INSA
INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
RENNES

NoSQL Databases

Nikos Parlavantzas

ueb
UNIVERSITÉ
EUROPÉENNE
DE BRETAGNE

Lecture overview

- Objective
  - Present the main concepts necessary for understanding NoSQL databases
  - Provide an overview of current NoSQL technologies

Outline

- Main concepts
  - Motivation for NoSQL
  - Data models
  - Distribution models
  - Consistency
- NoSQL databases
  - Key-value stores
  - Document databases
  - Column-family stores
  - Graph databases

Relational databases

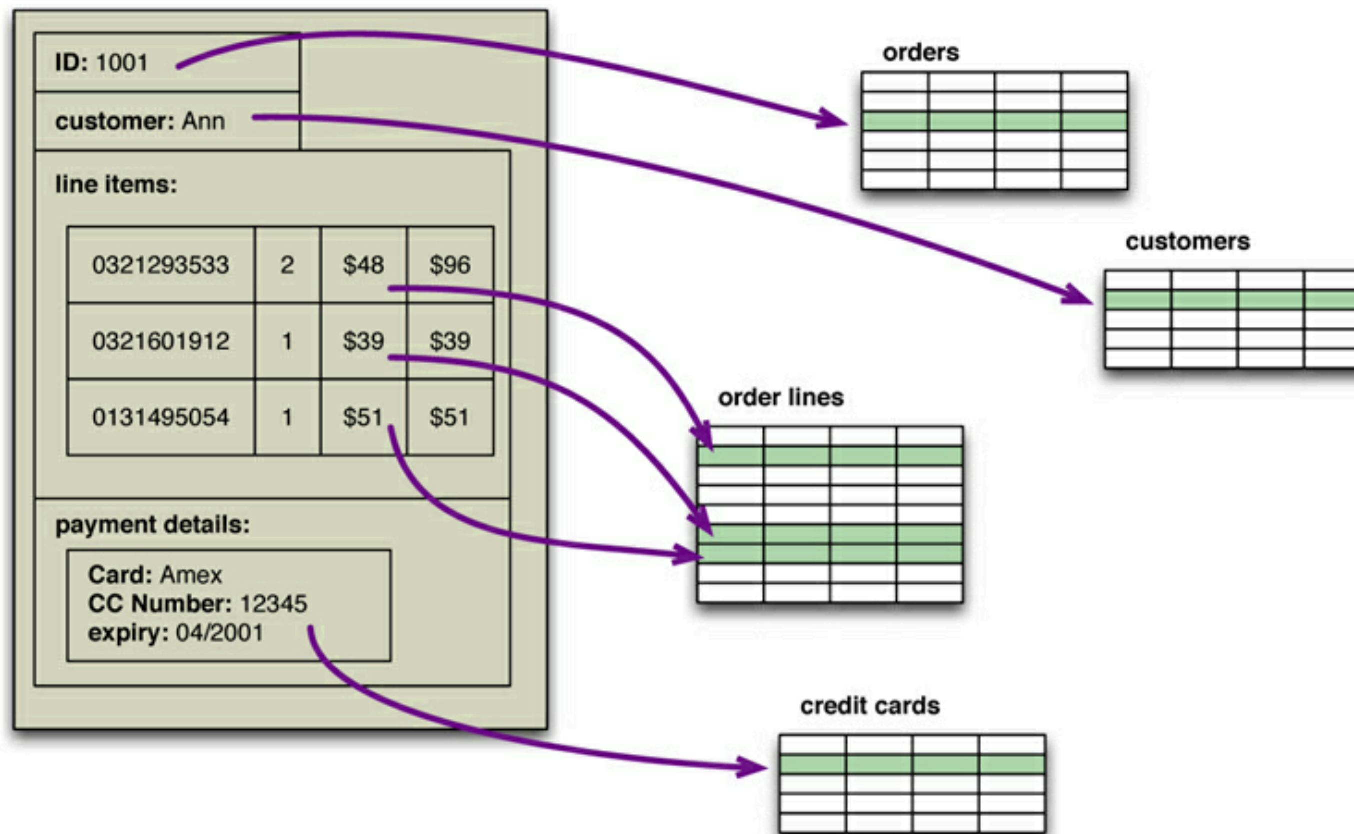
- Dominant model for the last 30 years
- Standard, easy-to-use, powerful query language (SQL)
- Reliability and consistency in the face of failures and concurrent access
  - Support for transactions (ACID properties)

Weaknesses

- Relational databases are not designed to run on multiple nodes (clusters)
  - Favour vertical scaling
  - Cannot cope with large volumes of data and operations (e.g., Big Data applications)

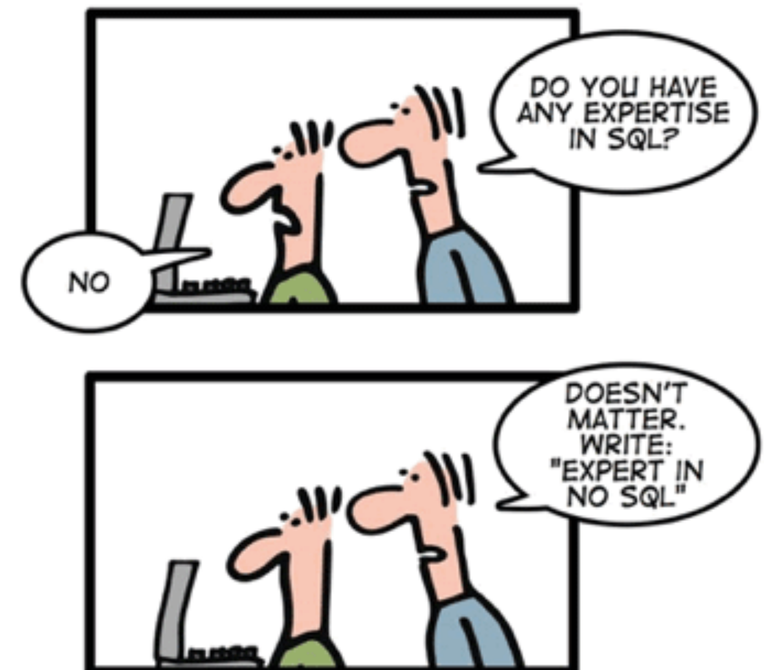
Weaknesses

- Mapping objects to tables is notoriously difficult (impedance mismatch)



NoSQL

- Definition
  - Term appeared in 2009
  - “No SQL” or “Not Only SQL”
  - Practically, anything that deviates from traditional relational database systems (RDBMSs)



NoSQL

- Common characteristics
  - Not supporting the relational model
  - Running well on clusters
  - Not needing a schema (schema-free)
  - Typically, relaxing consistency

Advantages

- Elasticity
- Big data
- Automated management
- Flexible data models

Data models

Data models

- Key-value
- Document
- Column family
- Graph

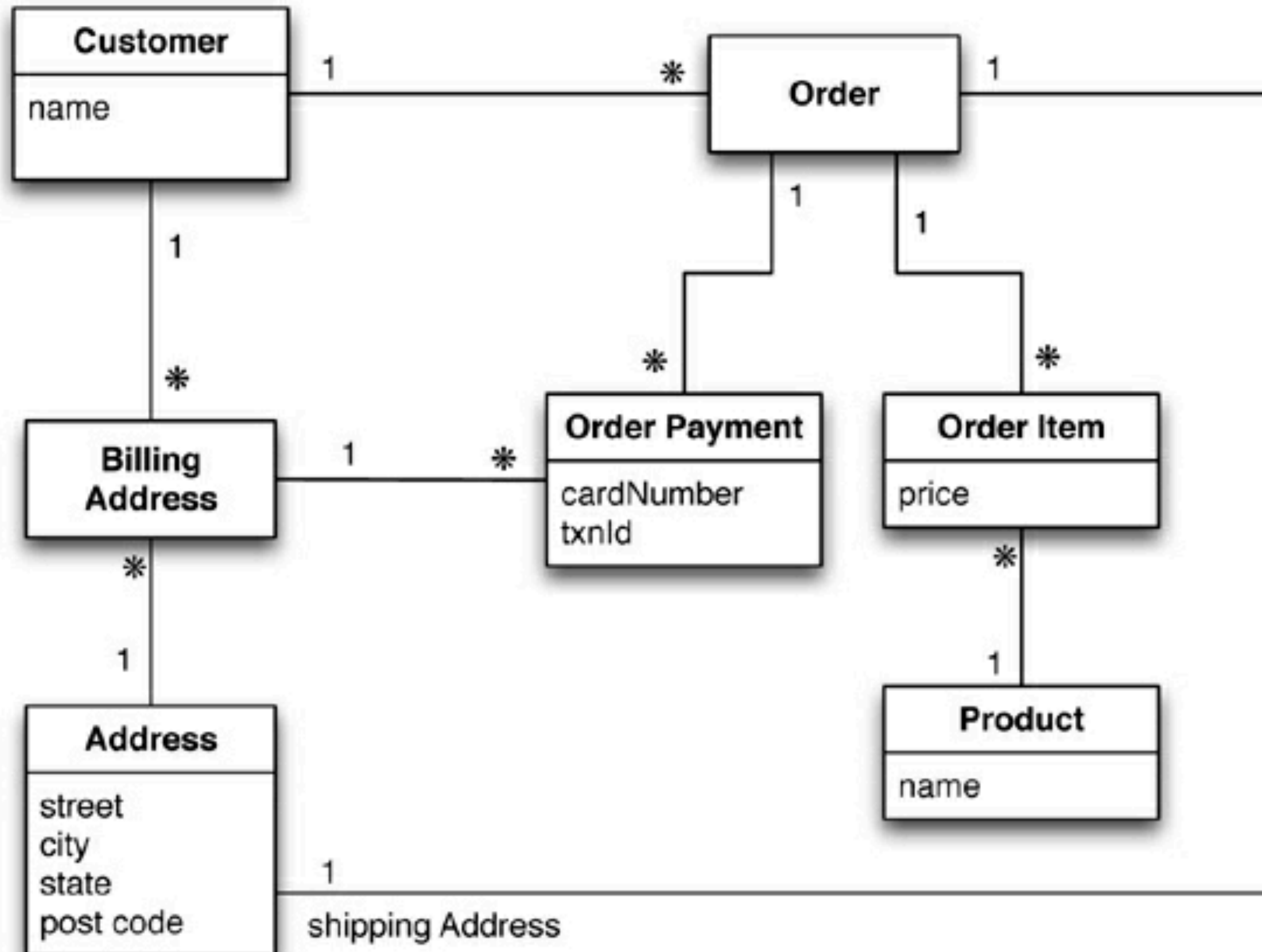
Aggregate oriented (Key-value, Document, Column family)

