CACHE FOR DATABASES

CACHE ISSUES APPLY TO DISK-RESIDENT DATA.

EXCEPT $ = MAIN MEMORY

MEM = DISK

THE BIG PICTURE

DATABASE APPS
WRITTEN IN SQL

SQL INTERPRETER

SEARCH TREE LIBRARY

OPERATING SYSTEM

MEMORY

DISK

SEARCH TREE OR DISK
Disk spins at 5400 RPM (90 Hz)
- 15000 RPM (250 Hz)
Surface coated with
Ferry Magnetic Medium
Iron Oxide or Cobalt Alloy

To read or write data:

1) move head radially to the right track
2) wait for data to rotate under head

- Fluid bearings
- Voice coil motor - head accelerates, coast, then decelerates
- Trends: Disk is > moans law: 67 to 6
Trends to watch:
- Heat up the gas with a laser
- Apply magnetic field

$\Rightarrow$ 67 TB by 2019

40% to 50% per year.

 Probably faster than Moore’s law

$\Rightarrow$ Disks will get cheaper/byte faster than SSD.

What is the right place are?

sector = 512 bytes

track = $\frac{1}{2}$ MB (?)

2 tracks?

One idea: define effective block size $B$

as the size where $\frac{1}{2}$ time is spent

seeking + hold is less

$10 \text{ ms} \approx \frac{B}{100 \text{ MB/s}}$

$10^{-2} \text{ s} = 10^{-3} \frac{\text{ s}}{\text{ ms}} \cdot B$

$10^6 \text{ bytes} = B$
Another idea: Build a data structure that works well for all block sizes.
Search trees on disk

Idea binary tree (2-3 tree, red-black tree, etc.)

- Somehow balanced. E.g. \( \text{depth}(L) \) is within \( 2x \) of \( \text{depth}(R) \)

\[ \Rightarrow \text{Depth is } O(\log N) \]

Search \((V, T)\)

if \( T = \text{null} \) return "not found"

if \( T \) is leaf
    return \( T\).value
else if \( T\).pivot < \( V \)
    return search \((V, T\).right\)
else
    return search \((V, T\).left\)

Note: Tail calls!

Search \((V, T)\)

L: if \( T \) is leaf return \( T\).value
    if \( T\).pivot < \( V \)
        \( T := T\).right
        goto L;
    else
        \( T := T\).left
        goto L;
Analysis:

Worst-case cost for search

\[ = 1 \text{ disk I/O per level in tree} \]

\[ = O(\log N) \text{ disk I/Os.} \]

Improvement: B-tree

disk blocks can hold \( B \) bytes

assume values are constant size,
a disk block can hold \( O(B) \) search tree nodes

Analysis: worst case lookup time

\[ O\left( \frac{\log N}{\log B} \right) = O\left( \frac{\log N}{\log B} \right) \]

Example: block size can hold 4096 values

\[ 4096 \text{ nodes} \rightarrow \log B \approx \log 4096 \approx 12 \]

If \( N \) is a billion rows, then

\[ \log N \approx 30 \]

\[ \frac{\log N}{\log B} > \frac{30}{12} = 3 \]

In practice, \# of disk I/Os = 1, since root is cached.
Inserts are the same cost as search

For binary trees & B trees: $O(\log N) \ O(\log_B N)$

Is that optimal?

Here's a data structure for faster inserts:

![Diagram of a data structure]

appeal to an array

Analysis: Insert $O(1)$ cpu cycles

$O(1/B)$ disk reads

Lookup $O(N/B)$ disk reads.

Can you do fast lookups & inserts?

This problem is what *Tokutek* is solving.
Cache-Oblivious Lookup Table Arrays (cont.)

