▶ **一张表，里面有 ID 自增主键，当 insert 了 17 条记录之后，删除了第 15,16,17 条记录， 再把 Mysql 重启，再 insert 一条记录，这条记录的 ID 是 18 还是 15 ？**

如果表的类型是MylSAM，那么这条记录的ID就是18，因为MylSAM表会把自增主键的最大ID记录到数据文件里面，重启MYSQL后，自增主键的最大ID也不会丢失

如果表的类型是InnoDB，这条记录的ID是15。因为InnoDB表只把自增主键的最大ID记录到内存中，所以重启数据库或者对表OPTIMIZE操作，都会使最大ID丢失。

注：如果在这17条记录里面删除的是中间的几个记录（比如删除的是10,11,12三条记录），重启MySQL数据库后，insert一条记录后，ID都是18。因为内存或者数据库文件存储都是自增主键最大ID

▶**如何区分 FLOAT 和 DOUBLE？**

以下是 FLOAT 和 DOUBLE 的区别： 

浮点数以 8 位精度存储在 FLOAT 中，并且有四个字节。 

浮点数存储在 DOUBLE 中，精度为 18 位，有八个字节。

▶**索引失效的几种情况（MySQL）**

1.前导模糊查询不能利用索引(like '%XX'或者like '%XX%')

假如有这样一列code的值为'AAA','AAB','BAA','BAB' ,如果where code like '%AB'条件，由于前面是模糊的，所以不能利用索引的顺序，必须一个个去找，看是否满足条件。这样会导致全索引扫描或者全表扫描。如果是这样的条件where code like 'A % '，就可以查找CODE中A开头的CODE的位置，当碰到B开头的数据时，就可以停止查找了，因为后面的数据一定不满足要求。这样就可以利用索引了。

2.如果是组合索引的话，如果不按照索引的顺序进行查找，比如直接使用第三个位置上的索引而忽略第一二个位置上的索引时，则会进行全表查询

索引为c1,c2,c3,c4



上图结果显示直接使用c3是全表查询，无法使用该索引的，所以c3字段使用索引的前提是c1，c2两字段均使用了索引。

3.条件中有or

应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num=10 or num=20

可以这样查询：

select id from t where num=10 union all select id from t where num=20

4.索引无法存储null值，所以where的判断条件如果对字段进行了null值判断，将导致数据库放弃索引而进行全表查询，如

select id from t where num is null

可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：

select id from t where num=0

a.单列索引无法储null值，复合索引无法储全为null的值。

b.查询时，采用is null条件时，不能利用到索引，只能全表扫描。

为什么索引列无法存储Null值？

a.索引是有序的。NULL值进入索引时，无法确定其应该放在哪里。（将索引列值进行建树，其中必然涉及到诸多的比较操作，null 值是不确定值无法比较，无法确定null出现在索引树的叶子节点位置。）

5.应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

6.in 和 not in 也要慎用，否则会导致全表扫描，如：

select id from t where num in(1,2,3)

对于连续的数值，能用 between 就不要用 in 了：

select id from t where num between 1 and 3

7. 应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

–name以abc开头的id

select id from t where substring(name,1,3)='abc'

8.应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where num/2=100

应改为:

select id from t where num=100\*2

▶**mvcc原理**

MVCC (Multiversion Concurrency Control) 中文全程叫多版本并发控制，是现代数据库（包括 MySQL、Oracle、PostgreSQL 等）引擎实现中常用的处理读写冲突的手段，目的在于提高数据库高并发场景下的吞吐性能

多版本控制MVCC，也就是Copy on Write的思想。MVCC除了支持读和读并行，还支持读和写、写和读 的并行，但为了保证一致性，写和写是无法并行的。

一、*MVCC概念*

MVCC（Multi Version Concurrency Control）被称为多版本控制，是指在数据库中为了实现高并发的 数据访问，对数据进行多版本处理，并通过事务的可见性来保证事务能看到自己应该看到的数据版本。 多版本控制很巧妙地将稀缺资源的独占互斥转换为并发，大大提高了数据库的吞吐量及读写性能。

如何生成的多版本？每次事务修改操作之前，都会在Undo日志中记录修改之前的数据状态和事务号， 该备份记录可以用于其他事务的读取，也可以进行必要时的数据回滚。

二、*MVCC实现原理*

MVCC大的好处是读不加锁，读写不冲突。在读多写少的系统应用中，读写不冲突是非常重要的，极 大的提升系统的并发性能，这也是为什么现阶段几乎所有的关系型数据库都支持 MVCC 的原因，不过目 前MVCC只在 Read Commited 和 Repeatable Read 两种隔离级别下工作。 在 MVCC 并发控制中，

读操作可以分为两类: 快照读（Snapshot Read）与当前读 （Current Read）。

快照读：读取的是记录的快照版本（有可能是历史版本），不用加锁。（select）

当前读：读取的是记录的新版本，并且当前读返回的记录，都会加锁，保证其他事务不会再并发 修改这条记录。（select... for update 或lock in share mode，insert/delete/update