Schema free (Key-value, Document, Column family, Graph)

Aggregate oriented

- **Aggregate**: a collection of data with complex structure, manipulated as a unit
- Unit of data retrieval
- Unit of atomic update

Example relational model



Sample tables

Customer

Id	Name
1	Martin

Orders

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

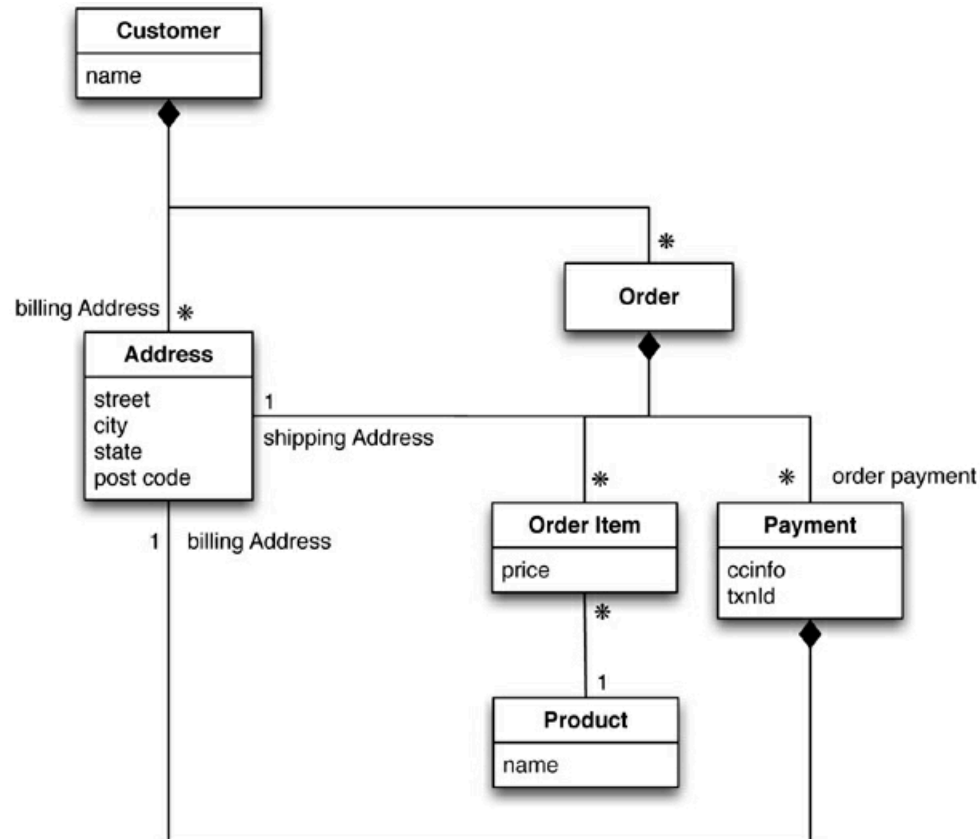
Address

Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Aggregate model




```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
    {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
    }],
  }]
}
}
```

Aggregate orientation

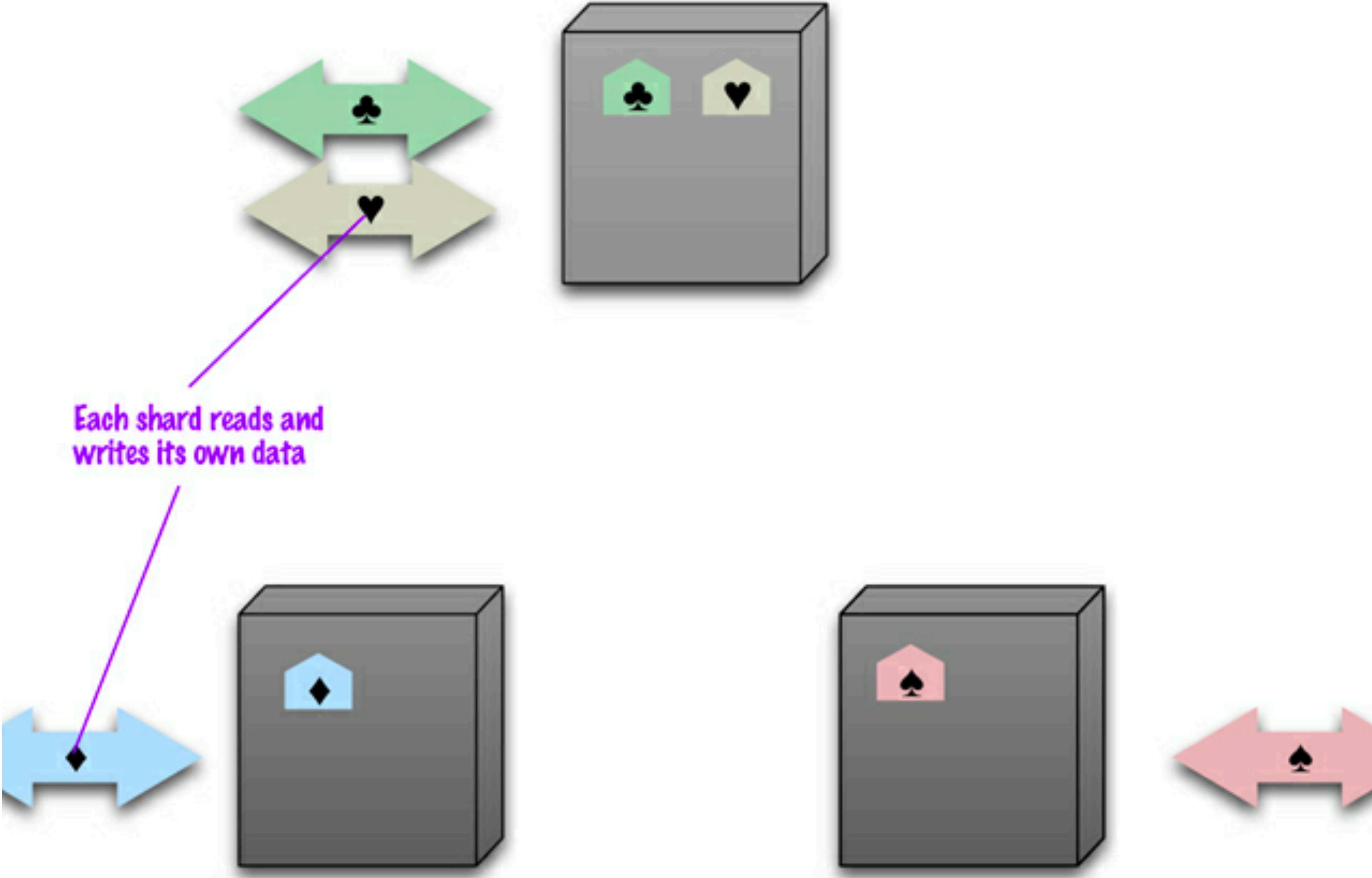
- Addresses impedance mismatch
- Helps with running on a cluster
  - Provides information to the database about which data is manipulated together and thus should live on the same node
- BUT makes some manipulations difficult
  - Does not support unanticipated ways of organising data (unlike RDBMSs)

Being schema-free

- No need to create a schema before storing data into a NoSQL database
- Facilitates evolution
- Handles non-uniform data (e.g., records with different fields)
- BUT knowledge about structure remains in application code (implicit schema)
  - May be problematic if multiple applications access same database

