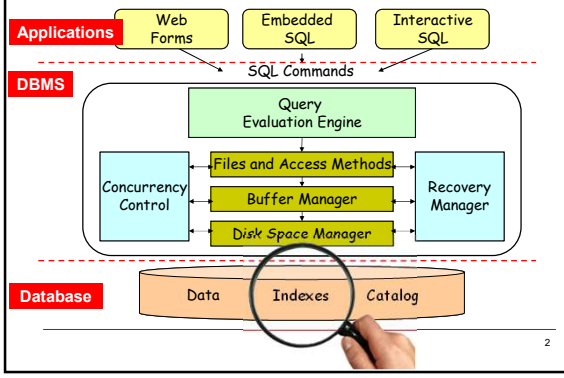
Comp2411 Database Systems

File Organization and Indexing (II)

1

Database Management System (DBMS)



2

2

Book Index

INDEX

3

3

Book Index

- Book index: lists important terms in alphabetical order with a list of page number(s) where the term appears
- Searching in a book:
  1. Search the book index for a term to find a list of addresses (i.e., page numbers)
  2. Use these addresses to retrieve the specified pages
  3. Search for the term in each page
- Otherwise, search through the entire book word by word (≈ linear search)
- A database index is similar to a book index!

4

4

Index Structure

- Single-level Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Multi-level Indexes

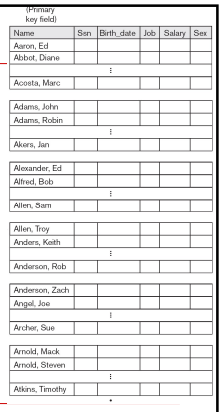
5

5

Primary Index

- Defined on an **ordered** data file
- The data file is ordered on a **key** field
- A **primary index** is an ordered file
- An entry ***"i"*** in the primary index is:

  <K(i), P(i)>
  - ***K(i):*** the key field
  - ***P(i):*** a pointer to disk block (i.e., block address)



6

6

Primary Index

Example: entry 1 = <K(1), P(1)>
<K(1) = "Aaron, Ed", P(1)= block 1>

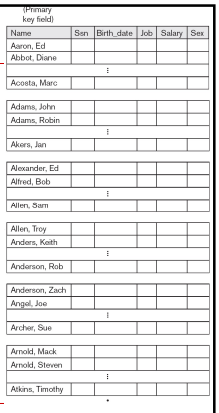
What is entry 2?!

<K(2) = "Abbot, Diane", P(2)= block 1> ✗
Or
<K(2) = "Adams, John", P(2)= block 2> ✓

<K(2) = "Abbot, Diane", P(2)= block 1>

is **redundant**: since the file is sorted then "Abbot, Diane" has to be in between "Aaron, Ed" and "Adams, John" (i.e., block 1)

Block anchor next slide…



7

7

Primary Index

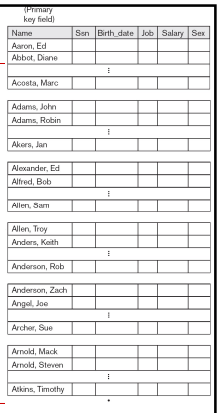
Index File
<K(i), P(i)> entries

Block anchor primary key value K(i)

Block Pointer P(i)

Entry

Aaron, Ed
Adams, John
Alexander, Ed
Allen, Troy
Anderson, Zach
Arnold, Mack



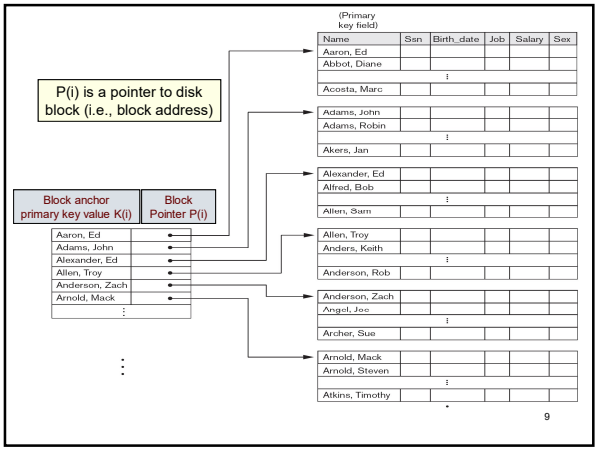
8

8

P(i) is a pointer to disk block (i.e., block address)

Block anchor primary key value K(i)

Block Pointer P(i)



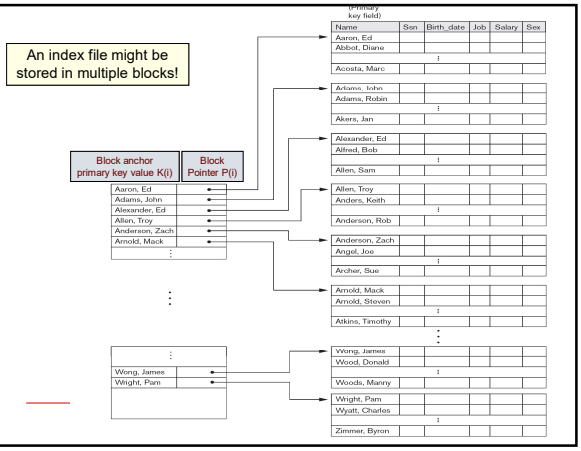
9

9

An index file might be stored in multiple blocks!

Block anchor primary key value K(i)

Block Pointer P(i)



10

10

Primary Index

- One index entry for each block in the data file
  - The index entry has the key field value for the first record in the block (***block anchor***)
- A primary index is **non-dense (sparse)**
  - It includes an entry for each disk block of the data NOT for each search value
- A primary index typically occupies multiple blocks
- But does a primary index make queries run faster?!

