COMP 3311
DATABASE MANAGEMENT SYSTEMS

LECTURE 14
INDEXING:
HASH INDEX & BITMAP INDEX

STATIC HASHING INDEX

- Hashing can be used not only for file organization, but also for creating an index.
- A hash index organizes the search keys, with their associated record pointers, into a hash file organization.

☞ **Hash indexes are always secondary indexes.**

- If the file itself is organized using hashing, a separate primary hash index on it using the same search-key is not necessary.

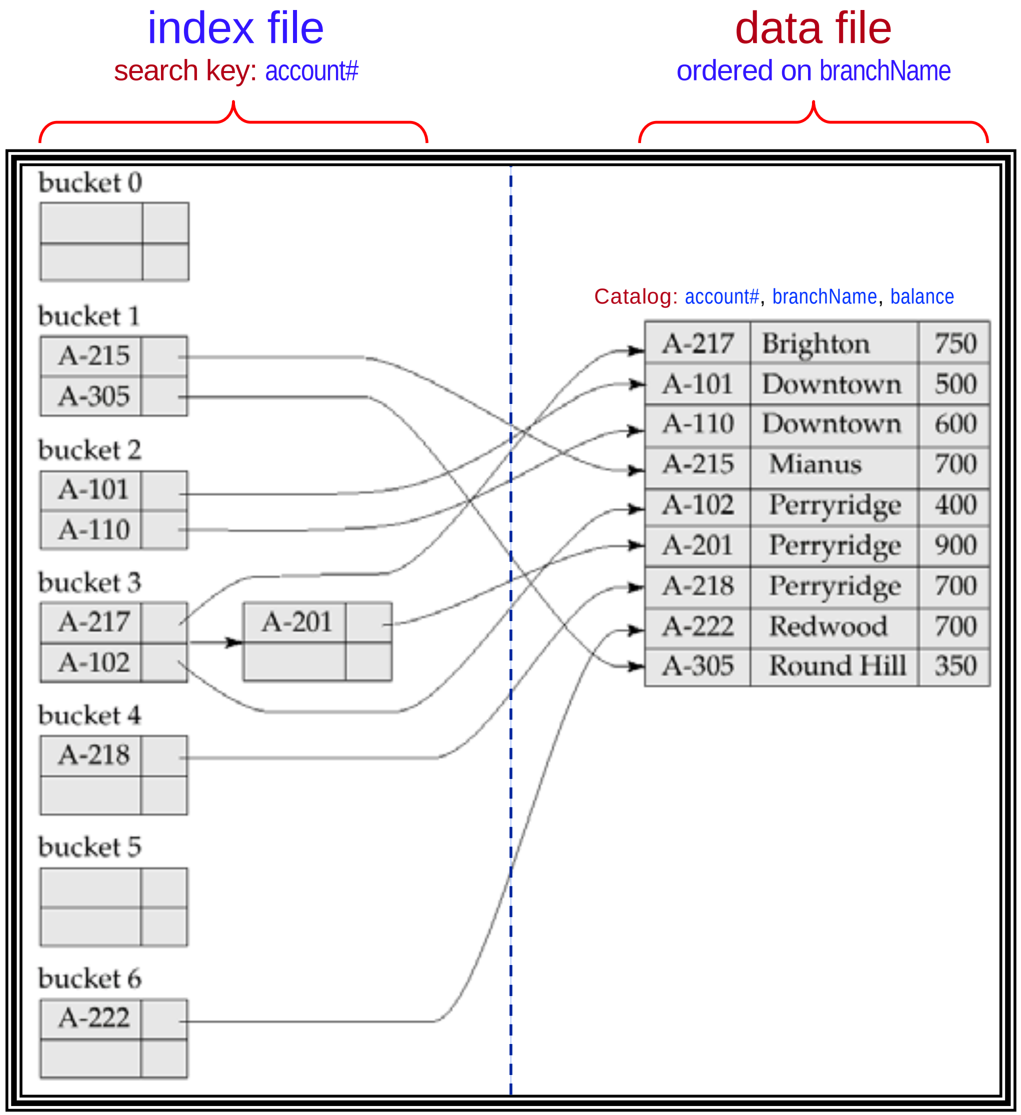
WHY?

- The hash index discussed here is for relatively static tables.
  - We want to build a hash index for an existing table; we expect the number of records not to change too much.

EXAMPLE STATIC HASHING INDEX

Issues to consider:

1. **hash function**
   - We want values to be assigned to pages randomly.
   - Typical hash functions perform computation on the internal binary representation of the search key.
2. **overflow**
   - We want to avoid long chains of overflow pages as this degrades performance.



