COMP 3311
DATABASE MANAGEMENT SYSTEMS

LECTURE 11
INDEXING: INTRODUCTION

INDEXING: OUTLINE

Indexing Basic Concepts

Ordered Index

- Dense vs. Sparse
- Clustering vs. Non-clustering

B$^+$-tree Index

Hash Index

- Static Hashing
- Dynamic Hashing

Bitmap Index

INDEXING BASIC CONCEPTS

1. Assume that you work in a government office and you maintain the records of 8 million Hong Kong residents.

2. The record of each resident contains the hkid, address, phone number, etc.

3. People come to your office and ask you to retrieve the records of persons given their hkid, for example:

   Find me the record of the person with hkid A634569.

4. Let's forget about computers for now. You just want to keep the records in a paper-based catalog, so you answer these queries by manually looking up the catalog.
   - Assuming you can put 8 records per printed page, the catalog will be 1 million pages!

Records: 8,000,000
Records/page: 8
Pages: 1,000,000

INDEXING BASIC CONCEPTS (CONT'D)

How would you arrange the records in the catalog?

- Your goal is to minimize the cost (i.e., effort) of finding records.
- We measure this cost as the number of pages you have to "access" before finding the record.

Solution 1: Random order

If the catalog records are in random order of hkid, then in the worst case you must search the entire catalog (cost = 1,000,000 page accesses) before finding a record, or to determine that the hkid does not exist in the catalog. **What would be the average case page access cost?**

Solution 2: Records ordered on hkid

What would be the average case page access cost?

Records: 8,000,000
Records/page: 8
Pages: 1,000,000

INDEXING BASIC CONCEPTS (CONT'D)

How would you arrange the records in the catalog?

- Your goal is to minimize the cost (i.e., effort) of finding records.
- We measure this cost as the number of pages you have to "access" before finding the record.

- The same considerations apply when we use computers; instead of paper pages, we have disk pages of a fixed size.
  - Every time we read something from the disk (i.e., do a page I/O), we need to bring an entire page into main memory.
  - The major cost is how many pages we read because disk operations are much more expensive than CPU operations.

☞ **Can we reduce the cost even more?**

Records: 8,000,000
Records/page: 8
Pages: 1,000,000

INDEXING BASIC CONCEPTS (CONT'D)

- Continuing with our catalog example, let's keep the ordered file, but also build an additional index (e.g., at the front of the catalog).
  - Each index entry is a small record, that contains a hkid and the page where you can find this hkid.
  - For example, <A634569, 259> means that hkid A634569 is on page 259 of the catalog.

    ☞ hkid is the search key of the index.

    Recall: A search key is not the same as a primary key or a candidate key!
  - Each index entry is much smaller than the actual record.
  - Let's assume that we can fit 100 index entries per paper page.

☞ **The index entries are also ordered on hkid.**

index entries/page: 100

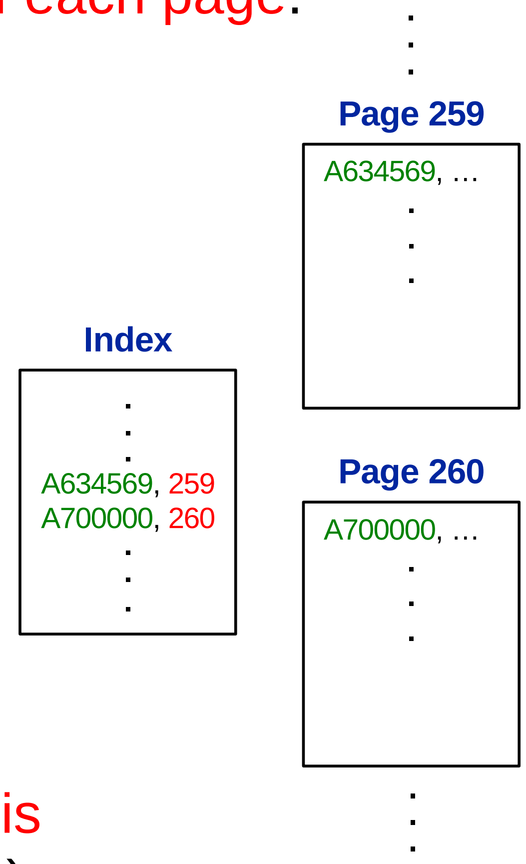
INDEXING BASIC CONCEPTS (CONT'D)

Do we need an index entry for each of the 8,000,000 records?

