数据库复习

YouXam

目录

一 概念

超码 超码是一个属性或属性的集合，能够唯一标识关系中每一个元组。

候选键 最小的超码。

二 关系代数

2.1 选择

$\sigma_{\theta(R)}$：从关系 (R) 中选取满足特定条件（谓词）的元组（行）。

例如：$\sigma_{\text{age}>20}(\text{Student})$ 选取 Student 表中年龄大于 20 的元组。

2.2 投影

$\Pi_{A_1,A_2,\ldots,A_n}(R)$：从关系 (R) 中选取特定属性（列）。

例如：$\Pi_{\text{name, age}}(\text{Student})$ 选取 Student 表中的姓名和年龄。

2.3 交、并、差

- $R \cap S$：返回两个关系的交集。
- $R \cup S$：返回两个关系的并集。
- $R - S$：返回属于 R 但不属于 S 的元组。

2.4 笛卡尔积

$R \times S$：返回两个关系的笛卡尔积。

例如：Student × Dept 返回 Student 表和 Dept 表的笛卡尔积。

2.5 连接

2.5.1 等值连接

$R \bowtie_\theta S$：根据连接条件 θ 连接两个关系。

例如：$\text{Student} \bowtie_{\text{Student.DeptID} = \text{Dept.DeptID}} \text{Dept}$ 返回 Student 表和 Dept 表根据 DeptID 连接的结果。

2.5.2 自然连接

$R \bowtie S$：根据两个关系的公共属性连接。

2.5.3 外连接

等值连接和自然连接称为内连接，外连接则包括左外连接、右外连接和全外连接。

- 左外连接：$R \bowtie S$ 返回 R 中的所有元组和 S 中匹配的元组，没有匹配的用 NULL 填充。

例如 Employees ⋈ Departments 返回所有员工的信息和员工的部门信息，即使员工没有部门也会填充 NULL。

- 右外连接：$R \bowtie S$ 返回 S 中的所有元组和 R 中匹配的元组，没有匹配的用 NULL 填充。
- 全外连接：$R \bowtie S$ 返回 R 和 S 中的所有元组，没有匹配的用 NULL 填充。

2.6 重命名

$\rho_{\text{new\_name}}(R)$：重命名关系 R。

例如：$\sigma_{\text{instructor.salary} < \text{d.salary}}(\text{instructor} \times \rho_d(\text{instructor}))$[1]

2.7 聚集

${}_{\text{group\_by}}\gamma_{\text{expr as name}}(R)$：根据属性 group_by 分组，并根据 expr 计算，重命名为 name。

例如：

$$ {}_{\text{dept\_id}}\gamma_{\text{average(salary) as avg\_salary}}(\text{Student}) $$

返回按照 dept_id 分组的平均工资。

三 SQL 语言

3.1 SELECT

```
SELECT [DISTINCT] 列名/表达式 [AS 别名], ...
FROM 表名 [AS 别名]
WHERE 条件
GROUP BY 列名
HAVING 条件
ORDER BY 列名 [ASC|DESC];
```

3.1.1 多关系查询

```
SELECT S.StudentName, D.DeptName
FROM Student S, Department D
WHERE S.DeptID = D.DeptID
  AND D.DeptName = 'Computer Science';
```

此写法在现代 SQL 中常用显式 JOIN(后文介绍)来替代，语义更清晰。

3.2 集合操作：交、差、并

```
-- 并
SELECT StudentID FROM CS_Student
UNION
```

[1] 这个表达式应当简化为 $\text{instructor} \bowtie_{\text{instructor.salary} < \text{d.salary}} \rho_d(\text{instructor})$ 以优化查询。

```
SELECT StudentID FROM Math_Student;

-- 交
SELECT StudentID FROM CS_Student
INTERSECT
SELECT StudentID FROM Math_Student;

-- 差
SELECT StudentID FROM CS_Student
EXCEPT
SELECT StudentID FROM Math_Student;
```

3.3 聚集函数

```
-- 按 DeptID 分组，统计每个部门的学生人数
SELECT DeptID, COUNT(StudentID) AS NumOfStudents
FROM Student
GROUP BY DeptID
HAVING COUNT(StudentID) > 10;  -- 只要学生数 > 10 的部门
```

3.4 嵌套子查询

3.4.1 集合成员资格（IN / NOT IN）

```
SELECT StudentID, StudentName
FROM Student
WHERE DeptID IN (
  SELECT DeptID
  FROM Department
  WHERE DeptName = 'Computer Science'
);
```

