TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

NoSQL Data Stores

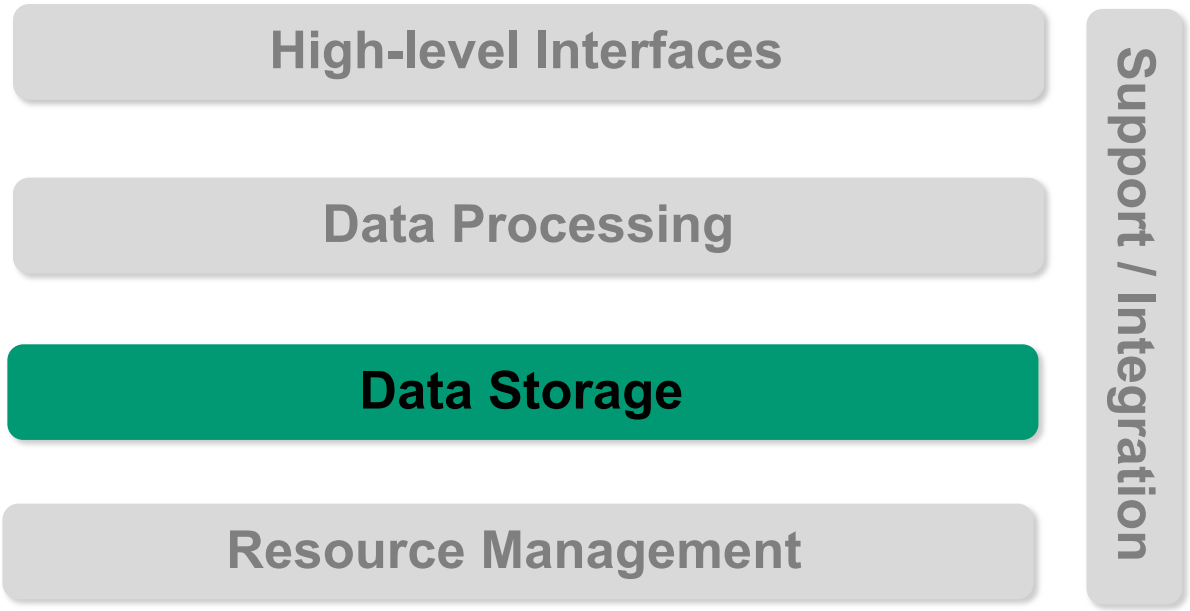
Corso di Sistemi e Architetture per Big Data

A.A. 2024/25

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

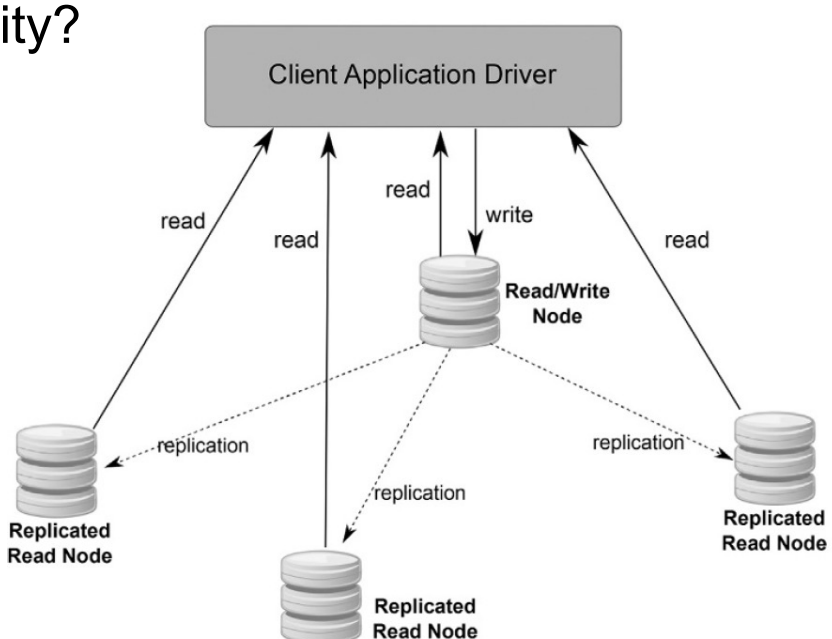
- Let's scale RDBMSs
  - But they were not designed to be distributed
- How to scale RDBMSs?
  - Replication
  - Sharding

