Chapter 13: Disk Storage, Basic File Structures, and Hashing

Chapter Outline

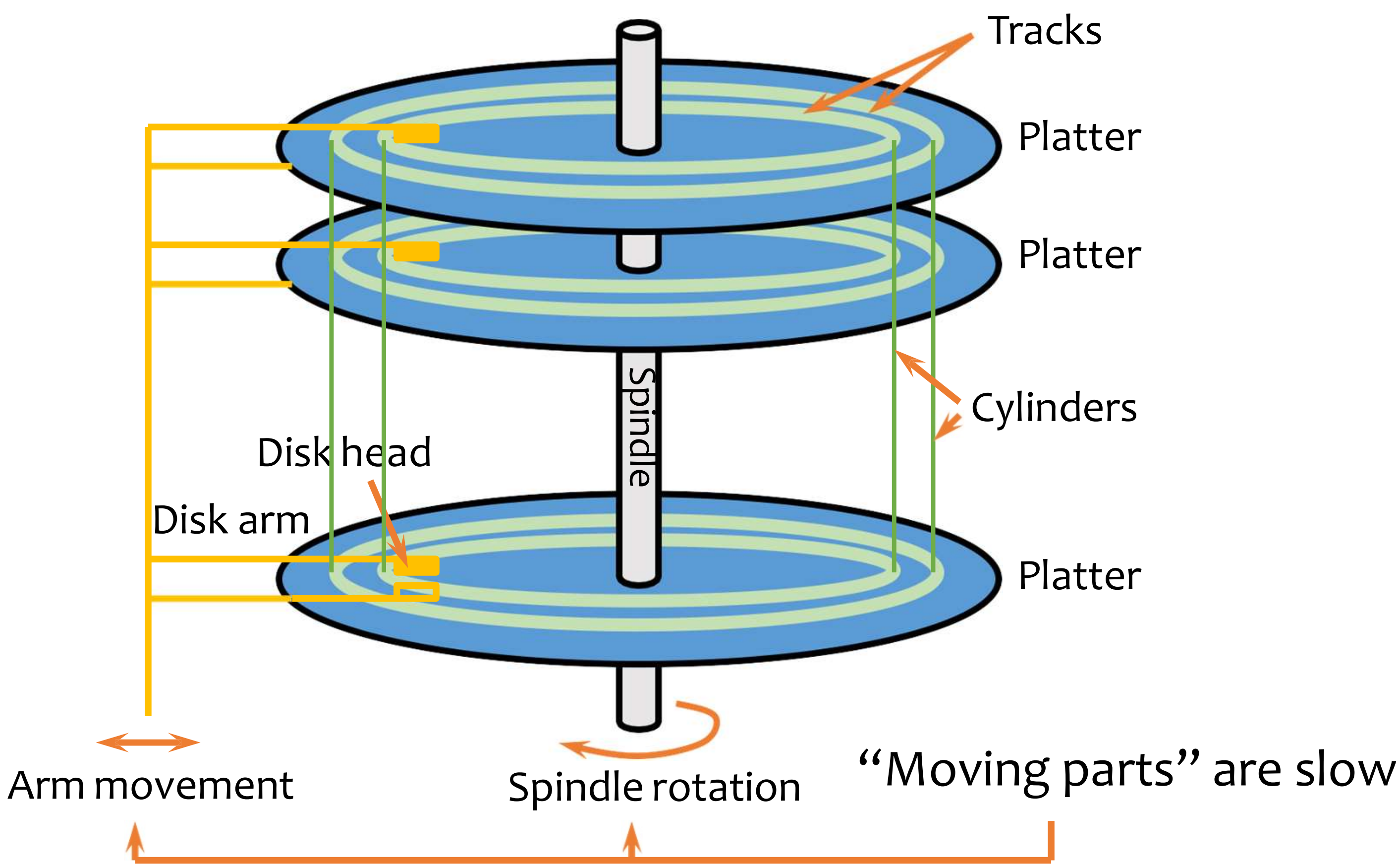
- Disk Storage Devices
- Files of Records
- Operations on Files
- Unordered Files
- Ordered Files
- Hashed Files
  - Dynamic and Extendible Hashing Techniques

Disk Storage Devices

- Data stored as magnetized areas on magnetic disk surfaces.
- A disk pack contains several magnetic disks connected to a rotating spindle.
- Disks are divided into concentric circular tracks on each disk surface.
  - Track capacities vary typically from 4 to 50 Kbytes or more
- A track is divided into smaller blocks or sectors
  - because it usually contains a large amount of information

