数据库概论

5. 实体关系模型

世界是由一组称作实体的基本对象和这些对象之间的联系构成的。

数据库设计过程：需求分析，概念数据库设计（E-R模型），逻辑数据库设计，物理数据库设计。

概念数据库模型：按用户观点对数据和信息建模，表现从现实世界中抽象出来的事务以及他们之间的联系。

标准E-R图

实体：客观存在并且可以相互区分的事物。

属性：实体所具有的某一特性，实体可以由若干个属性进行刻画。

域：实体的取值范围。

实体型：实体名与其属性名集合共同构成实体型，实体型描述了实体的一个结构。

实体集：同型实体的集合称作实体集。

简单属性：不可再分的属性。

符合属性：可以划分为更小的属性的属性。eg：生日=年+月+日。

单值属性：每个特定的实体在该属性上取值唯一。eg：性别。

多值属性：特定实体在该属性上取值可以多于一个。eg：电话号码。

NULL属性：表示无意义（实体在该属性上没有值）或值未知（实体在该属性上有值，但目前没有获得这一信息）。

基属性：无法从其他实体或属性中派生出的属性值。类比数据库基本表。

派生属性：可以从其他相关属性或实体派生出来的属性值。类比数据库视图。

超码：一个或多个属性的集合，这些属性可以唯一确定每一个实体。

候选码：超码的最小的若干个子集。

主码：候选码之一，一般选取简单的、符合认知规律的候选码作为主码。

关系集合：标明多个实体之间的联系。$\{(e_1,\cdots,e_n|e_1\in E_1,\cdots,e_n\in E_n)\}$。关系集合也可以具有特定的属性。

关系集合的度：表示参与一个关系集合的实体的数目，一般而言是2，也存在更多的情况。

参与：实体集之间的关联称作参与，即实体参与联系。eg：张三选修数据库中“张三”和“数据库”两个实体参与了联系“选修”。

全部参与：实体集中的每个实体都参与到联系集中的至少一个联系。

部分参与：实体集中只有部分实体参与到联系集的联系之中。

角色：实体在联系中的作用称为实体的角色，用于区分同一实体集不止一次参与一个联系集的情况。

映射的基数：一个实体通过一个联系集能够与另一个实体集向关联的实体的数目。有一对一，一对多，多对多几种情况。

E-R图：长方形表示实体集合，菱形表示关系集合；连线连接实体/关系集合和实体集合属性，双线表示完全参与，单线表示部分参与，角色写在连线上下；椭圆表示单值属性，双椭圆表示多值属性，虚椭圆表示派生属性，椭圆树表示符合属性；下划线表示实体集合的主码属性；连线箭头表示映射基数，没有箭头表示该实体为多方实体集，有箭头表示该实体集为单方实体集。

非二元关系到二元关系的转换：1. 创建一个新的实体集合表示该非二元关系，该实体集合同其余实体集建立二元关系联系。2. 将一个非二元关系拆分为若干个独立的二元关系，这种拆分会导致原非二元关系的信息丢失，eg原非二元关系的属性无法附加在任何一个拆分后的二元关系之上。

E-R模型扩展

存在依赖：若实体X的存在依赖于实体Y的存在，即Y被删除X也必须被删除，则称X存在依赖于Y，Y为支配实体，X为从属实体。

弱实体集：一个没有主码的实体集合称作弱实体集合。弱实体和标识实体集合之间存在多对一关联关系。弱实体的分辨符（部分码）用于区分弱实体集合中的实体，弱实体的主码通过其依赖的强实体的主码和弱实体自身的分辨符来表示。

弱实体的转化：通过添加弱实体依赖的强实体的主码，可以将弱实体集转化为强实体集，但会引起数据冗余，并且丧失依赖关系（无法自动删除弱实体集）。

弱实体集的E-R图表示：弱实体集用双框矩形表示，标示性联系用双框菱形表示，联系集与弱实体集见双线连接，联系集与强实体集间单线箭头连接，弱实体集分辨符下划虚线标明。

特殊化：IS A，表示高层实体与低层实体之间的父子类联系。用标记为ISA的倒三角表示。

概括：各个实体集根据其共有的性质，合成一个高层的实体集，概括为高层实体集与低层实体集之间的包含关系。概括与特殊化互逆，E-R图中表示相同。

属性继承：高层实体集的属性被低层实体集自动继承，低层实体集的特有属性仅适用于特定的低层实体集。

层次结构Hierarchy：实体集作为低层实体集，只能参与到一个ISA联系之中。

格结构Lattice：实体集作为低层实体集，可以参与到多个ISA联系之中。

成员资格：确定哪些实体能够成为给定低层实体集的成员。

条件定义的成员资格：一个实体成员资格的确定，基于该实体是否满足一个显式地条件或谓词。eg：学生实体集具有属性“本/硕/博”，该属性确定高层的学生实体属于哪个低层实体集。

用户定义的成员资格：由数据库用户来指定一个实体归入哪个低层实体集。

成员身份：同一个概括之中，一个实体能否属于多个不同的低层实体集。不相交的成员身份Disjoint，一个实体至多属于一个低层实体集；有重叠的成员身份Overlapping，一个实体可以同时属于同一概括的多个低层实体集。

全部性约束：确定高层实体集中的每个实体是否至少属于一个低层实体集。分作全部的和部分的。全部的用双线连接实体集和ISA表示。

E-R图模式转换

E-R图转换为表关系：实体集合关系集表示为表，其均具有主码；一个E-R图对应一个多表集合；对E-R图中的每个实体集合关系集，都在数据库中建立一个与之对应的表，数据库表的列对应各个属性。

复合属性转换：将复合属性最小的组合属性作为实体的属性，只保留属性树的叶属性。

多值属性转换：将多值属性转换为一个新的一对多或多对多联系。

一对一联系转换：若联系双方部分参与，则定义一个新的联系，属性为参与双方的码；若联系的一方全部参与，则省略关系表，将部分参与方的码作为全部参与方的属性。

一对多联系转换：将单方参与实体的码作为多方参与实体的属性，省略中间关系表。

多对多联系转换：将联系定义为新的关系，属性为双方的码。

弱实体集转换：对应关系的码由弱实体集本身的分辨符加上依赖的强实体的码组成，省略中间关系表。

概括/特殊化转换：高层实体集和低层实体集分别建表，低层实体集的关系包括高层实体集的吗；若概括不相交且是全部的，则可以不为高层实体集建表，只建立低层实体集的表，低层实体集表包括上层实体集所有属性。

