零声教育出品 Mark 老师 QQ: 2548898954

索引

> 索引分类：主键索引、唯一索引、普通索引、组合索引、以及全文索引（elasticsearch）；

主键索引

> 非空唯一索引，一个表只有一个主键索引；在 innodb 中，主键索引的 B+ 树包含表数据信息；

```
PRIMARY KEY(key)
```

唯一索引

> 不可以出现相同的值，可以有NULL值；

```
UNIQUE(key)
```

普通索引

> 允许出现相同的索引内容；

```
INDEX(key)
-- OR
KEY(key[,...])
```

组合索引

> 对表上的多个列进行索引

```
INDEX idx(key1,key2[,...]);
UNIQUE(key1,key2[,...]);
PRIMARY KEY(key1,key2[,...]);
```

全文索引

> 将存储在数据库当中的整本书和整篇文章中的任意内容信息查找出来的技术；关键词 FULLTEXT；
>
> 在短字符串中用 `LIKE %`；在全文索引中用 `match` 和 `against`；

主键选择

> `innodb` 中表是索引组织表，每张表有且仅有一个主键；
>
> 1. 如果显示设置 `PRIMARY KEY`，则该设置的key为该表的主键；
> 2. 如果没有显示设置，则从非空唯一索引中选择；
>   1. 只有一个非空唯一索引，则选择该索引为主键；
>   2. 有多个非空唯一索引，则选择声明的第一个为主键；
> 3. 没有非空唯一索引，则自动生成一个 6 字节的 `_rowid` 作为主键；

约束

> 为了实现数据的完整性，对于innodb，提供了以下几种约束，primary key，unique key，foreign key， default, not null；

外键约束（事务性）

> 外键用来关联两个表，来保证参照完整性；MyISAM存储引擎本身并不支持外键，只起到注释作用；而innodb完整支持外键；

```
create table parent (
    id int not null,
    primary key(id)
) engine=innodb;
create table child (
    id int,
    parent_id int,
    foreign key(parent_id) references parent(id) ON DELETE CASCADE ON UPDATE CASCADE
) engine=innodb;
-- 被引用的表为父表，引用的表称为子表；
-- 外键定义时，可以设置行为 ON DELETE 和 ON UPDATE，行为发生时的操作可选择：
-- CASCADE    子表做同样的行为
-- SET NULL   更新子表相应字段为 NULL
-- NO ACTION 父类做相应行为报错
-- RESTRICT   同 NO ACTION
INSERT INTO parent VALUES (1);
INSERT INTO parent VALUES (2);
INSERT INTO child VALUES (10, 1);
INSERT INTO child VALUES (20, 2);
DELETE FROM parent WHERE id = 1;
```

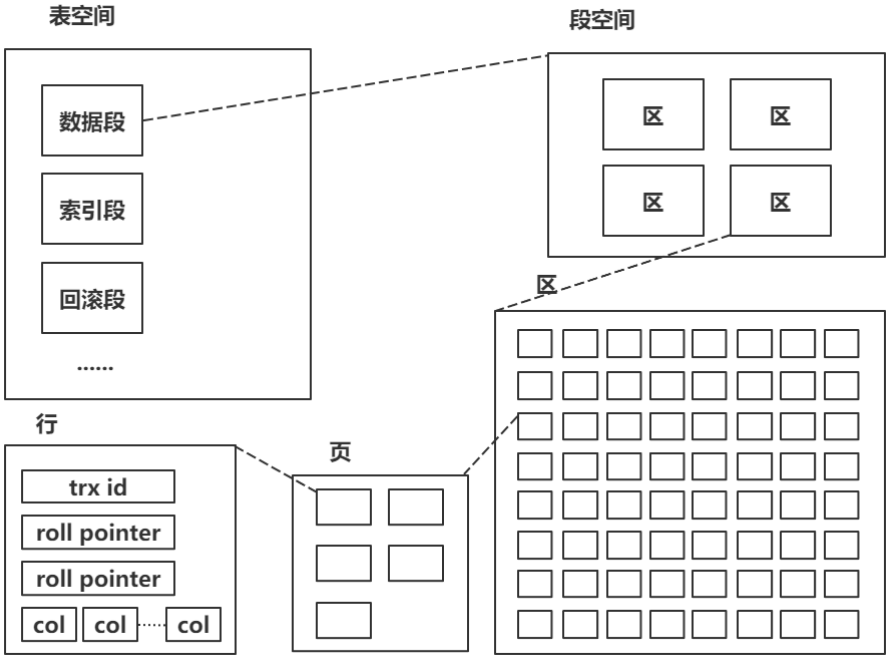
约束与索引的区别

> 创建主键索引或者唯一索引的时候同时创建了相应的约束；但是约束时逻辑上的概念；索引是一个数据结构既包含逻辑的概念也包含物理的存储方式；

索引实现

索引存储

> innodb由段、区、页组成；段分为数据段、索引段、回滚段等；区大小为 1 MB（一个区由64个连续页构成）；页的默认值为16k；页为逻辑页，磁盘物理页大小一般为 4K 或者 8K；为了保证区中的页的连续，存储引擎一般一次从磁盘中申请 4~5 个区；

