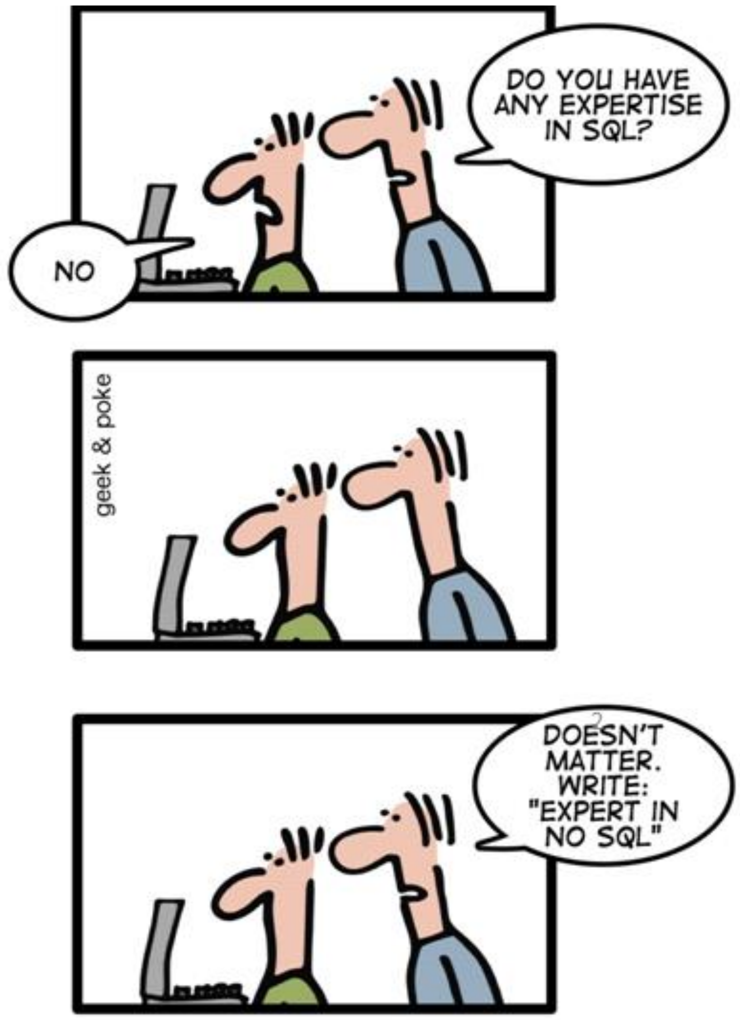
NOSQL

EECS4415 – Big Data Systems

How to leverage the NOSQL boom?



Leverage the NoSQL boom

Overview

- Part I: Structured, unstructured, semi-structured data
- Part II: What is NOSQL?
- Part III: NOSQL taxonomy

Thanks to Suprio Ray for some material in these slides

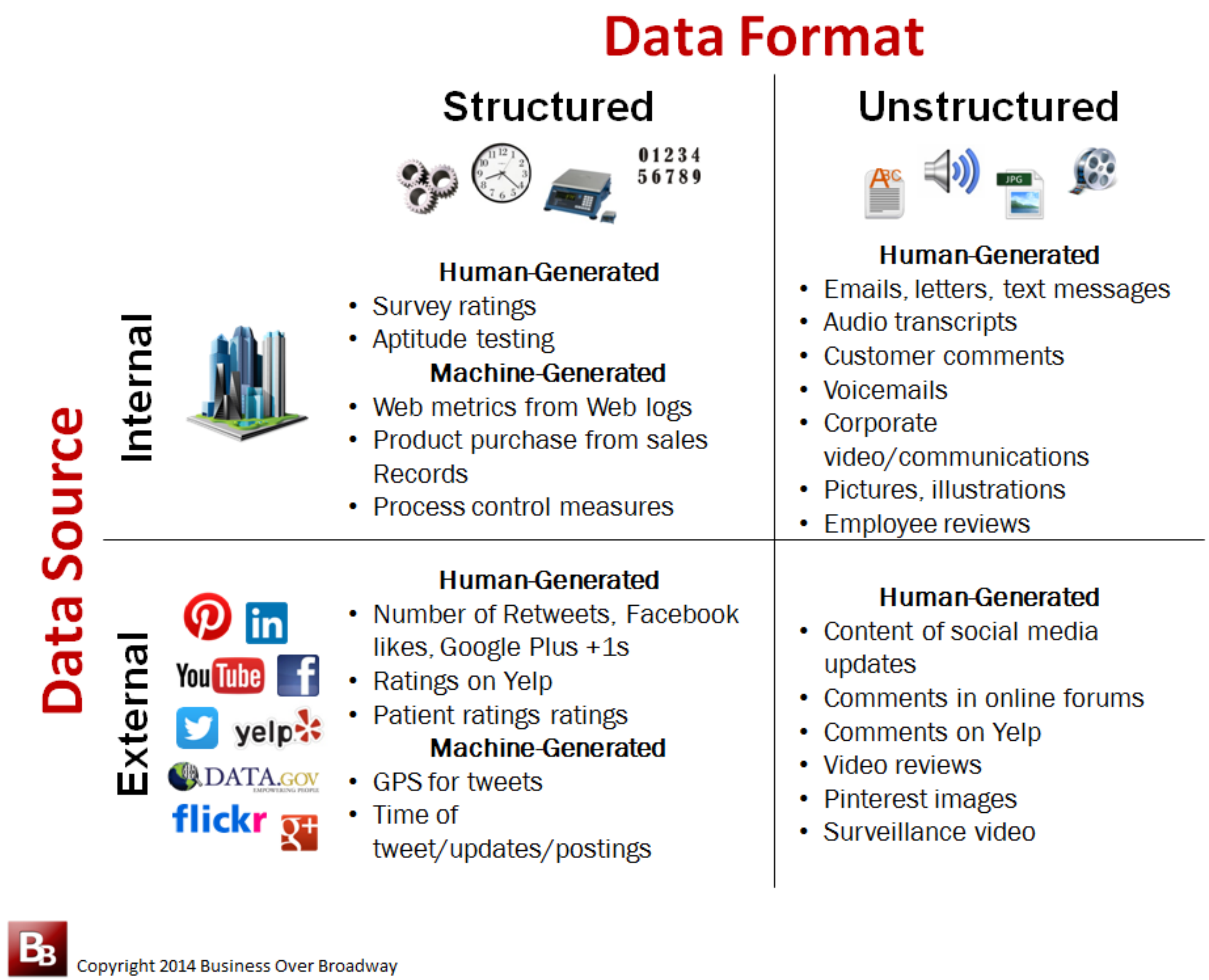
Part I: Structured, Unstructured and Semi-structured Data

Structured vs. unstructured data

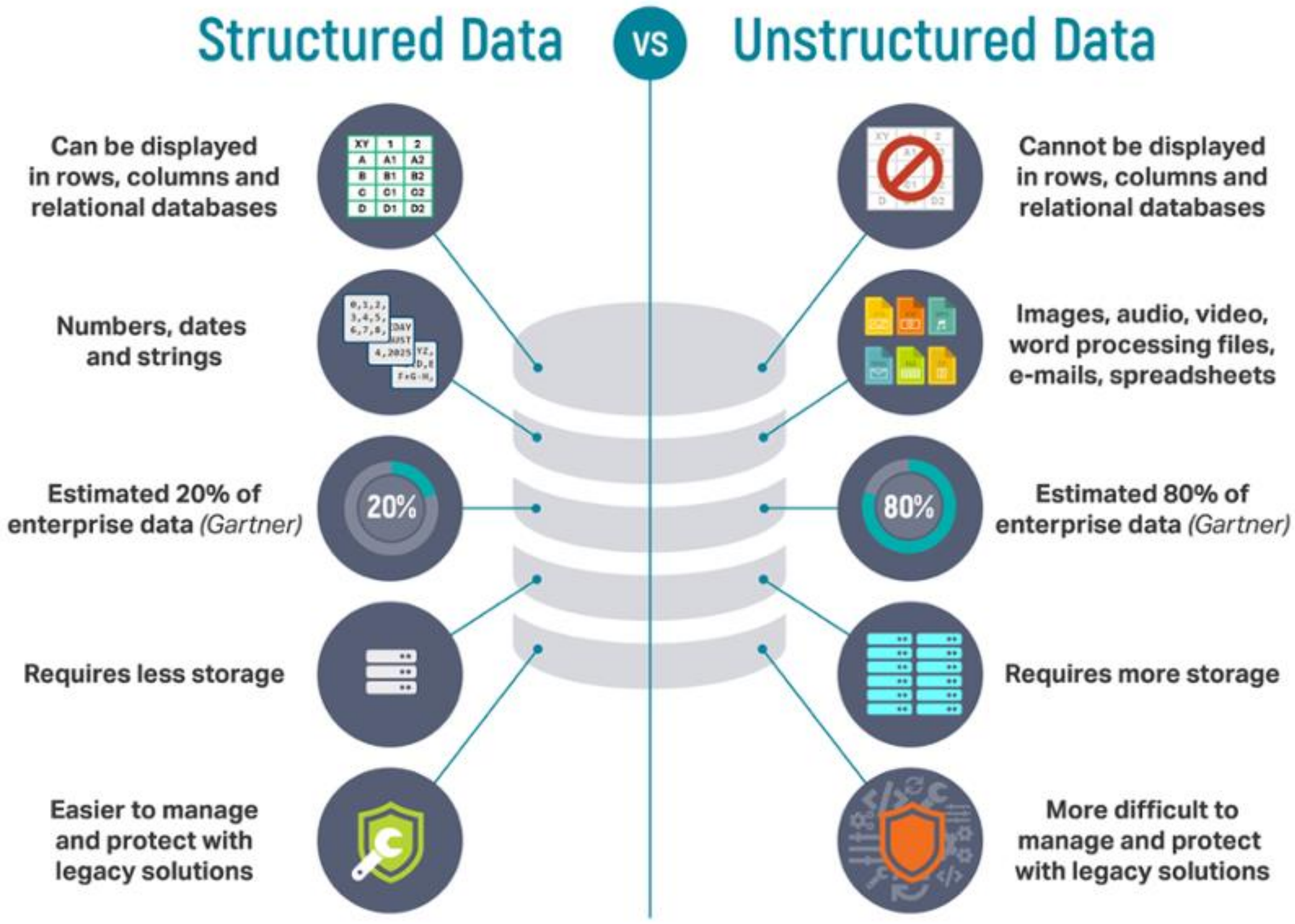
- Databases are highly structured
  - Well-known data format: relations and tuples
  - Every tuple conforms to a known schema
  - Data independence? Woe unto you if you lose the schema
- Plain text is unstructured
  - Cannot assume any predefined format
  - Apparent organization makes no guarantees
  - Self-describing: little external knowledge needed

