期末模拟试题 A

一、 (8 分)

数据独立性是借助数据库管理数据的一个显著优点, 请解释何谓物理独立性？

答：

物理独立性是指用户程序与数据存储相互独立。具体解释如下：

定义与内涵：物理独立性是指用户的应用程序与存储在磁盘上的数据库中数据是相互独立的。数据在磁盘上的存储方式，包括组织结构、存取方法等，由数据库管理系统（DBMS）负责管理。

实现机制：当数据库的存储结构发生改变时，数据库管理员会对模式/内模式映像做相应调整，以确保模式保持不变。这样，用户的应用程序就无需进行修改，从而保证了数据与程序的物理独立性。

物理独立性的存在，使得数据的物理存储方式可以灵活调整，而不会影响应用程序的正常运行，这是数据库系统相较于文件系统的一个重要优势。

二、 (8 分)

请给出自主存取控制的定义。

答：

自主存取控制方法（Discretionary Access Control，DAC）是一种基于主题定制的存取控制方式。在这种方式下，主题的拥有者可以自主地选择特定的用户或用户组授予某些权限。特别是，该方式使主题拥有者透过特定标签或时间参数给予他人必须的权限。

三、 (8 分)

何谓静态转储？

答：

静态转储是指在系统中无运行事务时进行的转储操作。在这种转储方式下，数据库在转储开始时处于一致状态，并且在转储期间不允许对数据库进行任何存取或修改活动。静态转储的目的是确保转储的数据在转储结束时是完整的，从而保证数据的有效性。然而，这种转储方式会降低数据库的可用性，因为转储必须在用户事务结束后进行，新的事务必须等待转储结束才能执行。

四、 (10 分)

什么是封锁粒度? 并分析不同的封锁粒度对于并发性与系统开销的影响。

答：

封锁的数据目标可以是这样一些逻辑单元：数据库、表、记录、字段等，封锁数据目标的大小叫封锁粒度。封锁的粒度小，并发度高，但封锁机构复杂，系统开销大。反之，封锁的粒度大，并发度小，但封锁机构简单，系统开销小。

五、 (14 分)

设有关系模式 R(U,F), 其中 U={A,B,C,D,E,G}，

函数依赖集 F={B->G, CE->B, C->A, CE->G, B->D, C->D}

（1）求 F 的最小函数依赖集 Fmin , 请给出求解的主要过程。 (3 分)

答：

为了找到给定函数依赖集 F 的最小函数依赖集 Fmin，我们需要遵循以下步骤：

确定每个属性的闭包：

- 首先计算每个单独属性的闭包，以便理解每个属性可以推导出的其他属性。

删除冗余的函数依赖：

- 如果一个函数依赖可以从其他函数依赖推导出来，那么它就是冗余的，可以被删除。
- 使用推理规则，如合并规则、分解规则和伪传递规则，来简化函数依赖集。

确保每个非主属性都由候选键完全决定：

- 找出候选键，并确保每个非主属性都由候选键决定。

步骤 1：计算属性的闭包

首先，我们需要计算每个属性的闭包。闭包是给定属性通过函数依赖可以推导出的所有属性的集合。

- 对于 B：
- B^+ = {B, G, D} （由 B->G 和 B->D）
- 对于 CE：
- CE^+ = {C, E, B, G, D, A} （由 CE->B, B->G, B->D, CE->G, C->A）
- 对于 C：
- C^+ = {C, A, D} （由 C->A 和 C->D）

步骤 2：删除冗余的函数依赖

现在我们来检查每个函数依赖，看它们是否冗余：

- B->G：这是必要的，因为 B 不能通过其他依赖推导出 G。
- CE->B：这是必要的，因为 CE 不能通过其他依赖推导出 B。
- C->A：这是必要的，因为 C 不能通过其他依赖推导出 A。
- CE->G：这是冗余的，因为 CE->B 和 B->G 可以推导出 CE->G。
- B->D：这是必要的，因为 B 不能通过其他依赖推导出 D。
- C->D：这是必要的，因为 C 不能通过其他依赖推导出 D。