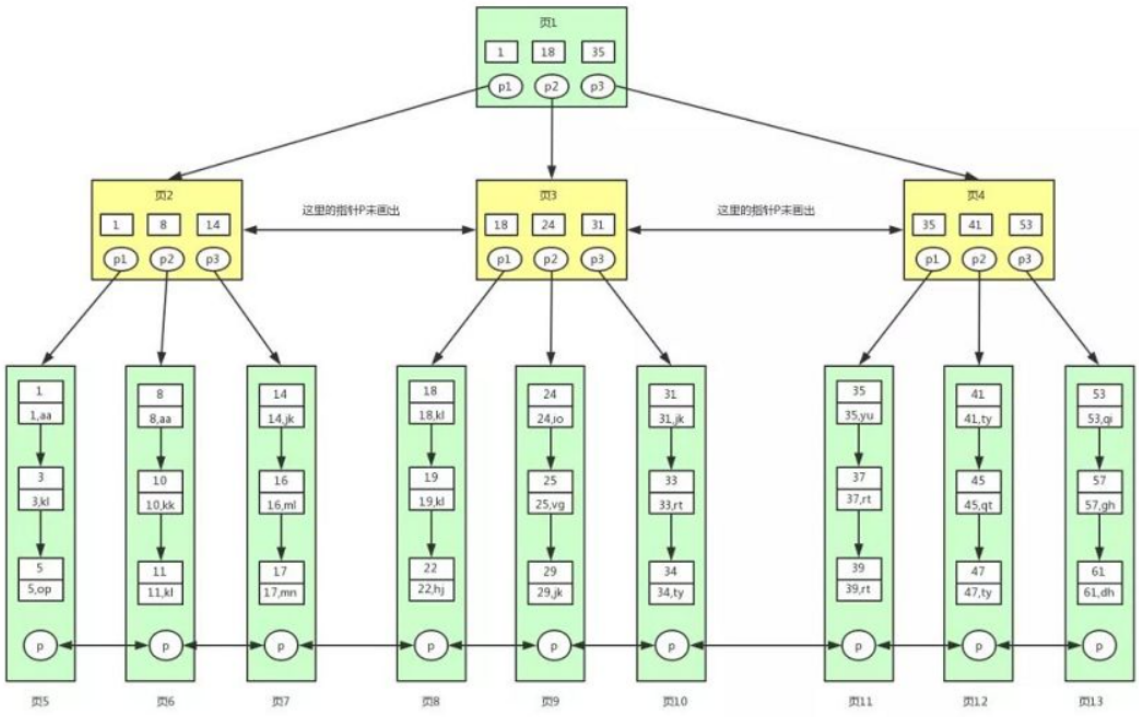


页

> 页是 innodb 磁盘管理的最小单位；默认16k，可通过 `innodb_page_size` 参数来修改；
>
> B+ 树的一个节点的大小就是该页的值；

B+树

> 全称：多路平衡搜索树，减少磁盘访问次数；用来组织磁盘数据，以页为单位，物理磁盘页一般为4K，innodb 默认页大小为 16K；对页的访问是一次磁盘io，缓存中会缓存常访问的页；
>
> 特征：非叶子节点只存储索引信息，叶子节点存储具体数据信息；叶子节点之间互相连接，方便范围查询；
>
> 每个索引对应着一个 B+ 树；



B+树层高问题

> B+树的一个节点对应一个数据页；B+树的层越高，那么要读取到内存的数据页越多，io次数越多；
>
> innodb一个节点16kB；
>
> 假设：
>
> key为10byte且指针大小6byte，假设一行记录的大小为1kB；
>
> 那么一个非叶子节点可存下16kB/16byte=1024个（key+point）；每个叶子节点可存储1024行数据；
>
> 结论：
>
> 2层B+树叶子节点1024个，可容纳最大记录数为： 1024 * 16 = 16384；
>
> 3层B+树叶子节点1024 * 1024，可容纳最大记录数为：1024 * 1024 * 16 = 16777216；
>
> 4层B+数叶子节点1024 * 1024 * 1024，可容纳最大记录数为：1024 * 1024 * 1024 * 16 = 17179869184；

