Advanced Topics

Indexes, Distributed Systems, noSQL

Week 10: Database Systems (4CCS1DBS)

20 Mar 2017

Administrivia

- **Quizzes** —
  - Quiz 3 being graded.
  - Quiz 4 is today.

- **Relational Review Coursework** —
  - 15 Relational Algebra Problems
  - Feedback Only (not formally accessed)
  - NO Feedback on Late Submissions
  - Due Sunday April 2nd at 11:55pm

- **Mock Exam Review** Next Week Monday Lecture —
  - MCQ Questions in same format as May Exam
  - Take the Exam as a KEATs Quiz before next week
  - Solutions to Questions will be provided as a KEATs Quiz

Topics

- Improving Performance with Indexes
- Distributed Databases / CAP Theorem
- *noSQL* Databases
  - Key/Value
  - Columnar
  - Document
  - Graph

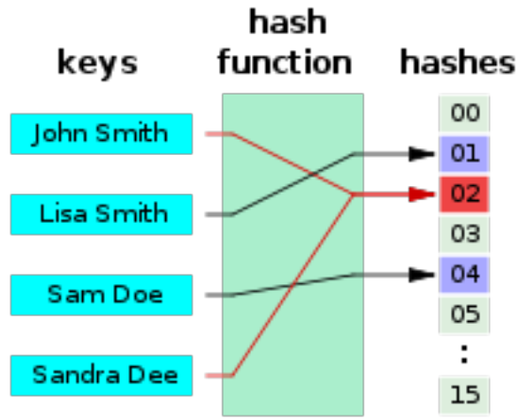
RELAX! *None of this material will be on the Final Exam!*

BUT it will be super important for your growth and a professional computer scientists…

Improving Performance with Indexes

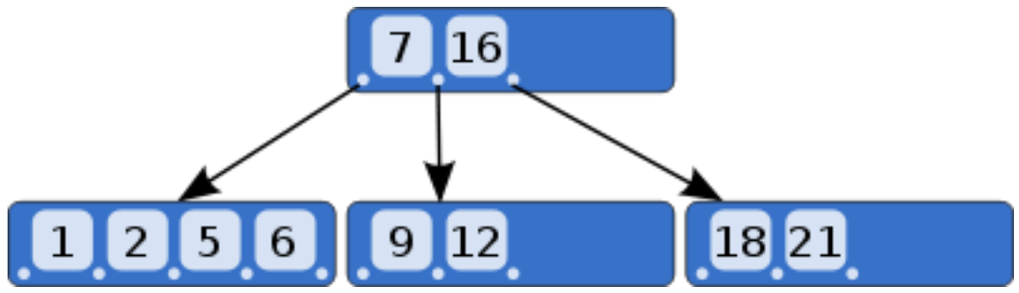
- More data, more time for queries to run.
- An **Index** is a special data structure built to avoid a full table scan when performing a query.

- A ***hash*** data structure allows for simple/fast access.



https://en.wikipedia.org/wiki/Hash_function

- A ***B-Tree*** *data structure* allows for more flexible ranged data indexing (i.e. less-than/greater-than/equals-to matching).

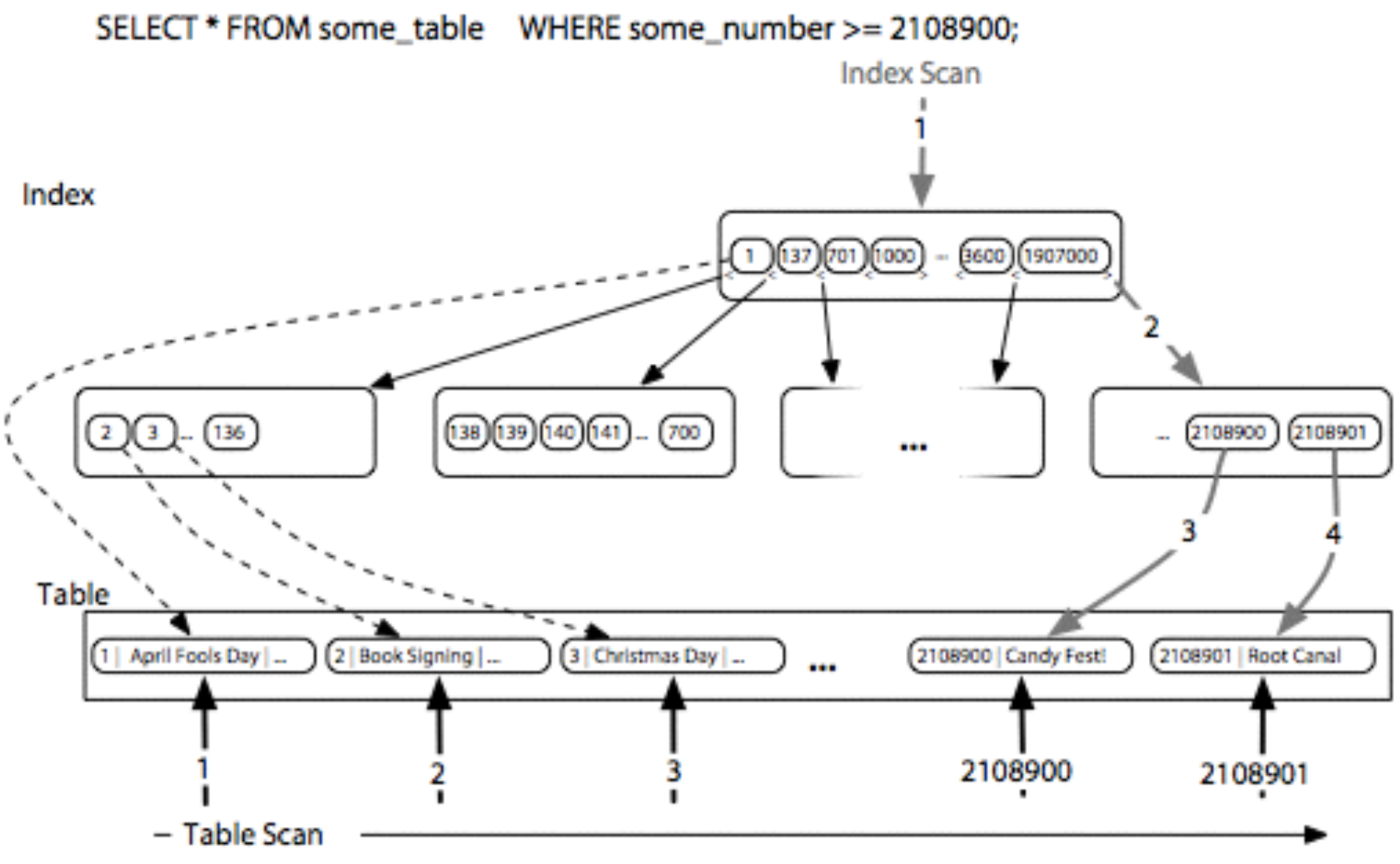


https://en.wikipedia.org/wiki/B-tree

- In SQL, often times `PRIMARY KEY / UNIQUE` will implicitly create an INDEX.

