Indexing

- Sorted File
- Basic Concepts
- Ordered Indices
- B+-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Multiple-Key Access

Access Path

- Refers to the algorithm + 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
- This choice can have a major effect on the execution time of the statement

Example Relational Schema

- Transcript (StudID, CrsCode, Semester, Grade)
- Student(StudID, Name, DeptID, Address, Status)
- Course(CrsCode, DeptID, CrsName, Descr)

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_2F 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.CrsCode,
T.Grade
FROM  Transcript T
WHERE  T.StudId = 123456
```

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

Transcript Stored as a Heap File

666666	MGT123	F1994	4.0	page 0
123456	CS305	S1996	4.0	
987654	CS305	F1995	2.0	

717171	CS315	S1997	4.0	page 1
666666	EE101	S1998	3.0	
765432	MAT123	S1996	2.0	
515151	EE101	F1995	3.0	

234567	CS305	S1999	4.0	page 2
878787	MGT123	S1996	3.0	

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	

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 (because shifting is necessary to make space)
- **Partial Solution 1**: Leave empty space in each page: *fill factor*
- **Partial Solution 2**: Use *overflow pages (chains).*
  - Disadvantages:
    - Successive pages no longer stored contiguously
    - Overflow chain not sorted, hence cost no longer $log_2 F$

Overflow

Pointer to overflow chain

3			
111111	MGT123	F1994	4.0
111111	CS305	S1996	4.0
111111	ECO101	F2000	3.0
122222	REL211	F2000	2.0

page 0

These pages are Not overflown

-			
123456	CS315	S1997	4.0
123456	EE101	S1998	3.0
232323	MAT123	S1996	2.0
234567	EE101	F1995	3.0

page 1

-			
234567	CS305	S1999	4.0
313131	MGT123	S1996	3.0

page 2

Pointer to next block in chain

7			
111654	CS305	F1995	2.0
111233	PSY 220	S2001	3.0

page 3

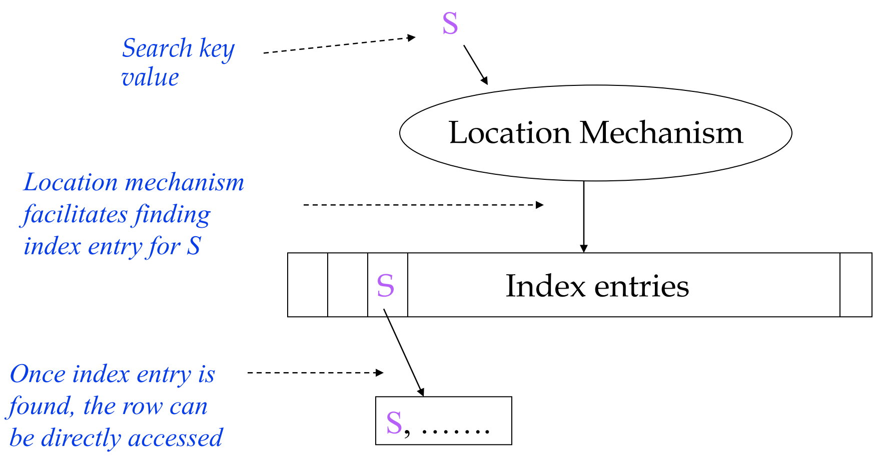
Index

- 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
- Don't confuse candidate key with search key:
  - *Candidate key*:
