CIS 761 – Database System Concepts

Lecture 33

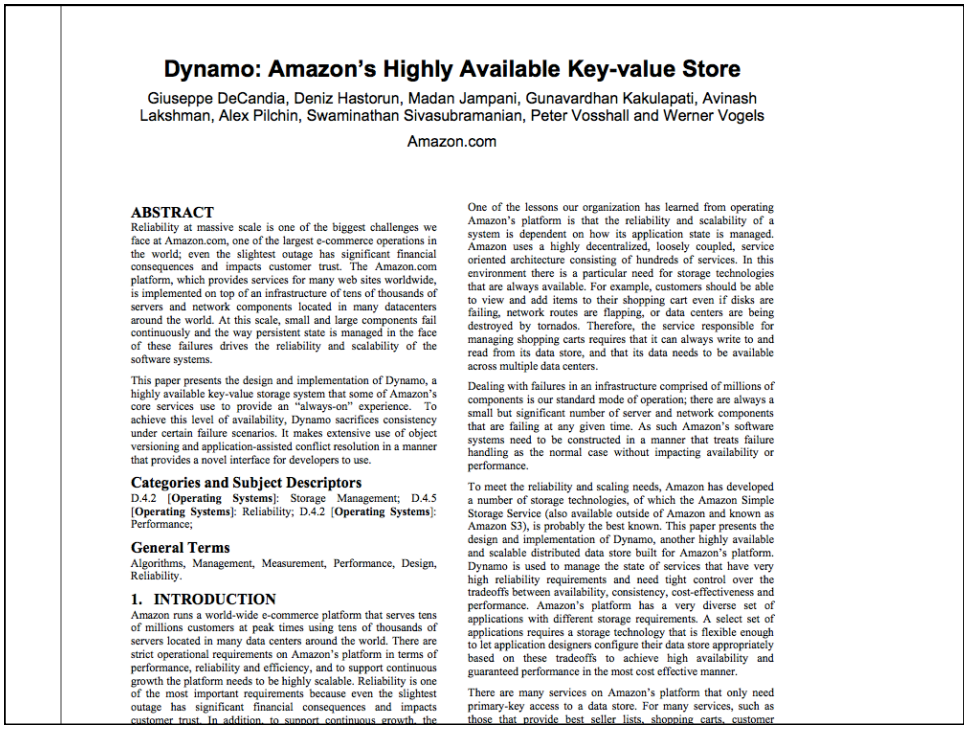
Key-Value & Wide-Column Stores

December 2, 2013

Credits for slides: Shanley, Modak, Mandal, Venutolo.

Where we are

- Last: Hive QL and Pig Latin
- Today:
  - Key-Value Stores
    - Dynamo: Amazon's Highly Available Key-value Store (2007)
  - Wide-Column Stores or Column Families
    - Bigtable: A Distributed Storage System for Structured Data (2006)
- Next:
  - Cassandra
  - MongoDB/CouchDB



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 around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the

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 failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer