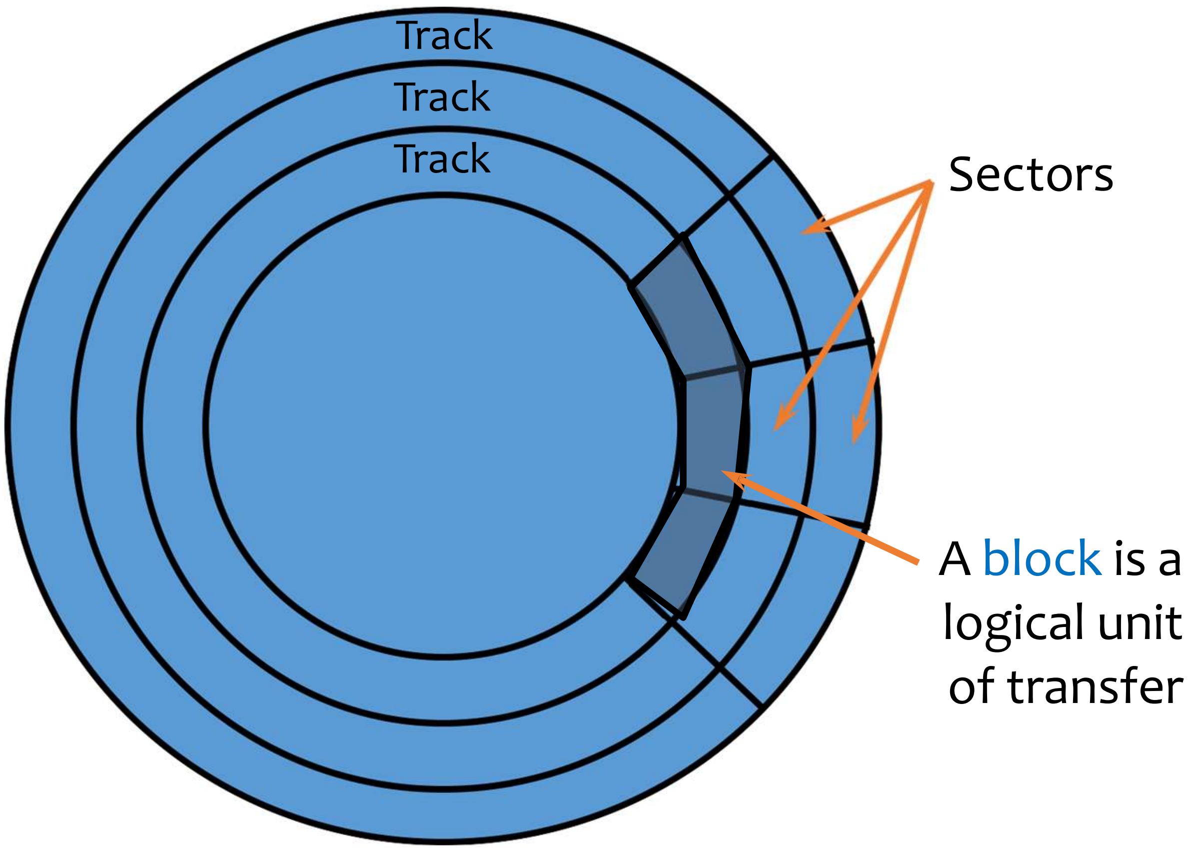
*

A typical hard drive



Top view

"Zoning": more sectors/data on outer tracks



Disk access time

Sum of:

- Seek time: time for disk heads to move to the correct cylinder
- Rotational delay: time for the desired block to rotate under the disk head
- Transfer time: time to read/write data in the block (= time for disk to rotate over the block)

Data on External Storage

- Data must persist on disk across program executions in a DBMS
  - Data is huge
  - Must persist across executions
  - But has to be fetched into main memory when DBMS processes the data

- The unit of information for reading data from disk, or writing data to disk, is a page

- Disks: Can retrieve random page at fixed cost
  - But reading several consecutive pages is much cheaper than reading them in random order

Disk Space Management

- Lowest layer of DBMS software manages space on disk
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Size of a page = size of a disk block
  = data unit
- Request for a sequence of pages often satisfied by allocating contiguous blocks on disk
- Space on disk managed by Disk-space Manager
  - Higher levels don't need to know how this is done, or how free space is managed

