Outline

- The NoSQL Movement
- What is NoSQL?
- NoSQL Database Categories
  - Key-Value stores
  - ☞ **Document-oriented Stores**
  - Column-oriented Databases
  - Graph based Databases

Tuple-based Stores

- A **tuple store** is similar to a key-value store, but it pairs a unique key with a vector of data instead of a single value associated with a key.
  - `marc -> ("Marc", "McLastName", 25, "Germany")`
- Unlike rigid relational databases, tuple stores do not require a uniform structure for the data – Each tuple can very in length and composition, providing *schema-less* and schema flexible.
- Tuple stores can also organize entries into semantical groups known as collections or tables, akin to tables in relational databases
  - **Example**: Collection "Person" may include two entires:

```
Person:marc -> ("Marc", "McLastName", 25, "Germany")
Person:harry -> ("Harry", "Smith", 29, "Belgium")
```

Document Stores

- **Document stores**, also known as **document-oriented databases**, employ a key-value approach to data storage but with notable distinctions from key-value stores.
- In a document database, data is stored as documents.
- These documents contain both structural information and data values, providing a more flexible and expressive data model compared key-value store

```
{
   Title      = "Harry Potter"
   ISBN       = "111-1111111111"
   Authors    = [ "J.K.  Rowling" ]
   Price      = 32
   Dimensions = "8.5 x 11.0 x 0.5"
   PageCount  = 234
   Genre      = "Fantasy"
}
```

Document Stores

- Document databases are among the most widely used types of NoSQL databases.
- They offer APIs or query languages that allow you to retrieve documents based on attribute values.
- These document-oriented databases offer flexibility, scalability, and ease of development, making them suitable for a wide range of applications and use cases.

Document Stores vs. Key Value Store

Document Stores

Key	Value
CGHScore	{ "Title": "Chris Gayle Innings…" "UserId": "nrules" "Tags": ["IPL", "Cricket", "RCB"] "Body": "CG at his best…" }
RPForm	{ "Title": "Ricky P Innings…" "UserId": "nrules" "Tags": ["IPL", "Cricket", "MI"] "Body": "RP not his best…" } …

vs.

Key Value Stores

Key	Value
CGHScore	….
RPForm	…

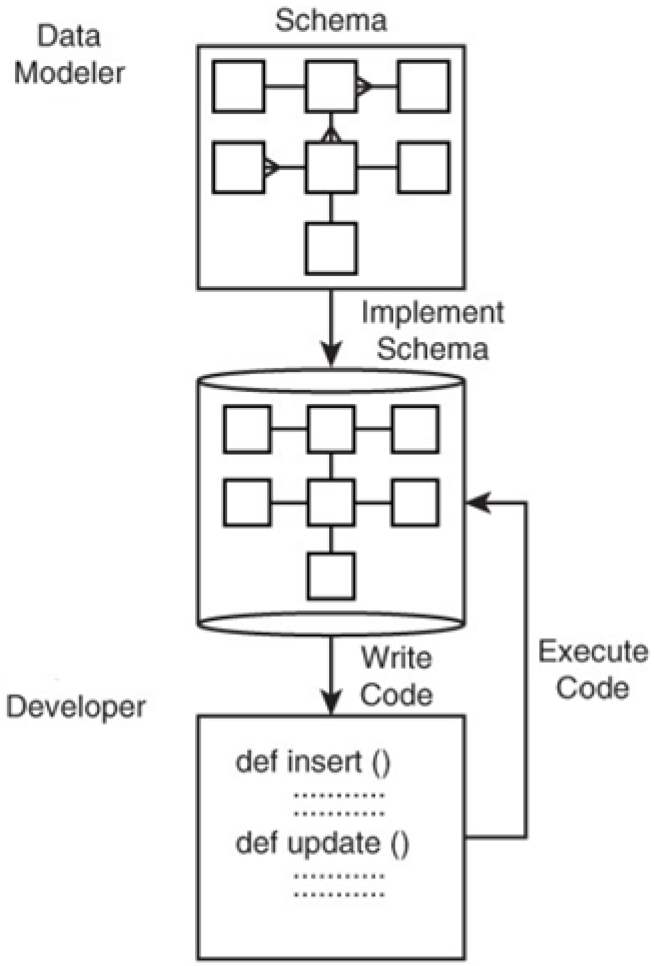
- In a key-value store the data is considered to be inherently opaque to the database, whereas a document-oriented system relies on internal structure in the *document* in order to extract metadata that the database engine uses for further optimization.
- Document stores provide a richer API than key-value stores

Schema Definition in Relational vs. Document Databases

- **Relational Databases**
  - Relational databases require an explicitly defined schema.
  - Database designers must define tables and their structure before developers can interact with the database.
- **Document Databases**
  - Document databases do not require an explicitly defined schema.
  - Each document in the database can have a different structure, allowing for flexibility and adaptability.
    - For instance, each product document can contain attributes specific to its category, such as "Name," "Price," "Description," and additional properties unique to certain product types (e.g., "Author" for books, "Resolution" for monitors).

Schema Definition in Relational vs. Document Databases

Relational Databases



Document Databases

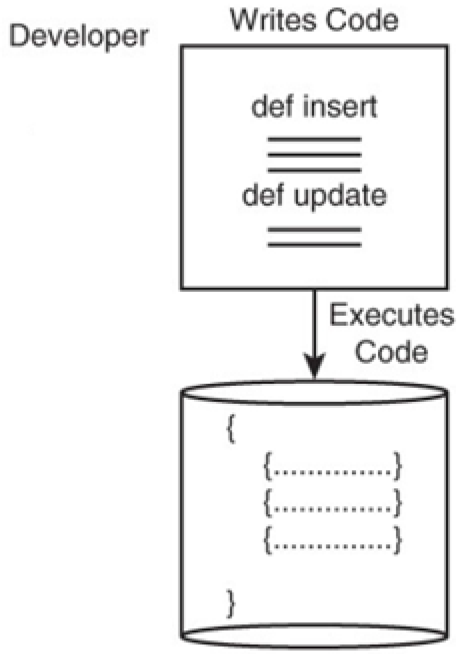


Figure. Relational databases require an explicitly defined schema, but document databases do not.

Schema-On-Read and Dynamic Schema Evolution

- **Schema-On-Read:** Document databases allow data to be inserted without prior schema definition. The data structure materializes naturally as documents are inserted into the database, adapting to the content of each document.
- **Dynamic Schema Evolution:** New attributes can be introduced to documents on-the-fly. As application requirements change or new features are introduced, developers can seamlessly add new attributes to documents without the need for schema migrations.
- **Developer Responsibility:** While adding attributes is flexible, it's the developer's responsibility to ensure that the application logic consistently manages these attributes across the database. Failure to do so could result in data inconsistency or unexpected behavior.

