# Basi di dati II

## Esame — 4 luglio 2012 — Compito A

## Cenni sulle soluzioni

Rispondere su questo fascicolo.

Tempo a disposizione: due ore e quarantacinque minuti (prova lunga), un'ora e trenta minuti (prova breve).

**Cognome** _____ **Nome** _____ **Matricola** _____ **Ordin.** _____

**Domanda 1** (15% per la prova lunga, 30% per la prova breve)
Considerare la relazione sotto schematizzata, definita su vari attributi, uno dei quali è la chiave, i cui valori sono mostrati. Supponendo una disponibilità di buffer abbastanza ampia, ma non sufficiente a caricare in memoria l'intera relazione (supporre ad esempio una disponibilità di 5 buffer, con un fattore di blocco pari a 3 e quindi uno spazio occupato dalla relazione pari a 9 blocchi), considerare l'esecuzione di un mergesort a più vie (e due passate) sulla relazione e mostrare lo stato delle strutture in memoria centrale e secondaria dopo l'esecuzione di cinque chiamate al metodo `next()` sullo scan che implementa il mergesort. In particolare, mostrare i "run" (cioè le porzioni di file ordinate durante prima passata) memorizzati su disco e i buffer in memoria centrale, evidenziando per ciascun buffer il record corrente. Mostrare anche i record prodotti dalle prime cinque chiamate di `next()`.

| |
|---|
| 111 ... |
| 421 ... |
| 722 ... |
| 211 ... |
| 521 ... |
| 322 ... |
| 942 ... |
| 871 ... |
| 601 ... |
| 124 ... |
| 406 ... |
| 515 ... |
| 125 ... |
| 635 ... |
| 946 ... |
| 855 ... |
| 705 ... |
| 126 ... |
| 308 ... |
| 219 ... |
| 529 ... |
| 138 ... |
| 447 ... |
| 848 ... |
| 717 ... |
| 637 ... |

Run su disco

| |
|---|
| 111 ... |
| 211 ... |
| 322 ... |
| 421 ... |
| 521 ... |
| 601 ... |
| 722 ... |
| 871 ... |
| 942 ... |

| |
|---|
| 124 ... |
| 125 ... |
| 126 ... |
| 406 ... |
| 515 ... |
| 635 ... |
| 705 ... |
| 855 ... |
| 946 ... |

| |
|---|
| 138 ... |
| 219 ... |
| 308 ... |
| 447 ... |
| 529 ... |
| 637 ... |
| 717 ... |
| 848 ... |

Buffer

| | |
|---|---|
| 111 | ... |
| 211 | ... ← |
| 322 | ... |

| | |
|---|---|
| 406 | ... ← |
| 515 | ... |
| 635 | ... |

| | |
|---|---|
| 138 | ... |
| 219 | ... ← |
| 308 | ... |

Record prodotti dalle prime 5 `next()`

| |
|---|
| 111 ... |
| 124 ... |
| 125 ... |
| 126 ... |
| 138 ... |

Il numero ideale di buffer (da utilizzare sia nella prima sia nella seconda passata) è pari alla radice quadrata del numero dei blocchi e quindi a tre.
Nella prima passata, si costruiscono quindi tre run ordinati, ciascuno a partire da tre blocchi del file originario. L'ordinamento di ciascun run può essere effettuato in memoria centrale, usando tre buffer. Poiché il risultato della prima passata viene materializzato, vengono mostrati i run ordinati.
Nella seconda passata, si carica un blocco per ciascuno dei run e si fa il merge usando nuovamente tre buffer. La seconda passata viene svolta in pipeline, e quindi i buffer sono mostrati con il contenuto che hanno quando è stato appena prodotto il quinto record.

**Domanda 4** (10%)
Considerare il seguente scenario in cui due client diversi inviano richieste ad un gestore del controllo di concorrenza. Ciascun client può inviare una richiesta solo dopo che è stata eseguita o rifiutata la precedente (se invece una richiesta viene bloccata da un lock, allora il client rimane inattivo fino alla concessione o allo scadere del timeout). Si supponga che, in caso di stallo, abortisca la transazione che ha avanzato la richiesta per prima. In caso di abort, si supponga che il client rilanci immediatamente la stessa transazione.

| client 1 | client 2 |
|---|---|
| read(x) | |
| | read(x) |
| x = x + 10 | |
| write(x) | |
| | x = x + 20 |
| | write(x) |
| commit | |
| | commit |

Considerare uno scheduler che utilizzi il controllo di concorrenza basato su 2PL e livelli di isolamento SERIALIZABLE e READ COMMITTED. Assumiamo che (come avviene di solito) 2PL preveda

- SERIALIZABLE: lock a due fasi stretto, con lock condivisi per letture e esclusivi per scritture.
- READ COMMITTED: lock condivisi per la lettura senza 2PL (possono essere rilasciati prima della acquisizione di altri lock) ed esclusivi per la scrittura con 2PL stretto (mantenuti fino a commit o abort).

Mostrare il comportamento dello scheduler nei due casi seguenti, supponendo che il valore iniziale dell'oggetto x sia 200. Indicare le operazioni che vengono eseguite nell'ordine con, per ciascuna, il valore che viene letto o scritto. In conclusione, per ciascun caso, dire se si verificano o meno anomalie.

SERIALIZABLE

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| x = x + 10 | x vale 210 | | |
| xlock(x) | bloccata | | |
| | | x = x + 20 | x vale 220 |
| | | xlock(x) | bloccata |
| abort | | | |
| slock(x) | bloccata | | |
| | | write(x) | scrive 220 |
| | | commit | |
| read(x) | legge 220 | | |
| x = x + 10 | x vale 230 | | |
| write(x) | scrive 230 | | |
| commit | | | |

non c'è anomalia

READ COMMITTED

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| x = x + 10 | x vale 210 | | |
| write(x) | scrive 210 | | |
| | | x = x + 20 | x vale 220 |
| | | xlock(x) | bloccata |
| commit | | | |
| | | write(x) | scrive 220 |
| | | commit | |

anomalia: perdita di aggiornamento

**Domanda 5** (10%)
Considerare il seguente scenario in cui due client diversi inviano richieste ad un gestore del controllo di concorrenza. Ciascun client può inviare una richiesta solo dopo che è stata eseguita o rifiutata la precedente (se invece una richiesta viene bloccata da un lock, allora il client rimane inattivo fino alla concessione o allo scadere del timeout). Si supponga che, in caso di stallo, abortisca la transazione che ha avanzato la richiesta per prima. In caso di abort, si supponga che il client rilanci immediatamente la stessa transazione.

| client 1 | client 2 |
|---|---|
| read(x) | |
| | read(x) |
| | x = x + 20 |
| | write(x) |
| | commit |
| read(x) | |
| commit | |

Considerare uno scheduler che utilizzi il controllo di concorrenza basato su multiversioni e livelli di isolamento SERIALIZABLE e READ COMMITTED. Assumiamo che (come avviene di solito) multiversioni preveda

- SERIALIZABLE: le letture fanno riferimento allo stato della base di dati all'inizio della transazione e le scritture di una transazione T sono soggette ad un lock a due fasi stretto (solo per le scritture) e sono ammesse solo se il dato non è stato modificato, dopo l'inizio di T, da altre transazioni.
- READ COMMITTED: le letture fanno riferimento allo stato della base di dati all'inizio della specifica lettura e le scritture sono soggette ad un lock a due fasi stretto (solo per le scritture).

Mostrare il comportamento dello scheduler nei due casi seguenti, supponendo che il valore iniziale dell'oggetto x sia 200. Indicare le operazioni che vengono eseguite nell'ordine con, per ciascuna, il valore che viene letto o scritto. In conclusione, per ciascun caso, dire se si verificano o meno anomalie.

