创建数据库

```
CREATE DATABASE IF NOT EXISTS BarDrinkn
DEFAULT CHARACTER SET utf8mb4
DEFAULT COLLATE utf8mb4_unicode_ci;

show databases;
SHOW CHARACTER SET;
SHOW COLLATION;
```

创建关系模式（表）

```
Use bardrink;
CREATE TABLE Sells (
    bar CHAR(20),
    beer VARCHAR(20),
    price REAL,
    PRIMARY KEY (bar,beer)
 );
```

primary key / unique约束

UNIQUE可以有多个，可以出现空值，但不能出现多个空值；PRIMARY KEY只可以有一个，且不能为空

```
CREATE TABLE Sells (
    bar CHAR(20) UNIQUE,
    beer VARCHAR(20) UNIQUE,
    price REAL
);

CREATE TABLE Sells (
    bar CHAR(20),
    beer VARCHAR(20),
    price REAL,
    UNIQUE(bar,beer)
);

CREATE TABLE Sells (
    bar CHAR(20),
    beer VARCHAR(20),
    price REAL,
    PRIMARY KEY (bar, beer)
);
```

外键（mysql仅支持显式声明）

FOREIGN KEY 的属性一定加括号！

```
CREATE TABLE Sells (
    bar CHAR(20),
    beer CHAR(20),
    price REAL,
    FOREIGN KEY (beer) REFERENCES Beers(name)
);
```

或

```
CREATE TABLE Sells (
    bar CHAR(20),
    beer CHAR(20)  REFERENCES Beers(name),
    price REAL
);
```

级联

- **删除**
  - **直接删除**

```
CREATE TABLE Sells(
    bar   CHAR(20),
    beer  CHAR(20),
    price REAL,
    FOREIGN KEY(beer) REFERENCE Beers(name) ON DELETE CASCADE
    );
```

  - **置NULL**

```
CREATE TABLE Sells(
    bar   CHAR(20),
    beer  CHAR(20),
    price REAL,
    FOREIGN KEY(beer) REFERENCE Beers(name) ON DELETE SET NULL
    );
```

- **更新**
  - **直接更新**

```
CREATE TABLE Sells(
    bar   CHAR(20),
    beer  CHAR(20),
    price REAL,
    FOREIGN KEY(beer) REFERENCE Beers(name) ON UPDATE CASCADE
    );
```

  - **置NULL**

```
CREATE TABLE Sells(
    bar   CHAR(20),
    beer  CHAR(20),
    price REAL,
    FOREIGN KEY(beer) REFERENCE Beers(name) ON UPDATE SET NULL
    );
```

not null / default

```
CREATE TABLE Sells (
    bar CHAR(20),
    beer VARCHAR(20) DEFAULT 'HouseBeer',
    price REAL NOT NULL,
    PRIMARY KEY (bar,beer)
 );
```

check

```
ALTER TABLE Sells ADD CHECK (bar = 'Joe''s Bar' OR price <> 5.00); # <>表示不等于

CREATE TABLE Sells (
    bar CHAR(20),
    beer CHAR(20),
    price REAL,
    CHECK (bar = 'Joe''s Bar' OR price <= 5.00)
);
```

check的约束不如foreign key强

```
CREATE TABLE Sells (
    bar CHAR(20),
    beer CHAR(20) CHECK (beer IN (SELECT name FROM Beers)),
    price REAL CHECK (price <= 5.00)
);
```

该示例中，check仅在`Sells`表中的`beer`属性发生插入和更新时予以约束；但当`Beers`表中的元组发生删除或更新时不起任何作用。

Assertion（断言）

SQL 断言（Assertion）是一种用于强制实施数据库约束条件的机制。SQL 断言可以对数据库中的数据进行规则检查，并在不符合规则时防止插入、更新或删除操作。

```
CREATE ASSERTION price_check CHECK (NOT EXISTS (SELECT * FROM products WHERE price < 0));
```

定义了一个名为 price_check 的断言，它检查 products 表中是否存在价格小于 0 的记录。如果存在，则插入、更新或删除操作将被阻止。

```
DROP ASSERTION price_check;# 删除断言
```