11

11

Advantages of Primary Index

- A primary index might be stored in multiple blocks, but: it occupies much smaller space than a data file, because:
  1. There are fewer index entries than records
  2. Each index entry is typically smaller than a data record
     - Index record: only 2 fields
- **More index entries than data records fit in a block**
- Binary search is more efficient on index file!
  - Let size of data file = B_{data} blocks
  - Let size of index file = B_{index} blocks
  - Typically: $B_{index} << B_{data}$ (much smaller)
  - Then: $Log_2 B_{index} < log_2 B_{data}$

12

12

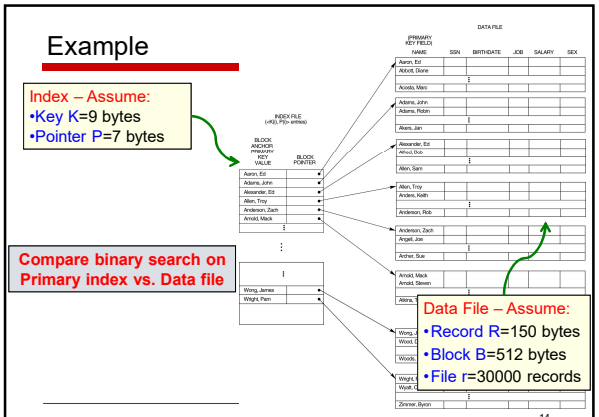
Index File vs. Data File!

13

13

Example

Index – Assume:
•Key K=9 bytes
•Pointer P=7 bytes



Compare binary search on Primary index vs. Data file

Data File – Assume:
•Record R=150 bytes
•Block B=512 bytes
•File r=30000 records

14

14

Example

Index – Assume:
•Key K=9 bytes
•Pointer P=7 bytes

index entry size E = 16 bytes
index bfr = ⌊B/E⌋ = ⌊512/16⌋ = 32 entries/block
#index blocks b=⌈**10000**/32⌉ = 313 blocks
Binary search: ⌈$\boldsymbol{log_2}$ 313⌉ = **9 block accesses**

data record size R = 150 bytes
blocking factor bfr=⌊B/R⌋=⌊512/150⌋=3 records/block
data blocks b=⌈r/bfr⌉=⌈30000/3⌉ = 10000 blocks
Binary search: ⌈$\boldsymbol{log_2}$10000⌉= **14 block accesses**

Data File – Assume:
•Record R=150 bytes
•Block B=512 bytes
•File r=30000 records

15

15

"Data" Access

- A binary search on the index yields a **pointer** to the file record
  - Extra I/O access is needed to fetch data
- To search for a record:
  - Binary search in the index file
  - Fetch (i.e., read) the block from the data file
- In the previous example, number of blocks accessed:
  - 9 accesses: binary search in index file, **plus**
  - 1 extra access: read from data file
  - Total: 10 block accesses

16

16

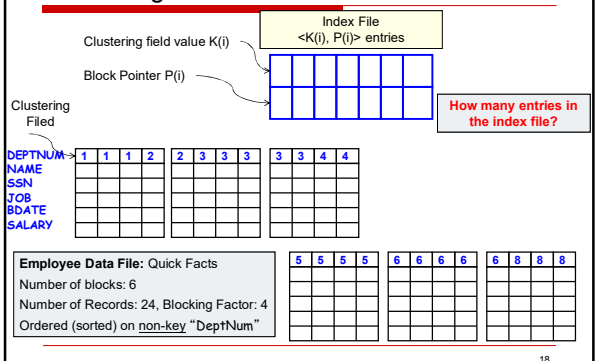
Clustering Index

- We know that a Primary Index is:
  - Defined on an **ordered** data file, and
  - That data file is ordered on a **key** field
- What if we have an:
  - **Ordered** data file
  - But the data file is ordered on a **non-key** field
- Clustering index *next slide…*

17

17

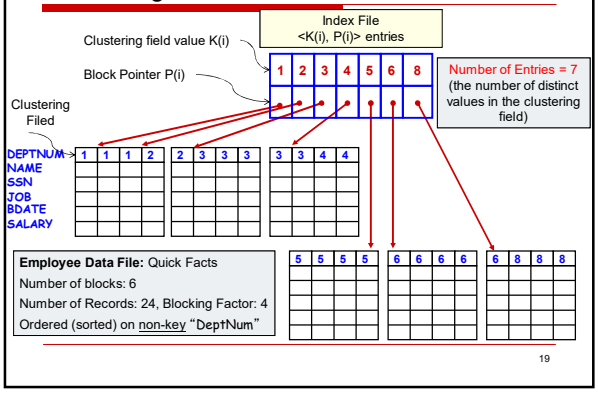
Clustering Index



18

18

Clustering Index



19

19

Clustering Index

- Defined on an **ordered** data file
- The data file is ordered on a **non-key** field
- One entry *for* each distinct value of the field
  - The index entry points to the first data block that contains records with that field value
- It is another example of **sparse** index

20

20

Clustering Index

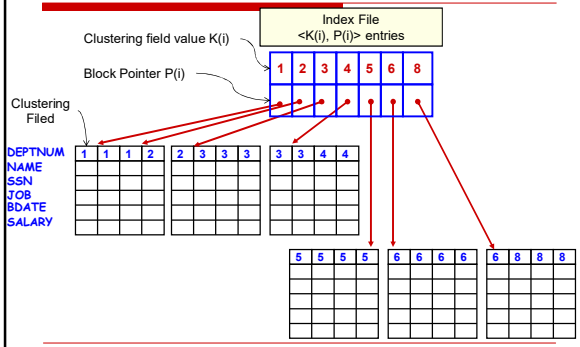
- Record insertion and deletion cause **problems**
  - Because the data records are physically ordered
  - Same problem happens in primary index!