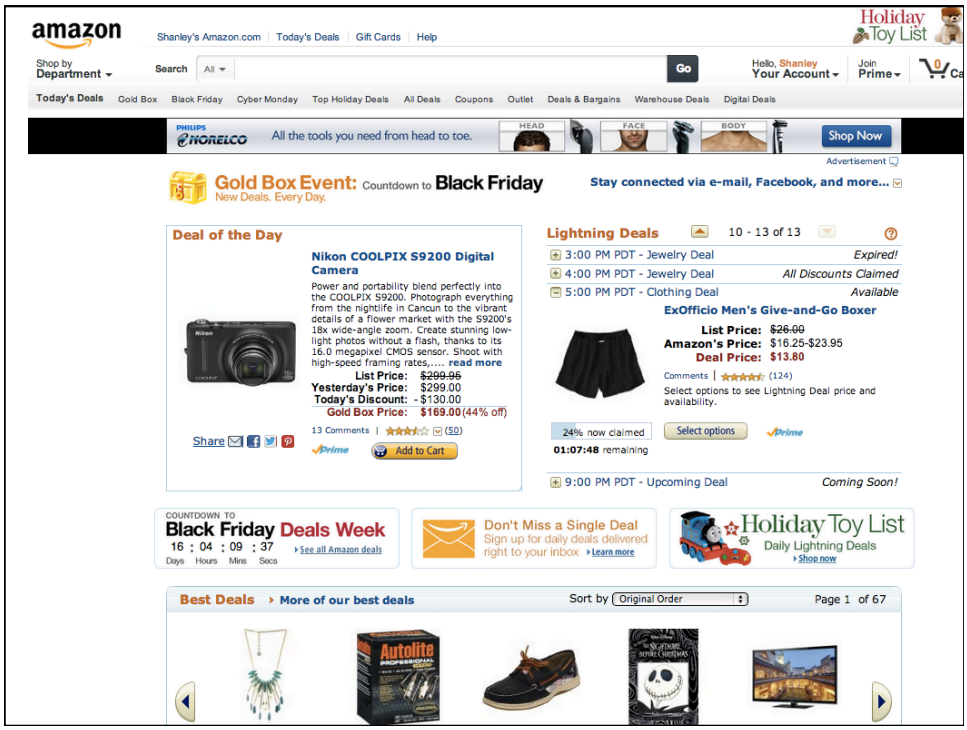
Open-source implementations

- Voldemort
- Dynomite
- KAI
- …

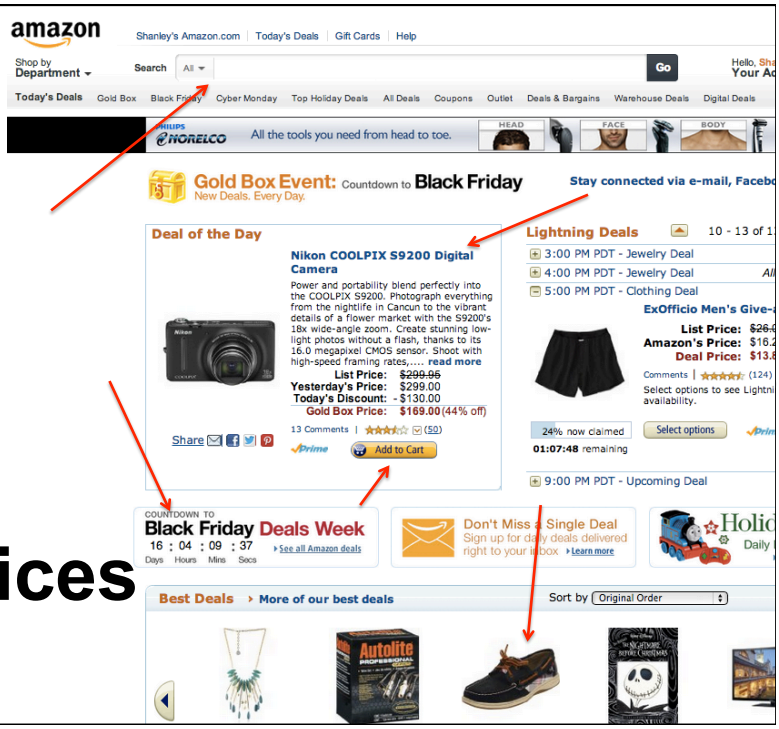
http://nosql-database.org

Motivation

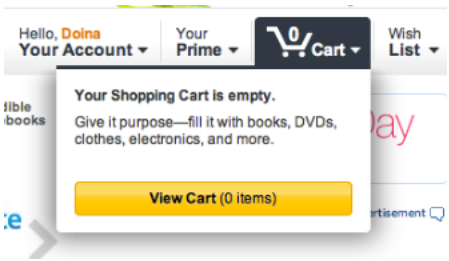
- Even the slightest outage has significant financial consequences and impacts customer trust.
- The platform is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world.
- Persistent state is managed in the face of these failures - drives the reliability and scalability of the software systems



150 Services



Shopping Cart



A canonical use case of an application requiring the high availability and low-latency that Dynamo-based systems provide.

"Customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados."

Other Dynamo Requirements

- Global access
- Multiple machines
- Multiple datacenters
- Scale to peak loads easily
- Tolerance of continuous failure

Traditionally production systems store their state in relational databases. For many of the more common usage patterns of state persistence, however, a relational database is a solution that is far from ideal.

Most of these services only store and retrieve data by primary key and do not require the complex querying and management functionality offered by an RDBMS.

