MySQL

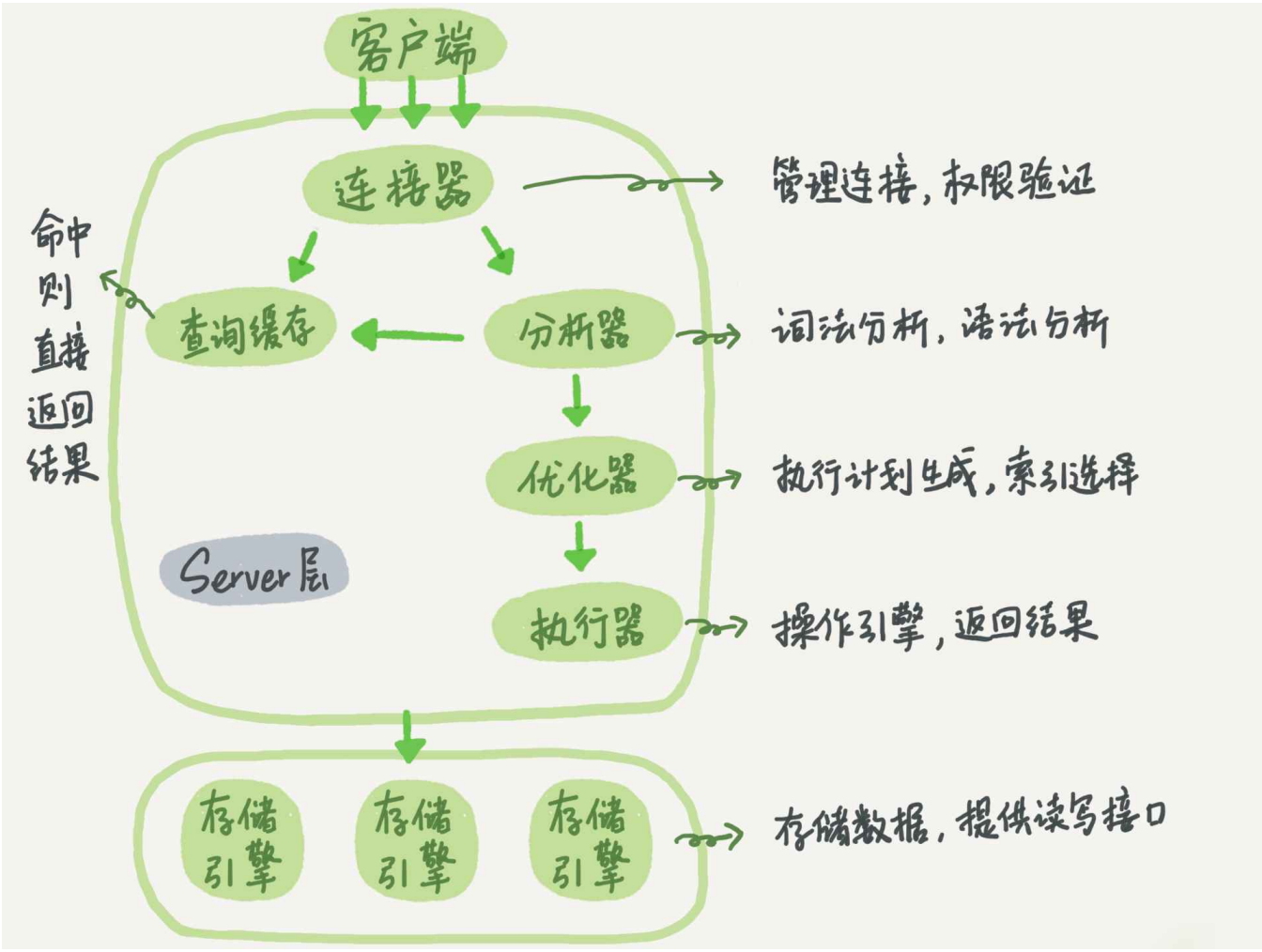
参考文章

- 整体类：掘金 贾不假
- 程序员库森
- 企业面试题｜最常问的MySQL面试题集合（一）
- 企业面试题｜最常问的MySQL面试题集合（二）
- 企业面试题｜最常问的MySQL面试题集合（三）

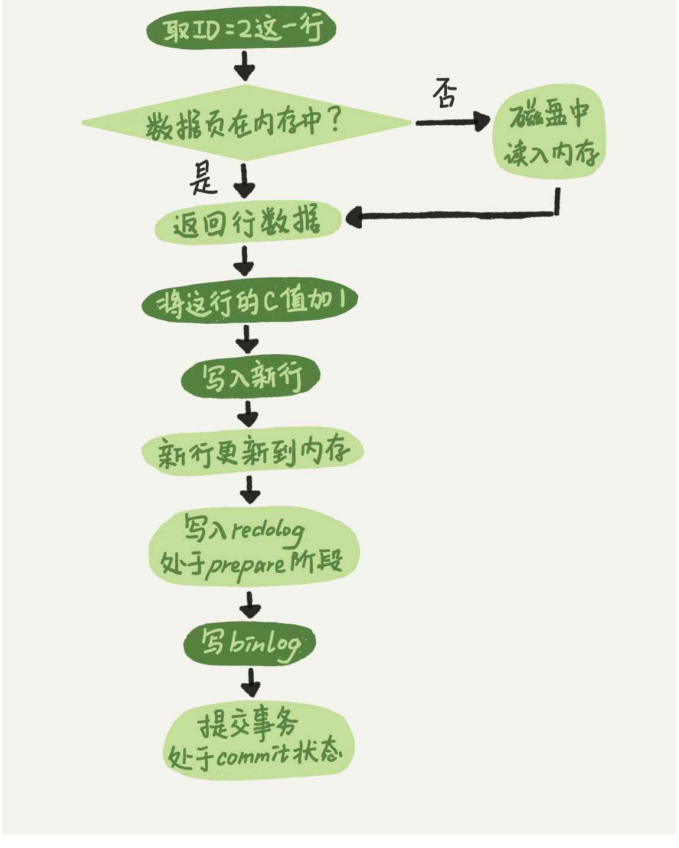
整体性问题

- Conception：
  a. 架构：客户端、连接器、查询缓存、分析器、优化器、执行器、存储引擎
  b. Mongodb和MySQL区别
  c. 关系型数据库和非关系型数据库
- Question：
  a. 一条查询语句是怎么执行的？/MySQL架构？
    - 客户端：与连接器建立连接
    - 连接器：查询缓存，命中则返回，没有进入分析器
    - 查询缓存
    - 分析器：分析sql语义
    - 优化器：优化sql语句，选择索引、join表的连接顺序
    - 执行器：权限验证，通过则执行sql
    - 存储引擎：存储数据

    - Serve层和引擎层



b. 一条更新语句是怎么执行的？（两阶段提交）



- 写入顺序（涉及到WAL）
  - 日志写入内存
  - 数据写入内存
  - 内存中日志落盘
  - 内存中数据落盘
- 两阶段提交：
  - 写入redolog，prepare状态

- 写入binlog
- commit状态（保证两个日志都落盘了）

- 如果没有进行两阶段提交的问题：
  - 先写redo再写bin：进程异常重启，redolog重新恢复把0变为1，binlog没有记录，之后恢复时候还是0（**恢复时候用binlog，进行恢复时候会少内容**）
  - 先写bin再写redo：redolog没有记录，这一行为0，binlog之后某次恢复时候恢复成了1（**事务回滚时候用redolog，redolog没有记录，事务无效并没有回滚着一行**）

c. Mongodb和MySQL区别？

- MongoDB：基于分布式文件存储的数据库，非关系型，Json的存储格式

	Mongodb	**MySQL**
数据库模型	非关系型、文档型	关系型
成熟度	新型数据库，成熟度较低	成熟度较高
数据处理方式	基于内存，热数据在内存中，高速读写	不同引擎特点不同
缺点	不支持事务，开放文档不完善	海量数据处理较慢

d. 关系型数据库和非关系型数据库？

- 关系型数据库：
  - **表**结构，欠缺灵活度
  - 优点：易于理解；通用SQL语言使得操作关系型数据库非常方便；丰富的完整性；必须具有ACID特性
  - 缺点：并发量高，**磁盘I/O限制瓶颈**；每天产生的数据多，查询效率低；难以进行横向扩展，升级扩展时候需要进行停机维护和数据迁移
  - MySQL、Oracle
- 非关系型数据库：
  - 存储**key-value**形式，文档形式，图片形式
  - 优点：**查询速度快**，根据键值对就可以完成查询，不需要进行多表联合查询；
  - 缺点：**只适合一些简单的数据，较复杂查询时候适合关系型数据库**
  - MongoDB（文档型）、Redis（key-value型）、Memcached

基础/语法问题

- Conception：
  a. 数据库三范式
  b. 反范式
  c. char和varchar
  d. drop、delete、truncate
  e. 存储过程
  f. 触发器
- Question：

a. 说说三范式?
   - 第一范式：数据库中的每一列具有原子性，不可再分（原子性，否则就不是关系数据库）
   - 第二范式：有主键，非关键字段完全依赖主键（唯一性，一个表只说明一个事务）
   - 第三范式：非主键字段不能相互依赖（不存在传递关系）
   - BCNF：不允许候选码以外的的东西对非主属性起到依赖性作用
   - 并不是所有的表一定要满足三大范式
b. 反范式?
   - 数据库要求满足三范式，不能有冗余字段
   - 但是有的冗余字段可以方便查询（eg：单价*数量=金额，但是多一列金额可以提高查询速度）
c. 数据类型?
   - 整数类型：
     - TINYINT、SMALLINT、MEDIUMINT、INT、BIGINT（1字节、2字节、3字节、4字节、8字节）；
     - 长度大多数场景没有意义，不会限制值的合法范围，只会影响字符的个数，和UNSIGNED ZEROFILL（INT(5)，存12，实际为00012）属性配合使用才有意义
   - 实数类型：
     - FLOAT、DOUBLE、DECIMAL（存储比BIGINT还大的整型，能存储精确的小数）
   - 字符串类型：
     - VARCHAR、CHAR、TEXT、BLOB
     - 尽量避免使用TEXT/BLOB类型，查询时会使用临时表，导致严重的性能开销。
   - 枚举类型：
     - ENUM
     - ENUM存储非常紧凑，会把列表值压缩到一个或两个字节；ENUM在内部存储时，其实存的是整数。
     - 尽量避免使用数字作为ENUM枚举的常量，因为容易混乱
   - 日期和时间类型：尽量使用timestamp，空间效率高于datetime；如果需要存储微妙，可以使用bigint存储
d. char和varchar区别?
   - char(n)：定长字段
   - varchar：变长字段，占用空间要+1（存长度）
   - 检索效率：char > varchar
   - 如果使用中可以确定某个字段值的长度，可以用char，否则尽量使用varchar
e. varchar为什么建议不超过255?
   - varchar需要一个字节保存长度
   - 大于255时，长度标识需要两个字节，建立的索引也会失效（表索引的前缀长度最长是767字节，5.6版本之前）
f. varchar和text区别?

