数据库_MySQL

InnoDB 逻辑存储结构

- InnoDB 存储引擎中的所有数据都存放在表空间中！

  表空间分为多个段：索引段、数据段、回滚段

  - 索引段：存放 B+ 树的非叶子结点（Non-Leaf node segment）
  - 数据段：存放 B+ 树的 叶子节点（Leaf node segment）
  - 回滚段：存放 回滚日志（undo log）
- 数据段 分为多个 区（Extent），区 由许多 页（Page） 组成，而页 是行（Row）记录的集合！

  所以数据段的数据单元可分为 段->区->页->行

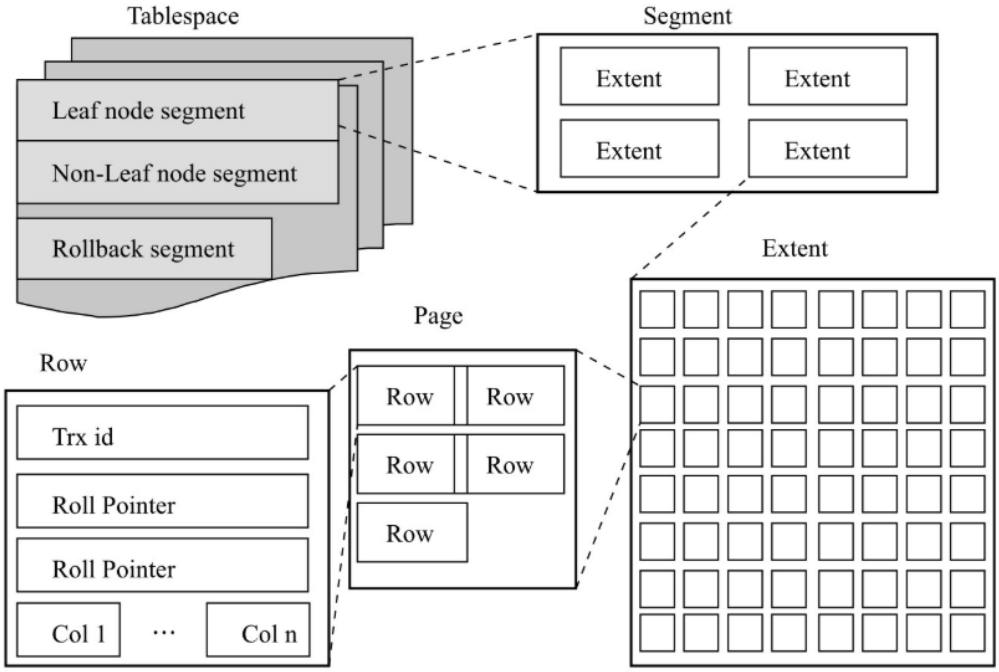


图4-1 InnoDB逻辑存储结构

- 区

  区 是由连续的页组成，任何情况下，每个区的大小都为 1MB

  InnoDB 存储引擎会一次性从磁盘中申请 4~5 个区，默认情况下，页的大小为 16K，此时，一个区中共有 64 个连续的页！

- 页

  每页大小固定为 16K，可以通过参数 innodb_page_size 将页的大小设置为 4K、8K、16K，一旦设定，不可修改！

- 行

InnoDB 存储引擎 是`面向列（row-oriented）`，即：数据是按 **行** 进行存放的！

每个页中存放的行记录有硬性规定，由于 每页大小为 16K，所以每页中最多允许存放的记录：`16 * 1024 / 2 - 200 = 7992`行！

```
存在有 面向行 的数据库：
    column-oriented ：面向行：数据按照 列 进行存放！
```

InnoDB 体系结构

介绍

- 如下图，展示了 InnoDB 存储引擎的体系架构：

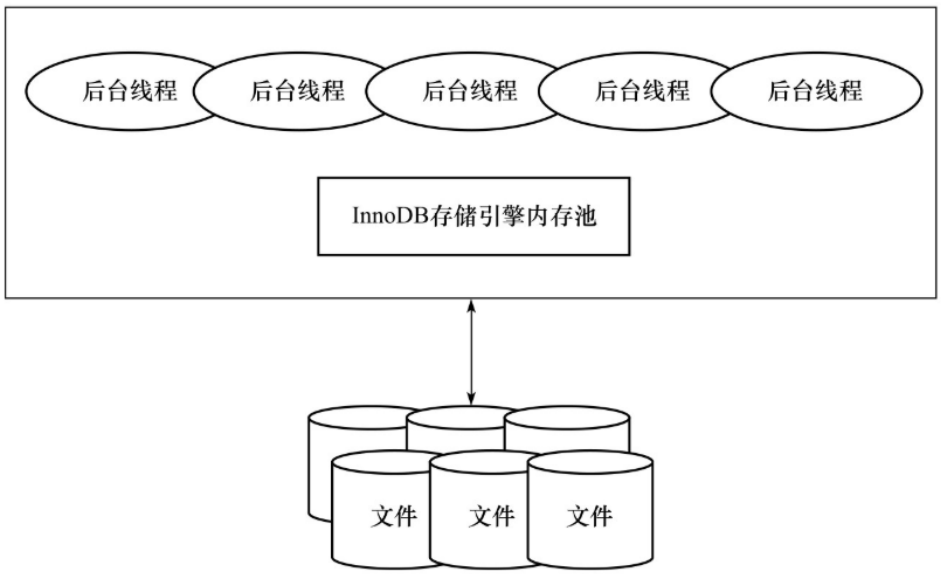


图2-1 InnoDB存储引擎体系架构

- 体系架构中包括 后台线程、内存池、文件！

后台线程

- 后台线程的主要作用 是 负责刷新内存池中的数据，保证缓冲池中的内存缓存的是最近的数据。此外将已修改的数据文件刷新到磁盘文件中，同时保证在数据库发生异常的情况下 InnoDB 能够恢复到正常状态！

Master Thread

- 是 InnoDB 的核心线程，主要负责将缓冲池中的数据异步刷新到 磁盘，保证数据的一致性，包括：脏页的刷新、合并插入缓冲区、undo log 页 的回收等

IO Thread

- InnoDB 提供了大量的 AIO（Async IO） 来处理写请求，极大的提高数据库的性能！而 IO Thread 主要负责将这些 IO 请求回调（call back）处理。
- IO Thread 包括：write、read、insert buffer 和 log 等 IO Thread

Purge Thread

- 使用 Purge Thread 回收不再使用的 undo log！
  - insert undo log：在事务提交成功后就会被清理
  - update undo log：当 不再有事务对该 undo log 执行一致性读时，就会被清理！

Page Cleaner Thread

- 在 InnoDB 1.2 中引入，将 master thread 的脏页刷新工作 交由 page cleaner thread 执行，可以减轻 master thread 的工作及对于用户查询线程的阻塞，进一步提高 InnoDB 存储引擎的性能！

内存池

- 通过 `show engine innodb status` 指令查看 InnoDB 内存使用状态

  也可通过`select * from information_schema.INNODB_BUFFER_POOL_STATS`查看

  ```
  buffer pool size # 缓冲池
  free buffers # free list 中页的数量
  database pages # LRU list 中页的数量
  pages made young # 页从 old 移动到 new 次数
  buffer pool hit rate # 缓冲池的命中率：当该值 > 95% 时表明运行良好，否则，需要观察是否存在 由于全表扫描引起了 LRU 列表被污染的情乱！
  ```

缓冲池

- InnoDB 是基于磁盘的数据库系统，它将 行记录 以页的方式进行管理！
- 缓冲池是一个块内存区域，用过内存的速度来弥补磁盘速度较慢对系统性能的影响！

  读取页：将页从磁盘 读取到 缓冲池：将页 “FIX” 到缓冲池！

  修改页：先在 缓冲池中修改，并通过 CheckPoint 机制将其刷新会磁盘，而不是每次更新都刷新磁盘！
- 缓冲池的数据类型：
  - 索引页、数据页、undo 页、插入缓冲（insert buffer）、自适应哈希索引、InnoDB 锁信息（lock info）、数组字典信息等！
  - 缓冲池中并非只是 索引页 与 数据页！

