**MySQL**

**SQL基础**

**一、数据库操作**

**创建数据库**

(1)CREATE DATABASE 数据库名;

(2)CREATE DATABASE IF NOT EXISTS数据库名DEFAULT CHARSET utf8 COLLATE utf8\_general\_ci;

创建数据库，该命令的作用：

 1. 如果数据库不存在则创建，存在则不创建。

 2. 创建数据库，并设定编码集为utf8

**drop 命令删除数据库**

drop database <数据库名>;

**使用数据库**

use数据库;

**切换数据库**

Database changed

**二、创建表**

CREATE TABLE mytable (

# int 类型，不为空，自增

id INT NOT NULL AUTO\_INCREMENT,

# int 类型，不可为空，默认值为 1，不为空

col1 INT NOT NULL DEFAULT 1,

# 变长字符串类型，最长为 45 个字符，可以为空

col2 VARCHAR(45) NULL,

# 日期类型，可为空

col3 DATE NULL,

# 设置主键为 id

PRIMARY KEY (`id`)

# ENGINE 设置存储引擎，CHARSET设置编码。

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

**三、修改表**

添加列

ALTER TABLE mytable ADD col CHAR(20);

删除列

ALTER TABLE mytable DROP COLUMN col;

删除表

DROP TABLE mytable;

**四、插入**

普通插入

INSERT INTO mytable(col1, col2) VALUES(val1, val2);

如果添加过主键自增（PRINARY KEY AUTO\_INCREMENT）第一列在增加数据的时候，可以写为0或者null，这样添加数据可以自增， 从而可以添加全部数据，而不用特意规定那几列添加数据。

插入检索出来的数据

INSERT INTO mytable1(col1, col2) SELECT col1, col2 FROM mytable2;

将一个表的内容插入到一个新表

CREATE TABLE newtable AS SELECT \* FROM mytable;

**五、更新**

UPDATE mytable SET col = val WHERE id = 1;

**六、删除**

DELETE FROM mytable WHERE id = 1;

**TRUNCATE TABLE** 可以清空表，也就是删除所有行。

TRUNCATE TABLE mytable;

使用更新和删除操作时一定要用 WHERE 子句，不然会把整张表的数据都破坏。可以先用 SELECT 语句进行测试，防止错误删除。

**七、查询**

**DISTINCT**

相同值只会出现一次。它作用于所有列，也就是说所有列的值都相同才算相同。

SELECT DISTINCT col1, col2 FROM mytable;

**LIMIT**

限制返回的行数。可以有两个参数，第一个参数为起始行，从 0 开始；第二个参数为返回的总行数。

返回前 5 行：

SELECT \* FROM mytable LIMIT 5;

SELECT \* FROM mytable LIMIT 0, 5;

返回第 3 ~ 5 行：

SELECT \*

FROM mytable

LIMIT 2, 3;

**八、排序**

* **ASC** ：升序（默认）
* **DESC** ：降序

可以按多个列进行排序，并且为每个列指定不同的排序方式：

SELECT \* FROM mytable ORDER BY col1 DESC, col2 ASC;

**九、过滤**

不进行过滤的数据非常大，导致通过网络传输了多余的数据，从而浪费了网络带宽。因此尽量使用 SQL 语句来过滤不必要的数据，而不是传输所有的数据到客户端中然后由客户端进行过滤。

SELECT \* FROM mytable WHERE col IS NULL;

下表显示了 WHERE 子句可用的操作符

| **操作符** | **说明** |
| --- | --- |
| = | 等于 |
| < | 小于 |
| > | 大于 |
| <> != | 不等于 |
| <= !> | 小于等于 |
| >= !< | 大于等于 |
| BETWEEN | 在两个值之间 |
| IS NULL | 为 NULL 值 |

应该注意到，NULL 与 0、空字符串都不同。

**AND 和 OR** 用于连接多个过滤条件。优先处理 AND，当一个过滤表达式涉及到多个 AND 和 OR 时，可以使用 () 来决定优先级，使得优先级关系更清晰。

**IN** 操作符用于匹配一组值，其后也可以接一个 SELECT 子句，从而匹配子查询得到的一组值。

**NOT** 操作符用于否定一个条件。

**十、通配符**

通配符也是用在过滤语句中，但它只能用于文本字段。

**%** 匹配 >=0 个任意字符；

**\_** 匹配 ==1 个任意字符；

**[ ]** 可以匹配集合内的字符，例如 [ab] 将匹配字符 a 或者 b。

用脱字符 ^ 可以对其进行否定，也就是不匹配集合内的字符。

使用 Like 来进行通配符匹配。

SELECT \*

FROM mytable

WHERE col LIKE '[^AB]%'; -- 不以 A 和 B 开头的任意文本

不要滥用通配符，通配符位于开头处匹配会非常慢。

**十一、计算字段**

在数据库服务器上完成数据的转换和格式化的工作往往比客户端上快得多，并且转换和格式化后的数据量更少的话可以减少网络通信量。

计算字段通常需要使用 **AS** 来取别名，否则输出的时候字段名为计算表达式。

SELECT col1 \* col2 AS alias FROM mytable;

**CONCAT()** 用于连接两个字段。许多数据库会使用空格把一个值填充为列宽，因此连接的结果会出现一些不必要的空格，使用 **TRIM()** 可以去除首尾空格。

SELECT CONCAT(TRIM(col1), '(', TRIM(col2), ')') AS concat\_col

FROM mytable;

**十二、函数**

各个 DBMS 的函数都是不相同的，因此不可移植，以下主要是 MySQL 的函数。

**汇总**

| **函 数** | **说 明** |
| --- | --- |
| AVG() | 返回某列的平均值 |
| COUNT() | 返回某列的行数 |
| MAX() | 返回某列的最大值 |
| MIN() | 返回某列的最小值 |
| SUM() | 返回某列值之和 |

AVG() 会忽略 NULL 行。

使用 DISTINCT 可以汇总不同的值。

SELECT AVG(DISTINCT col1) AS avg\_col

FROM mytable;

**文本处理**

| **函数** | **说明** |
| --- | --- |
| LEFT() | 左边的字符 |
| RIGHT() | 右边的字符 |
| LOWER() | 转换为小写字符 |
| UPPER() | 转换为大写字符 |
| LTRIM() | 去除左边的空格 |
| RTRIM() | 去除右边的空格 |
| LENGTH() | 长度 |
| SOUNDEX() | 转换为语音值 |

其中， **SOUNDEX()** 可以将一个字符串转换为描述其语音表示的字母数字模式。

SELECT \*

FROM mytable

WHERE SOUNDEX(col1) = SOUNDEX('apple')

**日期和时间处理**

日期格式：YYYY-MM-DD

时间格式：HH:MM:SS

| **函 数** | **说 明** |
| --- | --- |
| ADDDATE() | 增加一个日期（天、周等） |
| ADDTIME() | 增加一个时间（时、分等） |
| CURDATE() | 返回当前日期 |
| CURTIME() | 返回当前时间 |
| DATE() | 返回日期时间的日期部分 |
| DATEDIFF() | 计算两个日期之差 |
| DATE\_ADD() | 高度灵活的日期运算函数 |
| DATE\_FORMAT() | 返回一个格式化的日期或时间串 |
| DAY() | 返回一个日期的天数部分 |
| DAYOFWEEK() | 对于一个日期，返回对应的星期几 |
| HOUR() | 返回一个时间的小时部分 |
| MINUTE() | 返回一个时间的分钟部分 |
| MONTH() | 返回一个日期的月份部分 |
| NOW() | 返回当前日期和时间 |
| SECOND() | 返回一个时间的秒部分 |
| TIME() | 返回一个日期时间的时间部分 |
| YEAR() | 返回一个日期的年份部分 |

mysql> SELECT NOW();

2018-4-14 20:25:11

**数值处理**

| **函数** | **说明** |
| --- | --- |
| SIN() | 正弦 |
| COS() | 余弦 |
| TAN() | 正切 |
| ABS() | 绝对值 |
| SQRT() | 平方根 |
| MOD() | 余数 |
| EXP() | 指数 |
| PI() | 圆周率 |
| RAND() | 随机数 |

**十三、分组**

把具有相同的数据值的行放在同一组中。

可以对同一分组数据使用汇总函数进行处理，例如求分组数据的平均值等。

指定的分组字段除了能按该字段进行分组，也会自动按该字段进行排序。

SELECT col, COUNT(\*) AS num

FROM mytable

GROUP BY col;

GROUP BY 自动按分组字段进行排序，ORDER BY 也可以按汇总字段来进行排序。

SELECT col, COUNT(\*) AS num

FROM mytable

GROUP BY col

ORDER BY num;

WHERE 过滤行，HAVING 过滤分组，行过滤应当先于分组过滤。

SELECT col, COUNT(\*) AS num

FROM mytable

WHERE col > 2

GROUP BY col

HAVING num >= 2;

分组规定：

* GROUP BY 子句出现在 WHERE 子句之后，ORDER BY 子句之前；
* 除了汇总字段外，SELECT 语句中的每一字段都必须在 GROUP BY 子句中给出；
* NULL 的行会单独分为一组；
* 大多数 SQL 实现不支持 GROUP BY 列具有可变长度的数据类型。

**十四、子查询**

子查询中只能返回一个字段的数据。

可以将子查询的结果作为 WHRER 语句的过滤条件：

SELECT \*

FROM mytable1

WHERE col1 IN (SELECT col2

FROM mytable2);

下面的语句可以检索出客户的订单数量，子查询语句会对第一个查询检索出的每个客户执行一次：

SELECT cust\_name, (SELECT COUNT(\*)

FROM Orders

WHERE Orders.cust\_id = Customers.cust\_id)

AS orders\_num

FROM Customers

ORDER BY cust\_name;

**十五、连接**

连接用于连接多个表，使用 JOIN 关键字，并且条件语句使用 ON 而不是 WHERE。

连接可以替换子查询，并且比子查询的效率一般会更快。

可以用 AS 给列名、计算字段和表名取别名，给表名取别名是为了简化 SQL 语句以及连接相同表。

**内连接**

内连接又称等值连接，使用 INNER JOIN 关键字。

SELECT A.value, B.value

FROM tablea AS A INNER JOIN tableb AS B

ON A.key = B.key;

可以不明确使用 INNER JOIN，而使用普通查询并在 WHERE 中将两个表中要连接的列用等值方法连接起来。

SELECT A.value, B.value

FROM tablea AS A, tableb AS B

WHERE A.key = B.key;

**自连接**

自连接可以看成内连接的一种，只是连接的表是自身而已。

一张员工表，包含员工姓名和员工所属部门，要找出与 Jim 处在同一部门的所有员工姓名。

子查询版本

SELECT name

FROM employee

WHERE department = (

SELECT department

FROM employee

WHERE name = "Jim");

自连接版本

SELECT e1.name FROM employee AS e1 INNER JOIN employee AS e2 ON e1.department = e2.department AND e2.name = "Jim";

**自然连接**

自然连接是把同名列通过等值测试连接起来的，同名列可以有多个。

内连接和自然连接的区别：内连接提供连接的列，而自然连接自动连接所有同名列。

SELECT A.value, B.value

FROM tablea AS A NATURAL JOIN tableb AS B;

**外连接**

外连接保留了没有关联的那些行。分为左外连接，右外连接以及全外连接，左外连接就是保留左表没有关联的行。

检索所有顾客的订单信息，包括还没有订单信息的顾客。

SELECT Customers.cust\_id, Orders.order\_num

FROM Customers LEFT OUTER JOIN Orders

ON Customers.cust\_id = Orders.cust\_id;

customers 表：

| **cust\_id** | **cust\_name** |
| --- | --- |
| 1 | a |
| 2 | b |
| 3 | c |

orders 表：

| **order\_id** | **cust\_id** |
| --- | --- |
| 1 | 1 |
| 2 | 1 |
| 3 | 3 |
| 4 | 3 |

结果：

| **cust\_id** | **cust\_name** | **order\_id** |
| --- | --- | --- |
| 1 | a | 1 |
| 1 | a | 2 |
| 3 | c | 3 |
| 3 | c | 4 |
| 2 | b | Null |

**十六、组合查询**

使用 **UNION** 来组合两个查询，如果第一个查询返回 M 行，第二个查询返回 N 行，那么组合查询的结果一般为 M+N 行。

每个查询必须包含相同的列、表达式和聚集函数。

默认会去除相同行，如果需要保留相同行，使用 UNION ALL。

只能包含一个 ORDER BY 子句，并且必须位于语句的最后。

SELECT col

FROM mytable

WHERE col = 1

UNION

SELECT col

FROM mytable

WHERE col =2;

**十七、视图**

视图是虚拟的表，本身不包含数据，也就不能对其进行索引操作。

对视图的操作和对普通表的操作一样。

视图具有如下好处：

简化复杂的 SQL 操作，比如复杂的连接；

只使用实际表的一部分数据；

通过只给用户访问视图的权限，保证数据的安全性；

更改数据格式和表示。

CREATE VIEW myview AS

SELECT Concat(col1, col2) AS concat\_col, col3\*col4 AS compute\_col

FROM mytable

WHERE col5 = val;

