Homework #4

1. 假设磁盘块大小为 8 KB，块中存储 200 字节的定长记录，块首部只包括一个 8 字节的模式指针和一个偏移量表。对于插入块内的每条记录，在偏移量表中都增加一个 2 字节的指针指向该记录。假设每天向块内插入 4 条记录（空间不足时允许插入部分记录后结束全部操作），删除 2 条记录。假设每天的删除记录操作总是发生在插入记录之前，删除记录使用一个“删除标记”代替记录在偏移量表中的指针。给定一个磁盘块，如果刚开始块是空的，则几天后不能再向该块内插入记录？此时，该块内一共有多少条记录？

第一天,插入 4 条记录: 800 + 8 = 808 B;
以后每天都增加: - 400 + 808 = 408 B;
808 + 408 x <= 8192 – 8
⇨ x <= 18.1
因 x 为整数所以 x=18.当 x=18 时,即第 19 天插入记录后,占据的磁盘空间为: 808+408*18=8152,此时共有记录数=4+18*2=40。
第 20 天时,首先删除 2 条记录,此时磁盘空间为 8152-400=7752,还可以继续插入 2 条记录,即 7752+404=8156.之后,磁盘不能再插入记录,操作结束,此时磁盘块中的记录数仍为 40.
综上,第 20 天后不可插入,磁盘块中一共有 40 条记录.

2. 假设我们采用 LRU 作为缓冲区置换策略，当我们向 Buffer Manager 发出一个读页请求时，请讨论一下：

(1) 如果页不在缓冲区中，我们需要从磁盘中读入该页。请问如何才能在缓冲区不满的时候快速地返回一个 free 的 frame？请给出至少两种策略，并分析一下各自的时间复杂度。

策略一:
将所有的空闲的 frame id 插入到一个链表中,每次从链表头部返回一个空闲 frame id,时间复杂度为 O(1).
策略二:
维护一个栈,将所有的空闲的 frame id 都压入栈中,每次从栈顶返回一个空闲 frame id,时间复杂度为 O(1).

(2) 如何才能快速地判断所请求的页是否在缓冲区中？如果请求的页在缓冲区中，如何快速返回该页对应的 frame 地址？ 请给出至少两种策略，并分析一下各自的时间复杂度。

策略一:
将所有的映射关系(page id => frame id)以键值对形式保存在一个 hash 表中,可以在 O(1)的时间复杂度找到对应的键值对,从而找到 frame id.
策略二:
将所有的映射关系(page id => frame id)以键值对形式保存在一个树形数据结构中(例如红黑树),可以在 O(logn)的时间复杂度找到对应的键值对,从而找到 frame id.

Test #1

1. 已知有关系模式 R(A,B,C,D,E)，R 上的一个函数依赖集为：F={AD→CE, C→BE, A→D, AB→C, AC→DE}

(1) 求 F 最小函数依赖集以及 R 的所有候选码；

F 最小函数依赖集：
将右边写出单属性并去除重复 FD:
F={AD→C, AD→E, C→B, C→E, A→D, AB→C, AC→D, AC→E}
消去左部冗余属性:
由 C→E 可得 AC→E 是冗余属性;
由 A→D 可得 AC→D 是冗余属性;
由 A→D, AD→C 可推出 A→C,因此可去除 AD→C 中的 D;
由 A→D, AD→E 可推出 A→E,因此可去除 AD→E 中的 D;
由 A→C, AC→D 可推出 A→D,因此可去除 AC→D 中的 C;
由 A→C, AC→E 可推出 A→E,因此可去除 AC→E 中的 C;
由 A→C 可得 AB→C 是冗余属性;
F={A→C, A→E, C→B, C→E, A→D}
消去冗余函数依赖:
由 A→C, C→E 可推出 A→E,所以 A→E 是冗余属性;
F={ A→C, C→B, C→E, A→D}

R 的所有候选码:
由 A→C, C→B 可推出 A→B;
所以 A→{A,B,C,D,E},候选码是 A.

(2) R 属于第几范式？为什么？

答: R 属于 2NF,因为 R 的每一个非主属性都完全函数依赖于 A; R 不属于 3NF,因为 B 传递依赖于 A.

