数据库系统 期末速通教程

7. 数据管理

7.1 可串行化调度

[**定义 7.1.1**] [**交易**, **Transaction**] **交易**是有如下四条性质的程序单元:

(1) [**原子性**, **Atomicity**] 要么全部成功, 要么全部失败.

(2) [**一致性**, **Consistency**] 将 DB 从一个一致性状态转移到另一个一致性状态.

(3) [**独立性**, **Independence**] 一个交易的执行与其它交易独立, 一个交易可与其它交易同时执行.

(4) [**耐用性**, **Durability**] DB 的故障不能导致已完成的交易被撤销. 一旦提交, 效果在失败后可恢复.

[**定义 7.1.2**]

(1) [**有效调度**, **Valid Schedule**] n 个交易 $T_1, \cdots, T_n$ 的**有效调度** S 是一个顺序, s.t. 对每个交易 T_i $(1 \leq i \leq n)$, T_i 在 S 中的操作需以与 T_i 中相同的顺序出现.

(2) 符号:

① $R_i(X)$: 交易 T_i $\mathrm{read}(X)$.

② $W_i(X)$: 交易 T_i $\mathrm{write}(X)$.

[**例 7.1.1**] 交易 $T_1 : R_1(X)\ W_1(X)$, $T_2 : R_2(X)\ W_2(X)$.

(1) 有效调度: $R_1(X)\ R_2(X)\ W_1(X)\ W_2(X)$.

(2) 无效调度: $W_1(X)\ R_1(X)\ R_2(X)\ W_2(X)$.

[**定义 7.1.2**] [**串行调度**, **Serial Schedule**] 若一个调度中不同交易间的操作不交叉, 则称该调度是**串行调度**.

[**注**]

(1) 只允许串行调度性能很低.

(2) 串行调度可保证交易的一致性, 但保证交易的一致性未必需串行调度.

[**例 7.1.2**]

(1) 串行调度: $R_1(X)\ W_1(X)\ R_2(X)\ W_2(X)$, $R_2(X)\ W_2(X)\ R_1(X)\ W_1(X)$.

(2) 非串行调度: $R_1(X)\ R_2(X)\ W_1(X)\ W_2(X)$.

[**定义 7.1.3**]

(1) [**冲突操作**, **Conflict Operations**] 一个调度中的两个操作**冲突**, 如果它们属于不同的交易但访问相同的数据, 且其中至少一个操作为写.

(2) 分类:

① 读写冲突: $(R_1(X), W_2(X))$, $(W_1(X), R_2(X))$.

② 写写冲突: $(W_1(X), W_2(X))$.

[**定义 7.1.4**] [**等价调度**, **Equivalent Schedules**] 称对一组交易的两个调度是(**冲突**)**等价的**(conflict equivalent), 如果两交易中任意两冲突操作的顺序都相同.

[**定义 7.1.5**] [**可串行化调度**, **Serializable Schedules**] 称一个调度是(**冲突**)**可串行化的**(conflict serializable), 如果它等价于至少一个串行调度.

[**注**]

(1) 可串行化调度可保证交易的一致性.

(2) 非串行但可串行化的调度常允许更高的并发度.

[**例 7.1.3**] 设交易 $T_1 = R_1(X)\ W_1(X)\ R_1(Y)\ W_1(Y)$, $T_2 = R_2(X)\ W_2(X)$.

如下两调度等价:

① $S_1 : R_1(X)\ W_1(X)\ R_2(X)\ W_2(X)\ R_1(Y)\ W_1(Y)$.

② $S_2 : R_1(X)\ W_1(X)\ R_1(Y)\ W_1(Y)\ R_2(X)\ W_2(X)$.

因 S_1 等价于串行调度 S_2 , 则 S_1 是可串行化调度.

[**算法 7.1.1**] [**Test Schedule Serializability**, **TSS**]

(1) 输入: n 个交易 $T_1, \cdots, T_n$ 的一个调度 S .

(2) 输出: 判断 S 是否可串行化.

(3) 算法:

① 建立 S 的**前驱图**(precedence gragh) G :

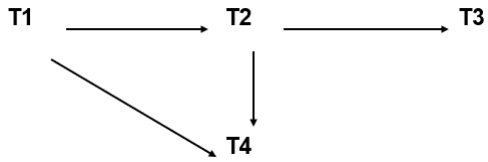
(i) 每个交易 T_i $(i = 1, \cdots, n)$ 是图中的一个节点.

(ii) 对每对冲突操作 (op_1, op_2) , 若 $op_1 \in T_i$, $op_2 \in T_j$, 且 op_1 在 op_2 前发生, 则 T_i 向 T_j 连有向边.

② 对 G 拓扑排序, 若无环(即 S 是 DAG), 则 S 是可串行化的, 且任一拓扑排序的结果是其等价的串行化调度;

否则 S 不可串行化.

[**例 7.1.4**] 调度 $S: R_1(X)\ R_2(Y)\ W_1(X)\ R_2(X)\ W_2(Y)\ W_2(X)\ R_3(Y)\ W_3(Y)\ R_4(X)\ W_4(X)$ 的前驱图如下:



拓扑排序的结果为: ① $T_1\ T_2\ T_3\ T_4$ 或 ② $T_1\ T_2\ T_4\ T_3$.

