Systems Infrastructure for Data Science

Web Science Group
Uni Freiburg
WS 2014/15

Web Databases and NoSQL

Topics

- Web Databases: General Ideas
- Distributed Facilities in MySQL
- Cassandra
- Google BigTable/Hbase
- H-Store (VoltDB): OLTP

Web Applications and Databases

- (Social) Web Application:
  - End user facing
    - Users hate high response time
    - Non-professional users do simple operations (like/poke, comment, share, subscribe)
  - Interactive and in real-time
  - It is about information sharing => quite simple operations (no complex analytics) but very database-intensive
  - The number of users can be potentially high and can grow unexpectedly => easy to scale infinitely
- Traditional Enterprise Applications **and** Map-Reduce
  - Almost all the above points in reverse
- Real systems, different tradeoffs than research!

Web Applications and Databases: Requirements

- Support for simple operations
- Low response time
- 24/7 availability
- Easy to scale - Can you do it “at Facebook scale”?

MySQL
Distributed Facilities

- Represents most common “classical” distributed DB
- Used in many web data setups if relational features are needed
- Two relevant approaches:
  - MySQL Replication
  - MySQL Cluster

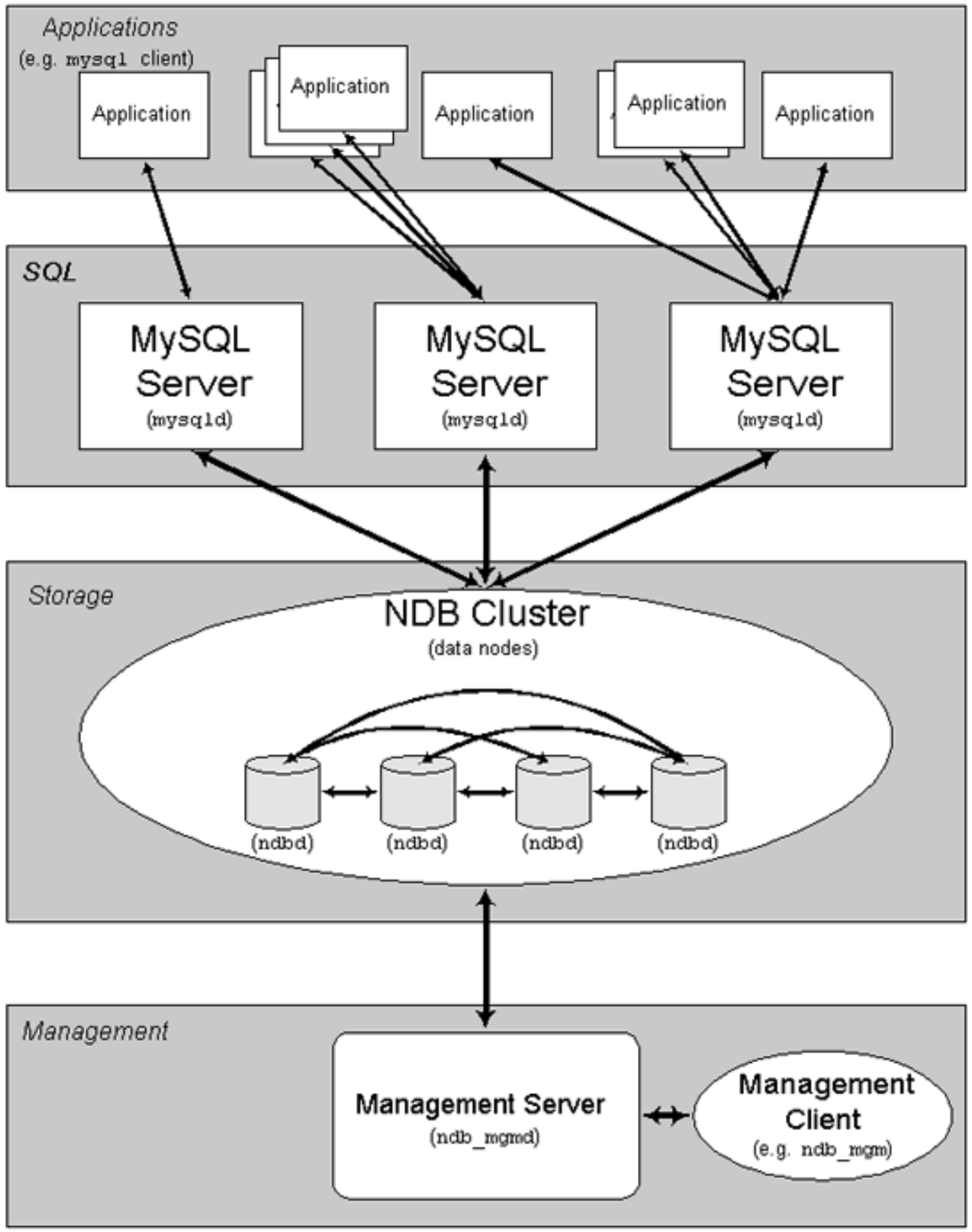
MySQL Replication

- One-way, asynchronous replication with single master and multiple slaves:
  - All updates are performed on the master
  - Updates are propagated from the master to slaves via log shipping (periodically in the background)
  - Queries can be performed on the master or slaves
  - Asynchronous => Stale data reads
- This approach is also called Hot Standby
- Benefits:
  - Scale query-intensive workload
  - Increase availability (switch from the master to a slave in case of the master failure)
  - Database backups using a slave server without disturbing the master

MySQL Cluster

- Shared-nothing high-available extension for MySQL
- Implemented by providing a new storage engine Networked Data Base (NDB) in addition to MyISAM and InnoDB

