Storage and Indexing

(courtesy : The University of Sydney)

Disks and Files

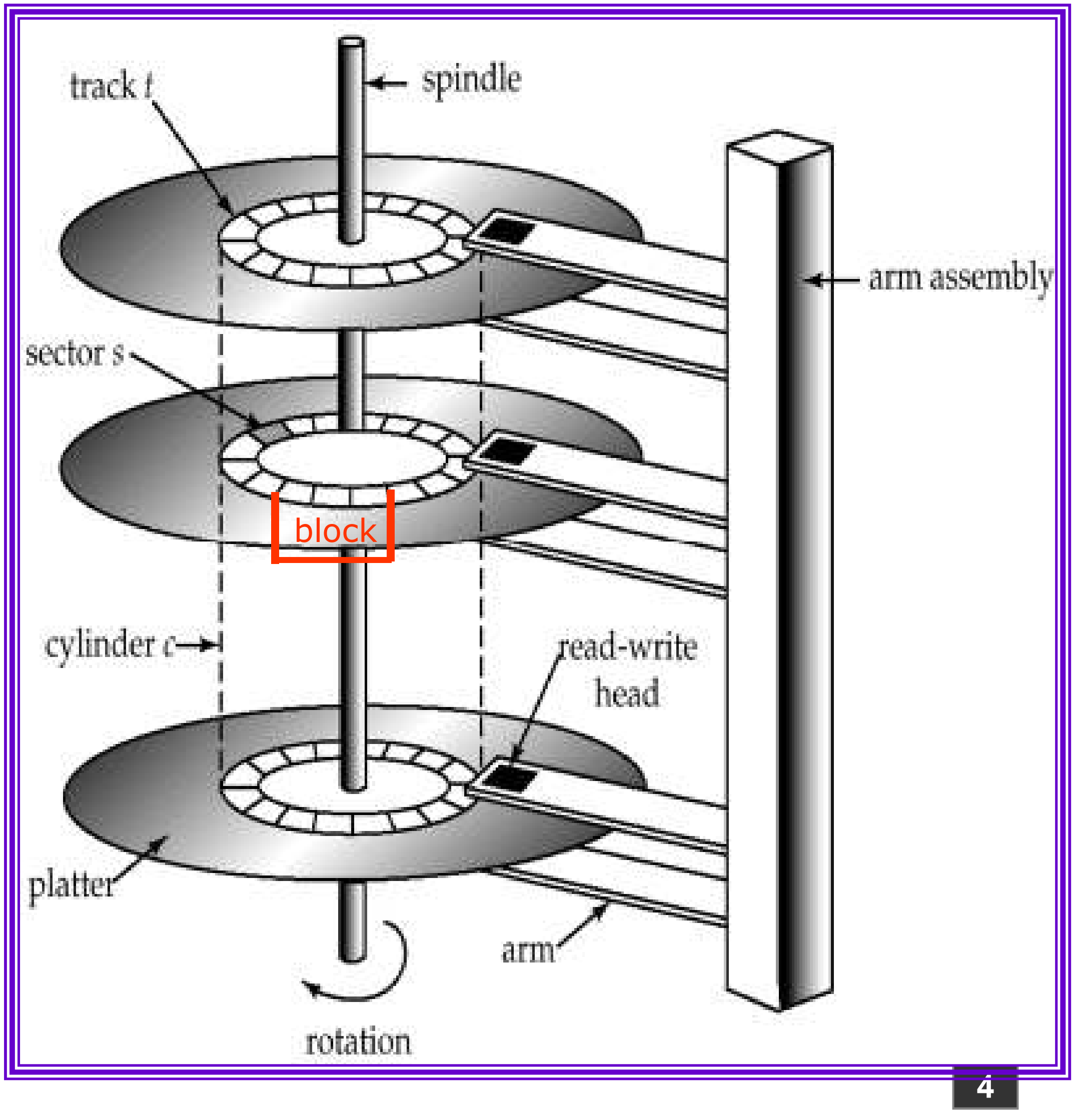
- **DBMS stores information on (“hard”) disks.**
- **This has major implications for DBMS design!**
  - READ: transfer data from disk to main memory (RAM).
  - WRITE: transfer data from RAM to disk.
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!
  - Indeed, overall performance is determined largely by the number of disk I/Os done

Disks

- **Secondary storage device of choice.**
  - Main advantage over tapes: random access vs. sequential.
- **Data is stored and retrieved in units called disk blocks or pages.**
- **Unlike RAM, time to retrieve a disk page varies depending upon location on disk.**
  - Therefore, relative placement of pages on disk has real impact on DBMS performance!
- **Trends: Disk capacity is growing rapidly, but access speed is not!**

Components of a Disk

- The platters spin (say, 120rps).
- The arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a cylinder (imaginary!).
- Only one head reads/writes at any one time.
- Block size is a multiple of sector size (which is fixed).



Accessing a Disk Page

- **Time to access (read/write) a disk block:**
  - seek time (moving arms to position disk head on track)
  - rotational delay (waiting for block to rotate under head)
  - transfer time (actually moving data to/from disk surface)
- **Seek time and rotational delay dominate.**
  - Seek time varies from about 1 to 20msec
  - Rotational delay varies from 0 to 10msec
  - Transfer rate is about 1msec per 4KB page
- **Key to lower I/O cost: reduce seek/rotation delays! Hardware vs. software solutions?**

RAID

- Data Array: arrangement of several disks
- RAID: Redundant Arrays of Independent Disks
  - Data striping + redundancy
- Data striping
  - distribute data over several disks
    - High capacity and high speed
  - the more disk, the lower reliability
- Redundancy
  - redundant information is maintained
    - high reliability by storing data redundantly, so that data can be recovered even if a disk fails

Storage Access

- A database file is partitioned into fixed-length storage units called blocks (also: page). Blocks are units of both storage allocation and data transfer.

- Database system seeks to minimize the number of block transfers between the disk and memory.
  - We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.

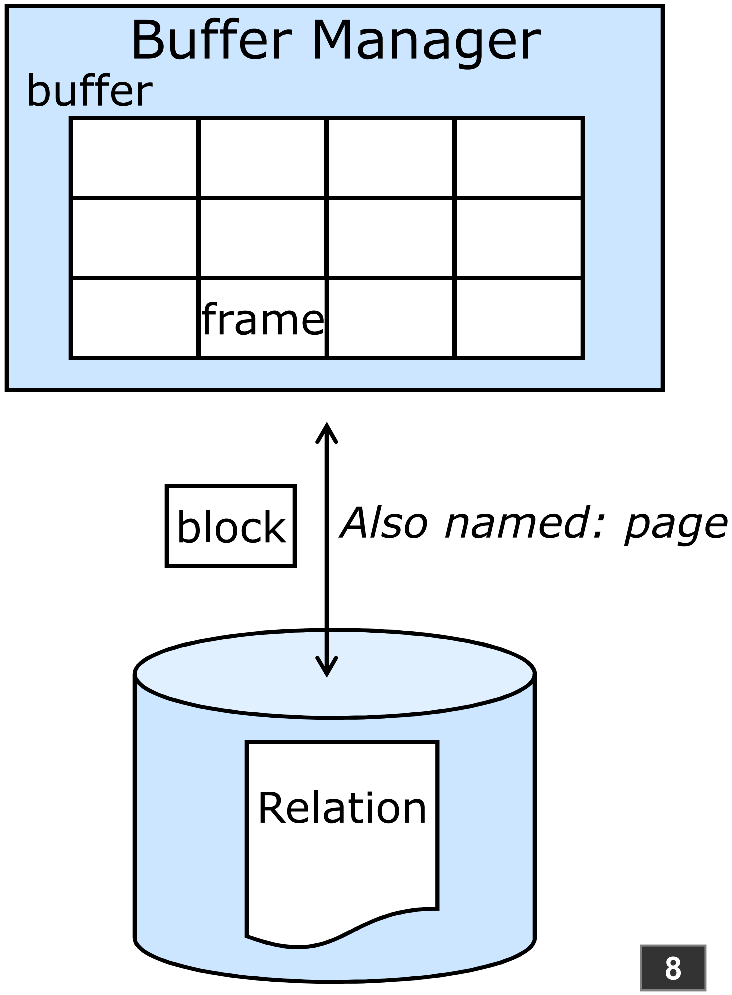
- Buffer – portion of main memory available to store copies of disk blocks.
  - Each portion is called a buffer frame

- Buffer manager – subsystem responsible for allocating buffer space in main memory.

