MySQL
数据
存储引擎
行格式
字长字段长度列表
NULL值列表
记录头信息
delete_mask
min_rec_mask
n_owned
heap_no
record_type
next_record
记录的真实数据
row_id
隐式一张叫一表记录
transaction_id
事务ID
roll_pointer
回滚指针
数据列
页目录(Page Directory)
1. 插入时是顺序插入的，每个页都会有一个最小和最大的记录，分别为Infimum和Supremum
2. 页目录是一个稀疏目录，也就是说页目录中的记录并不是每条记录都有一个slot，而是每隔几条记录才会有一个slot
3. 页目录中的记录是一个单向链表，通过每条记录的next_record指向下一条记录，最后一条记录的next_record指向Supremum
4. 页目录中的记录是有序的，因为页目录中的slot是按照主键顺序排列的，这样可以通过二分法快速定位到目标记录
5. 页目录中的记录是有序的，因为页目录中的slot是按照主键顺序排列的，这样可以通过二分法快速定位到目标记录
6. 页目录中的记录是有序的，因为页目录中的slot是按照主键顺序排列的，这样可以通过二分法快速定位到目标记录
InnoDB数据页结构
1. File Header：固定大小38字节，用于记录页的通用信息，如页号、校验和、前一页和后一页的指针、LSN（日志序列号）等
2. Page Header：固定大小56字节，记录页的状态信息，如记录数、空闲空间、页目录槽数、最小记录和最大记录的指针、页的类型、页的校验和、最近修改的LSN等
3. Infimum和Supremum：两条虚拟记录，分别表示页中最小和最大的记录，用于在页内快速定位
4. User Records：真正存储用户数据的记录区域，这个区域是一个单向链表，通过每条记录的next_record指针连接
5. Free Space：空闲空间，当删除记录时，被删除的记录会变成可用空间，后续插入新记录可以优先使用这块空间
6. Page Directory：页目录区域，用于加速在页内查找记录，通过二分法快速定位到目标记录
7. File Trailer：用于检测页是否完整写入磁盘，通过校验和来实现
MyISAM存储引擎与InnoDB存储引擎的区别？
1. 事务支持：MyISAM不支持事务，而InnoDB支持事务，通过undo log和redo log实现事务的原子性、一致性、隔离性和持久性（ACID）
2. 锁粒度：MyISAM使用表级锁，而InnoDB使用行级锁，这使得InnoDB在高并发写入场景下性能更优
3. 崩溃恢复：MyISAM崩溃后需要手动修复，而InnoDB通过redo log实现崩溃恢复，能够自动回滚未提交的事务并重做已提交的事务
4. 外键支持：MyISAM不支持外键，而InnoDB支持外键，可以保证数据的引用完整性
5. 全文索引：MyISAM支持全文索引，而InnoDB从MySQL 5.6版本开始才支持全文索引
6. 存储结构：MyISAM将数据和索引分别存储在.MYD和.MYI文件中，而InnoDB将数据和索引存储在.ibd文件中，采用聚簇索引结构
Memory内存引擎
Memory引擎的特点？
1. 数据全部存储在内存中，不会将数据写入磁盘，因此读写速度极快，但服务器重启后数据会丢失，内存容量是瓶颈
2. 默认使用哈希索引，适合等值查询（如 id = 100），不支持范围查询
3. 表级锁，并发性能较差，多线程写入时会成为瓶颈
4. 不支持大数据类型，不支持 BLOB、TEXT 等大对象
5. 适合临时表、缓存、中间结果集
Memory vs MyISAM
1. 数据持久性：Memory 重启后数据全丢，MyISAM 数据保存在磁盘
2. 锁粒度：Memory 表级锁，MyISAM 表级锁
3. 内存占用：Memory 全部数据在内存，MyISAM 只缓存索引
4. 适用场景：Memory 适合临时表、缓存、中间结果集，MyISAM 适合大表、读多写少、需要持久化的场景
InnoDB表空间分类
1. 系统表空间（ibdata1）：存储内部元数据、undo 日志、change buffer、doublewrite buffer 等，所有表共享，默认大小 12MB，可通过 innodb_data_file_path 配置
2. 独立表空间（.ibd 文件）：每个表单独存储数据和索引，通过 innodb_file_per_table 开启，推荐生产环境使用，便于管理和回收空间
3. 通用表空间：手动创建的共享表空间，可以存放多个表，适合小表较多的场景
4. 临时表空间：存储临时表和排序过程中的中间数据，重启后自动清空
5. 撤销表空间：存储 undo 日志，用于事务回滚和 MVCC 多版本读取，可独立配置以提高性能
表空间结构
1. 文件头（File Header）：固定 38 字节，记录表空间的通用信息，如表空间 ID、页面类型、校验和、前一页和后一页的指针、LSN 等
2. 文件信息（File Space Header）：记录表空间的整体信息，如表空间大小、扩展区（extent）的分配情况、碎片页的管理信息
