DB的查漏补缺

MySQL体系结构

MySQL服务端

- **连接层**：负责权限校验，连接处理等
- **服务层**：负责SQL分析和优化，SQL接口等
- **引擎层**：可插拔的存储引擎层，不同的**存储引擎**有不同的功能，索引是在这个层实现的
- **存储层**：将数据存储在磁盘中，并完成与存储引擎的交互

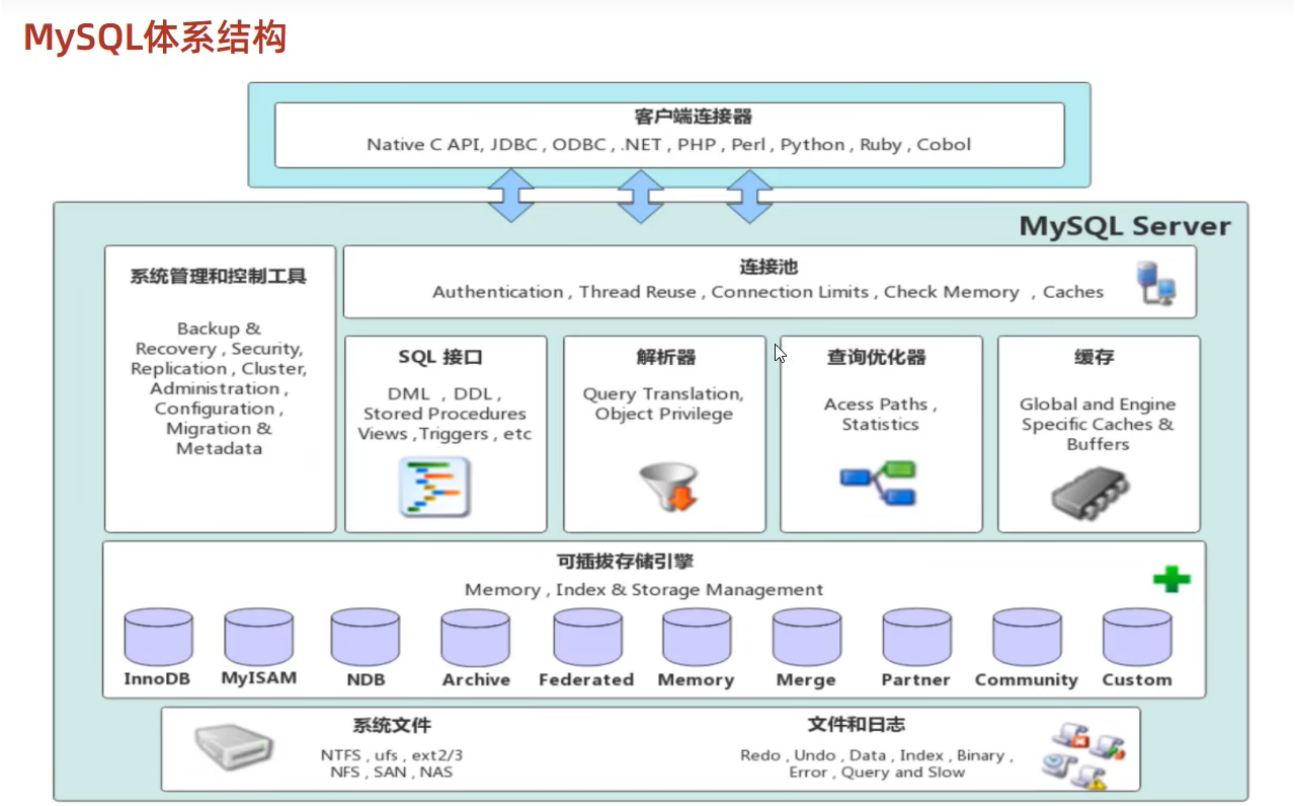


image-20260117170137026

InnoDB

MySQL默认使用的**存储引擎**，它提供

- **事务**
- **行级锁**
- **外键**
- 索引
- 高并发读写

等功能

特点

- 存储引擎是基于表的，**不同的表可以有不同的存储引擎**
- InnoDB引擎的每张表都对应一个表空间文件 `.ibd` ，存储表**结构、数据、索引**

常用语法

```
# 查看存储引擎
show engines;

# 建表时指定存储引擎
create table tb_name(...) engine = Innodb;
```

逻辑存储结构

主要包括

- `TableSpace`：表空间，`.idb`文件，用于存储记录、索引等
- `Segment`：段，分为数据段、索引段、回滚段
  - InnoDB是索引组织表，**数据段**就是B+树叶子节点，**索引段**就是B+树非叶子节点
  - 段用来管理多个区
- `Extent`：区，大小是1M，一个区默认包含64个连续的页
  - InnoDB存储引擎每次从磁盘申请4-5个区，**保证申请到的页是连续的**
- `Page`：页，**磁盘操作的最小单元**，大小是16KB
- `Row`：行
  - `Trx_id`：最后一次操作事务的id
  - `Roll_pointer`：指针，指向`undo log`里的一条`undo`记录