    ... but have to infer what the data means

Structured vs. unstructured data (examples)



Structured vs unstructured data



source: https://www.igneous.io/blog/structured-data-vs-unstructured-data

Semi-structured data

- Observation: most data has some structure
  - Text: sentences, paragraphs, sections, ...
  - Books: chapters
  - Web pages: HTML
- Idea of semistructured data:
  - Enforce “well-formatted” data

  => Always know how to read/parse/manipulate it

  - Optionally, enforce “well-structured” data also

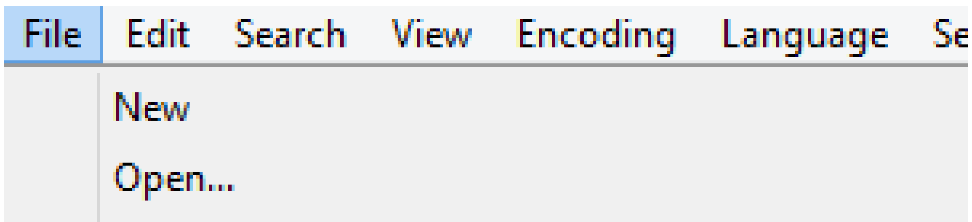
  => Adheres to a less-strict schema

  => Might help us interpret the data, too

Pro: highly portable *Con: verbose/redundant*

Semi-structured data: JSON

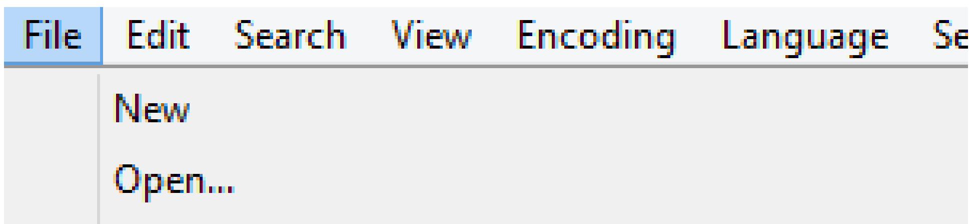
Describing a menu:




```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

Semi-structured data: XML

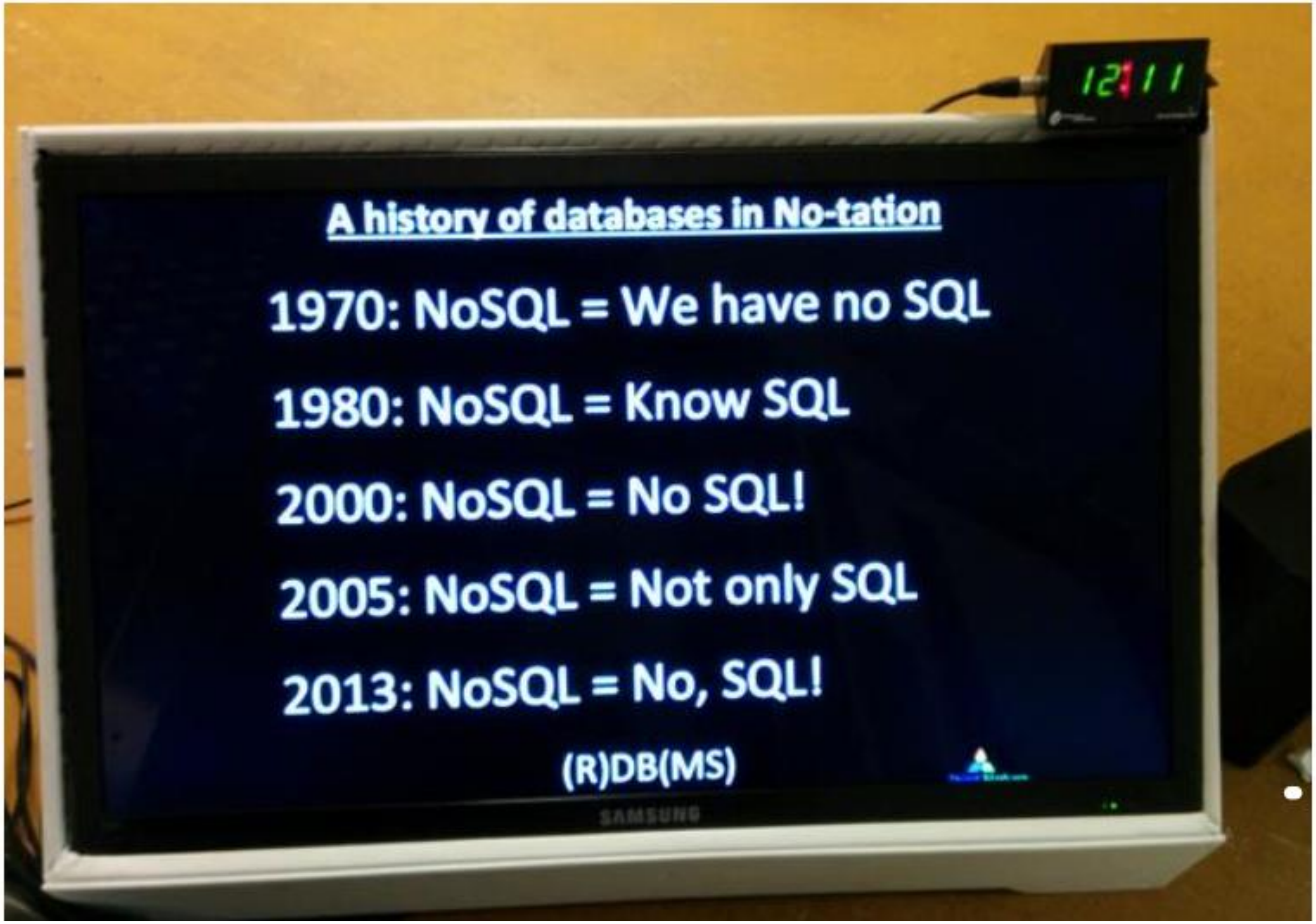
Describing a menu:




```xml
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()" />
    <menuitem value="Open" onclick="OpenDoc()" />
    <menuitem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```

Part II: What is NOSQL?

NoSQL



source: Mark Madsen

NoSQL Definition

From www.nosql-database.org:

> Next generation databases mostly addressing some of the points: being *non-relational*, *distributed*, *open-source* and *horizontal scalable*. The original intention has been modern web-scale databases. The movement began early 2009 and is growing rapidly. Often more characteristics apply as: *schema-free*, *easy replication support*, *simple API*, *eventually consistent* / BASE (not ACID), a huge data amount, and more.

Motivation: avoid RDBMS/SQL limitations

- Harder to scale - **expensive**
- Joins across multiple nodes - **hard**
- How does RDBMS handle data growth - **hard**
- Rigid schema design - **not manageable**
- Need for a DBA - **expensive**

NoSQL Distinguishing Characteristics

- Can handle large data volumes
  - "big data"
- Scalable replication and distribution
  - Thousands of machines distributed around the world
  - "Queries" can return answers quickly
- Schema-less (schema-at-read vs schema-at-write)
- ACID transaction properties are not needed – BASE
- CAP Theorem

Scaling vertically vs. horizontally

Vertical Scaling / Scale Up

- Upgrade to more powerful hardware
- Issues:
  - additional investment
  - single point of failure (SPOF)

Horizontal Scaling / Scale Out

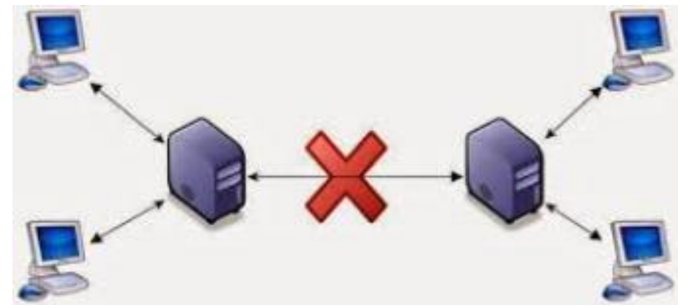
- Add extra identical boxes to server
- Issues
  - network communication
  - workload balancing
  - additional Investment