**十八、存储过程**

我们常用的操作数据库语言SQL语句在执行的时候需要要先编译，然后执行，而存储过程（Stored Procedure）是一组为了完成特定功能的SQL语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给定参数（如果该存储过程带有参数）来调用执行它。

一个存储过程是一个可编程的函数，它在数据库中创建并保存。它可以有SQL语句和一些特殊的控制结构组成。当希望在不同的应用程序或平台上执行相同的函数，或者封装特定功能时，存储过程是非常有用的。数据库中的存储过程可以看做是对编程中面向对象方法的模拟。它允许控制数据的访问方式。

优点：

(1).存储过程增强了SQL语言的功能和灵活性。存储过程可以用流控制语句编写，有很强的灵活性，可以完成复杂的判断和较复杂的运算。

(2).存储过程允许标准组件是编程。存储过程被创建后，可以在程序中被多次调用，而不必重新编写该存储过程的SQL语句。而且数据库专业人员可以随时对存储过程进行修改，对应用程序源代码毫无影响。

(3).存储过程能实现较快的执行速度。如果某一操作包含大量的Transaction-SQL代码或分别被多次执行，那么存储过程要比批处理的执行速度快很多。因为存储过程是预编译的。在首次运行一个存储过程时查询，优化器对其进行分析优化，并且给出最终被存储在系统表中的执行计划。而批处理的Transaction-SQL语句在每次运行时都要进行编译和优化，速度相对要慢一些。

(4).存储过程能过减少网络流量。针对同一个数据库对象的操作（如查询、修改），如果这一操作所涉及的Transaction-SQL语句被组织程存储过程，那么当在客户计算机上调用该存储过程时，网络中传送的只是该调用语句，从而大大增加了网络流量并降低了网络负载。

(5).存储过程可被作为一种安全机制来充分利用。系统管理员通过执行某一存储过程的权限进行限制，能够实现对相应的数据的访问权限的限制，避免了非授权用户对数据的访问，保证了数据的安全。

存储过程可以看成是对一系列 SQL 操作的批处理。

使用存储过程的好处：

* 代码封装，保证了一定的安全性；
* 代码复用；
* 由于是预先编译，因此具有很高的性能。

命令行中创建存储过程需要自定义分隔符，因为命令行是以 ; 为结束符，而存储过程中也包含了分号，因此会错误把这部分分号当成是结束符，造成语法错误。

包含 in、out 和 inout 三种参数。

给变量赋值都需要用 select into 语句。

每次只能给一个变量赋值，不支持集合的操作。

delimiter //

create procedure myprocedure( out ret int )

begin

declare y int;

select sum(col1)

from mytable

into y;

select y\*y into ret;

end //

delimiter ;

call myprocedure(@ret);

select @ret;

**十九、游标**

在存储过程中使用游标可以对一个结果集进行移动遍历。

游标主要用于交互式应用，其中用户需要对数据集中的任意行进行浏览和修改。

使用游标的四个步骤：

1. 声明游标，这个过程没有实际检索出数据；
2. 打开游标；
3. 取出数据；
4. 关闭游标；

delimiter //

create procedure myprocedure(out ret int)

begin

declare done boolean default 0;

declare mycursor cursor for

select col1 from mytable;

# 定义了一个 continue handler，当 sqlstate '02000' 这个条件出现时，会执行 set done = 1

declare continue handler for sqlstate '02000' set done = 1;

open mycursor;

repeat

fetch mycursor into ret;

select ret;

until done end repeat;

close mycursor;

end //

delimiter ;

**二十、触发器**

触发器会在某个表执行以下语句时而自动执行：DELETE、INSERT、UPDATE。

触发器必须指定在语句执行之前还是之后自动执行，之前执行使用 BEFORE 关键字，之后执行使用 AFTER 关键字。BEFORE 用于数据验证和净化，AFTER 用于审计跟踪，将修改记录到另外一张表中。

INSERT 触发器包含一个名为 NEW 的虚拟表。

CREATE TRIGGER mytrigger AFTER INSERT ON mytable

FOR EACH ROW SELECT NEW.col into @result;

SELECT @result; -- 获取结果

DELETE 触发器包含一个名为 OLD 的虚拟表，并且是只读的。

UPDATE 触发器包含一个名为 NEW 和一个名为 OLD 的虚拟表，其中 NEW 是可以被修改的，而 OLD 是只读的。

MySQL 不允许在触发器中使用 CALL 语句，也就是不能调用存储过程。

**二十一、事务管理**

基本术语：

* 事务（transaction）指一组 SQL 语句；
* 回退（rollback）指撤销指定 SQL 语句的过程；
* 提交（commit）指将未存储的 SQL 语句结果写入数据库表；
* 保留点（savepoint）指事务处理中设置的临时占位符（placeholder），你可以对它发布回退（与回退整个事务处理不同）。

不能回退 SELECT 语句，回退 SELECT 语句也没意义；也不能回退 CREATE 和 DROP 语句。

MySQL 的事务提交默认是隐式提交，每执行一条语句就把这条语句当成一个事务然后进行提交。当出现 START TRANSACTION 语句时，会关闭隐式提交；当 COMMIT 或 ROLLBACK 语句执行后，事务会自动关闭，重新恢复隐式提交。

设置 autocommit 为 0 可以取消自动提交；autocommit 标记是针对每个连接而不是针对服务器的。

如果没有设置保留点，ROLLBACK 会回退到 START TRANSACTION 语句处；如果设置了保留点，并且在 ROLLBACK 中指定该保留点，则会回退到该保留点。

START TRANSACTION

// ...

SAVEPOINT delete1

// ...

ROLLBACK TO delete1

// ...

COMMIT

**二十二、字符集**

基本术语：

* 字符集为字母和符号的集合；
* 编码为某个字符集成员的内部表示；
* 校对字符指定如何比较，主要用于排序和分组。

除了给表指定字符集和校对外，也可以给列指定：

CREATE TABLE mytable

(col VARCHAR(10) CHARACTER SET latin COLLATE latin1\_general\_ci )

DEFAULT CHARACTER SET hebrew COLLATE hebrew\_general\_ci;

可以在排序、分组时指定校对：

SELECT \* FROM mytable

ORDER BY col COLLATE latin1\_general\_ci;

**二十三、权限管理**

MySQL 的账户信息保存在 mysql 这个数据库中。

USE mysql;

SELECT user FROM user;

**创建账户**

新创建的账户没有任何权限。

CREATE USER myuser IDENTIFIED BY 'mypassword';

**修改账户名**

RENAME USER myuser TO newuser;

**删除账户**

DROP USER myuser;

**查看权限**

SHOW GRANTS FOR myuser;

**授予权限**

账户用 username@host 的形式定义，username@% 使用的是默认主机名。

GRANT SELECT, INSERT ON mydatabase.\* TO myuser;

**删除权限**

GRANT 和 REVOKE 可在几个层次上控制访问权限：

* 整个服务器，使用 GRANT ALL 和 REVOKE ALL；
* 整个数据库，使用 ON database.\*；
* 特定的表，使用 ON database.table；
* 特定的列；
* 特定的存储过程。

REVOKE SELECT, INSERT ON mydatabase.\* FROM myuser;

**更改密码**

必须使用 Password() 函数进行加密。

SET PASSWROD FOR myuser = Password('new\_password');

**主 键：**数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。

**超 键：**在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。

**候选键：**是最小超键，即没有冗余元素的超键。

**外 键**：在一个表中存在的另一个表的主键称此表的外键

**索引的工作原理及其种类：**

**索引**

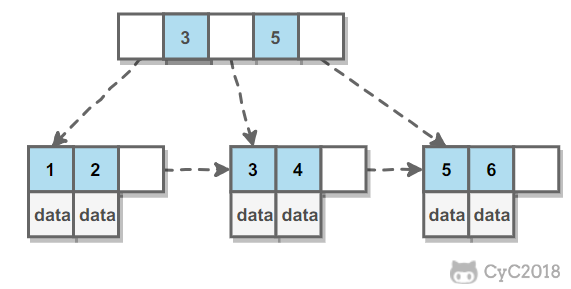
**B+ Tree 原理**

**1. 数据结构**

B Tree 指的是 Balance Tree，也就是平衡树。平衡树是一颗查找树，并且所有叶子节点位于同一层。

B+ Tree 是基于 B Tree 和叶子节点顺序访问指针进行实现，它具有 B Tree 的平衡性，并且通过顺序访问指针来提高区间查询的性能。

在 B+ Tree 中，一个节点中的 key 从左到右非递减排列，如果某个指针的左右相邻 key 分别是 keyi 和 keyi+1，且不为 null，则该指针指向节点的所有 key 大于等于 keyi 且小于等于 keyi+1。

[](https://github.com/CyC2018/CS-Notes/blob/master/notes/pics/33576849-9275-47bb-ada7-8ded5f5e7c73.png)

**2. 操作**

进行查找操作时，首先在根节点进行二分查找，找到一个 key 所在的指针，然后递归地在指针所指向的节点进行查找。直到查找到叶子节点，然后在叶子节点上进行二分查找，找出 key 所对应的 data。

插入删除操作会破坏平衡树的平衡性，因此在插入删除操作之后，需要对树进行一个分裂、合并、旋转等操作来维护平衡性。

**3. 与红黑树的比较**

红黑树等平衡树也可以用来实现索引，但是文件系统及数据库系统普遍采用 B+ Tree 作为索引结构，主要有以下两个原因：

（一）更少的查找次数

平衡树查找操作的时间复杂度和树高 h 相关，O(h)=O(logdN)，其中 d 为每个节点的出度。

红黑树的出度为 2，而 B+ Tree 的出度一般都非常大，所以红黑树的树高 h 很明显比 B+ Tree 大非常多，查找的次数也就更多。

（二）利用磁盘预读特性

为了减少磁盘 I/O 操作，磁盘往往不是严格按需读取，而是每次都会预读。预读过程中，磁盘进行顺序读取，顺序读取不需要进行磁盘寻道，并且只需要很短的磁盘旋转时间，速度会非常快。

操作系统一般将内存和磁盘分割成固定大小的块，每一块称为一页，内存与磁盘以页为单位交换数据。数据库系统将索引的一个节点的大小设置为页的大小，使得一次 I/O 就能完全载入一个节点。并且可以利用预读特性，相邻的节点也能够被预先载入。

**MySQL 索引**

索引是在存储引擎层实现的，而不是在服务器层实现的，所以不同存储引擎具有不同的索引类型和实现。

**1. B+Tree 索引**

是大多数 MySQL 存储引擎的默认索引类型。

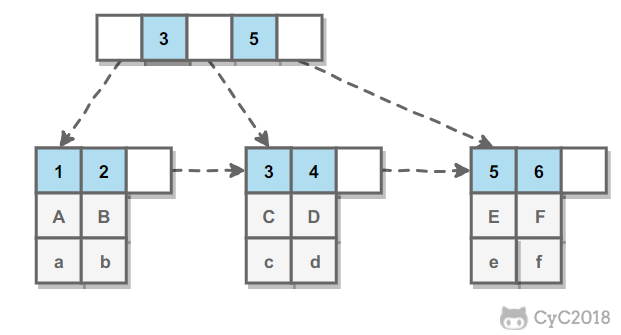
因为不再需要进行全表扫描，只需要对树进行搜索即可，所以查找速度快很多。

因为 B+ Tree 的有序性，所以除了用于查找，还可以用于排序和分组。

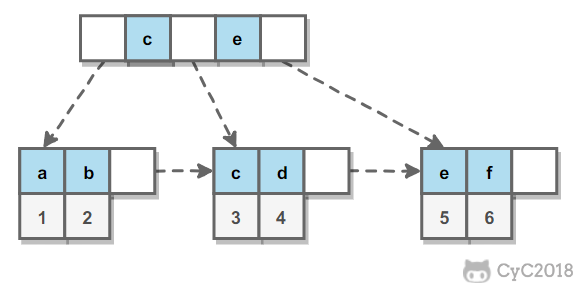
可以指定多个列作为索引列，多个索引列共同组成键。

适用于全键值、键值范围和键前缀查找，其中键前缀查找只适用于最左前缀查找。如果不是按照索引列的顺序进行查找，则无法使用索引。

InnoDB 的 B+Tree 索引分为主索引和辅助索引。主索引的叶子节点 data 域记录着完整的数据记录，这种索引方式被称为聚簇索引。因为无法把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引。

[](https://github.com/CyC2018/CS-Notes/blob/master/notes/pics/45016e98-6879-4709-8569-262b2d6d60b9.png)

辅助索引的叶子节点的 data 域记录着主键的值，因此在使用辅助索引进行查找时，需要先查找到主键值，然后再到主索引中进行查找。

[](https://github.com/CyC2018/CS-Notes/blob/master/notes/pics/7c349b91-050b-4d72-a7f8-ec86320307ea.png)

**2. 哈希索引**

哈希索引能以 O(1) 时间进行查找，但是失去了有序性：

* 无法用于排序与分组；
* 只支持精确查找，无法用于部分查找和范围查找。

InnoDB 存储引擎有一个特殊的功能叫“自适应哈希索引”，当某个索引值被使用的非常频繁时，会在 B+Tree 索引之上再创建一个哈希索引，这样就让 B+Tree 索引具有哈希索引的一些优点，比如快速的哈希查找。

