Lecture 15: Access Paths or Index Structures for Files

CS 1555: Database Management Systems

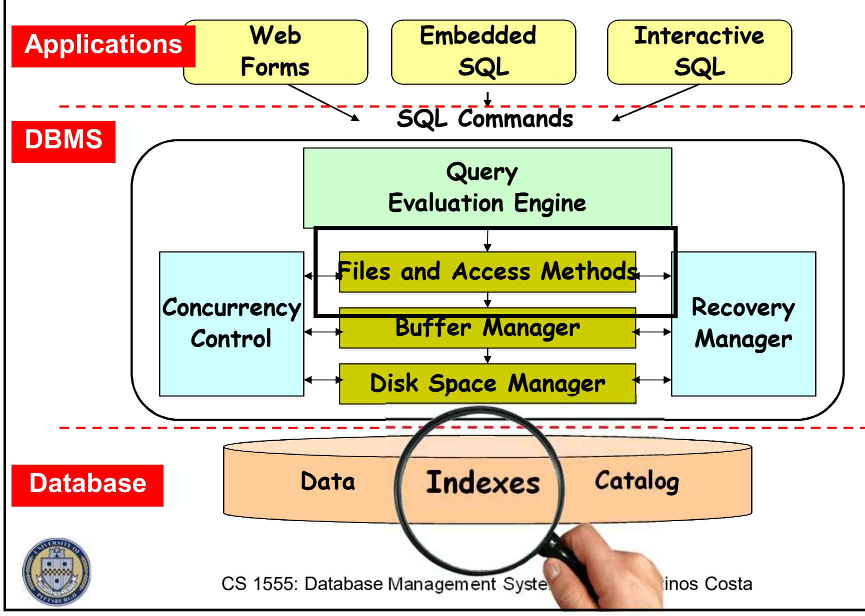
Constantinos Costa

http://db.cs.pitt.edu/courses/cs1555/current.term/

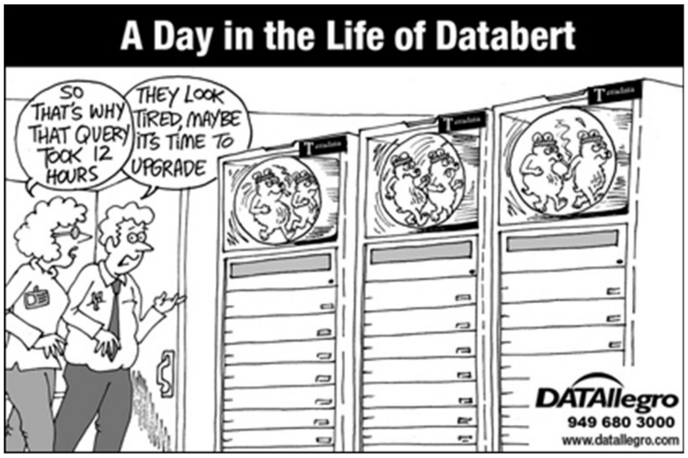
March 21, 2019, 16:00-17:15
University of Pittsburgh, Pittsburgh, PA

Lectures based: P. Chrysanthis & N. Farnan Lectures

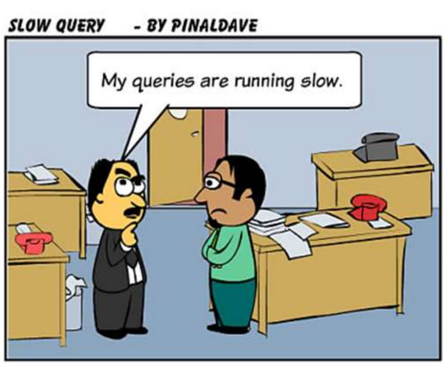
Database Management System (DBMS)



Queries are slow!



Create Index!



Create Index!

Access Paths or Methods

- Tuples are typically stored and retrieved based on the value of the primary key (instead of on some internal Rid)
- Special case of context addressability (alias associate access)
- ***Access Paths*** or ***Address algorithms*** is a class of algorithms designed for translating attribute values into Rid or other type of internal addresses.
- ***Selection predicate*** is the condition based on which the associate access is done:
  - E.g., based on primary key, range queries on primary key, based on secondary keys, etc.