- To overcome the problem of insertion:
  1. **Reserve** a whole block (or a set of adjacent blocks) for each distinct value of the clustering field
  2. **Place** all records that have the same value in one block (or adjacent blocks)

21

21

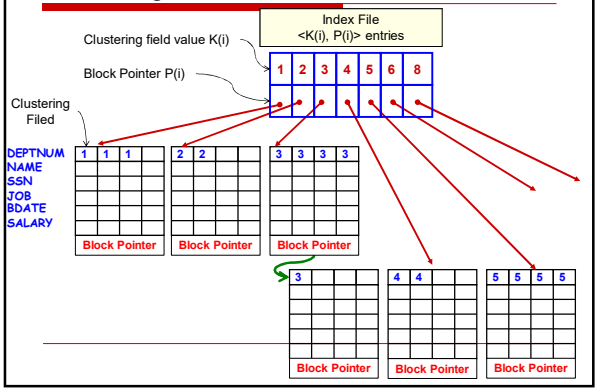
Clustering Index Without Block Reservation



22

22

Clustering Index With Block Reservation



23

23

Indexes

- So far, we know that:
  1. Primary Index is defined on:
     - An **ordered** data file, and
     - That data file is ordered on a **key** field
  2. Clustering Index is defined on:
     - An **ordered** data file, and
     - That data file is ordered on a **non-key** field
- What is missing?

24

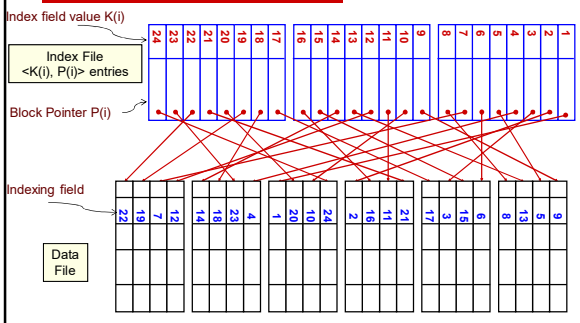
24

Secondary Index

	Ordered File	Unordered File
Key	Primary Index	**Secondary Index**
Non-Key	Clustering Index	**Secondary Index**

25

Secondary Index – on **Key** Field



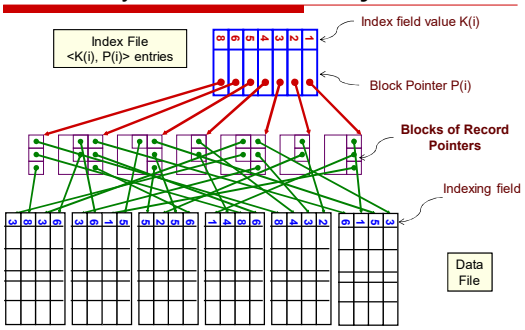
26

Secondary Index – on **Key** Field

- Includes one index entry for each record in the data file
- An index entry points to a data block that contains the record
- It is a **dense index**
  - Number of index entries = number of data records
- What is the advantage?
  - The index is still smaller than the data file
    - Each index entry has only 2 fields <K(i), P(i)>
  - Allows binary search (via the ordered secondary index)
    - Instead of linear search on the unordered data file
  - Efficient data update on the unordered data file

27

Secondary Index – on **Non-key** Field



28

Secondary Index – on **Non-key** Field

- Two-level Index (extra level of indirection)
- The 1st level (top level) contains one entry *for* each distinct value of the indexing field
  1. Each index entry (1st level) points to an index block that contains a **set of record pointers**
  2. Each record pointer (2nd level) points to a data block that contains a record with that field value
- Is it dense or sparse?

29

Summary

	Number of Entries	Dense/Sparse
Primary	?	?
Clustering	?	?
Secondary (key)	?	?
Secondary (non-key)	?	?

30

Summary

	Number of Entries	Dense/Sparse
Primary	Number of blocks in data file	Sparse
Clustering	Number of distinct values	Sparse
Secondary (key)	Number of records in data file	Dense
Secondary (non-key)	Number of distinct values (1st level) Number of records (2nd level)	Dense

31

31

Index Structure

- Single-level Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Multi-level Indexes

32

32

Why Multi-Level Indexes?

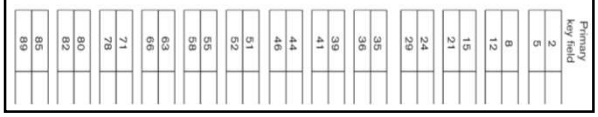
- In all the single-level indexing schemes we have seen so far, we have:
  - An **ordered** index file
    - The size of an index file depends on the index type (i.e., primary, clustering, or secondary)
  - **Binary** search is applied to the index file
    - To locate pointers P_i to disk blocks having a specific index field value K_i
  - Binary search requires $\mathbf{\log_2 B_i}$
    - In an index file of size B_i blocks, each step of binary search halves the number of remaining index blocks to search
- Can we do better than $\log_2 B_i$?!!

33

33

Example

What kind of index to build on this key field?
If index bfr = 4, how many index blocks?
Answer: see next slide..

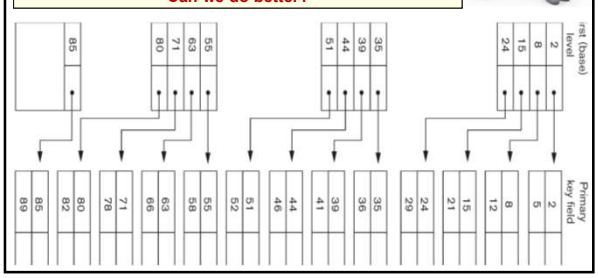


34

34

Example

What kind of index to build on this key field? **Primary index**
If index bfr = 4, how many index blocks? **4 blocks**
To find a record: **Binary search** the 4 index blocks
Can we do better?



35

35

Create an index on the index!

