NoSQL systems

Key Value stores

Introduction

- Internet Revolution led to the need for highly scalable distributed storage systems
  - E.g., Google stores Billions of URLs, many versions (~20K/version)
  - Per user data (preference settings, recent queries/search results) for hundreds of million Users, thousands of queries per second 100+TB of satellite image data
  - Same story for Yahoo, Amazon, …
- Such storage had to be highly available anytime from anywhere in the world
  - 24/7 access by 100s of thousands users

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Where should such data be stored?

- Distributed file systems ?
  - Too low level an interface for rapid application development. Takes us back several decades.

- Relational Database Systems?

The “Traditional” RDBMSs offer Some Advantages…

- **Clustered**
  - Traditional Enterprise RDBMS provide the ability to cluster and replicate data over multiple servers – providing reliability
  - Oracle, Microsoft SQL Server and even MySQL have traditionally powered enterprise and online data clouds
- **Highly Available**
  - Provide Synchronization (“Always Consistent”), Load-Balancing and High-Availability features to provide nearly 100% Service Uptime
- **Structured Querying –**
  - Allow for complex data models and structured querying – It is possible to off-load much of data processing and manipulation to the back-end database

But …

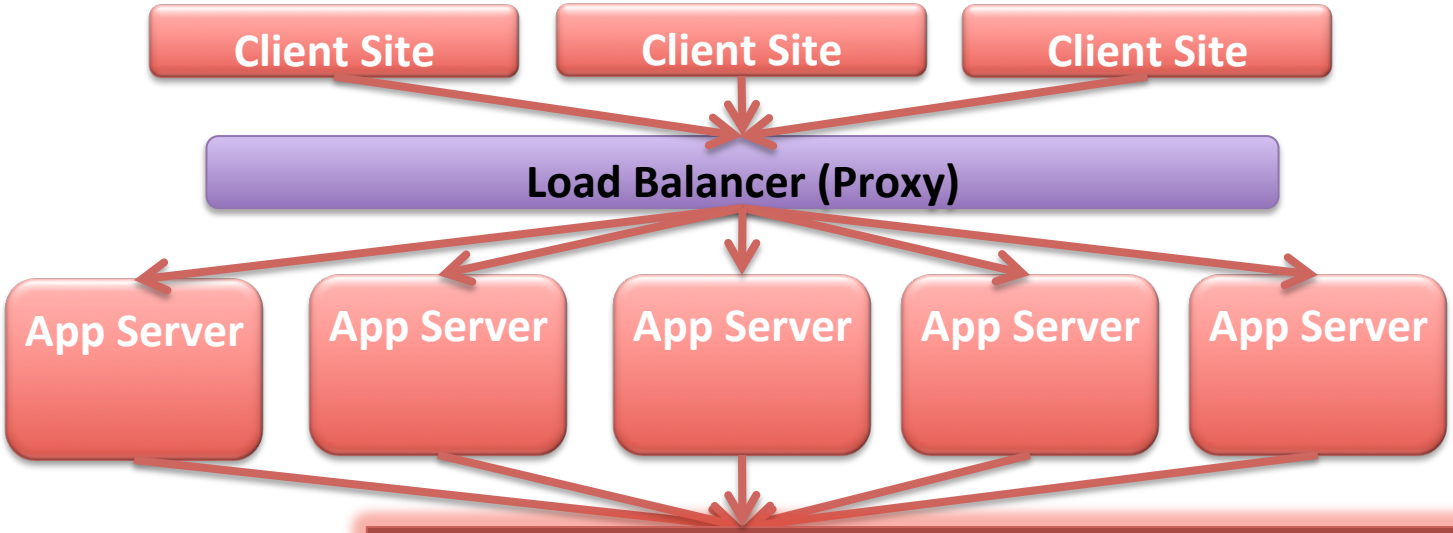
- Classical RDBMS could not scale to these workloads while using commodity hardware.
- What about parallel Database technology?
  - Data is still too large scale!
  - Using a commercial approach would be too expensive
    - Building internally means system can be applied across many projects for low incremental cost
  - Low-level storage optimizations significantly improve performance
    - Difficult to do when running on a DBMS

Blog Wisdom

- "If you want vast, on-demand scalability, you need a non-relational database." Since scalability requirements:
  - Can change very quickly and,
  - Can grow very rapidly.
- Difficult to manage with a single in-house RDBMS server.
- RDBMS scale well:
  - When limited to a single node, but
  - Overwhelming complexity to scale on multiple server nodes.

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Scaling in the Cloud

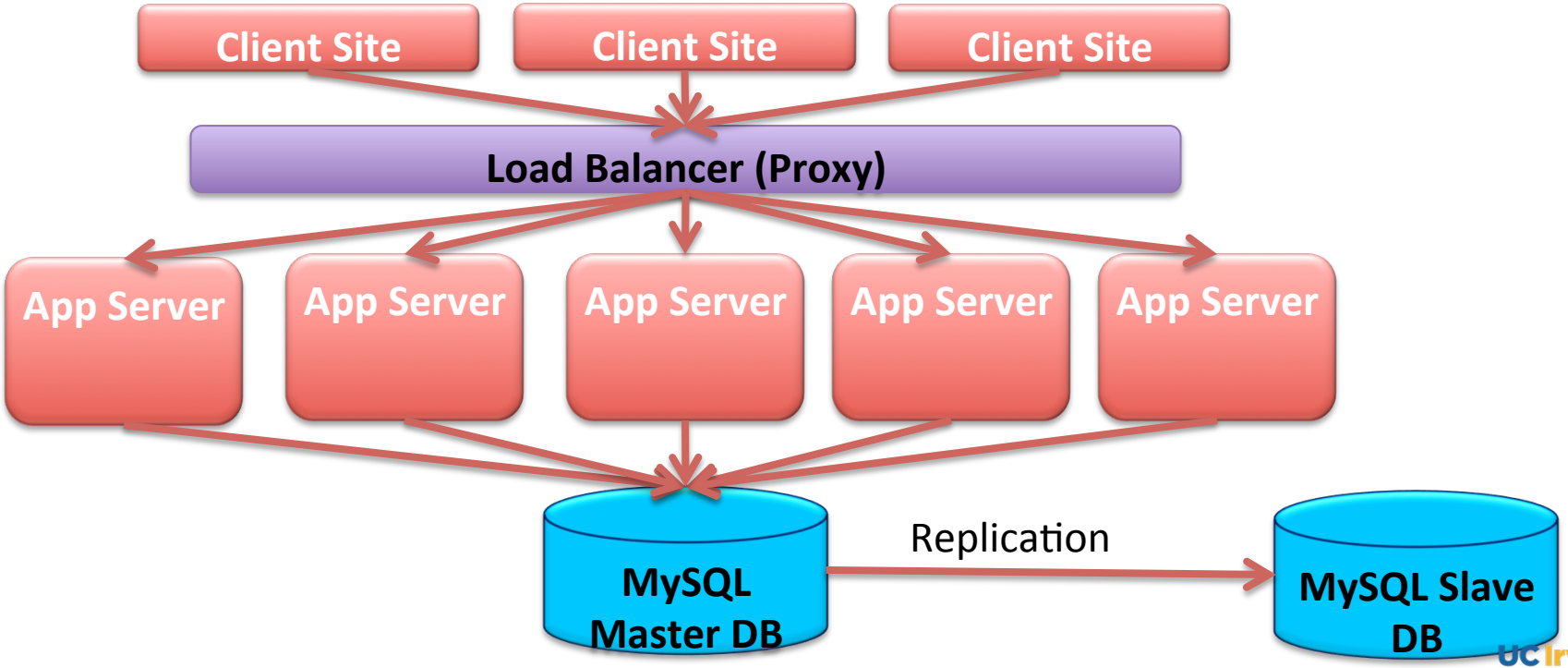


Database becomes the Scalability Bottleneck

Cannot leverage elasticity

Source: EDBT 2011 Tutorial

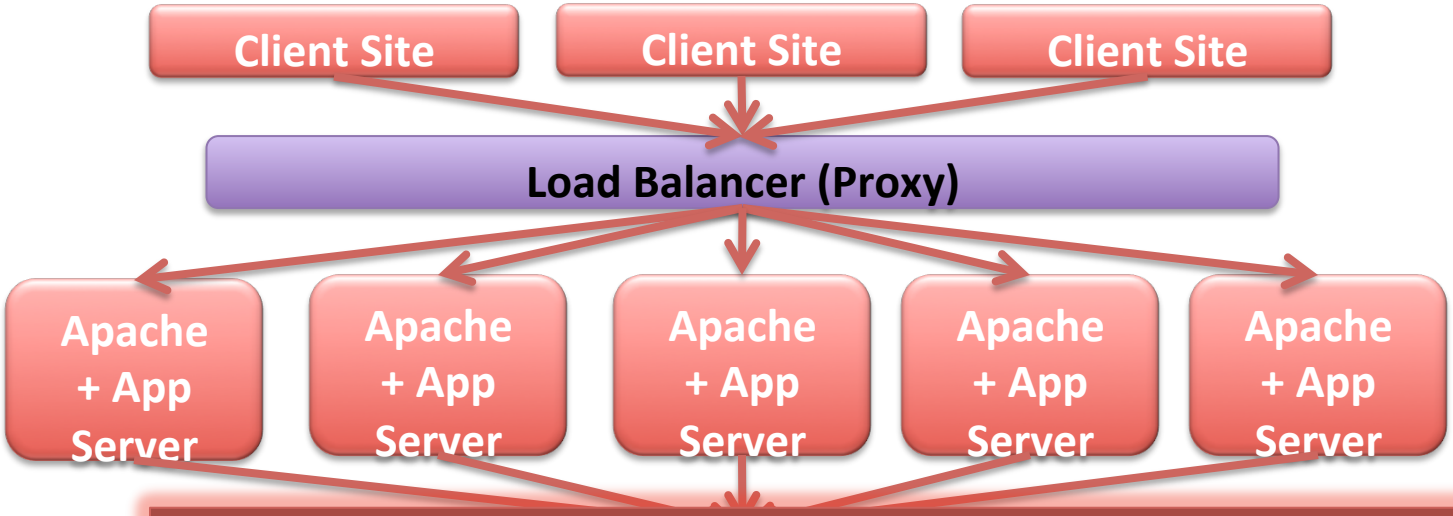
Scaling in the Cloud



Source: EDBT 2011 Tutorial

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Scaling in the Cloud



Scalable and Elastic,
but limited consistency and operational flexibility

Source: EDBT 2011 Tutorial

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Key Value Stores

- Provide a very simple data model. Object consists of <Key, Value> pairs. We can store and retrieve objects based on their key.
  - Key is the unique identifier
  - Key is the granularity for consistent access
  - Value can be structured or unstructured
- Gained widespread popularity
  - In house: **Bigtable** (Google), **PNUTS** (Yahoo!), **Dynamo** (Amazon)
  - Open source: **HBase, Hypertable, Cassandra, Voldemort**
- Popular choice for the modern breed of web-applications

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Design Goals

- **Scale out: designed for scale**
  - Commodity hardware
  - Low latency updates
  - Sustain high update/insert throughput
- **Elasticity – scale up and down with load**
- **High availability – downtime implies lost revenue**
  - Replication (with multi-mastering)
  - Geographic replication
  - Automated failure recovery

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

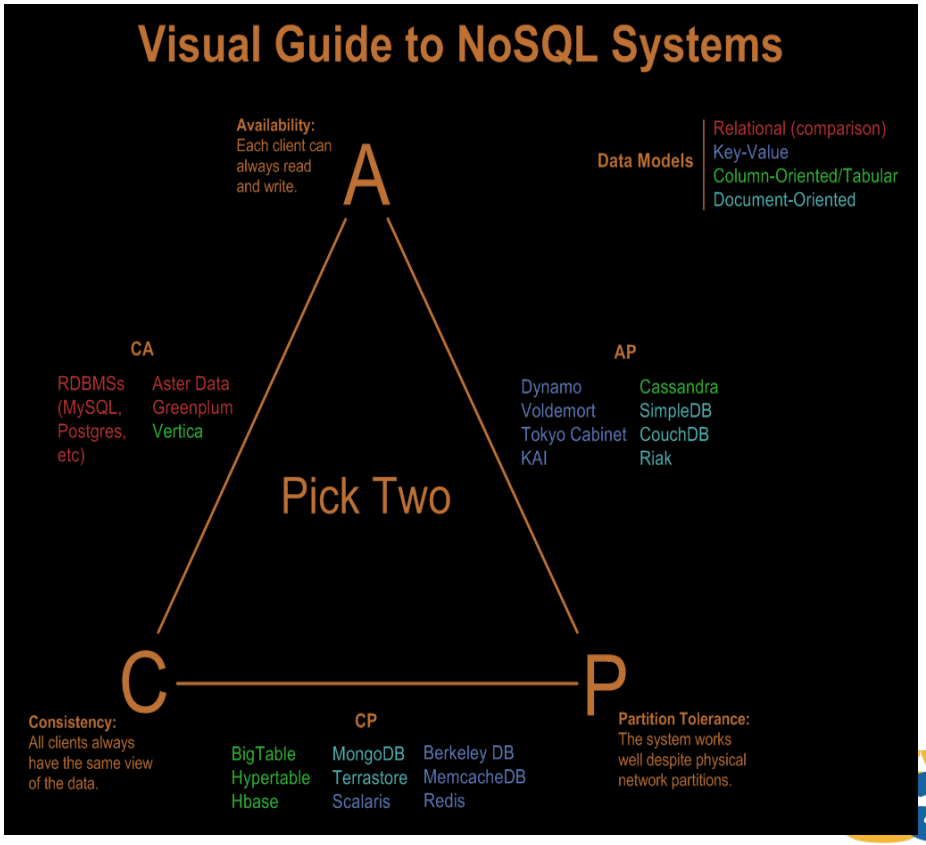
Lower Priorities

- **No Complex querying functionality**
  - No support for SQL
  - CRUD operations through database specific API
- **No support for joins**
  - Materialize simple join results in the relevant row
  - Give up normalization of data?
- **No support for transactions**
  - Most data stores support single row transactions
  - Tunable consistency and availability (e.g., Dynamo)

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Interplay with CAP

- **C**onsistency, **A**vailability, and Network **P**artitions
  - Only have two of the three together
- Large scale operations – be prepared for network partitions
- Role of CAP – During a network partition, choose between Consistency and Availability
  - RDBMS choose *consistency*
  - Key Value stores choose *availability* [low replica consistency]



Why sacrifice Consistency?