(3) 请将 R 无损连接并且保持函数依赖地分解到 3NF。

按左部分组: q={R1(ACD), R2(BCE)};
R 的主码是 A,所以 p=q∪{R(X)}={R1(ACD), R2(BCE), R(A)};
因为{A}是 R1 的子集,所以从 p 中去掉 R(A);
p={R1(ACD), R2(BCE)}.

2. 单关系上的聚簇索引可以使记录在磁盘块中有序存储，从而可以加快聚簇字段上的查询。请问：

(1) 单个关系上能否设计多个聚簇索引?为什么?

答: 不能.因为聚簇索引利用了记录在物理存储上的排列顺序,索引的顺序与记录存储的顺序必须保持一致,所以只能有一个聚簇索引.

(2) 在实际数据库设计中，什么样的字段不适合设计聚簇索引?请解释理由。

答: 频繁插入/删除/更新的字段不适合设计聚簇索引,因为在聚簇索引上执行插入/删除/更新需要移动多个记录的物理位置.

3. 假设关系数据库中有三个只包含定长记录的表：C（公司表）：记录长度为 6000 字节，共有 20000 记录。主键为 C.cid，长度为 20 字节。E（负责人表）：记录长度为 2000 字节，共有 40000 记录。其中 E.cid 为指向 C.cid 的外键，长度为 20 字节，表示该负责人所在的公司，主键为 E.eid，长度 20 字节，表示该产品所关联的负责人 ID。S（产品表）：记录长度为 1000 字节，共有 120000 记录。其中 S.eid 是指向 E.eid 的外键，长度 20 字节，表示该产品所关联的负责人 ID。假设：1 个公司固定有 2 个负责人，1 个负责人固定负责 3 个产品；磁盘块大小为 32KB，块首部大小为 50 字节；记录在磁盘块中不允许跨块存储。现在我们考虑两种存储策略：1) S1：C 表连续存储在磁盘块集合 B1 上，E 表连续存储在磁盘块集合 B2 上，S 表连续存储在磁盘块集合 B3 上。B1、B2、B3 不相交。2) S2：将每条公司记录、该公司关联的 2 条负责人记录、这 2 个负责人所关联的 6 条产品记录一起存储在同一磁盘块上。请计算 S1 和 S2 两种策略各自需要多少磁盘块才能存储上述三个表的数据？

S1:
由于记录在磁盘块中不允许跨块存储,每个块中能够存储表 C 中的记录条数为:

$$\lfloor (32 * 1024 - 50)/6000 \rfloor = 5$$

所以,存储表 C 一共需要的块数为 20000 / 5 = 4000;
同理,每个块中能够存储表 E 中的记录条数为:

$$\lfloor (32 * 1024 - 50)/2000 \rfloor = 16$$

所以,存储表 E 一共需要的块数为 40000 / 16 = 2500;
每个块中能够存储表 S 中的记录条数为:

$$\lfloor (32 * 1024 - 50)/1000 \rfloor = 32$$

所以,存储表 S 一共需要的块数为 120000 / 32 = 3750;
S1 需要的磁盘块数为 4000 + 2500 + 3750 = 10250.
S2:
一个聚簇需要的空间代价为 6000 + 2 * (2000 - 20) + 6 * (1000 - 20) = 15840 字节;
所以,每个块中能够存储的聚簇数目为:

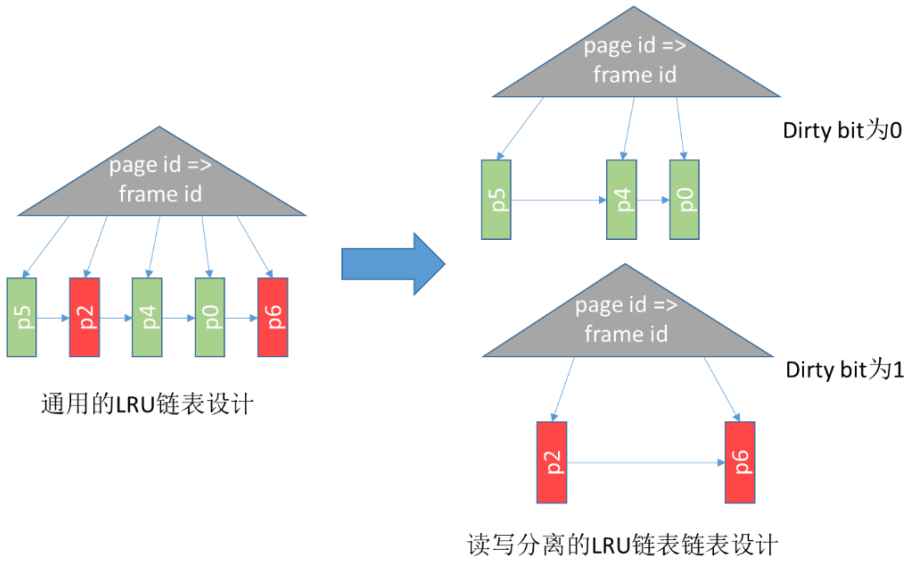
$$\lfloor (32 * 1024 - 50)/15840 \rfloor = 2$$

