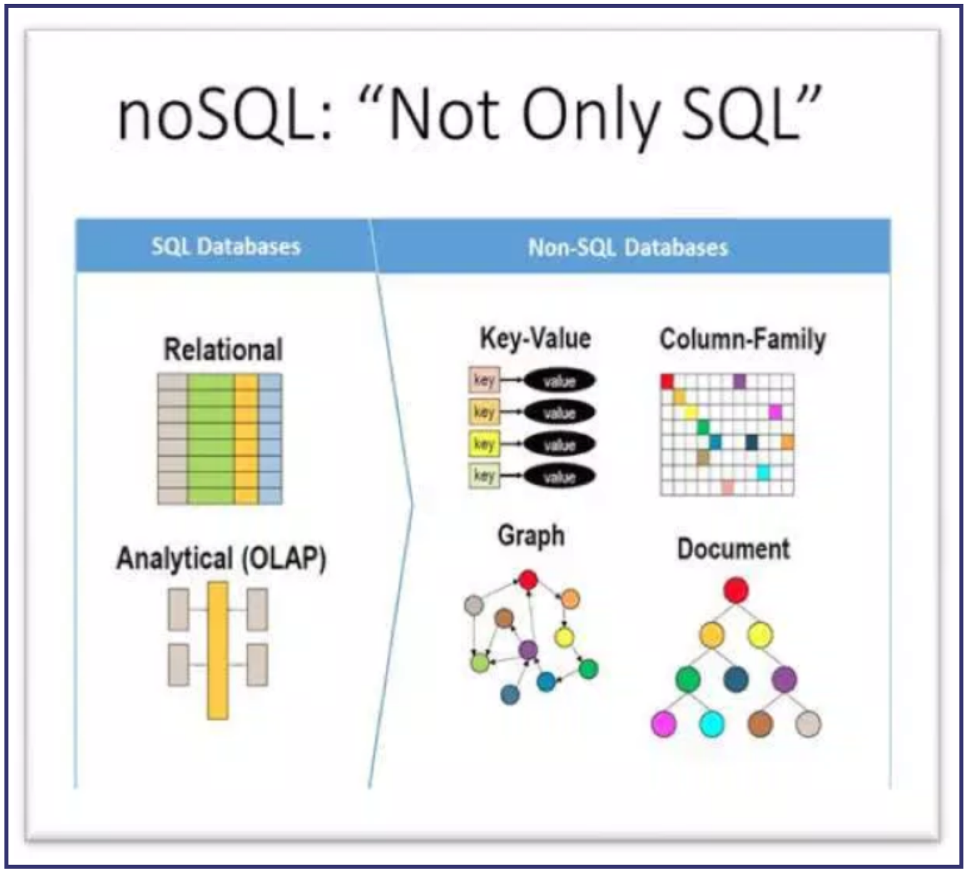
# NoSQL databases

## Types of NoSQL databases (based on underlying data model)

There are [only] FOUR types of NoSQL DBs, based on their underlying data representation:



- key-value store: DynamoDB (Amazon), Project Voldemort, Redis, Tokyo Cabinet/Tyrant..
- column-family store: Cassandra, HBase..
- document store: MongoDB, CouchDB, MarkLogic..
- graph store: Neo4j, HyperGraphDB, Seasame..

Let's take a look at the above in more detail..

HOW MANY of these (NoSQL DBs) are there? :)

## Comparing NoSQL DBs

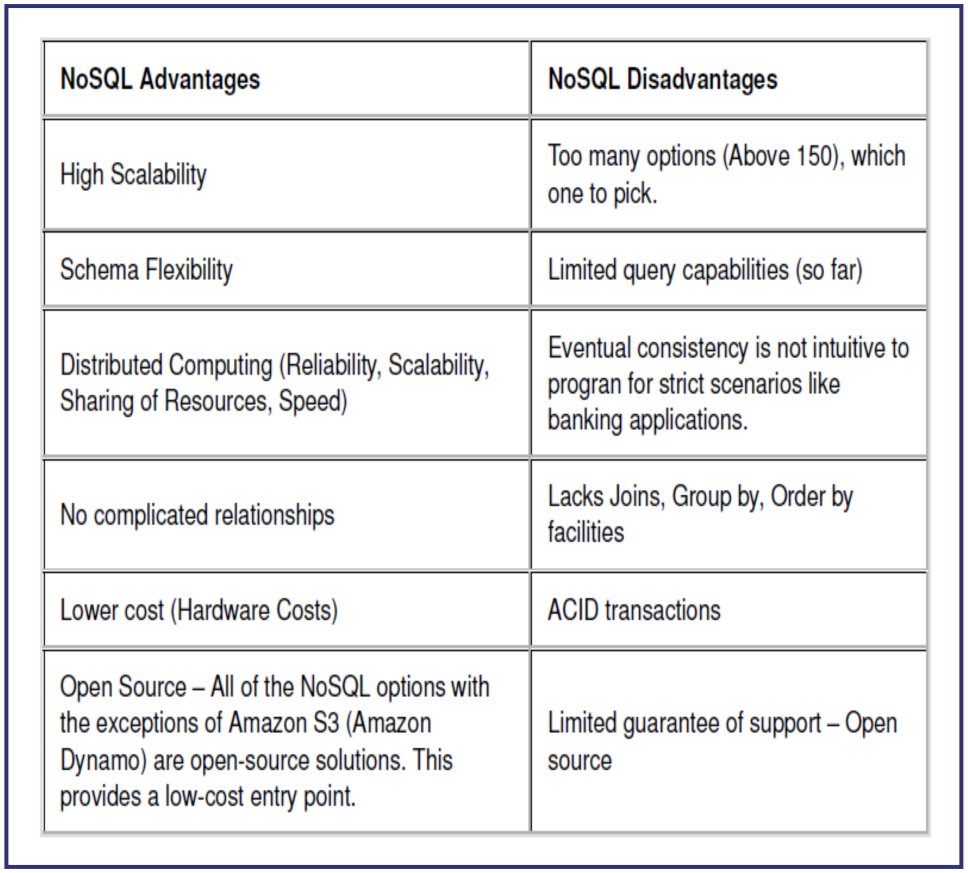
Because there are so many NoSQL DBs with varied architectures, it would be good to have a standard way (item list) to compare them.

Here is one comparison metric:

- Architecture
  - Topology
  - Consistency
  - Fault tolerance (and partition tolerance)
  - Structure and format
- Administration
  - Logging and Statistics
  - Configuration management
  - Backup
  - Disaster recovery
  - Maintenance

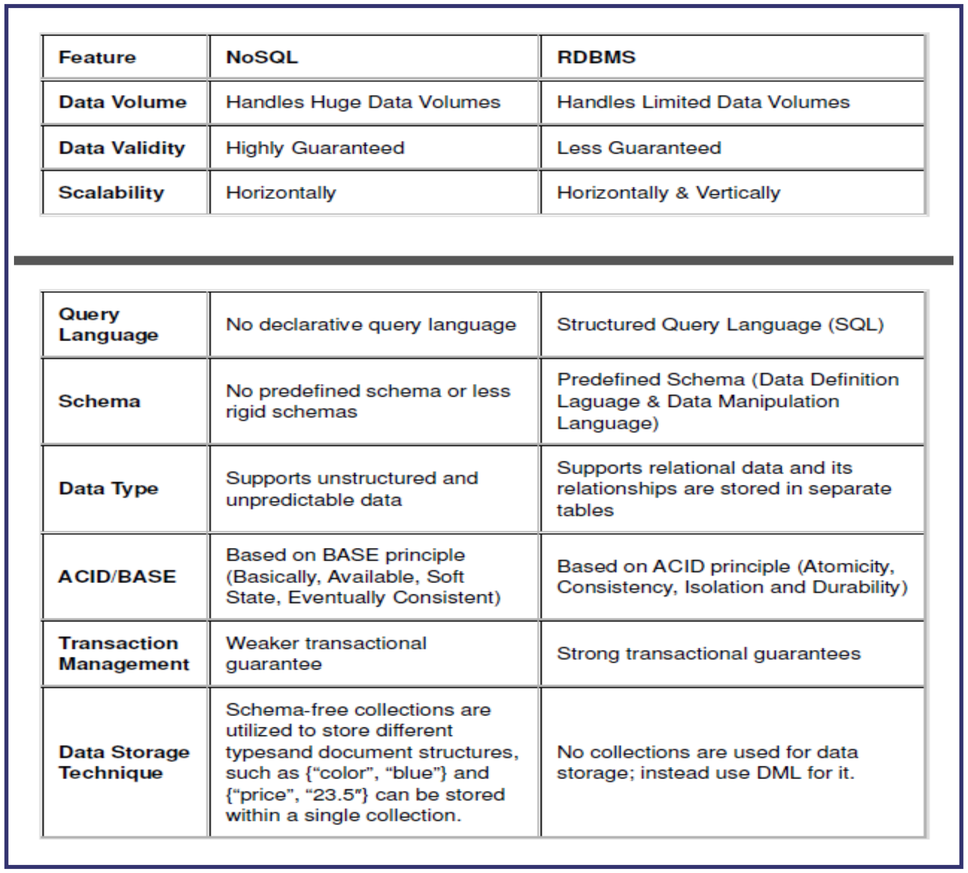
  - Recovery
  - Monitoring functionality
  - Security
- Deployment
  - Availability
  - Stability
- Development
  - Documentation
  - Integration
  - Support
  - Usability
- Performance and scalability
  - Performance
  - Scalability

