1. 介绍SQL

SQL is used to deal with the relational data base.

1. SQL注释
2. 单行注释 —
3. 多行煮熟 /\* \*/
4. 对于SQL每句结尾都要加 “**;**”
5. 选择数据集:

USE xxxx

1. 选择某一个表的所有列

SELECT \*

选中特殊列: SELECT last\_name, points+10

FROM customers;

1. WHERE
2. 某列=什么条件 > , >=, <, <=,=,!=, <>
3. AND, OR(AND 运算级别比OR 高), NOT
4. WHERE state IN ('VA','FL','GA')
5. BETWEEN (WHERE points BETWEEN 1000 AND 3000)
6. LIKE(用于查找满足部分条件的情况，比如名字中包含b 的人: WHERE last\_name LIKE '%b%', %没有固定长度，\_ 表示只有一位, ‘\_y’表示只有两个字母且最后一位为y)
7. REGEXP:更简便的搜索。

WHERE last\_name LIKE ‘%field%’

WHERE last\_name REGEXP ‘field’ $表示以什么结尾 ’field$’表示筛选”field”结尾的名字。’A | B’查找含有A或者B ， “^A”以A开头。 [gim]e代表ge, ie, me

-- ^ beginning

-- $ end

-- | logical or

-- [abcd]

--[a-f]

g. 寻找NULL值: WHERE phone IS NULL

1. ORDER BY first\_name DESC排序
2. 换名字用AS

SELECT

(points+10) \* 100 AS discount\_factor

1. SELECT DISTINCT state
2. FROM customers
3. 寻找NULL值: WHERE phone IS NULL
4. LIMIT语句:

LIMIT 3前三行

LIMIT 6,3 (6为偏移量)

第三章 连接分为内连接和外连接

SELECT \*

FROM orders

JOIN customers

ON orders.customer\_id = customers.customer\_id

1. JOIN

JOIN customers

ON orders.customer\_id = customers.customer\_id

2. 当多个表都含有同一列时，须指明具体的表 orders.customer\_id

3. 使用缩写:

FROM orders o

JOIN customers c

4. 连接不同数据库的数据：表前加前缀即可

5. 自连接:每个表都要加前缀

6. 多表连接:多用连JOIN ON

7. 复合连接条件: 在表里设置时，黄色开头为主键，有主键的时候可以与别的进行复合连接。

ON oi.order\_id = oin.order\_id

AND oi.product\_id = oin.product\_id

8. 隐式连接语法:

SELECT \*

FROM orders o, customers c

WHERE o.customer\_id = c.customer\_id

最好不用隐式连接语法

9. 外连接: 外连接分为左连接和右连接

使用左连接时，左表的所有项目全部返回，不管条件对错

尽量用左连接，不要用右连接。

10. USING使用条件:

ON o.customer\_id = c.customer\_id

USING (customer\_id)

使用条件: 两个表的名称一样才可以使用

USING (order\_id, product\_id)

11. NATURAL JOIN让计算机自己连接，不推荐

12. 交叉连接:

CROSS JOIN 将两个表匹配

1. UNION:

SELECT

order\_id,

order\_date,

'Active' AS status

FROM orders

WHERE order\_date >= '2019-01-01'

UNION

SELECT

order\_id,

order\_date,

'Archived' AS status

FROM orders

WHERE order\_date < '2019-01-01'

四、第四章

1. 类型varchar是指长度不确定的字符串，如果是char(5)必须有5个字符。

PK primary key

NN: 是否可以为null值，打钩意味着不能为空

AI：自动递增

2. 插入单行

INSERT INTO customers(

first\_name,

last\_name,

birth\_date,

address,

city,

state)

VALUES (

'John',

'Smith',

'1990-01-01',

'address',

'city',

'CA')

3. 插入多行

Values (‘1’),

(‘2’),

4. 插入分层行:

INSERT INTO order\_items

VALUES(last\_insert\_id(),1, 1,2.9)

5.创建表格:

CREATE TABLE orders\_archived AS

SELECT \*

FROM orders

···6. 插入新的表格

INSERT INTO orders\_archived

SELECT \*

FROM orders

WHERE order\_date < "2019-01-01"

7. UPDATE更新一行

UPDATE invoices

SET payment\_total = invoice\_total \*0.5,

payment\_date = due\_date

WHERE invoice\_id = 3

8.更新多行(需要在preferences取消safe)

UPDATE invoices

SET payment\_total = invoice\_total \*0.5,

payment\_date = due\_date

WHERE client\_id IN (3,4)

9. 在UPDATE中使用子查询

UPDATE invoices

SET payment\_total = invoice\_total \*0.5,

payment\_date = due\_date

WHERE client\_id IN

(SELECT client\_id

FROM clients

WHERE state in ('CA','NY'))

10 删除某行

DELETE FROM invoices

WHERE xxxx

五、第五章

1. 聚合函数

MAX()

MIN()

AVG()

SUM(a \* B)

count(payment\_date) AS count\_of\_payments 计算的都是非空值

COUNT(\*) 代表计算所有行.

