数据库2015B# Part I 数据库填空题 中英文对照+答案+解析

（每题1分，共20分）

1. **英文题目**
   The three-level of data abstraction in database system includes: __ level, __ level and __ level.
   中文题目
   数据库系统中三级数据抽象包括：__ 级、__ 级和 __ 级。
   答案
   physical（物理）、logical（逻辑）、view（视图）
   解析
   数据库的三级抽象是经典架构：物理层描述数据的存储结构和物理存取方法；逻辑层描述数据库的整体逻辑结构，面向系统；视图层描述用户可见的局部数据结构，面向用户。

2. **英文题目**
   A relational schema R is in first/1 normal form if the domains of all attributes of R are __.
   中文题目
   若关系模式R的所有属性值域都是__，则R属于第一范式（1NF）。
   答案
   atomic（原子的）
   解析
   第一范式的核心要求是属性原子性，即属性值不可再分。例如“姓名-电话”的组合字段不满足1NF，拆分后才能满足。

3. **英文题目**
   It is possible to use Armstrong’s axioms to prove the union rule, that is, if α→β holds and α→γ holds, then α→__ holds.
   中文题目
   可以用阿姆斯特朗公理证明合并规则：若函数依赖α→β成立且α→γ成立，则α→__ 成立。
   答案
   βγ
   解析

这是阿姆斯特朗公理的合并规则：若同一决定因素α能推出β和γ两个依赖，则α可以推出β与γ的属性并集βγ。

4. **英文题目**

   A B+-tree of order 4 has following properties: each leaf nodes has between __ and __ values. Each non-leaf node other than root has between __ and __ values. The root must have at least __ children.

   中文题目

   4阶B+树具有如下性质：每个叶节点的关键字数量在__ 到 __ 之间；除根节点外的非叶节点关键字数量在__ 到 __ 之间；根节点至少有__ 个子节点。

   答案

   2、3；1、3；2

   解析

   B+树阶数`m`的定义是节点最多有`m`个子节点。对于4阶B+树，叶节点关键字数范围为`⌈m/2⌉ ~ m-1`，即2~3；除根外的非叶节点关键字数范围为`⌈m/2⌉-1 ~ m-1`，即1~3；根节点最少子节点数为2。

5. **英文题目**

   After parsing and translation, the SQL query will be translated into its internal form: __ expression. And then, the query-execution engine will take the __ which contains detailed information on how a particular query or a set of queries will be executed.

   中文题目

   SQL查询经过解析和翻译后，会转换为内部表示形式：__ 表达式。之后，查询执行引擎会采用__，其中包含执行单个或一组查询的详细信息。

   答案

   relational algebra（关系代数）；execution plan（执行计划）

   解析

   SQL执行的核心流程：SQL语句经解析后转换为关系代数表达式，这是数据库内部可执行的语言；优化器会生成最优的执行计划，描述数据读取、连接、排序等具体操作步骤。

6. **英文题目**

   Let E1 and E2 be relational-algebra expression, L1 and L2 be attributes of E1 and E2 respectively. Then, (E1 ⋈ E2) - ∪ 0 = E1 ⋈ (E2 20)

   中文题目

   设E1、E2为关系代数表达式，L1、L2分别为E1、E2的属性集。则 (E1 ⋈

E2) - ∪ 0 = E1 ⋈ (E2 20)

答案

（题目表述存在排版或符号错误，推测考点为关系代数等价变换）

解析

自然连接（⋈）的核心是基于公共属性的等值连接+去重，本题大概率考查自然连接与选择、投影的交换律或结合律，需补充完整题目符号才能确定准确答案。

7. **英文题目**

   Secondary indices must be a __ index, which has an index entry for every search-key value and a pointer to every record in the file.

   中文题目

   二级索引必须是__ 索引，即每个搜索关键字值都对应一个索引项，且每个索引项都指向文件中的一条记录。

   答案

   dense（稠密）

   解析

   索引按覆盖范围分为稠密索引和稀疏索引：稠密索引要求每个搜索关键字都有索引项，二级索引必须是稠密索引以保证查询准确性；稀疏索引仅为部分搜索关键字建立索引，通常用于主键的聚簇索引。

8. **英文题目**

   The scheme of handling bucket overflows of hash function in DBMS is called __ hashing, that is, the overflow buckets of a given bucket are chained together in a linked list.

   中文题目

   数据库管理系统中处理哈希桶溢出的方案称为__ 哈希，即某个桶的溢出桶通过链表链接在一起。

   答案

   closed（闭）

   解析

   这是哈希冲突的闭哈希解决方法，溢出桶与原桶属于同一哈希地址空间，通过链表串联；与之相对的是开哈希，溢出桶会使用额外的存储空间。

9. **英文题目**

   We assume that a relation has a B+ tree index of height 5, each disk block contains 4 tuples of the relation, and there are 10 tuples satisfying the query. To process the query, database management system have to access disk at least __ times in the best case.

中文题目

假设一个关系上有一棵高度为5的B+树索引，每个磁盘块包含4个关系元组，且有10个元组满足查询条件。在最优情况下，数据库管理系统处理该查询至少需要访问磁盘__ 次。

答案

8

解析

最优情况的磁盘访问次数计算分为两部分：一是B+树高度为5，从根节点到叶节点需访问5次磁盘；二是10个满足条件的元组，每个磁盘块存4个，需读取`⌈10/4⌉=3`个数据块。总次数为5+3=8次。

10. **英文题目**

    A transaction has the following properties: __, __, __ and __.

