

# MySQL 索引与事务

MySQL Index & Transaction

主讲人：西毒老师

系统设计最核心的内容就是数据库设计，没有掌握数据库基础知识点的话就无法更好地进行系统设计。

## 充分体现系统的需求

数据库是为应用服务的，好的数据库设计应该首先能满足应用系统的业务需求，能够准确表达数据间关系。

## 提高数据的查询效率

通过合理表结构，安排物理存储分区、增加**索引**等方式、提高数据库的读取速度、提高查询效率



## 能够保证数据的准确性和一致性

通过外键、非空、限制、**唯一索引**、**事务**等保证数据的健壮

## 有好的扩展性

好的数据库设计在必要时应根据需求进行数据结构扩展

# 为什么要了解数据库索引和事务？

在面试的过程中，面试官可能会直接问数据库的问题



B+Tree 的原理是什么？



InnoDB 和 MyISAM 的区别？



什么是事务的隔离级别？

如果不理解基础知识，就很难理解后面的实战设计题。

# 01 索引 Index

索引是一种特殊的数据库结构，由数据表中的一列或多列组成，可以用来快速查询数据表中有某一特定值的记录。



## 生活中的索引

目 录	
推荐序	
前言	
第 1 章 MySQL 体系结构和存储引擎	1
1.1 定义数据库和实例	1
1.2 MySQL 体系结构	3
1.3 MySQL 存储引擎	5
1.3.1 InnoDB 存储引擎	6
1.3.2 MyISAM 存储引擎	7
1.3.3 NDB 存储引擎	7
1.3.4 Memory 存储引擎	8
1.3.5 Archive 存储引擎	9
1.3.6 Federated 存储引擎	9
1.3.7 Maria 存储引擎	9
1.3.8 其他存储引擎	9
1.4 各存储引擎之间的比较	10
1.5 连接 MySQL	13
1.5.1 TCP/IP	13
1.5.2 命名管道和共享内存	15
1.5.3 UNIX 域套接字	15
1.6 小结	15
第 2 章 InnoDB 存储引擎	17
2.1 InnoDB 存储引擎概述	17
2.2 InnoDB 存储引擎的版本	18
2.3 InnoDB 体系架构	19
2.3.1 后台线程	19
2.3.2 内存	22
2.4 Checkpoint 技术	32
2.5 Master Thread 工作方式	36
2.5.1 InnoDB 1.0.x 版本之前的 Master Thread	36
2.5.2 InnoDB 1.2.x 版本之前的 Master Thread	41
2.5.3 InnoDB 1.2.x 版本的 Master Thread	45
2.6 InnoDB 关键特性	45
2.6.1 插入缓冲	46
2.6.2 两次写	53
2.6.3 自适应哈希索引	55
2.6.4 异步 IO	57
2.6.5 刷新邻接页	58
2.7 启动、关闭与恢复	58
2.8 小结	61
第 3 章 文件	62
3.1 参数文件	62
3.1.1 什么是参数	63
3.1.2 参数类型	64
3.2 日志文件	65
3.2.1 错误日志	66
3.2.2 慢查询日志	67
3.2.3 查询日志	72
3.2.4 二进制日志	73
3.3 套接字文件	83
3.4 pid 文件	83
3.5 表结构定义文件	84
3.6 InnoDB 存储引擎文件	84
3.6.1 表空间文件	85
3.6.2 重做日志文件	86
3.7 小结	90
第 4 章 表	91
4.1 索引组织表	91

## 目录的特点

01

单独的结构

02

指向了具体内容的位置

03

加速了对书籍内容的查找

04

有些东西还是没法放到目录里面

05

书变厚了。。。

根据考号查询考试成绩?

```
SELECT * FROM `exam_grade` WHERE `id`=104003
```

高考成绩

考号	姓名	性别	学校	成绩
100704	小王	男	一中	600
103809	小明	男	一中	601
104003	小红	女	二中	610
100001	小张	女	二中	590
...				

01

方法一


从头至尾，一条一条地对比，直到找到对应的考号

 全国数百万考生，如何查找?

02

方法二

如果已经根据[考号]排好序了，在一个有序的列表中，二分地查找。

 对其他需要查找的字段能不能也排个序?

索引 (Index) 是帮助MySQL高效获取数据的数据结构。

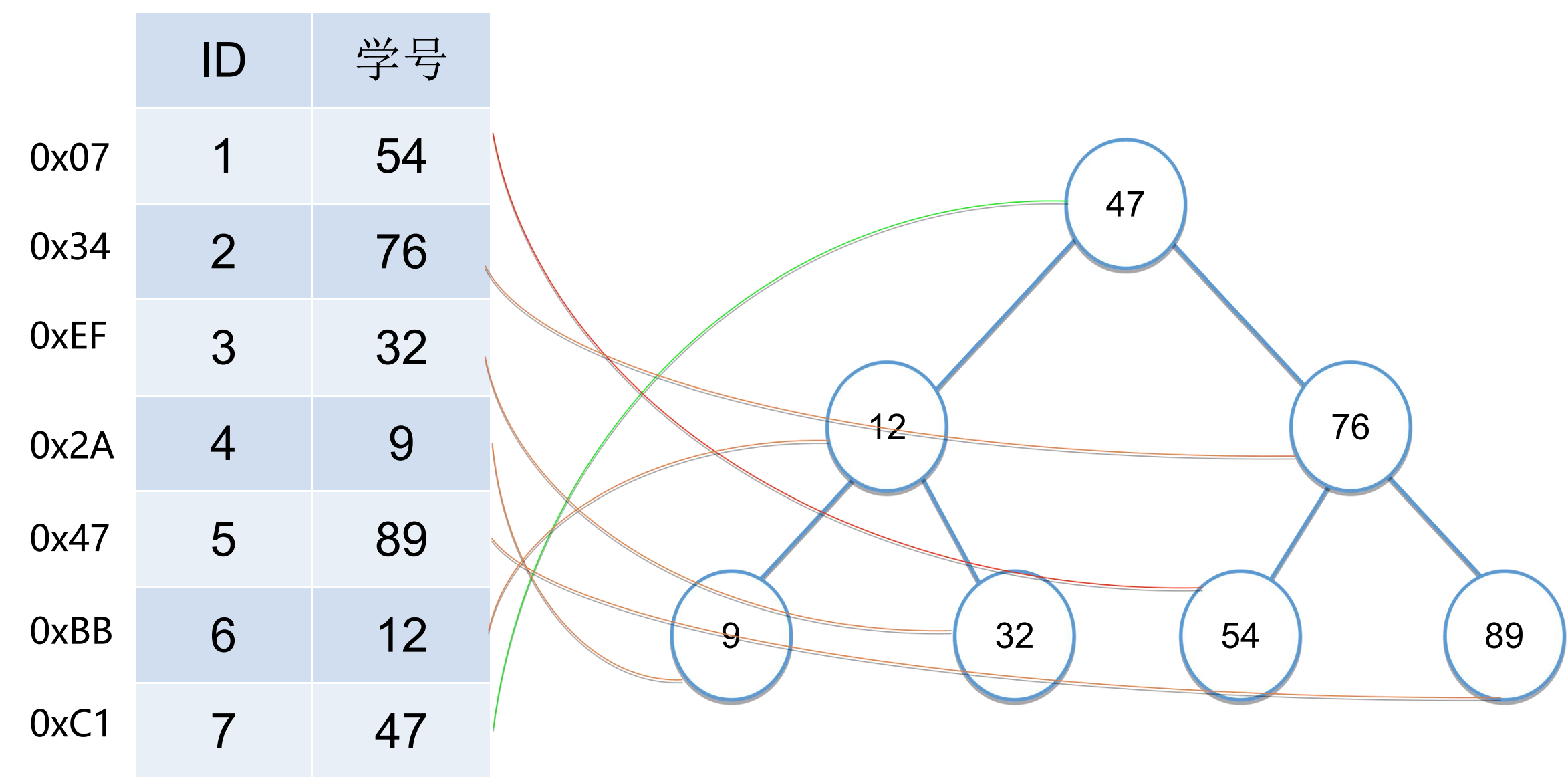
数据库查询是数据库的最主要功能之一



各种查找算法都有局限。数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式指向数据，这样就可以在这些数据结构上实现高级查找算法。



## 一种可能的索引方式



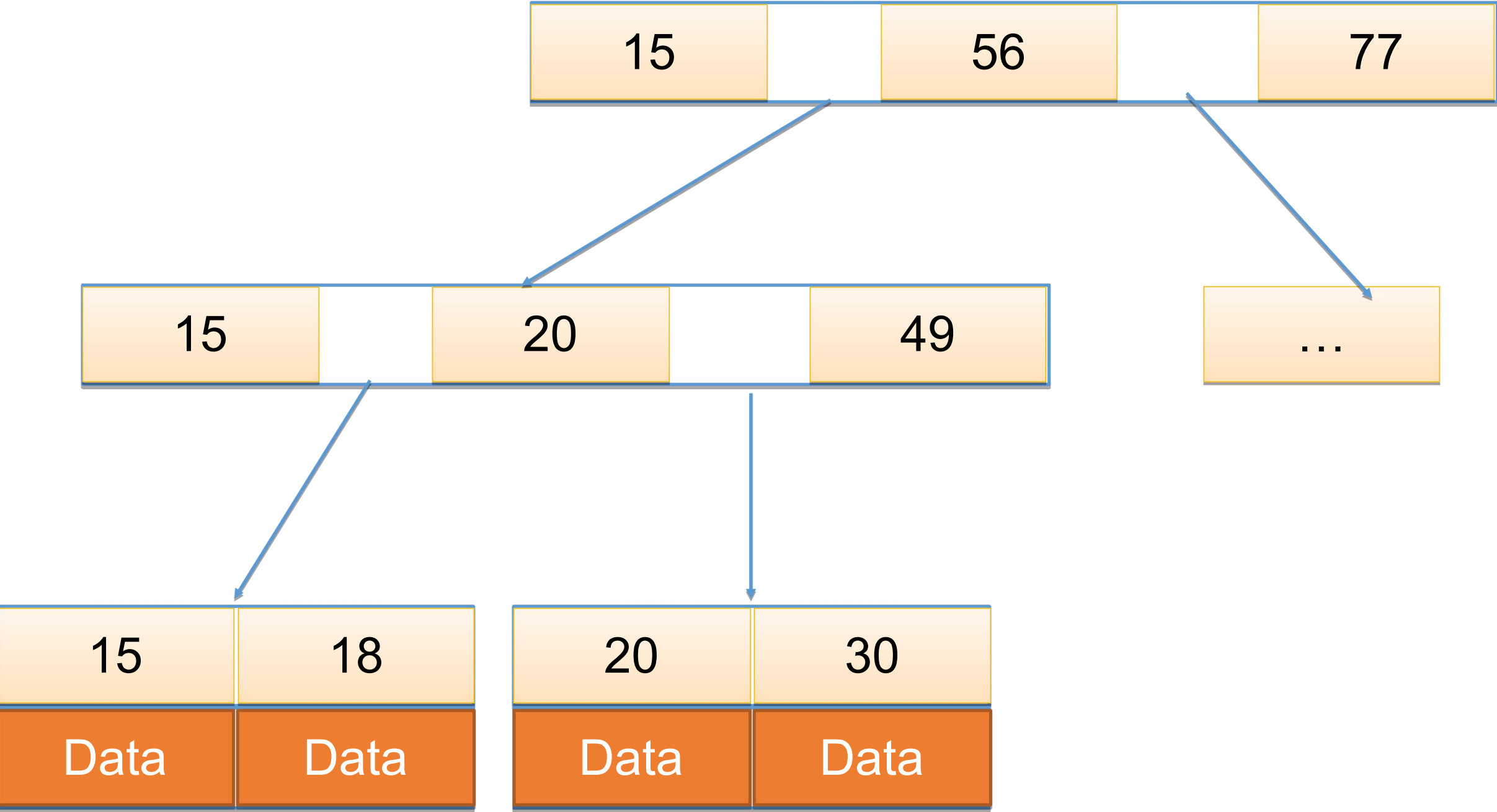
为了加快对「学号」的查找，可以维护二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针 (Pointer) 。这样复杂度为  $O(\log_2 n)$