Replication

- Primary backup with master/worker architecture

✓ Replication improves read scalability

✗ Write scalability?



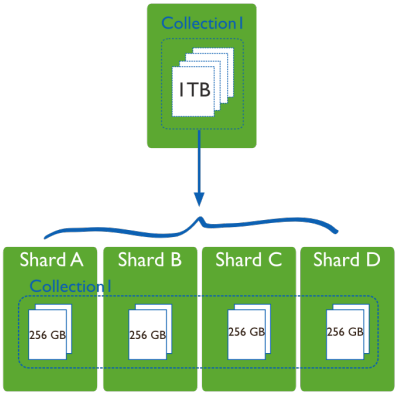
Sharding

- Horizontal partitioning of data across servers

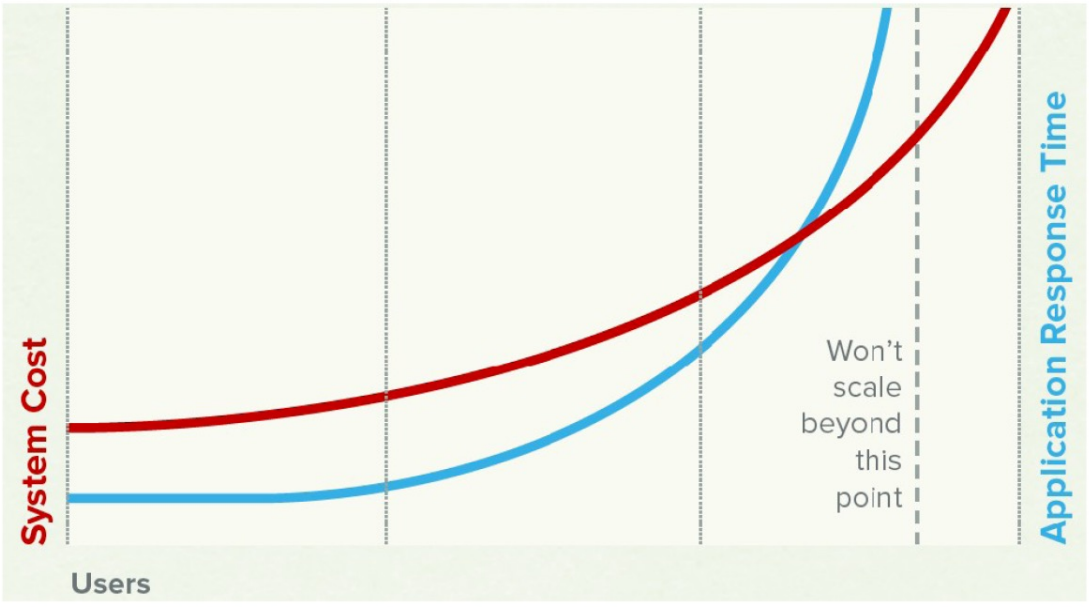
✓ Read and write operations scale

✗ 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

- NoSQL = **Not Only SQL**
  - SQL-style querying is not the crucial objective
- Designed to offer more flexibility and scalability compared to RDBMSs

NoSQL data stores: main features

- Support flexible schema
  - No requirement for fixed rows in table's schema
  - Well suited for agile development process
- Support horizontal scaling
  - Partitioning of data and processing over multiple nodes
- Provide high availability
  - Data replication on multiple nodes, sometimes geo-distributed
- Mainly utilize shared-nothing architecture
  - With exception of graph-based databases
- Often avoid unnecessary complexity by eliminating certain operations, like joins
- Often support weaker consistency models
  - **BASE** rather than ACID: trade-off between consistency and performance

