Physical Data Organization

CompSci 316
Introduction to Database Systems

Announcements (Tue. Nov. 6)

- Homework #3 due today
- Project Milestone #2 due next Thursday
- Homework #4 will also be assigned next Thursday

Outline

- It’s all about disks!
  - That’s why we always draw databases as
  - And why the single most important metric in database processing is (oftentimes) the number of disk I/O’s performed
- Storing data on a disk
  - Record layout
  - Block layout
Storage hierarchy

Registers
Cache
Memory
Disk
Tapes

Why a hierarchy?

How far away is data?

<table>
<thead>
<tr>
<th>Location</th>
<th>Cycles</th>
<th>Location</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>1</td>
<td>On-chip cache</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>On-board cache</td>
<td>10</td>
</tr>
<tr>
<td>Memory</td>
<td>100</td>
<td>Disk</td>
<td>$10^6$</td>
</tr>
<tr>
<td>Disk</td>
<td></td>
<td>Tape</td>
<td>$10^9$</td>
</tr>
</tbody>
</table>

(Source: AlphaSort paper, 1995)

The gap has been widening!

I/O dominates—design your algorithms to reduce I/O!

A typical disk

Tracks
Platter
Platter
Cylinders
Platter

Arm movement
Spindle rotation
"Moving parts" are slow
Top view
“Zoning”: more sectors/data on outer tracks

A block is a logical unit of transfer consisting of one or more sectors

Disk access time

Sum of:
- Seek time: time for disk heads to move to the correct cylinder
- Rotational delay: time for the desired block to rotate under the disk head
- Transfer time: time to read/write data in the block (= time for disk to rotate over the block)

Random disk access

Seek time + rotational delay + transfer time
- Average seek time
  - Time to skip one half of the cylinders?
  - Not quite
  - “Typical” value: 5 ms
- Average rotational delay
  - “Typical” value: 4.2 ms (7200 RPM)
Sequential disk access

Seek time + rotational delay + transfer time

- Seek time
  - 0 (assuming data is on the same track)
- Rotational delay
  - 0 (assuming data is in the next block on the track)
- Easily an order of magnitude faster than random disk access!

Performance tricks

- Disk layout strategy
  - Keep related things (what are they?) close together: same sector/block → same track → same cylinder → adjacent cylinder
- Double buffering
  - While processing the current block in memory, prefetch the next block from disk (overlap I/O with processing)
- Disk scheduling algorithm
  - Example: "elevator" algorithm
- Track buffer
  - Read/write one entire track at a time
- Parallel I/O
  - More disk heads working at the same time

Record layout

Record = row in a table

- Variable-format records
  - Rare in DBMS—table schema dictates the format
  - Relevant for semi-structured data such as XML
- Focus on fixed-format records
  - With fixed-length fields only, or
  - With possible variable-length fields
Fixed-length fields

- All field lengths and offsets are constant
  - Computed from schema, stored in the system catalog
- Example: `CREATE TABLE Student(SID INT, name CHAR(20), age INT, GPA FLOAT);`
  
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>Bart (padded with space)</td>
<td>10</td>
<td>2.3</td>
<td></td>
</tr>
</tbody>
</table>

- Watch out for alignment
  - May need to pad; reorder columns if that helps
- What about NULL?

Variable-length records

- Example: `CREATE TABLE Student (SID INT, name VARCHAR(20), age INT, GPA FLOAT, comment VARCHAR(100));`
- Approach 1: use field delimiters (`\0` okay?)
  
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>142</td>
<td>10</td>
<td>2.3</td>
<td>\0</td>
<td>Bart\0 Weird kid\0</td>
</tr>
</tbody>
</table>

- Approach 2: use an offset array
  
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>18</td>
<td>22</td>
</tr>
<tr>
<td>142</td>
<td>10</td>
<td>2.3</td>
<td>Bart</td>
<td>\0</td>
<td>Weird kid</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Put all variable-length fields at the end (why?)
- Update is messy if it changes the length of a field

LOB fields

- Example: `CREATE TABLE Student(SID INT, name CHAR(20), age INT, GPA FLOAT, picture BLOB(32000));`
- Decomposition (automatically done by DBMS and transparent to the user)
  - `Student(SID, name, age, GPA)`
  - `StudentPicture(SID, picture)`
Block layout

How do you organize records in a block?

- **NSM (N-ary Storage Model)**
  - Most commercial DBMS
- **PAX (Partition Attributes Across)**
  - Ailamaki et al., *VLDB* 2001

---

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

---

Options

- Reorganize after every update/delete to avoid fragmentation (gaps between records)
  - Need to rewrite half of the block on average
- What if records are fixed-length?
  - Reorganize after delete
    - Only need to move one record
    - Need a pointer to the beginning of free space
  - Do not reorganize after update
    - Need a bitmap indicating which slots are in use
Cache behavior of NSM

- Query: SELECT SID FROM Student WHERE GPA > 2.0;
- Assumption: cache line size < record size
- Lots of cache misses
  - SID and GPA are not close enough by memory standards

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>456</td>
<td>Ralph</td>
<td>2.3</td>
</tr>
<tr>
<td>857</td>
<td>Lisa</td>
<td>4.3</td>
</tr>
<tr>
<td>142</td>
<td>Bart</td>
<td>2.3</td>
</tr>
<tr>
<td>123</td>
<td>Milhouse</td>
<td>3.1</td>
</tr>
</tbody>
</table>

**Assumption:** cache line size < record size

---

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

<table>
<thead>
<tr>
<th>Number of records</th>
<th>(for variable-length records only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>123</td>
</tr>
<tr>
<td>857</td>
<td>456</td>
</tr>
</tbody>
</table>

Reorganize after every update and delete to keep fields together

**Reorganize after every update**

---

Beyond block layout: column stores

- The other extreme: store tables by columns instead of rows
- Advantages (and disadvantages) of PAX are magnified
  - Not only better cache performance, but also fewer I/Os for queries involving many rows but few columns
  - Aggressive compression to further reduce I/Os
- 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
  - PAX: a layout that tries to improve cache performance
- Column store: NSM transposed, beyond blocks