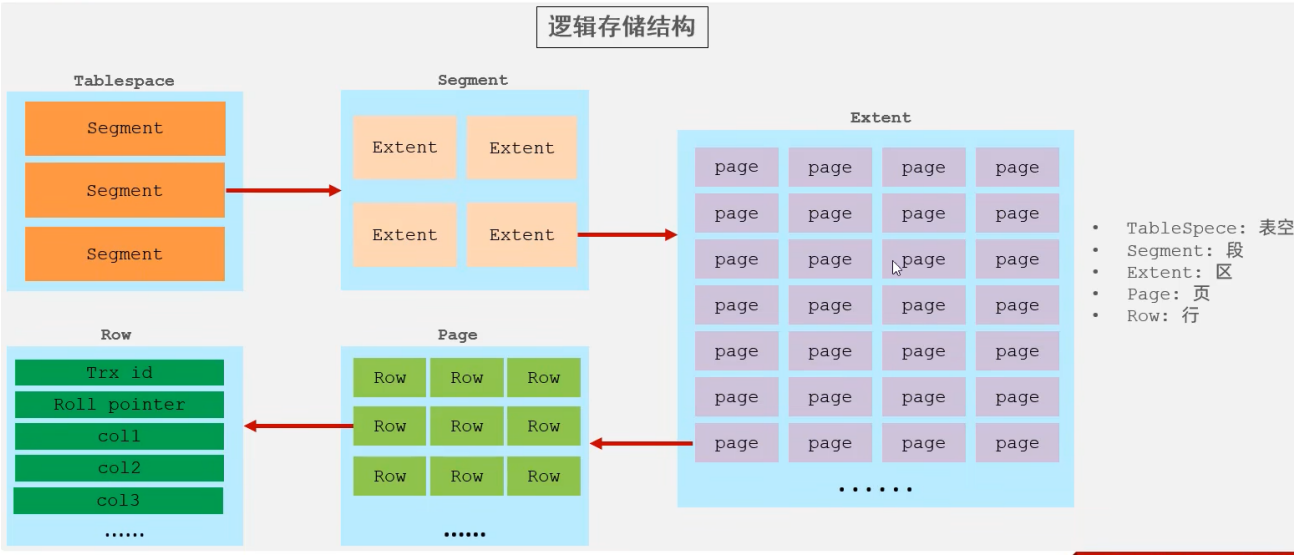


image-20260117171642309

常见存储引擎区别

MyISAM

- 不支持事务、外键
- 不支持行锁
- 访问速度快
- `.sdi`存储表结构，`.MYD`存储数据，`.MYI`存储索引

Memory

- 数据存放在内存中，适合做临时表
- 访问速度快
- 使用hash索引
- 只有 `.sdi` 文件，存放表结构

特点	InnoDB	MyISAM	Memory
事务	支持	不支持	不支持
锁	行锁	表锁	表锁
外键	支持	不支持	不支持

架构

左侧为**内存架构**，右侧为**磁盘架构**

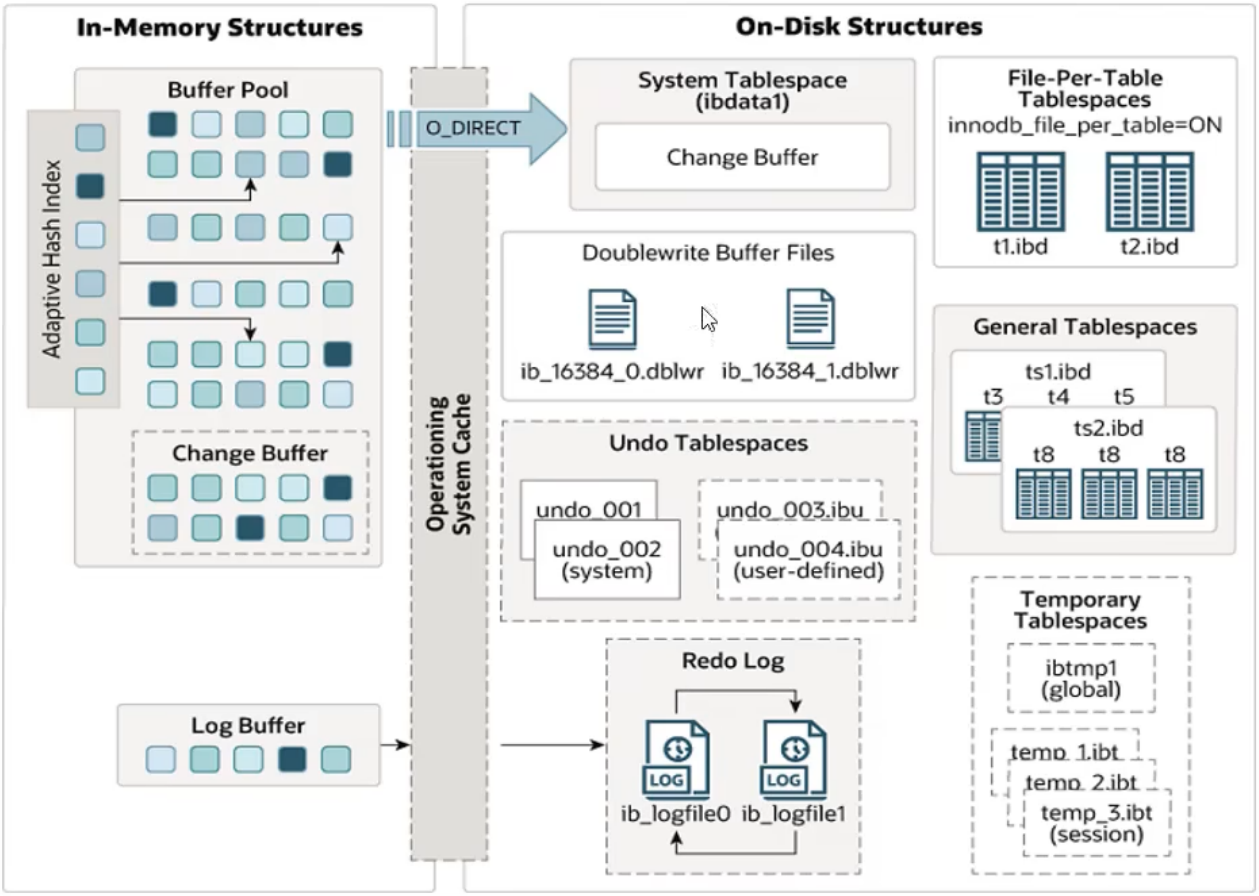


image-20260117192823578

内存架构

`Buffer Pool`：缓冲池，是**主内存的一个区域**，可以缓存**磁盘上的真实数据**

- 执行增删改查操作时，**先操作缓冲池的数据**（缓冲池无数据，则从磁盘加载并缓存）
- 会以**一定频率把缓冲池的数据刷新到磁盘**，减少磁盘IO，加快处理速度
- **以页为单位，采用链表的数据结构管理页**，有三种类型的页

    - `free page`：空闲页，未被使用
    - `clean page`：被使用的页，数据未被修改
    - `dirty page`：脏页，被使用的页，数据被修改过，数据与磁盘的数据产生不一致

`Change Buffer`：更改缓冲区

- **针对非聚簇索引**执行增删改语句且这些数据没有在`Buffer Pool`的时候，会先把**数据变更**缓存在`Change Buffer`中
- 未来数据被读取时，再将数据合并恢复到`Buffer Pool`中，再将合并后的数据刷新到磁盘中
- **存在的意义**：增删改数据时，在修改聚簇索引的同时还要修改非聚簇索引，但是非聚簇索引的目标页可能不在`Buffer Pool`里，如果每变更一下数据就去磁盘找非聚簇索引的目标页进行更改，就会造成大量的磁盘IO。`Change Buffer`使用**延迟合并换取更少的磁盘IO**，提升效率

`Log Buffer`：日志缓冲区，用来保存要写入到磁盘中的日志数据

- 主要是`redo log buffer`
- 默认大小为16MB，数据会定期刷新到磁盘中
- 参数：
    - `innodb_log_buffer_size`：缓冲区大小
    - `innodb_flush_log_at_trx_commit`：日志刷新到磁盘时机