步骤 3：构建最小函数依赖集

根据以上分析，最小函数依赖集 Fmin 为：

Fmin = {B->G, CE->B, C->A, B->D, C->D}

这就是给定函数依赖集 F 的最小函数依赖集 Fmin。

（2）求出 R 的候选码，请给出求解过程。 (3 分)

答：

为了求出关系模式 R(U,F)的候选码，我们需要通过一系列步骤来找出能够唯一标识关系中每个元组的属性或属性集。以下是详细的求解过程：

步骤 1：确定必须包含的属性

首先，我们观察函数依赖集 F，找出所有作为决定者的属性，并考虑它们是否可能是候选码的一部分。

- B->G：B 决定 G，但 B 不能单独成为候选码，因为还有其他属性（如 A、D）不能由 B 单独决定。
- CE->B：CE 决定 B，这意味着 CE 可能是一个候选码或者候选码的一部分，因为通过 B，我们可以进一步决定其他属性。
- C->A：C 决定 A，但 C 不能单独成为候选码，因为还有其他属性（如 B、D、G）不能由 C 单独决定。
- CE->G：这是冗余的，因为 CE->B 和 B->G 可以推导出 CE->G。
- B->D：B 决定 D，但这并不增加 B 作为候选码的可能性，因为 B 仍然不能决定所有属性。
- C->D：C 决定 D，但同样，C 不能单独成为候选码。

步骤 2：使用属性闭包来验证候选码

接下来，我们使用属性闭包的概念来验证哪些属性集能够决定关系中的所有属性。

- 我们已经知道 CE 能够决定 B，而 B 能够决定 G 和 D。因此，CE 的闭包至少包含{B, G, D}。
- 由于 C 还能够决定 A，所以 CE 的闭包实际上包含{A, B, C, D, E, G}中的所有属性。
- 这意味着 CE 能够决定关系中的所有属性，因此 CE 是一个候选码。

步骤 3：确认没有更小的候选码

- 我们需要检查是否存在比 CE 更小的属性集也能成为候选码。
- 单独一个 C 或 E 都不能决定所有属性，因此它们不是候选码。
- 任何小于 CE 的属性集（如 C、E、CB（B 是冗余的，因为 CE 已经能决定 B）等）也都不能决定所有属性。

结论

因此，关系模式 R(U,F)的候选码是{CE}。这个属性集能够唯一标识关系中的每个元组，且没有更小的属性集能够满足这一条件。

（3）判断 R 中哪些属性是主属性, 哪些是非主属性? (2 分)

答：

在关系模式 R(U,F)中，主属性是包含在候选码中的属性，而非主属性则是不包含在候选码中的属性。

首先，我们已经知道关系模式 R 的候选码是{C, E}（这是根据之前的分析得出的，但也可以在这里重新验证）。候选码是能够唯一标识关系模式中每个元组的属性或属性集。

现在，我们来确定哪些属性是主属性，哪些是非主属性：

主属性：

- 根据候选码的定义，候选码中的属性就是主属性。
- 因此，在 R 中，主属性是 C 和 E。

非主属性：

- 非主属性是那些不包含在候选码中的属性。
- 在 R 中，除了 C 和 E 之外的其他属性都是非主属性。
- 因此，非主属性是 A、B、D 和 G。

综上所述，关系模式 R(U,F)中：

- 主属性是：C, E
- 非主属性是：A, B, D, G

（4）对 R 属于哪一类范式? 为什么? (3 分)

答：

为了判断关系模式 R(U,F)属于哪一类范式，我们需要根据范式的定义来逐步分析。

首先，我们回顾一下范式的定义：

- **第一范式（1NF）**：关系模式中的每个属性都是原子的，即不可再分的。
- **第二范式（2NF）**：关系模式满足 1NF，并且非主属性完全依赖于候选码，而不是依赖于候选码的一部分。
- **第三范式（3NF）**：关系模式满足 2NF，并且非主属性不传递依赖于候选码。

