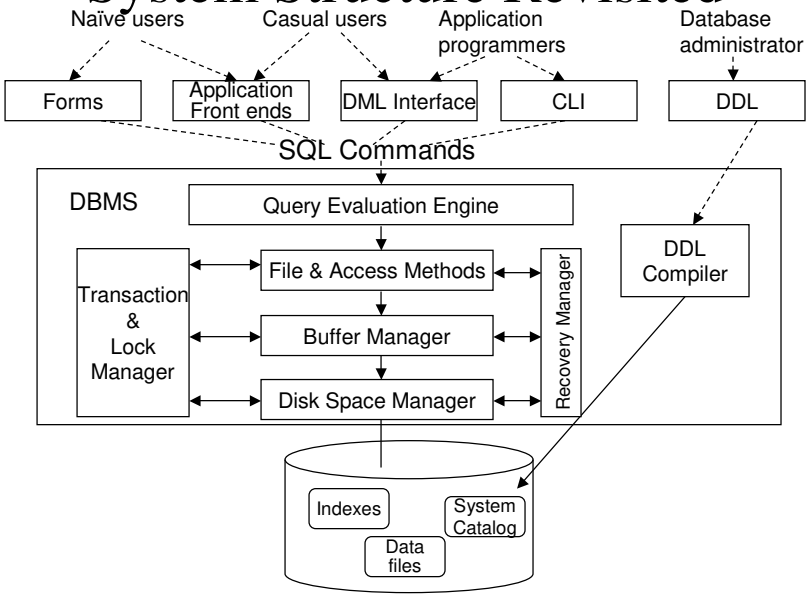
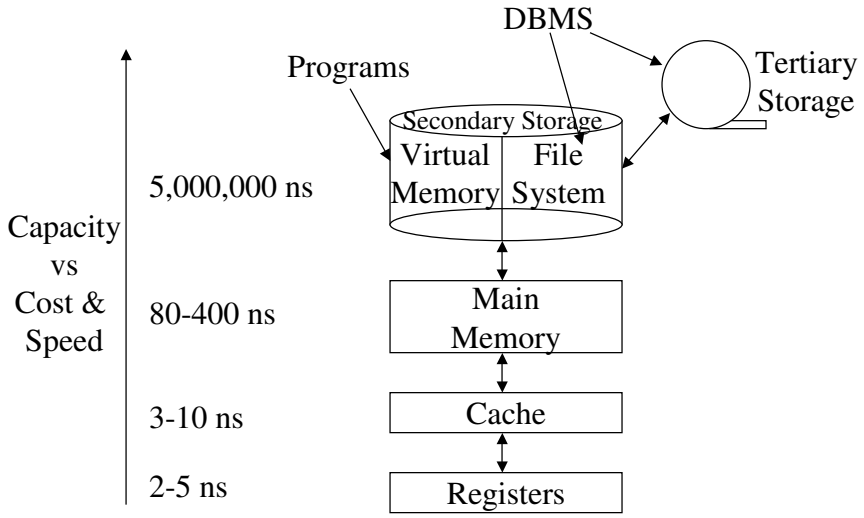
System Structure Revisited



Disk Access Process (Overly Simplifed)

- Some DBMS component indicates it wants to read record *R*
- File Manager
  - Does security check
  - Uses access structures to determine the page it is on
  - Asks the buffer manager to find that page
- Buffer Manager
  - Checks to see if the page is already in the buffer
  - If so, gives the buffer address to the requestor
  - If not, allocates a buffer frame
  - Asks the Disk Manager to get the page
- Disk Manager
  - Determines the physical address(es) of the page
  - Asks the disk controller to get the appropriate block of data from the physical address
- Disk controller instructs disk driver to do dirty job

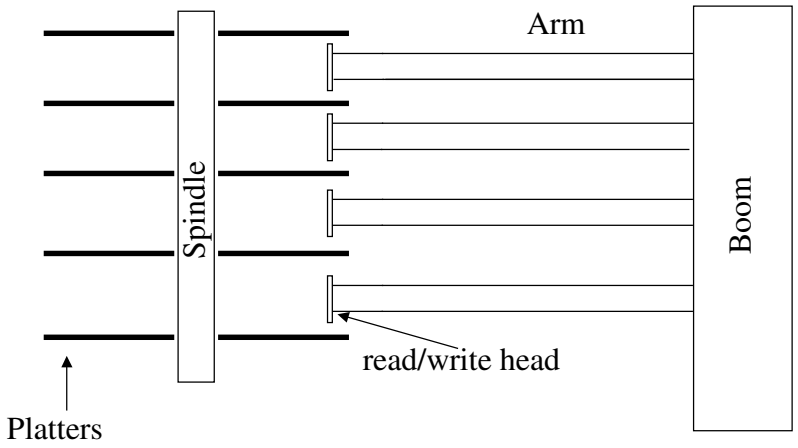
Storage Hierarchy



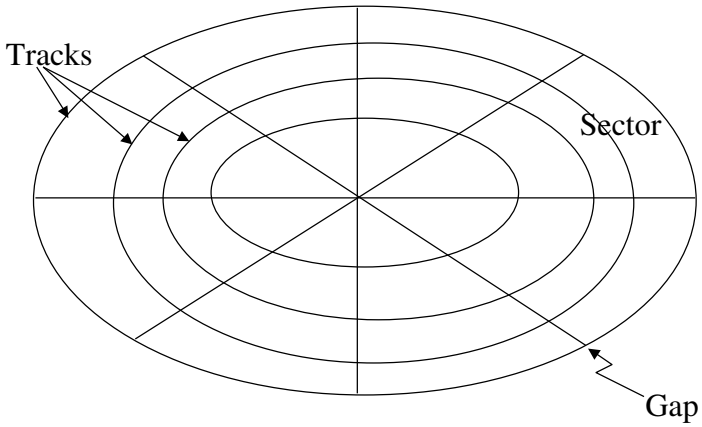
Disks

- Direct access storage devices (DASD)
  - It is possible to access data directly as well as sequentially
  - Accessing data has a lower overhead than serial devices (e.g., tapes)
- Types
  - Fixed-head hard disks
  - Removable hard disks
  - Floppy disks

Disk Drives

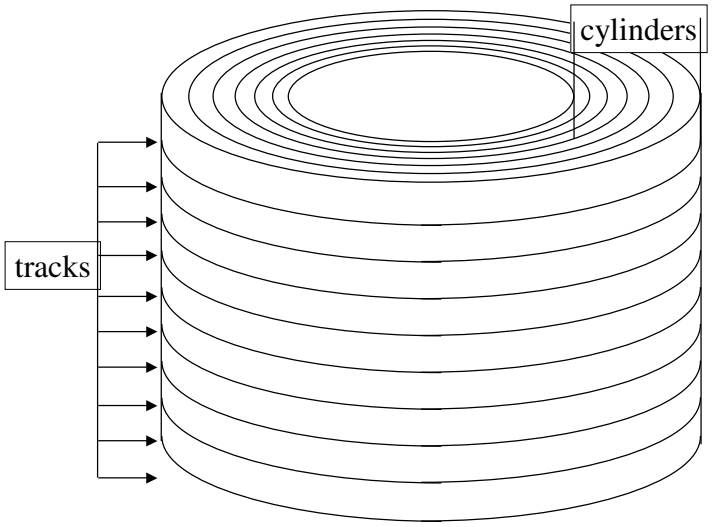


Disk Organization



Sector: the smallest addressable portion of a disk.

