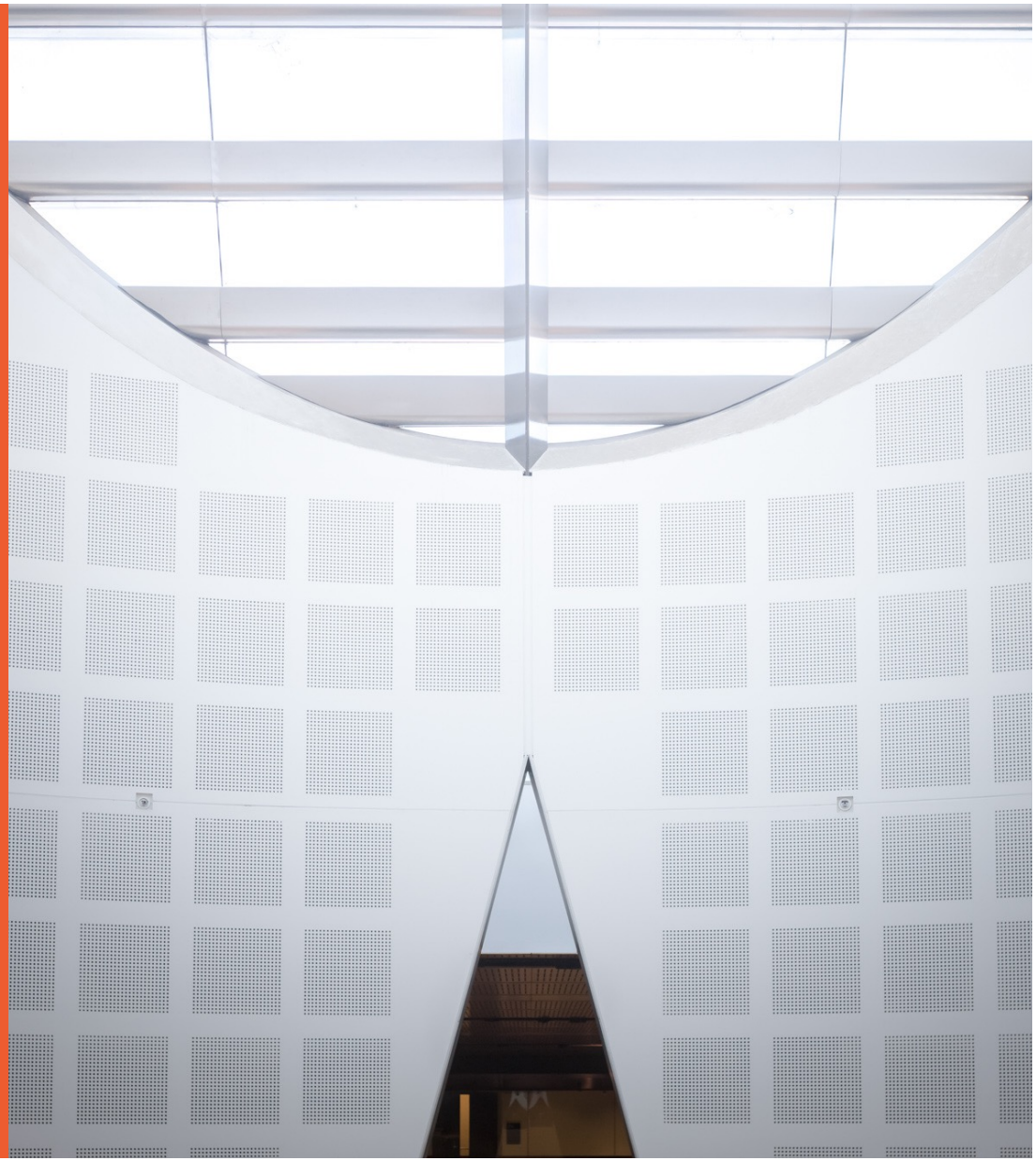
COMP5339: Data Engineering

Week 5: Semistructured Data and NoSQL Databases

Presented by
Lijun Chang
School of Computer Science

THE UNIVERSITY OF SYDNEY



Semistructured Data

Semistructured Data

- HTML, XML and JSON are examples of **semistructured data models**
  - data with non-rigid structure

- Characteristics of **semistructured data**
  - Missing or additional attributes
  - Nesting: semistructured objects ('documents') are hierarchical / have tree-structure
  - Different types in different objects
  - Heterogeneous collections

Self-describing, irregular data, no a priori structure

HTML vs. XML

- While HTML is mainly for web page design, XML is the more structured "cousin" for data exchange

- Some web services can be asked to send XML rather than HTML pages
- Also common in enterprise data exchange, or open data sets

- Similar idea to HTML:
  - Semi-structured format
  - tag markup
  - Main difference: XML has **user-defined** tags, while HTML has **pre-defined** tags according to the WWW standard

Example: XML Document

```
<?xml version='1.0'?>
<!-- This file represents a fragment of a -->
<!-- book store inventory document        -->
<bookstore>
   <book genre="autobiography">
      <title>The Autobiography of
             Benjamin Franklin</title>
      <author>
         <first-name>Benjamin</first-name>
         <last-name>Franklin</last-name>
      </author>
      <price>8.99</price>
   </book>
   <book genre="novel">
      ...
   </book>
</bookstore>
```

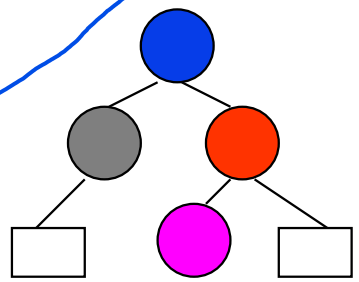
XML describes the content

Logical Document Structure

- XML refers to its objects as ***elements***
- The top-most element is called the ***root*** or ***document element***.
- Elements are bound by ***tags***:
  - Every *opening tag* must have matching *closing tag*
  - Tag names are "case-sensitive"
  - Tags cannot overlap; e.g. the following is not allowed:
    <A>…<B>…</A>…</B>
  - Tags of empty elements have a special syntax.
- Tree structure! (not a graph)

```
<book>
   <title>
    Autobiography of
      Benjamin Franklin
   </title>
   <price>
      <AUD/>
       8.99
   </price>
</book>
```