Index Structures

- An index is an ***auxiliary file*** that makes it more efficient to search for a record in a ***data*** file.
- An index is usually specified on one field value of the data file.
- An index is an ordered file of entries

  <*field-value*, *pointer*>

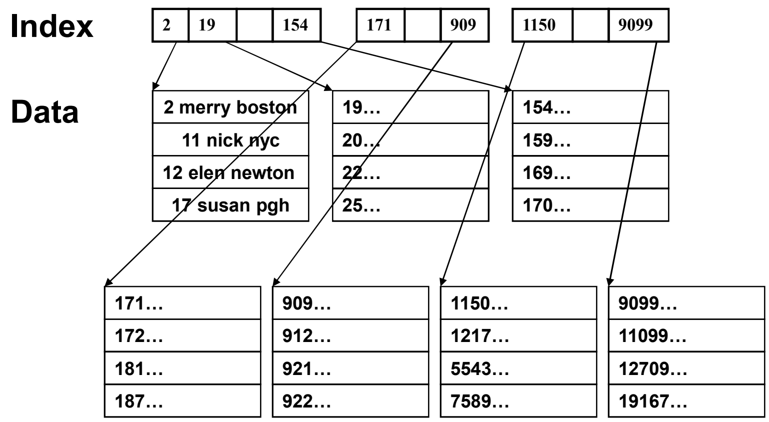
  ordered by field value.

- Examples:
  - Index Sequential Access Method (ISAM)
    - primary, secondary and clustering indexes
  - B tree and B^+ tree

Index-Sequential Access Method (ISAM)

- It is a ***primary index***.
- Defined on an ***ordered*** data file based on a key value.
- The first record in a data block is called ***block anchor***.
- Includes one index entry *for each block* in the data file.
- The field value of an index entry is the key value of the block anchor.
- A similar scheme can use the ***last record*** in a data block.

Example of Primary ISAM



Advantages of a Primary Index

- A primary index might be stored in multiple blocks, but:

it occupies <u>much smaller space</u> than a data file, because:

1. There are <u>fewer</u> index entries than records
2. Each index entry is typically <u>smaller</u> than a data record
   - Index record only 2 fields

- **More index entries than data $_{\text{records}}$ 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: $\text{Log}_2\, B_{index} < \log_2 B_{data}$

Index File vs. Data File!

Data file

Index file

Index Structure

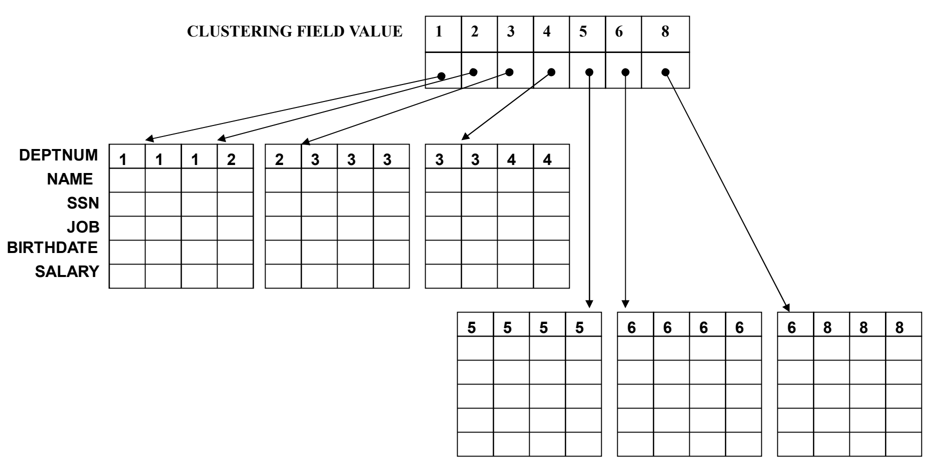
- Single-level Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Multi-level Indexes

Clustering Index

- Defined on an ***ordered*** data file 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

