

Big Data lesson 7

SQL 章节：

程序 = 算法 + 数据结构 MySQL 的底层实现是 B+ 树

Oracle 的底层实现是 B 树

Hash 的原理 DataBase 的存储结构：数据页 page

数据库服务器的存储介质 < 内存：临时存储，容量有限，意外会造数据丢失
硬盘：永久存储

索引放在硬盘上，I/O 存取消耗的时间高很多

所以我们用二分查找法查找 data $O(\log n)$

数据的结构会影响二叉树的结构，worst case 会变成链表 $O(n)$

平衡二叉树 (AVL 树)：在 binary tree 上加了约束

约束：每个节点的左子树和右子树的高度差不能超过 1 ($\text{bf} = 1$)，并且左右两个子树都是一棵平衡二叉树

B 树 \rightarrow Balance 树

当数据量大时，使用平衡二叉树进行检查，效率好吗？

不一定，当数据量为 n ，树的深度为 $O(\log n)$ ，当 n 比较大时，深度也是比较高
数据查询时间依赖于磁盘 I/O 的次数 \Rightarrow 效率不高

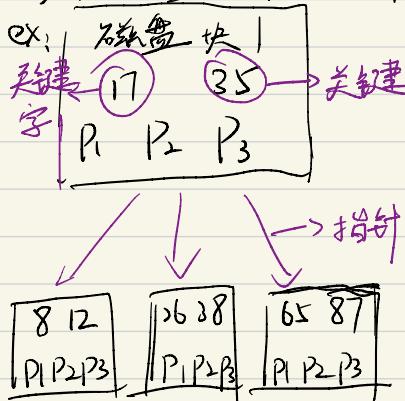
可以将二叉树改成 M 叉树，改变深度，当数据量很大时，可以降低树的深度
 \Rightarrow 提升检索效率

B 树是平衡的多路搜索树，它们的高度远小于平衡二叉树的高度

B 树的每个节点最多可以包括 M 个子节点 (M 称为 B 树的阶)