Disk Packs



Cost of Disk Access

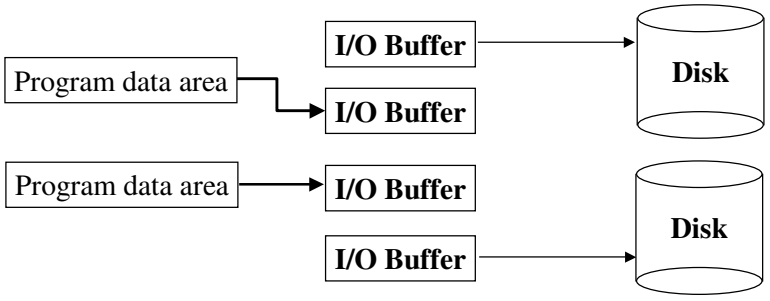
- Disk Access Time = Seek time + Rotational delay + transfer time
- Seek time: time to move access arm to correct cylinder
  - $T(\text{seek}) = S + \text{Const} * N$,
    where S = Initial time, N = # of cylinders moved
- Rotational delay: time to find correct place on track
  - on average: half a revolution
- Transfer time: time to move data
  - (# of bytes transferred)÷(# of bytes of track) * rotation time

Improving Access Time

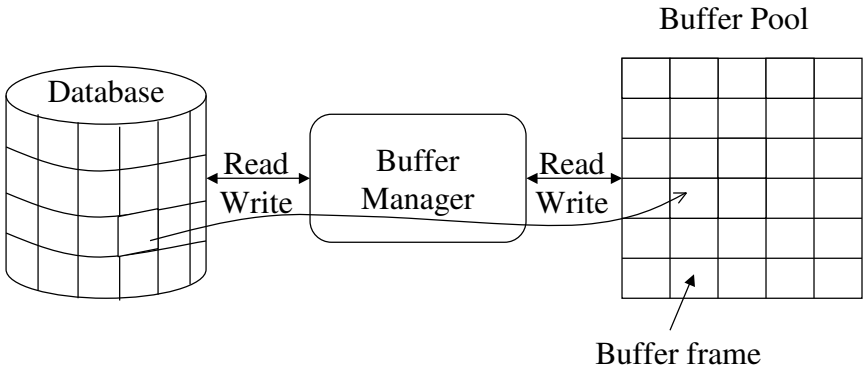
- Organize data by cylinders
- Use multiple disks - use mirroring
  - RAID
- More intelligent disk scheduling
  - Elevator algorithm
- Prefetching and buffering

Buffer Management

- Goal: reduce the number of disk accesses
- Reside in RAM
- Buffer strategies
  - multiple buffering
  - buffer pooling



Buffer Pool



Buffer Manager:

- Maintain pin_count
- Maintain dirty bit

DBMS Structures & Files

DBMS Structures	File Structures
Attribute	Field
Tuple	Record
Relation	File

Field Separation Alternatives

- Fixed length fields
  - A given field (e.g., NAME) is the same size for all records
  - Easy and fast reading but wastes space

320587	Joe Smith	SC	95	3
184923	Kathy Lee	EN	92	3
249793	Albert Chan	SC	94	3

- Length indicator at the beginning of each field
  - Also wastes space (at least 1 byte per field)
  - You have to know the length before you store

    63205879Joe Smith2SC29513
    61849238Kathy Li2EN29213
    624979311Albert Chan2SC29413

Field Separation Alternatives

- Separate fields with delimeters
  - Use white space characters (blank, new line, tab)
  - Easy to read, uses one byte per field, have to be careful in the choice of the delimeter

    |320587|Joe Smith|SC|95|3|
    |184923|Kathy Li|EN|92|**3**|
    |249793|Albert Chan|SC|94|3|
- Use keywords
  - Each field has a keyword that indicates what the field is
  - Self describing but high space overhead

    ID=320587**NAME=**Joe Smith**FACULTY=**SC**DEG=**92**YEAR=**3
    ID=184923**NAME=**Kathy Li**FACULTY=**EN**DEG=**92**YEAR=3**
    ID= 249793**NAME=**Albert Chan**FACULTY=**SC**DEG=**94**YEAR=**3

Record Organization Alternatives

- Fixed length records
  - All records are the same length

320587	Joe Smith	SC	95	3
184923	Kathy Lee	EN	92	3
249793	Albert Chan	SC	94	3

  - The number and size of fields in each record may be variable

|320587|Joe Smith|SC|95|3| ← **Padding** →
|184923|Kathy Li|EN|92|**3**| ← **Padding** →
|249793|Albert Chan|SC|94|3| ← **Padding** →

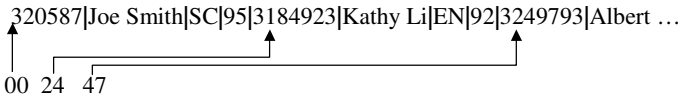
Record Organization Alternatives

- Variable Length Records
  - Fixed number of fields
    - Count the fields to detect the end of record
  - Length field at the beginning
    - Put the length of each record in front of it
    - You have to buffer the record before writing

24320587|Joe Smith|SC|95|3**23**184923|Kathy Li|EN|92|3**26**249793|Albert Chan|SC|94|3

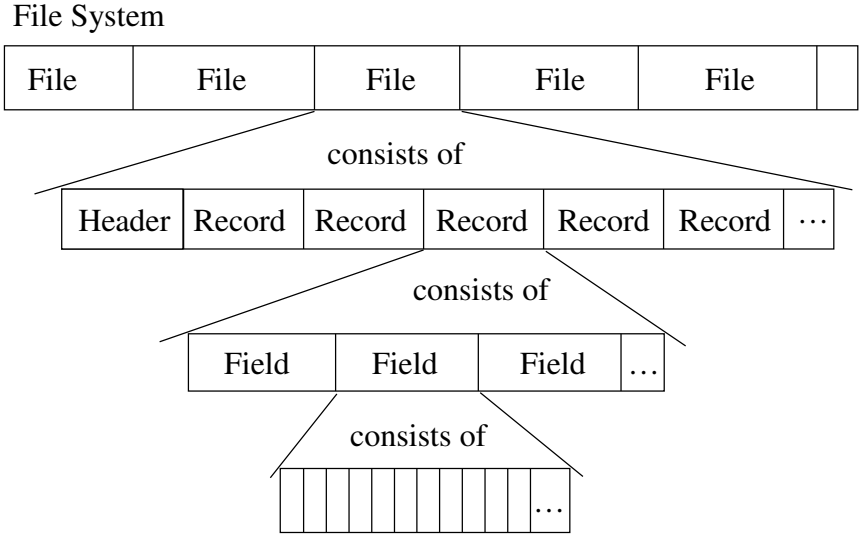
Record Organization Alternatives

- Variable Length Records (cont'd)
  - Index the beginning
    - Build a secondary index that shows where each record begins



  - End-of-record markers
    - Put a special end-of-record marker

Summary



Accessing a File

- Sequential access
  - Scan the file
  - Useful when file is small or most (all) of the file needs to be searched
  - Complexity O(n) where n is the number of disk reads
  - Block records to reduce n
  - Block size should match physical disk organization
    - multiples of sector size
- Direct access
  - Based on relative record number (RRN)
  - Record-based file systems can jump to the record directly
  - Stream-based systems calculate byte offset = RRN * record length

