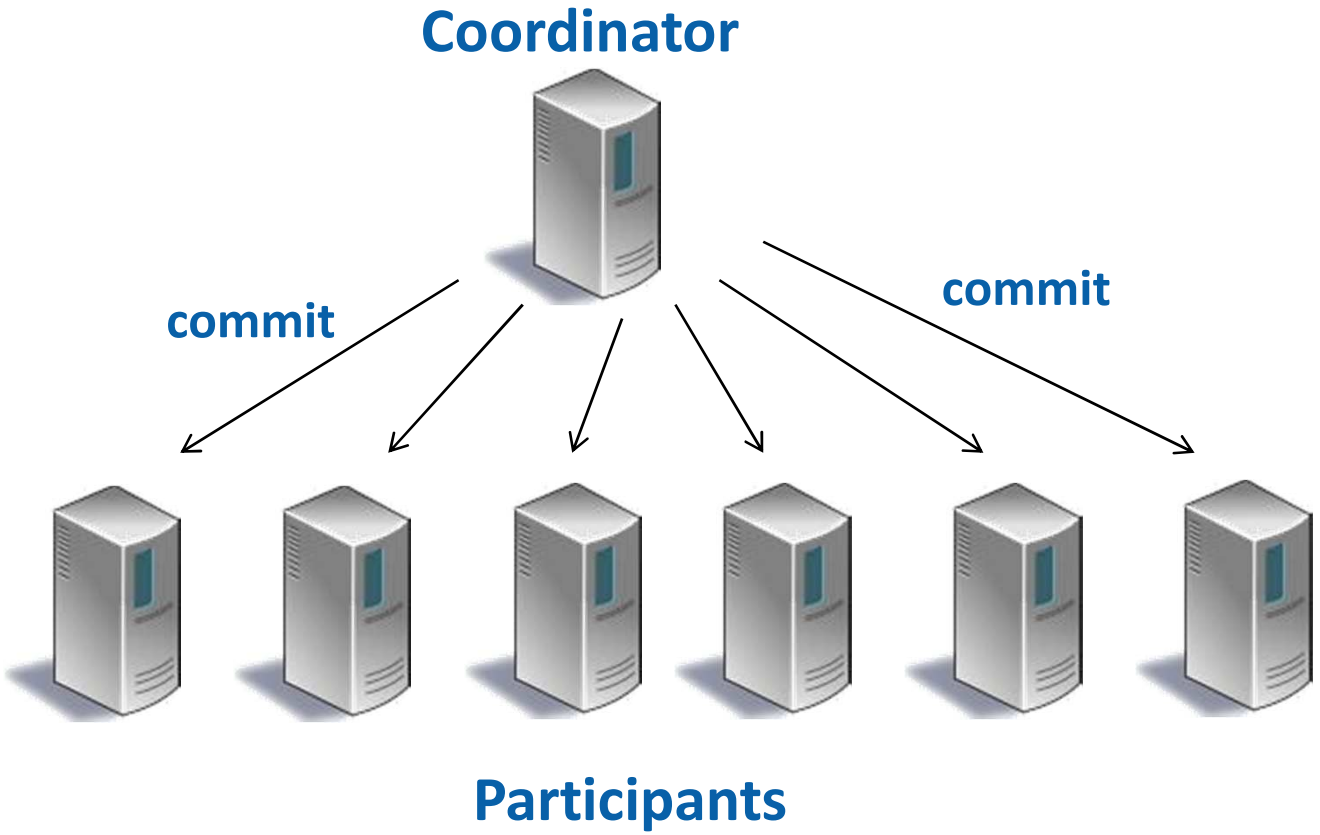
- Coordinator向每个participant发送query to commit消息

