CS150: Database & Datamining

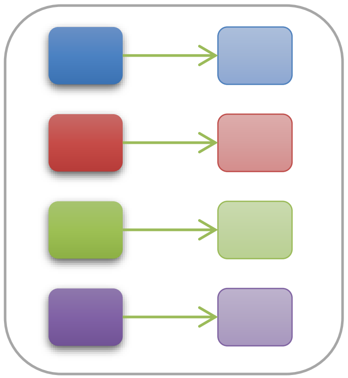
Lecture 30: NoSQL III

Xuming He

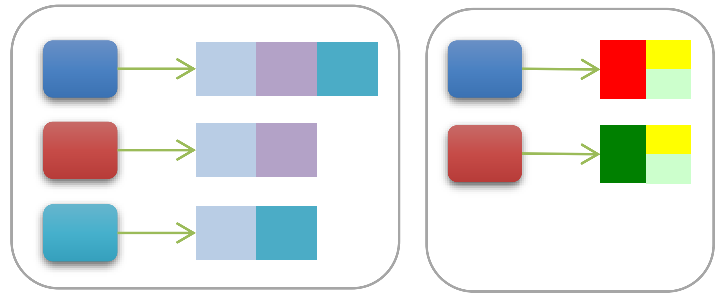
Spring 2019

Acknowledgement: Slides are adopted from the Berkeley course CS186 by Joey Gonzalez and Joe Hellerstein, Stanford CS145 by Peter Bailis, IIT Course 236363.

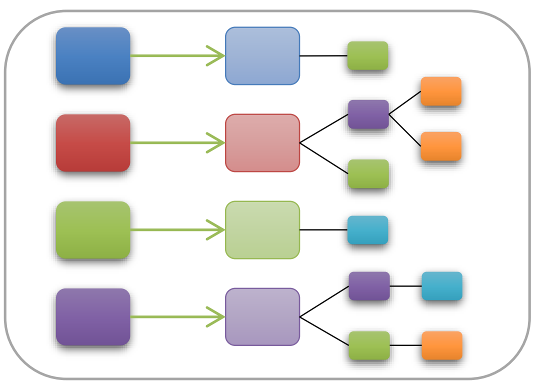
We Will Look at 4 Data Models



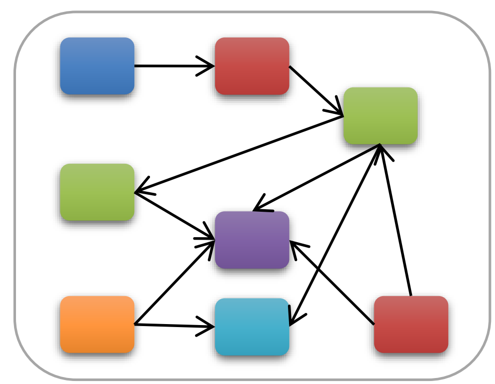
Key/Value Store



Column-Family Store



Document Store



Graph Databases

Document Databases: Fundamentals

- Basic concept of 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, …
- Documents in a collection should be “similar”
  - Their schema can differ
- Often: Documents stored as values of key-value
  - Key-value stores where the values are examinable
  - Building search indexes on various keys/fields

Why Document Databases

- XML and JSON are popular for data exchange
  - Recently mainly JSON
- Data stored in document DB can be used directly

- Databases often store objects from memory
  - Using RDBMS, we must do Object Relational Mapping (ORM)
    - ORM is relatively demanding
  - JSON is much closer to structure of memory objects
    - It was originally for JavaScript objects
    - Object Document Mapping (ODM) is faster

Document Databases: Representatives

mongoDB

Apache CouchDB relax

OrientDB

RAVENDB

terrastore

PostgreSQL

amazon DynamoDB

MS Azure DocumentDB

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

MongoDB

- Initial release: 2009
  - Written in C++
  - Open-source
  - Cross-platform
- JSON documents
- Basic features:
  - High performance – many indexes
  - High availability – replication + eventual consistency + automatic failover
  - Automatic scaling – automatic sharding across the cluster
  - MapReduce support

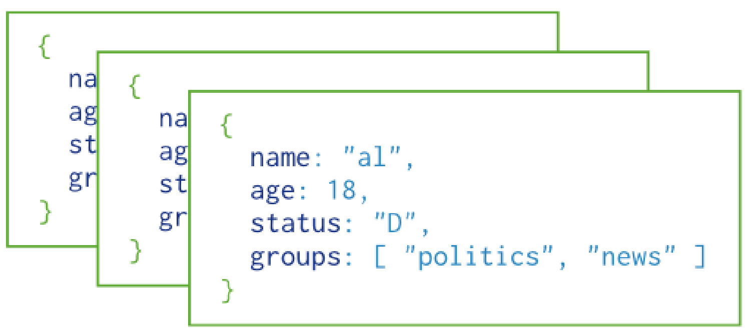
```
{
  name: "sue",                          field: value
  age: 26,                              field: value
  status: "A",                          field: value
  groups: [ "news", "sports" ]          field: value
}
```

http://www.mongodb.org/

MongoDB: Terminology

RDBMS	MongoDB
database instance	MongoDB instance
schema	database
table	collection
row	document
rowid	_id

- each JSON document:
  - belongs to a collection
  - has a field _id
    - unique within the collection
- each collection:
  - belongs to a “database”



Collection

http://www.mongodb.org/

Documents

- Use JSON for API communication
- Internally: BSON
  - Binary representation of JSON
  - For storage and inter-server communication

- Document has a maximum size: 16MB (in BSON)
  - Not to use too much RAM
  - GridFS tool can divide larger files into fragments

Document Fields

- Every document must have field _id
  - Used as a primary key
  - Unique within the collection
  - Immutable
  - Any type other than an array
  - Can be generated automatically
- Restrictions on field names:
  - The field names cannot start with the $ character
    - Reserved for operators
  - The field names cannot contain the . character
    - Reserved for accessing sub-fields

Database Schema

- Documents have flexible schema
  - Collections do not enforce specific data structure
  - In practice, documents in a collection are similar

- Key decision of data modeling:
  - References vs. embedded documents

  - In other words: Where to draw lines between aggregates
    - Structure of data
    - Relationships between data

Schema: Embedded Docs

mongoDB

- Related data in a single document structure
  - Documents can have subdocuments (in a field or array)