E-R模型设计

实体集和属性的选择：独立性越强，越应该创建实体，对应独立的表。若存在其他属性描述了属性A，或属性A参与不止一个关系，或属性A是多值属性，则考虑为属性A构造实体。

局部E-R模式设计：需求分析，确定局部结构范围，实体定义，联系定义，属性分配。

全局E-R模式设计：局部E-R模式，确定公共实体类型，合并两个局部E-R模式，检查并消除冲突，检查是否有未合并的局部模式。

属性冲突：包括域冲突，取值单位冲突，和命名冲突（同名异义和异名同义）。

结构冲突：同一对象在不同应用中有的被抽象为实体，有的被抽象为属性；同一实体属性组成不同；实体间的联系组成不同。

6. 关系模式设计

依赖和码

函数依赖：设$R(U)$是属性集U上的关系模式，$X,Y\subset U$，r是$R(U)$上的任意一个关系，若如下条件成立，则称X函数决定Y，或Y函数依赖于X，记作$X\to Y$，X为决定因素。

$$\forall t,s\in r,\ \text{若}\ t[X]=s[X],\ \text{则}\ t[Y]=s[Y]$$
$$\text{或}\ \neg\exists t,s\in r,\ t[X]=s[X],\ \text{但}\ t[Y]\neq s[Y]$$

函数依赖不是从数据中推导出来的，而是来自客观世界对于数据的约束。

函数依赖满足：测试关系在给定函数依赖下是否合法。若一个关系r在函数依赖集合F下合法，则称r满足F。

函数依赖保持：测试约束是否在合法关系上都成立。若R的所有合法关系都满足函数依赖F，则F在R上保持。

平凡函数依赖：若$X\to Y$且$Y\subset X$，则Y平凡函数依赖于X；若$Y\not\subset X$，则Y非平凡函数依赖于X。

完全函数依赖：$X\to Y$且X的任何真子集X'都有$X'\not\to Y$，则Y完全函数依赖于X，记作$X\overset{F}{\to}Y$。

部分函数依赖：不是完全函数依赖的函数依赖，记作$X\overset{P}{\to}Y$。

传递函数依赖：若$X\to Y,Y\to Z,Y\not\to X$，且$(X\cup Y)\cap Z=NULL$，则Z传递函数依赖与X。

SQL函数依赖：SQL通过候选码和主码直接支持函数依赖，其余函数依赖通过断言完成。

超码：设K为R<U,F>的属性或属性组，若$K \to U$，则称K为R的超码。

候选码：若$K \overset{F}{\to} U$，则称K为R的候选码。

主码：若R(U,F)中有多个候选码，则从中选定一个作为R的主码。

主属性：包含在**任意一个**候选码中的属性。

全码：关系模式的码由整个属性组组成。

多值依赖：关系模式$R(U), X, Y, Z \subset U$，并且$Z = U - X - Y$，多值依赖$X \to\to Y$成立当且仅当对R(U)的任意关系r，给定一对(X,Z)值，有一组Y值，这组值仅仅决定于X值而与Z值无关。

$$
\begin{gathered}
\text{关系模式} R(U), X, Y, Z \in U,\ Z = U - X - Y, \\
\forall r \in R(U)\text{，若} \exists t_1, t_2\text{，使得} t_1[X] = t_2[X] \\
t_1 = (t_1[X], t_1[Y], t_1[Z]), t_2 = (t_2[X], t_2[Y], t_2[Z]) \\
\text{那么就必然存在} t_3 \text{和} t_4\text{，使得} \\
t_3 = (t_2[X], t_2[Y], t_1[Z]), t_4 = (t_1[X], t_1[Y], t_2[Z]) \\
\text{则} Y \text{多值依赖于} X\text{，} X \to\to Y
\end{gathered}
$$

多值依赖的对称性：若$X \to\to Y$，则$X \to\to Z$，其中$Z = U - X - Y$。

函数依赖是多值依赖的特例：若$X \to Y$，则$X \to\to Y$

平凡的多值依赖：$X \to\to Y, U - X - Y = \Phi$，则$X \to\to Y$为平凡的多值依赖。

多值依赖的范围：$X \to\to Y$的有效性与属性集范围有关。

$$
\begin{gathered}
\text{对于} XY \subseteq W \subseteq U \\
X \to\to Y \text{在属性集} W \text{上成立，但在} U \text{上不一定成立} \\
X \to\to Y \text{在} U \text{上成立，则} X \to\to Y \text{在属性集} W \text{上成立} \\
X \to\to Y \text{在} R(U) \text{上成立，不能断言对于} Y' \subseteq Y\text{，} X \to\to Y' \text{是否成立}
\end{gathered}
$$

范式

范式不良特性：插入异常，由于没有信息导致无法插入；删除异常，删除导致信息消失；更新异常，元组间有联系，必须更改多次；数据冗余，相同的数据出现多次。

1NF：关系中的每个分量不可再分，即属性集合中没有复合属性。

2NF：若$R \in 1NF$，且每个非主属性都完全依赖于码，则称$R \in 2NF$。消除了非主属性对码的部分函数依赖。

3NF：关系模式R<U,F>中，若不存在码X，属性组Y，及非主属性Z($Z \not\subset Y$)，使得$X \to Y, Y \to Z, Y \nrightarrow X$，则称$R \in 3NF$。消除了非主属性对码的传递函数依赖。

BCNF：关系模式R<U,F>中，若$X \to Y, Y \not\subset X$时，X必含有码，则$R \in BCNF$。非平凡的函数依赖必定包含码。

4NF：关系模式R<U,F>$\in 1NF$，若$X \to\to Y (Y \not\subset X)$是非平凡的多值依赖，且X含有码。消除非平凡的且不是函数依赖的多值依赖。

3NF⊂2NF：反证法证明。若$R \in 3NF, R \notin 2NF$，则一定有非主属性部分依赖于码。设X为R的码，则存在X的真子集X'，以及非主属性$Z(Z \not\subset X')$，使得$X' \to Z$。那么在R中存在码X，属性组X'，和非主属性Z，使得$X \to X', X' \to Z, X' \nrightarrow X, (X \cup X') \cap Z = NULL$成立，产生矛盾！

BCNF⊂3NF：反证法证明。若$R \in BCNF, R \notin 3NF$，则一定有非主属性对码的传递依赖。存在X为R的码，属性组Y，以及非主属性$Z(Z \not\subset Y)$，使得$X \to Y, Y \to Z, Y \nrightarrow X$成立。那么Y重含有码（BCNF和推出$Y \to Z$），则$Y \to X$，产生矛盾！

4NF⊂BCNF：反证法证明。若$R \in 4NF, R \notin BCNF$，则一定有非平凡的函数依赖$X \to Y$其中X不包含码。那么$X \to Y \Rightarrow X \to\to Y$，且X不包含码，则$R \notin 4NF$，产生矛盾！

模式分解

模式分解：是提高模式规范化程度的手段。R(U,F)的一个分解指如下形式。与模式分解相关的基本代数运算包括投影和自然连接。

$$\rho = \{R_1 < U_1, F_1 >, R_1 < U_2, F_2 >, \cdots, R_n < U_n, F_n >\}$$
$$\text{其中} U = Y_{i=1}^{n} U_i\text{，且不存在} U_i \subseteq U_j, 1 \le i, j \le n$$

