2-1-1 Select Operation

(1) $\sigma_{(position='Manager') \wedge (city='London') \wedge (Staff.branchNo=Branch.branchNo)}$ (Staff X Branch)

(1) (1000 + 50) + 2*(1000 * 50) = 101 050

- Read both `Staff` and `Branch` : 1000 + 50
- Compute cartesian product and write back to disk : 1000 * 50
- Read the previous result from disk : 1000 * 50
- Cost = (1000 + 50) + 2*(1000*50) = 101050

```
for each s in Staff:
  for each b in Branch:
    check if s.position='Manager' and b.city='London' and s.branchNo = b.branchNo
```

(2) $\sigma_{(position='Manager') \wedge (city='London')}$(Staff $\bowtie_{Staff.branchNo=Branch.branchNo}$ Branch)

(2) 2*1000 + (1000 + 50) = 3 050

- Read both `Staff` and `Branch` : 1000 + 50
- Write `Staff join Branch` back to disk : 1000
- Read the previous result from disk : 1000
- Cost = (1000 + 50) + 2 * 1000 = 3050

```
for each s in Staff:
  for each b in Branch:
    if s.branchNo = b.branchNo:
      put the tuple into Temp.
// Temp = Staff Join Branch
for each s in Temp:
  check if s.position='Manager' and s.city='London'
```

$$(3)\ (\sigma_{position='Manager'}(Staff)) \bowtie_{Staff.branchNo=Branch.branchNo} (\sigma_{city='London'}(Branch))$$

$$(3)\quad 1000 + 2*50 + 5 + (50 + 5) = 1\,160$$

- Read both `Staff` and `Branch` : 1000 + 50
- Compute selection result and write back to disk : 50 + 5
- Read the previous result from disk : 50 + 5
- Cost = (1000 + 50) + 2*(50 + 5) = 1160

```
// Select from S
for each s in Staff:
  if s.position='Manager'
    put s into TempS
// Select from B
for each b in Branch:
  if b.city='London'
    put b into TempB
for each s in TempS:
  for each b in TempB:
    check if s.branchNo=b.branchNo
```

2-1-2. Join Operation

- Assume r1 is R and r2 is S.
- r1(R) has 40000 tuples
- r2(S) has 30000 tuples
- 20 tuples of r1 fit in one block
- 10 tuples of r2 fit in one block.
- There are $40000/20 = 2000$ partitions of R: $R1 \sim R2000$
- There are $30000/10 = 3000$ partitions of S: $S1 \sim S3000$

Assume S and R are contiguous in storage for part a.)

a. Nested-loops join; sorted; only 3 memory block

Assume S and R are contiguous in storage for part a.)

> Outer Relation: S x R

> Pseudo Code:

```
for each tuple s in S:
  for each tuple r in R:
    check if r.C = s.C
```

> Block Transferred:

- For each s in S, For each r in R: ns * br
- For each s in S: bs
- Total: ns * br + bs = 30000 * 2000 + 3000 = 60003000

> Block Saught:

- For each s in S, seek R for once because it's contiguous: ns
- Seek S for once: 1
- Total = ns + 1 = 30001.

b. Nested-loops join; unsorted; 102 memory blocks

> Outer Relation: S x R

> Pseudo Code:

```
for each tuple s in S:
  for each tuple r in block R:
    check if r.C = s.C
```

> Block Transferred:

- For each s in S, we still need to transfer all R: ns * br
- For each s in S: bs
- Total: ns * br + bs = 30000 * 2000 + 3000 = 60003000

> Block Saught:

- For each s in S, seek all R: ns * br
- We need to seek for each block of S: bs
- Total = ns * br + bs = 30000 * 2000 + 3000 = 60003000.

c. Block Nested-loops join; unsorted; 102 memory blocks

> Outer Relation: R x S

> Pseudo Code:

```
for each group of 100 blocks in R:
  for each block Si in S:
    for each tuple r in group of 100 blocks in R:
      for each tuple s in Si:
        check if r.c = s.c
```

> Block Transferred:

- We transfer 100 blocks of R for each block Si: (br / 100) * bs
- For each block Ri: br
- Total: (br / 100) * bs + br = 20 * 3000 + 2000 = 62000.

