CS 586 Introduction to Databases
Assignment 6 – Disks, Indexes, Operators
He, Haomin
11/15/2017

Part I Disk Access
Consider a hard disk with the following characteristics:
Average seek time: 5ms
Average rotational delay: 3ms
Page transfer time: 0.04ms
Assume the disk has one platter with two surfaces

Question 1. (5 points) Estimate the time to read a list of 150 pages from disk, where the pages are randomly distributed across the disk.
– seek time (moving arms to position disk head on track)
– rotational delay (waiting for block to rotate under head)
– transfer time (actually moving data to/from disk surface)
Answer: 150*(5+3+0.04) = 1206ms

Question 2. (5 points) Estimate the time to read a list of 150 pages from disk, where the pages are randomly distributed in tracks of a single cylinder.
Answer: 5+150*(3+0.04) = 461ms

Question 3. (5 points) Estimate the time to read a list of 150 pages from disk, where the pages are arranged consecutively in tracks of a single cylinder.
Answer: 5+3+150*0.04 = 14ms

Question 4. (5 point) Suppose we don't care about the order in which the 150 pages are read. Suggest how to speed up the times for Questions 1 and 2 above.
Answer: Make all the pages sequential or consecutive, and pages are sequentially distributed in tracks of a single cylinder, in order to reduce seek time and rotational delay. Query cost is often measured in the number of page I/Os. Key to lower I/O cost: reduce seek/rotation delays. Random I/O is more expensive than Sequential I/O.

Part II: Evaluating Queries with Just Indexes
It is sometimes possible to evaluate a particular query using only indexes, without accessing the actual data records.
Consider a database with two tables:
Pokemon(charName, attack, stamina, cType)
Captured(charName, player, difficulty)
Assume three unclustered indexes, where the leaf entries have the form

[search-key value, RID].
<stamina> on Pokemon
<cType> on Pokemon
<difficulty, charName> on Captured
For Questions 5-12, say which queries can be evaluated with just data from these indexes.
- If the query can, describe how.
- If the query can't, explain why.

Question 5:
SELECT MAX(difficulty)
FROM Captured;
Answer: Yes, we can use the indexes to evaluate this query without accessing any actual data stored in Captured table and using the index <difficulty, charName> on Captured.
The main reason is that we can utilize the <difficulty, charName> index to find the maximum difficulty over all characters and players from Captured. Because of the unclustered index for <difficulty, charName>, each search key in the <difficulty, charName> index must be associated with a data record in files. In meanwhile, all of the difficulty's entries are found according to the search-key value, so the maximum difficulty value can be retrieved in this query. (Note: we do not actually look up any record when evaluating this query.)

Question 6:
SELECT AVERAGE(difficulty)
FROM Captured
GROUP BY player;
Answer: No, we cannot use the indexes to evaluate this query without accessing any actual data stored in Captured table. There is a ‘group by’ operation on player, so it has to look through rows in Captured and group players who have the same name. Index <difficulty, charName> on Captured cannot be used.

Question 7:
SELECT charName, MIN(difficulty)
FROM Captured
GROUP BY charName;
Answer: Yes, we can use the indexes to evaluate this query without accessing any actual data stored in Captured table and using the index <difficulty, charName> on Captured. The ‘group by’ operation on charName and finding minimum on difficulty can be easily done because of the unclustered index for <difficulty, charName>. Each search key in the <difficulty, charName> index must be associated with a data record in files. In meanwhile, all of the charName’s and difficulty's entries are found according to the search-key value, so the minimum difficulty value and charName group by value can be retrieved in this query. (Note: we do not actually look up any record when evaluating this query.)

Question 8:
SELECT COUNT(*)
FROM Pokemon
WHERE stamina = 85 AND cType = 'Fire';

Answer: No, we cannot use the indexes to evaluate this query without accessing any actual data stored in Pokemon table and using the index <stamina> and <cType> on Pokemon. We still need to compare/look up record values based on stamina entries and cType entries are found according to the search-key value.

Question 9:
SELECT cType, COUNT(charName)
FROM Pokemon
GROUP BY cType;
Answer: Yes, we can use the indexes to evaluate this query without accessing any actual data stored in Pokemon table and using the index <cType> on Pokemon. The 'group by' operation on cType and finding count on charName can be easily done because of the unclustered index for <cType>. Each search key in the <cType> index must be associated with a data record in files. In meanwhile, all of the cType's entries are found according to the search-key value, so we just need to count how many charName in each cType group, and these information can be retrieved in this query. (Note: we do not actually look up any record when evaluating this query.)