    - set of attributes;
    - guarantees uniqueness
  - *Search key*:
    - sequence of attributes;
    - does not guarantee uniqueness –just used for search

Index Structure

- Contains:
  - *Index entries*
    - Can contain the data tuple itself (index and table are *integrated* in this case); or
    - Search key value and a pointer to a row having that value; table stored separately in this case – *unintegrated* index
  - *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)

Index Structure

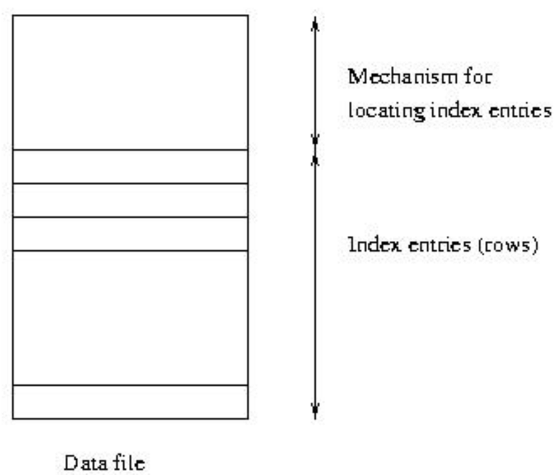


Storage Structure

- 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

Integrated Storage Structure

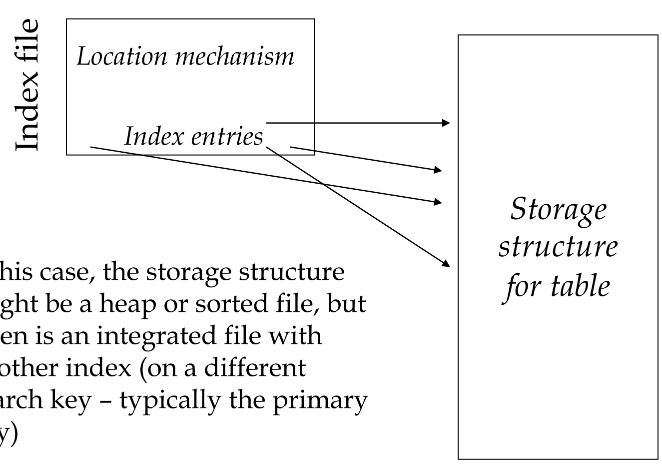
Contains table and (main) index



CSC443 – w20

13

Index File With Separate Storage Structure



In this case, the storage structure might be a heap or sorted file, but often is an integrated file with another index (on a different search key – typically the primary key)

CSC443 – w20 14

14

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

Indices: The Down Side

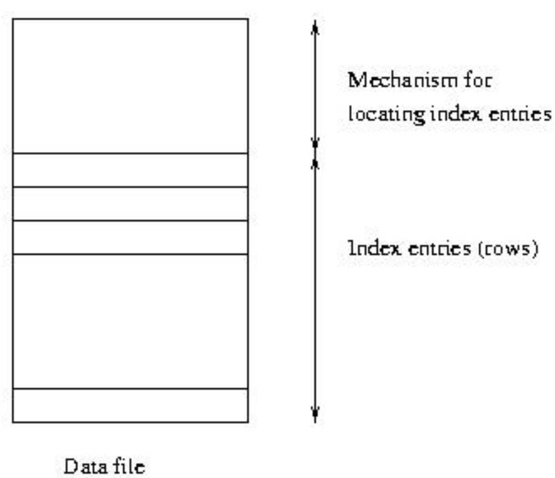
- Additional I/O to access index pages (except if index is small enough to fit in main memory)
- Index must be updated when table is modified.
- SQL-92 does not provide for creation or deletion of indices
  - Index on primary key generally created automatically
  - Vendor specific statements:
    - CREATE INDEX ind ON Transcript (*CrsCode*)
    - DROP INDEX ind

Clustered Index

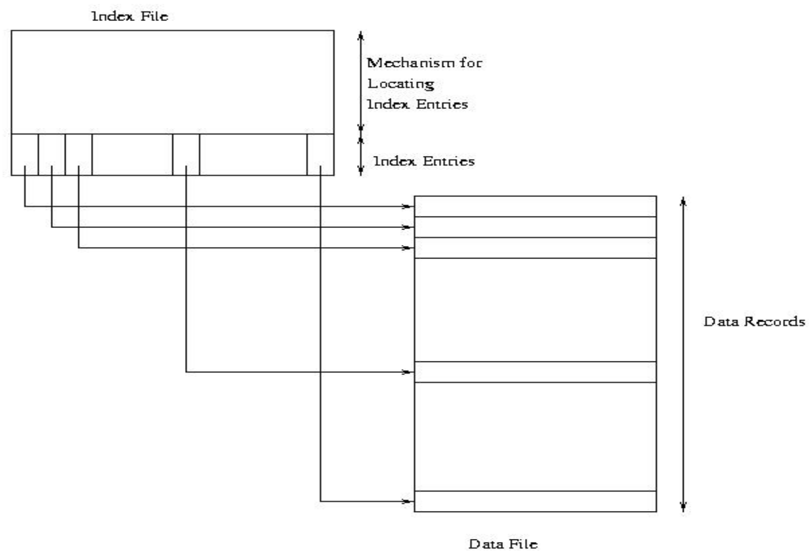
- ❑ *Clustered index*: index entries and rows are ordered in the same way
  - An integrated storage structure is always clustered (since rows and index entries are the same)
  - The particular index structure (eg, hash, tree) dictates how the rows are organized in the storage structure
    - There can be at most one clustered index on a table
  - CREATE TABLE generally creates an integrated, clustered (main) index on primary key

Clustered Main Index

Storage structure contains table and (main) index; rows are contained in index entries