索引以**索引文件**的形式存储在**磁盘**上。

这样导致索引查找过程中就要产生磁盘I/O消耗。相对于内存存取，I/O存取的消耗要高几个数量级。

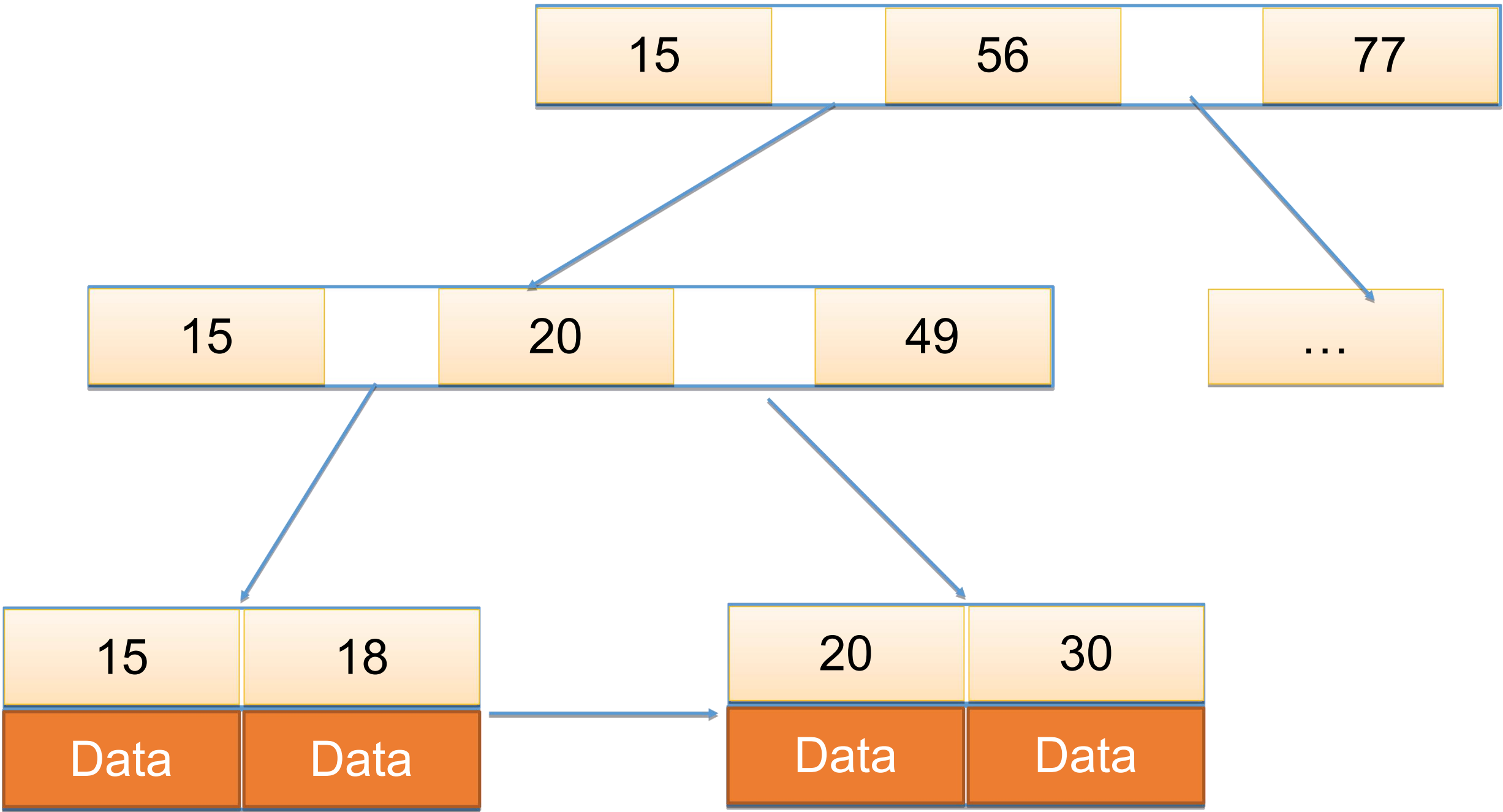
评价一个数据结构作为索引的优劣最重要的指标就是在**查找过程中磁盘I/O操作的次数**。  
换句话说，索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。



一个M阶的B+树具有如下几个特征——

- 1. 根结点至少有两个子女。
- 2. 每个中间节点都至少包含 $\text{ceil}(m / 2)$ 个孩子，最多有m个孩子。
- 3. 所有的叶子结点都位于同一层。
- 4. 每个节点中的元素从小到大排列。
- 5. 内结点不存储数据，只存储键值；叶子结点不存储指针。

数据库系统B+Tree的基础上进行了优化，增加了顺序访问指针。



提高区间访问的性能——

🧑🏻‍🔍 如果要查找 18 到 30 之间的数据，如何查找？

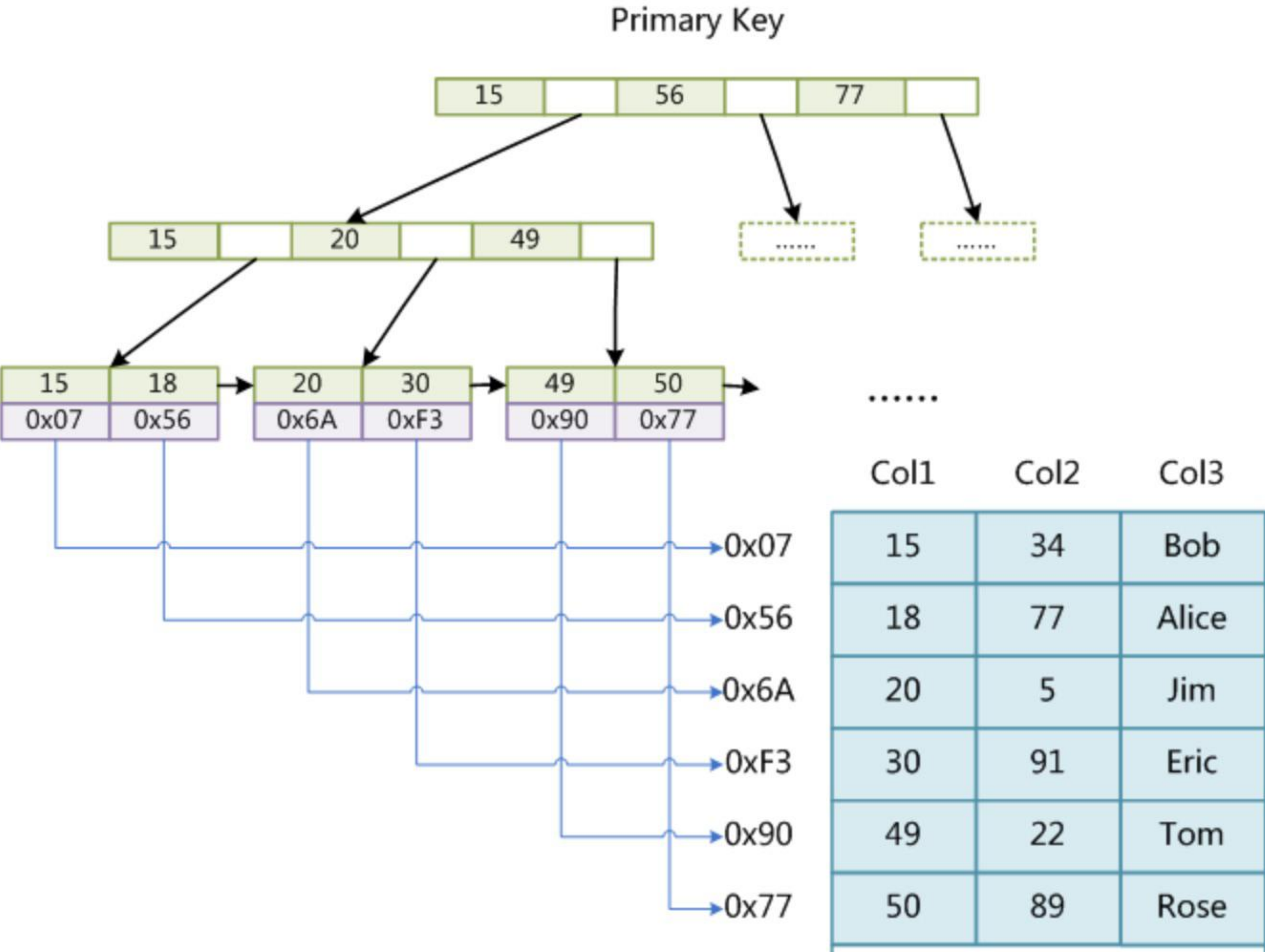
在MySQL中，索引属于存储引擎(Storage Engine)级别的概念，不同存储引擎对索引的实现方式是不同的