ACID vs BASE: ACID

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

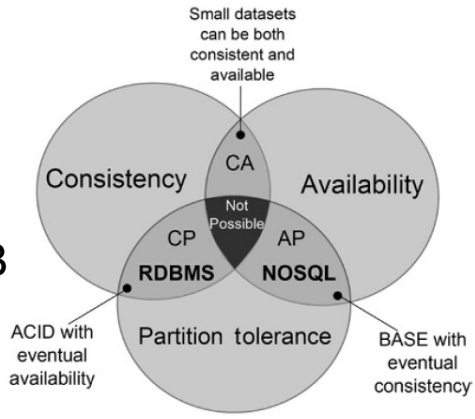
    *Pick two of **C**onsistency, **A**vailability and **P**artition tolerance*
- ACID: traditional approach for RDBMSs
  - *Pessimistic* approach: prevents conflicts from occurring
    - Usually implemented with write locks managed by system
    - Leads to performance degradation and deadlocks (hard to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

ACID vs BASE: BASE

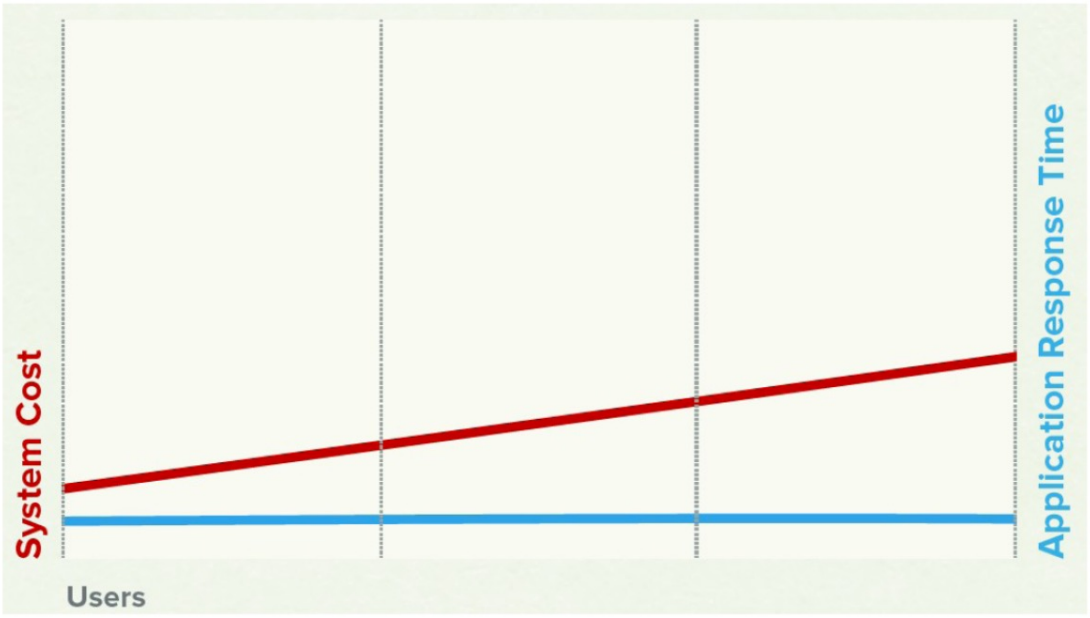
- **BASE**: **B**asically **A**vailable, **S**oft state, **E**ventual consistency
  - **B**asically **A**vailable: the system is available most of the time and there could exist a subsystem temporarily unavailable
  - **S**oft state: data is not durable that is, its persistence is in the hand of the 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 value just before updating
    - *Save both updates*: record that they are in conflict and then merge them

