CS 4604: Introduction to Database Management Systems

B. Aditya Prakash

Lecture #9: Storing and Indexes
Announcement

- No class on Tuesday.
- BUT
  - Project Assignment 1 is still due (in class)
  - We will return HW1
  - Pranav and Qianzhou will be present in classroom during the lecture time (as extra office hours)
DBMS Layers:

- Query Optimization and Execution
- Relational Operators
- Files and Access Methods
- Buffer Management
- Disk Space Management

TODAY →
Leverage OS for disk/file management?

- Layers of abstraction are good ... but:
Leverage OS for disk/file management?

- Layers of abstraction are good ... but:
  - Unfortunately, OS often gets in the way of DBMS
Leverage OS for disk/file management?

- DBMS wants/needs to do things “its own way”
  - Specialized prefetching
  - Control over buffer replacement policy
    - LRU not always best (sometimes worst!!)
  - Control over thread/process scheduling
    - “Convoy problem”
      - Arises when OS scheduling conflicts with DBMS locking
  - Control over flushing data to disk
    - WAL protocol requires flushing log entries to disk
DBMS stores information on disks.

– but: disks are (relatively) VERY slow!

Major implications for DBMS design!
Major implications for DBMS design:

- **READ:** disk -> main memory (RAM).
- **WRITE:** reverse

- Both are high-cost operations, relative to in-memory operations, so must be planned carefully!
Why Not Store It All in Main Memory?
Why Not Store It All in Main Memory?

- Costs too much.
  - disk: ~$1/Gb; memory: ~$100/Gb
  - High-end Databases today in the 10-100 TB range.
  - Approx 60% of the cost of a production system is in the disks.

- Main memory is volatile.

- Note: some specialized systems do store entire database in main memory.
The Storage Hierarchy

- Smaller, Faster
- Bigger, Slower
The Storage Hierarchy

- Main memory (RAM) for currently used data.
- Disk for the main database (secondary storage).
- Tapes for archiving older versions of the data (tertiary storage).
Jim Gray’s Storage Latency Analogy: How Far Away is the Data?

10^9 Tape  
10^6 Disk  
100 Memory  
10 On Board Cache  
2 On Chip Cache  
1 Registers

Andromeda: 2,000 Years
This Building: 1.5 hr
This Room: 10 min
My Head: 1 min
Disks

- Secondary storage device of choice.
- Main advantage over tapes: *random access* vs. *sequential*.
- Data is stored and retrieved in units called *disk blocks* or *pages*.
- Unlike RAM, time to retrieve a disk page varies depending upon location on disk.
  - relative placement of pages on disk is important!
Anatomy of a Disk

- Sector
- Track
- Cylinder
- Platter
- Block size = multiple of sector size (which is fixed)
Accessing a Disk Page

- Time to access (read/write) a disk block:
  - 
  - 
  - 
  - 
Accessing a Disk Page

- Time to access (read/write) a disk block:
  - *seek time*: moving arms to position disk head on track
  - *rotational delay*: waiting for block to rotate under head
  - *transfer time*: actually moving data to/from disk surface
Accessing a Disk Page

- Relative times?
  - *seek time:*
  - *rotational delay:*
  - *transfer time:*
Accessing a Disk Page

- Relative times?
  - *seek time*: about 1 to 20msec
  - *rotational delay*: 0 to 10msec
  - *transfer time*: < 1msec per 4KB page
Seek time & rotational delay dominate

- Key to lower I/O cost: reduce seek/rotation delays!
- Also note: For shared disks, much time spent waiting in queue for access to arm/controller
Arranging Pages on Disk

- “Next” block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder

- Accessing ‘next’ block is cheap

- A useful optimization: pre-fetching
  - See textbook page 323
Rules of thumb...

1. Memory access much faster than disk I/O (~ 1000x)
   - “Sequential” I/O faster than “random” I/O (~ 10x)
Disk Arrays: RAID

- Benefits:
  - Higher throughput (via data “striping”)
  - Longer MTTF (via redundancy)
Recall: DBMS Layers

Queries

Query Optimization and Execution

Relational Operators

Files and Access Methods

Buffer Management

Disk Space Management

DB

TODAY →
Buffer Management in a DBMS

Page Requests from Higher Levels

(copy of a) disk page

free frame

buffer pool

choice of frame dictated by replacement policy

MAIN MEMORY

DISK

DB

Just FYI

Prakash 2014
Files

- **FILE**: A collection of pages, each containing a collection of records.

- Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)
Alternative File Organizations

Several alternatives (w/ trade-offs):

– **Heap files**: Suitable when typical access is a file scan retrieving all records.

– **Sorted Files**: 

– **Index File Organizations**: 

} later
Files of records

- Heap of pages
  - as linked list or
  - directory of pages
Heap File Using Lists

- The header page id and Heap file name must be stored someplace.
- Each page contains 2 `pointers’ plus data.
Heap File Using a Page Directory

- Header Page
- DIRECTORY
- Data Page 1
- Data Page 2
- Data Page N
Heap File Using a Page Directory

- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages; linked list implementation is just one alternative.
  
  – Much smaller than linked list of all HF pages!
Page Formats

- fixed length records
- variable length records
Page Formats

Important concept: \textit{rid} == record id

Q0: why do we need it?

Q1: How to mark the location of a record?

Q2: Why not its byte offset in the file?
Important concept: \textit{rid} == record id

Q0: why do we need it?
   A0: eg., for indexing

Q1: How to mark the location of a record?
   A1: \textit{rid} = record id = page-id \& slot-id

Q2: Why not its byte offset in the file?
   A2: too much re-organization on ins/del.
Fixed length records

Q: How would you store them on a page/file?
Fixed length records

Q: How would you store them on a page/file?

A1: How about:

`Packed`

```
+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
  ...  
+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+

N
```

slot #1

slot #2

slot #N

free space

number of full slots
Fixed length records

- A1: How about: **BUT**: On insertion/deletion, we have too much to reorganize/update

![Diagram showing fixed length records]

'Packed'

...  

slot #1  
slot #2

free space

slot #N

number of full slots

N
Fixed length records

- What would you do?
Fixed length records

- Q: How would you store them on a page/file?
- A2: Bitmaps
Variable length records

Q: How would you store them on a page/file?
Variable length records

- Q: How would you store them on a page/file?

- Pack them
- Keep pointers to them

Diagram:
- Occupied records
- Page header
- Slot directory
- Other info (# slots etc)
Variable length records

Q: How would you store them on a page/file?

- pack them
- keep ptrs to them
- mark start of free space

occupied records

page header

slot directory

other info (# slots etc)
Variable length records

- SLOTTED PAGE organization - popular.
Conclusions---Storing

- Memory hierarchy
- Disks: (>1000x slower) - thus
  - pack info in blocks
  - try to fetch nearby blocks (sequentially)
- Record organization: Slotted page
TREE INDEXES
Declaring Indexes

- No standard!
- Typical syntax:

```sql
CREATE INDEX StudentsInd ON Students(ID);
CREATE INDEX CoursesInd ON Courses(Number, DeptName);
```
Types of Indexes

- **Primary**: index on a key
  - Used to enforce constraints

- **Secondary**: index on non-key attribute

- **Clustering**: order of the rows in the data pages correspond to the order of the rows in the index
  - Only one clustered index can exist in a given table
  - Useful for range predicates

- **Non-clustering**: physical order not the same as index order
Given a value \( v \), the index takes us to only those tuples that have \( v \) in the attribute(s) of the index.

E.g. (use CourseInd index)

```sql
SELECT Enrollment FROM Courses
WHERE Number = "4604" and DeptName = "CS"
```
Using Indexes (1): Equality Searches

- Given a value $v$, the index takes us to only those tuples that have $v$ in the attribute(s) of the index.

- Can use Hashes, but see next
Using Indexes (2): Range Searches

- ``Find all students with gpa > 3.0’ ’’
- may be slow, even on sorted file
- Hashes not a good idea!
- What to do?
Range Searches

- "Find all students with gpa > 3.0"
- may be slow, even on sorted file
- Solution: Create an `index` file.
Range Searches

- More details:
- if index file is small, do binary search there
- Otherwise??
B-trees

- the most successful family of index schemes (B-trees, B+-trees, B*-trees)
- Can be used for primary/secondary, clustering/non-clustering index.
- balanced “n-way” search trees
B-trees

- Eg., B-tree of order \( d=1 \):
B - tree properties:

- each node, in a B-tree of order d:
  - Key order
  - at most \( n=2d \) keys
  - at least \( d \) keys (except root, which may have just 1 key)
  - all leaves at the same level
  - if number of pointers is \( k \), then node has exactly \( k-1 \) keys
  - (leaves are empty)
Properties

- “block aware” nodes: each node is a disk page
- $O(\log (N))$ for everything! (ins/del/search)
- typically, if $d = 50 - 100$, then 2 - 3 levels
- utilization $\geq 50\%$, guaranteed; on average 69%
Queries

- Algo for exact match query? (eg., ssn=8?)
JAVA animation

- http://slady.net/java/bt/
Queries

- Algo for exact match query? (eg., ssn=8?)
 Queries

- Algo for exact match query? (eg., ssn=8?)
Queries

- Algo for exact match query? (eg., ssn=8?)
Queries

- Algo for exact match query? (eg., ssn=8?)
Queries

- what about range queries? (eg., 5<salary<8)
- Proximity/ nearest neighbor searches? (eg., salary ~ 8 )
Queries

- what about range queries? (eg., 5<salary<8)
- Proximity/ nearest neighbor searches? (eg., salary ~ 8 )
Queries

- what about range queries? (e.g., 5<salary<8)
- Proximity/nearest neighbor searches? (e.g., salary ~ 8)
Queries

- what about range queries? (eg., 5<salary<8)
- Proximity/ nearest neighbor searches? (eg., salary ~ 8 )
Queries

- what about range queries? (eg., 5<salary<8)
- Proximity/ nearest neighbor searches? (eg., salary ~ 8 )
Variations

- How could we do even better than the B-trees above?
B+ trees - Motivation

- B-tree – print keys in sorted order:
B+ trees - Motivation

- B-tree needs back-tracking – how to avoid it?

Diagram showing a B+ tree with keys 1, 3, 7, 9, and 13.
B+ trees - Motivation

- Stronger reason: for clustering index, data records are scattered:
Solution: B+ - trees

- facilitate sequential ops
- They string all leaf nodes together
- AND
- replicate keys from non-leaf nodes, to make sure every key appears at the leaf level
- (vital, for clustering index!)
B+ trees
B+ trees

Index Pages

Data Pages

<6

>=6

>=9

<9

1 3

6 9

6 7

9 13

Prakash 2014

VT CS 4604
B+ trees

- More details: next (and textbook)
- In short: on split
  - at leaf level: COPY middle key upstairs
  - at non-leaf level: push middle key upstairs (as in plain B-tree)
Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf
- Search for 5*, 15*, all data entries >= 24* ...

Based on the search for 15*, we know it is not in the tree!
Inserting a Data Entry into a B+ Tree

- Find correct leaf L.
- Put data entry onto L.
  - If L has enough space, done!
  - Else, must split L (into L and a new node L2)
    - Redistribute entries evenly, copy up middle key.

- parent node may overflow
  - but then: push up middle key. Splits “grow” tree; root split increases height.
Example B+ Tree – Inserting 30*

Root

13 17 24

2* 3* 5* 7* 14* 16* 19* 20* 22* 23* 24* 27* 29*
Example B+ Tree – Inserting 30*

Root

2* 3* 5* 7* 14* 16* 19* 20* 22* 23* 24* 27* 29* 30*

Prakash 2014
Example B+ Tree - Inserting 8*
Example B+ Tree - Inserting 8*

No Space
Example B+ Tree - Inserting 8*

So Split!
Example B+ Tree - Inserting 8*

So Split!

And then push middle UP

Prakash 2014
Example B+ Tree - Inserting 8*

Final State
Example B+ Tree - Inserting 21*
Example B+ Tree - Inserting 21*

Root is Full, so split recursively

Prakash 2014
Example B+ Tree: Recursive split

• Notice that root was also split, increasing height.
Example: Data vs. Index Page Split

- leaf: ‘copy’
- non-leaf: ‘push’
- why not ‘copy’ @ non-leaves?
Same Inserting 21*: The Deferred Split

Note this has free space. So…
Inserting 21*: The Deferred Split

LEND keys to sibling, through PARENT!
Inserting 21*: The Deferred Split

Shorter, more packed, faster tree
Insertion examples for you to try

Insert the following data entries (in order): 28*, 6*, 25*
After inserting 28*, 6*

After inserting 25*
Answer...