Index: Table Scan vs. Index Scan

```
CREATE INDEX some_table_idx
ON some_table (some_number)
USING BTREE;
```

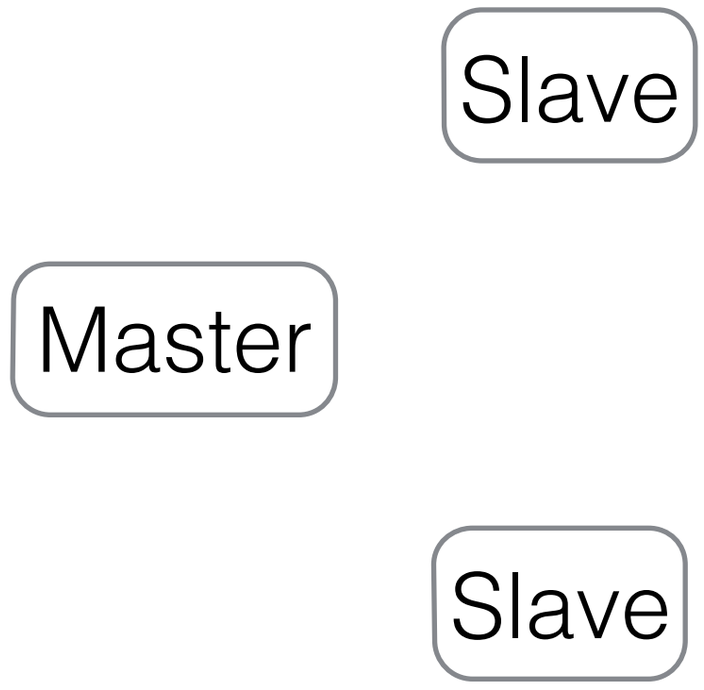


Improving Performance with Indexes

- Example in MySQL
  - Using EXPLAIN to see the query plan
  - CREATE INDEX in SQL

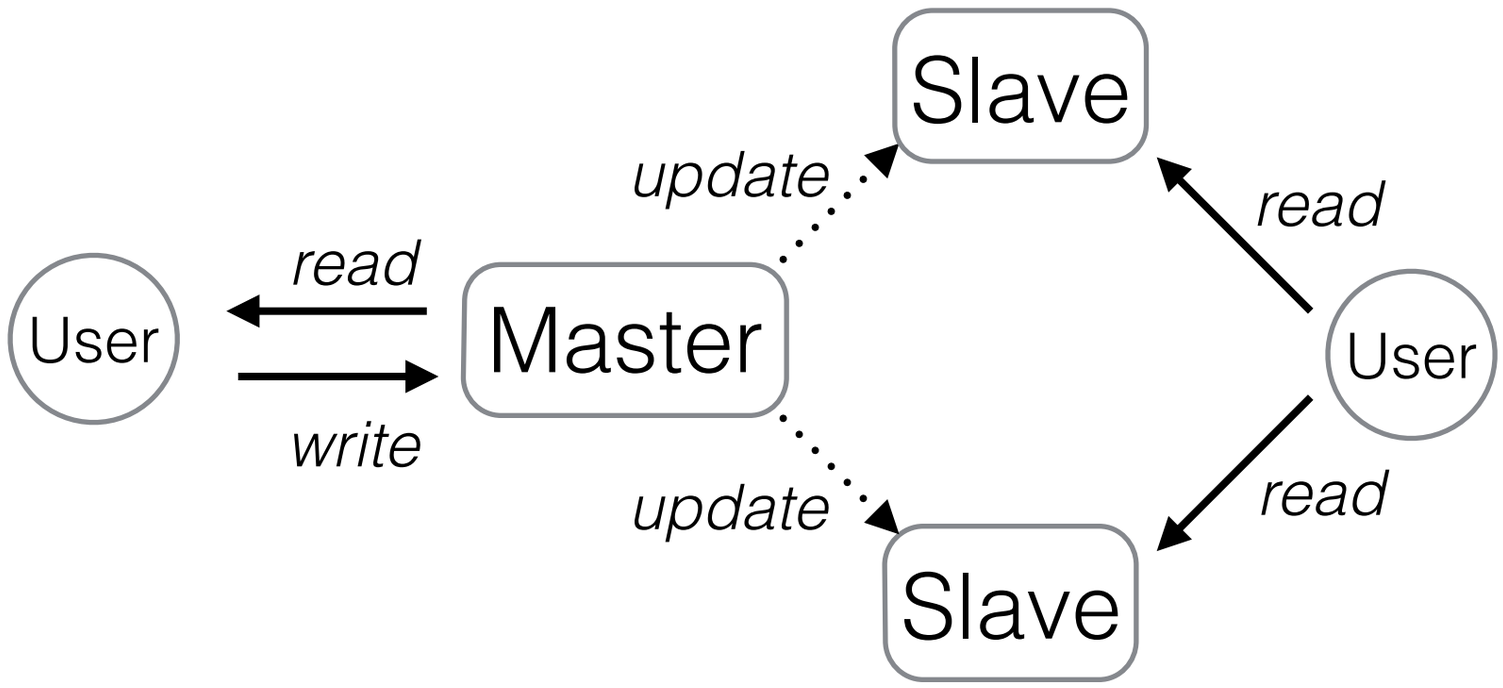
```
CREATE INDEX hours_idx
ON works_on (hours)
USING BTREE;
```

Distributed Databases



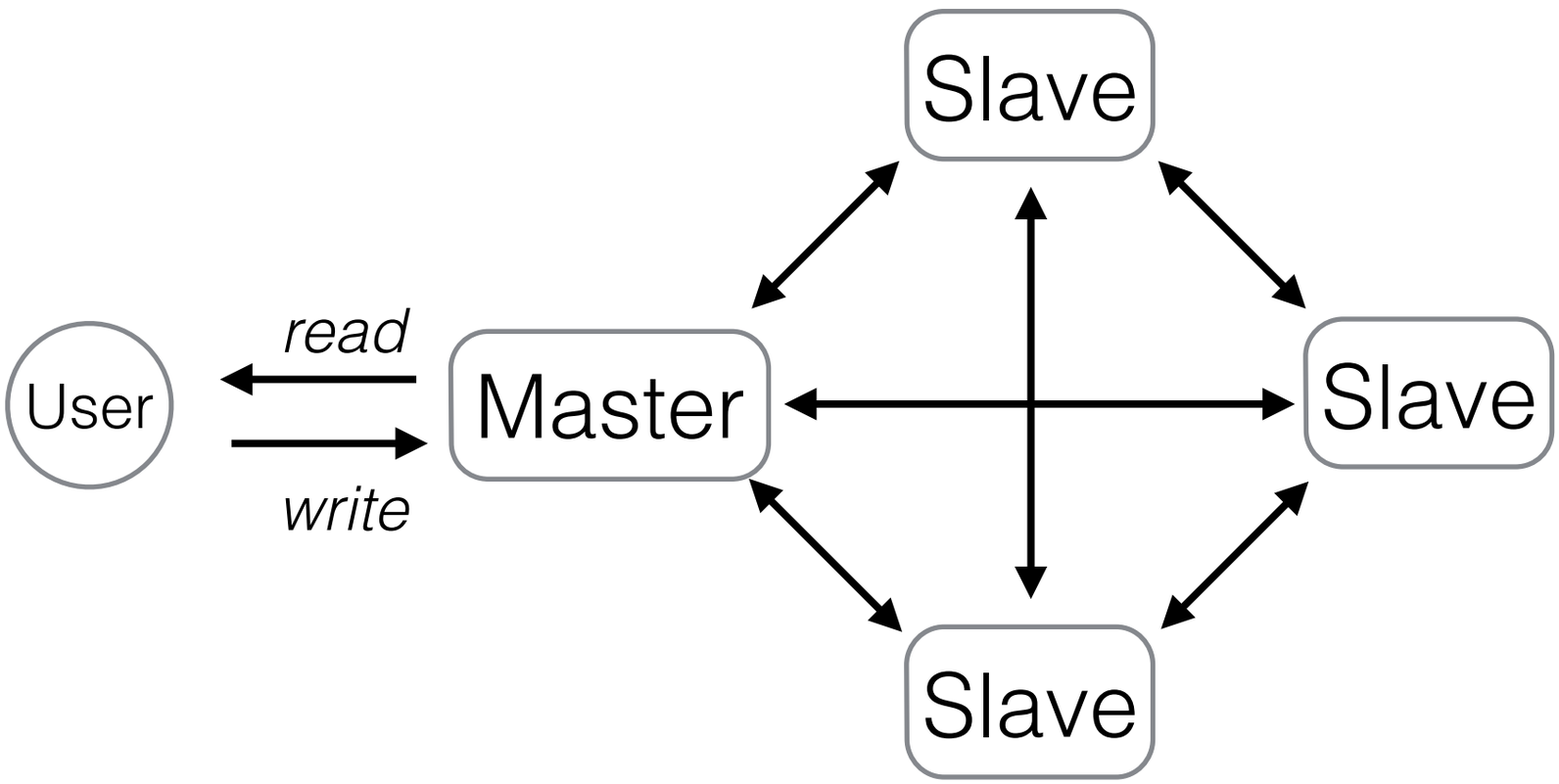
- Modern large database systems are *distributed systems*
  - *networked **nodes** (servers) each running an instance of the database systems*
- Master/Slave (Supervisor/Worker) architecture
- Two Distribution Strategies:
  1. **Replication**
  2. **Partitioning**