Buffer Manager

- DBMS calls the buffer manager when it needs a block from disk.

1. If the block is already in the buffer, the address of the block in main memory is returned
2. If the block is not in the buffer,
   a. the buffer manager chooses an empty frame if possible.
   b. if all frames are used, replaces (throwing out) some other block
      - If the block that is thrown out, was modified (marked 'dirty'), it is written back to disk.
   c. Once a frame is allocated in the buffer, the buffer manager reads the block from the disk.



Also named: page

Buffer-Replacement Policies

- The algorithm by which the buffer manager decides which buffer frame to choose is called buffer-replacement policy
- Several policies available which decide based on age or usage of a frame
  - FIFO (first in, first out)
  - LFR (least-frequently-referenced)
  - LRU (least recently used), CLOCK, MRU, …
- Very common is a least recently used (LRU) strategy
  - replaces the buffer frame that has not been accessed longest
  - Most DBMS use a variant of LRU called CLOCK
- Sometimes concurrency control or recovery constrains replacement
  - A block may be pinned (not allowed to be replaced) or at times forced to be copied to disk (but it can stay in buffer)

DBMS vs. OS File System

OS does disk space & buffer mgmt: why not leave OS to manage these tasks for the DBMS?

- Differences in OS support: portability issues
- Some limitations, e.g., files can’t span disks.
- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool, force a page to disk (important for implementing CC & recovery),
  - adjust *replacement policy,* and pre-fetch pages based on access patterns in typical DB operations.

File Organisation

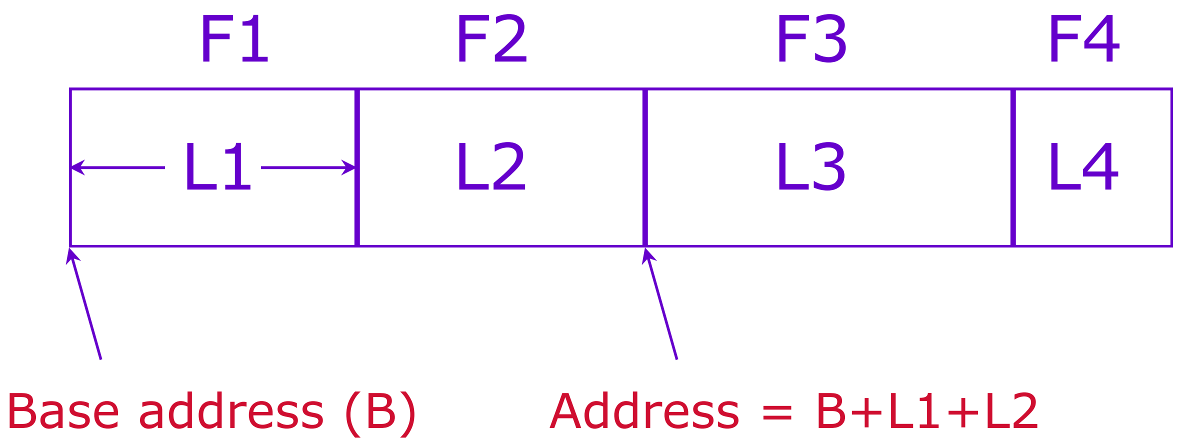
- File organization: Method of arranging a file of records on external storage.

- The database is stored as a collection of ***files***. Each file is a collection of ***records**.* A record is a sequence of ***fields***.
- Issues:
  - How to put arrange the fields in a record
  - How to arrange the records in a file
- Remember: a goal is to get fast access to given information

Record Layout

- Two approaches to structure of individual records:
  - Fixed-length records
    - All records in a single file have the same size and structure
    - Different files are used for different relations
  - Variable-length records
    - Record types that allow variable lengths for one or more fields.
    - Or, storage of multiple record types in a file.

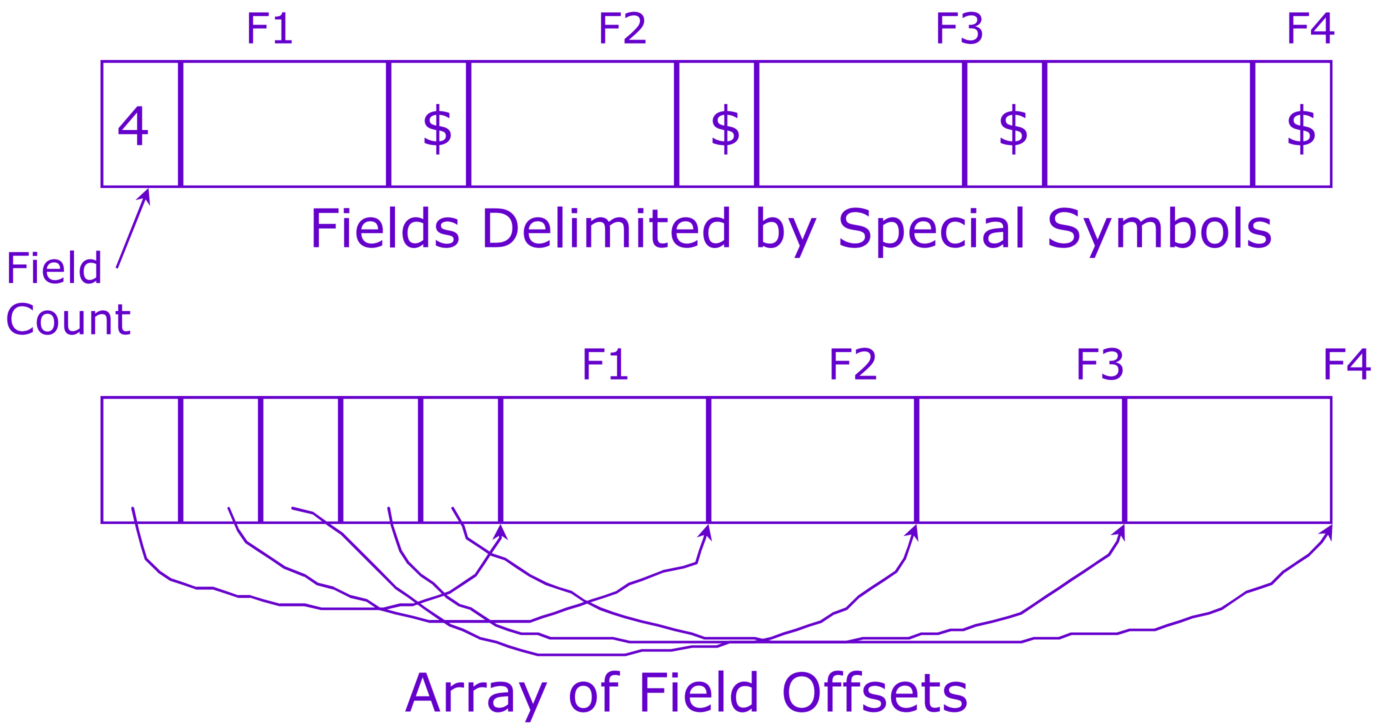
Fixed Length Records



- Information about field types same for all records in a file; stored in system catalogs.
- Finding i'th field does not require scan of record.

