MySQL基础

SQL语句分为哪几类

类型	缩写	示例	作用
数据查询语言	**DQL**	`SELECT`	查询数据
数据操作语言	**DML**	`INSERT`、`UPDATE`、`DELETE`	修改数据
数据定义语言	**DDL**	`CREATE`、`ALTER`、`DROP`	定义表结构
数据控制语言	**DCL**	`GRANT`、`REVOKE`	管理权限
事务控制语言	**TCL**	`COMMIT`、`ROLLBACK`	控制事务

MySQL性能差的原因有哪些

- 全表扫描，
- 未正确使用索引
- 查询语句过于复杂，如多表 JOIN 或嵌套子查询。
- 单表数据量过大，导致查询效率降低。

数据库的三大范式是什么

三大范式的作用是为了减少数据冗余，提高数据完整性。

第一范式

- 确保表的每一列都是不可分割的基本数据单元，比如说用户地址，应该拆分成省、市、区、详细信息等 4 个字段。

第二范式

- 每个非主键字段必须完全依赖于主键，而不能是部分依赖。

第三范式

- 非主键字段必须直接依赖于主键，而不能通过另一个非主键字段传递依赖。

什么场景下不用遵循三大范式

1. 查询频繁、性能要求高的业务场景

- 在 **大数据量、查询频率高** 的情况下，过多的表关联（JOIN）操作会降低查询性能。
- **适当冗余一些字段**，避免频繁 `JOIN`，可以提高查询效率。

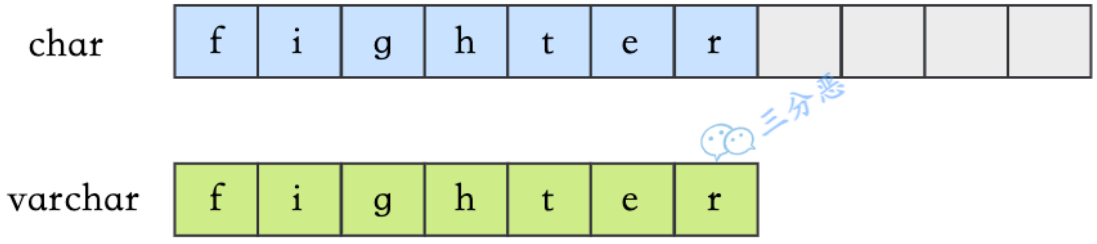
2. 数据写入远少于读取的场景（以读为主）

- 例如 **电商系统的订单记录**，订单创建后不会频繁修改，但需要**快速查询**，可以适当冗余数据。

建表的时候考虑哪些问题

- 在建表的时候，首先可以考虑表是否符合数据库范式，也就是确保字段不可再分，消除非主键依赖，确保字段仅依赖于主键。
- 选择字段类型时，尽量选择合适的数据类型。
- 字符集上，尽量选择 utf8mb4，这样不仅可以支持中文和英文，还可以支持表情符号等。
- 当数据量较大时（比如上千万行数据），需要考虑分表。比如订单表，可以采用水平分表的方式来分散存储压力。

char和varchar的区别



- 因为长度固定，所以char的存取速度要比 varchar 快很多，甚至能快 50%，但正因为其长度固定，所以会占据多余的空间，是空间换时间的做法；

对于长度相对固定的字符串，可以使用 char，对于长度不确定的，使用 varchar 更合适一些。

blob和text有什么区别

blob

- 二进制数据（如图片、音频、视频、PDF）
- 不使用字符集，存储原始字节
- 区分大小写

text

- 文本数据（如文章、日志、JSON）
- 使用字符集，存储文本
- 不区分大小写

DATETIME和TIMESTAMP有什么区别

相同点

- 两个数据类型存储时间的表现格式一致。均为 `YYYY-MM-DD HH:MM:SS`
- 两个数据类型都包含「日期」和「时间」部分。
- 两个数据类型都可以存储微秒的小数秒（秒后 6 位小数秒）

不同点

- **日期范围**：DATETIME 的日期范围是 `1000-01-01 00:00:00.000000` 到 `9999-12-31 23:59:59.999999`；TIMESTAMP 的时间范围是`1970-01-01 00:00:01.000000` UTC `到 ``2038-01-09 03:14:07.999999` UTC
- **存储空间**：DATETIME 的存储空间为 8 字节；TIMESTAMP 的存储空间为 4 字节
- **时区相关**：DATETIME 存储时间与时区无关；TIMESTAMP 存储时间与时区有关，显示的值也依赖于时区
- **默认值**：DATETIME 的默认值为 null；TIMESTAMP 的字段默认不为空(not null)，默认值为当前时间(CURRENT_TIMESTAMP)

in 和 exists 的区别

in

- 子查询先执行，返回完整数据集，主查询匹配
- 子查询结果小（<1000 条）时更快

exisits

- 外层表逐行检查，子查询找到一条匹配就终止
- 外层表大时更快（避免全表扫描）

记录货币用什么字段比较好

货币在数据库中 MySQL 常用 Decimal 和 Numeric 类型表示，这两种类型被 MySQL 实现为同样的类型。他们被用于保存与货币有关的数据。

例如 salary DECIMAL(9,2)，9(precision)代表将被用于存储值的总的小数位数，而 2(scale)代表将被用于存储小数点后的位数。存储在 salary 列中的值的范围是从-9999999.99 到 9999999.99。

DECIMAL 和 NUMERIC 值作为字符串存储，而不是作为二进制浮点数，以便保存那些值的小数精度。

之所以不使用 float 或者 double 的原因：因为 float 和 double 是以二进制存储的，所以有一定的误差。

怎么存储emoji

emoji 表情（😊）是 4 个字节的 UTF-8 字符，所以在 MySQL 中存储 emoji 表情时，需要使用 utf8mb4 字符集。

drop、delete和truncate的区别

区别	delete	truncate	drop
类型	属于 DML	属于 DDL	属于 DDL
回滚	可回滚	不可回滚	不可回滚
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，删除表中的所有数据	从数据库中删除表，所有数据行，索引和权限也会被删除
删除速度	删除速度慢，需要逐行删除	删除速度快	删除速度最快

什么是DDL和DML

DDL（Data Definition Language）

- 用于定义和管理数据库结构（Schema），影响的是数据库对象（表、索引、视图等），DDL 语句会 自动提交，不可回滚。

DML（Data Manipulation Language）

- 用于操作表中的数据（CRUD：增删改查），影响的是数据本身，DML 语句 不会自动提交，可以回滚（ROLLBACK）

UNION 和 UNION ALL的区别

`UNION` 和 `UNION ALL` 都用于合并两个或多个 `SELECT` 语句的查询结果

UNION

- 自动去重，且去重需要排序
- 效率低

UNION ALL

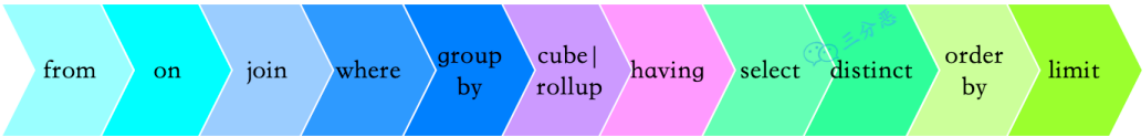
- 不去重，直接合并所有查询结果
- 性能好

如果合并没有刻意要删除重复行，那么就使用 UNION All

count(1)、count(*) 与 count(列名) 的区别

- count(*)包括了所有的列，相当于行数，在统计结果的时候，不会忽略列值为 NULL
- count(1)包括了忽略所有列，用 1 代表代码行，在统计结果的时候，不会忽略列值为 NULL
- count(列名)只包括列名那一列，在统计结果的时候，会忽略列值为空（这里的空不是只空字符串或者 0，而是表示 null）的计数，即某个字段值为 NULL 时，不统计。

SQL查询语句的执行顺序了解吗




```sql
SELECT DISTINCT department, COUNT(*)
FROM employees
JOIN departments ON employees.department_id = departments.id
WHERE salary > 5000
GROUP BY department
HAVING COUNT(*) > 10
ORDER BY COUNT(*) DESC
LIMIT 5;
```

执行顺序：

1. `FROM` → 选择 `employees` 和 `departments` 两个表。
2. `ON` → 连接 `employees.department_id = departments.id`。
3. `JOIN` → 执行 `JOIN` 连接两个表的数据。
4. `WHERE` → 过滤 `salary > 5000` 的行。
5. `GROUP BY` → 按 `department` 分组。
6. `HAVING` → 仅保留 `COUNT(*) > 10` 的分组。
7. `SELECT` → 选择 `department` 和 `COUNT(*)` 作为最终输出列。
8. `DISTINCT` → 确保 `department` 没有重复值（通常 `GROUP BY` 之后已去重）。
9. `ORDER BY` → 按 `COUNT(*)` 降序排序。
10. `LIMIT` → 返回前 5 条记录。

MySQL第3 - 10条记录怎么查?

可以使用limit语句，结合偏移量offset和行数row_count来实现

limit 语句用于限制查询结果的数量，偏移量表示从哪条记录开始，行数表示返回的记录数量。

```sql
SELECT * FROM table_name LIMIT 2, 8;
```

- 2：偏移量，表示跳过前两条记录，从第三条记录开始。
- 8：行数，表示从偏移量开始，返回 8 条记录。

偏移量是从 0 开始的，即第一条记录的偏移量是 0；如果想从第 3 条记录开始，偏移量就应该是 2。

用过哪些MySQL函数

- 字符串函数

    - `CONCAT()`：连接两个或多个字符串。
    - `LENGTH()`：返回字符串的长度。
    - `LOWER()` 和 `UPPER()`：分别将字符串转换为小写或大写。
    - `TRIM()`：去除字符串两侧的空格或其他指定字符。
- 数值函数
    - `ABS()`：返回一个数的绝对值。
    - `CEILING()`：返回大于或等于给定数值的最小整数。
    - `FLOOR()`：返回小于或等于给定数值的最大整数。
    - `ROUND()`：四舍五入到指定的小数位数。
    - `MOD()`：返回除法操作的余数。
- 日期和时间函数
    - `NOW()`：返回当前的日期和时间。
    - `DATE_ADD()` 和 `DATE_SUB()`：在日期上加上或减去指定的时间间隔。
    - `DATEDIFF()`：返回两个日期之间的天数。
- 汇总函数
    - `SUM()`：计算数值列的总和。
    - `AVG()`：计算数值列的平均值。
    - `COUNT()`：计算某列的行数。
    - `MAX()` 和 `MIN()`：分别返回列中的最大值和最小值。
- 逻辑函数
    - `IF()`：如果条件为真，则返回一个值；否则返回另一个值。
- 类型转换函数
    - `CAST()`：将一个值转换为指定的数据类型。
    - `CONVERT()`：类似于`CAST()`，用于类型转换。

说说SQL的隐式数据类型转换

在 SQL 中，当不同数据类型的值进行运算或比较时，会发生隐式数据类型转换。

比如说，当一个整数和一个浮点数相加时，整数会被转换为浮点数，然后再进行相加。

```
SELECT 1 + 1.0; -- 结果为 2.0
```

比如说，当一个字符串和一个整数相加时，字符串会被转换为整数，然后再进行相加。

```
SELECT '1' + 1; -- 结果为 2
```

可以通过显式转换来规避这种情况。

```
SELECT CAST('1' AS SIGNED INTEGER) + 1; -- 结果为 2
```

说说SQL的语法树解析

SQL语法树解析过程通常包含**词法分析**、**语法分析**、**语义分析**

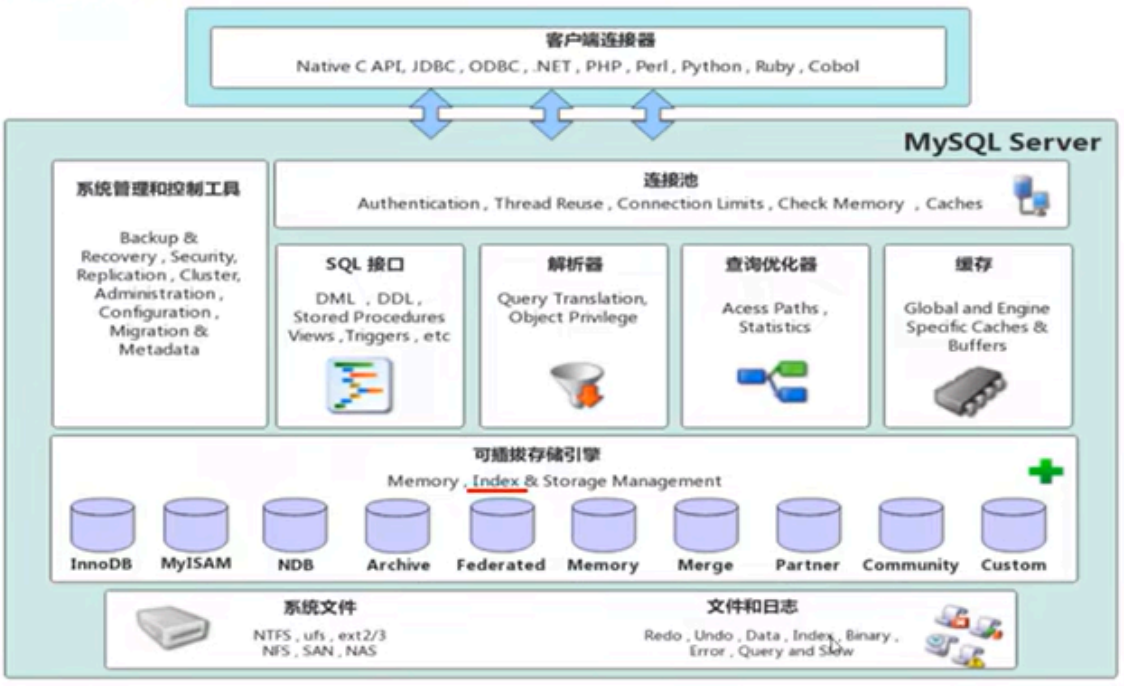
```
SELECT name, age FROM users WHERE age > 18;
```

```
      SELECT
    /   |   \
 COLUMNS FROM WHERE
  /   \    |    \
name  age users  CONDITION
                      / | \
                   age  >  18
```