Polymorphic Schema

- Document databases are often described as having a **polymorphic schema**, deriving from the Latin term "polymorph," meaning "many shapes."
- While there may not be explicit constraints enforced by the database, the application logic typically assumes a certain structure for the documents it retrieves.
  - For example, consider a collection of documents representing various types of products. While each product document may have different attributes based on its type (e.g., electronic, clothing, or furniture), the application code expects certain common attributes such as "name," "price," and "description" to be present across all documents.

Benefits of Polymorphic Schema

- **Flexibility**: The polymorphic schema allows for the storage of diverse data types within a single collection, accommodating a wide range of document structures without requiring predefined schemas.
- **Adaptability**: Applications can evolve over time without the need for extensive schema migrations. New document types can be seamlessly introduced, and existing documents can be modified without disrupting the database schema.
- **Simplified Development**: Developers can focus on application logic and functionality without being constrained by rigid data schemas. This promotes agile development practices and faster iteration cycles.

Document Database Systems

- **Apache CouchDB**
  - An open-source document-oriented database that uses JSON to store data, JavaScript as its query language (MapReduce views), and HTTP for its API.
- **MongoDB**
  - A cross-platform, document-oriented database that provides high performance, high availability, and easy scalability. It stores data in flexible, JSON-like documents.
- **DocumentDB**
  - A fully managed NoSQL database service offered by Amazon Web Services (AWS). It is compatible with MongoDB, making it easy for MongoDB users to migrate their existing applications to DocumentDB.

Document Database Systems (cont.)

- **IBM Domino**: A platform for social business applications. It provides document storage and replication capabilities, allowing users to create and manage documents with rich text, attachments, and structured data.
- **OrientDB**: A multi-model database management system that supports graph, document, object, and key/value models. It combines the flexibility of documents with the power of graphs for data representation and querying.

Outline

- The NoSQL Movement
- What is NoSQL?
- NoSQL Database Categories
  - Key-Value stores
  - **Document-oriented Stores**
    - Tuple Stores
    - Document Stores
    - ☞ **Designing Documents**
    - Operations on Document Databases
  - Column-oriented Databases
  - Graph based Databases

Anatomy of Document

- Documents contain both structure information and data.
- A **document** is essentially a set of name-value pairs, where each name is also referred to as a key, associated with a corresponding value.
  - **Names** are represented as strings of characters, providing a unique identifier for each attribute within the document.
  - **Values** in a document can be basic data types such as numbers, strings, and Booleans. Additionally, they can also represent more complex structures like arrays and objects.
- The name in a name-value pair indicates an attribute and the value assigned to that name represents the actual data stored for that attribute.

Document Representation

- Most modern NoSQL databases choose to represent documents using JSON or XML
  - JSON document

```
{
   "title": "Harry Potter",
   "authors": ["J.K.  Rowling", "R.J.  Kowling"],
   "price": 32.00,
   "genres": ["fantasy"],
   "dimensions": {
            "width": 8.5,
            "height": 11.0,
            "depth": 0.5
   },
   "pages": 234,
   "in_publication": true,
   "subtitle": null
}
```

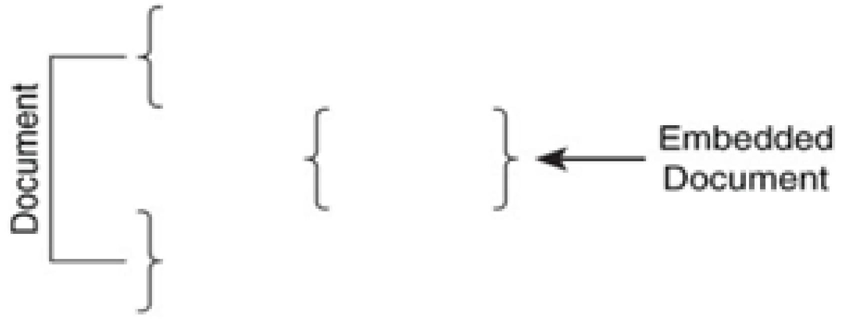
  - XML document

```
<customer_record>
<customer_id>187693</customer_id>
<name>“Kiera Brown”</name>
<address>
<street>“1232 Sandy Blvd.”</street>
<city>“Vancouver”</city>
<state>“Washington”</state>
<zip>“99121”</zip>
</address>
<first_order>“01/15/2013”</first_order>
<last_order>“06/27/2014”</last_order>
</customer_record>
```

Embedded Documents

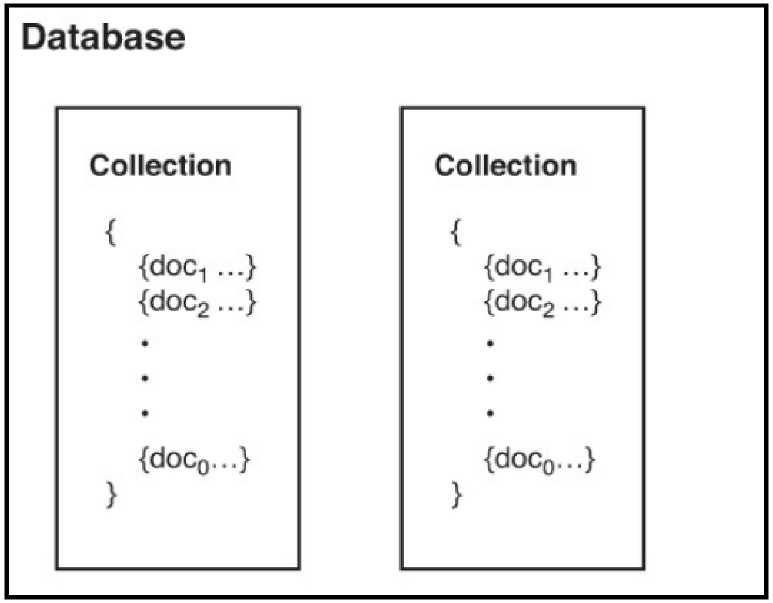
- Documents can embed other documents within them.
- This capability allows users to store related data in a single document.
- Embedding is useful for storing data that is frequently accessed together or shares a close relationship

```
{
  firstName: “Bob”,
  lastName: “Wilson”,
  positionTitle: “Manager”,
  officeNumber: “2-130”,
  officePhone: “555-222-3478”,
  hireDate: “1-Feb-2010”,
  terminationDate: “12-Aug-2014”
  PreviousPositions: [
    { position: “Analyst”,
    StartDate:“1-Feb-2010”,
    endDate:“10-Mar-2011”
    } {
    position: “Sr. Analyst”,
    startDate: “10-Mar-2011”
    endDate:“29-May-2013”
    }
  ]
}
```



Database, Collections and Documents

- **Database** is the highest-level logical structure in a document-oriented store and contains collections of documents
- **Documents** are generally grouped into collections of similar documents. **Collections** are lists of documents.
  - Note that documents within the same collection don't necessarily have to adhere identical structures, but they should share some common attributes or characteristics
- Collections support additional data structures, such as indexes, to enhance the efficiency of operations performed on those groups of documents.