CS 1555: Database Management Systems - Constantinos Costa

Clustering Index: Example 1

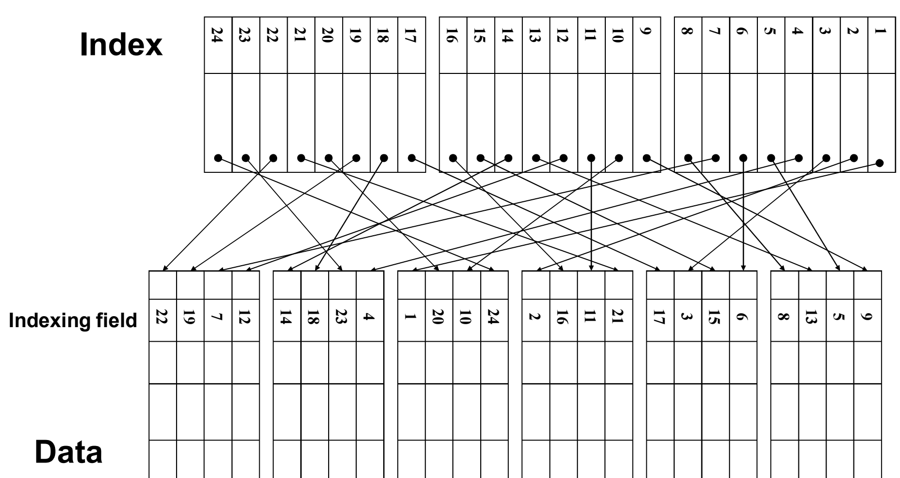


CS 1555: Database Management Systems - Constantinos Costa

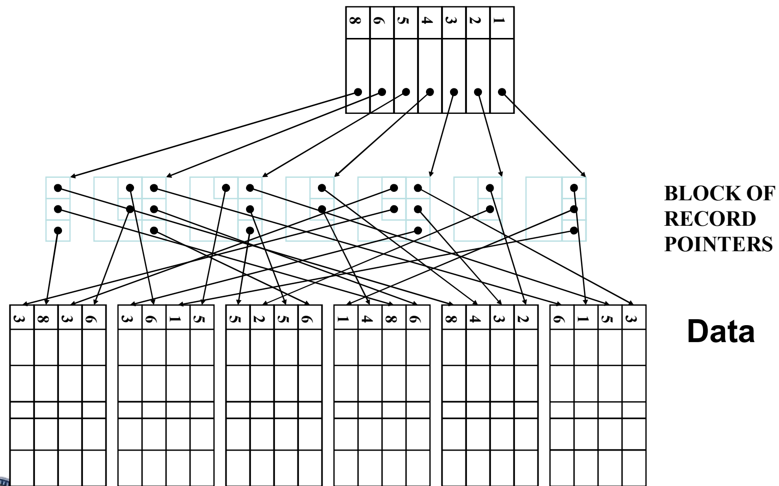
Secondary Index

- Also called ***nonclustering*** index.
- Defined on:
  - an **unordered** data file, or
  - on a **non-ordering** field of an ordered data file
- Can be defined on a key or non-key field.
- Includes one index entry for ***each record*** in the data file.
- Thus, it is a dense index and also called a ***dense index.***
- The index entry points to …?!

Secondary Index on key field



Secondary Index on non-key field



Summary

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

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 (nonkey)	Number of distinct values (1^{st} level)	Sparse
	Number of records/pointers (2^{nd} level)	Dense

Large Index

- For Big Databases

  the Index could be very large to fit in main memory!

