

NoSQL and MongoDB

CS 4417B

The University of Western Ontario

- MapReduce
 - Shuffle and Sort
 - Examples: PageRank
- MongoDB
 - Aggregation pipeline
- Assignment 3
- Under the hood
 - How data is distributed
 - Consistency
 - Fault tolerance

Introduction

- Motivating “NoSQL” by considering relational databases and modern big data applications.
- We will highlight the limitations of relational databases:
 - Data models
 - Assumptions of the execution environment

Relational Databases (SQL)

- Data stored in tables (columns and rows)
- Requires predefined data models prior to use i.e., static schemas
 - A “schema” tells you what data are stored about each entity in the database, e.g.
 - Person:
 - Name, Address, Phone, E-mail
- Focus on integrity, atomicity
- Distributed implementations an afterthought

Application: Personalization

- Personalization is a means of meeting a customer's needs more effectively and efficiently
- Personalization engines create customized online experiences for customers in real time based on analysis of behavioral and demographic profiles, historical interactions, and preferences.
- Examples:
 - Netflix/spotify, amazon shopping, YouTube recommendations, ...

<https://www.mongodb.com/use-cases/personalization>

Application: Personalization

Data Modelling

- Customer data is more than names and addresses
- Example: For online stores it includes browsing habits to determine ads to display
 - The ads vary based on the product
 - The data model needed for a diverse set of product information typically does not fit well in the rows and columns of a relational database
 - Why? A fixed schema for diverse products is going to be either 1) insufficient and fast or 2) enormous and slow

Application: Personalization – Data Modeling

- Common information about a product includes product name, type
- However, other details may depend on the product:
 - Example:
 - A book has author, title, isbn
 - A song has artist, title, length
- A single schema for all products would be huge
- What happens if you get a new type of product?
 - Do you want to come up with a new schema?
 - Changing the schema of an enormous pre-existing table is a huge computational burden
- **Compositional structure** of products is challenging to capture

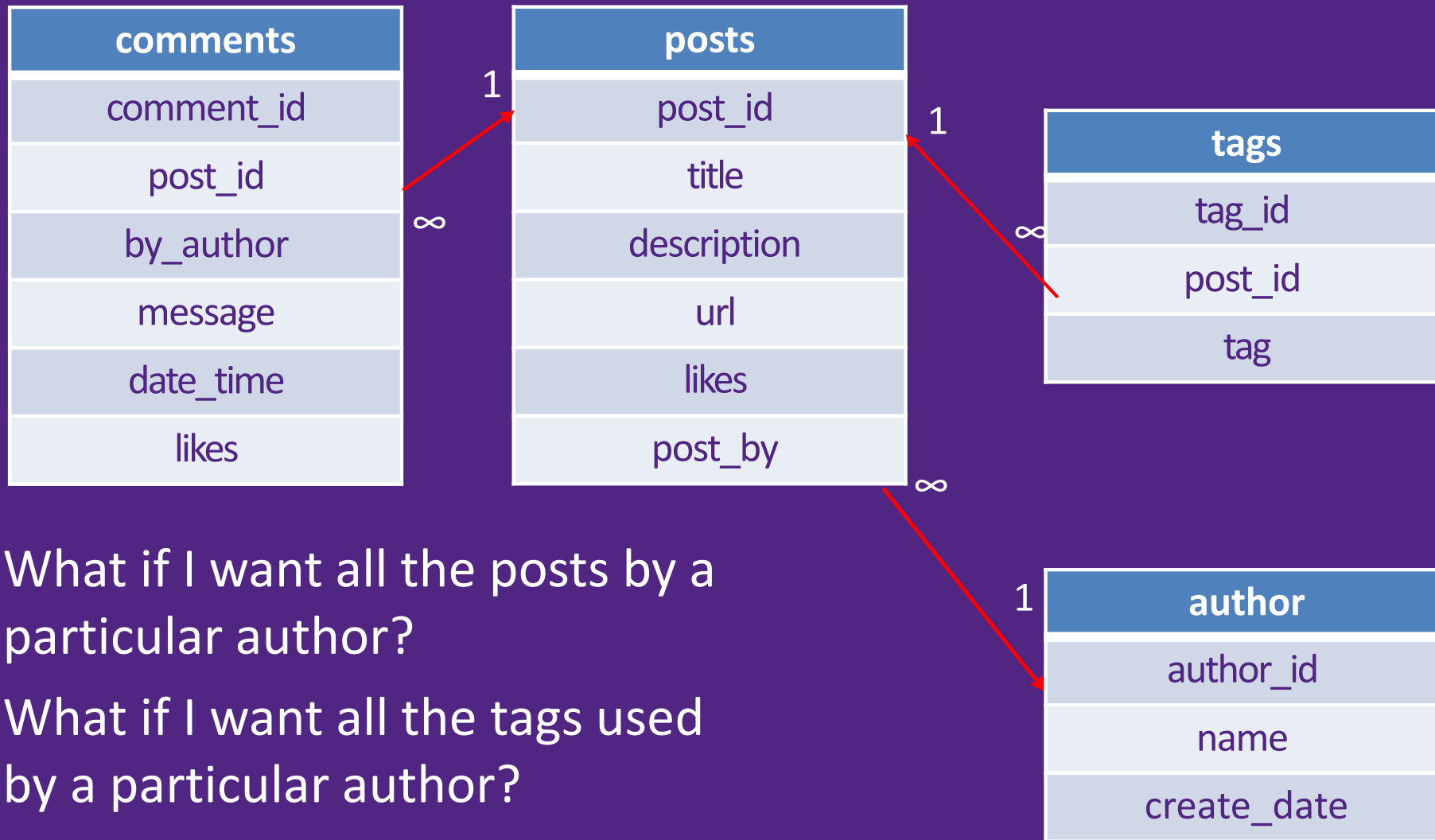
Application: Blogs (e.g. Twitter)