Buffer Management

Suppose

- 1 million pages in db, but only space for 1000 in memory
- A query needs to scan the entire file
- DBMS has to
  - bring pages into main memory
  - decide which existing pages to replace to make room for a new page
  - called Replacement Policy
- Managed by the Buffer manager
  - Files and access methods ask the buffer manager to access a page mentioning the “record id” (soon)
  - Buffer manager loads the page if not already there

Files of Records

- Page or block is OK when doing I/O, but higher levels of DBMS operate on records, and files of records
- FILE: A collection of pages, each containing a collection of records
- Must support:
  - insert/delete/modify record
  - read a particular record (specified using record id)
  - scan all records (possibly with some conditions on the records to be retrieved)

File Organization

- File organization: Method of arranging a file of records on external storage
  - One file can have multiple pages
  - Record id (rid) is sufficient to physically locate the page containing the record on disk
  - Indexes are data structures that allow us to find the record ids of records with given values in index search key fields
- NOTE: Several uses of “keys” in a database
  - Primary/foreign/candidate/super keys
  - Index search keys

Alternative File Organizations

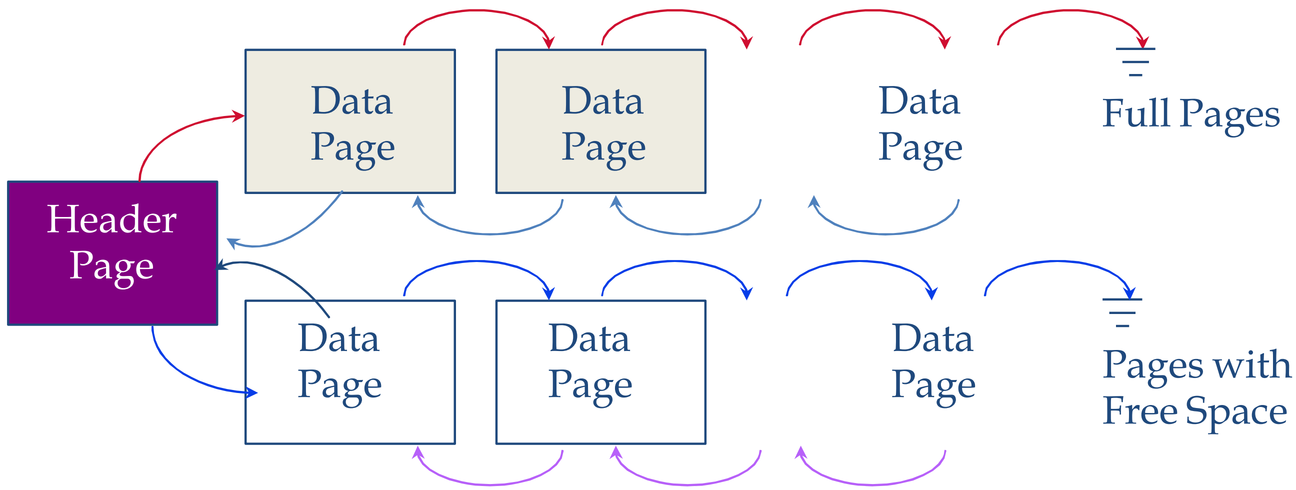
Many alternatives exist, each ideal for some situations, and not so good in others:

- Heap (random order) files: Suitable when typical access is a file scan retrieving all records
- Sorted Files: Best if records must be retrieved in some order, or only a "range" of records is needed.
- Indexes: Data structures to organize records via trees or hashing
  - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
  - Updates are much faster than in sorted files

Unordered (Heap) Files

- Simplest file structure contains records in no particular order
- As file grows and shrinks, disk pages are allocated and de-allocated
- To support record level operations, we must:
  - keep track of the pages in a file
  - keep track of free space on pages
  - keep track of the *records* on a page
- There are many alternatives for keeping track of this

Heap File Implemented as a List



- The header page id and Heap file name must be stored someplace
- Each page contains 2 `pointers' plus data
- Problem?
  - to insert a new record, we may need to scan several pages on the free list to find one with sufficient space

How do we arrange a collection of records on a page?

- Each page contains several slots
  - one for each record

- Record is identified by <page-id, slot-number>

- Fixed-Length Records
- Variable-Length Records

- For both, there are options for
  - Record formats (how to organize the fields within a record)
  - Page formats (how to organize the records within a page)

Ordered Files

- Also called a sequential file.
- File records are kept sorted by the values of an ordering field.
- Insertion is expensive: records must be inserted in the correct order.
  - It is common to keep a separate unordered *overflow* (or *transaction*) file for new records to improve insertion efficiency; this is periodically merged with the main ordered file.
- A binary search can be used to search for a record on its ordering field value.
  - This requires reading and searching $\log_2$ of the file blocks on the average, an improvement over linear search.
- Reading the records in order of the ordering field is quite efficient.