3. 数据页（Index）：存储实际的表数据和索引，每页 16KB，是 InnoDB 的最小单位
4. 文件尾（File Trailer）：固定 8 字节，用于检测页是否完整写入磁盘，通过校验和和 LSN 实现
B+树索引结构特点
1. 所有数据只存在叶子节点，非叶子节点只存索引，叶子节点通过双向链表连接，支持范围查询
2. 非叶子节点通过指针指向子节点，形成多层结构，通常 3 层可存约 2000 万条记录
3. 叶子节点按主键顺序排列，插入时可能触发页分裂，删除时可能触发页合并
B+树索引的优势
1. 范围查询：叶子节点通过双向链表连接，范围查询只需顺序遍历叶子节点，无需回表
2. 磁盘 I/O 优化：非叶子节点只存索引，一个页能存更多索引项，减少磁盘 I/O 次数
3. 稳定性能：所有查询都需要从根节点到叶子节点，时间复杂度固定为 O(log n)，性能可预测
B+树 vs B树
1. 叶子节点：B+树叶子节点通过双向链表连接，支持范围查询；B 树叶子节点独立，范围查询需要回表
2. 非叶子节点：B+树非叶子节点只存索引，一个页能存更多索引项，减少磁盘 I/O；B 树非叶子节点存数据，一个页能存的索引项少，树高度更高
3. 查询性能：B+树所有查询都需要到叶子节点，性能稳定；B 树可能在非叶子节点提前找到数据，平均查询速度更快
4. 适用场景：B+树适合范围查询和顺序扫描，是 MySQL 的默认选择；B 树适合点查多、范围查询少的场景，如内存数据库
B+树 vs 哈希索引
1. 等值查询：哈希索引 O(1)，B+树 O(log n)，哈希索引略快
2. 范围查询：哈希索引不支持范围查询，只能全表扫描；B+树支持高效范围查询
3. 排序：哈希索引无序，B+树天然有序，可直接用于 ORDER BY
4. 内存占用：哈希索引需要额外内存存储哈希表，B+树主要占用磁盘空间
5. 适用场景：哈希索引适合内存表、等值查询为主的场景；B+树是通用选择，适合大多数场景
聚簇索引与非聚簇索引的区别
1. 聚簇索引：叶子节点存储完整的数据，每个表只能有一个，通常是主键索引
2. 非聚簇索引：叶子节点存储主键值，需要回表查询完整数据，可以有多个
3. 性能影响：聚簇索引范围查询快，因为数据连续存储；非聚簇索引需要随机 I/O 回表，性能较差
4. 选择建议：优先使用主键作为聚簇索引，避免频繁回表；合理设计索引覆盖查询，减少回表次数
回表查询
1. 定义：通过非聚簇索引找到主键值，再用主键在聚簇索引中查找完整数据的过程
2. 性能影响：每次回表需要一次随机 I/O，大量回表会成为性能瓶颈
3. 优化方法：使用覆盖索引（select 只查索引列）、调整索引顺序、增加覆盖列、使用 EXPLAIN 检查是否回表
4. 适用场景：回表适合查询结果集较小的场景，结果集大时建议优化
覆盖索引
1. 定义：查询的列全部在索引中，无需回表即可获取数据
2. 实现方式：在非聚簇索引中包含所需的列，如 SELECT name FROM t WHERE id = 1 可建索引 (id, name)
3. 性能优势：减少磁盘 I/O，大幅提升查询速度，是最有效的优化手段
4. 适用场景：频繁查询固定列的场景，如列表页只需要 id、name、avatar 等字段
最左前缀原则
1. 定义：复合索引 (a, b, c) 可以支持 a、a,b、a,b,c 三种查询，但不支持 b、c、b,c 单独查询
2. 原理：B+树按复合键的字典序排序，只有最左列有全局有序性，后续列在前一列相同的情况下才有序
3. 实现：EXPLAIN 中 key_len 反映实际使用的索引长度，key_len 越大说明使用的列越多
4. 设计建议：将最常用、最频繁的查询条件放在最左侧，提高索引利用率
EXPLAIN 关键字
1. type：连接类型，从最优到最差依次为 system > const > eq_ref > ref > range > index > ALL，目标是至少达到 range 级别
2. key：实际使用的索引名，NULL 表示未使用索引
3. rows：预估扫描的行数，越小越好，反映查询效率
4. Extra：额外信息，如 Using index（覆盖索引）、Using where（服务层过滤）、Using filesort（需要排序）、Using temporary（使用临时表）
5. 优化流程：先看 type 是否合理，再看 key 是否使用了预期索引，最后看 rows 是否可接受
Buffer Pool
为了提升磁盘中的页，把 MySQL 数据最近的页面缓存在内存中，减少磁盘 I/O。
Buffer Pool的组成结构
1. 缓冲区（Buffer）：固定大小的内存区域，通过 innodb_buffer_pool_size 配置，通常占内存的 70%-80%
2. 空闲链表（Free List）：记录所有空闲页的指针，分配新页时从这里取
3. 脏页链表（Flush List）：记录被修改过但还未刷盘的页，按修改时间排序，后台线程从这里开始刷盘
4. 哈希表（Page Hash）：通过表空间 ID + 页号快速定位内存中的页，O(1) 查找