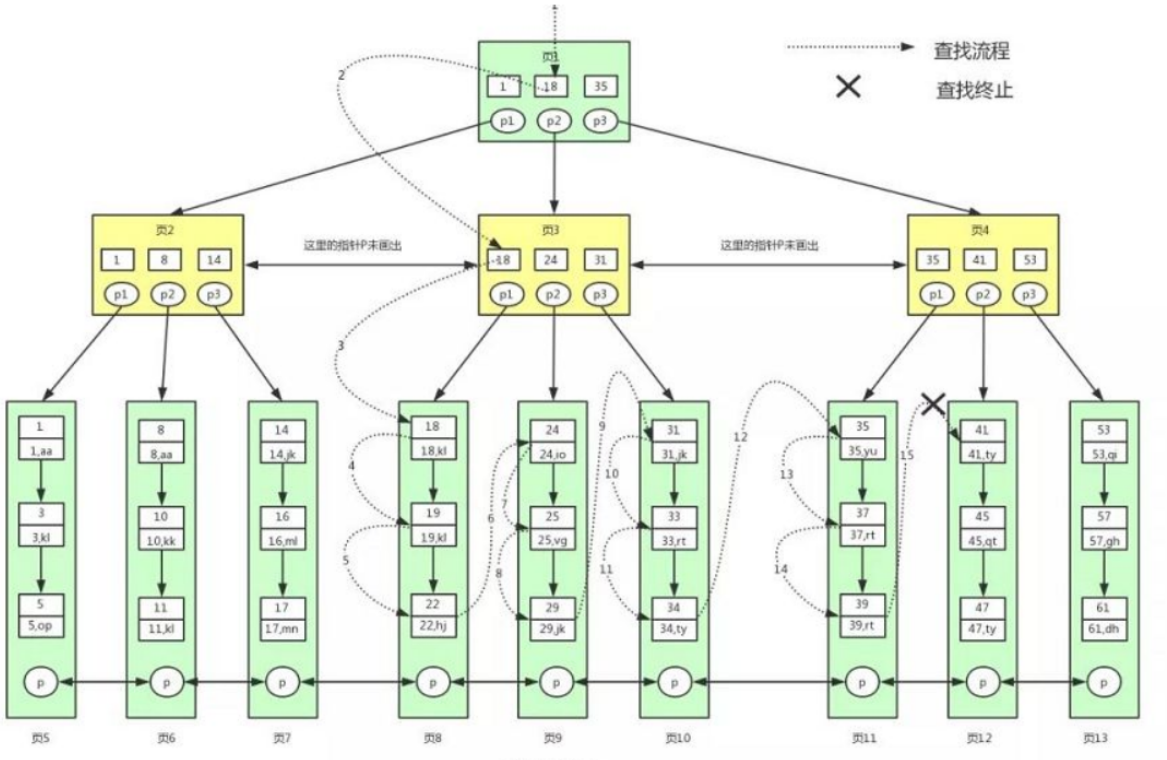
关于自增id

> 超过类型最大值会报错；
>
> 类型 `bigint` 范围：$(-2^{63}, 2^{63}-1)$；
>
> 假设采用 `bigint` 1秒插入1亿条数据，大概需要5849年才会用完索引；