故 S 可串行化, 且等价于上述两个串行化调度.

7.2 锁

[**定义 7.2.1**]

(1) [**基于锁的并发控制协议**, **Locking-based Concurrency Control Protocols**] 在并发访问中为数据加锁, 以保证调度可串行化.

(2) 锁的分类:

① **读锁** $rl_i(X)$ 表示在交易 T_i 中为数据 X 加读锁.

② **写锁** $wl_i(X)$ 表示在交易 T_i 中为数据 X 加写锁.

(3) 释放锁 $ul_i(X)$ 表示在交易 T_i 中释放数据 X 的锁.

(4) 加锁规则:

① 只读: 加读锁.

② 只写: 加写锁.

③ 又读又写: 加写锁.

(5) 一个数据可加多个**兼容**(compatible)的锁, 但不能加两个**不兼容**(incompatible)的锁.

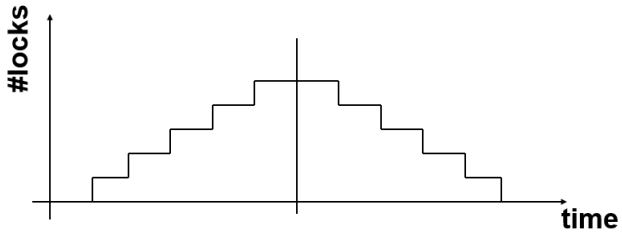
锁的兼容性:

	$rl_i(X)$	$wl_i(X)$
$rl_j(X)$	兼容	不兼容
$wl_j(X)$	不兼容	不兼容

[**定义 7.2.2**]

(1) [**两阶段锁协议**, **Two-phase Locking Protocol**, **2 PL**] 一个交易释放一个锁后不能再加锁.

(2) 两阶段:



① [**增长阶段**, **Growing-phase**] 若需加锁, 则加锁.

② [**缩减阶段**, **Shrinking-phase**] 释放无需的锁.

(3) 对一组交易, 可能存在多个满足 2 PL 的调度, 可加限制 s.t. 满足 2 PL 的调度唯一:

① [**First-come-first-service**] 在满足 2 PL 的前提下, 先到达的请求先进行.

② [**Release a lock as early as possible**] 在满足 2 PL 的前提下, 不需要的锁尽早释放.

[**注**]

(1) 满足 2 PL 的调度可串行化.

(2) 2 PL 可能导致级联终止.

(3) 2 PL 无法避免死锁.

[**例 7.2.1**] 交易 $T: R(X)\ X = X + 1\ W(X)\ R(Y)\ Y = Y - 1\ W(Y)$.

(1) 调度 $S_1: wl_1(X)\ R_1(X)\ W_1(X)\ ul_1(X)\ wl_1(Y)\ R_1(Y)\ W_1(Y)\ ul_1(Y)$ 不满足 2 PL .

(2) 如下两调度满足 2 PL :

① 调度 $S_2: wl_1(X)\ R_1(X)\ W_1(X)\ wl_1(Y)\ R_1(Y)\ W_1(Y)\ ul_1(X)\ ul_1(Y)$.

② 调度 $S_3: wl_1(X)\ R_1(X)\ W_1(X)\ wl_1(Y)\ ul_1(X)\ R_1(Y)\ W_1(Y)\ ul_1(Y)$.

7.3 死锁

[**定义 7.3.1**]

(1) [**死锁**, **Deadlocked**] 若存在一组交易, 其中每个交易都在等待另一交易, 则系统发生**死锁**.

(2) [**死锁避免协议**, **Deadlock Prevention Protocols**]

① [**预声明**, **Predeclaration**] 每个交易执行前给它所需的所有数据加锁.

② [**基于图的协议**, **Graph-based Protocol**] 将所有数据部分排序, 要求交易只能按指定顺序加锁数据.

③ [**非抢占性方案**, **Wait-die Scheme**, **non-preemptive**] 老交易可能等待新交易释放数据; 新交易不等待老交易, 进行回滚. 一个交易在得到所需的数据前可能会 die 若干次.

④ [**抢占性方案**, **Wound-wait Scheme**, **preemptive**] 新交易迫使老交易回滚, 新交易可能等待老交易. 该方案比 Wait-die Scheme 的回滚次数更少.

[**算法 7.3.1**] 周期性地死锁检测算法判断是否发生死锁.

(1) 建立**等待图**(wait-for graph) $G=(V,E)$:

① 每个交易是图中的一个节点.

② 若交易 T_i 等待交易 T_j 释放数据, 则连有向边 $T_i \to T_j$.

③ 若交易 T_i 不再等待交易 T_j 释放数据, 则删除有向边 $T_i \to T_j$.

(2) 对 G 拓扑排序, 若有环, 则发生死锁.

[**注**] 检测到死锁后, 选择一个交易作为**受害者**(victim)回滚, 以破坏死锁. 高效的回滚是回滚至恰破坏死锁.