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
             level: 5,
             group: "dev"
           }
}
```

Embedded sub-document

Embedded sub-document

http://www.mongodb.org/

Schema: Embedded Docs (2)

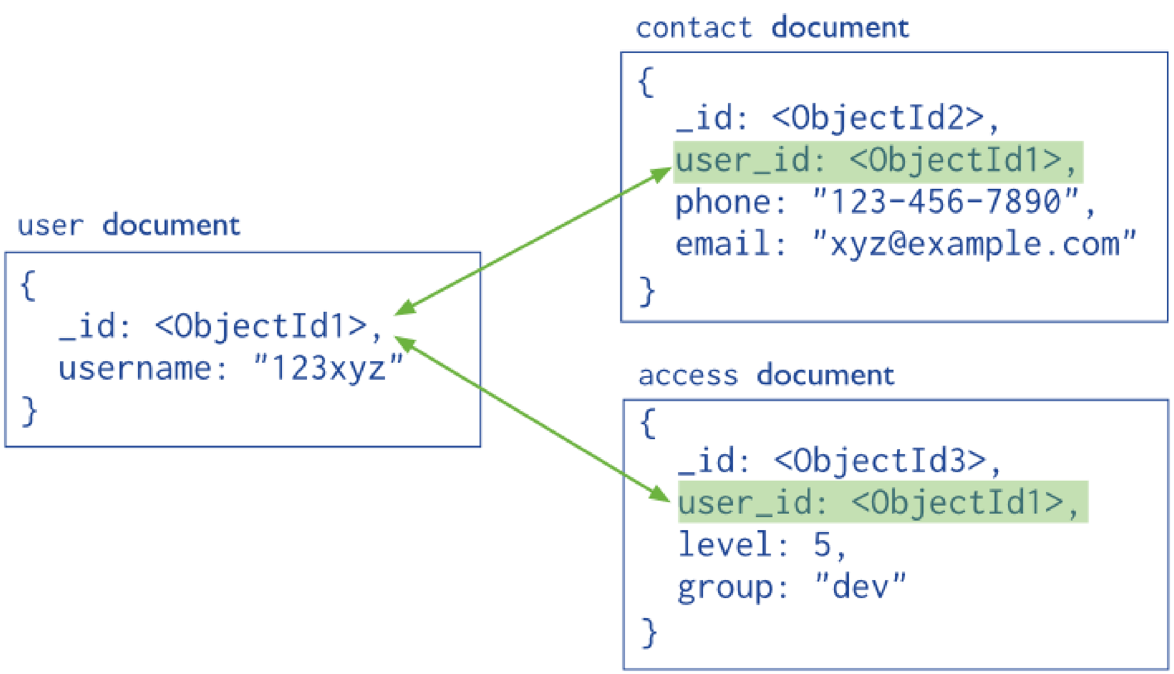
mongoDB

- Denormalized schema
- Main advantage:
  Manipulate related data in a single operation
- Use this schema when:
  - One-to-one relationships: one doc “contains” the other
  - One-to-many: if children docs have one parent document
- Disadvantages:
  - Documents may grow significantly during the time
  - Impacts both read/write performance
    - Document must be relocated on disk if its size exceeds allocated space
    - May lead to data fragmentation on the disk

Schema: References

mongoDB

- Links/references from one document to another
- Normalization of the schema



http://www.mongodb.org/

Schema: References (2)

- More flexibility than embedding
- Use references:
  - When embedding would result in duplication of data
    - and only insignificant boost of read performance
  - To represent more complex many-to-many relationships
  - To model large hierarchical data sets

- Disadvantages:
  - Can require more roundtrips to the server
    - Documents are accessed one by one

Querying: Basics

- Mongo query language
- A MongoDB query:
  - Targets a specific collection of documents
  - Specifies criteria that identify the returned documents
  - May include a projection to specify returned fields
  - May impose limits, sort, orders, ...

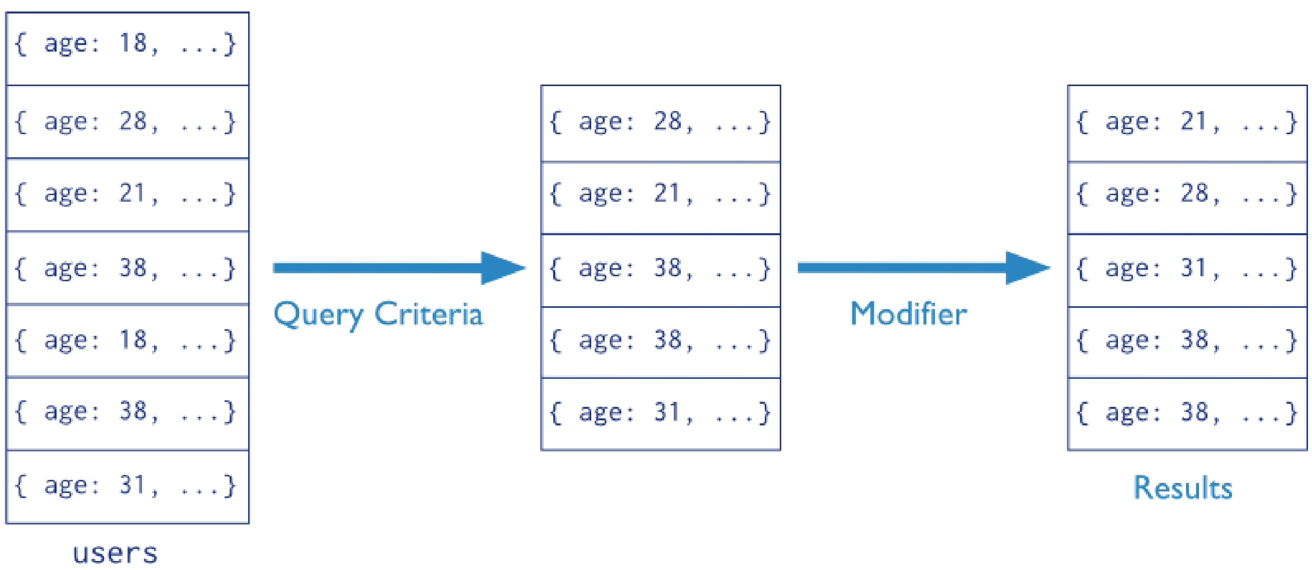
- Basic query - all documents in the collection:

```
db.users.find()
db.users.find( {} )
```

Querying: Example

Collection Query Criteria Modifier

```
db.users.find( { age: { $gt: 18 } } ).sort( {age: 1 } )
```



http://www.mongodb.org/

Querying: Selection

```
db.inventory.find({ type: "snacks" })
```

- All documents from collection inventory where the type field has the value snacks

```
db.inventory.find({ type: { $in: [ 'food',
'snacks' ] } } )
```

- All inventory docs where the type field is either food or snacks

```
db.inventory.find( { type: 'food', price: {
$lt: 9.95 } } )
```

- All ... where the type field is food and the price is less than 9.95

Inserts

```
db.inventory.insert( { _id: 10, type: "misc",
item: "card", qty: 15 } )
```

- Inserts a document with three fields into collection inventory
  - User-specified _id field

```
db.inventory.insert( { type: "book", item:
"journal" } )
```

- The database generates `_id` field

```
$ db.inventory.find()
{ "_id": ObjectId("58e209ecb3e168f1d3915300"),
type: "book", item: "journal" }
```

Updates

```
db.inventory.update(
    { type: "book", item : "journal" },
    { $set: { qty: 10 } },
    { upsert: true }   )
```

- Finds all docs matching query
  `{ type: "book", item : "journal" }`
- and sets the field `{ qty: 10 }`

- `upsert: true`
  - if no document in the inventory collection matches
  - creates a new document (generated _id)
    - it contains fields _id, type, item, qty

MapReduce

```
collection "accesses":
{
  "user_id": <ObjectId>,
  "login_time": <time_the_user_entered_the_system>,
  "logout_time": <time_the_user_left_the_system>,
  "access_type": <type_of_the_access>
}
```

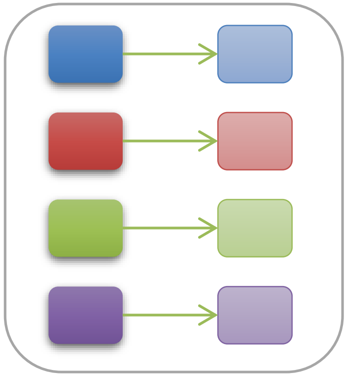
- How much time did each user spend logged in
  - Counting just accesses of type “regular”