2 Phase Commit： phase 1 (voting)



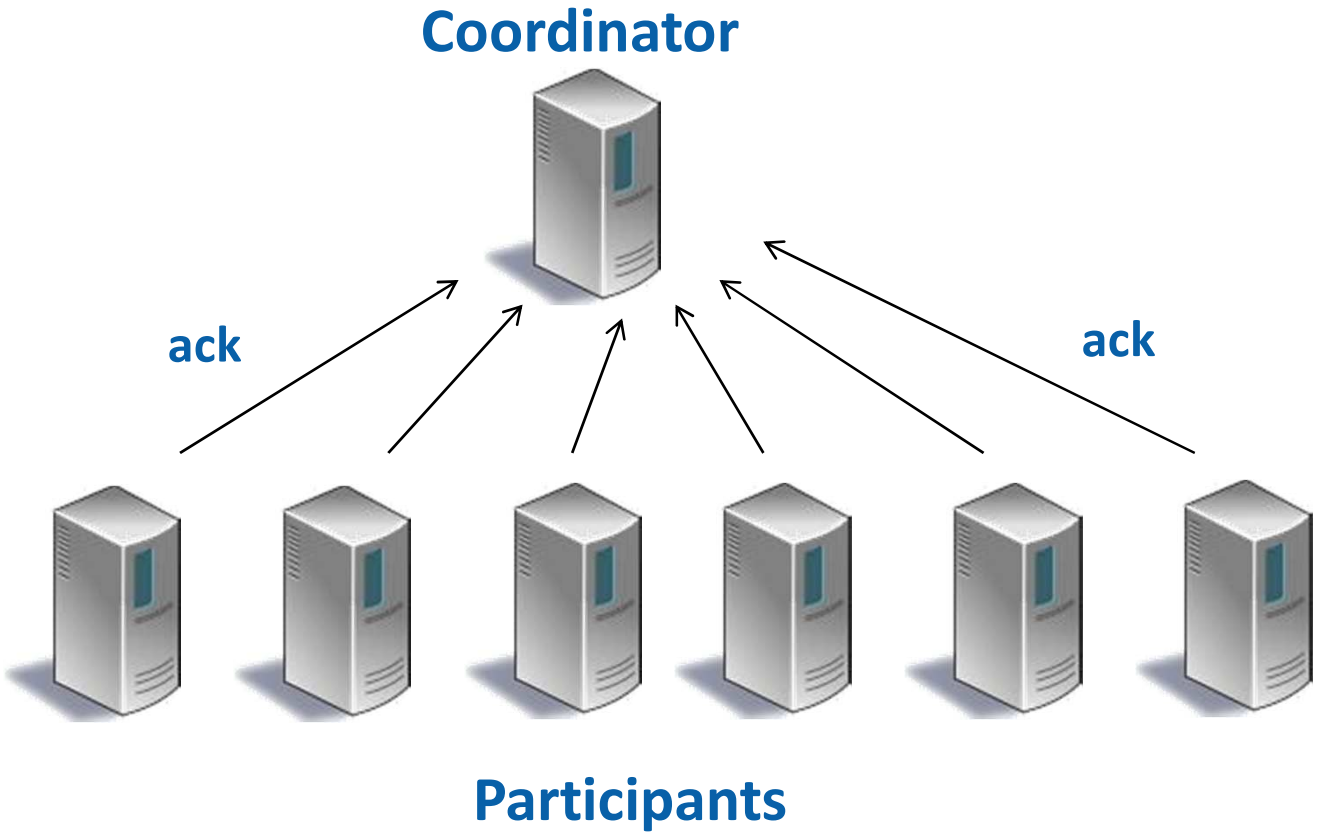
- Coordinator向每个participant发送query to commit消息
- 每个participant根据本地情况回答yes 或 no

