第十二章

实践习题

12.1

（在本题中为了简单起见）假设一个块中只能放入一个元组，并且内存最多容纳 3 个块。当应用排序-归并算法对下述元组按第一个属性进行排序时，请给出各趟所产生的归并段：(kangaroo,17)、(wallaby,21)、(emu,1)、(wombat,13)、(platypus,3)、(lion,8)、(warthog,4)、(zebra,11)、(meerkat,6)、(hyena,9)、(hornbill,2)、(baboon,12)。

我们将使用元组编号 t1 到 t12 来引用元组(kangaroo, 17)到(baboon, 12)。初始归并的归并段如下:

$$r_{11} = \{t_3, t_1, t_2\}$$
$$r_{12} = \{t_6, t_5, t_4\}$$
$$r_{13} = \{t_9, t_7, t_8\}$$
$$r_{14} = \{t_{12}, t_{11}, t_{10}\}$$

第一趟归并段结束后是:

$$r_{21} = \{t_3, t_1, t_6, t_9, t_5, t_2, t_7, t_4, t_8\}$$
$$r_{22} = \{t_{12}, t_{11}, t_{10}\}$$

第二趟归并结束时，元组被完全排序为最终结果:

$$r_{31} = \{t_{12}, t_3, t_{11}, t_{10}, t_1, t_6, t_9, t_5, t_2, t_7, t_4, t_8\}$$

上述方案没有为输出保留一个块，若与书上示例一样为输出保留一个块更好。

12.7

请写出实现索引嵌套-循环连接的迭代算子的伪码，其中外层关系是流水线化的。要求伪码必须定义标准的迭代算子函数 open()、next()和 close()。请给出在不同调用之间迭代算子必须维护的状态信息是什么。

设 outer 为迭代器，从流水线化的外部关系返回连续元组。设 inner 为迭代器，该迭代器返回在 join 属性处具有给定值的内部关系的连续元组。内部迭代器通过执行索引查找返回这些元组。函数 IndexedNLJoin::open、IndexedNLJoin::close 和 IndexedNLJoin::next 实现了索引嵌套循环连接迭代器，如下所示。两个迭代器 outer 和 inner，最后一个读取的外部关系元组t_r的值和一个标志$done_r$，指示是否已经到达外部关系扫描的结束，这些都是 IndexedNLJoin 在调用之间需要记住的状态信息。

```
IndexedNLJoin::open()
begin
    outer.open();
    inner.open();
    done_r := false;
    if(outer.next() ≠ false)
        move tuple from outer's output buffer to t_r;
    else
        done_r := true;
end

IndexedNLJoin::close()
begin
    outer.close();
    inner.close();
end
```

```
boolean IndexedNLJoin::next()
begin
    while(¬done_r)
    begin
        if(inner.next(t_r[JoinAttrs]) ≠ false)
        begin
            move tuple from inner's output buffer to t_s;
            compute t_r ⋈ t_s and place it in output buffer;
            return true;
        end
        else
            if(outer.next() ≠ false)
            begin
                move tuple from outer's output buffer to t_r;
                rewind inner to first tuple of s;
            end
            else
                done_r := true;
    end
    return false;
end
```

12.18

请考虑关系 r1(A,B,C)、r2(C,D,E)和 r3(E,F),它们的主码分别为 A、C 和 E。假设 r1 有 1000 个元组，r2 有 1500 个元组，r3 有 750 个元组。请估计 r1⋈r2⋈r3 的规模，并给出一种高效的策略来计算该连接。

由于连接的结合律和交换律，r1 r2 r3 的连接所得到的关系无论我们用哪种方式连接它们都是一样的。因此，我们将根据((r1⋈r2)⋈r3)的策略来考虑规模大小。连接 r1 和 r2 最多将产生 1000 个元组的关系，因为 C 是 r2 的主码。同样地，将该结果与 r3 连接将产生最多 1000 个元组的关系，因为 E 是 r3 的主码。因此，最终的关系最多有 1000 个元组。
计算这个连接的有效策略是在属性C上为关系r2创建索引,在属性E上为关系r3创建索引。

然后，对于 r1 中的每个元组，我们执行以下操作:

在 r2 中使用在 C 上创建的索引查找最多一个与 r1 的 C 值匹配的元组。

在 r3 中使用在 E 上创建的索引查找最多一个与 r2 中 E 的唯一值匹配的元组。

12.19

请考虑实践习题 12.18 中的关系 r1(A,B,C)、r2(C,D,E)和 r3(E,F)。假设除了整个模式外不存在主码。令 V(C,r1)为 900，V(C,r2)为 1100，V(E,r2)为 50，V(E,r3)为 100。假设 r1 有 1000 个元组，r2 有 1500 个元组，r3 有 750 个元组。请估计 r1⋈r2⋈r3 的规模，并给出一种高效的策略来计算该连接。

关系的估计大小可以通过计算与第二个关系的每个元组连接的元组的平均数量来确定。在这种情况下，对于 r1 中的每个元组，r2 中的 1500/V(C,r2)= 15/11 个元组(平均)将与它连接。中间的关系会有 15000/11 个元组。将这个关系与 r3 连接，得到大约 10227 个元组(15000/11 × 750/100 = 10227)。好的策略应该先连接 r1 和 r2，然后再与 r3 连接。因为 r1 和 r2 连接的中间关系的大小和 r1 或 r2 差距不大，而 r2 和 r3 连接的中间关系的大小远超 r2 和 r3。

