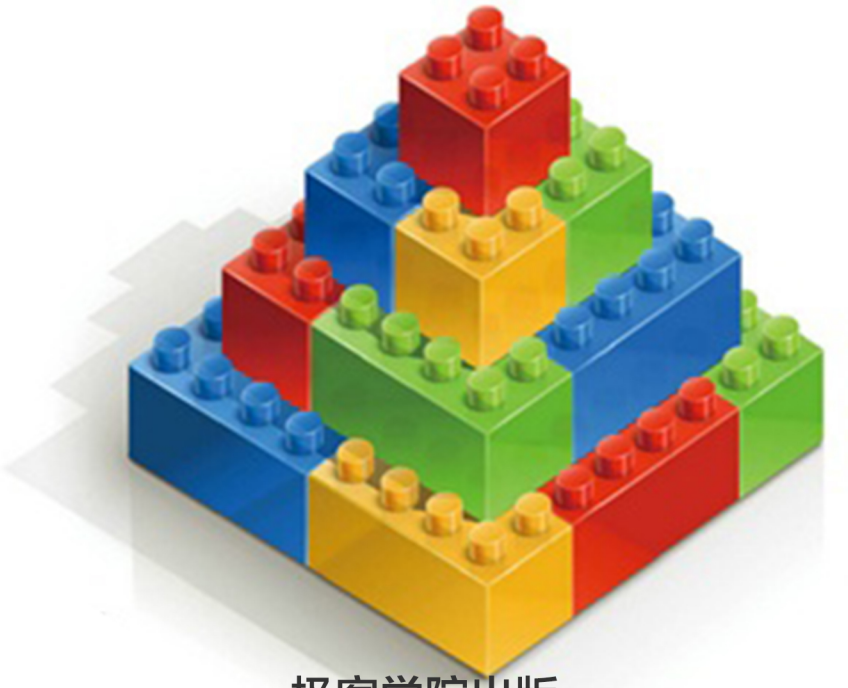
一门 ANSI 的标准计算机语言，用来访问和操作数据库系统

SQL

语法入门

The Language of SQL



极客学院出版

前言

SQL 是一种数据库语言，被设计用来检索及管理关系数据库中的数据。SQL 是 Structured Query Language（结构化查询语言）的缩写。

本指南是一本非常详细的基础教程，涉及常用 SQL 语言的所有知识点，能够让你对 SQL 的语法和语义有个清晰的认识。

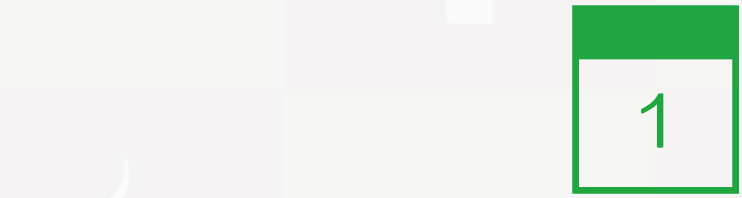
适用人群

本参考的目的在于帮助初学者深入浅出地学习 SQL 语言。

学习前提

本参考准备了各种各样的示例，在正式开始练习之前，我假定你对什么是数据库——尤其是关系型数据库管理系统（RDBMS）——已经有所了解，同时也知道什么是计算机编程语言。

目录



SQL 教程

概览

SQL 指南给你学习结构化查询语言的历程带来独特的体验，它能够帮助你交互地学习 SQL 命令。SQL 是一种数据库语言，能够完成数据库的创建、删除，取回或者修改其中的数据等工作。

SQL 是一个 ANSI（American National Standard Institute，美国国家标准协会）标准。不过，SQL 语言有很多不同的版本存在。

什么是 SQL?

SQL 是结构化查询语言（Structured Query Language），一种用于存储、操作或者检索存储在关系型数据库中数据的计算机语言。

SQL 是关系型数据库系统（Relation Database System）的标准语言。所有的关系型数据库管理系统，例如 MySQL、MS Access、Oracle、Sybase、Informix、Postgres SQL 和 SQL Server，都使用 SQL 作为其标准数据库语言。

当然，它们用的都是不同的 SQL 方言。例如：

- 微软的 SQL Server 使用的是 T-SQL
- Oracle 使用的是 PL/SQL
- 微软的 Access 中的 SQL 叫做 JET SQL（本地格式）等等

为什么要用 SQL?

- 允许用户访问关系型数据库中的数据
- 允许用户对数据做出描述
- 允许用于定义数据库中的数据，并对其进行操作
- 允许通过 SQL 模块、库或者预编译器的等方式，嵌入到其他语言中
- 允许用户创建或删除数据库和表
- 允许用户在数据库中创建视图、存储过程和函数
- 允许用户对表、过程和视图设进行权限设置

史海回眸

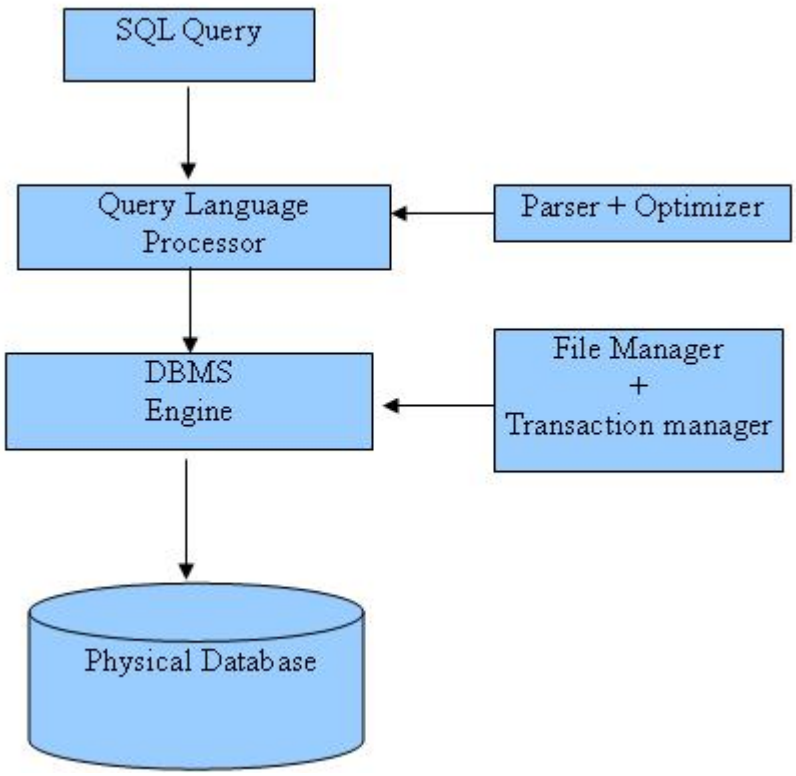
- 1970 年，IBM 的埃德加 · 科德博士提出了一种数据库关系模型，他因此被称作关系型数据库之父。
- 1974 年，结构化查询语言面世。
- 1978 年，IBM 对 Codd 提出的概念进行了深入研究，并发布了一款名为 System/R 的产品。
- 1986 年，IBM 开发出了第一个关系型数据库原型，该原型随后被 ANSI 接纳并进行了标准化工作。Relational Software——Oracle 的前身——发布了第一款商业关系型数据库产品。

SQL 流程

在任何一种 RDBMS 上执行 SQL 命令，数据库管理系统都会判断出执行请求的最佳方式，并由 SQL 引擎推算出具体如何完成任务。

这一流程涉及到了各种各样的组件，包括查询调度器（Query Dispatcher）、优化引擎（Optimization Engines）、经典查询引擎（Classic Query Engine）和 SQL 查询引擎（SQL Query Engine）等等。经典查询引擎用于处理所有的非 SQL 查询，而 SQL 查询引擎则不处理逻辑文件。

下面这张图片简要说明了 SQL 的架构：



图片 1.1 SQL Architecture

SQL 命令

用于与关系型数据库交互的标准 SQL 命令有 CREATE、SELECT、INSERT、UPDATE、DELETE 和 DROP，这些命令按用途分成如下几组：

数据定义语言

命令	描述
CREATE	创建新的表、视图或者其他数据库中的对象
ALTER	修改现存数据库对象，比如一张表
DROP	删除表、视图或者数据库中的其他对象

数据操纵语言

命令	描述
SELECT	从一张或者多张表中检索特定的数据
INSERT	创建一条新记录
UPDATE	修改记录
DELETE	删除记录

数据控制语言

命令	描述
GRANT	赋予用户特权
REVOKE	收回赋予用户的特权

关系型数据库管理系统

什么是关系型数据库管理系统?

RDBMS 是关系型数据库管理系统（Relational DataBase Management System)的缩写，它是 SQL 以及所有现代数据库系统，例如 MS SQL Server、IBM DB2、Oracle、MySQL 和 MS Access等的基础。

关系型数据库管理系统（RDBMS）是一种基于 E.F. 科德提出的关系模型的数据库管理系统。

什么是表?

RDBMS 中的数据存储在被称作**表**的数据库对象中。表是相互关联的数据记录的集合，由一系列的行和列组成。

谨记，表是关系型数据库中最常见也是最简单的数据存储形式。下面是一个客户信息表的例子:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

什么是字段?

每张表都能够划分成更小的实体——**字段**。例如，上面的客户信息表中有 ID、NAME、AGE、ADDRESS 和 SALARY 五个字段。

一个字段限定了数据表中的列，被用来维护表中所有记录的特定信息。

什么是记录或者数据行?

记录或者说**数据行**是存在于数据表中的独立条目。例如，上面的客户信息表中有 7 条记录。下面是客户信息表中的一条记录：

```
+----+----------+-----+-----------+----------+
| 1 | Ramesh  | 32 | Ahmedabad |  2000.00 |
+----+----------+-----+-----------+----------+
```

记录就是表中水平排列的数据构成的实体。

什么是列?

列是表中竖直排列的实体，它包含了表中与某一特定字段相关的所有信息。

例如，上面的客户信息表中有字段为 ADDRESS 的列，存储了客户的地址，其内容如下所示：

```
+-----------+
| ADDRESS   |
+-----------+
| Ahmedabad |
| Delhi     |
| Kota      |
| Mumbai    |
| Bhopal    |
| MP        |
| Indore    |
+----+------+
```

什么是 NULL 值?

NULL 值是表中以空白形式出现的值，表示该记录在此字段处没有设值。

一定要明白 NULL 值同 0 值或者包含空格的字段是不同的。值为 NULL 的字段是在记录创建的时候就被留空的字段。

SQL 约束

约束是表中的数据列必须遵守的规则，用于限制表中数据的类型。约束保证了数据库中数据的精确性和可靠性。

约束可以限制列或者表。列级的约束只限制单一的列，而表级的约束作用于整个表。

以下是 SQL 中常见的约束：

- NOT NULL 约束：保证列中数据不能有 NULL 值
- DEFAULT 约束：提供该列数据未指定时所采用的默认值
- UNIQUE 约束：保证列中的所有数据各不相同
- 主键：唯一标识数据表中的行/记录
- 外键：唯一标识其他表中的一条行/记录
- CHECK 约束：此约束保证列中的所有值满足某一条件
- 索引：用于在数据库中快速创建或检索数据

NOT NULL 约束

默认情况下，数据表中的字段接受 NULL 值。如果你不想让某个字段接受 NULL 值，那么请为该字段定义此约束，以指明该字段不接受 NULL 值。

NULL 并不是指**没有**数据，而是指该字段数据**未知**。

示例：

例如，下述 SQL 语句创建了一个新的数据表 CUSTOMERS，并添加了五个字段，其中三个字段——ID、NAME 和 AGE——被指定为 NOT NULL：

```
CREATE TABLE CUSTOMERS(
    ID   INT              NOT NULL,
    NAME VARCHAR (20)     NOT NULL,
    AGE  INT              NOT NULL,
    ADDRESS  CHAR (25) ,
    SALARY   DECIMAL (18, 2),
    PRIMARY KEY (ID)
);
```

对于 Oracle 和 MySQL 来说，如果 CUSTOMERS 表已经存在，此时再要给 SALARY 字段添加 NOT NULL 约束的话，SQL 语句应当如下：

```
ALTER TABLE CUSTOMERS
  MODIFY SALARY  DECIMAL (18, 2) NOT NULL;
```

DEFAULT 约束

DEFAULT 约束在 INSERT INTO 语句没有提供的情况下，为指定字段设置默认值。

示例:

例如，下述 SQL 语句创建了一个名为 CUSTOMERS 的新表，并添加了五个字段。这里，SALARY 字段的默认值为 5000。因此，如果 INSERT INTO 没有为该字段提供值的话，该字段就为默认值 5000。

```
CREATE TABLE CUSTOMERS(
    ID   INT            NOT NULL,
    NAME VARCHAR (20)     NOT NULL,
    AGE  INT            NOT NULL,
    ADDRESS  CHAR (25) ,
    SALARY   DECIMAL (18, 2) DEFAULT 5000.00,
    PRIMARY KEY (ID)
);
```

如果 CUSTOMERS 表已经存在，此时再要给 SALARY 字段添加 DEFAULT 约束的话，你需要类似下面的语句:

```
ALTER TABLE CUSTOMERS
  MODIFY SALARY  DECIMAL (18, 2) DEFAULT 5000.00;
```

删除 DEFAULT 约束:

要删除 DEFAULT 约束的话，请使用下面的 SQL 语句:

```
ALTER TABLE CUSTOMERS
  ALTER COLUMN SALARY DROP DEFAULT;
```

UNIQUE 约束

UNIQUE 约束使得某一字段对任意两条记录来说都不能相同。例如，在 CUSTOMERS 表中，你或许想让任何人的年龄（age）都不相同。

示例:

例如，下述 SQL 语句创建了一个名为 CUSTOMERS 的新表，并添加了五个字段，其中 AGE 字段被设为 UNIQUE，于是任意两条记录的 AGE 都不同:

```
CREATE TABLE CUSTOMERS(
    ID   INT              NOT NULL,
    NAME VARCHAR (20)     NOT NULL,
    AGE  INT              NOT NULL UNIQUE,
    ADDRESS  CHAR (25) ,
    SALARY   DECIMAL (18, 2),
    PRIMARY KEY (ID)
);
```

如果 CUSTOMERS 表已经存在，再要为 AGE 字段添加 UNIQUE 约束的话，你需要像下面这样写 SQL 语句：

```
ALTER TABLE CUSTOMERS
  MODIFY AGE INT NOT NULL UNIQUE;
```

还可以使用如下所示的语法，该语法还支持对作用于多个字段的约束进行命名：

```
ALTER TABLE CUSTOMERS
  ADD CONSTRAINT myUniqueConstraint UNIQUE(AGE, SALARY);
```

删除 UNIQUE 约束

要删除 UNIQUE 约束的话，请使用如下 SQL 语句：

```
ALTER TABLE CUSTOMERS
  DROP CONSTRAINT myUniqueConstraint;
```

如果你在使用 MySQL，那么下面的语法也是可行的：

```
ALTER TABLE CUSTOMERS
  DROP INDEX myUniqueConstraint;
```

主键

主键是数据表中唯一确定一条记录的字段。主键必须包含唯一值，并且不能为 NULL。

每张数据表只能有一个主键，不过一个主键可以包含一个或者多个字段。如果主键由多个字段组合而成，这些字段就被称作**组合键**。

如果一个字段被定义为了某表的主键，则任意两条记录在该字段处不能相同。

注意：在创建数据表的时候，需要用到这些概念。

创建主键：

如下是将 ID 定义为 CUSTOMERS 表主键的语法：

```
CREATE TABLE CUSTOMERS(
    ID   INT              NOT NULL,
    NAME VARCHAR (20)     NOT NULL,
    AGE  INT              NOT NULL,
    ADDRESS  CHAR (25) ,
    SALARY   DECIMAL (18, 2),
    PRIMARY KEY (ID)
);
```

如果 CUSTOMERS 表已经存在了，再要将 ID 定义为主键的话，请使用下面的语句：

```
ALTER TABLE CUSTOMER ADD PRIMARY KEY (ID);
```

注意：如果你要使用 ALTER TABLE 语句来添加主键，那么主键所在的列必须已经被声明为 NOT NULL 了。

要用多个字段来定义主键的话，请使用如下 SQL 语法：

```
CREATE TABLE CUSTOMERS(
    ID   INT              NOT NULL,
    NAME VARCHAR (20)     NOT NULL,
    AGE  INT              NOT NULL,
    ADDRESS  CHAR (25) ,
    SALARY   DECIMAL (18, 2),
    PRIMARY KEY (ID, NAME)
);
```

如果 CUSTOMERS 表已经存在，此时再要将 ID 和 NAMES 字段定义为主键的话，请使用如下 SQL 语法：

```
ALTER TABLE CUSTOMERS
  ADD CONSTRAINT PK_CUSTID PRIMARY KEY (ID, NAME);
```

删除主键

你可以将主键约束从数据表中删除，语法如下：

```
ALTER TABLE CUSTOMERS DROP PRIMARY KEY ;
```

外键

外键用于将两个数据表连接在一起，有时候也被称作“参照键”。

外键为单一字段或者多个字段的组合，并与另外一个数据表的主键相匹配。

两个表之间的关系是：一个表的主键与另一个表的外键相匹配。

示例：

考虑如下两个表的结构：

CUSTOMERS 表：

```
CREATE TABLE CUSTOMERS(
    ID   INT              NOT NULL,
    NAME VARCHAR (20)     NOT NULL,
    AGE  INT              NOT NULL,
    ADDRESS  CHAR (25) ,
    SALARY   DECIMAL (18, 2),
    PRIMARY KEY (ID)
);
```

ORDERS 表：

```
CREATE TABLE ORDERS (
    ID         INT       NOT NULL,
    DATE       DATETIME,
    CUSTOMER_ID INT references CUSTOMERS(ID),
    AMOUNT     double,
    PRIMARY KEY (ID)
);
```

如果 ORDERS 表已经存在，并且没有设置外键，那么可以使用下面的语法来修改数据表以指定外键。

```
ALTER TABLE ORDERS
  ADD FOREIGN KEY (Customer_ID) REFERENCES CUSTOMERS (ID);
```

删除外键约束：

要删除外键约束的话，语法如下所示：

```
ALTER TABLE ORDERS
  DROP FOREIGN KEY;
```

CHECK 约束

CHECK 约束使用某一条件来对记录中的值进行检查。如果条件最终为假（false），即约束条件不能得到满足，则该记录不能写入数据表中。

示例：

例如，下述 SQL 语句创建了一个名为 CUSTOMERS 的新表，并为其添加了五个字段。在此，我们为 AGE 字段设置了 CHECK 约束，以拒绝任何年龄低于 18 的顾客：

```
CREATE TABLE CUSTOMERS(
    ID   INT              NOT NULL,
    NAME VARCHAR (20)     NOT NULL,
```

```
    AGE  INT            NOT NULL CHECK (AGE >= 18),
    ADDRESS  CHAR (25) ,
    SALARY   DECIMAL (18, 2),
    PRIMARY KEY (ID)
);
```

如果 CUSTOMERS 表已经存在，再要为 AGE 字段设置 CHECK 约束的话，就需要像下面这样写 SQL 语句：

```
ALTER TABLE CUSTOMERS
  MODIFY AGE INT NOT NULL CHECK (AGE >= 18 );
```

或者也可以使用下面的语法，该语法还支持对作用于多个字段的约束命名：

```
ALTER TABLE CUSTOMERS
  ADD CONSTRAINT myCheckConstraint CHECK(AGE >= 18);
```

删除 CHECK 约束：

要删除 CHECK 约束的话，请使用下面的 SQL 语句，不过该语句在 MySQL 中不起作用：

```
ALTER TABLE CUSTOMERS
  DROP CONSTRAINT myCheckConstraint;
```

索引

索引用于在数据库中快速地创建和检索数据。索引可以由表中的一个或者多个字段创建。创建索引时，每一行都会获得一个 ROWID（在数据进行排序之前）。

合理运用索引可以提高大型数据库的性能。但是，创建索引之前还是要三思而后行。为哪些字段创建索引，则取决于 SQL 查询最常用到到哪些字段。

示例：

例如，下面的 SQL 语句创建了一个名为 CUSTOMERS 的新表，并为其添加了五个字段：

```
CREATE TABLE CUSTOMERS(
    ID   INT            NOT NULL,
    NAME VARCHAR (20)     NOT NULL,
    AGE  INT            NOT NULL,
    ADDRESS  CHAR (25) ,
    SALARY   DECIMAL (18, 2),
    PRIMARY KEY (ID)
);
```

现在，你就可以使用下面的语法来为一个或者多个字段创建索引了：

```
CREATE INDEX index_name
  ON table_name ( column1, column2.....);
```

例如，可以在 AGE 字段上创建索引，以优化对特定年龄的顾客的查询，其语法如下所示：

```
CREATE INDEX idx_age
  ON CUSTOMERS ( AGE );
```

删除索引约束：

要删除索引约束的话，可以使用下面的 SQL 语句：

```
ALTER TABLE CUSTOMERS
  DROP INDEX idx_age;
```

数据完整性

下面几类数据完整性存在于各个 RDBMS 中：

- **实体完整性**：表中没有重复的行
- **域完整性**：通过限制数据类型、格式或者范围来保证给定列的数据有效性
- **参照完整性**：不能删除被其他记录引用的行
- **用户定义完整性**：施加某些不属于上述三种完整性的业务规则

数据库规范化

数据库规范化指的是对数据库中的数据进行有效组织的过程。对数据库进行规范化主要有两个目的：

- 消除冗余数据，例如相同数据出现在不同的表中。
- 保证数据依赖性合理。

这两个目标都值得我们努力，因为它们可以减少数据的空间占用，并确保了数据的逻辑完备。规范化包含一系列的指导方针，以帮助你创建出优良的数据库结构。

规范化指导方针分为几种范式（form），你可以把范式想做是数据库的格式或者其结构的布局方式。使用范式的目标是对数据库结构进行整理，从而使其遵循第一范式，接着是第二范式，最终遵循第三范式。

要不要更进一步到达第四范式、第五范式甚至更高的范式取决于你。一般来说，第三范式足矣。

第一范式（1NF）

第一范式设定了对数据库进行组织的最基本的规范：

- 定义需要的数据项，因为这些项将会成为数据表中的字段。将相关的数据项放在一个表中。
- 保证不存在重复的数据。
- 保证有一个主键。

1NF 的第一规则：

你必须定义所需的数据项。这意味着查看要存储的数据，按照字段对其进行组织，定义各个字段的数据类型，最终将相关的字段放在同一个表中。

例如，将所有与会议地点相关的字段放在 Location 表中，将所有同与会成员相关的字段放在 MemberDetails 表中等等。

1NF 的第二规则：

下一步是保证不存在重复的数据集合。考虑如下的数据表：

```
CREATE TABLE CUSTOMERS(
    ID   INT            NOT NULL,
    NAME VARCHAR (20)    NOT NULL,
    AGE  INT            NOT NULL,
    ADDRESS  CHAR (25),
    ORDERS   VARCHAR(155)
);
```

如果我们用同一个顾客的多笔订单来填充该表，将会得到类似下面的数据表：

ID	NAME	AGE	ADDRESS	ORDERS
100	Sachin	36	Lower West Side	Cannon XL-200
100	Sachin	36	Lower West Side	Battery XL-200
100	Sachin	36	Lower West Side	Tripod Large

但是，按照 1NF 我们必须保证没有重复的数据集合。所以，可以将上表分成两部分，然后使用一个键将两个表连接起来。

CUSTOMERS 表：

```
CREATE TABLE CUSTOMERS(
    ID   INT            NOT NULL,
    NAME VARCHAR (20)    NOT NULL,
    AGE  INT            NOT NULL,
```

```
    ADDRESS  CHAR (25),
    PRIMARY KEY (ID)
);
```