        - 0：每秒将日志写入并刷新到磁盘一次
        - 1：日志在每次事务提交时写入并刷新到磁盘
        - 2：日志在每次事务提交后写入，并每秒刷新到磁盘一次

`Adaptive Hash Index`：自适应哈希索引，用于优化对`Buffer Pool`数据的查询

- 如果MySQL观察到`hash`索引可以提升速度，则建立`hash`索引
- 无需人工干预
- 适用于等值查询
- 参数：`adaptive_hash_index`

磁盘架构

`System Tablespace`：系统表空间，存储更改缓冲区的数据，**它可能包含表和索引数据**

- `innodb_data_file_path`：存储路径

`File-Per-Table-Tablespaces`：每个表文件的表空间，存放单个`InnoDB`表的数据和索引，存储在文件系统的单个数据文件

- `innodb_file_per_table`：是否开启

`General Tablespaces`：通用表空间，需要自己创建，可以指定特定的表存在这个表空间

`Undo Tablespaces`：撤销表空间，用于存储`undo log`日志

`Temporary Tablespaces`：临时表空间，用于存储用户创建的临时表

`Doublewrite Buffer Files`：双写缓冲区，InnoDB将数据页从`Buffer Pool`刷到磁盘前，会先把数据写入到双写缓冲区中

- 为`.dblwr`文件

`Redo Log`：重做日志，实现**事务的持久性**

- 由重做日志缓冲区（在内存中）和重做日志文件（在磁盘中）组成
- 事务提交之后会把所有修改信息存到该日志中，可以用于数据恢复

后台线程

作用：把缓冲池的数据刷新到磁盘中

包括

- `Master Thread`：核心后台线程，负责调度其它线程，将缓冲池数据异步刷新到磁盘，脏页刷新，合并插入缓存，undo页的回收
- `IO Thread`：Innodb使用异步非阻塞IO处理IO请求，`IO Thread`负责这些IO请求的回调
  - `Read Thread`：负责读操作
  - `Write Thread`：负责写操作
  - `Log Thread`：负责将日志缓冲区刷新到磁盘
  - `Insert Buffer Thread`：负责将写缓冲区内容刷新到磁盘
- `Purge Thread`：用于回收事务已经提交的`undo log`
- `Page Cleaner Thread`：协助`Master Thread`把刷新脏页到磁盘的线程