现在，我们来分析关系模式 R(U,F)：

1. 确定候选码：
根据之前的分析，我们已经知道关系模式 R 的候选码是{CE}。

2. 检查是否满足 1NF：
由于题目没有提供属性的具体数据类型，我们通常假设每个属性都是原子的。因此，可以认为 R 满足 1NF。

3. 检查是否满足 2NF：
要满足 2NF，非主属性必须完全依赖于候选码。在 R 中，非主属性有 A、B、D、G。

- A 依赖于 C（C->A），但 C 不是完整的候选码，所以 A 不完全依赖于候选码 CE。
- B 依赖于 CE（CE->B），满足完全依赖。

- D 依赖于 B（B->D），而 B 依赖于 CE，所以 D 传递依赖于 CE，但这不是问题，因为 2NF 只关心非主属性是否直接依赖于候选码。然而，由于存在 C->D，D 也直接依赖于 C，这同样不满足完全依赖。
- G 依赖于 B（B->G），与 D 的情况类似，也不满足完全依赖。

由于存在非主属性不完全依赖于候选码，所以 R 不满足 2NF。

4. 检查是否满足 3NF：

由于 R 不满足 2NF，它自然也不满足 3NF，因为 3NF 是 2NF 的一个更严格的条件。

结论：

关系模式 R(U,F)不满足第二范式（2NF），因此它也不满足第三范式（3NF）。它只满足第一范式（1NF）。这是因为存在非主属性（如 A 和 D）不完全依赖于候选码{CE}。

（5）若 R 未达到 3NF, 请模式分解 R 以达到 3NF, 并保持函数依赖性。(3 分)

答：

为了将关系模式 R(U,F)分解到第三范式（3NF）并保持函数依赖性，我们需要首先识别出那些不满足 3NF 的部分，并将它们分解出来形成新的关系模式。

分析不满足 3NF 的原因：

在 R(U,F)中，我们已经知道候选码是{CE}。现在我们来检查每个非主属性是否满足 3NF 的条件：

- A 依赖于 C（C->A），但 C 不是候选码，所以 A 不满足 3NF 的条件，因为它传递依赖于候选码 CE。
- B 依赖于 CE（CE->B），满足 2NF，但我们还需要检查 B 是否满足 3NF。
- G 依赖于 B（B->G），而 B 依赖于 CE，所以 G 不满足 3NF 的条件，因为它传递依赖于候选码 CE。
- D 依赖于 B（B->D）和 C（C->D），同样不满足 3NF 的条件，因为它也传递依赖于候选码 CE。

分解过程

为了将 R 分解到 3NF，我们可以考虑以下步骤：

分解出包含候选码的关系：

- 创建一个新关系 R1(CE, B)，包含候选码 CE 和 B，以及函数依赖 CE->B。

分解出其他非主属性的关系：

- 创建一个新关系 R2(B, G)，包含 B 和 G，以及函数依赖 B->G。
- 创建一个新关系 R3(C, A)，包含 C 和 A，以及函数依赖 C->A。
- 创建一个新关系 R4(C, D)，虽然 C->D 存在，但为了避免数据冗余，我们可以选择将 D 与 C 一起放在一个新关系中，或者根据实际需求将其与其他关系合并。这里我们选择创建 R4(C, D)。

验证分解结果：

现在我们有以下关系模式：

- R1(CE, B)
- R2(B, G)
- R3(C, A)

- R4(C, D)

我们需要验证这些关系模式是否都满足 3NF：

- R1 中，B 完全依赖于候选码 CE，满足 3NF。
- R2 中，G 完全依赖于 B，且没有传递依赖，满足 3NF。
- R3 中，A 完全依赖于 C，且没有传递依赖，满足 3NF。
- R4 中，D 完全依赖于 C，且没有传递依赖，满足 3NF。

