# 数据库相关 面试积累

# 正则表达式

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Mysql**  **正则表达式** | MySQL可以通过 LIKE ...% 来进行模糊匹配。  MySQL中使用 REGEXP 操作符来进行正则表达式匹配。  操作正则：   |  | | --- | | **REGEXP** 或 **NOT REGEXP**  SELECT \* FROM Websites WHERE name REGEXP '^[GFs]';  ^[A-H] 以 A 到 H 字母开头  ^[^A-H] 不以 A 到 H 字母开头 ^[GFs] G"、"F" 或 "s" 开始 |   下表中的正则模式可应用于 REGEXP 操作符中。   |  |  | | --- | --- | | 模式 | 描述 | | ^ | 匹配输入字符串的开始位置。如果设置了 RegExp 对象的 Multiline 属性，^ 也匹配 '\n' 或 '\r' 之后的位置。 | | $ | 匹配输入字符串的结束位置。如果设置了RegExp 对象的 Multiline 属性，$ 也匹配 '\n' 或 '\r' 之前的位置。 | | . | 匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用像 '[.\n]' 的模式。 | | [...] | 字符集合。匹配所包含的任意一个字符。例如， '[abc]' 可以匹配 "plain" 中的 'a'。 | | [^...] | 负值字符集合。匹配未包含的任意字符。例如， '[^abc]' 可以匹配 "plain" 中的'p'。 | | p1|p2|p3 | 匹配 p1 或 p2 或 p3。例如，'z|food' 能匹配 "z" 或 "food"。'(z|f)ood' 则匹配 "zood" 或 "food"。 | | \* | 匹配前面的子表达式零次或多次。例如，zo\* 能匹配 "z" 以及 "zoo"。\* 等价于{0,}。 | | + | 匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。 | | {n} | n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。 | | {n,m} | m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。 |   以下我们将列出几个小实例(表名：person\_tbl )来加深我们的理解：  查找name字段中以'st'为开头的所有数据：  mysql> SELECT name FROM person\_tbl WHERE name REGEXP '^st';  查找name字段中以'ok'为结尾的所有数据：  mysql> SELECT name FROM person\_tbl WHERE name REGEXP 'ok$';  查找name字段中包含'mar'字符串的所有数据：  mysql> SELECT name FROM person\_tbl WHERE name REGEXP 'mar';  查找name字段中以元音字符开头或以'ok'字符串结尾的所有数据：  mysql> SELECT name FROM person\_tbl WHERE name REGEXP '^[aeiou]|ok$'; |