S2 需要的磁盘块数为 20000 / 2 = 10000.

4. LRU 算法只考虑了最近一次访问时间，但如果被置换的 frame 是 dirty 页，换出时会带来额外的写 I/O。现在很多 DBMS 采用 SSD 作为存储介质，它具有写慢读快的特点，因此我们希望在 LRU 的基础上优先置换 clean 的 frame，从而减少对 SSD 的写操作。请给出一种基于上述思路的 SSD 写友好的 LRU 策略，简要解释一下总体思路，给出数据结构图示，并给出置换算法过程。

思路:维护两个 LRU 链表,其中一个存储 clean 页,另一个存储 dirty 页.

图示:



通用的LRU链表设计

读写分离的LRU链表链表设计

置换算法:

(1) 如果读 LRU 链表(存储 clean 页)不为空,则从该链表末尾弹出一个页 p,转(3).

(2) 从写 LRU 链表(存储 dirty 页)的末尾弹出一个页 p.

(3) 返回 p 的页号.

Homework #5

1. 假设我们在数据库中设计了如下基本表来存储文献：paper(id: int, title: varchar(200), abstract: varchar(1000))。最常见的文献查询可以描述为“查询 title 中同时包含给定关键词的文献”，关键词可以是一个，也可以是多个。请回答下面问题（假设所有文献都是英文文献）：

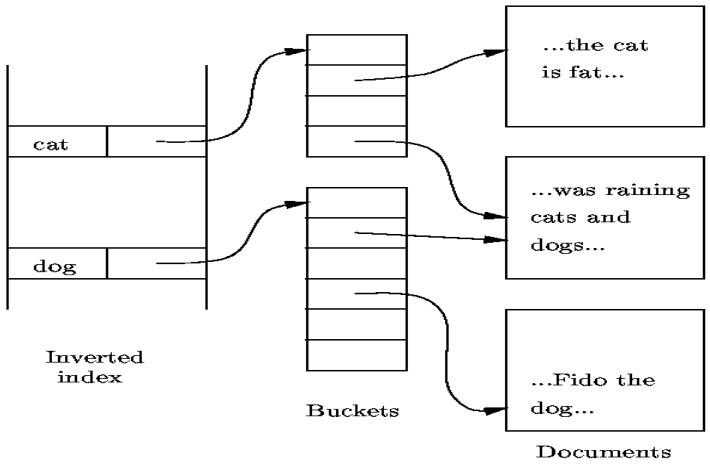
1) 假如在 title 上创建了 B+-tree 索引，能不能提高此查询的效率（须解释理由）？

答: 不能.文献查询并不是精确(whole-key)点查询,而是范围(partial-key)查询,查询的单词可能出现在 title 的任意位置.由于 B+树是根据键的字典序排序的,仅支持前缀查询,不支持中缀和后缀范围查询,因此使用 B+树结构仍要扫描所有数据项.

2) 由于文献 title 的关键词中存在很多重复词语，因此上述文献查询可以借鉴我们课上讲述的支持重复键值的辅助索引技术来进一步优化。请基于此思想画出一种优化的索引结构，简要说明该索引上的记录插入过程以及文献查询过程。

答: 使用倒排索引.倒排索引根据单词索引到 title 包含该单词的所有文档.相比于 B+树结构,采用倒排索引优点如下: (a)避免重复单词的存储; (b)对于文献查询不需要扫描所有数据项,仅通过少数几次索引查询即可完成搜索.

