CC5212-1
PROCESAMIENTO MASIVO DE DATOS
OTOÑO 2015

Lecture 9: NoSQL I

Aidan Hogan
aidhog@gmail.com

Information Retrieval: Storing Unstructured Information

BIG DATA: STORING STRUCTURED INFORMATION

Relational Databases

Relational Databases: One Size Fits All?

"One Size Fits All": An Idea Whose Time Has Come and Gone

Michael Stonebraker
Computer Science and Artificial Intelligence Laboratory, M.I.T., and StreamBase Systems, Inc.
stonebraker@csail.mit.edu

Uğur Çetintemel
Department of Computer Science Brown University, and StreamBase Systems, Inc.
ugur@cs.brown.edu

Abstract

The last 25 years of commercial DBMS development can be summed up in a single phrase: "One size fits all". This phrase refers to the fact that the traditional DBMS architecture (originally designed and optimized for business data processing) has been used to support many data-centric applications with widely varying characteristics and requirements.

In this paper, we argue that this concept is no longer applicable to the database market, and that the commercial world will fracture into a collection of independent database engines, some of which may be unified by a common front-end parser. We use examples from the stream-processing market and the data-warehouse market to bolster our claims. We also briefly discuss other markets for which the traditional architecture is a poor fit and argue for a critical rethinking of the current factoring of systems services into products.

of multiple code lines causes various practical problems, including:

- *a cost problem*, because maintenance costs increase at least linearly with the number of code lines;
- *a compatibility problem*, because all applications have to run against every code line;
- *a sales problem*, because salespeople get confused about which product to try to sell to a customer; and
- *a marketing problem*, because multiple code lines need to be positioned correctly in the marketplace.

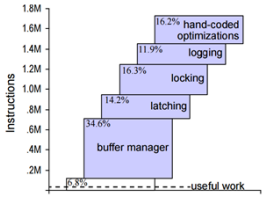
To avoid these problems, all the major DBMS vendors have followed the adage "put all wood behind one arrowhead". In this paper we argue that this strategy has failed already, and will fail more dramatically off into the future.

The rest of the paper is structured as follows. In Section 2, we briefly indicate why the single code-line strategy has failed already by citing some of the key

RDBMS: Performance Overheads

- Structured Query Language (SQL):
  - Declarative Language
  - Lots of Rich Features
  - **Difficult to Optimise!**
- Atomicity, Consistency, Isolation, Durability (ACID):
  - Makes sure your database stays correct
    - Even if there's a lot of traffic!
  - **Transactions incur a lot of overhead**
    - **Multi-phase locks, multi-versioning, write ahead logging**
- Distribution not straightforward

Transactional overhead: the cost of ACID



- 640 tps for system with transactional support
- 12,700 tps for system without logs, transactions or lock scheduling

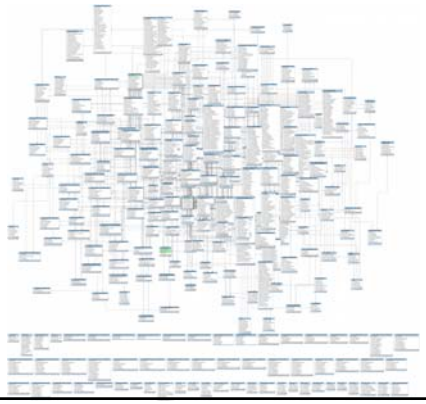

OLTP Through the Looking Glass, and What We Found There

Stavros Harizopoulos, HP Labs, Palo Alto, CA, stavros@hp.com
Daniel J. Abadi, Yale University, New Haven, CT, dna@cs.yale.edu
Samuel Madden, Michael Stonebraker, Massachusetts Institute of Technology, Cambridge, MA, {madden, stonebraker}@csail.mit.edu