- A blog consists of a collection of text entries or **posts**
- A **tag** is a word or two that reflects the content of the post
- Main data entity is a post

Application: Blogs

- A blog post has an author, some text and many comments
- The comments are unique per post, but one author has many posts
- A comment is associated with a post
- A post may have multiple comments
- An author may have many comments spread out over multiple posts

Relational Data Model for Blogs



- What if I want all the posts by a particular author?
- What if I want all the tags used by a particular author?
- Using the SQL operations of join, select is non-trivial

Relational structure may be hard to manage at scale.

Problems With Relational Databases

- Overhead for complex select, update, delete operations
 - Select: Joining too many tables to create a huge size intermediate table.
 - Update: Each update can affect many other tables
 - Delete: Must guarantee the consistency of data
- Schemas may be difficult to define and may evolve over time.

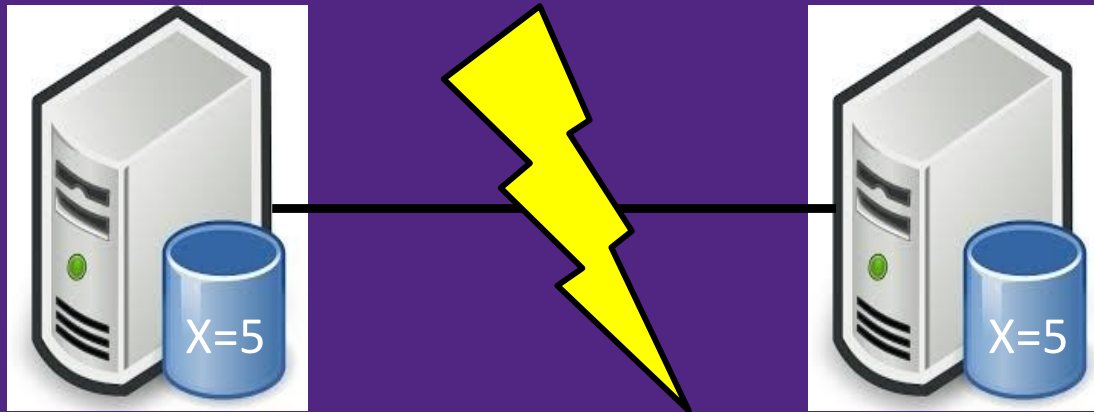
Relational Databases

- Relational databases are expected to have these properties:
 - Atomicity: Each transaction must be “all or nothing” i.e. if one part fails then the transaction fails
 - Example: Let’s say that you are updating two tables that have fields in common. We expect this update to be atomic.
 - Consistency: A write to a table means that any read of the table after the write will see the most recent value i.e.,
 - Every read receives the most recent write or an error