索引结构(PPT adb07 p27):

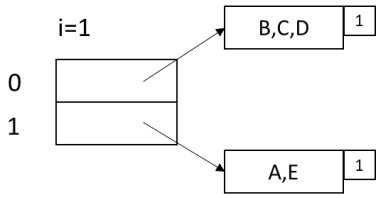


记录插入过程:
对 title 进行分词,提取出所有的关键词.对于每一个关键词,在倒排索引中执行查找.如果该关键词不存在于索引中,则在倒排索引中插入该关键词,并分配一个新的间接桶,关键词对应的指针指向该间接桶.最后,在间接桶中加入指向该文献的指针.如果该关键词存在于索引中,则依据索引找到关键词对应的间接桶,并在间接桶中加入指向该文献的指针.

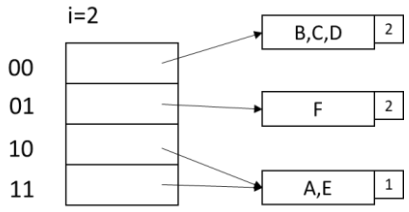
文献查询过程:
对于给定的关键词,在倒排索引中搜索该关键词,得到该关键词对应的间接桶,桶中所指向的文献均是包含该关键词的文献.如果存在多个关键词,则重复上述过程,并求出所有桶中文献的交集.

2. 假设有如下的键值，现用 5 位二进制序列来表示每个键值的 hash 值。回答问题：
A [11001] B [00111] C [00101] D [00110] E [10100] F [01000] G [00011]
H [11110] I [10001] J [01101] K [10101] L [11100] M [01100] N [11111]
1) 如果将上述键值按 A 到 N 的顺序插入到可扩展散列索引中，若每个桶大小为一个磁盘块，每个磁盘块最多可容纳 3 个键值，且初始时散列索引为空，则全部键值插入完成后该散列索引中共有几个桶？并请写出键值 E 所在的桶中的全部键值。

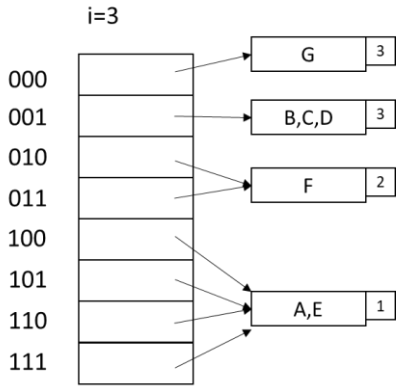
答:
(1) i=1,插入 A-E:



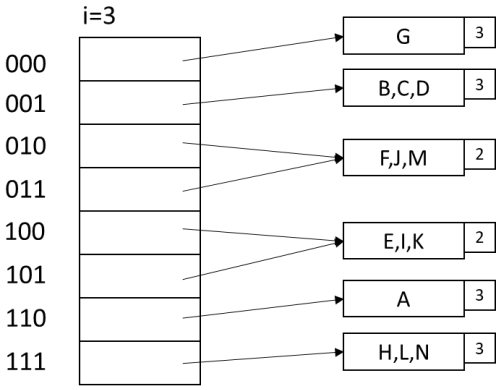
(2) 插入 F,触发分裂, i=2:



(3) 插入 G,触发分裂, i=3:



(4) 插入 H-N,触发两次分裂,共有 6 个桶:

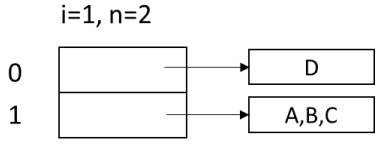


(5) 全部键值插入完成后该散列索引中共有 6 个桶;键值 E 所在的桶中的全部键值为 E, I, K.