Distributed Databases - Replication



- ***Replication*** —identical replicas in different database nodes
  - Improved availability of data (query times / multi-site)
  - Parallelism for reading data
  - More overhead on updating data, keeping everything in sync
  - Fault Tolerance (Individual database nodes can crash)

Distributed Databases - Partitioning Data



- **Fragmentation/Partitioning** —splitting the data up across database nodes
  - *Horizontal* - distribute rows across different nodes
  - *Vertical* - table schema is split across different nodes (i.e. columns)
- BIG Data - a table cannot fit in-memory/on-disk
- Partitioning Data as an Index to improve query performance

Distributed Databases - Partitioning Data

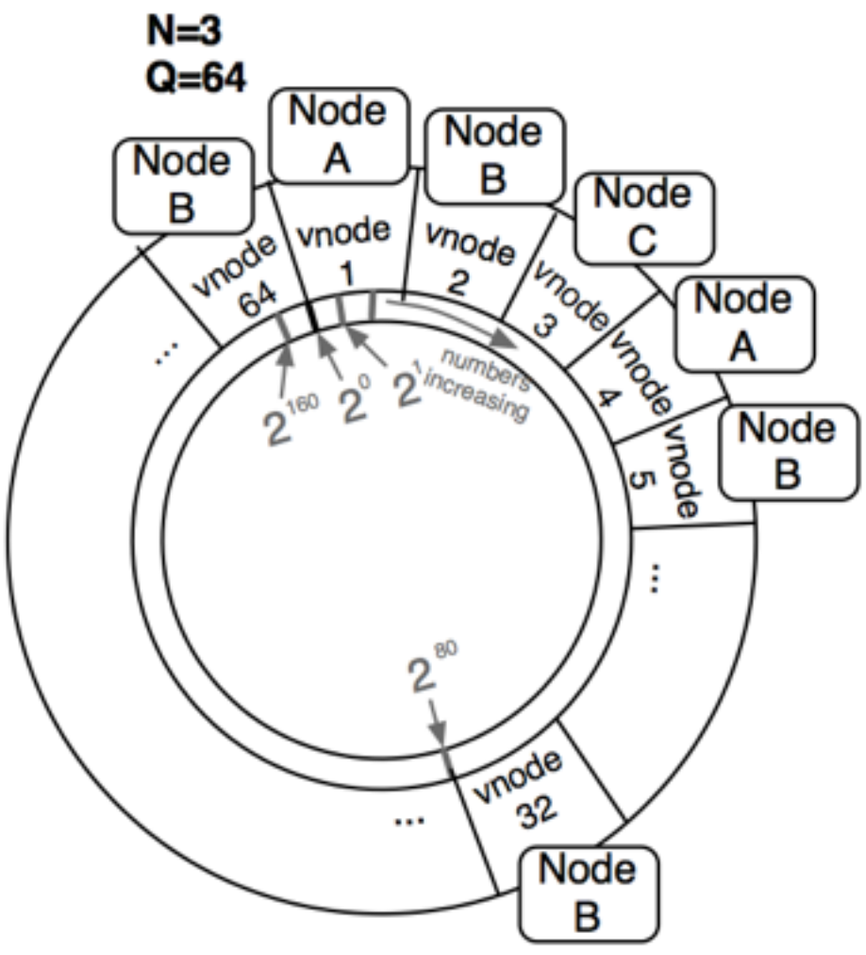


Figure 8—"The Riak ring" of sixty-four vnodes, assigned across three physical nodes

- **Example Partitioning Schema**— Riak server
  - server configuration ordained in a *ring* denoted by a 160-bit number
  - cluster is divided up into *partitions*, called virtual nodes (*vnodes*)
  - key is hashed to find a corresponding vnode

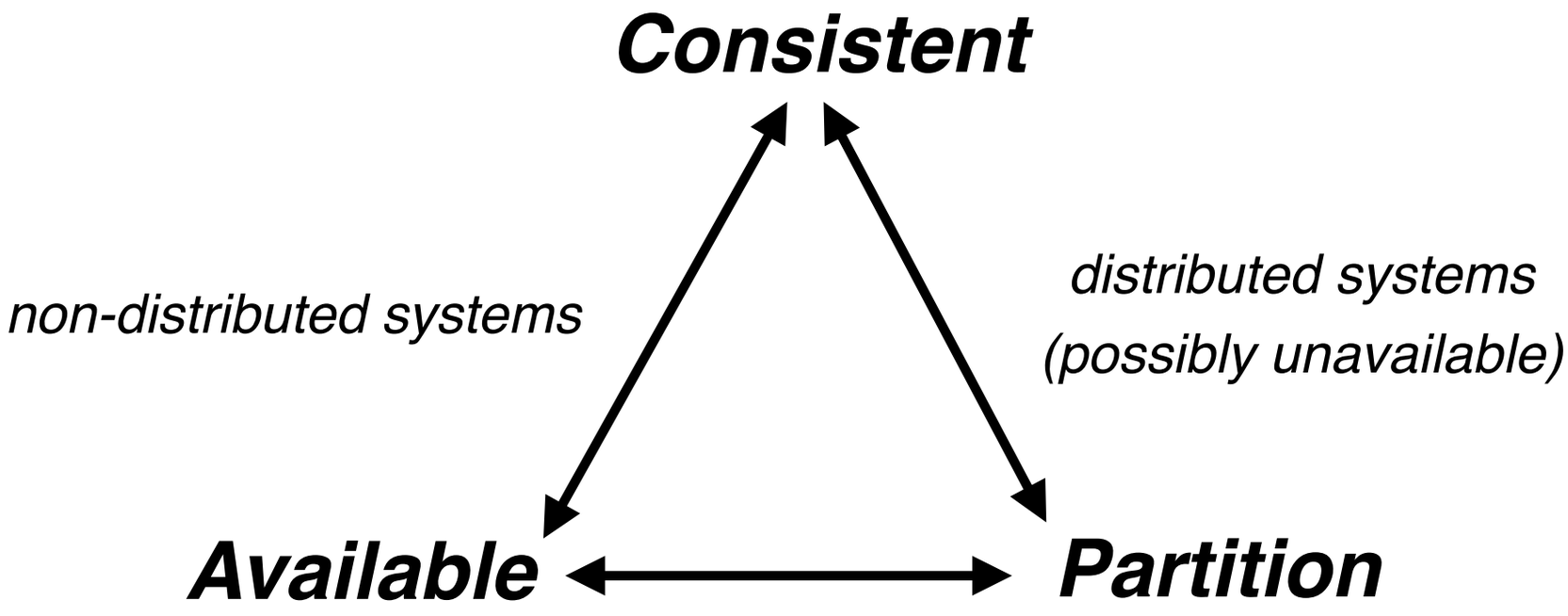
CAP Theorem

- How do distributed databases behave in the face of *network instability*?