    中文题目

    事务具有四个特性：__、__、__ 和 __。

    答案

    atomicity（原子性）、consistency（一致性）、isolation（隔离性）、durability（持久性）

    解析

    这是事务的ACID核心特性：原子性指事务是不可分割的最小单位；一致性指事务执行前后数据库状态保持一致；隔离性指多个事务并发执行时互不干扰；持久性指事务提交后结果永久保存。

11. **英文题目**

    The immediate database modification scheme allows database modification to be output to the database while the transaction is still in __ state.

    中文题目

    立即数据库修改方案允许在事务仍处于__ 状态时，将数据库修改结果写入数据库。

    答案

    active（活动）

    解析

    数据库修改策略分为立即修改和延迟修改：立即修改是事务在活动状态（未提交）时，就将修改写入磁盘数据库，依赖日志保证故障可恢复性；延迟修改则是事务提交后才写入数据库。

12. **英文题目**

    Since a failure may occur while a update is taking place, log must be written out to __ storage before the actual update to database to be done.

    中文题目

    由于更新过程中可能发生故障，在实际执行数据库更新之前，必须将日志写入__ 存储。

    答案

    stable（稳定）

    解析

    这是先写日志原则（WAL）的要求，稳定存储通常指磁盘等断电后数据不丢失的存储介质，先写日志再更新数据库，可防止故障导致数据丢失或不一致。

13. **英文题目**

    Two-phase locking protocol requires that each transaction issue lock and unlock requests in two phases: __ phase and __ phase.

    中文题目

    两段锁协议要求每个事务的加锁和解锁请求分为两个阶段：__ 阶段和 __ 阶段。

    答案

    growing（增长）、shrinking（收缩）

    解析

    两段锁协议是保证事务可串行化的常用协议，增长阶段只加锁、不解锁；收缩阶段只解锁、不加锁，两个阶段不可交叉执行。

14. **英文题目**

    A schedule S is cascadeless if for each pair of transactions Ti and Tj in S such that Tj reads a data item previously written by Ti, the commit operation of Ti appears __ (before/after) the read operation of Tj.