	varchar	text

数量	<=255时候+1；>255时候+2	不能指定数量，实际字符数+2字节
默认值	可以有	不能有默认值
索引	直接创建索引	创建索引需要指定前多少个字符
查询		查询text需要创建临时表
	>255时候优化成text，+2字节存储长度	TINYTEXT(255长度)、TEXT(65535)、MEDIUMTEXT（int最大值16M），和LONGTEXT(long最大值4G)

g. SQL约束（对表中数据的一种约束）？
   - 表中数据的限制条件，保证表中的记录完整性和有效性
   - NOT NULL：非空约束
   - UNIQUE：唯一约束
   - PRIMARY KEY：主键约束，不能重复不能为空
   - FOREIGN KEY：外键约束
   - CHECK：检查约束，检查数据表中字段值有效性
   - DEFAULT：默认值约束

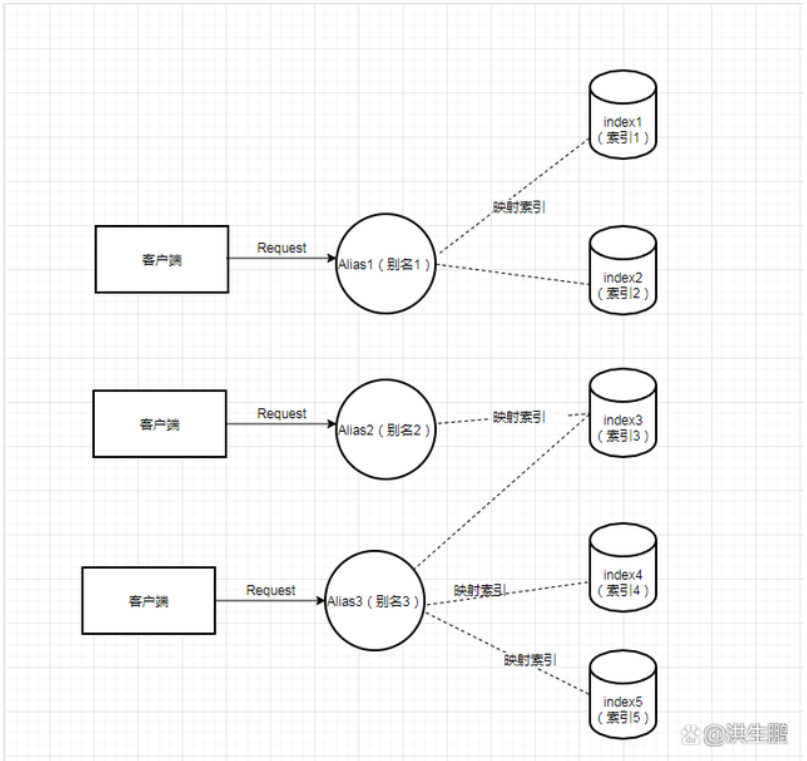
h. drop、delete、truncate区别?

	DELETE	**TRUNCATE**	**DROP**
类型	DML	DDL	DDL
回滚	可以	不可以	不可以
删除内容	表结构还在，删除表的全部或者一部分数据行	表结构还在，删除表中每一行	从数据库删除表，所有数据行，索引和权限也删除
删除速度	慢，逐行删除	较快	最快

i. in和exists区别?

	IN	EXISTS
返回结果	返回结果集	返回true/false
两张表如何操作	把内外表做hash连接	对外表作loop循环，每次loop循环再对内表进行查询
语法	select * from where field in (value1,value2,value3,…)	• SELECT … FROM table WHERE EXISTS (subquery)
适用场景	A表大	A表小

   - MySQL小表驱动大表：第一层需要进行数据库连接，所以适用exists时候外部表A比较小，也就是SELECT x FROM A WHERE EXISTS (SELECT FROM B);



j. 说说存储过程？

- 预编译的SQL语句，由一些T-SQL语句组成的代码块，用到这个功能时候调用即可
- 执行效率高，一个存储过程代替大量T-SQL语句，提高通信效率，确保数据安全
- 不太推荐存储过程

k. 触发器

- 六种：Before Insert、After Insert、Before Update、After Update、Before Delete、After Delete

引擎

- Conception：

  a. InnoDB、MyISAM（咪塞姆）、Memory

- Question：

  a. InnoDB和MyISAM区别

	InnoDB（MySQL5.5.8之后默认InnoDB）	**MyISAM（5.5.8之前）**
事务	支持事务	不支持事务
外键	支持外键	不支持外键
索引	是聚簇索引，B+tree为索引结构，数据文件和索引绑在一起；不支持全文索引	非聚簇索引，B+tree为索引结构，索据文件分离，索引保存的是数据文件针；支持全文索引（不如使用外部的ElasticSearch或Lucene）
文件组成	frm：表定义文件；idb：数据文件（数据文件和索引在一块） 同一时刻表中的**行数**对于不同的事务而言是不一样，所以不记录行数	frm：表定义文件；myd：数据文件；索引文件
锁	表级锁（写锁、排他锁）、行级锁（读锁、共享锁）	表级锁
主键	**必须有主键，如果每指定会自己生成默认主键**	可以没有
日志	redolog	没有redolog

行数	**不保存具体行数**	**一个变量保存了整个表的行数**
应用场景		保存日志；系统临时表

b. InnoDB特性1：自适应hash索引（adapt hash index）

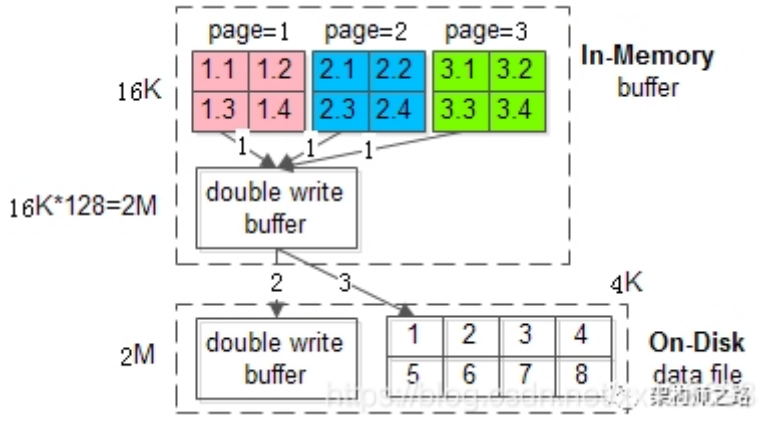
- 某二级索引被频繁访问，二级索引成为热数据，建立哈希索引带来速度提升

```
#查看是否打开
show variables like '%ap%hash_index';
```

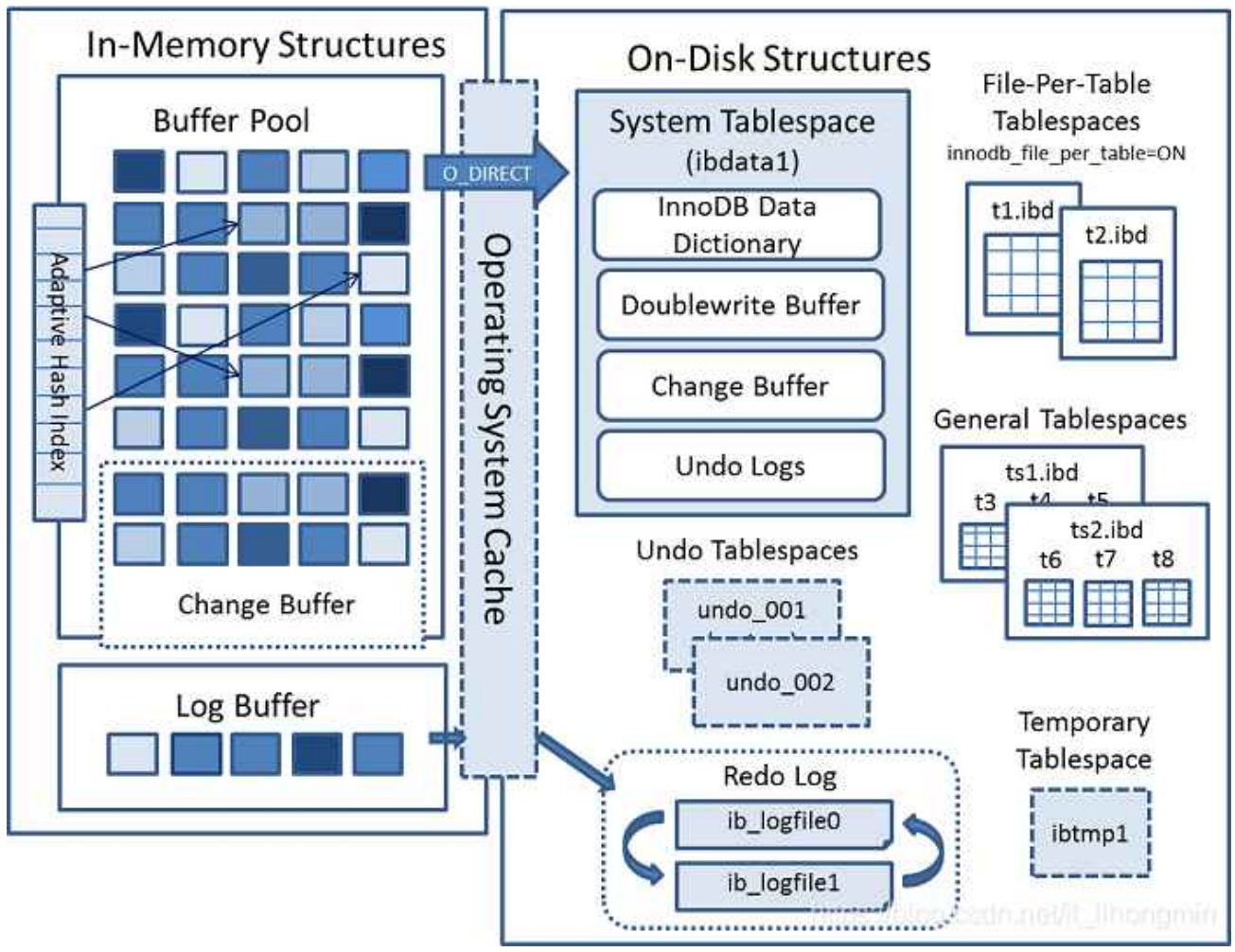
- 优点：没有树高，便于访问
- 缺点：占用innoDB buffer pool空间；只适合等值查询

