数据库概述

关系型数据库以`行(row)`和`列(column)`的形式存储数据

非关系型数据库，基于键值对存储数据，不需要经过SQL层的解析，`性能非常高`。

SQL

DDL（Data Definition Languages、数据定义语言）

定义了不同的数据库、表、视图、索引等数据库对象，还可以用来创建、删除、修改数据库和数据表的结构。

DML（Data Manipulation Language、数据操作语言）

对数据增删改查

DCL（Data Control Language、数据控制语言）

定义数据库、表、字段、用户的访问权限和安全级别。

字符串型和日期时间类型的数据可以使用单引号（' '）表示

列的别名，尽量使用双引号（" "），而且不建议省略as

数据导入 mysql> **source** d:\mysqldb.sql

空值参与运算结果都为空

（%）

（<=>）安全等于，可以判断null

& | ^ ~ >> <<

DML

INSERT INTO table_name(column1 [, column2, ..., columnn])
VALUES
(value1 [,value2, ..., valuen]),
.....
(value1 [,value2, ..., valuen]);

UPDATE table_name
SET column1=value1, column2=value2, ... , column=valuen
[WHERE condition]

DELETE FROM table_name [WHERE condition]

select (distinct)

from ``

(inner/left/right/cross) **join on** mysql不支持full join

UNION (ALL)

NATURAL JOIN 自动查询两张连接表中`所有相同的字段`，然后进行`等值连接`。

USING USING (department_id)只能和JOIN一起使用，而且要求**两个**关联字段在关联表中名称一致，而且只能表示**关联字段值相等**

多表连接就相当于嵌套 for 循环一样，非常消耗资源

超过三个表禁止 join。需要 join 的字段，数据类型保持绝对一致；多表关联查询时， 保证被关联的字段需要有索引。

where

group by xxx , xxx (WITH ROLLUP增加一条记录，计算查询出的所有记录的总和，与ORDER BY互斥)

和聚合函数一起使用的列都要分组

having

having中可以使用聚合函数，如having max(salary) > 10000

能用where就用where，having的效率较低

order by (desc) 可以使用select中的字段

limit offset , rows

每一步都会生成一个虚拟表，下一步在上一步生成的虚拟表的基础上进行

单行函数

单行函数可以嵌套

一行数据返回一个值

数值函数

ABS(x) **LEAST**(e1,e2,e3...) **GREATEST**(e1,e2,e3...)

字符串函数

CHAR_LENGTH(s) **CONCAT**(s1,s2,.....,sn) **REPLACE**(str, a, b)

UPPER(s) **LEFT**(str,n) **TRIM**(s) **SUBSTR**(s,index,len)

REVERSE(s)

日期和时间函数

CURDATE() **CURTIME**() **NOW**()

UNIX_TIMESTAMP() **UNIX_TIMESTAMP**(date)

YEAR(date) **MONTH**(date) **DAY**(date)

HOUR(time) **MINUTE**(time) **SECOND**(time)

TIME_TO_SEC(time)

DATE_FORMAT(date,fmt)

流程控制函数

IF(value,value1,value2)

CASE WHEN 条件1 THEN 结果1 WHEN 条件2 THEN 结果2 .... [ELSE resultn] END

CASE expr WHEN 常量值1 THEN 值1 WHEN 常量值1 THEN 值1 .... [ELSE 值n] END

加密与解密函数

MD5(str)

聚合函数

聚合函数不能嵌套

一组数据返回一个值

AVG() SUM() MAX() MIN() COUNT()

子查询

子查询要包含在括号内

子查询生成的表一定要取别名

单行子查询

多行子查询

IN() ANY() ALL()

相关子查询

通常情况下都是因为**子查询用到了外部的表，并进行了条件关联**，因此每执行一次外部查询，子查询都要重新计算一次

子查询的执行依赖于外部查询，子查询中使用了主查询的列

相关子查询按照一行接一行的顺序执行，主查询的每一行都执行一次子查询。

EXISTS() NOT EXISTS()

非相关子查询

在许多 DBMS 的处理过程中，对于连接的处理速度要比子查询快得多。

DDL

CREATE DATABASE IF NOT EXISTS 数据库名;

DROP DATABASE IF EXISTS 数据库名;

ALTER DATABASE 数据库名 CHARACTER SET 字符集;

SHOW DATABASES;

USE 数据库名;

SELECT DATABASE();

SHOW TABLES FROM 数据库名;

SHOW CREATE DATABASE 数据库名;

CREATE TABLE [IF NOT EXISTS] 表名(
    字段1, 数据类型 [约束条件] [默认值],
    ……
    [表约束条件]
);

计算列：某一列的值通过别的列计算得来，CREATE TABLE 和 ALTER TABLE 中都支持增加计算列

c INT GENERATED ALWAYS AS (a + b) VIRTUAL

DROP TABLE [IF EXISTS] 数据表1 [, 数据表2, ..., 数据表n]; 无法回滚

ALTER TABLE 表名 **ADD** 【COLUMN】 字段名 字段类型 【FIRST|AFTER 字段名】;

ALTER TABLE 表名 **MODIFY** 【COLUMN】 字段名1 字段类型 【DEFAULT 默认值】【FIRST|AFTER 字段名2】;

ALTER TABLE 表名 **CHANGE** 【column】 列名 新列名 新数据类型;

ALTER TABLE 表名 **DROP** 【COLUMN】字段名

RENAME TABLE emp TO myemp;

TRUNCATE TABLE detail_dept; 清空表，无事务不能回滚，使用 DELETE 语句删除数据，可以回滚

DESC employees显示表结构

SHOW CREATE TABLE 表名

数据类型

整数类型

整数类型	字节	有符号数取值范围	无符号数取值范围
TINYINT	1	-128~127	0~255
SMALLINT	2	-32768~32767	0~65535

整数类型	字节	有符号数取值范围	无符号数取值范围
MEDIUMINT	3	-8388608~8388607	0~16777215
INT、INTEGER	4	-2147483648~2147483647	0~4294967295
BIGINT	8	-9223372036854775808~9223372036854775807	0~18446744073709551615

UNSIGNED

浮点类型 可以处理小数

FLOAT 4 DOUBLE 8

因为浮点数是**不准确**的，所以我们**要避免使用"="来判断两个数是否相等。**

在一些对精确度要求较高的项目中，千万不要使用浮点数

定点数类型

数据类型	字节数	含义
DECIMAL(M,D),DEC,NUMERIC	M+2字节	有效范围由M和D决定

高精度，没有误差，适合于对精度要求极高的场景 （比如涉及金额计算的场景）

定点数在MySQL内部是以`字符串`的形式进行存储，这就决定了它一定是精准的。

日期与时间类型

类型	名称	字节	日期格式	最小值	最大值
YEAR	年	1	YYYY或YY	1901	2155
TIME	时间	3	HH:MM:SS	-838:59:59	838:59:59
DATE	日期	3	YYYY-MM-DD	1000-01-01	9999-12-03
DATETIME	日期时间	8	YYYY-MM-DD HH:MM:SS	1000-01-01 00:00:00	9999-12-31 23:59:59
TIMESTAMP	日期时间	4	YYYY-MM-DD HH:MM:SS	1970-01-01 00:00:00 UTC	2038-01-19 03:14:07UTC

DATETIME用的最多

如果涉及到需要计算，则一般用时间戳

文本字符串类型

字符串(文本)类型	特点	长度	长度范围	占用的存储空间
CHAR(M)	固定长度	M	0 <= M <= 255	M个字节

字符串(文本)类型	特点	长度	长度范围	占用的存储空间
VARCHAR(M)	可变长度	M	0 <= M <= 65535	(实际长度 + 1) 个字节

类型	特点	空间上	时间上	适用场景
CHAR(M)	固定长度	浪费存储空间	效率高	存储不大，速度要求高
VARCHAR(M)	可变长度	节省存储空间	效率低	非CHAR的情况

TEXT类型

文本字符串类型	特点	长度	长度范围	占用的存储空间
TINYTEXT	小文本、可变长度	L	0 <= L <= 255	L + 2 个字节
TEXT	文本、可变长度	L	0 <= L <= 65535	L + 2 个字节
MEDIUMTEXT	中等文本、可变长度	L	0 <= L <= 16777215	L + 3 个字节
LONGTEXT	大文本、可变长度	L	0 <= L<= 4294967295（相当于4GB）	L + 4 个字节

TEXT文本类型，可以存比较大的文本段，搜索速度稍慢，因此如果不是特别大的内容，建议使用CHAR，VARCHAR来代替。

在定义数据类型时，如果确定是`整数`，就用`INT`； 如果是`小数`，一定用定点数类型`DECIMAL(M,D)`； 如果是日期与时间，就用`DATETIME`。

任何字段如果为非负数，必须是 UNSIGNED

如果存储的字符串长度几乎相等，使用 CHAR 定长字符串类型。

VARCHAR 是可变长字符串，不预先分配存储空间，长度不要超过 5000。如果存储长度大于此值，定义字段类型为 TEXT，独立出来一张表，用主键来对应，避免影响其它字段索引效率。

约束

- **NOT NULL 非空约束，规定某个字段不能为空**
- **UNIQUE 唯一约束，规定某个字段在整个表中是唯一的**
- **PRIMARY KEY 主键(非空且唯一)约束**
- **FOREIGN KEY 外键约束**

  [CONSTRAINT <外键约束名称>] FOREIGN KEY（从表的某个字段) references 主表名(被参考字段)

  当创建外键约束时，系统默认会在所在的列上建立对应的普通索引。但是索引名是外键的约束名。

删除外键约束后，必须手动删除对应的索引

外键约束（FOREIGN KEY）不能跨引擎使用。

- **CHECK 检查约束** MySQL8.0开始支持

  gender char check ('男' or '女')
- **DEFAULT 默认值约束**
- **AUTO_INCREMENT**

列级约束

表级约束

[constraint 约束名] unique key(字段名，)

[constraint 约束名] primary key(字段名，)

不得使用外键与级联，一切外键概念必须在应用层解决。

视图

用来查询

视图是**基于查询语句创建的一种**虚拟表，一个或者多个数据表里的数据的逻辑显示，视图并不存储数据

CREATE [OR REPLACE]
[ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
VIEW 视图名称 [(字段列表)]
AS 查询语句
[WITH [CASCADED|LOCAL] CHECK OPTION]

DROP VIEW IF EXISTS 视图名称1,视图名称2,视图名称3,...;

ALTER VIEW 视图名称
AS
查询语句

SHOW TABLES; DESC / DESCRIBE 视图名称;

SHOW CREATE VIEW 视图名称;

可基于视图创建视图

存储过程与函数

存储过程

一组经过预先编译的 SQL 语句的封装。

存储过程的参数类型可以是IN、OUT和INOUT。根据这点分类如下：

1、没有参数（无参数无返回）
2、仅仅带 IN 类型（有参数无返回）
3、仅仅带 OUT 类型（无参数有返回）
4、既带 IN 又带 OUT（有参数有返回）

5、带 INOUT（有参数有返回）

DELIMITER $

CREATE PROCEDURE 存储过程名(IN|OUT|INOUT 参数名 参数类型,...)
[characteristics ...]
BEGIN
  sql语句1;
  sql语句2;

END $

DELIMITER ;

CALL 存储过程名(实参列表)

函数

CREATE FUNCTION 函数名(参数名 参数类型,...)
RETURNS 返回值类型
[characteristics ...]
BEGIN
  函数体 #函数体中肯定有 RETURN 语句

END

FUNCTION中总是默认为**IN参数**

SELECT 函数名(实参列表)

	关键字	**调用语法**	**返回值**	**应用场景**
存储过程	PROCEDURE	CALL 存储过程()	理解为有0个或多个	一般用于更新
存储函数	FUNCTION	SELECT 函数()	只能是一个	一般用于查询结果为一个值并返回时

SHOW CREATE {PROCEDURE | FUNCTION} 存储过程名或函数名

SHOW {PROCEDURE | FUNCTION} STATUS [LIKE 'pattern']

ALTER {PROCEDURE | FUNCTION} 存储过程或函数的名 [characteristic ...]

DROP {PROCEDURE | FUNCTION} [IF EXISTS] 存储过程或函数的名

存储过程优点

1、存储过程可以一次编译多次使用。存储过程只在**创建时进行编译，之后的使用都不需要重新编译，这就提升了 SQL 的执行效率。**

2、可以减少开发工作量。将代码`封装`成模块，实际上是编程的核心思想之一，这样可以把复杂的问题拆解成不同的模块，然后模块之间可以`重复使用`，在减少开发工作量的同时，还能保证代码的结构清晰。

3、存储过程的安全性强。我们在设定存储过程的时候可以`设置对用户的使用权限`，这样就和视图一样具有较强的安全性。

4、可以减少网络传输量。因为代码封装到存储过程中，每次使用只需要调用存储过程即可，这样就减少了网络传输量。

5、良好的封装性。在进行相对复杂的数据库操作时，原本需要使用一条一条的 SQL 语句，可能要连接多次数据库才能完成的操作，现在变成了一次存储过程，只需要`连接一次即可`。

存储过程缺点

1、可移植性差。存储过程**不能跨数据库移植**，比如在 MySQL、Oracle 和 SQL Server 里编写的存储过程，在换成其他数据库时都需要重新编写。

2、调试困难。只有少数 DBMS 支持存储过程的调试。对于复杂的存储过程来说，开发和维护都不容易。虽然也有一些第三方工具可以对存储过程进行调试，但要收费。

3、存储过程的版本管理很困难。比如数据表索引发生变化了，可能会导致存储过程失效。我们在开发软件的时候往往需要进行版本管理，但是存储过程本身没有版本控制，版本迭代更新的时候很麻烦。

4、它不适合高并发的场景。高并发的场景需要减少数据库的压力，有时数据库会采用分库分表的方式，而且对可扩展性要求很高，在这种情况下，存储过程会变得难以维护，`增加数据库的压力`，显然就不适用了。

变量、流程控制与游标

`系统变量`

全局系统变量（需要添加`global`关键字）

SHOW GLOBAL VARIABLES;

SET @@global.变量名=变量值;

SELECT @@global.变量名;

全局变量持久化：

使用SET GLOBAL语句设置的变量值只会`临时生效`。`数据库重启`后，服务器又会从MySQL配置文件中读取变量的默认值。

MySQL8新增 SET PERSIST global max_connections = 1000;

会话系统变量（需要添加`session`关键字）

SHOW SESSION VARIABLES;

SET @@session.变量名=变量值;

SELECT @@session.变量名;

`用户自定义变量`

会话用户变量

SET @用户变量 = 值;

SELECT 表达式 INTO @用户变量 [FROM 等子句];

SELECT @用户变量

局部变量

在 **BEGIN 和 END 语句块**中有效。局部变量只能在`存储过程和函数`中使用。

只能放在 BEGIN ... END 中，而且**只能放在第一句**

DECLARE 变量名 类型 [default 值];

SET 变量名=值;

SELECT 字段名或表达式 INTO 变量名 FROM 表;

SELECT 局部变量名;

定义条件与处理程序

`定义条件`是事先定义程序执行过程中可能遇到的问题，`处理程序`定义了在遇到问题时应当采取的处理方式

流程控制

IF 表达式1 THEN 操作1
[ELSEIF 表达式2 THEN 操作2].....
[ELSE 操作N]
END IF

CASE 表达式
WHEN 值1 THEN 结果1或语句1(如果是语句，需要加分号)
WHEN 值2 THEN 结果2或语句2(如果是语句，需要加分号)
...
ELSE 结果n或语句n(如果是语句，需要加分号)
END [case]（如果是放在begin end中需要加上case，如果放在select后面不需要）

CASE
WHEN 条件1 THEN 结果1或语句1(如果是语句，需要加分号)
WHEN 条件2 THEN 结果2或语句2(如果是语句，需要加分号)
...
ELSE 结果n或语句n(如果是语句，需要加分号)
END [case]（如果是放在begin end中需要加上case，如果放在select后面不需要）

[loop_label:] **LOOP**
  循环执行的语句
END LOOP [loop_label]

[while_label:] **WHILE** 循环条件 DO
  循环体
END WHILE [while_label];

[repeat_label:] **REPEAT**
　　循环体的语句
UNTIL 结束循环的条件表达式
END REPEAT [repeat_label]

LEAVE 标记名

可以用在循环语句内，或者以 BEGIN 和 END 包裹起来的程序体内，表示跳出循环或者跳出程序体的操作

ITERATE label

只能用在循环语句（LOOP、REPEAT和WHILE语句）内，类似continue

游标

能够**对结果集中的每一条记录进行定位，逐条读取结果集中的数据**

可以通过操作游标来对数据行进行操作

DECLARE cursor_name CURSOR FOR select_statement;

OPEN cursor_name

FETCH cursor_name INTO var_name [, var_name] ...