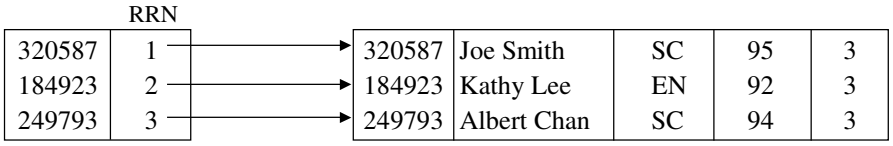
Search Problem

Find a record with a given key value

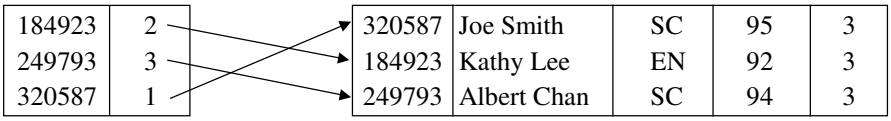
- Sequential search: O(n)
- Binary search: O(log n)
  - the file must be sorted
  - how to maintain the sorting order?
    - deleting, insertion
  - variable length records
- Sorting
  - RAM sort: read the whole file into RAM, sort it, and then write it back to disk
  - Keysort: read the keys into RAM, sort keys in RAM and then rearrange records according to sorted keys
  - Index

Keysorting

Before sorting



After sorting



Problem: Now the physical file has to be rearranged

Indexing

- A tool used to find things
  - book index, student record indexes
  - A function from keys to addresses
- A record consisting of two fields
  - key: on which the index is searched
  - reference: location of data record associated with the key
- Advantages
  - smaller size of the index file makes RAM index possible
  - binary search from files of variable length records
  - rearrange keys without moving records
  - multiple indexes
    - primary and secondary

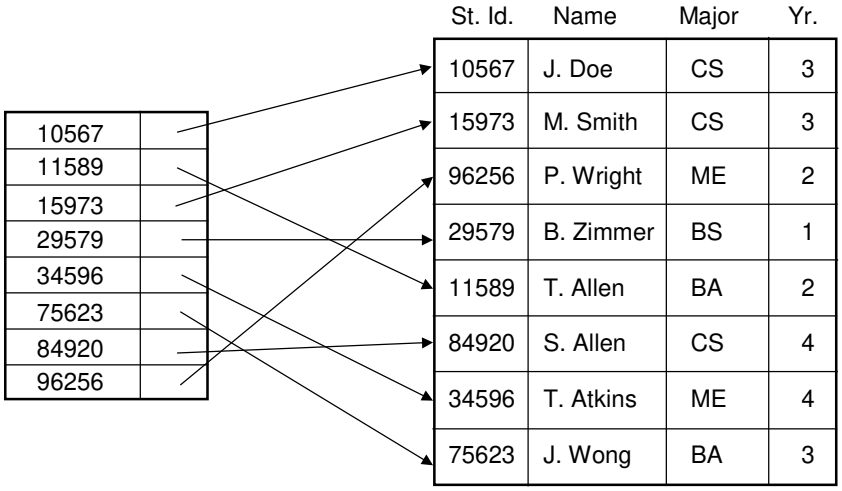
Types of Indexes

- Indexes on ordered vs unordered files
  - Dense vs sparse indexes
- Primary indexes vs secondary indexes
- Single-level vs multi-level
  - Single-level ones allow binary search on the index table and access to the physical records/blocks directly from the index table
  - Multi-level ones are tree-structured and require a more elaborate search algorithm

Single-Level Indexes on Unordered Files

- The physical records are not ordered; the index provides a physical order
- Physical files are called entry-sequenced since the order of physical records is that of entry order.
  - Append records to the physical file as they are inserted
  - Build an index (primary and/or secondary) on this file
  - Deletion/Update of physical records require reorganization of the file and the reorganization of primary index

Primary Index on Unordered Files



Operations

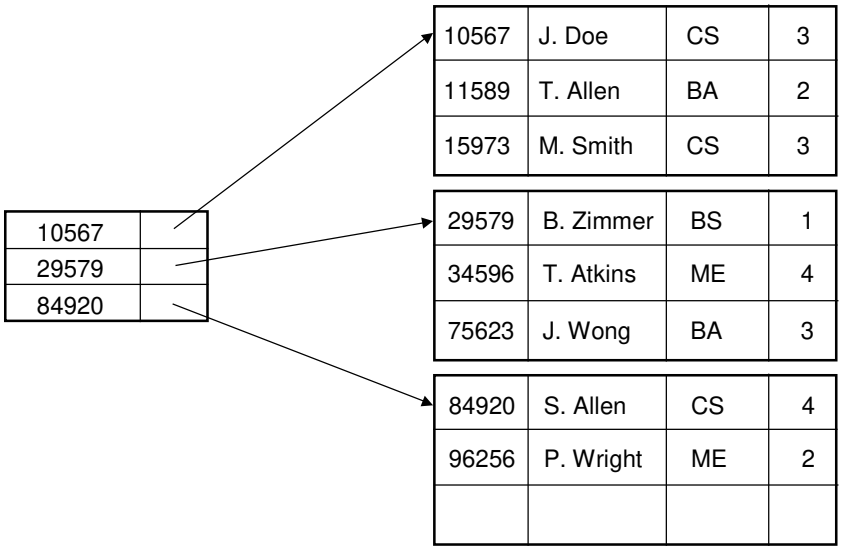
- Record addition
  - Append the record to the end; Insert a record to the appropriate place in the index
  - Requires reorganization of the index
- Record deletion
  - Delete the physical record using any feasible technique
  - Delete the index record and reorganize
- Record updates
  - If key field is affected, treat as delete/add
  - If key field is not affected, no problem.

Primary Index on Ordered Files

- Physical records may be kept ordered on the primary key
- The index is ordered but only one index record for each block
- Reduces the index requirement, enabling binary search over the values (without having to read all of the file to perform binary search).

27

Primary Index on Ordered Files



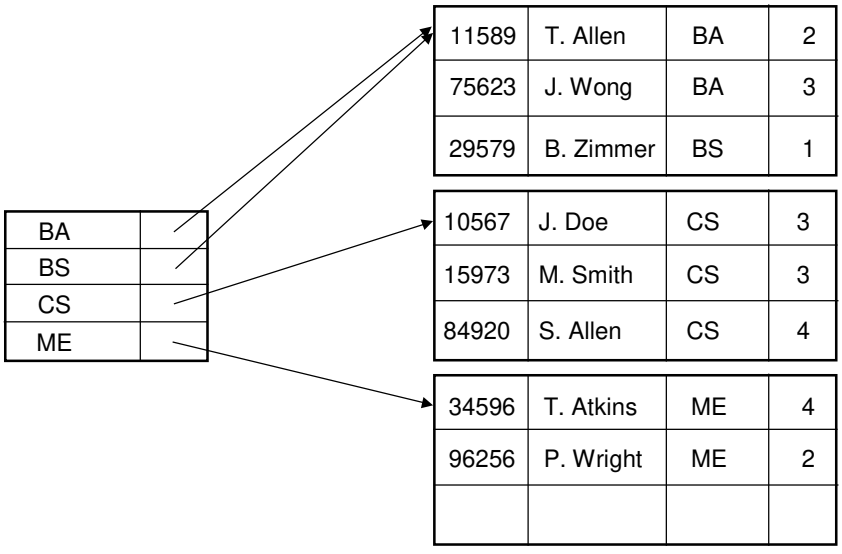
28

Secondary Index

- In addition to the primary index, establish indexes on non-key attributes to facilitate faster access
- Also called inverted file index
- Secondary indexes typically point to primary index
  - Advantage:
    - Record deletion and update causes less work
  - Disadvantage:
    - Less efficient

