

NEW PLAN

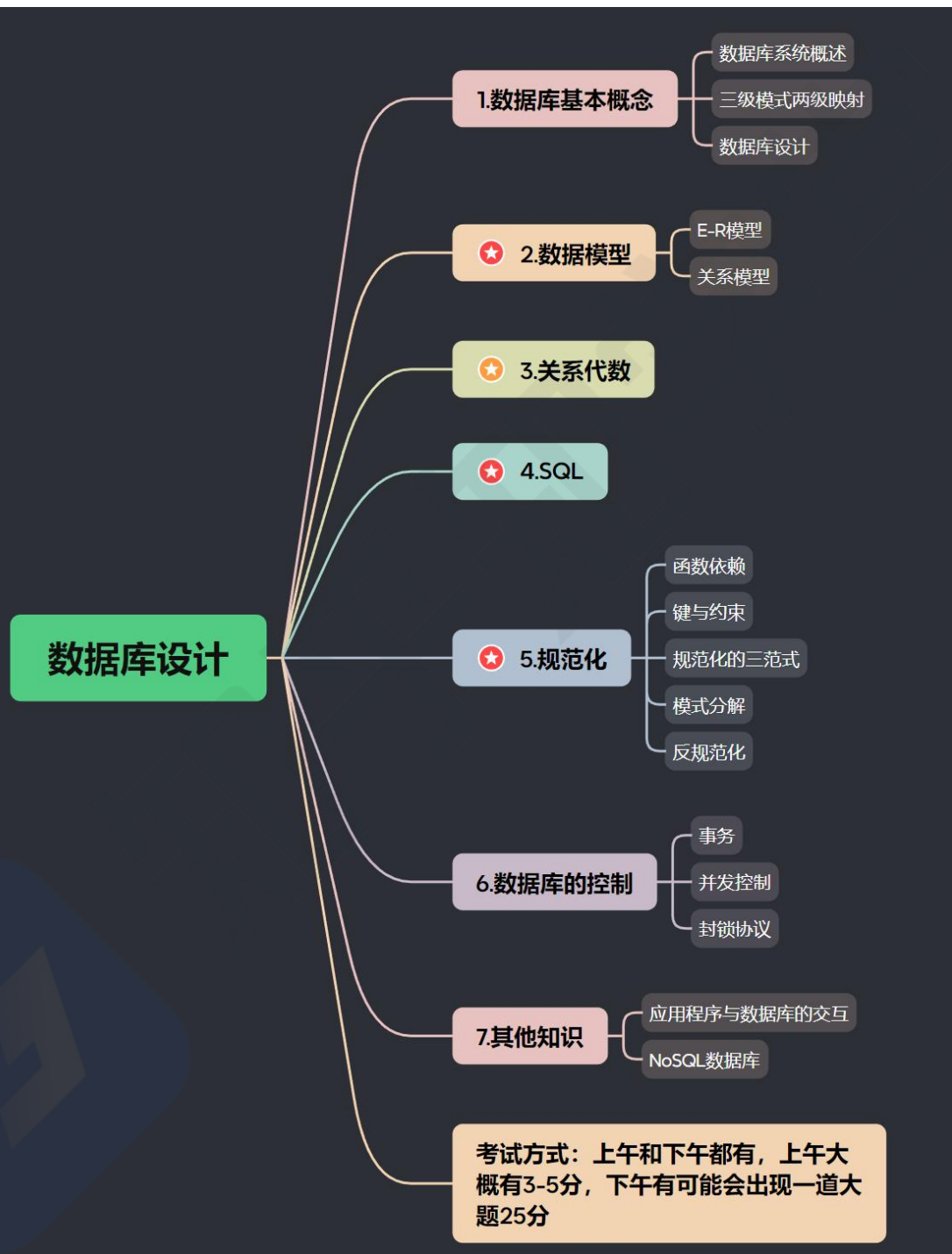
# 软考高级架构师

• 一段新征程 •





# 大纲介绍



## 反规范化



反规范化技术：规范化设计后，数据库设计者希望**牺牲部分规范化来提高性能**

- 采用反规范化技术的益处：降低连接操作的需求、降低外码和索引的数目，还可能**减少表的数目**，能够提高**查询效率**。
- 可能带来的问题：数据的**重复存储**，浪费了磁盘空间；可能出现数据的**完整性问题**，为了保障数据的一致性，增加了数据维护的复杂性，会**降低修改速度**。