```
db.accesses.mapReduce(
  function() { emit (this.user_id, this.logout_time - this.login_time); },
  function(key, values) { return Array.sum( values ); },
  {
    query: { access_type: "regular" },
    out: "access_times"
  }
)
```

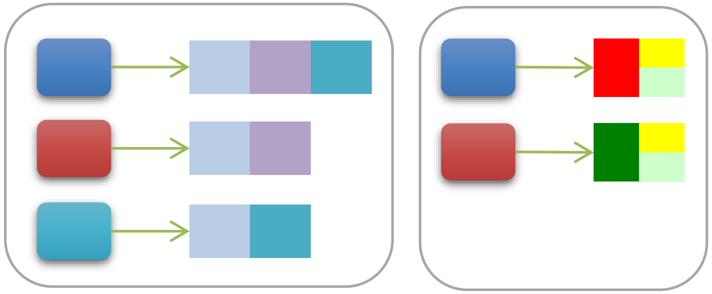
Document Stores Summary

- Similar in nature to key-value store, but value is tree structured as a *document*
- Motivation: avoid joins; ideally, all relevant joins already encapsulated in the document structure
- A document is an atomic object that cannot be split across servers
  - But a document *collection* will be split
- Moreover, transaction atomicity is typically guaranteed within a single document
- Model generalizes column-family and key-value stores

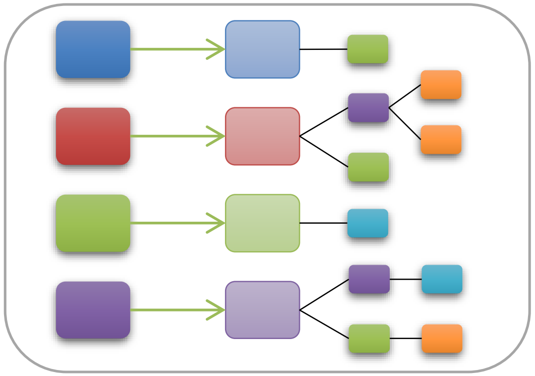
We Will Look at 4 Data Models



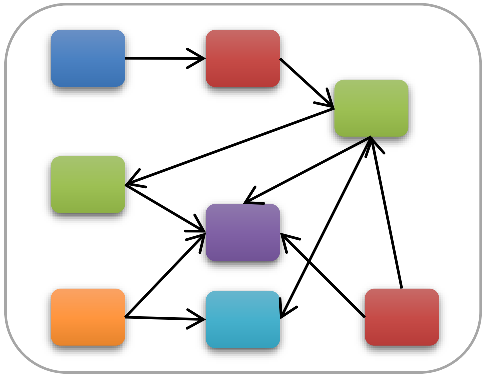
Key/Value Store



Column-Family Store



Document Store



Graph Databases

2 Types of Column Store

sid	name	address	year	faculty
861	Alma	Haifa	2	NULL
753	Amir	Jaffa	NULL	CS
955	Ahuva	NULL	2	IE

Standard RDB

id	sid
1	861
2	753
3	955

id	name
1	Alma
2	Amir
3	Ahuva

id	address
1	Haifa
2	Jaffa

id	year
1	2
3	2

id	faculty
2	CS
3	IE

Column Store: each column stored separately (still SQL)

Why? Efficiency (fetch only required columns), compression, sparse data for free

(Cassandra model)

keyspace

column family

timestamp for conflicts

1	sid:861 name:Alma address:Haifa ts:20
2	sid:753 name:Amir address:Jaffa ts:22
3	sid:955 name:Ahuva ts:32

"column"

column family

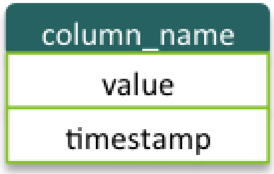
1	year:2 ts:26
2	faculty:CS ts:25 email:{prime:c@d ext:c@e}
3	year:2 faculty:IE ts:32 email:{prime:a@b ext:a@c}

"supercolumn"

Column-Family Store: NoSQL

Data Model: Column

- **Column = the** basic data item
  - a 3-tuple consisting of
    - column **name**
    - **value**
    - **timestamp**



  - Can be modeled as follows

```
{ name: "firstName",
  value: "Martin",
  timestamp: 12345667890 }
```

- In the following, we will ignore the timestamp

Data Model: Row

- **Row:** a collection of columns attached to row key
  - Columns can be added to any row at any time
    - without having to add it to other rows

```
// row
"martin-fowler" : {
    firstName: "Martin",
    lastName: "Fowler",
    location: "Boston"
 }
```

Row key1	Column Key1	Column Key2	Column Key3	...
	Column Value1	Column Value2	Column Value3	
⋮				

http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/

Data Model: Column Family

- **CF =** Set of columns containing “related” data

user_id (*row key*)	*column key*	*column key*	...
	column value	*column value*	...
1	login	first_name	...
	honza	Jan	...
4	login	age	...
	david	35	...
5	first_name	last_name	...
	Irena	Holubová	...
...			

I. Holubová, J. Kosek, K. Minařík, D. Novák. Big Data a NoSQL databáze. Praha: Grada Publishing, 2015. 288 p.

Data Model: Column Family (2)

- Column family - example as JSON

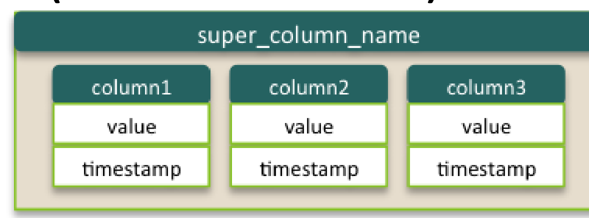
```
{ // row (columns from a CF)
  "pramod-sadalage" : {
    firstName: "Pramod",
    lastName: "Sadalage",
    lastVisit: "2012/12/12"
}
```

```
// row (cols from the same CF)
  "martin-fowler" : {
    firstName: "Martin",
    lastName: "Fowler",
    location: "Boston",
    activ: "true" }
}
```

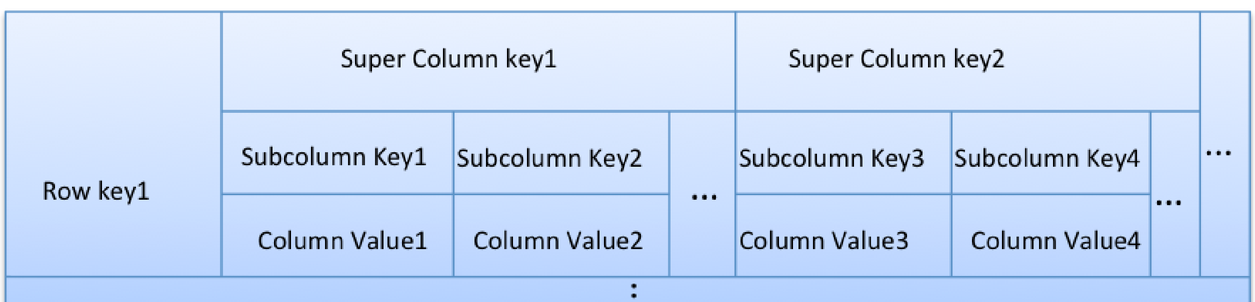
source: Sadalage & Fowler: NoSQL Distilled, 2012

Data Model: Super Column Family

- **Super column**
  - A column whose value is composed of a map of columns
  - Used in some column-family stores (Cassandra 1.0)



- **Super column family**
  - A column family consisting of super columns



http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/

Super Column Family: Example

user_id (*row key*)	*super column key*			*super column key*			*...*
	subcolumn key	*subcolumn key*	...	*subcolumn key*	*subcolumn key*	...	...
	subcolumn value	*subcolumn value*	...	*subcolumn value*	*subcolumn value*	...	
1	home_address			work_address			
	city	street	...	city	street	...	
	Brno	Krásná 5	...	Praha	Pracovní 13	...	
4	home_address			temporary_address			
	city	street	...	city	PSČ		
	Plzeň	sady Pětatřicátníků 35	...	Praha	111 00		
...							

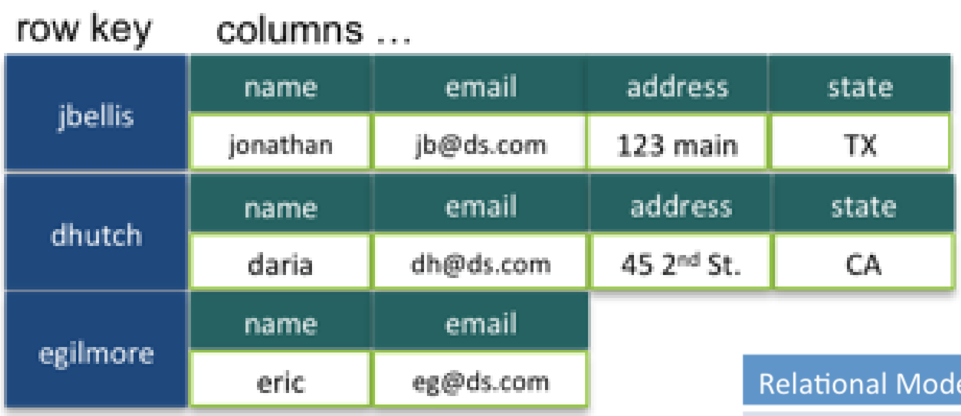
I. Holubová, J. Kosek, K. Minařík, D. Novák. Big Data a NoSQL databáze. Praha: Grada Publishing, 2015. 288 p.

Super Column Family: Example (2)