No We only need an entry for the first record of each page.

Example

If there are two consecutive entries <A634569, 259>, <A700000, 260> in the index, then we know that every hkid starting from A634569 and *up to, but not including*, A700000 must be on page 259.



Therefore, we need only 1,000,000 index entries (one for each page of the main catalog).

Since we can fit 100 index entries per page, and we have 1,000,000 index entries, the index is $\lceil 1{,}000{,}000/100 \rceil$ = 10,000 pages (i.e., 10^4 pages).

index entries/page: 100
index pages: 10,000 (10^4)

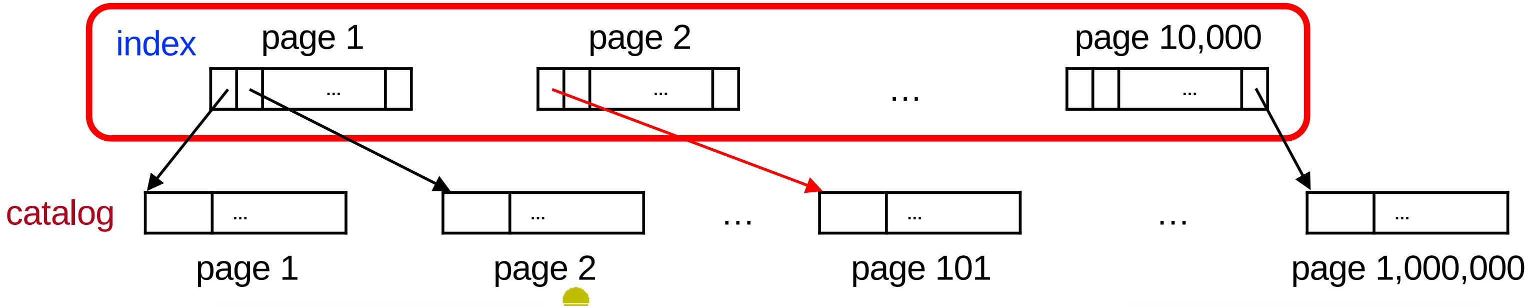
INDEXING BASIC CONCEPTS (CONT'D)

How can we use the index to speed up search for a record?

- Use binary search on the index to find the index page containing the largest hkid value that is <u>smaller or equal to</u> the search hkid value.
  - The cost for this search is $\lceil \log_2 10^4 \rceil = 14$ page accesses.
- Then, follow the pointer from that index entry to the actual catalog page.
  - The cost for this is 1 page access.

Total cost: 14 + 1 = 15 page accesses.

(Page accesses reduced from 20 →15)



INDEXING BASIC CONCEPTS (CONT'D)

Can we reduce the cost even further?

Yes Build an index on the index (i.e., a second level index)!

– The second level index contains 10,000 index entries, one for each page of the first index, and requires $\lceil 10{,}000/100 \rceil$ = 100 (10^2) pages.

– Use binary search on the second level index to find the index page containing the largest hkid that is smaller or equal to the search hkid.

