CMPT 321
FALL 2017

Alternative data models

Lecture 08.01

By Marina Barsky

Relational model has many benefits

- Logical data independence: views
- Ad hoc queries
- Mature technologies:
  - Disk algorithms
  - ✓Indexes
  - Query optimizer
  - ✓Concurrent transactions
  - Write-ahead logging

Semantics – the study of meaning

We like mushrooms

Mushrooms scare Ann

- The same word - different semantics
- People deduce meaning implicitly: from the rules of the language plus context
- Computer program needs explicit sematics

Case study.
Restaurant search

Data about restaurants:

- prices
- location
- cuisine
- hours

Single-table model: spreadsheets

Restaurant	Address	Cuisine	Price	Open
Deli Llama	Peachtree Rd	Deli	$	Mon, Tue, Wed, Thu, Fri
Peking Inn	Lake St	Chinese	$$$	Thu, Fri, Sat
Thai Tanic	Branch Dr	Thai	$$	Thu, Fri, Sat, Sun
Lord of the Fries	Flower Ave	Fast Food	$$	Tue, Wed, Thu, Fri, Sat, Sun
Wok This Way	Second St	Chinese	$	Mon, Tue, Wed, Thu, Fri, Sat, Sun
Award Wieners	Dorfold Mews	Fast Food	$	Mon, Tue, Wed, Thu, Fri, Sat

How do we know the meaning of word ‘Chinese’?

Semantics of a single table

- The row and column explains what the value means to a person reading the data
- The fact that Chinese is in the row Peking Inn and in the column Cuisine tells us that “Peking Inn serves Chinese food.”
- You know this because you understand what restaurants and cuisines are and because you’ve previously learned how to read a table

Limitations of a single table model

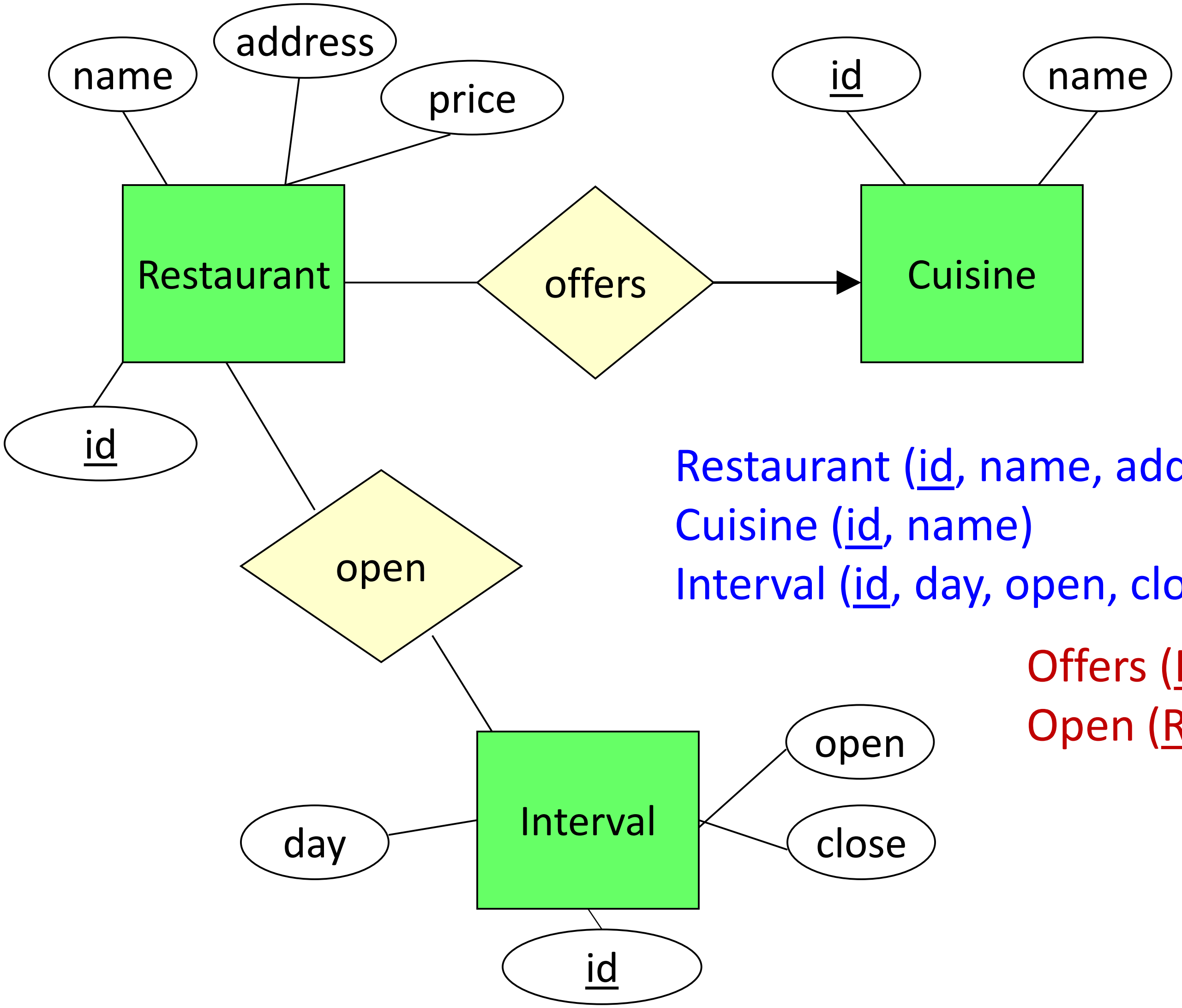
- Multi-valued columns are not searchable:

  find the restaurants that will be open late on Friday night?

- Connecting more tables referencing the same data: our friends’ reviews of the restaurants

  no easy way to search across both documents to find restaurants near our homes that our friends recommend

Relational model



Restaurant (<u>id</u>, name, address, price)
Cuisine (<u>id</u>, name)
Interval (<u>id</u>, day, open, close)

Offers (<u>RestaurantID</u>, <u>cuisineID</u>)
Open (<u>RestaurantID</u>, <u>IntervalID</u>)

Relational model: tables

Restaurant			
id	**Name**	**Address**	**Price**
1	Deli Llama	Peachtree Rd	$
2	Peking Inn	Lake St	$$$

RestaurantCuisine	
RestID	**CuisineID**
1	1
2	2

Cuisine	
id	**Name**
1	Deli
2	Chinese
3	Thai
4	Fast food

Intervals			
id	**Day**	**Open**	**Close**
1	Mon	11	16
2	Tue	11	16
3	Wed	11	16
4	Thu	11	19
5	Fri	11	20
6	Thu	5	22
7	Fri	5	23
8	Sat	5	23

Hours	
RestID	**IntervalID**
1	1
1	2
1	3
1	4
1	5
2	6
2	7
2	8

Benefits: no redundancy

- Ad hoc queries: Find all the restaurants that will be open at 10 p.m. on a Friday

```
SELECT R.Name, C.Name, I.Open, I.Close
FROM   Restaurant R, Cuisine C, Intervals I,
       RestaurantCuisine RC, Hours H
WHERE R.id = RC.RestaurantID AND RC.CuisineID=C.ID
AND R.id=Hours.RestaurantID AND I.id = H.intervalID
AND I.Day="Fri"
AND I.Open<22
AND I.Close>22
```

Relational model: tables

Restaurant

id	**Name**	**Address**	**Price**
1	Deli Llama	Peachtree Rd	$
2	Peking Inn	Lake St	$$$

RestaurantCuisine

RestID	**CuisineID**
1	1
2	2

Cuisine

id	**Name**
1	Deli
2	Chinese
3	Thai
4	Fast food

Intervals

id	**Day**	**Open**	**Close**
1	Mon	11	16
2	Tue	11	16
3	Wed	11	16
4	Thu	11	19
5	Fri	11	20
6	Thu	5	22
7	Fri	5	23
8	Sat	5	23

Hours

RestID	**IntervalID**
1	1
1	2
1	3
1	4
1	5
2	6
2	7
2	8

How do we know the meaning of tuple (1,1) in *RestaurantCuisine*?

Semantics of relational model

- The meaning of each value is described by the schema
- Each datum is labeled with what it means by the table in which it appears and by the column

- We convey this semantics to the computer program: we do not need to define what the restaurant is, but we can still get a list of restaurants with given properties

Extending scope of our search web app

- Our restaurant search app is up and running
- We receive a new data to handle: *Bars*

Bar			
Name	Address	**DJ**	**Specialty drink**
The bitter end	14th avenue	No	Beer
Peking Inn	Lake St	No	Scorpion Bowl
Hammer Time	Wildcat Dr	Yes	Hennessey

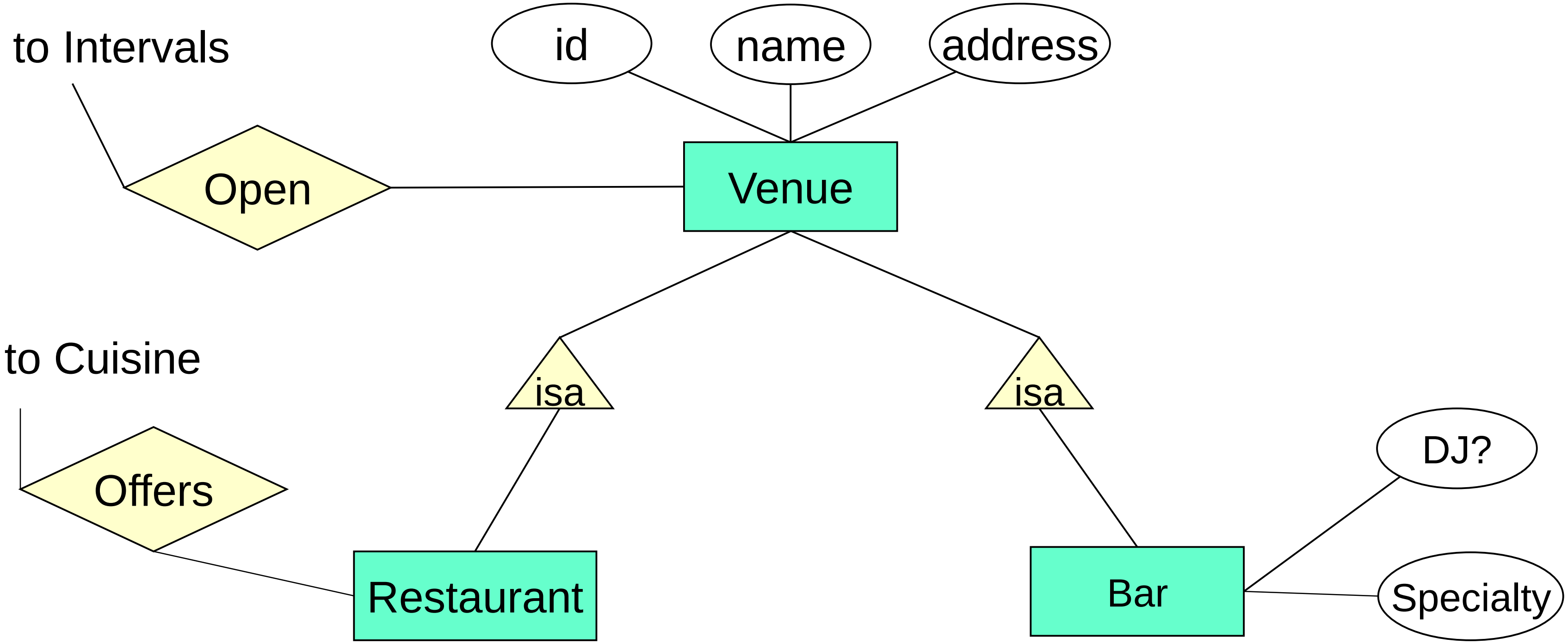
Integrating new data with existing model

- We cannot completely detach bars from restaurants and store them in a separate table:
    - Many restaurants serve as bars later in the evening
    - Bars and restaurants have *common properties*
    - Someone might want to query across both tables

Bar			
Name	Address	**DJ**	**Specialty drink**
The bitter end	14th avenue	No	Beer
Peking Inn	Lake St	No	Scorpion Bowl
Hammer Time	Wildcat Dr	Yes	Hennessey

This cannot be a separate table!