**3. 全文索引**

MyISAM 存储引擎支持全文索引，用于查找文本中的关键词，而不是直接比较是否相等。

查找条件使用 MATCH AGAINST，而不是普通的 WHERE。

全文索引使用倒排索引实现，它记录着关键词到其所在文档的映射。

InnoDB 存储引擎在 MySQL 5.6.4 版本中也开始支持全文索引。

**4. 空间数据索引**

MyISAM 存储引擎支持空间数据索引（R-Tree），可以用于地理数据存储。空间数据索引会从所有维度来索引数据，可以有效地使用任意维度来进行组合查询。

必须使用 GIS 相关的函数来维护数据。

**索引优化**

**1. 独立的列**

在进行查询时，索引列不能是表达式的一部分，也不能是函数的参数，否则无法使用索引。

例如下面的查询不能使用 actor\_id 列的索引：

SELECT actor\_id FROM sakila.actor WHERE actor\_id + 1 = 5;

**2. 多列索引**

在需要使用多个列作为条件进行查询时，使用多列索引比使用多个单列索引性能更好。例如下面的语句中，最好把 actor\_id 和 film\_id 设置为多列索引。

SELECT film\_id, actor\_ id FROM sakila.film\_actor

WHERE actor\_id = 1 AND film\_id = 1;

**3. 索引列的顺序**

让选择性最强的索引列放在前面。

索引的选择性是指：不重复的索引值和记录总数的比值。最大值为 1，此时每个记录都有唯一的索引与其对应。选择性越高，每个记录的区分度越高，查询效率也越高。

例如下面显示的结果中 customer\_id 的选择性比 staff\_id 更高，因此最好把 customer\_id 列放在多列索引的前面。

SELECT COUNT(DISTINCT staff\_id)/COUNT(\*) AS staff\_id\_selectivity,

COUNT(DISTINCT customer\_id)/COUNT(\*) AS customer\_id\_selectivity,

COUNT(\*)

FROM payment;

staff\_id\_selectivity: 0.0001

customer\_id\_selectivity: 0.0373

COUNT(\*): 16049

**4. 前缀索引**

对于 BLOB、TEXT 和 VARCHAR 类型的列，必须使用前缀索引，只索引开始的部分字符。

前缀长度的选取需要根据索引选择性来确定。

**5. 覆盖索引**

索引包含所有需要查询的字段的值。

具有以下优点：

* 索引通常远小于数据行的大小，只读取索引能大大减少数据访问量。
* 一些存储引擎（例如 MyISAM）在内存中只缓存索引，而数据依赖于操作系统来缓存。因此，只访问索引可以不使用系统调用（通常比较费时）。
* 对于 InnoDB 引擎，若辅助索引能够覆盖查询，则无需访问主索引。

**索引的优点**

* 大大减少了服务器需要扫描的数据行数。
* 帮助服务器避免进行排序和分组，以及避免创建临时表（B+Tree 索引是有序的，可以用于 ORDER BY 和 GROUP BY 操作。临时表主要是在排序和分组过程中创建，不需要排序和分组，也就不需要创建临时表）。
* 将随机 I/O 变为顺序 I/O（B+Tree 索引是有序的，会将相邻的数据都存储在一起）。

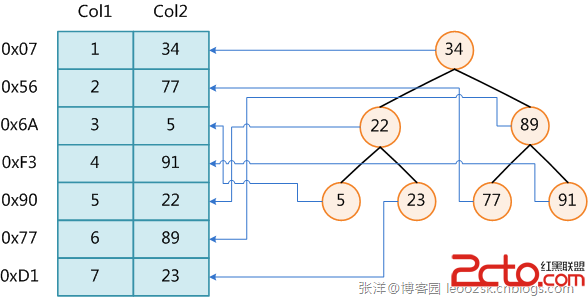
**索引的使用条件**

* 对于非常小的表、大部分情况下简单的全表扫描比建立索引更高效；
* 对于中到大型的表，索引就非常有效；
* 但是对于特大型的表，建立和维护索引的代价将会随之增长。这种情况下，需要用到一种技术可以直接区分出需要查询的一组数据，而不是一条记录一条记录地匹配，例如可以使用分区技术。

数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用B树及其变种B+树。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。



图展示了一种可能的索引方式。左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快Col2的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在O(log2n)的复杂度内获取到相应数据。

创建索引可以大大提高系统的性能。

第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？因为，增加索引也有许多不利的方面。

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。一般来说，应该在这些列上创建索引：在经常需要搜索的列上，可以加快搜索的速度；在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

同样，对于有些列不应该创建索引。一般来说，不应该创建索引的的这些列具有下列特点：

第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

第三，对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。

第四，当修改性能远远大于检索性能时，不应该创建索引。这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

根据数据库的功能，可以在数据库设计器中创建三种索引：唯一索引、主键索引和聚集索引。

**唯一索引**

唯一索引是不允许其中任何两行具有相同索引值的索引。

当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。数据库还可能防止添加将在表中创建重复键值的新数据。例如，如果在employee表中职员的姓(lname)上创建了唯一索引，则任何两个员工都不能同姓。 主键索引 数据库表经常有一列或列组合，其值唯一标识表中的每一行。该列称为表的主键。 在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。 聚集索引 在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。

如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度。

**局部性原理与磁盘预读**

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分分之一，因此为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高I/O效率。

预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

**B-/+Tree索引的性能分析**

到这里终于可以分析B-/+Tree索引的性能了。

上文说过一般使用磁盘I/O次数评价索引结构的优劣。先从B-Tree分析，根据B-Tree的定义，可知检索一次最多需要访问h个节点。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：

每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O(logdN)。一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。

而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。

综上所述，用B-Tree作为索引结构效率是非常高的。

**SQL语言包含4个部分：**

数据定义语言（如create,drop,alter等语句）

数据查询语言（select语句）

数据操纵语言（insert,delete,update语句）

数据控制语言（如grant,revoke,commit,rollback等语句）

数据操纵语言针对表中的数据，而数据定义语言针对数据库或表

MySQL 5.0 以上的版本：

1、一个汉字占多少长度与编码有关：

UTF－8：一个汉字＝3个字节

GBK：一个汉字＝2个字节

2、varchar(n) 表示 n 个字符，无论汉字和英文，Mysql 都能存入 n 个字符，仅是实际字节长度有所区别

3、MySQL 检查长度，可用 SQL 语言来查看：

select LENGTH(fieldname) from tablename

**数据库存储引擎**

数据库存储引擎是数据库底层软件组织，数据库管理系统（DBMS）使用数据引擎进行创建、查询、更新和删除数据。不同的存储引擎提供不同的存储机制、索引技巧、锁定水平等功能，使用不同的存储引擎，还可以 获得特定的功能。现在许多不同的数据库管理系统都支持多种不同的数据引擎。MySQL的核心就是存储引擎。

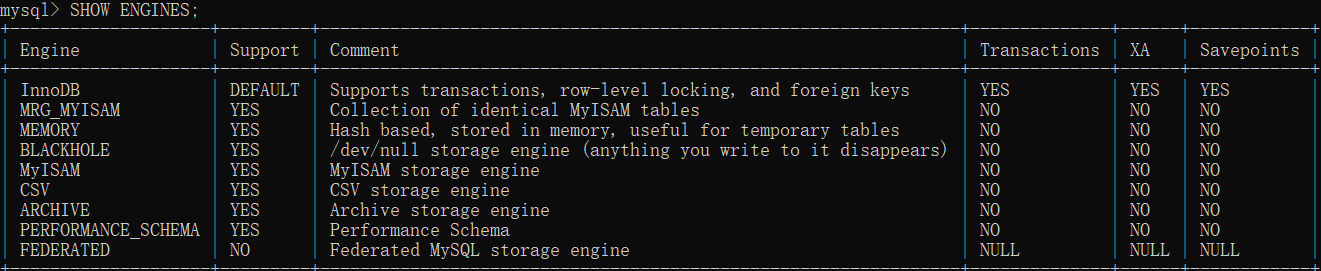
**1. MyIsam ,2. Mrg\_Myisam, 3. Memory, 4. Blackhole,5. CSV,**

**6. Performance\_Schema, 7. Archive, 8. Federated , 9.InnoDB**

**存储引擎查看**：

SHOW ENGINES

命令来查看MySQL使用的引擎，命令的输出为：



看到MySQL给用户提供了这么多存储引擎，包括处理事务安全表的引擎和出来了非事物安全表的引擎。

如果要想查看数据库默认使用哪个引擎，可以通过使用命令：

SHOW VARIABLES LIKE 'storage\_engine';

在MySQL中，不需要在整个服务器中使用同一种存储引擎，针对具体的要求，可以对每一个表使用不同的存储引擎。Support列的值表示某种引擎是否能使用：YES表示可以使用、NO表示不能使用、DEFAULT表示该引擎为当前默认的存储引擎 。下面来看一下其中几种常用的引擎。

**InnoDB存储引擎**

InnoDB是事务型数据库的首选引擎，支持事务安全表（ACID），支持行锁定和外键，上图也看到了，InnoDB是默认的MySQL引擎。InnoDB主要特性有：

1、InnoDB给MySQL提供了具有提交、回滚和崩溃恢复能力的事物安全（ACID兼容）存储引擎。InnoDB锁定在行级并且也在SELECT语句中提供一个类似[**Oracle**](http://lib.csdn.net/base/oracle)的非锁定读。这些功能增加了多用户部署和性能。在SQL查询中，可以自由地将InnoDB类型的表和其他MySQL的表类型混合起来，甚至在同一个查询中也可以混合

2、InnoDB是为处理巨[**大数据**](http://lib.csdn.net/base/hadoop)量的最大性能设计。它的CPU效率可能是任何其他基于磁盘的关系型数据库引擎锁不能匹敌的

3、InnoDB存储引擎完全与MySQL服务器整合，InnoDB存储引擎为在主内存中缓存数据和索引而维持它自己的缓冲池。InnoDB将它的表和索引在一个逻辑表空间中，表空间可以包含数个文件（或原始磁盘文件）。这与MyISAM表不同，比如在MyISAM表中每个表被存放在分离的文件中。InnoDB表可以是任何尺寸，即使在文件尺寸被限制为2GB的[**操作系统**](http://lib.csdn.net/base/operatingsystem)上

4、InnoDB支持外键完整性约束，存储表中的数据时，每张表的存储都按主键顺序存放，如果没有显示在表定义时指定主键，InnoDB会为每一行生成一个6字节的ROWID，并以此作为主键

5、InnoDB被用在众多需要高性能的大型数据库站点上

InnoDB不创建目录，使用InnoDB时，MySQL将在MySQL数据目录下创建一个名为ibdata1的10MB大小的自动扩展数据文件，以及两个名为ib\_logfile0和ib\_logfile1的5MB大小的日志文件

**MyISAM存储引擎**

MyISAM基于ISAM存储引擎，并对其进行扩展。它是在Web、数据仓储和其他应用环境下最常使用的存储引擎之一。MyISAM拥有较高的插入、查询速度，但**不支持事物**。MyISAM主要特性有：

1、大文件（达到63位文件长度）在支持大文件的文件系统和操作系统上被支持

2、当把删除和更新及插入操作混合使用的时候，动态尺寸的行产生更少碎片。这要通过合并相邻被删除的块，以及若下一个块被删除，就扩展到下一块自动完成

3、每个MyISAM表最大索引数是64，这可以通过重新编译来改变。每个索引最大的列数是16

4、最大的键长度是1000字节，这也可以通过编译来改变，对于键长度超过250字节的情况，一个超过1024字节的键将被用上

5、BLOB和TEXT列可以被索引

6、NULL被允许在索引的列中，这个值占每个键的0~1个字节

7、所有数字键值以高字节优先被存储以允许一个更高的索引压缩

8、每个MyISAM类型的表都有一个AUTO\_INCREMENT的内部列，当INSERT和UPDATE操作的时候该列被更新，同时AUTO\_INCREMENT列将被刷新。所以说，MyISAM类型表的AUTO\_INCREMENT列更新比InnoDB类型的AUTO\_INCREMENT更快

9、可以把数据文件和索引文件放在不同目录

10、每个字符列可以有不同的字符集

11、有VARCHAR的表可以固定或动态记录长度

12、VARCHAR和CHAR列可以多达64KB

使用MyISAM引擎创建数据库，将产生3个文件。文件的名字以表名字开始，扩展名之处文件类型：frm文件存储表定义、数据文件的扩展名为.MYD（MYData）、索引文件的扩展名时.MYI（MYIndex）

**MEMORY存储引擎**

MEMORY存储引擎将表中的数据存储到内存中，未查询和引用其他表数据提供快速访问。MEMORY主要特性有：

1、MEMORY表的每个表可以有多达32个索引，每个索引16列，以及500字节的最大键长度

2、MEMORY存储引擎执行HASH和BTREE缩影

3、可以在一个MEMORY表中有非唯一键值

4、MEMORY表使用一个固定的记录长度格式

5、MEMORY不支持BLOB或TEXT列

6、MEMORY支持AUTO\_INCREMENT列和对可包含NULL值的列的索引

7、MEMORY表在所由客户端之间共享（就像其他任何非TEMPORARY表）

8、MEMORY表内存被存储在内存中，内存是MEMORY表和服务器在查询处理时的空闲中，创建的内部表共享

9、当不再需要MEMORY表的内容时，要释放被MEMORY表使用的内存，应该执行DELETE FROM或TRUNCATE TABLE，或者删除整个表（使用DROP TABLE）