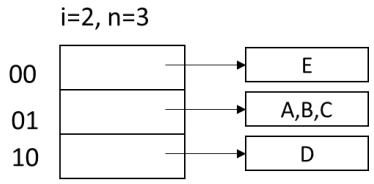
2) 前一问题中，如果换成线性散列索引，其余假设不变，同时假设只有当插入新键值后空间利用率大于 80%时才增加新的桶，则全部键值按序插入完成后该散列索引中共有几个桶？并请写出键值 B 所在的桶中的全部键值（包括溢出块中的键值）。

答:

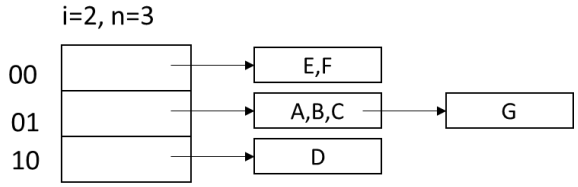
(1) i=1, n=2,插入 A-D:



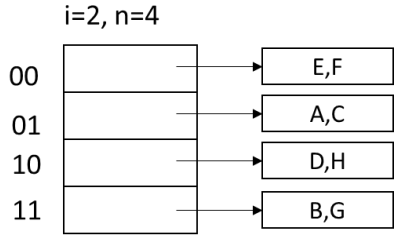
(2) 插入 E, 5/6 > 0.8,增加新桶, i=2, n=3:



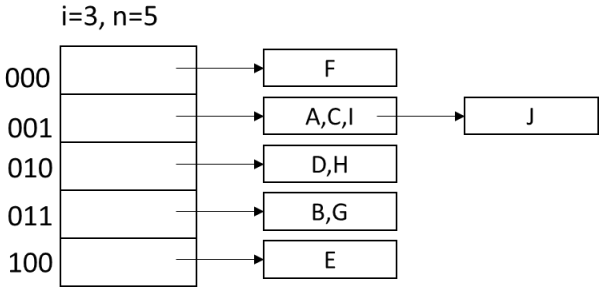
(3) 插入 F, G, 7/9 <= 0.8,不增加新桶,使用溢出块:



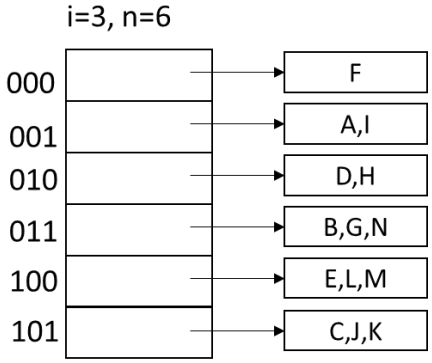
(4) 插入 H, 8/9 > 0.8,增加新桶, i=2, n=4:



(5) 插入 I, J, 10/12 > 0.8,增加新桶, i=3, n=5:



(6) 插入 K, L, M, 13/15 > 0.8,增加新桶, i=3, n=6;然后插入 N:



(7) 全部键值按序插入完成后该散列索引中共有 6 个桶;键值 B 所在的桶中的全部键值为 B, G, N.

3. 对于 B+树，假设有以下的参数：

参数	含义	参数	含义
N	记录数	S	读取一个磁盘块时的寻道时间
n	B+树的阶，即节点能容纳的键数	T	读取一个磁盘块时的传输时间
R	读取一个磁盘块时的旋转延迟	m	在内存的 m 条记录中查找 1 条记录的时间（线性查找）

假设所有磁盘块都不在内存中。现在我们考虑一种压缩 B+树，即对 B+树的节点键值进行压缩存储。假设每个节点中的键值压缩 1 倍，即每个节点在满的情况下可压缩存储 2n 个压缩前的键值和 2n+1 个指针。额外代价是记录读入内存后必须解压，设每个压缩键值的内存解压时间为 c。给定 N 条记录，现使用压缩 B+树进行索引，请问在一棵满的 n 阶压缩 B+树中查找给定记录地址的时间是多少？（使用表格中的参数表示，n+1 或 n-1 可近似表示为 n）？

答:

对于未压缩的 B+树,查找时间为:树高*(读取一个磁盘块时间+内存查找时间),其中树高为 $log_n N$.

对于压缩的 B+树,树高为$log_{2n} N$,读取一个磁盘块的时间为 $R+S+T$, 内存查找时间包括解压时间与线性查找时间,为 $2cn+2n$.

所以,总的查找时间为 $log_{2n} N * (R + S + T + 2cn + 2n)$.