- **根节点**：通常是 SQL 语句的主要操作，例如 SELECT、INSERT、UPDATE、DELETE 等。
- **内部节点**：表示语句中的操作符、子查询、连接操作等。例如，WHERE 子句、JOIN 操作等。
- **叶子节点**：表示具体的标识符、常量、列名、表名等。例如，users 表、id 列、常量 1 等

数据库架构

MySQL体系结构



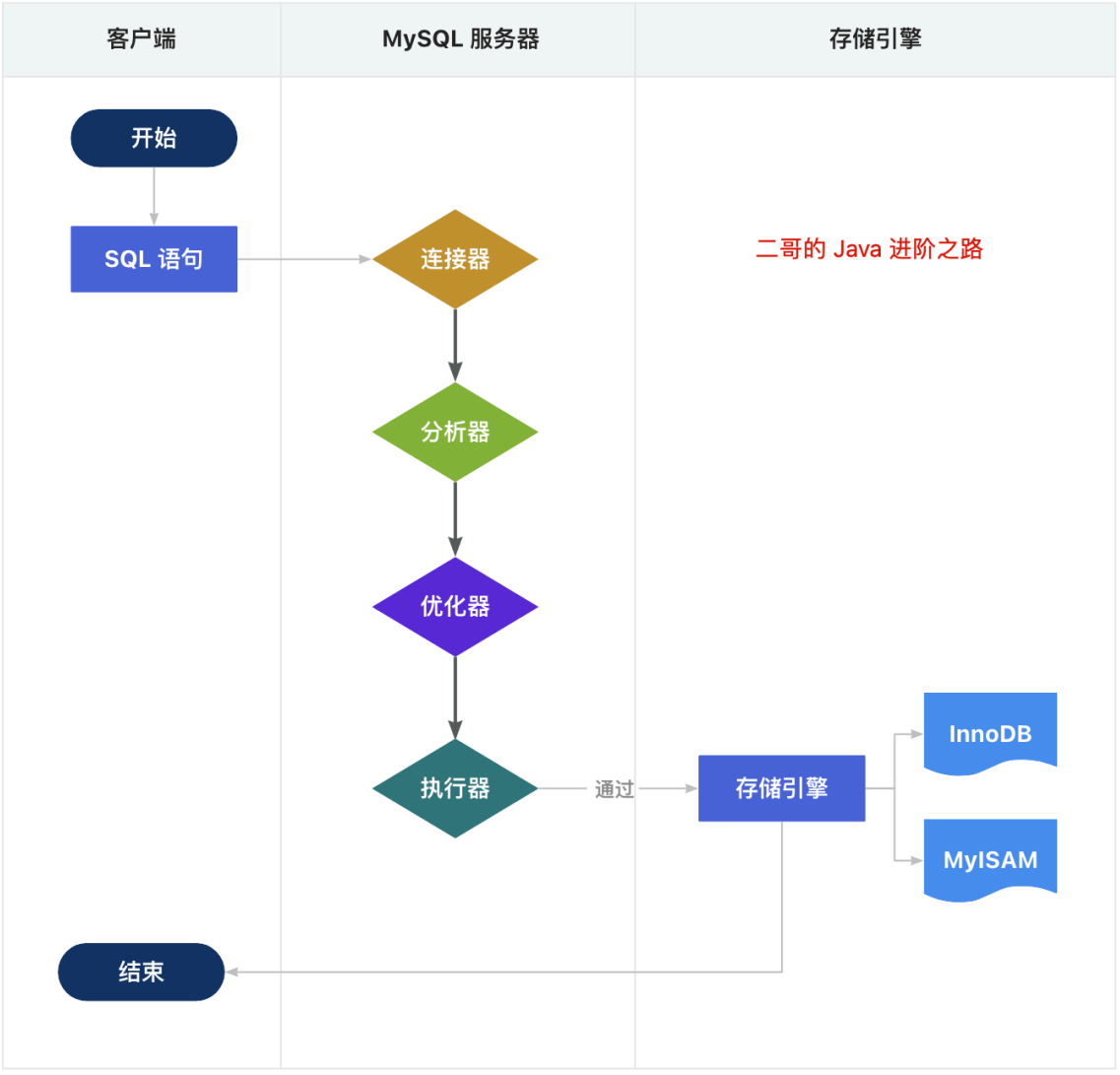
- 连接层
  - 完成一些类似于连接处理、授权认证、及相关的安全方案
- 服务层

    - 完成大多数的核心服务功能，包括查询解析、优化、执行等操作
- 引擎层
    - 存储引擎真正的负责了MySQL中数据的存储和提取
- 存储层
    - 主要是将数据存储在文件系统之上，并完成与存储引擎的交互

binlog写入在哪一层

binlog 在服务层，负责记录 SQL 语句的变化。它记录了所有对数据库进行更改的操作，用于数据恢复、主从复制等。

一条查询语句如何执行

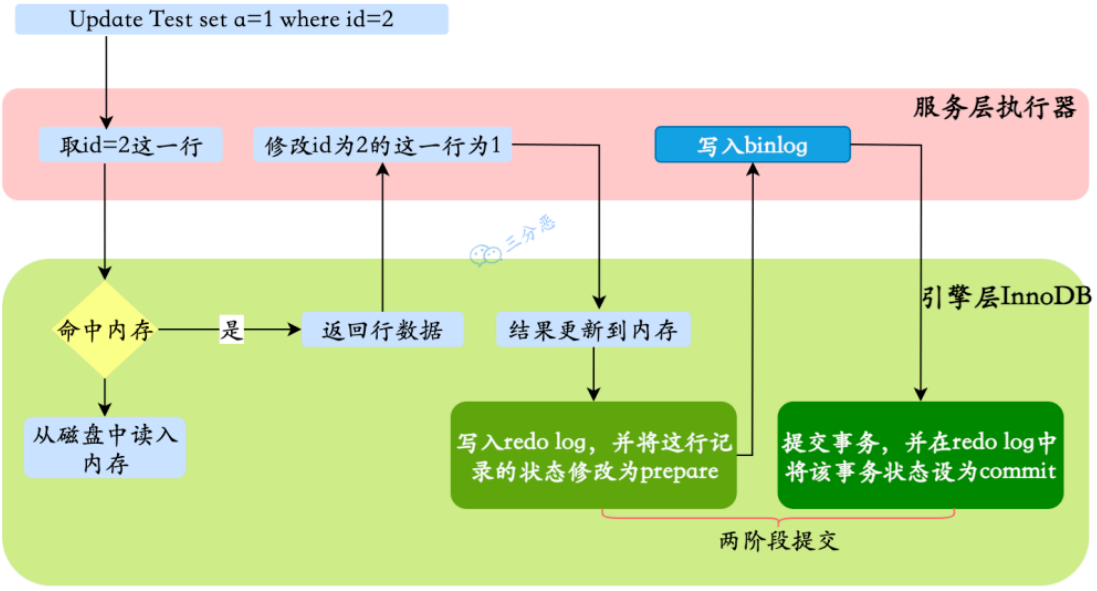


1. 客户端发送 SQL 查询语句到 MySQL 服务器。
2. MySQL 服务器的连接器开始处理这个请求，跟客户端建立连接、获取权限、管理连接。
3. 解析器对 SQL 语句进行解析，检查语句是否符合 SQL 语法规则，确保引用的数据库、表和列都是存在的，并处理 SQL 语句中的名称解析和权限验证。
4. 优化器负责确定 SQL 语句的执行计划，这包括选择使用哪些索引，以及决定表之间的连接顺序等。

5. 执行器会调用存储引擎的 API 来进行数据的读写。
6. MySQL 的存储引擎是插件式的，不同的存储引擎在细节上面有很大不同。例如，InnoDB 是支持事务的，而 MyISAM 是不支持的。之后，会将执行结果返回给客户端
7. 客户端接收到查询结果，完成这次查询请求。

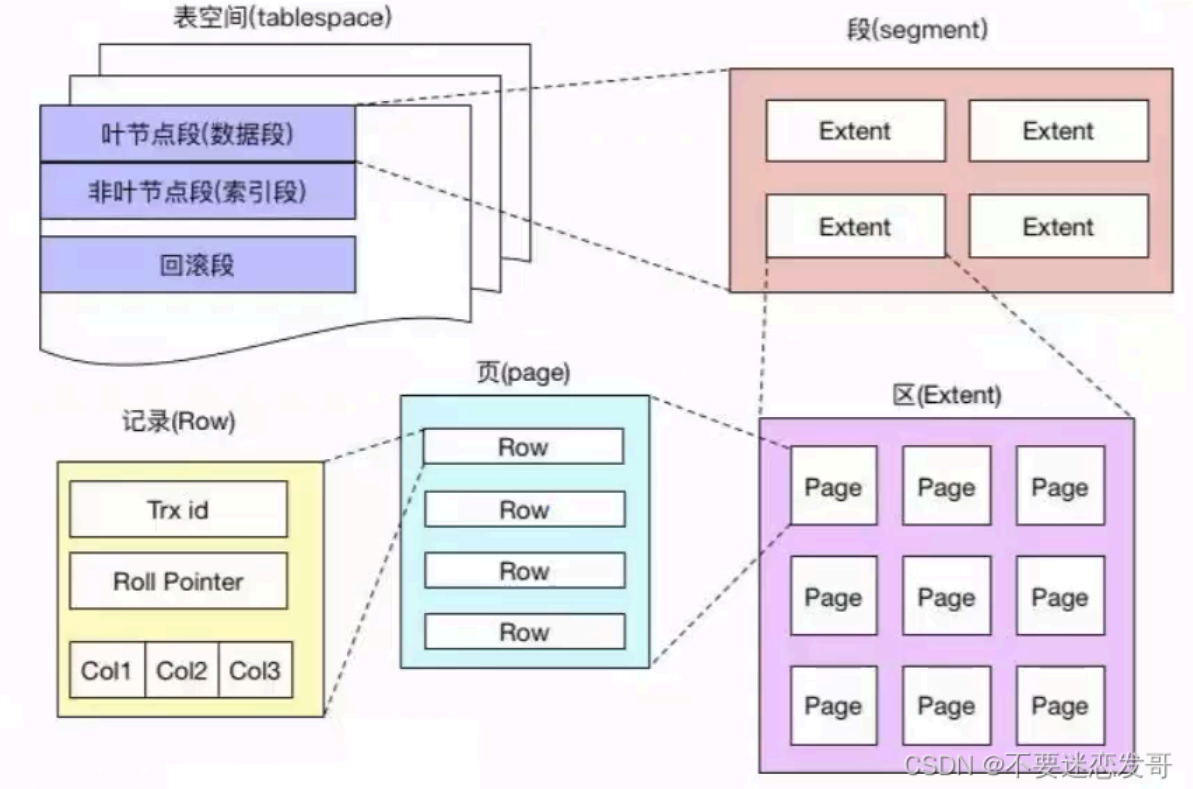
一条更新语句怎么执行

对于更新语句，数据除了要写入表中，还要记录相应的日志。



说说MySQL的数据存储形式

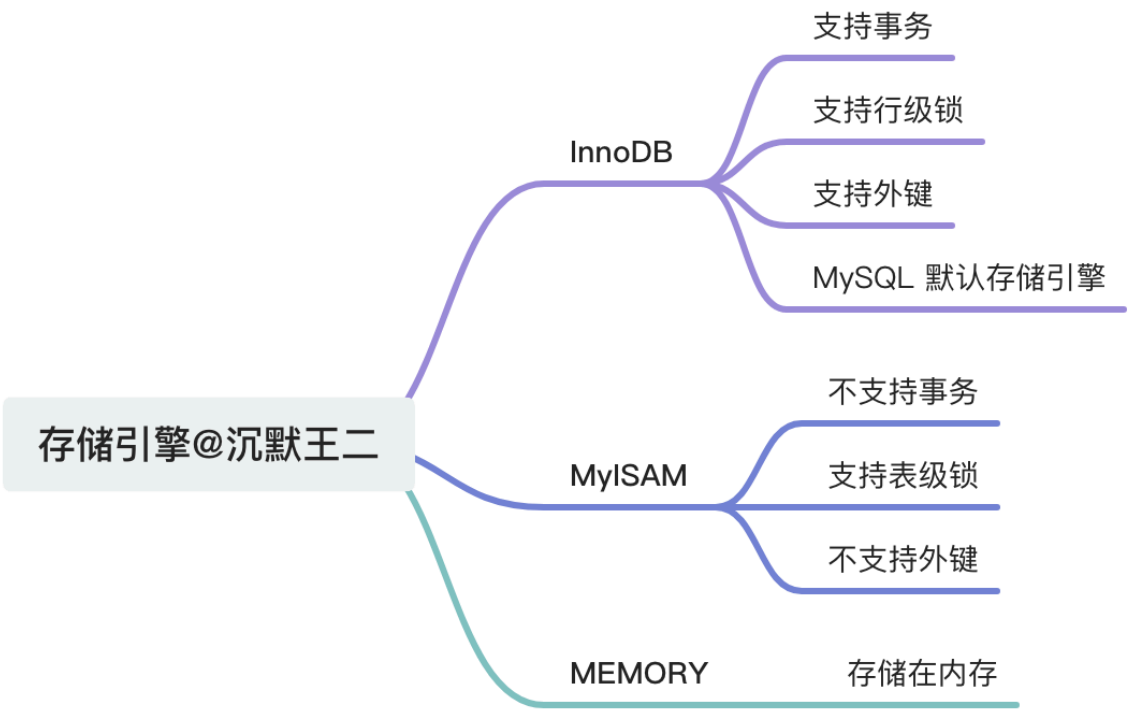
MySQL 是以表的形式存储数据的，而表空间的结构则由段、区、页、行组成。



- 表空间由多个段组成
    - 数据段：存储表的数据
    - 索引段：存储 B+ 树索引
    - 回滚段：存储 事务回滚日志（Undo Log）
- 段是为了隔离数据、索引和回滚
- 区是为了保证同一张表的数据尽可能连续存储，减少磁盘碎片
    - 每个区的大小为1MB，每页为16KB，一个区有64个页
- MySQL读取数据按页为单位，减少I/O次数
- 记录是为了支持行级锁和MVCC