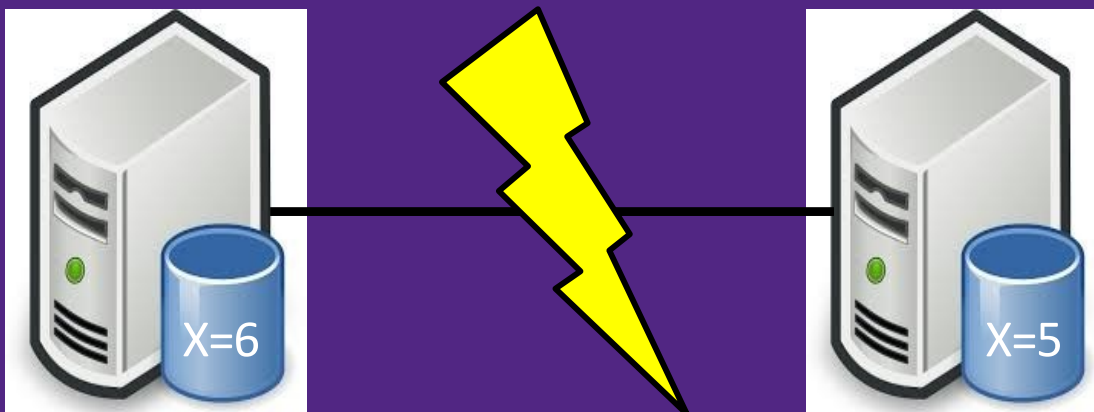
Relational Databases

- These were designed to run on a single server to maintain the integrity of the tables
 - The reason for this is that distributing data and ensuring atomicity and consistency is hard

Relational Databases



- X is to become 6
- What if the network fails after one of the updates is done?



- Possible to handle by e.g. deciding no write is valid unless all replicas have been updated, but this negatively impacts performance

Large-Scale Databases

- When companies such as Google and Amazon were designing large-scale databases, 24/7 availability was a key requirement
 - A few minutes of downtime means lost revenue
- When horizontally scaling databases to 1000s of machines, the likelihood of at least one node or network failure increases tremendously
- Difficult to have strict consistency

Problems With Relational Databases

- Relational databases were defined for certain types of applications
 - Examples: Banking, finance
 - These were the primary applications 40+ years ago
 - “Truth” and consistency are necessary requirements
- Scaling these kinds of databases is hard

How to store, query and
process these data efficiently?

NoSQL

- NoSQL = Non SQL or Not only SQL
- NoSQL is a better solution for many types of data where
 - Strict schema won't work
 - Relational structures are complicated
 - Atomicity and consistency maybe not crucial
 - Distribution/efficiency is crucial
 - Fault-tolerance
- Definition (Wikipedia)
 - "...storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases."

NoSQL

- A form of database management system that is non-relational
- Systems are often schema-less, avoid joins and are easy to scale, by
 - Adding computers
 - Sharding and replication
- The term NoSQL was coined in 1998 by Carlo Strouzi













Why NoSQL?

- The data we store is more complex and dynamic than 20-40 years ago
- Data can't fit on a single server
- Easy Distribution
 - With all this data there is a need to make use of multiple servers
 - Should be able to add servers in response to demand
 - No disruption of service in the presence of failures

Different Types of NoSQL Systems

- Document Store
- Key-value store
- Graph
- BigTable

Examples of NoSQL Systems

Document Database	Graph Databases
  Couchbase  MarkLogic™  mongoDB	 Neo4j  InfiniteGraph The Distributed Graph Database
Wide Column Stores	Key-Value Databases
 redis  amazon DynamoDB  AEROSPIKE  riak	 ACCUMULO™ HYPERTABLE INC  Cassandra APACHE HBASE Amazon SimpleDB

What will we focus on?

- This week's lectures focus on the following:
 - MongoDB – This is an example of NoSQL database that supports document store
 - Example related to challenges with NoSQL
 - Relational vs NoSQL
 - Other models of NoSQL

References

[1] Mikayel Vardanyan, Picking the right NoSQL Database Tool:
<http://blog.monitis.com/index.php/2011/05/22/picking-the-right-nosql-database-tool/>

MongoDB – Part 1

A Document Store

Who Uses MongoDB?

