Physical Data Organization and Indexing

Chapter 11

1

Access Path

- Refers to the algorithm and data structure (*e.g.*, an index) used for retrieving and storing data in a table
- The choice of an access path to use in the execution of an SQL statement has no effect on the semantics of the statement
- The choice can have a major effect on the execution time of the statement

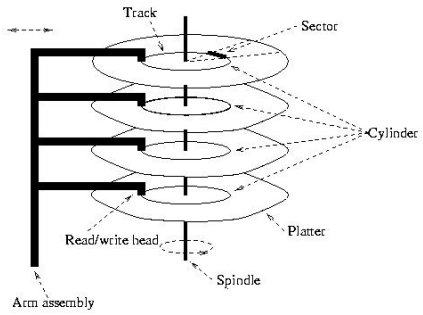
2

Disks

- Capable of storing large quantities of data cheaply
- Non-volatile
- Extremely slow compared with cpu speeds
- Performance of DBMS largely a function of the number of disk I/O operations that must be performed

3

Physical Disk Structure



4

Pages and Blocks

- Data files decomposed into *pages*
  - Fixed size piece of contiguous information in the file
  - Unit of exchange between disk and main memory
- Disk divided into page size *blocks* of storage
  - Page can be stored in any block
- Application's request for read item satisfied by:
  - Read page containing item to buffer in DBMS
  - Transfer item from buffer to application
- Application's request to change item satisfied by
  - Read page containing item to buffer in DBMS (if it is not already there)
  - Update item in DBMS buffer
  - (Eventually) copy buffer page to page on disk

5

I/O Time to Access a Page

- Seek latency- time to position heads over cylinder containing page (avg = ~10 - 20 ms)
- Rotational latency - additional time for platters to rotate so that start of block containing page is under head (avg = ~5 - 10 ms)
- Transfer time - time for platter to rotate over block containing page (depends on size of block)
- Latency = seek latency + rotational latency
- Goal - minimize average latency, reduce number of page transfers

6

Reducing Latency

- Store pages containing related information close together on disk
  - Justification: If application accesses *x*, it will next access data related to *x* with high probability
- Page size tradeoff:
  - Large page size - data related to *x* stored in same page; hence additional page transfer can be avoided
  - Small page size - reduce transfer time, reduce buffer size in main memory
  - Typical page size - 4096 bytes

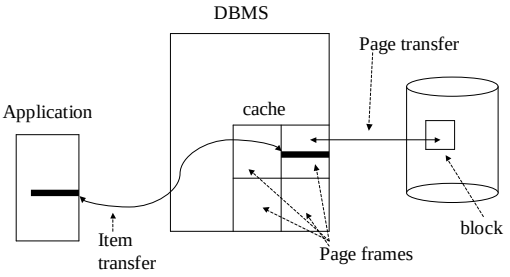
7

Reducing Number of Page Transfers

- Keep cache of recently accessed pages in main memory
  - Goal: request for page can be satisfied from cache instead of disk
  - Purge pages when cache is full
    - For example, use LRU algorithm
    - Record clean/dirty state of page (clean pages don't have to be written)

8

Accessing Data Through Cache



9

Heap Files

- Rows appended to end of file as they are inserted
  - Hence the file is unordered
- Deleted rows create gaps in file
  - File must be periodically compacted to recover space

10

Transcript Stored as a Heap File

666666	MGT123	F1994	4.0
123456	CS305	S1996	4.0
987654	CS305	F1995	2.0

page 0

717171	CS315	S1997	4.0
666666	EE101	S1998	3.0
765432	MAT123	S1996	2.0
515151	EE101	F1995	3.0

page 1

234567	CS305	S1999	4.0
878787	MGT123	S1996	3.0

page 2

11

Heap File - Performance

- Assume file contains *F* pages
- Inserting a row:
  - Access path is scan
  - Avg. *F/2* page transfers if row already exists
  - *F+1* page transfers if row does not already exist
- Deleting a row:
  - Access path is scan
  - Avg. *F/2+1* page transfers if row exists
  - *F* page transfers if row does not exist

12

Heap File - Performance

- Query
  - Access path is scan
  - Organization efficient if query returns all rows and order of access is not important
    - SELECT * FROM Transcript
  - Organization inefficient if a *unique* row is requested
    - Avg. *F/2* pages read to get information from a single page

```
SELECT  T.Grade
FROM  Transcript T
WHERE  T.StudId=12345 AND  T.CrsCode ='CS305'
          AND  T.Semester = 'S2000'
```

13

Heap File - Performance

– Organization inefficient when a subset of rows is requested: *F* pages must be read

```
SELECT  T.Course, T.Grade
FROM  Transcript T                    -- equality search
WHERE  T.StudId = 123456

SELECT  T.StudId, T.CrsCode
FROM  Transcript T                    -- range search
WHERE  T.Grade BETWEEN 2.0 AND 4.0
```