Document Identification and Primary Key

- Each document needs a unique identifier to distinguish it within a collection.
- This identifier can be included directly within the document's attributes, typically as **a primary key**.
- Example: MongoDB uses "**_id**" as the primary key attribute for each document.
  - The _id field can be specified by the user or left out.
  - If omitted, MongoDB will automatically generate a unique, random identifier for the document.

Document Identification and Primary Key (cont.)

- The primary key serves multiple purposes.
  - It ensures data uniqueness within the collection, preventing duplicate entries.
  - It facilitates efficient data retrieval and manipulation.
  - The primary key can be utilized as a partitioning key in distributed environments, enabling the database to determine where each document should be stored based on a hash of its primary key.

Organizing Documents into Collections

- When modeling document databases, a critical decision revolves around organizing documents into collections.
- Ideally, each collection should store documents related to a specific type of entity or share common attributes and structures.
- Storing heterogeneous document types within the same collection can lead to performance issues. Filtering and querying collections with diverse document structures can be slower and less efficient

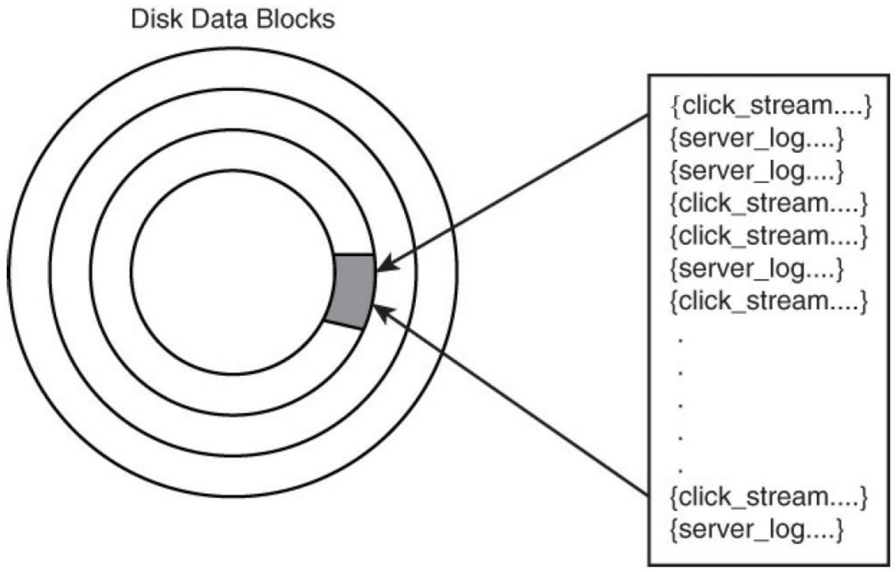


Figure Mixing document types can lead to multiple document types in a disk data block. This can lead to inefficiencies because data is read from disk but not used by the application that filters documents based on type.

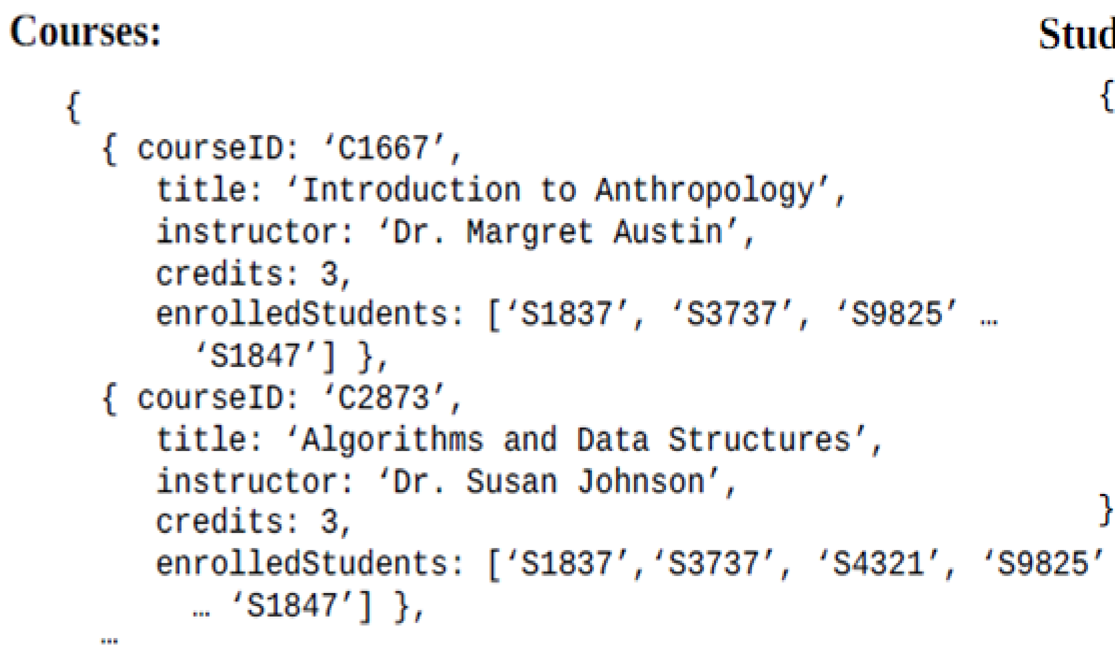
Modeling One-to-Many Relations

- Designing document databases offers designers a high degree of flexibility, especially when modeling relations between documents.
- **One-to-Many Relations**: Both entities can be represented using a document embedded within another document

```
{
   customer_id: 76123,
   name: 'Acme Data Modeling Services',
   person_or_business: 'business',
   address : [
                  { street: '276 North Amber St',
                    city: 'Vancouver',
                    state: 'WA',
                    zip: 99076} ,
                  { street: '89 Morton St',
                    city: 'Salem',
                    state: 'NH',
                    zip: 01097}
                 ]
   }
```

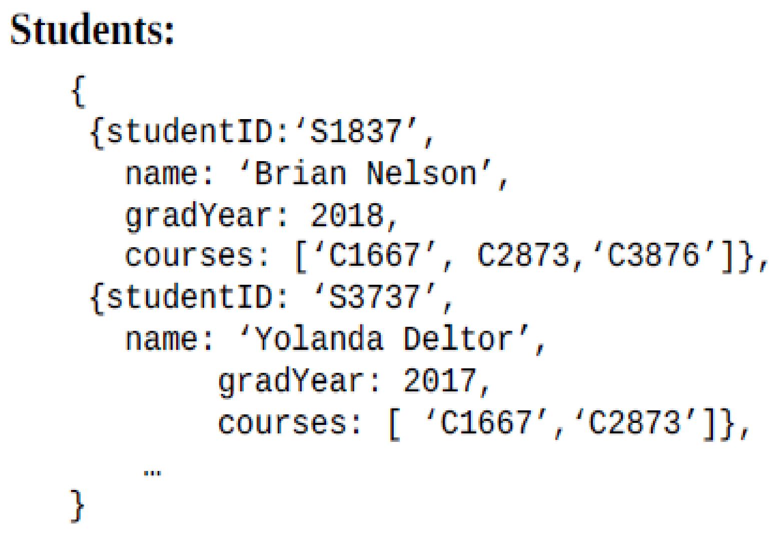
Modeling Many-to-Many Relations

- **Many-to-Many Relations** are modeled using two separate collections – one for each type of entity. Each document in either collection maintains a list of identifiers that reference related entities from the other collection.