## Comparison



| NoSQL Advantages | NoSQL Disadvantages |
|---|---|
| High Scalability | Too many options (Above 150), which one to pick. |
| Schema Flexibility | Limited query capabilities (so far) |
| Distributed Computing (Reliability, Scalability, Sharing of Resources, Speed) | Eventual consistency is not intuitive to progran for strict scenarios like banking applications. |
| No complicated relationships | Lacks Joins, Group by, Order by facilities |
| Lower cost (Hardware Costs) | ACID transactions |
| Open Source – All of the NoSQL options with the exceptions of Amazon S3 (Amazon Dynamo) are open-source solutions. This provides a low-cost entry point. | Limited guarantee of support – Open source |

## Another comparison



| Feature | NoSQL | RDBMS |
|---|---|---|
| Data Volume | Handles Huge Data Volumes | Handles Limited Data Volumes |
| Data Validity | Highly Guaranteed | Less Guaranteed |
| Scalability | Horizontally | Horizontally & Vertically |

| | | |
|---|---|---|
| Query Language | No declarative query language | Structured Query Language (SQL) |
| Schema | No predefined schema or less rigid schemas | Predefined Schema (Data Definition Laguage & Data Manipulation Language) |
| Data Type | Supports unstructured and unpredictable data | Supports relational data and its relationships are stored in separate tables |
| ACID/BASE | Based on BASE principle (Basically, Available, Soft State, Eventually Consistent) | Based on ACID principle (Atomicity, Consistency, Isolation and Durability) |
| Transaction Management | Weaker transactional guarantee | Strong transactional guarantees |
| Data Storage Technique | Schema-free collections are utilized to store different typesand document structures, such as {“color”, “blue”} and {“price”, “23.5”} can be stored within a single collection. | No collections are used for data storage; instead use DML for it. |

## 'Polyglot persistence'

Martin Fowler (of 'Code Refactoring' fame) has this article on post-RDBMS alternatives: http://martinfowler.com/articles/nosql-intro-original.pdf

Polyglot persistence means the use of different storage technologies (ie NoSQL DBs) to store different parts (data) of a single application. These individual data stores are then tied together by the application, using DB APIs or web services APIs.

## Some examples of NoSQL adoption (2015)

Gannett, publisher of USA Today and 90+ media properties, replaced relational database technology with NoSQL to power its digital publishing platform.

Marriott deployed NoSQL to modernize its hotel reservation system that supports **38billioninannualbookings.**

Ryanair moved to NoSQL to power its mobile app that serves over 3 million passengers.

## So in summary, why choose a NoSQL DB?

Two broad reasons:

- "to improve programmer productivity by using a database that better matches an application's needs"
- "to improve data access performance via some combination of handling larger data volumes, reducing latency, and improving throughput"

## A pair of terms

In NoSQL dbs, we talk a lot about nodes and clusters.

Node: a single NoSQL db instance - holds a part of a db.

Cluster: a collection of nodes - holds an entire db.

## Key-value DBs

"A key-value store, or key-value database, is a data storage paradigm designed for storing, retrieving, and managing associative arrays, a data structure more commonly known today as a dictionary or hash."

So the whole db is a dictionary, which has records ("rows") which have fields (columns).

Because there is no schema, the records all don't have to contain identical fields!

## Key-value DBs

The 'key' in a key-value DB, is comparable to a PK in a relation (table); the 'value' is an aggregate of all the dependent (non-PK) columns.

Querying occurs only on keys. When querying on a key, the entire value (aggregate) (for matching keys) is returned. Likewise, the entire value needs to be updated when necessary (no per-field updating).

## Key-value DBs: Memcached

Memcached [aka memcached - all lowercase] is a high-performance, in-memory, data caching system, with a VERY simple API:

- store (SET) a value, given a key (eg. memcache->set(key,val))
- retrieve (GET) the value, given the key (eg. val=memcache->get(key))

The value that is stored does not have to be atomic, it can even be an associative array (k-v pairs, aka dictionary, or hash) - that said, stored values are usually rather small (~1M). Eg. the following PHP snippet shows how an array of 6 different elements [int, float.. object] is stored in memcached, as 6 k-v pairs [note that we could have stored the entire array as a single value, but that would make it inefficient to access an individual element such as the boolean]:

```php
$checks = array(
    123,
    4542.32,
    'a string',
    true,
    array(123, 'string'),
    (object)array('key1' => 'value1'),
);
foreach ($checks as $i => $value) {
    print "Checking WRITE with Memcache\n";
    $key = 'cachetest' . $i;
    $memcache->set($key, $value);
    usleep(100);
    $val = $memcache->get($key);
    $valD = $memcacheD->get($key);
    if ($val !== $valD) {
        print "Not compatible!";
        var_dump(compact('val', 'valD'));
    }
```

Memcached is commonly used with a 'backend' SQL store (relational DB) - a COPY of the frequently accessed data is held in memcached, for fast retrieval and update.

Quite a few languages are supported: C/C++, Java, JavaScript, Python, PHP, Ruby, Perl, .NET, Erlang, Lua, Lisp, Ocaml..

## If you have access to a Linux server (including being able to run one locally on your machine), you can experiment with memcached like so.

## Key-value DBs: Redis

Redis is an extremely popular K-V database, used by a variety of (social media etc.) companies to hold vast amounts of data.

For values, rather than just atomic datatypes (number, string, boolean), Redis offers richer types such as list, set, dictionary.

## Key-value DBs: Redis, and Instagram

Redis ALSO offers a way to use hashing, for storing keys. A large set of keys (eg. 1M) can be stored in sets of 1000 (for ex), in 1000 different hashes. Within each hash "set", Redis encodes the subkeys quite efficiently (small memory footprint) - here is more detail.

Result - dramatic reduction in storage reqs, eg. from 21G to 5G for Instagram :)

## Key-value DBs: Amazon's Dynamo

Dynamo is what is internally used at Amazon (as distinct from S3, offered to cloud services users).

From the Dynamo paper that they published:

- the infrastructure is made up by tens of thousands of servers and network components located in many datacenters around the world.
- commodity hardware is used.
- component failure is the 'standard mode of operation'.
- 'Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services'.

"Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance."

## Key-value DBs: Amazon's Dynamo - k/v design

In Dynamo, values are stored as BLOBS (opaque bytes of binary data). Operations are limited to a single k/v pair at a time.

Only two ops are supported:

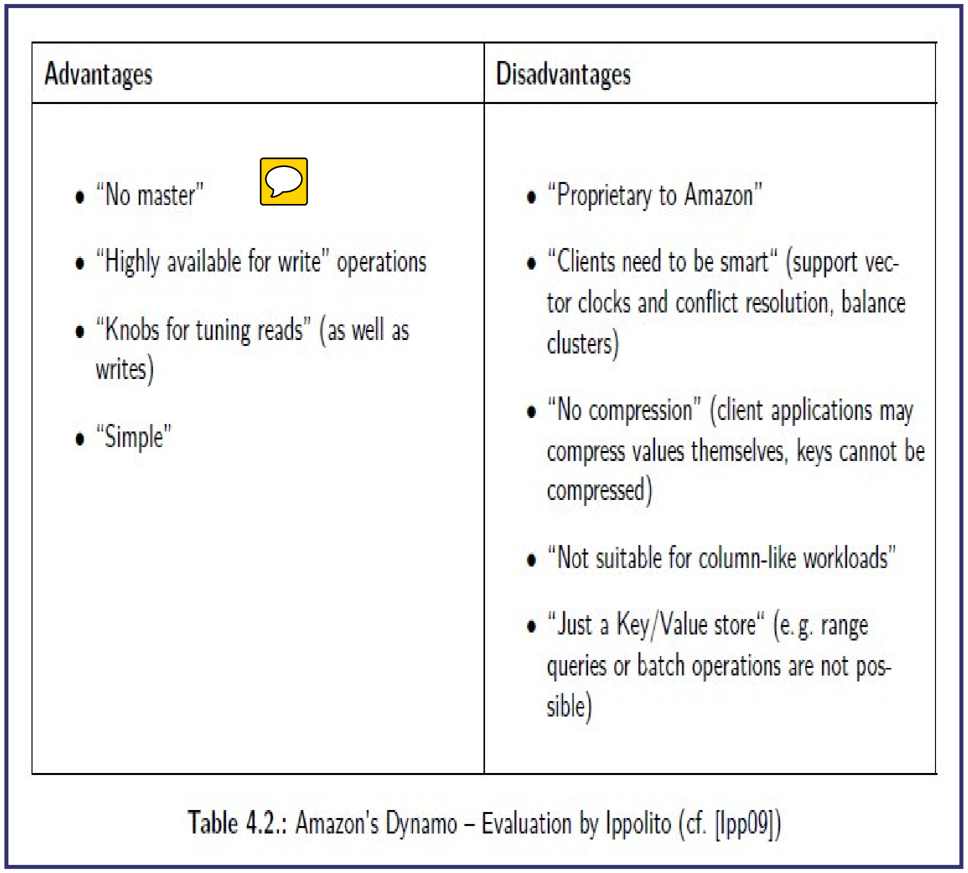
- get(key) - returns a list of objects and a context
- put(key, context, object) - no return value

In the above, 'context' is used to store metadata about the BLOB values, eg. version numbers (this is used during 'eventual consistency' DB updating).