事务原理

`redo log`和`undo log`保证事务的**原子性、一致性、持久性**

- `redo log`：重做日志，保证事务的**持久性**，记录**数据页的物理修改**
  - 改`buffer pool`的页，然后把数据变更的的情况写入`redo log buffer`，再按一定策略**顺序写入**磁盘的`redo log files`
  - 可以避免因宕机导致内存的数据丢失而无法保证数据持久性
- `undo log`：回滚日志，保证事务的**原子性**
  - **作用**：提供回滚和MVCC
  - 是逻辑日志，描述的是如何恢复一行到旧状态
  - 进行`insert`时，产生的`undo log`日志在事务被提交后**可被立刻删除**
  - 进行`update`、`delete`的时候，产生的`undo log`日志在回滚时需要，在快照读时也需要，**不会被立刻删除**
- `redo log`和`undo log`保证事务的**一致性**

锁和MVCC保证事务的**隔离性**

MVCC

多版本并发控制：Multi-Version Concurrency Control，维护一个数据的多个版本，使读写操作没有冲突

- 具体实现依赖于数据库记录中的**三个隐式字段、**`undo log`**版本链、**`readView`
- 隐式字段：`DB_TRX_ID`、`DB_ROLL_PTR`、`DB_ROW_ID`
  - `DB_TRX_ID`：最近修改的事务ID
  - `DB_ROLL_PTR`：回滚指针，指向上一个版本
  - `DB_ROW_ID`：隐式主键，表结构无主键，就会生成该字段

基本概念

- 当前读：读取的是记录的最新版本，读取时要保证其它并发事务不能修改读取记录（否则造成不可重复读），会对**读取的事务进行加锁**
- 快照读：简单的`select`，不加锁，读取的是记录数据的可见版本，可能是历史数据

不同隔离级别的区别

- 读已提交：每次查询都生成一个快照读
- 可重复读：开启事务后第一个`select`语句才是快照读的地方

- 串行化：快照读退化成当前读

实现原理

- `undo log` **版本链**：不同事务或相同事务对同一条记录进行修改，会导致该记录的 `undolog` 生成一条记录版本链表，链表头部是最新的旧纪录，尾部是最老的旧纪录
- `ReadView` ：读视图，是**快照读SQL执行**时MVCC提取数据的依据，记录并维护系统当前**活跃的未提交的事务ID**、
  - 包含四个**核心字段**
    - `m_ids` ：创建快照那一刻，活跃的未提交的事务ID集合
    - `min_trx_id` ： `m_ids` 里的最小值
    - `max_trx_id` ：预分配事务ID， `m_ids` 里的最大事务ID + 1
    - `creator_trx_id` ： `ReadView` 创建者的事务ID
  - **版本链数据访问规则**， `trx_id` 代表某个行版本的创建/最后修改事务ID
    - `trx_id == creator_trx_id` ：可以访问该版本，说明数据是当前这个事务更改的
    - `trx_id < min_trx_id` ：可以访问该版本，说明数据已经提交
    - `trx_id > max_trx_id` ：不可以访问该版本，当前事务是在 `ReadView` 生成之后开启
    - `min_trx_id <= trx_id <= max_trx_id`
      - `trx_id` 在 `m_ids` 里，不可访问该版本， `trx_id` 还未提交
      - `trx_id` 不在 `m_ids` 里，可访问该版本， `trx_id` 在快照创建前已经提交
  - **RC（读已提交）**：每次执行快照读都会生成 `ReadView`
  - **RR（可重复读）**：仅在事务中第一次执行快照读生成 `ReadView` ，后续复用该 `ReadView`

索引

帮助MySQL高效获取数据的**有序数据结构**，使用 `.MYI` 文件**存放索引**

优缺点

- **优点**：提高查询效率，提高排序效率
- **缺点**：占用磁盘空间，降低数据的增删改效率

索引结构

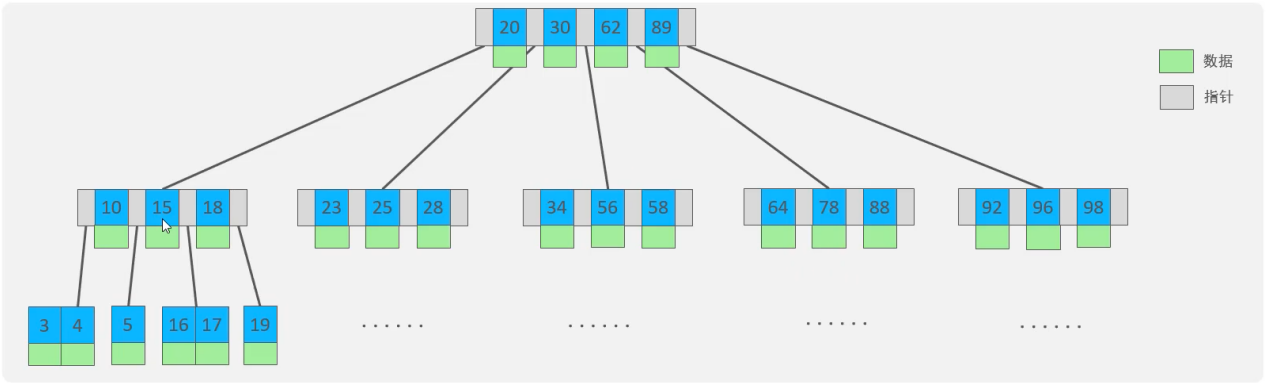
MySQL的索引是在**存储引擎层**实现的，不同的存储引擎有不同的索引结构，如：