此外，我们还需要确保原始关系模式 R 中的所有函数依赖都能在新的关系模式中得到保持。通过检查，我们可以发现：

- 原始关系中的 B->G 在 R2 中得到了保持。
- 原始关系中的 CE->B 在 R1 中得到了保持。
- 原始关系中的 C->A 在 R3 中得到了保持。
- 原始关系中的 C->D 在 R4 中得到了保持。
- 原始关系中的 CE->G 可以通过 R1 和 R2 的组合来推导出来（即先通过 CE->B 在 R1 中找到 B，再通过 B->G 在 R2 中找到 G）。
- 原始关系中的 B->D 在 R2 中得到了保持，但也可以通过 R1 和 R2 的组合来推导出来。

因此，我们成功地将关系模式 R(U,F)分解到了 3NF，并且保持了所有的函数依赖性。

六、（16 分）

一、已知某部门的销售系统，其部分关系模式如下：

- Orders(OrderID, CustomerID, OrderDate, RequiredDate, ShippedDate);

含义：订单表(订单编号, 客户编号, 订购日期, 预计到达日期, 发货日期);

- OrderDetails(OrderID, ProductID, UnitPrice, Quantity)

含义：订单明细表(订单编号, 产品编号, 单价, 订购数量) ;

- Products(ProductID, ProductName, SupplierID, CategoryID, UnitsInStock)

含义：产品表(产品编号, 产品名称, 供应商编号, 类型编号, 库存数量)

- Customers(CustomerID, CompanyName, ContactName, ContactTitle, Address, Phone)

含义：客户表(客户编号, 所在公司名称, 客户姓名, 客户头衔, 联系地址, 电话)

请完成下面的 SQL 查询：

（1）查询订单编号为“10248”的订单的发货日期。(4 分)

```
SELECT ShippedDate
FROM Orders
WHERE OrderID = '10248';
```

（2）查询各个订单的订单金额，列标题显示为:订单编号，订单金额。(4 分)

答：为了查询各个订单的订单金额，并显示订单编号和对应的订单金额，我们需要结合 Orders 表和 OrderDetails 表。Orders 表提供了订单的框架，而 OrderDetails 表则详细列出了每个订单中的产品、单价以及数量。订单金额是通过将 OrderDetails 表中的每个产品的 UnitPrice（单价）乘以 Quantity（数量），然后对同一个 OrderID（订单编号）下的所有产品进行求和得

到的。

```
SELECT
    Orders.OrderID AS  订单编号,
    SUM(OrderDetails.UnitPrice * OrderDetails.Quantity) AS  订单金额
FROM
    Orders JOIN OrderDetails ON Orders.OrderID = OrderDetails.OrderID
GROUP BY
    Orders.OrderID;
```

（3）更新订单明细表，将库存量少于 500 的产品的单价上调 10%。(4 分)

答：为了更新订单明细表（OrderDetails）中库存量少于 500 的产品的单价，我们首先需要确定哪些产品的库存量少于 500。这一信息存储在 Products 表中。接着，我们需要根据这些产品的 ProductID 来找到 OrderDetails 表中对应的记录，并将它们的 UnitPrice 上调 10%。

以下是一个 SQL 查询语句，用于实现这一更新操作：

```
UPDATE OrderDetails
SET UnitPrice = UnitPrice * 1.10
WHERE ProductID IN (
    SELECT ProductID
    FROM Products
    WHERE UnitsInStock < 500
);
```

（4）查询这样订单的订单编号：同一个订单同时购买了所有产品类型编号为“DT” 的产品。(4 分)