2. GROUP BY 分类

SELECT

client\_id,

SUM(invoice\_total) AS total\_sales

FROM invoices

GROUP BY client\_id

ORDER BY total\_sales DESC

3. 语句顺序:

SELECT

client\_id,

SUM(invoice\_total) AS total\_sales

FROM invoices

WHERE invoice\_date >= '2019-07-01'

GROUP BY client\_id

ORDER BY total\_sales DESC

选择多行条件

GROUP BY A , B

4. 分类完之后的筛选用HAVING

SELECT

client\_id,

SUM(invoice\_total) AS total\_sales

FROM invoices

WHERE invoice\_date >= '2019-07-01'

GROUP BY client\_id

HAVING total\_sales > 500 WITH ROLLUP

**(WITH ROLLUP在GROUP BY 之后计算总数)**

ORDER BY total\_sales DESC

**总结：**

1. **WHERE是在分组前筛选数据，HAVING在分组之后筛选数据。**
2. **WHERE不管有没有在SELECT中选择都可以选这列，HAVING必须在SELECT中选中才能使用。HAVING要用原来的名字例如COUNT(\*) AS num Having COUNT(\*) > 1**

六、第六章

1. NOT IN

2. ALL 所有值

SELECT \*

FROM invoices

WHERE invoice\_total > ALL (

SELECT invoice\_total

FROM invoices

WHERE client\_id = 3

)

1. ANY/SOME 任意值

SELECT \*

FROM invoices

WHERE invoice\_total > ANY/SOME (

SELECT invoice\_total

FROM invoices

WHERE client\_id = 3

)

**4. 相关子查询：**

**-- Select employees whose salary is**

**-- above the avergae in their office**

**USE sql\_hr;**

**SELECT \***

**FROM employees e**

**WHERE salary > (**

**SELECT AVG(salary)**

**FROM employees**

**WHERE office\_id = e.office\_id**

**)**

1. **EXISTS(提高效率), 在IN里面生成很大的数据集，来挑，在EXISTS是看是否满足条件选取TRUE or FALSE 来筛选条件。**

**USE sql\_store;**

**SELECT \***

**FROM products p**

**WHERE NOT EXISTS(**

**SELECT product\_id**

**FROM order\_items**

**WHERE product\_id = p.product\_id**

**)**

1. **SELECT中可以再次套用SELECT**

**AS重新命名之后的函数要用SELECT才能选择  
SELECT**

**invoice\_id,**

**invoice\_total,**

**(SELECT AVG(invoice\_total)**

**FROM invoices) AS invoice\_average,**

**invoice\_total - (SELECT invoice\_average) AS difference**

**FROM invoices**

1. **可以在选择语句的FROM子句中写子查询**

七、第七章

1. 数值函数

-- 四舍五入

SELECT ROUND(5.73,1);

-- 截断数字

SELECT TRUNCATE(5.7345,2);

-- 取上届

SELECT CEILING(5.2)

ABS(x)

FLOOR(x)

RAND()生成0-1的随机数

2. 字符型函数

-- SELECT LENGTH('sky')

-- 大写

-- SELECT UPPER('sky')

-- 小写

-- SELECT LOWER('sky')

-- 删除空格

-- SELECT LTRIM(' sky')

-- SELECT RTRIM('sky ')

-- SELECT TRIM(' sky')

-- 得到几个字母

-- SELECT LEFT('Kindergarten',4)

-- SELECT RIGHT('Kindergarten',4)

-- 第二个数字为起始位置,第三个参数为长度

-- SELECT SUBSTRING('Kindergarten',2,4)

-- 第一个参数为想要查询的参数

-- SELECT LOCATE('garten','Kindergarten')

-- 替换字符串

-- SELECT REPLACE('Kindergarten','garten','garden')

-- 连接字符串

-- SELECT CONCAT('A','B')

================================================

3. 时间函数

-- SELECT YEAR(NOW()),CURDATE(),CURTIME()

-- 得到星期

-- SELECT DAYNAME(NOW())

-- can be used for other sql languages

-- SELECT EXTRACT(DAY FROM NOW())

SELECT \*

FROM orders

WHERE YEAR(order\_date) = 2019

SELECT DATE\_FORMAT(NOW(),'%M %d %Y')

4. **IFNULL 返回所有的空值，COALESCE先看shipper\_id，若shipper\_id为空，在看comments是否为空，可以传输多个值，均为空时才替换**

SELECT

order\_id,

-- IFNULL(shipper\_id,'Not assigned') AS shipper

COALESCE(shipper\_id, comments,'Not assigned') AS shipper

FROM orders

5. IF函数

IF(条件，结果1，否则结果2)

IF(

YEAR(order\_date)=2019,

'Active',

'Archived') AS category

1. CASE函数

CASE

WHEN YEAR(order\_date) = 2019 THEN 'Active'

WHEN YEAR(order\_date) = 2019-1 THEN 'Last Year'

WHEN YEAR(order\_date) < 2018 THEN 'Archived'

ELSE 'Future'