Distribution models

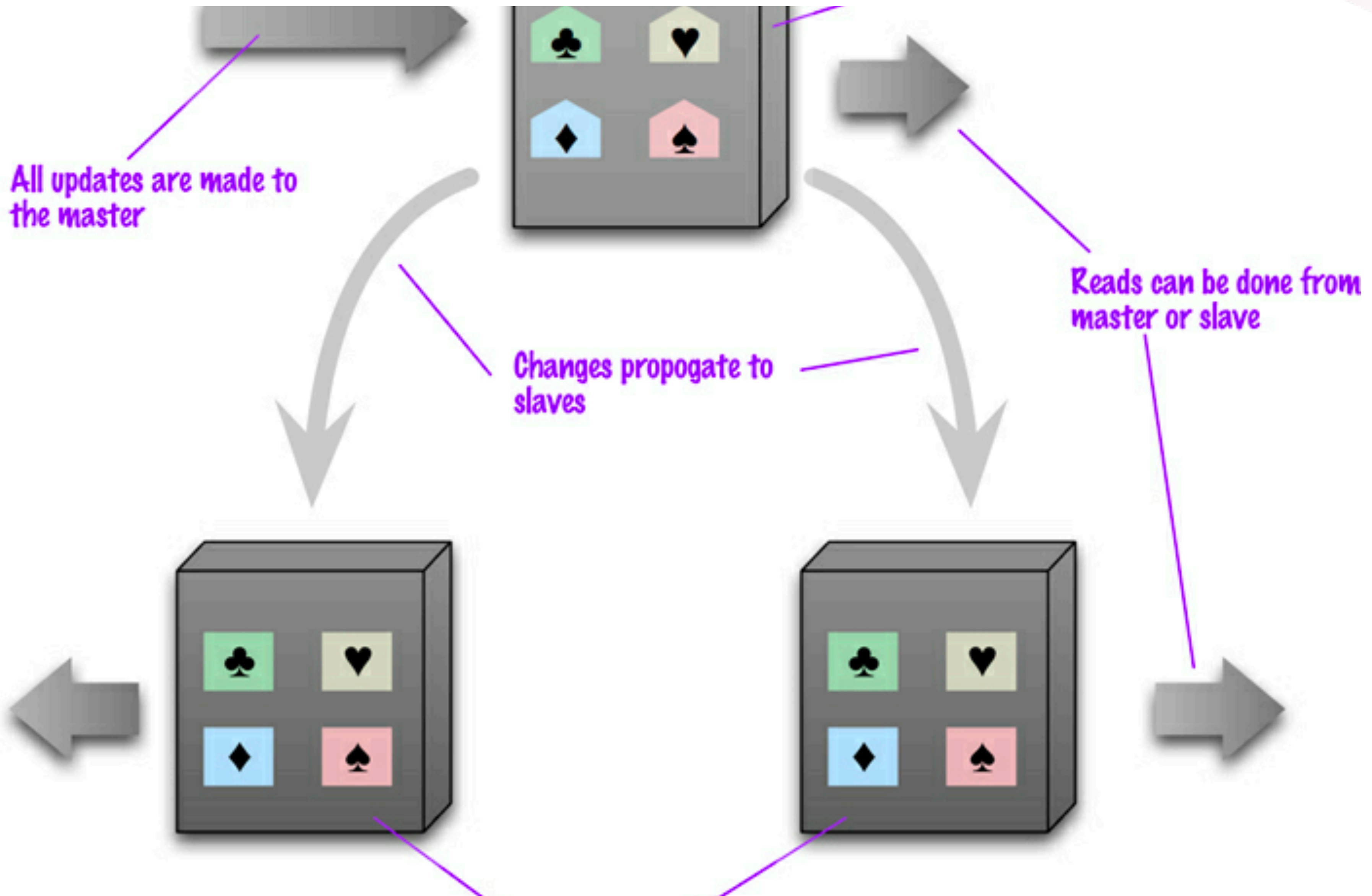
Sharding



Sharding

- Improves both read and write performance
- Does not provide resilience
- Distributing data becomes challenging
  - Ideally, a user interacts with one node, and load is balanced between nodes
  - Aggregate orientation helps
  - Auto-sharding is provided by some NoSQL databases

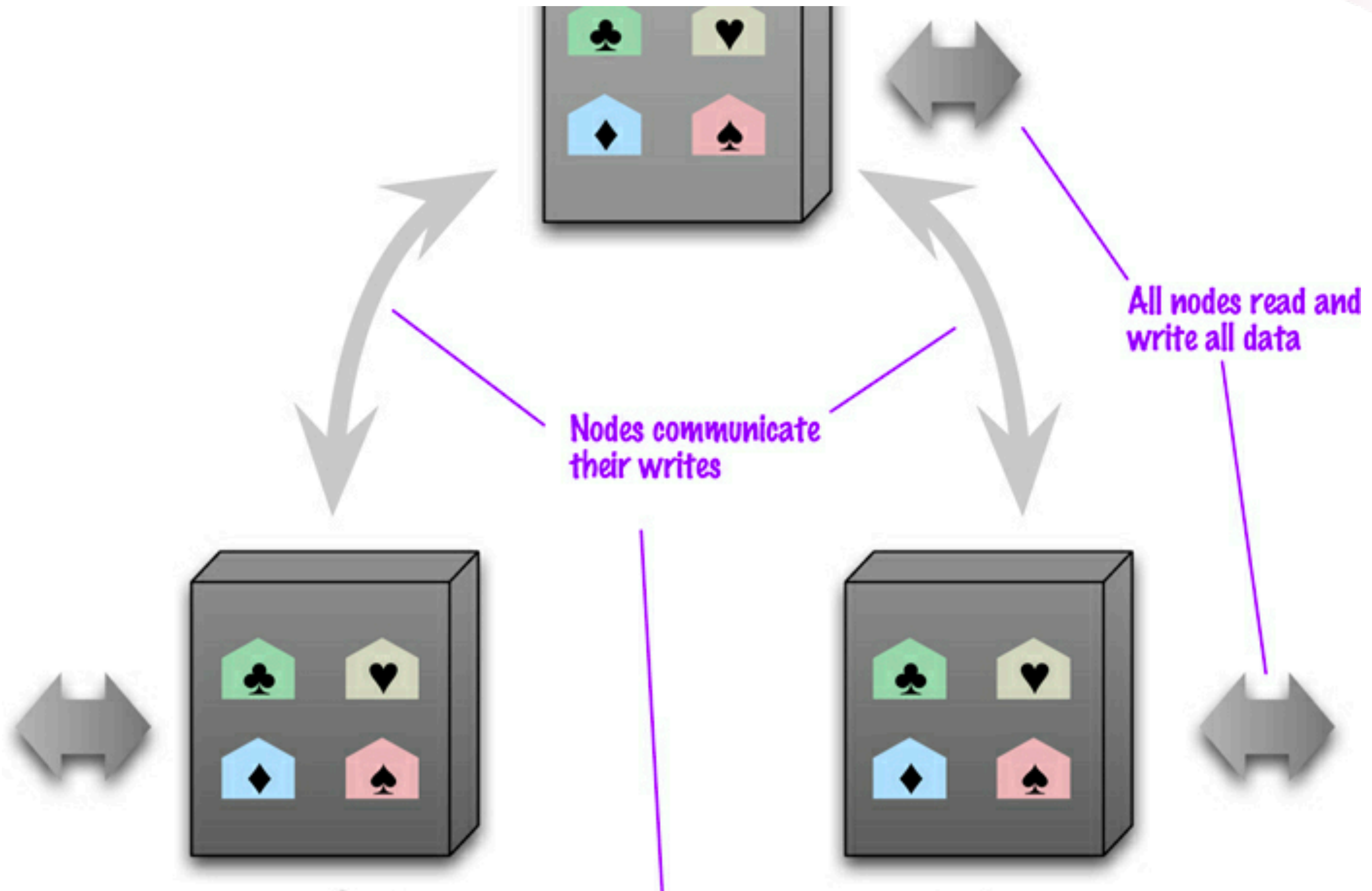
Master-slave replication



Master-slave replication

- Read scalability
  - More slaves → more read requests
  - Does not help with write load
- Read resilience
  - Slaves can still handle read requests after a master failure
- Possibility of inconsistency
  - Clients of different slaves may see different values

Peer-to-peer replication



Peer-to-peer replication

- Scalability and resilience
  - No central element to form bottleneck or single point of failure
  - Easy to add nodes and to handle node failures
- Maintaining consistency is challenging
  - Wide range of options (e.g., coordinating to avoid conflicts, merging inconsistent writes)

Possible combinations

master for two shards

slave for two shards

master for one shard

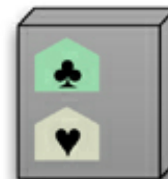
Master-slave & sharding

master for one shard and slave for a shard

slave for two shards

slave for one shard

Peer-to-peer & sharding



Consistency

Reminder: CAP Theorem

- A networked shared-data system can satisfy at most 2 of the following properties at a time
  - **C**onsistency
  - **A**vailability
  - **P**artition tolerance
- Partitions are inevitable; must trade-off consistency and availability
- Consistency is also traded off against latency

Client-side consistency

- Strong consistency
  - After the update completes, any subsequent access will return the updated value
- Eventual consistency
  - If no new updates are made, eventually all accesses will return the last updated value

Server-side consistency

N = # replicas of the data

W = # replicas that need to confirm a write

R = # replicas contacted for a read

- W+R > N: strong consistency
- W+R<=N: eventual consistency

- Can configure N, W, R to tune consistency, availability, read/write performance
  - W=N, R=1: optimise for read; any failure makes write unavailable
  - W=1, R=N: optimise for write, any failure makers read unavailable, etc.