# SQL和数据库基础

SQL是最基本的数据库考核和实践的内容。基础中的基础。

|  |  |
| --- | --- |
|  | **SQL 对大小写不敏感**。  root 用户拥有最高权限。 |
| Mysql 连接 | [root@host]# mysql -u root -p  Enter password:\*\*\*\*\*\* |
| 创建数据库 | CREATE DATABASE 数据库名; |
| 删除数据库 | drop database 数据库名; |
| 选择数据库 | use RUNOOB; |
| 创建数据库表 | CREATE TABLE table\_name (column\_name column\_type);  CREATE TABLE IF NOT EXISTS `runoob\_tbl`(  `runoob\_id` INT UNSIGNED AUTO\_INCREMENT,  `runoob\_title` VARCHAR(100) NOT NULL,  `runoob\_author` VARCHAR(40) NOT NULL,  `submission\_date` DATE,  PRIMARY KEY ( `runoob\_id` )  )ENGINE=InnoDB DEFAULT CHARSET=utf8; |
| 删除数据库表 | DROP TABLE table\_name ; |
| alter 修改表结构 | |  | | --- | | 使用 ADD 子句来向数据表中添加列  **mysql> ALTER TABLE testalter\_tbl ADD i INT;** | | **以下实例，指定字段 j 为 NOT NULL 且默认值为100 。**  **mysql> ALTER TABLE testalter\_tbl MODIFY j BIGINT NOT NULL DEFAULT 100;**  **如果你不设置默认值，MySQL会自动设置该字段默认为 NULL。** | | **将数据表 testalter\_tbl 重命名为 alter\_tbl：**  **mysql> ALTER TABLE testalter\_tbl RENAME TO alter\_tbl** | | **修改存储引擎：修改为myisam**  **alter table tableName engine=myisam;** | | **删除外键约束：keyName是外键别名**  **alter table tableName drop foreign key keyName;** | |
| **Select 查询** | SELECT DISTINCT ；DISTINCT 关键词用于返回唯一不同的值。  SELECT DISTINCT country FROM Websites;   |  |  | | --- | --- | | limit | SELECT column\_name(s) FROM table\_name LIMIT number;  SELECT \* FROM Persons LIMIT 5;  规定要返回的记录的数目 | | offset | [ OFFSET M] 设定数据偏移量。 | | **union**  **union all** | union 合并两个查询到一个结果中，去掉重复值  **union all 才包含重复值。**  SELECT country FROM Websites UNION SELECT country FROM apps ORDER BY country; | |
| **Where子句** | SELECT \* FROM Websites WHERE country='CN';  文本值 要用 单引号。 数值字段无需引号。  **比较运算：**   |  | | --- | | <> 不等于。 |   **逻辑运算：**   |  | | --- | | And:与 同时满足两个条件的值。  Select \* from emp where sal > 2000 and sal < 3000;  查询 EMP 表中 SAL 列中大于 2000 小于 3000 的值。 | | Or:或 满足其中一个条件的值  Select \* from emp where sal > 2000 or comm > 500;  查询 emp 表中 SAL 大于 2000 或 COMM 大于500的值。 | | 结合 and or  SELECT \* FROM Websites  WHERE alexa > 15  AND (country='CN' OR country='USA'); | | Not:非 满足不包含该条件的值。  select \* from emp where not sal > 1500;  查询EMP表中 sal 小于等于 1500 的值。 | | 逻辑运算的优先级：  () not and or |   **特殊条件**   |  | | --- | | **1.空值判断： is null**  Select \* from emp where comm is null;  查询 emp 表中 comm 列中的空值。 | | **2.between and (在 之间的值)**  Select \* from emp where sal between 1500 and 3000;  查询 emp 表中 SAL 列中大于 1500 的小于 3000 的值。 | | **3.In**  Select \* from emp where sal in (5000,3000,1500);  查询 EMP 表 SAL 列中等于 5000，3000，1500 的值。  SELECT \* FROM Websites WHERE name IN ('Google','菜鸟教程'); | | **4.like （还有not like）**  Like模糊查询  Select \* from emp where ename like 'M%';  查询 EMP 表中 Ename 列中有 M 的值，M 为要查询内容中的模糊信息。  % 表示多个字值，\_ 下划线表示一个字符；  M% : 为能配符，正则表达式，表示的意思为模糊查询信息为 M 开头的。  %M% : 表示查询包含M的所有内容。  %M\_ : 表示查询以M在倒数第二位的所有内容。 |   **不带比较运算符的 WHERE 子句：**   |  | | --- | | WHERE 子句并不一定带比较运算符，当不带运算符时，会执行一个隐式转换。当 0 时转化为 false，1 转化为 true。例如：  SELECT studentNO FROM student WHERE 0  则会返回一个空集，因为每一行记录 WHERE 都返回 false。  SELECT studentNO FROM student WHERE 1  返回 student 表所有行中 studentNO 列的值。因为每一行记录 WHERE 都返回 true。 | |
| order by 排序 | 用于对结果集按照一个列或者多个列进行排序。   |  | | --- | | 默认按照升序（ASC）； DESC 降序  **SELECT \* FROM Websites**  **ORDER BY alexa** DESC; | | SELECT \* FROM Websites  ORDER BY country,alexa;  ORDER BY 多列的时候，先按照第一个column name排序，在按照第二个column name排序 | | order by A,B 这个时候都是默认按升序排列  order by A desc,B 这个时候 A 降序，B 升序排列  order by A ,B desc 这个时候 A 升序，B 降序排列  即 desc 或者 asc 只对它紧跟着的第一个列名有效，其他不受影响，仍然是默认的升序。 | | **MySQL 拼音排序**  如果字符集采用的是 gbk(汉字编码字符集)，直接在查询语句后边添加 ORDER BY：  SELECT \*  FROM runoob\_tbl  ORDER BY runoob\_title;  如果字符集采用的是 utf8(万国码)，需要先对字段进行转码然后排序：  SELECT \*  FROM runoob\_tbl  ORDER BY CONVERT(runoob\_title using gbk); | |
| GROUP BY 分组 | **GROUP BY 语句根据一个或多个列对结果集进行分组。**  **在分组的列上我们可以使用 COUNT, SUM, AVG,等函数。**  **SELECT name, COUNT(\*) FROM employee\_tbl GROUP BY name;**   |  | | --- | | 1. **WITH ROLLUP**：在group分组字段的基础上再进行统计数据。   mysql> SELECT name, SUM(singin) as singin\_count FROM employee\_tbl GROUP BY name WITH ROLLUP;  +--------+--------------+  | name | singin\_count |  +--------+--------------+  | 小丽 | 2 |  | 小明 | 7 |  | 小王 | 7 |  | NULL | 16 |  +--------+--------------+  4 rows in set (0.00 sec) | | **我们可以使用 coalesce 来设置一个可以取代 NUll 的名称，coalesce 语法：**  **select coalesce(a,b,c);**  **参数说明：如果a==null,则选择b；如果b==null,则选择c**  如果名字为空我们使用总数代替：  mysql> SELECT coalesce(name, '总数'), SUM(singin) as singin\_count FROM employee\_tbl GROUP BY name WITH ROLLUP;  +--------------------------+--------------+  | coalesce(name, '总数') | singin\_count |  +--------------------------+--------------+  | 小丽 | 2 |  | 小明 | 7 |  | 小王 | 7 |  | 总数 | 16 |  +--------------------------+--------------+  4 rows in set (0.01 sec) | | **SELECT**  **sum(sale\_amount - payment\_amount) AS creditAmount, SUM(goods\_amount),**  **coalesce(buyer\_name, '总金额') as buyerName**  **FROM**  **dk\_sale\_order\_head**  **WHERE**  **stall\_id = 10065**  **and buyer\_name in ('tcc','QTL')**  **GROUP BY**  **buyer\_name WITH ROLLUP;** | |