END AS category

八、第八章-View视图

一个表包含数据，一个视图只是一个SELECT语句，它已保存在数据库中。视图的优点是它可以连接来自多个表的数据，从而创建它的新视图。假设您有一个带工资的数据库，您需要对其进行一些复杂的统计查询。

您可以将查询保存为视图，然后SELECT \* FROM view，而不是将复杂查询始终发送到数据库。

1. 创建视图:

USE sql\_invoicing;

CREATE VIEW clients\_balance AS

SELECT

c.client\_id,

c.name,

SUM(invoice\_total - payment\_total) AS balance

FROM clients c

JOIN invoices i

USING (client\_id)

GROUP BY client\_id

1. 删除视图

DROP VIEW sales\_by\_client

1. 修改视图

CREATE OR REPLACE VIEW sales\_by\_client AS

SELECT

c.client\_id,

c.name,

SUM(invoice\_total) AS total\_sales

FROM clients c

JOIN invoices i USING (client\_id)

GROUP BY client\_id, name

1. 当视图可更新时，可以删除列，行，更新视图

DELETE FROM invoices\_with\_balance

WHERE invoice\_id = 1

1. **当更新视图时，若一个操作可能导致行的消失，加上WITH CHECK OPTION后即报错。**

CREATE OR REPLACE VIEW sales\_by\_client AS

SELECT

c.client\_id,

c.name,

SUM(invoice\_total) AS total\_sales

FROM clients c

JOIN invoices i USING (client\_id)

GROUP BY client\_id, name

**WITH CHECK OPTION**

1. **VIEW的好处: (1) simplify queries; (2) reduce the impact of changes; (3) restrict access to the data**

