3. Find the names of all customers who have an account at all branches located in Brooklyn City, in tuple relational calculus.

(10 points)

Answer:

$$\{t \mid \exists\, r \in \text{customer}\ (r[\text{customer-name}] = t[\text{customer-name}]) \wedge$$
$$(\forall\, u \in \textit{branch}(u[\textit{branch-city}] = \text{“Brooklyn”} \Rightarrow$$
$$\exists\, s \in \textit{depositor}\ (t[\textit{customer-name}] = s[\textit{customer-name}]$$
$$\wedge \exists\, w \in \textit{account}\ (\ w[\textit{account-number}] = s[\textit{account-number}]$$
$$\wedge\ w[\textit{branch-name}] = u[\text{branch-name}])))\}$$

4. Is referential integrity checked anywhere in a transaction or just at the end of a transaction? Why? Would you please give an example to illustrate?

(10 points)

Answer:

Referential integrity is only checked at the end of a transaction.

Intermediate steps are allowed to violate referential integrity provided later steps remove the violation.

Otherwise it would be impossible to create some database states, e.g. insert two tuples whose foreign keys point to each other
E.g. spouse attribute of relation
marriedperson(name, address, spouse)

4. 参照完整性是在事务的任何阶段都会检查，还是仅在事务结束时检查？为什么？请举例说明。(10分)

参考答案翻译

参照完整性仅在事务结束时检查。

在事务执行过程中允许暂时违反参照完整性，只要后续操作能消除这种违反即可。否则某些数据库状态将无法实现，例如插入两个互相引用外键的元组。

举例说明：已婚人士关系表marriedperson(name, address, spouse)，其中spouse属性需要参照同表的name属性。

详细解释

检查时机

参照完整性约束确实只在事务提交时进行检查，而不是在事务执行过程中。这是数据库管理系统(DBMS)的通用实现方式。

原因分析

1. **操作可行性**：许多合法操作需要暂时违反参照完整性
   - 示例1：互相引用的配偶记录

```sql
BEGIN;
INSERT INTO marriedperson VALUES ('张三', '北京地址', '李四');  -- 此时李四记录还不存在
INSERT INTO marriedperson VALUES ('李四', '上海地址', '张三');  -- 现在两条记录互相引用
COMMIT;  -- 此时检查完整性，通过
```

   - 示例2：先删除被引用记录再插入替代记录

```sql
BEGIN;
DELETE FROM departments WHERE dept_id = 10;  -- 违反员工表的外键约束
INSERT INTO departments VALUES (10, '新部门');  -- 重新建立有效引用
COMMIT;  -- 最终状态合法
```

2. **性能考虑**：实时检查会显著降低事务性能
3. **逻辑完整性**：事务的原子性要求要么全部成功，要么全部回滚

1 Referencing each other

2 A foreign key constraint is violated(Delete the referenced record before inserting it)

1. **步骤1：删除部门**
   - 删除 `dept_id=10` 的部门记录
   - **此时**：如果有员工属于这个部门，就违反了外键约束（员工表的dept_id指向不存在的部门）
   - **但**：数据库不会立即报错，因为完整性检查被延迟了
2. **步骤2：插入新部门**
   - 重新插入 `dept_id=10` 的新部门记录（可以理解为部门重命名或重组）
   - 外键引用关系被恢复
3. **COMMIT时**
   - 数据库检查参照完整性
   - 发现最终状态中所有外键引用都有效（部门10又存在了）
   - 事务成功提交

7. What is a primary index? What is a secondary index? Is sequential scan using secondary index efficient?

(10 points)

Answer:

- Primary index: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
- Secondary index: an index whose search key specifies an order different from the sequential order of the file.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive.