NoSQL and consistency

- Key distinction from RDBMS
  - RDBMS: strong consistency, are either CA systems or CP systems (depending on configuration)
- The 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



NoSQL cost and performance



Source: Couchbase technical report

Pros and cons of NoSQL

Pros

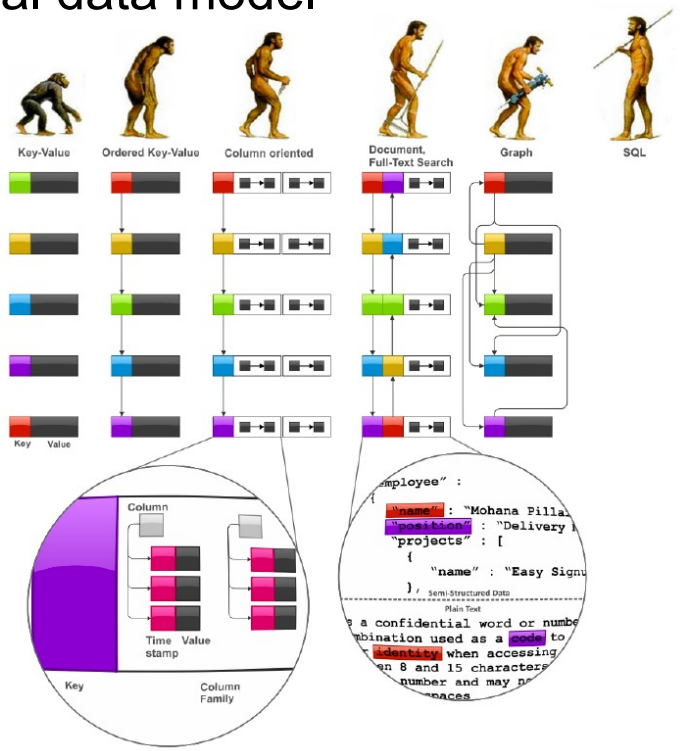
- ✓ Easy to scale-out
- ✓ Higher performance for massive data scale
- ✓ Allow data sharing across multiple servers
- ✓ HA and fault tolerance provided by data replication
- ✓ Many are open-source
- ✓ Support complex data structures and objects
- ✓ No fixed schema, support unstructured data
- ✓ Fast retrieval 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 vendor 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
- Data model is usually independent of storage model
- Data models for NoSQL systems:
  - Aggregate-oriented models: **key-value (KV)**, **document**, and **column-family**
  - **Graph-based** models

Aggregate

- Data as single unit with a complex structure
  - More structure than just a set of tuples
  - E.g., complex record with fields, arrays, records nested inside
- Aggregate pattern in Domain-Driven Design
  https://martinfowler.com/bliki/DDD_Aggregate.html
  - 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

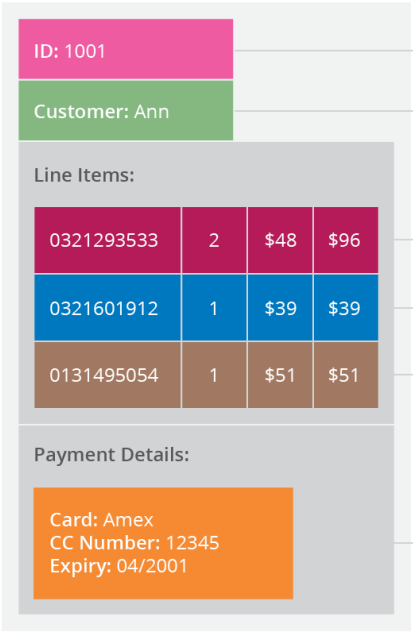
Aggregate

- Pros
  - Easier for application programmers to work with
  - Easier for data store systems to handle ops
- Trade-offs
  - Redundancy: aggregation can lead to data duplication
  - Complexity in updates: careful handling of updates, particularly when updating nested structures or arrays