Ordered Files (contd.)

	NAME	SSN	BIRTHDATE	JOB	SALARY	SEX
block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					
block 3	Alexander, Ed					
	Alfred, Bob					
	⋮					
	Allen, Sam					
block 4	Allen, Troy					
	Anders, Keith					
	⋮					
	Anderson, Rob					
block 5	Anderson, Zach					
	Angeli, Joe					
	⋮					
	Archer, Sue					
block 6	Arnold, Mack					
	Arnold, Steven					
	⋮					
	Atkins, Timothy					
⋮						
block n –1	Wong, James					
	Wood, Donald					
	⋮					
	Woods, Manny					
block n	Wright, Pam					
	Wyatt, Charles					
	⋮					
	Zimmer, Byron					

Average Access Times

The following table shows the average access time to access a specific record for a given type of file

TABLE 13.2 AVERAGE ACCESS TIMES FOR BASIC FILE ORGANIZATIONS

TYPE OF ORGANIZATION	ACCESS/SEARCH METHOD	AVERAGE TIME TO ACCESS A SPECIFIC RECORD
Heap (Unordered)	Sequential scan (Linear Search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary Search	$\log_2 b$

*

Hashed Files

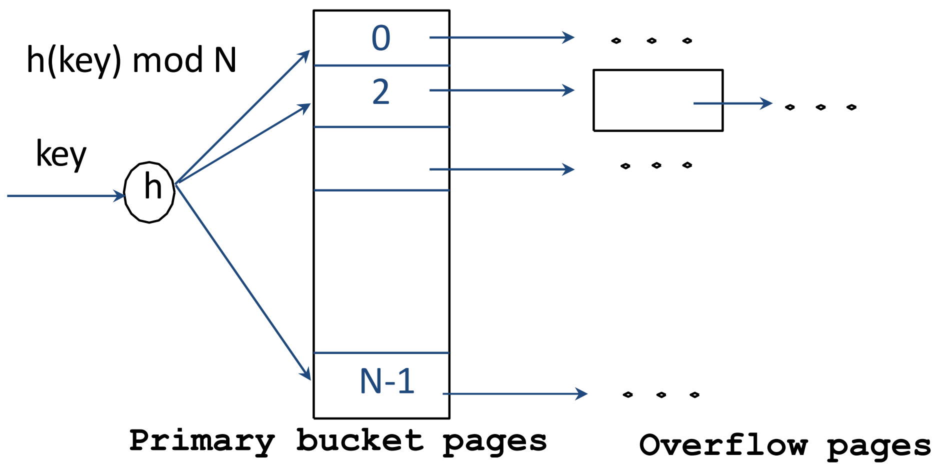
- Hashing for disk files is called External Hashing
- The file blocks are divided into M equal-sized buckets, numbered $bucket_0$, $bucket_1$, ..., $bucket_{M-1}$.
  - Typically, a bucket corresponds to one (or a fixed number of) disk block.
- One of the file fields is designated to be the hash key of the file.
- The record with hash key value K is stored in bucket i, where i=h(K), and h is the hashing function. E.g., h(k)= K mod M
- Search is very efficient on the hash key.
- Collisions occur when a new record hashes to a bucket that is already full.
  - An overflow file is kept for storing such records.
  - Overflow records that hash to each bucket can be linked together.

Hashed Files (contd.)

- There are numerous methods for collision resolution, including the following:
  - Open addressing: Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.
  - Chaining: For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. In addition, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location.
  - Multiple hashing: The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.