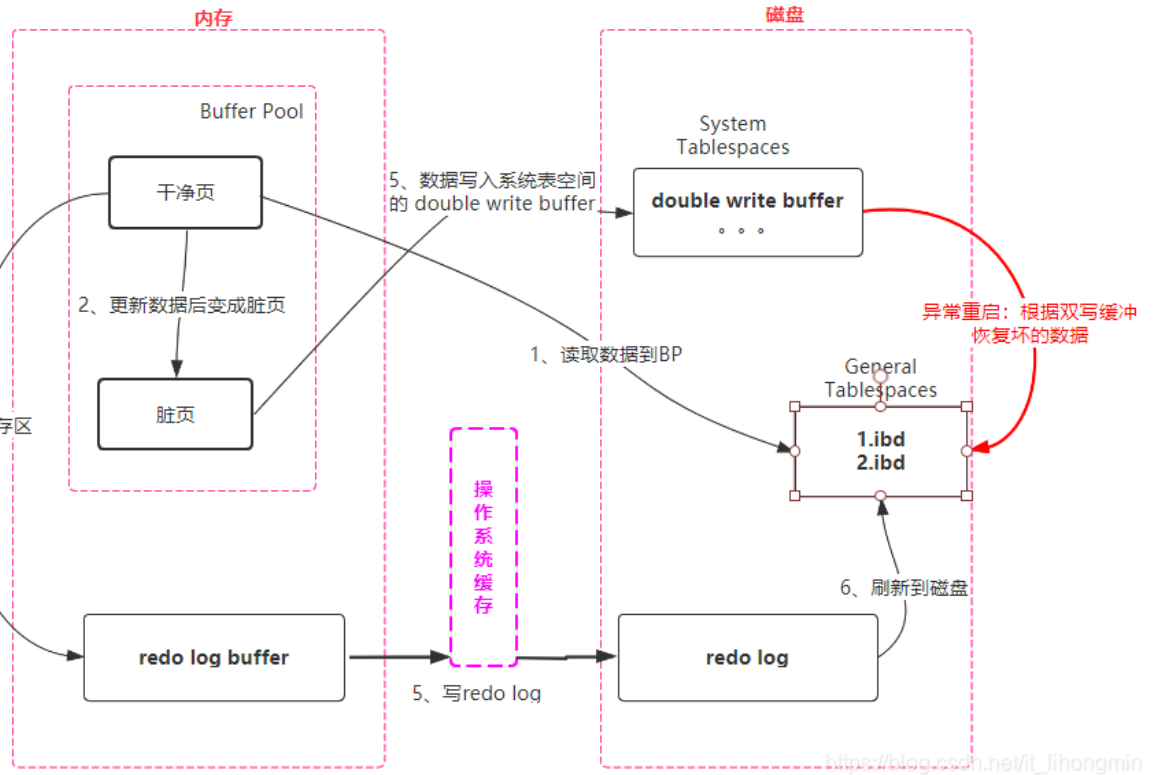
c. InnoDB特性2：double write buffer

- 页数据先拷贝到DWB的内存中
- DWB内存，先刷到DWB磁盘上（此时宕机，磁盘数据完整）
- DWB内存，刷数据到磁盘上（此时宕机，DWB磁盘上存有数据）
- 上述两步写了两次，故double write



- DWB由128个页构成，容量只有2M。

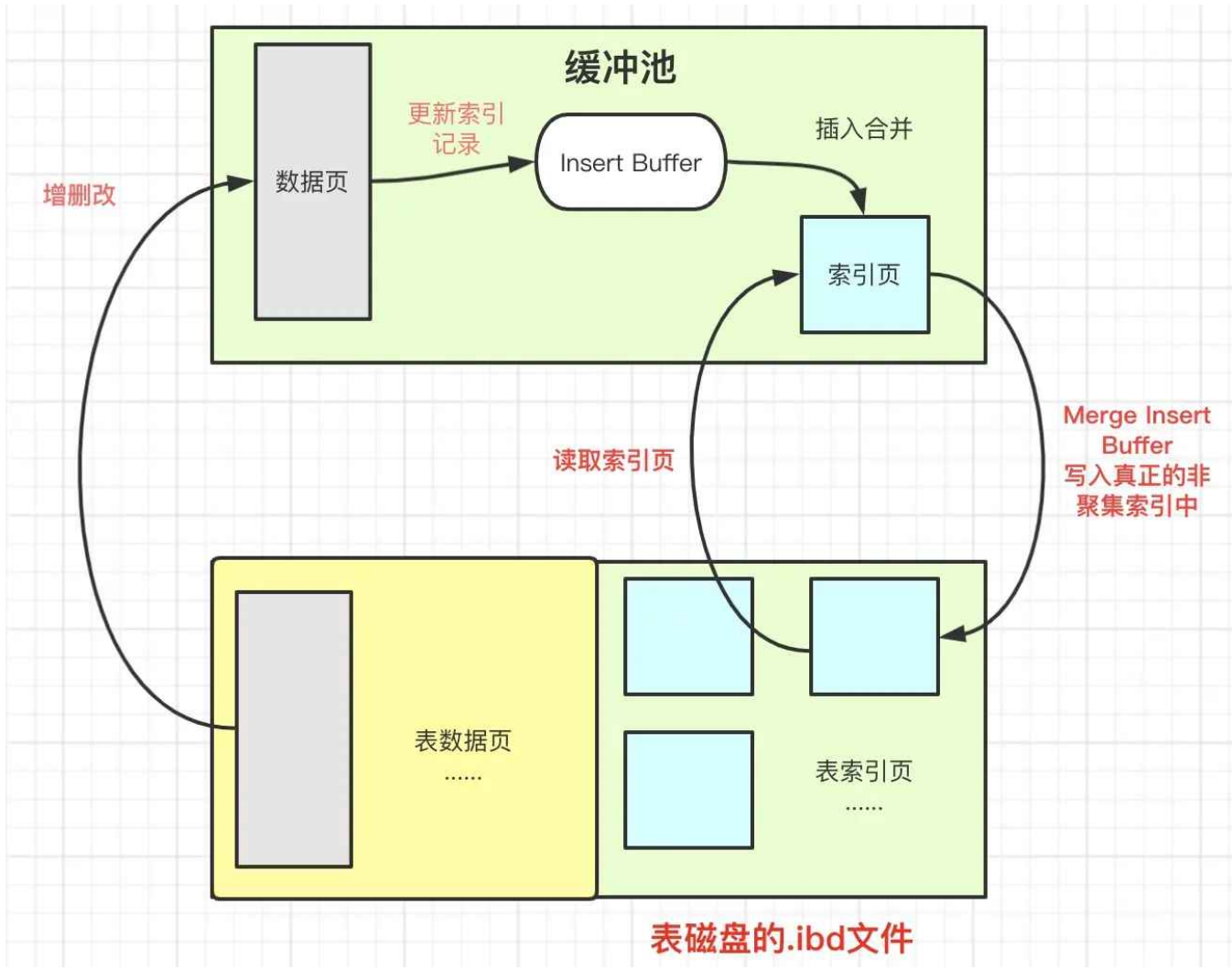




- DWB作用：
  - 能减少redo log量，有double write，redo log只记录二进制变化量，等同于bin log
  - **提高了页写回的效率，内存中的page在disk中不一定连续，写入到DWB可以实现连续写，提高了效率**
  - 弊端：开辟额外的内存和磁盘空间

d. InnoDB特性3：insert buffer

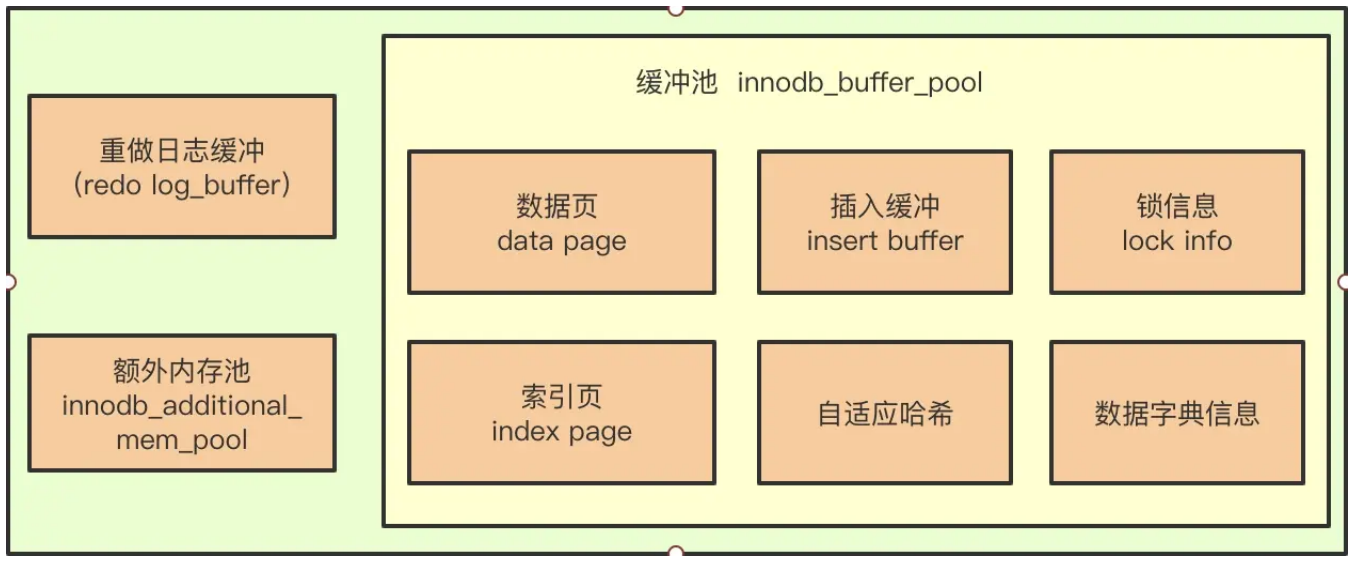
- Insert Buffer好文
- insert buffer本身就是B+树插入数据之前先在buffer中进行修改，之后再写入磁盘
- 条件：索引是非聚簇索引（聚簇索引一般都是主键索引，自增，插入本来就有序）；索引不是唯一的（缓存池中先构建一个B+树索引，若索引必须唯一，那么要先去查一次当前索引是否唯一，失去了buffer的意义）。



- 5.5之前叫Insert Buffer；之后优化为change buffer（Insert Buffer、Delete Buffer、Purgebuffer -- INSERT、DELETE、UPDATE）

e. InnoDB 内存管理机制：Buffer Pool

- 一块内存区域，为了提高数据库的性能，数据库操作数据的时候，把硬盘上的数据加载到buffer pool，不直接和硬盘打交道，操作的是 buffer pool 里面的数据，数据库的增删改查都是在 buffer pool 上进行
- buffer pool 里面缓存的数据内容也是一个个数据页
- 其中「有三大双向链表」：
- 「free 链表」
  - 用于帮助我们找到空闲的缓存页
- 「flush 链表」
  - 用于找到脏缓存页，也就是需要刷盘的缓存页
- 「lru 链表」
  - 用来淘汰不常被访问的缓存页，分为热数据区和冷数据区，冷数据区主要存放那些不常被用到的数据



f. InnoDB预读（read-ahead）：
    - 预料某些页马上会被读到
    - 预读算法：
        - Linear read-ahead：临近的页
        - Random read-ahead：根据缓存池中已有的页面预测，如果在缓冲池中找到相同范围的13个连续页面，则InnoDB异步发出请求以预取该范围的其余页面

g.