使用 cursor_name 这个游标来读取当前行，并且将数据保存到 var_name，然后游标指针指到下一行。

var_name必须在声明游标之前就定义好。

游标的查询结果集中的字段数，必须跟 INTO 后面的变量数一致

CLOSE cursor_name

在使用游标的过程中，**会对数据行进行**`加锁`**，这样在业务并发量大的时候**，不仅会影响业务之间的效率，还会`消耗系统资源`，造成内存不足，这是因为游标是在内存中进行的处理。

用完之后就关闭

触发器

MySQL的触发器和存储过程一样，都是嵌入到MySQL服务器的一段程序。

触发器是由`事件来触发`某个操作，这些事件包括`INSERT`、`UPDATE`、`DELETE`事件。

DELIMITER //

CREATE TRIGGER 触发器名称
{BEFORE|AFTER} {INSERT|UPDATE|DELETE} ON 表名
FOR EACH ROW
BEGIN
　触发器执行的语句块;

END //

DELIMITER ;

SHOW TRIGGERS

SHOW CREATE TRIGGER 触发器名

DROP TRIGGER IF EXISTS 触发器名称;

逻辑架构和存储引擎

MySQL体系结构

连接层

主要完成连接处理，授权认证等，引入了线程池的概念

服务层

主要完成大多数的核心服务功能，如SQL接口，并完成缓存的查询，SQL的分析和优化，部 分内置函数的执行。所有跨存储引擎的功能也在这一层实现

引擎层

存储引擎真正的负责了MySQL中数据的存储和提取，服务器通过API和存储引擎进行通信，数据库 中的索引是在存储引擎层实现的。

存储层

主要是将数据(如: redolog、undolog、数据、索引、二进制日志、错误日志、查询 日志、慢查询日志等)存储在文件系统之上，并完成与存储引擎的交互。

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**

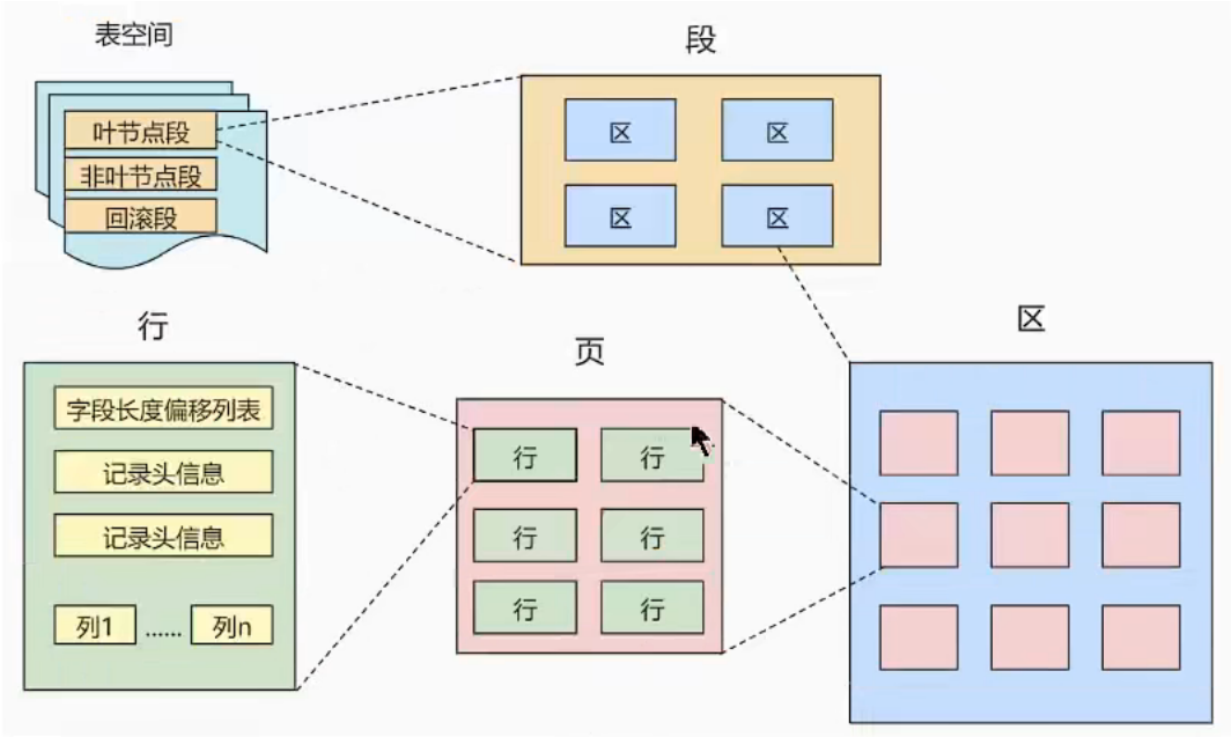
ENGINE = INNODB

INNODB

支持事务

行锁和表锁，提高并发访问性能

支持外键



xxx.ibd：xxx代表的是表名，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、数据和索引信息。

InnoDB将数据划分为若干个页，InnoDB中页的大小默认为 **16KB**

以`页`作为**磁盘和内存之间交互**的`基本单位`，也就是一次最少从磁盘中读取16KB的内容到内存中，一次最少把内存中的16KB内容刷新到磁盘中。

页a、页b、页c ...页n这些页可以不`在物理结构上相连`，只要通过`双向链表`相关联即可

每个数据**页中的记录会按照主键值从小到大的顺序组成一个**`单向链表`

一个区会分配`64个连续的页`。因为InnoDB中的页大小默认是16KB，所以一个区的大小是64*16KB= 1MB。

`段是数据库中的分配单位，不同类型的数据库对象以不同的段形式存在。`当创建数据表、索引的时候，就会相应创建对应的段，比如**创建一张表时会创建一个表段，创建一个索引时会创建一个索引段。**

索引

索引（Index）是帮助MySQL高效获取数据的数据结构。

提高数据检索的效率，降低数据库的IO成本

减少查询中分组和排序的时 间 ， 降低了CPU的消耗。

创建索引和维护索引要耗费时间

索引需要占磁盘空间

降低更新表的速度，当对表 中的数据进行增加、删除和修改的时候，索引也要动态地维护

索引建立实际上是自上而下的过程

二级索引的**非叶子节点**实际上会存储：

索引列的值

主键值

页号

二级索引索引列值是可以重复的

索引数据结构选择

Hash结构 数组+链表+红黑树

等值查询，增删改查效率很高

不适合范围查询

数据存储没有顺序

如果索引列重复值很多，效率就会降低

INNODB不支持hash索引，但支持自适应hash索引

索引创建与设计原则

MySQL的索引包括普通索引、唯一性索引、全文索引、单列索引、多列索引和空间索引等。

- 从`功能逻辑`上说，索引主要有 4 种，分别是普通索引、唯一索引、主键索引、全文索引。
- 按`照物理实现方式`，索引可以分为 2 种：聚簇索引和非聚簇索引。
- 按照`作用字段个数`进行划分，分成单列索引和联合索引。

1. 普通索引 index
2. 唯一性索引 unique
3. 主键索引 primary
4. 单列索引
5. 多列(组合、联合)索引

   使用时遵循最左前缀
6. 全文索引

   一般用elasticsearch
7. 补充：空间索引

CREATE [UNIQUE | FULLTEXT] INDEX index_name ON table_name (index_col_name,...) ;

SHOW INDEX FROM table_name ;

DROP INDEX index_name ON table_name ;

在需要大量删除表数据，修改表数据时，可以考虑先删除索引。等修改完数据之后再插入

MySQL8 支持**降序索引**

降序存储键值

CREATE TABLE ts1(a int, b int, index idx_a_b(a, b desc)) ;

MySQL8 支持**隐藏索引**

将**待删除的索引设置为隐藏索引**，使查询优化器不再使用这个索引（即使使用force index（强制使用索引），优化器也不会使用该索引）**确认将索引设置为隐藏索引后系统不受任何响应，就可以彻底删除索引。**`这种通过先将索引设置为隐藏索引，再删除索引的方式就是软删除`。就是可以**不直接删除，先看看有没有影响，确认没有影响再删**

想验证某个索引删除之后的`查询性能影响`，就可以暂时先隐藏该索引

CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX index_name ON table_name (col_name[length] [ASC | DESC] ,...) [INVISIBLE|VISIBLE]

ALTER TABLE book2 alter index idx_name visible; # 切换成非隐藏索引
ALTER TABLE book2 alter index idx_name invisible; # 切换成非隐藏索引

当索引被隐藏时，它的内容仍然是和正常索引一样实时更新的。如果一个索引需要长期被隐藏，那么可以将其删除，因为索引的存在会影响插入、更新和删除的性能。

可以通过查询优化器的一个开关（use_invisible_indexes）来打开某个设置，使隐藏索引对查询优化器可见。如果 use_invisible_indexes设置为off(默认)，优化器会忽略隐藏索引。**如果设置为on，即使隐藏索引不可见，优化器在生成执行计划时仍会考虑使用隐藏索引。**

适合创建索引的情况

1.**具有唯一特性的字段，即使是组合字段**

2.频繁作为 **WHERE 查询条件的字段**

3.经常 **GROUP BY 和 ORDER BY 的列**

4.UPDATE、DELETE 的 WHERE 条件列

5.**DISTINCT 字段需要创建索引**

6.**多表 JOIN 连接操作**时，对WHERE条件创建索引，对用于连接的字段创建索引

7.使用**列的类型小的创建索引**

8.在 **varchar 字段上建立索引时，使用字符串前缀创建索引**

```
create table shop(address varchar( 120 ) not null);
alter table shop add index(address( 12 ));

select count(distinct left(列名, 索引长度))/count(*) from tbl_name #计算选择度，越大越好
```

要排序的字段不能添加索引列前缀

使用前缀索引就用不上覆盖索引对查询性能的优化了，这也是你在选择是否使用前缀索引时需要考虑的一个因素。

9.**区分度高(散列性高)的列适合作为索引**

10.使用**最频繁的列放到联合索引的左侧**

11.在多个字段都要创建索引的情况下，**联合索引优于单值索引**

不适合创建索引的情况

1.在where中使用不到的字段，不要设置索引

2.数据量小的表最好不要使用索引

3.有大量重复数据的列上不要建立索引

4.避免对经常更新的表创建过多的索引

5.**不建议用无序的值作为索引**

6.删除不再使用或者很少使用的索引

7.不要定义冗余或重复的索引

性能分析

SHOW GLOBAL STATUS LIKE 'Com%';

可以查看当前数据库的INSERT、UPDATE、DELETE、SELECT的访问频次

慢查询日志

慢查询日志记录了所有执行时间超过指定参数（long_query_time，单位：秒，默认10秒）的所有 SQL语句的日志。

EXPLAIN

定位了查询慢的SQL之后，我们就可以使用EXPLAIN或DESCRIBE工具做**针对性的分析查询语句。**

MySQL中有专门负责优化SELECT语句的优化器模块，主要功能: 通过计算分析系统中收集到的统计信息，为客户端请求的Query提供它认为最优的 `执行计划`

EXPLAIN`语句来帮助我们查看某个查询语句的具体执行计划

列名	描述

列名	描述
id	在一个大的查询语句中每个SELECT关键字都对应一个 唯一的id，也有例外的可能，查询优化器做了优化（查询优化器可能对涉及子查询的查询语句进行重写,转变为多表查询的操作） **id如果相同**，可以认为是一组，**从上往下顺序执行** **在所有组中，id值越大，优先级越高，越先执行** 关注点：**id号每个号码，表示一趟独立的查询，一个sql的查询趟数越少越好**
select_type	MySQL为**每一个SELECT关键字代表的小查询**都定义了一个称之为`select_type`的属性，意思是我们只要知道了某个小查询的`select_type属性`，就知道了这个`小查询在整个大查询中扮演了一个什么角色` **SIMPLE**：查询语句中不包含`UNION`或者子查询的查询都算作是`SIMPLE`类型 **PRIMARY、UNION、UNION RESULT**：对于包含`UNION`或者`UNION ALL`或者子查询的大查询来说，它是由几个小查询组成的，其中最左边的那个查询的`select_type`值就是`PRIMARY`，其余的查询的`select_type`值就是`UNION`，`MySQL`选择使用临时表来完成`UNION`查询的去重工作，针对该临时表的查询的`select_type`就是`UNION RESULT` **SUBQUERY**：如果包含子查询的查询语句不能够转为对应的`semi-join`的形式（即不能将子查询优化为多表连接），并且该子查询是不相关子查询。该子查询的第一个`SELECT`关键字代表的那个查询的`select_type`就是`SUBQUERY` **DEPENDENT SUBQUERY**：如果包含子查询的查询语句不能够转为对应的`semi-join`的形式，并且该子查询是相关子查询，则该子查询的第一个`SELECT`关键字代表的那个查询的`select_type`就是`DEPENDENT SUBQUERY` **DEPENDENT UNION**：在包含`UNION`或者`UNION ALL`的大查询中，如果各个小查询都依赖于外层查询的话，那除了最左边的那个小查询之外，其余的小查询的`select_type`的值就是`DEPENDENT UNION`。 **DERIVED**：对于包含`派生表`的查询，该派生表对应的子查询的`select_type`就是`DERIVED` **MATERIALIZED**：当查询优化器在执行包含子查询的语句时，选择将子查询物化之后与外层查询进行连接查询时，该子查询对应的`select_type`属性就是`MATERIALIZED`
table	表名，用到多少个表，就会有多少条记录 UNION去重，有一个临时表
partitions	匹配的分区信息，代表分区表中的命中情况，非分区表，该项为NULL。一般情况下我们的查询语句的执行计划的partitions列的值都是NULL。

列名	描述
type	**针对单表的访问方法**（重要） `system` ， `const` ， `eq_ref` ， `ref` ， `fulltext` ， `ref_or_null` ， `index_merge` ， `unique_subquery` ， `index_subquery` ， `range` ， `index` ， `ALL` 。越靠前越好 **system**：当表中`只有一条记录`并且该表使用的存储引擎的统计数据是精确的，比如MyISAM、Memory，那么对该表的访问方法就是`system`。 **const**：当我们根据主键或者唯一二级索引列与常数进行等值匹配时，对单表的访问方法就是`const` **eq_ref**：在连接查询时，如果被驱动表是通过主键或者唯一二级索引列等值匹配的方式进行访问的，（如果该主键或者唯一二级索引是联合索引的话，所有的索引列都必须进行等值比较），则对该被驱动表的访问方法就是`eq_ref` **ref**：当通过普通的二级索引列与常量进行等值匹配时来查询某个表，那么对该表的访问方法就可能是`ref` **fulltext**：全文索引 **ref_or_null**：当对普通二级索引进行等值匹配查询，该索引列的值也可以是`NULL`值时，那么对该表的访问方法就可能是`ref_or_null` **index_merge**：针对一张表，同时使用多个索引进行查询，然后将各个索引查出来的结果进行进一步的操作 **unique_subquery**：`unique_subquery`是针对在一些包含`IN`子查询的查询语句中，如果查询优化器决定将`IN`子查询转换为`EXISTS`子查询，而且子查询可以使用到主键进行等值匹配的话，那么该子查询执行计划的`type`列的值就是`unique_subquery`（子查询返回不重复集合） **index_subquery**：区别于unique_subquery，子查询使用到非唯一索引进行等值匹配，子查询会返回重复集合（等值匹配指的是比如WHERE key2 IN (SELECT id FROM....)这里id为主键，就用到了主键进行等值匹配） **range**：如果使用索引获取某些`范围区间`的记录，那么就可能使用到`range`访问方法 **index**：当我们可以使用索引覆盖，但需要扫描全部的索引记录时，该表的访问方法就是`index` **all**：全表扫描 至少要达到range级别，要求是ref，最好是const
possible_keys	可能用到的索引
key	**实际上使用的索引**
key_len	**实际使用到的索引长度**（字节数） key_len越小 索引效果越好 短一点效率更高 但是在联合索引里面，命中一个key_len加一次长度。越长代表精度越高，效果越好
ref	当使用索引列等值查询时，与索引列进行等值匹配的对象信息
rows	**预估的需要读取的记录条数**
filtered	某个表经过搜索条件过滤后剩余记录条数的百分比