习题

12.29

假设你需要对一个 40GB 的关系进行排序，使用 4KB 规模的块以及 40MB 规模的内存。假设一次寻道的代价是 5 毫秒，而磁盘传输速率是每秒 40MB。
a.对于b_b=1 和b_b=100 的情况，以秒为单位分别计算对该关系进行排序的代价。
b.在上述每种情况下，各需要多少遍归并。
c.假设使用一个闪存设备来替代磁盘，并且它的延迟为 1 微秒，而传输速率是每秒 40MB，在这样的设置下，对于b_b=1 和b_b=100 的情况，以秒为单位分别重新计算对关系排序的代价。

a.主存缓冲区中可用于排序的块数(M)是 $\frac{40\times10^6}{4\times10^3}=10^4$.包含给定关系(br)的记录的块数为 $\frac{40\times10^9}{4\times10^3}=10^7$。那么排序关系的代价为:(磁盘寻道次数×磁盘寻道代价)+(块传输次数×块传输时间)。其中寻盘代价为 5×10^{-3}秒，并且块传输时间为10^{-4}秒$(\frac{4\times10^3}{40\times10^6})$。块传输次数和$b_b$无关，等于 25×$10^6$.

Case 1: b_b=1
磁盘寻道次数为 5002×10^3，因此排序关系的代价是(5002×10^3)×(5×10^-3)+(25×10^6)×(10^-4)=25×10^3+2500=27500 秒。
Case 2: b_b=100
磁盘寻道次数为 52×10^3，因此排序关系的代价是(52×10^3)×(5×10^-3)+(25×10^6)×(10^-4)=260+2500=2760 秒。

b.磁盘存储所需的合并次数由$\lceil log_{M-1}(\frac{b_r}{M})\rceil$给出。这与$b_b$无关。把上面的值代入，得到 $\lceil log_{10^4-1}(\frac{10^7}{10^4})\rceil$，结果为 1。

c.闪存存储：
Case 1: b_b=1
磁盘寻道次数为 5002×10^3，因此排序关系的代价是(5002×10^3)×(1×10^-6)+(25×10^6)×(10^-4)=5.002+2500=2505.002 秒。
Case 2:b_b=100
磁盘寻道次数为 52×10^3，因此排序关系的代价是(52×10^3)×(1×10^-6)+(25×10^6)×(10^-4)=0.052+2500≈2500 秒。

12.31
请设计一种混合归并-连接算法的变体来用于这种情况：两个关系都在物理上没有排序，但各自都有连接属性上的有序的辅助索引。

我们将第一个已排序二级索引的叶项与第二个已排序二级索引的叶项合并。结果文件包含一对地址，每对中的第一个地址指向第一个关系中的元组，第二个地址指向第二个关系中的元组。这个结果文件首先根据第一个关系的地址排序。然后按照物理存储顺序扫描该关系，结果文件中的地址被实际的元组值替换。然后根据第二个关系的地址对结果文件进行排序，允许按照物理存储顺序扫描第二个关系以完成连接。

12.36
假设 department 关系在(dept_name,building)上有 B+树索引可用。则处理下列选择的最佳方式是什么？

σ(building<"Watson")∧(budget<55000)∧(dept_name="Music")(department)

使用(dept_name,building)上的索引，我们定位第一个元组拥有(building=“Watson” and dept name=“Music”)。然后，只要 building 小于“Watson”，我们就按照指针检索连续的元组。从检索到的元组中，不满足(budget<55000)条件的元组被拒绝。

12.40

请解释如何使用直方图来估算形如 σ(A<=v)(r)的选择的规模。

假设存储关系 r 中值分布的直方图 H 被分为$r_1,\ldots,r_n$。对于每个范围r_i都具有较低的值$r_{i:low}$和高值$r_{i:high}$，如果$r_{i:high}$小于 v，我们将由 $H(r_i)$ 给出的元组的数量相加得到估计的总数。

如果 v <$r_{i:high}$，v >$r_{i:low}$，我们假设 ri 内的值是均匀分布的，我们将 $H(r_i) * \frac{v - r_{i:low}}{r_{i:high} - r_{i:low}}$ 加到估计的总数中。