- You can create a **distributed database** that is:
  - ***consistent***: writes are atomic and all subsequent requests retrieve the new value
  - ***available***: the database will always return a value as long as a single server is running
  - ***partition tolerant***: system will still function even if server communication is temporarily lost (i.e. a network partition)

- **CAP Theorem states that you can only pick 2 of the 3.**

CAP Theorem



- ***consistent*** -***available*** - are what traditional *non-distributed systems*
- For distributed systems, trade-off is between:
  - ***consistent-partition tolerant***
  - ***available-partition tolerant***
    - Can choose to be available have *eventual consistency*

CAP Theorem - Eventual Consistency

- **Example:** Imagine a database made of politically active people.
- It's 2005 and the query is *who is UK Prime Minister?*
- Years pass, it's 2007, *who is UK Prime Minister*?
- Someone goes on cruise, gets ship-wrecked on deserted isle… (**partitioned**!)
- 10 years pass (it's 2017), and they get rescued (yeah!)
- The first question they are asked is *who is UK Prime Minister*?
  - Answer depends on system availability behavior…

    answer or *decline to answer*

- **Now Suppose:** on changes to Prime Minister, bottles are thrown into the sea with the date and the update (i.e. (2010, "David Cameron"), ("2016", "Theresa May")
- Now upon rescue, you can answer (i.e. be available) with knowledge that you will eventually have a consistent answer

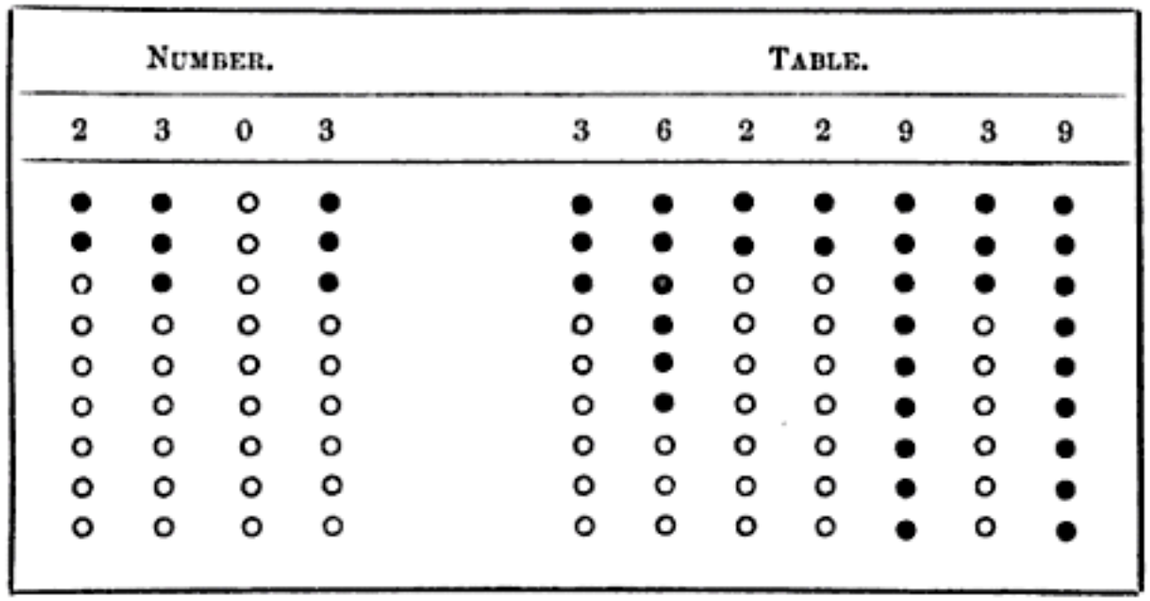
DNS Works with *eventual consistency*. A DNS server will always respond to a request. But a freshly registered domain will take time to get propagated.

noSQL - Alternatives to Relational Databases

- **noSQL** - ***"non-relational"*** database
  - Traditionally SQL / Relational DBs dominate in usage
  - Since ~2009, alternatives to Relational Database started appearing - Google / Amazon Web-Services / Start-ups
  - *BIG Data* requirements (how do you query more data that can be held in memory?) Google BigTable/MapReduce
  - *Schema-less* (being less strict with rigid table-based data models)
    - hierarchical data, heterogenous record-to-record variation
- Relaxing *Consistency* in trade-off toward *High Availability/ reducing latency* towards real-time web systems (i.e. Twitter)
- Optimizing for *graph data-structures* (difficult in Relational Databases)
- *polyglot persistence* module - using more than 1 type of database in a project/system
- paradigm shifts in *query languages* (i.e. Map-Reduce query model), embedded languages (i.e. Javascript)

Key-Value Databases

Data punch cards for Babbage Analytical Engine (1834-71)



Number.				Table.						
2	3	0	3	3	6	2	2	9	3	9

Proposed base-10 key-value pair for base-ten logarithm (Babbage Analytical Engine)

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

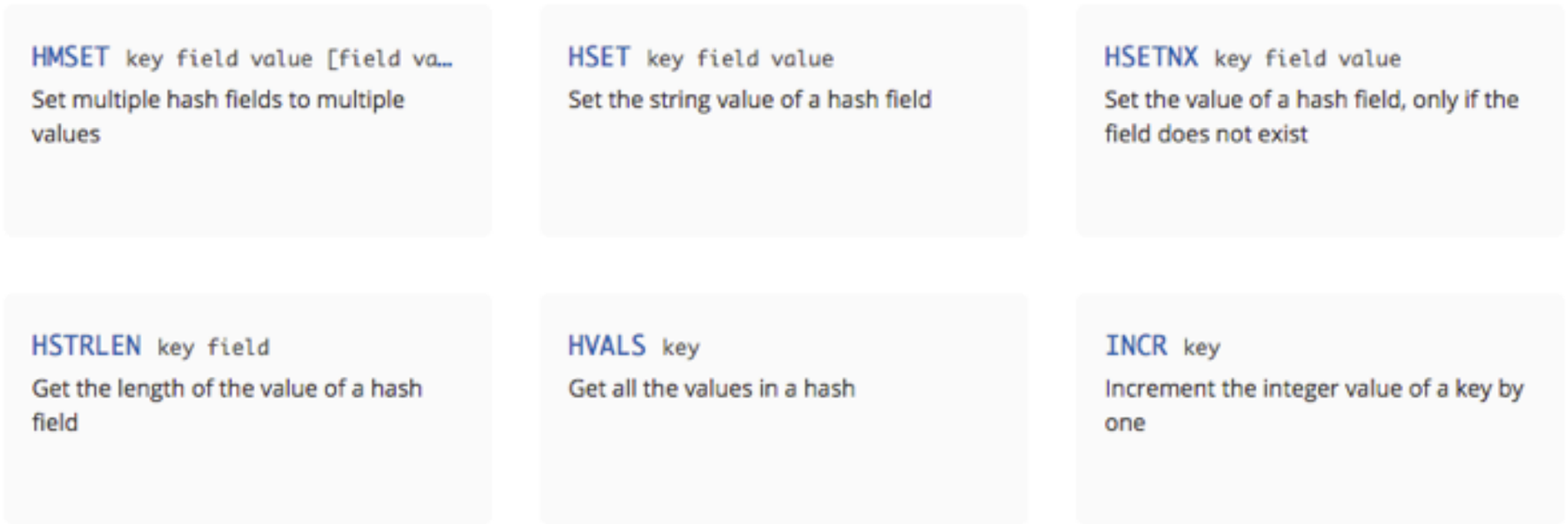
https://en.wikipedia.org/wiki/Key-value_database#/media/File:KeyValue.PNG

Key-Value Databases: REDIS

- Example: **REDIS (**http://redis.io/**)**

  - REmote DIctionary Service

- Trade durability for speed.
- Used as a blocking queue (or stack)
- Publish-subscribe system
- Many Commands **(**http://redis.io/commands**)**



- Famous for use by Twitter for timeline of tweets
  - (300k/sec in 2012)
- **Try it out:** http://try.redis.io/

Key-Value Databases

- **Key-Value (KV**) maps simple keys to (possibily) more complex values like a huge hashtable (associative array).
- Relatively simple - primitive to complex data-types
- Fast.
- Easily distributed - Horizontally scalable.
- Data not related (i.e. no JOINs) (i.e. web sessions per user)

Draw-backs:

- Lack indexes and good scanning capabilities.
- Not good for relational data (i.e. doing a JOIN)
- Only basic CRUD (Create, Read, Update, Delete) operations
- Lacks more complex queries (i.e. Aggregates and Group By ops)

Columnar (Column-oriented / column-family)

- In **row-oriented databases**, data is typically stored by **row** in blocks on disk.

SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL

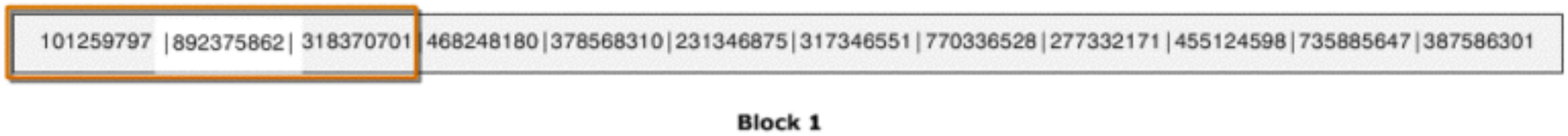
- Disk-access locality is optimized for row access
  - Blocks contain one or more rows
- Great for **OLTP** (Online Transaction Processing)
  - i.e. updating a Bank Account record (which requires all columns)
- Frequent reading and writing an entire record of data.

http://docs.aws.amazon.com/redshift/latest/dg/c_columnar_storage_disk_mem_mgmnt.html

Columnar (Column-oriented / column-family)

- In **columnar data storage**, each data block stores values of a single column for multiple rows.

SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL



- Disk-access locality is optimized for ***column*** access.
  In this example, a Block contains 3 times the amount of data for a column as the previous example.
- Great for **OLAP** (Online Analytic Processing), for analyzing columns across multiple records.
  — e.g. web-site statistics (click streams etc…)

Columnar (Column-oriented / column-family)

- **Advantages**: Blocks easily Compressed because data is very similar (i.e. same column).

Encoding type	Keyword in CREATE TABLE and ALTER TABLE	Data types
Raw (no compression)	RAW	All
Byte dictionary	BYTEDICT	All except BOOLEAN
Delta	DELTA DELTA32K	SMALLINT, INT, BIGINT, DATE, TIMESTAMP, DECIMAL INT, BIGINT, DATE, TIMESTAMP, DECIMAL
LZO	LZO	All except BOOLEAN, REAL, and DOUBLE PRECISION
Mostly*n*	MOSTLY8 MOSTLY16 MOSTLY32	SMALLINT, INT, BIGINT, DECIMAL INT, BIGINT, DECIMAL BIGINT, DECIMAL
Run-length	RUNLENGTH	All
Text	TEXT255 TEXT32K	VARCHAR only VARCHAR only
Zstandard	ZSTD	All

http://docs.aws.amazon.com/redshift/latest/dg/c_columnar_storage_disk_mem_mgmnt.html

Columnar (Column-oriented / column-family)

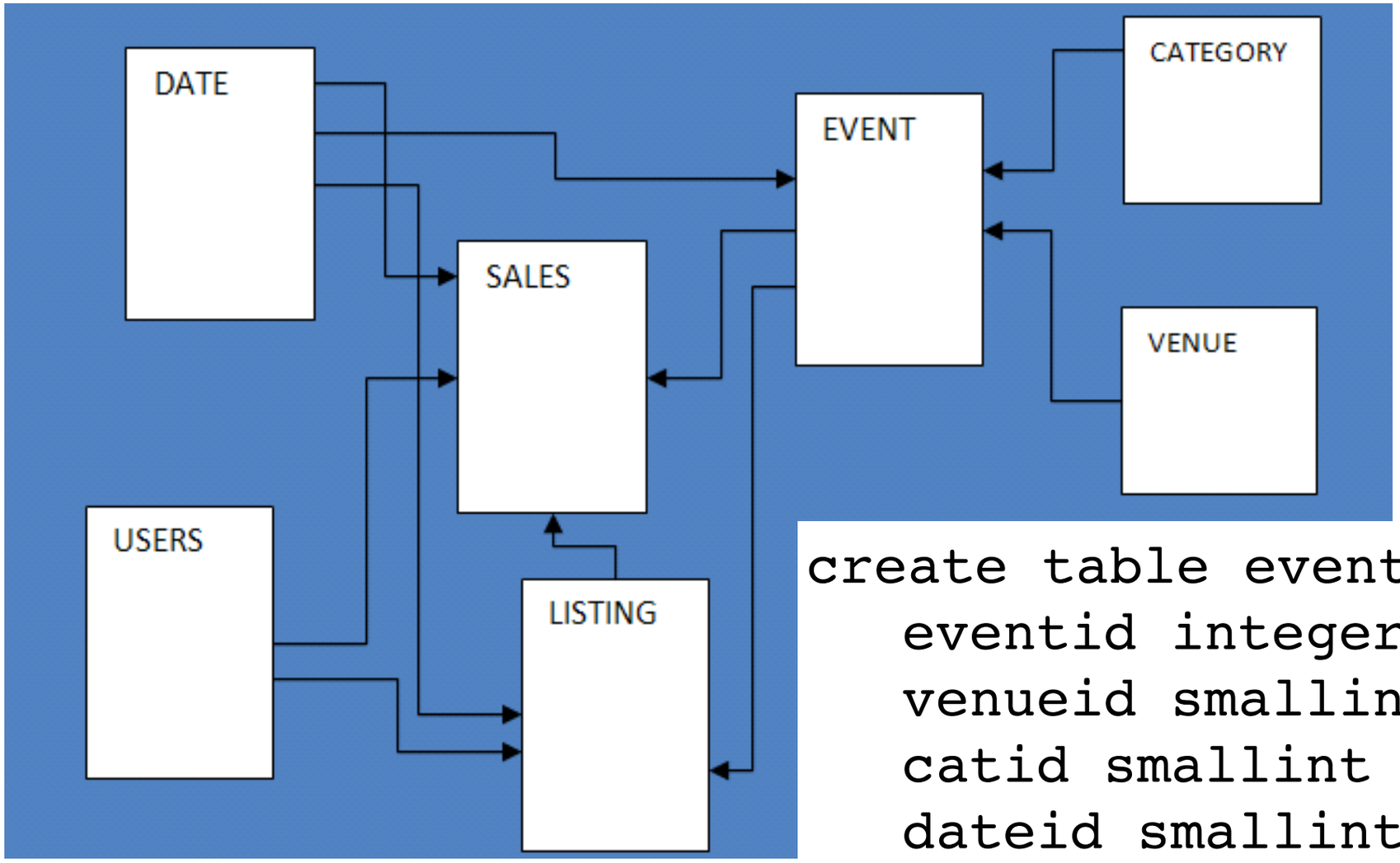
- Example: **HBase (**https://hbase.apache.org/**)**
- **Part of *Hadoop*** ecosystem (Big Data)
- Modeled after Google Bigtable

http://research.google.com/archive/bigtable.html

row keys	column family "color"	column family "shape"
row "first"	"red": "#F00" "blue": "#00F" "yellow": "#FF0"	"square": "4"
row "second"		"triangle": "3" "square": "4"

Columnar (Column-oriented / column-family)

- Example: **Amazon Redshift**
- SQL Syntax - annotate columns with **distribution keys** and **sort keys**




```
create table event(
   eventid integer not null distkey,
   venueid smallint not null,
   catid smallint not null,
   dateid smallint not null sortkey,
   eventname varchar(200),
   starttime timestamp);
```

Columnar (Column-oriented / column-family)

- Example: **Amazon Redshift**

Query-Plan: http://docs.aws.amazon.com/redshift/latest/dg/c-the-query-plan.html

Simple Query not using distributed keys

```
explain select eventname, count(*) from event group by eventname;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
XN HashAggregate  (cost=131.97..133.41 rows=576 width=17)
  ->  XN Seq Scan on event  (cost=0.00..87.98 rows=8798 width=17)
```

Columnar (Column-oriented / column-family)