READ COMMITTED

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| | | x = x + 20 | x vale 220 |
| | | write(x) | scrive 220 |
| | | commit | |
| read(x) | legge 220 | | |
| commit | | | |

anomalia: lettura inconsistente; la seconda lettura del client 1 legge il valore corrente di x

SERIALIZABLE

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| | | x = x + 20 | x vale 220 |
| | | write(x) | scrive 220 |
| | | commit | |
| read(x) | legge 200 | | |
| commit | | | |

nessuna anomalia; la seconda lettura del client 1 legge il valore di x all'inizio della transazione

**Domanda 6** (20%)
Si consideri una base di dati con le relazioni (entrambe con indice sulla chiave primaria, costituita, in entrambi i casi, da valori interi positivi consecutivi)

$$R1(\underline{A},B,C),\ R2(\underline{D},E,F)$$

Eseguendo le interrogazioni seguenti, su Postgres (ma il comportamento su altri sistemi è probabilmente lo stesso), si rilevano le seguenti scelte per l'operatore di join:

| | | |
|---|---|---|
| 1. | `select *`<br>`from R1 join R2 on C=D` | Hash join |
| 2. | `select *`<br>`from R1 join R2 on C=D`<br>`where A<500000` | Hash join |
| 3. | `select A, B, C`<br>`from R1 join R2 on C=D`<br>`where A<25` | Nested loop join |

Motivare ciò, valutando, per ciascuna delle tre interrogazioni, il costo di un piano di esecuzione con hash join e uno con nested loop, e supponendo che

- le relazioni abbiano rispettivamente $L_1$=1.000.000 ed $L_2$=2.000.000 ennuple, (con fattore di blocco rispettivamente $f_1$=10 e $f_2$=20)
- gli indici abbiano entrambi i=4 livelli (contando anche radice e foglie) e il fattore di blocco massimo dell'indice sia, in entrambi i casi $f_i$=90
- l'operazione possa contare su un numero di pagine di buffer pari a circa q=400.

Rispondere riempiendo la tabella sottostante, indicando il costo in modo sia simbolico sia numerico (considerare, ovviamente, anche l'eventuale selezione)

| | Hash join | Nested loop join |
|---|---|---|
| 1. | senza uso dei buffer (il che non è realistico), ciascuna delle relazioni viene letta (un accesso per blocco) e rimemorizzata (un accesso per ogni record), poi si leggono in parallelo le due rimemorizzazioni (un accesso per blocco); trascuriamo il costo della generazione del risultato, perché non viene materializzato; il costo è circa<br>$2 \times \frac{L_1}{f_1} + L_1 + 2 \times \frac{L_2}{f_2} + L_2 = 3.600.000$<br>Con l'uso dei buffer il costo si può ridurre, anche di molto; in particolare, con un numero di buffer pari a circa la radice del numero di blocchi del file più piccolo (che abbiamo in questo caso), si ha un costo:<br>$3 \times (\frac{L_1}{f_1} + \frac{L_2}{f_2}) = 600.000$<br>L'algoritmo costruisce rimemorizzazioni hash con tante liste di blocchi quante sono le pagine di buffer disponibili e poi costruisce le ennuple del join esaminendo le liste omologhe; se le liste del file più piccolo entrano in memoria, basta una rilettura per ciascun file (e il costo è quello sopra), altrimenti è un po' maggiore, perché la rimemorizzazione della relazione più piccola va riletta più volte | $\frac{L_1}{f_1} + L_1 \times (i - 2 + 1) = 3.100.000$<br>(nei compiti B e D) $= 6.100.000$<br>(i-2+1): i livelli dell'indice, meno quelli nel buffer (mediamente 2), più 1 per l'accesso al record |
| 2. | sia $a = 500.000$, il numero di valori di A selezionati dalla `where`<br>Il costo è simile al precedente, con l'unica differenza che la memorizzazione e la rilettura del primo file richiedono solo $a/f_1$ accessi invece di $L_1/f_1$; costo complessivo circa 450.000 | $\frac{L_1}{f_1} + a \times (i - 2 + 1) = 1.500.000$<br>l'unica differenze rispetto a sopra è nel numero di accessi diretti; |
| 3. | sia $a = 24$, numero di valori di A selezionati dalla `where`<br>$i + a + \frac{L_2}{f_2} =$ ca 100.000<br>Alla relazione esterna si accede con l'indice: profondità più scansione, perché un intervallo; le poche ennuple vengono organizzate in memoria secondo la struttua hash e quindi basta poi una scansione della seconda relazione | $(i + a) + a \times (i - 1 + 1) =$ ca 100<br>Alla relazione esterna si accede con l'indice: $(i + a)$ per accedere ai record di interesse di R1; poi $a$ accessi diretti a R2, con la sola radice nel buffer |

**Domanda 7** (10%)
Si supponga di avere un recovery manager che utilizzi un checkpoint non quiescente e che scriva record di update ("SetStringRecord" secondo la terminologia di SimpleDB) aventi la forma seguente:

<SETSTRING, TxID, TableName, BlkNo, Offset, BeforeValue, AfterValue >

Si noti che, rispetto alla notazione usata sul libro, l'oggetto dell'operazione viene identificato da TableName (nome della relazione o meglio del file che la memorizza), BlkNo (numero del blocco nel file), Offset (posizione del valore di interesse nel blocco).

In tale contesto, supporre che il recovery manager, al riavvio dopo un crash, trovi i seguenti record nel log:

```
<START, 2>
<SETSTRING, 2, Impiegati, 33, 0, xxxx, Rossi>
<START, 1>
<SETSTRING, 1, Impiegati, 12, 0, xxxx, Neri>
<START, 3>
<COMMIT, 2>
<SETSTRING, 3, Impiegati, 33, 0, Rossi, Verdi>
<START, 4>
<SETSTRING, 4, Impiegati, 35, 0, xxxx, Bruni>
<NQCKPT, 1, 3, 4>
<SETSTRING, 4, Impiegati, 35, 0, Bruni, Neri>
<SETSTRING, 4, Impiegati, 66, 0, xxxx, Bianchi>
<START, 5>
<COMMIT, 4>
```

1. Fino a quale record del log arriva la scansione a ritroso?

<START, 1>

2. Quali modifiche sulla base di dati debbono essere eseguite durante un recovery di tipo undo-redo?

undo(SETSTRING,3,...), undo(SETSTRING,1,...), redo(SETSTRING, 4, ...,35, Bruni), redo(SETSTRING, 4, ...,35, Neri), redo(SETSTRING, 4, ...,35, Bianchi)
In effetti, la prima delle redo non serve, se il checkpoint, come succede di solito (nelle strategie diverse dalla redo-only), salva tutte le pagine sporche del buffer

3. Quali modifiche sulla base di dati debbono essere eseguite durante un recovery di tipo redo-only?

redo(SETSTRING, 4, ...,35, Bruni), redo(SETSTRING, 4, ...,35, Neri), redo(SETSTRING, 4, ...,35, Bianchi)
In questo caso serve anche la prima redo, perché al checkpoint si possono salvare solo le pagine sporche modificate da transazioni andate in commit

4. È possibile che la transazione $T_1$ abbia modificato il buffer contenente il blocco 37 della relazione Impiegati? Spiegare perché e in caso affermativo spiegarne le conseguenze.

Sì, perché la modifica può essere avvenuta con il record corrispondente pure scritto nel log, ma senza che il log sia stato scritto su disco. Nessuna conseguenza.

5. In caso di strategia redo-only, è possibile che la transazione $T_1$ abbia modificato su disco il blocco 37 della relazione Impiegati? Spiegare.

No (con nessuna strategia), perché la regola "write-ahead log" richiede che la modifica su disco venga fatta solo dopo la scrittura su disco del corrispondente record di log

6. È possibile che la transazione $T_1$ abbia modificato su disco il blocco 12 della relazione Impiegati? Spiegare.

Nel caso di strategia redo-only, non può essere avvenuta. Negli altri casi, di regola è avvenuta (almeno al checkpoint)

# Basi di dati II

## Esame — 4 luglio 2012 — Compito B

## Cenni sulle soluzioni

Rispondere su questo fascicolo.

