TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

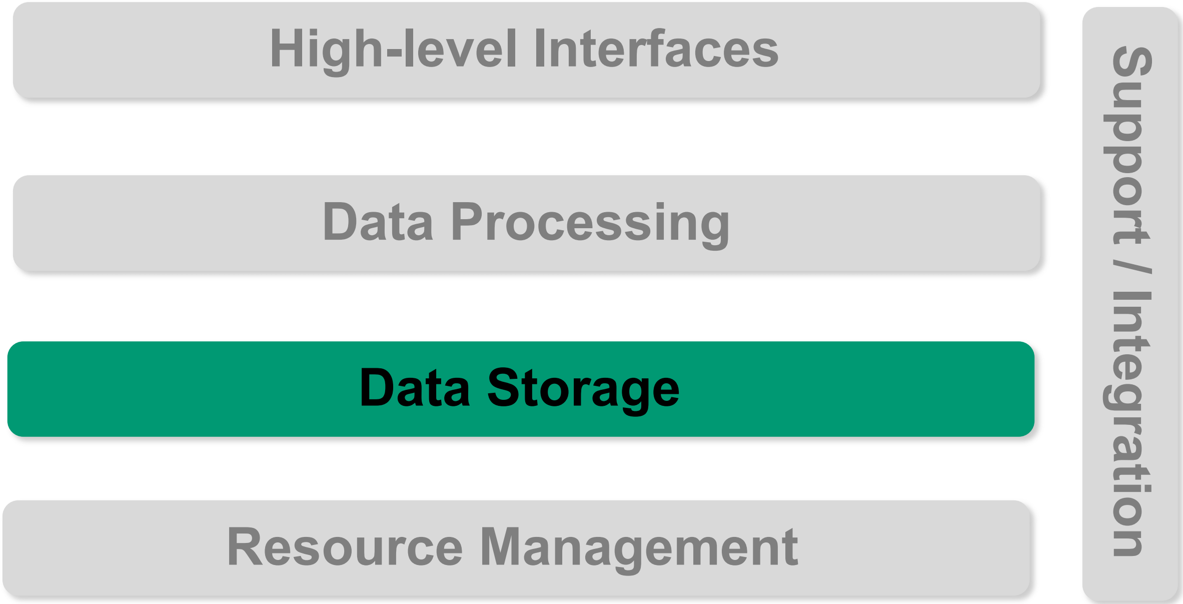
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

NoSQL Data Stores

Corso di Sistemi e Architetture per Big Data
A.A. 2023/24
Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

The reference Big Data stack



Traditional RDBMSs

- Relational DBMSs (RDBMSs)
  - Traditional technology to store structured data in web and business applications
- SQL is good
  - Rich language and toolset
  - Easy to use and integrate
  - Many vendors
- RDBMSs promise ACID guarantees

ACID properties

- **A**tomicity
  - All statements in a transaction are either executed or the whole transaction is aborted without affecting the database: “all or nothing” rule that is, transactions do not occur partially
- **C**onsistency
  - A database is in a consistent state before and after a transaction; it refers to the correctness of a database
- **I**solation
  - Transactions cannot see uncommitted changes in the database (i.e., the results of incomplete transactions are not visible to other transactions)
- **D**urability
  - Changes are written to disk (i.e., non-volatile memory) before a database commits a transaction so that committed data cannot be lost if a system failure occurs

RDBMS constraints

- Domain constraints
  - Restrict domain or set of possible values for each attribute
- Entity integrity constraints
  - No primary key value can be null
- Referential integrity constraints
  - To maintain consistency among tuples in two relations: every value of one attribute of a relation should exist as a value of another attribute in another relation
- Foreign key
  - To cross-reference between multiple relations: it is a key in a relation that matches the primary key of another relation

Pros and cons of RDBMS

Pros

✓ Well-defined consistency model

✓ ACID guarantees

✓ Relational integrity maintained through entity and referential integrity constraints

✓ Well suited for OLTP apps

✓ Sound theoretical foundation

✓ Stable and standardized DBMSs available

✓ Well understood

Cons

✗ Performance as major constraint, scaling is difficult

✗ Limited support for complex data structures

✗ Complete knowledge of DB schema required to build new queries

✗ Commercial DBMSs are expensive

✗ Some DBMSs have field size limits

✗ Data integration from multiple RDBMSs can be cumbersome

RDBMS challenges

- Workload spikes
  - Internet-scale data size
  - High read-write rates
  - Frequent schema changes

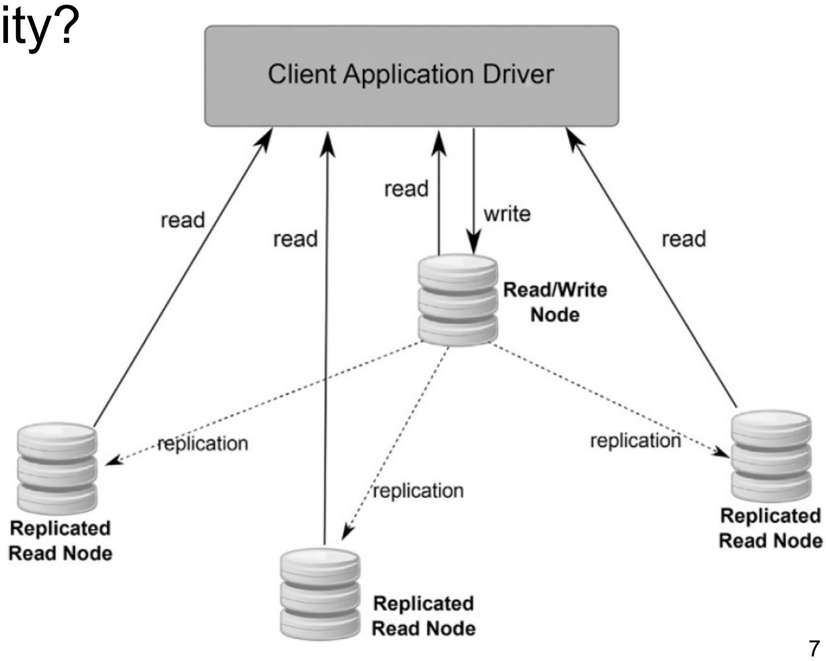
- Let’s scale RDBMSs
  - But they were not designed to be distributed
- How to scale RDBMSs?
  - Replication
  - Sharding

Replication

- Primary backup with master/worker architecture

✓ Replication improves read scalability

✗ Write scalability?



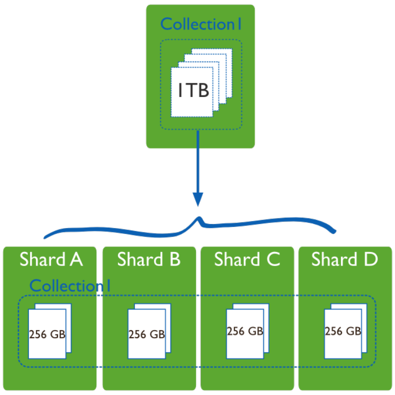
Sharding

- Horizontal partitioning of data across servers

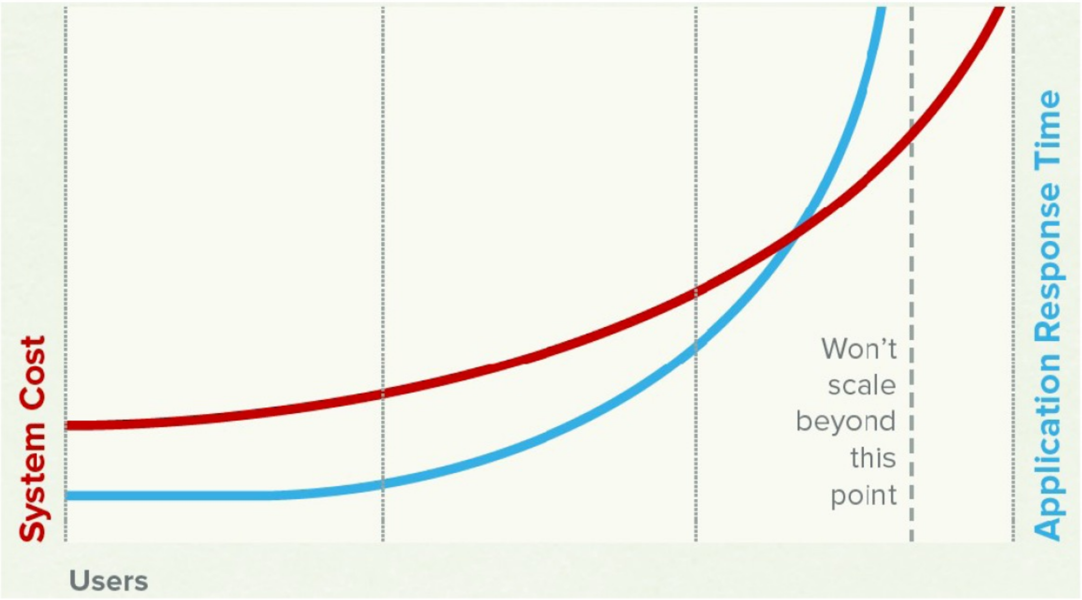
✓ Read and write operations scale

✗ Cannot execute transactions across shards (partitions)

- Consistent hashing can be use to determine *which* server any shard is assigned to
  - Hash both data and server using the same hash function in the same ID space



Scaling RDBMSs is expensive and inefficient



Source: Couchbase technical report

NoSQL data stores

DO YOU HAVE ANY EXPERTISE IN SQL?
NO
geek & poke
DOESN'T MATTER. WRITE: "EXPERT IN NO SQL"

Leverage the NoSQL boom

- NoSQL = **Not Only SQL**
  – SQL-style querying is not the crucial objective

NoSQL data stores: main features

- Support flexible schema
  – No requirement for fixed rows in a table schema
  – Well suited for agile development process
- Support horizontal scaling
  – Partitioning of data and processing over multiple nodes
- Provide high availability
  – By replicating data in multiple nodes, often geo-distributed
- Mainly utilize shared-nothing architecture
  – With exception of graph-based databases
- Avoid unneeded complexity
  – E.g., elimination of join operations
- Support weaker consistency models
  – **BASE** rather than ACID: compromising reliability for better performance

ACID vs BASE: ACID

- Two design philosophies at opposite ends of the consistency-availability spectrum
  - Keep in mind **CAP theorem**

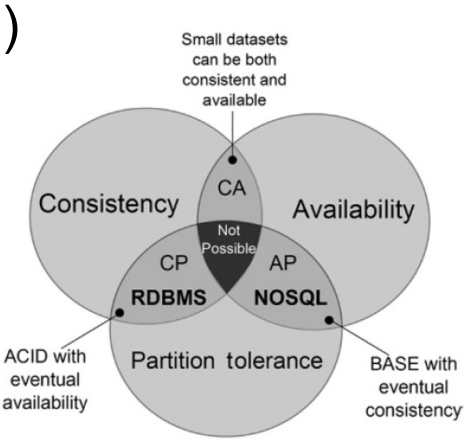
    *Pick two of **C**onsistency, **A**vailability and **P**artition tolerance*
- ACID: traditional approach for RDBMSs
  - *Pessimistic* approach: prevents conflicts from occurring
    - Usually implemented with write locks managed by system
    - Leads to performance degradation and deadlocks (hard to prevent and debug)
  - Does not scale well when handling petabytes of data (remember of latency!)

ACID vs BASE: BASE

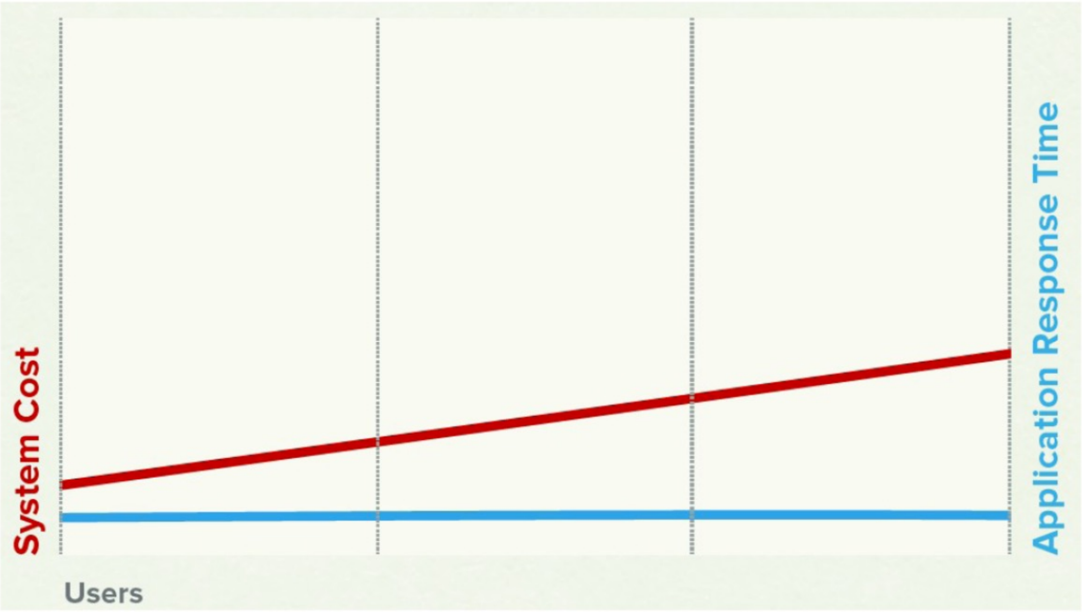
- **BASE**: **B**asically **A**vailable, **S**oft state, **E**ventual consistency
  - **B**asically **A**vailable: the system is available most of the time and there could exist a subsystem temporarily unavailable
  - **S**oft state: data is not durable that is, its persistence is in the hand of the user that must take care of refreshing it
  - **E**ventually consistent: the system eventually converges to a consistent state
- Optimistic approach
  - Lets conflicts occur, but detects them and takes action to sort them out: how?
    - *Conditional updates*: test the value just before updating
    - *Save both updates*: record that they are in conflict and then merge them

NoSQL and consistency

- Biggest change from RDBMS
  - RDBMS: strong consistency
  - Traditional RDBMS are either CA systems or CP systems (depending on configuration)
- Majority of NoSQL systems provide eventual consistency (i.e., AP systems)
- Some NoSQL systems provide strong consistency or *tunable* consistency
  - E.g., Cassandra and MongoDB


Small datasets can be both consistent and available
CA
Consistency
Availability
Not Possible
CP
AP
RDBMS
NOSQL
Partition tolerance
ACID with eventual availability
BASE with eventual consistency