- It is a simple solution
  - nobody understands what sacrificing P means
  - sacrificing A is unacceptable in the Web
  - possible to push the problem to app developer
- C not needed in many applications
  - Banks do not implement ACID (classic example wrong)
  - Airline reservation only transacts reads (Huh?)
  - MySQL et al. ship by default in lower isolation level
- Data is noisy and inconsistent anyway
  - making it, say, 1% worse does not matter

[Vogels, VLDB 2007]

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Key Value Store

- In its simplest form, allow applications/users to
  - Insert a <key, value> pair (put operation)
  - Retrieve the <key, value> pair based on key (get operation)
- Different KV stores differ in various ways:
  - Data model and query mechanisms supported
  - How data is distributed and stored
  - How fault-tolerance is provided
  - How (and how much) consistent access is supported

Design Choices: Data Model

- KV store is a table of records where each record is key-value pair.
- Systems differ in what they support as value:
  - ***Blob Data Model***, value is an uninterpreted binary string object, i.e., a blob.
  - ***Relational data model,*** value is a flat row-like structure similar to relational data.
  - ***Column Family Data Model*** is one where the columns in the value field are grouped together into *column families*, each consisting of a set of columns
  - Multiple versions of each record, appropriately timestamped, can be supported.

uc Irvine
1965 2015
Bright Past. Brilliant Future.

Query Support

- All KV stores support atomic write (put) and read (get) based on key.
- Some may additionally support:
  - Atomic read-modify-write operation
  - Limited selections and projections (restricted to single table)

Data Distribution

- A KV store needs to partition the data to distribute it over a cluster of servers.
- The two main approaches for partitioning:
  - **range partitioning** – order records lexicographically based on key and divide it based on ordering
  - **hash partitioning –** hash records based on key to linear address space, divide space among different servers
- Different systems user different strategies for range and hash partitioning
  - E.g., Google big table uses range partitioning
  - Amazon Dynamo uses hash partitioning similar to Chord

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Query Routing

- Queries need to be routed to the server containing the KV pair of interest.
- Routing Strategies:
  - Centralized:
    - Routing logic in the client library or specialized routing servers.
    - It may maintain a full map of key intervals to servers or a pointer to index (e.g., B-tree) where leaf contain pointers
  - Decentralized:
    - Clients request routed through consistent hashing among distributed peer to peer servers.

Cluster Management

- ***Centralized approach***:
  - master keeps track of all data servers using a highly available fault-tolerant service
  - In case of failure of a data server, its KV partition is assigned to a new server.
  - If master fails, a new master is chosen to take over.
- **Decentralized approach**:
  - All data servers work as peers.
  - Gossip messages are typically used to enable each server to learn about the failure or recovery of a server. The failure of a server is detected when a gossip message from that server is missing.

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

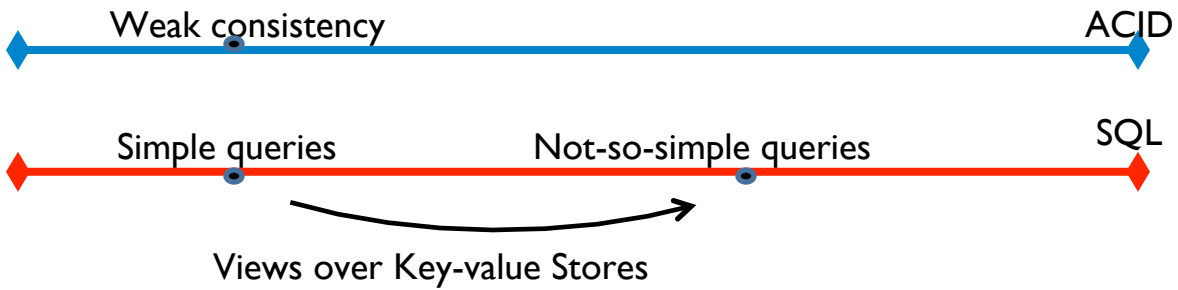
Fault-Tolerance & Replication

- Fault-tolerance is handled by replicating data on multiple servers.
- Replication
  - **Implicit** -- provided by the lower level distributed file system where data is stored. Possible when the KV store is built on top of a separate file system. (e.g., Big Table on top of GFS)
  - **Explicit** – the KV store explicity manages replicas and propagation of read/write request to the replicas.
- Replicas may support various consistency models
  - Strong Consistency -- all replicas of a given record return the same value to a read operation
  - Weak Consistency – Different copies may have different/conflicting values. Applications need to deal with inconsistency. System provides ways for application to detect inconsistency (e.g., vector clock).
  - Timeline Consistency – all copies of record apply updates in the same order. Reads can read any version.
  - Eventual Consistency – Updates eventually propagate to all copies. In the meantime, older values may be available to read.

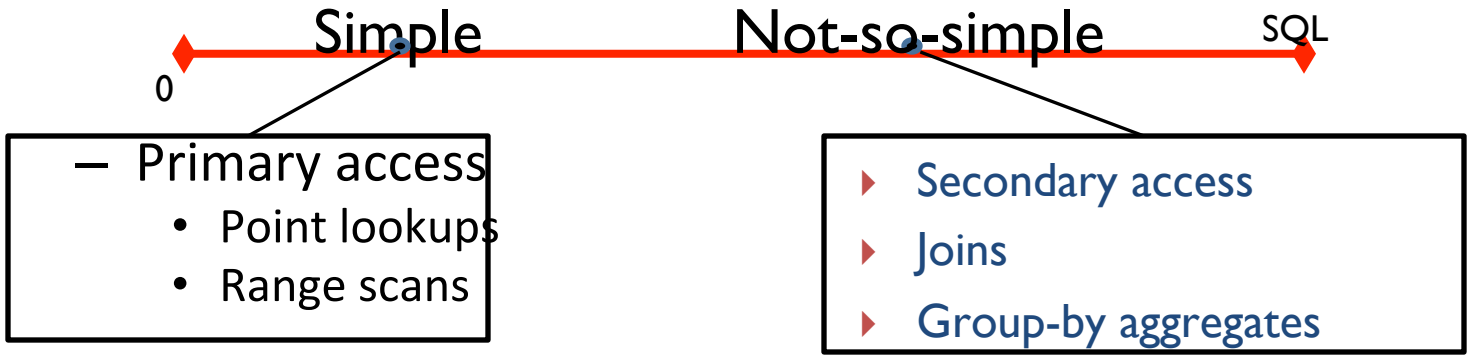
UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Key-value Stores Summary

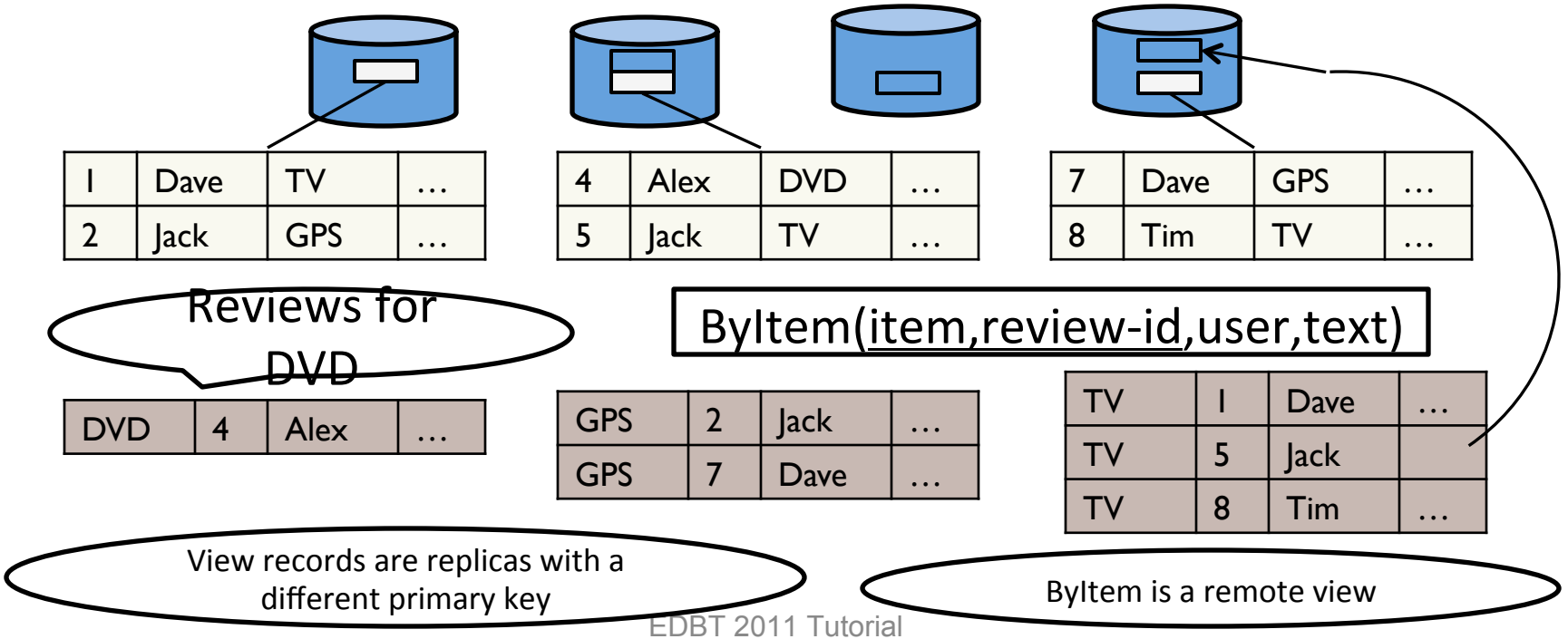
- Data-stores for Web applications & analytics:
  - PNUTS, BigTable, Dynamo, …
- Massive scale:
  - Scalability, Elasticity, Fault-tolerance, & Performance



Simple to Not-so-simple Queries



Secondary Access



SAMPLE KEY VALUE STORES

PNUTS: Yahoo!’s Hosted Data Serving Platform

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

What is PNUTS?

- PNUTS, a **massively parallel and geographically distributed** database system for Yahoo!'s web application
- Foremost requirements of a web application are
  - **Scalability –** Ability to scale by adding resources with minimal effort and minimal impact on system performance.
  - **Response Time and Geographic Scope-** Guarantee fast response times to geographically distributed users. E.g. A particular users data may be accessed by both users in USA and in China.
  - **High Availability and Fault Tolerance-** Applications must be able to read data in the presence of failures.
    - Downtime means money lost !!
    - Users disappointed if the pages are not rendered on time.
  - **Relaxed Consistency Guarantees -**
    - Most web applications tend to manipulate only one record at a time. E.g. Changing your Status, Avatar etc.
    - Fully serializable transactions over globally replicated and distributed systems is unnecessary, very expensive and inefficient.

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Data Model

- Flat row-like relational tables

Posted date	*Listing id*	*Item*	*Price*
6/1/07	424252	Couch	$570
6/1/07	763245	Bike	$86

- ❑ Flexible Schema: new attributes can be added anytime Consistency Model
- ❑ Rows may not contain values for all attributes

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

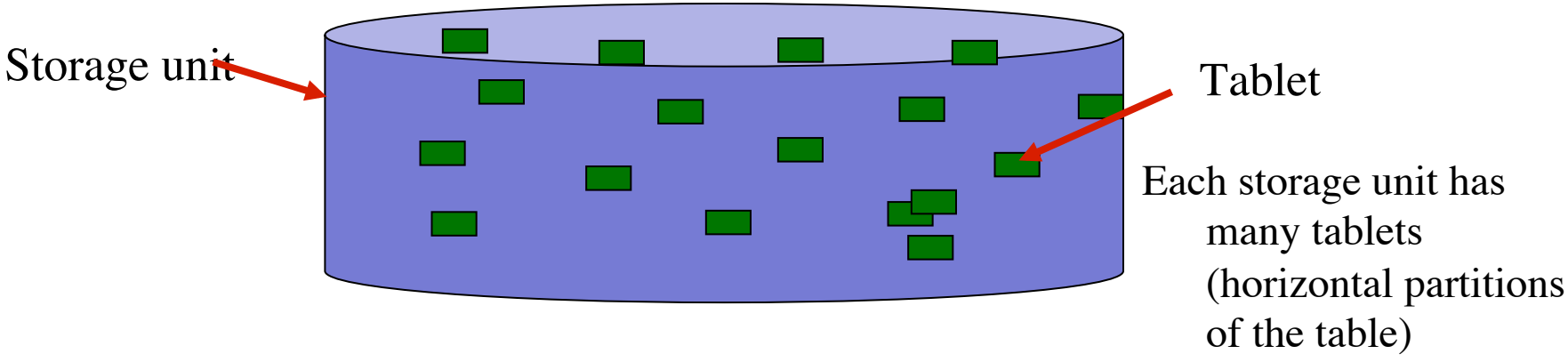
Query Support

- Simple Selection and projection queries on single table
- Updates and deletes via primary key
- Atomicity and isolation guaranteed at the granularity of only a single key-value pair
  - No guarantee for multi key-value pair access
- Offers per-record time-line consistency
  - When record replicated over multiple sites, all copies of the record apply all updates in the same order.
- Reads can specify the version of data they wish to read – most current version or a specific version.