▶**数据库的隔离级别**

MySQL 默认为 ：REPEATABLE\_READ；

Oracle，sql server 默认为：READ\_COMMITTED；



▶**Mysql 表中允许有多少个 TRIGGERS？**

在 Mysql 表中允许有六个触发器，如下： 

BEFORE INSERT 

AFTER INSERT 

BEFORE UPDATE 

AFTER UPDATE 

BEFORE DELETE and 

AFTER DELETE

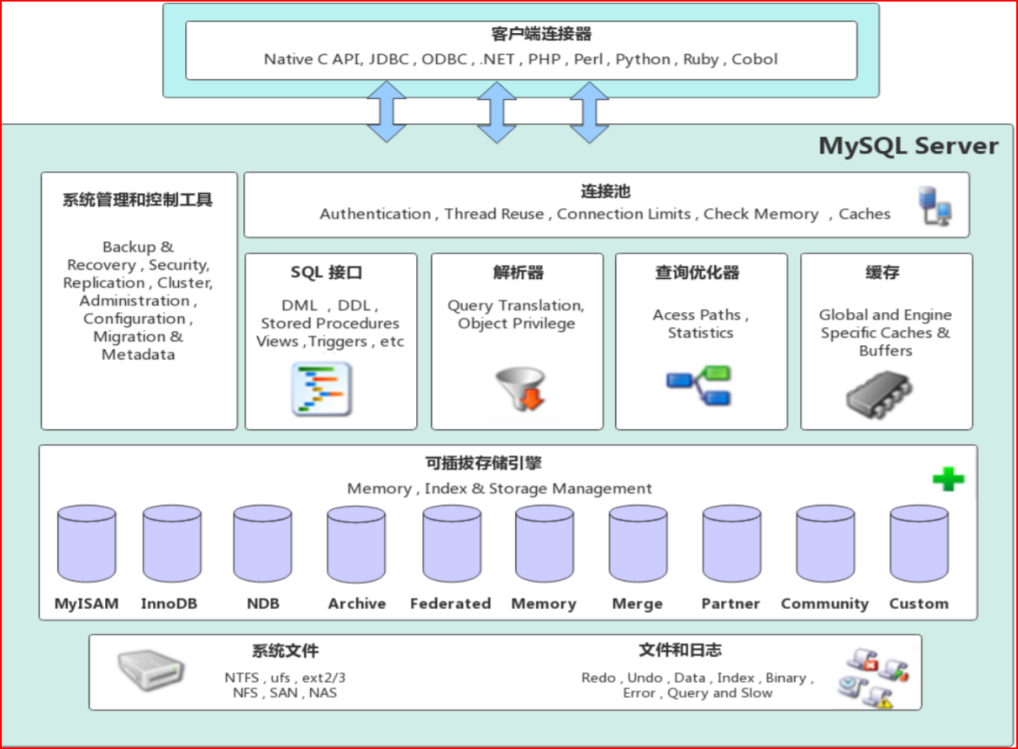
▶**Mysql 中有哪几种锁？**

MyISAM 支持表锁，InnoDB 支持表锁和行锁，默认为行锁 

表级锁：开销小，加锁快，不会出现死锁。锁定粒度大，发生锁冲突的概率最高，并发量最低 

行级锁：开销大，加锁慢，会出现死锁。锁力度小，发生锁冲突的概率小，并发度最高

▶**MySQL体系架构**



1. **网络连接层**

•客户端连接器（Client Connectors）：提供与MySQL服务器建立的支持。

1. **服务层（MySQL Server）**

•**连接池（Connection Pool）**：负责存储和管理客户端与数据库的连接，一个线程负责管理一个连接。

•**系统管理和控制工具（Management Services & Utilities）**：例如备份恢复、安全管理、集群 管理等

•**SQL接口（SQL Interface）**：用于接受客户端发送的各种SQL命令，并且返回用户需要查询的结 果。比如DML、DDL、存储过程、视图、触发器等。

•**解析器（Parser）**：负责将请求的SQL解析生成一个"解析树"。然后根据一些MySQL规则进一步 检查解析树是否合法。

•**查询优化器（Optimizer）**：当“解析树”通过解析器语法检查后，将交由优化器将其转化成执行计划，然后与存储引擎交互。

•**缓存（Cache&Buffer）**： 缓存机制是由一系列小缓存组成的。比如表缓存，记录缓存，权限缓 存，引擎缓存等。如果查询缓存有命中的查询结果，查询语句就可以直接去查询缓存中取数据。

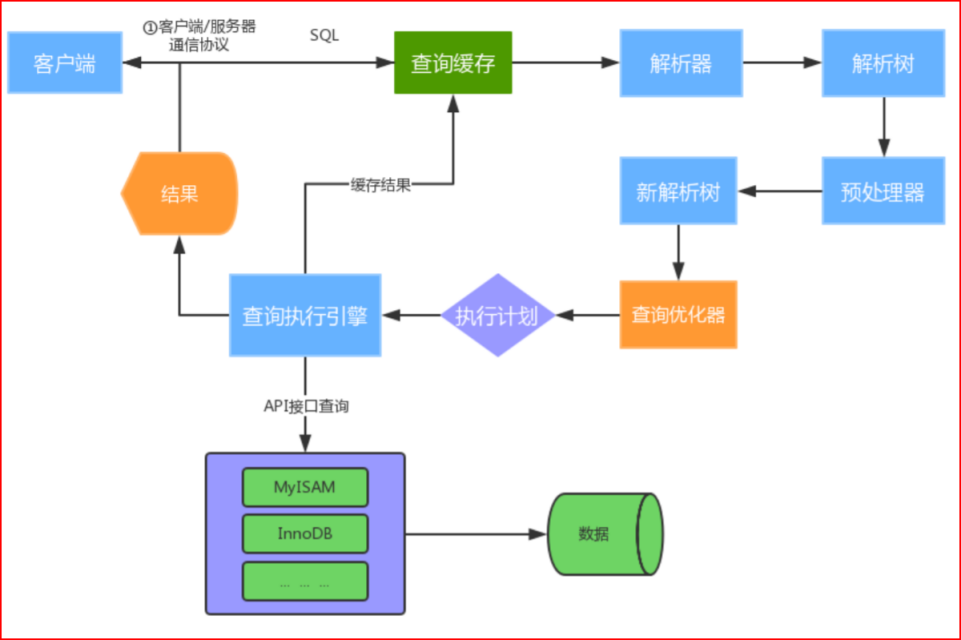
1. **存储引擎层（Pluggable Storage Engines）**

存储引擎负责MySQL中数据的存储与提取，与底层系统文件进行交互。最常见的是MyISAM和InnoDB。

1. **系统文件层（File System）**

该层负责将数据库的数据和日志存储在文件系统之上，并完成与存储引擎的交互，是文件的物理存储 层。主要包含日志文件，数据文件，配置文件，pid 文件，socket 文件等。