Network partition

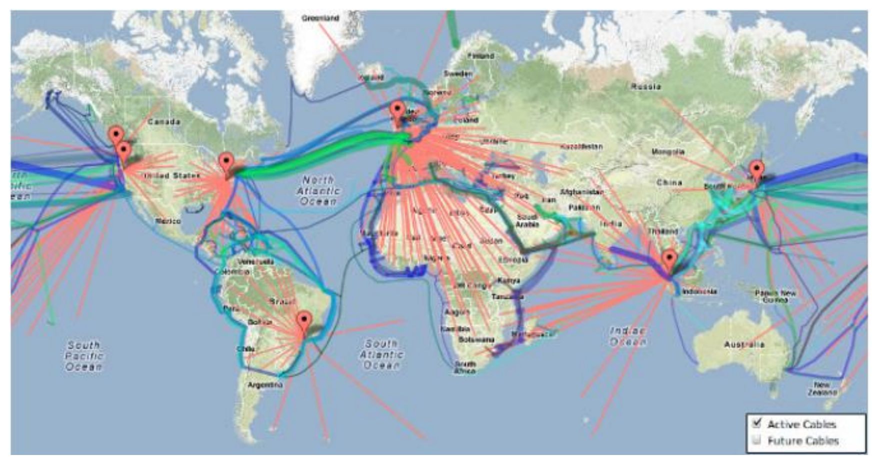
To scale out, you need a distributed store (cluster of servers)

=> can lead to network partition

=> refers to failures of network that causes communication interruptions



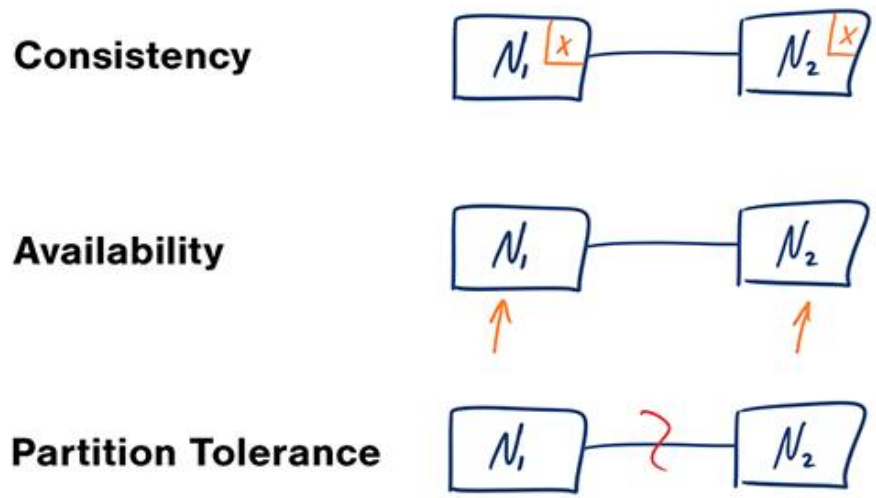
AWS data centers with worldwide underwater cables



(src: http://turnkeylinux.github.io/aws-datacenters/)

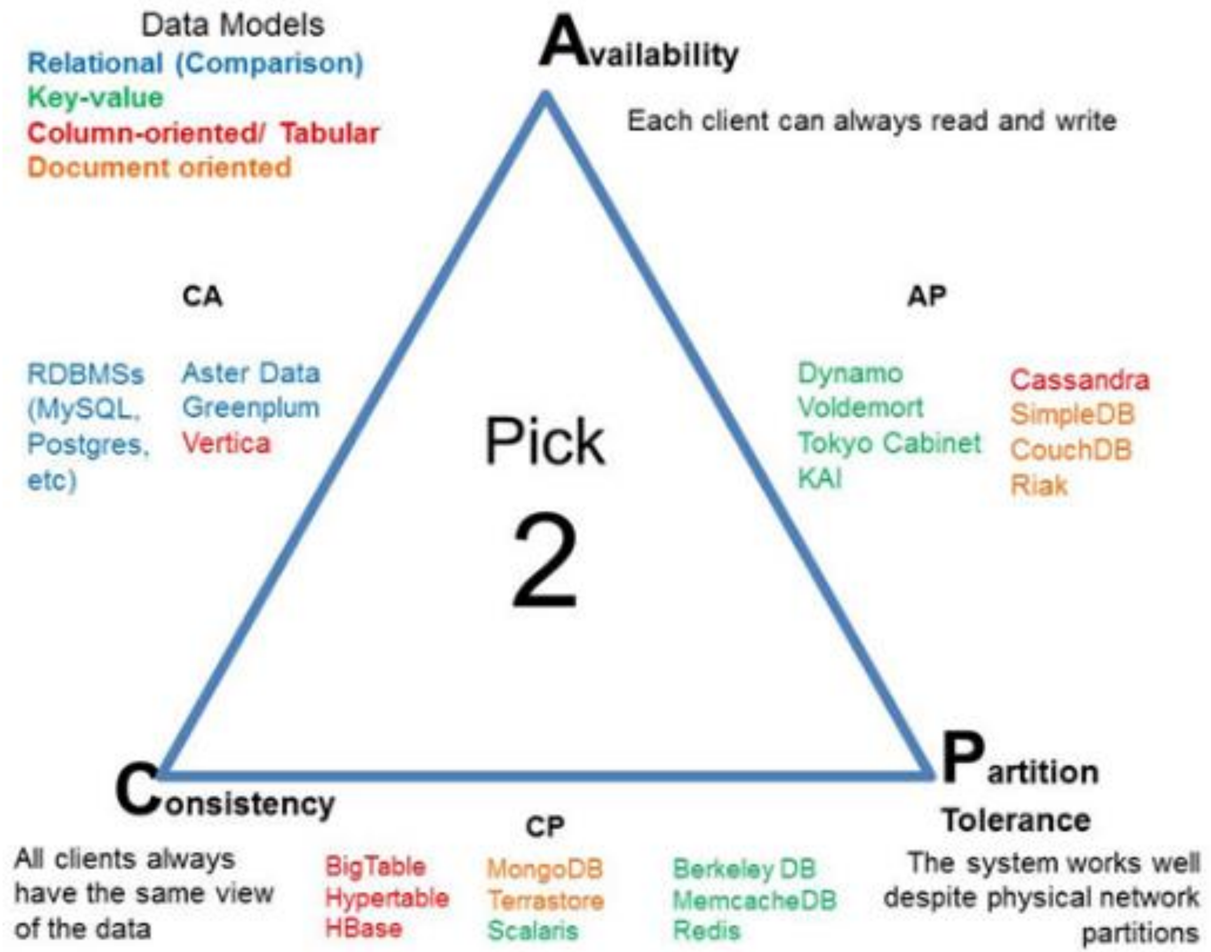
CAP Theorem

It is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees



Consistency: Every read receives the most recent write or an error
Availability: Every request receives a (non-error) response – without guarantee that it contains the most recent write
Partition tolerance: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

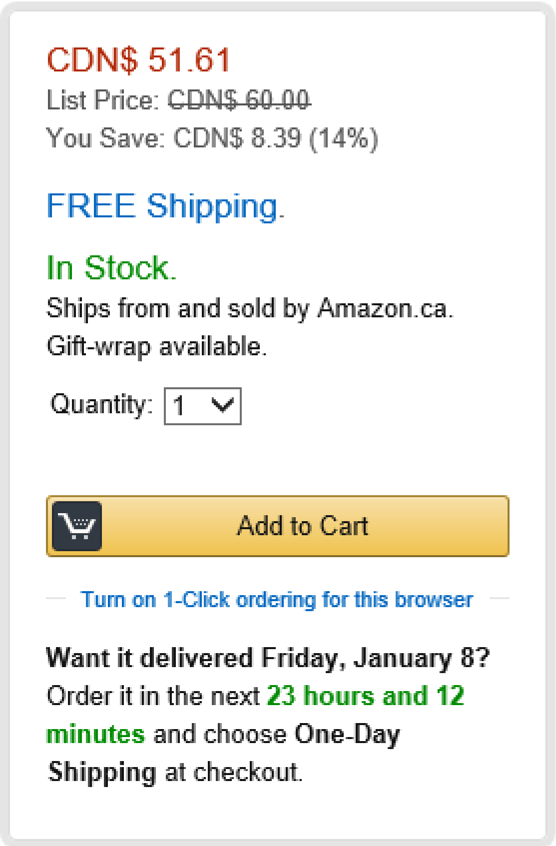
CAP Theorem & example data stores



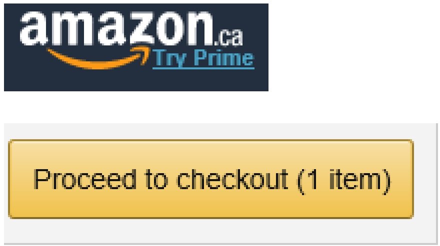
CAP Theorem in real-life

Amazon shopping cart: <u>adding to the shopping cart</u>

- **A**vailability
  - **always want to honor requests** to add items to a shopping cart
- **C**onsistency