Buffer Pool 的作用
1. 减少磁盘 I/O：热数据常驻内存，大部分查询直接命中内存，不需要读磁盘
2. 提高写入性能：脏页先写内存，后台异步刷盘，写入操作变成顺序写
3. 支持并发：多线程共享 Buffer Pool，通过锁机制保证数据一致性
Buffer Pool 的淘汰策略
1. 核心问题：内存有限，需要淘汰旧页为新页腾出空间
2. 朴素 LRU 的问题：全表扫描会把大量冷数据加入 LRU 头部，把热数据挤出去，导致命中率下降
3. 改进方案：将 LRU 分为 young 区（热数据）和 old 区（新数据），只有在 old 区停留超过 innodb_old_blocks_time 毫秒（默认 1000ms）的页才会晋升到 young 区，防止扫描污染热数据
4. 淘汰流程：从 old 区尾部开始淘汰，脏页需先刷盘再淘汰，干净页直接淘汰
Buffer Pool 的刷盘机制
1. 后台线程：主线程每 1 秒唤醒一次，检查脏页比例，超过阈值（默认 75%）开始刷盘
2. 刷盘策略：按 LSN 顺序刷盘，保证日志先于数据写入（WAL 原则）
3. 自适应刷盘：根据脏页产生速度和磁盘 I/O 能力动态调整刷盘速度，避免突发 I/O 尖峰
4. 强制刷盘：checkpoint 时必须把指定 LSN 之前的脏页全部刷盘，保证崩溃恢复时数据完整
Buffer Pool 的监控与调优
1. 关键指标：命中率（Hit Ratio）、脏页比例、空闲页数、LRU 长度
2. 命中率计算：Hits / (Hits + Misses)，生产环境应保持在 99% 以上
3. 调优方向：命中率低→增大 buffer pool；脏页多→调大 innodb_io_capacity；频繁淘汰→检查是否有大表全扫描
4. 监控工具：SHOW ENGINE INNODB STATUS、performance_schema、sys 视图
锁
锁的分类
1. 按粒度：表锁（MyISAM 默认）、行锁（InnoDB 默认）、页锁（BDB 引擎，已淘汰）
2. 按模式：共享锁（S，允许读）、排他锁（X，独占写）
3. 按意向：意向共享锁（IS）、意向排他锁（IX），表示事务将在某行加锁，用于快速判断表是否可加表锁
4. 按算法：记录锁（Record Lock）、间隙锁（Gap Lock）、临键锁（Next-Key Lock）
5. 按状态：显式锁（SELECT ... FOR UPDATE）、隐式锁（由 DML 自动加锁）
行锁的实现
1. 记录锁（Record Lock）：锁定单条记录，防止其被修改
2. 间隙锁（Gap Lock）：锁定两条记录之间的间隙，防止插入新记录，只在 RR 隔离级别生效
3. 临键锁（Next-Key Lock）：记录锁 + 间隙锁，锁定记录及其前面的间隙，是 RR 隔离级别的默认行锁
4. 作用：防止幻读，保证范围查询的一致性
锁的兼容性
1. 共享锁（S）：可与其他 S 锁兼容，不与 X 锁兼容
2. 排他锁（X）：与所有锁互斥，独占资源
3. 意向锁（IS/IX）：表示事务将在某行加锁，用于快速判断表是否可加表锁，不阻塞行级锁
4. 间隙锁：只阻塞插入操作，不阻塞读取
死锁
1. 定义：两个或多个事务互相等待对方释放锁，形成循环等待，导致无法继续执行
2. 典型场景：事务 A 锁 id=1 等 id=2，事务 B 锁 id=2 等 id=1
3. 检测机制：InnoDB 每隔 1 秒检测一次等待图，发现环后回滚代价最小的事务
4. 预防方法：按固定顺序访问资源、减少事务持锁时间、使用乐观锁
5. 监控：SHOW ENGINE INNODB STATUS 查看最近死锁信息
事务的隔离级别
1. 脏读（Dirty Read）：读取到其他事务未提交的数据，可能读到回滚后不存在的数据
2. 不可重复读（Non-Repeatable Read）：同一事务内两次读取同一条记录，结果不一致，因为中间有其他事务修改了数据
3. 幻读（Phantom）：同一事务内两次范围查询，第二次查出了第一次没有的新行，因为其他事务插入了符合条件的记录
4. 级别对应关系：RU 允许全部，RC 允许 2 和 3，RR 允许 3，Serializable 全部不允许
5. 实现方式：RU 直接读内存，RC 每次 SELECT 生成新 ReadView，RR 复用事务开始时的 ReadView，Serializable 加共享锁
RR 如何防止幻读
1. 快照读：通过 MVCC 多版本读取，每次看到的是事务开始时的数据，天然不会出现新插入的行
2. 当前读：SELECT ... FOR UPDATE 等语句会读取最新数据，此时需要间隙锁来防止幻读
3. 间隙锁的作用：锁住范围内的间隙，阻止其他事务在该范围内插入新记录
4. 局限性：间隙锁会降低并发性，在高冲突场景下可能成为瓶颈
RC 与 RR 的选择
1. 默认推荐：RR 是 MySQL 默认隔离级别，适合大多数场景
2. 切换 RC 的场景：需要减少间隙锁冲突、主从复制使用 binlog_format=mixed、业务逻辑能接受幻读
