Physical Data Organization and Indexing

CSE 532, Theory of Database Systems

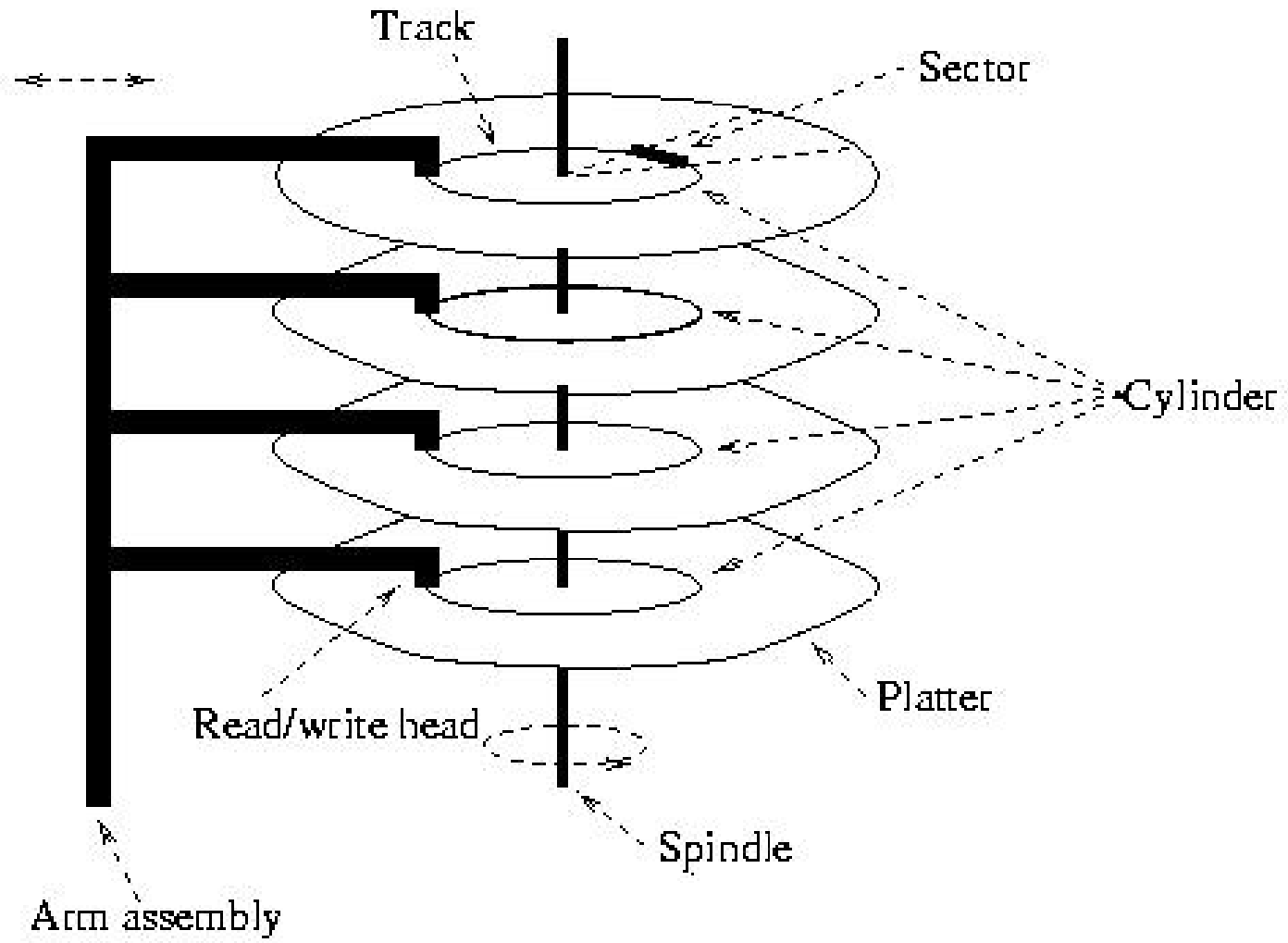
Stony Brook University

http://www.cs.stonybrook.edu/~cse532

Disks

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

Physical Disk Structure



Disks

- The time to access a sector, S, can be divided into three components:

1. Seek time = the time to position the arm assembly over the cylinder containing S.
2. Rotational latency = the additional time it takes, after the arm assembly is over the cylinder, for the platters to rotate to the angular position at which S is under the read/write head.
3. Transfer time = the time it takes for the platter to rotate through the angle subtended by S.

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 (main memory) buffer
  - (Eventually) copy buffer page to page on disk

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
- Our goal – minimize average latency, reduce number of page transfers

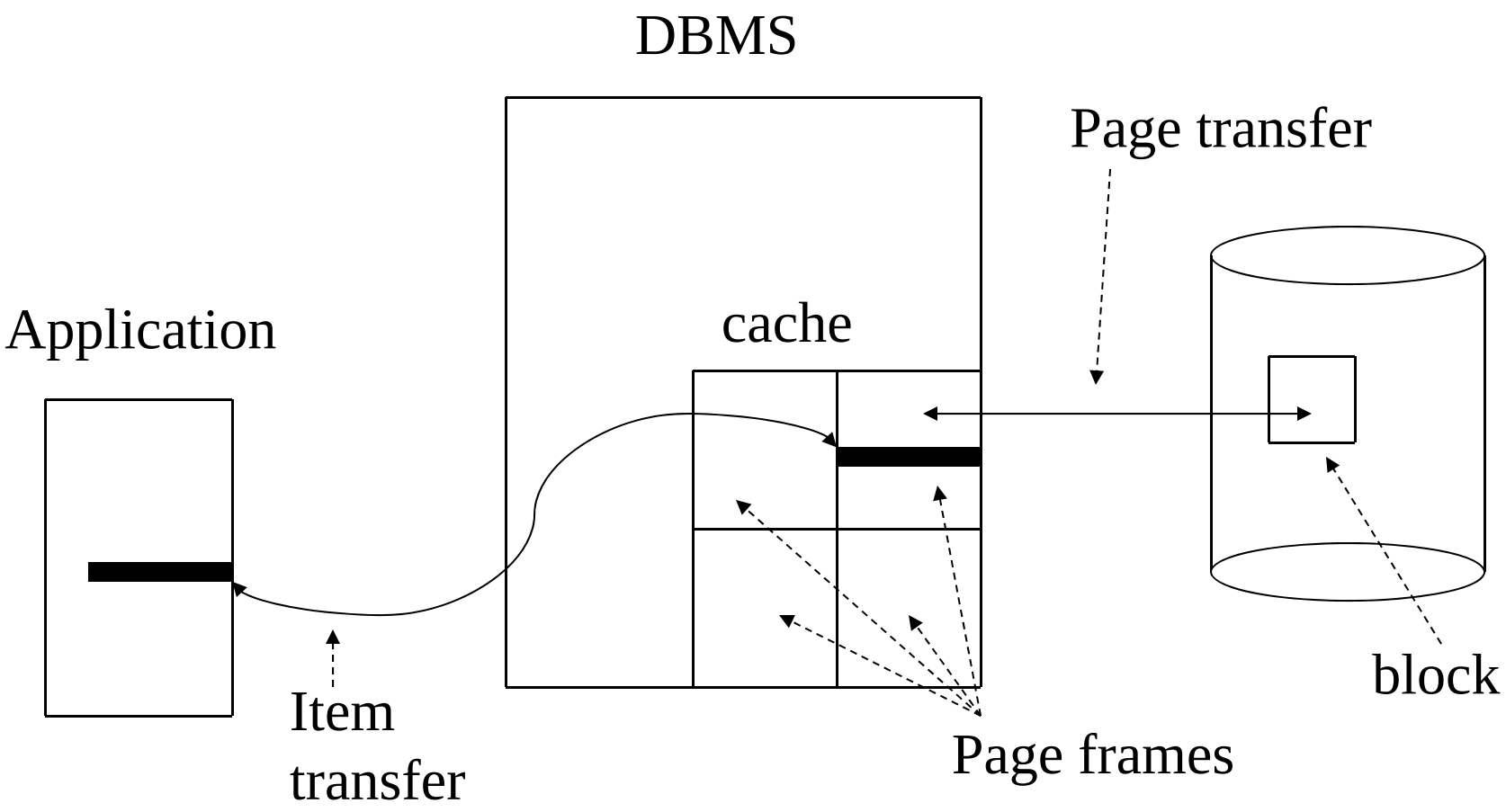
Reducing Latency

- 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

Reducing Number of Page Transfers

- Keep cache of recently accessed pages in main memory
  - *Rationale*: 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)

Accessing Data Through Cache

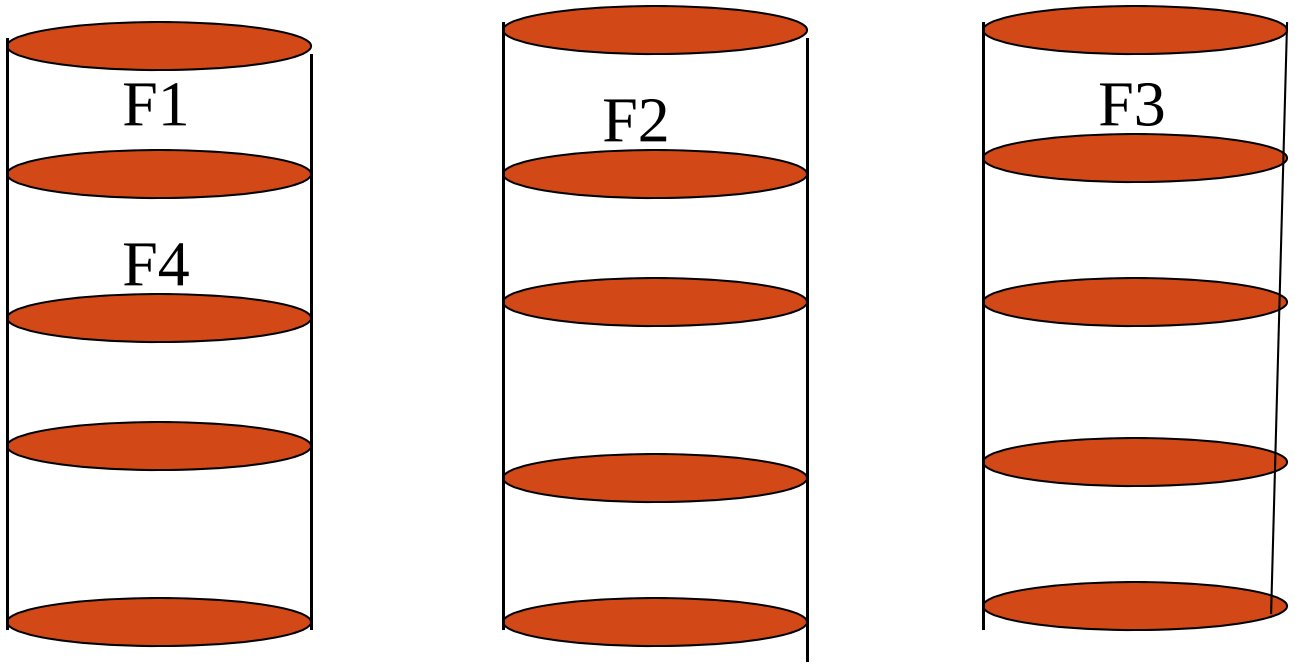


RAID Systems

- RAID (Redundant Array of Independent Disks) is an array of disks configured to behave like a single disk with
  - Higher throughput
    - Multiple requests to different disks can be handled independently
    - If a single request accesses data that is stored separately on different disks, that data can be transferred in parallel
  - Increased reliability
    - Data is stored redundantly
    - If one disk should fail, the system can still operate