3.4.2 集合比较（ANY / ALL / SOME）

```
SELECT S.StudentID
FROM Student S
WHERE S.Age > ANY (
  SELECT Age FROM Student WHERE DeptID = 3
);
```

3.4.3 空关系测试（EXISTS / NOT EXISTS）

```
SELECT DeptID
FROM Department D
WHERE EXISTS (
  SELECT * FROM Student S
```

```
  WHERE S.DeptID = D.DeptID
    AND S.Age > 20
);
```

如果存在至少一条满足 `S.DeptID = D.DeptID AND S.Age > 20` 的记录，则该 DeptID 被选出。

3.4.4 重复元组存在性测试（UNIQUE / NOT UNIQUE）

```
SELECT T.course_id
FROM course AS T
WHERE UNIQUE (SELECT R.course_id
    FROM section AS R
    WHERE T.course_id= R.course_id AND
      R.year = 2017);
```

3.4.5 FROM 子句中的子查询

```
SELECT T.DeptID, T.NumOfSt
FROM (
  SELECT DeptID, COUNT(*) AS NumOfSt
  FROM Student
  GROUP BY DeptID
) AS T
WHERE T.NumOfSt > 10;
```

3.4.6 WITH 子句

```
WITH DeptCount AS (
  SELECT DeptID, COUNT(*) AS NumOfSt
  FROM Student
  GROUP BY DeptID
)
SELECT * FROM DeptCount
WHERE NumOfSt > 10;
```

3.4.7 标量子查询

```
SELECT StudentID,
       (SELECT DeptName
        FROM Department D
        WHERE D.DeptID = S.DeptID) AS DeptName
FROM Student S;
```

3.5 数据的插入、删除和更新

```
-- 普通插入单条记录
INSERT INTO Student(StudentID, StudentName, DeptID, Age)
```

```
VALUES (1001, 'Alice', 3, 19);

-- 通过子查询插入多条
INSERT INTO HonorStudent (StudentID, DeptID)
SELECT StudentID, DeptID
FROM Student
WHERE Age > 20;

-- 删除所有年龄大于 25 的学生
DELETE FROM Student
WHERE Age > 25;

-- 将 DeptID 为 3 的学生年龄全部加 1 岁
UPDATE Student
SET Age = Age + 1
WHERE DeptID = 3;
```

3.6 连接表达式

3.6.1 内连接

```
SELECT S.StudentName, D.DeptName
FROM Student S
INNER JOIN Department D
  ON S.DeptID = D.DeptID
WHERE D.DeptName = 'Computer Science';
```

3.6.2 外连接

左外连接 保留左表所有行，右表匹配不上则以 NULL 填充。

```
SELECT S.StudentName, D.DeptName
FROM Student S
LEFT JOIN Department D
  ON S.DeptID = D.DeptID;
```

右外连接 保留右表所有行，左表匹配不上则以 NULL 填充。

全外连接 保留左右两表所有行，不匹配的部分以 NULL 填充。

自然连接 两表中所有同名属性相等的元组连接。

```
SELECT S.StudentName, D.DeptName
FROM Student S NATURAL JOIN Department D;
```

3.7 视图

```sql
CREATE VIEW CS_Students AS
SELECT StudentID, StudentName
FROM Student
WHERE DeptID = (
  SELECT DeptID FROM Department WHERE DeptName = 'Computer Science'
);
```

3.8 完整性约束

```sql
-- 假设已有 Department 表：
CREATE TABLE Department (
  dept_id INT PRIMARY KEY,
  dept_name VARCHAR(100) NOT NULL
);

-- 现在创建 Student 表，展示各种完整性约束：
CREATE TABLE Student (
  student_id INT NOT NULL,
  student_name VARCHAR(50) NOT NULL,
  email VARCHAR(100) UNIQUE,
  age INT DEFAULT 18 CHECK (age > 0),
  dept_id INT,
  -- 定义表级别的主键、外键等约束
  CONSTRAINT pk_student PRIMARY KEY (student_id),
  CONSTRAINT fk_student_dept
    FOREIGN KEY (dept_id)
    REFERENCES Department(dept_id)
    ON DELETE SET NULL -- 当 Department 被删除时，本表相应行的 dept_id 设为 NULL
    ON UPDATE CASCADE, -- 当 Department 的 dept_id 更新时，自动更新本表对应字段
  -- 可以额外示例一个自定义检查约束（表级别）
  CONSTRAINT chk_name_length
    CHECK (LENGTH(student_name) >= 2)
);
```