```
{ // row
  ”Cath”: {
    ”username”: { ”firstname”: ”Cath”, ”lastname”: ”Yoon” },
    ”address”: { ”city”: ”Seoul”, ”postcode”: ”1234” }
    }
  // row
  ”Terry”: {
    ”username”: { ”firstname”: ”Terry”, ”lastname”: ”Cho” },
    ”account”: { ”bank”: ”Hana”, ”accounted”: 1234 },
    ”preferences”: { ”color”: ”blue”, ”style”: ”simple” }
   }
 }
```

Data Model: Interpretation 1

1. Each column family = a relational table
    - with (a lot of) **null** values



Relational Model	Cassandra Model
Database	Keyspace
Table	Column Family (CF)
Primary key	Row key
Column name	Column name/key
Column value	Column value

http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/

Data Model: Interpretation 2

2. Column family = a map of maps (nested map)

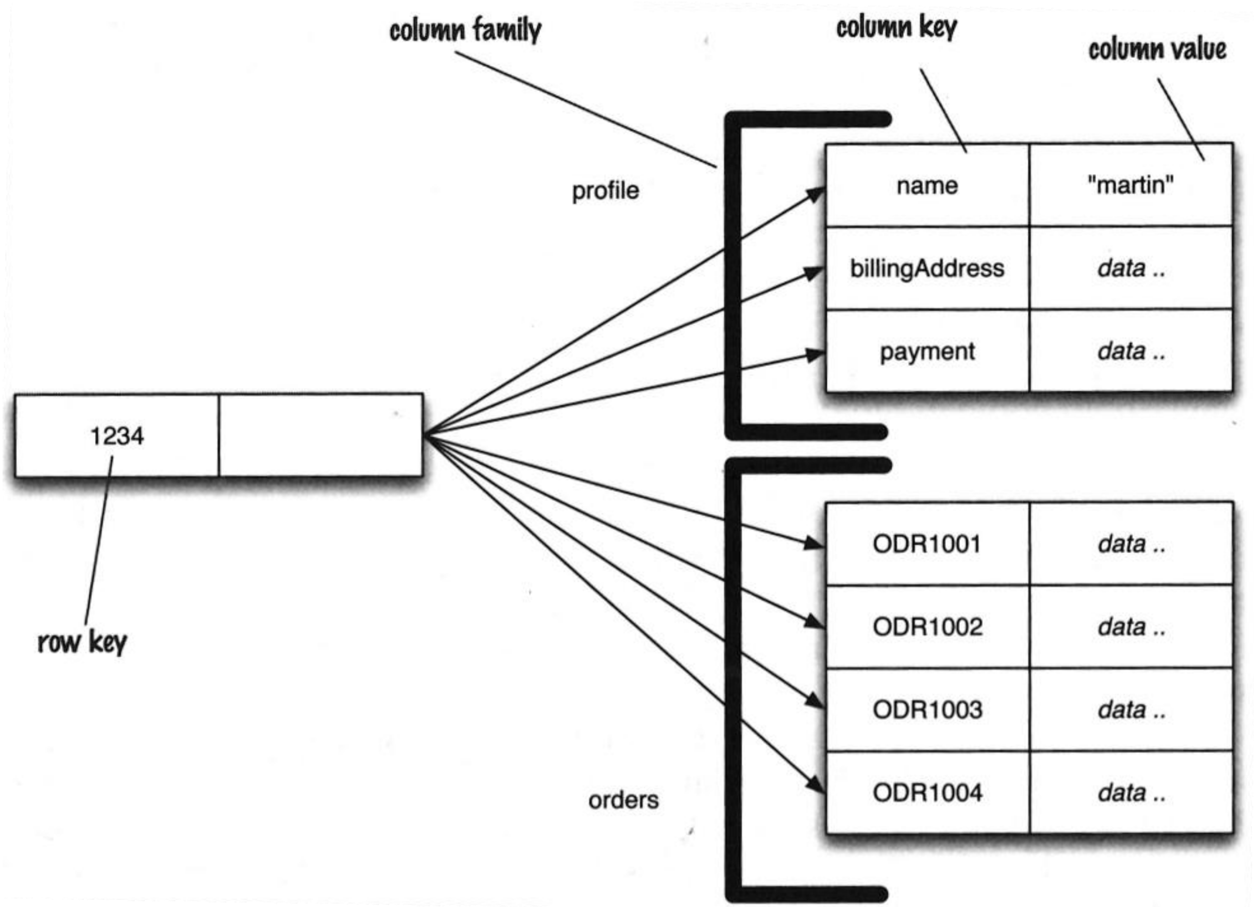
```
Map<RowKey, Map<ColumnKey, ColumnValue>>
```

- Super column family:

```
Map<RowKey, Map<SuperColumnKey,
                Map<ColumnKey, ColumnValue>>>
```

- The column-family data model can be viewed as *JSON-like documents with restrictions on the format*

Example: Visualization



source: Sadalage & Fowler: NoSQL Distilled, 2012

Column Family Stores: Features

- Data model: Column families
- System architecture
  - Data partitioning
- Local persistence
  - update log, memory, disk...
- Data replication
  - balancing of the data
- Query processing
  - query language
- Indexes

Representatives

Google BigTable

Cassandra

HBASE

HYPERTABLE

accumulo™

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

BigTable

- Google's paper:
  - Chang, F. et al. (2008). Bigtable: A Distributed Storage System for Structured Data. ACM TOCS, 26(2), pp 1–26.
- Proprietary, not distributed outside Google
  - used in Google Cloud Platform

- Data model: column families as defined above

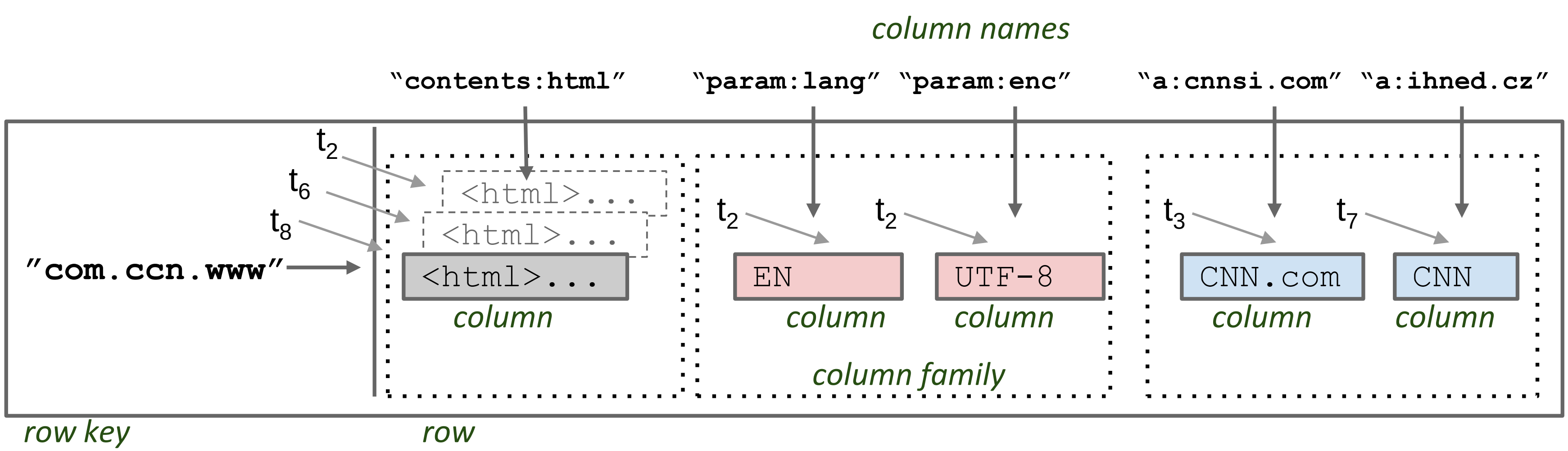
"A table in Bigtable is a sparse, distributed, persistent multidimensional sorted map. "

`(row:string, column:string, time:int64) → string`

http://en.wikipedia.org/wiki/BigTable

BigTable: Example

- “BigTable = sparse, distributed, persistent, multi-dimensional sorted map indexed by *(row_key, column_key, timestamp)*”



HBase

"Open source, non-relational, distributed database modeled after Google's BigTable. "

- Initial release: 2008
- Implementation: Java
  - Based on Apache Hadoop (HDFS)
- Open source: Apache Software License 2.0
- Systems: Linux, Unix, Windows (only via Cygwin)

"If you have hundreds of millions or billions of rows, then HBase is a good candidate. "

http://hbase.apache.org/

Cassandra

- Developed at Facebook
  - now, Apache Software License 2.0
- Initial release: 2008 (stable release: 2013)
- Written in: Java
- OS: cross-platform
- Operations:
  - CQL (Cassandra Query Language)
  - MapReduce support (can cooperate with Hadoop)
- Professional support by DataStax
  - http://www.datastax.com/

http://cassandra.apache.org/

Data Sharding in Cassandra

- Entries in each table are split by partition key
  - Which is a selected column (or a set of columns)
  - Specifically, the first column (or columns) from the primary key is the partition key of the table

```
CREATE TABLE tab ( a int, b text, c text, d text,
    PRIMARY KEY ( a, b, c)
);
```

```
CREATE TABLE tab ( a int, b text, c text, d text,
    PRIMARY KEY ( (a, b), c)
);
```

Data Sharding in Cassandra (2)

- All entries with the same partition key
  - Will be stored on the same physical node
  - => efficient processing of queries on one partition key

```
CREATE TABLE mytable (
    row_id int, column_name text, column_value text,
    PRIMARY KEY (row_id, column_name) );
```

- The rest of the columns in the primary key
  Are so called clustering columns
  - Rows are locally sorted by values in the clustering columns
    - the order for physical storing rows

Data Replication

- Cassandra adopts peer-to-peer replication
  - The same principles like in key-value stores & document DB
  - Read/Write quora to balance between availability and consistency guarantees

- HBase (and Google BigTable)
  - Physical data distribution & replication is done by the underlying distributed file system
  - HDFS, GFS

Local Persistence

- Organization of local data store at nodes
- Objectives:
  - Persistent, durable (ensure persistence after commit)
  - High performance of reads & writes

- Approach:
  - Memory tables
  - Append-only update log
  - SSTable disk-storage format: immutable
  - Compaction

Cassandra Query Language (CQL)

- The syntax of CQL is similar to SQL
  - But search just in one table (no joins)

```
SELECT <selectExpr>
FROM [<keyspace>.]<table>
[WHERE <clause>]
[ORDER BY <clustering_colname> [DESC]]
[LIMIT m];

SELECT column_name, column_value
FROM mytable
WHERE row_id=3
ORDER BY column_value;
```

CQL: Limitations on "Where" Part

```
CREATE TABLE mytable (
  row_id int,
  column_name text,
  column_value text,
  PRIMARY KEY
      (row_id, column_name)
);
```

- The search condition can be:
  - on columns in the partition key
    - And only using operators == and IN

```
... WHERE row_id IN (3, 4, 5)
```

    - Therefore, the query hits only one or several physical nodes (not all)

  - on columns from the clustering key
    - Especially, if there is also condition on the partitioning key

```
... WHERE row_id=3 AND column_name='login'
```

    - If it is not, the system must filter all entries

