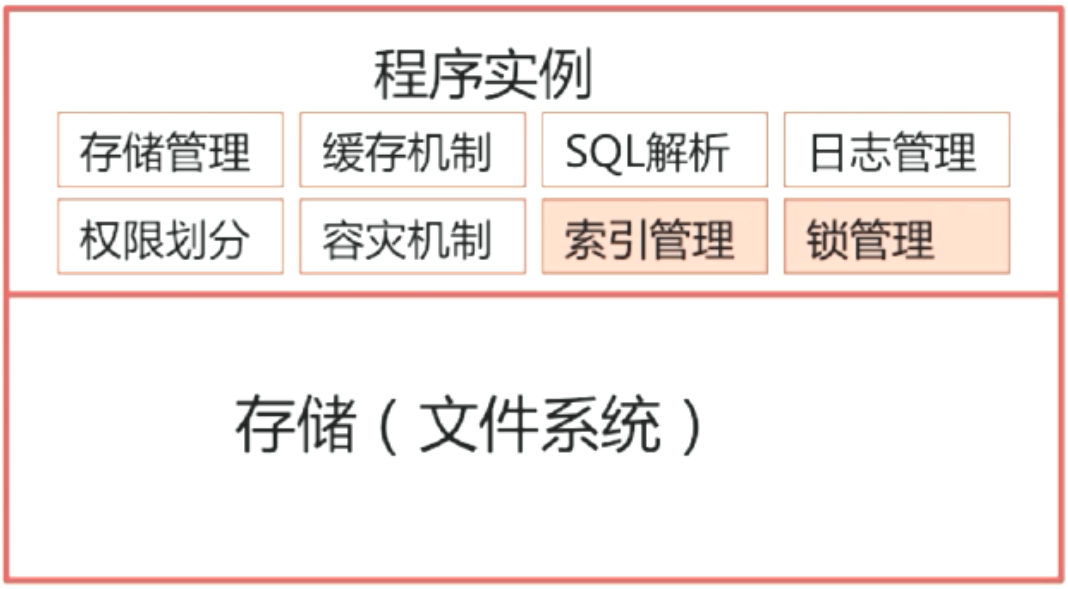
数据库架构

数据库架构可以分为存储文件系统和程序实例两大块，而程序实例根据不同的功能又可以分为如下小模块。



索引模块

常见的问题有：

- 为什么要使用索引
- 什么样的信息能成为索引
- 索引的数据结构
- 密集索引和稀疏索引的区别

为什么要使用索引

使用索引就像查字典一样，可以快速查询数据

什么样的信息能成为索引

主键、唯一键以及普通键等

索引的数据结构

- 生成索引，建立二叉查找树进行二分查找
- 生成索引，建立 B Tree 结构结构进行查找
- 生成索引，建立 B+ Tree 结构进行查找
- 生成索引，建立 Hash 结构进行查找

什么是 B Tree 索引？

B-Tree 是为磁盘等外存储设备设计的一种平衡查找树。因此在讲 B-Tree 之前先了解下磁盘的相关知识。

- 系统从磁盘读取数据到内存时是以磁盘块（block）为基本单位的，位于同一个磁盘块中的数据会被一次性读取出来，而不是需要什么取什么。
- InnoDB 存储引擎中有页（Page）的概念，页是其磁盘管理的最小单位。InnoDB 存储引擎中默认每个页的大小为 16 KB，可通过参数 innodb_page_size 将页的大小设置为 4K、8K、16K ，在 MySQL 中可通过如下命令查看页的大小：

```
1    mysql> show variables like 'innodb_page_size';
```

- 而系统一个磁盘块的存储空间往往没有这么大，因此 InnoDB 每次申请磁盘空间时都会是若干地址连续磁盘块来达到页的大小 16KB 。InnoDB 在把磁盘数据读入到磁盘时会以页为基本单位，在查询数据时如果一个页中

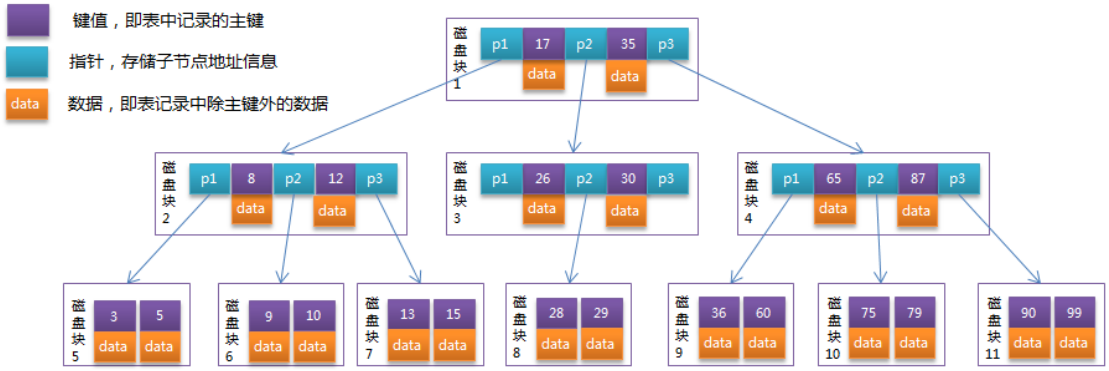
的每条数据都能有助于定位数据记录的位置，这将会减少磁盘 I/O 次数，提高查询效率。