Striping

- Data that is to be stored on multiple disks is said to be *striped*
  - Data is divided into *chunks*
    - Chunks might be bytes, disk blocks etc.
  - If a file is to be stored on three disks
    - First chunk is stored on first disk
    - Second chunk is stored on second disk
    - Third chunk is stored on third disk
    - Fourth chunk is stored on first disk
    - And so on



The striping of a file across three disks

Levels of RAID System

- **Level 0**: Striping but no redundancy (no R in RAID)
  - A striped array of *n* disks
  - The failure of a single disk ruins everything

RAID Levels (con't)

- **Level 1**: Mirrored Disks (no striping)
  - An array of *n* mirrored disks
    - All data stored on two disks
  - Increases reliability
    - If one disk fails, the system can continue
  - Increases speed of reads
    - Both of the mirrored disks can be read concurrently
  - Decreases speed of writes
    - Each write must be made to two disks
  - Requires twice the number of disks

RAID Levels (con't)

- **Level 3**: Data is striped over n disks and an $(n+1)^{th}$ disk is used to stores the exclusive or (XOR) of the corresponding bytes on the other n disks
  - The $(n+1)^{th}$ disk is called the parity disk
  - Chunks are bytes

Level 3 (con't)

- Redundancy increases reliability
  - Setting a bit on the parity disk to be the XOR of the bits on the other disks makes the corresponding bit on each disk the XOR of the bits on all the other disks, including the parity disk

    1 0 1 0 1 1 (parity disk)
  - If any disk fails, its information can be reconstructed as the XOR of the information on all the other disks

Level 3 (con't)

- Whenever a write is made to any disk, a write must by made to the parity disk

  New_Parity_Bit = Old_Parity_Bit XOR (Old_Data_Bit XOR New_Data_Bit)

- Thus each write requires 4 disk accesses
  - 2 reads and 2 writes
- The parity disk can be a bottleneck since all writes involve a read and a write to the parity disk

RAID Levels (con't)

- **Level 5:** Data is striped and parity information is stored as in level 3, but
  - The chunks are disk blocks
  - The parity information is itself striped and is stored in turn on each disk
    - Eliminates the bottleneck of the parity disk
  - Level 5 most often recommended for transaction processing applications

RAID Levels (con't)

- **Level 10**: A combination of levels 0 and 1 (not an official level)
  - A striped array of n disks (as in level 0)
  - Each of these disks is mirrored (as in level 1)
    - Achieves best performance of all levels
    - Requires twice as many disks

Controller Cache

- To further increase the efficiency of RAID systems, a controller cache can be used in memory
  - When reading from the disk, a larger number of disk blocks than have been requested can be read into memory
  - In *write back cache,* the RAID system reports that the write is complete as soon as the data is in the cache (before it is on the disk)
    - Requires some redundancy of information in cache
  - If all the blocks in a stripe are to be updated, the new value of the parity block can be computed in the cache and all the writes done in parallel

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

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

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

Heap File - Performance

- Assume file contains F pages
- Inserting a row:
  - Before the insert, we must ensure that A's key does not duplicate the key of a row already in the table.
    - If a duplicate exists, it will be discovered in F=2 page reads on average, and at that point the insertion is abandoned
  - If the row does not already exist:
    - The entire file has to be read in order to conclude that no duplicate is present, and then the last page (with A inserted) has to be rewritten, yielding a total cost of F + 1 page transfers
- Deleting a row:
  - Access path is scan
  - Avg. F/2+1 page transfers if row exists
  - F page transfers if row does not exist

Heap File - Performance

- Query
  - Access path is scan
  - Organization is efficient if query returns all rows and order of access is not important

    SELECT * FROM Transcript
  - Organization is inefficient if a *few* rows are requested
    - Average $F/2$ pages read to get get a single row

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

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
```

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
```

Transcript Stored as a Sorted File

111111	MGT123	F1994	4.0
111111	CS305	S1996	4.0
123456	CS305	F1995	2.0

page 0

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

page 1

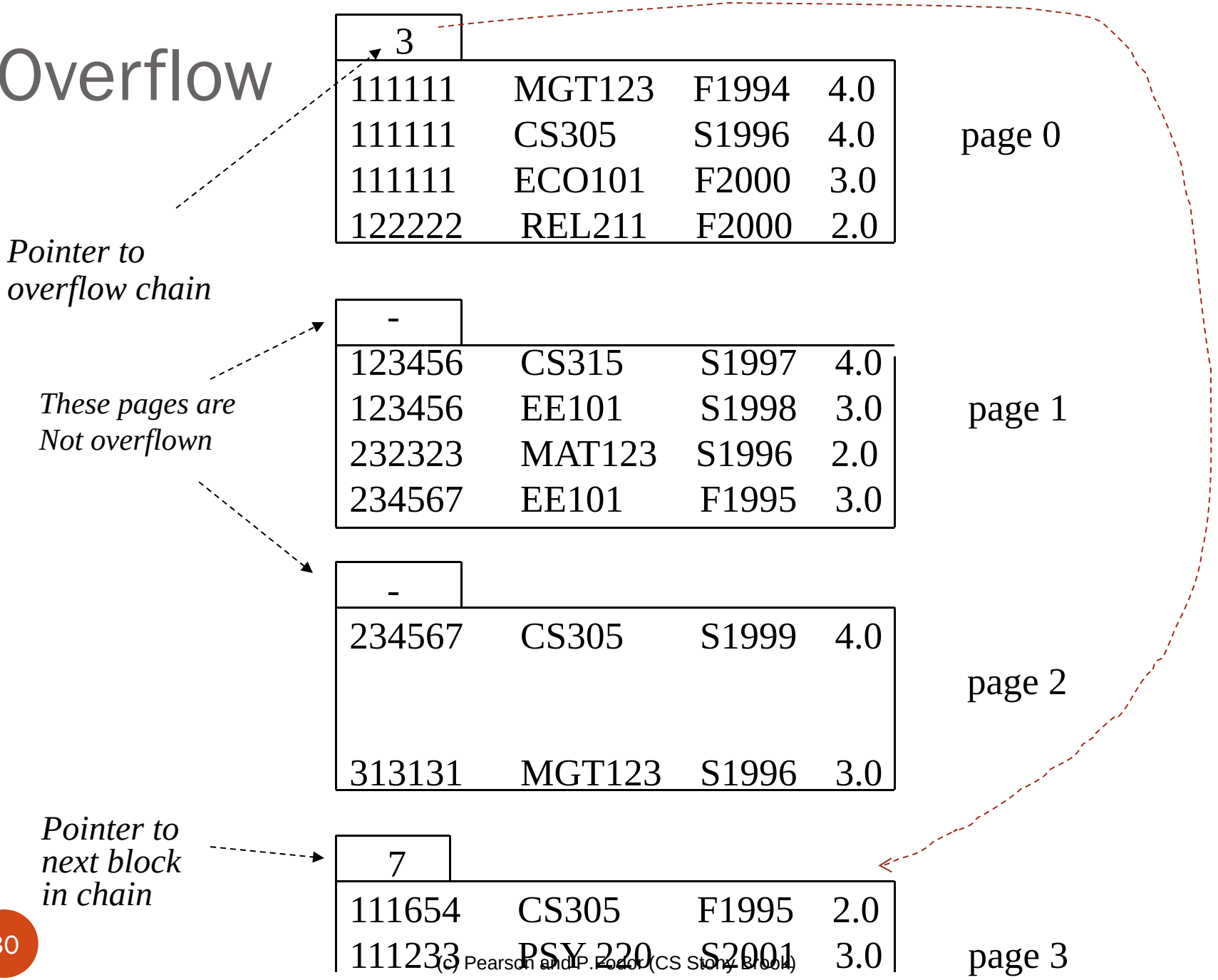
234567	CS305	S1999	4.0
313131	MGT123	S1996	3.0

page 2

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: *fillfactor*
- **Partial Solution 2**: Use *overflow pages* (*chains*).
  - Disadvantages:
    - Successive pages no longer stored contiguously
    - Overflow chain not sorted, hence cost no longer $log_2 F$