第十三章

实践习题

13.6

请考虑图 13-16 所示的优先图，相应的调度是冲突可串行化的吗？请解释你的答案。

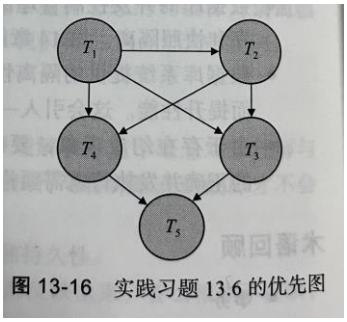


图 13-16 实践习题 13.6 的优先图

因为该图是无环的，并且该优先图存在相对应的可串行化次序，通过拓扑排序得到可能的调度，即 T1, T2, T3, T4, T5。所以相应的调度是冲突可串行化的。

13.7

什么是无级联调度？为什么要求无级联调度？是否存在要求允许级联调度的情况？请解释你的答案。

无级联调度：对于每一对事务 Ti 和 Tj 都满足如果 Tj 读取之前由 Ti 写入的数据项时，则 Ti 的提交操作必须出现在 Tj 的读取操作之前。

为什么需要无级联调度：级联回滚导致要撤销大量的工作，是不希望发生的，对调度施加限制避免发生级联回滚的情况是有必要的。

存在，如果很少发生故障，即因单个事务失效导致一系列事务回滚的现象很少发生，那么级联调度可能是可取的。

13.8

发生丢失更新异常是指如果事务 Tj 读取了一个数据项，然后另一个事务 Tk 写该数据项（可能基于先前的读取），这之后 Tj 再写该数据项。于是 Tk 所做的更新就丢失了，因为 Tj 所做的更新忽视了 Tk 所写的值。

a. 请给出一个表明丢失更新异常的调度示例。

a. A schedule showing the Lost Update Anomaly:

T_1	T_2
read(A)	
	read(A)
	write(A)
write(A)	

In the above schedule, the value written by the transaction T_2 is lost because of the write of the transaction T_1.

在上面的调度中，由于事务 T1 的写入，事务 T2 写入的值将丢失。

b. 请给出一个表明在已提交读隔离性级别下可能发生丢失更新异常的调度示例。

b. Lost Update Anomaly in Read Committed Isolation Level

T_1	T_2
lock-S(A)	
read(A)	
unlock(A)	
	lock-X(A)
	read(A)
	write(A)
	unlock(A)
	commit
lock-X(A)	
write(A)	
unlock(A)	
commit	

The locking in the above schedule ensures the Read Committed isolation level. The value written by transaction T_2 is lost due to T_1's write.

上述计划中的锁定可确保已提交读隔离级别。事务 T2 写入的值由于 T1 写入而丢失。

c. 请解释为什么在可重复读隔离性级别下丢失更新异常不可能发生。

在可重复读隔离级别中，读取数据项 X 的事务 T1 在 X 上持有共享锁直到结束。这使得新的事务 T2 不可能写入 X 的值(这需要 lock-X)，直到 T1 结束。这将强制序列化顺序为 T1, T2，因此由 T2 写入的值不会丢失。

13.11

一个调度的定义假设操作是可以完全按时间排列的。请考虑一个在具有多个处理器的系统上运行的数据库系统，它并不总是能对于运行在不同处理器上的操作确定一种准确的次序。但是一个数据项上的操作是完全可以排序的。

以上情况是否给冲突可串行化的定义带来问题？请解释你的答案。

给定的情况不会对冲突可串行化的定义造成任何问题，因为每个数据项上操作的顺序对于冲突可串行化是必要的，而不同数据项上操作的顺序并不重要。

T_1	T_2
read(A)	
	read(B)
write(B)	

为了使上面的调度是冲突可串行化的，唯一的排序要求是 read(B)→write(B)。read(A)和 read(B)可以是任意顺序。因此，只要对一个数据项的操作是完全有序的，冲突可串行化的定义就应该适用于给定的多处理器系统。

习题

13.14

请解释术语串行调度和可串行化调度之间的区别。

属于一个事务的所有指令一起出现的调度称为串行调度。

可串行化调度有一个较弱的限制，即它在某种意义下等价于一个串行调度。

13.15

请考虑以下两个事务：

T_{13}: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).

T_{14}: read(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A).

令一致性需求为$A = 0 \vee B = 0$，**初值是 A=B=0.**

a.请说明包括这两个事务的每一个串行执行都保持了数据库的一致性。

b.请给出 T13 和 T14 的一次并发执行，它产生了不可串行的调度。

c.存在产生可串行化调度的 T13 和 T14 的并发执行吗？

a.

There are two possible executions: T_{13} T_{14} and T_{14} T_{13}.

Case 1:

	A	B
initially	0	0
after T_{13}	0	1
after T_{14}	0	1

Consistency met: $A = 0 \vee B = 0 \equiv T \vee F = T$

Case 2:

	A	B
initially	0	0
after T_{14}	1	0
after T_{13}	1	0

Consistency met: $A = 0 \vee B = 0 \equiv F \vee T = T$

b.

T13 和 T14 的任何并发执行都会产生不可串行的调度。

T_{13}	T_{14}
read(A)	
	read(B)
	read(A)
read(B)	
if $A = 0$ then $B = B + 1$	
	if $B = 0$ then $A = A + 1$
	write(A)
write(B)	

c.