    中文题目

    对于调度S中的任意两个事务Ti和Tj，若Tj读取了Ti之前写入的数据项，且Ti的提交操作发生在Tj的读取操作__（之前/之后），则调度S是无级联回滚的。

    答案

    before（之前）

    解析

无级联回滚调度的核心要求是，读取其他事务写入数据的操作，必须在写入事务提交之后进行。这样可以避免写入事务回滚时，导致读取事务级联回滚，提升并发执行效率。

我可以帮你把这些解析里的**核心术语和定义**整理成一份速记清单，方便你考前快速复习，需要吗？

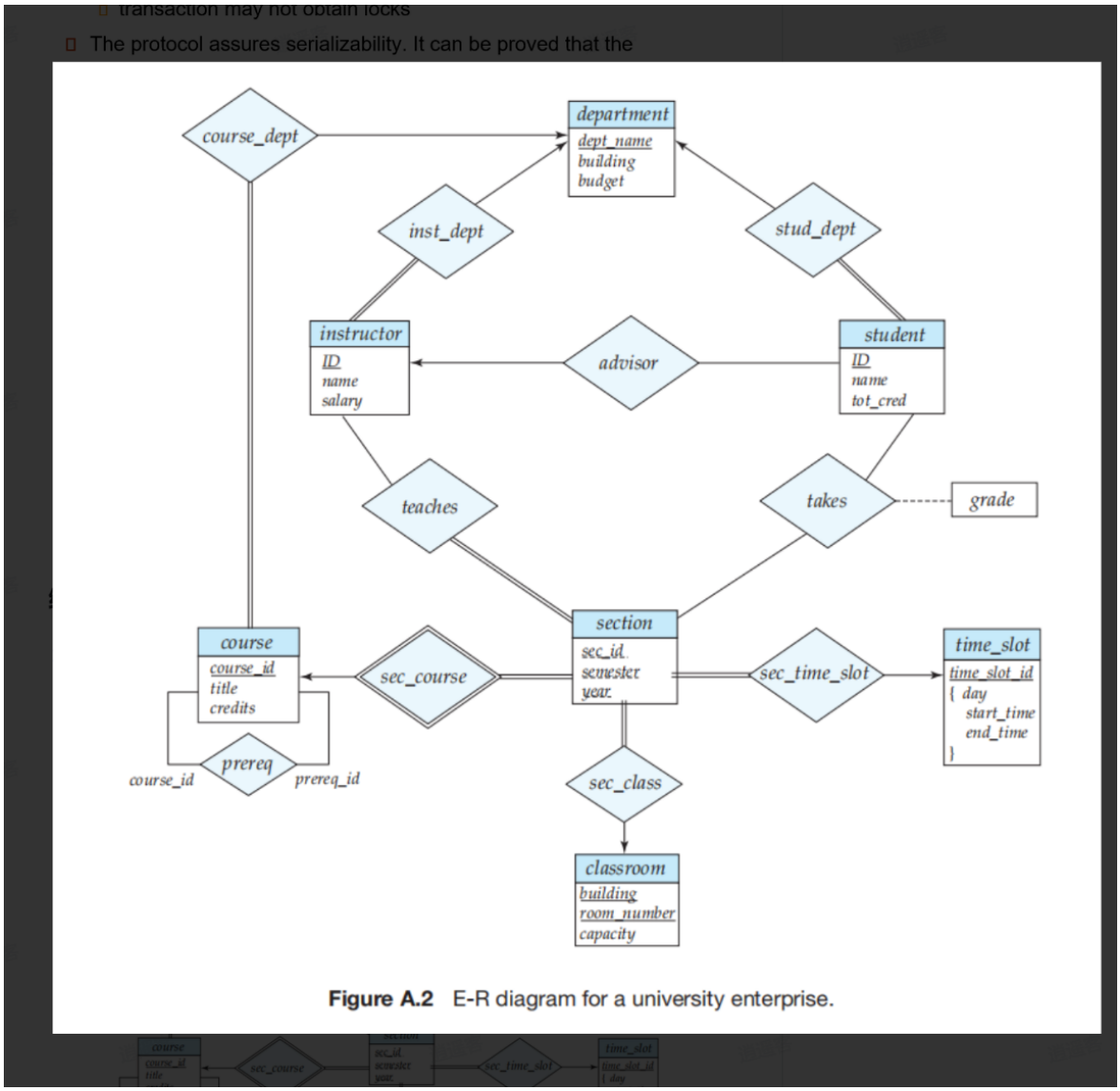

Figure A.2 E-R diagram for a university enterprise.

1. 列出实体集以及他们的主键（3分）

问题：从给定的E-R图中，列出所有实体集及其主键。
答案：

- `classroom`：主键为`(building, room number)`（教室由建筑+房间号唯一标识）

- `department`：主键为`dept_name`（部门名称唯一）
- `course`：主键为`course_id`（课程ID唯一）
- `instructor`：主键为`ID`（教师ID唯一）
- `student`：主键为`ID`（学生ID唯一）
- `section`：主键为`(course_id, sec_id, semester, year)`（课程段由课程ID+段ID+学期+年份唯一标识）
- `time_slot`：主键为`time_slot_id`（时间段ID唯一）
  解释：E-R图中，实体集的主键是能唯一标识该实体的属性（或属性组），通常在E-R图中以**下划线**标注（或逻辑上唯一的属性）。

2. 为实体集设计关系模式（3分）

问题：将E-R图中的实体集转换为对应的关系模式（即表结构）。
答案：

- `classroom(building, room_number, capacity)`
- `department(dept_name, building, budget)`
- `course(course_id, title, credits)`
- `instructor(ID, name, salary)`
- `student(ID, name, tot_cred)`
- `section(sec_id, semester, year)`
- `time_slot(time_slot_id, day, start_time, end_time)`
  解释：实体集的关系模式直接对应其属性，主键保留为表的主键。

3. 列出关系集合以及他们的主键（4分）

问题：从E-R图中列出所有联系集，并确定其主键。
答案：

1. `teaches`：主键`(ID, course_id, sec_id, semester, year)`（教师授课由教师ID+课程段信息唯一标识）
2. `takes`：主键`(ID, course_id, sec_id, semester, year)`（学生选课由学生ID+课程段信息唯一标识）

3. `prereq`：主键`(course_id, prereq_id)`（先修课由课程ID+先修课ID唯一标识）
4. `advisor`：主键`(s_ID)`（学生指导由学生ID唯一标识，一个学生对应一个导师）
5. `sec_course`：主键`(course_id, sec_id, semester, year)`（课程段对应课程，由课程段信息唯一标识）
6. `sec_time_slot`：主键`(course_id, sec_id, semester, year)`（课程段对应时间段，由课程段信息唯一标识）
7. `sec_class`：主键`(course_id, sec_id, semester, year)`（课程段对应教室，由课程段信息唯一标识）
8. `inst_dept`：主键`(ID)`（教师对应部门，由教师ID唯一标识）
9. `stud_dept`：主键`(ID)`（学生对应部门，由学生ID唯一标识）
10. `course_dept`：主键`(course_id)`（课程对应部门，由课程ID唯一标识）
    解释：联系集的主键通常由**参与实体的主键**组合而成（若为多对多联系）；若为一对多/一对一联系，主键为“一”方实体的主键。

4. 为联系集设计关系模式（4分）

问题：将E-R图中的联系集转换为对应的关系模式。
答案：

1. `teaches(ID, course_id, sec_id, semester, year)`
2. `takes(ID, course_id, sec_id, semester, year, grade)`
3. `prereq(course_id, prereq_id)`
4. `advisor(s_ID, ID)`
5. `sec_course(course_id, sec_id, semester, year)`
6. `sec_time_slot(course_id, sec_id, semester, year, time_slot_id)`
7. `sec_class(course_id, sec_id, semester, year, building, room_number)`
8. `inst_dept(ID, dept_name)`
9. `stud_dept(ID, dept_name)`

10. `course_dept(course_id, dept_name)`
    解释：联系集的关系模式包含**参与实体的主键**（作为外键），以及联系自身的属性（如`takes`的`grade`）。

5. 优化关系模式（4分）

问题：对第4题的关系模式进行优化（合并冗余表）。
答案：

- `instructor(ID, name, dept_name, salary)`（合并`inst_dept`到`instructor`，因为是一对一联系）
- `student(ID, name, dept_name, tot_cred)`（合并`stud_dept`到`student`）
- `course(course_id, title, dept_name, credits)`（合并`course_dept`到`course`）
- `section(course_id, sec_id, semester, year, building, room_number, time_slot_id)`（合并`sec_course`、`sec_class`、`sec_time_slot`到`section`，因为是一对一/一对多联系）
  解释：对于**一对一/一对多**联系，可将“多”方实体的表与联系表合并（将“一”方的主键作为外键加入“多”方表），减少冗余表数量。