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Data Distribution & Routing

- Tables are horizontally partitioned into tablets
  - Tablet = group of records
- Tablets stored at Storage Units
- Tablet controller maintain info about mapping from key ranges → servers
- Routers cache the mapping



Tablets—Ordered Table

	Name	*Description*	*Price*
A	Apple	Apple is wisdom	$1
	Avocado	But at what price?	$3
	Banana	The perfect fruit	$2
	Grape	Grapes are good to eat	$12
H	Kiwi	New Zealand	$8
	Lemon	How much did you pay for this lemon?	$1
	Lime	Limes are green	$9
	Orange	Arrgh! Don't get scurvy!	$2
Q	Strawberry	Strawberry shortcake	$900
	Tomato	Is this a vegetable?	$14
Z			

Key -> interval -> storage unit

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Tablets—Hash Table

	Name	*Description*	*Price*
0x0000	Grape	Grapes are good to eat	$12
	Lime	Limes are green	$9
	Apple	Apple is wisdom	$1
	Strawberry	Strawberry shortcake	$900
0x2AF3	Orange	Arrgh! Don't get scurvy!	$2
	Avocado	But at what price?	$3
	Lemon	How much did you pay for this lemon?	$1
	Tomato	Is this a vegetable?	$14
0x911F	Banana	The perfect fruit	$2
0xFFFF	Kiwi	New Zealand	$8

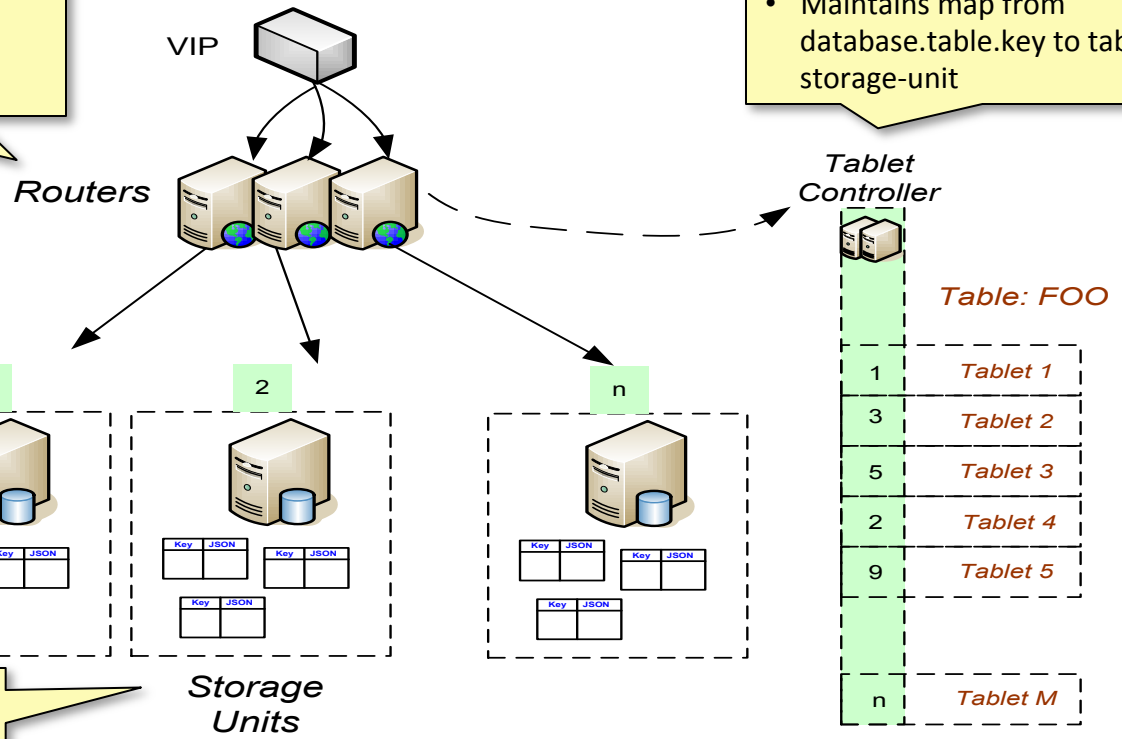
Key -> *hash value* -> interval -> storage unit

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

PNUTS-Single Region

- Routes client requests to correct storage unit
- Caches the maps from the tablet controller

- Maintains map from database.table.key to tablet to storage-unit

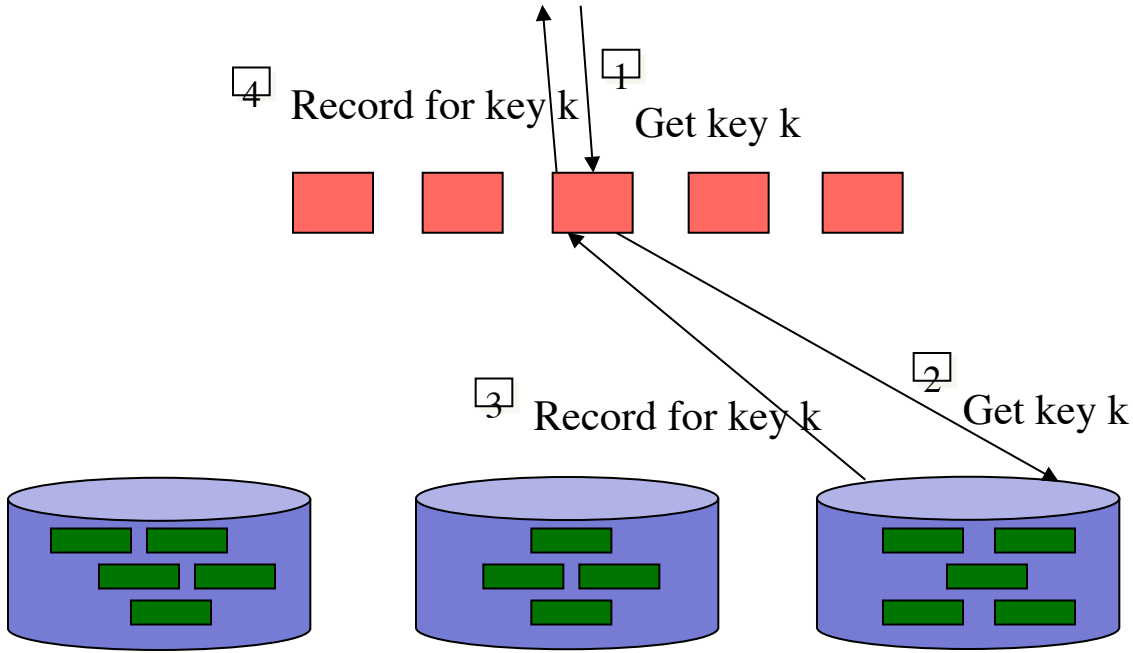


- Stores records
- Services get/set/delete requests

Source: Silberstein's OSCON 2011 slide

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Accessing Data

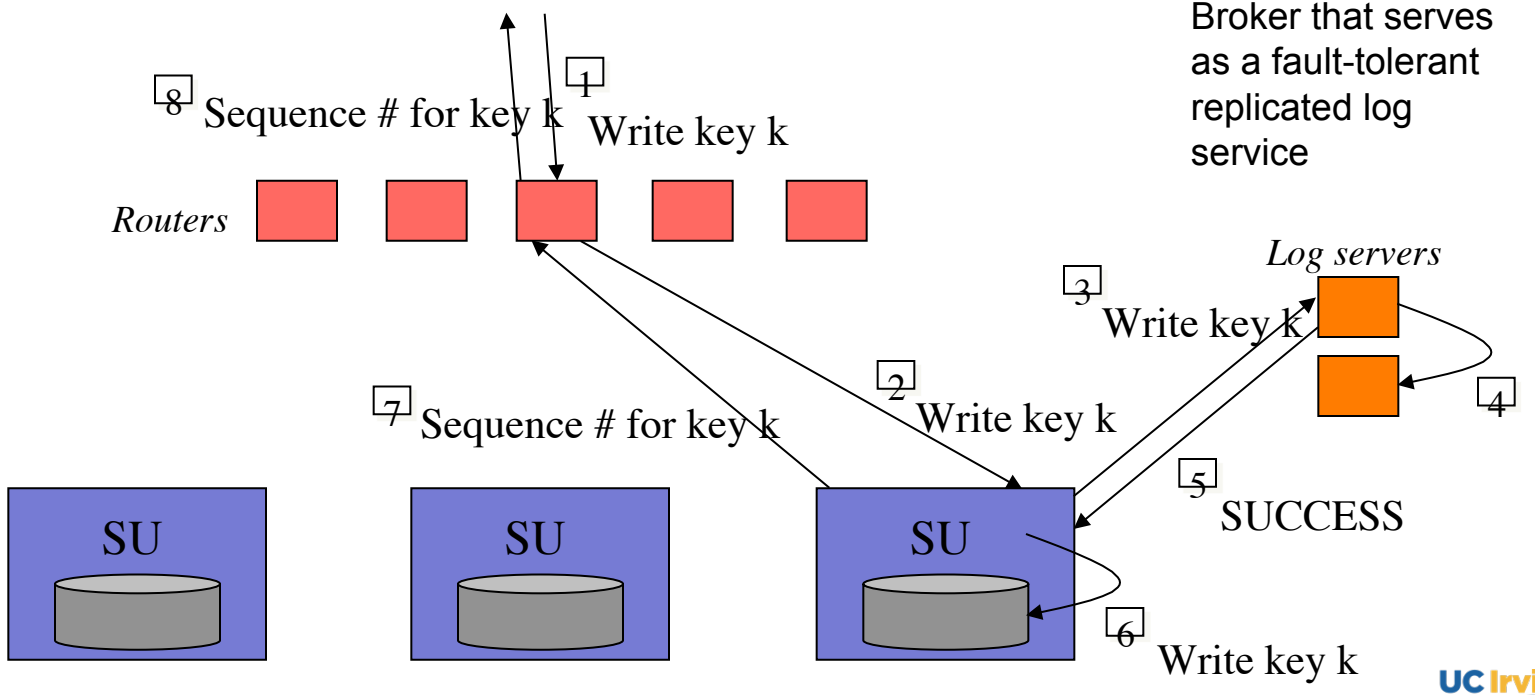


Source: VLDB 2008

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Updates

Logging done by Yahoo Message Broker that serves as a fault-tolerant replicated log service



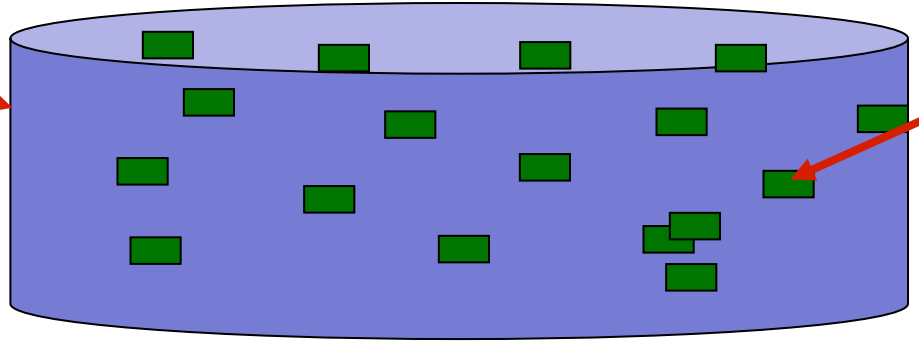
Source: VLDB 2008

Tablet Evolution

Tablets may grow over time

Overfull tablets split

Storage Units may shed load by moving tablets to other servers



Each storage unit has many tablets (horizontal partitions of the table)

Data Replication

Data fully replicated across diverse geographical locations

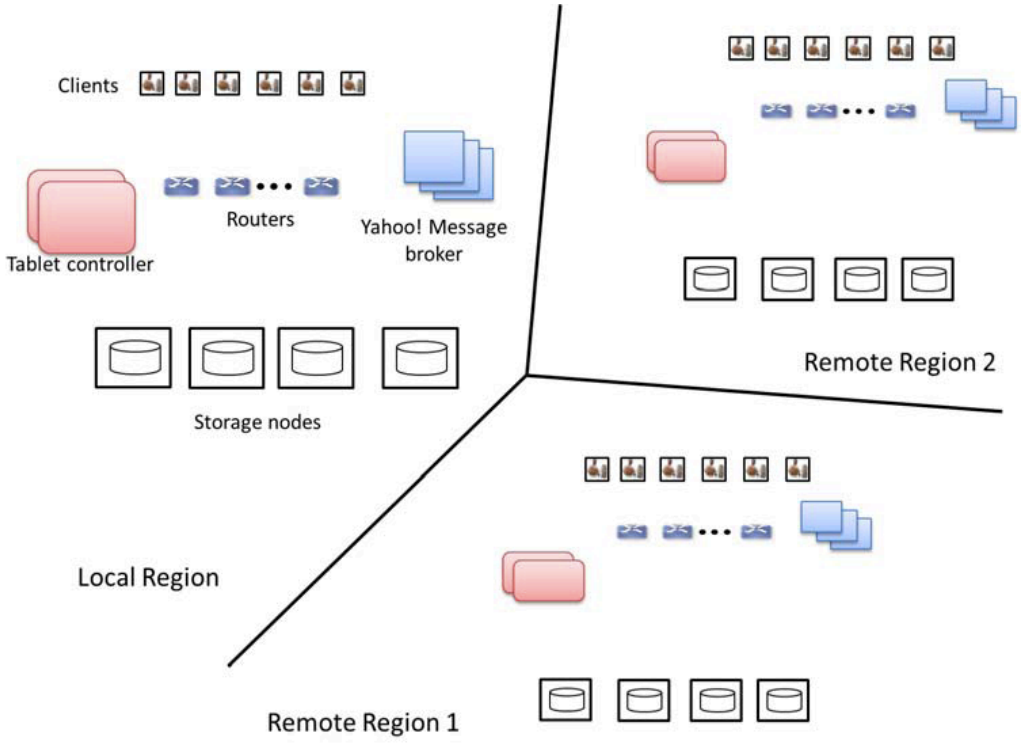
Replication through Yahoo Message Broker (YMB)

YMB guarantees ordered delivery of updates and guarantees record-level timeline consistency for replicas

For this purpose, per-record mastering strategy used.