列名	描述
Extra	一些额外的信息，更准确的理解MySQL到底将如何执行给定的查询语句 **No tables used**：当查询语句的没有FROM子句时将会提示该额外信息 **Impossible WHERE**： 查询语句的`WHERE`子句永远为`FALSE`时将会提示该额外信息 **Using where**：当我们使用全表扫描来执行对某个表的查询，并且该语句的`WHERE`子句中有针对该表的搜索条件时，在`Extra`列中会提示上述额外信息。当使用索引访问来执行对某个表的查询，并且该语句的`WHERE`子句中有除了该索引包含的列之外的其他搜索条件时，在`Extra`列中也会提示上述额外信息。 **Using index**：当我们的查询列表以及搜索条件中只包含属于某个索引的列，也就是在可以使用覆盖索引的情况下，在`Extra`列将会提示该额外信息。 **Using index condition**：使用了索引条件下推 **Using join buffer (Block Nested Loop)**： 没有索引的字段进行表关联。在连接查询执行过程中，当被驱动表不能有效的利用索引加快访问速度，MySQL一般会为其分配一块名叫`join buffer`的内存块来加快查询速度，也就是我们所讲的`基于块的嵌套循环算法` **Using filesort**：如果某个查询需要使用文件排序的方式执行查询，就会在执行计划的Extra列中显示`Using filesort`提示 **Using temporary**：在许多查询的执行过程中，MySQL可能会借助临时表来完成一些功能，比如去重、排序之类的，比如我们在执行许多包含`DISTINCT`、`GROUP BY`、`UNION`等子句的查询过程中，如果不能有效利用索引来完成查询，MySQL很有可能寻求通过建立内部的临时表来执行查询。如果查询中使用到了内部的临时表，在执行计划的`Extra`列将会显示`Using temporary`提示

EXPLAIN可以输出四种格式：`传统格式`， `JSON格式`， `TREE格式`以及`可视化输出`

使用完explain 后紧接着使用`SHOW WARNINGS \G`可以看到查询优化器真正执行的语句，粘出来并不一定可以运行

索引使用及优化与查询优化

- 索引失效、没有充分利用到索引——索引建立
- 关联查询太多JOIN (设计缺陷或不得已的需求)——SQL优化
- 服务器调优及各个参数设置(缓冲、线程数等)———调整my.cnf。
- 数据过多——分库分表

- **物理查询优化**是通过`索引`和`表连接方式`等技术来进行优化，这里重点需要掌握索引的使用。
- **逻辑查询优化**就是通过SQL`等价变换`提升查询效率，直白一点就是说，**换一种查询写法执行效率可能更高。**

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，索引将会部分失效(后面的字段索引失效)。

对于最左前缀法则指的是，查询时，**最左变的列必须存在，否则索引全部失效。 而且中间不能跳过某一列，否则该列后面的字段索引将失效。**

范围查询

联合索引中，出现范围查询(>,<)，**范围查询右侧的列索引失效。**

尽可能的使用类似于 >= 或 <= 这类的范围查询

索引失效情况

用不用索引，最终都是优化器说了算。优化器是基于什么的优化器?基于`cost开销(CostBaseOptimizer)`，它不是基于`规则(Rule-BasedOptimizer)`，也不是基于`语义`。**怎么样开销小就怎么来。**另外，**SQL语句是否使用索引，跟数据库版本、数据量、数据选择度都有关系。**

1.不要在**索引列上进行运算、函数操作**， 索引将失效。

2.字符串类型字段使用时，不加引号，**索引将失效。（存在隐式类型转换）**

3.如果仅仅是尾部模糊匹配，索引不会失效。如果是**头部模糊匹配，索引失效。**

4.当**or连接的条件，左右两侧字段都有索引时，索引才会生效。**

5.如果MySQL评估使用索引比全表更慢，则不使用索引。

6.**不等于**(!= 或者<>)索引失效 （索引只能查到知道的东西）

7.is null可以使用索引，**is not null**无法使用索引

8.**not like** 也无法使用索引

9.order by 时不limit，索引可能失效，增加limit 减少回表的数量，优化器觉得走索引快，会使用索引

10.order by时规则不一致,索引失效（方向反，不索引）

其实失不失效还得看具体成本情况，都说不准

SQL优化

多个字段都创建索引的话创建联合索引最好

避免这样无谓的性能损耗，最好让插入的记录的 主键值依次递增

插入数据优化

如果我们需要一次性往数据库表中插入多条记录，可以从以下三个方面进行优化。

批量插入数据 手动控制事务 主键顺序插入，性能要高于乱序插入。

如果一次性需要插入大批量数据(比如: 几百万的记录)，使用insert语句插入性能较低，此时可以使 用MySQL数据库提供的**load指令**进行插入。

主键优化

满足业务需求的情况下，尽量降低主键的长度。

插入数据时，尽量选择顺序插入，选择使用AUTO_INCREMENT自增主键

尽量不要使用UUID做主键或者是其他自然主键，如身份证号。

业务操作时，避免对主键的修改。

关联查询优化

LEFT JOIN条件用于确定如何从右表搜索行，左边一定都有，所以 右边是我们的关键点,一定需要建立索引 。如果只能添加一边的索引，，那就给 `被驱动表` 添加上索引。

explain中驱动表在上，被驱动表在下

`内连接` 主被驱动表是由优化器决定的。优化器认为哪个成本比较小，就采用哪种作为驱动表。

如果两张表只有一个有索引，那有索引的表作为`被驱动表`。

　原因：驱动表要全查出来。有没有索引你都得全查出来。

两个索引都存在的情况下， 数据量大的 作为`被驱动表`（小表驱动大表）

　原因：驱动表要全部查出来，而大表可以通过索引加快查找

join语句原理：

join方式连接多个表，本质就是各个表之间数据的循环匹配。

Simple Nested-Loop Join(简单嵌套循环连接)

Index Nested-Loop Join(索引嵌套循环连接)

Block Nested-Loop Join(块嵌套循环连接)

整体效率比较:INLJ > BNLJ > SNLJ

永远用小结果集驱动大结果集(其本质就是减少外层循环的数据数量)

为被驱动表匹配的条件增加索引(减少内层表的循环匹配次数)

增大join buffer size的大小(一次缓存的数据越多，那么内层包的扫表次数就越少)

减少`驱动表`不必要的字段查询（字段越少，join buffer 所缓存的数据就越多)

- 保证被驱动表的JOIN字段已经创建了索引
- 需要JOIN 的字段，数据类型保持绝对一致。
- LEFT JOIN 时，选择小表作为驱动表， `大表作为被驱动表` 。减少外层循环的次数。
- INNER JOIN 时，MySQL会自动将 `小结果集的表选为驱动表` 。选择相信MySQL优化策略。

- 能够直接多表关联的尽量直接关联，不用子查询。(减少查询的趟数)
- 不建议使用子查询，建议将子查询SQL拆开结合程序多次查询，或使用 JOIN 来代替子查询。
- 衍生表建不了索引

从MySQL的8.0.20版本开始将废弃BNLJ，因为从MySQL8.0.18版本开始就加入了hash join默认都会使用hash join

子查询优化

子查询的执行效率不高。原因:

①执行子查询时MySQL**需要为内层查询语句的查询结果**`建立一个临时表`，然后外层查询语句从临时表中查询记录。查询完毕后，再`撤销这些临时表`。这样会消耗过多的CPU和IO资源，产生大量的慢查询。

②子查询的结果集存储的临时表，不论是内存临时表还是磁盘临时表都`不会存在索引`，所以查询性能会受到一定的影响。

③对于返回结果集比较大的子查询，其对查询性能的影响也就越大。

在MySQL中，可以使用连接（JOIN）查询来替代子查询。连接查询不需要`建立临时表`，其`速度比子查询要快`，如果查询中使用索引的话，性能就会更好。

尽量不要使用NOT IN或者NOT EXISTS，用LEFT JOIN Xxx ON xx WHERE xx IS NULL替代

排序优化

在MySQL中，支持两种排序方式，分别是`FileSort`和`Index`排序。

- Index排序中，索引可以保证数据的有序性，不需要再进行排序，`效率更高`。
- FileSort排序则一般在`内存中`进行排序，占用`CPU较多`。如果待排结果较大，会产生临时文件I/O到磁盘进行排序的情况，效率较低。

1. SQL中，可以在WHERE子句和ORDER BY子句中使用索引，目的是在WHERE子句中`避免全表扫描`，在ORDER BY子句`避免使用FileSort排序`。当然，某些情况下全表扫描，或者FileSort排序不一定比索引慢。但总的来说，我们还是要避免，以提高查询效率。
2. 尽量使用Index完成ORDER BY排序。如果WHERE和ORDER BY后面是相同的列就使用单索引列;如果不同就使用联合索引。索引只会用到一个，没办法一个索引用来where 一个索引用来 order by。但是可以建立联合索引。
3. 无法使用Index时，需要对FileSort方式进行调优。

A. 根据排序字段建立合适的索引，多字段排序时，也遵循最左前缀法则。

B. 尽量使用覆盖索引。

C. 多字段排序, 一个升序一个降序，此时需要注意联合索引在创建时的规则（ASC/DESC）。

D. 如果不可避免的出现filesort，大数据量排序时，可以适当增大排序缓冲区大小 sort_buffer_size(默认256k)。

GROUP BY优化

- group by使用索引的原则几乎跟order by一致，group by即使没有过滤条件用到索引，也可以直接使用索引。.
- **group by先排序再分组**，遵照索引建的最佳左前缀法则
- 当无法使用索引列，增大`max_length_for_sort_data`和`sort_buffer_size`参数的设置
- where效率高于having，能写在where限定的条件就不要写在having中了
- 减少使用order by，和业务沟通能不排序就不排序，或将排序放到程序端去做
- Order by、group by、distinct这些语句较为耗费CPU，数据库的CPU资源是极其宝贵的。
- 包含了order by、group by、distinct这些查询的语句，where条件过滤出来的结果集请保持在1000行以内，否则SQL会很慢。

A. 在分组操作时，可以通过索引来提高效率。

B. 分组操作时，索引的使用也是满足最左前缀法则的。

分页查询优化

一般分页查询时，通过创建覆盖索引能够比较好地提高性能。一个常见又非常头疼的问题就是limit 2000000,10，此时需要MySQL排序前2000010记录，仅仅返回2000000 - 2000010的记录，其他记录丢弃，查询排序的代价非常大。

优化思路一
在索引上完成排序分页操作，最后根据主键关联回原表查询所需要的其他列内容。（覆盖索引+子查询）

如：explain select * from tb_sku t , (select id from tb_sku order by id limit 2000000,10) a where t.id = a.id;

SQL提示

在SQL语句中加入一些人为的提示来达到优 化操作的目的。

use index

ignore index

force index

覆盖索引

尽量使用覆盖索引，减少select *。 那么什么是覆盖索引呢？ 覆盖索引是指 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 。

其它优化：

普通索引和唯一索引这两类索引在查询能力上是没差别的，主要考虑的是
对 更新性能 的影响。所以，建议你 尽量选择普通索引

唯一索引用不上change buffer的优化机制

普通索引 和 change buffer 的配合使用，对于 数据量大 的表的更新优化还是很明显的。

如果所有的更新后面，都马上 伴随着对这个记录的查询 ，那么你应该 关闭change buffer 。而在其他情况下，change buffer都能提升更新性能。

count优化：

用法：count（*）、count（主键）、count（字段）、count（数字）

InnoDB 引擎就麻烦了，它执行 count(*) 的时候，需要把数据一行一行地从引擎里面读出 来，然后累积计数。

按照效率排序的话，count(字段) < count(主键 id) < count(1) ≈ count(*)，所以尽 量使用 count(**)。

update优化：

最好根据索引字段进行更新

InnoDB的行锁是针对索引加的锁，不是针对记录加的锁 ,并且该索引不能失效，否则会从行锁 升级为表锁 。

EXISTS 和 IN 的区分：

选择与否还是要看表的大小。你可以将选择的标准理解为`小表驱动大表`。在这种方式下效率是最高的。

比如下面这样:

```
SELECT *FROM A WHERE cc IN (SELECT cc FROM B)

SELECT *FROM A WHERE EXISTS (SELECT cc FROM B WHERE B.cc=A.cc)
```

当A小于B时，用EXISTS。因为EXISTS的实现，相当于外表循环，实现的逻辑类似于:

```
for i in A
    for j in B
        if j.cc == i.cc then ...
```

当B小于A时用IN，因为实现的逻辑类似于:

```
for i in B
    for j in A
        if j.cc == i.cc then ...
```

哪个表小就用哪个表来驱动，A表小就用EXISTS，B表小就用IN。

关于SELECT(*)

在表查询中，建议明确字段，不要使用 * 作为查询的字段列表，推荐使用SELECT <字段列表> 查询。原因：

① MySQL 在解析的过程中，会通过 `查询数据字典` 将"*"按序转换成所有列名，这会大大的耗费资源和时间。

② 无法使用 `覆盖索引`

LIMIT 1 对优化的影响

针对的是会扫描全表的 SQL 语句，如果你可以确定结果集只有一条，那么加上 LIMIT 1 的时候，当找到一条结果的时候就不会继续扫描了，这样会加快查询速度。

如果数据表已经对字段建立了唯一索引，那么可以通过索引进行查询，不会全表扫描的话，就不需要加上 `LIMIT 1` 了。

多使用COMMIT

只要有可能，在程序中尽量多使用 COMMIT，这样程序的性能得到提高，需求也会因为 COMMIT 所释放的资源而减少。

COMMIT 所释放的资源：

- 回滚段上用于恢复数据的信息
- 被程序语句获得的锁
- redo / undo log buffer 中的空间
- 管理上述 3 种资源中的内部花费

推荐的主键设计

`非核心业务`：对应表的主键自增ID，如告警、日志、监控等信息。

`核心业务`：主键设计至少应该是全局唯一且是单调递增。全局唯一保证在各系统之间都是唯一的，单调递增是希望插入时不影响数据库性能。

UUID

索引条件下推

Index Condition Pushdown(ICP)是MySQL 5.6中新特性，是一种**在存储引擎层使用索引过滤数据的一种优化方式。**

数据库设计规范

范式

在关系型数据库中，关于**数据表设计的基本原则、规则就称为范式**

关系型数据库有六种常见范式，按照范式级别，从低到高分别是：**第一范式（1NF）、第二范式（2NF）、第三范式（3NF）、巴斯-科德范式（BCNF）、第四范式(4NF）和第五范式（5NF，又称完美范式）**

一般关系型数据库的设计中，**最高也就BCNF，一般为3NF**，但也不绝对，有时为了提高某些查询性能，需要破坏范式规则，反范式化

高阶范式一定符合低阶范式的要求

范式的定义会使用到主键和候选键，数据库中的键(Key)由一个或者多个属性组成。数据表中常用的几种键和属性的定义:

- 超键：能**唯一标识**元组的属性集叫做超键。
- 候选键：就是最小的超键，如果超键不包括多余的属性，那么这个超键就是候选键。·主键:用户可以从**候选键中选择一个作为主键。**（只有一个属性的超键）
- 外键：如果数据表R1中的某属性集不是R1的主键，而是另一个数据表R2的主键，那么这个属性集就是数据表R1的外键。
- 主属性:包含在任一**候选键中的属性**称为主属性。
- 非主属性:与主属性相对，指的是不包含在任何一个候选键中的属性。

第一范式（1NF）：

第一范式主要是确保数据表中每个字段的值必须具有原子性，也就是说数据表中**每个字段的值为**不可再次拆分**的最小数据单元。**

比如表中有个字段为user_info，包含了用户各种信息，这个各种信息是可拆分的，所以不满足1NF

第二范式（2NF）：

第二范式要求，在满足第一范式的基础上，还要满足数据表里的每一条数据记录，都是可唯一标识的。而且**所有非主键字段，都必须完全依赖主键，不能只依赖主键的一部分。**如果知道主键的所有属性的值，就可以检索到任何元组(行)的任何属性的任何值。(要求中的主键，其实可以拓展替换为候选键)。