存储引擎

什么是存储引擎



存储引擎就是存储数据、建立索引、更新/查询数据等技术的实现方式

存储引擎是基于表的，不是基于库的，因此存储引擎也可被称为表类型

InnoDB存储引擎是什么

在MySQL5.5之后，InnoDB是默认的MySQL存储引擎

特点：

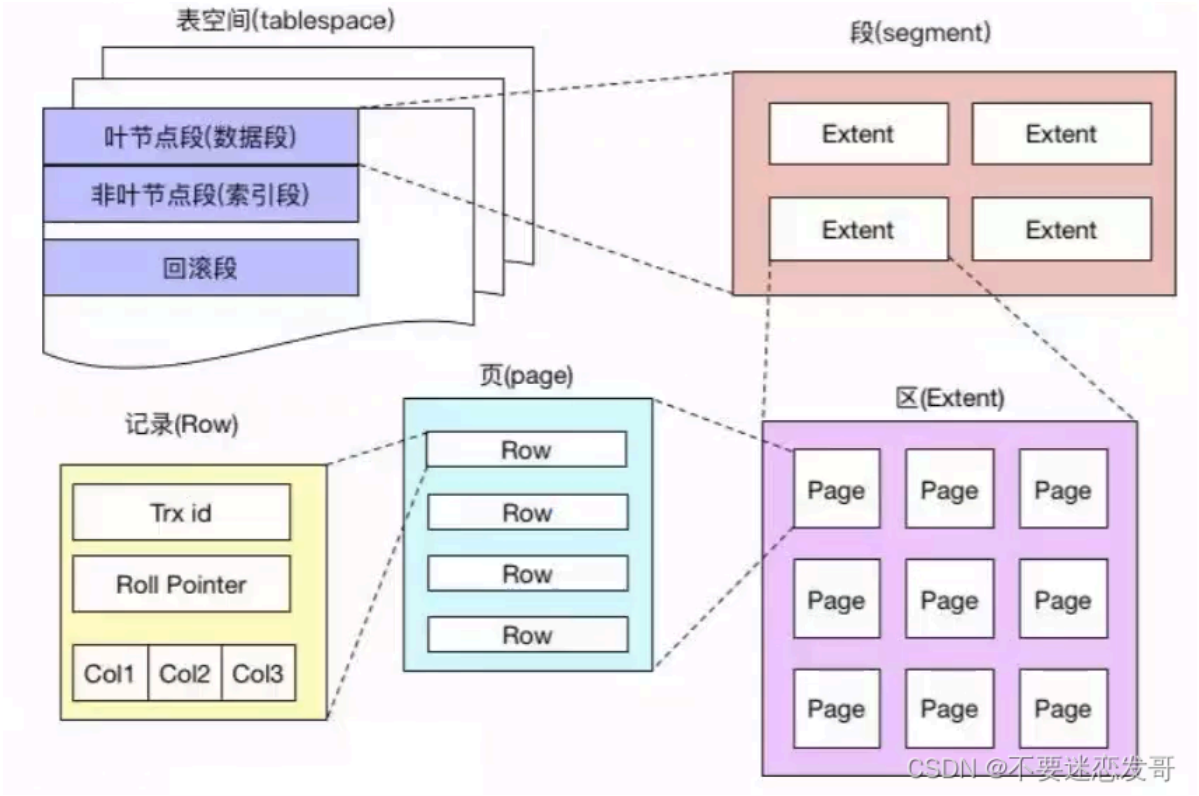
- DML操作遵循ACID模型，支持**事务**
- **行级锁**，提高并发访问性能

- 支持**外键**FOREIGN KEY约束，保证数据的完整性和正确性

文件：

- xxx.ibd：xxx代表的表名，innoDB引擎的每张表都会对应这样一个表空间文件，存储该表的表结构、数据和缩影

逻辑存储结构：



页的大小是16KB，区的大小是1MB

MyISAM存储引擎是什么

MyISAM是MySQL早期的默认存储引擎

特点：

- 不支持事务，不支持外键
- 支持表锁，不支持行锁
- 访问速度快

文件：

- xxx.sdi：存储表结构信息
- xxx.MYD：存储数据
- xxx.MYI：存储索引

如何选择合适的存储引擎

InnoDB：

- 如果应用对事物的完整性有比较高的要求，在并发条件下要求数据的一致性，数据操作除了插入和查询之外，还包含很多的更新、删除操作，那么选择InnoDB

MyISAM:

- 如果应用是以读操作和插入操作为主，并且对事物的完整性、并发性要求不是很高，那么可以选择MyISAM（日志、电商中的足迹、评论数据）
- 被mongoDB替代

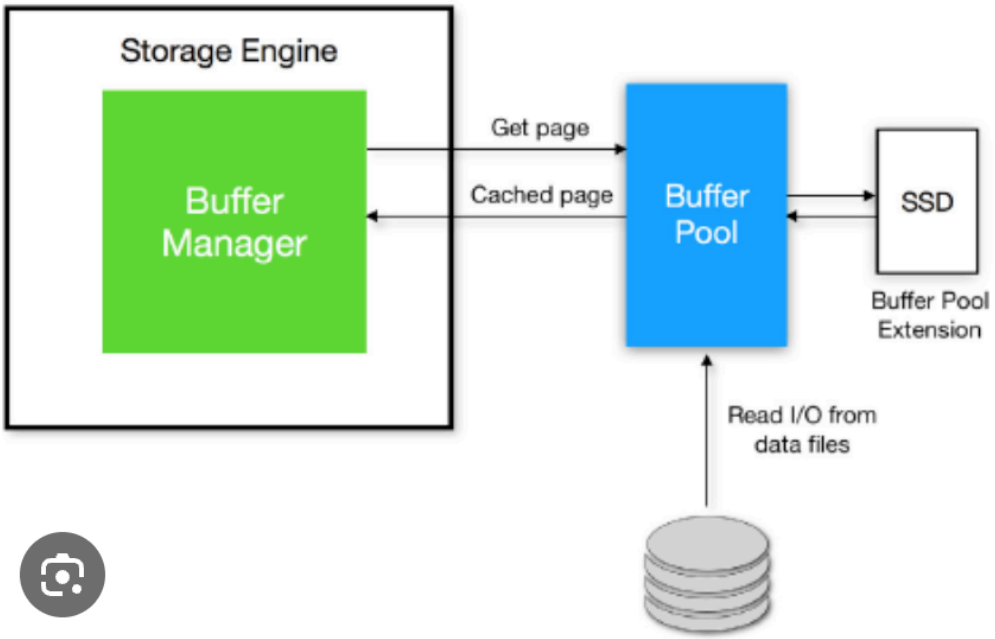
MEMORY:

- 将所有数据保存在内存中，访问速度快，通常用于临时表及缓存
- 被Redis替代

MyISAM和InnoDB有什么区别

- 存储格式不同
- 事务支持
  - MyISAM不支持事务
- 最小锁粒度
  - MyISAM是表级锁，高并发中写操作存在性能瓶颈
  - InnoDB是行级锁，并发写入性能高
- 外键支持
  - MyISAM不支持外键
- 索引类型
  - MyISAM没有聚簇索引，是非聚簇索引，索引和数据分开存储，索引保存的是数据文件的指针
- 主键必需
  - MyISAM表可以没有主键
- 表的具体行数
  - MyISAM表的具体行数存储在表的属性种，查询时直接返回
  - InnoDB表的具体行数需要扫描整个表才能返回

InnoDB的Buffer Pool了解吗

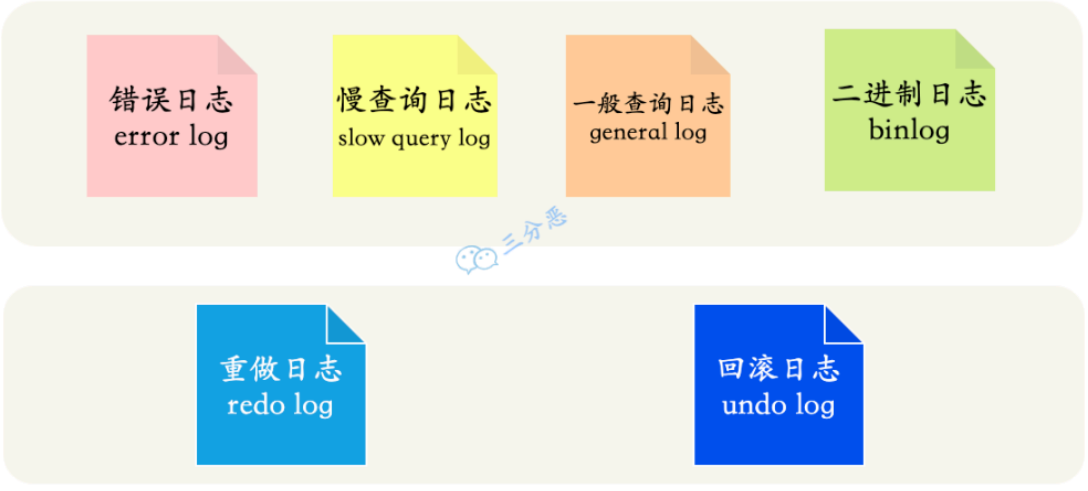


Buffer Pool 是 InnoDB 存储引擎中的一个内存缓冲区，它会将数据以页（page）的单位保存在内存中，当查询请求需要读取数据时，优先从 Buffer Pool 获取数据，避免直接访问磁盘。

- 修改数据时，也会先在缓存页面中修改。当数据页被修改后，会在 Buffer Pool 中变为脏页。
- 脏页不会立刻写回到磁盘。
- InnoDB 会定期将这些脏页刷新到磁盘，保证数据的一致性。通常采用改良的 LRU 算法来管理缓存页，也就是将最近最少使用的数据移出缓存，为新数据腾出空间。

日志

MySQL的日志文件有哪些



1. 错误日志

- 记录MySQL服务器启动、运行或停止时出现的问题

2. 慢查询日志

- 记录执行时间超过 long_query_time 值的所有 SQL 语句。这个时间值是可配置的，默认情况下，慢查询日志功能是关闭的。

3. 一般查询日志

- 记录所有 MySQL 服务器的连接信息及所有的 SQL 语句，不论这些语句是否修改了数据。

4. binlog

- 记录了所有修改数据库状态的 SQL 语句，以及每个语句的执行时间，如 INSERT、UPDATE、DELETE 等，但不包括 SELECT 和 SHOW 这类的操作。

5. redo log

- 记录了对于 InnoDB 表的每个写操作，不是 SQL 级别的，而是物理级别的，主要用于崩溃恢复。

6. undo log

- 记录数据被修改前的值，用于事务的回滚

binlog是干什么的

binlog 是一种物理日志，会在磁盘上记录下数据库的所有修改操作，以便进行数据恢复和主从复制。

- 当发生数据丢失时，binlog 可以将数据库恢复到特定的时间点。
- 主服务器（master）上的二进制日志可以被从服务器（slave）读取，从而实现数据同步。

binlog三种模式:

- statement模式
  - 记录的是数据库执行的原生sql语句
  - 二进制日志文件最小，性能最高
  - 容易出现主从不一致的问题（uuid(), now()等函数）
- row模式
  - 记录的是数据行的更改情况，即数据行在更改前、更改后的变化情况
  - 二进制日志文件最大，性能较低
- mixed模式
  - statement + row

有了binlog为什么还要undolog redolog

- binlog 是 MySQL Server 层提供的日志，独立于存储引擎。
- redo log 主要用于数据持久化和崩溃恢复。redo log 是 InnoDB 存储引擎特有的日志，用于记录数据的物理修改，确保数据库在崩溃或异常宕机后能够恢复到一致状态。
- undo log 主要用于支持事务回滚和多版本并发控制（MVCC）。undo log 是 InnoDB 存储引擎提供的逻辑日志，用于记录数据的逻辑操作，如删除、更新前的数据快照。

什么是binlog和redolog的两阶段提交

保证主从一致性

```
1. redo log: "UPDATE users SET age = 30 WHERE id = 1; (prepare)"
2. binlog: "UPDATE users SET age = 30 WHERE id = 1;"
3. redo log: "UPDATE users SET age = 30 WHERE id = 1; (commit)"
```

1. 先写 `redo log`（prepare），保证数据可以恢复。
2. 再写 `binlog`，保证主从复制数据一致。
3. 最后提交 `redo log`（commit），确保事务最终持久化。

这样，即使最后redolog提交失败了，数据库也能根据binlog是否提交重新提交redolog

`binlog` 是主从复制 & 增量恢复的依据，只要 `binlog` 里有事务提交，就说明事务应该是成功的，所以 MySQL 可以补交 `redo log`。

什么是WAL

WAL（Write-Ahead Logging）的思想是**先写日志，再写数据**，即在对数据进行任何修改之前，必须先将修改的日志记录（redo log）持久化到磁盘。

通过先写日志，确保系统在发生故障时可以通过重做日志恢复数据。

SQL优化

慢SQL怎么定位

通过启用慢查询日志，记录那些超过指定执行时间的查询

```
SET GLOBAL slow_query_log = ON;
```

设置慢查询阈值

```
SET GLOBAL long_query_time = 1;  # 记录执行时间超过 1 秒的 SQL
```

使用 EXPLAIN 查看查询执行计划，判断查询是否使用了索引，是否有全表扫描等

```
EXPLAIN SELECT * FROM your_table WHERE conditions;
```