ABSTRACT

1. INTRODUCTION

RDBMS: Complexity

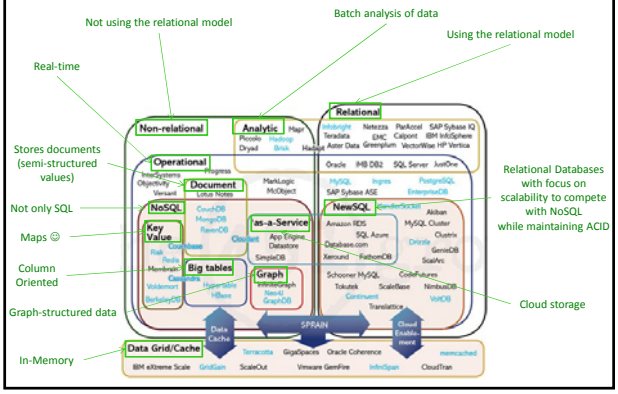


ALTERNATIVES TO RELATIONAL DATABASES FOR QUERYING BIG STRUCTURED DATA?

NoSQL

What do you guys know about NoSQL?

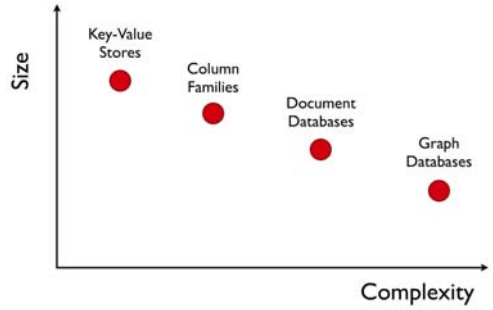
The Database Landscape



Rank	Last Month	DBMS	Database Model	Score	Changes
1.	1.	Oracle	Relational DBMS	1502.74	-11.34
2.	2.	MySQL	Relational DBMS	1309.10	+16.43
3.	3.	Microsoft SQL Server	Relational DBMS	1207.80	-2.63
4.	4.	PostgreSQL	Relational DBMS	240.64	+10.41
5.	5.	MongoDB	Document store	224.62	+10.28
6.	6.	DB2	Relational DBMS	186.47	+1.89
7.	7.	Microsoft Access	Relational DBMS	145.36	+2.60
8.	8.	SQLite	Relational DBMS	89.29	-0.88
9.	9.	Cassandra	Wide column store	81.73	+3.01
10.	10.	Sybase ASE	Relational DBMS	80.00	+1.87
11.	11.	Solr	Search engine	67.16	+4.28
12.	12.	Teradata	Relational DBMS	65.53	+3.80
13.	13.	Redis	Key-value store	62.04	+3.58
14.	14.	FileMaker	Relational DBMS	55.59	+1.21
15.	↑ 16.	HBase	Wide column store	40.27	+3.66
16.	↓ 15.	Informix	Relational DBMS	36.51	-0.20
17.	↑ 19.	Elasticsearch	Search engine	32.06	+2.27
18.	↓ 17.	Hive	Relational DBMS	31.76	+0.74
19.	↓ 18.	Memcached	Key-value store	31.74	+0.76
20.	↑ 21.	Splunk	Search engine	24.68	+2.17
21.	↓ 20.	CouchDB	Document store	22.85	+0.30
22.	22.	Neo4j	Graph DBMS	21.46	+0.91
23.	23.	SAP HANA	Relational DBMS	20.19	+2.17

http://db-engines.com/en/ranking

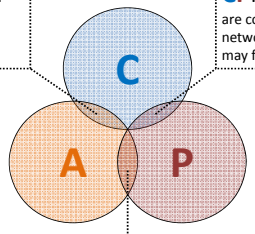
NoSQL



NoSQL: CAP (not ACID)

CA: Guarantees to give a correct response but only while network works fine (*Centralised / Traditional*)

CP: Guarantees responses are correct even if there are network failures, but response may fail (*Weak availability*)



(No intersection)

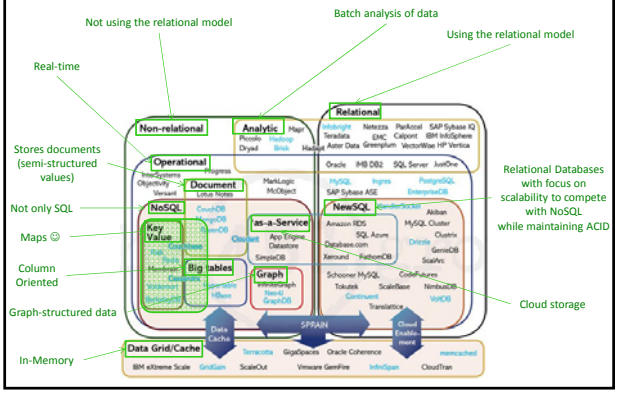
AP: Always provides a "best-effort" response even in presence of network failures (*Eventual consistency*)