四 数据库设计

4.1 无损分解

一个关系模式 R 被分解为 R_1, R_2，如果

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

则称该分解是无损的。

判断是否是无损分解：$R_1 \cap R_2$ 要么是 R_1 的超码，要么是 R_2 的超码。

4.2 保持依赖

保持依赖 原函数依赖集的闭包与分解后的模式的函数依赖集的并集的闭包相同

$$(F_1 \cup F_2)^+ = F^+$$

4.2.1 保持依赖的判断

4.2.1.1 定义

使用定义

$$(F_1 \cup F_2)^+ = F^+$$

判断是否保持依赖。

4.2.1.2 保持依赖的充分条件

如果 F 的每一个函数依赖都可以在分解后的一个关系上验证，那么这个分解就是保持依赖的。

4.2.1.3 另一种判断方法

1. 对于 F 中的每个函数依赖 $\alpha \to \beta$，令 结果 为 α
2. 对于分解后的每一个模式 R_i，将 结果 和模式 R_i 的交集的闭包和模式取交集，添加到 结果 中

$$\text{result} = \text{result} \cup \left((\text{result} \cap R_i)^+ \cap R_i \right)$$

3. 重复上一步直到 结果 不再变化
4. 判断 结果 是否包含 β

4.3 计算 F 下 α 的闭包 α^+

1. 将 α 添加到 结果 中
2. 对于 F 中的每个函数依赖 $\beta \to \gamma$，如果 β 是 结果 的子集，则将 γ 合并到 结果 中
3. 重复上一部直到 结果 不再变化

4.4 公理

自反律 若 $\beta \subseteq \alpha$，则 $\alpha \to \beta$

增补律 若 $\alpha \to \beta$，则 $\alpha\gamma \to \beta\gamma$

传递律 若 $\alpha \to \beta$ 且 $\beta \to \gamma$，则 $\alpha \to \gamma$

合并律 若 $\alpha \to \beta$ 且 $\alpha \to \gamma$，则 $\alpha \to \beta\gamma$

分解律 若 $\alpha \to \beta\gamma$，则 $\alpha \to \beta$ 和 $\alpha \to \gamma$

伪传递律 若 $\alpha \to \beta$ 且 $\beta\gamma \to \delta$，则 $\alpha\gamma \to \delta$

4.5 范式理解

1NF 每个属性都是原子的。也就是说在逻辑上来说，每个属性都不可再分。例如“地址”和“电话”应当分开存储。

2NF 消除非主属性的部分函数依赖[2]。像 仓库名称→管理员, (仓库名称,物品名称)→物品数量 这样的表就不符合 2NF。

3NF 消除非主属性的传递函数依赖。也就是说，每个非主属性必须 *直接* 依赖于码。例如 (订单编号→客户编号→客户姓名) 这样的表就不符合 3NF，应当将客户姓名提取出来。

BCNF 消除主属性的部分函数和传递函数依赖。也就是说所有非平凡函数依赖的左侧必须包含完整的码。

举例说明：

仓库名称	管理员编号	管理员姓名	产品名称和数量
华南中心仓	5	张明	ThinkPad T14, 156 台
北方物流仓	6	王建国	小米手环 8 Pro, 324 个
成都配送中心	7	刘华	小米手环 8 Pro, 1280 个
成都配送中心	7	刘华	AirPods Pro, 345 台

这个数据库很明显不符合 1NF，因为产品名称和数量是一个属性。将其拆分为两个属性：

仓库名称	管理员编号	管理员姓名	产品名称	产品数量
华南中心仓	5	张明	ThinkPad T14	156
北方物流仓	6	王建国	小米手环 8 Pro	324
成都配送中心	7	刘华	小米手环 8 Pro	1280
成都配送中心	7	刘华	AirPods Pro	345

该表的函数依赖集为：

$$\textbf{仓库名称} \to \textbf{管理员编号}, \textbf{管理员编号} \to \textbf{管理员姓名}, (\textbf{仓库名称}, \textbf{产品名称}) \to \textbf{产品数量}$$