没有并行执行导致可串行化的调度。

从 a 部分我们知道，串行执行导致 A= 0∨B = 0。假设我们从 T13 read(A)开始。那么当调度结束时，无论我们什么时候运行 T14 的步骤，B = 1。现在假设我们在 T13 完成之前开始执行 T14。那么 T14 read(B)会给 B 一个 0 的值。当 T14 结束时，A= 1。因而 B = 1∧A = 1→¬(A = 0 ∨B = 0)。

13.16

请给出具有两个事务的一个可串行化调度的示例，其中事务的提交次序与串行化的次序是不同的。

T_1	T_2
read(A)	
	read(B)
unlock(A)	
	write(B)
	read(A)
	write(A)
	commit
commit	

正如我们所看到的，上面的调度是可以用等效的串行调度 T1, T2 串行化的。在上面的调度中，T2 在 T1 之前提交。请注意，这里添加了解锁指令，以显示即使使用严格的两阶段锁定(排他锁被持有到提交后释放，防止其他事务读取这些数据，但共享锁可以以两阶段方式提前释放)，该调度也会发生。

13.17

什么是可恢复的调度？为什么要求调度的可恢复性？存在需要允许出现不可恢复调度的情况吗？请解释你的回答。

可恢复调度：对于每一对事务 Ti 和 Tj, Tj 读取之前由 Ti 写入的数据项，Ti 的提交操作出现在 Tj 的提交操作之前。

为什么要求调度的可恢复性：因为事务失败可能将使系统进入不可逆的不一致状态。

当由于时间限制，更新必须尽早可见时，有时可能需要不可恢复的调度，即使它们还没有被提交，这对于持续时间非常长的事务可能是必需的。

第十四章

14.1

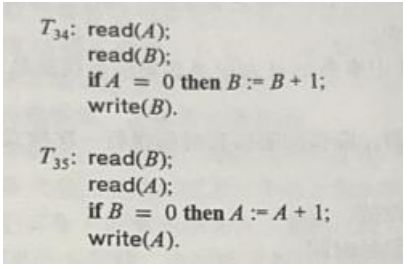
请证明两阶段封锁协议保证冲突可串行化，并且事务可以根据它们的封锁点来串行化。

假设两阶段封锁协议不能确保可序列化性，那么存在一组事务 T0，T1，…Tn−1，它遵循两阶段封锁协议并产生一个不可序列化的调度。一个不可序列化的调度意味着优先图中存

在一个循环，我们将说明两阶段封锁协议不能产生这样的循环。在不失一般性的前提下，假设优先图中存在以下循环:T0→T1→T2→⋯→Tn−1→T0。设 ai 为 Ti 获得最后一次锁定的时间(即 Ti 的锁定点)。然后对于 Ti→Tj 的所有事务，ai < aj。对于这个循环我们有 $\alpha_0 < \alpha_1 < \alpha_2 < \ldots < \alpha_{n-1} < \alpha_0$ ，因为 a0 < a0 是一个矛盾，所以不可能存在这样的循环。因此两阶段封锁协议不能产生不可序列化的调度。由于对于 Ti→Tj, ai < aj 的所有事务，事务的锁点排序也是优先图的拓扑排序。因此，可以根据它们的锁定点对事务进行序列化。

14.2

请考虑下面两个事务：



请给事务 T34 与 T35 增加封锁和解锁指令，使它们遵从两阶段封锁协议。这两个事务的执行会导致死锁吗?

T_{34}:		T_{35}:	
	lock-S(A)		**lock-S**(B)
	read(A)		**read**(B)
	lock-X(B)		**lock-X**(A)
	read(B)		**read**(A)
	if $A = 0$		**if** $B = 0$
	then $B := B + 1$		**then** $A := A + 1$
	write(B)		**write**(A)
	unlock(A)		**unlock**(B)
	unlock(B)		**unlock**(A)

这些事务的执行可能导致死锁。例如,
考虑以下部分时间表:

T_{31}	T_{32}
lock-S (A)	
	lock-S (B)
	read(B)
read(A)	
lock-X (B)	
	lock-X (A)

事务现在陷入了死锁。

14.3

强两阶段封锁带来了什么好处?它与其他形式的两阶段封锁相比有何异同?

强两阶段封锁具有严格两阶段封锁的优点。此外，对于两个冲突的事务，它们的提交顺序是它们的可序列化顺序。在一些系统中，用户可能期望这种行为。

14.14

请解释为什么 undo-list 中事务的日志记录必须由后往前处理，而执行重做时日志记录则由前往后处理。

在 undo-list 中的单个事务中，假设一个数据项更新了不止一次，比如从 1 更新到 2，然后从 2 更新到 3。如果按正序处理 undo 日志记录，则数据项的最终值将被错误地设置为 2，而如果按反序处理，则数据项的最终值将被设置为 1。同样的逻辑也适用于 undo-list 上由多个事务更新的数据项。

使用与上面相同的例子，但是假设事务已经提交，很容易看出，如果重做处理按正向顺序处理记录，最终值将正确地设置为 3，但如果按反向顺序处理，最终值将错误地设置为 2。

14.15

请解释检查点机制的目的。应该间隔多长时间执行一次检查点?执行检查点的频率对以下各项有何影响?