MySQL Cluster



MySQL Cluster

- Partitioning:
  - Data within NDB is automatically partitioned across the data nodes
  - Via hashing based on the primary key on the table
  - In the 5.1 release, users can define their own partitioning strategies
- Replication:
  - synchronous replication via two-phase commit

MySQL Cluster

- Query execution: distributed facilities are localized in the storage engine =>
  - Low-level operations are distribution-aware (e.g primary key lookup - contact a single node by hashing, index/table scan - sent in parallel to all the nodes) http://bit.ly/bezpxC
  - No distributed join supported: http://bit.ly/cxV9ZZ
- Hybrid Storage:
  - All indexed columns are stored in memory (distributed)
  - Non indexed columns can also be maintained in memory (distributed) or can be maintained on disk with an in-memory page cache

Cassandra

- Origins
- Implementation
  - Data distribution: partition and replication
  - CAP and consistency levels
  - Eventual consistency mechanisms: read repair and AE
  - Scaling
  - Load balancing
  - Gossip (mechanism to build peer-to-peer to achieve high availability avoiding masters)
- Data Model

Cassandra: Origins

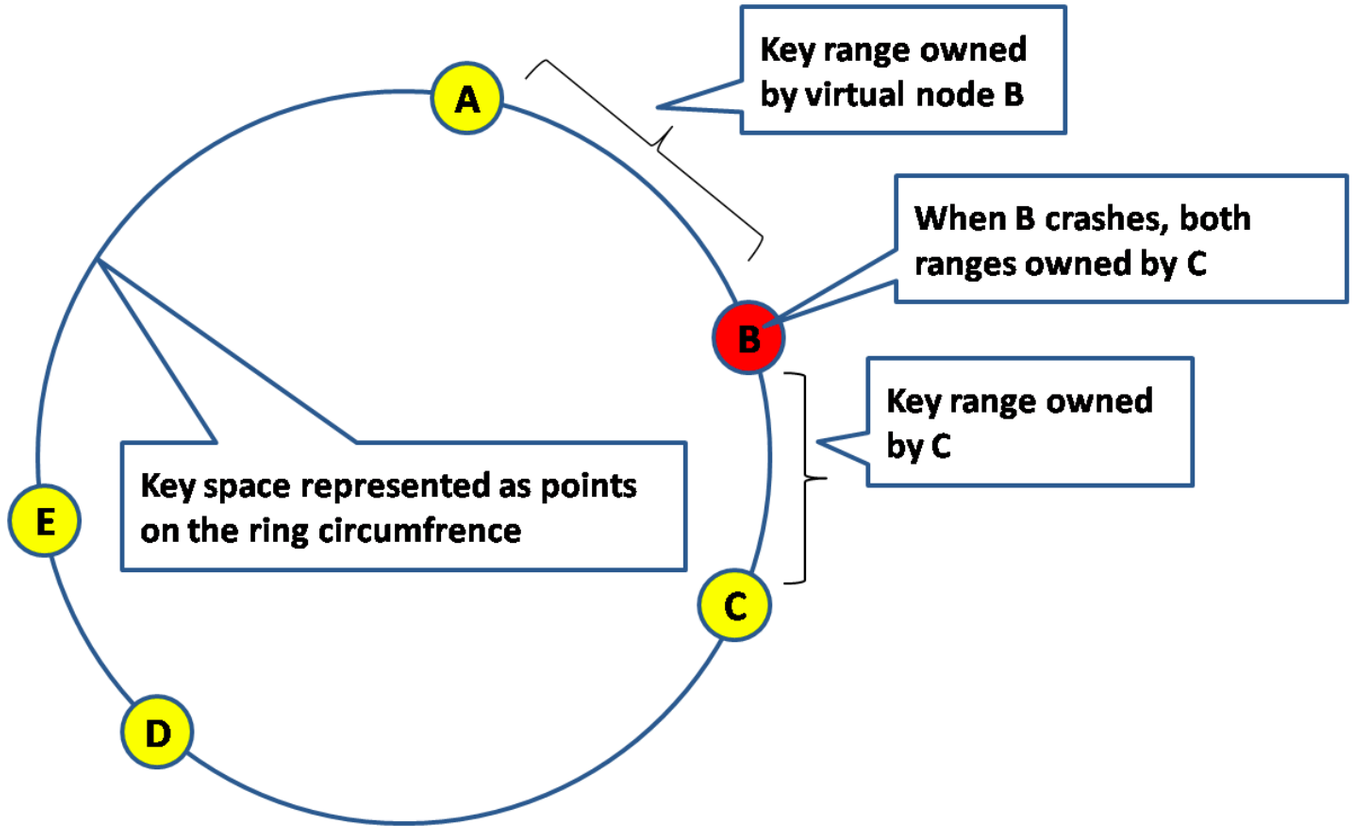
- Amazon Dynamo was introduced in 2007
  - Scalable and high available shopping cards
- Facebook implemented Cassandra
  - Inbox search
- Release open-source in 2008

Cassandra Data Model: Quick Introduction

- It is a key-value store distributed across nodes by key
  - Not a relational table with many columns, many access possibilities
  - Instead a key->value mapping like in a hash table
- A value can have a complex structure as it is inside the node - in Cassandra it is columns and super columns (explained later)