模式分解中可能存在的问题：分解之后导致信息丢失无法重构原表；分解之后导致函数依赖的检查无法在单表上完成（必须在多表连接上完成）。

模式分解的要求：**无损连接分解**，分解不会导致信息丢失；**保持依赖**，分解最好保持单表上函数依赖；**无冗余**，各个关系最好属于BCNF或3NF。

逻辑蕴涵：关系模式R，F是其上的函数依赖，X和Y是其属性子集，若从F能够推出$X \to Y$，则称F逻辑蕴涵$X \to Y$，记作$F \vdash X \to Y$。

函数依赖集闭包：被F所逻辑蕴含的函数依赖的全体所构成的集合称为F的闭包，记作F^+

$$F^+ = \{X \to Y | F \vdash X \to Y\}$$

保持依赖：若分解模式中投影的函数依赖的闭包和原模式的函数依赖闭包等价，则分解是保持依赖的。

Armstrong公理系统：一套正确的完备的推理规则集，包括自反律，增广律和传递律。X，Y，Z是属性集。

$$\begin{aligned}
&\text{自反律：若} Y \subseteq X\text{，则} X \to Y\\
&\quad\text{证明：} t[X] = s[X]\text{且} Y \subseteq X\\
&\qquad \Rightarrow t[Y] = s[Y] \Rightarrow X \to Y\\
&\text{增广律：若} X \to Y\text{，则} XZ \to YZ\\
&\quad\text{证明：} t[XZ] = s[XZ] \Rightarrow t[X] = s[X]\\
&\qquad\text{又有} X \to Y \Rightarrow t[Y] = s[Y]\\
&\qquad\quad t[XZ] = s[XZ] \Rightarrow t[Z] = s[Z]\\
&\qquad\text{综上} t[YZ] = s[YZ] \Rightarrow XZ \to YZ\\
&\text{传递律：若} X \to Y, Y \to Z\text{，则} X \to Z\\
&\quad\text{证明：} t[X] = s[X], X \to Y \Rightarrow t[Y] = s[Y]\\
&\qquad\text{又有} Y \to Z \Rightarrow t[Z] = s[Z]\\
&\qquad \Rightarrow X \to Z
\end{aligned}$$

Armstrong公理系统的推论：合并律，分解率和伪传递律。

$$
\begin{aligned}
&\text{合并律：}X \to Y, X \to Z\text{，则}X \to YZ\\
&\quad\text{证明：}X \to Y \Rightarrow X \to XY\ (\text{增广律})\\
&\qquad X \to Z \Rightarrow XY \to ZY\ (\text{增广律})\\
&\qquad\Rightarrow X \to YZ\ (\text{传递律})\\
&\text{分解律：}X \to YZ\text{，则}X \to Y, X \to Z\\
&\quad\text{证明：}X \to YZ, YZ \to Y\ (\text{自反律})\\
&\qquad\Rightarrow X \to Y\ (\text{传递律})\text{，同理}X \to Z\\
&\text{伪传递律：若}X \to Y, WY \to Z\text{，则}WX \to Z\\
&\quad\text{证明：}X \to Y \Rightarrow WX \to WY\ (\text{增广律})\\
&\qquad\text{又有}WY \to Z \Rightarrow WX \to Z\ (\text{传递律})
\end{aligned}
$$

函数依赖集的闭包算法：函数依赖集F，计算其闭包F^+。

```
F+ = F
repeat
    for each f in F+
        对f应用自反律和增广律，并将结果函数依赖加入F+
    for each f1 and f2 in F+
        if f1和f2可以利用传递律合并
            then 将结果函数依赖加入F+
until F+不再变化
```

属性集闭包：F为属性集U上的一组函数依赖，$X \subseteq U$，X_F^+为X关于关于F的闭包，当且仅当下式成立。

$$
X_F^+ = \{A | X \to A\text{能由}F\text{根据}Armstrong\text{公理导出}\}
$$

属性集闭包性质：$\alpha \to \beta$在F^+出现 $\Leftrightarrow \beta \subseteq \alpha^+$

属性集的闭包算法

```
a+ = a
repeat
    for each b->c in F
        if b in a+
            then a+ = a+ 并 c
until a+不再变化
```

属性闭包测试函数依赖：通过检测$\beta \subseteq \alpha^+$是否成立，检测$\alpha \to \beta$是否成立。

属性闭包测试超码：通过检测$U \subset \alpha^+$是否成立，检测α是否是R(U)的超码。

计算候选码：遍历测试属性组合，多删少补。可以使用启发式规则如下，进行初始化，之后不断向集合中加入新的属性直至得到最小的超码。

$$
\begin{aligned}
&\text{左部属性必定是主属性 — 只出现在}F\text{左边的属性}\\
&\text{右部属性必是非主属性 — 只出现在}F\text{右边的属性}\\
&\text{外部属性必定是主属性 — 不出现在}F\text{左右的属性}
\end{aligned}
$$

正则覆盖：F的正则覆盖是指与F等价的极小的函数依赖集合，其中没有冗余依赖，也没有冗余元素属性。

F_c是F的正则覆盖
$\Rightarrow F$逻辑蕴涵F_c中所有函数依赖
F_c逻辑蕴涵F中所有函数依赖
F_c中函数依赖没有五官属性
F_c中函数依赖的左部都唯一

检测属性无关：函数依赖集合F，其中有函数依赖$\alpha \to \beta$。检测属性无关时，考虑原依赖和删去该属性的新依赖的表达能力，若表达能力更弱的一方可以推导出表达能力更强的一方，则说明属性无关。

检测属性$A \in \alpha$在α中是否无关，新依赖表达能力强
检测F下的$(\alpha - \{A\})^+$是否包含β，若包括则说明A无关
检测属性$A \in \beta$在β中是否无关，原依赖表达能力强
检测$F' = (F = \{\alpha \to \beta\}) \cup \{\alpha \to (\beta - A)\}$下的$\alpha$是否包含$\beta$
若包括，则说明A无关

正则覆盖的算法

```
repeat
    利用合并律合并左部相同的函数依赖
    if 可以找出含有无关属性的函数依赖 a->b (无关属性在a或b之中)
        then 从a->b中删去无关属性，得到新的F
until F不再变化
```

无损连接分解：关系模式R<U,F>，$U = Y_{i=1}^{n} U_i$，$\rho = \{R_1 < U_1, F_1 >, \cdots, R_n < U_n, F_n >\}$是R的一个分解。若下式成立，则$\rho$是R的一个无损连接分解。

$$r\text{时}R\text{上的一个关系，定义}m_\rho(r) = \bowtie_{i=1}^{n} \prod R_i(r)$$
$$\text{若}r = m_\rho(r)\text{，则}\rho\text{是}R\text{上的无损连接分解}$$