▶**MySQL运行机制**



•建立连接（Connectors&Connection Pool），通过客户端/服务器通信协议与MySQL建立连接。

•查询缓存（Cache&Buffer）

•解析器（Parser）将客户端发送的SQL进行语法解析，生成"解析树"。

•查询优化器（Optimizer）根据“解析树”生成最优的执行计划

•查询执行引擎负责执行 SQL 语句

Sql执行顺序 ：from---where--group by---having---select---order by

▶**MySQL存储引擎**

◇***InnoDB和MyISAM对比***

**区别：**

•**事务和外键：**InnoDB支持事务和外键，具有安全性和完整性，适合大量insert或update操作 MyISAM不支持事务和外键，它提供高速存储和检索，适合大量的select查询操作

•**锁机制：**InnoDB支持行级锁，锁定指定记录。基于索引来加锁实现。 MyISAM支持表级锁，锁定整张表。

•**索引结构：**InnoDB使用聚集索引（聚簇索引），索引和记录在一起存储，既缓存索引，也缓存记录。 MyISAM使用非聚集索引（非聚簇索引），索引和记录分开。

•**并发处理能力：**MyISAM使用表锁，会导致写操作并发率低，读之间并不阻塞，读写阻塞。 InnoDB读写阻塞可以与隔离级别有关，可以采用多版本并发控制（MVCC）来支持高并发

•**存储文件：**InnoDB表对应两个文件，一个.frm表结构文件，一个.ibd数据文件。InnoDB表最大支持64TB；

MyISAM表对应三个文件，一个.frm表结构文件，一个MYD表数据文件，一个.MYI索引文件。从 MySQL5.0开始默认限制是256TB。

◇***适用场景***

•**MyISAM**

○不需要事务支持（不支持）

○并发相对较低（锁定机制问题）

○数据修改相对较少，以读为主

○数据一致性要求不高

•**InnoDB**

○需要事务支持（具有较好的事务特性）

○行级锁定对高并发有很好的适应能力

○数据更新较为频繁的场景

○数据一致性要求较高

○硬件设备内存较大，可以利用InnoDB较好的缓存能力来提高内存利用率，减少磁盘IO

▶**MySQL日志介绍**

•**Undo Log**

意为撤销或取消，以撤销操作为目的，返回指定某个状态的操作。

作用：

○实现事务的原子性

○实现多版本并发控制（MVCC）快照读

事务A手动开启事务，执行更新操作，首先会把更新命中的数据备份到 Undo Buffer 中。

事务B手动开启事务，执行查询操作，会读取 Undo 日志数据返回，进行快照读

•**Redo Log**

顾名思义就是重做。以恢复操作为目的，在数据库发生意外时重现操作。

作用：

○在数据库发生意外时重现操作。

•**Bin Log**

Binlog是记录所有数据库表结构变更以及表数据修改的二进制日志

作用：

○主从复制：在主库中开启Binlog功能，这样主库就可以把Binlog传递给从库，从库拿到 Binlog后实现数据恢复达到主从数据一致性。

○数据恢复：通过mysqlbinlog工具来恢复数据。

▶**MySQL索引原理**

◇***索引类型***

•**从索引存储结构划分**：B Tree索引、Hash索引、FULLTEXT全文索引、R Tree索引

•**从应用层次划分**：

○**普通索引**：比如：CREATE INDEX <索引的名字> ON tablename (字段名);

○**唯一索引：**比如 CREATE UNIQUE INDEX <索引的名字> ON tablename (字段名);

○**主键索引：**比如：CREATE TABLE tablename ( [...], PRIMARY KEY (字段名) );

○**复合索引：**比如：CREATE INDEX <索引的名字> ON tablename (字段名1，字段名2...);

复合索引使用注意事项：

1、何时使用复合索引，要根据where条件建索引，注意不要过多使用索引，过多使用会对更新操作效率有很大影响。

2、如果表已经建立了(col1，col2)，就没有必要再单独建立（col1）；如果现在有(col1)索引，如果查 询需要col1和col2条件，可以建立(col1,col2)复合索引，对于查询有一定提高。

○**全文索引：**比如：CREATE FULLTEXT INDEX <索引的名字> ON tablename (字段名);

查询操作在数据量比较少时，可以使用like模糊查询，但是对于大量的文本数据检索，效率很低。如果 使用全文索引，查询速度会比like快很多倍。在MySQL 5.6 以前的版本，只有MyISAM存储引擎支持全 文索引，从MySQL 5.6开始MyISAM和InnoDB存储引擎均支持。

和常用的like模糊查询不同，全文索引有自己的语法格式，使用 match 和 against 关键字，比如：

select \* from user where match(name) against('aaa');