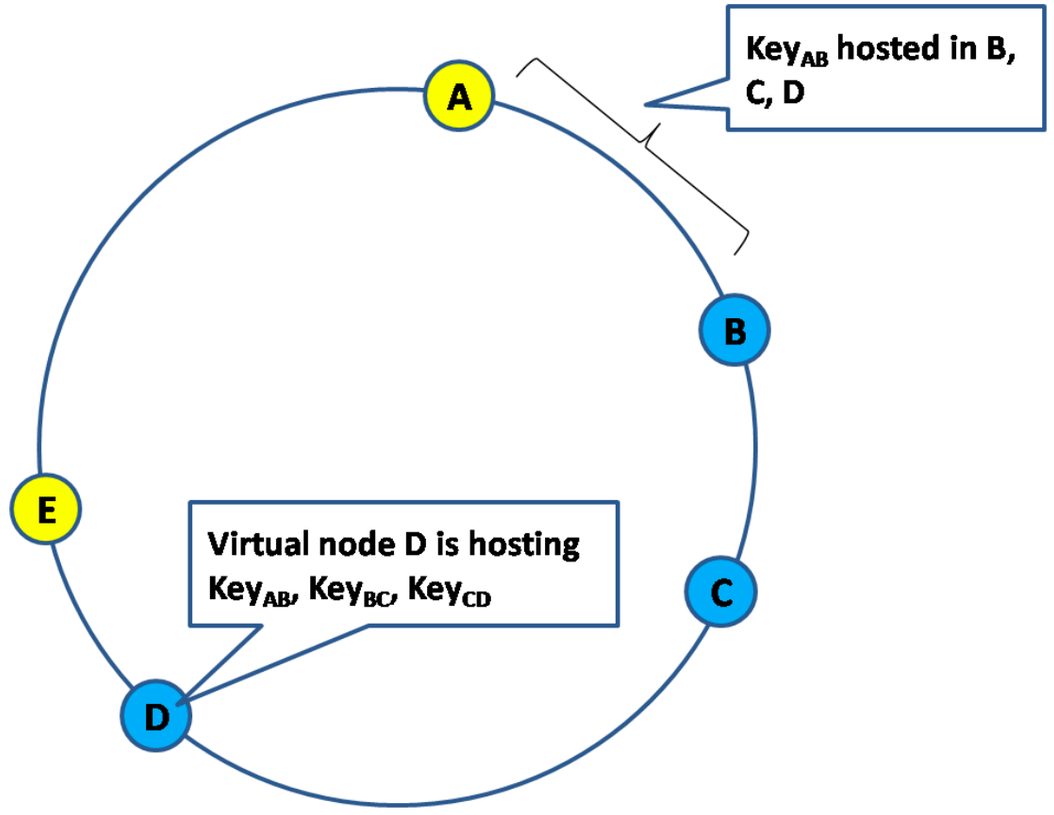
Data Partitioning: Consistent Hashing

- Problem with hashing: arrival or departure of a node requires global rehashing
- Idea: Hash keys and node IDs onto the same circled key space
- Advantage: Key redistribution happens only within the neighbor of the crashed node



Data Replication

- Why:
  - To achieve high availability data are replicated at N nodes
  - Improved performance by spreading workload across multiple replicas
- How:
  - Storing replicas on subsequent N nodes in the ring



Consistency Levels: Motivation

- Brewer's CAP Theorem - pick 2 out of 3:
  - Consistency (C) - You always read your previous writes
  - Availability (A)
  - Network partition tolerance (P)
- Options:
  - CA - Corruption possible if live nodes cannot communicate (network partition)
  - CP - Completely inaccessible if any nodes are dead
  - AP - Always available but may not always read most recent writes
- Let us make it tunable!
  - Cassandra prefers AP but makes "C versus A" configurable by allowing the user to specify a consistency level for each operation

Consistency Levels

- Parameters:
  - N - replication factor
  - W - number of replica nodes that must acknowledge the write
  - R - number of replica nodes that must respond to the read request
- Options:
  - W=1 => Block until first node written successfully
  - W=N => Block until all nodes written successfully
  - W=0 => Async write (cross fingers)
  - R=1 => Block until first node returns an answer
  - R=N => Block until all nodes return answers
  - R=0 => Does not make sense
- Note that it always reads/writes all replica nodes but waits for different numbers of responses.
- How to switch consistency on when you need it:
  - Quorum: W + R > N => Fully consistent database (i.e. you read your own previous writes) otherwise it might happen that you cannot see your previous write.
  - For example: R = N / 2 +1, W = N / 2 + 1 => Quorum achieved

Eventual consistency

- When W < N (not all replicas are updated) the update is propagated in background
- It is called Eventual Consistency
- Versions resolution:
  - Each value in a database has a timestamp => key, value, timestemp
  - The timestamp is the timestamp of the latest update of the value (the client must provide a timestamp with each update)
  - When an update is propagated, the latest timestamp wins (Could there be problems?)
- There are two mechanisms to propagate updates:
  - Read repair
  - Anti-Entropy (AE)

Eventual consistency: Read repair

- On client's read:
  - do reconciliation and write back if replicas are out of sync

Eventual consistency: Anti-Entropy

- AE is used to repair cold keys - keys that have not been read, since they were last written
- AE works as follows:
  - It generates Merkle Trees for tables periodically
  - These trees are then exchanged with remote nodes as a part of the Gossip conversation (explained later)
  - When ranges in the trees disagree, the corresponding data are transferred between replicas to repair those ranges
- Merkle Tree is a compact representation of data for comparison:
  - A Merkle tree is a hash tree where leaves are hashes of individual values. Parent nodes higher in the tree are hashes of their respective children. The principal advantage of Merkle tree is that each branch of the tree can be checked independently without requiring nodes to download the entire data set.