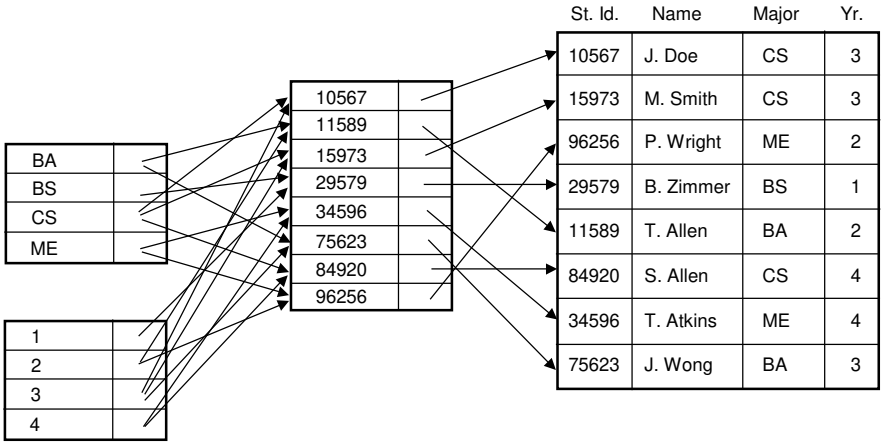
29

Clustering Index on Ordered Files



30

Secondary Index



Problems With Inverted Files

❶ Inverted file indexes cannot be maintained in main memory as the database sizes and number of keywords increase.

❷ Binary search over indexes that are stored in secondary storage is expensive.

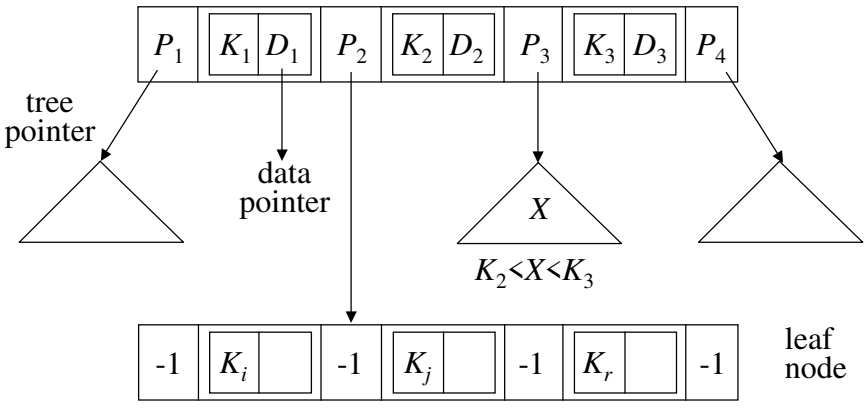
- Too many seeks and I/Os.

❸ Maintaining the indexes in sorted key order is difficult and expensive.

B-Trees

- A B-tree of order m is a paged multi-way search tree such that
  - Each page contains a maximum of m-1 keys
  - Each page, except the root, contains at least
    $\left\lceil \frac{m}{2} \right\rceil - 1$ keys
  - Root has at least 2 descedants unless it is the only node
  - A non-leaf page with k keys has k+1 descendants
  - All the leaves appear at the same level
- Build the tree bottom-up to maintain balance
  - Split & promotion for overflow during insertion
  - Redistribution & concatenation for underflow during deletion

B-Tree Structure



B-Tree Properties

Given a B-tree of order m

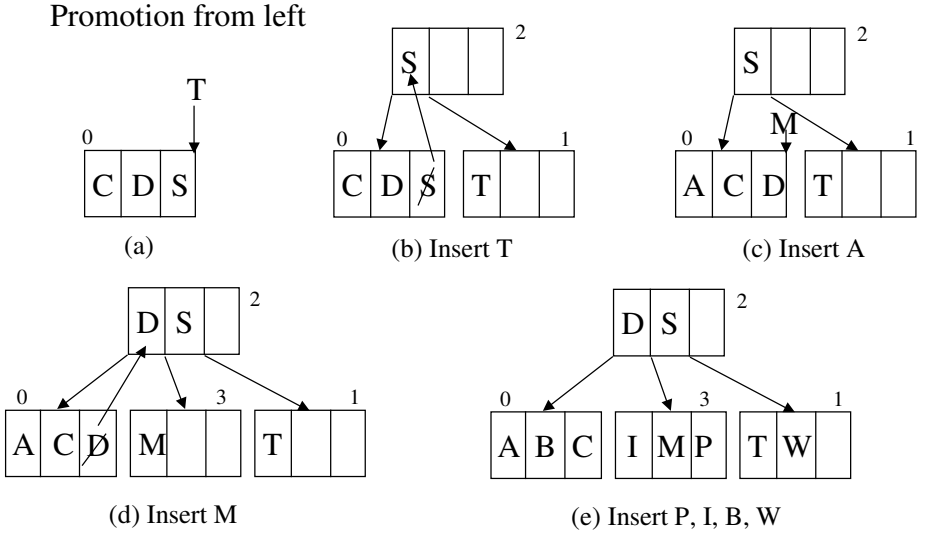
- Root page
  - $1 \leq$ keys $\leq m - 1$
  - $2 \leq$ descendents $\leq m$
- Other pages
  - $\lceil \frac{m}{2} \rceil - 1 \leq$ keys $\leq m - 1$
  - $\lceil \frac{m}{2} \rceil \leq$ descendents $\leq m$
  - $K_1 < K_1 < \ldots < K_{m\text{-}1}$
- Leaf pages
  - all at the same level; tree pointers are null

Operations

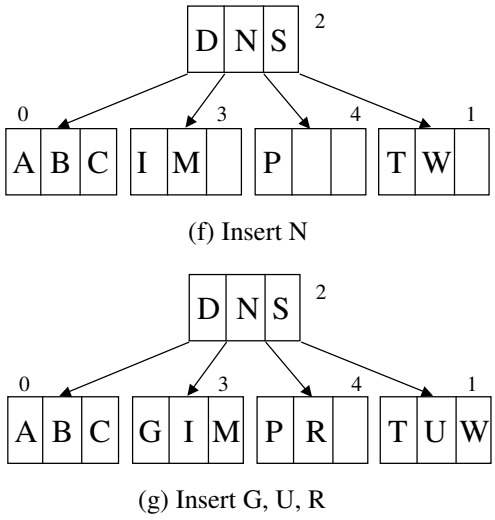
- Insertion
  - Insert the key into an appropriate leaf page (by search)
  - When overflow: split and promotion
    - Split the overflow page into two pages
    - Promote a key to a parent page
  - If the promotion in the previous step causes additional overflow, then repeat the split-promotion

