Basi di dati II
Esercizi di autovalutazione — 13 maggio 2013

Esercitazioni pratiche facoltative
I seguenti esercizi sono proposti ma non ne è richiesta la consegna, perché l'obiettivo è la comprensione dei concetti fondamentali, relativamente agli algoritmi di esecuzione delle interrogazioni, in particolare con riferimento all'utilizzo dei buffer e al rapporto fra pipelining e materializzazione.
Gli esercizi si svolgono realizzando, nel DBMS didattico SimpleDB, nuove classi che estendono le funzionalità di classi esistenti.

Esercizio 1 (Mergesort a più vie)
Come illustrato a lezione, l'ordinamento di un file può essere realizzato in un DBMS con un numero di accessi che può addirittura essere lineare rispetto alle dimensioni del file. In effetti, supponendo di avere k buffer disponibili, si può pensare di ordinare porzioni (chiamate *run*) del file lunghe k blocchi e poi fondere k porzioni alla volta. Ad esempio, con un file di 90 blocchi e 10 buffer disponibili, si potrebbero prima ordinare run lunghi 10 blocchi (e si avrebbero 9 run) e poi fare una fusione (*merge*) dei 9 run. Con un file di 900 blocchi e sempre 10 buffer disponibili, servirebbero due fasi di merge dopo l'ordinamento iniziale dei run. È importante osservare che l'ordinamento iniziale dei run e le fusioni diversa dall'ultima richiedono una memorizzazione del risultato intermedio, mentre l'ultima fusione può essere realizzata per mezzo di uno scan che produce le ennuple a mano a mano che servono, in pipelining.
SimpleDB, nella versione scaricabile, realizza un mergesort tradizionale a due vie (cioè in cui la fusione è fatta su due run alla volta) e partendo da run iniziali molto brevi (le porzioni già ordinate del file). Il tutto è realizzato dalle classi:

- `SortPlan`, implementazione dell'interfaccia `Plan`, che nel metodo `open()` esegue anche le operazioni che richiedono materializzazione e cioè la divisione in run e i passi di merge precedenti l'ultimo.
- `SortScan`, implementazione dell'interfaccia `Scan`, che, con i classici metodi `next()` e `getXXX()`, realizza l'ultimo passo di merge nel corso della scansione.

Si propone di scrivere due classi che, modificando le precedenti, realizzino un mergesort a più vie. Osservazioni:

- Una questione preliminare è l'individuazione del numero di buffer da utilizzare, che è opportuno sia una radice (quadrata, cubica, quarta, etc) del numero di blocchi del file e precisamente la più grande radice che sia minore del numero di buffer disponibili: ad esempio, con file di 8000 blocchi e 100 buffer disponibili, se ne utilizzerebbero 90 (arrotondamento della radice quadrata di 8000) mentre con 60 blocchi disponibili se ne utilizzerebbero 20 (radice cubica di 8000, perché la radice quadrata 90 è maggiore di 60). Per individuare tale numero, è disponibile il metodo `bestRoot()` della classe `BufferNeeds`.
- Il primo passo dell'algoritmo è la divisione del file in run. Come si può notare dalle classi esistenti, SimpleDB utilizza una tabella temporanea per ciascun run. Per realizzare correttamente l'algoritmo, si dovrebbero creare tabelle temporanee di k blocchi ciascuna. Una soluzione corretta ed efficiente di questo passo richiede una serie di artifici tecnici (legati agli altri moduli di SimpleDB) che sarebbero forse troppo onerosi e si suggerisce quindi di non realizzare questo passo, riutilizzando il metodo `splitIntoRuns()` della classe `SortPlan`.
- I passi di fusione vanno realizzati confrontando k run alla volta e vanno realizzati modificando quelli in `SortPlan` (che li gestisce tutti escluso l'ultimo) e `SortScan` (che gestisce l'ultimo).
- Si suggerisce di scrivere una classe di test nel package `materialize` che, dopo avere inizializzato le necessarie classi del server (ad esempio con `SimpleDB.init()`) utilizzi una tabella (creata separatamente o nel test stesso) e la ordini (aprendo uno scan su di essa e richiamando la `next()` fino alla fine). Per documentazione, stampare il numero di blocchi e il numero di record della tabella e il numero di run all'inizio e dopo ciascuna iterazione. Stampare anche il numero di buffer disponibili e il numero di quelli utilizzati (modificare opportunamente il numero di buffer del sistema).

Esercizio 2 (Hash join)
L'hash-join intuitivamente si realizza con un algoritmo con la seguente struttura (con k buffer disponibili):

0. Date due relazioni R_1 ed R_2, con R_2 più piccola (in termini di numero di blocchi)
1. Partizionare ciascuna delle relazioni in k porzioni dette *bucket* sulla base di una funzione hash sul campo di join
2. Per ogni bucket di R_2, caricarlo per intero in memoria (è necessario che abbia una dimensione non superiore a k blocchi) ed eseguire uno scan del corrispondente bucket di R_1 (cioè con lo stesso valore della funzione di hash) confrontando il record corrente dello scan con tutti i record del bucket di R_2 nei buffer

Il passo 1 richiede una materializzazione (in SimpleDB realizzabile con un array di k tabelle temporanee in cui si inserisce ogni record nel bucket di competenza), mentre il passo 2 si esegue in pipelining. Affinché la condizione del passo 2 sia soddisfatta, la relazione R_2 deve avere un numero di blocchi non superiore al quadrato di k (in effetti, un po' minore, perché i bucket non hanno tutti la stessa dimensione). Qualora la condizione non sia soddisfatta, si possono partizionare (ciascuno ulteriormente in k porzioni) i bucket troppo lunghi. Trascuriamo nel seguito questa eventualità.

Si propone di realizzare le classi SimpleDB per il plan e lo scan per l'hash join. Osservazioni:

- Per il calcolo del numero k di buffer necessari si può procedere come nel caso del mergesort, con il metodo `bestRoot()` della classe `BufferNeeds` (oppure, più semplicemente, supponendo che R_2 abbia un numero di blocchi non superiore al quadrato dei buffer disponibili, si calcola la radice quadrata del numero di blocchi di R_2)
- Si esegua il partizionamento con una scansione nell'ambito della quale il record corrente viene copiato (inserito) nella tabella temporanea (cioè nel bucket) corrispondente al valore della funzione hash (il valore della funzione deve essere compreso fra 0 e $k-1$; se i valori del campo di join sono interi casuali, si può senz'altro usare come funzione hash il resto della divisione per k)
- Per la realizzazione del confronto di cui al passo 2, si può utilizzare la classe `MultiBufferProductScan` (disponibile in SimpleDB) associata ad una `SelectScan`: per ogni i compreso fra 0 e $k-1$ si fa il prodotto cartesiano delle due tabelle temporanee associate (nei due partizionamenti) al valore i della funzione hash, seguito dalla selezione di uguaglianza sui campi di join.
- Anche qui è opportuno realizzare una classe di test che, avendo a disposizione le due tabelle apra ed esaurisca uno scan (di tipo hash join) su di esse.