Overflow



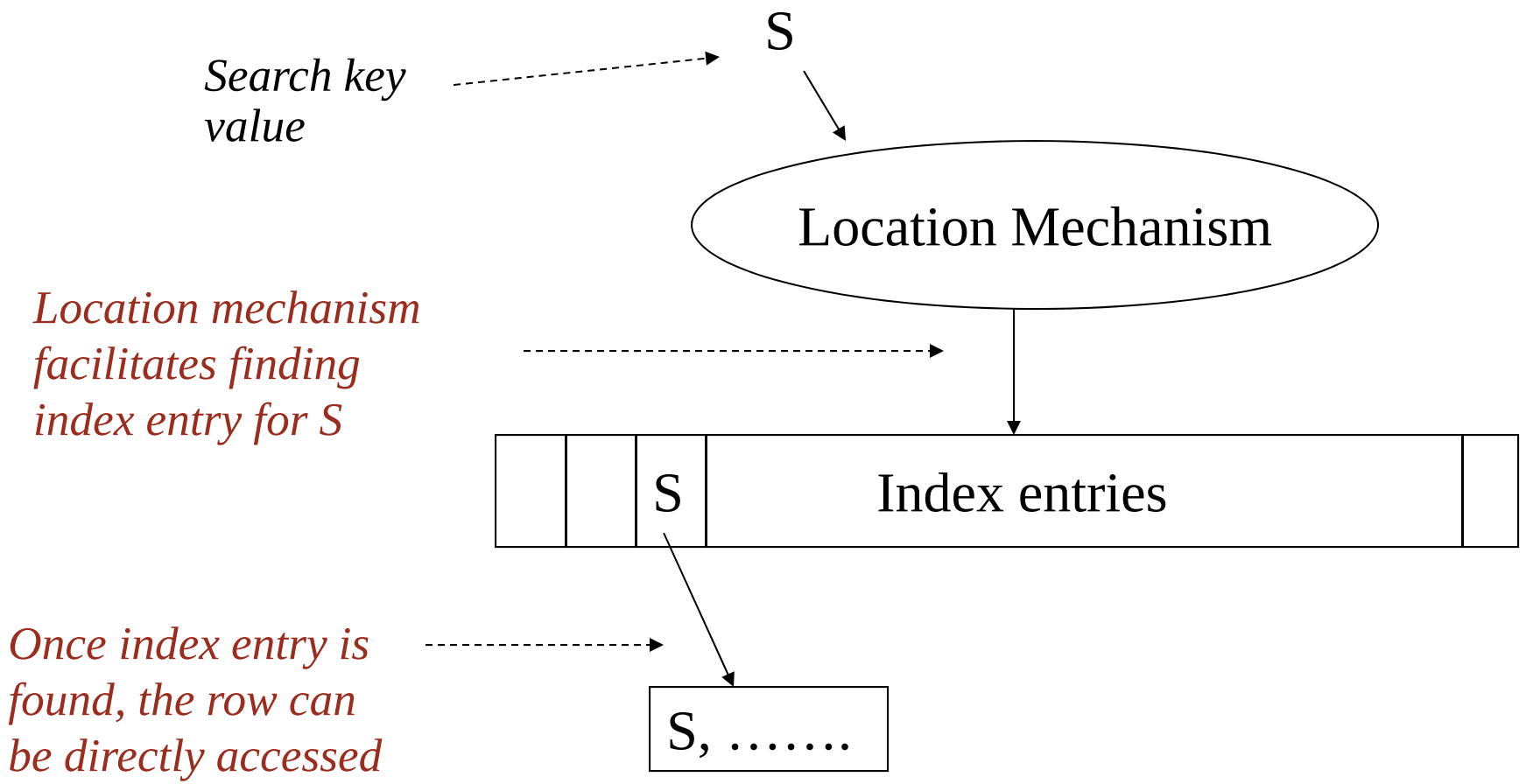
Index

- 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

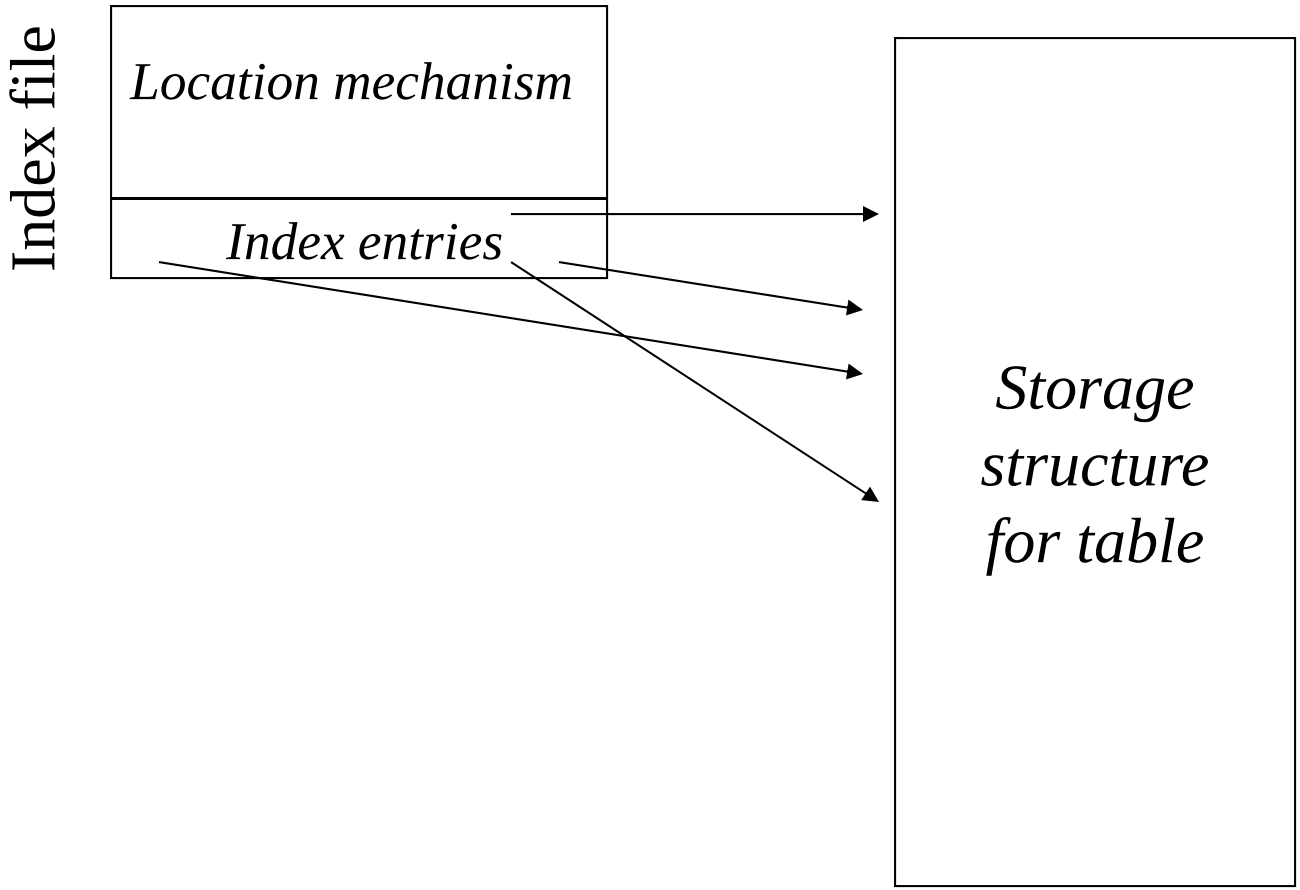
- Contains:
  - *Index entries*
    - Can contain the data tuple itself (index and table are *integrated* in this case); or
    - Search key value and a pointer to a row having that value; table stored separately in this case – *unintegrated* index
  - *Location mechanism*
    - Algorithm + data structure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



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)



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

Examples

- DROP INDEX CourseTran;
- CREATE INDEX CourseTran ON Transcript (CourseId);

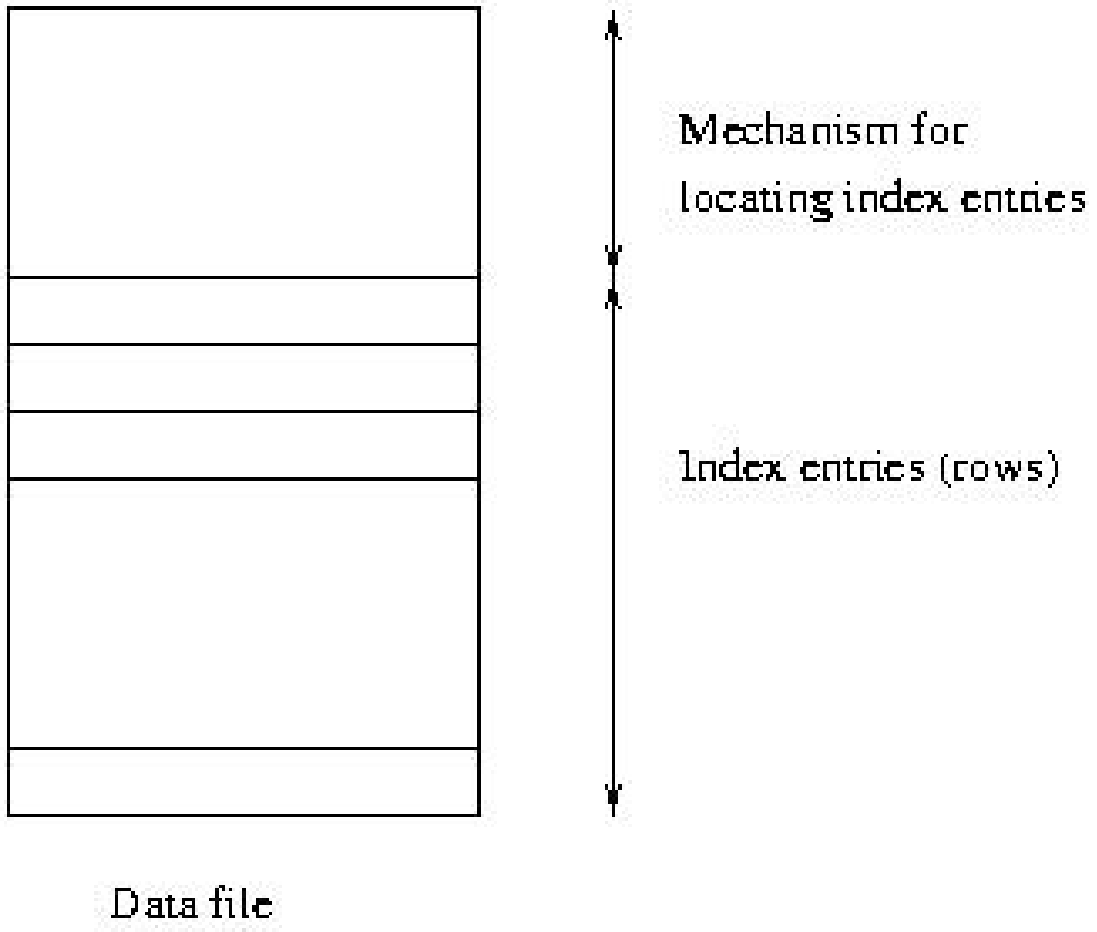
- DROP INDEX DeptProf;
- CREATE INDEX DeptProf ON Professor (DeptId);

