Basi di dati II, primo modulo 24 giugno 2011— Compito breve
Cenni sulle soluzioni

Cognome _____ **Nome** _____ **Matricola** _____

Domanda 1 Come noto, si stanno diffondendo applicazioni nelle quali è necessaria una grande scalabilità e che vengono quindi realizzate con architetture parallele con centinaia o migliaia di processori. Queste architetture sono di solito di tipo "shared-nothing," cioè ogni processore ha la propria memoria centrale e i propri dischi. Sulla base delle conoscenze relative alla gestione dei lock e dei buffer, spiegare perché invece una architettura "shared-disk," in cui ogni processore ha la propria memoria ma l'insieme dei dischi è condiviso, risulterebbe poco scalabile, nonostante l'ampia disponibilità di memoria centrale.

In una architettura shared-disk, la tabella dei lock e i buffer dovrebbero essere presenti su ogni nodo, con evidenti difficoltà di sincronizzazione

Per queste applicazioni vengono usati di solito sistemi che premettono di eseguire solo operazioni semplici e localizzate sui singoli nodi dell'architettura. Di solito, i dati venogno partizionati fra i vari nodi sulla base di un campo chiave (o pseudochiave selettiva) e ogni operazione fa riferimento a uno o pochi valori della chiave. Il criterio di distribuzione è noto e il suo utilizzo non richiede l'accesso ai singoli nodi (dato un valore della chiave si individua direttamente il nodo competente). Spiegare perché le operazioni di ricerca diventano efficienti, a patto di una distribuzione bilanciata dei dati e delle richieste fra i vari nodi. Spiegare in particolare perché è fondamentale la caratteristica sopra citata per il criterio di distribuzione.

Se il criterio di distribuzione è noto e può essere utilizzato dal front-end, allora solo un nodo è coinvolto in ciascuna operazione e se la distibuzione tanto dei dati quanto delle operazioni è uniforme, allora il carico si distribuisce di conseguenza.

Spiegare perché la localizzazione è fondamentale per le transazioni (cioè perché transazioni su più nodi sarebbero inefficienti).

Se le transazioni non sono locali, serve il commit a due fasi, che porta a significativi overhead di scambi di messaggi e attese di sincronizzazione.

Basi di dati II, primo modulo 24 giugno 2011— Compito A
Cenni sulle soluzioni

Cognome _____ **Nome** _____ **Matricola** _____

Domanda 1 (25%) Come noto, si stanno diffondendo applicazioni nelle quali è necessaria una grande scalabilità e che vengono quindi realizzate con architetture parallele con centinaia o migliaia di processori. Queste architetture sono di solito di tipo "shared-nothing," cioè ogni processore ha la propria memoria centrale e i propri dischi. Sulla base delle conoscenze relative alla gestione dei lock e dei buffer, spiegare perché invece una architettura "shared-disk," in cui ogni processore ha la propria memoria ma l'insieme dei dischi è condiviso, risulterebbe poco scalabile, nonostante l'ampia disponibilità di memoria centrale.

In una architettura shared-disk, la tabella dei lock e i buffer dovrebbero essere presenti su ogni nodo, con evidenti difficoltà di sincronizzazione

Per queste applicazioni vengono usati di solito sistemi che premettono di eseguire solo operazioni semplici e localizzate sui singoli nodi dell'architettura. Di solito, i dati venogno partizionati fra i vari nodi sulla base di un campo chiave (o pseudochiave selettiva) e ogni operazione fa riferimento a uno o pochi valori della chiave. Il criterio di distribuzione è noto e il suo utilizzo non richiede l'accesso ai singoli nodi (dato un valore della chiave si individua direttamente il nodo competente). Spiegare perché le operazioni di ricerca diventano efficienti, a patto di una distribuzione bilanciata dei dati e delle richieste fra i vari nodi. Spiegare in particolare perché è fondamentale la caratteristica sopra citata per il criterio di distribuzione.

Se il criterio di distribuzione è noto e può essere utilizzato dal front-end, allora solo un nodo è coinvolto in ciascuna operazione e se la distibuzione tanto dei dati quanto delle operazioni è uniforme, allora il carico si distribuisce di conseguenza.

Spiegare perché la localizzazione è fondamentale per le transazioni (cioè perché transazioni su più nodi sarebbero inefficienti).

Se le transazioni non sono locali, serve il commit a due fasi, che porta a significativi overhead di scambi di messaggi e attese di sincronizzazione.