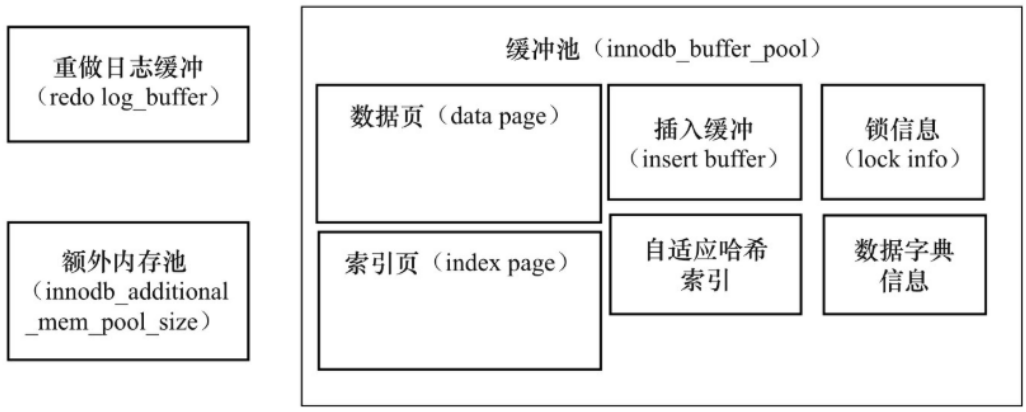


图2-2 InnoDB内存数据对象

- 缓冲池管理：LRU List、Free List、Flush List

缓冲池 LRU List

- 在 InnoDB 存储引擎中，缓冲池中 页的默认大小为 16K，InnoDB 使用 LRU 算法管理缓冲池！

  与 传统的 LRU 算法不同，InnoDB 对 LRU 做了优化！

- InnoDB 为 LRU List 维护了一个 中点（midpoint），每次被访问的页都会被插入到 midpoin 位置，这个算法被称为 中点插入策略。

  InnoDB 存储引擎将 midpoint 之后的列表称为 old 列表，之前的列表称为 new 列表，可以简单的认为 new 列表中的数据都是活跃的热点数据！

  默认配置下，该位置在 LRU 列表长度的 5/8 处（距离尾部 3/8 处），可以通过 innodb_old_blocks_pct 设置：

  ```
  # 设置 old 端大小(表示整个 List 的 30%)
  set global innodb_old_blocks_pct=30;

  # 该值得到的是 百分数，表示插入到 LRU 列表尾端的距离
  show variables like 'innodb_old_blocks_pct'\G
  ```

- 为什么不使用传统的 LRU，将读取的页放到 LRU列表 首部呢？

  这是因为，如果直接将读取到的页放到 LRU 首部，那么某些 SQL 操作可能会使缓冲池中的页被刷新出去，从而影响缓冲池的效率，常见操作为： 索引 或者 数据 的扫描操作！

  这类操作需要访问表中的许多页，甚至是全部的页，而这些页通常来说又仅仅是在本次查询中使用到，并不是活跃的热点数据。

  如果页被放入 LRU 列表的首部，那么非常可能将所需要的热点数据页从 LRU 列表中移除，而下一次需要获取该页时，InnoDB 存储引擎需要再次访问磁盘！

- Innodb_old_blocks_time：表示页读取到 midpoint 位置之后，需要等待多久才会被加入到 LRU 列表的 热端！

  当需要执行全表扫描（索引扫描、数据扫描）时，可以将该值设置得较大，防止冷数据进入到 热端！

  ```
  page made young : 页从 LRU 列表的 old 部分加入到 new 部分 的数量
  page not made young : 由于 innodb_old_blocks_time，导致页没有从 old 移动到 new 部分 的数量
  ```

  ```
  # 设置
  set global innodb_old_blocks_time=1000;
  # 如果用户预估自己活跃的热点数据不止 63% (即为：5/8)，则可将 innodb_old_blocks_pct 将 old 端设置的较小！
  ```

缓冲池 Free List

- LRU List 用来管理已经读取的页，但当数据库刚启动时，LRU 列表是空的，没有任何页！

  此时的页 都放在 Free 列表中。

- 当 需要从 缓冲池中分页时，首先从 Free 列表中查找是否有可用的空闲页，若有，则将该页从 Free 列表中删除，放到 LRU 列表中。否则，根据 LRU 算法淘汰 LRU 列表尾部的 页，将该内存空间分配给新的页！

缓冲池 Flush List

- 在 LRU 列表中的页被修改之后，该页便被称为 “脏页”，即：缓冲池中的页磁盘中的页数据不一致了！

  此时 数据库会通过 CheckPoint 机制将脏页刷新到磁盘中，而被修改的 页 就会被存放到 Flush 列表中！

- 注意：“脏页” 既存在于 LRU 列表中，也存在于 Flush 列表中，两者作用不同，且互不影响：
  - LRU 列表中的脏页：用来管理缓冲池中页的可用性
  - Flush 列表中的脏页：用来管理将 页 刷新回磁盘
- 倘若每次一个页发生变化，就将新页的版本刷新到磁盘，那么这个开销是非常大的；同时，如果从缓冲池将页的新版本刷新到磁盘时发生了宕机，那么数据就不能恢复了，为避免这种状况，数据库普遍采用了 Write Ahead Log 策略

  Write Ahead Log 策略：当事务提交时，先写重做日志，再修改页。如果出现宕机 而 导致数据丢失，则通过重做日志完成数据恢复 —— 这也是 ACID 中 D 的要求！

重做日志缓冲

- 重做日志缓冲（ redo log buffer）

  InnoDB 存储引擎 首先 将重做日志信息 先放到这个缓冲区，然后按照一定频率将其刷新到重做日志文件。

  ```
  重做日志（Redo Log）记录的是 数据页 的 物理修改！
  回滚日志（Undo Log）记录的是 数据库 的 修改操作！
  ```

- 重做日志缓冲刷新到磁盘的情况：

  1. Master Thread 每一秒将重做日志缓冲刷新到重做日志文件！
  2. 每个事务提交时，会将重做日志缓冲刷新到重做日志文件！
  3. 当重做日志缓冲池 剩余空间 小于 1/2 时，重做日志缓冲刷新到重做日志文件！

- CheckPoint（检查点） 机制**？？？**

  作用：

  1. 缩短数据库的恢复时间！

     ```
     当数据库 宕机 时，数据库不需要重做所有日志，因为 CheckPoint 之前的页都已经被刷新到磁盘。
     数据库只需要对 Checkpoint 之后的重做日志进行恢复！这样大大缩短了数据库的恢复时间！
     ```

  2. 缓冲池空间不够用时，将脏页刷新到磁盘！

     ```
     当缓冲池不够用时，根据 LRU 算法会溢出最近最少使用的 页，如果该页为 脏页，那么需要强制执行 Checkpoint，将脏页刷新到磁盘！
     ```

  3. 重做日志不可用时，将脏页刷新到磁盘！

     ```
     redo log buffer 是一个环状的缓冲区，当 buffer 满了之后，新的重做日志就会覆盖就的重做日志：
         1. 此时，如果重做日志不再需要（已经被刷新到磁盘了），则直接覆盖！
         2. 如果 旧的重做日志还需要使用（未刷新到磁盘），则强制执行 Checkpoint，将缓冲池中的页至少刷新到当前重做日志的位置！
     ```

索引

B+ Tree 索引

- B+ 树是 B树 的变种，是一棵 平衡的多叉树！
- **B 树 特性**：

  1. *首先是一棵 m 叉查找树！（有序性）*
  2. *任意一个结点中最多包含 m 个子结点：结点中最多包含 m-1 个关键字！*

3. *对于 根结点，至少有 两个 孩子结点，除非 根结点 就是叶子结点！*
4. *对于 非根结点外的所有 非叶子结点，结点至少有 `m/2（向上取整）` 个子结点！*
5. *所有叶子结点在同一层，并且，都是空结点！*