**存储引擎的选择**

不同的存储引擎都有各自的特点，以适应不同的需求，如下表所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功 能 | MYISAM | Memory(Heap) | InnoDB | Archive |
| 存储限制 | 256TB | RAM | 64TB | None |
| 支持事物 | No | No | Yes | No |
| 支持全文索引 | Yes | No | No | No |
| 支持数索引 | Yes | Yes | Yes | No |
| 支持哈希索引 | No | Yes | No | No |
| 支持数据缓存 | No | N/A | Yes | No |
| 支持外键 | No | No | Yes | No |

如果要提供提交、回滚、崩溃恢复能力的事物安全（ACID兼容）能力，并要求实现并发控制，InnoDB是一个好的选择

如果数据表主要用来插入和查询记录，则MyISAM引擎能提供较高的处理效率

如果只是临时存放数据，数据量不大，并且不需要较高的数据安全性，可以选择将数据保存在内存中的Memory引擎，MySQL中使用该引擎作为临时表，存放查询的中间结果

如果只有INSERT和SELECT操作，可以选择Archive，Archive支持高并发的插入操作，但是本身不是事务安全的。Archive非常适合存储归档数据，如记录日志信息可以使用Archive

使用哪一种引擎需要灵活选择，**一个数据库中多个表可以使用不同引擎以满足各种性能和实际需求**，使用合适的存储引擎，将会提高整个数据库的性能

MyISAM与InnoDB比较

事务：InnoDB 是事务型的，可以使用 Commit 和 Rollback 语句。

并发：MyISAM 只支持表级锁，而 InnoDB 还支持行级锁。

外键：InnoDB 支持外键。

备份：InnoDB 支持在线热备份。

崩溃恢复：MyISAM 崩溃后发生损坏的概率比 InnoDB 高很多，而且恢复的速度也更慢。

其它特性：MyISAM 支持压缩表和空间数据索引。

封锁

封锁粒度

MySQL 中提供了两种封锁粒度：行级锁以及表级锁。

应该尽量只锁定需要修改的那部分数据，而不是所有的资源。锁定的数据量越少，发生锁争用的可能就越小，系统的并发程度就越高。

但是加锁需要消耗资源，锁的各种操作（包括获取锁、释放锁、以及检查锁状态）都会增加系统开销。因此封锁粒度越小，系统开销就越大。

在选择封锁粒度时，需要在锁开销和并发程度之间做一个权衡。

封锁类型

1. 读写锁

排它锁（Exclusive），简写为 X 锁，又称写锁。

共享锁（Shared），简写为 S 锁，又称读锁。

有以下两个规定：

一个事务对数据对象 A 加了 X 锁，就可以对 A 进行读取和更新。加锁期间其它事务不能对 A 加任何锁。

一个事务对数据对象 A 加了 S 锁，可以对 A 进行读取操作，但是不能进行更新操作。加锁期间其它事务能对 A 加 S 锁，但是不能加 X 锁。

锁的兼容关系如下：

|  |  |  |
| --- | --- | --- |
|  | X | S |
| X | × | × |
| S | × | √ |

2. 意向锁

使用意向锁（Intention Locks）可以更容易地支持多粒度封锁。

在存在行级锁和表级锁的情况下，事务 T 想要对表 A 加 X 锁，就需要先检测是否有其它事务对表 A 或者表 A 中的任意一行加了锁，那么就需要对表 A 的每一行都检测一次，这是非常耗时的。

意向锁在原来的 X/S 锁之上引入了 IX/IS，IX/IS 都是表锁，用来表示一个事务想要在表中的某个数据行上加 X 锁或 S 锁。有以下两个规定：

一个事务在获得某个数据行对象的 S 锁之前，必须先获得表的 IS 锁或者更强的锁；

一个事务在获得某个数据行对象的 X 锁之前，必须先获得表的 IX 锁。

通过引入意向锁，事务 T 想要对表 A 加 X 锁，只需要先检测是否有其它事务对表 A 加了 X/IX/S/IS 锁，如果加了就表示有其它事务正在使用这个表或者表中某一行的锁，因此事务 T 加 X 锁失败。

各种锁的兼容关系如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| - | X | IX | S | IS |
| X | × | × | × | × |
| IX | × | √ | × | √ |
| S | × | × | √ | √ |
| IS | × | √ | √ | √ |

解释如下：

任意 IS/IX 锁之间都是兼容的，因为它们只是表示想要对表加锁，而不是真正加锁；

S 锁只与 S 锁和 IS 锁兼容，也就是说事务 T 想要对数据行加 S 锁，其它事务可以已经获得对表或者表中的行的 S 锁。

数据库事务的四个特性及含义

数据库事务transanction正确执行的四个基本要素(ACID)。

**原子性(Atomicity)：**整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。

**一致性(Correspondence)：**在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。

**隔离性(Isolation)：**隔离状态执行事务，使它们好像是系统在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行 相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，为了防止事务操作间的混淆，必须串行化或序列化请 求，使得在同一时间仅有一个请求用于同一数据。

**持久性(Durability)：**在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

视图的作用，视图可以更改么？

视图是虚拟的表，与包含数据的表不一样，视图只包含使用时动态检索数据的查询；不包含任何列或数据。使用视图可以简化复杂的sql操作，隐藏具体的细节，保护数据；视图创建后，可以使用与表相同的方式利用它们。

视图不能被索引，也不能有关联的触发器或默认值，如果视图本身内有order by 则对视图再次order by将被覆盖。

创建视图：create view XXX as XXXXXXXXXXXXXX;

对于某些视图比如未使用联结子查询分组聚集函数Distinct Union等，是可以对其更新的，对视图的更新将对基表进行更新；但是视图主要用于简化检索，保护数据，并不用于更新，而且大部分视图都不可以更新。

**drop, delete 与truncate 的区别**

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。

（1） DELETE语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存以便进行进行回滚操作。TRUNCATE TABLE 则一次性地从表中删除所有的数据并不把单独的删除操作记录记入日志保存，删除行是不能恢复的。并且在删除的过程中不会激活与表有关的删除触发器。执行速度快。

（2） 表和索引所占空间。当表被TRUNCATE 后，这个表和索引所占用的空间会恢复到初始大小，而DELETE操作不会减少表或索引所占用的空间。drop语句将表所占用的空间全释放掉。

（3） 一般而言，drop > truncate > delete

（4） 应用范围。TRUNCATE 只能对TABLE；DELETE可以是table和view

（5） TRUNCATE 和DELETE只删除数据，而DROP则删除整个表（结构和数据）。

（6） truncate与不带where的delete ：只删除数据，而不删除表的结构（定义）drop语句将删除表的结构被依赖的约束（constrain),触发器（trigger)索引（index);依赖于该表的存储过程/函数将被保留，但其状态会变为：invalid。

（7） delete语句为DML（data maintain Language),这个操作会被放到 rollback segment中,事务提交后才生效。如果有相应的 tigger,执行的时候将被触发。

（8） truncate、drop是DLL（data define language),操作立即生效，原数据不放到 rollback segment中，不能回滚

（9） 在没有备份情况下，谨慎使用 drop 与 truncate。要删除部分数据行采用delete且注意结合where来约束影响范围。回滚段要足够大。要删除表用drop;若想保留表而将表中数据删除，如果于事务无关，用truncate即可实现。如果和事务有关，或老师想触发trigger,还是用delete。

（10） Truncate table 表名 速度快,而且效率高,因为:

truncate table 在功能上与不带 WHERE 子句的 DELETE 语句相同：二者均删除表中的全部行。但 TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少。DELETE 语句每次删除一行，并在事务日志中为所删除的每行记录一项。TRUNCATE TABLE 通过释放存储表数据所用的数据页来删除数据，并且只在事务日志中记录页的释放。

（11） TRUNCATE TABLE 删除表中的所有行，但表结构及其列、约束、索引等保持不变。新行标识所用的计数值重置为该列的种子。如果想保留标识计数值，请改用 DELETE。如果要删除表定义及其数据，请使用 DROP TABLE 语句。

（12） 对于由 FOREIGN KEY 约束引用的表，不能使用 TRUNCATE TABLE，而应使用不带 WHERE 子句的 DELETE 语句。由于 TRUNCATE TABLE 不记录在日志中，所以它不能激活触发器。

**数据库范式**

**1 第一范式（1NF）**

在任何一个关系数据库中，第一范式（1NF）是对关系模式的基本要求，不满足第一范式（1NF）的数据库就不是关系数据库。

所谓第一范式（1NF）是指数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。在第一范式（1NF）中表的每一行只包含一个实例的信息。简而言之，第一范式就是无重复的列。

**2 第二范式（2NF）**

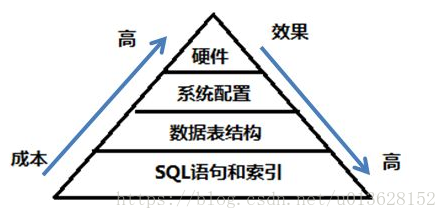
第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。第二范式（2NF）要求数据库表中的每个实例或行必须可以被惟一地区分。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。这个惟一属性列被称为主关键字或主键、主码。

第二范式（2NF）要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性，如果存在，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，新实体与原实体之间是一对多的关系。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。简而言之，第二范式就是非主属性非部分依赖于主关键字。

**3 第三范式（3NF）**

满足第三范式（3NF）必须先满足第二范式（2NF）。简而言之，第三范式（3NF）要求一个数据库表中不包含已在其它表中已包含的非主关键字信息。例如，存在一个部门信息表，其中每个部门有部门编号（dept\_id）、部门名称、部门简介等信息。那么在员工信息表中列出部门编号后就不能再将部门名称、部门简介等与部门有关的信息再加入员工信息表中。如果不存在部门信息表，则根据第三范式（3NF）也应该构建它，否则就会有大量的数据冗余。简而言之，第三范式就是属性不依赖于其它非主属性。（我的理解是消除冗余）

**数据库优化**



1. SQL以及索引的优化

首先要根据需求写出结构良好的SQL，然后根据SQL在表中建立有效的索引。但是如果索引太多，不但会影响写入的效率，对查询也有一定的影响。

2. 合理的数据库是设计

根据数据库三范式来进行表结构的设计。设计表结构时，就需要考虑如何设计才能更有效的查询。

有时候可以根据场景合理地反规范化：

A：分割表。

B：保留冗余字段。当两个或多个表在查询中经常需要连接时，可以在其中一个表上增加若干冗余的字段，以 避免表之间的连接过于频繁，一般在冗余列的数据不经常变动的情况下使用。

C：增加派生列。派生列是由表中的其它多个列的计算所得，增加派生列可以减少统计运算，在数据汇总时可以大大缩短运算时间。

数据库五大约束：

A：PRIMARY key:设置主键约束；

B：UNIQUE：设置唯一性约束，不能有重复值；

C：DEFAULT 默认值约束

D：NOT NULL：设置非空约束，该字段不能为空；

E：FOREIGN key :设置外键约束。

字段类型选择：

A：尽量使用TINYINT、SMALLINT、MEDIUM\_INT作为整数类型而非INT，如果非负则加上UNSIGNED

B：VARCHAR的长度只分配真正需要的空间

C：使用枚举或整数代替字符串类型

D：尽量使用TIMESTAMP而非DATETIME

E：单表不要有太多字段，建议在20以内

F：避免使用NULL字段，很难查询优化且占用额外索引空间

3. 系统配置的优化

例如：MySQL数据库my.cnf

4. 硬件优化

更快的IO、更多的内存。一般来说内存越大，对于数据库的操作越好。但是CPU多就不一定了，因为他并不会用到太多的CPU数量，有很多的查询都是单CPU。另外使用高的IO（SSD、RAID），但是IO并不能减少数据库锁的机制。所以说如果查询缓慢是因为数据库内部的一些锁引起的，那么硬件优化就没有什么意义。

三：优化方案

1.对查询进行优化，要尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。  
2.应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num is null

最好不要给数据库留NULL，尽可能的使用 NOT NULL填充数据库.