Subclasses?



- Venue (<u>id</u>, name, address)
- RestaurantCuisine (<u>id</u>, <u>cuisineID</u>)
- Bar (<u>id</u>, DJ, specialty)

Constantly evolving schema

- Relational databases:
    - Well-defined data models
    - Know upfront that schema will be stable
    - Typical usage pattern

❑Product catalogs
❑Contact lists
❑Payroll systems

- Data integration across the Web:
    - Rapidly changing types of data:
    - Cannot predict how data will be evolving
    - Do not know how data will be usedused

❑Venues += live music hall
❑Venues+= rental space for events

Changing schema each time is expensive!

- Schema migration:
    - Load data from old tables into new tables
    - Update all triggers, functions and procedures
    - Update all queries and views
    - Update web site code

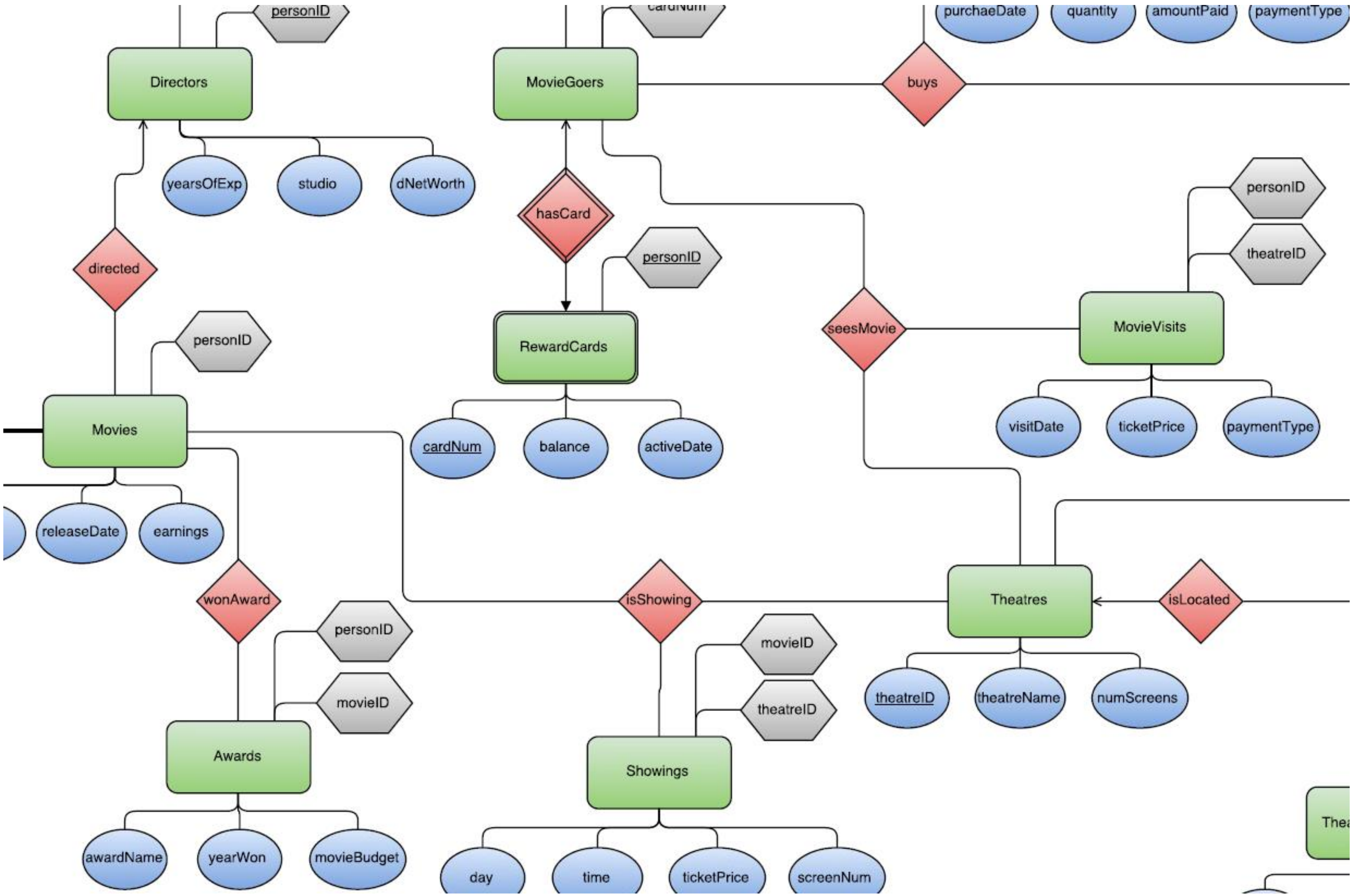
Techniques for schema migration:
☐ ORM (Hibernate)
☐ Stored procedures
☐ …

Complexities and bugs …
Downtime…

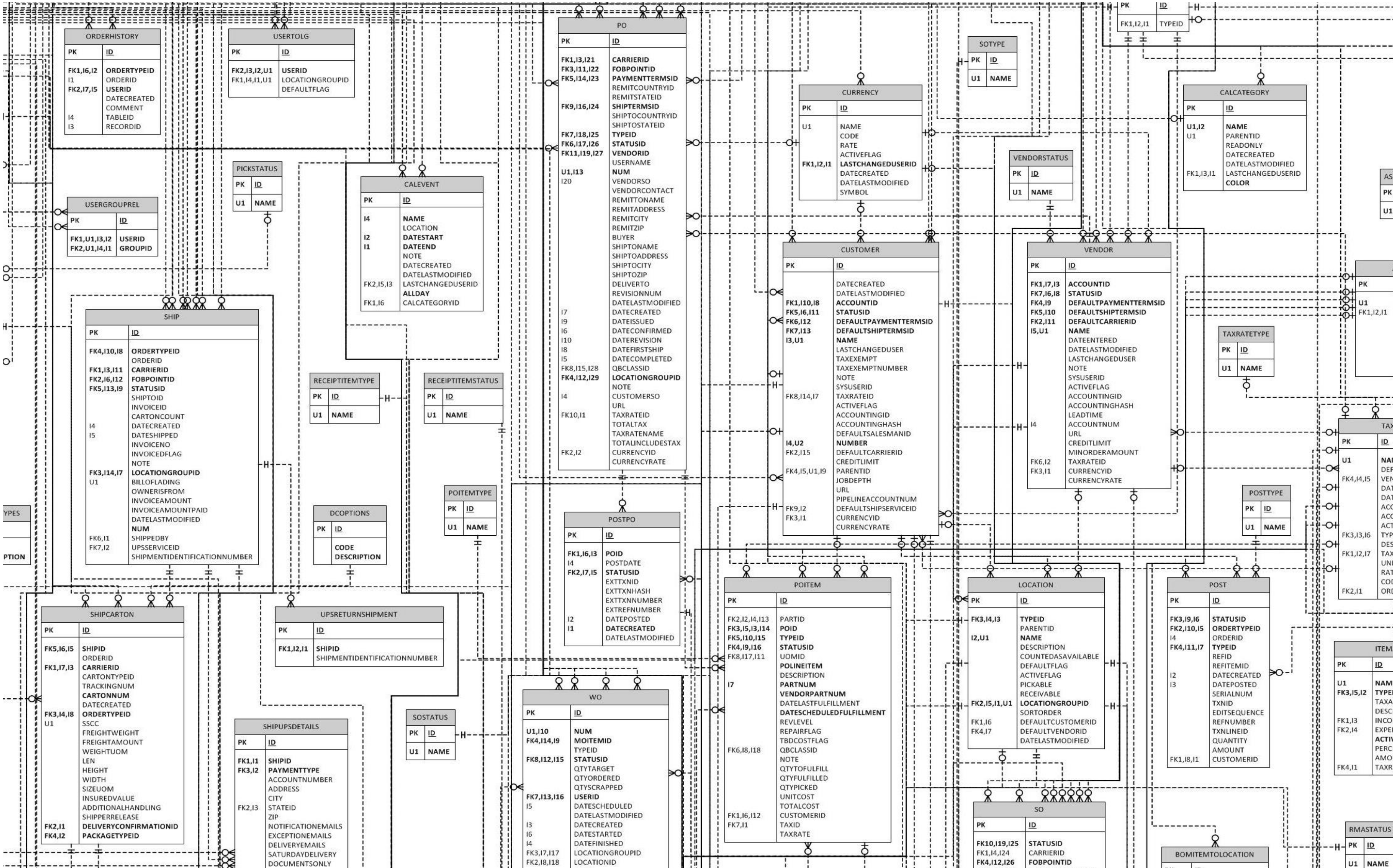
Semantics: very complex schemas

- Incredibly complicated schemas which include different data types
- Hundreds or thousands of inter-connected entities
- Understanding meaning of data is hard -> impossible

Movies and movie goers E/R

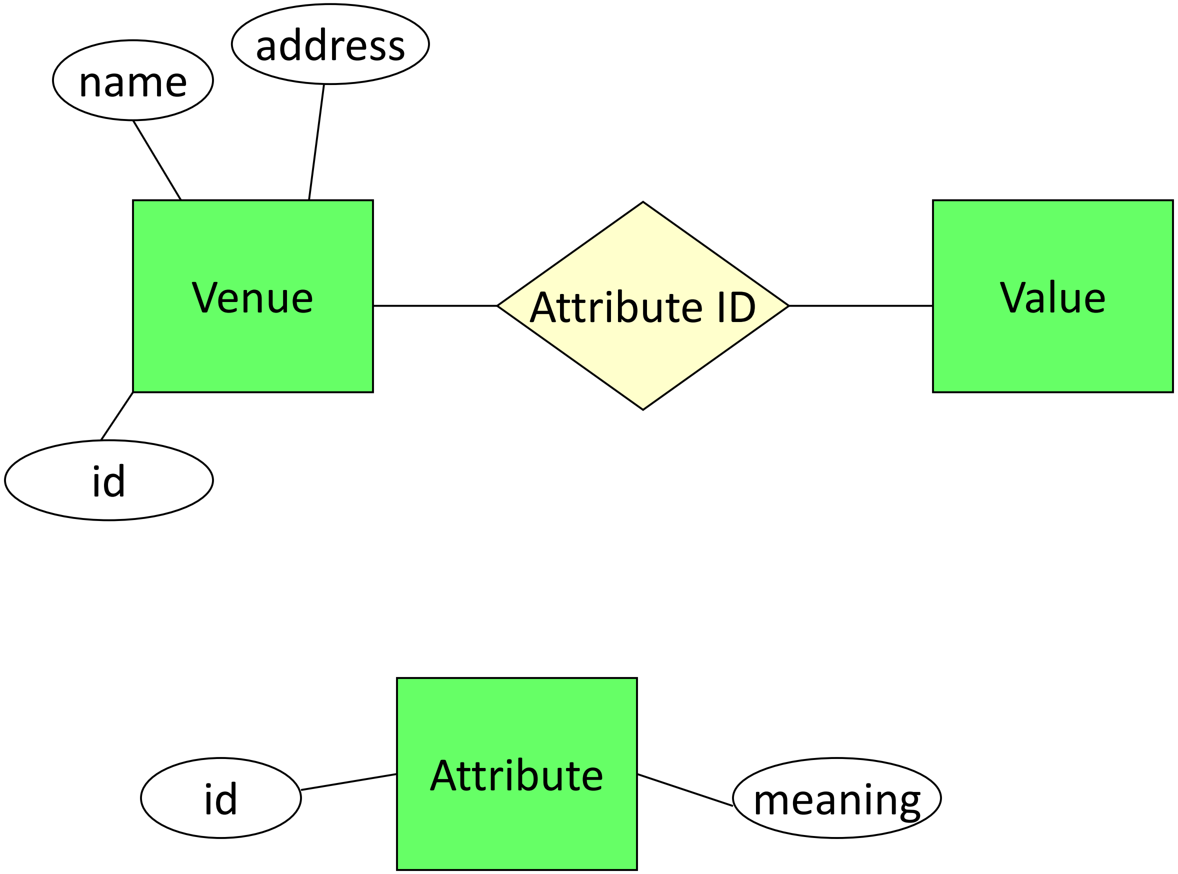


Hospital E/R (left upper corner)



Can we design more flexible data model?

