COMP7640 @ HKBU: Tutorial 7
Query Processing

Wei Wang
weiw AT cse.unsw.edu.au

School of Computer Science & Engineering
University of New South Wales

December 5, 2014
Indexes can be classified according to several dimensions:

- Clustered/Non-clustered
- Primary/Secondary
- Sparse/Dense

Others:

- indexed attribute(s) is/are a candidate key or not
- tree-based or hash-based

Not all the combinations are valid.

- E.g., sparse index must be clustered, but not every clustered index is sparse.

Also need to realize the impact of different kind of indexes when it comes to query processing. E.g., usually a bad idea to use a non-clustered index to answer range queries with large selectivities.

Advanced

What is the essence of an index?
Indexes can be classified according to several dimensions:
- Clustered/Non-clustered
- Primary/Secondary
- Sparse/Dense

Others:
- indexed attribute(s) is/are a candidate key or not
- tree-based or hash-based

Not all the combinations are valid.
- E.g., sparse index must be clustered, but not every clustered index is sparse.

Also need to realize the impact of different kind of indexes when it comes to query processing. E.g., usually a bad idea to use a non-clustered index to answer range queries with large selectivities.

Advanced

What is the essence of an index? ⇒ Some data structure that is
- able to provide a superset of answers for a given type of query, and/or
- smaller than the data
External Sorting

Must know

- **Algorithm:**
  1. (Phase I) Run-generation phase
  2. (Phase II) Merging Phase (possibly need multiple passes)

- **Buffer usage**
  1. (Phase I) 0 output buffer page
  2. (Phase II) 1 output buffer page

- **Cost:** \( M \cdot (2 \cdot (1 + k) - 1) \), where \( k = \lceil \log_{B-1} \left( \frac{M}{B} \right) \rceil \)

  1. -1 because the cost of writing out the final result is uniformly ignored here.
  2. However, when computing the cost of naive sort-merge join (i.e., not the hybrid sort-merge join), the cost should include writing out the final result, i.e., \( M \cdot 2 \cdot (1 + k) \).
Query Processing Using Indices I

Consider the relations:

Student(id, name, age, course)
Subject(code, title, description)

which contain 50,000 and 5,000 records respectively and are stored in files sorted on id and code respectively. Imagine that we are using simple flat indexes (i.e., one-level index). Assume that record pointers are 4-bytes and that data pages and index pages are all 4096 bytes.

1. Although you may build a dense index on any field, you cannot do so for sparse indexes. On which field(s) could you build a sparse index?

2. If the student id is a 4-byte quantity, how large would a dense index on Student.id be?

3. If the subject code is an 8-byte quantity, how large would a dense index on Subject.code be?
If the *blocking factor* for Student is 100 and for Subject is 20, and other values are as above, how large would sparse indexes on Student.id and Subject.code be?

If you had just dense indexes on Student.id and Subject.code, briefly describe efficient processing strategies for the following queries and their costs.

1. SELECT name FROM Student WHERE id=2233445
2. SELECT title FROM Subject WHERE code >= 'COMP1000' AND code <= 'COMP2000'
3. SELECT id, name FROM Student WHERE age=18
4. SELECT code, title FROM Subject WHERE title LIKE 'Comput%'

What techniques could you use to improve the performance of each of the above queries? And how would it impact the other queries?
You can only build sparse indexes on a field on which the relation is sorted (i.e., `Student.id` and `Subject.code`), since a sparse index has one entry per block, and assumes that all records in that block have key values less than all key values in the next block.

The index entry size is $4 + 4 = 8$, and a page can fit $\left\lfloor \frac{4096}{8} \right\rfloor = 512$ index entries. The number of index entries is equal to the number of tuples, i.e., 50000. The number of index pages is then $\left\lceil \frac{50000}{512} \right\rceil = 98$.

Similar to above. A page can fit $\left\lfloor \frac{4096}{12} \right\rfloor = 341$, and the number of index page is then $\left\lceil \frac{5000}{341} \right\rceil = 15$.

For a sparse index, we need only one index entry per block of the data file. The `Student` file has $\left\lceil \frac{500000}{100} \right\rceil = 500$ pages, and the `Subject` file has $\left\lceil \frac{5000}{20} \right\rceil = 250$ pages. The sizes of the index entries have been calculated before, thus index pages for `Student` = $\left\lceil \frac{500}{512} \right\rceil = 1$, and index pages for `Subject` = $\left\lceil \frac{50}{341} \right\rceil = 1$.

```
SELECT name FROM Student WHERE id=2233445
```
Solution II

Use Do a binary search through the index (worst case $\log_2 98 = 7$ page reads) to find the entry for 2233445 and then fetch the data block containing that record (if it exists). The cost is $7 + 1 = 8$ page I/Os. Note that this is essentially A3 method in Chap 13.