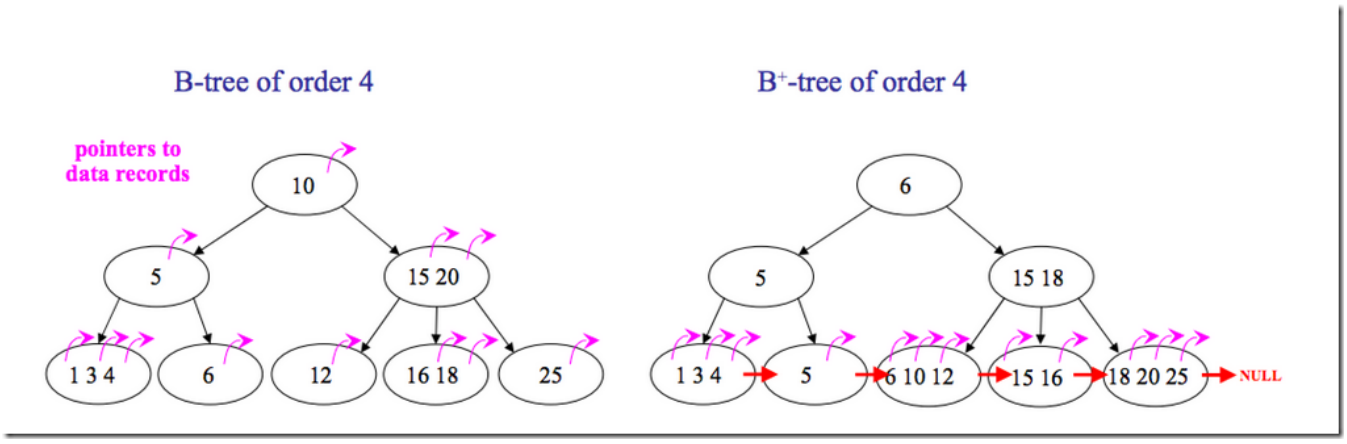
索引

- Conception：
    a. 数据结构角度：
    b. B+树索引、Full-Text全文索引、哈希索引、R-Tree索引
    c. 物理存储角度：
    d. 聚簇索引：叶子节点存数据
    e. 非聚簇索引（辅助索引、二级索引）：叶子节点存主键
    f. 逻辑角度：
    g. 唯一索引
    h. 普通索引
    i. 主键索引
    j. 单值索引、单列索引
    k. 联合索引、复合索引、多列索引
    l. 覆盖索引：查询语句的结果从索引中就可以取得，不需要回表
    m. 冗余索引、重复索引
    n. 一些规则：
    o. 最左前缀原则
    p. 索引下推
    q. 不太常见：
    r. 倒排索引、反向索引
    s. 倒序索引
    t. 前缀索引：把很长字段的前面的一部分作为索引
    u. 空间索引
- Question：
    a. 索引是什么？为什么要有索引？
        - 一个排序的数据结构，提高查询效率，字典中的目录
        - 一种特殊的文件，包含着对数据表里所有记录的引用指针
    b. 索引的底层实现？
        - 哈希索引（全文搜索引擎的索引）：将一个字段使用哈希函数映射到一个地址，冲突用拉链法
        - B树索引：

- B+树索引：

c. 索引为什么用B+树？

- B树和B+树对比
  - B+树只有叶子节点存数据，内部节点不存指向数据的指针，内存节点更小，磁盘能容纳的关键字数量越多，IO读写次数少
  - 方便扫库，只扫描一次叶子节点即可，更适合范围查询；B树的话需要进行中序遍历



- Hash与B+树对比：
  - 可以快速定位但是没有顺序
  - 只有Memory引擎显式支持哈希索引
  - 适合等值查询，不适合范围查询
  - 大量重复键值时效率低
- 二叉树与B+树对比：
  - 树高度不均匀，不能自平衡，IO代价高
- 红黑树和B+树对比：
  - 树的高度随着数据量增加而增加，IO代价高

d. 索引的优缺点？

- 优点：
  - 顺序扫描，加快检索速度
  - 随机I/O变成顺序I/O
- 缺点：
  - 时间（创建所以、每次修改数据需要维护索引）
  - 空间（索引文件占用物理空间）

e. 如何创建索引？

- CREATE TABLE时候
- ALTER TABLE命令增加索引
- CREATE INDEX创建索引

f. 创建索引时候需要注意什么？

- 非空字段：指定列为NOT NULL，MySQL中含有空值的列很难进行查询优化，用0/特殊值/空串替代空值
- 离散程度大的值（count()函数可以查看差异值）
- 索引字段小好，IO次数少

g. 创建索引的原则

- 最左前缀匹配原则
- =和in可以乱序，查询优化器会优化
- 尽量选择区分度高的列作为索引（比例越大扫描的次数越少）
- 索引列不能参与计算
- 尽量扩展索引，不要新建索引（已经有a，要建a,b，直接修改a）
- 选择较短的数据类型，减少磁盘占用率

h. 索引分类？
- 存储结构：B+树索引、哈希索引、全文索引、RTree索引
- 物理存储：聚簇索引、非聚簇索引（辅助索引、二级索引）
- 逻辑角度：
  - 唯一索引、普通索引、主键索引
  - 单值索引/单列索引、联合索引/复合索引/多列索引
  - 覆盖索引、冗余索引、重复索引
- 不常见：倒排索引、反向索引、倒序索引、前缀索引、空间索引

i. 普通索引和唯一索引怎么选择？
- 普通索引：更新场景（将数据页读入内存，更新数据页，可用到change buffer优化；更新数据还需要判断唯一索引是否唯一）
- 唯一索引：查询场景（唯一索引扫整张表，查到即返回；普通索引还需要向后遍历）
- change buffer：数据页在内存中直接更新，不在的话将缓存操作缓存在change buffer，下次访问时候写入页（节省随机读磁盘的IO消耗），普通索引更新可用到，唯一索引不行

j. 主键索引、唯一索引？
- 主键索引是唯一索引，唯一索引不一定是主键索引

k. 什么是回表？
- 先通过非主键索引找到**主键id**，再通过**主键**从**主键索引**里取出数据

l. 聚簇索引和非聚簇索引？

聚簇索引	**非聚簇索引**
索引的叶子节点存储了数据	索引的叶子节点存储了主键值
不需要回表，直接找到了数据	需要回表（如果没有发生覆盖情况的话）

- MyISAM无论主键索引还是二级索引都是非聚簇索引，InnoDB主键索引是聚簇索引，二级索引是非聚簇索引
- 自己建的索引基本都是非聚簇索引

m. 非聚簇索引一定会回表吗？
- 不一定，如果包含了所有需要查询字段的值（覆盖索引）就不回表

n. 联合索引？为什么注意联合索引的顺序？
- 使用多个字段同时建立一个索引
- 需要安装建立索引时的顺序挨个使用，否则无法命中（原因：先按第一列排序再按第二列排序）；将频繁使用的列放在前面

o. 前缀索引？

- 原因：索引字段太长占据空间不利于维护
- 做法：把某个列开始的部分字符串作为索引值

p. 说说最左前缀原则?

- 最左面的一个列不触发，之后的索引都无法触发

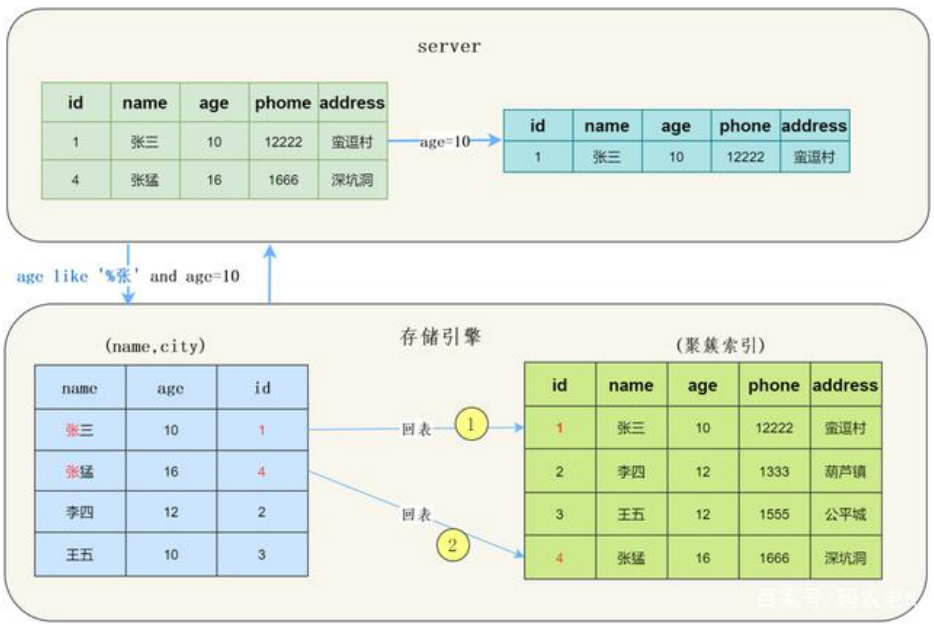
```
# 存在(a,b,c)的索引
#a,b走索引
where a = 1 and b = 1
#b没触发，c也不走索引
where a = 1 and c = 1
#a没触发，b,c都不走索引
where b = 1 and c = 1
```