**九、第九章**

**1. 储存数据 (1)更好储存和管理sql；(2) 更快的执行; (3)数据安全**

**2. create a store procedure：**

**-- DELIMITER 自己设置符号,新的分隔符;把这些语句当做一个整体！**

**USE sql\_invoicing;**

**DELIMITER $$**

**CREATE PROCEDURE get\_clients()**

**BEGIN**

**SELECT \* FROM clients;**

**END$$**

**DELIMITER ;**

**CALL sql\_invoicing.get\_clients();**

**可以右键stored procedures create xxx然后只用在里面修改就好**

**3. 删除procedure：**

**DROP PROCEDURE IF EXISTS get\_clients**

**4. 传入参数(参数必填，不然会报错)**

**DROP PROCEDURE IF EXISTS get\_clients\_by\_state;**

**DELIMITER $$**

**CREATE PROCEDURE get\_clients\_by\_state**

**(**

**-- String 2 VARCHAR不知道固定的长度**

**state CHAR(2)**

**)**

**BEGIN**

**SELECT \* FROM clients c**

**WHERE c.state = state;**

**END$$**

**DELIMITER ;**

**5. 传入默认值:**

**DELIMITER $$**

**CREATE PROCEDURE get\_clients\_by\_state**

**(**

**state CHAR(2)**

**)**

**BEGIN**

**IF state IS NULL THEN**

**SET state = 'CA';**

**END IF;**

**SELECT \* FROM clients c**

**WHERE c.state = state;**

**END$$**

**DELIMITER ;**

**BEGIN**

**IF state IS NULL THEN**

**SELECT \* FROM clients;**

**ELSE**

**SELECT \* FROM clients c**

**WHERE c.state = state;**

**END IF;**

**END$$**

**可以被替换成：**

**BEGIN**

**SELECT \* FROM clients c**

**WHERE c.state = IFNULL(state , c.state);**

**END$$**

1. **参数验证**

**IF payment\_amount <= 0 THEN**

**SIGNAL SQLSTATE '22003'**

* **22003是错误之一**

**SET MESSAGE\_TEXT = 'Invalid payment amount';**

**END IF;**

**UPDATE invoice i**

**SET**

**i.payment\_total = payment\_amount,**

**i.payment\_date = payment\_date**

**WHERE i.invoice\_id = invoice\_id;**

1. **输出参数**

**@参数名——初始化参数**

**CREATE DEFINER=`root`@`localhost` PROCEDURE `get\_unpaid\_invoices\_for\_client`(**

**client\_id INT,**

**OUT invoices\_count INT,**

**OUT invoices\_total DECIMAL(9,2)**

**)**

**BEGIN**

**SELECT COUNT(\*), SUM(invoice\_total)**

**INTO invoices\_count,invoices\_total**

**FROM invoices i**

**WHERE i.client\_id = client\_id**

**AND payment\_total = 0;**

**END**

1. **设置变量**

**SET @invoices\_count = 0**

**-- User or session variables**

**-- Local variable 仅仅在执行过程中有用**

**-- DECIMAL(9,2) 最多9位数,有2位小数**

**CREATE PROCEDURE get\_risk\_factor ()**

**BEGIN**

**DECLARE risk\_factor DECIMAL(9,2) DEFAULT 0;**

**DECLARE invoices\_total DECIMAL(9,2);**

**DECLARE invoices\_count INT;**

**SELECT COUNT(\*),SUM(invoice\_total)**

**INTO invoices\_count,invoices\_total**

**FROM invoices;**

**SET risk\_factor = invoices\_total / invoices\_count \* 5;**

**select risk\_factor;**

**END**

1. **创建函数（返回单一值）**

**CREATE FUNCTION get\_risk\_factor\_for\_client**

**(**

**client\_id INT**

**)**

**RETURNS INTEGER**

**-- 设置属性确定性，给定同样的一组值，返回相同的值**

**-- DETERMINISTIC**

**-- 函数配置选择语句，用以读取一些数据**

**READS SQL DATA**

**-- 函数有插入、更新或者删除**

**-- MODIFIES SQL DATA**

**BEGIN**

**DECLARE risk\_factor DECIMAL(9,2) DEFAULT 0;**

**DECLARE invoices\_total DECIMAL(9,2);**

**DECLARE invoices\_count INT;**

**SELECT COUNT(\*),SUM(invoice\_total)**

**INTO invoices\_count,invoices\_total**

**FROM invoices i**

**WHERE i.client\_id = client\_id;**

**SET risk\_factor = invoices\_total / invoices\_count \* 5;**

**RETURN IFNULL(risk\_factor,0);**

**END**

**\*/**

**SELECT**

**client\_id,**

**name,**

**get\_risk\_factor\_for\_client(client\_id) AS risk\_factor**

**FROM clients**

**十、第十章**

**1. 创建触发器**

**trigger: 在插入、更新和删除语句前后自动执行的一堆sql代码**

**DELIMITER $$**

**-- 可以修改除了这张表以外的任意表**

**-- 表明+ after/before+动作(insert/delete/update)**

**CREATE TRIGGER payments\_after\_insert**

**AFTER INSERT ON payments**

**-- 每行都会触发**

**FOR EACH ROW**

**BEGIN**

**UPDATE invoices**

**-- NEW (返回刚刚插入的行)/OLD(更新或者删除行的时候很有用，返回更新前的行以及对应的数值)**

**SET payment\_total = payment\_total + NEW.amount**

**WHERE invoice\_id = NEW.invoice\_id;**

**END $$**

**DELIMITER ;**

**use sql\_invoicing;**

**INSERT INTO payments**

**VALUES (DEFAULT, 5,3,'2019-01-01',10,1)**

**2. 查看触发器**

**-- 查看trigger**

**SHOW TRIGGERS**

**筛选triggers**

**SHOW TRIGGERS LIKE 'payments%'**

**3. 删除触发器**

**DROP TRIGGER IF EXISTS payments\_after\_insert**

**4. 审计表:**

**记录每一步操作**

**DELIMITER $$**

**DROP TRIGGER IF EXISTS payments\_after\_delete;**

**CREATE TRIGGER payments\_after\_delete**

**AFTER DELETE ON payments**

**FOR EACH ROW**

**BEGIN**

**UPDATE invoices**

**SET payment\_total = payment\_total - OLD.amount**

**WHERE invoice\_id = OLD.invoice\_id;**

**INSERT INTO payments\_audit**

**VALUES (OLD.client\_id, OLD.date,OLD.amount,'DELETE',NOW());**

**END $$**

**DELIMITER ;**

**5. 创建事件**

**-- 事件 task 是根据计划执行的任务或一堆SQL代码**

**-- SHOW VARIABLES LIKE 'event%';**

**-- SET GLOBAL event\_scheduler = ON**

**DELIMITER $$**

**CREATE EVENT yearly\_delete\_sale\_audit\_rows**

**ON SCHEDULE**

**-- AT '2019-01-10' 定期做一次**

**-- EVERY 1 HOUR**

**EVERY 1 YEAR STARTS '2019-01-01' ENDS '2029-01-01'**

**DO BEGIN**

**DELETE FROM payments\_audit**

**WHERE action\_date < NOW() - INTERVAL 1 YEAR;**

**END $$**

**DELIMITER ;**

**6. SHOW EVENTS LIKE 'yearly%';**

**-- 删除**

**DROP EVENT IF EXISTS yearly\_delete\_sale\_audit\_rows;**

**-- 改变**

**-- 把CREATE 改成ALTER DISABLE**

**ALTER EVENT yearly\_delete\_sale\_audit\_rows ENABLE;**

1. **事务(transactions): A group of SQL statements that represent a single unit of work.**

**同时执行，保证一致ACID**

1. **原子性: 每一个事务就是一个工作单元(要么全部执行，要么退回去所有更改被撤销)**
2. **一致性: 保证每个表对应内容一致**
3. **隔离性: 互不干扰**
4. **持久性: 事务产生的反应是持久的**
5. **创建事务:**