最后，根据分析结果，通过添加或优化索引、调整查询语句或者增加内存缓冲区来优化 SQL

show profiles是什么

show profiles能够在做SQL优化时帮助我们了解时间都耗费到哪里去了

启动show profiles

```
SET profiling = 1;
```

查看指定query_id的SQL语句各个阶段的耗时情况

```
show profile for query query_id
```

有哪些优化SQL的方式

1. 避免不必要的列

- 尽量避免使用 `select *`，只查询需要的列，减少数据传输量

2. 分页优化

当数据量巨大时，传统的`LIMIT`和`OFFSET`可能会导致性能问题，因为数据库需要扫描`OFFSET + LIMIT`数量的行。

- 延迟关联

  ```
  SELECT e.id, e.name, d.details
  FROM employees e
  JOIN department d ON e.department_id = d.id
  ORDER BY e.id
  LIMIT 1000, 20;
  ```

  改为

  ```
  SELECT e.id, e.name, d.details
  FROM (
      SELECT id
      FROM employees
      ORDER BY id
      LIMIT 1000, 20
  ) AS sub
  JOIN employees e ON sub.id = e.id
  JOIN department d ON e.department_id = d.id;
  ```

- 书签

  书签方法通过记住上一次查询返回的最后一行的某个值，然后下一次查询从这个值开始，避免了扫描大量不需要的行。

  ```
  SELECT id, name
  FROM users
  WHERE id > last_max_id  -- 假设last_max_id是上一页最后一行的ID
  ORDER BY id
  LIMIT 20;
  ```

3. 索引优化

- 利用覆盖索引

  使用非主键索引查询数据时需要回表，但如果索引的叶节点中已经包含要查询的字段，那就不会再回表查询了，这就叫覆盖索引。

  ```
  1 select name from test where city='上海'
  ```

  改为

  ```
  1 alter table test add index index1(city,name);
  ```

- 避免使用 != 或 <>操作符

  `!=` 或者 `<>` 操作符会导致 MySQL 无法使用索引，从而导致全表扫描。

  例如，可以把`column<>'aaa'`，改成`column>'aaa' or column<'aaa'`，就可以使用索引了。

- 避免列上使用函数，会导致索引失效
- 最左前缀原则，应该按照查询中的字段顺序来创建索引。

JOIN优化

- 优化子查询

  子查询，特别是在 select 列表和 where 子句中的子查询，往往会导致性能问题，因为它们可能会为每一行外层查询执行一次子查询。

  ```
  1 select name from A where id in (select id from B);
  ```

  改为

  ```
  1 select A.name from A join B on A.id=B.id;
  ```

- 小表驱动大表
  - 假设一张表10条，另一张表100万条，join
    - 小表驱动大表：

      对每一行小表的数据，在大表中查找匹配记录（假设有索引）

      总查找次数 ≈ 10 × log(100万)
    - 大表驱动小表

      总查找次数 ≈ 100万 × log(10) ≈ 100万

如何进行排序优化

- 如果 name 字段上有索引，那么 MySQL 可以直接利用索引的有序性，避免排序操作

怎么查看执行计划explain

在 select 语句前加上 `explain` 关键字就可以了

```sql
explain select * from students where id =9
```

type列：表示 MySQL 在表中找到所需行的方式，性能从最优到最差分别为：system > const > eq_ref > ref > range > index > ALL。

- system：表只有一行，一般是系统表，往往不需要进行磁盘 IO，速度非常快
- const：表中只有一行匹配，或通过主键或唯一索引获取单行记录。通常用于使用主键或唯一索引的精确匹配查询，性能非常高。

  ```sql
  SELECT * FROM users WHERE id = 5;
  ```

- eq_ref： 唯一索引（PRIMARY KEY 或 UNIQUE KEY）的等值查询，被驱动表的每一行最多匹配一条记录，通常出现在 JOIN 查询 中

  ```sql
  SELECT *
  FROM orders o
  JOIN users u ON o.user_id = u.id;  -- u.id是PRIMARY KEY
  ```

- ref：非唯一索引，意味着 查询可能返回多行，但仍然是基于索引查找。

  ```sql
  SELECT * FROM products WHERE category = 'Electronics';
  ```

- range：只检索给定范围的行，使用索引来检索。在`where`语句中使用`bettween...and`、`<`、`>`、`<=`、`in`等条件查询`type`都是`range`。
- index：全索引扫描，即扫描整个索引而不访问数据行。
- ALL：全表扫描，效率最低。

possible key列：可能会用到的索引

key列：实际使用的索引

key_len列：MySQL 决定使用的索引长度（以字节为单位）。当表有多个索引可用时，key_len 字段可以帮助识别哪个索引最有效。通常情况下，更短的 key_len 意味着数据库在比较键值时需要处理更少的数据。

索引

什么是索引

索引（index）是帮助MySQL高效获取数据的数据结构（有序）

索引是在引擎层实现的

优点：

- 提高查询效率
- 提高排序效率

缺点：

- 索引也是要占空间的

- 降低了更新表的速度

为什么索引能加快查询

数据库的数据存储在磁盘中，

- 如果没有索引，数据库需要从磁盘中读取整个表，所有数据页都需要被加载到内存，导致大量的磁盘I/O
- 有索引，先访问索引，定位存储位置，直接读取相关数据页

索引的数据结构

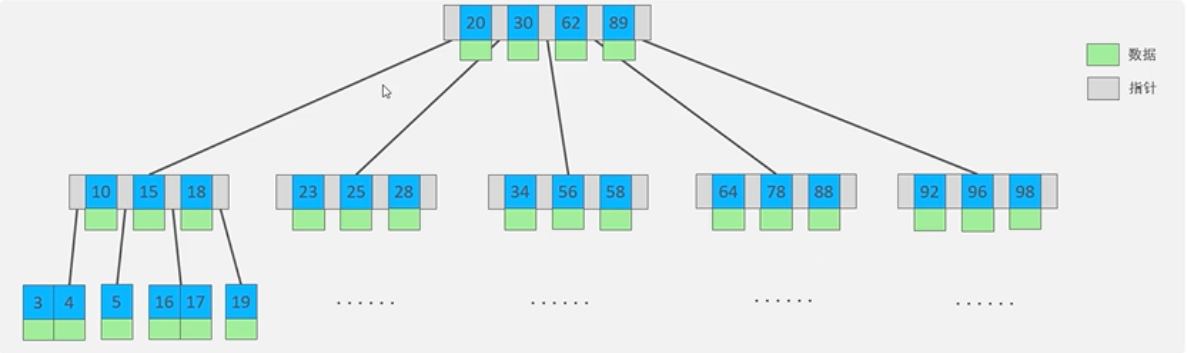
二叉搜索树：

- 顺序插入时，会形成一个链表，查询性能大大降低。
- 大数据量下，层级较深，相当于全表扫描
- 二叉树最多两个节点，层数较深

红黑树

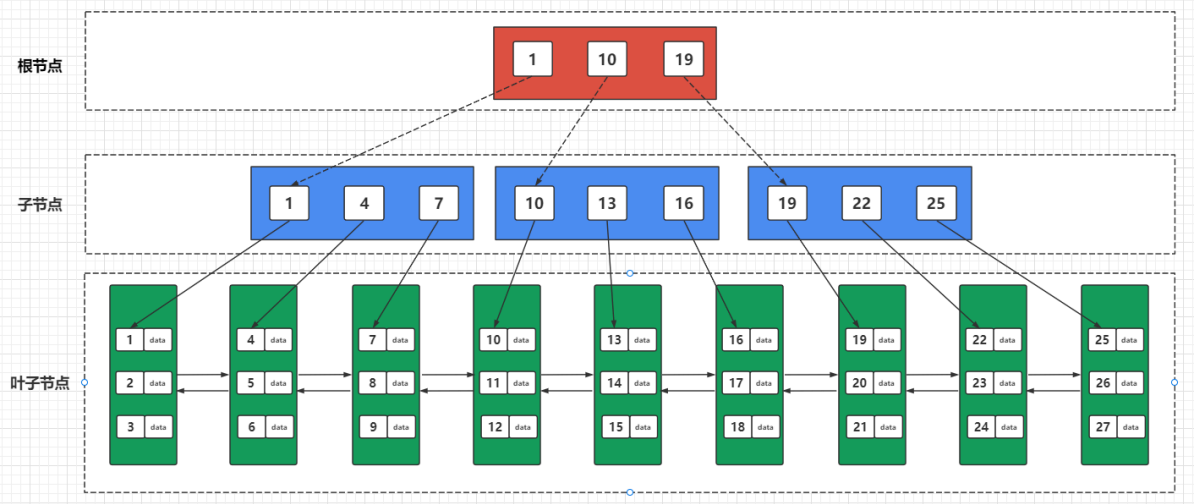
- 二叉树，大数据量下，层级较深，检索速度慢

B树



- n个key会有n+1个指针
- 每个key下面都会有数据

B+树



- 所有key都会出现在叶子节点
- 叶子节点形成一个单向链表

为什么用B+树作为索引而不用B树

- B+树的内部节点仅存储索引，不存储数据，使得一个页能容纳更多索引值，从而降低树的高度，减少了磁盘IO的次数
- B+树的叶子节点是双向链表，适合范围查询

为什么用B+树不用跳表

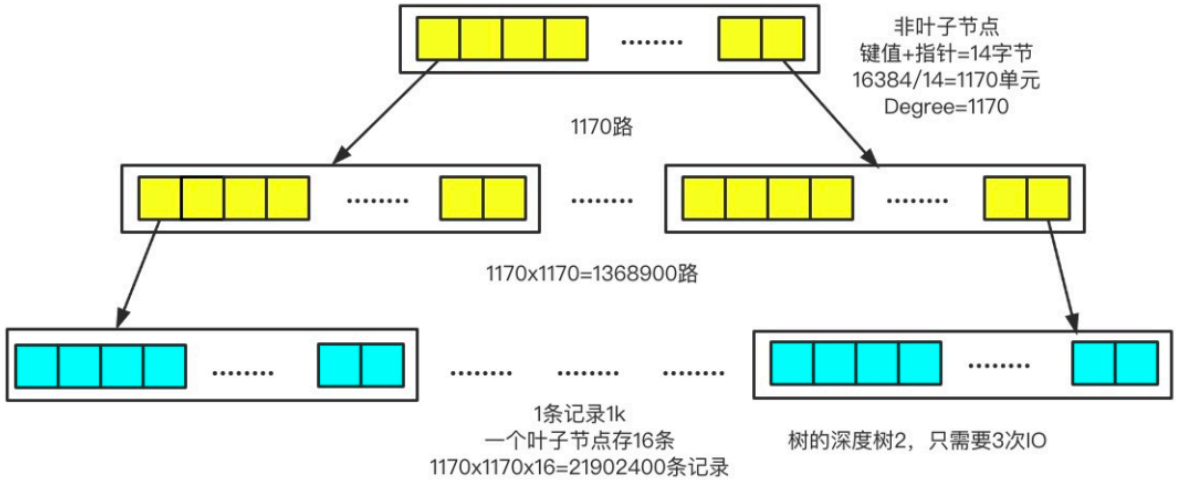
为什么MongoDB索引使用B树，而MySQL使用B+树

那么在查找单条数据时，B 树的查询效率可能会更高，因为每个节点都存储数据，所以最好情况就是 O(1)。

但由于 B 树的节点之间没有指针链接，所以并不适合做范围查询，因为范围查询需要遍历多个节点。

- MySQL 属于关系型数据库，所以范围查询会比较多，所以采用了 B+树；
- MongoDB 属于非关系型数据库，在大多数情况下，只需要查询单条数据，所以 MongoDB 选择了 B 树。

一棵B+树能存储多少条数据



- 假设主键ID是bigint类型，8字节
- 指针大小在InnoDB中设置为6字节

所以非叶子节点(一页16k)可以存储 16384/14=1170 个这样的单元(键值+指针)。

- 每条记录1kB，一页可以放16条记录

一个指针指向一个存放记录的页，一页可以放 16 条数据，树深度为 2 的时候，可以存放 1170*16=**18720** 条数据。

同理，树深度为 3 的时候，可以存储的数据为 1170*1170*16=**21902400**条记录。

索引分为哪几类

功能分类

1. 主键索引

- PRIMARY修饰，默认自动创建

2. 唯一索引

- UNIQUE修饰，默认自动创建

3. 普通索引

4. 全文索引

- FULLTEXT修饰，默认自动创建

在InnoDB存储引擎中，根据索引的存储形式，又可以分为以下两种：

1.聚簇索引

- 索引结构的叶子节点保存了行数据
- 必须有，而且只能有一个

2. 非聚簇索引

- 索引结构的叶子节点关联的是对应的主键

回表查询：先根据非聚簇索引查找到主键的值，再根据聚簇索引拿到这行的值

什么是最左前缀法则

如果索引了多列（联合索引），要遵守最左前缀法则。

最左前缀法则指的是查询从索引的最左列开始，并且不跳过索引中的列。

如果跳跃某一列，**索引将部分失效（后面的字段索引失效）**

范围查询(>，<)右侧的列索引失效，尽量使用>= 和 <=

- 为什么范围查询会使右侧索引失效
  - 假设索引是(age, salary)
  - 因为 `salary` 只有在 `age` 固定时是有序的，而 `age > 25` 时，我们的查询数据可能来自不同的 `age` 段，`salary` 在不同 `age` 组之间没有顺序！

只要索引的最左列存在就好，顺序无所谓，如

- 索引（profession, age, status）

```
select * from tb_user where age=31 and status='0' and profession='软件工程'
```

这条能用到所有索引

索引失效还有什么情况

1. 违反最左前缀法则

2. 在索引列上进行运算或函数

3. 字符串不加引号

4. 模糊匹配

- 尾部模糊匹配，索引不失效（like '软件%'）
- 头部模糊匹配，索引失效（like '%软件'）

5. or分隔开的条件必须都有索引，否则索引失效

6. MySQL评估使用索引比全表慢，不走索引

索引不适合哪些场景

- 数据表较小，直接全表扫描
- 频繁更新的列
- 低区分度的列上的索引
    - 区分度 = 字段的唯一值数量 / 字段的总记录数
- 使用函数、运算符的字段