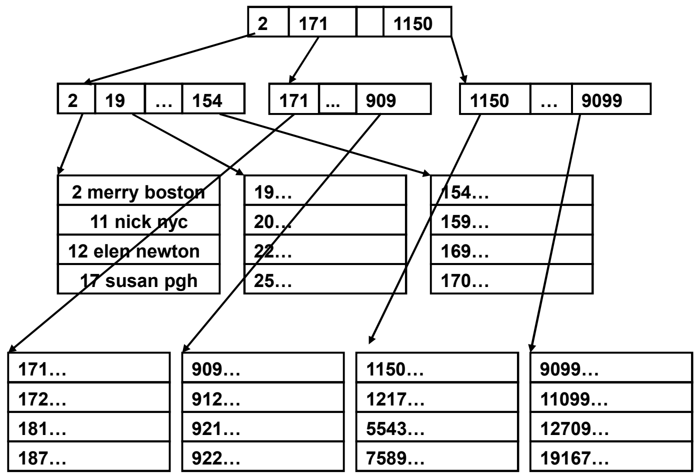
- *Can we do better?*

Multi-Level Index vs. Data!

Multi-Level Indexes

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

Multi-level ISAM

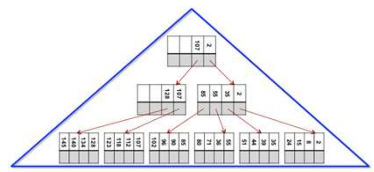


Drawbacks of ISAM

- A static structure.
  - It needs monitoring for dynamic databases.
- Insertion/deletion of new index entry is a <u>problem</u>, why?
  - every level of the index is an **ordered file**
  - Insertion is handled by some form of overflow blocks.
- Active files need frequent reorganization.
- No guaranteed performance for searching based on the key for active files.
  - anywhere between O(***log n***) to O(***n***).

Multi-level index as a tree

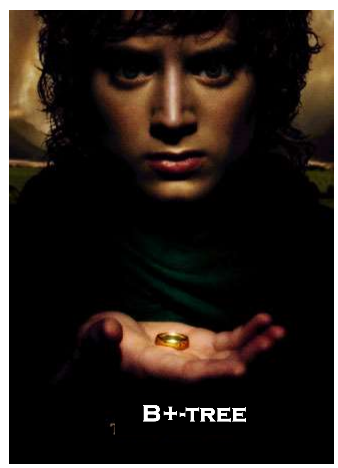
- Multi-level index is a form of search tree
  - Each node has pointers
  - By following a pointer, we restrict our search to a subtree and ignore all other nodes
  - The number of pointers (fan-out) equals the index blocking factor
- But the multi-level indexing we have seen so far is **static**!



B+-Tree

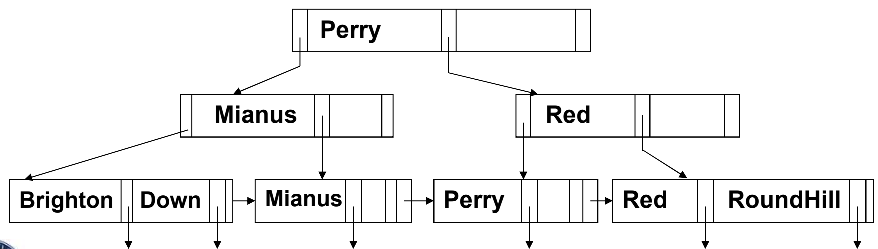
Is there:

One B+-tree to rule them all,
One B+-tree to find them,
One B+-tree to bring them all,
And in the darkness bind them



B-trees and B⁺-trees

- Dynamic multi-level indexes.
- A multi-level index is a form of search tree
- B-trees and B⁺-trees are variations of search trees that allow efficient insertion and deletion of new values.
- Each node in the tree is a disk block
- Each node is kept between half-full and completely full.

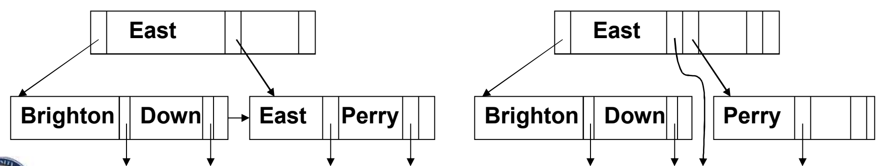


B/B+ tree Performance

- Insertion:
  - Efficient if there is space
  - Otherwise a full node is split.
  - Splitting may propagate to other levels.
- Deletion:
  - Efficient if it does not cause the node to be less than half-full.
  - Otherwise it must be merged with its sibling node or, if not possible (i.e., sibling is full), accept half of the keys of its sibling node.