全文索引使用注意事项：

1、全文索引必须在字符串、文本字段上建立。

2、全文索引字段值必须在最小字符和最大字符之间的才会有效。（innodb：3-84；myisam：4- 84）

•**从索引键值类型划分**：主键索引、辅助索引（二级索引）

•**从数据存储和索引键值逻辑关系划分**：聚集索引（聚簇索引）、非聚集索引（非聚簇索引）

◇***索引原理***

•***二分查找法：***二分查找法也叫作折半查找法，它是在有序数组中查找指定数据的搜索算法。

○首先定位left和right两个指针;

○计算(left+right)/2

○判断除2后索引位置值与目标值的大小比对

○索引位置值大于目标值就-1，right移动；如果小于目标值就+1，left移动

•***Hash结构***

Hash底层实现是由Hash表来实现的，是根据键值 <key,value> 存储数据的结构。非常适合根据key查找 value值，也就是单个key查询，或者说等值查询。

•***B+Tree结构***

MySQL数据库索引采用的是B+Tree结构，在B-Tree结构上做了优化改造。

1. Tree结构:

索引值和data数据分布在整棵树结构中

每个节点可以存放多个索引值及对应的data数据

树节点中的多个索引值从左到右升序排列

B+Tree结构

非叶子节点不存储data数据，只存储索引值，这样便于存储更多的索引值

叶子节点包含了所有的索引值和data数据

叶子节点用指针连接，提高区间的访问性能

•***聚簇索引和非聚簇索引***

B+Tree的叶子节点存放主键索引值和行记录就属于聚簇索引；如果索引值和行记录分开存放就属于非聚簇索引

•***主键索引和辅助索引***

B+Tree的叶子节点存放的是主键字段值就属于主键索引；如果存放的是非主键值就属于辅助索引（二级索引）。

▶**索引分析与优化**

◇***EXPLAIN:***比如EXPLAIN SELECT \* from user WHERE id < 3;

◇***type***

•**ALL**：表示全表扫描，性能最差。

•**index**：表示基于索引的全表扫描，先扫描索引再扫描全表数据。

•**range**：表示使用索引范围查询。使用>、>=、<、<=、in等等。

•**ref**：表示使用非唯一索引进行单值查询。

•**eq\_ref**：一般情况下出现在多表join查询，表示前面表的每一个记录，都只能匹配后面表的一 行结果。

•**const**：表示使用主键或唯一索引做等值查询，常量查询。

•**NULL**：表示不用访问表，速度最快。

排序效率：system>const>eq\_ref>ref>range>index>ALL

◇**索引设计方案**

•限制每张表上的索引数量

•Innodb 表必须有个主键

•WHERE 从句中的列 和 包含在 ORDER BY、GROUP BY、DISTINCT 中的字段 需要创建索引

•索引不会包含有NULL的列

•使用短索引 如果有一个char（255）的列，如果在前10个或20个字符内，多数值是唯一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作

•对于那些定义为text、image和bit数据类型的列不应该增加索引。因为这些列的数据量要么相当大，要么取值很少。

◇**查询优化**

•慢查询定位

开启慢查询日志

•分页优化

利用子查询优化

利用回表查询

◇**慢查询原因总结**

•全表扫描：explain分析type属性all

•全索引扫描：explain分析type属性index

•索引过滤性不好：靠索引字段选型、数据量和状态、表设计

•频繁的回表查询开销：尽量少用select \*，使用覆盖索引

▶**MySQL事务**

◇**ACID 特性**

•原子性

•持久性

•隔离性

•一致性

◇**ACID靠什么保证的**

•原子性由undo log日志保证，它记录了需要回滚的日志信息，事务回滚时撤销已经执行成功的sql

•一致性一般由代码层面来保证

•隔离性由MVCC来保证

•持久性由内存+Redo log来保证，mysql修改数据同时在内存和Redo log记录这次操作，事务提交的时候通过Redo log刷盘，宕机的时候可以从Redo log恢复

◇**并发事务**

•**脏读**:一个事务读取到了另一个事务修改但未提交的数据。

•**不可重复读:**一个事务中多次读取同一行记录不一致，后面读取的跟前面读取的不一致。

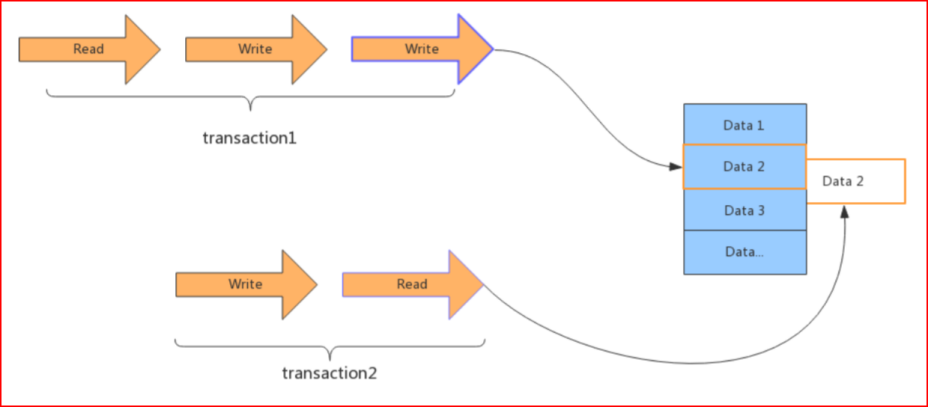
•**幻读:**一个事务中多次按相同条件查询，结果不一致。后续查询的结果和面前查询结果不同，多了或少了几行记录。

◇**锁**

•排他锁

•读写锁：读写锁就是进一步细化锁的颗粒度，区分读操作和写操作，让读和读之间不加锁

•MVCC：多版本控制MVCC，也就是Copy on Write的思想。MVCC除了支持读和读并行，还支持读和写、写和读 的并行，但为了保证一致性，写和写是无法并行的。