- **B+Tree索引**：最常见索引类型
- **Hash索引**：使用哈希表实现，不支持范围查询
- **R-tree索引（空间索引）**：用于地理空间数据类型
- **Full-Text（全文索引）**：建立倒排索引，常用于elasticsearch

B树与B+树

B-Tree：**多路**平衡查找树

- 每一个节点可以存储 `n` 个key，那么就会有5个指针指向子节点
- **核心**：中间元素向上分裂
- 一个节点是一页，16KB
- 叶子节点和非叶子节点都会存储数据，导致一页中存储的键值减少，指针也减少，导致树的高度增加



知识小贴士: 树的度数指的是一个节点的子节点个数。

image-20260116134036265

B+Tree

- 非叶子节点主要起到索引的作用
- 所有的数据都会出现在叶子节点，叶子节点形成一个单向链表

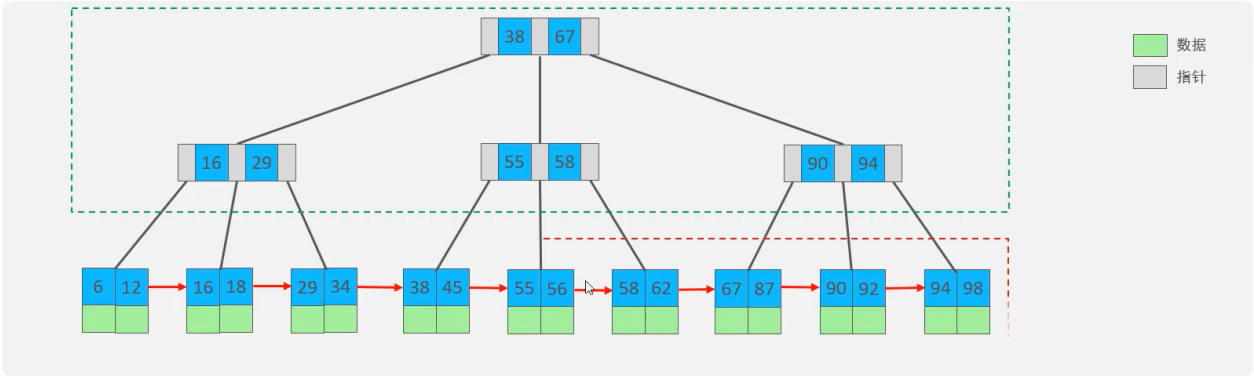


image-20260116134013876

MySQL中在原B+树基础上增加一个**指向相邻叶子节点的链表指针**

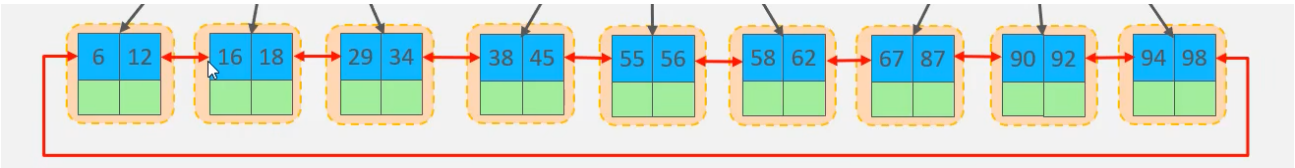


image-20260116134539535

索引分类

有四种分类：

- **主键索引**：只能有一个，关键字为 `PRIMARY`
- **唯一索引**：可以有多个，关键字为 `UNIQUE`
- **常规索引**：可以有多个
- **全文索引**：可以有多个，关键字为 `FULLTEXT`

聚簇索引与非聚簇索引

也叫做**聚焦索引**和**二级索引**

- **聚簇索引**：数据存储与索引放在一起，索引结构的叶子节点存放了行数据，**必须有且只有一个**
  - 本质上就是表本身，而主键就是聚簇索引键
- **非聚簇索引**：数据与索引分开存储，索引结构的叶子节点关联的是对应的主键
  - 额外的独立B+树，索引键是**手动指定**的

聚簇索引选取规则

- 如果存在主键，**主键索引**就是聚簇索引
- 如果不存在主键，则使用**第一个唯一索引**作为聚簇索引
- 如果表没有主键也没有合适的唯一索引，InnoDB会自动生成一个 `rowid` 作为**隐藏的聚簇索引**