STATIC HASHING FUNCTIONS

- In the worst case, the hash function maps all search-key values to the same page.
- This makes access time proportional to the number of search-key values in the file.
- The ideal hash function is random, so each page will have the same number of records assigned to it irrespective 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 representation of all the characters in the string could be added and the sum modulo the number of pages could be returned.

HANDLING Page OVERFLOWS

- Page overflow can occur because of
  - Insufficient number of pages.
  - Skew in the distribution of records, which can occur for two reasons.
    - Multiple records have the same search-key value.
    - The chosen hash function produces a non-uniform distribution of key values.

- Page overflow is handled by using *overflow pages.*
  - If overflow pages also overflow, they are chained together in a linked list.
  - Long chains degrade performance because a query has to search all pages in the chain.

STATIC HASHING DEFICIENCIES

- In static hashing, a function *h* maps search-key values to a fixed set *B* of page addresses.
  - Databases usually grow over time.
  - If the initial number of fixed pages is too small, performance will degrade due to too many overflows.
  - If the file size at some point in the future is anticipated to grow and the number of pages allocated accordingly, a significant amount of space will be wasted initially.
  - If the database shrinks, again space will be wasted.
  - One option is periodic re-organization of the file with a new hash function, but this is very expensive. **WHY?**
- These problems can be avoided by using techniques that allow dynamic modification of the hash function (number of pages).

DYNAMIC HASHING: EXTENDABLE HASHING

Situation

A page (primary page) becomes full and overflows.

Why not re-organize the file by *doubling the number of pages*?

☞ **However, reading and writing all pages is expensive!**

Idea

Use a *directory of pointers to pages*; to double the number of pages that can be addressed, *double the directory,* but split just the page that overflowed.

- The directory is much smaller than the data file, so doubling it is much cheaper.
- Only one page of data entries is split. *No overflow page*!

☞ **The technique relies on how the hash function is adjusted!**

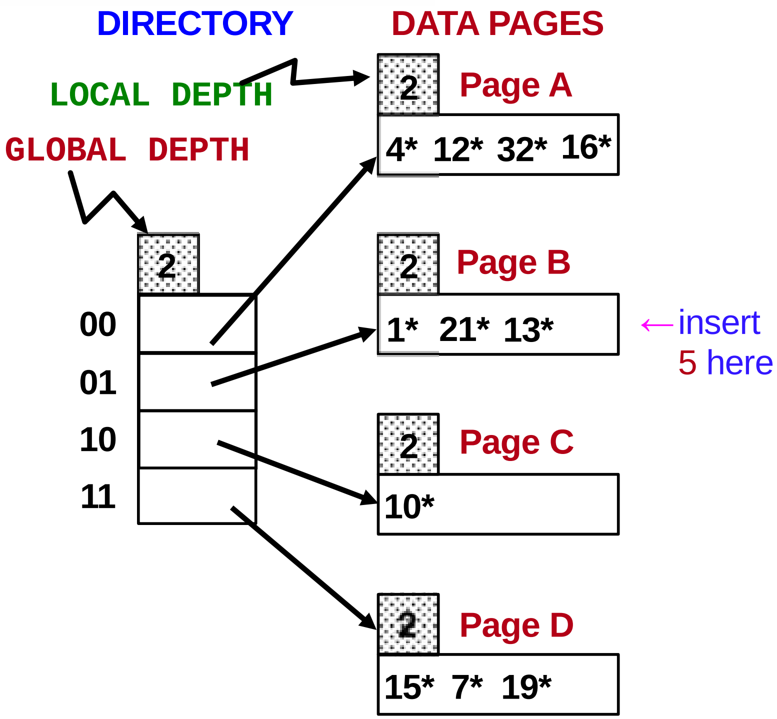
EXTENDABLE HASHING INDEX

Insertion: (page size is 4)

If *r* is the record to insert, first compute its hash **h**(*r*).

To find the page to store *r*, use the right-most *global depth bits* of the binary representation of **h**(*r*).

Example: If **h**(*r*) = 5 ⟹ binary 101, *r* is stored in the page pointed to by 01 since global depth is 2 bits.



