Stony Brook University

CSE 305 / CSE532

Lecture 17 (Chapter 9)

Physical Data Organization and Indexing

Lecturer: Sael Lee

Slide adapted from the author's and Dr. Ilchul Yoon's slides.

SUNY Korea

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

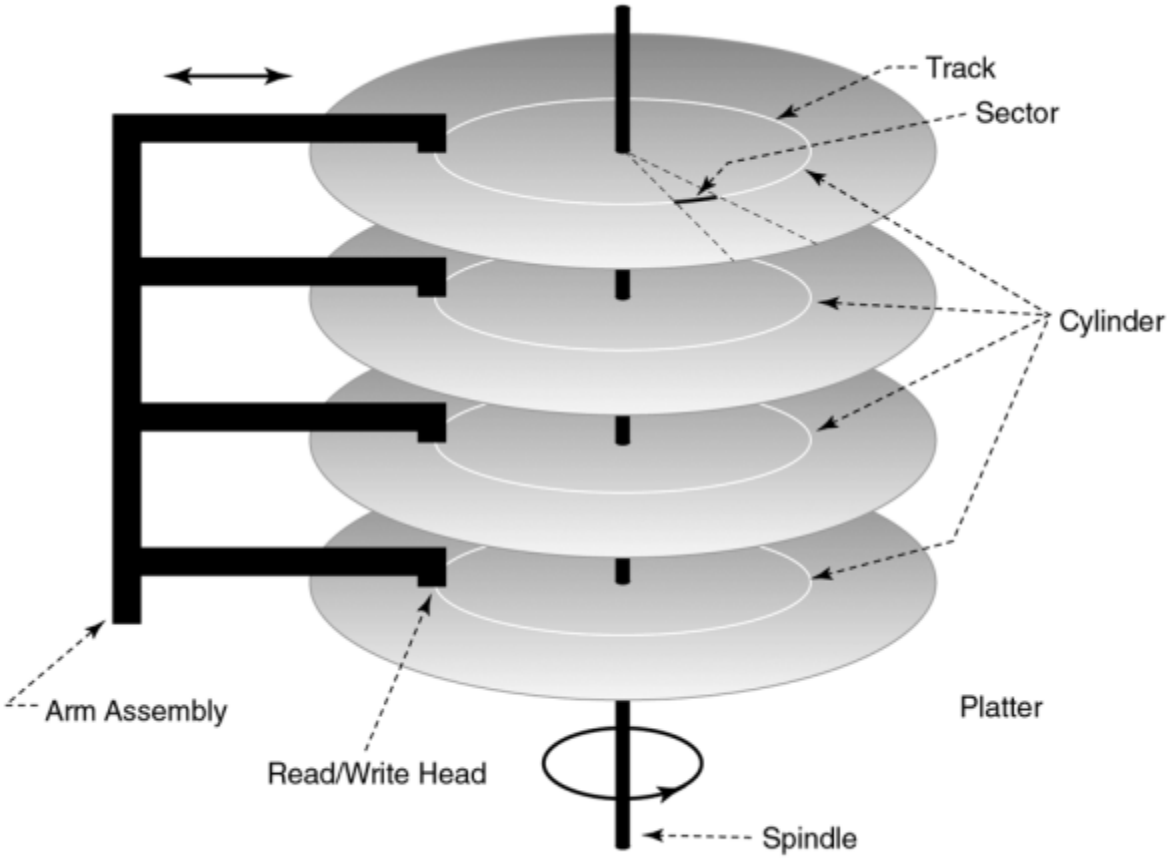


FIGURE 9.1 Physical organization of a disk storage unit.

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
  - 1st HDD: 600ms, mid-70s: 25ms, in 80s: 20ms,
  - ~2013: 4~10ms, SSD: 0.08~0.16ms
- *Rotational latency* – additional time for platters to rotate so that start of block containing page is under head
  - For 7200rpm HDD, ~ 4ms
- *Transfer time* – time for platter to rotate over block containing page (depends on size of block)
  - Small

- *Latency* = seek latency + rotational latency
- *Goal: minimize average latency, reduce number of page transfers*

Reducing Latency

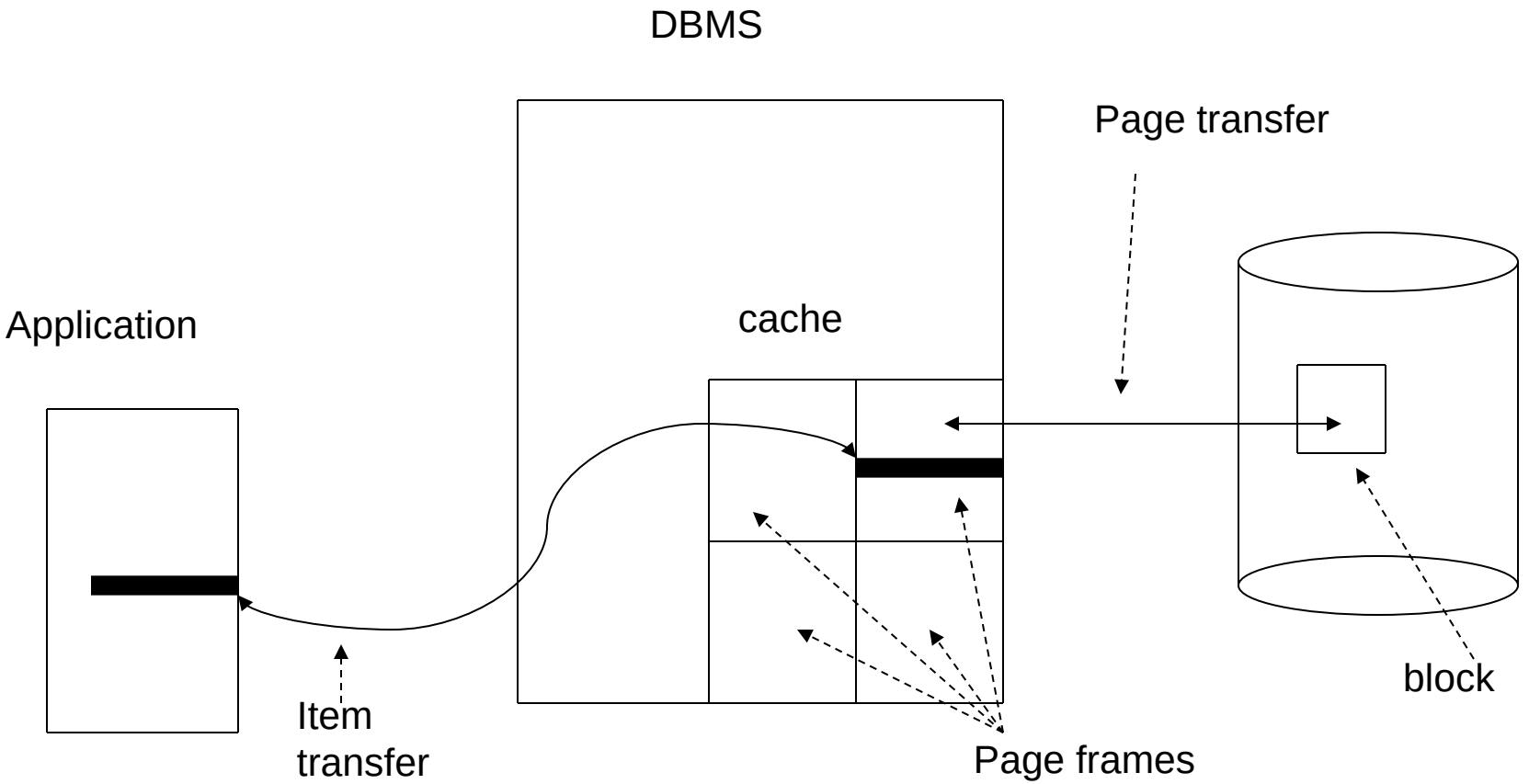
- Store pages containing related information close together on disk
  - *Justification*: If application accesses *x,* it will next access data related to *x* with high probability

- 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 (least recently used) algorithm
    - Record clean/dirty state of page (clean pages don't have to be written)

Accessing Data Through Cache



RAID Systems

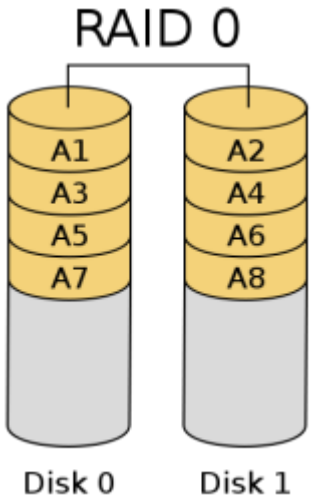
- RAID (Redundant Array of Independent Disks) is an array of disks configured to behave like a single disk with
  - Higher throughput
    - Multiple requests to different disks can be handled independently
    - If a single request accesses data that is stored separately on different disks, that data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

Striping

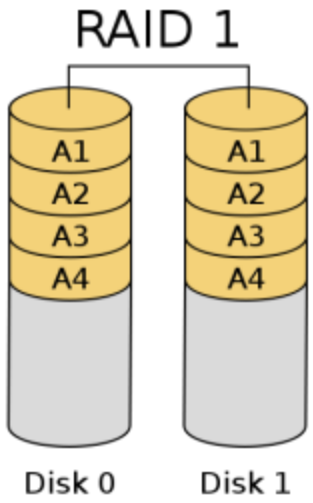
- Data that is to be stored on multiple disks is said to be *striped*
  - Data is divided into *chunks*
    - Chunks might be bytes, disk blocks etc.
  - If a file is to be stored on three disks
    - First chunk is stored on first disk
    - Second chunk is stored on second disk
    - Third chunk is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

Levels of RAID System

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

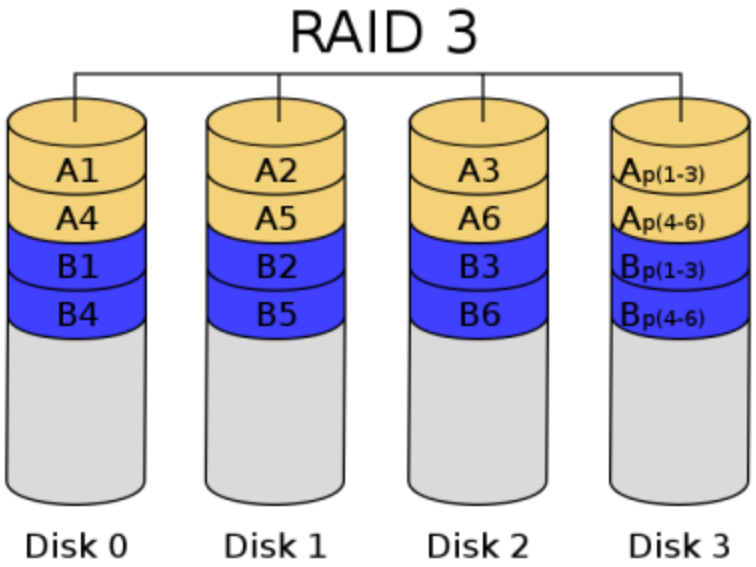


- **Level 1:** Mirrored Disks (no striping)
  - An array of n mirrored disks - all data stored on two disks
  - Increases reliability?
    - If one disk fails, the system can continue
  - Increases speed of reads?
    - Both disks can be read concurrently
  - Decreases speed of writes?
    - Each write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



RAID Levels (con't)

- **Level 3**: bit-interleaved parity
  - 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)