聚集索引

> 按照主键构造的 B+ 树；叶子节点中存放数据页；数据也是索引的一部分；

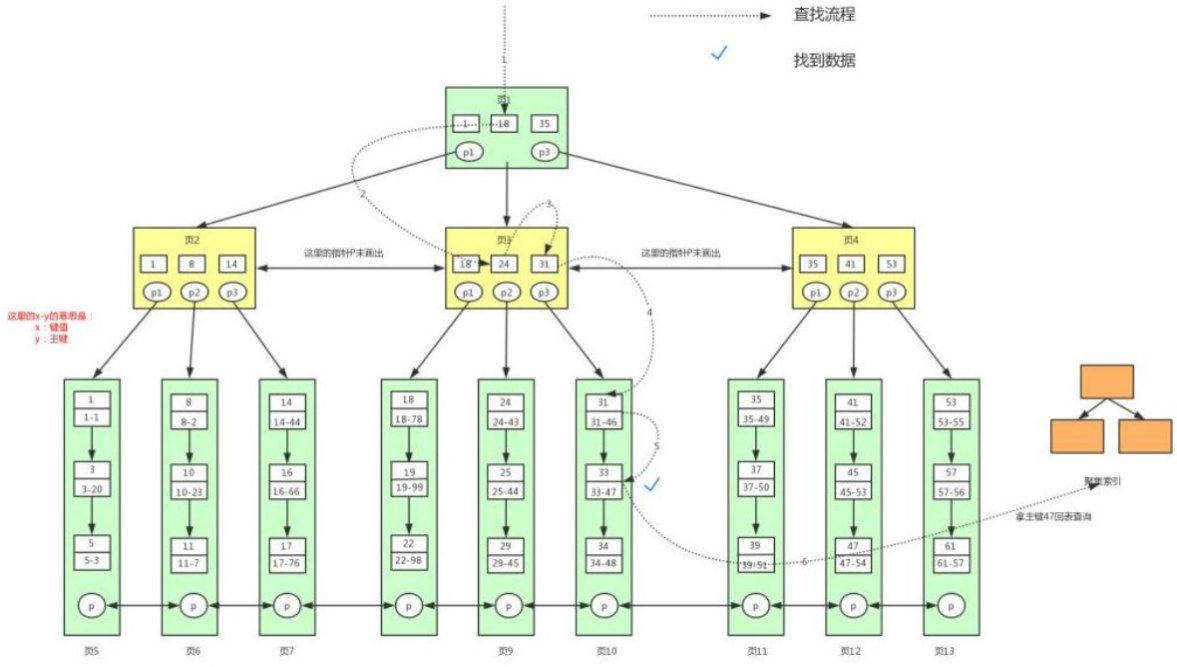
```sql
# table  id  name
select * from user where id >= 18 and id < 40;
```



