Why am I here?

I have no idea.
Why are you here?

I have 3 theories...
Why are you here?

1. You thought this was the Dreamworks talk.
Why are you here?

2. You’re still drunk from last night.
Why are you here?

3. You can’t manage what you don’t understand.
Overview

1. Hashes & Caches
2. Bloom Filters
3. Distributed Hash Tables (DHTs)
4. Key/Value Stores (NoSQL)
5. Google Bigtable
Disclaimer #1

There will be hand-waving.

The Presence of Slides

!=

“Being Prepared”
Disclaimer #2

You could learn most of this from Wikipedia. Really. Did I mention they’re talking about Shrek in the other room?
Disclaimer #3

My LISA 2008 talk also conflicted with a talk from Dreamworks.
To understand this talk, you must understand:

Hashes
Caches
What is a Hash?

A fixed-size summary of a large amount of data.
Checksum

- Simple checksum:
  - Sum the byte values. Take the last digit of the total.
  - Pros: Easy. Cons: Change order, same checksum.
- Improvement: Cyclic Redundancy Check
  - Detects change in order.
Hash

- “Cryptographically Unique”
- Difficult to generate 2 files with the same MD5 hash
- Even more difficult to make a “valid second file”:
  - The second file is a valid example of the same format. (i.e. both are HTML files)
How do crypto hashes work?

“It works because of math.”
Matt Blaze, Ph.D
Reversible/Irreversible Functions

\[ \quad \quad \quad + \quad 105 \quad = \quad 205 \]

\[ \quad \quad \quad \text{mod} \quad 10 \quad = \quad 4 \]
Some common hashes

<table>
<thead>
<tr>
<th>Hash</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD4</td>
<td>😞</td>
</tr>
<tr>
<td>MD5</td>
<td>😞</td>
</tr>
<tr>
<td>SHA1</td>
<td>😞</td>
</tr>
<tr>
<td>SHA2</td>
<td>😊😊😊</td>
</tr>
<tr>
<td>AES-Hash</td>
<td>😊😊😊😊</td>
</tr>
</tbody>
</table>
Caches
What is a Cache?

- Using a small/expensive/fast thing to make a big/cheap/slow thing faster.
Database

User

Cache

Fast but expensive.

Database

Big, Slow, Cheap

Thursday, November 11, 2010
- Metric used to grade?
  - The “hit rate”: hits / total queries
- How to tune?
  - Add additional storage
  - Smallest increment: Result size.
Suppose cache is X times faster

...but Y times more expensive

Balance cost of cache vs. savings you can get:

- Web cache achieves 30% hit rate, costs $/MB
- 33% of cachable traffic costs $/MB from ISP.

- What about non-cachable traffic?
- What about query size?
- Value of next increment is less than the previous:
  - 10 units of cache achieves 30% hit rate
  - +10 units, hit rate goes to 32%
  - +10 more units, hit rate goes to 33%
Data

Big, Slow, Cheap

User

Cache

Fast but expensive.
Big, Slow, Cheap

NYC

CHI

LAX

Cache

Cache

Cache

Cache

Fast but expensive.

Data

Thursday, November 11, 2010
<table>
<thead>
<tr>
<th></th>
<th>Simple Cache</th>
<th>NCACHE</th>
<th>Intelligent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add new data?</td>
<td>Ok</td>
<td>Not found</td>
<td>Ok</td>
</tr>
<tr>
<td>Delete data?</td>
<td>Stale</td>
<td>Stale</td>
<td>Ok</td>
</tr>
<tr>
<td>Modify data?</td>
<td>Stale</td>
<td>Stale</td>
<td>Ok</td>
</tr>
</tbody>
</table>

Thursday, November 11, 2010
Bloom Filters
What is a Bloom Filter?

- Knowing when NOT to waste time seeking out data.
- Invented in Burton Howard Bloom in 2070
What is a Bloom Filter?