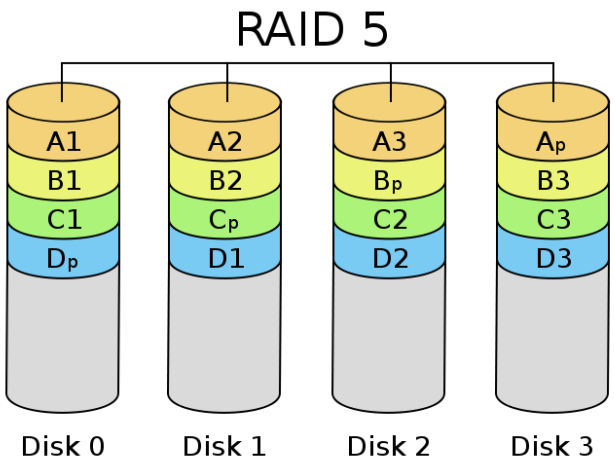
- Redundancy increases reliability
  - Setting a bit on the parity disk to be the XOR of the bits on the other disks makes the corresponding bit on each disk the XOR of the bits on all the other disks, including the parity disk

    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
- Whenever a write is made to any disk, a write must be 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
- *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:** Block-interleaved distributed parity
  - 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 most often recommended for transaction processing applications



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

- *However, 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

666666666	MGT123	F1994	4.0
123454321	CS305	S1996	4.0
987654321	CS305	F1995	2.0
111111111	MGT123	F1997	3.0

page 0

123454321	CS315	S1997	4.0
666666666	EE101	S1991	3.0
123454321	MAT123	S1996	2.0
234567890	EE101	F1995	3.0

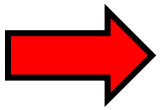
page 1

234567890	CS305	S1996	4.0
111111111	EE101	F1997	4.0
111111111	MAT123	F1997	3.0
987654321	MGT123	F1994	3.0

page 2

425360777	CS305	S1996	3.0
666666666	MAT123	F1997	3.0

page 3



666666666	MGT123	F1994	4.0
123454321	CS305	S1996	4.0
987654321	CS305	F1995	2.0

123454321	CS315	S1997	4.0
666666666	EE101	S1991	3.0
123454321	MAT123	S1996	2.0
234567890	EE101	F1995	3.0

234567890	CS305	S1996	4.0
987654321	MGT123	F1994	3.0

425360777	CS305	S1996	3.0
666666666	MAT123	F1997	3.0
666666666	CS305	S1998	3.0

FIGURE 9.3 TRANSCRIPT table stored as a heap file. At most four row

FIGURE 9.4 TRANSCRIPT table of Figure 9.3 aft

Heap File - Performance

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

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

- 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

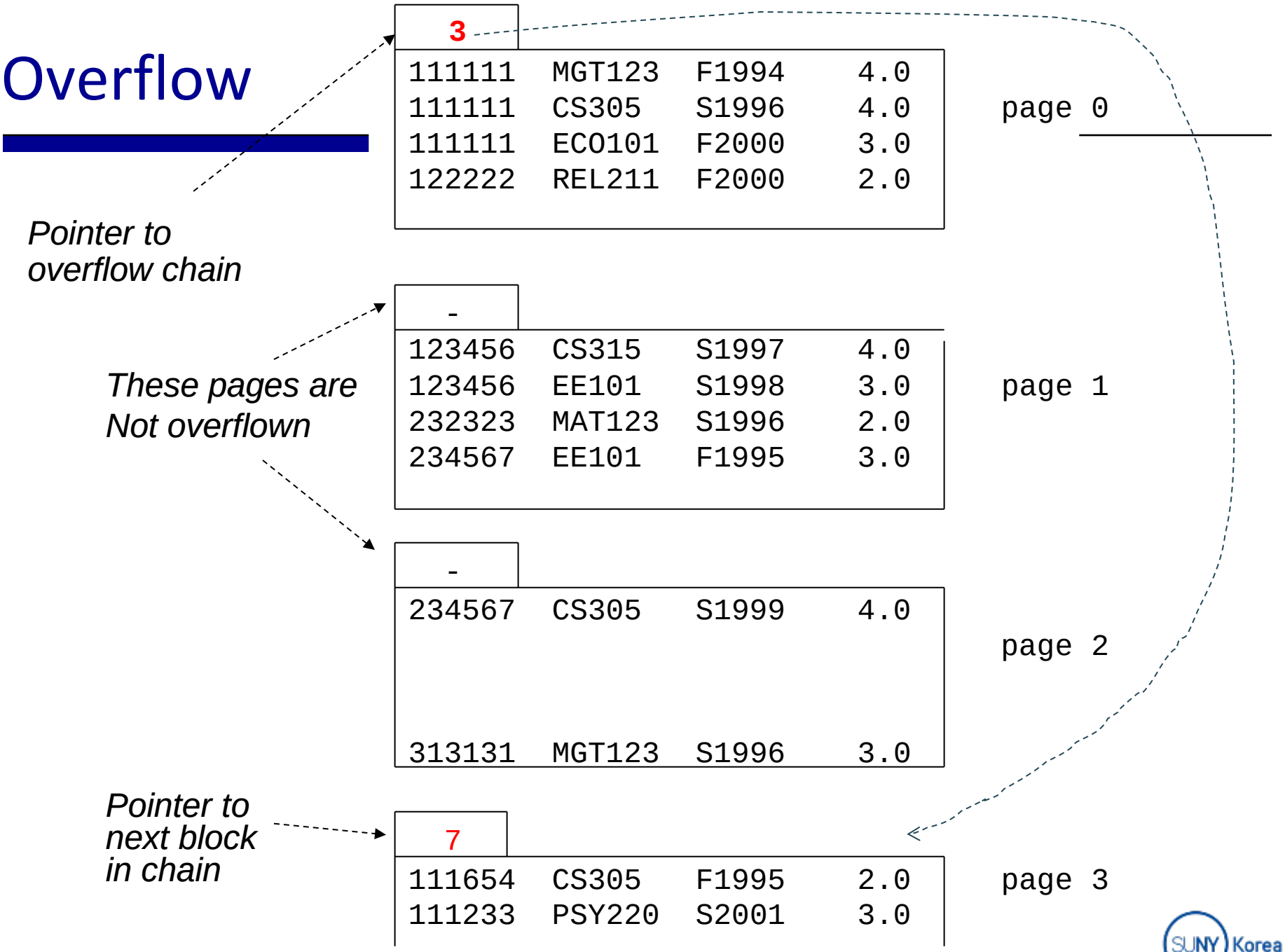
111111111	MGT123	F1997	3.0	
111111111	EE101	F1997	4.0	page 0
111111111	MAT123	F1997	3.0	
123454321	CS305	S1996	4.0	
123454321	CS315	S1997	4.0	page 1
123454321	MAT123	S1996	2.0	
234567890	EE101	F1995	3.0	
234567890	CS305	S1996	4.0	page 2
425360777	CS305	S1996	3.0	
666666666	MGT123	F1994	4.0	
666666666	MAT123	F1997	3.0	page 3
666666666	EE101	S1991	3.0	
987654321	MGT123	F1994	3.0	
987654321	CS305	F1995	2.0	page 4

FIGURE 9.5 TRANSCRIPT table stored as a sorted file. At most four rows fit in a page.

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


3
111111 MGT123 F1994 4.0
111111 CS305 S1996 4.0
111111 ECO101 F2000 3.0
122222 REL211 F2000 2.0
page 0
Pointer to overflow chain
-
123456 CS315 S1997 4.0
123456 EE101 S1998 3.0
232323 MAT123 S1996 2.0
234567 EE101 F1995 3.0
page 1
These pages are Not overflown
-
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 PSY220 S2001 3.0
page 3