36

36

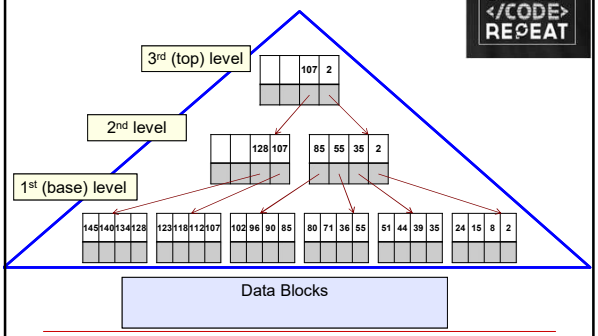
Multi-Level Indexes

- Because a single-level index is an ordered file: create a primary index to the index itself!
  - Original index file is called first-level index
  - Index to index is called the second-level index
- Repeat the process until all entries of the top level fit in one disk block
- A multi-level index can be used on any type of first-level index: primary, secondary, clustering

37

37

Multi-level Index: Example



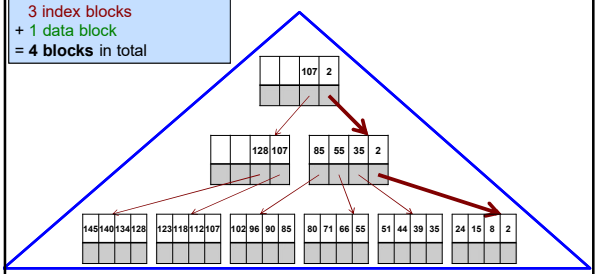
38

38

Multi-level Index: Example

Search for 24

Number of accessed blocks:
3 index blocks
+ 1 data block
= **4 blocks** in total



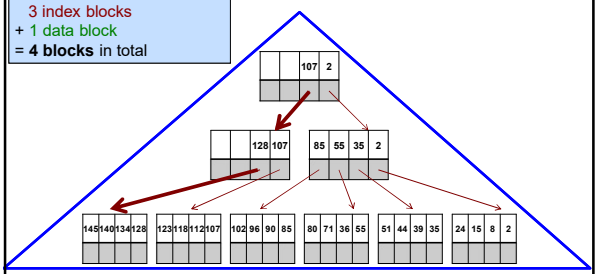
39

39

Multi-level Index: Example

Search for 140

Number of accessed blocks:
3 index blocks
+ 1 data block
= **4 blocks** in total



40

40

Single-level Index vs. Data File!

41

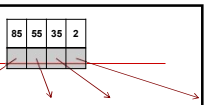
41

Multi-Level Index vs. Data File!

42

42

Search in a Multi-level Index



- How many levels do we need?
- Let:
  - Number of index entries $\mathbf{e_i}$ = 1024
  - Index blocking factor $\mathbf{bfr_{index}}$ = 4 (aka **fan-out fo**)
- Goal: only **one** block at the top level
  - 1st (base) level = **1024** entries
  - 2nd level= 1024/4 = **256** entries
  - 3rd level= 256/4= **64** entries
  - 4th level = 64/4=**16** entries
  - 5th (top) level = 16/4= **4** entries
  - 1024 >> 256 >> 64 >> 16 >> 4 = $4^5 >> 4^4 >> 4^3 >> 4^2 >> 4^1$
  - Number of levels = $\log_4 1024 = 5 = \mathbf{\log_{fo}\ e_i}$

43

43

Search in a Multi-level Index

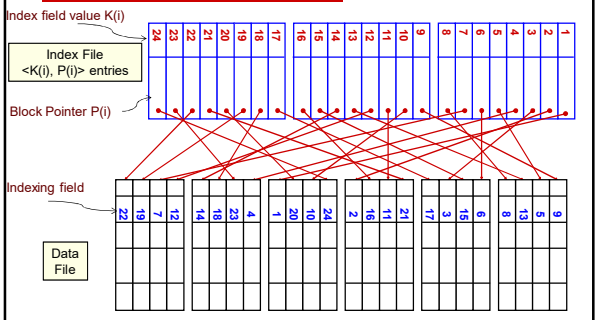
- **Searching** for a record using a multi-level index:
  1) Read one block at each level in the index
     - Number of accessed index blocks = Number of levels
       $= \mathbf{\log_{fo}\ e_i}$
  2) Read the data block containing the searched record
- **Total** number of accessed blocks = $\mathbf{\log_{fo}\ e_i} + 1$
  - Assume searching on the key field
- How much better than single-level?

See next slides (vs. a single-level secondary index)

44

44

Secondary Index – on **Key** Field



45

45

Example: Using Secondary Index

Index – Assume:
- Key K=9 bytes
- Pointer P=6 bytes

index entry size E = 15 bytes
index bfr = ⌊B/E⌋ = ⌊1024/15⌋ = 68 entries/block
index entries = # data records = **30000**
#index blocks b=⌈30000/68⌉ = 442 blocks
Binary search: ⌈$\log_2$ 442⌉ = **9 block accesses**

Block Pointer P(i)

Indexing field

Data File – Assume:
- Record R=100 bytes
- Block B=1024 bytes
- File r=30000 records

data record size R = 100 bytes
data bfr= ⌊B/R⌋ = ⌊1024/100⌋ = 10 records/block
data blocks b=⌈r/bfr⌉ =⌈30000/10⌉ =3000 blocks
linear search: 3000/**2**= **1500 block accesses**

46

46

Example: Using Multi-level Index

Multi-level Index:
- Key K=9 bytes
- Pointer P=6 bytes

Multi-level index entry size E = 15 bytes
index bfr = ⌊B/E⌋ = ⌊1024/15⌋ = 68 entries/block
index entries = # data records = 30000

of 1st level blocks b_1=⌈30000/68⌉ = 442 blocks
# of 2nd level blocks b_2 =⌈442/68⌉ = 7 blocks
# of 3rd level blocks b_3 =⌈7/68⌉ = 1 block

Number of levels = **3** (⌈$\log_{68}$ 30000⌉)

Block Pointer P(i)

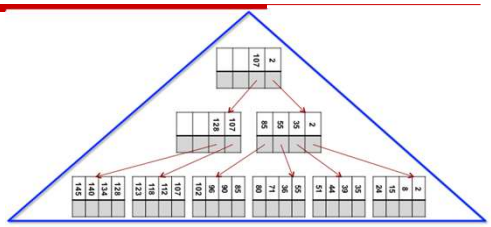
Indexing field

To search for a record:
Read **one** block at each level in the index (3 accesses) **plus**
Read **one** block that contains the searched record in data file (1 access)
Total = 3 + 1 = **4 block accesses** (vs. 9+1 for single-level secondary index)

47

47

Dynamic Data



- A multi-level index is a form of **search tree**
- Insert/delete of an index entry is a problem because:
  - Every level of the index is **ordered**
  - What happens when inserting new entry 82?