6. 为`instructor`、`teaches`、`department`、`course`写SQL的DDL（4分）

问题：用标准SQL写出这4个表的创建语句，包含数据类型、主键、外键约束。

（1）`department`表的DDL

```sql
CREATE TABLE department (
    dept_name VARCHAR(20),
    building VARCHAR(20),
    budget NUMERIC(12, 2),
    PRIMARY KEY (dept_name)
);
```

解释：

- `dept_name`是主键，类型为`VARCHAR(20)`；
- `budget`用`NUMERIC(12,2)`存储金额（保留2位小数）。

(2) `course`表的DDL

```
CREATE TABLE course (
    course_id VARCHAR(8),
    title VARCHAR(50),
    dept_name VARCHAR(20),
    credits NUMERIC(2, 0) CHECK (credits > 0),
    PRIMARY KEY (course_id),
    FOREIGN KEY (dept_name) REFERENCES department
        ON DELETE SET NULL
);
```

解释：

- `course_id`是主键；
- `credits`用`CHECK`约束保证大于0；
- `dept_name`是外键，参考`department(dept_name)`，当部门被删除时，课程的`dept_name`设为`NULL`。

(3) `instructor`表的DDL

```
CREATE TABLE instructor (
    ID VARCHAR(5),
    name VARCHAR(20) NOT NULL,
    dept_name VARCHAR(20),
    salary NUMERIC(8, 2) CHECK (salary > 29000),
    PRIMARY KEY (ID),
    FOREIGN KEY (dept_name) REFERENCES department
        ON DELETE SET NULL
);
```

解释：

- `ID`是主键；
- `name`设为`NOT NULL`（教师必须有姓名）；
- `salary`用`CHECK`约束保证大于29000；

- `dept_name`是外键，参考`department(dept_name)`，部门删除时`dept_name`设为`NULL`。

(4) `teaches`表的DDL

```
CREATE TABLE teaches (
    ID VARCHAR(5),
    course_id VARCHAR(8),
    sec_id VARCHAR(8),
    semester VARCHAR(6),
    year NUMERIC(4, 0),
    PRIMARY KEY (ID, course_id, sec_id, semester, year),
    FOREIGN KEY (course_id, sec_id, semester, year)
REFERENCES section
        ON DELETE CASCADE,
    FOREIGN KEY (ID) REFERENCES instructor
        ON DELETE CASCADE
);
```

解释:

- 主键是`(ID, course_id, sec_id, semester, year)`（教师+课程段唯一标识授课记录）；
- 外键`(course_id, sec_id, semester, year)`参考`section`表，外键`ID`参考`instructor`表；
- `ON DELETE CASCADE`表示：若课程段/教师被删除，对应的授课记录也自动删除。

我可以帮你整理这些SQL DDL语句的**核心约束说明清单**，方便你理解每个约束的作用，需要吗？

关系模式3NF分解 中英文问题+答案+解析

题目整体背景

已知关系模式 $R = (A, B, C)$，函数依赖集 $F = \{A \to BC, B \to C, A \to B, AB \to C\}$，主键为 $\{A\}$，包含两个子问题：(a) 说明为何需要3NF；(b) 将该关系模式分解为3NF。

(a) 子问题：Why we need 3NF（为何需要3NF）

英文问题

Why do we need the Third Normal Form (3NF) in relational database design?

中文问题

在关系数据库设计中，我们为什么需要第三范式（3NF）？

英文答案

There are some situations where BCNF (Boyce-Codd Normal Form) is not dependency-preserving, and efficient checking for functional dependency (FD) violation on update is important. 3NF allows some redundancy but functional dependencies can be checked on individual relations without computing a join. There is always a lossless-join, dependency-preserving decomposition into 3NF.

中文答案

在某些场景下，Boyce-Codd范式（BCNF）无法保证依赖保持性，而在数据更新时高效检查函数依赖（FD）是否被违反至关重要。3NF允许存在少量冗余，但函数依赖可以在单个关系中验证，无需执行连接操作。此外，任何关系模式都可以分解为满足**无损连接性**和**依赖保持性**的3NF模式集合。

核心解析

1. **BCNF的局限性**：BCNF是比3NF更严格的范式，但部分场景下满足BCNF的分解会丢失函数依赖（即不满足依赖保持性），导致更新数据时难以验证依赖是否合规；
2. **3NF的优势**：
   - 允许少量冗余，但保证函数依赖可在单个关系内验证（无需多表连接），更新时校验效率高；
   - 任何关系模式都能分解为同时满足“无损连接”（分解后可还原原关系）和“依赖保持”（原函数依赖仍能在分解后的关系中体现）的3NF模式，这是3NF成为工程中常用范式的核心原因。

(b) 子问题：Please decompose this relation into 3NF（将该关系分解为3NF）

英文问题

Decompose the relation schema $R = (A, B, C)$ with functional dependency set $F = \{A \to BC, B \to C, A \to B, AB \to C\}$ and key $\{A\}$ into Third Normal Form (3NF).