在事务1开始写操作的时候会copy一个记录的副本，其他事务读操作会读取这个记录副本，因此不影响其他事务对此记录的读取，实现写和读并行。

◇**MVCC**

•**3个隐式字段**

○**DB\_TRX\_ID：最近修改(修改/插入)事务ID**

○**DB\_ROLL\_PTR：回滚指针，指向这条记录的上一个版本**

○**DB\_ROW\_ID：隐含的自增ID**

•**Read View （详细内容自己查）**

○**快照的生成时机的不同, 造成了RC,RR两种隔离级别的不同可见性;**

RR隔离级别下, 事务在之后的第一条select读操作后, 会创建一个快照, 将当前系统中活跃 的其他事务记录记录起来;

RC隔离级别下, 事务中每条select语句都会创建一个快照;

○**四个重要的内容**

**﹟m\_ids:**表示在生成ReadView时当前系统中活跃的读写事务的事务id列表。

**﹟min\_trx\_id:**表示在生成ReadView时当前系统中活跃的读写事务中最小的事务id，也就是m\_ids 中的最小值。

**﹟max\_trx\_id:**表示生成ReadView时系统中应该分配给下一个事务的id值。

**﹟creator\_trx\_id:**表示生成该ReadView的事务的事务id。

○**判断版本链中的哪个版本是当前事务可见的**

**﹟**如果被访问版本的trx\_id属性值与ReadView中的creator\_trx\_id值相同，意味着当前事务在访 问它自己修改过的记录，所以该版本可以被当前事务访问。

**﹟**如果被访问版本的trx\_id属性值小于ReadView中的min\_trx\_id值，表明生成该版本的事务在当 前事务生成ReadView前已经提交，所以该版本可以被当前事务访问。

**﹟**如果被访问版本的trx\_id属性值大于或等于ReadView中的max\_trx\_id值，表明生成该版本的事 务在当前事务生成ReadView后才开启，所以该版本不可以被当前事务访问。

**﹟**如果被访问版本的trx\_id属性值在ReadView的min\_trx\_id和max\_trx\_id之间，那就需要判断一 下trx\_id属性值是不是在m\_ids列表中，如果在，说明创建ReadView时生成该版本的事务还是活 跃的，该版本不可以被访问；如果不在，说明创建ReadView时生成该版本的事务已经被提交，该 版本可以被访问。

○**MVCC作用**

**﹟**并发读写下，读不加锁，读写不冲突

**﹟**解决脏读，幻读，不可重复读等事务隔离问题

◇**隔离级别类型**



•**读未提交**

Read Uncommitted 读未提交：解决了回滚覆盖类型的更新丢失，但可能发生脏读现象，也就是 可能读取到其他会话中未提交事务修改的数据。

•**已提交读**

Read Committed 读已提交：只能读取到其他会话中已经提交的数据，解决了脏读。但可能发生 不可重复读现象，也就是可能在一个事务中两次查询结果不一致。

•**可重复读**

Repeatable Read 可重复读：解决了不可重复读，它确保同一事务的多个实例在并发读取数据 时，会看到同样的数据行。

不过理论上会出现幻读，简单的说幻读指的的当用户读取某一范围的数 据行时，另一个事务又 在该范围插入了新行，当用户在读取该范围的数据时会发现有新的幻影行。

•**可串行化**

Serializable 串行化：所有的增删改查串行执行。它通过强制事务排序，解决相互冲突，从而 解决 幻度的问题。这个级别可能导致大量的超时现象的和锁竞争，效率低下。

◇**MySQL隔离级别控制**

MySQL默认的事务隔离级别是Repeatable Read（**可重复读）**，查看MySQL当前数据库事务隔离级别命令：

show variables like 'tx\_isolation';

或

select @@tx\_isolation;

设置事务隔离级别可以如下命令：

set tx\_isolation='READ-UNCOMMITTED';

set tx\_isolation='READ-COMMITTED';

set tx\_isolation='REPEATABLE-READ';

set tx\_isolation='SERIALIZABLE';

▶**MySQL锁**

◇**锁分类**

•从操作的粒度可分为**表级锁**、**行级锁**和**页级锁**。

•从操作的类型可分为**读锁（S锁）**和**写锁（X锁）**。

•从操作的性能可分为**乐观锁**和**悲观锁**。

◇**行锁原理**

在InnoDB引擎中，**行锁**又分为**共享锁**和**排他锁**。InnoDB行锁是通过对索引数据页上的记录加锁实现的，主要实现算法有 3 种：**Record Lock、Gap Lock** 和 **Next-key Lock**。

•**RecordLock锁**：锁定单个行记录的锁。（记录锁，RC（已提交读）、RR（可重复读）隔离级别都支持）

•**GapLock锁**：间隙锁，锁定索引记录间隙，确保索引记录的间隙不变。（范围锁，RR隔离级别支持）

•**Next-key Lock 锁**：记录锁和间隙锁组合，同时锁住数据，并且锁住数据前后范围。（记录锁+范围 锁，RR隔离级别支持）

在RR隔离级别，InnoDB对于记录加锁行为都是先采用Next-Key Lock，但是当SQL操作含唯 一索引时，Innodb会对Next-Key Lock进行优化，降级为RecordLock，仅锁住索引本身非范围。