中文翻译：

7. 什么是主索引？什么是辅助索引？使用辅助索引进行顺序扫描是否高效？

（10 分）

答案：

- 主索引：在一个顺序排序的文件中，其查找键指定文件顺序排序方式的索引。
- 辅助索引：其查找键指定的顺序与文件的顺序排序方式不同的索引。
- 使用主索引进行顺序扫描是高效的，但使用辅助索引进行顺序扫描成本较高（效率低）。

9. We say that an ideal hash function is both uniform and random. What does that statement mean? Will you give some example to illustrate these two concepts?

(10 points)

Answer:

- An ideal hash function is uniform, i.e., each bucket is assigned the same number of search-key values from the set of all possible values.
- Ideal hash function is random, so each bucket will have the same number of records assigned to it irrespective of the actual distribution of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
  For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned.

我们说理想的哈希函数兼具均匀性和随机性。这句话是什么意思？你能举些例子来说明这两个概念吗？

- 理想的哈希函数是均匀的，即从所有可能的值集合中，每个桶被分配相同数量的查找键值。
- 理想的哈希函数是随机的，因此无论文件中查找键值的实际分布如何，每个桶都会被分配相同数量的记录。
- 典型的哈希函数对查找键的内部二进制表示进行计算。例如，对于字符串类型的查找键，可将字符串中所有字符的二进制表示相加，再对桶的数量取模并返回结果。

均匀性(Uniformity)解释

定义：均匀性意味着哈希函数能将所有可能的键值**均匀分布**到各个哈希桶中。

随机性(Randomness)解释

定义：随机性指哈希结果**不受输入键值分布模式影响**，表现如随机函数。

均匀性 (Uniform) 示例

想象把全班50个同学分配到5个小组：

好哈希：

- 每组分配10人（完美均匀）
- 如：学号 % 5（学号尾数0-4分别去第1-5组）

差哈希：

- 第1组45人，其他组各1-2人（极不均匀）
- 如：按姓氏首字母A-M去第1组，其余去其他组

随机性 (Random) 示例

用生日月份分组：

好哈希（随机性强）：

- (月份 × 学号) % 5
- 结果：1月生日的同学会随机分散到所有组

差哈希（模式明显）：

- 直接用月份 % 5
- 结果：1月全在第1组，2月全在第2组...（受原始数据规律影响）

10. For conjunctive selection such as:

Conjunction: $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$

We can firstly use corresponding index for each condition, and take intersection of all the obtained sets of record pointers. Then fetch records from file.
If some conditions do not have appropriate indices, we can apply test in memory.

Does this method also work for disjunctive selection such as

Disjunction: $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$?

(10 points)

Answer:

- Applicable if all conditions have available indices. Otherwise we prefer to use linear scan.
- Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
- Then fetch records from file.

10. 对于诸如以下这样的合取选择操作：

合取：$\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \theta_n}(r)$

我们可以首先对每个条件使用对应的索引，然后对所有得到的记录指针集合求交集。之后从文件中获取记录。

如果某些条件没有合适的索引，我们可以在内存中进行测试。

这种方法是否也适用于诸如以下这样的析取选择操作呢？

析取：$\sigma_{\theta_1 \vee \theta_2 \vee \ldots \theta_n}(r)$？（10 分）

答案：

- 若所有条件都有可用索引，则适用。否则，我们更倾向于使用线性扫描。
- 对每个条件使用对应的索引，然后对所有得到的记录指针集合求并集。
- 之后从文件中获取记录。

索引如何加速查询?

合取查询(AND)的索引使用

```sql
SELECT * FROM products
WHERE category = '电子产品' AND price < 1000;
```

执行过程:

1. 使用category索引找到所有'电子产品'的记录指针集合A
2. 使用price索引找到所有price<1000的记录指针集合B
3. 对A和B取**交集**得到最终结果指针
4. 根据指针从数据文件读取完整记录

析取查询(OR)的索引使用

```sql
SELECT * FROM products
WHERE category = '电子产品' OR price < 100;
```

执行过程:

1. 使用category索引找到'电子产品'的记录指针集合A
2. 使用price索引找到price<100的记录指针集合B
3. 对A和B取**并集**得到最终结果指针
4. 根据指针从数据文件读取完整记录

