Exam Exercises

PHYSICAL DATABASES

A tale of two Joins (*31 Jan 2012*)

STUDENT (Matricola, FirstName, LastName, BirthDate, …)

contains 150K students on 10K blocks in a primary hash over attribute Matricola.

EXAM (Matricola, CourseId, Date, Grade)

contains 2M tuples on 8K blocks in a entry-sequenced structure
we also have a secondary B+ index over Matricola, with fanout 200.

Estimate and compare the cost (in terms of number of disk accesses) of executing the query

```
select *
from Student S join Exam E on S.Matricola = E.Matricola
```

a) with a **hash join** in which Exam is re-structured in a suitable primary hash storage
b) with a **nested loop join** with Student as **external** table.
c) with a **nested loop join** with Student as **internal** table.

(Ignore collisions and caching)

STUDENT has 15 tuples per block, each hash-based access costs 1 i/o.

EXAM has 256 tuples per block, and has no hash-based access.

We need to build a hash structure (in order to ba able to use it!). This requires scanning the whole table, extracting each tuple and inserting it into the right «bucket». The number of buckets is of course the same, as the hash funcion is the same (e.g., *Matricola mod 10.000*): these buckets however will be rather «empty» w.r.t. the blocks in the previous representation (the storage grows overall from 8K to 10K blocks).

We therefore need to perform **8K i/o ops** in order to ***read*** and as much as **4M i/o ops to *update***, as the right bucket must be found for each exam, and the corresponding block needs to be read (moved to main mamory), updated with the new exam, and then re-written to disk.

The **hash-join** per se then costs **2 * 10K = 20K i/o**, to an overall cost of **4M + 28K i/o**

(*a remarkable cost, that should be payed off by the efficient future executions of the same join)*

Calcoliamo ora il costo del nested loop con ESAME come tabella **interna** (cioè nel caso in cui leggiamo un blocco della tabella STUDENTE (iterazione esterna) e per ogni studente cerchiamo in ESAME (iterazione interna) tutti gli esami sostenuti da quel particolare studente. Leggiamo un nuovo blocco di STUDENTE solo dopo aver esaurito tutti gli studenti nel blocco corrente).

L’albero ha profondità media pari a circa 3 ed è relativamente sparso (ha spazio per circa 8M valori di chiave). La **ricerca degli esami di un dato studente** costa quindi **3 accessi ai nodi dell’albero**, **più 1** accesso ai blocchi della struttura primaria **entry-sequenced** per recuperare tutti i campi dell’esame (trascuriamo il caso in cui i puntatori agli esami di uno stesso studente siano ripartiti su più di un nodo foglia dell’albero). In totale pagheremo 3 accessi per ogni studente e uno per ogni esame, cioè **3*150K + 2M accessi**.

Il caricamento di tutti i blocchi di STUDENTE nell**’iterazione esterna costa inoltre 10K** accessi, per cui in totale avremo = **2M + 460K accessi**.

È un costo inferiore a quello di costruirsi la struttura ad-hoc (poco più di metà), il quale però può ancora considerarsi un buon investimento se, conservando la struttura creata, velocizza poi sostanzialmente le esecuzioni dello stesso join (*20K contro 2,5M*).

Calcoliamo infine il costo del nested loop con ESAME come tabella **esterna** (che a prima vista sembra l'opzione più sensata, dato che gli accessi random basati su matricola alla struttura primaria a hash dello STUDENTE costano molto meno di quelli a ESAME basati sull'indice B+ (1 contro 4)).

Possiamo **scandire tutta la tabella ESAME pagando solo 8K** accessi per la sua lettura, sfruttando la struttura entry-sequenced (non avrebbe senso, infatti, scandire in sequenza le foglie dell'indice B+ e accedere ai blocchi primari "uno studente alla volta", dato che avere gli esami in ordine di matricola non ci aiuta), e dobbiamo poi **accedere 2M volte alla struttura hashed**, una volta per ogni esame, a recuperare i dati dello studente (più volte lo stesso studente, ogni volta che ci imbattiamo in un suo esame – ma stiamo trascurando gli effetti della cache), per un totale di **2M+8K accessi**.