Empty Element

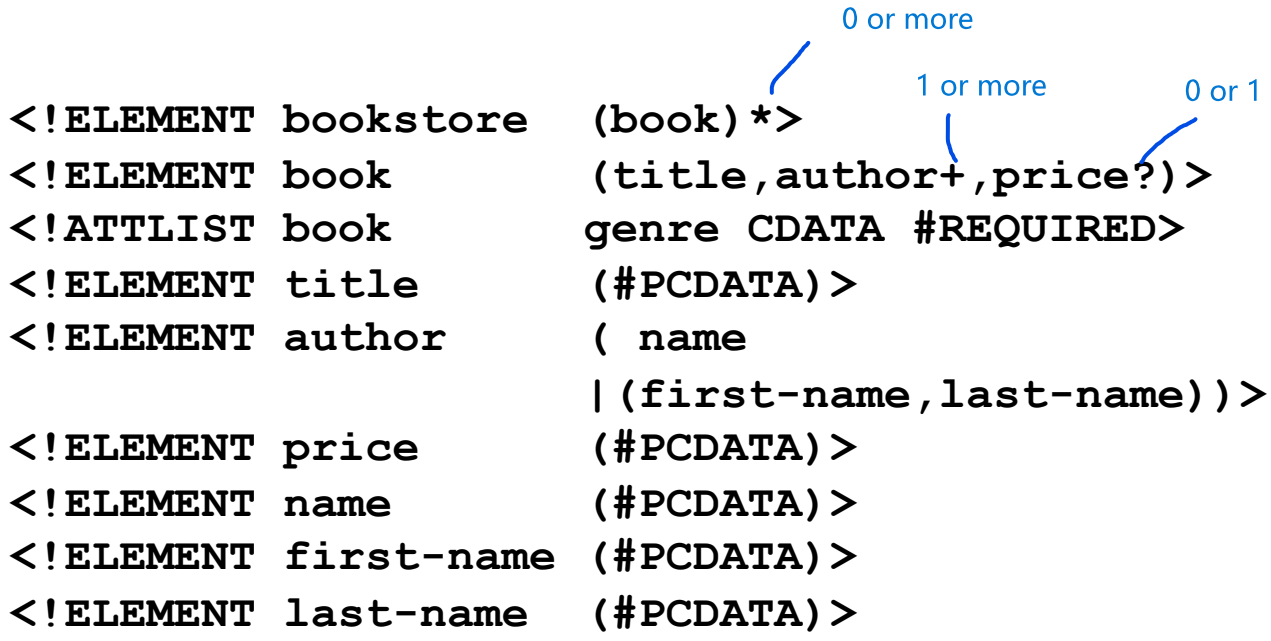


Document Type Definition (DTD)

- XML defines general syntactical properties with *user-defined* tags.
- Question: Which tags are actually *allowed* in an XML document?
- The "grammar" of an XML document is specified using a so-called document type definition (DTD).

- In the following:
  - What does a DTD look like?
  - What can be specified in a DTD?
  - How to associate an XML document with a DTD?

Example: Bookstore DTD



Sort of like a schema but not really.

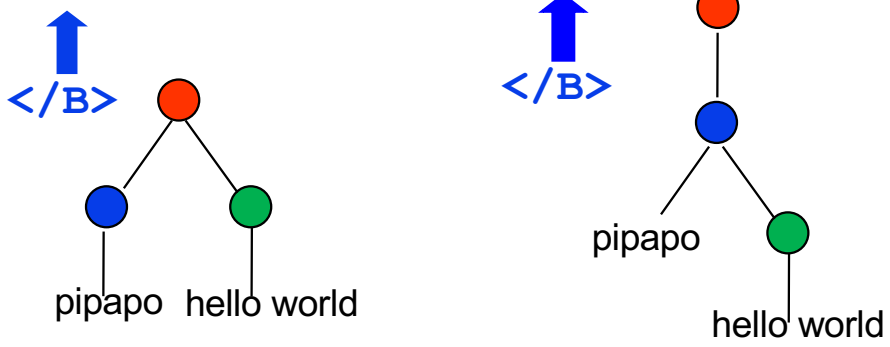
- BNF grammar establishing constraints on element structure and content + definitions of entities

XML-Conformance: Well-Formed vs. Valid

- **Well-formed**
  document satisfies XML syntax constraints (matching tags),
- **Valid**
  document is well-formed and satisfies a pre-defined schema, as formalised in a document type definition (DTD).
- Example: an incomplete marked document fragment

```
<A><B>pipapo<C>hello world</C></A>
```

possible structures:



Example: Document Instance with DTD

```
<?xml version='1.0'?>
<!-- This file represents a fragment of a bookstore inventory doc -->
<!DOCTYPE bookstore SYSTEM "http://www.books.com/bookstore.dtd">
<bookstore>
   <book genre="autobiography">
      <title>The Autobiography
             of Benjamin Franklin</title>
      <author>
         <first-name>Benjamin</first-name>
         <last-name>Franklin</last-name>
      </author>
      <price currency="USD">8.99</price>
   </book>
   <book genre="novel">
      <title>The Confidence Man</title>
      <author>
         <first-name>Herman</first-name>
         <last-name>Melville</last-name>
      </author>
      <price currency="Dollar">11.99</price>
   </book>
   ...
</bookstore>
```

```
<!ELEMENT bookstore   (book)*>
<!ELEMENT book        (title,author+,price?)>
<!ATTLIST book        genre CDATA #REQUIRED>
<!ELEMENT title       (#PCDATA)>
<!ELEMENT author      (name
                    |(first-name,last-name))>
<!ELEMENT price       (#PCDATA)>
<!ATTLIST price       currency (CHF,DEM,USD)>
<!ELEMENT name        (#PCDATA)>
<!ELEMENT first-name  (#PCDATA)>
<!ELEMENT last-name   (#PCDATA)>
```