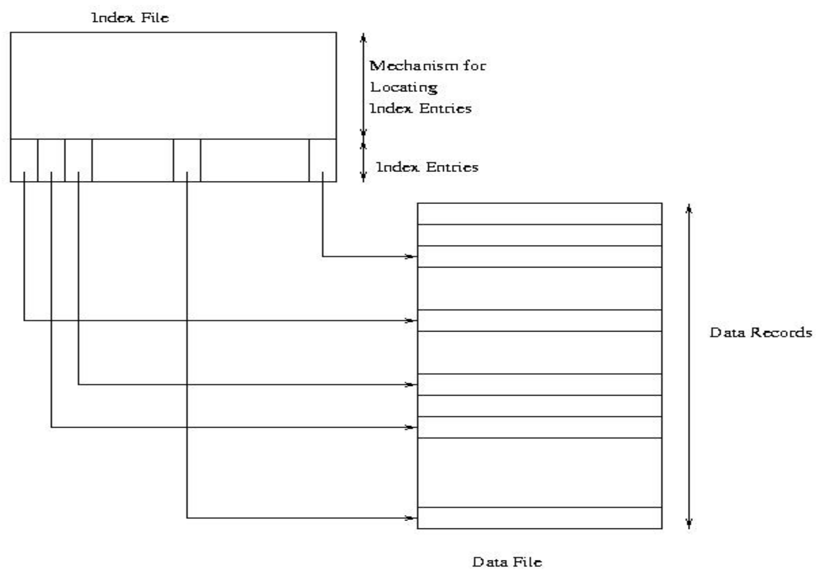
Clustered Index



Unclustered Index

- *Unclustered* (secondary) index: index entries and rows are not ordered in the same way
- A secondary index might be clustered or unclustered with respect to the storage structure it references
  - It is generally unclustered (since the organization of rows in the storage structure depends on main index)
  - There can be many secondary indices on a table
  - Index created by CREATE INDEX is generally an unclustered, secondary index

Unclustered Secondary Index



Clustered Index

- Good for range searches when a range of search key values is requested
  - Use location mechanism to locate index entry at start of range
    - This locates first row.
  - Subsequent rows are stored in successive locations if index is clustered (not so if unclustered)
  - Minimizes page transfers and maximizes likelihood of cache hits

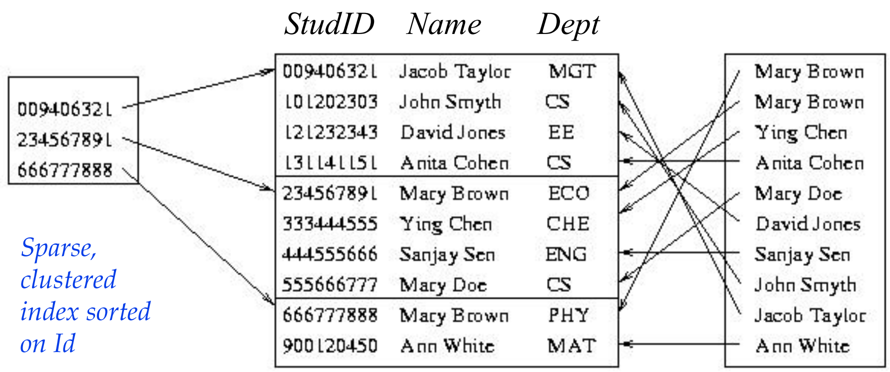
Example – Cost of Range Search

- Data file has 10,000 pages, 100 rows in search range
- Page transfers for table rows (assume 20 rows/page):
  - Heap: 10,000 (entire file must be scanned)
  - File sorted on search key: $\log_2 10000 + (5 \text{ or } 6) \approx 19$
  - Unclustered index: ≤ 100
  - Clustered index: 5 or 6
- Page transfers for index entries (assume 200 entries/page)
  - Heap and sorted: 0
  - Unclustered secondary index: 1 or 2 (all index entries for the rows in the range must be read)
  - Clustered secondary index: 1 (only first entry must be read)

Sparse vs. Dense Index

- *Dense index*: has index entry for each data record
  - Unclustered index *must* be dense
  - Clustered index need not be dense
- *Sparse index*: has index entry for each page of data file

Sparse Vs. Dense Index cont'd



Sparse, clustered index sorted on Id

Data file sorted on StudID

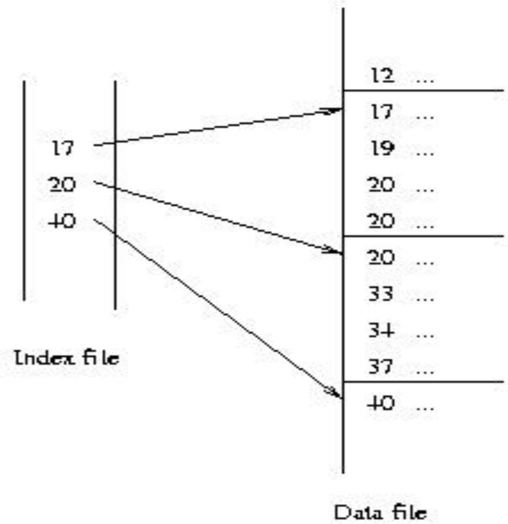
Dense, unclustered index sorted on Name

Sparse Index

- 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.