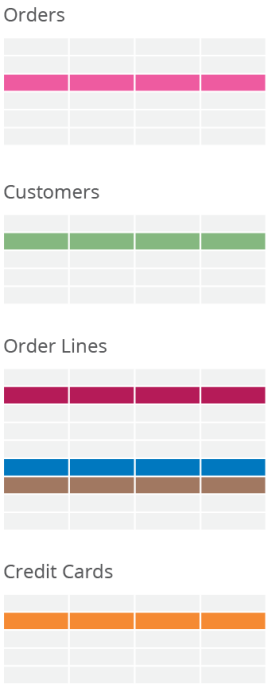
See https://www.thoughtworks.com/insights/blog/nosql-databases-overview

Aggregates: example

- With NoSQL



- With RDBMS



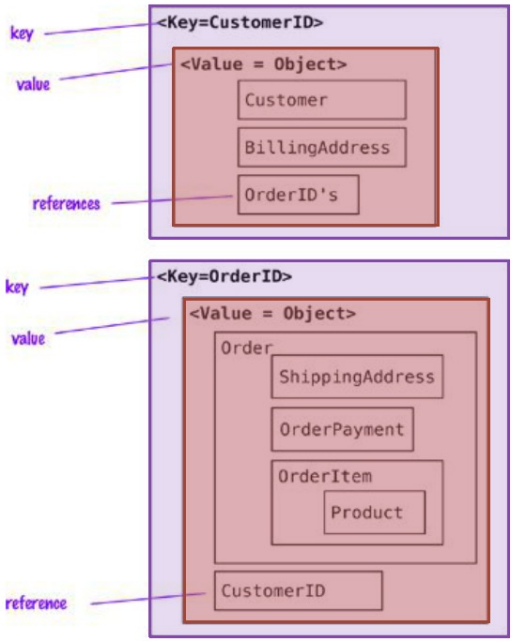
Transactions?

- Relational databases have ACID transactions
- Aggregate-oriented data stores
  - Support atomic transactions, but only *within* single aggregate
  - Most data stores don't support 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 (e.g., lexicographical) so that keys can be efficiently iterated over (e.g., all keys that start with a certain letter, within a certain period of time if the key is a timestamp)
- Some maintain data in RAM, while others employ HDDs, SSDs, flash memory
- Some let developers implement user-defined functions (UDFs) to extend processing capabilities
- Wide range of consistency models

https://www.influxdata.com/key-value-database/

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 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 store allows to search inside the value using a full-text search: see Apache Solr https://solr.apache.org/

KV: 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 user's preferences 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: the most famous example
- Amazon DynamoDB
  - Data model and name from Dynamo, but different design
- Oracle NoSQL Database
  https://www.oracle.com/database/nosql/technologies/nosql/
- Embedded (not distributed) KV stores
  - Berkeley DB
    https://www.oracle.com/database/technologies/related/berkeleydb.html
    - Pre- precursor to NoSQL, key ordering based on Btree+
  - LevelDB https://github.com/google/leveldb
    - By Google, key ordering
  - RocksDB https://rocksdb.org
- Distributed in-memory KV data stores: Memcached, Redis, Hazelcast https://memcached.org https://hazelcast.com
  - Also used alongside another data store as a cache to handle read requests

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 blob of mostly meaningless bits
  - Can store whatever you like in the aggregate
  - Can only access 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 aggregate rather than whole aggregate
  - Can create indexes based on aggregate content
    - 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 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 aggregate into lists or sets
  - Other KV stores support querying by full-text 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 (denormalization)
    - One-to-many: referencing (normalization)
    - One-to-squillions: parent-referencing
  - Some example https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design