表中记录如下：

ID	NAME	AGE	ADDRESS
100	Sachin	36	Lower West Side

ORDERS 表：

```
CREATE TABLE ORDERS(
    ID   INT            NOT NULL,
    CUSTOMER_ID INT      NOT NULL,
    ORDERS   VARCHAR(155),
    PRIMARY KEY (ID)
);
```

表中记录如下：

ID	CUSTOMER_ID	ORDERS
10	100	Cannon XL-200
11	100	Battery XL-200
12	100	Tripod Large

1NF 的第三规则：

第一范式的最后一条规则是，为每一个数据表创建一个主键。

第二范式（2NF）

第二范式规定，数据表必须符合第一范式，并且所有字段与主键之间不存在部分依赖关系。

考虑顾客与订单之间的关系，你可能会想要存储顾客 ID、顾客姓名、订单 ID、订单明细以及购买日期：

```
CREATE TABLE CUSTOMERS(
    CUST_ID    INT            NOT NULL,
    CUST_NAME VARCHAR (20)     NOT NULL,
    ORDER_ID   INT            NOT NULL,
    ORDER_DETAIL VARCHAR (20)  NOT NULL,
    SALE_DATE  DATETIME,
    PRIMARY KEY (CUST_ID, ORDER_ID)
);
```

该表符合第一范式，因为它满足第一范式的所有规则。表中的主键有 CUST_ID 和 ORDER_ID。二者一起作为主键，我们假定同一个顾客不会购买相同的东西。

然而，该表不符合第二范式，因为表中的字段和主键之间存在部分依赖关系。CUST_NAME 依赖于 CUST_ID，而 CUST_NAME 和所购物品之间没有直接的联系。订单明细和购买日期依赖于 ORDER_ID，但是他们并不依赖于 CUST_ID，因为 CUST_ID 和 ORDER_DETAIL 以及 SALE_DATE 之间并不存在联系。

要使该表遵守第二范式，你需要将其分为三个数据表。

首先，创建如下的数据表来保存客户详情：

```
CREATE TABLE CUSTOMERS(
    CUST_ID    INT           NOT NULL,
    CUST_NAME VARCHAR (20)     NOT NULL,
    PRIMARY KEY (CUST_ID)
);
```

接着创建一个表来存储每个订单的详细信息：

```
CREATE TABLE ORDERS(
    ORDER_ID   INT           NOT NULL,
    ORDER_DETAIL VARCHAR (20)  NOT NULL,
    PRIMARY KEY (ORDER_ID)
);
```

最后，创建一个表来存储 CUST_ID 和 ORDER_ID 来记录同一顾客的所有订单：

```
CREATE TABLE CUSTMERORDERS(
    CUST_ID    INT           NOT NULL,
    ORDER_ID   INT           NOT NULL,
    SALE_DATE  DATETIME,
    PRIMARY KEY (CUST_ID, ORDER_ID)
);
```

第三范式（3NF）

一个数据表符合第三范式，当其满足：

- 符合第二范式；
- 所有的非主键字段都依赖于主键；

非主键字段之间的依赖关系存在于数据之中。例如下表中，街道（street）、城市（city）和省份（state）显然与邮政编码（zip Code）之间存在密不可分的关系。

```
CREATE TABLE CUSTOMERS(
    CUST_ID      INT            NOT NULL,
    CUST_NAME    VARCHAR (20)     NOT NULL,
    DOB          DATE,
    STREET       VARCHAR(200),
    CITY         VARCHAR(100),
    STATE        VARCHAR(100),
    ZIP          VARCHAR(12),
    EMAIL_ID     VARCHAR(256),
    PRIMARY KEY (CUST_ID)
);
```

邮政编码和地址之间的关系称作传递相关性（transitive dependency）。要使得数据表符合第三范式，需要将街道、城市、省份等字段移到另一张表中，可以称其为 Zip Code 表：

```
CREATE TABLE ADDRESS(
    ZIP          VARCHAR(12),
    STREET       VARCHAR(200),
    CITY         VARCHAR(100),
    STATE        VARCHAR(100),
    PRIMARY KEY (ZIP)
);
```

接着，按照如下方式更改 CUSTOMERS 表：

```
CREATE TABLE CUSTOMERS(
    CUST_ID      INT            NOT NULL,
    CUST_NAME    VARCHAR (20)     NOT NULL,
    DOB          DATE,
    ZIP          VARCHAR(12),
    EMAIL_ID     VARCHAR(256),
    PRIMARY KEY (CUST_ID)
);
```

移除传递相关性可以起到事半功倍的效果。首先是数据冗余度降低了，数据库体积因此缩小。第二个好处是保证数据完整性。当重复数据改变的时候，很有可能只更新部分数据，尤其是当其分布在数据库的各个地方的情况下。例如，如果地址和邮政编码分别存储在三个或者四个不同的数据表中，那么任何对邮编的改变，都需要对这三个或者四个表同时进行更改。

数据库

现在有很多种流行的关系型数据库管理系统可供选择使用。下面我们就简要介绍其中最为流行的几种，以帮助你对它们的基本特征做出比较。

MySQL

MySQL 是一个开源的 SQL 数据库管理系统，由瑞典公司 MySQL AB 开发。MySQL 的发音为“my ess-que-ell”，而 SQL 的发音则为“sequel”。

MySQL 对多种平台都有良好的支持，包括 Microsoft Windows、主要的 Linux 发行版、UNIX 和 Mac OS X 等。

MySQL 有免费和付费两种版本，免费或付费取决于其用途（非商业用途 / 商业用途）和所支持的特性。MySQL 附带了一个高效、多线程、多用户，并且非常健壮的 SQL 数据库服务器。

历史：

- 1994 年 Michael Widenius 和 David Axmark 开始开发 MySQL。
- 第一个内部版本于 1995 年 5 月 23 日发布。
- 1998 年 一月 8 日发布 Windows 版，支持 Windows 95 和 Windows NT。
- 3.23 版：2000 年 6 月发布 beta 版，2001 年 1 月产品发布。
- 4.0 版： 2002 年 8 月发布 beta 版， 2003 年 3 月产品发布。
- 4.01 版：2003 年 8 月发布 beta 版，Jyoti 公司开始采用 MySQL 用于数据库追踪。
- 4.1 版：2004 年 6 月发布 beta 版，2004 年 10 月产品发布。
- 5.0 版：2005 年 3 月 发布 beta 版，2005 年 10 月产品发布。
- 2008 年 2 月 26 日，Sun 公司收购 MySQL AB。
- 5.1 版：2008 年 11 月 27 日产品发布。

特性：

- 高性能

- 高可用性
- 可扩展性和灵活性
- 健壮的事务（Transaction）支持
- 在网络和数据仓库方面见长
- 全面的数据库应用开发支持
- 管理方便
- 开源、自由，而且 24 x 7 支持
- 总体费用最低

MS SQL Server

MS SQL Server 是微软公司开发的一款关系型数据库管理系统，它所采用的查询语言主要有：

- T-SQL
- ANSI-SQL

历史：

- 1987 年，Sybase 公司发布了用于 UNIX 的 SQL Server。
- 1988 年，微软、Sybase 和 Aston-Tate 将 SQL Server 移植到了 OS/2。
- 1989 年，微软、Sybase 和 Aston-Tate 将 SQL Server 发布 OS/2 平台的 SQL Server 1.0 版。
- 1990 年，SQL Server 1.1 发布，该版本包含对 Windows 3.0 的支持。
- Aston-Tate 退出 SQL Server 开发工作。
- 2000 年，微软发布 SQL Server 2000。
- 2001 年，微软发布 XML for SQL Server Web Release 1。
- 2002 年，微软发布 SQLXML 2.0（由 XML for SQL Server 更名而来）。
- 2002 年，微软发布 SQLXML 3.0。
- 2005 年 11 月 7 日，微软发布 SQL Server 2005。

特性：

- 高性能
- 高可用性
- 数据库镜像
- 数据库快照
- CLR 集成
- 服务代理
- 数据库定义语言（DDL）触发器
- 排名函数
- 基于行版本控制的隔离级别
- XML 集成
- TRY...CATCH
- 数据库邮件

ORACLE

Oracle 是一款由甲骨文公司开发的大型多用户关系型数据库管理系统。

Oracle 能够在多个终端在网络中同时发送请求和数据的情况下，有效管理它的资源——整个数据库的信息。

对于客户端/服务器架构的计算需求来说，Oracle 是绝佳的选择。Oracle 支持所有主流的操作系统的客户端和服务器版，包括 MSDOS、NetWare、UnixWare、OS/2 和大多数类 UNIX 系统。

历史：

Oracle 开始于 1977 年，截至 2009 年，它已经在业界走过了 32 个年头。

- 1977 年，拉里·埃里森、鲍勃·迈纳和爱德·奥茨共同创建了软件开发实验室（Software Development Laboratory），以从事开拓性的软件开发工作。
- 1979 年，Oracle 2.0 版发布，它是第一款商业关系型数据库管理系统，也是第一款 SQL 数据库。公司也在这一年更名为 Relational Software Inc（RSI）。

? 1981 年，RSI 公司开始为 Oracle 开发工具软件。

? 1982 年，RSI 更名为 Oracle Corporation。

? 1983 年，Oracle 3.0 发布，这一版由 C 语言重写而成，并且开始加入多平台支持。

? 1984 年，Oracle 4.0 发布，这一版开始加入并发控制、版本间读取一致性（multi-version read consistency）等特性。

? 2007 年，Oracle 发不了 Oracle 11g，新版数据库着力于对数据库分区更好的支持以及更容易进行数据迁移工作等。

特性:

? 并发

? 读取一致性

? 锁机制

? 支持数据库的静默模式

? 可移植性

? 自管理能力

? SQL*Plus

? ASM

? 调度器

? 资源管理器

? 数据库仓库

? 物化视图

? 位图索引

? 并行执行

? SQL 分析函数（Analytic SQL）

? 数据挖掘

? 分区

MS ACCESS

Access 是微软最受欢迎的产品之一，它是一款入门级的数据库管理系统。对于小型项目来说，Access 不仅便宜，而且功能强大。

MS Access 使用 Jet 数据库引擎，该数据库引擎使用了一种特殊的 SQL 方言（有时候称作 Jet SQL）作为其查询语言。

MS Access 包含在 MS Office 专业版套件中，拥有易用直观的图形用户界面。

- 1992年，Access 1.0 版发布。
- 1993年，Access 1.1 版发布，提升了与 Access Basic 语言的兼容性。
- Access 最重要的转变发生于 Access 97 到 Access 2000 的过程中。
- 2007 年，Access 2007 版开始支持新的数据库格式 ACCDB，该格式支持诸如多值字段和附加字段等复杂类型。

特性:

- 用户可以创建表、查询、表单和报表等，并可以用宏（macro）将其组合在一起。
- 支持以多种格式导入和导出数据，包括 Excel、Outlook、ASCII、dBase、Paradox、FoxPro、SQL Server、Oracle、ODBC 等等。
- 其专有格式为 Jet 数据库格式（MDB，Access 2007 及之后版本为 ACCDB），该格式可以在一个文件中同时包含应用程序和数据。这使得整个程序的分发非常方便，用户可以在离线环境中运行包含在其中的程序。
- Access 支持参数化查询。Access 中的查询和表可以通过 DAO 或者 ADO 等技术在其他程序（例如 VB6 或者 .NET）中引用。
- Access 是一款基于文件服务器的数据库。同其他客户端/服务器关系型数据库管理系统（RDBMS）不同，Access 没有内建对数据库触发器、存储过程以及事务日志等的支持。

语法

SQL 遵循一组称为“语法”的规则和指南。本教程列出了所有的 SQL 基础语法供你快速学习 SQL 之用。

所有的 SQL 语句都以下列关键字之一开始：SELECT、INSERT、UPDATE、DELETE、ALTER、DROP、CREATE、USE、SHOW，并以一个分号（;）结束。

有一点需要特别注意：SQL **不区分大小写**，也就是说 SELECT 和 select 在 SQL 语句中有相同的含义。然而，MySQL 在表的名称方面并不遵循此规定。所以，如果你在使用 MySQL 的话，你需要在程序中严格按照它们在数据库中名字进行使用。

> 本教程列出的所有的例子都在 MySQL 下进行了测试。

SQL SELECT 语句

```
SELECT column1, column2....columnN
FROM   table_name;
```

SQL DISTINCT 子句

```
SELECT DISTINCT column1, column2....columnN
FROM   table_name;
```

SQL WHERE 子句

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION;
```

SQL AND/OR 子句

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION-1 {AND|OR} CONDITION-2;
```

SQL IN 子句

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name IN (val-1, val-2,...val-N);
```

SQL BETWEEN 子句

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name BETWEEN val-1 AND val-2;
```

SQL LIKE 子句

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name LIKE { PATTERN };
```

SQL ORDER BY 子句

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION
ORDER BY column_name {ASC|DESC};
```

SQL GROUP BY 子句

```
SELECT SUM(column_name)
FROM   table_name
WHERE  CONDITION
GROUP BY column_name;
```

SQL COUNT 子句

```
SELECT COUNT(column_name)
FROM   table_name
WHERE  CONDITION;
```

SQL HAVING 子句

```
SELECT SUM(column_name)
FROM   table_name
WHERE  CONDITION
GROUP BY column_name
HAVING (arithematic function condition);
```

SQL CREATE TABLE 语句

```
CREATE TABLE table_name(
column1 datatype,
column2 datatype,
column3 datatype,
.....
columnN datatype,
PRIMARY KEY( one or more columns )
);
```

SQL DROP TABLE 语句

```
DROP TABLE table_name;
```

SQL CREATE INDEX 语句

```
CREATE UNIQUE INDEX index_name
ON table_name ( column1, column2,...columnN);
```

SQL DROP INDEX 语句

```
ALTER TABLE table_name
DROP INDEX index_name;
```

SQL DESC 语句

```
DESC table_name;
```

SQL TRUNCATE TABLE 语句

```
TRUNCATE TABLE table_name;
```

SQL ALTER TABLE 语句（重命名）

```
ALTER TABLE table_name RENAME TO new_table_name;
```

SQL INSERT INTO 语句

```
INSERT INTO table_name( column1, column2....columnN)
VALUES ( value1, value2....valueN);
```

SQL UPDATE 语句

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

SQL DELETE 语句

```
DELETE FROM table_name
WHERE  {CONDITION};
```

SQL CREATE DATABASE 语句

```
CREATE DATABASE database_name;
```

SQL DROP DATABASE 语句

```
DROP DATABASE database_name;
```

SQL USE 语句

```
USE database_name;
```

SQL COMMIT 语句

```
COMMIT;
```

SQL ROLLBACK 语句

```
ROLLBACK;
```

数据类型

SQL 数据类型是一种属性，它指定了任何 SQL 对象中数据的类型。在 SQL 中，任意一个列、变量或者表达式都有其数据类型。

创建表的时候，你会用到这些数据类型。你应该根据需要为表中的每一个列选择合适的数据类型。

SQL Server 提供了六种数据类型供你使用：

精确数值数据类型

数据类型	下限	上限
bigint	-9,223,372,036,854,770,000	9,223,372,036,854,770,000
int	-2,147,483,648	2,147,483,647
smallint	-32,768	32,767
tinyint	0	255
bit	0	1
decimal	1E+38	10^38 -1
numeric	1E+38	10^38 -1
money	-922,337,203,685,477.00	922,337,203,685,477.00
smallmoney	-214,748.36	214,748.36

近似数值数据类型

数据类型	下限	上限
float	-1.79E + 308	1.79E + 308
real	-3.40E + 38	3.40E + 38

日期和时间数据类型

数据类型	下限	上限
datetime	Jan 1, 1753	31-Dec-99
smalldatetime	1-Jan-00	6-Jun-79
date	存储一个日期数据，例如 June 30, 1991	
time	存储一个时间数据，例如 12:30 P.M.	

注意：datetime 的时间和精度为 3.33 ms，而 smalldatetime 的时间精度为 1 min。

字符串数据类型

数据类型	下限	上限
char	char	最大长度为 8,000 字符。（定长非 Unicode 字符）
varchar	varchar	最大长度为 8,000 字符。（变长非 Unicode 数据）
varchar(max)	varchar(max)	最大长度为 231 字符, 变长非 Unicode 数据 (仅限 SQL Server 2005).
text	text	变长非 Unicode 字符数据，最大长度 2,147,483,647 字符。

Unicode 字符串数据类型

数据类型	描述
nchar	最大长度 4000 字符。（定长 Unicode 字符串）
nvarchar	最大长度 4000 字符。（变长 Unicode 字符串）
nvarchar(max)	最大长度 231 字符 。（仅限 SQL Server 2005）。（变长 Unicode 字符串）
ntext	最大长度 1,073,741,823 字符。（变长 Unicode 字符串）

二进制数据类型

数据类型	描述
binary	最大长度 8000 字节。（定长二进制数据）
varbinary	最大长度 8000 字节。（变长二进制数据）
varbinary(max)	最大长度 231字节（仅限 SQLServer 2005）。（变长二进制数据）
image	最大长度 2,147,483,647 字节。（变长二进制数据）

其他数据类型

数据类型	描述
sql_variant	存储多种 SQL 支持的数据类型，text、ntext、timestamp 除外。
timestamp	一个数据库级的唯一值，每当有行更新此数据就会更新。
uniqueidentifier	全局唯一标识符（GUID）
xml	存储 XML 数据。你可以在列或者变量中存储 XML 实例。（仅限 SQL Server 2005）
cursor	指向 cursor 对象。
table	存储结果，以备后用。

操作符

每个操作符都是一个保留字，主要用于在 SQL 语句的 WHERE 子句中执行各种操作，例如比较和算术运算等。

操作符在 SQL 语句中指定了条件，并可以将同一语句中的不同条件连接起来。

- 算术运算符
- 比较运算符
- 逻辑运算符
- 用于否定条件的运算符

SQL 算术运算符

这里一些有关 SQL 算术运算符如何使用的简单示例：

```
SQL> select 10+ 20;
+--------+
| 10+ 20 |
+--------+
|    30 |
+--------+
1 row in set (0.00 sec)

SQL> select 10 * 20;
+---------+
| 10 * 20 |
+---------+
|    200 |
+---------+
1 row in set (0.00 sec)

SQL> select 10 / 5;
+--------+
| 10 / 5 |
+--------+
| 2.0000 |
+--------+
1 row in set (0.03 sec)

SQL> select 12 %  5;
```

```
+---------+
| 12 % 5 |
+---------+
|     2 |
+---------+
1 row in set (0.00 sec)
```

假设变量 a 的值为 10， 变量 b 的值为 20，那么：

操作符	描述	示例
+	相加：将符号两边的数值加起来。	a + b 得 30
−	相减：从最边的操作数中减去右边的操作数。	a − b 得 −10
*	相乘：将两边的操作数相乘。	a * b 得 200
/	相除：用右边的操作数除以左边的操作数。	b / a 得 2
%	取余：用右边的操作数除以左边的操作数，并返回余数。	b % a 得 0

SQL 比较运算符

考虑 CUSTOMERS 表，表中的记录如下所示：

```
SQL> SELECT * FROM CUSTOMERS;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     |  1500.00 |
| 3 | kaushik  | 23 | Kota      |  2000.00 |
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   | 27 | Bhopal    |  8500.00 |
| 6 | Komal    | 22 | MP        |  4500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
7 rows in set (0.00 sec)
```

下面是一些关于如何使用 SQL 比较运算符的简单示例：

```
SQL> SELECT * FROM CUSTOMERS WHERE SALARY > 5000;
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
| 4 | Chaitali |  25 | Mumbai  |  6500.00 |
| 5 | Hardik   | 27 | Bhopal  |  8500.00 |
| 7 | Muffy    | 24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
```

```
3 rows in set (0.00 sec)

SQL>  SELECT * FROM CUSTOMERS WHERE SALARY = 2000;
+----+---------+-----+-----------+---------+
| ID | NAME    | AGE | ADDRESS   | SALARY  |
+----+---------+-----+-----------+---------+
|  1 | Ramesh  |  32 | Ahmedabad | 2000.00 |
|  3 | kaushik |  23 | Kota      | 2000.00 |
+----+---------+-----+-----------+---------+
2 rows in set (0.00 sec)

SQL>  SELECT * FROM CUSTOMERS WHERE SALARY != 2000;
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  2 | Khilan   |  25 | Delhi   |  1500.00 |
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  5 | Hardik   |  27 | Bhopal  |  8500.00 |
|  6 | Komal    |  22 | MP      |  4500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
5 rows in set (0.00 sec)

SQL> SELECT * FROM CUSTOMERS WHERE SALARY <> 2000;
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  2 | Khilan   |  25 | Delhi   |  1500.00 |
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  5 | Hardik   |  27 | Bhopal  |  8500.00 |
|  6 | Komal    |  22 | MP      |  4500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
5 rows in set (0.00 sec)

SQL> SELECT * FROM CUSTOMERS WHERE SALARY >= 6500;
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  5 | Hardik   |  27 | Bhopal  |  8500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
3 rows in set (0.00 sec)
```

假设变量 a 的值为 10， 变量 b 的值为 20，那么：

操作符	描述	示例
=	检查两个操作数的值是否相等，是的话返回 true。	(a = b) 不为 true。
!=	检查两个操作数的值是否相等，如果不等则返回 true。	(a != b) 为 true。
<>	检查两个操作数的值是否相等，如果不等则返回 true。	(a <> b) 为真。
>	检查左边的操作数是否大于右边的操作数，是的话返回真。	(a > b) 不为 true。
<	检查左边的操作数是否小于右边的操作数，是的话返回真。	(a < b) 为 true。
>=	检查左边的操作数是否大于或等于右边的操作数，是的话返回真。	(a >= b) 不为 true。
<=	检查左边的操作数是否小于或等于右边的操作数，是的话返回真。	(a <= b) 为 true。
!<	检查左边的操作数是否不小于右边的操作数，是的话返回真。	(a !< b) 为 false。
!>	检查左边的操作数是否不大于右边的操作数，是的话返回真。	(a !> b) 为 true。

SQL 逻辑运算符

考虑 CUSTOMERS 表，表中的记录如下所示：

```
SQL> SELECT * FROM CUSTOMERS;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
7 rows in set (0.00 sec)
```