Il costo è minore, ma comunque dello stesso ordine di grandezza di quelli precedentemente calcolati. Un ottimizzatore "miope" potrebbe continuare a scegliere questa ultima opzione senza mai "imbarcarsi" nell'impresa di costruire la struttura hash-based, che invece si rivela strategicamente la più conveniente se il join viene ripetuto frequentemente (come è probabile).

B+-primary and Hash-secondary structures (21 Feb. 2012)

A table STUDENT (Matricola,FirstName,LastName,BirthDate,…) has 200K tuples in a ***primary B+ tree*** on attribute *Matricola*, on 10K blocks with a maximum fanout of 100;

there is also a ***secondary hash index*** on *LastName* that takes 2K blocks (val(LastName) = 50K).

Estimate the cost of the following queries, ignoring collisions and caching:

1) select * from Student where Matricola = '623883'

2) select * from Student
where LastName in ('Braga','Campi','Comai','Paraboschi') **and** Matricola > '575478'

3) select * from Student where Lastname < 'B'

Ogni nodo/blocco *interno* all’albero contiene fino a 99 matricole e 100 puntatori fisici, e in ogni nodo/blocco *foglia* dell'albero ci sono invece 20 studenti (le intere tuple) e 1 solo puntatore fisico (al blocco successivo nella “catena delle foglie”).

Immaginando molto densa la struttura ad albero (nodi sostanzialmente tutti pieni e albero perfettamente bilanciato), oltre alla radice abbiamo un livello intermedio di 100 blocchi interni e 10K blocchi “foglia”, il che corrisponde alla dimensione indicata della struttura. L’albero avrà quindi una profondità pari a 3.

Nei blocchi dell'indice hash (la cui chiave di accesso non è una chiave primaria) ci sono puntatori fisici ai blocchi che contengono i dati di circa (in media) 200K / 2K = 100 puntatori fisici, corrispondenti a meno di 100 valori di chiave (sarebbero esattamente 100 se non esistessero studenti con lo stesso cognome, in realtà se ne avranno in media sensibilmente di meno). I blocchi avranno un fattore di riempimento che dipende dal rapporto di dimensione tra cognomi e puntatori fisici, che i dati non lasciano stimare.

Il costo è quindi

(**query 1**). 3 accessi tramite albero (2 blocchi interni e una foglia), e ho subito l'intera tupla cercata.

Il costo è quindi

(**query 1**). 3 accessi tramite albero (2 blocchi interni e una foglia), e ho subito l’intera tupla cercata.

(**query 2**). Se uso l’indice sul Cognome pago 4 accessi per il lookup nello hash e poi seguo 4 x 200K/50K = 4 x 4 = 16 puntatori, in totale 20 accessi.
avendo 4 condizioni in OR e 4 puntatori ai blocchi fisici da seguire.

È ragionevole ritenere che la condizione sulla matricola sia inservibile (non riteniamo probabile che esistano molto meno di 20x20 = 400 studenti con matricola superiore a 575478) e una strategia di interval-query fallimentare.

Il costo è quindi

(**query 1**). 3 accessi tramite albero (2 blocchi interni e una foglia), e ho subito l'intera tupla cercata.

(**query 2**). Se uso l'indice sul Cognome pago 4 accessi per il lookup nello hash e poi seguo 4 x 200K/50K = 4 x 4 = 16 puntatori, in totale 20 accessi.
avendo 4 condizioni in OR e 4 puntatori ai blocchi fisici da seguire.

È ragionevole ritenere che la condizione sulla matricola sia inservibile (non riteniamo probabile che esistano molto meno di 20x20 = 400 studenti con matricola superiore a 575478) e una strategia di interval-query fallimentare.

