IF2034 Basis Data

File Structure

Tricya Widagdo
Program Studi Teknik Informatika
Institut Teknologi Bandung

Tujuan Instruksional Khusus:

Pada kuliah ini diberikan pengetahuan umum tentang berbagai struktur penyimpanan data di se

- menjelaskan berbagai jenis penyimpanan (fisik) data di *secondary storage*
- menyampaikan kelebihan dan kekurangan dari setiap metode penyimpanan

Referensi:

[GRO86] Grosshans, D., *File Systems Design & Implementation*, Prentice-Hall, Inc., 1986

[SIL99] Silberschatz, A., Korth, H. F., Sudarshan, S., *Database System Concepts*, 3rd edition, M

Storage Access

Performance Measure of Disks:

- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins.
  Consists of:
  - **Seek time** – time it takes to reposition the arm over the correct track.
    - 4 to 10 milliseconds on typical disks
  - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
    - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
  - 4 to 8 MB per second is typical

Disk-Block Access

- **Block** – a contiguous sequence of sectors from a single track
  - data is transferred between disk and main memory in blocks
  - sizes range from 512 bytes to several kilobytes
    - Typical block sizes today range from 4 to 16 kilobytes
- **File organization** – optimize block access time by organizing the blocks to correspond to how data will be accessed
  - E.g. Store related information on the same or nearby cylinders.
  - Files may get **fragmented** over time

Block size:

- Smaller blocks: more transfers from disk
- Larger blocks: more space wasted due to partially filled blocks

Files may get fragmented:

- E.g. if data is inserted to/deleted from the file
- Or free blocks on disk are scattered, and newly created file has its blocks scattered over the di
- Sequential access to a fragmented file results in increased disk arm movement

Some systems have utilities to defragment the file system, in order to speed up file access

Storage Access

- A database file is partitioned into fixed-length storage units called **blocks**. 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.
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory.

Buffer Manager

- Programs call on the buffer manager when they need a block from disk.
  1. If the block is already in the buffer, the requesting program is given the address of the block in main memory
  2. If the block is not in the buffer,
     1. the buffer manager allocates space in the buffer for the block, replacing (throwing out) some other block, if required, to make space for the new block.
     2. The block that is thrown out is written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
     3. Once space is allocated in the buffer, the buffer manager reads the block from the disk to the buffer, and passes the address of the block in main memory to requester.

Choosing the block to *throw*:

- Most operating systems replace the block **least recently used** (LRU strategy)
- Idea behind LRU – use past pattern of block references as a predictor of future references
- Queries have well-defined access patterns (such as sequential scans), and a database system ca
  - LRU can be a bad strategy for certain access patterns involving repeated scans of data
    - e.g. when computing the join of 2 relations r and s by a nested loops
      for each tuple *tr* of *r* do
        for each tuple *ts* of *s* do
          if the tuples *tr* and *ts* match …
  - Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable
- **Pinned block** – memory block that is not allowed to be written back to disk.
- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of th
- Most recently used (MRU) strategy – system must pin the block currently being processed. A
- Buffer manager can use statistical information regarding the probability that a request will ref
  - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary block
- Buffer managers also support forced output of blocks for the purpose of recovery (more in Ch

Storage Organization

- The database is stored as a collection of *files*. Each file is a sequence of *records.* A record is a sequence of *fields*.
- One approach:
  - assume record size is fixed
  - each file has records of one particular type only
  - different files are used for different relations

  This case is easiest to implement; will consider variable length records later.

E.g.: to store records of type

type *deposit* = **record**

account-number : char(10);
branch-name : char(22);
balance : real;

end

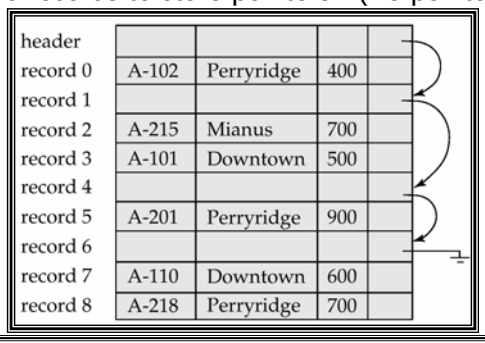
Fixed-Length Records

- Simple approach:
  - Store record *i* starting from byte $n * (i - 1)$, where *n* is the size of each record.
  - Record access is simple but records may cross blocks
    - Modification: do not allow records to cross block boundaries
- Deletion of record *I*; alternatives*:*
  - move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  - move record *n* to *i*
  - do not move records, but link all free records on a *free list*

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Free Lists

- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as pointers since they "point" to the location of a record.
- More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)



It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires

Header not only contains the address of the first deleted record. It stores a variety of information abou

Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields.
  - Record types that allow repeating fields (used in some older data models).