- B 树 变种

  B+ 树在 B树 的基础上，做了两个改变：

  1. **非叶子结点中只存放索引，叶子结点中只存放完整的数据！**

     B 树 不需要遍历到叶子结点 就能返回数据，而 B+ 树 必须遍历到叶子结点，才能获取数据页！

  2. **叶子结点之间通过 指针 前后连接，形成（有序）链表！**

     方便范围查询时，通过链表访问数据，而不需要像 B树 一样遍历树！

  当 B+ 树不满足 B树 定义时，就会通过 合并 / 分裂达到定义标准！

- **B+ 树**

  InnoDB 存储结构就是 B+ 树，树的 非叶子结点存放了 索引，而 所有数据存放在 叶子结点中！对于 InnoDB 来说，一个叶子结点 对应了 一页（页大小固定，通常为 16KB）。InnoDB 所有的页都在同一层，并且页与页之间通过指针相连，形成链表！

  注意：*正是因为 叶子结点中存放的是页，所以 InnoDB 存储引擎并不能在 B+树 中找到具体的数据行，它只通过 索引检索到 数据行所在的 页，并将页完整的读入到内存中，在内存中查找到数据* ！

- 使用场景
  - 经常更新的表，适合处理多重并发的更新请求
  - 支持事务
  - 可以从灾难中恢复 (使用 bin-log 日志等可以恢复)
  - 仅 innodb 支持外键约束
  - 支持 自增列属性（只有主键为数值类型时可自增）

MySQL 索引结构

- B+ Tree 索引（B+ 树）

```
<!--
 · B+ 树索引使用 B+Tree 实现，所有数据存放到叶子结点中，并且叶子结点之间使用指针相连，链表！
    由于 B+Tree 的有序性，所以叶子节点链表是有序链表！

· 优点
    1. B+Tree 索引不需要使用全文扫描，只需要对树进行检索，能有效加快检索速度！
```

```
    2. 由于 B+Tree 中的数据都是有序存放，这不仅方便范围查询，还能够高效的排序 以及 分组实现！
    3. 允许指定多个列作为组合索引，多个索引列共同构成一个树结点中的一个键！
    4. B+ 树支持 全键值、键值范围、键前缀查找（最左前缀）。如果不是按照索引列的顺序进行查找则无法使用索引？
    5. MySQL 只有 B+TRee 支持覆盖索引，因为其他索引结构都不会存储 索引列 的值！

· InnoDB 的 B+Tree 索引分为 主索引和辅助索引：
    聚族索引（主索引）：按照主键值 构建 B+Tree，叶子结点中存放的完整的数据，一个表中只能由一个聚族索引！
    辅助索引：按照其他索引列 构建 B+Tree，叶子节点中存放的是 主索引的值。在辅助索引中 检索 主键值，再到主索引树中查找！

· 主键值 是 一列或者多列 的组合，其值能够唯一标识表中的每一行，通过它可以强制表的实体的完整性！
-->
```

- Hash 索引

```
<!--
 · Hash 索引基于 哈希表实现，能以 O(1) 时间查找，但是无法保证数据的有序性！
    对于每一行数据，存储引擎都会对所有的索引列计算一个 哈希码（hash code）！
    哈希索引 将所有的 哈希码 存储在索引中，同时在哈希表中保留 指向每个数据行 的指针！如果出现 哈希冲突，索引会以 链表的方式存放多个记录到同一个 哈希条目中！

 · 优点：
        能快速地查找！
 · 缺点
        1. 数据无序存放，因此不支持 排序和分组！
        2. 由于计算的是 索引值为 哈希码，因此，只支持等值匹配，无法模糊查询、部分查找、范围查询！
            等值比较查询，包括 "="、in、"<=>"（注意:<> 和 <=> 相反）！
        3. 哈希表的哈希的是 所有索引列的 哈希值，而非部分索引列，因此，必须是等值匹配 所有索引列！
            不支持部分索引列 匹配！
        4. 当访问到 哈希冲突 的行时，存储引擎会遍历 链表中的所有行指针，逐行比较，找出所有符合条件的行！
        5. 如果哈希冲突很多，那么索引段 维护代价也很高：
            当删除某一行时，存储引擎需要遍历对应哈希值的链表中的每一行，并找到应该删除的行，冲突越多，代价越大！
```

```
· Memory 引擎默认使用 Hash 索引，同时也支持 B+Tree。

· InnoDB 支持 “自适应的 哈希 索引”
    当 InnoDB 注意到某些索引值被使用的非常频繁时，它会在内存中，基于 B+Tree 索引之上，再创建一个 哈希索引，这样就能让 InnoDB 借助 哈希索引 加快查找速度！
-->
```

- 空间数据索引（R-Tree）
  - MyISAN 存储引擎支持 空间索引，可以用作地理数据存储！
  - 空间索引会从所有维度 来 索引数据，可以有效的使用任意维度来组合查询！
- 全文索引（FULLTEXT）
  - 全文索引是一种特殊类型的索引，它用于查找文本中的关键词，而不是直接比较是否相等！它使用 MATCH AGAINST 查找关键字，而 不是普通的 WHERE！
  - 全文索引使用 **倒排序** 实现，它记录着关键词到其所在文档的映射！
  - 应用

    MyISAM 存储引擎支持 全文索引；

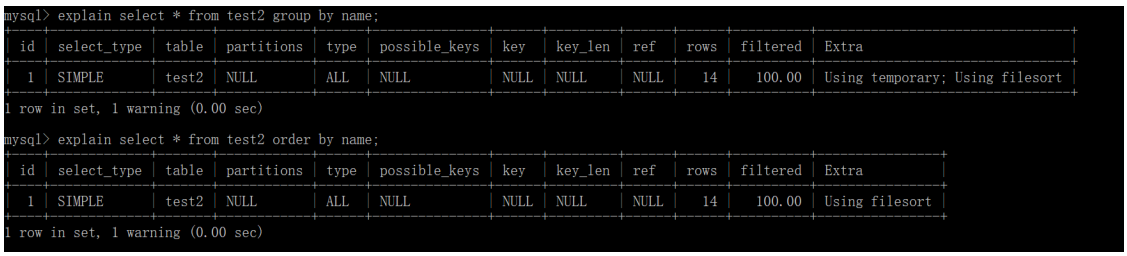
    MySQL 5.6.4 开始，InnoDB 存储引擎也支持 全文索引！在相同的列上同时创建全文索引 和 基于值的 B+Tree 索引并不会有冲突。

索引优点

1. 能够高效地检索数据
2. 由于 B+Tree 的有序性，与 索引 相关的值都会存储在一起，MySQL 能够通过 order by 和 group by 轻松的实现 排序和分组，而避免创建 临时表！

   对于普通列 的排序和分组，需要创建 临时表 实现！

   如下：test2 表中的 name 字段是普通字段，分别对其执行 order by 和 group by



3. 索引可以将随机 IO 变为 顺序 IO（B+Tree 索引是有序的，会将相邻的数据都存储在一起）

索引分类

唯一索引 vs 主键索引

- **唯一索引（unique）**

  索引列 不允许 出现相同的索引值，确保索引的唯一性！

- **主键索引（primary key）**

  主键索引 是 唯一索引的特殊类型，由 一列或者多列 的组成，主键值能够唯一标识表中的每一行，通过它可以强制表的实体的完整性！

  主键索引不可为空！

  InnoDB 会按照主键值 创建 存储引擎表，它决定了数据的存放顺序！

聚集索引 vs 辅助索引

- InnoDB 的 B+Tree 索引分为 主索引和辅助索引！
- **聚集索引（主索引）**

  按照主键值 构建 B+Tree，非叶子结点中存放索引，叶子结点中存放的完整的数据（数据页）！

  由于实际的数据页只能按照一棵 B+ 树进行排序，索引一个表中只能有一个聚族索引！

  查询优化器更倾向于采用聚集索引，因为聚集索引能够在叶子节点上直接找到数据，同时 叶子节点的数据有序，聚集索引也能快速地访问针对范围值的查询！

  ```
  1. InnoDB 存储引擎表 是 索引组织表，即表中数据按照主键顺序存放，而 聚集索引 就是按照每张表的主键构造的 B+Tree！
  2. 聚集索引的特性 决定了 索引组织表中的数据 也是索引的一部分！
  3. 聚集索引 的存储 是逻辑上连续的，而非物理连续：
      1. 页是通过 双向链表连接！
      2. 每个页的记录也是通过双向链表连续的！
  ```