14

Sorted File

- Rows are sorted based on some attribute(s)
  - Access path is binary search
  - Equality or range query based on that attribute has cost $log_2 F$ to retrieve page containing first row
  - Successive rows are in same (or successive) page(s) and cache hits are likely
  - By storing all pages on the same track, seek time can be minimized
- Example - Transcript sorted on StudId :

```
SELECT  T.Course, T.Grade
FROM  Transcript T
WHERE  T.StudId = 123456
```

```
SELECT  T.Course, T.Grade
FROM  Transcript T
WHERE T.StudId BETWEEN
      111111 AND 199999
```

15

Transcript Stored as a Sorted File

111111	MGT123	F1994	4.0	page 0
111111	CS305	S1996	4.0	
123456	CS305	F1995	2.0	
123456	CS315	S1997	4.0	page 1
123456	EE101	S1998	3.0	
232323	MAT123	S1996	2.0	
234567	EE101	F1995	3.0	
234567	CS305	S1999	4.0	page 2
313131	MGT123	S1996	3.0	

16

Maintaining Sorted Order

- **Problem**: After the correct position for an insert has been determined, inserting the row requires (on average) *F/2* reads and *F/2* writes
- **(Partial) Solution 1**: Leave empty space in each page: *fillfactor*
- **(Partial) Solution 2**: Use overflow pages (*chains*).
  - Disadvantages:
    - Successive pages no longer stored contiguously
    - Overflow chain not sorted, hence cost no longer $log_2 F$

17

Overflow

Pointer to overflow chain

3				
111111	MGT123	F1994	4.0	page 0
111111	CS305	S1996	4.0	
111111	ECO101	F2000	3.0	
122222	REL211	F2000	2.0	
-				
123456	CS315	S1997	4.0	page 1
123456	EE101	S1998	3.0	
232323	MAT123	S1996	2.0	
234567	EE101	F1995	3.0	
-				
234567	CS305	S1999	4.0	page 2
313131	MGT123	S1996	3.0	

Chain pointer

7				
111654	CS305	F1995	2.0	page 3
111233	PSY 220	S2001	3.0	

18

Index

- Mechanism for efficiently locating row(s) without having to scan entire table
- Based on a *search key:* rows having a particular value for the search key attributes can be quickly located
- Candidate key vs search key:
  - Candidate key: *set* of attributes; *guarantees* uniqueness
  - Search key: *sequence* of attributes; *does not guarantee* uniqueness

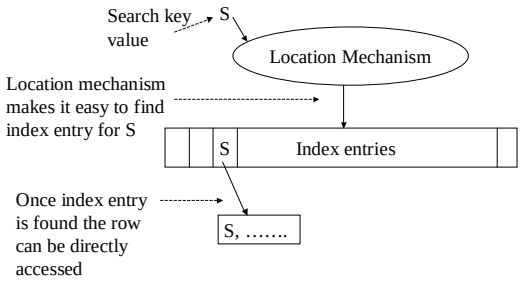
19

Index Structure

- Contains:
  - *Index entries*
    - The row itself (index and table are integrated in this case), or
    - Search key value and a pointer to a row having that value (table is stored separately in this case)
  - *Location mechanism*
    - Algorithm + data structure for locating an index entry with a given search key value
  - Index entries are stored in accordance with the search key value
    - Entries with the same search key value are stored together (hash, B tree)
    - Entries may be sorted on search key value (B tree)

20

Index Structure



21

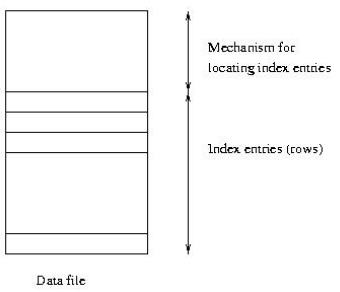
Storage Structure

- Structure of file containing a table
  - Heap file (no index, not integrated)
  - Sorted file (no index, not integrated)
  - Integrated file containing index and rows (index entries contain rows in this case)
    - ISAM
    - B^+ tree
    - Hash

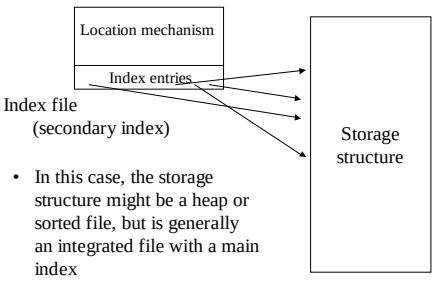
22

Integrated Storage Structure

Contains table and (main) index



Index File With Separate Storage Structure



- In this case, the storage structure might be a heap or sorted file, but is generally an integrated file with a main index

24