Clustered Index

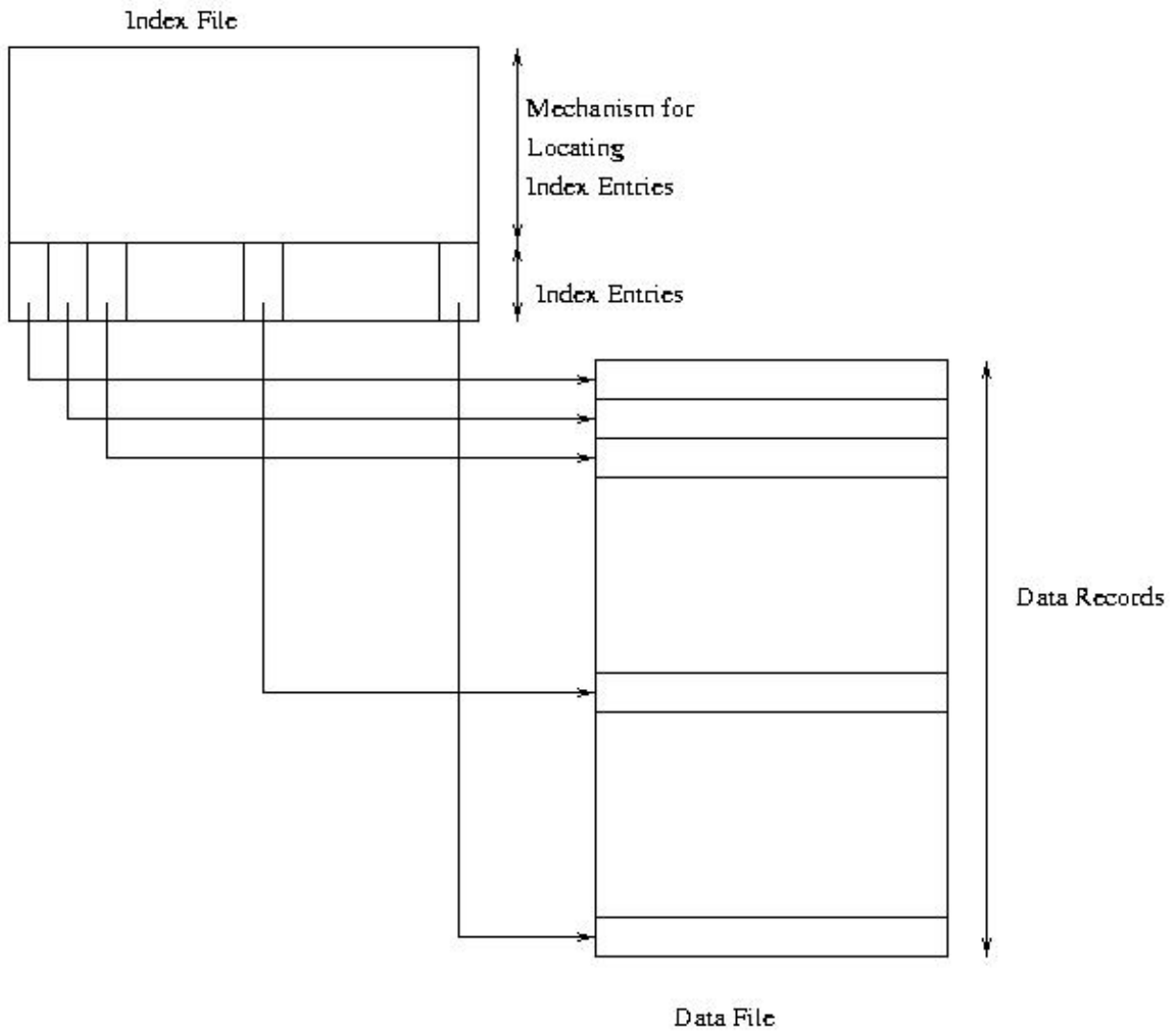
- *Clustered index*: index entries and rows are ordered in the same way
  - An integrated storage structure is always clustered (since rows and index entries are the same)
  - The particular index structure (eg, hash, tree) dictates how the rows are organized in the storage structure
    - There can 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



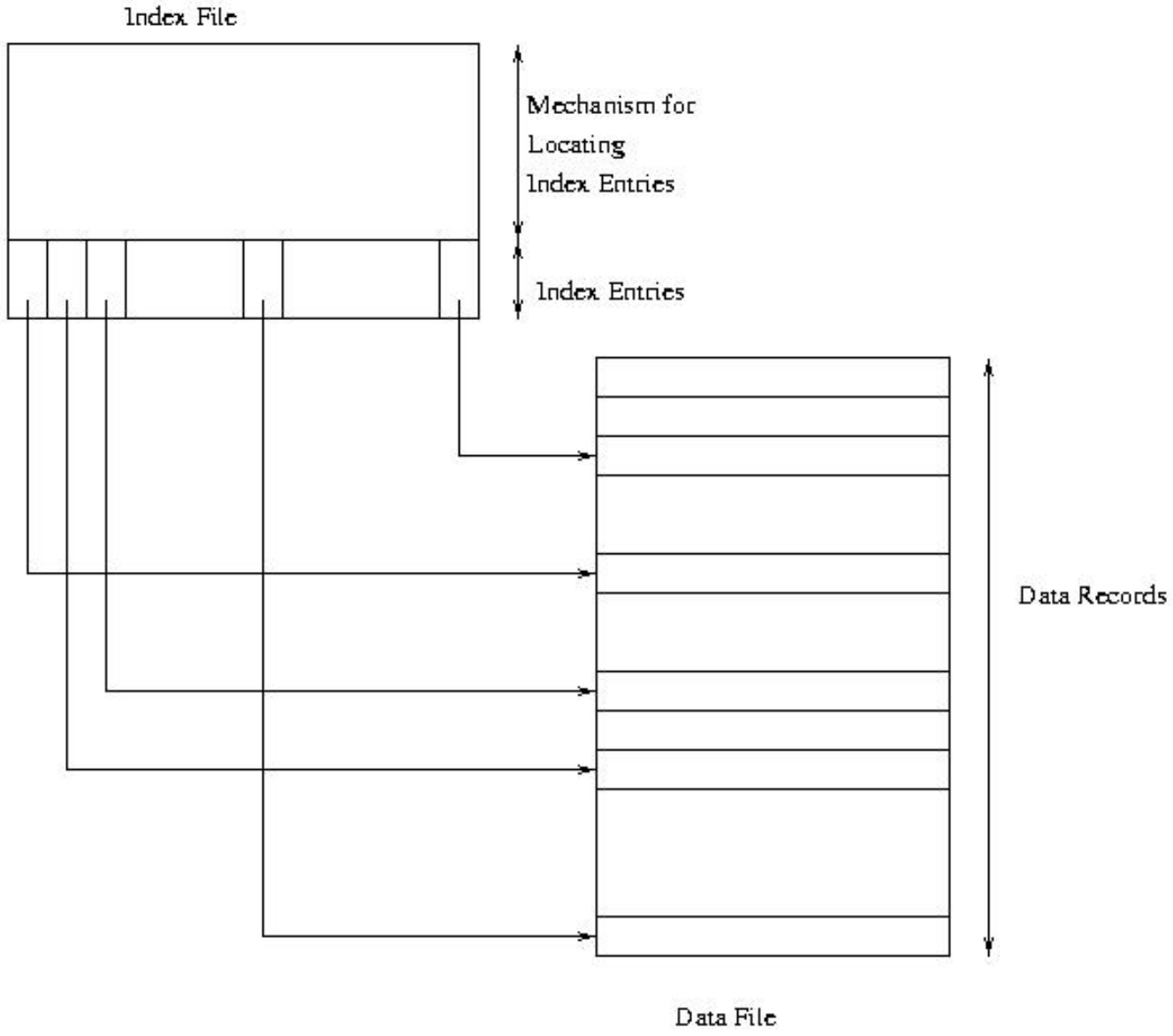
Clustered Secondary Index



Unclustered Index

- Unclustered (secondary) index: index entries and rows are not ordered in the same way
- An 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

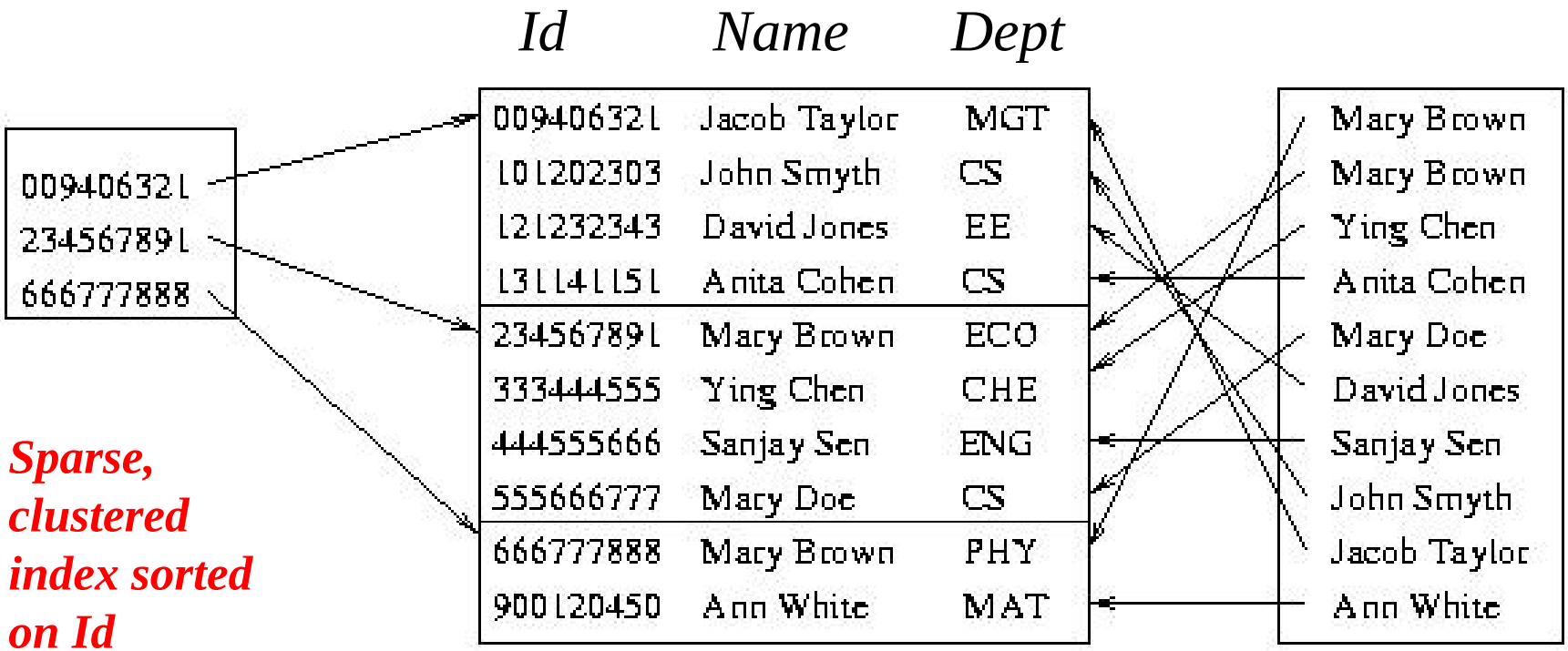
Example – Cost of Range Search

- 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 (to locate)+ (5 or 6 pages=~100rows) ≈ 19
  - Unclustered secondary index: ≤ 100 (range index)
  - Clustered index: 5 or 6 (constant to locate + pages~100rows)
- 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
  - Clustered index