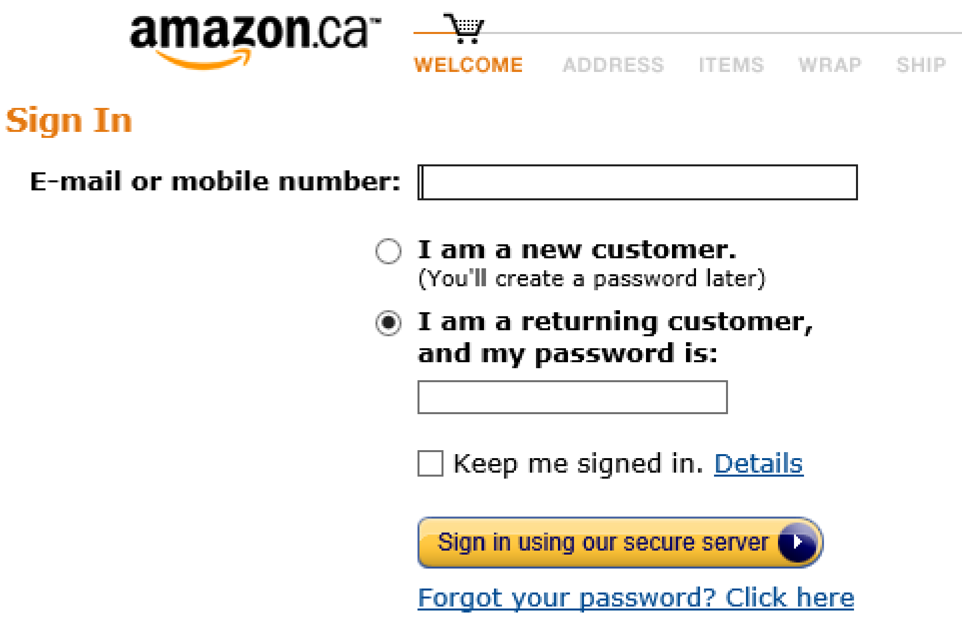


CAP Theorem in real-life



Amazon shopping cart: checkout process

- **A**vailability
- **C**onsistency
  - you favor consistency because several services are simultaneously accessing the data (credit card processing, shipping and handling, reporting)



ACID vs. BASE

Relational

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

⟺

NoSQL

- **B**asically
- **A**vailable (CP)
- **S**oft-state
- **E**ventually consistent (AP)

Pritchett, D.: BASE: An Acid Alternative (queue.acm.org/detail.cfm?id=1394128)

Recap: Transactions – ACID Properties

- **Atomic**: all of the work in a transaction completes (commit) or none of it completes
- **Consistent**: a transaction transforms the database from one consistent state to another consistent state; consistency is defined in terms of constraints
- **Isolated**: the results of any changes made during a transaction are not visible until the transaction has committed
- **Durable**: the results of a committed transaction survive failures

BASE Transactions

Acronym contrived to be the opposite of ACID

- **Basically Available**: system seems to work all the time - some parts of system remain available on failure
- **Soft state**: it does not have to be consistent all the time
- **Eventually Consistent**: as the data is written, the latest version is on at least one node. The data is then versioned/replicated to other nodes within the system. Eventually, the same version is on all nodes

BASE Transactions

- Characteristics
  - Availability first
  - Best effort
  - Weak consistency – stale data OK
  - Approximate answers OK
  - Simpler and faster

NoSQL advantages

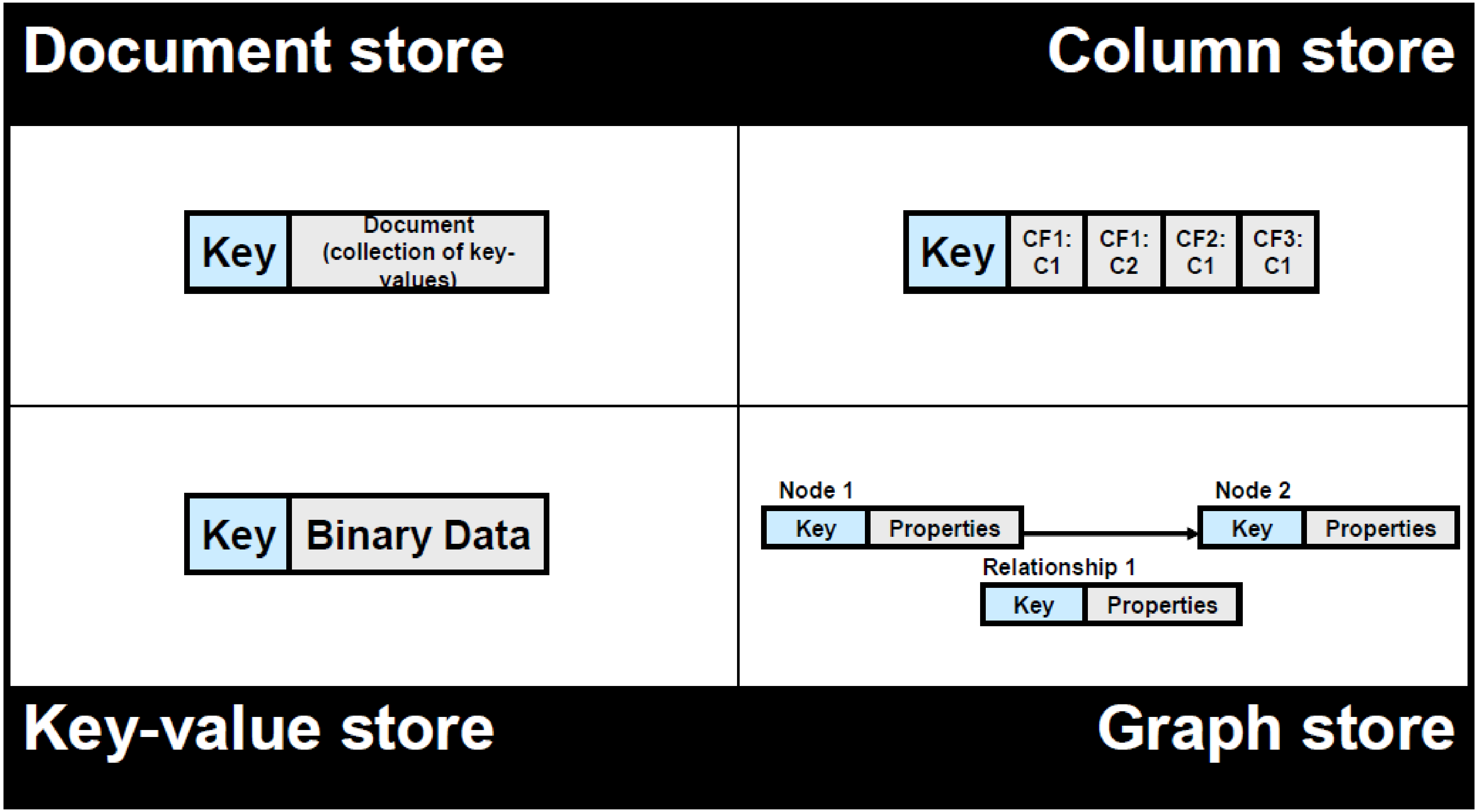
- Cheap, easy to implement (open source)
- Data are replicated to multiple nodes (therefore identical and fault-tolerant) and can be partitioned
  - Down nodes easily replaced
  - No single point of failure
- Can scale up and down
- Doesn't require a schema

What am I giving up?

- Joins (in many cases)
- ACID transactions
- SQL, as a sometimes frustrating, but still powerful query language
- Easy integration with other SQL-based applications

Part III: NOSQL Taxonomy

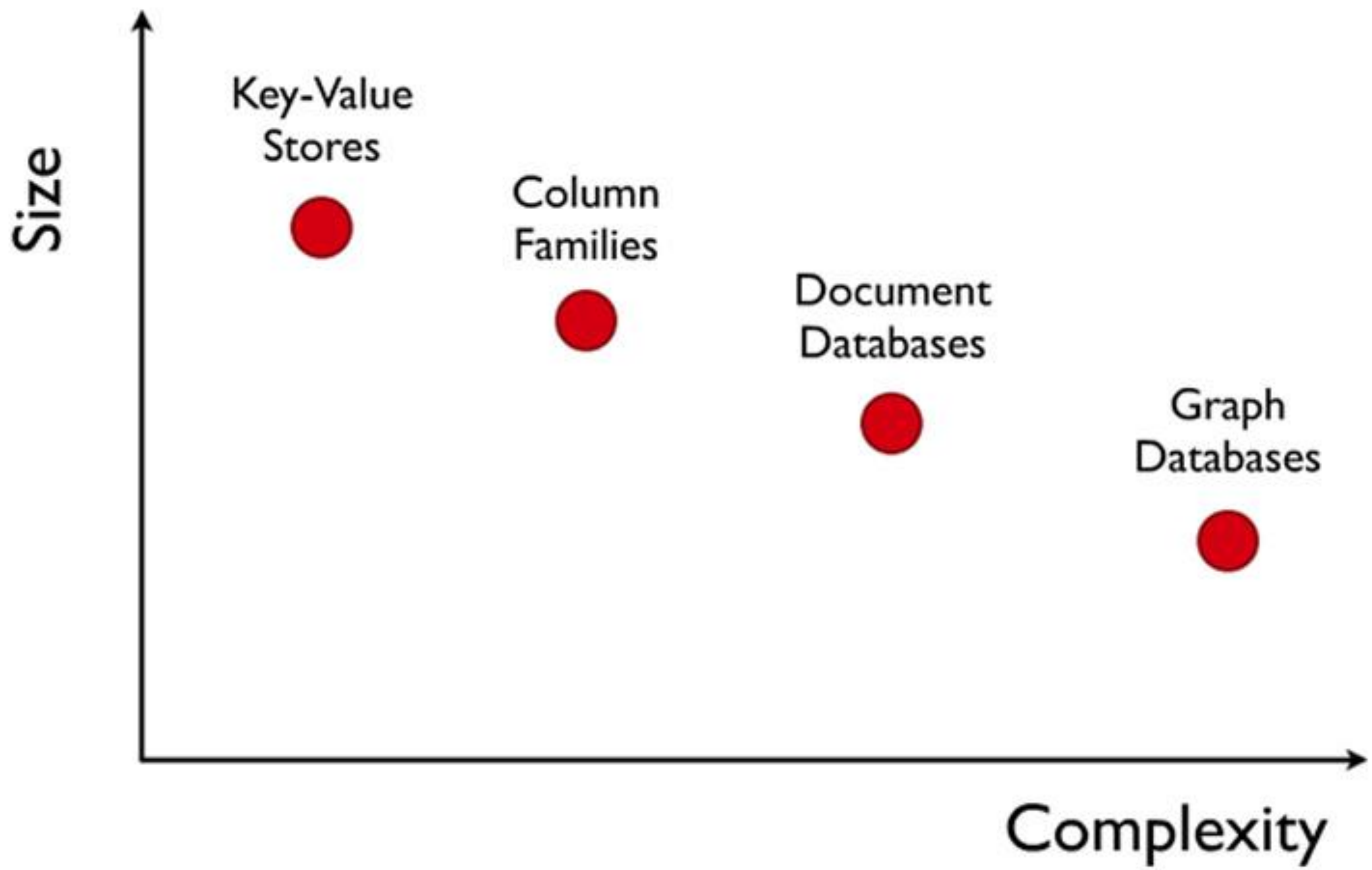
NoSQL Taxonomy



NoSQL Taxonomy - example data stores

Type	Examples
Document store	CouchDB relax, mongoDB
Column store	Cassandra, HBASE
Key-value store	redis, riak
Graph store	InfiniteGraph, Neo4j