Making it extendable from the beginning



3 and only 3?

Venue		
id	Name	Address
1	Deli Llama	Peachtree Rd
2	Peking Inn	Lake St
3	Thai Tanic	Branch Dr

Attributes	
id	Meaning
1	Cuisine
2	Price
3	Specialty
4	DJ

Properties		
VenueID	Attribute ID	Value
1	1	Deli
1	2	$
2	1	Chinese
2	2	$$$
2	3	Scorpion Bowl
2	4	No

Test: Adding concert venues

Venue		
id	Name	Address
1	Deli Llama	Peachtree Rd
2	Peking Inn	Lake St
3	Thai Tanic	Branch Dr

Attributes	
id	Meaning
1	Cuisine
2	Price
3	Specialty
4	DJ
5	Live Music
6	Music Genre

Properties		
VenueID	Attribute ID	Value
1	1	Deli
1	2	$
2	1	Chinese
2	2	$$$
2	3	Scorpion Bowl
2	4	No
3	5	Yes
3	6	Jazz

2 tables?

Attributes	
id	Meaning
1	Cuisine
2	Price
3	Specialty
4	DJ
5	Live Music
6	Music Genre
7	Name
8	Address

Properties		
VenueID	Attribute ID	Value
1	1	Deli
1	2	$
2	1	Chinese
2	2	$$$
2	3	Scorpion Bowl
2	4	No
1	7	Deli Llama
2	7	Peking Inn
1	8	Peachtree Road
2	8	Lake ST

Joining everything into a single table

Venues		
VenueID	**Attribute**	**Value**
1	Cuisine	Deli
1	Price	$
2	Cuisine	Chinese
2	Price	$$$
2	Specialty	Scorpion Bowl
2	DJ	No
1	Name	Deli Llama
2	Name	Peking Inn
1	Address	Peachtree Road
2	Address	Lake ST

This data format is called ***triples***

Semantic meaning

Venue 1	has name	Deli Llama
Subject	Predicate	Object

Venue 1	serves	deli
Subject	Predicate	Object

- Single table - represents arbitrary facts about food and music venues
- Each triple is composed of a subject, a predicate, and an object.
- Each triple represents a simple linguistic statement

Semantic table

Venues		
Subject	**Predicate**	**Object**
S1	Cuisine	Deli
S1	Price	$
S2	Cuisine	Chinese
S2	Price	$$$
S2	Specialty	Scorpion Bowl
S2	DJ	No
S1	Name	Deli Llama
S2	Name	Peking Inn
S1	Address	Peachtree Road
S2	Address	Lake ST

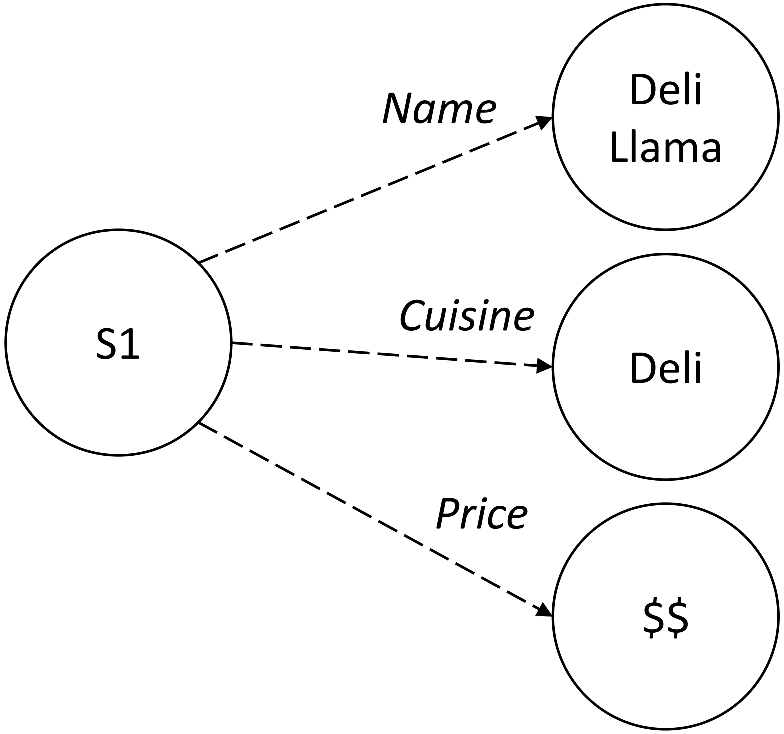
Semantic modeling

- The **subject** corresponds to an entity—a “thing” for which we have a conceptual class:
  - People
  - Places
  - Even periods of time and ideas
- **Predicates** are properties of the entity to which they are attached.
  - A person’s name or birth date
  - Restaurant location
- **Objects** fall into two classes:
  - Entities that can be the subject in other triples
  - Scalar values such as strings or numbers.

Data graph

- Multiple triples can be tied together by using the same subjects and objects in different triples
- As we assemble these chains of relationships, they form a *directed labeled graph*

Graph of venues: sample node

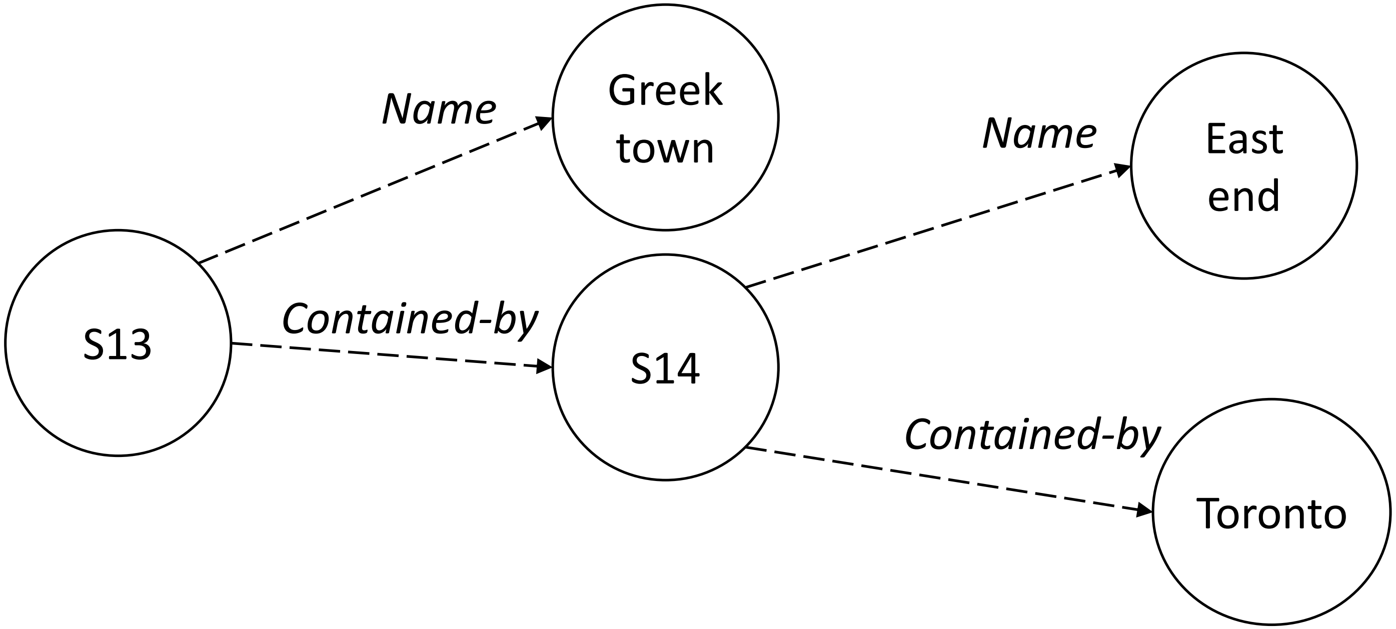


Integrating new entity: *neighborhood*

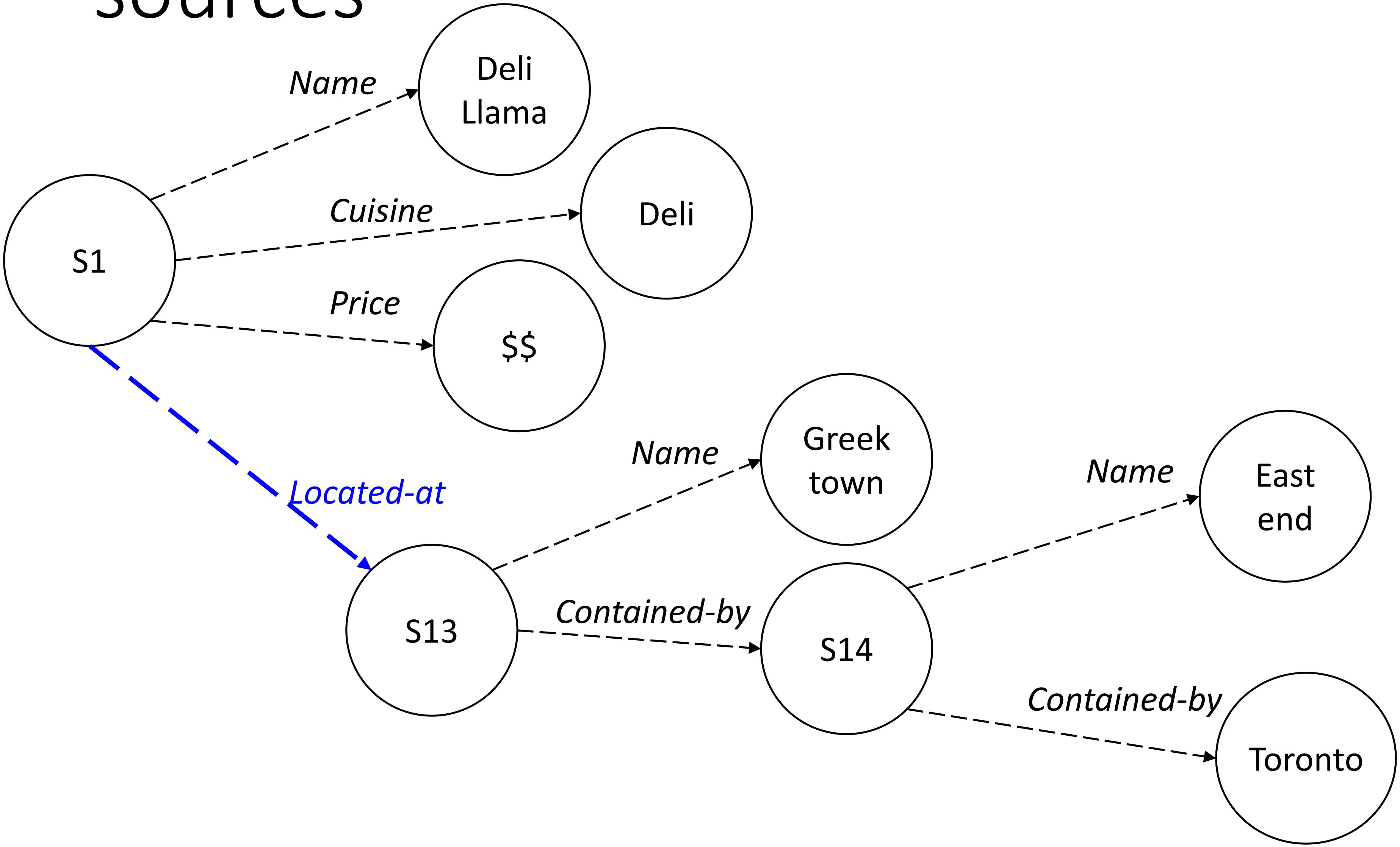
Neighborhoods		
Subject	**Predicate**	**Object**
S11	Name	Financial District
S11	Contained-by	S12
S12	Name	Downtown core
S12	Contained-by	Toronto
S13	Name	Greektown
S13	Contained-by	S14
S14	Name	East end
S14	Contained-by	Toronto

We can append neighborhood information to the same table as our venue data!