什么是覆盖索引

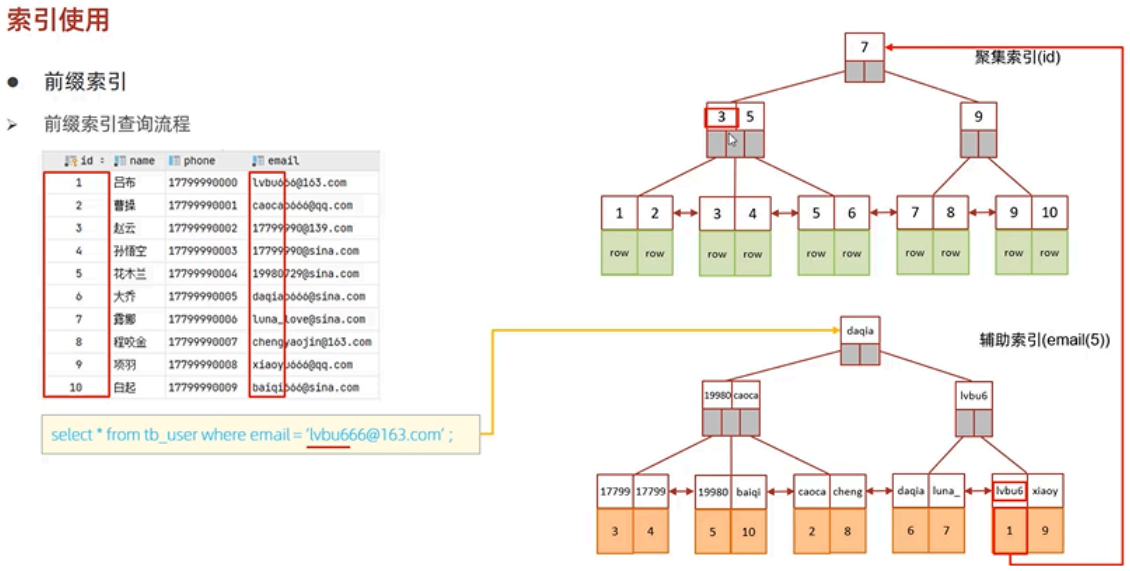
将查询的字段都放在索引中，查询使用了索引，并且需要返回的列，在该索引中已经全部能够找到，不需要回表查询（避免select *）

```
CREATE INDEX idx_xxx ON table_name(colunm(n))
```

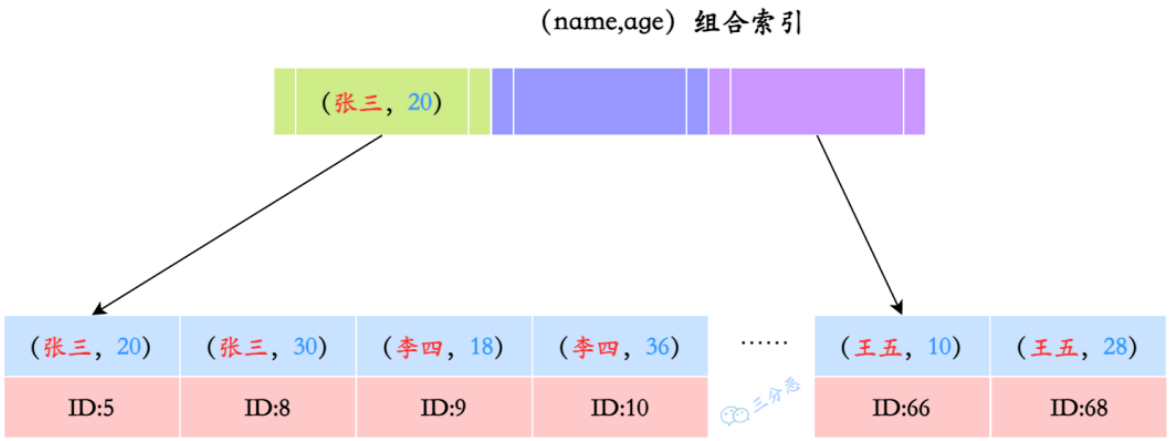
什么是前缀索引

当字段类型为字符串（varchar、text等）时，有时需要索引很长的字符串，这会让索引变得很大，查询时，浪费大量的磁盘IO，影响查询效率。

此时可以只将字符串的一部分前缀建立索引，这样可以大大节约索引空间，从而提高索引效率。



联合索引的数据结构



name 是有序的，age 是无序的。当 name 相等的时候，age 才有序。

当我们使用 `where name= '张三' and age = '20'` 去查询的时候， B+ 树会优先比较 name 来确定下一步应该搜索的方向，往左还是往右。

如果 name 相同的时候再比较 age。

但如果查询条件没有 name，就不知道应该怎么查了，因为 name 是 B+树中的前置条件，没有 name，索引就派不上用场了。

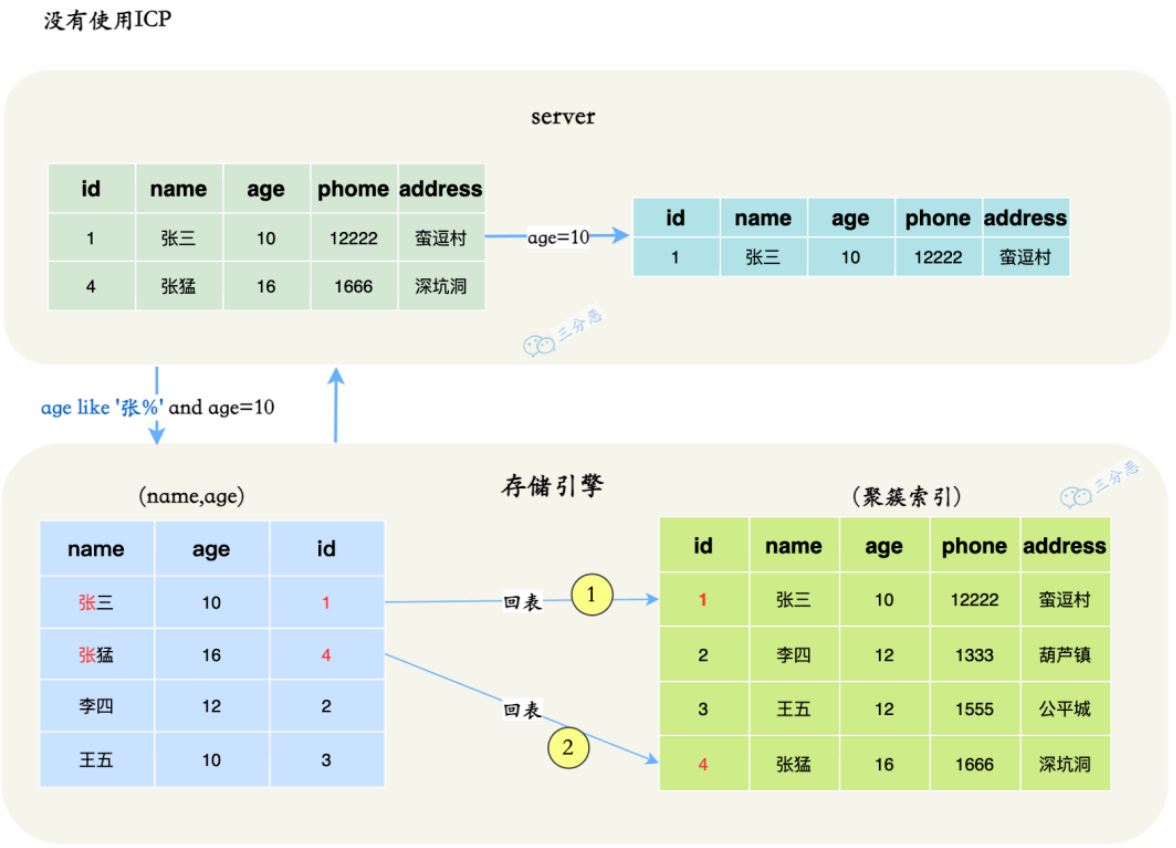
索引优化的思路

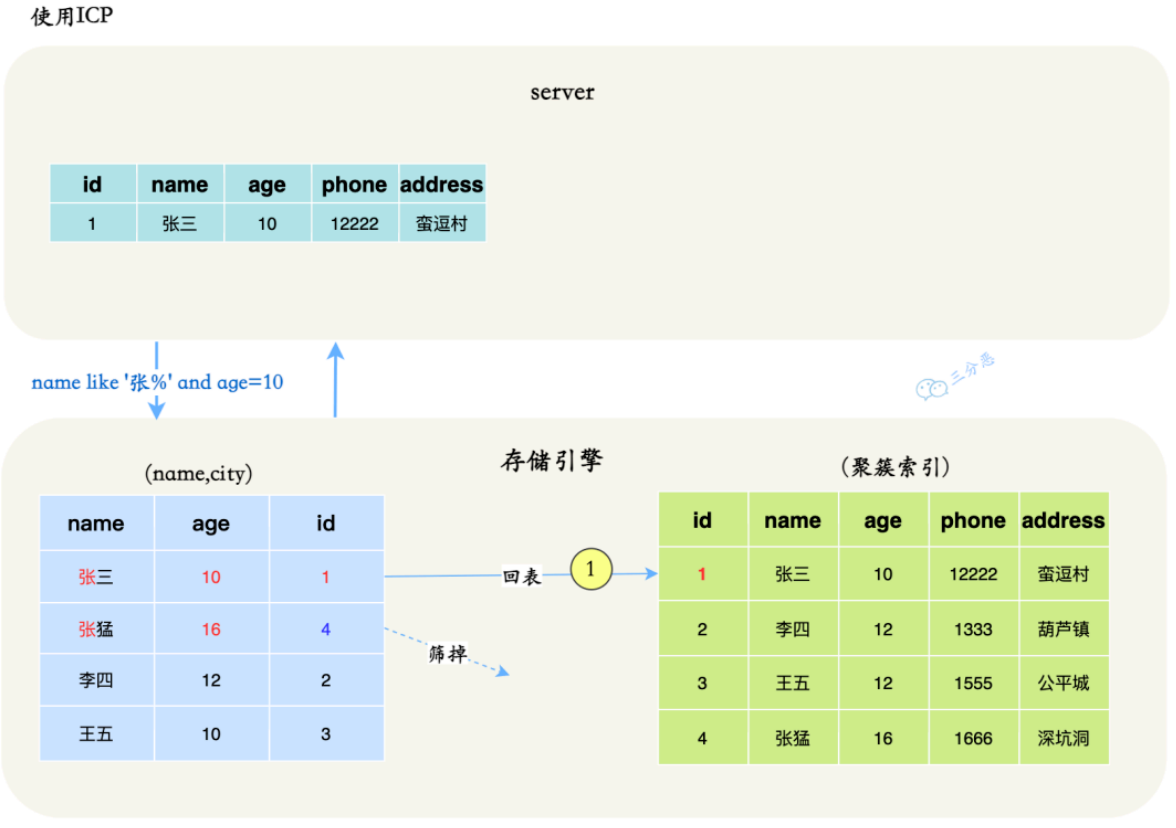
- 选择合适的索引类型
  - 等值查询或者范围查询，使用B+树
  - 文本数据的全文搜索，选择全文索引
- 创建适当的索引，尽量使用覆盖索引

什么是索引下推优化

传统索引扫描时，MySQL 只利用索引做**主键查找**，然后再回表检查 `WHERE` **条件。**

索引下推优化允许 MySQL 在**索引遍历阶段提前过滤不符合** `WHERE` **条件的数据，减少回表次数，提高查询速度！**





锁

MySQL中的锁分为几类

1. 全局锁：锁定数据库中的所有表，加锁后整个实例处于只读状态

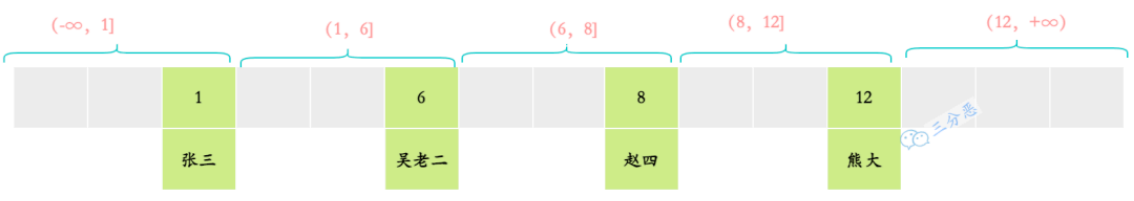
- 单行的使用场景是做全库的逻辑备份，对所有表进行锁定，保证一致性

2. 表级锁：每次操作锁住整张表

- **表锁**：
  - **表共享读锁**：不阻塞其它客户端的读操作，阻塞其它客户端的写操作，自己只能读不能写
  - **表排他写锁**：阻塞其它客户端的读和写操作，自己可以进行读写
- **元数据锁**：
  - 无需显式使用，在访问一张表的时候会自动加上
  - 作用是维护表元数据的一致性
  - 在对一张表进行增删改查时，加元数据读锁；当对表结构进行变更时，加元数据写锁
- **意向锁**：
  - 为了避免DML在执行时，加的行锁与表锁的冲突，在InnoDB中引入了意向锁，使得表锁不用检查每行数据是否加锁，使用意向锁来减少表锁的检查
  - **意向共享锁**：由语句select ... lock in share mode添加
    - 与表共享读锁兼容，与表排他写锁互斥
  - **意向排他锁**：由insert、update、delete、select ... for update添加
    - 与表共享读锁和表排他写锁都互斥
  - 意向锁之间不会互斥