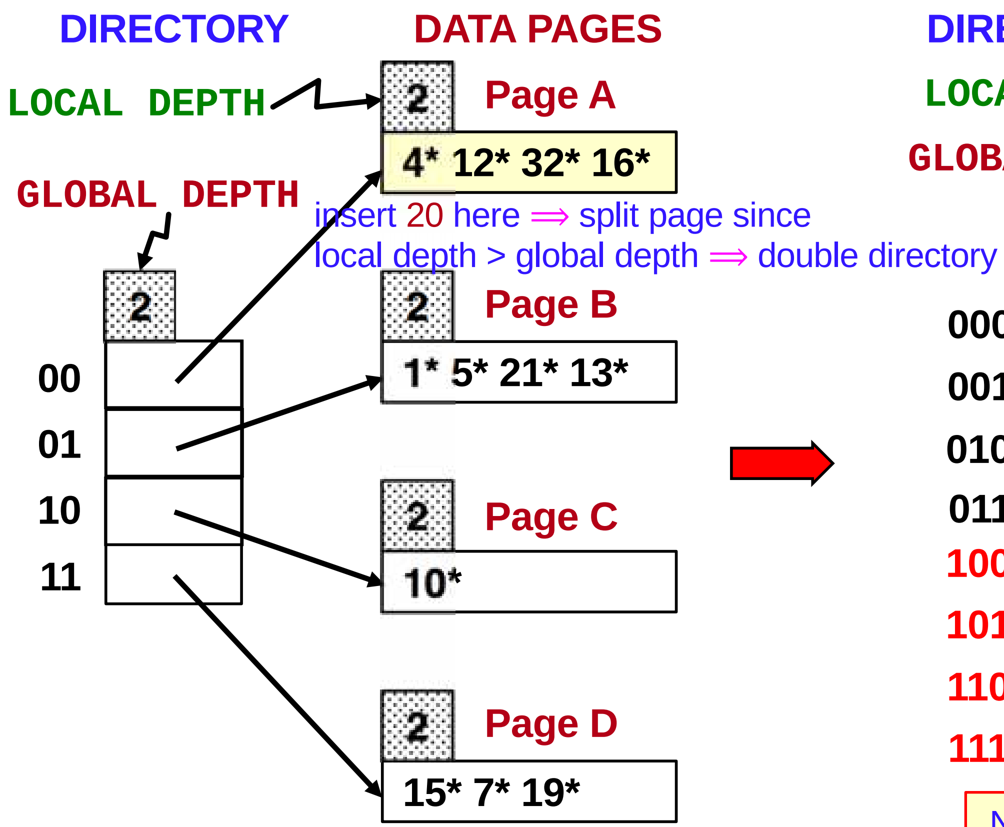
If a page is full, *split* it (*allocate a new page and re-distribute*).

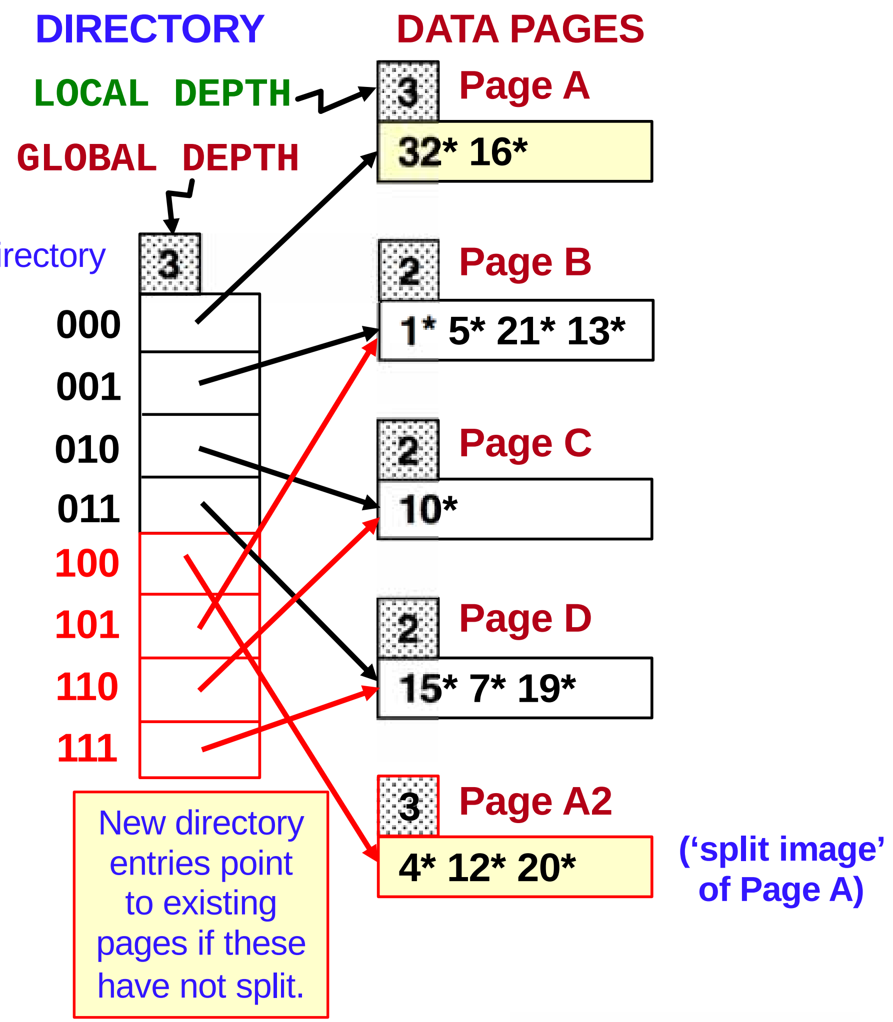
If necessary, double the directory. Splitting a page does not always require doubling the directory; to tell whether to double the directory, we compare *global depth* with *local depth* for the split page. If the local depth of the split page is greater than the global depth (i.e., we are using more bits in the page than in the directory), then the directory needs to be doubled.

