

Database Implementation Issues

CPSC 315 – Programming Studio

Project 1, Lecture 5

Slides adapted from those used by Jennifer Welch

Database Implementation

- Typically, we assume databases are very large, used by many people, etc.
- So, specialized algorithms are usually used for databases
 - Efficiency
 - Reliability

Storing Data

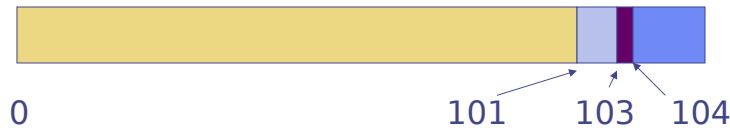
- Other terminology for implementation
 - Relation is a *table*
 - Tuple is a *record*
 - Attribute is a *field*

Storing a Record (Tuple)

- Often can assume all the fields are fixed (maximum) length.
- For efficiency, usually concatenate all fields in each tuple.
- Variable length: store max length possible, plus one bit for termination
- Store the offsets for concatenation in a schema

Example: tuple storage

- Senator
 - Name – variable character (100 + 1 bytes)
 - State – fixed character (2 bytes)
 - YearsInSenate – integer (1 byte)
 - Party – variable character (11 + 1 bytes)



More on tuples/records

- So, schema would store:
 - Name: 0
 - State: 101
 - YearsInSenate: 103
 - Party: 104
- Note that HW/efficiency considerations might give minimum sizes for each field
 - e.g. multiple of 4 or 8 bytes

Variable Length Fields

- Storing max size may be problematic
 - Usually nowhere close – waste space
 - Could make record too large for a “unit” of storage
- Store fixed-length records, followed by variable-length
- Header stores info about variable fields
 - Pointer to start of each

Record Headers

- Might want to store additional key information in *header* of each record
 - Schema information (or pointer to schema)
 - Record size (if variable length)
 - Timestamp of last modification

Record Headers and Blocks

- Records grouped into *blocks*
 - Correspond with a “unit” of disk/storage
 - Header information with record positions
 - Also might list which relation it is part of.
 - Concatenate records



Addresses

- Addresses of (pointers to) data often represented
- Two types of address
 - Location in database (on disk)
 - Location in memory
- *Translation table* usually kept to map items currently in virtual memory to the overall database.
 - Pointer swizzling: updating pointers to refer to disk vs. memory locations

Records and Blocks

- Sometimes want records to *span* blocks
 - Generally try to keep related records in the same block, but not always possible
 - Record too large for one block
 - Too much wasted space
- Split parts are called *fragments*
- Header information of record
 - Is it a fragment
 - Store pointers to previous/next fragments

Adding, Deleting, Modifying Records

- Insertion
 - If order doesn't matter, just find a block with enough free space
 - Later come back to storing tables
- If want to keep in order:
 - If room in block, just do insertion sort
 - If need new block, go to overflow block
 - Might rearrange records between blocks
 - Other variations

Adding, Deleting, Modifying Records

- Deletion
 - If want to keep space, may need to shift records around in block to fill gap created
 - Can use “tombstone” to mark deleted records
- Modifying
 - For fixed-length, straightforward
 - For variable-length, like adding (if length increases) or deleting (if length decreases)

Keeping Track of Tables

- We have a bunch of records stored (somehow).
- We need to query them (SELECT * FROM table WHERE condition)
- Scanning every block/record is far too slow
- Could store each table in a subset of blocks
 - Saves time, but still slow
- Use an *index*

Indexes

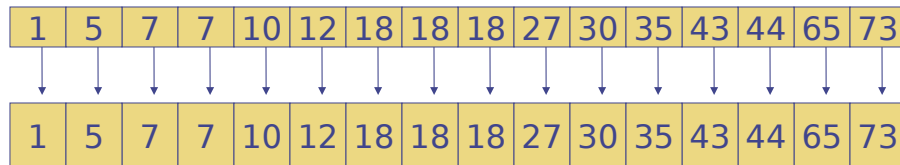
- Special data structures to find all records that satisfy some condition
- Possible indexes
 - Simple index on sorted data
 - Secondary index on unsorted file
 - Trees (B-trees)
 - Hash Tables

Sorted files

- Sort records of the relation according to field (attribute) of interest.
 - Makes it a I file
- Attribute of interest is *search key*
 - Might not be a “true” key
- Index stores (K,a) values
 - K = search key
 - a = address of record with K