After inserting 25*
Deleting a Data Entry from a B+ Tree

- Start at root, find leaf $L$ where entry belongs.
- Remove the entry.
  - If $L$ is at least half-full, done!
  - If $L$ underflows
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as $L$).
    - If re-distribution fails, merge $L$ and sibling.
      - update parent
      - and possibly merge, recursively
Deletion from B+Tree
Example: Delete 19* & 20*

Deleting 19* is easy:

- DeleAng 19* -→ redistribution (no accrued)
- 27 copied up

• Deleting 20* -→ re-distribution (notice: 27 copied up)
... And Then Deleting 24*

Praka: • Must merge leaves: OPPOSITE of insert
... And Then Deleting 24*

Root

3

... but are we done??

Root

4

Praka: • Must **merge** leaves: OPPOSITE of insert
... Merge Non-Leaf Nodes, Shrink Tree

Prakash 2014
VT CS 4604
Example of Non-leaf Re-distribution

- Tree is shown below during deletion of 24*.
- Now, we can re-distribute keys
- need only re-distribute ‘20’; did ‘17’, too
- why would we want to re-distribute more keys?

After Re-distribution
Main observations for deletion

- If a key value appears twice (leaf + nonleaf), the above algorithms delete it from the leaf, only
- why not non-leaf, too?
Main observations for deletion

- If a key value appears twice (leaf + nonleaf), the above algorithms delete it from the leaf, only
- why not non-leaf, too?
- ‘lazy deletions’ - in fact, some vendors just mark entries as deleted (~ underflow), – and reorganize/compact later
Recap: main ideas

- on overflow, split (and ‘push’, or ‘copy’)
  - or consider deferred split

- on underflow, borrow keys; or merge
  - or let it underflow...
B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = \(2 \times 100 \times 0.67 = 134\)

- Typical capacities:
  - Height 4: 1334 = 312,900,721 entries
  - Height 3: 1333 = 2,406,104 entries
B+ Trees in Practice

- Can often keep top levels in buffer pool:
  - Level 1 = 1 page = 8 KB
  - Level 2 = 134 pages = 1 MB
  - Level 3 = 17,956 pages = 140 MB
B+ trees with duplicates

- Everything so far: assumed unique key values
- How to extend B+-trees for duplicates?
  - Alt. 2: <key, rid>
  - Alt. 3: <key, {rid list}>
- 2 approaches, roughly equivalent
B+ trees with duplicates

- approach#1: repeat the key values, and extend B+ tree algo’s appropriately - e.g. many ‘14’ s
B+ trees with duplicates

- approach#1: subtle problem with deletion:
- treat rid as part of the key, thus making it unique
B+ trees with duplicates

- approach #2: store each key value: once
- but store the \{rid list\} as variable-length field (and use overflow pages, if needed)
B+trees in Practice

- prefix compression;
- bulk-loading;
- ‘order’
Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic’; can often compress them.
Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic`; can often compress them.
Bulk Loading of a B+ Tree

- In an empty tree, insert many keys
- Why not one-at-a-time?
  - Too slow!
Bulk Loading of a B+ Tree

- Initialization: Sort all data entries
- scan list; whenever enough for a page, pack
- <repeat for upper level>
Bulk Loading of a B+ Tree

Root

Data entry pages not yet in B+ tree

Bulk Loading of a B+ Tree

Root

Data entry pages not yet in B+ tree

Prakash 2014
A Note on `Order’

- Order (d) concept replaced by physical space criterion in practice (`at least half-full’).

- Why do we need it?
  - Index pages can typically hold many more entries than leaf pages.
  - Variable sized records and search keys mean different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value (duplicates) can lead to variable-sized data entries (if we use Alternative (3)).
A Note on `Order’

- Many real systems are even sloppier than this: they allow underflow, and only reclaim space when a page is completely empty.

- (what are the benefits of such ‘slopiness’?)
Conclusions

- B+tree is the prevailing indexing method
- Excellent, $O(\log N)$ worst-case performance for ins/del/search; (~3-4 disk accesses in practice)
- Guaranteed 50% space utilization; avg 69%
Conclusions

- Can be used for any type of index: primary/secondary, sparse (clustering), or dense (non-clustering)
- Several fine-extensions on the basic algorithm
  - deferred split; prefix compression; (underflows)
  - bulk-loading
  - duplicate handling