Example: directory doubling

(page size is 4)

Insert: **h**(*r*)=20 (10100)





key value	4	12	16	32
binary value	100	100	000	000

EXTENDABLE INDEX HASHING: POINTS TO NOTE

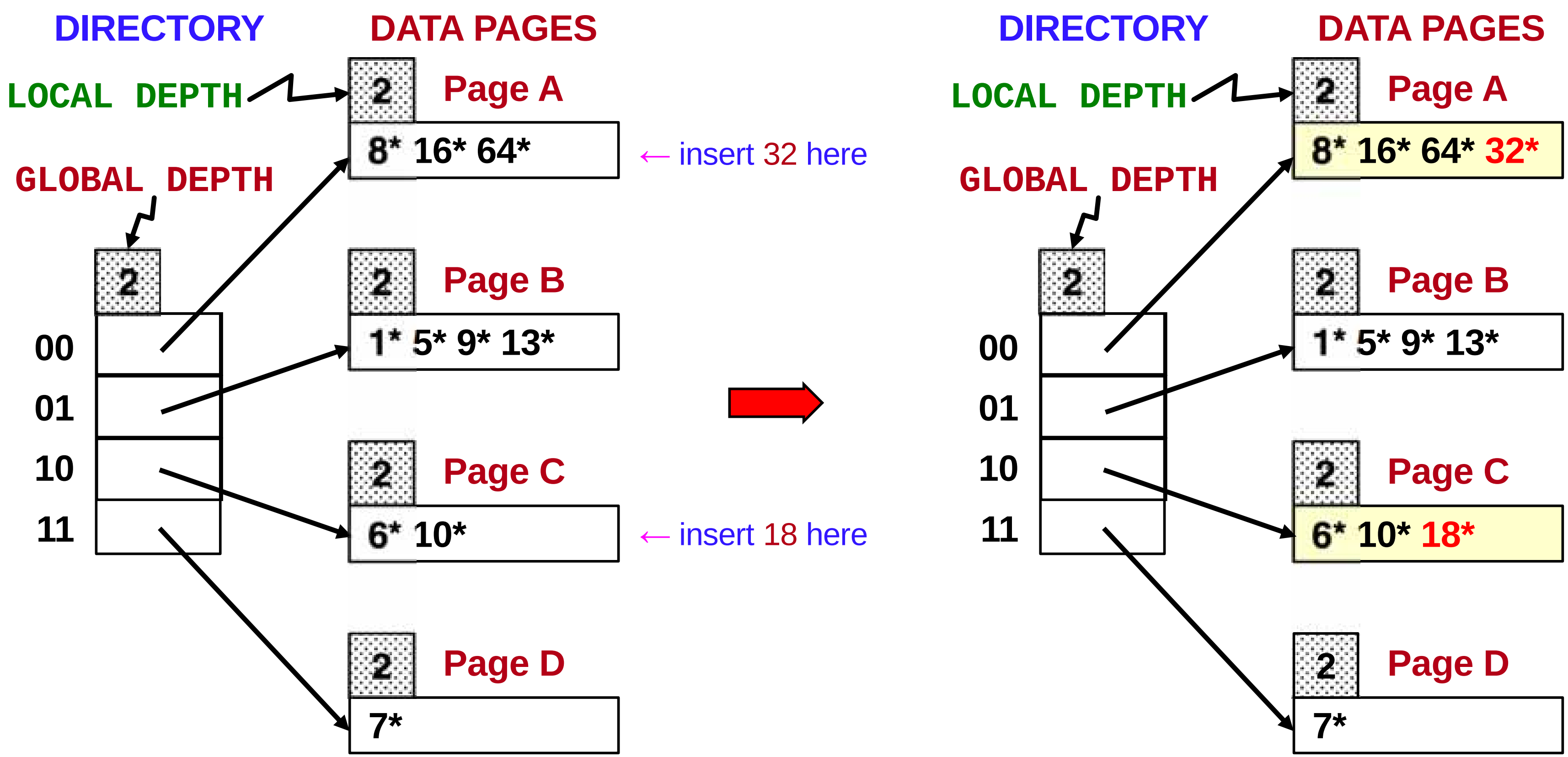
- 20 = binary 10100. Last **2** bits (00) tell us *r* belongs in page A or A2. Last **3** bits needed to tell to which page *r* belongs.
  - **Global depth of directory:** max # of bits needed to tell which page an entry belongs to.
  - **Local depth of a page:** # of bits used to determine if an entry belongs to this page.
- When does page split cause directory doubling?
  - Before an insert, *local depth* of page = *global depth*.
  - If an insert causes the *local depth* to become > *global depth*, the directory is doubled by *copying it* and ‘adjusting’ the pointer to the split image page.
  - Use of least significant bits enables efficient doubling via copying of directory!