具体方式：

- 增加冗余列：在**多个表中保留相同的列**，通过增加数据冗余减少或避免查询时的连接操作。
- 增加派生列：在表中增加可以由**本表或其它表中数据计算生成的列**，减少查询时的连接操作并避免计算或使用集合函数。
- 重新组表：如果许多用户需要查看两个表连接出来的结果数据，则把这**两个表重新组成一个表来减少连接而提高性能**。
- 水平分割表：根据一列或多列数据的值，把**数据放到多个独立的表中**，主要用于表数据规模很大、表中数据相对独立或数据需要存放到多个介质上时使用。
- 垂直分割表：对表进行分割，将**主键与部分列放到一个表中**，主键与其它列放到另一个表中，在查询时减少I/O次数。

## 模式分解



**模式分解**：是关系数据库规范化设计中的一个重要概念。它指的是通过对关系模式进行拆分，来消除模式中的混合组合依赖，达到将模式分解为更小的模式的过程。一般分为以下两种：

- **是否保持函数依赖分解**：对于关系模式R，有依赖集F，若对R进行分解，分解出来的多个关系模式，保持原来的依赖集不变，则为保持函数依赖的分解。另外，注意要消除掉冗余依赖(如传递依赖)
- **有损无损分解**：分解后的关系模式能够还原出原关系模式，就是无损分解，不能还原就是有损



**是否保持函数依赖案例**：设原关系模式R(A, B, C)，依赖集F( $A \rightarrow B$ ,  $B \rightarrow C$ ,  $A \rightarrow C$ )，将其分解为两个关系模式R1(A, B)和R2(B, C)，此时R1中保持依赖 $A \rightarrow B$ ，R2保持依赖 $B \rightarrow C$ ，说明分解后的R1和R2是保持函数依赖的分解，因为 $A \rightarrow C$ 这个函数依赖实际是一个冗余依赖，可以由前两个依赖传递得到，因此不需要管。



## 模式分解



当分解为两个关系模式，可以通过以下定理判断是否无损分解：

- 定理：如果R的分解为 $p=\{R1, R2\}$ ，F为R所满足的函数依赖集合，**分解p具有无损连接性的充分必要条件是 $R1 \cap R2 \rightarrow (R1 - R2)$ 或者 $R1 \cap R2 \rightarrow (R2 - R1)$** 。
- 当分解为三个及以上关系模式时，可以通过表格法求解，如下：

思考题：

有关系模式：成绩（学号，姓名，课程号，课程名，分数）

函数依赖：学号 $\rightarrow$ 姓名，课程号 $\rightarrow$ 课程名，（学号，课程号） $\rightarrow$ 分数

若将其分解为：

成绩（学号，课程号，分数）

学生（学号，姓名）

课程（课程号，课程名）

请思考该分解是否为无损分解？

由于有：学号 $\rightarrow$ 姓名，所以：

成绩（学号，课程号，分数，姓名）

由于有：课程号 $\rightarrow$ 课程名，所以：

成绩（学号，课程号，分数，姓名，课程名）

初始表如下：

|    | 学号 | 姓名 | 课程号 | 课程名 | 分数 |
|----|----|----|-----|-----|----|
| 成绩 | √  | ×  | √   | ×   | √  |
| 学生 | √  | √  | ×   | ×   | ×  |
| 课程 | ×  | ×  | √   | √   | ×  |

根据学号 $\rightarrow$ 姓名，对上表进行处理，将×改成符号√；然后考虑课程号 $\rightarrow$ 课程名，将×改为√，得下表：

|    | 学号 | 姓名 | 课程号 | 课程名 | 分数 |
|----|----|----|-----|-----|----|
| 成绩 | √  | √  | √   | √   | √  |
| 学生 | √  | √  | ×   | ×   | ×  |
| 课程 | ×  | ×  | √   | √   | ×  |

从上图中可以看出，第1行已全部为√，因此本次R分解是无损联接分解。



假设关系模式 $R(U, F)$ ，属性集 $U=\{A, B, C\}$ ，函数依赖集 $F=\{A \rightarrow B, B \rightarrow C\}$ 。若将其分解为 $p=\{R_1(U_1, F_1), R_2(U_2, F_2)\}$ ，其中 $U_1=\{A, B\}$ ， $U_2=\{A, C\}$ 。那么，分解 $p$ ( )。

