

Yannis Papakonstantinou, UCSD

Indexing

- Data Structures used for quickly locating tuples that meet a specific type of condition
 - Equality* condition: find Movie tuples where Director=X
 - Other conditions possible, eg, *range* conditions: find Employee tuples where Salary>40 AND Salary<50
- Many types of indexes. Evaluate them on
 - Access time
 - Insertion time
 - Deletion time
 - Disk Space needed

Yannis Papakonstantinou, UCSD

Topics

- Conventional Indexes
- B-Tree* Indexes
- Hashing* Indexes (very different from the two above)

Yannis Papakonstantinou, UCSD

Terms and Distinctions

- Primary index**
 - the index on the attribute (a.k.a. search key) that determines the sequencing of the table
- Secondary index**
 - index on any other attribute
- Dense index**
 - every value of the indexed attribute appears in the index
- Sparse index**
 - many values do not appear

A Dense Primary Index

Yannis Papakonstantinou, UCSD

Dense and Sparse Primary Indexes

Dense Primary Index

Sparse Primary Index

Find the index record with largest value that is less or equal to the value we are looking.

+ can tell if a value exists without accessing file (consider projection)

+ better access to overflow records

+ less index space

more + and - in a while

Yannis Papakonstantinou, UCSD

Multi-Level Indexes

- Treat the index as a file and build an index on it
- “Two levels are usually sufficient. More than three levels are rare.”
- Q: Can we build a dense second level index for a dense index ?

Yannis Papakonstantinou, UCSD

A Note on Pointers

- Record pointers* consist of *block pointer* and position of record in the block
- Using the block pointer only saves space at no extra disk accesses cost

Yannis Papakonstantinou, UCSD

Representation of Duplicate Values in Primary Indexes

- Index may point to first instance of each value only

Yannis Papakonstantinou, UCSD

Deletion from Dense Index

- Deletion from dense primary index file with no duplicate values is handled in the same way with deletion from a sequential file
- Q: What about deletion from dense primary index with duplicates

Delete 40, 80

Yannis Papakonstantinou, UCSD

Deletion from Sparse Index

- if the deleted entry does not appear in the index do nothing

Delete 40

Yannis Papakonstantinou, UCSD

Deletion from Sparse Index (cont'd)

- if the deleted entry does not appear in the index do nothing
- if the deleted entry appears in the index replace it with the next search-key value
 - comment: we could leave the deleted value in the index assuming that no part of the system may assume it still exists without checking the block

Delete 30

Yannis Papakonstantinou, UCSD

Deletion from Sparse Index (cont'd)

- if the deleted entry does not appear in the index do nothing
- if the deleted entry appears in the index replace it with the next search-key value
- unless the next search key value has its own index entry. In this case delete the entry

Delete 40, then 30

Yannis Papakonstantinou, UCSD

Insertion in Sparse Index

- if no new block is created then do nothing
- else create an index entry with the new value

Yannis Papakonstantinou, UCSD

Secondary Indexes

- The file is not sorted according to the secondary search key
- lowest level of index has to be dense

Sparse Index	Dense Index	Unsorted file
10	10	90
50	20	50
100	30	10
150	40	70
	50	10
	70	100
	80	40
	90	30
	100	80
	120	20
		70
		120

Yannis Papakonstantinou, UCSD

Duplicate Values and Secondary Indexes

- store together all pointers with the same search key value

Yannis Papakonstantinou, UCSD

Duplicate Values and Secondary Indexes: Buckets

- store together all pointers with the same search key value
- introduce a separate level of buckets
 - if many pointers for each search key value it is better to separate the pointers from the values

Yannis Papakonstantinou, UCSD

Advantage of Buckets: Process Queries Using Pointers Only

Find employees of the Toys dept with 4 years in the company
 SELECT Name FROM Employee
 WHERE Dept="Toys" AND Year=4

Dept Index

Toys	→	1
PCs	→	2
Pens	→	3
Suits	→	4

Aaron	Suits	4
Helen	Pens	3
Jack	PCs	4
Jim	Toys	4
Joe	Toys	3
Nick	PCs	2
Walt	Toys	5
Yannis	Pens	1

Year Index

1	←	1
2	←	2
3	←	3
4	←	4

Yannis Papakonstantinou, UCSD

Buckets and Pointers Operation Used in Information Retrieval

- known as "inverted lists"
- an entry in an inverted list represents occurrence of a word in an article
- lists range from 1 to 1000000 words
- compression also used

Inverted Lists	Articles
cat	my cat is fat and hairy...
dog	my cat and dog fight all the time...
	Mary hates John's dog