其中候选键为 (仓库名称, 产品名称) ，但管理员编号只依赖于仓库名称，所以该表不符合 2NF。将其拆分为两个表：

[2]部分函数依赖是指在一个关系模式中，某个非主属性只依赖于码（候选码）的一部分，而不是完全依赖于整个码。

仓库名称	产品名称	产品数量
华南中心仓	ThinkPad T14	156
北方物流仓	小米手环 8 Pro	324
成都配送中心	小米手环 8 Pro	1280
成都配送中心	AirPods Pro	345

仓库名称	管理员编号	管理员姓名
华南中心仓	5	张明
北方物流仓	6	王建国
成都配送中心	7	刘华

现在我们考虑第二个表，它不符合 3NF，因为 仓库名称 $\rightarrow$ 管理员编号 $\rightarrow$ 管理员姓名，管理员姓名并不直接依赖于仓库名称。将其拆分为两个表：

管理员编号	管理员姓名
5	张明
6	王建国
7	刘华

仓库名称	管理员编号
华南中心仓	5
北方物流仓	6
成都配送中心	7
成都配送中心	7

现在符合 3NF。

4.6 BCNF

BCNF 对于所有非平凡的函数依赖 $\alpha \rightarrow \beta (\alpha \in R \land \beta \in R)$，$\alpha$ 是模式 R 的超码。

4.6.1 BCNF 的检测

- 检测一个非平凡的函数依赖 $\alpha \rightarrow \beta$ 是否违反 BCNF，只需检查 α^+ 是否包含 R 的所有属性，也就是检查 α 是否是超码。
- 检测一个模式 R 是否是 BCNF，只需要检测 F 中的所有函数依赖是否是 BCNF。
- 检测一个模式分解 $\{R_1, R_2, ..., R_n\}$ 是否是 BCNF，需要对于每一个 R_i 检测：

R_i 属性的每个子集 α 要不然推不出 R_i 的任何其他属性 $(R_i - \alpha)$，要不然能够推出 R_i 的所有属性。如果都不满足，说明该分解不是 BCNF，并且 $\alpha \to R_i \cap (\alpha^+ - \alpha)$ 是违反 BCNF 的函数依赖。

例如模式分解 $(A, B)(A, C, D, E)$ 和函数依赖 $A \to B, BC \to D$。对于 $\alpha = AC$, $\alpha^+ = ABCD$，能推出 D 但不能推出 E，所以该分解不是 BCNF。很明显，$AC \to D$ 是违反 BCNF 的函数依赖。

4.6.2 BCNF 分解

1. 找到一个不满足 BCNF 的，在 R_i 上成立的非平凡函数依赖 $\alpha \to \beta$, 且 α 不包含 β。
2. 将 R_i 替换为：
   a. (α, β)
   b. $(R - \beta)$
3. 重复上一步直到所有模式都满足 BCNF。

4.7 3NF

3NF 要求模式中的每个函数依赖 $\alpha \to \beta$ 要么是 BCNF，要么 $\beta - \alpha$ 中的每个属性都属于候选码[3]。

4.7.1 3NF 分解

1. 令 F_c 为 F 的正则覆盖
2. 对于 F_c 中的每一个函数依赖 $\alpha \to \beta$，添加一个模式 $R_i = (\alpha, \beta)$
3. 如果没有模式 R_i 包含 R 的候选码，则再添加一个模式，属性为 R 的任意候选码
4. 如果有任意模式 $R_j \subseteq R_i$，就删除 R_j。

4.8 正则覆盖

无关属性 去除该属性而不改变该函数依赖集的闭包。

从函数依赖的左侧删除某个属性使其变为更强的约束，因为去除了无关属性；从函数依赖的右侧删除某个属性使其变为更弱的约束，因为减少了可以决定的属性。

正则覆盖 “最小”的能够推出函数闭包的函数依赖集。

4.8.1 计算正则覆盖 F_c

1. 令 $F_c = F$
2. 将 F_c 中任何形如 $\alpha_1 \to \beta_1$ 和 $\alpha_1 \to \beta_2$ 的依赖替换成 $\alpha_1 \to \beta_1\beta_2$

[3] 每个属性可以属于不同的候选码[4]

[4] 候选码可以有多个，例如 $(A \to B, B \to A, AC \to D)$ 中，候选码就可以是 AC 或 BC

3. 在 F_c 中寻找一个函数依赖 $\alpha \to \beta$，满足 α 或 β 中有一个无关属性 *(注意：使用 F_c 检验无关属性)*
4. 如果找到一个无关属性，则将它从 F_c 中的 $\alpha \to \beta$ 删除
5. 若 F_c 有变化，则回到第二步。