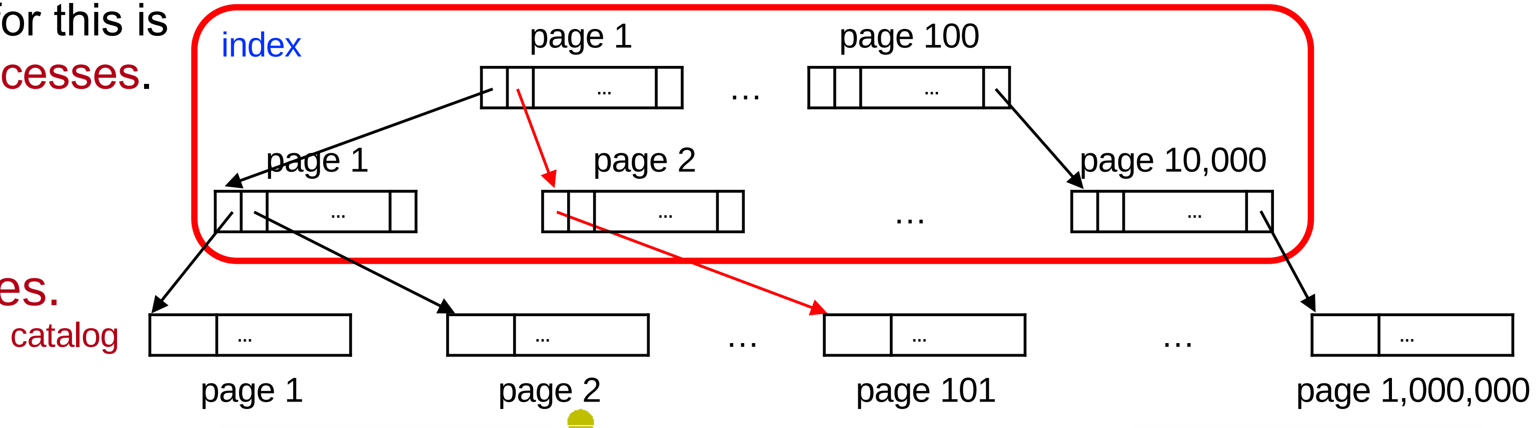
- The cost is $\lceil \log_2 10^2 \rceil$ = 7 page accesses.

– Then, follow the pointer from that index entry to the first level index and finally follow the pointer to the actual catalog page.

- The cost for this is 2 page accesses.

Total cost: 7 + 2 = 9 page accesses.

(Page accesses reduced from 20 → 15 → 9.)



index entries/page: 100
index pages: 10,000 (10^4)

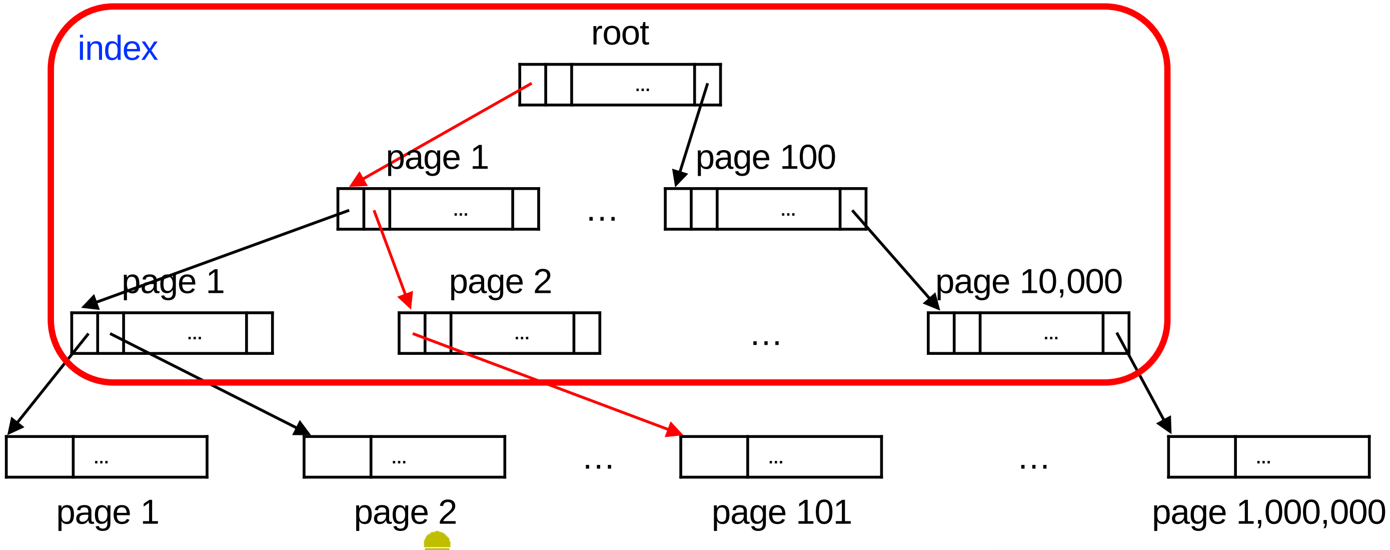
INDEXING BASIC CONCEPTS (CONT'D)

Can we reduce the cost even further?