每个磁盘块中包括了关键字和子节点的指针

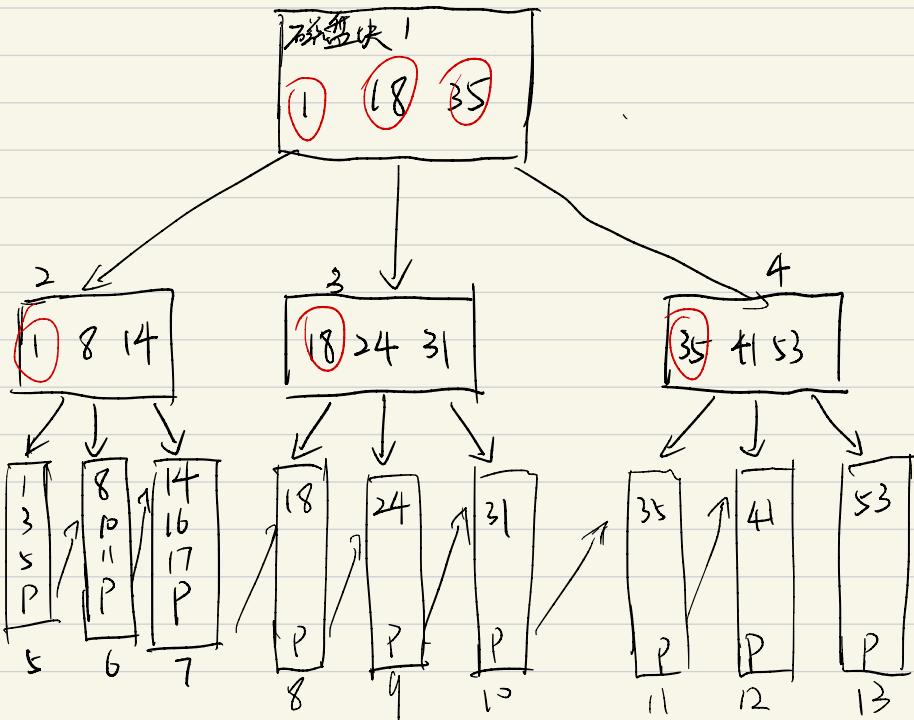
如果一个磁盘块中包括x个关键字,那么指针数就为x+1



比较的次数并不少,但实际上在内存内进行比较 \Rightarrow 时间可忽略不计,读取磁盘块的操作不行

磁盘 I/O 是用时的主要因素 \Rightarrow B树相比于平衡二叉树来说磁盘 I/O 操作较少,所以效率高

B+树: 基于B树的改进, MySQL采用的是B+树的索引方式



- 1) k 个孩子的节点就有 k 个关键字 而在 B 树中，孩子数量 = 关键字数 + 1
- 2) 非叶子节点的关键字也会同时存在在子节点中，并且是子节点中所有关键字的
最大(或最小)
- 3) 非叶子节点仅用于索引，不保存数据记录，跟记录有关的信息都放入
叶子节点中。
- 4) 所有关关键字都在叶子节点出现，叶子节点构成一个有序链表，而且叶子节点
本身按照关键字从小到大顺序连接

B+ 树的中间节点并不直接存储 data 的好处：

- ① 查询效率更稳定，B+ 树每次只有访问到叶子节点才能找到数据，而 B 树，
非叶子节点也会存储数据
- ② B+ 树的查询效率更高（针对单个关键字）
- ③ 对索引进行范围查询时，B+ 树效率更高

Hash 索引：

Hash: Hash 本身是散列函数，可大幅提高检索数据的效率，MD5 就是 Hash 函数
的一种

(SHA1, SHA2, SHA3)

↓
加密

Hash 算法 将输入转为输出

相同的输入永远可以得到相同的输出，假设输入内容有微小偏差，在输出中
通常有不同的结果

→ 所以想验证两个文件是否相同，只需对 Hash 函数的结果进行比较即可
Hash 的时间复杂度：O(1) 数组：O(n)

Hash > B+ > O(n)

为什么 MySQL 中不都用 Hash？因为 Hash 是等值查询；而 MySQL 中很多是范围查询
避免 Hash 冲突，因为 SQL 中会有重复值

输入 space > 桶

Hash索引的字节数多少较适合：一般4个字节就够了。在 Hash 值相同情况下，会进一步比较桶(Bucket)中的键值，来找到最终的数据行

Hash值的字节数多可以是 16位, 32位 等, ex: MD5 32位已很安全, 重复率低

diff between Hash索引 & B+树索引：

- 1) Hash索引不能进行范围查询, B+树可以, 因为 Hash索引指向的 data 是无序的
- 2) Hash索引不支持联合索引的最左侧原则, 而 B+树可以
- 3) Hash索引不支持 order by 排序, 也无法用 Hash索引进行模糊查询, 因为
- 4) 对于等值查询来说, Hash索引的效率非常高, 除非重复值很多

B+树在RDBMS中的使用更广泛

在健值型(key-value)数据库中, Hash index 效率非常高, Redis存储的核心就是 Hash表

MySQL中的Memory存储引擎支持Hash存储

MySQL的InnoDB存储引擎还有个“全局 Hash索引”：当某个索引值使用非常频繁时, 它会在B+树索引基础上再创建 Hash索引, B+树也有3种Hash的优点

database 的存储结构:

记录是按照行来存储的, 但数据库读取不是以行为单位, 否则一次读取了所有存储空间的基本单位是页(page), 一个页中可存储多个行记录 (Row)

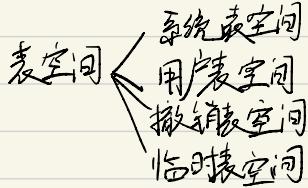
在数据库中, 还存在区(Extent)、段(segment)和表空间(Tablespace)



区(Extent)比页大一级，在InnoDB中，一个区会分配64个连续的页。因为InnoDB中一页大小默认是16 KB，所以一个区大小： $64 \times 16\text{ KB} = 1\text{ MB}$

段(segment)，由一个或多个区组成，区在文件系统是一个连续分配的空间，但在段中不要求区与区之间是相邻的。段是数据库的分配单位。

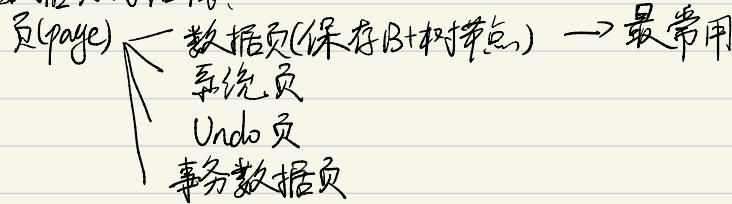
表空间(Tablespace)，是一个逻辑容器，表空间存储的对象是段，一个表空间可有多个段，但一个段只能有一个表空间