Updates routed to master and propagated asynchronously to replicas

Remastering occurs when the master fails



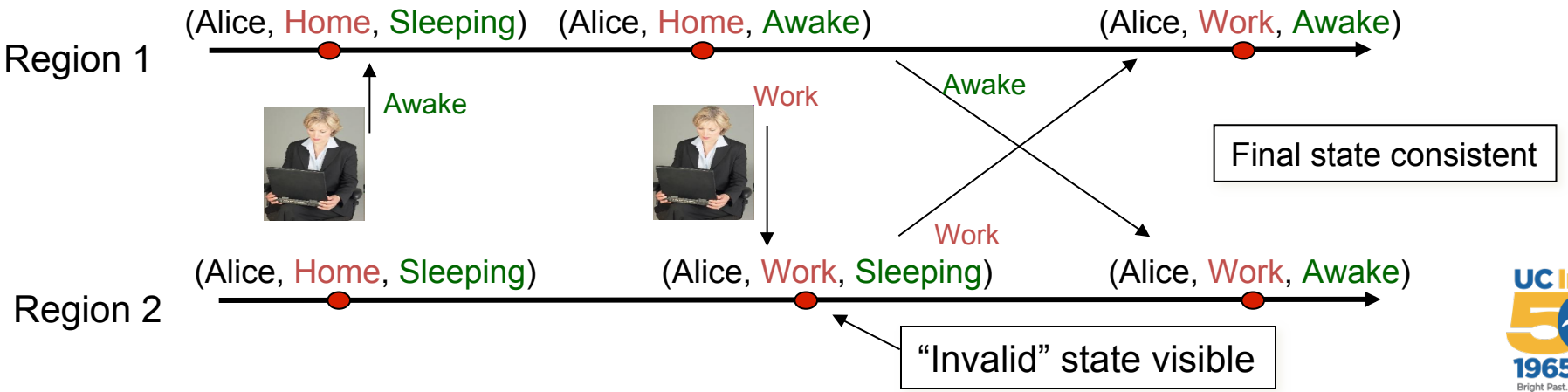
Yahoo! Message Broker (Log Server)

- **Pub/sub (publish/subscribe)**
  - Clients subscribe to updates
  - Updates done by a client are propagated to other clients
  - Thus, the subscribing clients are **notified** of the updates
- **Usages**
  - **Updates**: updates are “committed” only when they are published to YMB

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

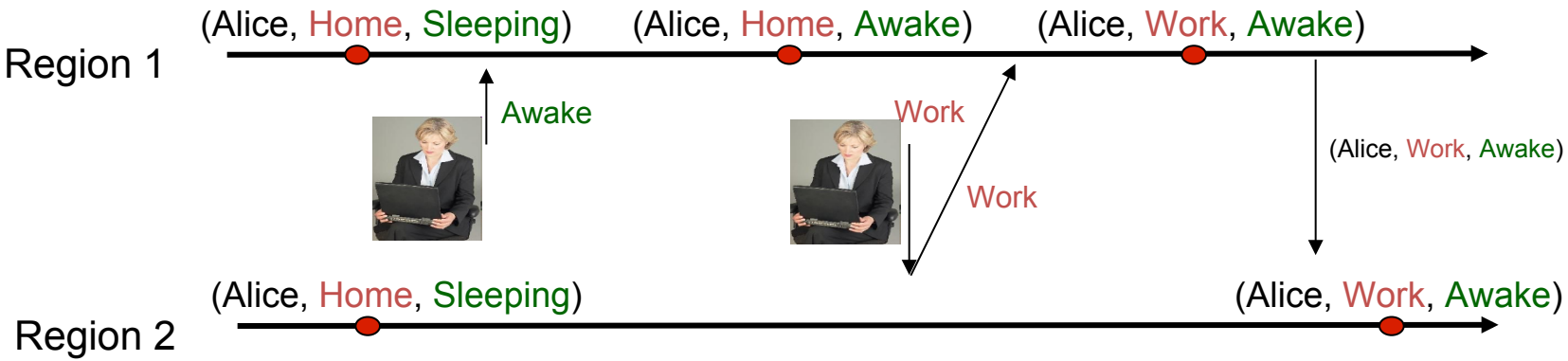
Consistency levels

- Eventual consistency
  - Transactions:
    - Alice changes status from "Sleeping" to "Awake"
    - Alice changes location from "Home" to "Work"



Consistency levels

- Timeline consistency
  - Transactions:
    - Alice changes status from "Sleeping" to "Awake"
    - Alice changes location from "Home" to "Work"



PNUTS Summary

- Designed for Yahoo web applications to meet the need for large number of clients
- All user wants is to upload his photo, share a status, thus we don't need
  - Complicated Queries
  - Strong Transactions
- But we need Scalability , Availability and Flexibility
- Core elements of the PNUTS solution
  - Simple record oriented data model
  - Limited queries
  - Geographical distribution over WAN
  - Asynchronous replicatin using yahoo message broker
  - Per-record timeline consistency
    - Impleented using single record mastering

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

BIGTABLE

BigTable

- Distributed storage system for managing structured data
- Designed to scale to a very large size
  - Petabytes of data across thousands of servers, Terabytes of in-memory data
  - Millions of reads/writes per second, efficient scans
- Hugely successful within Google – used for many Google projects
  - Web indexing, Personalized Search, Google Earth, Google Analytics, Google Finance, …
- Highly-available, reliable, flexible, high-performance solution for all of Google's products
- Self-managing
  - Servers can be added/removed dynamically
  - Servers adjust to load imbalance
- Offshoots/followons:
  - Spanner: Time-based consistency
  - LevelDB: Open source incorporating aspects of Big Table

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, …
- Big Data scale
  - Billions of URLs, many versions/page (~20K/version)
  - Hundreds of millions of users, thousands or q/sec
  - 100TB+ of satellite image data

Goals

- A general-purpose data-center storage system
- Asynchronous processes continuously updating different pieces of data
  - Access most current data at any time
  - Examine changing data (e.g., multiple web page crawls)
- Need to support:
  - Durability, high availability, and very large scale
  - Big or little objects
  - Very high read/write rates (millions of ops per second)
  - Ordered keys and notion of locality
    - Efficient scans over all or interesting subsets of data
    - Efficient joins of large one-to-one and one-to-many datasets

Building Blocks

- Big Table builds on top of many Google systems
  - Google File System (GFS): Raw storage
  - Scheduler: schedules jobs onto machines
  - Chubby 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 serving BigTable
  - Lock service: master election, location bootstrapping
  - Map Reduce: often used to read/write BigTable data

BigTable Data Model

- A big sparse, distributed persistent multi-dimensional sorted map
  - Rows are sort order
  - Atomic operations on single rows
  - Scan rows in order
  - Locality by rows first

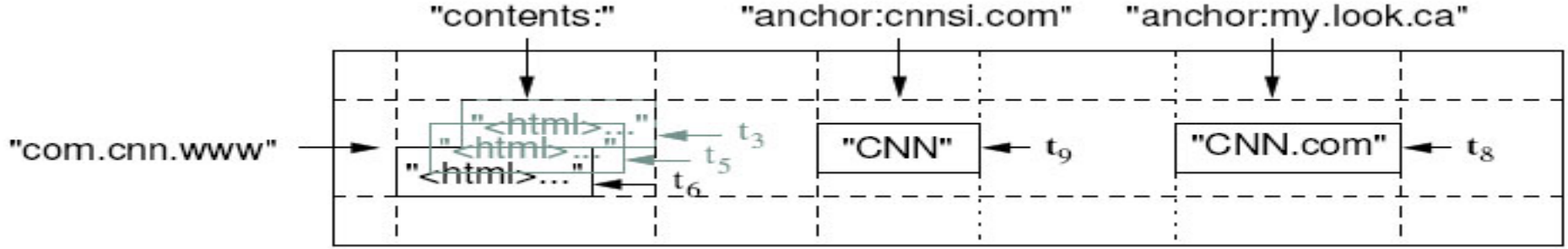
- Columns: properties of the row
  - Variable schema: easily create new columns
  - Column families: groups of columns
    - For access control (e.g. private data)
    - For locality (read these columns together, with nothing else)
    - Harder to create new families

- Multiple entries per cell using timestamps
  - Enables multi-version concurrency control across rows

Basic Data Model

- Multiple entries per cell using timestamps
  - Enables multi-version concurrency control across rows

(row, column, timestamp) ➔ cell contents



- Good match for most Google applications:
  - Large collection of web pages and related information
  - 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

- Row creation is implicit upon storing data
  - Rows ordered lexicographically
  - Rows close together lexicographically usually on one or a small number of machines
  - Rows with consecutive keys are grouped together as “tablets”.
- 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`

Columns

- Columns have two-level name structure:
    - family:optional_qualifier
- Column family
  - Unit of access control
  - Has associated type information
- Qualifier gives unbounded columns
  - Additional levels of indexing, if desired

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"*

Client APIs

- **Bigtable APIs provide functions for:**
  - Creating/deleting tables, column families
  - Change table and column family metadata such as access control rights
  - Lookup, write, delete values, iterate over rows.
  - Support for single row transactions (including read-modify-write)
  - No multi-row transactions
  - Client supplied scripts can be executed in the address space of servers

Data Distribution

- Table range partitioned into tablets based on row key
  - Tablets are units of distribution and load balancing.
  - Tablets stored at tablet servers that handle read/ write request to tablets.

Tablet Location & Routing

- All tables in the system are indexed together using a B-tree style 3 level data structure
  - **Root Tablet** *(implemented as a file in Chubby)* contains location of *Metadata tablets*
  - **Metadata table** contains location of user tablets
    - Row-Key: [Tablet's Table ID] + [End Row]
- Tablet location cached at client library for faster access. If data stale, clients can access hierarchical data structure for routing.

Three part implementation

- Client Library (every client)
  - Caches tablet locations, communicates directly with tablet servers to retrieve data, and to write data. Acquires locks using chubby for its read/ write requests
- One master server
- Many tablet servers – that serve parts of many tables.