Sparse Index cont'd

Search key should be candidate key of data file (else additional measures required)



CSC443 – w20 27

27

Multiple Attribute Search Key

- CREATE INDEX Inx ON Tbl (Attr1, Attr2)
- Search key is a sequence of attributes; index entries are lexically ordered
- Supports finer granularity equality search:
  - Find row with value (A1, A2)
- Supports range search (tree index only):
  - Find rows with values between (A1, A2) and (A3, A4)
- Supports partial key searches (tree index only):
  - Find rows with values of Att1 between A1 and A3
  - But not Find rows with values of Att2 between A2 and A4

CSC443 – w20 28

28

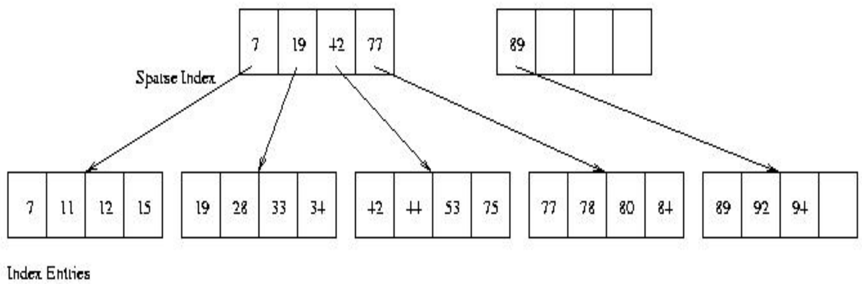
Locating an Index Entry

- Use binary search (index entries sorted)
  - If Q pages of index entries, then log_2Q page transfers (which is a big improvement over binary search of the data pages of an F page data file since $F >> Q$)
- Use multilevel index: Sparse index on sorted list of index entries

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.

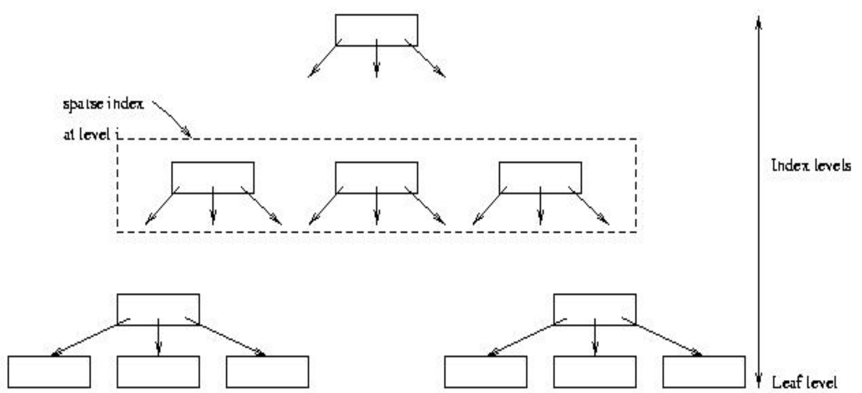
Two-Level Index



- *Separator level* is a sparse index over pages of index entries
- *Leaf level* contains index entries
- Cost of searching the separator level << cost of searching index level since separator level is sparse

- Cost or retrieving row once index entry is found is 0 (if integrated) or 1 (if not)

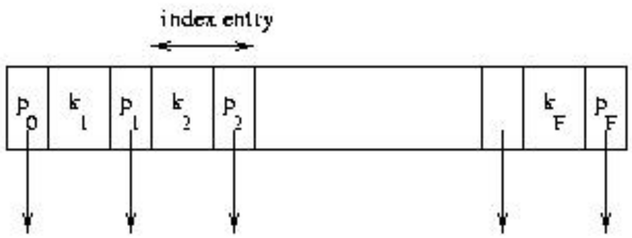
Multilevel Index cont'd



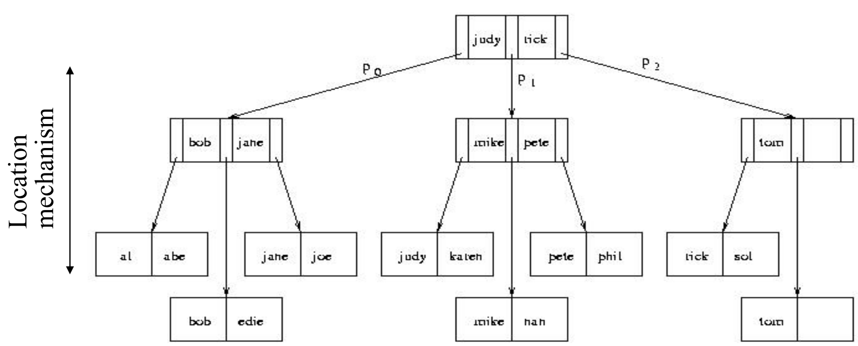
- Search cost = number of levels in tree
- If Φ is the fanout of a separator page, cost is $log_{\Phi} Q + 1$
- Example: if $\Phi = 100$ and $Q = 10{,}000$, cost = 3
  (reduced to 2 if root is kept in main memory)

Index Sequential Access Method (ISAM)

- Generally, an integrated storage structure
  - Clustered, index entries contain rows
- *Separator entry* = (k_i , p_i);
  - k_i is a search key value;
  - p_i is a pointer to a lower level page
- k_i separates set of search key values in the two subtrees pointed at by p_{i-1} and p_i.