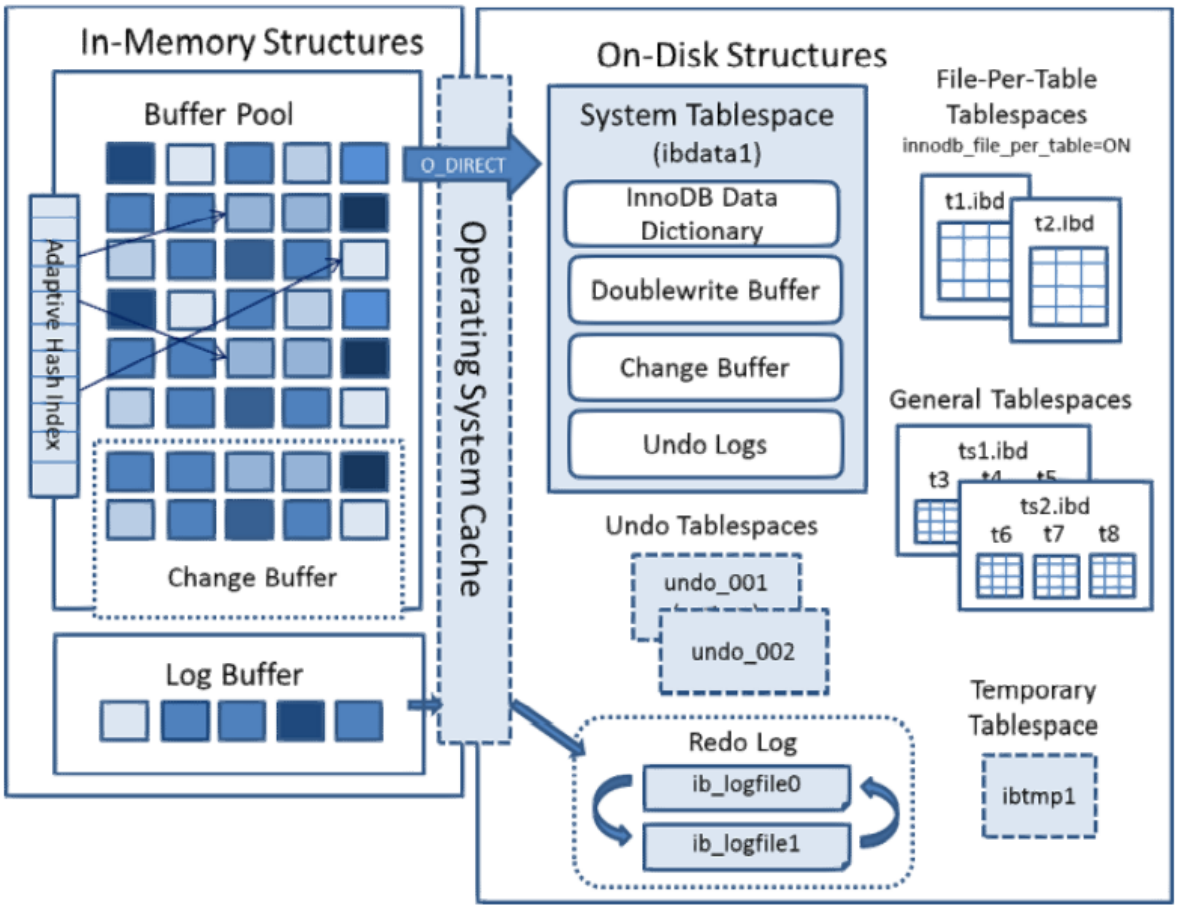
辅助索引

> 叶子节点不包含行记录的全部数据；辅助索引的叶子节点中，除了用来排序的 key 还包含一个 bookmark ； 该书签存储了聚集索引的 key；

```
-- 某个表 包含 id name lockyNum; id是主键，lockyNum存储辅助索引；
select * from user where lockyNum = 33;
```