Courses:

```
{
 { courseID: 'C1667',
    title: 'Introduction to Anthropology',
    instructor: 'Dr. Margret Austin',
    credits: 3,
    enrolledStudents: ['S1837', 'S3737', 'S9825' …
      'S1847'] },
 { courseID: 'C2873',
    title: 'Algorithms and Data Structures',
    instructor: 'Dr. Susan Johnson',
    credits: 3,
    enrolledStudents: ['S1837','S3737', 'S4321', 'S9825'
      … 'S1847'] },
 …
```



Students:

```
{
 {studentID:'S1837',
   name: 'Brian Nelson',
   gradYear: 2018,
   courses: ['C1667', C2873,'C3876']},
 {studentID: 'S3737',
   name: 'Yolanda Deltor',
        gradYear: 2017,
        courses: [ 'C1667','C2873']},
     …
}
```

Modeling Hierarchies

- Hierarchies, representing parent-child or part-subpart relations, are common in many data scenarios.
- Document databases offer a simple technique for implementing hierarchies by maintaining references between parent and child entities

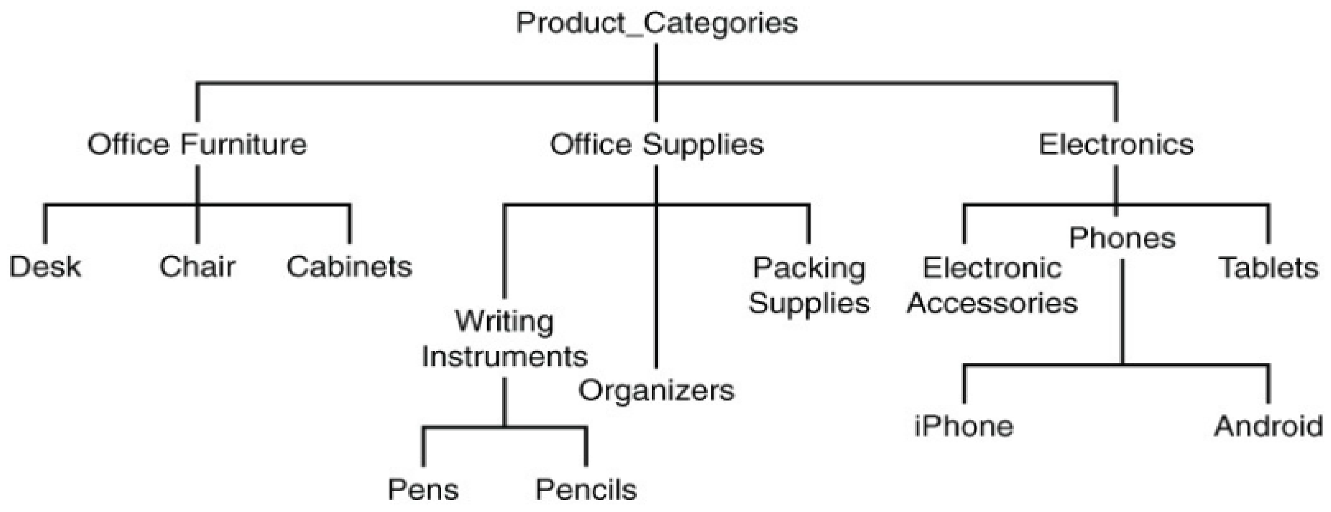
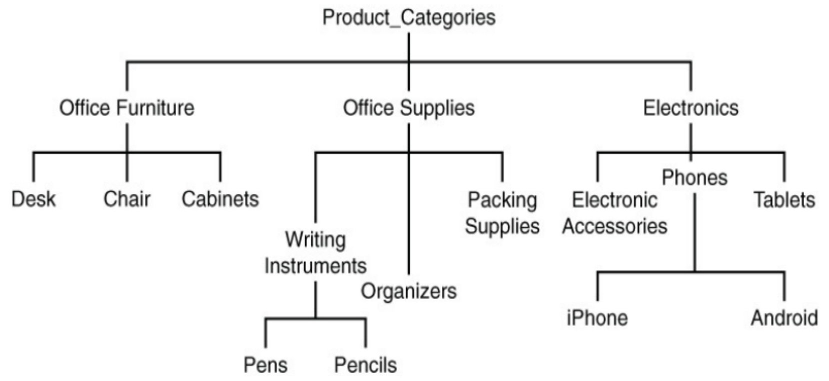


Figure: In a hierarchical relationship scenario, where entities are organized in a tree-like structure with parent and child nodes.

Modeling Hierarchies in Document Databases

- **Parent-Child References**: Each document representing an entity in the hierarchy contains a reference to either its parent or its children, depending on the direction of traversal required.
  - Each child document maintains a reference to its parent document.
  - Alternatively, each parent document can maintain a list of references to its child documents.



```
{
   {productCategoryID: ‘P01’, name:‘Product Categories’,
    childrenIDs: [‘P37’,‘P39’,‘P41’]},
    {productCategoryID: ‘PC37’, name:‘Office Supplies’,
     childrenIDs: [‘PC72’,‘PC73’,‘PC74”]},
     {productCategoryID: ‘PC72’, name:‘Writing
       Instruments’, childrenIDs: [‘PC233’,‘PC234’]’},
      {productCategoryID: ‘PC233’, name:‘Pencils’}
}
```

```
{
   {productCategoryID: ‘PC233’, name:‘Pencils’,
     parentID:‘PC72’},
   {productCategoryID: ‘PC72’, name:‘Writing Instruments’,
     parentID: ‘PC37”},
   {productCategoryID: ‘PC37’, name:‘Office Supplies’,
     parentID: ‘P01’},
   {productCategoryID: ‘P01’, name:‘Product Categories’ }
}
```

Outline

- The NoSQL Movement
- What is NoSQL?
- NoSQL Database Categories
  - Key-Value stores
  - **Document-oriented Stores**
    - Tuple Stores
    - Document Stores
    - Designing Documents
    - ☞ **Operations on Document Databases**
  - Column-oriented Databases
  - Graph based Databases

Basic Operations in Document Databases

- Basic operations on document databases
  - Inserting documents
  - Deleting documents
  - Updating documents
  - Retrieving documents
- Unlike relational databases, there is no standard data manipulation language (DML) used across document databases to perform these operations.
- Instead, document databases provide APIs or query languages tailored to their specific implementation. Developers interact with the database using these APIs or query languages to perform insertions, deletions, updates, and retrievals.