MyISAM



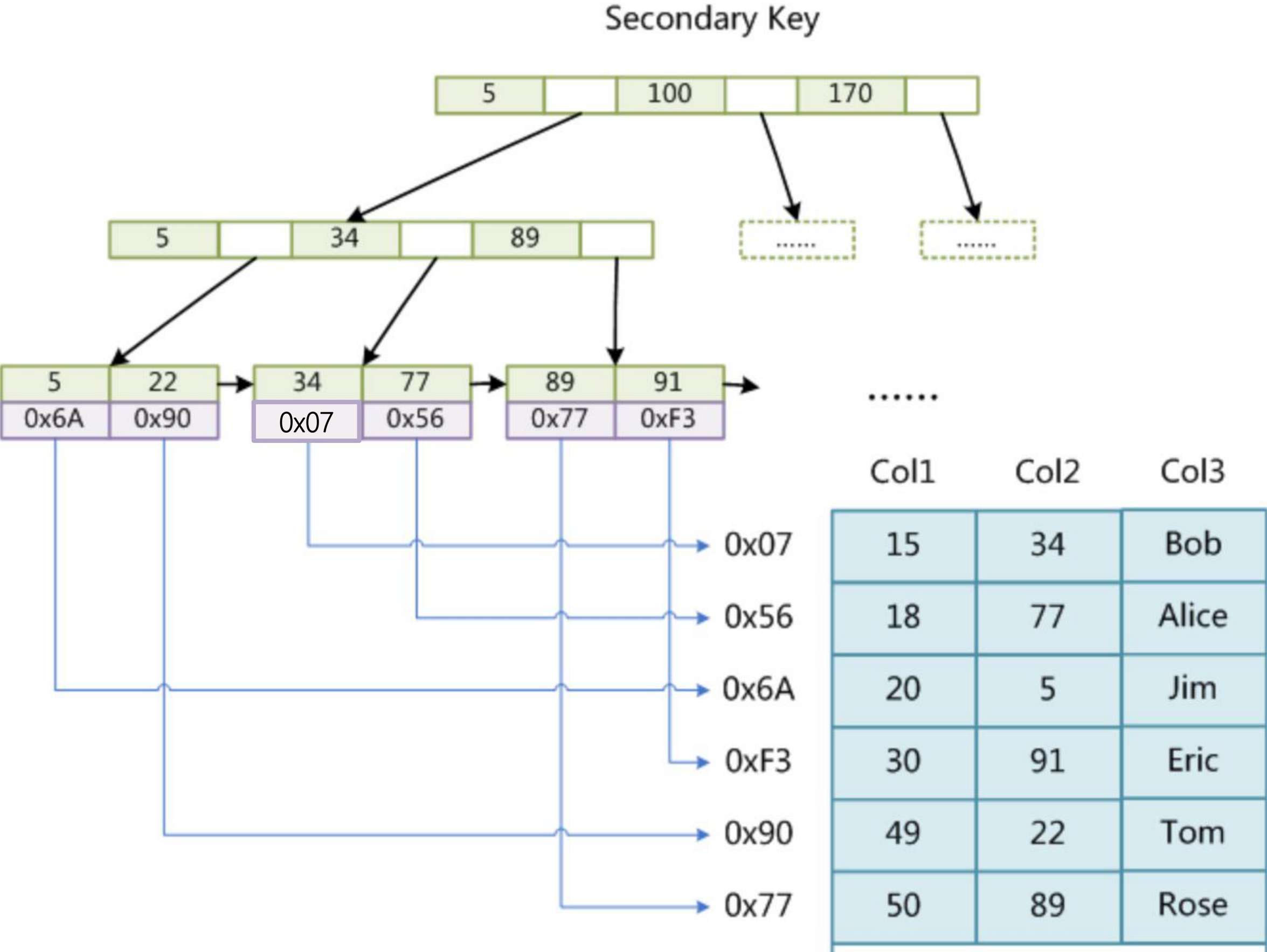
InnoDB



MyISAM引擎使用B+树作为索引结构，叶结点的数据域存放的是实际数据记录的地址。

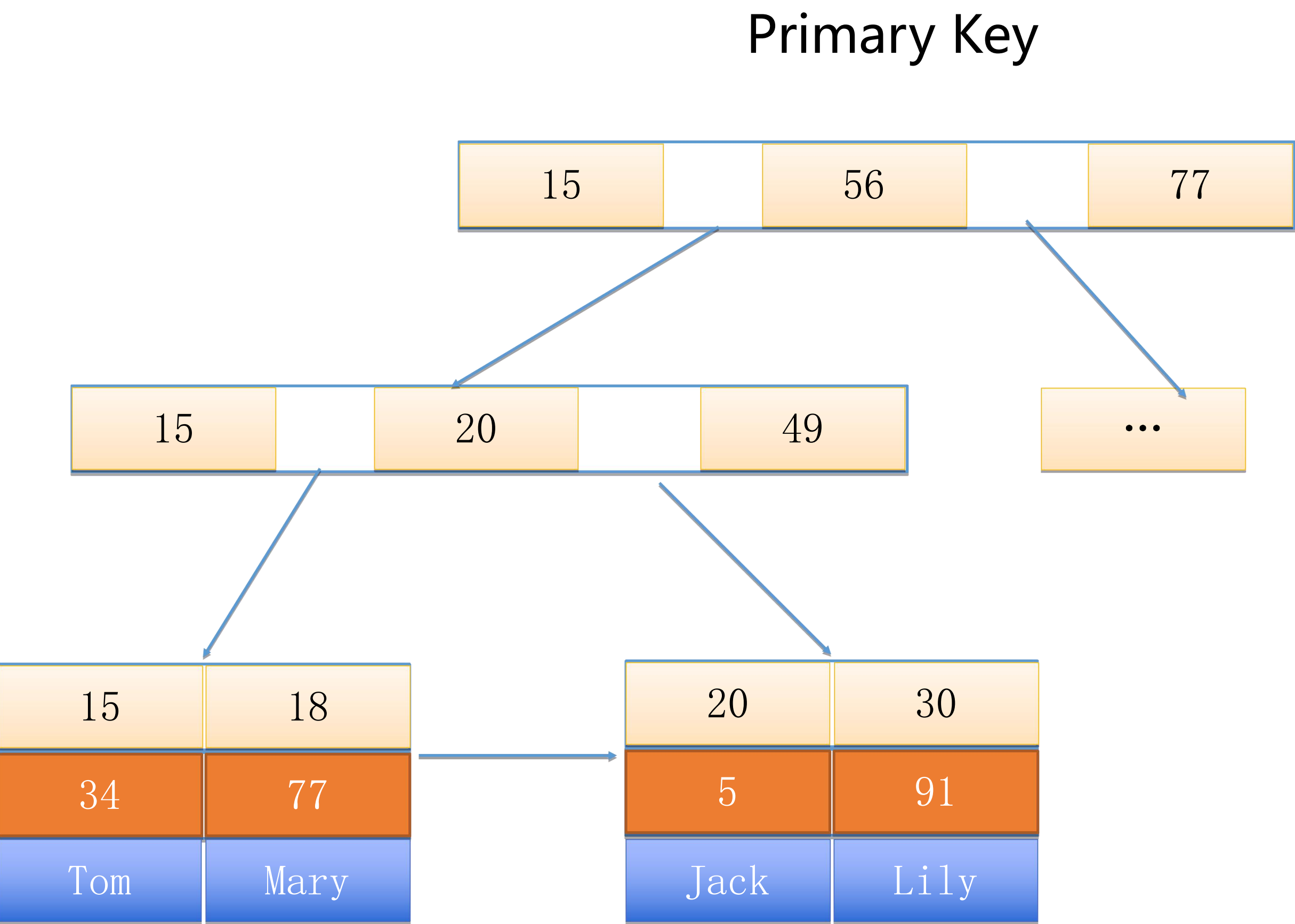
在MyISAM中，主键索引(Primary Key)和辅助索引(Secondary key)在结构上没有差别。只是主键索引要求唯一，而辅助索引可以重复。





辅助索引(Secondary key)同样也是一棵B+树，数据域(Data Field)保存数据记录的地址。

1. 按照B+树搜索算法搜索索引
2. 如果指定的键存在，则取出其数据域的值
3. 以数据域的值作为地址，读取相应数据记录。



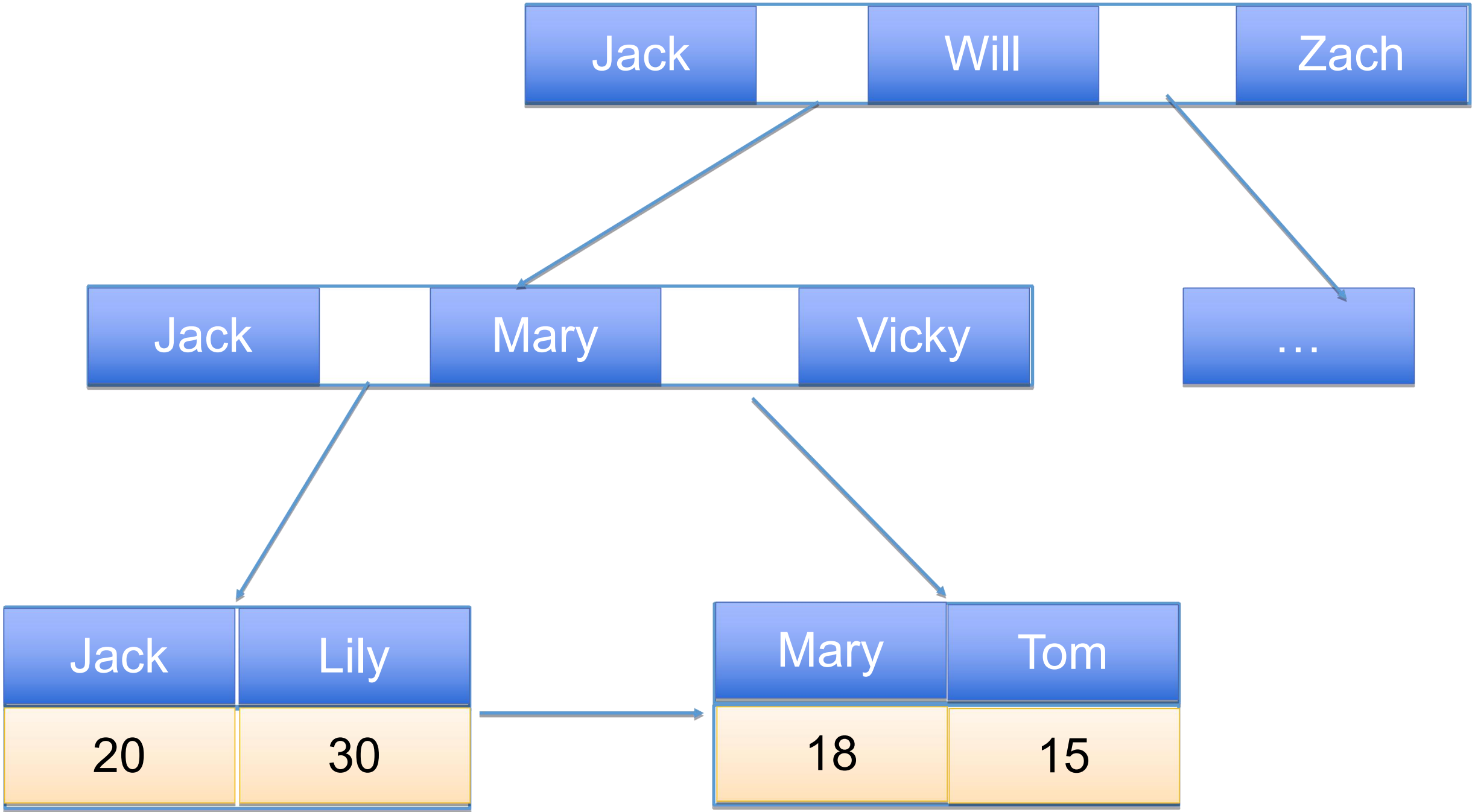
InnoDB的数据文件本身就是索引文件。  
树的叶结点数据域(Data field)保存了完整的数据记录。这个索引的键是数据表的主键。