In Good Company



MongoDB

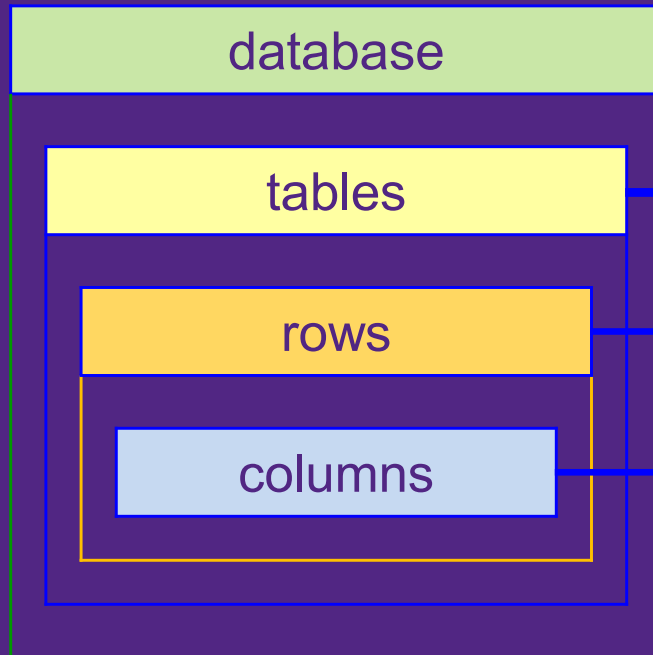
- An open source and document-oriented database.
- Documents are in BSON format, consisting of field-value pairs
 - Binary JSON; like JSON but less space
- Designed for scalability and developer agility
- Dynamic schemas (more like no schema at all.)

JSON and BSON

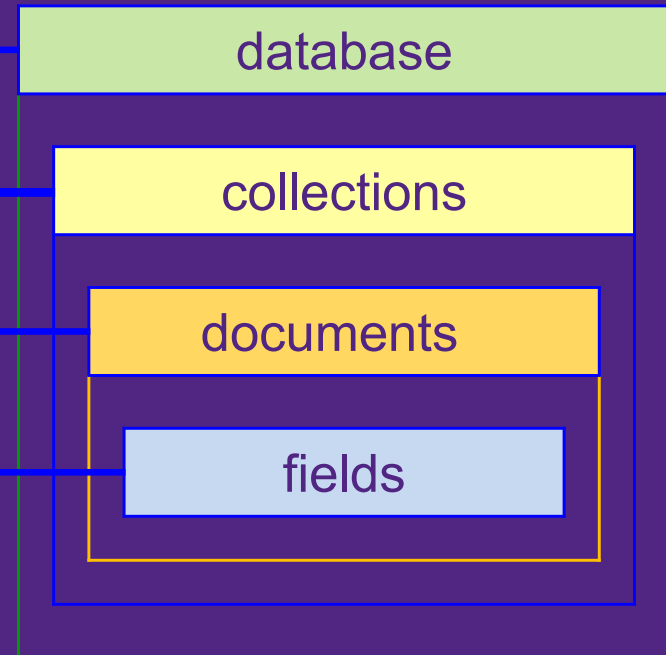
- JSON: JavaScript Object Notation
 - Built on
 - Name and value pairs
 - Objects can be nested
- BSON – Binary JSON
 - Binary encoded serialization of JSON-like documents
- Embedded structure reduces the need for joins

