Persistence problem

• How to capture our ending point today?

• Want to recreate an adequate starting point for Wednesday:
  – Record ending slide number
  – Record last topic fully covered
  – At start of lecture: *briefly* review last topic covered
  – Restart any partly-covered topic (don’t try to start in middle)

• This is like what we’ll learn about with databases
This lecture

• Historic context
• Why databases?
• The Bluffer’s Guide to Database Concepts
• Introduction to Mongo DB
Big picture

Reality

change

Model / Abstraction

transaction
"52 1/3 talents of reeds from Kuli Ur-Shashaga (has received) Month of Shunumun (sixth month of the year)."
Cuneiform Tablets

• Mostly recorded commercial transactions
• Medium was used from 34th century BCE to 2nd century CE (nearly 4000 years!)
• Script in use evolved dramatically in that time (see below – same sign!)
Current inventory

• Currently $O(10^6)$ tablets held by museums
• Roughly 1% of those have been translated
  – Resource constrained, not lack of knowledge
Implications

• Writing was invented for databases
• We already have persistent media that last longer than the producing civilization
• Changing database formats is an old problem
• Large amounts of unused data is an old problem
What about now?
Some big databases (2013)

• World Data Centre for Climate
  – 6+ PB total data (PB = $10^{15}$)

• Google
  – 91 million searches per day (1000+/sec avg)

• Sprint
  – 365 million call-record insertions per day (4200+/sec avg)
Bluffdale, Utah

Utah Data Center
NSA data storage

• Estimated near-term 3-12 exabytes ($10^{18}$)
• Claimed long-term 1 yottabyte ($10^{24}$)
  – ~petabyte / human ??
WHY DATABASES?
Who needs databases, anyway?

• We have files, right?
• So what’s the problem?
Challenging Dimensions

• Size of database
  – Big elements
  – Big collections
• Operations/sec
• Distribution of data
• Required uptime
Concurrent access

• Shared valuable data may be used by many parties simultaneously

• Leads to same problems as with shared variables and threads
  – Remember: concurrency bugs are hard to detect

• “I can solve that with file locks”
  – And when you’re all done, you’ll have built a database 😊
Big collections

• Storing lots of stuff
• Only selecting tiny fraction of it
• Different programs using different techniques to select
Hiding data representation

• Allow reorganization of physical data without affecting client programs
  – Replication: multiple copies for performance, persistence, availability
  – Sharding: spreading across more servers for performance
  – Clustering: grouping elements that are accessed together
Tolerating failure

• Don’t let hardware failure corrupt data
• Keep data available despite hardware failure
• Don’t let client-program failure corrupt data
THE BLUFFER’S GUIDE TO DATABASE CONCEPTS
Separation of concerns

• How to *find* entities of interest
• How to *operate on* entities of interest
• How to record state changes
• How to recover from failures
• How to keep system available
<table>
<thead>
<tr>
<th>Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexing / Querying</td>
</tr>
<tr>
<td>Locking / Concurrency Control</td>
</tr>
<tr>
<td>Logging / Recovery</td>
</tr>
</tbody>
</table>

Data Model
Usual starting point

• Relational data model (tables)
• Explicit, relatively-static schema
• SQL for querying
• Per-element locking (table/row)
• Single-site multi-table ACID transactions
• Highly reliable logging to disk

“Middle ground”
  – Simplify to reduce costs but sacrifice performance
  – Pay more to add capabilities
Data Model

• What are the “things” the database stores?
  – Relations (tables)
  – Objects
    • As in programming languages – complex graphs
  – Objects encoded as relations
  – XML documents
  – ...

• Choice has pervasive impact
• Subject of much quasi-religious argument
Layers

<table>
<thead>
<tr>
<th>Indexing / Querying</th>
<th>Data Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locking / Concurrency Control</td>
<td></td>
</tr>
<tr>
<td>Logging / Recovery</td>
<td></td>
</tr>
</tbody>
</table>
Logging

• Like a “to-do” list for persistent changes
• Mediates between fast storage used for operations and large slow storage for persistence
• In ordinary operation, like a pipeline:
  – New persistent state changes added to log
  – Previously logged items applied to persistent storage
  – Periodically clean up old unneeded log
Recovery

• If we rely on the log, we have to be sure we always handle anything “left over”