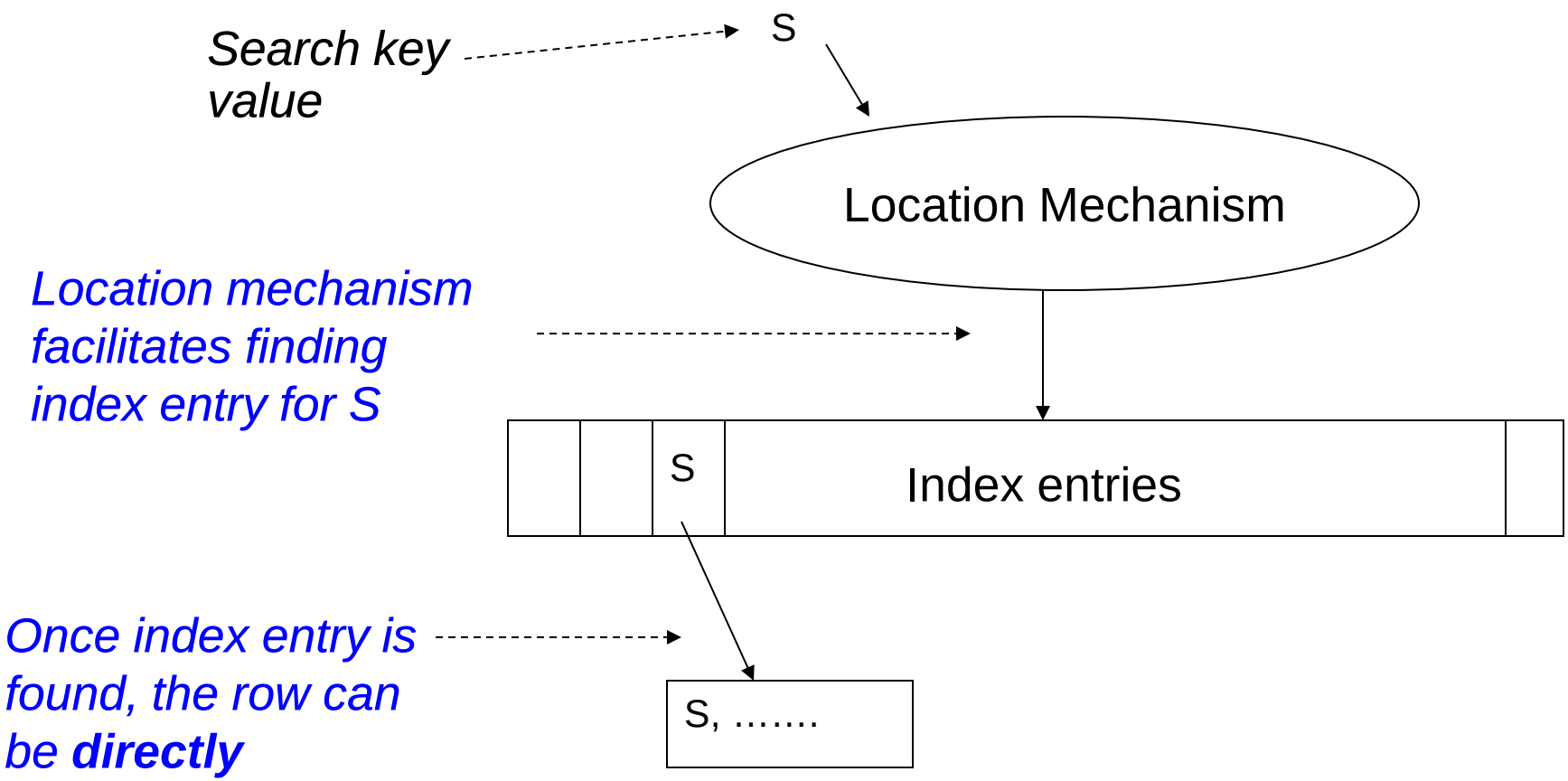
Index

- 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

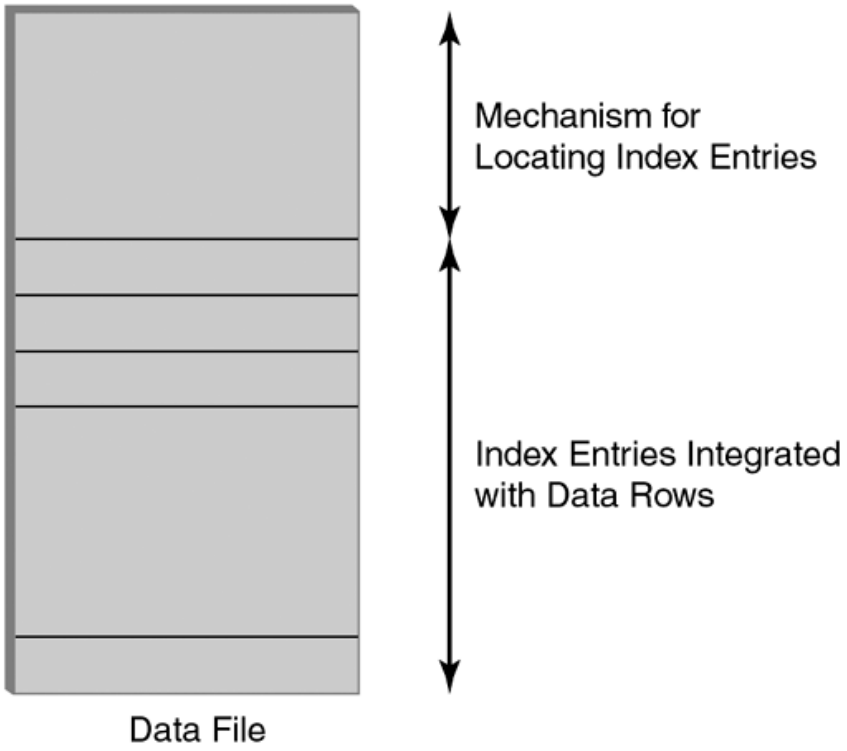


Storage Structure

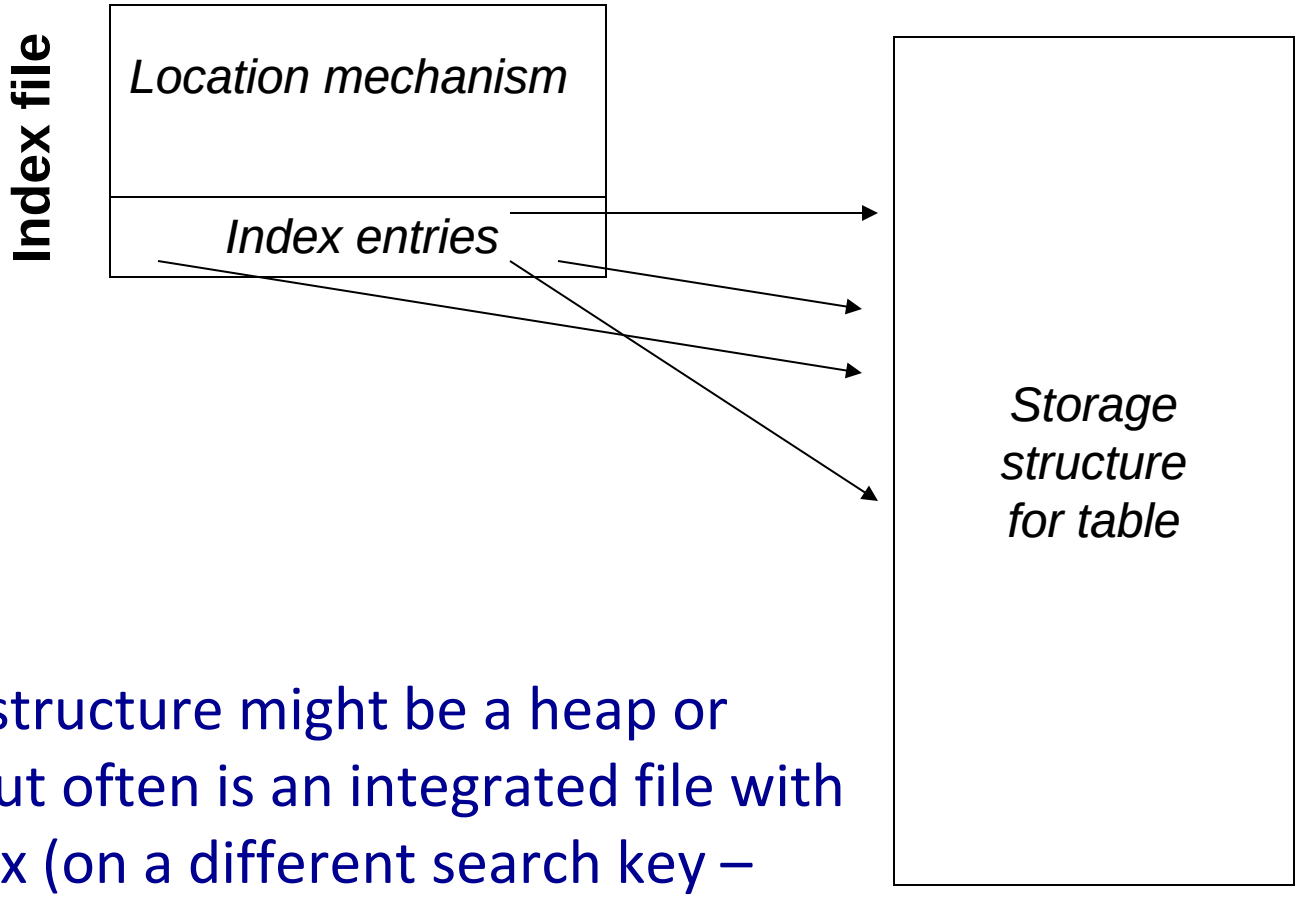
- Structure of file containing a table
  - Heap file (no index, not integrated)
  - Sorted file (no index, not integrated)
  - Integrated file containing index and rows (index entries contain rows in this case)
    - ISAM
    - B+ tree
    - Hash

Integrated Storage Structure

- *Contains table and (main) index*



Index File With Separate Storage Structure



- 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.

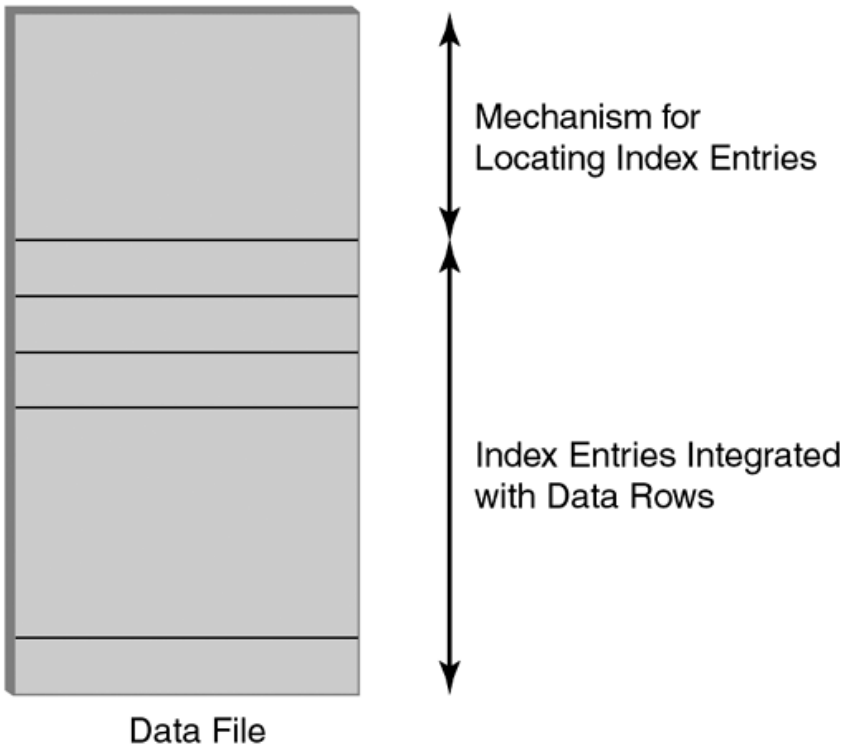
- SQL-92 does not provide for creation or deletion of indices
  - Index on primary key generally created automatically
  - Vendor specific statements:
    - CREATE INDEX ind ON Transcript (*CrsCode*)
    - DROP INDEX ind

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 (e.g., 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

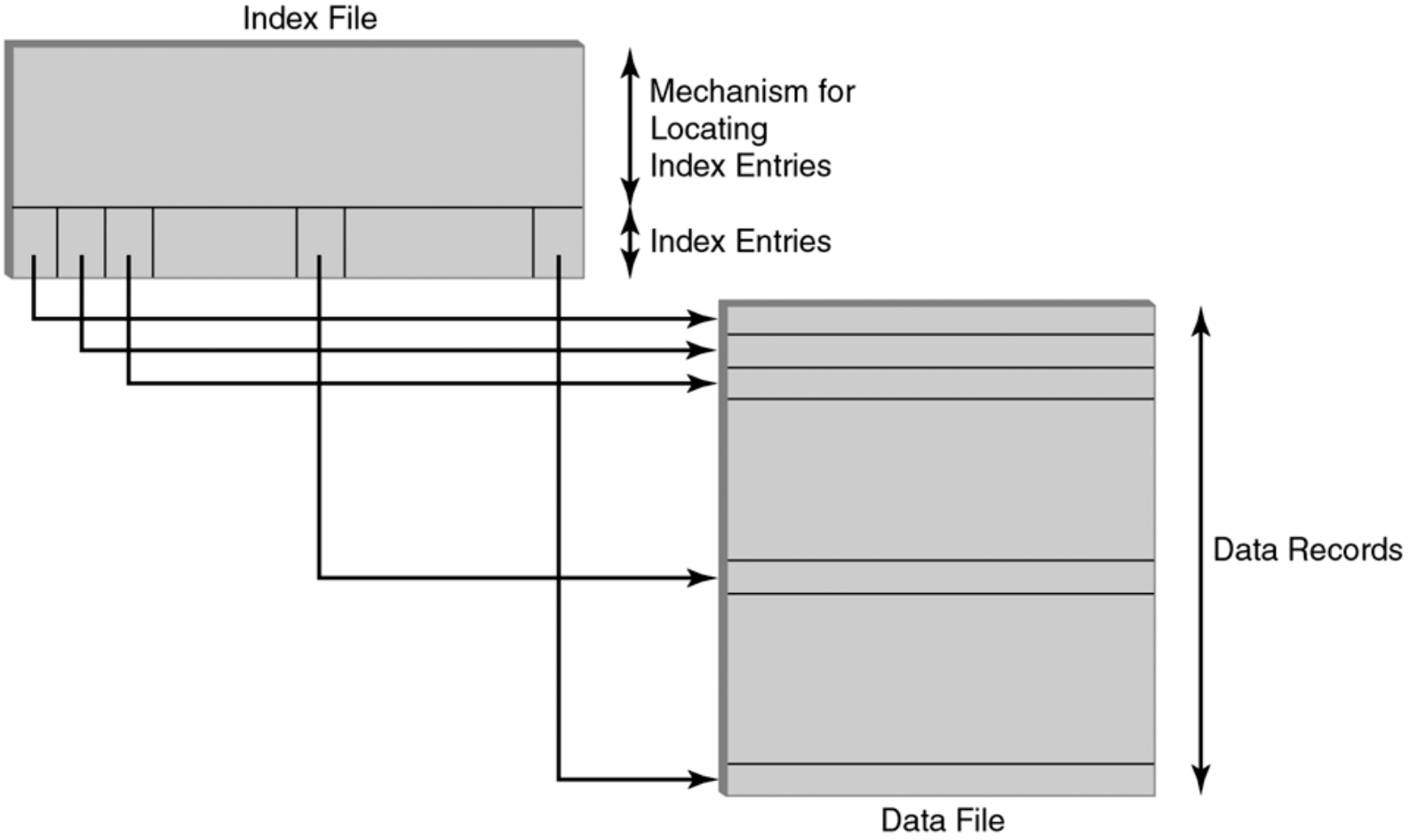


FIGURE 9.8 A clustered index that references a separate data file.