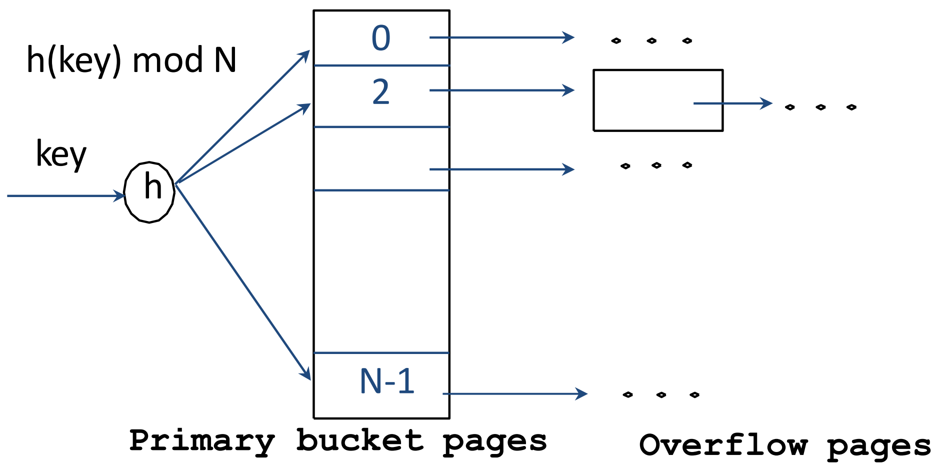
Static Hashing

- Pages containing data = a collection of buckets
  - each bucket has one primary page, also possibly overflow pages
  - buckets contain data entries k*



Static Hashing

- # primary pages fixed
  - allocated sequentially, never de-allocated, overflow pages if needed.
- **h**(k) mod N = bucket to which data entry with key k belongs
  - N = # of buckets



Static Hashing

- Hash function works on search key field of record r
  - Must distribute values over range 0 ... N-1
  - h(key) = (a * key + b) usually works well
    - bucket = h(key) mod N
  - a and b are constants – chosen to tune h
- Advantage:
  - #buckets known – pages can be allocated sequentially
  - search needs 1 I/O (if no overflow page)
  - insert/delete needs 2 I/O (if no overflow page) (why 2?)
- Disadvantage:
  - Long overflow chains can develop if file grows and degrade performance (data skew)
  - Or waste of space if file shrinks
- Solutions:
  - keep some pages say 80% full initially
  - Periodically rehash if overflow pages (can be expensive)
  - or use Dynamic Hashing