• Before handling new state changes, process log to get to consistent starting state

• Recovery is tricky because you can have another failure while you’re recovering!
  – Still easier to get cases right than if you didn’t use a log
Distribution

• Client/server database
  – Users are separate from data

• Distributed database
  – Data items are in more than one location
Replication

• Multiple copies of data
  – May improve performance
  – May improve availability

• Requires coordination
  – Prevent divergence of different copies of “same” item
  – Handle merges if divergence does happen
Sharding

• Dividing up a collection
• Dividing up the storage/representation of elements in a collection
• Typically to increase resources – spread collection across servers
• May also be part of geographic distribution
Clustering

• Grouping elements for common storage or access
Layers

Indexing / Querying

Locking / Concurrency Control

Logging / Recovery

Data Model
Locking

• Ensuring controlled access to shared data
• Issues familiar from concurrent programming
  – Uncontrolled access produces “impossible” states
  – Concurrency bugs often hard to reproduce
Concurrency control

• Not all controlled access uses locks
• Can have multiversion schemes – everyone sees their own copy
• Can have optimistic schemes – don’t prevent conflict with locks, detect it and undo
Transactions

• Unifying abstraction for state transition, concurrency control, persistence

• “Start” and “End” of transaction bracket a series of database changes

• All actions in that transaction either succeed together, or there is no effect
  – “ACID” properties (next slide)
ACID

• Atomic
• Consistent
• Isolated
• Durable
Atomic

• “All or nothing”
• No partial execution
  – Partial execution turns into non-execution
  – Partial effects are undone
Consistent

• Transforms from correct state to correct state
  – Is a correct program
• Note: no guarantee about effect of starting in incorrect/inconsistent state!
  – Consistent action not required to \textit{restore} consistency
Isolation

• Concurrent activity invisible
• No effects caused by concurrent interleaving
• All visible states equivalent to doing only one thing at a time (serial ordering)
Durability

• Completed activity stays completed despite failures
ACID vs. BASE

- Atomic
- Consistent
- Isolated
- Durable

- Basically Available
- Soft-state
- Eventually consistent
Layers

- Indexing / Querying
- Locking / Concurrency Control
- Logging / Recovery

Data Model
Schema

- Identify structural invariants of data
  - Like typing in programming language
- Because of size / significance of data, schema changes may be very expensive
  - Schema design is often very important
Querying

• Finding the elements of interest
  – Almost never want to handle whole database
    • Often impractical anyway
  – Relevant fraction typically shrinks as database grows
Indexing

• Data structures that make efficient selection possible
Usual starting point (revisited)

- Relational data model (tables)
- Explicit, relatively-static schema
- Per-element locking (table/row)
- SQL for querying
- Single-site multi-table ACID transactions
- Highly reliable logging to disk

Mongo is different…
INTRODUCTION TO MONGO DB
MongoDB choices

• Data representation: JSON documents (serialized as BSON)
  – Fields of object stored together
  – Can have sub-objects
• Dynamic schema
• Non-SQL query language
• Locking per database (one-writer, multi-reader)
  – No multi-document transactions
• Atomicity only per document
• Limited isolation
  – May see data not yet fully written
  – May see data that subsequently disappears
• Sharding / replication (not relevant to 6.170)
What’s good, what’s bad

• It’s easy to get started
• It’s easy to relate the data model to the JS programming model
  – Similar objects, similar “typing”
• It scales well to large collections
• It’s missing some “high end” capabilities
  – Can’t write complex queries
  – Can’t maintain complex invariants
  – May see odd effects (incomplete isolation)
Demo: Command line Mongo
Command Line Mongo

• Database is itself a server (mongod)
• Mongo command is a simple REPL client
• Mongo can be useful for seeing what’s “really” in the database
• Mongo isn’t fussy about what you ask it to store
Revisit: Replaceable prefix
Role of persistence

• Two successive runs of the server start with same default value
• We might well want the new prefix to persist across invocations
• Use the database to accomplish that
Demo: Persistent replaceable prefix
Persistent replaceable prefix

• Almost too simple: database expects *collections of key-value pairs*

• Easy to extend so that we are dealing with the multi-prefix case
Class Logistics

• P2 is out now (first phase due next Tuesday)
• Monday lecture starts with short quiz on servers and persistent state