NoSQL

- **Distributed!**
  - Sharding: splitting data over servers "horizontally"
  - Replication
- Lower-level than RDBMS/SQL
  - Simpler ad hoc APIs
  - But you build the application (programming not querying)
  - Operations simple and cheap
- Different flavours (for different scenarios)
  - Different CAP emphasis
  - Different scalability profiles
  - Different query functionality
  - Different data models

NOSQL: KEY–VALUE STORE

The Database Landscape



Key–Value Store Model

It's just a Map / Associate Array ☺

- `put(key,value)`
- `get(key)`
- `delete(key)`

Key	Value
Afghanistan	Kabul
Albania	Tirana
Algeria	Algiers
Andorra la Vella	Andorra la Vella
Angola	Luanda
Antigua and Barbuda	St. John's
…	….

But You Can Do a Lot With a Map

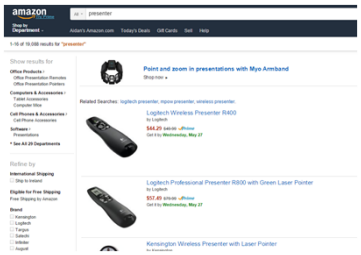
Key	Value
country:Afghanistan	capital@city:Kabul,continent:Asia,pop:31108077#2011
country:Albania	capital@city:Tirana,continent:Europe,pop:3011405#2013
…	…
city:Kabul	country:Afghanistan,pop:3476000#2013
city:Tirana	country:Albania,pop:3011405#2013
…	…
user:10239	basedIn@city:Tirana,post:{103,10430,201}
…	…

… actually you can model any data in a map (but possibly with a lot of redundancy and inefficient lookups if unsorted).

THE CASE OF AMAZON

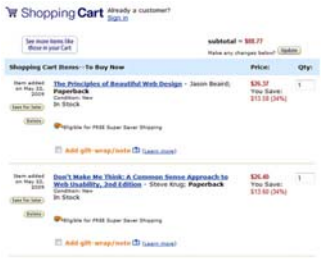
The Amazon Scenario

Products Listings: prices, details, stock



The Amazon Scenario

Customer info: shopping cart, account, etc.

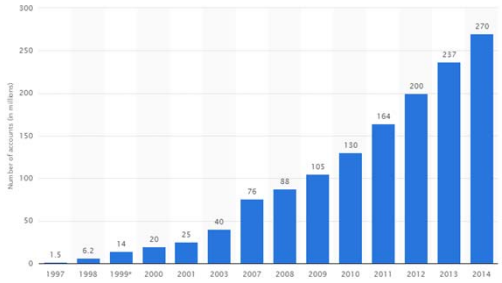


The Amazon Scenario

Recommendations, etc.:

The Amazon Scenario

- Amazon customers:

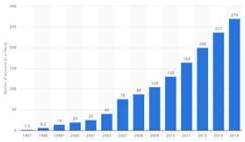


The Amazon Scenario

amazon web services™

The Amazon Scenario

Databases struggling …



But many Amazon services don't need:

- SQL (a simple map often enough)

or even:

- transactions, strong consistency, etc.