Yes Build a third level index!

- The third level index contains 100 index entries, one for each page of the second level index, and requires $\lceil 100/100 \rceil$ = 1 page.
- Read this page to find the largest hkid that is smaller or equal to the search hkid, and follow the pointer to the second level index, then follow the pointer to the first level index and finally follow the pointer to the actual catalog page.

Total cost: 4 page accesses!
(1 access for each index level plus 1 access to the catalog)
(Page accesses reduced from 20 → 15 → 9 → 4!)



INDEXING BASIC CONCEPTS (CONT'D)

Search key The attribute, or set of attributes, used to search for records in a file.

☞ **Do not confuse with the concept of primary or candidate key.**

- A primary key is always also a search key.
- A search key is not necessarily a primary key (it can be any table attribute).

– In the preceding example, the search key was hkid since records were found given a value for hkid.

– To find records given the name (or another attribute) additional indexes need to be constructed.

Index file A file consisting of records (called index entries) of the form <search key, pointer>.

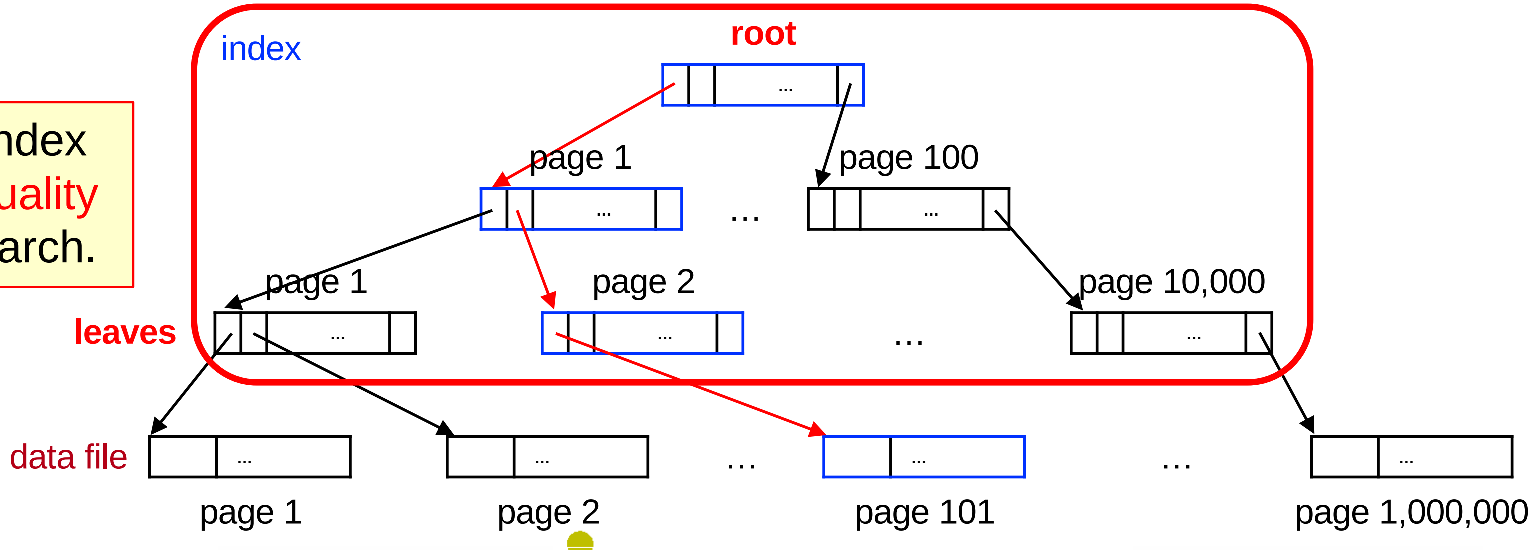
– Index files are typically much smaller than the original file as they do not store all the attributes, but only search-key values and pointers.