Homework #6

1. 已知有关系模式 R(A, B, C)和 S(B, C, D)，每个属性都占 10 个字节，请估计下面的逻辑查询计划的 T(U), S(U)以及结果关系中每个属性的 V 值（假设满足“Containment of Value Sets”，并且选择条件中的值都在关系中存在）：

$$\mathrm{U} = \pi_{\mathrm{AD}}[(\sigma_{\mathrm{A}=3 \wedge \mathrm{B}=5}\mathrm{R}) \bowtie \mathrm{S}]$$

相应的统计量如下：

$$\mathrm{T(R)} = 100000, \quad \mathrm{V(R,A)} = 20, \quad \mathrm{V(R,B)} = 50, \quad \mathrm{V(R,C)} = 150$$
$$\mathrm{T(S)} = 5000, \quad \mathrm{V(S,B)} = 100, \quad \mathrm{V(S,C)} = 200, \quad \mathrm{V(S,D)} = 30$$

答:

令 $\mathrm{W}_1 = \sigma_{\mathrm{A}=3 \wedge \mathrm{B}=5}\mathrm{R}$,则 $\mathrm{T}(\mathrm{W}_1) = \frac{\mathrm{T(R)}}{\mathrm{V(R,A)} * \mathrm{V(R,B)}} = 100,$

$\mathrm{V}(\mathrm{W}_1, \mathrm{A}) = \mathrm{V}(\mathrm{W}_1, \mathrm{B}) = 1,\ \mathrm{V}(\mathrm{W}_1, \mathrm{C}) = 100.$ (注:V(W,C)不能大于 T(W))

令 $\mathrm{W}_2 = \mathrm{W}_1 \bowtie \mathrm{S} = (\sigma_{\mathrm{A}=3 \wedge \mathrm{B}=5}\mathrm{R}) \bowtie \mathrm{S},$

则 $\mathrm{T}(\mathrm{W}_2) = \frac{\mathrm{T}(\mathrm{W}_1) * \mathrm{T(S)}}{\max\{\mathrm{V}(\mathrm{W}_1,\mathrm{B}),\mathrm{V(S,B)}\} * \max\{\mathrm{V}(\mathrm{W}_1,\mathrm{C}),\mathrm{V(S,C)}\}} = \frac{100*5000}{100*200} = 25,$

$\mathrm{V}(\mathrm{W}_2, \mathrm{A}) = \mathrm{V}(\mathrm{W}_2, \mathrm{B}) = 1,\ \mathrm{V}(\mathrm{W}_2, \mathrm{C}) = 25,\ \mathrm{V}(\mathrm{W}_2, \mathrm{D}) = 25.$

$\mathrm{U} = \pi_{\mathrm{AD}}\mathrm{W}_2$,则 $\mathrm{T(U)} = 25,\ \mathrm{S(U)} = 20,\ \mathrm{V}(\mathrm{U}, \mathrm{A}) = 1,\ \mathrm{V}(\mathrm{U}, \mathrm{D}) = 25.$

2. 固态硬盘（Solid State Drive，SSD）是一种基于闪存的新型存储器，它与传统磁盘的主要区别之一是：传统磁盘的读写操作的速度相同，而 SSD 的读速度远快于写速度。同时，SSD 的读速度要远高于磁盘，而写速度则比磁盘慢。现在我们想将传统的两阶段多路归并排序算法移植到 SSD 上。假设 SSD 上一次读块操作的时间是 t，一次写块操作的时间是 50t，磁盘上的读/写块时间是 30t。对于给定关系 R：
   - R 包含 100000 个元组，即 T(R) = 100000.
   - 一个磁盘块大小为 4000 bytes.
   - R 的元组大小为 400 bytes，即 S(R) = 400.
   - 关系 R 在磁盘上非连续存放
   - 排序字段的大小为 32 bytes.
   - 记录指针的大小为 8 bytes.