Insertion Example

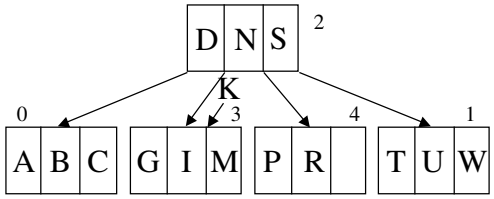
Promotion from left



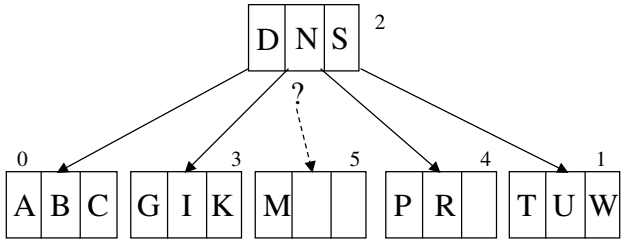
Insertion Example



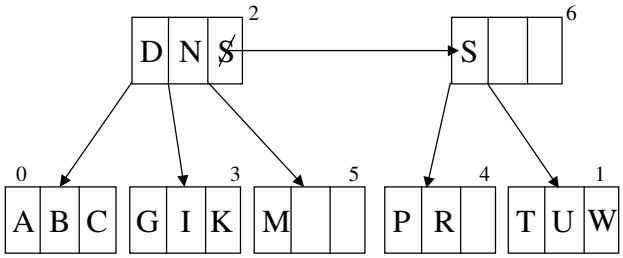
Insertion Example



(g) Insert K

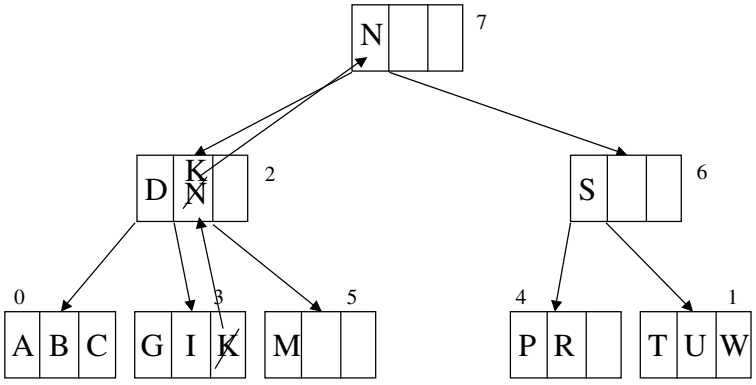


Insertion Example



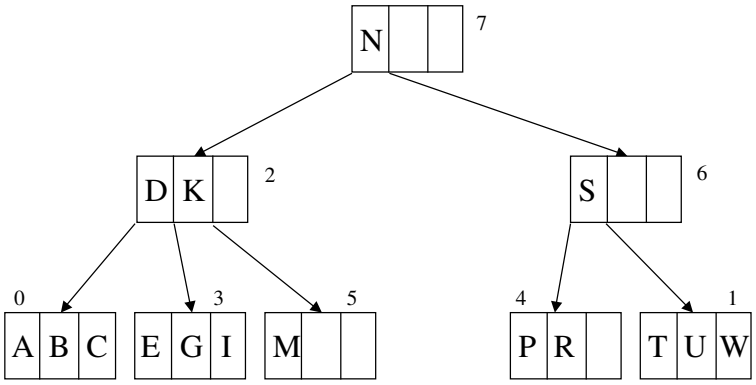
(g) Insert K

Insertion Example



(h) Insert K

Insertion Example



(h) Insert E

Operations

- Search
  - Recursively traverse the tree
  - What is the search performance?
    - What is the depth of the B-tree?
  - B-tree of order m with N keys and depth d
    - Best case: maximum number of descendents at each node

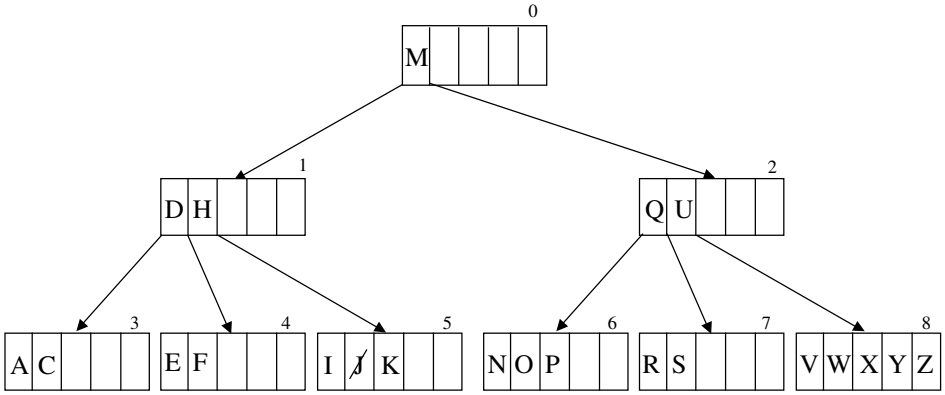
      $N = m^d$
    - Worst case: minimal number of descendents at each node
- Theorem $N = 2 \times \left\lceil \frac{m}{2} \right\rceil^{d-1}$

$$\log_m (N+1) \le d \le \log_{\lceil \frac{m}{2} \rceil} (\frac{N+1}{2}) + 1$$

Operations

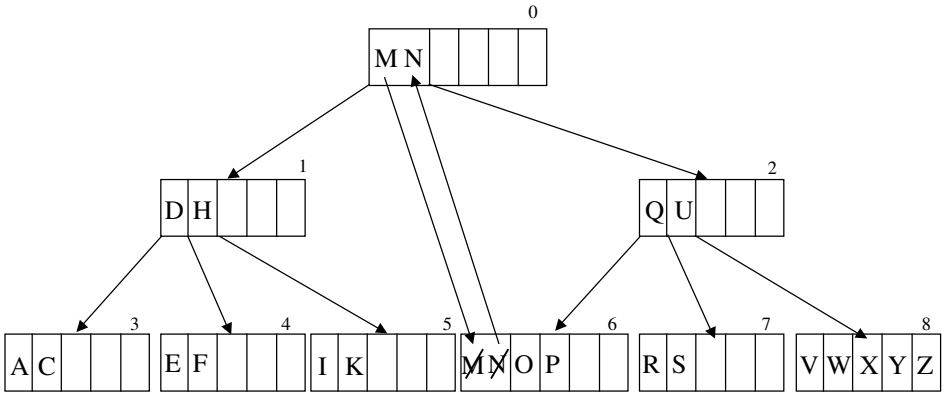
- Deletions
  - Search B-tree to find the key to be deleted
  - Swap the key with its immediate successor, if the key is not in a leaf page
    - Note only keys in a leaf may be deleted
  - When underflow: redistribution or concatenation
    - Redistribute keys among an adjacent sibling page, the parent page, and the underflow page if possible (need a rich sibling)
    - Otherwise, concatenate with an adjacent page, demoting a key from the parent page to the newly formed page.
  - If the demotion causes underflow, repeat redistribution-concatenation