| **insert into**  **插入** | |  | | --- | | **第一种形式无需指定要插入数据的列名，只需提供被插入的值即可：**  **INSERT INTO table\_name**  **VALUES (value1,value2,value3,...);**  没有指定要插入数据的列名的形式需要列出插入行的每一列数据: | | **第二种形式需要指定列名及被插入的值：**  **INSERT INTO table\_name (column1,column2,column3,...)**  **VALUES (value1,value2,value3,...);**  **INSERT INTO Websites (name, url, alexa, country)**  **VALUES ('百度','https://www.baidu.com/','4','CN');** | |
| **表复制** | |  | | --- | | INSERT INTO SELECT 语句从一个表复制数据，然后把数据插入到一个**已存在**的表中。 | | 我们可以从一个表中复制所有的列插入到另一个已存在的表中：  INSERT INTO *table2* SELECT \* FROM *table1;* | | 只复制希望的列插入到另一个已存在的表中：  INSERT INTO *table2* *(column\_name(s))* SELECT *column\_name(s)* FROM *table1;* | | **insert into select 和select into from（mysql不支持） 的区别**  **insert into scorebak select \* from socre where neza='neza' --插入一行,要求表scorebak 必须存在**  **select \* into scorebak from score where neza='neza' --也是插入一行,要求表scorebak 不存在** | |
| **Update** | |  | | --- | | UPDATE table\_name SET column1=value1,column2=value2,... WHERE some\_column=some\_value; where 子句指定要更新的行。可以省略。  **没有where的话，将更新所有的行。要慎重。** | | **在 MySQL 中可以通过设置 sql\_safe\_updates 这个自带的参数**  **当该参数开启的情况下，你必须在update 语句后携带 where 条件，否则就会报错**  **set sql\_safe\_updates=1; 表示开启该参数** | |
| **delete** | |  | | --- | | DELETE FROM table\_name WHERE some\_column=some\_value; 如果省略了 WHERE 子句，所有的记录都将被删除！ | | 您可以在不删除表的情况下，删除表中所有的行。  DELETE FROM *table\_name*; 或 DELETE \* FROM *table\_name*;  **注释：**在删除记录时要格外小心！因为您不能重来！ | |
| **SQL约束** | |  | | --- | | * **NOT NULL** - 指示某列不能存储 NULL 值。   CREATE TABLE Persons  (  P\_Id int NOT NULL,  LastName varchar(255) NOT NULL,  )  ALTER TABLE Persons MODIFY Age int NOT NULL; | | * **UNIQUE** - 保证某列的每行必须有唯一的值。 【索引】   PRIMARY KEY 约束拥有自动定义的 UNIQUE 约束。  请注意，每个表可以有多个 UNIQUE 约束，但是每个表只能有一个 PRIMARY KEY 约束。  CREATE TABLE Persons  (  P\_Id int NOT NULL,  LastName varchar(255) NOT NULL,  UNIQUE (P\_Id)  )  当表已经创建时  添加 ALTER TABLE Persons ADD UNIQUE (P\_Id)  删除 ALTER TABLE Persons DROP INDEX P\_Id | | * **PRIMARY KEY** - NOT NULL 和 UNIQUE 的结合。确保某列（或两个列多个列的结合）有唯一标识，有助于更容易更快速地找到表中的一个特定的记录。   PRIMARY KEY 约束唯一标识数据库表中的每条记录。  主键必须包含唯一的值。  主键列不能包含 NULL 值。  每个表都应该有一个主键，并且每个表只能有一个主键。  CREATE TABLE Persons  (  P\_Id int NOT NULL,  LastName varchar(255) NOT NULL,  PRIMARY KEY (P\_Id)  )  CREATE TABLE Persons ( P\_Id int NOT NULL, LastName varchar(255) NOT NULL, FirstName varchar(255), CONSTRAINT pk\_PersonID PRIMARY KEY (P\_Id,LastName) )  **注释：**在上面的实例中，只有一个主键 PRIMARY KEY（pk\_PersonID）。然而，pk\_PersonID 的值是由两个列（P\_Id 和 LastName）组成的。  当表已经创建时  添加 ALTER TABLE Persons ADD PRIMARY KEY (P\_Id)  删除（无论一列还是多列） ALTER TABLE Persons DROP PRIMARY KEY | | * **FOREIGN KEY** - 保证一个表中的数据匹配另一个表中的值的参照完整性。   CREATE TABLE Orders  (  O\_Id int NOT NULL,  OrderNo int NOT NULL,  P\_Id int,  PRIMARY KEY (O\_Id),  **FOREIGN KEY (P\_Id) REFERENCES Persons(P\_Id)**  )  命名：  CONSTRAINT fk\_PerOrders **FOREIGN KEY 。。。。。。**  添加：  ALTER TABLE Orders  ADD FOREIGN KEY (P\_Id) REFERENCES Persons(P\_Id)  删除：  ALTER TABLE Orders DROP FOREIGN KEY fk\_PerOrders | | * **CHECK** - 保证列中的值符合指定的条件。   CREATE TABLE Persons ( P\_Id int NOT NULL, LastName varchar(255) NOT NULL, CHECK (P\_Id>0) )  添加：  ALTER TABLE Persons ADD CHECK (P\_Id>0)  命名 CHECK 约束，并定义多个列的 CHECK 约束  ALTER TABLE Persons ADD CONSTRAINT chk\_Person CHECK (P\_Id>0 AND City='Sandnes')  删除：  ALTER TABLE Persons DROP CHECK chk\_Person | | * **DEFAULT** - 规定没有给列赋值时的默认值。 | |
| 范式 | **第一范式就是无重复的列**  **第二范式就是非主属性非部分（即完全）依赖于主关键字，不能出现只依赖关键字的一部分的属性。**  **第三范式就是属性不包含其它表中的非主属性。（我的理解是消除冗余，员工表中除了部门编号之外不要再有部门名称等信息了。）** |
| 连接 | |  |  | | --- | --- | | 内连接  **join**  **inner join** | * 有点像交集。 join 或者是 inner join。   select \* from table1 **join** table2 **on** table1.id=table2.id | | 外连接  left join  left outer join | * 左连接：返回左表所有行，右边对应行没有匹配的话为null。   select \* from table1 **left join** table2 **on** table1.id=table2.id   * 右连接：包含table2的所有，   select \* from table1 **right join** table2 **on** table1.id=table2.id   * 完整外连接：返回左表和右表中的所有行。当某行在另一个表中没有匹配行时，则另一个表的选择列表列包含空值   select \* from table1 full join table2 on table1.id=table2.id | | **on** 和 **where** 条件的区别 | 1、**on** 条件是在生成临时表时使用的条件，它不管 **on** 中的条件是否为真，都会返回左边表中的记录。  2、**where** 条件是在临时表生成好后，再对临时表进行过滤的条件。这时已经没有 left join 的含义（必须返回左边表的记录）了，条件不为真的就全部过滤掉。 | | 全连接  FULL OUTER JOIN | 并集  SELECT column\_name(s) FROM table1 FULL OUTER JOIN table2 ON table1.column\_name=table2.column\_name; | | 总结 | **A inner join B 取交集。**  **A left join B 取 A 全部，B 没有对应的值为 null。**  **A right join B 取 B 全部 A 没有对应的值为 null。**  **A full outer join B 取并集，彼此没有对应的值为 null。**  **对应条件在 on 后面填写。** | |
| 表复制 |  |
| NULL 值的处理 | 在 MySQL 中，NULL 值与任何其它值的比较（即使是 NULL）永远返回 NULL，即 NULL = NULL 返回 NULL 。  MySQL 中处理 NULL 使用 IS NULL 和 IS NOT NULL 运算符。 |
| 删除 | drop直接删掉表 。 truncate删除表中数据，再插入时自增长id又从1开始 。 delete删除表中数据，可以加where字句。 |
| 临时表 | 做复杂题目的时候，可以让表结构更清晰。 |