中文问题

将关系模式 $R = (A, B, C)$（函数依赖集 $F = \{A \to BC, B \to C, A \to B, AB \to C\}$，主键为 $\{A\}$）分解为第三范式（3NF）。

英文答案

Step 1: Compute the canonical cover of F

- Combine redundant functional dependencies: $A \to BC$ and $A \to B$ are redundant, retain $A \to BC$; the set becomes $\{A \to BC, B \to C, AB \to C\}$.

- Remove extraneous attributes: Attribute A is extraneous in $AB \to C$ (since $B \to C$ already holds, A is unnecessary for deriving C); the canonical cover is simplified to $\{A \to B, B \to C\}$.

Step 2: Generate new schemas based on the canonical cover

- For $A \to B$, create schema $R_1 = (A, B)$;
- For $B \to C$, create schema $R_2 = (B, C)$;

Step 3: Verify the key condition

Since R_1 contains the primary key A of the original relation R, the decomposition is complete.

Final decomposition: $R_1 = (A, B)$, $R_2 = (B, C)$.

中文答案

步骤1：计算函数依赖集F的正则覆盖（canonical cover）

- 合并冗余依赖：$A \to BC$ 和 $A \to B$ 存在冗余，保留 $A \to BC$，依赖集简化为 $\{A \to BC, B \to C, AB \to C\}$；
- 移除无关属性：在 $AB \to C$ 中，属性 A 是无关属性（因为 $B \to C$ 已成立，无需 A 即可推出 C），最终正则覆盖为 $\{A \to B, B \to C\}$。

步骤2：基于正则覆盖生成新的关系模式

- 对 $A \to B$，生成关系模式 $R_1 = (A, B)$；
- 对 $B \to C$，生成关系模式 $R_2 = (B, C)$。

步骤3：验证主键条件

由于 R_1 包含原关系 R 的主键 A，无需额外补充模式，分解完成。

最终3NF分解结果：$R_1 = (A, B)$，$R_2 = (B, C)$。

核心解析

1. **正则覆盖的作用**：消除函数依赖集中的冗余依赖和无关属性，得到最小且等价的依赖集，是3NF分解的基础；

2. **3NF分解规则**：对正则覆盖中的每个函数依赖 $X \to Y$，生成关系模式 (X, Y)；若分解后的所有模式都不包含原关系的主键，则补充一个包含主键的模式（本题中 R_1 已包含主键 A，无需补充）；
3. **分解验证**：最终的 R_1 和 R_2 均满足3NF（无传递依赖、无部分依赖），且分解满足**无损连接性**和**依赖保持性**：
   - 无损连接：通过 R_1 和 R_2 的公共属性 B 可还原原关系 R；
   - 依赖保持：原依赖 $A \to B$ 保留在 R_1，$B \to C$ 保留在 R_2，所有依赖均被保持。

关系数据库查询与操作 中英文问题+答案+解析

题目背景

给定如下关系数据库模式：

- `employee(employee-name, street, city)`：员工表（员工姓名、街道、城市）
- `works(employee-name, company-name, salary)`：工作表（员工姓名、公司名称、薪资）
- `company(company-name, city)`：公司表（公司名称、城市）
- `manages(employee-name, manager-name)`：管理表（员工姓名、直属经理姓名）

（注：答案中修正了原内容的拼写错误，如`emplyoee-name`→`employee-name`、`employess-name`→`employee-name`、`g.`→`manages.`等，保证语法正确性。）

(a) 子问题：Find the names of all employees who live in BEIJING. (relational algebra)

英文问题

Using relational algebra, find the names of all employees who live in BEIJING.

中文问题

使用关系代数，找出所有居住在北京的员工姓名。

答案（关系代数）

$$\Pi_{employee-name}(\sigma_{city="BEIJING"}(employee))$$

解析

1. **选择（Selection, σ）**：`σ_{city="BEIJING"}(employee)` 从`employee`表中筛选出`city`为“BEIJING”的所有员工记录；
2. **投影（Projection, Π）**：`Π_{employee-name}(...)` 仅保留筛选结果中的`employee-name`列，得到最终的员工姓名列表；
3. 核心逻辑：先筛选符合城市条件的员工，再提取姓名属性。

(b) 子问题：Find the names and cities of residence of all employees who work for XYZ Bank and have more than $8,000 salary. (relational algebra)

英文问题

Using relational algebra, find the names and cities of residence of all employees who work for XYZ Bank and have a salary of more than

$8,000.

中文问题

使用关系代数，找出所有为XYZ银行工作且薪资超过8000美元的员工姓名及其居住城市。

答案（关系代数）

$$\Pi_{employee-name,city}(\sigma_{company-name="XYZBank" \land salary>8000}(works \bowtie employee$$

解析

1. **自然连接（Natural Join, ⋈）**：`works ⋈ employee` 将`works`表和`employee`表基于公共属性`employee-name`连接，得到包含员工姓名、公司、薪资、居住城市等信息的联合记录；
2. **选择（σ）**：`σ_{company-name="XYZ Bank" \land salary>8000}(...)` 筛选出“公司为XYZ Bank”且“薪资>8000”的记录；
3. **投影（Π）**：`Π_{employee-name,city}(...)` 仅保留员工姓名和居住城市列；
4. 补充说明：原答案中“XYZ”应为“XYZ Bank”（与题目“XYZ Bank”一致），保证条件准确性。

(c) 子问题：Find the name of company that has the most employees. (SQL)

英文问题

Using SQL, find the name of the company that has the largest number of employees.