Deletion Example – Simple



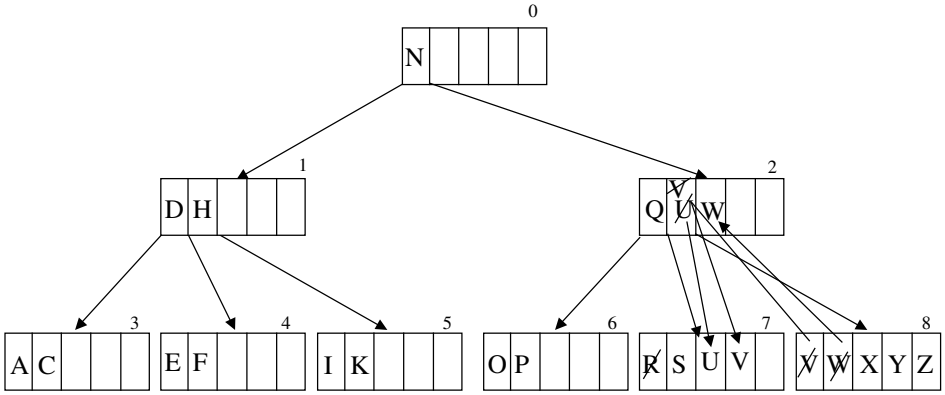
(a) Delete J

Deletion Example – Exchange



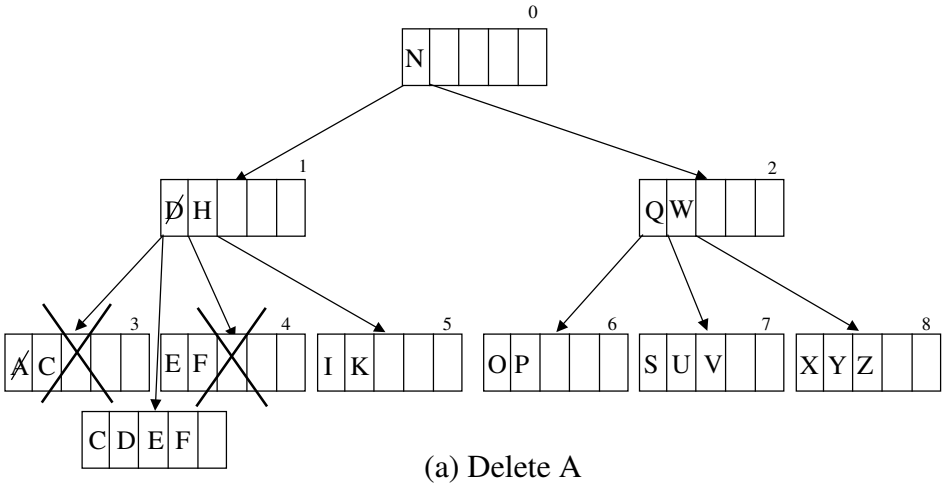
(a) Delete M

Deletion Example – Redistribution



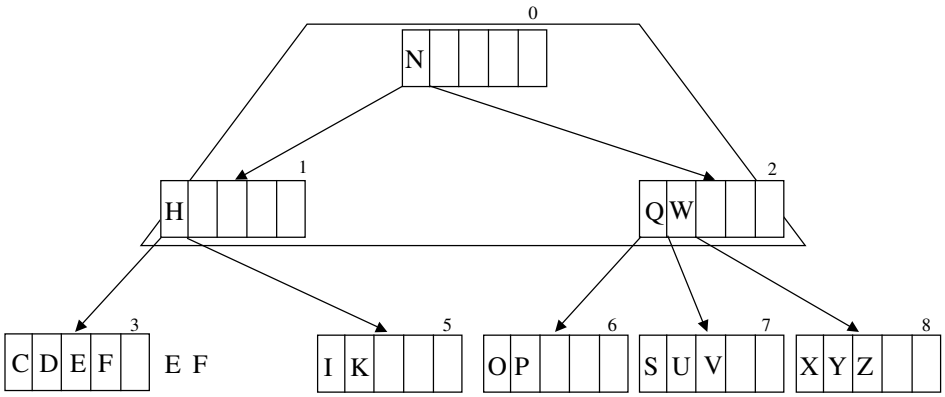
(a) Delete R

Deletion Example – Concatenation



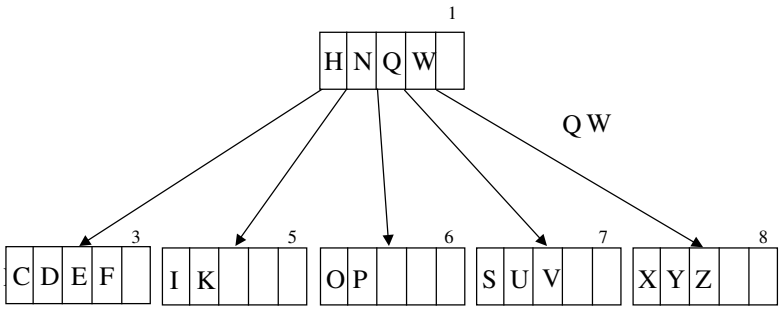
(a) Delete A

Deletion Example – Propagation



(a) Delete A (continued)

Deletion Example – Propagation



A C

(a) Delete A (final form)

Improvements

- Redistribution during insertion
  - A way of avoiding, or at least, postponing the creation of a new page by redistributing overflow keys into its sibling pages
  - Improve space utilization: $67\% \rightarrow 86\%$
- B*-trees
  - If redistribution takes placed during insertion, at time of overflow, at least one other sibling is full
  - Two-to-three split
    - Distribute all keys in two full pages into three sibling pages evenly
  - Each page contains at least $\left\lfloor \frac{2m-1}{3} \right\rfloor$ keys
  - Special handling of the root

Improvements

- Virtual B-trees
  - B-trees that uses RAM page buffers
  - Buffer strategies
    - Keep the root page
    - Retain the pages of higher levels
    - LRU (the Least Recently Uses page in the buffer is replaced by a new page)

Indexed Sequential Access

- Primary problem:
  - Efficient sequential access and indexed search (dual mode applications)
- Possible solutions:
  - Sorted files:
    - good for sequential accesses
    - unacceptable performance for random access
    - maintenance costs too high
  - B-trees:
    - good for indexed search
    - very slow for sequential accesses (tree traversal)
    - maintenance costs low
  - B+ trees: a file with a B-tree structure + a sequence set

Sequence Sets

- Arrange the file into blocks
  - Usually clusters or pages
- Records within blocks are sorted
- Blocks are logically ordered
  - Using a linked list
- If each block contains *b* records, then sequential access can be performed in *N*/*b* disk accesses

head = 2

1	D, E, F, G	4
2	A, B, C	1
3	J, K, L, M, N	-1
4	H, I	3

Maintenance of Sequence Sets

- Changes to blocks
  - Goal: keep blocks at least half full
    - Accommodates variable length records

file updates	problems	solutions
insertion	overflow	split w/o promotion
deletion	underflow	redistribution concatenation

- Choice of block size
  - The bigger the better
  - Restricted by size of RAM, buffer, access speed, track and sector sizes

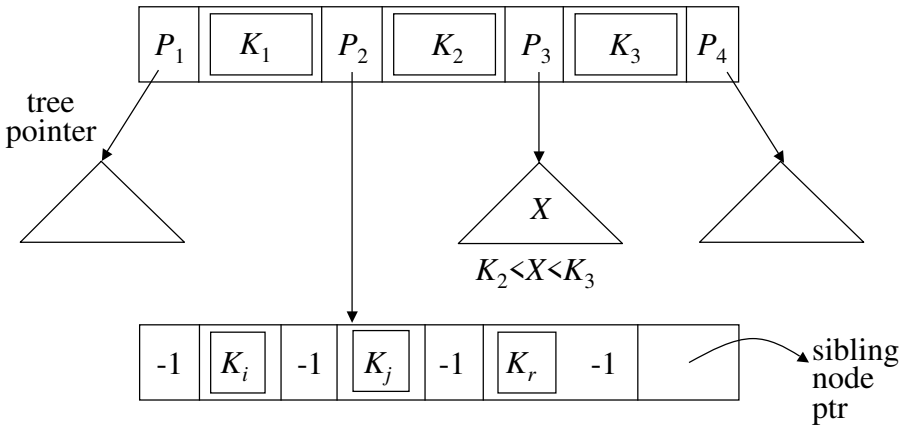
Indexed Access to Sequence Sets

- Keys of last record in each block
  - Similar to what you find on dictionary pages
  - Creates a one-level index
  - Has to be able to fit memory
    - binary search
    - index maintenance
- Separator: a shortest string that separates keys in two consecutive blocks
  - Increases the size of the index that can be maintained in memory

