Tuesday 24/2/2015: Data Storage

Lecture Topics

I. Data on External Storage
II. File Organization
III. Cost Model
IV. Data Structures
V. Indexing

I. Data on External Storage

SQL

DBMS

Query Evaluation Engine

Files and Process Methods

Buffer Manager

Recycle Manager

Concurrency Control

Index File

Data Files

II. Files

II.1. File is a sequence of records

- Records are fixed or variable size

Logical view: File is a sequence of blocks (fixed size, not contiguous)

Physical view: File is a sequence of blocks

- Easy to find: first, last, next, prev block

II.2. Assumptions

- Fixed size records
- No record in more than 1 block
- Several records per block
- "Left over" space at end of block

II.3. Records

<table>
<thead>
<tr>
<th>Records</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>11200</td>
</tr>
<tr>
<td>2</td>
<td>7120</td>
</tr>
<tr>
<td>3</td>
<td>5200</td>
</tr>
<tr>
<td>1</td>
<td>1220</td>
</tr>
<tr>
<td>8</td>
<td>8120</td>
</tr>
<tr>
<td>8</td>
<td>8120</td>
</tr>
<tr>
<td>3</td>
<td>2120</td>
</tr>
<tr>
<td>9</td>
<td>91200</td>
</tr>
</tbody>
</table>

II.4. To answer a query:

- Read all blocks into RAM
- Get relevant data from blocks
- Additional processing to answer

Q: How to make this fast?
III. Cost Model

In this lecture:

- Reader Write block = 1 unit of time
- Processing RAM = free
- Ignore Caching

Justification:

- Disk IO > CPU
- Don’t want to model disk contiguity
- Don’t want to model cache slots

Goal => Minimize number of block accesses

Heuristic => Make each block read/write as “useful” as possible

Implications:

If you know where E2, E4 are

- data structure cost model = 2 (RAM access)
- database cost model = 1 (block access)

#2, #1

- data structure = 2
- data base = 2

III.2 Operations

- Scan
- Equality
- Range
- Insert
- Delete

Tools:

1. File Organization

2. Indexes (structure slowly where records are)

(1) = when you read a block, many useful records
(2) = know where the blocks are

- Maintaining F.O. and index is not free
- Extreme cases: only read or only write

IV. Data Structures

- Heap (unsorted sequence, different from data structure “heap” and process “heap”)

- Sorted sequence

- Hashing

- 2-3 tree

IV.1 Heap

Find: $O(n)$ operations
Delete: $O(n)$ operations => maybe computed?
Insert: $O(1)$ or $O(n)$

IV.2 Sorted Sequence

Find: $O(\log N)$ binary search
Delete: $O(\log N)$ or $O(\log N + W)$

Insert: $O(\log N)$ or $O(\log N + W)$

Find and push to tail
III. Hashing
- Pick a B bigger than N
- function h
  h: $\mathbb{Z} \rightarrow B$
- bucket directory

- Assume complexity 'h' is tree
  - Find: $O(1)$ or $O(n)$
  - Insert: $O(1)$ or $O(n)$
  - Problem if B is too small, need to "grow" and rebalance
  - Can "amortize" this cost
  - Delete: $O(1)$ or $O(n+1)$

IV.4 2-3 trees

- Rooted (has a root) and directed (order of children matters)
- All paths from root to leaves are the same
- For each child of a node, there is an index value
- For non-leaf nodes, index indicates the largest value of the leaf in the subtree
- Each leaf has 2 or 3 values

May need to "restructure" when you insert or delete
- Restructuring is linear in the number of levels of the tree $= O(\log_2 N)$ or $O(\log_3 N)$

Find: $O(\log_2 N)$
Insert: $O(\log_3 N)$
Delete: $O(\log_3 N)$

Which to use?
- if large $N$, use hash or 2-3
- if many "range", use 2-3
- if not many range, use hash
1. Indexing
   - Data file of blocks
     - block of records
     - Each record has a "key"
   - Index file
     - Records of the form (key, block address)
     - Each block points to the block of the file
     - Not saying that there is an index record for every key

   ```
   index
   file
   = binary search tree
   ```

   Binary Search tree:

   ```
   1
   2 4
   5 7
   8 9
   ```

   Dense index - for every record (key) in the database, there is a pointer in the index to the block containing it.

   Sparse index - opposite of dense

   => remember, pointers point to blocks, not records. Once the block is in RAM, it can be found quickly.

   - Clustered - a file can be "fully" sorted, without many records between blocks
   - Unclustered - opposite

   Best Case Scenario:
   - Clustered + Sparse
     - Clustered => lots of "related" records
     - Sparse => efficient to find a block
   - Example: easy to get all records
     - with a value greater than 4

   Summary:
   - Sparse + unclustered = Bad, cannot find records easily
   - Dense + clustered = unnecessarily large index
   - Dense + unclustered = good
   - Sparse + clustered = best