L07: Introduction to Indexing

Dr. Kenneth LEUNG

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China
• Assume that you work in a government office, and you maintain the records of 10 million HK residents
• The record of each resident contains the HK ID, address, telephone, etc.
• People come to your office, and ask you to retrieve the records of persons given their HK ID (i.e., “show me the record of person with HK ID: 5634569”)
• Let’s forget computers for now. You just want to print the records in a catalog, so you answer these queries by manually looking up the catalog
  – Assuming that you can put 10 records per printed page, the catalog will be 1 million pages
How would you arrange the records in the catalog?
   - Your goal is to minimize the effort of finding records
   - We measure this cost as the number of pages you have to "open" before finding the record

**Solution 1: random order**
   - If the catalog contains records in random order of HK ID, then in the worst case you have to search the entire catalog (cost = $10^6$) before finding a record, or to determine that the HK ID does not exist in the catalog

**Solution 2: records sorted on HK ID**
   - You can apply binary search with cost $\lceil \log_2 10^6 \rceil = 20$

Same discussion applies when we use computers; instead of the printed pages, we have disk pages (e.g., size 8 KB)
   - Every time we read something from the disk, we need to bring an entire page in main memory. The major cost is how many pages we read because disk operations are much more expensive than CPU operations

Can you make it even faster?
Let’s keep the sorted file and build an additional index (e.g., at the beginning of the catalog)

- Each index entry is a small record, that contains a HK ID and the page where you can find this ID (e.g., \( <5634569, 259> \) means that HK ID 5634569 is on page 259 of the catalog)
- HK ID is called the search key of the index
- Since each index entry is much smaller than the record, let’s assume that we can fit 100 entries per page
- The index entries are also sorted on HK ID

Do we need an entry for each of the 10,000,000 records?

- No, we only need an entry for the first record of each page
  - Example: If I have two consecutive entries \( <5634569, 259>, <5700000, 260> \) in the index, then I know that every HK ID between 5634569 and 5700000 must be on page 259
- Therefore, we need only 1,000,000 index entries (one for each page of the main catalog)
• Given that I can fit 100 entries per page, and I have 1,000,000 entries, my index is 10,000 pages.

• How I can use the index to speed up search for a record?
  - Use binary search on the index to find the largest HK ID that is smaller or equal to input HK ID. The cost is $\lceil \log_2 10^4 \rceil = 14$
  - Then, follow the pointer from that entry to the actual catalog (cost 1)
  - Total cost: $14 + 1 = 15$
• Can I drop the cost even further?

• Yes: Build an index on the index
  - The second level index contains 10,000 entries (one for each page of the first index) in 100 pages
  - Use binary search on the second level index to find the largest HK ID that is smaller or equal to input HK ID. The cost is \( \lceil \log_2 10^2 \rceil = 7 \)
  - Then, follow the pointer from that entry to first level index and finally to the actual catalog (cost 2)
  - Total cost: 7 + 2 = 9

• Finally, build a third level index containing 100 entries, one for each page of the second level index
  - These entries fit in one page. Read this page - find the largest HK ID that is smaller or equal to input HK ID and follow the pointers to second level index, first level index and file
  - Total cost: 4
• Indexes speed up access to desired data

• **Search Key:** attribute or set of attributes used to look up records in a file
  - Not to be confused with the concept primary or candidate key
  - In the previous slides the search key was HKID - we can find records given the HKID
  - If we want to find records given the name (or another attribute) we need to build additional indexes

• An **index file** consists of records (called **index entries**) of the form `<search key, pointer>`
  - Index files are typically **much smaller** than the original file (they skip most attributes)
The index that we built is an **ordered index** (also called **tree**)
- Because the index entries are sorted on the search key (e.g., HKID)
- Good for equality and range search
- There are other types of indexes (e.g., hash indexes, bitmaps)

When looking for a record, we always start from the root, and follow a single path to the leaf that contains the search key of the record. Then, we perform an additional access to get the record from the data file
- The cost, in terms of page accesses, is the height of the tree (number of levels) plus 1

An index page is also called index **node**

The number of children (pointers) of a node is called the **fanout**
- In our example, the fanout is 100

The height of the tree is \[ \left\lfloor \log_{\text{fanout}}(\#\text{index entries}) \right\rfloor \]
- In our example, the height is \[ \left\lfloor \log_{100}(10^6) \right\rfloor \]
Primary vs. Secondary Indexes