EXTENDABLE HASHING INDEX EXAMPLE

Consider insertions on the following hash index where the hash function is determined by the least significant (right-most) bits.

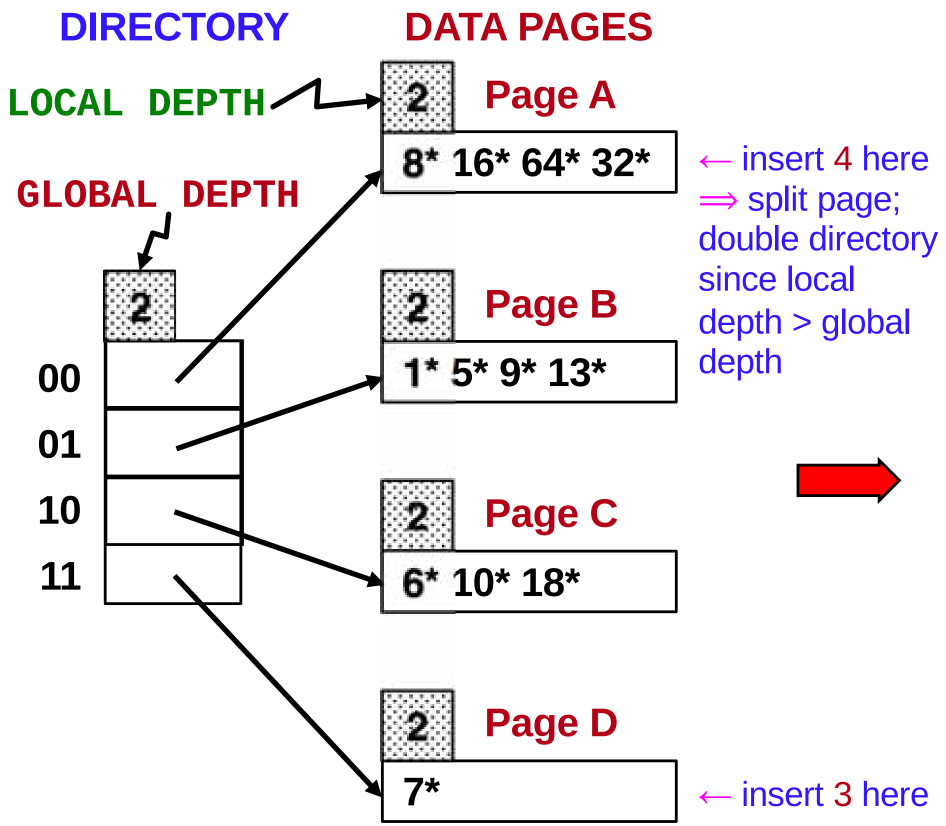
page size is 4

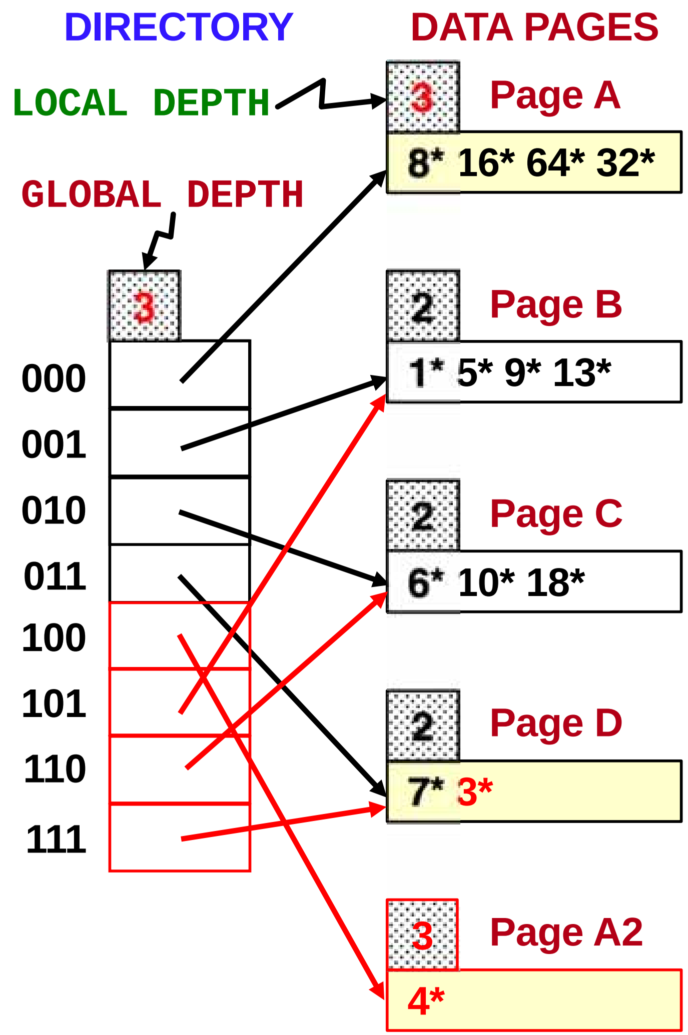
Insert: 18 (010010), 32 (100000)