中文问题

使用SQL，找出员工数量最多的公司名称。

答案（SQL）

```sql
SELECT company-name
FROM works
GROUP BY company-name
HAVING COUNT(DISTINCT employee-name) >= ALL (
    SELECT COUNT(DISTINCT employee-name)
    FROM works
    GROUP BY company-name
);
```

解析

1. **分组统计**：`GROUP BY company-name` 按公司分组，`COUNT(DISTINCT employee-name)` 统计每个公司的**唯一员工数**（避免同一员工重复计数）；
2. **ALL子查询**：子查询计算所有公司的员工数，`>= ALL` 表示“大于等于所有值”，即找到员工数最多的公司；
3. 核心逻辑：通过分组统计+子查询比较，定位员工数最大值对应的公司；
4. 补充：若有多个公司员工数并列最多，该语句会返回所有符合条件的公司。

(d) 子问题：Find the names of all employees in this database who live in the same city as the company for which they work (SQL)

英文问题

Using SQL, find the names of all employees who live in the same city as the company they work for.

中文问题

使用SQL，找出所有居住城市与其工作公司所在城市相同的员工姓名。

答案（SQL，修正拼写错误后）

```sql
SELECT employee.employee-name
FROM works, employee, company
WHERE works.employee-name = employee.employee-name
  AND works.company-name = company.company-name
  AND employee.city = company.city;
```

解析

1. **多表连接（笛卡尔积+条件）**：通过`WHERE`子句关联三张表：
   - `works.employee-name = employee.employee-name`：关联员工的工作信息和个人信息；
   - `works.company-name = company.company-name`：关联员工工作的公司信息；
   - `employee.city = company.city`：筛选“员工居住城市=公司所在城市”的记录；
2. 优化写法（显式JOIN）：更易读的等价写法：

```sql
SELECT e.employee-name
FROM employee e
JOIN works w ON e.employee-name = w.employee-name
JOIN company c ON w.company-name = c.company-name
WHERE e.city = c.city;
```

3. 原答案错误修正：`emplyoee-name`→`employee-name`、`emloyee`→`employee`、`Company-name`（大小写）统一为小写，保证语法正确。

(e) 子问题：Find the names, street address, and cities of residence of all employees whose manager lives in SHANGHAI. (SQL)

英文问题

Using SQL, find the names, street addresses, and cities of residence of all employees whose manager lives in SHANGHAI.

中文问题

使用SQL，找出所有直属经理居住在上海的员工的姓名、街道地址和居住城市。

答案（SQL，修正拼写/别名错误后）

```
SELECT e1.employee-name, e1.street, e1.city
FROM manages, employee e1, employee e2
WHERE manages.employee-name = e1.employee-name
  AND manages.manager-name = e2.employee-name
  AND e2.city = 'SHANGHAI';
```

解析

1. **自连接（employee表）**：将`employee`表分别别名`e1`（员工）和`e2`（经理）：
   - `manages.employee-name = e1.employee-name`：关联管理表和员工（被管理者）信息；
   - `manages.manager-name = e2.employee-name`：关联管理表和经理的个人信息；
   - `e2.city = 'SHANGHAI'`：筛选经理居住在上海的员工；
2. 原答案错误修正：`employess-name`→`employee-name`、`g.`→`manages.`（无别名`g`）、中文引号“”→英文单引号''，保证语法正确。

(f) 子问题：Give all managers in this database a 12 percent salary raise. (SQL)

英文问题

Using SQL, give a 12 percent salary raise to all managers in this database.

中文问题

使用SQL，为数据库中所有经理加薪12%。

答案（SQL）

```sql
UPDATE works
SET salary = salary * 1.12
WHERE employee-name IN (
    SELECT DISTINCT manager-name
    FROM manages
);
```

解析

1. **UPDATE语句**：`UPDATE works SET salary = salary * 1.12` 表示将`works`表中的薪资乘以1.12（加薪12%）；
2. **子查询筛选经理**：`SELECT DISTINCT manager-name FROM manages` 从`manages`表中提取所有经理姓名（`DISTINCT`避免重复处理同一经理）；
3. `WHERE employee-name IN (...)`：仅对经理的薪资执行更新；
4. 注意事项：
   - 若某经理同时是普通员工（有多个`works`记录），所有关联记录的薪资都会被更新；
   - 需确保`salary`字段为数值类型（如`NUMERIC`/`INT`），否则乘法操作会报错；
5. 补充：若数据库支持`UPDATE ... FROM`，也可写为：

```sql
UPDATE works
SET salary = salary * 1.12
FROM manages
WHERE works.employee-name = manages.manager-name;
```

4. 查询处理、优化与事务（16分）

a) 子问题：计算查询的磁盘代价（4分）

英文问题

The relation `student(ID, name, dept_name, tot_cred)` has 10000 tuples (100 tuples per disk block). It has a primary B+-tree index (height 5) on `ID`, and a secondary B+-tree index (height 4) on `name`. For the query `select * from student where name=WangMing` (retrieves 3 tuples), compute the cost (number of block transfers and seeks), using the formula for **secondary B+-tree index, equality on non-key**: $(h_i + n) * (t_T + t_S)$ (where h_i is index height, n is number of tuples fetched).