Sparse Vs. Dense Index



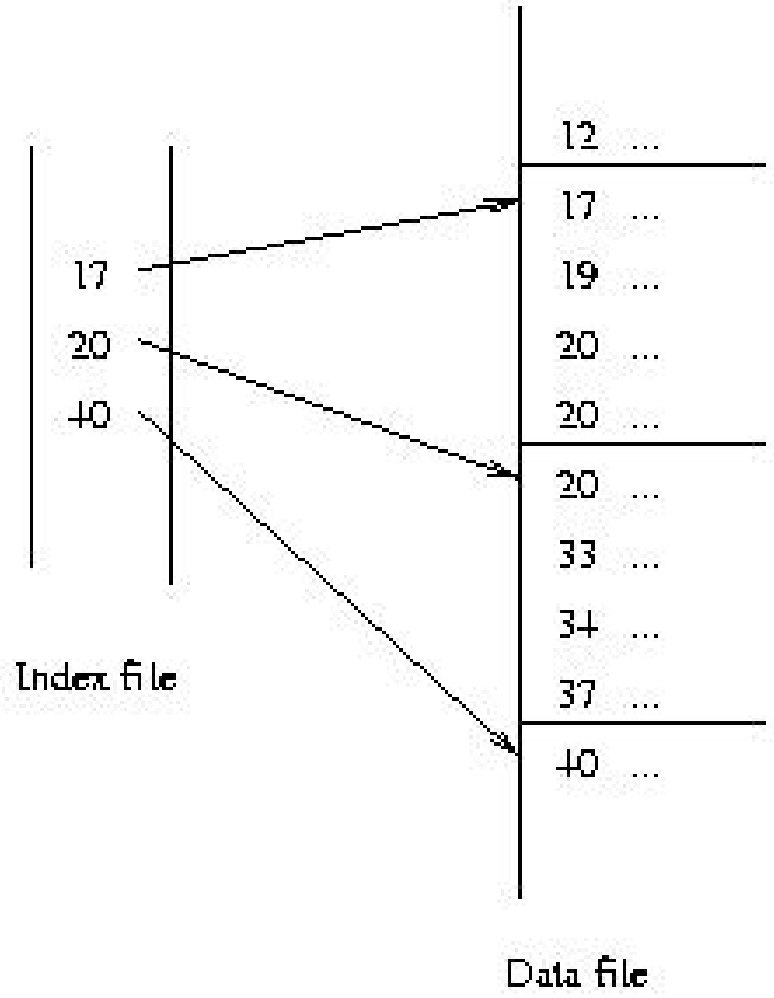
Sparse, clustered index sorted on Id

Data file sorted on Id

Dense, unclustered index sorted on Name

Sparse Index

Search key should be candidate key of data file



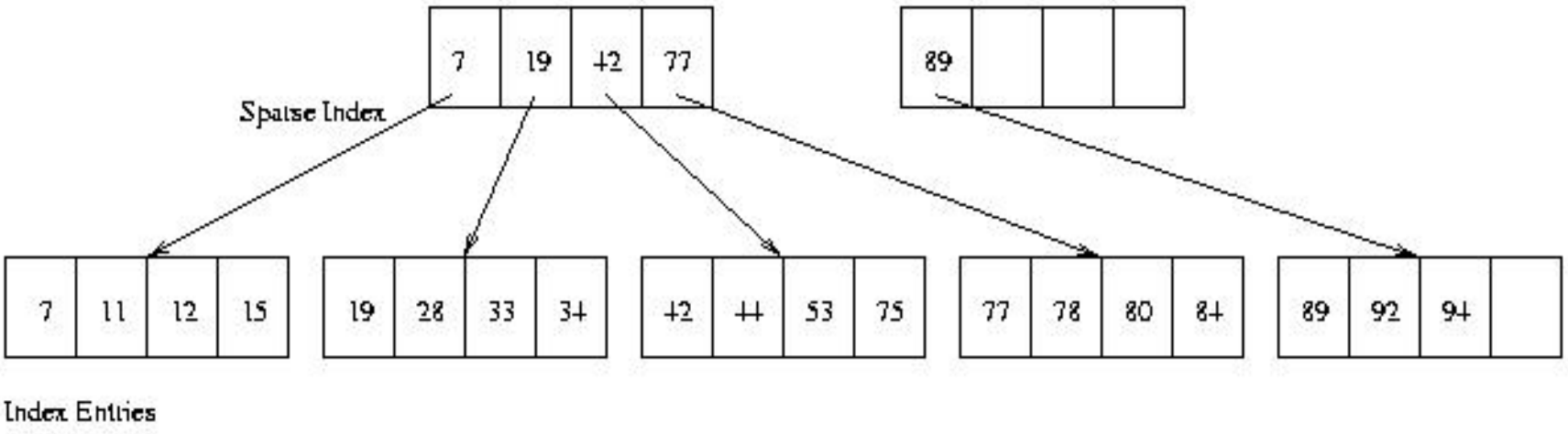
Multiple Attribute Search Key

- CREATE INDEX Inx ON Tbl (A*tt1*, A*tt2*)
- 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 (A1′, A2′) "
- Supports partial key searches (tree index only):
  - Find rows with values of A*tt1* between A1 and A1′
  - But not "Find rows with values of A*tt2* between A2 and A2′ "

Locating an Index Entry

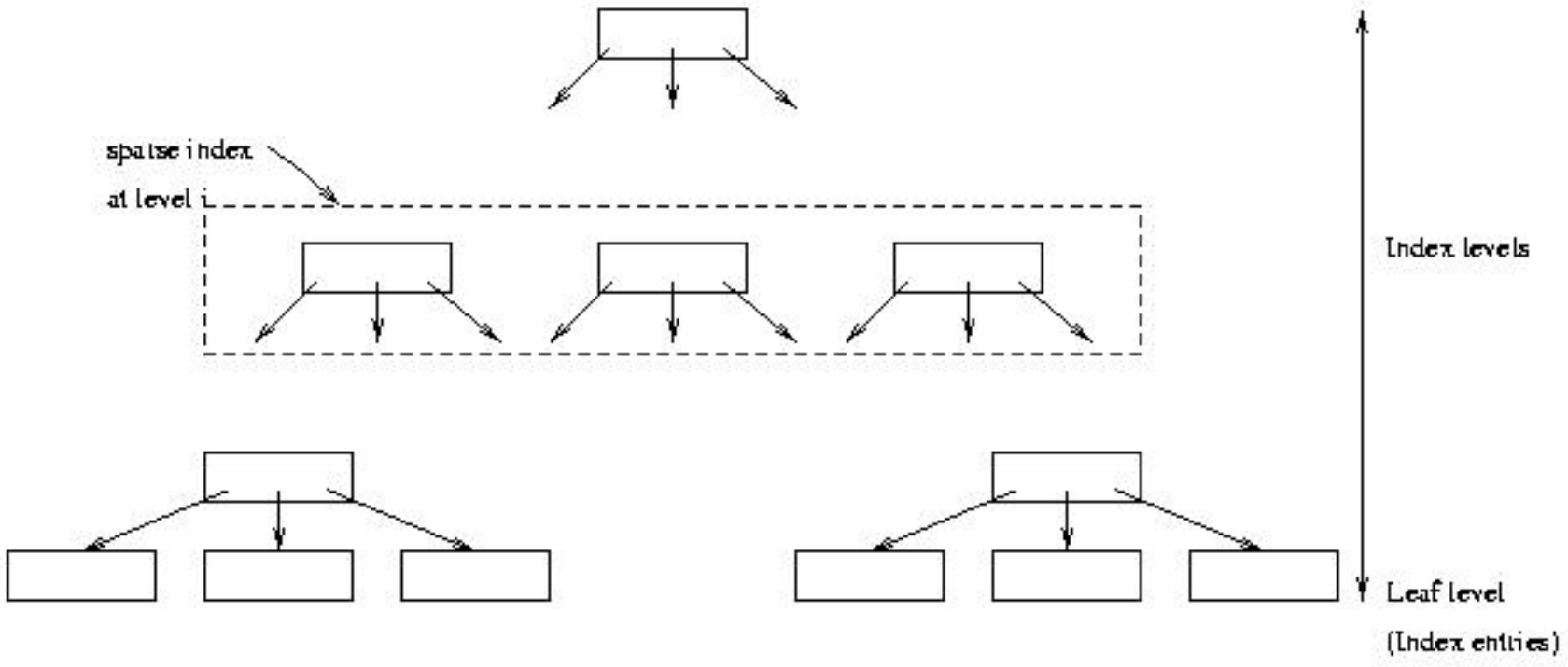
- Use binary search (index entries sorted)
  - If Q pages of index entries, then $log_2 Q$ page transfers (which is a big improvement over binary search of the data pages of a F page data file since $F >> Q$)
- Use multilevel index: Sparse index on sorted list of index entries

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)