View（视图）

在 SQL 中，视图（View）是一种虚拟的表，它是基于一个或多个表的查询结果，而不是实际存在的表。视图可以看作是一个预定义的 SELECT 语句，可以将复杂的查询语句封装成简单的视图，从而方便用户进行查询和数据操作。

视图并不是实际存在的表，它只是一个查询结果的映射，因此视图中的数据是动态的，它们随着原始表中数据的变化而变化。

```
CREATE VIEW 职员 AS
SELECT empno, projectno, enterdate FROM workson WHERE job='职员';

CREATE VIEW v_count(projectno, countproject) AS
SELECT projectno, COUNT(*) FROM workson GROUP BY projectno;

DROP VIEW v_count;
```

Index（索引）

在 SQL 中，索引（Index）是一种用于加快查询速度的数据结构，它可以大大提高数据库的查询效率。索引可以类比于书籍的目录，通过索引可以快速定位到需要查询的数据，而不必扫描整个数据表。

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

# 示例
CREATE INDEX i_empno ON workson(empno);
CREATE INDEX i_pjno_job ON workson(projectno,job);

DROP INDEX i_empno;
DROP INDEX i_pjno_job;
```

删除数据库/表/索引/触发器

```
DROP DATABASE 数据库名1{,数据库名2…...}
DROP TABLE 表名1{,表名2…...}
DROP INDEX
DROP TRIGGER
```

增/删 表的属性列

```
ALTER TABLE Bars ADD phone CHAR(16) DEFAULT 'unlisted';
ALTER TABLE Bars DROP COLUMN license;
```

查询

符号： =, <>, <, >, <=, >=, AND, OR, NOT

```
SELECT name as newname   # as用来为结果中的该列重命名，可以不写
FROM   Beers
WHERE manf = 'Anheuser-Busch'; # where manf in ('A co.','B co.')


SELECT bar, beer, price*120 as priceInYen #为price*120重命名为priceInYen
FROM Sells; # 没有where，返回所有元组


# 为结果关系表新建一列，使得该列在每行显示指定字符串
# 表Likes(drinker, beer)
SELECT drinker, 'likes Bud' AS whoLikesBud FROM Likes WHERE beer = 'Bud';


SELECT price FROM Sells WHERE bar = 'Joe''s Bar' AND beer = 'Bud'; # 单引号之间的引号为双引号

SELECT empname FROM employee WHERE deptno in (SELECT deptno FROM department WHERE location='天津' OR
location='北京'); # OR 表示选择工作地在天津或北京的雇员
```

- **Patterns**

```
SELECT *
FROM employee
WHERE empname like '李%'
#   % 代表任意长度字符串
    _ 代表一个字符
#   NOT LIKE 表反义
```

- **空值查询**

```
SELECT bar FROM Sells WHERE price IS NOT NULL;
```

等价于

```
SELECT bar FROM Sells WHERE price like '%';
```

- **输出排序**

```
# ASC (升序，默认值)，DESC (降序)
SELECT empname FROM Employee WHERE deptno='d2' ORDER BY empno DESC;
```

- **多表查询**
  - **连接查询**

```
SELECT * FROM employee, department WHERE location='天津'AND department.deptno=employee.deptno;
#  --------------------------------
SELECT * FROM employee NATURAL JOIN department WHERE location='天津';
```

  - **嵌套查询（先内后外）**

```
# Find the name and manufacturer of beers that Fred likes.
# Beers(name, manf); Likes(drinker, beer)
SELECT * FROM Beers WHERE name IN (SELECT beer FROM Likes WHERE drinker = 'Fred');
# NOT IN 表反义
```

  - **exists查询（先外后内）**

"EXISTS(relation)" is true iff the relation is nonempty.

