Introduction to NOSQL Databases

Adopted from slides and/or materials by P. Hoekstra,
J. Lu, A. Lakshman, P. Malik, J. Lin, R. Sunderraman,
T. Ivarsson, J. Pokorny, N. Lynch, S. Gilbert, J. Widom,
R. Jin, P. McFadin, C. Nakhli, and R. Ho, Dan Gunter

Background

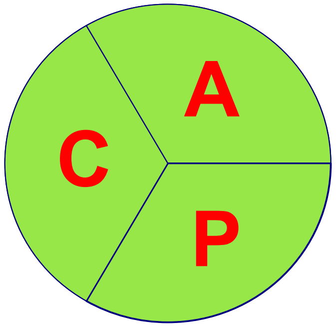
- Relational databases → mainstay of business
- Web-based applications caused spikes
  - explosion of social media sites (Facebook, Twitter) with large data needs
  - rise of cloud-based solutions such as Amazon S3 (simple storage solution)
- Hooking RDBMS to web-based application becomes trouble

Issues with ***scaling up***

- Best way to provide ACID and rich query model is to have the dataset on a single machine
- Limits to ***scaling up*** (or ***vertical scaling*: make a "single" machine more powerful) $\rightarrow$ dataset is just too big!
- ***Scaling out*** (or ***horizontal scaling*: adding more smaller/cheaper servers) is a better choice
- Different approaches for horizontal scaling (multi-node database):
  - Master/Slave
  - Sharding (partitioning)

CAP Theorem

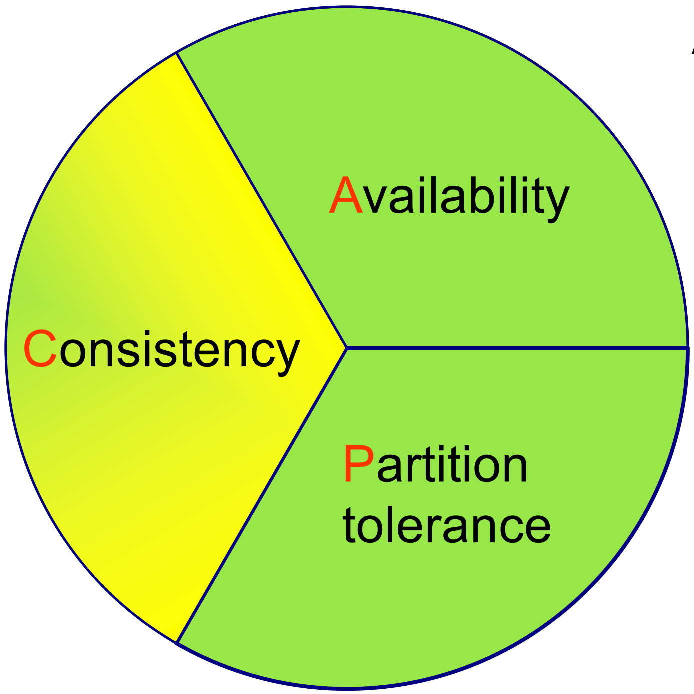
- Suppose three properties of a distributed system (sharing data)
  - **Consistency:**
    - all copies have same value
  - **Availability:**
    - reads and writes always succeed
  - **Partition-tolerance:**
    - system properties (consistency and/or availability) hold even when network failures prevent some machines from communicating with others



CAP Theorem

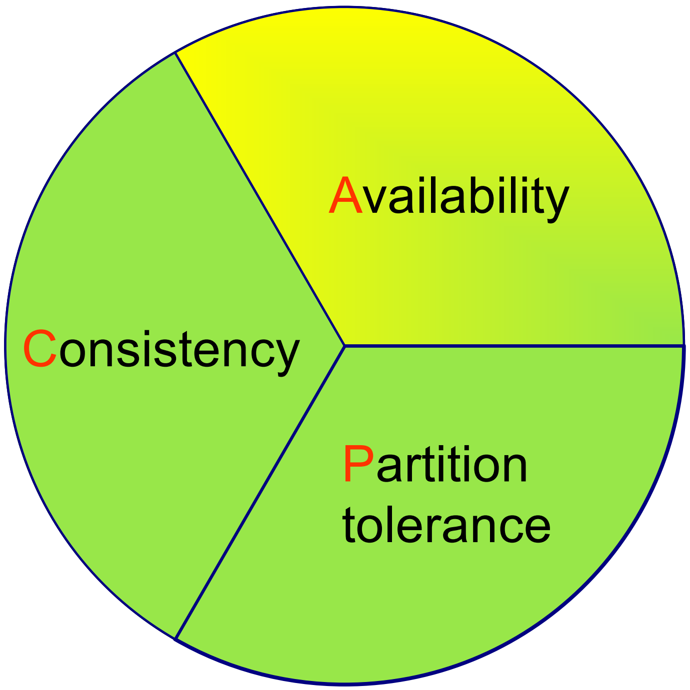
- **Brewer's CAP Theorem:**
  - *For any system sharing data, it is "impossible" to guarantee simultaneously all of these three properties*
  - You can have at most two of these three properties for any shared-data system
- Very large systems will "partition" at some point:
  - That leaves either **C** or **A** to choose from (traditional DBMS prefers **C** over **A** and **P**)
  - In almost all cases, you would choose **A** over **C** (except in specific applications such as order processing)

CAP Theorem



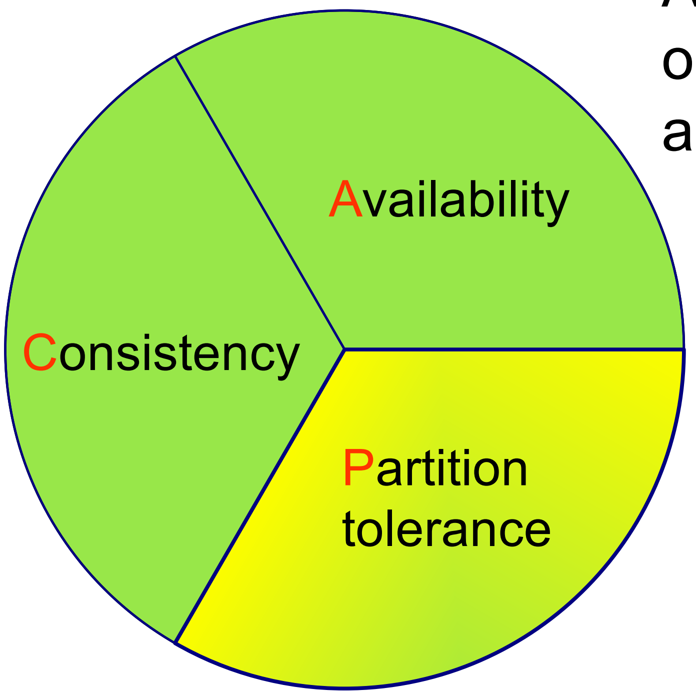
All client always have the same view of the data