A.有损连接但保持函数依赖

B.既无损连接又保持函数依赖

C.有损连接且不保持函数依赖

D.无损连接但不保持函数依赖



给定关系模式 $R\langle U, F \rangle$ ， $U=\{A, B, C, D, E\}$ ， $F=\{B \rightarrow A, D \rightarrow A, A \rightarrow E, AC \rightarrow B\}$ ，则 $R$ 的候选关键字为( )，分解 $p=\{R_1(ABCE), R_2(CD)\}$ ( )。

A.CD

B.ABD

C.ACD

D.ADE

A.具有无损连接性，且保持函数依赖

B.不具有无损连接性，但保持函数依赖

C.具有无损连接性，但不保持函数依赖

D.不具有无损连接性，也不保持函数依赖

## 并发控制



**事务：**由一系列DML操作组成，这些操作，要么全做，要么全不做，它从第一个DML操作开始，rollback、commit或者DDL结束，拥有以下四种特性，详解如下：

- **(操作)原子性：**要么全做，要么全不做，例如银行转账，在没到最后一步完成转账之前，前面的所有操作都是无效的。
- **(数据)一致性：**事务发生后数据是一致的，例如银行转账，不会存在A账户转出，但是B账户没收到的情况。
- **(执行)隔离性：**任一事务的更新操作直到其成功提交的整个过程对其他事务都是不可见的，不同事务之间是隔离的，互不干涉。
- **(改变)持续性：**事务操作的结果是持续性的。



事务是并发控制的前提条件，并发控制就是**控制不同的事务并发执行，提高系统效率**，但是并发控制中存在下面三个问题：

## 并发控制



**丢失更新**：事务1对数据A进行了修改并写回，事务2也对A进行了修改并写回，此时事务2写回的数据会覆盖事务1写回的数据，就丢失了事务1对A的更新。即对数据A的更新会被覆盖。



**不可重复读**：事务2读A，而后事务1对数据A进行了修改并写回，此时若事务2再读A，发现数据不对。即一个事务重复读A两次，会发现数据A有误。



**读脏数据**：事务1对数据A进行了修改后，事务2读数据A，而后事务1回滚，数据A恢复了原来的值，那么事务2对数据A做的事是无效的，读到了脏数据。

### ◆ 丢失更新

| T1                           | T2                    |
|------------------------------|-----------------------|
| ①读A=10<br>②<br>③A=A-5写回<br>④ | 读A=10<br><br>A=A-8 写回 |

### ◆ 不可重复读

| T1                                                                       | T2                       |
|--------------------------------------------------------------------------|--------------------------|
| ①读A=20<br>读B=30<br>求和=50<br>②<br><br>③读A=70<br>读B=30<br>求和=100<br>(验算不对) | 读A=20<br>A←A+50<br>写A=70 |

### ◆ 读“脏”数据

| T1                                                   | T2    |
|------------------------------------------------------|-------|
| ①读A=20<br>A←A+50<br>写回70<br>②<br>③ROLLBACK<br>A恢复为20 | 读A=70 |

## 封锁协议



**X锁是排它锁(写锁)**。若事务T对数据对象A加上x锁，则只允许T读取和修改A，其他事务都不能再对A加任何类型的锁，直到T释放A上的锁。



**S锁是共享锁(读锁)**。若事务T对数据对象A加上S锁，则只允许T读取A，但不能修改A，其他事务只能再对A加S锁(也能读不能修改)，直到T释放A上的S锁。



共分为三级封锁协议，如下：

- 一级封锁协议：事务在修改数据R之前必须先对其加X锁，直到事务结束才释放。可解决丢失更新问题。

| T1                                                               | T2                                                      |
|------------------------------------------------------------------|---------------------------------------------------------|
| ① 对A加写锁<br>②<br>③ 读A=10<br>④ A=A-5写回<br>⑤ 释放对A的写锁<br>⑥<br>⑦<br>⑧ | 对A加写锁<br>等待<br>等待<br>等待<br>读A= 5<br>A=A-8 写回<br>释放对A的写锁 |



## 封锁协议

- 二级封锁协议：一级封锁协议的基础上加上事务T在**读数据R之前必须先对其加S锁**，**读完后即可释放S锁**。可解决丢失更新、读脏数据问题。

| T1                                                                                                                                                                                        | T2                                                                                                        |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>① 对A加写锁</li><li>② 读A=20</li><li>③ <math>A \leftarrow A + 50</math></li><li>④ 写回70</li><li>⑤</li><li>⑥ ROLLBACK</li><li>⑦ A恢复为20</li><li>⑧</li></ul> | <ul style="list-style-type: none"><li>对A加读锁</li><li>等待</li><li>等待</li><li>读A=20</li><li>释放对A的读锁</li></ul> |