回表查询：先走二级查询（非聚簇索引查询）拿到主键值，再根据主键值到聚簇索引中拿到数据的所有信息

索引语法

创建索引

- 单列索引：一个索引关联一个字段
- 联合索引：一个索引关联多个字段，用多个字段来创建B+树

```
-- unique|fulltext: 确定索引类型为unique或者fulltext
-- index_name: 索引名称
-- table_name: 表名
-- index_col_name: 字段名
create [unique|fulltext] index index_name on table_name (index_col_name,...);
```

查看索引

```
show index from table_name;
```

删除索引

```
drop index index_name on table_name;
```

联合索引的最左匹配原则

联合索引：一个索引关联了多个字段

最左匹配法则：查询从索引的最左列开始，并且**不跳过**索引中的列，如果跳过某一列，后面的**字段索引失效**

```sql
-- 联合索引: (name,age,school)

select * from tb_name where name = "wjh" and age = 30 and school = "ecust"; -- 不失效

select * from tb_name where name = "wjh" and school = "ecust"; -- 只用了name列的索引，school列索引失效

select * from tb_name where age = 30 and school = "ecust"; -- 不失效
```

范围查询：联合索引中出现范围查询(`>,<`)，**范围查询右侧的索引失效**

- 最好使用 `>=` 或者 `<=`

失效与否是看**联合索引中列的先后顺序**

```sql
-- 联合索引: (name,age,school) 会失效
-- 联合索引: (name,school,age) 不会失效
select * from tb_name where name = "wjh" and age > 30 and school = "ecust";
```

索引失效

导致索引失效的情况

- 在索引列上进行运算操作
- 字符串类型字段使用不加引号
- 头部模糊匹配（尾部模糊匹配**不会**导致索引失效）
- 使用 `or` 进行连接，一侧有索引，一侧无索引
- MySQL评估**走索引比全表扫描慢**

SQL提示

当一个列被多个索引绑定时，可以使用SQL提示指定要使用哪个索引

use index：使用指定索引，MySQL可能会拒绝使用指定索引

```sql
-- index_name: 索引名字
explain select * from tb_name use index(index_name)
```

ignore index：忽略指定索引

```sql
-- index_name: 索引名字
explain select * from tb_name ignore index(index_name)
```

force index：强制使用指定索引

```sql
-- index_name: 索引名字
explain select * from tb_name force index(index_name)
```

索引使用

覆盖索引：当索引包含了查询所需要的所有字段，**数据库引擎就可以直接从索引中获取数据而无需回表查询**

- `using index condition`：查找使用了索引，但需要做**回表查询**
- `using where;using index`：查找使用了索引，但是需要的数据在索引列中都可以找到，**性能高**

前缀索引：取长字符串的**一部分前缀**建立索引，可以降低索引体积，节约空间

- **语法**

```sql
-- idx_name: 索引名称
-- column: 列名
-- n: 取多少个字符
create index idx_name on tb_name(column(n));
```

- **前缀长度**：可以根据索引的选择性来确定
  - 选择性：不重复的索引值和数据表的记录总数的比值，选择性越高则查询效率越高
  - 实际业务场景需要在**前缀长度和选择性之间做平衡和取舍**

索引设计原则

1. 针对数据量较大（超过100万条），且查询比较频繁的表建立索引
2. 对常作为查询条件（where）、排序（order by）、分组（group by）操作的字段建立索引
3. 选择区分度高的列作为索引，尽量建立**唯一索引**
4. 字符串类型的字段若长度较长，建立前缀索引
5. 尽量使用联合索引，减少单列索引，因为可避免回表查询
6. 控制索引数量，索引太多会影响增删改的效率
7. 如果列不存储NULL值，在创建表时使用 `NOT NULL` 进行约束，可以提升优化器查询的效率

SQL性能分析

查看执行频次

使用下列语句得到SQL语句的执行频次

```sql
# 跟7个下划线
show global status like 'Com_______'
```

慢查询日志

在 `/etc/my.cnf` 进行配置

```
# 开启慢查询日志开关
show_query_log=1

# 设置慢查询的时间为2s，SQL语句执行时间超过2s就会被视为是慢查询
long_query_time=2
```

profile详情

使用 `show profiles` 命令帮助了解每条SQL命令的耗时情况

使用以下命令可以查看当前MySQL是否支持 `profile` 操作

```
select @@have_profiling;
```

默认 `profiling` 是关闭的，可以使用 `set` 命令打开

```
set profiling=1;
```

常见命令

```
# 查看每条SQL命令耗时情况
show profilies;

# 查看指定query_id的SQL语句各个阶段耗时情况
show profile for query query_id;

# 查看指定query_id的SQL语句CPU使用情况
show profile cpu for query query_id;
```

explain执行计划

使用 `explain/desc` 可以获取MySQL如何执行 `select` 语句的信息

```
# 直接在select语句之前加上关键字explain/desc
explain select * from tb_name;
```

执行计划详情

- `id` ： `select` 查询的序列号，表示查询中执行 `select` 子句或者操作表的顺序
  - id相同，执行顺序从上到下
  - id不同，值越大越先执行
- `select_type` ： 表示 `select` 的类型