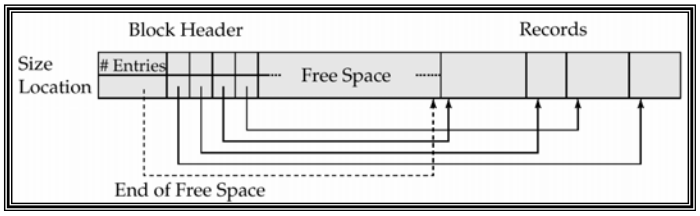
Example: account information is stored by using 1 variable record for each branch-name and for all th
There is no limit on how large a record can be.

Variable-Length Records: Byte string representation

0	Perryridge	A-102	400	A-201	900	A-218	700	⊥
1	Round Hill	A-305	350	⊥				
2	Mianus	A-215	700	⊥				
3	Downtown	A-101	500	A-110	600	⊥		
4	Redwood	A-222	700	⊥				
5	Brighton	A-217	750	⊥				

- Attach an *end-of-record* (⊥) control character to the end of each record
- Difficulty with deletion: not easy to reuse space occupied formerly by a deleted record.
  - Lead to a large number of small fragments of dist storage that are wasted
- Difficulty with growth

Variable-Length Records: Slotted Page Structure



- Slotted page header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record (within an array)
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
- (Other) Pointers should not point directly to record — instead they should point to the entry for the record in header.

A modified form of byte-string representation.

The actual records are allocated contiguously in the block, starting from the end of the block.

Variable-Length Records: Fixed-length representation

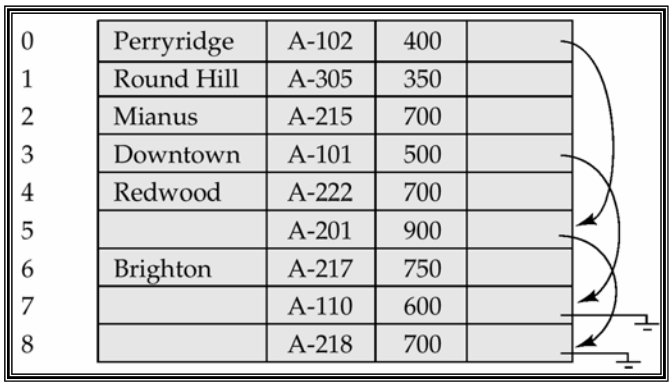
- Reserved space
  - can use fixed-length records of a known maximum length
  - unused space in shorter records filled with a null or end-of-record symbol.

0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	⊥	⊥	⊥	⊥
2	Mianus	A-215	700	⊥	⊥	⊥	⊥
3	Downtown	A-101	500	A-110	600	⊥	⊥
4	Redwood	A-222	700	⊥	⊥	⊥	⊥
5	Brighton	A-217	750	⊥	⊥	⊥	⊥

This method is useful when most record have a length close to the maximum →prevent too many was

Variable-Length Records: Fixed-length representation

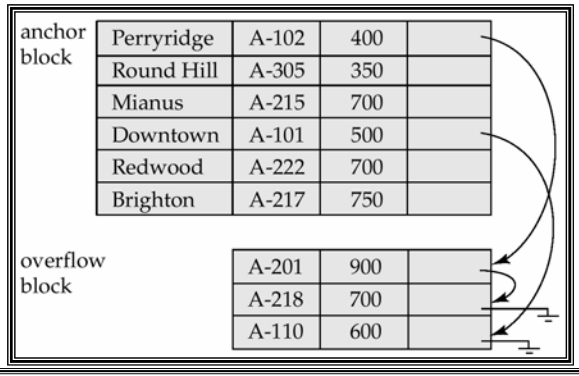
- Pointer method (List representation)
  - A variable-length record is represented by a list of fixed-length records, chained together via pointers.
  - Can be used even if the maximum record length is not known



0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

Variable-length record: Pointer Method (Cont.)

- Disadvantage to pointer structure: space is wasted in all records except the first in a chain.
- Solution is to allow two kinds of block in file:
  - Anchor block – contains the first records of chain
  - Overflow block – contains records other than those that are the first records of chairs.