Update Idempotency

- If client observes an update failure it is still possible that this update has been executed
  - Because Cassandra does not support transactional rollback
- Examples:
  - N=3, W=2 but only one node is updated successfully => the client gets error => but this update is not rolled back from the node and will be propagated to the other replicas by read repair or AE
  - The whole update can be successfully executed but the return message is lost
- The client usually retries the failed update until it is successful => the same update can be executed several times!
- All updates should be idempotent (i.e. repeated update applications have the same effect as one)

Transaction Support

- There is no rollback support but Cassandra does support transactional atomicity, just eventually.
  - For a single operation: eventually update is propagated to all the replicas
  - For a set of operations: atomic batches (added in v1.2)
    Around 30% overhead for updates
- Instead of rollback (and possible block with 2PC), the client and async triggers retry until successful
- In terms of ACID:
  - Atomicity (eventual), Consistency (eventual), Isolation (none), Durability

Scaling

- Scaling is easy because of consistent hashing
- How to identify overloaded nodes:
  - Cassandra monitors the size of data stored at each node - a simple and good enough criteria to identify overloaded nodes
  - When a new node is added to the ring, Cassandra can choose the position for the new node automatically to unload the most overloaded node

Load Balancing

- The client can connect to any node in the ring, the node becomes the read/write coordinator
- It read/write the replicas according to the specified consistency level for this operation and replies to the client - one hop to access the data
- The client can query Cassandra for data distribution and cache it so that the client can connect to one of the replicas (it becomes the update coordinator) to avoid the hop

Gossip Protocol

- Cassandra is a peer-to-peer system
  - All nodes are equal and can process any client or maintenance operation
  - No masters
  - Helps to handle network partitions
- Every node has to know important information about every other node's state including which nodes are unreachable/down
- Gossip is a protocol to disseminate such information between nodes

Gossip Protocol: How it works

- Every second each node randomly chooses one live and one dead node and starts a state exchange round with them.
- It is proven that state is disseminated in O(log N) rounds where N is the number of nodes in the cluster.

Cassandra Data Model: Motivation

- Cassandra is a key-value database where data are distributed across nodes by key.
- Main idea: a value is stored on a single node, it can have a complex structure - dealing with the structure of a value does not involve expensive inter-node communications => operations are still single-node

Data Model (1)

- Data model is based on key-value model
  - A database consists of column families
  - Column family is a set of key-value pairs
  - An analogy with relational model:
    - Column family ~ Table
    - Key-value pair ~ Record

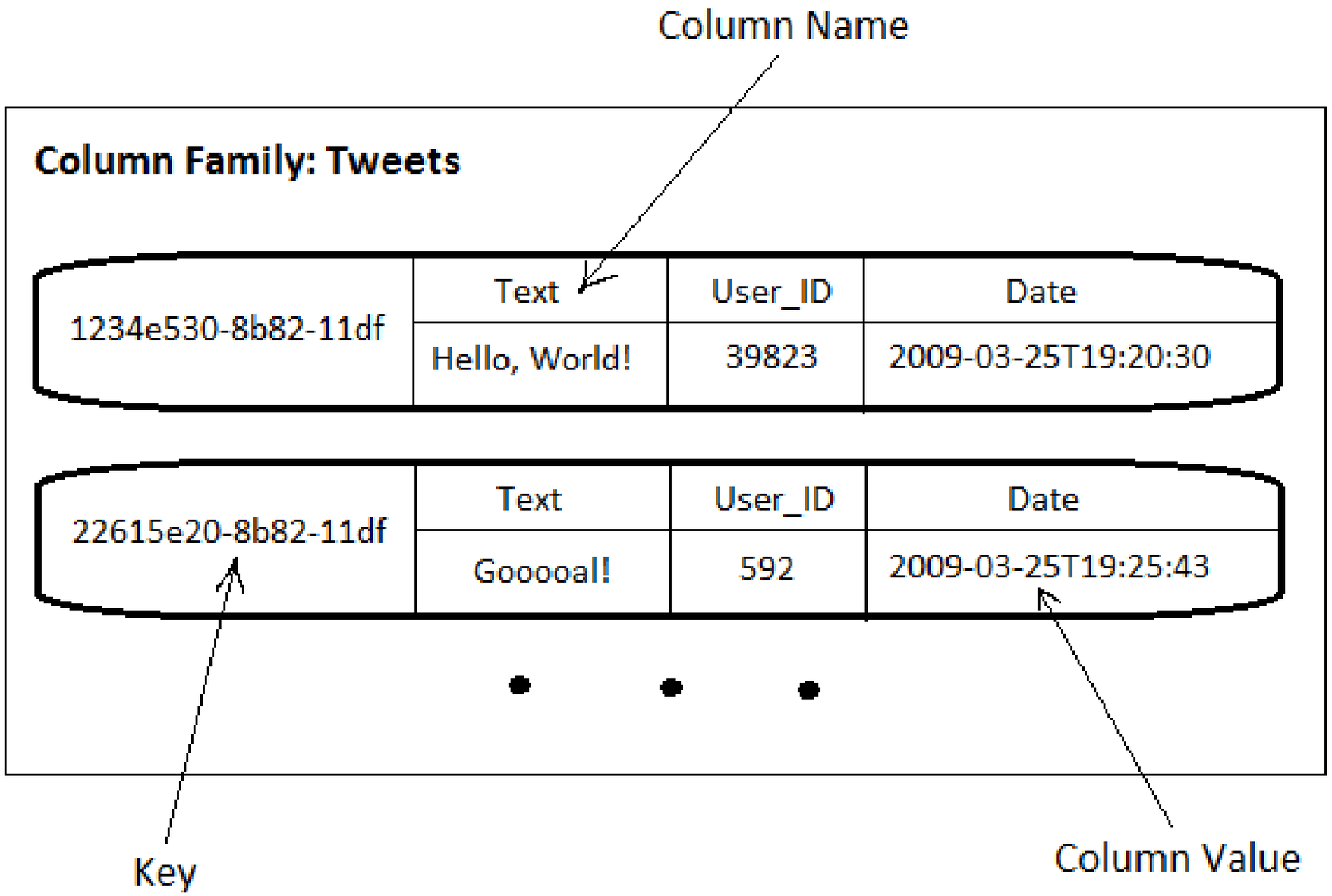
Data Model (2)