- Knowing when NOT to waste time seeking out data.
- Invented in Burton Howard Bloom in 1970
I invented Bloom Filters when I was 10 years old.
User \rightarrow \text{Bloom} \rightarrow \text{Data}

(Or, precocious 10 year old)
Using the last 3 bits of hash:

Olson 000100001111 000
Polk 0000000000011 001
Smith 001011101110 010
Singh 001000011110 011 ✓

100
101
110 ✓
111 ✓
Using the last 3 bits of hash:

Olson 000100001111
Polk 000000000011
Smith 001011101110
Singh 001000011110
Lakey 111110000000
Baird 001011011111
Camp 001101001010
Johns 010100010100
Burd 111000001101
Bloom 110111000011

37
Using the last 4 bits of hash:

<table>
<thead>
<tr>
<th>Name</th>
<th>Hash</th>
<th>Last 4 Bits</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Olson</td>
<td>0001000001111</td>
<td>1111</td>
<td>1000</td>
</tr>
<tr>
<td>Polk</td>
<td>0000000000011</td>
<td>0001</td>
<td>1001</td>
</tr>
<tr>
<td>Smith</td>
<td>001011101110</td>
<td>1101</td>
<td>1010</td>
</tr>
<tr>
<td>Singh</td>
<td>001000011110</td>
<td>0011</td>
<td>1011</td>
</tr>
<tr>
<td>Lakey</td>
<td>111110000000</td>
<td>0101</td>
<td>1101</td>
</tr>
<tr>
<td>Baird</td>
<td>001011011111</td>
<td>0110</td>
<td>1110</td>
</tr>
<tr>
<td>Camp</td>
<td>001101001010</td>
<td>0111</td>
<td>1111</td>
</tr>
<tr>
<td>Johns</td>
<td>010100010100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Burd</td>
<td>1110000001101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bloom</td>
<td>1101110000111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7/16 = 44%
<table>
<thead>
<tr>
<th>bits of hash</th>
<th># Entries</th>
<th>Bytes</th>
<th>&lt;25% 1’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>$2^3$</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>$2^4$</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>$2^5$</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>$2^6$</td>
<td>64</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>$2^7$</td>
<td>128</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>$2^8$</td>
<td>256</td>
<td>32</td>
</tr>
<tr>
<td>20</td>
<td>$2^8$</td>
<td>1048576</td>
<td>131072</td>
</tr>
<tr>
<td>24</td>
<td>$2^{32}$</td>
<td>16777216</td>
<td>2M</td>
</tr>
<tr>
<td>32</td>
<td>$2^{64}$</td>
<td>4294967296</td>
<td>512M</td>
</tr>
</tbody>
</table>

20 bits of hash can store up to 1 billion entries.

24 bits of hash can store up to 4.1 million entries.

32 bits of hash can store up to 512 million entries.

<25% 1’s

3 bits: 2
4 bits: 4
5 bits: 8
6 bits: 16
7 bits: 32
8 bits: 64
20 bits: 262144
24 bits: 4.1 million
32 bits: 1 billion
- When to use? Sparse Data
- When to tune: When more than x% are “1”
- Pitfall: To resize, must rescan all keys.

- Minimum Increment doubles memory usage:
  - Each increment is MORE USEFUL than the previous.
  - But exponentially MORE EXPENSIVE!
Bloom Filter sample uses

- Databases: Accelerate lookups of indices.
- Simulations: Often having, big, sparse databases.
- Routers: Speeds up route table lookups.
Distributed Bloom Filters?
What if your Bloom Filter is out of date?

- New data added: BAD. Clients may not see it.
- Data changed: Ok
- Data deleted: Ok, but not as efficient.
How to perform updates?

- Master calculates bitmap once.
- Sends it to all clients
- For a 20-bit table, that’s 130K. Smaller than most GIFs!
- Reasonable for daily, hourly, updates.
$ cd ~/Library/Application\ Support/Google/Chrome
$ ls -lh *Bloom*
-rw-r--r--@ 1 tlim 5000  6.2M Nov 10 15:05 Safe Browsing Bloom
-rw-------@ 1 tlim 5000  1.8M Nov 10 15:05 Safe Browsing Bloom Filter 2
-rw-r--r--@ 1 tlim 5000  0B Nov 10 17:02 Safe Browsing Bloom_new
$
Big Bloom Filters often use 96, 120 or 160 bits!
Bloom Filters
Hash Tables
What is a Hash Table?

- It’s like an array.
- But the index can be anything “hashable”.
Hash tables

- Perl hash:
  - $thing{‘b’} = 123;
  - $thing{‘key2’} = “value2”;
  - print $thing{‘key2’};

- Python Dictionary or “dict”:
  - thing = {};
  - thing[‘b’] = 123
  - thing[‘key2’] = “value2”
  - print thing[‘key2’]
<table>
<thead>
<tr>
<th>Bucket</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>78f825</td>
<td>(“cow”, “moo”)</td>
</tr>
<tr>
<td>92eb5f</td>
<td>(“bee”, “buzz”), (“sheep”, “baah”)</td>
</tr>
</tbody>
</table>
Hash Tables
Distributed Hash Tables (DHTs)
What is a DHT?