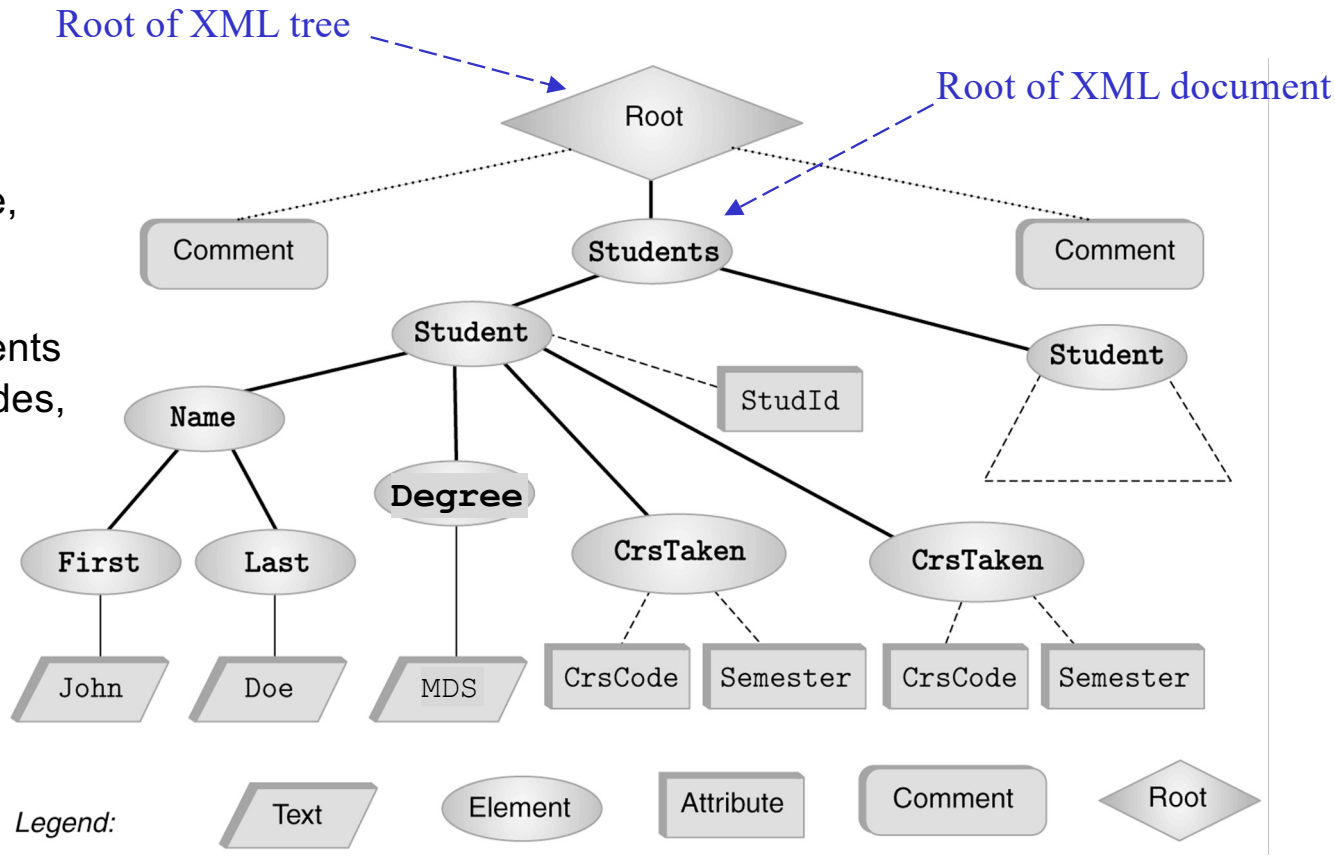
How to Query or Filter XML?

- DOM Navigation
  - XML documents represent a tree structure which can be navigated using XML's **Document Object Model (DOM)**
- XPath
  - XPath expressions allow to query single values, node(s) or whole subtrees within one XML document
- XQuery
  - XQuery builds on Xpath
  - It is a declarative query language over a set of XML documents
  - Beyond the scope of this lecture
- CSS Selectors – for HTML documents

XPath Document Model Tree

XPath views an XML document as a tree

- Root of the tree is a *new* node, which doesn't correspond to anything in the document
- Internal nodes are XML elements
- Leaves are attributes, text nodes, comments etc



[Source: Kifer/Bernstein/Lewis, Fig. 15.14]

Semi-Structured Data vs. Structured Data

- Relational World
  - Schema-first, rich type system for attributes, integrity constraints
  - "First Normal Form": only atomic type attributes allowed
    ⇒ many different tables, joins needed

- Semi-structured World
  - Self-describing data with flexible structure
  - Nested data model with tree-structure
  - optional attributes, grammar, schema and vocabulary

JSON vs XML

- JSON has a low-overhead (read: less verbose) than XML
  - XML:

```
<person name="John Smith">
  <address street="1 Cleveland Street"
                 city="Sydney"
               state="NSW"
           zipcode="2006" />
  <phoneNumbers>
    <phoneNumber type="work" number="9351 0000"/>
    <phoneNumber type="fax"    number="9351 3838"/>
  </phoneNumbers>
</person>
```

JSON:

```
{
  "name": "John Smith",
  "address": {
      "street":   "1 Cleveland Street",
      "city":       "Sydney",
      "state":     "NSW",
      "zipcode": 2006
   },
  "phoneNumbers": [
    { "type": "work", "number": "9351 0000" },
    { "type": "fax",    "number": "9351 3838" }
   ]
}
```

Storing Semistructured Data

THE UNIVERSITY OF SYDNEY

Storing Extracted Data?