- The basic key-value model is extended with two levels of nesting:
  - First level: the value of a record is in turn a sequence of key-value pairs:
    - The key-value pairs are called columns. The Key is called column name. The Value is called column value
    - You can say: a record in a column family has a key and consists of columns
  - Second level (arbitrary): the value of the nested key-value pairs can be a sequence of key-value pairs as well:
    - When the second level of nesting, outer key-value pairs are called super columns with key being the name of the super column and inner key-value pairs are called columns

Data Model (3)

- The names of both columns and super columns can be used in two ways:
  - Names can play the role of attribute name (e.g. Email, Address, etc)
  - Names can be values. For example, Blog column family contains records where the record key is the blog identifier, column names are post identifiers, column values are the texts.
    - No restrictions on the number of columns/supercolumns - they should only fit a single node
    - Names are byte arrays so you can encode any value in it.
- Columns and super columns are stored ordered by names
  - Sorting behavior is specified by treating column names as Byte Type, ASCII Type, UTF8 Type, Lexical/Time UUID Type

Column Family Example: Columns are attributes

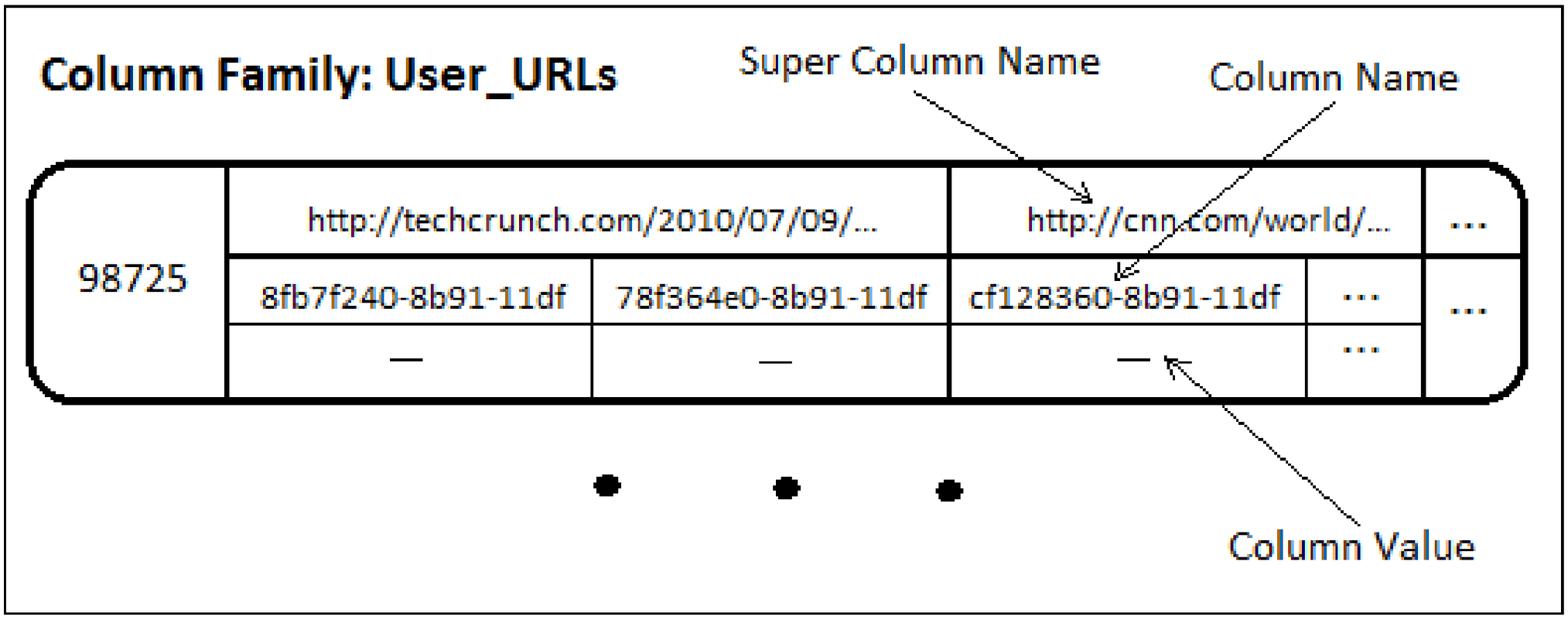


Column Family Example: Column names are values

Column Family: User_Timelines

39823	cef7be80-8b88-11df	1234e530-8b82-11df	...
	–	–	...
592	f0137940-8b8a-11df	22615e20-8b82-11df	...
	–	–	...