- Example: **Amazon Redshift**

Query-Plan: http://docs.aws.amazon.com/redshift/latest/dg/c-the-query-plan.html

Joining tables using the distribution and sort keys

```
explain select * from sales, listing, event
where sales.listid = listing.listid and sales.eventid = event.eventid;
                                  QUERY PLAN
------------------------------------------------------------------------------
XN Hash Join DS_BCAST_INNER  (cost=109.98..3871130276.17 rows=172456 width=132)
  Hash Cond: ("outer".eventid = "inner".eventid)
  ->  XN Merge Join DS_DIST_NONE  (cost=0.00..6285.93 rows=172456 width=97)
        Merge Cond: ("outer".listid = "inner".listid)
        ->  XN Seq Scan on listing  (cost=0.00..1924.97 rows=192497 width=44)
        ->  XN Seq Scan on sales  (cost=0.00..1724.56 rows=172456 width=53)
  ->  XN Hash  (cost=87.98..87.98 rows=8798 width=35)
        ->  XN Seq Scan on event  (cost=0.00..87.98 rows=8798 width=35)
```

Columnar (Column-oriented / column-family)

- **Columnar databases** store like data by columns, rather than keeping data together by rows.
- Like KV, rows are queried by matching keys
- Like relations, rows are are groups of zero or more columns.
- Adding Columns are *trivial*.
- Versioning of data is often built-in.
- Good for “Big Data” problems, largely distributed.
- Data Warehousing: reading very large number of rows for a small number of columns (i.e. selecting 5 columns from a 100 column table = 5% table access, great when there are a billion rows).
- Indexing web-pages is a classic use case.

Common Drawbacks:

- Design of schema based on ***how the data*** will be queried
- Not good for fast *ad hoc* reporting
- Not the best solution for **OLTP** (Online Transaction Processing), where all columns in a row are often accessed and updated.

Document-oriented databases

user document

```
{
  _id: <ObjectId1>,
  username: "123xyz"
}
```

contact document

```
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone: "123-456-7890",
  email: "xyz@example.com"
}
```

access document

```
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```

Normalized - Schema

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

Embedded sub-document

Embedded sub-document

Embedded - Schema

Document databases

- Example: **MongoDB (**https://www.mongodb.org/**)**
- Large, distributed document store "*huMONGOus*"
- Data Modeling Patterns for relationships

https://docs.mongodb.org/manual/applications/data-models/

- JSON file storage
- Javascript language is native to database