SQL vs MongoDB

SQL Terms/Concepts



MongoDB Terms/Concepts



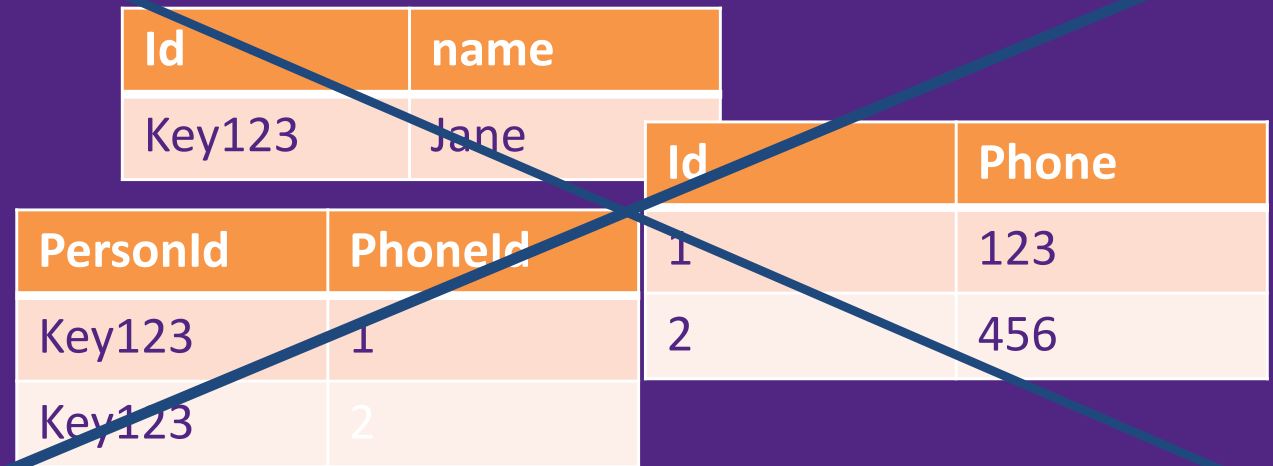
JSON vs Normalization

E.g. keeping track of business cell phones

JSON-format

```
{  
  _id : "Key123",  
  name: "Jane",  
  phones: [123, 456],  
  ...  
}
```

No Normalization



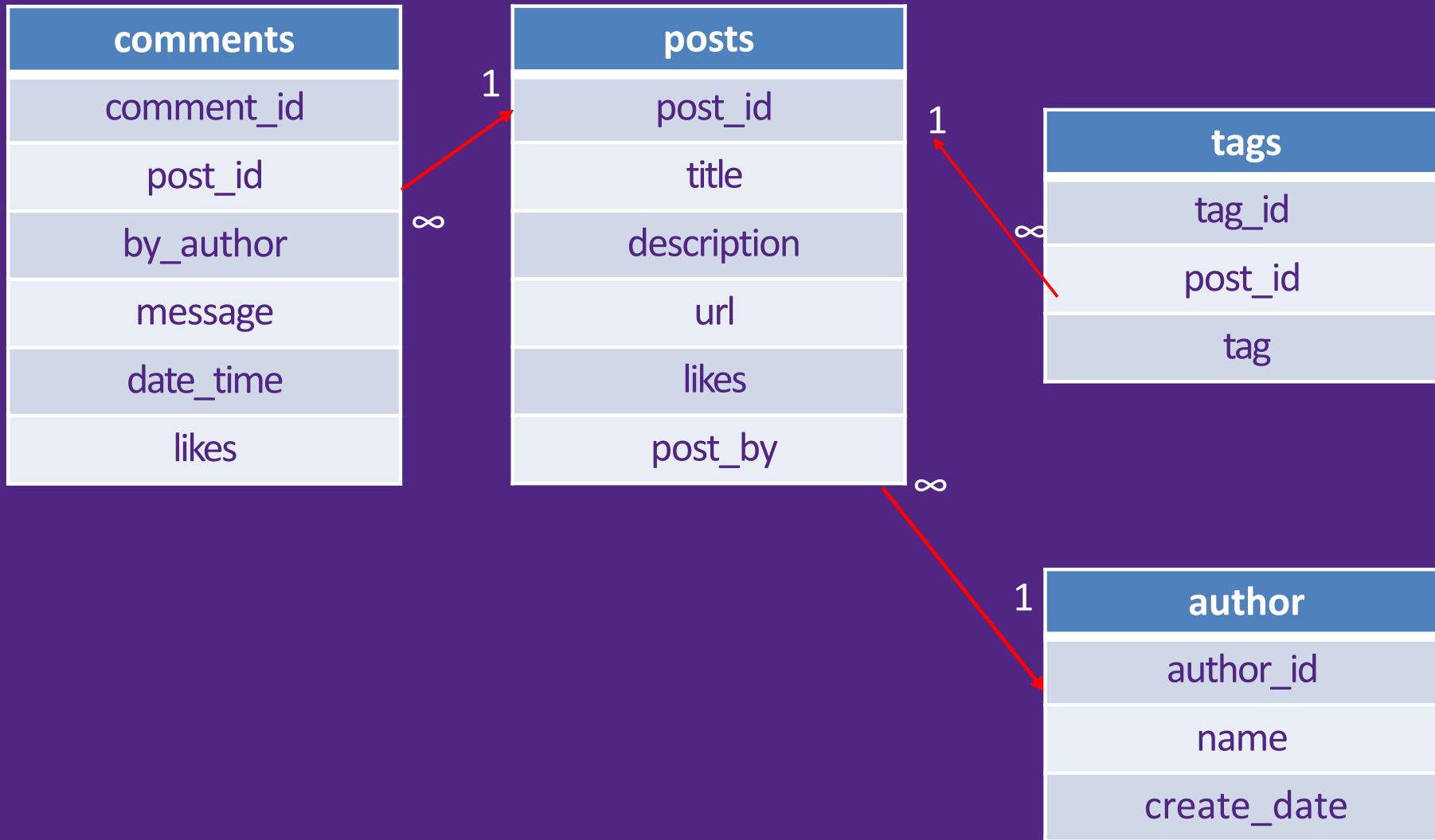
The diagram illustrates a non-normalized database structure. It consists of three tables. The first table has columns 'Id' and 'name'. The second table has columns 'PersonId' and 'PhoneId'. The third table has columns 'Id' and 'Phone'. A large blue 'X' is drawn over the entire set of tables, indicating that this structure is not normalized.