Domanda 2 (30%) Si considerino un sistema con blocchi di dimensione $B = 1000$ byte e puntatori ai blocchi di $P = 2$ byte e una relazione *R(A,C,D)* di cardinalità pari circa a $T = 2.000.000$, con ennuple di $L = 20$ byte e campo chiave A di $L_A = 5$ byte e campo C di $L_C = 8$ byte. Il campo C non è chiave e ogni suo valore è presente, mediamente, in $e = 4$ ennuple. Valutare i pro e i contro relativamente alla presenza di un indice secondario sulla chiave A e di un altro, pure secondario, su C, in presenza del seguente carico applicativo:

1. inserimento di una nuova ennupla (con verifica del soddisfacimento del vincolo di chiave), con frequenza $f_1 = 20.000$
2. ricerca di una ennupla sulla base del valore della chiave A, con frequenza $f_2 = 10.000$
3. ricerca di ennuple sulla base del valore di C, con frequenza $f_3 = 10$

Ragionare in termini di costo degli accessi a memoria secondaria, assumendo disponibilità di buffer che permettano di mantenere stabilmente in memoria due livelli per ciascun indice e considerando che la relazione possa essere memorizzata in forma contigua (assumendo che il tempo di posizionamento della testina sia $r = 100$ volte maggiore del tempo di lettura). Trascurare le problematiche relative alla concorrenza e considerare il costo della lettura pari a quello della scrittura. Rispondere negli spazi sottostanti, in forma sia simbolica sia numerica.

Costo scansione sequenziale (SEQ) in multipli del tempo di posizionamento
numero di blocchi: NB $= \frac{T}{B/L} = 40.000$
numero di accessi: $1 + (\text{NB} - 1) \times \frac{1}{r} = \sim 400$

Numero livelli indice su A (PA)
il fattore di blocco dell'indice è 1.000/7, circa 160 e quindi, con un riempimento di circa il 60-70%, il fan-out è circa 100 e quindi i livelli necessari sono 4

Numero livelli indice su C (PC)
il fattore di blocco dell'indice è 1.000/10, circa 100 e quindi, con un riempimento di circa il 60-70%, il fan-out è circa 65 e quindi i livelli necessari sono pure 4

	nessun indice	indice solo su A	indice solo su C	indice su A e su C
Costo unit. Op. 1	SEQ = 400	PA - 2 + 1 + 1 = 4	SEQ + ... = $\sim$ 400	PA - 2 + PC - 2 + 1 + 1 = 6
Costo unit. Op. 2	SEQ = 400	PA - 2 + 1 = 3	SEQ = 400	PA - 2 + 1 = 3
Costo unit. Op. 3	SEQ = 400	SEQ = 400	PC - 2 + e = 6	PC - 2 + e = 6
Costo complessivo	ca 12.000.000	ca 110.000	ca 12.000.000	ca 150.000

Domanda 3 (25%)
Considerare i due seguenti scenari in ciascuno dei quali 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 ignorino le successive richieste della transazione che ha abortito (senza rilanciarla).

scenario 1

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

scenario 2

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 100. 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.

Scenario 1
READ COMMITTED

client 1		client 2	
read(x)	legge 100		
		read(x)	legge 100
x = x + 10	x vale 110		
write(x)	scrive 110		
commit			
		x = x + 20	x vale 120
		write(x)	scrive 120
		commit	

anomalia: perdita di aggiornamento

Scenario 2
SERIALIZABLE

client 1		client 2	
read(x)	legge 100		
		read(x)	legge 100
		x = x + 20	x vale 120
		write(x)	scrive 120
		commit	
read(x)	legge 100		
commit			

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

Domanda 4 (20%)
Come noto, esistono tecniche leggermente diverse per il recovery, che prevedono solo redo (**"redo-only"**), solo undo (**"undo-only"**) oppure tanto undo quanto redo (**"undo-redo"**). Le tecniche differiscono oltre che nella procedura di recovery, anche nella modalità di esecuzione delle scritture effettive su disco (operazioni di "flush" delle pagine di buffer), che possono essere le seguenti (con riferimento ad un utilizzo di checkpoint non quiescente):

flush-nel-commit le pagine modificate da una transazione vengono scritte su disco (subito) prima del commit della transazione stessa;

flush-all-nel-ckpt nel corso del checkpoint (subito prima della sua conclusione), vengono scritte su disco le pagine modificate da tutte le transazioni;

flush-committed-nel-ckpt nel corso del checkpoint (subito prima della sua conclusione), vengono scritte su disco le pagine modificate dalle transazioni andate in commit.

Nella tabella seguente, indicare per ciascuna tecnica quali approcci alla flush possono o debbono essere utilizzati (indicare il nome sintetico e fornire una breve spiegazione; si noti che potrebbe essere utilizzabile una sola tecnica oppure più di una; in questo secondo caso, indicare e spiegare quale sarebbe preferibile).