3. 配置方式：SET GLOBAL transaction_isolation = 'READ-COMMITTED'
binlog
binlog 的作用
1. 主从复制：主 Master 将 binlog 发送给从 Slave，从 Slave 重放 binlog 达到数据一致
2. 数据恢复：通过重放 mysqlbinlog 工具将某段时间的 binlog 应用到备份上，实现时间点恢复
binlog 的格式
1. STATMENT：记录执行的原始语句，体积小但可能因函数、子查询、LIMIT 等语句在主从上产生不一致
2. ROW：记录每行数据的变更前后内容，保证主从一致性，但体积较大，适合生产环境
3. MIXED：默认优先用 STATMENT，遇到不确定语句自动切换到 ROW
binlog 的刷盘时机
1. 事务提交时：每个事务的 binlog 在 COMMIT 时写入磁盘
2. 缓冲区满：binlog_cache_size 满了会强制刷盘
3. 定时刷盘：sync_binlog 参数控制每 N 个事务刷盘一次，默认 1 表示每个事务都刷盘
binlog 的清理
1. 自动清理：expire_logs_days 参数控制保留天数，超过的文件会被删除
2. 手动清理：PURGE BINARY LOGS 命令可以指定时间点或文件名清理
3. 注意事项：主从复制未完成的 binlog 不能删除，否则从库会报错
redo log
redo log 的作用
1. 崩溃恢复：MySQL 重启时通过 redo log 重放未刷盘的脏页，保证已提交的事务不丢失
2. 性能优化：将随机写磁盘变成顺序写 redo log，大幅提升写入性能
redo log 的刷盘时机
1. 事务提交时：每个事务的 redo log 在 COMMIT 时写入磁盘
2. 后台线程：每秒刷盘一次，即使没有事务提交
3. 缓冲区满：redo log buffer 满了会强制刷盘
redo log 的清理
1. 循环覆写：redo log 文件固定大小，写满后从头开始覆写
2. 检查点（Checkpoint）：记录已刷盘的位置，之前的 redo log 可以覆写
redo log 的大小配置
1. 参数：innodb_log_file_size，单个文件大小
2. 建议：生产环境设置为 1G 左右，减少频繁切换文件的开销
redo log 与 binlog 的区别
1. redo log 是 InnoDB 引擎特有的，binlog 是 MySQL 层面的
2. redo log 是物理日志，binlog 是逻辑日志
3. redo log 是循环写，binlog 是追加写
redo log 的两阶段提交
1. 为什么需要：保证 redo log 和 binlog 的一致性，防止主从不一致
2. 流程：先写 redo log 标记为 prepare，再写 binlog，最后标记 redo log 为 commit
3. 崩溃恢复：重启后检查 prepare 状态的 redo log，如果 binlog 已经写入则提交，否则回滚
undo log
undo log 的作用
1. 事务回滚：记录修改前的数据，用于执行 ROLLBACK 时恢复数据
2. 多版本读取：为 MVCC 提供历史版本，实现快照读
undo log 的结构
1. 每条 undo 记录包含：表 ID、主键值、修改前的列值、指向上一条 undo 记录的指针
2. 通过指针形成链表，按时间倒序排列
undo log 的清理
1. 自动清理：purge 线程定期扫描，删除不再被任何 ReadView 引用的 undo 记录
2. 长事务影响：长事务会阻止 purge 线程清理，导致 undo 表空间膨胀
3. 监控：SHOW ENGINE INNODB STATUS 查看 History list length，超过 10000 需关注
undo log 与 redo log 的区别
1. undo log 是逻辑日志，记录的是修改前的数据
2. undo log 是循环写，redo log 是追加写
3. undo log 用于回滚和 MVCC，redo log 用于崩溃恢复
binlog 与 redo log 的配合
1. 两阶段提交：redo log 先 prepare，再写 binlog，最后 commit
2. 崩溃恢复：重启后检查 prepare 状态的 redo log，如果 binlog 已写入则提交，否则回滚
3. 一致性保证：保证主从复制和崩溃恢复时数据一致
redo log 的大小配置
1. 参数：innodb_log_file_size，单个文件大小
2. 建议：生产环境设置为 1G 左右，减少频繁切换文件的开销
redo log 的刷盘时机
1. 事务提交时：每个事务的 redo log 在 COMMIT 时写入磁盘
2. 后台线程：每秒刷盘一次，即使没有事务提交
3. 缓冲区满：redo log buffer 满了会强制刷盘
redo log 的清理
1. 循环覆写：redo log 文件固定大小，写满后从头开始覆写