This excess functionality requires expensive hardware and highly skilled personnel for its operation, making it a very inefficient solution.

In addition, the available replication technologies are limited and typically choose consistency over availability.

Although many advances have been made in the recent years, it is still not easy to scale-out databases or use smart partitioning schemes for load balancing.

Dynamo: Amazon's Highly Available Key-value Store

Database design is driven by a virtuous tension between the requirements of the app, the profile of developer productivity, and the limitations of the operational scenario.

requirements of the app

- Stringent latency requirements measured at the 99.9% percentile
- Highly available
- Always writeable
- Modeled as keys/values

the profile of developer productivity

- Choice to manage conflict resolution themselves or manage on the data store level
  - Simple, primary-key only interface
  - No need for relational data model

limitations of the operational scenario

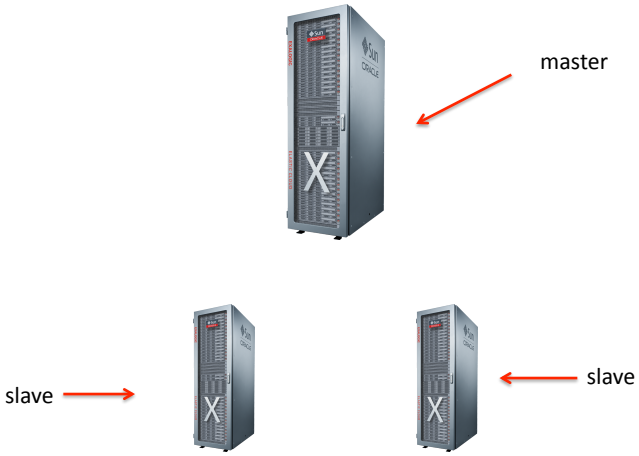
- Functions on commodity hardware
- Each object must be replicated across multiple machines
- Can scale out one node at a time with minimal impact on system and operators

Aspects of the database

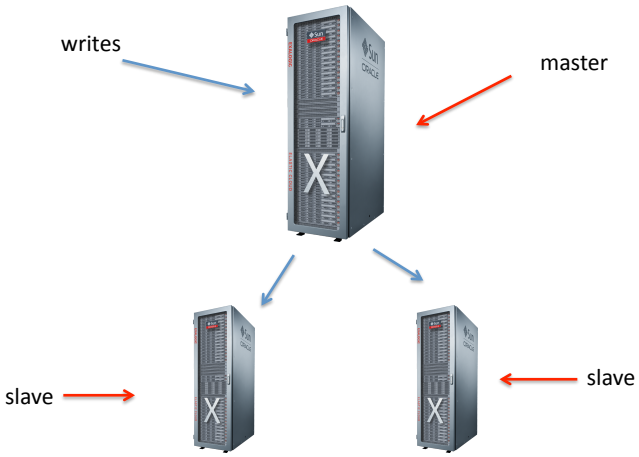
- **How to distribute data around the cluster**
- **Adding new nodes**
- **Replicating data**
- **Resolving data conflicts**
- **Dealing with failure scenarios**
- **Data model**

•Dynamo can really be seen as a collection of technologies and approaches that produce the desired properties of the database