现在我们考虑下面一种改进的归并排序算法。原来的两阶段归并排序的第一阶段是将排序后的整个元组写到 chunk 中，现在我们仅将排序后的<sortingKey, recordPointer>写出。第一阶段，我们在内存中将记录按<sortingKey, recordPointer>排序，当<sortingKey, recordPointer>记录填满内存时将其写到 chunk 中。第二阶段，读入各个 chunk 中的<sortingKey, recordPointer>并在内存中归并。通过记录指针(recordPointer)我们可以读取记录的其它部分(从 R 的磁盘块中)，并将排好序的记录写回到外存。请回答：

1) 如果 R 存储在磁盘上，这一改进排序算法的 I/O 代价（用 t 的表达式表示，包括最后写出到排序文件中的代价）是多少？并解释该算法性能是否能优于原来的排序算法。

答:
R 存储在磁盘上,且 R 在磁盘上非连续存放,第一阶段,将元组全部读入内存的时间为:
100000 × 30t = 3000000t
一个磁盘块可以容纳<sortingKey, recordPointer>的数量为: 4000 ÷ (32 + 8) = 100
将排序后的<sortingKey, recordPointer>写入磁盘的时间为: 100000 ÷ 100 × 30t = 30000t
第二阶段,从磁盘中将<sortingKey, recordPointer>读入到内存的时间为:
100000 ÷ 100 × 30t = 30000t
通过 recordPointer 读取记录的其它部分到内存的时间为: 100000 × 30t = 3000000t
所有元组排序后写回到磁盘的时间为: 100000 ÷ (4000 ÷ 400) × 30t = 300000t
综上所述,改进后的排序算法的总时间代价为:
3000000t + 30000t + 30000t + 3000000t + 300000t = 6360000t
原来的排序算法的总时间代价为: 100000 × 30t + 100000 ÷ (4000 ÷ 400) × 30t × 3 = 3900000t
由于 6360000t > 3900000t,故改进的排序算法没有优于原来的排序算法.

2) 如果 R 存储在 SSD 上，这一改进排序算法的 I/O 代价（用 t 的表达式表示，包括最后写出到排序文件中的代价）是多少？并解释该算法性能是否能优于原来的排序算法。

答:
R 存储在 SSD 上,第一阶段,将元组全部读入内存的时间为: 100000 × t = 100000t
将排序后的<sortingKey, recordPointer>写入磁盘的时间为: 100000 ÷ 100 × 50t = 50000t
第二阶段,从磁盘中将<sortingKey, recordPointer>读入到内存的时间为:
100000 ÷ 100 × t = 1000t
通过 recordPointer 读取记录的其它部分到内存的时间为: 100000 × t = 100000t
所有元组排序后写回到磁盘的时间为: 100000 ÷ (4000 ÷ 400) × 50t = 500000t

综上所述,改进后的排序算法的总时间代价为:
100000t + 50000t + 1000t + 100000t + 500000t = 751000t
原来的排序算法的总时间代价为: 100000 × t + 100000 ÷ (4000 ÷ 400) × (t + 50t × 2) = 1110000t
由于 751000t < 1110000t,故改进的排序算法优于原来的排序算法.

3. 我们在课本上讨论的归并排序算法是一个两趟算法。设两个连接关系为 R1 和 R2，在基于两趟归并排序的排序连接算法中，我们要求内存 M 必须满足条件 $M \geq \max\{\sqrt{R1}, \sqrt{R2}\}$。现在我们考查关系 R 的两趟归并排序算法，我们发现当内存 M 不满足条件 $M \geq \sqrt{R}$ 时，我们仍可以采用一种多趟算法来完成归并排序操作。请用自然语言或伪码给出这一多趟归并连接算法的简要描述和步骤，并给出当 B(R1) = 10000，B(R2) = 5000，M = 20 时该算法的 I/O 代价，这里我们假设 R1 和 R2 都是连续存放的。

多趟归并连接算法:
(1) 将 R1 和 R2 分别划分为大小为 M 个磁盘块的段,每次将一个段读入内存,排序并写回磁盘,此时共有 $\left\lceil \frac{B(Rn)}{M} \right\rceil$ 个有序的段(n 为 1 或 2).
(2) 循环调用 M 路归并排序算法,即每次将 M 个有序的段归并为一个有序的段.则循环调用 k 次后,共有 $\left\lceil \frac{B(Rn)}{M^{k+1}} \right\rceil$ 个有序的段.当只剩余一个有序的段时,循环结束，即 $\left\lceil \frac{B(Rn)}{M^{k+1}} \right\rceil = 1$.
(3) 按顺序读取 R1 和 R2,并执行连接操作.