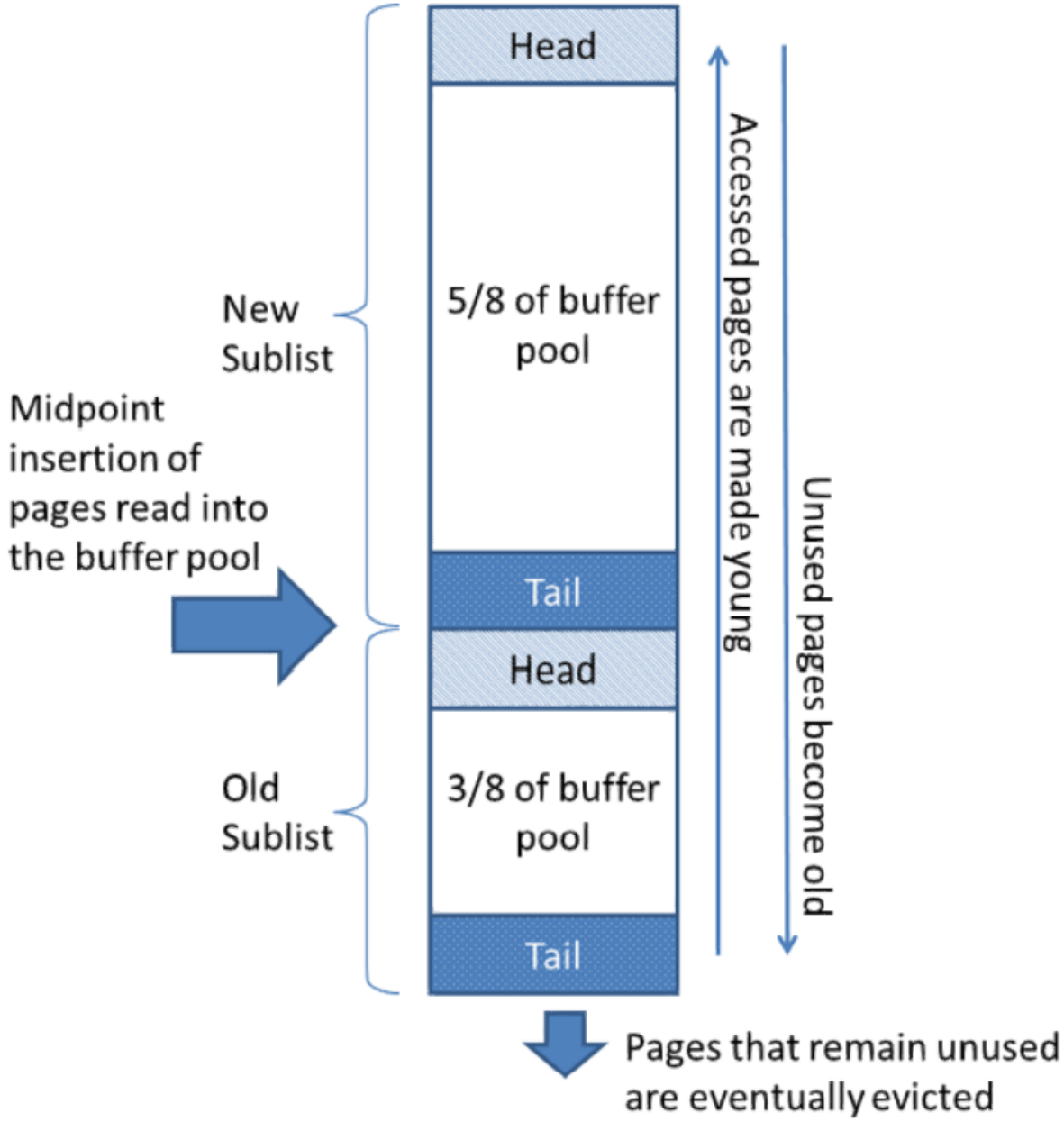


innodb 体系结构



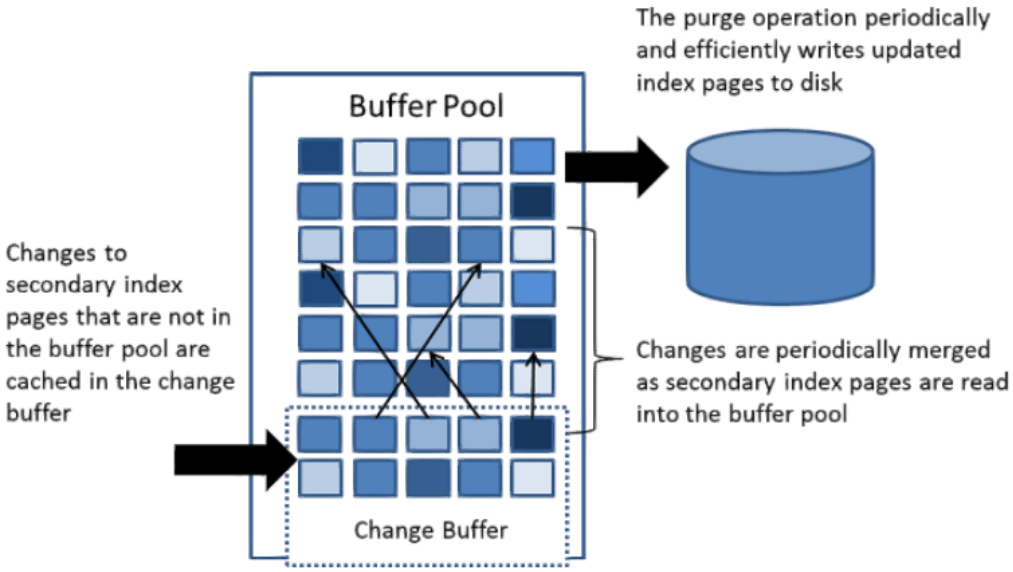
Buffer pool

> Buffer pool 缓存表和索引数据；采用 LRU 算法（原理如下图）让 Buffer pool 只缓存比较热的数据；



Change buffer

> Change buffer 缓存非唯一索引的数据变更（DML操作），Change buffer 中的数据将会异步 merge 到磁盘当中；



最左匹配原则

> 对于组合索引，从左到右依次匹配，遇到 `> < between like` 就停止匹配；

覆盖索引

> 从辅助索引中就能找到数据，而不需通过聚集索引查找；利用辅助索引树高度一般低于聚集索引树；较少磁盘 io；

索引失效

- `select ... where A and B` 若 A 和 B 中有一个不包含索引，则索引失效；
- 索引字段参与运算，则索引失效；例如：`from_unixtime(idx) = '2021-04-30';`
- 索引字段发生隐式转换，则索引失效；例如：`'1'` 隐式转换为 `1`；
- `LIKE` 模糊查询，通配符 `%` 开头，则索引失效；例如：`select * from user where name like '%Mark';`
- 在索引字段上使用 `NOT <> !=` 索引失效；如果判断 `id <> 0` 则修改为 `idx > 0 or idx < 0`；
- 组合索引中，没使用第一列索引，索引失效；
- in + or 索引失效；单独的in 是不会失效的；not in 肯定失效的；

索引原则

- 查询频次较高且数据量大的表建立索引；索引选择使用频次较高，过滤效果好的列或者组合；
- 使用短索引；节点包含的信息多，较少磁盘io操作；比如：smallint，tinyint；
- 对于很长的动态字符串，考虑使用前缀索引；

> 有时候需要索引很长的字符串，这会让索引变的大且慢，通常情况下可以使用某个列开始的部分字符串，这样大大的节约索引空间，从而提高索引效率，但这会降低索引的区分度，索引的区分度是指不重复的索引值和数据表记录总数的比值。索引的区分度越高则查询效率越高，因为区分度更高的索引可以让mysql在查找的时候过滤掉更多的行。对于`BLOB`，`TEXT`，`VARCHAR` 类型的列，必要时使用前缀索引，因为mysql 不允许索引这些列的完整长度，使用该方法的诀窍在于要选择足够长的前缀以保证较高的区分度。

```
select count(distinct left(name,3))/count(*) as sel3,
count(distinct left(name,4))/count(*) as sel4,
count(distinct left(name,5))/count(*) as sel5,
count(distinct left(name,6))/count(*) as sel6,
from user;
alter table user add key(name(4));
-- 注意：前缀索引不能做 order by 和 group by
```