```
# Find the beers that are the unique beer by their manufacturer.
SELECT name FROM Beers b1 WHERE NOT EXISTS (SELECT * FROM Beers WHERE manf = b1.manf AND name <>
b1.name);
```

```
# 查找部门中只有一名员工的员工姓名
# Employee(empno,empname,deptno)
SELECT empname FROM Employee e1 WHERE NOT EXISTS (SELECT * FROM Employee WHERE deptno=e1.deptno AND
empname <> b1.empname);
```

- **显式元组变量（自连接查询）**

```
# Find pairs of beers by the same manufacturer
SELECT b1.name, b2.name
FROM Beers b1, Beers b2
WHERE b1.manf = b2.manf AND b1.name < b2.name;
```

- **Any / ALL**

  x = ANY() is true iff x equals **at least one** tuple in the relation.

  x <> ALL() is true iff for **every** tuple t in the relation , x is not equal to t.

```
# Find the beer(s) sold for the highest price.
SELECT beer FROM Sells WHERE price >= ALL(SELECT price FROM Sells);

# Find the beer(s) not sold for the lowest price
SELECT beer FROM Sells WHERE price > ANY(SELECT price FROM Sells);
```

- **UNION（mysql没有交和差）、INTERSECT、EXCEPT 、DISTINCT**

```
#【并集】 UNION 操作符用于合并两个 SELECT 语句的结果集，并去除其中的重复行。
(SELECT * FROM Likes)
UNION
(SELECT drinker, beer FROM Sells, Frequents WHERE Frequents.bar = Sells.bar);

#【交集】 INTERSECT 操作符用于获取两个 SELECT 语句的结果集的交集，也会去除其中的重复行。

#【差集】 EXCEPT 操作符用于获取第一个 SELECT 语句的结果集中不在第二个 SELECT 语句的结果集中的行，也会去除其中的重复行。

SELECT DISTINCT price # 去重
FROM Sells;
```

- **Aggregations（sum, avg, min, max, count, and count(*) ）**
- **Grouping & Having**

  GROUP的优先级要低于笛卡尔积跟选择运算。

  HAVING 子句必须出现在 GROUP BY 子句之后，因为 HAVING 子句是用于筛选 GROUP BY 子句分组后的聚合结果，而 GROUP BY 子句则用于对查询结果进行分组。另外，HAVING 子句也可以使用聚合函数和比较运算符来进行过滤条件的筛选。

```
# Find, for each drinker, the average price of Bud at the bars they frequent
# Sells(bar, beer, price);
# Frequents(drinker, bar)
SELECT drinker, AVG(price) FROM Frequents, Sells WHERE beer='Bud' AND Frequents.bar = Sells.bar
GROUP BY drinker;

# Find the average price of those beers that are either served in at least 3 bars or manufactured by Busch.
# Beers(name, manf); Sells(bar, beer, price)
SELECT beer, AVG(price) FROM Sells GROUP BY beer HAVING COUNT(*) >= 3 OR beer IN (SELECT name FROM Beers WHERE manf = 'Busch');
```

插入

```
INSERT INTO Likes(drinker, beer) VALUES('Sally', 'Bud');

INSERT INTO PotBuddies (SELECT DISTINCT d2.drinker FROM Frequents d1, Frequents d2 WHERE d1.drinker =
'Sally' AND d2.drinker <> 'Sally' AND d1.bar = d2.bar);
```

删除

```
DELETE FROM Likes WHERE drinker = 'Sally' AND beer = 'Bud';

DELETE FROM Likes; #清空表
```

更改

```
UPDATE Drinkers SET phone = '555-1212' WHERE name = 'Fred';

UPDATE Sells SET price = 4.00 WHERE price > 4.00;
```

其他操作

触发器

```
# 插入操作，保证参照完整性
DELIMITER //
CREATE TRIGGER WorksonInsertTrig
BEFORE INSERT ON workson FOR EACH ROW
BEGIN
IF NEW.erpno NOT IN (SELECT empno FROM employee) THEN
INSERT INTO empleyee(empno) values(NEW.empno) ;
END IF;
END; //
DELIMITER ;

