Physical Data Organization

Introduction to Databases
CompSci 316 Fall 2019
Announcements (Wed., Oct. 30)

• Homework 3 due Monday

• Project milestone 2 due Wednesday
  • Remember the weekly progress update due today
  • No weekly progress update due next week
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
  • Column stores
Storage hierarchy

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>My head</td>
<td>1 min.</td>
</tr>
<tr>
<td>On-chip cache</td>
<td>2</td>
<td>This room</td>
<td>2 min.</td>
</tr>
<tr>
<td>On-board cache</td>
<td>10</td>
<td>Duke campus</td>
<td>10 min.</td>
</tr>
<tr>
<td>Memory</td>
<td>100</td>
<td>Washington D.C.</td>
<td>1.5 hr.</td>
</tr>
<tr>
<td>Disk</td>
<td>$10^6$</td>
<td>Pluto</td>
<td>2 yr.</td>
</tr>
<tr>
<td>Tape</td>
<td>$10^9$</td>
<td>Andromeda</td>
<td>2000 yr.</td>
</tr>
</tbody>
</table>

(Source: AlphaSort paper, 1995)

The gap has been widening!

 занят

I/O dominates—design your algorithms to reduce I/O!
Latency Numbers
Every Programmer Should Know

Latency Comparison Numbers
------------------------
<table>
<thead>
<tr>
<th>Latency Event</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5 ns</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5 ns</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 ns</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25 ns</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 ns</td>
</tr>
</tbody>
</table>

- 14x L1 cache
- 20x L2 cache, 200x L1 cache

<table>
<thead>
<tr>
<th>Latency Event</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>3,000 ns</td>
</tr>
<tr>
<td>Send 1K bytes over 1 Gbps network</td>
<td>10,000 ns</td>
</tr>
<tr>
<td>Read 4K randomly from SSD*</td>
<td>150,000 ns</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000 ns</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000 ns</td>
</tr>
</tbody>
</table>

- ~1GB/sec SSD
- ~1GB/sec SSD, 4X memory
- 20x datacenter roundtrip

<table>
<thead>
<tr>
<th>Latency Event</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk seek</td>
<td>10,000,000 ns</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>20,000,000 ns</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000 ns</td>
</tr>
</tbody>
</table>

- 20x SSD
- 80x memory, 20X SSD

Notes
-----
1 ns = 10^-9 seconds
1 us = 10^-6 seconds = 1,000 ns
1 ms = 10^-3 seconds = 1,000 us = 1,000,000 ns

Credit
-----
By Jeff Dean: http://research.google.com/people/jeff/
Originally by Peter Norvig: http://norvig.com/21-days.html#answers
A typical hard drive

A typical hard drive

“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; should be time to skip a third of them (why?)
  • “Typical” value: 5 ms

• Average rotational delay
  • Time for a half rotation (a function of RPM)
  • “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!
What about SSD (solid-state drives)?
What about SSD (solid-state drives)?

• No mechanical parts
• Mostly flash-based nowadays
• 1-2 orders of magnitude faster random access than hard drives (under 0.1ms vs. several ms)
  • But still much slower than memory (~0.1μs)
• Little difference between random vs. sequential read performance
• Random writes still hurt
  • In-place update would require erasing the whole “erasure block” and rewriting it!
Important consequences

• 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 generally cheaper than random I/O
Performance tricks

• Disk layout strategy
  • Keep related things (what are they?) close together: same sector/block → same track → same cylinder → adjacent cylinder

• Prefetching
  • While processing the current block in memory, fetch the next block from disk (overlap I/O with processing)

• Parallel I/O
  • More disk heads working at the same time

• Disk scheduling algorithm
  • Example: “elevator” algorithm

• Track buffer
  • Read/write one entire track at a 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 User(uid INT, name CHAR(20), age INT, pop 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>24</td>
<td>36</td>
</tr>
<tr>
<td>142</td>
<td>Bart (padded with space)</td>
<td>10</td>
<td>0.9</td>
</tr>
</tbody>
</table>

• Watch out for alignment
  • May need to pad; reorder columns if that helps
• What about NULL?
  • Add a bitmap at the beginning of the record
Variable-length records

• Example: `CREATE TABLE User(uid INT, name VARCHAR(20), age INT, pop FLOAT, comment VARCHAR(100));`

• Approach 1: use field delimiters (‘\0’ okay?)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>142</td>
<td>10</td>
<td>0.9</td>
<td>Bart\0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Weird kid!\0</td>
</tr>
</tbody>
</table>

• Approach 2: use an offset array

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>18</th>
<th>22</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>142</td>
<td>10</td>
<td>0.9</td>
<td></td>
<td></td>
<td>Bart</td>
<td>Weird kid!</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 User(uid INT, name CHAR(20), age INT, pop FLOAT, picture BLOB(32000));

• User records get “de-clustered”
  • Bad because most queries do not involve picture

• Decomposition (automatically and internally done by DBMS without affecting the user)
  • (uid, name, age, pop)
  • (uid, picture)
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

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
  - uid and pop are not close enough by memory standards

```
142  Bart    10  0.9  123 Milhouse 10  0.2
857  Lisa    8   0.7
456  Ralph   8   0.3
```

```
142  Bart  10
0.9  123 Milhouse
10  0.2  857  Lisa
8   0.7
456  Ralph  8
0.3
```
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

• 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
Example: Apache Parquet

• A table is horizontally partitioned into row groups (~512MB-1GB/row group); each group is stored consecutively
  • On a “block” of HDFS (Hadoop Distributed File System)
• A row group is vertically divided into column chunks, one per column
• Each column chunk is stored in pages (~8KB/page); each page can be compressed/encoded independently

☞ Not designed for in-place updates though!
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 stores: NSM transposed, beyond blocks