- **Primary index** (also called *clustering index*): when the file is sorted on search key of the index (e.g., our index on HKID)
  - **Index-sequential file**: ordered sequential file with a primary index (also called ISAM - indexed sequential access method)

- **Secondary index** (also called *non-clustering index*): when the file is not sorted on search key of the index
Sparse vs. Dense Indexes

• **Sparse Index**: contains index records for only some search-key values.
  - Only applicable to primary indexes
  - In our HKID index, we only have index entries for the first record in each page of the file
  - In general, we have an index entry for every file page, corresponding to the minimum search-key value in the page.
  - To locate a record with search-key value $K$ we:
    • Find index record with largest search-key value $\leq K$
    • Search file sequentially starting at the record to which the index record points
  - Less space and less maintenance overhead for insertions and deletions

• **Dense Index**: has an entry for every search key value
Continuing the previous example, assume again the file with the 10 million records and the primary index on HKID.

In addition to HKID, you want to be able to find records given the name:
- Somebody gives you the name of a person and wants his/her record. How do we find the record fast?
- Answer: build another index on the name.
- Since the file is sorted on the HKID, the new index must be secondary and dense.
- Assuming that all names are distinct, your index will contain 10 million entries.
- Assuming that the fanout is again 100, cost of finding a record given the name is \( \log_{100}(10^7) \).

A secondary index is almost as good as a primary index (in terms of cost) when retrieving a single record:
- However, it may be very expensive when retrieving many records (e.g., for range queries). More on this in subsequent classes.
Assume that we want to build an index on the name, but there may be several people with the same name.

If the index is primary and sparse, this is not a problem. Otherwise, there are three options.

**Option 1:** use variable length index entries
- Each entry contains a name, and pointers to all records with this name
- Example: \(<\text{Qiong Luo}, \text{pnt1}, \text{pnt2}, \ldots, \text{pntn}>\)
- **Problem:** complicated implementation as it needs a file organization that supports records of variable length

**Option 2:** use multiple index entries per name
- There is an entry for every person, if he/she shares the same name with other people
- **Example:** \(<\text{Qiong Luo}, \text{pnt1}>\), \(<\text{Qiong Luo}, \text{pnt2}>\), \ldots, \(<\text{Qiong Luo}, \text{pntn}>\)
- **Problem:** Redundancy - you repeat the name many times

**Option 3:** use an extra level of indirection (most common option)
- Index entry points to a bucket that contains pointers to all the actual records with that particular name
Secondary Index on *balance* field of Account
• Hashing can be used not only for file organization, but also for index-structure creation
• A **hash index** organizes the search-keys, with their associated record pointers, into a hash file structure
• Strictly speaking, hash indices are always **secondary indices**
  - **Note:** if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary
• The version that we will discuss is for relatively **static** datasets
  - We want to build a hash index for an existing dataset - we expect the number of records not to change too much
Example of Hash Index

bucket 0

bucket 1
A-215
A-305

bucket 2
A-101
A-110

bucket 3
A-217
A-102
A-201

bucket 4
A-218

bucket 5

bucket 6
A-222

A-217 Brighton 750
A-101 Downtown 500
A-110 Downtown 600
A-215 Mianus 700
A-102 Perryridge 400
A-201 Perryridge 900
A-218 Perryridge 700
A-222 Redwood 700
A-305 Round Hill 350
• In the **worst case**, the hash function maps all search-key values to the **same bucket**; this makes access time proportional to the number of search-key values in the file.

• An **ideal hash function** is **random**, so each bucket will have the same number of records assigned to it irrespectively of the *actual distribution* of search-key values in the file.

• Typical hash functions perform computation on the internal binary representation of the search-key
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned.
Bucket overflow can occur because of
- Insufficient number of buckets
- Skew in distribution of records. This happens because of two reasons:
  - multiple records have the same search-key value
  - chosen hash function produces non-uniform distribution of key values

Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using overflow buckets
- Overflow chaining: overflow buckets are chained together in a linked list. Long chains degrade performance because a query has to read all buckets in the chain.
• **Access types supported efficiently**
  - E.g., hash indexes are in general good for equality selection queries, but do not support ranges
  - Secondary ordered indexes are not good for queries that retrieve many records

• **Update time**
  - When a file is modified, every index on the file must be updated

• **Space overhead**
  - The index should be in general much smaller than the data file