CAP Theorem

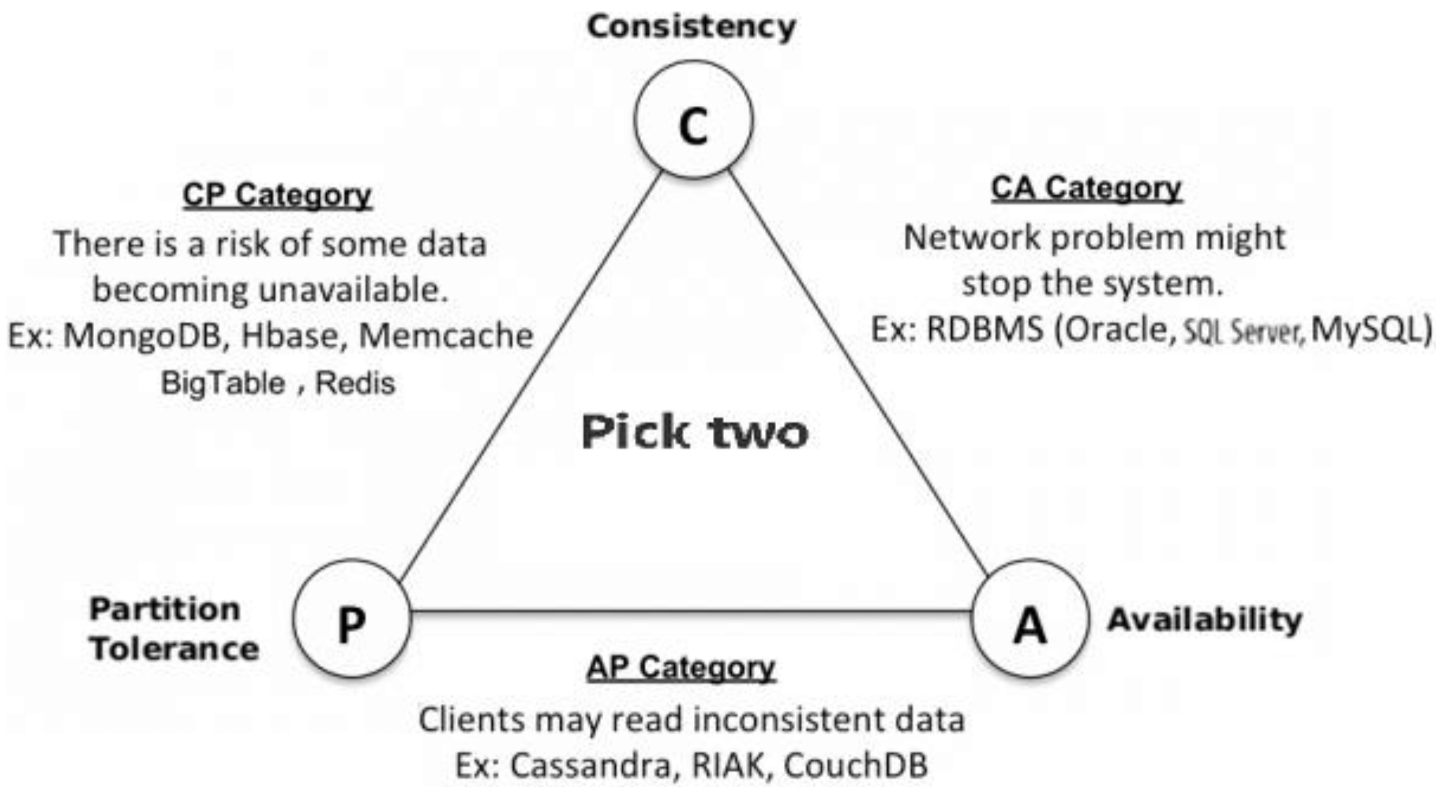


Each client always can read and write.

CAP Theorem



A system can continue to operate in the presence of a network partitions


Consistency
C
CP Category
There is a risk of some data becoming unavailable.
Ex: MongoDB, Hbase, Memcache
BigTable , Redis
CA Category
Network problem might stop the system.
Ex: RDBMS (Oracle, SQL Server, MySQL)
Pick two
Partition Tolerance
P
A
Availability
AP Category
Clients may read inconsistent data
Ex: Cassandra, RIAK, CouchDB

CAP, ACID, and BASE

- RDBMS systems and research focus on ACID: **A**tomicity, **C**onsistency, **I**solation, and **D**urability
  - concurrent operations act as if they are serialized
- Brewer's point is that this is one end of a *spectrum*, one that sacrifices Partition-tolerance and Availability for Consistency
- So, at the other end of the spectrum we have **BASE**: **B**asically **A**vailable **S**oft-state with **E**ventual consistency
  - Stale data may be returned
  - Optimistic locking (e.g., versioned writes)
  - Simpler, faster, easier evolution

CAP Theorem

- A consistency model determines rules for visibility and apparent order of updates
- Example:
  - Row X is replicated on nodes M and N
  - Client A writes row X to node N
  - Some period of time t elapses
  - Client B reads row X from node M
  - **Does client B see the write from client A?**
  - Consistency is a continuum with tradeoffs
  - **For NOSQL, the answer would be: “maybe”**
  - CAP theorem states: *“strong consistency can't be achieved at the same time as availability and partition-tolerance”*

CAP Theorem

- Eventual consistency
  - When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent
- Cloud computing
  - ACID is hard to achieve, moreover, it is not always required, e.g. for blogs, status updates, product listings, etc.

Some intuition about Client-Server Consistency

- *Strong Consistency (ACID-style)*. After an update occurs, everyone sees this value.
- *Weak Consistency*. After an update, system may wait to show some read operations the value.
- *Eventual Consistency (EC).* If there are no updates made to an object, eventually it reaches a consistent state.

EC gives flexibility to system to improve availability. Why?

ACID v. BASE

- Strong Consistency
- Isolation
- Focus on “Commit”
- Nested TXs
- Availability?
- Conservative
- Difficult Evolution

- Weak Consistency
- Availability First
- Best Effort
- Apx Answers
- Aggressive
- Simpler, Faster
- Easier Evolution

CAP Theorem

	All robust distributed systems live here	
Forfeit partition-tolerance	**Forfeit availability**	**Forfeit consistency**
Single-site databases, cluster databases, LDAP	Distributed databases w/pessimistic locking, majority protocols	Coda, web caching, DNS, **Dynamo**

Scaling out RDBMS: Master/Slave

- Master/Slave
  - All writes are written to the master
  - All reads performed against the replicated slave databases
  - Critical reads may be incorrect as writes may not have been propagated down
  - Large datasets can pose problems as master needs to duplicate data to slaves

Scaling out RDBMS: Sharding

- Sharding (Partitioning)
  - Scales well for both reads and writes
  - Not transparent, application needs to be partition-aware
  - Can no longer have relationships/joins across partitions
  - Loss of referential integrity across shards

Other ways to scale out RDBMS

- Multi-Master replication
- INSERT only, not UPDATES/DELETES
- No JOINs, thereby reducing query time
  - This involves de-normalizing data
- In-memory databases

Cattell, SIGMOD Record 2010

Scalable Relational Systems

- Means RDBMS that offer sharding
- **Key difference**:
  - NoSQL difficult or impossible to perform large-scope operations and transactions (to ensure performance),
  - Scalable RDBMS do not **preclude** these operations, but users pay a price only when they need them.
- MySQL Cluster, VoltDB, Clusterix, ScaleDB, Megastore (the new BigTable), GreenPlum
- Many more **NewSQL** systems coming online…
  - Column stores

What is NOSQL?

- The Name:
  - Stands for **N**ot **O**nly **SQL**
  - The term NOSQL was introduced by Carl Strozzi in 1998 to name his file-based database
  - It was again re-introduced by Eric Evans when an event was organized to discuss open source distributed databases
  - Eric states that *"… but the whole point of seeking alternatives is that you need to solve a problem that relational databases are a bad fit for. …"*

The Perfect Storm

- Large datasets, acceptance of alternatives, and dynamically-typed data has come together in a “perfect storm”
- Not a backlash against RDBMS
- SQL is a rich query language that cannot be rivaled by the current list of NOSQL offerings

Why NOSQL?

