mysql

一些基础

varchar可变长度但char固定

VARCHAR 在存储时需要使用 1 或 2 个额外字节记录字符串的长度，检索时不需要处理。

decimal是金融 避免浮点数损失 java类的bigdecimal

timestamp四个字节 而datetime需要八个字节

timestamp 表示时间范围更少 只到2037

查Null只能用IS NULL 不能用等于

null需要占空间 ''不需要

null影响聚合的效果 一些聚合函数会忽略null 包括count

null表示不确定的值

自连接要注意去重！ 最好用DISTINCT

!= 和 <>效果一样

定位慢查询

超过1s 时间过长

方案1：

调试工具：Arthas

运维工具：Prometheus、**Skywalking**

> *skywalking*的底层是什么？怎么实现？

方案2： 慢日志 （调试阶段）

设置开关

设置慢日志的时间阈值 超过就记录

如何分析慢**SQL**

可以采用EXPLAIN 或者DESC命令获取mysql如何执行select语句的信息

possible_key 可能使用到的索引

key 当前实际命中的索引

key_len 索引占用大小

这两个查看是否可能会命中索引

Extra额外的优化建议

Extra	含义
Using where; Using Index	查找使用了索引，需要的数据都在索引列中能找到，不需要回表查询数据
Using index condition	查找使用了索引，但是需要回表查询数据

下者意味着有优化空间

> ***type*** *sql*连接的类型 左往右性能变差
>
> *null system const eq_ref ref range index all*

null没用到表 无需关注数

system 查询系统中的表

const:根据主键查询

eq_ref: 主键索引查询或者唯一索引查询 （只返回一条数据）

ref:索引查询 等值匹配 多行记录

range:范围查询 适用于 索引范围查询，例如 `BETWEEN`、`>`, `<`, `IN()` 等。

index：索引树扫描

all:全盘扫描

index 和all意味着要优化

索引

全盘查 效率不高

添加索引 提升速度

如where查询 group by order by的字段

没有主键，会选择第一个不包含Null的字段来作为索引，如果都没有 自动生成隐式自增id作为聚簇索引的索引键

聚簇索引（Clustered Index）是数据库中一种特殊的索引类型，它决定了数据表中 数据行的物理存储顺序。换句话说，数据表的记录按照聚簇索引的顺序存储在磁盘上。（一般为主键）

> *B+树*

相比即存索引又存 记录的 B 树，B+树的非叶子节点可以存放更多的索引，因此 B+ 树可以比 B 树更「矮胖」，查询底层节点的 磁盘 I/O次数会更少。

B+ 树叶子节点之间用链表连接了起来，有利于范围查询，而 B 树要实现范围查询，因此只能通过树的遍历来 完成范围查询，这会涉及多个节点的磁盘 I/O 操作，范围查询效率不如 B+ 树

主键索引的 B+Tree 的叶子节点存放的是实际数据，所有完整的用户记录都存放在主键索引的 B+Tree 的叶子 节点里； 二级索引的 B+Tree 的叶子节点存放的是主键值，而不是实际数据

> 聚集索引和二级索引

(聚集索引）主键查询：直接在 主键索引 所在的 B+ 树中查询，然后直接返回查询到的叶子节点（此时，叶子节点里面就是整行 记录）

如果没有主键和唯一索引 会自动生成隐藏的rowid作为聚集索引

二级索引查询：首先，在普通索引所在的 B+ 树中，查询到待查询记录的 主键； 然后，再根据这些查到的 主键， 执行“主键查询”（即，回表）

覆盖查询

覆盖索引（Covering Index）是指 查询所需的所有字段 都已经包含在索引中，无需回表（回到原表中查找数据），从而提高查询效率

比如说二级索引里面 是索引字段+存主键 如果是查字段+主键 那也是覆盖索引 直接从二级索引中返回

不过当查询的数据是能在二级索引的 B+Tree 的叶子节点里查询到，这时就不用再查主键索引查，这种在二级索引 的 B+Tree 就能查询到结果的过程就叫作「覆盖索引」 由于覆盖索引可以减少树的搜索次数，显著提升查询性能，所以使用覆盖索引是一个常用的性能优化手段。