NoSQL cost and performance



Source: Couchbase technical report

Pros and cons of NoSQL

Pros

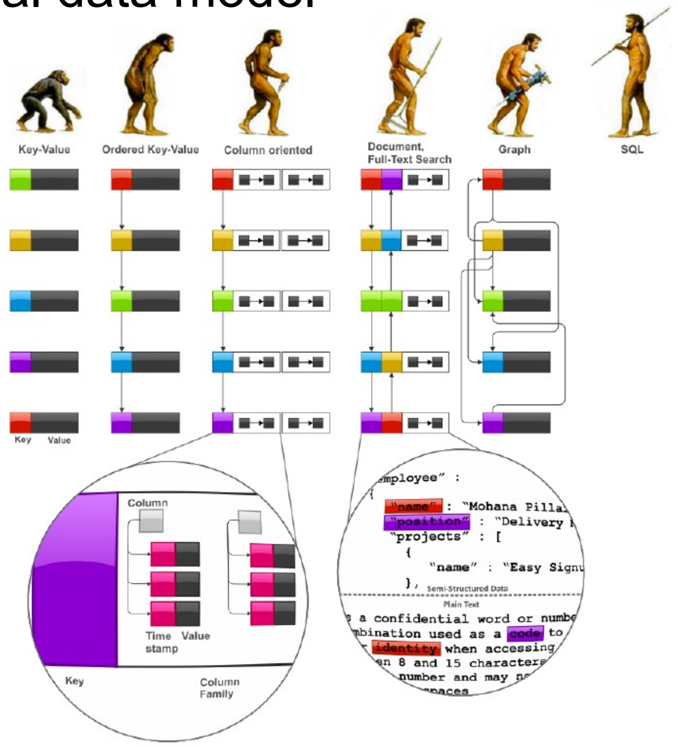
- ✓ Easy to scale-out
- ✓ Higher performance for massive data scale
- ✓ Allow data sharing across multiple servers
- ✓ HA and fault tolerance provided by data replication
- ✓ Most are either open-source or cheaper
- ✓ Support complex data structures and objects
- ✓ No fixed schema, support unstructured data
- ✓ Fast retrieval of data, suitable for real-time apps

Cons

- ✗ Many do not support ACID, less suitable for OLTP apps
- ✗ No common data storage model -> no well-defined approach for design
- ✗ Lack of standardization (e.g., standard query language)
- ✗ Many do not support join ops
- ✗ Lack of reference model can lead to solution lock-in

NoSQL data models

- A number of largely diverse data stores not based on relational data model



NoSQL data models

- *Data model*: set of constructs for representing information
  - Relational model: tables, columns and rows
- *Storage model*: how the data store management system stores and manipulates data internally
- A data model is usually independent of the storage model
- Data models for NoSQL systems:
  - Aggregate-oriented models: **key-value (KV)**, **document**, and **column-family**
  - **Graph-based** models

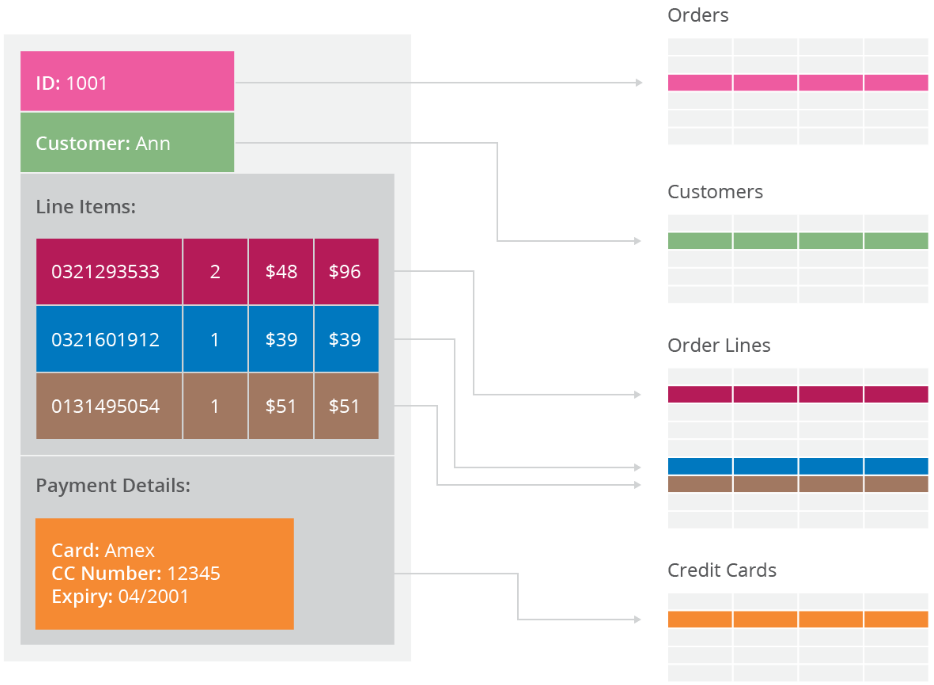
Aggregates

- Data as single unit with a complex structure
  - More structure than just a set of tuples
  - E.g., complex record with simple fields, arrays, records nested inside
- Aggregate pattern in Domain-Driven Design
  - Cluster of domain objects that we treat as a single unit (e.g., order and its items, playlist and its songs)
  - Unit for data manipulation and consistency management
- Advantages of aggregates
  - Easier for application programmers to work with
  - Easier for data store systems to handle ops

Read www.thoughtworks.com/insights/blog/nosql-databases-overview

Aggregates: example

- With NoSQL
- With RDBMS



ID: 1001

Customer: Ann

Line Items:

0321293533 2 $48 $96

0321601912 1 $39 $39

0131495054 1 $51 $51

Payment Details:

Card: Amex
CC Number: 12345
Expiry: 04/2001

Orders

Customers

Order Lines

Credit Cards

Transactions?

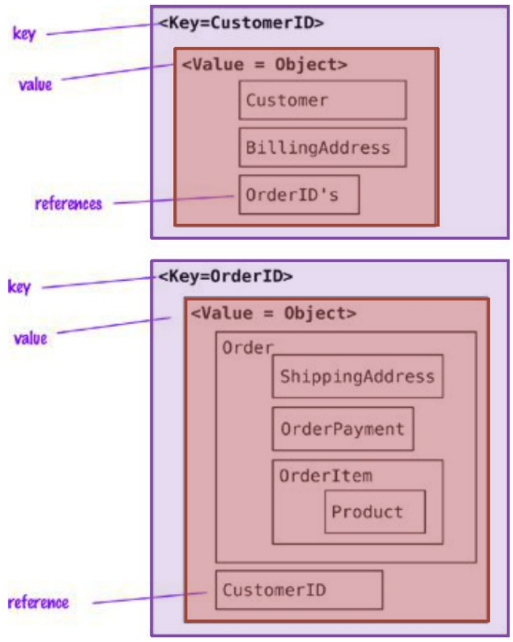
- Relational databases have ACID transactions
- Aggregate-oriented data stores
  - Support atomic transactions, but only *within* single aggregate
  - Most data stores don’t have ACID transactions that span multiple aggregates
    - In case of update over multiple aggregates: possible inconsistent reads

  ☞ Take it into account when deciding how to aggregate data
- Graph databases tend to support ACID transactions

Key-value (KV): data model

- Simple data model: data is represented as a schema-less collection of key-value pairs
  - Associative array (map or dictionary) as fundamental data model
- Strongly aggregate-oriented
  - Lots of aggregates
  - Each aggregate has a key
- Data model:
  - Set of <key, value> pairs
  - Value: aggregate instance
- Aggregate is opaque to data store
  - Just a big blob of mostly meaningless bits
- Access to aggregate: lookup based on its key
- Richer data models can be implemented on top

KV: data model example



KV: types of data stores

- Some data stores support key ordering
  - Data is stored sorted by key in a particular order (e.g., lexicographic) to handle keys more easily
- Some maintain data in RAM, while others employ HDDs, SSDs or even flash memory
- Some let developers implement user-defined functions (UDFs) to extend the data store processing capabilities
- Wide range of consistency models

KV: consistency

- Consistency ranges from weak (e.g., eventual) to strong (e.g., serializability)
  - Serializability: guarantee about transactions over multiple items
    - It guarantees that the execution of a set of transactions (with read and write operations) over multiple items is equivalent to some serial execution (total ordering) of the transactions
    - Gold standard in DB community: serializability is the traditional Isolation in ACID
  - Examples:
    - AP: Dynamo, Riak KV
    - CP: Redis, Berkeley DB

KV: query features

- Only query by the key!
  - There is a key and there is the rest of the data (the value)
- Basic ops: put(key,value), get(key), delete(key)
- Most KV data stores provide access operations on groups of related key-value pairs
- Cannot lookup for some attribute of the value
  - E.g., KV stores usually do not have a WHERE clause such as RDBMSs or if they do, it requires a slow scan of all values
- The key needs to be suitably chosen
  - E.g., session ID for storing session data
- What if we don't know the key?
  - Some KV system allows to search inside the value using a full-text search (e.g., using Apache Solr)

KV: suitable use cases

- Session info in web app
  - Each user session has a unique id: session id as key
  - Store session data using a single put, retrieve using get
- User profile and preferences
  - Almost every user has a unique user id, username, …, as well as preferences such as language, list of searched and recommended, …
  - Put preferences of a user into the value, so getting takes a single operation
- Shopping cart data
  - Put shopping information into the value, whose key is the user id
- Product recommendations

KV: products

- Amazon’s Dynamo is the most notable example
  - Riak KV: open-source implementation
- Other KV data stores include:
  - Amazon DynamoDB: data model and name from Dynamo, but different implementation
  - Berkeley DB: open-source embedded library (not distributed!), key ordering (based on Btree+)
  - Oracle NoSQL Database
  - upscaledb
  - LevelDB: by Google, a KV storage library with key ordering
  - RocksDB: by Facebook, evolution of LevelDB
  - Memcached, Redis, Hazelcast: distributed in-memory KV data stores
  - Ehcache: Java-based cache
  - Aerospike: tunable consistency (AP or CP)

Document: data model

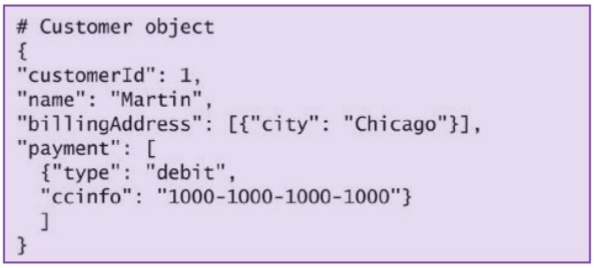
- Strongly aggregate-oriented
  - Lots of aggregates
  - Each aggregate has a key
- Document: collection of named fields and data
  - Encapsulates and encodes data in some standard formats or encodings: JSON, BSON, XML, YAML, …
- Similar to key-value store (unique key), but API or query/update language to query or update based on document’s internal structure
  - Document content is no longer opaque
- Similar to column-family store, but values can have complex documents, instead of fixed format

Document: data model

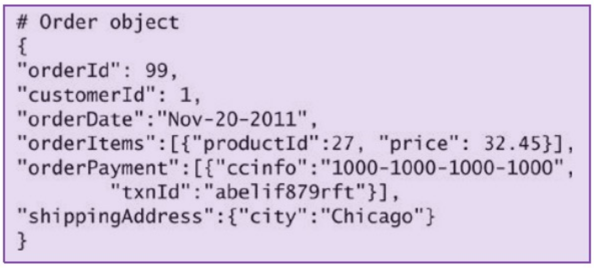
- Data model
  - A set of <key, document> pairs
  - Document: an aggregate instance
- Aggregate structure is visible
  - Limits on what we can place in it
- Access to aggregate
  - Queries based on the fields in the aggregate
- Flexible schema
  - Documents do not need to have same structure
  - Better flexibility: apps can store different data in documents as business requirements change
    - No need of schema migration efforts

Document: data model example

- JSON format



```
# Customer object
{
"customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"payment": [
  {"type": "debit",
  "ccinfo": "1000-1000-1000-1000"}
  ]
}
```



```
# Order object
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",
        "txnId":"abelif879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```

Document: data store API

- Usual CRUD operations (not standardized)
  - Create (or insert)
  - Retrieve (or get, query, search, find)
    - Not only simple key-to-document lookup
    - Query language allows the user to retrieve documents based on the values of one or more fields
  - Update (or edit)
    - Not only the entire document but also individual fields of the document
  - Delete (or remove)
- Read and write operations over multiple fields in a single document are usually atomic
- Some document data stores support indexing to facilitate fast lookup of documents

KV vs. document data stores

- KV data store
  - A key plus a big blob of mostly meaningless bits
  - Can store whatever you like in the aggregate
  - Can only access an aggregate by lookup based on its key
- Document data store
  - A key plus a structured aggregate
  - More flexibility in accessing and updating data
    - Can query based on aggregate fields
    - Can retrieve/update part of the aggregate rather than the whole aggregate
  - Can create indexes based on aggregate content
    - In general, indexes speed up read accesses but slow down write accesses, thus should be designed carefully

KV vs. document data stores

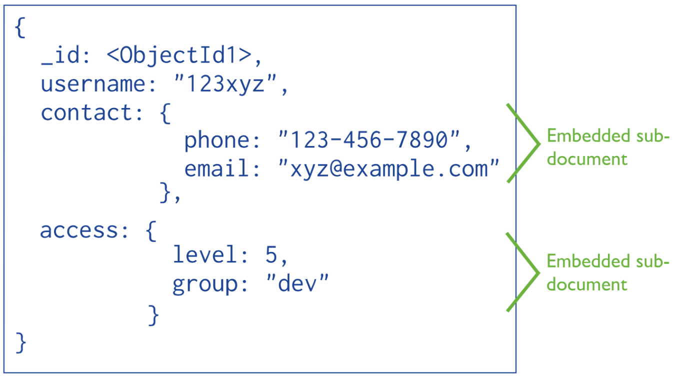
- The line between KV and document gets a bit blurry
  - People often use document store to do a simple KV style lookup
- Data stores classified as KV may allow you to structure data beyond just an opaque, e.g.,
  - Redis allows you to break down the aggregate into lists or sets
  - Riak KV allows you to put aggregates into buckets
  - Others support querying by search tools

Some data model design choices

- Be careful: no universal rule
  - It depends on how your app tends to manipulate data!
- How to model 1:N relationship
  - Simple rule of thumb: how large is N?
    - One-to-few: embedding
    - One-to-many: referencing
    - One-to-squillions: parent-referencing
  - See some example www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1

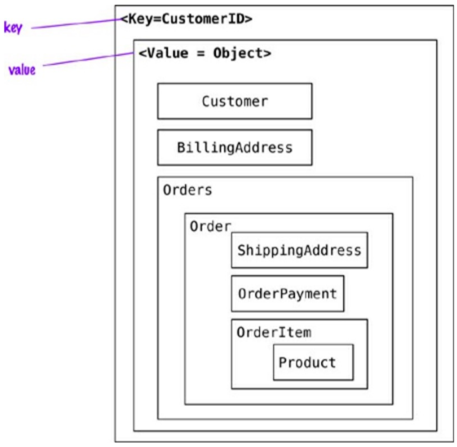
Some data model design choices

- Denormalization
  - Denormalized data models embed related data in a single document