# Sql优化

1. 避免在where子句中使用!= 或> < ，对字段进行null值判断，not in ， exists等否则将放弃索引使用全盘扫描。
2. 避免使用存储过程和触发器（pass 都是用代码来整合。）避免多表关联查询，因为这个依赖数据库去实现，高并发情况下可能导致数据库瘫痪。
3. Where条件里面必须有一个带索引。

**如何用explain分析sql执行效率？**

explain select \* from news;

输出： | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

# Mysql引擎

**InnoDB和MyIsam的区别？**

Mysql5.5之前MyIsam是默认引擎，不支持事务和行级锁。崩溃后无法恢复。

Mysql5.5之后引入了InnoDB（事务性数据库引擎），5.5之后的默认引擎为InnoDB。

1. 是否支持行级锁：Myisam只有表锁，InnoDB支持行锁和表锁。
2. 是否支持事务崩溃后的安全恢复：MyIsam强调性能，执行速度比InnoDB快，但不支持事务。InnoDB支持事务，可以回滚和崩溃修复。
3. 是否支持外键：MyIsam不支持，InnoDB支持外键。
4. 是否支持MVCC：只有InnoDB支持。
5. MyIsam 非事务安全型，InnoDB 是事务安全型。

**MVCC？多版本并发控制。**

* 1. 高并发事务，MVCC比加锁更高效。
  2. 只在Read-committed和Repeatable-read两个隔离级别下工作。
  3. 可以实用乐观锁和悲观锁来实现。
  4. 各种数据库中MVCC的实现方式不统一。