**USE sql\_store;**

**START TRANSACTION;**

**INSERT INTO orders (customer\_id, order\_date,status)**

**VALUES (1,'2019-01-01',1);**

**INSERT INTO order\_items**

**VALUES(LAST\_INSERT\_ID(),1,1,1);**

**-- close the statement**

**-- COMMIT;**

**-- 检查语句 退回事务并撤销所有更改**

**-- ROLLBACK;**

**3. 并发和锁定**

**当一个用户修改其他用户正在检索或修改的数据时，并发可能会成为一个问题。**

**一般情况下你执行一个事务时，这行会被锁定，其他的事务会等这个执行完再执行。**

1. **并发问题**

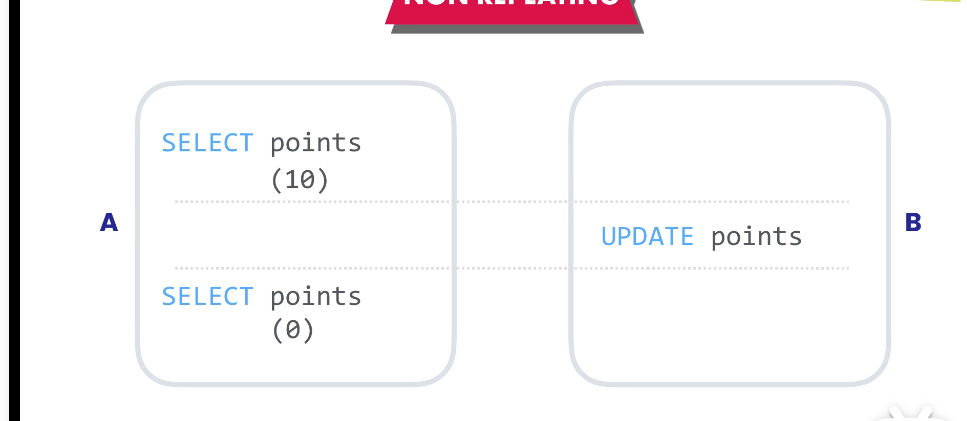
**Lost Updates: 当两个事务尝试更新相同的数据并且没有上锁时，就会发生这种情况。较晚提交的事务会覆盖较早事务做的更改。使用锁**

**Dirty Reads: 当一个事务读取了尚未被提交的数据。使用隔离技巧**

1. **READ COMMITTED**

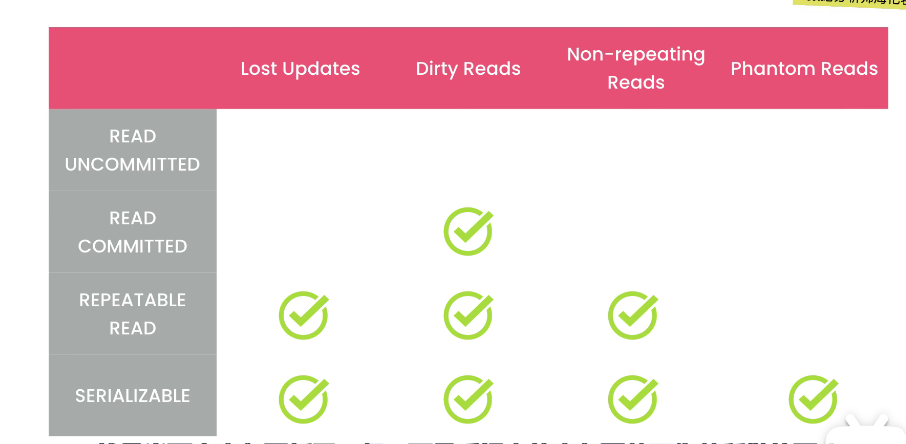
**Non-repeating Reads: 增强隔离性, 在事务中读取了相同的数据两次，但得到了不同的结果**

**(2)repeatable read**

****

**Phantom Reads(幻取): 在执行查询后才添加、更新或删除的(3) serializable序列化**

****

1. ****

**READ COMMITTED：只能读已经提交了的事务。**

**repeatable read**

**隔离级别越高，会存在越重的性能和可拓展性问题，一般默认是repeatable read**

1. **读未提交隔离级别**
2. **死锁：两个事务都在等待对方完成，永远完不成。**

**检查事务中两句指令的顺序，否则容易出现死锁。只能降低死锁的概率。**

**第十二章 数据类型**

1. **字符串类型：**

**CHAR(√): 固定长度**

**VARCHAR(√):可变长度; VARCHAR(50): short strings; VARCHAR(255): medium-length strings;**

**MEDIUMTEXT: max:16MB**

**LONGTEXT: max: 4GB**

**TINYTEXT: max:255**

**TEXT: max:64kb**

**打钩的可以使用索引**

1. **数值型**
2. **Integer Type：**

**TINYINT: 1b [-128,127]**

**UNSIGNED TINYINT: [0,255] (避免负数)**

**SAMLLINT: 2b**

**MEDIUMINT：3b**

**INT:4b**

**BIGINT:8b**

**ZEROFILL:用0覆盖数值的情况下很有用，能拥有一样的数位。**

**INT(4) = 0001**

1. **Rationals:**

**DECIMAL(p,s):精度和小数位，DECIMAL(9,2)代表储存9位数字，小数点后2位。**

**FLOAT 4b 精度不怎么样**

**DOUBLE8b 精度不怎么重要**

1. **Booleans:**

**BOOL**

**BOOLEAN**

**UPDATE posts**

**SET is\_published = TRUE (1)/FALSE(0)**

1. **ENUMS(布尔型) ENUM(‘small’,’medium’,’big’)不好用**

**SET（）**

**不可重复、修改麻烦，替换方法是创建一张查询表**

1. **Date and Time Types：**

**DATE:**

**TIME:**

**DATETIME: 8b**

**TIMESTAMP: 4b (up to 2038)**

**YEAR:**

1. **Blob types：存储大型二进制数据，图像、视频、PDF、word文件**

**TINYBLOB**

**BLOB**

**MEDIUMBLOB**

**LONGBLOB**

1. **Spatial Types: 存储几何或者地区值的空间类型**
2. **JSON: Lightweight format for storing and transferring data over the Internet.**