Key-value stores

Key-value stores

- Basically, a simple hash table where all access is done via a key
- Basic operations
  - Get the value for a key
  - Put the value for a key
  - Delete a key and its value
- Example systems
  - Redis, Riak, Memcached, …

Key-value stores

- When to use
  - Storing session Information, user profiles, preferences
  - Storing shopping cart data
- When not to use
  - Need for relationships among data
  - Transactions involving multiple keys
  - Querying by data inside value part
  - Operating on multiple keys at the same time

Riak

- Open-source, distributed key-value store
  - First release: 2010
  - Inspired by Amazon’s Dynamo
- Written in Erlang
- Decentralised architecture
- Built-in MapReduce support

Data access

- Default interface is HTTP
  - GET (retrieve), PUT (update), POST (create), DELETE (delete)
- Keys are stored in buckets
  - namespaces with common properties (e.g., replication factor)
  - Keys may be user-specified or generated by system
- Paths:
  - /riak/<bucket>
  - /riak/<bucket>/<key>

Examples

- Store a JSON file into a bucket

```
curl -v -X PUT
http://localhost:8091/riak/animals/ace -H
"Content-Type: application/json”
-d '{"nickname" : "The Wonder Dog", "breed" :
"German Shepherd"}’
```

- Retrieve a value from a bucket

```
curl
http://localhost:8091/riak/animals/6VZc2o7zKxq2B3
4kJrm1SOma3PO
```

Links

- Links are metadata that establish one-way relationships between objects
- Stored in the HTTP Link header

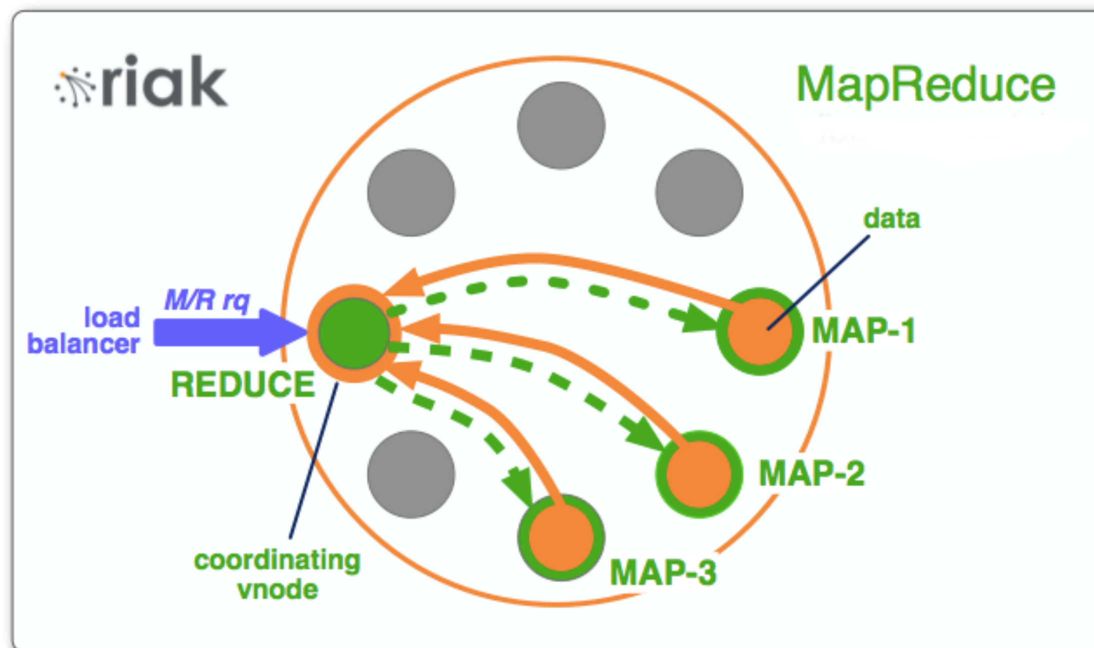
```
curl -X PUT http://localhost:8091/riak/cages/1 \
-H "Content-Type: application/json" \
-H
"Link:</riak/animals/polly>;riaktag=\"contains\”
-d '{"room" : 101}’
```

- Used to retrieve data through ‘link walking’

```
curl
http://localhost:8091/riak/cages/1/_,contains,_
```

Advanced queries

- MapReduce
  - Clients submit map/reduce functions (in JavaScript or Erlang) and lists of keys. The functions are executed in parallel on nodes where values are located.

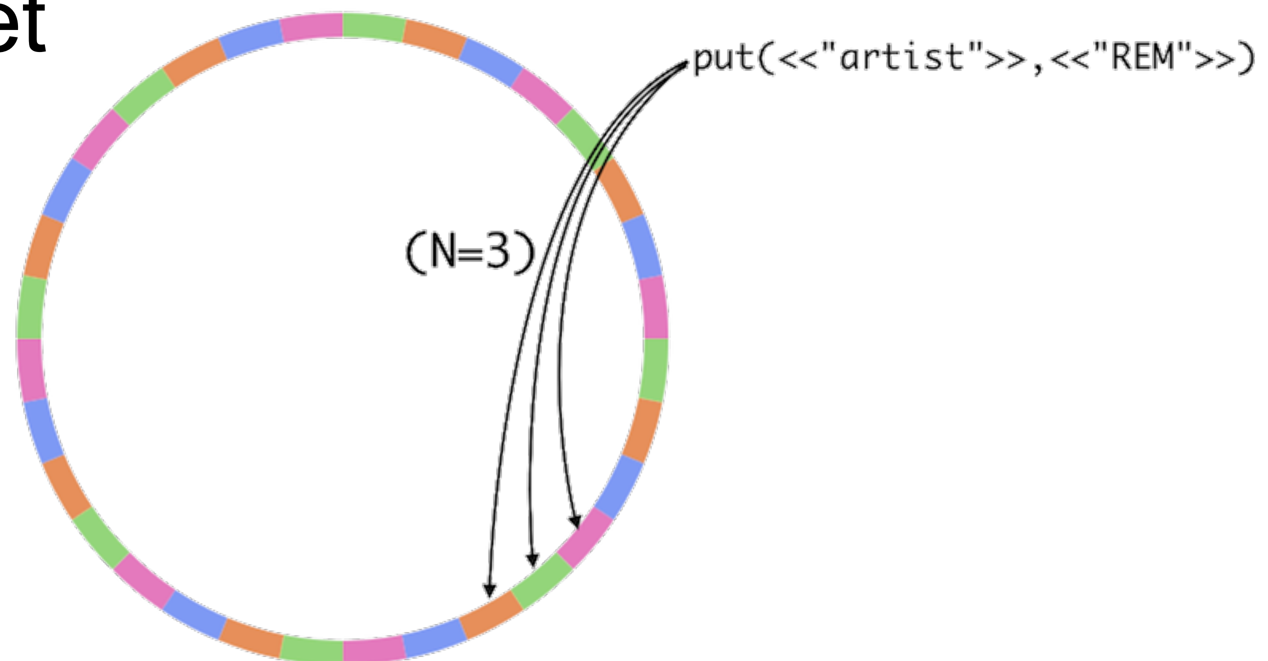


Advanced queries

- Secondary indexes
  - Tagging objects with metadata and supporting exact match and range queries on metadata
- Riak search
  - Full-text search and indexing using the objects' values

Distribution model

- Data is automatically distributed around the cluster based on keys (consistent hashing)
- The number of replicas (N) can be set for each bucket



Consistency

- Clients can configure W,R *per request* to achieve desired levels of consistency, performance, availability
- Common values for W, R
  - One, All (i.e., N), Quorum (i.e., N/2+1)

N = # replicas of the data

W = # replicas that need to confirm a write

R = # replicas contacted for a read

Horizontal scaling

- Can dynamically add and remove nodes; data is redistributed automatically
- All nodes are equal

Document databases

Document databases

- Main concept is document
  - self-describing, hierarchical data structures
  - JSON, BSON, XML, etc.
- Similar to key-value stores where the values are examinable
- Example systems
  - MongoDB, Couchbase, Terrastore, RavenDB, Lotus Notes, etc.