Keys are hashed using the MD5 hashing algorithm, to result in appropriate storage locations (nodes).

To ensure availability and durability, each k/v pair is replicated N times (N=3, commonly) and stored in N adjacent nodes starting from the key's hash's node.

## Key-value DBs: Amazon's Dynamo - pros, cons



| Advantages | Disadvantages |
|---|---|
| • "No master"<br>• "Highly available for write" operations<br>• "Knobs for tuning reads" (as well as writes)<br>• "Simple" | • "Proprietary to Amazon"<br>• "Clients need to be smart" (support vector clocks and conflict resolution, balance clusters)<br>• "No compression" (client applications may compress values themselves, keys cannot be compressed)<br>• "Not suitable for column-like workloads"<br>• "Just a Key/Value store" (e.g. range queries or batch operations are not possible) |

Table 4.2.: Amazon's Dynamo – Evaluation by Ippolito (cf. [Ipp09])

Summary - k/v DBs are lightweight (simple), schema-less, transaction-less.

## Column family DBs

After k/v DBs, column (aka column family) DBs are the next form of data storage.

Examples: BigTable (Google), HBase (Apache), Cassandra, Amazon SimpleDB, Hypertable.

## Column family DBs - terminology

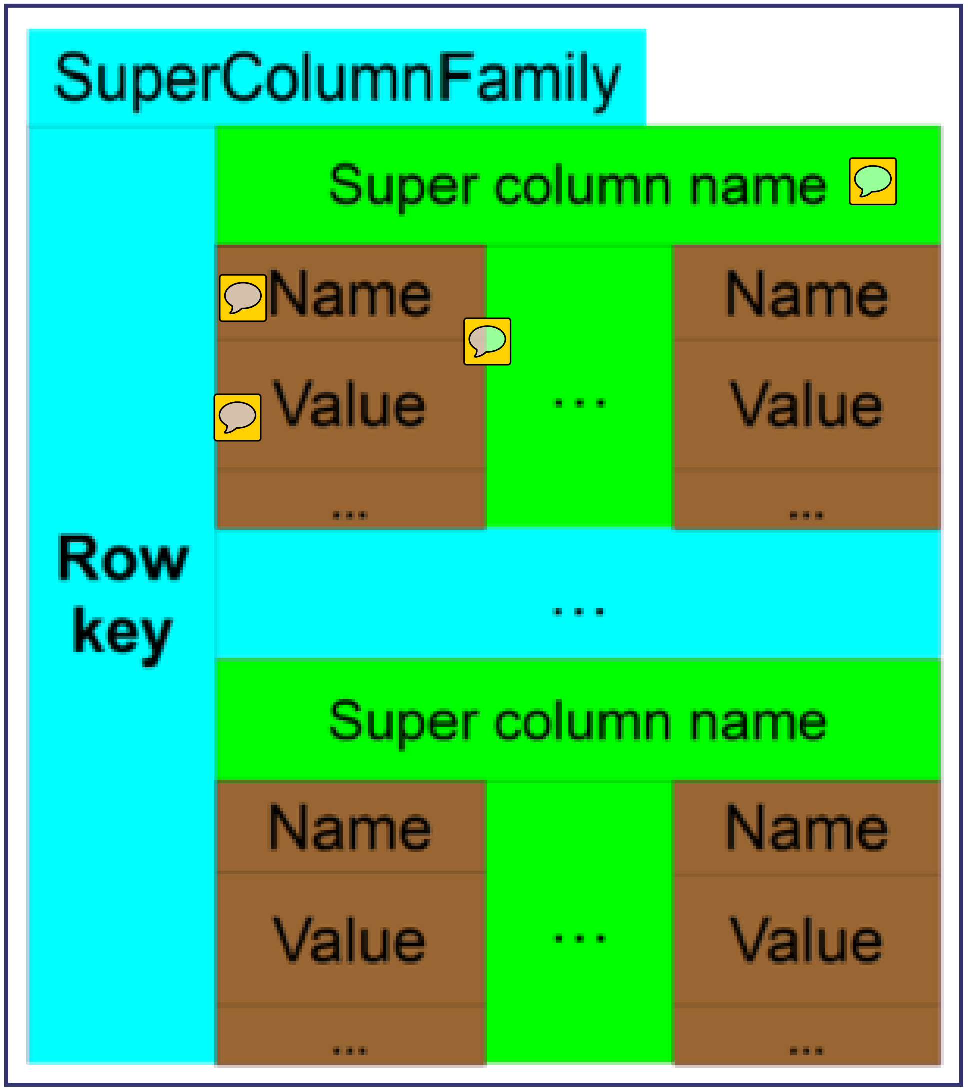
As the name signifies, data is stored as groups of columns (column family), as opposed to an RDBMS where data is stored as rows.

Column family: contains columns of related data (eg. for a Users DB, the columns might be Name, Age, DOB, Address, Email). A column family can be thought of as a 'namespace' for a group of columns. A column family would have many rows of data, where for each row, there would be multiple columns and values. The SQL eqvt of this would be ONE TABLE PER ROW!!

Super column family: a collection of super columns. A super column has a name (key), and contains as its value, other column names and values! Eg:

```
UserList={
   Cath:{
      username:{firstname:'Cath',lastname:'Yoon'}
      address:{city:'Seoul',postcode:'1234'}
   }
   Terry:{
      username:{firstname:'Terry',lastname:'Cho'}
      account:{bank:'hana',accounted:'1234'}
   }
 }
```

Pictorially, the data for 'Cath' above (for ex) would look like this:

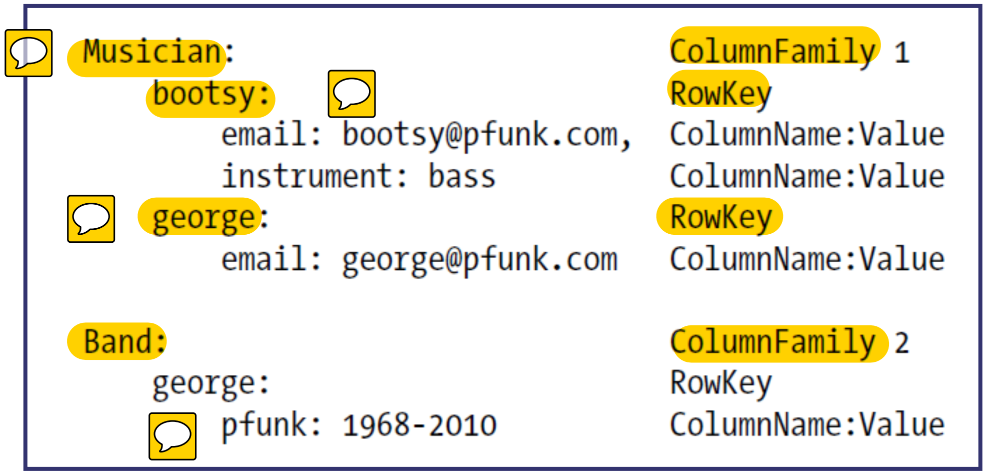


Super column families are meant for 'inverted indexing', we don't *need* to use them in our column family DB.

Column: consists of a name, and a value (and a timestamp).

In a column family DB, data is stored using 'row keys' (each row is assigned a unique key).

Here is another example of a column family (two column families, actually) - note that ColumnFamily1 has 'jagged' (uneven) data:

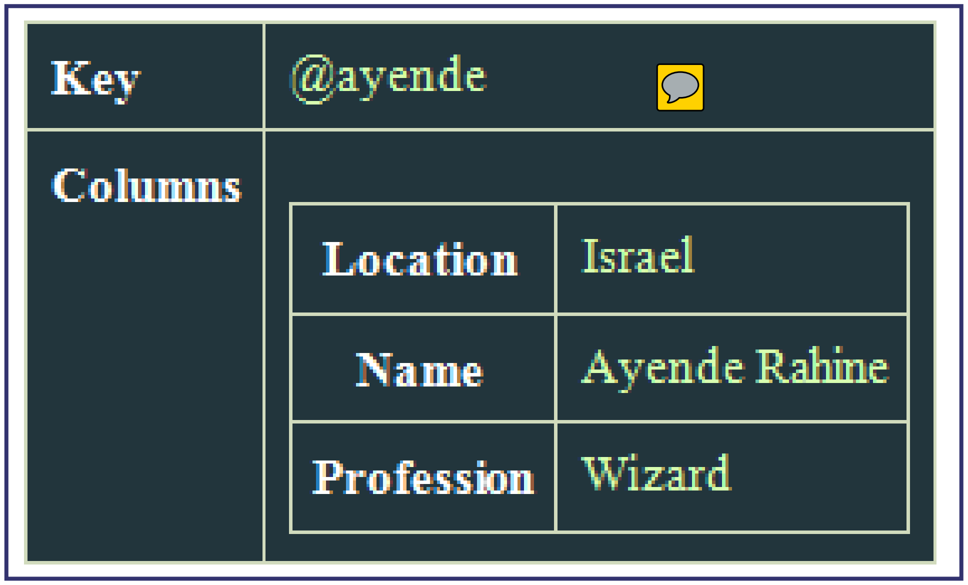


```
Musician:                                  ColumnFamily 1
    bootsy:                                RowKey
        email: bootsy@pfunk.com,           ColumnName:Value
        instrument: bass                   ColumnName:Value
    george:                                RowKey
        email: george@pfunk.com            ColumnName:Value

Band:                                      ColumnFamily 2
    george:                                RowKey
        pfunk: 1968-2010                   ColumnName:Value
```