如果存在不完全依赖，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体（这个属性完全依赖主关键字的这一部分），新实体与元实体之间是一对多的关系。

1NF 告诉我们字段属性需要是原子性的，而 2NF 告诉我们一张表就是一个独立的对象，一张表只表达一个意思

第三范式（3NF）：

第三范式是在第二范式的基础上，确保数据表中的**每一个非主键字段都和主键字段直接相关**，也就是说，要求数据表中的**所有非主键字段不能依赖于其他非主键字段。**(即，不能存在非主属性A依赖于非王属性B，非士属任B依赖于主键C的情况，即存在"A→B一C""的决定关系）通俗地讲，该规则的意思是**所有**非主键属性**之间不能有依赖关系，必须**相互独立**。**

这里的主键可以拓展为候选键。

范式的优点: 数据的标准化有助于**消除数据库中的**数据冗余，第三范式(3NF）通常被认为在性能、扩展性和数据完整性方面达到了最好的平衡。

范式的缺点: 范式的使用，**可能**降低查询的效率。因为**范式等级越高，设计出来的数据表就越多**、越精细，数据的冗余度就越低，进行数据查询的时候就可能需要关联多张表，这不但代价昂贵，也可能使一些索引策略无效。

范式只是提出了设计的标准，实际上设计数据表时，未必一定要符合这些标准。开发中，我们会出现为了性能和读取效率违反范式化的原则，通过`增加少量的冗余`或重复的数据来提高数据库的`读性能`，减少关联查询,join表的次数，实现空间换取时间的目的。因此在实际的设计过程中要理论结合实际，灵活运用。

巴斯-科德范式（BCNF）：

主属性（仓库名）对于候选键（管理员，物品名）是部分依赖的关系，管理员就能决定仓库名，这样就有可能导致异常情况。因此引入BCNF，**它在 3NF 的基础上消除了主属性对候选键的部分依赖或者传递依赖关系。**

每个属性都不部分依赖于候选键也不传递依赖于候选键

消除了任何属性对主键的部份依赖和传递依赖

第四范式(4NF)：

第五范式（5NF，又称完美范式）：

反范式化

有的数据看似冗余，其实对业务来说十分重要

有时候如果**完全按照范式设计数据表，查询时会产生大量的关联查询，影响性能**，此时反范式化也是一种优化思路，通过**增加冗余字段来提高数据库的读性能**

规范化 vs 性能

1. 为满足某种商业目标，数据库性能比规范化数据库更重要
2. 在数据规范化的同时，要综合考虑数据库的性能
3. 通过在给定的表中添加额外的字段，以大量减少需要从中搜索信息所需的时间
4. 通过在给定的表中插入计算列，以方便查询

反范式的新问题

- 存储`空间变大`了
- 一个表中字段做了修改，另一个表中冗余的字段也需要做同步修改，否则`数据不一致`
- 若采用存储过程来支持数据的更新、删除等额外操作，如果更新频繁，会非常`消耗系统资源`
- 在`数据量小`的情况下，反范式不能体现性能的优势，可能还会让数据库的设计更加`复杂`

当冗余信息有价值或者能`大幅度提高查询效率`的时候，我们才会采取反范式的优化。

比如employee表和department表通过department_id进行关联，如果要经常查询员工所在部门名称，则干脆在employee表中添加department_name字段，这样就不用每次都连接查询了

增加冗余字段的建议

1）这个冗余字段`不需要经常进行修改`

2）这个冗余字段`查询的时候不可或缺`

数据表的设计原则

数据表设计的一般原则："三少一多"

1. 数据表的个数越少越好

2. 数据表中的字段个数越少越好

3. 数据表中联合主键的字段个数越少越好

4. 使用主键和外键越多越好

> 注意：这个原则并不是绝对的，有时候我们需要牺牲数据的冗余度来换取数据处理的效率。

数据库其它调优策略

如何定位调优问题

一般情况下，有如下几种方式：

用户的反馈（主要）

日志分析（主要）

服务器资源使用监控

数据库内部状况监控

其它 除了活动会话监控以外，我们也可以对 事务 、 锁等待 等进行监控，这些都可以帮助我们对数据库的运行状态有更全面的认识。

调优的维度和步骤

选择适合的DBMS

优化表设计

优化逻辑查询

优化物理查询

使用Redis或 Memcached作为缓存

库级优化

读写分离

数据分片

对数据库分库分表。当数据量级达到千万级以上时，有时候我们需要把一个数据库切成多份，放到不同的数据库服务器上，减少对单一数据库服务器的访问压力。如果你使用的是MySQL，就可以使用MySQL自带的分区表功能，当然你也可以考虑自己做`垂直拆分（分库）`、`水平拆分`（分表）、`垂直+水平拆分（分库分表）`。

优化MySQL服务器

优化MySQL服务器主要从两个方面来优化，一方面是对`硬件`进行优化;另一方面是对MySQL`服务的参数`进行优化。这部分的内容需要较全面的知识，一般只有`专业的数据库管理员`才能进行这一类的优化。对于可以定制参数的操作系统，也可以针对MySQL进行操作系统优化。

优化数据库结构

拆分表：冷热数据分离

增加中间表

增加冗余字段

优化数据类型

情况1：对整数类型数据进行优化。

遇到整数类型的字段可以用`INT 型`。这样做的理由是，INT 型数据有足够大的取值范围，不用担心数据超出取值范围的问题。刚开始做项目的时候，首先要保证系统的稳定性，这样设计字段类型是可以的。但在数据量很大的时候，数据类型的定义，在很大程度上会影响到系统整体的执行效率。

对于`非负型`的数据（如自增ID、整型IP）来说，要优先使用无符号整型`UNSIGNED`来存储。因为无符号相对于有符号，同样的字节数，存储的数值范围更大。如tinyint有符号为-128-127，无符号为0-255，多出一倍的存储空间。

情况2：既可以使用文本类型也可以使用整数类型的字段，要选择使用整数类型。

跟文本类型数据相比，大整数往往占用`更少的存储空间`，因此，在存取和比对的时候，可以占用更少的内存空间。所以，在二者皆可用的情况下，尽量使用整数类型，这样可以提高查询的效率。如：将IP地址转换成整型数据。

情况3：避免使用TEXT、BLOB数据类型

情况4：避免使用ENUM类型

情况5：使用TIMESTAMP存储时间

情况6：用DECIMAL代替FLOAT和DOUBLE存储精确浮点数

优化插入记录的速度

InnoDB引擎的表：

① 禁用唯一性检查

② 禁用外键检查

③ 禁止自动提交

在设计字段的时候，如果业务允许，建议尽量使用非空约束

结合实际的业务需求进行权衡

服务器语句超时处理：在MySQL 8.0中可以设置 服务器语句超时的限制 ，单位可以达到 毫秒级别 。

事务

事务：一组逻辑操作单元，使数据从一种状态变换到另一种状态。

原子性（atomicity）：

原子性是指事务是一个不可分割的工作单位，要么全部提交，要么全部失败回滚。

一致性（consistency）：

一致性是指**事务执行前后，数据从一个** `合法性状态` **变换到另外一个** `合法性状态` **。**这种状态是 `语义上` 的而不是语法上的，跟具体的业务有关。

比如银行账户转账的例子，两个账户一共800元，互相转账后，总额还会是800元

隔离型（isolation）：

事务的隔离性是指**一个事务的执行** `不能被其他事务干扰` ，即一个事务内部的操作及使用的数据对 `并发` 的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

持久性（durability）：

持久性是指一个**事务一旦被提交，它对数据库中数据的改变就是** `永久性的` ，接下来的其他操作和数据库故障不应该对其有任何影响。

事物的状态

部分提交的（partially committed）

当事务中的最后一个操作执行完成，但由于操作都在内存中执行，所造成的影响并 `没有刷新到磁盘` 时，我们就说该事务处在 `部分提交的` 状态。

提交的（committed）

当一个处在 `部分提交的` 状态的事务将修改过的数据都 `同步到磁盘` 上之后，我们就可以说该事务处在了 `提交的` 状态。

中止的（aborted）

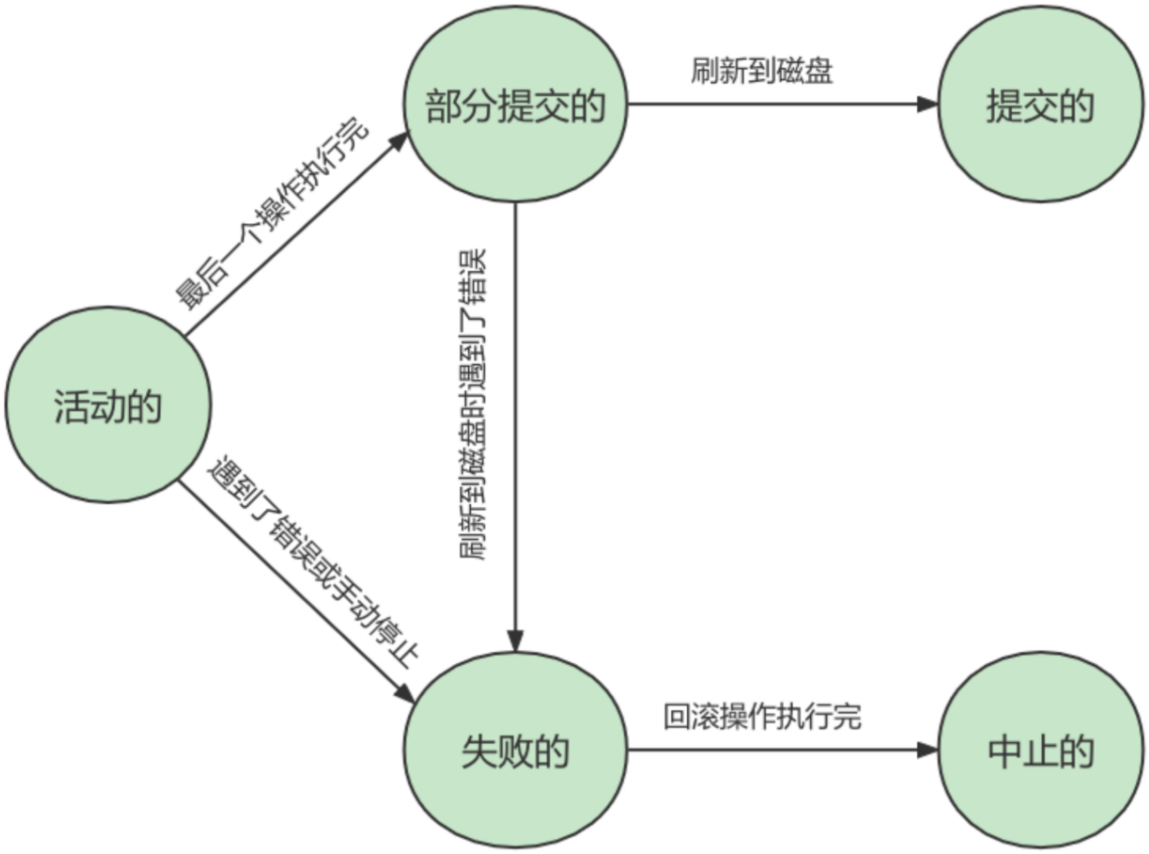
当 `回滚` 操作执行完毕时，也就是数据库恢复到了执行事务之前的状态，我们就说该事务处在了 `中止的` 状态。

活动的（active）

事务对应的数据库操作正在执行过程中时，我们就说该事务处在 `活动的` 状态。

失败的（failed）

当事务处在 `活动的` 或者 `部分提交的` 状态时，可能遇到了某些错误（数据库自身的错误、操作系统错误或者直接断电等）而无法继续执行，或者人为的停止当前事务的执行，我们就说该事务处在 `失败的` 状态。



使用事务

显式事务

mysql> BEGIN;
mysql> START TRANSACTION;后边能跟随几个修饰符

① READ ONLY： 标识当前事务是一个 只读事务 ， 也就是属于该事务的数据库操作只能读取数据，而不能修改数据。
② READ WRITE： 标识当前事务是一个 读写事务 ， 也就是属于该事务的数据库操作既可以读取数据，也可以修改数据。
③ WITH CONSISTENT SNAPSHOT： 启动一致性读。

数据库系统会创建一个事务性的快照或视图，保证在整个读取过程中，读取的数据保持一致，不会受到其他并发事务的修改影响。

mysql> COMMIT;

mysql> ROLLBACK;

mysql> ROLLBACK TO [SAVEPOINT]

隐式事务

set autocommit = 0

事务隔离级别

脏写（ Dirty Write ）

对于两个事务 Session A、Session B，如果事务Session A 修改了 另一个 未提交 事务Session B 修改过 的数据，那就意味着发生了 脏写

脏读（ Dirty Read ）

对于两个事务 Session A、Session B，Session A 读取 了已经被 Session B 更新 但还 没有被提交 的字段。之后若 Session B 回滚 ， Session A 读取 的内容就是 临时且无效 的。
Session A和Session B各开启了一个事务，Session B中的事务先将studentno列为1的记录的name列更新为'张三'，然后Session A中的事务再去查询这条studentno为1的记录，如果读到列name的值为'张三'，而Session B中的事务稍后进行了回滚，那么Session A中的事务相当于读到了一个不存在的数据，这种现象就称之为 脏读 。

不可重复读（ Non-Repeatable Read ）

对于两个事务Session A、Session B，Session A 读取 了一个字段，然后 Session B 更新 了该字段。 之后 Session A 再次读取 同一个字段， 值就不同 了。那就意味着发生了不可重复读。
我们在Session B中提交了几个 隐式事务 （注意是隐式事务，意味着语句结束事务就提交了），这些事务都修改了studentno列为1的记录的列name的值，每次事务提交之后，如果Session A中的事务都可以查看到最新的值，这种现象也被称之为 不可重复读 。

幻读（ Phantom ）

对于两个事务Session A、Session B, Session A 从一个表中 读取 了一个字段, 然后 Session B 在该表中 插入 了一些新的行。 之后, 如果 Session A 再次读取 同一个表, 就会多出几行。那就意味着发生了幻读。

Session A中的事务先根据条件 studentno > 0这个条件查询表student，得到了name列值为'张三'的记录；之后Session B中提交了一个 隐式事务 ，该事务向表student中插入了一条新记录；之后Session A中的事务再根据相同的条件 studentno > 0查询表student，得到的结果集中包含Session B中的事务新插入的那条记录，这种现象也被称之为 幻读 。（session A一直未commit）我们把新插入的那些记录称之为 幻影记录 。

严重程度：脏写 > 脏读 > 不可重复读 > 幻读

隔离级别	脏读可能性	不可重复读可能性	幻读可能性	加锁读
READ UNCONMITED	Yes	Yes	Yes	No
READ COMMITED	No	Yes	Yes	No
REPEATABLE READ	No	No	Yes	No
SERIALIZABLE	No	No	No	Yes

脏写这个问题太严重了，不论是哪种隔离级别，都不允许脏写的情况发生。

不同的隔离级别有不同的现象，并有不同的锁和并发机制，隔离级别越高，数据库的并发性能就越差

MySQL的默认隔离级别为REPEATABLE READ（解决了幻读）

SELECT @@transaction_isolation;

SET [GLOBAL|SESSION] TRANSACTION_ISOLATION = '隔离级别'

事务日志

事务的隔离性由 `锁机制` 实现。

而**事务的原子性、一致性和持久性由事务的 redo 日志和undo 日志来保证。**

- REDO LOG 称为 `重做日志` ，提供再写入操作，恢复提交事务修改的页操作，用来保证事务的**持久性。**
- UNDO LOG 称为 `回滚日志` ，回滚行记录到某个特定版本，用来保证事务的**原子性、一致性。**

REDO和UNDO都可以视为是一种 `恢复操作`

- redo log:是**存储引擎层(innodb)生成的日志**，记录的是"`物理级别`"上的页修改操作，比如页号xxx、偏移量yyy写入了'zzz'数据。主要为了保证数据的可靠性;

  > 提交，由redo log来保证事务的持久化。

- undo log:是**存储引擎层(innodb)生成的日志**，记录的是 `逻辑操作` 日志，比如对某一行数据进行了INSERT语句操作，那么undo log就记录一条与之相反的DELETE操作。主要用于 `事务的回滚` **(undo log记录的是每个修改操作的** `逆操作` **)和** `一致性非锁定读` **(undo log回滚行记录到某种特定的版本---MVCC**，即多版本并发控制)。