- Renewed interest originated with *global* internet companies (Google, Amazon, Yahoo!, FaceBook, etc.) that hit limitations of standard RDBMS solutions for one or more of:
  - Extremely high transaction rates
  - Dynamic analysis of huge volumes of data
  - Rapidly evolving and/or semi-structured data
- At the same time, these companies – unlike the financial and health services industries using **M** and friends – did not particularly need "ACID" transactional guarantees
  - Didn't want to run z/OS on mainframes
  - And had to deal with the ugly reality of distributed computing: networks break your $&#!

NoSQL: Overview

- Main objective: implement distributed state
  - Different objects stored on different servers
  - Same object replicated on different servers
- Main idea: give up some of the ACID constraints to improve performance
- Simple interface:
  - Write (=Put): needs to write all replicas
  - Read (=Get): may get only one
- Eventual consistency ← Strong consistency

Cattell, SIGMOD Record 2010

NoSQL

"Not Only SQL" or "Not Relational".

Six key features:

1. Scale horizontally "simple operations"
2. Replicate/distribute data over many servers
3. Simple call level interface (contrast w/ SQL)
4. Weaker concurrency model than ACID
5. Efficient use of distributed indexes and RAM
6. Flexible schema

Cattell, SIGMOD Record 2010

Terminology

- Simple operations = key lookups, read/writes of one record, or a small number of records
- Sharding = horizontal partitioning by some key, and storing records on different servers in order to improve performance. How?
- Horizontal scalability = distribute both data *and* load over many servers
- Vertical scaling = when a dbms uses multiple cores and/or CPUs

What is NOSQL?

- Key features (advantages):
  - non-relational
  - don't require schema
  - data are replicated to multiple nodes (so, identical & fault-tolerant) and can be partitioned:
    - down nodes easily replaced
    - no single point of failure
  - horizontal scalable
  - cheap, easy to implement (open-source)
  - massive write performance
  - fast key-value access

What is NOSQL?

- Disadvantages:
  - Don’t fully support relational features
    - no join, group by, order by operations (except within partitions)
    - no referential integrity constraints across partitions
  - No declarative query language (e.g., SQL) $\rightarrow$ more programming
  - Relaxed ACID (see CAP theorem) $\rightarrow$ fewer guarantees
  - No easy integration with other applications that support SQL

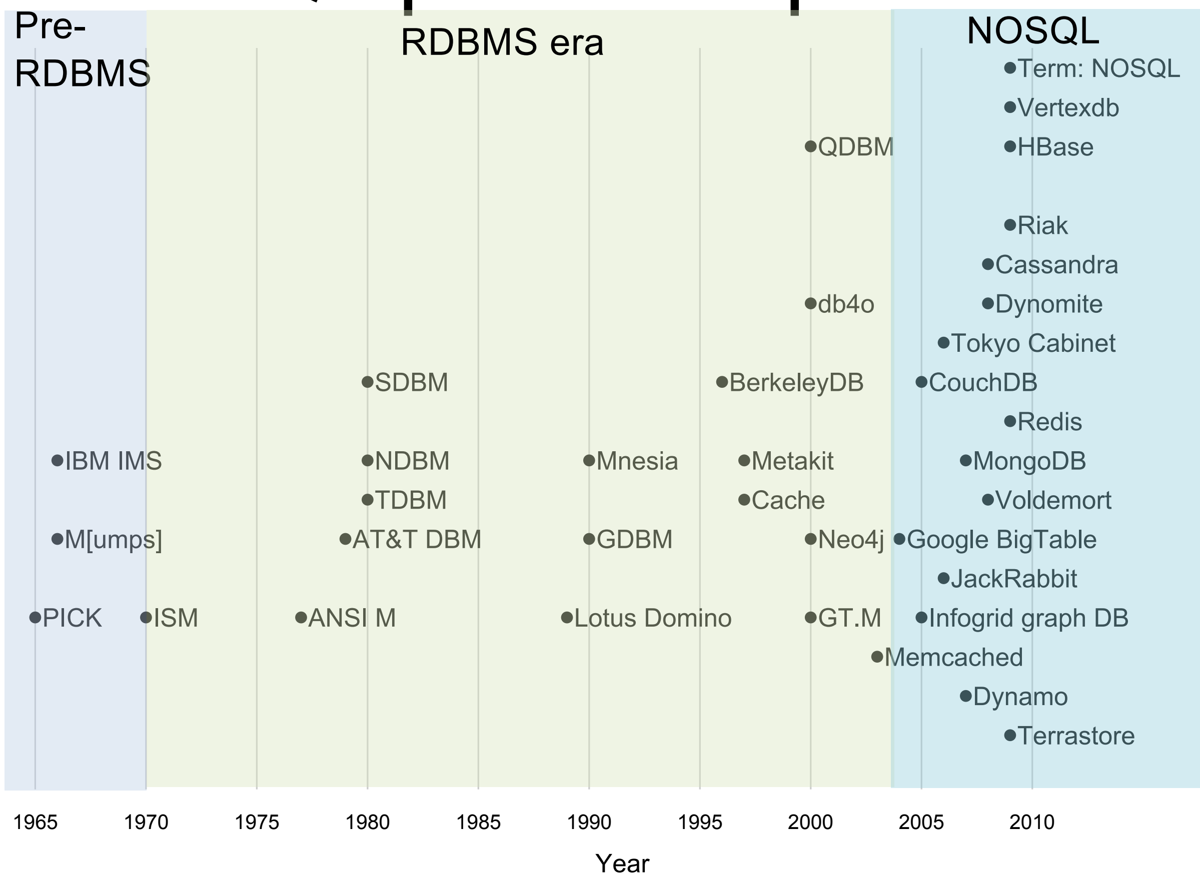
Terminology: NOSQL and “Schemaless”

- First: not terribly important or deep in meaning
- But “NOSQL” has gained currency
  - Original, and best, meaning: **Not Only SQL**
    - Wikipedia credits it to Carlo Strozzi in 1998, re-introduced in 2009 by Eric Evans of Rackspace
    - May use non-SQL, typically simpler, access methods
    - Don’t need to follow all the rules for RDBMS’es
  - Lends itself to “No (use of) SQL”, but this is misleading
- Also referred to as “schemaless” databases
  - Implies dynamic schema evolution

Perspectives

- RDBMS:
  - Domain based model
  - “What answers do I have”
- NoSQL
  - Query based model
  - “What questions do I have”
- Where do I do the joins?