EXTENDABLE HASHING INDEX EXAMPLE (CONT'D)

Insert: 3 (011), 4 (100)

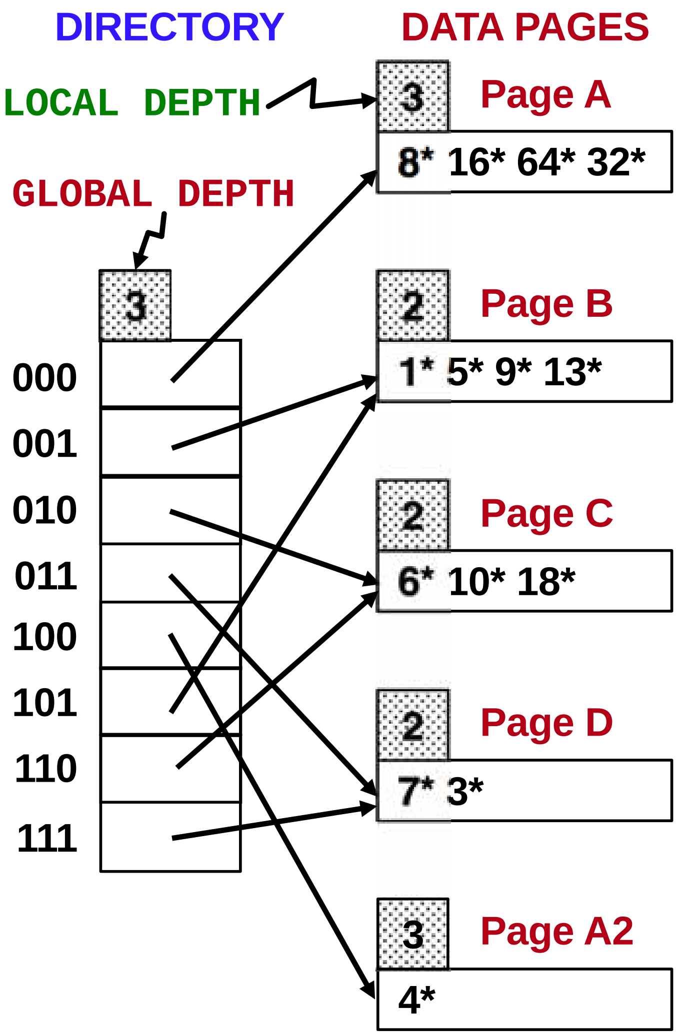




key value	4	8	16	32	64
binary value	100	000	000	000	000

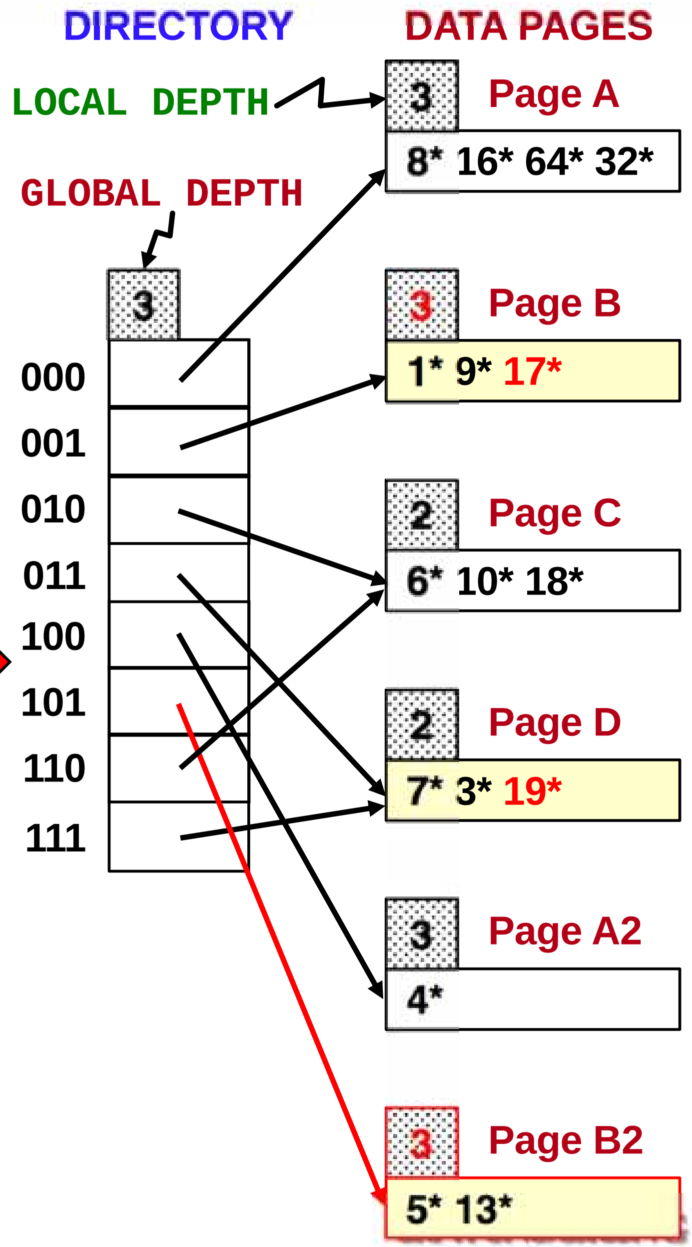
EXTENDABLE HASHING INDEX EXAMPLE (CONT'D)

Insert: 19 (10011), 17 (10001)