1. **"undo-only"**

flush-nel-commit, in quanto garantisce che non ci sia mai bisogno di redo (i dati sono su disco al momento del commit); non si può aspettare il checkpoint, perché se ci fosse un crash prima di esso allora servirebbero redo

2. **"undo-redo"**

flush-all-nel-ckpt: serve per garantire che le transazioni concluse al checkpoint non abbiano bisogno di essere ribadite; può portare ad undo per transazioni non concluse, ma è più semplice da gestire delle altre due, che pure sarebbero corrette

3. **"redo-only"**

flush-committed-nel-ckpt: serve a garantire che le transazioni andate in commit prima del checkpoint non siano da rifare; non scrive altro su disco e quindi non c'è necessità di undo. Non va bene la **flush-nel-commit** perché ci potrebbe essere un crash fra la flush e il commit e allora servirebbe l'undo. Analogamente, non va bene la **flush-all-nel-ckpt** perché si scriverebbero su disco dati non confermati da commit e quindi portebbero servire redo.

Basi di dati II, primo modulo 24 giugno 2011— Compito B
Cenni sulle soluzioni

Cognome _____ **Nome** _____ **Matricola** _____

Domanda 1 (25%) Come noto, si stanno diffondendo applicazioni nelle quali è necessaria una grande scalabilità e che vengono quindi realizzate con architetture parallele con centinaia o migliaia di processori. Queste architetture sono di solito di tipo "shared-nothing," cioè ogni processore ha la propria memoria centrale e i propri dischi. Sulla base delle conoscenze relative alla gestione dei lock e dei buffer, spiegare perché invece una architettura "shared-disk," in cui ogni processore ha la propria memoria ma l'insieme dei dischi è condiviso, risulterebbe poco scalabile, nonostante l'ampia disponibilità di memoria centrale.

In una architettura shared-disk, la tabella dei lock e i buffer dovrebbero essere presenti su ogni nodo, con evidenti difficoltà di sincronizzazione

Per queste applicazioni vengono usati di solito sistemi che premettono di eseguire solo operazioni semplici e localizzate sui singoli nodi dell'architettura. Di solito, i dati venogno partizionati fra i vari nodi sulla base di un campo chiave (o pseudochiave selettiva) e ogni operazione fa riferimento a uno o pochi valori della chiave. Il criterio di distribuzione è noto e il suo utilizzo non richiede l'accesso ai singoli nodi (dato un valore della chiave si individua direttamente il nodo competente). Spiegare perché le operazioni di ricerca diventano efficienti, a patto di una distribuzione bilanciata dei dati e delle richieste fra i vari nodi. Spiegare in particolare perché è fondamentale la caratteristica sopra citata per il criterio di distribuzione.

Se il criterio di distribuzione è noto e può essere utilizzato dal front-end, allora solo un nodo è coinvolto in ciascuna operazione e se la distibuzione tanto dei dati quanto delle operazioni è uniforme, allora il carico si distribuisce di conseguenza.

Spiegare perché la localizzazione è fondamentale per le transazioni (cioè perché transazioni su più nodi sarebbero inefficienti).

Se le transazioni non sono locali, serve il commit a due fasi, che porta a significativi overhead di scambi di messaggi e attese di sincronizzazione.

Domanda 2 (30%) Si considerino un sistema con blocchi di dimensione $B = 1000$ byte e puntatori ai blocchi di $P = 2$ byte e una relazione *T(A,C,D)* di cardinalità pari circa a $R = 2.000.000$, con ennuple di $L = 20$ byte e campo chiave A di $L_A = 5$ byte e campo C di $L_C = 8$ byte. Il campo C non è chiave e ogni suo valore è presente, mediamente, in $c = 4$ ennuple. Valutare i pro e i contro relativamente alla presenza di un indice secondario sulla chiave A e di un altro, pure secondario, su C, in presenza del seguente carico applicativo:

1. inserimento di una nuova ennupla (con verifica del soddisfacimento del vincolo di chiave), con frequenza $n_1 = 20.000$
2. ricerca di una ennupla sulla base del valore della chiave A, con frequenza $n_2 = 10.000$
3. ricerca di ennuple sulla base del valore di C, con frequenza $n_3 = 10$

Ragionare in termini di costo degli accessi a memoria secondaria, assumendo disponibilità di buffer che permettano di mantenere stabilmente in memoria due livelli per ciascun indice e considerando che la relazione possa essere memorizzata in forma contigua (assumendo che il tempo di posizionamento della testina sia $r = 100$ volte maggiore del tempo di lettura). Trascurare le problematiche relative alla concorrenza e considerare il costo della lettura pari a quello della scrittura. Rispondere negli spazi sottostanti, in forma sia simbolica sia numerica.