Index Sequential Access Method

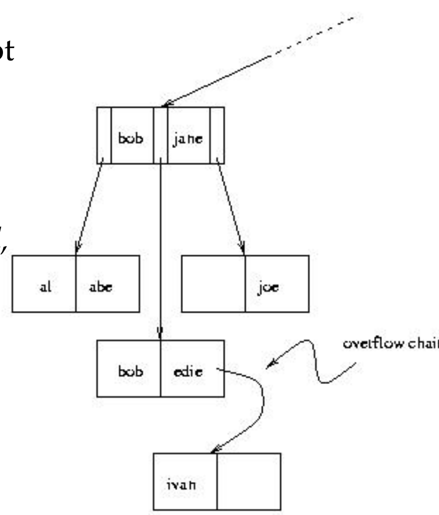


Index Sequential Access Method

- The index is static:
  - Once the separator levels have been constructed, they never change
  - Number and position of leaf pages in file stays fixed
- Good for equality and range searches
  - Leaf pages stored sequentially in file when storage structure is created to support range searches
    - if, in addition, pages are positioned on disk to support a scan, a range search can be very fast (static nature of index makes this possible)
- Supports multiple attribute search keys and partial key searches

Overflow Chains

- Contents of leaf pages change
- Row deletion yields empty slot in leaf page
- Row insertion can result in overflow leaf page and ultimately overflow chain
  - *Chains can be long, unsorted, scattered on disk*
  - *Thus ISAM can be inefficient if table is dynamic*

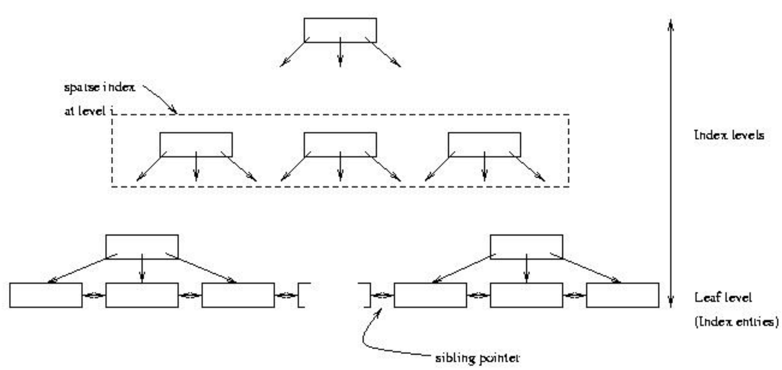


B^+ Tree

- Supports equality and range searches, multiple attribute keys and partial key searches
- Either a secondary index (in a separate file) or the basis for an integrated storage structure

Responds to dynamic changes in the table

B^+ Tree Structure



- Leaf level is a (sorted) linked list of index entries
- Sibling pointers support range searches in spite of allocation and deallocation of leaf pages (but leaf pages might not be physically contiguous on disk)

B^+-Tree Node Structure

- Typical node

P_1	K_1	P_2	. . .	P_{n-1}	K_{n-1}	P_n

  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

Leaf Nodes in B^+-Trees

Properties of a leaf node

- For $i = 1, 2, \ldots, n-1$, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i. Only need bucket structure if search-key does not form a primary key.