## 封锁协议

- 三级封锁协议：一级封锁协议加上事务T在读取数据R之前先对其加S锁，直到事务结束才释放。可解决丢失更新、读脏数据、数据重复读问题。

| T1                                                                                                           | T2                                                                                                   |
|--------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| <p>① 对A与B加S锁（读锁）<br/>读A=20<br/>读B=30<br/>求和=50</p> <p>②</p> <p>③ 读A=20<br/>读B=30<br/>求和=50<br/>释放对A和B的读锁</p> | <p>对A加X锁（写锁）<br/>注：由于A已加了读锁，所以等待<br/>等待<br/>等待<br/>等待<br/>读A=20<br/>A←A+50<br/>写A=70<br/>释放对A的写锁</p> |



当多个事务并发执行时，任一事务的更新操作直到其成功提交的整个过程对其他事务都是不可见的，这一性质通常被称为事务的( )。

A.原子性

B.一致性

C.隔离性

D.持久性



若事务T1对数据D1加了共享锁，事务T2、T3分别对数据D2、D3加了排它锁，则事务T1对数据( );事务T2对数据( )

A.D2、D3加排它锁都成功

B.D2、D3加共享锁都成功

C.D2加共享锁成功， D3加排它锁失败

D.D2、D3加排它锁和共享锁都失败

A.D1、D3加共享锁都失败

B.D1、D3加共享锁都成功

C.D1加共享锁成功， D3加排它锁失败

D.D1加排它锁成功， D3加共享锁失败

# SQL语言



SQL(Structured Query Language): 是关系数据库管理系统的标准查询语言, 用于存取数据以及查询、更新和管理关系数据库



## SQL的主要功能包括:

1. **数据定义语言(DDL)**: 用于定义数据库对象, 如表、视图、索引等。包括CREATE、ALTER、DROP等语句。
2. **数据操作语言(DML)**: 用于对数据库中表的数据进行增删改操作。包括INSERT、UPDATE、DELETE等语句。
3. **数据查询语言(DQL)**: 用于查询数据库中表的记录。主要是SELECT语句及其子句
4. **事务控制语言(TCL)**: 用于管理数据库事务。包括COMMIT、ROLLBACK、SAVEPOINT等语句。
5. **数据控制语言(DCL)**: 用于管理数据库的权限和安全性。主要是GRANT和REVOKE语句。

# SQL语言



## 常见的DDL操作:

1. 创建操作: `create database/table/index/view 名称;`
2. 修改操作: `alter table 表名 add/drop/modify 列名;`
3. 删除操作: `drop database/table/index/view 名称;`



## 常见的DML操作:

1. 数据库插入: `insert into values(): insert into t1 values('a', 66)`
2. 数据库删除: `delete from..where: delete from t1 where sno=4`
3. 数据库修改: `update ... set ... where ...: update t1 set sname='aa' where sno=3`

## SQL语言



常见的**DQL**操作: `select ... from ... where ...group by ...having...order by...limit...`

1. 排序order by, 默认为升序(ASC), 降序要加关键字DESC: `select * from t1 order by sno`
2. 分页limit, `limit startIndex,pageSize`, `startIndex`代表从第几项开始, `pageSize`代表展示多少项数据, `startIndex`从0开始算: `select * from t1 limit 0,10`
3. 分组查询group by, 分组时要注意select后的列名要适应分组, having为分组查询附加条件: `select sno, avg(score) from student group by sno having(avg(score)>60)`
4. 更名运算as: `select sno as“学号”from table`
5. 字符串匹配: like, %匹配多个字符串, \_匹配任意一个字符串: `select * from t1 where sname like'a_'`

常见的**TCL**操作:



1. 开启事务: `start transaction`
2. 事务提交: `commit`
3. 事务回滚: `rollback`
4. 设置保存点: `save point`
5. 回滚至保存点: `rollback to savepoint`