- MySQL新版本会优化WHERE子句后面的列顺序

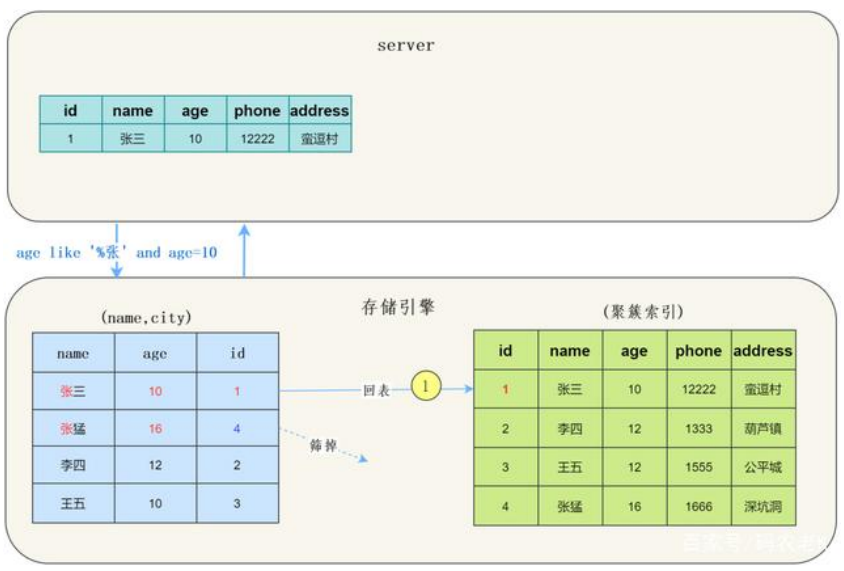
q. 说说索引下推?

```
# 有(name,age)的联合索引
select * from userinfo where name like "ming%" and age=20;
```

- 5.6之前返回name符合的结果，拿着id值再去回表查询



- 5.6开始索引内部判断age是否为20



- 减少了回表次数

- 索引下推InnDB引擎只适用于二级索引/非主键索引（聚簇索引会将整行数据读到缓冲区，索引下推减少IO次数就失去了意义，数据已经在内存中，不用再去读取了）
- 索引下推InnDB引擎只适用于二级索引（聚簇索引会将整行数据读到缓冲区，索引下推减少IO次数就失去了意义，数据已经在内存中，不用再去读取了）

r. 怎么查看有没有用到索引?
   - EXPLAIN语句
   - possible_key：查询中可能用到的索引
   - key：当前查询真正使用的索引

s. 为什么推荐使用自增长主键作为索引?
   - 自增主键是连续的
   - 每次插入数据都是插到最后，减少列分列和移动的频率

t. 使用索引一定能提高查询性能吗?
   - 通过索引查询通常比全表扫描快
   - 不一定能够提高查询性能，索引需要维护

u. 索引使用场景?
   - 字段数值有**唯一性**的限制
     - 业务上具有唯一特性的字段，即使是组合字段，也必须建成唯一索引
     - 不要以为唯一索引影响了insert的速度，这个速度损耗可以忽略，但提高查询速度是明显的
   - 使用**最频繁**的列放到联合索引的**最左侧**
   - **频繁作为where查询条件的字段**添加index普通索引
   - 只用**字符串前缀**添加索引
   - 在多个字段都要创建索引**联合索引**优于单值索引
   - 经常group by 和order by的列（同时存在使用联合索引，group by 索引放前面，单独就单独创建）
   - 更新update,删除delete的where条件列
   - 去重字段需要创建索引
   - 多表join字段需要创建索引
   - 首先join表的数量尽量不要超过三张，其次对where条件创建索引，对于连接的字段创建索引
   - 使用类型小的列创建索引
   - 区分度高（散列性高） 的列适合做索引

v. 什么时候索引失效？什么时候不走索引?
   - 最左前缀法则
   - 列涉及到运算范围条件（between、>、<、in）
   - 类型不一致
   - 计算函数导致
   - 运算符（+-*/!）导致
   - OR引起：or前面的条件中的列有索引，后面的没有，所有列的索引都不会被用到

- 模糊搜索：使用like以通配符开头（‘%字符串’）

```
WHERE name LIKE "%wang%"
#以上语句用不到索引，可以用外部的ElasticSearch、Lucene等全文搜索引擎替代。
```

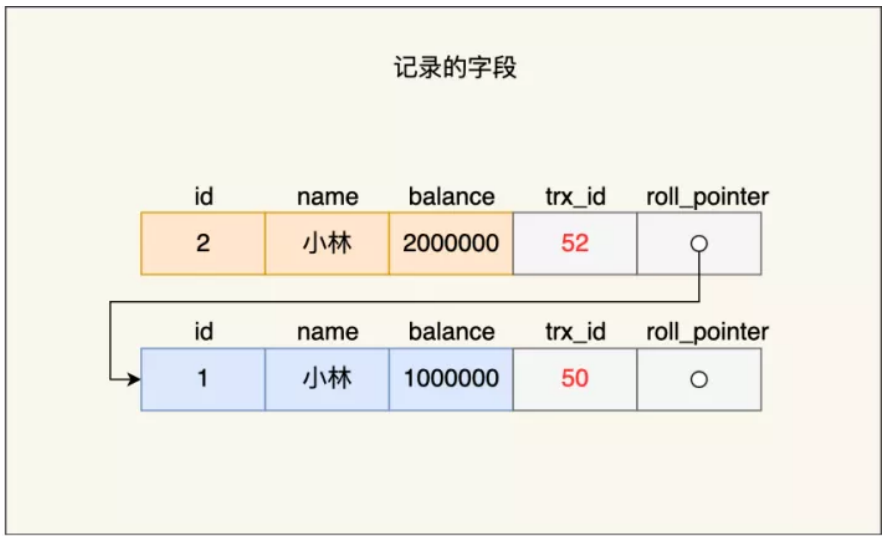
- 索引字段是字符串，但查询时不加单引号
- NOT IN、NOT EXISTS导致索引失效
-

事务

- Conception：
  a. 四个特性（ACID）
  b. 并发事务访问相同记录的情况（读读，读写写读，写写）
  c. 脏写脏读、不可重复读、幻读
  d. 4个隔离级别
  e. 当前读、快照读
  f. MVCC
  g. READ VIEW
- Question：
  a. 事务是什么？事务的四个特性？
     - 事务：一系列操作必须全部完成，有一个失败则全部失败，数据库并发控制的基本单位
     - 原子性：要么全部执行，要不全不执行（undolog实现）
     - 一致性：**事务前后数据完整性必须一致（原子性+持久性+隔离性实现，eg：转账的例子）**
     - 隔离性：各个事务之间不能被互相干扰（MVCC+锁）
     - 持久性：事务完成后改变是永久性的（redolog实现）
  b. 事务四种隔离级别和三种读的问题解决了哪些？
     - 事务隔离机制的实现基于：锁+MVCC
     - 读未提交：能够读到没有提交的数据
     - 提交读：一个事务只能读到已经提交的数据（每条语句前生成一个READ VIEW）
     - 可重复读：同一事务内，任意时刻读到的数据是一样的（事务开启前生成一个READ VIEW）
     - 串行化：不管多个事务，都是按序一个一个执行（分布式事务）

	脏读	**不可重复读**	**幻读**	**如何实现**
读未提交	可能	可能	可能	直接读取最新版本
提交读	不可能	可能	可能	MVCC
可重复读	不可能	不可能	可能	MVCC
串行化	不可能	不可能	不可能	加锁

c. 说说脏读、不可重复读、幻读?
   - 脏读：读到了未提交的数据，回滚后读到的数据不一致
   - 不可重复读：针对修改，前后读到的数据不一致
   - 幻读：针对增加、删除，前后读到的行数不一样
d. 事务的实现原理/MySQL事务日志说一下?
   - 基于redo log和undo log
   - redo log：先将事务的所有操作写入redo log，保证原子性和持久性
   - undo log：可以撤销到事务开始前的状态，保证一致性
e. 在事务中可以混合使用存储引擎吗?
   - MySQL服务器不管理事务，存储引擎层实现事务
   - 尽量不要使用
   - 需要回滚时会出现问题
   - 为每张表选择合适的存储引擎非常重要
f. **MVCC?**
   - 多版本并发控制（Multi-Version Concurrency Control）
   - 用处：用于支持提交读和可重复读的实现，**使得读写操作没有冲突**
   - 实现：
     - **Read View + undo日志 + 隐藏字段回滚指针**
     - **通过read-view机制与undo版本链进行对比，提供不同版本的数据**
   - **Read View和数据中的隐藏列**进行比对，决定是否可读



   - 事物的READ VIEW　　聚簇索引的隐藏字段
g. 当前读、快照读?
   - **当前读**：读取的是记录的**最新版本**；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
   - 当前读通过 next-key 锁(行记录锁+间隙锁)；适用于 insert，update，delete， select ... for update， select ... lock in share mode 语句，以及加锁了的 select 语句
   - **快照读**：快照读可能读到的并不一定是数据的最新版本，而有可能是之前的历史版本
   - 快照读基于 MVCC 和 undo log 来实现，适用于select语句
h. Read View
   - **MVCC内部使用的一致性读快照/读视图**称为Read View

- READ VIEW有四个数据，有一个版本链（创建readview时候活跃的事务：在动的事务）
- 当前状态下访问某个数据，需要将其版本号与readview中的版本号进行对应，从而决定能不能访问
- 自己的最新的更新总是可见
- 版本未提交：不可见
- 版本已提交，但是是在视图创建后提交的：不可见
- 版本已提交，而且是在视图创建前提交的：可见
- 讲解
- **RC隔离级别：每次读取数据前，都生成一个readview；**
- **RR隔离级别：在事务开启前，生成一个readview；**
- READ COMMITTED：只承认在语句启动前就已经提交完成的数据
- REPEATABLE READ：只承认在事务启动前就已经提交完成的数据

i. **可重复读解决幻读了吗？**

- 解决了读数据的幻读，没解决修改数据的**幻写(update)**
- for update是当前读（幻写）
- 要解决就要串行化/**MVCC + next-key锁，锁住间隙，不允许再插入**

j.