Document databases

- Documents are organised into collections
- The structure of documents in the same collection can be different
  - No predefined schema
- New attributes can be created without the need to define them or to change the existing documents

Document databases

- When to use
  - Event Logging
  - Content management systems, blogging platforms
  - Web analytics or real-time analytics
  - E-Commerce applications
- When not to use
  - Need for complex transactions spanning different documents

MongoDB

- Open-source document database
  - First released in 2009
- Written in C++
- JSON documents with dynamic schemas
- Ad hoc queries
- Replication, automatic sharding
- MapReduce support

Data model

- JSON stored as BSON (binary representation)

RDBMS	MongoDB
Database	Database
Table, View	Collection
Row	Document (JSON, BSON)
Column	Field
Index	Index
Join	Embedded Document
Foreign Key	Reference
Partition	Shard

```
> db.user.findOne({age:39})
{
	"_id" :
ObjectId("5114e0bd42…"),
	"first" : "John",
	"last" : "Doe",
	"age" : 39,
	"interests" : [
		"Reading",
		"Mountain Biking ]
	"favorites": {
		"color": "Blue",
		"sport": "Soccer"}
}
```

Example operations

```
> db.user.insert({
	first: "John",
	last : "Doe",
	age: 39
})
```

```
> db.user.find ()
{
	"_id" : ObjectId("51…"),
	"first" : "John",
	"last" : "Doe",
	"age" : 39
}
```

```
> db.user.update(
	{"_id" :
ObjectId("51…")},
	{
		$set: {
			age: 40,
			salary: 7000}
	})
```

```
> db.user.remove({
	"first": /^J/
})
```

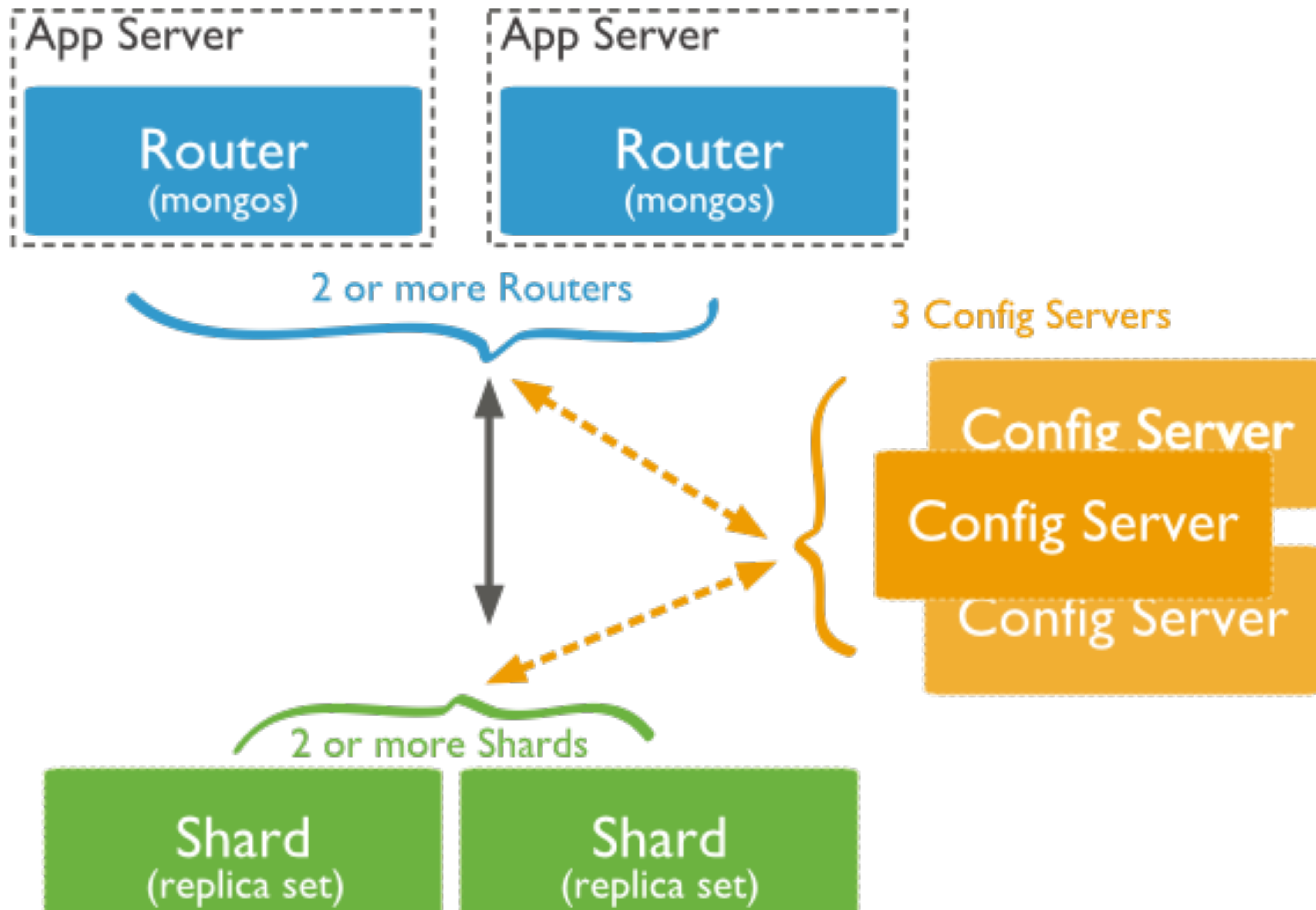
Queries

- Complex queries using ranges, set inclusion, comparison, negation, sorting, counting, etc.
- Querying nested structured data
- MapReduce support

Distribution model

- Master-slave replication
  - Availability, read performance
  - Automatic failover
    - If master fails, a new master is elected
- Automatic sharding
  - Adding /removing shards
  - Automatic data balancing using migrations

Distribution model



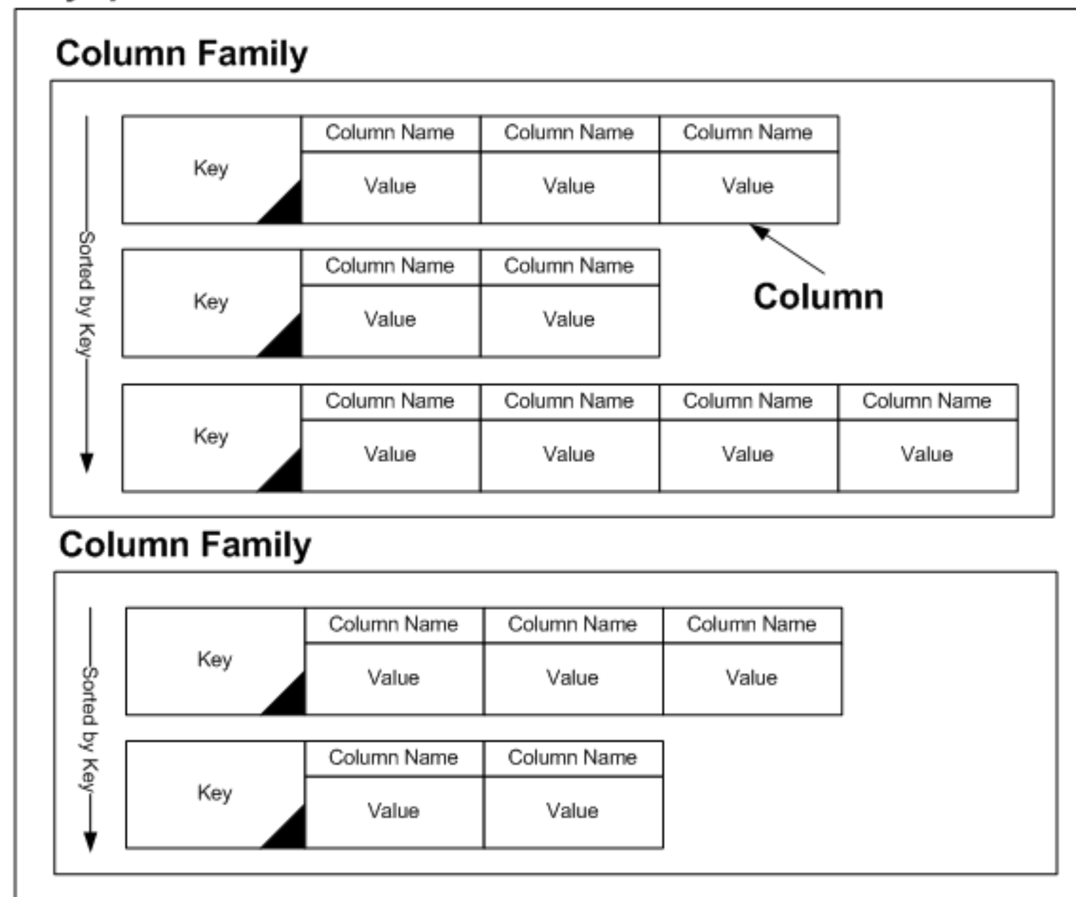
Consistency

- The level of guarantee can be configured per operation
  - Allows setting W (number of servers that need to confirm the update)

Column-family stores

Column-family stores

- Column family: an ordered collection of rows, each of which is an ordered collection of columns



Column-family stores

- Column families are predefined; columns are not
  - Can freely add columns
  - Can have rows with thousands of columns
- Data for a particular column family is usually accessed together
- Columns have attached timestamps
- Example systems
  - Cassandra, HBase, SimpleDB, …