Tempo a disposizione: due ore e quarantacinque minuti (prova lunga), un'ora e trenta minuti (prova breve).

**Cognome** _____ **Nome** _____ **Matricola** _____ **Ordin.** _____

**Domanda 1** (15% per la prova lunga, 30% per la prova breve)
Considerare la relazione sotto schematizzata, definita su vari attributi, uno dei quali è la chiave, i cui valori sono mostrati. Supponendo una disponibilità di buffer abbastanza ampia, ma non sufficiente a caricare in memoria l'intera relazione (supporre ad esempio una disponibilità di 5 buffer, con un fattore di blocco pari a 3 e quindi uno spazio occupato dalla relazione pari a 9 blocchi), considerare l'esecuzione di un mergesort a più vie (e due passate) sulla relazione e mostrare lo stato delle strutture in memoria centrale e secondaria dopo l'esecuzione di cinque chiamate al metodo `next()` sullo scan che implementa il mergesort. In particolare, mostrare i "run" (cioè le porzioni di file ordinate durante prima passata) memorizzati su disco e i buffer in memoria centrale, evidenziando per ciascun buffer il record corrente. Mostrare anche i record prodotti dalle prime cinque chiamate di `next()`.

| |
|---|
| 101 ... |
| 411 ... |
| 712 ... |
| 201 ... |
| 511 ... |
| 312 ... |
| 942 ... |
| 871 ... |
| 601 ... |
| 124 ... |
| 406 ... |
| 515 ... |
| 125 ... |
| 635 ... |
| 946 ... |
| 855 ... |
| 705 ... |
| 126 ... |
| 308 ... |
| 219 ... |
| 529 ... |
| 138 ... |
| 447 ... |
| 848 ... |
| 717 ... |
| 637 ... |

Run su disco

| |
|---|
| 101 ... |
| 201 ... |
| 312 ... |
| 411 ... |
| 511 ... |
| 601 ... |
| 712 ... |
| 871 ... |
| 942 ... |

| |
|---|
| 124 ... |
| 125 ... |
| 126 ... |
| 406 ... |
| 515 ... |
| 635 ... |
| 705 ... |
| 855 ... |
| 946 ... |

| |
|---|
| 138 ... |
| 219 ... |
| 308 ... |
| 447 ... |
| 529 ... |
| 637 ... |
| 717 ... |
| 848 ... |

Buffer

| | |
|---|---|
| 101 | ... |
| 201 | ... |
| 312 | ... |

←

| | |
|---|---|
| 406 | ... |
| 515 | ... |
| 635 | ... |

←

| | |
|---|---|
| 138 | ... |
| 219 | ... |
| 308 | ... |

←

Record prodotti dalle prime 5 `next()`

| |
|---|
| 101 ... |
| 124 ... |
| 125 ... |
| 126 ... |
| 138 ... |

Il numero ideale di buffer (da utilizzare sia nella prima sia nella seconda passata) è pari alla radice quadrata del numero dei blocchi e quindi a tre.
Nella prima passata, si costruiscono quindi tre run ordinati, ciascuno a partire da tre blocchi del file originario. L'ordinamento di ciascun run può essere effettuato in memoria centrale, usando tre buffer. Poiché il risultato della prima passata viene materializzato, vengono mostrati i run ordinati.
Nella seconda passata, si carica un blocco per ciascuno dei run e si fa il merge usando nuovamente tre buffer. La seconda passata viene svolta in pipeline, e quindi i buffer sono mostrati con il contenuto che hanno quando è stato appena prodotto il quinto record.

**Domanda 4** (10%)
Considerare il seguente scenario in cui due client diversi inviano richieste ad un gestore del controllo di concorrenza. Ciascun client può inviare una richiesta solo dopo che è stata eseguita o rifiutata la precedente (se invece una richiesta viene bloccata da un lock, allora il client rimane inattivo fino alla concessione o allo scadere del timeout). Si supponga che, in caso di stallo, abortisca la transazione che ha avanzato la richiesta per prima. In caso di abort, si supponga che il client rilanci immediatamente la stessa transazione.

| client 1 | client 2 |
|---|---|
| read(x) | |
| | read(x) |
| | x = x + 20 |
| | write(x) |
| | commit |
| read(x) | |
| commit | |

Considerare uno scheduler che utilizzi il controllo di concorrenza basato su multiversioni e livelli di isolamento SERIALIZABLE e READ COMMITTED. Assumiamo che (come avviene di solito) multiversioni preveda

- SERIALIZABLE: le letture fanno riferimento allo stato della base di dati all'inizio della transazione e le scritture di una transazione T sono soggette ad un lock a due fasi stretto (solo per le scritture) e sono ammesse solo se il dato non è stato modificato, dopo l'inizio di T, da altre transazioni.
- READ COMMITTED: le letture fanno riferimento allo stato della base di dati all'inizio della specifica lettura e le scritture sono soggette ad un lock a due fasi stretto (solo per le scritture).

Mostrare il comportamento dello scheduler nei due casi seguenti, supponendo che il valore iniziale dell'oggetto x sia 200. Indicare le operazioni che vengono eseguite nell'ordine con, per ciascuna, il valore che viene letto o scritto. In conclusione, per ciascun caso, dire se si verificano o meno anomalie.

**SERIALIZABLE**

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| | | x = x + 20 | x vale 220 |
| | | write(x) | scrive 220 |
| | | commit | |
| read(x) | legge 200 | | |
| commit | | | |

nessuna anomalia; la seconda lettura del client 1 legge il valore di x all'inizio della transazione

**READ COMMITTED**

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| | | x = x + 20 | x vale 220 |
| | | write(x) | scrive 220 |
| | | commit | |
| read(x) | legge 220 | | |
| commit | | | |

anomalia: lettura inconsistente; la seconda lettura del client 1 legge il valore corrente di x

**Domanda 5** (10%)
Considerare il seguente scenario in cui due client diversi inviano richieste ad un gestore del controllo di concorrenza. Ciascun client può inviare una richiesta solo dopo che è stata eseguita o rifiutata la precedente (se invece una richiesta viene bloccata da un lock, allora il client rimane inattivo fino alla concessione o allo scadere del timeout). Si supponga che, in caso di stallo, abortisca la transazione che ha avanzato la richiesta per prima. In caso di abort, si supponga che il client rilanci immediatamente la stessa transazione.

| client 1 | client 2 |
|---|---|
| read(x) | |
| | read(x) |
| x = x + 10 | |
| write(x) | |
| commit | |
| | x = x + 20 |
| | write(x) |
| | commit |

Considerare uno scheduler che utilizzi il controllo di concorrenza basato su 2PL e livelli di isolamento SERIALIZABLE e READ COMMITTED. Assumiamo che (come avviene di solito) 2PL preveda

- SERIALIZABLE: lock a due fasi stretto, con lock condivisi per letture e esclusivi per scritture.
- READ COMMITTED: lock condivisi per la lettura senza 2PL (possono essere rilasciati prima della acquisizione di altri lock) ed esclusivi per la scrittura con 2PL stretto (mantenuti fino a commit o abort).

Mostrare il comportamento dello scheduler nei due casi seguenti, supponendo che il valore iniziale dell'oggetto x sia 200. Indicare le operazioni che vengono eseguite nell'ordine con, per ciascuna, il valore che viene letto o scritto. In conclusione, per ciascun caso, dire se si verificano o meno anomalie.