为什么合取(AND)查询可以部分使用索引而析取(OR)查询不行

核心原因：结果集规模差异

合取(AND)查询特性

1. **结果集越来越小**：每个AND条件都过滤掉一部分数据
   - 即使只有部分条件使用索引，也能显著减少需要检查的记录数
   - 示例：10万条数据 → 索引过滤后剩1万条 → 内存过滤剩100条
2. **部分索引工作流程**：

```sql
SELECT * FROM products
WHERE category = '电子产品'(有索引) AND price < 1000(无索引);
```

   - 先用category索引快速定位1万条"电子产品"
   - 然后在内存中对这1万条检查price条件

析取(OR)查询特性

1. **结果集越来越大**：每个OR条件都会增加结果数据
   - 即使大部分条件使用索引，最终可能仍需读取大部分数据
   - 示例：10万条数据 → 条件A返回3万条 → 条件B返回4万条 → 并集可能有5万条
2. **全索引要求原因**：

```sql
SELECT * FROM products
WHERE category = '电子产品'(有索引) OR price < 100(无索引);
```

   - 用category索引找到3万条"电子产品"
   - 无price索引必须全表扫描10万条检查price条件
   - 最终需要读取10万条(全表)而不是3万条

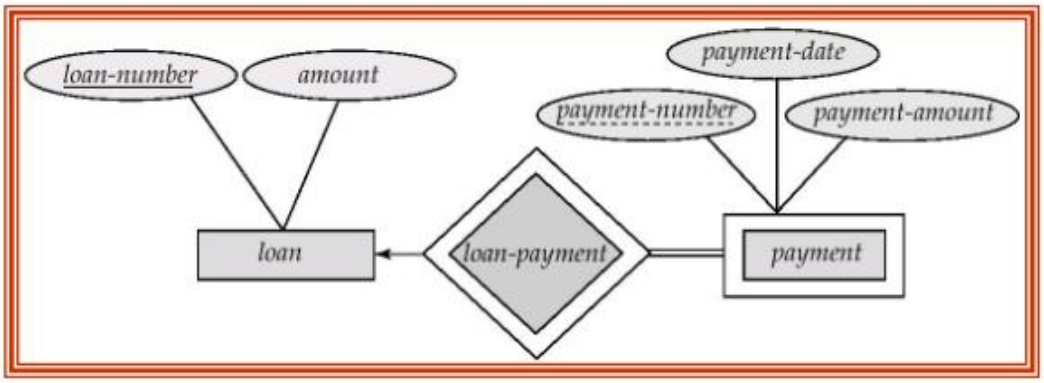
1. **AND的短路特性**：
   - 只要有一个条件过滤性很强，整体效率就高
   - 可以"先索引后过滤"分阶段处理
2. **OR的累加特性**：
   - 必须获取所有条件的完整结果集
   - 任一条件无索引就会拖累整体效率

1. What is a weak entity set? What is a strong entity set? Draw an E-R diagram to illustrate that the weak entity set payment depends on the strong entity set loan via the relationship set loan-payment.

(10 points)

Answer:

An entity set that may not have sufficient attributes to form a primary key is called a weak entity set. An entity set that has a primary key is termed a strong entity set.



1. 什么是弱实体集？什么是强实体集？绘制一个 E-R 图来说明弱实体集 "payment（还款）" 如何通过关系集 "loan - payment（贷款 - 还款）" 依赖于强实体集 "loan（贷款）"。（10 分）

答案：

可能不具备足够属性来构成主键的实体集被称为弱实体集。拥有主键的实体集被称为强实体集。（随后是对应

7. Please compare dense index and sparse index. Is it meaningful to organize a sparse secondary index?

(10 points)

Answer:

- Dense index — Index record appears for every search-key value in the file.
- Sparse Index — Contains index records for only some search-key values.

- Compared with dense index, sparse index is applicable only when records are sequentially ordered on search-key. Sparse index uses less space and less maintenance overhead for insertions and deletions. And generally, sparse index is slower than dense index for locating records.