Graph of neighborhoods: sample node



Integrating data from multiple sources



Advantages of semantic model 1/5

- We can add any new data type into the same table

  Espresso machine locations, coffee shops, book stores, gas stations …

Advantages of semantic model 2/5

- We can add any new data type into the same table
- Self-describing data – do not need a special schema definition

  the semantic relationships that previously were inferred from the table and column are contained in data itself

Advantages of semantic model 3/5

- We can add any new data type into the same table
- Self-describing data – do not need a special schema definition
- Easy integration of data from multiple sources

  Just add new data to the same table and create a link to the old data if needed

Advantages of semantic model 4/5

- We can add any new data type into the same table
- Self-describing data – do not need a special schema definition
- Easy integration of data from multiple sources
- We can add new features without affecting legacy software

  no schema migration, there is the same simple schema all the time

Advantages of semantic model 5/5

- We can add any new data type into the same table
- Self-describing data – do not need a special schema definition
- Easy integration of data from multiple sources
- We can add new features without affecting legacy software
- Simple common data interface

  everyone can write an app in Python, or Ruby to plot crime statistics on the map or find cuisines in the walking distance from the movie

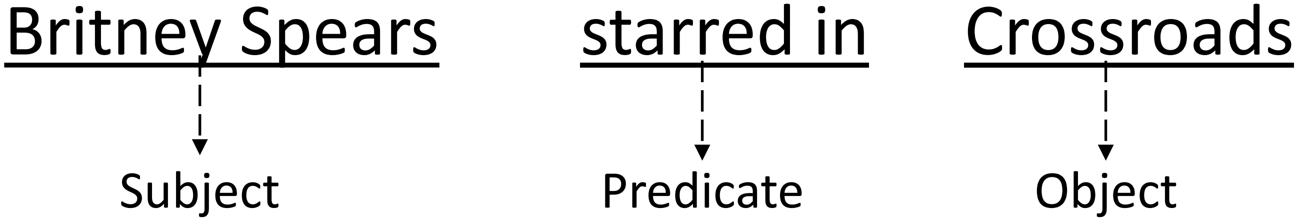
Semantic web

- RDF (Resource Description Framework) web data can be thought of in terms of a decentralized directed labeled graph wherein the arcs start with subject URIs, are labeled with predicate URIs, and end up pointing to object URIs or scalar values
- Uniform Resource Identifier (URI) is a string of characters used to uniquely identify a resource (for example for books - urn:isbn:0-486-27557-4)

Example: Celebrities dataset

- Entities – celebrity, relationship, rehab, album, movie
- Entities can be both subject and object
- Predicates:
  - enemy
  - person
  - released_album
  - starred_in
  - start
  - end
  - with

Let’s model *celebrity*



Britney Spears starred in Crossroads

Subject

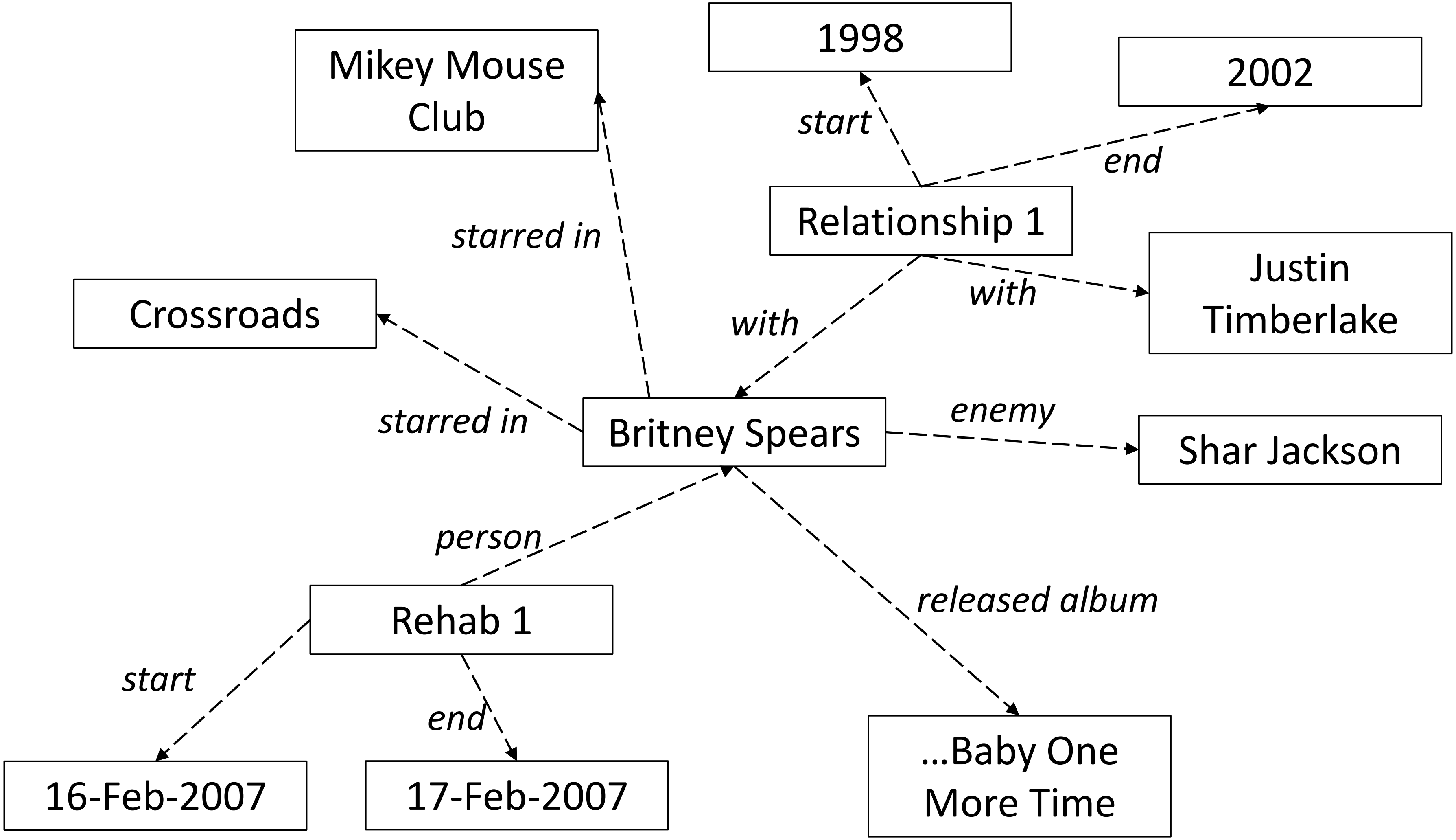
Predicate

Object

Let’s model *relationships*

Relationship1	with	Britney Spears
Subject	Predicate	Object
Relationship1	with	Justin Timberlake
Subject	Predicate	Object
Relationship1	start	1998
Subject	Predicate	Object
Relationship1	end	2002
Subject	Predicate	Object

Celebrity graph: sample node



Example 1. Which celebrities have dated more than one movie star?

```
CREATE VIEW movie_stars AS
SELECT distinct subject FROM celebrities
WHERE predicate = 'starred_in';

CREATE VIEW relationships AS
SELECT distinct R1.object AS celeb1, R2.object AS celeb2
FROM celebrities R1, celebrities R2
WHERE R1.predicate = 'with' AND R2.predicate = 'with'
AND R1.subject = R2.subject AND R1.object < R2.object;

SELECT distinct celeb1, COUNT(celeb2) AS cnt FROM relationships
WHERE celeb2 IN (SELECT * FROM movie_stars )
GROUP BY celeb1
HAVING cnt >=2;
```

Example 2. Which musicians have spent time in rehab?

```
CREATE VIEW musicians
AS select distinct subject from celebrities
where predicate = 'released_album';

 CREATE VIEW rehab_celebs
 AS SELECT distinct object FROM  celebrities
 WHERE predicate = 'person';

SELECT * from musicians
INTERSECT
SELECT * from rehab_celebs;
```

Triplestore implementation: indexes

- A common technique: cross-indexing the subject, predicate, and object in all different permutations so that all triple queries can be answered through fast lookups
- Each of the indexes holds a different permutation of each triple that is stored in the graph
- The name of the index (ops, osp, pos, pso, sop, spo) indicates the ordering of the terms in the index (i.e., the pos index stores the predicate, then the object, and then the subject, in that order)

Triplestore implementation: query format

- The basic query method takes a (subject, predicate, object) pattern and returns all triples that match the pattern.
- Terms in the triple that are set to None are treated as wildcards.
- The query determines which index to use based on which terms of the triple are wildcarded, and then iterates over the appropriate index

Queries can be implemented as triple matchings

(*, ‘with’, ‘Britney Spears’)

- We can put the results into a list variable – relationships

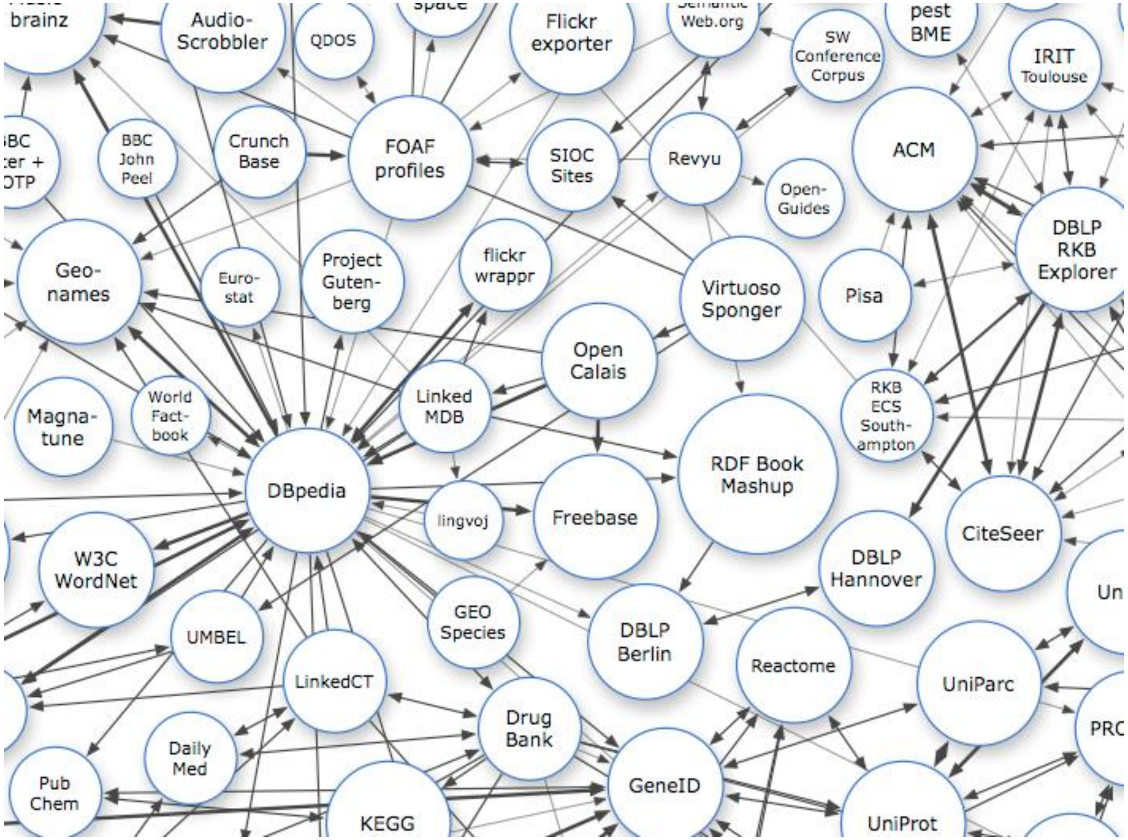
(‘?relationships’, ‘with’, ‘Britney Spears’)

