常用的SQL约束：

[NOT NULL约束](http://www.yiibai.com/sql/sql-not-null.html) - 确保列不能有NULL值

[默认值约束](http://www.yiibai.com/sql/sql-default.html) - 提供未指定时为列的值时的默认值

[唯一值约束](http://www.yiibai.com/sql/sql-unique.html) - 确保了在一列中的所有的值是唯一的

[主键](http://www.yiibai.com/sql/sql-primary-key.html) - 唯一标识数据库表中的每一行/记录

[外键](http://www.yiibai.com/sql/sql-foreign-key.html) - 唯一标识任何其他数据库表中的行/记录

[检查约束](http://www.yiibai.com/sql/sql-check.html) - CHECK约束可以确保列中的所有值满足一定的条件

[索引](http://www.yiibai.com/sql/sql-index.html) - 使用非常快速地创建和检索数据库中的数据。

**什么是存储过程？用什么来调用？**

答：存储过程是一个预编译的SQL语句，优点是允许模块化的设计，就是说只需创建一次，以后在该程序中就可以调用多次。如果某次操作需要执行多次SQL，使用存储过程比单纯SQL语句执行要快。可以用一个命令对象来调用存储过程。

**索引的作用？和它的优点缺点是什么？**

答：索引是一张特殊的表，这张表保存了主键与索引字段。索引可以大大提高对数据库的检索速度。

索引分为单列索引和组合索引。

索引应该用在SQL查询语句的条件中（WHERE的条件）

缺点：降低更新表的速度，如对表进行INSERT、UPDATE和DELETE（会同时更新索引）。同时也增加了数据库的大小。

**维护数据库的完整性和一致性**

尽可能使用约束，如check，主键，外键，非空字段等来约束，这样做效率最高，也最方便。

其次是使用触发器，这种方法可以保证，无论什么业务系统访问数据库都可以保证数据的完整新和一致性。最后考虑的是自写业务逻辑，但这样做麻烦，编程复杂，效率低下。

**什么是事务？什么是锁？**

事务就是一组连续的数据库操作的SQL语句。 事务处理可以用来维护数据库的完整性，保证成批的 SQL 语句要么全部执行，要么全部不执行。

要将一组SQL语句作为事务考虑，就需要通过ACID[**测试**](http://lib.csdn.net/base/softwaretest)，即原子性，一致性，隔离性和持久性。

命令行下默认事务是自动提交（COMMIT）的，可以使用BEGIN 或 START TRANSACTION开启一个事务。

 锁是实现事务的关键，锁可以保证事务的完整性和并发性。

## 事务特性：

事务具有以下四个标准属性，通常由首字母缩写ACID简称：

* **原子性** :  确保事务要么全部被执行，要么就全部不被执行。
* **一致性** :  确保数据库从一种正确状态转换成另一种正确状态。
* **隔离性** :  事务操作彼此独立和透明。在事务正确提交之前，不允许把该事务对数据的任何改变提供给任何其他事务
* **持久性** :  事务正确提交后，其结果将永久保存在数据库中，即使在事务提交后有了其他故障，事务的处理结果也会得到保存。

## 事务控制：

用来控制事务有如下命令：

* BEGIN:开始一个事务。
* COMMIT: 提交事务。
* ROLLBACK: 回滚更改。撤消尚未被保存到数据库事务的事务命令
* SET AUTOCOMMIT=0 :禁止自动提交; SET AUTOCOMMIT=1:开启自动提交。

**三范式**

第一范式：强调的是列的原子性，即列不能够再分成其他几列。

考虑这样一个表：学生信息表（姓名，性别，课程，成绩）   
如果在实际场景中，一个学生有多个课程，那么这种表结构设计就没有达到 1NF。对于每个课程都要插入一个记录。要符合 1NF 我们需要把列（课程，成绩）拆分，即：（姓名，性别，课程1，成绩1，课程2，成绩2）。

第二范式：首先满足INF，同时还要求数据库表中的每行必须可以被唯一地区分。

要满足两个条件：一是表必须有一个主键；二是非主键的列必须完全依赖于主键。

学生成绩表（学号，姓名，课程号，课程名，成绩），如果以学号和课程号作为主键。姓名只需要学号就能唯一确定，课程名只需要课程号唯一确定。这样设计就不符合2NF。可以进行拆分：学生信息表（学号，姓名）和课程信息表（课程号，课程名），然后再新建一个学生成绩表（学号，课程号，成绩）。