答：

这个查询的逻辑解释如下：

连接 OrderDetails 和 Products 表：

- 我们通过 JOIN 操作将 OrderDetails 表和 Products 表连接起来，基于它们共有的 ProductID 字段。

筛选“DT”类型的产品：

- 在 WHERE 子句中，我们只选择那些 CategoryID 为“DT”的产品记录。

按订单编号分组：

- 使用 GROUP BY 子句按 OrderID 分组，这样我们可以对每个订单进行独立的处理。

使用 HAVING 子句进行过滤：

- HAVING 子句用于过滤分组后的结果。在这里，我们检查每个订单中不同“DT”类型产品的数量是否等于所有“DT”类型产品的总数。
- 子查询(SELECT COUNT(DISTINCT ProductID) FROM Products WHERE CategoryID = 'DT')计算了所有“DT”类型产品的不同数量。

- HAVING COUNT(DISTINCT p.ProductID) = ...确保了当前分组（即当前订单）中的“DT”类型产品数量与总数相等。

如果满足这个条件，那么该订单就同时购买了所有产品类型编号为“DT”的产品，并且其订单编号会被包含在查询结果中。

```
SELECT od.OrderID
FROM OrderDetails od
JOIN Products p ON od.ProductID = p.ProductID
WHERE p.CategoryID = 'DT'
GROUP BY od.OrderID
HAVING COUNT(DISTINCT p.ProductID) = (
    SELECT COUNT(DISTINCT ProductID)
    FROM Products
    WHERE CategoryID = 'DT'
);
```

七、（12 分）

设有一个金融数据库，包括 client、bcard、insurance、property 4 个关系模式：
关系模式 **client**（cnum 客户编码 PRIMARY KEY，cname 客户名称，cphone 客户电话，cpassword 客户登录密码，cage 客户年龄，cgender 客户性别）；
关系模式 **bcard**（bnum 银行卡号 PRIMARY KEY，btype 银行卡类型，cnum 所属客户编号 注：FOREIGN KEY 引用自 client 表的 cnum 字段）；
关系模式 **insurance**（iname 保险名称，inum 保险编号 PRIMARY KEY，iamount 保险金额，iperson 适用人群，iyear 保险年限，iproject 保障项目）；
关系模式 **property**（id 资产编号 PRIMARY KEY，cnum 客户编号 说明：FOREIGN KEY 引用自 client 表的 cnum 字段，pifid 商品编号 说明：FOREIGN KEY 引用自 insurance 表的 inum 字段，quan 商品数量，ptime 购买时间）。

client（客户）表

字段名称	说明
cnum	客户编码PRIMARY KEY
cname	客户名称
cphone	客户电话
cpassword	客户登录密码
cage	客户年龄
cgender	客户性别

bcard（银行卡）表

字段名称	说明
bnum	银行卡号PRIMARY KEY
btype	银行卡类型
cnum	所属客户编号 注：**FOREIGN KEY**引用自client表的 cnum 字段。

insurance（保险）表

字段名称	说明
iname	保险名称
inum	保险编号PRIMARY KEY
iamount	保险金额
iperson	适用人群
iyear	保险年限
iproject	保障项目

property（资本）表

字段名称	说明
id	资产编号PRIMARY KEY
cnum	客户编号 说明：**FOREIGN KEY**引用自client表的cnum字段。
pifid	商品编号 说明：**FOREIGN KEY**引用自insurance表的inum字段。
quan	商品数量
ptime	购买时间

用关系代数完成如下查询：

（1）查询购买了 10 份保险（商品数量为 10）的客户编号；（4 分）

$\pi_{cnum}(\sigma_{quan=10}(property))$

（2）查询没有购买适用于儿童的保险的客户编号；（4 分）

1)从 insurance 关系中选择适用于儿童的保险：

$(\sigma_{\{iperson='儿童'\}}(insurance))$

这会得到一个包含所有适用于儿童的保险的记录集。

2)投影出这些保险的编号：

$(\pi_{\{inum\}}(\sigma_{\{iperson='儿童'\}}(insurance))$

这会得到一个包含所有适用于儿童的保险的编号的集合。

3)找出购买了这些保险的客户编号：

我们需要将 property 关系与上一步得到的保险编号集合进行连接，然后投影出客户编号。这里我们使用半连接（semijoin）的概念，即只保留在 property 中有对应保险编号的记录，并投影出这些记录的客户编号。

$$\pi_{\{cnum\}}(((\sigma_{\{iperson='儿童'\}}(insurance))\bowtie_{\{inum=pifid\}}(property)))$$