READ COMMITTED

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| x = x + 10 | x vale 210 | | |
| write(x) | scrive 210 | | |
| commit | | | |
| | | x = x + 20 | x vale 220 |
| | | write(x) | scrive 220 |
| | | commit | |

anomalia: perdita di aggiornamento

SERIALIZABLE

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| x = x + 10 | x vale 210 | | |
| xlock(x) | bloccata | | |
| | | x = x + 20 | x vale 220 |
| | | xlock(x) | bloccata |
| abort | | | |
| slock(x) | bloccata | | |
| | | write(x) | scrive 220 |
| | | commit | |
| read(x) | legge 220 | | |
| x = x + 10 | x vale 230 | | |
| write(x) | scrive 230 | | |
| commit | | | |

non c'è anomalia

**Domanda 6** (20%)
Si consideri una base di dati con le relazioni (entrambe con indice sulla chiave primaria, costituita, in entrambi i casi, da valori interi positivi consecutivi)

$$T1(\underline{A},B,C),\ T2(\underline{D},E,F)$$

Eseguendo le interrogazioni seguenti, su Postgres (ma il comportamento su altri sistemi è probabilmente lo stesso), si rilevano le seguenti scelte per l'operatore di join:

| | | |
|---|---|---|
| 1. | `select *`<br>`from T1 join T2 on C=D` | Hash join |
| 2. | `select *`<br>`from T1 join T2 on C=D`<br>`where A<500000` | Hash join |
| 3. | `select A, B, C`<br>`from T1 join T2 on C=D`<br>`where A<25` | Nested loop join |

Motivare ciò, valutando, per ciascuna delle tre interrogazioni, il costo di un piano di esecuzione con hash join e uno con nested loop, e supponendo che

- le relazioni abbiano rispettivamente $N_1$=2.000.000 ed $N_2$=1.000.000 ennuple, (con fattore di blocco rispettivamente $f_1$=20 e $f_2$=10)
- gli indici abbiano entrambi p=4 livelli (contando anche radice e foglie) e il fattore di blocco massimo dell'indice sia, in entrambi i casi $f_i$=90
- l'operazione possa contare su un numero di pagine di buffer pari a circa q=400.

Rispondere riempiendo la tabella sottostante, indicando il costo in modo sia simbolico sia numerico (considerare, ovviamente, anche l'eventuale selezione)

| | Hash join | Nested loop join |
|---|---|---|
| 1. | senza uso dei buffer (il che non è realistico), ciascuna delle relazioni viene letta (un accesso per blocco) e rimemorizzata (un accesso per ogni record), poi si leggono in parallelo le due rimemorizzazioni (un accesso per blocco); trascuriamo il costo della generazione del risultato, perché non viene materializzato; il costo è circa<br>$2 \times \frac{N_1}{f_1} + N_1 + 2 \times \frac{N_2}{f_2} + N_2 = 3.600.000$<br>Con l'uso dei buffer il costo si può ridurre, anche di molto; in particolare, con un numero di buffer pari a circa la radice del numero di blocchi del file più piccolo (che abbiamo in questo caso), si ha un costo:<br>$3 \times (\frac{N_1}{f_1} + \frac{N_2}{f_2}) = 600.000$<br>L'algoritmo costruisce rimemorizzazioni hash con tante liste di blocchi quante sono le pagine di buffer disponibili e poi costruisce le ennuple del join esaminendo le liste omologhe; se le liste del file più piccolo entrano in memoria, basta una rilettura per ciascun file (e il costo è quello sopra), altrimenti è un po' maggiore, perché la rimemorizzazione della relazione più piccola va riletta più volte | $\frac{N_1}{f_1} + N_1 \times (p - 2 + 1) = 3.100.000$<br>(nei compiti B e D) $= 6.100.000$<br>(p-2+1): i livelli dell'indice, meno quelli nel buffer (mediamente 2), più 1 per l'accesso al record |
| 2. | sia $a = 500.000$, il numero di valori di A selezionati dalla `where`<br>Il costo è simile al precedente, con l'unica differenza che la memorizzazione e la rilettura del primo file richiedono solo $a/f_1$ accessi invece di $N_1/f_1$; costo complessivo circa 450.000 | $\frac{N_1}{f_1} + a \times (p - 2 + 1) = 1.500.000$<br>l'unica differenze rispetto a sopra è nel numero di accessi diretti; |
| 3. | sia $a = 24$, numero di valori di A selezionati dalla `where`<br>$p + a + \frac{N_2}{f_2} =$ ca 100.000<br>Alla relazione esterna si accede con l'indice: profondità più scansione, perché un intervallo; le poche ennuple vengono organizzate in memoria secondo la struttua hash e quindi basta poi una scansione della seconda relazione | $(p + a) + a \times (p - 1 + 1) =$ ca 100<br>Alla relazione esterna si accede con l'indice: $(p + a)$ per accedere ai record di interesse di T1; poi $a$ accessi diretti a T2, con la sola radice nel buffer |

**Domanda 7** (10%)
Si supponga di avere un recovery manager che utilizzi un checkpoint non quiescente e che scriva record di update ("SetStringRecord" secondo la terminologia di SimpleDB) aventi la forma seguente:

<SETSTRING, TxID, TableName, BlkNo, Offset, BeforeValue, AfterValue >

Si noti che, rispetto alla notazione usata sul libro, l'oggetto dell'operazione viene identificato da TableName (nome della relazione o meglio del file che la memorizza), BlkNo (numero del blocco nel file), Offset (posizione del valore di interesse nel blocco).

In tale contesto, supporre che il recovery manager, al riavvio dopo un crash, trovi i seguenti record nel log:

```
<START, 2>
<SETSTRING, 2, Impiegati, 33, 0, xxxx, Rossi>
<START, 1>
<SETSTRING, 1, Impiegati, 37, 0, xxxx, Neri>
<START, 3>
<COMMIT, 2>
<SETSTRING, 3, Impiegati, 33, 0, Rossi, Verdi>
<START, 4>
<SETSTRING, 4, Impiegati, 35, 0, xxxx, Bruni>
<NQCKPT, 1, 3, 4>
<SETSTRING, 4, Impiegati, 35, 0, Bruni, Neri>
<SETSTRING, 4, Impiegati, 66, 0, xxxx, Bianchi>
<START, 5>
<COMMIT, 4>
```

1. Fino a quale record del log arriva la scansione a ritroso?

<START, 1>

2. Quali modifiche sulla base di dati debbono essere eseguite durante un recovery di tipo undo-redo?

undo(SETSTRING,3,...), undo(SETSTRING,1,...), redo(SETSTRING, 4, ...,35, Bruni), redo(SETSTRING, 4, ...,35, Neri), redo(SETSTRING, 4, ...,35, Bianchi)
In effetti, la prima delle redo non serve, se il checkpoint, come succede di solito (nelle strategie diverse dalla redo-only), salva tutte le pagine sporche del buffer

3. Quali modifiche sulla base di dati debbono essere eseguite durante un recovery di tipo redo-only?

redo(SETSTRING, 4, ...,35, Bruni), redo(SETSTRING, 4, ...,35, Neri), redo(SETSTRING, 4, ...,35, Bianchi)
In questo caso serve anche la prima redo, perché al checkpoint si possono salvare solo le pagine sporche modificate da transazioni andate in commit

4. È possibile che la transazione $T_1$ abbia modificato il buffer contenente il blocco 12 della relazione Impiegati? Spiegare perché e in caso affermativo spiegarne le conseguenze.

Sì, perché la modifica può essere avvenuta con il record corrispondente pure scritto nel log, ma senza che il log sia stato scritto su disco. Nessuna conseguenza.

5. In caso di strategia redo-only, è possibile che la transazione $T_1$ abbia modificato su disco il blocco 12 della relazione Impiegati? Spiegare.

No (con nessuna strategia), perché la regola "write-ahead log" richiede che la modifica su disco venga fatta solo dopo la scrittura su disco del corrispondente record di log

6. È possibile che la transazione $T_1$ abbia modificato su disco il blocco 37 della relazione Impiegati? Spiegare.

Nel caso di strategia redo-only, non può essere avvenuta. Negli altri casi, di regola è avvenuta (almeno al checkpoint)

# Basi di dati II

## Esame — 4 luglio 2012 — Compito C

## Cenni sulle soluzioni

Rispondere su questo fascicolo.