Super Column Family Example



BigTable (HBase)

- Distributed key-value store developed by Google (published in 2006)
- Open source counterpart - HBase hbase.apache.org

BigTable Data Model

- Cassandra borrowed the BigTable data model
- Cassandra versus BigTable data models
  - Cassandra supports Super Columns (additional level of nesting)
    - BigTable does not
  - BigTable stores versions of column values automatically that can be queried by timestamp ranges
    - In Cassandra you have to use TimeUUID for keys or column names explicitly to query by time (timestamps are used only for version reconciliations)
  - Cassandra supports various partition strategies (e.g. random or range)
    - In BigTable rows are ordered by key - range partitioning
- Summary: To an application, a table appears to be a list of tuples sorted by row key ascending, column name ascending and timestamp descending

BigTable Data Distribution

- Physically, tables are partitioned into row ranges called tablets (regions in HBase)
  - Tables are assigned and split/merged (for load balancing and space management) via a B+-tree-like structure
- Data replication is provided by GFS

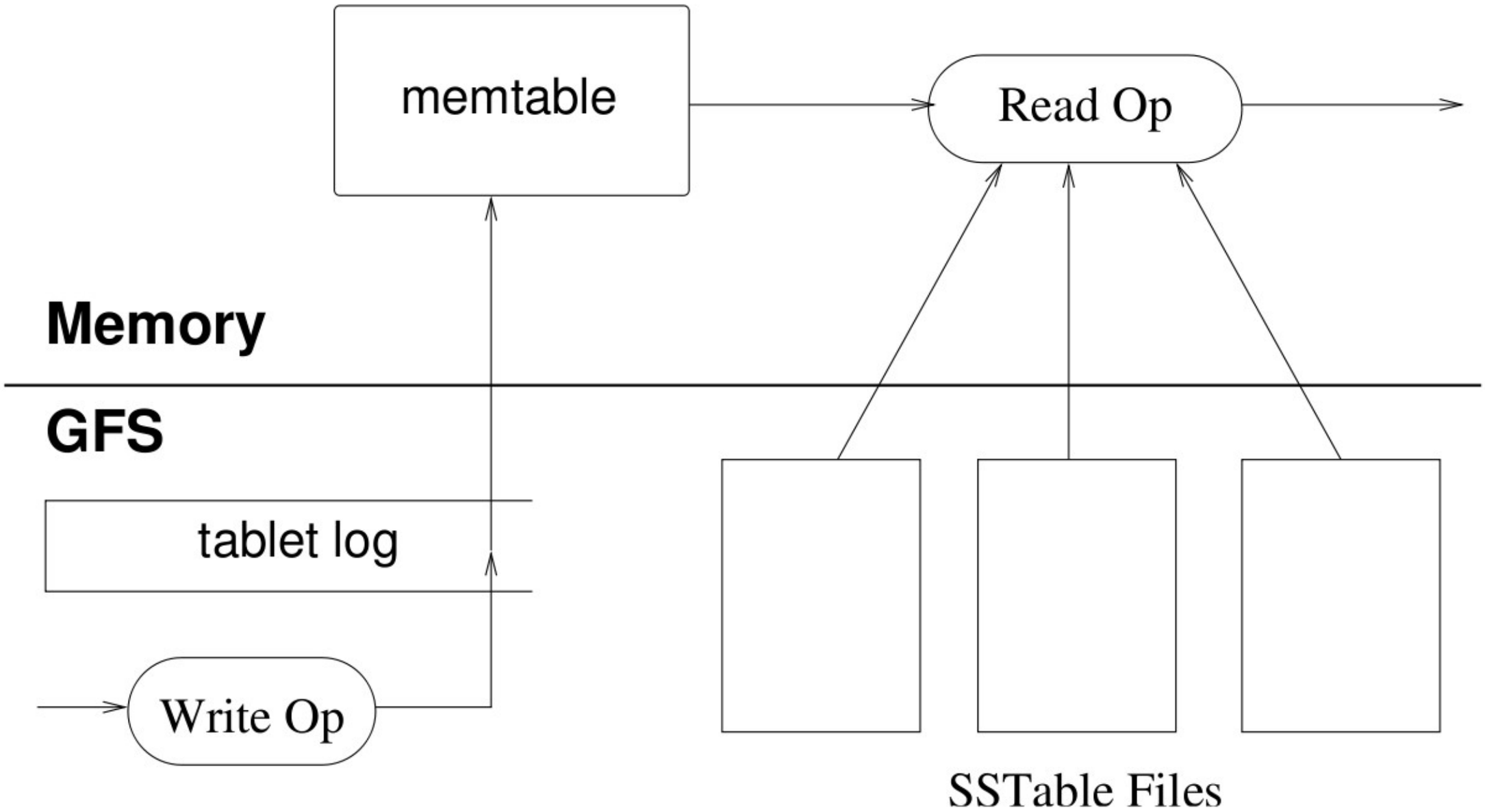
BigTable Architecture

- Single master, multiple tablet servers
  - The master: assigning tablets to tablet servers, detecting the addition/removal of tablet servers, and handling database schema changes
  - Tablet server: managing a set of tablets, handling reads and writes to the tablets, and splitting tablets that have grown too large
- Master and tablet servers share state and synchronize via a distributed lock and metadata storage service called Google Chubby
  - Bigtable uses Chubby for: ensuring there is one active master at any time, storing the location of data, discovering tablet servers, storing database schema, etc.
- Still single point of failure
  - Master failure does not cause BigTable unavailability. It can be restarted and recovers from Chubby. Tablet servers can still process user requests
  - If Chubby becomes unavailable, Bigtable becomes unavailable
- As in GFS, clients do not move through the master: clients cache tablet locations and communicate with tablet servers directly for reads/writes

BigTable Storage

- "Log-structured" storage based on Google File System
- How it works:
  - Collect insert (replace) and delete operations in buffer (aka memtable)
  - As the memtable is full, it is frozen, a new one is created, the frozen one is written to disk as SSTable (aka minor compaction)
  - SSTables are merged periodically in the background (aka major compaction)
  - Read operation has to merge SSTables and memtable during its execution
  - SSTable and memtable are sorted => the merge (during compaction or read) can be done efficiently
- Pros and cons:
  - Fast writes but slow reads
  - Can be easily implemented onto a file system without random writes (BigTable on GFS, HBase on Hadoop FS)
- memtable/SSTable storage is used in Cassandra but implemented from scratch (no GFS)
- BigTable/HBase provides good integration with MapReduce as data are stored in GFS