日志

- Conception：
  a. Write Ahead Loging技术
  b. 逻辑日志、物理日志
  c. redo log
  d. undo log
  e. bin log
  f. relay log、slow query log、error log
  g. gereral log
  h. 两阶段提交
  i. 二进制日志（binary log）
  j. double write buffer
- Question：
  a. WAL是什么？有什么好处？
    - Write-Ahead Logging：所有的修改都先被写入到日志（log）中，然后再写磁盘。如果事务失败，WAL中的记录会被忽略，如果成功，某个时间点写回到数据库文件，提交修改。
    - 保证原子性、持久性
    - 好处：读写可以并发执行，不会相互阻塞（写写依旧阻塞）；先写log再写磁盘，随机写变为数据写，IO次数降低；可以用日志恢复磁盘数据
  b. bin log是什么？redo log是什么？undo log是什么？relay log是什么？

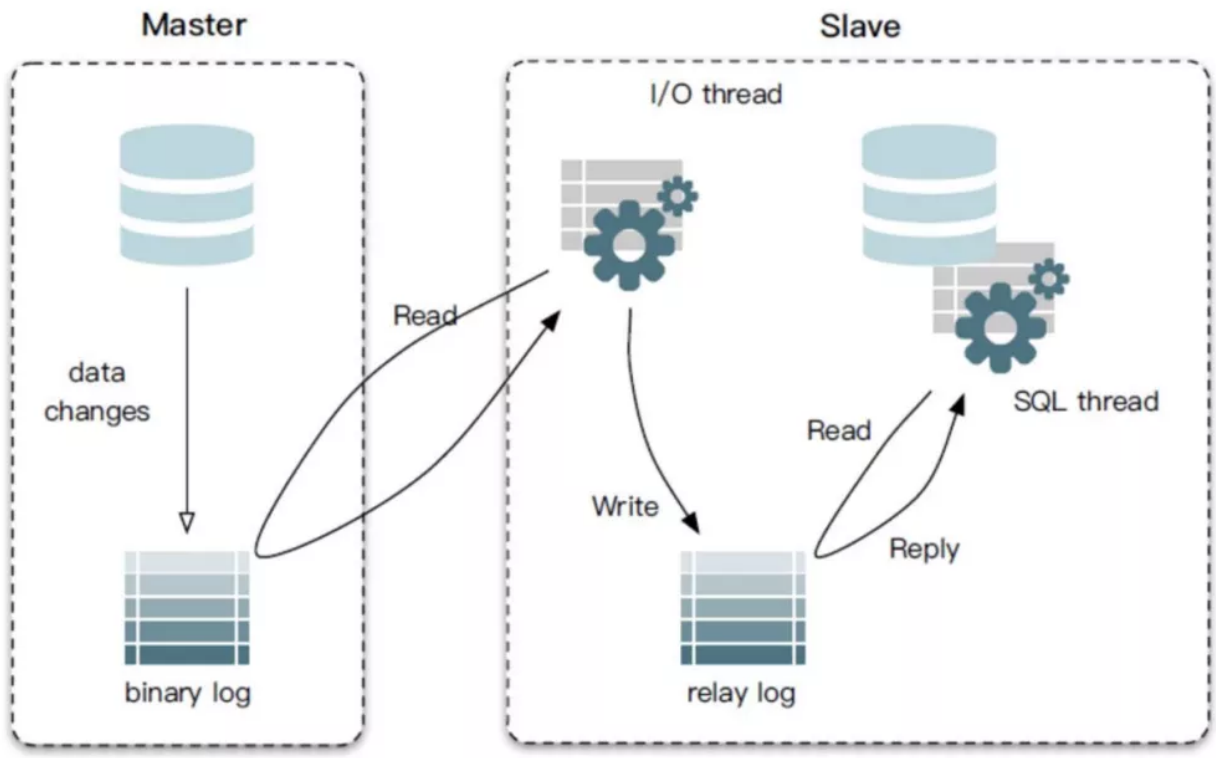
	作用	
binlog	主从复制	
redolog	持久性	先写日志再落盘，保证持久性
undolog	原子性	
relay log	中继日志	

	bin log	**redo log**
日志类型	逻辑日志（记的是SQL语句）	物理日志（记的是修改后的数据）
写的方式	追加写，不会覆盖	循环写，空间会被用完
实现层	Server层实现，所有引擎都可以	InnoDB特有
	没有 crash-safe 的能力，只能用于归档	保证crash-safe能力（进程异常重启，系统 redolog，将未写入MySQL的数据恢复进去
写入时机	事务提交时一次性写入	事务开始就逐步写入
作用	**主从复制**、数据恢复（恢复到某个时间点）	确保事务的持久性（redo log来记录已成功 的修改信息，redolog把没有持久化到磁盘 行恢复）
tips		mysql闲下来的时候才将redolog写入到磁

- binlog三种格式：statement（基于SQL语句的模式）、row（记录的是行的变化）、mixed（根据语句选择是statement还是row）
- undo log：InnoDB 存储引擎的日志，保证原子性，记录事务发生前的数据的一个版本，用于回滚，提供MVCC；主要作用：事务回滚、MVCC

redo log	**undo log**
数据修改前先写入redo log，断电意外之后，可以继续完成	意外之后撤销回之前的状态
物理日志	逻辑日志
保证事务原子性和持久性	保证一致性

- relay log：中继日志，主从复制时候用到，中介临时的日志文件，存储master节点同步过来的binlog日志内容



- binlog dump线程
- slave I/O线程
- slave SQL线程

c. 两阶段提交？Innodb事务为什么要两阶段提交？

- update T set c=c+1 where id=2;
- 查找id=2，加载到缓存中
- 将c+1并写进数据库，再更新到内存
- 写入redolog（prepare状态）
- 生成binlog，把binlog写入磁盘
- 提交事务，redolog改为commit状态
- 更新完成
- 为什么两阶段提交：
- redolog是恢复没有持久化到磁盘的操作，binlog是恢复到此前的某一状态；
- 先写redo再写bin：进程异常重启，redolog重新恢复把0变为1，binlog没有记录，之后恢复时候还是0（**恢复时候用binlog，进行恢复时候会少内容**）
- 先写bin再写redo：redolog没有记录，这一行为0，binlog之后某次恢复时候恢复成了1（**事务回滚时候用redolog，redolog没有记录，事务无效并没有回滚着一行**）

d. **二进制日志记录了什么信息（binary log）？**

- 记录修改或可能引起数据变更的MySQL语句，不记录select和show语句
- 还记录语句发生时间、执行时长、操作数据等信息

e. **刷脏页，抖动?**

- 脏页：内存数据页跟磁盘数据页内容不一致的时候
- 抖动：写内存和日志，“抖”的瞬间，就是在刷脏页
- flush场景：
  - redolog写满了，停止所有操作，checkpoint往前推进，redo log留出空间继续写
  - 系统内存不足，淘汰数据页
  - MySQL 认为系统“空闲”，刷一些脏页

- MySQL 正常关闭，把内存的脏页都 flush 到磁盘上，下次 MySQL 启动的时候，就可以直接从磁盘上读数据，启动速度会很快

锁

- Conception：
  a. 全局锁
  b. 粒度分：
  c. 表锁（表锁、MDL、意向锁、AUTO-INC锁）
  d. 行锁（Record Lock记录锁、Gap Lock间隙锁、Next-key Lock临键锁）
  e. 页锁
  f. 属性/类别分：
  g. 读锁|共享锁、写锁|排他锁
  h. 乐观锁、悲观锁
  i. 死锁
- Question：
  a. 锁的分类?
    - 粒度分：

行锁	表锁	页锁
粒度最细	粒度最大	粒度介于行级锁和表级锁
会出现死锁	不会出现死锁	会出现死锁
加锁开销大，并发度高	加锁开销小，并发度低	介于两者之间，并发度一般

    - 类别分：

共享锁/读锁	排他锁/写锁
可以同时加多个	只能加一个

乐观锁	悲观锁
尽可能直接做下去，提交时候才去锁定	先取锁再访问（分为共享锁和排他锁）
假设不会发生冲突	假设会发生冲冲突
多读场景	多写场景
省去锁的开销，加大吞吐量	降低了性能

    - 讲解
  b. 表级锁?

	表锁	MDL（MetaData Lock）	意向锁	自增锁AUT
含义		对数据库表操作时候，自动给这个表加MDL，事务提交后释放	向数据库中的行加共享/独占锁之前，数据库自动申请表的意向锁	专门针对事 AUTO_INCR 列，一个事 入记录，所 入必须等待 务插入的行 值
作用			保证用户对当前表操作时，防止其他线程对表结构的改变	加速判断表 加锁
使用方法	lock table xxx read/write;		select ... lock in share mode / for update;（检索行时的语句，数据库自己加意向共享锁 / 意向独占锁）	innodb_au e配置，，可 锁的模式与

c. 行级锁?

	Record Lock	Gap Lock	Next-key Lock
含义	记录锁	锁定一个范围，不包含记录本身	Record Lock + Gap Lock锁定一个范围并且锁定记录本身
区间		前开后开	前开后闭

d. 行锁怎么实现的?

- 通过给索引上的索引项加锁来实现的
- InnoDB这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！

e. 为什么加锁?

- 并发情况下产生多个事务同时存取同一数据的情况，不加控制就会产生读取或存储不正确，破坏一致性
- 保证多用户环境下保证数据库完整性和一致性

f. 乐观锁的实现方式?

- 版本号机制：多一个version字段，必须版本号一制才能更新数据
- CAS（compare and swap）：

g. InnoDB行锁怎么实现的?

- 通过给索引上的索引项加锁来实现
- 这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！（不走索引时候加的是表锁）

h. **什么场景下产生间隙锁?**

- eg场景：试图去锁定某一条数据，用的是for update，这一条数据不存在，此时产生的什么锁（间隙锁）

i. 死锁?

- 多个事务在同一资源上互相占用，锁定对方请求的资源，恶性循环
- 解决死锁方法：一个事务中尽可能一次锁定所需要的所有资源；升级锁的粒度，通过表级锁减少死锁产生概率；不同程序并发使用多个表，尽量约定以相同的顺序

j. 隔离级别与锁的关系？

- 读未提交：不加共享锁
- 读提交：加共享锁，语句执行完后释放
- 可重复读：加共享锁，事务完成前不释放
- 串行化：锁定整个范围的键，一直持有锁直到事务完成

k. 优化锁的意见？

- 设计索引，尽量用索引访问数据，加锁更加准确
- 事务大小合理，一次性请求足够的锁
- 查询时候不是必要不要加锁
- 不同程序并发使用多个表，尽量约定以相同的顺序
- 特殊情况用表锁避免死锁

键

- Conception：

a. 超键、候选键、主键、外键

- Question：

a. 说说超键、候选键、主键、外键？

- 超键：能唯一标识元组的属性集
- 候选键：最小超键
- 主键：关系模式中用户正在使用的候选键、一个数据列只能有一个主键，且不能缺失不能为空值（NULL）
- 外键：当前表中存在的另一个表的主键

b. 为什么推荐使用自增 id 作为主键？

- 普通索引的B+树上叶子节点存的是主键索引的值，会导致索引文件存储空间大
- 自增id新增时直接插入到页尾，顺序插入即可，减少页分裂维护的成本

c. **MySQL的自增id是严格自增的吗？**

- 文章
- 5.7之前：每次去找自增值的最大值max(id)，将max(id)+1作为这个表当前的自增值
- 8.0之后
  - 自增值的变更记录在了redo log中
    - 唯一键冲突会导致主键id不连续
    - 事务回滚也会导致主键id不连续