- And use the results in a subsequent queries:

(‘relationships’, ‘with’, ‘?partners’)

http://linkeddata.org/

The goal: exposing, sharing, and connecting pieces of data, information, and knowledge on the Semantic Web using URIs and RDF



Semantic modeling example: international databases

- Consider a database that stores outlets of a business (McDonald’s?) in different countries
- We can model a business address as a sematic table
  - USA: address, zipcode, city, [county], state, country
  - Canada: address, zipcode, [county], province, country
  - France: address, zipcode, [region], country

NoSQL ("Not only SQL") databases

NoSQL database systems

- New generation of non-relational database systems

- Properties:
  - Flexibility: schema-less
  - Scalability: inherently parallelizable

Main types of NoSQL systems

✓ Graph databases: store data as connected nodes of a graph

  HyperGraphDB, multiple implementations of semantic RDF triplestores

- Key-value databases: key-value pairs

  Redis, SimpleDB

- Document databases: key-value stores where values are entire documents

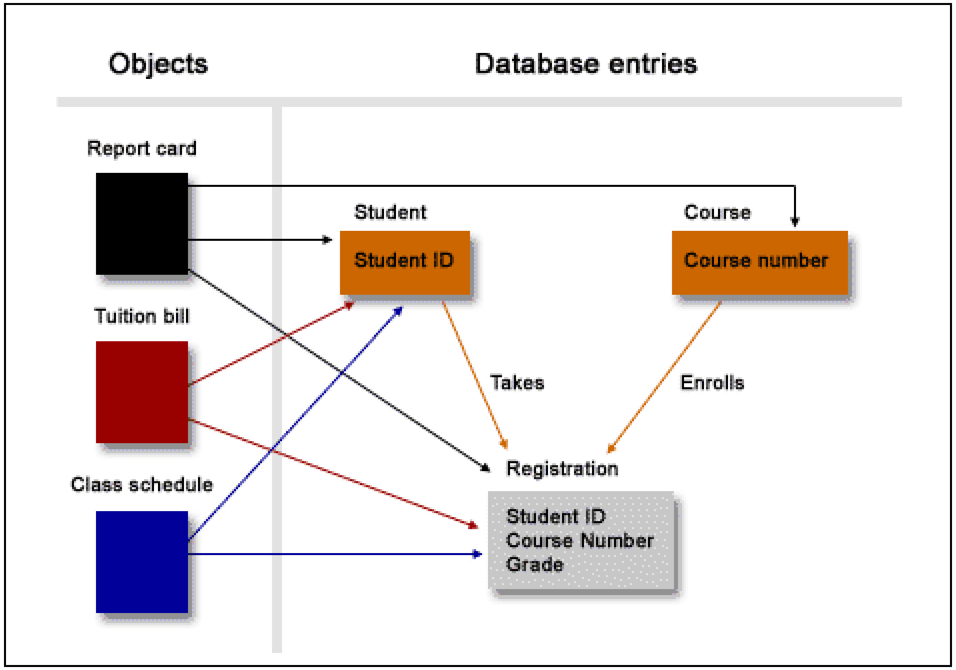
  CouchDB, MongoDB

- Wide-column databases: multi-dimensional sorted map

  Google's BigTable, Cassandra

Impedance mismatch

- Mismatch between tables and data structures in memory
  - For object-oriented languages: invented Object-Relational Mapping (ORM)
  - For other languages (functional, c) – **data structures just do not match!**



Relational databases predominate

1980

Relational databases

1990

Relational databases

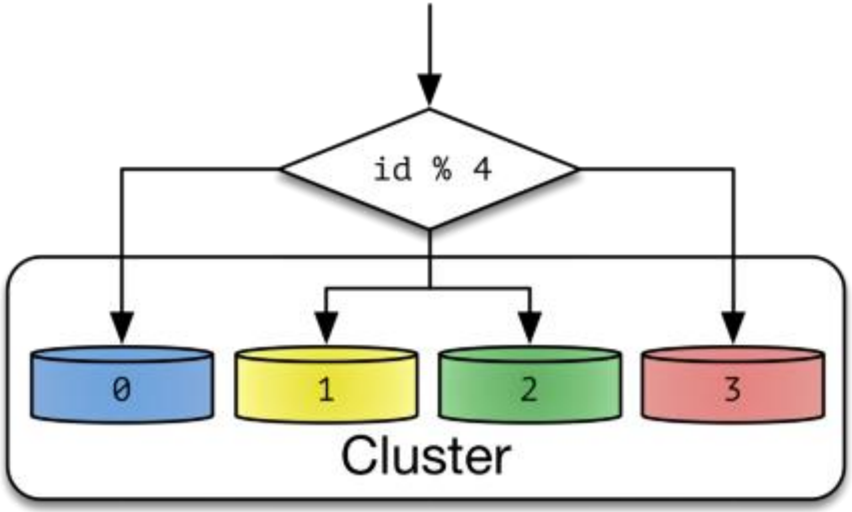
2000

2010

Scaling up

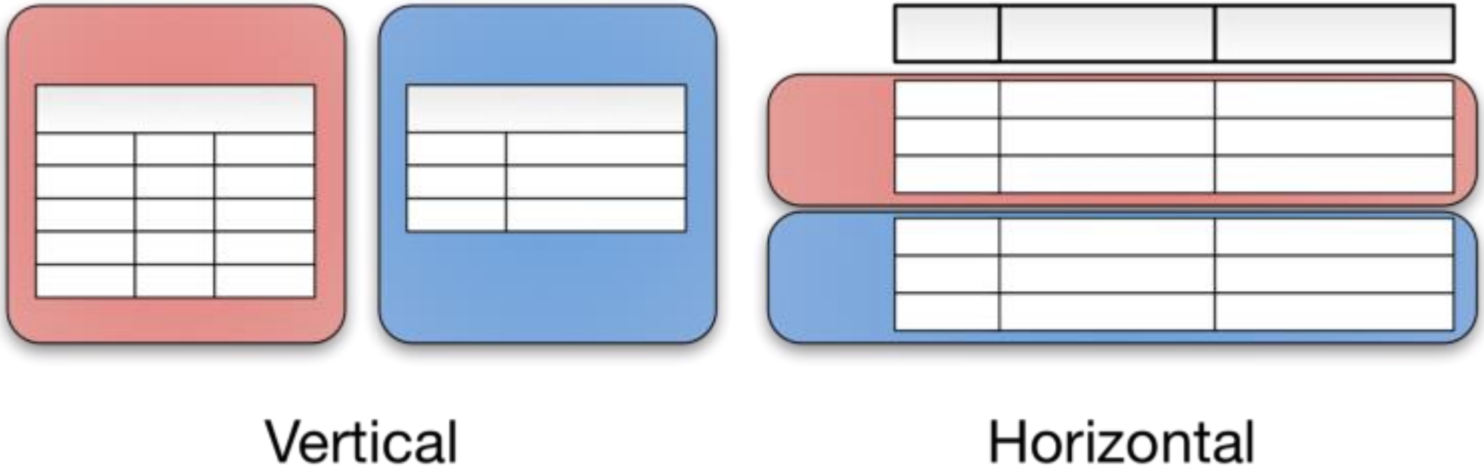
Two alternatives:

- Bigger servers
- Lots of little boxes in massive grids



Partitioning

- Vertical: normalization, splitting into smaller tables
- Horizontal: splitting single table into multiple sets of rows
  - Horizontal partitioning when rows are distributed across multiple nodes based on some attribute (for example, zip code) is called *sharding*



Parallelism is not natural for relational databases

- SQL designed to run as a single node
- Both vertical partitioning and horizontal partitioning introduce **performance bottlenecks**:
  - Increased latency when querying across more than one shard
  - Indexes are sharded by one dimension, so that some searches are optimal, and others are slow or impossible
  - Cross-shard consistency and durability is hard to achieve due to the more complex failure modes of a set of servers

New requirements on data management

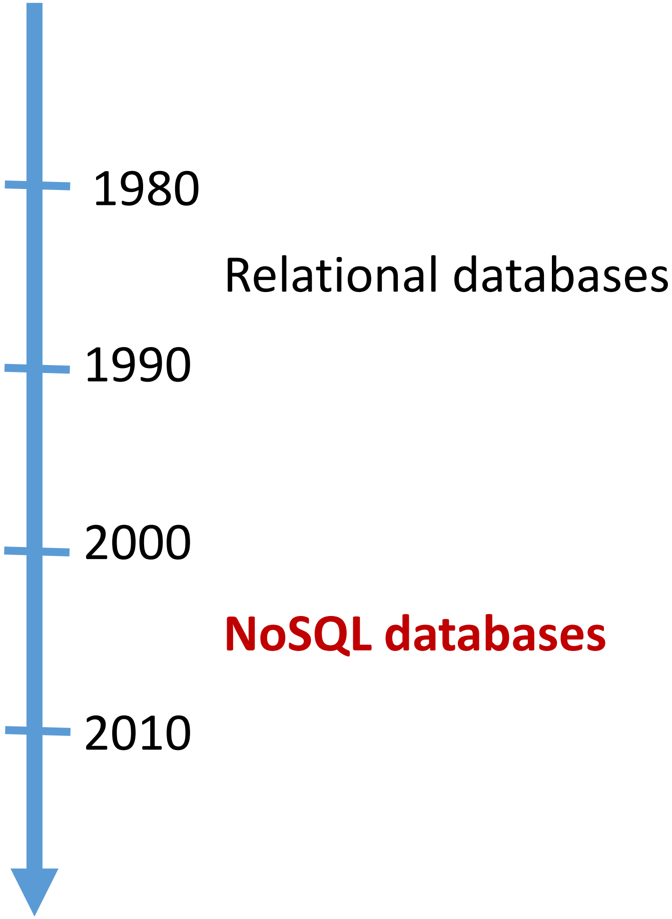
Trends

- **Volume** of data
- **Cloud** comp. (IaaS)
- **Velocity** of data
- **Big** traffic
- **Variety** of data

Requirements

- Real **scalability**
  - massive database distribution
  - dynamic resource management
  - horizontally scaling systems
- Frequent **update** operations
- Massive **read** throughput
- **Flexible** database schema