无故障发生时的系统性能。

从系统崩溃中恢复所需的时间。

从介质(磁盘)故障中恢复所需的时间。

检查点使用基于日志的恢复方案来完成，以减少崩溃后恢复所需的时间。如果没有检查点，则必须在崩溃后搜索整个日志，并从日志中撤消/重做所有事务。如果已经执行了检查点，那么在恢复时可以忽略检查点之前的大多数日志记录。

执行检查点的另一个原因是在稳定存储空间满时从它中清除日志记录

由于检查点在执行时会导致一些性能损失，如果快速恢复不是关键，那么应该降低它们的频率。如果我们需要快速恢复，检查点频率应该增加。如果可用的稳定存储较少，频繁的检查点是不可避免的。检查点对从磁盘崩溃中恢复没有影响;归档转储相当于从磁盘崩溃中恢复的检查点。

14.21

大部分数据库系统实现都采用严格的两阶段封锁。请说明这种协议流行的三点理由

① 实现起来相对简单，

② 由于无级联调度，因此产生了较低的回滚开销，

③ 通常允许可接受的并发级别。

14.24

如果通过死锁避免机制避免了死锁后，那么仍有可能饿死吗?请解释你的答案。

事务可能多次成为防止死锁回滚的受害者，从而造成潜在的饥饿情况。

14.34

如果与某块有关的某些日志记录没有在该块输出到磁盘之前先被输出到稳定存储器中，请解释数据库可能会变得怎样不一致。

考虑一个银行方案和一个将 50 美元从账户 a 转到账户 b 的交易。该交易有以下步骤:

a. **read**(A,a_1)

b. $a_1 := a_1 - 50$

c. **write**(A,a_1)

d. **read**(B,b_1)

e. $b_1 := b_1 + 50$

f. **write**(B,b_1)

假设在事务提交之后，但在其日志记录被刷新到稳定存储之前，系统崩溃。进一步假设在崩溃时，仅在第三步中对 A 的更新实际上已经传播到磁盘，而包含 B 的缓冲页还没有写入磁盘。当系统启动时，它处于不一致的状态，但是无法恢复，因为稳定存储中没有与此事务对应的日志记录。

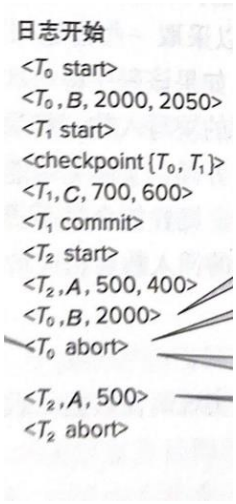
14.36

假设使用了两阶段封锁，但排他锁是提前释放的，也就是说，封锁不是以严格的两阶段方式实现的。举例说明当使用基于日志的恢复算法时，事务回滚为什么会导致错误的最终状态。

如果并发控制机制允许被一个事务 T1 修改过的数据项 X 在 T1 提交前进而被另一个事务 T2 修改，那么通过将 X 重置为它的旧值（T1 更新之前的值）来撤销 T1 的影响的同时也会撤销 T2 的影响。

14.37

考虑图 14-5 中的日志。假设恰好在<T0，abort>日志记录被写出之前系统崩溃。请解释在系统恢复时会发生什么。



恢复过程如下:

Redo :

a. Undo-List = T_0, T_1

b. Start from the checkpoint entry and perform the redo operation.

c. C = 600

d. T_1 is removed from the Undo-list as there is a commit record.

e. T_2 is added to the Undo list on encountering the $< T_2$ start $>$ record.

f. A = 400

g. B = 2000

Undo:

a. Undo-List = T_0, T_2

b. Scan the log backwards from the end.

c. A = 500; output the redo-only record $< T_2$, A, 500 $>$

d. output $< T_2$ abort $>$

e. B = 2000; output the redo-only record $< T_0$, B, 2000 $>$

f. output $< T_0$ abort $>$

在恢复过程结束时，系统的状态如下:

A = 500
B = 2000
C = 600

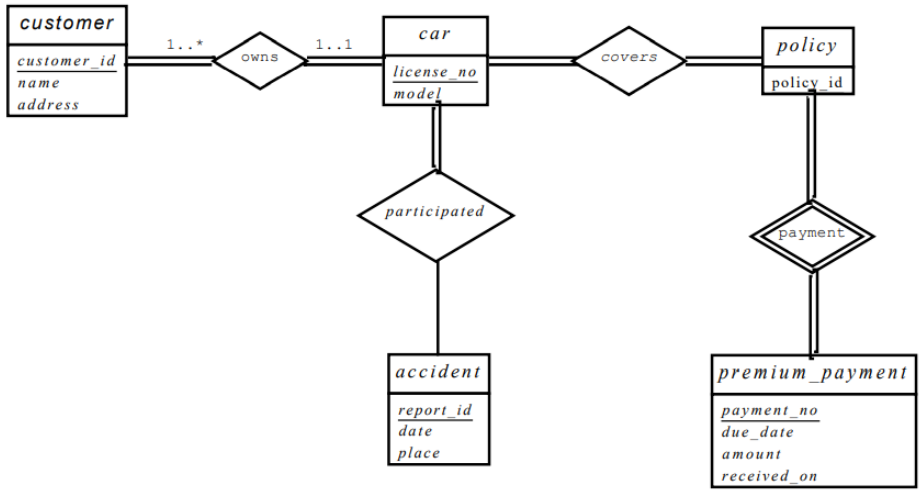
恢复过程中添加的日志记录如下:

$< T_2$, A, 500 $>$
$< T_2$ abort $>$
$< T_0$, B, 2000 $>$
$< T_0$ abort $>$

第七章

7.1

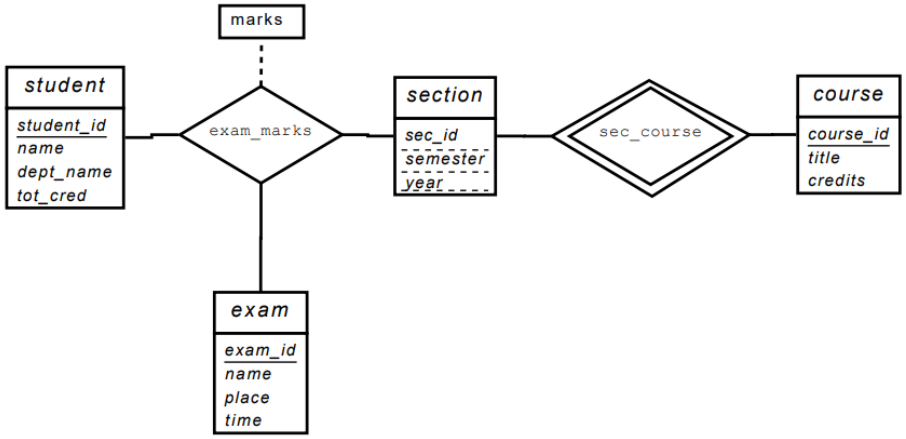
为车辆保险公司构建一张 E-R 图，其每位客户拥有一辆或多辆车。每辆车关联零次或任意多次事故的记录。每张保险单为一辆或多辆车投保，并与一次或多次保费支付相关联。每次支付只针对一段特定的时间，具有关联的到期日和缴费日。



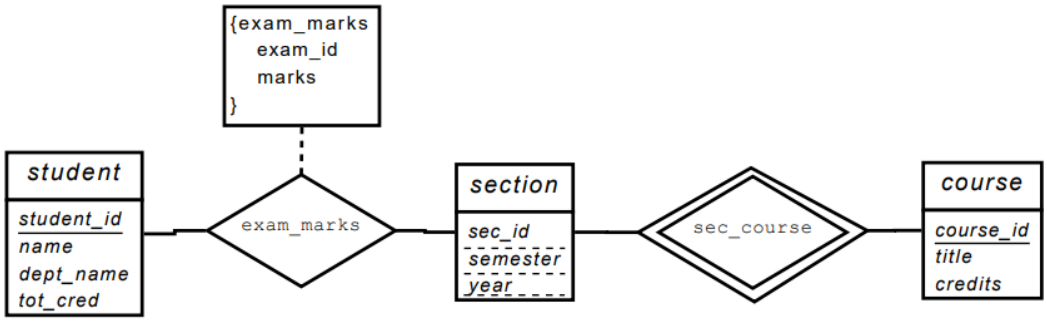
7.2

考虑一个数据库，它包含来自大学模式的 **student**、**course** 和 **section** 实体集，并且另外还记录了学生在不同课程段的不同考试中所获得的分数。

a. 构建一张 **E-R** 图，将考试建模为实体，并使用三元联系作为设计的一部分。

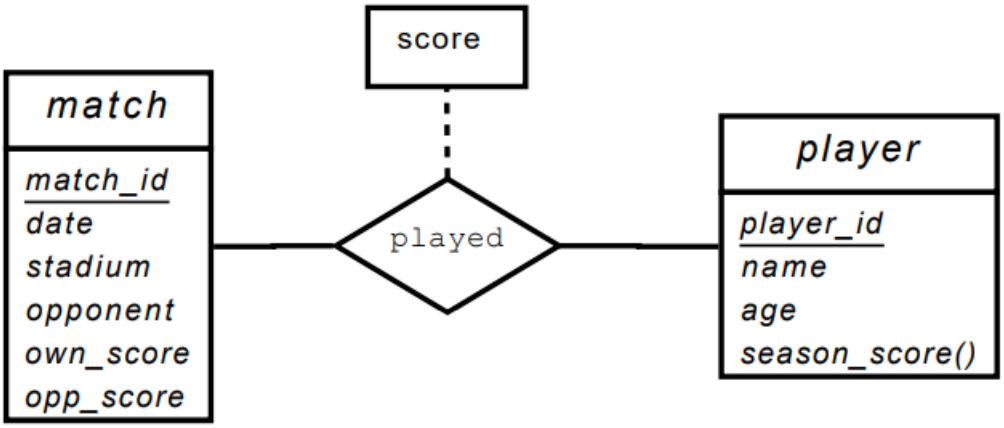


b. 构造一张可替代的 **E-R** 图，它只使用 **student** 和 **section** 之间的二元联系。保证在特定的 **student** 和 **section** 对之间只存在一个联系，而且可以表示出学生在不同考试中所得的分数。