Question 10
SELECT cType, difficulty
FROM Pokemon, Captured
WHERE stamina = 90 AND Pokemon.charName = Captured.charName;
Answer: No, we cannot use the indexes to evaluate this query without accessing any actual data stored in Pokemon and Captured table. Even though we can allocate value 'stamina = 90' very fast based on index <stamina> on Pokemon, but we still need to evaluate equation 'Pokemon.charName = Captured.charName', and make sure this condition is satisfied.

Question 11:
SELECT COUNT(DISTINCT stamina)
FROM Pokemon
WHERE cType = 'Fire';
Answer: No, we cannot use the indexes to evaluate this query without accessing any actual data stored in Pokemon table. Even though we can allocate value 'cType = 'Fire'' very fast based on index <cType> on Pokemon, but we still need to make sure stamina values are unique by using 'distinct'.

Question 12: (think carefully about this one!):
SELECT charName, AVERAGE(difficulty)
FROM Captured
GROUP BY charName
HAVING COUNT(DISTINCT player) > 1;
Answer: Yes, we can use the indexes to evaluate this query without accessing any actual data stored in Captured table and using the index <difficulty, charName> on Captured. The primary key in Captured is 'charName, player'. So when we use 'group by' on charName, it can be sure that player values uniquely exist. The 'group by' operation on charName, finding count on distinct player and finding average on difficulty can be easily done because of the unclustered index for <difficulty, charName>. Each search key in the <difficulty, charName> index must be associated with a data record in files. In meanwhile, all

of the charName's, player's and difficulty's entries are found according to the search-key value, so the average difficulty value and charName group by value can be retrieved in this query. (Note: we do not actually look up any record when evaluating this query.)

Part III: Operator Implementation
We want to compute Captured RIGHT OUTER JOIN Pokemon ON charName

Question 13 (5 points): Consider computing this join using a simple hash-join algorithm when Captured will fit in memory. That is, it will be the inner input, which is inserted into a hash table. Explain how to adapt the hash-join algorithm to produce the additional rows needed for the outer join.
Answer: Simple hash-join algorithm, Captured fits in memory, build an in-memory hash index for Captured, proceed as for index nested-loops join.

```
foreach row pr in Pokemon do
        foreach row cr in Captured do
                if Pokemon.charName == Captured.charName do
                        add <pr, cr> to result
                else
                        add <pr, empty cr value> to result
```

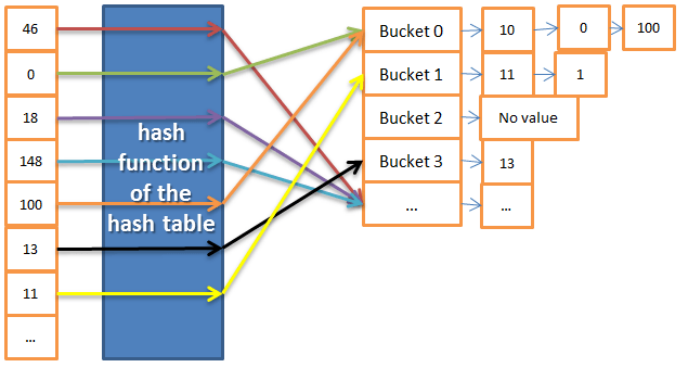
Build phase: For each row in Captured, calculate hash value of equi join columns. Store row in a hash table, use calculated hash as key.
Probe Phase: For each row in Pokemon, calculate hash value of equi join columns. Check if hash match in hash table, if so check actual columns -> output hash match. If hash doesn't match in hash table -> output < Pokemon value, empty Captured value>.

Question 14 (10 points): Now suppose that Pokemon will fit in memory, but Captured won't (so Pokemon will be the inner, hashed input). Explain how to adapt the hash join to produce the additional rows. Be sure to describe any additional information that the algorithm must maintain.
Answer: Simple hash-join algorithm, Pokemon fits in memory, build an in-memory hash index for Pokemon, proceed as for index nested-loops join.

```
foreach row cr in Captured do
        foreach row pr in Pokemon do
                if Pokemon.charName == Captured.charName do
                        add <cr, pr> to result
                else
                        add <cr, empty pr value> to result
```

Build phase: For each row in Pokemon, calculate hash value of equi join columns. Store row in a hash table, use calculated hash as key.
Probe Phase: For each row in Captured, calculate hash value of equi join columns. Check if hash match in hash table, if so check actual columns -> output hash match. If hash doesn't match in hash table -> output < Captured value, empty Pokemon value>.

Hash Join



Outer relation Inner relation (in-memory hash table)

References: lecture slides, Piazza, Youtube, ask classmates questions (Zicheng Ren).