48

48

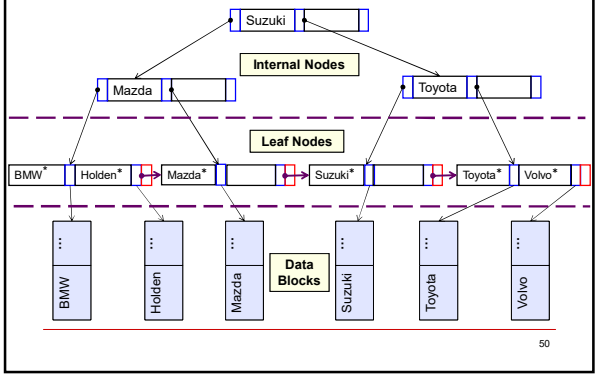
Dynamic Multi-Level Index (B+ trees)

- Because of the insertion/deletion problem, most multi-level indexes use **B+-trees:**
- In B+-Tree data structures, each tree node corresponds to a disk block
- **Leave space** in each tree node
  - Each node is kept between half-full and completely full
  - Allows efficient insertion and deletion of search values

49

49

Example B+ Tree



50

50

Internal vs. Leaf Nodes

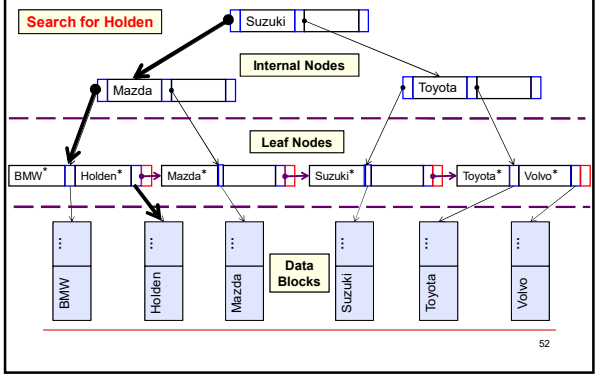
- **Internal Nodes**
  - Guide the search to leaf nodes (see next slide)
  - Pointers are tree pointers
- **Leaf Nodes**
  - Have an entry for every value of the search field
  - Pointers are data pointers
  - Data pointers are stored only at leaf nodes

The structure of leaf nodes **differs** from the structure of internal nodes

51

51

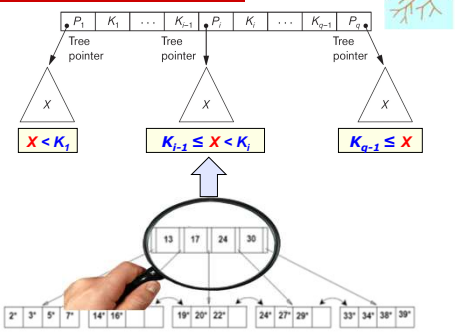
Example B+ Tree



52

52

B+ Tree Index: Internal Node

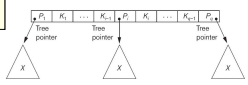


53

53

B+ Tree Index: Internal Node

An internal node is of the form:
[p_1, k_1, .. , k_{i-1}, p_i, k_i, .., k_{q-1}, p_q]
p_i is pointer and k_i is field value (key)



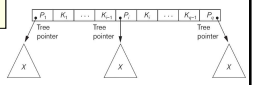
1. **Keys**
   - $k_1 < k_2 < \ldots < k_i < \ldots < k_{q-1}$
   - Some search values from the leaf nodes are repeated in the internal nodes to guide the search
   - For all search values X in the subtree pointed at by p_i, we have:
     - $X < K_i$, and $X \geq K_{i-1}$

54

54

B+ Tree Index: Internal Node

An internal node is of the form:
$[p_1, k_1, .., k_{i-1}, p_i, k_i, .., k_{q-1}, p_q]$
p_i is pointer and k_i is field value (key)

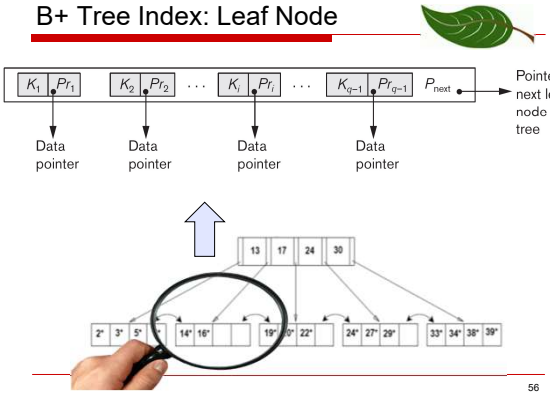


2. **Pointers:**
   - Pointers are **tree pointers** to blocks that are tree nodes
   - For a tree of **Order q** (index blocking factor / fan-out):
     - Each internal node has at most:
       - ***q*** tree pointers
     - Each internal node has at least:
       - ***q/2*** tree pointers
       - Except the root, which can have at least 2 tree pointers

55

55

B+ Tree Index: Leaf Node

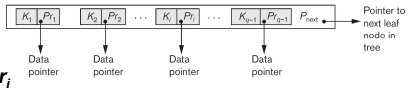


56

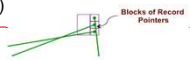
56

B+ Tree Index: Leaf Node

A leaf node is of the form:
$[<k_1, pr_1>, .. , <k_{q-1}, pr_{q-1}>, p_{next}]$



1. **Pointers pr_i**
   - Each pr_i is a data pointer to k_i's record(s):
     - If k_i is a key field (i.e., unique)
       - pr_i points to a data block that contains the record
     - If k_i is a non-key (i.e., repeated)
       - pr_i points to a block containing pointers to the data file records (as in secondary index)