```
SELECT * FROM mytable
WHERE column_name IN ('login', 'name') ALLOW FILTERING;
```

CQL: Limitations on “Where” Part (1)

- Other columns can be queried
  - If there is an index built on the column

- Indexes can be built also on collection columns (set, list, map)
  - And then queried by `CONTAINS` like this

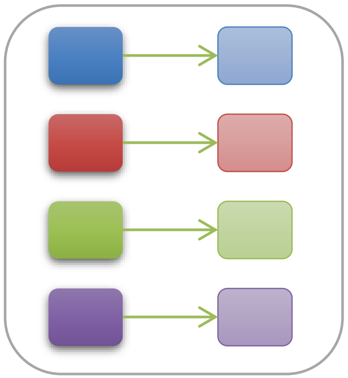
```
SELECT login FROM users
    WHERE emails CONTAINS 'jn@firma.cz';
SELECT * FROM users
    WHERE profile CONTAINS KEY 'colorschema';
```

Summary

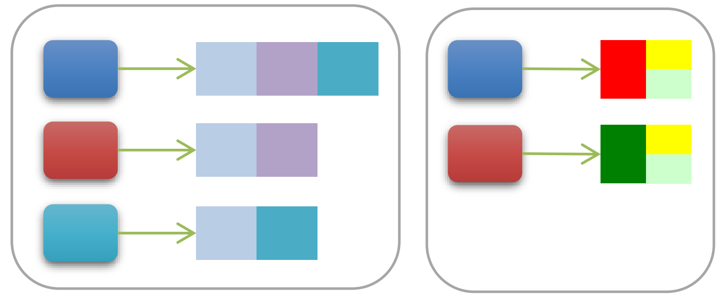
- Column-family stores
  - are worth only for large data and large query throughput
  - two ways to see the data model:
    - large sparse tables or multidimensional (nested) maps
  - data distribution is via row key
    - analogue of document ID or key in document or key-value stores
  - efficient disk + memory local data storage

- Cassandra
  - CQL: structured after SQL, easy transition from RDBMS

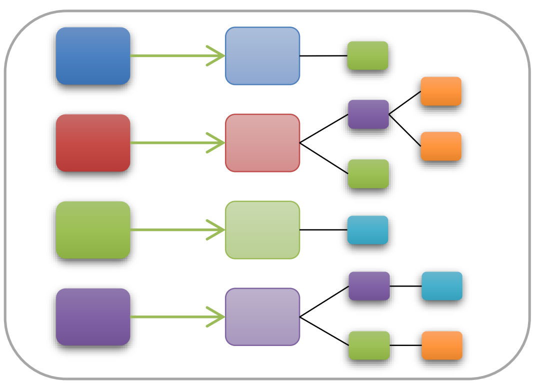
We Will Look at 4 Data Models



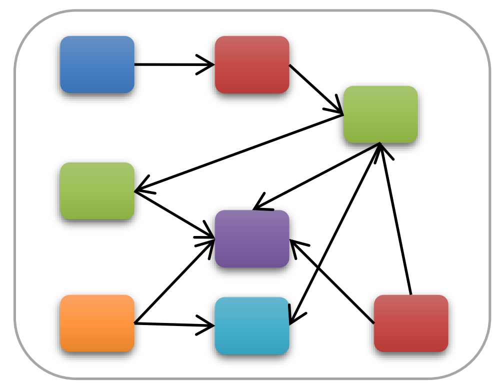
Key/Value Store



Column-Family Store

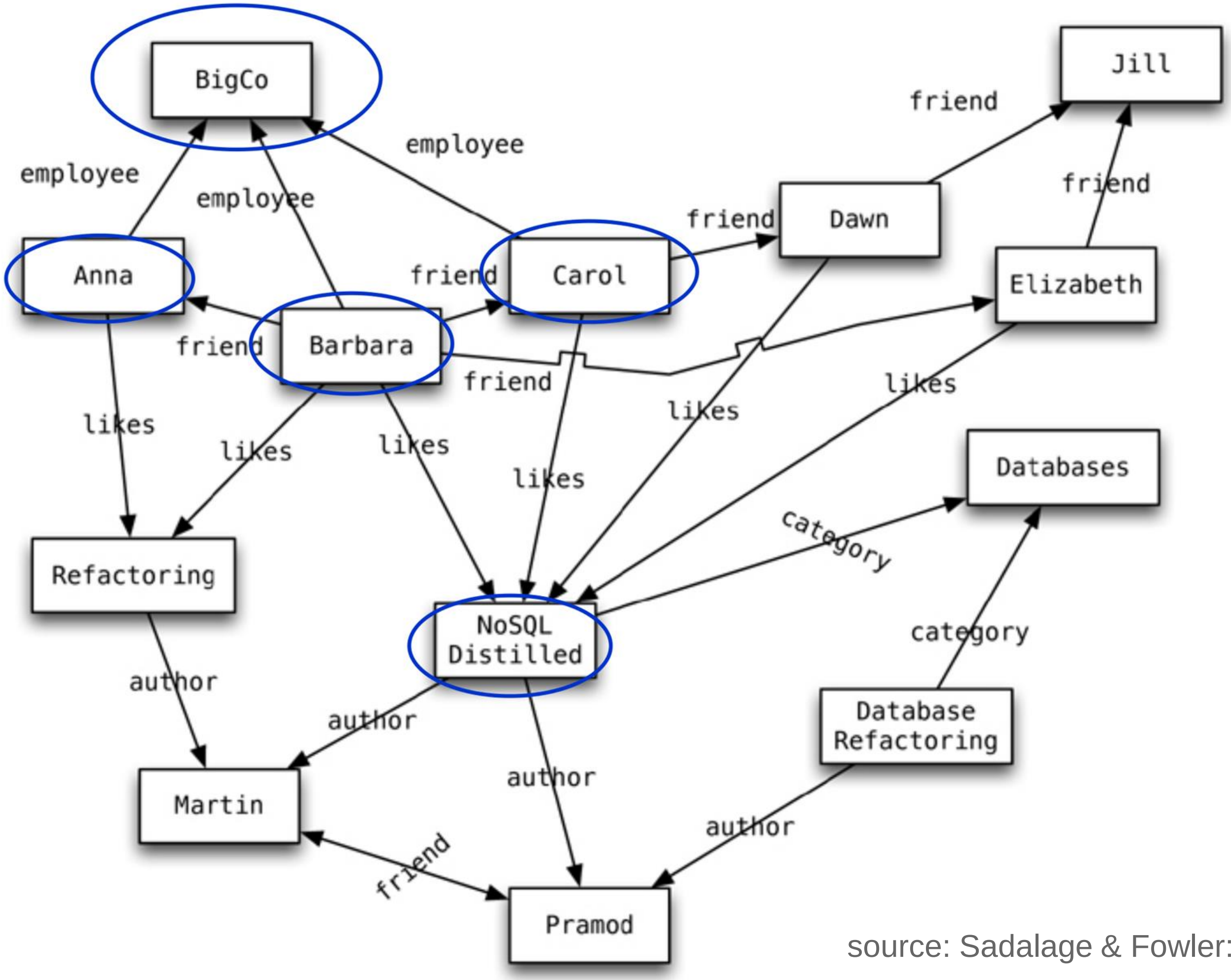


Document Store



Graph Databases

Graph Databases: Example



source: Sadalage & Fowler: NoSQL Distilled, 2012

Graph Databases: Mission

- To store entities and relationships between them
  - Nodes are instances of objects
  - Nodes have properties, e.g., name
  - Edges connect nodes and have directional significance
  - Edges have types e.g., likes, friend, …

- Nodes are organized by relationships
  - Allow to find interesting patterns
  - **example:** Get all nodes that are “employee” of “Big Company” and that “likes” “NoSQL Distilled”

Graph Databases: Representatives

Neo4j

OrientDB®

APACHE GIRAPH

TITAN

InfiniteGraph

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

Types of Graphs

- Single-relational graphs
  - Edges are homogeneous in meaning
    - e.g., all edges represent friendship
- Multi-relational (property) graphs
  - Edges are typed or labeled
    - e.g., friendship, business, communication
  - Vertices and edges maintain a set of key/value pairs
    - Representation of non-graphical data (properties)
    - e.g., name of a vertex, the weight of an edge

Graph Databases

- A graph database = a set of graphs

- Types of graph databases:
  - Transactional = large set of small graphs
    - e.g., chemical compounds, biological pathways, …