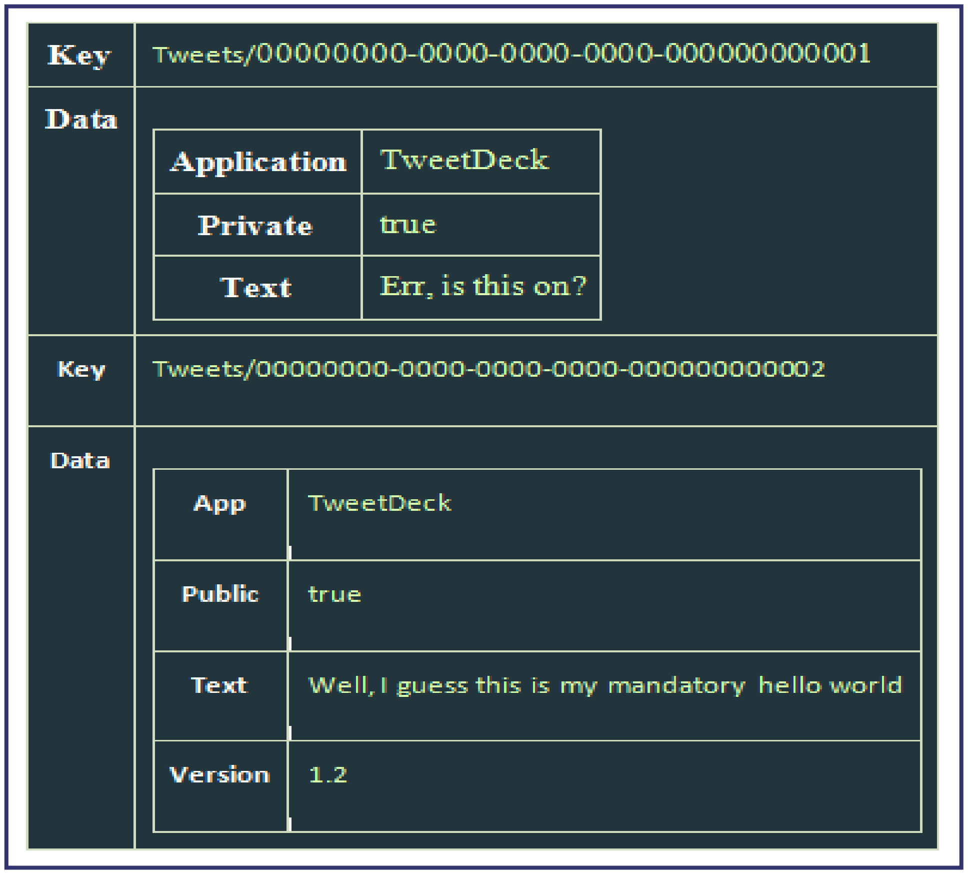
## Column family DBs - storing tweets

Here is how we'd store tweets, using a column family DB [example from https://ayende.com/blog/4500/that-no-sql-thing-column-family-databases]. We create two column families - Users, Tweets. We also create a super column family - UsersTweets.

Here are Users and Tweets:

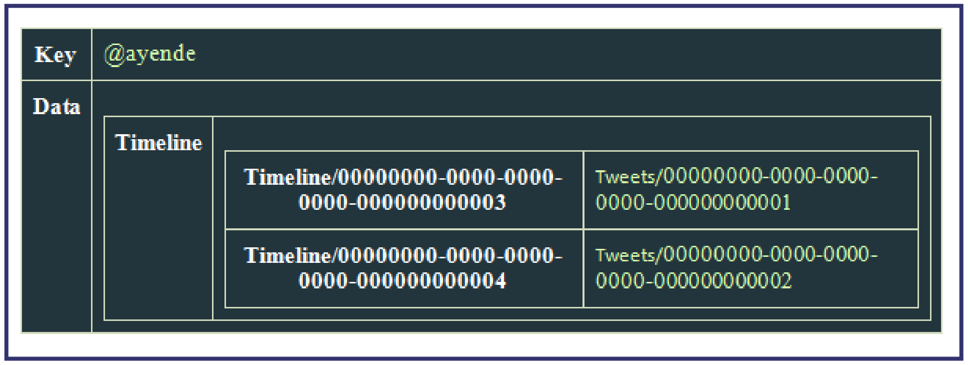


| | | |
|---|---|---|
| Key | @ayende | |
| Columns | Location | Israel |
| | Name | Ayende Rahine |
| | Profession | Wizard |



| | |
|---|---|
| **Key** | Tweets/00000000-0000-0000-0000-000000000001 |
| **Data** | **Application**: TweetDeck; **Private**: true; **Text**: Err, is this on? |
| **Key** | Tweets/00000000-0000-0000-0000-000000000002 |
| **Data** | **App**: TweetDeck; **Public**: true; **Text**: Well, I guess this is my mandatory hello world; **Version**: 1.2 |

## And here is the super column (UsersTweets):



| | | | |
|---|---|---|---|
| **Key** | @ayende | | |
| **Data** | **Timeline** | **Timeline/00000000-0000-0000-0000-000000000003** | Tweets/00000000-0000-0000-0000-000000000001 |
| | | **Timeline/00000000-0000-0000-0000-000000000004** | Tweets/00000000-0000-0000-0000-000000000002 |

## To query the above, we'd do this:

```
var tweetIds =
      cfdb.UsersTweets.Get('@ayende')
             .Fetch('timeline')
             .Take(25)
             .OrderByDescending()
             .Select(x=>x.Value);

var tweets = cfdb.Tweets.Get(tweetIds);
```

Note that we execute queries at two levels - from the UsersTweets super column family we get tweetIds as VALUES, which we then use as KEYS in the Tweets family.

## Column family DBs - Google's BigTable

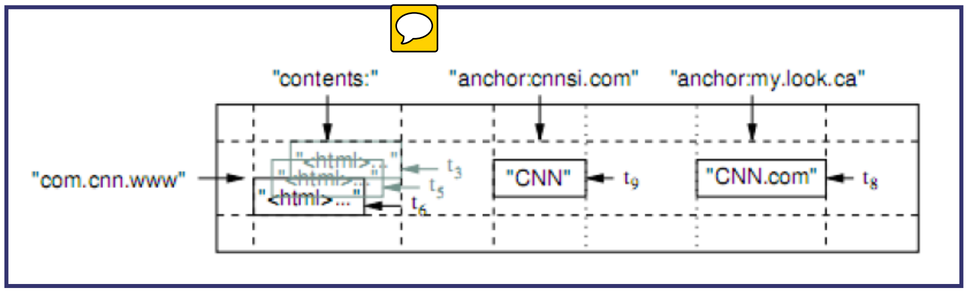
BigTable - "a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers".

Used in Google Earth, Google Analytics, Google Docs, etc.

"BigTable has achieved several goals: wide applicability, scalability, high performance, and high availability."

# Column family DBs - Google's BigTable

The storage data structure is "a sparse, distributed, persistent multidimensional sorted map". Example:



Values are addressed by (row-key, column-key, timestamp).

In the above, we have two column families: content (one column), anchor (two columns).

## Column family DBs - BigTable derivatives

Hypertable

Written in C++, based on HDFS and distributed lock managing.

Can have column-families with an arbitrary number of distinct columns.

Tables are partitioned by ranges of row keys (like in BigTable) and the resulting partitions get replicated between servers.

Features HQL, a C++ native API and the 'Thrift' API.

## Column family DBs - BigTable derivatives

HBase

BigTable clone written in Java, as part of the Hadoop implementation.

An HBase db can be the source or target for a MapReduce task running on Hadoop.

Has a native Java API, Thrift API and REST web services.

## Column family DBs - Cassandra

Was originally developed by Facebook and open-sourced in 2008.

Also used by Twitter, Digg, Rackspace.

Data model is as follows:

Rows - which are identified by a string-key of arbitrary length. Operations on rows are atomic per replica no matter how many columns are being read or written.

Column Families - which can occur in arbitrary number per row. As in Bigtable, column-families have to be defined in advance, i. e. before a cluster of servers comprising a Cassandra instance is launched. The number of column-families per table is not limited; however, it is expected that only a few of them are specified. A column family consists of columns and supercolumns which can be added dynamically (i. e. at runtime) to column-families and are not restricted in number.