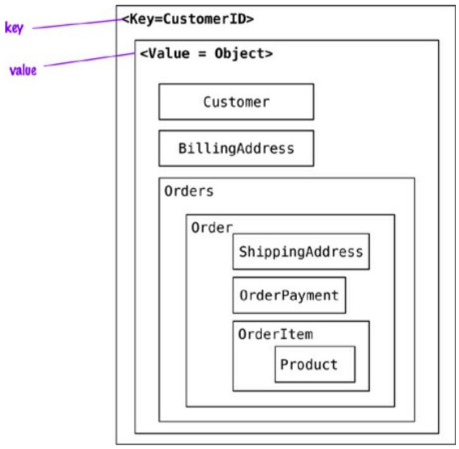
Some data model design choices

- 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



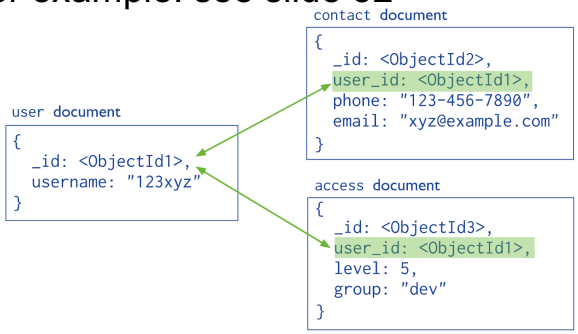
See https://www.mongodb.com/docs/manual/data-modeling/

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) http://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 32



  - In general, use normalization
    - When embedding would result in data duplication without sufficient read performance gains
    - To represent complex many-to-many relationships
    - To model large hierarchical datasets

Document: use cases

- To store and manage large collections of semi-structured data with 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
- Examples
  - Log data
  - IoT data from edge devices
  - Inventory management (e.g., catalogue)
  - Customer data used for personalization

https://www.influxdata.com/document-database/

Document: when not to use

- Complex transactions spanning multiple documents
  - MongoDB supports distributed transactions but they incur greater performance cost over single-document writes
  - Solution: use embedding rather than referencing, but size limit

  In most cases, a distributed transaction incurs a greater performance cost over single document writes, and the availability of distributed transactions should not be a replacement for effective schema design. For many scenarios, the denormalized data model (embedded documents and arrays) will continue to be optimal for your data and use cases. That is, for many scenarios, modeling your data appropriately will minimize the need for distributed transactions. https://www.mongodb.com/docs/manual/core/transactions/
- 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

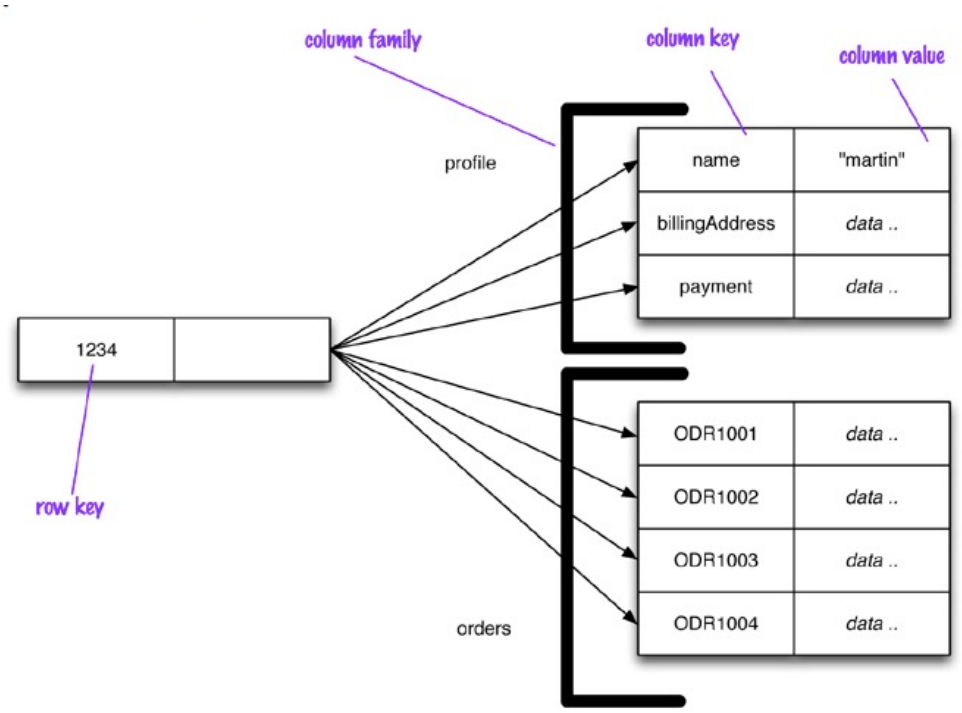
- MongoDB: the most popular
- Aerospike https://aerospike.com
  - KV and document models, tunable consistency
- ArangoDB https://arangodb.com
  - Document and graph models
- Couchbase https://www.couchbase.com
- Apache CouchDB https://couchdb.apache.org
- RavenDB https://ravendb.net/
  - ACID, Raft-based multi-master replication, student licence
- Cloud services
  - Amazon DocumentDB (compatible with MongoDB) https://aws.amazon.com/documentdb/
  - Microsoft Azure CosmosDB https://azure.microsoft.com/services/cosmos-db/