下面是一些关于如何使用 SQL 逻辑运算符的简单示例：

```
SQL> SELECT * FROM CUSTOMERS WHERE AGE >= 25 AND SALARY >= 6500;
+----+----------+-----+---------+---------+
| ID | NAME     | AGE | ADDRESS | SALARY  |
+----+----------+-----+---------+---------+
|  4 | Chaitali |  25 | Mumbai  | 6500.00 |
|  5 | Hardik   |  27 | Bhopal  | 8500.00 |
+----+----------+-----+---------+---------+
2 rows in set (0.00 sec)


SQL> SELECT * FROM CUSTOMERS WHERE AGE >= 25 OR SALARY >= 6500;
+----+----------+-----+-----------+----------+
```

```
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
| 2 | Khilan   |  25 | Delhi     |  1500.00 |
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   |  27 | Bhopal    |  8500.00 |
| 7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
5 rows in set (0.00 sec)

SQL>  SELECT * FROM CUSTOMERS WHERE AGE IS NOT NULL;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
| 2 | Khilan   |  25 | Delhi     |  1500.00 |
| 3 | kaushik  |  23 | Kota      |  2000.00 |
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   |  27 | Bhopal    |  8500.00 |
| 6 | Komal    |  22 | MP        |  4500.00 |
| 7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
7 rows in set (0.00 sec)

SQL> SELECT * FROM CUSTOMERS WHERE NAME LIKE 'Ko%';
+----+-------+-----+---------+---------+
| ID | NAME  | AGE | ADDRESS | SALARY  |
+----+-------+-----+---------+---------+
| 6 | Komal |  22 | MP      | 4500.00 |
+----+-------+-----+---------+---------+
1 row in set (0.00 sec)

SQL> SELECT * FROM CUSTOMERS WHERE AGE IN ( 25, 27 );
+----+----------+-----+---------+---------+
| ID | NAME     | AGE | ADDRESS | SALARY  |
+----+----------+-----+---------+---------+
| 2 | Khilan   |  25 | Delhi   | 1500.00 |
| 4 | Chaitali |  25 | Mumbai  | 6500.00 |
| 5 | Hardik   |  27 | Bhopal  | 8500.00 |
+----+----------+-----+---------+---------+
3 rows in set (0.00 sec)

SQL> SELECT * FROM CUSTOMERS WHERE AGE BETWEEN 25 AND 27;
+----+----------+-----+---------+---------+
| ID | NAME     | AGE | ADDRESS | SALARY  |
```

```
+----+----------+-----+---------+---------+
| 2 | Khilan   | 25 | Delhi   | 1500.00 |
| 4 | Chaitali | 25 | Mumbai  | 6500.00 |
| 5 | Hardik   | 27 | Bhopal  | 8500.00 |
+----+----------+-----+---------+---------+
3 rows in set (0.00 sec)

SQL> SELECT AGE FROM CUSTOMERS
WHERE EXISTS (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);
+-----+
| AGE |
+-----+
| 32 |
| 25 |
| 23 |
| 25 |
| 27 |
| 22 |
| 24 |
+-----+
7 rows in set (0.02 sec)

SQL> SELECT * FROM CUSTOMERS
WHERE AGE > ALL (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);
+----+--------+-----+-----------+---------+
| ID | NAME   | AGE | ADDRESS   | SALARY  |
+----+--------+-----+-----------+---------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
+----+--------+-----+-----------+---------+
1 row in set (0.02 sec)

SQL> SELECT * FROM CUSTOMERS
WHERE AGE > ANY (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);
+----+----------+-----+-----------+---------+
| ID | NAME     | AGE | ADDRESS   | SALARY  |
+----+----------+-----+-----------+---------+
| 1 | Ramesh   | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan   | 25 | Delhi     | 1500.00 |
| 4 | Chaitali | 25 | Mumbai    | 6500.00 |
| 5 | Hardik   | 27 | Bhopal    | 8500.00 |
+----+----------+-----+-----------+---------+
4 rows in set (0.00 sec)
```

下面列出了 SQL 中可用的逻辑运算符。

运算符	描述

ALL	ALL 运算符用于将一个值同另一个值集中所有的值进行比较。
AND	AND 运算符使得在 WHERE 子句中可以同时存在多个条件。
ANY	ANY 运算符用于将一个值同条件所指定的列表中的任意值相比较。
BETWEEN	给定最小值和最大值，BETWEEN 运算符可以用于搜索区间内的值。
EXISTS	EXISTS 运算符用于在表中搜索符合特定条件的行。
IN	IN 运算符用于将某个值同指定的一列字面值相比较。
LIKE	LIKE 运算符用于使用通配符对某个值和与其相似的值做出比较。
NOT	NOT 操作符反转它所作用的操作符的意义。例如，NOT EXISTS、NOT BETWEEN、NOT IN 等。这是一个求反运算符。
OR	OR 运算符用于在 SQL 语句中连接多个条件。
IS NULL	NULL Operator 用于将某个值同 NULL 作比较。
UNIQUE	UNIQUE 运算符检查指定表的所有行，以确定没有重复。

表达式

表达式是一个或者多个值、运算符和 SQL 函数的组合。每个表达式都有值，通过求值可以得到。

SQL 表达式看起来就像数学公式一样，它们以查询语言写就。你也可以用它们在数据库中查询符合特定条件的数据。

语法

考虑如下所示的 SELECT 语句的基本语法：

```
SELECT column1, column2, columnN
FROM table_name
WHERE [CONDITION|EXPRESSION];
```

SQL 表达式有很多种不同的类型，如下所示：

布尔表达式

SQL 布尔表达式以单值条件检索数据，其语法如下：

```
SELECT column1, column2, columnN
FROM table_name
WHERE SINGLE VALUE MATCHTING EXPRESSION;
```

考虑如下所示的客户信息表：

```
SQL> SELECT * FROM CUSTOMERS;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
7 rows in set (0.00 sec)
```

下面是一个展示 SQL 布尔表达式的简单例子：

```
SQL> SELECT * FROM CUSTOMERS WHERE SALARY = 10000;
+----+-------+-----+---------+----------+
| ID | NAME  | AGE | ADDRESS | SALARY   |
+----+-------+-----+---------+----------+
|  7 | Muffy |  24 | Indore  | 10000.00 |
+----+-------+-----+---------+----------+
1 row in set (0.00 sec)
```

数值表达式

数值表达式用于在查询中执行数学运算。其语法如下：

```
SELECT numerical_expression as  OPERATION_NAME
[FROM table_name
WHERE CONDITION] ;
```

这里 numerical_expression 为数学运算或者任何公式。下面是一个说明 SQL 数值表达式用法的简单例子：

```
SQL> SELECT (15 + 6) AS ADDITION
+-----------+
| ADDITION |
+-----------+
|       21 |
+-----------+
1 row in set (0.00 sec)
```

SQL 有一系列的内建函数，例如 avg()、sum()、count() 等，这些函数用于在表上或者表中的特定列上执行聚合数据运算。

```
SQL> SELECT COUNT(*) AS "RECORDS" FROM CUSTOMERS;
+---------+
| RECORDS |
+---------+
|       7 |
+---------+
1 row in set (0.00 sec)
```

时间表达式

时间表达式返回当前系统的日期和时间：

```
SQL>  SELECT CURRENT_TIMESTAMP;
+---------------------+
| Current_Timestamp   |
+---------------------+
| 2009-11-12 06:40:23 |
+---------------------+
1 row in set (0.00 sec)
```

下面是一个日期表达式：

```
SQL>  SELECT  GETDATE();;
+-------------------------+
| GETDATE                 |
+-------------------------+
| 2009-10-22 12:07:18.140 |
+-------------------------+
1 row in set (0.00 sec)
```

创建数据库

SQL CREATE DATABASE 语句用于创建新的 SQL 数据库。

语法

CREATE DATABASE 的基本语法如下所示：

```
CREATE DATABASE DatabaseName;
```

在 RDBMS 中，数据库的名字应该是唯一的。

示例

如果你先想要创建一个新数据库 <testDB>，那么 CREATE DATABASE 语句应该这么写：

```
SQL> CREATE DATABASE testDB;
```

创建数据库之前，请确保你有管理员权限。数据库一旦创建成功就会出现在数据库列表中。

```
SQL> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| AMROOD             |
| TUTORIALSPOINT     |
| mysql              |
| orig               |
| test               |
| testDB             |
+--------------------+
7 rows in set (0.00 sec)
```

删除数据库

SQL DROP DATABASE 语句用于删除现存的数据库。

语法

DROP DATABASE 的基本语法如下：

```
DROP DATABASE DatabaseName;
```

无论任何时候，RDBMS 中数据库的名字都应该是唯一的。

示例

如果你想要删除数据库 <testDB>，那么 DROP DATABASE 语句应该这么写：

```
SQL> DROP DATABASE testDB;
```

注意：执行数据库删除操作应当十分谨慎，因为数据库一旦删除，存储的所有数据都会丢失。

删除任何数据库之前，请确保你有管理员权限。数据库删除之后，你可以在数据库列表中看到变化：

```
SQL> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| AMROOD             |
| TUTORIALSPOINT     |
| mysql              |
| orig               |
| test               |
+--------------------+
6 rows in set (0.00 sec)
```

选择数据库，USE 语句

如果你的数据库架构中有多个数据库同时存在，那么在开始操作之前必须先选定其中一个。

SQL USE 语句用于选取当前数据库架构中存在的任一数据库。

语法：

USE 语句的基本语法如下：

```
USE DatabaseName;
```

再次强调，RDBMS 中数据库的名字应该唯一。

示例：

按照以下方式查看所有可用的数据库：

```
SQL> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| AMROOD             |
| TUTORIALSPOINT     |
| mysql              |
| orig               |
| test               |
+--------------------+
6 rows in set (0.00 sec)
```

如果你想要操作 AMROOD 数据库的话，可以执行下面的 SQL 命令选中它，然后开始执行你所需要的操作。

```
SQL> USE AMROOD;
```

创建表

创建一个基本的表需要做的工作包括：命名表、定义列和各列的数据类型。

SQL 语言使用 CREATE TABLE 语句来创建新表。

语法：

CREATE TABLE 的基本语法如下所示：

```
CREATE TABLE table_name(
   column1 datatype,
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
   PRIMARY KEY( one or more columns )
);
```

CREATE TABLE 向数据库系统指明了你的意图。在此例中，你想要创建一个新表，新表的唯一名称（或者说标识符）紧跟在 CREATE TABLE 后面。

随后的圆括号以列表的形式定义了表中的列以及各列所属的数据类型。下面的示例对该创建新表的语法做出了更清晰的阐释。

> 将 CREATE TABLE 语句和 SELECT 语句结合起来可以创建现有表的副本。详细信息请见利用现有表创建新表.

示例：

下面的示例创建了一个 CUSTOMERS 表，主键为 ID，某些字段具有 NOT NULL 的约束，表示在创建新的记录时这些字段不能为 NULL。

```
SQL> CREATE TABLE CUSTOMERS(
   ID   INT              NOT NULL,
   NAME VARCHAR (20)     NOT NULL,
   AGE  INT              NOT NULL,
   ADDRESS  CHAR (25) ,
   SALARY   DECIMAL (18, 2),
```

```
   PRIMARY KEY (ID)
);
```

你可以通过查看 SQL 服务器返回的消息来确定新表创建成功，或者也可以像下面这样使用 DESC 命令：

```
SQL> DESC CUSTOMERS;
+---------+---------------+------+-----+---------+-------+
| Field   | Type          | Null | Key | Default | Extra |
+---------+---------------+------+-----+---------+-------+
| ID      | int(11)       | NO   | PRI |         |       |
| NAME    | varchar(20)   | NO   |     |         |       |
| AGE     | int(11)       | NO   |     |         |       |
| ADDRESS | char(25)      | YES  |     | NULL    |       |
| SALARY  | decimal(18,2) | YES  |     | NULL    |       |
+---------+---------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

现在数据库中已经有 CUSTOMERS 表了，你可以用它来存储和客户有关的信息。

删除表

SQL DROP TABLE 语句用于移除表定义以及表中所有的数据、索引、触发器、约束和权限设置。

注意：使用此命令应当特别小心，因为数据表一旦被删除，表中所有的信息就会永久丢失。

语法：

DROP TABLE 语句的基本语法如下所示：

```
DROP TABLE table_name;
```

示例：

先确认操作的是 CUSTOMERS 表，才能将其从数据库中删除：

```
SQL> DESC CUSTOMERS;
+---------+---------------+------+-----+---------+-------+
| Field   | Type          | Null | Key | Default | Extra |
+---------+---------------+------+-----+---------+-------+
| ID      | int(11)       | NO   | PRI |         |       |
| NAME    | varchar(20)   | NO   |     |         |       |
| AGE     | int(11)       | NO   |     |         |       |
| ADDRESS | char(25)      | YES  |     | NULL    |       |
| SALARY  | decimal(18,2) | YES  |     | NULL    |       |
+---------+---------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

返回结果表明 CUSTOMERS 表在数据库中，接着让我们用下面的命令删除它：

```
SQL> DROP TABLE CUSTOMERS;
Query OK, 0 rows affected (0.01 sec)
```

现在，如果你再用 DESC 命令的话，会得到如下所示的错误信息：

```
SQL> DESC CUSTOMERS;
ERROR 1146 (42S02): Table 'TEST.CUSTOMERS' doesn't exist
```

这里，TEST 是示例所用的数据库的名称。

INSERT 语句

SQL INSERT INTO 语句用于向数据库中的表添加新行。

语法:

INSERT INTO 有两种基本的语法，第一种语法格式如下:

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)
VALUES (value1, value2, value3,...valueN);
```

这里 column1, column2,...columnN 是表中字段的名字，你必须为新记录的这些字段填充数据。

如果要为表中所有的字段都添加数据的话，就不需要指定字段名了。不过这种情况下，必须保证值的顺序按照表中字段出现的顺序排列。 此时 SQL INSERT INTO 语句的语法如下:

```
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

示例:

下面的语句将在 CUSTOMERS 表中创建六条新记录:

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 22, 'MP', 4500.00 );
```

你也可以使用第二种语法在表中创建一条新记录：

```
INSERT INTO CUSTOMERS
VALUES (7, 'Muffy', 24, 'Indore', 10000.00 );
```

上面所有的语句执行完毕之后，CUSTOMERS 表中所有的记录应当如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

利用

你还可以用 SELECT 语句将一个表中相应字段的数据填充到另一个表中，其语法形式如下：

```
INSERT INTO first_table_name [(column1, column2, ... columnN)]
   SELECT column1, column2, ...columnN
   FROM second_table_name
   [WHERE condition];
```

SELECT 语句

SQL SELECT 语句用于从数据库的表中取回所需的数据，并以表的形式返回。返回的表被称作结果集。

语法：

SELECT 语句的基本语法如下：

```
SELECT column1, column2, columnN FROM table_name;
```

这里，column1, column2...是你想要从表中取回的字段。如果要取回表中所有字段的话，可以使用下面的语法：

```
SELECT * FROM table_name;
```

示例：

考虑 CUSTOMERS 表，该表包含的记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面的例子将从 CUSTOMERS 表中获取客户的 ID、Name 和 Salary 字段：

```
SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS;
```

运行结果如下所示：

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  1 | Ramesh   |  2000.00 |
|  2 | Khilan   |  1500.00 |
```

```
| 3 | kaushik  |  2000.00 |
| 4 | Chaitali |  6500.00 |
| 5 | Hardik   |  8500.00 |
| 6 | Komal    |  4500.00 |
| 7 | Muffy    | 10000.00 |
+----+----------+----------+
```

如果想要取回 CUSTOMERS 表中所有的字段的话，SQL 查询应该这么写：

```
SQL> SELECT * FROM CUSTOMERS;
```

运行结果如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
| 2 | Khilan   |  25 | Delhi     |  1500.00 |
| 3 | kaushik  |  23 | Kota      |  2000.00 |
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   |  27 | Bhopal    |  8500.00 |
| 6 | Komal    |  22 | MP        |  4500.00 |
| 7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

WHERE 子句

SQL WHERE 子句用于有条件地从单个表中取回数据或者将多个表进行合并。

如果条件满足，则查询只返回表中满足条件的值。你可以用 WHERE 子句来过滤查询结果，只获取必要的记录。

WHERE 子句不仅可以用于 SELECT 语句，还可以用于 UPDATE、DELETE 等语句，其用法见后面的章节。

语法：

在 SELECT 语句中使用 WHERE 子句的基本句法如下：

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

在指定条件时，可以使用关系运算符和逻辑运算符，例如 `>`、`<`、`=`、`LIKE`、`NOT` 等。

示例

考虑 CUSTOMERS 表，表中含有如下所示的记录：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面的示例将从 CUSTOMERS 表中选取 Salary 字段的值大于 2000 的所有记录，并返回这些记录的 ID、Name、Salary 三个字段。

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000;
```

运行结果如下所示：

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  4 | Chaitali |  6500.00 |
|  5 | Hardik   |  8500.00 |
|  6 | Komal    |  4500.00 |
|  7 | Muffy    | 10000.00 |
+----+----------+----------+
```

下面的示例将从 CUSTOMERS 表中选取名字为 Hardik 的客户的记录，并返回其 ID、Name 和 Salary 三个字段。这里值得注意的是，所有的字符串都必须写在单引号中，就像上面的例子中所有的数值都不能放在引号中一样。

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE NAME = 'Hardik';
```

结果如下所示：

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  5 | Hardik   |  8500.00 |
+----+----------+----------+
```

AND 和 OR 连接运算符

SQL AND 和 OR 运算符可以将多个条件结合在一起，从而过滤 SQL 语句的返回结果。这两个运算符被称作连接运算符。

AND 运算符

AND 运算符使得 SQL 语句的 WHERE 子句中可以同时存在多个条件。

语法

在 WHERE 子句中使用 AND 运算符的基本语法如下：

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] AND [condition2]...AND [conditionN];
```

你可以将 N 个条件用 AND 运算符结合在一起。对于 SQL 语句要执行的动作来说——无论是事务还是查询，AND 运算符连接的所有条件都必须为 TRUE。

示例

考虑如下所示的 CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     |  1500.00 |
| 3 | kaushik  | 23 | Kota      |  2000.00 |
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   | 27 | Bhopal    |  8500.00 |
| 6 | Komal    | 22 | MP        |  4500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面的示例将从 CUSTOMERS 表中选取所有 Salary 大于 2000 且 Age 小于 25 的记录，并返回其 ID、Name 和 Salary 字段。

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000 AND age < 25;
```

结果如下所示：

```
+----+-------+----------+
| ID | NAME  | SALARY   |
+----+-------+----------+
|  6 | Komal |  4500.00 |
|  7 | Muffy | 10000.00 |
+----+-------+----------+
```

OR 运算符

OR 运算符用于将 SQL 语句中 WHERE 子句的多个条件结合起来。

语法

在 WHERE 子句中使用 OR 运算符的基本语法如下：

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] OR [condition2]...OR [conditionN]
```

你可以将 N 个条件用 OR 运算符结合在一起。对于 SQL 语句要执行的动作来说——无论是事务还是查询，OR 运算符连接的所有条件中只需要有一个为 TRUE 即可。

示例

考虑如下所示的 CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
```

```
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面的示例将从 CUSTOMERS 表中选取所有 Salary 大于 2000 或 Age 小于 25 的记录，并返回其 ID、Name 和 Salary 字段。

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000 OR age < 25;
```

结果如下所示：

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
| 3 | kaushik  | 2000.00 |
| 4 | Chaitali | 6500.00 |
| 5 | Hardik   | 8500.00 |
| 6 | Komal    | 4500.00 |
| 7 | Muffy    | 10000.00 |
+----+----------+----------+
```

UPDATE 语句

SQL UPDATE 语句用于修改表中现有的记录。

你可以在 UPDATE 语句中使用 WHERE 子句来修改选定的记录，否则所有记录都会收到影响。

语法：

带 WHERE 子句的 UPDATE 语句的基本语法如下：

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

WHERE 子句中，你可以将 N 个条件用 AND 或者 OR 连接在一起。

例子：

考虑 CUSTOMERS 表，表中记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     |  1500.00 |
| 3 | kaushik  | 23 | Kota      |  2000.00 |
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   | 27 | Bhopal    |  8500.00 |
| 6 | Komal    | 22 | MP        |  4500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下例将会更新 ID 为 6 的客户的 ADDRESS 字段：

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune'
WHERE ID = 6;
```

更新之后，表中的记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     |  1500.00 |
| 3 | kaushik  | 23 | Kota      |  2000.00 |
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   | 27 | Bhopal    |  8500.00 |
| 6 | Komal    | 22 | Pune      |  4500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

如果你想要修改 CUSTOMERS 表中所有记录的 ADDRESS 和 SALARY 字段，只要把 WHERE 子句去掉即可。此时，UPDATE 语句如下所示：

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune', SALARY = 1000.00;
```

上述语句执行后，CUSTOMERS 表中的记录如下：

```
+----+----------+-----+---------+---------+
| ID | NAME     | AGE | ADDRESS | SALARY  |
+----+----------+-----+---------+---------+
| 1 | Ramesh   | 32 | Pune    | 1000.00 |
| 2 | Khilan   | 25 | Pune    | 1000.00 |
| 3 | kaushik  | 23 | Pune    | 1000.00 |
| 4 | Chaitali |  25 | Pune    | 1000.00 |
| 5 | Hardik   | 27 | Pune    | 1000.00 |
| 6 | Komal    | 22 | Pune    | 1000.00 |
| 7 | Muffy    | 24 | Pune    | 1000.00 |
+----+----------+-----+---------+---------+
```

DELETE 语句

SQL DELETE 语句用于删除表中现有的记录。

你可以在 DELETE 语句中使用 WHERE 子句来删除选定的记录，否则所有的记录都会被删除。

语法：

带 WHERE 子句的 DELETE 语句的基本语法如下：

```
DELETE FROM table_name
WHERE [condition];
```

WHERE 子句中，你可以将 N 个条件用 AND 或者 OR 连接在一起。

例子：

考虑 CUSTOMERS 表，表中记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面的示例代码将从中**删除** ID 为 6 的客户：

```
SQL> DELETE FROM CUSTOMERS
WHERE ID = 6;
```

上述代码运行之后，CUSTOMERS 表中的记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
```

```
| 1 | Ramesh  | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan  | 25 | Delhi     | 1500.00 |
| 3 | kaushik | 23 | Kota      | 2000.00 |
| 4 | Chaitali |  25 | Mumbai    | 6500.00 |
| 5 | Hardik  | 27 | Bhopal    | 8500.00 |
| 7 | Muffy   | 24 | Indore    |10000.00 |
+----+----------+-----+-----------+----------+
```

如果你想要删除 CUSTOMERS 表中所有的记录的话，只要将 WHERE 子句去掉即可。此时，DELETE 语句如下所示：

```
SQL> DELETE FROM CUSTOMERS;
```

现在，CUSTOMERS 表中就空空如也了。

LIKE 子句

SQL LIKE 子句通过通配符来将一个值同其他相似的值作比较。可以同 LIKE 运算符一起使用的通配符有两个：

？ 百分号（%）

？ 下划线（_）

百分号代表零个、一个或者多个字符。下划线则代表单个数字或者字符。两个符号可以一起使用。

语法：

% 和 _ 的基本语法如下：

```
SELECT FROM table_name
WHERE column LIKE 'XXXX%'

or

SELECT FROM table_name
WHERE column LIKE '%XXXX%'

or

SELECT FROM table_name
WHERE column LIKE 'XXXX_'

or

SELECT FROM table_name
WHERE column LIKE '_XXXX'

or

SELECT FROM table_name
WHERE column LIKE '_XXXX_'
```

你可以将多个条件用 AND 或者 OR　连接在一起。这里，XXXX　为任何数字值或者字符串。

示例:

下面这些示例中，每个 WHERE 子句都有不同的 LIKE 子句，展示了 % 和 _ 的用法:

语句	描述
WHERE SALARY LIKE '200%'	找出所有 200 打头的值
WHERE SALARY LIKE '%200%'	找出所有含有 200 的值
WHERE SALARY LIKE '_00%'	找出所有第二位和第三位为 0 的值
WHERE SALARY LIKE '2_%_%'	找出所有以 2 开始，并且长度至少为 3 的值
WHERE SALARY LIKE '%2'	找出所有以 2 结尾的值
WHERE SALARY LIKE '_2%3'	找出所有第二位为 2，并且以3结束的值
WHERE SALARY LIKE '2___3'	找出所有以 2 开头以 3 结束的五位数

让我们来看一个真实的例子，考虑含有如下所示记录的 CUSTOMERS 表:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     |  1500.00 |
| 3 | kaushik  | 23 | Kota      |  2000.00 |
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   | 27 | Bhopal    |  8500.00 |
| 6 | Komal    | 22 | MP        |  4500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面的例子将 CUSTOMERS 表中 SALARY 字段以 200 开始的记录显示出来:

```
SQL> SELECT * FROM CUSTOMERS
WHERE SALARY LIKE '200%';
```