- **非聚集索引（辅助索引）**

  按照其他索引列 构建 B+Tree，叶子节点中存放的是 主索引的值。在辅助索引中 检索 主键值，再到 聚集索引 中查找具体的数据！

  辅助索引的存在 并不影响据定数据的存放，数据的存放取决于聚集索引，因此，表中可以有多个辅助索引！

索引失效

1. 在查询条件中 计算索引字段 或者 在索引字段上使用函数 ！

   会对索引全表扫描，计算每一个索引！

2. 在索引字段上出现 隐式类型转换

   例如：索引字段为 varchar 类型，但参数为 int 类型

3. 使用 复合索引（组合索引）时，不遵循最左前缀规则！

   复合索引：InnoDB 会按照 列的定义顺序 生成键值，检索时，从左往右 顺序匹配！

```
<!--
 · 例如：在 col1 + col2 + col3 上建立组合索引！

 · 则：
    col1 (+ col3)       ：只用 col1 匹配
    col1+col2            ：使用 col1 匹配后，再使用 col2 匹配
    col1+col2+col3       ：从左往右依次匹配！

 · 而如下则无法使用索引：因为 InnoDB 的结点键值是从左往右匹配的，左边的未知那将无法索引！
    col2
    col3
    col2+col3
-->
```

4. 模糊查询时，对前缀进行模糊匹配（以 % 开头）！

   键值的 从左往右 匹配，如果前缀未知，那将无法走索引匹配的方式！

5. 使用 or 分隔开的条件，如果 or 前的条件中包含索引列，而后面的列中没有索引，那么涉及到的索引将不会被用到！

   我测试的结果：
   使用 or 时，只有当 select 后面的列为 索引列，并且 or 前后必须同属于符合索引中的列，才会使用索引！

6. 如果 mysql 优化器觉得使用索引比全表扫描慢，则会放弃使用索引！

索引优化

组合索引

- 以多列 作为条件进行查询时，使用 组合索引 比 多个单列索引 性能更好： ？

  多个单例索引：数据库会创建多个 辅助索引

- 例如：

```
select * from school where class_id = 1 and student_id = 2;

# 应将 class_id 和 student_id 作为组合索引!
```

组合索引 顺序

- 应将 选择性最强的列 放在最前面
  - 选择性：不重复的索引值总数 与 行记录总数 的比值。

    选择性 越高 说明，索引值的区分度越高，当选择性 为 1 时，表明每一行都被索引唯一标识!

    索引列 的 区分度越高，查询效率就 越高!
- 例如：

```
select
    count( distinct staff_id) / count(*) as
staff_id_selectivity,
    count( distinct customer_id) / count(*) as
customer_id_selectivity,
    count(*)
from payment;

# 得出结果如下时:
  staff_id_selectivity      : 0.0001
  customer_id_selectivity   : 0.0373
            COUNT(*)        : 16049

# 说明  customer_id 区分度更高，因此，组合索引时，应将其放在最前面!
```

前缀索引（短索引）

- 如果对 字符串列(BLOB、TEXT、VARCHAR) 进行索引，必须使用前缀索引!

  如果索引前 n 个字符就 具有高度区分性，那么应该指定前 n 个字符为该列的索引!
- 短索引可以提高查询速度，节省磁盘IO

  短索引 不仅能够节省大量的索引空间（索引越长，结点中的键占用空间越大），还能较快的对比! 此外索引高速缓存中的块能够容纳更多的键值，增加了 不用读取索引中较多块的可能性!
- 例如：

```
alter table table_1 add index `idx_test`( test(20) );
# 在 table_1 中为 test 列增加一个索引，指定 test 的前 20 个字符为索引 idx_test !
```

覆盖索引

- 覆盖索引：索引中 包含了 所要查询的字段值！

  > 通过 expalin 查看时， Extra = using index 就表明 从索引中返回值，而不需要从数据页中获取！

- 优点

  1. 索引通常远 小于 数据行的大小，只读取索引能够大大减少数据访问量！
  2. 一些存储引擎（例如：MyISAM）在内存中只缓存索引，而数据依赖于操作系统来缓存。

     因此，只访问索引，可以比避免系统调用（系统调用比较耗时）
  3. 对于 InnoDB 存储引擎，若辅助索引能够覆盖查询结果，则无需访问主索引！

- 注意：

  并不是所有的 索引结构 都支持覆盖索引：覆盖索引 必须要存储索引列的值，而哈希索引、空间索引、全文索引 都不会存储索引列的值，因此 MySQL 只能用 B+Tree 索引实现 覆盖索引！

删除索引

- 索引并不是越多越好，InnoDB 需要维护每一个索引，因此，如果对于不常使用的列，应该避免创建索引！
- 去掉重复索引
  - Primary、Unique、Index ： 对于 MySQL 而言 Primary、Unique 都是通过 Index 实现的！

    如果一个列上同时创建了这三个索引，那么，相当于重复创建了三个 Index 索引，不可取！
  - 除非在同一列上创建 多种不同类型的索引，以满足不同的需求，例如：KEY(col) 于 FULLTEXT(col)
- 去掉 冗余索引
  - 如果 某一列 当作组合索引的 前缀，同时又将该列作为 单独的索引，这就是冗余索引！

    因为组合索引就是应该使用 最左前缀原则！

例如：KEY(A, B) 与 KEY(A)

- 冗余索引 通常发生在 为表 添加新索引的时候，此时，应该尽量在原索引列上扩展新索引，而不是 创建一个新的组合索引！

索引使用场景

- 索引用于优化查询数据，提高数据库查询性能！
  1、经常作为查询选择的字段：适合作为索引
  2、经常作为表连接的字段：适合作为索引
  3、经常出现在在order by、group by、distinct等关键字后面的字段：适合作为索引

查看索引使用情况

- Handler_read_key 表示了某一行被索引值读的次数

  若该值很低，说明索引不经常使用，增加索引并不能改善性能！

- Handler_read_rnd_next 表示在数据文件中读取下一行的请求数！

  若该值很高，说明正在进行大量的表扫描，查询低效 —— 说明表索引不正确，或者查询没有使用到索引！

```
show status like 'Handler_read%'
```

查询性能优化

- Explain 用来查看 优化器的执行计划，我们可以通过 Explain 结果来优化查询语句！

  explain 详解

  ```
  explain # 查询执行计划
  show warnings; # 根据查询计划，查看具体执行 sql 语句
  ```

优化 数据访问

- 减少请求的数据量
  - 只返回必要的列：不要使用 SELECT *
  - 只返回必要的行：使用 LIMIT 做分页处理
  - 缓存重复查询的数据：将热点数据缓存，可以避免大量请求压到数据库，造成大量的重复的查询，使用缓存可以大福度提升性能！
- 减少 MySQL 扫描的行数
  - 最有效的方式就是 使用 索引覆盖查询！

重构 查询方式

- 切分大查询

  如果一次性执行一个大查询，可能会一次锁住很多数据，阻塞很多小的 但 比较重要的查询，同时占满整个事务日志，耗尽系统资源！

  - 例如：

    ```
    DELETE FROM messages WHERE create < DATE_SUB(NOW(),
    INTERVAL 3 MONTH);
    ```

    可以通过 JDBC 限制每次删除 1000 个！

    ```
    rows_affected = 0
    do {
        rows_affected = do_query("DELETE FROM messages WHERE
    create  < DATE_SUB(NOW(), INTERVAL 3 MONTH) LIMIT 10000");
    } while rows_affected > 0
    ```