3. 行级锁：每次操作锁住对应的行数据

- 锁定粒度最小，发生锁冲突的概率最低，并发度最高
- InnoDB存储引擎中，行锁不是直接锁住物理行，而是锁住索引项。换句话说，InnoDB 依赖索引来加行锁，没有索引的情况下，可能会升级为表锁。
- **行锁**：锁定单个行记录的锁，防止其它事务对此进行update和delete
  - 共享锁
  - 排他锁
- **间隙锁**：锁定索引记录间隙（不含该记录），防止其它事务在这个间隙进行insert，产生幻读
- **临键锁**：行锁和间隙锁结合，同时锁住数据，并锁住数据前面的间隙



全局锁了解吗

全局锁就是对整个数据库实例进行加锁，在 MySQL 中，可以使用 `FLUSH TABLES WITH READ LOCK` 命令来获取全局读锁。

全局锁的作用是保证在备份数据库时，数据不会发生变化【数据更新语句（增删改）、数据定义语句（建表、修改表结构等）和更新事务的提交语句】。当我们需要备份数据库时，可以先获取全局读锁，然后再执行备份操作。

表锁了解吗

表锁就是锁住整个表。在 MySQL 中，可以使用 `LOCK TABLES` 命令来锁定表。

- 在进行大规模的数据导入、导出或删除操作时，为了防止其他事务对数据进行并发操作，可以使用表锁。
- 或者在进行表结构变更（如添加列、修改列类型）时，为了确保变更期间没有其他事务访问或修改该表，可以使用表锁。

行锁了解吗

在 MySQL 中，InnoDB 存储引擎支持行级锁。通过 `SELECT ... FOR UPDATE` 可以加排他锁，通过 `LOCK IN SHARE MODE` 可以加共享锁。

如果查询条件未使用索引，SELECT FOR UPDATE 可能锁定整个表或大量的行，因为查询需要执行全表扫描。

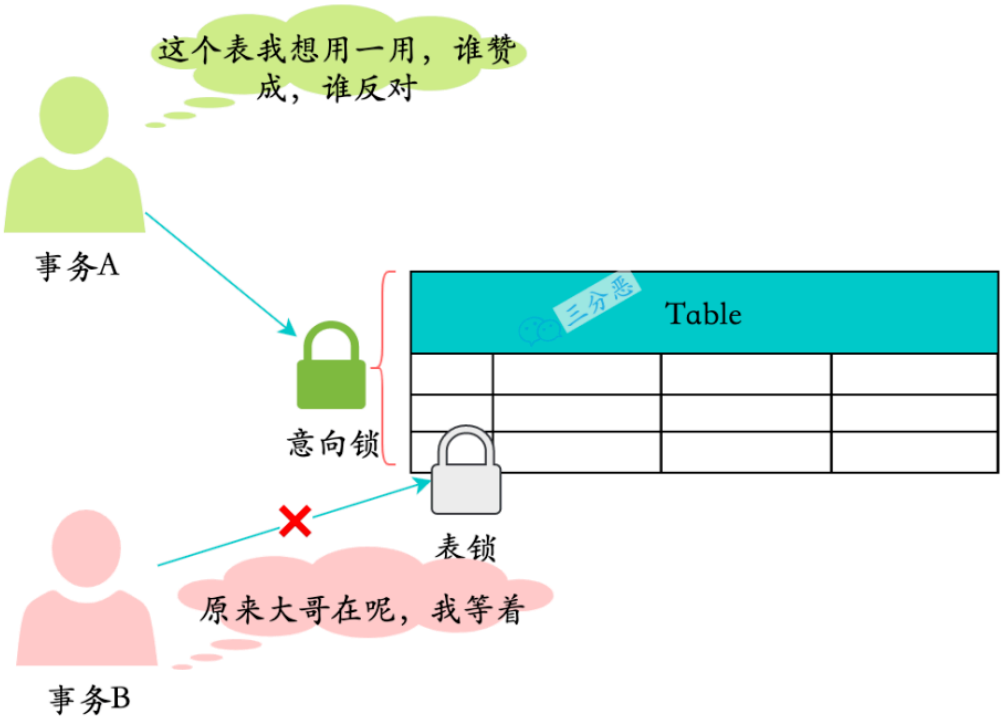
- 在高并发环境中，行级锁能够提高系统的并发性能，因为锁定的粒度较小，只会锁住特定的行，不会影响其他行的操作

间隙锁了解吗

MySQL 默认行锁类型就是临键锁。当使用唯一性索引，等值查询匹配到一条记录的时候，临键锁会退化成记录锁；没有匹配到任何记录的时候，退化成间隙锁。

1. 索引上的等值查询(唯一索引)，给不存在的记录加锁时，优化为间隙锁 。
2. 索引上的等值查询(普通索引)，向右遍历时最后一个值不满足查询需求时，next-key lock 退化为间隙锁。
3. 索引上的范围查询(唯一索引)--会访问到不满足条件的第一个值为止。

意向锁是什么知道吗



意向锁是一个表级锁，意向锁的出现是为了支持 InnoDB 的多粒度锁，它解决的是表锁和行锁共存的问题。

当我们需要给一个表加表锁的时候，我们需要根据去判断表中有没有数据行被锁定，以确定是否能加成功。

假如没有意向锁，那么我们就得遍历表中所有数据行来判断有没有行锁；

有了意向锁这个表级锁之后，则我们直接判断一次就知道表中是否有数据行被锁定了。

事务

什么是事务

事务是一组操作的集合，是一个不可分割的工作单位，事务会把所有操作作为一个整体一起向系统提交或撤销操作请求，**要么同时成功，要么同时失败**

什么是事务的四大特性

1. 原子性（Atomicity）

- 事务是不可分割的最小操作单元，要么全部成功，要么全部失败

2. 一致性（Consistency）

- 事务完成时，必须使所有的数据保持一致状态

3. 隔离性（Isolation）

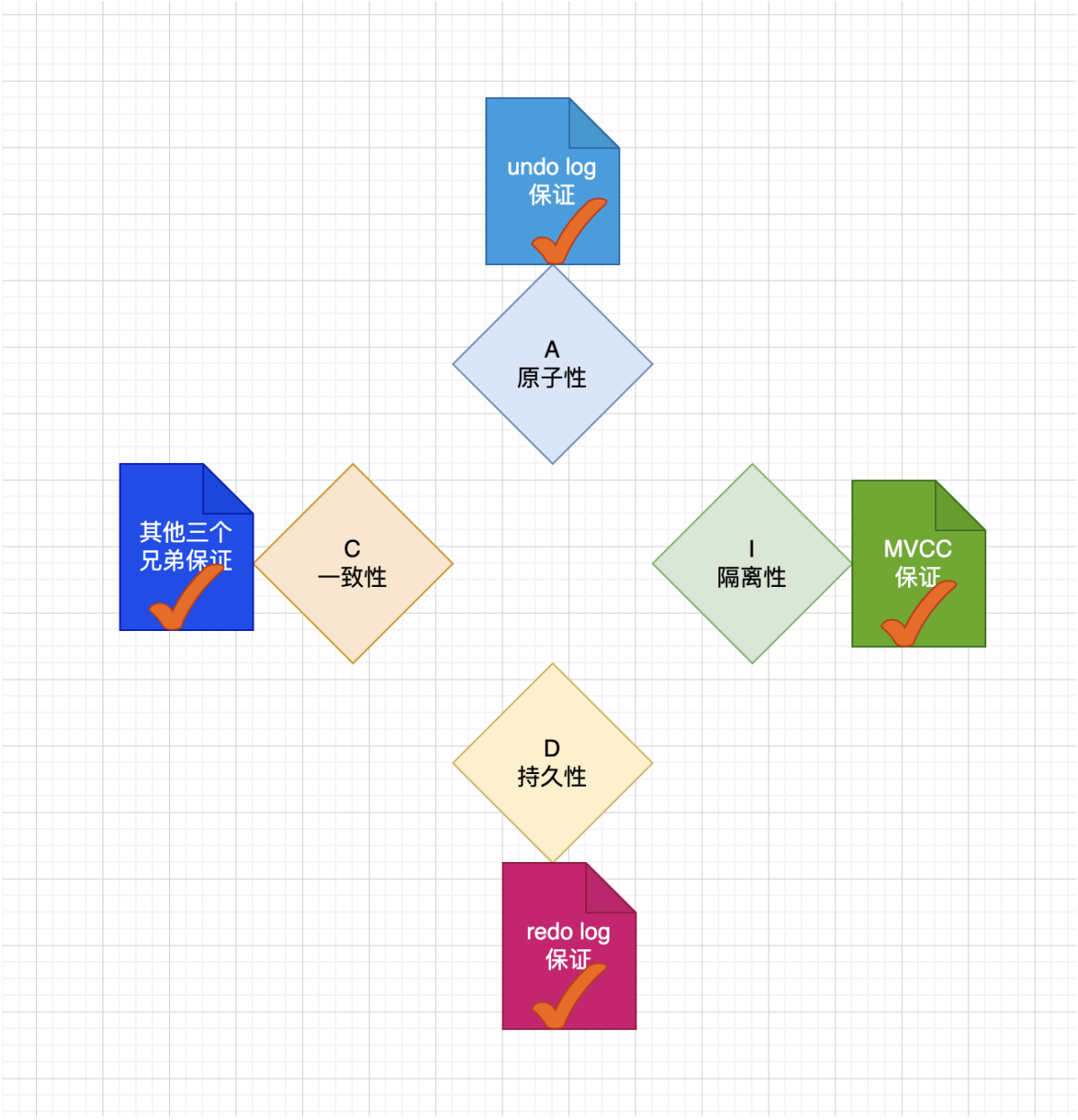
- 保证事务在不受外部并发操作影响的独立环境下运行

4. 持久性（Durability）

- 事务一旦提交或回滚，对数据库中的数据的改变就是永久的

ACID靠什么保证

MySQL通过事务、undo log、redo log来确保ACID



保证原子性

- 当事务开始时，MySQL 会在**undo log**中记录事务开始前的旧值。如果事务执行失败，MySQL 会使用**undo log**中的旧值来回滚事务开始前的状态；如果事务执行成功，MySQL 会在某个时间节点将**undo log**删除

保证一致性

- 如果其它三个特性得到了保证，一致性就自然而然得到保证

保证隔离性

- MySQL 定义了多种隔离级别，通过 **MVCC** 来确保每个事务都有专属自己的数据版本，从而实现隔离性

保证持久性

- **redo log** 是一种物理日志，当执行写操作时，MySQL 会先将更改记录到 **redo log** 中。当 **redo log** 填满时，MySQL 再将这些更改写入数据文件中。
- 如果 MySQL 在写入数据文件时发生崩溃，可以通过 **redo log** 来恢复数据文件，从而确保持久性（Durability）

事务会不会自动提交

- 在MySQL中，默认情况下事务是自动提交的，每执行一条SQL语句（如INSERT、UPDATE），都会被当作一个事务自动提交
- 如果需要手动控制事务，可以使用 START TRANSACTION 开启事务，并通过 COMMIT 或 ROLLBACK 完成事务。

并发事务问题是什么

1. 脏读

- 一个事务读到另一个事务还没有提交的数据

2. 不可重复读

- 一个事务先后读取同一条事务，但两次读取的数据不同

3. 幻读

- 一个事务按照条件查询数据时，没有对应的数据行，但是在插入数据时，又发现这行数据已经存在，好像出现了幻影

事物的隔离级别是什么

隔离级别	脏读	不可重复读	幻读
Read Uncommitted	✔	✔	✔
Read Committed	✘	✔	✔
Repeatable Read（默认）	✘	✘	✔
Serializable	✘	✘	✘

事务隔离级别越高，数据越安全，但是性能越低

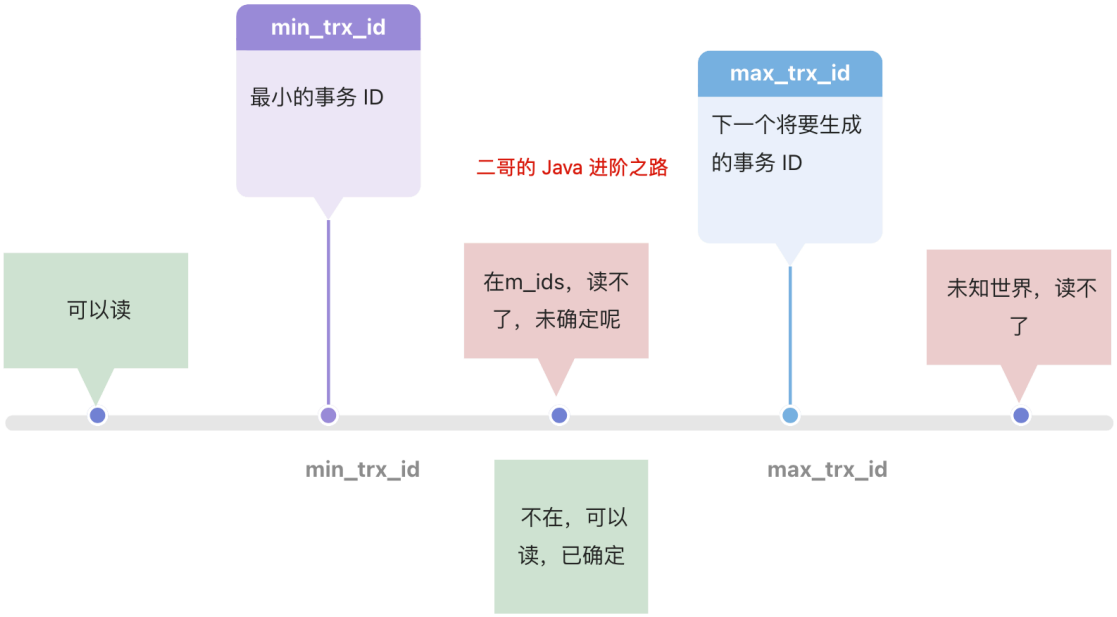
如何避免幻读

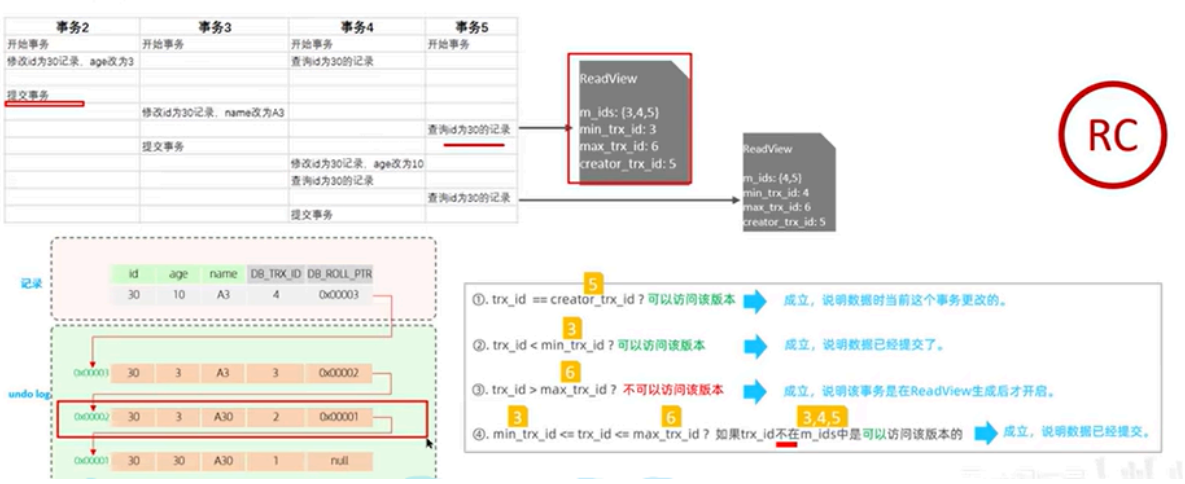
- 使用Serializable级别
- Repeatable Read 隔离级别下，通过 **间隙锁（Gap Lock）** 避免幻读：