(**query 3**). Siccome in base alla traccia non possiamo garantire che la funzione di hash dia valori correlati all'ordinamento alfabetico, nessuna delle due strutture di accesso ci aiuta, e dobbiamo scandire l'intera tabella (10K accessi)

2015/09/30 - Actors and Roles

A table Role(Actor, Movie, Character) records 400K roles played by Hollywood actors in over many decades. Estimate the execution cost (under reasonable assumptions) of the following query in the scenarios listed below. Please briefly describe the considered query plan in each scenario.

```
select Actor, Movie, count(*) as NumberOfCharacters
from Role
group by Actor, Movie      // Extracts actors playing 3+ roles in the same movie
having count(*) > 2
```

1. The table is primarily stored in 16K blocks, with tuples in no particular order. There is also a hash based secondary index with Movie as key, with 5K buckets of 1 block each (val(Movie)=20K, val(Actor)=25K).
2. The table is primarily stored in 16K blocks, with tuples sequentially ordered according to the Movie attribute (as they are sequentially appended as soon as new movies are released), and there are no secondary access structures.
3. The table is primarily stored as in case 1, but the secondary structure, instead of being a hash, is a B+ tree with two attributes as key (Actor, Movie) – i.e., the key is composed of the two attributes, in this order. The tree has depth 3 (a root, an intermediate level, and 3.5K leaf nodes).

1. The hash-based index contains the pointers to the roles of each movie already grouped into the buckets.

Scanning this index allows to retrieve the roles of each movie all together, just by following an average of 20 pointers per movie (20 pointers = 400K tuples / 20K distinct movies).

For each of the 20K movies in the hash (of size 5K blocks) we therefore follow the (average) 20 pointers and process the count directly in memory:

The query plan is: scan the secondary index and retrieve all roles of each movie, counting the number of tuples of each distinct actor.

The execution cost is: 5K i/o to scan the index + 20K x 20 to follow all the pointers (and consider all tuples) = **405K**

2. Due to the ordering in the primary storage, in this case a sequential scan shows the tuples in Movie order (while in the previous case there was no particular ordering). Also, a block contains in average 400K/16K = 25 roles, and there are in average 400K/20K = 20 roles per Movie. Groups (as defined by the SQL query) are therefore confined to a few, adjacent blocks, and the query should be executable with a single scan, with marginal, if not negligible support by the caching system.

The query plan is: scan the primary storage once, and compute the count directly in memory

The execution cost is: **16K** to scan the table (+ possibly little overhead for movies with a very large number of roles).

3. *If we assume that the leaf nodes of the B+ contain* ***as many pointers to the blocks as the tuples of the primary storage***, then there is no need to retrieve the tuples!

The leaf nodes contain the pointers already “grouped” by specific values for actors and movies, and the count(*) can be computed by just counting the pointers. The query plan would be: just scan the leaf nodes of the B+ tree and count the pointers for each (*actor*,*movie*) pair. The execution cost is: **3.5 K** for the scan.

If, instead, the B+ is “optimized” so that, in case two tuples for the same actor and the same movie happen to be contained in the same block, the pointer is not repeated, then the previous approach is not applicable. All groups are potential contributors to the result (one block in the primary storage can contain up to 25 roles, potentially all of the same actor in the same movie!), and all pointers are to be followed. The cost is therefore again in the order of **400K**, as in scenario 1.

2015/09/07 - Possibly Pale Blue

A table T(PK, A, B, C, RefToIDofS) is primarily stored as entry-sequenced, with 40K tuples into 8K blocks. A much larger table S(ID, X, Y) contains 1M tuples in a primary hash-based storage, indexed by the primary key, with 100K buckets and very sparse, virtually free from overflow chains.

Knowing that PK<1000 for 2% of the tuples in T, that A is a unique attribute, and that val(B) = 125 (homogeneously distributed), estimate the execution cost of the query below, in the following three scenarios:

1. No secondary indexes are available

2. There is a B+ index with F = 200 for T, on the primary key

3. There is also another B+ index on attribute B, with depth 3 (a root, an intermediate level, and 1.25K leaf nodes).

```
select *
from S join T on RefToIDofS = ID
where PK  <  1000 or B = "pale blue" and A <> 13472
```

The result size is between

40K x 2% = 800 tuples

and

40K x 2% + 40K / 125 = 800+320 = 1.12K tuples

depending on the *correlation* between the values of A and B.

Attribute C is totally immaterial w.r.t. estimating the result size, as its contribution is at most to exclude 1 tuple.

In order to be conservative, we consider the “worst” case, i.e., that with the largest result size.