Column-family stores

- When to use
  - Event Logging, content management systems, blogging platforms
  - Data that expires
- When not to use
  - Complex queries
  - Aggregating data (needs MapReduce)
  - Unknown query patterns

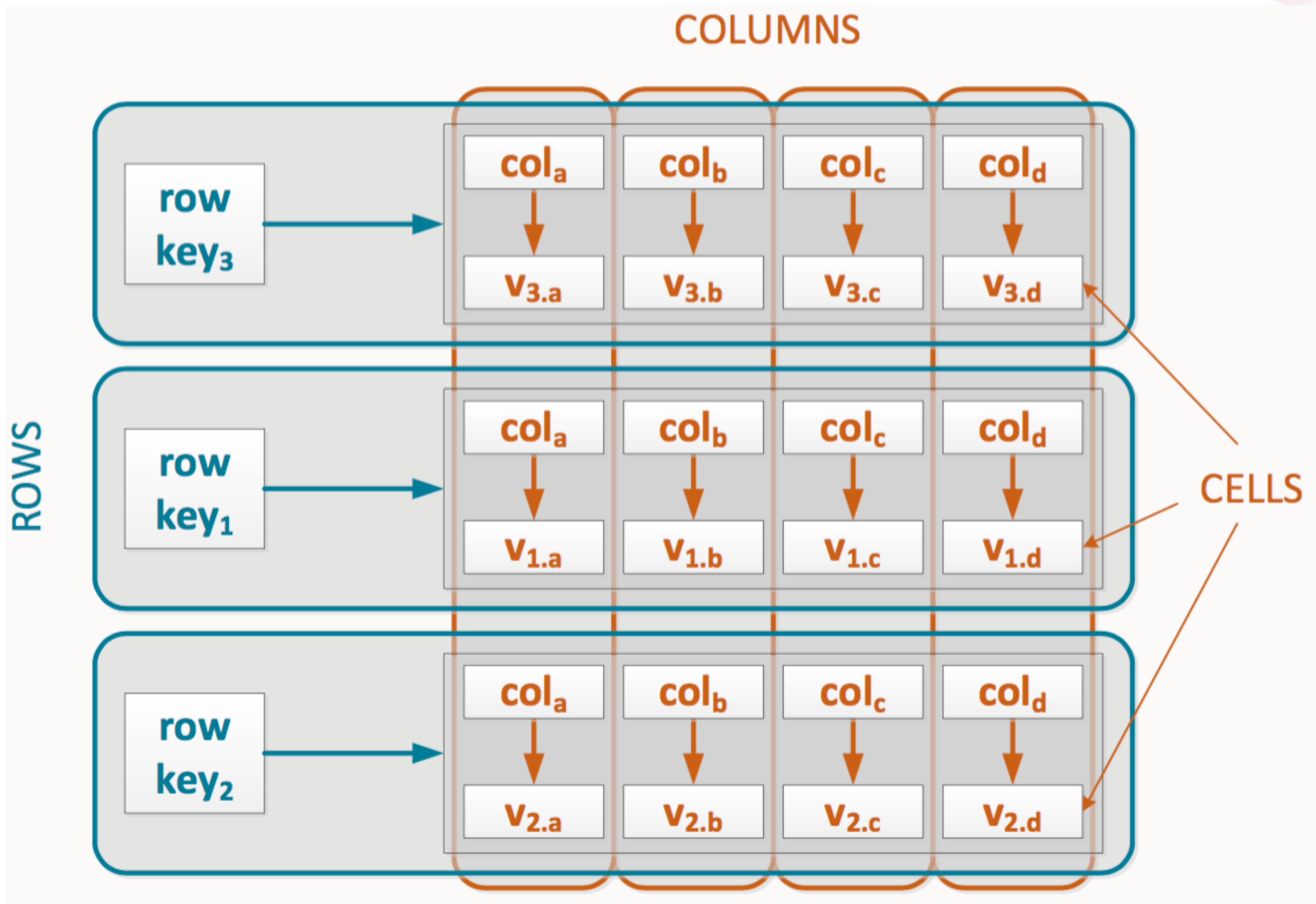
Cassandra

- Open-source distributed database
  - Developed in Facebook
  - Initial release 2008
- Written in Java
- Decentralised architecture
- MapReduce support
- CQL (Cassandra Query Language)

Data model

- A keyspace contains a set of tables
- A table contains a set of of rows
- A table has a primary key, a sequence of columns consisting of partition key columns and (optionally) clustering key columns
- A partition is a set of rows that share the same value for the partition key
- Rows in a multi-row partition are ordered by clustering key values

Data model



Data model

- Table with single-row partitions

partition key

columns

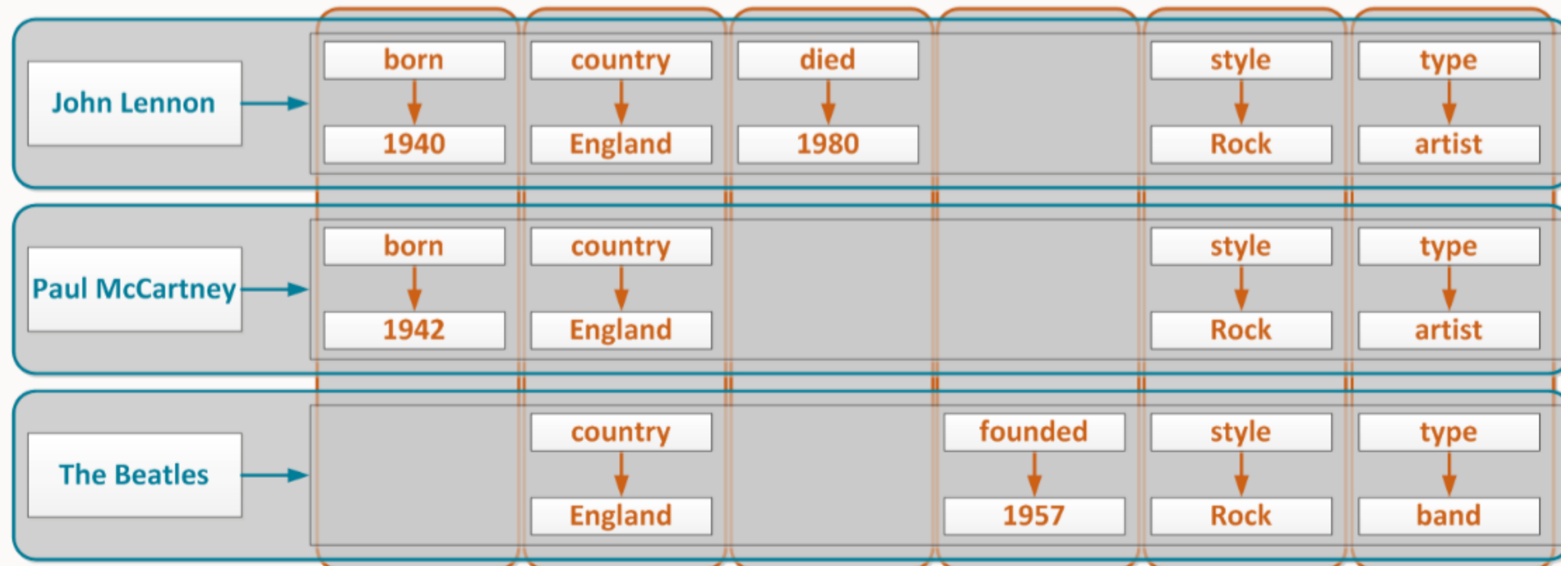
performer	born	country	died	founded	style	type
John Lennon	1940	England	1980		Rock	artist
Paul McCartney	1942	England			Rock	artist
The Beatles		England		1957	Rock	band