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

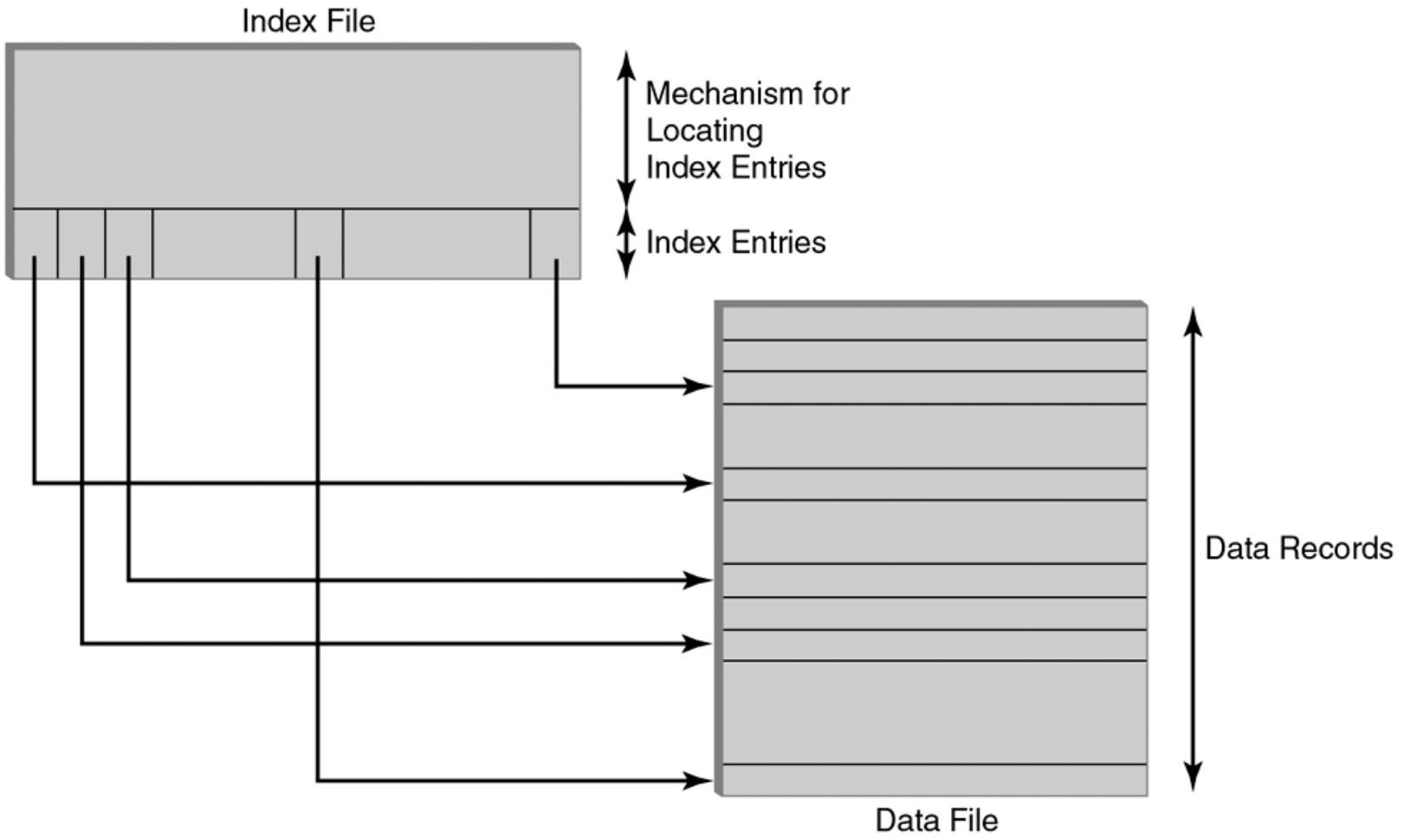


FIGURE 9.9 An unclustered index over a data file.

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 – Range Search Cost Comparison

- Data file has 10,000 pages, 100 rows in search query 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)

Stony Brook University

CSE 305 / CSE532

Lecture 18 (Chapter 9)

Physical Data Organization and Indexing

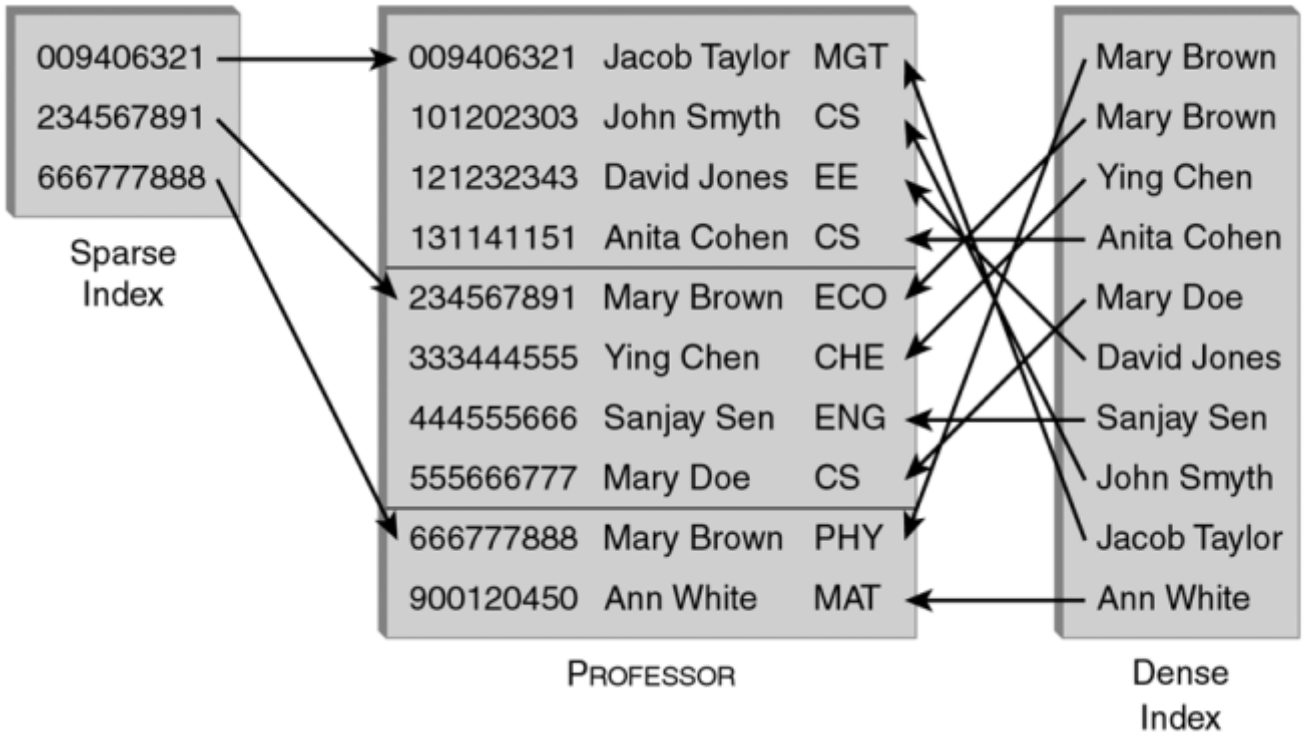
Lecturer: Sael Lee

Slide adapted from the author's and Dr. Ilchul Yoon's slides.

SUNY Korea

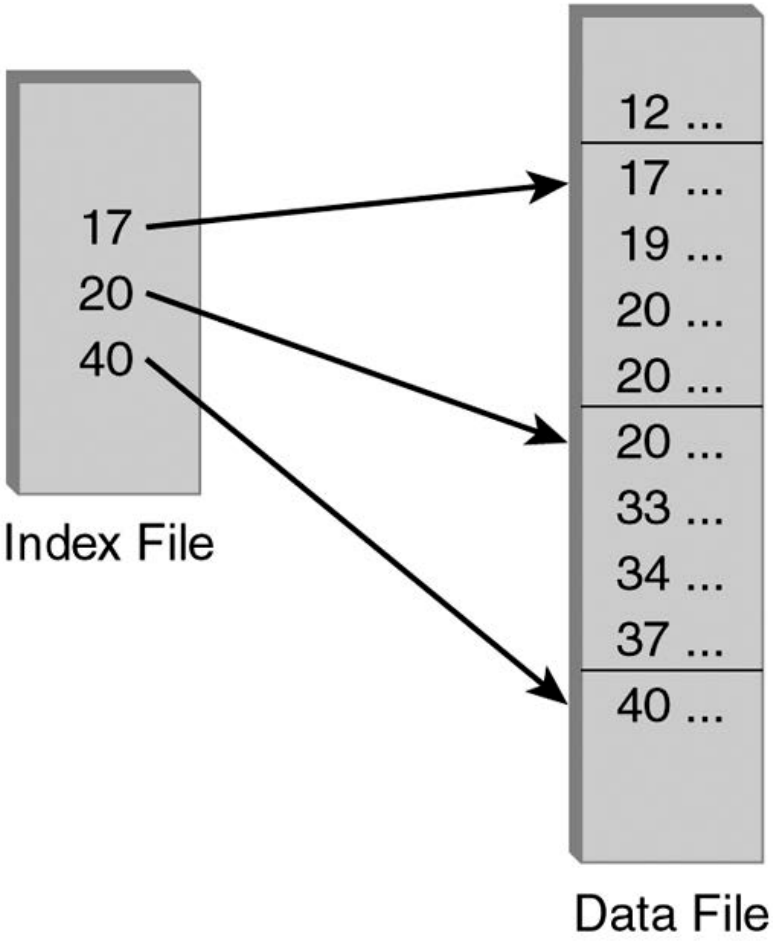
Sparse vs. Dense Index

- *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 Index

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



A problem when search key is not a CK

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
  - Further reduce the cost of locating index entry

Two-Level Index

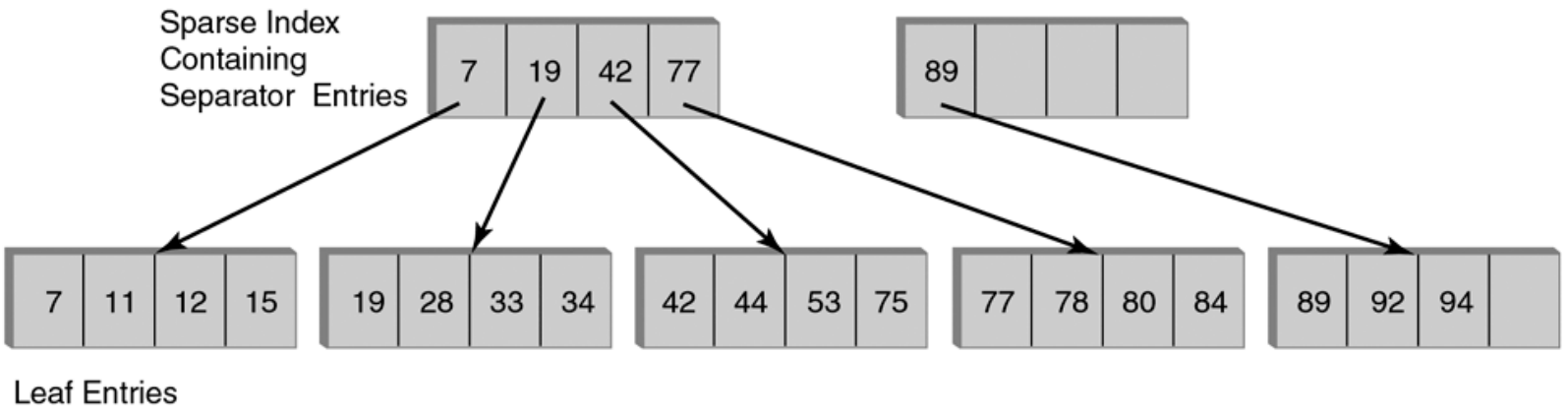


FIGURE 9.13 A two-level index. At most four entries fit in a page.