I/O 代价:
对于 R1, k = 3,即需要执行 3 次 M 路归并;对于 R2, k = 2,需要执行 2 次 M 路归并.总 I/O 代价为: 10000 × (4 × 2 + 1) + 5000 × (3 × 2 + 1) = 125000
(注: 最后一次归并排序可以与连接合并进行,此时需要的内存大小为 2 + 13 = 15 < M, I/O 代价为: 10000 × (3 × 2 + 1) + 5000 × (2 × 2 + 1) = 95000)

Homework #7

1. 目前许多 DBMS 例如 MySQL 都默认不支持嵌套事务（即在一个事务内部又开始另一个事务），请分析一下：如果 DBMS 支持嵌套事务，将面临哪些问题（至少写出点以上并且要给出自己的分析）？

答: (1)如果支持嵌套事务,则会出现外层事务还没 commit 时内层事务已经 commit 的情况,此时若外层事务选择 abort,如果选择同时 abort 内层事务,则破坏了内层事务的持久性;如果选择不 abort 内层事务,则破坏了外层事务的原子性. (2)如果采用嵌套事务,则现有的事务日志方式将无法工作,因为它无法区分哪些事务是外层事务哪些是内层事务.

2. 下面是一个数据库系统开始运行后的日志记录，该数据库系统支持检查点。

1) <T1, Begin Transaction>
2) <T1, A, 10, 40>
3) <T2, Begin Transaction>
4) <T1, B, 20, 60>
5) <T1, A, 40, 75>
6) <T2, C, 30, 50>
7) <T2, D, 40, 80>
8) <T1, Commit Transaction>
9) <T3, Begin Transaction>
10) <T3, E, 50, 90>
---------- ①
11) <T2, D, 80, 65>
12) <T2, C, 50, 75>
13) <T2, Commit Transaction>
---------- ②
14) <T3, Commit Transaction>
15) **<CHECKPOINT>**
16) <T4, Begin Transaction>
17) <T4, F, 60, 120>
18) <T4, G, 70, 140>
---------- ③
19) <T4, F, 120, 240>
20) <T4, Commit Transaction>

设日志修改记录的格式为<Tid, Variable, Old value, New value>，请给出对于题中所示①、②、③三种故障情形下，数据库系统恢复的过程以及数据元素 A, B, C, D, E, F 和 G 在执行了恢复过程后的值。

答:

① UNDO = {T2, T3}, REDO = {T1};
先 UNDO: E = 50, D = 40, C = 30;
再 REDO: A = 40, B = 60, A = 75;
<abort, T2>, <abort, T3>;
最终 A = 75, B = 60, C = 30, D = 40, E = 50, F = 60, G = 70.

② UNDO = {T3}, REDO = {T1, T2};
先 UNDO: E = 50;
再 REDO: A = 40, B = 60, A = 75, C = 50, D = 80, D = 65, C = 75;
<abort, T3>;
最终 A = 75, B = 60, C = 75, D = 65, E = 50, F = 60, G = 70.

③ UNDO = {T4}, REDO = {};
UNDO: G = 70, F = 60;
<abort, T4>;
最终 A = 75, B = 60, C = 75, D = 65, E = 90, F = 60, G = 70.

3. 采用了两阶段锁协议的事务是否一定不会出现脏读问题？如果不会，请解释理由；如果会，请给出一个例子。

答: 会出现脏读问题.
示例:

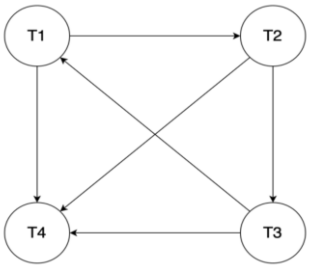
t	T1	T2
1	xL1(A)	
2	A=A-100	
3	xL1(B)	
4	Write(A)	
5	xL1(B)	
6	Unlock(A)	
7		sL2(A)
8	B=B+100	Read(A)
9	Write(B)	
10	Unlock(B)	
11	Rollback	

脏读

4. 判断下面的并发调度是否冲突可串？如果是，请给出冲突等价的串行调度事务顺序；如果不是，请解释理由。

w3(D); r1(A); w2(A); r4(A); r1(C); w2(B); r3(B); r3(A); w1(D); w3(B); r4(B); r4(C); w4(C); w4(B)

答: 根据并发调度画出优先图如下:



可以看到优先图中存在环路 T1→T2→T3→T1,因此该并发调度不是冲突可串的.