Yannis Papakonstantinou, UCSD

Summary of Indexing So Far

- Basic topics in conventional indexes
 - multiple levels
 - sparse/dense
 - duplicate keys and buckets
 - deletion/insertion similar to sequential files
- Advantages
 - simple algorithms
 - index is equential file
- Disadvantages
 - eventually sequentiality is lost because of overflows

Yannis Papakonstantinou, UCSD

B+ and B-Tree Indexes

- Balanced trees
 - but not extremely balanced
- Advantages (over conventional indexes)
 - no need for reorganization
 - guaranteed upper limits on access, insert, delete times
- Disadvantages
 - overhead on insert, delete, *space*

Yannis Papakonstantinou, UCSD

Properties of B+-trees

if primary index then pointer to tuple for 2
else (if secondary index) pointer to bucket for 2

- every page contains at most $2m$ items
- every page, with the exception of the root, contains at least m items
- leaf pages form a dense index and are at the same depth

Yannis Papakonstantinou, UCSD

Lookup Algorithm

```

lookup(page p, item x)
if p is a leaf page
  if x==p.item[k] return p.pointer[k]
  else return NULL
else if x<p.item[1]
  return lookup(p.pointer[1], x)
else if for some j p.item[j] <= x < p.item[j+1]
  return lookup(p.pointer[j+1], x)
else return lookup(p.pointer[n+1], x)
/*p.pointer[n+1] is the last pointer in the page*/
  
```

Yannis Papakonstantinou, UCSD

Insertion Algorithm

- first locate the leaf page where the item should appear
- if the leaf page is not full simply include item in the page

Yannis Papakonstantinou, UCSD

Insertion Algorithm: Splitting Nodes

if the leaf page has $2m+1$ items after the insertion then

- create a new page with m items
- insert the pointer of the new page and the first item in the parent directory

Yannis Papakonstantinou, UCSD

Insertion: Splitting Recursively

Yannis Papakonstantinou, UCSD

Insertion: Splitting Recursively

Eventually we may have to split the root and create one more level

Yannis Papakonstantinou, UCSD

Deletion: The No-Combining Pages Case

- if the leaf page B where the item x appears has $\geq m+1$ items
 - if x is not the first in the page, simply delete it from the page
 - else find the parent of the leaf page where x appears and update it

Yannis Papakonstantinou, UCSD

Deletion: The No-Combining Pages Case

- if the leaf page block where the item x appears has $\geq m+1$ items
 - if x is not the first in the page, simply delete it from the page
 - else find the parent of the leaf page where x appears and update it

Yannis Papakonstantinou, UCSD

Deletion: The Case for Transferring Items From Siblings

if the leaf page B where x appears has m items
 if there is a neighbor B' (left or right) with $> m$ items
 then transfer the first (or the last item) of B' to B
 and update the appropriate ancestors of B

* transfer the last element of the left neighbor or the first of the right neighbor

Yannis Papakonstantinou, UCSD

Deletion: The Case for Combining Pages

if the leaf page B where x appears has m items
 if there is a neighbor B' (left or right) with $> m$ items
 then ... (see previous page)
 else combine B with a neighbor B'
 delete from parent of B the first item of the rightmost of B and B'

* the deletion from parent may ripple recursively

Yannis Papakonstantinou, UCSD

Deletion: Reducing Levels

When the root is left with two children a deletion may cause removal of a level

deletion often not implemented

Yannis Papakonstantinou, UCSD

The B-Tree Variant: B+-Tree Remains the Winner

- Avoid storing the same item at multiple levels
 - + saves space (but who cares?)
 - non-leaf and leaf nodes contain different numbers of nodes
 - deletion more complicated
 - [Korth&Silberschatz] claims faster lookup for B-Trees because the height of the tree is smaller (because items are stored more compactly). Why this is false ?

Yannis Papakonstantinou, UCSD

Comparison: Static (Conventional) Indexes Vs B+-Trees

- Size and access time comparison
 - for an 8000 block file, after 32000 inserts and 16000 lookups the static index saves enough accesses to "pay" for a reorganization
- Administration
 - a DBA must be in charge of reorganizations
- Buffering
 - B+tree has fixed requirements
 - Static index depends on size of overflow list

Yannis Papakonstantinou, UCSD

An Interesting Problem

- You have the right to set the disk page size for the disk where a B-tree will reside.
- Compute the optimum page size n assuming that
 - The items are 4 bytes long and the pointers are also 4 bytes long.
 - Time to read a node from disk is $12 + .003n$
 - Time to process a block in memory is unimportant
 - B+tree is full (I.e., every page has the maximum number of items and pointers)
- What happens to the optimum n as the transfer time is reduced much more than the seek time ?