五 查询优化

STEP 1 根据

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}\big(\sigma_{\theta_2}(E)\big)$$

将 conjunctive selection 分解为多个单独的选择操作，以使单个选择操作尽可能沿查询树下移

根据选择操作的交换率和分配率（对连接操作，对集合并、交、差）：

$$\sigma_{\theta_1}\big(\sigma_{\theta_2}(E)\big) = \sigma_{\theta_2}\big(\sigma_{\theta_1}(E)\big)$$

当 θ_0 只包含其中的一个表达式的属性时：

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = \sigma_{\theta_0}(E_1) \bowtie_\theta E_2$$

当 θ_1 只包含 E_1 的属性，θ_2 只包含 E_2 的属性时：

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = \big(\sigma_{\theta_1}(E_1)\big) \bowtie_\theta \big(\sigma_{\theta_2}(E_2)\big)$$

$$\sigma_\theta(E_1 \cup E_2) = \sigma_\theta(E_1) \cup \sigma_\theta(E_2)$$

$$\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - \sigma_\theta(E_2) \quad \text{and} \quad \sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - E_2$$

$$\sigma_\theta(E_1 \cap E_2) = \sigma_\theta(E_1) \cap \sigma_\theta(E_2) \quad \text{and} \quad \sigma_\theta(E_1 \cap E_2) = \sigma_\theta(E_1) \cap E_2$$

将查询树上的每个选择尽可能移向叶节点，更早的执行选择操作。

STEP 2 根据连接操作的结合律和交换率，使用

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

和

$$\big(E_1 \bowtie_{\theta_1} E_2\big) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} \big(E_2 \bowtie_{\theta_2} E_3\big)$$

重新安排查询树中的连接操作，使得产生的中间结果所含记录尽可能少。

STEP 3 利用

$$\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$$

以连接操作代替相邻的选择和笛卡尔乘积操作。

STEP 4 利用

(当 $L_1 \subseteq L_2 \subseteq ... \subseteq L_n$ 时)

$$\Pi_{L_1}\Big(\Pi_{L_2}...\Big(\Pi_{L_n}(E)\Big)...\Big) = \Pi_{L_1}(E)$$

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Big(\Pi_{L_1}(E_1)\Big) \bowtie_\theta \Big(\Pi_{L_2}(E_2)\Big)$$

(设 L_3 为在 θ 中但不在 L_1 中的属性集，L_4 为在 θ 中但不在 L_2 中的属性集)

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}\Big(\Big(\Pi_{L_1 \cup L_3}(E_1)\Big) \bowtie_\theta \Big(\Pi_{L_2 \cup L_4}(E_2)\Big)\Big)$$

$$\Pi_L(E_1 \cup E_2) = \Pi_L(E_1) \cup \Pi_L(E_2)$$

将查询树上的投影操作尽可能下移，以便尽早执行投影操作，减少中间计算结果。

STEP 5 将最后的查询树分解为多个子树，使子树中的各操作可以采用流水线方式 执行，以减少对外设的访问次数。

六 事务

6.1 事务的四个特性（ACID）

原子性（Atomicity） 事务的所有操作在数据库中要么全部正确反映出来，要么完全不反映。

一致性（Consistency） 以隔离方式执行事务（即，没有其他事务的并发执行）以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，但系统保证：对于任何一对事务 T_i 和 T_j，在 T_i 看来，T_j 要么在 T_i 开始之前已经完成执行，要么在 T_i 完成之后 T_j 才开始执行。因此，每个事务都感觉不到系统中有其他事务在并发地执行。

持久性（Durability） 在一个事务成功完成之后，它对数据库的改变必须是永久的，即使出现系统故障也是如此。

6.2 冲突可串行化

构造一个调度 S，称为优先图。边集由满足三个条件之一的所有 $T_i \to T_j$ 组成：

1. T_i 执行 write(Q) 之后，T_j 执行 read(Q)；
2. T_i 执行 read(Q) 之后，T_j 执行 write(Q)；
3. T_i 执行 write(Q) 之后，T_j 执行 write(Q)。

如果优先图中存在一条 $T_i \to T_j$ 的边，那么在所有等价于 S 的串行调度中，必须先执行 T_i，再执行 T_j。所以如果优先图中不存在环，那么 S 是冲突可串行化的。

