CSE 373: Memory hierarchy and B-trees

Michael Lee

Monday, Jan 29, 2018

Warmup

Consider the two programs:

```
// table has size n x m
int sum1(int n,int m,int[][] table) {
    int output = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            output += table[i][j];
        }
    }
    return output;
}
```

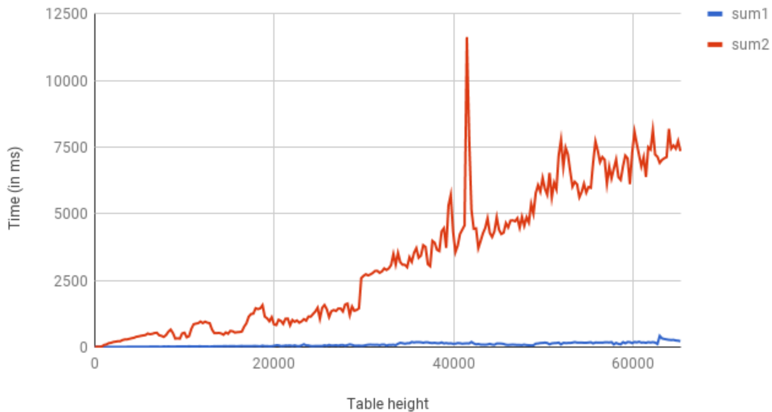
```
// table has size n x m
int sum2(int n,int m,int[][] table) {
    int output = 0;
    for (int j = 0; j < m; j++) {
        for (int i = 0; i < n; i++) {
            output += table[i][j];
        }
    }
    return output;
}
```

With your neighbor:

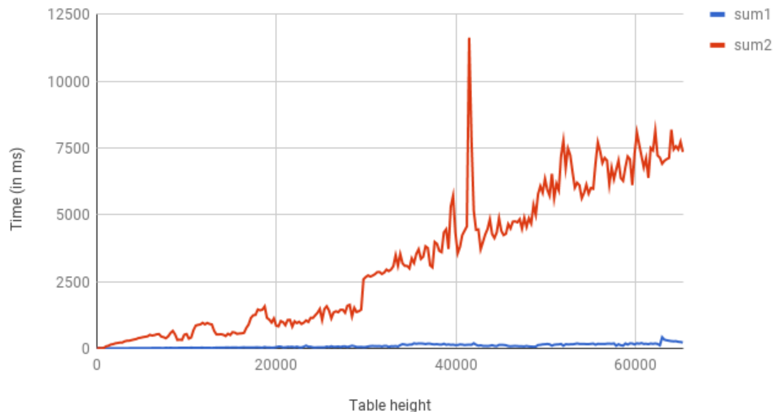
- What do these two methods do?
- What the big-Θ bound on the worst-case runtime of sum1 and sum2 in terms of n and m?

bound: theta(n * m)

Warmup



Warmup



For table of size 65000×4096, sum1 took about 220ms to complete, and sum2 took about 7300 ms! This is about 33 times slower!

Revisiting assumptions

Assumption:

Previously, we've assumed that accessing memory is a quick and constant-time operation.

Revisiting assumptions

Assumption:

Previously, we've assumed that accessing memory is a quick and constant-time operation.

Reality:

Not always true!

Revisiting assumptions

Assumption:

Previously, we've assumed that accessing memory is a quick and constant-time operation.

Reality:

Not always true!

- Sometimes, accessing some memory might be cheaper and easier then others

Revisiting assumptions

Assumption:

Previously, we've assumed that accessing memory is a quick and constant-time operation.

Reality:

Not always true!

- Sometimes, accessing some memory might be cheaper and easier then others
- Sometimes, accessing memory is slow

Revisiting assumptions

Assumption:

Previously, we've assumed that accessing memory is a quick and constant-time operation.

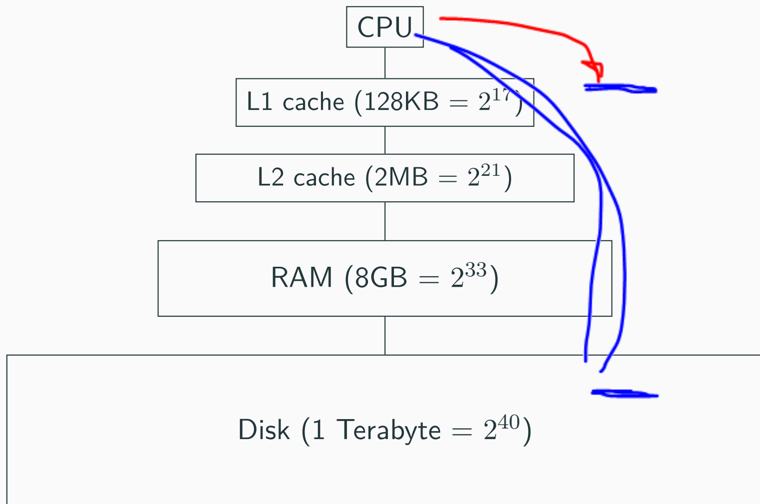
Reality:

Not always true!