无损连接分解定理：关系模式R(U)的分解$\rho = \{R_1, R_2\}$，ρ时无损连接分解的条件是$R_1 \cap R_2 \to R_1 - R_2$或$R_1 \cap R_2 \to R_2 - R_1$成立。

保持函数依赖的模式分解：若R(U,F)的分解$\rho = \{R_1, \cdots, R_n\}$满足下式，则称$\rho$是保持依赖的模式分解。$\prod$为函数依赖集的在属性集上的投影，定义如下。

$$\prod_Z(F) = \{X \to Y | X \to Y \in F^+ \wedge XY \subseteq Z\},\ Z \subseteq U$$
$$F^+ = (Y_{i=1}^{n} \prod R_i(F))^+$$

检查保持函数依赖的算法：检查$\alpha \to \beta$在R到$R_1, \cdots, R_n$的分解中是否保持。

```
res = a
repeat
    for each Ri in {R1,...,Rn}
        t = (res 交 Ri) 交 Ri
        res = res 交 t
until res不再变化
if res包含b中所有属性
    then a->b 得到保持
else 没有得到保持
```

模式分解算法

BCNF分解算法：保证无损连接分解，不一定保持函数依赖。

```
res = {R}
done = False
compute F+
while (not done)
    if res中存在Ri不属于BCNF then
        找出Fi中不包含码的非平凡的函数依赖a->b
        res = (res-Ri) 并 (Ri-b) 并 (a,b)
    else done = True
```

判定BCNF的算法：任意属性子集的闭包，要么只包含该子集自身，要么包含该模式所有属性。指数时间的算法。

3NF分解算法：保证无损连接，一定保持函数依赖。

```
Fc是F的正则覆盖
n = 0
for a->b in Fc
    if 没有模式Rj (1<=j<=n)包含a和b then
        n = n + 1
        Rn = a b
if 没有模式Rj (1<=j<=n)包含R的候选码 then
    n = n + 1
    Ri = R的任意一个候选码
return (R1,...Rn)
```

7. XML数据管理

XML基础

HTML：具有固定的标签集，标签的语义固定。

XML：提供了定义标签和描述标签间结构的功能，没有固定标签机，有固定语法但没有固定语义。

XML标签：数据节的标号。

XML元素：以<tagname>开始并以配套</tagname>结束的数据节，元素支持栈式嵌套。

XML属性：元素开始标签中的name=value对，一个元素可以有多个名字不同的属性。

XML命名空间：为了能让XML数据可以在不同的组织之间交换，使用唯一的串作为元素名前缀以避免交换时文档混淆，用unique-name:element-name的方式。

XML树模型：一个XML文档可以表示为节点标签的有序树，元素节点的子节点可以是属性页节点或子元素节点。

XML结构约束

结构约束：相当于数据模式，XML数据目前一般遵循某种结构约束，尤其是数据交换的双方应当共同遵循共同的结构约束。

文档类型定义DTD：描述了XML数据硬钢遵循的结构约束，包括可以出现什么元素，以及元素可以或必须具有什么属性，每个元素可以或必须出现什么子元素，以及出现多少次。**XML中所有值都表示为串。**

```
<!ELEMENT element(subelements-specification)>
<!ATTLIST element(attributes)>
```

DTD元素说明：子元素可以说明为元素表达式或者字符串（#PCDATA），子元素说明时可以使用正则表达式（|+*等）。

DTD属性说明：对每个属性说明其名字，属性类型（CDATA字符类型，ID唯一标识，IDREF(S)对其他元素(一个或多个)的引用）和是否强制（#REQUIRED强制的，#IMPLIED可选的）。ID每个元素只能有一个，并且在整个文档中唯一。

DTD的局限：元素都是串，不支持数据类型；难以使用正则表达式说明子元素出现的次数；ID和IDREF在整个文档中唯一，而不是同类元素中唯一；XML的DTD不通过XML的形式来表达。

XML Schema：XML的另外一种模式定义语言，使用XML语法说明，支持值的类型，支持对元素最小/大次数进行限制，支持自定义数据类型，但比DTD复杂很多。

XML查询

XML查询：获取某些节点的值；转换XML结构到另外一个XML结构；合并并集成多文档的数据；连接和聚集。

XPATH向下导航：文法如下。.为当前节点，E为元素名称，*表示任意元素，@A为属性，/和|表示父子关系，//为递归的后代或自身，[q]为筛选条件谓词，op为比较符号，c为常数。

```
Q <- . | E | @A | Q/Q | Q|Q | //Q | /Q | Q[q]
q <- Q | Q op c | q and q | q or q | not(q)
```

XPATH向上导航：文法如下。.../为父节点，ancestor和ancestor-or-self为祖先节点。

```
Q <- .../Q | ancestor::Q | ancestor-or-self::Q
```

XPATH左右导航：文法如下。following-sibling为下一个节点，preceding-sibling为下一个节点，position()为节点定位。

```
Q <- following-sibling::Q | preceding-sibling::Q | following::Q | preceding::Q
```

XQuery查询：FOR ... LET ... WHERE ... RETURN ...对应SQL中的FROM，WITH，WHERE，SELECT。示例如下。

```
找出余额大于400的账户并逐个编号
FOR $x in /bank-2/account
LET $acctno := $x/@account-number
WHERE $x/balance > 400
RETURN <account-number> $acctno </account-number>
```

FOR $var in expr：绑定$var到列表中的每一个值。

LET $var = expr：绑定$var到整个列表。

路径表达式：用于FOR中和变量绑定，或用于LET中定义临时变量，或用于WHERE中限制变量的取值，或用于RETURN中得到变量的特定路径的值。

函数：distinct()用于删除重复元素，document(name)返回给定文档树的根节点，sum()和count()等用聚集函数。

8. 事务处理

事务和ACID性质

事务：事务是程序的逻辑运行单位，必须保持数据库的一致性。

原子性Atomicity：事务操作要么全做，要么全不做。

一致性Consistency：如果事务执行前数据库状态是一致的，则执行后状态依然是一致的。

隔离性Isolation：事务并发进行，但每个事物都不了解其他兵法执行的事务，事物的中间过程对其它并发事务不可见。

持久性Durability：事务成功结束后，对数据库所做的更新将永久花，即使数据库故障也不受影响。

事务状态：Active，初始状态，事务执行时也处于该状态；Partially committed，最后一条语句执行后，开始一系列恢复准备工作；Fail，发现不能继续正常执行；Aborted，事务已经回滚并且数据库恢复到事务开始前的状态；Committed，事务成功结束。

```
Active -> 最后一条语句执行后 -> Partial committed
Active -> 事务无法继续正常执行 -> Failed
Partial committed -> 成功完成，永久写入数据库 -> Committed
Partial committed -> 事务无法正常完成 -> Failed
Failed -> 事务回滚，数据库恢复到事务开始前的状态 -> Aborted
```