1. **REDO LOG（重做日志）：**
   - Redo Log是数据库系统中的一种日志记录，它**记录了对数据库进行的修改操作，如INSERT、UPDATE、DELETE等。**它的作用是用于保**障数据库的持久性。**
   - 在**事务提交时，数据库会首先将事务所做的修改操作写入Redo Log中**，然后再将数据写入数据库文件。这样，即使在写入数据库文件过程中发生故障，数据库可以通过Redo Log中的信息来重新执行事务的修改操作，从而确保数据库的一致性和完整性。
   - Redo Log通常是循环写入的，一旦Redo Log的空间用尽，旧的Redo Log记录可能会被覆盖。数据库系统会根据需要将Redo Log中的数据同步到磁盘，以保证数据的持久性。
2. **UNDO LOG（回滚日志）：**
   - Undo Log也是数据库系统中的一种日志记录，它记录了事务的修改操作的撤销信息。它的作用是**用于事务的回滚和MVCC（多版本并发控制）的实现。**
   - **在事务执行过程中，当某个事务更新了数据库中的数据，数据库会将原始数据备份到Undo Log中，保留了修改前的数据版本。如果该事务需要回滚，或者其他事务需要读取旧版本的数据（MVCC），数据库可以通过Undo Log中的信息将数据还原到之前的状态。**
   - Undo Log通常在事务提交后才会释放空间，因为只有在事务提交后，对应的修改操作才能确定不需要撤销。因此，Undo Log的存在也可以确保数据库的一致性和隔离性。

REDO LOG

InnoDB存储引擎是以 `页为单位` 来管理存储空间的。在**真正访问页面之前需要把在** `磁盘上` **的页缓存到内存中的** `Buffer Pool` **之后才可以访问。**所有的变更都必须 `先更新缓冲池中` 的数据，然后缓冲池中的 `脏页` 会以一定的频率被刷入磁盘（ `checkPoint` 机制），通过缓冲池来优化CPU和磁盘之间的鸿沟，这样就可以保证整体的性能不会下降太快。

InnoDB引擎的事务采用了WAL技术(`Write-Ahead Logging`)，这种技术的思想就是**先写日志，再写磁盘，只有日志写入成功，才算事务提交成功**，这里的日志就是redo log。**当发生宕机且数据未刷到磁盘的时候，可以通过redo log来恢复**，保证ACID中的D，这就是redo log的作用。

某个事务将系统表空间中 `第10号` 页面中偏移量为 `100` 处的那个字节的值 `1` 改成 `2` 。我们只需要记录一下：将第0号表空间的10号页面的偏移量为100处的值更新为 2 。

REDO日志的好处、特点

1. 好处

- **redo日志降低了刷盘频率**

  存储表空间ID、页号、偏移量以及需要更新的值，**所需的存储空间是很小的**，刷盘快。

2. 特点

- **redo日志是顺序写入磁盘的**

  在执行事务的过程中，每执行一条语句，就可能产生若干条redo日志，这些日志是按照产生的`顺序写入磁盘`的，也就是**使用顺序IO**，效率比随机IO快。

- **事务执行过程中，redo log不断记录**

  redo log跟bin log的区别，redo log是`存储引擎层`产生的，而bin log是`数据库层`广生的。假设一个事务，对表做10万行的记录插入，在这个过程中，一直不断的往redo log顺序记录，而bin log不会记录，直到这个事务提交，才会一次写入到bin log文件中。

  Binlog（二进制日志）是MySQL数据库中的一种日志记录机制，它用于记录数据库的修改操作，包括INSERT、UPDATE、DELETE等数据变更操作。Binlog是以二进制格式记录的，因此称为"二进制日志"。

redo的组成

Redo log可以简单分为以下两个部分：

- `重做日志的缓冲（redo log buffer）`，**保存在内存中，是易失的。**

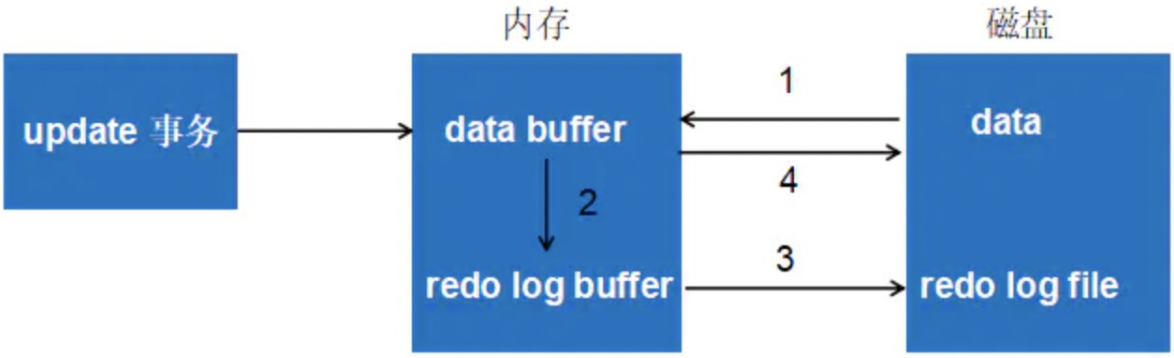
  在服务器启动时就向操作系统申请了一大片称之为redo log buffer的`连续内存`空间，翻译成中文就是redo日志缓冲区。这片内存空间被划分成若干个连续的`redo log block`。一个redo log block占用`512字节`大小。

  参数设置：innodb_log_buffer_size：

  redo log buffer 大小，默认 `16M` ，最大值是4096M，最小值为1M。

- `重做日志文件(redologfile)`，**保存在硬盘中，是持久的。**

  REDO日志文件如图所示，其中的`ib_logfile0`和`ib_logfile1`即为redo log日志。



第1步：先将原始数据从磁盘中读入内存中来，修改数据的内存拷贝
第2步：生成一条重做日志并写入redo log buffer，记录的是数据被修改后的值
第3步：当事务commit时，将redo log buffer中的内容刷新到 redo log file，对 redo log file采用追加写的方式
第4步：定期将内存中修改的数据刷新到磁盘中

redo log的刷盘策略

redo log的写入并不是直接写入磁盘的，InnoDB引擎会在写redo log的时候先写redo log buffer，之后以`一定的频率刷`入到真正的redo log file 中。这里的一定频率怎么看待呢？这就是我们要说的刷盘策略。

redo log buffer刷盘到redo log file的过程并不是真正的刷到磁盘中去，只是刷入到`文件系统缓存`（page cache）中去（这是现代操作系统为了提高文件写入效率做的一个优化），**真正的写入会交给系统自己来决定（比如page cache足够大了）**。那么对于InnoDB来说就存在一个问题，如果交给系统来同步，同样如果系统宕机，那么数据也丢失了（虽然整个系统宕机的概率还是比较小的）。

针对这种情况，InnoDB给出`innodb_flush_log_at_trx_commit`参数，该参数控制 commit提交事务时，如何将 redo log buffer 中的日志刷新到 redo log file 中。它支持三种策略：

- `设置为0`：表示每次事务提交时不进行刷盘操作。（系统默认master thread每隔1s进行一次重做日志的同步）

  InnoDB存储引擎有一个后台线程，每隔1秒，就会把`redo log buffer中`的内容写到文件系统缓存(`page cache`)，然后调用刷盘操作。

  也就是说，一个没有提交事务的`redo log`记录，也可能会刷盘。因为在事务执行过程redo log记录是会写入`redo log buffer`中，这些redo log记录会被`后台线程`刷盘。

  后台线程每秒`1次`的轮询操作，还有一种情况，当`redo log buffer`占用的空间即将达到`innodb_log_buffer_size`(这个参数默认是16M)的一半的时候，后台线程会主动刷盘。

- `设置为1`：表示每次事务提交时都将进行同步，刷盘操作（`默认值`）
- `设置为2`：表示每次事务提交时都只把 redo log buffer 内容写入`page cache`，不进行同步。由os自己决定什么时候同步到磁盘文件。

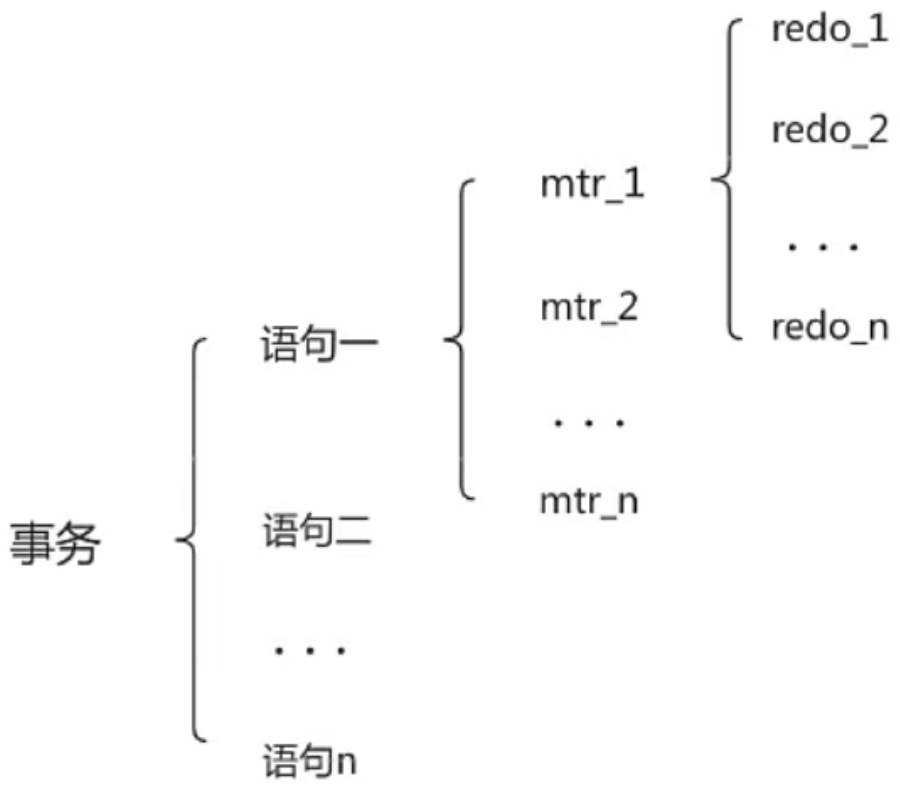
  为2时，只要事务提交成功,`redo log buffer`中的内容只写入文件系统缓存（page cache）。

  如果仅仅只是`MySQL`挂了不会有任何数据丢失，但是操作系统宕机可能会有`1`秒数据的丢失

Mini-Transaction

MySQL把对底层页面中的一次原子访问的过程称之为一个`Mini-Transaction`，简称`mtr`，比如，向某个索引对应的B+树中插入一条记录的过程就是一个`Mini-Transaction`。一个所谓的`mtr`可以包含一组redo日志，在进行崩溃恢复时这一组`redo`日志作为一个不可分割的整体。

一个事务可以包含若干条语句，每一条语句其实是由若干个 `mtr` 组成，每一个 `mtr` 又可以包含若干条redo日志，画个图表示它们的关系就是这样：



一个mtr执行过程中可能产生若干条redo日志，`这些redo日志是一个不可分割的组`，所以其实并不是每生成一条redo日志，就将其插入到log buffer中，而是每个mtr运行过程中产生的日志先暂时存到一个地方，当该mtr结束的时候，将过程中产生的一组redo日志再全部复制到log buffer中。

UNDO LOG

redo log是事务持久性的保证，undo log是事务原子性的保证。在事务中 `更新数据` 的 `前置操作` 其实是要先写入一个 `undo log` 。

每当我们要对一条记录做改动时(这里的`改动`可以指`INSERT`、`DELETE`、`UPDATE`)，都需要"留一手"——把回滚时所需的东西记下来。比如:

- 你`插入一条记录时`，至少要把这条记录的主键值记下来，之后回滚的时候只需要把这个主键值对应的`记录删掉`就好了。(对于每个INSERT, InnoDB存储引擎会完成一个DELETE)
- 你`删除了一条记录`，至少要把这条记录中的内容都记下来，这样之后回滚时再把由这些内容组成的记录插入到表中就好了。(对于每个DELETE,InnoDB存储引擎会执行一个INSERT)
- 你`修改了一条记录`，至少要把修改这条记录前的旧值都记录下来，这样之后回滚时再把这条记录`更新为旧值`就好了。(对于每个UPDATE，InnoDB存储引擎会执行一个相反的UPDATE，将修改前的行放回去)

MySQL把这些为了回滚而记录的这些内容称之为撤销日志或者回滚日志(即undo log)。注意，由于查询操作(SELECT)并不会修改任何用户记录，所以在查询操作行时，并不需要记录相应的undo日志

此外，undo log 会产生redo log，也就是undo log的产生会伴随着redo log的产生，这是因为undo log也需要持久性的保护

锁

锁是计算机协调多个进程或线程并发访问某一资源的机制。

并发事务访问相同记录的情况大致可以划分为 3 种：

读读

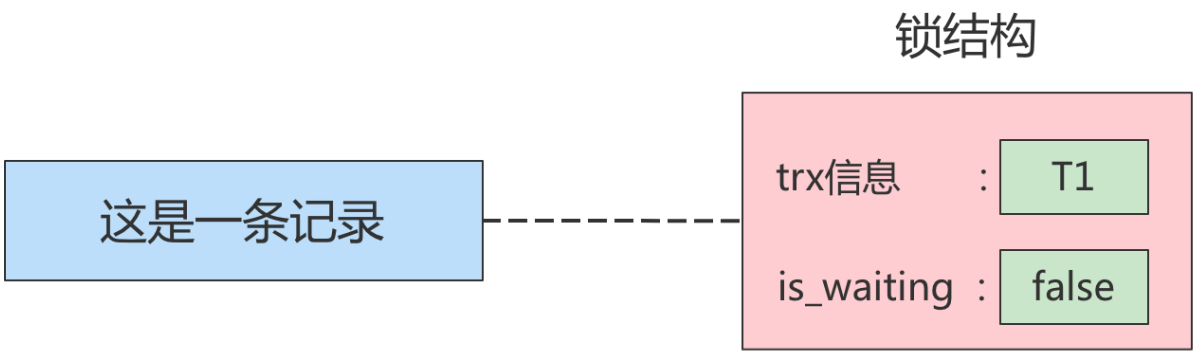
并不会引起什么问题，所以允许这种情况的发生。

写写

会发生脏写的问题，任何一种隔离级别都不允许这种问题的发生。所以在多个未提交事务相继对一条记录做改动时，需要让它们排队执行

排队的过程其实是通过锁来实现的。这个所谓的锁其实是一个内存中的结构

当一个事务想对这条记录做改动时，首先会看看内存中有没有与这条记录关联的锁结构，当没有的时候就会在内存中生成一个锁结构与之关联。比如，事务T1要对这条记录做改动，就需要生成一个锁结构与之关联：



在锁结构里有很多信息，为了简化理解，只把两个比较重要的属性拿了出来:

- trx信息:代表这个**锁结构是哪个事务生成的。**
- is_waiting :代表**当前事务是否在等待。**

当事务T1改动了这条记录后，就生成了一个锁结构与该记录关联，因为之前没有别的事务为这条记录加锁，所以is_waiting属性就是false，我们把这个场景就称之为获取锁成功，或者加锁成功，然后就可以继续执行操作了。

在事务T1提交之前，另一个事务T2也想对该记录做改动，那么先看看有没有锁结构与这条记录关联，发现有一个锁结构与之关联后，然后也生成了一个锁结构与这条记录关联，不过锁结构的is_waiting属性值为true,表示当前事务需要等待，我们把这个场景就称之为获取锁失败，或者加锁失败

在事务T1提交之后，就会把该事务生成的`锁结构释放`掉，然后看看还有没有别的事务在等待获取锁，发现了事务T2还在等待获取锁，所以把事务T2对应的锁结构的`is_waiting`属性设置为`false`，然后把该事务对应的线程唤醒，让它继续执行，此时事务T2就算获取到锁了。

- **不加锁**

  意思就是不需要在内存中生成对应的`锁结构`，可以直接执行操作。

- **获取锁成功**，或者加锁成功

  意思就是在内存中生成了对应的`锁结构`，而且锁结构的`is_waiting`属性为`false`，也就是事务可以继续执行操作。

- **获取锁失败**，或者加锁失败，或者没有获取到锁

  意思就是在内存中生成了对应的锁结构，不过锁结构的`is_waiting`属性为`true`，也就是事务需要等待，不可以继续执行操作。

读写/写读

可能发生`脏读`、`不可重复读`、`幻读`的问题。

怎么解决`脏读`、`不可重复读`、`幻读`这些问题呢?