- `type`：表示连接类型，性能**由好到差**分别为 `NULL、system、const、eq_ref、ref、range、index、all`
- `possible_key`：可能用到的索引
- `key`：实际用到的索引，如果为NULL则没有使用索引
- `key_len`：索引中使用的字节数，值为索引字段最大可能长度，保证精度的同时长度越短越好
- `rows`：MySQL认为必须要执行查询的行数，是估计值
- `filtered`：返回结果的行数占需读取函数的百分比，值**越大越好**

SQL优化

insert优化

- **执行批量插入**
  - 一次插入不要超过1000条
- **手动提交事务**
- **主键顺序插入**

如果一次性需要插入大批量数据，使用 `load` 指令进行插入

```
# 客户端连接服务端时，加上参数 --local-infile
mysql --local-infile -u root -p

# 设置全局参数local_infile为1，开启从本地加载文件导入数据的开关
set global local_infile=1;

# 执行load指令将准备好的数据加载到表结构中
-- local_file: 本地文件路径
-- tb_name: 表名
-- ,: 每一个字段用逗号分隔
-- .: 每一行用'.'分隔
load data local infile 'local_file' into table 'tb_name' fields terminated by ','
lines terminated by '\n';
```

主键优化

索引组织表：在InnoDB中，表数据是根据**主键顺序**组织存放的

页分裂

- 主键乱序插入时会导致页分裂
- 开启新的数据页，找到第一个数据页百分之50的位置，将这个位置后面的数据放到新的数据页，并且把当前插入的数据也放到这个新的数据页，同时更替指针

页合并

- 当删除一行记录时，实际上记录并没有被物理删除，只是进行逻辑删除，它的空间**允许被其它记录声明使用**
- 当页中删除的记录到达 `MERGE_THERESHOLD`（默认为50%）时，InnoDB会寻找最近的页进行合并

优化原则

- 尽量降低主键长度

- 尽量选择顺序插入，选择使用 `AUTO_INCREMENT` 自增主键
- 不要选择UUID或者其它自然主键
- 避免对主键修改

order by优化

有两种排序方法

- `Using filesort` ：通过表的索引或全表扫描获取符合条件的数据行，在**排序缓冲区** `sort buffer` 完成排序操作。所有**不是通过索引直接返回排序结果**的排序叫做 `FileSort` 排序
  - 排序缓冲区大小默认为256K，不可避免排序的时候可以增大缓冲区大小
- `Using index` ：通过有序索引顺序扫描直接返回有序数据，不需要额外排序，效率高

几种会**出现** `FileSort` **的情况**

- 查询时的条件违背最左前缀法则
- 没有对查询时指定的字段建立索引
- 索引默认是升序排列，查询时对多个字段分别进行 `asc` 、 `desc` ，此时可以创建对应的索引优化掉 `FileSort`

```
create index idx_name on tb_name (age asc,name desc);
```

- 没有使用覆盖索引造成回表查询

group by优化

- `Using temporary` ：使用临时表来分组，效率较低
- `Using index` ：直接通过索引返回分组数据，效率较高

注意

- 分组操作可以建立索引来提高分组效率
- 分组操作时，索引使用满足最左前缀法则

limit优化

在大数据量的情况下， `limit` 分页越往后效率越低

解决方法：覆盖索引+子查询

count优化

count的几种用法

- `count(*)` ：统计所有行数（包括NULL），InnoDB专门**对此做了优化**
- `count(主键)` ：有几个主键结果就是多少
- `count(字段)` ：统计指定列**非NULL**值的行数
- `count(1)` ：统计所有行数（包括NULL）

效率排序： `count(字段) < count(主键) < count(1) = count(*)`

update优化

InnoDB的行锁是**针对索引**加的锁，如果索引失效或者匹配条件（where）的字段无索引，**行锁会升级成表锁**，降低并发性能

事务

`autocommit`：是否为自动提交，1为自动提交事务，0为手动提交事务

ACID：原子性、一致性、隔离性、持久性

注意

- 修改数据但未提交，修改后的数据会出现在内存中
- 提交之后，修改的数据会被刷到磁盘

并发事务问题

脏读

事务A读到了事务B**尚未提交的修改**

不可重复读

一个事务先后读取到同一条记录，但**两次读取的数据不同**，通常是因为别的事务在中间提交了`UPDATE/DELETE`

幻读

在同一事务中，多次执行相同的查询，但**返回的结果集的行数发生变化**，通常是因为别的事务在中间提交了`INSERT/DELETE`

事务隔离级别

从上到下**性能依次降低，数据安全性依次提高**

RU（读未提交）

`Read uncommitted`：无法解决并发事务问题

RC（读已提交）

`Read committed`：可以解决脏读

RR（可重复读）

`Repeatable Read`：可以解决不可重复读、脏读

串行化

`Serializable`：可以解决脏读、不可重复读、幻读

常用语法

```
# 查看事务隔离级别
select @@transaction_isolation;

# 设置事务隔离级别
set [session|global] transaction isolation level [read uncommitted|read
committed|repeatable read| serializable]
```