- Secondary indices have to be dense.

翻译

7. 请比较稠密索引（dense index）和稀疏索引（sparse index）。组织一个稀疏的辅助索引（secondary index）有意义吗？（10 分）

答案：

- 稠密索引 —— 文件中每个搜索键值（search - key value）都有对应的索引记录。
- 稀疏索引 —— 仅对部分搜索键值包含索引记录。
- 与稠密索引相比，稀疏索引仅适用于记录按搜索键顺序排列的情况。稀疏索引占用空间更少，插入和删除操作的维护开销也更小。而且一般来说，在定位记录时，稀疏索引比稠密索引速度慢。
- 辅助索引必须是稠密的。

9. To preserve integrity of data, database system must ensure the ACID properties of a transaction. What do they mean? Please explain.

(10 points)

Answer:

- **Atomicity**. Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency**. Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation**. Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
    - That is, for every pair of transactions Ti and Tj, it appears to Ti that either Tj, finished execution before Ti started, or Tj started execution after Ti finished.
- **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

9. 为了维护数据的完整性，数据库系统必须确保事务的 ACID 属性。它们分别是什么意思？请解释。（10 分）

答案：

- 原子性（Atomicity）：事务的所有操作要么都完整地反映在数据库中，要么都不反映。
- 一致性（Consistency）：孤立地执行一个事务时，能保持数据库的一致性。
- 隔离性（Isolation）：尽管多个事务可能并发执行，但每个事务必须感知不到其他并发执行的事务。中间的事务结果必须对其他并发执行的事务隐藏。也就是说，对于每一对事务 Ti 和 Tj，在 Ti 看来，要么 Tj 在 Ti 开始前就完成了执行，要么 Tj 在 Ti 完成后才开始执行。
- 持久性（Durability）：一个事务成功完成后，它对数据库所做的更改会持久保留，即便发生系统故障。

10. Multiple transactions are allowed to run concurrently in a database system. Why do we need the concurrency of transactions? What are the advantages?

(10 points)

Answer:

- increased processor and disk utilization, leading to better transaction throughput: one transaction can be using the CPU while another is reading from or writing to the disk
- reduced average response time for transactions: short transactions need not wait behind long ones.

10. 数据库系统中允许多个事务并发运行。我们为何需要事务的并发性？有哪些优势？（10 分）

答案：

- 提高处理器和磁盘的利用率，进而提升事务吞吐量：一个事务可以使用 CPU，同时另一个事务可以进行磁盘读写 。
- 缩短事务的平均响应时间：短事务无需在长事务之后等待 。

1. Explain the distinctions among the terms primary key, candidate key, and super key. (10 points)

Answer:

A superkey is a set of one or more attributes that, taken collectively, allows us to identify uniquely an entity in the entity set. A superkey may contain extraneous attributes. If K is a superkey, then so is any superset of K. A superkey for which no proper subset is also a superkey is called a candidate key. It is possible that several distinct sets of attributes could serve as candidate keys. The primary key is one of the candidate keys that is chosen by the database designer as the principal means of identifying entities within an entity set.

解释“主键（primary key）”、“候选键（candidate key）”和“超键（super key）”这些术语之间的区别。（10 分）

答案：

超键是一个由一个或多个属性组成的集合，这些属性集合起来能够让我们唯一地识别实体集中的一个实体。超键可能包含多余的属性。如果 K 是一个超键，那么 K 的任何超集也是超键。若一个超键的任何真子集都不是超键，那么这个超键就被称为候选键。可能有几个不同的属性集合都可以作为候选键。主键是数据库设计者从候选键中选定的、作为在实体集中识别实体的主要手段的那个候选键 。

3. In tuple relational calculus, please form an expression to find the names of all customers who have a loan at the Perryridge branch, but no account at any branch of the bank.