NOSQL past and present



NoSQL Database Types

Key/Value Store
- Memcached
- Redis
- Tokyo Cabinet
- Dynamo
- Dynomite
- Riak
- Project Voldemort

Columnar *or* Extensible record
- Google BigTable
- HBase
- Cassandra
- HyperTable

Document Store
- CouchDB
- MongoDB
- SimpleDB
- Lotus Domino

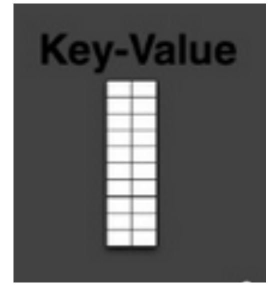
Graph DB
- Neo4j
- FlockDB
- InfiniteGraph

Cattell, SIGMOD Record 2010

Data Model

- Tuple = row in a relational db
- Document = nested values, extensible records (think XML or JSON)
- Extensible record = families of attributes have a schema, but new attributes may be added
- Object = like in a programming language, but without methods

Key-value



- Focus on scaling to huge amounts of data
- Designed to handle massive load
- Based on Amazon’s dynamo paper
- Data model: (global) collection of Key-value pairs
- *Dynamo ring partitioning* and *replication*
- Example: (DynamoDB)
  - *items* having one or more attributes (name, value)
  - An *attribute* can be single-valued or multi-valued like set.
  - items are combined into a *table*

Key-value

- Basic API access:
  - get(key): extract the value given a key
  - put(key, value): create or update the value given its key
  - delete(key): remove the key and its associated value
  - execute(key, operation, parameters): invoke an operation to the value (given its key) which is a special data structure (e.g. List, Set, Map .... etc)

Key-value

Pros:

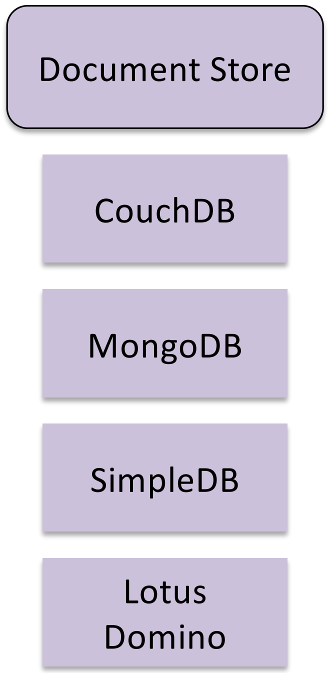
- very fast
- very scalable (horizontally distributed to nodes based on key)
- simple data model
- eventual consistency
- fault-tolerance

Cons:

- Can’t model more complex data structure such as objects

Document-based

- Can model more complex objects
- Inspired by Lotus Notes
- Data model: collection of documents
- Document: JSON (**J**ava**S**cript **O**bject **N**otation is a data model, key-value pairs, which supports objects, records, structs, lists, array, maps, dates, Boolean with **nesting**), XML, other semi-structured formats.



- Store objects (not really documents)
  - think: nested maps
- Varying degrees of consistency, but not ACID
- Allow queries on data contents (M/R or other)
- May provide atomic read-and-set operations

Document-based

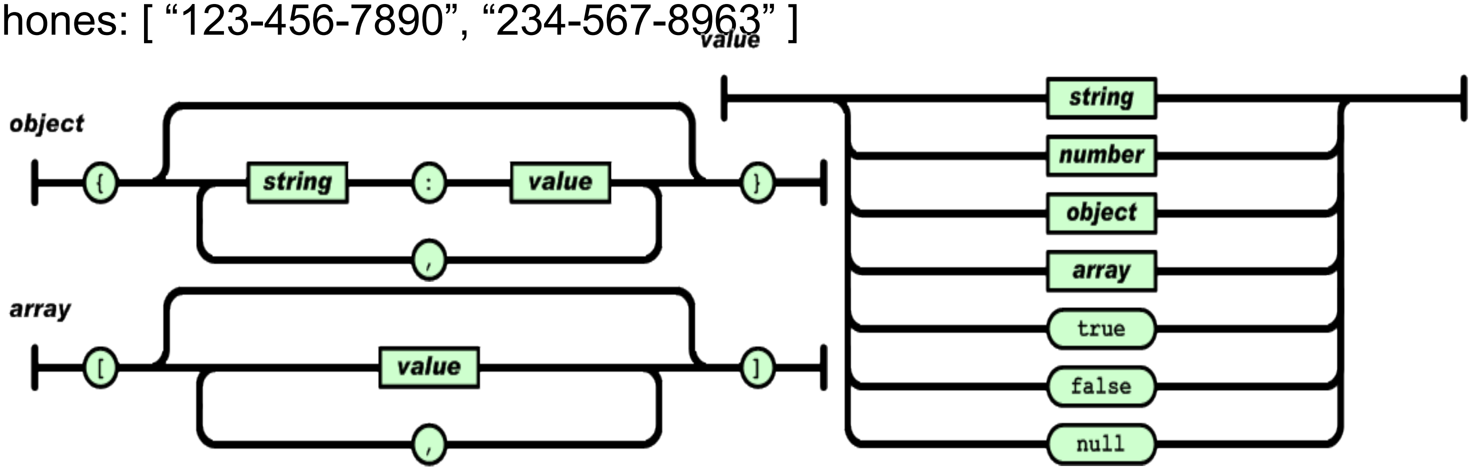
- Example: (MongoDB) document
  - {Name:"Jaroslav",

    Address:"Malostranske nám. 25, 118 00 Praha 1”,

    Grandchildren: {Claire: "7", Barbara: "6", "Magda: "3", "Kirsten: "1", "Otis: "3", Richard: "1“}

    Phones: [“123-456-7890”, “234-567-8963”]

    }



Capturing N to M is tricky…..

Queries

```
db.inventory.insertMany([
    { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
    { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "A" },
    { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
    { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
    { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" }
]);
```

Mongo Queries

- db.inventory.find({})
  - **SELECT** * **FROM** inventor
- db.inventory.find({ status: "D" })
  - **SELECT** * **FROM** inventory **WHERE** status = "D"
- db.inventory.find({ status: { $in: ["A", "D"] } })
  - **SELECT** * **FROM** inventory **WHERE** status **in** ("A", "D")

Mongo Queries on Nested Documents

- db.inventory.find({ size: { h: 14, w: 21, uom: "cm" } })
- db.inventory.find({ "size.uom": "in" })
- db.inventory.find({ "size.h": { $lt: 15 } })

Document-based

Name	Producer	Data model	Querying
MongoDB	10gen	object-structured documents stored in collections; each object has a primary key called ObjectId	manipulations with objects in collections (find object or objects via simple selections and logical expressions, delete, update,)
Couchbase	Couchbase[1]	document as a list of named (structured) items (JSON document)	by key and key range, views via Javascript and MapReduce

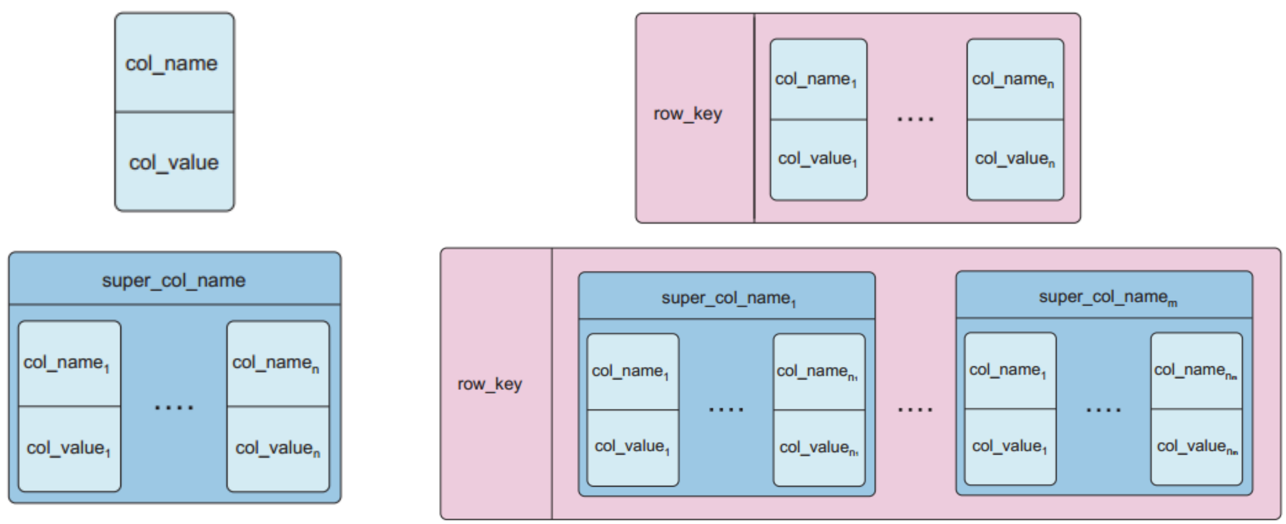
Cattell, SIGMOD Record 2010

Document Stores Scalability

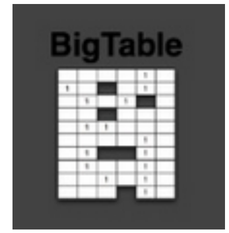
- Replication (e.g. SimpleDB, CounchDB – means entire db is replicated),
- Sharding (MongoDB);
- Both