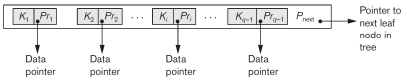


57

57

B+ Tree Index: Leaf Node

A leaf node is of the form:
$[<k_1, pr_1>, .. , <k_{q-1}, pr_{q-1}>, p_{next}]$

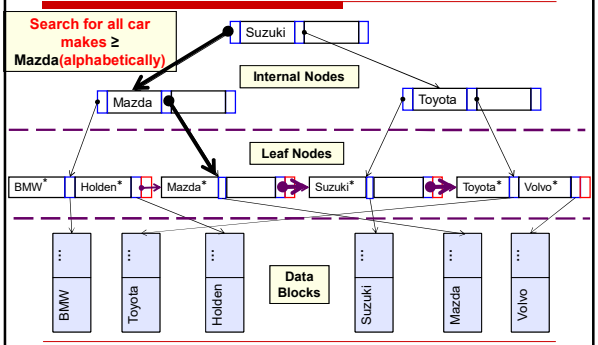


2. **Pointer p_{next}**
   - p_{next} points to the next leaf node
   - Leaf nodes are linked to provide ordered access on the indexing field
     - Traverse leaf nodes as a linked list
     - Very useful for range search
       - E.g., cars between BMW and Mazda

58

58

Example B+ Tree



59

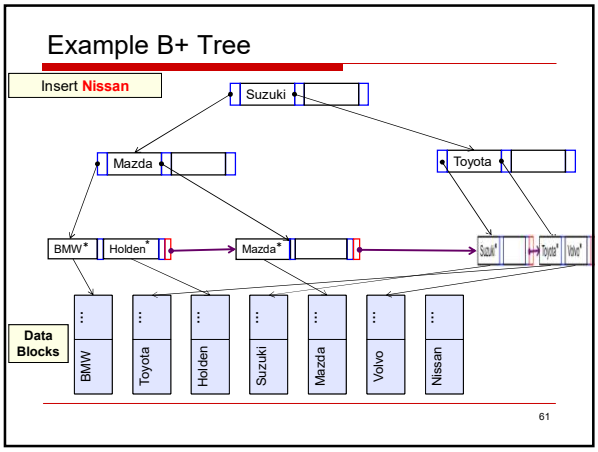
59

Dynamic Multi-Level Index (B+ trees)

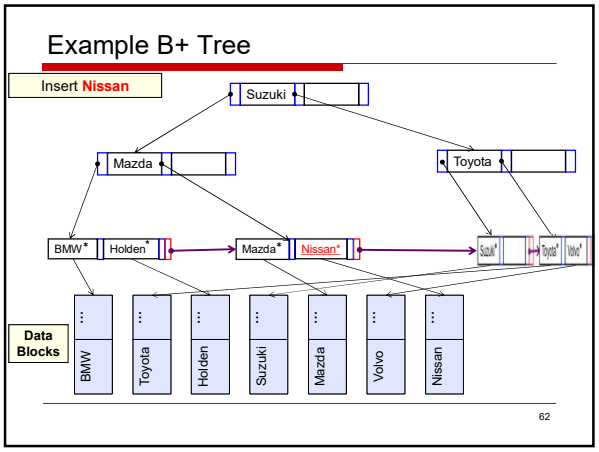
- **Insertion:**
  - Into a node that is **not full** is quite efficient
  - If a node is full it is split into two nodes
  - Splitting may propagate to other tree levels
- **Deletion:**
  - Efficient if a node is **more than half full**
  - Otherwise, it is merged with neighboring nodes
  - Merging may propagate to other tree levels