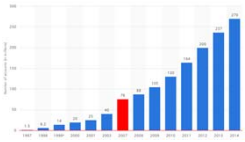
Key–Value Store: Amazon Dynamo(DB)

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

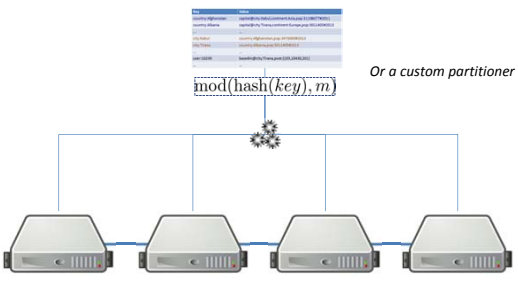


Goals:
Scalability (able to grow)
High availability (reliable)
Performance (fast)

Don't need full SQL, don't need full ACID

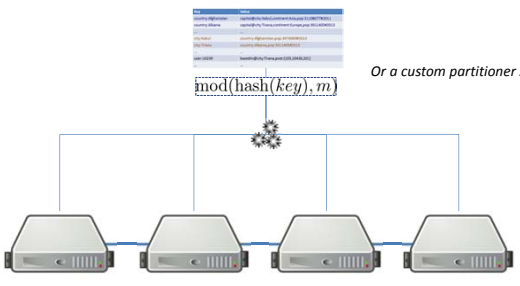
Key–Value Store: Distribution

How might a key–value store be distributed over multiple machines?



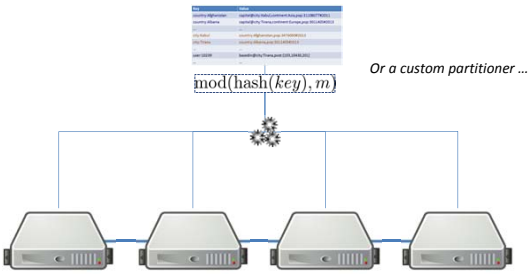
Key–Value Store: Distribution

What happens if a machine joins or leaves half way through?



Key–Value Store: Distribution

How can we solve this?

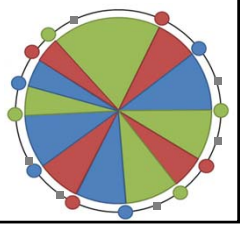


Consistent Hashing

Avoid re-hashing everything

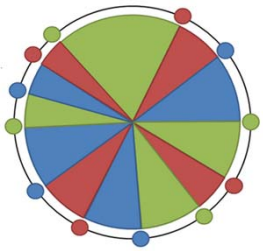
- Hash using a ring
- Each machine picks *n* psuedo-random points on the ring
- Machine responsible for arc after its point
- If a machine leaves, its range moves to previous machine
- If machine joins, it picks new points
- Objects mapped to ring ☺

How many keys (on average) need to be moved if a machine joins or leaves?



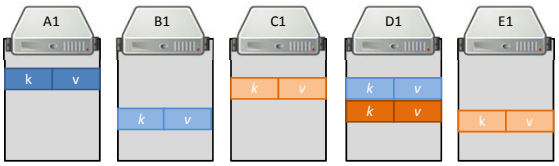
Amazon Dynamo: Hashing

- Consistent Hashing (128-bit MD5)



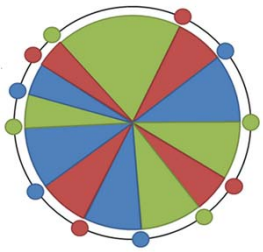
Key–Value Store: Replication

- A set replication factor (here 3)
- Commonly primary / secondary replicas
  - Primary replica elected from secondary replicas in the case of failure of primary



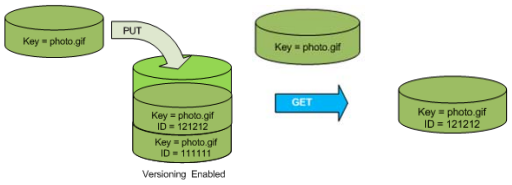
Amazon Dynamo: Replication

- Replication factor of *n*
  - Easy: pick *n* next buckets (different machines!)