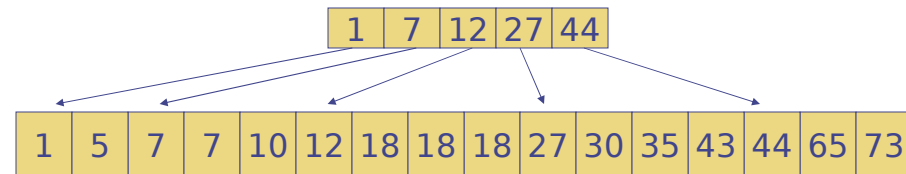
Dense Index

- One index entry per record
 - Useful if records are huge, and index can be small enough to fit in memory
- Can search efficiently and then examine/retrieve single record only



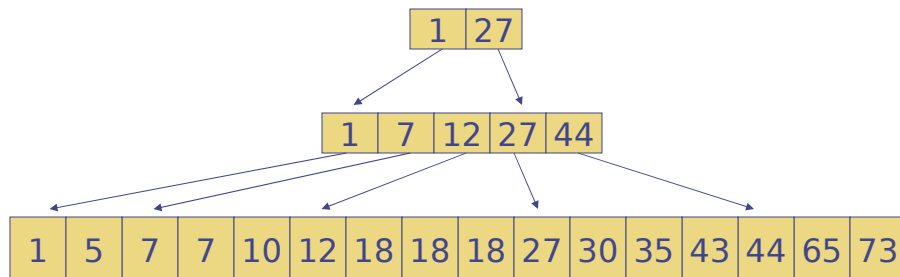
Sparse Index (on sequential file)

- Store an index for only every n records
- Use that to find the one before, then search sequentially.



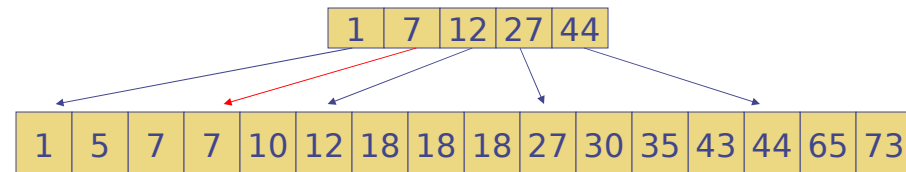
Multiple Indices

- Indices in hierarchy
- B-trees are an example



Duplicate Keys

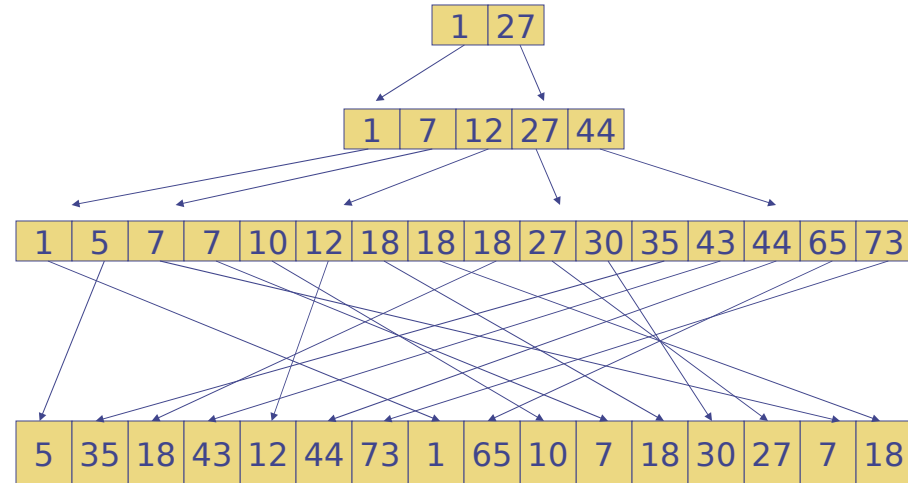
- Can cause issues, in both dense and sparse indexes, need to account for



What if not sorted?

- Can be the case when we want two or more indices on the same data
 - e.g. Senator.name, Senator.party
- Must be dense (sparse would make no sense)
- Can sort the *index* by the search key
- This *second level index* can be sparse

Example – Secondary Index



Buckets

- If there are lots of repeated keys, can use buckets
- Buckets are in between the secondary index and the data file
- One entry in index per key – points to bucket file
- Bucket file lists all records with that key

Storage Considerations

- Memory Hierarchy
 - Cache
 - Main Memory
 - Secondary storage (disk)
 - Tertiary storage (e.g. tape)
- Smaller amounts but faster access
- Need to organize information to minimize “cache misses”

Storage Considerations: Making things efficient

- Placing records together in blocks for group fetch
- Prefetching
 - Prediction algorithm
- Parallelism
 - placing across multiple disks to read/write faster
- Mirroring
 - double read speed
- Reorder read/write requests in batches

Storage Considerations Making it reliable

- Checksums
- Mirroring disks
- Parity bits
- RAID levels