(10 points)

Answer:

$$\{t \mid \exists s \in borrower(\, t[customer\text{-}name] = s[\text{customer-name}] \wedge \exists u \in loan(u[branch\text{-}name] = \text{“Perryridge”} \wedge u[loan\text{-}number] = s[\text{loan-}number])) \wedge \textbf{not}\ \exists v \in depositor\,(v[customer\text{-}name] = t[\text{customer-name}])\}$$

4. Analyze the assertion below and describe its purpose.

```
create assertion balance-constraint check
    (not exists (
        select * from loan
          where not exists (
            select *
                from borrower, depositor, account
                where loan.loan-number = borrower.loan-number
                        and borrower.customer-name = depositor.customer-name
                        and depositor.account-number = account.account-number
                        and account.balance >= 1000)))
```

(10 points)

Answer:

Every loan has at least one borrower who maintains an account with a minimum balance or $1000.00

答案：

每笔贷款至少有一个借款人，该借款人持有一个余额至少为 1000 美元的账户 。

5. When we decompose a relation schema R with a set of functional dependencies F into R1, R2,….., Rn, we want the decomposition to satisfy three criteria. What are these three criteria? Describe them briefly.

(10 points)

Answer:

- Lossless-join decomposition: Otherwise decomposition would result in information loss.
- No redundancy: The relations R_i preferably should be in either Boyce-Codd Normal Form or Third Normal Form.
- Dependency preservation: Let F_i be the set of dependencies in F^+ that include only attributes in R_i.
  - Preferably the decomposition should be dependency preserving, that is, $(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$
  - Otherwise, checking updates for violation of functional dependencies may require computing joins, which is expensive.

5. 当我们使用一组函数依赖 F 将关系模式 R 分解为 $R_1, R_2, \ldots, R_n$ 时，希望分解满足三个标准。这三个标准是什么？简要描述它们。（10 分）

答案：

- **无损连接分解（Lossless - join decomposition）**：否则分解会导致信息丢失。
- **无冗余（No redundancy）**：关系 R_i 最好应处于博伊斯 - 科德范式（Boyce - Codd Normal Form）或第三范式（Third Normal Form）。
- **依赖保持（Dependency preservation）**：设 F_i 是 F^+（F 的闭包）中仅包含 R_i 属性的依赖集。最好分解是依赖保持的，即 $(F_1 \cup F_2 \cup \cdots \cup F_n)^+ = F^+$；否则，检查更新是否违反函数依赖可能需要进行连接计算，成本很高。

7. Why do we need index mechanism in DBMS? Somebody said that the more the indices, the better the performance. Do you agree with him? Why?

(10 points)

Answer:

- Indices offer substantial benefits when searching for records.

- I don't agree with him. Because when a file is modified, every index on the file must be updated. Updating indices imposes overhead on database modification.

7. 为什么数据库管理系统（DBMS）中需要索引机制？有人说 "索引越多，性能越好"。你同意这种说法吗？为什么？（10 分）

答案：

- 索引在查找记录时能带来显著好处。
- 我不同意这种说法。因为当文件被修改（插入、删除、更新等操作）时，该文件上的每个索引都必须被更新。更新索引会给数据库修改操作带来额外开销（如时间、资源等成本）。

8. Would you please compare B-tree index with B^+-tree index. What are the advantages and disadvantages of B-tree index? Can we conclude that B-tree index is better than B^+-tree index?

(10 points)

Answer:

- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B+-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B+-Tree
  - Insertion and deletion more complicated than in B+-Trees
  - Implementation is harder than B+-Trees.
- Typically, advantages of B-Trees do not out weigh disadvantages.

- **B - 树索引的优点**：
  - 与对应的 B^+ - 树相比，可能使用更少的树节点。
  - 有时有可能在到达叶节点之前就找到搜索键值。
- **B - 树索引的缺点**：
  - 只有一小部分搜索键值能在早期找到。
  - 非叶节点更大，所以扇出（fan - out，节点能指向的子节点数量）减少。因此，B - 树通常比对应的 B^+ - 树深度更大。
  - 插入和删除操作比在 B^+ - 树中更复杂。
  - 实现难度比 B^+ - 树更高。
- 通常情况下，B - 树的优点无法抵消其缺点（即不能得出 B - 树索引更好的结论）。

9. There are five transaction states: Active, Partially committed, Failed, Aborted, and Committed. Please draw the state-transformation-diagram and explain all the transformation processes.