InnoDB < 共享表空间：多个表共用一个表空间
 独立表空间：每张表有一个独立的表空间，数据和索引都会保存在自己的表空间中

独立的表空间可以在不同的数据库之间进行迁移

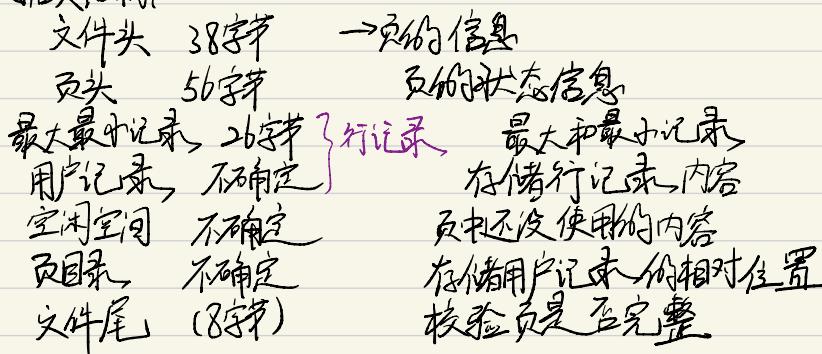
数据页的结构：



表页的大小限定了表行的最大长度

MySQL InnoDB default 16KB SQL Server 8KB Oracle Block=page 2KB
4, 8, 16, 32, 64 KB

数据页结构：



文件通用部分(文件头和文件尾)

类似于集装箱，将页的内容封装，用文件头和尾校验来确保页的传输是完整

文件尾的校验，采用Hash算法校验，假如在页传输时，中断电，造成该页传输不完整，use 文件尾的校验和 (checksum值)与文件头校验和作比较

=, ✓ ≠, 有问题

记录部分：作用是存储记录，“最小和最大记录”和“用户记录”占页结构的主要空间

索引部分，对应的是页目录，起到记录的索引作用，记录在页中以单向链表形式存储的单向链表。insert & delete 非常方便，但检索效率不高，最差情况下需要遍历链表上的所有节点才能完成检索。所以在页目录中提供了二分查找的方式。

页目录存储的槽，槽相当于分组记录的索引，通过槽查找记录，实际就是二分查找。

B+树按节点类型可分为：

叶子节点：存储行记录

非叶子节点：节点的高度大于0，存储索引键和页面指针，并不存储记录本身

在B+树中，每个节点都是一个页，每次新建节点，就申请一个页空间。
同层上的节点之间，通过页的结构构成一个双向链表。

非叶子节点：包括多个索引行，每个索引行中

① B+树的记录检索原理：通过B+树的索引查询记录，首先是从B+树的根开始逐层检索，直到找到叶子节点。
② 将数据页加载到内存中，页目录中的槽(slot)采用二分查找的方式先找粗略的记录分组，再在分组中通过链表遍历来查找记录。

database：由一个或多个表空间组成
表空间 > 段 > 区 > 页 > 行

SQL事务处理：5.5版本后默认存储引擎是InnoDB，原因是InnoDB支持事务。

(transaction)

事务的特性：事务保证了一次处理的完整性，也保证 database 中数据一致性

如果在 add, delete, update 时某个环节出错，允许我们回滚还原

事务非常应用在安全性高的场景，如金融等行业

ACID 特性：

原子性：不可分割，要么执行，要么不执行

一致性：数据库在进行事务操作后，会由原来的一致状态，变成另一种一致的状态。

隔离性：每个事务都是彼此独立的，不会受其他事务的执行影响，这意味着一个事务在提交之前，对其他事务是不可见的

持久性：事务提交之后对数据的修改是持久性的，即使在系统出故障或存储介质发生故障，数据的修改依然有效。

因为当事务完成，数据库日志就会更新，这时可通过日志，让系统恢复到最后一次成功的更新状态

在事务中：A 是基础，B 是手段，C 是约束条件，D 是我们的目的

事务的控制语句：1) START TRANSACTION 或 BEGIN 启动一个事务

2) COMMIT 提交事务，提交后，对数据库的修改是永久性的

3) ROLLBACK 或 ROLLBACK TO [SAVEPOINT]