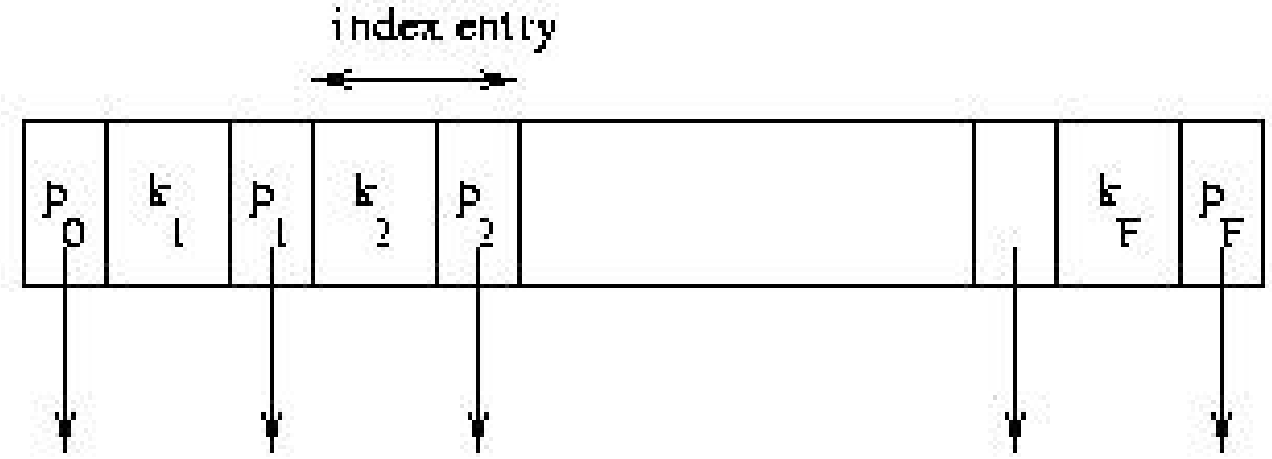
Multilevel Index



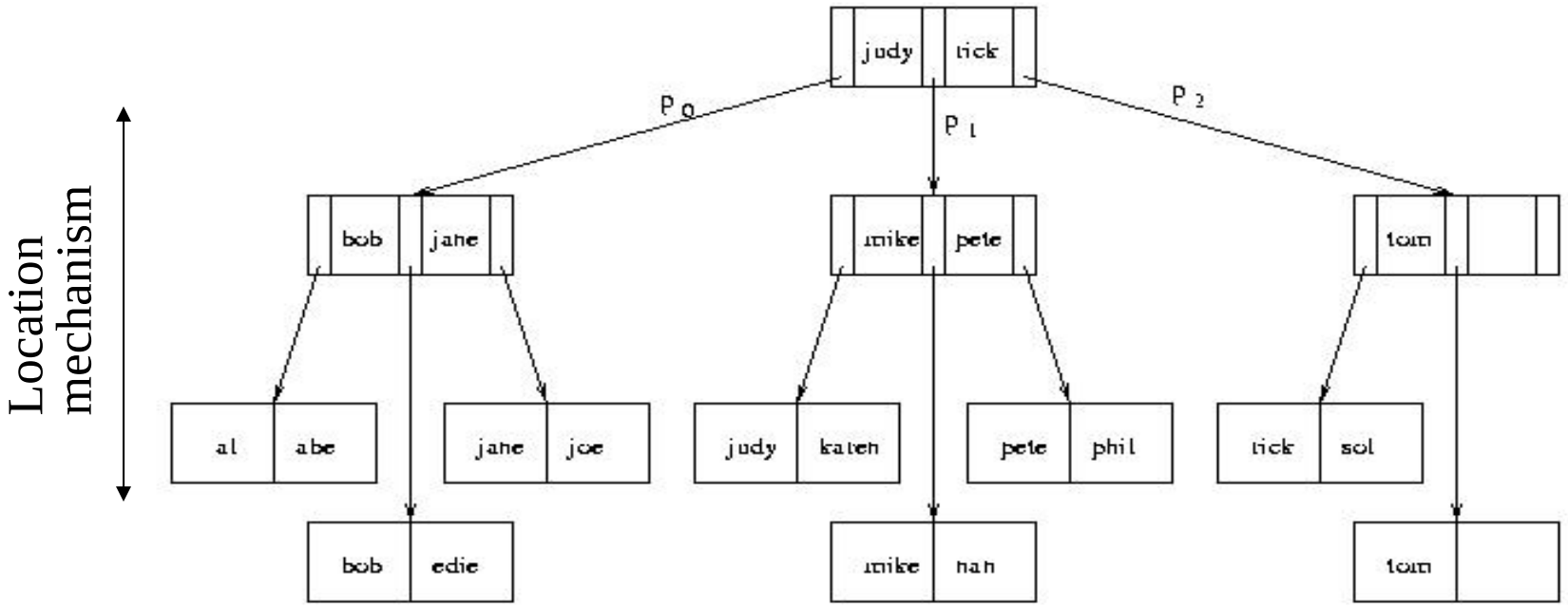
- Search cost = number of levels in tree
- If Φ is the fanout of a separator page, cost is $log_{\Phi} Q + 1$
- Example: if $\Phi = 100$ and $Q = 10{,}000$, cost = 3
  (reduced to 2 if root is kept in main memory)

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

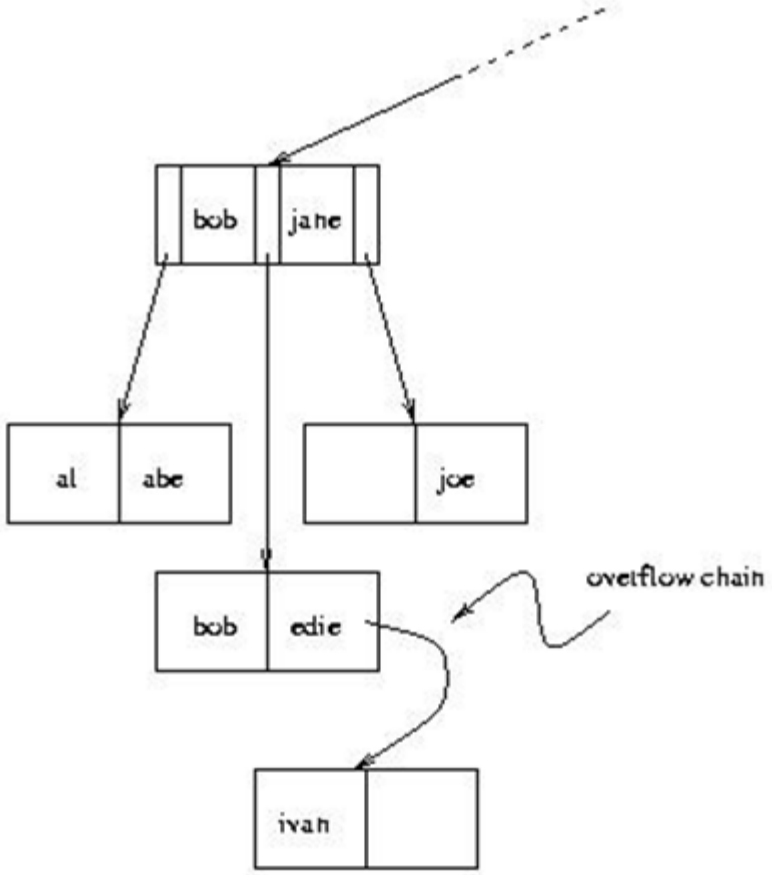


Index Sequential Access Method

- 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

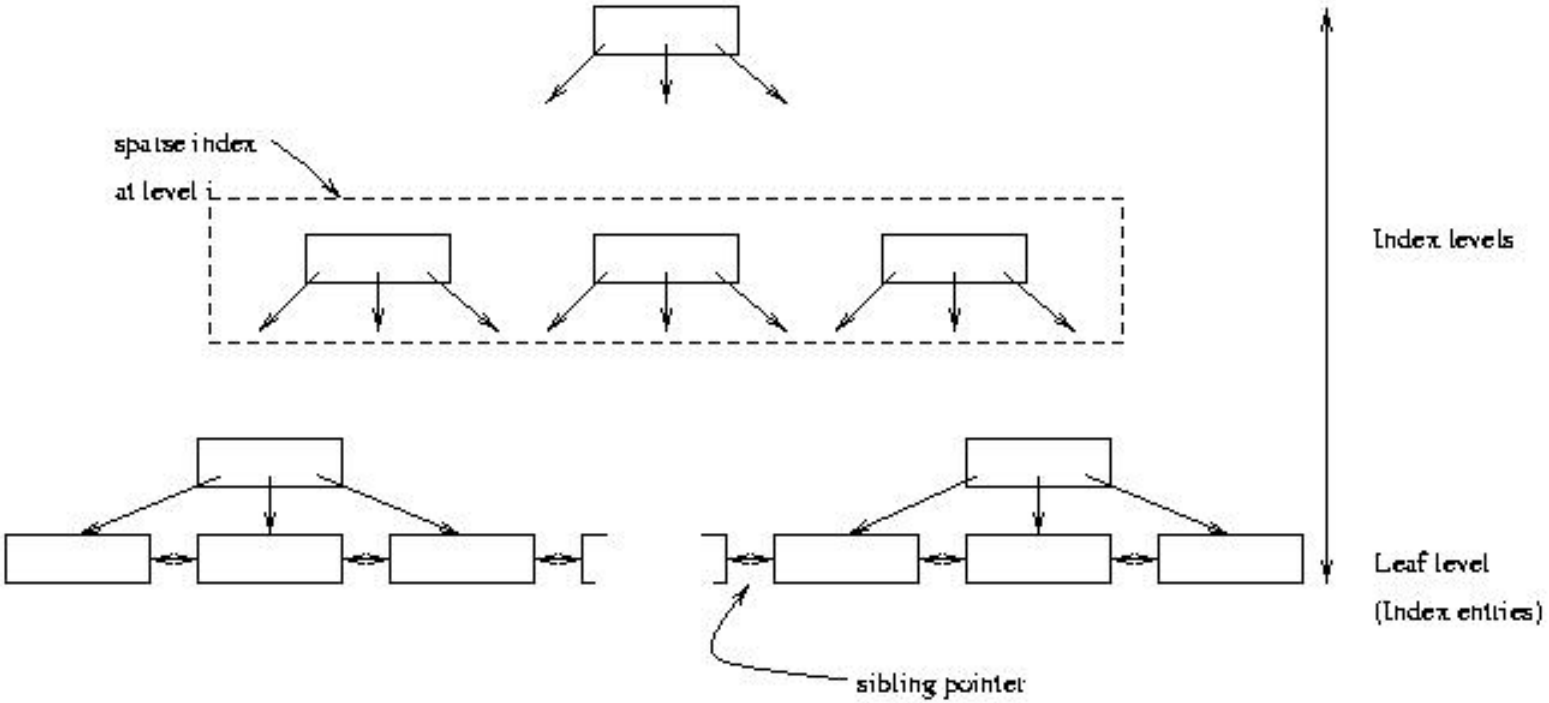
- Contents of leaf pages change
  - Row deletion yields empty slot in leaf page
  - Row insertion can result in overflow leaf page and ultimately overflow chain
    - *Chains can be long, unsorted, scattered on disk*
    - *Thus ISAM can be inefficient if table is dynamic*



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



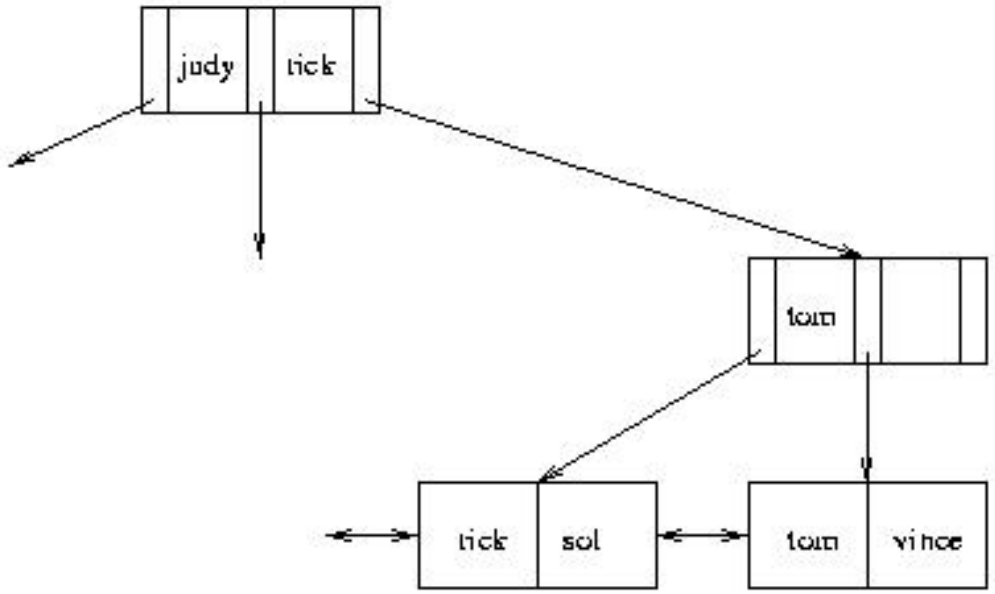
– Leaf level is a (sorted) linked list of index entries
– Sibling pointers support range searches in spite of allocation and deallocation of leaf pages (but leaf pages might not be physically contiguous on disk)

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)