History



1980

Relational databases

1990

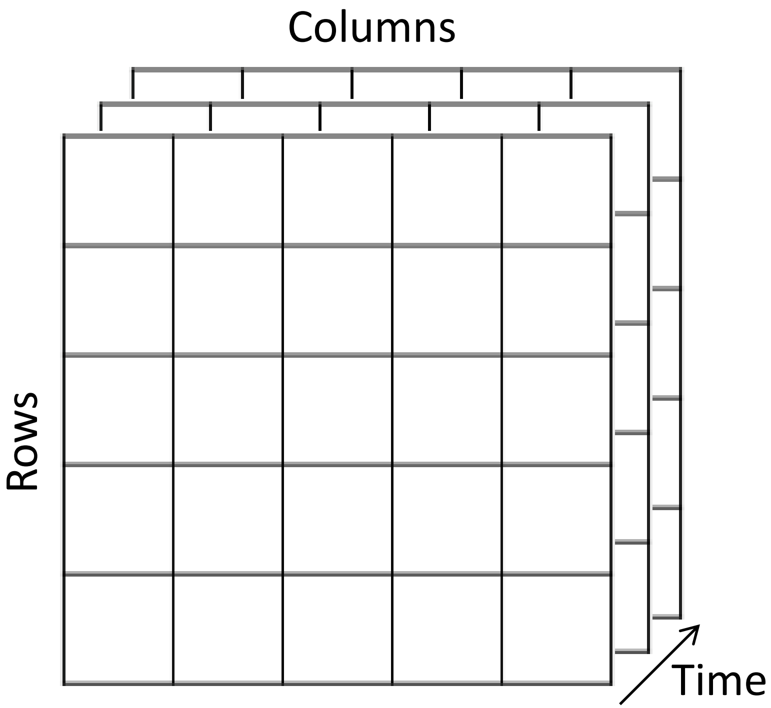
2000

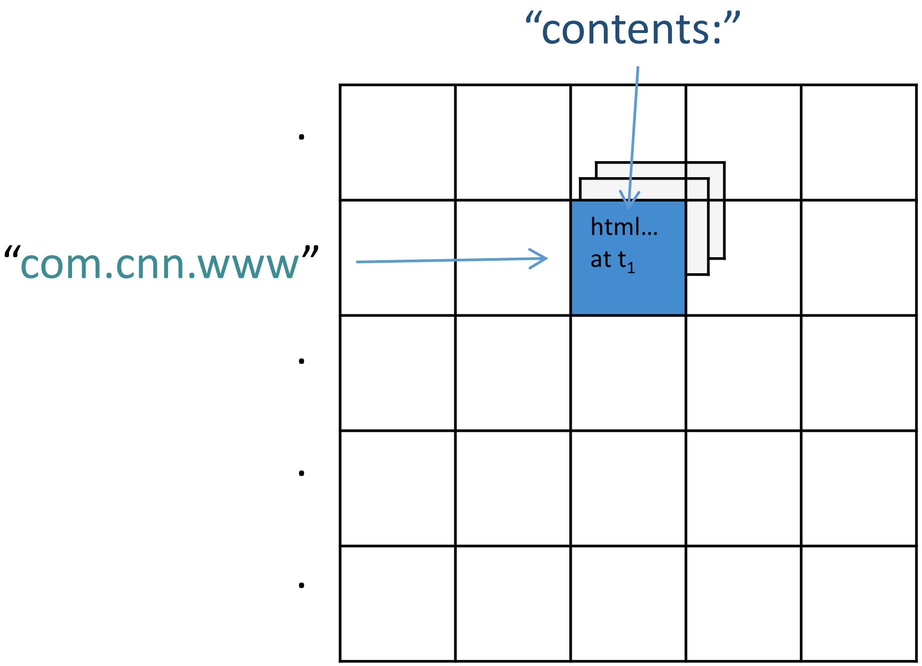
NoSQL databases

2010

Google BigTable (2006)

- Data model: three-dimensional indexed sorted map
  - Input (row, column, timestamp) → Output (cell contents)

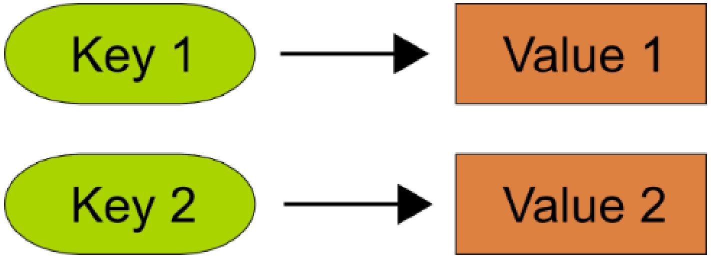




http://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf

Amazon: Dynamo DB (2007)

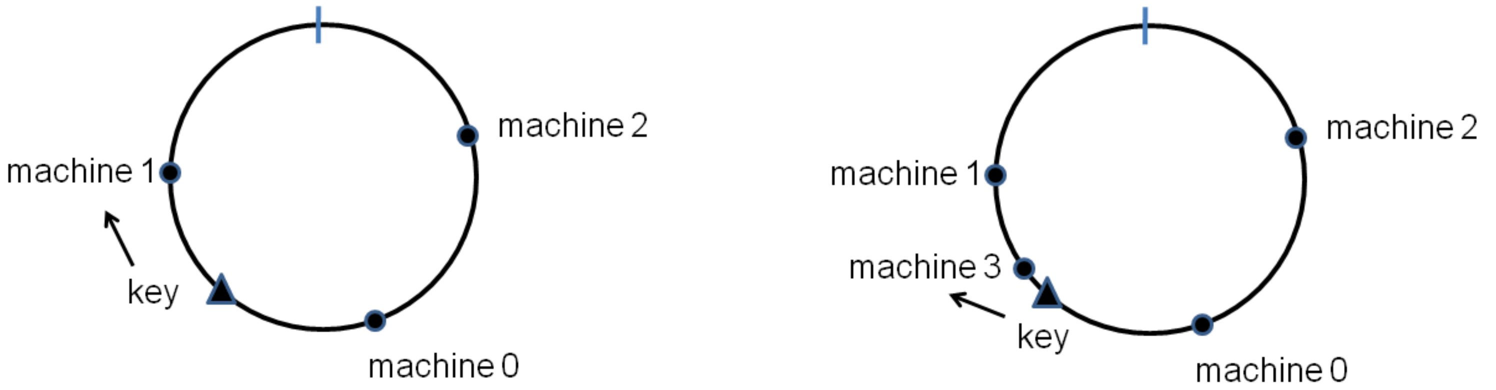
- Data model:
  simple hash table (map): key-value data store



http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

Dynamo: architecture

- Implemented as distributed hash table (DHT) based on consistent hashing – hashing into the place on the ring
- Elastic scalability: able to scale out one node at a time, with minimal impact on the system
- Decentralization



Common characteristics of NoSQl databases

- Not relational
- Cluster-friendly
- Schema-less

- Open source (mostly)

NoSQL categories by data models

1. Graph
2. Wide-column
3. Key - value (hash table)
4. Key - document

1. Graph Databases: Representatives

Neo4j

OrientDB®

APACHE GIRAPH

TITAN

InfiniteGraph

Ranked list: http://db-engines.com/en/ranking/graph+dbms

2. Column-family Stores: Representatives

Cassandra

H·BASE

HYPERTABLE

accumulo™

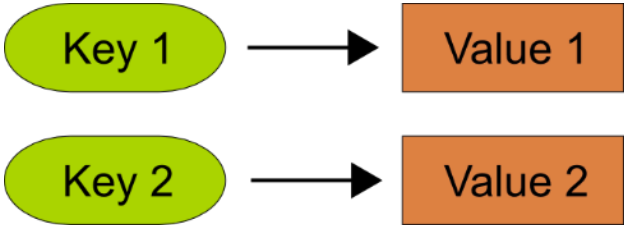
Ranked list: http://db-engines.com/en/ranking/wide+column+store

3. Key-value stores

- Value can be anything
- Search only by key – no structure inside the value

- Basic operations:

Get the value for the key	`value:= get(key)`
Put a value for a key	`put(key, value)`
Delete a key-value	`delete(key)`



3. Key-value Stores: Representatives

redis

MEMCACHED

riak

LevelDB

MapDB

ORACLE BERKELEY DB 12c

ORACLE NOSQL DATABASE

Infinispan

amazon DynamoDB

Project Voldemort

Ranked list: http://db-engines.com/en/ranking/key-value+store

4. Document stores

- Also key-value pairs
- But value is a semi-structured text data - document
- Documents are self-describing pieces of data
- Hierarchical tree data structures
  - Nested associative arrays (maps), collections, scalars
  - XML, JSON (JavaScript Object Notation), BSON, …
- Can query inside document: building search indexes on various document keys/fields

Document Data Formats

- Structured Text Data
  - JSON, BSON (Binary JSON)
    - JSON is currently number one data format used on the Web
  - XML: eXtensible Markup Language
  - RDF: Resource Description Framework

- Binary Data
  - often, we want to store objects (class instances)
  - objects can be binary-serialized (marshalled)
    - and kept in a key-value store
  - there are several popular serialization formats
    - Protocol Buffers, Apache Thrift

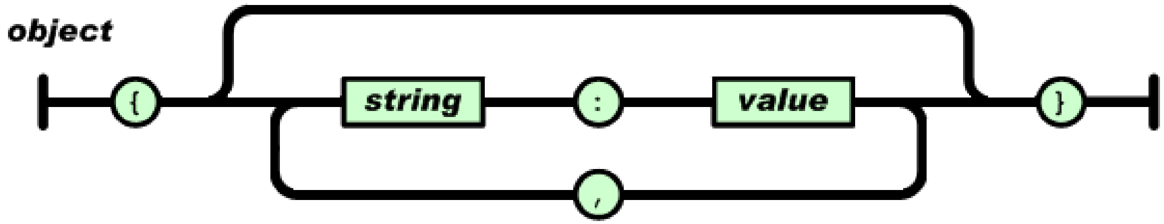
JSON: Basic Information

- Text-based open standard for data interchange
  - Serializing and transmitting structured data
- JSON = JavaScript Object Notation
  - Originally specified by Douglas Crockford in 2001
  - Derived from JavaScript scripting language
  - Uses conventions of the C-family of languages
- Filename: *.json
- Internet media (MIME) type: application/json

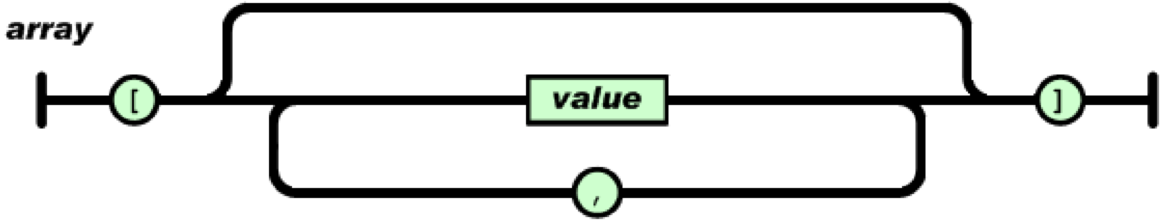
http://www.json.org

JSON: Data Types (1)

- object – an unordered set of key+value pairs
  - these pairs are called properties (members) of an object
  - syntax: { key: value, key: value, key: value, ...}