Master Server

- Master Server tasks
  - Assigning tablets to servers
  - Detection the addition/expiration of servers
  - Balancing servers' loads
  - Garbage collection in GFS
  - Handling schema changes

- Clients only deal with master to create/delete tables and column family changes

Tablet Server

- Tablet server tasks:
  - Handling R/W requests to the loaded tablets
  - Splitting tablets
    - Server commits by recording new tablet's info in Metadata
    - Notifies the master
  - Implemented using Google File System (tablets stored in SStable format)
- Clients communicate with servers directly
  - Master lightly loaded
- Each table
  - One tablet at the beginning
  - Splits as grows, each tablet of size 100-200 MB

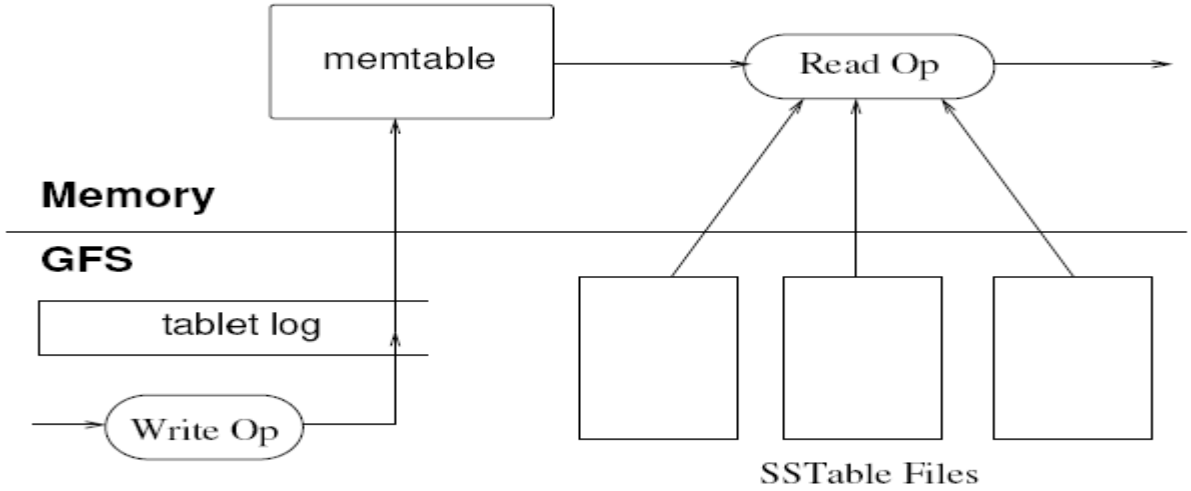
Updates to Tablets

- Writes go to log then to in-memory table “memtable” (key, value)

- Periodically: move in memory table to disk => SSTable(s)
  - SSTable = immutable ordered subset of table: range of keys and subset of their columns
    - One locality group per SSTable (for columns)
  - Tablet = all of the SSTables for one key range + the memtable
    - Tablets get split when they get too big
  - Some values may be stale (due to new writes to those keys)

- Prior to writes being reflected on memtable, they are written to a REDO log (write ahead logging).
- Redo log implemented using GFS

Tablet Serving



Tablet Recovery:
- Server reads its list of SSTables from METADATA Table
- List = (Comprising SSTables + Set of ptrs to REDO commit logs
- Server reconstructs the status and memtable by applying REDOs

Locality Groups

- Group column families together into an SSTable
  - Avoid mingling data, ie page contents and page metadata
  - Can keep some groups all in memory
- Can compress locality groups
- Bloom Filters on locality groups – avoid searching SSTable
  - Efficient test for set membership: member(key) ➔ true/false
    - False => definitely not in the set, no need for lookup
    - True => probably is in the set (do lookup to make sure and get value)
  - Generally supports adding elements, but not removing them
    - … but some tricks to fix this (counting)
    - … or just create a new set once in a while

Bloom Filters

- Basic version:
  - m bit positions
  - k hash functions
  - for insert: compute k bit locations, set them to 1
  - for lookup: compute k bit locations
    - all = 1 => return true (may be wrong)
    - any = 0 => return false
  - 1% error rate ~ 10 bits/element
    - Good to have some a priori idea of the target set size
- Use in BigTable
  - Avoid reading all SSTables for elements that are not present (at least mostly avoid it)
    - Saves many seeks

Reads on Tablets

- Reads: maintain in-memory map of keys to {SSTables, memtable}
  - Current version is in exactly one SSTable or memtable
  - Reading based on timestamp requires multiple reads
  - May also have to read many SSTables to get all of the columns

- Scan = merge-sort like merge of SSTables in order
  - Easy since they are in sorted order

Compaction

- Minor compaction
  - (Memtable size > threshold) → New memtable
  - Old one converted to an SSTable, written to GFS
    - Shrink memory usage & Reduce log length in recovery
- Major compaction
  - Rewrites all SSTables into exactly one table
  - BT reclaim resources for deleted data
  - Deleted data disappears (sensitive data)

Other Refinements

- Two level caching in servers
  - Scan cache (K/V pairs)
  - Block cache (SSTable blocks read from GFS)
- Commit log implementation
  - Each tablet server has a single commit log
  - Complicates recovery
    - Master coordinates sorting log file *<Table, Row, Log Seq)*

PNUTs versus BigTable – discussion

- Data Model?
- Consistency?
- Data distribution and routing?
- Leveraging existing systems?

- Most differences in design a result of differences in goals
  - Bigtable – scalable, flexible, general purpose, storage for a large data center.
  - Pnuts – storage architecture to meet the need for a company requiring wide area replication

Building Blocks

- Big Table builds on top of many Google systems
  - Google File System (GFS): Raw storage
  - Scheduler: schedules jobs onto machines
  - Chubby 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 serving BigTable
  - Lock service: master election, location bootstrapping
  - Map Reduce: often used to read/write BigTable data

Chubby Lock Service

- What is *Chubby* ?
  - Highly available persistent lock service.
  - Simple file system with directories and small files
  - Reads and writes to files are atomic.
  - When session ends, clients loose all locks

- What is it used for in BigTable implementation
  - Store the root tablet, schema information, access control lists.
  - Synchronize and detect tablet servers

Google Cluster Management

- What is it used for in BigTable?
  - Scheduling jobs
  - Managing resources on shared machines
  - Monitoring machine status
  - Dealing with machine failures

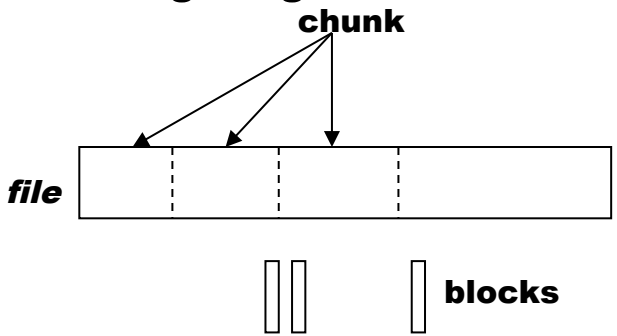
Google File System

- GFS is a distributed file system designed by Google.
    - Designed for large files, usually written once but read often for which the most common operation is append.
  - supports strongly consistent replicated data service
  - Designed for a single data center. So outage may make data unavailable.
- What is it used for in BigTable implementation?
  - Serves as the underlying storage system for BigTable. Stores and retrieves tablets (in Sstable format).

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

GFS Architecture

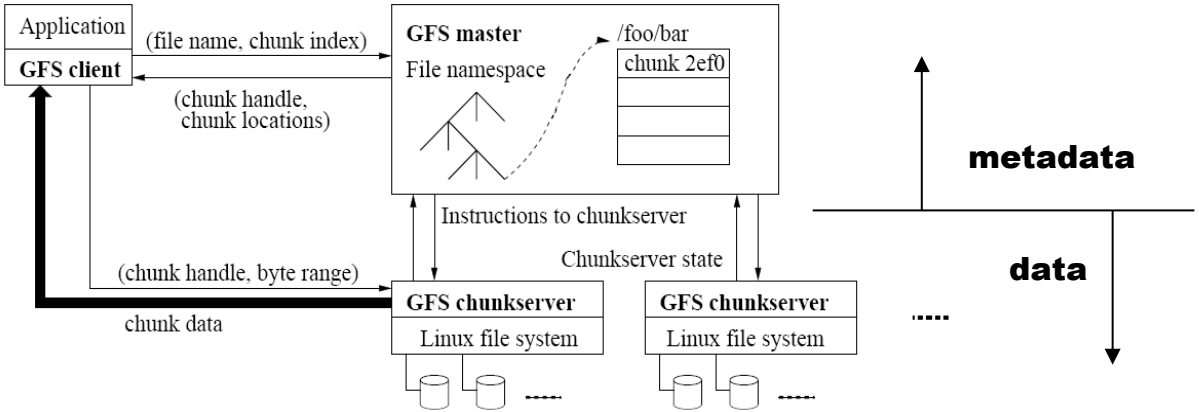
- Single master + multiple chunk servers
- Files are divided into chunks of fixed size (eg. 64 MB)
- Each chunk is assigned a unique chunk handle and contains several replicas in system.
- Master contains metadata including: Namespace, Mapping from files to chunks, current location of chunks..
- Master communicates with chunk server through regular heartbeat message



UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Architecture (contd.)

- Master -Manages namespace/metadata
  - Manages chunk creation, replication, placement
  - Performs checkpointing, logging and snapshots of changes to metadata
- Chunkservers
  - Stores chunk data and checksum for each block
  - On startup/failure recovery, reports chunks to master
  - Periodically reports sub-set of chunks to master (to detect no longer needed chunks)



Read Operation

- Client translate offset of a file into chunk index
- Client send request to master, master replies with chunk handle and location of replicas
- Client cache the result from master, send a request to one of the replicas and retrieve the data
- Further read of same replica requires no client-master interaction

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Write Operation

- Client ask master about the primary chunk and location of other replicas
- Client push data to all replicas through shortest route
- After all replica receive data successfully, the primary replica decide serial writing order and transmit the order to all replicas
- After all replica finish writing, primary replica is notified and then client is notified

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Master Operation

- Locking mechanism to allow concurrent mutation
- Choose location of replica, replicate stale replica, rebalance whole system
- Garbage collection: stale chunk, deleted files
- Fault tolerance and diagnosis: each chunk has chunksum. if chunksum in chunk server mismatch that in master, stale chunk is deleted and replicated

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

NOW BACK TO BIGTABLE….

Amazon Dynamo

Used by Amazon's EC2 Cloud Hosting Service. Powers their Elastic Storage Service called S3 as well as their E-commerce platform

Offers a simple key-value lookup on distributed low cost virtualized nodes

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Dynamo Goals

- Scale – adding systems to network causes minimal impact
- Symmetry – No special roles, all features in all nodes
- Decentralization – No Master node(s)
- Highly Available – Focus on end user experience
- Service Level Agreements – System can be adapted to an application's specific needs, allows flexibility

Dynamo Data Model

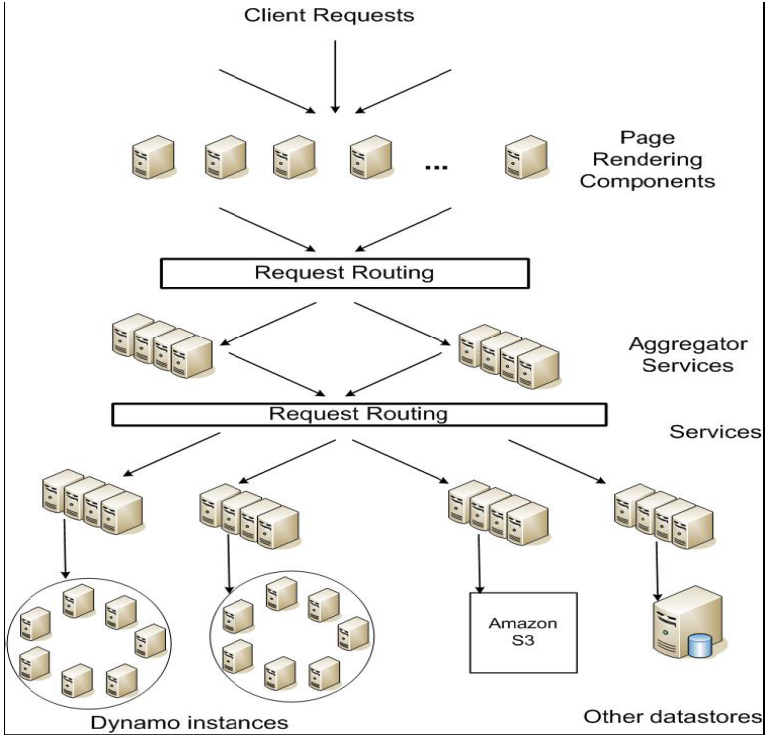
- Table consisting of <key, value> pairs where value is a blob
- Operations supported:
  - Put
  - Get
  - No delete
  - No multi object queries

Dynamo Assumptions

- Atomicity, Consistency, Isolation, Durability
  - Operations either succeed or fail, no middle ground
  - System will be eventually consistent, no sacrifice of availability to assure consistency
  - Conflicts can occur while updates propagate through system
  - System can still function while entire sections of network are down
- Efficiency – Measure system by the 99.9th percentile
  - Important with millions of users, 0.1% can be in the 10,000s
- Non Hostile Environment
  - No need to authenticate query, no malicious queries
  - Behind web services, not in front of them

Service Level Agreements (SLA)

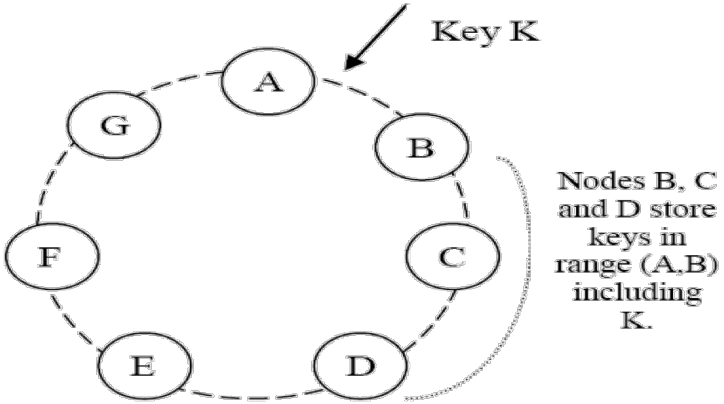
- Application can deliver its functionality in a bounded time:
  - Every dependency in the platform needs to deliver its functionality with even tighter bounds.
- Example: service guaranteeing that it will provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second
- Contrast to services which focus on mean response time



Service-oriented architecture of Amazon's platform

Partitioning and Routing Algorithm

- Consistent hashing:
  - the output range of a hash function is treated as a fixed circular space or “ring”.
- Virtual Nodes:
  - Each physical node can be responsible for more than one virtual node
  - Used for load balancing
- Routing: “zero-hop”
  - Every node knows about every other node
  - Queries can be routed directly to the root node for given key
  - Also – every node has sufficient information to route query to all nodes that store information about that key



Consistent Hashing (recap)

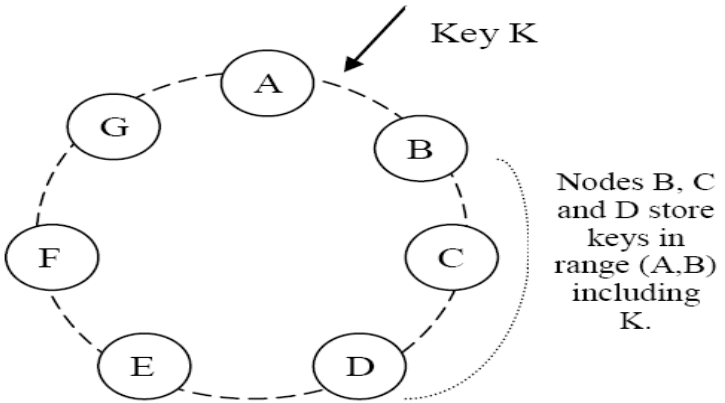
Basics :-

- Assigns each node and key an *identifier* using a base hash function.
- The hash space is large, and is treated as if it wraps around to form a circle - hence *hash ring*. Identifiers are ordered on an *identifier circle* modulo M.
- Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position.