加锁行为：表里所有行和间隙都会加X锁。（当没有索引时，会导致全表锁定，因为InnoDB引 擎锁机制是基于索引实现的记录锁定）。

**加锁行为：**

表里所有行和间隙都会加X锁。（当没有索引时，会导致全表锁定，因为InnoDB引擎

锁机制是基于索引实现的记录锁定）。

◇**悲观锁**

指在数据处理过程，将数据处于锁定状态，一般使用数据库的锁机制实现。行锁、表锁、读锁、写锁、共享锁、排他锁等，这些都属于悲观锁范畴。

◇**乐观锁**

**乐观锁实现的关键点：冲突的检测。**

**乐观锁实现原理**

•**使用版本字段（version）：**先给数据表增加一个版本(version) 字段，每操作一次，将那条 记录的版本号加 1。version 是用来查看被读的记录有无变化， 作用是防止记录在业务处理期间被其他事务修改

•**使用时间戳（Timestamp）：**与使用version版本字段相似，同样需要给数据表增加一个字 段，字段类型使用timestamp 时间戳。也是在更新提交的时候 检查当前数据库中数据的时间戳和自己更新前取到的时间戳 进行对比，如果一致则提交更新，否则就是版本冲突，取消作。。

◇**死锁**

•**表锁死锁**

用户A访问表A（锁住了表A），然后又访问表B；另一个用户B访问表B（锁住了表B），然后 企图 访问表A；这时用户A由于用户B已经锁住表B，它必须等待用户B释放表B才能继续， 同样用户B要 等用户A释放表A才能继续，这就死锁就产生了。

•**行级死锁**

如果在事务中执行了一条没有索引条件的查询，引发全表扫描，把行级锁上升为全表记录锁定 （等价于表级锁），多个这样的事务执行后，就很容易产生死锁和阻塞，最终应用系统会越来 越慢，发 生阻塞或死锁

•**死锁排查**

○**查看死锁日志**

通过**show engine innodb status\G**命令查看近期死锁日志信息。

使用方法：

1. 查看近期死锁日志信息；

2、使用explain查看下SQL执行计划

○**查看锁状态变量**

通过**show status like'innodb\_row\_lock%‘**命令检查状态变量，分析系统中的行锁的争 夺情况

•**如何尽可能避免死锁**

○**大事务拆小。**

○**降低隔离级别。**

○**为表添加合理的索引。**

▶**MySQL集群架构设计**

•**架构设计理念**

**在集群架构设计时，主要遵从下面三个维度：**

**A：可用性设计：冗余，主从**

**P：扩展性设计：分库分表**

**C：一致性设计：不使用从库**

•**mysql主从复制用途**：

○实时灾备，用于故障切换（高可用）

○读写分离，提供查询服务（读扩展）

○数据备份，避免影响业务（高可用）

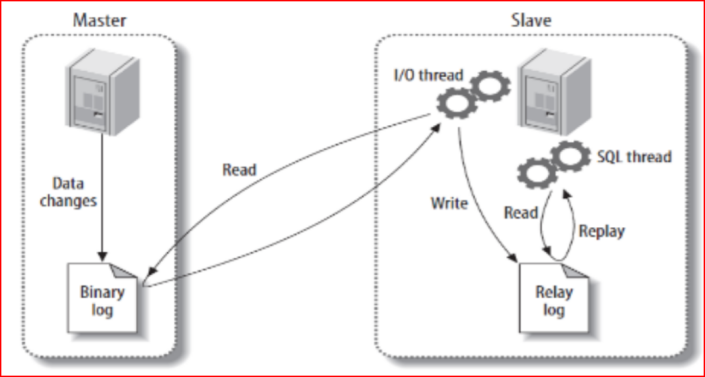
•**主从部署必要条件：**

○从库服务器能连通主库

○主库开启binlog日志（设置log-bin参数）

○主从server-id不同

•**实现原理**



主从同步原理

（Master服务器对数据库更改操作记录在Binlog中，BinlogDump Thread接到写入请求后，读取 Binlog信息 推送给Slave的I/O Thread。