- Sometimes, accessing some memory might be cheaper and easier then others
- Sometimes, accessing memory is slow
- When? How can we exploit this? Work around this?

The big picture

A hierarchy of memory. Exact sizes will differ, but one "typical" configuration is:



Times and sizes

Times and sizes

Device	Typical size	Time
CPU register	32 bits $= 2^5$	Basically free
L1 cache	128KB $= 2^{17}$	0.5 nanoseconds
L2 cache	2MB $= 2^{21}$	7 nanoseconds
RAM	8GB $= 2^{33}$	100 nanoseconds
Disk	1TB $= 2^{40}$	8,000,000 nanoseconds

Times and sizes

Device	Typical size	Time
CPU register	32 bits $= 2^5$	Basically free
L1 cache	128KB $= 2^{17}$	0.5 nanoseconds
L2 cache	2MB $= 2^{21}$	7 nanoseconds
RAM	8GB $= 2^{33}$	100 nanoseconds
Disk	1TB $= 2^{40}$	8,000,000 nanoseconds

Comparision:

1 ns vs 8,000,000 ns is like **1 second vs 3 months**

Punchline

It's faster to do...

- 5 million arithmetic ops then 1 disk access
- 2500 L2 cache accesses then 1 disk access
- 400 RAM accesses then 1 disk access

Punchline

It's faster to do...

- 5 million arithmetic ops then 1 disk access
- 2500 L2 cache accesses then 1 disk access
- 400 RAM accesses then 1 disk access

The punchline:

- The more memory something can store, the slower it's likely to be (not a rule, a heuristic)
- Disk access is *very slow*

Punchline

Why are computers built this way?

- Physics (speed of light, physical closeness to CPU)
- Cost
- It's “good enough”: can use “faster layers” to help achieve overall speedup in many cases

Locality

So, how does your operating system try and minimize disk accesses?

Locality

So, how does your operating system try and minimize disk accesses?

Core observation: Most programs have *locality*

Locality

So, how does your operating system try and minimize disk accesses?

Core observation: Most programs have *locality*

- **Spatial locality**
  If you access a chunk of memory, you are very likely to access more memory that's close by.
  (Think arrays, fields)

Locality

So, how does your operating system try and minimize disk accesses?

Core observation: Most programs have *locality*

- **Spatial locality**
  If you access a chunk of memory, you are very likely to access more memory that's close by.
  (Think arrays, fields)
- **Temporal locality**
  If you access memory, you are very likely to access that same piece of memory in the near future.

Locality

So, how does your operating system try and minimize disk accesses?

Core observation: Most programs have *locality*

- **Spatial locality**
  If you access a chunk of memory, you are very likely to access more memory that's close by.
  (Think arrays, fields)
- **Temporal locality**
  If you access memory, you are very likely to access that same piece of memory in the near future.

Locality

So, how does your operating system try and minimize disk accesses?

Core observation: Most programs have *locality*

- **Spatial locality**
  If you access a chunk of memory, you are very likely to access more memory that's close by.
  (Think arrays, fields)
- **Temporal locality**
  If you access memory, you are very likely to access that same piece of memory in the near future.

How can we exploit these assumptions?

Exploiting spatial locality

Exploiting spatial locality

- When looking up an address on a “slow layer”, don’t just bring in those one or two bytes into the “faster layer”: bring in a bunch of surrounding data.

Exploiting spatial locality

Exploiting spatial locality

- When looking up an address on a “slow layer”, don’t just bring in those one or two bytes into the “faster layer”: bring in a bunch of surrounding data.
- Cost of bringing in 1 byte vs several bytes is the same
- Metaphor: if we’re going by car, might as well carpool.

Exploiting temporal locality

Exploiting temporal locality

- Once we load something into RAM/into cache, might as well keep it around for awhile.

Exploiting temporal locality

Exploiting temporal locality

- Once we load something into RAM/into cache, might as well keep it around for awhile.
- Question: faster layers are smaller. When do we *evict* a chunk of memory to make room for newer ones?
  (e.g. if RAM is full, OS may temporarily move stuff in RAM to disk)

Interlude: Caches

General programming and computer science technique:

When accessing something is slow, stick a cache in between.