超大分页怎么处理

limit分页查询 越往后 分页查询效率越低

分页查询 通常需要排序

分页查询（Pagination Query）是一种按页返回数据的查询方式，适用于大数据集查询，可以避免一次性加载所有数据，减少数据库压力，提高前端加载效率。

`LIMIT` + `OFFSET` 需要 扫描并丢弃 前面的数据，再返回后面的数据。

索引优化有限，MySQL 仍然需要读取数据到内存后再丢弃。 所以很慢 其实就是要接近于全盘扫 扫到限定位置为止

覆盖索引+子查询形式进行优化 本质上是先取需要的主键 再根据主键索引取行 少了 查的过程中所有行都扫一遍

```sql
SELECT id, name FROM users
WHERE id IN (
    SELECT id FROM users ORDER BY id LIMIT 900, 100
)
ORDER BY id;
```

这个查询需要返回 `id` 和 `name` 字段。如果在 `users` 表上有包含 `id` 和 `name` 的联合索引（例如：`idx_users_id_name(id, name)`），那么查询可以通过这个索引直接返回结果，没有联合索引也要回表

```sql
SELECT * FROM users WHERE id IN (...) ORDER BY id;
 查到了id 但需要回表
```

> 索引建立原则

1.量大且查询频繁的表要索引

2.常查询条件 where order by group by 操作的字段建立索引

3.区分度高的列作为索引，尽量建立唯一索引

4.字符串长度长，可以用字段的特点 建立前缀索引

5.尽量使用联合索引 ，因为可以覆盖索引，避免回表

6.索引多也不是好事 删改效率低

7.如果不能存储Null值，要设置not null约束

> 索引失效

如何判断索引失效？ 执行explain

key和**key len** 为**null** 则为索引失效 或者**key_len**长度不对

1.违反最左匹配原则

查询从最左列开始并连续

如果是AC这种情况 跳过中间 那只有最左索引生效

2.范围查询的字段的右边的列，索引失效

例如 a=2 b>1xxxx

a=2是等值 索引生效 但是b>1

扫描所有 `b > 1` 的数据

再逐行过滤 `c = 3`（即回表，性能下降）

解决方法：拆分索引 a和b c

3.在索引列上进行运算操作，索引失效

4.字符串不加单引号，索引可能失效

发生类型转换就会失效

5.%开头的模糊查询会失效，尾部%模糊查，索引不会失效

sql优化的经验

1.表的设计优化

2.索引优化--->创建原则和索引失效 最左匹配原则

3.sql语句优化

4.主从复制、读写分离

5.分库分表

> 表的设计优化

比如设置合适的类型 数值 内容的长度 节省存储成本

比如字符串类型 定长和不定长 效率

> *sql*语句优化

1.避免使用select *

2.避免索引失效的写法

3.用union all 代替union Union多一次过滤 效率低 union all返回重复的

4.避免在where 对字段进行表达式操作

5.连接优化 能用inner join 就不要用left join right join ，必须要用 一定要以小表为驱动

内连接会自动把小表放外边 但是left和right 不会

`LEFT JOIN` 和 `RIGHT JOIN` 需要返回左表或右表中的所有行，即使没有匹配的记录也会返回 `NULL`，因此它们的计算开销更大，特别是当表的数据量较大时。

