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

• Allocation of TA/UTA for each project has been done
  • see private piazza threads
  • use it for all communications
  • receive comments on the milestone report soon
  • keep working on the projects
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);

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>24</th>
<th>28</th>
<th>36</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart (padded with space)</td>
<td>10</td>
<td>0.9</td>
<td></td>
</tr>
</tbody>
</table>
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.

```
header  record1  record2  ....  free space
```
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?)

```
0  4  8  16
142 10  0.9 Bart\0 Weird kid!\0
```

• Approach 2: use an offset array

```
0  4  8  16  18  22  32
142 10  0.9 Bart Weird kid!
```

why no pointer to Bart?

• Pros/cons of approach 2?
  • (-) Update is messy if it changes the length of a field – may not fit in the block
  • (+) direct access to i-th field, efficient storage of nulls
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
PAX

- Most queries only access a few columns
- Cluster values of the same columns in each block
  - When a particular column of a row is brought into the cache, the same column of the next row is brought in together

Reorganize after every update (for variable-length records only) and delete to keep fields together

(IS NOT NULL bitmap)
Beyond block layout: column stores

• The other extreme: store tables by columns instead of rows

• PAX affects data layout within a single page
  • e.g. one relation can store NSM, other PAX
  • or first do vertical partitioning, then use PAX for storing

• Advantages (and disadvantages) of PAX are magnified
  • Not only better cache performance, but also fewer I/O’s for queries involving many rows but few columns
  • Aggressive compression to further reduce I/O’s

• More disruptive changes to the DBMS architecture are required than PAX
  • Not only storage, but also query execution and optimization
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

• Column store: NSM transposed, beyond blocks
Index
What are indexes for?

• Given a value, locate the record(s) with this value
  
  \[
  \text{SELECT} \ast \text{ FROM } R \text{ WHERE } A = \text{value}; \\
  \text{SELECT} \ast \text{ FROM } R, S \text{ WHERE } R.A = S.B;
  \]

• Find data by other search criteria, e.g.
  
  • Range search
  
  \[
  \text{SELECT} \ast \text{ FROM } R \text{ WHERE } A > \text{value};
  \]
  
  • Keyword search

Focus of this lecture
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

Sparse index on *uid*

<table>
<thead>
<tr>
<th>uid</th>
<th>Name</th>
<th>Score</th>
<th>Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>279</td>
<td>Jessica</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>345</td>
<td>Martin</td>
<td>8</td>
<td>2.3</td>
</tr>
<tr>
<td>456</td>
<td>Ralph</td>
<td>8</td>
<td>0.3</td>
</tr>
<tr>
<td>512</td>
<td>Nelson</td>
<td>10</td>
<td>0.4</td>
</tr>
<tr>
<td>679</td>
<td>Sherri</td>
<td>10</td>
<td>0.6</td>
</tr>
<tr>
<td>697</td>
<td>Terri</td>
<td>10</td>
<td>0.6</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>8</td>
<td>0.7</td>
</tr>
<tr>
<td>912</td>
<td>Windel</td>
<td>8</td>
<td>0.5</td>
</tr>
<tr>
<td>997</td>
<td>Jessica</td>
<td>8</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Dense index on *name*

- Bart
- Jessica
- Lisa
- Martin
- Milhouse
- Nelson
- Ralph
- Sherri
- Terri
- Windel
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
Clustered vs. Unclustered Indexes

• **CREATE INDEX UserPopIndex ON User(pop);**

• What happens if multiple records with the same value of “pop”?

  • Clustered:
    • records with the same pop value are physically stored close to each other
    • access one page, access many records with the same pop

  • Unclustered
    • no such guarantee
    • may need to access a page for each record

• At most one clustered index in each relation
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, usually clustered

• Secondary index
  • Usually dense and unclustered

• 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):
    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

Example: insert 107
Example: delete 129

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

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

Max fan-out: 4
**Sample B⁺-tree nodes**

Max fan-out: 4

Non-leaf

120 | 150 | 180
---|---|---

- to keys
  - 100 ≤ k

Leaf

120 | 130
---|---

- to keys
  - 100 ≤ k < 120
  - 120 ≤ k < 150
  - 150 ≤ k < 180
  - 180 ≤ k

- to next leaf node in sequence
- to records with these k values;
or, store records directly in leaves
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>$\lceil f/2 \rceil$</td>
<td>$\lceil f/2 \rceil - 1$</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>$\lfloor f/2 \rfloor$</td>
<td>$\lfloor f/2 \rfloor$</td>
</tr>
</tbody>
</table>

End of lecture 14
B⁺-tree to be continued in Lecture 15