```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
             level: 5,
             group: "dev"
           }
}
```

Embedded sub-document
Embedded sub-document



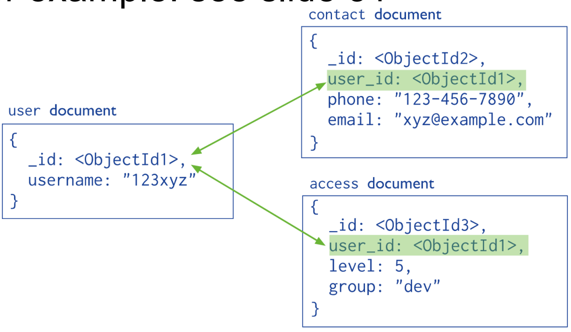
  - See docs.mongodb.com/manual/core/data-modeling-introduction

Some data model design choices

- Denormalization
  - Pros:
    - ✓ Store related pieces of information in same document: fewer queries and updates
    - ✓ Update data within same document in a single atomic write operation
  - Cons:
    - ✗ Document size limit (e.g., 16MB in MongoDB) docs.mongodb.com/manual/core/document
    - ✗ Cannot perform atomic update on multiple documents
    - ✗ Only makes sense when high read to write ratio

Some data model design choices

- Normalization
  - Normalized data models describe relationships using references between documents
    - Another example: see slide 31



contact document

```
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone: "123-456-7890",
  email: "xyz@example.com"
}
```

user document

```
{
  _id: <ObjectId1>,
  username: "123xyz"
}
```

access document

```
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```

  - In general, use normalized data models
    - When embedding would result in duplication of data but would not provide sufficient read performance gains
    - To represent complex many-to-many relationships
    - To model large hierarchical datasets

Document: suitable use cases

- Good for storing and managing big data collections of semi-structured data with a varying number of fields
  - Textual documents, email messages, …
  - Conceptual documents like denormalized representations of DB entities (e.g., product, customer)
  - Sparse data in general, i.e., irregular (semi-structured) data that would require an extensive use of nulls in RDBMS
    - Nulls being placeholders for missing or nonexistent values
- Examples of use cases
  - Log data
  - Product data management (e.g., catalogue)
  - Content personalization
  - User comments (e.g., blog posts)

Document: when not to use

- Complex transactions spanning multiple documents
  - MongoDB 4.x supports distributed transactions but they incur greater performance cost over single-document writes
  - Consider embedding rather than referencing
- Queries against varying aggregate structure
  - Since data is saved as an aggregate, if aggregate structure constantly changes, the aggregate is saved at the lowest level of granularity. In this scenario, document data stores may not perform well

Document: products

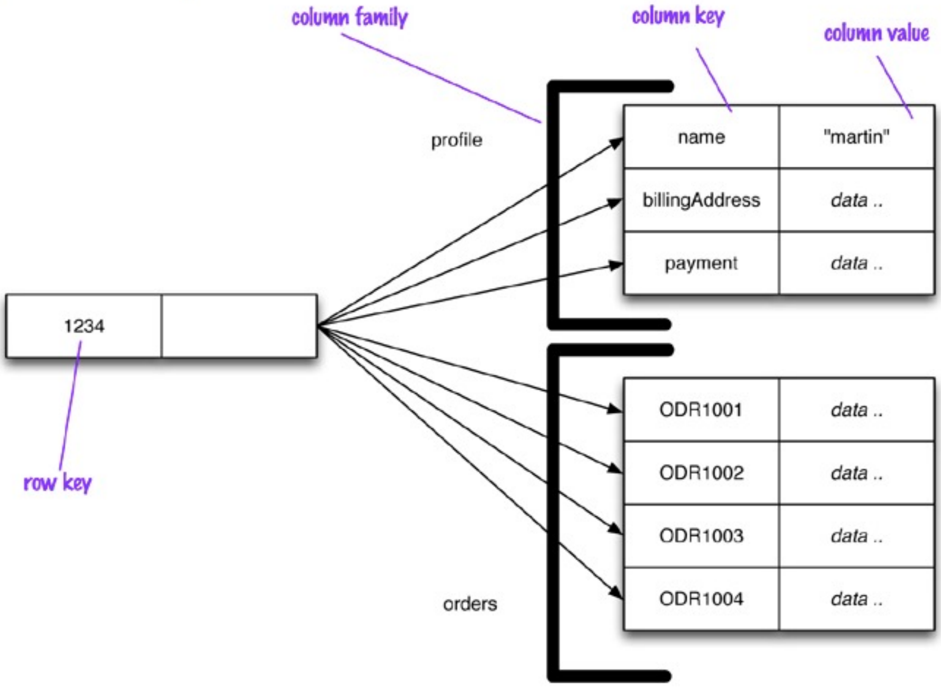
- MongoDB: the most popular
- Aerospike: both KV and document models
- ArangoDB
- Couchbase
- Apache CouchDB
- RavenDB: open source, student licence available
  - ACID, multi-master replication using Raft
- Cloud services
  - Amazon DocumentDB (compatible with MongoDB)
  - Microsoft Azure CosmosDB
- Some of them (e.g., Aerospike, ArangoDB) are multi-model data stores

Column-family: data model

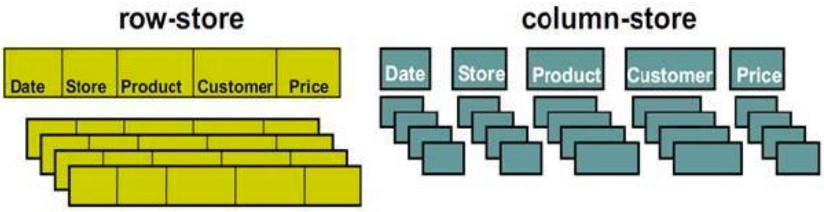
- Strongly aggregate-oriented
  - Lots of aggregates
  - Each aggregate has a key
- Data model: two-level map structure
  - A set of <row-key, aggregate> pairs
  - Each aggregate is a group of pairs <column-key, value>
  - Column: a set of data values of a particular type
- Similar to key-value store, but value can have multiple attributes (*columns*)
- Similar to document store because aggregate structure is visible
- Columns can be organized in families
  - Data usually accessed together

Column-family: data model example

- Representing customer information as column-family



Row-store vs. column-store organization



- Row-store systems: store and process data by rows
  - However, RDBMSs support indexes to improve performance of set-wide operations on whole tables
- Column-store systems: store and process data by columns
  - Can access data faster rather than scanning and discarding unwanted data in row, e.g., for aggregate queries (avg, max, …)
  - Examples: C-Store (pre-NoSQL), Vertica, MariaDB ColumnStore
  - Do not confuse them with column-family data stores

Column-family: features

- Column-family data stores: no column stores in the original sense of the term, because they have a two-level structure with column families
- Table's rows and columns can be split over multiple servers by means of sharding to achieve scalability
- In addition, column families are located on the same partition to facilitate query performance
- Column-family stores are suitable for read-mostly, read-intensive, large data repositories

Column-family: features

- Each column:
  - Has to be part of a single column family
  - Acts as unit for access
- Can get a particular column
  - See slide 43: `get('1234', 'name')`
- Can add any column to any row, and rows can have different columns
- Two ways to think about how data is structured:
  - Row-oriented
    - Each row is an aggregate (e.g., customer with id 1234)
    - Column families represent useful chunks of data within that aggregate (e.g., profile, order history)
  - Column-oriented
    - Each column family defines a record type (e.g., customer profiles)
    - A row is the join of records in all column families

Column-family: suitable use cases

- Queries that involve only a few columns
- Aggregation queries against vast amounts of data
  - E.g., average, maximum
- Apps with truly large volumes of data, such as PBs
- Apps that are geographically distributed over multiple data centers
  - See Cassandra geo-distribution

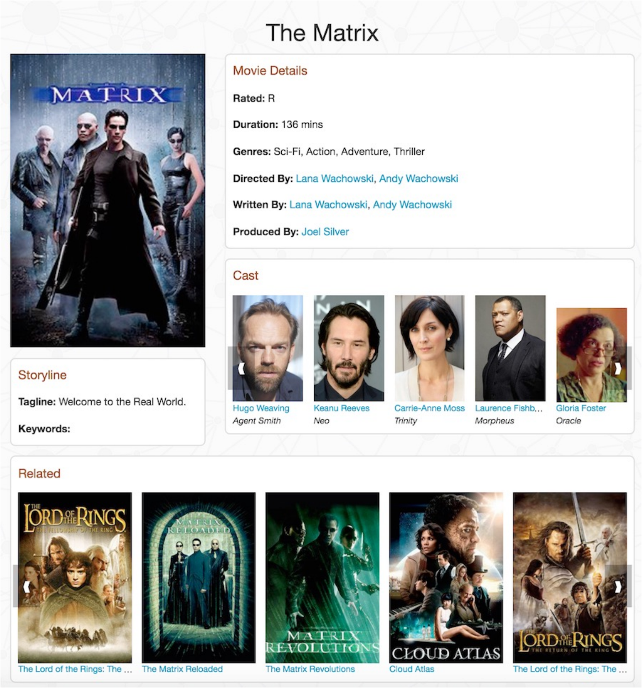
Column-family: products

- Google's Bigtable is the most notable, uses GFS for distributed data storage
- Apache HBase: open-source implementation of Bigtable on top of HDFS
  - Apache Phoenix: SQL query engine on top of HBase
- Other popular column-family data stores
  - Apache Accumulo: based on Bigtable design, uses HDFS to store data and Zookeeper for consensus
    - Different APIs and different nomenclature from HBase, but same in operational and architectural standpoint
    - Better security
  - Apache Cassandra
- Cloud services
  - Google Cloud Bigtable
  - HBase through Amazon EMR or Azure HDInsight

Graph: data model

- Uses graph structure with nodes, edges, and properties to represent stored data
  - Nodes are the entities
    - E.g., users, posts
  - Edges are the relationships between the entities
    - E.g., a user posts a comment
  - Edges can be directed or undirected
  - Nodes and edges also have a set of properties (attributes) consisting of key-value pairs
- Replaces relational tables with structured relational graphs of interconnected key-value pairs

Graph: data model example



The Matrix

Movie Details

Rated: R

Duration: 136 mins

Genres: Sci-Fi, Action, Adventure, Thriller

Directed By: Lana Wachowski, Andy Wachowski

Written By: Lana Wachowski, Andy Wachowski

Produced By: Joel Silver

Cast

Hugo Weaving	Keanu Reeves	Carrie-Anne Moss	Laurence Fishb...	Gloria Foster
Agent Smith	*Neo*	*Trinity*	*Morpheus*	*Oracle*

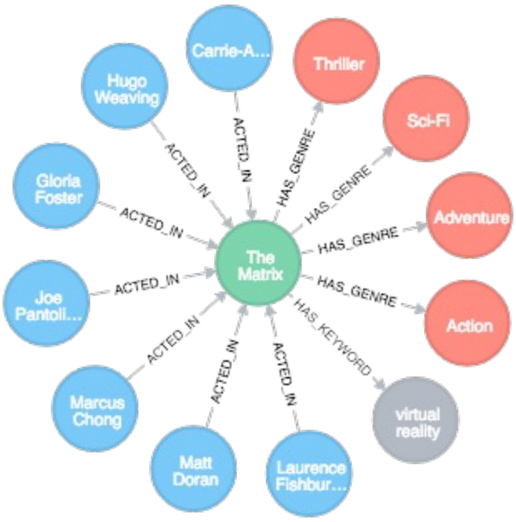
Storyline

Tagline: Welcome to the Real World.

Keywords:

Related

The Lord of the Rings: The ... | The Matrix Reloaded | The Matrix Revolutions | Cloud Atlas | The Lord of the Rings: The ...



github.com/neo4j-graph-examples/movies

Graph: data model

- Powerful data model
  - Differently from other types of NoSQL stores, it concerns itself with relationships
  - Focus on visual representation of information: more human-friendly than other NoSQL stores
  - Other types of NoSQL stores are poor for interconnected data
- Ad-hoc languages to query and manipulate data in graphs, e.g.,
  - Cypher: declarative language (see slides on Neo4j)
  - Gremlin: functional, data-flow language

Graph databases: pros and cons

- Pros:
  - Explicit graph structure
  - Each node knows its adjacent nodes: as number of nodes increases, cost of local hop is the same
  - Can define indexes to make lookups more efficient
- Cons:
  - Sharding data by distributing it on multiple servers to achieve horizontal scalability
    - More difficult than for other types of NoSQL data stores
    - If graph data is stored on different servers, traversing multiple servers is not performance-efficient
  - Require a design effort with respect to SQL

Graph databases vs. aggregate-oriented stores

- Very different data models
- Aggregate-oriented data stores
  - Distributed on multiple servers, also geographically
  - Simple query languages
  - No ACID guarantees
- Graph databases
  - More likely to run on single server (but distributed architectures exist, e.g., OrientDB)
  - Graph-based query languages
  - ACID guarantees: transactions maintain consistency over multiple graph nodes and edges

Graph databases: suitable use cases

- Good for apps where you need to model entities and relationships between them, e.g.,
  - Social networking
  - Dependency analysis
  - Recommender systems
  - Fraud detection
  - Drug discovery
  - Network security
- Good for apps in which focus is on querying for relationships between entities and analyzing relationships
  - Computing relationships and querying related entities is simpler and faster than in RDBMS

Graph databases: products

- Neo4j
- InfiniteGraph: proprietary, distributed
- MemGraph: open source, in-memory
- NebulaGraph: open source, distributed
- OrientDB: open source, distributed, multi-model
- Apache Tinkerpop
  - Not properly a DB but rather a computing framework for graph databases (OLTP) and graph analytic systems (OLAP)
  - Gremlin as graph traversal language
- Cloud services:
  - Amazon Neptune
  - Azure Cosmos DB (multi-model)

Case studies

- Key-value data stores
  - Amazon's Dynamo (and Riak KV)
  - Redis
- Document-oriented data stores
  - MongoDB
- Column-family data stores
  - Google's Bigtable and Hbase
  - Cassandra
- Graph databases
  - Neo4j
- NoSQL Cloud services
  - DynamoDB and Cloud Bigtable

In blue: Hands-on lessons

Case study: Amazon's Dynamo

- Highly available and scalable distributed key-value data store built for Amazon's platform
  - A very diverse set of Amazon applications with different storage requirements
  - Need for storage technologies that are always available on a commodity hardware infrastructure
    - E.g., shopping cart service: "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"
  - Meet stringent Service Level Agreements (SLAs)
    - E.g., "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."