InnoDB 表数据文件本身就是主索引。

InnoDB要求表必须有主键(MyISAM可以没有)。  
如果没有指定主键，怎么办？

1. InnoDB自动选择一个可以唯一标识数据记录的列作为主键
2. 自动生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

## Secondary Key



InnoDB 的辅助索引(Secondary key)数据域存储相应记录主键的值而不是地址

## InnoDB

优势——

主键索引(Primary Key)查询  
高效，直接拿到数据

劣势——

辅助索引(Secondary key)  
需要查两次 B+ 树

1. 查辅助索引拿到主键值
2. 查主键索引拿到数据

## MyISAM

优势——

辅助索引和主键索引效率相同

劣势——

主键索引查到后，需要根据地址再访问一次磁盘

MyISAM不支持  
**事务操作!!!**

# 联合索引(Composite Index)和多列单索引

联合索引——对两列或多列数据创建共同的索引。

如「电话簿」——先按照「姓」排序，在同一个姓下，按照「名」排序，则是「姓」和「名」的联合索引。

## 联合索引特点

- 多个列的联合索引是**有顺序**的。
- 对索引中**所有列**查询或**前几列**查询，索引有用。
- 对索引中后几列查询，索引没用。

## 多列单索引特点

- 多条件查询时，能自动优化，可能都用上，不用关心顺序。
- 会建立多个 B+ 树，占用空间。



不允许有索引值相同的行，如「学号」。

- 1.在建立索引时检查表中是否有相同索引值。
- 2.在插入、更新数据时，检查是否重复。

什么是联合唯一索引  
(Composite Unique Index)?

不要使用过长字段作主键，如很长的字符串(这边最好能举个例子)。

辅助索引(Secondary Key)引用了主键，如果主键太长，那么辅助索引会变得很大。

不要使用非单调(Non-monotonic)的字段作为主键，如随机的字符串、随机的数字。

非单调的主键使得在插入新数据时，数据库文件为了维持B+树的特性而频繁的分裂调整

单调递增(monotonic increasing)的整数是个好选择，  
最好和业务无关。

既然索引能提高查询效率，那么  
直接给所有列加上索引可好？

索引的代价——

1. 索引文件本身消耗空间
2. 插入、更新、删除数据时，维护索引结构，添加负担
3. MySQL 自身运行时也要消耗资源维持索引

## 哪些列不要建索引——

1. 不太会查询到的列
2. 本身没有顺序，或者特别长的列。如，「地址」。
3. 表记录比较少，就不用建索引了，直接全表扫描。  
如，「班级同学」表。
4. 选择性低，即重复率高，可枚举的(Enumerable)列。  
如「性别」。

### 选择性——

$$\text{Selectivity} = \text{Cardinality} / T$$

Cardinality : 不重复的索引值数量

T: 表中数据条数

Selectivity 约趋近于 1, 索引价值越大

# 开放式实践题

设计一张表，包含今年全国的高考考生(假设大家都做一套卷子)以及他们的成绩。

- 1.姓
- 2.名
- 3.身份证号
- 4.性别
- 5.省份
- 6.总分
- 7.语文成绩
- 8.数学成绩
- 9.英语成绩
- 10.综合科成绩

用什么做主键  
(Primary Key)?

哪些列建立单索引  
(Single Index)?

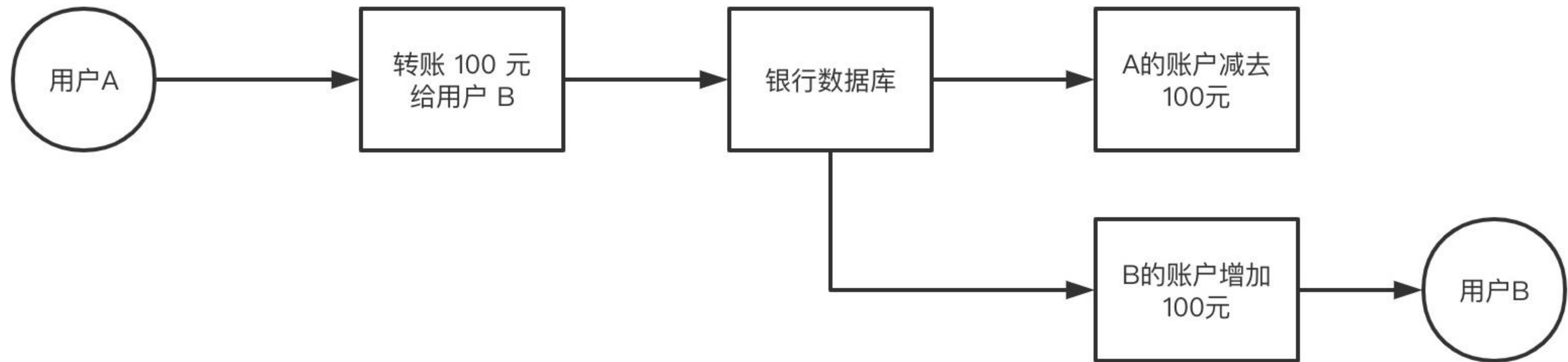
哪些列建立联合索引  
(Composite Index)?

哪些列建立唯一索引  
(Unique Index)?



## 02 事务 Transaction

当多个用户访问同一数据时，一个用户在更改数据的过程中可能有其它用户同时发起更改请求，为保证数据的一致性状态，MySQL 引入了事务。



## 转账过程中可能出现的问题

1. 第一步执行成功，A 账户上的钱减少了 100 元。但是第二步执行失败或者系统崩溃，导致 B 账户并没有相应增加 100 元。
2. 转账操作刚完成就数据库就崩溃，重启恢复后，丢失了之前的转账记录。
3. A 在银行 ATM 机取钱的瞬间，立刻给 B 用微信转账。

数据库事务是构成**单一逻辑工作单元(Single logical unit of work)**的操作集合

```
BEGIN TRANSACTION
SQL1
SQL2
...
COMMIT/ROLLBACK TRANSACTION
```

```
BEGIN TRANSACTION
UPDATE account SET balance = balance - 100
WHERE user = A;
UPDATE account SET balance = balance + 100
WHERE user = B;
COMMIT/ROLLBACK TRANSACTION
```

1. 事务可以包含一个或多个对数据库的操作，这些操作构成一个**逻辑上的整体**。
2. 这些数据库操作，要么全部执行成功，要么全部不执行。
3. 这些数据库操作，要么全都对数据库产生影响，要么全都不产生影响。即不管事务是否执行成功，数据库总能保持一致性(consistency)状态。
4. 以上即使在数据库出现故障或者并发(Concurrent)事务存在的情况下依然成立。