- Crawling web pages takes time, hence it is a good idea to store the data locally once extracted
  - Avoids to re-crawl the remote server every time for a new analysis
- Two main options
  - File systems (CSV, XML or JSON files)
  - Database

Example: Storing tabular data from a HTML table in CSV files using Pandas

```
import pandas as pd
import requests
from bs4 import BeautifulSoup

page = requests.get("https://www.health.nsw.gov.au/infectious/diseases/Pages/covid-19-latest.aspx")
content = BeautifulSoup(page.text, 'html5lib')

data  = content.find_all("table")
df    = pd.read_html(str(data))[0]
df.tail()

df.to_csv('covid_stats_nsw.csv')
```

Example: Storing JSON response from Web API as file

```
import requests, json, pandas as pd

base_url = 'https://nominatim.openstreetmap.org/search'
my_params= {'q': '1 Cleveland Street,Darlington,Australia','format':'json'}
whoami   = {'User-Agent': 'COMP5339'}

response = requests.get(base_url, params=my_params, headers=whoami)
results = response.json()

# Option 1: write full results to a json file
with open("locations_all.json", mode="w", encoding="utf-8") as write_file:
    json.dump(results, write_file)

# Option 2: convert 1st result as DataFrame and then write that as nortmalised json
df = pd.json_normalize(results[0])
df.to_json("location.json")
```

JSON and XML Support by Databases

- How does semistructured data work with databases?
  - Some Relational DBMS (such as PostgreSQL) support JSON and XML
  - eg. **SQL/XML** standard widely available:
    - Provides **XML** data type to store XML natively in SQL databases
    - Integrates XML support functions
      - for querying (xPath/xQuery based) and for XML data creation
  - PostgreSQL and JSON
    - two native JSON data types: **JSON** and **JSONB**
    - can create JSON data from any query result using **::json** constructor
    - http://www.postgresql.org/docs/current/static/datatype-json.html
  - NoSQL databases
    - eg. MongoDB for JSON data

Does not keep duplicate, will only keep the last value

Storing JSON in RDBMS: Data Types JSON & JSONB

- Support for storage, querying and JSON export
  - JSON type stores exact copy of JSON data in attribute
  - JSONB type stores a binary, decomposed version
    - More overhead for inserts, but claims significant faster querying

```
CREATE  TABLE  StudentJSON (
        sid       INTEGER,
        details   JSONB );

INSERT  INTO  StudentJSON(sid, details)
VALUES(12345, '{"name":    {"first": "Bob", "last": "Smith" },
              "degree":  "BSc(CS)",
              "crsTake": {"crsCode":"INFO1003","semester":"2017sem1"},
              "crsTake": {"crsCode":"INFO1103","semester":"2016sem2"}
             }'  );

SELECT sid, details->'name'->'first' AS name
  FROM StudentJSON
 WHERE details->'degree' ? 'BSc(CS)';
```


[cf. http://www.postgresql.org/docs/current/static/datatype-json.html]

Summary

- Storing semistructured data
  - Many RDBMS provide support for XML and JSON
  - **SQL/XML** standard is widely available
  - JSON/JSONB are native datatypes in, e.g., PostgreSQL
  - SQL querying can be interesting though when SQL+XPath need to be combined…
- Alternatively, NoSQL stores

NoSQL Databases

THE UNIVERSITY OF SYDNEY

NoSQL

- Traditional DBMS platforms were relational (SQL as query language; relational data model) and also powerful (lots of features for integrity, security, tuning), expensive, resource-intensive, hard to administer
  - Mostly focused on scale-up (run on powerful expensive servers to get excellent performance)
- The rise of cloud computing shifted focus to scale-out on many commodity simple servers, with fault-tolerance
  - New systems were designed, and described as "NoSQL" because they gave up features of traditional platforms
  - Simpler data model, simpler queries and updates (eg without cross-table joins or triggers), weaker guarantees for consistency and integrity
  - Often open-source

NoSQL

- Over time, the new platforms added features like joins, triggers and integrity (under pressure from users) while old platforms added support for more diverse data models

- The phrase “Not only SQL” has been used for these systems

The Different Flavors of NoSQL

- In the blue corner: Relational DBMS
  - Set of fixed-structured tuples; joins; SQL; transactions
- In the red corner: 'NoSQL'
  - 'document-oriented' databases
    - Nested, hierarchical lists of key-value pairs
    - e.g. MongoDB or XML databases
  - NoSQL column stores
    - e.g. Amazon's SimpleDB, Google's BigTable, or Apache HBase
  - Pure Key-Value stores
    - e.g. Apache Cassandra, Amazon DynamoDB, Redis, RocksDB
  - Graph databases
    - E.g. Neo4J
- Note: no standard data model or API for 'NoSQL'

Schema-first or schema-late

- One vital property of many of the new systems is that data can be inserted without a schema already being defined for it
  - In contrast: **Relational databases need schema-first**, before data can be inserted

这些是 Schema-Late的例子

- Eg. JSON or XML documents carry description of the attributes in the data, and allow variation between data items

