Physical Data Organization and Indexing

Introduction to Databases
CompSci 316 Fall 2017

Announcements (Mon., Mar. 6)

- Homework #3 to be posted today
  - will be updated after each lecture

Today:

- Finish physical organization
- Indexes

Recall: cost for DB = mostly I/O

- Reading from/writing to disk is a major source of cost
- It’s all about reducing I/O’s!
- Cache blocks from stable storage in memory
  - DBMS maintains a memory buffer pool of blocks
  - Reads/writes operate on these memory blocks
  - Dirty (updated) memory blocks are “flushed” back to stable storage
- Sequential I/O is much faster than random I/O
  - try to store records that are likely to be accessed together close to each other

Recall: different storage layouts

- Record layouts
  - how attributes are stored in a record
- Block layout
  - how records are stored in a block
  - block = unit of I/O
  - sometimes unit of I/O in terms of a “page”, and a block can contain multiple pages
  - basic idea remains the same

Recall: Record layout

- Record = row or tuple in a table
  - “fixed format” dictated by table schema in relational databases
- Fixed-Length Records
- Variable-Length Records
Fixed-length fields

- All field lengths and offsets are constant
  - Computed from schema, stored in the system catalog
- Common to start fields at locations multiple of 4 or 8
- Often record starts with a header
  - Pointer to the schema
  - Length of the record
  - Timestamp for last read/write
  - Pointers to the fields
- Example: CREATE TABLE User(uid INT, name CHAR(20), age INT, pop FLOAT);

Block layout for fixed length records

- Header may contain
  - Links to some other “related” blocks, e.g. from overflow in indexes
  - Information about the relation the block belongs to
  - Directory for offset of each record
  - Timestamp for last read/write
  - Etc.

Variable Length Records - motivation

1. Data size may vary
   - Address (up to 255 bytes, typically < 50 bytes), name
   - Waste of space in fixed length
2. Repeating fields
   - E.g. pointers for a many-many relationship
   - The number of references may vary
3. Variable format records
   - Do not know at the beginning (XML)
4. Enormous fields
   - Like videos
   - Recall BLOBs from Lecture 13

Variable-length records

- Put all variable-length fields at the end after all fixed length fields (why?)
- Example: CREATE TABLE User(uid INT, name VARCHAR(20), age INT, pop FLOAT, comment VARCHAR(100));
- Approach 1: use field delimiters (’\0’ okay)
- Approach 2: use an offset array
- Pros/cons of approach 2?

Specific approaches

- NSM:
  - N-ary storage model
  - Standard row-major order
- PAX:
  - Partition Attributes Across
  - (If you are interested, see this: http://www.pdl.cmu.edu/PDL-FTP/database/pax.pdf)
- Column store
  - Store records in column-major order

NSM

- Store records from the beginning of each block
- Use a directory at the end of each block
  - To locate records and manage free space
  - Necessary for variable-length records

Why store data and directory at two different ends?

So both can grow easily!
Options

• Reorganize after every update/delete to avoid fragmentation (gaps between records)
  • Need to rewrite half of the block on average

• A special case: What if records are fixed-length?
  • Option 1: reorganize after delete
    • Only need to move one record
    • Need a pointer to the beginning of free space
  • Option 2: do not reorganize after update
    • Need a bitmap indicating which slots are in use

Cache behavior of NSM

• Query: SELECT uid FROM User WHERE pop > 0.8;
  • Assumptions: no index, and cache line size < record size
  • Lots of cache misses
    • loads unnecessary attributes
    • uid and pop are not close enough by memory standards

Beyond block layout: column stores

• The other extreme: store tables by columns instead of rows
  • e.g. one relation can store NSM, other PAX
  • or first do vertical partitioning, then use PAX for storing

Summary

• Storage hierarchy
  • Why I/O's dominate the cost of database operations

• Disk
  • Steps in completing a disk access
  • Sequential versus random accesses

• Record layout
  • Handling variable-length fields
  • Handling NULL
  • Handling modifications

• Block layout
  • NSM: the traditional layout (row store)
  • PAX: a layout that tries to improve cache performance

Index
What are indexes for?

- Given a value, locate the record(s) with this value
  
  $\text{SELECT * FROM } R \text{ WHERE } A = \text{value};$
  
  $\text{SELECT * FROM } R, S \text{ WHERE } R.A = S.B;$
  
- Find data by other search criteria, e.g.
  
  - Range search
    
    $\text{SELECT * FROM } R \text{ WHERE } A > \text{value};$
  
  - Keyword search

Index classification

- Dense vs. Sparse
- Clustered vs. unclustered
- Primary vs. Secondary
- Tree-based vs. Hash-based
  
  - we will only do tree indexes in 316

- Discussion on structure of indexes and pages: on whiteboard

Dense and sparse indexes

- **Dense:** one index entry for each search key value
  
  - One entry may "point" to multiple records (e.g., two users named Jessica)
  
- **Sparse:** one index entry for each block
  
  - Records must be clustered according to the search key

Dense versus sparse indexes

- **Index size**
  
  - Sparse index is smaller

- **Requirement on records**
  
  - Records must be clustered for sparse index

- **Lookup**
  
  - Sparse index is smaller and may fit in memory
  
  - Dense index can directly tell if a record exists

- **Update**
  
  - Easier for sparse index

Primary and secondary indexes

- **Primary index**
  
  - Created for the primary key of a table
  
  - Records are usually clustered by the primary key
  
  - Can be sparse

- **Secondary index**
  
  - Usually dense

- **SQL**
  
  - PRIMARY KEY declaration automatically creates a primary index, UNIQUE key automatically creates a secondary index
  
  - Additional secondary index can be created on non-key attribute(s): $\text{CREATE INDEX UserPopIndex ON User(pop)};$

ISAM

- What if an index is still too big?
  