第三范式：首先是 2NF，另外非主键列必须直接依赖于主键，不能存在传递依赖。

学生信息表（学号，姓名，学院编号，学院名）,如果只以学号作为主键，每个非主键属性都完全由主键确定。符合2NF。但是学院名可以由学院编号确定，学院编号又由学号确定，存在依赖传递。

可以这样拆分：删除学院名属性，同时新建一个学院信息表（学院编号，学院名）。

**数据库操作**

CREATE DATABASE database\_name;

DROP DATABASE database\_name;

USE database\_name;

**创建表：**

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL UNIQUE,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2) DEFAULT 5000.00,

PRIMARY KEY (ID) --主键必须是 NOT NULL

--PRIMARY KEY (ID,NAME) --多个主键（复合键）

);

CHAR 和 VARCHAR 的区别：CAHR 长度是固定的，而VARCHAR 的长度可以变化。因为CHAR长度固定，所有存储和查找效率高，同时需要更多的空间。

**显示创建表的语句**

SHOW CREATE TABLE

**复制表的数据**

INSERT INTO clone\_TABLE (col1,clo2,...,cloN)

SELECT col1,clo2,...,cloN FROM TABLE;

**导出数据**

SELECT \* FROM table\_name INTO OUTFILE '/tmp/dump.txt';

**导入数据**

LOAD DATA INFILE 'dump.txt' INTO TABLE table\_name;

**复制或备份表和数据**

备份数据表

mysqldump -u username -p dbname table1 table2 > BackupName.sql

备份多个数据库

mysqldump -u username -p --databases dbname1 dbname2 > Backup.sql

mysqldump -u username -p -all-databases > BackupName.sql

**还原数据库**

mysql -u username -p [dbname] < backup.sql

**修改表的约束：**

--增加NOT NULL

ALTER TABLE CUSTOMERS

MODIFY SALARY DECIMAL (18,2) NOT NULL;

--增加DEFAULT

ALTER TABLE CUSTOMERS

MODIFY SALARY DECIMAL (18,2) DEFAULT 10000.00;

--删除列的默认值约束

ALTER TABLE CUSTOMERS

ALTER COLUMN SALARY DROP DEFAULT;

--增加主键约束

ALTER TABLE CUSTOMER

ADD PRIMARY KEY (ID);

--删除主键约束

ALTER TABLE CUSTOMER

DROP PRIMARY KEY (ID);

**支持多列的修改操作：**

--设置约束myConstraint

ALTER TABLE CUSTOMERS

ADD CONSTRAINT myConstraint UNIQUE(ADDRESS,SALARY);

--删除多列的约束

ALTER TABLE CUSTOMERS

DROP CONSTRAINT myConstraint;

--表重命名

ALTER TABLE table\_name RENAME TO new\_table\_name;

--删除表

DROP TABLE CUSTOMERS;

CREATE TABLE ORDERS (

ID INT NOT NULL,

DATE DATETIME,

CUSTOMER\_ID INT references CUSTOMERS(ID),

--设置外键，连接CUSTOMERS表的ID列

AMOUNT double,

AGE INT NOT NULL CHECK (AGE >= 18), --CHECK约束

PRIMARY KEY (ID)

);

--增加外键

ALTER TABLE ORDERS

ADD FOREIGN KEY (CUSTOMER\_ID) REFERENCES CUSTOMERS (ID);

--删除外键

ALTER TABLE ORDERS

DROP FOREIGN KEY;

--在CUSTOMERS的AGE列上创建索引

CREATE INDEX index\_name

ON CUSTOMERS ( AGE );

--删除索引

ALTER TABLE CUSTOMERS

DROP INDEX index\_name;

## 什么时候避免使用索引？

1. 索引不应该用在小型表上。
2. 有频繁的，大批量更新或插入操作的表。
3. 索引不应该用于对包含大量NULL值的列。
4. 列经常操纵不应该被索引。

|  |  |
| --- | --- |
| **操作符** | **描述** |
| ALL | ALL运算符是用来在另一个值设定比较值的所有值 |
| AND | AND运算允许多个条件在SQL语句中，存在WHERE子句 |
| ANY | ANY运算符用于根据条件在列表中的值进行比较的任何应用值 |
| BETWEEN | BETWEEN运算符用于搜索是在一组值的那个值，给定的最小值和最大值 |
| EXISTS | EXISTS运算符用于搜索行中指定的表，以满足某些标准的存在 |
| IN | IN运算符用于一个值进行比较，以已被指定的文字值的列表 |
| LIKE | LIKE运算符用来比较使用通配符运算符相似的值。%:表示0、1或多个字符。\_ :表示单个字符。可以用AND或OR来组合。 |
| NOT | NOT运算符反转与它被使用的逻辑运算器的含义。例如：NOT EXISTS，NOT BETWEEN，NOT IN等，这是一个否定运算符 |
| OR | OR运算符是用来多个条件WHERE子句结合起来的SQL语句 |
| IS NULL | NULL运算符用来比较一个NULL值 |
| UNIQUE | UNIQUE操作搜索指定表的每一行的唯一性（不重复） |