> 主从复制、读写分离

master负责写 从节点负责读

事务

> 怎么实现？

START TRANSACTION-----执行sql语句---COMMIT要不全部完成 要不全部失败

> *ACID*原则

原子性：操作同时成功或者同时失败 失败就回滚

一致性：商品库存：当库存减少时，总销售量应同步增加，数据间的约束关系保持一致。

隔离性：并发事务之间相互独立，不能互相干扰。一个事务未提交前，其操作对其他事务是不可见的。

持久性：事务一旦提交，所做的修改将永久保存在数据库中，即使发生系统故障。

> 隔离级别

脏读: 一个事务读到了另外一个事务还没提交的数据 （并发问题 抢跑）

幻读：一个事务查的时候没有 但是插入的时候又发现这行数据已经存在

不可重复读：同一事务内 两次读取同一条数据 但不一致+

读未提交<读已提交<可重复读<串行化

读未提交都解决不了

串行化解决所有问题 但效率低

读未提交：由于事务可以读取其他未提交事务的数据，所以可能会读取到“脏数据”。也就是说，如果一个事务修改了数据，但未提交，另一个事务就可以看到这个修改，但如果第一个事务最终回滚，则第二个事务就会读到无效的数据。

读已提交：事务只能读取 已经提交事务的数据，即它 不能读取未提交的数据。 如果在同一事务中读取相同数据的两次，可能会得到不同的结果。因为在这两次读取之间，其他事务可能已经修改并提交了数据，导致读取到的数据不同。

每次查询时，数据库会加锁当前被查询的行（即读取到的数据），但每次查询完毕后，锁会被释放。因此，其他事务可以在事务A的查询过程中修改其他未加锁的数据行。

可重复读：事务中的 所有查询在事务开始后都能看到相同的数据，即使其他事务提交了数据也不受影响。 隔离级别下，为了保证同一事务在不同时间读取相同数据时不受其他事务影响，数据库会使用 行级锁（或者在某些数据库中使用更严格的锁策略）来 锁定查询的行。幻读通常发生在使用范围查询时（如查询 `age > 30` 的所有记录）。如果在事务A查询

时，事务B在其后插入了一行满足查询条件的记录，那么事务A再次查询时，会发现数据集发生了变化。

锁会持续整个事务的生命周期，直到事务提交或回滚

可重复读和读已提交基于MVCC（保证并发）

串行化是基于锁

> 约束模式

`SET CONSTRAINTS`设置当前事务里的约束检查的特性。`IMMEDIATE` 约束是在每条语句后面进行检查。`DEFERRED`约束一直到事务提交时才检查。 每个约束都有自己的`IMMEDIATE`或`DEFERRED`模式。

MVCC（建议重看视频 搞不懂）

> *undo log*和*redo log*的区别

sql语句操作时 先去看

缓冲池 :内存的区域，先去看内存有没有 没有的话再从磁盘加载到内存中

数据页:存储引擎磁盘管理的最小单元 每个页大小默认为16kb 页中存储的是行数据

减少磁盘的IO

脏页 内存的数据没存到磁盘 同步失败 内存数据有可能消失

> *redo log*

redo log buffer存在内存 redo log file 存在磁盘

修改信息存在redo log file 用于在刷新脏页到磁盘 发生错误时，进行数据恢复使用 顺序磁盘IO 性能快

Write ahead logging： 先写日志，再写数据

当 事务提交 或 日志缓冲区满 时，系统会将日志缓冲区中的内容刷新到磁盘上的 **Redo Log** 文件。这个过程是通过顺序写入的方式来提高性能。

只有在 Redo Log 被刷新到磁盘之后，事务才会被视为提交成功。

如果在事务提交后系统崩溃，数据库启动时会扫描 Redo Log，根据日志中的记录 回放 所有已提交事务的操作，直到恢复到崩溃前的状态。 未提交的就失败回滚了

> *undo log*

记录数据被修改前的信息 提供回滚和**MVCC**

undo log 是逻辑日志 delete一条记录 就记录insert记录 记录相反的记录

> *mvcc* 也能控制隔离

维护一个数据的多个版本 使得读写操作没有冲突

依赖于隐藏字段 **undo log**日志 **readView**

隐藏字段：

DB TRX_ID 最近修改事务**ID**

DB_ROLL_PTR 回滚指针 指向这条记录的上一个版本 配合**undo log**

DR_ROW_ID 隐藏主键

undo log日志 配合指针 链表！！！！！！！！

当insert的时候，产生的undo log日志只在回滚时需要，在事务提交后，可被立即删除。

而update、delete的时候，产生的undo log日志不仅在回滚时需要，mvcc版本访问也需要，不会立即被删除。

> *readview:* 快照读*sql*执行时 *MVCC*提取数据的依据 记录和维护系统当前活跃的（未提交）事务*id*

当前读：读取的是记录的已提交的最新版本（会发生不可重复读、发生在串行化和读已提交） 读取时还要保证其他并发事务不能修改当前记录 会对读取的记录进行加锁（读锁） 日常操作都是一种当前读