Costo scansione sequenziale (SEQ) in multipli del tempo di posizionamento
numero di blocchi: NB $= \frac{R}{B/L} = 40.000$
numero di accessi: $1 + (\text{NB} - 1) \times \frac{1}{r} = \sim 400$

Numero livelli indice su A (PA)
il fattore di blocco dell'indice è 1.000/7, circa 160 e quindi, con un riempimento di circa il 60-70%, il fan-out è circa 100 e quindi i livelli necessari sono 4

Numero livelli indice su C (PC)
il fattore di blocco dell'indice è 1.000/10, circa 100 e quindi, con un riempimento di circa il 60-70%, il fan-out è circa 65 e quindi i livelli necessari sono pure 4

	nessun indice	indice solo su A	indice solo su C	indice su A e su C
Costo unit. Op. 1	SEQ = 400	PA - 2 + 1 + 1 = 4	SEQ + ... = $\sim$ 400	PA - 2 + PC - 2 + 1 + 1 = 6
Costo unit. Op. 2	SEQ = 400	PA - 2 + 1 = 3	SEQ = 400	PA - 2 + 1 = 3
Costo unit. Op. 3	SEQ = 400	SEQ = 400	PC - 2 + c = 6	PC - 2 + c = 6
Costo complessivo	ca 12.000.000	ca 110.000	ca 12.000.000	ca 150.000

Domanda 3 (25%)
Considerare i due seguenti scenari in ciascuno dei quali 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 ignorino le successive richieste della transazione che ha abortito (senza rilanciarla).

scenario 1

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

scenario 2

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 100. 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.

Scenario 1
SERIALIZABLE

client 1		client 2	
read(x)	legge 100		
		read(x)	legge 100
x = x + 10	x vale 110		
write(x)	scrive 110		
commit			
		x = x + 20	x vale 120
		abort	

non c'è anomalia; l'abort è dovuto al fatto che x è stato modificato dopo l'avvio della transazione che sta scrivendo

Scenario 2
READ COMMITTED

client 1		client 2	
read(x)	legge 100		
		read(x)	legge 100
		x = x + 20	x vale 120
		write(x)	scrive 120
		commit	
read(x)	legge 100		
commit			

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

Domanda 4 (20%)
Come noto, esistono tecniche leggermente diverse per il recovery, che prevedono solo redo (**"redo-only"**), solo undo (**"undo-only"**) oppure tanto undo quanto redo (**"undo-redo"**). Le tecniche differiscono oltre che nella procedura di recovery, anche nella modalità di esecuzione delle scritture effettive su disco (operazioni di "flush" delle pagine di buffer), che possono essere le seguenti (con riferimento ad un utilizzo di checkpoint non quiescente):

flush-nel-commit le pagine modificate da una transazione vengono scritte su disco (subito) prima del commit della transazione stessa;

flush-all-nel-ckpt nel corso del checkpoint (subito prima della sua conclusione), vengono scritte su disco le pagine modificate da tutte le transazioni;

flush-committed-nel-ckpt nel corso del checkpoint (subito prima della sua conclusione), vengono scritte su disco le pagine modificate dalle transazioni andate in commit.

Nella tabella seguente, indicare per ciascuna tecnica quali approcci alla flush possono o debbono essere utilizzati (indicare il nome sintetico e fornire una breve spiegazione; si noti che potrebbe essere utilizzabile una sola tecnica oppure più di una; in questo secondo caso, indicare e spiegare quale sarebbe preferibile).

1. **"undo-redo"**

flush-all-nel-ckpt: serve per garantire che le transazioni concluse al checkpoint non abbiano bisogno di essere ribadite; può portare ad undo per transazioni non concluse, ma è più semplice da gestire delle altre due, che pure sarebbero corrette

2. **"redo-only"**

flush-committed-nel-ckpt: serve a garantire che le transazioni andate in commit prima del checkpoint non siano da rifare; non scrive altro su disco e quindi non c'è necessità di undo. Non va bene la **flush-nel-commit** perché ci potrebbe essere un crash fra la flush e il commit e allora servirebbe l'undo. Analogamente, non va bene la **flush-all-nel-ckpt** perché si scriverebbero su disco dati non confermati da commit e quindi portebbero servire redo.

3. **"undo-only"**

flush-nel-commit, in quanto garantisce che non ci sia mai bisogno di redo (i dati sono su disco al momento del commit); non si può aspettare il checkpoint, perché se ci fosse un crash prima di esso allora servirebbero redo