A hash table so big you have to spread it over multiple of machines.
Wouldn’t an infinitely large hash table be awesome?
Web server

- `lookup(url) -> page contents`
- ‘index.html’ -> ‘<html><head>...’
- ‘/images/smile.png’ -> 0x4d4d2a...
Virtual Web server

- lookup(vhost/url) -> page contents
- ‘cnn.com/index.html’ -> ‘<html><he...’
- ‘time.com/images/smile.png’ -> 0x4d...
Virtual FTP server

- lookup(host:path/file) -> file contents
- `ftp.gnu.org:public/gcc.tgz`
- `ftp.usenix.org:public/usenix.bib`
NFS server

- lookup(host:path/file) -> file contents
  - ‘srv1:home/tlim/Documents/foo.txt’ -> file contents
  - ‘srv2:home/tlim/TODO.txt‘ -> file contents
Usenet (remember usenet?)

- lookup(group:groupname:artnumber) -> article
- lookup('group:comp.sci.math:987765')
- lookup(id:message-id) -> pointer
- lookup('id:foo-12345@uunet') -> 'group:comp.sci.math:987765'
IMAP

- `lookup('server:user:folder:NNNN')` -> email message
Our DVD Collection

- hash(disc image) -> disc image

- How do I find a particular disk?
  - Keep a lookup table of name -> hash

- Benefit: Two people with the same DVD?
  It only gets stored once.
How would this work?
Load it up!

Root
Host

01001001111011001
0001000101100011
1001110101100111
1100010100100110
0011000000000100
'01...'

Root

Host

1

2

4

01001001111011001
0001000101100011
1001110100110111
1110001010010110
0011000000000100
0110000111101100
0100000001101011
0010111000000001
0011000101111000
0011000101111000
'0...'

Root

Host

1

2

01001001111011001
0001000101100011
1001110100110111
1110001010010110
0011000000000100
0110001111101100
0100000001101011
0010111000000001
0011000101111000

0

4

1

3
Root Host

0100100111011001
0001000101100011
1001110100110111
1110001010010110
0011000000000100
0110000111101100
0100000001101011
0010111000000001
0011000101111000

Thursday, November 11, 2010
Find: 0100100111011001...
Find: 0100110111011...
Each host stores:

- All the data that “leaf” there.
- The list of parent nodes talking to it.
- The list of children it knows about.
Dynamically Adjusting:

- Data hashes in “clumps” making some hosts under-full and some hosts over-full.

- Host running out of storage?
  - Split in two. Give half the data to another node.

- Host running out of bandwidth?
  - Clone data and load-balance.
Real DHTs in action

- Peer 2 Peer file-sharing networks.
- Content Delivery Networks (CDNs like Akamai)
- Cooperative Caches
Distributed Hash Tables (DHTs)
Key/Value Stores
Some common Key/Value Stores

- "NoSQL"
- CouchDB
- MongoDB
- Apache Cassandra
- Terrastore
- Google Bigtable
<table>
<thead>
<tr>
<th>Name</th>
<th>Email</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom Limoncelli</td>
<td><a href="mailto:tlim@google.com">tlim@google.com</a></td>
<td>1515 Main Street</td>
</tr>
<tr>
<td>Mary Smith</td>
<td><a href="mailto:mary@example.com">mary@example.com</a></td>
<td>111 One Street</td>
</tr>
<tr>
<td>Joe Bond</td>
<td><a href="mailto:joe@007.com">joe@007.com</a></td>
<td>7 Seventh St</td>
</tr>
<tr>
<td>Name</td>
<td>Email</td>
<td>Address</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Tom Limoncelli</td>
<td><a href="mailto:tlim@google.com">tlim@google.com</a></td>
<td>1515 Main Street</td>
</tr>
<tr>
<td>Mary Smith</td>
<td><a href="mailto:mary@example.com">mary@example.com</a></td>
<td>111 One Street</td>
</tr>
<tr>
<td>Joe Bond</td>
<td><a href="mailto:joe@007.com">joe@007.com</a></td>
<td>7 Seventh St</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User</th>
<th>Transaction</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom Limoncelli</td>
<td>Deposit</td>
<td>100</td>
</tr>
<tr>
<td>Mary Smith</td>
<td>Deposit</td>
<td>200</td>
</tr>
<tr>
<td>Tom Limoncelli</td>
<td>Withdraw</td>
<td>50</td>
</tr>
</tbody>
</table>