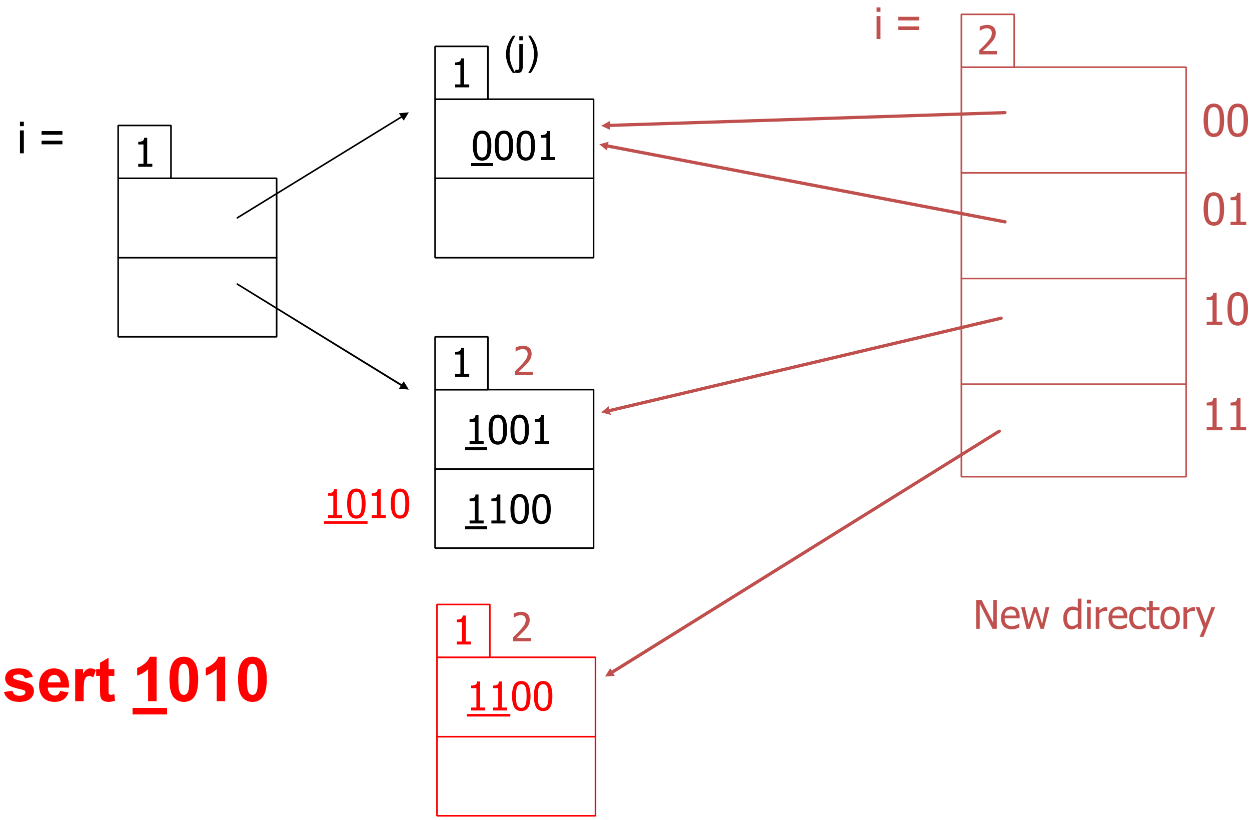
Extendible Hashing

- Consider static hashing
- Bucket (primary page) becomes full

- Why not re-organize file by doubling # of buckets?
  - Reading and writing (double #pages) all pages is expensive

- Idea: Use directory of pointers to buckets
  - double # of buckets by doubling the directory, splitting just the bucket that overflowed
  - Directory much smaller than file, so doubling it is much cheaper
  - Only one page of data entries is split
  - No overflow page (new bucket, no new overflow page)
  - Trick lies in how hash function is adjusted