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than L_j's search-key values
- P_n points to next leaf node in search-key order

Non-Leaf Nodes in B⁺-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_{m-1}

P_1	K_1	P_2	. . .	P_{n-1}	K_{n-1}	P_n

Observations about B⁺-trees

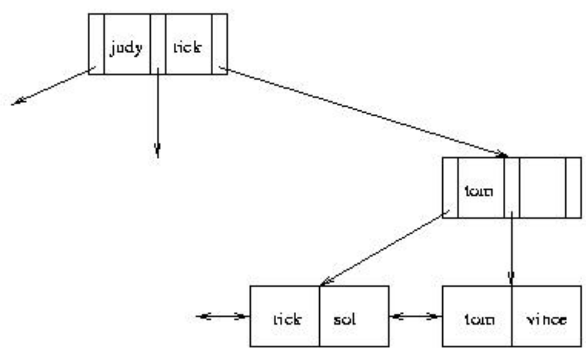
- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The non-leaf levels of the B⁺-tree form a hierarchy of sparse indices.
- The B⁺-tree contains a relatively small number of levels (logarithmic in the size of the main file), thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

Insertion and Deletion in B⁺ Tree

- Structure of tree changes to handle row insertion and deletion – *no* overflow chains
- Tree remains *balanced*: all paths from root to index entries have same length
- Algorithm guarantees that the number of separator entries in an index page is between $\Phi/2$ and Φ
  - Hence the maximum search cost is $log_{\Phi/2}Q + 1$ (with ISAM search cost depends on length of overflow chain)

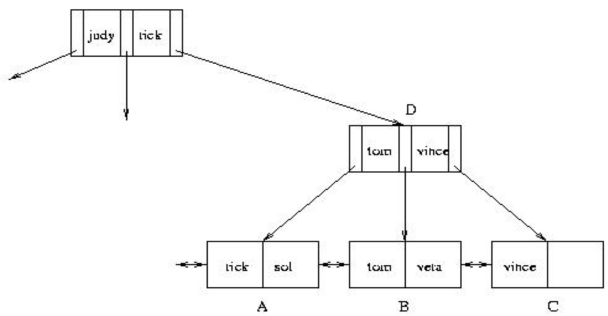
Handling Insertions - Example

- Insert "vince"



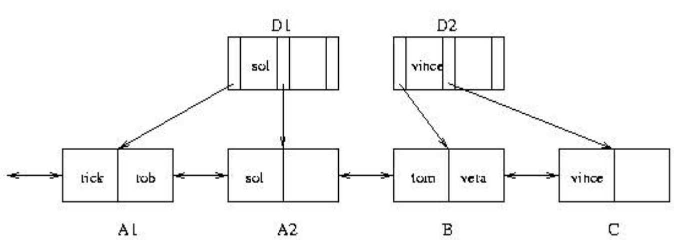
Handling Insertions cont'd

– Insert "vera": Since there is no room in leaf page:
1. Create new leaf page, C
2. Split index entries between B and C (but maintain sorted order)
3. Add separator entry at parent level



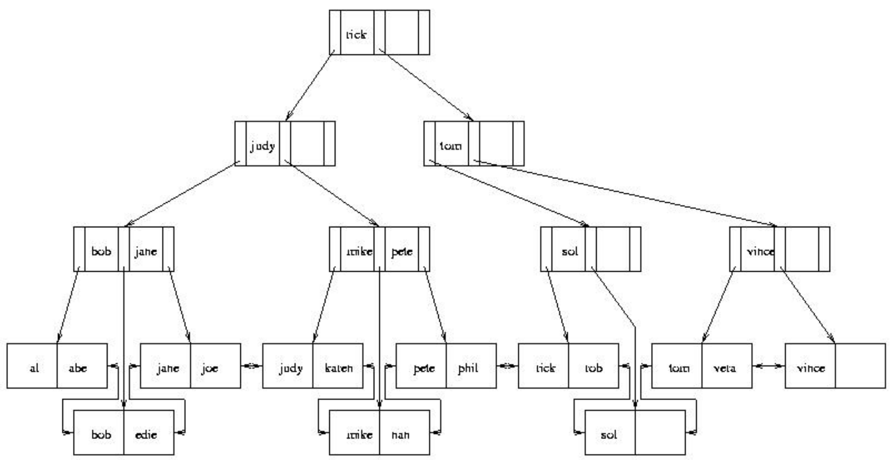
Handling Insertions cont'd

– Insert "rob". Since there is no room in leaf page A:
1. Split A into A1 and A2 and divide index entries between the two (but maintain sorted order)
2. Split D into D1 and D2 to make room for additional pointer
3. Three separators are needed: "sol", "tom" and "vince"



Handling Insertions cont'd

– When splitting a separator page, push a separator up
– Repeat process at next level
– Height of tree increases by one



Handling Deletions

- ❑ Deletion can cause page to have fewer than $\Phi/2$ entries
  - Entries can be redistributed over adjacent pages to maintain minimum occupancy requirement
  - Ultimately, adjacent pages must be merged, and if merge propagates up the tree, height might be reduced
  - See book
- ❑ In practice, tables generally grow, and merge algorithm is often not implemented
  - *Reconstruct tree to compact it*

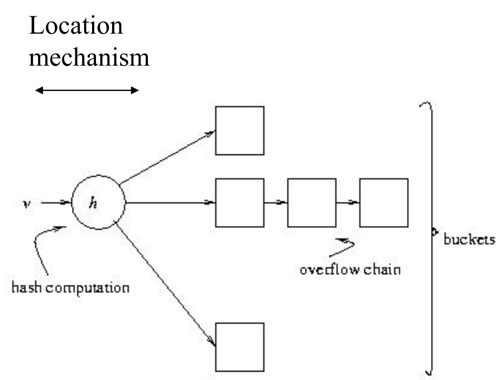
Hash Index