## 原子性(Atomicity)

所有操作作为一个整体不可分割，要么全部成功，要么全部失败。

## 一致性(Consistency)

必须使数据库从一个一致性状态到另一个一致性状态。  
比如转账前后两个账户的金额总和应该保持不变。

## 隔离性(Isolation)

并发执行的事务不会相互影响，对数据库的影响就跟它们串行执行时一样。

## 持久性(Durability)

事务一旦提交，其对数据库的变更就是持久的。  
任何事务或数据库故障都不会导致数据丢失。

一致性是事务的根本追求  
而对数据一致性的破坏主要来自两个方面

1. 事务的并发执行
2. 事务故障或系统故障

# 事务的特性 - 一致性保障

## 并发控制(Concurrency Control)

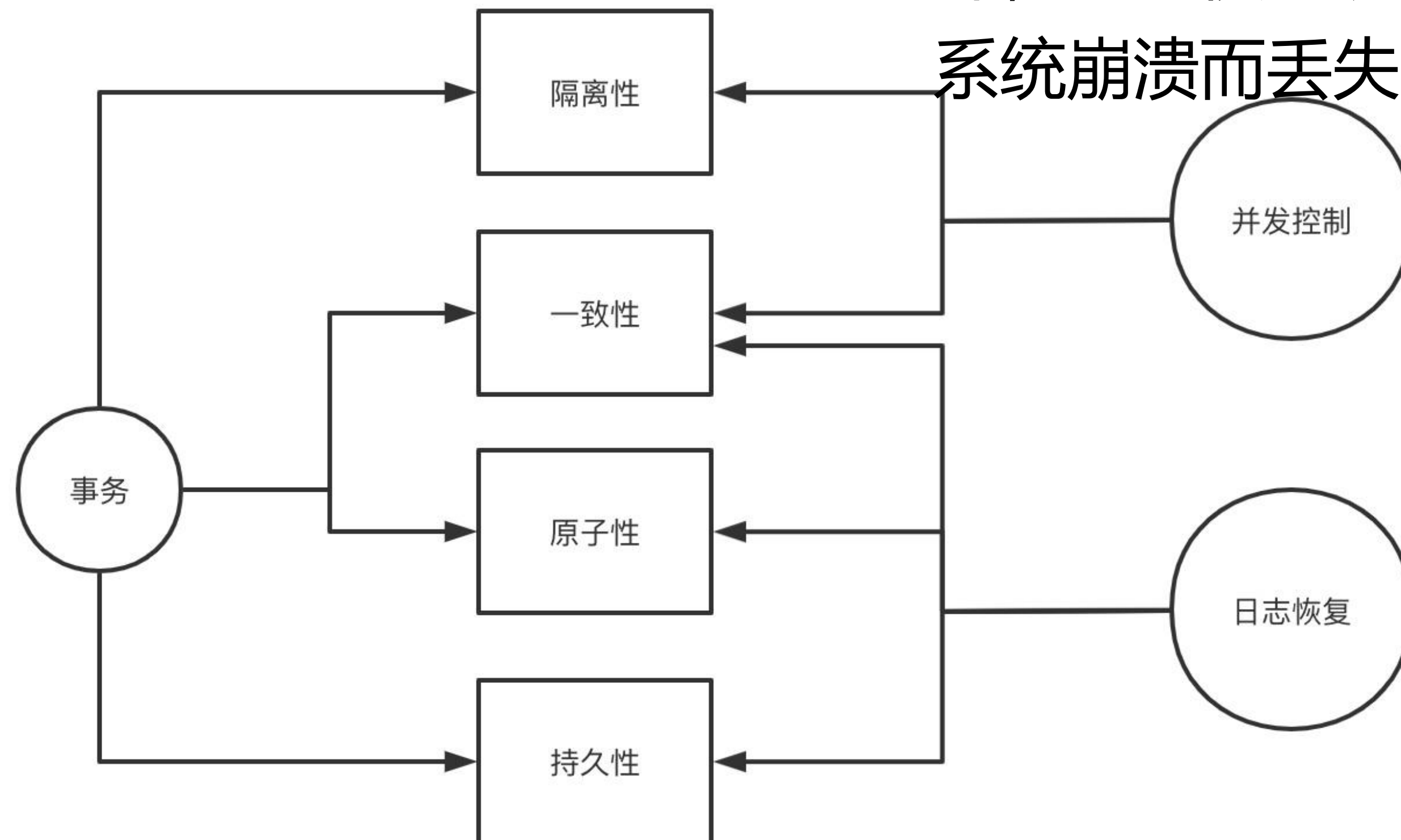
保证了事务的**隔离性(Isolation)**。

使一致性(consistency)状态不会因为并发执行的操作被破坏。

## 日志恢复(Log Recovery)

保证了事务的**原子性(Atomicity)**和**持久性(Durability)**。

使一致性状态不会因事务或系统故障被破坏，同时使已提交的对数据库的修改不会因系统崩溃而丢失。





# 事务并发控制

## 隔离性(Isolation) 和 一致性 (consistency) 保障

当多个进程(Process)或线程(Thread)并发地操作数据库时，有 3 种冲突类型：

读 - 读：不会引发任何并发(Concurrent)问题。

读 - 写：有隔离性(Isolation)的问题，可能读到不正确的数据。

写 - 写：有隔离性的问题，可能丢失掉已经更新的内容。

# 事务的隔离性和隔离级别(Isolation Levels)

## 隔离性(Isolation)

理论上来说事务之间的执行不应该相互产生影响，其对数据库的影响应该和它们串行执行时一样。

## 引发(Concurrent)的问题

完全的隔离性会导致系统并发性能低下，降低了资源利用率。所以实际上对隔离性的要求会有所放宽。这也会造成对数据库一致性(consistency)要求降低。

事务的隔离级别越低，可能出现的并发问题就越多，而系统能的并发能力一般就会就越强。

1. 读未提交 Read Uncommitted

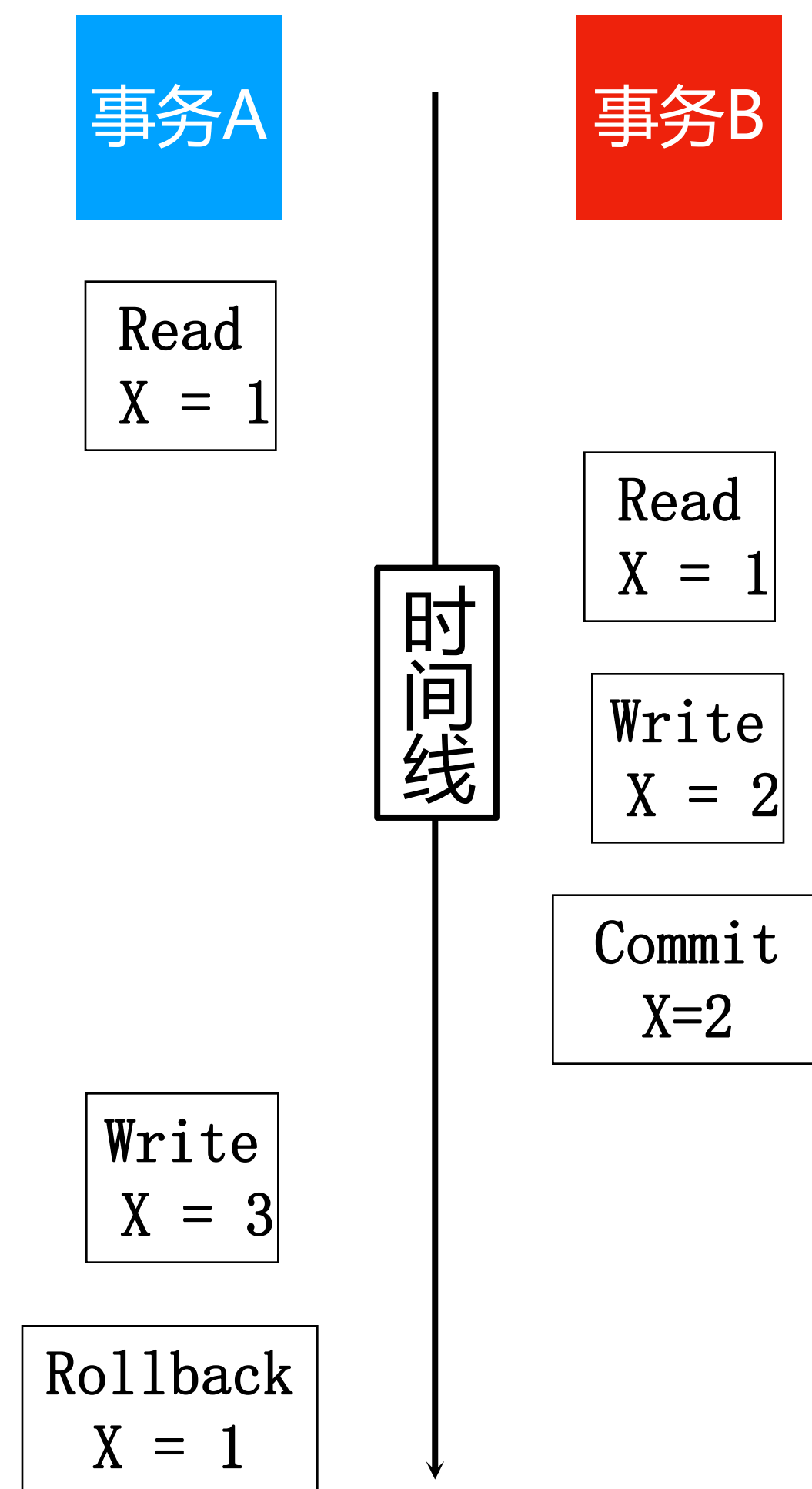
2. 读已提交 Read Committed

3. 可重复读 Repeatable Read

4. 串行化 Serializable

# 并发问题 - 脏写(Dirty Writes)问题

指事务回滚(RollBack)了其他事务对数据项已提交的变更。

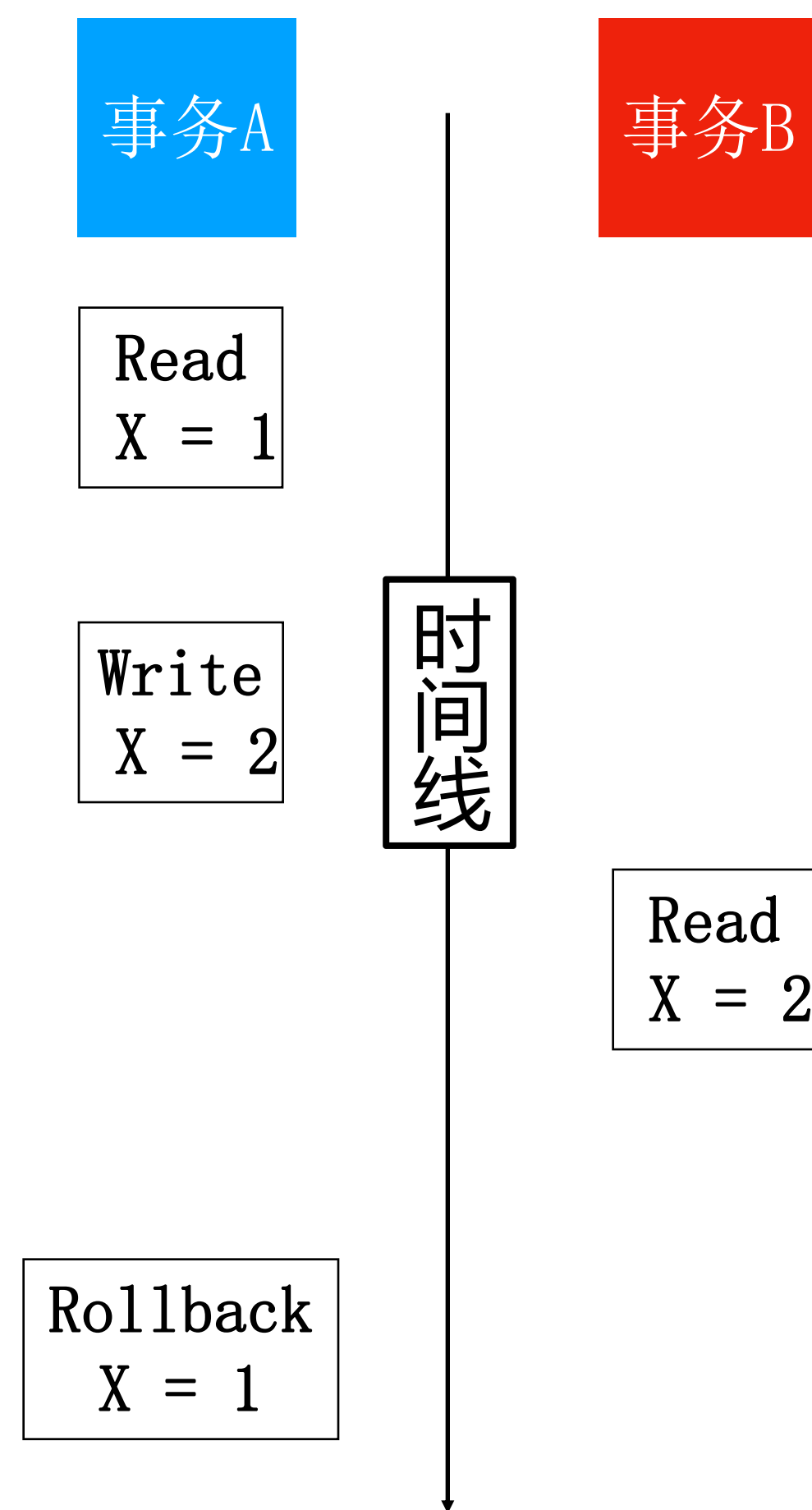