这会得到一个包含所有购买了适用于儿童的保险的客户编号的集合。

4)找出没有购买这些保险的客户编号：

最后，我们需要从 client 关系中的所有客户编号中排除上一步得到的购买了儿童保险的客户编号。这可以通过差集（difference）操作来实现。

$$\pi_{\{cnum\}}(client)-\pi_{\{cnum\}}(((\sigma_{\{iperson='儿童'\}}(insurance))\bowtie_{\{inum=pifid\}}(property)))$$

（3）查询至少拥有客户编号为 001 所拥有银行卡类型的客户姓名和电话。（4 分）

1)首先，我们需要获取客户编号为 001 的银行卡类型：

从 bcard 关系中，我们可以选择出 客户编号 为 001 的记录，并获取其 银行卡类型。

$$\sigma_{所属客户编号='001'}(bcard)$$

这个操作的结果是一个包含银行卡类型和客户编号的集合。

2)接着，我们需要根据银行卡类型来查找所有拥有该银行卡类型的其他客户：

为了实现这一点，我们可以将从第 1 步中得到的银行卡类型与 bcard 关系进行连接（自然连接），得到拥有相同银行卡类型的客户列表。

$$\sigma_{cnum='001'}(bcard)\bowtie (bcard)$$

这里，我们对 bcard 和 bcard_001 进行连接，连接的条件是银行卡类型相同。连接结果会包含所有拥有与客户编号为 001 相同银行卡类型的客户。

3)最后，我们提取这些客户的姓名和电话：

我们需要从 client 关系中提取这些客户的姓名和电话，首先根据 客户编号 进行连接，然后投影出 客户姓名 和 客户电话。假设 client 关系是 client(客户编码, 客户名称, 客户电话, 客户登录密码, 客户年龄, 客户性别)，我们可以进行如下操作：

$$\pi_{cname,cphone}((\sigma_{cnum='001'}(bcard)\bowtie (bcard))\bowtie client)$$

八、（12 分）

设有一个 SPJ 数据库，包括 S、P、J 和 SPJ 共 4 个关系模式：

1. 供应商表由供应商代码 SNO、供应商姓名 SNAME、供应商状态 STATUS、供应商所在城市 CITY 组成：S（SNO, SNAME, STATUS, CITY）主码：SNO
2. 零件表由零件代码（PNO）、零件名（PNAME）、颜色（COLOR）、重量（WEIGHT）组成：P(PNO, PNAME, COLOR, WEIGHT) 主码：PNO
3. 工程项目表 J 由工程项目代码（JNO）、工程项目名（JNAME）、工程项目所在城市（CITY）组成：J(JNO, JNAME, CITY) 主码：JNO
4. 供应情况表 SPJ 由供应商代码 SNO、零件代码（PNO）、工程项目代码（JNO）、

供应数量（QTY）组成：SPJ(SNO, PNO, JNO, QTY)

主码：（SNO, PNO, JNO）

外码：SNO，引用了供应商表的供应商代码 SNO

PNO，引用了零件表的零件代码 PNO

JNO，引用了工程项目表的工程项目代码 JNO

用关系代数完成如下查询：供应一汽工程（工程项目名为'一汽'）红色零件（零件的颜色为'红'）的供应商代码 SNO。

（1）画出上述查询的关系代数语法树；（4 分）

（2）对题 1 中的关系代数语法树进行优化，画出优化后的语法树。（8 分）

首先，我们来构建关系代数查询，以找出供应一汽工程（工程项目名为'一汽'）红色零件（零件的颜色为'红'）的供应商代码SNO。

关系代数查询可以表示如下：

```text
π SNO (σ JNAME = '一汽' ∧ COLOR = '红' (SPJ ⋈ (J ⋈ (P ⋈ S))))
```