(10 points)

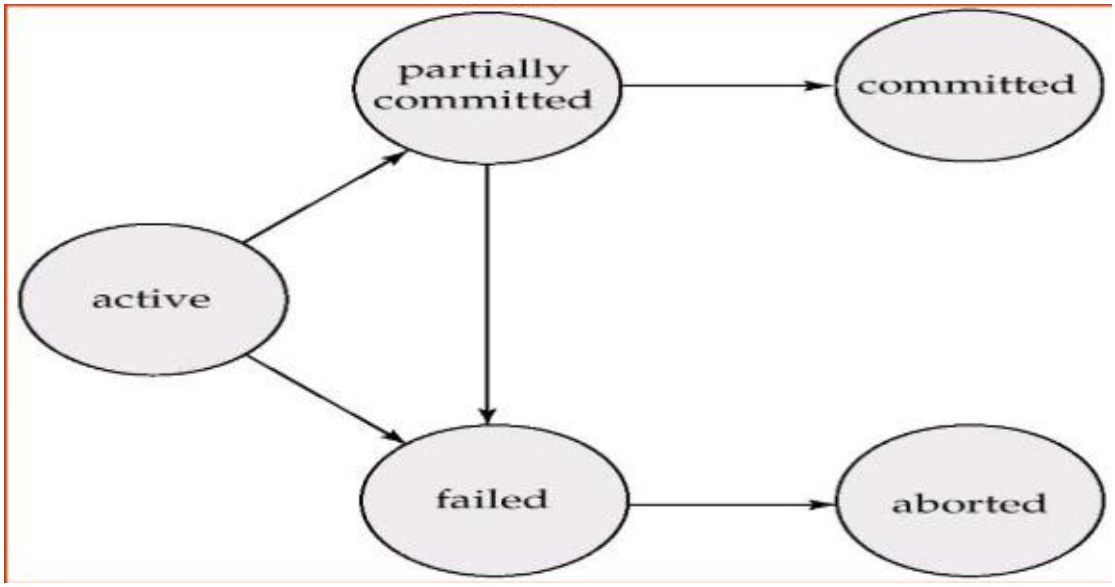
Answer:

- **Active**, the initial state; the transaction stays in this state while it is executing.
- **Partially committed**, after the final statement has been executed.
- **Failed**, after the discovery that normal execution can no longer proceed.
- **Aborted**, after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction – only if no internal logical error
  - kill the transaction
- **Committed**, after successful completion.

9. 事务有五种状态：活动态（Active）、部分提交态（Partially committed）、失败态（Failed）、中止态（Aborted）和提交态（Committed）。请绘制状态转换图并解释所有转换过程。（10 分）

答案：

- **活动态（Active）**：初始状态；事务执行期间处于此状态。
- **部分提交态（Partially committed）**：执行完最后一条语句后进入此状态。
- **失败态（Failed）**：发现无法继续正常执行后进入此状态。
- **中止态（Aborted）**：事务回滚且数据库恢复到事务开始前状态后进入此状态。中止后有两个选择：
  - 重启事务 —— 仅当无内部逻辑错误时可行。
  - 终止事务。
- **提交态（Committed）**：事务成功完成后进入此状态。



10. Is the following schedule conflict serializable? Why?

T3	T4
read(Q)	
	write(Q)
write(Q)	

(10 points)

Answer:

It is not conflict serializable. We are unable to swap instructions in the schedule to obtain either the serial schedule < T3, T4 >, or the serial schedule < T4, T3 >.

它不是冲突可串行化的。我们无法通过交换调度中的指令来得到串行调度<T3, T4>或串行调度<T4, T3>。