Tempo a disposizione: due ore e quarantacinque minuti (prova lunga), un'ora e trenta minuti (prova breve).

**Cognome** _____ **Nome** _____ **Matricola** _____ **Ordin.** _____

**Domanda 1** (15% per la prova lunga, 30% per la prova breve)
Considerare la relazione sotto schematizzata, definita su vari attributi, uno dei quali è la chiave, i cui valori sono mostrati. Supponendo una disponibilità di buffer abbastanza ampia, ma non sufficiente a caricare in memoria l'intera relazione (supporre ad esempio una disponibilità di 5 buffer, con un fattore di blocco pari a 3 e quindi uno spazio occupato dalla relazione pari a 9 blocchi), considerare l'esecuzione di un mergesort a più vie (e due passate) sulla relazione e mostrare lo stato delle strutture in memoria centrale e secondaria dopo l'esecuzione di cinque chiamate al metodo `next()` sullo scan che implementa il mergesort. In particolare, mostrare i "run" (cioè le porzioni di file ordinate durante prima passata) memorizzati su disco e i buffer in memoria centrale, evidenziando per ciascun buffer il record corrente. Mostrare anche i record prodotti dalle prime cinque chiamate di `next()`.

| |
|---|
| 101 ... |
| 421 ... |
| 722 ... |
| 201 ... |
| 521 ... |
| 322 ... |
| 942 ... |
| 871 ... |
| 601 ... |
| 124 ... |
| 406 ... |
| 515 ... |
| 125 ... |
| 635 ... |
| 946 ... |
| 855 ... |
| 705 ... |
| 126 ... |
| 308 ... |
| 219 ... |
| 529 ... |
| 138 ... |
| 447 ... |
| 848 ... |
| 717 ... |
| 637 ... |

Run su disco

| |
|---|
| 101 ... |
| 201 ... |
| 322 ... |
| 421 ... |
| 521 ... |
| 601 ... |
| 722 ... |
| 871 ... |
| 942 ... |

| |
|---|
| 124 ... |
| 125 ... |
| 126 ... |
| 406 ... |
| 515 ... |
| 635 ... |
| 705 ... |
| 855 ... |
| 946 ... |

| |
|---|
| 138 ... |
| 219 ... |
| 308 ... |
| 447 ... |
| 529 ... |
| 637 ... |
| 717 ... |
| 848 ... |

Buffer

| | |
|---|---|
| 101 | ... |
| 201 | ... ← |
| 322 | ... |

| | |
|---|---|
| 406 | ... ← |
| 515 | ... |
| 635 | ... |

| | |
|---|---|
| 138 | ... |
| 219 | ... ← |
| 308 | ... |

Record prodotti dalle prime 5 `next()`

| | |
|---|---|
| 101 | ... |
| 124 | ... |
| 125 | ... |
| 126 | ... |
| 138 | ... |

Il numero ideale di buffer (da utilizzare sia nella prima sia nella seconda passata) è pari alla radice quadrata del numero dei blocchi e quindi a tre.
Nella prima passata, si costruiscono quindi tre run ordinati, ciascuno a partire da tre blocchi del file originario. L'ordinamento di ciascun run può essere effettuato in memoria centrale, usando tre buffer. Poiché il risultato della prima passata viene materializzato, vengono mostrati i run ordinati.
Nella seconda passata, si carica un blocco per ciascuno dei run e si fa il merge usando nuovamente tre buffer. La seconda passata viene svolta in pipeline, e quindi i buffer sono mostrati con il contenuto che hanno quando è stato appena prodotto il quinto record.

**Domanda 4** (10%)
Considerare il seguente scenario in cui due client diversi inviano richieste ad un gestore del controllo di concorrenza. Ciascun client può inviare una richiesta solo dopo che è stata eseguita o rifiutata la precedente (se invece una richiesta viene bloccata da un lock, allora il client rimane inattivo fino alla concessione o allo scadere del timeout). Si supponga che, in caso di stallo, abortisca la transazione che ha avanzato la richiesta per prima. In caso di abort, si supponga che il client rilanci immediatamente la stessa transazione.

| client 1 | client 2 |
|---|---|
| read(x) | |
| | read(x) |
| x = x + 10 | |
| write(x) | |
| | x = x + 20 |
| | write(x) |
| commit | |
| | commit |

Considerare uno scheduler che utilizzi il controllo di concorrenza basato su 2PL e livelli di isolamento SERIALIZABLE e READ COMMITTED. Assumiamo che (come avviene di solito) 2PL preveda

- SERIALIZABLE: lock a due fasi stretto, con lock condivisi per letture e esclusivi per scritture.
- READ COMMITTED: lock condivisi per la lettura senza 2PL (possono essere rilasciati prima della acquisizione di altri lock) ed esclusivi per la scrittura con 2PL stretto (mantenuti fino a commit o abort).

Mostrare il comportamento dello scheduler nei due casi seguenti, supponendo che il valore iniziale dell'oggetto x sia 200. Indicare le operazioni che vengono eseguite nell'ordine con, per ciascuna, il valore che viene letto o scritto. In conclusione, per ciascun caso, dire se si verificano o meno anomalie.

SERIALIZABLE

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| x = x + 10 | x vale 210 | | |
| xlock(x) | bloccata | | |
| | | x = x + 20 | x vale 220 |
| | | xlock(x) | bloccata |
| abort | | | |
| slock(x) | bloccata | | |
| | | write(x) | scrive 220 |
| | | commit | |
| read(x) | legge 220 | | |
| x = x + 10 | x vale 230 | | |
| write(x) | scrive 230 | | |
| commit | | | |

non c'è anomalia

READ COMMITTED

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| x = x + 10 | x vale 210 | | |
| write(x) | scrive 210 | | |
| | | x = x + 20 | x vale 220 |
| | | xlock(x) | bloccata |
| commit | | | |
| | | write(x) | scrive 220 |
| | | commit | |

anomalia: perdita di aggiornamento

**Domanda 5** (10%)
Considerare il seguente scenario in cui due client diversi inviano richieste ad un gestore del controllo di concorrenza. Ciascun client può inviare una richiesta solo dopo che è stata eseguita o rifiutata la precedente (se invece una richiesta viene bloccata da un lock, allora il client rimane inattivo fino alla concessione o allo scadere del timeout). Si supponga che, in caso di stallo, abortisca la transazione che ha avanzato la richiesta per prima. In caso di abort, si supponga che il client rilanci immediatamente la stessa transazione.

| client 1 | client 2 |
|---|---|
| read(x) | |
| | read(x) |
| | x = x + 20 |
| | write(x) |
| | commit |
| read(x) | |
| commit | |

Considerare uno scheduler che utilizzi il controllo di concorrenza basato su multiversioni e livelli di isolamento SERIALIZABLE e READ COMMITTED. Assumiamo che (come avviene di solito) multiversioni preveda

- SERIALIZABLE: le letture fanno riferimento allo stato della base di dati all'inizio della transazione e le scritture di una transazione T sono soggette ad un lock a due fasi stretto (solo per le scritture) e sono ammesse solo se il dato non è stato modificato, dopo l'inizio di T, da altre transazioni.
- READ COMMITTED: le letture fanno riferimento allo stato della base di dati all'inizio della specifica lettura e le scritture sono soggette ad un lock a due fasi stretto (solo per le scritture).

Mostrare il comportamento dello scheduler nei due casi seguenti, supponendo che il valore iniziale dell'oggetto x sia 200. Indicare le operazioni che vengono eseguite nell'ordine con, per ciascuna, il valore che viene letto o scritto. In conclusione, per ciascun caso, dire se si verificano o meno anomalie.