但是，这个查询可以进一步简化，因为我们不需要在所有关系之间都执行笛卡尔积。我们可以按照以下步骤来构建更优化的查询：

1. 首先，从 `J` 关系中选择工程项目名为'一汽'的元组：

```text
J1 := σ JNAME = '一汽' (J)
```

2. 然后，从 `P` 关系中选择颜色为'红'的元组：

```text
P1 := σ COLOR = '红' (P)
```

3. 接下来，我们需要将 `SPJ` 关系与 `J1` 和 `P1` 进行连接，以找出供应一汽工程红色零件的供应商代码。这可以通过两次自然连接来实现：

```text
SPJ1 := SPJ ⋈ J1
SPJ2 := SPJ1 ⋈ P1
```

4. 最后，从 `SPJ2` 中投影出供应商代码SNO：

```text
π SNO (SPJ2)
```

现在，我们来画出关系代数语法树：

原始语法树（未优化）：

```text
        π SNO
         |
        σ JNAME = '一汽' ∧ COLOR = '红'
         |
        ⋈
       /   \
     SPJ    ⋈
          /   \
         J     ⋈
             /   \
            P     S
```

优化后的语法树：

```text
        π SNO
         |
        ⋈ (P1)
       /   \
    ⋈ (J1)  P1: σ COLOR = '红' (P)
   /   \
SPJ   J1: σ JNAME = '一汽' (J)
```

在优化后的语法树中，我们首先分别选择了满足条件的 `J` 和 `P` 关系（即 `J1` 和 `P1`），然后将它们与 `SPJ` 关系进行连接，并最终投影出供应商代码SNO。这样的查询更加高效，因为它避免了不必要的笛卡尔积操作，并且只处理了满足条件的数据。

九、（12 分）

为某公司建立一个管理数据库，要求如下：

①该公司有多个部门，各部门有许多员工，但每一个员工仅属于一个部门；

②每个部门可以管理多个工程，每个工程由唯一的部门负责管理；

③每个员工可在多项工程中工作，工种可以是普通工作或管理工作等，每项工程可有多个员工做工；

④“部门”有属性：部门号、部门名、部门负责人、电话；

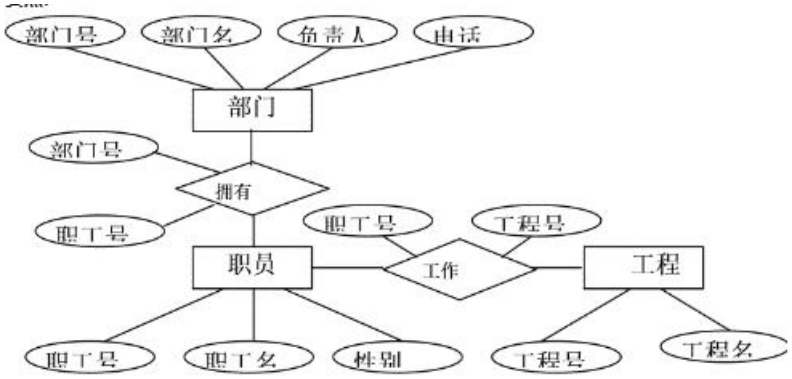
“员工”有属性：员工号，员工名，性别；

“工程”有属性：工程号，工程名，预算

根据以上要求，完成：

（1）画出 E-R 图，并注明属性和联系类型（6 分）。

（2）将 E-R 图转换成关系模型，并注明主码和外码（6 分）。



下面给出已经转换好的等价关系模型结构。

①部门（部门号，部门名，部门负责人，电话） （主关键字为部门号）

②职员（职工号，职工名，性别，部门号） （主关键字为职工号）

③工程（工程号，工程名，项目负责人） （主关键字为工程号）

④工作（职工号，工程号，工种） [主关键字为（职工号，工程号）]