事务A 对数据  $X$  的回滚,导致事务B对  $X$  的已提交变更也被回滚了。

所有事务隔离级别都不允许出现脏写。

# 并发问题 - 脏读(Dirty Reads)问题

指一个事务读取了另一个事务**未提交的数据**。



在事务A对  $X$  的处理过程中，事务B读取了  $X$  的值。

但之后事务A回滚，导致事务B读取的  $X$  是未提交的脏数据。

「**读未提交**」隔离级别，会导致「脏读」。  
避免脏读，需要将隔离级别提升至「**读已提交**」（读其他事务已经提交过的数据）。



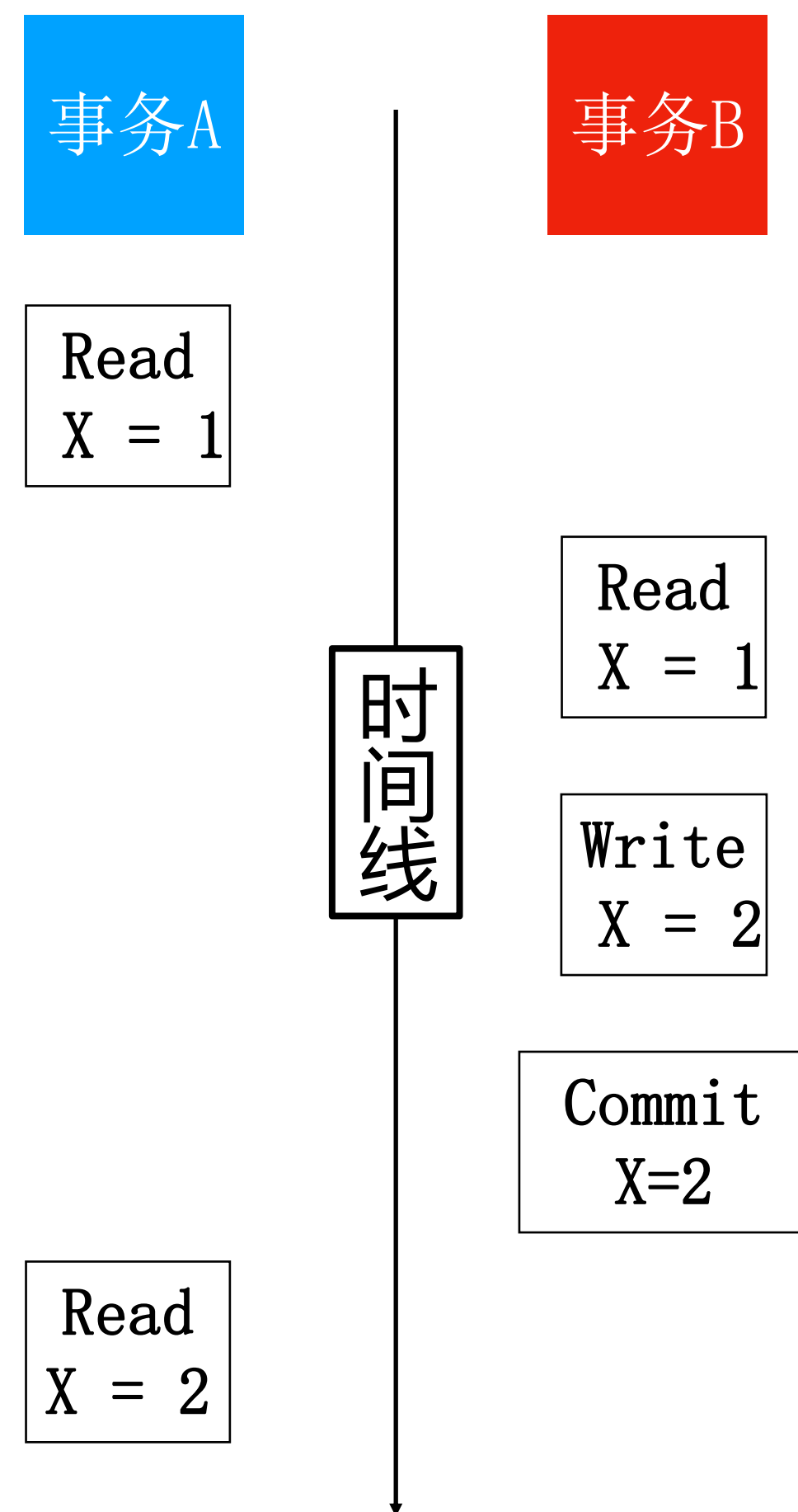
# 并发问题 - 不可重复读(Non-repeatable Reads)问题

指一个事务对**同一数据**的读取结果前后不一致。

脏读(Dirty Reads)和不可重复读的  
区别在于——

**脏读**读取的是事务**未提交的脏数据**。

**不可重复读**读取的是事务**已提交的数据**。只不过因为数据被其他事务修改过导致前后读取的结果不一样。



由于事务B对 X 的已提交修改，  
导致事务A前后两次读取的结果  
不一致。

「**读已提交**」隔离级别，会导致  
「不可重复读」。

避免不可重复读，需要将隔离级  
别提升至「**可重复读**」（有事务  
读了某条数据后，其他事务不能  
去写该数据）。

# 并发问题 - 幻读(Phantom Reads)问题

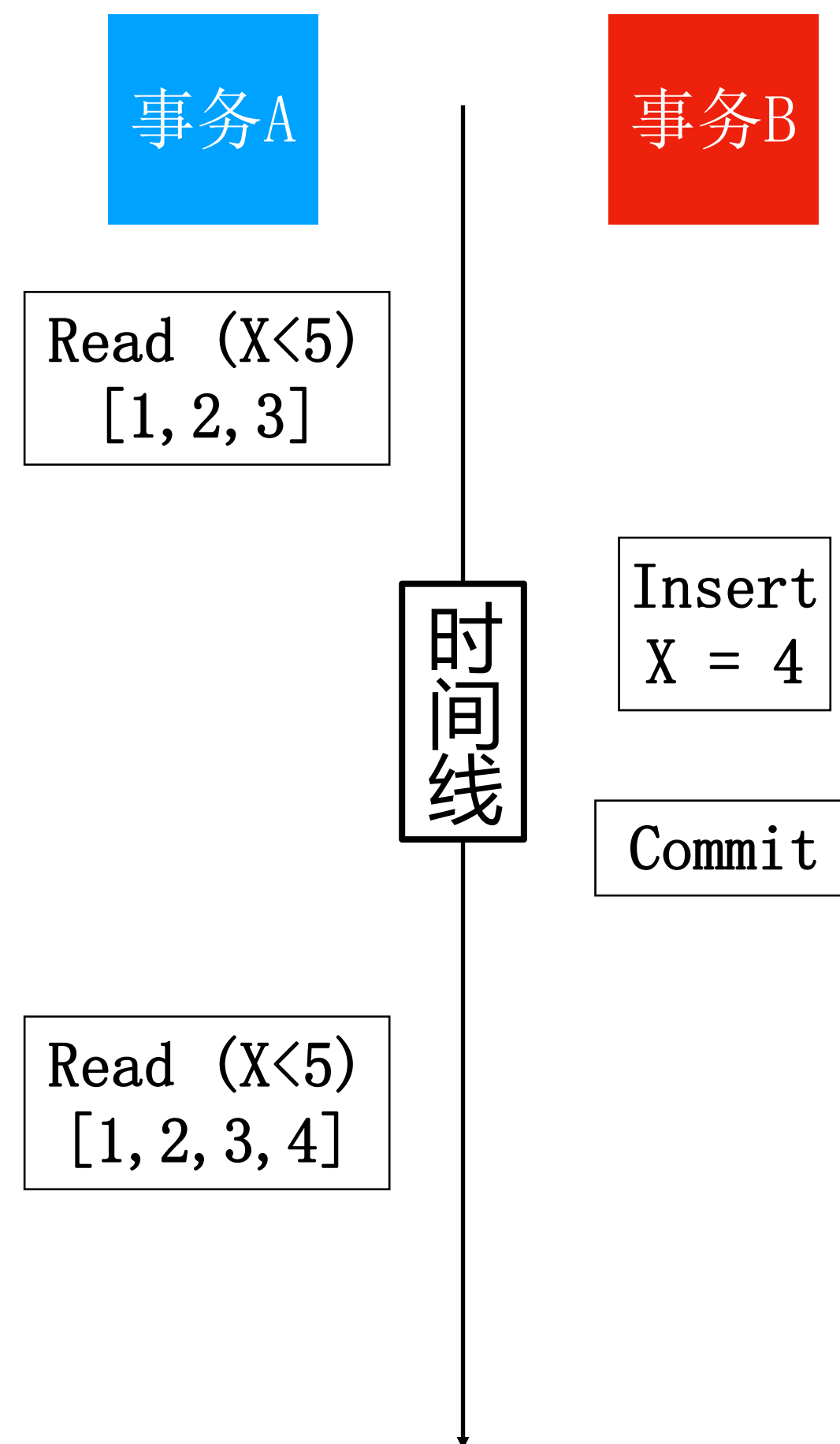
指事务读取**某个范围的数据**时,  
因为其他事务的操作导致前后读  
取的结果不一致。

幻读和不可重复读(Non-  
repeatable Reads)的区别在  
于——

**不可重复读**是针对**某一行确定的数据**。

**幻读**是针对不确定的**多行数据**。

幻读一般出现在有查询条件的范  
围查询中。



事务A查询  $X < 5$  的数据。由  
于事务B插入了一条  $X = 4$  的  
数据,导致事务A两次查询得  
到的结果不一样。

**「可重复读 Repeatable Read」**  
隔离级别, 会导致「幻读」。  
避免幻读, 需要将隔离级别提升  
至 **「串行化 Serializable」** (所  
有事务挨个排队执行)。

	可能导致的并发问题			
隔离级别	脏写 Dirty Writes	脏读 Dirty Reads	不可重复读 Non-repeatable Reads	幻读 Phantom Reads
读未提交 Read Uncommitted				
读已提交 Read Committed				
可重复读 Repeatable Read				
串行化 Serializable				