结果如下所示:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 3 | kaushik  | 23 | Kota      |  2000.00 |
+----+----------+-----+-----------+----------+
```

TOP、LIMIT 和 ROWNUM 子句

SQL TOP 子句用于从一张数据表中取回前 N 个或者 X% 的记录。

注意：所有的数据库系统都不支持 TOP 子句。例如，MySQL 支持 LIMIT 子句，用以取回有限数量的记录，而 Oracle 则使用 ROWNUM 子句来实现这一功能。

语法

在 SELECT 语句中使用 TOP 子句的基本语法如下所示：

```
SELECT TOP number|percent column_name(s)
FROM table_name
WHERE [condition]
```

示例

考虑含有如下所示记录的 CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     |  1500.00 |
| 3 | kaushik  | 23 | Kota      |  2000.00 |
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   | 27 | Bhopal    |  8500.00 |
| 6 | Komal    | 22 | MP        |  4500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面的例子将从 CUSTOMERS 表中取回前 3 条记录：

```
SQL> SELECT TOP 3 * FROM CUSTOMERS;
```

结果如下所示：

```
+----+---------+-----+-----------+---------+
| ID | NAME    | AGE | ADDRESS   | SALARY  |
+----+---------+-----+-----------+---------+
```

```
| 1 | Ramesh  |  32 | Ahmedabad | 2000.00 |
|  2 | Khilan  |  25 | Delhi     | 1500.00 |
|  3 | kaushik |  23 | Kota      | 2000.00 |
+----+---------+-----+-----------+---------+
```

如果你在使用 MySQL 数据库服务器，那么等价的例子如下所示：

```
SQL> SELECT * FROM CUSTOMERS
LIMIT 3;
```

结果如下所示：

```
+----+---------+-----+-----------+---------+
| ID | NAME    | AGE | ADDRESS   | SALARY  |
+----+---------+-----+-----------+---------+
| 1 | Ramesh  |  32 | Ahmedabad | 2000.00 |
|  2 | Khilan  |  25 | Delhi     | 1500.00 |
|  3 | kaushik |  23 | Kota      | 2000.00 |
+----+---------+-----+-----------+---------+
```

如果你在使用 Oracle 数据库服务器，那么等价的例子如下所示：

```
SQL> SELECT * FROM CUSTOMERS
WHERE ROWNUM <= 3;
```

结果如下所示：

```
+----+---------+-----+-----------+---------+
| ID | NAME    | AGE | ADDRESS   | SALARY  |
+----+---------+-----+-----------+---------+
| 1 | Ramesh  |  32 | Ahmedabad | 2000.00 |
|  2 | Khilan  |  25 | Delhi     | 1500.00 |
|  3 | kaushik |  23 | Kota      | 2000.00 |
+----+---------+-----+-----------+---------+
```

ORDER BY 子句

SQL ORDER BY 子句根据一列或者多列的值，按照升序或者降序排列数据。某些数据库默认以升序排列查询结果。

语法

ORDER BY 子句的基本语法如下所示：

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

ORDER BY 子句可以同时使用多个列作为排序条件。无论用哪一列作为排序条件，都要确保该列在存在。

示例：

考虑含有如下所示记录的 CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面的例子将查询结果按照 NAME 和 SALARY 升序排列：

```
SQL> SELECT * FROM CUSTOMERS
    ORDER BY NAME, SALARY;
```

结果如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
+----+----------+-----+-----------+----------+
```

下面的例子将查询结果按照 NAME 降序排列：

```
SQL> SELECT * FROM CUSTOMERS
    ORDER BY NAME DESC;
```

结果如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
+----+----------+-----+-----------+----------+
```

GROUP BY 子句

SQL GROUP BY 子句与 SELECT 语句结合在一起使用，可以将相同数据分成一组。

在 SELECT 语句中，GROUP BY 子句紧随 WHERE 子句，在 ORDER BY 子句之前。

语法：

GROUP BY 子句的基本语法如下所示。GROUP BY 子句必须在 WHERE 子句的条件之后，ORDER BY 子句（如果有的话）之前。

```
SELECT column1, column2
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2
ORDER BY column1, column2
```

示例：

考虑含有如下所示记录的 CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     |  1500.00 |
| 3 | kaushik  | 23 | Kota      |  2000.00 |
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   | 27 | Bhopal    |  8500.00 |
| 6 | Komal    | 22 | MP        |  4500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

如果你想要知道每个客户的薪水如何，可以写一个带有 GROUP BY 子句的查询：

```
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
    GROUP BY NAME;
```

结果如下所示：

```
+----------+-------------+
| NAME     | SUM(SALARY) |
+----------+-------------+
| Chaitali |     6500.00 |
| Hardik   |     8500.00 |
| kaushik  |     2000.00 |
| Khilan   |     1500.00 |
| Komal    |     4500.00 |
| Muffy    |    10000.00 |
| Ramesh   |     2000.00 |
+----------+-------------+
```

现在，让我们换一张 CUSTOMERS 表，表中记录的 NAME 字段有重复值：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Ramesh   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | kaushik  |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

同样，如果你想要知道每个客户的薪水如何的话，可以写一个带有 GROUP BY 子句的查询：

```
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
   GROUP BY NAME;
```

结果如下所示：

```
+---------+-------------+
| NAME    | SUM(SALARY) |
+---------+-------------+
| Hardik  |     8500.00 |
| kaushik |     8500.00 |
| Komal   |     4500.00 |
| Muffy   |    10000.00 |
| Ramesh  |     3500.00 |
+---------+-------------+
```

DISTINCT 关键字

SQL DISTINCT 关键字同 SELECT 语句一起使用，可以去除所有重复记录，只返回唯一项。

有时候，数据表中可能会有重复的记录。在检索这些记录的时候，应该只取回唯一的记录，而不是重复的。

语法：

使用 DISTINCT 关键字去除查询结果中的重复记录的基本语法如下所示：

```
SELECT DISTINCT column1, column2,.....columnN
FROM table_name
WHERE [condition]
```

示例：

考虑含有如下记录的 CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

首先，让我们看一下下面的 SELECT 语句是如何返回重复记录的：

```
SQL> SELECT SALARY FROM CUSTOMERS
   ORDER BY SALARY;
```

上述语句的运行结果如下所示，2000 的薪水出现了两次，表明原表中存在（SALARY 字段）重复记录。

```
+----------+
| SALARY   |
+----------+
```

```
| 1500.00 |
| 2000.00 |
| 2000.00 |
| 4500.00 |
| 6500.00 |
| 8500.00 |
|10000.00 |
+----------+
```

现在，我们在 SELECT 语句中使用 DISTINCT 关键字，然后看有什么样的结果：

```
SQL> SELECT DISTINCT SALARY FROM CUSTOMERS
   ORDER BY SALARY;
```

这一次结果中就没有重复的条目了：

```
+----------+
| SALARY   |
+----------+
| 1500.00 |
| 2000.00 |
| 4500.00 |
| 6500.00 |
| 8500.00 |
|10000.00 |
+----------+
```

对结果进行排序

SQL ORDER BY 子句根据一列或者多列的值，按照升序或者降序排列数据。某些数据库默认以升序排列查询结果。

语法：

用于将结果按照升序或者降序排列的 ORDER BY 子句的基本语法如下所示：

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

ORDER BY 子句可以同时使用多个列作为排序条件。无论用哪一列作为排序条件，都要确保该列在存在。

示例：

考虑含有如下所示记录的 CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面的例子将查询结果按照 NAME 和 SALARY 升序排列：

```
SQL> SELECT * FROM CUSTOMERS
    ORDER BY NAME, SALARY;
```

结果如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
+----+----------+-----+-----------+----------+
```

下面的例子将查询结果按照 NAME 降序排列：

```
SQL> SELECT * FROM CUSTOMERS
    ORDER BY NAME DESC;
```

结果如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
+----+----------+-----+-----------+----------+
```

以自定义的方式排序查询结果的 SELECT 语句如下所示：

```
SQL> SELECT * FROM CUSTOMERS
   ORDER BY (CASE ADDRESS
   WHEN 'DELHI'     THEN 1
   WHEN 'BHOPAL'    THEN 2
   WHEN 'KOTA'      THEN 3
   WHEN 'AHMADABAD' THEN 4
   WHEN 'MP'    THEN 5
   ELSE 100 END) ASC, ADDRESS DESC;
```

结果如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
```

```
+----+----------+-----+-----------+----------+
| 2 | Khilan   | 25 | Delhi     | 1500.00 |
| 5 | Hardik   | 27 | Bhopal    | 8500.00 |
| 3 | kaushik  | 23 | Kota      | 2000.00 |
| 6 | Komal    | 22 | MP        | 4500.00 |
| 4 | Chaitali | 25 | Mumbai    | 6500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
| 1 | Ramesh   | 32 | Ahmedabad | 2000.00 |
+----+----------+-----+-----------+----------+
```

这样查询结果就会先以你所定义的顺序按照 ADDRESS 排列。对于其余的 ADDRESS 值，查询结果仍然按照自然方式排列。亦即，除列出的 ADDERSS 之外，其余的 ADDRESS 按照字母表的逆序排列。

SQL 进阶

约束

约束是作用于数据表中列上的规则，用于限制表中数据的类型。约束的存在保证了数据库中数据的精确性和可靠性。

约束有列级和表级之分，列级约束作用于单一的列，而表级约束作用于整张数据表。

下面是 SQL 中常用的约束，这些约束虽然已经在关系型数据库管理系统一章中讨论过了，但是仍然值得在这里回顾一遍。

- NOT NULL 约束：保证列中数据不能有 NULL 值
- DEFAULT 约束：提供该列数据未指定时所采用的默认值
- UNIQUE 约束：保证列中的所有数据各不相同
- 主键约束：唯一标识数据表中的行/记录
- 外键约束：唯一标识其他表中的一条行/记录
- CHECK 约束：此约束保证列中的所有值满足某一条件
- 索引：用于在数据库中快速创建或检索数据

约束可以在使用 CREATE TABLE 创建表的时候指定，也可以在表创建之后由 ALTER TABLE 设置。

删除约束

任何现有约束都可以通过在 ALTER TABLE 命令中指定 DROP CONSTRAINT 选项的方法删除掉。

例如，要去除 EMPLOYEES 表中的主键约束，可以使用下述命令：

```
ALTER TABLE EMPLOYEES DROP CONSTRAINT EMPLOYEES_PK;
```

一些数据库实现可能提供了删除特定约束的快捷方法。例如，要在 Oracle 中删除一张表的主键约束，可以使用如下命令：

```
ALTER TABLE EMPLOYEES DROP PRIMARY KEY;
```

某些数据库实现允许禁用约束。这样与其从数据库中永久删除约束，你可以只是临时禁用掉它，过一段时间后再重新启用。

完整性约束

完整性约束用于保证关系型数据库中数据的精确性和一致性。对于关系型数据库来说，数据完整性由参照完整性（referential integrity，RI）来保证。

有很多种约束可以起到参照完整性的作用，这些约束包括主键约束（Primary Key）、外键约束（Foreign Key）、唯一性约束（Unique Constraint）以及上面提到的其他约束。

使用连接

SQL **连接（JOIN）**子句用于将数据库中两个或者两个以上表中的记录组合起来。连接通过共有值将不同表中的字段组合在一起。

考虑下面两个表，（a）CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

（b）另一个表是 ORDERS 表：

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

现在，让我们用 SELECT 语句将这个两张表连接（JOIN）在一起：

```
SQL> SELECT ID, NAME, AGE, AMOUNT
     FROM CUSTOMERS, ORDERS
     WHERE  CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

上述语句的运行结果如下所示：

```
+----+----------+-----+--------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+--------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
```

```
| 2 | Khilan  | 25 |  1560 |
| 4 | Chaitali | 25 |  2060 |
+----+----------+-----+---------+
```

SQL 连接类型

SQL 中有多种不同的连接：

- 内连接（INNER JOIN）：当两个表中都存在匹配时，才返回行。
- 左连接（LEFT JOIN）：返回左表中的所有行，即使右表中没有匹配的行。
- 右连接（RIGHT JOIN）：返回右表中的所有行，即使左表中没有匹配的行。
- 全连接（FULL JOIN）：只要某一个表存在匹配，就返回行。
- 笛卡尔连接（CARTESIAN JOIN）：返回两个或者更多的表中记录集的笛卡尔积。

内连接

最常用也最重要的连接形式是**内连接**，有时候也被称作“EQUIJOIN”（等值连接）。

内连接根据连接谓词来组合两个表中的字段，以创建一个新的结果表。SQL 查询会比较逐个比较表 1 和表 2 中的每一条记录，来寻找满足连接谓词的所有记录对。当连接谓词得以满足时，所有满足条件的记录对的字段将会结合在一起构成结果表。

语法：

内连接的基本语法如下所示：

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_field = table2.common_field;
```

示例：

考虑如下两个表格，（a）CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh  | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan  | 25 | Delhi     |  1500.00 |
| 3 | kaushik  | 23 | Kota      |  2000.00 |
| 4 | Chaitali | 25 | Mumbai    |  6500.00 |
| 5 | Hardik  | 27 | Bhopal    |  8500.00 |
```

```
| 6 | Komal    | 22 | MP        |  4500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

（b）ORDERS 表：

```
+-----+---------------------+-------------+--------+
| OID | DATE                |         ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |          3 |   3000 |
| 100 | 2009-10-08 00:00:00 |          3 |   1500 |
| 101 | 2009-11-20 00:00:00 |          2 |   1560 |
| 103 | 2008-05-20 00:00:00 |          4 |   2060 |
+-----+---------------------+-------------+--------+
```

现在，让我们用内连接将这两个表连接在一起：

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   INNER JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

上述语句将会产生如下结果：

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

左连接

左链接返回左表中的所有记录，即是右表中没有任何满足匹配条件的记录。这意味着，如果 ON 子句在右表中匹配到了 0 条记录，该连接仍然会返回至少一条记录，不过返回的记录中所有来自右表的字段都为 NULL。

这就意味着，左连接会返回左表中的所有记录，加上右表中匹配到的记录，或者是 NULL（如果连接谓词无法匹配到任何记录的话）。

语法：

左连接的基本语法如下所示：

```
SELECT table1.column1, table2.column2...
FROM table1
LEFT JOIN table2
ON table1.common_field = table2.common_field;
```

这里，给出的条件可以是任何根据你的需要写出的条件。

示例：

考虑如下两个表格，（a）CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

（b）ORDERS 表：

```
+-----+---------------------+-------------+--------+
| OID | DATE                |         ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |          3 |   3000 |
| 100 | 2009-10-08 00:00:00 |          3 |   1500 |
| 101 | 2009-11-20 00:00:00 |          2 |   1560 |
| 103 | 2008-05-20 00:00:00 |          4 |   2060 |
+-----+---------------------+-------------+--------+
```

现在，让我们用左连接将这两个表连接在一起：

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
    FROM CUSTOMERS
    LEFT JOIN ORDERS
    ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

上述语句将会产生如下结果：

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   NULL | NULL                |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
```

```
| 3 | kaushik |  3000 | 2009-10-08 00:00:00 |
| 3 | kaushik |  1500 | 2009-10-08 00:00:00 |
| 4 | Chaitali |  2060 | 2008-05-20 00:00:00 |
| 5 | Hardik  |  NULL | NULL                |
| 6 | Komal   |  NULL | NULL                |
| 7 | Muffy   |  NULL | NULL                |
+----+----------+--------+---------------------+
```

左连接

右链接返回左表中的所有记录，即是左表中没有任何满足匹配条件的记录。这意味着，如果 ON 子句在左表中匹配到了 0 条记录，该连接仍然会返回至少一条记录，不过返回的记录中所有来自左表的字段都为 NULL。

这就意味着，右连接会返回右表中的所有记录，加上左表中匹配到的记录，或者是 NULL（如果连接谓词无法匹配到任何记录的话）。

语法：

右连接的基本语法如下所示：

```
SELECT table1.column1, table2.column2...
FROM table1
RIGHT JOIN table2
ON table1.common_field = table2.common_field;
```

这里，给出的条件可以是任何根据你的需要写出的条件。

示例：

考虑如下两个表格，（a）CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     | 1500.00 |
| 3 | kaushik  | 23 | Kota      | 2000.00 |
| 4 | Chaitali | 25 | Mumbai    | 6500.00 |
| 5 | Hardik   | 27 | Bhopal    | 8500.00 |
| 6 | Komal    | 22 | MP        | 4500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

（b）ORDERS 表：

```
+-----+---------------------+-------------+--------+
| OID | DATE                |         ID | AMOUNT |
```

```
+-----+---------------------+------------+--------+
| 102 | 2009-10-08 00:00:00 |          3 |   3000 |
| 100 | 2009-10-08 00:00:00 |          3 |   1500 |
| 101 | 2009-11-20 00:00:00 |          2 |   1560 |
| 103 | 2008-05-20 00:00:00 |          4 |   2060 |
+-----+---------------------+------------+--------+
```

现在，让我们用右连接将这两个表连接在一起：

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   RIGHT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

上述语句将会产生如下结果：

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

全连接

全连接将左连接和右连接的结果组合在一起。

语法：

全连接的基本语法如下所受：

```
SELECT table1.column1, table2.column2...
FROM table1
FULL JOIN table2
ON table1.common_field = table2.common_field;
```

这里，给出的条件可以是任何根据你的需要写出的条件。

示例：

考虑如下两个表格，（a）CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
```

```
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     | 1500.00 |
| 3 | kaushik  | 23 | Kota      | 2000.00 |
| 4 | Chaitali |  25 | Mumbai    | 6500.00 |
| 5 | Hardik   | 27 | Bhopal    | 8500.00 |
| 6 | Komal    | 22 | MP        | 4500.00 |
| 7 | Muffy    | 24 | Indore    |10000.00 |
+----+----------+-----+-----------+----------+
```

（b）ORDERS 表：

```
+-----+---------------------+-------------+--------+
| OID | DATE                |         ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |          3 |   3000 |
| 100 | 2009-10-08 00:00:00 |          3 |   1500 |
| 101 | 2009-11-20 00:00:00 |          2 |   1560 |
| 103 | 2008-05-20 00:00:00 |          4 |   2060 |
+-----+---------------------+-------------+--------+
```

现在让我们用全连接将两个表连接在一起：

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   FULL JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

上述语句将会产生如下结果：

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    1 | Ramesh   |   NULL | NULL                |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|    5 | Hardik   |   NULL | NULL                |
|    6 | Komal    |   NULL | NULL                |
|    7 | Muffy    |   NULL | NULL                |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

如果你所用的数据库不支持全连接，比如 MySQL，那么你可以使用 UNION ALL子句来将左连接和右连接结果组合在一起：

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
    FROM CUSTOMERS
    LEFT JOIN ORDERS
    ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION ALL
    SELECT  ID, NAME, AMOUNT, DATE
    FROM CUSTOMERS
    RIGHT JOIN ORDERS
    ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
```

笛卡尔连接（交叉连接）

笛卡尔连接或者**交叉连接**返回两个或者更多的连接表中记录的笛卡尔乘积。也就是说，它相当于连接谓词总是为真或者缺少连接谓词的内连接。

语法：

笛卡尔连接或者说**交叉连接**的基本语法如下所示：

```
SELECT table1.column1, table2.column2...
FROM  table1, table2 [, table3 ]
```

示例：

考虑如下两个表格，（a）CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     | 1500.00 |
| 3 | kaushik  | 23 | Kota      | 2000.00 |
| 4 | Chaitali |  25 | Mumbai    | 6500.00 |
| 5 | Hardik   | 27 | Bhopal    | 8500.00 |
| 6 | Komal    | 22 | MP        | 4500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

（b）ORDERS 表：

```
+-----+---------------------+-------------+--------+
| OID | DATE                |         ID | AMOUNT |
+-----+---------------------+-------------+--------+
```

```
   | 102 | 2009-10-08 00:00:00 |          3 |   3000 |
   | 100 | 2009-10-08 00:00:00 |          3 |   1500 |
   | 101 | 2009-11-20 00:00:00 |          2 |   1560 |
   | 103 | 2008-05-20 00:00:00 |          4 |   2060 |
   +-----+---------------------+------------+--------+
```

现在，让我用内连接将这两个表连接在一起：

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
    FROM CUSTOMERS, ORDERS;
```

上述语句将会产生如下结果：

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
| 1 | Ramesh   |   3000 | 2009-10-08 00:00:00 |
| 1 | Ramesh   |   1500 | 2009-10-08 00:00:00 |
| 1 | Ramesh   |   1560 | 2009-11-20 00:00:00 |
| 1 | Ramesh   |   2060 | 2008-05-20 00:00:00 |
| 2 | Khilan   |   3000 | 2009-10-08 00:00:00 |
| 2 | Khilan   |   1500 | 2009-10-08 00:00:00 |
| 2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
| 2 | Khilan   |   2060 | 2008-05-20 00:00:00 |
| 3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
| 3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
| 3 | kaushik  |   1560 | 2009-11-20 00:00:00 |
| 3 | kaushik  |   2060 | 2008-05-20 00:00:00 |
| 4 | Chaitali |   3000 | 2009-10-08 00:00:00 |
| 4 | Chaitali |   1500 | 2009-10-08 00:00:00 |
| 4 | Chaitali |   1560 | 2009-11-20 00:00:00 |
| 4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
| 5 | Hardik   |   3000 | 2009-10-08 00:00:00 |
| 5 | Hardik   |   1500 | 2009-10-08 00:00:00 |
| 5 | Hardik   |   1560 | 2009-11-20 00:00:00 |
| 5 | Hardik   |   2060 | 2008-05-20 00:00:00 |
| 6 | Komal    |   3000 | 2009-10-08 00:00:00 |
| 6 | Komal    |   1500 | 2009-10-08 00:00:00 |
| 6 | Komal    |   1560 | 2009-11-20 00:00:00 |
| 6 | Komal    |   2060 | 2008-05-20 00:00:00 |
| 7 | Muffy    |   3000 | 2009-10-08 00:00:00 |
| 7 | Muffy    |   1500 | 2009-10-08 00:00:00 |
| 7 | Muffy    |   1560 | 2009-11-20 00:00:00 |
| 7 | Muffy    |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

UNION 子句

SQL UNION 子句/运算符用于将两个或者更多的 SELECT 语句的运算结果组合起来。

在使用 UNION 的时候，每个 SELECT 语句必须有相同数量的选中列、相同数量的列表达式、相同的数据类型，并且它们出现的次序要一致，不过长度不一定要相同。

语法

UNION 子句的基本语法如下所示：

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

UNION

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

这里的条件可以是任何根据你的需要而设的条件。

示例

考虑如下两张表，（a）CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

（b）另一张表是 ORDERS 表，如下所示：

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

现在，让我们用 SELECT 语句将这两张表连接起来：

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   LEFT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION
   SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   RIGHT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

结果如下所示：

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    1 | Ramesh   |   NULL | NULL                |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|    5 | Hardik   |   NULL | NULL                |
|    6 | Komal    |   NULL | NULL                |
|    7 | Muffy    |   NULL | NULL                |
+------+----------+--------+---------------------+
```

UNION ALL 子句：

UNION ALL 运算符用于将两个 SELECT 语句的结果组合在一起，重复行也包含在内。

UNION ALL 运算符所遵从的规则与 UNION 一致。

语法：

UNION ALL的基本语法如下：

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

UNION ALL

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

示例：

考虑如下两张表，（a）CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

（b）另一张表是 ORDERS 表，如下所示：

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

结果如下所示：

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    1 | Ramesh   |   NULL | NULL                |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|    5 | Hardik   |   NULL | NULL                |
|    6 | Komal    |   NULL | NULL                |
|    7 | Muffy    |   NULL | NULL                |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

另外，还有两个子句（亦即运算法）与 UNION 子句非常相像：

- SQL INTERSECT 子句：用于组合两个 SELECT 语句，但是只返回两个 SELECT 语句的结果中都有的行。
- SQL EXCEPT 子句：组合两个 SELECT 语句，并将第一个 SELECT 语句的结果中存在，但是第二个 SELECT 语句的结果中不存在的行返回。

NULL 值

SQL 中，NULL 用于表示缺失的值。数据表中的 NULL 值表示该值所处的字段为空。

值为 NULL 的字段没有值。尤其要明白的是，NULL 值与 0 或者包含空白（spaces）的字段是不同的。

语法：

创建表的时候，NULL 的基本语法如下：