Column-based

- Based on Google's BigTable paper
- Like column oriented relational databases (store data in column order) but with a twist
- Tables similarly to RDBMS, but handle semi-structured
- Data model:
  - Collection of Column Families
  - Column family = (key, value) where value = set of **related** columns (standard, super)
  - indexed by *row key*, *column key* and *timestamp*



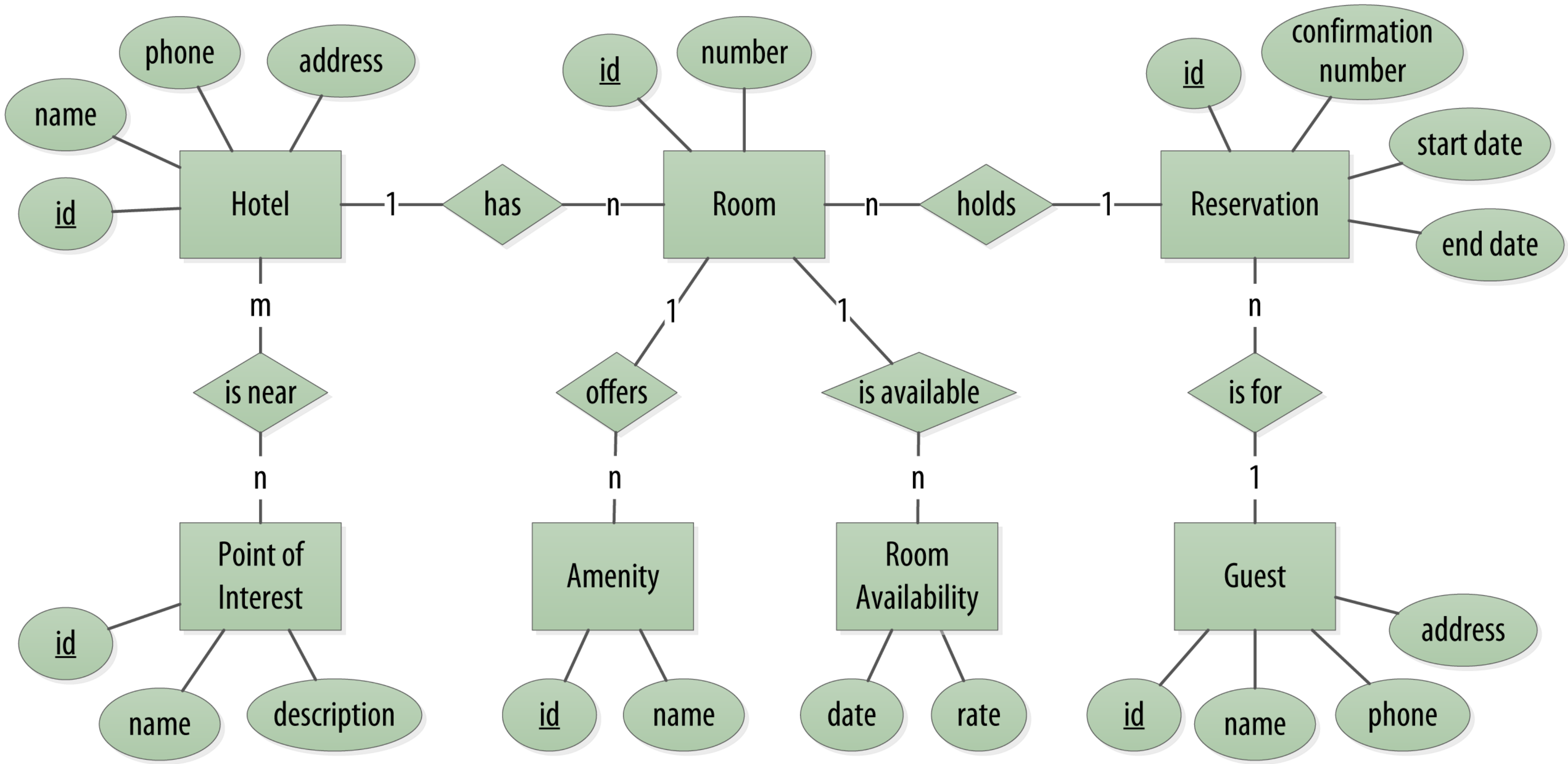
Column-based



- One column family can have variable numbers of columns
- Cells within a column family are sorted “physically”
- Very sparse, most cells have null values
- **Comparison:** RDBMS vs column-based NOSQL
  - Query on multiple tables
    - **RDBMS:** must fetch data from several places on disk and glue together
    - **Column-based NOSQL:** only fetch column families of those columns that are required by a query (all columns in a column family are stored together on the disk, so multiple rows can be retrieved in one read operation → data locality)