G. DeCandia et al., Dynamo: Amazon's highly available key-value store, *Proc. of ACM SOSP '07*

Dynamo: Features

- Simple key-value API
  - Simple operations to read (get) and write (put) objects uniquely identified by a key
  - Each operation involves only one object at time
- Focus on eventually consistent store
  - Sacrifice consistency for availability
  - BASE rather than ACID
- Efficient usage of resources
- Scale-out to manage increasing data or request rates
- Internal use
  - Security is not an issue since operation environment is assumed to be non-hostile

Dynamo: Design principles

- Sacrifice consistency for availability: AP system
- Use optimistic replication techniques
- Possible conflicting changes must be detected and resolved: *when* to resolve them and *who* resolves them?
  - *When*: execute conflict resolution *during reads* rather than writes, i.e., “always writeable” data store
  - *Who*: data store or application; if data store, use simple policy (e.g., “last write wins”)
- Other key principles:
  - Incremental scalability
    - Scale-out with minimal impact on the system
  - Symmetry and decentralization
    - P2P techniques
  - Heterogeneity

Dynamo: API

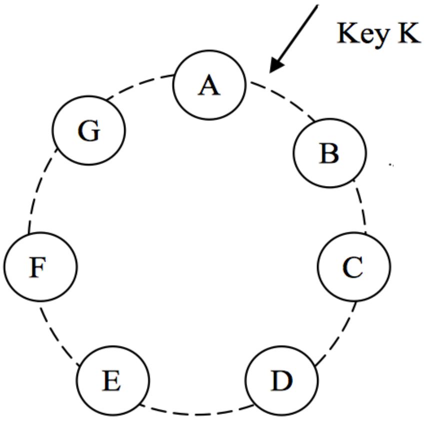
- Each stored object has an associated key
- Simple API including `get` and `put` operations to read and write objects
  - `get(key)`
    - Returns single object or list of objects with conflicting versions and context
    - Conflicts are handled on reads, never reject a write
  - `put(key, context, object)`
    - Determines where the replicas of the object should be placed based on the associated key, and writes the replicas to disk
    - Context encodes system metadata, e.g., version number
  - Both key and object treated as opaque array of bytes
  - Key: 128-bit MD5 hash applied to client supplied key

Dynamo: Used techniques

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

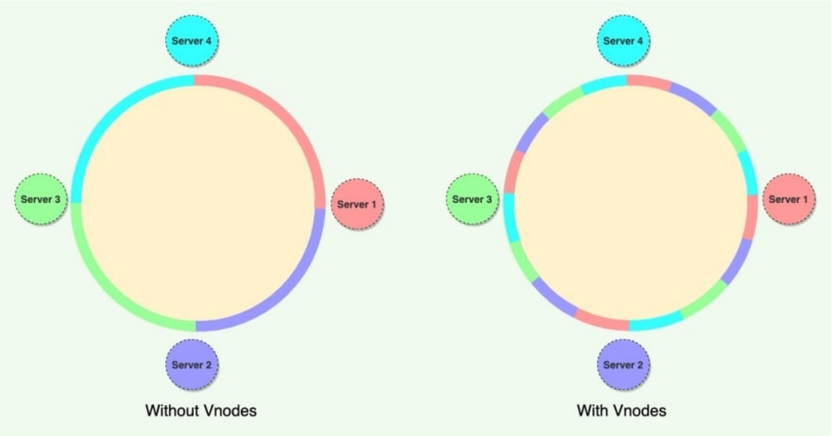
Dynamo: Data partitioning

- Consistent hashing: server nodes and data are both mapped on the same ring using MD5 hashing algorithm (similar to Chord)
  - MD5(key) -> node (position on ring)
  - Differently from Chord: zero-hop DHT, i.e., all nodes know about all nodes; complete routing table
  - The node handling the client request is known as *coordinator*



Dynamo: Data partitioning

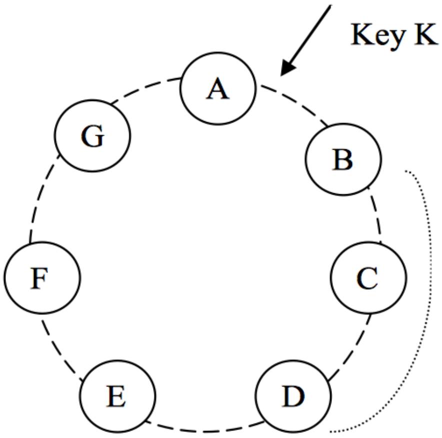
- To address load balance and heterogeneous servers, Dynamo introduces virtual nodes
  - Each physical node can be responsible for multiple virtual nodes
  - Powerful physical nodes can host more virtual nodes (# virtual nodes ∝ capacity)



Source: [How to Use Consistent Hashing in a System Design Interview?]

Dynamo: Replication

- Each object is replicated on *N* nodes
  - *N* is a configurable *replication factor*, set per application
  - The coordinator plus *N*-1 clockwise successor nodes
- *Preference list*: list of nodes that are responsible for storing a particular key
  - More than *N* nodes to account for node failures
  - See figure: object identified by key K is replicated on nodes B, C and D
    - Node D will store keys in ranges (A, B], (B, C], and (C, D]



Nodes B, C and D store keys in range (A,B) including K.

Dynamo: Used techniques

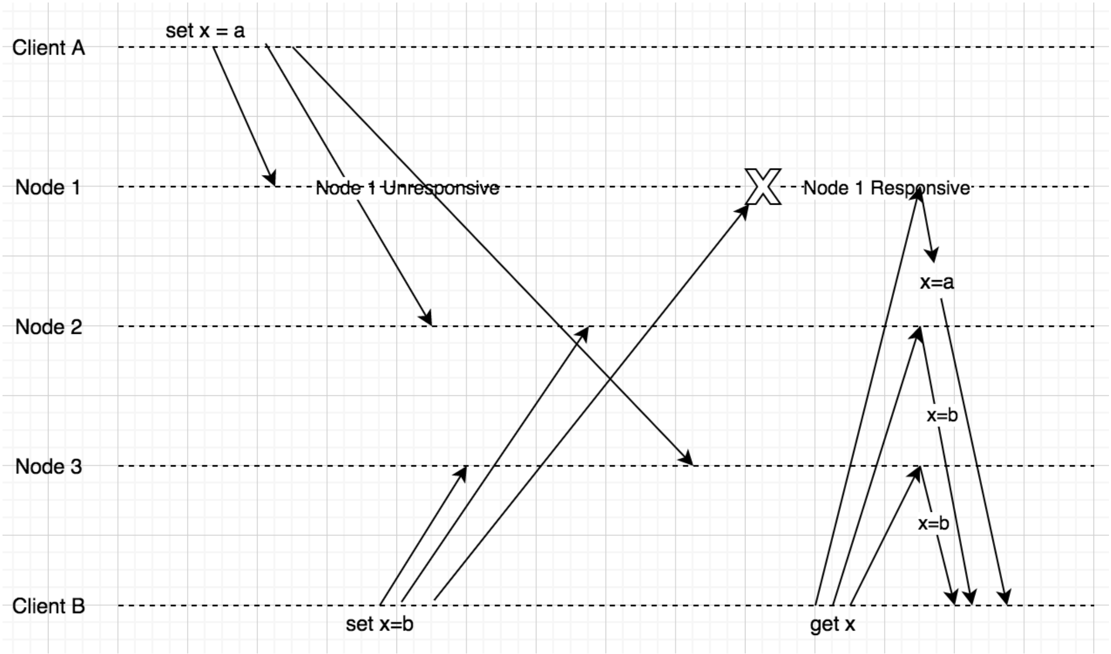
Problem	Technique	Advantage
Partitioning	Consistent hashing	Incremental scalability
High availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability 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

Dynamo: Data versioning

- `put()` may return to client before update is applied to all replicas
- `get()` may return an object version that does not have the latest update
- Version branching can also happen due to failures (node outages, network partitions)
- Problem: multiple conflicting versions of same object, that Dynamo needs to reconcile
- Solution: use vector clocks to capture causality among conflicting versions
  - If causality: older versions can be forgotten (last write wins)
  - If concurrent: conflict exists, requiring reconciliation

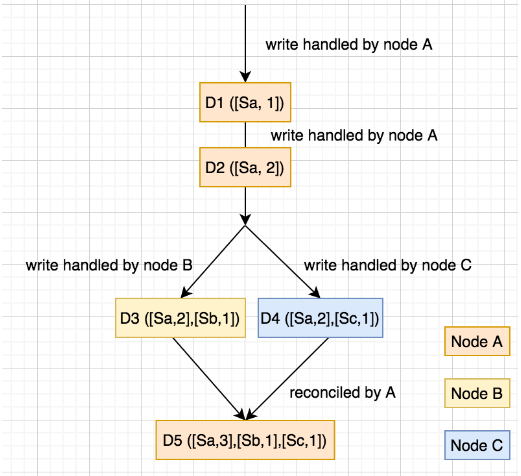
Dynamo: Data versioning

- Example: conflicting versions of object x (a and b)



Dynamo: Data versioning

- Example: conflict resolution
  - Client puts a new object and A is the coordinator node for it
  - The same client updates the object twice
  - Before 3rd write, A crashes and write is handled by B and C, but network partition between B and C occurs
  - When A resumes from crash, the two versions are reconciled



Dynamo: Used techniques

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

Dynamo: put and get ops

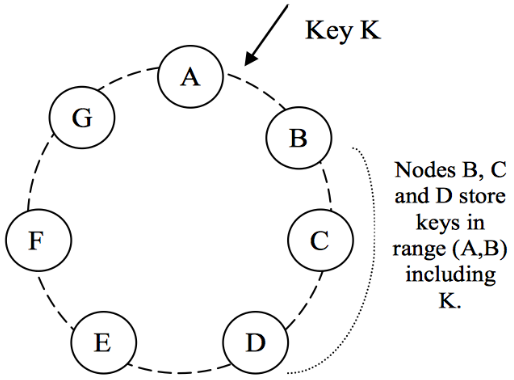
- *R* (*W*): minimum number of nodes that must participate in a successful read (write) op
- `put` op
  - The coordinator generates new vector clock and writes new version locally
  - Sends it to *N*-1 nodes
  - Waits for response from *W* nodes
- `get` op
  - The coordinator sends read requests to *N*-1 nodes
  - Waits for response from *R* nodes
  - If multiple conflicting versions, returns to client all those that are causally unrelated (i.e., concurrent)
  - Conflicting versions are reconciled and winning version is written back

Dynamo: Sloppy quorum

- Setting *R* + *W* > *N* yields a quorum-like protocol
  - `get` (or `put`) latency depends on the slowest replica
  - *R* and *W* are usually configured to be less than *N* to reduce latency
  - Typical configuration in Dynamo: (*N*, *R*, *W*) = (3, 2, 2)
    - To balance latency, durability, and availability
- *Sloppy* quorum
  - Due to network partitions, quorum might not exist
  - Sloppy quorum: create transient replicas (called *hinted* replicas)
    - *N* healthy nodes from the preference list (may not always be the first *N* nodes encountered while walking the ring)

Dynamo: Hinted handoff

- Hinted handoff for transient failures
- Consider *N* = 3; if A is temporarily down or unreachable, `put` will use D
- D knows that the replica belongs to A (since failure is transient)
- Later, D detects A is alive
  - Sends replica to A
  - Removes replica
- Again, “always writeable” principle



Dynamo: Used techniques

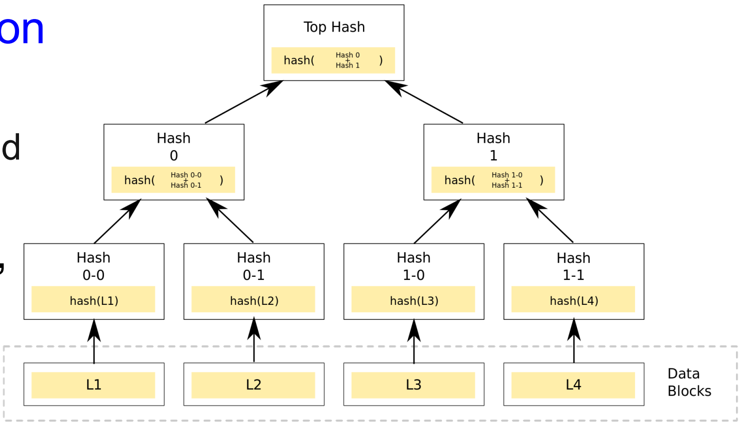
Problem	Technique	Advantage
Partitioning	Consistent hashing	Incremental scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability 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

Dynamo: Membership management, failure detection and management

- Dynamo administrator explicitly adds and removes nodes
- Gossiping is used to propagate membership changes
  - Eventually consistent view
  - O(1) hop overlay
- Passive failure detection
  - Use pings only for detection from failed to alive
  - In the absence of client requests, node A doesn't need to know if node B is alive
- To recover after permanent failures or partitions and keep replicas synchronized, Dynamo uses an anti-entropy mechanism based on Merkle trees

Merkel tree

- Hash-based data structure that generalizes a hash list
- Tree structure in which every leaf node is a hash of a block of actual data and every non-leaf node is a hash of its child nodes
- Tree root summarizes all the data in one hash value
- Merkle trees are used in distributed systems for efficient data verification
  - Efficient because can transmit hashes instead of data blocks
- Used by Dynamo, Git, Cassandra, Bitcoin



en.wikipedia.org/wiki/Merkle_tree

Dynamo: Permanent failure management

- When a node fails and recovers, it needs to quickly determine whether it needs to resynchronize or not
  - Transferring entire (key, value) pairs for comparison is not viable
- Use Merkle trees to rapidly detect inconsistency and limit amount of transferred data
  - Each node maintains a Merkle tree for each key range it hosts
  - Two nodes exchange the root of Merkle tree of the key ranges that they host in common to check if the key ranges they store are updated
  - Tree branches can be checked without having to traverse the entire tree: a branch is traversed only when the hash values at the top of the branch differ
  - Amount of data transferred for synchronization is thus minimized

Riak KV

riakKV

- Distributed NoSQL key-value data store inspired by Dynamo
  - Open-source docs.riak.com/riak/kv/latest
- Like Dynamo
  - Consistent hashing to partition and replicate data in a ring
  - Gossiping to propagate membership changes
  - Vector clocks to resolve conflicts
  - Nodes can be added and removed from Riak cluster as needed
  - Update conflicts can be solved in two ways:
    - Last write wins
    - Conflicting values are returned to client for resolution
- Full-text search within aggregates using integration with Solr