备注、描述、评论之类的可以设置为 NULL，其他的，最好不要使用NULL。

不要以为 NULL 不需要空间，比如：char(100) 型，在字段建立时，空间就固定了， 不管是否插入值（NULL也包含在内），都是占用 100个字符的空间的，如果是varchar这样的变长字段， null 不占用空间。  
可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：

select id from t where num = **0**  
3.应尽量避免在 where 子句中使用 != 或 <> 操作符，否则将引擎放弃使用索引而进行全表扫描。  
4.应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num=**10** or Name = 'admin'

可以这样查询：

select id from t where num = **10**

union all

select id from t where Name = 'admin'  
5.in 和 not in 也要慎用，否则会导致全表扫描，如：

select id from t where num in(**1**,**2**,**3**)

对于连续的数值，能用 between 就不要用 in 了：

select id from t where num between **1** and **3**

很多时候用 exists 代替 in 是一个好的选择：

select num from a where num in(select num from b)

用下面的语句替换：

select num from a where exists(select **1** from b where num=a.num)

6.下面的查询也将导致全表扫描：

select id from t where name like ‘%abc%’

若要提高效率，可以考虑全文检索。  
7.如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然 而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：

select id from t where num = @num

可以改为强制查询使用索引：

select id from t with(index(索引名)) where num = @num

.应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where num/**2** = **100** 应改为:select id from t where num = **100**\***2**  
9.应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where substring(name,**1**,**3**) = ’abc’ -–name以abc开头的id

select id from t where datediff(day,createdate,’**2005**-**11**-**30**′) = **0 -**–‘**2005**-**11**-**30**’ --生成的id

应改为:

select id from t where name like 'abc%'

select id from t where createdate >= '2005-11-30' and createdate < '2005-12-1'  
10.不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。  
11.在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。  
12.不要写一些没有意义的查询，如需要生成一个空表结构：

select col1,col2 into #t from t where **1**=**0**

这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：  
create table #t(…)

13.Update 语句，如果只更改1、2个字段，不要Update全部字段，否则频繁调用会引起明显的性能消耗，同时带来大量日志。  
14.对于多张大数据量（这里几百条就算大了）的表JOIN，要先分页再JOIN，否则逻辑读会很高，性能很差。  
15.select count(\*) from table；这样不带任何条件的count会引起全表扫描，并且没有任何业务意义，是一定要杜绝的。  
16.索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。  
17.应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。  
18.尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连 接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。  
19.尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。  
20.任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。  
21.尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。  
22. 避免频繁创建和删除临时表，以减少系统表资源的消耗。临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件， 最好使用导出表。  
23.在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。  
24.如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。  
25.尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。  
26.使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。  
27.与临时表一样，游标并不是不可使用。对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时 间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。  
28.在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。  
29.尽量避免大事务操作，提高系统并发能力。  
30.尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

代码优化

之所以把代码放到第一位，是因为这一点最容易引起技术人员的忽视。很多技术人员拿到一个性能优化的需求以后，言必称缓存、异步、JVM等。实际上，第一步就应该是分析相关的代码，找出相应的瓶颈，再来考虑具体的优化策略。有一些性能问题，完全是由于代码写的不合理，通过直接修改一下代码就能解决问题的，比如for循环次数过多、作了很多无谓的条件判断、相同逻辑重复多次等。

举个栗子：

一个update操作，先查询出entity，再执行update，这样无疑多了一次数据库交互。还有一个问题，update语句可能会操作一些无需更新的字段。

​

我们可以将表单中涉及到的属性，以及updateTime，updateUser等赋值到entity，直接通过pdateByPrimaryKeySelective，去update特定字段。

定位慢SQL，并优化

这是最常用、每一个技术人员都应该掌握基本的SQL调优手段（包括方法、工具、辅助系统等）。这里以MySQL为例，最常见的方式是，由自带的慢查询日志或者开源的慢查询系统定位到具体的出问题的SQL，然后使用explain、profile等工具来逐步调优，最后经过测试达到效果后上线。

SqlServer执行计划：

通过执行计划，我们能得到哪些信息：

A：哪些步骤花费的成本比较高

B：哪些步骤产生的数据量多，数据量的多少用线条的粗细表示，很直观

C：每一步执行了什么动作

具体优化手段：

A：尽量少用（或者不用）sqlserver 自带的函数

select id from t where substring(name,1,3) = ’abc’

select id from t where datediff(day,createdate,’2005-11-30′) = 0

可以这样查询：

select id from t where name like ‘abc%’

select id from t where createdate >= ‘2005-11-30’ and createdate < ‘2005-12-1’

B：连续数值条件，用BETWEEN不用IN：SELECT id FROM t WHERE num BETWEEN 1 AND 5

C：Update 语句，如果只更改1、2个字段，不要Update全部字段，否则频繁调用会引起明显的性能消耗

D：尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型

E：不建议使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。尽量避免向客户 端返回大数据量，若数据量过大，应该考虑相应需求是否合理

F：表与表之间通过一个冗余字段来关联，要比直接使用JOIN有更好的性能

G：select count(\*) from table；这样不带任何条件的count会引起全表扫描

连接池调优

我们的应用为了实现数据库连接的高效获取、对数据库连接的限流等目的，通常会采用连接池类的方案，即每一个应用节点都管理了一个到各个数据库的连接池。随着业务访问量或者数据量的增长，原有的连接池参数可能不能很好地满足需求，这个时候就需要结合当前使用连接池的原理、具体的连接池监控数据和当前的业务量作一个综合的判断，通过反复的几次调试得到最终的调优参数。

合理使用索引

索引一般情况下都是高效的。但是由于索引是以空间换时间的一种策略，索引本身在提高查询效率的同时会影响插入、更新、删除的效率，频繁写的表不宜建索引。

选择合适的索引列，选择在where，group by，order by，on从句中出现的列作为索引项，对于离散度不大的列没有必要创建索引。

主键已经是索引了，所以primay key 的主键不用再设置unique唯一索引

索引类型

主键索引 （PRIMARY KEY）

唯一索引 （UNIQUE）

普通索引 （INDEX）

组合索引 （INDEX）

全文索引 （FULLTEXT）

可以应用索引的操作符

大于等于

Between

IN

LIKE 不以 % 开头

不能应用索引的操作符

NOT IN

LIKE %\_ 开头

如何选择索引字段

A：字段出现在查询条件中，并且查询条件可以使用索引

B：通常对数字的索引和检索要比对字符串的索引和检索效率更高

C：语句执行频率高，一天会有几千次以上

D：通过字段条件可筛选的记录集很小

​​

无效索引

A：尽量不要在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描

B：应尽量避免在 where 子句中使用 != 或 <> 操作符，否则将引擎放弃使用索引而进行全表扫描。

C：应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，将导致引擎放弃使用索引而进行全表扫描

select id from t where num=10 or Name = ‘admin’

可以这样查询：

select id from t where num = 10

union

select id from t where Name = ‘admin’

union all 返回所有数据，不管是不是重复。 union会自动压缩，去除重复数据。

D：不做列运算

where age + 1 = 10，任何对列的操作都将导致表扫描，它包括数据库教程函数、计算表达式等

E：查询like，如果是 ‘%aaa’ 不会使用到索引

分表

分表方式

水平分割（按行）、垂直分割(按列)

分表场景

A： 根据经验，mysql表数据一般达到百万级别，查询效率就会很低。

B： 一张表的某些字段值比较大并且很少使用。可以将这些字段隔离成单独一张表，通过外键关联，例如考试成绩，我们通常关注分数，不关注考试详情。

水平分表策略

按时间分表：当数据有很强的实效性，例如微博的数据，可以按月分割。

按区间分表：例如用户表 1到一百万用一张表，一百万到两百万用一张表。

hash分表：通过一个原始目标id或者是名称按照一定的hash算法计算出数据存储的表名。

读写分离

当一台服务器不能满足需求时，采用读写分离【写: update/delete/add】的方式进行集群。

一台数据库支持最大连接数是有限的，如果用户的并发访问很多，一台服务器无法满足需求，可以集群处理。mysql集群处理技术最常用的就是读写分离。

主从同步：数据库最终会把数据持久化到磁盘，集群必须确保每个数据库服务器的数据是一致的。从库读主库写，从库从主库上同步数据。

读写分离：使用负载均衡实现，写操作都往主库上写，读操作往从服务器上读。

缓存

缓存分类

本地缓存：HashMap/ConcurrentHashMap、Ehcache、Guava Cache等

缓存服务：Redis/Tair/Memcache等

使用场景

短时间内相同数据重复查询多次且数据更新不频繁，这个时候可以选择先从缓存查询，查询不到再从数据库加载并回设到缓存的方式。此种场景较适合用单机缓存。

高并发查询热点数据，后端数据库不堪重负，可以用缓存来扛。

缓存作用

减轻数据库的压力，减少访问时间。

缓存选择

如果数据量小，并且不会频繁地增长又清空（这会导致频繁地垃圾回收），那么可以选择本地缓存。具体的话，如果需要一些策略的支持（比如缓存满的逐出策略），可以考虑Ehcache；如不需要，可以考虑HashMap；如需要考虑多线程并发的场景，可以考虑ConcurentHashMap。

其他情况，可以考虑缓存服务。目前从资源的投入度、可运维性、是否能动态扩容以及配套设施来考虑，我们优先考虑Tair。除非目前Tair还不能支持的场合（比如分布式锁、Hash类型的value），我们考虑用Redis。

缓存穿透

一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该去后端系统查找（比

如DB）。如果key对应的value是一定不存在的，并且对该key并发请求量很大，就会对后端系统造

成很大的压力。这就叫做缓存穿透。

对查询结果为空的情况也进行缓存，缓存时间设置短点，或者该key对应的数据insert了之后清理缓存。

缓存并发

有时候如果网站并发访问高，一个缓存如果失效，可能出现多个进程同时查询DB，同时设置缓存的情况，

如果并发确实很大，这也可能造成DB压力过大，还有缓存频繁更新的问题。

对缓存查询加锁，如果KEY不存在，就加锁，然后查DB入缓存，然后解锁；其他进程如果发现有锁就

等待，然后等解锁后返回数据或者进入DB查询。

缓存雪崩(失效)

当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统(比如DB)

带来很大压力。

不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀.

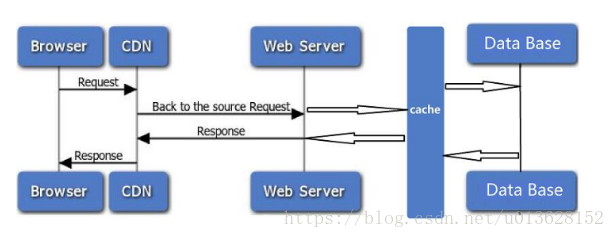
防止缓存空间不够用

① 给缓存服务，选择合适的缓存逐出算法，比如最常见的LRU。

② 针对当前设置的容量，设置适当的警戒值，比如10G的缓存，当缓存数据达到8G的时候，就开始发出报警，提前排查问题或者扩容。

③ 给一些没有必要长期保存的key，尽量设置过期时间。

我们看下图，在WebServer（Dao层）和DB之间加一层cache，这层cache一般选取的介质是内存，因为我们都知道存入数据库的数据都具有持久化的特点，那么读写会有磁盘IO的操作，内存的读写速度远比磁盘快得多。（选用存储介质，提高访问速度：内存>>磁盘；减少磁盘IO的操作，减少重复查询，提高吞吐量）



常用开源的缓存工具有：ehcache、memcache、redis。

ehcache 是一个纯Java的进程内缓存框架，hibernate使用其做二级缓存。同时，ehcache可以通过多播的方式实现集群。本人主要用于本地的缓存，数据库上层的缓存。

memcache是一套分布式的高速缓存系统，提供key-value这样简单的数据储存，可充分利用CPU多核，无持久化功能。在做web集群中可以用做session共享，页面对象缓存。

redis高性能的key-value系统，提供丰富的数据类型，单核CPU有抗并发能力，有持久化和主从复制的功能。本人主要使用redis的redis sentinel，根据不同业务分为多组。

redis注意事项

A：在增加 key 的时候尽量设置过期时间，不然 Redis Server 的内存使用会达到系统物理内存的最大值，导致 Redis 使用 VM 降低系统性能；

B：Redis Key 设计时应该尽可能短,Value 尽量不要使用复杂对象；

C：将对象转换成 JSON 对象（利用现成的 JSON 库）后存入 Redis；

D：将对象转换成 Google 开源二进制协议对象（Google Protobuf，和 JSON 数据格式类似，但是因为是二进制表现，所以性能效率以及空间占用都比 JSON 要小；缺点是 Protobuf 的学习曲线比 JSON 大得多）；

E：Redis 使用完以后一定要释放连接。

读取缓存中是否有相关数据，如果缓存中有相关数据，则直接返回，这就是所谓的数据命中“hit”

如果缓存中没有相关数据，则从数据库读取相关数据，放入缓存中，再返回。这就是所谓的数据未命中“miss”

缓存的命中率 = 命中缓存请求个数/总缓存访问请求个数 = hit/(hit+miss)

NoSQL

与缓存的区别

先说明一下，这里介绍的和缓存不一样，虽然redis等也可以用来做数据存储方案（比如Redis或者Tair），但NoSql是把它作为DB来用。如果当作DB来用，需要有效保证数据存储方案的可用性、可靠性。

使用场景

需要结合具体的业务场景，看这块业务涉及的数据是否适合用NoSQL来存储，对数据的操作方式是否适合用NoSQL的方式来操作，或者是否需要用到NoSQL的一些额外特性（比如原子加减等）。