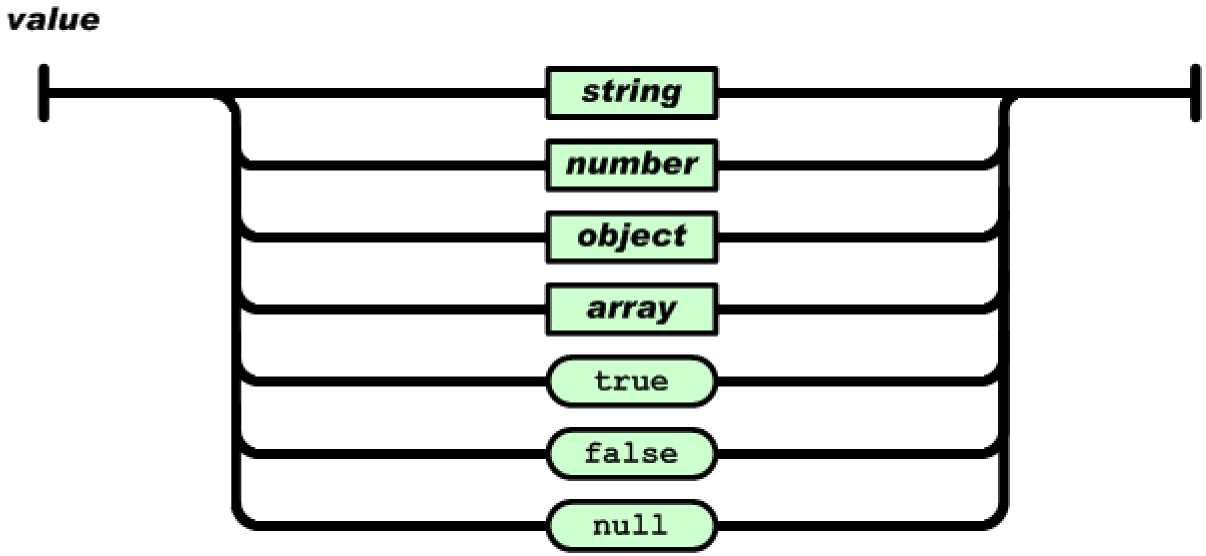


- array – an ordered collection of values (elements)
  - syntax: [comma-separated values]



JSON: Data Types (2)

- value – string in double quotes / number / true or false (i.e., Boolean) / null / object / array
  - Can be nested



Most documents have JSON format

```
key=3 ->   { "personID": "3",
             "firstname": "Martin",
             "likes": [ "Biking","Photography" ],
             "lastcity": "Boston",
             "visited": [ "NYC", "Paris" ] }

key=5 ->   { "personID": "5",
             "firstname": "Pramod",
             "citiesvisited": [ "Chicago", "London","NYC" ],
             "addresses": [
                { "state": "AK",
                  "city": "DILLINGHAM" },
                { "state": "MH",
                  "city": "PUNE" }  ],
             "lastcity": "Chicago“ }
```

Document store: sample query

Example in MongoDB syntax

- Query language expressed via JSON
- clauses: where, sort, count, sum, etc.

```
SQL:       SELECT * FROM users
MongoDB:   db.users.find()

SELECT * FROM users WHERE personID = "3"
db.users.find({"personID":"3"})

SELECT firstname,lastcity FROM users WHERE personID=5
db.users.find({"personID":"5"},
                  {firstname:1,lastcity:1})
```

Document Databases: Representatives

mongoDB

Apache
CouchDB
relax

RAVENDB

{ }
terrastore

Ranked list: http://db-engines.com/en/ranking/document+store

Schema-less?

`anOrder [“price”]*anOrder[“qty”]`

- Need to know the names of attributes
- Implicit schema: figure out the meaning of data

Consistency and concurrency

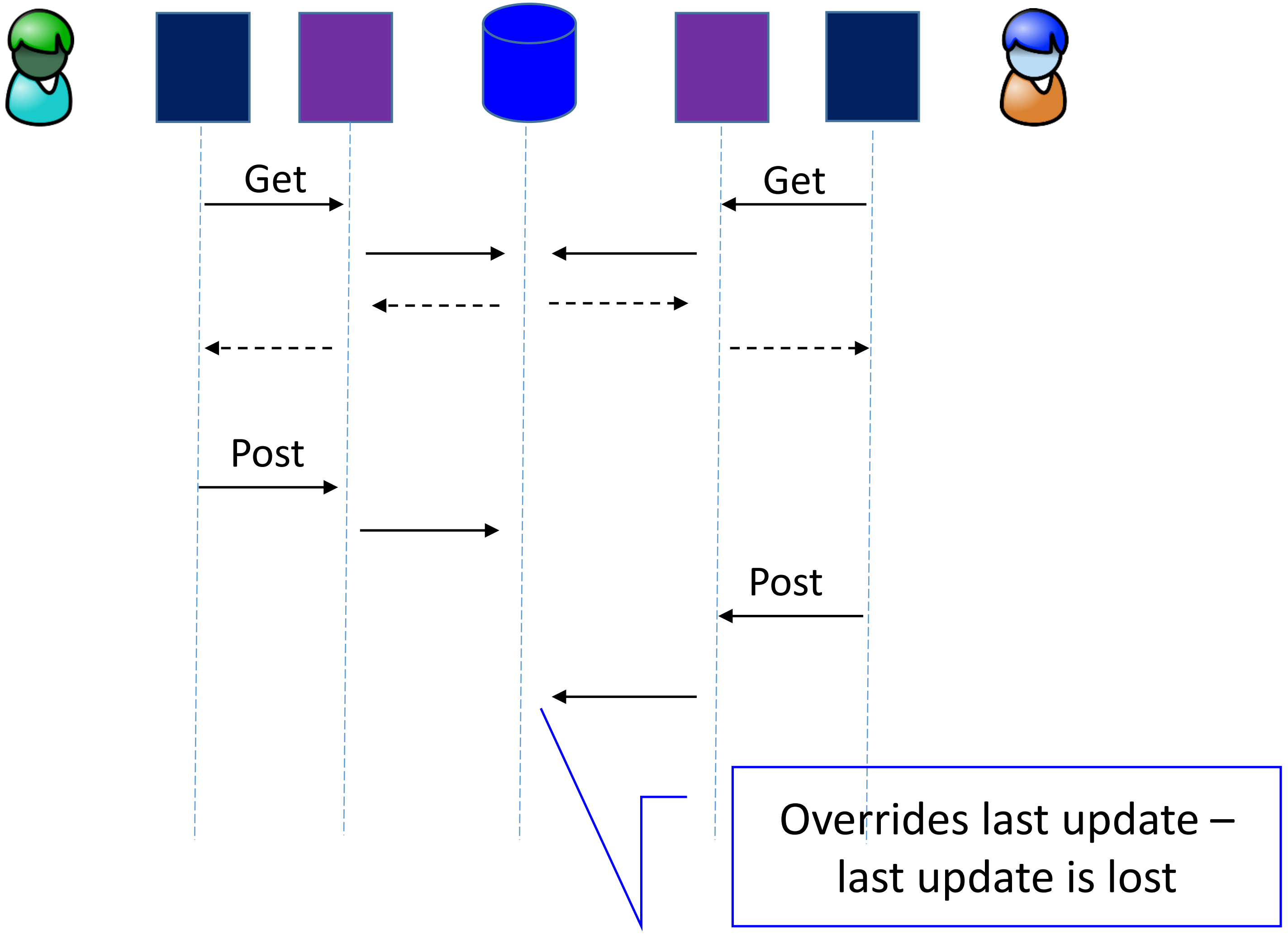
Consistency

- RDBMSs need ACID transactions – because data is in pieces
- We cannot afford that data is updated in chunks and parts of it are overridden
- We use transactions to wrap things together
- Graph databases do ACID updates

Multi-client system

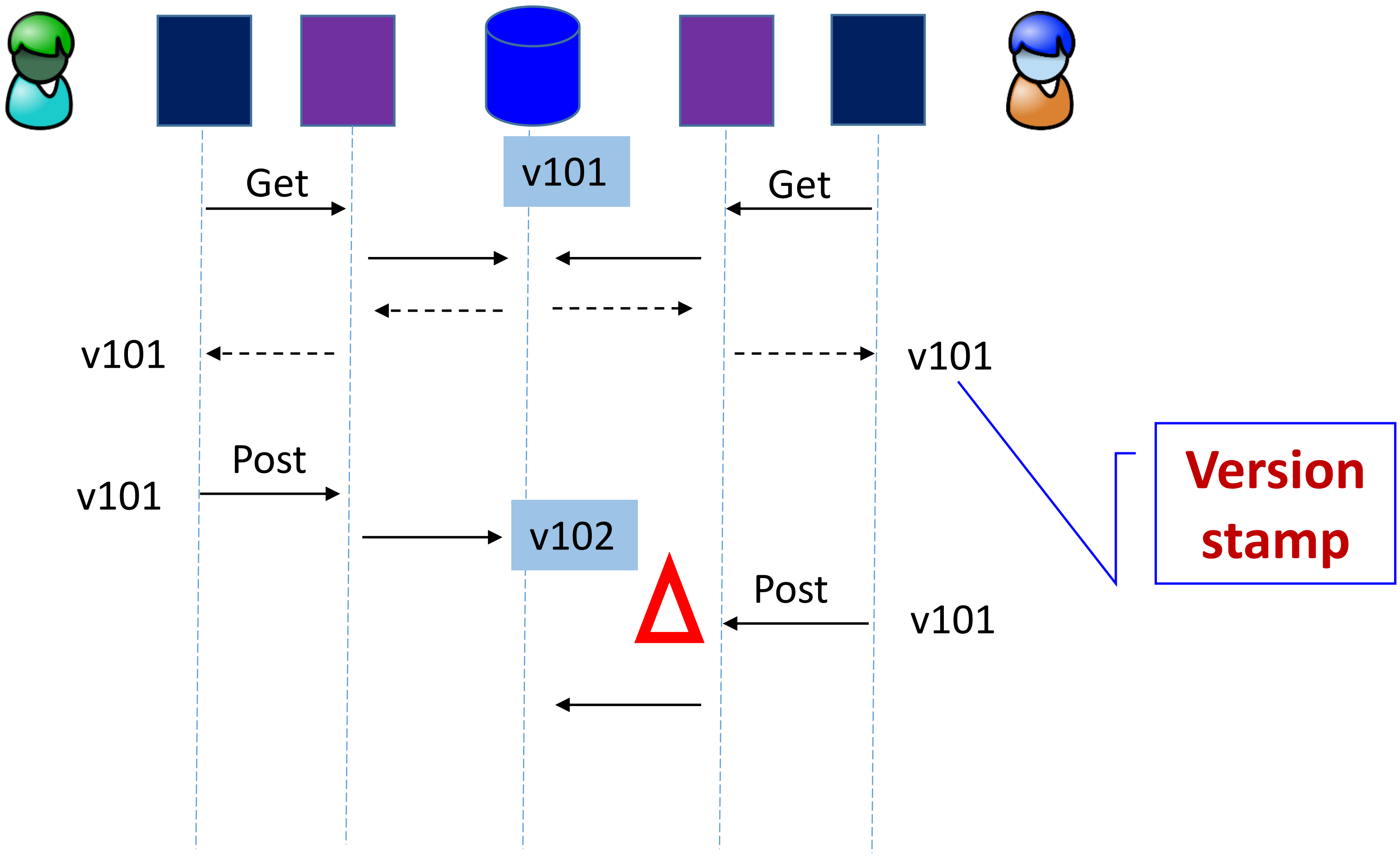
- ACID requires additional handling, because we cannot lock the entire table in web app domain
- Holding a transaction open – degrades performance