Columns have a name and store a number of values per row which are identified by a timestamp (like in Bigtable). Each row in a table can have a different number of columns, so a table cannot be thought of as a rectangle. Client applications may specify the ordering of columns within a column family and supercolumn which can either be by name or by timestamp.

Supercolumns have a name and an arbitrary number of columns associated with them. Again, the number of columns per super-column may differ per row.

## Column family DBs - Cassandra API

Very simple:

- get(table, key, columnName)
- insert(table, key, rowMutation)
- delete(table, key, columnName)

Thrift API, plus language bindings for Ruby, Python, C#, Java.

## Column family DBs - Cassandra CQL

Query language is "CQL" - somewhat like SQL, but no WHERE, JOIN, GROUP BY, ORDER BY.

```
CREATE KEYSPACE animalkeyspace
WITH REPLICATION = { 'class' : 'SimpleStrategy' ,
 'replication_factor' : 1 };

use animalkeyspace;

CREATE TABLE Monkey (
  identifier uuid,
  species text,
  nickname text,
  population int,
  PRIMARY KEY ((identifier), species));

INSERT INTO monkey (identifier, species, nickname,
population)
VALUES ( 5132b130-ae79-11e4-ab27-0800200c9a66,
'Capuchin monkey', 'cute', 100000);

Select * from monkey;



./sstable2json
$YourDataDirectory/data/animalkeyspace/monkey/animalkeyspace
-monkey-jb-1-Data.db




[
  {
    "key": "5132b130ae7911e4ab270800200c9a66", // The
row/partition key
    "columns": [                              // All
Cassandra internal columns
      [
        "Capuchin monkey:",                         // The
Cluster key. Note the cluster key does not have any data
associated with it. The key and the data are same.
        "",
        1423586894518000                            // Time stamp
which records when this internal column was created.
      ],
      [
        "Capuchin monkey:nickname",                // Header for
the nickname internal column. Note the cluster key is always
prefixed for every additional internal column.
        "cute",                                    // Actual data
        1423586894518000
      ],
      [
        "Capuchin monkey:population",             // Header for
the population internal column
        "100000",                                 // Actual Data
        1423586894518000
      ]
```

```
          ]
     }
]
```

## Recap: 4 types of NoSQL DBs

All non-relational ("NoSQL") DBs can be grouped into these categories:

- key-value DBs
- column family DBs
- document DBs
- graph DBs (including triple stores)

So far we learned about key-value DBs and column family DBs; next we'll look at document DBs and graph DBs..

## Document DBs: terminology

A document DB is a 'collection' of 'documents' (analogy with an RDMS: documents are equivalent to rows, and a collection is equivalent to a table).

The basic unit of storage in a document DB is, well, a document - this can be JSON, XML, etc. There can be an arbitrary number of fields (columns and values, ie. k/v pairs) in each document.

## Document DBs: keys, values

A document DB can be considered to be a more sophisticated version of a k-v store.

In a document DB, a key is paired with a document (which is its 'value'), where the document itself can contain multiple k/v pairs, key-array pairs, or even key-document pairs (ie nested documents).

## Document DB examples

Here are leading document-oriented DBs:

- Couchbase
- CouchDB ['Cluster of unreliable commodity hardware' :) :)]
- MongoDB
- OrientDB

You might enjoy this Couchbase application note :)

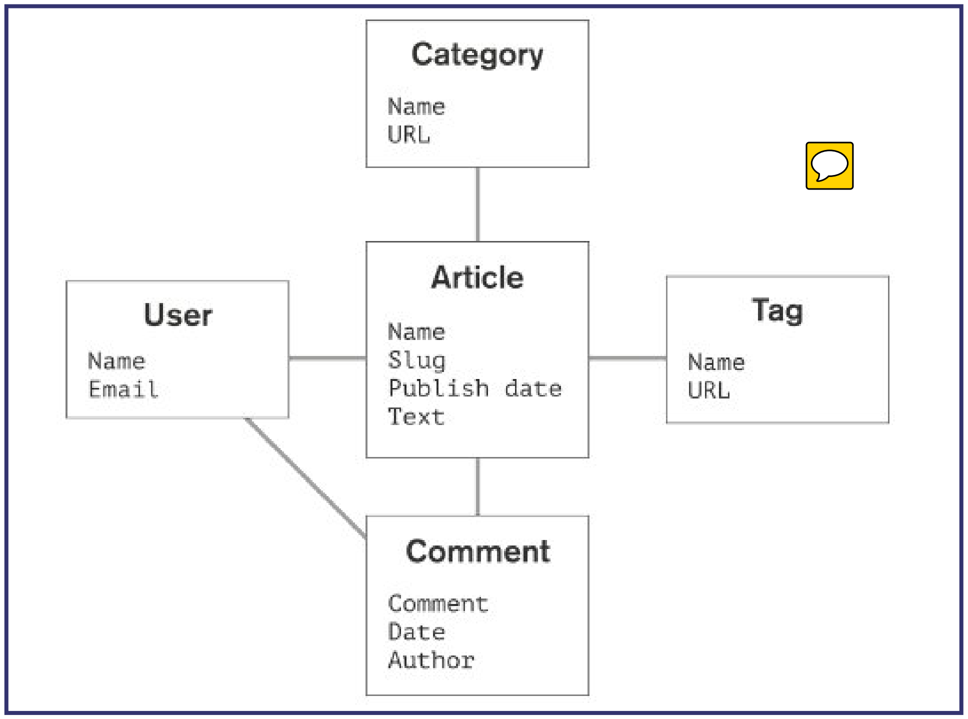
## Who uses document DBs?

LOTS of organizations, including eBay, ADP, MetLife, MTV, BuzzFeed..

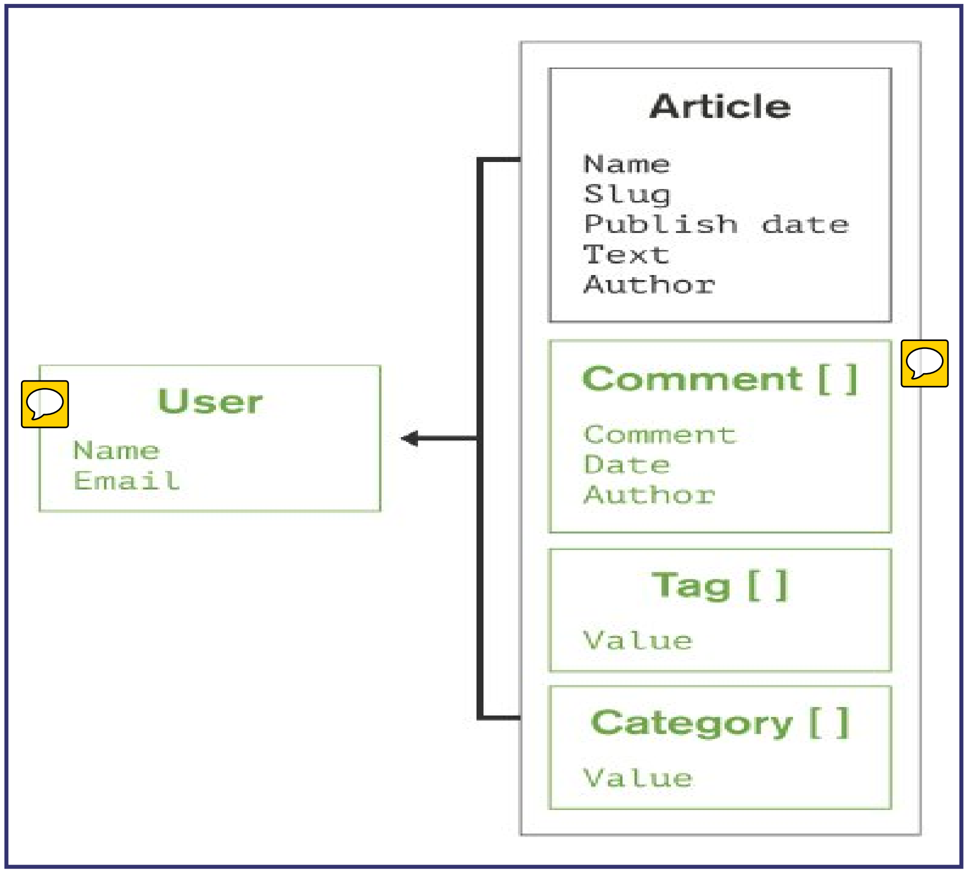
Because entire documents are stored, there is no need to perform (expensive) JOIN operations.. Also, queries can be parallelized using MapReduce.

# Document DB: example representation

Consider the following diagram, which shows how blog posts could be organized:



## The corresponding document store would look like this:



Note that document contents are saved as BSON (Binary JSON) in order to save disk space. As a reminder, values can be atomic data, arrays or objects; fields can vary from document to document (no set schema).