60

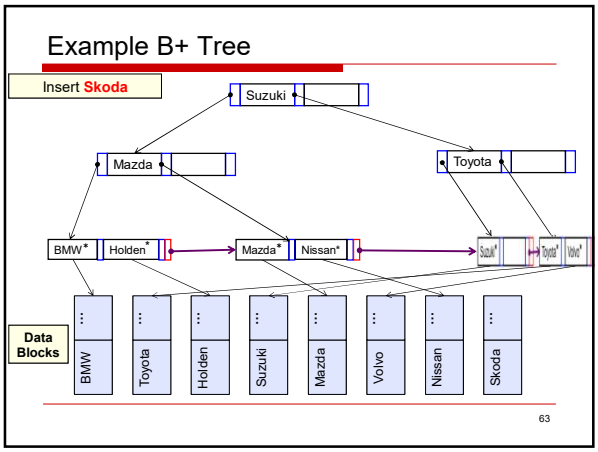
60



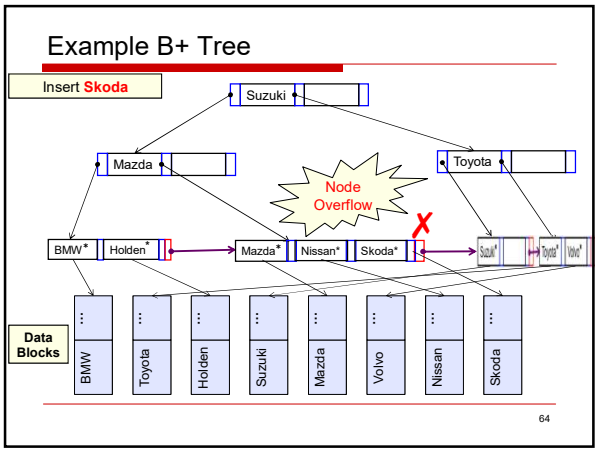
61



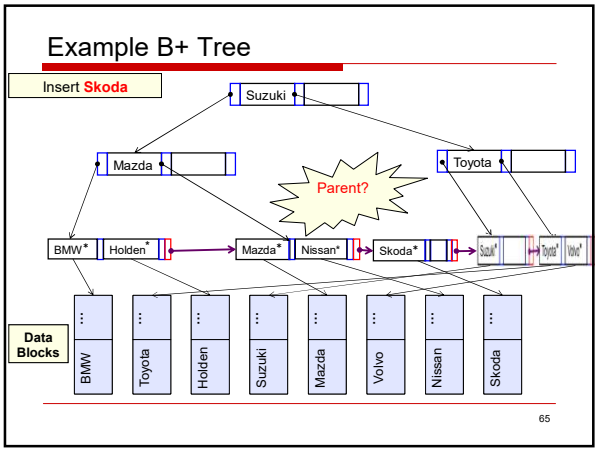
62



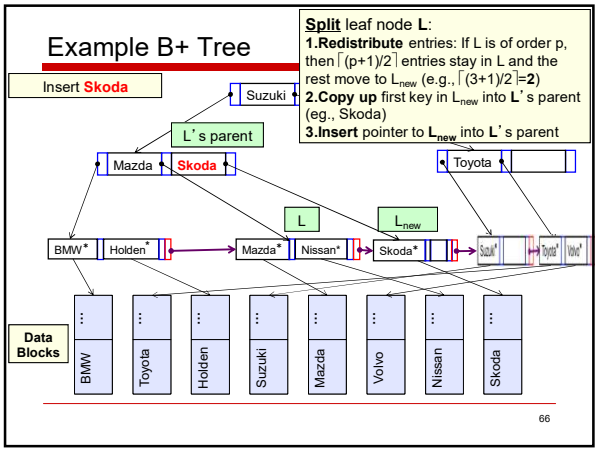
63



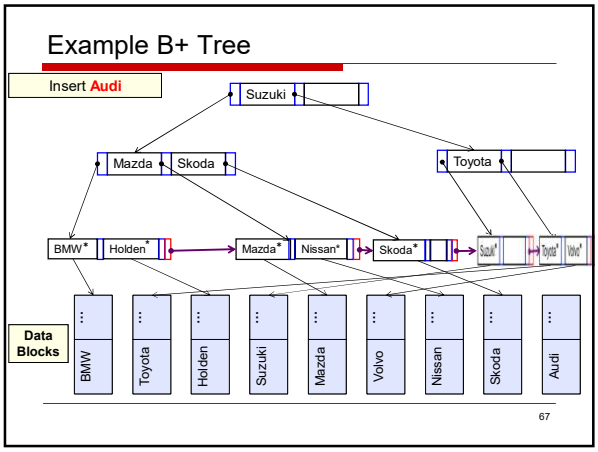
64



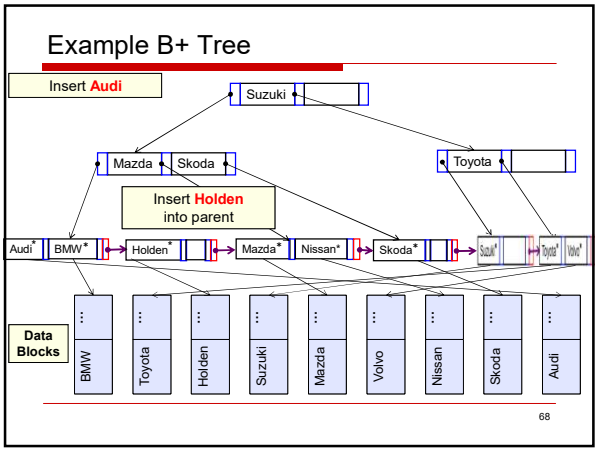
65



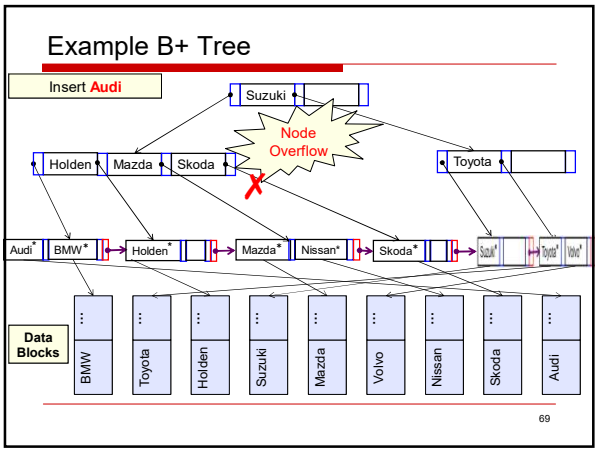
66



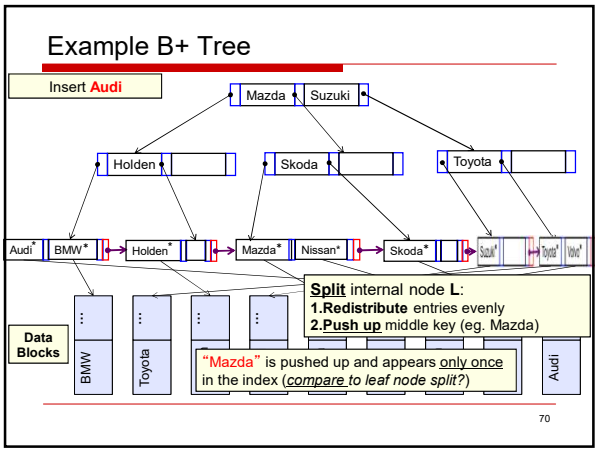
67



68



69



70

Inserting a New Entry Into a B+ Tree

- Find correct leaf **L,** add new entry into **L**
  - If **L** has enough space, done ☺
  - Else, must split **L** (into **L** and a new node L_{new})
    - **Redistribute** entries evenly
      - If L is of order p, then $\lceil (p+1)/2 \rceil$ entries stay in L and the rest move to L_{new} (or some other similar rule)
    - **Copy up** first key in L_{new} into **L**'s parent
    - **Insert** pointer to L_{new} into **L**'s parent
- This can happen recursively
  - To split internal node:
    - **Redistribute** entries evenly
    - **Push up** middle key (vs. copy up)

71

71

Dynamic Multi-Level Index (B+ trees)

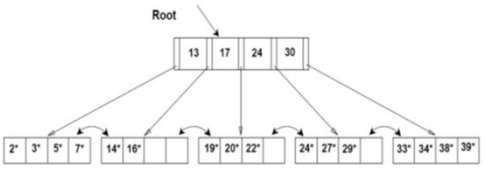
- **Insertion:**
  - Into a node that is **not full** is quite efficient
  - If a node is full it is split into two nodes
  - Splitting may propagate to other tree levels
- **Deletion:**
  - Efficient if a node is **more than half full**
  - Otherwise, it is merged with neighboring nodes
  - Merging may propagate to other tree levels

72

72

Deletion in B+ trees

- Do it yourself
  - Delete 7?
  - Delete 14?