原子性和持久性

影子数据库：假设每次仅有一个事务处于Active状态，有一个指针指向当前数据库的一致拷贝；所有更新都是对影子数据库进行，事务Commit后，影子数据库回写，指针指向更新后的数据库，原数据库作为影子数据库进行一致性更新；事务失败，则直接删除影子数据库。

影子数据库效率极低，每个事务都要拷贝一次数据库；改进方式是使用影子页面。

事务调度：事务执行的顺序成为一个调度。一组调度必须保证包含所有事务的操作指令，且一个事务之中的指令顺序不变。

串行调度：同一事务的指令紧挨在一起，不存在交叉。串行事务效率低，但一定可以保证一致性。

并发调度：来自不同事务的指令可以相互交叉执行。并发调度效率高，系统吞吐量高，担忧可能会破坏系统一致性。

可串行化：用于描述哪些事务是对的。其基本假设是每个事务都保持数据库的一致性，因此一组事务的串行执行也可以保持数据库的一致性。**一个并发调度是可串行化的，当且仅当它与一个串行调度等价。**

冲突：事务Ti和Tj的指令Ii和Ij冲突，当且仅当存在Ii和Ij都存取数据项Q，并且二者之中至少有一条写Q的指令。

冲突可串行化：若S通过一系列交换非冲突指令的操作，可以转换成调度S'，则S与S'**冲突等价。**若S'是串行调度，则调度S时冲突可串行化的。

视图等价：S和S'是相同事务集合的两个调度，S和S'试图等价当且仅当1) 对每个数据项Q，若S中事务Ti读到Q的初始值，则S'中事务Ti也必须读到Q的初始值；2) 对每个数据项Q，若在调度S中事务Ti执行了read(Q)，而该值来自于事务Tj，那么在调度S'中事务Ti也必须读到事务Tj产生的Q值；3) 对每个数据项Q，在调度S中执行最后一条write(Q)的事务，在S'中也必须执行最后一条write(Q)。

视图可串行化：调度S视图等价于一个串行调度，则称S是视图可串行化的。所有冲突可串行化的调度必然也是视图可串行化的；每个视图可串行化而非冲突可串行化的调度必然存在盲写，即没有被读值的写操作。

事务恢复：若事务失败，则必须撤销其对数据库的所有影响；若某个事务读取了失败事务写入的数据，则该事务也要撤销。

可恢复调度：若事务T2读取了事务T1写的数据，则T2必须后于T1提交。

无级联调度：若事务T2希望读取事务T1所写的数据，则T2读取时T1必须已经提交。无级联调度时可恢复调度。

优先图：有效图G=(V,E)，V时顶点集，E时边集，对应一个调度S，顶点集由所有参与调度的事务组成，边集对应所有冲突，由先者指向后者。

冲突可并行化判定：若优先图中有环，则调度S时非冲突可并行化的；否则无环，S是冲突可并行化的，串行序列为DAG的拓扑排序序列。

调度完成后再检测可串行化，发现违例则回滚事务，是一种可行但不可接收的方法；一般不检查优先图，而是通过协议施加约束来避免非可串行化调度。

隔离性

一致性保证：为了保证一致性，调度必须是冲突可串行化的或视图可串行化的，可恢复的，并且最好是无级联回滚的。

隔离性级别：应用环境对并发一致性要求不高，或应用程序自身保证一致性，则可以放松DB一致性要求，设置不同的隔离性级别。

丢失修改：T1和T2读入同一数据并修改，T1提交的结果破坏了T2提交的结果，导致T1的修改丢失。

读脏数据：T1修改数据并写回磁盘，T2读取T1写入的数据后，T1由于某种原因撤销，则T1回滚，数据恢复原值，此时T2独到的数据与数据库不一致，该数据为脏数据。

不能重复读：T2读取某数据后，T1对其进行了修改，当T2再次读取时，发现值不同。

发生幻象：T2按一定条件读取某数据后，T1插入了一些满足这些条件的数据，当T2再次按同样的条件读取，发现记录增多。

SQL的隔离性级别：Serializable，一个调度的执行必须等价于串行调度结果（不会发生不一致）；repeatable read，只允许读取已经提交的记录，并且要求一个事务对同一记录的两次读取之间，其他事物不能对其进行更新（会发生幻象）；read committed，只允许读取已经提交的事务，但不要求可重复读取（会发生幻象，并且不能重复读）；read uncommitted，允许读取未提交的记录（发生幻象，不能重复读，会读脏数据）。

9. 并发控制

事务调度器：产生调度，决定哪些事务执行，哪些事务回滚。

事务协议：不先产生调度再判别是否可串行化，而是根据某些约束，使得事物自然可以保持可串行化调度。

悲观的控制协议：通过锁机制实现并发，采取**等待**方法避免冲突。若一个事务可以获取其需要的锁，则事务指令一定能够执行完成；可能会出现死锁。

乐观的控制协议：假设冲突的概率较小，大部分事务只读，采取**回滚**的方法来避免冲突。

基于锁的并发控制协议

排它锁X：数据项可被读写，使用lock-X指令进行请求。

共享锁S：数据项只能被读，使用lock-S指令进行请求。

锁兼容矩阵：若事务在某数据项的锁请求与其他事务在该数据项上已有的锁兼容，则请求被批准。

	S	X
S	True	False
X	False	False

基于锁的协议：任意数目的事务可以同时对一个数据持有共享锁，若一个事务对某数据持有排它锁，则其他事物不得持有该数据上的任何锁；若不能授予锁，则请求锁的事务等待，直至持有不兼容锁的其他事物释放，之后才可以授权。

两阶段锁协议：阶段1为增长阶段，事务可以获得锁但不能释放锁；阶段2为收缩阶段，事务可以释放锁但不能获取锁。可以确保冲突可串行化调度的协议，事务可以按照lock points（事务获取最后一个锁的点）的次序串行化。不能避免死锁，不能避免级联回滚。

短锁：在事务中途就可以释放的锁。短X锁可能导致级联回滚。

长锁：保持到事务结束才释放的锁。

严格的两阶段锁协议：事务必须保持其所有排它锁直至提交或失败。不能避免死锁，可以避免级联回滚，不能读其他事物没有提交的数据。

严密的两阶段锁协议：所有锁都必须保持到事务提交或失败，事务按照其提交的次序进行串行化。

带有锁转换的两阶段锁协议：增长阶段允许获得S锁和X锁，并允许将S锁升级为X锁，缩减阶段允许释放S锁和X锁，并允许将X锁降级为S锁。

锁的自动获得：事务发出标准的R/W指令，无需显示进行封锁调用。

```
READ(D):
if 该事务在D上有锁 then
    read(D)
else
    if 其余事务在D上有X锁 then
        等待直至没有事务在D上有X锁
    该事务在D上获取S锁
    read(D)
Write(D):
if 该事务在D上有X锁 then
    write(D)
else if 该事务在D上有S锁 then
    将该S锁升级为X锁
    write(D)
else
    if 其余事务在D上有锁 then
```

```
        等待直至没有事务在D上有锁
    该事务获取X锁
    write(D)
```