Case study: Google's Bigtable

- Column-family data store designed by Google
- Built on Google products: GFS, Chubby, SSTable
  - GFS for data storage
- Cloud service: Google Cloud Bigtable
- Motivations for Bigtable design
  - Lots of semi-structured data at Google (e.g., URLs)
  - Big data: 10+G of URLs, 1+G users, 100+TB of satellite image data, …
- Used internally by Google services
  - Including MapReduce, Google Maps, Google Earth, YouTube and Gmail
- Does not support SQL, joins and transactions spanning multiple rows

Chang et al., Bigtable: A Distributed Storage System for Structured Data, *OSDI '06*

Bigtable: Main features

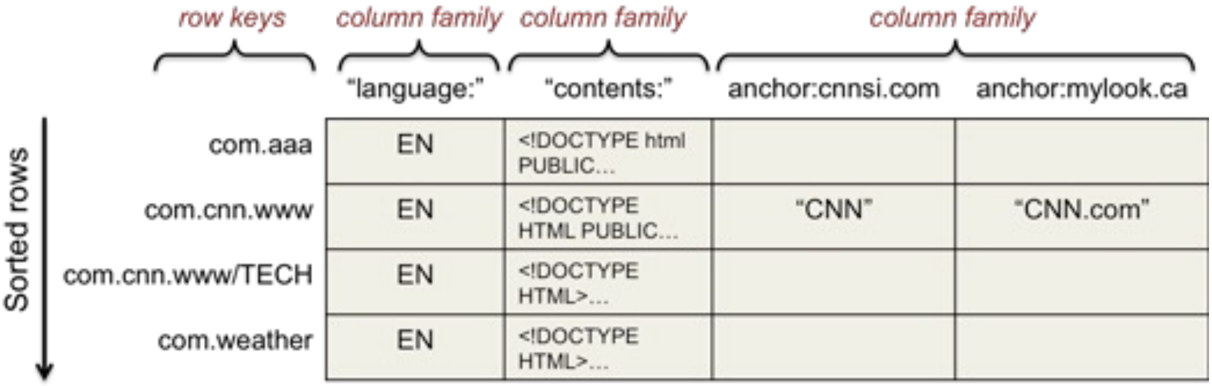
- Distributed storage structured as a large table
  - Distributed, multi-dimensional, sparse, sorted and time-based map
- Fault-tolerant
- Scalable and self-managing
- CP system: strong consistency and tolerance to network partition

Bigtable: Data model

- Table
  - Distributed, multi-dimensional, sparse, sorted and time-based **map**
  - Indexed by rows
- Rows
  - *Sorted* in lexicographical order by **row key**
  - Every read or write in a row is atomic: no concurrent ops on same row
- Columns
  - Unit of data access
  - *Sparse* table: different rows may have different columns
  - **Column family**: group of columns
    - Data within the column family are usually of same type
  - Column family allows for specific optimization for better access control, storage and data indexing
  - Column naming: *column-family:column*

Bigtable: Data model

- *Multi-dimensional*: rows, column families and columns provide a three-level naming hierarchy in identifying data

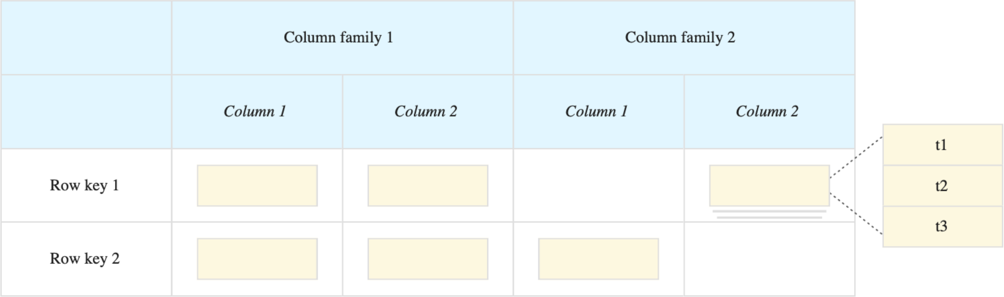


row keys	*column family*	*column family*	*column family*	
	"language:"	"contents:"	anchor:cnnsi.com	anchor:mylook.ca
com.aaa	EN	<!DOCTYPE html PUBLIC...		
com.cnn.www	EN	<!DOCTYPE HTML PUBLIC...	"CNN"	"CNN.com"
com.cnn.www/TECH	EN	<!DOCTYPE HTML>...		
com.weather	EN	<!DOCTYPE HTML>...		

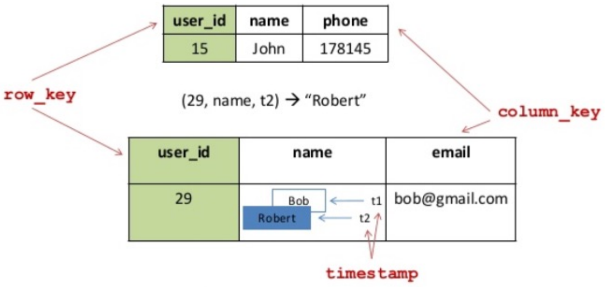
Sorted rows

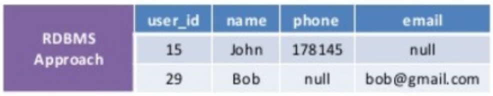
Bigtable: Data model

- *Time-based*
  - Multiple versions in each cell, each one having a timestamp



- Bigtable data model vs. relational data model

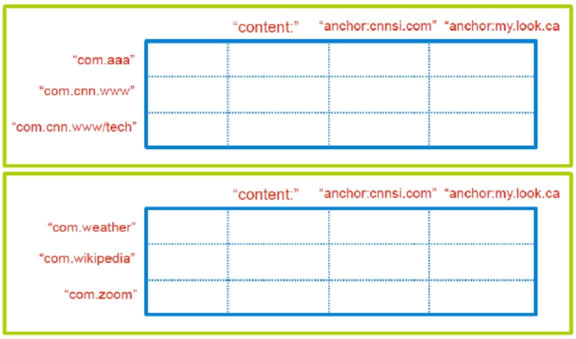




RDBMS Approach	user_id	name	phone	email
	15	John	178145	null
	29	Bob	null	bob@gmail.com

Bigtable: Tablet

- **Tablet**: group of consecutive rows of a table stored together
  - Unit for data storing and distribution
  - Table sorted by row keys: select row keys properly to improve data locality
- Each tablet is served by exactly one tablet server
- Auto-sharding: tablets split by Bigtable when they become too large

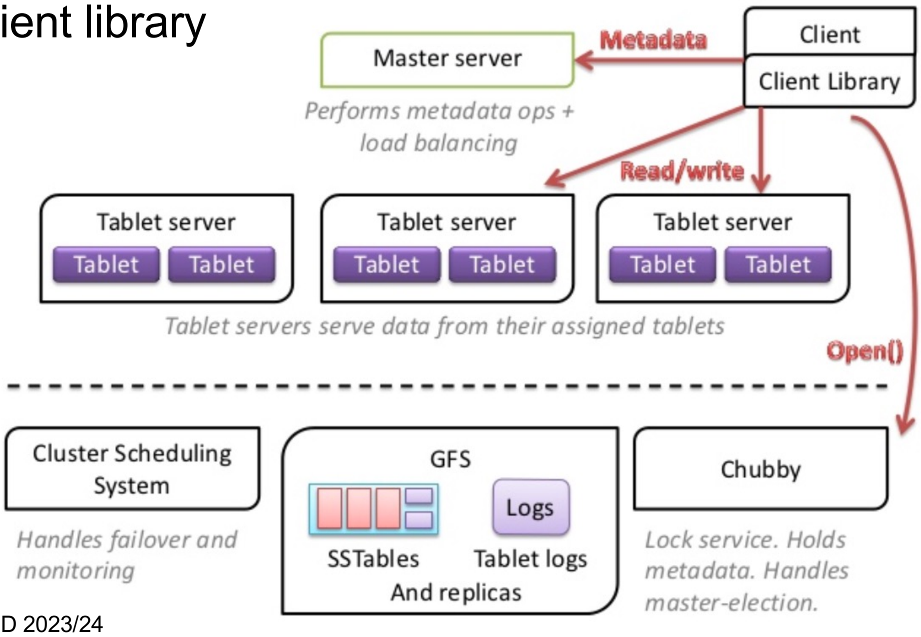


Bigtable: API

- Metadata ops
  - Create/delete tables and column families, change metadata
- Write ops: single-row, atomic
  - Write/delete cells in a row, delete all cells in a row
- Read ops: read arbitrary cells in a table
  - Each row read is atomic
  - One row, all or specific columns, certain timestamps, ...

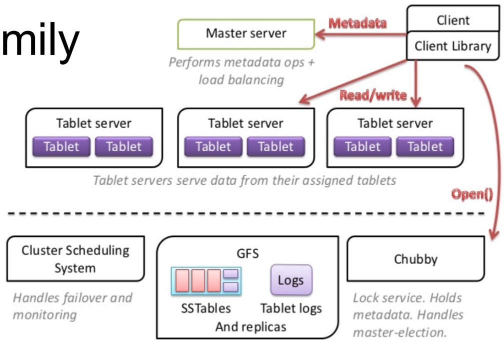
Bigtable: Architecture

- Main components:
  - Master server
  - Tablet servers
  - Client library


Master server
Metadata
Client
Client Library
Performs metadata ops + load balancing
Read/write
Tablet server
Tablet
Tablet
Tablet server
Tablet
Tablet
Tablet server
Tablet
Tablet
Tablet servers serve data from their assigned tablets
Open()
Cluster Scheduling System
Handles failover and monitoring
GFS
Logs
SSTables
Tablet logs
And replicas
Chubby
Lock service. Holds metadata. Handles master-election.

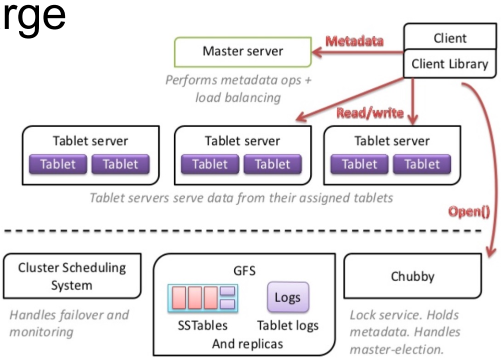
Bigtable: Master server

- Single master server
- Detects addition/deletion of tablet servers
- Assigns tablets to tablet servers
- Balances load among tablet servers
- Garbage collection of unneeded files in GFS
- Handles schema changes
  – e.g., table and column family creation


Master server
Performs metadata ops + load balancing
Metadata
Client
Client Library
Read/write
Tablet server
Tablet
Tablet
Tablet server
Tablet
Tablet
Tablet server
Tablet
Tablet
Tablet servers serve data from their assigned tablets
Open()
Cluster Scheduling System
Handles failover and monitoring
GFS
Logs
SSTables
Tablet logs
And replicas
Chubby
Lock service. Holds metadata. Handles master-election.

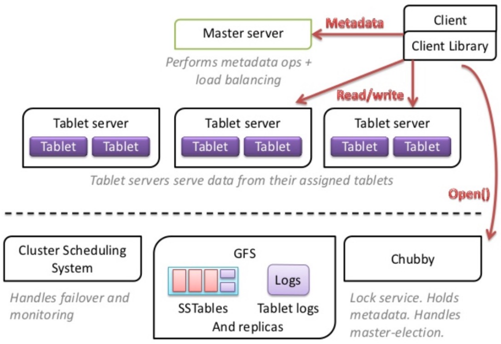
Bigtable: Tablet server

- Many tablet servers
- Can be added or removed dynamically
- Each tablet server:
  – Manages a set of tablets (typically 10-1000 tablets/server)
  – Handles read/write requests to tablets
  – Splits tablets when too large


Master server
Performs metadata ops + load balancing
Metadata
Client
Client Library
Read/write
Tablet server
Tablet
Tablet
Tablet server
Tablet
Tablet
Tablet server
Tablet
Tablet
Tablet servers serve data from their assigned tablets
Open()
Cluster Scheduling System
Handles failover and monitoring
GFS
Logs
SSTables
Tablet logs
And replicas
Chubby
Lock service. Holds metadata. Handles master-election.

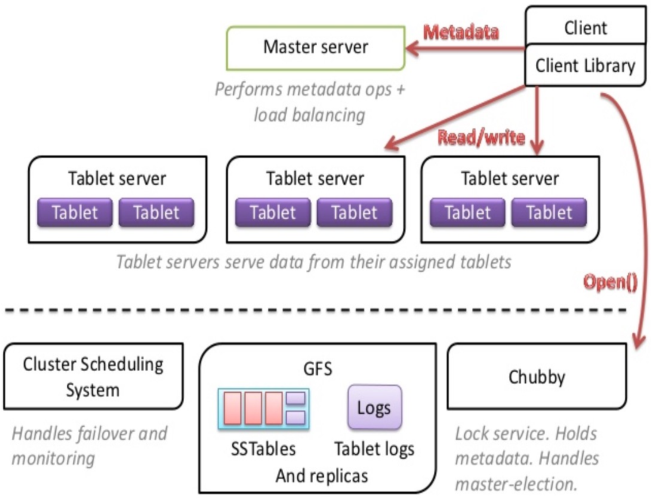
Bigtable: Client library

- Library used by client
- Client data does not go through master
  - Only metadata goes
  - Clients communicate directly with tablet servers for reads/writes
  - Recall GFS's design principles


Master server
Metadata
Client
Client Library
Performs metadata ops + load balancing
Read/write
Tablet server
Tablet
Tablet
Tablet server
Tablet
Tablet
Tablet server
Tablet
Tablet
Tablet servers serve data from their assigned tablets
Open()
Cluster Scheduling System
Handles failover and monitoring
GFS
Logs
SSTables
Tablet logs
And replicas
Chubby
Lock service. Holds metadata. Handles master-election.

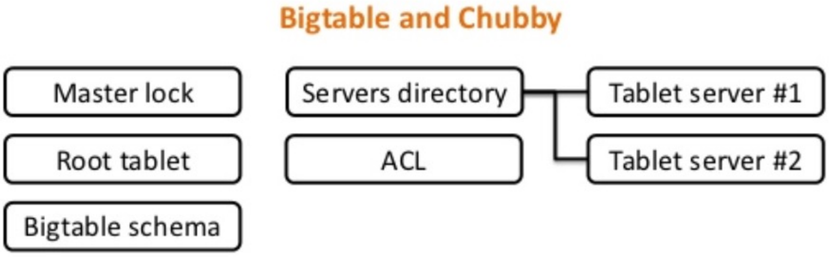
Bigtable: Building blocks

- External building blocks of Bigtable:
  - **Google File System** (GFS): data storage
  - **Chubby**: distributed lock service
  - **Cluster scheduler**: schedules jobs onto cluster servers


Master server
Metadata
Client
Client Library
Performs metadata ops + load balancing
Read/write
Tablet server
Tablet
Tablet
Tablet server
Tablet
Tablet
Tablet server
Tablet
Tablet
Tablet servers serve data from their assigned tablets
Open()
Cluster Scheduling System
Handles failover and monitoring
GFS
Logs
SSTables
Tablet logs
And replicas
Chubby
Lock service. Holds metadata. Handles master-election.