Column-family: data model

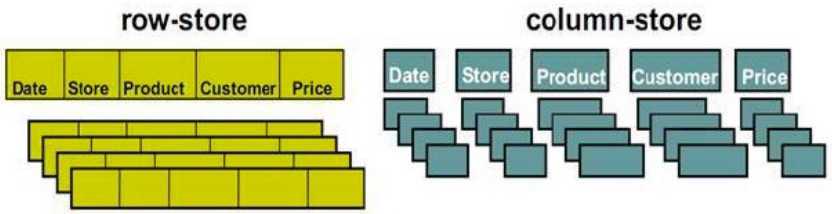
- Strongly aggregate-oriented
  - Lots of aggregates
  - Each aggregate has a key
- Data model: two-level map structure
  - A set of <row-key, aggregate> pairs
  - Each aggregate is a group of pairs <column-key, value>
  - Column: a set of data values of a particular type
- Similar to key-value store, 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



- Row-store systems: store and process data by rows
  - RDBMSs support indexes to improve performance of set-wide operations on whole tables
- Column-store systems: store and process data by columns
  - Faster data access rather than scanning and discarding unwanted data in row, e.g., for aggregate queries (avg, max, …)
  - Examples: C-Store (pre-NoSQL), OpenText https://www.opentext.com/products/analytics-database, MariaDB ColumnStore https://mariadb.com/
  - Do not confuse column-store with column-family

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 44: `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: 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 (PBs)
- Apps geographically distributed over multiple data centers
  - See Cassandra geo-distribution