BigTable Storage



HBase

- Distributed column-oriented database built on top of HDFS
  - modeled after Google's BigTable
  - not relational, no support for SQL
  - real-time read/write random access to very large data sets

- Main design goal: Scale linearly by adding nodes
  - sparsely-populated tables
  - billions of rows
  - millions of columns
  - automatic horizontal partitioning/replication across a cluster

Data Model

- Tables consist of rows and column families.
- Rows are sorted by key.
- Rows are horizontally partitioned, which form the basic unit of data distribution and load balancing.
- Columns are grouped into families, which form the basic unit of access control.
- Columns that belong to a given column family can be added on the fly.
- Cells are versioned by timestamp.

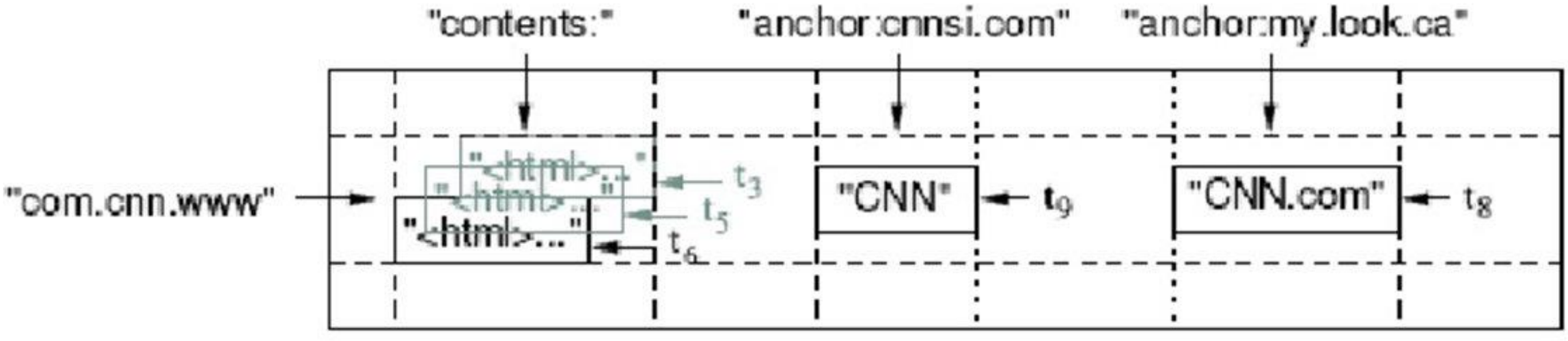
Data Model

- Multi-dimensional sorted map
- Indexed by a row key, column key, and a timestamp
- Each value in the map is an uninterpreted array of bytes.

- (row: string, column: string, time:int64) -> string

BigTable - Webtable Example

- A large collection of web pages
  - reverse URLs as row keys
  - other info about web pages as column names
  - web page contents in “contents” column family, versioned with the timestamps when they were fetched



Rows

- Arbitrary strings as row keys
- Lexicographic order by row key
- Atomic reads and writes to each row (single-row transactions)
- Row ranges are dynamically partitioned into “tablets” (“regions” in HBase), which form the unit of distribution and load balancing.
- Row keys affect locality of data access.
  - Example: Reversed URL’s in the Webtable example help group pages of the same domain together into contiguous rows.

Column Families

- Column families define units of access control.
  - Different sets of columns may have different properties and access patterns.
  - Column families are stored separately on disk.
  - Tables are configurable by family (e.g., version retention policy, compression, cache priority).
- Unbounded number of columns, smaller number of rarely changing column families.
- Column keys named as: family:qualifier
  - In the Webtable example:
  - anchor:<name of referring site>

Timestamps

- Each cell in a table can contain multiple versions of the same data, which are indexed by timestamp.
- Timestamps can be assigned by the system or by client apps.
- Versions are stored in decreasing timestamp order.
- Garbage collection:
  - Keep last n versions of a cell.
  - Keep last n days' versions of a cell.
- In the Webtable example: Keep only the most recent 3 versions of every page's content.

H-Store (VoltDB)

- Designed and Prototyped in MIT by Michael Stonebraker and others as part of "one size does not fit all"
- Productized as VoltDB (www.voltdb.com)
  - Available for download
- Applies some of the design ideas studied so far to OLTP (= transactional workloads)
- Breaks several assumptions how do DB design with updates

OLTP Design Considerations

- Main memory database
  - The overwhelming majority of OLTP databases are less than 1 Tbyte in size and growing in size quite slowly
  - Now machines with 100 Gbytes are becoming common
- Single-threaded execution model
  - Multiple-threads-per-CPU model helps to keep CPU busy while waiting for disk I/O or user input (aka disk and user stalls)
  - Multi-threaded model is also required in presence of long-running transactions not to block the system
  - But:
    - No disk I/O because main memory
    - No ad-hoc queries
    - OLTP transactions are lightweight (e.g. the heaviest transaction in TPC-C reads about 400 records - less than one millisecond for in-memory database)
  - Use single-thread-per-CPU:
    - To avoid threads synchronization overhead

OLTP Considerations (Cont.)