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

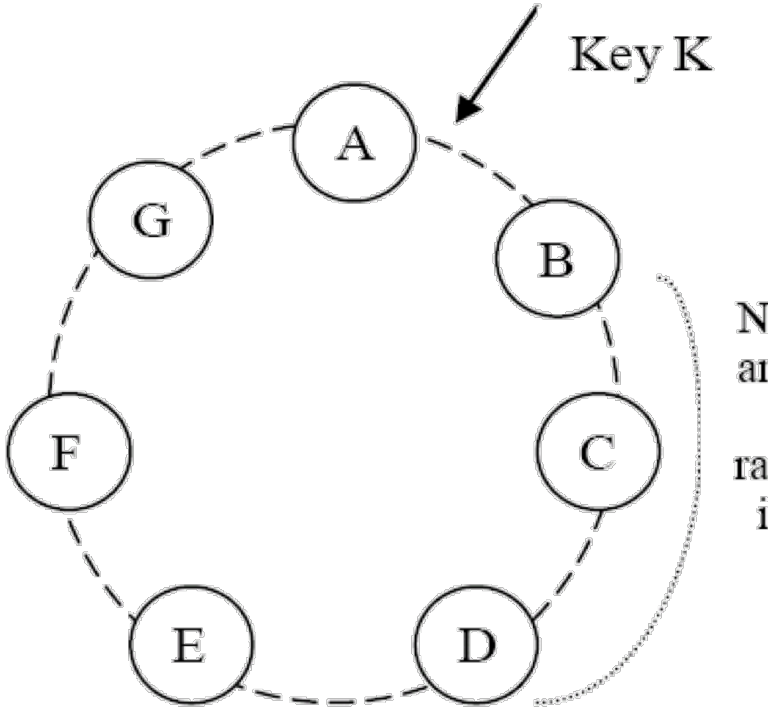
Advantages of using virtual nodes

- If a node becomes unavailable the load handled by this node is evenly dispersed across the remaining available nodes.
- When a node becomes available again, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes.
- The number of virtual nodes that a node is responsible can be decided based on its capacity, accounting for heterogeneity in the physical infrastructure.



Replication

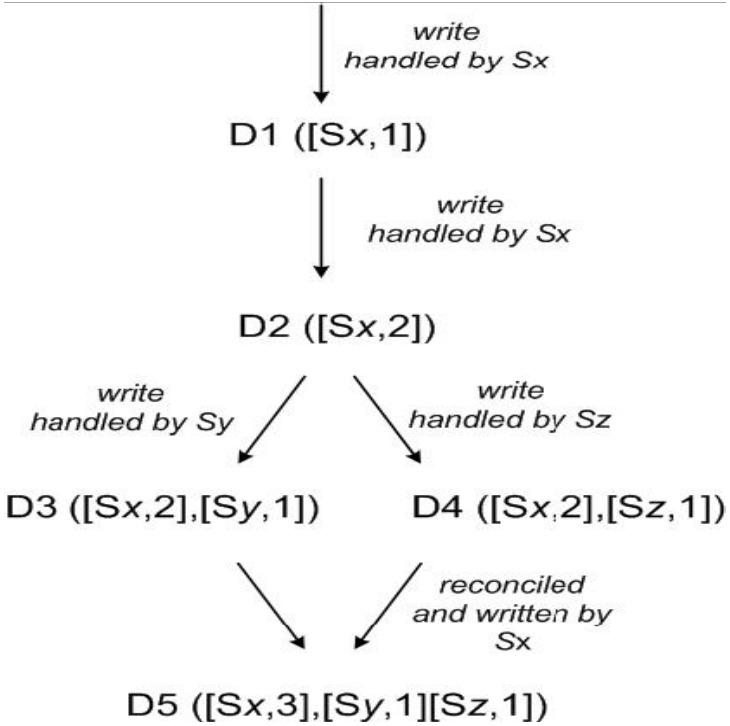
- Each data item is replicated at N hosts.
- "preference list": The list of nodes responsible for storing a particular key
  - Successive nodes not guaranteed to be on different physical nodes
  - Thus preference list includes physically distinct nodes



Data Versioning

- A put() call may return to its caller before the update has been applied at all the replicas
- A get() call may return many versions of the same object.
- Challenge:
  - an object having distinct version sub-histories, which the system will need to reconcile in the future.
- Solution:
  - uses vector clocks in order to capture causality between different versions of the same object.

Vector clock example



Conflicts (multiversion data)

- Client must resolve conflicts
  - Only resolve conflicts on reads
  - Different resolution options:
    - Use vector clocks to decide based on history
    - Use timestamps to pick latest version
  - Examples given in paper:
    - For shopping cart, simply merge different versions
    - For customer's session information, use latest version
  - Stale versions returned on reads are updated ("read repair")

Execution of get () and put () operations

- Route its request through a generic load balancer that will select a node based on load information
  - Simple idea, keeps functionality within Dynamo
- Use a partition-aware client library that routes requests directly to the appropriate coordinator nodes
  - Requires client to participate in protocol
  - Much higher performance

	99.9th percentile read latency (ms)	99.9th percentile write latency (ms)	Average read latency (ms)	Average write latency (ms)
Server-driven	68.9	68.5	3.9	4.02
Client-driven	30.4	30.4	1.55	1.9

Quorum for Consistency

- Dynamo supports quorums get() and put() must sychronously get values from/update R/W nodes to succeed.
  - Put() waits for acks from W replicas before succeeding.
  - Likewise, get() reads from R replicas before succeeding.

- If R + W > N, then strong consistency (quorum-like system)

- But, latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency.

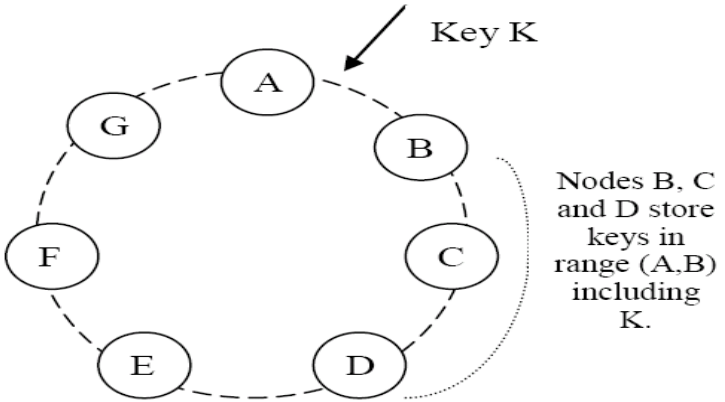
Sloppy Quorums

- Vary N, R, W to match requirements of applications
  - High performance reads: R=1, W=N
  - Fast writes with possible inconsistency: W=1
  - Common configuration: N=3, R=2, W=2

- When do branches occur?
  - Branches uncommon: 0.0006% of requests saw > 1 version over 24 hours
  - Divergence occurs because of high write rate (more coordinators), not necessarily because of failure

Hinted handoff

- Assume N = 3. When B is temporarily down or unreachable during a write, send replica to E
- E is hinted that the replica belongs to B and it will deliver to B when B is recovered.
- Again: “always writeable”

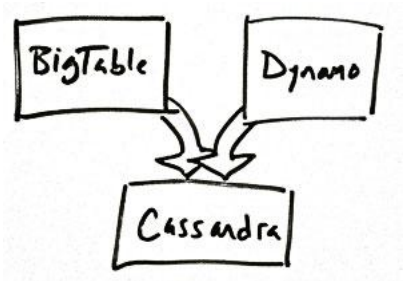


Implementation

- Java
  - Event-triggered framework similar to SEDA
- Local persistence component allows for different storage engines to be plugged in:
  - Berkeley Database (BDB) Transactional Data Store: object of tens of kilobytes
  - MySQL: object of > tens of kilobytes
  - BDB Java Edition, etc.

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.



Cassandra

- Facebook's data cloud for their servives
- Cassandra = BigTable data model + Dynamo distribution work together…

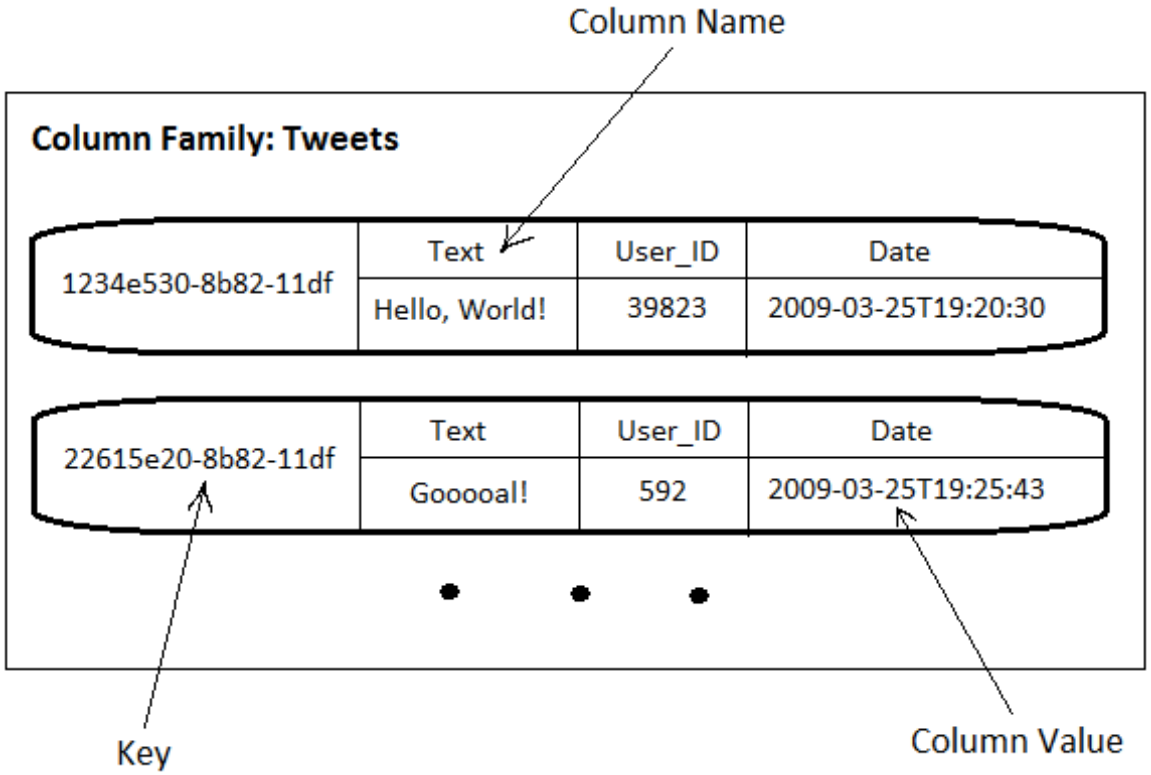
UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Data Model

- Multi-dimensional map.
  - key - string, generally 16-36 bytes long. No size restrictions.
- Column Family - group of columns.
  - Simple column family
  - Super - column family of column families.
  - Column order: sorted by timestamp of name. (Time sorting helps in Inbox Search)
- Column Access
  - Simple column
    - *column_family:column*
  - Super column
    - *column_family:super_column:column*

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Data Model (contd)



Relational Model	Cassandra Model
Database	Keyspace
Table	Column Family (CF)
Primary key	Row key
Column name	Column name/key
Column value	Column value

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

API

- Three simple methods
  - *insert (table, key, rowMutation)*

```
[default@keyspace] set User['vmeru']['fname'] = 'Varad';
[default@keyspace] set User['vmeru'][ascii('email')] = 'vmeru@uci.edu';
     cqlsh> INSERT INTO users (firstname, lastname, age, email, city)
      VALUES ('John', 'Smith', 46, 'johnsmith@email.com', 'Sacramento');
```

  - *get (table, key, columnName)*

```
[default@keyspace] get User['vmeru'];
=> (column=656d6169c, value=vmeru@uci.edu,timestamp=135225847342)
     cqlsh:demo> SELECT * FROM users where lastname= 'Doe';
```