B-trees Vs. B$^+$-trees

- In a B$^+$- tree, all pointers to data records exist at the leaf-level nodes.
- In a B-tree, pointers to data records exist at all levels.
- A B$^+$- tree can have less levels than the corresponding B-tree.
- In a B$^+$- tree, the search cost is the same for any value, O(***log n***). The tree is always balance.



B$^+$-tree Index

- A node is of the form:

$$[p_0, k_1, p_1, k_2, p_2, .., k_i, p_i, k_{i+1} .., k_n, p_n]$$

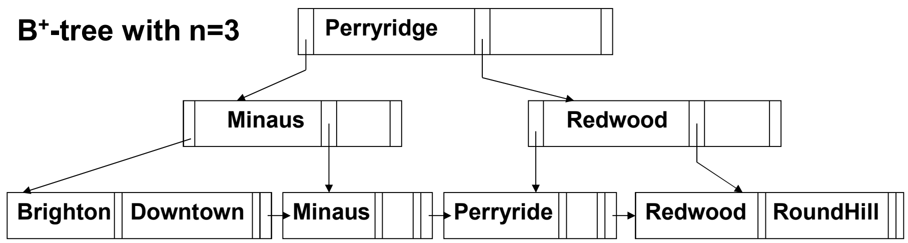
- p_i's are pointers and k_i's are field values (keys)
- ***Tree Order*** is the number of pointers, e.g., ***n***
- For every field value ***k*** in a node pointed to by p_i

$$k_i < k \leq k_{i+1} \quad \text{(alternative } k_i \leq k < k_{i+1}\text{)}$$

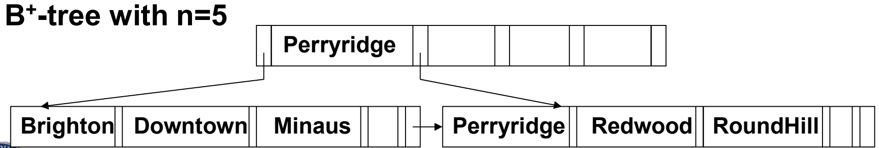
- Every node, except for the root, has between n/2 and n children or pointers
  - internal: $\lceil n/2 \rceil$ **tree pointers**; leaf: $\lfloor n/2 \rfloor$ data pointers
- Leaf nodes are chain to form a link list (***fast sequential access***)