B-Tree 结构的数据可以让系统高效的找到数据所在的磁盘块。为了描述 B-Tree，首先定义一条记录为一个二元组 [key, data] ，key 为记录的键值，对应表中的主键值，data 为一行记录中除主键外的数据。对于不同的记录，key 值互不相同。

一棵 m 阶的 B-Tree 有如下特性：

- 每个节点最多有 m 个孩子
    - 除了根节点和叶子节点外，其它每个节点至少有 Ceil(m/2) 个孩子
    - 若根节点不是叶子节点，则至少有 2 个孩子
- 所有叶子节点都在同一层，且不包含其它关键字信息
- 每个非叶子节点包含 n 个关键字信息（P0,P1,...Pn, k1,...kn）
    - 关键字的个数 n 满足：ceil(m/2)-1 <= n <= m-1
    - ki(i=1,...n) 为关键字，且关键字升序排序
    - Pi(i=0,...n) 为指向子树根节点的指针。P(i-1) 指向的子树的所有节点关键字均小于 ki ，但都大于 k(i-1)

B-Tree 中的每个节点根据实际情况可以包含大量的关键字信息和分支，如下图所示为一个 3 阶的 B-Tree：



- 每个节点占用一个盘块的磁盘空间，一个节点上有两个升序排序的 key 和三个指向子树根节点的 point ，point 存储的是子节点所在磁盘块的地址。两个 key 划分成的三个范围域，对应三个 point 指向的子树的数据的范围域。
- 以根节点为例，key 为 17 和 35 ，P1 指针指向的子树的数据范围为小于 17 ，P2 指针指向的子树的数据范围为 [17~35] ，P3 指针指向的子树的数据范围为大于 35 。

模拟查找 key 为 29 的过程：

- 1、根据根节点找到磁盘块 1 ，读入内存。【磁盘 I/O 操作第 1 次】
- 2、比较 key 29 在区间（17,35），找到磁盘块 1 的指针 P2 。
- 3、根据 P2 指针找到磁盘块 3 ，读入内存。【磁盘 I/O 操作第 2 次】
- 4、比较 key 29 在区间（26,30），找到磁盘块 3 的指针 P2。
- 5、根据 P2 指针找到磁盘块 8 ，读入内存。【磁盘 I/O 操作第 3 次】
- 6、在磁盘块 8 中的 key 列表中找到 eky 29 。

分析上面过程，发现需要 3 次磁盘 I/O 操作，和 3 次内存查找操作。由于内存中的 key 是一个有序表结构，可以利用二分法查找提高效率。而 3 次磁

盘 I/O 操作是影响整个 B-Tree 查找效率的决定因素。B-Tree 相对于 AVLTree 缩减了节点个数，使每次磁盘 I/O 取到内存的数据都发挥了作用，从而提高了查询效率。

什么是 B+Tree 索引？

B+Tree 是在 B-Tree 基础上的一种优化，使其更适合实现外存储索引结构，InnoDB 存储引擎就是用 B+Tree 实现其索引结构。

从上一节中的 B-Tree 结构图中可以看到，每个节点中不仅包含数据的 key 值，还有 data 值。而每一个页的存储空间是有限的，如果 data 数据较大时将会导致每个节点（即一个页）能存储的 key 的数量很小，当存储的数据量很大时同样会导致 B-Tree 的深度较大，增大查询时的磁盘 I/O 次数，进而影响查询效率。在 B+Tree 中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存储 key 值信息，这样可以大大加大每个节点存储的 key 值数量，降低 B+Tree 的高度。

B+Tree 相对于 B-Tree 有几点不同：

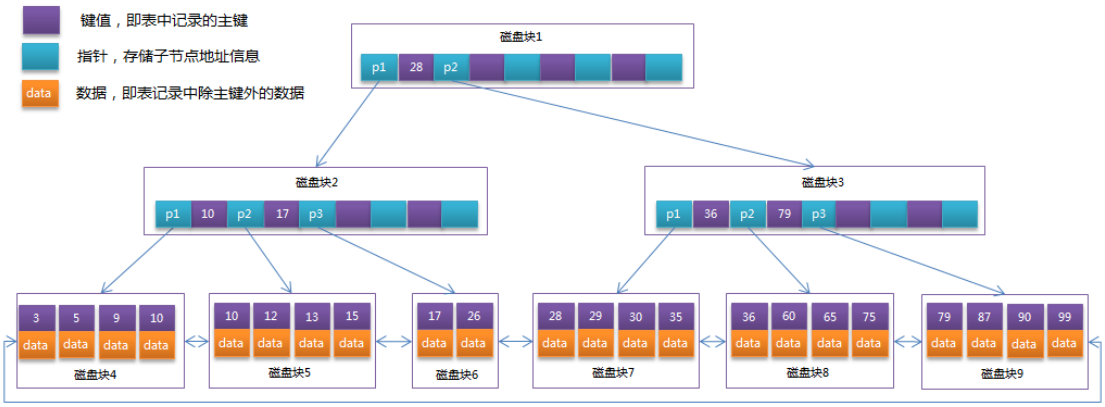
- 非叶子节点只存储键值信息。
- 所有叶子节点之间都有一个链指针。
- 数据记录都存放在叶子节点中。