# 删除操作，保证参照完整性
DELIMITER //
CREATE TRIGGER EmployeeDeleteTrig
BEFORE DELETE ON employee FOR EACH ROW
BEGIN
IF old.empno IN (SELECT empno FROM workson) THEN
DELETE FROM workson WHERE empno=old.empno ;
END IF;
END; //
DELIMITER ;
```

Stored Procedure

在 SQL 中，存储过程（Stored Procedure）是一种预编译的 SQL 代码块，它可以接受参数并执行一系列的 SQL 语句。存储过程通常用于执行复杂的数据库操作，例如，将多个 SQL 语句组合为一个事务，减少网络通信次数，提高数据库性能等。

```
CREATE TABLE print(printInfo CHAR(50));
DELIMITER \\
CREATE PROCEDURE printProc(inout printInfo CHAR(50)) # CREATE PROCEDURE printProc()
BEGIN # CREATE TABLE print(printInfo CHAR(50))
IF (SELECT COUNT(*) FROM workson WHERE projectno='p1')>3 THEN SET printInfo='The number in the project p1 is 4or more';
ELSE SET printInfo='The number in the project p1 is less than 4';
END IF;
END; \\
DELIMITER ;
CALL printProc(@a);
SELECT @a;
```

Stored Function

在 SQL 中，存储函数（Stored Function）是一种预编译的 SQL 代码块，它可以接受参数并返回一个值。与存储过程类似，存储函数通常用于执行复杂的数据库操作，并且可以在 SQL 语句中作为一个表达式使用。

```
CREATE FUNCTION hello (s CHAR(20))
RETURNS CHAR(50) DETERMINISTIC # 函数返回值类型
BEGIN #存储函数执行的SQL语句
    RETURN CONCAT('Hello,',s,'!'); # 函数体
END;
SELECT hello('world');# 输出 Hello,world!
```

数据库管理权限

GRANT（授权）

在 SQL 中，GRANT 是一种用于授权的命令，它允许数据库管理员授予用户或角色访问数据库对象的权限，例如表、视图、存储过程等。通过 GRANT 命令，管理员可以授权用户或角色执行某些特定的操作，或者限制他们对某些对象的访问权限，从而保护数据库的安全性。其语法如下：

```
GRANT permission ON object TO user;

# 示例
GRANT SELECT, UPDATE(price) ON Sells TO Sally;
# Now Sally has the right to issue any query on Sells and can update the price component only.
```

GRANT OPTION（授权plus）

在 SQL 中，GRANT OPTION 是一种用于授权的特殊权限，它允许被授权的用户或角色在不需要管理员干预的情况下，授予其他用户或角色与自己拥有相同权限的访问权限。通俗地说，GRANT OPTION 允许被授权者“代表”管理员授权。其语法如下：

```
GRANT permission ON object TO user WITH GRANT OPTION;

# 示例
GRANT UPDATE ON Sells TO Sally WITH GRANT OPTION;
# Now Sally can not only update ant attribute of Sells, but can grant to others the privilege UPDATE ON Sells.
```

REVOKING（撤销授权）

在 SQL 中，REVOKE 是一种用于撤销权限的命令，它允许数据库管理员撤销之前授予的权限，保护数据库的安全性。其语法如下：

```
REVOKE permission ON object FROM user;

# 示例
REVOKE SELECT ON Customers FROM John;
```

REVOKING OPTION（撤销授权plus）

在 SQL 中，REVOKE OPTION 是一种用于撤销 GRANT WITH GRANT OPTION 授权的命令，它允许管理员撤销之前授予某个用户或角色的 GRANT WITH GRANT OPTION 权限，保护数据库的安全性。但必须声明是以CASCADE或是RESTRICT模式。

1. **CASCADE** Any grants made by a revoke are also not in force, no matter how far the privilege was passed.
2. **RESTRICT** If the privilege has been passed to others, the REVOKE fails as a warning that something else must be done to "chase the privilege down."

其语法如下：

```
REVOKE permission ON object FROM user CASCADE;
REVOKE permission ON object FROM user RESTRICT;

# 示例
REVOKE SELECT ON Customers FROM John;
```