Examples of B+ trees

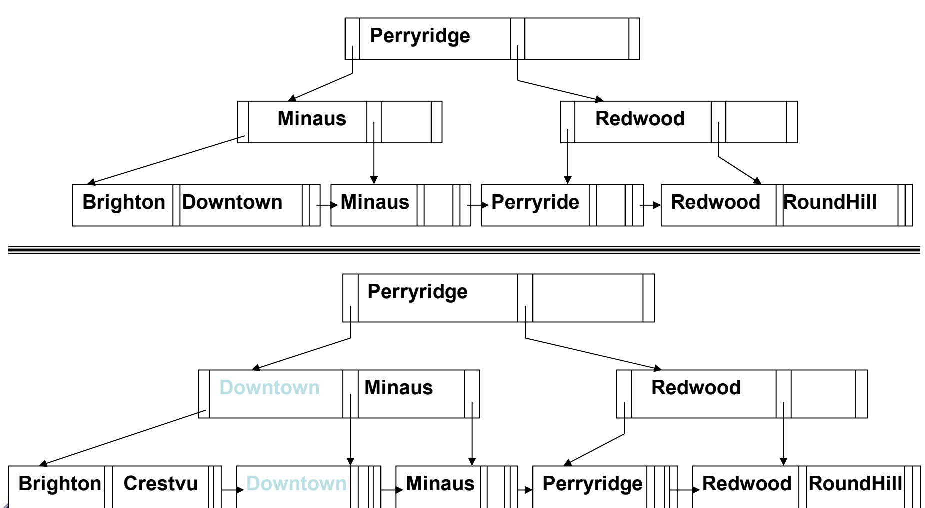
B$^+$-tree with n=3



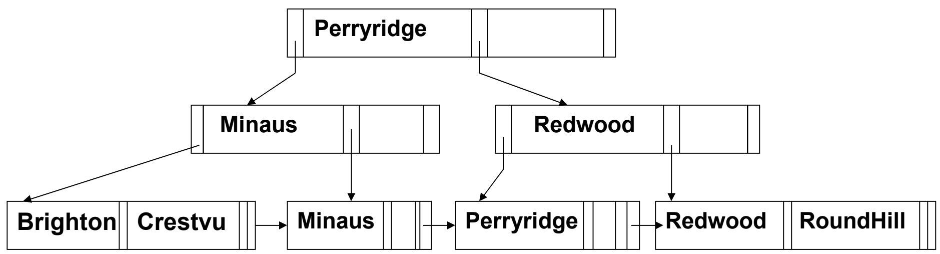
B$^+$-tree with n=5



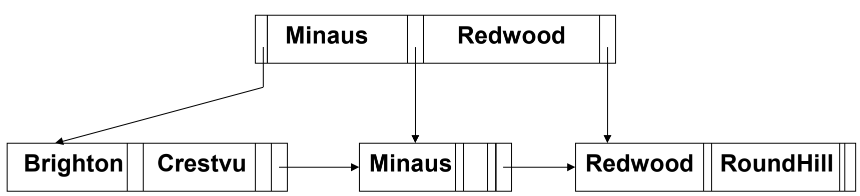
Insertion of "Crestvu"



Deletion of “Downtown”

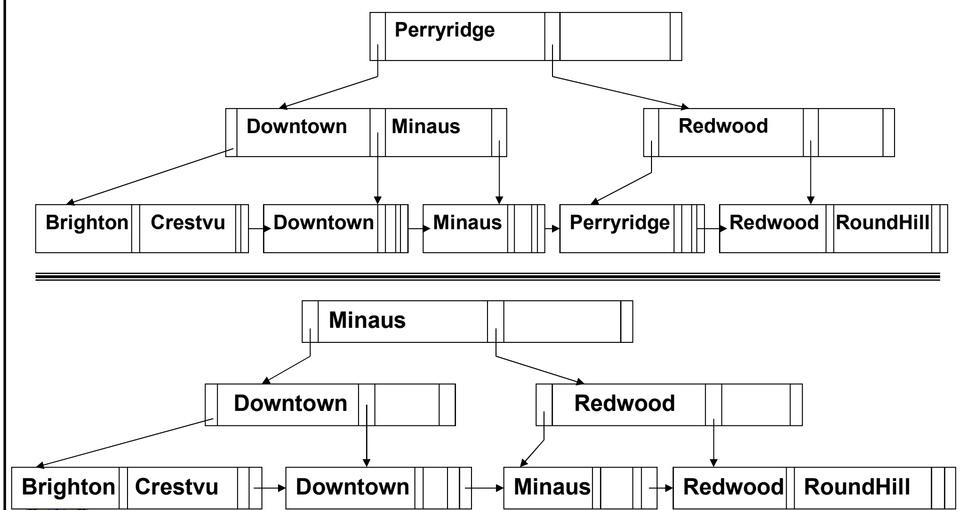


Deletion of “Perryridge”



Deletion of "Perryridge"

(w/out deletion of Downtown)

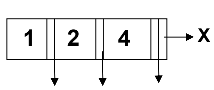


Examples of B+ trees

B$^+$-tree with n=4, i.e.,

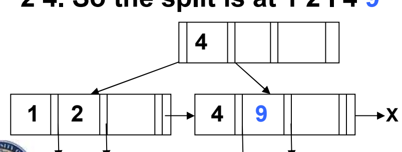
- **internal nodes should have $\lceil 4/2 \rceil$=2 tree pointers;**
- **Leaf nodes should have: $\lfloor 4/2 \rfloor$ = 2 data pointers**