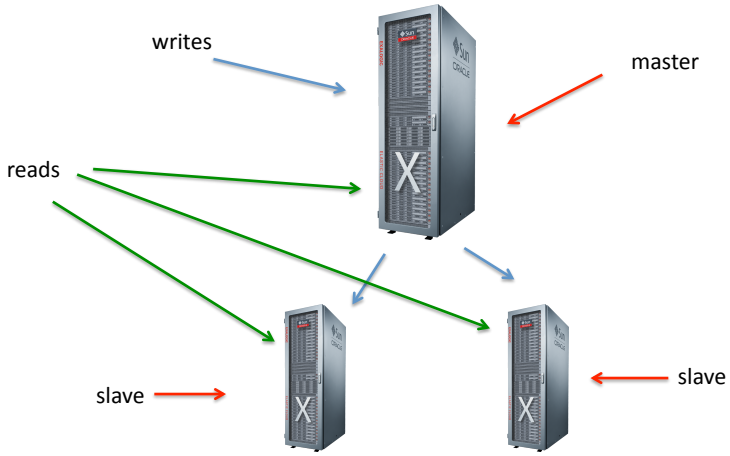
replicating data



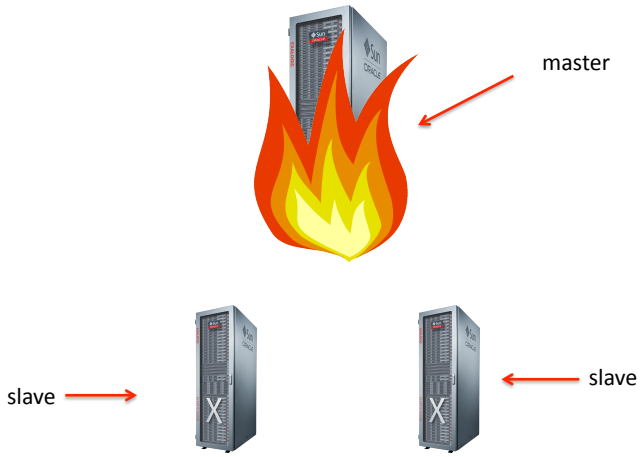
replicating data



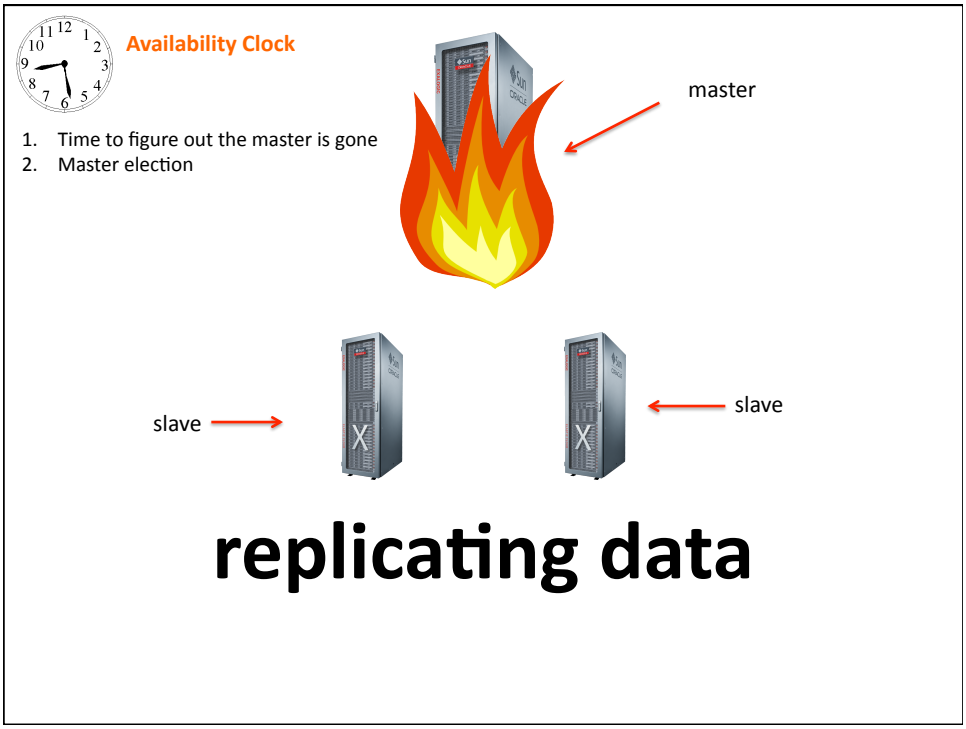
replicating data

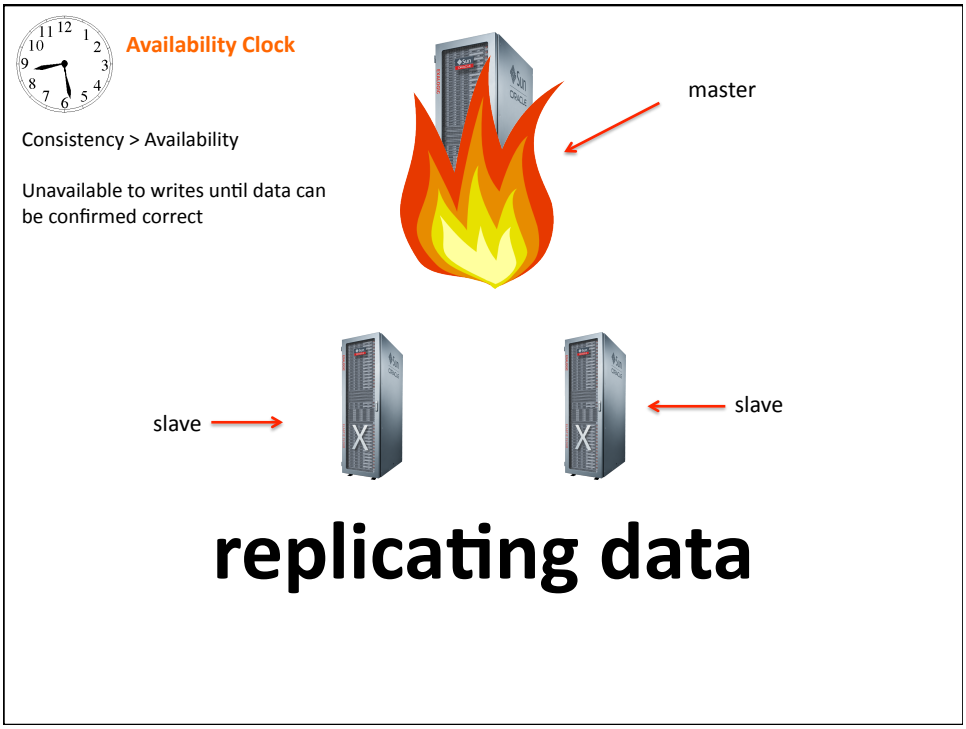


replicating data



replicating data


Availability Clock
1. Time to figure out the master is gone
2. Master election
master
slave
slave
replicating data

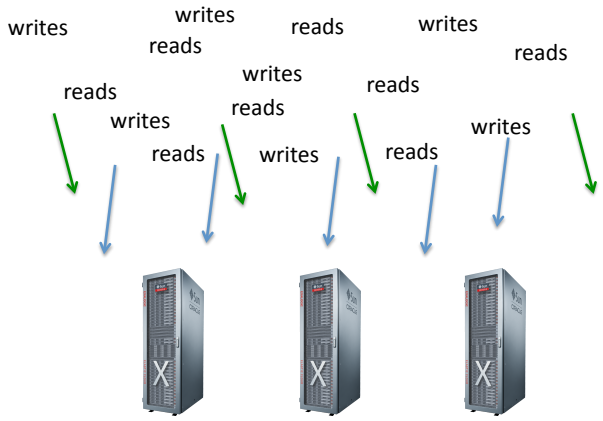

Availability Clock
Consistency > Availability
Unavailable to writes until data can be confirmed correct
master
slave
slave
replicating data

replicating data

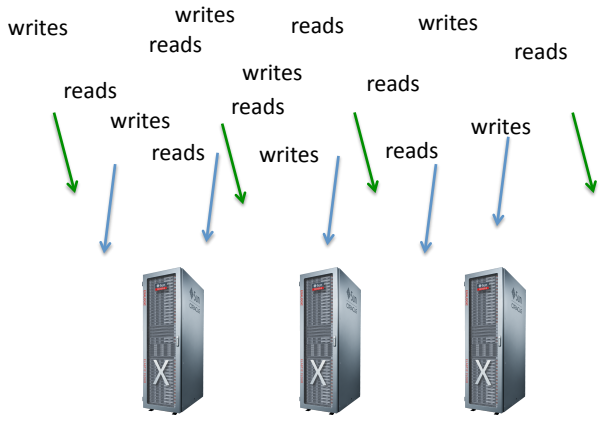
Dynamo paper states that there is no master that can cause write unavailability, and that all nodes are equal.

"Every node in Dynamo should have the same set of responsibilities as its peers; there should be no distinguished node or nodes that take special roles or extra set of responsibilities."

replicating data



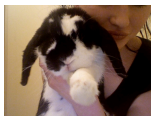
In a Dynamo-based system, all nodes responsible for a replica can serve read and write requests. The system uses eventual consistency – as opposed to all replicas seeing an update at the same time, all replicas will see updates eventually – in practice, usually a very small window until all replicas reflect the most recent value.