Thursday, November 11, 2010
<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Email</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tom Limoncelli</td>
<td><a href="mailto:tlim@google.com">tlim@google.com</a></td>
<td>1515 Main Street</td>
</tr>
<tr>
<td>2</td>
<td>Mary Smith</td>
<td><a href="mailto:mary@example.com">mary@example.com</a></td>
<td>111 One Street</td>
</tr>
<tr>
<td>3</td>
<td>Joe Bond</td>
<td><a href="mailto:joe@007.com">joe@007.com</a></td>
<td>7 Seventh St</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User Id</th>
<th>Transaction</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Deposit</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>Deposit</td>
<td>200</td>
</tr>
<tr>
<td>1</td>
<td>Withdraw</td>
<td>50</td>
</tr>
</tbody>
</table>

Thursday, November 11, 2010
<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Email</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tom Limoncelli</td>
<td><a href="mailto:tlim@google.com">tlim@google.com</a></td>
<td>1515 Main Street</td>
</tr>
<tr>
<td>2</td>
<td>Mary Bond</td>
<td><a href="mailto:mary@example.com">mary@example.com</a></td>
<td>111 One Street</td>
</tr>
<tr>
<td>3</td>
<td>Joe Bond</td>
<td><a href="mailto:joe@007.com">joe@007.com</a></td>
<td>7 Seventh St</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User Id</th>
<th>Transaction</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Deposit</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>Deposit</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>Withdraw</td>
<td>50</td>
</tr>
</tbody>
</table>
Relational Databases

- 1st Normal Form
- 2nd Normal Form
- 3rd Normal Form
- ACID: Atomicity, Consistency, Isolation, Durability
Key/Value Stores

- Keys
- Values
- BASE: Basically Available, Soft-state, Eventually consistent
Eventually?

- Who cares! This is the web, not payroll!
- Change the address listed in your profile.
- Might not propagate to Europe for 15 minutes.
- Can you fly to Europe in less than 15 minutes?
- And if you could, would you care?
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:tlim@google.com">tlim@google.com</a></td>
<td>BLOB OF DATA</td>
</tr>
<tr>
<td><a href="mailto:mary@example.com">mary@example.com</a></td>
<td>BLOB OF DATA</td>
</tr>
<tr>
<td><a href="mailto:joe@007.com">joe@007.com</a></td>
<td>BLOB OF DATA</td>
</tr>
</tbody>
</table>
### Key/Value example:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:tlim@google.com">tlim@google.com</a></td>
<td>{ 'name': 'Tom Limoncelli', 'address': '1515 Main Street' }</td>
</tr>
<tr>
<td><a href="mailto:mary@example.com">mary@example.com</a></td>
<td>{ 'name': 'Mary Smith', 'address': '111 One Street' }</td>
</tr>
<tr>
<td><a href="mailto:joe@007.com">joe@007.com</a></td>
<td>{ 'name': 'Joe Bond', 'address': '7 Seventh St' }</td>
</tr>
<tr>
<td>Key</td>
<td>Value</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| tlim@google.com | message Person { 
|               |   required string name = 1;                                          |
|               |   optional string address = 2;                                       |
|               |   repeated string phone = 3;                                         |
|               | }                                                                     |
|               | {'name': 'Mary Smith',                                               |
|               | 'address': '111 One Street',                                         |
|               | 'phone': ['201-555-3456', '908-444-1111']                            |
|               | }                                                                     |
| mary@example.com | {                                                               |
|               |   'name': 'Joe Bond',                                               |
|               |   'phone': ['862-555-9876']                                         |
|               | }                                                                     |
| joe@007.com | {                                                               |
|               |                                                                     |
|               | }                                                                     |