Id	name
Key123	Jane

PersonId	PhoneId
Key123	1
Key123	2

Id	Phone
1	123
2	456

Relational Data Model for Blogs



MongoDB "Data Model" for Blogs

```
{
  _id: POST_ID,
  title: TITLE_OF_POST,
  description: POST_DESCRIPTION,
  by: NAME,
  url: URL_OF_POST,
  tags: [TAG1,TAG2,TAG3],
  likes: TOTAL_LIKES,
  comments: [
    {
      user: COMMENT_BY,
      message: TEXT,
      dateCreated: DATE_TIME,
      likes: TOTAL_LIKES
    },
    {
      user: COMMENT_BY,
      message: TEXT,
      dateCreated: DATE_TIME,
      likes: TOTAL_LIKES
    }
  ]
}
```

...continued top of next column...

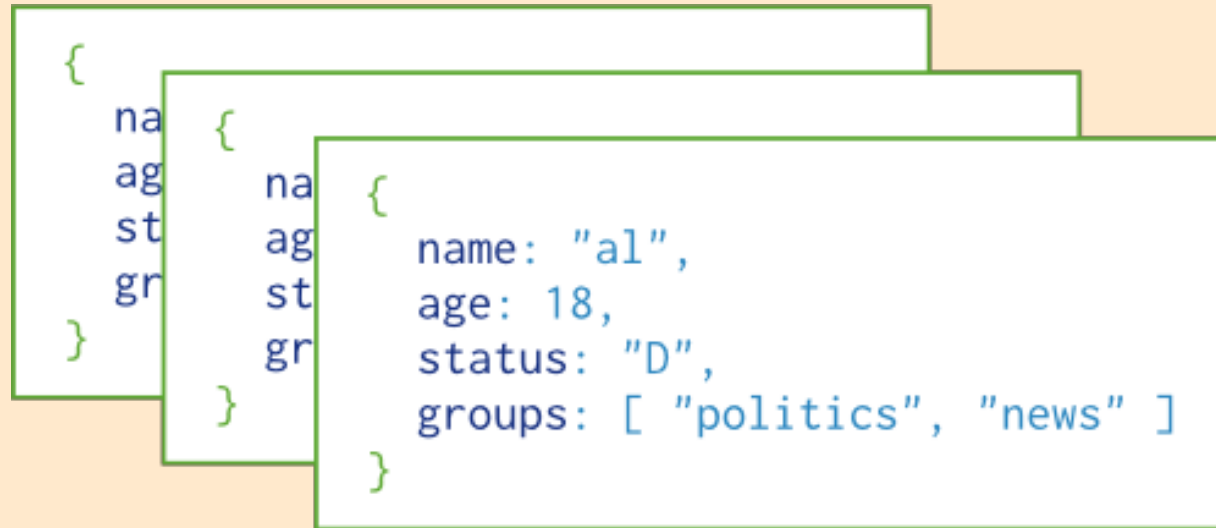
Note that this isn't a "real" data model – you don't need to give this to MongoDB to import documents