6.3 可恢复调度

可恢复调度 对于每对事务 T_i 和 T_j，如果 T_j 读取了 T_i 写的数据，那么 T_i 必须在 T_j 之前提交。否则如果 T_i 失效并回滚，而 T_j 已经提交，就遇到了不可正确恢复的情况。

T_i	T_j
read(A)	
write(A)	
	read(A)
	commit
read(B)	
abort?	
	commit

表 5 红色为不可恢复调度，绿色为可恢复调度

6.4 级联回滚

级联回滚 由于单个事务失效而导致一系列事务回滚的现象

无级联调度 可以避免级联回滚的调度。对于每对事务 T_i 和 T_j 都满足如果 T_i 读取了由 T_j 所写的一个数据项，则 T_i 的提交操作必须在 T_j 的这一读操作之前。

七 并发控制

7.1 两阶段封锁协议

两阶段封锁协议 该协议要求每个事物分两个阶段提出加锁和解锁申请：

1. 增长阶段：一个事务可以获得锁，但不能释放锁。
2. 缩减阶段：一个事务可以释放锁，但不能获得新锁。

严格两阶段封锁协议 不但要求封锁是两阶段的，还要求事务持有的 *排他模式* 的锁必须在事务提交后才释放。

强两阶段封锁协议 要求在事务提交前保留 *所有* 的锁。

锁转换 可以通过升级将共享锁转换为排他锁，也可以通过降级将排他锁转换为共享锁。但升级只能发生在增长阶段，而降级只能发生在缩减阶段。

7.2 多粒度中的意向锁

意向共享锁（IS） 在树的较低层加共享锁

意向排他锁（IX） 在树的较低层加排他锁

共享意向排他锁（SIX） 在当前节点加共享锁，在树的较低层加排他锁

	IS	IX	S	SIX	X

IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

八 恢复

8.1 检查点

1. 将当前的所有日志输出到稳定存储器。
2. 将所有修改过的缓存块输出到磁盘。
3. 将一条形如 $<\text{checkpoint}, L>$ 的日志记录写入日志，其中 L 是执行检查点时的活跃事务的列表。

恢复时，对于 L 中的所有事务以及检查点之后才开始执行的所有事务，系统执行 redo 或 undo 操作：

1. 如果既没有 commit 记录，也没有 abort 记录，则 undo；
2. 如果有 commit 或 abort 记录，则 redo。

8.2 事务恢复策略

考虑**正常操作**的事务回滚：

1. 从后往前扫描，对于所有形如 $<T_i, X_j, V_1>$ 的日志记录：
   a. 将值 V_1 更新到数据项 X_j 上。
   b. 向日志中写一条 redo-only 日志记录 $<T_i, X_j, V_1>$，其中 V_1 是本次回滚中数据项 X_j 被恢复到的值。
2. 一旦发现了 $<T_i$ start $>$，则停止回滚。

考虑**系统崩溃**的事务回滚，分两阶段进行：

1. 在**重做（redo）**阶段中，系统从最后一个检查点开始正向扫描日志并重演所有事务的更新：
   a. 将待回滚事务的列表 `undo-list` 初始化在检查点时的活跃事务列表。
   b. 一旦遇到形如 $<T_i, X_j, V_1, V_2>$ 的日志记录，或者形如 $<T_i, X_j, V_2>$ 的 redo-only 日志记录，就执行重做操作，也就是说将值 V_2 更新到数据项 X_j 上。
   c. 一旦发现形如 $<T_i$ start $>$ 的日志记录，就把 T_i 添加到 `undo-list` 中。
   d. 一旦发现形如 $<T_i$ abort $>$ 或 $<T_i$ commit $>$ 的日志记录，就把 T_i 从 `undo-list` 中删除。
2. 在**撤销（undo）**阶段中，系统从后往前开始逆向扫描日志并执行回滚。

a. 遇到属于 undo-list 中的事务的日志记录，就执行撤销动作，类似于正常操作的事务回滚。

b. 当系统发现属于 undo-list 中的事务的 $< T_i \text{ start} >$ 记录，就向日志中写一条 $< T_i \text{ abort} >$ 的日志记录，并将 T_i 从 undo-list 中删除。

c. 如果 undo-list 为空，则撤销阶段结束。