Yannis Papakonstantinou, UCSD

Hashing

- hash function $h(\text{key})$ returns address of bucket or record
- for secondary index buckets are required
- if the keys for a specific hash value do not fit into one page the bucket is a linked list of pages

Yannis Papakonstantinou, UCSD

How do we choose a hashing function ?

- Desired property: expected number of keys/bucket is the same for all buckets (*uniform* distribution)
 - and is relatively small
 - ideally all buckets consist of only one page
- Example of a bad hash function for a string attribute
 - first three letters of the string value
- A potentially good hash function
 - $(\text{char}1 + \text{char}2 + \dots + \text{char}N) \text{ modulo } b$, where b is the number of buckets
- Read Knuth Vol. 3 for more on good hashing functions

Yannis Papakonstantinou, UCSD

How Many Buckets ?

- Keep space *utilization* between 50% and 80%
 - utilization = #keys used / total #keys that fit
- How do we cope with growth ?
 - the #keys may be difficult to predict
 - Solution 1: reorganization
 - Solution 2: dynamic hashing
 - extensible hashing

Yannis Papakonstantinou, UCSD

Extensible Hashing

- use i of b bits output by hash function
 - i grows over time
- use directory (bucket address table)
- many consecutive directory entries may point to the same bucket
- associate with this bucket the length of the common prefix

Yannis Papakonstantinou, UCSD

Extensible Hashing Example

$h(\text{key})$ is 4 bits; 2 keys/bucket

Yannis Papakonstantinou, UCSD

Extensible Hashing Example (cont'd)

Yannis Papakonstantinou, UCSD

Indexing vs Hashing

- Hashing is more efficient for probes given key
 - SELECT ... FROM R WHERE R.A=5
- Indexing (conventional and B+trees) handles range searches
 - SELECT ... FROM R WHERE R.A>5 AND R.A<10
- NOTE: Can *not* specify type of index and parameters in SQL
 - system decides

Yannis Papakonstantinou, UCSD

Comparison of Index Methods

	CI	B+	Ha	EH
Cost of reorganization is acceptable				
Insertions/deletions are frequent				
Optimization of average instead of worse				
Range queries are common				

Yannis Papakonstantinou, UCSD

Multi-Key Indexing

- Motivation:** queries of the form
 - SELECT ... FROM R WHERE $cond1$ and $cond2$
 - $cond1$ and $cond2$ are equality or range conditions
- Solution 1:** use index for only one of the conditions
 - suggested if there is a very selective condition
- Solution 2:** pointer intersection
 - fairly selective conditions

SELECT Name FROM Employee WHERE Dept="Toys" AND Year > 3

```

      πName
      |
      σDept="Toys" AND Year>3
      |
      Employee
    
```

Rewriting & Optimization

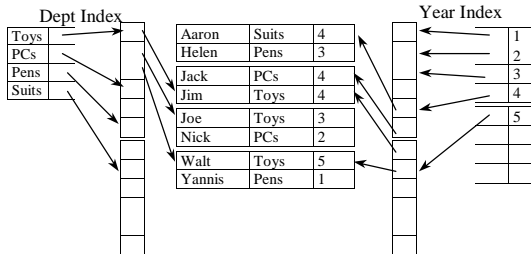
```

      πName
      |
      σSCAN
      |
      Year>3
      |
      σIND
      |
      Dept="Toys"
      |
      Employee
    
```

Employee

Pointer Intersection (Solution 2)

Find employees of the Toys dept with >3 years in the company
 SELECT Name FROM Employee
 WHERE Dept="Toys" AND Year > 3



Solution 3: Multi-Key Indexing

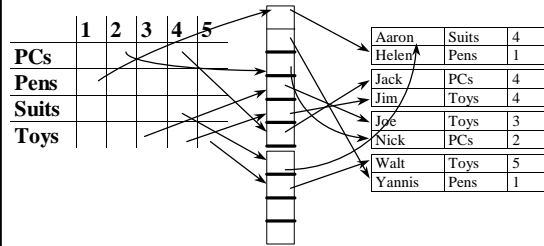
- Appropriate when
 - each condition is not very selective
 - but their conjunction is very selective
- Brute force
- Grid structure
- Partitioned Hash Function

Common Applications of Multi-Key Indexing

- Geographic Data
 - find the city located at latitude 35, longitude 50
 - find cities in within ... coordinates
- Many types of geographic index
 - R-trees: indexing of spatial objects
 - LSD trees: indexing of multidimensional points
 - k-d trees
- Similar indexing methods for multimedia queries
 - find k nearest neighbors



Grid Structure



- Space overhead (very sparse structure)
- Expensive insertion and deletion if new key values