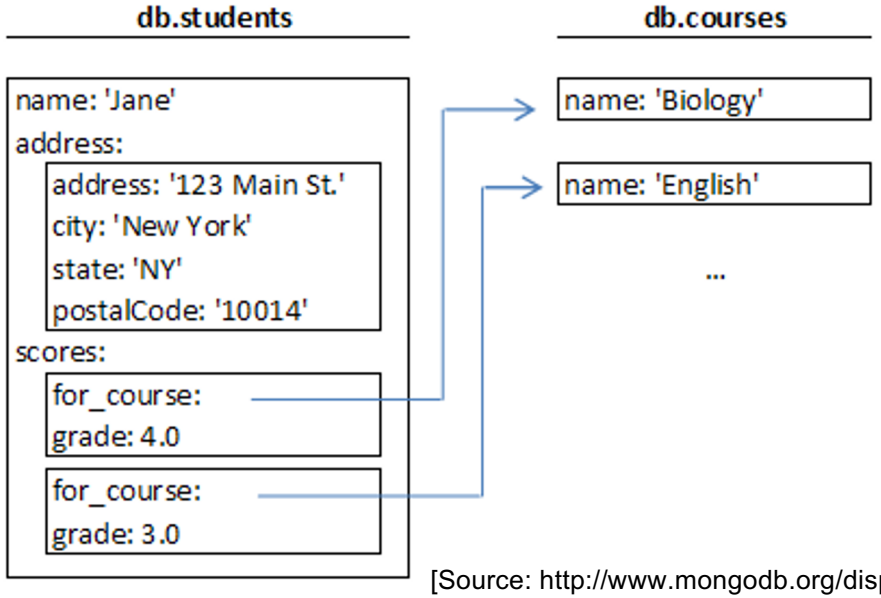
- This is very attractive for receiving data from data sources that have not been fully cleaned

- **schema-late** sometimes also referred to as **schema-on-read**

NoSQL Example: MongoDB

http://www.mongodb.org/

- Data Model
  - 'Document-oriented' storage with *flexible schema*
  - Collections of "documents", which are **(nested) key-value pairs** based on JSON format



Note:
Possible to use MongoDB also as pure (Key, Value) store

[Source: http://www.mongodb.org/display/DOCS/Schema+Design]

...

MongoDB Data Model

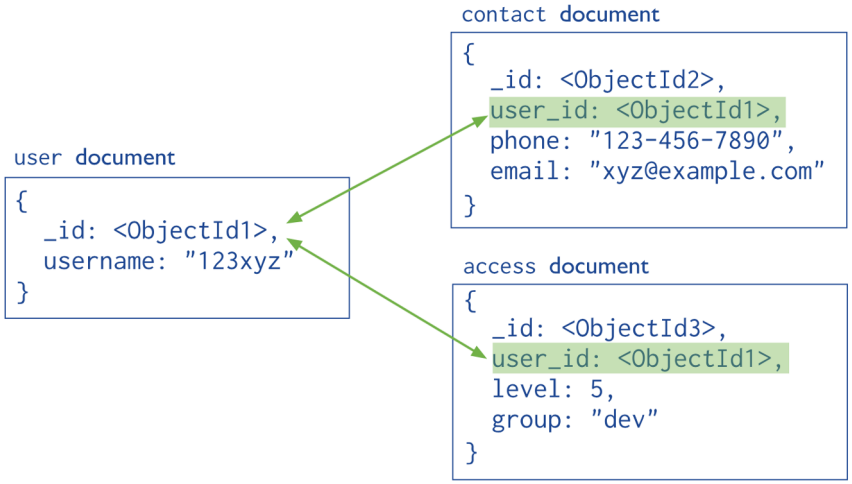
[cf. https://docs.mongodb.com/manual/core/data-modeling-introduction/]

- Basically a JSON store (JSON type system)
- *Flexible schema*: Document in a collection do not need to have the same structure
- All documents have an object ID (**_id**) – either user-defined or automatically generated
- Relationships: either via nested documents ("embedded sub-documents") or using references

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

Embedded sub-document

Embedded sub-document



[Source: https://docs.mongodb.com/manual/core/data-modeling-introduction/]

MongoDB vs. RDBMS Terminology

Relational Database	MongoDB
Database	Database
Table	Collection
Row (tuple, record)	Document
Column (attribute)	Field
Schema First; tuples in a relation follow the same schema	Flexible schema; documents in a collection do not need to have the same set of fields
Normalised Schema; 1NF	Denormalised data model: documents can be nested and can contain arrays

Working with MongoDB

[cf. https://docs.mongodb.com/manual/crud]

- Object-oriented API for variety of languages (C++, C#, Java, Python, PHP,…)
- Create Operations
  - insertOne() call on a collection allows to directly store JSON data
  - Example:

```
db.users.insertOne (
  {
    name: "sue",
    age: "26",
    status: "pending"
  }
)
```

collection

field: value
field: value
field: value

document

  - Read Operations
    - find() operation on collection works like SFW query, but no sql
    - Example:

```
db.users.find (
  { age: { $gt: 26 } },
  { name: 1, address: 1 }
).sort({name})
```

filter condition

projection

result modifier, such as sorting

MongoDB: Querying

- Query Model:
  - Predicates on the key-value pairs
  - Examples:
    - db.students.find(); // SELECT * FROM Students
    - db.students.find({name: 'Jane'}); // SELECT … WHERE name=…
    - db.students.find({name: 'Jane', address.city : 'Sydney' });
- Note: Implicit equality predicate and implicit conjunction (and)
  - Other predicates need a bit of work
  - Example:
    - db.students.find({ name: 'Jane', age: { **$gt : 20** } });
    - db.courses.find({ **$or:** [{name: 'Jane'} , {name: 'Joe'}] });
- Sorting
  - db.courses.find().sort({name: -1}); // ORDER BY name DESC

1是asc

[cf. http://www.mongodb.org/display/DOCS/Advanced+Queries]

MongoDB: Quotation Mark for Keys?

- JSON specification explicitly requires double quotes (") around keys and string values.
  - Single quotes (') or unquoted keys are not permitted and will cause parsing errors.

标准{ "name": "Alice", "age": 25 }
错误{ name: 'Alice', age: 25 }

- In MongoDB
  - When interacting through the MongoDB shell (JavaScript interface), the JSON-like objects **do not require quoted keys** because it is **JavaScript object** syntax, not strict JSON.
  - When working directly with JSON files (such as importing data using mongoimport or through MongoDB Compass), the keys must be quoted, following strict JSON rules.