- *Separator level* is a sparse index over pages of index entries
- *Leaf level* contains index entries
- Cost of searching separator level << cost of searching index level since separator level is sparse
- Once index entry is found, cost of retrieving row is 0 (if integrated) or 1 (if not)

Multilevel Index

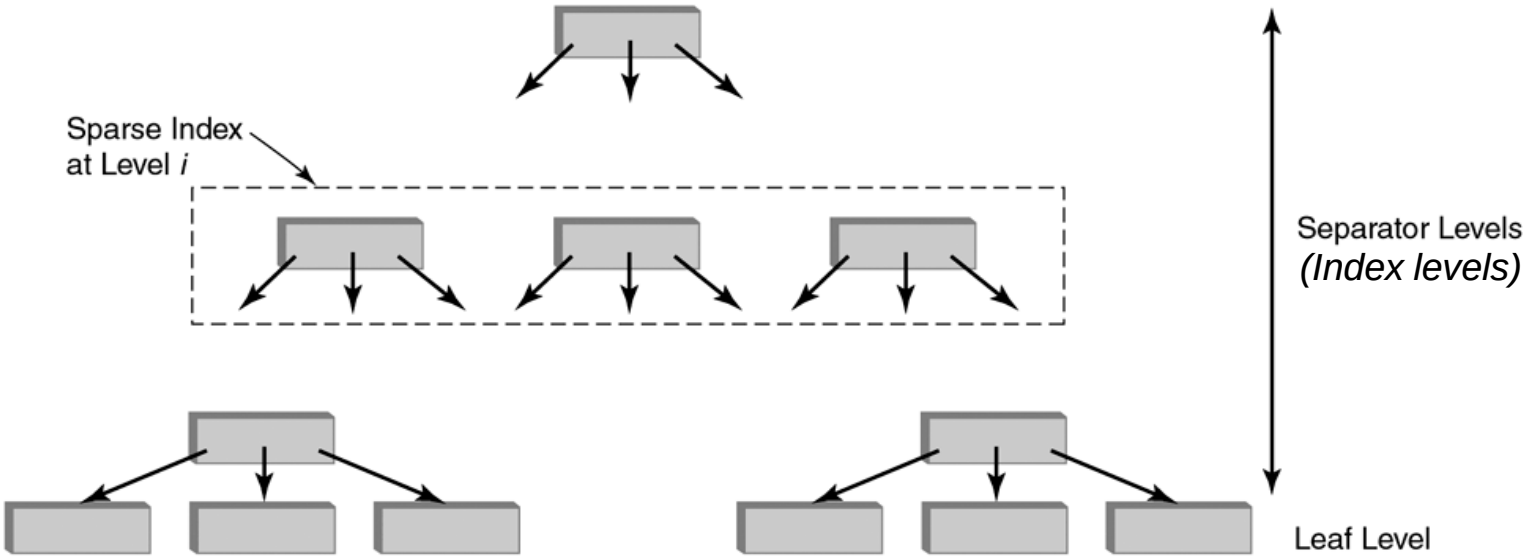


FIGURE 9.14 Schematic view of a multilevel index.

- Search cost = number of levels in tree
- If Φ is the fan-out of a separator page, cost is $log_{\Phi} Q + 1$
  - Q is the number of pages at the leaf level
- Example: if $\Phi = 100$ and $Q = 10{,}000$, cost = 3
  - Can be 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* sub-trees pointed at by p_{i-1} and p_i.

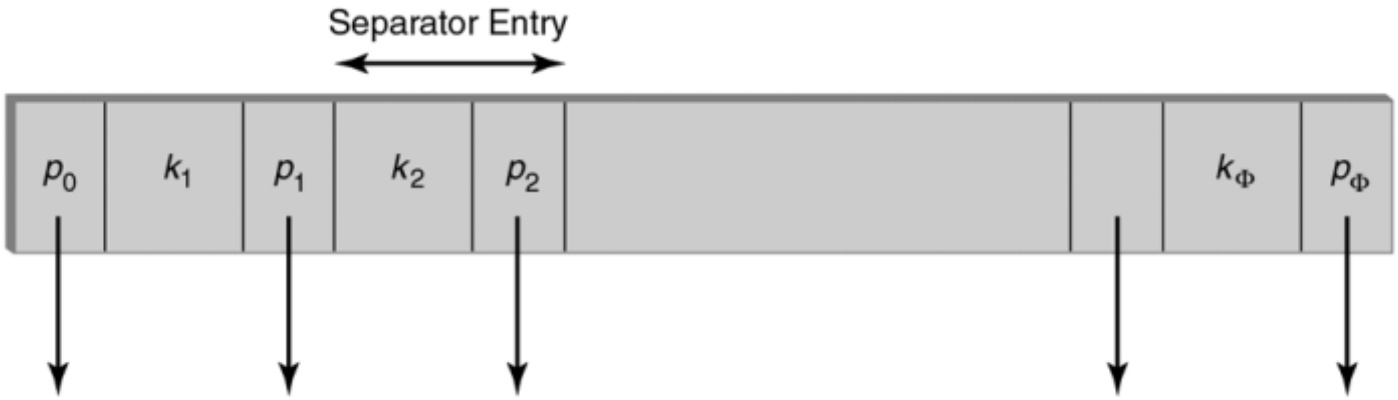


FIGURE 9.15 Page at a separator level in an ISAM index.

Index Sequential Access Method

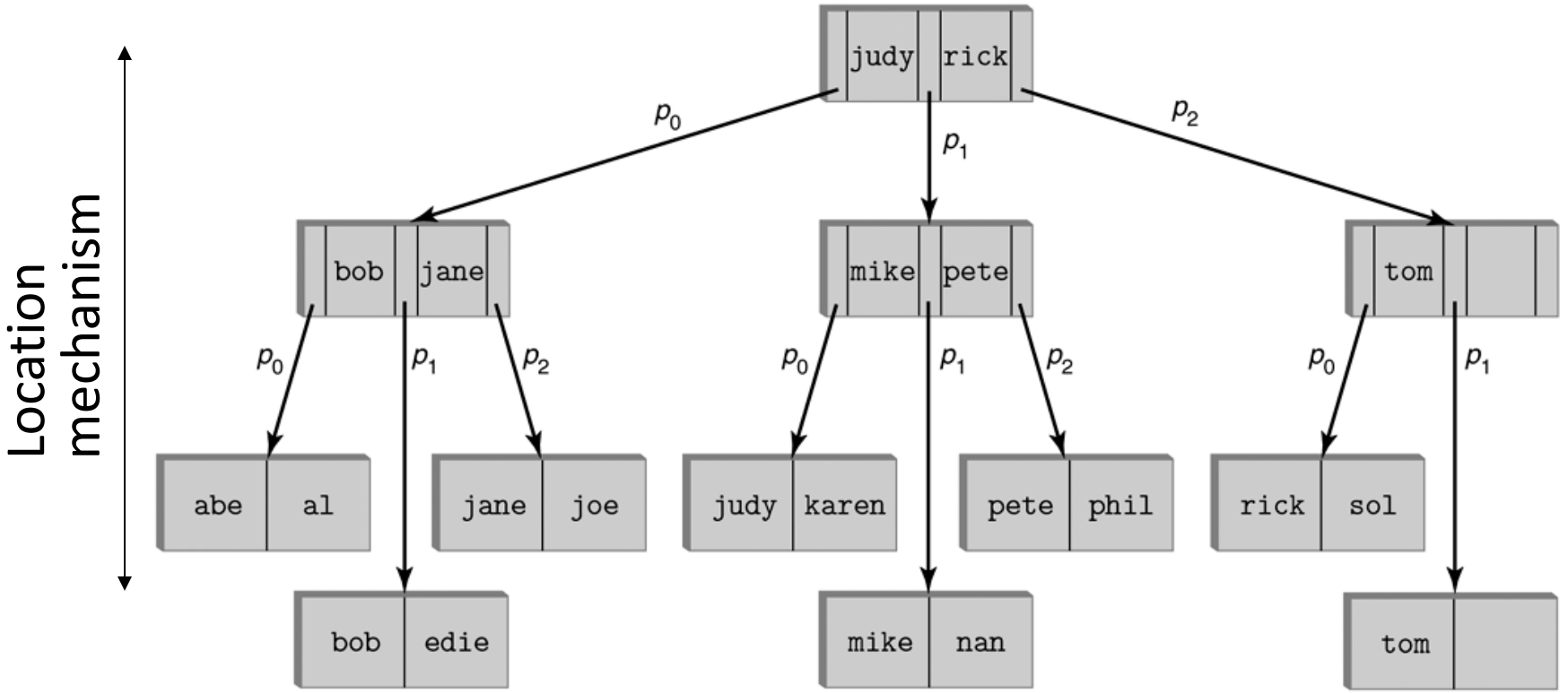


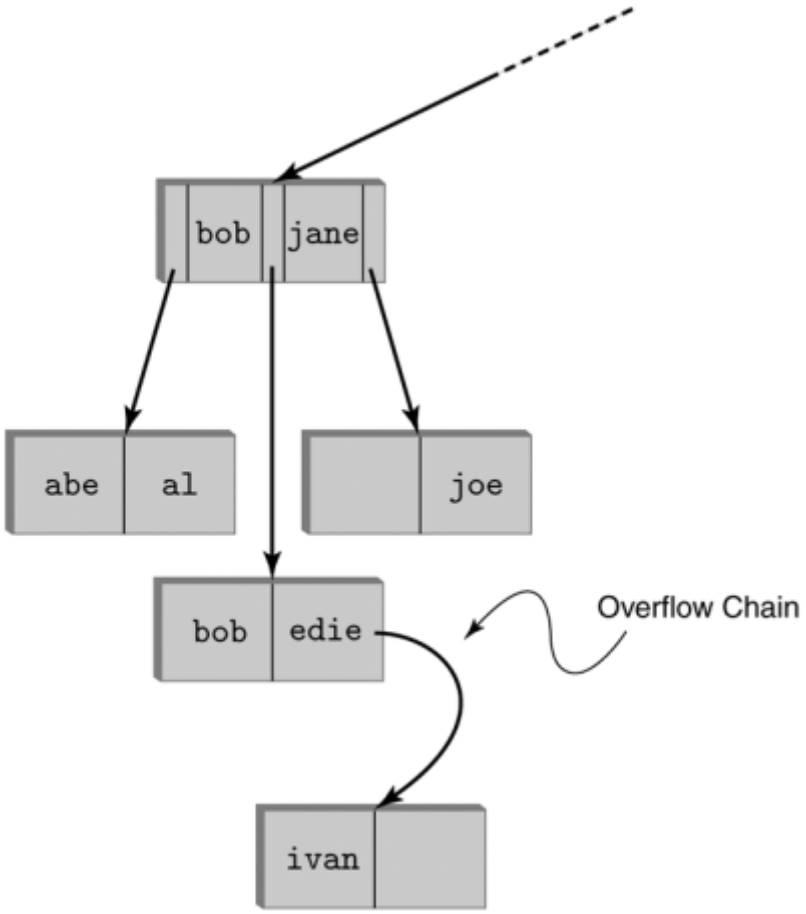
FIGURE 9.16 An example of an ISAM index.