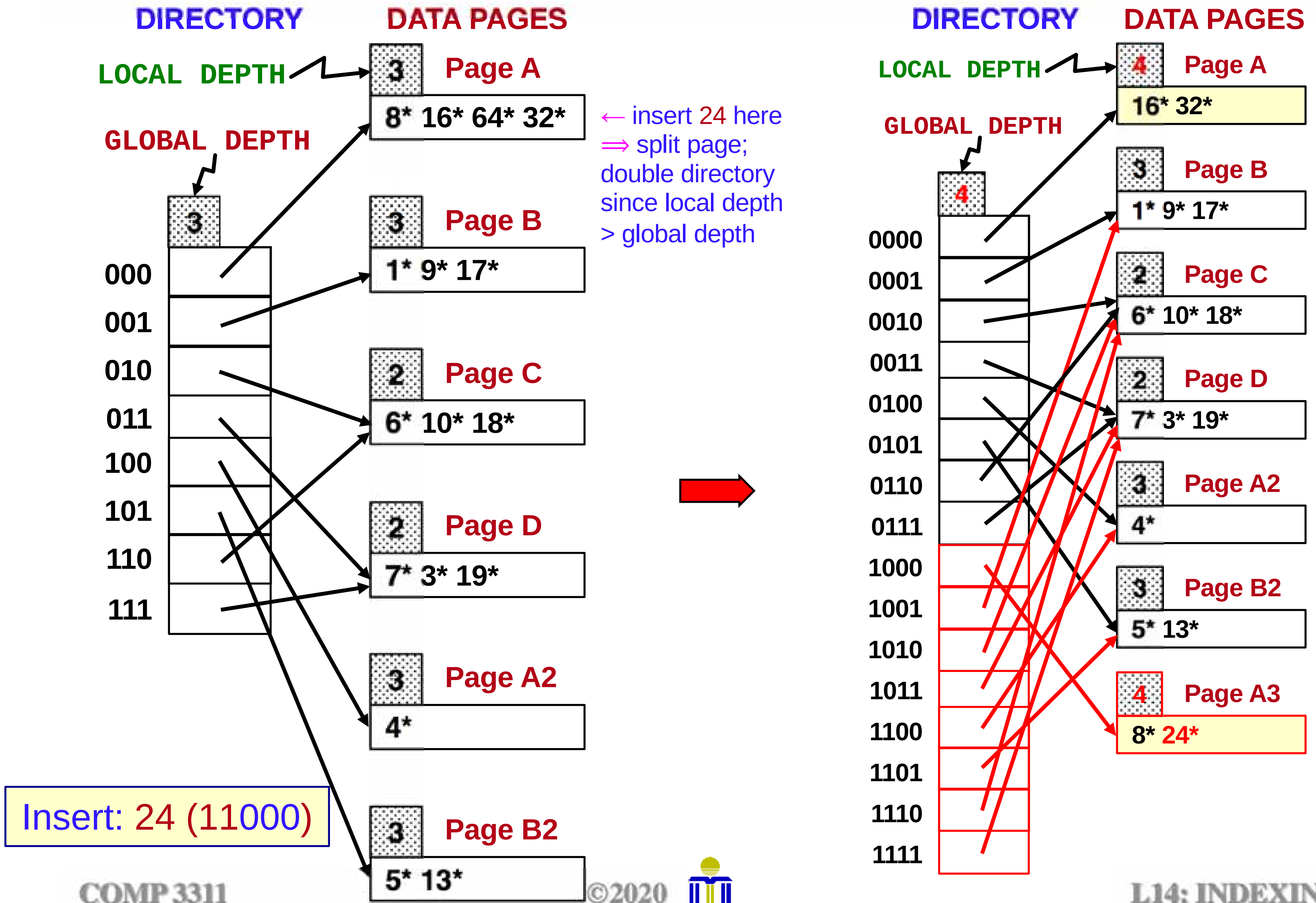


← insert 17 here
⇒ split page, but do not double the directory since local depth = global depth; adjust pointers

← insert 19 here



EXTENDABLE HASHING INDEX EXAMPLE (CONT'D)



EXTENDABLE HASHING INDEX: OBSERVATIONS

- If the directory fits in memory, equality search can be answered with one disk access; else two disk accesses are needed.
  - For a 100MB file with 100 bytes/record and 4K pages containing 1,000,000 records (as data records) and 25,000 directory entries, chances are high that the directory will fit in memory.
  - The directory grows in spurts and, if the distribution *of hash values* is skewed, the directory can grow large.
  - Multiple entries with the same hash value cause overflows!

Deletion

If removal of a data record makes a page empty, it can be merged with its 'split image'.

If each directory element points to the same page as its split image, this can halve the directory.

BITMAP INDEX

- Bitmap indexes are a special type of index designed for efficient querying on multiple search keys.

☞ **A bitmap is simply an array of bits.**

- Records in a table are assumed to be numbered sequentially from, say, 0.
  - Given a number *n* it must be easy to retrieve record *n.*
- Bitmap indexes are applicable for attributes that take on a relatively small number of distinct values.
  - E.g. gender, country, state, …
  - E.g. income-level (where income is broken up into a small number of categories such as 0-9999, 10000-19999, 20000-50000, 50000-infinity).

BITMAP INDEX (CONT'D)

- In its simplest form, a bitmap index on an attribute has a bitmap *for each value* of the attribute.

  ☞ **A bitmap has as many bits as there are records.**

  ☞ **In a bitmap for a value *v*, the bit for a record is 1 if the record has the value *v* for the attribute, and 0 otherwise.**

- Queries are answered using bitmap operations.
  - Intersection (and)
  - Union (or)
  - Complementation (not)

- Each operation takes two bitmaps of the same size and applies the operation on the corresponding bits to get the result bitmap.

BITMAP INDEX (CONT'D)

Query: Find records for males with income-level L1.

male bit vector: 10010
income-level L1 bit vector: 10100
10000

and the bit vectors together