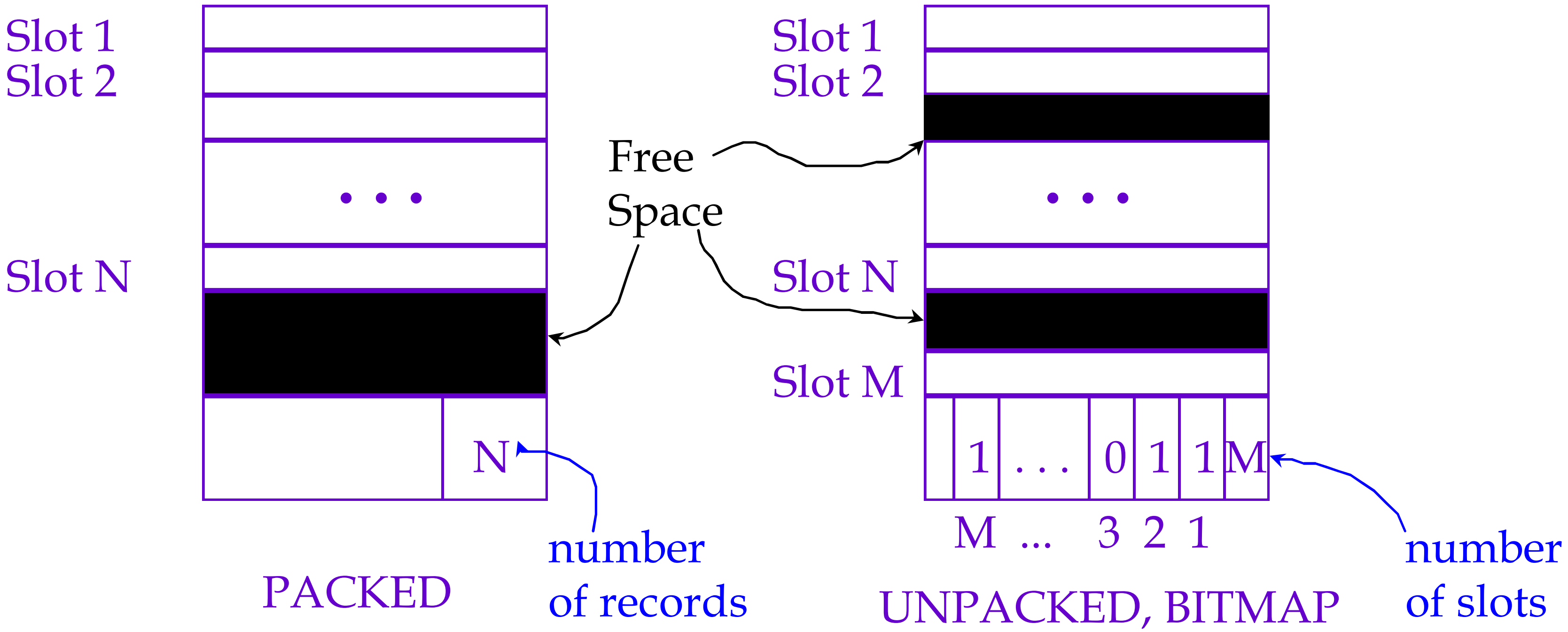
Variable Length Records

- Two alternative formats (# fields is fixed):



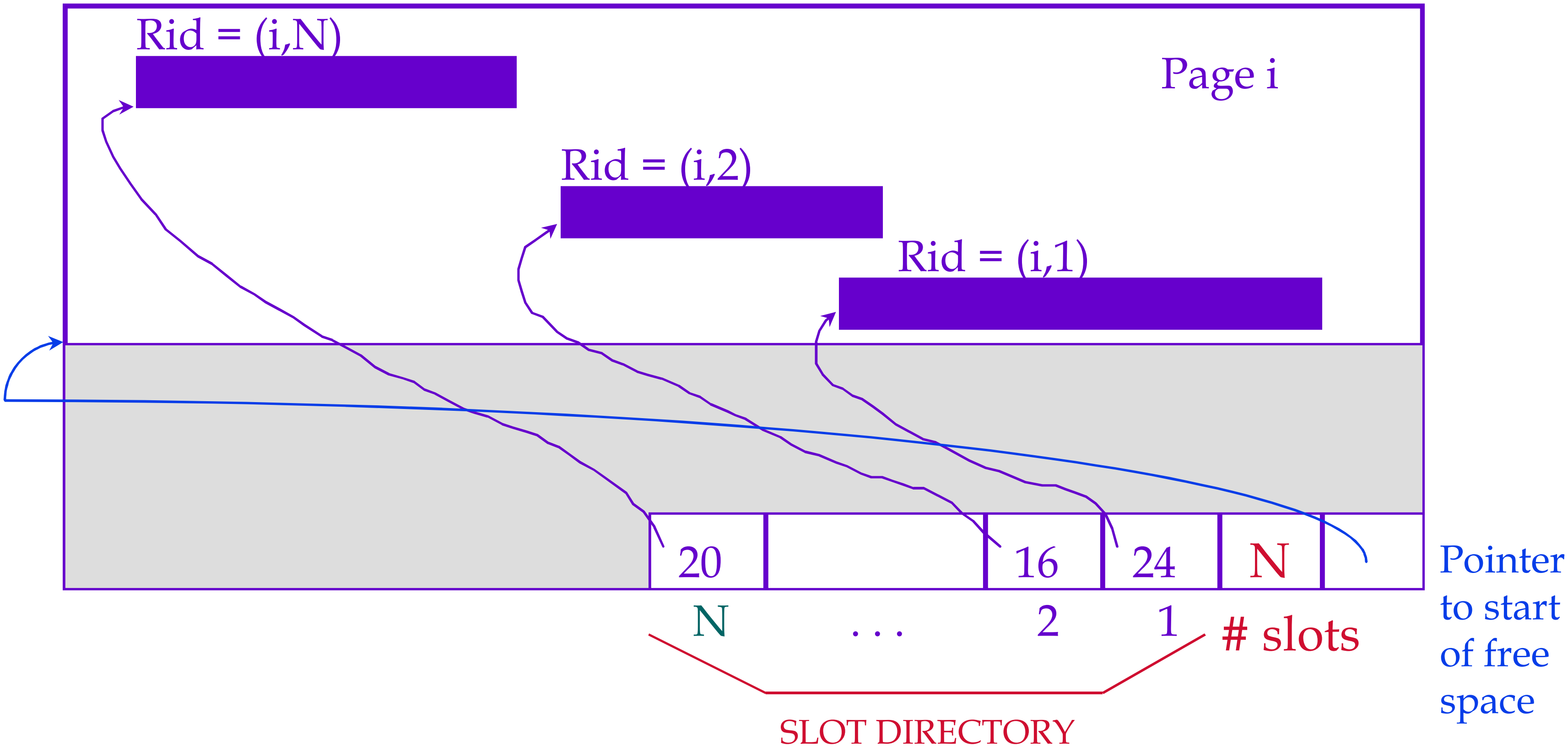
- Second offers direct access to i’th field, efficient storage of nulls (special don’t know value); small directory overhead.

Page Formats: Fixed Length Records



- *Record id = <page id, slot #>. In first alternative, moving records for free space management changes rid; may not be acceptable.*

Page Formats: Variable Length Records



- *Can move records on page without changing rid; so, attractive for fixed-length records too.*

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)

Organization of Records in Files

- Heap – a record can be placed anywhere in the file where there is space
- Sequential – store records in sequential order, based on the value of the search key of each record
- Hashing – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- Records of each relation may be stored in a separate file. In a clustering file organization records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O

Heap file

- Each record is inserted somewhere if there is space
  - Often at the end of the file
- The records are not arranged in any apparent way
- The only way to find something is to scan the whole file

Perryridge	A-201	900	Block 1
Brighton	A-217	750	
Downtown	A-110	600	
Perryridge	A-102	400	Block 2
Downtown	A-101	500	
Mianus	A-215	700	
Redwood	A-222	700	

Sequential file

- Records are kept in order based on some attribute
  - Search can be easier (eg binary search)
  - But rearrangement is needed for insertion or deletion or update of the ordering attribute

Brighton	A-217	750
Downtown	A-110	600
Downtown	A-101	500
Mianus	A-215	700
Perryridge	A-102	400
Perryridge	A-201	900
Redwood	A-222	700

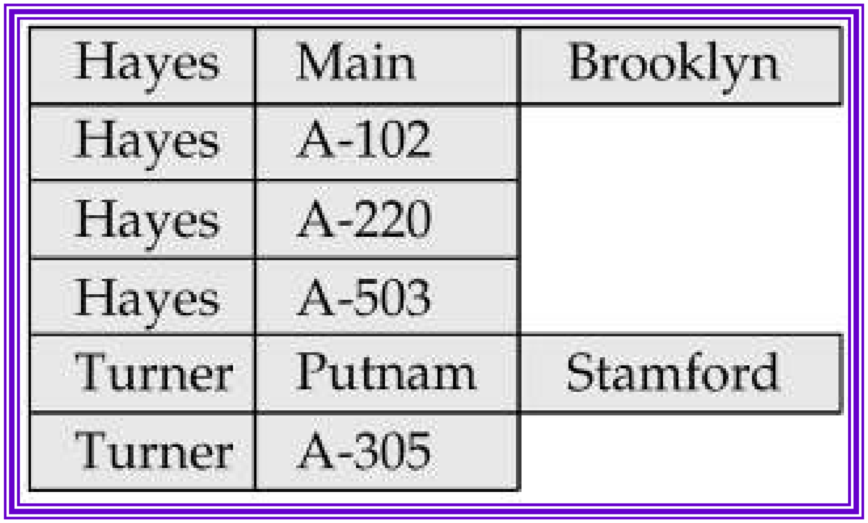
Block 1

Block 2

Account file ordered by branch

Clustering File Organization

- Simple file structure stores each relation in a separate file
- Can instead store several relations in one file using a clustering file organization
  - E.g., clustering organization of customer and depositor:



Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Turner	A-305	

Customer1 record

Depositor records
Related to customer1

Customer2 record

Depositor records
Related to customer2

  - good for join queries involving depositor and customer
  - good for queries involving one single customer and his accounts
  - bad for queries involving only customer
  - results in variable size records

Data Dictionary Storage

- Data dictionary (also called system catalog) stores metadata such as:
  - Information about relations
    - names of relations
    - names and types of attributes of each relation
    - names and definitions of views
    - integrity constraints
  - User and accounting information, including passwords
  - Statistical and descriptive data
    - number of tuples in each relation
  - Physical file organization information
    - How relation is stored (sequential/hash/…)
    - Physical location of relation
      - (operating system file names or disk addresses etc)
  - Information about indices
  - Typically stored as a set of relations (e.g. Oracle: USER_TABLES , MySQL : Information_schema)

Data Dictionary Storage

- Example
  - Attr_Cat(attr_name, rel_name, type, position)

attr_name	rel_name	type	position
attr_name	Attribute_Cat	string	1
rel_name	Attribute_Cat	string	2
type	Attribute_Cat	string	3
position	Attribute_Cat	integer	4
sid	Students	string	1
name	Students	string	2
login	Students	string	3
age	Students	integer	4
gpa	Students	real	5
fid	Faculty	string	1
fname	Faculty	string	2
sal	Faculty	real	3

Index Structures

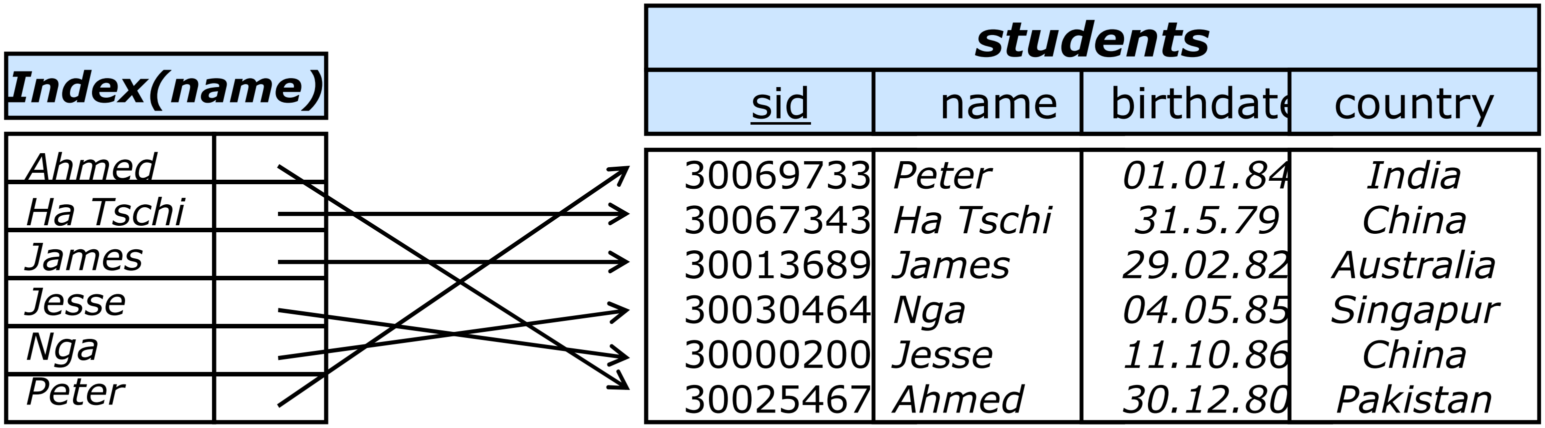
- An index on a relation is an access path to speed up selections on the search key fields for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - Search key is not the same as primary or candidate key (minimal set of fields that uniquely identify a record in a relation).

- An index consists of records (called data entries) each of which has a value for the search key eg of the form

search-key	pointer

- Index files are typically much smaller than the original file

Index Example



Index(name)

Ahmed	
Ha Tschi	
James	
Jesse	
Nga	
Peter	

students

sid	name	birthdate	country
30069733	*Peter*	*01.01.84*	*India*
30067343	*Ha Tschi*	*31.5.79*	*China*
30013689	*James*	*29.02.82*	*Australia*
30030464	*Nga*	*04.05.85*	*Singapur*
30000200	*Jesse*	*11.10.86*	*China*
30025467	*Ahmed*	*30.12.80*	*Pakistan*

- Ordered index: data entries are stored in sorted order by the search key
- Hash index: search keys are distributed uniformly across “buckets” using a “hash function”.
- Bitmap index

Alternatives for Data Entry k*

- Three alternatives for the information in the index, used to search for a value k of the search key:
  - Data record with value **k** for this attribute
  - <**k**, rid of one data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>

- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k**.
  - Examples of indexing techniques: B+ trees, hash-based structures
  - Typically, index contains auxiliary information that directs searches to the desired data entries

Alternatives for Data Entries

- Alternative 1:
  - If this is used, index structure is a file organization for data records (instead of a Heap file or sequential file).
  - At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)
  - If data records are very large, # of pages containing data entries is high. Implies size of auxiliary information in the index is also large, typically.

Alternatives for Data Entries

- Alternatives 2 and 3:
  - Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small. (Portion of index structure used to direct search, which depends on size of data entries, is much smaller than with Alternative 1.)
  - Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.

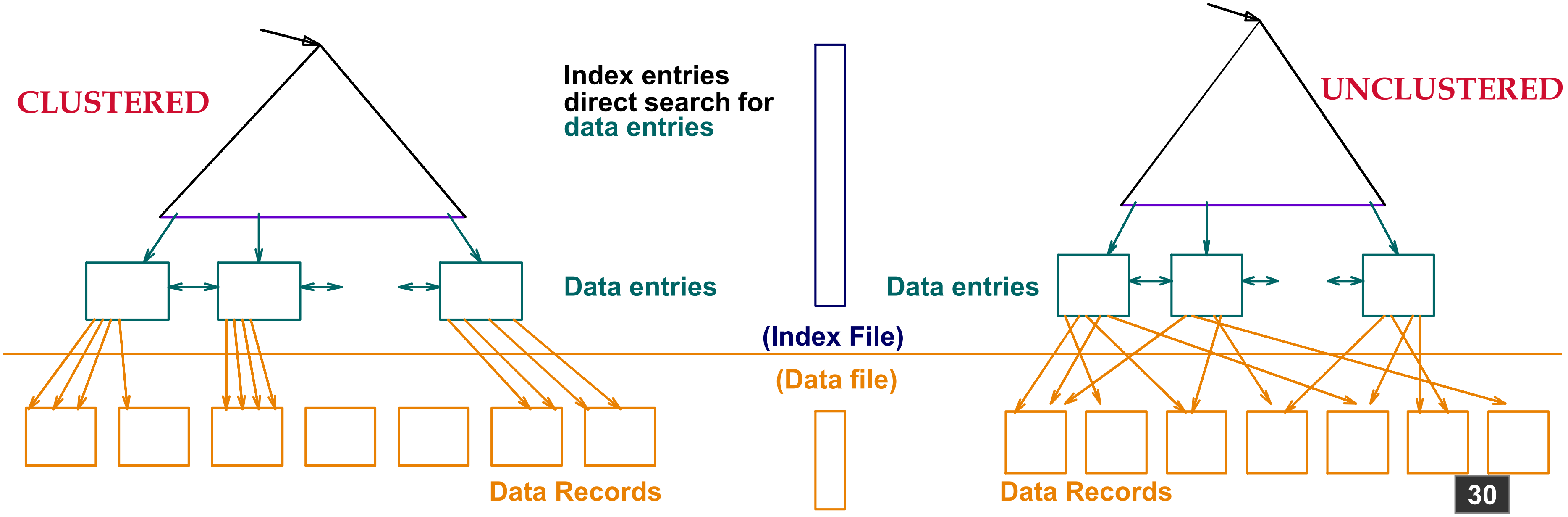
Index Classification

- *Primary* vs. *secondary*: If search key contains primary key, then called primary index.
  - *Unique* index: Search key contains a candidate key.