Thursday, November 11, 2010
Key/Value Stores
Bigtable

- Google’s very very large database.
- OSDI’06
- Petabytes of data across thousands of commodity servers.
- Web indexing, Google Earth, and Google Finance
- Can be very huge.
- Don’t have to have a value! (i.e. the value is “null”)
- Query by
  - Key
  - Key start/stop range (lexigraphical order)
Long keys are cool.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main St/123/Apt1</td>
<td></td>
</tr>
<tr>
<td>Main St/123/Apt2</td>
<td></td>
</tr>
<tr>
<td>Main St/200</td>
<td>Olson</td>
</tr>
</tbody>
</table>

Query range:  
Start: “Main St/123”  
End: infinity
Bigtable Values

- Values can be huge. Gigabytes.
- Multiple values per key, grouped in “families”:
  - “key:family:family:family:...”
Families

- Within a family:
  - Sub-keys that link to data.
  - Sub-keys are dynamic: no need to pre-define.
  - Sub-keys can be repeated.
Example: Crawl the web

- For every URL:
  - Store the HTML at that location.
  - Store a list of which URLs link to that URL.
  - Store the “anchor text” those sites used.

- `<a href="URL">ANCHOR TEXT</a>`
- http://www.cnn.com
- `<html>.........</html>`

- http://tomontime.com
- `<html>
  - `<p>As you may have read on `<a href="http://www.cnn.com">my favorite news site</a> there is...</p>`
<table>
<thead>
<tr>
<th>Key</th>
<th>contents:</th>
<th>anchor:tomontime.com</th>
<th>anchor:cnnsi.com</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.cnn.www</td>
<td>&lt;html&gt;...</td>
<td>my favorite news site</td>
<td>CNN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>contents:</th>
<th>anchor:everythingsysadmin.com</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.tomontime</td>
<td>&lt;html&gt;...</td>
<td>videos</td>
</tr>
</tbody>
</table>
Each Family has its own...

- Permissions (who can read/write/admin)
- QoS (optimize for speed, storage diversity, etc.)
All updates are timestamped.

Retains at least $n$ recent updates or “never”.

Expired updates are garbage collected “eventually”.

Plus “time”
Further Reading:

- Bigtable: http://research.google.com
- A visual guide to NoSQL: http://blog.nahurst.com/visual-guide-to-nosql-systems
- HashTables, DHTs, everything else
- Wikipedia
Other futuristic topics:

- Stop using “locks”, eliminate all deadlocks:
  - STM: Software Transactional Memory
- Centralized routing: (you’d be surprised)
  - 2 minute overview: www.openflowswitch.org
  - (the 4 minute demo video is MUCH BETTER)
- “Network Coding”: $n^2$ more bandwidth?
- SciAm.com: “Breaking Network Logjams”
How to do a query?
<table>
<thead>
<tr>
<th>KEY</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>bird</td>
<td><code>{ legs=2, horns=0, covering='feathers' }</code></td>
</tr>
<tr>
<td>cat</td>
<td><code>{ legs=4, horns=0, covering='fur' }</code></td>
</tr>
<tr>
<td>dog</td>
<td><code>{ legs=4, horns=0, covering='fur' }</code></td>
</tr>
<tr>
<td>spider</td>
<td><code>{ legs=8, horns=0, covering='hair' }</code></td>
</tr>
<tr>
<td>unicorn</td>
<td><code>{ legs=4, horns=1, covering='hair' }</code></td>
</tr>
</tbody>
</table>
“Which animals have 4 legs?”

- Iterate over entire list
- Open up each blob
- Parse data
- Accumulate list

SLOW!
<table>
<thead>
<tr>
<th>KEY</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>animal:bird</td>
<td>&quot;{ legs=2, horns=0, covering='feathers' }&quot;</td>
</tr>
<tr>
<td>animal:cat</td>
<td>&quot;{ legs=4, horns=0, covering='fur' }&quot;</td>
</tr>
<tr>
<td>animal:dog</td>
<td>&quot;{ legs=4, horns=0, covering='fur' }&quot;</td>
</tr>
<tr>
<td>animal:spider</td>
<td>&quot;{ legs=8, horns=0, covering='hair' }&quot;</td>
</tr>
<tr>
<td>animal:unicorn</td>
<td>&quot;{ legs=4, horns=1, covering='hair' }&quot;</td>
</tr>
</tbody>
</table>

- Itersate:
  - Start: "legs:4"
  - End: "legs:5"

Up to, but not including "end"
legs=4 AND covering=fur

- More indexes + the “zig zag” algorithm.
- More indexed attributes = the slower insertions
- Automatic if you use AppEngine’s storage system