Basi di dati II, primo modulo
Esercizi di autovalutazione — 27 marzo 2013
Cenni sulle soluzioni

Domanda 1 Il check-point (quiescente) prevede, fra le altre, le seguenti attività (durante le quali non sono accettate richieste di nuove transazioni):

1. scrittura su disco (flush) di tutte le pagine sporche del buffer
2. scrittura in memoria stabile di tutti i record del log incluso il record di checkpoint

Spiegare perché il checkpoint sarebbe inutilizzabile se le operazioni avvenissero in ordine inverso (prima la scrittura del record di checkpoint e poi la flush del buffer).

Discussione
Se la flush avvenisse dopo la force del checkpoint, un eventuale guasto fra la prima e la seconda operazione potrebbe rendere impreciso (e quindi inutilizzabile) il checkpoint: alcune transazioni andate in commit potrebbero non aver svolto tutte le operazioni di scrittura in memoria secondaria.

Domanda 2 Come noto, alcuni DBMS permettono una tecnica di memorizzazione chiamata "co-clustering" o "clustering eterogeneo," in cui un file contiene record di due o più relazioni e tali record sono allocati secondo i valori di opportuni campi dell'una e dell'altra relazione. Ad esempio, date due relazioni

- *Ordini(CodiceOrdine,Cliente,Data,Totale)*
- *LineeOrdine(CodiceOrdine,Linea,Prodotto,Importo)*

questa tecnica (con riferimento agli attributi *CodiceOrdine* delle due relazioni) permetterebbe una memorizzazione contigua di ciascun ordine con le rispettive "linee d'ordine," cioè dei prodotti ordinati (ciascun ordine fa riferimento a più prodotti, ognuno su una "linea").
Supponendo che, nell'uno come nell'altro caso, vi sia un indice secondario sulla chiave primaria in ciascuna delle relazioni (sulla relazione LineeOrdine, l'indice sia su *CodiceOrdine* e *Linea*). Con riferimento all'esempio, indicare quali delle seguenti operazioni possono trarre vantaggio dall'uso di questa opportunità e quali ne possono essere penalizzate (spiegare la risposta possibilmente anche in termini quantitativi, attraverso l'uso di esempi):

1. stampa di tutte le informazioni (incluse i dettagli delle linee d'ordine) di tutti gli ordini (ordinati per codice)
2. stampa di tutte le informazioni di un ordine
3. stampa delle informazioni sintetiche (codice, cliente, data, totale) di tutti gli ordini

Discussione (molto schematica e a ovvio rischio di imprecisioni)

L'aspetto cruciale da sottolineare è l'organizzazione del file in blocchi (in quanto, come noto, il numero di accessi in memoria secondaria è pari al numero di blocchi cui si deve accedere). La presenza di specifiche strutture fisiche (indici, hash, ordinamento) è marginale ai fini della risposta a questa domanda, perché esse possono avere sostanzialmente la stessa efficacia sia in presenza sia in assenza di "co-clustering." Le tre operazioni:

1. può ottenere un lieve vantaggio, in quanto il join si trova già preparato, ma la cosa è poco rilevante, perché è comunque necessario un ordinamento, che ha sostanzialmente lo stesso costo nei due casi;
2. qui si avrebbe un vantaggio, in quanto si troverebbero in uno stesso blocco sia le informazioni sintetiche sia i dettagli che sono per giunta raggruppati, mentre senza coclustering si possono trovare in blocchi diversi; ovviamente, nel caso singolo il vantaggio è piccolo, ma se l'operazione è molto frequente, esso diventa significativo, in quanto il costo unitario è più che dimezzato, perché si fa un solo accesso diretto (visita dell'indice) anziché due e si accede ad un solo blocco di dati anziché due o, solitamente, più di due, uno per linea d'ordine più quello per l'ordine;
3. in questo caso si ha sicuramente un peggioramento, perché si deve accedere a tutti blocchi del file che contengono entrambe le relazioni, mentre senza "co-clustering" si dovrebbe accedere solo ai blocchi relativi alla relazione *Ordini*; quantitativamente: supponiamo che i blocchi siano di 1000 byte e che *Ordini* abbia 10.000 ennuple di 100 byte ciascuna, mentre *LineeOrdine* abbia 200.000 ennuple di 50 byte ciascuna; senza "co-clustering" *Ordini* occupa 1000 blocchi (e quindi questa operazione richiede 1000 accessi) mentre con il "co-clustering" le due relazioni (non più scandibili separatamente) occupano 11.000 blocchi (e l'operazione richiede quindi 11.000 accessi).

Domanda 3 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, la notazione è diversa da quella usata nel 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, 1>
<START, 2>
<SETSTRING, 2, Impiegati, 33, 0, xxxx, Rossi>
<SETSTRING, 1, Impiegati, 44, 0, xxxx, Neri>
<START, 3>
<COMMIT, 2>
<SETSTRING, 3, Impiegati, 33, 0, Rossi, Verdi>
<START, 4>
<SETSTRING, 4, Impiegati, 55, 0, xxxx, Bruni>
<NQCKPT, 1, 3, 4>
<SETSTRING, 4, Impiegati, 55, 0, Bruni, Neri>
<SETSTRING, 4, Impiegati, 66, 0, xxxx, Bianchi>
<START, 5>
<COMMIT, 4>
```

1. Indicare quali modifiche sulla base di dati vengono eseguite durante un recovery di tipo undo-redo.

undo(SETSTRING,3,...), undo(SETSTRING,1,...), redo(SETSTRING, 4, ...,55, Bruni), redo(SETSTRING, 4, ...,55, Neri), redo(SETSTRING, 4, ...,66, 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

2. Indicare quali modifiche sulla base di dati debbono essere eseguite durante un recovery di tipo undo-only (cioè no-redo)

undo(SETSTRING,3,...), undo(SETSTRING,1,...)

3. Indicare quali modifiche sulla base di dati debbono essere eseguite durante un recovery di tipo redo-only (cioè no-undo)

redo(SETSTRING, 4, ...,55, Bruni), redo(SETSTRING, 4, ...,55, Neri), redo(SETSTRING, 4, ...,66, 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 sia andata in commit?

No, perché l'operazione di commit si completa solo con la scrittura del relativo record nel log, e tale record non c'è

5. è possibile che la transazione T_1 abbia modificato il buffer contenente il blocco 20 della relazione Impiegati?

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

6. è possibile che la transazione T_1 abbia modificato su disco il blocco 20 della relazione Impiegati?

No, 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

7. è possibile che la transazione T_1 non abbia modificato su disco il blocco 44 della relazione Impiegati?

In caso di strategia redo-only, la modifica sicuramente non è avvenuta (si scrive solo dopo il commit). Negli altri casi, il checkpoint viene di solito implementato copiando su disco tutte le pagine sporche del buffer