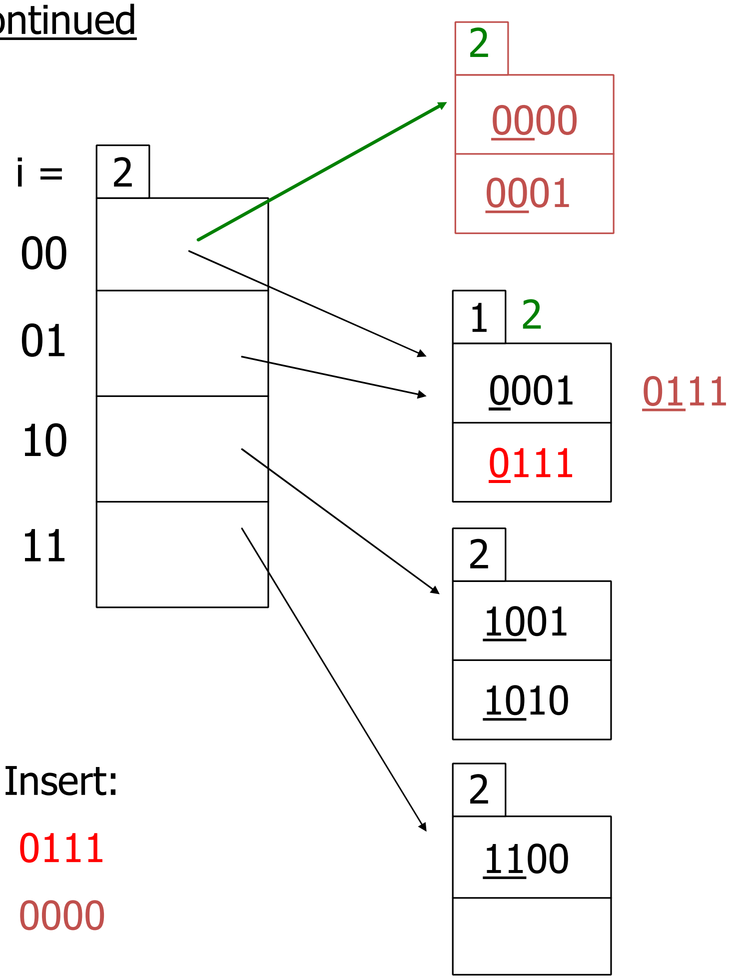
Example: h(k) is 4 bits; 2 keys/block



Insert 1010

*

Example continued

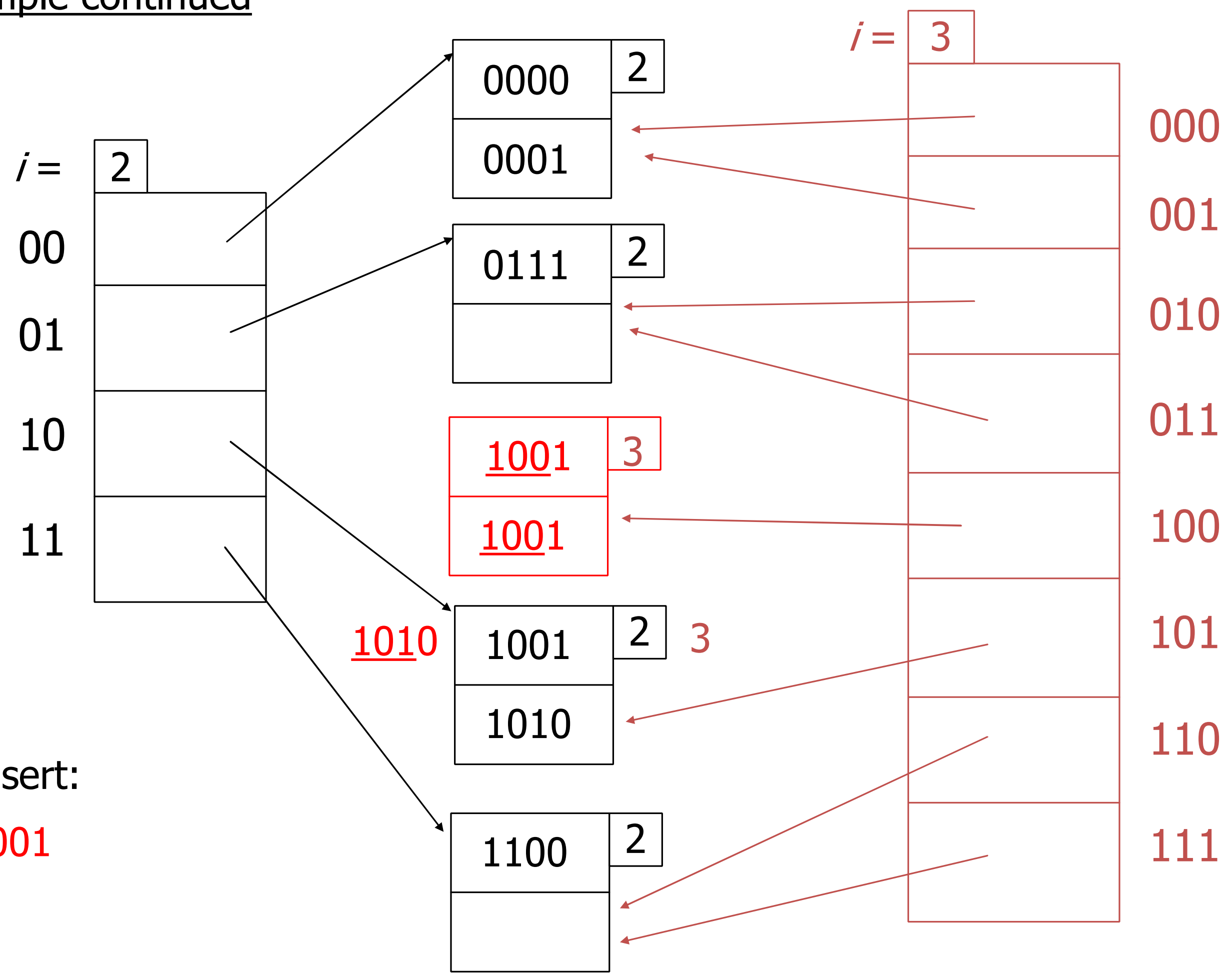


Insert:

0111

0000

Example continued



Insert:

1001

*

When does bucket split cause directory doubling?

- Before insert, local depth of bucket = global depth
- Insert causes local depth to become > global depth
- directory is doubled by copying it over and `fixing' pointer to split image page

Comments on Extendible Hashing

- If directory fits in memory, equality search answered with one disk access (to access the bucket); else:
  - Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large
  - Multiple entries with same hash value cause problems
- Delete:
  - If removal of data entry makes bucket empty, can be merged with `split image'
  - If each directory element points to same bucket as its split image, can halve directory.