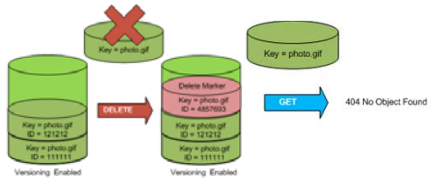
Amazon Dynamo: Object Versioning

- Object Versioning (per bucket)
  - PUT doesn't overwrite: pushes version
  - GET returns most recent version



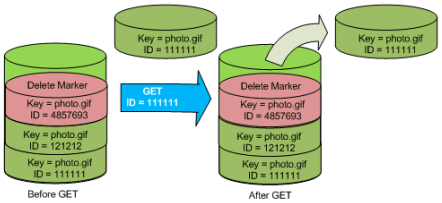
Amazon Dynamo: Object Versioning

- Object Versioning (per bucket)
  - DELETE doesn't wipe
  - GET will return not found



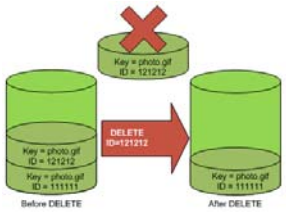
Amazon Dynamo: Object Versioning

- Object Versioning (per bucket)
  - GET by version



Amazon Dynamo: Object Versioning

- Object Versioning (per bucket)
  - PERMANENT DELETE by version … wiped



Amazon Dynamo: Model

- Named table with primary key and a value
- Primary key is hashed / unordered

Countries	
Primary Key	**Value**
Afghanistan	capital:Kabul,continent:Asia,pop:31108077#2011
Albania	capital:Tirana,continent:Europe,pop:3011405#2013
…	…

Cities	
Primary Key	**Value**
Kabul	country:Afghanistan,pop:3476000#2013
Tirana	country:Albania,pop:3011405#2013
…	…

Amazon Dynamo: Model

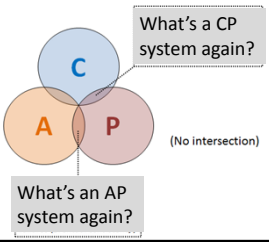
- Dual primary key also available:
  - Hash: unordered
  - Range: ordered

Countries		
Hash Key	**Range Key**	**Value**
Vatican City	839	capital:Vatican City,continent:Europe
Nauru	9945	capital:Yaren,continent:Oceania
…		…

Amazon Dynamo: CAP

Two options for each table:

- **AP**: Eventual consistency, High availability
- **CP**: Strong consistency, Lower availability

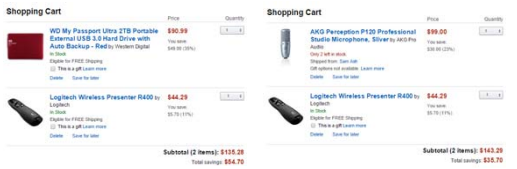


Amazon Dynamo: Consistency

- Gossiping
  - Keep alive messages sent between nodes with state
- Quorums:
  - *N* nodes responsible for a read write
  - Multiple nodes acknowledge read/write for success
  - At the cost of availability!
- Hinted Handoff
  - For transient failures
  - A node "covers" for another node while its down

Amazon Dynamo: Consistency

- Two versions of one shopping cart:

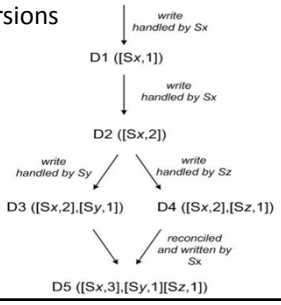


How best to handle multiple conflicting versions of a value (knowing as reconciliation)?

- Application knows best (… and must support multiple versions being returned)

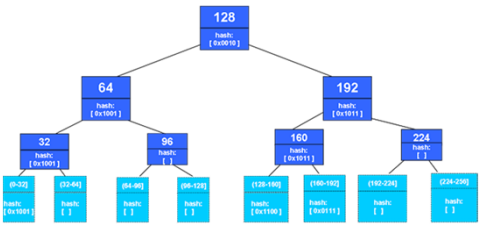
Amazon Dynamo: Vector Clocks

- Vector Clock: A list of pairs indicating a node (i.e., a server) and a time stamp
- Used to track/order versions



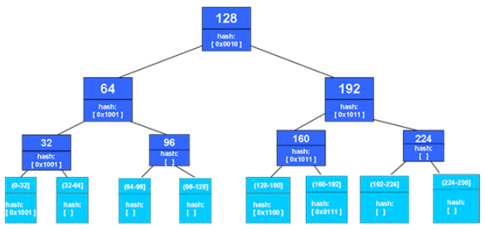
Amazon Dynamo: Eventual Consistency using Merkle Trees