什么是MVCC

多版本并发控制：事务隔离级别的无锁的实现方式，用于提高事务的并发性能

- 隐藏字段
  - DB_TRX_ID：最近修改事务ID
  - DB_ROLL_PTR：指向这条记录的上一个版本
  - DB_ROW_ID：隐藏字段
- 老数据存储在undo log中
- ReadView
  - 快照读SQL执行时MVCC提取数据的依据，记录并维护系统当前活跃的事务（未提交的）id
    - m_ids：当前活跃的事务ID集合
    - min_trx_id：最小活跃事务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





什么是快照读

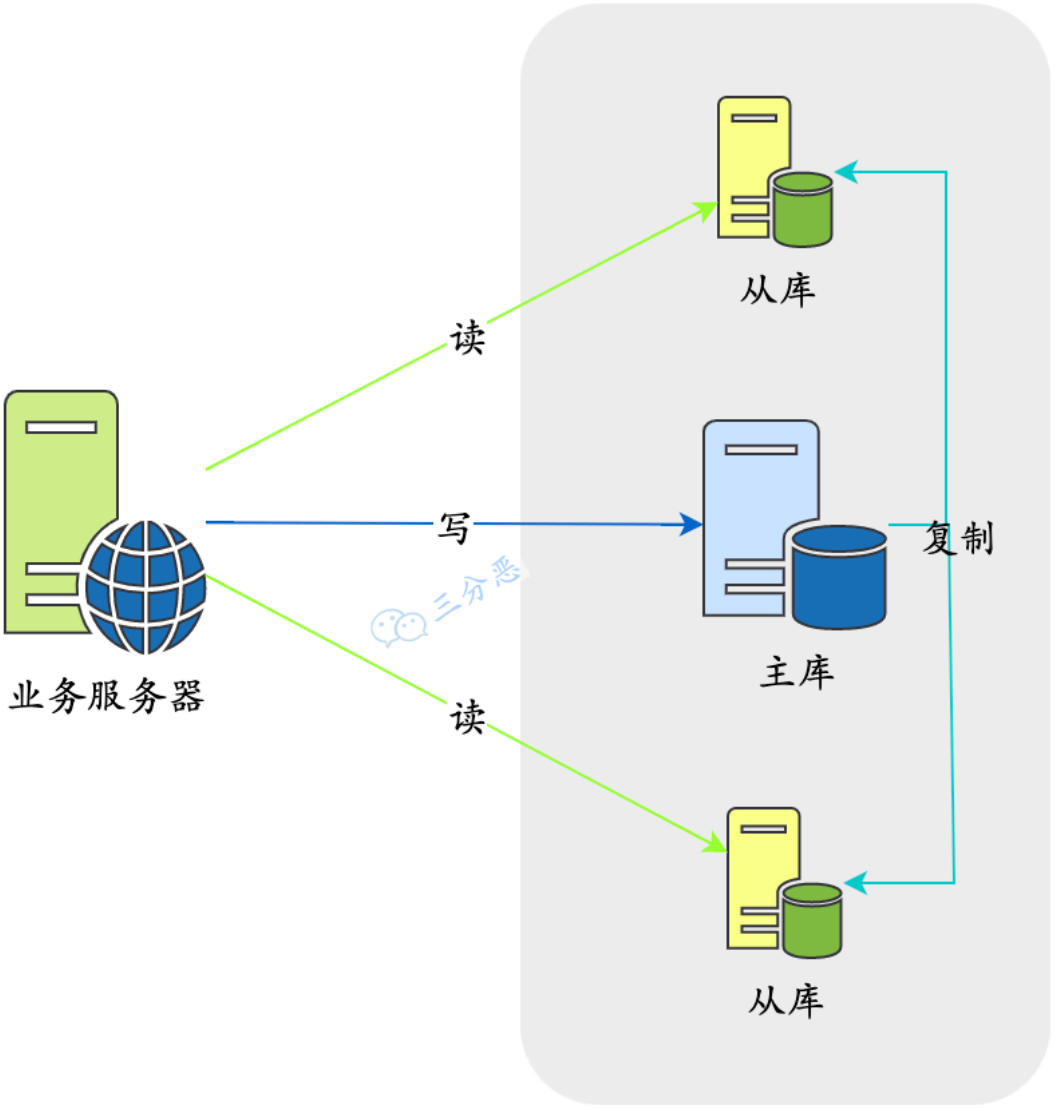
简单的select（不加锁）就是快照读，快照读读取的是记录数据的可见版本，有可能是历史数据，不加锁，是非阻塞读。

- Read Committed：每次select，都生成一个快照读，保证每次读操作都是最新的数据
- Repeatable Read：开启事务后第一个select语句才是生成快照读的地方，后续读操作都使用这个ReadView，保证事务内读取的数据是一致的

高可用/性能

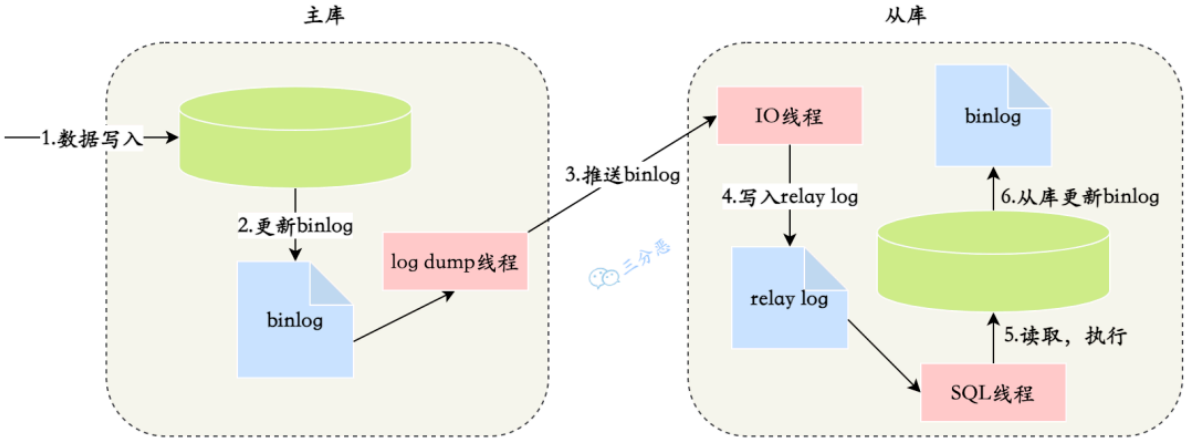
数据库读写分离了解吗

读写分离的基本原理是将数据库读写操作分散到不同的节点上



- 搭建主从集群
- 数据库主机负责读写操作，从机只负责读操作
- 数据库主机通过复制将数据同步到从机，每台数据库服务器都存储了所有的业务数据
- 业务服务器将写操作发给数据库主机，将读操作发给数据库从机

主从复制的原理了解吗



- 在主服务器上，所有修改数据的语句（INSERT、UPDATE、DELETE）会被记录到二进制日志中
- 主服务器上的一个线程（二进制日志转储线程）负责读取二进制日志的内容并发送给从服务器
- 从服务器接收到二进制日志数据后，会将这些数据写入自己的中继日志（Relay Log），中继日志是从服务器上的一个本地存储
- 从服务器上有一个SQL线程会读取中继日志，并在本地数据库上执行，从而将更改应用到从数据库中，完成同步

主从同步延迟怎么处理

主从同步延迟的产生原因：

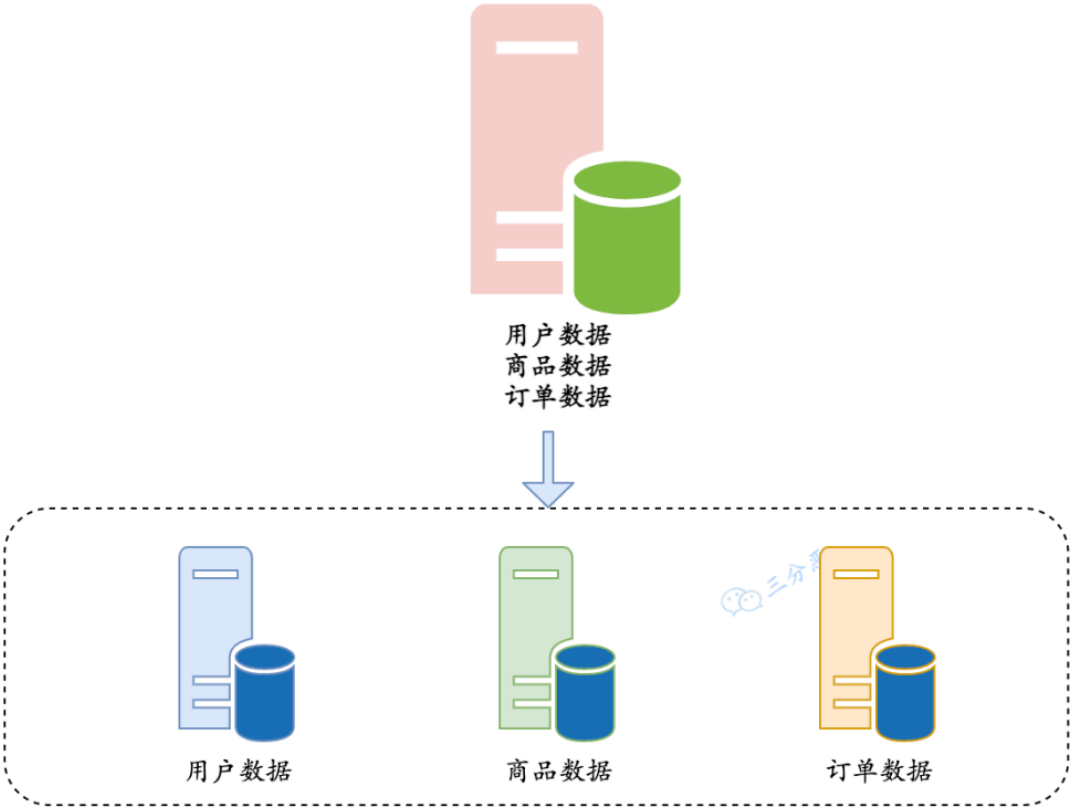
- 主库的binlog生成过快
- 从库的relay log处理速度慢

解决方法：

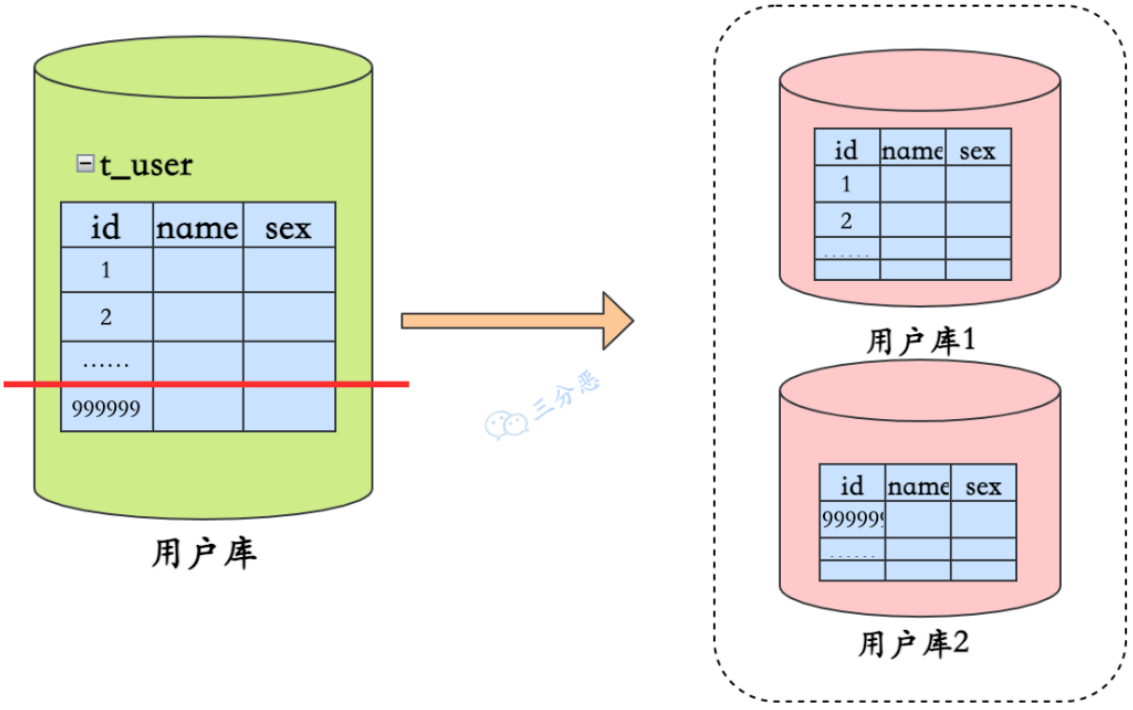
- 写操作后的读操作指定发给数据库主服务器
- 读从机失败后再读一次主机
- 关键业务读写操作全部指向主机，非关键业务采用读写分离