锁管理器：事务向锁管理器发送加锁和解锁请求，锁管理器发送授予消息来回答锁请求，或在死锁时发送回滚消息；事务发送请求后将等待，直至其请求被回答。

锁表：记录授予锁和挂起的请求的数据结构，通常表现为以被锁数据项名字为索引的内存散列表。

基于图的协议：要求数据访问按照特定的模式（具有偏序关系）进行。在所有数据项集合D={d1, ..., dn}上施加一个偏序，若$d_i \rightarrow d_j$则存取d_i和d_j的任何事务都必须先d_i后d_j，集合D可以视作有向无环图。通过偏序关系（DAG）避免死锁。

基于树的协议：图协议的简单形式。数据项上只允许排它锁，事务可以对任何数据项加第一个锁，之后事务对数据Q加锁当且仅当Q的父节点已经被加锁。数据项可以在任意时刻被释放锁（保证父子加锁的关系）。事务不允许在解锁数据项后再次加锁。

树协议中，哪个事务先获得锁，哪个事务先执行完成。一步领先，要保证步步领先，从而可以满足可串行化。

树协议不保证可恢复性，可以通过增加事务结束前不允许释放排它锁的约束来进行增强。

树协议的父节点锁要求，保证了加锁数据次序，从而确保产生可串行化调度。

树协议的排它锁要求，避免死锁，保证可串行化调度。

多粒度锁：定义数据粒度层次，允许数据项具有不同的大小，小粒度嵌入在大力度之中，将数据看作一棵树。当事务对树中节点加锁，它也对节点的所有后代节点加了同样的锁。细粒度并发度高且开销大，粗粒度并发度低开销小。

意向锁I：为某个节点加I锁，表明其内层节点已经发生事实上的封锁，防止其他事务再显式封锁该节点。I锁实施从根节点开始，沿路径加锁至显示加锁节点的父节点。

IS锁：对一个数据项加IS锁，说明其子孙节点有意向加S锁。

IX锁：对一个数据项加IX锁，说明其子孙节点有意向加X锁。

SIX锁：对一个数据项加SIX锁，表示对它加S锁，之后再加IX锁。

请求锁	已有IS	已有S	已有IX	已有SIX	已有X
IS	True	True	True	True	False
S	True	True	False	False	False
IX	True	False	True	False	False
SIX	True	False	False	False	False
X	False	False	False	False	False

多粒度锁：必须遵守如上锁兼容矩阵，加锁时由根到叶，根必须先被加锁，解锁时由叶到根，加锁规则遵守下表。

父节点加锁	子节点允许加的锁类型
IS	IS，S
IX	IS，S，IX，X，SIX
S	None
SIX	X，IX，SIX
X	None

死锁

锁协议的缺陷：具有死锁的危险性，具有饥饿的危险性。

死锁的条件：**互斥条件**，事务请求对资源的独占控制；**占有等待条件**，事务已经持有一定资源，又去申请并等待其它资源；**非抢占条件**，直到资源被持有它的事务释放前，它不可能被夺去；**循环等待条件**，存在事务互相等待的环。**当互斥条件，占有等待条件，非抢占条件满足时，循环等待条件是死锁的充要条件。**

死锁预防：保证系统永远不进入死锁。可能会回滚本不需要回滚的事务。

全或无封锁：一个事务必须预先占据所需的全部资源，要么一次性全部封锁，要么全不封锁。难点在于数据实用率地下但难于预知需要封锁那些资源。

资源排序：对资源进行预先排序，事务必须按顺序封锁资源。图协议。

事务排序：给事务分配时间戳，根据时间戳确定事务在获取资源时的优先级。

等待-死亡方案Wait-Die：非抢占式的事务排序死锁预防方案。老事务可以等待年轻事务释放数据项，年轻事务永远不会等待老事务而是回滚。事务越老越可能等待，新事物可能在获取数据项前死亡若干次。

伤害-等待方案Wound-Wait：抢占式的事务排序死锁预防方案。老事务强制回滚年轻事务而不等待，年轻事务可能等待老事务。可能会比等待-死亡方案有较少的回滚。

死锁检测和死锁恢复：允许进入死锁状态，运用死锁检测和死锁恢复进行恢复，需要进行回滚。

基于超时的死锁处理方案：介于预防和检测恢复之间。事务对一个锁只等待一个指定的时间，超时之后事务回滚。实现简单但会饥饿，超时间隔难以确定。

死锁检测：使用等待图G=(V,E)描述，若$T_i \rightarrow T_j$，则存在T_i等待T_j占据的资源。系统死锁，当且仅当G中存在圈。

死锁恢复：当检测到死锁时，确定哪些事务作出回滚（选择代价最小的事务进行牺牲），确定事务回滚的程度（完全回滚v.s.部分回滚至打破死锁），避免反复选择同一事务牺牲（发生饿死）。

乐观的并发控制协议

基于有效性检查的协议：读与执行阶段，具有时间戳START(T)，事务的写操作只写到临时局部变量中；有效性检查阶段，具有时间戳VALIDATION(T)，事务执行有效性检查，决定局部变量的值是否可以写回而不违反可串行化；写阶段，具有时间戳FINISH(T)，若事务通过有效性检查则更新数据库，否则回滚。为了增加并发性，可串行化次序由有效性检查时间戳的次序决定。

测试条件1：对于所有i，满足$T_i < T_j$，检查T_i是否在T_j开始之前结束

测试条件2：对于所有i，满足$T_i < T_j$，并且T_i结束的时间在T_j开始写之前，检查

$$WriteSet(T_i) \cap ReadSet(Tj) = \Phi$$

测试条件3：对于所有i，满足$T_i < T_j$，并且T_i在T_j开始写之后结束，检查

$$WriteSet(T_i) \cap ReadSet(Tj) = \Phi$$
$$WriteSet(T_i) \cap WriteSet(Tj) = \Phi$$

基于有效性检查的协议判定冲突的粒度较粗，会出现误判定为冲突的情况。

基于时间戳的协议：事务Ti进入系统时赋予时间戳TS(Ti)。时间戳决定串行化次序，违反的事务主动回滚。数据项Q维护两个时间戳W-timestamp(Q)和R-timestamp(Q)用于记录成功执行了WRITE(Q)和READ(Q)的所有事物中最大的时间戳。

事务T_i发出$READ(Q)$

1) 若$TS(T_i) < W-timestamp(Q)$，则T_i需要读Q的已经被覆盖了的值，拒绝该$READ$操作，T_i回滚。
2) 若$TS(T_i) \geq W-timestamp(Q)$，则$READ$操作可以执行，$R-timestamp(Q)$被置为$\max\{R-timestamp(Q), TS(T_i)\}$。