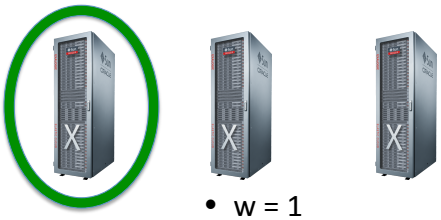


- Clients can read / write to any node
- All updates reach all replicas eventually

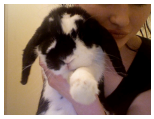
- w and r values: number of replicas that need to participate in a read/ write for a success response

Dynamo systems also provide w and r values on requests to maintain availability despite failure, and to allow the developer to tune to some extent the "correctness" of reads and writes.

put( **)**



- w = 1
- only one node needs to be available to complete write request
- Lower w and r values produce high availability and lower latency.

put()

- w = 3

With a w or r value equal to the number of replicas, higher correctness is possible.

Summary of techniques used in *Dynamo* and their advantages

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

developing apps

"Most of these services only store and retrieve data by primary key and do not require the complex querying and management functionality offered by an RDBMS."

developing apps

•"schema-less" • more flexibility, agility

Lots of unstructured, more apps that don't require a strong schema.

Using a "schema-less" key/value data model, you can eliminate some of the need for extensive data model "pre-planning", and change applications or develop new features without changing the underlying data model.

It's simpler and for some applications that fit a key/value model, more productive.

developing apps

developing apps

app type	key	value
Session	User/Session ID	Session Data
Advertising	Campaign ID	Ad Content

developing apps

app type	key	value
Session	User/Session ID	Session Data
Advertising	Campaign ID	Ad Content
Logs	Date	Log File

developing apps

app type	key	value
Session	User/Session ID	Session Data
Advertising	Campaign ID	Ad Content
Logs	Date	Log File
Sensor	Date, Date/Time	Updates

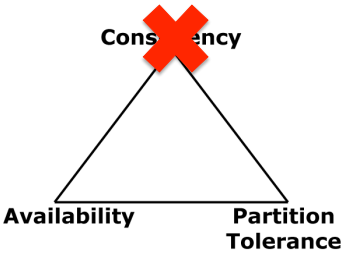
developing apps

app type	key	value
Session	User/Session ID	Session Data
Advertising	Campaign ID	Ad Content
Logs	Date	Log File
Sensor	Date, Date/Time	Updates
User Data	Login, Email, UUID	User Attributes

developing apps

app type	key	value
Session	User/Session ID	Session Data
Advertising	Campaign ID	Ad Content
Logs	Date	Log File
Sensor	Date, Date/Time	Updates
User Data	Login, Email, UUID	User Attributes
Content	Title, Integer, Etc.	Text, JSON, XML

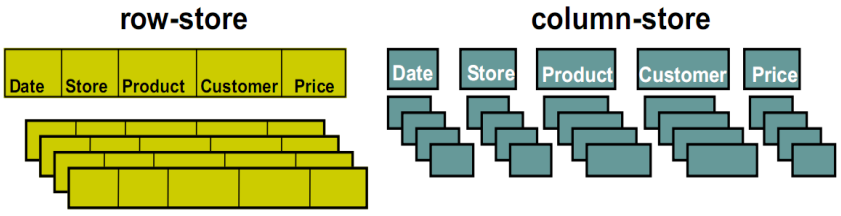
developing apps



Threw out a lot of useful things in the process. What about higher-level data types, search, aggregation tasks, and the developer-friendly things we love in relational databases ? We've done a lot so far to address some of the underlying architectural requirements of new apps and platforms, but there is more we can do to enable greater queriability and broader use cases.

Column stores

Row Store and Column Store



- In row store data are stored in the disk tuple by tuple.
- Whereas in column store data are stored in the disk column by column

Row Store and Column Store

- Most of the queries do not process all the attributes of a particular relation.
- For example the query

  Select c.name and c.address
  From CUSTOMES as c
  Where c.region=Mumbai;

- Only process three attributes of the relation CUSTOMER. But the customer relation can have more than three attributes.
- Column-stores are more I/O efficient for read-only queries, as they read only those attributes which are accessed by a query.