**JSON:**

**{**

**“key”:value**

**}**

**--创建jason**

**USE sql\_store;**

**UPDATE products**

**SET properties = '{**

**"dimensions":[1,2,3],**

**"weight":10,**

**"manufacturer":{"name":"sony"}**

**}**

**'**

**WHERE product\_id =1**

**-- 调用函数法 创建JSON**

**UPDATE products**

**SET properties = JSON\_OBJECT(**

**'weight',10,**

**'dimensions',JSON\_ARRAY(1,2,3),**

**"manufacturer",JSON\_OBJECT('name','sony')**

**)**

**WHERE product\_id =1**

1. **-- 读取JSON文件**

**SELECT**

**product\_id,**

**-- $ 表示当前json文件 '.'访问单独的属性或者键**

**JSON\_EXTRACT(properties, '$.weight') AS weight**

**-- -> 列路径运算符**

**properties -> '$.dimensions[0]',**

**-- ->>可以获得内容，不用字符串的双引号**

**properties ->> '$.manufacturer.name'**

**FROM products**

**WHERE properties ->> '$.manufacturer.name' = 'sony';**

1. **-- 修改JSON**

**UPDATE products**

**SET properties = JSON\_SET(**

**properties,**

**-- 添加一个或多个键值到这个对象里**

**'$.weight',20,**

**'$.age',10**

**)**

**WHERE product\_id =1**

1. **删除JSON**

**UPDATE products**

**SET properties = JSON\_REMOVE(**

**properties,**

**-- 删除**

**'$.age'**

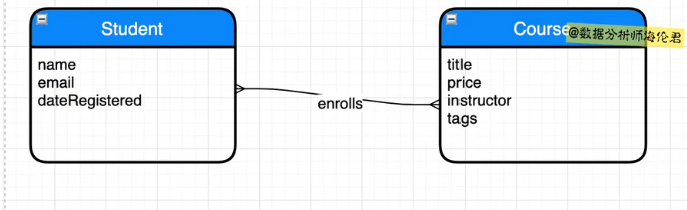
**)**

第十三章 设计数据库

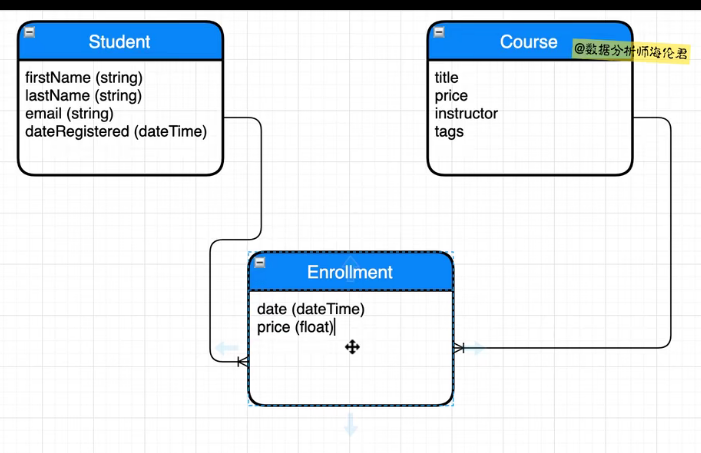
1. Understand the requirements 理解和分析业务需求
2. Build a conceptual model 构建业务的概念模型:识别业务中的实体、事物或概念以及它们之间的关系

**Entity Relationship**实体关系图/UML标准建模语言

Microsoft visio/draw.io/LucidCharts



1. Build a logical model: (尽量拆分，把它分成更小的部分，比如街道、城市、邮政编码和国家等等)



1. Build a physical model: 表主键、视图、存储对象、触发器等等

File->new model->EER: 增强实体关系(enhance entity relationship)

右键edit scheme

1. 主键:是唯一标识给定表里每条记录的列

First\_name,last\_name组合起来就叫复合主键，但是名字可能相同、email可能修改、但是主键不可更改，导入一个学生\_id来当主键，AI表示递增

关系只有1对1、1对多、要实现多对多需要链接表

1. 外键: 加入关系的一段、关系的一端称为父表或主键表、另一端称为子表或外键表。

外键是一张表中引用了另一张表主键的那列。

1. 外键约束:

On Update/Delete:

CASCADE: 级联，即主键改变，MySQL会自动更新子表中的记录

RESTRICT: 拒绝更新

SET NULL: 主键改变，外键值就会被设置成空值，孤儿记录

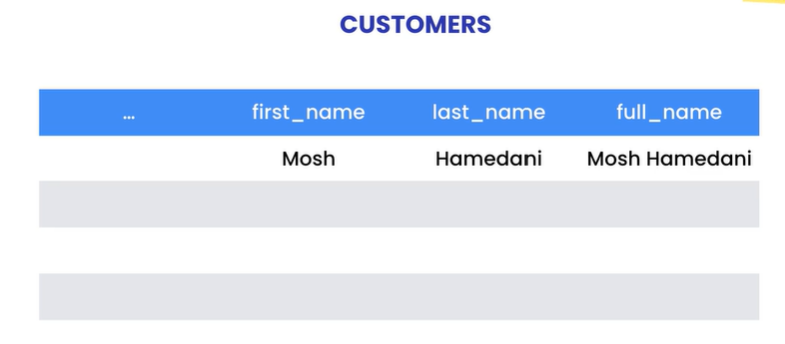
1. 标准化: 审查我们的设计，并确保它遵循一些防止数据重复的预定义规则。

遵循三条范式

1st-Normal Form: 一行中的每个单元都应该有单一值，且不能出现重复列。

2nd-Normal Form:每一张表只能代表一种且仅有一种实体类型。表中的每一列都应该用来描述那个实体。

3rd-Normal Form: 表中的列不应派生自其他列。



1. Solve today’s problems
2. 模型的正向工程: database-forward xxxx
3. 数据库同步模型 database-synchronize model
4. 模型的逆向工程: 确保原来的模型已经关闭再进行逆向的操作。Reverse engineer
5. 创建数据库

CREATE DATABASE IF NOT EXISTS sql\_store

DROP DATABASE IF EXISTS sql\_store

1. 创建表

CREATE TABLE customers

(customer\_id INT primary key, NOT NULL, …,

XXXXXXXXXXXX,

)