READ COMMITTED

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| | | x = x + 20 | x vale 220 |
| | | write(x) | scrive 220 |
| | | commit | |
| read(x) | legge 220 | | |
| commit | | | |

anomalia: lettura inconsistente; la seconda lettura del client 1 legge il valore corrente di x

SERIALIZABLE

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| | | x = x + 20 | x vale 220 |
| | | write(x) | scrive 220 |
| | | commit | |
| read(x) | legge 200 | | |
| commit | | | |

nessuna anomalia; la seconda lettura del client 1 legge il valore di x all'inizio della transazione

**Domanda 6** (20%)
Si consideri una base di dati con le relazioni (entrambe con indice sulla chiave primaria, costituita, in entrambi i casi, da valori interi positivi consecutivi)

$$R1(\underline{A},B,C),\ R2(\underline{D},E,F)$$

Eseguendo le interrogazioni seguenti, su Postgres (ma il comportamento su altri sistemi è probabilmente lo stesso), si rilevano le seguenti scelte per l'operatore di join:

| | | |
|---|---|---|
| 1. | `select *`<br>`from R1 join R2 on C=D` | Hash join |
| 2. | `select *`<br>`from R1 join R2 on C=D`<br>`where A<500000` | Hash join |
| 3. | `select A, B, C`<br>`from R1 join R2 on C=D`<br>`where A<25` | Nested loop join |

Motivare ciò, valutando, per ciascuna delle tre interrogazioni, il costo di un piano di esecuzione con hash join e uno con nested loop, e supponendo che

- le relazioni abbiano rispettivamente $L_1$=1.000.000 ed $L_2$=2.000.000 ennuple, (con fattore di blocco rispettivamente $f_1$=10 e $f_2$=20)
- gli indici abbiano entrambi i=4 livelli (contando anche radice e foglie) e il fattore di blocco massimo dell'indice sia, in entrambi i casi $f_i$=90
- l'operazione possa contare su un numero di pagine di buffer pari a circa q=400.

Rispondere riempiendo la tabella sottostante, indicando il costo in modo sia simbolico sia numerico (considerare, ovviamente, anche l'eventuale selezione)

| | Hash join | Nested loop join |
|---|---|---|
| 1. | senza uso dei buffer (il che non è realistico), ciascuna delle relazioni viene letta (un accesso per blocco) e rimemorizzata (un accesso per ogni record), poi si leggono in parallelo le due rimemorizzazioni (un accesso per blocco); trascuriamo il costo della generazione del risultato, perché non viene materializzato; il costo è circa<br>$2 \times \frac{L_1}{f_1} + L_1 + 2 \times \frac{L_2}{f_2} + L_2 = 3.600.000$<br>Con l'uso dei buffer il costo si può ridurre, anche di molto; in particolare, con un numero di buffer pari a circa la radice del numero di blocchi del file più piccolo (che abbiamo in questo caso), si ha un costo:<br>$3 \times (\frac{L_1}{f_1} + \frac{L_2}{f_2}) = 600.000$<br>L'algoritmo costruisce rimemorizzazioni hash con tante liste di blocchi quante sono le pagine di buffer disponibili e poi costruisce le ennuple del join esaminendo le liste omologhe; se le liste del file più piccolo entrano in memoria, basta una rilettura per ciascun file (e il costo è quello sopra), altrimenti è un po' maggiore, perché la rimemorizzazione della relazione più piccola va riletta più volte | $\frac{L_1}{f_1} + L_1 \times (i - 2 + 1) = 3.100.000$<br>(nei compiti B e D) $= 6.100.000$<br>(i-2+1): i livelli dell'indice, meno quelli nel buffer (mediamente 2), più 1 per l'accesso al record |
| 2. | sia $a = 500.000$, il numero di valori di A selezionati dalla `where`<br>Il costo è simile al precedente, con l'unica differenza che la memorizzazione e la rilettura del primo file richiedono solo $a/f_1$ accessi invece di $L_1/f_1$; costo complessivo circa 450.000 | $\frac{L_1}{f_1} + a \times (i - 2 + 1) = 1.500.000$<br>l'unica differenze rispetto a sopra è nel numero di accessi diretti; |
| 3. | sia $a = 24$, numero di valori di A selezionati dalla `where`<br>$i + a + \frac{L_2}{f_2} =$ ca 100.000<br>Alla relazione esterna si accede con l'indice: profondità più scansione, perché un intervallo; le poche ennuple vengono organizzate in memoria secondo la struttua hash e quindi basta poi una scansione della seconda relazione | $(i + a) + a \times (i - 1 + 1) =$ ca 100<br>Alla relazione esterna si accede con l'indice: $(i + a)$ per accedere ai record di interesse di R1; poi $a$ accessi diretti a R2, con la sola radice nel buffer |

**Domanda 7** (10%)
Si supponga di avere un recovery manager che utilizzi un checkpoint non quiescente e che scriva record di update ("SetStringRecord" secondo la terminologia di SimpleDB) aventi la forma seguente:

<SETSTRING, TxID, TableName, BlkNo, Offset, BeforeValue, AfterValue >

Si noti che, rispetto alla notazione usata sul libro, l'oggetto dell'operazione viene identificato da TableName (nome della relazione o meglio del file che la memorizza), BlkNo (numero del blocco nel file), Offset (posizione del valore di interesse nel blocco).

In tale contesto, supporre che il recovery manager, al riavvio dopo un crash, trovi i seguenti record nel log:

```
<START, 2>
<SETSTRING, 2, Impiegati, 33, 0, xxxx, Rossi>
<START, 1>
<SETSTRING, 1, Impiegati, 19, 0, xxxx, Neri>
<START, 3>
<COMMIT, 2>
<SETSTRING, 3, Impiegati, 33, 0, Rossi, Verdi>
<START, 4>
<SETSTRING, 4, Impiegati, 35, 0, xxxx, Bruni>
<NQCKPT, 1, 3, 4>
<SETSTRING, 4, Impiegati, 35, 0, Bruni, Neri>
<SETSTRING, 4, Impiegati, 66, 0, xxxx, Bianchi>
<START, 5>
<COMMIT, 4>
```

1. Fino a quale record del log arriva la scansione a ritroso?

<START, 1>

2. Quali modifiche sulla base di dati debbono essere eseguite durante un recovery di tipo undo-redo?

undo(SETSTRING,3,...), undo(SETSTRING,1,...), redo(SETSTRING, 4, ...,35, Bruni), redo(SETSTRING, 4, ...,35, Neri), redo(SETSTRING, 4, ...,35, Bianchi)
In effetti, la prima delle redo non serve, se il checkpoint, come succede di solito (nelle strategie diverse dalla redo-only), salva tutte le pagine sporche del buffer

3. Quali modifiche sulla base di dati debbono essere eseguite durante un recovery di tipo redo-only?

redo(SETSTRING, 4, ...,35, Bruni), redo(SETSTRING, 4, ...,35, Neri), redo(SETSTRING, 4, ...,35, Bianchi)
In questo caso serve anche la prima redo, perché al checkpoint si possono salvare solo le pagine sporche modificate da transazioni andate in commit

4. È possibile che la transazione $T_1$ abbia modificato il buffer contenente il blocco 42 della relazione Impiegati? Spiegare perché e in caso affermativo spiegarne le conseguenze.

Sì, perché la modifica può essere avvenuta con il record corrispondente pure scritto nel log, ma senza che il log sia stato scritto su disco. Nessuna conseguenza.

5. In caso di strategia redo-only, è possibile che la transazione $T_1$ abbia modificato su disco il blocco 42 della relazione Impiegati? Spiegare.

No (con nessuna strategia), perché la regola "write-ahead log" richiede che la modifica su disco venga fatta solo dopo la scrittura su disco del corrispondente record di log

6. È possibile che la transazione $T_1$ abbia modificato su disco il blocco 19 della relazione Impiegati? Spiegare.

Nel caso di strategia redo-only, non può essere avvenuta. Negli altri casi, di regola è avvenuta (almeno al checkpoint)

# Basi di dati II

## Esame — 4 luglio 2012 — Compito D

## Cenni sulle soluzioni

Rispondere su questo fascicolo.