B+ Tree 更适合用来做存储索引：

- B+ 数的磁盘读写代价更低

- B+ 数的查询效率更加稳定
- B+ 数更有利于对数据库的扫描（范围查询）

将上一节中的 B-Tree 优化，由于 B+Tree 的非叶子节点只存储键值信息，假设每个磁盘块能存储 4 个键值及指针信息，则变成 B+Tree 后其结构如下图所示：



磁盘块 4 中的 10 数据，画错了，范围在[K[i], K[i+1])，左闭右开

- 通常在 B+Tree 上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点，而且所有叶子节点（即数据节点）之间是一种链式环结构。因此可以对 B+Tree 进行两种查找运算：一种是对于主键的范围查找和分页查找，另一种是从根节点开始，进行随机查找。

可能上面例子中只有 22 条数据记录，看不出 B+Tree 的优点，下面做一个推算：

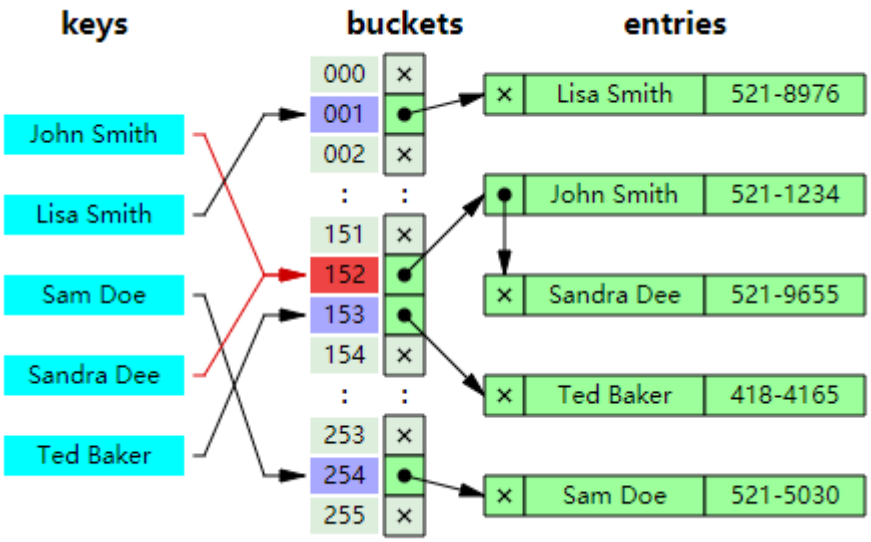
- InnoDB 存储引擎中页的大小为 16KB，一般表的主键类型为 INT（占用 4 个字节） 或 BIGINT（占用 8 个字节），指针类型也一般为 4 或 8 个字节，也就是说一个页（B+Tree 中的一个节点）中大概存储 16KB/(8B+8B)=1K 个键值（因为是估值，为方便计算，这里的 K 取值

为〖10〗^3）。也就是说一个深度为 3 的 B+Tree 索引可以维护 10^3 *10^3* 10^3 = 10 亿 条记录。

- 实际情况中每个节点可能不能填充满，因此在数据库中，B+Tree 的高度一般都在 2~4 层。MySQL 的 InnoDB 存储引擎在设计时是将根节点常驻内存的，也就是说查找某一键值的行记录时最多只需要 1~3 次磁盘 I/O 操作。

什么是 hash 索引？

基于哈希表实现，优点是查找非常快。如下图 ：



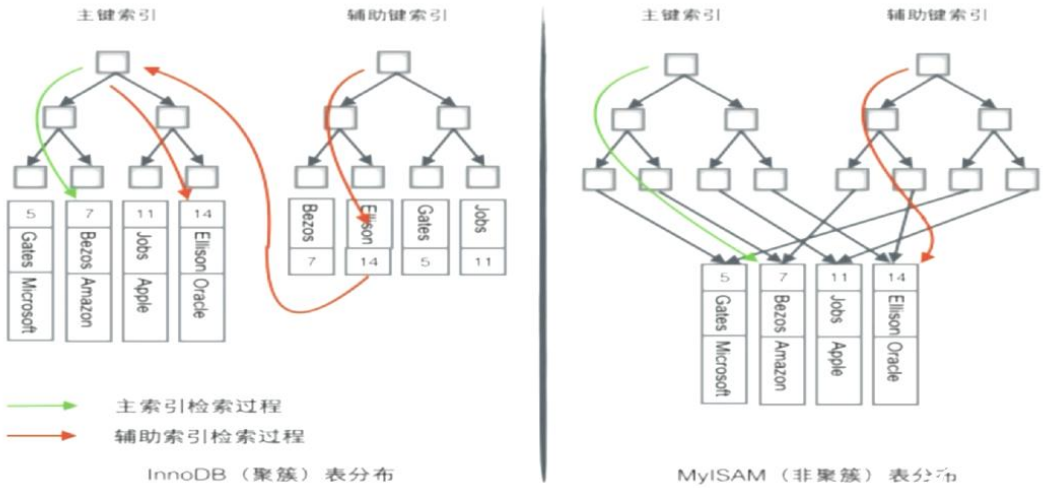
哈希索引就是采用一定的哈希算法，将键值换算成新的哈希值，检索时不需要想 B+Tree 那样从根结点开始查找，而是经过计算直接定位，所以速度很快。