← records that qualify are where the bit is 1

– *Counting* the number of matching records is even faster
⟹ sum up the bits (especially useful for aggregation queries).

record number	*name*	*gender*	*address*	*income-level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income-level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Each bitmap has one bit for each record in the file.

BITMAP INDEX (CONT'D)

- Bitmap indexes are useful for queries on multiple attributes, but are not particularly useful for single attribute queries.

- Bitmap indexes are generally very small compared with the size of a table.

- Deletion needs to be handled properly.
  - Can us an existence bitmap to record whether there is a valid record at a given location.
  - An existence bitmap is needed for complementation.
    - not(*A*=*v*): *(NOT bitmap-A-v) AND ExistenceBitmap*

- Bitmaps should be kept for all values of an attribute, even null values.
  - To correctly handle SQL null semantics for NOT(*A*=*v*):
    - intersect previous result with (NOT *bitmap-A-Null*).

INDEX SELECTION CONSIDERATIONS

- **Types of accesses supported efficiently**
  – Hash indexes are in general good for equality selection queries, but do not support range searches.
  – Secondary indexes are not good for queries that retrieve many records as the same page may be retrieved multiple times.
- **Update time**
  – When a data file is modified, every index on the data file must be updated.
- **Space overhead**
  – The index should be, in general, much smaller than the data file.

INDEXING: SUMMARY

Indexing reduces the overhead of searching a table by trading space (storage overhead) for time (faster search).

Dense and sparse index

- **Dense:** contain an index entry for every search-key value.
- **Sparse:** contain an index entry for only some search-key values.

Clustering and non-clustering (secondary) index

- **Clustering:** search-key sort order matches table sort order.
- **Non-clustering:** search-key sort order does not match table sort order.

Indexing methods

- **B^+-tree:** use search-key value to search a tree-index.
- **Hash:** apply a function to the search-key value to access a directory.
- **Bitmap:** apply appropriate bitmap for search-key value.