2. 检查点（Checkpoint）：记录已刷盘的位置，之前的 redo log 可以覆写
redo log 的大小配置
1. 参数：innodb_log_file_size，单个文件大小
2. 建议：生产环境设置为 1G 左右，减少频繁切换文件的开销
redo log 与 binlog 的区别
1. redo log 是 InnoDB 引擎特有的，binlog 是 MySQL 层面的
2. redo log 是物理日志，binlog 是逻辑日志
3. redo log 是循环写，binlog 是追加写
redo log 的两阶段提交
1. 为什么需要：保证 redo log 和 binlog 的一致性，防止主从不一致
2. 流程：先写 redo log 标记为 prepare，再写 binlog，最后标记 redo log 为 commit
3. 崩溃恢复：重启后检查 prepare 状态的 redo log，如果 binlog 已写入则提交，否则回滚
undo log
undo log 的作用
1. 事务回滚：记录修改前的数据，用于执行 ROLLBACK 时恢复数据
2. 多版本读取：为 MVCC 提供历史版本，实现快照读
undo log 的结构
1. 每条 undo 记录包含：表 ID、主键值、修改前的列值、指向上一条 undo 记录的指针
2. 通过指针形成链表，按时间倒序排列
undo log 的清理
1. 自动清理：purge 线程定期扫描，删除不再被任何 ReadView 引用的 undo 记录
2. 长事务影响：长事务会阻止 purge 线程清理，导致 undo 表空间膨胀
3. 监控：SHOW ENGINE INNODB STATUS 查看 History list length，超过 10000 需关注
undo log 的大小配置
1. 参数：innodb_undo_log_truncate，是否开启自动清理
2. 建议：生产环境开启自动清理，避免 undo 表空间膨胀
undo log 与 redo log 的区别
1. undo log 是逻辑日志，记录的是修改前的数据
2. undo log 是循环写，redo log 是追加写
3. undo log 用于回滚和 MVCC，redo log 用于崩溃恢复
undo log 的两阶段提交
1. 为什么需要：保证 undo log 和 redo log 的一致性，防止主从不一致
2. 流程：先写 undo log 标记为 prepare，再写 redo log，最后标记 undo log 为 commit
3. 崩溃恢复：重启后检查 prepare 状态的 undo log，如果 redo log 已写入则提交，否则回滚
undo log 的大小配置
1. 参数：innodb_undo_log_truncate，是否开启自动清理
2. 建议：生产环境开启自动清理，避免 undo 表空间膨胀

- Log
  - MySQL服务器层
    - binlog(逻辑日志)
      - binlog的组成部分主要包括两部分：文件头(Magic number)+多个event+结束标志(EOF)
      - binlog的刷盘时机：对于 binlog 来说，从开始到写到磁盘这段时间，一个事务的binlog是按照该事务的大小顺序，将每个事务的日志顺序写入到磁盘上的过程。
        - 事务提交
          - 对于 binlog 来说，从开始到写到磁盘这段时间，一个事务的binlog是按照该事务的大小顺序，将每个事务的日志顺序写入到磁盘上的过程。
          - 在事务提交的时候，会将binlog cache里面的内容，调用一次write操作，将binlog内容写入到page cache中，不落盘。
          - write 和 fsync的时机，是由参数sync_binlog控制的：
            - sync_binlog=0 的时候，表示每次提交事务都只调用write，由系统自行决定什么时候执行fsync。这种情况下，如果机器宕机，page cache里的binlog会全部丢失。
            - sync_binlog=1 的时候，表示每次提交事务都会调用fsync，这种情况下，binlog是持久的。
            - sync_binlog=N(N>1) 的时候，表示每次提交事务都会调用write，但是N个事务才会调用一次fsync，这种情况下，如果机器宕机，会丢失最近N个事务的binlog日志。
    - binlog vs redo log
      - redo log是属于innoDB层，binlog属于MySQL Server层的东西，这样的设计是为了兼容其他的存储引擎，保证其他存储引擎的数据恢复。
      - redo log是物理日志，记录的是“在某个数据页做了什么修改”，binlog是逻辑日志，记录的是这个语句的原始逻辑，比如“给ID=2这一行的c字段加1”。
      - redo log是循环写，日志空间大小是固定，全部写满就从头开始，覆盖之前的内容。binlog是追加写，文件写到一定大小后会生成新的文件，不会覆盖。