方案一：**读操作利用多版本并发控制（MVCC，下章讲解），写操作进行加锁。**

所谓的`MVCC`，就是生成一个`ReadView`，通过ReadView找到符合条件的记录版本（历史版本由`undo日志`构建)。查询语句只能`读`到在生成ReadView之前`已提交事务所做的更改`，在生成ReadView之前未提交的事务或者之后才开启的事务所做的更改是看不到的。而`写操作`肯定针对的是`最新版本的记`录，读记录的历史版本和改动记录的最新版本本身并不冲突，也就是采用MVCC时，`读-写`操作并不冲突。

> 普通的SELECT语句在READ COMMITTED和REPEATABLE READ隔离级别下会使用到MVCC读取记录。
>
> - 在`READ COMMITTED`隔离级别下，一个事务在执行过程中每次执行SELECT操作时都会生成一个ReadView，ReadView的存在本身就保证了`事务不可以读取到未提交的事务所做的更改`，也就是避免了脏读现象；
> - 在`REPEATABLE READ`隔离级别下，一个事务在执行过程中只有第一次执行SELECT操作才会生成一个ReadView，之后的SELECT操作都`复用`这个ReadView，这样也就避免了不可重复读和幻读的问题。

方案二：读、写操作都采用`加锁`的方式。

`读`操作和`写`操作也像`写-写`操作那样`排队`执行。

`脏读`的产生是因为当前事务读取了另一个未提交事务写的一条记录，如果另一个事务在写记录的时候就给这条记录加锁，那么当前事务就无法继续读取该记录了，所以也就不会有脏读问题的产生了。

`不可重复读`的产生是因为当前事务先读取一条记录，另外一个事务对该记录做了改动之后并提交之后，当前事务再次读取时会获得不同的值，如果在当前事务读取记录时就给该记录加锁那么另一个事务就无法修改该记录，自然也不会发生不可重复读了。

`幻读`问题的产生是因为当前事务读取了一个范围的记录，然后另外的事务向该范围内插入了新记录，当前事务再次读取该范围的记录时发现了新插入的新记录。**采用加锁的方式解决幻读问题就有一些麻烦，因为当前事务在第一次读取记录时幻影记录并不存在，所以读取的时候加锁就有点尴尬（因为你并不知道给谁加锁)。**

小结对比发现:

- **采用`MVCC`方式的话，`读-写`操作彼此并不冲突，`性能更高`。**
- **采用`加锁`方式的话，`读-写`操作彼此需要`排队执行`，影响性能。**

一般情况下我们当然愿意采用`MVCC`来解决`读-写`操作并发执行的问题，但是业务在某些特殊情况下，要求必须采用`加锁`的方式执行。

数据操作类型划分

共享锁S

读锁 可读

排他锁X

写锁 可读可写

对于 InnoDB 引擎来说，读锁和写锁可以加在表上，也可以加在行上。

兼容是指对同一张表或记录的锁的兼容性情况，只有S-S兼容

举例(行级读写锁)：如果一个事务T1已经获得了某个行r的读锁，那么此时另外的一个事务T2是可以去获得这个行r的读锁的，因为读取操作并没有改变行r的数据;但是，如果某个事务T3想获得行r的写锁，则它必须等待事务T1、T2释放掉行r上的读锁才行。

锁定读

在采用`加锁`方式解决脏读、不可重复读、幻读这些问题时，读取一条记录时需要获取该记录的S锁，其实是不严谨的，**有时候需要在读取记录时就获取记录的X锁，来禁止别的事务读写该记录**

对读取的记录**加S锁**:

```
SELECT ... LOCK IN SHARE MODE;
#或
SELECT ... FOR SHARE;#(8.0新增语法)
```

如果别的事务想要获取这些记录的`x锁`，那么它们会阻塞，直到当前事务提交之后将这些记录上的`S锁`释放掉。

对读取的记录**加X锁**:

```
SELECT ... FOR UPDATE;
```

如果别的事务想要获取这些记录的S锁或者X锁，那么它们会阻塞，直到当前事务提交之后将这些记录上的X锁释放掉。

在5.7及之前的版本，SELECT ..FOR UPDATE，如果获取不到锁，会一直等待，直到`innodb_lock_wait_timeout`超时。

而在8.0

通过添加**NOWAIT、SKIP LOCKED**语法，能够立即返回。如果查询的行已经加锁:

- 那么NOWAIT会立即报错返回（等不到锁立即报错）

- 而SKIP LOCKED也会立即返回，只是返回的结果中不包含被锁定的行。

如：SELECT. FOR UPDATE NOWAIT

写操作

无非是`DELETE`、`UPDATE`、`INSERT`这三种

- DELETE:
  对一条记录做DELETE操作的过程其实是先在`B+`树中定位到这条记录的位置，然后获取这条记录的`X锁`，再执行`delete mark`.操作。我们也可以把这个定位待删除记录在B+树中位置的过程看成是一个获取`X锁`的`锁定读`。
- UPDATE : 在对一条记录做UPDATE操作时分为三种情况:
  - 情况1: 未修改该记录的`键值`，并且被更新的列占用的存储空间在修改前后`未发生变化`。

    则先在`B+`树中定位到这条记录的位置，然后再获取一下记录的`X锁`，最后在原记录的位置进行修改操作。我们也可以把这个定位待修改记录在B+树中位置的过程看成是一个获取`X锁`的`锁定读`。
  - 情况2 : 未修改该记录的`键值`，并且至少有一个被更新的列占用的存储空间在修改前后发生变化。

    则先在B+树中定位到这条记录的位置，然后获取一下记录的X锁，将该记录彻底删除掉（就是把记录彻底移入垃圾链表)，最后再插入一条新记录。这个定位待修改记录在B+树中位置的过程看成是一个获取`X锁`的`锁定读`，新插入的记录由`INSERT`操作提供的`隐式锁`进行保护。
  - 情况3 : 修改了该记录的键值，则相当于在原记录上做DELETE操作之后再来一次INSERT操作，加锁操作就需要按照`DELETE`和`INSERT`的规则进行了。
- INSERT :
- 一般情况下，新插入一条记录的操作并不加锁，通过一种称之为`隐式锁`的结构来保护这条新插入的记录在本事务`提交前不被别的事务访问`。

数据操作的粒度划分

表锁

该锁会锁定整张表，它是MySQL中最基本的锁策略，`并不依赖于存储引擎`(不管你是MySQL的什么存储引擎，对于表锁的策略都是一样的)，并且表锁是`开销最小`的策略（因为粒度比较大)。由于表级锁一次会将整个表锁定，所以可以很好的`避免死锁`问题。当然，锁的粒度大所带来最大的负面影响就是出现锁资源争用的概率也会最高，导致`并发率大打折扣`。

表级别的S锁 X锁

在对某个表执行SELECT、INSERT、DELETE、UPDATE语句时，InnoDB存储引擎是不会为这个表添加表级别的`S锁`或者`X锁`的。**在对某个表执行一些诸如`ALTER TABLE`、`DROP TABLE`这类的`DDL`语句时，其他事务对这个表并发执行诸如SELECT、INSERT、DELETE、UPDATE的语句会发生阻塞。同理，某个事务中对某个表执行SELECT、INSERT、DELETE、UPDATE语句时，在其他会话中对这个表执行`DDL`语句也会发生阻塞。这个过程其实是通过在server层使用一种称之为`元数据锁`（英文名：`Metadata Locks`，简称`MDL`）结构来实现的。**

一般情况下，不会使用InnoDB存储引擎提供的 表级别 **的** S锁 **和** X锁 **。只会在一些特殊情况下，比方说** 崩溃恢复 过程中用到。比如，在系统变量 `autocommit=0，innodb_table_locks = 1` 时，手动获取InnoDB存储引擎提供的表t 的S锁或者X锁可以这么写：

- `LOCK TABLES t READ`：InnoDB存储引擎会对表t加表级别的S锁。
- `LOCK TABLES t WRITE`：InnoDB存储引擎会对表t加表级别的X锁。

不过尽量避免在使用InnoDB存储引擎的表上使用 `LOCK TABLES` 这样的手动锁表语句，它们并不会提供什么额外的保护，只是会降低并发能力而已。InnoDB的厉害之处还是实现了更细粒度的行锁，关于InnoDB表级别的S锁和X锁大家**了解一下就可以了。**

意向锁intention lock

InnoDB 支持 多粒度锁（`multiple granularity locking`），**它允许行级锁与表级锁共存**，而 **意向锁** 就是其中的一种 表锁 。

1、意向锁的存在是为了 协调行锁和表锁 的关系，支持多粒度（表锁与行锁)的锁并存。

2、意向锁是一种 不与行级锁冲突表级锁 ，这一点非常重要。

3、表明"**某个事务正在某些行持有了锁或该事务有意向去持有锁**"

意向锁要解决的问题

现在有两个事务，分别是T1和T2，其中T2试图在该表级别上应用共享或排它锁，如果没有意向锁存在，那么T2就需要去检查各个页或行是否存在锁;如果存在意向锁，那么此时就会受到由T1控制的 表级别意向锁的阻塞 。T2在锁定该表前不必检查各个页或行锁，而只需检查表上的意向锁。简单来说就是给更大一级别的空间示意里面是否已经上过锁。

在数据表的场景中，**如果我们给某一行数据加上了排它锁，数据库会自动给更大一级的空间，比如数据页或数据表加上意向锁，告诉其他人这个数据页或数据表已经有人上过排它锁了**(不这么做的话，想上表锁的那个程序，还要遍历有没有行所)，这样当其他人想要获取数据表排它锁的时候，只需要了解是否有人已经获取了这个数据表的意向排他锁即可。

- 如果事务想要获得数据表中某些记录的共享锁，就需要在数据表上 添加意向共享锁 。
- 如果事务想要获得数据表中某些记录的排他锁，就需要在数据表上 添加意向排他锁 。

这时，意向锁会告诉其他事务已经有人锁定了表中的某些记录。

意向锁分为两种：

- **意向共享锁 （intention shared lock, IS）** ：事务有意向对表中的某些行加 **共享锁 （S锁）**

```
-- 事务要获取某些行的 S 锁，必须先获得表的 IS 锁。
-- 会自动加，不用管
SELECT column FROM table ... LOCK IN SHARE MODE;
```

- **意向排他锁 （intention exclusive lock, IX）** ：事务有意向对表中的某些行加 **排他锁** （X锁）

```
-- 事务要获取某些行的 X 锁，必须先获得表的 IX 锁。
-- 会自动加，不用管
SELECT column FROM table ... FOR UPDATE;
```

即：**意向锁是由存储引擎`自己维护的`，用户无法手动操作意向锁，在为数据行加共享 / 排他锁之前，InooDB 会先获取该数据行`所在数据表的对应意向锁`。**

例：事务B检测事务A持有teacher表的意向排他锁，就可以得知事务A必然持有该表中某些数据行的排他锁，那么事务B对teacher表的加表锁请求就会被排斥（阻塞），而无需去检测表中的每一行数据是否存在排他锁。

意向锁之间是兼容的

	意向共享锁(IS)	意向排他锁(IX)
共享锁(S)表	兼容	互斥
排他锁（X)表	互斥	互斥

注意这里的排他/共享锁指的都是表锁，意向锁不会与行级的共享/排他锁互斥。

自增锁auto-inc锁

了解

元数据锁MDL锁

MDL 的作用是，保证读写的正确性。比如，如果一个查询正在遍历一个表中的数据，而执行期间另一个线程对这个`表结构做变更`，增加了一列，那么查询线程拿到的结果跟表结构对不上，肯定是不行的。

因此，**当对一个表做增删改查操作的时候，加 MDL读锁；当要对表做结构变更操作的时候，加 MDL 写锁。**

读锁之间不互斥，因此你可以有多个线程同时对一张表增删改查。读写锁之间、写锁之间是互斥的，用来保证变更表结构操作的安全性，解决了DML和DDL操作之间的一致性问题。`不需要显式使用`，**在访问一个表的时候会被自动加上。**

行锁

行锁(Row Lock)也称为记录锁，顾名思义，就是锁住某一行（某条记录row)。需要的注意的是，MySQL服务器层并没有实现行锁机制，**行级锁只在存储引擎层实现。**

优点: 锁定力度小，`发生锁冲突概率低`，可以实现的`并发度高`。

缺点: 对于`锁的开销比大`，加锁会比较慢，容易出现`死锁`情况。

记录锁（Record Locks)

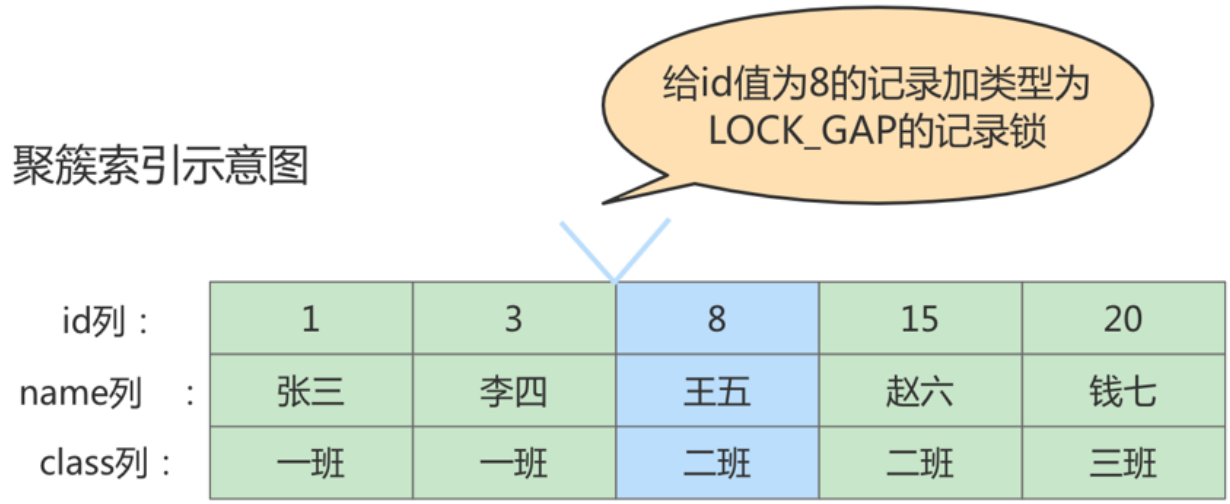
记录锁是有S锁和X锁之分的，称之为`S型记录锁`和`X型记录锁`。跟共享锁排他锁基本一致

- 当一个事务获取了一条记录的S型记录锁后，其他事务也可以继续获取该记录的S型记录锁，但不可以继续获取X型记录锁；

- 当一个事务获取了一条记录的X型记录锁后，其他事务既不可以继续获取该记录的S型记录锁，也不可以继续获取X型记录锁。

间隙锁（Gap Locks）

MySQL在`REPEATABLE READ`隔离级别下是可以解决幻读问题的，解决方案有两种，可以使用`MVCC`方案解决，也可以采用加锁方案解决。但是在使用`加锁`方案解决时有个大问题，就是**事务在第一次执行读取操作时，那些幻影记录尚不存在，我们无法给这些`幻影记录`加上`记录锁`**。InnoDB提出了一种称之为`Gap Locks`的锁，官方的类型名称为：`LOCK_GAP`，我们可以简称为`gap锁`。



图中**id值为 8 的记录加了gap锁，意味着**`不允许别的事务在id值为 8 的记录前边的间隙插入新记录`，其实就是id列的值(3 , 8)这个区间的新记录是不允许立即插入的。比如，有另外一个事务再想插入一条id值为 4 的新记录，它定位到该条新记录的下一条记录的id值为 8 ，而这条记录上又有一个gap锁，所以就会阻塞插入操作，直到拥有这个gap锁的事务提交了之后，id列的值在区间(3 , 8)中的新记录才可以被插入。

gap锁的提出仅仅是为了防止插入幻影记录而提出的。虽然有共享`gap锁`和`独占gap锁`这样的说法，但是它们起到的作用是相同的。而且如果对一条记录加了gap锁（不论是共享gap锁还是独占gap锁），并不会限制其他事务对这条记录加记录锁或者继续加gap锁。