7.3

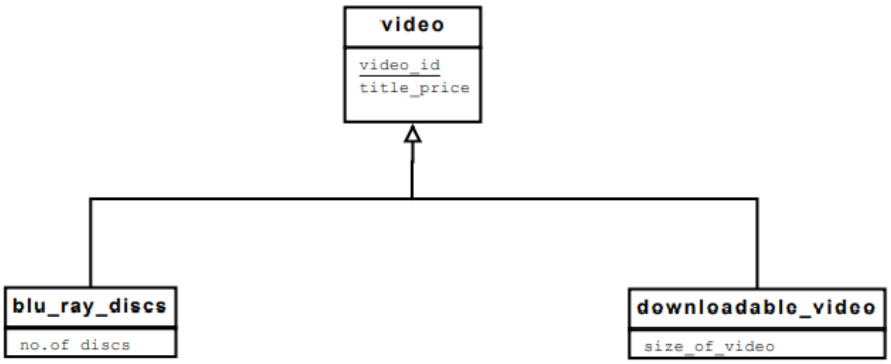
设计一张 **E-R** 图用于跟踪记录你喜欢的球队的得分统计数据。你应该保存打过的比赛、吗，每场比赛的比分、每场比赛的上场队员以及每个队员在每场比赛中的得分统计。总的统计数据应该被建模成派生属性，并附上解释以说明如何计算它们。



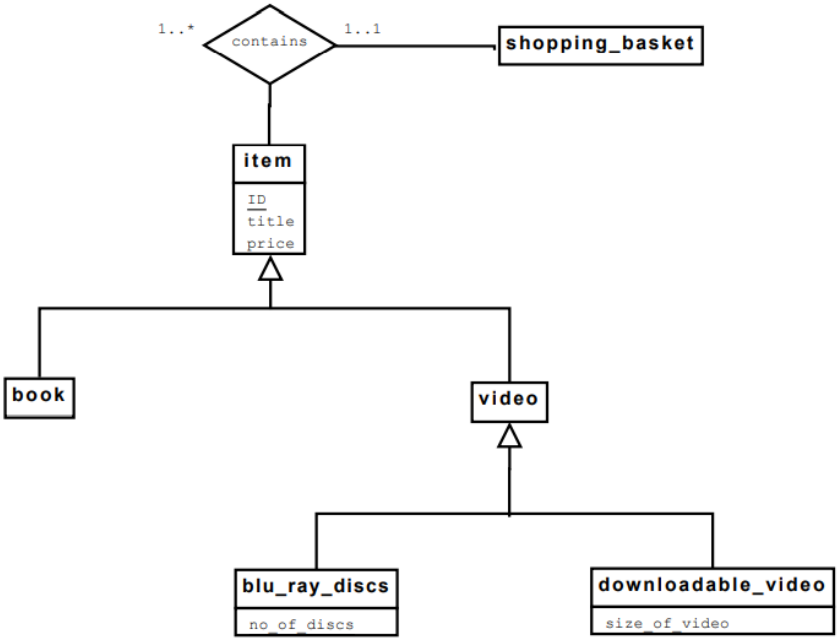
7.21

考虑图 7-30 中的 E-R 图，它对一家网上书店进行建模。

a.假设书店增加了蓝光光盘和可下载视频。相同的商品可能以一种格式或两种格式存在，对于不同格式具有不同的价格。绘制 E-R 图的一部分为这个新增需求建模，仅显示与视频相关的部分。