    - Searching for graphs that match the query
  - Non-transactional = few numbers of very large graphs
    - or one huge (not connected) graph
    - e.g., Web graph, social networks, …

Non-transactional Databases

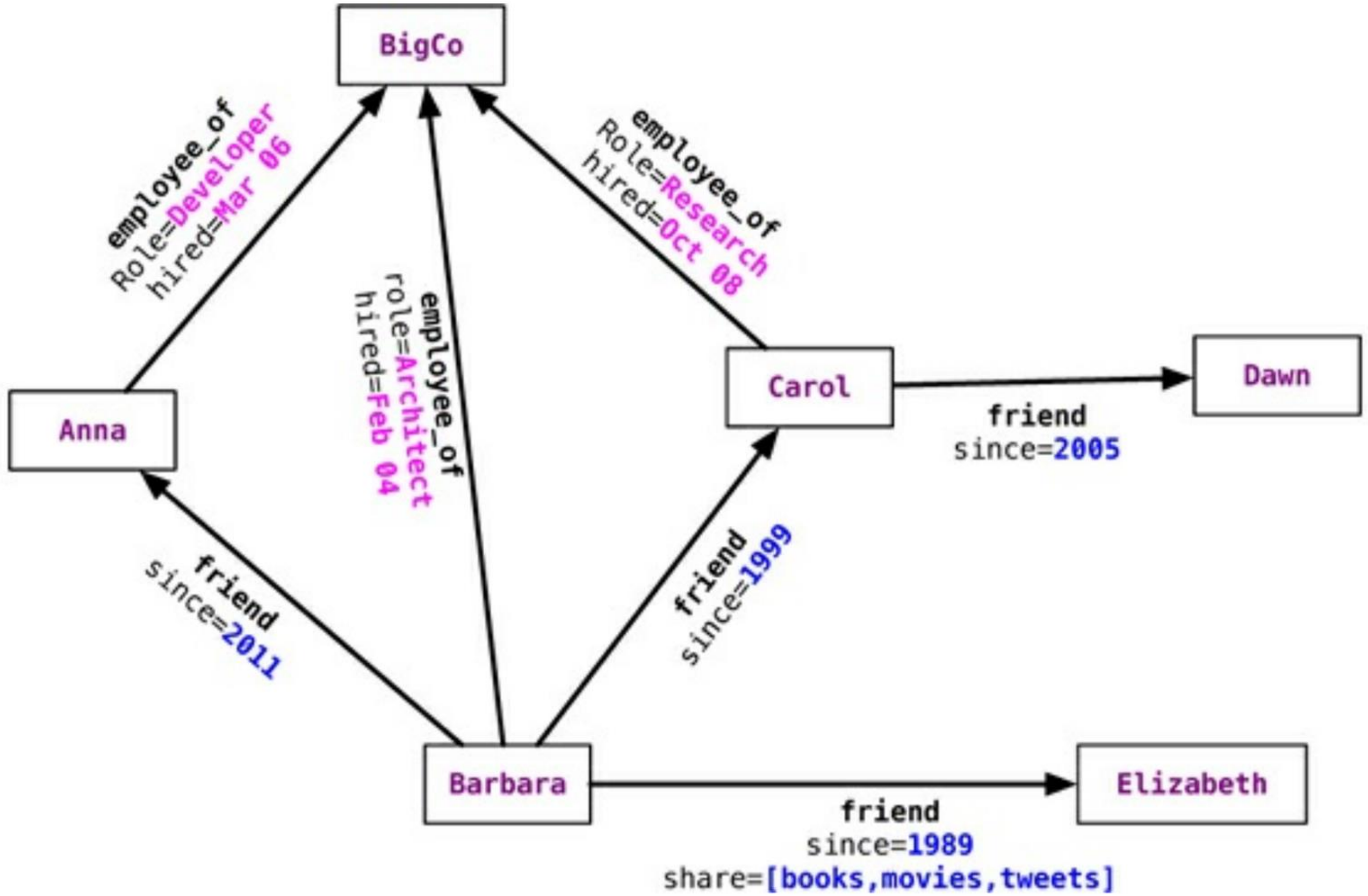
- A few very large graphs
  - e.g., Web graph, social networks, …
- Queries:
  - Nodes/edges with properties
  - Neighboring nodes/edges
  - Paths (all, shortest, etc.)

- Our example: Neo4j

Basic Characteristics

- Different types of relationships between nodes
  - To represent relationships between domain entities
  - Or to model any kind of secondary relationships
    - Category, path, time-trees, spatial relationships, …
- No limit to the number and kind of relationships
- Relationships have: type, start node, end node, own properties
  - e.g., “since when” did they become friends

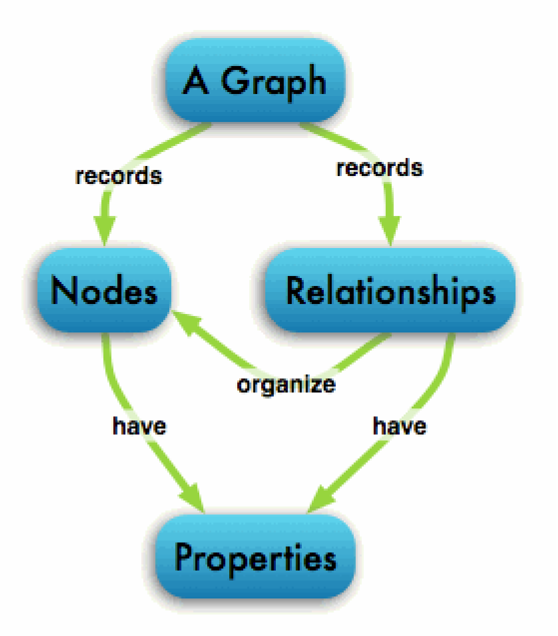
Relationship Properties: Example



source: Sadalage & Fowler: NoSQL Distilled, 2012

Neo4j: Basic Info

- Open source graph database
  - The most popular
- Initial release: 2007
- Written in: Java
- OS: cross-platform
- Stores data as nodes connected by directed, typed relationships
  - With properties on both
  - Called the “property graph”

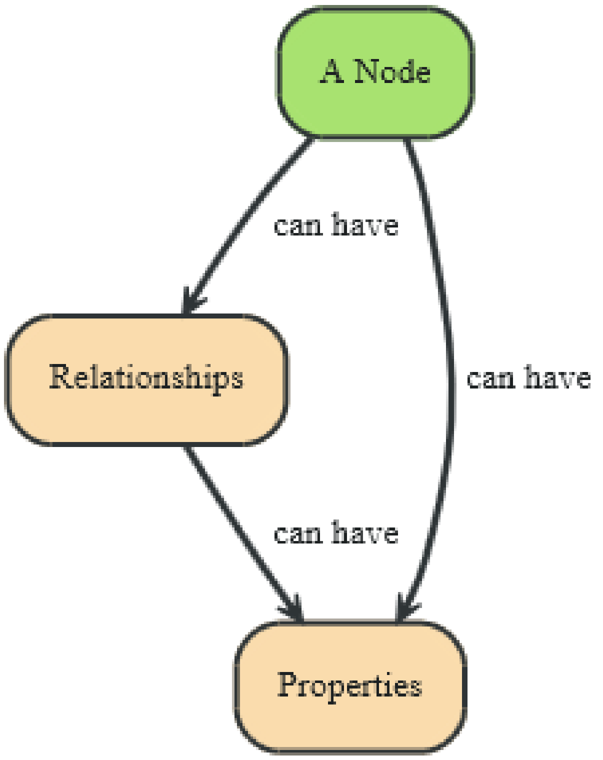


Neo4j: Basic Features

- reliable – with full ACID transactions
- durable and fast – disk-based, native storage engine
- scalable – up to several billion nodes/relationships/properties
- highly-available – when distributed (replicated)
- expressive – powerful, human readable graph query language
- fast – powerful traversal framework
- embeddable - in Java program
- simple – accessible by REST interface & Java API

Neo4j: Data Model

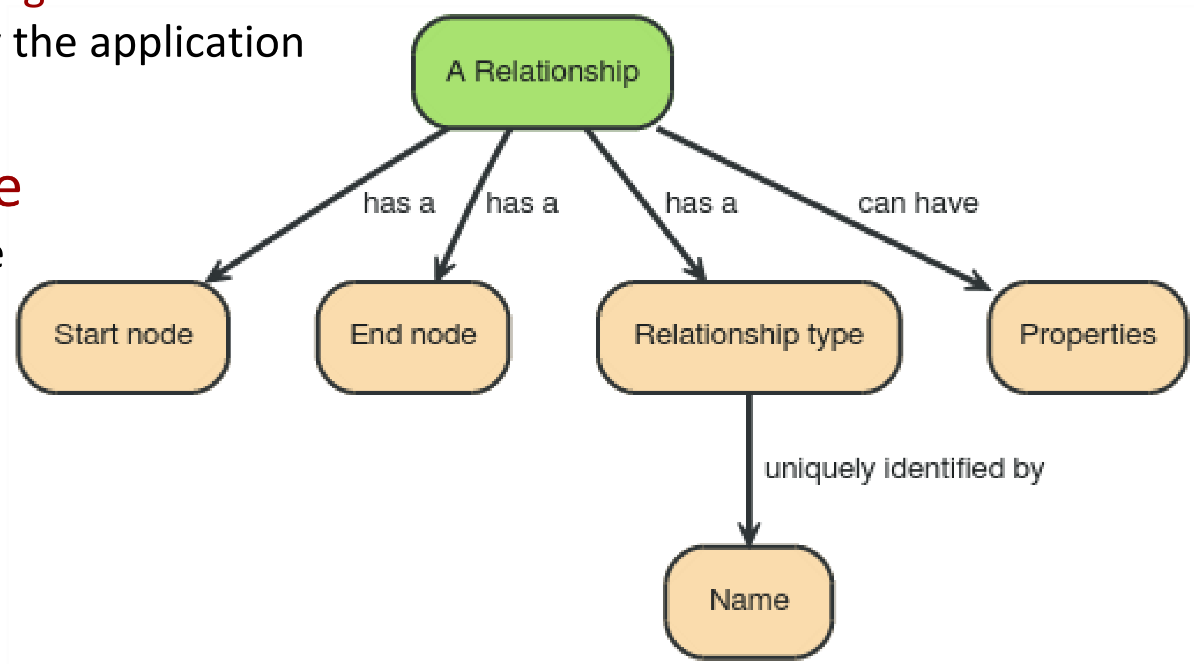
- Fundamental units: nodes + relationships
- Both can contain properties
  - Key-value pairs
  - Value can be of primitive type or an array of primitive type
  - null is not a valid property value
    - nulls can be modelled by the absence of a key



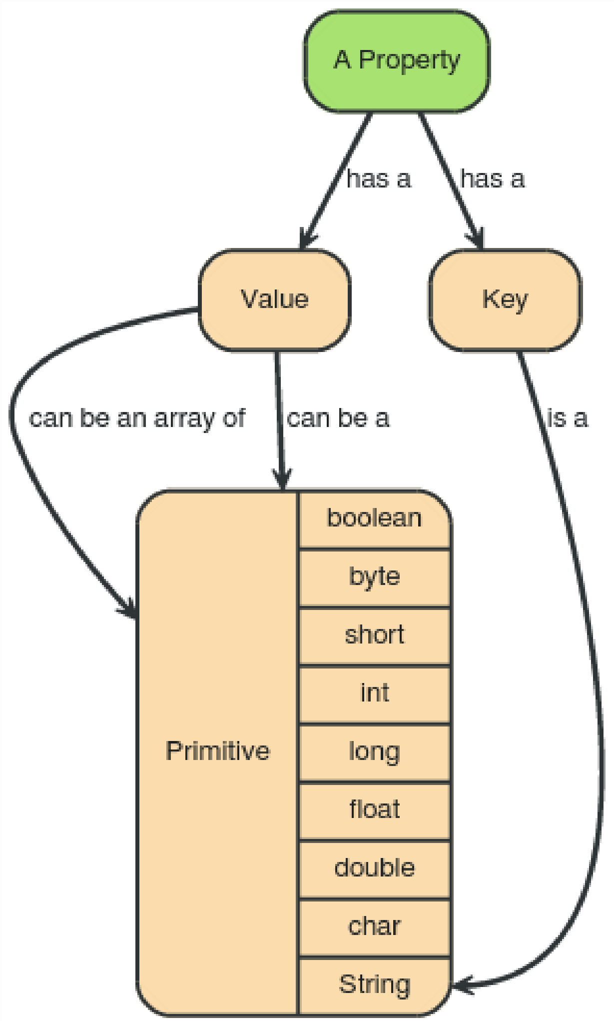
http://db-engines.com/en/system/Neo4j

Data Model: Relationships

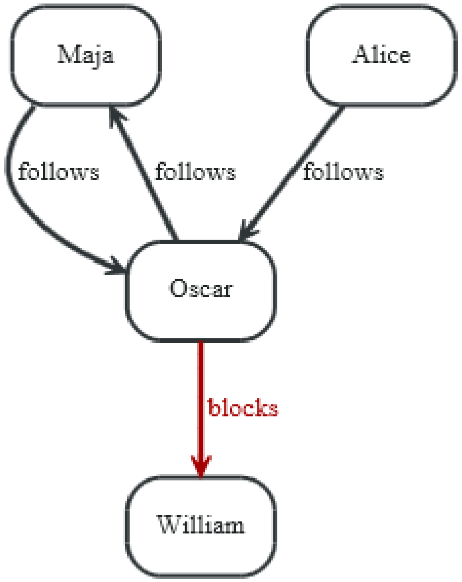
- Directed relationships (edges)
  - Incoming and outgoing edge
    - Equally efficient traversal in both directions
    - Direction can be ignored
      if not needed by the application
  - Always a start
    and an end node
    - Can be recursive



Data Model: Properties