2 Phase Commit： phase 2 (completion)



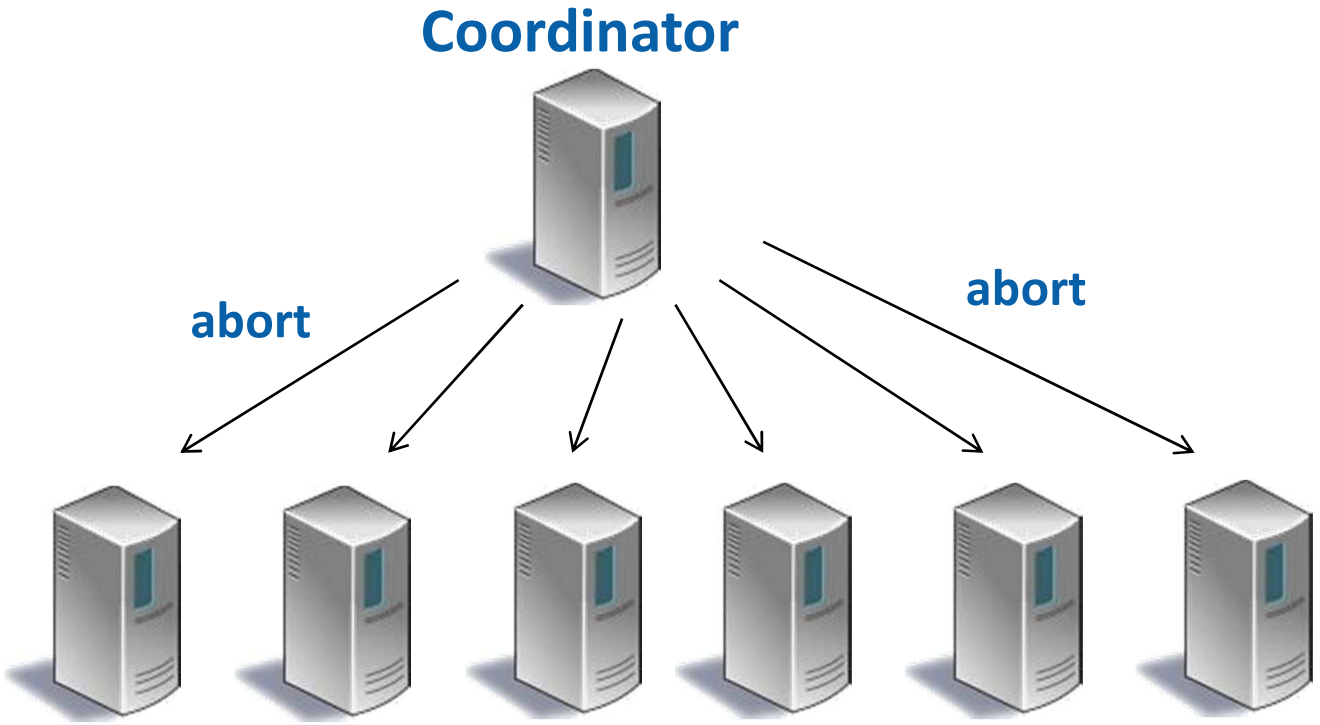
- 当所有的回答都是yes, transaction 将commit
- Coordinator向每个participant发送commit消息