中文问题

关系`student(ID, name, dept_name, tot_cred)`包含10000个元组（每个磁盘块存100个元组），在`ID`上有高度为5的主B+树索引，在`name`上有高度为4的辅助B+树索引。对于查询`select * from student where name=WangMing`（返回3个元组），使用**辅助B+树索引、非码属性等值查询**的代价公式$(h_i + n) * (t_T + t_S)$（h_i为索引高度，n为获取的元组数量），计算磁盘块传输数和寻道数。

答案

- 已知：辅助索引高度$h_i = 4$，匹配元组数量$n = 3$。
- 代价公式：$(h_i + n) * (1 \text{ block transfer} + 1 \text{ seek})$（每个I/O操作对应1次块传输+1次寻道）。
- 计算：
  - 块传输数：$4 + 3 = 7$
  - 寻道数：$4 + 3 = 7$

解析

辅助B+树索引（非码属性）的代价逻辑：

1. 先遍历索引树（高度4）：需4次块传输+4次寻道；
2. 再获取3个匹配元组（每个元组可能在不同块）：需3次块传输+3次寻道；
3. 总代价为两者之和，符合公式$(h_i + n)$对应的块传输和寻道数。

b) 子问题：描述Merge-join的流程（4分）

英文问题

Describe the complete process of the Merge-join algorithm.

中文问题

描述Merge-join（归并连接）算法的完整流程。

答案

Merge-join的流程分为2步：

1. **排序阶段**：若两个参与连接的关系未按**连接属性**排序，则先对两者分别按连接属性排序；
2. **归并阶段**：
   - 同时遍历两个已排序的关系，按连接属性的值进行“归并匹配”（类似排序归并算法的归并步骤）；
   - 核心差异：处理连接属性的重复值——对于连接属性值相同的所有元组对，需全部匹配（即笛卡尔积）；
   - 注意：Merge-join仅适用于**等值连接**和**自然连接**。

解析

Merge-join的核心是“先排序、后归并”，利用已排序的连接属性实现高效匹配，适用于连接属性有序的场景；若关系已排序，则可跳过排序阶段，直接进入归并阶段。

c) 子问题：描述两阶段锁协议，并证明其保证冲突可串行化但不保证无死锁（4分）

英文问题

Describe the Two-Phase Locking (2PL) protocol, prove that it ensures conflict-serializable schedules, and explain why it does not ensure freedom from deadlocks.

中文问题

描述两阶段锁（2PL）协议，证明其保证冲突可串行化调度，并解释为何它不保证无死锁。

答案

1. 两阶段锁协议的定义

2PL将事务的锁操作分为两个阶段：

- **增长阶段**：事务只能获取锁，不能释放锁；
- **收缩阶段**：事务只能释放锁，不能获取锁。

2. 证明2PL保证冲突可串行化

假设事务按“锁点”（事务获取最后一个锁的时刻）的顺序串行化：

- 对于任意冲突操作（如$T_i \rightarrow T_j$的写-读冲突），T_i的锁点必然早于T_j的锁点（因为T_j需等待T_i释放锁）；
- 按锁点顺序串行化的结果与原调度的冲突操作顺序一致，因此2PL保证冲突可串行化。

3. 2PL不保证无死锁的原因

2PL允许事务在增长阶段持续获取锁，可能出现“循环等待”：

- 例：T_3获取B的排他锁，T_4获取A的共享锁；随后T_3请求A的排他锁（等待T_4），T_4请求B的共享锁（等待T_3）；
- 此时形成$T_3 \leftrightarrow T_4$的循环等待，触发死锁。

d) 子问题：基于延迟数据库修改的恢复流程（4分）

英文问题

For the three log records (a), (b), (c) below, describe the recovery procedure using the **deferred database modification** scheme.

中文问题

针对以下3种日志记录（a）、（b）、（c），描述基于**延迟数据库修改**方案的恢复流程。

日志说明

日志格式：`<事务标识，数据项，旧值，新值>`；`<T start>`表示事务开始，`<T commit>`表示事务提交。

（a）日志的恢复流程

日志：`<T0 start>`→`<T0,A,1000,950>`→`<T0,B,2000,2050>`→`<T0 commit>`→`<T1 start>`→`<T1,C,700,600>`→`<T1 commit>`

- **分析**：T0和T1均已提交；
- **恢复操作**：执行**Redo（重做）**：将A设为950，B设为2050，C设为600。

（b）日志的恢复流程

日志：`<T0 start>`→`<T0,A,1000,950>`→`<T0,B,2000,2050>`→`<T0 commit>`→`<T1 start>`→`<T1,C,700,600>`

- **分析**：T0已提交，T1未提交；
- **恢复操作**：
  - Redo T0：将A设为950，B设为2050；
  - Undo T1：将C恢复为700。

（c）日志的恢复流程

日志：`<T0 start>`→`<T0,A,1000,950>`→`<T0,B,2000,2050>`→`<T1 start>`→`<T1,C,700,600>`

- **分析**：T0和T1均未提交；
- **恢复操作**：执行**Undo（撤销）**：将A恢复为1000，B恢复为2000，C恢复为700。

解析

延迟数据库修改的核心规则：

- 事务提交前，修改仅写入日志，不写入数据库；
- 恢复时，**已提交事务**执行Redo（将日志中的新值写入数据库）；

- **未提交事务**执行Undo（将数据项恢复为日志中的旧值）。

我会把这张图拆成**核心概念+每个算法的逐行解析**，帮你彻底搞懂~

先明确这张图的核心：数据库查询的I/O代价计算

数据库查询时，磁盘操作是主要开销（内存操作可忽略）。磁盘I/O代价包含两部分：

- `t_S`：**寻道时间**（磁盘臂移动到目标块的时间）
- `t_T`：**块传输时间**（将磁盘块读入内存的时间）

这张图是**不同查询算法的I/O代价公式+原理**，针对“用索引/不用索引”“查码属性/非码属性”等场景分类。

先解释图里的符号

- `b_r`：表的**总块数**（表的元组数 ÷ 每个块的元组数）
- `h_i`：索引的**高度**（B+树从根到叶的层数）
- `b`：匹配查询条件的**块数**（满足条件的元组所在的磁盘块数）
- `n`：匹配查询条件的**元组数量**