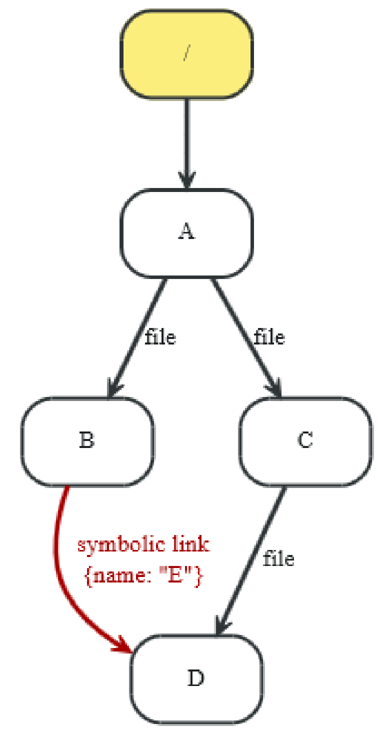


Type	Description
boolean	true/false
byte	8-bit integer
short	16-bit integer
int	32-bit integer
long	64-bit integer
float	32-bit IEEE 754 floating-point number
double	64-bit IEEE 754 floating-point number
char	16-bit unsigned integers representing Unicode characters
String	sequence of Unicode characters



What	How
get who a person follows	outgoing *follows* relationships, depth one
get the followers of a person	incoming *follows* relationships, depth one
get who a person blocks	outgoing *blocks* relationships, depth one

What	How
get the full path of a file	incoming *file* relationships
get all paths for a file	incoming *file* and *symbolic link* relationships
get all files in a directory	outgoing *file* and *symbolic link* relationships, depth one
get all files in a directory, excluding symbolic links	outgoing *file* relationships, depth one
get all files in a directory, recursively	outgoing *file* and *symbolic link* relationships



Access to Neo4j

- Embedded database in Java system
- Language-specific connectors
  - Libraries to connect to a running Neo4j server
- Cypher query language
  - Standard language to query graph data
- HTTP REST API
- Gremlin graph traversal language (plugin)
- etc.

Graph DBs: Suitable Use Cases

- Connected Data
  - Social networks
  - Any link-rich domain is well suited for graph databases
- Routing, Dispatch, and Location-Based Services
  - Node = location or address that has a delivery
  - Graph = nodes where a delivery has to be made
  - Relationships = distance
- Recommendation Engines
  - “your friends also bought this product”
  - “when buying this item, these others are usually bought”

Graph DBs: When Not to Use

- If we want to update all or a subset of entities
  - Changing a property on many nodes is not straightforward
    - e.g., analytics solution where all entities may need to be updated with a changed property
- Some graph databases may be unable to handle lots of data
  - Distribution of a graph is difficult

Concluding Remarks on Common NoSQL

- Aim to avoid join & ACID overhead
  - Joined within, correctness compromised for quick answers; believe in best effort
- Avoids the idea of a schema
- Query languages are more imperative
  - And less declarative
  - Developer better knows what's going on; less reliance on smart optimization plans
  - More responsibility on developers
- No standard well studied languages (yet)