2 Phase Commit： phase 2 (completion)



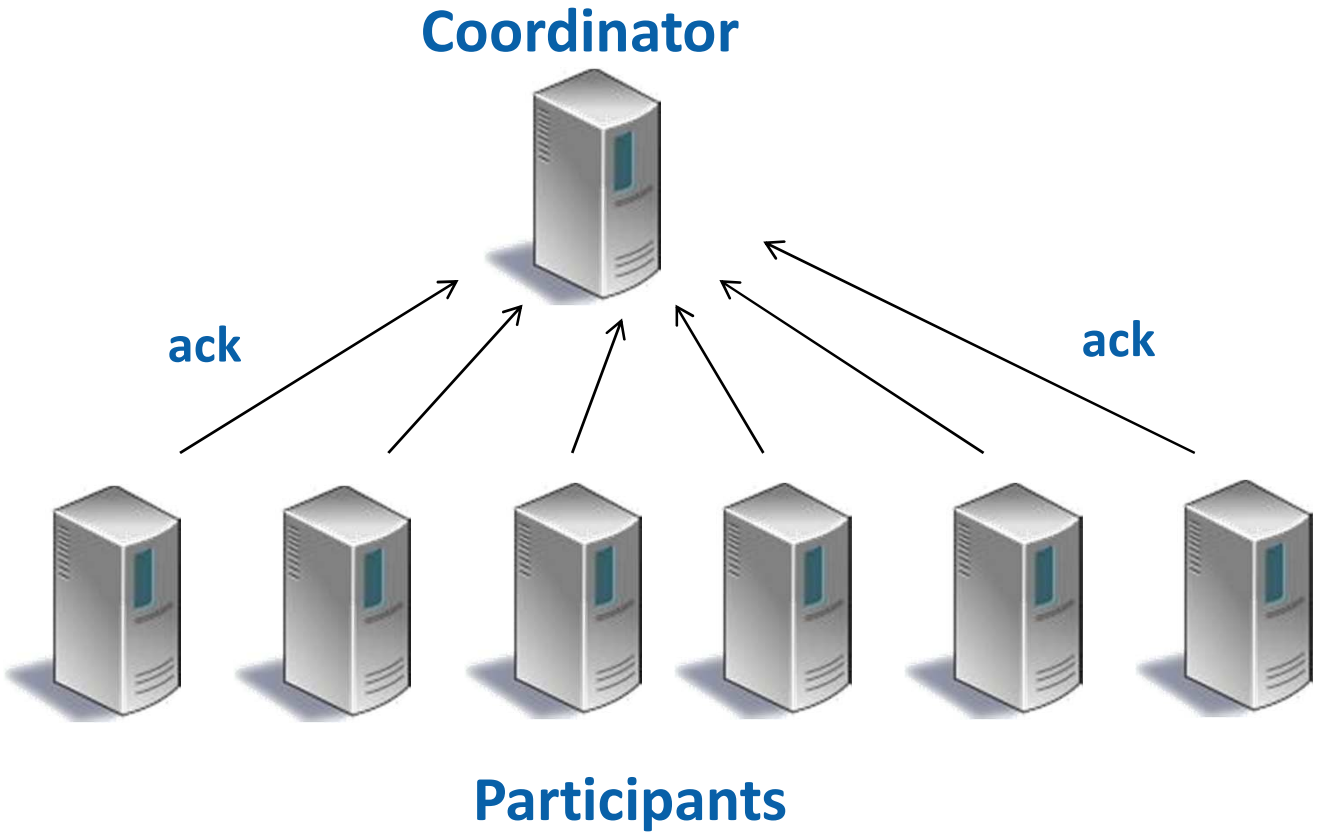
- 当所有的回答都是yes, transaction 将commit
- Coordinator向每个participant发送commit消息
- Participant 回答acknowledgment

2 Phase Commit： phase 2 (completion)



- 当至少一个的回答是no, transaction 将abort
- Coordinator向每个participant发送abort消息

2 Phase Commit： phase 2 (completion)



- 当至少一个的回答是no, transaction 将abort
- Coordinator向每个participant发送abort消息
- Participant 回答acknowledgment

崩溃恢复

- 恢复时日志中可能有下述情况
  - 有commit或abort记录：那么分布式事务处理结果已经收到，进行相应的本地commit或abort
  - 有prepare，而没有commit/abort：那么分布式事务的处理结果未知，需要和prepare记录中的coordinator进行联系
  - 没有prepare/commit/abort：那么本地abort

Outline

- 事务处理
  - ACID
  - Concurrency Control (并发控制)
  - Crash Recovery (崩溃恢复)
- 数据仓库
  - OLAP
  - 行式与列式数据库
- 分布式数据库
  - 系统架构
  - 分布式查询处理
  - 分布式事务处理