MySQL 默认 —— 可重复读

Oracle 默认 —— 读已提交

## 乐观并发控制(Optimistic Concurrency Control)

对于并发执行可能冲突的操作，假定其并不会真的冲突，允许并发执行。  
直到真正发生冲突时才去解决冲突，比如让事务回滚。

## 悲观并发控制(Pessimistic Concurrency Control)

对于并发执行可能冲突的操作，假定其必定发生冲突，不允许并发执行。  
通过让事务等待或者终止的方式使并行的操作串行化(Serializable)执行。

# 并发控制技术 - 加锁(Lock)

对于可能造成冲突的任何并发操作，比如读-写、写-读、写-写、通过加锁使它们互斥执行。

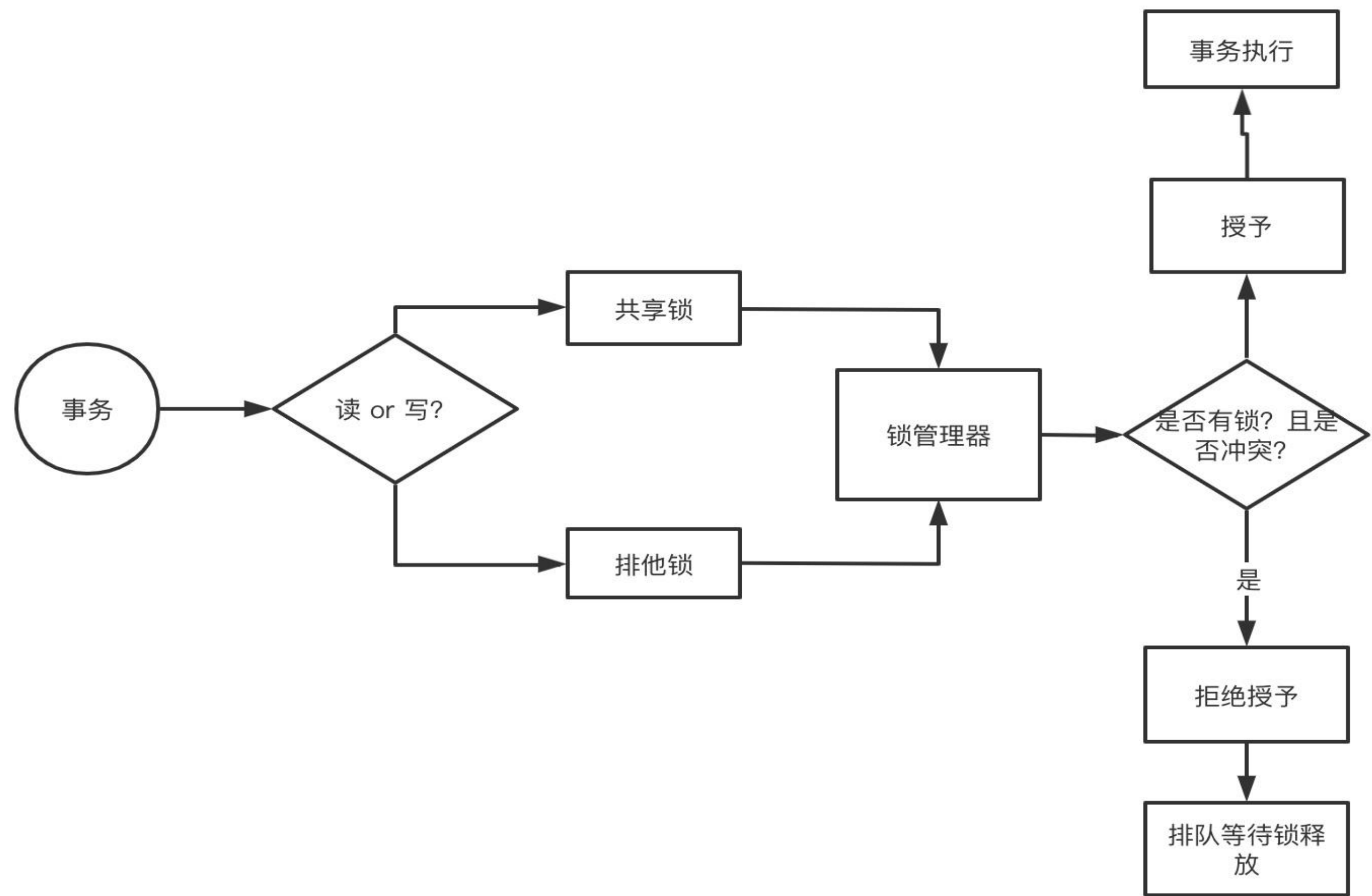
## 共享锁 Shared Locks (读锁)

如果 事务T 对 数据A 加上共享锁后，则其他事务只能对 A 再加共享锁，不能加排他锁。  
获准共享锁的事务**只能读数据**，不能修改数据。也叫读锁。

## 排他锁 Exclusive lock (写锁)

如果 事务T 对 数据A 加上排他锁后，则其他事务不能再对 A 加任何类型的锁。  
获准排他锁的事务**既能读数据，又能修改数据**。也叫写锁。





## 引发的问题

**死锁(Deadlock)**——  
多个事务持有锁并互相循环等待其他事务的锁导致所有事务都无法继续执行。

**饥饿(Hungry)**——  
数据A一直被加共享锁，导致事务一直无法获取A的排他锁(Exclusive lock)。

悲观并发控制

# 并发控制技术 - 时间戳(Timestamp)控制

对于可能造成冲突的任何并发操作，基于时间戳排序规则，选定某事务继续执行，其他事务回滚(Rollback)。

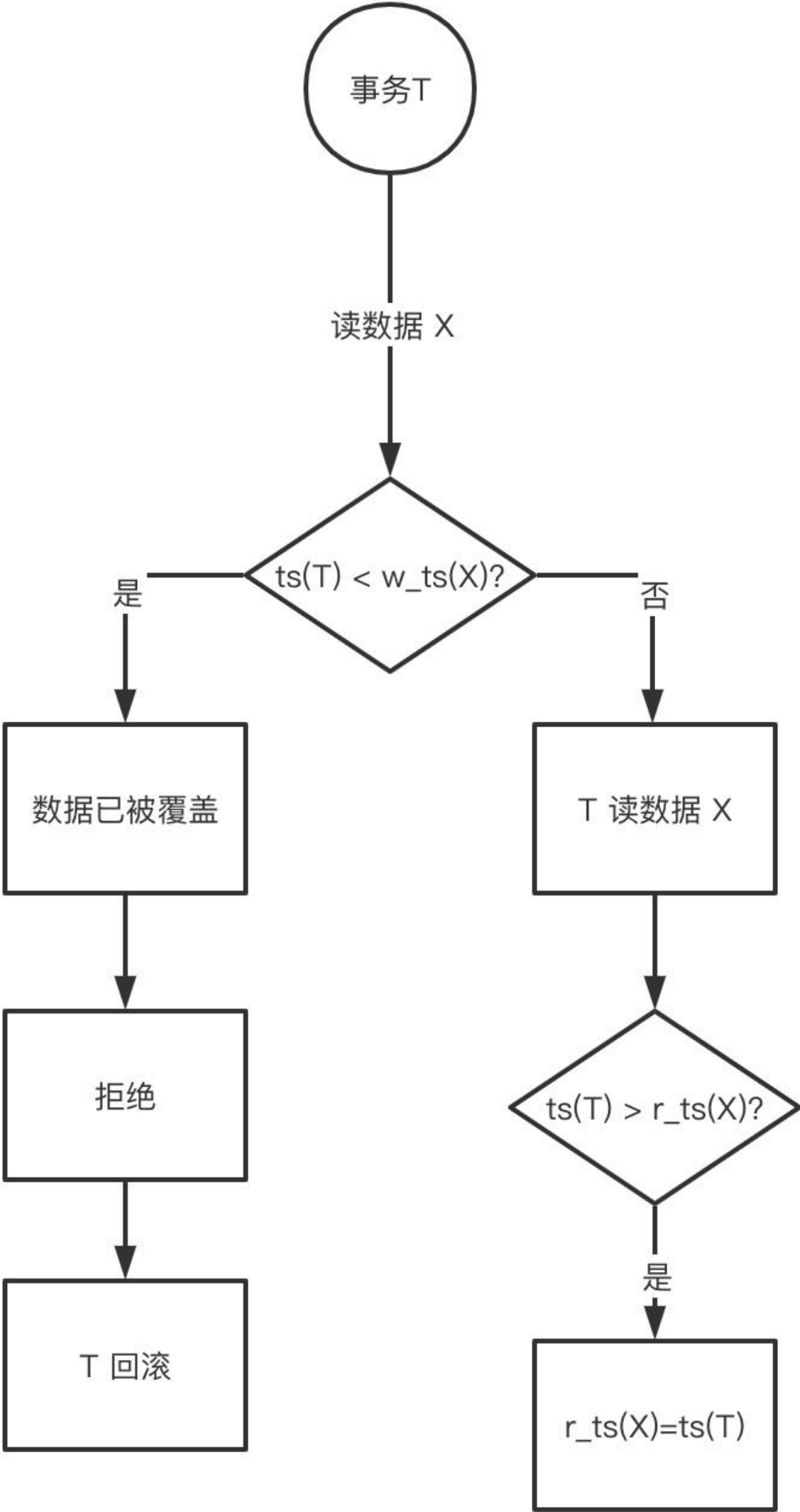
## 事务时间戳

每个事务开始时赋予其一个时间戳。可以是系统时钟，也可以是自增的计数器值。  
事务回滚时会赋予其一个新的时间戳。先开始的事务时间戳小于后开始事务的时间戳。  
标记为  $ts(T)$