Inserting Documents

- The examples that follow use a command structure similar to that of MongoDB, currently the most commonly used document database
- Add a single document describing a book titled “Mother Night” to the “books” collection:

```
db.books.insert({“title”: “Mother Night”, “author”:
“Kurt Vonnegut, Jr.”})
```

- Add a book document with a specific book ID, title and author name to the “books” collection:

```
db.books.insert({book_id: 1298747, “title”:“Mother
Night”, “author”: “Kurt Vonnegut, Jr.”})
```

Bulk Inserting

- In many cases, performing bulk inserts can be more efficient than a series of individual inserts.

```
db.books.insert(
    [
        {“book_id”: 1298747,
        “title”:“Mother Night”,
        “author”: “Kurt Vonnegut, Jr.”},
        {“book_id”: 639397,
        “title”:“Science and the Modern World”,
        “author”: “Alfred North Whitehead”},
        {“book_id”: 1456701,
        “title”:“Foundation and Empire”,
        “author”: “Isaac Asimov”}
    ]
}
```

Deleting Documents

- **Delete All Documents**
  - Delete all documents in the books collection:

    ```
    db.books.remove()
    ```
  - Note that the collection still exists, but it is empty.
- **Delete Documents by Primary Key**:
  - Document databases support fast filter of a particular item from a collection with the primary key of each item.
  - Delete the book with primary key:

    ```
    db.books.remove({“book_id”: 639397})
    ```
- **Delete Documents by Filter for Other Attributes:**
  - Delete the books by Kurt Vonnegut:

    ```
    db.books.remove({“author”: “Kurt Vonnegut” })
    ```

Updating Documents

- The update command adds an attribute name (key) if it does not exist and sets the value as indicated. If the key already exists, the update command changes the value associated with it.
- **Adding a New Attribute**
  - Add the quantity (key) of book_id 1298747 with the value of 10 to the books collection:

```
db.books.update ({“book_id”: 1298747},
                 {$set {“quantity” : 10 }})
```

    - The first parameter specifies the criteria to identify the document(s) to update.
    - The second parameter uses the **$set** operator to specify which attribute(s) and value(s) should be updated or added.

Updating Documents (cont.)

- **Updating a Numeric Value**
  - Change the quantity of book_id 1298747 from 10 to 15.

```
db.books.update ({“book_id”: 1298747},
                            {$inc {“quantity” : 5 }})
```

    - The second parameter uses the increment operator ($inc), which is used to increase the value of a key by the specified amount.

Retrieving Documents

- **Retrieve All Documents**
  - Return all documents in a collection, you can use find() method without any parameters.

    ```
    db.books.find()
    ```

- **Retrieve Documents by Criteria**:
  - Fast retrieval of a particular document from a collection **with the primary key** of each document
  - Also support the retrieval of documents **based on simple filters** for other attributes.
  - Return all books by Kurt Vonnegut, Jr.:

    ```
    db.books.find({“author”: “Kurt Vonnegut, Jr.”})
    ```

    - The first parameter, if provided, specifies the criteria to filter documents.

Retrieving Documents (cont.)

- **Retrieve by Range**
  - Retrieve all books with a quantity greater than or equal to 10 and less than 50

```
db.books.find({“quantity”:{“$gte” : 10, “$lt” : 50 }})
```

- **Supported Conditions and Booleans**
  - $lt : Less than
  - $let : Less than or equal to
  - $gt : Greater than
  - $gte : Greater than or equal to
  - $in : Query for values of a single key
  - $or : Query for values in multiple keys
  - $not : Negation

Indexes in Document Databases

- To speed up query performance, most document stores provide a variety of indexes over collections in a manner very similar to table indexes in a relational database

- Types of Indexes
  - **Unique indexes**
    - Ensure that each document in a collection has a unique value for a specific field.
    - Similar to primary keys in relational databases, unique indexes enforce data integrity.
  - **Non-unique indexes**
    - Allow duplicate values for indexed fields.

Indexes in Document Databases (cont.)

- Types of Indexes (cont.)
  - **Compound indexes**
    - Combine multiple fields into a single index.
    - Enable efficient querying on multiple criteria.
  - **Geospatial indexes**
    - Optimize queries involving geographical data, such as coordinates or shapes.
    - Support spatial queries like finding points within a radius or polygons intersecting a region.
  - **Text-based indexes**
    - Facilitate full-text search functionality.
    - Enable queries to match words or phrases within text fields.

Document Database APIs

- A document database system offers versatile APIs for interacting with its database, catering to diverse programming languages and use cases. Understanding the APIs is crucial for effective database management and application development.
- MongoDB APIs
  - MongoDB provides a range of APIs tailored to different programming languages, including Java, Python, Node.js, and more.
  - Each API offers methods and functions for performing CRUD (Create, Read, Update, Delete) operations, aggregation, indexing, and other database tasks.
  - APIs are designed to streamline development, enhance performance, and provide seamless integration with MongoDB databases.