Tempo a disposizione: due ore e quarantacinque minuti (prova lunga), un'ora e trenta minuti (prova breve).

**Cognome** _____ **Nome** _____ **Matricola** _____ **Ordin.** _____

**Domanda 1** (15% per la prova lunga, 30% per la prova breve)
Considerare la relazione sotto schematizzata, definita su vari attributi, uno dei quali è la chiave, i cui valori sono mostrati. Supponendo una disponibilità di buffer abbastanza ampia, ma non sufficiente a caricare in memoria l'intera relazione (supporre ad esempio una disponibilità di 5 buffer, con un fattore di blocco pari a 3 e quindi uno spazio occupato dalla relazione pari a 9 blocchi), considerare l'esecuzione di un mergesort a più vie (e due passate) sulla relazione e mostrare lo stato delle strutture in memoria centrale e secondaria dopo l'esecuzione di cinque chiamate al metodo `next()` sullo scan che implementa il mergesort. In particolare, mostrare i "run" (cioè le porzioni di file ordinate durante prima passata) memorizzati su disco e i buffer in memoria centrale, evidenziando per ciascun buffer il record corrente. Mostrare anche i record prodotti dalle prime cinque chiamate di `next()`.

| |
|---|
| 111 ... |
| 411 ... |
| 712 ... |
| 211 ... |
| 511 ... |
| 312 ... |
| 942 ... |
| 871 ... |
| 601 ... |
| 124 ... |
| 406 ... |
| 515 ... |
| 125 ... |
| 635 ... |
| 946 ... |
| 855 ... |
| 705 ... |
| 126 ... |
| 308 ... |
| 219 ... |
| 529 ... |
| 138 ... |
| 447 ... |
| 848 ... |
| 717 ... |
| 637 ... |

Run su disco

| |
|---|
| 111 ... |
| 211 ... |
| 312 ... |
| 411 ... |
| 511 ... |
| 601 ... |
| 712 ... |
| 871 ... |
| 942 ... |

| |
|---|
| 124 ... |
| 125 ... |
| 126 ... |
| 406 ... |
| 515 ... |
| 635 ... |
| 705 ... |
| 855 ... |
| 946 ... |

| |
|---|
| 138 ... |
| 219 ... |
| 308 ... |
| 447 ... |
| 529 ... |
| 637 ... |
| 717 ... |
| 848 ... |

Buffer

| | |
|---|---|
| 111 | ... |
| 211 | ... ← |
| 312 | ... |

| | |
|---|---|
| 406 | ... ← |
| 515 | ... |
| 635 | ... |

| | |
|---|---|
| 138 | ... |
| 219 | ... ← |
| 308 | ... |

Record prodotti dalle prime 5 `next()`

| |
|---|
| 111 ... |
| 124 ... |
| 125 ... |
| 126 ... |
| 138 ... |

Il numero ideale di buffer (da utilizzare sia nella prima sia nella seconda passata) è pari alla radice quadrata del numero dei blocchi e quindi a tre.
Nella prima passata, si costruiscono quindi tre run ordinati, ciascuno a partire da tre blocchi del file originario. L'ordinamento di ciascun run può essere effettuato in memoria centrale, usando tre buffer. Poiché il risultato della prima passata viene materializzato, vengono mostrati i run ordinati.
Nella seconda passata, si carica un blocco per ciascuno dei run e si fa il merge usando nuovamente tre buffer. La seconda passata viene svolta in pipeline, e quindi i buffer sono mostrati con il contenuto che hanno quando è stato appena prodotto il quinto record.

**Domanda 4** (10%)
Considerare il seguente scenario in cui due client diversi inviano richieste ad un gestore del controllo di concorrenza. Ciascun client può inviare una richiesta solo dopo che è stata eseguita o rifiutata la precedente (se invece una richiesta viene bloccata da un lock, allora il client rimane inattivo fino alla concessione o allo scadere del timeout). Si supponga che, in caso di stallo, abortisca la transazione che ha avanzato la richiesta per prima. In caso di abort, si supponga che il client rilanci immediatamente la stessa transazione.

| client 1 | client 2 |
|---|---|
| read(x) | |
| | read(x) |
| | x = x + 20 |
| | write(x) |
| | commit |
| read(x) | |
| commit | |

Considerare uno scheduler che utilizzi il controllo di concorrenza basato su multiversioni e livelli di isolamento SERIALIZABLE e READ COMMITTED. Assumiamo che (come avviene di solito) multiversioni preveda

- SERIALIZABLE: le letture fanno riferimento allo stato della base di dati all'inizio della transazione e le scritture di una transazione T sono soggette ad un lock a due fasi stretto (solo per le scritture) e sono ammesse solo se il dato non è stato modificato, dopo l'inizio di T, da altre transazioni.
- READ COMMITTED: le letture fanno riferimento allo stato della base di dati all'inizio della specifica lettura e le scritture sono soggette ad un lock a due fasi stretto (solo per le scritture).

Mostrare il comportamento dello scheduler nei due casi seguenti, supponendo che il valore iniziale dell'oggetto x sia 200. Indicare le operazioni che vengono eseguite nell'ordine con, per ciascuna, il valore che viene letto o scritto. In conclusione, per ciascun caso, dire se si verificano o meno anomalie.

**SERIALIZABLE**

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| | | x = x + 20 | x vale 220 |
| | | write(x) | scrive 220 |
| | | commit | |
| read(x) | legge 200 | | |
| commit | | | |

nessuna anomalia; la seconda lettura del client 1 legge il valore di x all'inizio della transazione

**READ COMMITTED**

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| | | x = x + 20 | x vale 220 |
| | | write(x) | scrive 220 |
| | | commit | |
| read(x) | legge 220 | | |
| commit | | | |

anomalia: lettura inconsistente; la seconda lettura del client 1 legge il valore corrente di x

**Domanda 5** (10%)
Considerare il seguente scenario in cui due client diversi inviano richieste ad un gestore del controllo di concorrenza. Ciascun client può inviare una richiesta solo dopo che è stata eseguita o rifiutata la precedente (se invece una richiesta viene bloccata da un lock, allora il client rimane inattivo fino alla concessione o allo scadere del timeout). Si supponga che, in caso di stallo, abortisca la transazione che ha avanzato la richiesta per prima. In caso di abort, si supponga che il client rilanci immediatamente la stessa transazione.

| client 1 | client 2 |
|---|---|
| read(x) | |
| | read(x) |
| x = x + 10 | |
| write(x) | |
| commit | |
| | x = x + 20 |
| | write(x) |
| | commit |

Considerare uno scheduler che utilizzi il controllo di concorrenza basato su 2PL e livelli di isolamento SERIALIZABLE e READ COMMITTED. Assumiamo che (come avviene di solito) 2PL preveda

- SERIALIZABLE: lock a due fasi stretto, con lock condivisi per letture e esclusivi per scritture.
- READ COMMITTED: lock condivisi per la lettura senza 2PL (possono essere rilasciati prima della acquisizione di altri lock) ed esclusivi per la scrittura con 2PL stretto (mantenuti fino a commit o abort).

Mostrare il comportamento dello scheduler nei due casi seguenti, supponendo che il valore iniziale dell'oggetto x sia 200. Indicare le operazioni che vengono eseguite nell'ordine con, per ciascuna, il valore che viene letto o scritto. In conclusione, per ciascun caso, dire se si verificano o meno anomalie.