在MongoDB shell里面，不要求双引号
在MongoDB compass里面，要求双引

MongoDB: Example

```python
from pymongo import MongoClient
import os

# Connect to MongoDB
client = MongoClient(os.environ.get('MONGO_HOST'))

# Select database
db = client['bookstore']

# Insert a document
book = {
    "title": "JSON for Beginners",
    "author": "Alice Brown",
    "year": 2022,
    "price": 19.99
}
result = db.books.insert_one(book)
print(f"Document inserted with _id: {result.inserted_id}")

# Query the document
book = db.books.find_one({"title": "JSON for Beginners"})
print(book)

# Close the connection
client.close()
```

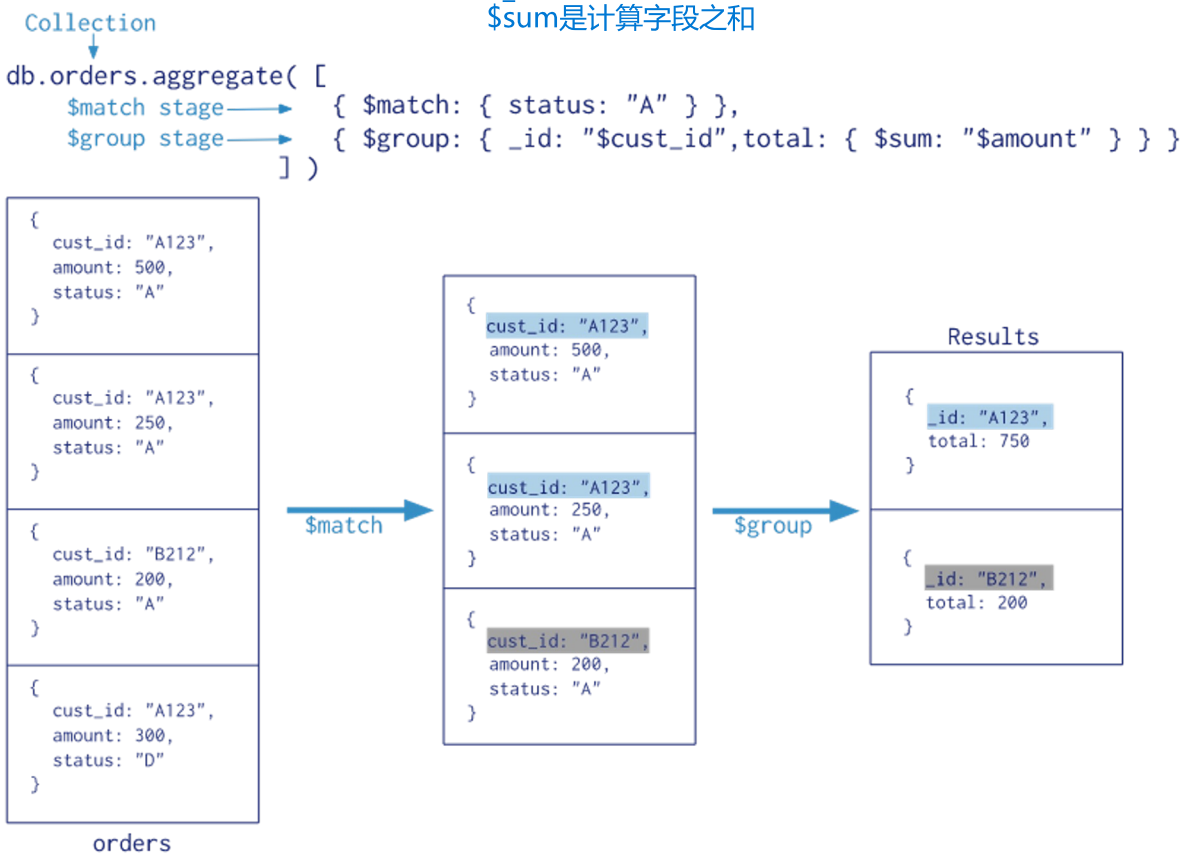
MongoDB Aggregations

这里是组合操作，意思是先match status==A的，

再进行group操作
$_id 指定分组字段
$sum是计算字段之和

Advanced data processing pipeline for transformations and analytics

- Multiple stages
- Similar to a unix pipe
  Build complex pipeline by chaining commands together
- Rich Expressions



[Source: MongoDB, “Architecture and Technology Overview“ PPT]