- Index entries partitioned into *buckets* in accordance with a *hash function, h(v)*, where *v* ranges over search key values
  - Each bucket is identified by an address, *a*
  - Bucket at address *a* contains all index entries with search key *v* such that *h(v)* = *a*
- Each bucket is stored in a page (with possible overflow chain)
- If index entries contain rows, set of buckets forms an integrated storage structure; else set of buckets forms an (unclustered) secondary index

Equality Search with Hash Index

Given *v*:
1. Compute *h(v)*
2. Fetch bucket at *h(v)*
3. Search bucket

Cost = number of pages in bucket (cheaper than B^+ tree, if no overflow chains)



Choosing a Hash Function

- Goal of *h*: map search key values randomly
  - Occupancy of each bucket roughly same for an average instance of indexed table
- Example: $h(v) = (c_1 x\ v + c_2) \bmod M$
  - *M* must be large enough to minimize the occurrence of overflow chains
  - *M* must not be so large that bucket occupancy is small and too much space is wasted

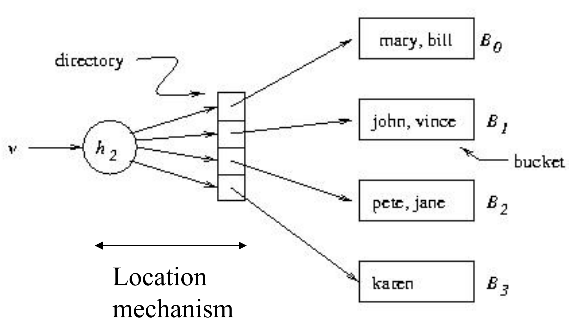
Hash Indices – Problems

- Does not support range search
  - Since adjacent elements in range might hash to different buckets, there is no efficient way to scan buckets to locate all search key values *v* between v_1 and v_2
- Although it supports multi-attribute keys, it does not support partial key search
  - Entire value of *v* must be provided to *h*
- Dynamically growing files produce overflow chains, which negate the efficiency of the algorithm

Extendable Hashing

- Eliminates overflow chains by splitting a bucket when it overflows
- Range of hash function has to be extended to accommodate additional buckets
- **Example:** family of hash functions based on h:
  - $h_k(v) = h(v) \mod 2^k$ (use the last k bits of $h(v)$)
  - At any time use only a prefix of the hash function to index into a table of bucket addresses
  - At any given time a unique hash, h_k, is used depending on the number of times buckets have been split

Extendable Hashing – Example



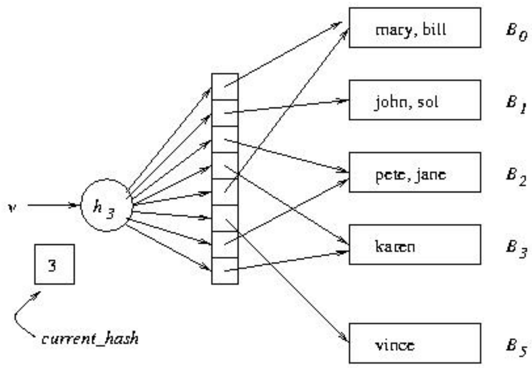
v	$h(v)$
pete	11010
mary	00000
jane	11110
bill	00000
john	01001
vince	10101
karen	10111

- Extendable hashing uses a directory (level of indirection) to accommodate family of hash functions

Suppose next action is to insert sol, where $h(sol) = 10001$.