**查询语句**

**返回唯一值**

SELECT DISTINCT column1, column2 FROM table\_name;

**AND|OR**

SELECT column1, column2 FROM table\_name

WHERE CONDITION1 AND CONDITION2;

属性值在IN指定值中

SELECT column1, column2 FROM table\_name

WHERE column\_name IN (val1, val2,...valN);

属性值在BETWEEN和AND之间

SELECT column1, column2 FROM table\_name

WHERE column\_name BETWEEN val1 AND val2;

通配符LIKE

SELECT column1, column2 FROM table\_name

WHERE column\_name LIKE ‘100%’; --’%100%’、’\_100%’

排序

SELECT column1, column2 FROM table\_name

WHERE CONDITION ORDER BY column\_name DESC; --ASC

合计函数分组

SELECT SUM(column\_name) FROM table\_name WHERE CONDITION

GROUP BY column\_name;

SELECT COUNT(column\_name) FROM table\_name WHERE CONDITION;

WHERE 关键字不能用在分组结果的条件

SELECT SUM(column\_name) FROM table\_name WHERE CONDITION

GROUP BY column\_name HAVING condition;

**数据操作**

INSERT INTO table\_name( column1, column2....columnN)

VALUES ( value1,value2....valueN);

UPDATE table\_name

SET column1 = value1, column2 = value2....columnN=valueN[ WHERE CONDITION ];

DELETE FROM table\_name WHERE {CONDITION};

**UNION**

SELECT column1 [, column2 ] FROM table1 [, table2 ][WHERE condition]

UNION

SELECT column1 [, column2 ] FROM table1 [, table2 ][WHERE condition]

**多表查询**

**引用两个表**

SELECT table1.column1, table2.column2 FROM table1, table2

WHERE table1.common\_field = table2.common\_field;

**INNER JOIN（内连接）**

**至少有一个匹配时，返回行。**

SELECT table1.column1, table2.column2 FROM table1

INNER JOIN table2 ON table1.common\_field = table2.common\_field;

**LEFT JOIN（左连接）：**从左边的表返回所有行，即使在右表中没有匹配的行。

**RIGHT JOIN（右连接）：**从右边的表返回所有行，即使在左表中没有匹配的行。

**FULL JOIN**：只要其中某个表存在匹配，就会返回行。

**大数据的SQL查询优化**

1. 对查询进行优化，应尽量避免全表扫描。
2. 首先应考虑在 where 及 order by 涉及的列上建立索引。
3. 将导致引擎放弃使用索引而进行全表扫描的情况：
4. 在 where 子句中对字段进行 is null 值判断。
5. 在 where 子句中使用!=或<>操作符。
6. 在 where 子句中使用 or 来连接条件。可以这样查询：select id from t where num=10 union all select id from t where num=20
7. 在 where 子句中使用变量。因为SQL只有在运行时才会解析局部变量，可以改为强制查询使用索引：select id from t with(index(索引名)) where num=@num
8. 在 where 子句中对字段进行表达式操作。应改为:select id from t where num=100\*2
9. 在where子句中对字段进行函数操作。如：select id from t where substring(name,1,3)='abc' ，name以abc开头的id应改为:select id from t where name like 'abc%'
10. 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。
11. 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。

redis 操作

键命令

|  |  |
| --- | --- |
| 基本键命令 | 描述 |
| **DEL** key | key 存在时删除 |
| **EXISTS** key | 检查 key 是否存在 |
| **EXPIRE** key seconds | 设置 key 的过期时间（秒） |
| **PEXPRIRE** key ms | 设置 key 的过期时间（毫秒） |
| **PERSIST** key | 移除 key 的过期时间 |
| **PTTL** key | 以毫秒返回过期时间 |
| **TTL** key | 以秒返回过期时间 |
| **RENAME** key newkey | 对 key 改名 |
| **TYPE** key | 返回 key 对应值的类型 |