1. A pure nested loop is unreasonable, as table S allows for effective lookups based on the ID, and the join is performed on the ID.

We therefore adopt a “scan T and lookup in S” strategy. In this way, only a small part of S will be explored.

The query plan is: scan T and immediately apply the condition on the PK and A. Only for the matching tuples, perform a lookup onto S based on the value of their RefToIDofS attribute via the hash.

The execution cost is: 8K i/o to scan S + up to 1.12K lookups x 1 i/o = **9.12 K** i/o

2. If a B+ on PK is available, then the 800 tuples with a low value for PK are retrievable by accessing the root and the initial leaf nodes of the tree (overall 5 nodes = 1 + 800 / F) and following the pointers. This would cost 805 i/o.

However, this would give no information on the values of attribute B (which is in OR)… the only way to include all "pale blue" results (**with PK >= 1000**) is still to perform a full scan of T, with the previous plan, at the same cost (9.12 K i/o)

3. If we also have this second B+ index, we can lookup both attributes PK and B on the respective structures. The tree on B has 1.25K leaf nodes for 125 colors → 10 blocks per color

The query plan is: read the initial leaf nodes on the B+(PK) and follow the pointers, lookup “pale blue” in the other B+ and follow the related pointers, *eliminate the possible duplicates*, and lookup onto S. No duplicate elimination was required previously, as each tuple of T was encountered just once in the scan, while now a tuple with low PK and of pale blue color would be extracted twice.

Execution cost: 1+4+800 for the PK part + 1+10+320 for the B part + up to 1.12K lookups x 1 i/o = **2.26K**

2015/06/30 - Philantropic Society of Trees

A philanthropic society plants trees all over the world.

Table TREE(Id, PlantingDate, ZipCode, Species, GeoRef) stores 55K tuples in a primary structure sequentially-ordered by PlantingDate; its size is 2.75K blocks.

A table MEMBER(SSN, BirthDate, City, ZipCode, Name, Email) uses 11K blocks to store 75K tuples in a primary hash-based structure, with a function h() that maps the ZipCode onto 6K buckets (there are non-negligible overflow chains, and the average number of i/o operations per access is 1.83). Knowing that 80% of trees are in areas (zipcode) were some members are also located, and that 40% of members were born on a day in which some trees were planted, estimate the execution cost of the query below in three scenarios: (i) no indices are available; (ii) the only secondary index is a hash structure that uses h() on the ZipCode for table TREE (7.2 K blocks, 1.2 i/o operations per access in average); (iii) the only secondary index is a B+ tree for MEMBER with BirthDate as key, with depth 3 and 4K leaf nodes.

```
select *              // Matches people with trees planted on their birthdate in hometown
from Member M, Tree T
where BirthDate = PlantingDate and M.ZipCode = T.ZipCode
```

(i) A pure nested loop approach is inconvenient and unreasonable (order of M i/os), as there is a primary hash that supports direct lookup of Members based on the Zipcode.

The query plan is: a sequential scan of TREE followed by a lookup on MEMBER for each tree

The execution cost is: 2.75K (scan TREE) + 55K (tuples) x 1.83 (i/os per lookup) = **103.4 K** i/o

(ii) A Hash Join on ZipCode in possible between the secondary index and the primary representation. Of course not all trees will be retrieved, but only those (80%) that satisfy the "zipcode" part of the join predicate

The query plan is: scan the two hash structures and join the zipcodes bucket-wise, retrieve the matching members and check the second part of the join predicate

The execution cost is: 11 K (scan prim. hash on MEMBER) + 7.2 K (scan sec. hash on TREE) + 80% x 55 K (pointers) = **62.2 K** i/o

(iii) A Merge-Scan on Dates is the most promising option in this scenario, scanning in parallel the primary representation of Tree and the leaf nodes of the B+. As only 40% of the members will satisfy the Date part of the join predicate, the pointers to the full storage will be followed only for those that qualify.

The query plan is: merge-scan the two ordered structures and retrieve 40% of the tuples of MEMBER

The execution cost is: 2.75K (scan Trees) + 4K (scan leaf nodes) + 40% x 75K (pointers) = **36.75 K** i/o