## Document DBs: querying

Queries are non-SQL, of course, and offer richness and flexibility. For example, MongoDB offers the following query types:

- simple k/v queries - can return a value for any field in the stored documents; usually, we do a simple PK search (given the doc's key, retrieve the entire doc)
- range queries: return values based on >, <=, BETWEEN..
- geo-spatial queries
- text queries - full text search, using AND, OR, NOT
- aggregation framework: column-oriented operations such as count, min, max, avg
- MapReduce queries - get executed via MapReduce

## Document DB: sample query commands

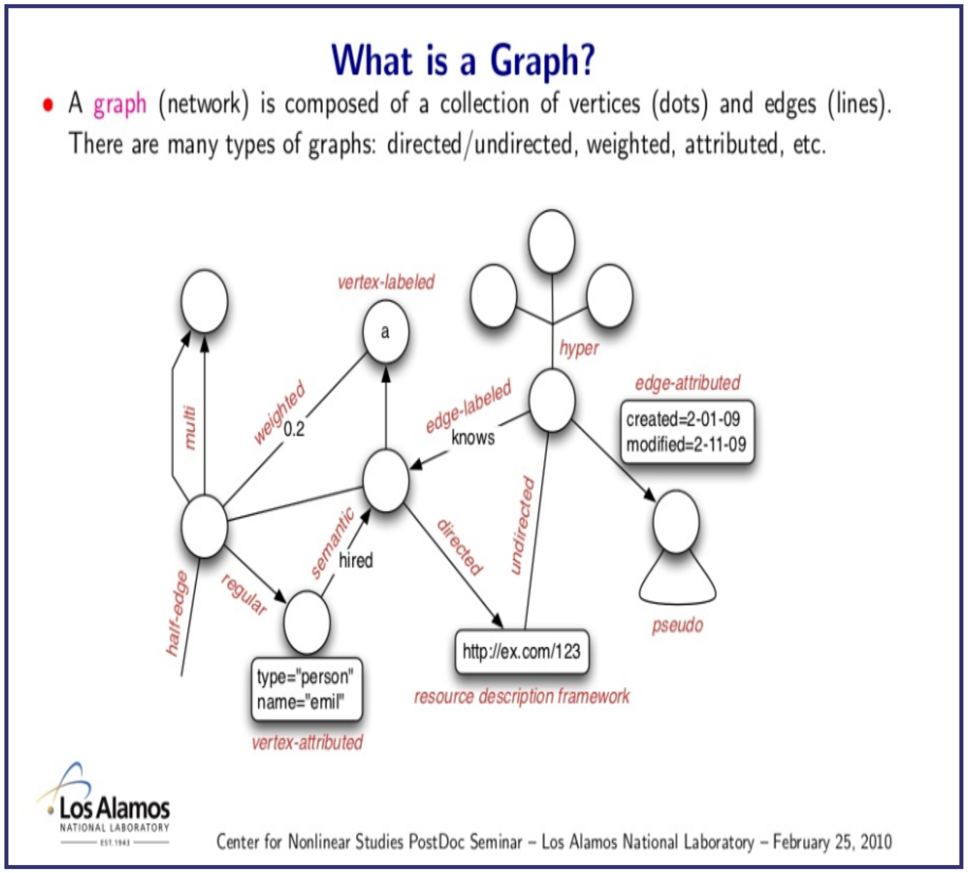
### The following snippets are for MongoDB.

```
// create a collection ("table")
db. createCollection (, {< configuration parameters >})



// a sample doc
{
title : " MongoDB ",
last_editor : "172.5.123.91" ,
last_modified : new Date ("9/23/2010") ,
body : " MongoDB is a..." ,
categories : [" Database ", " NoSQL ", " Document Database "] ,
reviewed : false
}

// add a doc into a coll
db.< collection >. insert ( { title : " MongoDB ", last_editor : ...
} );
```

```
// retrieve
db.< collection >. find ( { categories : [ " NoSQL ", " Document
Databases " ] } );

db.< collection >. find ( { title : " MongoDB " );

db.< collection >. find ( { where:" this .a ==1" } );

// array size comparison
{ categories : { size:2} }

// results processing
db.< collection >. find ( ... ). sort ({< field >: <1| -1 >}). limit
(). skip ();

// potentially DANGEROUS!
db. eval ( function (< formal parameters >) { ... }, );

// MapReduce
db.< collection >. mapreduce ( map : ,
reduce : < reduce - function >,
query : < selection criteria >,
sort : < sorting specificiation >,
limit : < number of objects to process >,
out: ,
outType : <" normal "|" merge "|" reduce ">,
keeptemp : < true |false >,
finalize : < finalize function >,
scope : < object with variables to put in global namespace >,
verbose : < true |false >
);
```

## Graph DBs

A graph is a data structure comprised of vertices and edges:

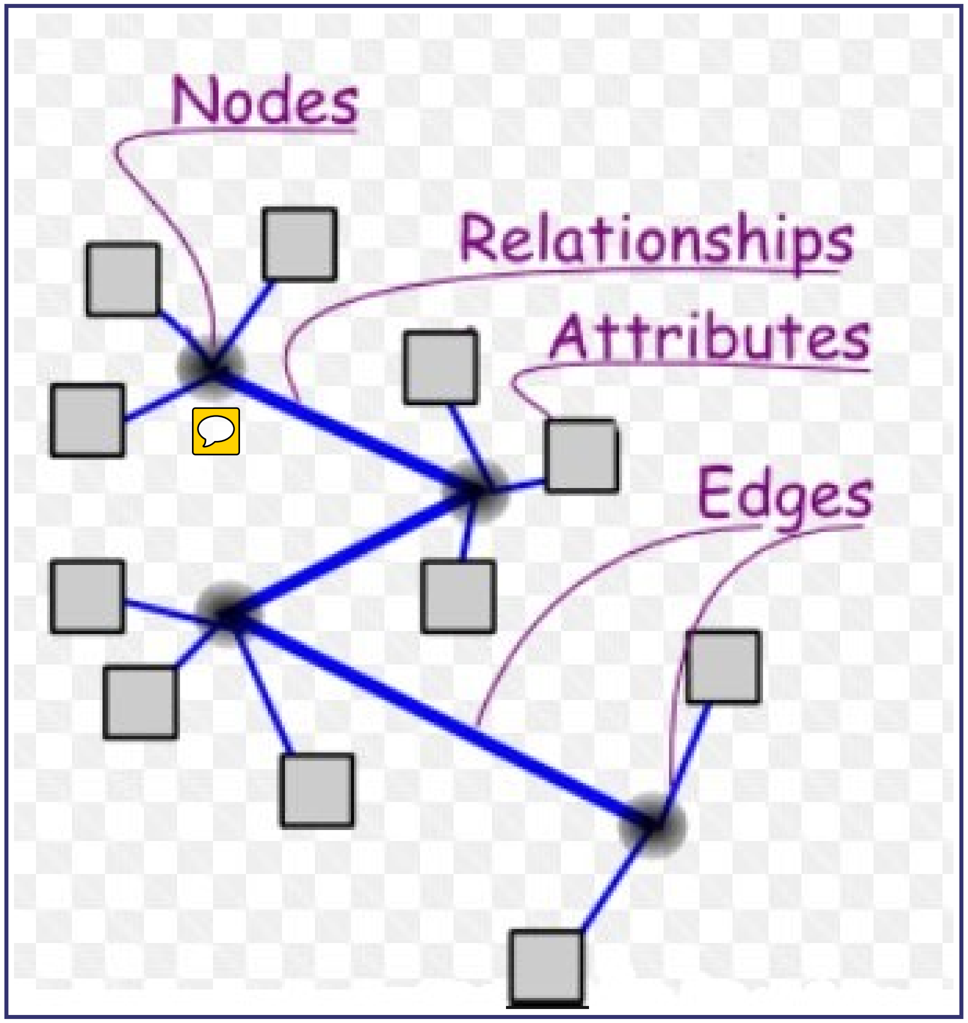

What is a Graph?

- A graph (network) is composed of a collection of vertices (dots) and edges (lines). There are many types of graphs: directed/undirected, weighted, attributed, etc.

Center for Nonlinear Studies PostDoc Seminar – Los Alamos National Laboratory – February 25, 2010

A graph database uses (contains) graph entities such as nodes (vertices), relations (edges), and properties (k-v pairs) on vertices

## and edges, to store data.



## Graph DBs: index-free adjacency

A graph DB is said to be 'index free', since each node directly stores pointers to its adjacent nodes.

In a graph db, the focus is on relationships between 'linked data'.

## Graph DBs: multiple types of graphs

A rich variety of graphs can be stored and manipulated - directed graphs, trees, weighted graphs, hypergraphs, etc.