1. 更改表

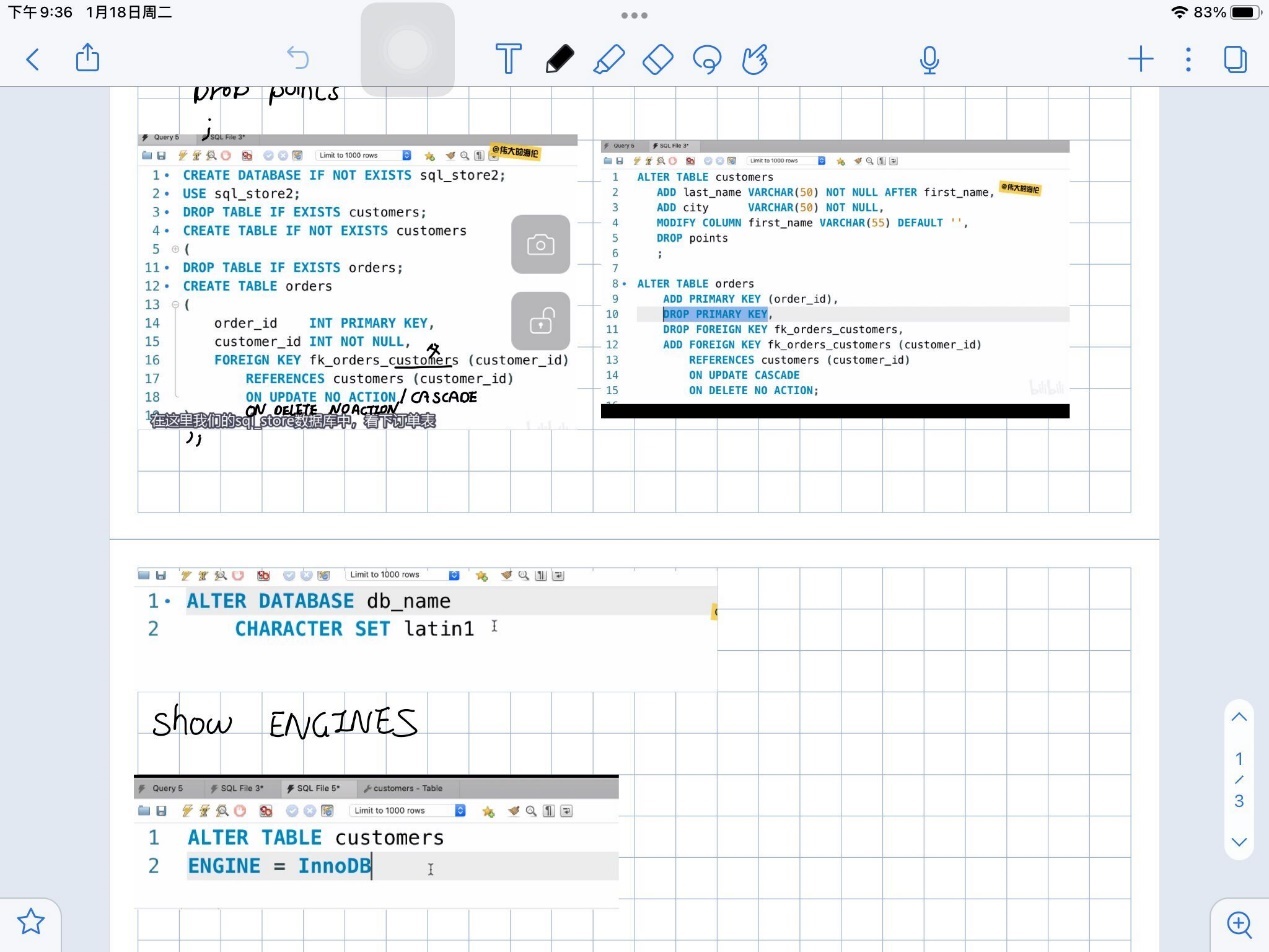
ALTER TABLE customers

ADD last\_name VARCHAR(50) ………,

MODIFY COLUMN

DROP points

;



**第十四章**

1. **索引: 索引本质上是数据库引擎用来快速查找数据的数据结构。**
2. **Cost of indexes: (1) increase the database; (2) slow down the writes; (3) design indexes based on your queries, not your tables.**
3. **创建索引**

**--EXPLAIN可以看到过程**

**EXPLAIN SELECT customer\_id**

**FROM customers**

**WHERE state = 'CA';**

**CREATE INDEX idx\_state ON customers(state);**

1. **查看索引**

**SHOW INDEXES IN customers;**

**ANALYZE TABLE customers;**

**外键自动为二级索引**

1. **前缀索引（索引字符串列）**

**-- 前缀索引 处理字符串**

**CREATE INDEX idx\_lastname ON customers(lastname(20));**

**-- 选择前缀长度**

**SELECT**

**COUNT(DISTINCT LEFT(last\_name,1)),**

**COUNT(DISTINCT LEFT(last\_name,5)),**

**COUNT(DISTINCT LEFT(last\_name,10))**

**FROM customers;**

1. **全文索引**

**-- 全文索引**

**USE sql\_blog;**

**SELECT \***

**FROM posts**

**WHERE title LIKE '%react redux%' OR**

**body LIKE '%react redux%';**

**CREATE FULLTEXT INDEX idx\_title\_body ON posts(title, body);**

**SELECT \*, MATCH(title, body) AGAINST('react redux')**

**FROM posts**

**-- WHERE MATCH(title, body) AGAINST('react redux');**

**-- 布尔模式/ - 表示不含redux + 表示必须包含 """handing a form""" 表示强制包含**

**WHERE MATCH(title, body) AGAINST('react -redux +form' IN BOOLEAN MODE);**

1. **复合索引 效率更高。**

**USE sql\_store;**

**DROP INDEX idx\_state ON customers;**

**CREATE INDEX idx\_state\_points ON customers(state, points);**

**EXPLAIN SELECT customer\_id FROM customers**

**WHERE state = 'CA' AND points > 1000;**

**复合索引中的列顺序**

1. **把最常用的列放在前面**
2. **把基数更高的列排在前面 (DISTINCT之后的数量) 还是要看情况。**
3. **Take your queries into account**
4. **索引无效**
5. **-- 当索引无效时 用OR时替换成UNION**

**CREATE INDEX idx\_points ON customers(points);**

**EXPLAIN**

**SELECT customer\_id FROM customers**

**WHERE state = 'CA'**

**UNION**

**SELECT customer\_id FROM customers**

**WHERE points > 1000;**

1. **EXPLAIN**

**SELECT customer\_id FROM customers**

**WHERE points + 10 > 2010**

**要换成: WHERE points > 2000**

**把列单独提出来**

1. **使用索引进行排序**

**-- (a,b)**

**-- a**

**-- 不能单独用b否则会成倍增加**

**-- a,b**

**-- a DESC b DESC**

**10. 覆盖索引**

**11. 维护索引 总是检查已有的索引。删除多余没用的索引。**

**第15章**

1. **创建一个用户**

**-- 创建新用户**

**CREATE USER john@ + ip地址**

**-- 从网站随意登录**

**CREATE USER john@ '%.codewithmosh.com'**

**-- 设置密码**

**CREATE USER john IDENTIFIED BY '1234'**

1. **查看用户**

**-- 查看用户**

**SELECT \* FROM mysql.user;**

1. **删除用户**

**DROP USER** [**bob@codewithmosh.com**](mailto:bob@codewithmosh.com)**;**

1. **更改密码**

**-- 更改密码**

**SET PASSWORD FOR john = '1234';**

**SET PASSWORD = '1234';**

1. **授予权限**

**-- 授予权限**

**-- 1: web/desktop application**

**CREATE USER moon\_app IDENTIFIED BY '1234';**

**GRANT SELECT, INSERT, UPDATE, DELETE,EXECUTE**

**--\*代表全部的表**

**ON sql\_store.\***

**TO moon\_app;**

**-- 2: admin**

**GRANT ALL**

**ON \*.\***

**TO john;**

1. **查看权限**

**SHOW GRANTS FOR john**

1. **取消权限**

**REVOKE CREATE VIEW**

**ON sql\_store.\***

**FROM moon\_app;**