事务T_i发出$WRITE(Q)$

1) 若$TS(T_i) < R-timestamp(Q)$，则T_i要产生的Q是曾经需要的值，而系统已经前进并假设该值不会产生了，拒绝该$WRITE$操作，T_i回滚。
2) 若$TS(T_i) < W-timestamp(Q)$，则T_i试图写一个Q的已经被覆盖了的值，拒绝该$WRITE$操作，T_i回滚。
3) 否则，执行$WRITE$操作，$W-timestamp(Q)$被置为$TS(T_i)$。

时间戳协议的优先图中边均从小TS事务指向大TS事务，因此优先图没有圈；不可能出现死锁，因为事务直接主动回滚，不会有事务等待；会有级联回滚的可能性，会出现不可恢复。

Thomas写规则：某一些写操作是无效的，因此过时的WRITE操作在一些情况下可以被忽略掉，从而避免回滚，允许更多潜在的并发性。

$Thomas$写规则下对时间戳协议的修改

事务T_i发出$WRITE(Q)$时，若$TS(T_i) < W-timestamp(Q)$，则T_i试图写一个过时的Q，忽略该$WRITE$操作而非拒绝并回滚T_i。

其余同时间戳协议相同。

读写冲突和写读冲突：读写冲突为当前事务视图读被年轻事务写的值覆盖的值时发生的冲突，写读冲突为当前事务视图写年轻事务读过的值的旧值时发生的冲突，二者均通过当前事务主动回滚的方式解决。

读操作频繁时，读写冲突可以通过保留旧版本的方式来解决，从而提高并发度。

写写冲突：在没有读写冲突和写读冲突时，当前事务视图写已经默认被年轻事务写过并覆盖的值时发生的冲突，通过Thomas写规则直接忽视解决。

多版本机制：数据库暂时保持历史数据，从而提高并发度。每个成功的WRITE操作都创建数据项的一个新版本；事务进入Active状态时，整个数据库的所有数据的最新版本是一致的，该事务在该一致的数据库上操作；当事务提交READ(Q)是，基于事务的TS选择一个合适的版本。

多版本时间戳序列：每个数据项Q具有一个版本序列<$Q_1, Q_2, \cdots, Q_m$>，每个Q_k中记录版本Q_k的内容C，创建版本Q_k的事务的时间戳W-timestamp(Q_k)和成功读取版本Q_k的事务集合中最大的时间戳R-timestamp(Q_k)。当事务T_i创建一个数据项新版本Q_k其W-timestamp和R-timestamp均被初始化为$TS(T_i)$

多版本时间戳协议：按照时间戳序列保证可串行化事务调度。假设事务T_i发出READ(Q)或者WRITE(Q)操作，Q_k表示Q的一个版本，其W-timestamp是比$TS(T_i)$小的最大值，即Q_k是Ti进入Active状态时看到的一致数据库中的数据项。

若T_i发出$READ(Q)$命令，则Q_k的内容C直接返回，
　　并且设置$R-timestamp(Q_k)=\max\{R-timestamp(Q_k),TS(T_i)\}$
若T_i发出$WRITE(Q)$命令
　1) 若$TS(T_i)<R-timestamp(Q_k)$，则事务T_i回滚
　2) 若$TS(T_i)=W-timestamp(Q_k)$，则直接覆盖Q_k中的内容
　3) 创建新的Q的版本，并设置$TS(T_i)$为该版本的$R-timestamp$和$W-timestamp$

多版本两阶段锁协议：使用多版本，使得读事务不能被阻塞；使用两阶段锁，使得事务更新等待而不回滚。序列化的次序需要同时考虑进入系统的时间（系统时间戳）和封锁点的次序。**只读事务**，只会读取数据项，其时间戳设置为事务启动时的系统逻辑时间戳ts值。**更新事物**，需要读锁和写锁，遵循严密两阶段锁协议将其持有到事务提交的最后；每次成功的WRITE操作都会创建一个新版本，版本时间戳使用ts，ts由系统维护，初值为0，每个更新事务提交时加1。

对于更新事务T_i
　读一个数据项时，获取该数据项S锁，读取其最后版本
　写一个数据项时，获取该数据项排它锁，创建新版本并设置其时间戳为∞
　提交时，将其创建的版本的时间戳更改为$ts+1$，并将ts更新为$ts+1$
对于T_i之后启动的只读事务T_j，可以看到T_i更新的值
对于T_i提交之前启动的只读事务T_j，看不到T_i更新的值

插入和删除

两阶段锁协议下：仅有当事务具有对删除元祖的X锁时，DELETE操作可以执行；事务插入元组时，获得该元组的X锁。

时间戳协议下：删除操作等同于Write操作，插入操作创建新数据并为其设置时间戳。

插入的幻影现象：只使用细粒度的元组锁时，会出现影响事务隔离性的，且无法检测到的冲突。eg. 扫描事务连续两次读表，两次读表之间，另一个插入事务完成插入并提交，会导致扫描事务两次读取的结果不一致，会出现幻影。

删除的幻影现象：删除事务在获得元组X锁的条件下完成删除，但删除也会自动地删除了锁，这会导致隔离性被破坏。eg. 删除事务先删除了元组，之后获取另一元组的X锁并进行更新，在这两个操作之间有另一个扫描事务完成读表并提交，获取到了删除事务的中间状态，出现幻影。

消除幻影现象：使用粗粒度锁，例如表级锁和索引锁。

10. 数据库恢复

故障和存储器

事务故障：事务没有运行到达预期的终点处，就被终止。**预期故障**，不能由事务程序处理的故障，例如运算溢出，死锁回滚等；**可预期故障**，可以有应用程序发现的事务故障，并由应用程序主动回滚，例如转账时出现金额不足。

系统故障：软故障，引起内存信息丢失，但外存数据正常的故障，例如突然断电。

介质故障：硬故障，磁盘故障，外存上的数据库损坏，例如磁盘被摧毁。

恢复：将数据库从错误状态恢复到某个正确状态的功能，从而确保了数据库的一致性。**恢复的基本原理是冗余。**

存储器：易失性存储器，无法在系统崩溃后保存，eg. 主存，cache；非易失性存储器，可以在系统崩溃后保存下来，eg. 磁盘，磁带，电池供电的非易失性RAM；稳定存储器，使用多个非易失性介质存储相同的副本，构成的虚构的能够经受任何故障的存储器。

块：物理块石位于磁盘上的块，缓冲块是临时位于主存中的块。

局部副本：每个事务Ti都有自己的私有工作区，用来保存它存取和更新的所有数据项的局部副本，Ti的对应于数据项X的局部副本记作Xi。

块移动原语：INPUT(B)原语操作，将物理块B传入主存；OUTPUT(B)原语操作，将缓冲块B传到磁盘并替换相应物理块。

事务传送数据原语：事务通过READ和WRITE来在缓冲块和它的私有工作区之间传送数据项。这两条命令在赋值前都有可能发出INPUT指令，而OUTPUT指令不必紧随WRITE执行，而是在系统认为合适的时候执行。

$$\text{事务工作区} \xleftrightarrow{READ/WRITE} \text{缓冲区} \xleftrightarrow{INPUT/OUTPUT} \text{磁盘}$$