Problem: This causes overflow in B_1

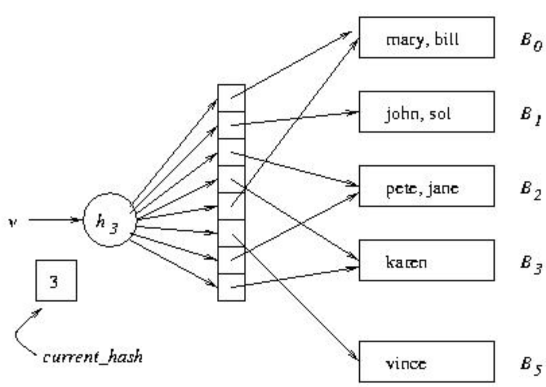
Extendable Hashing Example cont'd



Solution:
1. Switch to h_3
2. Concatenate copy of old directory to new directory
3. Split overflowed bucket, B, into B and B', dividing entries in B between the two using h_3
4. Pointer to B in directory copy replaced by pointer to B'

Note: Except for B' , pointers in directory copy refer to original buckets.
current_hash identifies current hash function.

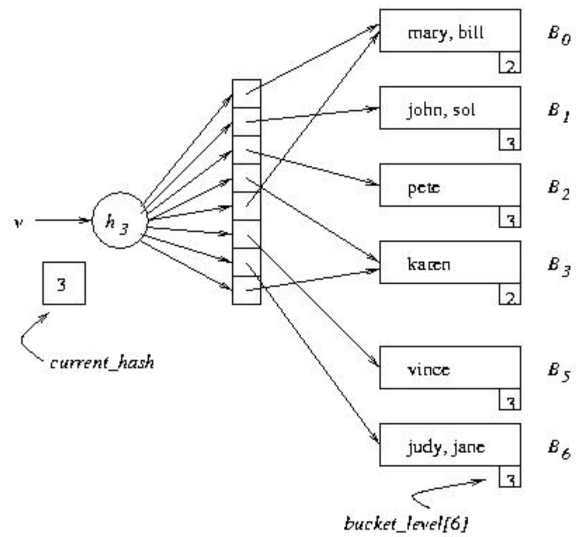
Extendable Hashing Example cont'd



Next action: Insert judy, where $h(judy) = 00110$
B_2 overflows, but directory need not be extended

Problem: When B_i overflows, we need a mechanism for deciding whether the directory has to be doubled
Solution: *bucket_level[i]* records the number of times B_i has been split. If *current_hash* > *bucket_level[i]*, do not enlarge directory

Extendable Hashing Example cont'd



Extendable Hashing

- Deficiencies:
  - Extra space for directory
  - Cost of added level of indirection:
    - If directory cannot be accommodated in main memory, an additional page transfer is necessary.

Choosing An Index

- An index should support a query of the application that has a significant impact on performance
  - Choice based on frequency of invocation, execution time, acquired locks, table size

CSC443 – w20 59

Choosing An Index: Examples

Example 1:

```
SELECT  E.Id
FROM    Employee E
WHERE   E.Salary < :upper  AND
        E.Salary > :lower
```

- This is a range search on *Salary*.
- Since the primary key is Id, it is likely that there is a clustered, main index on that attribute that is of no use for this query.
- Choose a secondary, B^+ tree index with search key *Salary*

CSC443 – w20 60

Choosing An Index: Examples

Example 2:

```
SELECT  T.StudId
FROM    Transcript T
WHERE   T.Grade = :grade
```

- This is an equality search on *Grade*.
- Since the primary key is (*StudId*, *Semester*, *CrsCode*) it is likely that there is a main, clustered index on these attributes that is of no use for this query.
- Choose a secondary, B+ tree or hash index with search key *Grade*

Choosing An Index: Examples

Example 3 cont'd:

```
SELECT  T.CrsCode, T.Grade
FROM    Transcript T
WHERE   T.StudId = :id  AND
        T.Semester = 'w2006'
```

- Equality search on *StudId* and *Semester*.
- If the primary key is (*StudId*, *Semester*, *CrsCode*) it is likely that there is a main, clustered index on this *sequence* of attributes.
- If the main index is a B^+ tree it can be used for this search.
- If the main index is a hash it cannot be used for this search. Choose B^+ tree or hash with search key *StudId*
  - (since *Semester* is not as selective as *StudId*) or (*StudId*, *Semester*)

Choosing An Index: Examples

Example 3:

```
SELECT  T.CrsCode, T.Grade
FROM    Transcript T
WHERE   T.StudId = :id  AND
        T.Semester = 'w2006'
```

- Suppose Transcript has primary key (*CrsCode*, *StudId*, *Semester*).
  Then, the main index is of no use (independent of whether it is a hash or a B^+ tree).