      - redo log为了保证事务的持久性，会在每次提交事务的时候，调用fsync把数据写到磁盘。binlog可以作为恢复数据使用，主从复制搭建。
  - redolog
    - 原理：日志先写入redo log buffer，然后在合适的时机（如事务提交时、buffer满了、后台线程定时刷新），通过fsync()系统调用，将buffer中的内容一次性地、顺序地写入磁盘的redo log文件。这个过程是WAL（Write-Ahead Logging）技术。
    - 为什么需要redo log?
      - 解决崩溃恢复时数据丢失问题：
        - 没有redo log，如果数据页在内存中被修改了，但还没来得及写回磁盘，MySQL服务器就宕机了，这些修改就会丢失，导致数据不一致。
        - 有了redo log，即使数据页没写回，只要redo log已经写到磁盘，MySQL重启后，就可以通过redo log来重做（redo）这些丢失的修改，保证数据的完整性。
      - 提升性能：redo log的写入是顺序追加（Append），这比随机写磁盘的数据页要快得多。通过先写redo log，可以把一个事务内对多个数据页的随机写入，变成一个顺序写入redo log的操作，大大提高了性能。
      - redo log的组成：redo log buffer（内存缓冲区）+ redo log文件（磁盘文件）。
      - 刷盘时机：redo log buffer的内容何时写入redo log文件，主要由innodb_flush_log_at_trx_commit参数控制：
        - 0：每秒一次，由后台线程统一刷盘。这种模式性能最高，但有风险，如果MySQL服务器宕机，最多可能丢失1秒钟的数据。
        - 1：每个事务提交时，都必须将redo log buffer的内容写入并fsync到磁盘，保证数据不丢失。这是默认值，也是最安全的模式。
        - 2：每个事务提交时，只将redo log buffer的内容写入文件，但不fsync。这种模式性能较高，但如果MySQL服务器宕机，但操作系统没有宕机，数据通常不会丢失。
    - redo log文件格式
      - 一个固定大小的环形日志组成（redo log buffer），其中包含两个重要的指针：
        - write pos：指向redo log当前正在写入的位置。
        - check point：指向redo log中，数据已经成功写入磁盘的位置。
      - 工作原理：write pos不断向前移动，当它追上check point，就意味着redo log文件已满，需要暂停写入，等待check point向前移动，腾出空间。
    - 用户空间下的缓冲区数据一定丢失的两种场景：
      - 用户空间下的缓冲区（如redo log buffer）中的数据，在MySQL服务器宕机时，会全部丢失。
      - 因此，redo log buffer写入redo log文件（从redo log buffer到redo log文件）这个过程，是保证数据持久性的关键。
    - 两阶段提交（2PC）
      - 为了实现内存数据和磁盘数据的一致性，MySQL引入了两阶段提交（2PC）协议，来保证redo log和binlog的原子性（要么都成功，要么都失败）。
      - 2PC的两个阶段：
        - 阶段一：准备（Prepare）
          - 当一个事务提交时，MySQL会先将redo log buffer中的日志写入redo log文件，但暂时不标记为“提交”。
          - 这个过程，是将redo log buffer的内容，通过fsync()，持久化到磁盘上。
          - 此时，redo log文件中，这条日志的状态是“prepare”（准备）。
        - 阶段二：提交（Commit）
          - 当binlog成功写入磁盘后，MySQL会再次修改redo log文件中，这条日志的状态，将其标记为“commit”（提交）。
          - 这个操作，通常只需要在redo log文件末尾追加一个小的commit记录，速度非常快。
      - 崩溃恢复时的决策：
        - 当MySQL服务器重启，开始恢复数据时，它会扫描redo log文件，寻找处于“prepare”状态的事务。
        - 对于每一个“prepare”状态的事务，MySQL会去检查binlog，看这个事务的binlog是否已经成功写入磁盘。
        - 如果binlog已经成功写入，那么MySQL会认为这个事务是成功的，并将redo log中的记录标记为“commit”，然后完成恢复。
        - 如果binlog没有成功写入，那么MySQL会认为这个事务是失败的，并会回滚redo log中的记录，以保证数据的一致性。