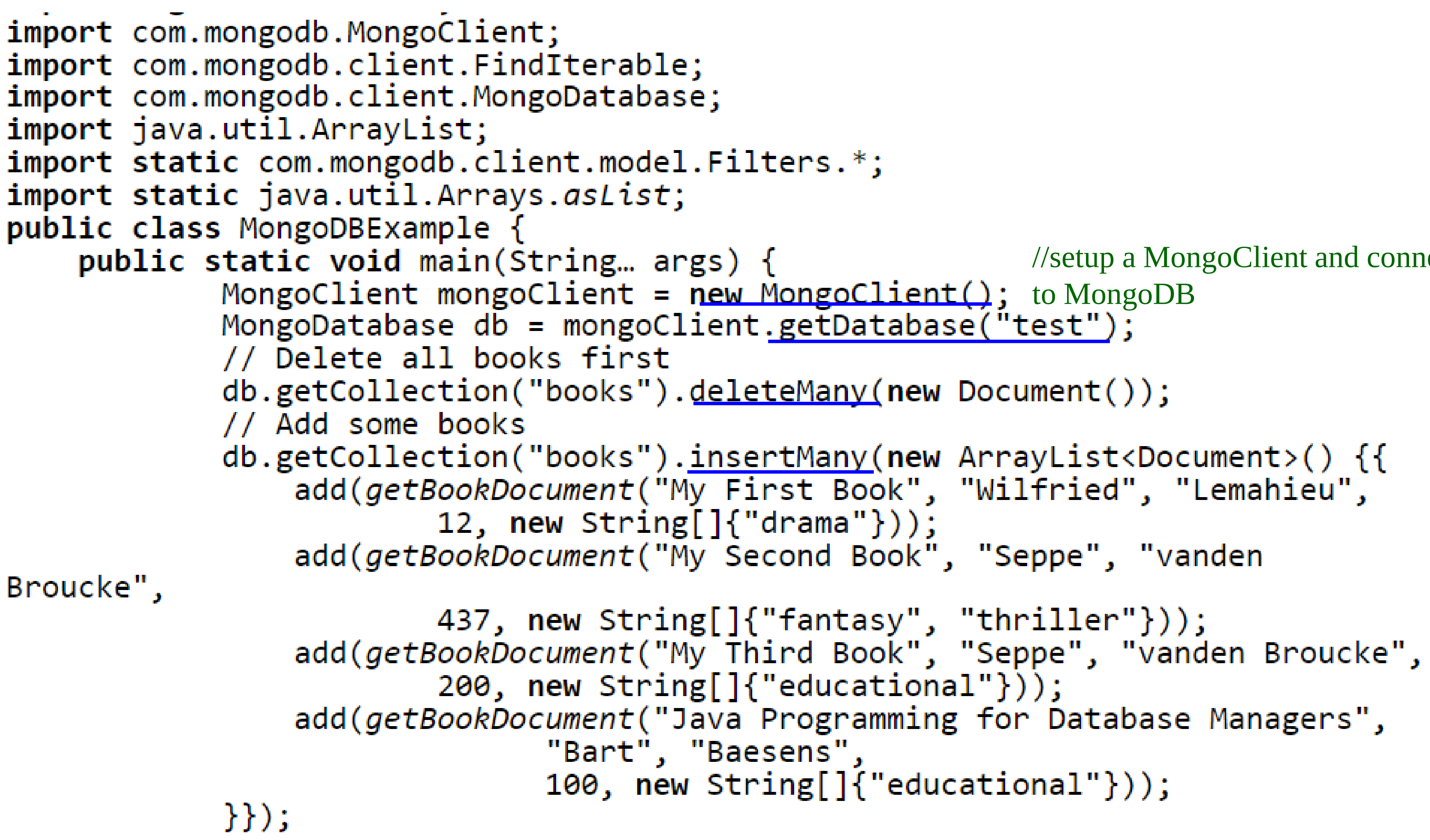
MongoDB APIs

- **MongoClient()**
  - The MongoClient() API is the entry point for connecting to a MongoDB deployment.
  - It represents a connection pool and manages connections to the MongoDB server.
  - Accepts connection string parameters for configuration.
- **getDatabase()**
  - The getDatabase() API is used to access a specific database within the MongoDB deployment.
  - It returns a database object representing the specified database.
- **deleteMany()**
  - The deleteMany() API is used to delete multiple documents that match the specified filter from a collection.
  - It accepts a filter document as a parameter to specify the criteria for deletion.

MongoDB APIs (cont.)

- **insertMany()**
  - The insertMany() API is used to insert multiple documents into a collection.
  - It accepts an array of documents to be inserted as a parameter.
- **find()**
  - The find() API is used to query documents from a collection.
  - It returns a cursor object that can be iterated to access the retrieved documents.
- **close()**
  - The close() API is used to close the MongoClient connection.
  - It releases any resources held by the MongoClient instance.
- …

MongoDB APIs: Usage Example



```
import com.mongodb.MongoClient;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoDatabase;
import java.util.ArrayList;
import static com.mongodb.client.model.Filters.*;
import static java.util.Arrays.asList;
public class MongoDBExample {
    public static void main(String… args) {
            MongoClient mongoClient = new MongoClient();
            MongoDatabase db = mongoClient.getDatabase("test");
            // Delete all books first
            db.getCollection("books").deleteMany(new Document());
            // Add some books
            db.getCollection("books").insertMany(new ArrayList<Document>() {{
                add(getBookDocument("My First Book", "Wilfried", "Lemahieu",
                        12, new String[]{"drama"}));
                add(getBookDocument("My Second Book", "Seppe", "vanden
Broucke",
                        437, new String[]{"fantasy", "thriller"}));
                add(getBookDocument("My Third Book", "Seppe", "vanden Broucke",
                        200, new String[]{"educational"}));
                add(getBookDocument("Java Programming for Database Managers",
                              "Bart", "Baesens",
                              100, new String[]{"educational"}));
            }});
```

//setup a MongoClient and connect to MongoDB

```
        // Perform query
        FindIterable<Document> result = db.getCollection("books").find(
                        and(
                                    eq("author.last_name", "vanden
Broucke"),
                                    eq("genres", "thriller"),
                                    gt("nrPages", 100)));
        for (Document r: result) {
            System.out.println(r.toString());
            // Increase the number of pages:
            db.getCollection("books").updateOne(
                  new Document("_id", r.get("_id")),
                  new Document("$set",
                      new Document("nrPages", r.getInteger("nrPages") +
100)));
        }

        mongoClient.close();
    }

    public static Document getBookDocument(String title,
            String authorFirst, String authorLast,
            int nrPages, String[] genres) {
        return new Document("author", new Document()
                                                            .append("first_name",
authorFirst)
                                                            .append("last_name",
authorLast))
                                              .append("title", title)
                                              .append("nrPages", nrPages)
                                              .append("genres", asList(genres));
    }
}
```

The Running Result

- The result obtained from running the Java code with MongoDB is:

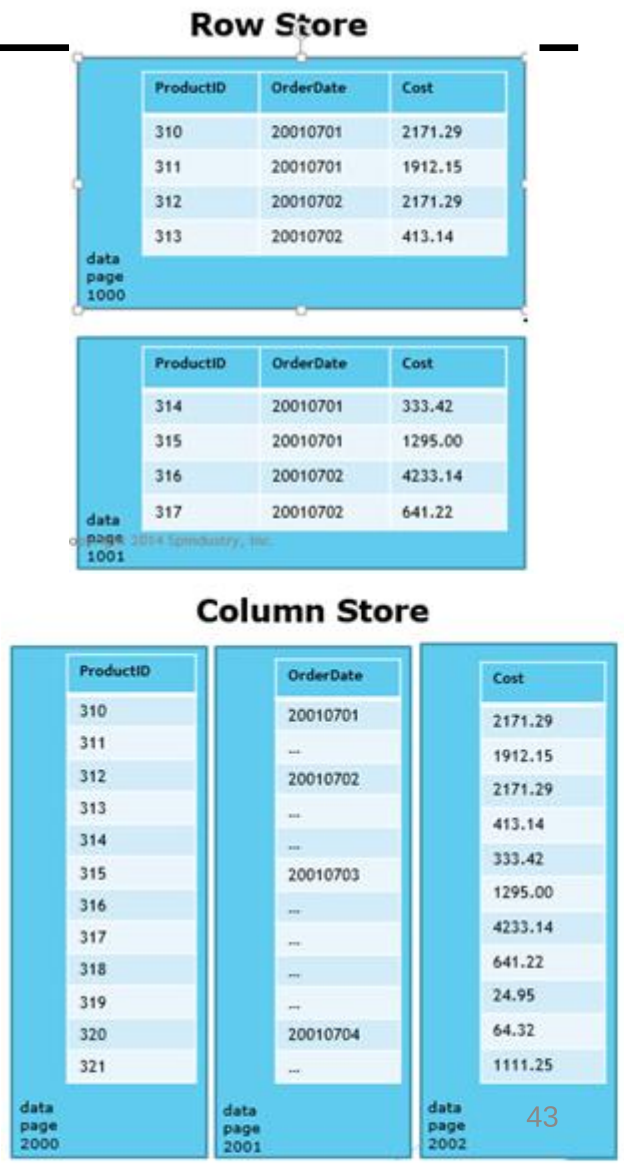
```
Document{{_id=567ef62bc0c3081f4c04b16c,
          author=Document{{first_name=Seppe,
          last_name=vanden Broucke}},
          title=My Second Book, nrPages=437,
          genrese=[fantasy, thriller]}}
```