Bigtable: Chubby lock service

- Chubby: distributed and highly available lock service used in many Google's products
  - File system {directory/file} for locking
  - Paxos for consensus to keep replicas consistent
- Bigtable uses Chubby to:
  - Ensure there is only one active master (i.e., master lock)
  - Store bootstrap location of Bigtable data (i.e., root tablet)
  - Store Bigtable schema information
  - Discover tablet servers
  - Store access control lists (ACLs)

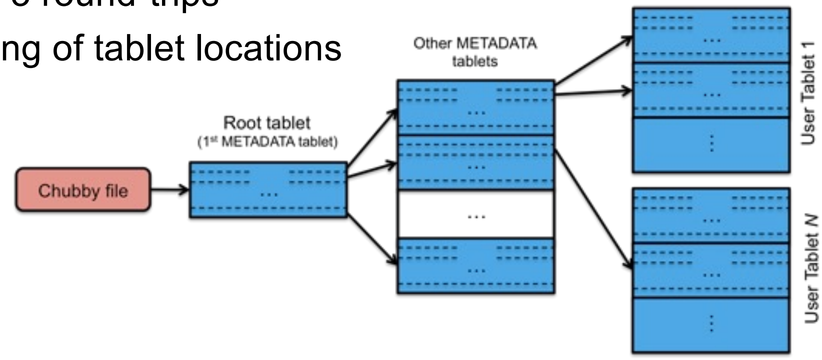


Bigtable: Locating rows

- Three-level indexing hierarchy

1. Chubby stores the location of the root tablet
2. Root tablet stores the location of all METADATA tablets in a special METADATA tablet
3. Each METADATA tablet stores the location of a set of user tablets

- To reduce latency, clients cache tablet locations
  - Empty cache: 3 round-trips
  - Also prefetching of tablet locations (data locality)



Bigtable: Master startup

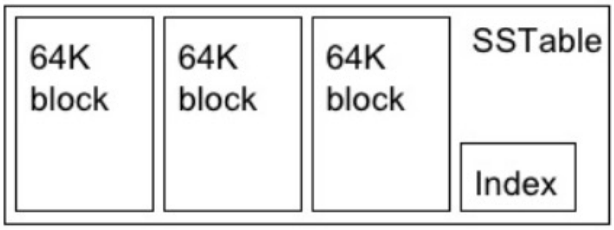
- At startup, master executes the following steps:
  - Grabs a unique master lock in Chubby (leader election)
  - Scans tablet servers directory in Chubby to find live servers
  - Communicates with live tablet servers to find which tablets are assigned to them
  - Scans METADATA tablets to learn the set of tablets that exist and determines the tablets not yet assigned to tablet servers

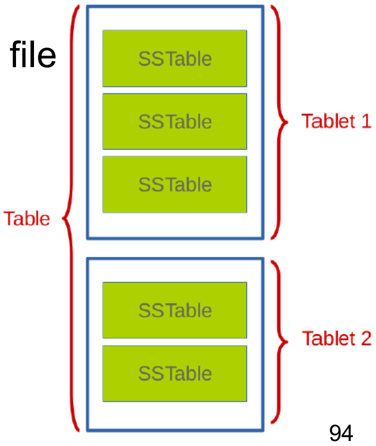
Bigtable: Tablet assignment

- Each tablet assigned to only one tablet server at a time
- Master uses Chubby to keep tracks of live tablet serves and unassigned tablets
- When a tablet server starts, it creates and acquires an exclusive lock in Chubby
- Master detects the lock status of each tablet server by checking Chubby periodically
- Master is responsible for finding when tablet server is no longer serving its tablets and reassigning those tablets as soon as possible to other servers

Bigtable: SSTable

- Data is never stored in tablet servers; each server has pointers to a set of tablets that are stored in GFS
- Sorted Strings Table (**SSTable**): file format used to store Bigtable data durably in GFS
  - Persistent, immutable key-value map, sorted by keys
  - Stored on disk as data blocks plus an index of block ranges
  - Index is used to locate blocks faster and is loaded into memory when SSTable is opened
  - In Bigtable, each SSTable is stored in a GFS file
  - File format used also by other NoSQL data stores (e.g., Cassandra, ScyllaDB)





Bigtable: SSTable

- To speed-up reads (read ops need to read from all SSTables that make up the tablet) Bigtable employs a data structure known as Bloom filter
- Stored in memory, one per SSTable, and used to test if row data exists before accessing SSTable on disk

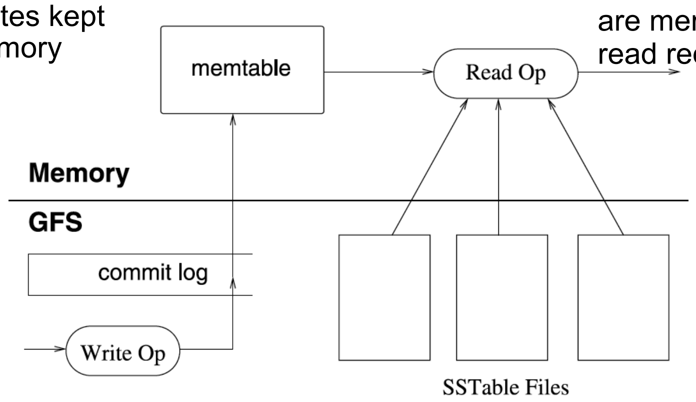
- Bloom filter: space and time-efficient probabilistic data structure used to know whether an element is present in a set
  - Probabilistic: the element either definitely is not in the set or may be in the set (i.e., false positives are possible but false negatives are not)

Bigtable: writing tablets and reading from tablets

- How to support fast writes with SSTables?
  - Write in memory!
- Updates committed to a separate commit log
- Recently committed writes are cached in memory in a memtable, older writes are stored in a series of SStables
- To serve reads, the tablet server merges SSTables and memtable into a single view

Recent updates kept sorted in memory

memtable and SSTables are merged to serve a read request



Write ops are logged

Bigtable: Loading tablets

- To load a tablet, a tablet server:
  - Finds tablet location through its METADATA tablet
    - Metadata for a tablet contains list of SSTables
  - Read SSTables index blocks into memory
  - Read the commit log since the redo point and reconstructs the memtable

Bigtable: Consistency and availability

- Strong consistency: Bigtable is a CP system
  - Only one tablet server is responsible for a given data
  - Data replication is handled by GFS
- Tradeoff with availability
  - If a tablet server fails, its portion of data is temporarily unavailable until a new tablet server is assigned

Comparing Dynamo and Bigtable

	Dynamo	Bigtable
Data model	Key-value	Column-family
API	Single value	Single value and range
Data partition	Random	Ordered
Optimized for	Writes	Reads
Consistency	Eventual	Strong (all reads and writes to a single row are atomic)
Multiple versions	Version number	Timestamp
Replication	Quorum	GFS
Persistency	Local and replicated	Replicated and distributed file system
Architecture	Decentralized	Hierarchical (master/worker)

Google's Cloud Bigtable

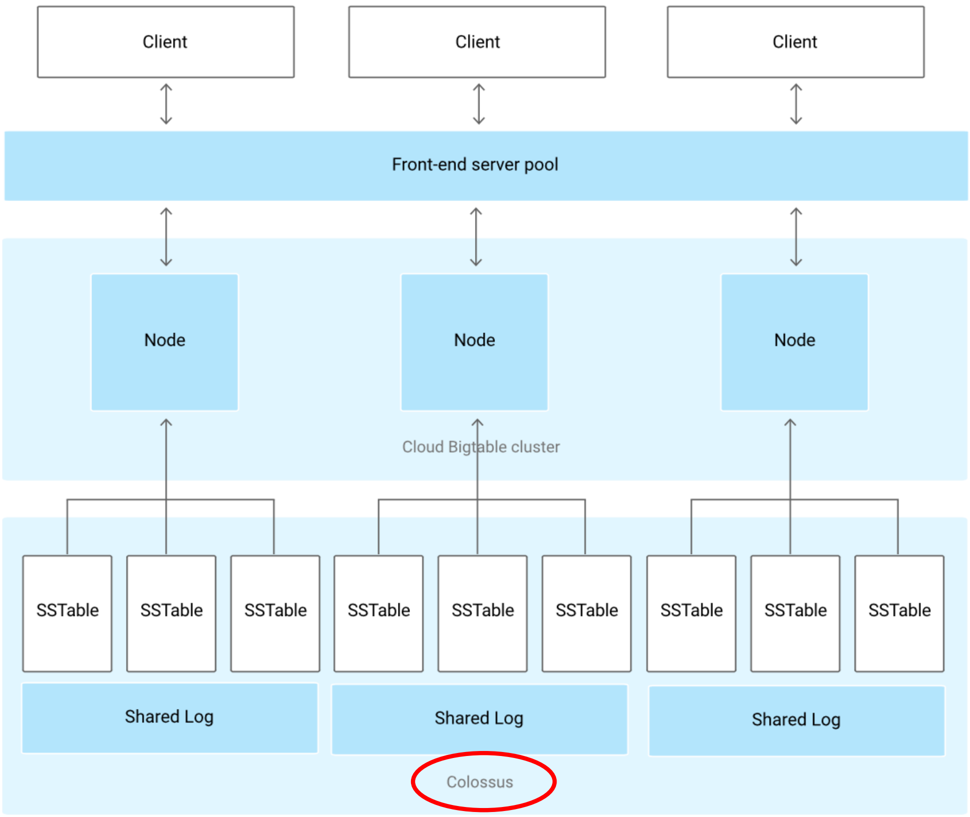
- Bigtable as Cloud service: same concepts!
- Sparse table that can scale to billions of rows and thousands of columns
  - Table: sorted key-value map
  - Table composed of rows and columns
    - Each row describes a single entity
    - Each column contains individual values for each row
    - Each row/column intersection can contain multiple cells at different timestamps
    - Row indexed by a single row key
    - Columns can be grouped together into a column family
      - Column identified by column family and column qualifier
- Supports *multi-region* replication to increase availability
  - By default, replication is eventually consistent

Cloud Bigtable: use cases

- When to use
  - To store large amounts of single-keyed data with low latency for apps that need high throughput and scalability for non-structured key-value data
    - Single value no larger than 10 MB
    - At least 1 TB of data
  - Examples
    - Marketing data (e.g., purchase histories, customer preferences)
    - Financial data (e.g., transaction histories, stock prices)
    - IoT data (e.g., usage reports from energy meters and home appliances)
    - Time-series data (e.g., CPU and memory usage over time for multiple servers)

cloud.google.com/bigtable/docs/overview

Cloud Bigtable: architecture



Cloud Bigtable: usage

- Through command-line tools
  - Using cbt (native CLI in Go) cloud.google.com/bigtable/docs/cbt-overview
  - Using HBase shell
- Through Cloud Client Libraries for Cloud Bigtable API

Cloud Bigtable: schema design example

- Dataset on Kaggle about New York City buses
  www.kaggle.com/stoney71/new-york-city-transport-statistics
- More than 300 bus routes and 5,800 vehicles following those routes
  - Timestamp, origin, destination, vehicle id, vehicle latitude and longitude, expected and scheduled arrival times
- Keep in mind
  - A table has only one index (row key), no secondary indexes
  - Rows are automatically sorted lexicographically by row key
  - Cloud Bigtable allows for queries using point lookups by row key or row-range scans
    - Try to avoid slow operations, i.e., multiple row lookups or full table scans
  - Keep all information for an entity in a single row

Cloud Bigtable: schema design example

- Queries about NYC buses
  - Get locations of a specific bus over an hour
  - Get locations of an entire bus line over an hour
  - Get locations of all buses in Manhattan in an hour
  - Get most recent locations of all buses in Manhattan in an hour
  - Get locations of an entire bus line over the month
  - Get locations of an entire bus line with a certain destination over an hour

Java code: codelabs.developers.google.com/codelabs/cloud-bigtable-intro-java

Cloud Bigtable: schema design example

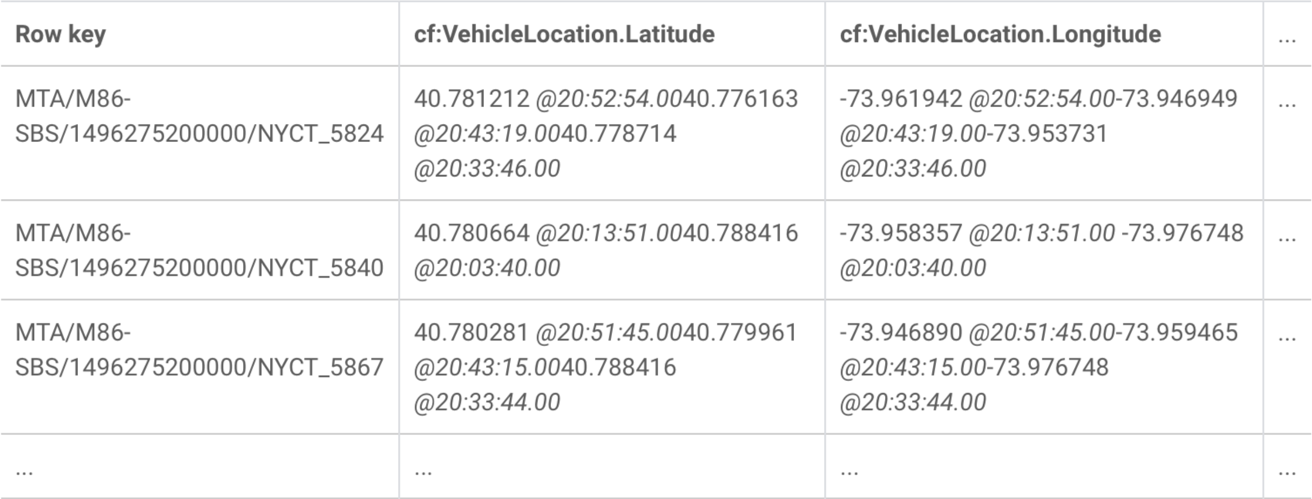
- Row key design is crucial for Bigtable performance
- How to design row key?
  - Consider how you will use stored data
  - Keep row key reasonably short
  - Use human-readable values instead of hashing
  - Include multiple identifiers in row key

- Common mistake: make time the first value in row key
  - Can cause hotspots and result in poor performance: most queries would be managed by a single tablet server

Cloud Bigtable: schema design example

- How to design the row key for the NYC buses?