- 对于组合索引，考虑最左侧匹配原则和覆盖索引；
- 尽量选择区分度高的列作为索引；该列的值相同的越少越好；

```
select count(distinct idx) / count(*) from table_name;
-- 或者
show index from student;
*************************** 1. row ***************************
        Table: student
   Non_unique: 0
     Key_name: PRIMARY
  Seq_in_index: 1
  Column_name: id
    Collation: A
  Cardinality: 7
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
1 row in set (0.00 sec)
-- Cardinality 这个值代表 select count(distinct idx) / count(*) from
table_name;
-- 该值决定了优化器的执行计划的选择；
-- 立马更新 Cardinality 值
analyze table student;
-- 在非高峰时间段，对数据库中几张核心表做 analyze table 操作，这能使优化器和索引更
好的为你工作；
```

- 尽量扩展索引，在现有索引的基础上，添加复合索引；最多6个索引
- 不要 `select *`；尽量只列出需要的列字段；方便使用覆盖索引；
- 索引列，列尽量设置为非空；
- 可选：开启自适应 hash 索引或者调整 change buffer；

```
select @@innodb_adaptive_hash_index;
set global innodb_adaptive_hash_index=1; -- 默认是开启的

select @@innodb_change_buffer_max_size;
-- 默认值为25 表示最多使用1/4的缓冲池内存空间 最大值为50
set global innodb_change_buffer_max_size=30;
```

优化器成本分析

> mysql 优化器主要针对 IO 和 CPU 会计算语句的成本；可能不会按照分析的原理来执行语句；

成本分析步骤

- 找出所有可能需要使用到的索引；
- 计算全表扫描的代价；
- 计算不同索引执行查询的代价；
- 对比找出代价最小的执行方案；

SQL优化

> MySQL :: MySQL 5.7 Reference Manual :: 8 Optimization

EXPLAIN

> 用来查看SQL语句的具体执行过程。
>
> 原理：模拟优化器执行 SQL 查询语句，从而知道 mysql 是如何处理 sql 语句的。

执行计划

Column	Meaning
id	The `SELECT` identifier （查询id）
select_type	The `SELECT` type （查询类型）
table	The table for the output row （输出结果集的表）
partitions	The matching partitions （匹配的分区）
type	The join type （表的连接类型）
possible_keys	The possible indexes to choose（可能使用的索引）
key	The index actually chosen （实际使用的索引）
key_len	The length of the chosen key （索引字段的长度）
ref	The columns compared to the index （列与索引的比较）
rows	Estimate of rows to be examined （预估扫描行数）
filtered	Percentage of rows filtered by table condition （按表条件过滤的行百分比）
extra	Additional information （额外信息，如是否使用索引覆盖）

`id`

> select 查询的序列号，包含一组数字，表示查询中执行 select 子句或者操作表的顺序；
>
> id 号分为三种情况：
>
> 1. id 相同，那么执行顺序从上到下；
> 2. id 不同，id 越大越先执行；
> 3. id 有相同的也有不同的，id 相同的按 1 执行，id 不同的按 2 执行；

`select_type`

> 主要用来分辨查询的类型，是普通查询还是联合查询还是子查询

select_type Value	Meaning
SIMPLE	Simple SELECT (not using UNION or subqueries) （简单查询-没有联合查询和子查询）
PRIMARY	Outermost SELECT (最外层select)
UNION	Second or later SELECT statement in a UNION （若第二个select出现在union之后，则被标记为union）
DEPENDENT UNION	Second or later SELECT statement in a UNION, dependent on outer query （union或union all联合而成的结果会受外部表影响）
UNION RESULT	Result of a UNION. （从union表获取结果的select）
SUBQUERY	First SELECT in subquery （在select或者where列表中包含子查询）
DEPENDENT SUBQUERY	First SELECT in subquery, dependent on outer query（subquery的子查询要受到外部表查询的影响）
DERIVED	Derived table（ from子句中出现的子查询，也叫做派生表）
UNCACHEABLE SUBQUERY	A subquery for which the result cannot be cached and must be re-evaluated for each row of the outer query（表示使用子查询的结果不能被缓存）
UNCACHEABLE UNION	The second or later select in a UNION that belongs to an uncacheable subquery (see UNCACHEABLE SUBQUERY)（表示union的查询结果不能被缓存：sql语句未验证）

`table`

> 对应行正在访问哪一个表，表名或者别名，可能是临时表或者union合并结果集;
>
> 1. 具体表名或者表的别名，从具体的物理表中获取数据;
> 2. 表明为derivedN的形式，表示 id 为 N 的查询产生的衍生表;
> 3. 当有union result的时候，表名是union n1,n2等的形式，n1,n2表示参与union的id;