Access Methods [GRO86]

- Sequential Files
  - The records are inserted into the file in chronological order
    - New records can be added only at the logical end of the file.
  - The records can only be accessed sequentially.
  - A DELETE request logically deletes the specified record from the file (by turning on a delete flag).
- Relative Files
  - Any individual record can be accessed directly or randomly.
  - A file is made up of fixed-length blocks, each block consists of fixed-length cells, each cell can hold one record.
  - A record is placed in whatever *cell* is associated with the target record.
    - The key to locate an individual record is the record number which is unique within the file (= *M*) which can be used to find Block number and Cell number of the record.
- Direct Files
  - Similar to relative files, but using the key value of the record

Access Methods (Cont.)

- Indexed Sequential Files (ISAM)
  - The records are kept in sequence by ascending key value.
  - The index structure sits on top of this ordered file.
  - Allows access sequentially and randomly.
  - A file can have more than one index.
    - The index that is ordered in the same sequence as the data is called primary index.
    - All other indexes are called secondary index.
- Indexed Files
  - Similar to ISAM, but data records are not kept in ascending key sequence.
  - Allows access sequentially and randomly.

File Organization

- **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

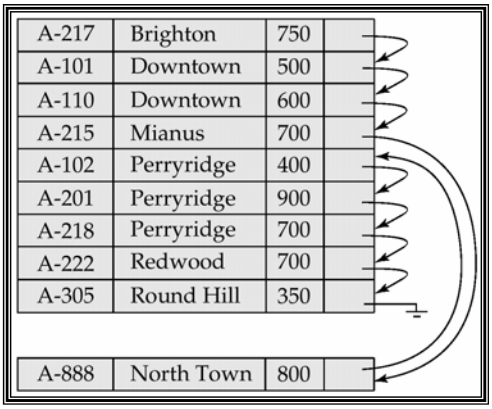
Sequential File Organization

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key

A search key is any attribute or set of attributes.

Sequential File Organization (Cont.)

- Deletion – use pointer chains
- Insertion – locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order



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	

  - good for queries involving depositor $\bowtie$ customer, and for queries involving one single customer and his accounts
  - bad for queries involving only customer
  - results in variable size records

In order to access only the records from one relation, we need to chain together all the records of that r

Data Dictionary Storage

Data dictionary (also called system catalog) stores metadata: that is, data about data, 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 name or
    - disk addresses of blocks containing records of the relation
- Information about indices

Indexing

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Index Evaluation Metrics

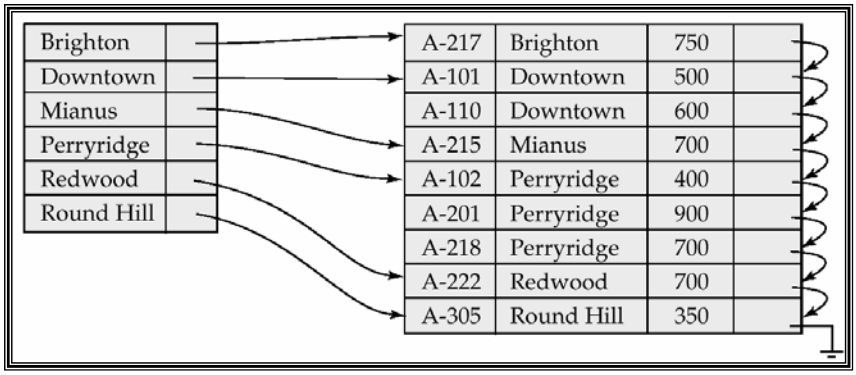
- Access types supported efficiently. E.g.,
  - records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index**.**
- Index-sequential file**:** ordered sequential file with a primary index.