Interlude: Caches

General programming and computer science technique:

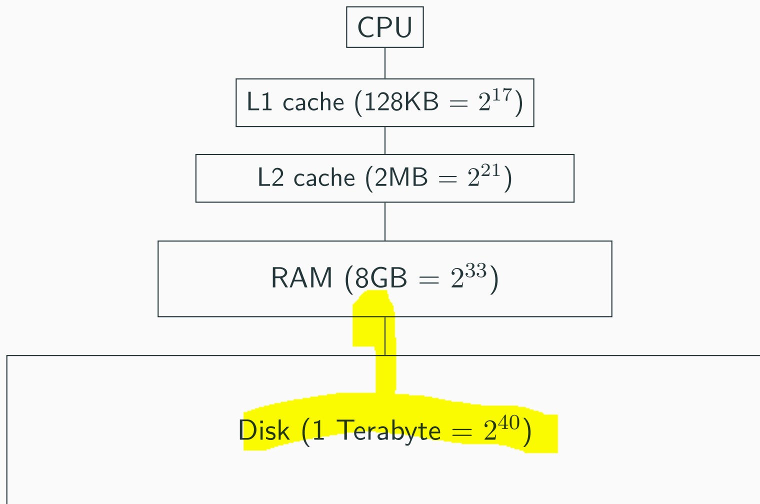
When accessing something is slow, stick a cache in between.

Universal questions:

- How big is the cache?
- How much time do we keep stuff around in the cache?
- How long do we keep stuff around in the cache?

The big picture

Memory in computers: really, a series of multiple caches sitting between the CPU and the disk



Locality and caching

Amount of memory moved from **disk** to **RAM**:

- Called a "**block**" or "**page**"
- Usually, 1 page is 4kb (4096 bytes), but can depend on system
- Interesting fact: smallest unit of data on disk is a page. Can't ask hardware to store half a page.

Locality and caching

Amount of memory moved from **disk** to **RAM**:

- Called a "**block**" or "**page**"
- Usually, 1 page is 4kb (4096 bytes), but can depend on system
- Interesting fact: smallest unit of data on disk is a page. Can't ask hardware to store half a page.

Amount of memory moved from **RAM** to **L1 and L2 cache**:

- Called a "**cache line**"
- Usually, a cache line is 64 bytes, but can depend on system

Locality and caching

Important: None of this is under the programmer's control!

- Page and cache line size is controlled by the system
- Operating system controls when to move things to cache/evict things

The best we can do: write programs that are *aware of* and *work with* what the hardware and OS are likely to do.

Revisiting the warmup

Going back to the warmup:

```
// table has size n x m
int sum1(int n,int m,int[][] table) {
    int output = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            output += table[i][j];
        }
    }
    return output;
}
```

```
// table has size n x m
int sum2(int n,int m,int[][] table) {
    int output = 0;
    for (int j = 0; j < m; j++) {
        for (int i = 0; i < n; i++) {
            output += table[i][j];
        }
    }
    return output;
}
```

Based on what you now know, why is sum1 faster then sum2?

Revisiting the warmup

Answer: `sum1` takes advantage of spatial and temporal locality

What does Java do?

What happens when you use the `new` keyword in Java?

What does Java do?

What happens when you use the new keyword in Java?

- Your program asks the JVM for more memory from the *heap*
- The JVM, if necessary, asks the OS for more memory

What does Java do?

What happens when you use the new keyword in Java?

- Your program asks the JVM for more memory from the *heap*
- The JVM, if necessary, asks the OS for more memory

Important behavior: hardware allocates memory only one page at a time. So, if we want just 100 bytes, we over-allocate to 4kb! JVM and OS try and help mitigate this by trying to use the entire page.

What does Java do?

What happens when you create a new array?

What does Java do?

What happens when you create a new array?

- The program asks the JVM (which may ask the OS) for one, long, mostly **contiguous** chunk of memory that can hold the array.*

What does Java do?

What happens when you create a new array?

- The program asks the JVM (which may ask the OS) for one, long, mostly **contiguous** chunk of memory that can hold the array.*

What happens when you create a new object?

- The program asks the JVM (which may ask the OS) for any random place in memory to store that object.

*The memory isn't guaranteed to be contiguous under-the-hood, but each page's worth of data is guaranteed to be contiguous

What does Java do?

What happens when you read some array index or field?

What does Java do?

What happens when you read some array index or field?

- The program asks the JVM (which may ask the OS) for that address. The OS will first check the L1 cache, the L2 cache, then RAM, then disk to find it
- Once we find the data, the OS loads it into our caches to speed up any future lookups

What does Java do?