- Cost-effective High Availability
  - Currently, many systems implement a hot standby via log shipping:
  - Peer-to-peer (all replicas can do any operation) is more cost-effective
- No knobs - easy to design and maintain
  - Personnel now costs more than hardware/software

H-Store Implementation

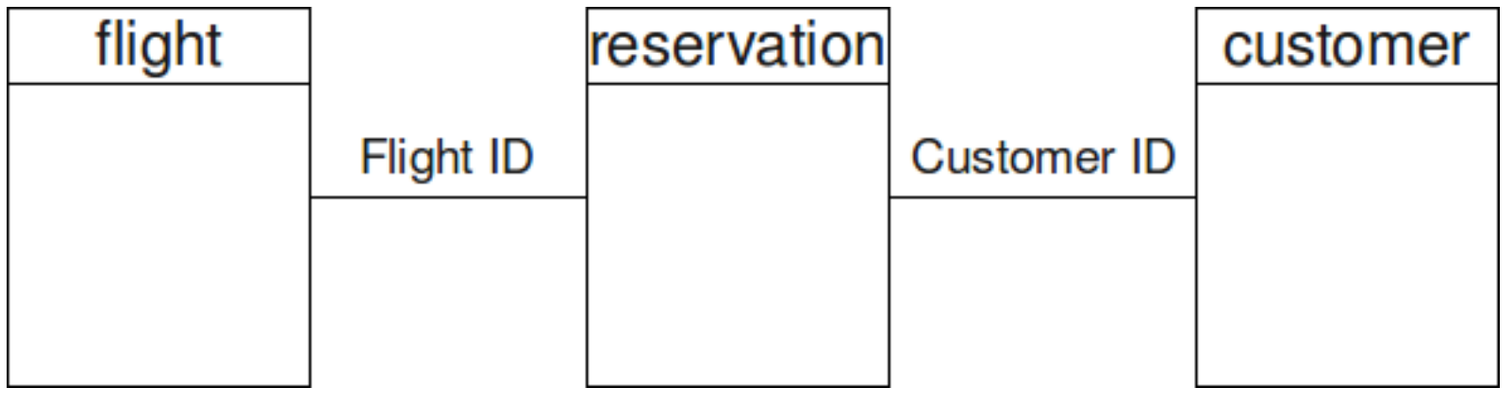
- Shared-nothing architecture: data are partitioned and replicated across a cluster of nodes
- Main memory storage for tables and B-tree indexes on each node
- Each physical node is decomposed into a number of local sites, one for each available core:
  - each site is like independent physical site with its own indexes, table storage, etc
  - in particular main memory on the node is partitioned among local nodes - no shared memory
  - each site has a dedicated CPU, is single threaded and performs transaction sequentially (!)

H-Store Implementation (cont.)

- Supports distributed transactions but prefers single-sited transactions
- Client can only call stored procedures: execute trans_name (parameter_list)
  - To avoid costly communication rounds over network - no JDBC/ODBC
- All transactions are known in advance - not designed for ad-hoc transactions
  - Data are partitioned/replicated to promote single-sited transactions wherever possible
  - From docs: “In fact, for 20 years or more OLTP application designers have used these design principles to get the most out of commercial database products”

Example: Flight Reservation Database

- The goal: the most frequent transactions should be single-sited
- For example, let us assume you are designing a flight reservation system with the following schema



- Flight (FlightID, DepartTime, Origin, Destination, NumberOfSeats, PRIMARY KEY(FlightID));
  Reservation (ReserveID, FlightID, CustomerID, Seat, Confirmed, PRIMARY KEY(ReserveID));
  Customer (CustomerID, FirstName, LastName, PRIMARY KEY(CustomerID));

Example: Flight Reservation Database

- In addition to schema, consider: expected volume and estimated workload

- Flights: 2,000
- Reservations: 200,000
- Customers: 1,000,000

Use case	Frequency
Look up a flight (by origin and destination)	10,000/sec
See if a flight is available (has sufficient space)	5,000/sec
Make a reservation	1,000/sec
Cancel a reservation	200/sec
Look up a reservation (by reservation ID)	200/sec
Look up a reservation (by customer ID)	100/sec
Update flight info	1/sec
Take off (close reservations)	1/sec

Example: Flight Reservation Database

- Reservation table:
  - Partition by Reservation ID is not good because there are only two transactions keyed by Res. ID that are infrequent:
    - Looking up a reservation by ID (200/sec)
    - Canceling a reservation (200/sec)
  - Partition by Flight ID
    - See if a flight is available (5,000/sec)
- Customer table:
  - Partition by Customer ID as it is usually accessed by ID
- Flight table:
  - Has the most frequent access (10k/sec)
  - However, these transactions may involve any combination of three columns: the point of origin, the destination, and the departure time => not easy to choose a column for partitioning
  - On the other hand: it is very small and read-only (except add flights and take off)
  - Replicate to all sites => 10k/sec transactions can be distributed across all sites

VoltDB versus Cassandra

- Benchmark by VoltDB: http://voltdb.com/blog/key-value-benchmarking
- A column family with records: 50 byte key + 50 columns (index as name + integer value)
- Load 500,000 records
- Do:
  - Randomly select 1 key K and 2 columns A and B
  - Read the value from the record K and column A
  - If the value is odd, write a new random value to the record K and the column B

VoltDB versus Cassandra:

Configuration	VoltDB	Cassandra	Throughput Increase
1 node	111,000	24,200	4.6x
3 nodes w/o Replication	293,000	38,900	7.5x
3 nodes w/ Replication	176,000	24,700	7.1x

Summary

- Web Databases also aim for massive scalability
- Make different tradeoffs:
  - Achieve very fast response time
  - Avoid blocking for updates
  - Only provide minimal operations
  - Compromise on consistency