partitions

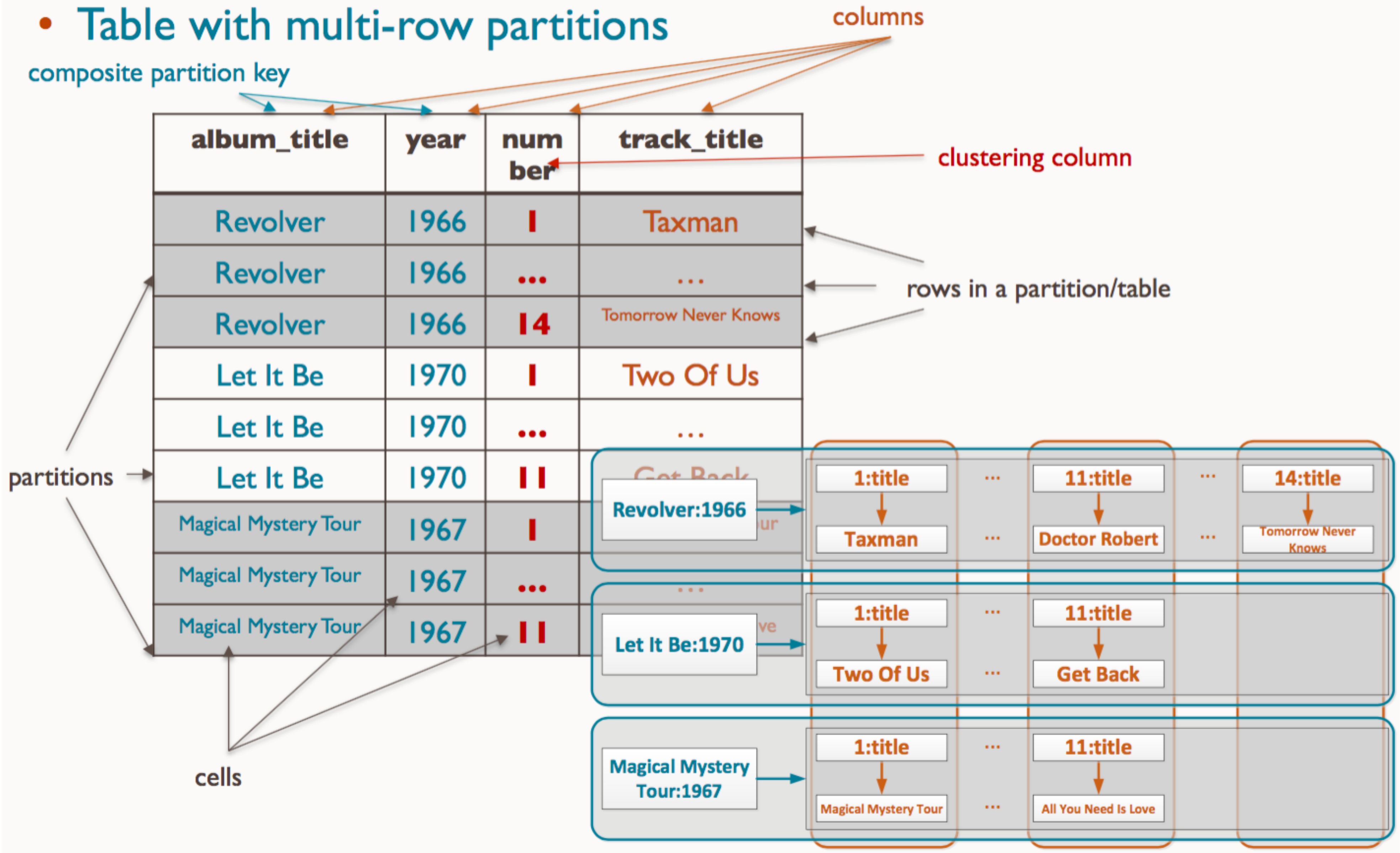
rows

cells

- Column family view



Data model



Queries

- **CQL**: SQL-like language adapted to the Cassandra data model and architecture
  - Does not support joins or sub-queries
  - Enables modifying tables dynamically without blocking updates and queries
  - Emphasises denormalisation (e.g., supports collection columns)

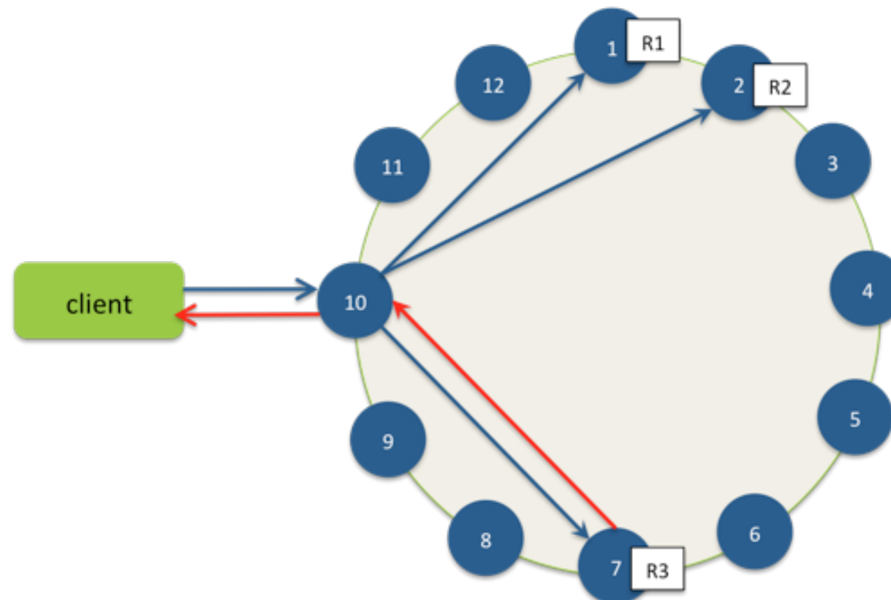
```
CREATE TABLE users (
    user_id text PRIMARY KEY,
    first_name text,
    last_name text,
    emails set<text> );
```

Cassandra Data Modelling

- Model around the queries
  - Determine what queries to support
  - Create tables that can satisfy the queries by reading the minimum number of partitions
    - Each partition may reside on a different node

Distribution model

- Peer-to-peer replication and automatic sharding
  - Any user can connect to any node; all writes are partitioned and replicated automatically



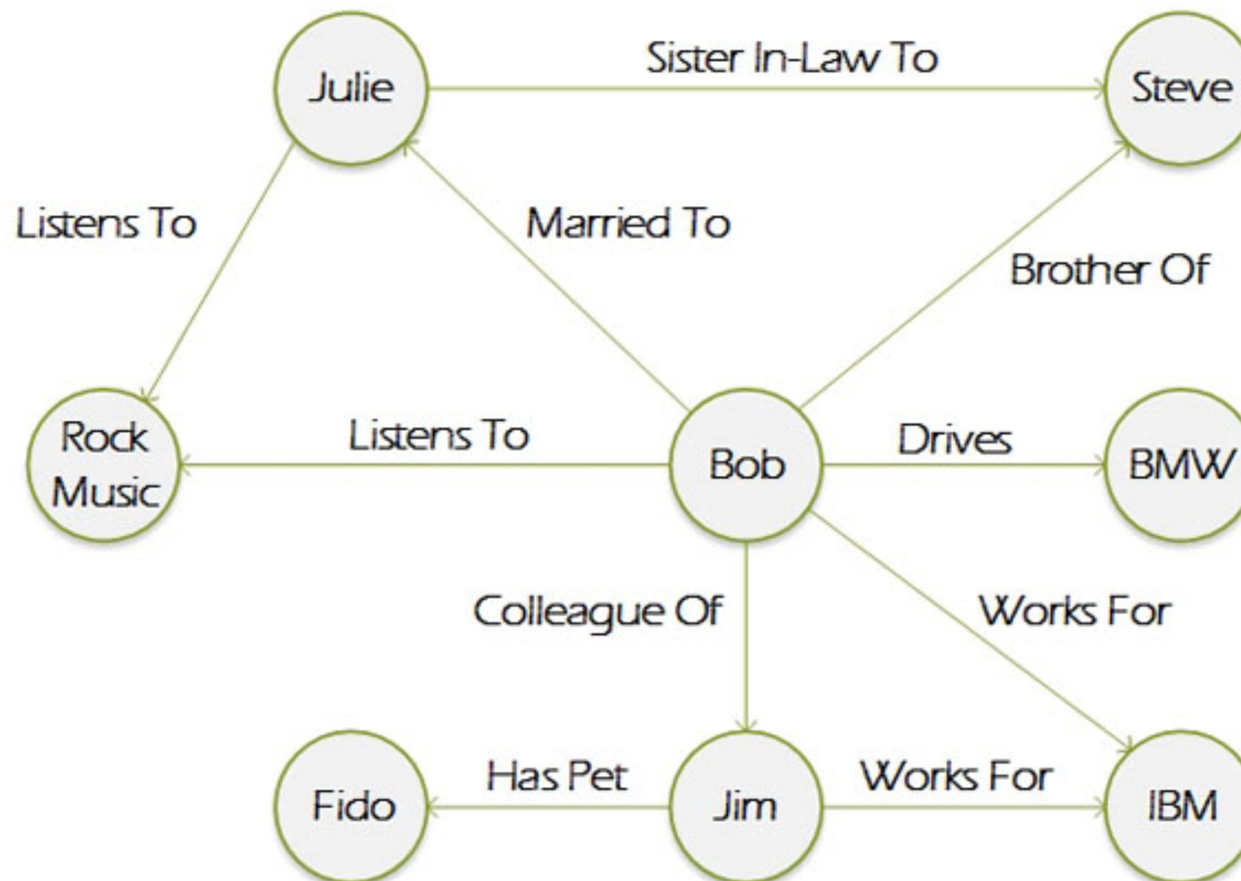
Consistency and scalability

- Tunable consistency on a per-operation basis
  - Can set N, W, R
- Can dynamically add and remove nodes

Graph databases

Graph databases

- A graph contains nodes and relationships; both have properties



Graph databases

- Graph traversal is fast
- Declarative, domain-specific query languages
  - Matching patterns of nodes and relationships in the graph and extracting information
  - e.g., “Find all people that work for Big Co. whose friends listen to Rock Music”
- Example systems
  - Neo4j, Infinite Graph, FlockDB, …