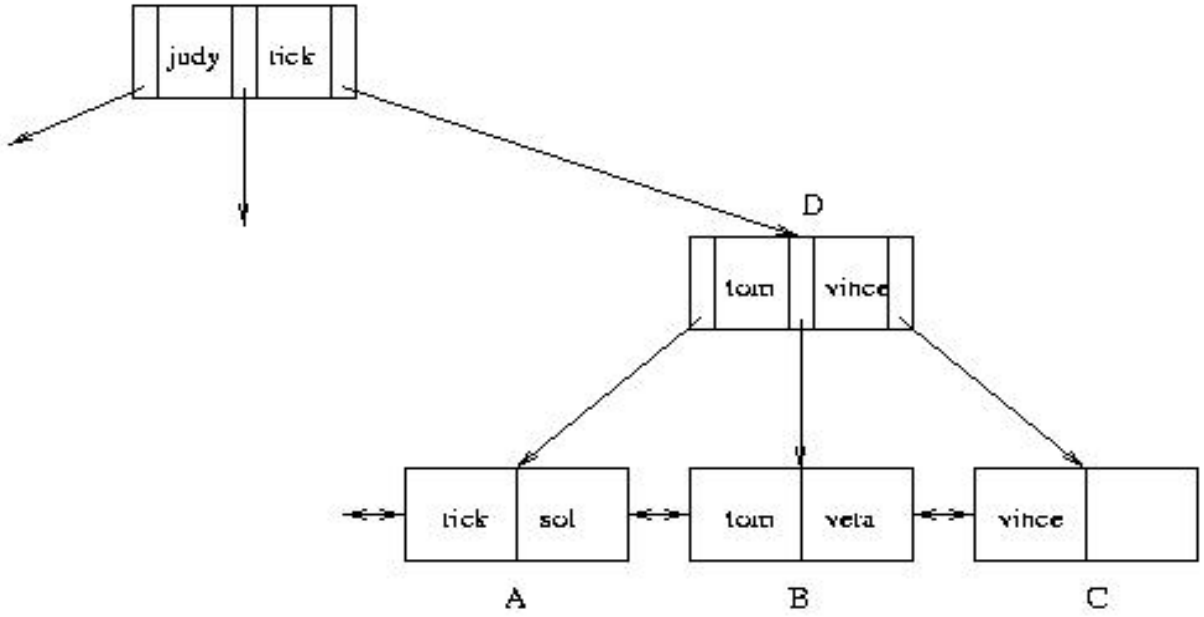
Handling Insertions - Example

- Insert “vince”



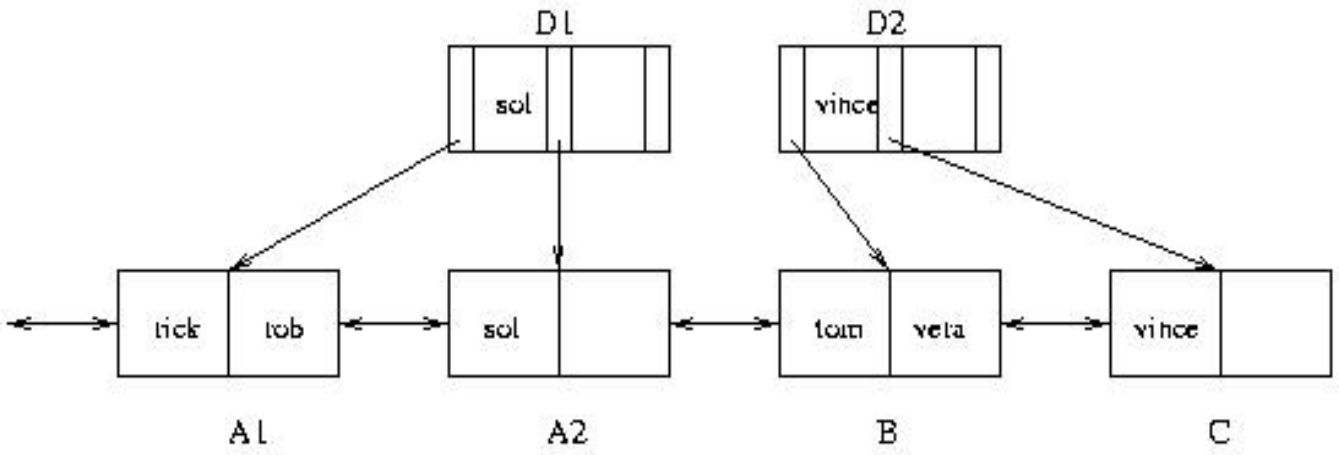
Handling Insertions (cont'd)

– 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



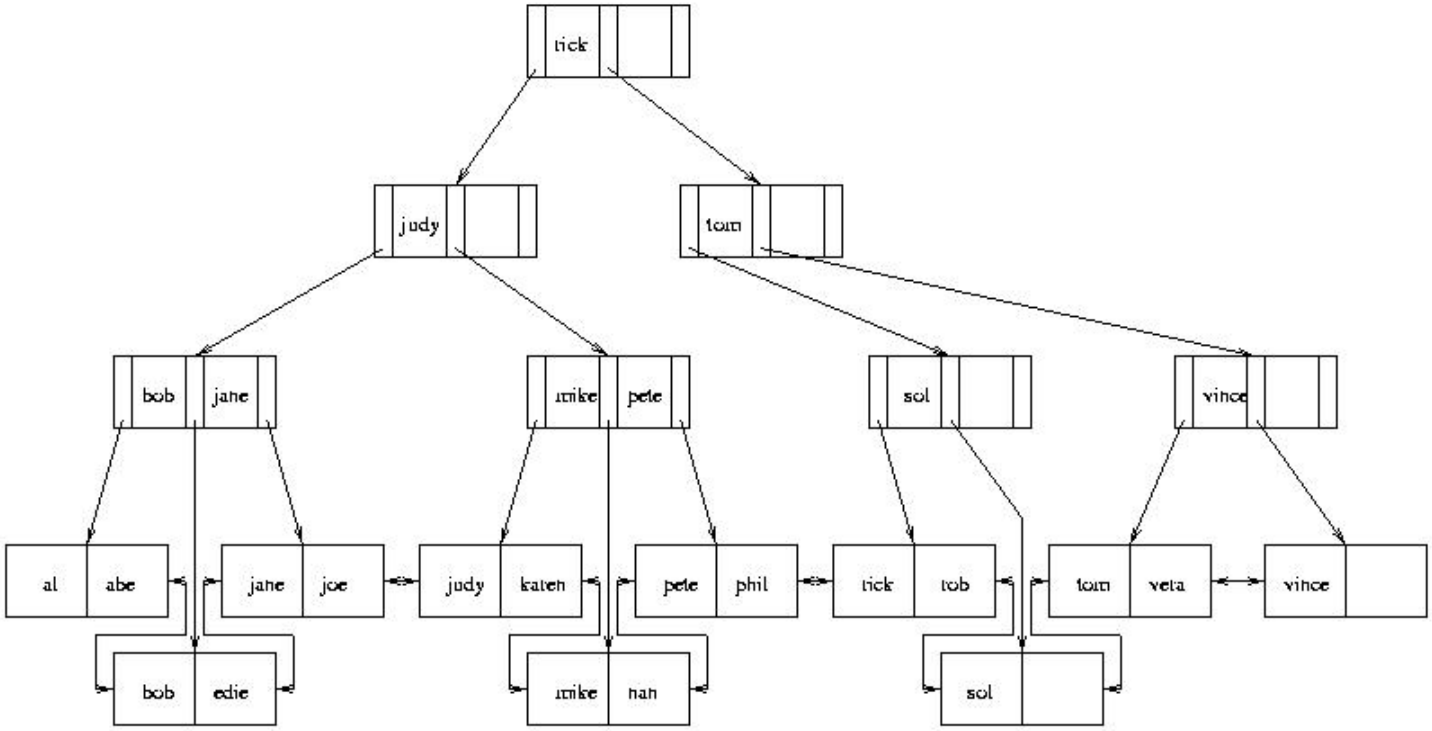
Handling Insertions (con't)

– 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

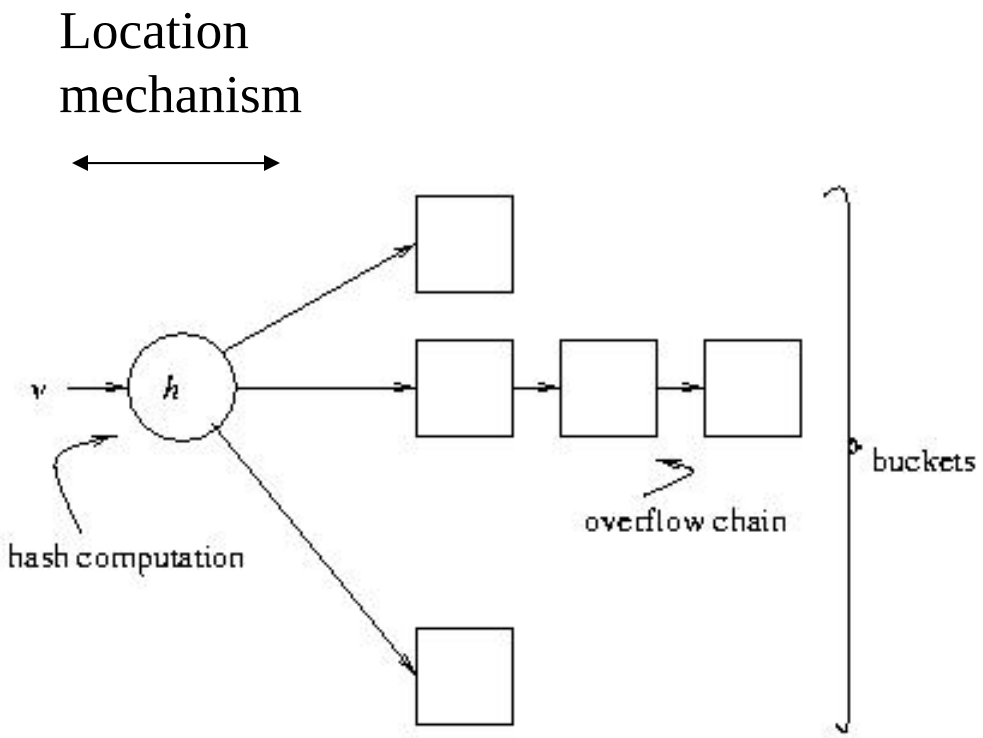
- Index entries partitioned into *buckets* in accordance with a *hash function*, $h(v)$, where v ranges over search key values
  - Each bucket is identified by an address, a
  - Bucket at address a contains all index entries with search key v such that $h(v) = a$
- Each bucket is stored in a page (with 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: $h(v) = (c_1 * v + c_2) \bmod M$
  - 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