Step 1: insert 1, 2, 4



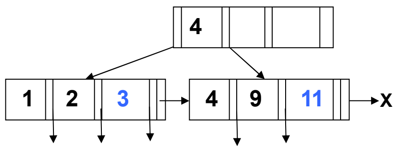
Step 2: insert 9

Although you push 9 on the stack & inserted afterwards when allocating a new block, it is considered when deciding the split: 1 2 4. So the split is at 1 2 I 4 9



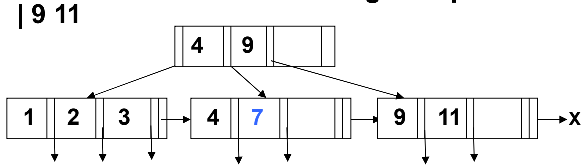
Examples of B+ trees

Step 3: insert 3, 11



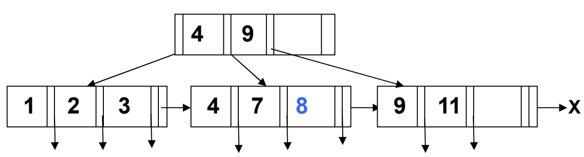
Step 4: insert 7

Although you push 7 on the stack when allocating a new block, it is considered when deciding the split: 4 9 11. So the split is at 4 7 | 9 11



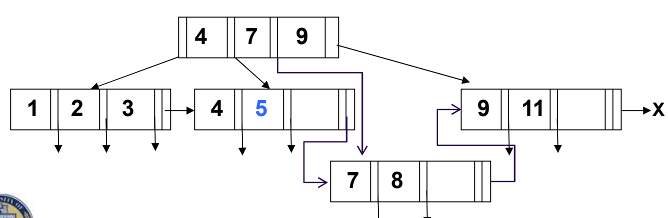
Examples of B+ trees

Step 5: insert 8



Step 6: insert 5

Although you push 5 on the stack when allocating a new block, it is considered when deciding the split: 4 7 8. So the split is at 4 5 | 7 8



Multiple-Key Access

Select name
From Student
Where
((State = 'PA') and
(year(Birthdate) =
'1992'));

- Index only on State
- Index only on Birthdate
- Separate indexes on State and Birthdate:
  - (State or Birthday)
- Index on both State and Birthdate:
  - (State and Birthdate)

Indexes on Multiple Attributes

- ***Multiple Attribute*** indexes use *composite search key*
  - Form of tuple of values: $(a_1, a_2, \ldots, a_n)$
  - *Lexicographical ordering on tuples*

  $(a_1, a_2) < (b_1, b_2) \Rightarrow (a_1 < b_1) \vee ((a_1 = b_1) \wedge (a_2 < b_2))$

- What about comparison conditions (range queries)?

Select name From Student
Where ((State = 'PA') and
(year(Birthdate) < '1992'));

Or

Select name From Student
Where ((year(Birthdate) < '1992') and
(State = 'PA'));

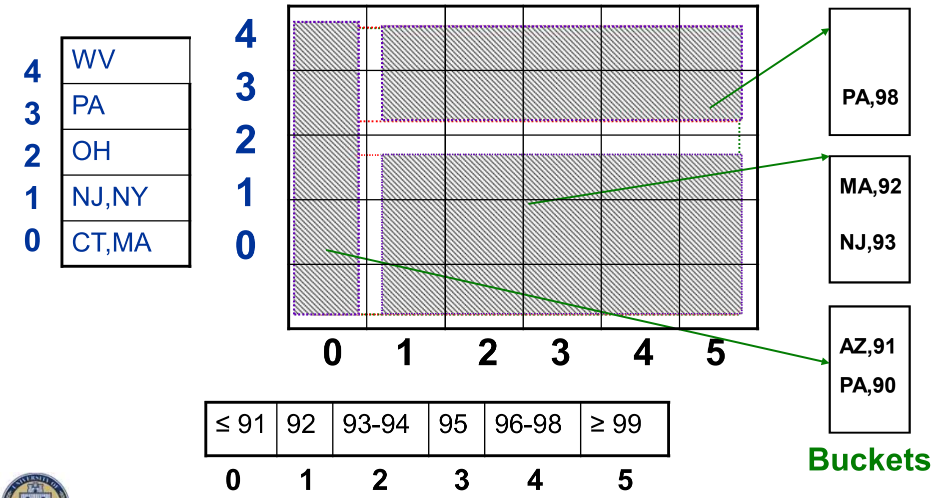
Point Access Methods (PAMs)