Extendible Hashing

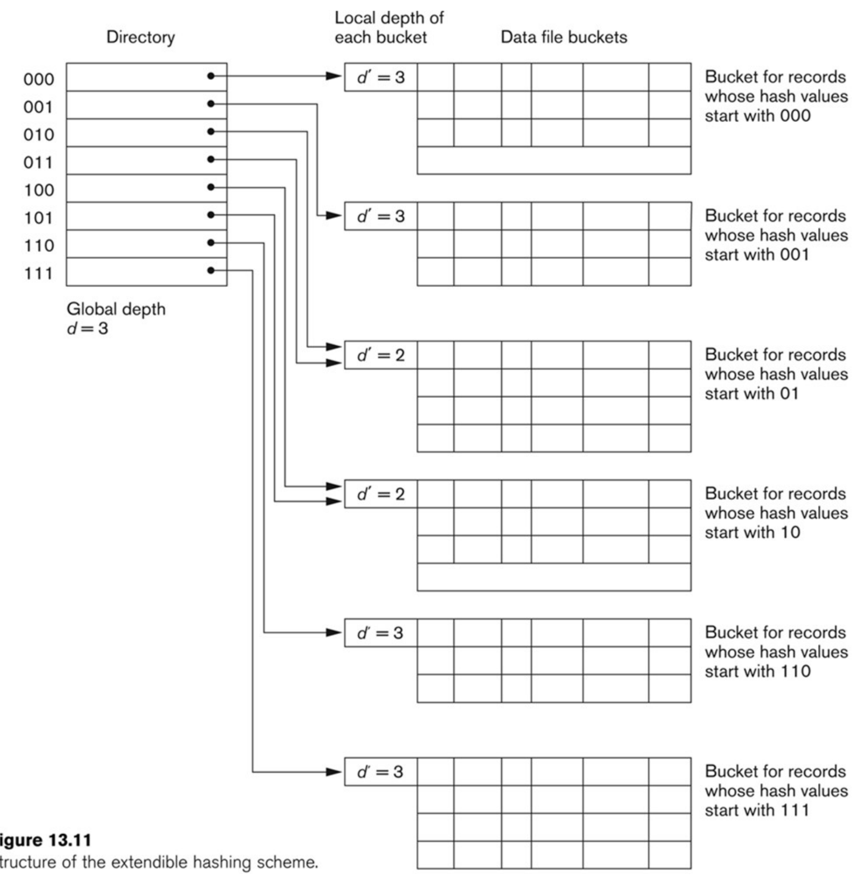


Figure 13.11
Structure of the extendible hashing scheme.

Dynamic Hashing

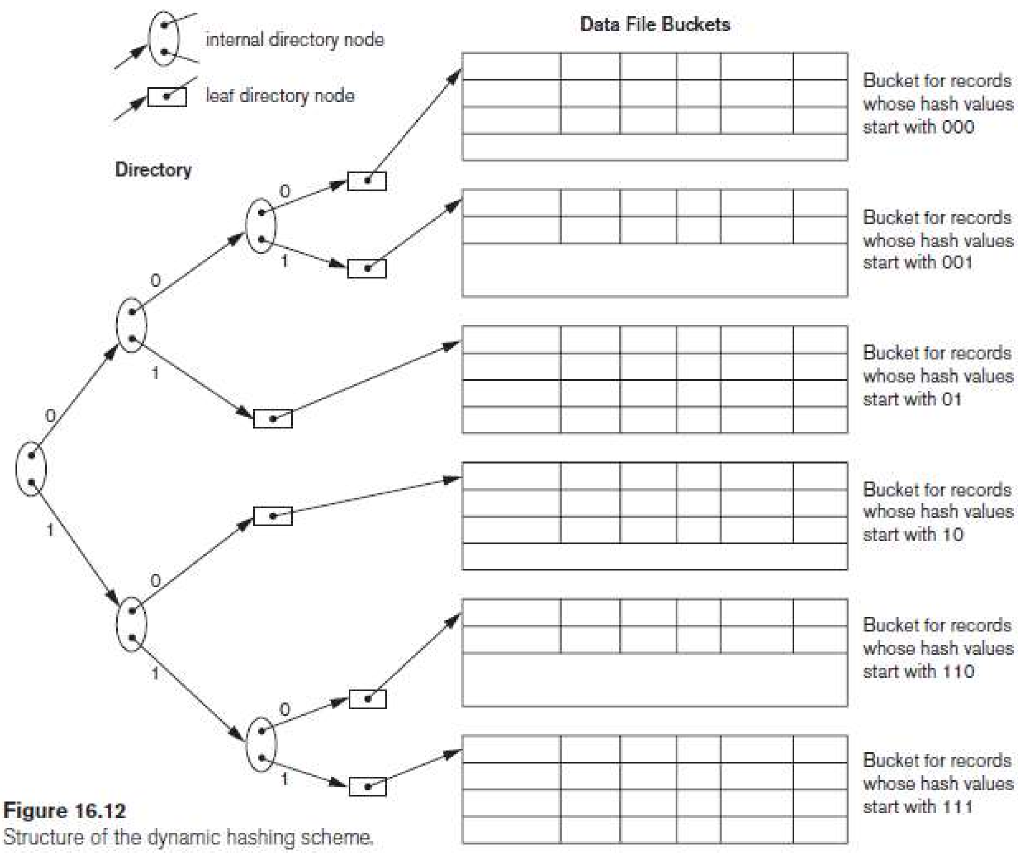


Figure 16.12
Structure of the dynamic hashing scheme.