Complexity vs size



Key-Value store

Type	Examples
Document store	CouchDB relax, mongoDB
Column store	Cassandra, HBASE
Key-value store	redis, riak
Graph store	InfiniteGraph, Neo4j

Key-Value stores

- Very simple interface
  - Data model: (key, value) pairs
  - Operations:
    - **put**(key, value)
    - value = **get**(key)
- Implementation: efficiency, scalability, fault-tolerance
  - Records distributed to nodes based on key
  - Replication: scalability and fault-tolerance
- Examples
  - Redis, Memcached, Riak

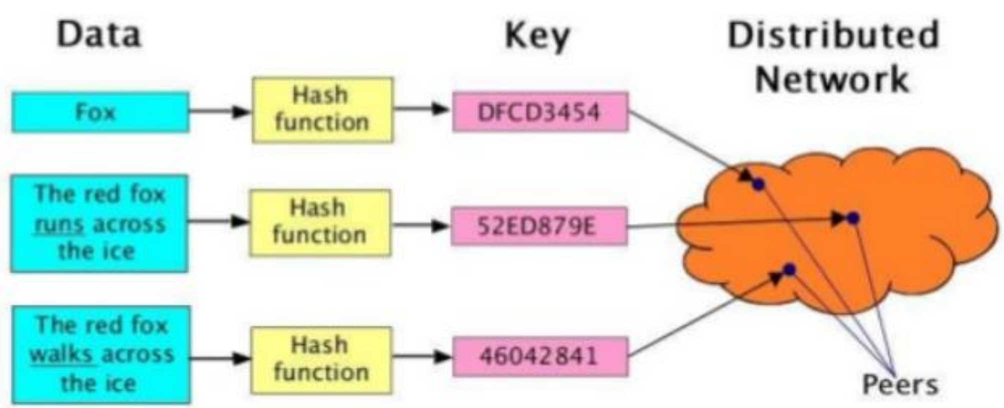
Redis

- History
  - Started in early 2009 - Salvatore Sanfilippo, an Italian developer
  - He was working on a real-time web analytics solution and realized that MySQL could **not** provide necessary performance
- Distributed data structure server
- Simple API
- Automatic data partitioning across multiple nodes

redis

Distributed data structure

- Distributed hash table (DHT)
  - Decentralized hash lookup service
  - (key, value) pairs are stored in DHT and any participating node can retrieve the value given a key
  - The key-space is spread across many buckets on the network
  - Each bucket is replicated (for fault-tolerance)

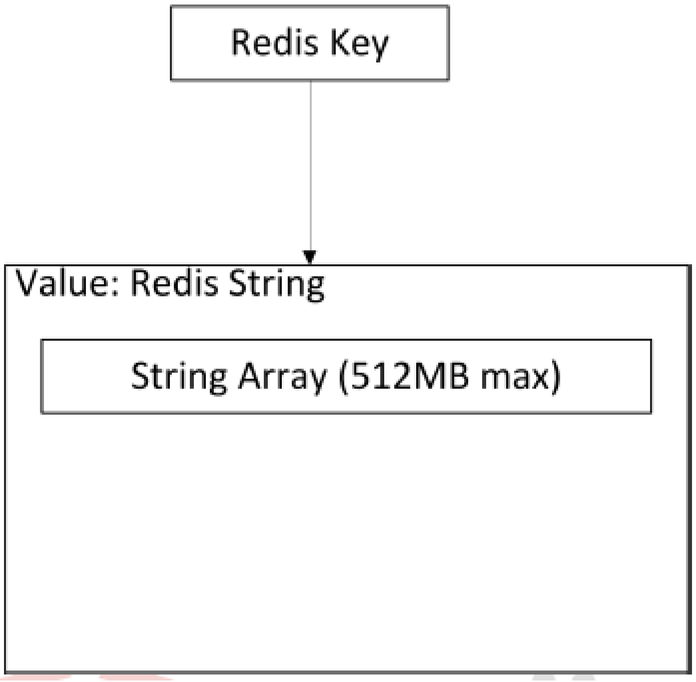


Logical data model

- Key
  - Printable ASCII
- Value
  - Primitives
    - Strings
  - Containers (of strings)
    - Hashes
    - Lists
    - Sets
    - Sorted Sets

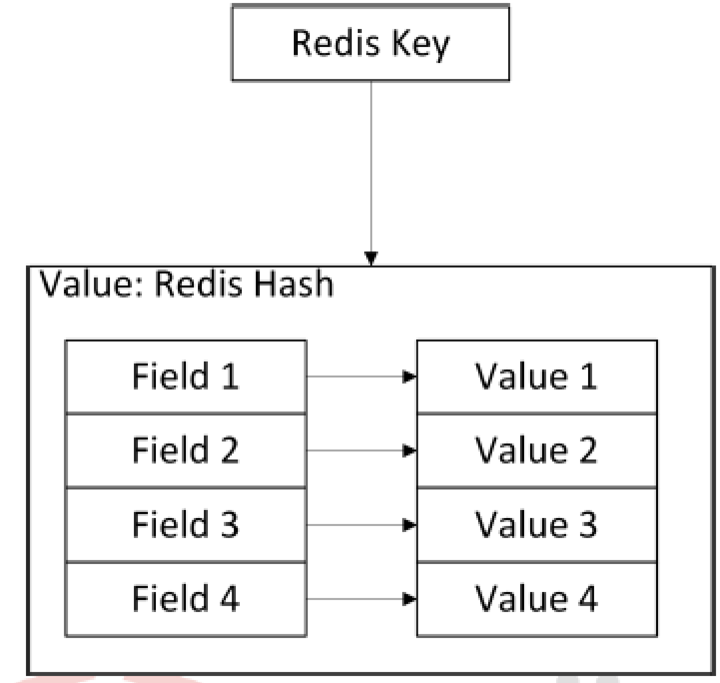
Logical data model

- Key
  - Printable ASCII
- Value
  - Primitives
    - **Strings**
  - Containers (of strings)
    - Hashes
    - Lists
    - Sets
    - Sorted Sets



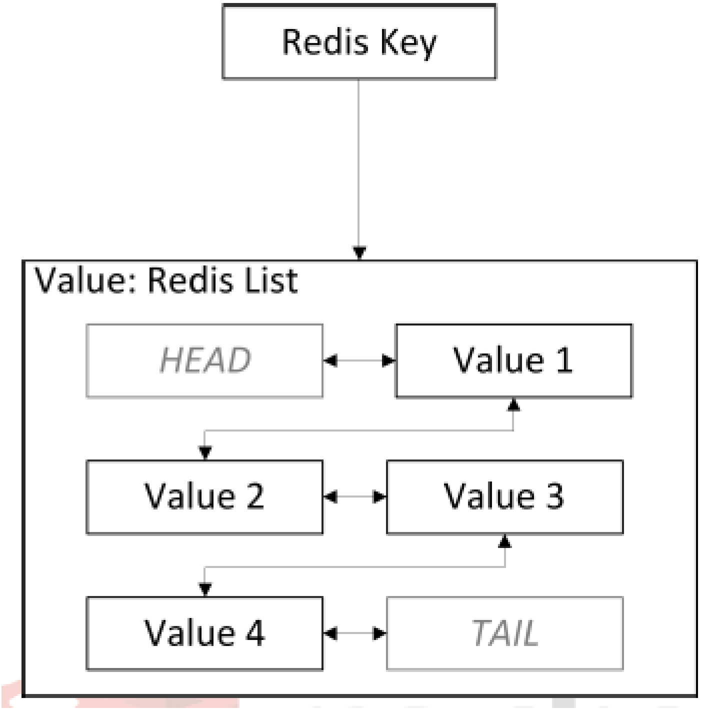
Logical data model

- Key
  - Printable ASCII
- Value
  - Primitives
    - Strings
  - Containers (of strings)
    - **Hashes**
    - Lists
    - Sets
    - Sorted Sets