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



After an insertion (ivan) and a deletion (jane) on the previous example

Stony Brook University

CSE 305 / CSE532

Lecture 19 (Chapter 9)

Physical Data Organization and Indexing

Lecturer: Sael Lee

Slide adapted from the author's and Dr. Ilchul Yoon's slides.

SUNY Korea

B^+ Tree

- The most widely used index structure
- A *balanced tree* in which every path from the root to a leaf is of the same length.
  - Each non-root has between $\lceil n/2 \rceil$ and n children, where n is a fixed value for a 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

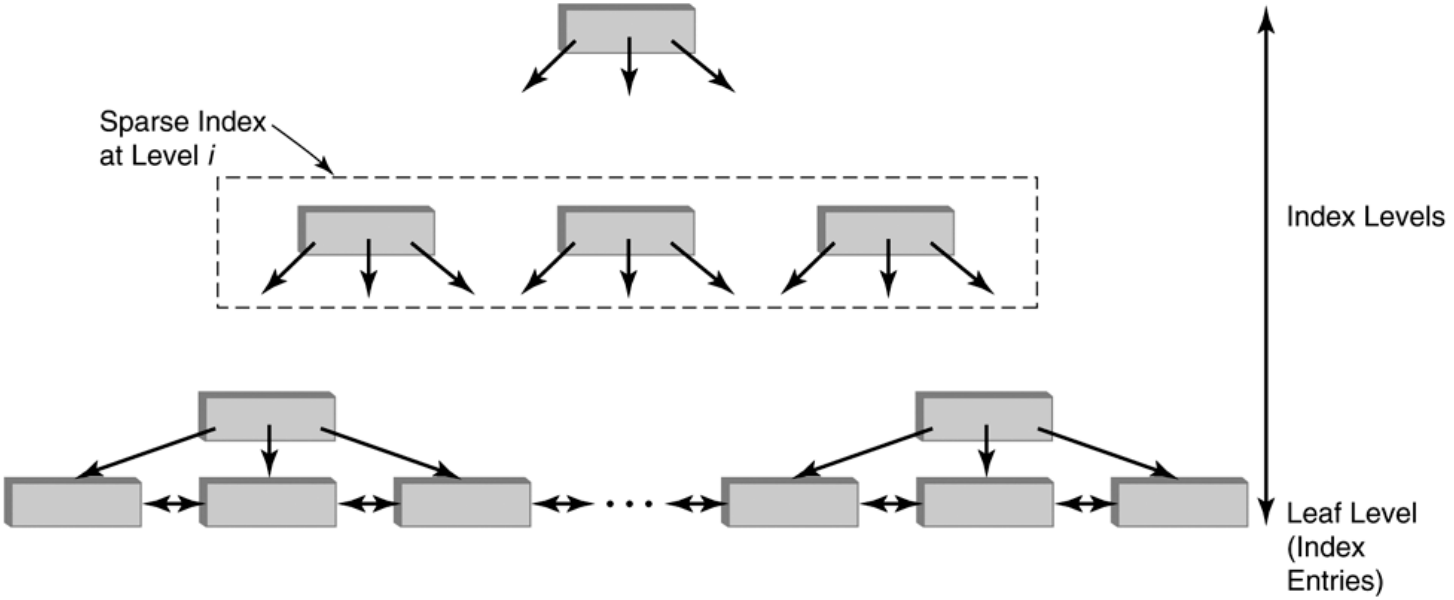


FIGURE 9.18 Schematic view of a B+ tree.

- 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

- Tree structure 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 $log_{\Phi/2}Q + 1$
  - Note - with ISAM search, the cost depends on length of overflow chain

Handling Insertions - Example

- Insert “vince”

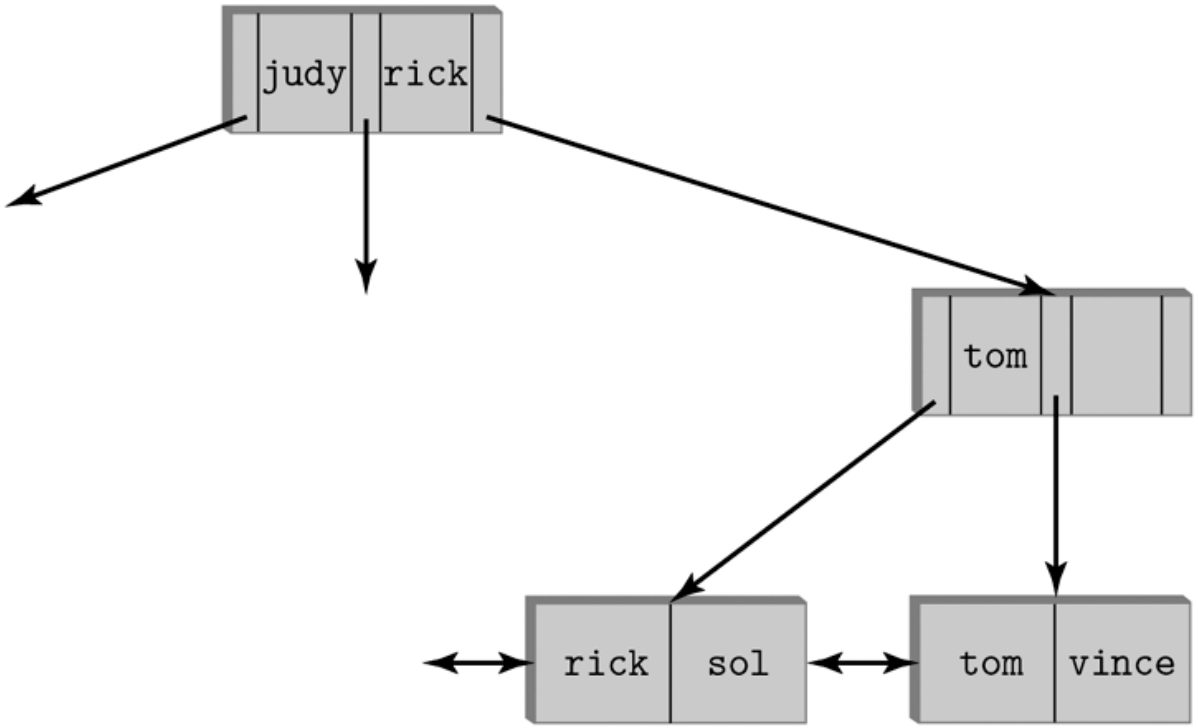


FIGURE 9.19 Portion of the index of Figure 9.16 after insertion of an entry for `vince`.

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

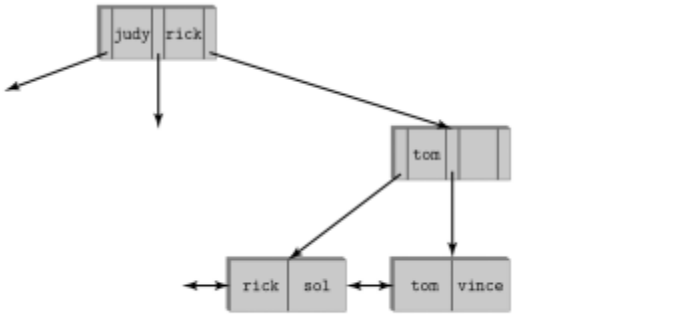


FIGURE 9.19 Portion of the index of Figure 9.16 after insertion of an entry for vince.

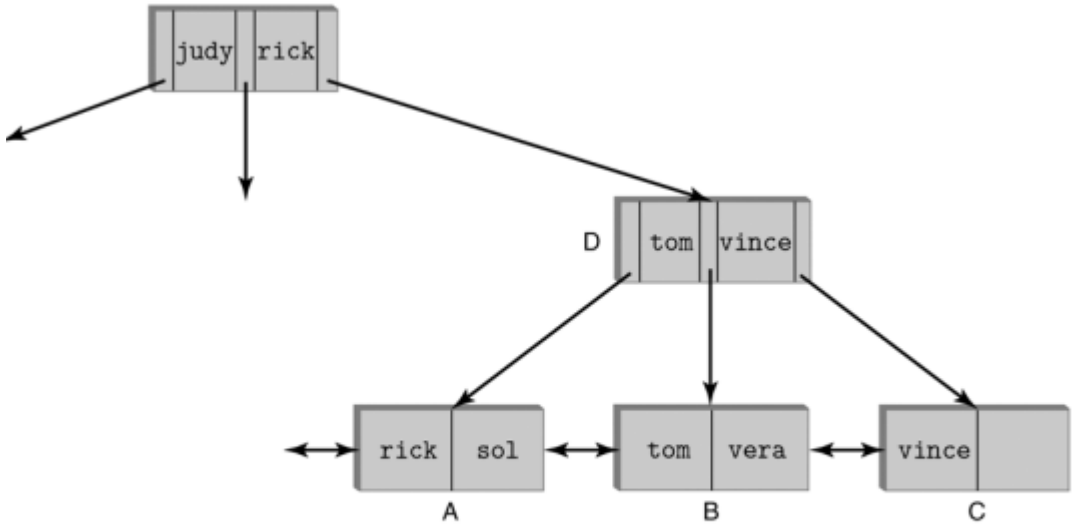


FIGURE 9.20 Index subtree of Figure 9.19 after the insertion of vera has caused the split of a leaf page.

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"

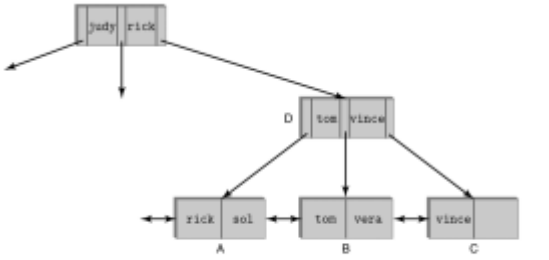


FIGURE 9.20 Index subtree of Figure 9.19 after the insertion of vera has caused the split of a leaf page.