但是也有限制：

- 只支持精确查找，不能用于部分查找和范围查找。无法排序和分组。因为原来有序的键值经过哈希算法很可能打乱。

- 如果哈希冲突很多，查找速度很慢。比如在有大量重复键值的情况下。
- 不能利用部分索引查询
- 不能

聚集索引与非聚集索引



MyISAM 索引与 InnoDB 索引的区别？

- InnoDB 索引是聚簇索引，MyISAM 索引是非聚簇索引。
- InnoDB 的主键索引的叶子节点存储着行数据，因此主键索引非常高效。
- MyISAM 索引的叶子节点存储的是行数据地址，需要再寻址一次才能得到数据。
- InnoDB 非主键索引的叶子节点存储的是主键和其他带索引的列数据，因此查询时做到覆盖索引会非常高效

如何定位并优化慢查询的 Sql

需要具体场景具体分析，其大致思路

- 根据慢日志定位到慢查询的 sql

- 使用 explain 等工具分析 sql
- 修改 sql 或者尽量让 sql 走索引

定位慢查询 sql

开启慢查询日志即可

文件方式配置 MySQL 慢查询的方法：

- 查询 MySQL 慢查询状态的方法：

```
1	SHOW VARIABLES LIKE '%query%';
```

- 在 mysql 配置文件 my.cnf 中增加：

```
1	log-slow-queries=/opt/data/slowquery.log
2	long_query_time=2
3	log-queries-not-using-indexes
```

- 命令方式配置 MySQL 慢查询的方法：

```
1	set global slow_query_log=on;
2	set global long_query_time=1;
3	set global slow_query_log_file='/opt/data/slow_query.log';
```

- 解析 MySQL 慢查询日志的方法，按照 sql 执行时间最长的前 20 条 sql：

```
mysqldumpslow -s t -t 20 -g 'select' /opt/data/slowquery.log
```

在 log 中就能找到慢查询的 sql。

Explian 关键字

- Explain 命令在解决数据库性能上是第一推荐使用命令，大部分的性能问题可以通过此命令来简单的解决，Explain 可以用来查看 SQL 语句的执行效 果，可以帮助选择更好的索引和优化查询语句，写出更好的优化语句。

  Explain 语法：explain select … from … [where …]

  例如：explain select * from news;

  输出：

```
+----+-------------+-------+-------+------------------+---------+---------+---
| id | select_type | table | type  | possible_keys | key | key_len | ref | rows
+----+-------------+-------+-------+------------------+---------+---------+---
```

  下面对各个属性进行了解：

  1、id：这是 SELECT 的查询序列号

  2、select_type：select_type 就是 select 的类型，可以有以下几种：

  SIMPLE：简单 SELECT(不使用 UNION 或子查询等)

  PRIMARY：最外面的 SELECT

UNION：UNION 中的第二个或后面的 SELECT 语句

DEPENDENT UNION：UNION 中的第二个或后面的 SELECT 语句，取决于外面的查询

UNION RESULT：UNION 的结果。

SUBQUERY：子查询中的第一个 SELECT

DEPENDENT SUBQUERY：子查询中的第一个 SELECT，取决于外面的查询

DERIVED：导出表的 SELECT(FROM 子句的子查询)