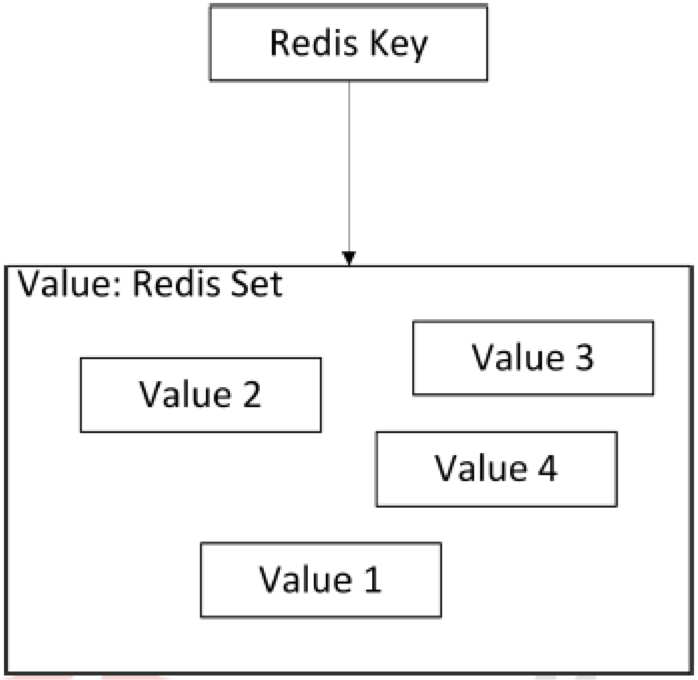
Logical data model

- Key
  - Printable ASCII
- Value
  - Primitives
    - Strings
  - Containers (of strings)
    - Hashes
    - **Lists**
    - Sets
    - Sorted Sets



Logical data model

- Key
  - Printable ASCII
- Value
  - Primitives
    - Strings
  - Containers (of strings)
    - Hashes
    - Lists
    - **Sets**
    - Sorted Sets



Redis-cli

- API: primitive
  - SET foo bar
  - GET foo

  => bar
- API: list
  - LPUSH mylist a // now mylist holds 'a‘
  - LPUSH mylist b // now mylist holds 'b','a'
  - LPUSH mylist c // now mylist holds 'c','b','a’

  - LRANGE mylist 0 1

  => c,b

Redis-cli

- API: hash
  - HMSET *myuser* **name** Salvatore **surname** Filippo **country** Italy
  - HGET *myuser* surname

  $\Rightarrow$ Filippo

- API: set
  - SADD myset a
  - SADD myset b
  - SADD myset foo
  - SADD myset bar
  - SMEMBERS myset

  => bar,a,foo,b

Column stores

Type	Examples
Document store	CouchDB relax, mongoDB
Column store	Cassandra, HBASE
Key-value store	redis, riak
Graph store	InfiniteGraph, Neo4j

Column *family* store

- Not to be confused with the relational-db version of it
  - Sybase-IQ, etc.
- Multi-dimensional map
- Sparsely populated table whose rows can contain arbitrary columns → Column families
- Examples
  - Cassandra
  - Hbase
  - Amazon SimpleDB

Some statistics

- Facebook Search
- MySQL > 50 GB Data
  - Writes Average : ~300 ms
  - Reads Average : ~350 ms
- Rewritten with Cassandra > 50 GB Data
  - Writes Average : 0.12 ms
  - Reads Average : 15 ms

Document stores

Type	Examples
Document store	CouchDB relax, mongoDB
Column store	Cassandra, HBASE
Key-value store	redis, riak
Graph store	InfiniteGraph, Neo4j

Document store

- Key-document store
  - the document can be seen as a value so you can consider this is a super-set of key-value
- Big difference with key-value store
  - that in document stores ***one can query also on the document***, i.e. the document portion is structured (not just a blob of data)
- Examples
  - **MongoDB**
  - CouchDB

MongoDB

- A document-oriented database
  - documents encapsulate and encode data
- Uses BSON/JSON format
- Schema-less
  - No more configuring database columns with types
- No transactions
- No joins

MongoDB basics

- A MongoDB instance may have zero or more databases
- A database may have zero or more collections
  - Can be thought of as the relation (table) in RDBMS, but with differences
- A collection may have zero or more documents
  - Docs in the same collection don't even need to have the same fields
  - Docs are the records in RDBMS
  - Docs can embed other documents
  - Documents are addressed in the database via a unique key
- A document may have one or more fields
- MongoDB Indexes is much like their RDBMS counterparts

RDBMS vs MongoDB

RDBMS	MongoDB
Database	Database
Table, View	Collection
Row	Document (JSON, BSON)
Column	Field

RDBMS vs MongoDB

RDBMS	MongoDB
Database	Database
Table, View	Collection
Row	Document (JSON, BSON)
Column	Field

JSON is a human-readable format

BSON (Binary Structured Object Notation) is a serialization encoding format for **JSON** used for storing and accessing documents

Example JSON document

```
{
  "_id": ObjectId("5114e0bd42…"),
  "first": "John",
  "last": "Doe",
  "age": 39,
  "interests": ["Mountain Biking”]
}
```

Collection example

```
{
    "_id": ObjectId("5114e0bd42…"),
    "first": "John",
    "last": "Doe",
    "age": 39,
    "interests": ["Mountain Biking “]
},
{
    "_id": ObjectId(“4a14e0f361…"),
    "first": “Caroline",
    "last": “Smith",
    "age": 32,
    "interests": ["Reading", “Yoga”]
}
```

Obligatory, and automatically generated by MongoDB

DB Operations

- Inserting a record

```
> db.comedy.insert({name:"Wayne's World", year:1992})
> db.comedy.insert({name:'The School of Rock', year:2003})
```

- Query (the whole collection)

```
> db.comedy.find()
{ "_id" : ObjectId("4e9ebb318c02b838880ef412"), "name" : "Bill & Ted's Exc
{ "_id" : ObjectId("4e9ebb478c02b838880ef413"), "name" : "Wayne's World",
{ "_id" : ObjectId("4e9ebd5d8c02b838880ef414"), "name" : "The School of Ro
```

- Query (all titles released earlier than 1994)

```
> db.comedy.find({year:{$lt:1994}})
{ "_id" : ObjectId("4e9ebb318c02b838880ef412"), "name" : "Bill & Ted's Exc
{ "_id" : ObjectId("4e9ebb478c02b838880ef413"), "name" : "Wayne's World",
```

Graph stores

Type	Examples
Document store	CouchDB relax mongoDB
Column store	Cassandra HBASE
Key-value store	redis riak
Graph store	InfiniteGraph Neo4j

Graph store

- Based on Graph Theory
- Scale vertically
- You can use graph algorithms easily
- Example, Neo4j

Relational vs. Graph: data model

Finding friends

Person	
ID	**Person**
1	Alice
2	Bob
...	...
99	Zach

PersonFriend	
PersonID	**FriendID**
1	2
2	1
2	99
...	...
99	1

Relational vs. Graph: data model

Finding friends

- *Bob’s friends*

```
SELECT p1.Person
FROM Person p1

JOIN PersonFriend
ON PersonFriend.FriendID = p1.ID

JOIN Person p2
ON PersonFriend.PersonID = p2.ID

WHERE p2.Person = 'Bob'
```

Relational vs. Graph: data model

Finding friends

- *Bob’s friends-of-friends*

```
SELECT p1.Person AS PERSON, p2.Person AS
FRIEND_OF_FRIEND
FROM PersonFriend pf1

JOIN Person p1
ON pf1.PersonID = p1.ID

JOIN PersonFriend pf2
ON pf2.PersonID = pf1.FriendID

JOIN Person p2
ON pf2.FriendID = p2.ID

WHERE p1.Person = ‘Bob' AND pf2.FriendID <> p1.ID
```