MongoDB: Scalability

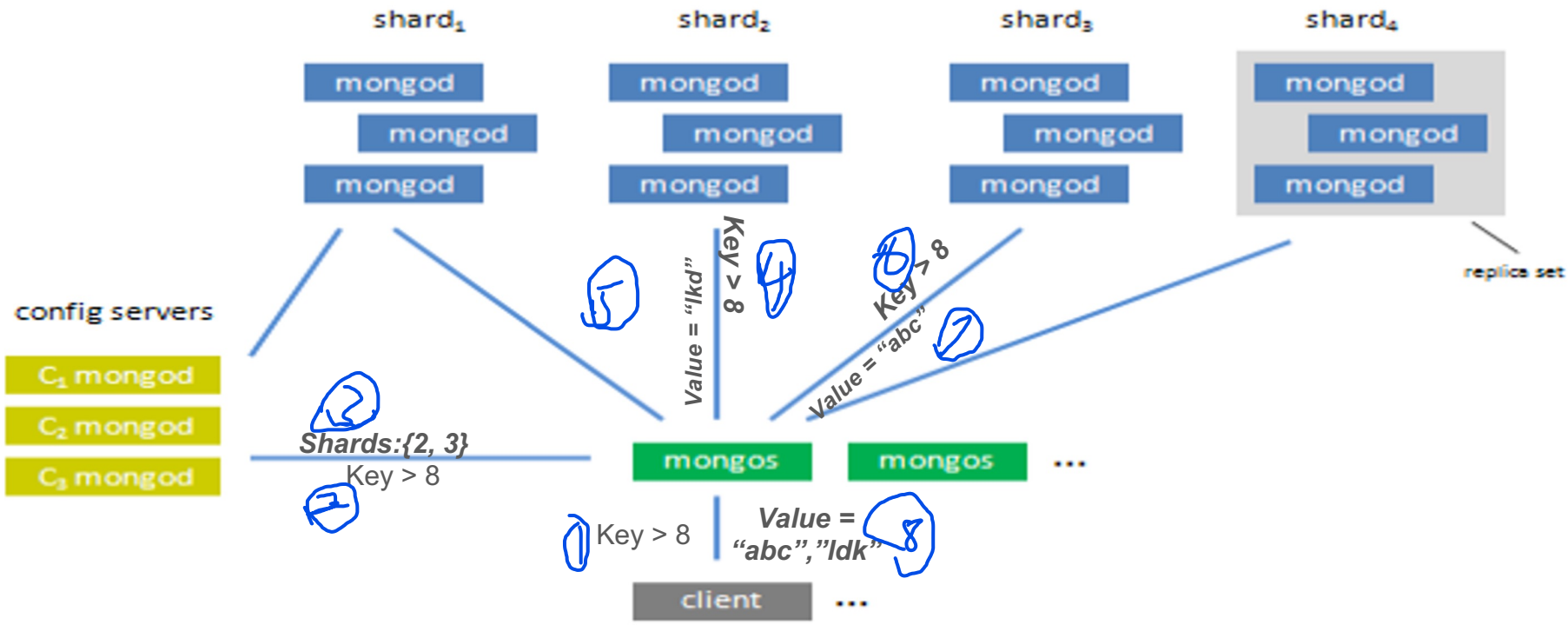
- Automatic Data Partitioning 把一个大集合（collection）的数据 分布到多台机器上，减轻单台机器压力。
  - 'auto sharding' of large collections:
    "partitioning of data among multiple machines in an order-preserving manner"
  - Means: horizontal range partitioning per 'table'

每个 shard 内部有多个副本（replica set），但 只有一个 primary（主节点）接收

- Single-Leader Replication
  - Asynchronous single-leader replication within one 'shard'
  - Single leader to receive updates
  - Main goal: Availability
  - Optionally: Queries to followers, but then only 'eventual consistency'

- Load Balancing 将 请求（读写或查询）均匀分配到多个节点，避免某个节点过载。

MongoDB: Distributed Architecture



[Source: http://www.mongodb.org/display/DOCS/Sharding+Introduction]

Summary

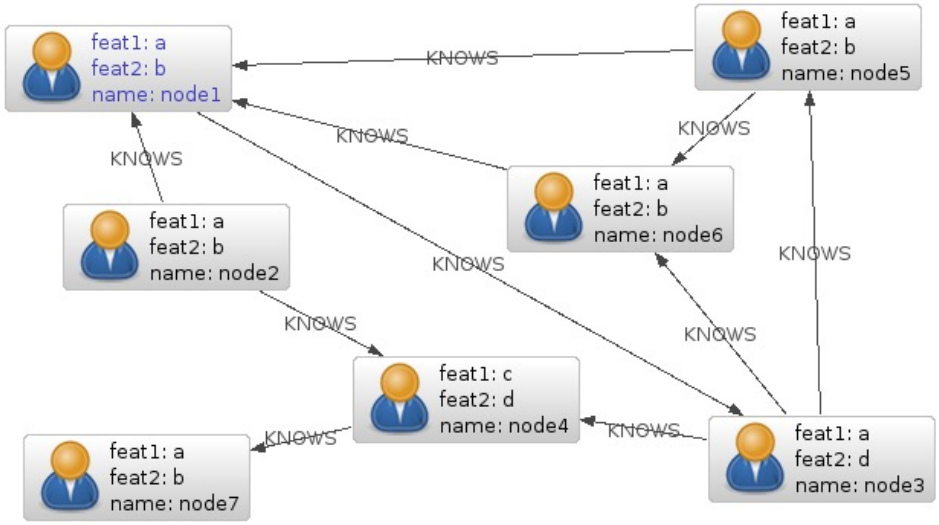
- NoSQL databases allow us to go beyond the rigid structure of the relational data model
- NoSQL databases for semi-structured data
  - Flexibility: Schema-first vs. schema-late
  - Fast: less functionality means less overhead
  - (proprietary) querying of semi-structured data
  - Scalability
- Various systems on offer
  - Examples are MongoDB, Couchbase, Azure Cosmos DB etc.

Graph Databases

Graph Databases

Graph databases are designed to store, query, and manage graph-structured data.

Data Model: Consists of nodes (entities), edges (relationships), and properties (attributes of nodes and edges).