- 分解大连接查询

  将一个 大连接查询 分解成 对每一个表 进行一次单表查询，然后在应用程序种进行关联。

  优点：

  - 让缓存更高效

    对于连接查询，如果一个表发生了表化，那么整个查询缓存就无法使用。而分解后的多个查询，即使其中一个表发生了表化，对其他表的缓存依然起作用！

  - 分解成多个单表查询，这些单表查询的缓存结果更可能被其他查询使用到，从而减少了冗余的记录查询！

  - 减少锁竞争

    一次性 大连接查询 会同时锁住多个表，这将会阻塞其他很小的但是重要的查询，同时也会占满整个事务日志，耗尽系统资源！

  - 在应用层进行连接，可以更加容易的对数据库进行拆分，从而更加容易的做到高性能 和 可伸缩性！

  - 查询本身的效率有所提升！

    例如：使用 IN() 代替连接查询，可以使得 MySQL 按照 ID 顺序进行查询，这可能比随机连接要更高效！

```
select * from tag
join tag_post on tag_post.id=tag.id
join post on tag_post.post_id=post_id
where tag.tag='mysql';

# 可在应用层将其拆分成多个查询语句，并关联查询!
select * from tag where tag='mysql';
select * from tag_post where tag_id=1234;
select * from post where post.id IN(123,45,23,543);
```

优化 范围查询

1. 若条件查询中涉及到范围查询，则尽量将 范围查询子句 移到 索引查询的后面，以便优化器能够更多的利用到索引！因为查询只能使用索引的最左前缀，直到遇到第一个范围条件列，在范围查询之后的将不再会用到 索引 查询!
2. 可以使用 IN() 代替范围查询，从 EXPLAIN 执行计划来看，范围查询和 IN() 的 type 都是 range ， 但是两者是不同的!
    - 范围查询 是从值的范围访问数据!
    - IN() 是通过 多个等值条件查询的!

      对于范围条件查询，MySQL 无法再使用范围列后面的其他索引列了，而对于 “多个等值条件查询” 则没有这个限制!

- 也不能滥用 IN() 条件，每额外增加一个IN() 条件，优化器需要做的组合都将以指数的形式增加，最终可能会极大地降低查询性能!

优化 排序

- 文件排序 对小数据集而言是很快的，但是查询上百万时，需要对其优化!
    1. 使用 limit 做分页，一次处理少量数据

       分页时：如果用户翻到了比较靠后的页，也将耗费大量时间：`limit 1000000，10` 因为这意味着 排好序后的 前 1000000 条数据都没有用，而只需要这 10 条，丢弃了大量的行数据，非常浪费资源!

       ```
       select * from table_1 where sex='M' order by rating limit 100000,10;
       ```

    2. 加索引

       ```
       KEY( sex,rating ) # 尽管 sex 是一个低识别率的列!
       ```

       一定程度上能够提升性能，但是，无论是否创建索引，这种查询都是个严重的问题!

3. 延迟关联

   通过 连接查询，在子查询种覆盖索引，再将根据主键值 回表访问数据！能有效提升性能！

```
select <cols> from table_1
inner join (
    select primary_key_cols from table_1 where
table_1.sex='M' order by rating limit 1000000,10
) AS temp using(primary_col_cols);
```

避免 索引失效

1. 不要在索引列上使用函数，也不要 让索引列参与表达式运算！
2. 避免类型的隐式转换
3. 复合索引时，遵循最左前缀原则
4. 模糊查询时，应精确匹配前缀，模糊匹配后面的内容！
5. 避免使用 or 判断！

其他优化

优化 INSERT

- 如果执行批量插入，应尽量使用 多个值表 的 inert 语句

  这种方式将大大缩短客户端和数据库之间的连接、关闭等消耗，使得效率比分开执行的单个 insert 更好！

```
insert into table_1 ... values(...),(...),(...);
```

- 如果从不同的客户端批量插入很多行
  - 使用 DELAYED 可以令 insert 立即执行

    其实数据都被放到内存的队列中，并没有真正写入磁盘！
  - 使用 LOW_PRIORITY 则刚好相反！

    在所有其他用户对表的读写完成之后才进行插入！
- 当从一个 文本文件 装载入一个表时，使用 LOAD DATA INFILE 命令，比通常的 INSERT 高效20倍！

优化 Order By

尽量减少额外的排序，通过索引直接返回有序数据：

WHERE条件和ORDER BY使用相同的索引，并且ORDER BY的顺序和索引顺序相同，并且ORDER BY的字段都是升序或者都是降序。否则肯定需要额外的排序操作，这样就会出现Filesort。

优化 Group By

默认情况下，MySQL对所有GROUP BY col1,col2,…的字段进行排序。这与在查询中指定ORDER BY col1,col2,…类似。

因此，如果显式包括一个包含相同列的ORDER BY子句，则对MySQL的实际执行性能没有什么影响。

如果查询包括GROUP BY但用户想要避免排序结果的消耗，则可以指定ORDER BY NULL禁止排序，这样能避免在非索引列上排序耗费大量时间！（filesort）

优化 子查询

有些情况下：子查询 可以 被更有效率的 Join 替代

优化 OR 条件

- 对于含有OR的查询子句，如果要利用索引，则OR之间的每个条件列都必须用到索引；
- 如果没有索引，则应该考虑增加索

IN 与 NOT IN

- 慎用 in 和 not in 查询
  - in 一般来说会使用 索引，但是如果使用 in+子查询 ，则外层循环 将会做全表扫描，可以将其改成 表连接！或者 使用 exists 替换 in 查询！

```
# 例如：外层查询会全表扫描
select id from t1 where num in(select id from t2 where id
> 10);

# 改成：表连接
select id from t1,(select id from t1 where id > 10)t2
where t1.id = t2.id;
```

```
<!--
 · 原因：使用 临时表算法 实现的视图，在某些时候性能会很糟糕 ！
    MYSQL 以递归的方式执行这类视图，先会执行外层查询，即使外层查询优化器将其优化的很好，但是 Mysql 优化器可能无法像其他数据库那样做更多的内外结合优化，外层查询的 WHERE 条件无法“下推”到构建视图的临时表的查询中，临时表也无法建立索引！
-->
```

  - not in 不会使用 索引！(不会使用非主键索引？)

表连接

外联 优于 内联！

因为 外连接有基础数据：
    如 `A left join B` ：基础数据是 A
    而内联：`A inner join B` ：无基础数据，会先使用 笛卡尔积，再根据联接条件得到内联接结果集

其他

- 尽量不使用 * 、union、union all 、or关键字，尽量使用 等值查询！
- 尽量避免 where 字句中进行 null 值判断，否则引擎将放弃索引，而全表扫描
- 避免不等值判断

  尽量避免在 where字句 中使用 "!=、<>"等判断，否则 mysql 将放弃使用索引（放弃非主键索引？），进行全表扫描

存储引擎

InnoDB

InnoDB 是 MySQL5.5之后 默认的存储引擎，表的索引和数据存放在一起！

1. 支持事务，默认工作在 可重复读 级别。

   InnoDB 通过 MVCC 实现了 已提交读、可重复读级别的 无锁的一致性读，在 增删改 时，通过 Next-Key Locking 解决了幻读问题！（支持 事务、MVCC、行级锁）
2. InnoDB 默认使用 B+Tree 作为存储结构，使其支持高效的 范围查询、分组、排序 等操作！
3. 以 B+Tree 作为存储结构，使其支持 聚族索引，叶子结点中存放完整的数据页，而非叶子结点存放 主键值 作为索引！

   由于 B+Tree 的非叶子结点中存放的是索引列的值，因此 也支持覆盖索引，在某些情况下，可以直接返回数据，而无需访问磁盘，提高了查询性能！
4. InnoDB 在内部做了很多优化：
   - InnoDB 支持自适应的 哈希索引，如果 InnoDB 发现在某个索引值上，在 B+Tree 索引的基础上，建立 哈希索引，优化查询性能！
   - 磁盘预读特性

> · 为了减少 磁盘 IO 操作，磁盘往往不是严格的按需读取，而是每次都会预读！
> · 由于 B+Tree 的叶子结点有序，因此，读取数据的同时，可以将相邻结点预先载入内存中，对于相邻结点的数据页，磁盘会进行顺序读取，只需要很短的磁盘旋转时间，而不需要耗费磁盘寻道时间，非常快速！
>
> · 操作系统一般将内存和磁盘分割成固定大小的块，每一块称为一页，内存与磁盘以页为单位交换数据。数据库系统 中的一个叶子结点就对应了一页，每次查询时，都会将一页数据载入程序中，在程序中查找具体的行！