Relational vs. Graph: data model

Finding friends

- *Bob's friends-of-friends-of-….*

SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND

Join complexity increases with each additional depth

ON pf2.PersonID = pf1.FriendID

JOIN Person p2
ON pf2.FriendID = p2.ID

WHERE p1.Person = 'Bob' AND pf2.FriendID <> p1.ID

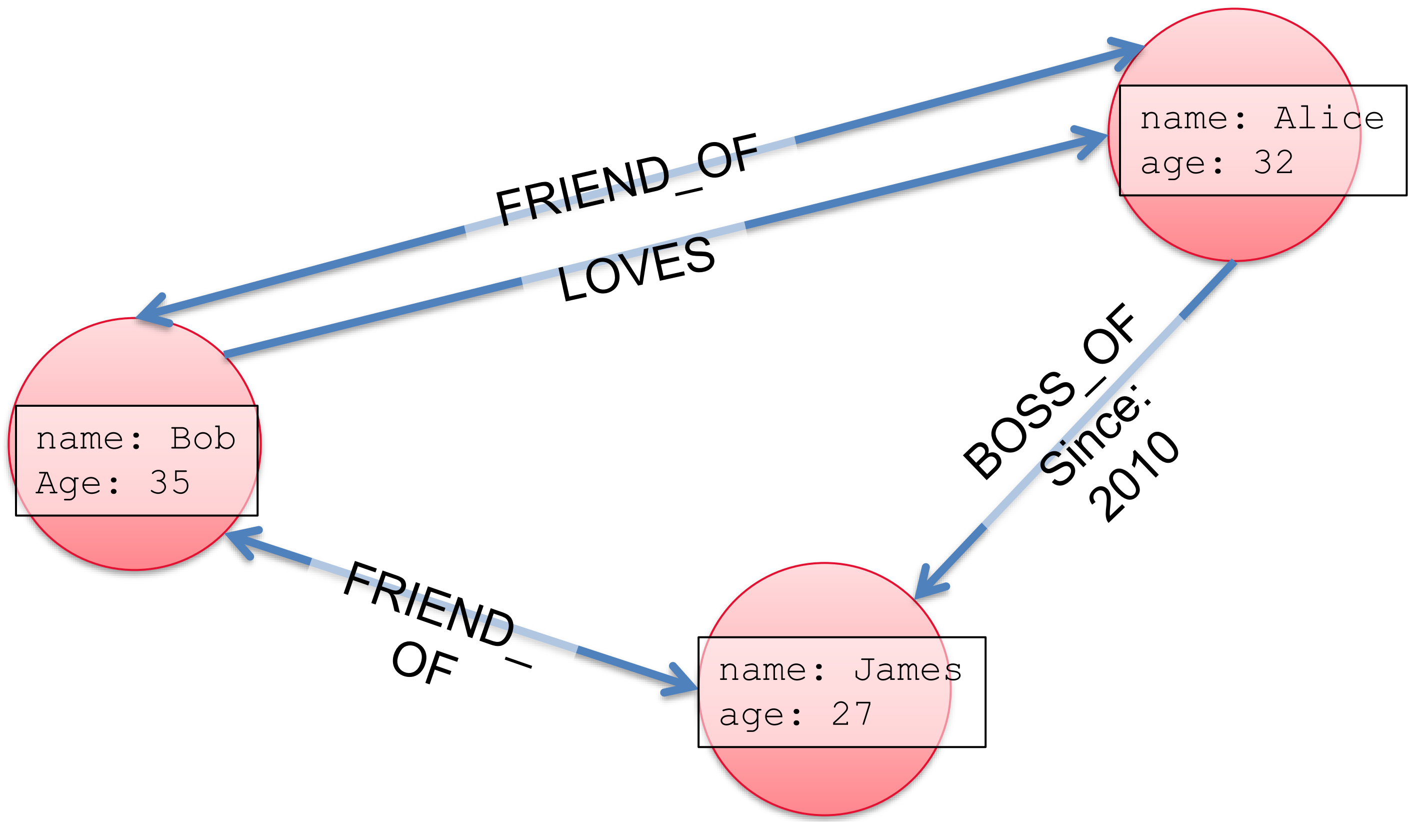
Relational model and connected data

- Relational model deals with connected data by means of join
- Join tables add **complexity**; they mix business data with foreign key metadata
- Foreign key constraints add additional development and maintenance overhead *just to make the database work*
- Things get more complex and more expensive the deeper we go into the network

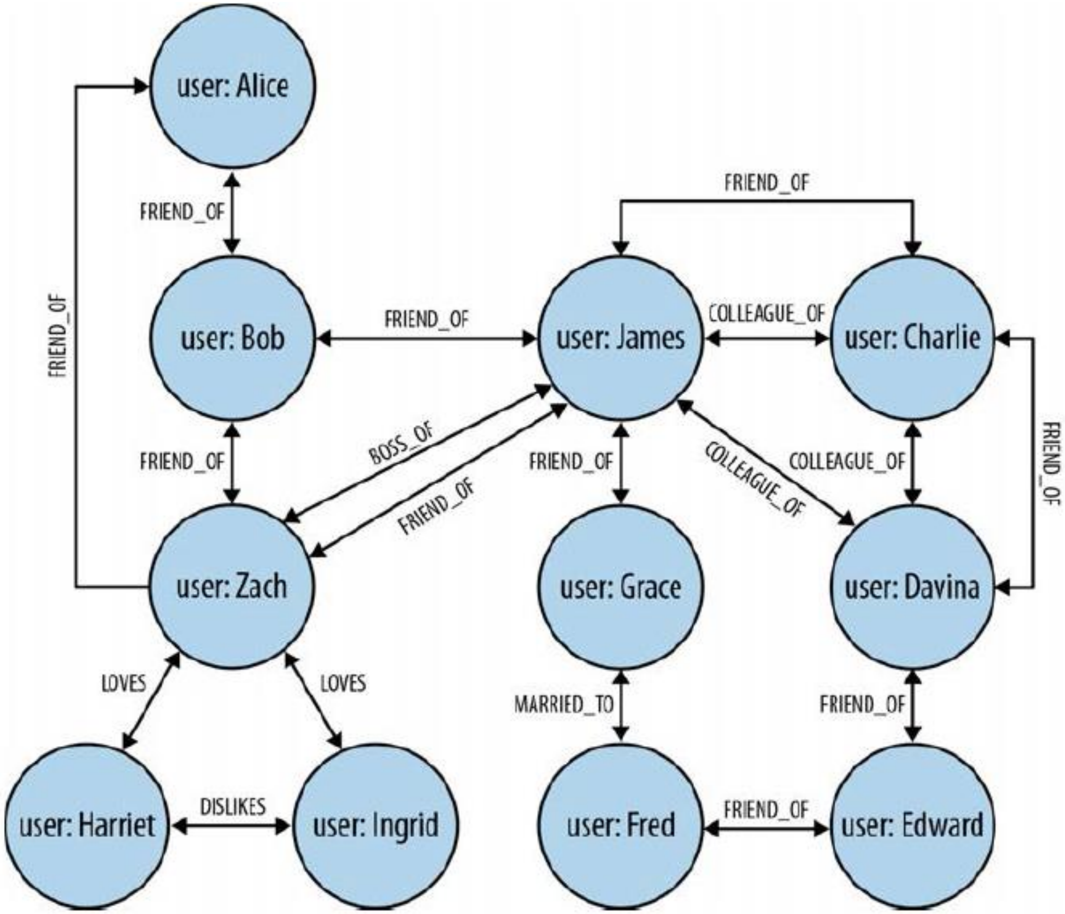
Enter, property graph model...

- Node
  - contain properties
- Relationship
  - connect nodes
  - a start node and an end node
  - always has a direction
  - a label
- Properties
  - keys are strings and the values are arbitrary data types

Property graph model



Finding relations is easy!



Advantages of property graph model

- Flexibility
  - Allow us to add new nodes and new relationships without compromising the existing network or migrating data
  - Original data and its intent remain intact
- Expressive power
  - We can see who LOVES whom (and whether that love is requited!)
  - We can see who’s MARRIED_TO someone else
  - We can see who is a COLLEAGUE_OF of whom and who is BOSS_OF them all
- Performance

Relational vs. Graph: performance

- Finding friends-of-friends in a social network
  - Maximum depth 5
  - 1 million people, each with approximately 50 friends

Depth	RDBMS execution time (s)	Neo4j execution time (s)	Records returned
2	0.016	0.01	~2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

Cypher: graph query language of NEO4J

- Declarative graph pattern matching language
  - “SQL for graphs”
  - Tabular results
- Cypher is evolving steadily
  - Syntax changes between releases
- Supports queries
  - Including aggregation, ordering and limits
  - Mutating operations in product roadmap

Two nodes, one relationship

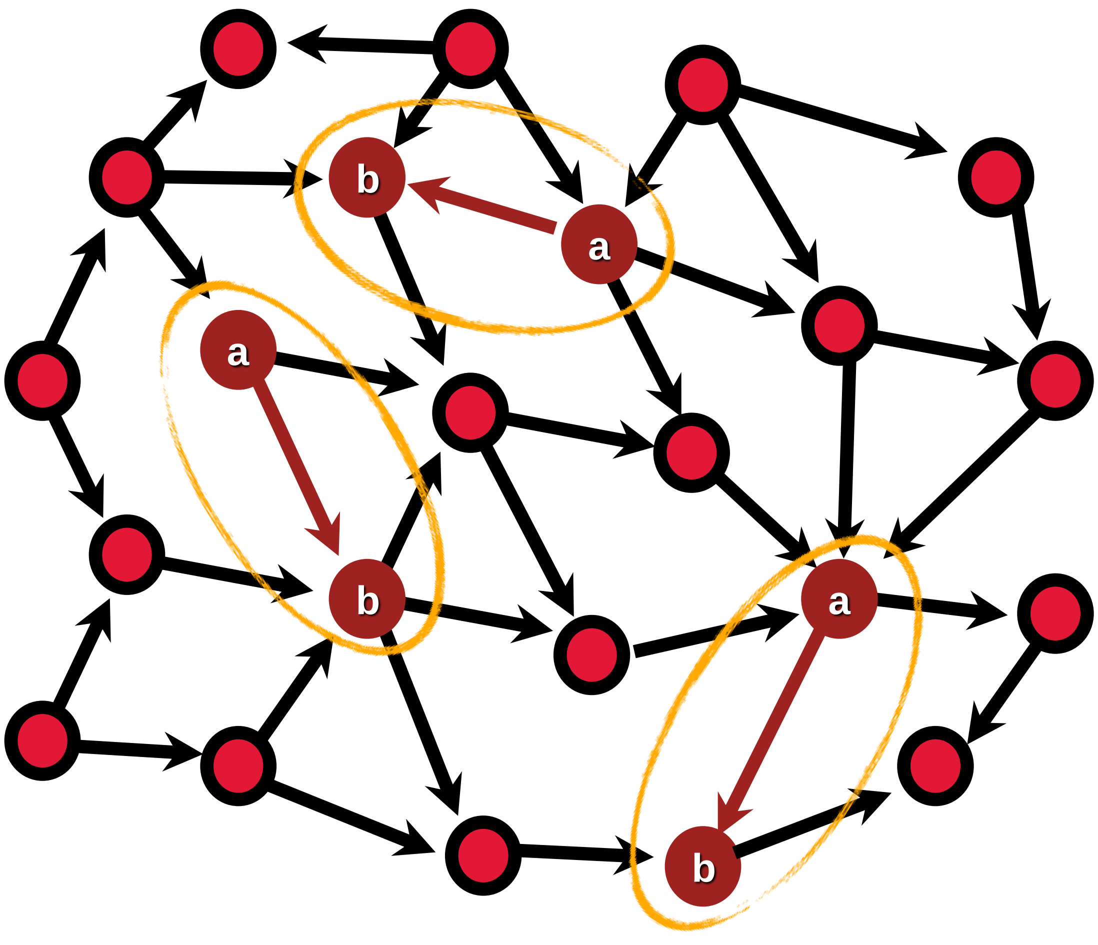
(a) --> (b)

Two nodes, one relationship


```
START a=node(*)
MATCH (a)-->(b)
RETURN a, b;
```