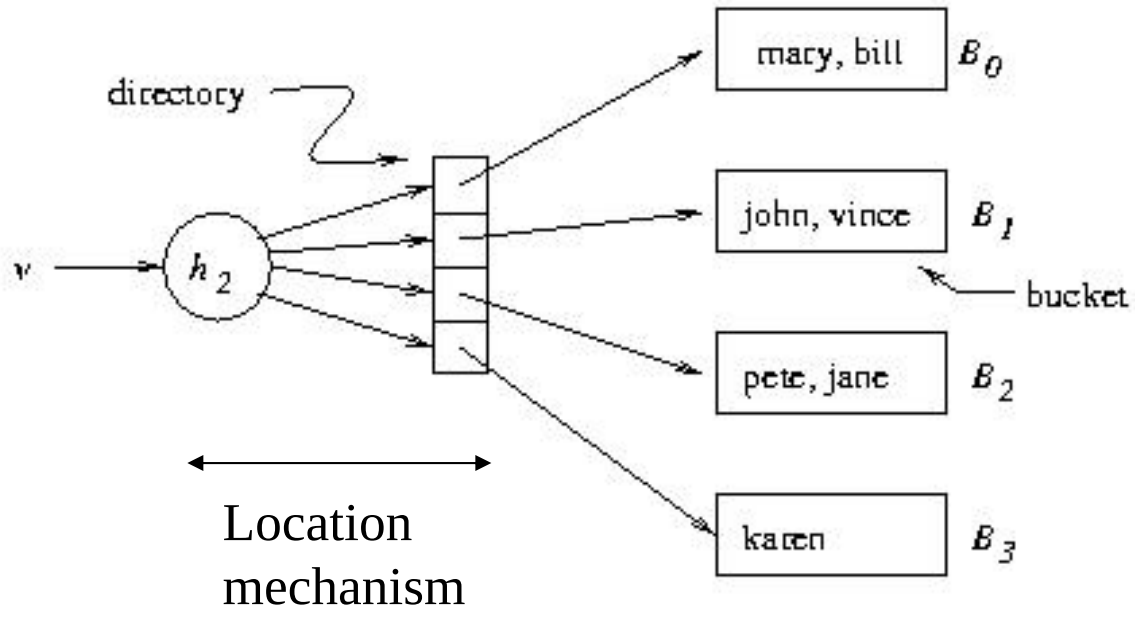
Hash Indices – Problems

- 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 given time a unique hash, h_k , is used depending on the number of times buckets have been split

Extendable Hashing – Example



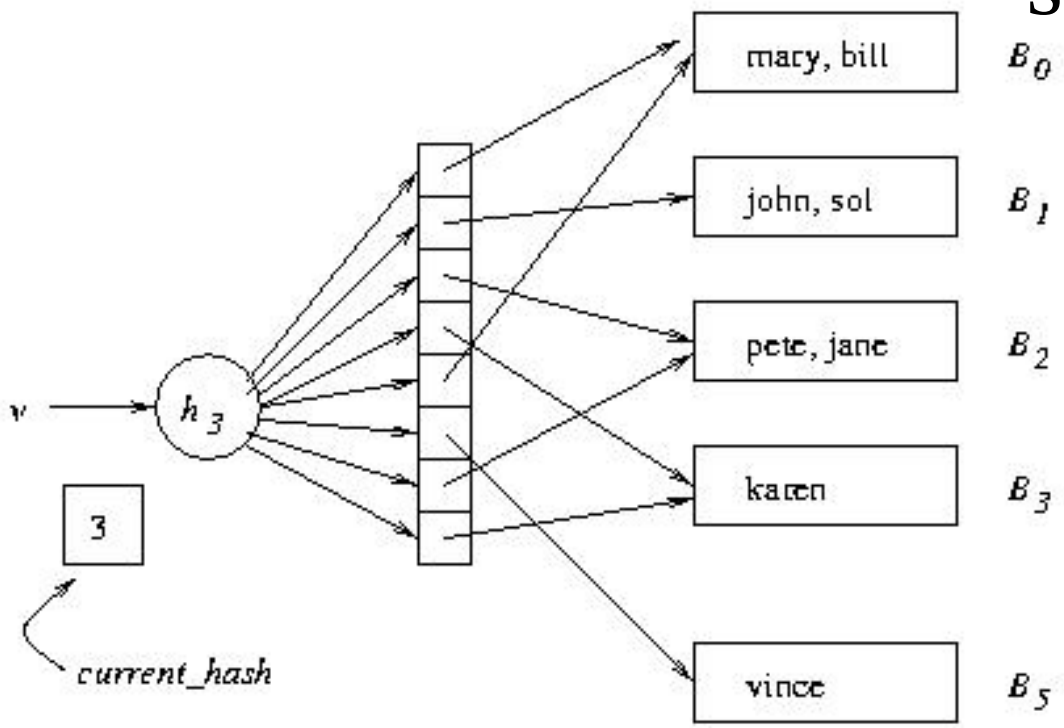
v	*h*(*v*)
pete	11010
mary	00000
jane	11110
bill	00000
john	01001
vince	10101
karen	10111

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

Example (cont'd)

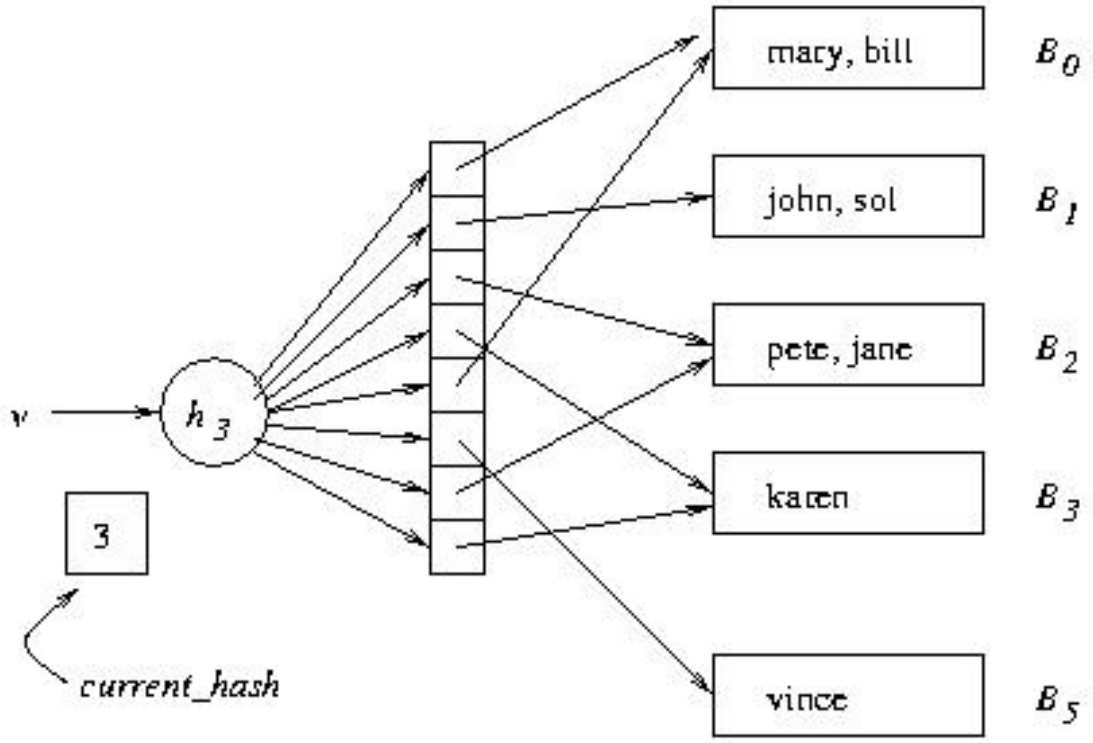


Solution:

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.

Example (cont'd)

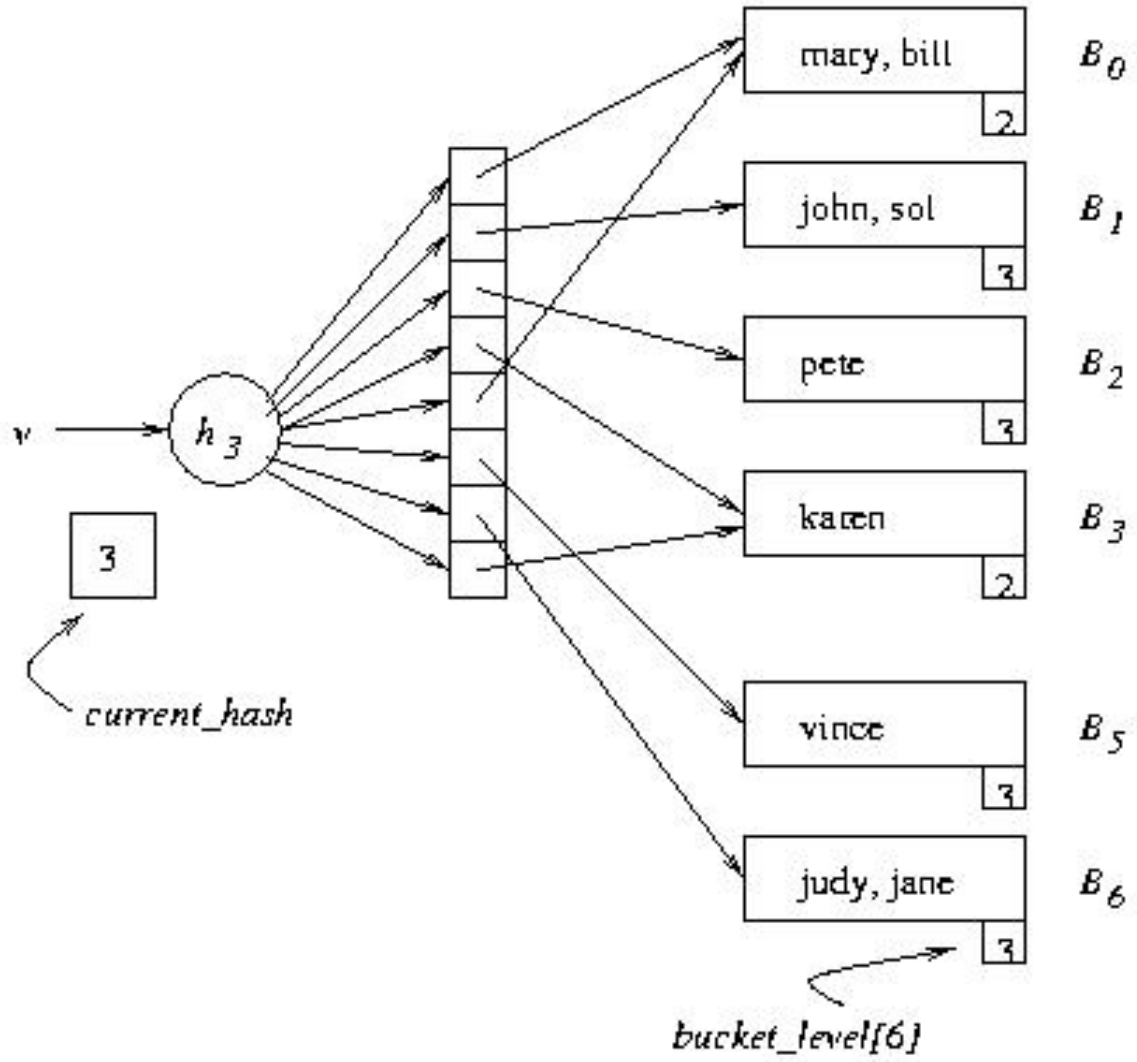


Next action: Insert judy, where $h(judy) = 00110$
B_2 overflows, but directory need not be extended

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

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

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*

Choosing An Index (cont'd)

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 (cont'd)

Example 3:

```
SELECT   T.CrsCode, T.Grade
FROM     Transcript T
WHERE    T.StudId = :id  AND  T.Semester = 'F2000'
```

– 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 (cont'd)

Example 3 (cont'd):

```
SELECT  T.CrsCode, T.Grade
FROM  Transcript T
WHERE  T.StudId = :id  AND  T.Semester = 'F2000'
```

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