Maintenance of B+ Trees

- Updates are first made to the sequence set and then changes to the index set are made if necessary
  - If blocks are split, add a new separator
  - If blocks are concatenated, remove a separator
  - If records in the sequence set are redistributed, change the value of the separator

B+-Tree Structure



Differences Between B-tree and B^+ Tree

- Node information content
  - In B-trees all the pages contains the keys and information (or a pointer to it)
  - In the B^+ tree, the keys and information are contained in the sequence set
- Tree structure
  - B^+ tree is usually shallower than a B-tree (Why?)
- Access speed
  - Ordered sequential access is faster in B^+ trees

Hashing

- Primary problem:
  - Direct Access: how to accomplish index access in one seek
- Three major modes of file access
  - Sequential access
  - Index search: B-trees
    - Index function: (Key → address)
      - non-computable function defined by a table
      - one-to-one and on-to function
  - Direct access: Hashing
    - Address obtained directly from key
      - computable index function (Key → address)
      - record can be found in O(1) seeks on average (independent of file size)

Hash Function

- Input: a field of a record; usually its key K (student id, name, …)
- Compute hash (index) function $H(K)$

  $$H(K): K \rightarrow A$$

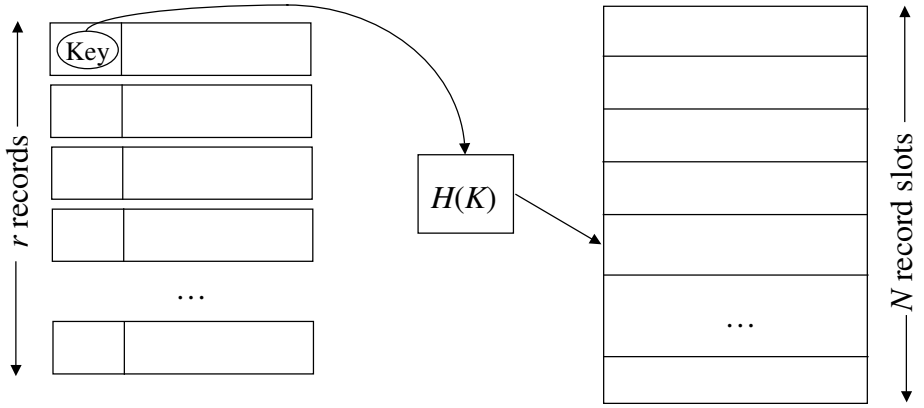
  to find the address of the record.

  $H(K)$ is the address of the record with key K

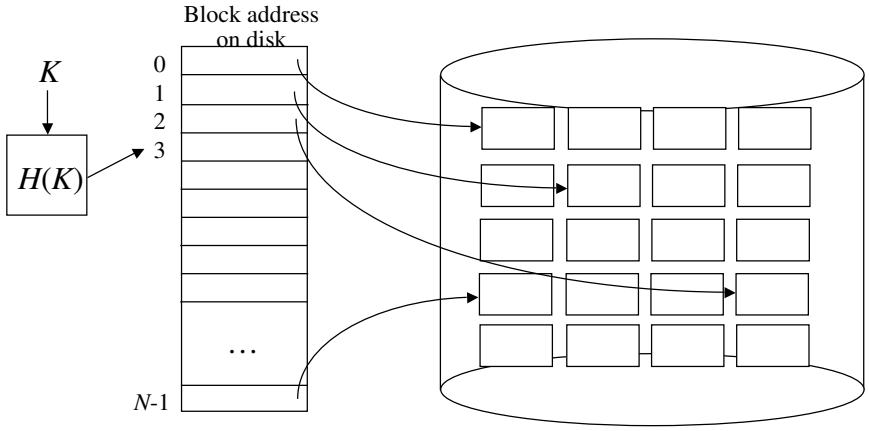
Types of Hashing

- What does $H(K)$ point to:
  - Internal hashing
    - Organize the entire file as a hash file
    - $H(K)$ gives the home address of the record
  - External hashing
    - The file is on disk and the hashing is done to the file header
    - $H(K)$ gives the bucket address from which the block address can be found using file header information.
- Organization of the file:
  - Static hashing
    - File size is fixed
  - Dynamic & extendible hashing
    - File size can grow

Internal Hashing



External Hashing



This is equivalent to scatter tables + buckets
where the bucket number/block address mapping is maintained
in file header

Dynamic Files & Hashing

- Problem with hashing is that the address space (N) is fixed.
  - Overflow results
- How to handle dynamic files better?
  - Dynamic hashing
  - Extendible hashing
  - Linear hashing

Extendible Hashing

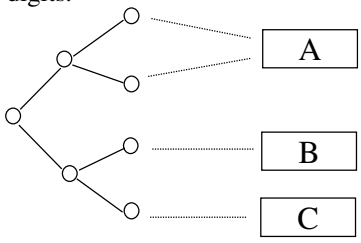
- Binary search tree ⟹ B-tree
  - dynamically self-adjusted balanced page tree
- Hashing: computable index function
  - problem: linear search for overflowed records
  - Solution: bucket size

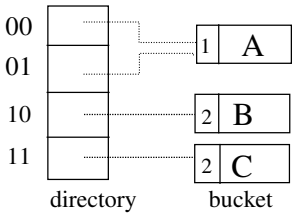
What if we have an unlimited bucket size ?

What if we use a dynamic, self-adjusted structure with unlimited bucket size?

Extendible Hashing

- Basic Idea:
  - Build an index based on the binary number representation of the hash value
  - Use minimal set of binary digits; use more as more is needed
- Example:
  - Assume each hashed key is a sequence of three digits, and in the beginning, we need only three buckets.
  - Usually, 8 buckets are needed for the set of all keys with three digits.

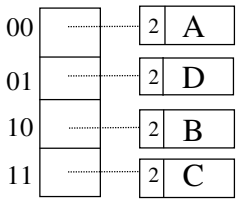




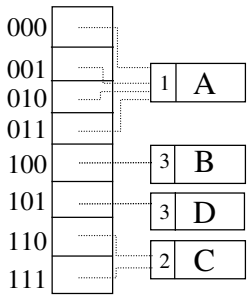
67

Example (cont'd)

- When A overflows, create a new bucket D to insert new records, and expanding the address of A into two addresses



- If, instead, B overflows, then create a new bucket D and expand the address of B into three digits



68

Deletions

- Find buddy buckets to collapse
- Two buckets are buddies if
  - They are at the same depth
  - Their initial bit strings are the same
- This is similar to finding siblings in B-trees