```
SQL> CREATE TABLE CUSTOMERS(
   ID   INT              NOT NULL,
   NAME VARCHAR (20)     NOT NULL,
   AGE  INT              NOT NULL,
   ADDRESS  CHAR (25) ,
   SALARY   DECIMAL (18, 2),
   PRIMARY KEY (ID)
);
```

这里，NOT NULL表示对于给定列，必须按照其数据类型明确赋值。有两列并没有使用 NOT NULL 来限定，也就是说这些列可以为 NULL。

值为 NULL 的字段是在记录创建的过程中留空的字段。

示例：

NULL 值会给选取数据带来麻烦。不过，因为 NULL 和其他任何值作比较，其结果总是未知的，所以含有 NULL 的记录不会包含在最终结果里面。

必须使用 IS NULL 或者 IS NOT NULL 来检测某个字段是否为 NULL。

考虑下面的 CUSTOMERS 数据表，里面包含的记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
```

```
| 5 | Hardik   | 27 | Bhopal    | 8500.00 |
| 6 | Komal    | 22 | MP        |         |
| 7 | Muffy    | 24 | Indore    |         |
+----+----------+-----+-----------+----------+
```

下面是 IS NOT NULL 运算符的用法：

```
SQL> SELECT  ID, NAME, AGE, ADDRESS, SALARY
   FROM CUSTOMERS
   WHERE SALARY IS NOT NULL;
```

上面语句的运行结果如下：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     | 1500.00 |
| 3 | kaushik  | 23 | Kota      | 2000.00 |
| 4 | Chaitali |  25 | Mumbai    | 6500.00 |
| 5 | Hardik   | 27 | Bhopal    | 8500.00 |
+----+----------+-----+-----------+----------+
```

下面是 IS NULL 运算符的用法：

```
SQL> SELECT  ID, NAME, AGE, ADDRESS, SALARY
   FROM CUSTOMERS
   WHERE SALARY IS NULL;
```

其运行结果如下：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 6 | Komal    | 22 | MP        |         |
| 7 | Muffy    | 24 | Indore    |         |
+----+----------+-----+-----------+----------+
```

别名

我们可以使用**别名（Alias）**来对数据表或者列进行临时命名。

使用别名意味着要用特定的 SQL 语句对表进行重命名。重命名是临时的，数据库中表的实际名字并不会改变。

对于特定的 SQL 查询，需要使用列别名来对表中的列进行重命名。

语法：

表别名的基本语法如下：

```
SELECT column1, column2....
FROM table_name AS alias_name
WHERE [condition];
```

列别名的基本语法如下：

```
SELECT column_name AS alias_name
FROM table_name
WHERE [condition];
```

示例：

考虑下面两个数据表，（a）CUSTOMERS 表，如下：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     | 1500.00 |
| 3 | kaushik  | 23 | Kota      | 2000.00 |
| 4 | Chaitali | 25 | Mumbai    | 6500.00 |
| 5 | Hardik   | 27 | Bhopal    | 8500.00 |
| 6 | Komal    | 22 | MP        | 4500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

（b）另一个是 ORDERS 表，如下所示：

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

下面是**表别名**的用法：

```
SQL> SELECT C.ID, C.NAME, C.AGE, O.AMOUNT
     FROM CUSTOMERS AS C, ORDERS AS O
     WHERE  C.ID = O.CUSTOMER_ID;
```

上面语句的运行结果如下所示：

```
+----+----------+-----+--------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+--------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+----+----------+-----+--------+
```

下面是**列别名**的用法：

```
SQL> SELECT  ID AS CUSTOMER_ID, NAME AS CUSTOMER_NAME
   FROM CUSTOMERS
   WHERE SALARY IS NOT NULL;
```

其运行结果如下所示：

```
+-------------+---------------+
| CUSTOMER_ID | CUSTOMER_NAME |
+-------------+---------------+
|           1 | Ramesh        |
|           2 | Khilan        |
|           3 | kaushik       |
|           4 | Chaitali      |
|           5 | Hardik        |
|           6 | Komal         |
|           7 | Muffy         |
+-------------+---------------+
```

索引

索引是一种特殊的查询表，可以被数据库搜索引擎用来加速数据的检索。简单说来，索引就是指向表中数据的指针。数据库的索引同书籍后面的索引非常相像。

例如，如果想要查阅一本书中与某个特定主题相关的所有页面，你会先去查询索引（索引按照字母表顺序列出了所有主题），然后从索引中找到一页或者多页与该主题相关的页面。

索引能够提高 SELECT 查询和 WHERE 子句的速度，但是却降低了包含 UPDATE 语句或 INSERT 语句的数据输入过程的速度。索引的创建与删除不会对表中的数据产生影响。

创建索引需要使用 CREATE INDEX 语句，该语句允许对索引命名，指定要创建索引的表以及对哪些列进行索引，还可以指定索引按照升序或者降序排列。

同 UNIQUE 约束一样，索引可以是唯一的。这种情况下，索引会阻止列中（或者列的组合，其中某些列有索引）出现重复的条目。

CREATE INDEX 命令：

CREATE INDEX命令的基本语法如下：

```
CREATE INDEX index_name ON table_name;
```

单列索引：

单列索引基于单一的字段创建，其基本语法如下所示：

```
CREATE INDEX index_name
ON table_name (column_name);
```

唯一索引：

唯一索引不止用于提升查询性能，还用于保证数据完整性。唯一索引不允许向表中插入任何重复值。其基本语法如下所示：

```
CREATE UNIQUE INDEX index_name
on table_name (column_name);
```

聚簇索引：

聚簇索引在表中两个或更多的列的基础上建立。其基本语法如下所示：

```
CREATE INDEX index_name
on table_name (column1, column2);
```

创建单列索引还是聚簇索引，要看每次查询中，哪些列在作为过滤条件的 WHERE 子句中最常出现。

如果只需要一列，那么就应当创建单列索引。如果作为过滤条件的 WHERE 子句用到了两个或者更多的列，那么聚簇索引就是最好的选择。

隐式索引：

隐式索引由数据库服务器在创建某些对象的时候自动生成。例如，对于主键约束和唯一约束，数据库服务器就会自动创建索引。

DROP INDEX 命令：

索引可以用 SQL DROP 命令删除。删除索引时应当特别小心，数据库的性能可能会因此而降低或者提高。

其基本语法如下：

```
DROP INDEX index_name;
```

什么时候应当避免使用索引?

尽管创建索引的目的是提升数据库的性能，但是还是有一些情况应当避免使用索引。下面几条指导原则给出了何时应当重新考虑是否使用索引：

- 小的数据表不应当使用索引；
- 需要频繁进行大批量的更新或者插入操作的表；
- 如果列中包含大数或者 NULL 值，不宜创建索引；
- 频繁操作的列不宜创建索引。

ALTER TABLE 命令

SQL ALTER TABLE 命令用于添加、删除或者更改现有数据表中的列。

你还可以用 ALTER TABLE 命令来添加或者删除现有数据表上的约束。

语法：

使用 ALTER TABLE 在现有的数据表中**添加新列**的基本语法如下：

```
ALTER TABLE table_name ADD column_name datatype;
```

使用 ALTER TABLE 在现有的数据表中**删除列**的基本语法如下：

```
ALTER TABLE table_name DROP COLUMN column_name;
```

使用 ALTER TABLE 更改现有的数据表中**列的数据类型**的基本语法如下：

```
ALTER TABLE table_name MODIFY COLUMN column_name datatype;
```

使用 ALTER TABLE 给某列添加 **NOT NULL 约束** 的基本语法如下：

```
ALTER TABLE table_name MODIFY column_name datatype NOT NULL;
```

使用 ALTER TABLE 给数据表添加 **唯一约束** 的基本语法如下：

```
ALTER TABLE table_name
ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...);
```

使用 ALTER TABLE 给数据表添加 **CHECK 约束** 的基本语法如下：

```
ALTER TABLE table_name
ADD CONSTRAINT MyUniqueConstraint CHECK (CONDITION);
```

使用 ALTER TABLE 给数据表添加 **主键约束** 的基本语法如下：

```
ALTER TABLE table_name
ADD CONSTRAINT MyPrimaryKey PRIMARY KEY (column1, column2...);
```

使用 ALTER TABLE 从数据表中 **删除约束** 的基本语法如下：

```
ALTER TABLE table_name
DROP CONSTRAINT MyUniqueConstraint;
```

如果你在使用 MySQL，代码应当如下：

```
ALTER TABLE table_name
DROP INDEX MyUniqueConstraint;
```

使用 ALTER TABLE 从数据表中 **删除主键约束** 的基本语法如下：

```
ALTER TABLE table_name
DROP CONSTRAINT MyPrimaryKey;
```

如果你在使用 MySQL，代码应当如下：

```
ALTER TABLE table_name
DROP PRIMARY KEY;
```

示例：

考虑 CUSTOMERS 表，表中记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面的示例展示了如何在现有的表中添加新的一列：

```
ALTER TABLE CUSTOMERS ADD SEX char(1);
```

现在，CUSTOMERS 已经被更改了，SELECT 语句的输出应当如下所示：

```
+----+---------+-----+-----------+----------+------+
| ID | NAME    | AGE | ADDRESS   | SALARY   | SEX  |
+----+---------+-----+-----------+----------+------+
|  1 | Ramesh  |  32 | Ahmedabad |  2000.00 | NULL |
|  2 | Ramesh  |  25 | Delhi     |  1500.00 | NULL |
|  3 | kaushik |  23 | Kota      |  2000.00 | NULL |
|  4 | kaushik |  25 | Mumbai    |  6500.00 | NULL |
```

```
| 5 | Hardik  | 27 | Bhopal    | 8500.00 | NULL |
| 6 | Komal   | 22 | MP        | 4500.00 | NULL |
| 7 | Muffy   | 24 | Indore    | 10000.00 | NULL |
+----+---------+-----+-----------+----------+------+
```

下面的示例展示了如何从 CUSTOMERS 表中删除 SEX 列：

```
ALTER TABLE CUSTOMERS DROP SEX;
```

现在，CUSTOMERS 已经被更改了，SELECT 语句的输出应当如下所示：

```
+----+---------+-----+-----------+----------+
| ID | NAME    | AGE | ADDRESS   | SALARY   |
+----+---------+-----+-----------+----------+
| 1 | Ramesh  | 32 | Ahmedabad |  2000.00 |
| 2 | Ramesh  | 25 | Delhi     | 1500.00 |
| 3 | kaushik |  23 | Kota      | 2000.00 |
| 4 | kaushik |  25 | Mumbai    | 6500.00 |
| 5 | Hardik  | 27 | Bhopal    | 8500.00 |
| 6 | Komal   | 22 | MP        | 4500.00 |
| 7 | Muffy   | 24 | Indore    | 10000.00 |
+----+---------+-----+-----------+----------+
```

TRUNCATE TABLE 命令

SQL TRUNCATE TABLE 命令用于删除现有数据表中的所有数据。

你也可以使用 DROP TABLE 命令来删除整个数据表，不过 DROP TABLE 命令不但会删除表中所有数据，还会将整个表结构从数据库中移除。如果想要重新向表中存储数据的话，必须重建该数据表。

语法：

TRUNCATE TABLE 的基本语法如下所示：

```
TRUNCATE TABLE  table_name;
```

示例：

考虑 CUSTOMERS 表，表中的记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面的示例展示了 TRUNCATE 命令的用法：

```
TRUNCATE TABLE CUSTOMERS;
```

现在，CUSTOMERS 表已经被清空了，SELECT 语句的输出应当如下所示：

```
SQL> SELECT * FROM CUSTOMERS;
Empty set (0.00 sec)
```

使用视图

视图无非就是存储在数据库中并具有名字的 SQL 语句，或者说是以预定义的 SQL 查询的形式存在的数据表的成分。

视图可以包含表中的所有列，或者仅包含选定的列。视图可以创建自一个或者多个表，这取决于创建该视图的 SQL 语句的写法。

视图，一种虚拟的表，允许用户执行以下操作：

- 以用户或者某些类型的用户感觉自然或者直观的方式来组织数据；
- 限制对数据的访问，从而使得用户仅能够看到或者修改（某些情况下）他们需要的数据；
- 从多个表中汇总数据，以产生报表。

创建视图：

数据库视图由 CREATE VIEW 语句创建。视图可以创建自单个表、多个表或者其他视图。

要创建视图的话，用户必须有适当的系统权限。具体需要何种权限随数据库系统实现的不同而不同。

CREATE VIEW 语句的基本语法如下所示：

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

和普通的 SQL SELECT 查询一样，你可以在上面的 SELECT 语句中包含多个数据表。

示例：

考虑 CUSTOMERS 表，表中的记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
```

```
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   |  27 | Bhopal    |  8500.00 |
| 6 | Komal    |  22 | MP        |  4500.00 |
| 7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面是由 CUSTOMERS 表创建视图的例子。该视图包含来自 CUSTOMERS 表的顾客的名字（name）和年龄（age）：

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS;
```

现在，你就可以像查询普通的数据表一样查询 CUSTOMERS_VIEW 了：

```
SQL > SELECT * FROM CUSTOMERS_VIEW;
```

上述语句将会产生如下运行结果：

```
+----------+-----+
| name     | age |
+----------+-----+
| Ramesh   |  32 |
| Khilan   |  25 |
| kaushik  |  23 |
| Chaitali |  25 |
| Hardik   |  27 |
| Komal    |  22 |
| Muffy    |  24 |
+----------+-----+
```

WITH CHECK OPTION

WITH CHECK OPTION 是 CREATE VIEW 语句的一个可选项。WITH CHECK OPTION 用于保证所有的 UPDATE 和 INSERT 语句都满足视图定义中的条件。

如果不能满足这些条件，UPDATE 或 INSERT 就会返回错误。

下面的例子创建的也是 CUSTOMERS_VIEW 视图，不过这次 WITH CHECK OPTION 是打开的：

```
CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS
```

```
WHERE age IS NOT NULL
WITH CHECK OPTION;
```

这里 WITH CHECK OPTION 使得视图拒绝任何 AGE 字段为 NULL 的条目，因为视图的定义中，AGE 字段不能为空。

更新视图：

视图可以在特定的情况下更新：

- ？ SELECT 子句不能包含 DISTINCT 关键字
- ？ SELECT 子句不能包含任何汇总函数（summary functions）
- ？ SELECT 子句不能包含任何集合函数（set functions）
- ？ SELECT 子句不能包含任何集合运算符（set operators）
- ？ SELECT 子句不能包含 ORDER BY 子句
- ？ FROM 子句中不能有多个数据表
- ？ WHERE 子句不能包含子查询（subquery）
- ？ 查询语句中不能有 GROUP BY 或者 HAVING
- ？ 计算得出的列不能更新
- ？ 视图必须包含原始数据表中所有的 NOT NULL 列，从而使 INSERT 查询生效。

如果视图满足以上所有的条件，该视图就可以被更新。下面的例子中，Ramesh 的年龄被更新了：

```
SQL > UPDATE CUSTOMERS_VIEW
    SET AGE = 35
    WHERE name='Ramesh';
```

最终更新的还是原始数据表，只是其结果反应在了视图上。现在查询原始数据表，SELECT 语句将会产生以下结果：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 35 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     |  1500.00 |
| 3 | kaushik  | 23 | Kota      |  2000.00 |
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   | 27 | Bhopal    |  8500.00 |
| 6 | Komal    | 22 | MP        |  4500.00 |
```

```
| 7 | Muffy   | 24 | Indore   | 10000.00 |
+----+----------+-----+-----------+----------+
```

向视图中插入新行：

可以向视图中插入新行，其规则同（使用 UPDATE 命令）更新视图所遵循的规则相同。

这里我们不能向 CUSTOMERS_VIEW 视图中添加新行，因为该视图没有包含原始数据表中所有 NOT NULL 的列。否则的话，你就可以像在数据表中插入新行一样，向视图中插入新行。

删除视图中的行：

视图中的数据行可以被删除。删除数据行与更新视图和向视图中插入新行遵循相同的规则。

下面的例子将删除 CUSTOMERS_VIEW 视图中 AGE=22 的数据行：

```
SQL > DELETE FROM CUSTOMERS_VIEW
    WHERE age = 22;
```

该语句最终会将原始数据表中对应的数据行删除，只不过其结果反应在了视图上。现在查询原始数据表，SELECT 语句将会产生以下结果：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 35 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     | 1500.00 |
| 3 | kaushik  | 23 | Kota      | 2000.00 |
| 4 | Chaitali |  25 | Mumbai    | 6500.00 |
| 5 | Hardik   | 27 | Bhopal    | 8500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

删除视图：

很明显，当我们不再需要某个视图的时候，需要有一种方式可以让我们将其删除。删除视图的语法非常简单，如下所示：

```
DROP VIEW view_name;
```

下面的例子展示了如何从 CUSTOMERS 表中删除 CUSTOMERS_VIEW 视图：

```
DROP VIEW CUSTOMERS_VIEW;
```

HAVING 子句

HAVING 子句使你能够指定过滤条件，从而控制查询结果中哪些组可以出现在最终结果里面。

WHERE 子句对被选择的列施加条件，而 HAVING 子句则对 GROUP BY 子句所产生的组施加条件。

语法：

下面可以看到 HAVING 子句在 SELECT 查询中的位置：

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

在 SELECT 查询中，HAVING 子句必须紧随 GROUP BY 子句，并出现在 ORDER BY 子句（如果有的话）之前。带有 HAVING 子句的 SELECT 语句的语法如下所示：

```
SELECT column1, column2
FROM table1, table2
WHERE [ conditions ]
GROUP BY column1, column2
HAVING [ conditions ]
ORDER BY column1, column2
```

示例：

考虑 CUSTOMERS 表，表中的记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     |  1500.00 |
| 3 | kaushik  | 23 | Kota      |  2000.00 |
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   | 27 | Bhopal    |  8500.00 |
| 6 | Komal    | 22 | MP        |  4500.00 |
```

```
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面是一个有关 HAVING 子句使用的实例，该实例将会筛选出出现次数大于或等于 2 的所有记录。

```
SQL > SELECT ID, NAME, AGE, ADDRESS, SALARY
FROM CUSTOMERS
GROUP BY age
HAVING COUNT(age) >= 2;
```

其执行结果如下所示：

```
+----+--------+-----+---------+---------+
| ID | NAME   | AGE | ADDRESS | SALARY  |
+----+--------+-----+---------+---------+
| 2 | Khilan |  25 | Delhi   | 1500.00 |
+----+--------+-----+---------+---------+
```

事务

事务是在数据库上按照一定的逻辑顺序执行的任务序列，既可以由用户手动执行，也可以由某种数据库程序自动执行。

事务实际上就是对数据库的一个或者多个更改。当你在某张表上创建更新或者删除记录的时，你就已经在使用事务了。控制事务以保证数据完整性，并对数据库错误做出处理，对数据库来说非常重要。

实践中，通常会将很多 SQL 查询组合在一起，并将其作为某个事务一部分来执行。

事务的属性：

事务具有以下四个标准属性，通常用缩略词 ACID 来表示：

- **原子性**：保证任务中的所有操作都执行完毕；否则，事务会在出现错误时终止，并回滚之前所有操作到原始状态。
- **一致性**：如果事务成功执行，则数据库的状态得到了进行了正确的转变。
- **隔离性**：保证不同的事务相互独立、透明地执行。
- **持久性**：即使出现系统故障，之前成功执行的事务的结果也会持久存在。

事务控制：

有四个命令用于控制事务：

- COMMIT：提交更改；
- ROLLBACK：回滚更改；
- SAVEPOINT：在事务内部创建一系列可以 ROLLBACK 的还原点；
- SET TRANSACTION：命名事务；

COMMIT 命令：

COMMIT 命令用于保存事务对数据库所做的更改。

COMMIT 命令会将自上次 COMMIT 命令或者 ROLLBACK 命令执行以来所有的事务都保存到数据库中。

COMMIT 命令的语法如下所示：

```
COMMIT;
```

示例：

考虑 CUSTOMERS 表，表中的记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 2 | Khilan   | 25 | Delhi     |  1500.00 |
| 3 | kaushik  | 23 | Kota      |  2000.00 |
| 4 | Chaitali |  25 | Mumbai    |  6500.00 |
| 5 | Hardik   | 27 | Bhopal    |  8500.00 |
| 6 | Komal    | 22 | MP        |  4500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面的示例将会删除表中 age=25 的记录，然后将更改提交（COMMIT）到数据库中。

```
SQL> DELETE FROM CUSTOMERS
   WHERE AGE = 25;
SQL> COMMIT;
```

上述语句将会从表中删除两行记录，再执行 SELECT 语句将会得到如下结果：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad |  2000.00 |
| 3 | kaushik  | 23 | Kota      |  2000.00 |
| 5 | Hardik   | 27 | Bhopal    |  8500.00 |
| 6 | Komal    | 22 | MP        |  4500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

ROLLBACK 命令：

ROLLBACK 命令用于撤销尚未保存到数据库中的事务。

ROLLBACK 命令只能撤销自上次 COMMIT 命令或者 ROLLBACK 命令执行以来的事务。

ROLLBACK 命令的语法如下所示：

```
ROLLBACK;
```

示例：

考虑 CUSTOMERS 表，表中的记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面的示例将会从表中删除所有 age=25 的记录，然后回滚（ROLLBACK）对数据库所做的更改。

```
SQL> DELETE FROM CUSTOMERS
   WHERE AGE = 25;
SQL> ROLLBACK;
```

结果是删除操作并不会对数据库产生影响。现在，执行 SELECT 语句将会得到如下结果：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

SAVEPOINT 命令：

SAVEPOINT 是事务中的一个状态点，使得我们可以将事务回滚至特定的点，而不是将整个事务都撤销。

SAVEPOINT 命令的记录如下所示：

```
SAVEPOINT SAVEPOINT_NAME;
```

该命令只能在事务语句之间创建保存点（SAVEPOINT）。ROLLBACK 命令可以用于撤销一系列的事务。

回滚至某一保存点的语法如下所示：

```
ROLLBACK TO SAVEPOINT_NAME;
```

下面的示例中，你计划从 CUSTOMERS 表中删除三条不同的记录，并在每次删除之前创建一个保存点（SAVEPOINT），从而使得你可以在任何任何时候回滚到任意的保存点，以恢复数据至其原始状态。

示例：

考虑 CUSTOMERS 表，表中的记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

操作序列如下所示：

```
SQL> SAVEPOINT SP1;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=1;
1 row deleted.
SQL> SAVEPOINT SP2;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=2;
```

```
1 row deleted.
SQL> SAVEPOINT SP3;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=3;
1 row deleted.
```

现在，三次删除操作已经生效了，如果此时你改变主意决定回滚至名字为 SP2 的保存点，由于 SP2 于第一次删除操作之后创建，所以后两次删除操作将会被撤销。

```
SQL> ROLLBACK TO SP2;
Rollback complete.
```

注意，由于你将数据库回滚至 SP2，所以只有第一次删除真正起效了：

```
SQL> SELECT * FROM CUSTOMERS;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
6 rows selected.
```