- 采用插入缓冲区 加速 插入操作！

  先将插入的数据放到缓冲区中，并未真正的插入到磁盘！

5. InnoDB 支持在线的热备份

   其他存储引擎不支持在线热备份，要获取一致性视图需要停止对所有表的写入，儿再读写混合的场景中，停止写入也可能意味着停止读取！

MyISAM

MySQL 5.1 及之前，MyISAM 是默认存储引擎，表存储在 数据文件和索引文件中！

- 提供了大量的特性，例如：全文索引、压缩表、空间数据索引等！
- 不支持事务，也不支持行级锁，只支持表锁！

  MyISAM 通过表锁，会一次性获取所需要的全部锁，避免死锁的出现！

  读取数据时，会对需要读到的表加上表共享锁，写入时，则对表加排他锁！

  读操作与读操作兼容，读写操作相互排除；但是 MyISAM 也支持读操作时，往表的末尾插入数据，这被称为并发插入（Concurrent Insert）

  ```
  lock table table_1 read;
  lock table table_1 read local; # 通过 local 指定，读操作时，允许在表的末尾插入数据 ：并发插入
  lock table table_1 write; # 排他锁
  ```

- 数据库崩溃后 无法安全恢复！

  MyISAM 可以用在 只读数据 或者 表比较小并可以容忍修复操作的 场景！

- 创建表是，如果指定了 DELAY_KEY_WRITE 选项，在每次修改执行完成之后，不会立即将修改的索引数据写入磁盘，而是会写入内存中的键缓冲区，只有在清理缓冲区或者关闭表的时候才会将对应的索引块写入磁盘。

  这种方式可以极大的提升写入性能，但是在数据库或者主机崩溃的时候会造成索引损坏，需要执行修复操作！

- 缓存

  > 对于 `MyISAM` 存储引擎表，`MySQL` 只缓存器索引文件，数据文件的缓存交由操作系统本身来完成，这与其他使用 `LRU` 算法缓存数据大部分数据库大不相同！

InnoDB vs MyISAM

(默认存储结构都是 B+Tree)

1. 事务：InnoDB 支持事务，可以通过 commit 提交，以及 rollback 回滚
2. 锁：MyISAM 支持表锁，InnoDB 支持表锁和行锁
3. 外键：MyISAM 不支持外键，InnoDB 支持外键
4. 备份：InnoDB 支持在线热备份，MyISAM 不支持！
5. 崩溃恢复：MyISAM 崩溃后，发生数据损坏的概率比 InnoDB 高，并且恢复速度慢！
6. 其他区别：

   InnoDB 支持聚集索引，数据和索引存放在同一个文件中，MyISAM 数据 和 索引分开存放！

   使用 Count(*) InnoDB 会全表扫描计数，而 MyISAM 维护了一个 记录行数的值，可以直接取出！

数据类型

字符串

- CHAR

  CHAR 列的长度固定，在创建表时指定，长度可以为 0~225 的任何值！

  > 定长：
  > - 超过长度的字符将不会保存！
  > - 定长：程序处理快，但是浪费空间，还需要处理尾空格！

  CHAR 列会删除尾部的空格！

- VARCHAR

  VARCHAR 列是可变长，长度可以指定 0~65535 的任何值！

  > 可变长：
  > - 实际存储的字符串长度取决于插入的字符串！
  > - 节省空间

  VARCHAR 保留尾部空格！

- CHAR vs VARCHAR

  > · InnoDB存储引擎：建议使用VARCHAR类型。
  >
  > 对于InnoDB数据表，内部的行存储格式没有区分固定长度和可变长度列（所有数据行都使用指向数据列值的头指针），因此在本质上，使用固定长度的CHAR列不一定比使用可变长度VARCHAR列性能要好。因而，主要的性能因素是数据行使用的存储总量。由于CHAR平均占用的空间多于VARCHAR，因此使用VARCHAR来最小化需要处理的数据行的存储总量和磁盘I/O是比较好的。

文本

- TEXT

  TEXT 只能保存字符数据！

- BLOB

  BLOB 能保存二进制数据！

- 注意：

  1. BLOB、TEXT 的使用会引起一些性能问题！

     例如：删除操作：

     删除操作会在数据表中留下大量的 “空洞”，以后填入这些 空洞的记录在插入上会有性能的问题！

     为了提高性能，需要定期执行 `OPTIMIZE TABLE` 整理碎片，避免空洞导致的性能问题！

  2. 可以使用合成的索引来提高大文本字段的查询性能！
  3. 在不必要的时候避免检索大型的 BLOB 和 TEXT
  4. 把 BLOB 和 TEXT 分离到单独的表中！