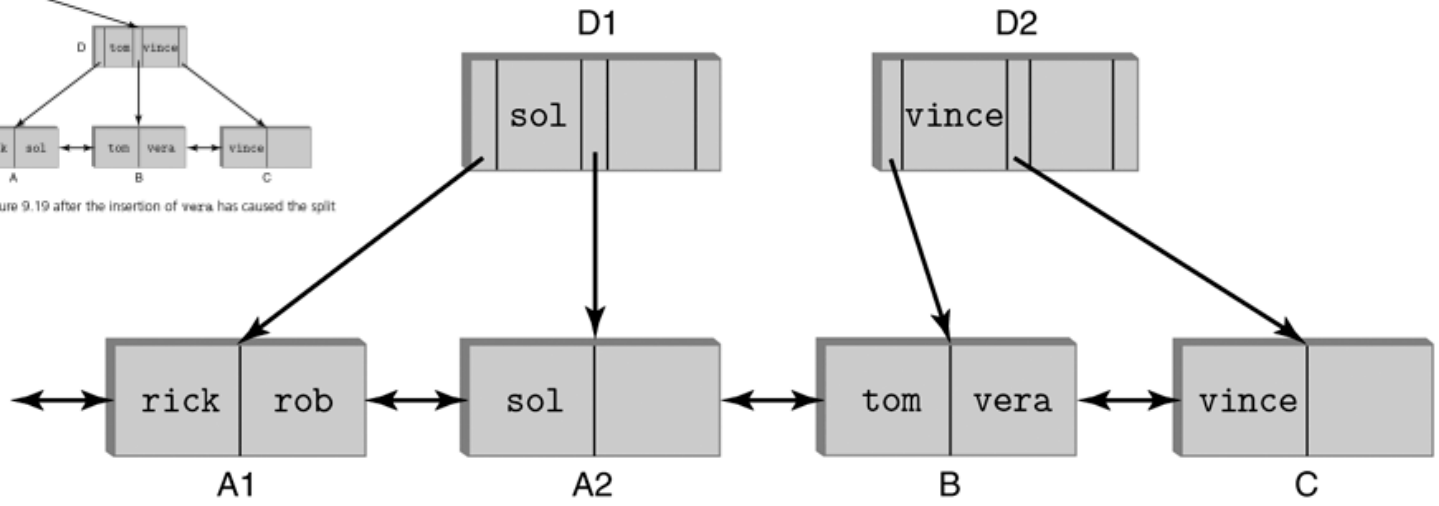


FIGURE 9.21 Index subtree of Figure 9.20 after the insertion of `rob` has caused the split of a leaf page and of a separator page.

Handling Insertions (cont'd)

- When splitting a separator page, push a separator up
  - Repeat process at next level
  - *Might increase the height of tree by one*

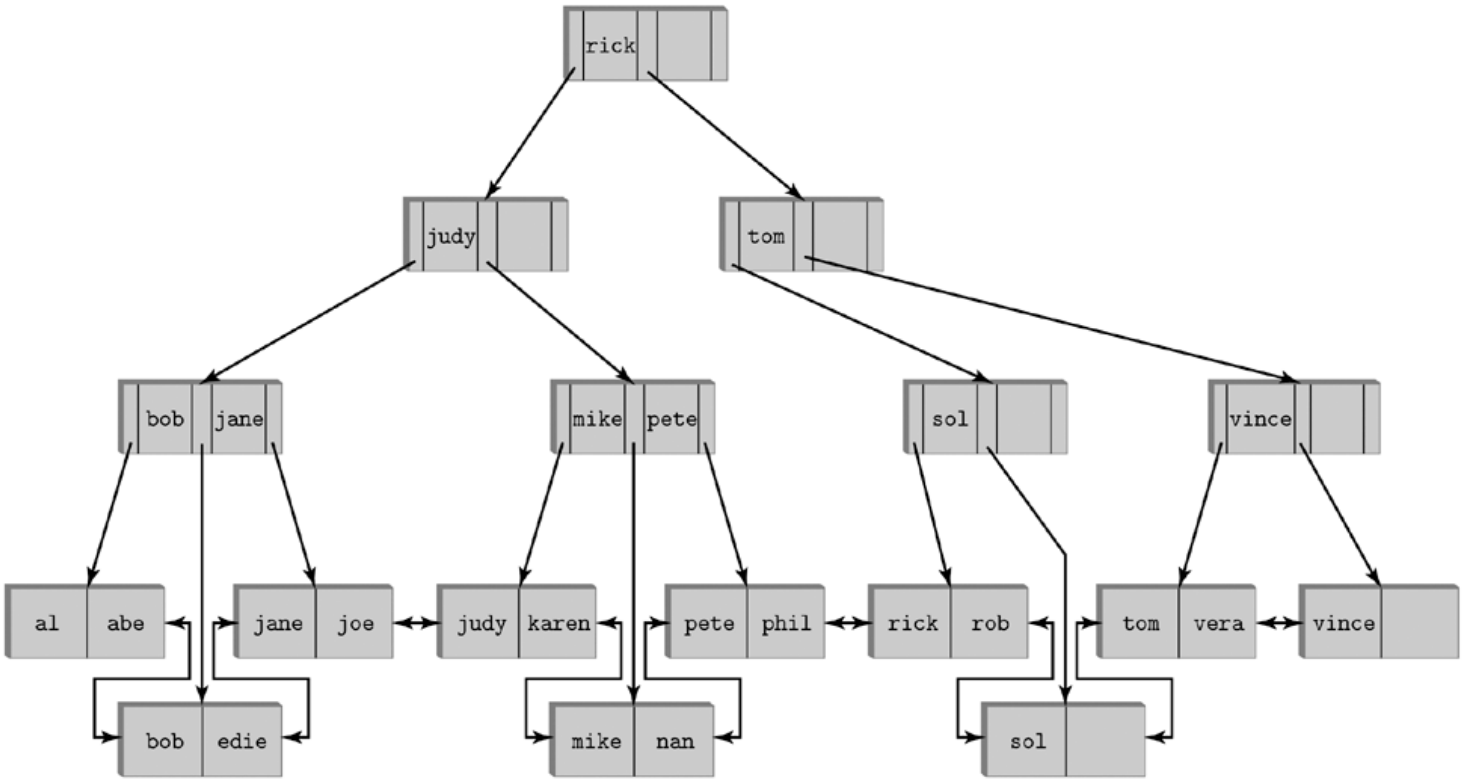


FIGURE 9.22 B^+ tree that results from the insertion of `vince`, `vera`, and `rob` into the index of Figure 9.16.

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*

Pseudocode for Insertion

FIGURE 9.23 A pseudocode rendering of the B^+ tree insertion algorithm. An entry in the index has the form $< k,\ P >$ where k is a search-key value and P is a pointer, and the tree nodes have the form $(P_0, < k_1, P_1 >, < k_2, P_2 >, \ldots, < k_{n-1}, P_{n-1} >, < k_n, P_n >)$. In the figure, `*ptr` denotes dereferencing (i.e., the actual tree nodes pointed to by the pointer `ptr`) and `&node` denotes the address of `node`.

```
proc insert(subtree, new, pushup)
// Insert entry *new into subtree with root page *subtree (new and subtree are pointers to nodes).
The maximum number of separators in a page is Φ (assumed to be even). pushup is null initially
and upon return unless the node pointed to by subtree is split. In the latter case it contains a pointer
to the entry that must be pushed up the tree. If the number of levels in the tree increases, *subtree
is the new root page when the outer level of recursion returns.

if *subtree is a non-leaf node
        (let's denote it N = (P_0, < k_1, P_1 >, ..., < k_n, P_n >)) then
  let n be the number of separators in N
  let i be such that k_i ≤ (search-key value of *new) < k_{i+1}
      or i = 0 if (search-key value of *new) < k_1
      or i = n if k_n ≤ (search-key value of *new);
  insert(P_i, new, pushup);
  if pushup is null return;
  else      // then *pushup has the form < key ,ptr >
    if N has fewer that Φ entries then // recall: N = *subtree
      insert *pushup in N in sorted order;
      pushup := null,
      return;
    else      // N has Φ entries
      add *pushup to a list of the entries in N in sorted order
      split N: first Φ/2 + 1 entries stay in N,
        last Φ/2 entries are placed in new page, N';
      pushup := &(<smallest key value in N', &(N')>);
      if N was the root of the entire B+ tree then
        create a new root-page N'' containing <&(N), *pushup>;
        subtree := &(N'');
      return;
```

Pseudocode for Insertion (Cont'd)

```
if *subtree is a leaf page (denoted L) then
  if L has fewer than Φ entries then
    insert *new in L in sorted order;
    pushup := null;
    return;
  else      // L has Φ entires
    add *new to a list of the entries in L in sorted order
    split L: first (Φ/2)+1 entries stay in L;
             the remaining Φ/2 entries placed in a new page, L';
    pushup := &(<smallest key value in L', &(L')>);
    set sibling pointers in L, L', and in the leaf page following L';
    return;
endproc
```

Pseudocode for Deletion

FIGURE 9.24 A pseudocode rendering of the first part of the B^+ tree deletion algorithm. An entry in the index has the form $< k,\ P >$ where k is a search-key value and P is a pointer, and the tree nodes have the form $(P_0, < k_1, P_1 >, < k_2, P_2 >, \ldots, < k_{n-1}, P_{n-1} >, < k_n, P_n >)$. In the figure, `*ptr` denotes dereferencing of `ptr` and `&node` denotes the address of `node`.

```
proc delete(parentptr, subtree, oldkey, removedptr)
// Delete oldkey from subtree with root *subtree.
// The minimum number of separators in a page is Φ/2 (Φ is assumed to be even).
// parentptr is null initially,
// but contains a pointer to the current index page of the caller thereafter.
// removedptr is null initially and upon return, unless a child page has been
// deleted. In that case, removedptr is a pointer to that deleted child.
// On return, *subtree is the (possibly new) root of the tree.

  if *subtree is a non-leaf node
        (henceforth denoted N = (P0, < k1, P1 >, ..., < kn, Pn >)) then
    let n be the number of separators in N;
    let i be such that ki ≤ oldkey < ki+1
        or i = 0 if oldkey < k1 or i = n if kn ≤ oldkey;
    delete(subtree, Pi, oldkey, removedptr);
    if removedptr is null then return;   // no pages deleted in the process
    else   // a child page has been deleted
      remove separator containing removedptr from N;
      if N is the root of the entire B+ tree then
        if N is not empty then return;
        else    // delete root node
          discard N;
          subtree := Pi;
          return;
      // N is not the root
```

Pseudocode for Deletion (Cont'd)

FIGURE 9.24 (continued)

```
if (number of entries in N) ≥ Φ/2 then
  removedptr := null;
  return;
else      // N has fewer than Φ/2 entries
  use parentptr to locate siblings of N;
  if N has a sibling, S, with more than Φ/2 entries then
    if S is a right sibling of N then
      redistributeleft(parentptr, subtree, &(S)); // recall that here *subtree = N
      removedptr := null;
      return;
    else    // S is a left sibling of N
      redistributeright(parentptr, &(S), subtree); // recall that here *subtree = N
      removedptr := null;
      return;
  else      // merge N and a sibling S
    choose a sibling, S;
    let M1 be the leftmost of the nodes N and S, and M2 the rightmost;
    removedptr := &(M2);
    move all entries from M2 to M1;
    discard M2;
    return;
```

Pseudocode for Deletion (Cont'd)

FIGURE 9.25 A continuation of the pseudocode from Figure 9.24 for the procedure `delete()`.

```
if *subtree is a leaf node (denoted L) then
  if oldkey is not in L then return;
  if (number of entries in L) > Φ/2 then
    delete entry containing oldkey from L;
    removedptr := null;
    return;
  else    // L has Φ/2 entries
    delete entry containing oldkey from L;
    use sibling pointers to locate siblings of L;
    if L has a sibling, S, with more than Φ/2 entries then
        if S is a right sibling of L then
          redistributeleft(parentptr, subtree, &(S)); // here *subtree = L
          removedptr := null;
          return;
        else   // S is a left sibling of L
          redistributeright(parentptr, &(S), subtree); // here *subtree = L
          removedptr := null;
          return;
    else      // merge L and a sibling S
      choose a sibling, S;
      let M1 be the leftmost of the nodes L and S, and M2 the rightmost;
      removedptr := &(M2);
      move all entries from M2 to M1;
      discard M2;
      adjust sibling pointers;
      return;
endproc
```