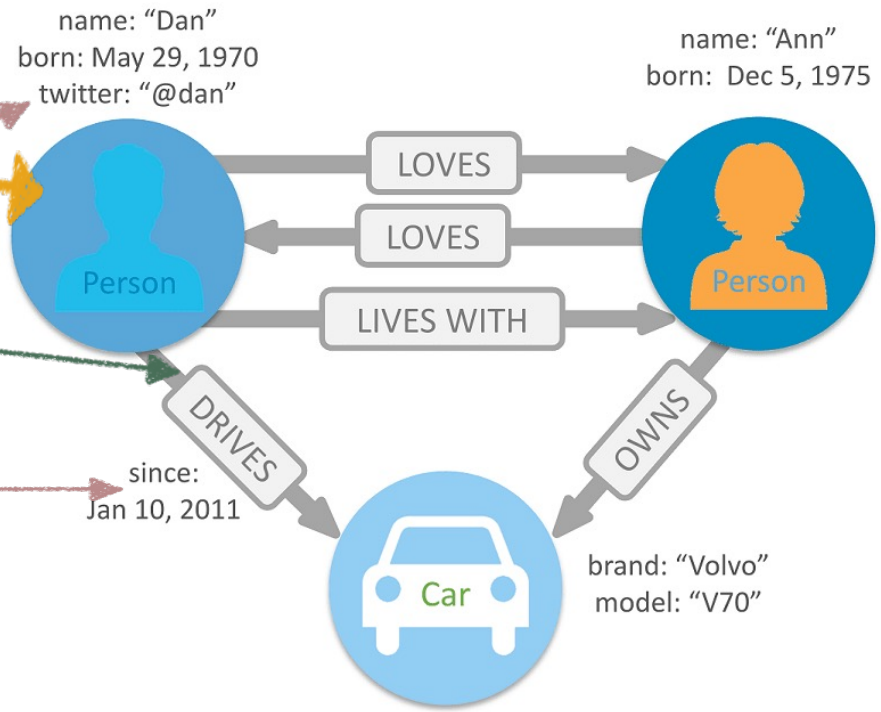


Key Concepts in Graph Data Model

Nodes: Represent entities such as people, products, or locations.

Edges: Represent relationships between nodes, such as friendships, transactions, or routes.

Properties: Attributes or metadata associated with nodes and edges



https://res.cloudinary.com/practicaldev/image/fetch/s--otONh6No--/c_limit,f_auto,fl_progressive,q_auto,w_800/https://dev-to-uploads.s3.amazonaws.com/uploads/articles/bkba06bdfhv7f8ogdqik.png

Advantages of Graph Databases

Efficient Relationship Handling: Optimised for querying complex (many-to-many) relationships.

Flexibility: Schema-less design allows for dynamic and evolving data structures.

Performance: Fast traversal and query performance for connected data.

Comparing RDBMS with Graph Databases

Feature	RDBMS (Relational Database)	Graph Database
Data Model	Tables with rows and columns	Nodes, edges, and properties
Schema	Fixed schema	Schema-less or flexible schema
Query Language	SQL	Cypher (Neo4j), Gremlin, SPARQL (RDF databases)
Performance	Optimized for ACID transactions and complex joins	Optimized for traversing relationships
Scalability	Vertical scaling (adding more power to a single server)	Horizontal scaling (adding more servers)
Use Cases	Traditional business applications, transactional systems	Social networks, recommendation engines, fraud detection
Example	Customer-Order relationship in an e-commerce system	Social network with users and their friendships

Example

RDBMS

```sql
-- Create tables
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100)
);

CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    order_date DATE,
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);

-- Insert data
INSERT INTO Customers (customer_id, name, email) VALUES (1, 'Alice', 'alice@exa
INSERT INTO Orders (order_id, order_date, customer_id) VALUES (101, '2024-07-23

-- Query data
SELECT Customers.name, Orders.order_date
FROM Customers
JOIN Orders ON Customers.customer_id = Orders.customer_id
WHERE Customers.name = 'Alice';
```

Graph

```
-- Create nodes and relationships
CREATE (c:Customer {customer_id: 1, name: 'Alice', email: 'alice@example.com'})
CREATE (o:Order {order_id: 101, order_date: '2024-07-23'})
CREATE (c)-[:PLACED_ORDER]->(o);

-- Query data
MATCH (c:Customer {name: 'Alice'})-[:PLACED_ORDER]->(o:Order)
RETURN c.name, o.order_date;
```

https://neo4j.com/product/auradb/

Neo4j

Definition: Neo4j is a popular open-source graph database.

Data Model: Uses the property graph model with nodes, relationships, and properties.

Key Features:

- ACID compliance
- High availability
- Flexible schema
- **Cypher** query language

Cypher Query Language

Cypher is a declarative graph query language used by Neo4j.

```
CREATE (a:Person {name: 'Alice', age: 30})
CREATE (b:Person {name: 'Bob', age: 25})
CREATE (a)-[:FRIEND_OF]->(b)
```

Basic Syntax: Simple and expressive syntax for creating and querying graph data.

```
-- Create nodes and relationships
CREATE (p:Person {name: 'Alice'})-[:FRIEND_OF]->(b:Person {name: 'Bob'})

-- Query for friends of Alice
MATCH (p:Person {name: 'Alice'})-[:FRIEND_OF]->(friend)
RETURN friend.name
```

Advanced Features of Neo4j

Indexing and Constraints: Improve query performance and enforce data integrity.

Full-Text Search: Allows searching within node and relationship properties.

Graph Algorithms: Built-in algorithms for analytics, such as shortest path, PageRank, and community detection.

Visualisation Tools: Tools like Neo4j Bloom for visualising and exploring graph data.

Python Virtual Environments

Python Virtual Environments

- What is a Virtual Environment?
  - An isolated Python workspace.
  - Allows independent Python package installations.

- Benefits
  - **Isolation**: each project has its own dependencies.
    - Avoid dependency conflicts.
  - **Clean Workspace**: no clutter of global packages.
    - Maintain project-specific packages
  - **Version Control**: manage package versions precisely per project.
  - **Collaboration**: easy sharing of requirements (requirements.txt).
    - Enhance reproducibility across different systems.

Create and Delete a Virtual Environment

- Step-by-step

```
# Create virtual environment
python -m venv myenv

# Activate on Windows
myenv\Scripts\activate

# Activate on macOS/Linux
source myenv/bin/activate
```

- Confirm activation

```
# Verify active environment
pip -V
which python
```

- Deactivate Environment

```
deactivate
```

- Delete Environment
  - Simply delete the folder

Manage Python Packages

- Install packages
  - pip install package_name
- Save requirements
  - pip freeze > requirements.txt
- Install from requirements
  - pip install -r requirements.txt

Idea Usage Scenario and Best Practices

- Ideal Usage Scenario
  - Python-only development with straightforward dependency management.
  - Quick testing and prototyping of Python scripts.
  - Small applications that don't need system-level isolation.

- Best Practices
  - Create a separate virtual environment for each project.
  - Regularly update and maintain your requirements file.
  - Avoid using global Python installation for project-specific dependencies.