如果业务数据不需要和其他数据作关联，不需要事务或者外键之类的支持，而且有可能写入会异常频繁，这个时候就比较适合用NoSQL（比如HBase）。

比如，美团点评内部有一个对exception做的监控系统，如果在应用系统发生严重故障的时候，可能会短时间产生大量exception数据，这个时候如果选用MySQL，会造成MySQL的瞬间写压力飙升，容易导致MySQL服务器的性能急剧恶化以及主从同步延迟之类的问题，这种场景就比较适合用Hbase类似的NoSQL来存储。

视图/存储过程

普通业务逻辑尽量不要使用存储过程，定时任务或报表统计函数可以根据团队资源情况采用存储过程处理。

GVM调优

通过监控系统（如没有现成的系统，自己做一个简单的上报监控的系统也很容易）上对一些机器关键指标（gc time、gc count、各个分代的内存大小变化、机器的Load值与CPU使用率、JVM的线程数等）的监控报警，也可以看gc log和jstat等命令的输出，再结合线上JVM进程服务的一些关键接口的性能数据和请求体验，基本上就能定位出当前的JVM是否有问题，以及是否需要调优。

异步/多线程

针对某些客户端的请求，在服务端可能需要针对这些请求做一些附属的事情，这些事情其实用户并不关心或者用户不需要立即拿到这些事情的处理结果，这种情况就比较适合用异步的方式处理这些事情。

异步作用

A：缩短接口响应时间，使用户的请求快速返回，用户体验更好。

B：避免线程长时间处于运行状态，这样会引起服务线程池的可用线程长时间不够用，进而引起线程池任务队列长度增大，从而阻塞更多请求任务，使得更多请求得不到技术处理。

C：线程长时间处于运行状态，可能还会引起系统Load、CPU使用率、机器整体性能下降等一系列问题，甚至引发雪崩。异步的思路可以在不增加机器数和CPU数的情况下，有效解决这个问题。

异步实现

A：额外开辟线程，这里可以采用额外开辟一个线程或者使用线程池的做法，在IO线程（处理请求响应）之外的线程来处理相应的任务，在IO线程中让response先返回。

B：使用消息队列（MQ）中间件服务

**存储过程与触发器的区别**

触发器与存储过程非常相似，触发器也是SQL语句集，两者唯一的区别是触发器不能用EXECUTE语句调用，而是在用户执行Transact-SQL语句时自动触发（激活）执行。触发器是在一个修改了指定表中的数据时执行的存储过程。通常通过创建触发器来强制实现不同表中的逻辑相关数据的引用完整性和一致性。由于用户不能绕过触发器，所以可以用它来强制实施复杂的业务规则，以确保数据的完整性。触发器不同于存储过程，触发器主要是通过事件执行触发而被执行的，而存储过程可以通过存储过程名称名字而直接调用。当对某一表进行诸如UPDATE、INSERT、DELETE这些操作时，SQLSERVER就会自动执行触发器所定义的SQL语句，从而确保对数据的处理必须符合这些SQL语句所定义的规则。

**sql语句分类：**

DDL：数据定义语言（create drop）

DML：数据操作语句（insert update delete）

DQL：数据查询语句（select ）

DCL：数据控制语句，进行授权和权限回收（grant revoke）

TPL：数据事务语句（commit collback savapoint）

**存储过程**

我们常用的操作数据库语言SQL语句在执行的时候需要要先编译，然后执行，而存储过程（Stored Procedure）是一组为了完成特定功能的SQL语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给定参数（如果该存储过程带有参数）来调用执行它。

一个存储过程是一个可编程的函数，它在数据库中创建并保存。它可以有SQL语句和一些特殊的控制结构组成。当希望在不同的应用程序或平台上执行相同的函数，或者封装特定功能时，存储过程是非常有用的。数据库中的存储过程可以看做是对编程中面向对象方法的模拟。它允许控制数据的访问方式。

优点：

(1).存储过程增强了SQL语言的功能和灵活性。存储过程可以用流控制语句编写，有很强的灵活性，可以完成复杂的判断和较复杂的运算。

(2).存储过程允许标准组件是编程。存储过程被创建后，可以在程序中被多次调用，而不必重新编写该存储过程的SQL语句。而且数据库专业人员可以随时对存储过程进行修改，对应用程序源代码毫无影响。

(3).存储过程能实现较快的执行速度。如果某一操作包含大量的Transaction-SQL代码或分别被多次执行，那么存储过程要比批处理的执行速度快很多。因为存储过程是预编译的。在首次运行一个存储过程时查询，优化器对其进行分析优化，并且给出最终被存储在系统表中的执行计划。而批处理的Transaction-SQL语句在每次运行时都要进行编译和优化，速度相对要慢一些。

(4).存储过程能过减少网络流量。针对同一个数据库对象的操作（如查询、修改），如果这一操作所涉及的Transaction-SQL语句被组织程存储过程，那么当在客户计算机上调用该存储过程时，网络中传送的只是该调用语句，从而大大增加了网络流量并降低了网络负载。

(5).存储过程可被作为一种安全机制来充分利用。系统管理员通过执行某一存储过程的权限进行限制，能够实现对相应的数据的访问权限的限制，避免了非授权用户对数据的访问，保证了数据的安全。

**mysql主从复制**

分类: 数据库

数据库损坏了？

主要理解为：业务不能使用数据库

外在原因：

网络问题

业务应用有问题，客户端损坏

数据库本身的原因：

物理损坏：机器坏了、硬盘坏了、存储坏了

逻辑损坏：误drop、delete、truncate、、update。

解决方案：

备份

主从复制

**MySQL复制概念**

指将主数据库的DDL和DML操作通过二进制日志传到复制服务器上，然后在复制服务器上将这些日志文件重新执行，从而使复制服务器和主服务器的数据保持同步。复制过程中一个服务器充当主服务器（master），而一个或多个其它服务器充当从服务器（slaves）。主服务器将更新重新写入二进制日志文件，并维护文件的一个索引以跟踪日志循环。这些日志可以记录发送到从服务器的更新。当一个从服务器连接主服务器时，它通知主服务器、从服务器在日志中读取的最后一次成功更新的位置。从服务器接受从那时起发生的任何更新，然后封锁并等待主服务器通知新的更新。

复制的用途：

通过主从复制（master-slave）的方式来同步数据，再通过读写分离（mysql-proxy）来提升数据库的并发负载能力，或者用来作为主备机的设计，保证在主机停止响应之后在很短的时间内就可以将应用切换到备机上继续运行。

优势：

（1）数据库集群系统具有多个数据库节点，在单个节点出现故障的情况下，其他正常节点可以继续提供服务。

（2）如果主服务器上出现了问题可以切换到从服务器上

（3）通过复制可以在从服务器上执行查询操作，降低了主服务器的访问压力，实现数据分布和负载均衡

（4）可以在从服务器上进行备份，以避免备份期间影响主服务器的服务。

**主从复制简介**

**能做什么？**

高可用

辅助备份

分担负载

**主从是怎么实现的？**

通过二进制日志

至少两台（主， 从）

主服务器的二进制日志”拿“到从服务器上再运行一遍。

通过网络连接两台机器，一般都会出现延迟的状态。也可以说是异步。

**MySQL主从复制的企业应用场景**

应用场景1：从服务器作为主服务器的实时数据备份

应用场景2：主从服务器实现读写分离，从服务器实现负载均衡

应用场景3：把多个从服务器根据业务重要性进行拆分访问

**主从复制原理**

1. 前提

主服务器一定要打开二进制日志

必须两台服务器（或者是多个实例）

从服务器需要一次数据初始化

如果主从服务器都是新搭建的话，可以不做初始化

如果主服务器已经运行很长时间了，可以通过备份将主库数据恢复到从库

主库必须要有对从库复制请求的用户。

从库需要有relay-log设置，存放从主库传过来的二进制日志

在第一次的时候，从库需要change master to 去连接主库

change mater 信息需要存放到master.info中

从库通过relay-log.info记录了已经应用过的relaylog信息发现了主库发生了新的变化

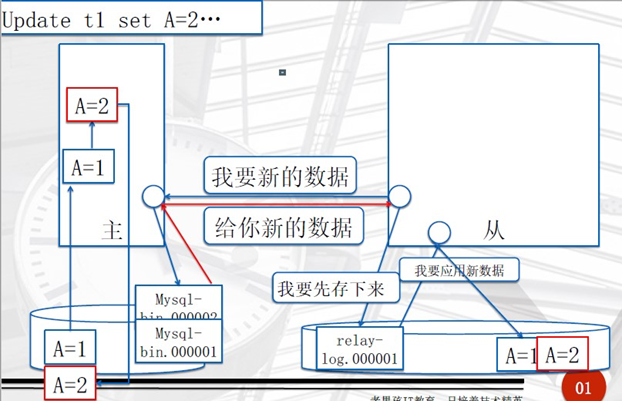
在复制过程中涉及到的线程

从库会开启一个IO thread 负责连接主库，请求binlog，接收binlog并写入relay-log中。

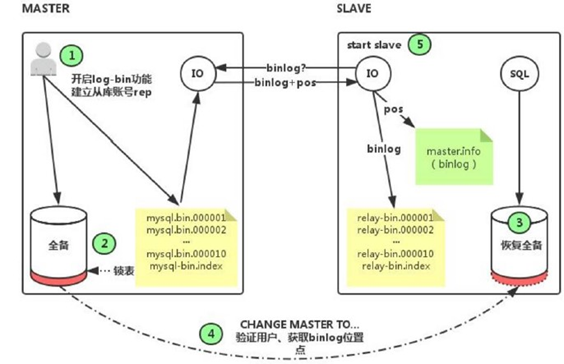
从库会开启一个SQ thread 负责执行relay-log中的事件

主库会开启一个dump thread 复制响应从IO thread中请求

**复制过程**



**复制原理**



**原理：第一次开启主从**

1.从库通过change master to 语句连接主库，并且让从库知道，二进制日志的起点位置（file名 position号）

    2.从库的IO和主库的dump线程建立连接

    3.从库根据changemaster to 语句提供的file名和position号，IO线程向主库发起binlog请求

    4.主库dump线程根据从库的请求，将本地binlog以events方式发给从库IO线程

    5.从库IO线程接收binlog events，并存放到本地relay-log中，传送过来的信息，会记录到master.info中。

    6.从库应用relay-log，并且把应用过来的记录到relay-log.info，默认情况下，已经应用过来饿的relay会自动被清理purge。

到此，一次主从复制就完成，一旦主从运行起来，就不需要手工执行changemaster to，因为信息都会被存放到master.info 中，其他的过程也是一样的。

注意：如果不是新搭建的主从，需要将原来的数据全备，进行同步到从库中，然后在开启change master to。

**数据库乐观锁和悲观锁**

悲观锁

悲观锁（Pessimistic Lock），顾名思义，就是很悲观，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block直到它拿到锁。悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。

Java synchronized 就属于悲观锁的一种实现，每次线程要修改数据时都先获得锁，保证同一时刻只有一个线程能操作数据，其他线程则会被block。

乐观锁

乐观锁（Optimistic Lock），顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在提交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。乐观锁适用于读多写少的应用场景，这样可以提高吞吐量。

乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。

乐观锁一般来说有以下2种方式：

使用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。

使用时间戳（timestamp）。乐观锁定的第二种实现方式和第一种差不多，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型使用时间戳（timestamp）, 和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则OK，否则就是版本冲突。

**事务的隔离级别**

|  |  |  |  |
| --- | --- | --- | --- |
| 隔离级别 | 脏读（Dirty Read） | 不可重复读（NonRepeatable Read） | 幻读（Phantom Read） |
| 未提交读（Read uncommitted） | 可能 | 可能 | 可能 |
| 已提交读（Read committed） | 不可能 | 可能 | 可能 |
| 可重复读（Repeatable read） | 不可能 | 不可能 | 可能 |
| 可串行化（Serializable ） | 不可能 | 不可能 | 不可能 |

**未提交读(Read Uncommitted)：**允许脏读，也就是可能读取到其他会话中未提交事务修改的数据。

**提交读(Read Committed)：**只能读取到已经提交的数据。Oracle等多数数据库默认都是该级别 (不重复读)。

**可重复读(Repeated Read)：**可重复读。在同一个事务内的查询都是事务开始时刻一致的，InnoDB默认级别。在SQL标准中，该隔离级别消除了不可重复读，但是还存在幻象读。

**串行读(Serializable)：**完全串行化的读，每次读都需要获得表级共享锁，读写相互都会阻塞。

**数据库水平切分与垂直切分**

垂直拆分就是要把表按模块划分到不同数据库表中（当然原则还是不破坏第三范式），这种拆分在大型网站的演变过程中是很常见的。当一个网站还在很小的时候，只有小量的人来开发和维护，各模块和表都在一起，当网站不断丰富和壮大的时候，也会变成多个子系统来支撑，这时就有按模块和功能把表划分出来的需求。其实，相对于垂直切分更进一步的是服务化改造，说得简单就是要把原来强耦合的系统拆分成多个弱耦合的服务，通过服务间的调用来满足业务需求看，因此表拆出来后要通过服务的形式暴露出去，而不是直接调用不同模块的表，淘宝在架构不断演变过程，最重要的一环就是服务化改造，把用户、交易、店铺、宝贝这些核心的概念抽取成独立的服务，也非常有利于进行局部的优化和治理，保障核心模块的稳定性。