RELEASE SAVEPOINT 命令：

RELEASE SAVEPOINT 命令用于删除先前创建的保存点。

RELEASE SAVEPOINT 的语法如下所示：

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

保存点一旦被释放，你就不能够再用 ROLLBACK 命令来撤销该保存点之后的事务了。

SET TRANSACTION 命令：

SET TRANSACTION 命令可以用来初始化数据库事务，指定随后的事务的各种特征。

例如，你可以将某个事务指定为只读或者读写。

SET TRANSACTION 命令的语法如下所示：

```
SET TRANSACTION [ READ WRITE | READ ONLY ];
```

通配符

我们已经讨论过 SQL 的 LIKE 操作符了，它可以利用通配符来对两个相似的值作比较。

SQL 支持以下两个通配符与 LIKE 操作符一起使用：

通配符	描述
百分号（%）	匹配一个或者多个字符。注意：MS Access 使用星号（*）作为匹配一个或者多个字符的通配符，而不是百分号（%）。
下划线（_）	匹配一个字符。注意：MS Access 使用问号（？），而不是下划线，来匹配任一字符。

百分号代表零个、一个或者多个字符。下划线代表单一的字符。这些符号可以组合在一起使用。

语法：

“%”和“_”的基本语法如下所示：

```
SELECT FROM table_name
WHERE column LIKE 'XXXX%'

or

SELECT FROM table_name
WHERE column LIKE '%XXXX%'

or

SELECT FROM table_name
WHERE column LIKE 'XXXX_'

or

SELECT FROM table_name
WHERE column LIKE '_XXXX'

or

SELECT FROM table_name
WHERE column LIKE '_XXXX_'
```

你可以用 AND 或 OR 操作符将多个条件合并在一起。这里，XXXX 可以为任何数值或者字符串。

示例：

语句	描述
WHERE SALARY LIKE '200%'	找出任何以 200 开头的值。
WHERE SALARY LIKE '%200%'	找出任何存在 200 的值。
WHERE SALARY LIKE '_00%'	找出任何第二个位置和第三个位置为 0 的值。
WHERE SALARY LIKE '2_%_%'	找出任何以 2 开始，并且长度至少为 3 的值。
WHERE SALARY LIKE '%2'	找出任何以 2 结尾的值。
WHERE SALARY LIKE '_2%3'	找出任何第二个位置为 2，并且以 3 结束的值。
WHERE SALARY LIKE '2___3'	找出任何以 2 开始，以 3 结束的五位数。

让我们来看一个真实的例子，考虑拥有如下记录的 CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面的示例将会找到 CUSTOMER 表中所有 SALARY 以 200 开头的记录，并显示出来：

```
SQL> SELECT * FROM CUSTOMERS
WHERE SALARY LIKE '200%';
```

结果如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
+----+----------+-----+-----------+----------+
```

日期函数

下面的列表中是 SQL 中所有与日期和时间相关的重要函数。你所用的 RDBMS 可能会支持更多其他的函数。下面的列表基于 MySQL 关系型数据库管理系统。

名称	描述
ADDDATE()	增加日期
ADDTIME()	增加时间
CONVERT_TZ()	将当前时区更改为另一时区
CURDATE()	返回当前日期
CURRENT_DATE(), CURRENT_DATE	CURDATE() 的别名
CURRENT_TIME(), CURRENT_TIME	CURTIME() 的别名
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP	NOW() 的别名
CURTIME()	返回当前时间
DATE_ADD()	将两个日期相加
DATE_FORMAT()	按照指定格式格式化日期
DATE_SUB()	将两个日期相减
DATE()	从 date 或者 datetime 表达式中提取出日期部分
DATEDIFF()	将两个日期相减
DAY()	DAYOFMONTH() 的别名
DAYNAME()	返回某天在用星期中的名称
DAYOFMONTH()	返回某天是当月的第几天（1-31）
DAYOFWEEK()	返回某天是该星期的第几天
DAYOFYEAR()	返回某天是一年中的第几天（1-366）
EXTRACT	提取日期中的某一部分
FROM_DAYS()	将天数转换为日期
FROM_UNIXTIME()	将某个日期格式化为 UNIX 时间戳
HOUR()	提取小时
LAST_DAY	返回参数日期所在月份的最后一天
LOCALTIME(), LOCALTIME	NOW() 的别名
LOCALTIMESTAMP, LOCALTIMESTAMP()	NOW() 的别名
MAKEDATE()	利用年份和某天在该年所处的天数来创建日期
MAKETIME	MAKETIME()

MICROSECOND()	由参数返回微秒
MINUTE()	由参数返回分钟
MONTH()	返回日期参数的月份
MONTHNAME()	返回月份的名字
NOW()	返回当前日期和时间
PERIOD_ADD()	向年月格式的日期数据之间添加一段时间
PERIOD_DIFF()	返回两个年月格式的日期数据之间的月份数
QUARTER()	返回日期参数所在的季度
SEC_TO_TIME()	将秒数转换为 'HH:MM:SS' 格式
SECOND()	返回参数中的秒数 (0-59)
STR_TO_DATE()	将字符串转换为日期数据
SUBDATE()	以三个参数调用的时候是 DATE_SUB() 的同义词
SUBTIME()	减去时间
SYSDATE()	返回函数执行的时的时刻
TIME_FORMAT()	格式化时间
TIME_TO_SEC()	将时间参数转换为秒数
TIME()	返回参数表达式中的时间部分
TIMEDIFF()	将两个时间相减
TIMESTAMP()	只有一个参数时，该函数返回 date 或者 datetime 表达式。当有两个参数时，将两个参数相加。
TIMESTAMPADD()	在 datetime 表达式上加上一段时间
TIMESTAMPDIFF()	在 datetime 表达式上减去一段时间
TO_DAYS()	将日期参数转换为天数
UNIX_TIMESTAMP()	返回 UNIX 时间戳
UTC_DATE()	返回当前 UTC 日期
UTC_TIME()	返回当前 UTC 时间
UTC_TIMESTAMP()	返回当前 UTC 日期和时间
WEEK()	返回参数的星期数
WEEKDAY()	返回日期参数时一个星期中的第几天
WEEKOFYEAR()	返回日期参数是日历上的第几周 (1-53)
YEAR()	返回日期参数中的年份
YEARWEEK()	返回年份和星期

ADDDATE(date, INTERVAL expr unit), ADDDATE(expr, days)

如果调用时第二个参数为 INTERVAL 形式的话，ADDDATE() 就是 DATE_ADD() 的同义词。同样的情况下，SUBDATE() 是 DATE_SUB() 的同义词。有关 INTERVAL 单位参数的信息，见有关 DATE_ADD() 的讨论。

```
mysql> SELECT DATE_ADD('1998-01-02', INTERVAL 31 DAY);
+------------------------------------------------------+
| DATE_ADD('1998-01-02', INTERVAL 31 DAY)              |
+------------------------------------------------------+
| 1998-02-02                                           |
+------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT ADDDATE('1998-01-02', INTERVAL 31 DAY);
+------------------------------------------------------+
| ADDDATE('1998-01-02', INTERVAL 31 DAY)               |
+------------------------------------------------------+
| 1998-02-02                                           |
+------------------------------------------------------+
1 row in set (0.00 sec)
```

如果调用时第二个参数为天数形式的话，则 MySQL 会将其作为整数加到 expr 上。

```
mysql> SELECT ADDDATE('1998-01-02', 31);
+------------------------------------------------------+
| DATE_ADD('1998-01-02', INTERVAL 31 DAY)              |
+------------------------------------------------------+
| 1998-02-02                                           |
+------------------------------------------------------+
1 row in set (0.00 sec)
```

ADDTIME(expr1,expr2)

ADDTIME() 将 expr2 加到 expr1 上，并返回结果。expr1 为 time 或者 datetime 表达式，expr2 为 time 表达式。

```
mysql> SELECT ADDTIME('1997-12-31 23:59:59.999999','1 1:1:1.000002');
+------------------------------------------------------+
| DATE_ADD('1997-12-31 23:59:59.999999','1 1:1:1.000002') |
+------------------------------------------------------+
```

```
| 1998-01-02 01:01:01.000001                          |
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

CONVERT_TZ(dt,from_tz,to_tz)

该函数将 datetime 类型的值 dt 的时区从 from_dt 转换为 to_dt，并返回结果。如果参数无效，则函数返回 NULL。

```
mysql> SELECT CONVERT_TZ('2004-01-01 12:00:00','GMT','MET');
+-----------------------------------------------------+
| CONVERT_TZ('2004-01-01 12:00:00','GMT','MET')       |
+-----------------------------------------------------+
| 2004-01-01 13:00:00                                 |
+-----------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT CONVERT_TZ('2004-01-01 12:00:00','+00:00','+10:00');
+-----------------------------------------------------+
| CONVERT_TZ('2004-01-01 12:00:00','+00:00','+10:00') |
+-----------------------------------------------------+
| 2004-01-01 22:00:00                                 |
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

CURDATE()

以 'YYYY-MM-DD'（字符串）或者 YYYYMMDD（数值）的形式返回当前日期， 具体形式取决于函数处于字符串还是数值型的上下文环境中。

```
mysql> SELECT CURDATE();
+------------------------------------------------------+
| CURDATE()                                            |
+------------------------------------------------------+
| 1997-12-15                                           |
+------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT CURDATE() + 0;
+------------------------------------------------------+
| CURDATE() + 0                                        |
+------------------------------------------------------+
```

```
| 19971215                                  |
+-------------------------------------------+
1 row in set (0.00 sec)
```

CURRENT_DATE and CURRENT_DATE()

CURRENT_DATE 和 CURRENT_DATE() 是 CURDATE() 的别名。

CURTIME()

以 'HH:MM:SS'（字符串）或者 HHMMSS（数值）的形式返回当前时间， 具体形式取决于函数处于字符串还是数值型的上下文环境中。该函数按照当前时区来表示返回值。

```
mysql> SELECT CURTIME();
+-----------+
| CURTIME() |
+-----------+
| 23:50:26  |
+-----------+
1 row in set (0.00 sec)

mysql> SELECT CURTIME() + 0;
+---------------+
| CURTIME() + 0 |
+---------------+
| 235026        |
+---------------+
1 row in set (0.00 sec)
```

CURRENT_TIME and CURRENT_TIME()

CURRENT_TIME 和 CURRENT_TIME() 是 CURTIME() 的别名。

CURRENT_TIMESTAMP and CURRENT_TIMESTAMP()

CURRENT_TIMESTAMP 和 CURRENT_TIMESTAMP() 是 NOW() 的别名。

DATE(expr)

提取 date 表达式或者 datetime 表达式中的日期部分。

```
mysql> SELECT DATE('2003-12-31 01:02:03');
+-----------------------------------------------------------+
| DATE('2003-12-31 01:02:03')                               |
+-----------------------------------------------------------+
| 2003-12-31                                                |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

DATEDIFF(expr1,expr2)

DATEDIFF() 返回 expr1 和 expr2 的差，以天数的形式表示。expr1 和 expr2 应为 date 或者 datetime 表达式。只有参数的日期部分参与了计算。

```
mysql> SELECT DATEDIFF('1997-12-31 23:59:59','1997-12-30');
+-----------------------------------------------------------+
| DATEDIFF('1997-12-31 23:59:59','1997-12-30')              |
+-----------------------------------------------------------+
| 1                                                         |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

DATE_ADD(date,INTERVAL expr unit), DATE_SUB(date,INTERVAL expr unit)

这些函数进行有关日期的算术运算。date 是一个 DATETIME 或者 DATE 类型的值，指明了起始时间。expr 表达式则是 date 要增加或者减去的时间间隔。expr 是一个字符串，可以以 '-' 开始来表示负时间区间。 unit 是一个关键词，指明了expr 的单位。

INTERVAL 关键字和 unit（单位）指示符不区分大小写。

下表列出了对于每种单位，expr 应有的形式。

unit 值	expr 应有的格式
MICROSECOND	微秒
SECOND	秒
MINUTE	分钟

HOUR	小时
DAY	天
WEEK	星期
MONTH	月
QUARTER	季度
YEAR	年
SECOND_MICROSECOND	'秒.微秒'
MINUTE_MICROSECOND	'分.微秒'
MINUTE_SECOND	'分:秒'
HOUR_MICROSECOND	'小时.微秒'
HOUR_SECOND	'时:分:秒'
HOUR_MINUTE	'时:分'
DAY_MICROSECOND	'天.微秒'
DAY_SECOND	'天 时:分:秒'
DAY_MINUTE	'天 时:分'
DAY_HOUR	'天 时'
YEAR_MONTH	'年-月'

QUARTER 和 WEEK 自 MySQL 5.0.0 起受到支持。

```
mysql> SELECT DATE_ADD('1997-12-31 23:59:59',
    -> INTERVAL '1:1' MINUTE_SECOND);
+-----------------------------------------------------------+
| DATE_ADD('1997-12-31 23:59:59', INTERVAL...               |
+-----------------------------------------------------------+
| 1998-01-01 00:01:00                                       |
+-----------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT DATE_ADD('1999-01-01', INTERVAL 1 HOUR);
+-----------------------------------------------------------+
| DATE_ADD('1999-01-01', INTERVAL 1 HOUR)                   |
+-----------------------------------------------------------+
| 1999-01-01 01:00:00                                       |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

DATE_FORMAT(date,format)

根据格式字符串对日期值进行格式化。

下面这些占位符可以用在格式字符串中，'%' 必须出现在特定的格式字符之前。

占位符	描述
%a	简写的星期名称（Sun..Sat）
%b	简写的月份名称（Jan..Dec）
%c	月份，以数值形式表示（0..12）
%D	月份中的日期，带有英文后缀（0th，1st，2nd，3rd 等等）
%d	月份中的日期，以数值表示 (00..31)
%e	月份中的日期，以数值表示 (0..31)
%f	微秒（000000..999999）
%H	小时(00..23)
%h	小时(01..12)
%I	小时(01..12)
%i	分钟,以数值表示(00..59)
%j	一年中的第几天（001..366）
%k	小时（0..23）
%l	小时（1..12）
%M	月份的名称（January..December）
%m	月份，以数值形式表示（00..12）
%p	AM 或者 PM
%r	时间，12 小时制(hh:mm:ss followed by AM or PM)
%S	秒(00..59)
%s	秒(00..59)
%T	时间，24小时制（hh:mm:ss）
%U	星期（00..53），此处星期日为一周的第一天
%u	星期（00..53），此处星期一为一周的第一天
%V	星期（01..53），此处星期日为一周的第一天；与 %X 一起使用。
%v	星期（01..53），此处星期一为一周的第一天；与 %x 一起使用。
%W	一周中日期的名称（Sunday..Saturday）
%w	一周中的第几天（0=Sunday..6=Saturday）
%X	以星期日为第一天的周所处的年份，四位数字表示；同 %V 一起使用。
%x	以星期一为第一天的周所处的年份，四位数字表示；同 %v 一起使用。
%Y	年份，四位数字表示。
%y	年份，两位数字表示。
%%	% 字面值
%x	x，针对任何以上没有列出的情况。

```
mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00', '%W %M %Y');
+-----------------------------------------------------------+
| DATE_FORMAT('1997-10-04 22:23:00', '%W %M %Y')          |
+-----------------------------------------------------------+
| Saturday October 1997                                     |
+-----------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT DATE_FORMAT('1997-10-04 22:23:00'
    -> '%H %k %I %r %T %S %w');
+-----------------------------------------------------------+
| DATE_FORMAT('1997-10-04 22:23:00.....                   |
+-----------------------------------------------------------+
|  22 22 10 10:23:00 PM 22:23:00 00 6                       |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

DATE_SUB(date,INTERVAL expr unit)

同 DATE_ADD() 函数相似。

DAY(date)

DAY() 是 DAYOFMONTH() 的别名。

DAYNAME(date)

返回 date 在星期中的名称。

```
mysql> SELECT DAYNAME('1998-02-05');
+-----------------------------------------------------------+
| DAYNAME('1998-02-05')                                     |
+-----------------------------------------------------------+
| Thursday                                                  |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

DAYOFMONTH(date)

返回 date 是当月的第几天，范围为 0 到 31。

```
mysql> SELECT DAYOFMONTH('1998-02-03');
+-----------------------------------------------------------+
| DAYOFMONTH('1998-02-03')                                  |
+-----------------------------------------------------------+
| 3                                                         |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

DAYOFWEEK(date)

返回 date 是其所在星期的第几天(1 = Sunday, 2 = Monday,.., 7 = Saturday)，这里一星期中日期的名称与数字的对应关系符合 ODBC 标准。

```
mysql> SELECT DAYOFWEEK('1998-02-03');
+-----------------------------------------------------------+
|DAYOFWEEK('1998-02-03')                                    |
+-----------------------------------------------------------+
| 3                                                         |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

DAYOFYEAR(date)

返回 date 是当年的第几天，范围为 1 到 366。

```
mysql> SELECT DAYOFYEAR('1998-02-03');
+-----------------------------------------------------------+
| DAYOFYEAR('1998-02-03')                                   |
+-----------------------------------------------------------+
| 34                                                        |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

EXTRACT(unit FROM date)

EXTRACT() 与 DATE_ADD() 和 DATE_SUB() 使用相同的表示单位的占位符，其作用是提取日期值中相应的组成部分，而不是进行日期运算。

```
mysql> SELECT EXTRACT(YEAR FROM '1999-07-02');
+-----------------------------------------------------+
| EXTRACT(YEAR FROM '1999-07-02')                     |
+-----------------------------------------------------+
| 1999                                                |
+-----------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT EXTRACT(YEAR_MONTH FROM '1999-07-02 01:02:03');
+-----------------------------------------------------+
| EXTRACT(YEAR_MONTH FROM '1999-07-02 01:02:03')      |
+-----------------------------------------------------+
| 199907                                              |
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

FROM_DAYS(N)

给出天数 N，返回 DATE 值。

```
mysql> SELECT FROM_DAYS(729669);
+-----------------------------------------------------+
| FROM_DAYS(729669)                                   |
+-----------------------------------------------------+
| 1997-10-07                                          |
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

在使用 FROM_DAYS() 处理比较老的日期的时候应当特别小心，该函数不适用于格里高利历诞生（1582）之前的日期。

FROM_UNIXTIME(unix_timestamp), FROM_UNIXTIME(unix_timestamp,format)

返回 UNIX 时间戳对应的日期值，根据函数所处的上下文环境不同，返回值得格式也不同，字符串上下文返回格式为 'YYYY-MM-DD HH:MM:SS'，数值型上下文返回格式则为 YYYYMMDDHHMMSS。返回值的时区为系统当前时区。UNIX 时间戳是一种系统内部时间表示，例如 UNIX_TIMESTAMP() 的返回值。

如果给定格式的话，返回结果将会根据格式字符串进行格式化，其规则同 DATE_FORMAT() 函数。

```
mysql> SELECT FROM_UNIXTIME(875996580);
+---------------------------------------------------------+
| FROM_UNIXTIME(875996580)                                |
+---------------------------------------------------------+
| 1997-10-04 22:23:00                                     |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

HOUR(time)

返回时间值的小时部分。对于一天中的时间来说，返回值的范围为 0 到 23。不过，TIME 类型的值可以大得多，所以 HOUR 函数可以返回比 23 大的值。

```
mysql> SELECT HOUR('10:05:03');
+---------------------------------------------------------+
| HOUR('10:05:03')                                        |
+---------------------------------------------------------+
| 10                                                      |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

LAST_DAY(date)

返回 date 或者 datetime 值所在月份的最后一天。如果参数无效的话，返回　NULL。

```
mysql> SELECT LAST_DAY('2003-02-05');
+---------------------------------------------------------+
| LAST_DAY('2003-02-05')                                  |
+---------------------------------------------------------+
| 2003-02-28                                              |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

LOCALTIME and LOCALTIME()

LOCALTIME 和 LOCALTIME() 是 NOW() 的别名。

LOCALTIMESTAMP and LOCALTIMESTAMP()

LOCALTIMESTAMP 和 LOCALTIMESTAMP() 是 NOW() 的别名。

MAKEDATE(year,dayofyear)

给定年份和（某天在一年中）的天数，返回对应的日期值。天数必须大于 0，否则返回值为 NULL。

```
mysql> SELECT MAKEDATE(2001,31), MAKEDATE(2001,32);
+-------------------------------------------------------+
| MAKEDATE(2001,31), MAKEDATE(2001,32)                  |
+-------------------------------------------------------+
| '2001-01-31', '2001-02-01'                            |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

MAKETIME(hour,minute,second)

根据参数给出的时、分、秒，返回对应的时间值。

```
mysql> SELECT MAKETIME(12,15,30);
+-------------------------------------------------------+
| MAKETIME(12,15,30)                                    |
+-------------------------------------------------------+
| '12:15:30'                                            |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

MICROSECOND(expr)

根据 time 或者 datetime 表达式 expr，返回微秒数，结果在 0 到 999999 之间。

```
mysql> SELECT MICROSECOND('12:00:00.123456');
+-----------------------------------------------------------+
| MICROSECOND('12:00:00.123456')                            |
+-----------------------------------------------------------+
| 123456                                                    |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

MINUTE(time)

返回时间型值中的分钟部分，范围为 0 到 59。

```
mysql> SELECT MINUTE('98-02-03 10:05:03');
+-----------------------------------------------------------+
| MINUTE('98-02-03 10:05:03')                               |
+-----------------------------------------------------------+
| 5                                                         |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

MONTH(date)

返回日期型值中的月份，范围为 0 到 12。

```
mysql> SELECT MONTH('1998-02-03')
+-----------------------------------------------------------+
| MONTH('1998-02-03')                                       |
+-----------------------------------------------------------+
| 2                                                         |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

MONTHNAME(date)

返回日期型值所处月份的全名。

```
mysql> SELECT MONTHNAME('1998-02-05');
+-----------------------------------------------------------+
| MONTHNAME('1998-02-05')                                   |
+-----------------------------------------------------------+
| February                                                  |
```

```
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

NOW()

返回当前的日期和时间，结果的格式为 'YYYY-MM-DD HH:MM:SS' 或者 YYYYMMDDHHMMSS，如果函数上下文环境为字符型，则返回前者，否则如果函数处于数值型的上下文环境，则返回后者。返回值的时区为系统当前时区。

```
mysql> SELECT NOW();
+-----------------------------------------------------+
| NOW()                                               |
+-----------------------------------------------------+
| 1997-12-15 23:50:26                                 |
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

PERIOD_ADD(P,N)

在时间 P（格式为 YYMM 或者 YYYYMM）上加上 N 个月，结果格式为 YYYYMM。注意，时间参数 P 并不是日期型值。

```
mysql> SELECT PERIOD_ADD(9801,2);
+-----------------------------------------------------+
| PERIOD_ADD(9801,2)                                  |
+-----------------------------------------------------+
| 199803                                              |
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

PERIOD_DIFF(P1,P2)

返回时间 P1 和 P2 之间相差的月份。P1 和 P2 的格式应为 YYMM 或者 YYYYMM。注意I，P1 和 P2 不是日期型值。

```
mysql> SELECT PERIOD_DIFF(9802,199703);
+-----------------------------------------------------+
| PERIOD_DIFF(9802,199703)                            |
+-----------------------------------------------------+
| 11                                                  |
```

```
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

QUARTER(date)

返回日期型值 date 所处的季度值，范围为 1 到 4。

```
mysql> SELECT QUARTER('98-04-01');
+-------------------------------------------------------+
| QUARTER('98-04-01')                                   |
+-------------------------------------------------------+
| 2                                                     |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

SECOND(time)