一般是怎么分库的

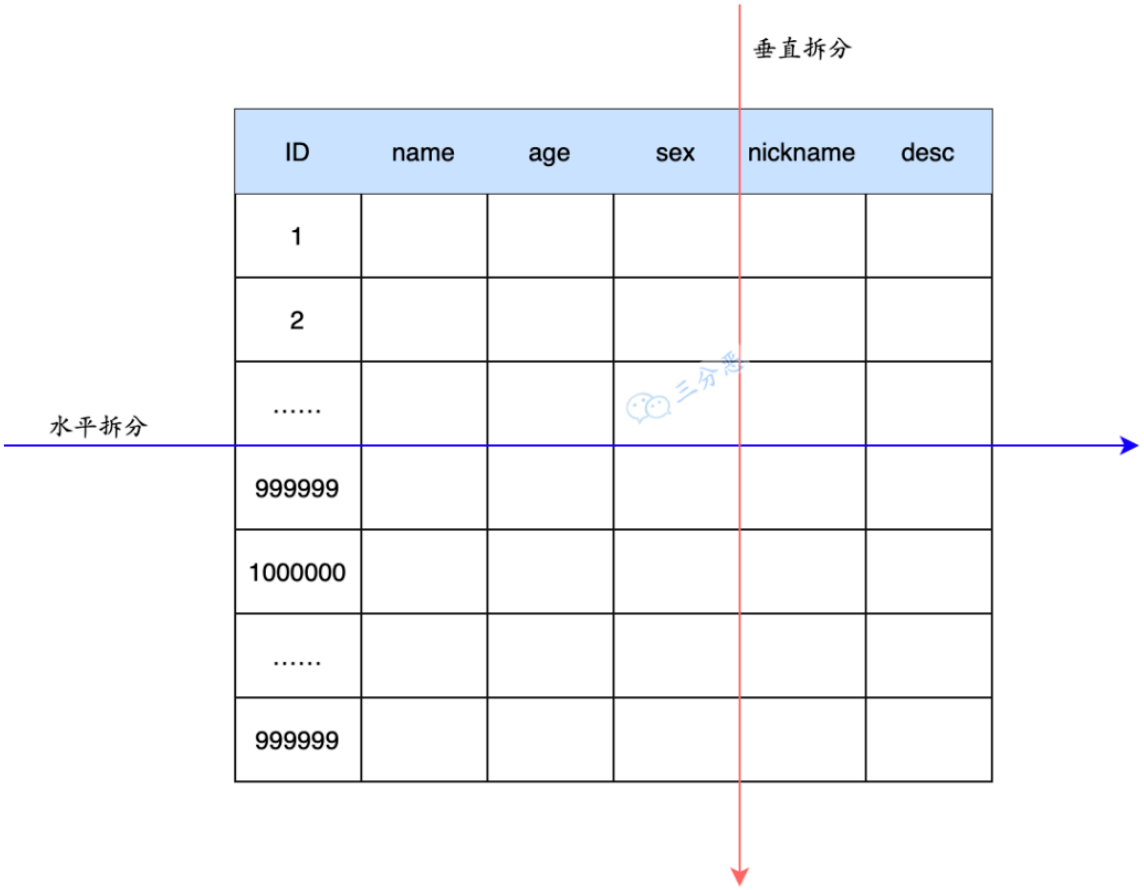
1. 垂直分库


用户数据
商品数据
订单数据
用户数据
商品数据
订单数据

2. 水平分库

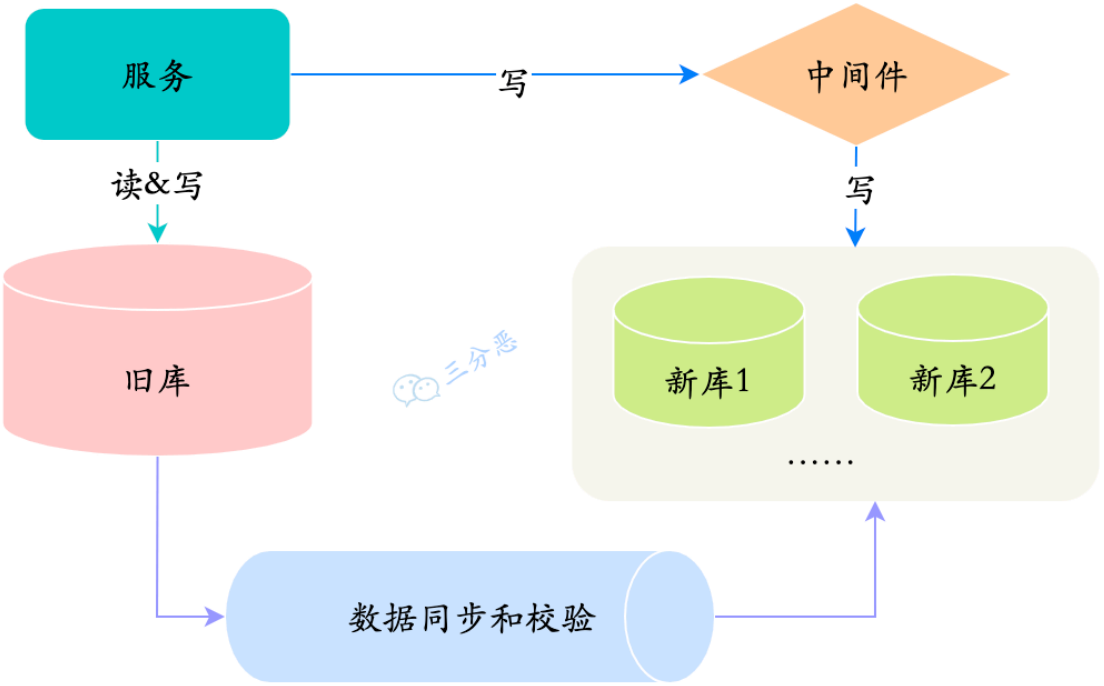

t_user
id
name
sex
1
2
.....
999999
用户库
id
name
sex
1
2
.....
用户库1
id
name
sex
999999
.....
用户库2

一般是怎么分表的

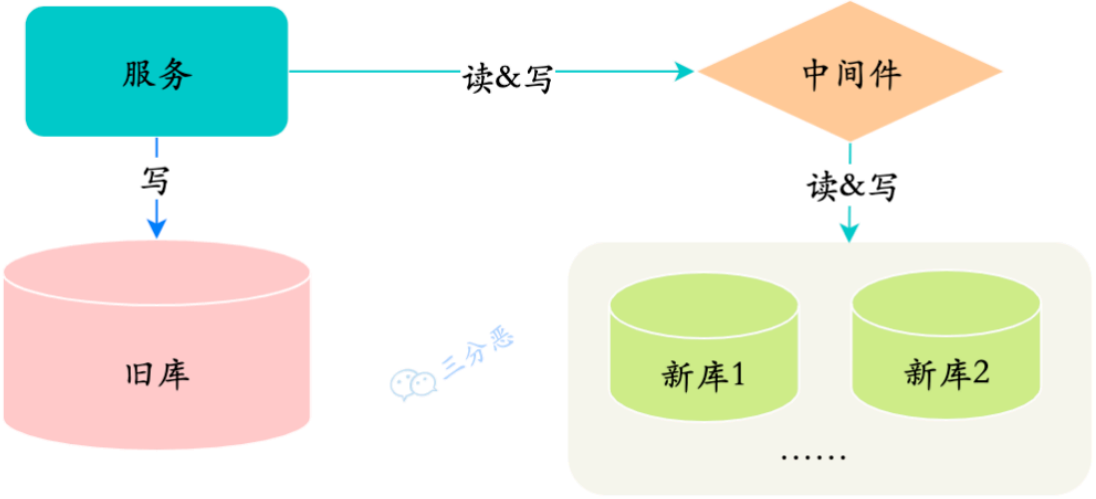


不停机扩容怎么实现

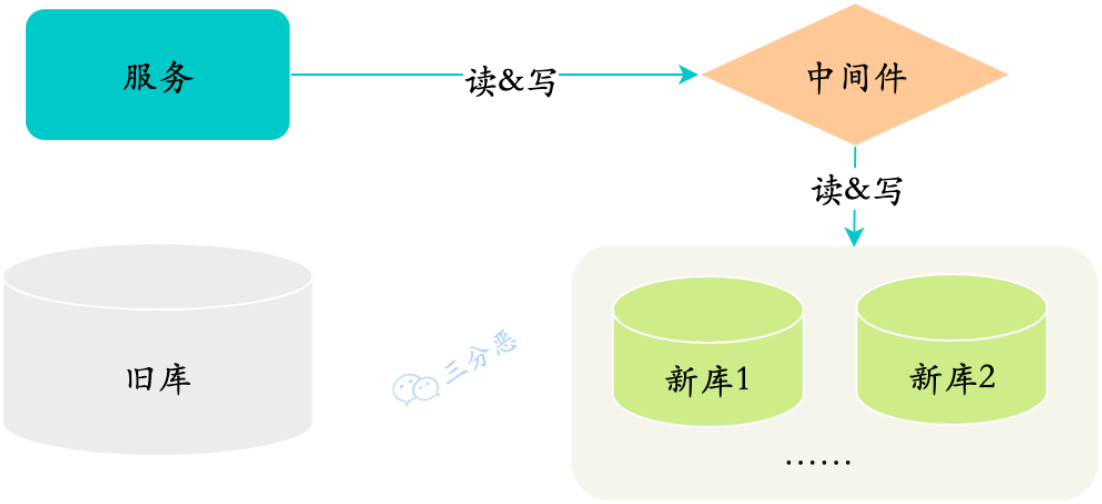
- **第一阶段：在线双写，查询走老库**
  - 建立好新的库表结构，数据写入久库的同时，也写入拆分的新库
  - 数据迁移，使用数据迁移程序，将旧库中的历史数据迁移到新库
  - 使用定时任务，新旧库的数据对比，把差异补齐



- **第二阶段：在线双写，查询走新库**
  - 完成了历史数据的同步和校验
  - 把对数据的读切换到新库



- **第三阶段：旧库下线**
  - 旧库不再写入新的数据
  - 经过一段时间，确定旧库没有请求之后，就可以下线老库



分库分表会带来什么问题呢

从分库的角度：

事务的问题

- 使用关系型数据库，很大一点在于它保证事务完整性
- 分库之后单机事务就用不上了，必须使用**分布式事务**(AT, XA)来解决

跨库JOIN问题

- 在一个库中的时候我们还可以利用 JOIN 来连表查询，而跨库了之后就无法使用 JOIN 了。
  - **在业务代码中进行关联**
  - **适当的冗余一些字段**
  - **数据异构**，通过 binlog 同步等方式，把需要跨库 join 的数据异构到 ES 等存储结构中，通过 ES 进行查询。

从分表的角度：

- 跨节点的count，order by，group by以及聚合函数问题
- ID问题
  - 数据库表被切分后，不能再依赖数据库自身的主键生成机制，所以需要一些手段来保证全局主键唯一。
  - UUID不是自增的，可能产生页分裂问题
  - 分布式 ID，比较出名的就是 Twitter 开源的 sonwflake 雪花算法

雪花算法了解吗

- 第一个部分：`1个bit`，无意义，固定为0。二进制中最高位是符号位，1表示负数，0表示正数。ID都是正整数，所以固定为0。
- 第二个部分：`41个bit`，表示时间戳，精确到毫秒，2^41/(1000_60_60_24_365)=69，大概可以使用 69 年。时间戳带有自增属性。
- 第三个部分：`10个bit`，表示10位的机器标识，最多支持2^10=1024个节点。此部分也可拆分成5位datacenterId和5位workerId，datacenterId表示机房ID，workerId表示机器ID。
- 第四部分：`12个bit`，表示序列号，同一毫秒时间戳时，通过这个递增的序列号来区分。即对于同一台机器而言，同一毫秒时间戳下，可以生成 2^12=4096 个不重复 id。

百万级别以上数据如何快速删除

由于索引需要额外的维护成本，每删除一条数据，数据库需要同步调整索引（B+ 树重平衡、维护索引页）

所以：

1. 我们想要删除百万数据的时候可以先删除索引
2. 然后删除表中的无用数据
3. 删除完成后重新创建索引也非常快

百万千万级大表如何添加字段

- 通过中间表转换过去
  - 创建一个临时的新表，把旧表的结构完全复制过去，添加字段，再把旧表数据复制过去，删除旧表，新表命名为旧表的名称，这种方式可能回丢掉一些数据。
- 先在从库添加 再进行主从切换
  - 如果一张表数据量大且是热表（读写特别频繁），则可以考虑先在从库添加，再进行主从切换，切换后再将其他几个节点上添加字段。

怎么处理MySQL CPU飙升

1. 查看MySQL进程的CPU使用率
2. 查看MySQL的活跃查询

```
SHOW FULL PROCESSLIST;
```

3. 找出消耗高的 sql，看看执行计划是否准确， 索引是否缺失，数据量是否太大。

参考资料

黑马程序员MySQL

- P51 - P88 事务到索引
- P121 - P132 锁
- P154 - P157 日志