Slave的I/O Thread将读取到的Binlog信息写入到本地Relay Log中。

Slave的SQL Thread检测到Relay Log的变更请求，解析Relay log中内容在从库上执行。）<记下面的>

○主库将数据库的变更操作记录到Binlog日志文件中

○从库读取主库中的Binlog日志文件信息写入到从库的Relay Log中继日志中

○从库读取中继日志信息在从库中进行Replay,更新从库数据信息

•**主从复制存在的问题**

○**数据丢失的问题**：主库宕机后，数据可能丢失

解决方案：半同步复制：Master需要在Redo日志提交后等待Slave返回ACK

○**从库复制延迟的问题**：从库只有一个SQL Thread，主库写压力大，复制很可能延时

解决方案：并行复制：SQL Thread多线程调整

•**读写分离存在的问题**

○主从同步延迟

写后立刻读：在写入数据库后，某个时间段内读操作就去主库，之后读操作访问从库。

二次查询：先去从库读取数据，找不到时就去主库进行数据读取。该操作容易将读压力返还给 主库

○读写分离落地

基于编程和配置实现（应用端）

基于服务器端代理实现（服务器端）

•**双主模式存在的问题**

建议大家使用**双主单写**，因为**双主双写**存在以下问题

分布式事务

分布式ID

SNOWFLAKE

更新丢失

MHA：自动故障转移

•**主备切换存在的问题**

**主备延迟**

○可靠性优先：在专门的HA高可用组件切换过程中某一时刻主库A和从库B都处于只读状态

○可用性优先：不等主从同步完成， 直接把业务请求切换至从库B ，并且让从库B可读写，存在数 据不一致

•**大表优化**

○限定数据的范围

禁止不带任何限制数据范围条件的查询语句

○读/写分离

主写从读

○垂直拆分

根据列的相关性拆分为多张表

○水平拆分

保持数据表结构不变，通过某种策略存储数据分片。每一片数据分散到不同的表或者库中

▶**分表分库**

**一. 数据切分**

**1、垂直（纵向）切分**

垂直切分常见有垂直分库和垂直分表两种。

**垂直切分的优点**：

•**解决业务系统层面的耦合，业务清晰**

•与微服务的治理类似，也能对不同业务的数据进行分级管理、维护、监控、扩展等

•高并发场景下，垂直切分一定程度的提升IO、数据库连接数、单机硬件资源的瓶颈

**缺点**：

•部分表无法join，只能通过接口聚合方式解决，提升了开发的复杂度

•分布式事务处理复杂

•依然存在单表数据量过大的问题（需要水平切分）

**2**、**水平（横向）切分**

**水平切分的优点**：

•不存在单库数据量过大、高并发的性能瓶颈，提升系统稳定性和负载能力

•应用端改造较小，不需要拆分业务模块

**缺点**：

•跨分片的事务一致性难以保证

•跨库的join关联查询性能较差

•数据多次扩展难度和维护量极大

水平切分后同一张表会出现在多个数据库/表中，每个库/表内容不同。几种典型的数据分片规则为：

**1）、根据数值范围**

按照时间区间或ID区间来切分。例如：按日期将不同月甚至是日的数据分散到不同的库中；将userId为1~9999的记录分到第一个库，10000~20000的分到第二个库，以此类推。

这样的**优点**在于：

单表大小可控

天然便于水平扩展，后期如果想对整个分片集群扩容时，只需要添加节点即可，无需对其他分片的数据进行迁移

使用分片字段进行范围查找时，连续分片可快速定位分片进行快速查询，有效避免跨分片查询的问题。

**缺点**：

热点数据成为性能瓶颈。连续分片可能存在数据热点，例如按时间字段分片，有些分片存储最近时间段内的数据，可能会被频繁的读写，而有些分片存储的历史数据，则很少被查询

**2）、根据数值取模**

一般采用hash取模mod的切分方式，例如：将 Customer 表根据 cusno 字段切分到4个库中，余数为0的放到第一个库，余数为1的放到第二个库，以此类推。

**优点**：

数据分片相对比较均匀，不容易出现热点和并发访问的瓶颈

**缺点**：

后期分片集群扩容时，需要迁移旧的数据（使用一致性hash算法能较好的避免这个问题）

容易面临跨分片查询的复杂问题。比如上例中，如果频繁用到的查询条件中不带cusno时，将会导致无法定位数据库，从而需要同时向4个库发起查询，再在内存中合并数据，取最小集返回给应用，分库反而成为拖累。

**二. 分库分表带来的问题**

**1、事务一致性问题**

分布式事务，没有简单的方案，一般可使用"XA协议"和"两阶段提交"处理。（二阶段提交、三阶段提交）

数据一致性

强一致性：XA协议

最终一致性：TCC、saga、Seata

**2、跨节点关联查询 join 问题**

**3、跨节点分页、排序、函数问题**

**4、全局主键避重复问题：**SNOWFLAKE：百度UidGenerator、美团Leaf、基于SNOWFLAKE算法实现

▶**ShardingSphere**

•Sharding-JDBC：被定位为轻量级Java框架，在Java的JDBC层提供的额外服务，以jar包形式使 用。

•Sharding-Proxy：被定位为透明化的数据库代理端，提供封装了数据库二进制协议的服务端版 本， 用于完成对异构语言的支持。

•Sharding-Sidecar：被定位为Kubernetes或Mesos的云原生数据库代理，以DaemonSet的形式代 理 所有对数据库的访问。



◇**Sharding-JDBC**

•**Sharding-JDBC主要功能**：

○数据分片

分库、分表

读写分离

分片策略

分布式主键

○分布式事务

标准化的事务接口

XA强一致性事务

柔性事务

○数据库治理

配置动态化

编排和治理

数据脱敏

可视化链路追踪

•**Sharding-JDBC初始化流程**：

○根据配置的信息生成Configuration对象

○通过Factory会将Configuration对象转化为Rule对象

○通过Factory会将Rule对象与DataSource对象封装

○Sharding-JDBC使用DataSource进行分库分表和读写分离操作

•**数据分片**

**1、分片算法**（ShardingAlgorithm）

○精确分片算法PreciseShardingAlgorithm

用于处理使用单一键作为分片键的=与IN进行分片的场景。

○范围分片算法RangeShardingAlgorithm

用于处理使用单一键作为分片键的BETWEEN AND、>、<、>=、<=进行分片的场景

○复合分片算法ComplexKeysShardingAlgorithm

用于处理使用多键作为分片键进行分片的场景

○Hint分片算法HintShardingAlgorithm

通过Hint指定分片值而非从SQL中提取分片值的方式进行分片

**2、分片策略（ShardingStrategy）：分片策略包含分片键和分片算法**

○**标准分片策略StandardShardingStrategy**

只支持单分片键，提供对SQL语句中的=, >, <, >=, <=, IN和BETWEEN AND的分片操作支持。

提供PreciseShardingAlgorithm和RangeShardingAlgorithm两个分片算法。

○**复合分片策略ComplexShardingStrategy**

支持多分片键。提供对SQL语句中的=, >, <, >=, <=, IN和BETWEEN AND的分片操作支持

○**行表达式分片策略InlineShardingStrategy**

只支持单分片键。使用Groovy的表达式，提供对SQL语句中的=和IN的分片操作支持，对于

简单的分片算法，可以通过简单的配置使用，从而避免繁琐的Java代码开发。如: t\_user\_$->

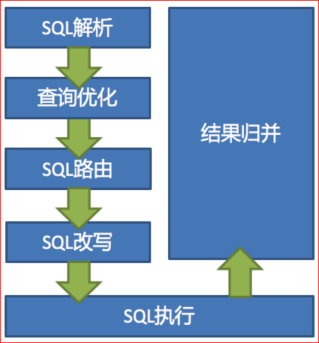
{u\_id % 8} 表示t\_user表根据u\_id模8，而分成8张表，表名称为t\_user\_0到t\_user\_7