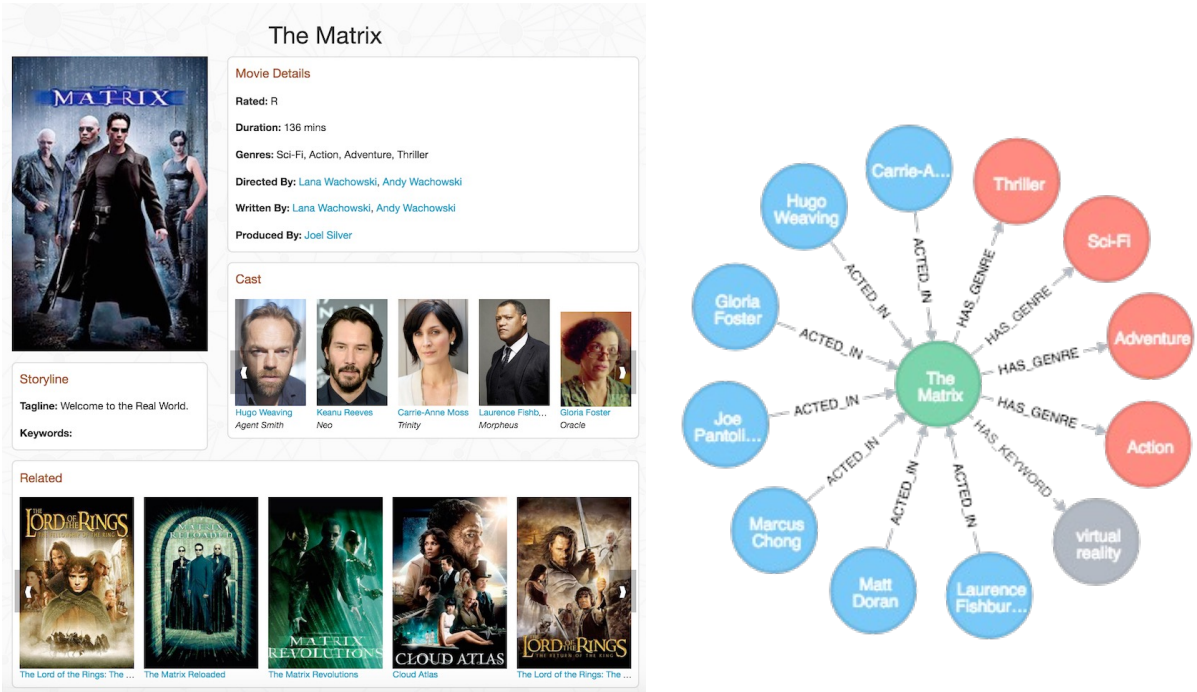
Column-family: products

- 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 https://phoenix.apache.org: SQL query engine on top of HBase
- Other popular column-family data stores
  - Apache Accumulo https://accumulo.apache.org/: based on Bigtable, HDFS to store data and Zookeeper for consensus
    - Different APIs and 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



https://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 for Neo4j
  - Gremlin https://tinkerpop.apache.org/gremlin.html: functional, data-flow language

Graph databases: pros and cons

- Pros:
  - ✓ Explicit graph structure
  - ✓ Index-free adjacency: since each node knows its adjacent nodes, performance of traversal does not depend on graph scale
  - ✓ Support for graph algorithms
  - ✓ Support for graph indexing to make lookups more efficient
  - ✓ Flexibility to handle complex and evolving data structures without requiring schema modifications
- Cons:
  - ✗ Horizontal scalability more difficult to achieve wrt to other models
    - Data sharding on multiple servers: traversing multiple servers is less efficient
  - ✗ Require more design effort with respect to SQL

https://www.oracle.com/it/autonomous-database/what-is-graph-database/

https://www.influxdata.com/graph-database/

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
  - Distributed architecture is more challenging
  - Ad-hoc graph-based query languages
  - ACID guarantees: transactions maintain consistency over multiple graph nodes and edges

Graph databases: use cases

- To model entities and relationships and query for relationships between entities, e.g.,
  - Social networking
  - Dependency analysis
  - Recommender systems
  - Fraud detection
  - Drug discovery
  - Network security
- To perform traversal queries based on connections and apply graph algorithms
  - To find patterns, paths, communities, influencers, single points of failure, and other relationships

Graph databases: products

- Neo4j
- InfiniteGraph https://infinitegraph.com
  - Proprietary, distributed
- MemGraph https://memgraph.com
  - Open source, in-memory
- NebulaGraph https://www.nebula-graph.io
  - Open source, distributed
- OrientDB http://orientdb.org
  - Open source, distributed, multi-model
- Apache Tinkerpop https://tinkerpop.apache.org
  - Graph computing framework for OLTP and OLAP that uses Gremlin as query language
- Cloud services:
  - Amazon Neptune https://aws.amazon.com/neptune
  - Azure Cosmos DB: multi-model

Case studies

- Key-value data stores
  - Amazon's Dynamo
  - 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