## Graph DBs: implementations

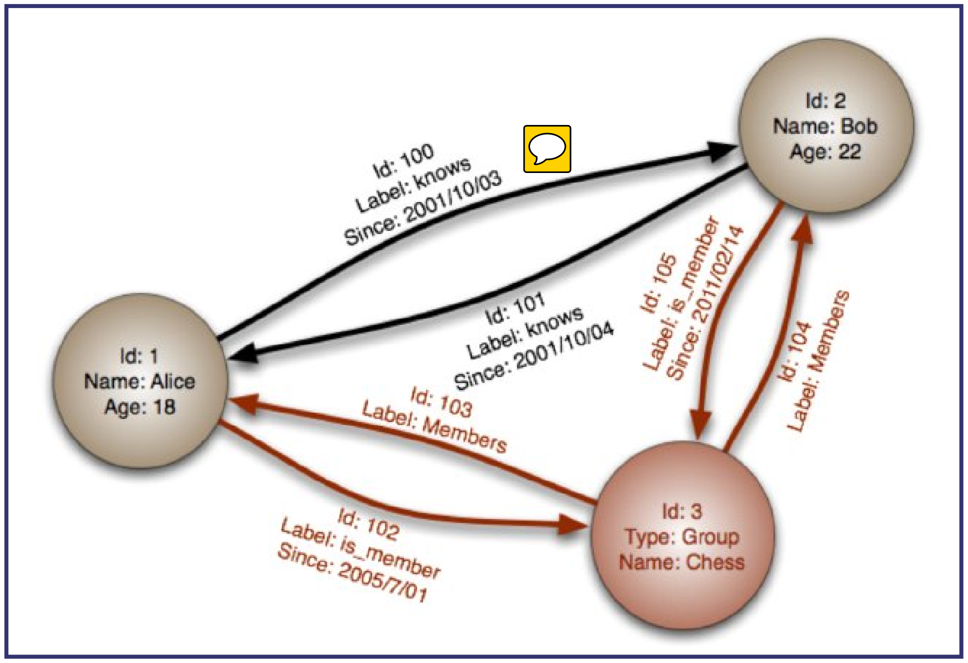
These are popular:

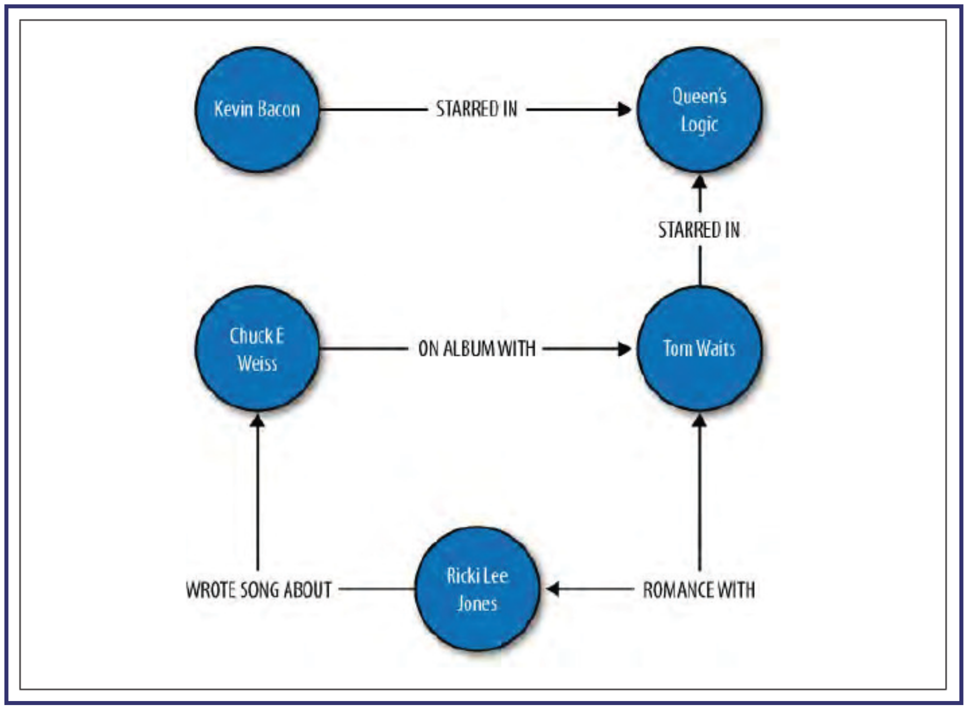
- FlockDB (from Twitter)
- Neo4J 💬
- HyperGraphDB
- InfiniteGraph
- InfoGrid (makers of MeshBase and NetMeshBase)
- OrientDB [wasn't this also listed as a document DB?!]
- Giraph (from Apache)
- GraphLab

Uses: social networks, recommendation engines..

Gartner: "Graph analysis is possibly the single most effective competitive differentiator for organizations pursuing data-driven operations and decisions, after the design of data capture."

# Graph DBs: a couple of sample graphs



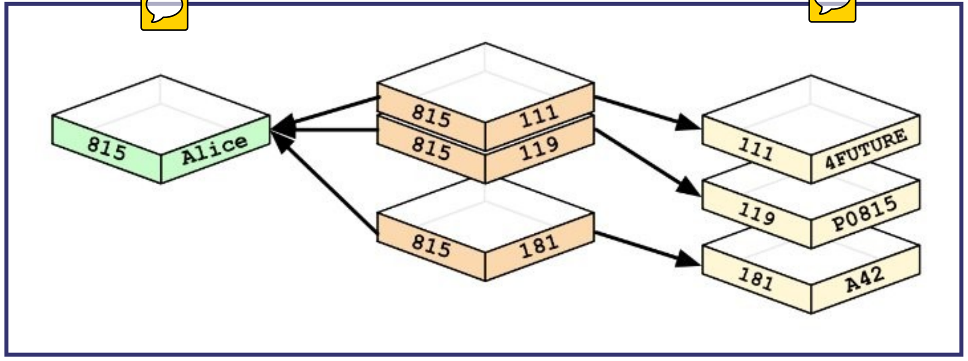


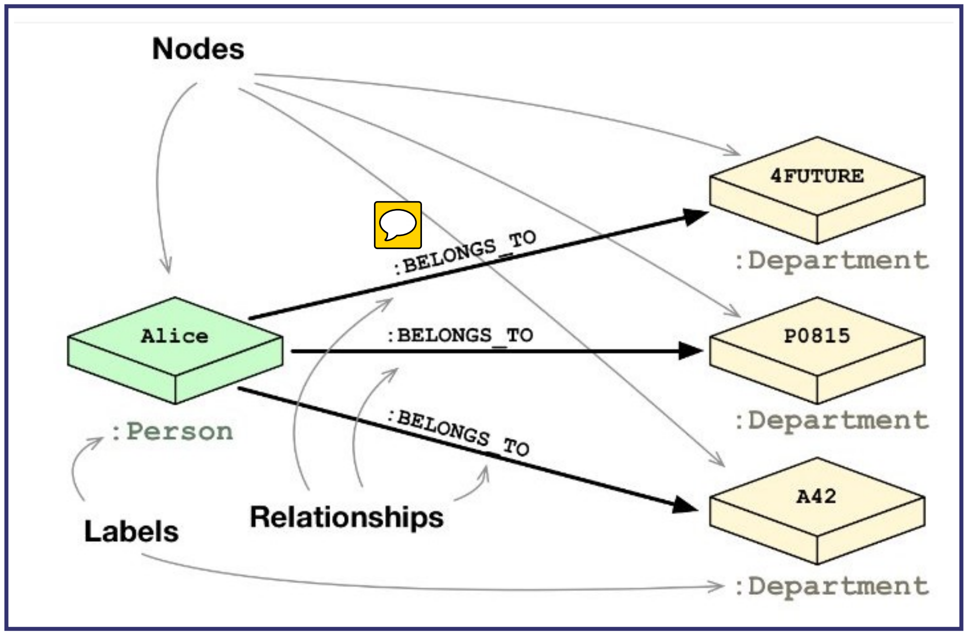
Note that in addition to internal names/IDs, nodes and edges can have properties (attributes and their values, ie. k/v pairs).

## GraphDBs: comparisons with relational modeling

Compared to a relational scheme, a graph offers a compact, normalized, intuitive way of expressing connections/relationships.

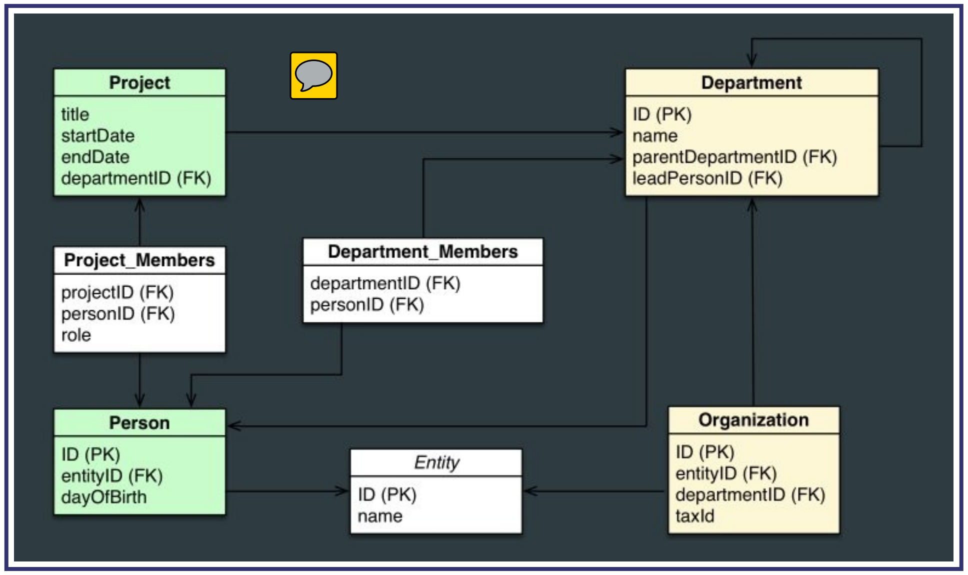
Here is a relational way to express employee-dept relations, and the corresponding graph-based way:

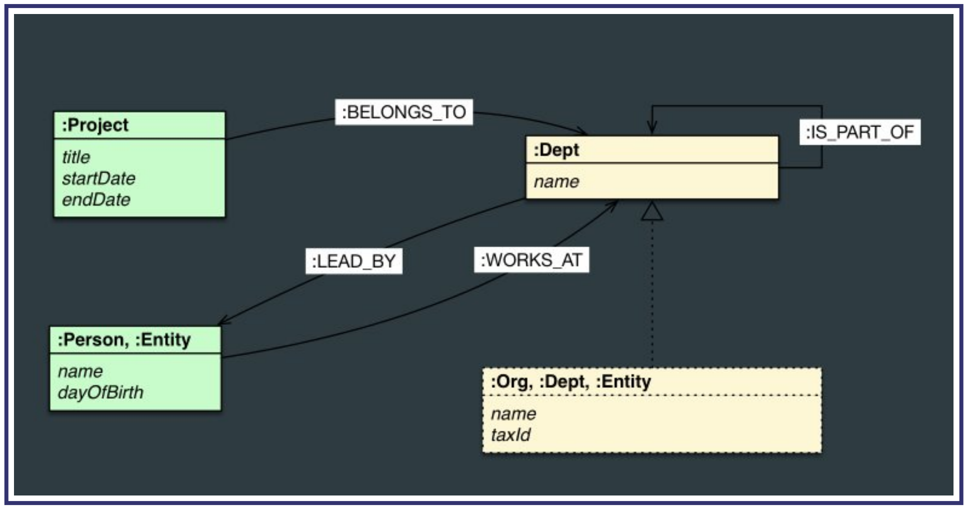




Each row in a table becomes a node, and columns (and their values), node properties.

## Here is an E-R diagram, and a graph version:





## As you can see, the graphs help model relationships in the form of connections between nodes.

## Graph DBs: querying

It is pretty straightforward to create nodes and edges, and attach properties to them; after that, it is equally simply to do queries on the resulting database.

In Neo4J, we use Cypher, a declarative graph query language - modeled after SQL, augmented with graph-specific functionality.

In the following, we compare a SQL query, and its Cypher equivalent:

```
SELECT name FROM Person
LEFT JOIN Person_Department
  ON Person.Id = Person_Department.PersonId
LEFT JOIN Department
  ON Department.Id = Person_Department.DepartmentId
WHERE Department.name = "IT Department"

MATCH (p:Person)<-[:EMPLOYEE]-(d:Department)
```

```
WHERE d.name = "IT Department"
RETURN p.name
```

## Using the Neo4J JDBC driver, a query to return "John's dept" can be embedded in Java:

```
Connection con =
DriverManager.getConnection("jdbc:neo4j://localhost:7474/");

String query =
    "MATCH (:Person {name:{1}})-[:EMPLOYEE]-(d:Department) RETURN
d.name as dept";
try (PreparedStatement stmt = con.prepareStatement(QUERY)) {
    stmt.setString(1,"John");
    ResultSet rs = stmt.executeQuery();
    while(rs.next()) {
        String department = rs.getString("dept");
        ....
    }
}
```

## GraphDBs: TinkerPop (Gremlin)

TinkerPop is a very interesting graph traversal language (actually, an entire graph computing framework). Here is how to get started on exploring it.

Aside: here is an interesting use case, for a graph DB.

## What is a triple store?

A triple store (or triplestore, or RDF) database stores triples of (subject,predicate,object) [or equivalently, (class,attribute,value)]. It is where the predicate is given equal status to subject and object [upcoming examples will make this clear].

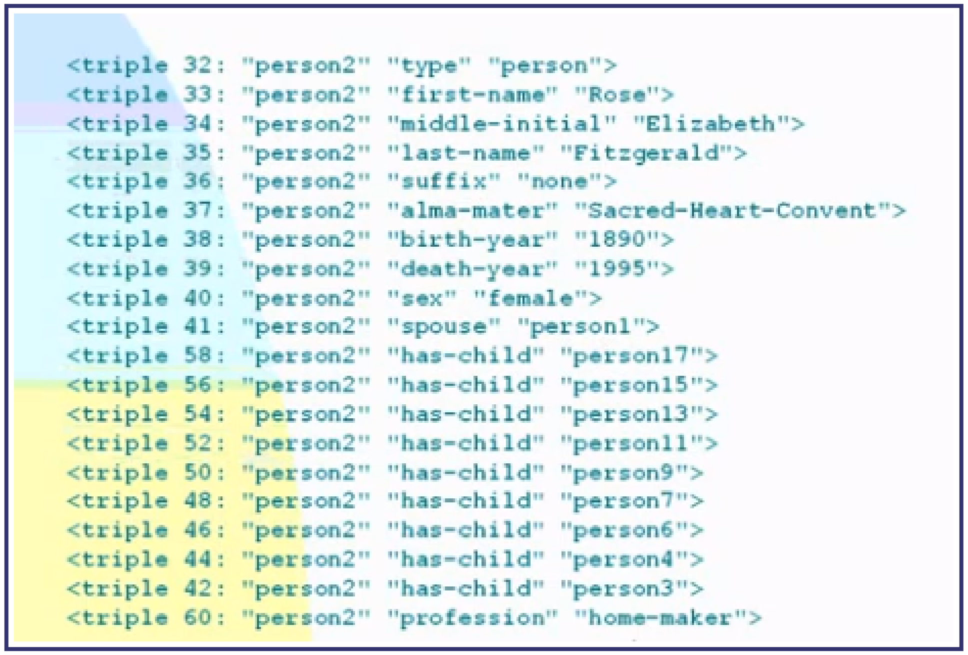
As an aside, if a fourth attribute (context) is also stored, then we'd call the DB a quad store, or 'named graph'. There are even 'quints' (with an extra 'name' or 'ID' attribute).

We issue 'semantic queries' to search a triple store DB.

Note that a triple store DB is a restricted form of a graph DB.

## Triple store: an example

Here is an example of a triple store - it is a flat (non-hierarchical) list (bag) of triplets, specified as subject (node), predicate (relationship), object (another node). The column on the left shows node IDs, that's not part of the triple.



```
<triple 32: "person2" "type" "person">
<triple 33: "person2" "first-name" "Rose">
<triple 34: "person2" "middle-initial" "Elizabeth">
<triple 35: "person2" "last-name" "Fitzgerald">
<triple 36: "person2" "suffix" "none">
<triple 37: "person2" "alma-mater" "Sacred-Heart-Convent">
<triple 38: "person2" "birth-year" "1890">
<triple 39: "person2" "death-year" "1995">
<triple 40: "person2" "sex" "female">
<triple 41: "person2" "spouse" "person1">
<triple 58: "person2" "has-child" "person17">
<triple 56: "person2" "has-child" "person15">
<triple 54: "person2" "has-child" "person13">
<triple 52: "person2" "has-child" "person11">
<triple 50: "person2" "has-child" "person9">
<triple 48: "person2" "has-child" "person7">
<triple 46: "person2" "has-child" "person6">
<triple 44: "person2" "has-child" "person4">
<triple 42: "person2" "has-child" "person3">
<triple 60: "person2" "profession" "home-maker">
```

The beauty is that such a triplet list can be grown 'endlessly', eventually connecting EVERYTHING to EVERYTHING ELSE! There is no schema to modify - just keep adding triplets to the DB!

## Triple store database implementations

- AllegroGraph
- MarkLogic
- SparkleDB
- Stardog (http://stardog.com/)

## Triplet store DBs: querying

Querying a triplet store can be done in one of several RDF (what is THAT?) query languages, eg. RDQL, SPARQL, RQL, SeRQL, Versa.. Of these, SPARQL is currently the most popular.

The output of a triple store query is called a 'graph'.

Queries tend to span disparate data, perform complex rule-based logic processing or inference chaining (AI-like).

Eg. given

```
    :human rdfs:subClassOf :mammal
    :man rdfs:subClassOf :human
```

an RDF database can infer

```
    :man rdfs:subClassOf :mammal
```

## If you want LARGE triple store datasets to play with, look at DBpedia, which is all of Wikipedia in RDF form!

## Triple store DBs: equivalent to a form of RDF databases!

RDF (Resource Description Framework) is a metadata data model - a set of specifications (from W3C), for modeling information, for use in KM applications and the semantic web.

Specifically, one of RDF's serialization formats is N-Triples, which is *exactly* what store in our triple store DBs - subject,predicate,object. In that sense, every triple store DB is an RDF DB as well (but not the other way around).

If you want to play with RDF, try Sesame.

## Triple store DBs: architecture

These databases are set up to run in one of three modes:

- in-memory: triples are stored in main memory
- native store: persistence provided by the DB vendors, eg. AllegroGraph
- non-native store: uses third-party service, eg. Jena SDB uses MySQL as the store