2. SELECT title FROM Subject WHERE code >= 'COMP1000' AND code <= 'COMP2000'

Do a binary search through the index (worst case $\log_2 10 = 6$ page reads) to find the first index entry that points to a record with code $\geq$ 'COMP1000', then follow the pointer and do a scan through the data file (till the first record that code > 'COMP2000'. The cost is $6 + x$ page I/Os, where $x$ is the number of pages the result occupies in the data file. Note that this is essentially A6 method in Chap 13.

3. SELECT id, name FROM Student WHERE age=18

Since the index provides no assistance, the best solution is to scan the entire data file (as age is not a candidate key) and select the 18-year-olds as you go. Sorting the file doesn’t help here. The cost is 500 page reads. Note that this is essentially A1 method in Chap 13.

4. SELECT code, title FROM Subject WHERE title LIKE 'Comput%'
Once again, the index doesn’t help, so a linear scan is the only option. The cost is $\lceil \frac{5000}{20} \rceil = 250$ page reads. Note that this is essentially A1 method in Chap 13.

What techniques could you use to improve the performance of each of the above queries? And how would it impact the other queries?

1. SELECT name FROM Student WHERE id=2233445
   You could do better than the above by using either hashing (which would require only 1.2 page access on average) or a B+-tree index, which usually require two to four page I/Os. Hashing would not be an option if there was some reason why the file had to be maintained in order on the student id.

2. SELECT title FROM Subject WHERE code >= 'COMP1000' AND code <= 'COMP2000'
   A B+-tree index on the code field would also help here.

3. SELECT id, name FROM Student WHERE age=18
   If this was a common kind of query (lookup by specific age), and if there was no requirement to keep the file in any particular order, you could organized the data file as a hash file based on age to improve performance. A more likely scenario would be to add a dense index on the age field.
SELECT code, title FROM Subject WHERE title LIKE 'Comput%'

Building a $B^+$-tree index on title will help, as the selection condition is essentially a range predicate.
Computing the Cost of External Sorting

You have an unsorted heap file containing 4500 records and a select query is asked that requires the file to be sorted. The DBMS uses an external merge-sort that makes efficient use of the available buffer space. Assume that: records are 48-bytes long (including a 4-byte sort key); the page size is 512-bytes; each page has 12 bytes of control information in it; 4 buffer pages are available.

1. How many sorted subfiles will there be after the initial pass of the sort algorithm? How long will each subfile be?

2. How many passes (including the initial pass considered above) will be required to sort this file?

3. What will be the total I/O cost for sorting this file (including writing out the final result)?

4. What is the largest file, in terms of the number of records, that you can sort with just 4 buffer pages in 2 passes? How would your answer change if you had 257 buffer pages?
Solution I

1. We need to first of all work out the how many records can be held within a page (also known as the "blocking factor") and then the number of data blocks. The available space for storing records in each page is $512 - 12 = 500$ bytes, so each page can store up to 10 48-byte records. This means that 450 pages are needed to store the 4500 records. Given that 4 buffer pages are available, there will be $\lceil \frac{450}{4} \rceil = 113$ sorted runs (aka., "sub-files") of 4 pages each, except the last run which is only 2 pages long.

2. Method is given in previous question: $\lceil \log_{4-1} 113 \rceil + 1 = 6$ passes.

3. Total cost for sorting this file is $(2 \times 6 - 1) \times 450 = 4,950$ I/Os (note that we do not include the cost of writing out the final result).
We first need to work out the answer in terms of number of pages \(b\), and then convert to number of records.

In pass 0, \(\lceil \frac{b}{B} \rceil\) runs are produced. In pass 1, we must have enough buffers to enable us to merge this many runs, so \(B - 1\) must be larger or equal to \(\lceil \frac{b}{B} \rceil\).

When \(B\) is given to be 4, we get \(b = 12\). The maximum number of records on 12 pages is \(12 \times 10 = 120\).

When \(B\) is given as 257, we get \(b = 65792\), with \(65792 \times 10 = 657920\) records.