- *Clustered* vs. *unclustered*: If order of data records is the same as, or `close to’, order of data entries, then called clustered index.
  - Alternative 1 implies clustered; in practice, clustered also implies Alternative 1 (since sorted files are rare).
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

Clustered vs. Unclustered Index

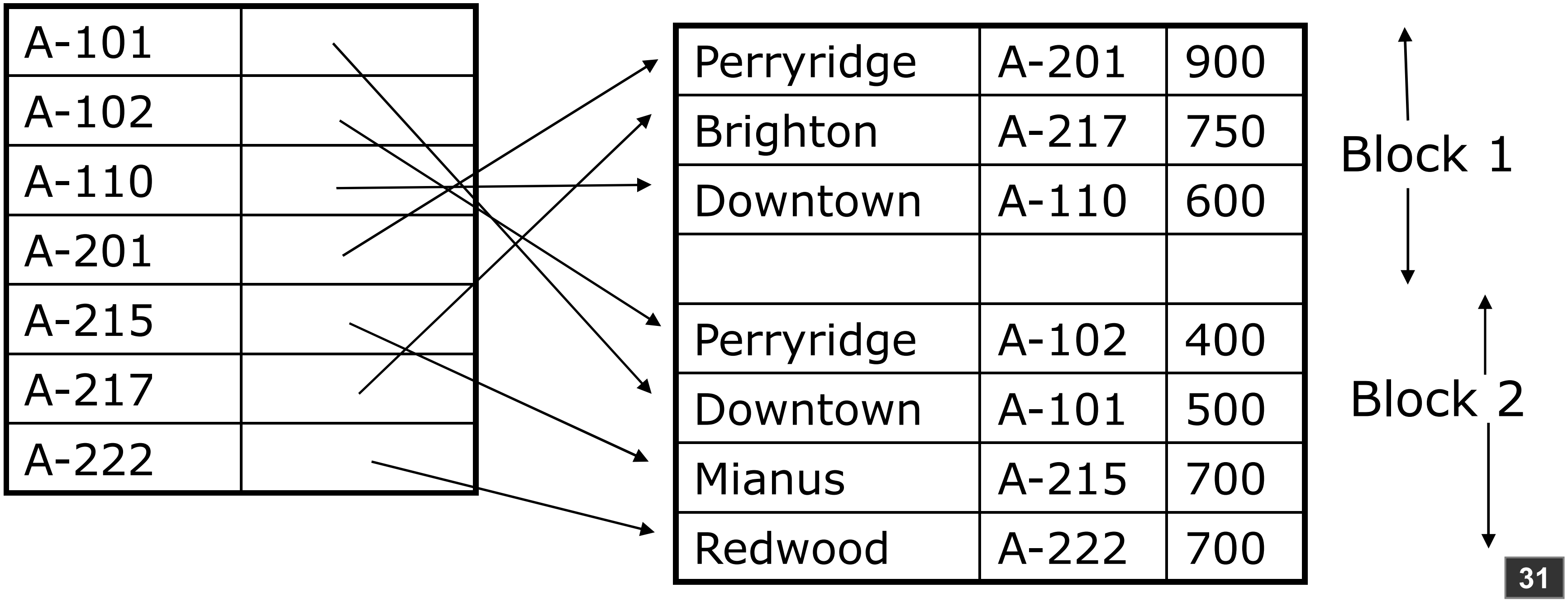
- Suppose that Alternative (2) is used for data entries
  - To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
  - Overflow pages may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)



Unclustered index for Heap file

- Data entries in index are sorted by the search key
- But the pointers go to data records that are all over the place

Index ordered by accountno

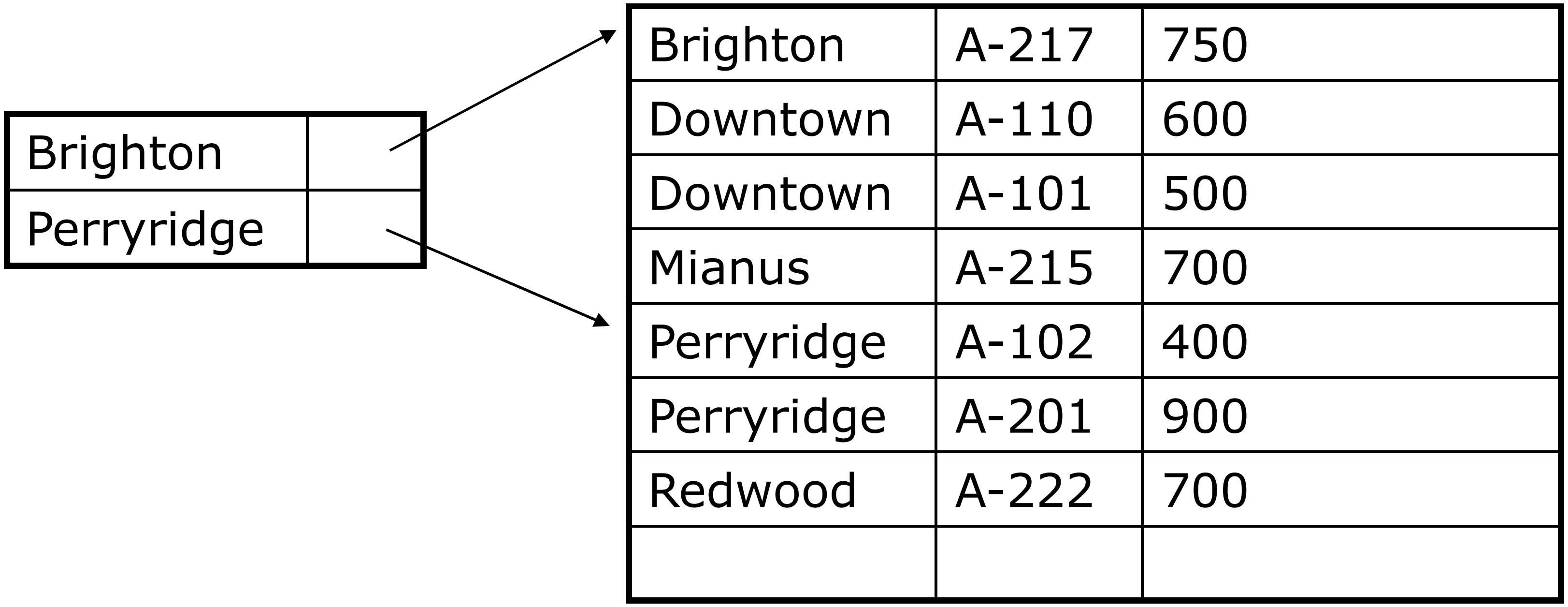


A-101	
A-102	
A-110	
A-201	
A-215	
A-217	
A-222	

Perryridge	A-201	900
Brighton	A-217	750
Downtown	A-110	600
Perryridge	A-102	400
Downtown	A-101	500
Mianus	A-215	700
Redwood	A-222	700

Clustered index for Sequential file

- Usually, a clustered index is *sparse*
  - Data entries are only used for the first data record in each data block
  - This makes the index very small compared to the data



Brighton	
Perryridge	

Brighton	A-217	750
Downtown	A-110	600
Downtown	A-101	500
Mianus	A-215	700
Perryridge	A-102	400
Perryridge	A-201	900
Redwood	A-222	700

Account file ordered by branch, index ordered by branch

Index Definition in SQL

- Create an index

  create index *<index-name>* **on** *<relation-name>*
  (<attribute-list>)