Dense Index Files

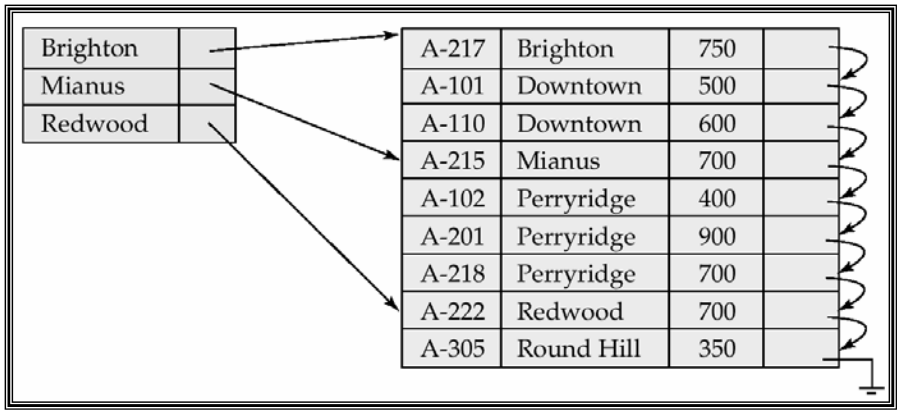
- Dense index — Index record appears for every search-key value in the file.



Sparse Index Files

- Sparse Index: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points
- Less space and less maintenance overhead for insertions and deletions.
- Generally slower than dense index for locating records.
- Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.

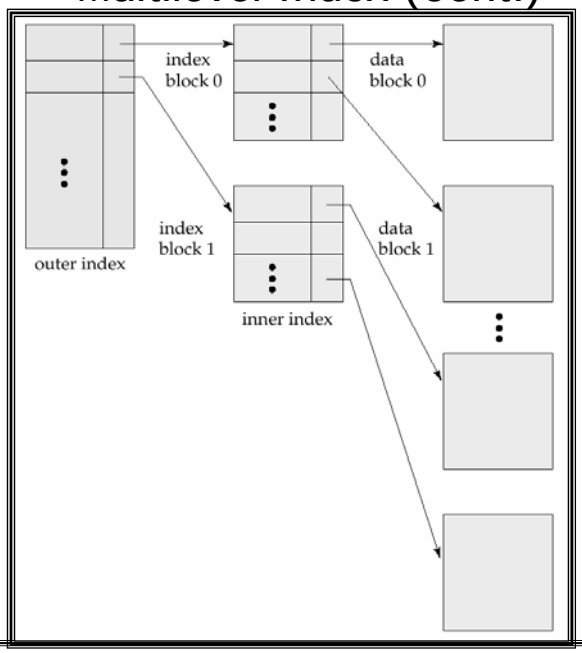
Example of Sparse Index Files



Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index (Cont.)



Index Update: Deletion

- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- Single-level index deletion:
  - Dense indices – deletion of search-key is similar to file record deletion.
  - Sparse indices – if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

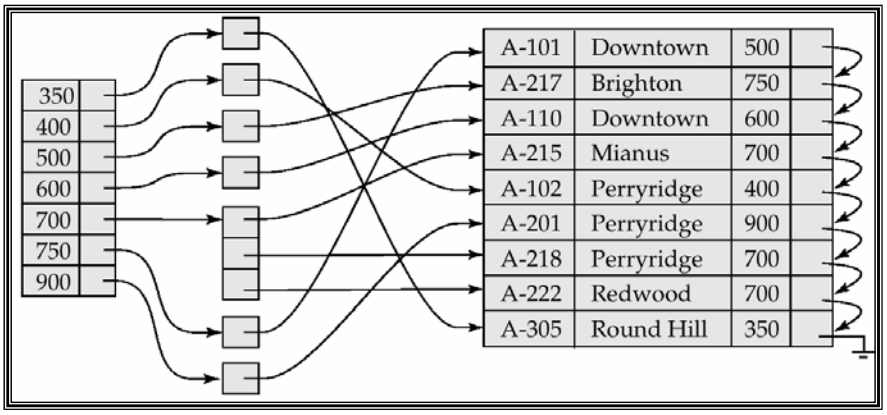
Index Update: Insertion

- Single-level index insertion:
  - Perform a lookup using the search-key value appearing in the record to be inserted.
  - Dense indices – if the search-key value does not appear in the index, insert it.
  - Sparse indices – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created. In this case, the first search-key value appearing in the new block is inserted into the index.
- Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms

Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index satisfy some condition.
  - Example 1: In the *account* database stored sequentially by account number, we may want to find all accounts in a particular branch
  - Example 2: as above, but where we want to find all accounts with a specified balance or range of balances
- We can have a secondary index with an index record for each search-key value; index record points to a bucket that contains pointers to all the actual records with that particular search-key value.

Secondary Index on ***balance*** field of ***account***



Primary and Secondary Indices

- Secondary indices have to be dense.
- Indices offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated. Updating indices imposes overhead on database modification.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - each record access may fetch a new block from disk