○**Hint分片策略HintShardingStrategy**

通过Hint指定分片值而非从SQL中提取分片值的方式进行分片的策略。

○**不分片策略NoneShardingStrategy**

•**数据分片功能主要流程**



**SQL解析：**词法解析和语法解析

**查询优化：**负责合并和优化分片条件，如OR等。

**SQL路由：**根据解析上下文匹配用户配置的分片策略，并生成路由路径

**SQL改写：**将SQL改写为在真实数据库中可以正确执行的语句

**SQL执行：**通过多线程执行器异步执行SQL。

**结果归并：**将多个执行结果集归并以便于通过统一的JDBC接口输出

•**SQL使用规范**

○不支持项（路由至多数据节点）

○不支持CASE WHEN、HAVING、UNION (ALL)

○无论嵌套多少层，只能解析至第一个包含数据表的子查询，一旦在下层嵌套中再次找到包含数据表 的子查询将直接抛出解析异常。

○由于归并的限制，子查询中包含聚合函数目前无法支持。

○不支持包含schema的SQL

○当分片键处于运算表达式或函数中的SQL时，将采用全路由的形式获取结果。

•**分布式事务**

○Saga模式（最终一致性）

一个Saga分布式事务看做是一个由多个本地事务组成的事务，每个本地事务都有一个与之对应 的补偿事务。在 Saga事务的执行过程中，如果某一步执行出现异常，Saga事务会被终止，同 时会调用对应的补偿事务完成相关的恢复操作。

○Seata框架

Seata（Simple Extensible Autonomous Transaction Architecture）是一套一站式分布式事 务解决方案，是阿里集团和蚂蚁金服联合打造的分布式事务框架。Seata目前的事务模式有AT、 TCC、 Saga和XA，默认是AT模式，AT本质上是2PC协议的一种实现。

•**分布式事务模式**

了解了分布式事务中的强一致性和最终一致性理论，下面介绍几种常见的分布式事务的解决方案。

○2PC模式（强一致性）

阶段 1：准备阶段

阶段 2：提交阶段

2PC 方案实现起来简单，实际项目中使用比较少，主要因为以下问题：

**﹟**性能问题：所有参与者在事务提交阶段处于同步阻塞状态，占用系统资源，容易导致性能瓶 颈。

**﹟**可靠性问题：如果协调者存在单点故障问题，如果协调者出现故障，参与者将一直处于锁定 状态。

**﹟**数据一致性问题：在阶段 2 中，如果发生局部网络问题，一部分事务参与者收到了提交消 息，另一部分事务参与者没收到提交消息，那么就导致了节点之间数据的不一致。

○3PC模式（强一致性）

3PC 三阶段提交，是两阶段提交的改进版本，与两阶段提交不同的是，引入超时机制。同时在协 调者和参与者中都引入超时机制。三阶段提交将两阶段的准备阶段拆分为 2 个阶段，插入了一个 preCommit 阶段，解决了原先在两阶段提交中，参与者在准备之后，由于协调者或参与者发生崩 溃或错误，而导致参与者无法知晓处于长时间等待的问题。如果在指定的时间内协调者没有收到参 与者的消息则默认失败。

**﹟**阶段1：canCommit 协调者向参与者发送 commit 请求，参与者如果可以提交就返回 yes 响应，否则返回 no 响 应。

**﹟**阶段2：preCommit 协调者根据阶段 1 canCommit 参与者的反应情况执行预提交事务或中断事务操作。 参与者均反馈 yes：协调者向所有参与者发出 preCommit 请求，参与者收到 preCommit 请求后，执行事务操作，但不提交；将 undo 和 redo 信息记入事务日志 中；各参与者向协调者反馈 ack 响应或 no 响应，并等待最终指令。 任何一个参与者反馈 no或等待超时：协调者向所有参与者发出 abort 请求，无论收到 协调者发出的 abort 请求，或者在等待协调者请求过程中出现超时，参与者均会中断事 务。

**﹟**阶段3：do Commit 该阶段进行真正的事务提交，根据阶段 2 preCommit反馈的结果完成事务提交或中断操作。

◇Sharding-Proxy：被定位为透明化的数据库代理端，提供封装了数据库二进制协议的服务端版本，用于完成对异构语言的支持。

◇Sharding-Sidecar：被定位为Kubernetes或Mesos的云原生数据库代理，以DaemonSet的形式代理所有对数据库的访问。

▶**SQL 开发规范**

•充分利用表上已经存在的索引

•禁止使用 SELECT \*

•禁止使用不含字段列表的 INSERT 语句

•子查询优化为 join 操作

•避免使用 JOIN 关联太多的表

•禁止对WHERE 从句中列进行函数转换和计算

•拆分复杂的大 SQL 为多个小 SQL

▶**数据库三范式**

•确保每列的原子性

•每个表只能描述一件事情

•不存在对非主键列的传递依赖

▶**char和varchar区别**

•char类型的长度是固定的，varchar的长度是可变的

这就表示，存储字符串'abc'，使用char(10)，表示存储的字符将占10个字节

•char类型的效率比varchar的效率稍高