Benefits

- When doing a query: Embedded objects/documents retrieved in the same query as parent object/document
 - Only 1 trip to the DB server required
- Objects in the same collection are generally stored contiguously on disk
 - Faster access
- Easier than specifying joins

MongoDB Data Model

A **collection** contains **documents**



Collection

MongoDB Data Model

Structure of a JSON document:

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

← field: value
← field: value
← field: value
← field: value

The value of
field

- Native data types
- Arrays
- Other documents

MongoDB Data Model

Embedded documents:

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

The "primary key"

Embedded sub-document

Embedded sub-document

- Example of specification of access to a field in an embedded document
 - Contact: phone, email
- Embedded documents means we do not need complicated join tables

MongoDB Data Model

Referencing or linking documents

user document

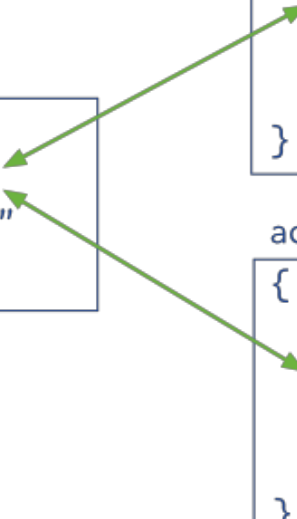
```
{  
  _id: <ObjectId1>,  
  username: "123xyz"  
}
```

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"  
}
```



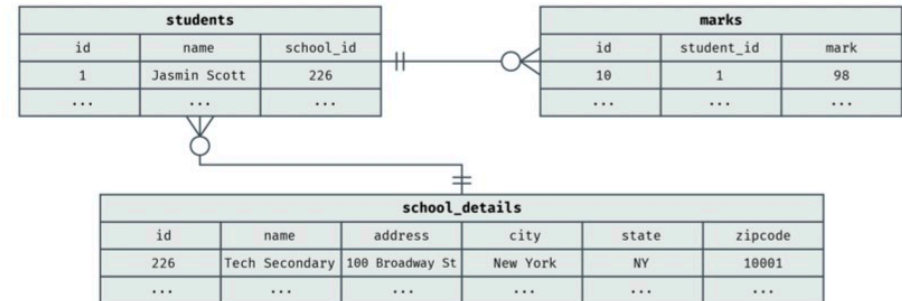
MongoDB

```
{
  "_id": 1,
  "student_name": "Jasmin Scott",
  "school": {
    "school_id": 226,
    "name": "Tech Secondary",
    "address": "100 Broadway St",
    "city": "New York",
    "state": "NY",
    "zipcode": "10001"
  },
  "marks": [98, 93, 95, 88, 100],
}
```

mongo

```
> db.students.find({"student_name":
  "Jasmin Scott"})
```

SQL



Results

name	mark	school_name	city
Jasmin Scott	98	Tech Secondary	New York
...

sql

```
SELECT s.name, m.mark, d.name as "school name",
d.city
FROM students s
INNER JOIN marks m ON s.id = m.student_id
INNER JOIN school_details d ON s.school_id = d.id
WHERE s.name = "Jasmin Scott";
```

References

- 1 Mikayel Vardanyan, Picking the right NoSQL Database Tool:
<http://blog.monitis.com/index.php/2011/05/22/picking-the-right-nosql-database-tool/>
- 2 BSON Specification: <http://bsonspec.org/>
- 3 MongoDB CRUD operations: <http://docs.mongodb.org/manual/crud/>
- 4 MongoDB Write operations: <http://docs.mongodb.org/manual/core/write-operations/>
- 5 MongoDB Investors: <http://www.mongodb.com/investors>
- 6 MongoDB Closes \$150 Million in Funding: <http://www.mongodb.com/press/mongodb-closes-150-million-funding>
- 7 MongoDB Aggregation introduction:
<http://docs.mongodb.org/manual/core/aggregation-introduction/>