73

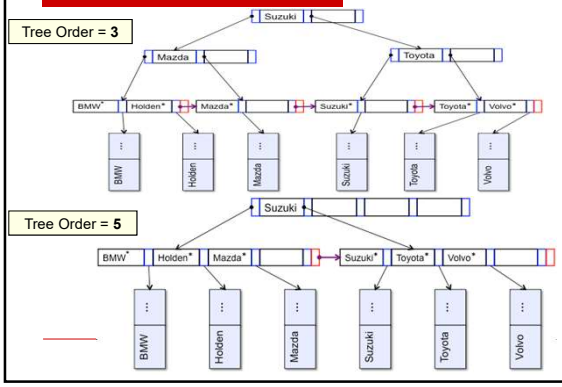
Search in a B+ Tree Index

- **Searching** for a record using B+ tree:
  1) Read one block at each level in the index
     - Number of accessed index blocks = **tree height**
  2) Read the data block containing the searched record
- **Total** number of accessed blocks = tree height + 1

Try to minimize tree height!

74

Impact of Tree Order



75

Search in a B+ Tree Index

- Tree height depends on:
  1. The number of entries in leaf nodes (e), and
  2. Tree order (i.e., fan-out (fo))
- Tree height is proportional to $\log_{fo} e$
- Fan-out = index blocking factor = block size/entry size

Try to minimize entry size!

- Index entry = <key value, pointer>
- The smaller the size of the key value, the more efficient the B+ tree (i.e., shorter)

76

Index Structure

- Single-level Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Multi-level Indexes (B+ tree)
- What if we are searching by multiple fields?
  - Bitmap Index

77

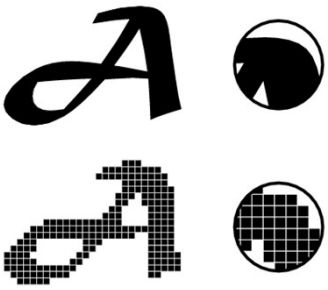
Search on Multiple Fields

```
SELECT *
FROM Students S
WHERE S.gender = "F"
AND S.dept = "COMP"
```

- To speed up this query, you might build:
  - Only one index on dept
  - Only one index on gender
  - Two separate indexes:
    1. One on dept, and
    2. One on gender
  - One index on both:
    - Dept and gender
    - Index on **multiple fields**

78

Bitmap Index



79

79

Bitmaps

Gender Index

	SID	Name	Gender
1	6007	Peter	M
2	6100	Ann	F
3	6107	Bob	M
4	6207	Jane	F
5	6240	Suzy	F
6	6350	Ben	M
7	6420	Peter	M
8	6500	Jenn	F

F	M
0	1
1	0
0	1
1	0
1	0
0	1
0	1
1	0

80

80

Bitmap Index

- **Bitmap index:** facilitates querying on multiple fields
- Bitmap for each distinct field value
  - Contains a "1" for each record in the relation where that attribute value is found
  - Contains a "0" for all other records
- Records are numbered from 1 to n
  - Record id or **row id**
- Row id should be easily mapped to a physical address (block address)

81

81

Bitmap Size

- **Size** of each bitmap (in bits) is equal to the number of rows in the relation
- **Number** of bitmaps for a field is equal to the number of distinct values of that field
- **Total** space needed to index one field (in bits)

  = number of distinct values x number of rows
- Whereas, **file size** (in bits)

  = record size in bits x number of rows
- In general, bitmap indexes are space-efficient
  - Not good for the key columns

82

82

Bitmap Index: Example

Gender Index | Dept Index

	SID	Name	Gender	Dept
1	6007	Peter	M	EEE
2	6100	Ann	F	COMP
3	6107	Bob	M	BE
4	6207	Jane	F	COMP
5	6240	Suzy	F	BE
6	6350	Ben	M	BE
7	6420	Peter	M	COMP
8	6500	Jenn	F	EEE

F	M	BE	COMP	EEE
0	1	0	0	1
1	0	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	0
0	1	1	0	0
0	1	0	1	0
1	0	0	0	1

83

83

Bitmap Index: Example

```
SELECT *
FROM Students S
WHERE S.dept = "COMP"
```

Return the Row_ids of:
all "1"s in the "COMP" bitmap
Rows: 2, 4, 7

	SID	Name	Gender	Dept	COMP
1	6007	Peter	M	EEE	0
2	6100	Ann	F	COMP	1
3	6107	Bob	M	BE	0
4	6207	Jane	F	COMP	1
5	6240	Suzy	F	BE	0
6	6350	Ben	M	BE	0
7	6420	Peter	M	COMP	1
8	6500	Jenn	F	EEE	0

84

84

Bitmap Index: Example

```
SELECT *
FROM Students S
WHERE S.dept = "COMP"
AND  S.gender = "F"
```

	SID	Name	Gender	Dept	COMP	F
1	6007	Peter	M	EEE	0	0
2	6100	Ann	F	COMP	1	1
3	6107	Bob	M	BE	0	0
4	6207	Jane	F	COMP	1	1
5	6240	Suzy	F	BE	0	1
6	6350	Ben	M	BE	0	0
7	6420	Peter	M	COMP	1	0
8	6500	Jenn	F	EEE	0	1

Return the Row_ids of:
all "1"s in the **intersection** of the "*COMP*" and "F" bitmaps
Rows: 2, 4

Disjunctive Query?

Range Query?

85

85

Bitmap Limitations

- Not good for data that is modified regularly
  - Updates will require modifying all the associated bitmap indexes
- Used in warehouse data sets which are large and are not updated frequently
  - Mainly for Online Analytical Processing (OLAP)

86

86