Outline

- The NoSQL Movement
- What is NoSQL?
- NoSQL Database Categories
  - Key-Value stores
  - Document-oriented Stores
  - ☞ **Column-oriented Databases**
  - Graph based Databases

Column-Oriented Databases

- A **column-oriented database** (also kwon as **column family database**) arranges table data into segments of columns rather than rows, diverging from the typical row-based structure seen in most DBMS implementations.
- In a row-based system, data retrieval is optimized for fetching entire rows efficiently.
- However, a column-based system excels when dealing with sparse data, where columns may contain numerous null values.



Row Store

ProductID	OrderDate	Cost
310	20010701	2171.29
311	20010701	1912.15
312	20010702	2171.29
313	20010702	413.14

ProductID	OrderDate	Cost
314	20010701	333.42
315	20010701	1295.00
316	20010702	4233.14
317	20010702	641.22

Column Store

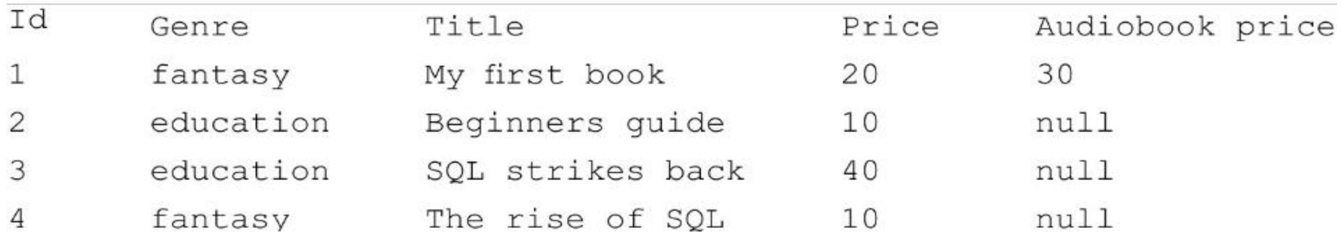
ProductID
310
311
312
313
314
315
316
317
318
319
320
321

OrderDate
20010701
...
20010702
...
...
20010703
...
...
...
...
20010704
...

Cost
2171.29
1912.15
2171.29
413.14
333.42
1295.00
4233.14
641.22
24.95
64.32
1111.25

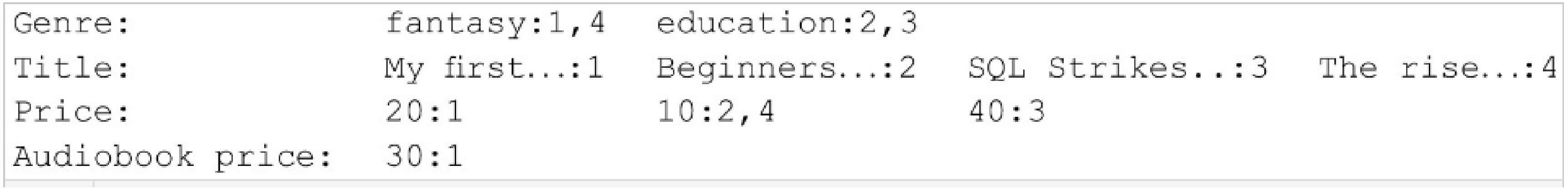
Row-Based vs. Column-Oriented Databases

- **Row-based database**
  - Not efficient for operation across the entire dataset (e.g., filtering prices above 20)
  - Require indexes, adding overhead



Id	Genre	Title	Price	Audiobook price
1	fantasy	My first book	20	30
2	education	Beginners guide	10	null
3	education	SQL strikes back	40	null
4	fantasy	The rise of SQL	10	null

- In a **column-oriented database**
  - Querying (e.g., filtering prices above 20) is efficient in a single operation
  - All values of a column are stored together on disk
  - Null values do not take up storage space anymore



Genre:	fantasy:1,4	education:2,3		
Title:	My first...:1	Beginners...:2	SQL Strikes..:3	The rise...:4
Price:	20:1	10:2,4	40:3	
Audiobook price:	30:1			

Enhancing Query Performance with Column-Oriented Databases

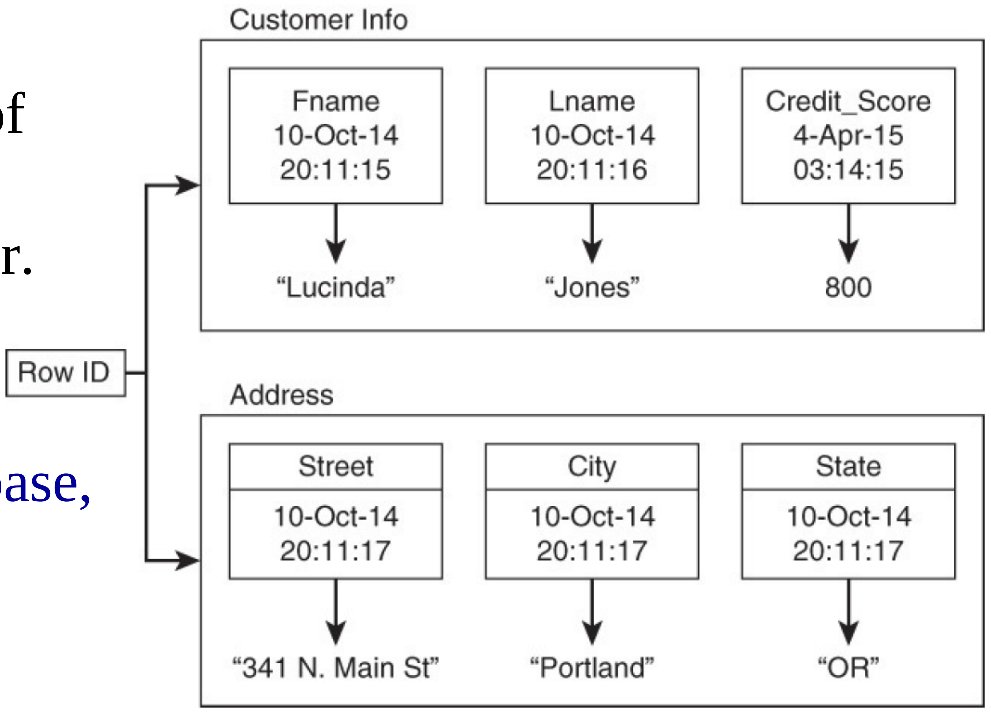
- Significantly enhances query performance, particularly for tasks involving aggregations or accessing specific attributes across multiple records.
  - Well-suited for scenarios where aggregates need to be computed frequently over large sets of related data items across columns.
- Ideal for analytics and reporting purposes, where data access and processing typically occur column-wise, enabling efficient and streamlined operations.
- However, column-oriented databases are less efficient for retrieving all attributes pertaining to a single entity