ORDERED INDEX

- The index constructed for hkid is an ordered (or tree) index.
  - The index entries are ordered (sorted) on the search key (e.g., hkid).
  - Searching for a record always starts from the root and follows a single path to the leaf that contains the search key of the record.
  - An additional access is then required to retrieve the record from the data file.

Page I/O cost: height of the tree (i.e., number of index levels) plus 1.

An ordered index is good for equality and range search.



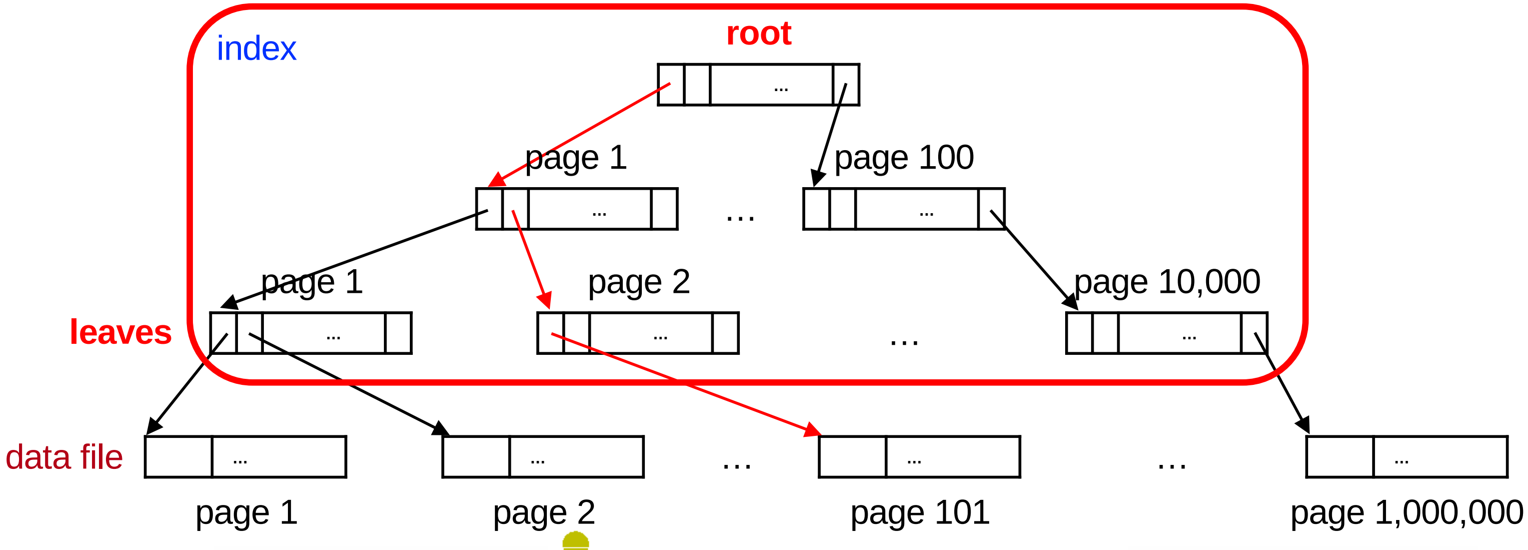
ORDERED INDEX (cont'd)

- An index page is also called an index node.
- The number of children (pointers) of an index node is called the fan-out.

  ☞ **In our example, the fan-out is 100.**

- The height of the tree is $\lceil \log_{\text{fan-out}}(\text{\# of leaf index entries}) \rceil$.

  ☞ **In our example, the height of the tree is** $\lceil \log_{100}(10^6) \rceil = 3$.



DENSE VS. SPARSE INDEX

Dense Index Contains an index entry for *every* search-key value.

Sparse Index Contains an index entry for *only some* search-key values.

Example: The hkid index only has index entries for the first record in each page of the file.