逐行解析每个算法（按A1~A6分类）

一、A1：Linear Search（线性搜索，即全表扫描）

这是“不用索引，从头到尾扫表”的场景。

1. **普通Linear Search（无条件/范围条件）**
   - **Cost（代价）**：`t_S + b_r * t_T`
   - **Reason（原理）**：
     先执行1次寻道（找到表的第1个块），然后传输表的所有块（共`b_r`个）。
     比如表有100块，就是“1次寻道 + 100次块传输”。

2. **Linear Search（码属性的等值查询）**
   - **Cost（代价）**：平均情况 `t_S + (b_r/2) * t_T`
   - **Reason（原理）**：
     码属性是“唯一标识元组”的属性（比如主键），所以**最多只有1个元组满足条件。**
     平均扫到表的一半就能找到目标，所以块传输数是`b_r/2`；最坏情况还是得扫全表（`b_r`块）。

二、A2：Primary B+-tree Index（主B+树索引，码属性的等值查询）

主索引是“索引键=表的码属性（如主键）”的B+树索引，且表的元组按索引键**有序存储**（聚簇索引）。

- **Cost（代价）**：`(h_i + 1) * (t_T + t_S)`
- **Reason（原理）**：
  步骤1：遍历B+树索引（高度`h_i`），每一层对应1次“寻道+块传输”，共`h_i`次I/O；
  步骤2：找到叶节点后，直接取对应的1个元组（主索引的叶节点直接存元组，或指向元组的块），再执行1次“寻道+块传输”；
  总I/O次数是`h_i + 1`，每次I/O都包含`t_S + t_T`，所以代价是`(h_i+1)*(t_S+t_T)`。

三、A3：Primary B+-tree Index（主B+树索引，非码属性的等值查询）

非码属性是“不唯一”的属性（比如“性别”“部门”），主索引的键是非码属性。

- **Cost（代价）**：`h_i * (t_T + t_S) + b * t_T`
- **Reason（原理）**：
  步骤1：遍历B+树索引（高度`h_i`），共`h_i`次“寻道+块传输”；
  步骤2：主索引的叶节点是**有序的块**（因为表按索引键聚簇存储），满足条件的元组会集中在连续的`b`个块里。所以只需1次寻道（找到第1个块），然后传输这`b`个块（无需额外寻道）；
  因此总代价是“索引遍历的代价 + 传输`b`个块的代价”。

四、A4：Secondary B+-tree Index（辅助B+树索引）

辅助索引是“索引键≠表的码属性”的B+树索引，表的元组**不按索引键存储**（非聚簇索引）。

1. **辅助索引（码属性的等值查询）**
   - **Cost（代价）**：`(h_i + 1) * (t_T + t_S)`
   - **Reason（原理）**：和A2（主索引码查询）类似——索引遍历`h_i`次I/O，找到叶节点后取1个元组（1次I/O），总`h_i+1`次I/O。
2. **辅助索引（非码属性的等值查询）**
   - **Cost（代价）**：`(h_i + n) * (t_T + t_S)`
   - **Reason（原理）**：
     步骤1：遍历B+树索引（高度`h_i`），共`h_i`次“寻道+块传输”；
     步骤2：辅助索引的叶节点存的是“索引键+元组的地址”，但表的元组是随机存储的——**每个满足条件的元组可能在不同的块里**，所以每取1个元组都要执行1次“寻道+块传输”；
     若有`n`个匹配元组，就需要`n`次I/O；
     因此总代价是“索引遍历的`h_i`次I/O + 取`n`个元组的`n`次I/O”。

五、A5：Primary B+-tree Index（主B+树索引，比较查询，比如>、<）

- **Cost（代价）**：和A3完全一样：`h_i * (t_T + t_S) + b * t_T`
- **Reason（原理）**：比较查询的结果是“连续的块”（因为主索引是聚簇的），所以代价和非码属性等值查询一致。

六、A6：Secondary B+-tree Index（辅助B+树索引，比较查询）

- **Cost（代价）**：和A4（辅助索引非码等值）完全一样：`(h_i + n) * (t_T + t_S)`
- **Reason（原理）**：比较查询的结果是`n`个元组，每个元组随机存储，所以每取1个元组都要1次I/O，代价和非码属性等值查询一致。

总结：核心差异是“主索引vs辅助索引”“码vs非码”

场景	核心特点	代价关键
主索引（聚簇）	元组按索引键有序存储，结果块连续	取结果只需1次寻道+传输b个块
辅助索引（非聚簇）	元组随机存储，结果块分散	取n个元组需要n次寻道+块传输
码属性查询	结果最多1个元组	取结果只需1次I/O
非码属性查询	结果多个元组	取结果需传输b个块（主索引）/n次I/O（辅助）

要不要我帮你整理一份**不同场景的代价公式对比表**，方便你直接对照使用？