Offline lock

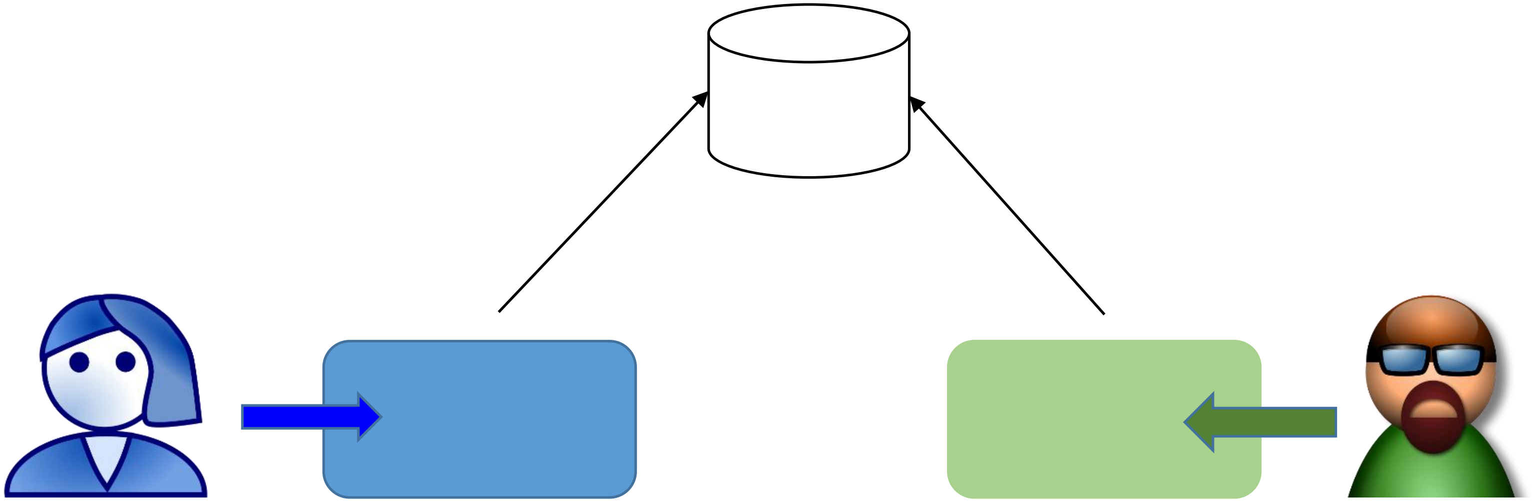


Overrides last update – last update is lost

Offline lock



Example: booking hotel rooms



- If the connection is temporarily lost at time of booking
- 2 alternatives
  - Prohibit
  - Allow double-booking

- Consistency vs availability
- This is a business choice, not a technical choice

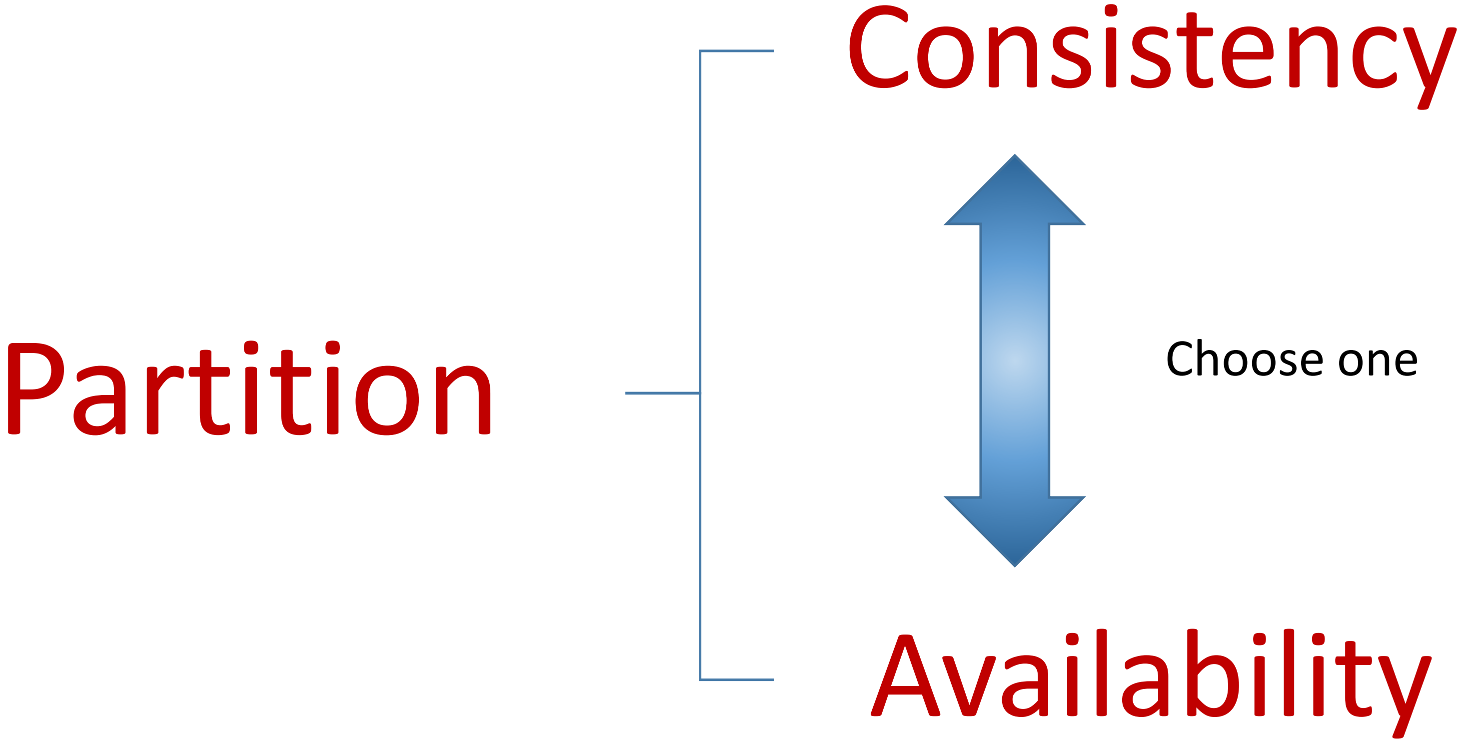
CAP theorem

- Tradeoff between:
  - Consistency (no overbooking)
  - Availability (response time)
  - Partition tolerance (parallelism)

- Can have only 2 out of 3

- Consistency vs response time of your server

In partitioned systems



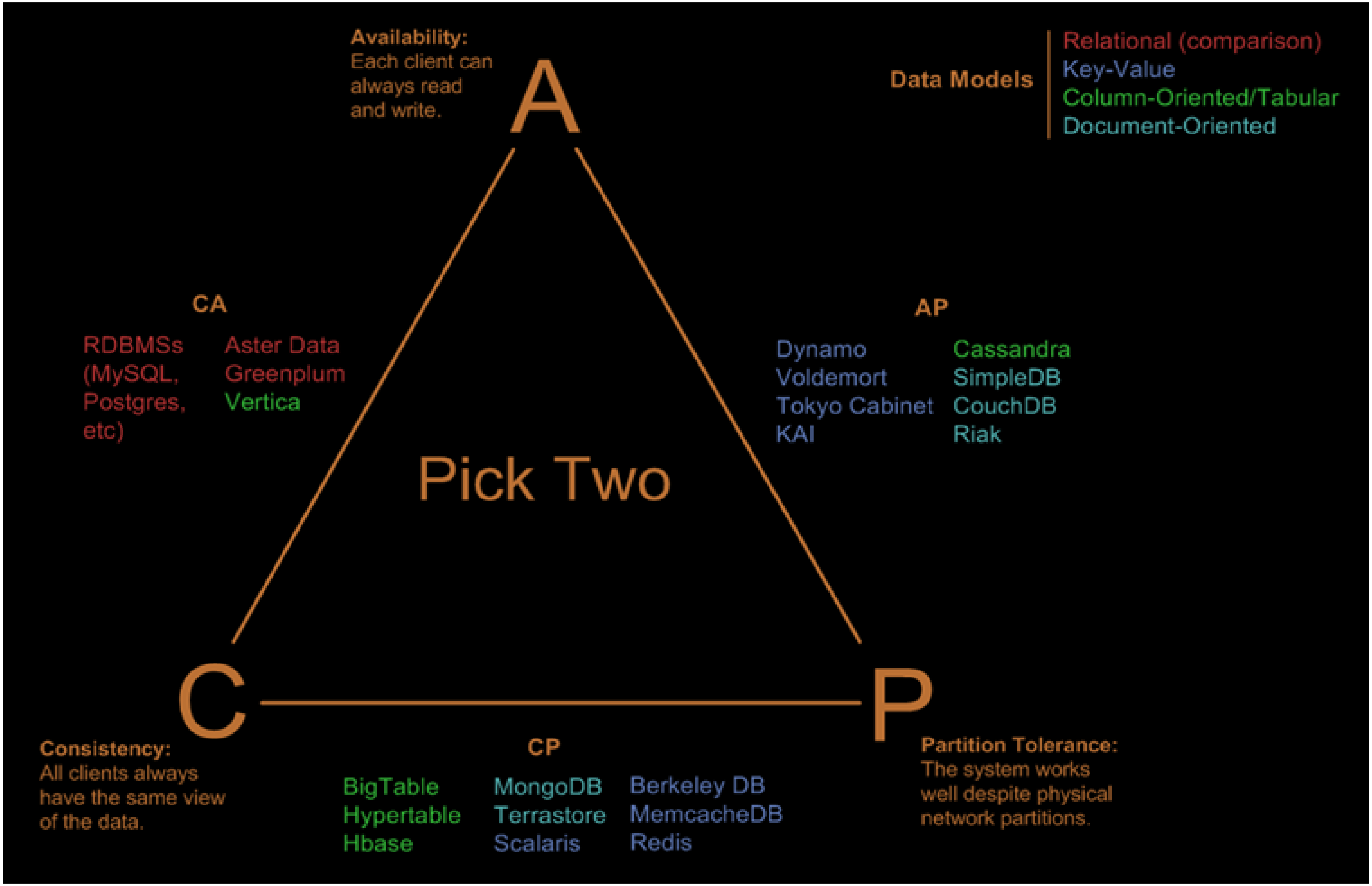
Partition

Consistency

Choose one

Availability

CAP theorem and DBMSs



Availability:
Each client can always read and write.

A

Data Models

Relational (comparison)
Key-Value
Column-Oriented/Tabular
Document-Oriented

CA

RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica

AP

Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak

Pick Two

C

P

Consistency:
All clients always have the same view of the data.

CP

BigTable
Hypertable
Hbase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis

Partition Tolerance:
The system works well despite physical network partitions.

When to use NoSQL

- Large amounts of data
- Complex evolving schema
- The domain matches graph or document
- Ease of development: rapid time to market
- Projects that give you a strategic advantage

http://www.tim-wellhausen.de/papers/NoSQL-Patterns/NoSQL-Patterns.html

Future?

1980

Relational databases

1990

2000

NoSQL databases

2010

Polyglot persistence

One Example of NoSQL Usage: Facebook

Facebook statistics (Spring 2014)

- 1.28 billion users (1.23B active monthly)
- 300 PB of user data stored
- 10 billion messages sent daily
- 250 billion stored photos (350 million uploaded daily)

2009: 10,000 servers
2010: 30,000 servers
2012: 180,000 servers (estimated)

source: http://expandedramblings.com/index.php/by-the-numbers-17-amazing-facebook-stats/

Database Technology Behind Facebook

Apache Hadoop http://hadoop.apache.org/

- Hadoop File System (HDFS)
  - over 100 PB in a single HDFS cluster
- an open source implementation of MapReduce:
  - Enables efficient parallel calculations on massive amounts of data

Apache Hive http://hive.apache.org/

- SQL-like access to Hadoop-stored data
- integration of MapReduce query evaluation

sources: http://goo.gl/SZ6jia http://royal.pingdom.com/2010/06/18/the-software-behind-facebook/

Database Technology Behind Facebook II

Apache HBase http://hbase.apache.org/

- a Hadoop column-family database
- used for e-mails, instant messaging and SMS
- replacement for MySQL and Cassandra

Memcached http://memcached.org/

- distributed key-value store
- used as a cache between web servers and MySQL servers since the beginning of FB

sources: http://goo.gl/SZ6jia http://royal.pingdom.com/2010/06/18/the-software-behind-facebook/

Database Technology Behind Facebook III

Apache Giraph http://giraph.apache.org/

- graph database
- facebook users and connections is one very large graph
- used since 2013 for various analytic tasks

RocksDB http://rocksdb.org/

- high-performance key-value store
- developed internally in FB, now open-source

sources: https://code.facebook.com/posts/509727595776839/scaling-apache-giraph-to-a-trillion-edges/ http://goo.gl/XNtG6p