– In general, there is an index entry for every data file page corresponding to the minimum search-key value in the page.

– To locate a record with search-key value *K* (single-level index):

- Find the index entry with largest search-key value ≤ *K*.
- Follow the pointer to the data file page.
- Starting at the first record on this page, search the data file sequentially until the search key value is found or the end of the data file is reached.

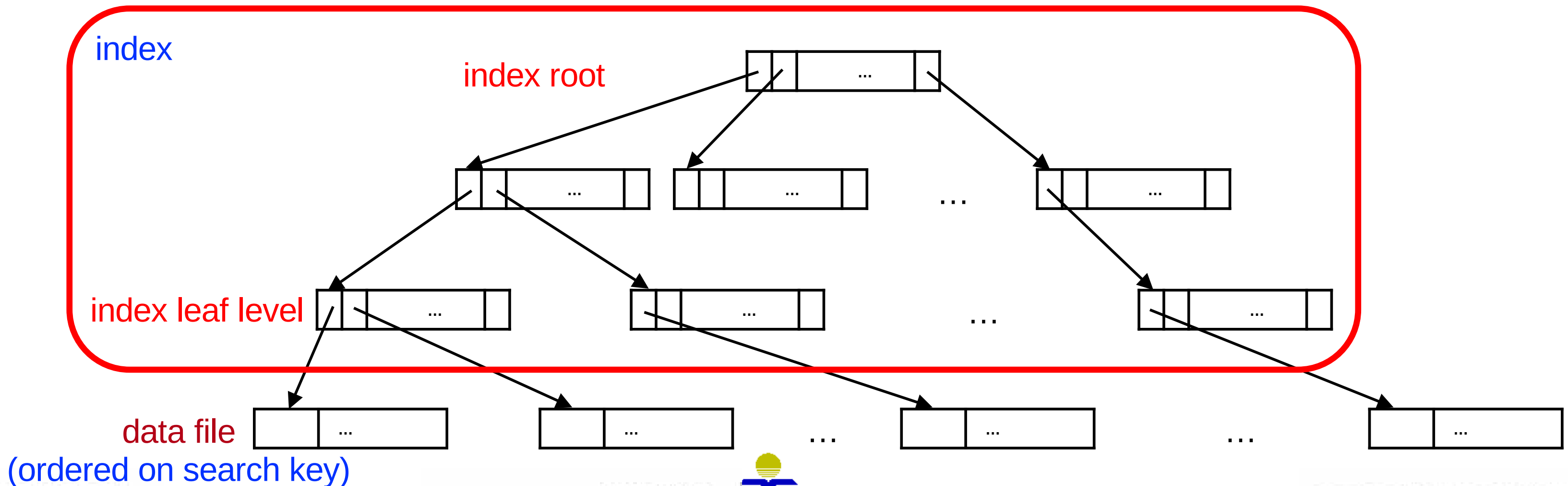
– Sparse indexes require less space and less maintenance overhead for insertions and deletions than dense indexes.

CLUSTERING / PRIMARY INDEX

- A clustering index is an index for which the data file is ordered on the search key of the index (e.g., the index on hkid).
- If a clustering index search key is the primary key, then the index is called a primary index.

  ☞ **There can only be one primary index for a data file.**

  ☞ **A primary index is usually sparse.**
- Index-sequential file: An ordered, sequential file with a primary index (also called ISAM - indexed sequential access method).