Graph databases

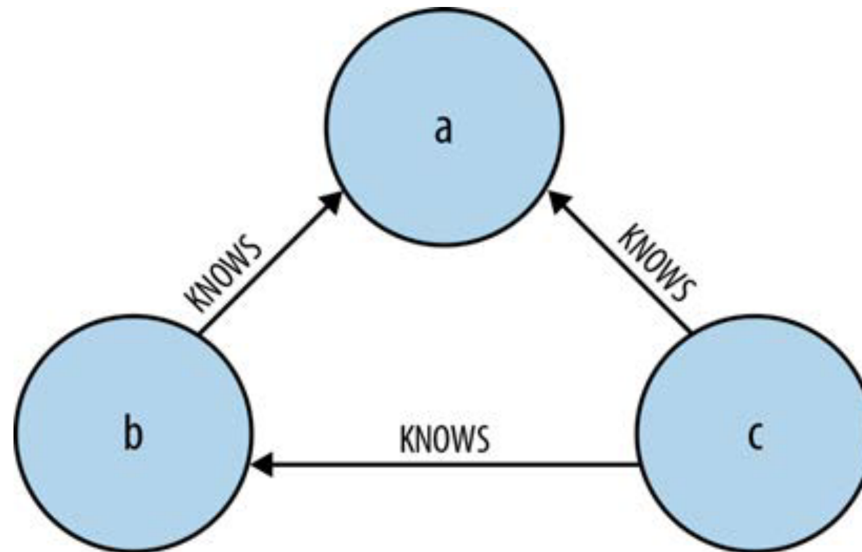
- When to use
  - Connected data, e.g. social networks
  - Routing, dispatch and location-based services
  - Recommendation engines
- When not to use
  - Multiple nodes need to be updated
  - Handling very big graphs (sharding is difficult)

Neo4j

- Open-source graph database
  - First release in 2010
- Written in Java
- ACID transactions
- Clustering for high availability
- Server mode or embedded (same process) mode

Cypher language

- Declarative graph query language
  - ASCII art-like patterns



START a=**node**:user(name='Michael')
MATCH (c)-[:KNOWS]->(b)-[:KNOWS]->(a), (c)-[:KNOWS]->(a)
RETURN b, c

Cypher language

- Supports SQL-like syntax (e.g., WHERE, ORDER BY, COUNT)

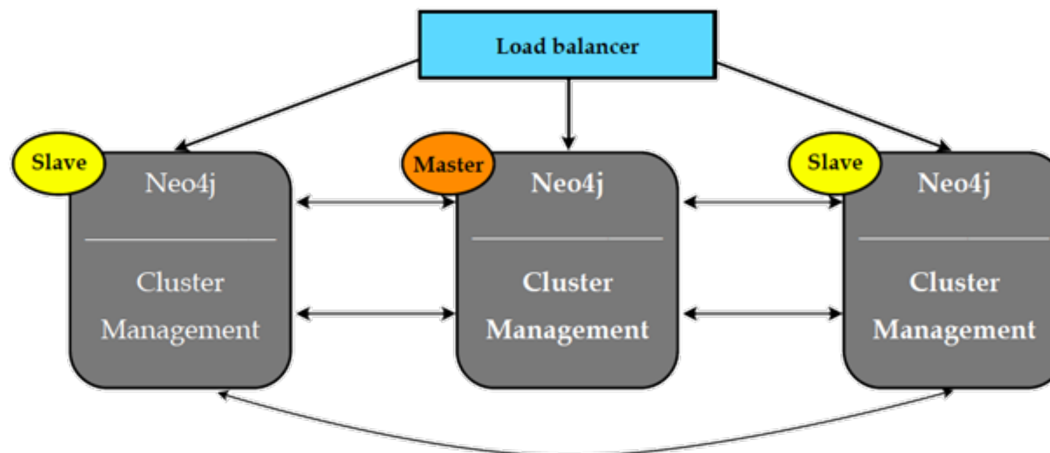
```
START theater=node:venue(name='Theatre Royal'), newcastle=node:city(name='Newcastle'),
        bard=node:author(lastname='Shakespeare')
MATCH (newcastle)<-[:STREET|CITY*1..2]-(theater)
<-[:VENUE]-()-[p:PERFORMANCE_OF]->()-[:PRODUCTION_OF]->
(play)<-[:WROTE_PLAY]-(bard)
RETURN play.title AS play, count(p) AS performance_count
ORDER BY performance_count DESC
```

Indexing

- Properties of nodes or relationships can be indexed
  - Useful for finding starting locations for traversals
  - e.g., theater=node:venue(name='Theatre Royal')

Distribution model

- Master-slave replication
  - High availability; scalability for reads
  - Writes to the master are eventually synchronized to slaves
- No support for sharding

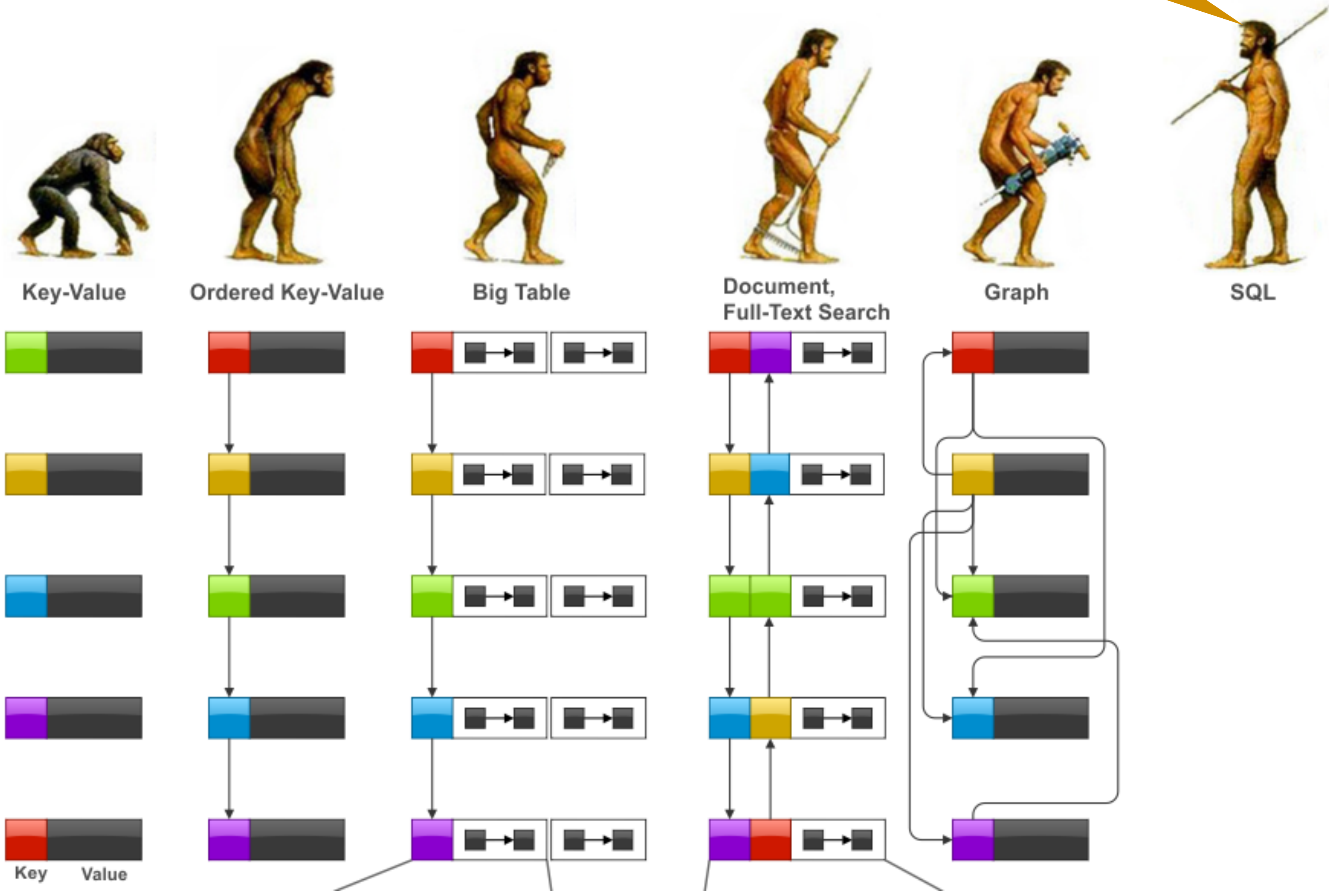


Summary

- The main characteristics of NoSQL technologies are:
  - Horizontal scalability
  - Support for flexible data models
  - Trading some degree of consistency for availability or performance
- Key-value, document, and column-family databases are suitable when most interactions are done with the same data unit. Graph databases are suitable for data with complex relationships.

References

- *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*, Pramod J. Sadalage and Martin Fowler, Pearson Education, 2012.
- *Seven Databases in Seven Weeks*, Eric Redmond and Jim Wilson, Pragmatic Bookshelf, O'Reilly, 2012.


Stop following me!
Key-Value
Ordered Key-Value
Big Table
Document,
Full-Text Search
Graph
SQL
Key
Value