临键锁（Next-Key Locks）

有时候我们既想`锁住某条记录`，又想`阻止`其他事务在该记录前边的`间隙插入新记录`，所以InnoDB就提出了一种称之为`Next-Key Locks的锁`，官方的类型名称为：`LOCK_ORDINARY`，我们也可以简称为next-key锁。`Next-Key Locks`是在存储引擎innodb、事务级别在`可重复读`的情况下使用的数据库锁，innodb默认的锁就是Next-Key locks。

可以说是记录锁和间隙锁的合体

插入意向锁（Insert Intention Locks）

我们说一个事务在`插入`一条记录时需要判断一下插入位置是不是被别的事务加了`gap锁`（`next-key锁`也包含`gap锁`），如果有的话，插入操作需要等待，直到拥有`gap锁`的那个事务提交。但是 **InnoDB规定事务在等待的时候也需要在内存中生成一个锁结构**，表明**有事务想在某个`间隙`中`插入`新记录，但是现在在等待**。InnoDB就把**这种类型的锁命名为Insert Intention Locks，官方的类型名称为：`LOCK_INSERT_INTENTION`，我们称为插入意向锁。**`插入意向锁`是一种`Gap锁`，不是意向锁，在insert操作时产生。

插入意向锁是在插入一条记录行前，由 `INSERT` 操作产生的一种间隙锁。

事实上 **插入意向锁并不会阻止别的事务继续获取该记录上任何类型的锁。**

页锁

页锁就是在 `页的粒度` 上进行锁定，锁定的数据资源比行锁要多，因为一个页中可以有多个行记录。当我们使用页锁的时候，会出现数据浪费的现象，但这样的浪费最多也就是一个页上的数据行。 **页锁的开销介于表锁和行锁之间，会出现死锁。锁定粒度介于表锁和行锁之间，并发度一般。**

每个层级的锁数量是有限制的，因为锁会占用内存空间， `锁空间的大小是有限的` **。当某个层级的锁数量超过了这个层级的阈值时，就会进行** `锁升级` **。锁升级就是用更大粒度的锁替代多个更小粒度的锁，比如InnoDB中行锁升级为表锁，这样做的好处是占用的锁空间降低了，但同时数据的并发度也下降了。**

对待锁的态度划分

乐观锁和悲观锁并不是锁，而是锁的 `设计思想` 。

悲观锁

悲观锁是一种思想，顾名思义，就是很悲观，对数据被其他事务的修改持保守态度，会通过数据库自身的锁机制来实现，从而保证数据操作的排它性。

悲观锁总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会 `阻塞` 直到它拿到锁（ 共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程 ）。比如行锁，表锁等，读锁，写锁等，都是**在做操作之前先上锁，当其他线程想要访问数据时，都需要阻塞挂起。**Java中 `synchronized` 和 `ReentrantLock` 等独占锁就是悲观锁思想的实现。

select ... for update就是MySQL中的悲观锁

select ... for update语句在执行过程中会把所有扫描的行都锁上，因此用悲观锁必须确定用了索引，而不是全表扫描，否则会把整个表锁住

悲观锁大多数情况下依靠数据库锁机制来实现，对性能开销影响很大

乐观锁

乐观锁认为对同一数据的并发操作不会总发生，属于小概率事件，不用每次都对数据上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，也就是 **不采用数据库自身的锁机制，而是通过程序来实现** 。在程序上，我们可以**采用** `版本号机制` **或者** `CAS机制` **实现。 乐观锁适用于多读的应用类型，这样可以提高吞吐量** 。在Java中java.util.concurrent.atomic包下的原子变量类就是使用了乐观锁的一种实现方式：CAS实现的。

1.乐观锁的版本号机制

在表中设计一个 `版本字段 version` ，第一次读的时候，会获取 version 字段的取值。然后对数据进行更新或删除操作时，会执行 `UPDATE ... SET version=version+1 WHERE version=version` 。此时如果已经有事务对这条数据进行了更改，修改就不会成功。

2. 乐观锁的时间戳机制

时间戳和版本号机制一样，也是在更新提交的时候，将当前数据的时间戳和更新之前取得的时间戳进行比较，如果两者一致则更新成功，否则就是版本冲突。

你能看到乐观锁就是程序员自己控制数据并发操作的权限，基本是通过给数据行增加一个戳（版本号或者时间戳），从而证明当前拿到的数据是否最新。

两种锁的适用场景

从这两种锁的设计思想中，我们总结一下乐观锁和悲观锁的适用场景：

1. `乐观锁` **适合** `读操作多` **的场景，**相对来说写的操作比较少。它的优点在于程序实现，不存在死锁

问题，不过适用场景也会相对乐观，因为它阻止不了除了程序以外的数据库操作。

2. `悲观锁` **适合** `写操作多` **的场景，**因为写的操作具有`排它性`。采用悲观锁的方式，可以在数据库层面阻止其他事务对该数据的操作权限，防止`读 - 写`和`写 - 写`的冲突。

加锁的方式划分

隐式锁

- 情景一： 对于聚簇索引记录来说，有一个trx_id隐藏列，该隐藏列记录着最后改动该记录的事务id。那么如果在当前事务中新插入一条聚簇索引记录后，该记录的trx_id隐藏列代表的的就是当前事务的事务id，如果其他事务此时想对该记录添加S锁或者X锁时，首先会看一下该记录的trx_id隐藏列代表的事务是否是当前的活跃事务，如果是的话，那么就帮助当前事务创建一个X锁（也就是为当前事务创建一个锁结构，is_waiting属性是false），然后自己进入等待状态（也就是为自己也创建一个锁结构，is_waiting属性是true）。
- 情景二： 对于二级索引记录来说，本身并没有trx_id隐藏列，但是在二级索引页面的PageHeader部分有一个PAGE_MAX_TRX_ID属性，该属性代表对该页面做改动的最大的事务id，如果PAGE_MAX_TRX_ID属性值小于当前最小的活跃事务id，那么说明对该页面做修改的事务都已经提交了，否则就需要在页面中定位到对应的二级索引记录，然后回表找到它对应的聚簇索引记录，然后再重复情景一的做法。

隐式锁的逻辑过程如下：

A. InnoDB的每条记录中都一个隐含的trx_id字段，这个字段存在于聚簇索引的B+Tree中。

B. 在操作一条记录前，首先根据记录中的trx_id检查该事务是否是活动的事务(未提交或回滚)。如果是活动的事务，首先将隐式锁转换为显式锁(就是为该事务添加一个锁)。

C. 检查是否有锁冲突，如果有冲突，创建锁，并设置为waiting状态。如果没有冲突不加锁，跳到E。

D. 等待加锁成功，被唤醒，或者超时。

E. 写数据，并将自己的trx_id写入trx_id字段。

显式锁

通过特定的语句进行加锁，我们一般称之为显示加锁

1. **隐式锁（Implicit Lock）：**
    - 隐式锁是由编程语言或数据库管理系统自动管理的锁。在使用隐式锁时，程序员无需显式地编写加锁和解锁的代码，锁的管理由语言或数据库系统自动完成。

- 隐式锁通常用于高级编程语言中的内建数据结构或数据库管理系统中的内部机制。例如，在数据库中使用隐式锁来控制事务的并发访问，或在编程语言中使用隐式锁来保护共享对象的访问。

2. **显式锁（Explicit Lock）：**
   - 显式锁是由程序员在代码中显式地编写加锁和解锁的操作。在使用显式锁时，程序员需要手动指定需要锁定的共享资源，并在使用完后手动释放锁。
   - 显式锁通常用于低级编程语言或需要更细粒度控制的场景。例如，在多线程编程中，使用显式锁来保护共享资源，防止多个线程同时访问而引发数据竞争和数据一致性问题。

全局锁

全局锁就是对`整个数据库实例`加锁。当你需要让整个库处于`只读状态`的时候，可以使用这个命令，之后其他线程的以下语句会被阻塞：数据更新语句（数据的增删改）、数据定义语句（包括建表、修改表结构等）和更新类事务的提交语句。全局锁的典型使用`场景`是：做`全库逻辑备份`。

全局锁的命令：

Flush tables with read lock

死锁

死锁是指两个或多个事务在同一资源上相互占用，并请求锁定对方占用的资源，从而导致恶性循环。

两个事务都持有对方需要的锁，并且都在等待对方释放，但双方都不会释放自己的锁

当出现死锁以后，有`两种策略`：

- 一种策略是，直接进入等待，直到超时。这个超时时间可以通过参数`innodb_lock_wait_timeout`来设置。
- 另一种策略是，发起死锁检测，发现死锁后，主动回滚死锁链条中的某一个事务（将持有最少行级排他锁的事务进行回滚），让其他事务得以继续执行。将参数`innodb_deadlock_detect`设置为on，表示开启这个逻辑。

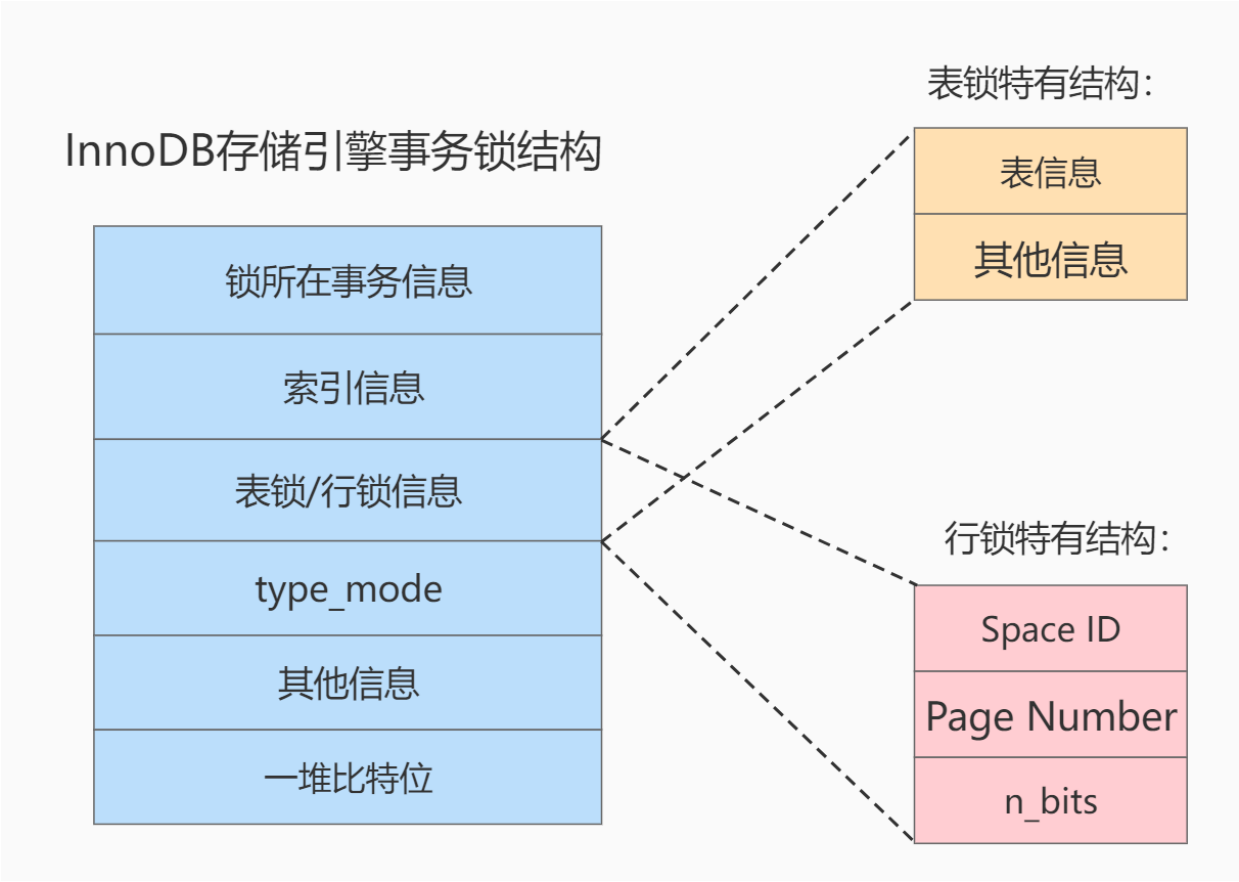
第二种策略的成本分析

方法 1： 如果你能确保这个业务一定不会出现死锁，可以临时把死锁检测关掉。 但是这种操作本身带有一定的风险，因为业务设计的时候一般不会把死锁当做一个严重错误，毕竟出现死锁了，就回滚，然后通过业务重试一般就没问题了，这是`业务无损的`。而**关掉死锁检测意味着可能会出现大量的超时**，这是`业务有损`的。

方法 2： 控制并发度。 如果并发能够控制住，比如同一行同时最多只有 10 个线程在更新，那么死锁检测的成本很低，就不会出现这个问题。

这个并发控制要做在`数据库服务端`。如果你有中间件，可以考虑在`中间件实现`；甚至有能力修改MySQL源码的人，也可以做在MySQL里面。基本思路就是，对于相同行的更新，在进入引擎之前排队，这样在InnoDB内部就不会有大量的死锁检测工作了。

锁内存结构



锁监控

关于MySQL锁的监控，我们一般可以通过检查`InnoDB_row_lock`等状态变量来分析系统上的行锁的争夺情况

多版本并发控制MVCC

MVCC （Multiversion Concurrency Control），多版本并发控制。顾名思义，MVCC 是通过**数据行的多个版本管理**来实现数据库的`并发控制`。这项技术使得在InnoDB的事务隔离级别下执行`一致性读`操作有了保证。换言之，就是为了查询一些正在被另一个事务更新的行，并且可以看到它们被更新之前的值，这样在做查询的时候就不用等待另一个事务释放锁。

MVCC没有正式的标准，在不同的DBMS中MVCC的实现方式可能是不同的，也不是普遍使用的(大家可以参考相关的DBMS文档)。这里讲解InnoDB 中MVCC的实现机制（MySQL其它的存储引擎并不支持它)。

MVCC在MySQL InnoDB中的实现主要是为了提高数据库并发性能，用**更好的方式去处理**`读-写冲突`，做到即使有读写冲突时，也能做到`不加锁，非阻塞并发读`，而**这个读指的就是**`快照读`**，而非**`当前读`。当前读实际上是一种加锁的操作，是悲观锁的实现。而MVCC本质是采用乐观锁思想的一种方式。

快照读

快照读又叫一致性读，读取的是快照数据。 **不加锁的简单的 SELECT 都属于快照读**，即不加锁的非阻塞读

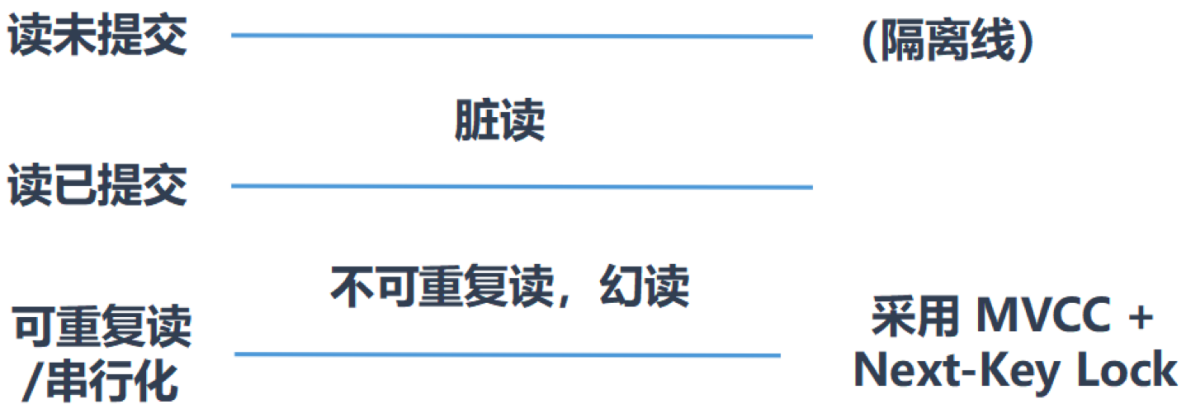
既然是基于多版本，那么快照读可能读到的并不一定是数据的最新版本，而有可能是之前的历史版本。