- Merkle tree is a hash tree
- Nodes have hashes of their children
- Leaf node hashes from data: keys or ranges



Amazon Dynamo: Eventual Consistency using Merkle Trees

- Easy to verify regions of the Map
- Can compare level-at-a-time



Amazon Dynamo: Budgeting

- Assign throughput per table: allocate resources
- Reads (4 KB resolution):

Expected Item Size	Consistency	Desired Reads Per Second	Provisioned Throughput Required
4 KB	Strongly consistent	50	50
8 KB	Strongly consistent	50	100
4 KB	Eventually consistent	50	25
8 KB	Eventually consistent	50	50

- Writes (1 KB resolution)

Expected Item Size	Desired Writes Per Second	Provisioned Throughput Required
1 KB	50	50
2 KB	50	100

Read More …

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are

OTHER KEY–VALUE STORES

Other Key–Value Stores

riak

COMCAST Symantec at&t BOEING BEST BUY Ask.com Aol.

Other Key–Value Stores

redis

flickr StackExchange digg github SOCIAL CODING

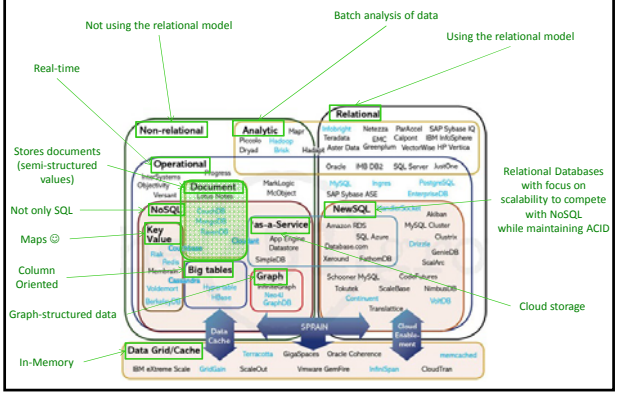
Other Key–Value Stores

cassandra

Adobe Instagram Fast beautiful photo sharing accenture High performance. Delivered. Answers.com DELL Disney CERN NASA

NOSQL: DOCUMENT STORE

The Database Landscape



Key–Value Stores: Values are Documents

Key	Value
country:Afghanistan	<country> <capital>city:Kabul</capital> <continent>Asia</continent> <population> <value>31108077</value> <year>2011</year> </population> </country>
…	…

- Document-type depends on store
  - XML, JSON, Blobs, Natural language
- Operators for documents
  - e.g., filtering, inv. indexing, XML/JSON querying, etc.

MongoDB: JSON Based

Key	Value (Document)
6ads786a5a9	{ "_id" : **ObjectId**("6ads786a5a9") , "name" : "Afghanistan" , "capital": "Kabul" , "continent" : "Asia" , "population" : { "value" : 31108077, "year" : 2011 } }
…	…

- Can invoke Javascript over the JSON objects
- Document fields can be indexed

Document Stores

CouchDB relax

Couchbase

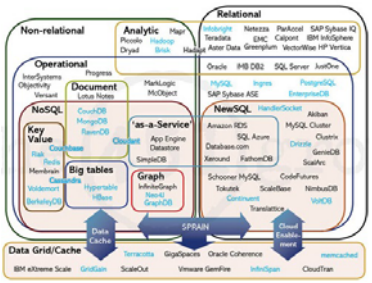
mongoDB

RECAP

Recap

- Relational Databases don't solve everything
  - SQL and ACID add overhead
  - Distribution not so easy
- NoSQL: what if you don't need SQL or ACID?
  - Something simpler
  - Something more scalable
  - Trade efficiency against guarantees

Recap



Recap

- Key–value stores inspired by Amazon Dynamo
  - Distributed maps
  - Hash keys and range keys
  - Table names
  - Consistent hashing
  - Replication
  - Object versioning / vector clocks
  - Gossiping / Quorums / Hinted Hand-offs
  - Merkle trees
  - Budgeting

- Document stores: documents as values
  - Support for JSON, XML values, field indexing, etc.

Questions