垂直拆分：单表大数据量依然存在性能瓶颈

水平拆分，上面谈到垂直切分只是把表按模块划分到不同数据库，但没有解决单表大数据量的问题，而水平切分就是要把一个表按照某种规则把数据划分到不同表或数据库里。例如像计费系统，通过按时间来划分表就比较合适，因为系统都是处理某一时间段的数据。而像SaaS应用，通过按用户维度来划分数据比较合适，因为用户与用户之间的隔离的，一般不存在处理多个用户数据的情况，简单的按user\_id范围来水平切分。

通俗理解：水平拆分行，行数据拆分到不同表中， 垂直拆分列，表数据拆分到不同表中。

**InnoDB引擎的行锁是通过加在什么上实现的？**答：基于索引；

**myisam与innodb select count(\*)哪个更快，为什么**答：myisam更快，因为myisam内部维护了一个计数器，可以直接调取。

**union、union all的区别**对重复结果的处理：UNION在进行表链接后会筛选掉重复的记录，Union All不会去除重复记录；对排序的处理：Union将会按照字段的顺序进行排序；UNION ALL只是简单的将两个结果合并后就返回；从效率上说，UNION ALL 要比UNION快很多。

**索引的作用？和它的优点缺点是什么？**答：索引就一种特殊的查询表，数据库的搜索引擎可以利用它加速对数据的检索。它很类似与现实生活中书的目录，不需要查询整本书内容就可以找到想要的数据。索引可以是唯一的，创建索引允许指定单个列或者是多个列。缺点是它减慢了数据录入的速度，同时也增加了数据库的尺寸大小。

使用like 'a%' 、like'%a'、like'%a%'查询时是否会使用索引答：'a%'会，其他两个不会；

mysql怎么在查询时给查出来的数据设置一个自增的序号？set @i=0;SELECT (@i:=@i+1) 别名 FROM table, (SELECT @i:=0) AS 别名 ;

**说一说你能想到的表结构优化，至少五种**永远为每张表设置一个ID (所有建表的时候不设置主键的程序猿都应该被辞退)；选择正确的存储引擎 ;使用可存下数据的最小的数据类型，整型 < date,time < char,varchar < blob；使用简单的数据类型，整型比字符处理开销更小，因为字符串的比较更复杂。如，int类型存储时间类型，bigint类型转ip函数；使用合理的字段属性长度，固定长度的表会更快。使用enum、char而不是varchar；尽可能使用not null定义字段(给空字段设置默认值)；尽量少用text;给频繁使用和查询的字段建立合适的索引；

**索引的作用？和它的优点缺点是什么？**答：索引就一种特殊的查询表，数据库的搜索引擎可以利用它加速对数据的检索。它很类似与现实生活中书的目录，不需要查询整本书内容就可以找到想要的数据。索引可以是唯一的，创建索引允许指定单个列或者是多个列。缺点是它减慢了数据录入的速度，同时也增加了数据库的尺寸大小。

**使用索引注意事项** 索引不会包含有NULL的列，复合索引中只要有一列含有NULL值，那么这一列对于此符合索引就是无效的；使用短索引，对串列进行索引，如果可以就应该指定一个前缀长度；短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作；mysql查询只使用一个索引，因此数据库默认排序可以符合要求的情况下不要使用排序操作，尽量不要包含多个列的排序，如果需要最好给这些列建复合索引；注意like，上文已经提到；不要在列上进行运算；不使用NOT IN 、<>、！=操作，但<,<=，=，>,>=,BETWEEN,IN是可以用到索引的；索引要建立在经常进行select操作的字段上；索引要建立在值比较唯一的字段上；对于那些定义为text、image和bit数据类型的列不应该增加索引；在where和join中出现的列需要建立索引；如果where字句的查询条件里使用了函数(如：where DAY(column)=…),mysql将无法使用索引；在join操作中(需要从多个数据表提取数据时)，mysql只有在主键和外键的数据类型相同时才能使用索引，否则及时建立了索引也不会使用；

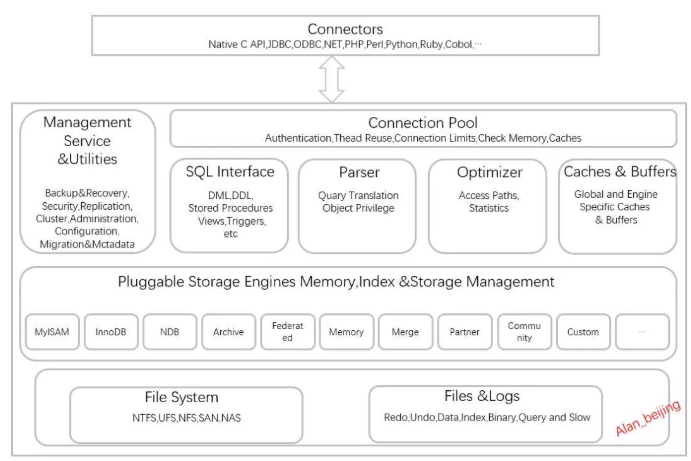
**说一说什么是外键，优缺点**答：外键指的是外键约束，目的是保持数据一致性，完整性，控制存储在外键表中的数据。使两张表形成关联，外键只能引用外表中列的值；优点：由数据库自身保证数据一致性，完整性，更可靠，因为程序很难100％保证数据的完整性，而用外键即使在数据库服务器当机或者出现其他问题的时候，也能够最大限度的保证数据的一致性和完整性。有主外键的数据库设计可以增加ER图的可读性，这点在数据库设计时非常重要。外键在一定程度上说明的业务逻辑，会使设计周到具体全面。缺点：可以用触发器或应用程序保证数据的完整性；过分强调或者说使用外键会平添开发难度，导致表过多，更改业务困难，扩展困难等问题；不用外键时数据管理简单，操作方便，性能高（导入导出等操作，在insert, update, delete 数据的时候更快）；

在什么时候你会选择使用外键，为什么答：在我的业务逻辑非常简单，业务一旦确定不会轻易更改，表结构简单，业务量小的时候我会选择使用外键。因为当不符合以上条件的时候，外键会影响业务的扩展和修改，当数据量庞大时，会严重影响增删改查的效率。

**什么叫视图？游标是什么？**视图是一种虚拟的表，具有和物理表相同的功能；可以对视图进行增，改，查，操作，视图通常是有一个表或者多个表的行或列的子集。对视图的修改不影响基本表。它使得我们获取数据更容易，相比多表查询。游标：是对查询出来的结果集作为一个单元来有效的处理。游标可以定在该单元中的特定行，从结果集的当前行检索一行或多行。可以对结果集当前行做修改。一般不使用游标，但是需要逐条处理数据的时候，游标显得十分重要。

mysql有没有rowid？答：没有，InnoDB如果没有定义主键，内部会生成一个主键编号rowid ，但是无法查询到。在平时InnoDB建表的时候我们最好自己确定主键，防止每次插入数据前数据库会去生成rowid。

**如何使用explain优化sql和索引？**explain sql ;table：显示这一行的数据是关于哪张表的；type：这是重要的列，显示连接使用了何种类型。从最好到最差的连接类型为const、eq\_reg、ref、range、index和ALL；all: full table scan ;MySQL将遍历全表以找到匹配的行；index ： index scan; index 和 all的区别在于index类型只遍历索引；range：索引范围扫描，对索引的扫描开始于某一点，返回匹配值的行，常见与between ，< ,>等查询；ref：非唯一性索引扫描，返回匹配某个单独值的所有行，常见于使用非唯一索引即唯一索引的非唯一前缀进行查找；eq\_ref：唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配，常用于主键或者唯一索引扫描；const，system：当MySQL对某查询某部分进行优化，并转为一个常量时，使用这些访问类型；如果将主键置于where列表中，MySQL就能将该查询转化为一个常量；possible\_keys：显示可能应用在这张表中的索引；如果为空，没有可能的索引；可以为相关的域从WHERE语句中选择一个合适的语句；key： 实际使用的索引；如果为NULL，则没有使用索引；很少的情况下，MySQL会选择优化不足的索引；这种情况下，可以在SELECT语句中使用USE INDEX(indexname)来强制使用一个索引或者用IGNORE INDEX(indexname)来强制MySQL忽略索引key\_len：使用的索引的长度；在不损失精确性的情况下，长度越短越好；ref：显示索引的哪一列被使用了，如果可能的话，是一个常数；rows：MySQL认为必须检查的用来返回请求数据的行数；Extra：关于MySQL如何解析查询的额外信息；



MySQL基架大致包括如下几大模块组件：

1.MySQL向外提供的交互接口（Connectors）：Connectors组件，是MySQL向外提供的交互组件，如java,.net,php等语言可以通过该组件来操作SQL语句，实现与SQL的交互。

2.管理服务组件和工具组件(Management Service & Utilities)：提供对MySQL的集成管理，如备份(Backup),恢复(Recovery),安全管理(Security)等。

3.连接池组件(Connection Pool)：负责监听对客户端向MySQL Server端的各种请求，接收请求，转发请求到目标模块。

每个成功连接MySQL Server的客户请求都会被创建或分配一个线程，该线程负责客户端与MySQL Server端的通信，接收客户端发送的命令，传递服务端的结果信息等。

4.SQL接口组件(SQL Interface)：接收用户SQL命令，如DML,DDL和存储过程等，并将最终结果返回给用户。

5.查询分析器组件(Parser)：首先分析SQL命令语法的合法性，并尝试将SQL命令分解成数据结构，若分解失败，则提示SQL语句不合理。

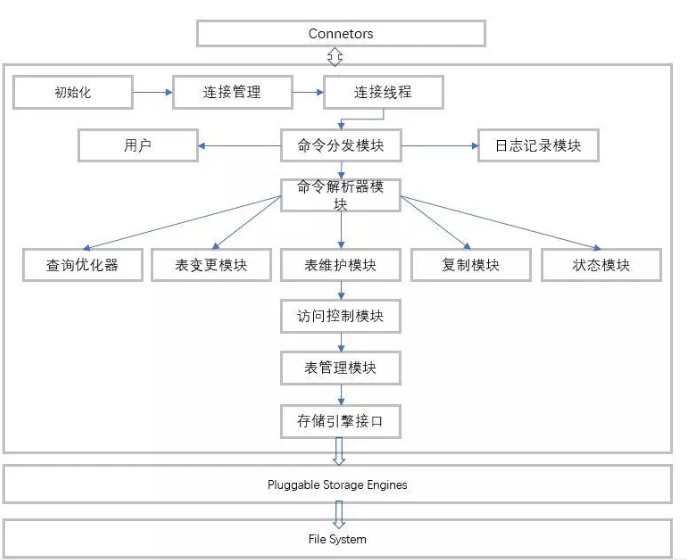
6.优化器组件（Optimizer）：对SQL命令按照标准流程进行优化分析。

7.缓存主件（Caches & Buffers）：缓存和缓冲组件

8.插件式存储引擎（Pluggable Storage Engines）

9.物理文件（File System）

**查询流程图**



**数值类型**

MySQL支持所有标准SQL数值数据类型。

这些类型包括严格数值数据类型(INTEGER、SMALLINT、DECIMAL和NUMERIC)，以及近似数值数据类型(FLOAT、REAL和DOUBLE PRECISION)。

关键字INT是INTEGER的同义词，关键字DEC是DECIMAL的同义词。

BIT数据类型保存位字段值，并且支持MyISAM、MEMORY、InnoDB和BDB表。

作为SQL标准的扩展，MySQL也支持整数类型TINYINT、MEDIUMINT和BIGINT。

**(1)数值**

|  |  |  |
| --- | --- | --- |
| 类型 | 大小 | 范围（有符号） |
| TINYINT(小整数) | 1 字节 | (-128，127) |
| SMALLINT(短整型) | 2 字节 | (-32 768，32 767) |
| MEDIUMINT(中整型) | 3 字节 | (-8 388 608，8 388 607) |
| INT或INTEGER(整形) | 4 字节 | (-2 147 483 648，2 147 483 647) |
| BIGINT(长整型) | 8 字节 | (-9,223,372,036,854,775,808，9 223 372 036 854 775 807) |
| FLOAT(浮点型) | 4 字节 | (-3.402 823 466 E+38，-1.175 494 351 E-38)，0，(1.175 494 351 E-38，3.402 823 466 351 E+38) |
| DOUBLE(双精度浮点) | 8 字节 | (-1.797 693 134 862 315 7 E+308，-2.225 073 858 507 201 4 E-308)，0，(2.225 073 858 507 201 4 E-308，1.797 693 134 862 315 7 E+308) |
| DECIMAL(小数值) | 对DECIMAL(M,D) ，如果M>D，为M+2否则为D+2 | 依赖于M和D的值 |

MySQL支持选择在该类型关键字后面的括号内指定整数值的显示宽度(例如，INT(4))。显示宽度并不限制可以在列内保存的值的范围，也不限制超过列的指定宽度的值的显示