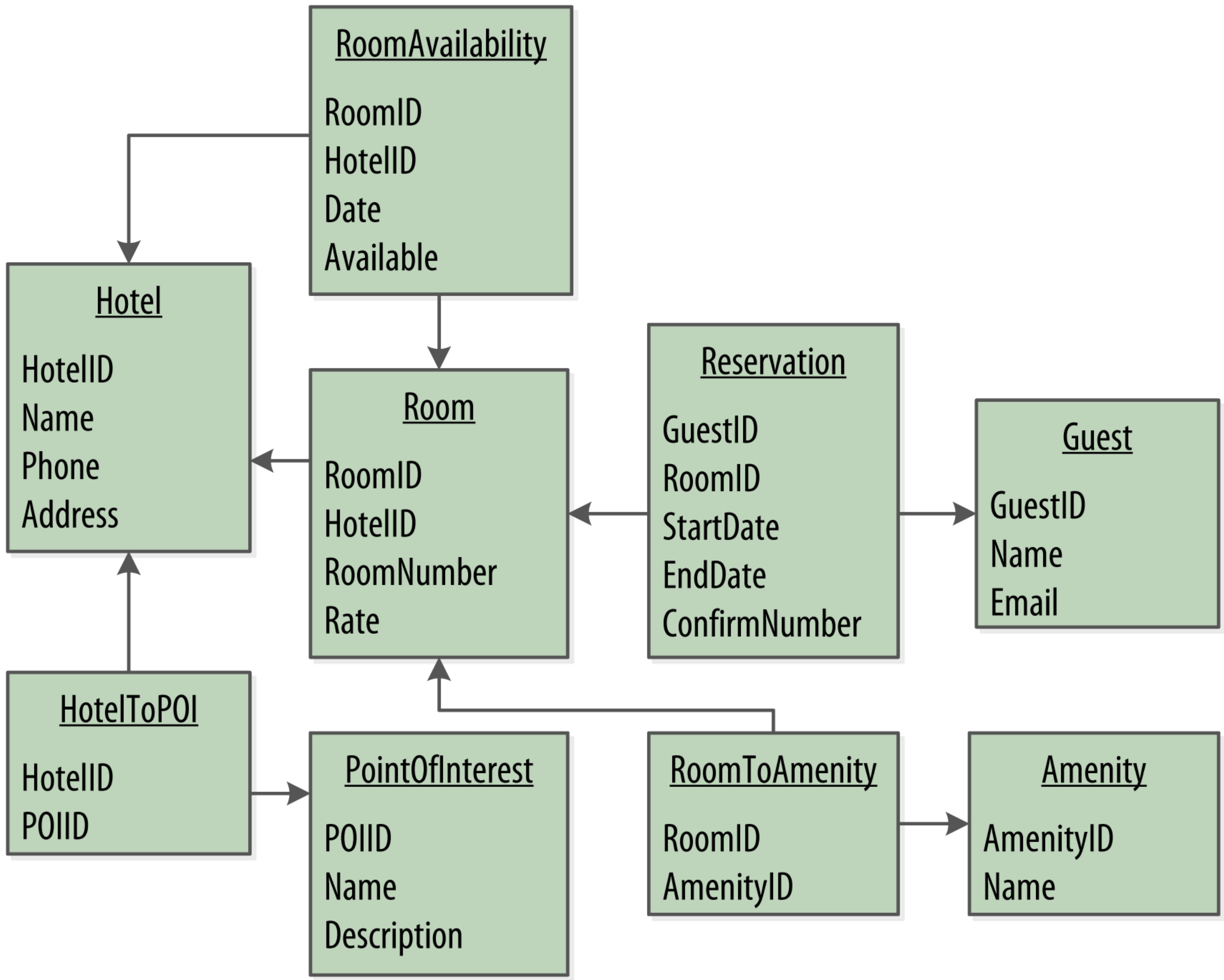
Cassandra

- No joins
- No referential integrity
- Denormalization
- Query-first design

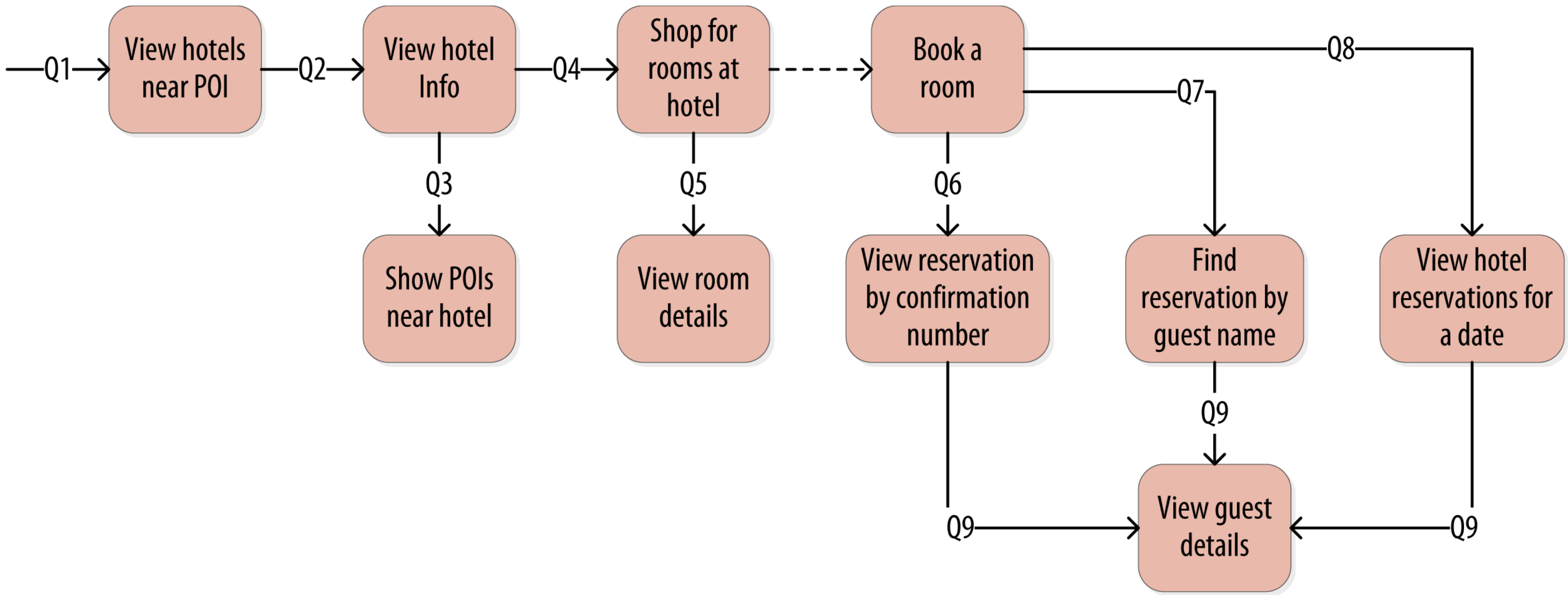

phone
address
name
id
Hotel
1
has
n
Room
id
number
n
holds
1
Reservation
id
confirmation number
start date
end date
m
is near
n
Point of Interest
id
name
description
1
offers
n
Amenity
id
name
1
is available
n
Room Availability
date
rate
n
is for
1
Guest
address
id
name
phone

Queries

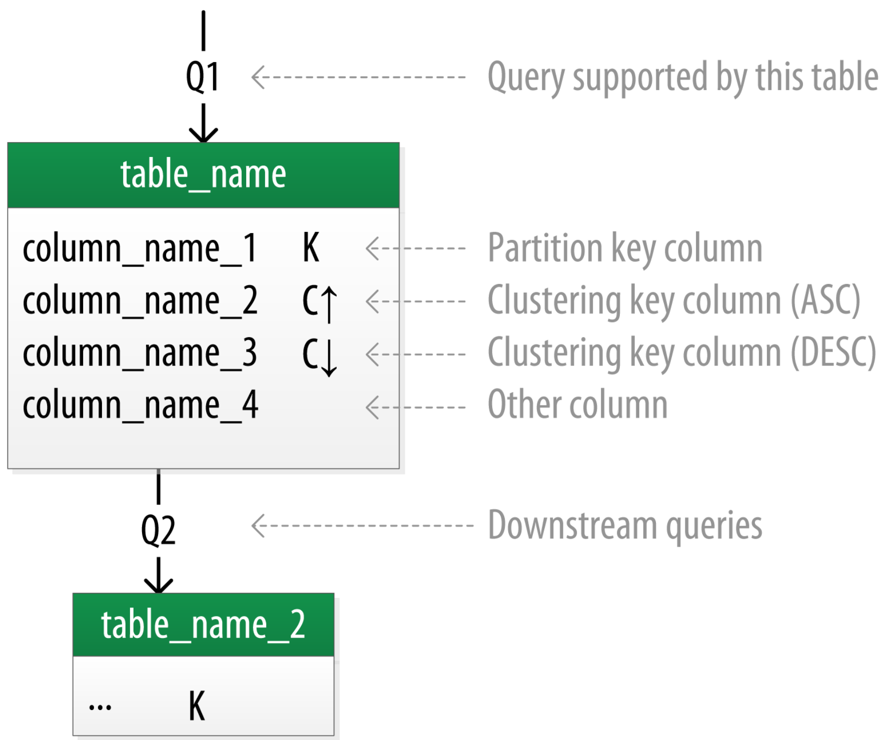
- Q1. Find hotels near a given point of interest.
- Q2. Find information about a given hotel, such as its name and location.
- Q3. Find points of interest near a given hotel.
- Q4. Find an available room in a given date range.
- Q5. Find the rate and amenities for a room.
- Q6. Lookup a reservation by confirmation number.
- Q7. Lookup a reservation by hotel, date, and guest name.
- Q8. Lookup all reservations by guest name.
- Q9. View guest details.