分区分库分表

- Conception：

a. 分区

b. 分表（垂直分表、水平分表）

c. 分库（垂直分库、水平分库）

- Question：

a. 为什么分表？为什么分库？

  - 单表数据量太大，根据某个列查询对对应的表操作即可
  - 用户请求并发量太大，访问一个库即可

b. 垂直拆分？水平拆分？

  - 垂直拆分：每个库表的结构不同；将访问频率低的放到一个表中，访问频率高的放到另一个表中，缓存一次性命中的越多
  - 垂直拆分缺点：查询所有数据需要join操作
  - 水平拆分：每个库的表结构相同，数据不同
  - 水平拆分缺点：查询所有数据需要UNION

c. 分库分表方式？

  - 按照range范围划分：少用，容易产生热点问题
  - 某个字段hash之后均匀分散：常用

d. 分表算法？

  - 中间表方法：中间表存id和库的对应关系
  - 求余取模
  - 范围方式
  - 一致性hash（哈希环）
  - 讲解

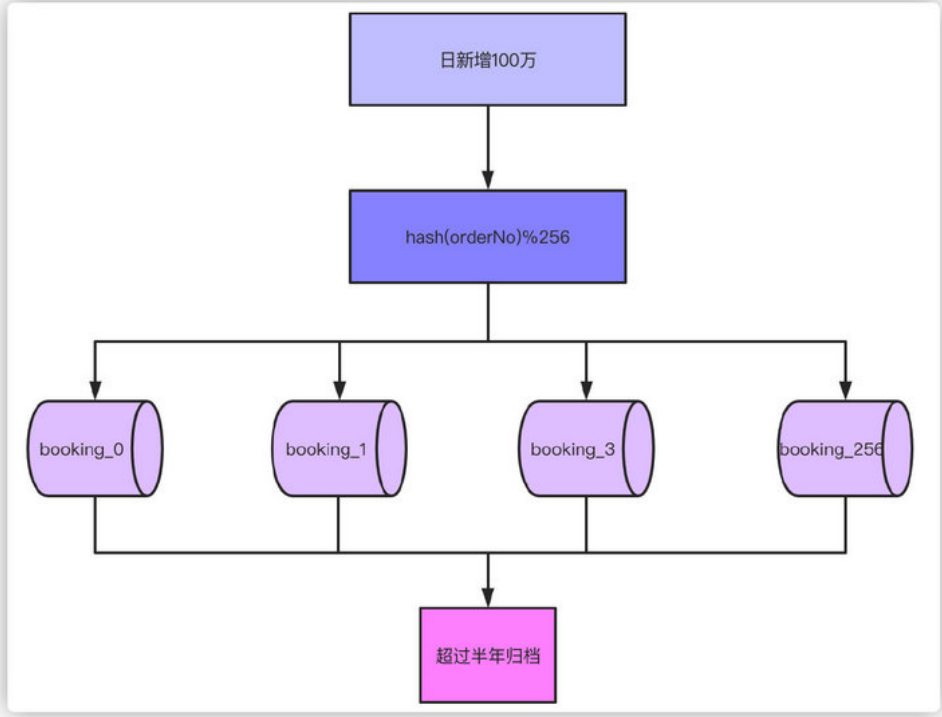
e. 分库分表中间件？

  - cobar（阿里）、mycat（基于cobar改造）、atlas（360开源）

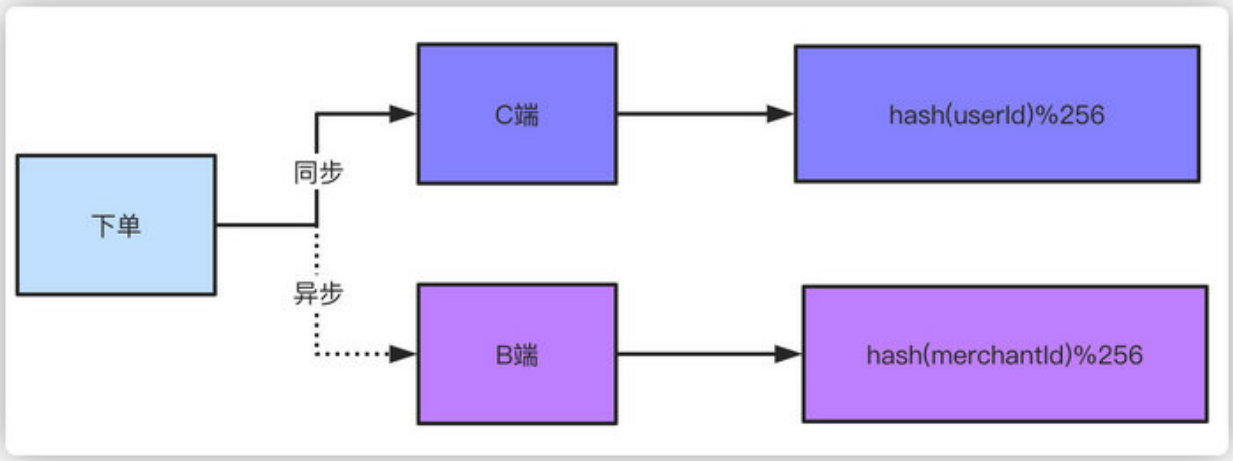
f. **分库分表后如何进行一个查询操作？**

  - 好文
  - 主键id如何生成：雪花算法Snowflake、滴滴Tinyid、美团Leaf

  - 分库分表



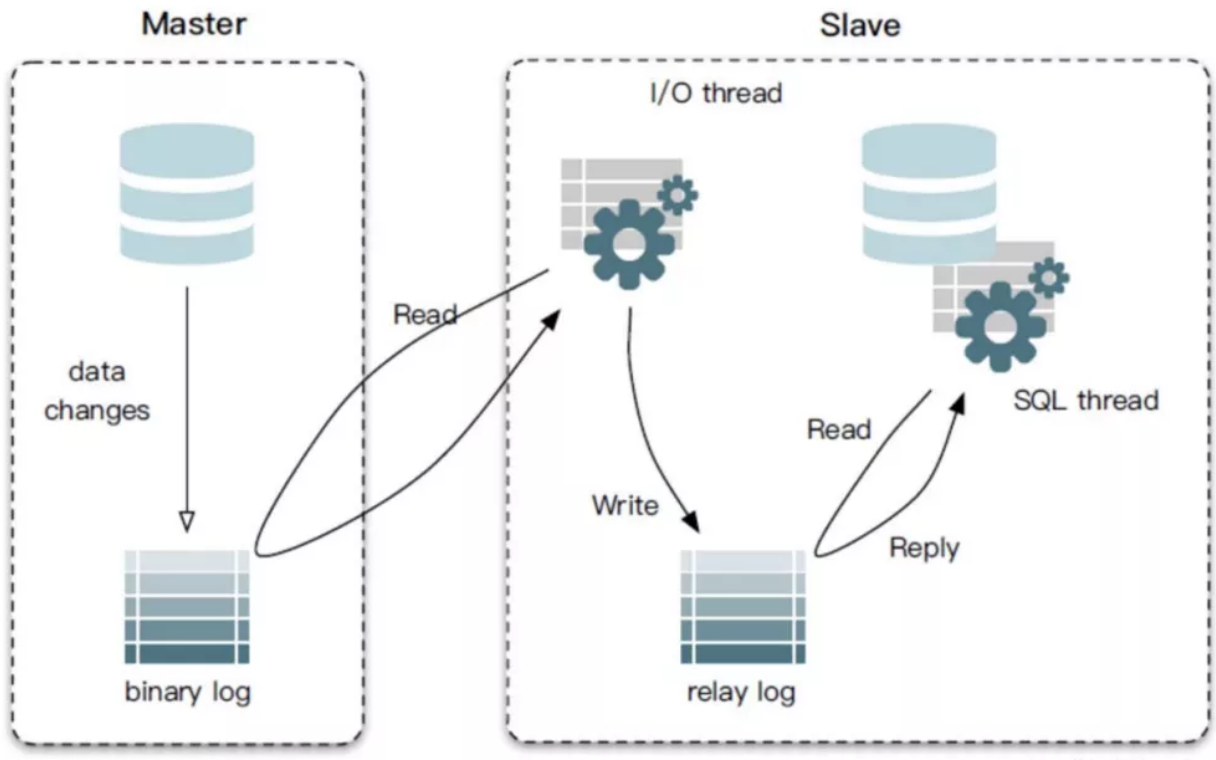
- 双写、es
- C端查询
- B端查询：可以接受延迟



g.

主从复制

- Conception：
  a. 主从复制
  b. 读写分离
  c. 主从延迟
- Question：
  a. 主从如何同步数据？（讲讲那个图）



- master将更新的时间写入到binlog中
- master创建log dump线程通知slave需要更新数据
- slave发起请求，将binlog内容存到本地relaylog中
- slave开启SQL线程执行relaylog中的内容，完成主从同步

b. 同步策略

- 全同步复制：主库强制同步日志到从库，等全部从库执行完才返回客户端，性能差
- 半同步复制（semi-sync）：主库收到至少一个从库确认就认为操作成功，从库写入日志成功返回ack确认

c. 主从同步目的/作用

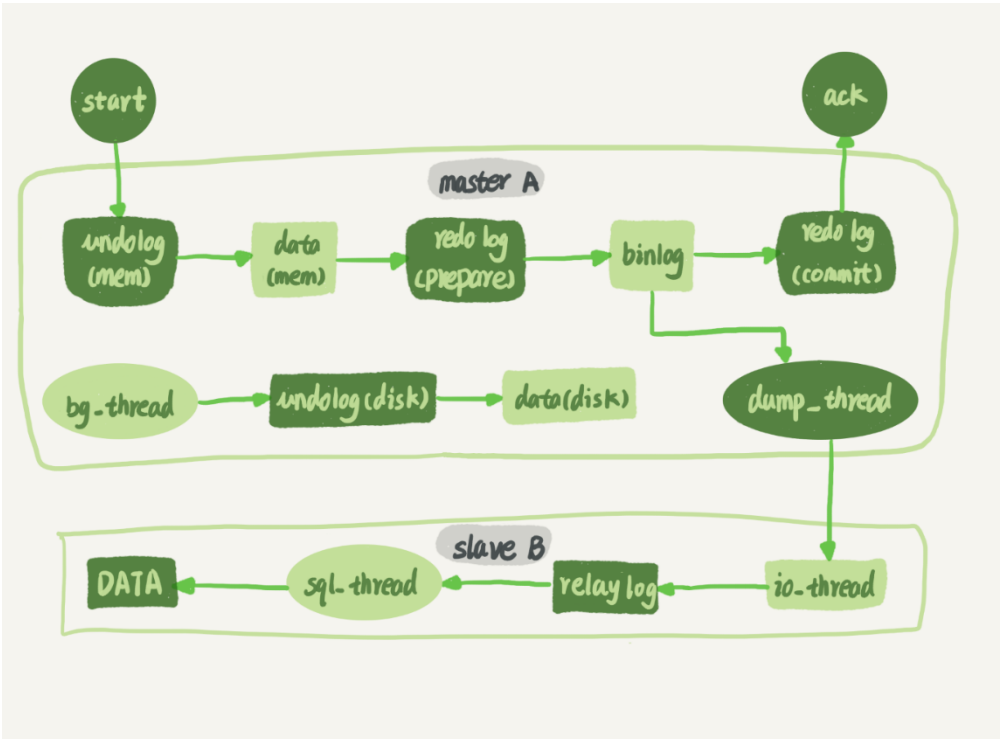
- 读写分离
- 数据备份
- 高可用：主服务器宕机，从服务器充当主服务器，保证正常运行

d. 主从复制涉及到哪三个线程?