  - 一条更新语句的执行流程
    - 客户端发送一个请求时，服务器会建立一个连接。
    - 服务器会检查权限，如果客户有权限，连接保持建立，直到客户主动断开连接。
    - 解析器模块接收到SQL语句的文本，并把它解析成一个结构化的内部表示（语法树）。
    - 解析器会检查语法和基本的语义，比如表和列是否存在。
    - 优化器模块接收语法树，并为这条语句生成一个最优的执行计划。
    - 它会考虑因素：
      - 对于这条更新语句，它可能有一个或多个可能的执行计划。例如，如果表上有多个索引，优化器会分析哪个索引能最有效地定位到要更新的行，以及哪个索引的维护成本最低。
      - 最终，它会选择一个它认为成本最低、最快的执行计划，并将这个计划传递给执行器。
    - 执行器模块接收优化器生成的执行计划，并开始调用存储引擎的接口来执行数据操作。
    - 存储引擎模块负责实际的数据读写操作，它会根据执行计划，从磁盘或内存中读取数据，并对数据进行修改。
    - 为了保证数据的原子性和持久性，InnoDB存储引擎会在内存中维护一个叫做Buffer Pool的缓冲区，大部分的数据页都会先在这里被修改。
    - 当一条更新语句执行完成后，InnoDB存储引擎会将这条语句的redo log（重做日志）写入到redo log buffer（重做日志缓冲区）中。
    - 这个过程是内存操作，速度非常快。
    - 为了保证redo log的持久性，MySQL会根据一个参数innodb_flush_log_at_trx_commit的设置，来决定redo log buffer的内容何时需要刷写到磁盘上的redo log文件。
    - 这个刷盘的时机，是redo log能够实现崩溃恢复的关键。
    - 为了保证binlog的原子性和持久性，MySQL会采用一种叫做两阶段提交（Two-Phase Commit）的机制。
    - 这个机制，会确保redo log和binlog这两个日志，要么都成功写入，要么都失败。
    - 两阶段提交的流程：
      - 阶段一：准备（Prepare）
        - 当一条更新语句执行完成后，InnoDB存储引擎会将redo log的状态，从“活跃”（Active）变为“准备”（Prepare）。
        - 这个状态，会被持久化到redo log文件中。
      - 阶段二：提交（Commit）
        - 当binlog成功写入磁盘后，MySQL服务器会通知InnoDB存储引擎，将redo log的状态，从“准备”（Prepare）变为“提交”（Commit）。
        - 这个状态，也会被持久化到redo log文件中。
    - 崩溃恢复时的决策：
      - 当MySQL服务器重启，开始恢复数据时，它会扫描redo log文件，寻找处于“prepare”状态的事务。
      - 对于每一个“prepare”状态的事务，MySQL会去检查binlog，看这个事务的binlog是否已经成功写入磁盘。
      - 如果binlog已经成功写入，那么MySQL会认为这个事务是成功的，并将redo log中的记录标记为“commit”，然后完成恢复。
      - 如果binlog没有成功写入，那么MySQL会认为这个事务是失败的，并会回滚redo log中的记录，以保证数据的一致性。
  - 锁
    - 乐观锁 & 悲观锁，不可重复读，幻读
      - 悲观锁：
        - 核心思想：假设冲突总会发生，所以在操作数据之前，就先加锁，独占资源，直到操作完成。
        - 实现：通过数据库的锁机制来实现。
        - 适用场景：冲突发生的概率很高，或者操作的成本很高，不能接受失败重试的场景。
        - 缺点：性能开销大，并发性低，可能会导致死锁。
      - 乐观锁：
        - 核心思想：假设冲突很少发生，所以在操作数据时，不加锁，允许多个事务同时读写。只在最后提交数据时，检查一下数据有没有被别人修改过。
        - 实现：通过版本号（Version）或时间戳（Timestamp）来实现。
        - 适用场景：冲突发生的概率很低，或者操作的成本很高，可以接受失败重试的场景。
        - 缺点：如果冲突发生了，就需要重新执行整个业务逻辑，成本可能很高。
      - 不可重复读：
        - 在一个事务内，对同一条数据，执行了两次查询，结果发现数据被别人修改了，导致两次查询结果不一致。
        - 这违反了数据库的一致性（Consistency）原则。
      - 幻读：
        - 在一个事务内，对一个满足某个条件的数据集合，执行了两次查询，结果发现第二次查询，多了一些第一次查询没有的新记录。
        - 这通常是由于其他事务在这两次查询之间，插入了新的数据行。
      - 解决方案：