快照读：读取的是可见版本 （无加锁的select) 有可能是历史数据 是非阻塞读 （发生在可重复读和读已提交的级别）

可重复读的第一个**select**语句才是快照读的地方

读可提交 每次**select** 都生成一个快照读

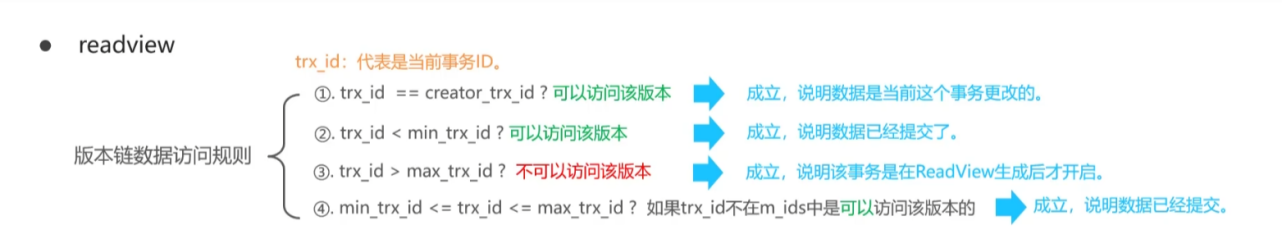
m_ids 当前活跃的事务ID集合

min_trx_id:最小的活跃集合id

max_trx_id：预分配事务ID，当前最大事务ID+1

creator_trx_id:readview创建者的事务ID

trx_id代表当前事务id （在记录的隐藏字段）



简单讲：存每个事务和数据的情况 根据事务和数据的情况来决定读哪个

比对记录的最近事务修改ID

undo log日志和快照读

主从同步原理

核心就是二进制日志 binlog

定义了所有 DDL和DML update insert 之类的 除了查询

从节点的IO-Thread 读主节点的binlog 然后写到relay log

SQL thread 再读relay log

其实就是缓存那一套 没啥

两个**thread**分开保证性能

写**binlog**----读**binlog**到**relay log**-----执行复制

两阶段提交

写入redolog (prepare阶段)

commit阶段：写入binlog和把redo log写入状态设置为commit

binlog无而redolog有 则要回滚

如果两者都有 则事务成功

分库分表

太大就分表呗

> 垂直分库

根据业务将不同表拆分到不同库

跟微服务有关系

> 垂直分表

以字段为依据， 把不同字段拆到不同表中

比如商品的description 跟ES有关系

冷热数据分离 减少IO过渡争抢

> 水平分库

拆成不同库

类似于分片集群　取模找库

对应redis的 哈希槽 16384

> 水平分表

一个表拆成多个表 （可以在同一个库）

找表也是取模呗

> 新问题

分布式事务一致性问题

跨节点关联查询

跨节点分页、排序函数

主键避重

中间件**Mycat** 来解决以上问题 （水平分）

具体技术

> *ShardingSphere*

核心是定义分片规则，比如基于某个字段（如 user_id）进行取模分库。在配置里，指定数据源（多个数据库实例）和分片策略，ShardingSphere 会自动路由 SQL 到对应的库

```
rules:
```

```yaml
- !SHARDING
  tables:
    apply_record:  # 逻辑表名
      actualDataNodes: ds.apply_record_$->{0..10}  # 11张分表
      tableStrategy:
        standard:
          shardingColumn: apply_id
          shardingAlgorithmName: apply_id_mod
  shardingAlgorithms:
    apply_id_mod:
      type: MOD
      props:
        sharding-count: 11
```