返回时间型值中秒的部分，范围为 0 到 59。

```
mysql> SELECT SECOND('10:05:03');
+-------------------------------------------------------+
| SECOND('10:05:03')                                    |
+-------------------------------------------------------+
| 3                                                     |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

SEC_TO_TIME(seconds)

将参数中的秒数转换为时分秒的格式 'HH:MM:SS' 或者 HHMMSS，如果函数所处的上下文为字符串型，则返回前者，否则如果上下文环境为数值型，则返回后者。

STR_TO_DATE(str,format)

这是 DATE_FORMATE() 函数的逆函数，其参数为表示时间和日期的字符串 str 和一个格式字符串 format。如果格式字符串中既有日期又有时间，则 STR_TO_DATE() 返回 DATETIME() 型的值，否则返回日期型（DATE）或者时间型（TIME）的值。

```
mysql> SELECT STR_TO_DATE('04/31/2004', '%m/%d/%Y');
+---------------------------------------------------+
| STR_TO_DATE('04/31/2004', '%m/%d/%Y')             |
+---------------------------------------------------+
| 2004-04-31                                        |
+---------------------------------------------------+
1 row in set (0.00 sec)
```

SUBDATE(date,INTERVAL expr unit) and SUBDATE(expr,days)

当第二个参数为 INTERVAL 形式时，SUBDATE() 就是 DATE_SUB() 的别名。INTERVAL 参数中单位的信息，请见有关 DATE_ADD() 的讨论。

```
mysql> SELECT DATE_SUB('1998-01-02', INTERVAL 31 DAY);
+---------------------------------------------------+
| DATE_SUB('1998-01-02', INTERVAL 31 DAY)           |
+---------------------------------------------------+
| 1997-12-02                                        |
+---------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT SUBDATE('1998-01-02', INTERVAL 31 DAY);
+---------------------------------------------------+
| SUBDATE('1998-01-02', INTERVAL 31 DAY)            |
+---------------------------------------------------+
| 1997-12-02                                        |
+---------------------------------------------------+
1 row in set (0.00 sec)
```

SUBTIME(expr1,expr2)

SUBTIME() 返回 expr1-expr2，结果的格式与 expr1 相同。expr1 是一个时间型（time）或者 datetime 型的表达式，expr2 是时间型值。

SYSDATE()

返回当前的日期和时间，格式为 'YYYY-MM-DD HH:MM:SS' 或 YYYYMMDDHHMMSS，如果函数所处的上下文环境为字符串，则返回前者，否则如果上下文环境为数值型，则返回后者。

```
mysql> SELECT SYSDATE();
+---------------------------------------------------------+
| SYSDATE()                                               |
+---------------------------------------------------------+
| 2006-04-12 13:47:44                                     |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

TIME(expr)

提取时间型或者 datetime 型表达式 expr 中的时间部分，返回结果为字符串。

```
mysql> SELECT TIME('2003-12-31 01:02:03');
+---------------------------------------------------------+
| TIME('2003-12-31 01:02:03')                             |
+---------------------------------------------------------+
| 01:02:03                                                |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

TIMEDIFF(expr1,expr2)

TIMEDIFF() 返回 expr1−expr2，结果为时间型值。expr1 和 expr2 可以为时间型或者 datetime 型表达式，不过二者必须为相同类型。

```
mysql> SELECT TIMEDIFF('1997-12-31 23:59:59.000001',
    -> '1997-12-30 01:01:01.000002');
+---------------------------------------------------------+
| TIMEDIFF('1997-12-31 23:59:59.000001'.....              |
+---------------------------------------------------------+
|  46:58:57.999999                                        |
+---------------------------------------------------------+
1 row in set (0.00 sec)
```

TIMESTAMP(expr), TIMESTAMP(expr1,expr2)

只有一个参数的时候，该函数由日期型或者 datetime 型表达式返回一个 datetime 型值。有两个参数的时候，该函数将 expr2 加到日期型或 datetime 型值 expr1 上，并返回 datetime 型的结果。

```
mysql> SELECT TIMESTAMP('2003-12-31');
+---------------------------------------------------------+
```

```
| TIMESTAMP('2003-12-31')                          |
+--------------------------------------------------+
| 2003-12-31 00:00:00                              |
+--------------------------------------------------+
1 row in set (0.00 sec)
```

TIMESTAMPADD(unit,interval,datetime_expr)

将整数型的表达式 interval 加到日期型或者 datetime 型表达式 datetime_expr 上。单位由 unit 参数给出，其取值应为以下几种中的一种：FRAC_SECOND、SECOND、MINUTE、HOUR、DAY、WEEK、MONTH、QUARTER 或者 YEAR。

单位 unit 可以为上述关键字中的一个，也可以添加一个 SQL_TSI_ 前缀，例如 DAY 和 SQL_TSI_DAY 都是合法的。

```
mysql> SELECT TIMESTAMPADD(MINUTE,1,'2003-01-02');
+--------------------------------------------------+
| TIMESTAMPADD(MINUTE,1,'2003-01-02')              |
+--------------------------------------------------+
| 2003-01-02 00:01:00                              |
+--------------------------------------------------+
1 row in set (0.00 sec)
```

TIMESTAMPDIFF(unit,datetime_expr1,datetime_expr2)

返回日期型或者 datetime 型表达式 datetime_expr1 和 datetime_expr2 的差。结果的单位由 unit 参数给出，unit 的取值规定同 TIMESTAMPADD() 函数。

```
mysql> SELECT TIMESTAMPDIFF(MONTH,'2003-02-01','2003-05-01');
+--------------------------------------------------+
| TIMESTAMPDIFF(MONTH,'2003-02-01','2003-05-01')   |
+--------------------------------------------------+
| 3                                                |
+--------------------------------------------------+
1 row in set (0.00 sec)
```

TIME_FORMAT(time,format)

该函数使用起来类似 DATE_FORMAT() 函数，但是格式字符串 format 中只能有与小时、分钟和秒有关的那些占位符。

如果时间型值的小时部分大于 23，则 %H 和 %k 格式占位符将会产生一个大于通常的 0-23 的值，其他与小时有关的占位符则会返回小时值除以 12 后的余数（modulo 12）。

```
mysql> SELECT TIME_FORMAT('100:00:00', '%H %k %h %I %l');
+-------------------------------------------------------+
| TIME_FORMAT('100:00:00', '%H %k %h %I %l')            |
+-------------------------------------------------------+
| 100 100 04 04 4                                       |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

TIME_TO_SEC(time)

将时间型值转换为秒。

```
mysql> SELECT TIME_TO_SEC('22:23:00');
+-------------------------------------------------------+
| TIME_TO_SEC('22:23:00')                               |
+-------------------------------------------------------+
| 80580                                                 |
+-------------------------------------------------------+
1 row in set (0.00 sec)、
```

TO_DAYS(date)

给定日期型值 date，返回天数（自公元 0 年以来的天数）。

```
mysql> SELECT TO_DAYS(950501);
+-------------------------------------------------------+
| TO_DAYS(950501)                                       |
+-------------------------------------------------------+
| 728779                                                |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

UNIX_TIMESTAMP(), UNIX_TIMESTAMP(date)

不带任何参数时，该函数返回一个 unsigned integer 型的 UNIX 时间戳（自 '1970-01-01 00:00:00' UTC 以来的秒数）。如果有一个参数 date 的话，该函数返回自 '1970-01-01 00:00:00' UTC 至 date 的秒数。date 可以是日期型的字符串、DATETIME 型的字符串、时间戳或者 YYMMDD 或 YYYYMMDD 格式的数字。

```
mysql> SELECT UNIX_TIMESTAMP();
+------------------------------------------------------+
| UNIX_TIMESTAMP()                                     |
+------------------------------------------------------+
| 882226357                                            |
+------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT UNIX_TIMESTAMP('1997-10-04 22:23:00');
+------------------------------------------------------+
| UNIX_TIMESTAMP('1997-10-04 22:23:00')                |
+------------------------------------------------------+
| 875996580                                            |
+------------------------------------------------------+
1 row in set (0.00 sec)
```

UTC_DATE, UTC_DATE()

返回当前 UTC 日期，格式为 'YYYY-MM-DD' 或者 YYYYMMDD，如果函数所处的上下文环境为字符串，则返回前者，否则如果上下文环境为数值型的，则返回后者。

```
mysql> SELECT UTC_DATE(), UTC_DATE() + 0;
+------------------------------------------------------+
| UTC_DATE(), UTC_DATE() + 0                           |
+------------------------------------------------------+
| 2003-08-14, 20030814                                 |
+------------------------------------------------------+
1 row in set (0.00 sec)
```

UTC_TIME, UTC_TIME()

返回当前 UTC 时间，格式为 'HH:MM:SS' 或者 HHMMSS，如果函数所处的上下文环境为字符串，则返回前者，否则如果上下文环境为数值型的，则返回后者。

```
mysql> SELECT UTC_TIMESTAMP(), UTC_TIMESTAMP() + 0;
+------------------------------------------------------+
| UTC_TIMESTAMP(), UTC_TIMESTAMP() + 0                 |
+------------------------------------------------------+
| 2003-08-14 18:08:04, 20030814180804                  |
+------------------------------------------------------+
1 row in set (0.00 sec)
```

WEEK(date[,mode])

该函数将返回 date 所在的周是当年的第几周。两个参数的 WEEK() 函数的使你能够指明一周起始于周日还是周一，以及返回值的范围应该是 0 到 53，还是 1 到 53。如果 mode 参数被忽略，则将使用 default_week_format 系统变量。

Mode	一周的第一天	范围	周 1 是第一周
0	Sunday	0–53	该年包括一个星期天
1	Monday	0–53	该年包含超过 3 天
2	Sunday	1–53	该年包括一个星期天
3	Monday	1–53	该年包含超过 3 天
4	Sunday	0–53	该年包含超过 3 天
5	Monday	0–53	该年包括一个星期一
6	Sunday	1–53	该年包含超过 3 天
7	Monday	1–53	该年包括一个星期一

```
mysql> SELECT WEEK('1998-02-20');
+-------------------------------------------------------------+
| WEEK('1998-02-20')                                          |
+-------------------------------------------------------------+
| 7                                                           |
+-------------------------------------------------------------+
1 row in set (0.00 sec)
```

WEEKDAY(date)

返回 date 是其所在星期的第几天 (0 = Monday, 1 = Tuesday, . 6 = Sunday)。

```
mysql> SELECT WEEKDAY('1998-02-03 22:23:00');
+-------------------------------------------------------------+
| WEEKDAY('1998-02-03 22:23:00')                              |
+-------------------------------------------------------------+
| 1                                                           |
+-------------------------------------------------------------+
1 row in set (0.00 sec)
```

WEEKOFYEAR(date)

返回 date 所在的周是当年的第几周，范围从 1 到 53. WEEKOFYEAR() 是一个兼容性函数，其功能同 WEEK(date, 3)相同。

```
mysql> SELECT WEEKOFYEAR('1998-02-20');
+-----------------------------------------------------------+
| WEEKOFYEAR('1998-02-20')                                  |
+-----------------------------------------------------------+
| 8                                                         |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

YEAR(date)

返回 date 的年份部分，范围为 1000 到 9999，对于日期 0 则返回 0。

```
mysql> SELECT YEAR('98-02-03');
+-----------------------------------------------------------+
| YEAR('98-02-03')                                          |
+-----------------------------------------------------------+
| 1998                                                      |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

YEARWEEK(date), YEARWEEK(date,mode)

返回 date 所在的年份和周数。mode 参数意义与 WEEK() 函数的完全一样。对于一年中的第一周和最后一周来说，结果中的年份可能会和 date 参数中的年份不同。

```
mysql> SELECT YEARWEEK('1987-01-01');
+-----------------------------------------------------------+
| YEAR('98-02-03')YEARWEEK('1987-01-01')                    |
+-----------------------------------------------------------+
| 198653                                                    |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

注意，这里的周数同 WEEK() 返回的不同，因为 WEEK() 函数的返回值在给定年份的的上下文环境中得出。

临时表

某些关系型数据库管理系统支持临时表。临时表是一项很棒的特性，能够让你像操作普通的 SQL 数据表一样，使用 SELECT、UPDATE 和 JOIN 等功能来存储或者操作中间结果。

临时表有时候对于保存临时数据非常有用。有关临时表你需要知道的最重要的一点是，它们会在当前的终端会话结束后被删除。

临时表自 MySQL 3.23 起受到支持。如果你的 MySQL 版本比 3.23 还老，那么你就不能使用临时表了，不过你可以使用堆表（heap table）。

如先前所言，临时表只在会话期间存在。如果你在 PHP 脚本中操作数据库，那么临时表将在脚本执行完毕时被自动销毁。如果你是通过 MySQL 的客户端程序连接到 MySQL 数据库服务器的，那么临时表将会存在到你关闭客户端或者手动将其删除。

示例：

下面的示例向你展示了如何使用临时表：

```
mysql> CREATE TEMPORARY TABLE SALESSUMMARY (
    -> product_name VARCHAR(50) NOT NULL
    -> , total_sales DECIMAL(12,2) NOT NULL DEFAULT 0.00
    -> , avg_unit_price DECIMAL(7,2) NOT NULL DEFAULT 0.00
    -> , total_units_sold INT UNSIGNED NOT NULL DEFAULT 0
);
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO SALESSUMMARY
    -> (product_name, total_sales, avg_unit_price, total_units_sold)
    -> VALUES
    -> ('cucumber', 100.25, 90, 2);

mysql> SELECT * FROM SALESSUMMARY;
+--------------+-------------+----------------+------------------+
| product_name | total_sales | avg_unit_price | total_units_sold |
+--------------+-------------+----------------+------------------+
| cucumber     |      100.25 |          90.00 |                2 |
+--------------+-------------+----------------+------------------+
1 row in set (0.00 sec)
```

当你下达 SHOW TABLES 命令的时候，临时表是不会出现在结果列表当中的。现在，如果你退出 MySQL 会话，然后再执行 SELECT 命令的话，你将不能从数据库中取回任何数据，你的临时表也已经不复存在了。

删除临时表：

默认情况下，所有的临时表都由 MySQL 在数据库连接关闭时删除。不过，有时候你还是会想要在会话期间将其删除，此时你需要使用 DROP TABLE 命令来达到目的。

下面是删除临时表的示例：

```
mysql> CREATE TEMPORARY TABLE SALESSUMMARY (
    -> product_name VARCHAR(50) NOT NULL
    -> , total_sales DECIMAL(12,2) NOT NULL DEFAULT 0.00
    -> , avg_unit_price DECIMAL(7,2) NOT NULL DEFAULT 0.00
    -> , total_units_sold INT UNSIGNED NOT NULL DEFAULT 0
);
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO SALESSUMMARY
    -> (product_name, total_sales, avg_unit_price, total_units_sold)
    -> VALUES
    -> ('cucumber', 100.25, 90, 2);

mysql> SELECT * FROM SALESSUMMARY;
+--------------+-------------+----------------+-----------------+
| product_name | total_sales | avg_unit_price | total_units_sold |
+--------------+-------------+----------------+-----------------+
| cucumber     |      100.25 |          90.00 |               2 |
+--------------+-------------+----------------+-----------------+
1 row in set (0.00 sec)
mysql> DROP TABLE SALESSUMMARY;
mysql>  SELECT * FROM SALESSUMMARY;
ERROR 1146: Table 'TUTORIALS.SALESSUMMARY' doesn't exist
```

克隆数据表

有些情况下，你可能需要原样拷贝某张数据表。但是，CREATE TABEL 却不能满足你的需要，因为复制表必须和原表拥有一样的索引、默认值等等。

如果你在使用 MySQL 关系型数据库管理系统的话，下面几个步骤可以帮你解决这个问题：

- 使用 SHOW CREATE TABLE 命令来获取一条指定了原表的结构、索引等信息的 CREATE TABLE 语句。
- 将语句中的表名修改为克隆表的名字，然后执行该语句。这样你就可以得到一张与原表完全相同的克隆表了。
- 如果你还想要复制表中的数据的话，请执行 INSERT INTO ... SELECT 语句。

示例：

请尝试下面的示例，为 TUTORIALS_TBL 创建一张克隆表，其结构如下所示：

步骤一：

获取数据表的完整结构：

```
SQL> SHOW CREATE TABLE TUTORIALS_TBL \G;
*************************** 1. row ***************************
       Table: TUTORIALS_TBL
Create Table: CREATE TABLE `TUTORIALS_TBL` (
  `tutorial_id` int(11) NOT NULL auto_increment,
  `tutorial_title` varchar(100) NOT NULL default '',
  `tutorial_author` varchar(40) NOT NULL default '',
  `submission_date` date default NULL,
  PRIMARY KEY  (`tutorial_id`),
  UNIQUE KEY `AUTHOR_INDEX` (`tutorial_author`)
) TYPE=MyISAM
1 row in set (0.00 sec)
```

步骤二：

改变表名，创建新表：

```
SQL> CREATE TABLE `CLONE_TBL` (
  -> `tutorial_id` int(11) NOT NULL auto_increment,
  -> `tutorial_title` varchar(100) NOT NULL default '',
  -> `tutorial_author` varchar(40) NOT NULL default '',
  -> `submission_date` date default NULL,
  -> PRIMARY KEY (`tutorial_id`),
  -> UNIQUE KEY `AUTHOR_INDEX` (`tutorial_author`)
  -> ) TYPE=MyISAM;
Query OK, 0 rows affected (1.80 sec)
```

步骤三：

执行完步骤二之后，数据库就会有克隆表了。如果你还想要复制旧表中的数据的话，可以执行 INSERT INTO... SELECT 语句。

```
SQL> INSERT INTO CLONE_TBL (tutorial_id,
    ->                      tutorial_title,
    ->                      tutorial_author,
    ->                      submission_date)
    -> SELECT tutorial_id,tutorial_title,
    ->       tutorial_author,submission_date,
    -> FROM TUTORIALS_TBL;
Query OK, 3 rows affected (0.07 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

最终，你将如期拥有一张完全相同的克隆表。

子查询

子查询（Sub Query）或者说内查询（Inner Query），也可以称作嵌套查询（Nested Query），是一种嵌套在其他 SQL 查询的 WHERE 子句中的查询。

子查询用于为主查询返回其所需数据，或者对检索数据进行进一步的限制。

子查询可以在 SELECT、INSERT、UPDATE 和 DELETE 语句中，同 =、<、>、>=、<=、IN、BETWEEN 等运算符一起使用。

使用子查询必须遵循以下几个规则：

- 子查询必须括在圆括号中。
- 子查询的 SELECT 子句中只能有一个列，除非主查询中有多个列，用于与子查询选中的列相比较。
- 子查询不能使用 ORDER BY，不过主查询可以。在子查询中，GROUP BY 可以起到同 ORDER BY 相同的作用。
- 返回多行数据的子查询只能同多值操作符一起使用，比如 IN 操作符。
- SELECT 列表中不能包含任何对 BLOB、ARRAY、CLOB 或者 NCLOB 类型值的引用。
- 子查询不能直接用在集合函数中。
- BETWEEN 操作符不能同子查询一起使用，但是 BETWEEN 操作符可以用在子查询中。

SELECT 语句中的子查询

通常情况下子查询都与 SELECT 语句一起使用，其基本语法如下所示：

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
    (SELECT column_name [, column_name ]
    FROM table1 [, table2 ]
    [WHERE])
```

示例：

考虑 CUSTOMERS 表，表中记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

现在，让我们试一下在 SELECT 语句中进行子查询：

```
SQL> SELECT *
    FROM CUSTOMERS
    WHERE ID IN (SELECT ID
                 FROM CUSTOMERS
                 WHERE SALARY > 4500) ;
```

上述语句的执行结果如下所示：

```
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  5 | Hardik   |  27 | Bhopal  |  8500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
```

INSERT 语句中的子查询：

子查询还可以用在 INSERT 语句中。INSERT 语句可以将子查询返回的数据插入到其他表中。子查询中选取的数据可以被任何字符、日期或者数值函数所修饰。

其基本语法如下所示：

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
        SELECT [ *|column1 [, column2 ]
        FROM table1 [, table2 ]
        [ WHERE VALUE OPERATOR ]
```

示例：

考虑与 CUSTOMERS 表拥有相似结构的 CUSTOMERS_BKP 表。现在要将 CUSTOMER 表中所有的数据复制到 CUSTOMERS_BKP 表中，代码如下：

```
SQL> INSERT INTO CUSTOMERS_BKP
   SELECT * FROM CUSTOMERS
   WHERE ID IN (SELECT ID
            FROM CUSTOMERS) ;
```

UPDATE 语句中的子查询：

子查询可以用在 UPDATE 语句中。当子查询同 UPDATE 一起使用的时候，既可以更新单个列，也可更新多个列。

其基本语法如下：

```
UPDATE table
SET column_name = new_value
[ WHERE OPERATOR [ VALUE ]
  (SELECT COLUMN_NAME
  FROM TABLE_NAME)
  [ WHERE) ]
```

示例：

假设我们有一份 CUSTOMERS_BKP 表作为 CUSTOMERS 表的备份。

下面的示例将 CUSTOMERS 表中所有 AGE 大于或者等于 27 的客户的 SALARY 字段都变为了原来的 0.25 倍：

```
SQL> UPDATE CUSTOMERS
   SET SALARY = SALARY * 0.25
   WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
            WHERE AGE >= 27 );
```

这将影响两行数据，随后 CUSTOMERS 表中的记录将如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
```

```
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 35 | Ahmedabad |   125.00 |
| 2 | Khilan   | 25 | Delhi     |  1500.00 |
| 3 | kaushik  | 23 | Kota      |  2000.00 |
| 4 | Chaitali | 25 | Mumbai    |  6500.00 |
| 5 | Hardik   | 27 | Bhopal    |  2125.00 |
| 6 | Komal    | 22 | MP        |  4500.00 |
| 7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

DELETE 语句中的子查询：

如同前面提到的其他语句一样，子查询还可以同 DELETE 语句一起使用。

其基本语法如下所示：

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
  (SELECT COLUMN_NAME
  FROM TABLE_NAME)
  [ WHERE) ]
```

示例：

假设我们有一份 CUSTOMERS_BKP 表作为 CUSTOMERS 表的备份。

下面的示例将从 CUSTOMERS 表中删除所有 AGE 大于或者等于 27 的记录：

```
SQL> DELETE FROM CUSTOMERS
    WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
                WHERE AGE > 27 );
```

这将影响两行数据，随后 CUSTOMERS 表中的记录将如下所示：

```
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
| 2 | Khilan   | 25 | Delhi   | 1500.00 |
| 3 | kaushik  | 23 | Kota    | 2000.00 |
| 4 | Chaitali | 25 | Mumbai  | 6500.00 |
| 6 | Komal    | 22 | MP      | 4500.00 |
| 7 | Muffy    | 24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
```

使用序列

序列是根据需要产生的一组有序整数：1, 2, 3 ... 序列在数据库中经常用到，因为许多应用要求数据表中的的每一行都有一个唯一的值，序列为此提供了一种简单的方法。

本节阐述在 MySQL 中如何使用序列。

使用 AUTO_INCREMENT 列：

在 MySQL 中使用序列最简单的方式是，把某列定义为 AUTO_INCREMENT，然后将剩下的事情交由 MySQL 处理：

示例：

试一下下面的例子，该例将会创建一张新表，然后再里面插入几条记录，添加记录时并不需要指定记录的 ID，因为该列的值由 MySQL 自动增加。

```
mysql> CREATE TABLE INSECT
    -> (
    -> id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    -> PRIMARY KEY (id),
    -> name VARCHAR(30) NOT NULL, # type of insect
    -> date DATE NOT NULL, # date collected
    -> origin VARCHAR(30) NOT NULL # where collected
);
Query OK, 0 rows affected (0.02 sec)
mysql> INSERT INTO INSECT (id,name,date,origin) VALUES
    -> (NULL,'housefly','2001-09-10','kitchen'),
    -> (NULL,'millipede','2001-09-10','driveway'),
    -> (NULL,'grasshopper','2001-09-10','front yard');
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
mysql> SELECT * FROM INSECT ORDER BY id;
+----+-------------+------------+------------+
| id | name        | date       | origin     |
+----+-------------+------------+------------+
|  1 | housefly    | 2001-09-10 | kitchen    |
|  2 | millipede   | 2001-09-10 | driveway   |
|  3 | grasshopper | 2001-09-10 | front yard |
```

```
+----+-------------+-----------+------------+
3 rows in set (0.00 sec)
```