快照读的前提是隔离级别不是串行级别，串行级别下的快照读会退化成当前读。

当前读

当前读读取的是**记录的最新版本**（最新数据，而不是历史版本的数据），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。**加锁的 SELECT，或者对数据进行增删改都会进行当前读。**

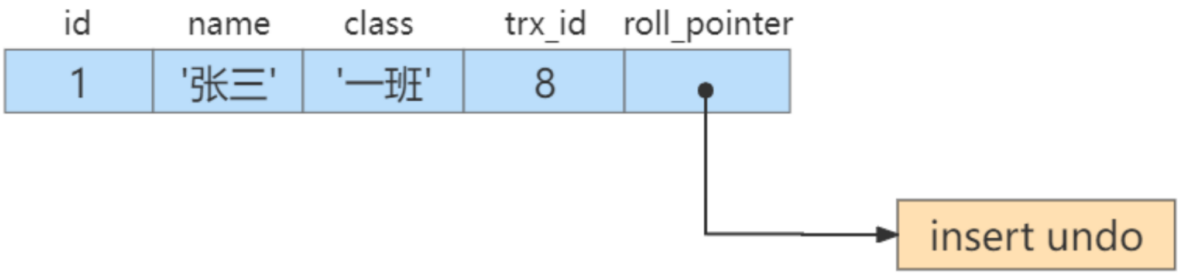
在MySQL中，MVCC 可以不采用锁机制，而是通过乐观锁的方式来解决不可重复读和幻读问题!它可以在大多数情况下替代行级锁，降低系统的开销。



回顾一下undo日志的版本链，对于使用`InnoDB`存储引擎的表来说，它的聚簇索引记录中都包含两个必要的隐藏列(字段)。

- trx_id：每次一个事务对某条聚簇索引记录进行改动时，都会把该事务的事务id赋值给trx_id隐藏列。
- roll_pointer：每次对某条聚簇索引记录进行改动时，都会把旧的版本写入到`undo日志`中，然后这个隐藏列就相当于一个指针，可以通过它来找到该记录修改前的信息。

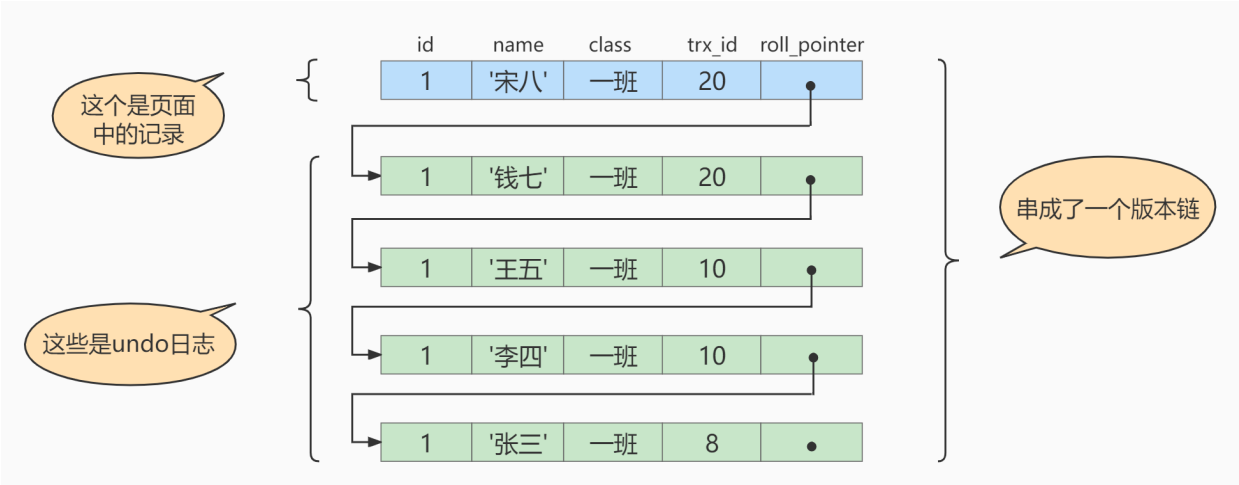
假设插入该记录的`事务id`为`8`，那么此刻该条记录的示意图如下所示:



insert undo只在事务回滚时起作用，当事务提交后，该类型的undo日志就没用了，它占用的UndoLog Segment也会被系统回收（也就是该undo日志占用的Undo页面链表要么被重用，要么被释放）。

Undo Log版本链

每次对记录进行改动，都会记录一条undo日志，每条undo日志也都有一个`roll_pointer`属性（INSERT操作对应的undo日志没有该属性，因为该记录并没有更早的版本），可以将这些`undo日志`都连起来，串成一个链表：



对该记录每次更新后，都会将旧值放到一条`undo日志`中，就算是该记录的一个旧版本，随着更新次数的增多，所有的版本都会被`roll_pointer`属性连接成一个链表，我们把这个链表称之为`版本链`，版本链的头节点就是当前记录最新的值。

每个版本中还包含生成该版本时对应的`事务id`。

MVCC实现原理之ReadView

MVCC 的实现依赖于： **隐藏字段、Undo Log、Read View** 。

什么是ReadView

在MVCC机制中，多个事务对同一个行记录进行更新会产生多个历史快照，这些历史快照保存在Undo Log里。如果一个事务想要查询这个行记录，需要读取哪个版本的行记录呢?这时就需要用到ReadView了，它帮我们解决了行的可见性问题。

ReadView就是某一个事务在使用MVCC机制进行快照读操作时产生的读视图。当事务启动时，会生成数据库系统当前的一个快照，InnoDB为每个事务构造了一个数组，用来记录并维护系统当前活`跃事务的ID`("活跃"指的就是，启动了但还没提交)。

普通的SELECT语句在READ COMMITTED和REPEATABLE READ隔离级别下会使用到MVCC读取记录。

- 在`READ COMMITTED`隔离级别下，一个事务在执行过程中每次执行SELECT操作时都会生成一个ReadView，ReadView的存在本身就保证了`事务不可以读取到未提交的事务所做的更改`，也就是避免了脏读现象；
- 在`REPEATABLE READ`隔离级别下，一个事务在执行过程中只有第一次执行SELECT操作才会生成一个ReadView，之后的SELECT操作都`复用`这个ReadView，这样也就避免了不可重复读和幻读的问题。

设计思路

使用`READ UNCOMMITTED`隔离级别的事务，由于可以读到未提交事务修改过的记录，所以直接读取记录的最新版本就好了。

使用`SERIALIZABLE`隔离级别的事务，InnoDB规定使用加锁的方式来访问记录。

使用`READ COMMITTED`和`REPEATABLE READ`隔离级别的事务，都必须保证读到已经提交了的事务修改过的记录。假如另一个事务已经修改了记录但是尚未提交，是不能直接读取最新版本的记录的，核心问题就是需要判断一下版本链中的哪个版本是当前事务可见的，这是ReadView要解决的主要问题。

这个ReadView中主要包含 4 个比较重要的内容，分别如下：

1. `creator_trx_id`，创建这个 Read View 的事务 ID。

   > 说明：只有在对表中的记录做改动时（执行INSERT、DELETE、UPDATE这些语句时）才会为事务分配事务id，否则在一个只读事务中的事务id值都默认为 0 。

2. `trx_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`。
3. `up_limit_id`，活跃的事务中最小的事务 ID。
4. `low_limit_id`，表示生成ReadView时系统中应该分配给下一个事务的`id`值。`low_limit_id` 是系统最大的事务id值，这里要注意是系统中的事务id，需要区别于正在活跃的事务ID。

> 注意：low_limit_id并不是trx_ids中的最大值，事务id是递增分配的。比如，现在有id为 1 ， 2 ， 3 这三个事务，之后id为 3 的事务提交了。那么一个新的读事务在生成ReadView时，trx_ids就包括 1 和 2 ，up_limit_id的值就是 1 ，low_limit_id的值就是 4 。

ReadView的规则

有了这个ReadView，这样在访问某条记录时，只需要按照下边的步骤判断记录的某个版本是否可见。

- 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`up_limit_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`low_limit_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up_limit_id和`low_limit_id`之间，那就需要判断一下trx_id属性值是不是在trx_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整体操作流程

了解了这些概念之后，我们来看下当查询一条记录的时候，系统如何通过MVCC找到它：

1. 首先获取事务自己的版本号，也就是事务 ID；
2. 生成 ReadView；
3. 查询得到的数据，然后与 ReadView 中的事务版本号进行比较；
4. 如果不符合 ReadView 规则，就需要从 Undo Log 中获取历史快照；
5. 最后返回符合规则的数据。

如果某个版本的数据对当前事务不可见的话，那就顺着版本链找到下一个版本的数据，继续按照上边的步骤判断可见性，依此类推，直到版本链中的最后一个版本。如果最后一个版本也不可见的话，那么就意味着该条记录对该事务完全不可见，查询结果就不包含该记录。

InnoDB中，MVCC是通过Undo Log + Read View进行数据读取，Undo Log保存了历史快照，而Read View规则帮我们判断当前版本的数据是否可见。

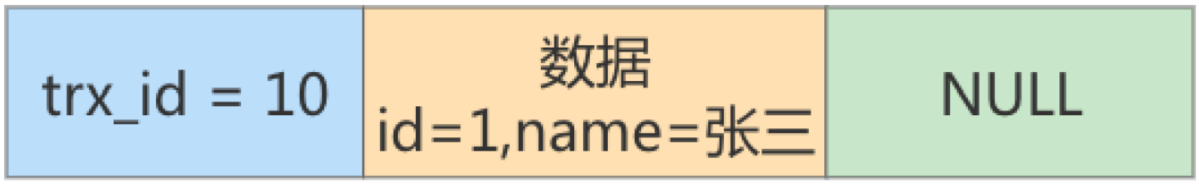
在隔离级别为读已提交（Read Committed）时，一个事务中的每一次 SELECT 查询都会重新获取一次Read View。

当隔离级别为可重复读的时候，就避免了不可重复读，这是因为一个事务只在第一次 SELECT 的时候会获取一次 Read View，而后面所有的 SELECT 都会复用这个 Read View

如何解决幻读

接下来说明InnoDB在可重复读下是如何解决幻读的。

假设现在表 student 中只有一条数据，数据内容中，主键 id=1，隐藏的 trx_id=10，它的 undo log 如下图所示。



假设现在有事务 A 和事务 B 并发执行，事务 A 的事务 id 为 20 ，事务 B 的事务 id 为 30 。

步骤 1 ： 事务 A 开始第一次查询数据，查询的 SQL 语句如下。

select * from student where id >= 1 ;

在开始查询之前，MySQL 会为事务 A 产生一个 ReadView，此时 ReadView 的内容如下：`trx_ids=[20,30]，up_limit_id=20，low_limit_id=31，creator_trx_id=20`。

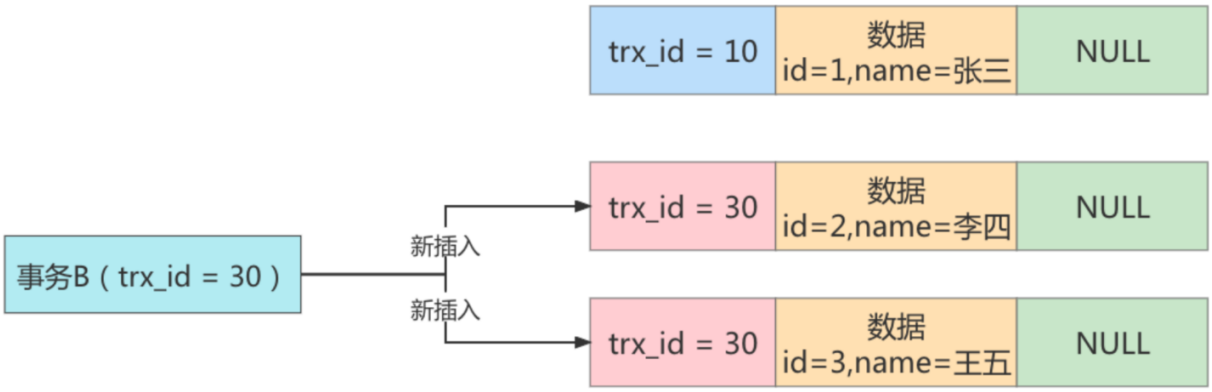
由于此时表 student 中只有一条数据，且符合 where id>=1 条件，因此会查询出来。然后根据 ReadView机制，发现该行数据的trx_id=10，小于事务 A 的 ReadView 里 up_limit_id，这**表示这条数据是事务 A 开启之前，其他事务就已经提交了的数据**，因此事务 A 可以读取到。

结论：事务 A 的第一次查询，能读取到一条数据，id=1。

步骤 2 ： 接着事务 B(trx_id=30)，往表 student 中新插入两条数据，并提交事务。

insert into student(id,name) values(2 ,'李四');
insert into student(id,name) values(3 ,'王五');

此时表student 中就有三条数据了，对应的 undo 如下图所示：

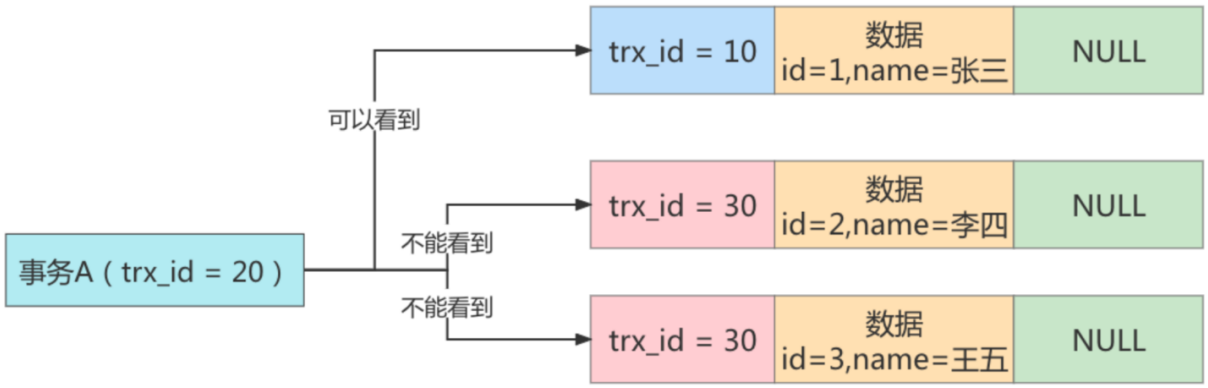


步骤 3： **接着事务 A 开启第二次查询，根据可重复读隔离级别的规则，此时事务 A 并不会再重新生成 ReadView。**此时表 student 中的 3 条数据都满足 where id>=1 的条件，因此会先查出来。然后根据 ReadView 机制，判断每条数据是不是都可以被事务 A 看到。

1 ）首先 id=1 的这条数据，前面已经说过了，可以被事务 A 看到。

2 ）然后是 id=2 的数据，它的 trx_id=30，此时事务 A 发现，这个值处于 up_limit_id 和 low_limit_id 之间，因此还需要再判断 30 是否处于 trx_ids 数组内。由于事务 A 的 trx_ids=[20,30]，因此**在数组内，这表示 id=2 的这条数据是与事务 A 在同一时刻启动的其他事务提交的，所以这条数据不能让事务 A 看到。**

3 ）同理，id=3 的这条数据，trx_id 也为 30 ，因此也不能被事务 A 看见。



结论：最终事务 A 的第二次查询，只能查询出 id=1 的这条数据。这和事务 A 的第一次查询的结果是一样的，因此没有出现幻读现象，所以说在 MySQL 的可重复读隔离级别下，不存在幻读问题。

通过MVCC我们可以解决:

1. `读写之间阻塞的问题`。通过MVCC可以让读写互相不阻塞，即读不阻塞写，写不阻塞读，这样就可以提升事
   务并发处理能力。
2. `降低了死锁的概率`。这是因为MVCC采用了乐观锁的方式，读取数据时并不需要加锁，对于写操作，也只锁
   定必要的行。
3. `解决快照读的问题`。当我们查询数据库在某个时间点的快照时，只能看到这个时间点之前事务提交更新的结果，而不能看到这个时间点之后事务提交的更新结果。