  E.g.: **create index** *b-index* **on** *branch(branch-name)*

- You can use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported

- To drop an index

  drop index *<index-name>*

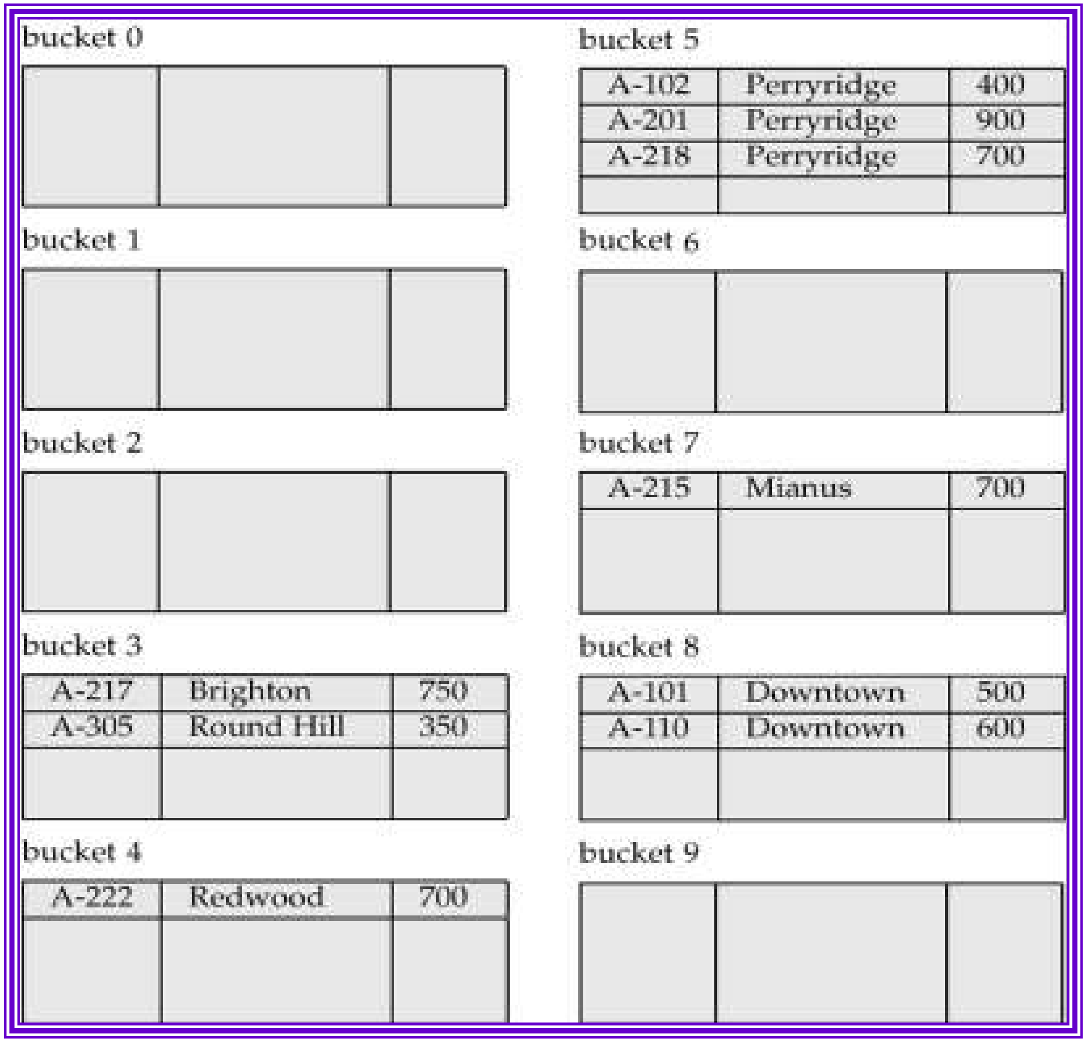
Static Hashing

- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization

- Hash file organization of account file, using branch-name as key

  e.g. h(Perryridge) = 5 h(Round Hill) = 3 h(Brighton) = 3



bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6

bucket 7

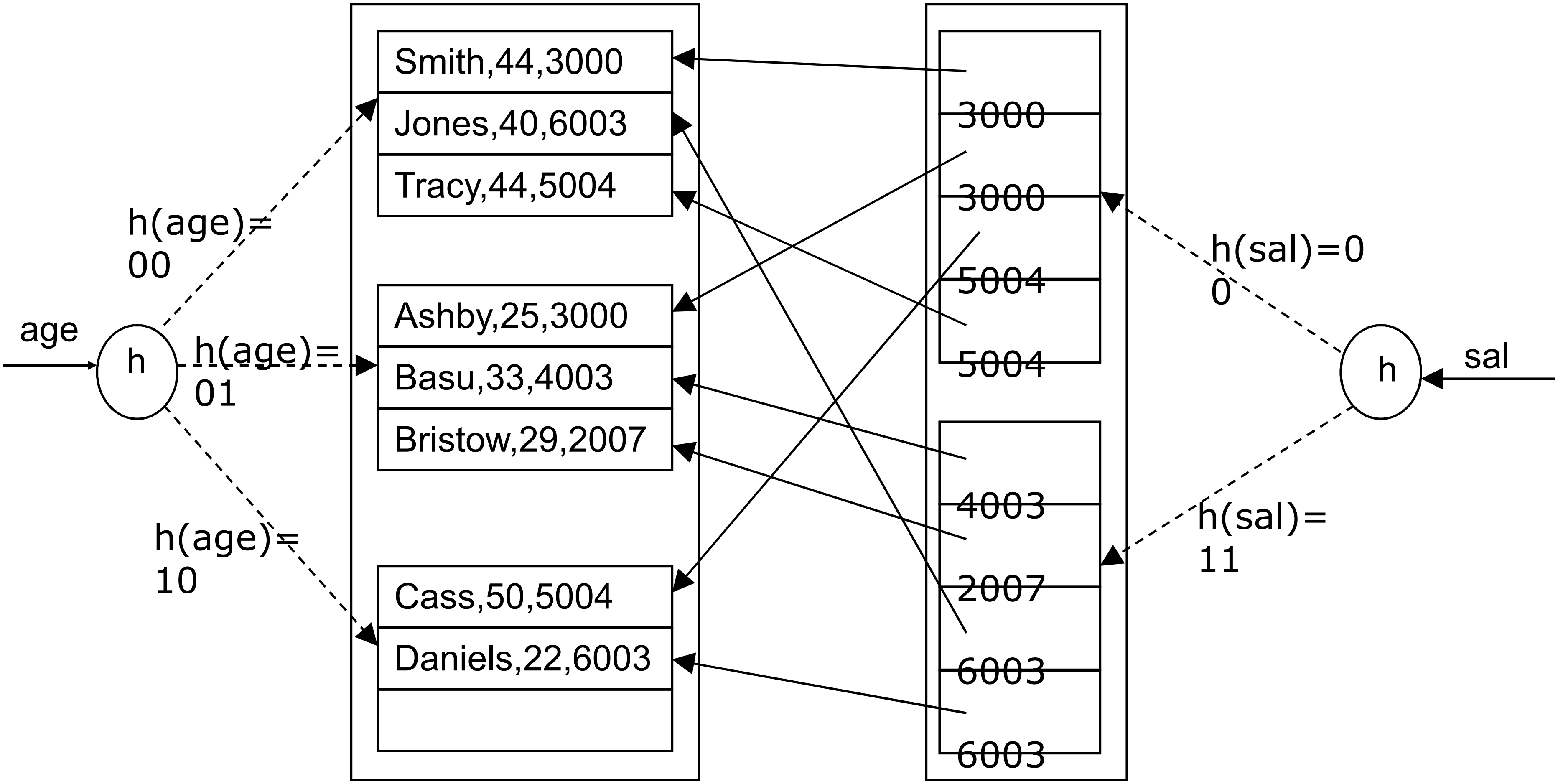
A-215	Mianus	700

bucket 8

A-101	Downtown	500
A-110	Downtown	600

bucket 9

8.3.1 Hash-based Index Examples



Smith,44,3000
Jones,40,6003
Tracy,44,5004

Ashby,25,3000
Basu,33,4003
Bristow,29,2007

Cass,50,5004
Daniels,22,6003