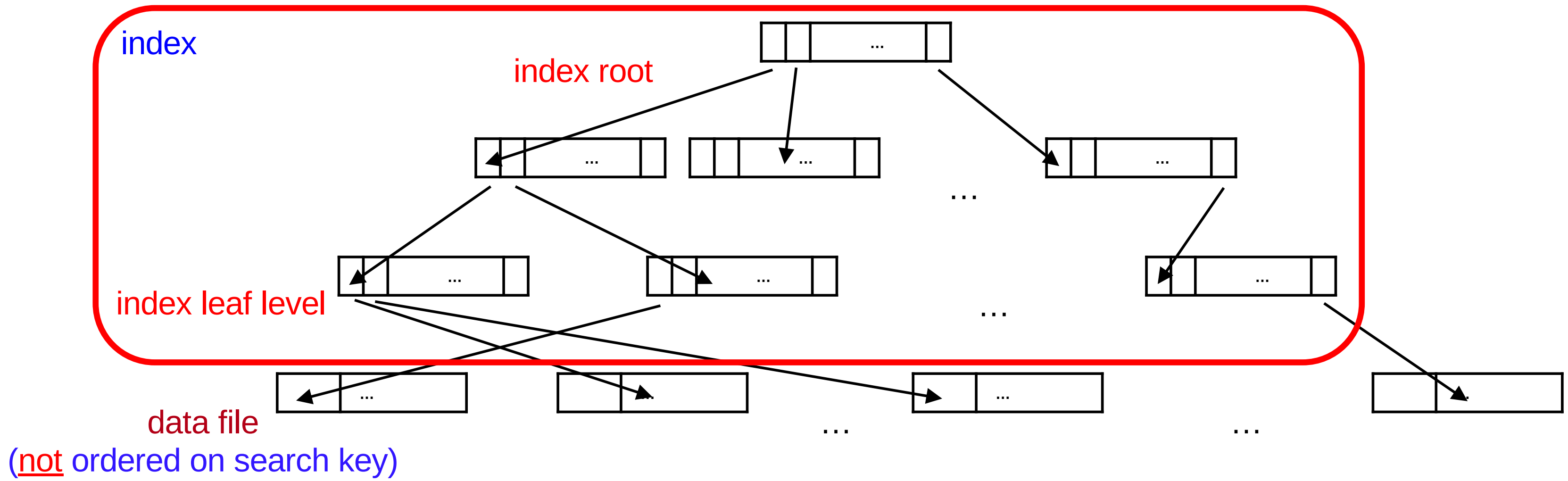


NON-CLUSTERING/SECONDARY INDEX

– A non-clustering/secondary index is an index for which the data file is not ordered on the search key of the index.

☞ **There can be several secondary indexes for a data file.**

☞ **A secondary index must be dense.**



SECONDARY INDEX EXAMPLE

For the catalog of Hong Kong residents, we also want to be able to find records given a name. **How to find the record fast?**

Solution: Build another index on the name

- Since the file is ordered on hkid, the new index must be secondary (since the file is not ordered on the search key) and dense (there is one entry for every search-key value).
- Assuming that all names are distinct (*not realistic!*), the index will contain 8 million entries.
- Assuming that the fan-out is again 100, the cost of finding a record given the name is $\underbrace{\lceil \log_{100}(8{,}000{,}000) \rceil}_{\text{height of the index}} + 1 = 4 + 1 = \underline{5}$ page I/Os.

- 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) and it requires more storage space.

INDEX ON NON-CANDIDATE SEARCH KEY

We want to build an index on name, but there may be several people with the same name.

⟹ Zero, one or more records are retrieved.

☞**Not a problem if the index is clustering and sparse.**

How would you do it?

INDEX ON NON-CANDIDATE SEARCH KEY

If the index is non-clustering (secondary) and dense.

Option 1: Use variable length index entries

– Each entry contains a name and pointers to all records with this name.

Example: <Jackie Chan, *pointer*$_1$, *pointer*$_2$, …., *pointer*$_n$>

Problem: Complicated implementation as a file organization that supports records of variable length is needed.

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: <Jackie Chan, *pointer*$_1$>, <Jackie Chan, *pointer*$_2$>, …, <Jackie Chan, *pointer*$_n$>