ShardingSphere 的水平分表支持 插入新数据，它会根据分片规则自动路由数据到正确的分表。

，**ShardingSphere** 可以通过 **XML** 配置 完成分库分表的设置，一旦配置好，应用层不需要再写分片相关的代码。

锁

innoDB支持表级锁和行级锁

行级锁：记录锁、间隙锁、临键锁

记录锁：单个行记录的锁

间隙锁：锁定一个范围 不包括记录本身

> 一个*bug*

当我们执行 `UPDATE`、`DELETE` 语句时，如果 `WHERE` 条件中字段没有命中唯一索引或者索引失效的话，就会导致扫描全表

相当于锁住的全表

逐行看会先上读锁 匹配到就上写锁

> 临键锁

临键锁 = 行锁（**Record Lock**）+ 间隙锁（**Gap Lock**）

临键锁不仅锁定查询到的记录本身，还会锁定该记录前的索引区间，确保查询范围内的数据不会被插入新行，从而避免幻读。

防止幻读，主要在 可重复读（**REPEATABLE READ**） 隔离级别下生效。

> 共享锁和排他锁

简称读锁和写锁

读锁很多线程都能拿到 防止上写锁

写锁互斥所有锁 只有一个线程能获取

> 意向锁

要用到表锁的话？

先判断有没有行锁 但是一行一行遍历肯定不行 如果没有意向锁，数据库在加表锁时，必须遍历整个表，检查每一行是否已经加锁，这会导致性能下降！

意向锁（Intention Lock）是 表级锁 标记表内是否有行锁，以便优化表级锁的加锁过程。

当事务要获取表锁（**S/X**）时，需要先检查意向锁

有行锁---------->上意向锁----->想要获得表锁要先等意向锁（除非是全读）

意向锁和行锁没冲突 只是充当媒介（信号量）

上了意向锁后再判断是否能上表级锁

意向共享锁：有意向对表中记录加共享锁，加共享锁之前先取得意向共享锁

意向排他锁：同上，加排他锁

意向锁之间互相兼容

意向锁和表级锁之间 会产生冲突

	IS 锁	IX 锁
S 锁	兼容	互斥
X 锁	互斥	互斥

优化器？

MySQL 优化器的作用是选择执行 SQL 查询的最佳执行计划，从而提高查询效率

生成多个执行计划：根据查询的不同方式（如索引使用、连接顺序等）生成多个可能的执行计划。

评估代价：使用统计信息（如表的行数、索引选择性等）计算每个执行计划的代价。

选择最优计划：选择代价最小的执行计划，并将其交给查询执行引擎执行。

并不能保证 100% 的最优，特别是在 SQL 写得不规范或索引失效时，优化器可能会选出次优计划。

> 原理

优化的核心原理是 成本估算，即选择代价最低的执行计划

优化器会尝试通过改写 SQL 查询的结构来提高执行效率（例如将子查询转换为 `JOIN`）。

优化器通过评估不同索引的使用情况来决定是否以及如何使用索引。

例如联合索引，优化器会优先选择

杂

> 自增值不连续的 *4* 个场景：

1. 自增初始值和自增步长设置不为 1 （这个不用说）
2. 唯一键冲突 ： 当插入数据时，如果主键或唯一索引冲突，插入会失败，但自增值仍然会被占用，导致 ID 出现跳跃。
3. 事务回滚:即使事务回滚，自增 ID 仍然会递增，不会回到之前的值。这是因为 MySQL 在分配自增 ID 时，并不会等事务提交才确定 ID，而是在 `INSERT` 语句执行时就占用了 ID。
4. 批量插入（如 `insert...select` 语句 ： `INSERT ... SELECT` 语句一次性插入多条数据时，MySQL 会先分配足够的 ID 以确保所有数据插入成功，但如果部分数据因冲突失败，ID 仍然会被消耗，导致 ID 不连续。