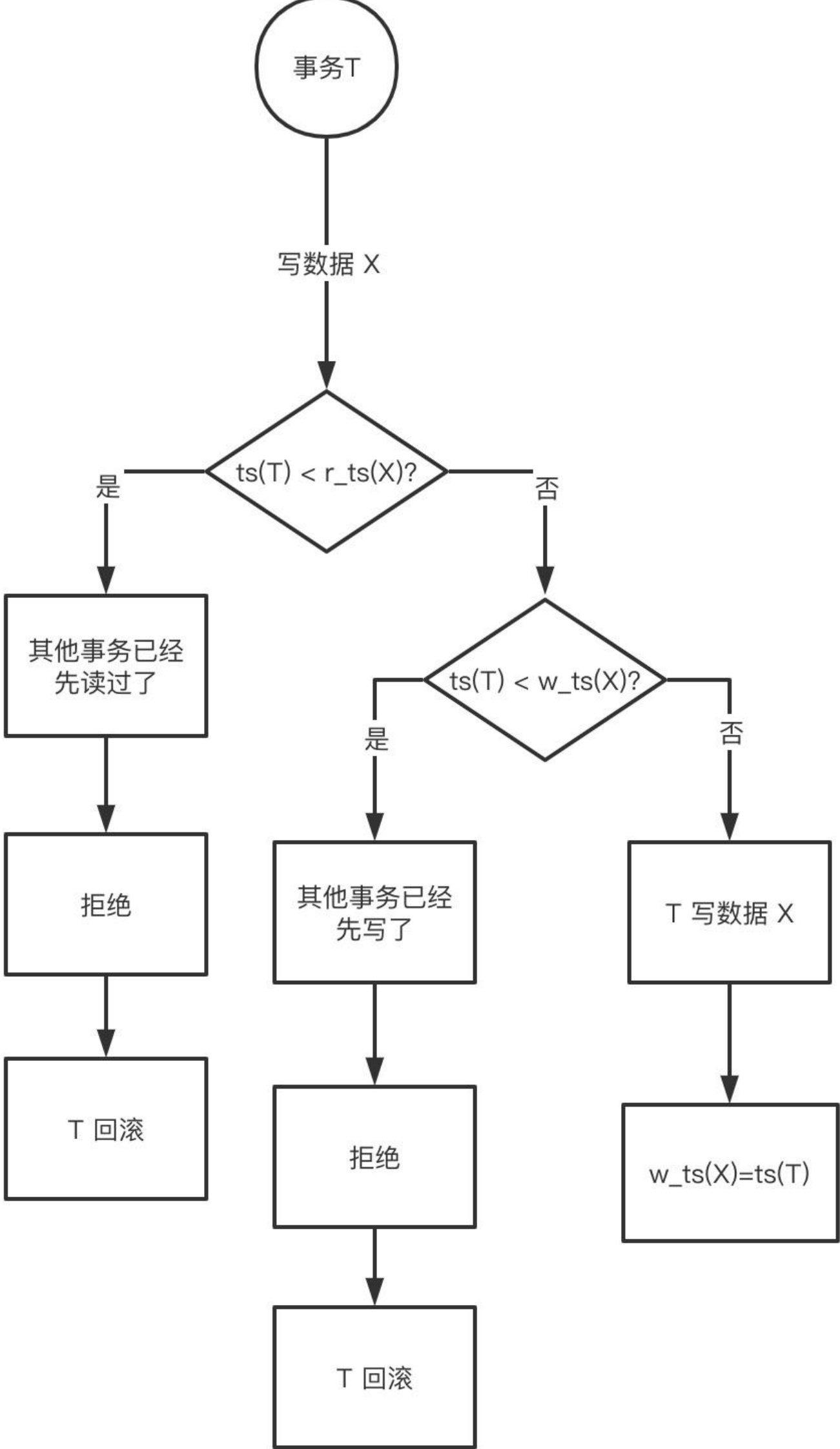
## 数据项时间戳

记录读写该数据的最新事务的时间戳  
标记为  $r\_ts(X)$ ,  $w\_ts(X)$

**读操作**  
事务 T 读数据 X



**写操作**  
事务 T 写数据 X



悲观并发控制

# 并发控制技术 - 有效性检查

事务对数据的更新首先在自己的工作空间进行，等到要写回数据库时才进行有效性检查，对不符合要求的事务进行回滚。

## 读阶段

数据项被读入并保存在事务的局部变量(Local Variable)中。所有后续写操作都是对局部变量进行，并不对数据库进行真正的更新。

## 有效性检查阶段

对事务进行有效性检查，判断是否可以执行写操作而不违反可串行性(Serializable)。如果失败，则回滚该事务。

## 写阶段

事务已通过有效性检查，将临时变量中的结果更新到数据库中。

对事务的时间戳进行比较完成的。

允许可能冲突的操作并发执行，因为每个事务操作的都是自己工作空间(Workspace)的局部变量。

直到有效性检查阶段发现了冲突才回滚。

乐观并发控制

# 数据库故障恢复

原子性(Atomicity)、持久性(Durability)  
和 一致性(Consistency) 保障



## 数据库系统故障

由于软件问题、硬件错误或者操作系统异常，导致数据库系统崩溃或终止。

## 事务故障

由于非法输入或者数据库出现死锁(Deadlock)等，导致事务无法继续执行。

故障会对事务、数据库状态（如数据库存储的数据）造成损坏。

需要对故障进行恢复，以保证数据库**一致性(Consistency)**，事务的**原子性(Atomicity)**以及**持久性(Durability)**。

以**日志(Write Ahead Log, WAL)**的方式记录对数据库的操作。

在故障时根据日志进行恢复，称为**日志恢复技术**。

问题——

增删查改，哪些操作需要记录日志？



1. 数据库系统为每个事务分配一个私有的工作区
2. 事务的读操作从磁盘中拷贝数据项到工作区中。执行写操作前，仅操作工作区内数据的拷贝。
3. 事务的写操作把数据输出到内存的缓冲区中。等到合适的时机，数据库的缓冲区管理器将数据写入到磁盘。

# 事务执行过程中的故障

## 立即修改

数据库在事务提交前出现故障，但是事务的部分修改已经写入磁盘中。这破坏了事务的**原子性(Atomicity)**。

## 延迟修改

数据库在事务提交后出现故障，但数据还在内存缓冲区(Buffer)中，未写入磁盘。系统恢复时将丢失此次已提交的修改。这破坏了事务的**持久性(Durability)**。

# 日志类别

## Undo 日志（撤销事务）

记录数据的旧值。事务撤销时，将事务更新的所有数据项恢复为日志中的旧值。

## Redo 日志（重做事务）

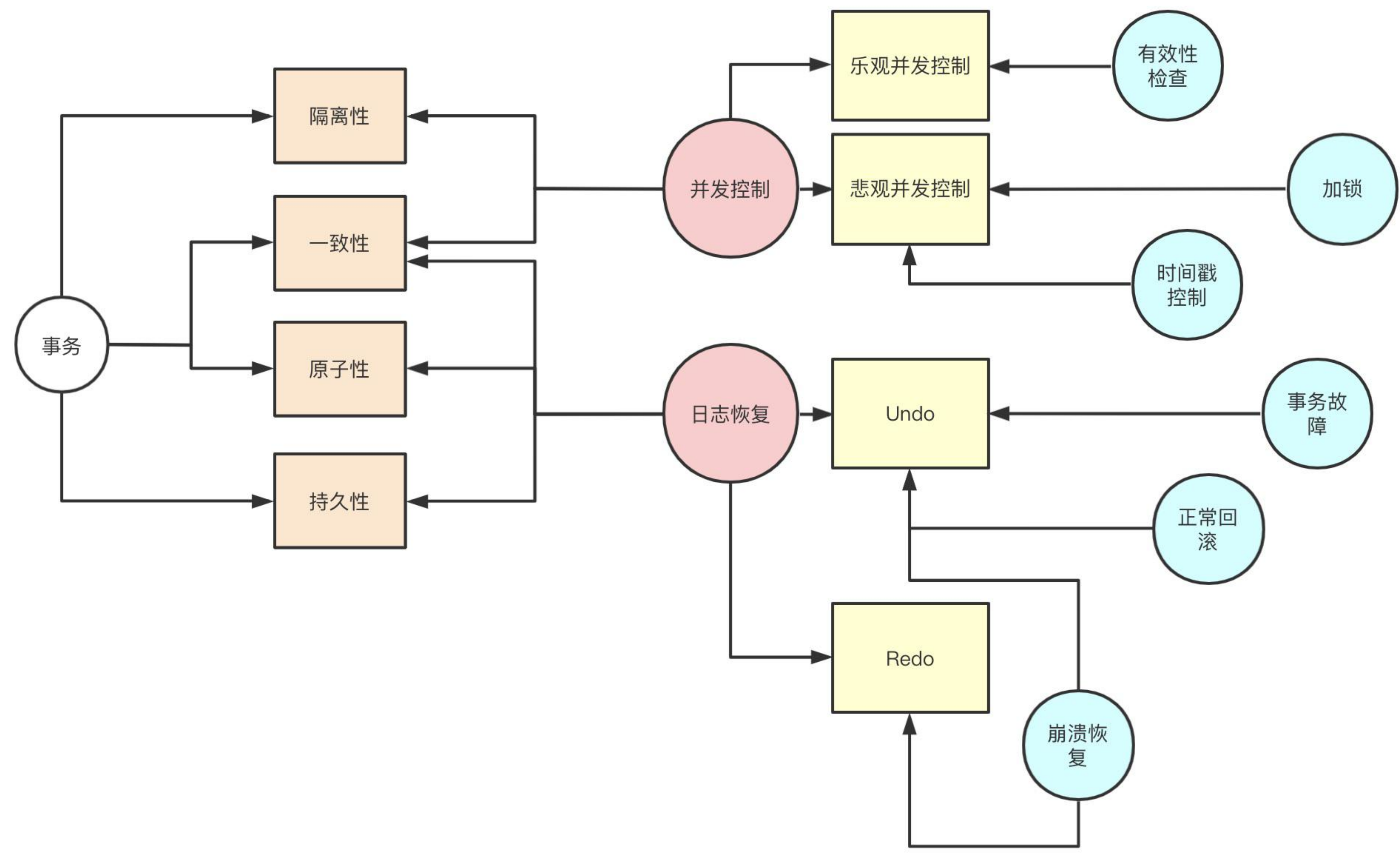
记录数据的新值。故障恢复时，将事务更新的所有数据项恢复为日志中的新值。

事务正常回滚 / 因事务故障终止事务

执行 Undo 日志

数据库系统从崩溃中恢复时

先执行 Redo，再执行 Undo。





## 参考资料

1. [https://docs.oracle.com/cd/B19306\\_01/server.102/b14231/ds\\_txns.htm](https://docs.oracle.com/cd/B19306_01/server.102/b14231/ds_txns.htm)
2. <https://tech.ebayinc.com/engineering/grit-a-protocol-for-distributed-transactions-across-microservices/>

## 分布式事务框架 seata

1. <https://hackernoon.com/fescar-a-distributed-transaction-solution-open-sourced-by-alibaba-f70c9b4c72a1>
2. <https://github.com/seata/seata>