b.现在对完整的 E-R 图进行扩展，从而对包含书、蓝光光盘或可下载视频的任意组合的购物篮的情况进行建模。



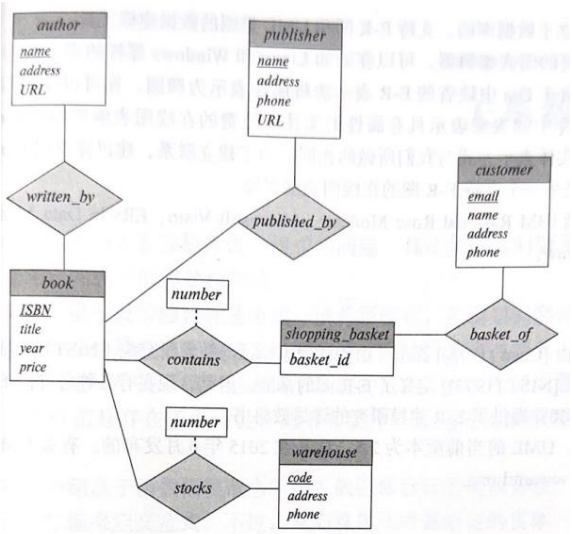


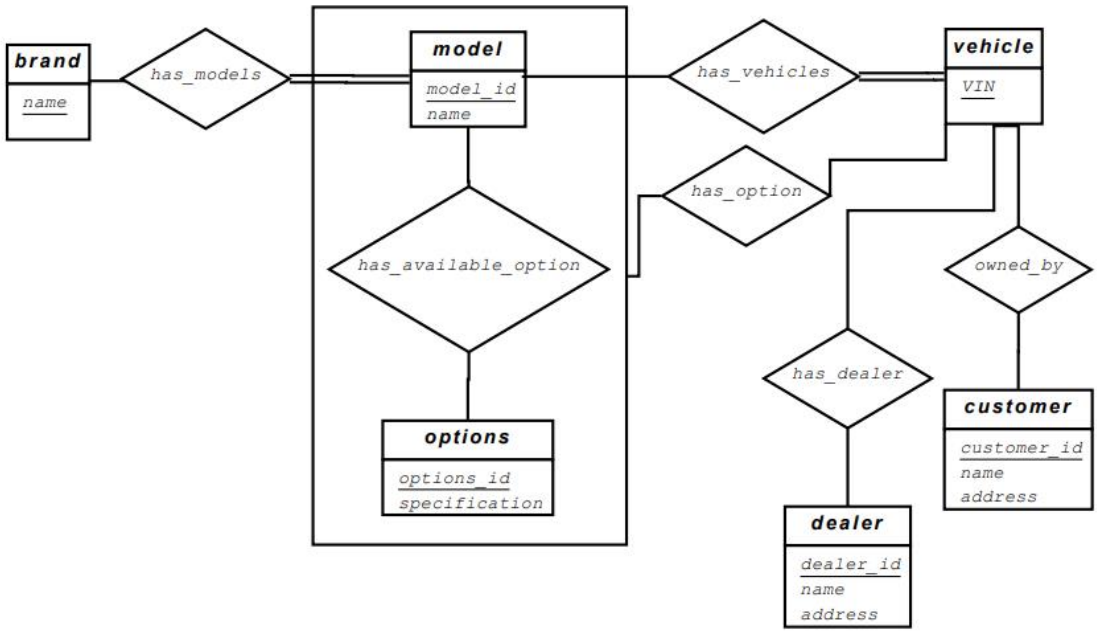
图 7-30　用于在线书店建模的 E-R 图

7.22

为一家汽车公司设计一个数据库，用于给它的经销商提供协助以维护客户记录和经销商库存,并协助销售人员订购车辆。

每辆车由车辆标识号（Vehicle Identification Number. VIN)来标识,每辆单独的车都是公司提供的特定品牌的特定模型(例如,XF 是塔塔汽车捷豹品牌车的模型)。每个模型都可以有各种选项,但是一辆车可能只有一些(或没有)可用的选项。数据库需要保存关于模型、品牌、选项的信息,以及每个经销商、客户和车的信息。

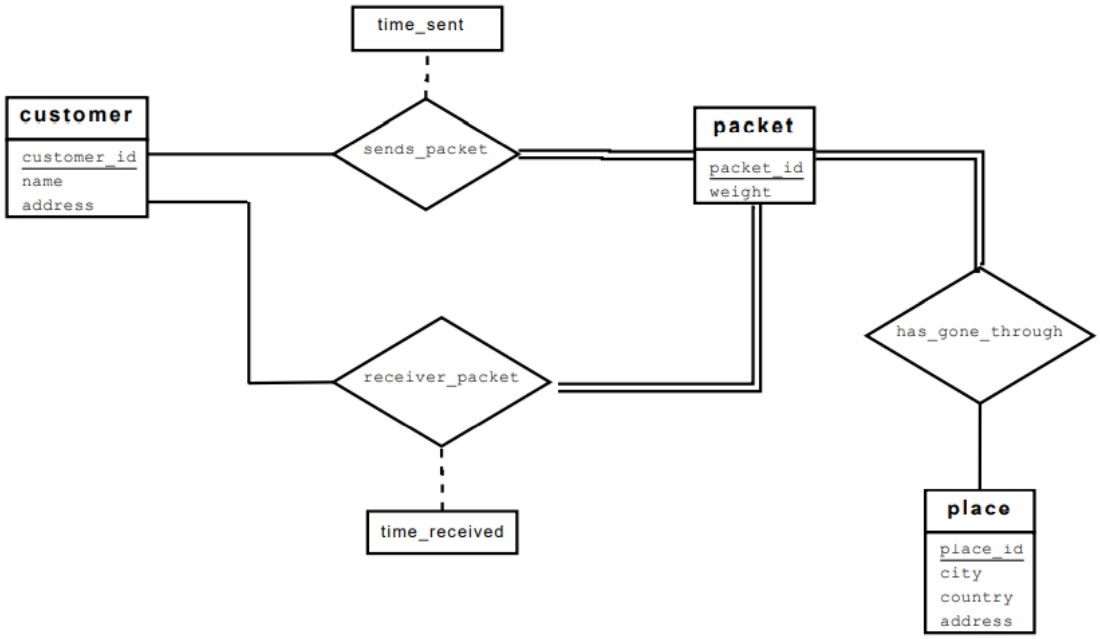
你的设计应该包括 E-R 图、关系模式的集合，以及包括主码和外码约束在内的一系列约束。



7.23

为全球包裹递送公司(例如 DHL 或者 FedEX)设计一个数据库。数据库必须能够追踪寄件客户和收件客户;有些客户可能两者都是。每个包裹必须是可标识且可追踪的，因此数据库必须能够存储包裹的位置以及它的历史位置。位置包括卡车、飞机、机场和仓库。

你的设计应该包括 E-R 图、关系模式的集合,以及包括主码和外码约束在内的一系列约束。


time_sent
customer
customer_id
name
address
sends_packet
packet
packet_id
weight
has_gone_through
receiver_packet
time_received
place
place_id
city
country
address