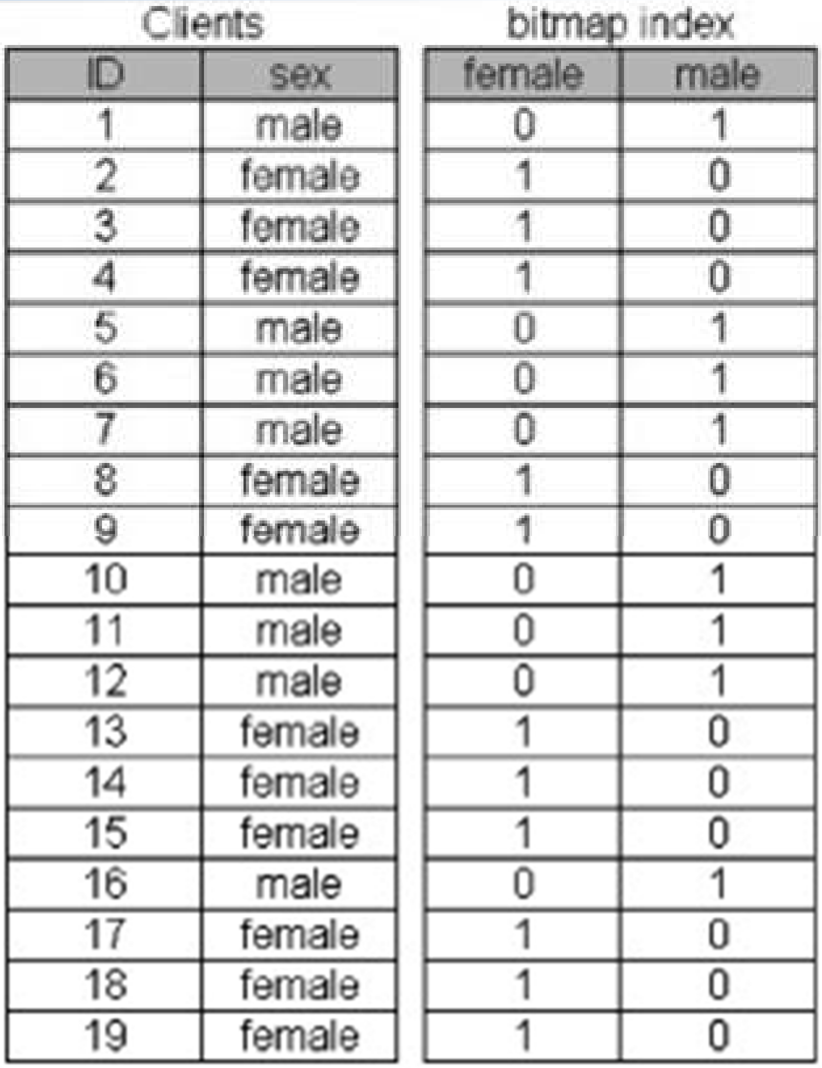
3000
3000
5004
5004

4003
2007
6003
6003

Employees file hashed on age

Index on salary

Bitmap indexing example

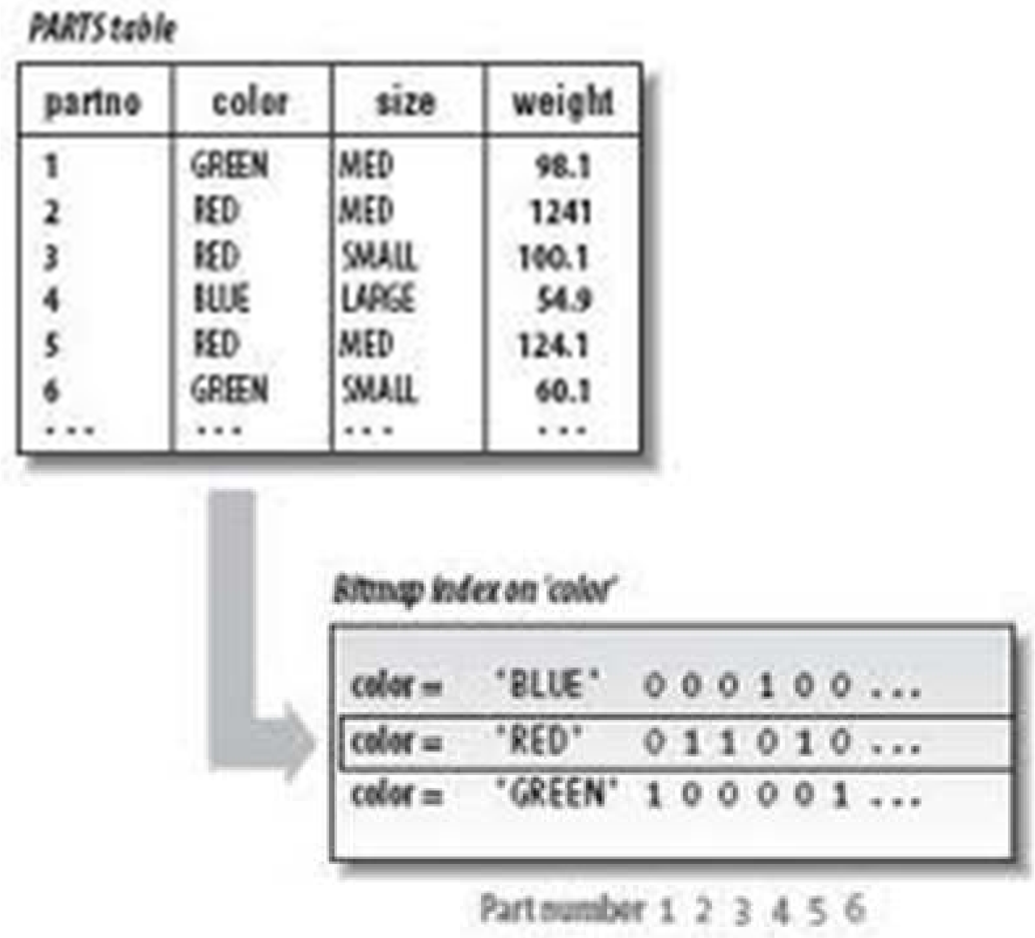


Clients

ID	sex
1	male
2	female
3	female
4	female
5	male
6	male
7	male
8	female
9	female
10	male
11	male
12	male
13	female
14	female
15	female
16	male
17	female
18	female
19	female

bitmap index

female	male
0	1
1	0
1	0
1	0
0	1
0	1
0	1
1	0
1	0
0	1
0	1
0	1
1	0
1	0
1	0
0	1
1	0
1	0
1	0



PARTS table

partno	color	size	weight
1	GREEN	MED	98.1
2	RED	MED	1241
3	RED	SMALL	100.1
4	BLUE	LARGE	54.9
5	RED	MED	124.1
6	GREEN	SMALL	60.1
...	...	...	...

Bitmap index on 'color'

color = 'BLUE' 0 0 0 1 0 0 ...

color = 'RED' 0 1 1 0 1 0 ...

color = 'GREEN' 1 0 0 0 0 1 ...