3、table：显示这一行的数据是关于哪张表的

4、type：这列最重要，显示了连接使用了哪种类别,有无使用索引，是使用 Explain 命令分析性能瓶颈的关键项之一。

结果值从好到坏依次是：

system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > range > index > ALL

一般来说，得保证查询至少达到 range 级别，最好能达到 ref，否则就可能会出现性能问题。

5、possible_keys：列指出 MySQL 能使用哪个索引在该表中找到行

6、key：显示 MySQL 实际决定使用的键（索引）。如果没有选择索引，键是 NULL

7、key_len：显示 MySQL 决定使用的键长度。如果键是 NULL，则长度为 NULL。使用的索引的长度。在不损失精确性的情况下，长度越短越好

8、ref：显示使用哪个列或常数与 key 一起从表中选择行。

9、rows：显示 MySQL 认为它执行查询时必须检查的行数。

10、Extra：包含 MySQL 解决查询的详细信息，也是关键参考项之一。

Distinct

一旦 MYSQL 找到了与行相联合匹配的行，就不再搜索了

Not exists

MYSQL 优化了 LEFT JOIN，一旦它找到了匹配 LEFT JOIN 标准的行，就不再搜索了

Range checked for each

Record（index map:#）

没有找到理想的索引，因此对于从前面表中来的每一 个行组合，MYSQL 检查使用哪个索引，并用它来从表中返回行。这是使用索引的最慢的连接之一

Using filesort

看 到这个的时候，查询就需要优化了。MYSQL 需要进行额外的步骤来发

> 现如何对返回的行排序。它根据连接类型以及存储排序键值和匹配条件的全部行的行指针来 排序全部行
>
> Using index
>
> 列数据是从仅仅使用了索引中的信息而没有读取实际的行动的表返回的，这发生在对表 的全部的请求列都是同一个索引的部分的时候
>
> Using temporary
>
> 看到这个的时候，查询需要优化了。这 里，MYSQL 需要创建一个临时表来存储结果，这通常发生在对不同的列集进行 ORDER BY 上，而不是 GROUP BY 上
>
> Using where
>
> 使用了 WHERE 从句来限制哪些行将与下一张表匹配或者是返回给用户。如果不想返回表中的全部行，并且连接类型 ALL 或 index， 这就会发生，或者是查询有问题

其他一些 Tip：

当 type 显示为 “index” 时，并且 Extra 显示为 “Using Index”，表明使用了覆盖索引。

联合索引的最左匹配原则的成因

看看如下博客即可

- 联合索引的最左前缀匹配原则
- mysql 索引最左匹配原则的理解?

索引是建立得越多越好的吗

- 数据量小的表不需要建立索引，建立会增加额外的索引开销
- 数据变更需要维护索引，因此更多的索引意味着更多的维护成本
- 更多的索引意味着也需要更多的空间

锁模块

常见问题

- MyISAM 与 InnoDB 关于锁方面的区别是什么
- 数据库事务的四大特性
- 事务隔离级别以及各级别下的并发访问问题
- InnoDB 可重复读隔离级别下如何避免幻读
- RC、RR 级别下的 InnoDB 的非堵塞如果实现

MyISAM 与 InnoDB 关于锁方面的区别是什么

- MyISAM 默认用的是表级锁，不支持行级锁
- InnoDB 默认用的是行级锁，也支持表级锁

数据库锁的分类

- 按锁的粒度划分，可分为表级锁、行级锁和页级锁
- 按锁的级别划分，可分为共享锁和排他锁
- 按加锁的方式划分，可分为自动锁和显示锁
- 按操作划分，可分为 DML 锁和 DDL 锁
- 按使用方式划分，可分为乐观锁和悲观锁