```
> printjson( db.towns.findOne({"_id" : ObjectId("4d0b6da3bb30773266f39fea")}) )
{
        "_id" : ObjectId("4d0b6da3bb30773266f39fea"),
        "country" : {
                "$ref" : "countries",
                "$id" : ObjectId("4d0e6074deb899521 6a8309e")
        },
        "famous_for" : [
                "beer",
                "food"
        ],
        "last_census" : "Thu Sep 20 2007 00:00:00 GMT        -0700 (PDT)",
        "mayor" : {
                "name" : "Sam    Adams",
                "party" : "D"
        },
        "name" : "Portland",
        "population" : 582000,
        "state" : "OR"
}
```

Collection

Database

Identifier

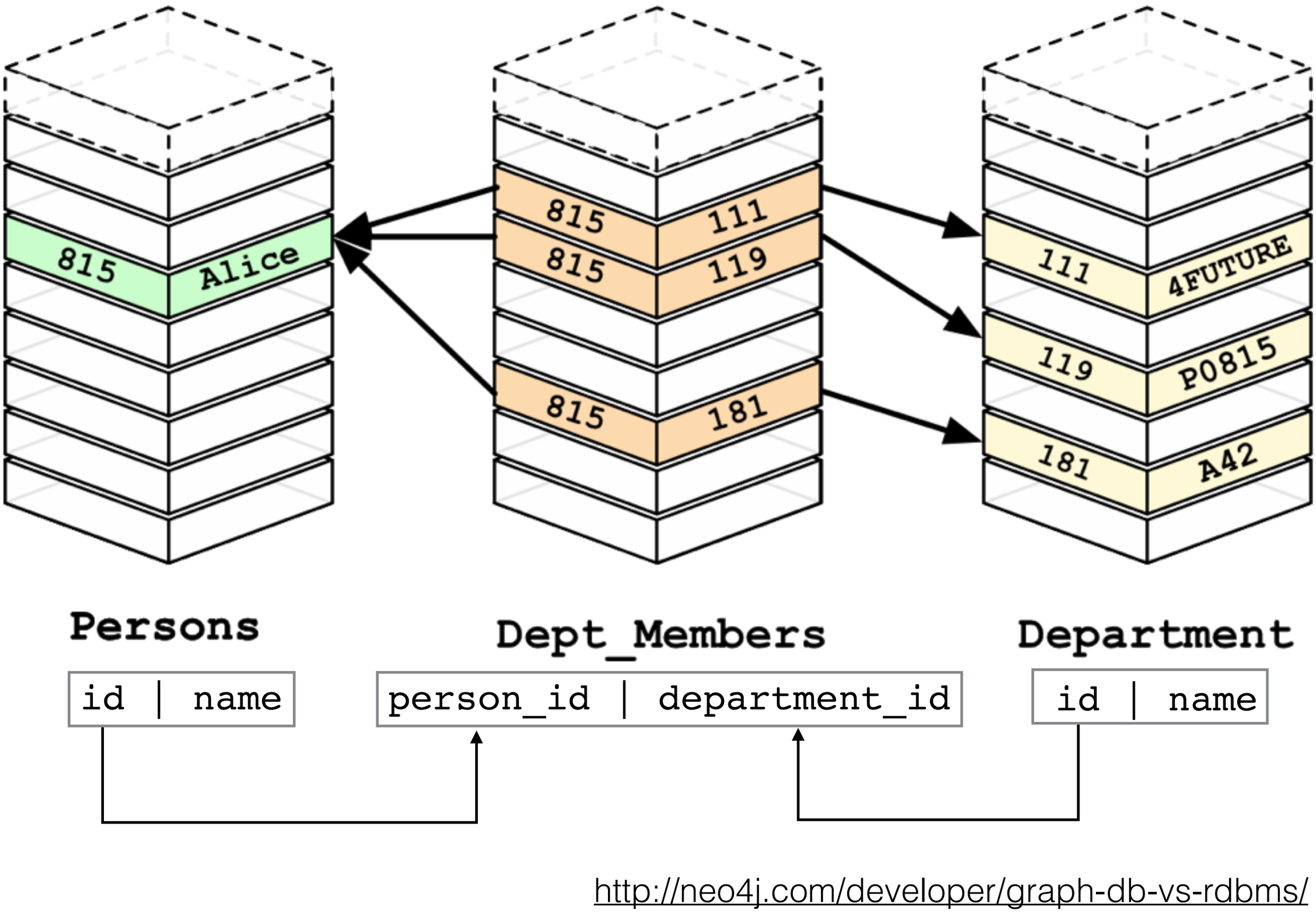
Document

Document databases

- **Document databases** allow for any number of fields per object and objects to be nested as value of other fields.
- Assume a standard file-encodings per object: XML, YAML, JSON, binary forms BSON.
- Tables -> Collections of Documents
- Objects referenced by unique *key.*
- Expressive query language/API is typical (i.e. Javascript)
- Great for highly variable domains — unsure of what the schema will be.
- Good for object-oriented programing model (less impedance mismatch between database and application layer)

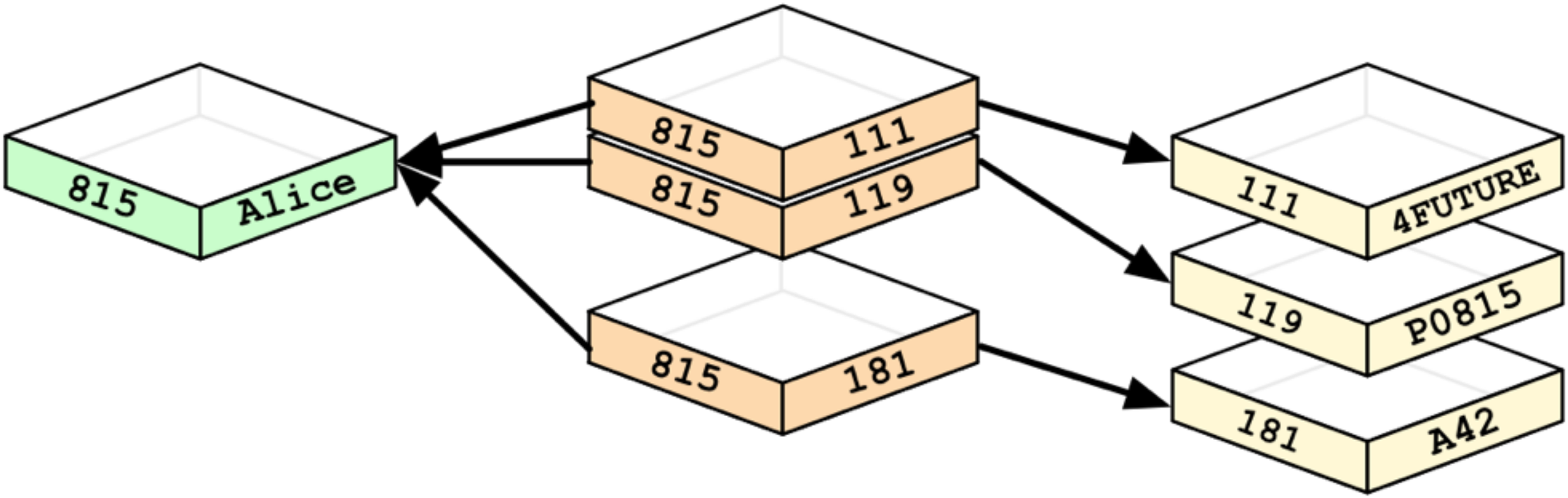
- **Drawbacks**:
  - Elaborate join queries on highly normalized relation data
  - Document should mostly be contained with everything relevant
  - Denormalizing the data is common (redundant embedded data)

Graph Databases



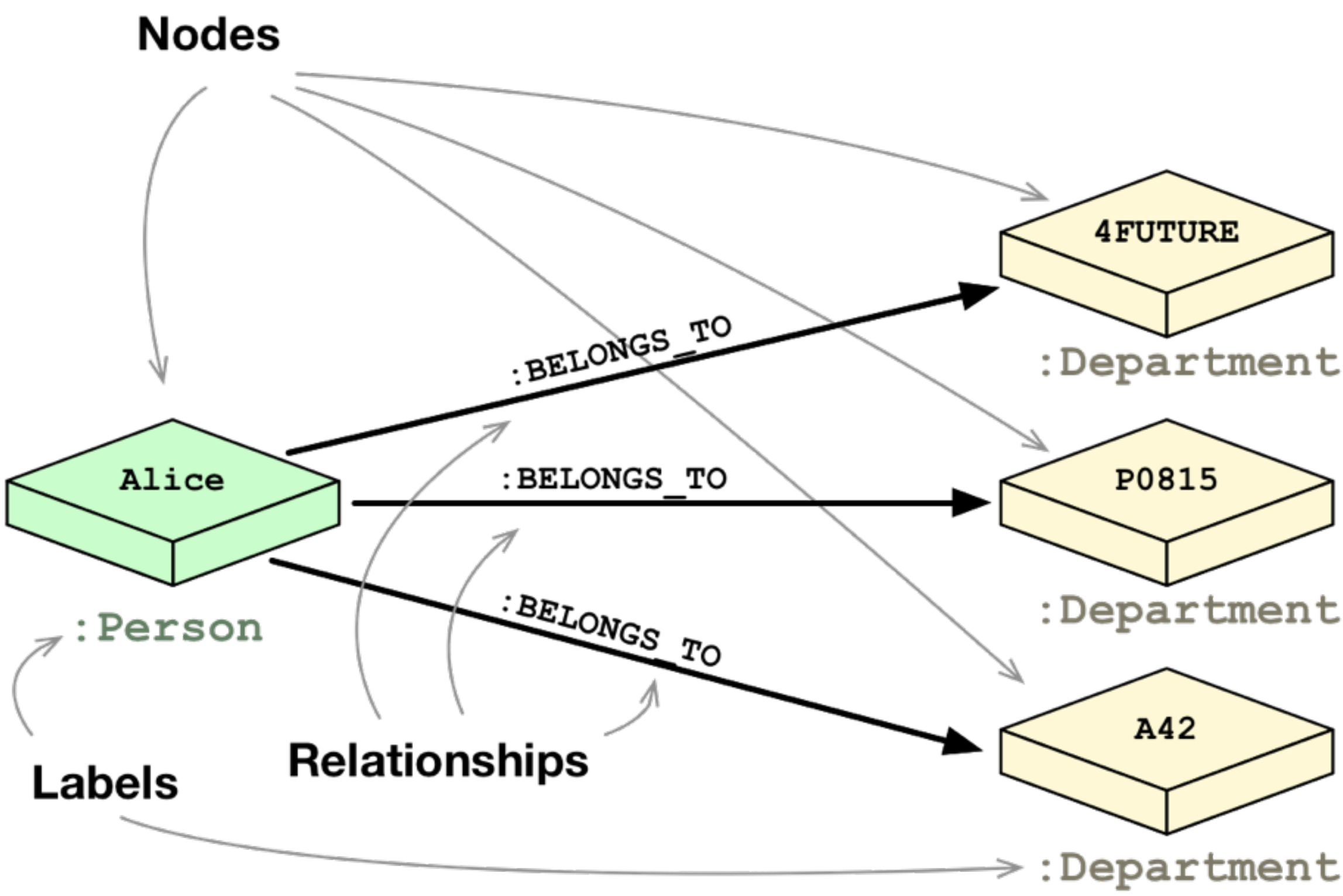
http://neo4j.com/developer/graph-db-vs-rdbms/

Graph Databases



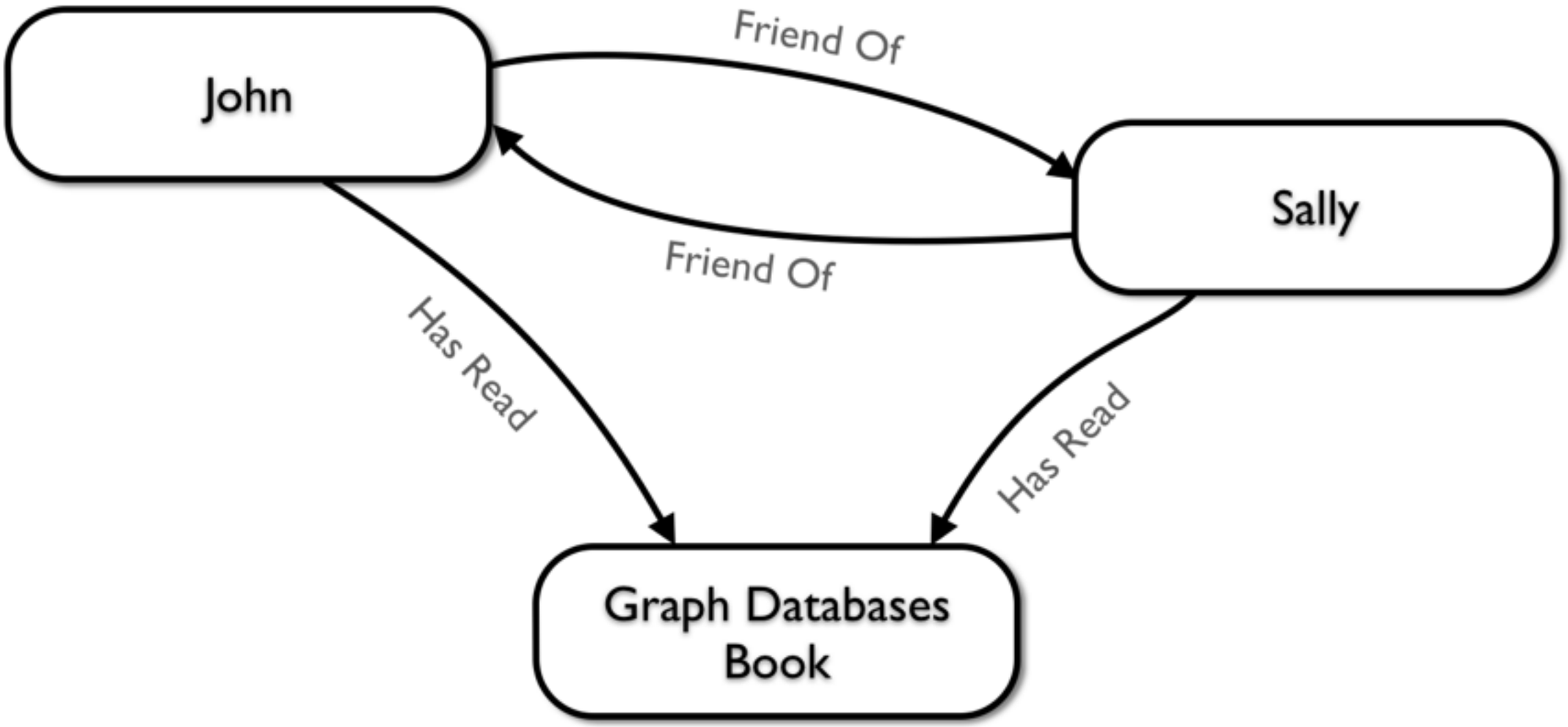
http://neo4j.com/developer/graph-db-vs-rdbms/

Graph Databases



http://neo4j.com/developer/graph-db-vs-rdbms/

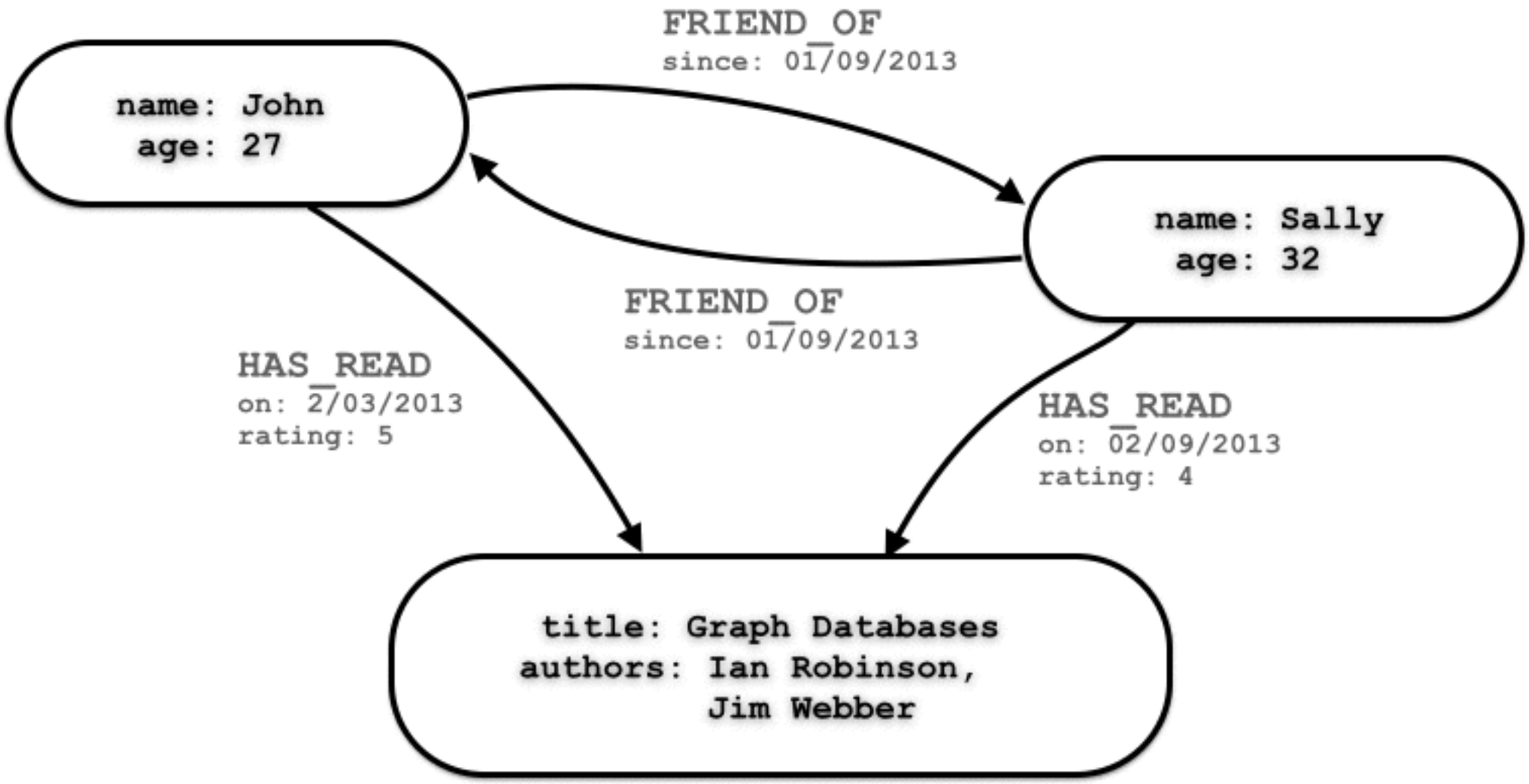
Graph Databases



- Similar to ER Diagram - ***except data instances*** are stored on a graph

http://neo4j.com/developer/guide-data-modeling/

Graph Databases



- Attributes stored on ***nodes*** and ***edges***

http://neo4j.com/developer/guide-data-modeling/

Graph Databases

- Example**: Neo4J** (http://neo4j.com/)
- Native graph storage (implemented in Java)
- Schema is flexible (hashes on Nodes and Edges)
- Consistent and Atomic operations like SQL DBs

  Fully **ACID** - **a**tomicity, **c**onsistency, **i**solation, **d**urability
- Specialized Graph Query Language (i.e. Cypher)

SQL Statement