READ COMMITTED

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| x = x + 10 | x vale 210 | | |
| write(x) | scrive 210 | | |
| commit | | | |
| | | x = x + 20 | x vale 220 |
| | | write(x) | scrive 220 |
| | | commit | |

anomalia: perdita di aggiornamento

SERIALIZABLE

| client 1 | | client 2 | |
|---|---|---|---|
| read(x) | legge 200 | | |
| | | read(x) | legge 200 |
| x = x + 10 | x vale 210 | | |
| xlock(x) | bloccata | | |
| | | x = x + 20 | x vale 220 |
| | | xlock(x) | bloccata |
| abort | | | |
| slock(x) | bloccata | | |
| | | write(x) | scrive 220 |
| | | commit | |
| read(x) | legge 220 | | |
| x = x + 10 | x vale 230 | | |
| write(x) | scrive 230 | | |
| commit | | | |

non c'è anomalia

**Domanda 6** (20%)
Si consideri una base di dati con le relazioni (entrambe con indice sulla chiave primaria, costituita, in entrambi i casi, da valori interi positivi consecutivi)

$$T1(\underline{A},B,C),\ T2(\underline{D},E,F)$$

Eseguendo le interrogazioni seguenti, su Postgres (ma il comportamento su altri sistemi è probabilmente lo stesso), si rilevano le seguenti scelte per l'operatore di join:

| | | |
|---|---|---|
| 1. | `select *`<br>`from T1 join T2 on C=D` | Hash join |
| 2. | `select *`<br>`from T1 join T2 on C=D`<br>`where A<500000` | Hash join |
| 3. | `select A, B, C`<br>`from T1 join T2 on C=D`<br>`where A<25` | Nested loop join |

Motivare ciò, valutando, per ciascuna delle tre interrogazioni, il costo di un piano di esecuzione con hash join e uno con nested loop, e supponendo che

- le relazioni abbiano rispettivamente $N_1$=2.000.000 ed $N_2$=1.000.000 ennuple, (con fattore di blocco rispettivamente $f_1$=20 e $f_2$=10)
- gli indici abbiano entrambi p=4 livelli (contando anche radice e foglie) e il fattore di blocco massimo dell'indice sia, in entrambi i casi $f_i$=90
- l'operazione possa contare su un numero di pagine di buffer pari a circa q=400.

Rispondere riempiendo la tabella sottostante, indicando il costo in modo sia simbolico sia numerico (considerare, ovviamente, anche l'eventuale selezione)

| | Hash join | Nested loop join |
|---|---|---|
| 1. | senza uso dei buffer (il che non è realistico), ciascuna delle relazioni viene letta (un accesso per blocco) e rimemorizzata (un accesso per ogni record), poi si leggono in parallelo le due rimemorizzazioni (un accesso per blocco); trascuriamo il costo della generazione del risultato, perché non viene materializzato; il costo è circa<br>$2 \times \frac{N_1}{f_1} + N_1 + 2 \times \frac{N_2}{f_2} + N_2 = 3.600.000$<br>Con l'uso dei buffer il costo si può ridurre, anche di molto; in particolare, con un numero di buffer pari a circa la radice del numero di blocchi del file più piccolo (che abbiamo in questo caso), si ha un costo:<br>$3 \times (\frac{N_1}{f_1} + \frac{N_2}{f_2}) = 600.000$<br>L'algoritmo costruisce rimemorizzazioni hash con tante liste di blocchi quante sono le pagine di buffer disponibili e poi costruisce le ennuple del join esaminendo le liste omologhe; se le liste del file più piccolo entrano in memoria, basta una rilettura per ciascun file (e il costo è quello sopra), altrimenti è un po' maggiore, perché la rimemorizzazione della relazione più piccola va riletta più volte | $\frac{N_1}{f_1} + N_1 \times (p - 2 + 1) = 3.100.000$<br>(nei compiti B e D) $= 6.100.000$<br>(p-2+1): i livelli dell'indice, meno quelli nel buffer (mediamente 2), più 1 per l'accesso al record |
| 2. | sia $a = 500.000$, il numero di valori di A selezionati dalla `where`<br>Il costo è simile al precedente, con l'unica differenza che la memorizzazione e la rilettura del primo file richiedono solo $a/f_1$ accessi invece di $N_1/f_1$; costo complessivo circa 450.000 | $\frac{N_1}{f_1} + a \times (p - 2 + 1) = 1.500.000$<br>l'unica differenze rispetto a sopra è nel numero di accessi diretti; |
| 3. | sia $a = 24$, numero di valori di A selezionati dalla `where`<br>$p + a + \frac{N_2}{f_2} =$ ca 100.000<br>Alla relazione esterna si accede con l'indice: profondità più scansione, perché un intervallo; le poche ennuple vengono organizzate in memoria secondo la struttua hash e quindi basta poi una scansione della seconda relazione | $(p + a) + a \times (p - 1 + 1) =$ ca 100<br>Alla relazione esterna si accede con l'indice: $(p + a)$ per accedere ai record di interesse di T1; poi $a$ accessi diretti a T2, con la sola radice nel buffer |

**Domanda 7** (10%)
Si supponga di avere un recovery manager che utilizzi un checkpoint non quiescente e che scriva record di update ("SetStringRecord" secondo la terminologia di SimpleDB) aventi la forma seguente:

<SETSTRING, TxID, TableName, BlkNo, Offset, BeforeValue, AfterValue >

Si noti che, rispetto alla notazione usata sul libro, l'oggetto dell'operazione viene identificato da TableName (nome della relazione o meglio del file che la memorizza), BlkNo (numero del blocco nel file), Offset (posizione del valore di interesse nel blocco).

In tale contesto, supporre che il recovery manager, al riavvio dopo un crash, trovi i seguenti record nel log:

```
<START, 2>
<SETSTRING, 2, Impiegati, 33, 0, xxxx, Rossi>
<START, 1>
<SETSTRING, 1, Impiegati, 42, 0, xxxx, Neri>
<START, 3>
<COMMIT, 2>
<SETSTRING, 3, Impiegati, 33, 0, Rossi, Verdi>
<START, 4>
<SETSTRING, 4, Impiegati, 35, 0, xxxx, Bruni>
<NQCKPT, 1, 3, 4>
<SETSTRING, 4, Impiegati, 35, 0, Bruni, Neri>
<SETSTRING, 4, Impiegati, 66, 0, xxxx, Bianchi>
<START, 5>
<COMMIT, 4>
```

1. Fino a quale record del log arriva la scansione a ritroso?

<START, 1>

2. Quali modifiche sulla base di dati debbono essere eseguite durante un recovery di tipo undo-redo?

undo(SETSTRING,3,...), undo(SETSTRING,1,...), redo(SETSTRING, 4, ...,35, Bruni), redo(SETSTRING, 4, ...,35, Neri), redo(SETSTRING, 4, ...,35, Bianchi)
In effetti, la prima delle redo non serve, se il checkpoint, come succede di solito (nelle strategie diverse dalla redo-only), salva tutte le pagine sporche del buffer

3. Quali modifiche sulla base di dati debbono essere eseguite durante un recovery di tipo redo-only?

redo(SETSTRING, 4, ...,35, Bruni), redo(SETSTRING, 4, ...,35, Neri), redo(SETSTRING, 4, ...,35, Bianchi)
In questo caso serve anche la prima redo, perché al checkpoint si possono salvare solo le pagine sporche modificate da transazioni andate in commit

4. È possibile che la transazione $T_1$ abbia modificato il buffer contenente il blocco 19 della relazione Impiegati? Spiegare perché e in caso affermativo spiegarne le conseguenze.

Sì, perché la modifica può essere avvenuta con il record corrispondente pure scritto nel log, ma senza che il log sia stato scritto su disco. Nessuna conseguenza.

5. In caso di strategia redo-only, è possibile che la transazione $T_1$ abbia modificato su disco il blocco 19 della relazione Impiegati? Spiegare.

No (con nessuna strategia), perché la regola "write-ahead log" richiede che la modifica su disco venga fatta solo dopo la scrittura su disco del corrispondente record di log

6. È possibile che la transazione $T_1$ abbia modificato su disco il blocco 42 della relazione Impiegati? Spiegare.

Nel caso di strategia redo-only, non può essere avvenuta. Negli altri casi, di regola è avvenuta (almeno al checkpoint)