ACID

1. 原子性（Atomicity）

事务被视为不可分割的最小单元，事务的所有操作要么全部提交成功，要么全部失败回滚。

回滚可以用回滚日志来实现，回滚日志记录着事务所执行的修改操作，在回滚时反向执行这些修改操作即可。

2. 一致性（Consistency）

数据库在事务执行前后都保持一致性状态。在一致性状态下，所有事务对一个数据的读取结果都是相同的。

3. 隔离性（Isolation）

一个事务所做的修改在最终提交以前，对其它事务是不可见的。

4. 持久性（Durability）

一旦事务提交，则其所做的修改将会永远保存到数据库中。即使系统发生崩溃，事务执行的结果也不能丢失。

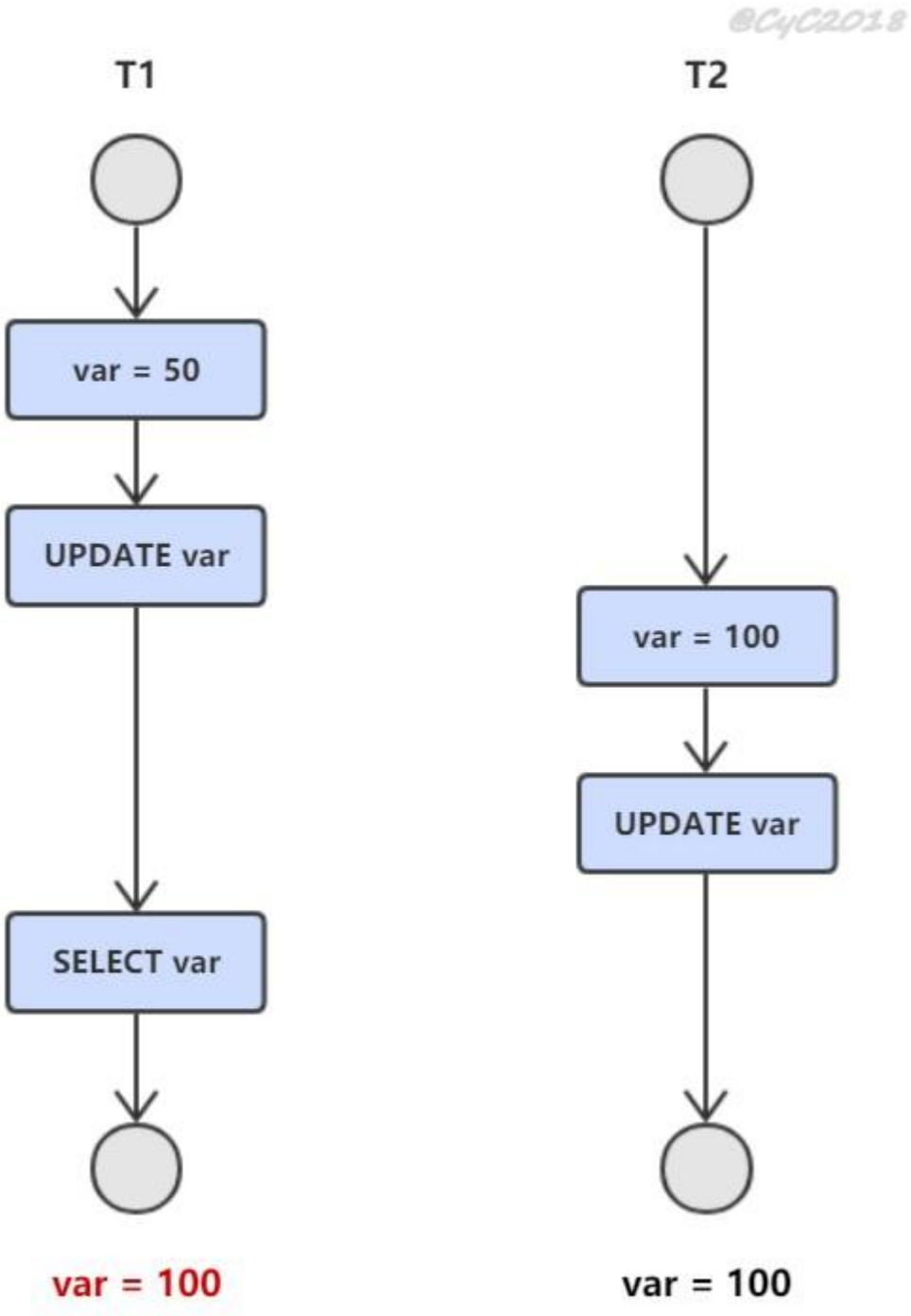
使用重做日志来保证持久性。