  - Put a another (sparse) index on top of that!

  - ISAM (Index Sequential Access Method), more or less

Example: look up 197
Updates with ISAM

- Overflow chains and empty data blocks degrade performance
  - Worst case: most records go into one long chain, so lookups require scanning all data!

Example: insert 107
Example: delete 129

Index blocks

Data blocks

B+-tree

- A hierarchy of nodes with intervals
- Balanced (more or less): good performance guarantee
- Disk-based: one node per block; large fan-out

B+-tree balancing properties

- Height constraint: all leaves at the same lowest level
- Fan-out constraint: all nodes at least half full (except root)

<table>
<thead>
<tr>
<th></th>
<th>Max # pointers</th>
<th>Max # keys</th>
<th>Min # active pointers</th>
<th>Min # keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-leaf</td>
<td>f</td>
<td>f – 1</td>
<td>[f/2]</td>
<td>[f/2] – 1</td>
</tr>
<tr>
<td>Root</td>
<td>f</td>
<td>f – 1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Leaf</td>
<td>f</td>
<td>f – 1</td>
<td>[f/2]</td>
<td>[f/2]</td>
</tr>
</tbody>
</table>

Lookups

- SELECT * FROM R WHERE k = 179;
- SELECT * FROM R WHERE k = 32;

Range query

- SELECT * FROM R WHERE k > 32 AND k < 179;

Sample B+-tree nodes

Not found

To keys

100 ≤ k

Non-leaf

Max fan-out: 4

To keys

100 ≤ k < 120

120 ≤ k < 150

150 ≤ k < 180

180 ≤ k

Leaf

To keys

100 ≤ k

To records with these k values;
or, store records directly in leaves

To next leaf node in sequence

Max fan-out: 4

Look up 32...

And follow next leaf pointers until you hit upper bound
**Insertion**

- Insert a record with search key value 32

**Another insertion example**

- Insert a record with search key value 152

**Node splitting**

**More node splitting**

- In the worst case, node splitting can “propagate” all the way up to the root of the tree (not illustrated here)
- Splitting the root introduces a new root of fan-out 2 and causes the tree to grow “up” by one level

**Deletion**

- Delete a record with search key value 130

**Stealing from a sibling**

- Need to add to parent node a pointer to the newly created node
- In the worst case, node splitting can “propagate” all the way up to the root of the tree (not illustrated here)
- Splitting the root introduces a new root of fan-out 2 and causes the tree to grow “up” by one level

Remember to fix the key in the least common ancestor of the affected nodes
Another deletion example

- Delete a record with search key value 179

Coalescing

- Deletion can “propagate” all the way up to the root of the tree (not illustrated here)
- When the root becomes empty, the tree “shrinks” by one level

Performance analysis

- How many I/O’s are required for each operation?
  - \( h \), the height of the tree (more or less)
  - Plus one or two to manipulate actual records
  - Plus \( O(h) \) for reorganization (rare if \( f \) is large)
  - Minus one if we cache the root in memory
- How big is \( h \)?
  - Roughly \( \log_{\text{base}} N \), where \( N \) is the number of records
  - \( B^* \)-tree properties guarantee that fan-out is least \( f/2 \) for all non-root nodes
  - Fan-out is typically large (in hundreds)—many keys and pointers can fit into one block
  - A 4-level \( B^* \)-tree is enough for “typical” tables

B*-tree in practice

- Complex reorganization for deletion often is not implemented (e.g., Oracle)
  - Leave nodes less than half full and periodically reorganize
- Most commercial DBMS use \( B^* \)-tree instead of hashing-based indexes because \( B^* \)-tree handles range queries

The Halloween Problem

- Story from the early days of System R...
  
  UPDATE Payroll
  SET salary = salary * 1.1
  WHERE salary <= 25000;
  
  - There is a \( B^* \)-tree index on Payroll(salary)
  - The update never stopped (why?)
- Solutions?
  - Scan index in reverse, or
  - Before update, scan index to create a “to-do” list, or
  - During update, maintain a “done” list, or
  - Tag every row with transaction/statement id

B*-tree versus ISAM

- ISAM is more static; \( B^* \)-tree is more dynamic
- ISAM can be more compact (at least initially)
  - Fewer levels and I/O’s than \( B^* \)-tree
- Overtime, ISAM may not be balanced
  - Cannot provide guaranteed performance as \( B^* \)-tree does
**B+-tree versus B-tree**

- **B-tree**: why not store records (or record pointers) in non-leaf nodes?
  - These records can be accessed with fewer I/O’s
- **Problems**:
  - Storing more data in a node decreases fan-out and increases \( h \)
  - Records in leaves require more I/O’s to access
  - Vast majority of the records live in leaves!

**Beyond ISAM, B-, and B+-trees, and hash**

- **Other tree-based indexes**: R-trees and variants, GiST, etc.
  - How about binary tree?

- **Text indexes**: inverted-list index, suffix arrays, etc.
- **Other tricks**: bitmap index, bit-sliced index, etc.

**B+ tree vs. Hash-based indexes**

- Extensible hashing, linear hashing, etc.
- Can only handle “=” in join or selection
- Cannot handle inequality predicates