RoomAvailability
RoomID
HotelID
Date
Available
Hotel
HotelID
Name
Phone
Address
Room
RoomID
HotelID
RoomNumber
Rate
Reservation
GuestID
RoomID
StartDate
EndDate
ConfirmNumber
Guest
GuestID
Name
Email
HotelToPOI
HotelID
POIID
PointOfInterest
POIID
Name
Description
RoomToAmenity
RoomID
AmenityID
Amenity
AmenityID
Name

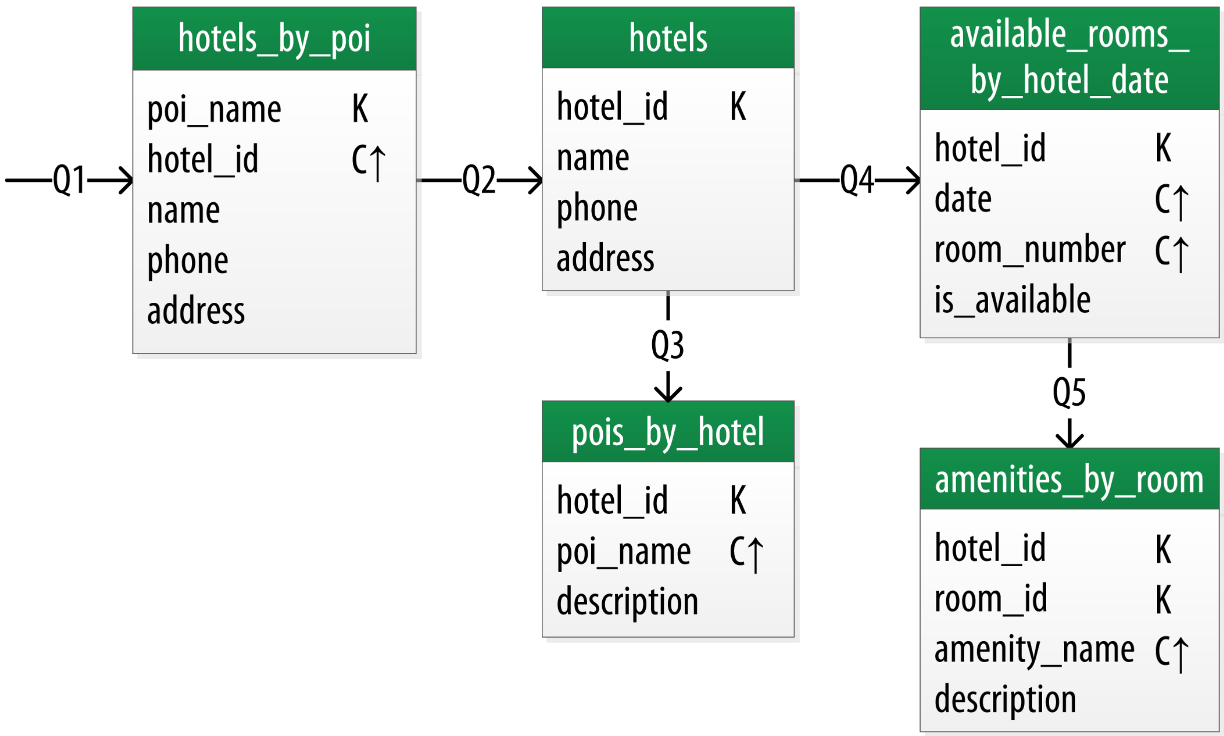
Application Queries



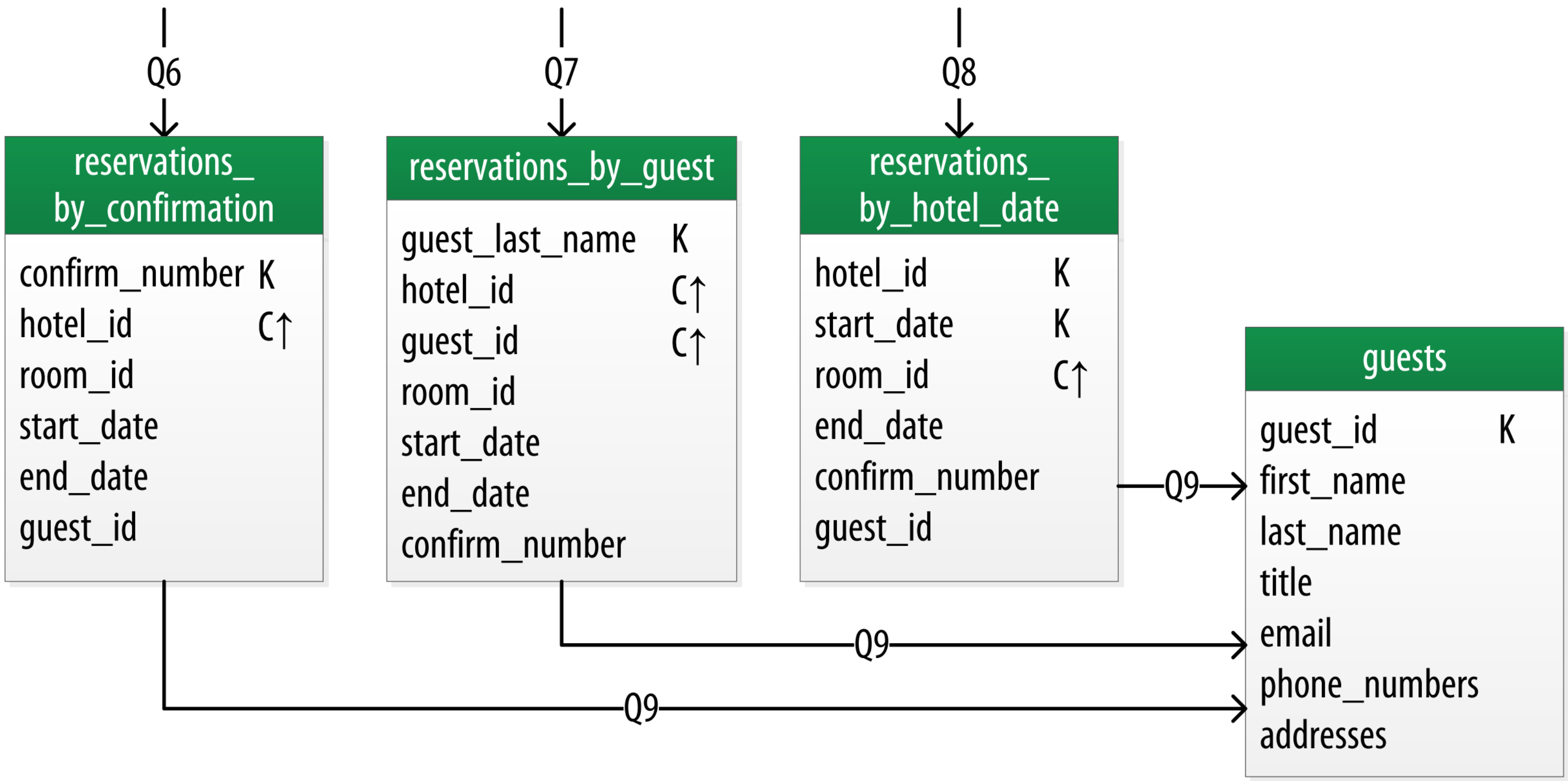
Logical Modeling



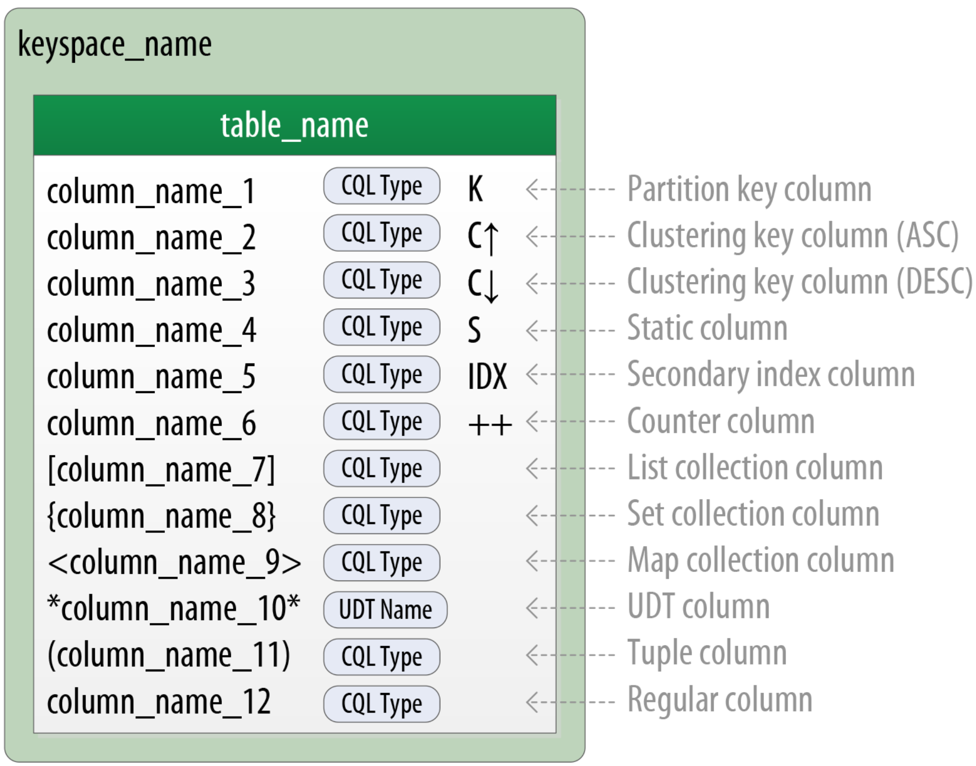
Hotel Logical Model



Reservation Logical Model



Physical Data Modeling



Hotel Physical Model

hotel keyspace

hotels

hotel_id	text	K
name	text	
phone	text	
address	address	

hotels_by_poi

poi_name	text	K
hotel_id	text	C↑
name	text	
phone	text	
address	address	

address

street	text
city	text
state_or_province	text
postal_code	text