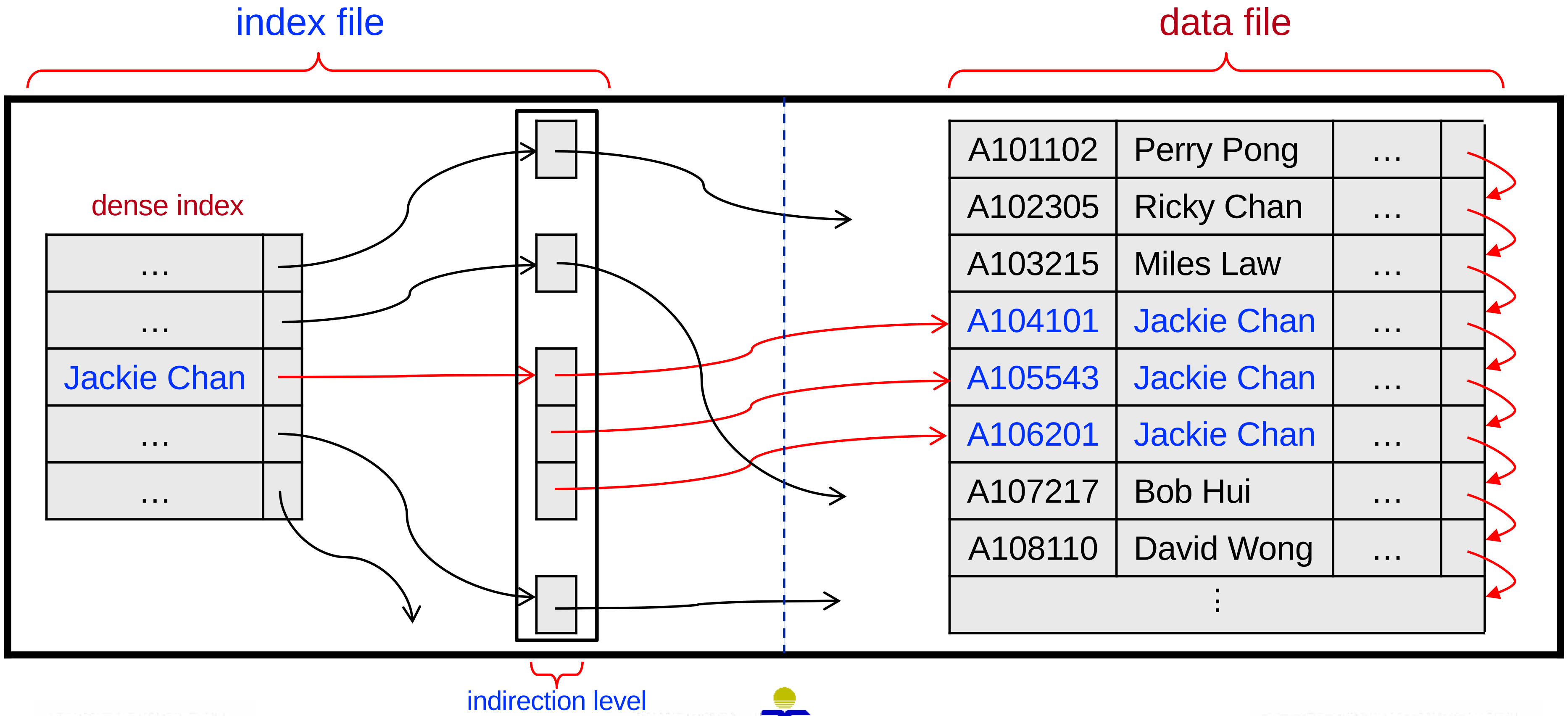
Problem: Redundancy – the name repeats many times.

INDEX ON NON-CANDIDATE SEARCH KEY (cont'd)

Option 3: Use an extra level of indirection (most common approach)

– An index entry points to a list that contains the pointers to all the records with the same name ⟹ requires one additional page access.

☞ **Also called an inverted file.**



INDEX ON COMPOSITE SEARCH KEY

- If a query often uses certain combinations of attributes together (e.g., hkid, age), then creating an index on this attribute combination can speed up retrieval.

☞ **A composite search key is a search key that consists of more than one attribute.**

- The index structure for a composite search key is the same as that for a single attribute search key.

- For a composite search key, the ordering of search key values is the lexicographic ordering.

Example: For two search keys (a_1, a_2) and (b_1, b_2):

$(a_1, a_2) < (b_1, b_2)$ if *either* $a_1 < b_1$ *or* $a_1 = b_1$ and $a_2 < b_2$

– This is basically the same as alphabetic ordering of words.