> Block Saught:

- We need one seek per transfer, so same as the answer above: (br / 100) * bs
- For each block Ri: br
- Total: (br / 100) * bs + br = 20 * 3000 + 2000 = 62000.

2-2-1. 2PL

T1	T2	T3	T4	Time
lock-S(B)	lock-S(B)		lock-S(C)	1
read(B)	lock-S(A)		lock-S(A)	2
	read(A)		read(A)	3
	unlock(A)		unlock(A)	4
	read(B)	lock-X(A)	read(C)	5
	unlock(B)	Write(A)	unlock(C)	6
lock-X(C)				7
unlock(B)				8
write(C)		lock-X(B)		9
unlock(C)		write(B)		10
		unlock(B)		11
		Unlock(A)		12

- So the minimum time is 12 seconds.

2-2-2. Deadlock

1. Find out where deadlock happens, if any, explain why.

T1	T2	T3
lock-X(A)		
Read(A)		
		lock-X(C)
		Read(C)
	lock-X(B)	
	Read(B)	
lock-S(C)		
Read(C)		
	lock-S(A)	
	Read(A)	
		lock-S(B)

- 紅色: T1 被 T3 卡住
- 藍色: T2 被 T1 卡住
- 綠色: T3 被 T2 卡住
- T3 動不了，沒辦法解鎖C → T1 動不了，沒辦法解鎖A → T2 動不了，沒辦法解鎖B。就產生deadlock了

2. Rewrite the schedule to deal with the deadlock with wait-die protocol.

- T1 比 T3 優先，所以他可以 Wait。
- T2 比 T1 後面，所以他會被 Abort，並且下次再進來 Schedule。
  - 在這個例子，他是等到 T1, T3 跑完才進去。

T1	T2	T3
lock-X(A)		
Read(A)		
		lock-X(C)
		Read(C)
	lock-X(B)	
	Read(B)	
lock-S(C) (Wait T3)		
Read(C)		
	lock-S(A) (Abort by T1)	
	Read(A)	lock-S(B)
		Read(B)
Write(A)		
Unlock(A)		
		Write(C)
		Unlock(C)
	lock-X(B)	
	Read(B)	
	lock-S(A)	
	Read(A)	
	Write(B)	
	Unlock(B)	

3. Rewrite the schedule to deal with the deadlock with wound-wait protocol.

- T1 比 T3 優先，所以把 T3 Abort 掉。
- T2 比 T1 後面，所以他可以先等。
- T3 要等 T1, T2 跑完才會再回來。

T1	T2	T3
lock-X(A)		
Read(A)		
		lock-X(C)
		Read(C)
	lock-X(B)	
	Read(B)	
lock-S(C) (Abort T3)		
Read(C)		
	lock-S(A) (Wait T1)	
Write(A)		
Unlock(A)		
Unlock(C)		
	Read(A)	
	Write(B)	
	Unlock(B)	
		lock-X(C)
		Read(C)
		lock-S(B)
		Read(B)
		Write(C)
		Unlock(C)

4. Rewrite the schedule to deal with the deadlock with timestamp-based.

- T1 先進來，要 Write(A) 的時候 T2 已經 Read(A)，所以 Abort 自己。
- T2 先進來，要 Write(B) 的時候 T3 已經 Read(B)，所以 Abort 自己。
- 等 T3 跑完後 T1, T2 再進來。
- 同第一個原因，T1 Abort，然後 T2 跑完之後剩下 T1 所以就讓他跑完。

T1	T2	T3
Read(A)		
		Read(C)
	Read(B)	
Read(C)		
	Read(A)	
		Read(B)
Write(A) (Abort due to T2 Read(A))		
	Write(B) (Abort due to T3 Read(B))	
		Write(C)
Read(A)		
	Read(B)	
Read(C)		
	Read(A)	
Write(A) (Abort again)		
	Write(B)	
Read(A)		
Read(C)		
Write(A)		