country	text

available_rooms_by_hotel_date

hotel_id	text	K
date	date	C↑
room_number	smallint	C↑
is_available	boolean	

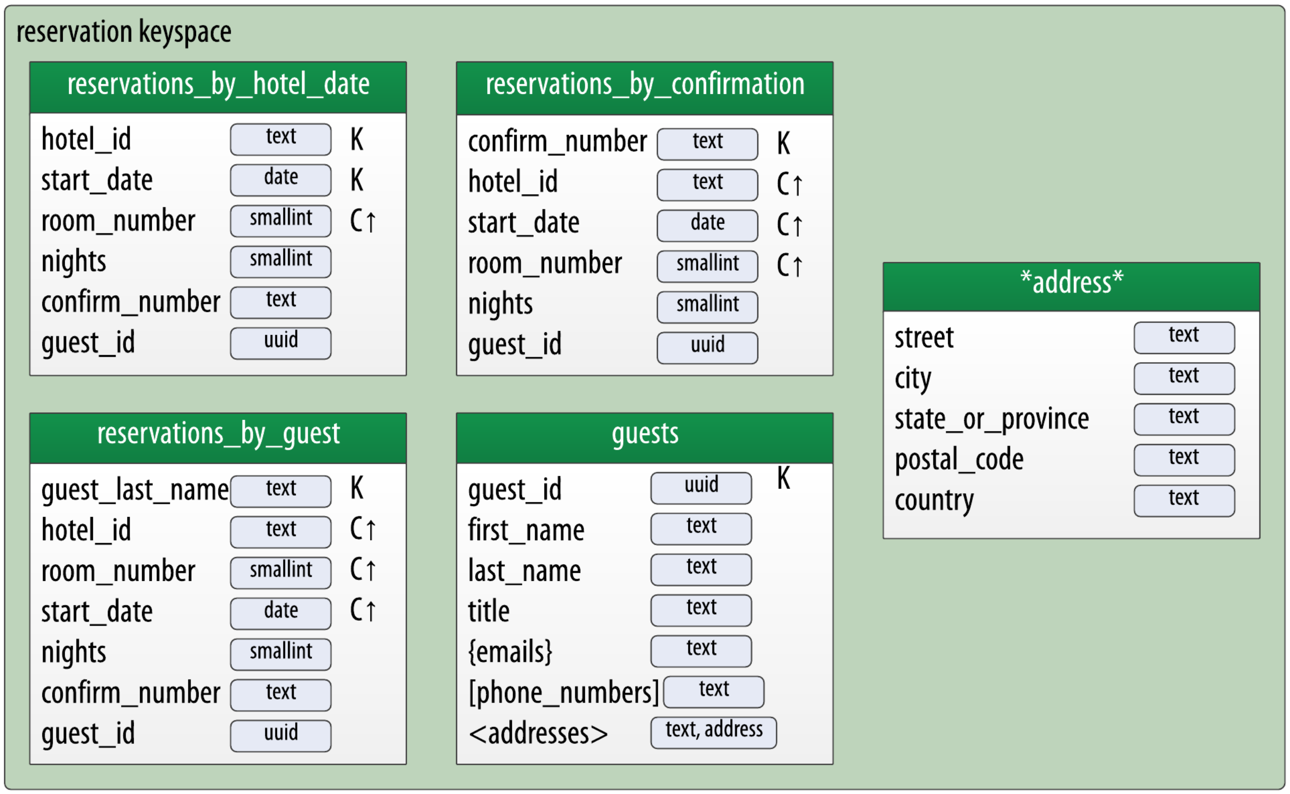
pois_by_hotel

hotel_id	text	K
poi_name	text	C↑
description	text	

amenities_by_room

hotel_id	text	K
room_number	smallint	K
amenity_name	text	C↑
description	text	

Reservation Physical Model



Graph-based

- Focus on modeling the structure of data (*interconnectivity*)
- Scales to the complexity of data
- Inspired by mathematical Graph Theory (G=(E,V))
- Data model:
  - (Property Graph) nodes and edges
    - Nodes may have properties (including ID)
    - Edges may have labels or roles
  - Key-value pairs on both
- Interfaces and query languages vary
- *Single-step* vs *path expressions* vs *full recursion*
- Example:
  - Neo4j, FlockDB, Pregel, InfoGrid …