**(2)日期和时间类型**

表示时间值的日期和时间类型为DATETIME、DATE、TIMESTAMP、TIME和YEAR。

每个时间类型有一个有效值范围和一个"零"值，当指定不合法的MySQL不能表示的值时使用"零"值。

TIMESTAMP类型有专有的自动更新特性，将在后面描述。

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 大小(字节) | 范围 | 格式 |
| DATE(日期值) | 3 | 1000-01-01/9999-12-31 | YYYY-MM-DD |
| TIME(时间值或持续时间) | 3 | '-838:59:59'/'838:59:59' | HH:MM:SS |
| YEAR(年份值) | 1 | 1901/2155 | YYYY |
| DATETIME(混合日期和时间值) | 8 | 1000-01-01 00:00:00/9999-12-31 23:59:59 | YYYY-MM-DD HH:MM:SS |
| TIMESTAMP  (混合日期和时间值，时间戳) | 4 | 1970-01-01 00:00:00/2038  结束时间是第 2147483647 秒，  北京时间 2038-1-19 11:14:07，  格林尼治时间 2038年1月19日 凌晨 03:14:07 | YYYYMMDD HHMMSS |

字符串类型

字符串类型指CHAR、VARCHAR、BINARY、VARBINARY、BLOB、TEXT、ENUM和SET。

|  |  |
| --- | --- |
| 类型 | 大小 |
| CHAR  (定长字符串) | 0-255字节 |
| TINYBLOB  (不超过 255 个字符的二进制字符串) | 0-255字节 |
| MEDIUMBLOB  (二进制形式的中等长度文本数据) | 0-16 777 215字节 |
| BLOB  (二进制形式的长文本数据) | 0-65 535字节 |
| LONGBLOB  (二进制形式的极大文本数据) | 0-4 294 967 295字节 |
| VARCHAR  (变长字符串) | 0-65535 字节 |
| TINYTEXT  (短文本字符串) | 0-255字节 |
| MEDIUMTEXT  (中等长度文本数据) | 0-16 777 215字节 |
| TEXT  (长文本数据) | 0-65 535字节 |
| LONGTEXT  (极大文本数据) | 0-4 294 967 295字节 |

**CHAR 和 VARCHAR**

CHAR的长度是不可变的，而VARCHAR的长度是可变的，相同长度，可变长度类型多占1字节。

CHAR空格补齐剩余位数。

CHAR的长度范围为0~255，VARCHAR的长度范围为0~65535

VARCHAR使用起来较为灵活，CHAR处理速度更快。

**BINARY 和 VARBINARY**

BINARY 和 VARBINARY 类似于 CHAR 和 VARCHAR，不同的是它们包含二进制字符串而不要非二进制字符串。也就是说，它们包含字节字符串而不是字符字符串。这说明它们没有字符集，并且排序和比较基于列值字节的数值值。

BLOB 是一个二进制大对象，可以容纳可变数量的数据。有 4 种 BLOB 类型：TINYBLOB、BLOB、MEDIUMBLOB 和 LONGBLOB。它们区别在于可容纳存储范围不同。

有 4 种 TEXT 类型：TINYTEXT、TEXT、MEDIUMTEXT 和 LONGTEXT。对应的这 4 种 BLOB 类型，可存储的最大长度不同，可根据实际情况选择。

**SET类型**

在创建表时，就指定SET类型的取值范围。

属性名 SET('值1','值2','值3'...,'值n')

“属性名”参数指字段的名称；“值n”参数表示列表中的第n个值，这些值末尾的空格将会被系统直接删除。

其基本形式与ENUM类型一样。SET类型的值可以取列表中的一个元素或者多个元素的组合。取多个元素时，不同元素之间用逗号隔开。SET类型的值最多只能是有64个元素构成的组合，根据成员的不同，存储上也有所不同：

1～8成员的集合，占1个字节。9～16成员的集合，占2个字节。17～24成员的集合，占3个字节。

25～32成员的集合，占4个字节。33～64成员的集合，占8个字节。

同ENUM类型一样，列表中的每个值都有一个顺序排列的编号。MySQL中存入的是这个编号，而不是列表中的值。

插入记录时，SET字段中的元素顺序无关紧要。存入MySQL数据库后，数据库系统会自动按照定义时的顺序显示。如果插入的成员中有重复，则只存储一次。

**ENUM类型**

ENUM类型(枚举类型)，与C#的概念一样，在定义时指定取值范围。

ENUM类型的值范围需要在创建表时通过枚举方式显式指定，对1～255个成员的枚举需要1个字节存储；对于256～65535个成员，需要2个字节存储。最多可以有65535个成员，而SET类型最多只能包含64个成员。两者的取值只能在成员列表中选取。

ENUM类型只能从成员中选择一个，例如 “性别”字段，而SET类型可以选择多个，例如“爱好”字段。

属性名 ENUM('值1','值2','值3'...'值n')

其中，属性名参数指字段的名称；“值n”参数表示列表中的第n个值，这些值末尾的空格将会被系统直接删除。ENUM类型的值只能取列表中的一个元素。其取值列表中最多能有65535个值。列表中的每个值都有一个顺序排列的编号，MySQL中存入的是这个编号，而不是列表中的值。

ENUM 有 NOT NULL 属性，其默认值为取值列表的第一个元素；

ENUM 无 NOT NULL，则ENUM类型将允许插入NULL，并且NULL为默认值；

CREATE TABLE Test4( Sex ENUM('男','女') );

INSERT INTO Test4 VALUES('男');

INSERT INTO Test4 VALUES('爷'); --这行报错

SELECT \* FROM Test4;

ENUM 是一个字符串对象，其值通常选自一个允许值列表中，该列表在表创建时的列规格说明中被明确地列举。

在下列某些情况下，值也可以是空串('') 或 NULL：

如果将一个无效值插入一个 ENUM (即，一个不在允许值列表中的字符串)，空字符串将作为一个特殊的错误值被插入。事实上，这个字符串有别于一个'普通的'空字符串，因为这个字符串有个数字索引值为 0。稍后有更详细描述。

如果一个 ENUM 被声明为 NULL，NULL 也是该列的一个合法值，并且该列的缺省值也将为 NULL 。如果一个 ENUM 被声明为 NOT NULL，该列的缺省值将是该列表所允许值的第一个成员。

每个枚举值均有一个索引值：

在列说明中列表值所允许的成员值被从 1 开始编号。

空字符串错误值的索引值为 0。这就意味着，你可以使用下面所示的 SELECT 语句找出被赋于无效 ENUM值的记录行。

SELECT \* FROM tbl\_name WHERE enum\_col=0;

NULL 值的索引值为 NULL。

、字段类原则

用好数值类型(用合适的字段类型节约空间);

字符转化为数字(能转化的最好转化,同样节约空间、提高查询性能);

避免使用NULL字段(NULL字段很难查询优化、NULL字段的索引需要额外空间、NULL字段的复合索引无效);

少用text类型(尽量使用varchar代替text字段);

3、索引类原则

合理使用索引(改善查询,减慢更新,索引一定不是越多越好);

字符字段必须建前缀索引;

不在索引做列运算;

innodb主键推荐使用自增列(主键建立聚簇索引,主键不应该被修改,字符串不应该做主键)(理解Innodb的索引保存结构就知道了);

不用外键(由程序保证约束);

4、sql类原则

sql语句尽可能简单(一条sql只能在一个cpu运算,大语句拆小语句,减少锁时间,一条大sql可以堵死整个库);

简单的事务;

避免使用trig/func(触发器、函数不用客户端程序取而代之);

不用select \*(消耗cpu,io,内存,带宽,这种程序不具有扩展性);

OR改写为IN(or的效率是n级别);

OR改写为UNION(mysql的索引合并很弱智);

select id from t where phone = ’159′ or name = ‘john’;

=>

select id from t where phone=’159′

union

select id from t where name=’jonh’

避免负向%;

慎用count(\*);

limit高效分页(limit越大，效率越低);

使用union all替代union(union有去重开销);

少用连接join;

使用group by;

请使用同类型比较;

打散批量更新;

核心原则

不在数据库做运算;

cpu计算务必移至业务层;

控制列数量(字段少而精,字段数建议在20以内);

平衡范式与冗余(效率优先；往往牺牲范式)

拒绝3B(拒绝大sql语句：big sql、拒绝大事物：big transaction、拒绝大批量：big batch);

5、性能分析工具

show profile;

mysqlsla;

mysqldumpslow;

explain;

show slow log;

show processlist;

MySQL

QPS 每秒查询量=查询数量/服务器启动时间

TPS 每秒传输事务处理的个数=（事务提交+事务回滚）/服务器启动时间

响应时间

压测工具

MySQLSlap，从5.6自带的压测工具

自动创建库 表 运行负载 测试完后自动删除

可自定义并发数，执行数，引擎等

MySQL架构（四层）

连接层

JDBC ODBC Native .NET

和数据库连接池

服务层

SQL Interface:DLL DML 视图（管理SQL语句）

转换器（）

缓存:SQL语句缓存（默认开启）和结果缓存（默认关闭）

可以在my.ini文件中关闭

sql解析顺序

优化器:优化掉 where 1=1，主键=null等

引擎层 级别表

show engines 查看数据库中的存储引擎

FRM存储表结构，所有引擎都有

MyISAM

不支持主外键

MYD数据文件

MYI索引文件

分开存放，非聚集索引

特征

并发性与表级别:表级别

支持全文检索（全文索引）

支持数据压缩 压缩索引文件 （myisampack -b -f 文件名），压缩后文件名够缀为OLD

该引擎使用场景 非事务应用 只读类应用 空间坐标（空间函数、坐标） 不适合高并发操作

Innodb

缓存索引和数据，对内存要求高

MySQL5.5之后默认应用该引擎

基本只有该引擎支持事务，默认行级锁，也支持表级锁

5.6之前系统表空间，之后独立表空间

修改set global innodb\_file\_par\_table=off

该为系统表空间

系统表空间无法简单的收缩文件大小，会会产生IO瓶颈

独立表空间可以通过optimize table收缩数据文件，可以同时向多个文件刷新数据

ibd文件存储存储数据+索引

聚集索引

支持高并发

CSV

数据以文本方式存储在文件

.CSV文件存储内容

.CSM文件存储表的元数据如表状态和数据量

所有列都不能为null

不支持索性（不适合大表，不适合在天线处理）

可以用记事本 Excle打开 编辑

用于财务类的

.

0

以zlib对表进行压缩，在arz为后缀的文件存储

只支持insert和select操作，只支持在自增id加索引

Memory

也称为HEAP存储引擎，数据保存在内存中，支持hash索引和BTree索引，不支持Blog和Text大字段

使用表级锁

最大大小由max\_heap\_table\_size参数决定

所有字段都是固定长度varchar（10）=char（10）

没数据存储文件

存储层

Ferderated

提供访问远程MySQL服务器上表的方法

本地不存储数据，数据全部放到远程服务器上

本地需要保存表结构和远程服务器的连接信息

使用场景 偶尔的统计分析及手工查询

锁

表级锁

开销小，加锁快，不会出现死锁，锁定粒度大，发生锁冲突概率最高，并发度最低

页面锁

会发生死锁，其他居中

行级锁

会发生死锁，反之

表级锁更适合查询，行级锁更适合事务

MyISAM表锁

只支持表锁

表共享读锁

表独占写锁，不能与写锁 读锁并行

InnoDb行锁

共享锁，又称读锁

排它锁，又称写锁

事务

一致性

持久性，并不是数据库角度完全能解决

隔离性，级别 未提交读…，默认可重复读

什么情况下，行锁会升级为表锁

隔离级别为可重复读，如果有索引的时候。以索引为条件更新数据，会存在间隙锁间、行锁、页锁的问题，从而锁住一些行，如果没有索引，更新数据时会锁住整张表

隔离级别为串行化是，读写数据都会锁住整张锁

逻辑设计

范式设计

所有字段都是单一属性

单一主键

单一依赖主键

反范式设计

物理设计

可读性原则

表意性原则

长名原则

数据类型的选择

优先数字类型，其次时间类型，最后字符类型

相同类型，越短越好

金额等精度要求高的使用 decimal

timestamp和时区有关，4字节，底层用数字类型

datetime与时区无关，5字节或8字节，5.6版本划分

慢查询

慢查询的定义

慢查询日志：查询慢的日志，指mysql记录所有执行超过long\_query\_time参数设定的时间阈值的sql语句的日志，默认关闭

分析工具

mysqldumpslow

pt\_query\_digest

索引

分类

普通索引 一个索引包含单个列，一个表可有多个单列索引

唯一索引 索引列的值必须唯一，但允许有空值

复合索引 即一个索引包含多个列

聚集索引（聚簇索引）文件的存储方式

执行计划

id相同，从上到下执行

id不同，从大到小执行

between

explain 执行计划，加在执行语句前

在utf-8编码下，一个字符占三个字节

null多占1节

可变长类型多占2字节

datetime在5.6之前占8字节，之后占5个字节

复合索引有最左前缀原则

索引长度为生效索引的和，若小于自定义索引长度，则说明有索引没生效

优化

尽量全值匹配

最佳左前缀原则 查询从索引的最左前列开始并且不跳过索印中的列（让索引不失效的策略）

不要在索引上做任何操作

范围条件放最后

覆盖索引尽量用

不等于要慎用

null not有影响

like要查询要当心

字符串要加引号

or改union