# 索引

索引是一种数据结构。目的是大大提高MySQL的检索速度。速度能有三轮和兰博那么大

|  |  |
| --- | --- |
| **普通索引** |  |
| 创建 | CREATE INDEX indexName ON table\_name (column\_name)  如果是CHAR，VARCHAR类型，length可以小于字段实际长度；如果是BLOB和TEXT类型，必须指定 length。 |
| 修改表结构时添加索引 | ALTER table tableName ADD INDEX indexName(columnName) |
| 创建表的时候指定 | CREATE TABLE mytable(  ID INT NOT NULL,  username VARCHAR(16) NOT NULL,  INDEX [indexName] (username(length))  ); |
| 删除索引 | DROP INDEX [indexName] ON mytable; |
|  |  |
| **唯一索引** | 索引列的值必须唯一，但允许有空值  如果是组合索引，则列值的组合必须唯一 |
| 创建 | CREATE UNIQUE INDEX indexName ON mytable(username(length)) |
| 修改表结构 | ALTER table mytable ADD UNIQUE [indexName] (username(length)) |
| 创建表的时候指定 | CREATE TABLE mytable(  ID INT NOT NULL,  username VARCHAR(16) NOT NULL,  UNIQUE [indexName] (username(length))  ); |
| **使用alter** |  |
| 添加索引 | * **ALTER TABLE tbl\_name ADD PRIMARY KEY (column\_list):** 该语句添加一个主键，这意味着索引值必须是唯一的，且不能为NULL。 * **ALTER TABLE tbl\_name ADD UNIQUE index\_name (column\_list):** 这条语句创建索引的值必须是唯一的（除了NULL外，NULL可能会出现多次）。 * **ALTER TABLE tbl\_name ADD INDEX index\_name (column\_list):** 添加普通索引，索引值可出现多次。 * **ALTER TABLE tbl\_name ADD FULLTEXT index\_name (column\_list):**该语句指定了索引为 FULLTEXT ，用于全文索引。   mysql> ALTER TABLE testalter\_tbl ADD INDEX (c); |
| 删除索引 | mysql> ALTER TABLE testalter\_tbl DROP INDEX c; 使用drop |
| 添加和删除主键 | 主键默认是 not null  mysql> ALTER TABLE testalter\_tbl MODIFY i INT NOT NULL;  mysql> ALTER TABLE testalter\_tbl ADD PRIMARY KEY (i);  删除主键  mysql> ALTER TABLE testalter\_tbl DROP PRIMARY KEY;  【注意，删除主键只需指定 primary key。 但删除索引必须知道索引名】 |
|  |  |

**为什么使用索引？**

加快数据的检索速度。

创建唯一索引，保证数据库表每一行数据的唯一性。

可以将随机IO变为顺序IO

帮助服务器避免排序和临时表

**为何不对每个字段都创建索引？**

1. 创建和维护索引（增删数据索引都要改）很耗费时间。
2. 索引占用物理空间。

**辅助索引：**

常用的字段添加的索引，并不是主键。

**索引是如何提升查询速度的？**

将无序的数据变成相对有序的数据（目录一样）

**索引失效的情况？**

1. 条件中有 or 即使有条件带索引也不会使用。除非or条件中的每个列都加索引。
2. 多列索引，不是使用的第一部分，则不会使用索引（最左前缀原则）
3. Like查询以%开头，索引失效。
4. 条件类型是字符串，条件需用引号括起来。不括起来需要类型转换，类型转换索引失效。
5. 索引列使用了函数，索引失效。
6. 当Mysql觉得全表扫描比使用索引快，则索引失效。比如数据少的时候。

使用explain + 实际SQL的方式判断索引是否生效 看type是否等于 ref

使用explain + 实际SQL的方式查看多表执行过程

**什么是聚簇索引和非聚簇索引？**

聚簇索引：数据与索引在一起，找到索引即找到数据。

非~~~~：索引结构的叶子节点指向数据行。（MyInsm存地址，InnoDB存主键）

InnoDB中，在聚簇索引上面创建的是辅助索引。访问数据需要（根据主键）二次查找。InnoDB的非聚簇索引都是辅助索引，复合索引、前缀索引、唯一索引、辅助索引的叶子节点存储的不再是行的物理位置，而是主键值。

Mysql中默认设置主键为聚簇索引。没有主键选择非空索引，没有非空索引，隐式定义主键作为聚簇索引。

**索引使用的数据结构？**

哈希索引：哈希表，单条记录查询性能最快。应用场景比较少。

有序数组：等值查询和范围查询性能优秀，二分查找可快速找到O（logN）,但是插入的数据成本高。所以有序数组适用于静态存储。

BTree索引：Mysql的BTree索引使用的B+Tree结构，MyIsam和InnoDB两种引擎的实现方式不同。

**MyISAM和InnoDB实现Btree索引方式的区别？**

MyISAM是非聚簇索引，树的叶子data域存的是地址。指向数据行。

InnoDB数据文件本身就是索引文件。叶子存着完整的数据。索引的key就是主键，因此InnoDB数据文件本身就是主索引，是聚簇索引。其他索引都是辅助索引，辅助索引的data域存主键而不是地址。所以根据辅助索引查找的时候，需要回表，先取主键值，再走一遍主索引。主键不能太长，最好是单调的。