缓冲区管理：数据缓冲区根据数据块的访问频率，尽可能地采取和事务管理器**相对独立**的策略，尽可能减少flush，提高IO效率。因而会有一下情况1) 事务提交了，缓冲区还没有刷新到磁盘上；2) 事务没有提交，临时的变动刷新到了磁盘上；3) 所有事务共享缓冲区，即使没有写入磁盘，其他事务也能读到更新后的数据。

恢复的原子性的保证：为了在故障的情况下确保原子性，首先向稳定存储器写描述更新的信息，而不是直接更新数据库本身。

串行事务的基于日志的恢复

日志记录：记录每个事务的所有操作，记录内容包括<Ti, X, V1, V2>，其中Ti为事务，X为数据项，V1位更新前的旧值，V2位更新后的新值。

日志：日志记录的序列，用于记录对数据库的更新活动。事务Ti启动时先写入登记自己的日志记录<Ti START>，之后每有更新则写入记录，提交后写入提交记录<Ti COMMIT>。

延迟更新数据库：在日志中记录所有的更新，但推迟WRITE直至部分提交之后。假设事务串行执行。

```
1. 事务启动，向日志写入<Ti START>记录
2. WRITE(X)操作，向日志写入<Ti, X, V>记录，但执行真正的写操作，该方案不需要保留旧值
3. 事务部分提交，向日志写入<Ti COMMIT>记录
4. 读取日志记录，执行延迟的WRITE操作
```

延迟更新的恢复：故障可能发生在事务运行状态时，或事务利用日志恢复状态时。恢复过程中，对日志中存在<Ti START>和<Ti COMMIT>d的事务进行重做（REDO），重做时将事务的所有数据项按顺序更新为新值；不完整的日志记录可以直接删除。

立即更新数据库：立即更新数据库方案允许活跃的未提交事务在发出写指令时直接进行修改（未提交修改），更新数据库缓冲区。由于可能需要回滚，因而日志记录必须包含新值和旧值。

立即更新的恢复：**UNDO(Ti)**操作，复原所有被Ti更新了的数据项的旧值，从Ti的最后一条日志记录向前进行；**REDO(Ti)**操作，将所有被Ti更新了的数据项的值置为新值，从第一条日志记录向后进行。UNDO和REDO都是幂等的，即不论执行多少次，效果都和执行一次相同；UNDO先做，REDO后做。

```
1. 若日志中包含<Ti START>但没有包含<Ti COMMIT>，则UNDO(Ti)
2. 若日志中包含<Ti START>和<Ti COMMIT>则REDO(Ti)
```

检查点：引入检查点，使得事务在检查点之前的所有修改都写入数据库中，检查点之前不需要进行REDO。在日志中写入<CHECKPOINT>记录表示检查点位置。

检查点恢复：假设串行调度，仅需要考虑检查点之前启动的最近的事务Ti，以及Ti之后启动的事务。

```
1. 从日志末尾反向扫描，找到最近的<CHECKPOINT>记录
2. 继续反向扫描直至第一个<Ti START>记录，只需考虑Ti及其之后的部分，之前的日志在恢复过程中忽略，需要的时刻（例如磁盘不足时）可以删除
3. 从检查点起，正向扫描，对于所有没有<Tj COMMIT> (j>=i)的事务，执行UNDO(Tj)
4. 从检查点起，反向扫描，对于所有由<Tj COMMIT> (j>=i)的事务，执行REDO(Tj)
```

并发事务的基于日志的恢复

并发事务的日志：所有事务共享单个磁盘缓冲区和单个日志，日志记录的结构不变，不同事务的日志记录会在日志文件之中交错分布。检查点日志记录修改为<CHECKPOINT L>，L为检查点时活跃事务的列表。

REDO-LIST和UNDO-LIST：REDO-LIST为需要进行REDO操作的事务列表，UNDO-LIST为需要进行UNDO操作的事务列表。获取REDO-LIST和UNDO-LIST的算法如下。

```
初始化，UNDO-LIST和REDO-LIST为空
从日志末尾反向扫描日志，直至发现第一条<CHECKPOINT L>记录时停止：
    对于反向扫描过程中发现的记录
    若为<Ti COMMIT>，则将Ti加入REDO-LIST之中
    若为<Ti START>，并且Ti不在REDO-LIST之中，则将其加入UNDO-LIST
对于L中的每一个Ti，若Ti不在REDO-LIST之中，则将其加入UNDO-LIST
```

并发事务的恢复：反向扫描日志进行UNDO直至UNDO-LIST空，从检查点起正向扫描日志进行REDO，UNDO先进行，REDO后进行。

```
从日志末尾起反向扫描日志，直至UNDO-LIST空：
    对于每条记录
    若其为<Ti START>并且Ti位于UNDO-LIST中，则从UNDO-LIST删除该事务
    若其对应事务Ti位于UNDO-LIST中，则对该记录执行UNDO
从<CHECKPOINT L>记录起正向扫描日志，直至日志末尾：
    对对应事务<Ti>位于REDO-LIST中的记录，进行REDO
```

日志记录缓冲：主存中对日志记录缓冲，而不是直接输出至稳定储存器，减少IO降低开销。

缓冲日志记录的原则：日志记录按创建次序输出到稳定储存器；当<Ti COMMIT>输出到稳定存储器，Ti才被视作完成提交；主存数据块输出至数据库前，所有和该块中数据相关的日志记录必须已经输出到稳定储存器（**先写日志规则，WAL**）。

LSN：日志记录的序列编号，通过LSN链表可以快速构造日志链。

基于影子页面的恢复

影子页面的基本思想：在事务执行期间，保持两个页面表——当前页面表和影子页面表。在事务运行之前，保存影子页面到非易失性存储器中，这些页面在运行期间不被修改。当系统崩溃时，不需要恢复工作，直接用影子页面替换当前页面，即可马上开始新的事务。

影子页面：事务开始运行时，当前页面表和影子页面表相同，当前页面表会在运行过程中进行修改。当事务提交后，输出当前页面表到磁盘，磁盘中记录影子页面的指针指向当前页面表，当前页面表变为新的影子页面表。

任何页面将要被第一次写入时
赋值该页面的内容到未使用页面之中
当前页面表指向保存了复制内容的页面地址
所有的更新操作针对当前页面表指向的复制页面进行

影子页面的优势：不需要写日志；恢复操作简单，不需要UNDO和REDO。

影子页面的缺点：复制页面表代价高昂；提交代价高昂；数据分散；每次事务结束后都需要进行页面垃圾回收；难以扩展至多用户并发。