Row Store and Column Store

Row Store	Column Store
(+) Easy to add/modify a record	(+) Only need to read in relevant data
(-) Might read in unnecessary data	(-) Tuple writes require multiple accesses

- So column stores are suitable for read-mostly, read-intensive, large data repositories

Why Column Stores?

- Can be significantly faster than row stores for some applications
  - Fetch only required columns for a query
  - Better cache effects
  - Better compression (similar attribute values within a column)
- But can be slower for other applications
  - OLTP with many row inserts, …
- Long war between the column store and row store camps
  - **Column-Stores vs. Row-Stores: How Different Are They Really?**

BigTable

Introduction

- BigTable is a distributed storage system for managing structured data.
- Designed to scale to a very large size
  - Petabytes of data across thousands of servers
- Used for many Google projects
  - Web indexing, Personalized Search, Google Earth, Google Analytics, Google Finance, Orkut, …
- Flexible, high-performance solution for Google's products

Google Motivation

- Lots of (semi-)structured data at Google
  - URLs:
    - Contents, crawl metadata, links, anchors, pagerank, …
  - Per-user data:
    - User preference settings, recent queries/search results, …
  - Geographic locations:
    - Physical entities (shops, restaurants, etc.), roads, satellite image data, user annotations, …
- Scale is large
  - Billions of URLs, many versions/page (~20K/version)
  - Hundreds of millions of users, thousands or q/sec
  - 100TB+ of satellite image data

Why not just use commercial DB?

- Scale is too large for most commercial databases
- Even if it wasn't, cost would be very high
  - Building a storage system internally means that the system can be applied across many projects for low incremental cost
- Low-level storage optimizations help performance significantly
  - Much harder to do when running on top of a database layer

Goals

- Want asynchronous processes to be continuously updating different pieces of data
  - Want access to most current data at any time
- Need to support:
  - Very high read/write rates (millions of ops per second)
  - Efficient scans over all or interesting subsets of data
  - Efficient joins of large one-to-one and one-to-many datasets
- Often want to examine data changes over time
  - E.g. Contents of a web page over multiple crawls

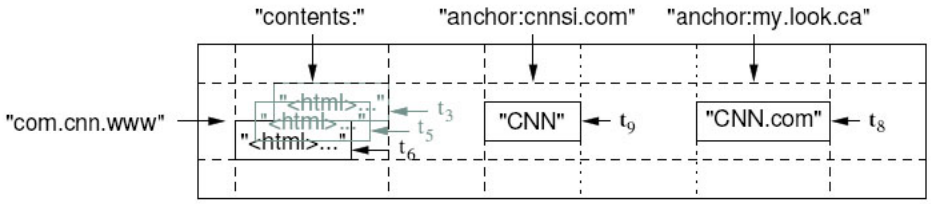
BigTable

- Distributed multi-level map
- Fault-tolerant, persistent
- Scalable
  - Thousands of servers
  - Terabytes of in-memory data
  - Petabyte of disk-based data
  - Millions of reads/writes per second, efficient scans
- Self-managing
  - Servers can be added/removed dynamically
  - Servers adjust to load imbalance

Data model: a big map

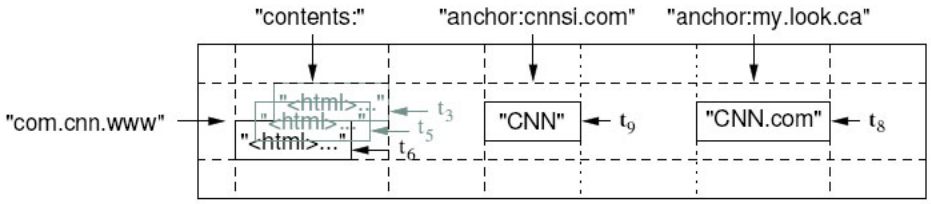
- A BigTable is a sparse, distributed persistent multi-dimensional sorted map
- <Row, Column,Timestamp> triple as key
  - <Row, Column,Timestamp> -> *cell contents*
- API has lookup, insert and delete operations based on the key
- Does not support a relational model
  - No table-wide integrity constraints
  - No multi-row transactions
- Good match for most Google applications

WebTable Example



- Want to keep copy of a large collection of web pages and related information (*Webtable*)
- Use URLs as row keys
- Various aspects of web page as column names
- Store contents of web pages in the `contents:` column under the timestamps when they were fetched.