[Bus company/Bus line/Timestamp rounded down to the hour/Vehicle ID]



Row key	**cf:VehicleLocation.Latitude**	**cf:VehicleLocation.Longitude**	...
MTA/M86-SBS/1496275200000/NYCT_5824	40.781212 *@20:52:54.00*40.776163 *@20:43:19.00*40.778714 *@20:33:46.00*	-73.961942 *@20:52:54.00*-73.946949 *@20:43:19.00*-73.953731 *@20:33:46.00*	...
MTA/M86-SBS/1496275200000/NYCT_5840	40.780664 *@20:13:51.00*40.788416 *@20:03:40.00*	-73.958357 *@20:13:51.00* -73.976748 *@20:03:40.00*	...
MTA/M86-SBS/1496275200000/NYCT_5867	40.780281 *@20:51:45.00*40.779961 *@20:43:15.00*40.788416 *@20:33:44.00*	-73.946890 *@20:51:45.00*-73.959465 *@20:43:15.00*-73.976748 *@20:33:44.00*	...
...	...	...	...

Case study: Cassandra

- Initially developed at Facebook, now Apache project
- A mixture of Dynamo and BigTable cassandra.apache.org

Dynamo

Distributed storage and replication techniques: P2P architecture (replication & partitioning based on consistent hashing), gossip-based membership and failure detection, Merkle tree for replica repair

BigTable

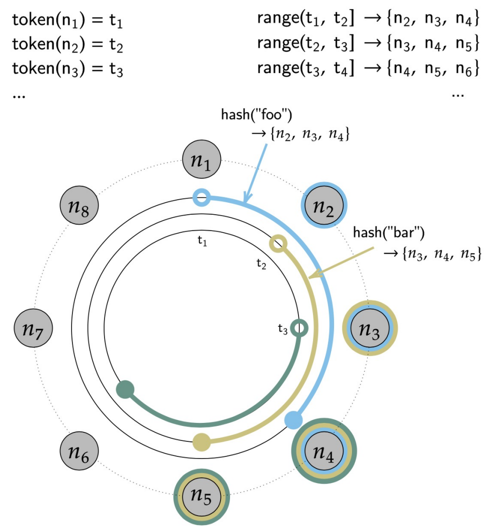
Data and storage engine model: sparse column-oriented data model, storage architecture (SSTables, memtable)

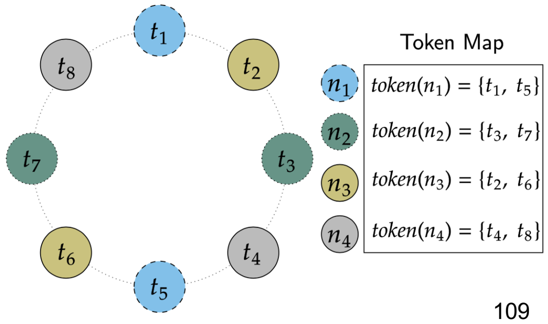
Cassandra

- Large-scale production deployments:
  - Apple: over 75,000, over 10 PB of data
  - Netflix: 2,500 nodes, 420 TB, over 1 trillion requests per day

Cassandra: Features

- Data partitioning
  - By means of consistent hashing using a token ring
- Incremental scalability
  - Virtual nodes: a single physical node takes multiple positions in the ring
- High availability
  - Failure detection based on gossiping
- Durability
  - By using replicas
- Support for geo-distributed data centers


token(n_1) = t_1
token(n_2) = t_2
token(n_3) = t_3
...
range(t_1, t_2] → {n_2, n_3, n_4}
range(t_2, t_3] → {n_3, n_4, n_5}
range(t_3, t_4] → {n_4, n_5, n_6}
...
hash("foo") → {n_2, n_3, n_4}
hash("bar") → {n_3, n_4, n_5}
t_1
t_2
t_3
n_1
n_2
n_3
n_4
n_5
n_6
n_7
n_8


t_1
t_2
t_3
t_4
t_5
t_6
t_7
t_8
Token Map
n_1 $token(n_1) = \{t_1, t_5\}$
n_2 $token(n_2) = \{t_3, t_7\}$
n_3 $token(n_3) = \{t_2, t_6\}$
n_4 $token(n_4) = \{t_4, t_8\}$

Cassandra: Features

- Data versioning
  - Updates resolve according to the conflict resolution rule of last write wins
- Replica versioning
  - As replicas in Cassandra can accept mutations (writes) independently, some replicas could have newer data than others. Similarly to Dynamo, Cassandra employs *best-effort* techniques to drive convergence of replicas including replica read repair during read operations and hinted handoff during write operations
    - Hinted handoof as in Dynamo: when replica nodes are unavailable to accept a mutation (e.g., due to failure), coordinators store temporary hints on their local filesystem for later application to the unavailable replica
  - Also anti-entropy repair by means of Merkle trees

Cassandra: Data model

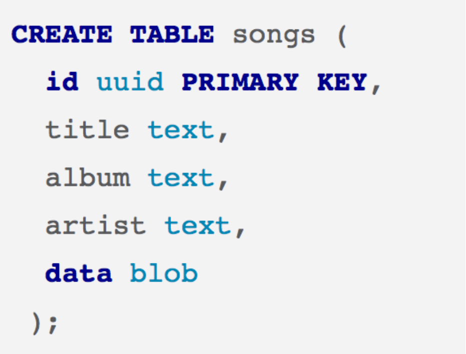
- Structured key-value store where columns are added only to specified rows
  - Distributed multi-dimensional map indexed by row key
  - Different rows can have different number of columns and columns are grouped into column families
  - Emphasis on denormalization instead of normalization and joins
- Write-oriented system (i.e., for write intensive workloads)
  - On the contrary, Bigtable is read-oriented

Cassandra Query Language (CQL)

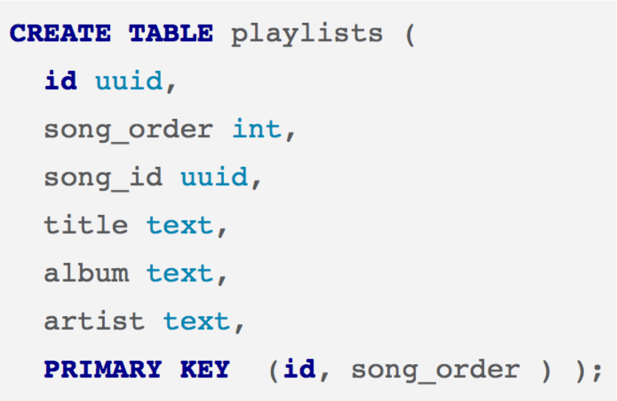
- SQL-like language
  cassandra.apache.org/doc/stable/cassandra/cql/
- CQL allows users to organize data within a cluster of Cassandra nodes usings
  - Keyspace: defines how a dataset is replicated, per data center; a keyspaces contains tables
  - Table: defines the typed schema for a collection of partitions. A table contains partitions, which contain rows, which contain columns; Cassandra tables can flexibly add new columns
  - Partition: defines the mandatory part of the primary key all rows in Cassandra must have to identify the node in a cluster where the row is stored
  - Row: contains a collection of columns identified by a unique primary key made up of the partition key and optionally additional clustering keys
  - Column: a single datum with a type which belongs to a row

CQL: example

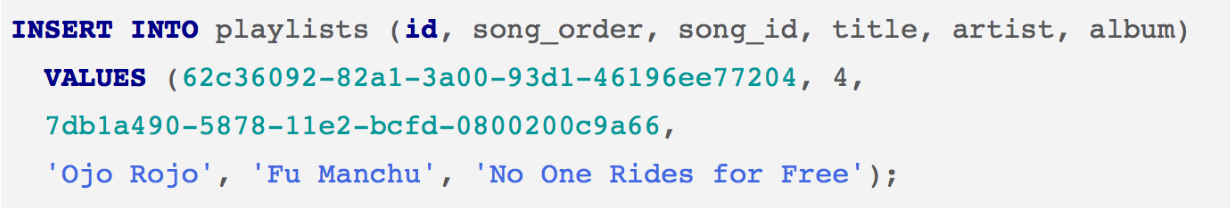
- Example: music service docs.datastax.com/en/cql-oss/3.1/cql/ddl/ddl_music_service_c.html



```
CREATE TABLE songs (
  id uuid PRIMARY KEY,
  title text,
  album text,
  artist text,
  data blob
 );
```



```
CREATE TABLE playlists (
  id uuid,
  song_order int,
  song_id uuid,
  title text,
  album text,
  artist text,
  PRIMARY KEY  (id, song_order ) );
```

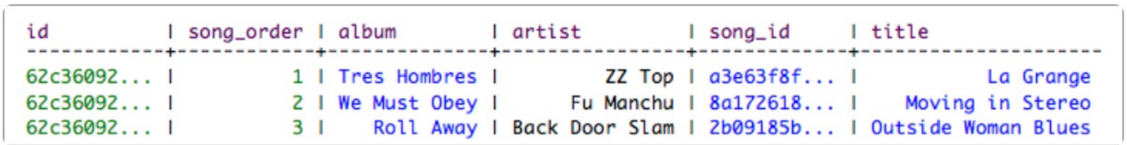


```
INSERT INTO playlists (id, song_order, song_id, title, artist, album)
  VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 4,
  7db1a490-5878-11e2-bcfd-0800200c9a66,
  'Ojo Rojo', 'Fu Manchu', 'No One Rides for Free');
```

CQL: example

- Use SELECT to display table data

```
SELECT * FROM playlists;
```

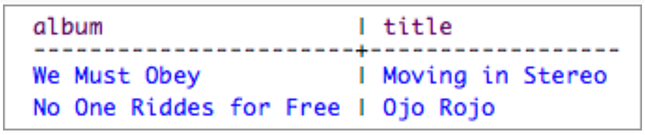


```
id           | song_order | album         | artist         | song_id      | title
-------------+------------+---------------+----------------+--------------+---------------------
62c36092... |          1 | Tres Hombres |         ZZ Top | a3e63f8f... |            La Grange
62c36092... |          2 | We Must Obey |      Fu Manchu | 8a172618... |     Moving in Stereo
62c36092... |          3 |     Roll Away | Back Door Slam | 2b09185b... | Outside Woman Blues
```

- To use artist as a filter, first create an index (because artist is not a partition key or clustering column) on artist and then select

```
CREATE INDEX ON playlists( artist );
```

```
SELECT album, title FROM playlists WHERE artist = 'Fu Manchu';
```



```
album                   | title
------------------------+------------------
We Must Obey            | Moving in Stereo
No One Riddes for Free  | Ojo Rojo
```

Cassandra: Consistency

- Multi-master replication
  - No single master node to coordinate reads and writes
- Consistency levels driven by a quorum-based protocol
  - In Cassandra, you choose from a menu of consistency levels which allow you to pick R and W behavior
  - Writes will be visible to subsequent reads when read consistency level contains enough nodes to guarantee a quorum intersection with write consistency level
    - i.e., R+W > N (being N the replication factor)
  - Some consistency levels
    - ONE: only a single replica must respond
    - QUORUM: a majority of the replicas must respond
    - ALL: all of the replicas must respond
    - LOCAL_QUORUM: a majority of the replicas in the local datacenter must respond

cassandra.apache.org/doc/stable/cassandra/architecture/dynamo.html#multi-master-replication-versioned-data-and-tunable-consistency

Cassandra: Consistency

- Consistency is tunable *per-operation*
- Example in CQL

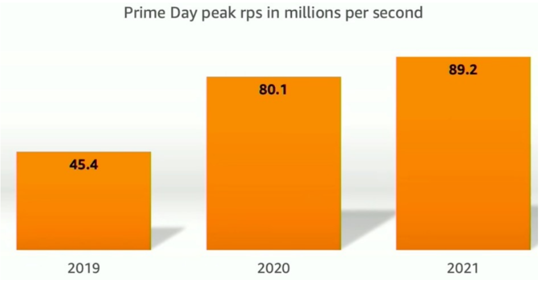
```
SELECT points
  FROM fantasyfootball.playerpoints USING CONSISTENCY QUORUM
 WHERE playername = ‘Tom Brady’;
```

CQL: operations not supported

- Cassandra explicitly chooses not to implement operations that require cross-partition coordination
- Why? They are typically slow and hard to provide high availability
- Cassandra does not support:
  - Cross-partition transactions
  - Distributed joins
  - Foreign keys or referential integrity

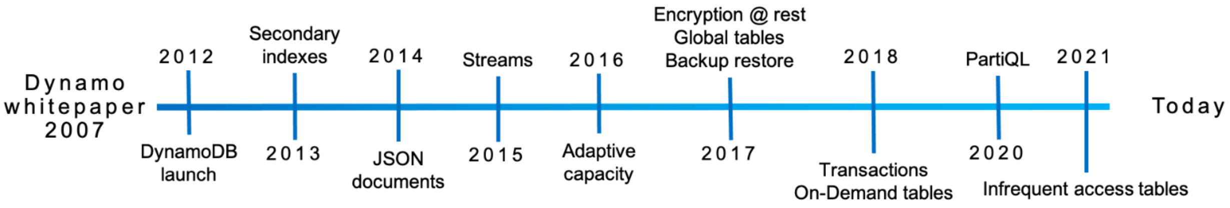
Amazon’s DynamoDB

- Fully managed NoSQL cloud data store service
- Multi-tenant
- Flexible schema: key-value data store
- Highly available
  - Replication across 3 availability zones
- Boundness scale for tables
- Predictable performance at any scale
  - During Amazon Prime Day in 2021, Amazon systems made trillions (10^{18}) of API calls to DynamoDB, while experiencing high availability with single-digit msec performance



Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service, USENIX ATC 2022

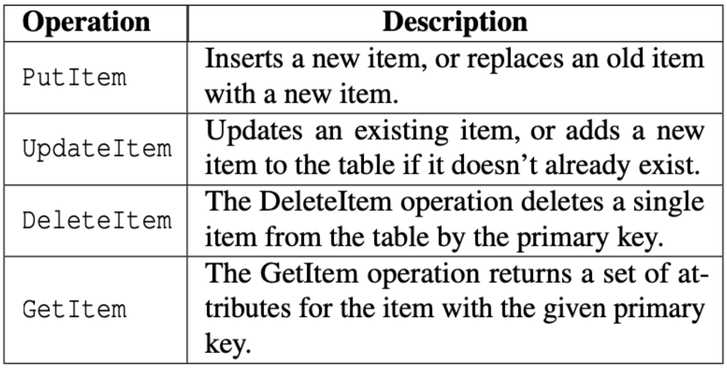
DynamoDB: evolution


Dynamo whitepaper 2007
2012 DynamoDB launch
2013 Secondary indexes
2014 JSON documents
2015 Streams
2016 Adaptive capacity
2017 Encryption @ rest, Global tables, Backup restore
2018 Transactions, On-Demand tables
2020 PartiQL
2021 Infrequent access tables
Today