What happens when you read some array index or field?

- The program asks the JVM (which may ask the OS) for that address. The OS will first check the L1 cache, the L2 cache, then RAM, then disk to find it
- Once we find the data, the OS loads it into our caches to speed up any future lookups

What happens when we open and read data from a file?

What does Java do?

What happens when you read some array index or field?

- The program asks the JVM (which may ask the OS) for that address. The OS will first check the L1 cache, the L2 cache, then RAM, then disk to find it
- Once we find the data, the OS loads it into our caches to speed up any future lookups

What happens when we open and read data from a file?

- Files are always stored on disk, so we make one or more disk accesses

Question:

Based on what we just learned, why would iterating over an array list be faster then iterating over a linked list?

Question:

Based on what we just learned, why would iterating over an array list be faster then iterating over a linked list?

Answer: linked list nodes could be stored anywhere in memory, which means we don't have much spatial locality. The array list's array is more likely to be stored in contiguous regions of memory, which means that we can take better advantage of the system's tendency to cache nearby addresses.

Motivation

Part 1: How does memory actually work?

Part 2: How can we apply these principles?

Motivation

What we've done so far: study different dictionary implementations

- ArrayDictionary
- SortedArrayDictionary
- Binary search trees
- AVL trees
- Hash tables

Motivation

What we've done so far: study different dictionary implementations

- ArrayDictionary
- SortedArrayDictionary
- Binary search trees
- AVL trees
- Hash tables

They all make one common assumption: *all our data is stored in in-memory, on RAM*.

Motivation

New challenge: what if our data is too large to store all in RAM? (For example, if we were trying to implement a database?)

How can we do this efficiently?

Motivation

New challenge: what if our data is too large to store all in RAM? (For example, if we were trying to implement a database?)

How can we do this efficiently?

Two techniques:

- A tree-based technique
  Excels for range-lookups (e.g. “find all users with an age between 20 and 30”, where “age” is the key)
- A hash-based technique
  Excels for specific key-value pair lookups

A tree-based technique

Idea 1: Use an AVL tree

Suppose the tree has a height of 50. In the best case, how many disk accesses do we need to make? In the worst case?

A tree-based technique

Idea 1: Use an AVL tree

Suppose the tree has a height of 50. In the best case, how many disk accesses do we need to make? In the worst case?

In the best case, the nodes we want happen to be stored in RAM, so we need zero accesses.

In the worst case, each node is stored on a different page on disk, so we need to make 50 accesses.

M-ary search trees

Idea 1:

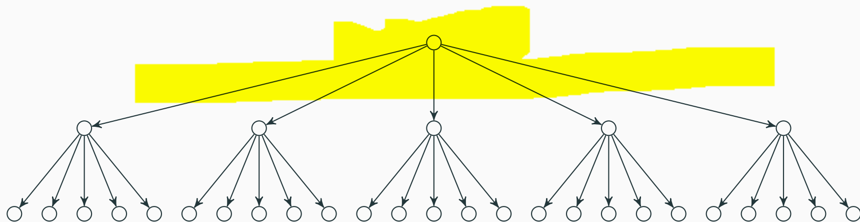
- Instead of having each node have 2 children, make it have M children. Each node contains a sorted array of children nodes.
- Pick M so that each node fits into a single page

M-ary search trees

Idea 1:

- Instead of having each node have 2 children, make it have M children. Each node contains a sorted array of children nodes.
- Pick M so that each node fits into a single page

Example:



M-ary search trees

- What is the **height** of an M-ary search tree in terms of M and n? Assume the tree is balanced.

- What is the worst-case runtime of `get(...)`?

M-ary search trees

- What is the **height** of an M-ary search tree in terms of M and n? Assume the tree is balanced.
  The height is approximately $\log_M(n)$.

- What is the worst-case runtime of `get(...)`?
  We need to examine $\log_M(n)$ nodes.
  Per each node, we need to find the child to pick.
  We can do so using *binary search*: $\log_2(M)$
  Total runtime: $\log_M(n)\log_2(M)$.

Idea 2:

- Do we *really* need to store values on every node? Most of the time, we're just "passing them by"

Idea 2:

- Do we *really* need to store values on every node? Most of the time, we're just "passing them by"
- **Internal nodes** just store keys and points to children
- **Leaf nodes** just store keys and values

Idea 2:

- Do we *really* need to store values on every node? Most of the time, we're just "passing them by"
- **Internal nodes** just store keys and points to children
- **Leaf nodes** just store keys and values