锁

定义：计算机协调多个进程或线程并发访问某一资源的机制

全局锁

锁定**数据库中的所有表**

- 对数据库实例加锁，导致实例只处于**可读状态**
- 常用在**数据备份业务场景**中，可以保证数据的一致性，保证数据完整

语法

```
# 给数据库加上全局锁
flush tables with read lock;

# 进行数据备份
-- -h: 数据库所在的主机地址
-- uroot: 登录的用户名
-- -p: 密码
-- dbName: 要进行备份的数据库
-- fileName: 备份到磁盘的文件名称
mysqldump -h host -uroot -p pwd dbName > fileName.sql

# 解锁
unlock tables;
```

存在的问题

- 在主库进行备份，需要加锁，但会导致**业务停摆**
- 在从库进行备份，备份期间从库不能执行主库同步过来的二进制日志，导致**主从延迟**

特点

在InnoDB引擎下加上 `--single-transaction` 可以实现不加锁的数据一致性备份

- 原理：基于MVCC机制的一致性快照
- 开启事务，创建一致性快照，实现数据备份
- 进行数据备份时，业务可正常执行，但备份的数据是事务开始时的一致性快照

表级锁

每次操作锁住**整张表**

主要分为**三类**

- 表锁
- 元数据锁
- 意向锁

表锁

保护整张表的所有数据

可分为两类

- **表共享读锁**（读锁/共享锁）：加锁的客户端只能读数据，不能写数据，其他客户端也如此
- **表独占写锁**（写锁/排他锁）：加锁的客户端可以读数据，也可以写数据，其他客户端不能读数据，也不能写数据

语法

```
# 加锁
-- tableName: 表名
-- read / write: 读/写锁
lock tables tableName read / write

# 释放锁
unlock tables
```

元数据锁

MDL：`Meta Data Lock`

MDL加锁过程是系统自动控制，无需显式使用，在访问表的时候会自动加上

作用：**维护表结构**的数据一致性，在表上有活动事务时，不能对元数据进行写入操作

特点：当对表进行增删改查时，加MDL读锁（共享锁）；当对表结构进行变更操作的时候，加MDL写锁（排他锁）

- MDL读锁
  - `select` 语句：加的锁类型是 `SHARED_READ`
  - `insert、update、delete` 语句：加的锁类型是 `SHARED_WRITE`
  - `SHARED_READ` 与 `SHARED_WRITE` 是兼容的，不会相互阻塞
- MDL写锁
  - `alter` 语句：加的锁类型是 `EXCLUSIVE` ，与其它锁类型互斥

意向锁

作用：为了避免增删改查执行时**加的行锁与表锁冲突**，加入意向锁使得表锁不用检查每行数据是否加锁，减少表锁的检查

类型

- **意向共享锁**（IS）：由 `select` 、 `lock in share mode` 添加
  - 与表锁共享锁**兼容**，与表锁排他锁**互斥**
- **意向排他锁**（IX）：由 `insert` 、 `update` 、 `delete` 、 `select ... for update` 添加
  - 与表锁共享锁和表锁排他锁**都互斥**

行级锁

每次操作锁住**对应的行数据**，通过对索引上的索引项加锁实现，而不是对记录加锁

可分为以下三类：

- 行锁
- 间隙锁
- 临键锁

针对 `select` 获取锁

- `lock in share mode`：获取共享锁
- `for update`：获取排他锁

行锁

锁定单个行记录，防止其它事务对此数据进行 `update` 和 `delete` ，在RC、RR隔离级别下支持

有两种类型：

- 共享锁：用于读数据，其他事务可以对该行**加共享锁，但不能加排他锁**
- 排他锁：用于写数据，其他事务**不能对该行加共享锁和排他锁**

SQL与对应的锁类型

SQL	锁类型
`insert`	排他锁
`update`	排他锁
`delete`	排他锁
`select`	**不加锁**
`select ... lock in share mode`	共享锁
`select ... for update`	排他锁

注意

- 针对唯一索引进行检索时，会被**自动优化为行锁**
- 不通过索引条件检索数据，**行锁会升级为表锁**

间隙锁

锁住索引记录间隙（不含该记录），防止其他事务在这个间隙进行 `insert` ，产生**幻读**，在RR隔离级别下支持

注意

- 索引（**唯一索引**）上的等值查询，给不存在的记录加锁，**临键锁优化为间隙锁**
- 索引（**普通索引**）上的等值查询，向右遍历到最后一个不满足查询条件的值，对这段间隙加锁，此时**临键锁退化为间隙锁**
  - 因为索引不唯一，可能在该值的前面和后面插入相同值的数据，导致幻读，因此要加间隙锁
- 索引（**唯一索引**）上的范围查询，使用**临键锁**，锁住数据的同时还锁住**查询范围内间隙**
- **间隙锁可以共存**，一个事务的间隙锁不会影响另一个事务在同一间隙上加间隙锁

临键锁

行锁和间隙锁的综合，锁住数据，并锁住数据前面的间隙，在RR隔离级别下支持

默认情况下，InnoDB使用**临键锁**进行搜索和扫描，**防止幻读**