Pattern matching

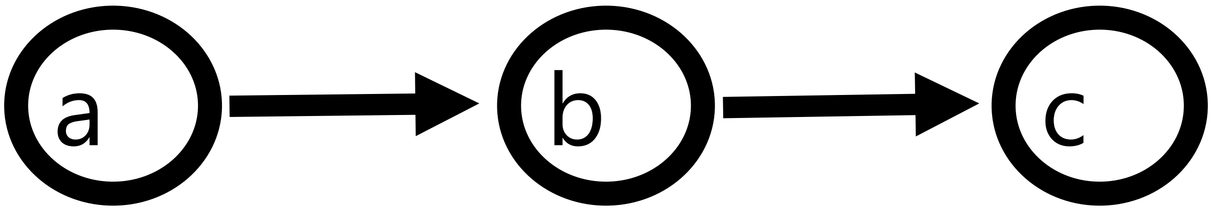
START a=node(*)
MATCH (a)-->(b)
RETURN a, b;



Two nodes, one relationship

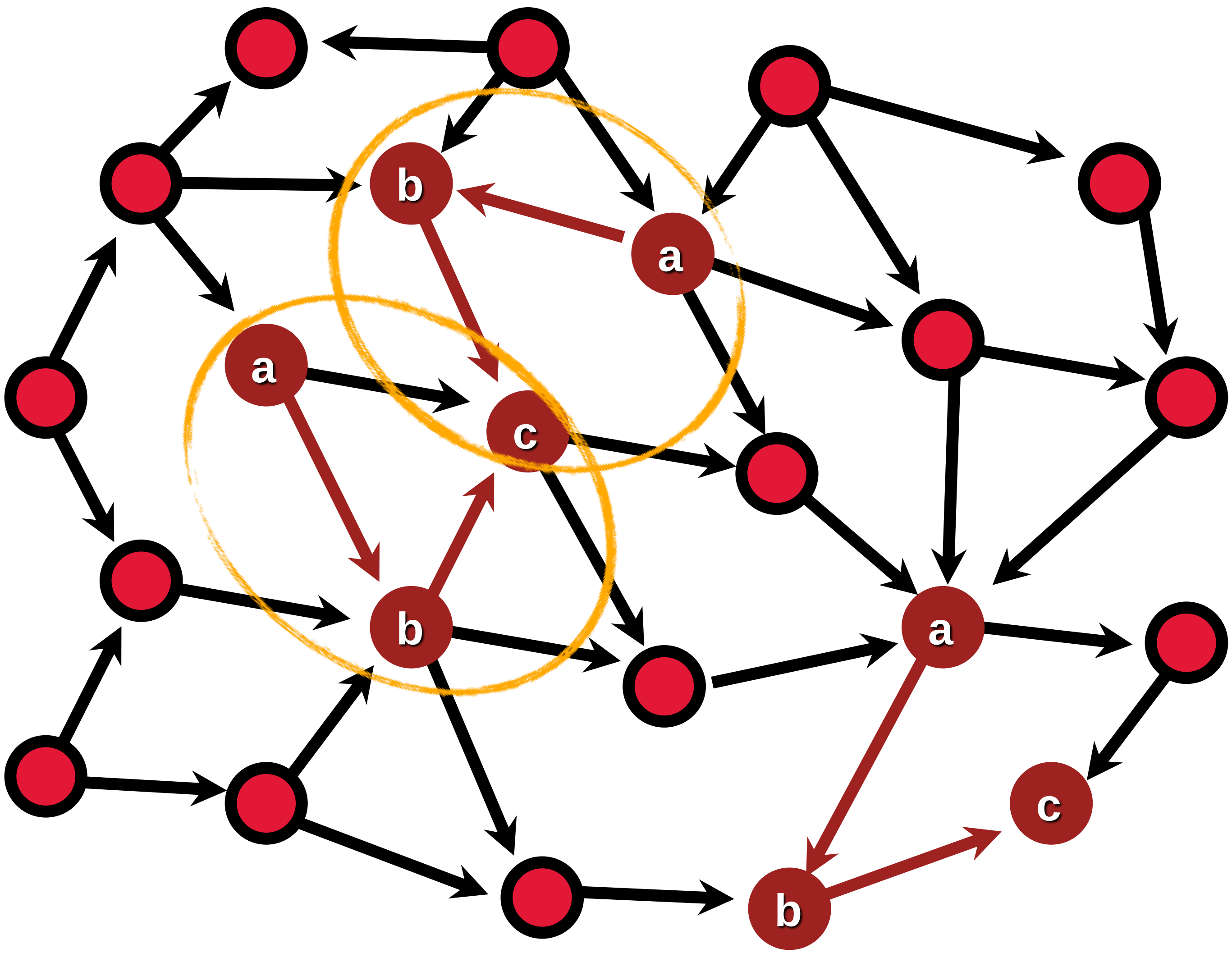
```
START a=node(*)
MATCH (a)-[:ACTED_IN]->(m)
RETURN a.name, r.roles, m.title;
```

Paths



(a)-->(b)-->(c)

Pattern matching



Sort & Limit

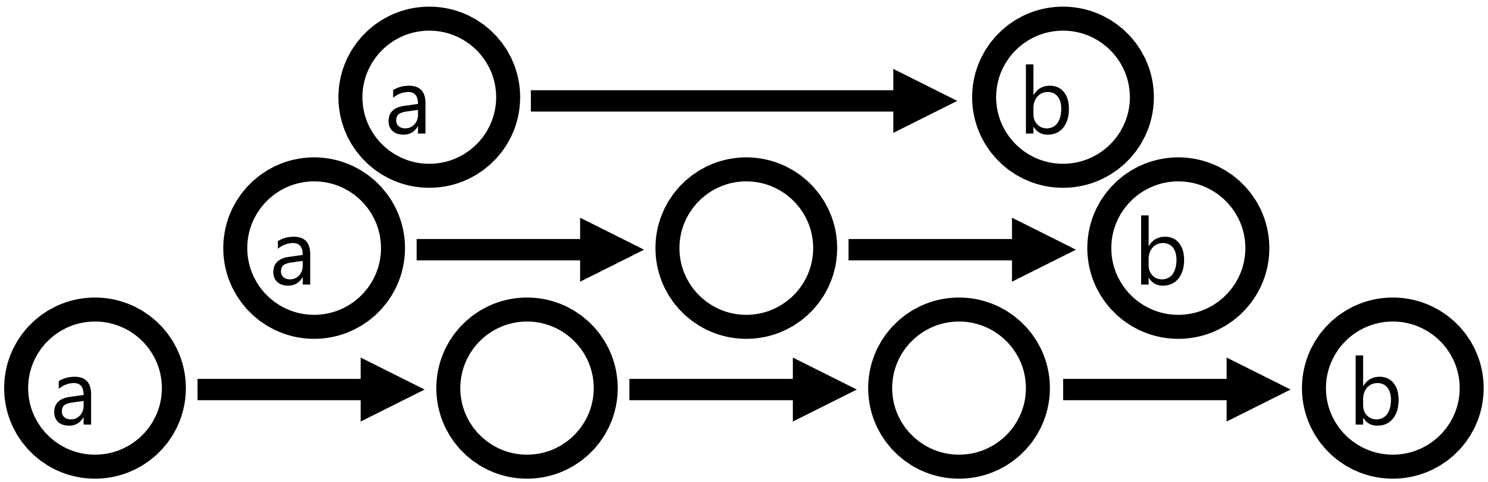
```
START a=node(*)
MATCH (a)-[:ACTED_IN]->(m)<-[:DIRECTED]-(d)
RETURN a.name, d.name, count(*) AS count
ORDER BY(count) DESC
LIMIT 5;
```

Constraints on properties

```
START tom=node:node_auto_index(name="Tom Hanks")
MATCH (tom)-[:ACTED_IN]->(movie)
WHERE movie.released < 1992
RETURN DISTINCT movie.title;
```

(Movies in which Tom Hanks acted, that were released before 1980)

Variable length paths



(a)-[*1..3]->(b)

Friends-of-Friends

```
START keanu=node:node_auto_index(name="Keanu Reeves")
MATCH (keanu)-[:KNOWS*2]->(fof)
RETURN DISTINCT fof.name;
```

NoSQL summary

- NoSQL databases reject:
  - Overhead of ACID transactions
  - “Complexity” of SQL
  - Burden of up-front schema design
- Programmer responsible for
  - Determining the consistency level
  - Navigating access path

Should I be using NoSQL Databases?

- NoSQL Data storage systems make sense for applications that need to deal with **very large semi-structured data**
  - log analysis
  - social networking feeds
- Most organizational databases are not that large and have low update/query rates
  - regular relational databases are the right solution for such environments

References

- I. Robinson, J. Webber, E. Eifrem. Graph Databases. O'Reilly, 2013
- Neo4J intro tutorial.
- NoSQL. Dr. Kristie Hawkey. Dalhousie University
- NoSQL. Perry Hoekstra. Perficient, Inc.
- NoSQL. Akmal Chaudhri
- Massively Parallel Cloud Data Storage Systems. S. Sudarshan. IIT Bombay
- NoSQL Theory, Implementations, an introduction. Firat Atagun
- http://www.datastax.com/docs/1.0/ddl/column_family
- http://redis.io/topics/twitter-clone
- REDIS. REmote DIctionary Server. Chris Keith and James Tavares
- Advanced Topics in Database Management. Stan Zdonik. Brown University
- An introduction to MongoDB. Rácz Gábor
- MongoDB. Mohamed Zahran. NYU
- Handling an 1,800 Percent Traffic Spike During Super Bowl XLVI. Jim Houska and Jim Houska