- \( \log N \) arrays of size \( 2^i \) each
- Each array is completely full or completely empty.
- Each array is sorted.

```plaintext
example

net 10

\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
net 20

\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
net 15 : a couple of choices. Easiest is just net 17 to
```

```plaintext
net 10

\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

\( \begin{array}{c}
15 \\
\end{array} \)

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
15 \\
\end{array} \)
```

```plaintext
\( \begin{array}{c}
10 \\
\end{array} \)
```
Analysis:

Binary search in Gis array:

\[ N = 2^n \text{ elts} \Rightarrow O(\log N) \text{ dim } \text{ rows} \]

Next smaller:

\[ \frac{N}{2} = 2^{n-1} \text{ elts} \Rightarrow O(\log \frac{N}{2}) = O(-1 + \log N) \]

Next smaller

Total cost

\[ \sum_{i=0}^{\log n} -i + \log n = \sum_{i=0}^{\log n} i = O(\log^2 n) \]

Pretty bad!

Idea:

Knowing where key belongs in one array, we can limit scope of search in next to \( O(1) \) elts.

\[ \Rightarrow O(\log N) \text{ still bad } (\text{as } \log \log N) \]
Insert cost: average case

Total work induced by inserting an element

on element moves

cost-of-moving = number-of-moves

number-of-moves = \log N

cost of moving: more than \log N of size N

\begin{center}
\begin{tikzpicture}
\fill[color=black!30!white] (0,0) rectangle (2,1);
\draw (0,0) -- (2,0);
\draw (0,1) -- (2,1);
\end{tikzpicture}
\end{center}

\begin{align*}
\text{reads} & \quad N/B \\
\text{writes} & \quad N/B
\end{align*}

\begin{align*}
2N/B \\
\text{disk I/Os to sort N objects}
\end{align*}

\Rightarrow \frac{4N}{B}

So cost per object = \frac{4N}{B} \cdot \frac{1}{N} = \frac{4}{B} = O(1/B)

total average cost: \frac{\log N}{B} \ll \frac{10}{\log B}
Objects = 100,000

If \( N = 10^{10} \) (100 TB)
\( B = 10^4 \) (objects per block)

\[
\frac{\log N}{\log B} = \frac{12}{4} = 3 \\
\frac{\log N}{6\log B} = \frac{12}{10000} = 0.0012
\]

To make it practical:
- Need to make worst-case better
- Need to handle sub-byte sized objects
- Need transactions (+ crash recovery)
- Need search to be \( O(\log_B N) \), not \( O(\log N) \)
<table>
<thead>
<tr>
<th></th>
<th>Binary</th>
<th>B-tree</th>
<th>COLA</th>
<th>CoLo-aunce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>$\log_2 N$</td>
<td>$\log_2 N / \log_2 B$</td>
<td>$\log_2 N / B$</td>
<td>$\log_2 N / N$</td>
</tr>
<tr>
<td>Insert</td>
<td>$\log_2 N$</td>
<td>$\log_2 N / \log_2 B$</td>
<td>$\log_2 N$</td>
<td>$\log_2 N / \log_2 B$</td>
</tr>
</tbody>
</table>
40 years of trying to make software reliably
do make programming easier, or at best predictable
Projects are still hard, busy.
Here are some approaches that have 'worked' for me:

- Less hyper. What do you do?
  - Write a test
  - Implement the fix
  - Verify that test detects bug + fix.
  - Run test every time you change anything
  - Tests rewritten

Don't: Write a bunch of tests for code that isn't changing.

Do: Write tests for code that changes

Problem: Documentation gets out of date.

For you to update documentation
Tests are a kind of documentation.

Work on making tests understandable.

Defn: A methodology is what you do when you
  don't know how to solve a problem.
OTHER THEMES:

**Build an end-to-end solution.**
- Better to solve the whole problem
  than to have a perfect subsystem.

Perl train is fun.
- We sped up our data struct from
  3000 ins/sec to 200,000 ins/sec
  by engineering effort.

Management is hard.
Money is a demotivator:
- Good people don’t work for money.
  (They may leave because of it,
   but they won’t stay because of it.)

Motivation is often about the work,
everything else can demotivate.

Good people are important.
Conversely, sometimes people need to leave.

**Tools:**
- Valgrind
- Race detectors