```
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"
```

Cypher Statement

```
MATCH (p:Person)<-[:EMPLOYEE]-(d:Department)
WHERE d.name = "IT Department"
RETURN p.name
```

Graph Databases

- **Graph databases** stores data on a graph with free interrelation of data than actual values.
- Queries based on *traversing nodes* (following edges).
- Prototypical use cases
  - social networking applications, recommendation engines, access control lists, geographic data.
- Good for object-oriented systems (references btw. objects)

Drawbacks:

- Terrible performance when partitioning a network (i.e. distributing across nodes with network hops).
- Scales poorly.
- Not the best choice for tabular analytical data.

Conclusion

Data Model	Performance	Scalability	Flexibility	Complexity	Functionality
Key–Value Store	high	high	high	none	variable (none)
Column-Oriented Store	high	high	moderate	low	minimal
Document-Oriented Store	high	variable (high)	high	low	variable (low)
Graph Database	variable	variable	high	high	graph theory
Relational Database	variable	variable	low	moderate	relational algebra

Source: http://www.slideshare.net/bscofield/nosql-codemash-2010

- *Performance* considerations with Database is inevitable
  - Indexes and Compression
- Think more about *distributed systems: CAP Theorem*
- *noSQL* Database landscape is still evolving
  - *Columnar* - BigTable, Cassandra, HBase
  - *Document* - MongoDB, CouchDB, DocumentDB,
  - *Key/Value* - Redis, Dynamo, Riak, Memchache
  - *Graph* - Neo4j, AllegroGraph, FlockDB

Resources and Onward Routes

- Good overview of Databases beyond the Relational Model

- Now that you have experienced the Relational Model, try implementing a database in an alternative noSQL model.
(*Summary Project!*)