  - *delete (table, key, columnName)*

```
[default@keyspace] del User['vmeru'];
row removed.
     cqlsh:demo> DELETE from users WHERE lastname = "Doe";
```

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

System Architecture

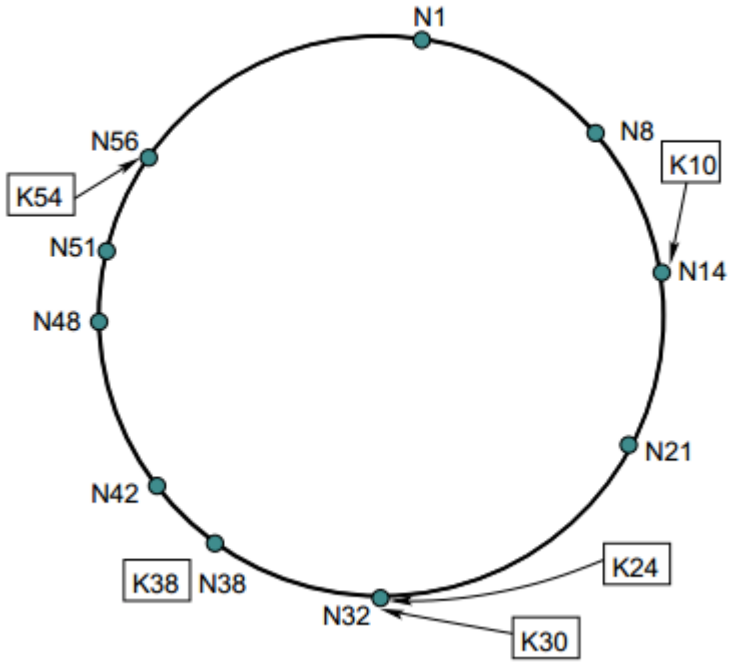
- Partitioning
- Replication
- Membership
- Failure
- Handling
- Scaling

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Partitioning

- Purpose:
  - The ability to dynamically partition the data over the set of nodes.
  - The ability to scale incrementally.

- Consistent hashing using an order preserving hash function



19652015
Bright Past. Brilliant Future.

Replication

Purpose: achieve high availability and durability

- Each data item is replicated at N hosts, where N is the replication factor
- Each key, *k*, is assigned to a coordinator node. The coordinator is in charge of the replication of the data items that fall within its range.

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Failure Detection

Purpose

- Locally determine if any other node in the system is up or down
- avoid attempts to communicate with unreachable nodes during various operations.

Modified version of the ΦAccrual Failure Detector :

- Doesn’t emit a Boolean value stating a node is up or down
- Emits a value which represents a suspicion level for each of monitored nodes.

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Bootstrapping

- When a node starts for the first time, it chooses a random token for its position in the ring.
- For fault tolerance, the mapping is persisted to disk locally and also in Zookeeper.
- The token information is then gossiped around the cluster.

This is how we know about all nodes and their respective positions in the ring. This enables any node to route a request for a key to the correct node in the cluster.

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Scaling the Cluster

When a new node is added into the system, it gets assigned a token such that it can alleviate a heavily loaded node.

- Bootstrap algorithm is initiated from any other node in the system
- The node giving up the data streams the data over to the new no de using kernel-kernel copy techniques.
- Operational experience has shown that data can be transferred at the rate of 40 MB/sec from a single node.
- Improving this by having multiple replicas take part in the bootstrap transfer thereby parallelizing the effort, similar to Bittorrent

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Local Persistence

- Typical write operation involves a write into a commit log for durability and recoverability and an update into an in-memory data structure.
- The write into the in-memory data structure is performed only after a successful write into the commit log
- When the in-memory data structure crosses a certain threshold, it dumps itself to disk
- Over time many such files could exist on disk and a merge process runs in the background to collate the different files into one file.

Speed Up

- Bloom Filter: avoid lookups into files that do not contain the key
- Column Indices: prevent scanning of every column on disk and which allow us to jump to the right chunk on disk for column retrieval

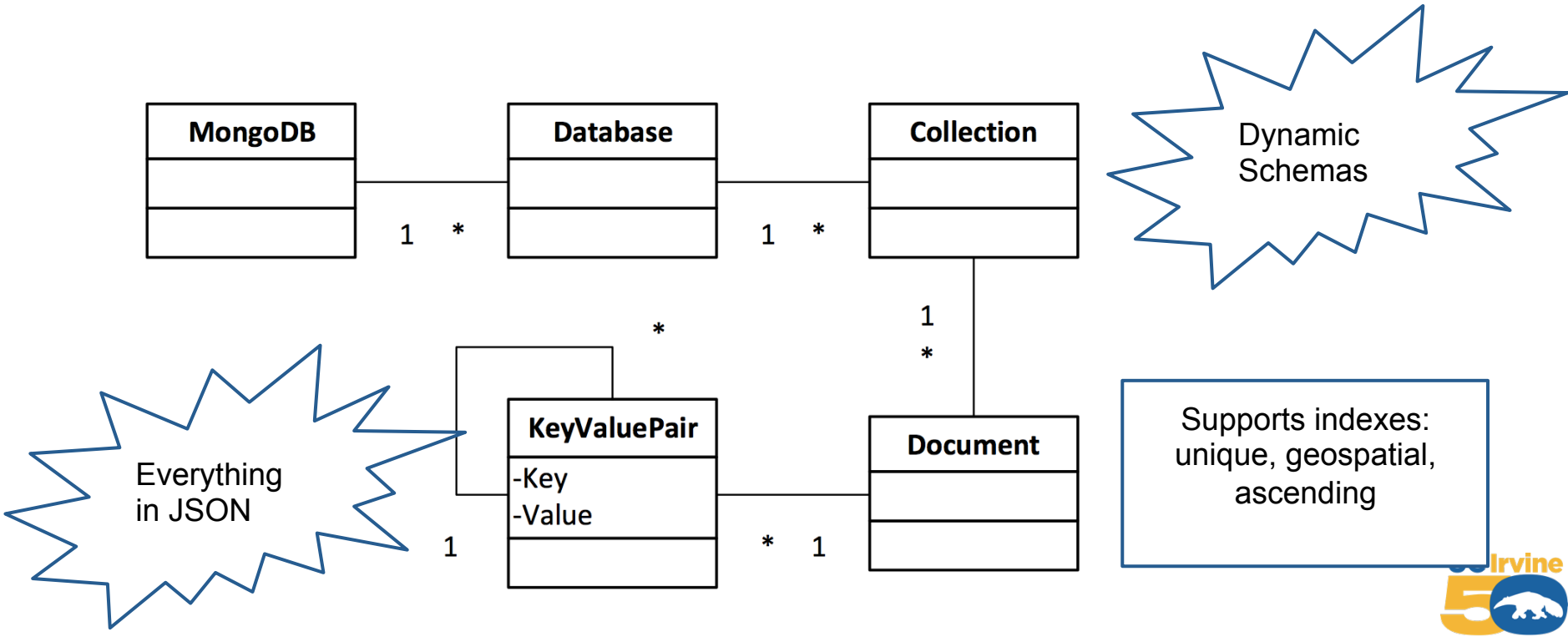
UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

MongoDB

- By 10gen in C++
- NoSQL - databases without schema
- Document-based databases
- Two types of replication
  - Master-slave: Master - full access, slaves read
  - Replica sets - same, election of new master
- Automatic Sharding
  - Data can be partitioned over multiple nodes
  - Sharding Key for each collection

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Model


MongoDB
Database
Collection
1
*
1
*
Dynamic Schemas
*
1
*
Everything in JSON
KeyValuePair
-Key
-Value
1
*
1
Document
Supports indexes: unique, geospatial, ascending

Limitations

- By 10gen in C++
  - Transactions not directly supported
  - Atomic Operations: findAndModify, $inc
- Two-phase commits
- 32-bit Mongo - data set limited to 2.5 GB
- No support for full single server durability
- Requires more storage
- Installation Instructions on Paper

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Benchmarks

- Testsuite in .net in C#
- Tested on
  - PostgresSQL (32 bit), MongoDB (32 bit)
  - Microsoft Windows, Quad core notebook with SSD drive
- Bulk data insert
  - Speed of inserting lot of data objects
  - Object with location and multiple tags
  - Mongo: faster as it does not use transactions or ensure durability.
  - Verdict: Use for schema-free data insertion
- Tag Search
  - Testing join behavior, querying nested objects
    - Mongo much slower
    - Verdict: Don't use for complex queries
    - MongoDB doesn't support foreign keys and joins

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Dremel

- Dremel is a scalable, interactive ad-hoc query system for analysis of read only nested data.
- Novel Columnar storage format for nested Data.
- Outline Dremel's query language and execution
- How execution trees used in web search systems can be applied to database processing.

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Data Model

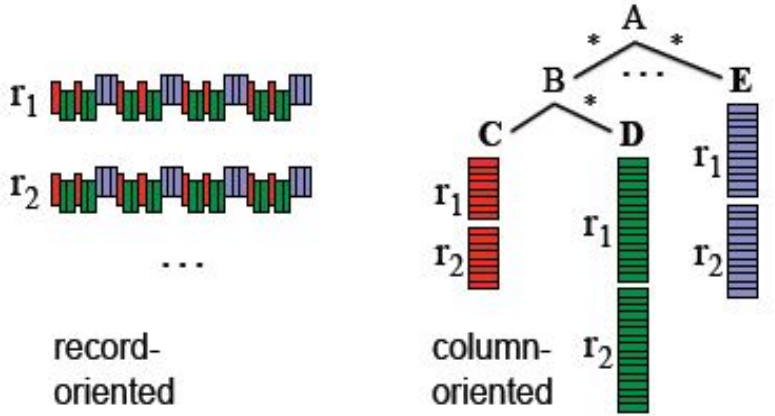
- Based on strongly-typed nested record
- T = dom | <A1 : T[*|?]..... An : T[*|?]?
  - Where T is an atomic type or a record type.
  - Atomic type can be integers, floating point numbers,strings
  - Records consist of one or multiple fields.

- Example: A document has a required integer DocId and optional links, containing a list of Forward and Backward entries holding DocIds of other pages.

- This data model backs a platform-neutral, extensible mechanism for serializing structured data.

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Nested Columnar Storage



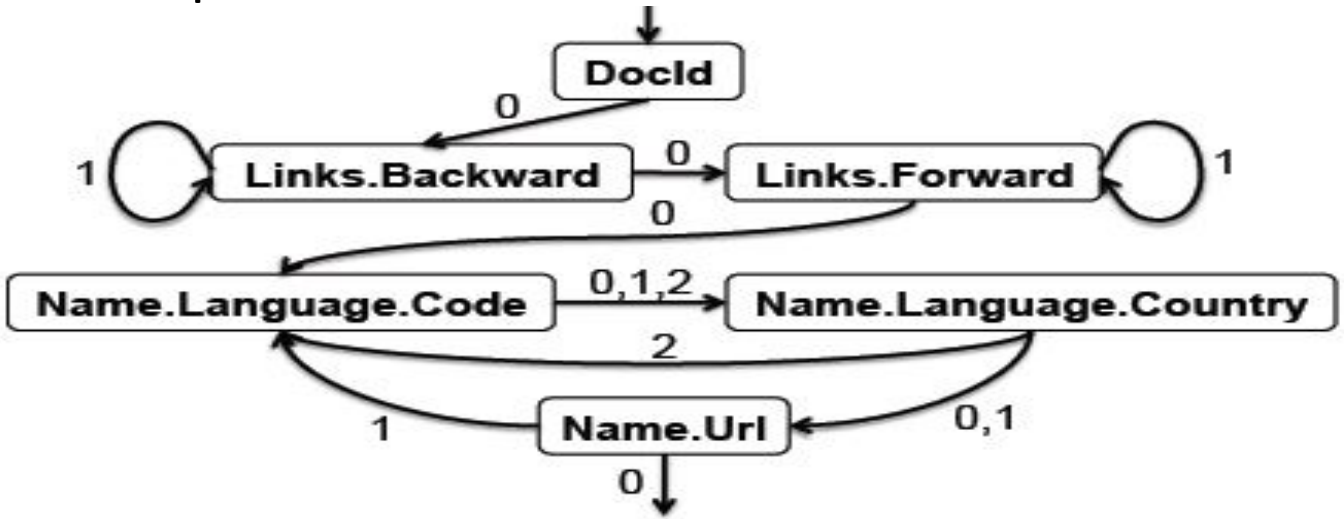
- Repetition and Definition Levels
  - It tells what repeated field in the field's path , the value has repeated.
- Splitting records into columns
  - The algorithm computes the level of each field values.
  - Repetition and definition levels may need to be computed even if field values are missing.

- All values of A.B.C is stored in a contiguous location
- Hence A.B.C can be retrieved without A.B.D or A.E

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Record Assemble FSM

- Create a finite state machine that reads the field values and levels for each field , and appends the values sequentially to the output records.



Query Language

- Based on SQL and designed to be efficiently implementable for columnar nested storage.
- Each SQL statement takes as input one or multiple tables and their schemas and produces a nested table and it's output schema.
- Language supports nested subqueries, inter and intra-record aggregation. Top-k joins etc.