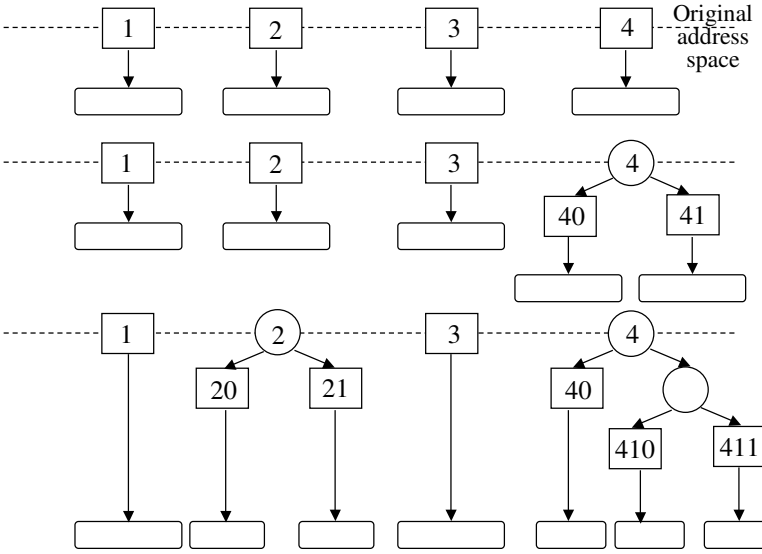
Access performance

- One seek if the directory can be kept in RAM
- Two seeks, otherwise
- Space utilization:
  - Average utilization is 69%
    - compared with 67% for simple B-trees which can be optimized to have utilization of 85%
  - Variations in space utilization are periodic
    - Go over 90% and then drop to 50%

Dynamic Hashing

- Very similar to extendible hashing
- Starts with a fixed address size (similar to static hashing) and then grows as needed
- Slow and incremental growth of the directory (extendible hashing doubles it when it grows)
- The node structure takes up more space (because of the maintenance of tree structure)
- Usually two hash functions
  - First try to see if the bucket is in the original address space
  - If not, try the second hash function to guide the search through the trie

Dynamic Hashing Example



Linear Hashing

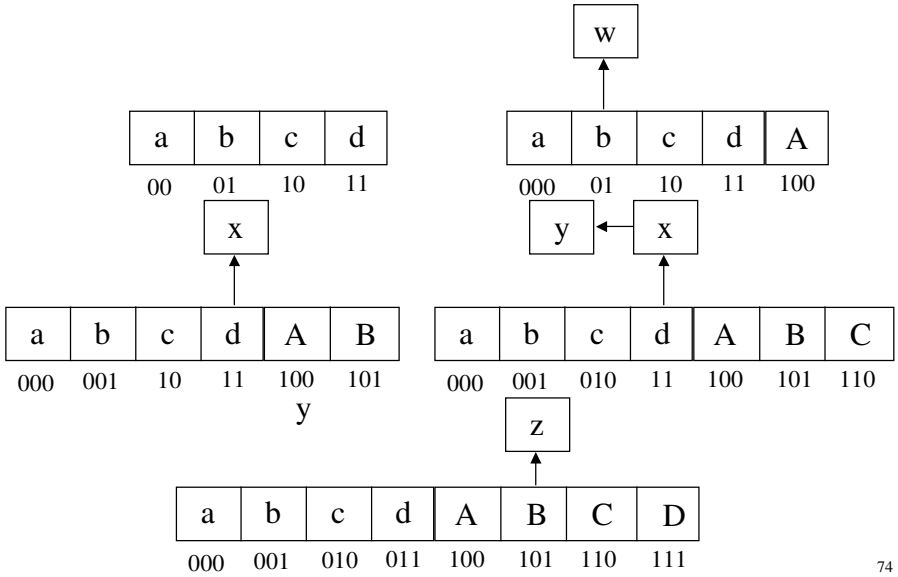
- Allows a hash file to expand and shrink dynamically without needing a directory
- Starts with b buckets and uses one hash function:

  $$H_d(K) = K \bmod b$$

  to find the bucket (d is the number of bits needed to identify b buckets).
- When a bucket, say b_i, fills, splits the next bucket, say b_j, in sequence to be split (not b_i) and moves half the records of b_j to the new bucket B_j using the hash function $H_{d+1}(K)$.
- Search (p is the pointer to the address of the next bucket to be split and extended):

  if $H_d(K) >= p$ address:= $H_d(K)$
  else address:= $H_{d+1}(K)$

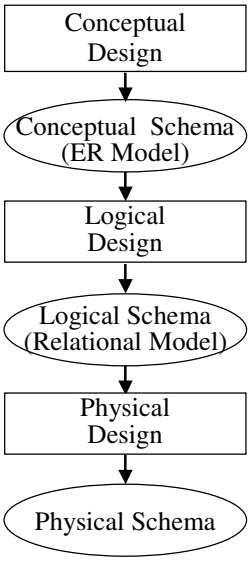
Linear Hashing Example



Hashing vs B+ Trees

- Times for exact search/update
  - Hashing can have O(1) or O(*N*) performance
  - B+-trees have O(log*N*) performance
- Space
  - Hash tables depend on load factor (controllable in linear hashing)
  - B+-trees use 65%-85% of space
- Range querying
  - Supported by B+-trees, but not by hashing

Design Process - Physical Design



Physical Design

- Choice of indexes
- Clustering of data
- May have to revisit and refine the conceptual and external schemas to meet performance goals.
- Most important is to understand the workload
  - The most important queries and their frequency.
  - The most important updates and their frequency.
  - The desired performance for these queries and updates.

Workload Modeling

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Physical Design Decisions

- What indexes should be created?
  - Relations to index
  - Field(s) to be used as the search key
  - Perhaps multiple indexes?
  - For each index, what kind of an index should it be?
    - Clustered? Hash/tree? Dynamic/static? Dense/sparse?
- Should changes be made to the conceptual schema?
  - Alternative normalized schemas
  - Denormalization
  - Partitioning (vertical & horizontal)
  - New view definitions
- Should the frequently executed queries be rewritten to run faster?

Choice of Indexes

- Consider the most important queries one-by-one
  - Consider the best plan using the current indexes
  - See if a better plan is possible with an additional index
  - If so, create it.
- Consider the impact on updates in the workload
  - Indexes can make queries go faster,
  - Updates are slower
  - Indexes require disk space, too.

Index Selection Guidelines

- Don't index unless it contributes to performance.
- Attributes mentioned in a WHERE clause are candidates for index search keys.
  - Exact match condition suggests hash index.
  - Range query suggests tree index.
    - Clustering is especially useful for range queries, although it can help on equality queries as well in the presence of duplicates.
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - If range selections are involved, order of attributes should be carefully chosen to match the range ordering.
  - Such indexes can sometimes enable index-only strategies for important queries.
    - For index-only strategies, clustering is not important!

Index Selection Guidelines (cont'd.)

- Try to choose indexes that benefit as many queries as possible. Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

Example

- Consider the query

```
SELECT A,B
FROM R
WHERE A>5 AND C='Waterloo'
```

- Assume no index
  - Sequentially scan relation R to check each tuple for A>5 and C='Waterloo'
  - Qualifying tuples are projected over attributes A and B.

Example (cont'd)

- Consider the query

```
SELECT A,B
FROM R
WHERE A>5 AND C='Waterloo'
```

- Assume index on A, B, C
  - Search index on A to find records where A>5
  - Retrieve these records
  - If C='Waterloo' then project over A, B
  - Should you do the index search on A or C?
    - ➠ More selective one

Example (cont'd)

- Consider the update query

```
UPDATE R
SET A TO 3
WHERE C='Waterloo'
```

- Assume you have index on A, C
  - Search the index on C to find C='Waterloo'
  - For each such record, update A in the record
  - For each such record, update A index