Rows



- Name is an arbitrary string
  - Access to data in a row is atomic
  - Row creation is implicit upon storing data
- Rows ordered lexicographically
  - Rows close together lexicographically usually on one or a small number of machines

Rows (cont.)

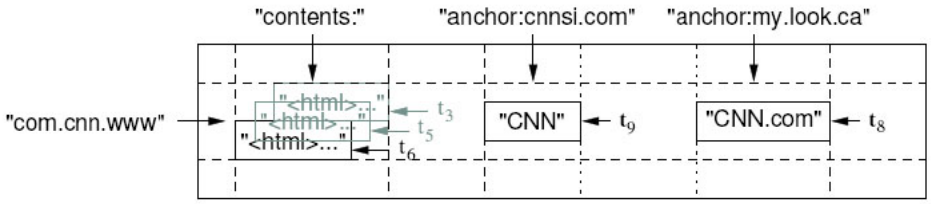
- Reads of short row ranges are efficient and typically require communication with a small number of machines.
- Can exploit this property by selecting row keys so they get good locality for data access.
- Example:

```
math.gatech.edu, math.uga.edu, phys.gatech.edu, phys.uga.edu
VS
edu.gatech.math, edu.gatech.phys, edu.uga.math, edu.uga.phys
```

Tablets

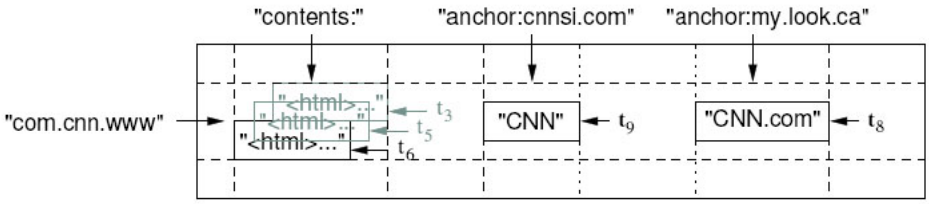
- Large tables broken into *tablets* at row boundaries
  - Tablet holds contiguous range of rows
    - Clients can often choose row keys to achieve locality
  - Aim for ~100MB to 200MB of data per tablet
- Serving machine responsible for ~100 tablets
  - Fast recovery:
    - 100 machines each pick up 1 tablet for failed machine
  - Fine-grained load balancing:
    - Migrate tablets away from overloaded machine
    - Master makes load-balancing decisions

Columns



- Columns have two-level name structure:
  - family:optional_qualifier
- Column family
  - Unit of access control
  - Has associated type information
- Qualifier gives unbounded columns

Timestamps



- Used to store different versions of data in a cell
  - New writes default to current time, but timestamps for writes can also be set explicitly by clients
- Lookup options:
  - *"Return most recent K values"*
  - *"Return all values in timestamp range (or all values)"*
- Column families can be marked w/ attributes:
  - *"Only retain most recent K values in a cell"*
  - *"Keep values until they are older than K seconds"*

APIs

- Metadata operations
  - Create/delete tables, column families, change metadata
- Writes
  - Set(): write cells in a row
  - DeleteCells(): delete cells in a row
  - DeleteRow(): delete all cells in a row
- Reads
  - Scanner: read arbitrary cells in a bigtable
    - Each row read is atomic
    - Can restrict returned rows to a particular range
    - Can ask for just data from 1 row, all rows, etc.
    - Can ask for all columns, just certain column families, or specific columns

API Examples: Write/Modify

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

atomic row modification

API Examples: Read

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

Return sets can be filtered using regular expressions:
anchor: com.cnn.*

Building Blocks

- Building blocks:
  - Google File System (GFS): Raw storage
  - Scheduler: schedules jobs onto machines
  - Lock service: distributed lock manager
  - MapReduce: simplified large-scale data processing
- BigTable uses of building blocks:
  - GFS: stores persistent data (SSTable file format for storage of data)
  - Scheduler: schedules jobs involved in BigTable serving
  - Lock service: master election, location bootstrapping
  - Map Reduce: often used to read/write BigTable data

Typical Cluster