切分

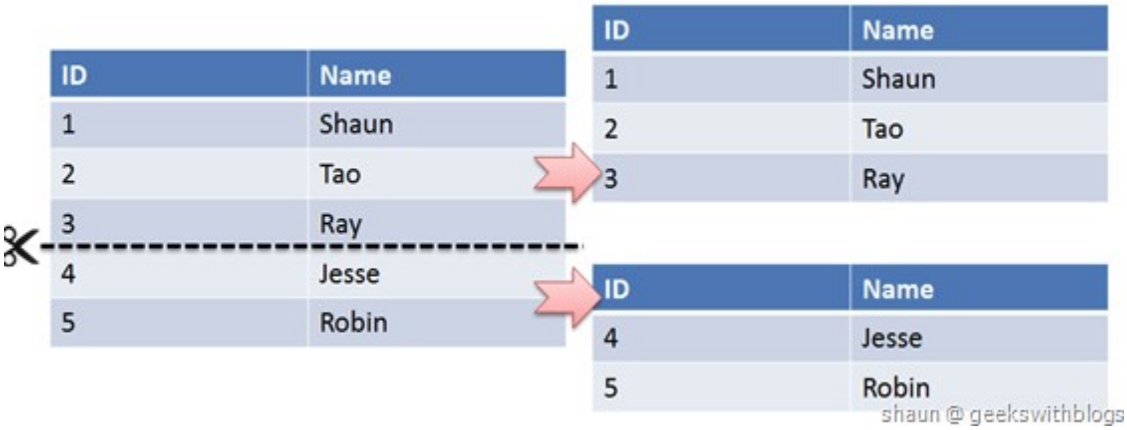
水平切分

介绍

- 水平切分（ Sharding）

  将同一个表中的记录拆分到多个相同结构的表中！

- 当一个表中的数据不断增多时，Sharding 是必然的选择，它可以将数据分布到集群的不同结点上，从而缓解单个数据库的压力！



Sharding 策略

1. 哈希取模：hash(key) % N
2. 范围：可以是 ID 范围，也可以是时间范围！
3. 映射表：使用单独的一个数据库存储映射关系！

Sharding 问题

1. 事务问题

   使用 分布式事务 来解决，例如 XA 接口！

2. 连接问题

   可以将原来的连接分解成多个单表查询，然后再用户程序中进行连接！

3. ID 唯一性 问题

   - 使用全局 ID（CUID）
   - 为每个分片指定一个 ID 范围
   - 分布式 ID 生成器（ Snowflake算法 ）

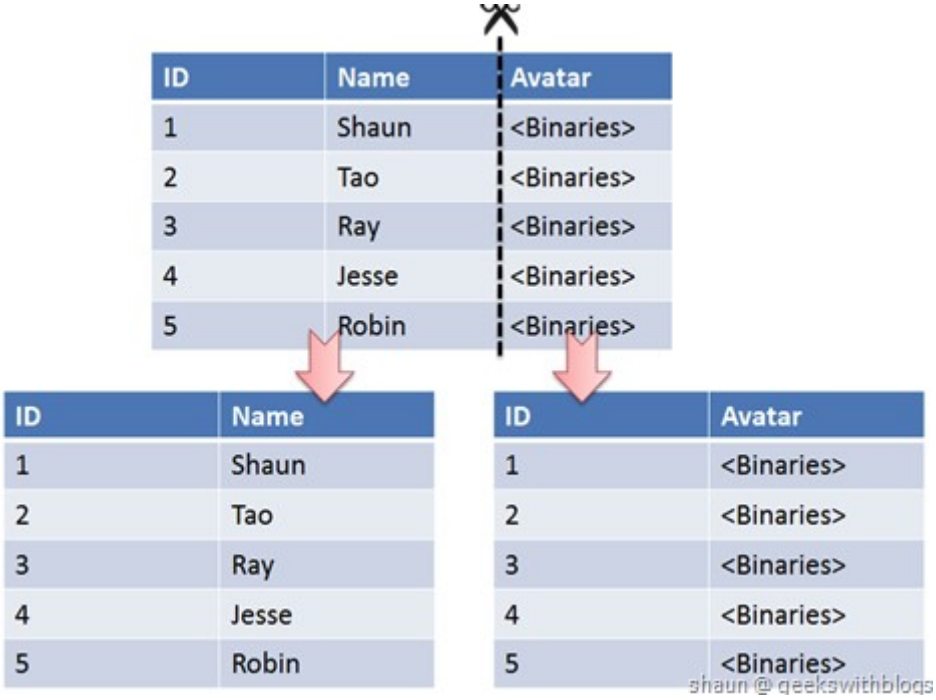
垂直切分

- 垂直切分

  将一张表按列拆分成多个表，通常是按照列的关系密集程度进行切分，也可以利用垂直切分将经常被使用的列和不经常使用的列切分到不同的表中！

- 在数据库层面使用垂直切分，将按照数据库中表的密集程度部署到不同的库中！

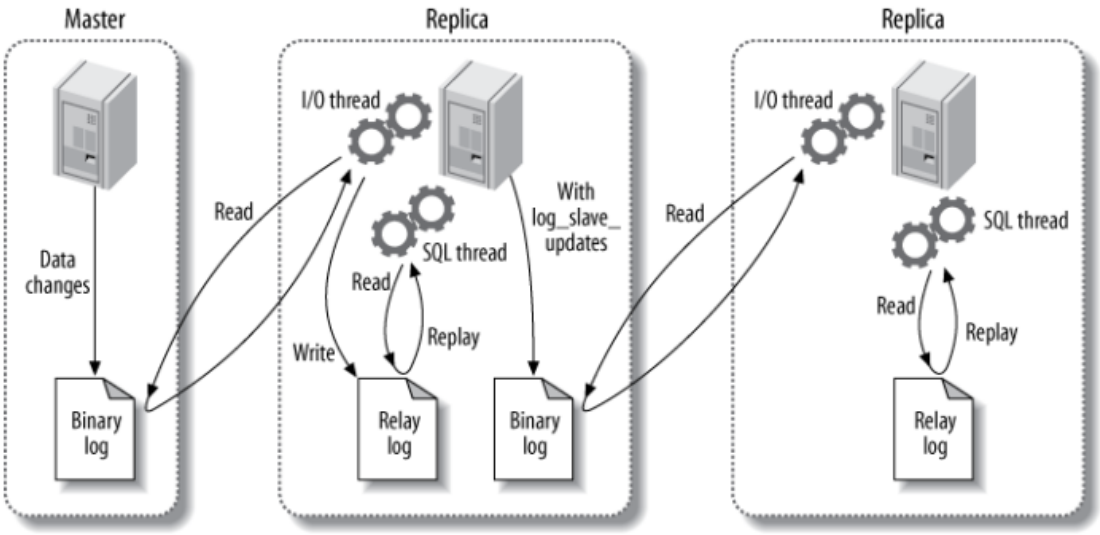
  例如：将原来的电商数据库垂直切分成商品数据库、用户数据库！



复制

主从复制

- 主从复制主要涉及三个线程：binlog 线程、I/O 线程、SQL 线程
  - binlog 线程：负责将主服务器上的数据更改写入二进制文件（Binary log）
  - I/O 线程：负责在 主服务器上读取二进制日志，并写入 从服务器 的中继日志（Relay log）
  - SQL 线程：负责读取中继日志，解析出主服务器 已经执行的数据更改，并在从服务器中重放（Replay）

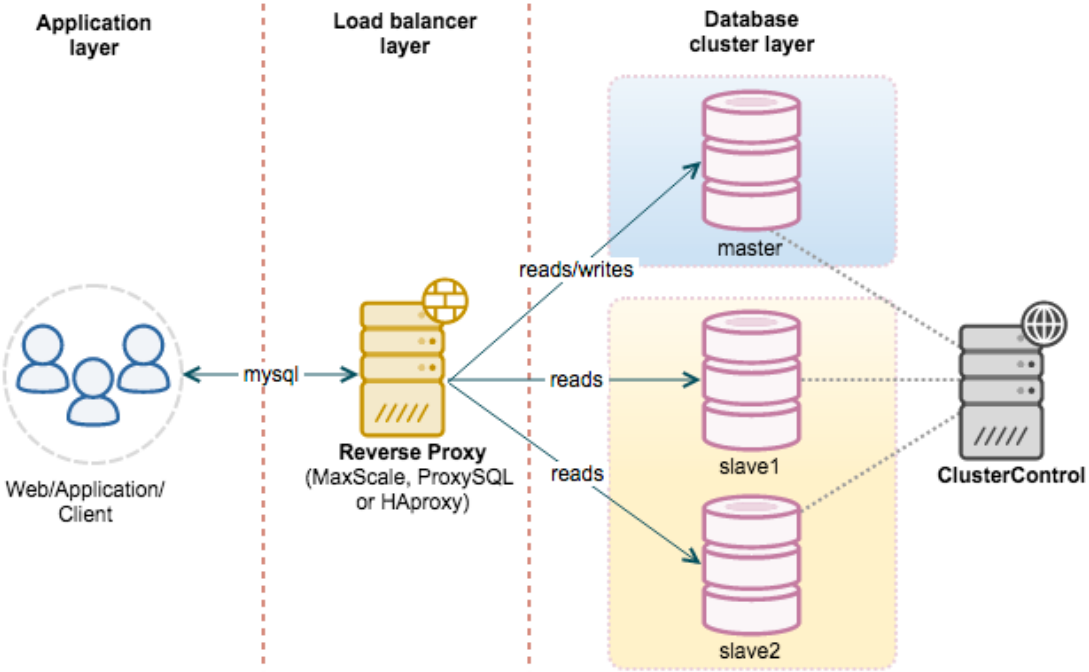


读写分离

- 主服务器上 主要执行 写操作 以及 实时性要求较高的读操作，而从服务器处理读操作!
- 为何能提升性能?

1. 主服务器负责各自的读写，极大程度的缓解了锁的争用！
2. 从服务器可以使用 MyISAM，提升查询性能，并节约系统开销！
3. 增加冗余，提高了可用性！

- 读写分离常用代理实现，代理服务器接收应用层传来的读写请求，然后决定转发到那个服务器！



分布式事务

外部 XA 事务

概要

- InnoDB 存储引擎提供了对 XA 事务的支持，并通过 XA事务 实现分布式事务！
- 分布式事务：

  允许多个独立的事务资源（transactional resources）参与到一个全局的事务中！

  只要参与在全局事务中的每个结点都支持 XA 事务，XA 事务就会允许不同数据库之间的分布式事务！
- 例如：银行转账！

  如果发生的操作不能全部提交或者回滚，那么任何一个结点出现问题都会导致严重的结果！

结构

- XA 事务由 一个或者多个 资源管理器、一个事务管理器 以及 一个应用程序 组成！
  - 资源管理器（Resource Managers）：提供访问事务资源的方法。该管理器必须可以提交或回滚自己管理的事务！通常一个数据库就是一个资源管理

器！
    - 事务管理器（Transaction Manager）：协调参与全局事务中的各个事务，需要和参与全局事务的所有资源管理器进行通信！
    - 应用程序（Application Program）：定义事务边界，指定全局事务中的操作！
- 例如：在MySQL 中