Dynamo	DynamoDB
Single tenant	Multi-tenant
Tunable consistency	Strong and eventual consistency
Availability over consistency	Consistency over availability
Routing and storage coupled	Routing and storage decoupled
Custom conflict resolution	Last write wins

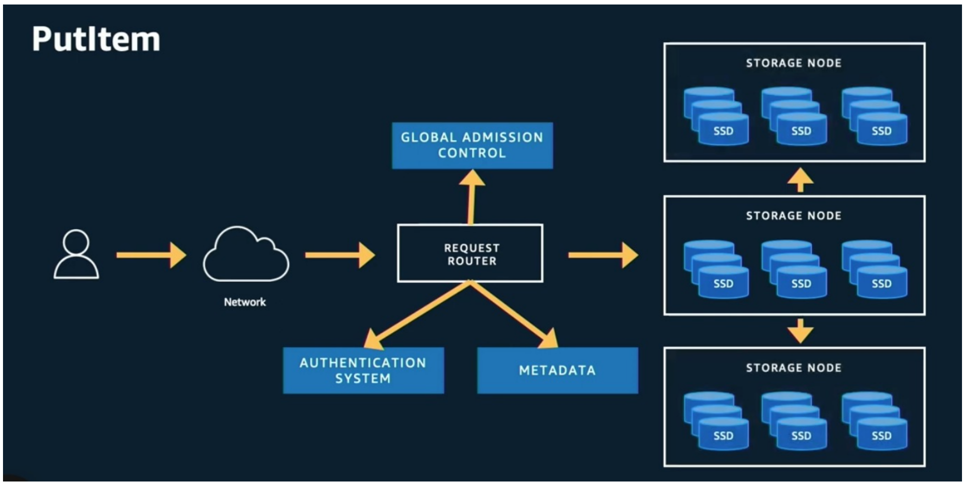
DynamoDB: CRUD API for items

- DynamoDB table: collection of items
- Item: collection of attributes, uniquely identified by a primary key
- Schema of primary key specified at the table creation time
- Primary key schema contains partition key or partition and sort key

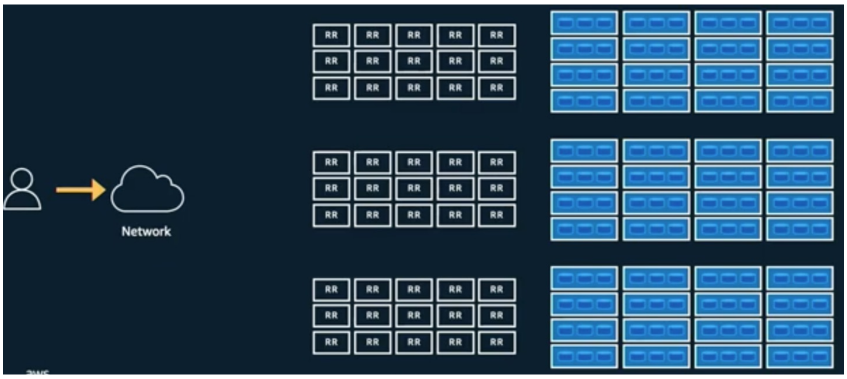


Operation	Description
`PutItem`	Inserts a new item, or replaces an old item with a new item.
`UpdateItem`	Updates an existing item, or adds a new item to the table if it doesn't already exist.
`DeleteItem`	The DeleteItem operation deletes a single item from the table by the primary key.
`GetItem`	The GetItem operation returns a set of attributes for the item with the given primary key.

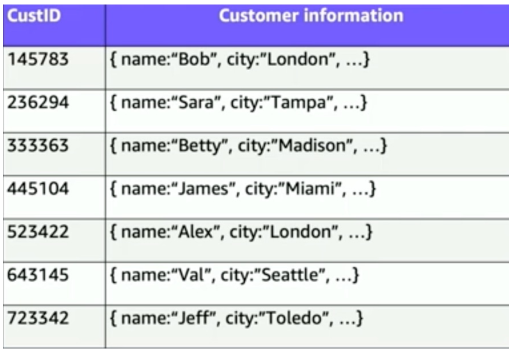
DynamoDB: Put item and architecture



At Amazon scale

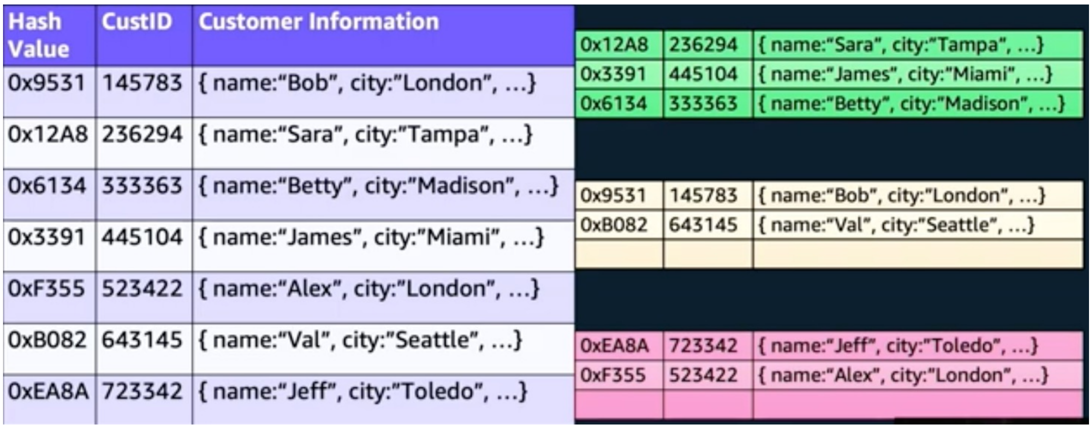


DynamoDB: Table partitioning



CustID	Customer information
145783	{ name:"Bob", city:"London", ...}
236294	{ name:"Sara", city:"Tampa", ...}
333363	{ name:"Betty", city:"Madison", ...}
445104	{ name:"James", city:"Miami", ...}
523422	{ name:"Alex", city:"London", ...}
643145	{ name:"Val", city:"Seattle", ...}
723342	{ name:"Jeff", city:"Toledo", ...}

- DynamoDB table is divided into multiple partitions to handle throughput and storage requirements
- Each partition hosts a disjoint and contiguous part of table's key-range
- Each partition has multiple replicas, which are distributed across different Availability Zones and form a replication group



Hash Value	CustID	Customer Information
0x9531	145783	{ name:"Bob", city:"London", ...}
0x12A8	236294	{ name:"Sara", city:"Tampa", ...}
0x6134	333363	{ name:"Betty", city:"Madison", ...}
0x3391	445104	{ name:"James", city:"Miami", ...}
0xF355	523422	{ name:"Alex", city:"London", ...}
0xB082	643145	{ name:"Val", city:"Seattle", ...}
0xEA8A	723342	{ name:"Jeff", city:"Toledo", ...}

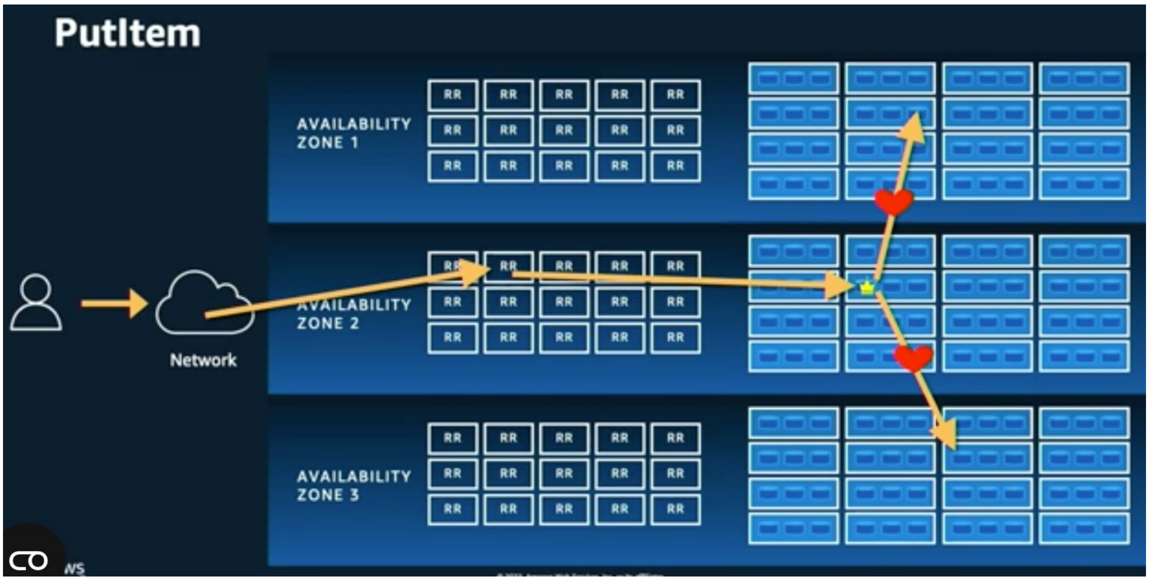
0x12A8	236294	{ name:"Sara", city:"Tampa", ...}
0x3391	445104	{ name:"James", city:"Miami", ...}
0x6134	333363	{ name:"Betty", city:"Madison", ...}

0x9531	145783	{ name:"Bob", city:"London", ...}
0xB082	643145	{ name:"Val", city:"Seattle", ...}

0xEA8A	723342	{ name:"Jeff", city:"Toledo", ...}
0xF355	523422	{ name:"Alex", city:"London", ...}

DynamoDB: High availability

- Replication group uses Multi-Paxos for leader election and consensus
- Only leader replica can serve write and strongly consistent read requests



How to choose the right data store

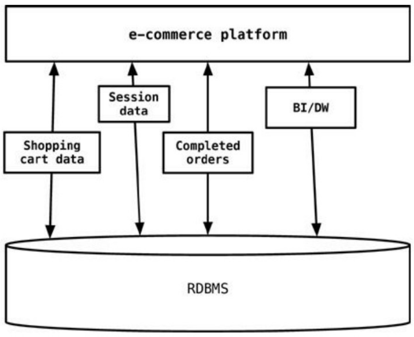
- Bad news: no one type that fits all
- Main factors to take into account
  - Data model, access pattern and data-related features
    - Access pattern: distribution of reads vs writes and random vs sequential access
    - Data-related features: volume, complexity, schema flexibility, durability
  - Query requirements
  - Non-functional properties
    - Performance, (auto-)scalability, consistency, partitioning, replication, load balancing, concurrency mechanisms, CAP tradeoffs, security, license model and cost, support

Performance comparison of NoSQL data stores

- Performance: important factor in choosing the right NoSQL data store solution
  - Throughput and latency as main metrics
- Bad news: many studies (both academic and industrial), no single “winner takes all” among NoSQL data stores
- Unsolved issue: no standard benchmark
  - Some studies use YCSB, others real datasets
  - Yahoo Cloud Serving Benchmark (YCSB): open-source workload generation tool github.com/brianfrankcooper/YCSB
- Be careful
  - Consider workload features: NoSQL data stores are mostly divided into read and write optimized
  - Parameters may be tuned for performance optimization
  - In some studies bias in data store setting (e.g., custom hardware, software setting)

Which data model/store to use?

- Different data models and data stores are designed to solve different problems
- Using a single data store engine for all of the requirements…
  - storing transactional data
  - caching session information
  - traversing graph of customers
  - performing OLAP operations



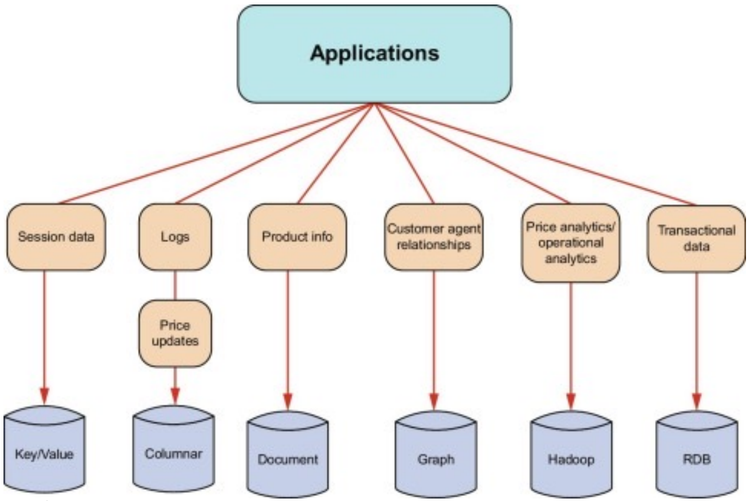
- … usually leads to non-performing solutions
- Also different needs for availability, consistency, etc.

Multi-model data management

- How to manage Variety of Big Data 3V model?
  1. **Polyglot persistence**
  2. **Multi-model databases**

Polyglot persistence

- Use multiple data storage technologies: polyglot persistence
  - Choose multiple data stores based upon the way data are used by apps or their components
  - Different kinds of data are best dealt with different data stores: pick the right tool for the right use case



Multi-model databases

- Second-generation of NoSQL products support multiple data models
- How?
  1. Tightly-integrated polystore: uses jointly multiple data storage technologies, chosen based upon the way data is being used by apps
  2. Single-store multi-model database: using one single, integrated backend

DBMS	Query language	Primary model	Secondary model	Storage strategy
ArangoDB	AQL	Document	Graph, KV	One engine
OrientDB	SQL-like	Graph	Document, KV	One engine
Redis	API	KV	Graph, Document	One engine
DynamoDB	API, PartiQL (SQL compatible)	Document, KV	Graph	Multiple engines
Cosmos DB	API, SQL	-	All	Multiple engines
Oracle 23c	SQL	Relational	All	Both

Multi-model databases

- With respect to polyglot persistence
  - ✓ Decrease in operational complexity and cost
  - ✓ No need to maintain data consistency across separate data stores
  - ✗ Performance not optimized for specific data model
  - ✗ Increased risk of vendor lock-in

References

- Sadalage and Fowler, NoSQL Distilled, Addison-Wesley, 2012
- Fowler, NoSQL Databases
- Golinger at al., Data management in cloud environments: NoSQL and NewSQL data stores, *J. Cloud Comp.*, 2013
- DeCandia et al., Dynamo: Amazon's highly available key-value store, *ACM SOSP 2007*
- Chang et al., Bigtable: a distributed storage system for structured data, *OSDI 2006*
- Lakshman and Malik, Cassandra - a decentralized structured storage system, *LADIS 2009*
- Elhemali et al., Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service, *USENIX ATC 2022*
- Database ranking according to popularity db-engines.com/en/ranking