Pseudocode for Deletion (Cont'd)

FIGURE 9.26 A pseudocode for the procedure `redistributeleft()` called by the procedure `delete()` in Figure 9.24. The procedure moves a key from $*rightptr$ to $*parentptr$ and from $*parentprt$ to $*leftptr$.

```
proc redistributeleft(parentptr, leftptr, rightptr)
//Let e1 be entry in *parentptr containing rightptr: e1 = < k1, rightptr >
//Let e2 be smallest entry in *rightptr: e2 = < k2, ptr >
//Let P0 be the leftmost pointer in *rightptr
   add < k1, P0 > to *leftptr;
   delete e1 from *parentptr;
   add < k2, rightptr > to *parentptr;
   delete e2 from *rightptr;
   set the leftmost pointer in *rightptr to ptr;
endproc
```

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

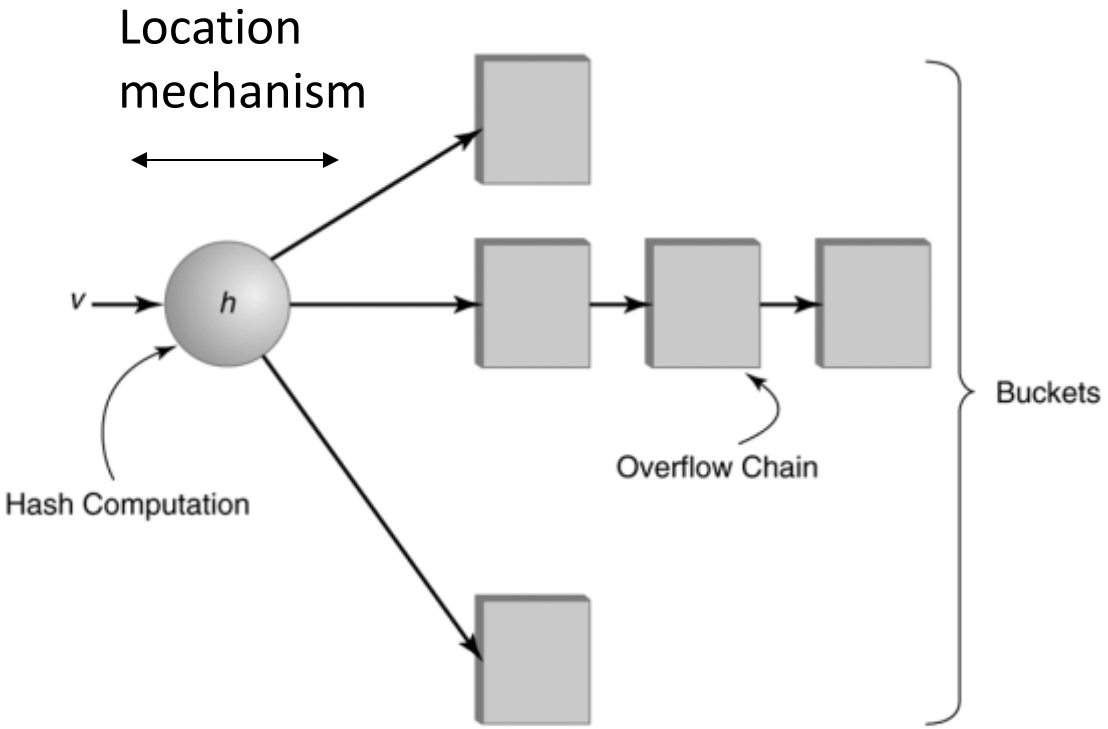


FIGURE 9.28 Schematic depiction of a hash index.

- 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

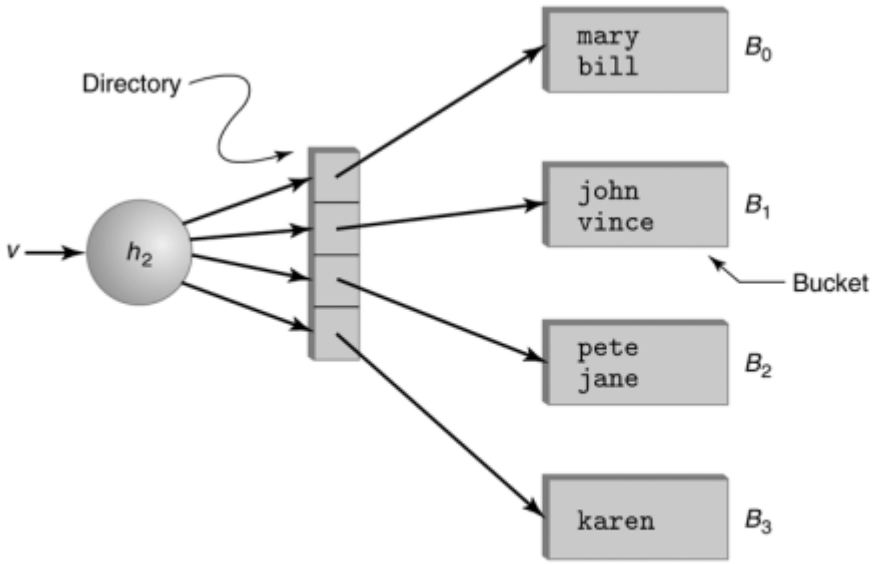
- Does not support range search
  - Since adjacent elements in range might hash to different buckets, there is no efficient way to scan buckets to locate all search key values *v* between v_1 and v_2
- Although it supports multi-attribute keys, it does not support partial key search
  - Entire value 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

- 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



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

FIGURE 9.29 With extendable hashing, the hash result is mapped to a bucket through a directory.

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′*

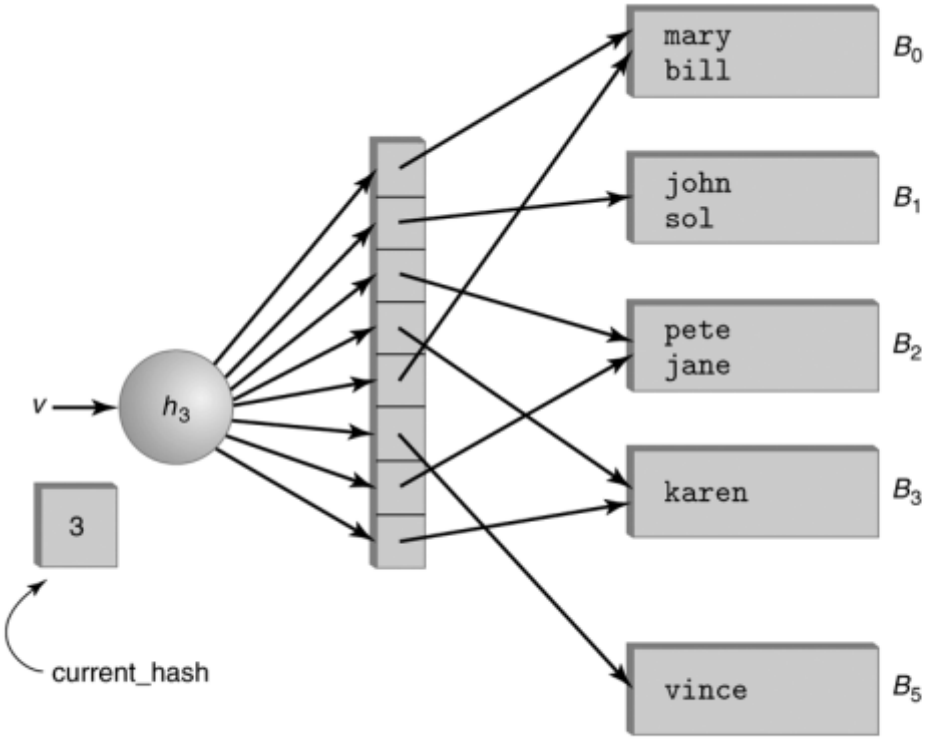


FIGURE 9.30 Bucket B_1 of Figure 9.29 has been split using extendable hashing.

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

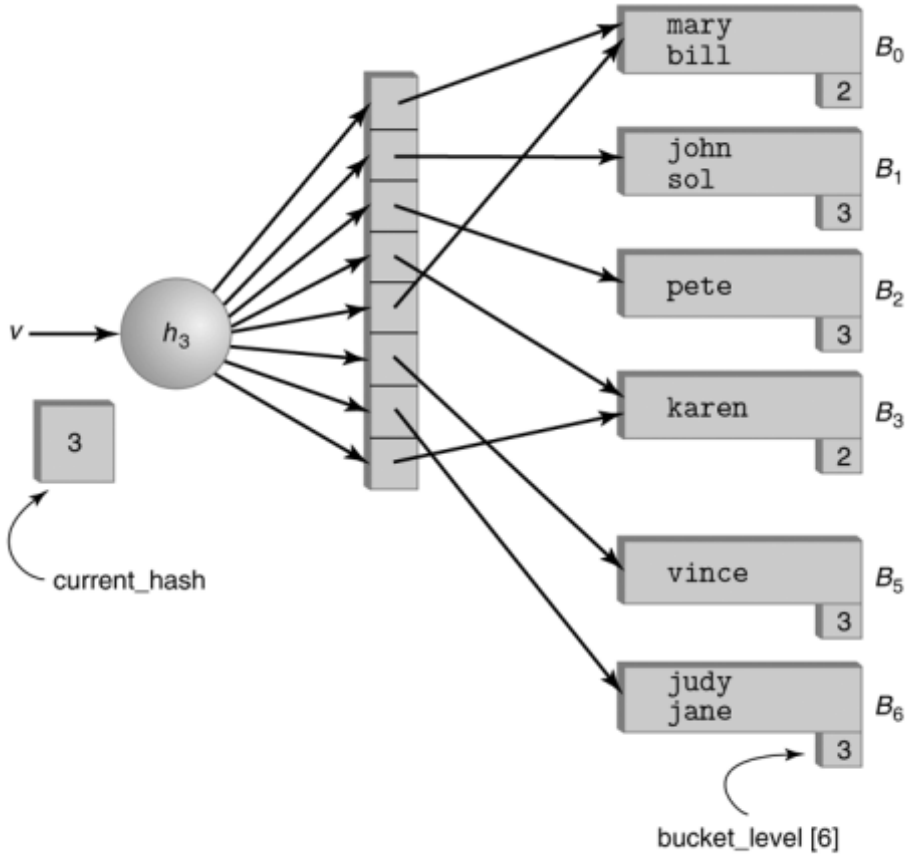


FIGURE 9.31 Bucket B_2 of Figure 9.30 is split, without enlargemen

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*)

- 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). -- the order matters here.

Indexing on Flash Memory

- All discussions so far are about data stored on magnetic disks
- Should indexing techniques be modified for flash drive?
  - Flash memory support fast block load for index lookups
    - Takes microseconds instead of milliseconds to seek and read a random block.
    - So…. B^+-tree node size can be smaller.
  - Drawback?
    - Data should be replaced logically instead of physical level
    - Every update is “copy + write” of an entire flash-memory block
      - Block erase time ~ 1 millisecond
    - So…. Issues are reducing the number of block erases.