string 字符串

|  |  |
| --- | --- |
| 基本字符命令 | 描述 |
| **SET** key value | 设置K-V |
| **GET** key | 获取键值 |
| **MSET**  key value [key value..] | 设置多个K-V对 |
| **MGET** key1 [key2..] | 获取多个键值 |
| **STRLEN** key | 字符串值长度 |
| **GETRANGE** key start end | 子串：从start到end |
| **SETRANGE** key offset value | 覆盖：用value覆盖从offset开始的字符串值 |
| **GETSET** key value | 设置返回：key的值设为value，返回原来的值 |
| **SETEX** key seconds value | 设置K-V和过期时间 |
| **INCR** key | 加1：key的数字值加1 |
| **INCRBY** key increment | 加值：key的数字值加给定值 |
| **DECR** key | 减1：key的数字值减1 |
| **DECRBY** key decrement | 减值：key的数字值减给定值 |
| **APPEND** key value | 追加：如果key存在且值是字符串，追加到末尾 |
| **GETBIT** key offset | 获取位：offset上的值 |
| **SETBIT** key offset value | 设置位：offset处设为value |

hash 哈希表

|  |  |
| --- | --- |
| 基本哈希命令 | 描述 |
| **HSET** key field value | 设置表中field-value |
| **HGET** key field | 获取字段field的值 |
| **HMSET**  key field1 value1 [field2 value2..] | 设置多个field-value对 |
| **HMGET** key field1 [field2..] | 获取多个field的值 |
| **HKEYS** key | 获取所有字段 |
| **HVALS** key | 获取所有值 |
| **HGETALL** key | 获取所有field-value |
| **HLEN** key | 字段数 |
| **HEXISTS** key field | 字符是否存在 |
| **HDEL** key field1 [field2..] | 删除多个字段 |
| **HINCRBY** key field increment | 加上增量：field字段的数值加increment |

list 列表

|  |  |
| --- | --- |
| 基本列表命令 | 描述 |
| **LPUSH** key value | 插入：一个或多个元素到头部 |
| **RPUSH**  key value [key value..] | 插入：一个或多个元素到尾部 |
| **LPOP** key | 移出并获取：第一个元素 |
| **RPOP** key1 | 移出并获取：最后一个元素 |
| **LLEN** key | 列表长度 |
| **LRANGE** key start end | 范围内元素：从start到end |
| **LTRIM** key start end | 修剪：只保留 [ start,end）内的元素 |
| **LREM** key count value | 移除：count个value，不足不操作 |
| **RPOPLPUSH** key src des | 移出并插入：src最后一个元素，添加到des |
| **BRPOPLPUSH** key src des timeout | 从src弹出一个元素，插入到des中并返回，如果没有阻塞到超时 |

set 集合

|  |  |
| --- | --- |
| 集合基本命令 | 描述 |
| **SADD** key mem1 [mem2..] | 添加：一个或多个成员到集合 |
| **SREM** key member1 [member2] | 移出：一个或多个成员 |
| **SISMEMBER** key merber | 是否为成员：判断member是否在集合中 |
| **SMEMBERS** key | 返回所有成员 |
| **SCARD**  key | 所有成员个数 |
| **SMOVE** src des member | 移动：member从src 到des |
| **SDIFF** key1 [key2] | 差集：给定所有集合的 |
| **SDIFFSTORE des** key1 [key2] | 存储差集：给定集合的差集存储到des |
| **SINTER** key1 [key2] | 交集：给定集合的 |
| **SINTERSTORE** des key1 [key2] | 存储交集：到des |
| **SUNION** key1 [key2] | 并集：给定集合的 |
| **SUNIONSTORE** des key1 [key2] | 存储并集：给定集合并集存储到des |

zset 有序集合

|  |  |
| --- | --- |
| 有序集合基本命令 | 描述 |
| **ZADD** key score1 member1  [score2 member2] | 添加：一个或多个成员到集合 |
| **SREM** key member1 [member2] | 移出：一个或多个成员 |
| **ZCARD**  key | 所有成员个数 |
| **ZRANGE** key start end | 索引区间成员 |
| **ZREVRANGE** key start end | 逆序索引区间成员 |
| **ZRANGEBYSCORE** key min max | 分数区间成员 |
| **ZREVRANGEBYSCORE** key max min | 逆序分数区间成员：分数从高到低 |
| **ZSCORE** key member | 分数值：指定成员 |
| **ZRANK** key member | 索引值：按分数递增排序 |
| **ZREVRANK**  key member | 逆序索引值，按分数递减排序 |
| **ZCOUNT** key min max | 分数区间成员数 |
| **ZREMRANGEBYRANK** key min max | 移出：索引区间成员 |
| **ZREMRANGEBYSCORE** key min max | 移出：分数区间成员 |
| **ZINCRBY** key increment member | 加上增量：指定成员的分数 |
| **ZINTERSTORE** des numkeys key [key] | 存储交集：到新的有序集合 |