**BTree与B+Tree的区别是？ 为啥B+Tree更适合做文件索引。**

BTree根据键值生成树，每个节点都带有数据。

B+Tree中，所有数据都是根据键值大小顺序存在一层叶子节点上的。非叶子节点只存key值信息。内部节点小，同样大小的盘块能容纳更多的索引结构，一次性读入内存的关键字信息更多，IO次数更少。

**什么是覆盖索引，举例？**

当索引包含所需查询字段的值，就叫覆盖索引。因为InnoDB如果走辅助索引，需要回表。覆盖索引就是把要查询的列加索引，不需要做回表操作。

创建索引（username，age）

查询sql：select username,age from user where username=”zhanghao” and age =24;

要查询的列和索引是对应的，不需要回表。

**最左前缀原则的理解？**

Mysql的索引可以以一定顺序引用多列，这种索引**叫联合索引**。最左前缀原则就是查询的时候匹配左边连续一列或几列，索引就可以被用到。（查询的时候如果顺序不一致，那么查询引擎可以自动优化为匹配联合索引的顺序。也是可以命中索引的。）

**索引下推优化（或称ICP优化）**

当进行索引查询时，**首先根据索引来查找记录，然后再根据where条件来过滤记录**；在支持ICP优化后，MySQL会在取出索引的同时，判断是否可以进行where条件过滤，也就是说提前执行where的部分过滤操作，在某些场景下，可以大大减少**回表**次数，从而提升整体性能。

# 事务

|  |  |
| --- | --- |
|  | 在 MySQL 中只有使用了 Innodb 数据库引擎的数据库或表才支持事务。 |
|  | 在 MySQL 命令行的默认设置下，事务都是自动提交的，即执行 SQL 语句后就会马上执行 COMMIT 操作  因此要显式地开启一个事务务须使用命令 BEGIN 或 START TRANSACTION，或者执行命令 SET AUTOCOMMIT=0，用来禁止使用当前会话的自动提交。 |
| 开启事务 | BEGIN 或 START TRANSACTION 显式地开启一个事务； |
| 提交 | COMMIT 也可以使用 COMMIT WORK 提交事务，并使已对数据库进行的所有修改成为永久性的 |
| 回滚 | ROLLBACK 也可以使用 ROLLBACK WORK 回滚会结束用户的事务，并撤销正在进行的所有未提交的修改 |
| 使用保留点 | 事务可以回滚到 savepoint 而不影响 savepoint 创建前的变化, 不需要放弃整个事务。  使用 SAVEPOINT  SAVEPOINT savepoint\_name; // 声明一个 savepoint  ROLLBACK TO savepoint\_name; // 回滚到savepoint  删除 SAVEPOINT  MySQL5 以来，可以用:  RELEASE SAVEPOINT savepoint\_name; // 删除指定保留点  保留点在事务处理完成（执行一条 ROLLBACK 或 COMMIT）后自动释放。 |
|  | ROLLBACK TO identifier 把事务回滚到标记点； |
| 设置隔离级别 | SET TRANSACTION 用来设置事务的隔离级别。  InnoDB 存储引擎提供事务的隔离级别有READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ 和 SERIALIZABLE。 |
| 事务测试： |  |
|  |  |
|  |  |

**什么是事务？**

事务是逻辑上的一组SQL，要么执行，要么不执行。

**事务的四个特性：**

* 原子性（Auto）：最小的单位，不可再分。要么全部完成，要么不起作用。
* 一致性（Consistency）：一个事务必须从数据库的一致状态变换到另一个一致状态。（其他三个特性都是为一致性服务的。跟并发有关。所以保证并发条件下的一致性，需要事务具有隔离性，实现隔离级别，引出了乐观锁和悲观锁。）
* 隔离性（Isolation）：并发访问数据的时候，一个事务不被其他事务干扰。
* 持久性（Durability）：事务提交之后，数据库中的数据改变是持久的。

**事务的并发问题有哪些？**

* 脏读：读取了另一个还未提交的事务中的数据，读到的是个错误数据。
* 丢失修改：事务修改的数据被其他事务修改后提交，前事务的修改便丢失了。
* 不可重复读：多次读取结果不一致，中间被其他事务修改了。（某条数据前后不一致）
* 幻读：读取到了其他事务插入的数据（数据行发生了改变）

**事务的隔离级别？**

* 读未提交**Read uncommitted**：查询语句不加锁，可能会读到未提交的行（产生脏读）
* 读已提交**Read committed**：写的时候加锁，读的时候没有阻塞update，可能前后读不一致（不可重复读）
* 可重复读**Repeatable read**：读事务阻塞其他写事务（包括update和delete 但是只给存在的行上加了锁），不包括insert，因为新行不存在，没有办法加锁。可能两次读取，行数变多了（幻读）
* 可串行化**Serializable**：事务顺序执行，读加共享锁，写加排他锁。这样读取事务可以并发，但是读写，写写事务之间都是互斥的，基本上是一个一个执行事务。