- bin log线程：主库记录改变放入bin log中
- IO线程：丛库从bin log拉取内容到relay log中
- sql执行线程：从库执行relay log同步数据

e. **MySQL主从如何保证强一致性?**

- 涉及到的东西：两阶段提交，binlog三种格式



- binlog写入主库从库都成功，redolog才会进入commit状态，最后才会ack
- 问题：binlog如何写入
  - statement：记录的是原始语句；row：记录的是行的变化；mixed：根据需要进行选择，因为statement存在unsafe问题，row存在格式太大问题，所以要自行进行选择
- 问题：双M架构下循环复制问题
  - 节点A和节点B之间互为主备关系，互为slave，存在循环复制问题
  - 解决：binlog中会记录server id，每个库在收到主库发过来的binlog日志时，先判断server id，如果与自己的相同说明是自己生成的，就会直接丢弃这个日志

f. 主从延迟原因？怎么解决？

- 并行复制：MySQL 5.6 版本以后，提供了一种并行复制的方式，通过将 SQL 线程转换为多个 work 线程来进行重放（解决主从同步时延问题）
- 半同步复制：主库收到至少一个ack就认为写操作完成（解决主库数据丢失问题）
- 提高机器配置
- 避免单表单库太大
- 避免长事务
- 避免让数据库做大量运算
- 对延迟敏感的业务，直接用主库读

问题

- Question：

a. 为什么一条sql语句查询一直慢？

- 没有用到索引，索引失效了
- 优化器用错了索引，force index强制走索引
- 数据量太大

b. 为什么一条sql语句查询偶尔慢？

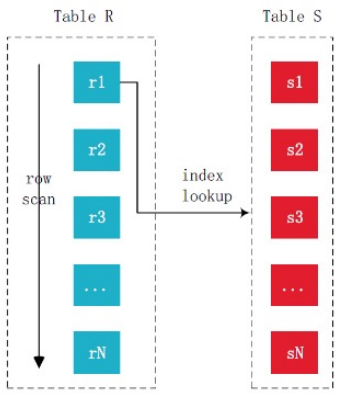
- 刷新脏页（内存数据页和磁盘数据页内容不一致）：刷脏页场景：redolog写满了，停止更新操作；系统内存不足，淘汰数据页；空闲时自动刷脏页
- 没有拿到锁，在等待锁

c. 删除数据表大小没变?
   - 删除数据行并不是真正的删除，是逻辑删除，InnoDB 仅仅是将其标记成可复用（只限于符合范围条件的数据复用）
   - 删掉一整页，整个数据页可以被复用（任何位置都可以复用）
d. 为什么不要使用长事务?
   - 并发情况下，数据库连接池容易被撑爆
   - 占用锁资源，造成阻塞和超时
   - 造成主从延迟
   - 回滚时间长
   - undolog越来越长
e. 数据库CPU飙升到500%怎么处理?
   - linux top命令查看是不是MySQL导致
   - 如果是MySQL造成，show processlist找出消耗高的sql，看看是不是索引缺失或者数据量太大
   - kill调这些线程，调整后重跑
   - 限制并发连接数
f. 如何定位及优化SQL语句的性能问题?
   - 使用执行计划，EXPLAIN命令
g. 超大分页怎么处理?
   - 靠缓存，可预测性的提前查到内容，缓存到redis
h. 慢查询？怎么优化?
   - 分析语句
   - 分析语句的执行计划
   - 表中数据太大可以横向或纵向分表
i. SQL语句执行慢的查询方法
   - show profile（查看资源消耗情况）
   - show status（查看状态）
   - show full processlist \G（看是否有大量线程处于不正常的状态）
   - explain（查看）

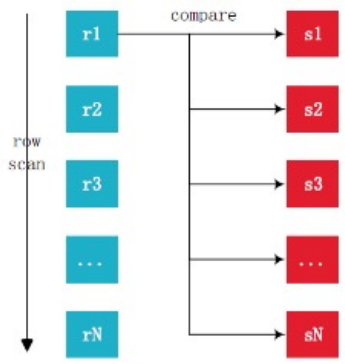
优化

- Conception：
  a. join
  b. 深分页
  c. 数据库连接池
- Question：
  a. **join相关?**
     - **驱动表是走全表扫描，而被驱动表是走树搜索。**时间复杂度近似于N + N*2*log2M，所以小表驱动大表，小表join大表

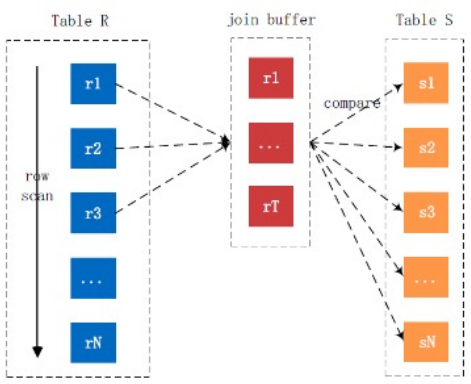
- 用的上索引的时候
  - INLJ（Index Nested-Loops Join）



- **被驱动表用不上索引**时候：
  - NLJ（Simple Nested-Loop Join 算法）算法：



  - BNL（Block Nested-Loop Join）算法：



    - 把表 t1 的数据读入线程内存 join_buffer 中，扫描表 t2，把表 t2 中的每一行取出来，跟 join_buffer 中的数据做对比
    - 这两个算法是一样的。但是Block Nested-Loop Join 算法的这 10 万次判断是**内存**操作，速度上会快很多，性能也更好
  -

- 优化：
  - **被驱动表**要有**索引**
  - **小表驱动大表**更好
- 极客时间讲解
- 文章

b. **深分页**

- LIMIT 100000, 1 的意思是扫描满足条件的 100001 行，**然后扔掉前 100000 行**
- 减少回表
- ORDER BY 索引失效
- 文章

c. **数据库连接池**
    - golang实现
        - 切片保存
        - 建立链接：
            - 没有时候新建
            - 切片有的时候拿出来最后一个
        - 释放链接：
            - 释放回切片中
        - https://github.com/golang/go/blob/master/src/database/sql/sql.go
d. COUNT(*)和各种COUNT()效率比较
    - COUNT(*)：不取值，按行累加
        - MyISAM存了表行数，InnoDB因为有了MVCC，所以只能一行一行读
    - COUNT(1)：遍历整张表，但不取值，对于返回的每一行，放一个数字1进去，判断不可能为空，就按行累加
    - COUNT(主键id)：遍历整张表，把每一行的id取出来，返回给server层。拿到id后，判断不可能为空，就按行累加
    - COUNT(字段)：如果字段定义为not null，一行行地从记录里面读出这个字段，判断不能为 null，按行累加；如果这个“字段”定义允许为 null，那么执行的时候，判断到有可能是 null，还要把值取出来再判断一下，不是 null 才累加。
    - count(*) ≈ count(1) > count(主键id) > count(字段)
e. 说说sql调优思路?
    - 索引：建索引时候选择合适字段；利用好索引下推，覆盖索引等功能；普通/唯一索引
    - 查询：避免索引失效；force index防止选错索引；小表驱动大表
    - 表结构：类型、大小选择；合理的增加冗余字段；新建字段一定要有默认值
    - 分库分表
f. 大表数据查询怎么优化?
    - 优化sql语句，用索引
    - 加缓存：memcached、redis
    - 主从复制、读写分离
    - 水平切分、垂直切分
g. 数据库设计方案优化?
    - 字段多的进行分表
    - 增加中间表
    - 增加冗余字段
h. 关联查询如何优化?
    - ON或者USING子句中是否有索引
    - GROUP BY和ORDER BY只有一个表中的列，这样MySQL才有可能使用索引
i. 如何优化查询过程中的数据访问?
    - 避免使用SELECT *返回全部列

- 避免检索大量超过需要的数据（太多列）
- 可以缓存数据，下次直接读取缓存
- 改变表的结构

j. 慢查询如何定位（explain详解）

- 不会看 Explain执行计划，劝你简历别写熟悉 SQL优化
- Extra列：索引、临时表、锁

k. **MySQL查询优化**

- 不要用select * ，减少访问的列
- COUNT(*)或者COUNT(1)而不是COUNT(列名)
- 查询缓存，避免直接打到数据库
- 用Union优化OR语句，OR可能导致索引失效
- 避免使用前置通配符，%abc
- 使用最左匹配原则，避免最左匹配原则失效
- EXISTS和IN使用场景（大表小编）

l. **MySQL性能优化经验**

- 为查询缓存优化你的查询，使用查询缓存
- EXPLAIN 你的 SELECT 查询
- 知道只会有一条结果，使用LIMIT 1
- 为搜索字段建索引
- 千万不要 ORDER BY RAND()
- 永远为每张表设置一个ID，设置上自动增加的AUTO_INCREMENT标志
- 使用 ENUM 而不是 VARCHAR
- 尽可能的使用 NOT NULL
- 二十种实战调优MySQL性能优化的经验
- 一文搞定MySQL性能调优

安全

- Conception：

a. SQL注入

一些其他概念

- Conception：

a. mysql 的sql 类型

- 一、数据定义语言 DDL：Create、Drop、Alter 操做。用于定义库和表结构的。sql
- 二、数据查询语言 DQL：select。用于查询数据的。数据库
- 三、数据操纵语言 DML：insert、update、delete。对行记录进行增删改操做。session
- 四、数据控制语言 DCL：grant、revoke、commit、rollback。控制数据库的权限和事务
- DDL, DML, DCL, TCL

b. T-SQL