获取 AUTO_INCREMENT 值：

LAST_INSERT_ID() 是一个 SQL 函数，可以用在任何能够执行 SQL 语句地方。另外，Perl 和 PHP 各自提供了其独有的函数，用于获得最后一条记录的 AUTO_INCREMENT 值。

Perl 示例：

使用 mysql_insertid 属性来获取 SQL 查询产生的 AUTO_INCREMENT 值。根据执行查询的方式不同，该属性可以通过数据库句柄或者语句句柄来访问。下面的示例通过数据库句柄取得自增值：

```
$dbh->do ("INSERT INTO INSECT (name,date,origin)
VALUES('moth','2001-09-14','windowsill')");
my $seq = $dbh->{mysql_insertid};
```

PHP 示例：

在执行完会产生自增值的查询后，可以通过调用 mysql_insert_id() 来获取此值：

```
mysql_query ("INSERT INTO INSECT (name,date,origin)
VALUES('moth','2001-09-14','windowsill')", $conn_id);
$seq = mysql_insert_id ($conn_id);
```

重新编号现有序列：

当你从表中删除了很多记录后，可能会想要对所有的记录重新定序。只要略施小计就能达到此目的，不过如果你的表与其他表之间存在连接的话，请千万小心。

当你觉得不得不对 AUTO_INCREMENT 列重新定序时，从表中删除该列，然后再将其添加回来，就可以达到目的了。下面的示例展示了如何使用这种方法，为 INSECT 表中的 ID 值重新定序：

```
mysql> ALTER TABLE INSECT DROP id;
mysql> ALTER TABLE insect
    -> ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT FIRST,
    -> ADD PRIMARY KEY (id);
```

从特定值的序列

默认情况下，MySQL 中序列的起始值为 1，不过你可以在创建数据表的时候，指定任意其他值。下面的示例中，MySQL 将序列的起始值设为 100：

```
mysql> CREATE TABLE INSECT
    -> (
    -> id INT UNSIGNED NOT NULL AUTO_INCREMENT = 100,
    -> PRIMARY KEY (id),
    -> name VARCHAR(30) NOT NULL, # type of insect
    -> date DATE NOT NULL, # date collected
    -> origin VARCHAR(30) NOT NULL # where collected
);
```

或者，你也可以先创建数据表，然后使用 ALTER TABLE 来设置序列的起始值：

```
mysql> ALTER TABLE t AUTO_INCREMENT = 100;
```

处理重复数据

有时候，数据表中会存在相同的记录。在获取表中记录时，相较于取得重复记录来说，取得唯一的记录显然更有意义。

我们之前讨论过的 SQL DISTINCT 关键字，与 SELECT 语句一起使用可以时，可以达到消除所有重复记录，只返回唯一记录的目的。

语法：

利用 DISTINCT 关键字来消除重复记录的基本语法如下所示：

```
SELECT DISTINCT column1, column2,.....columnN
FROM table_name
WHERE [condition]
```

示例：

考虑 CUSTOMERS 表，表中记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

首先，让我们先看一下 SELECT 语句是如何返回重复的薪水记录的：

```
SQL> SELECT SALARY FROM CUSTOMERS
   ORDER BY SALARY;
```

运行上述语句将会得到以下结果，其中 SALARY 为 2000 的记录出现了两次，即来自原始数据表的重复记录：

```
+----------+
| SALARY   |
+----------+
| 1500.00 |
| 2000.00 |
| 2000.00 |
| 4500.00 |
| 6500.00 |
| 8500.00 |
|10000.00 |
+----------+
```

现在，让我们在上面的 SELECT 查询中使用 DISTINCT 关键字，然后观察将会得到什么结果：

```
SQL> SELECT DISTINCT SALARY FROM CUSTOMERS
   ORDER BY SALARY;
```

上述语句将会产生如下结果，这一再没有任何重复的条目了：

```
+----------+
| SALARY   |
+----------+
| 1500.00 |
| 2000.00 |
| 4500.00 |
| 6500.00 |
| 8500.00 |
|10000.00 |
+----------+
```

注入

如果你从网页中获取用户输入，并将其插入到 SQL 数据库中的话，那么你很可能已经暴露于一种被称作 SQL 注入的安全风险之下了。

本节将会教你如何防止 SQL 注入，以及如何保护 Perl 这样的服务器端脚本中的程序和 SQL 语句。

注入通常发生在获取用户输入的时候，例如预期得到用户的名字，但是得到的却是一段很可能会在你不知情的情况下运行的 SQL 语句。

绝对不要相信用户提供的数据，处理这些数据之前必须进行验证；通常，验证工作由模式匹配来完成。

下面的例子中，name 仅限由字母、数字和下划线组成，并且长度在 8 到 20 之间（你可以根据需要修改这些规则）。

```
if (preg_match("/^\w{8,20}$/", $_GET['username'], $matches))
{
  $result = mysql_query("SELECT * FROM CUSTOMERS
                    WHERE name=$matches[0]");
}
else
{
  echo "user name not accepted";
}
```

为了展示问题所在，请考虑下面这段代码：

```
// supposed input
$name = "Qadir'; DELETE FROM CUSTOMERS;";
mysql_query("SELECT * FROM CUSTOMSRS WHERE name='{$name}'");
```

下面的函数调用本来是要从 CUSTOMERS 表中取得 name 字段与用户给定的输入相匹配的记录。通常情况下，$name 只包含字母和数字，或许还有空格，例如字符串 ilia。但是，这里通过在 $name 上附加一段全新的查询语句，将原有的函数调用变为了数据库的灾难：注入的 DELETE 语句将会删除表中所有的记录。

幸运的是，如果你在使用　MySQL 的话，mysql_query() 函数不允许查询堆积（query stacking），或者说在一次函数调用中执行多次 SQL 查询。如果你试图进行堆积式查询的话，函数调用将会失败。

然而，其他的 PHP 数据库扩展，例如 SQLite 和 PostgreSQL 会愉快地接受堆积式查询，执行字符串中所有的查询，并由此产生严重的安全问题。

阻止 SQL 注入：

你可以在 Perl 或者 PHP 等脚本语言中巧妙地处理所有的转义字符。PHP 的 MySQL 扩展提供了一个 mysql_real_escape_string() 函数，来转义那些对 MySQL 有特殊意义的字符。

```
if (get_magic_quotes_gpc())
{
  $name = stripslashes($name);
}
$name = mysql_real_escape_string($name);
mysql_query("SELECT * FROM CUSTOMERS WHERE name='{$name}'");
```

LIKE 困境：

要破解 LIKE 困境，必须有一种专门的转义机制，将用户提供的 '%' 和 '_' 转换为字面值。为此你可以使用 addcslashes() 函数，该函数允许指定要进行转义的字符的范围。

```
$sub = addcslashes(mysql_real_escape_string("%str"), "%_");
// $sub == \%str\_
mysql_query("SELECT * FROM messages
          WHERE subject LIKE '{$sub}%'");
```

3

SQL 常用函数

COUNT 函数

COUNT函数是 SQL 中最简单的函数了，对于统计由 SELECT 语句返回的记录非常有用。

要理解 COUNT 函数，请考虑 employee_tbl 表，表中的记录如下所示：

```
SQL> SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|   1 | John | 2007-01-24 |              250 |
|   2 | Ram  | 2007-05-27 |              220 |
|   3 | Jack | 2007-05-06 |              170 |
|   3 | Jack | 2007-04-06 |              100 |
|   4 | Jill | 2007-04-06 |              220 |
|   5 | Zara | 2007-06-06 |              300 |
|   5 | Zara | 2007-02-06 |              350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

现在，假设你想要统计上表中记录的总数，那么可以依如下所示步骤达到目的：

```
SQL>SELECT COUNT(*) FROM employee_tbl ;
+----------+
| COUNT(*) |
+----------+
|       7 |
+----------+
1 row in set (0.01 sec)
```

类似地，如果你想要统计 Zara 的数目，就可以像下面这样：

```
SQL>SELECT COUNT(*) FROM employee_tbl
    -> WHERE name="Zara";
+----------+
| COUNT(*) |
+----------+
|       2 |
+----------+
1 row in set (0.04 sec)
```

注意：所有的 SQL 查询都是不区分大小写的，因此在 WHERE 子句的条件中，ZARA 和 Zara 是没有任何区别的。

MAX 函数

MAX函数用于找出记录集中具有最大值的记录。

要理解 MAX 函数，请考虑 employee_tbl 表，表中记录如下所示：

```
SQL> SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|   1 | John | 2007-01-24 |              250 |
|   2 | Ram  | 2007-05-27 |              220 |
|   3 | Jack | 2007-05-06 |              170 |
|   3 | Jack | 2007-04-06 |              100 |
|   4 | Jill | 2007-04-06 |              220 |
|   5 | Zara | 2007-06-06 |              300 |
|   5 | Zara | 2007-02-06 |              350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

现在，假设你想要获取 daily_typing_pages 的最大值，那么你可以用如下命令来达到目的：

```
SQL> SELECT MAX(daily_typing_pages)
   -> FROM employee_tbl;
+------------------------+
| MAX(daily_typing_pages) |
+------------------------+
|                    350 |
+------------------------+
1 row in set (0.00 sec)
```

你还可以使用 GROUP BY 子句，为每个名字（name）找出 daily_typing_pages 的最大值：

```
SQL> SELECT id, name, MAX(daily_typing_pages)
   -> FROM employee_tbl GROUP BY name;
+------+------+------------------------+
| id   | name | MAX(daily_typing_pages) |
+------+------+------------------------+
|   3 | Jack |                    170 |
|   4 | Jill |                    220 |
|   1 | John |                    250 |
|   2 | Ram  |                    220 |
|   5 | Zara |                    350 |
```

```
+------+------+-----------------------+
5 rows in set (0.00 sec)
```

你还可以将 MIN 函数同 MAX 函数一起使用，来找出最小值。试一下下面的例子：

```
SQL> SELECT MIN(daily_typing_pages) least, MAX(daily_typing_pages) max
   -> FROM employee_tbl;
+-------+------+
| least | max  |
+-------+------+
|   100 |  350 |
+-------+------+
1 row in set (0.01 sec)
```

你还可以将 MIN 函数同 MAX 函数一起使用，来找出最小值。试一下下面的例子：

```
SQL> SELECT MIN(daily_typing_pages) least,
   -> MAX(daily_typing_pages) max
   -> FROM employee_tbl;
+-------+------+
| least | max  |
+-------+------+
|   100 |  350 |
+-------+------+
1 row in set (0.01 sec)
```

MIN 函数

MIN函数用于找出记录集中具有最大值的记录。

要理解 MIN 函数，请考虑 employee_tbl 表，表中记录如下所示：

```
SQL> SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|   1 | John | 2007-01-24 |              250 |
|   2 | Ram  | 2007-05-27 |              220 |
|   3 | Jack | 2007-05-06 |              170 |
|   3 | Jack | 2007-04-06 |              100 |
|   4 | Jill | 2007-04-06 |              220 |
|   5 | Zara | 2007-06-06 |              300 |
|   5 | Zara | 2007-02-06 |              350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

现在，假设你想要获取 daily_typing_pages 的最小值，那么你可以用如下命令来达到目的：

```
SQL> SELECT MIN(daily_typing_pages)
   -> FROM employee_tbl;
+------------------------+
| MIN(daily_typing_pages) |
+------------------------+
|                  100 |
+------------------------+
1 row in set (0.00 sec)
```

你还可以使用 GROUP BY 子句，为每个名字（name）找出 daily_typing_pages 的最小值：

```
SQL> SELECT id, name, work_date, MIN(daily_typing_pages)
   -> FROM employee_tbl GROUP BY name;
+------+------+------------------------+
| id   | name | MIN(daily_typing_pages) |
+------+------+------------------------+
|   3 | Jack |                  100 |
|   4 | Jill |                  220 |
|   1 | John |                  250 |
|   2 | Ram  |                  220 |
|   5 | Zara |                  300 |
```

```
+------+------+-----------------------+
5 rows in set (0.00 sec)
```

AVG 函数

AVG函数用于找出表中记录在某字段处的平均值。

要理解 AVG 函数，请考虑 employee_tbl 表，表中记录如下所示：

```
SQL> SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|   1 | John | 2007-01-24 |              250 |
|   2 | Ram  | 2007-05-27 |              220 |
|   3 | Jack | 2007-05-06 |              170 |
|   3 | Jack | 2007-04-06 |              100 |
|   4 | Jill | 2007-04-06 |              220 |
|   5 | Zara | 2007-06-06 |              300 |
|   5 | Zara | 2007-02-06 |              350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

现在，假设你想要获取 daily_typing_pages 的平均值，那么你可以用如下命令来达到目的：

```
SQL> SELECT AVG(daily_typing_pages)
   -> FROM employee_tbl;
+-------------------------+
| AVG(daily_typing_pages) |
+-------------------------+
|               230.0000 |
+-------------------------+
1 row in set (0.03 sec)
```

你还可以使用 GROUP BY 子句来计算不同记录分组的平均值。下面的例子将会计算得出每个人平均值，你将能够得到平均每个人每天打的页数。

```
SQL> SELECT name, AVG(daily_typing_pages)
   -> FROM employee_tbl GROUP BY name;
+------+-------------------------+
| name | AVG(daily_typing_pages) |
+------+-------------------------+
| Jack |               135.0000 |
| Jill |               220.0000 |
| John |               250.0000 |
| Ram  |               220.0000 |
```

```
| Zara |              325.0000 |
+------+-----------------------+
5 rows in set (0.20 sec)
```

SUM 函数

SUM函数用于找出表中记录在某字段处的总和。

要理解 SUM 函数，请考虑 employee_tbl 表，表中记录如下所示：

```
SQL> SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|   1 | John | 2007-01-24 |              250 |
|   2 | Ram  | 2007-05-27 |              220 |
|   3 | Jack | 2007-05-06 |              170 |
|   3 | Jack | 2007-04-06 |              100 |
|   4 | Jill | 2007-04-06 |              220 |
|   5 | Zara | 2007-06-06 |              300 |
|   5 | Zara | 2007-02-06 |              350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

现在，假设你想要获取 daily_typing_pages 的总和，那么你可以用如下命令来达到目的：

```
SQL> SELECT SUM(daily_typing_pages)
   -> FROM employee_tbl;
+------------------------+
| SUM(daily_typing_pages) |
+------------------------+
|                   1610 |
+------------------------+
1 row in set (0.00 sec)
```

你还可以使用 GROUP BY 子句来得出不同记录分组的总和。下面的例子将会计算得出每个人的总和，，你将能够得到每个人每天打的总页数。

```
SQL> SELECT name, SUM(daily_typing_pages)
   -> FROM employee_tbl GROUP BY name;
```

SQRT 函数

SQRT 函数用于计算得出任何数值的平方根。你可以像下面这样使用 SELECT 语句计算任何数值的平方根：

```
SQL>  select SQRT(16);
+-----------+
| SQRT(16) |
+-----------+
| 4.000000 |
+-----------+
1 row in set (0.00 sec)
```

你在这里看到的是浮点数，因为 SQL 以浮点数类型来进行平方根的计算。

你还可以使用 SQRT 函数来计算表中记录的平方根。要获得对 SQRT 函数更深入的了解，请考虑 employee_tbl 表，表中记录如下所示：

```
SQL> SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|   1 | John | 2007-01-24 |              250 |
|   2 | Ram  | 2007-05-27 |              220 |
|   3 | Jack | 2007-05-06 |              170 |
|   3 | Jack | 2007-04-06 |              100 |
|   4 | Jill | 2007-04-06 |              220 |
|   5 | Zara | 2007-06-06 |              300 |
|   5 | Zara | 2007-02-06 |              350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

现在，假设你想要获取每个记录中 daily_typing_pages 的平方根，那么你可以用如下命令来达到目的：

```
SQL> SELECT name, SQRT(daily_typing_pages)
   -> FROM employee_tbl;
+------+--------------------------+
| name | SQRT(daily_typing_pages) |
+------+--------------------------+
| John |              15.811388 |
| Ram  |              14.832397 |
| Jack |              13.038405 |
| Jack |              10.000000 |
| Jill |              14.832397 |
```

```
| Zara |          17.320508 |
| Zara |          18.708287 |
+------+--------------------------+
7 rows in set (0.00 sec)
```

RAND 函数

SQL 有一个 RAND 函数，用于产生 0 至 1 之间的随机数：

```
SQL>  SELECT RAND( ), RAND( ), RAND( );
+------------------+-----------------+------------------+
| RAND( )          | RAND( )         | RAND( )          |
+------------------+-----------------+------------------+
| 0.45464584925645 | 0.1824410643265 | 0.54826780459682 |
+------------------+-----------------+------------------+
1 row in set (0.00 sec)
```

当以某个整数值作为参数来调用的时候，RAND() 会将该值作为随机数发生器的种子。对于每一个给定的种子，RAND() 函数都会产生一列可以复现的数字：

```
SQL>  SELECT RAND(1), RAND( ), RAND( );
+------------------+-----------------+------------------+
| RAND(1 )         | RAND( )         | RAND( )          |
+------------------+-----------------+------------------+
| 0.18109050223705 | 0.75023211143001 | 0.20788908117254 |
+------------------+-----------------+------------------+
1 row in set (0.00 sec)
```

你可以使用 ORDER BY RAND() 来对一组记录进行随机化排列，如下所示：

```
SQL> SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|   1 | John | 2007-01-24 |              250 |
|   2 | Ram  | 2007-05-27 |              220 |
|   3 | Jack | 2007-05-06 |              170 |
|   3 | Jack | 2007-04-06 |              100 |
|   4 | Jill | 2007-04-06 |              220 |
|   5 | Zara | 2007-06-06 |              300 |
|   5 | Zara | 2007-02-06 |              350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

现在，试试下面的命令：

```
SQL> SELECT * FROM employee_tbl ORDER BY RAND();
+------+------+------------+--------------------+
```

```
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|    5 | Zara | 2007-06-06 |                300 |
|    3 | Jack | 2007-04-06 |                100 |
|    3 | Jack | 2007-05-06 |                170 |
|    2 | Ram  | 2007-05-27 |                220 |
|    4 | Jill | 2007-04-06 |                220 |
|    5 | Zara | 2007-02-06 |                350 |
|    1 | John | 2007-01-24 |                250 |
+------+------+------------+--------------------+
7 rows in set (0.01 sec)

SQL> SELECT * FROM employee_tbl ORDER BY RAND();
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|    5 | Zara | 2007-02-06 |                350 |
|    2 | Ram  | 2007-05-27 |                220 |
|    3 | Jack | 2007-04-06 |                100 |
|    1 | John | 2007-01-24 |                250 |
|    4 | Jill | 2007-04-06 |                220 |
|    3 | Jack | 2007-05-06 |                170 |
|    5 | Zara | 2007-06-06 |                300 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

CONCAT 函数

CONCAT 函数用于将两个字符串连接为一个字符串，试一下下面这个例子：

```
SQL> SELECT CONCAT('FIRST ', 'SECOND');
+----------------------------+
| CONCAT('FIRST ', 'SECOND') |
+----------------------------+
| FIRST SECOND               |
+----------------------------+
1 row in set (0.00 sec)
```

要对 CONCAT 函数有更为深入的了解，请考虑 employee_tbl 表，表中记录如下所示：

```
SQL> SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|    1 | John | 2007-01-24 |                250 |
|    2 | Ram  | 2007-05-27 |                220 |
|    3 | Jack | 2007-05-06 |                170 |
|    3 | Jack | 2007-04-06 |                100 |
|    4 | Jill | 2007-04-06 |                220 |
|    5 | Zara | 2007-06-06 |                300 |
|    5 | Zara | 2007-02-06 |                350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

现在，假设你想要将上表中所有的姓名（name）、id和工作日（work_date）连接在一起，那么可以通过如下的命令来达到目的：

```
SQL> SELECT CONCAT(id, name, work_date)
    -> FROM employee_tbl;
+----------------------------+
| CONCAT(id, name, work_date) |
+----------------------------+
| 1John2007-01-24            |
| 2Ram2007-05-27             |
| 3Jack2007-05-06            |
| 3Jack2007-04-06            |
| 4Jill2007-04-06            |
| 5Zara2007-06-06            |
| 5Zara2007-02-06            |
```

```
+----------------------------+
7 rows in set (0.00 sec)
```

COUNT 函数

COUNT函数是 SQL 中最简单的函数了，对于统计由 SELECT 语句返回的记录非常有用。

要理解 COUNT 函数，请考虑 employee_tbl 表，表中的记录如下所示：

```
SQL> SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|   1 | John | 2007-01-24 |              250 |
|   2 | Ram  | 2007-05-27 |              220 |
|   3 | Jack | 2007-05-06 |              170 |
|   3 | Jack | 2007-04-06 |              100 |
|   4 | Jill | 2007-04-06 |              220 |
|   5 | Zara | 2007-06-06 |              300 |
|   5 | Zara | 2007-02-06 |              350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

现在，假设你想要统计上表中记录的总数，那么可以依如下所示步骤达到目的：

```
SQL>SELECT COUNT(*) FROM employee_tbl ;
+----------+
| COUNT(*) |
+----------+
|       7 |
+----------+
1 row in set (0.01 sec)
```

类似地，如果你想要统计 Zara 的数目，就可以像下面这样：

```
SQL>SELECT COUNT(*) FROM employee_tbl
    -> WHERE name="Zara";
+----------+
| COUNT(*) |
+----------+
|       2 |
+----------+
1 row in set (0.04 sec)
```

注意：所有的 SQL 查询都是不区分大小写的，因此在 WHERE 子句的条件中，ZARA 和 Zara 是没有任何区别的。

COUNT 函数

COUNT函数是 SQL 中最简单的函数了，对于统计由 SELECT 语句返回的记录非常有用。

要理解 COUNT 函数，请考虑 employee_tbl 表，表中的记录如下所示：

```
SQL> SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|   1 | John | 2007-01-24 |              250 |
|   2 | Ram  | 2007-05-27 |              220 |
|   3 | Jack | 2007-05-06 |              170 |
|   3 | Jack | 2007-04-06 |              100 |
|   4 | Jill | 2007-04-06 |              220 |
|   5 | Zara | 2007-06-06 |              300 |
|   5 | Zara | 2007-02-06 |              350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

现在，假设你想要统计上表中记录的总数，那么可以依如下所示步骤达到目的：

```
SQL>SELECT COUNT(*) FROM employee_tbl ;
+----------+
| COUNT(*) |
+----------+
|        7 |
+----------+
1 row in set (0.01 sec)
```

类似地，如果你想要统计 Zara 的数目，就可以像下面这样：

```
SQL>SELECT COUNT(*) FROM employee_tbl
    -> WHERE name="Zara";
+----------+
| COUNT(*) |
+----------+
|        2 |
+----------+
1 row in set (0.04 sec)
```

注意：所有的 SQL 查询都是不区分大小写的，因此在 WHERE 子句的条件中，ZARA 和 Zara 是没有任何区别的。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/sql/