回滚事务，撤销正在进行的所有没提交的修改，或将事务回滚到某保存点

4) SAVEPOINT 在事务中创建保存点，方便后续针对保存点回滚。一个事务中可有多个保存点

5) RELEASE SAVEPOINT 删除某个保存点

6) SET TRANSACTION 设置事务的隔离级别

隐式事务：自动提交，Oracle default 不自动提交，需手写 COMMIT 命令
而 MySQL default 自动提交

MysqL> set autocommit = 0; //关闭自动提交
> set autocommit = 1; //开启自动提交

Completion_type参数，3种可能：1) completion=0，默认情况

执行 COMMIT 的时候会提交事务，在执行下一个事务时，还需要使用 START TRANSACTION OR BEGIN 开启

2) completion=1 当提交事务后，相当于执行 COMMIT AND CHAIN，开启一个连式事务
即提交事务后会开启一个相同隔离级别的事务

3) completion=2

这时 COMMIT = COMMIT AND RELEASE，也就是提交后，会自动与服务器断开连接

事务并发处理，可在在三种异常：

1) 脏读：读到了其他事务还没提交的数据

2) 不可重复读：对某数据进行读取，发现两次读取的结果不同，即没有读到相同的内存（因为有其他事务对这个 data 同时进行了修改或删除）

3) 幻读：ex：事务 A 根据条件查询得到了 N 条数据，但此时事务 B 更改或 add M 条符合事务 A 查询条件的数据，这样当事务 A 再次查询的时候发现会有 N+M 条 data 产生幻读

4 种事务的隔离级别：

1) 读未提交：允许读未提交的数据，此时的查询是不会使用锁的，可能会产生脏读、不可重复读、幻读等

2) 读已提交：只能读已提交的内容，可避免脏读，RDBMS 常见的 default 隔离级别 (Oracle & SQL Server)

if avoid 不可重复读或幻读，需使用带加锁的 SQL 语句

3) 可重复读 (MySQL default 的隔离级别)

保证一个事务在相同查询条件下两次查询结果是一致的，可避免不可重复读和脏

误但无法避免幻读

可串行化: 将事务进行串行化，即在十段列中按顺序执行

↓
最高级别的隔离

	脏读	不可重读	幻读
读未提交 Read Uncommitted	√	√	√
读已提交 Read Committed	X	√	√
可重读 Repeatable read	X	X	√
可串行化 Serializable	X	X	X

为什么DBMS不都 default use 可串行化：会牺牲系统的并发性（性能被牺牲）

隔离通过锁来实现：

1) 行锁：锁定粒度小，发生锁冲突概率低，可实现并发度高，对锁的开销比较大，加锁比较慢，容易出现死锁情况

2) 页锁：锁定的数据比行锁多（一个页中可能有多个行记录）

使用页锁会出现数据浪费

页锁的开销介于表锁和行锁之间，会降低并发度

3) 表锁：对数据表进行锁定，锁定粒度很大

→ 锁冲突概率高，并发度低，锁的使用开销小，加锁快

行锁 表锁 表锁

InnoDB

√ √

MyISAM

√

BDB

√ √

Oracle

√

SQL Server

√ √ √

行锁用的更多一点

锁升级：每个层级的锁数量是有限制的，因锁会占用内存空间，锁空间的大小有限的

当某个层级的锁数量超过上层级别的阈值，就会进行锁升级

→ 用更大粒度的锁替换多个更小粒度的锁。

比如InnoDB中行锁升级为表锁 → 占用的锁空间降低了，但同时并发度也下降了。

按管理划分：

1) 共享锁：锁定的资源可被其他客户端读取，但不能改

读锁

代码：LOCK TABLE name READ;

解锁：UNLOCK TABLE;

对某数据行加锁：SELECT * FROM ... WHERE ... LOCK IN SHARE MODE

写锁

2) 排他锁：锁定的数据只允许进行锁定操作的事务使用，其他事务无法对已锁定的数据查询或修改

LOCK TABLE ... WRITE; UNLOCK TABLE;

对某数据行加锁：SELECT * FROM table_name WHERE ... FOR UPDATE;

3) Intent Lock意向锁：就是给更大级别的空间（表里面是否已上过锁）

↑

共享 排他锁

共享锁会发生死锁的原因：当多个事务对同一数据获得读锁时，这时如果都想对data修改的话，就可能出现死锁

程序员划分锁：

乐观锁：不用每次都对data lock，不用database的lock机制，用程序实现

悲观锁：对数据被其他事务修改持保守态度，会用database本身lock机制实现

乐观锁：适合读操作多，写操作少

悲观锁：适合读操作少，写操作多