`type`

> type 显示访问类型；采用怎么样的方式来访问数据；效率从好到坏依次为：
>
> system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > range > index > ALL
>
> `ALL`： 全表扫描；如果数据量大则需要进行优化；
>
> `index`：全索引扫描这个比 `ALL` 的效率要好，主要有两种情况，一种是当前的查询时覆盖索引，即我们需要的数据在索引中就可以索取，或者是使用了索引进行排序，这样就避免数据的重排序；
>
> `range`：表示利用索引查询的时候限制了范围，在指定范围内进行查询，这样避免了 `index` 的全索引扫描，适用的操作符： `=`, `<>`, `>`, `>=`, `<`, `<=`, `IS NULL`, `BETWEEN`, `LIKE`, or `IN()`

> `index_subquery`：利用索引来关联子查询，不再扫描全表；
>
> `unique_subquery`：该连接类型类似与 `index_subquery`，使用的是唯一索引；
>
> `index_merge`：在查询过程中需要多个索引组合使用；
>
> `ref_or_null`：对于某个字段即需要关联条件，也需要 `null` 值的情况下，查询优化器会选择这种访问方式；
>
> `ref`：使用了非唯一性索引进行数据的查找；
>
> `eq_ref` ：使用唯一性索引进行数据查找；
>
> `const`：这个表至多有一个匹配行；
>
> `system`：表只有一行记录（等于系统表），这是 `const` 类型的特例；

`possible_keys`

> 查询涉及到字段的索引，则这些索引都会列举出来，但是不一定采纳；

`key`

> 实际使用的索引，如果为 `NULL`，则没有使用索引

`key_len`

> 表示索引中使用的字节数；查询中使用的索引长度；在不损失精度的情况下长度越短越好。

`ref`

> 显示索引的哪一列被使用了，如果可能的话，是一个常数；

`rows`

> 大致估算出找出所需记录需要读取的行数，反映了sql找了多少条数据，该值越小越好。

`extra`

> 额外信息；
>
> `using filesort`：使用了文件排序；
>
> `using temporary`：建立临时表来保存中间结果，查询完成之后把临时表删除；
>
> `using index`：采用覆盖索引，直接从索引中读取数据，而不用访问数据表。如果同时出现 `using where` 表明索引被用来执行索引键值的查找；如果没有，表明索引被用来读取数据，而不是真的查找；
>
> `using index condition`：采用索引下推，减少回表次数；
>
> `using where`：使用where进行条件过滤；
>
> `using join buffer`：使用连接缓存；
>
> `impossible where`：`where` 语句的结果总是 `false`；

优化器选择过程

> 优化器根据解析树可能会生成多个执行计划，然后选择最优的的执行计划；

```
SHOW VARIABLES LIKE 'optimizer_trace';
-- 启用优化器的追踪
SET optimizer_trace='enabled=on';
-- 执行一条查询语句
SELECT * FROM information_schema.optimizer_trace;
-- 用完关闭
SET optimizer_trace="enabled=off";
SHOW VARIABLES LIKE 'optimizer_trace';
```

慢日志查询

详情

> MySQL :: MySQL 5.7 Reference Manual :: 5.4.5 The Slow Query Log

开启

> 查看
>
> ```
> SHOW GLOBAL VARIABLES LIKE 'slow_query%';
> SHOW GLOBAL VARIABLES LIKE 'long_query%';
> ```
>
> 设置
>
> ```
> SET GLOBAL slow_query_log = ON;  -- on 开启 off 关闭
> SET GLOBAL long_query_time = 4;   -- 单位秒；默认10s；此时设置为4s
> ```
>
> 或者修改配置
>
> ```
> slow_query_log = ON
> long_query_time = 4
> slow_query_log_file = D:/mysql/mysql57-slow.log
> ```

mysqldumpslow

> 查找最近10条慢查询日志
>
> ```
> mysqldumpslow -s t -t 10 -g 'select' D:/mysql/mysql57-slow.log
> ```

SHOW PROFILE

```
# 查看是否开启
SELECT @@profiling;
# 设置开启
SET profiling = 1;
# 查看所有 profiles
show profiles;
# 查看query id 为 10 那条查询
show profile for query 10;
# 查看最后一条查询
show profile;
# 最后关闭
SET profiling = 0;
```

SHOW PROCESSLIST

> 查看连接线程；可以查看此时线上运行的 sql 语句；
>
> 如果要查看完整的SQL语句：SHOW FULL PROCESSLIST; 然后优化该语句；