### 常见的DCL操作:

1. 创建用户 (CREATE USER) : 用于创建新的数据库用户。
2. 删除用户 (DROP USER) : 用于删除已存在的数据库用户。
3. 修改用户密码 (ALTER USER) : 用于修改数据库用户的密码。
4. 授予权限 (GRANT) : 用于为用户或角色授予对数据库对象的权限。
5. 撤销权限 (REVOKE) : 用于撤销用户或角色已获得的对数据库对象的权限。
6. 创建角色 (CREATE ROLE) : 用于创建新的数据库角色。
7. 删除角色 (DROP ROLE) : 用于删除已存在的数据库角色。
8. 授予角色 (GRANT ROLE) : 用于将角色授予给用户或其他角色。
9. 撤销角色 (REVOKE ROLE) : 用于撤销用户或角色已获得的角色。



### 其他的表操作:

1. 指定主键 (primary key()) : 用于设置某个表的字段为主键。
2. 指定外键 (foreign key()) : 用于设置某个表的某个字段为外键, 后面还需要接关联哪个表的主键字段。



## 真题



某销售公司数据库的零件关系P(零件号, 零件名称, 供应商, 供应商所在地, 库存量), 函数依赖集F={零件号→零件名称, (零件号, 供应商)→库存量, 供应商→供应商所在地。零件关系P属于()。查询各种零件的平均库存量、最多库存量与最少库存量之间差值的SQL语句如下: SELECT 零件号, () FROM P ()

A.1NF

B.2NF

C.3NF

D.4NF

A.AVG(库存量) AS 平均库存量, MAX(库存量)-MIN(库存量) AS 差值

B.平均库存量 AS AVG(库存量), 差值AS MAX(库存量)-MIN(库存量)

C.AVG库存量 AS 平均库存量, MAX库存量-MIN库存量 AS 差值

D.平均库存量 AS AVG库存量, 差值AS MAX(库存量)-MIN库存量

A.ORDER BY 供应商

B.ORDER BY 零件号

C.GROUP BY 供应商

D.GROUP BY 零件号



## 应用程序与数据库交互



应用程序与数据交互：应用程序通过程序接口来访问数据库并进行操作。

- 库函数级别访问接口：最底层的访问方式，比如使用OCI来访问数据库，开发效率低，依赖特定的数据库，学习难度高
- 嵌入SQL访问接口：直接将SQL语句写入到变成语句的源码中，需要数据库厂商提供一个嵌入式SQL的预编译器，方便对嵌入式SQL的代码进行预编译并操作数据库，操作起来比较麻烦，需要和DBMS进行交互，而且性能也不高
- 通用数据接口标准：为不同的数据库提供统一的接口(ODBC)，允许我们在程序里面书写SQL语句进行操作数据库，不通过DBMS，直接使用SQL操作数据库中的数据，但是直接使用ODBC比较麻烦，后来又发展出了DAO、RDO和ADO这些数据库访问接口以及专门给.NET使用的ADO和Java使用的JDBC
- ORM访问接口：使用框架技术让对象和数据库中的表产生映射，让我们直接操作对象就能修改表数据，典型的有Hibernate、MyBatis、JPA等框架，这种开发效率最高，降低了程序员对数据库知识的要求

# NoSQL数据库



**NoSQL:** **Non-Relational**或者是**Not Only SQL**，泛指非关系型数据库，区分开关系型数据库，并且不保证关系型数据库的**ACID**特性。



**NoSQL的分类:**

- **列式存储数据库:** 跟传统的关系型数据库一样，数据按行列进行存储，这种类别通常用来应对分布式数据库的存储海量数据，比如HBase
- **键值对存储数据库:** 以key-value的形式来存储数据，特点是简单、容易部署，比如Redis
- **文档型数据库:** 类似于键值对数据库，可以看成是键值对数据库的升级版，允许嵌套键值，处理复杂数据的时候比传统的键值对存储效率高，比如MongoDB
- **图数据库:** 使用灵活的图形模型来存储数据，能够拓展到多个服务器上，适合存储通过图进行建模的数据，比如社交网络、交通网络等，常见的产品有Neo4J



**NoSQL的特征:** **易拓展、大数据量、高性能、灵活的数据模型、高可用**



**NoSQL的框架分层(从下至上):** **数据持久层、数据分布层、数据逻辑模型层和接口层**，层次之间相辅而成，协调工作



**NoSQL适用于哪些场景:** **数据模型比较简单、需要灵活性更强的系统、对数据性能要求高、不需要高度的数据一致性**

T H E   E N D

功不唐捐，玉汝于成！

• 开 启 新 征 程 •