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Query Execution

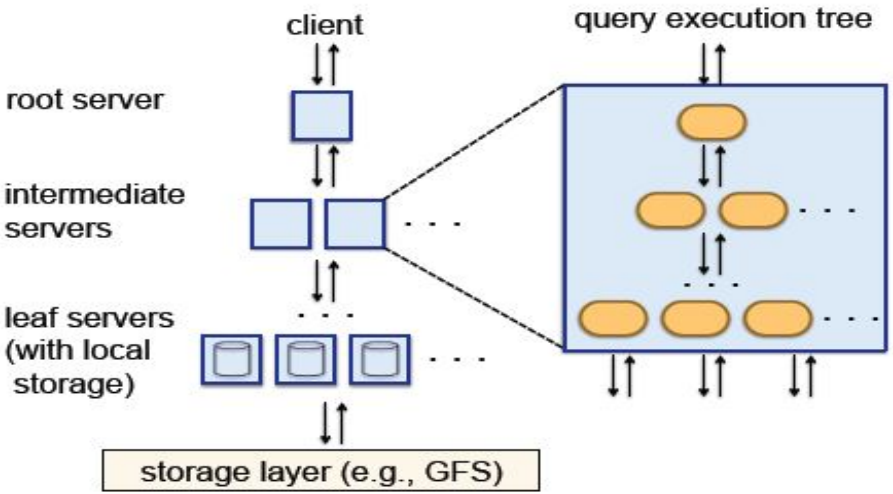


Figure 7: System architecture and execution inside a server node

- Uses multi-level serving tree to execute queries.
- A root server receives incoming queries, reads metadata from the tables, and routes the queries to the next level in the serving tree.
- The leaf servers communicate with the storage layer or access the data on local disk.

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Query Dispatcher

- Several queries are executed simultaneously.
- Leaf servers read stripes of nested data in columnar representation.
- Blocks in each stripe are prefetched asynchronously
- Each server has an internal execution tree which corresponds to a physical query execution plan.

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Benchmarks

- Bounding box location Search
  - Test for Geospatial usage
  - Tested with integers and geospatial data types
  - Postgres much faster
  - Verdict: Don’t use, slower than PostGIS
- Conclusion: Use for simple schema-less web applications

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Extra

- Mongo has leader election and replication management
- Accept data loss if your application doesn't need consistency
- Use WriteConcern.MAJORITY to reduce probability of write loss at the cost of performance
- Neither CP or AP system (Brewer's theorem)

http://aphyr.com/posts/284-call-me-maybe-mongodb

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.

Benchmarking Top NoSQL Databases

Introduction

- The best way to evaluate platform is to conduct a formal *Proof-Of-Concept(POC)* in the environment in which database will run

- This paper examines the performance of the top 3 NoSQL databases using the *Yahoo Cloud Serving Benchmark (YCSB)*

- 3 NoSQL databases are Apache Cassandra, Apache HBase and MongoDB

Benchmark Configuration (1/2)

- The tests ran in the cloud on Amazon Web Services (AWS) EC2 instances
  - The tests ran exclusively on m1 .xlarge size instances (15GB RAM and 4 CPU cores)
    - The instances use customized Ubuntu 12.04 LTS AMI's with Oracle Java 1.6 installed as a base.
  - The YCSB test was utilized for the various benchmark workloads
- Configuration

	Cassandra	**HBase**	**MongoDB**
Version	1.1.6	1.1.1	2.2.2
Node ID	(nodeID) * 2127/ (node count)	The head node on the latest stable (0.94.3)	The head node on start up

Benchmark Configuration (2/2)

- Testing parameters
  - Fieldcount = 20 to produce 2KB records
  - The request distribution = “zipfian” , workload6 request distribution = “latest”
  - The workloads that involves scans (4 and 5) max scan length were limited to 100
  - Max execution time was limited to 1 hour
  - Targeted 128 client threads per data instance (32 per CPU core)

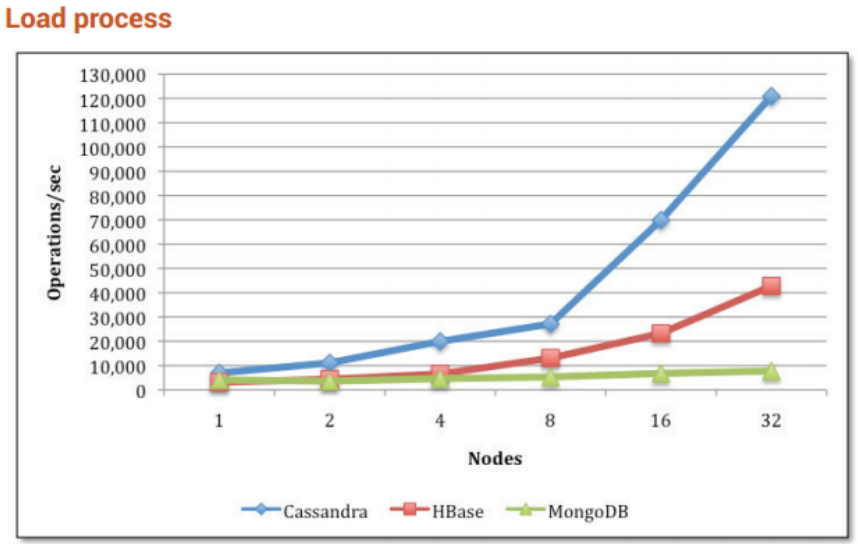
Date Nodes	Client Nodes	Total Records	Records/Client	Total Threads	Threads/Client
1	1	15M	15M	128	128
2	1	30M	30M	256	256
4	2	60M	30M	512	256
8	3	120M	40M	1024	341
16	6	240M	40M	2048	341
32	11	480M	43.6M	4096	372

Tested Workloads

- Read-mostly workload, based on YCSB's provided workload B:95% read to 5% update ratio
- Read/Write combination, based on YCBS's workload A:50% read to 50% update ratio
- Write-mostly workload: 99% update to 1% read
- Read/scan combination: 47% read, 47% scan, 6% update
- Read/write combination with scans: 25% read, 25% scan, 25% update, 25% insert
- Read latest workload, based on YCSB workload D:95% read to 5% insert
- Read-modify-write, based on YCSB workload F:50% read to 50% read-modify-write

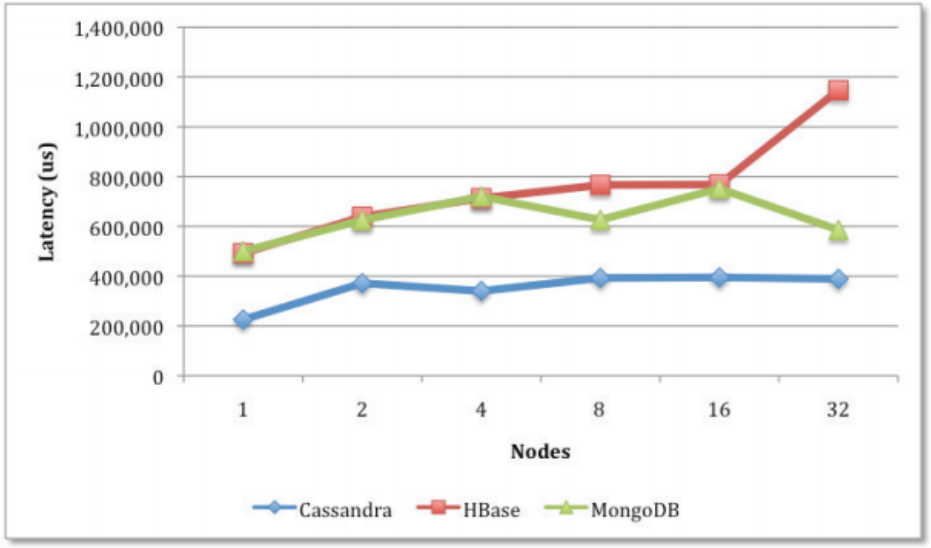
Throughput Result

- Operations-per-second (more is better), the raw numbers along with the percentage at which Cassandra outpaces HBase and MongoDB follows each graph
- For the load process, MongoDB was not able to scale effectively to 32 nodes and produced errors until the thread count was reduced to 20



Latency Result

- Less is better



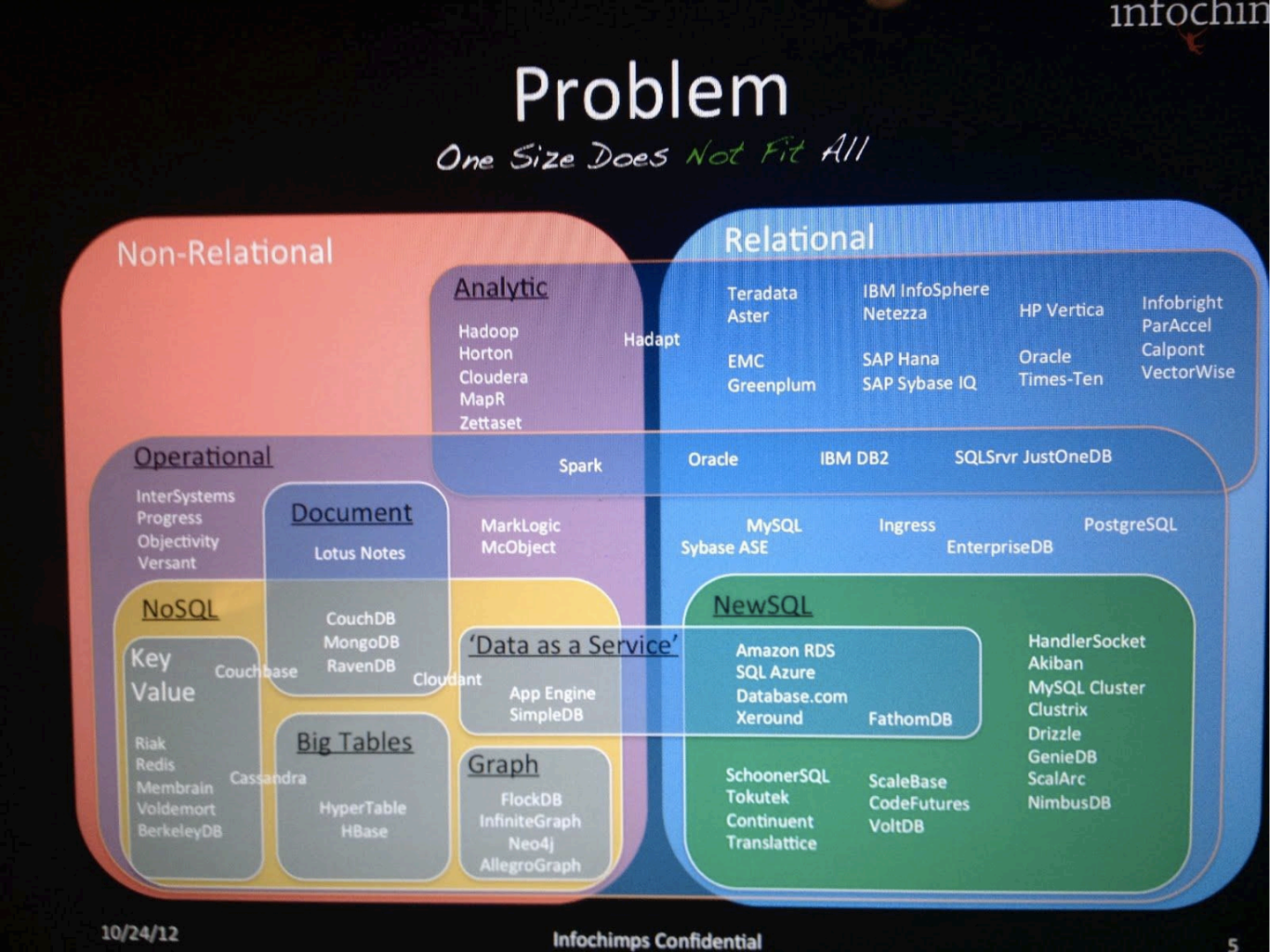
Shards	Cassandra	HBase	MongoDB	%/HBase	%/MongoDB
1	225509.64	492450.15	499810.74	-118.37%	-121.64%
2	371531.12	638811.96	625352.17	-71.94%	-68.32%
4	340136.80	713152.11	719959.11	-109.67%	-111.67%
8	392649.61	766141.42	626442.52	-95.12%	-59.54%
16	395099.39	768299.09	751722.25	-94.46%	-90.26%
32	388945.97	1147158.86	585785.52	-194.94%	-50.61%

Conclusion

- From an operations-per-second/throughput perspective, Cassandra proved to outdistance the performance of both HBase and MongoDB across all node configurations (1 to 32 nodes)

- Latency metrics for Cassandra proved to be the lowest across all workloads in the tests

- By conducting a formal POC in the environment in which the database will run and under the expected data and concurrent user workloads



One Size does not fit all

From:http://tcrn.ch/1x3RCSp

References

- 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. Appears in *Proceedings of the Symposium on Operating Systems Design and Implementation* (OSDI), 2007
- Bigtable: a distributed storage system for structured data. Appears in *Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation* (OSDI), 2006
- Dremel: Interactive Analysis of Web-Scale Datasets *Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis*
- The Google File System, Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
- PNUTS: Yahoo!'s hosted data serving platform
- Cassandra – A decentralized structured storage system
- Benchmarking top NoSQL databases
- MongoDB – An introduction and performance analysis

UC Irvine 50 1965 2015 Bright Past. Brilliant Future.