**隔离级别如何实现？**

* 一种是在读取数据前，对其加锁，阻止修改。
  + 共享锁（读锁）
  + 互斥锁（写锁）
* **MVCC 多版本并发控制**
  + 维护很多个版本的数据，可以读取旧版数据。
  + 读不加锁，读写没有冲突。 但是每行记录都要额外的存储空间。

在MVCC并发控制中，读操作可以分成两类：快照读 (snapshot read)与当前读 (current read)。

快照读，读取的是记录的可见版本 (有可能是历史版本)，不用加锁

当前读，读取的是记录的最新版本，并且，当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录。

**MySQL InnoDB中**

快照读：简单的select操作，属于快照读，不加锁。(当然，也有例外)  **select \* from table where ?;**

当前读：特殊的读操作，插入/更新/删除操作，属于当前读，需要加锁。

**select \* from table where ? lock in share mode; 对数据加 共享锁（S锁）**

**select \* from table where ? for update; 加 排它锁 （X锁）**

**insert into table values (…); 加 排它锁 （X锁）**

**update table set ? where ?; 加 排它锁 （X锁）**

**delete from table where ?; 加 排它锁 （X锁）**

**Mysql的默认隔离级别？在实际企业中一般使用的隔离级别？**

Mysql默认是可重复读。

互联网项目一般用读已提交。

不采用串行化，因为每次读都要加锁，快照读失效。性能不佳。

可重复读隔离级别下，有间隙锁，出现死锁的几率比读已提交更大。

select \* from test where id <3 for update;

可重复读，会锁住间隙 防止其他事务插入。

读已提交，不存在间隙锁，其他事务可以插入。

可重复读隔离级别下，条件列未命中索引会锁表。在读已提交，只锁行。

**项目中用读已提交，那么不可重复读问题需要解决吗？**

不用解决，毕竟数据已经提交了。ORACLE的默认隔离级别就是读已提交，没见人改过。

# 锁

锁是实现事务的关键，锁可以保证事务的完整性和并发性

我们认知中的行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁，**这些锁统称为悲观锁**

**Mysql 的锁**

|  |  |
| --- | --- |
| 表级锁 | 开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高,并发度最低 |
| 行级锁 | 开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低,并发度也最高 |
| 页面锁 | 开销和加锁时间界于表锁和行锁之间；会出现死锁；  锁定粒度界于表锁和行锁之间，并发度一般 |

表级锁更适合于以查询为主，只有少量按索引条件更新数据的应用，如Web应用；

行级锁则更适合于有大量按索引条件并发更新少量不同数据，同时又有并发查询的应用，如一些在线事务处理（OLTP）系统。

**MyISAM表锁**

表级锁有两种模式：**表共享读锁（Table Read Lock）**和**表独占写锁（Table Write Lock）**

**Innodb锁**

|  |  |
| --- | --- |
| **行锁：** | 共享锁（S）：我读的时候，你可以读，但是不能写**select … lock in share mode：**  排他锁（X）：我写的时候，你不能读也不能写**select … for update**  *其实就类似于SAM的读锁和写锁*  **InnoDB行锁是通过给索引上的索引项加锁来实现的、**  **意味着 ：如果不通过索引访问数据，InnoDB会使用表锁。** |
| **表锁：** | 意向共享锁（IS）：表示事务准备给数据行加入共享锁，也就是说一个数据行加共享锁前必须先取得该表的IS锁  意向排他锁（IX）：类似上面，表示事务准备给数据行加入排他锁，说明事务在一个数据行加排他锁前必须先取得该表的IX锁。 |

这两种方式在事务(Transaction) 时，SELECT 到同一个数据表时，都必须等待其它事务数据被提交(Commit)后才会执行。而主要的不同在于LOCK IN SHARE MODE 在有一方事务要Update 同一个表单时很容易造成死锁 (所以实际项目中基本不使用共享锁S)。

如果SELECT 后面若要UPDATE 同一个表单，最好使用SELECT … for UPDATE

**间隙锁**

出现在RR隔离级别下的一种锁，目的是防止幻读，MySQL会锁住相应数据的临近范围而避免其他事务再去插入新的数据，我们把它叫做间隙锁

防止幻读，是需要间隙锁和行锁配合，锁住条件范围内的整个区间实现的。

**在使用范围条件检索并锁定记录时，InnoDB这种加锁机制会阻塞符合条件范围内键值的并发插入，这往往会造成严重的锁等待。因此，在实际应用开发中，尤其是并发插入比较多的应用，我们要尽量优化业务逻辑，尽量使用相等条件来访问更新数据，避免使用范围条件。**