  资源管理器：MySQL 数据库

  事务管理器：连接 MySQL 服务器的客户端

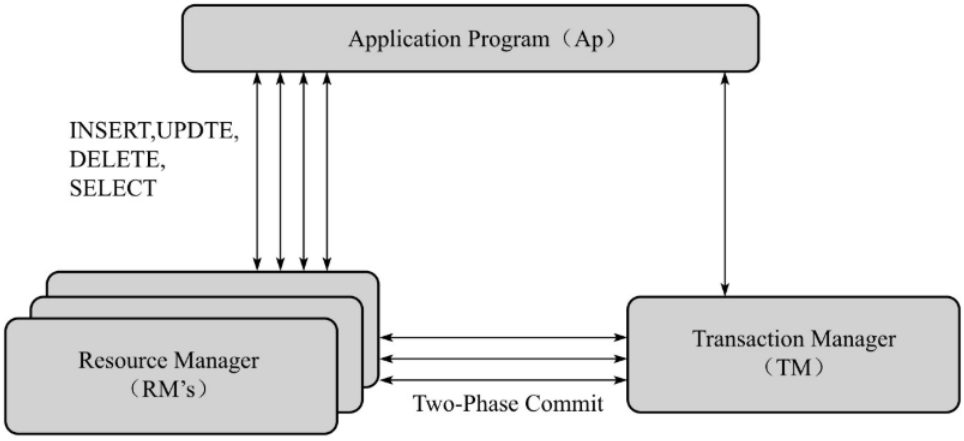


图7-22 分布式事务模型

- 分布式事务采用 两段式提交 的方式！
- 当前 Java 中的 JTA（Java Transaction API）可以很好的支持 MySQL 分布式事务，可以使用 JTA 完成分布式事务！

两段式提交

1. 第一阶段

   所有参与全局事务的结点都开始准备（Prepare），告诉事务管理器他们准备好提交了！

2. 第二阶段

   事务管理器告诉资源管理器执行 rollback 还是 commit！

   如果任意一个结点显示不能提交，则所有的结点都需要被告知 回滚！

- 分布式事务 vs 本地事务

  与本地事务不同的是，分布式事务需要多一次 Prepare 操作，待收到所有结点的同意信息后，再进行 commit 或者 rollback！

内部 XA 事务

- 外部 XA 事务：资源管理器是 MySQL 数据库本身！
- 内部 XA 事务：事务存在于 存储引擎 与 插件之间，或者 存储引擎与存储引擎之间！

  最常见的 内部XA事务 存在于 binlog 和 InnoDB 存储引擎之间！

  > · 由于主从复制，需要开启 `binlog` 功能，主结点将数据更改的内容写入 `binlog` 中！
  >
  > 在事务提交时，先写 二进制日志，再写 `InnoDB` 存储引擎的重做日志！而，这两个写动作要求是原子性的，即：二进制日志 与 重做日志 必须同时写入，若 二进制文件写了，而写入 `InnoDB` 存储引擎时发生了宕机，那么从结点可能会受到 `master`传来的 二进制文件并进行同步！最终导致主从不一致！

  为了解决 这个问题，MySQL 数据库在 binlog 和 InnoDB 之间使用了 XA 事务！
- 步骤：
  1. 当事务提交时，InnoDB 存储引擎会先做一个 Prepare 操作，将事务的 xid 写入！
  2. 接着进行 二进制日志写入！

     如果在 InnoDB 存储引擎提交前，MySQL 数据库宕机了，那么 MySQL 数据库在重启后会先检查准备的 UXID 事务是否已经提交！若没有，则存储引擎层 再进行一次 提交操作！

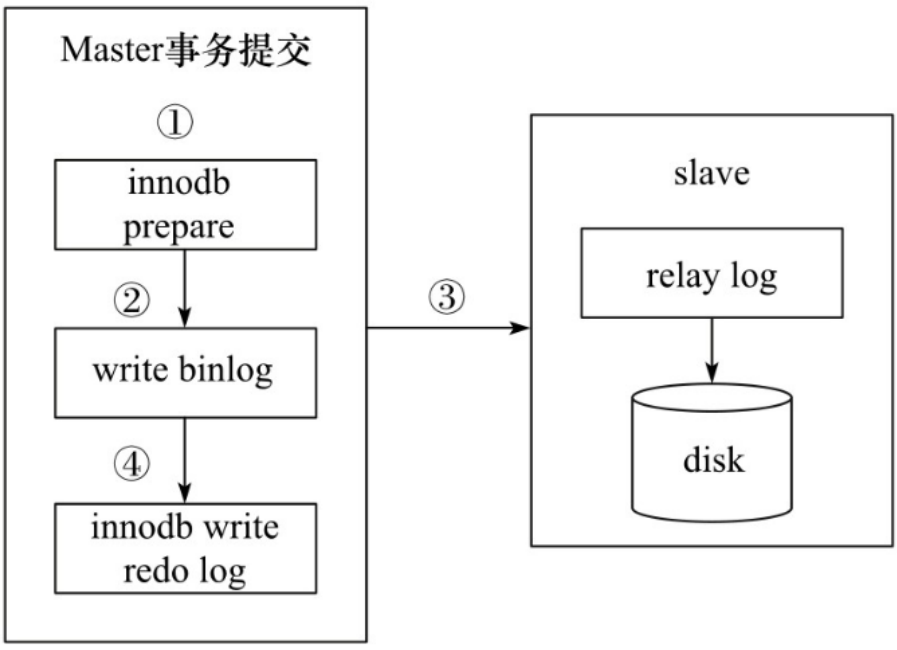


图7-24 MySQL数据库通过内部XA事务保证主从数据一致

日志

二进制日志 binlog

- binlog 记录了所有 DDL（数据定义语句）、DML（数据操纵语句），但是不包括 DQL（数据查询语句）！

  语句以 “事件” 的形式保存，它描述了数据的更改过程，对于数据的完整性 和 安全性 其中重要作用！

- 作用：

  1. 恢复：

     一个数据库全备文件恢复后，用户可以通过 二进制日志 进行 point-in-time 恢复！

     在数据库灾难时的数据恢复！

  2. 复制

     通过复制和执行二进制日志 使一台远程 MySQL 数据库（一般称为：slava 或 standby）与一台 MySQL 数据库（一般称为 ： master 或 primary）进行实时同步！

     主从复制时，从节点将会解析这个日志文件，将数据同步到从节点上！

  3. 审计（audit）

     用户可以使用 二进制日志 中的信息 审计判断 是否有对数据库进行注入的攻击！

- 启动 binlog

  通过配置参数 **log-bin[=name]** 可以启动二进制日志！

  如果不指定 name，则默认为 二进制日志文件名 为主机名，后缀为 二进制日志的序列号，所在路径为数据库所在目录！

- 注意：

  binlog 中记录的是 二进制文件，不能像 慢查询日志那样使用 cat、head、tail 命令查看，而是要通过 mysql 提供的 mysqlbinlog 工具查看！

慢查询日志 slowlog

- 慢查询日志记录了所有执行时间超过参数 **long_query_time**（单位：秒）并且扫描记录数 不低于 **min_examined_row_limit** 的所有 SQL 语句的日志
- 注意：
  - 获得表锁定的时间不算作执行时间！
  - **long_query_time 默认为 10s ，最小为 0，可以精确到 微妙！**
  - **min_examined_row_limit 默认为 0**
- 默认情况下：两类常见语句不会记录到慢查询日志

  1. 管理语句

```
# 如果要监控这类 SQL 语句：则需要使用参数：
--log-slow-admin-statements
```

2. 不使用索引进行查询的语句

```
# 如果要监控这类 SQL 语句：则需要使用参数：
log_queries_not_using_indexes

# 将慢查询日志 输出到 mysql 库的 slow_log 表中，同时数据到文件中！（默认只有文件）
set global log_output='TABLE,FILE';
# show variables like 'log_queries_not_using_indexes'; 查看状态！

# 开启无索引慢查询监控
set global log_queries_not_using_indexes='ON';

# 设置每分钟 允许记录到 slow log 但未使用索引的次数。
# 默认为 0 表示没有限制：这回导致 slow_log 文件的大小不断增大！
set global log_throttle_queries_not_using_indexes=100;
```

- 示例：

```
# 设置慢查询日志时间参数 : 2s
set global long_query_time=2;
set global min_examined_row_limit=5;
# 查看慢查询日志(mysql 库中的 slow_log 表中)
user mysql;
select * from slow_log;
```

redo / undo / binlog

- 详解
- 一条 SQL 语句在 MySQL 中如何执行
- 通俗易懂的讲解一条 SQL 是如何执行的

MySQL

- ★★☆ 水平切分与垂直切分。
- ★★☆ 主从复制原理、作用、实现。