并发一致性问题

在并发环境下，事务的隔离性很难保证，因此会出现很多并发一致性问题。

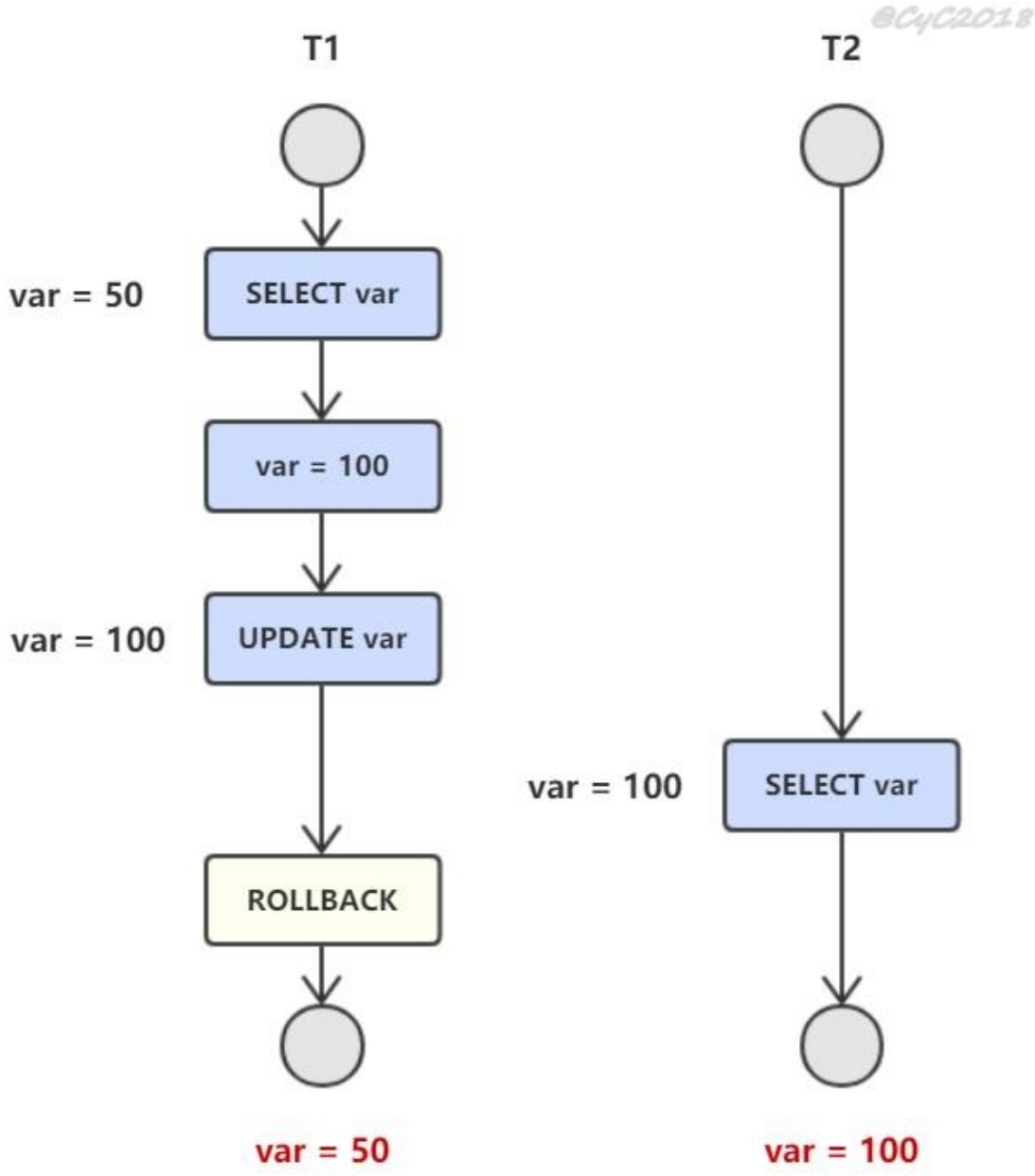
丢失修改

T1 和 T2 两个事务都对一个数据进行修改，T1 先修改，T2 随后修改，T2 的修改覆盖了 T1 的修改。



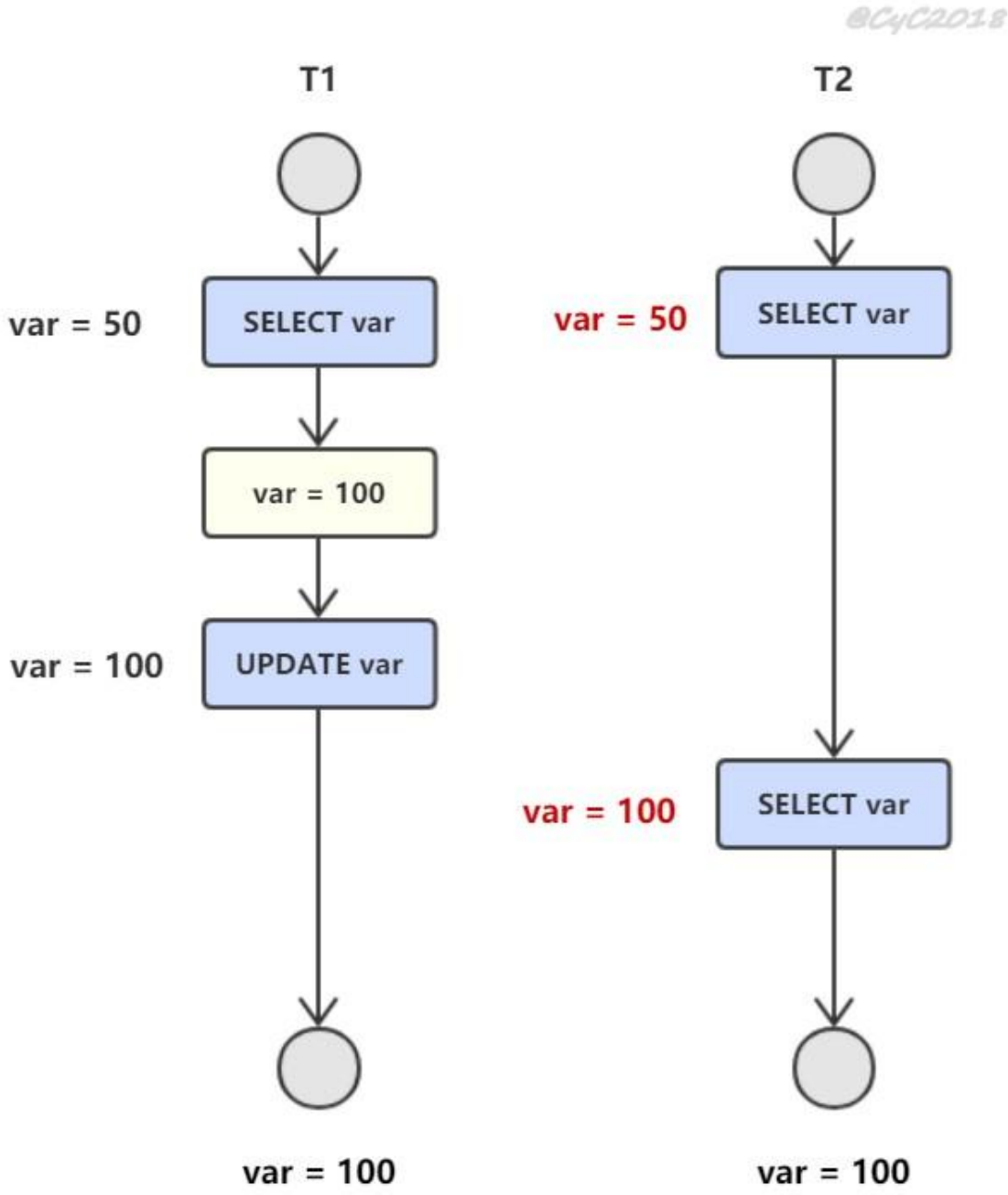
读脏数据

T1 修改一个数据，T2 随后读取这个数据。如果 T1 撤销了这次修改，那么 T2 读取的数据是脏数据。



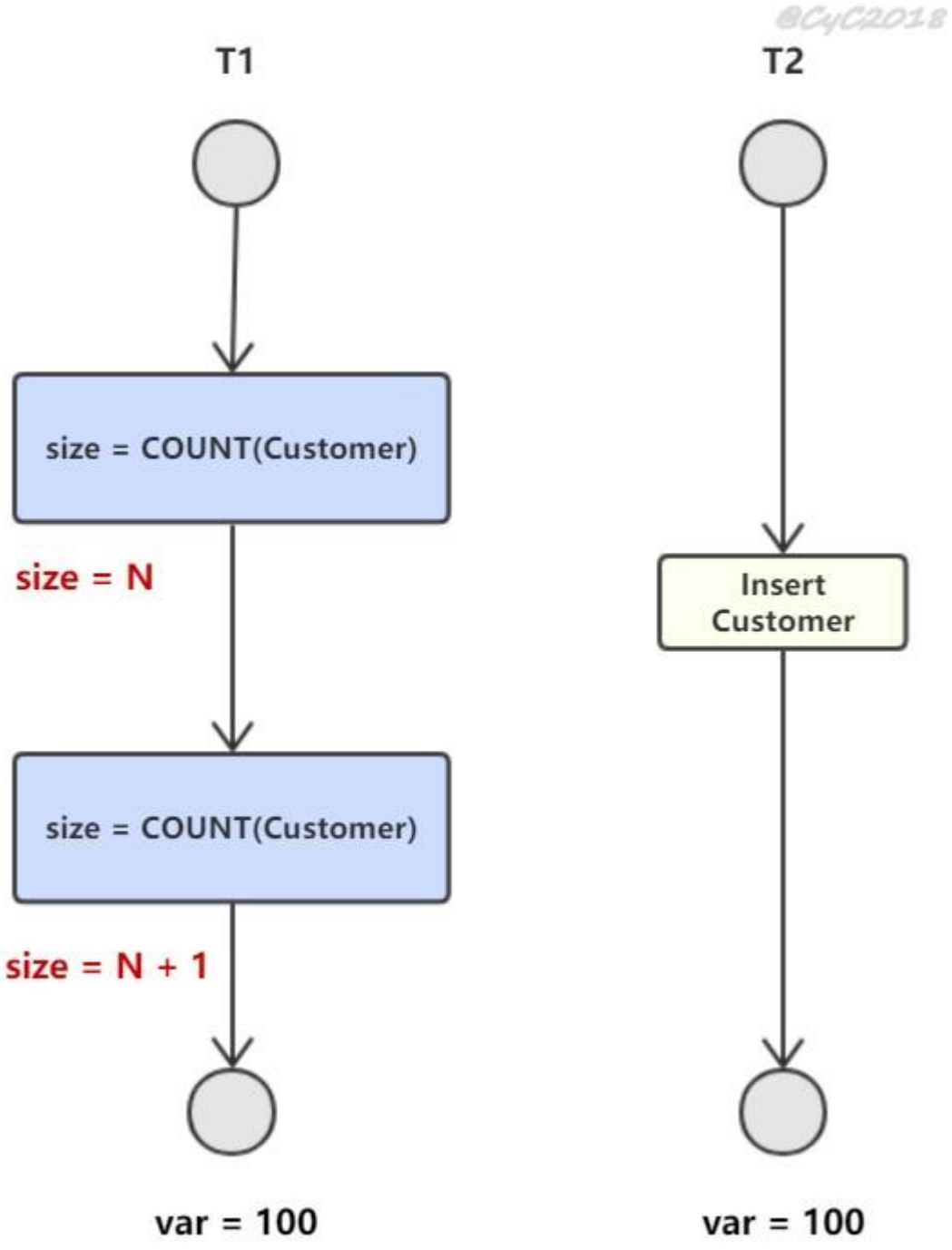
不可重复读

T2 读取一个数据，T1 对该数据做了修改。如果 T2 再次读取这个数据，此时读取的结果和第一次读取的结果不同。



幻影读

T1 读取某个范围的数据，T2 在这个范围内插入新的数据，T1 再次读取这个范围的数据，此时读取的结果和和第一次读取的结果不同。



隔离级别

未提交读（READ UNCOMMITTED）

事务中的修改，即使没有提交，对其它事务也是可见的。

提交读（READ COMMITTED）

一个事务只能读取已经提交的事务所做的修改。换句话说，一个事务所做的修改在提交之前对其它事务是不可见的。

可重复读（REPEATABLE READ）

保证在同一个事务中多次读取同样数据的结果是一样的。

可串行化（SERIALIZABLE）

强制事务串行执行。

隔离级别	脏读	不可重复读	幻影读
未提交读	√	√	√
提交读	×	√	√
可重复读	×	×	√
可串行化	×	×	×