所以在很多情况下，我们会直接为了避免出现间隙锁，直接放弃RR这个MySql默认隔离级别而采用RC。

**请举出可能形成数据库死锁的原因？如何能避免死锁？**

死锁就是操作系统中两个或者以上的进程因抢夺资源而造成的互相等待的现象。

原因1：事务对资源访问顺序交替。

是程序BUG造成的，在数据库多表操作的时候，应该尽量按照相同的顺序进行处理，尽量避免同时锁住两个资源，必须同时锁住两个资源时，要保证任何时刻都应该按照相同的顺序来锁定资源。

原因2：并发修改同一条记录。[一个离奇的死锁](http://www.bimowu.com/kstudy-web/clienttopic/insSelfView.do?type=javaweb&topicId=784#%E5%8F%A6%E4%B8%80%E8%B5%B7%E5%9F%BA%E4%BA%8E%E7%B4%A2%E5%BC%95%E7%9A%84%E5%8A%A0%E9%94%81%E5%92%8C%E6%AD%BB%E9%94%81%E9%97%AE%E9%A2%98)

使用乐观锁，写-写并发。？？？

使用悲观锁，依赖数据库的锁机制。

原因3：索引不当导致死锁

当全盘扫描次数过多时（全盘扫描将行级锁上升为了表级锁数据量非常庞大，而索引建的过少或者不合适都会发生全盘扫描，最终发生阻塞或死锁。）

解决： SQL语句中不要使用太复杂的关联度表查询，对于有全盘扫描的SQL语句，应该建立相应的索引进行优化。

**如何避免死锁？**

以固定的顺序访问表和行，这样就不会造成互相等待，按顺序申请锁。

尽量将大事物拆成小事物，大的事物会更容易死锁

在同一个事物中尽可能一次性锁定所需要的所有资源。

降低隔离级别，从可重复读变成读已提交，可以避免很多因为间隙锁造成的死锁。

为表添加合理的索引，如果不走索引，将会为表的每一行记录都添加上否字符的概率会大大增加。

**数据库中的乐观锁和悲观锁有什么区别？各适用于什么场景？**

悲观所就是很悲观的，每次去拿数据都会认为别人会修改，所以每次拿数据的时候都会上锁，这样别人想拿到这个数据就会阻塞，直到他拿到锁，比如传统的行锁表所读所写所都是在操作之前先上锁。

乐观锁很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在这期间别人有没有去更新这个数据，可以使用版本号等机制，它适用于多读的类型，这样可以提高吞吐量。

* 响应速度：如果需要非常高的响应速度，建议采用乐观锁方案，成功就执行，不成功就失败，不需要等待其他并发去释放锁
* 冲突频率：如果冲突频率非常高，建议采用悲观锁，保证成功率，如果冲突频率大，乐观锁会需要多次重试才能成功，代价比较大
* 重试代价：如果重试代价大，建议采用悲观锁

乐观锁适用于写比较少的情况，省去了锁的开销，加大了系统的吞吐量，响应速度快。

如果经常产生冲突，上层的应用会不断的进行重试，这样反倒降低了性能，这种情况下用悲观锁就比较合适。如果重试代价大的话建议采用悲观锁。

**数据库中的乐观锁和悲观锁的具体应用。**

悲观锁：

首先关闭Mysql数据库的自动提交属性。 当执行一个更新操作后，my circle会立刻将结果进行提交。想要使用事物的话就要设置为非auto commit的模式。

|  |
| --- |
| *set autocommit=0;*  *//设置完autocommit后，我们就可以执行我们的正常业务了。具体如下：*  *//开始事务*  *begin;/begin work;/start transaction; (三者选一就可以)*  *//查询出商品信息*  *select status from items where id=10000 for update;*  *//根据商品信息生成订单*  *insert into orders (id,item\_id) values (null,10000);*  *//修改商品status为2*  *update items set status=2 where id=10000;*  *//提交事务*  *commit;* |

从begin开始到commit结束是一个事务。

乐观锁：

加一个version字段，每次修改version++

|  |
| --- |
| *//查询出商品信息*  *select (status,version) from items where id=#{id}*  *//修改商品status为2，表示已被下单。*  *update items set status=2,version=version+1 where id=#{id} and version=#{version};*  *//以库存数作为乐观锁,只要保证库存数大于0即可。*  *update item set*  *quantity=quantity-#sub\_quantity#*  *where*  *item\_id = #id#*  *and quantity-#sub\_quantity# > 0* |

# 日志

**事务日志：**  
基于InnoDB存储引擎的MySQL之所以可以从崩溃中恢复，正是依赖于事务日志，当数据库实例宕机后，重启时MySQL会自行检查事务日志，然后依次处理；  
事务日志分为redo log和undo log两种：  
(1)、对于事务日志中未正常提交的事务，则会记录到**undo log**中，因为事务未正确执行完，因此必须回滚，从而保证数据一致性  
(2)、对于事务日志中已正常提交但未同步到持久化存储上时，则会记录到**redo log**中，因此MySQL会重新执行一遍事务，然后让数据存储到磁盘上，从而保证数据一致性