- Point Access Methods (PAMs)
  - A k-attribute record is envisioned as a point in a k-dimensional space
  - Can handle range queries
  - Can handle both points and spatial objects
- Examples:
  - Grid Files
  - Quadtrees
  - k-d trees
  - R-trees

Grid Files

- Generalization of extensible hashing with a multi-dimensional directory, but static wrt directories
  - Fixed linear scales or dimensions/directories
  - Dynamic on bucket/block allocation
- Idea:
  - Impose a grid on the address space
  - Adapt grid to data density (re-organization)
  - Each grid cell corresponds to a disk block
  - One or more cells can share a disk block

Grid File (State, Year)

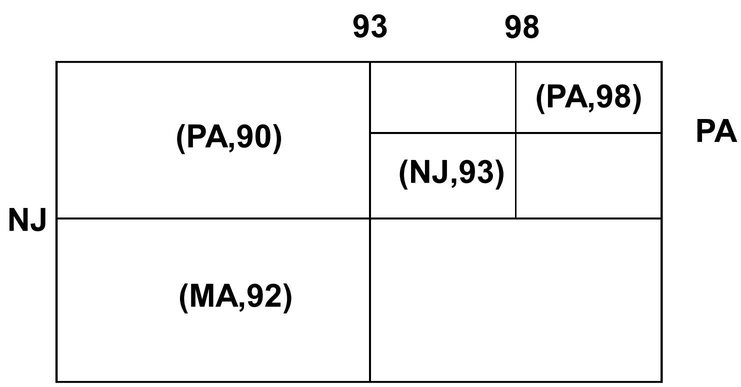


Grid Files Properties

- Pros:
  - Ensures 2 disk accesses for exact match
  - Symmetric w.r.t. the attributes
  - Adapts to non-uniform distributions
- Cons:
  - It does not work well if attributes are correlated
  - It requires extra space for directory which can grow large
  - If insertions are frequently, reorganization becomes costly
    - Hmm, how can this be addressed ?

QuadTree Example

- Dynamic Grid Files: No fixed linear scales
  - Split dimensions on demand



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

Bitmap Index

- **Bitmap index:** facilitates querying on multiples keys
- 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)

Bitmap Index: Example

	SID	Name	Gender	State		Gender Index: F	Gender Index: M		State Index: NY	State Index: PA	State Index: MI
1	6007	Peter	M	MI		0	1		0	0	1
2	6100	Ann	F	PA		1	0		0	1	0
3	6107	Bob	M	NY		0	1		1	0	0
4	6207	Jane	F	PA		1	0		0	1	0
5	6240	Suzy	F	NY		1	0		1	0	0
6	6350	Ben	M	NY		0	1		1	0	0
7	6420	Peter	M	PA		0	1		0	1	0
8	6500	Jenn	F	MI		1	0		0	0	1

Bitmap Index: Example

	SID	Name	Gender	State	PA
1	6007	Peter	M	MI	0
2	6100	Ann	F	PA	1
3	6107	Bob	M	NY	0
4	6207	Jane	F	PA	1
5	6240	Suzy	F	NY	0
6	6350	Ben	M	NY	0
7	6420	Peter	M	PA	1
8	6500	Jenn	F	MI	0

```
SELECT *
FROM Students S
WHERE S.state = "PA"
```

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

Bitmap Index: Example

	SID	Name	Gender	State	PA	F
1	6007	Peter	M	MI	0	0
2	6100	Ann	F	PA	1	1
3	6107	Bob	M	NY	0	0
4	6207	Jane	F	PA	1	1
5	6240	Suzy	F	NY	0	1
6	6350	Ben	M	NY	0	0
7	6420	Peter	M	PA	1	0
8	6500	Jenn	F	MI	0	1

```
SELECT *
FROM Students S
WHERE S.state = "PA"
AND  S.gender = "f"
```

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

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

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)