Part number 1 2 3 4 5 6

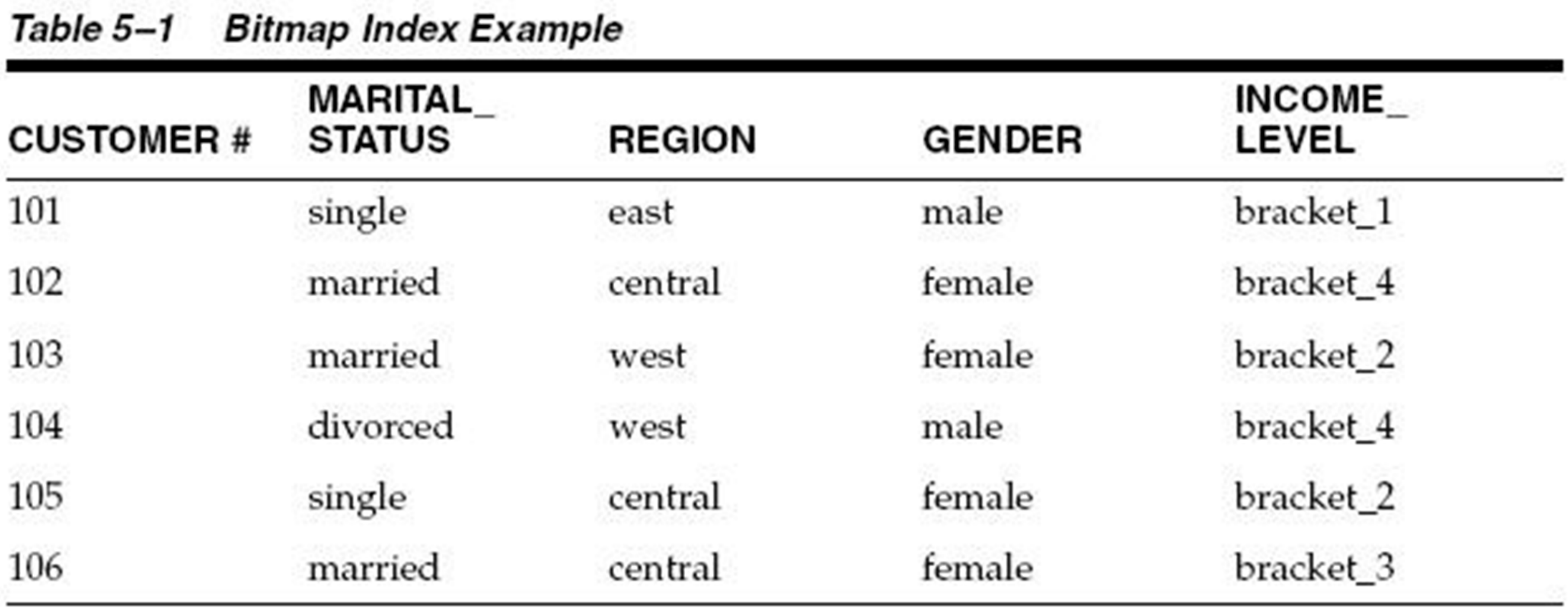
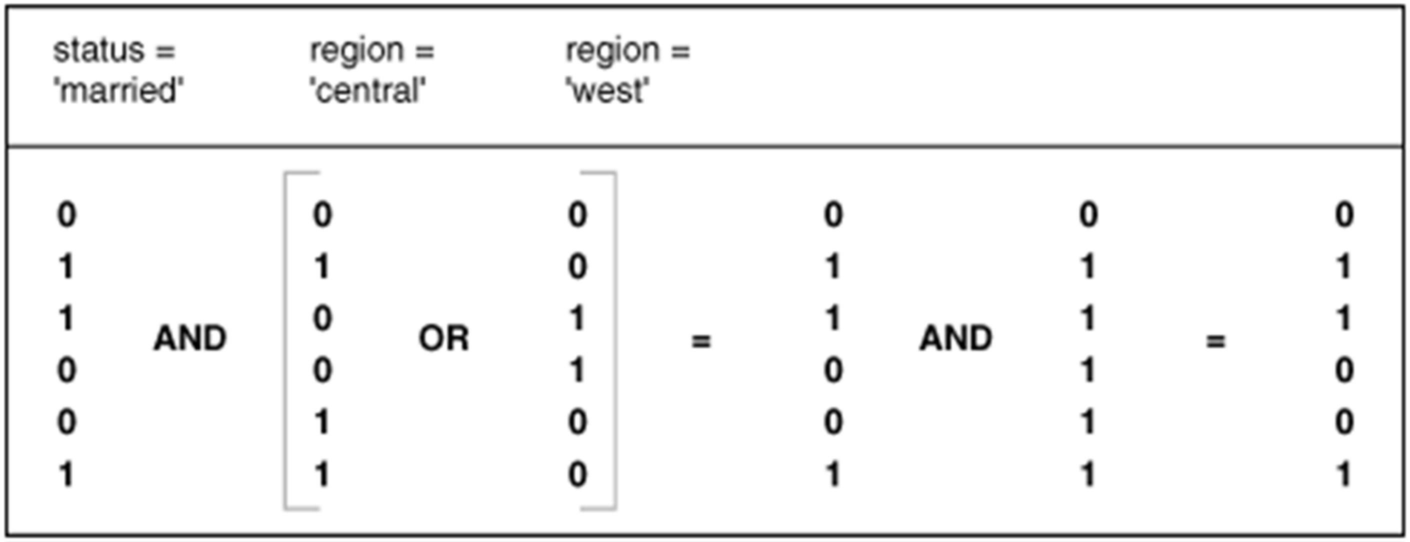
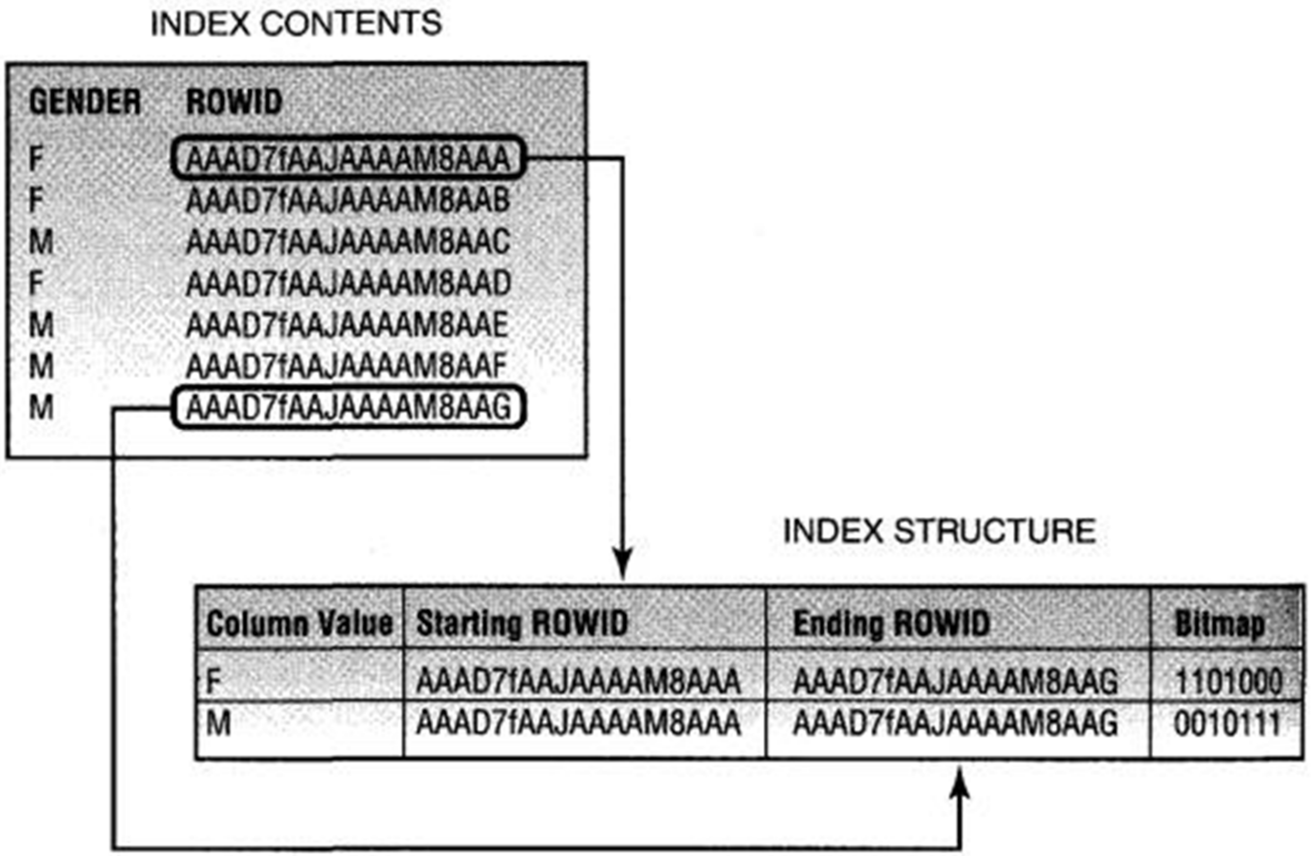


Table 5–1 Bitmap Index Example

CUSTOMER #	MARITAL_ STATUS	REGION	GENDER	INCOME_ LEVEL
101	single	east	male	bracket_1
102	married	central	female	bracket_4
103	married	west	female	bracket_2
104	divorced	west	male	bracket_4
105	single	central	female	bracket_2
106	married	central	female	bracket_3



status = 'married'		region = 'central'		region = 'west'						
0		0		0		0		0		0
1		1		0		1		1		1
1	AND	0	OR	1	=	1	AND	1	=	1
0		0		1		0		1		0
0		1		0		0		1		0
1		1		0		1		1		1



INDEX CONTENTS

GENDER	ROWID
F	AAAD7fAAJAAAAM8AAA
F	AAAD7fAAJAAAAM8AAB
M	AAAD7fAAJAAAAM8AAC
F	AAAD7fAAJAAAAM8AAD
M	AAAD7fAAJAAAAM8AAE
M	AAAD7fAAJAAAAM8AAF
M	AAAD7fAAJAAAAM8AAG

INDEX STRUCTURE

Column Value	Starting ROWID	Ending ROWID	Bitmap
F	AAAD7fAAJAAAAM8AAA	AAAD7fAAJAAAAM8AAG	1101000
M	AAAD7fAAJAAAAM8AAA	AAAD7fAAJAAAAM8AAG	0010111