        - 对于不可重复读，通过MVCC（多版本并发控制）来实现，让每个事务看到一个一致的数据快照。
        - 对于幻读，通过MVCC和Next-Key Lock（间隙锁）来共同实现，既锁住现有的记录，也锁住记录之间的间隙，防止新记录插入。
    - 行锁
      - 行锁的分类：
        - 共享锁（Shared Lock）：
          - 允许多个事务同时持有对同一行的共享锁，用于读取数据，以保证数据在读取期间不被修改。
          - 语法：SELECT ... LOCK IN SHARE MODE;
        - 排他锁（Exclusive Lock）：
          - 一个事务一旦获得了排他锁，其他事务就不能再对这行数据加任何锁，也不能读取。
          - 语法：SELECT ... FOR UPDATE;
      - 锁的粒度：
        - 行锁：
          - 只锁住被操作的那一行数据，其他行数据可以被其他事务自由操作。
          - 这是InnoDB默认的锁粒度，也是并发性最高的。
        - 表锁：
          - 锁住整个表，任何事务都不能对这个表进行任何操作（除非是只读）。
          - 这是MyISAM存储引擎的默认锁粒度，并发性很低。
      - 锁的类型：
        - 共享锁（S锁）：
        - 排他锁（X锁）：
        - 意向共享锁（IS锁）：
        - 意向排他锁（IX锁）：
        - 记录锁（Record Lock）：
        - 间隙锁（Gap Lock）：
        - 临键锁（Next-Key Lock）：
        - 插入意向锁（Insert Intention Lock）：
        - 自增锁（AUTO-INC Lock）：
      - 两阶段锁（2PL）：
        - 核心思想：一个事务在其生命周期内，会先不断地加锁，直到最后才一次性地释放所有锁。
        - 目的：这是实现可串行化（Serializable）隔离级别的关键。它能确保一个事务的操作，不会被其他事务所干扰，从而保证数据的一致性。
        - 实现：
          - 在事务开始时，通过一条语句（如 BEGIN）来开启。
          - 在事务的整个生命周期内，根据需要，不断地加锁（如 SELECT ... LOCK IN SHARE MODE 或 UPDATE）。
          - 在事务结束时，通过一条语句（如 COMMIT 或 ROLLBACK）来结束，并一次性地释放掉所有锁。
      - 死锁：
        - 定义：两个或多个事务，因为互相等待对方持有的锁，而陷入的一种僵局。
        - 解决：
          - 超时机制：设置一个锁等待的最大时间，超时就放弃。
          - 死锁检测：主动检测死锁的发生，并选择一个事务回滚，来打破僵局。
  - 主从复制
    - 原理：
      - 主库（Master）：
        - 当一个写操作（INSERT, UPDATE, DELETE）发生时，主库会将这个操作的信息，记录到一个叫做binlog的文件中。
        - 这个过程，是顺序追加写入，性能很高。
      - 从库（Slave）：
        - 从库会启动两个线程：
          - 一个IO线程，负责从主库的binlog文件中，实时地拉取（Pull）这些新的日志，并把它们写入到自己的relay log（中继日志）文件中。
          - 一个SQL线程，负责读取relay log文件中的日志，并在自己的数据库上，重新执行这些操作，以保证数据的一致性。
      - 主从延迟：
        - 原因：
          - 主库的写入，是并发的。
          - 从库的SQL线程，是单线程的。
          - 网络传输，也需要时间。
        - 解决方案：
          - 并行复制：让从库的SQL线程，变成多个线程来执行。
          - 半同步复制：主库在提交事务时，会等待至少一个从库确认收到了binlog，才算提交成功。
    - 分库分表
      - 为什么需要分库分表？
        - 单表数据量过大：
        - 单库性能瓶颈：
        - 单点故障：
      - 分库分表的方式：
        - 垂直拆分：
        - 水平拆分：
      - 分库分表后，会带来哪些问题？
        - 跨库查询：
        - 跨库事务：
        - 全局唯一ID：
        - 数据迁移：
      - 分库分表的中间件：
        - ShardingSphere
        - MyCat
        - Vitess