Column-oriented Databases

- In the Beginning, **Google BigTable**.
  - In 2006, Google published a paper entitled "BigTable: A Distributed Storage System for Structured Data", introducing a revolutionary database paradigm known as the **column family database**
- Popular column-oriented (column family) databases**:**
  - Google BigTable,
  - Cassandra,
  - Apach HBase,
  - Parquet, etc.

Column Family Database Structure

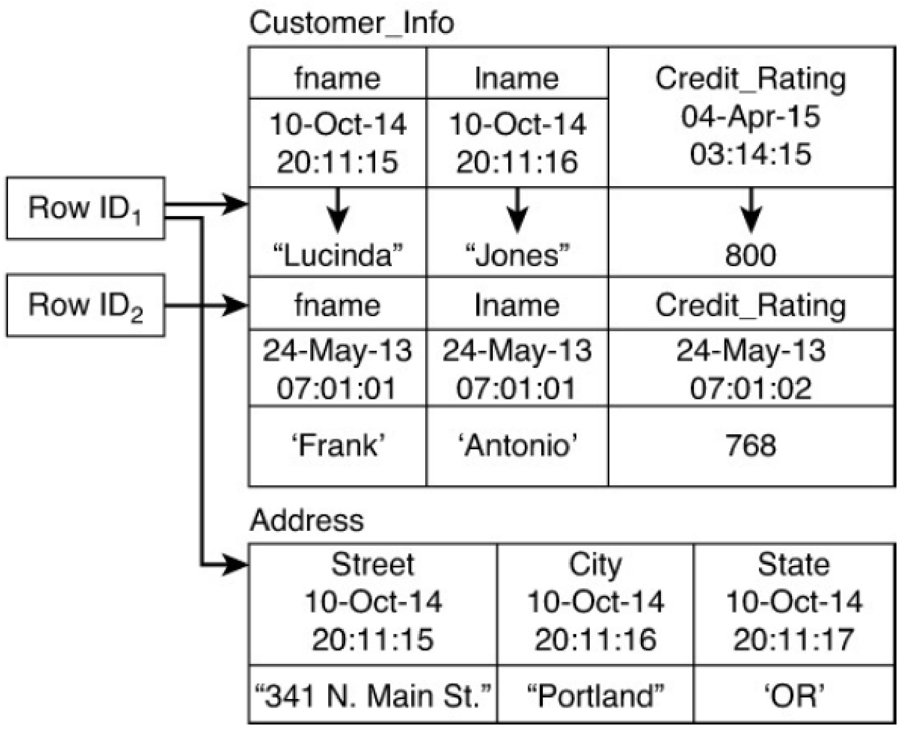
- In column family databases, **rows** are composed of several column families
- Each **column family** consists of a set of related **columns**
  - **Column families** are organized into groups of data items that are frequently used together.
  - Column families are defined prior to implementing the database,
  - but, columns can be dynamically added to a column family



– semi-structured data, no need to update a schema definition

Column Family Database Structure (cont.)

- A **column** consists of three elements:
  - **Unique name** with column family and column: used to reference the column
  - **Value**: the content of the column
  - **Timestamp**: the system timestamp used to determine the valid content.



- Column families for a single row may or may not be near each other when stored on disk, but columns within a column family are kept together

Column Family Databases vs. Key-Value Stores

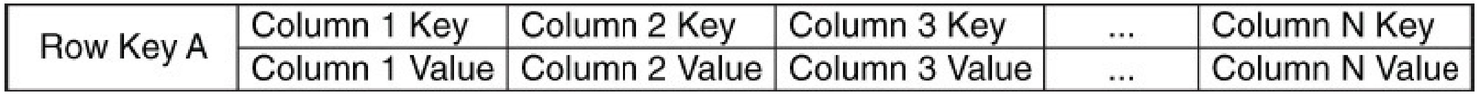
- Column families in column-oriented databases resemble keyspaces in key-value stores
- Developers have the same liberty to add columns and values to column families as they do with keys and values in key-value stores
- Unlike key-value stores, column values are indexed by a row identifier, column family, column and time stamp.

Column Family Databases vs Document Databases

- Column family databases, like document databases, do not mandate all columns in every rows.
  - Certain rows in a column family database may contain values for all columns, while others may hold values for only specific columns within particular column families.
- As document databases enable querying and filtering based on items within the document, column family databases facilitate selecting subsets of data (columns) present in a row.
  - E.g., Cassandra Query Language employs a SELECT statement similar to SQL for data retrieval.

Column Family Databases vs. Relational Databases

- Both utilize unique identifiers for rows of data
  - **Row keys** in column family databases vs. Primary keys in relational databases.
  - Both row keys and primary keys are indexed for rapid retrieval.
- Both systems can be conceptualized as storing tabular data, but their actual storage model are different.
  - Column family databases organize data into structure that maps column names to their corresponding values within a row



Row Key A	Column 1 Key	Column 2 Key	Column 3 Key	...	Column N Key
	Column 1 Value	Column 2 Value	Column 3 Value	...	Column N Value

Column Family Databases vs. Relational Databases

- Column family databases don’t enforce **typed column** – Column values are considered as a sequence of bytes interpreted by applications, not the database, offering significant flexibility to developers.
- Column family databases do not support multirow transactions, where two or more operations are executed as single transaction.
- In column family databases, the necessity for joins and subqueries is minimized.
  - Related data is often stored together within the same row of a column family
  - All the related information for a particular entity or subject can be retrieved in a single query without the need for joins

Column-oriented Databases

Name	Producer	Data model	Querying
BigTable	Google	set of couples (key, {value})	selection (by combination of row, column, and time stamp ranges)
HBase	Apache	groups of columns (a BigTable clone)	JRUBY IRB-based shell (similar to SQL)
Hypertable	Hypertable	like BigTable	HQL (Hypertext Query Language)
Cassandra	Apache (originally Facebook)	columns, groups of columns corresponding to a key (supercolumns)	simple selections on key, range queries, column or columns ranges
PNUTS	Yahoo	(hashed or ordered) tables, typed arrays, flexible schema	selection and projection from a single table (retrieve an arbitrary single record by primary key, range queries, complex predicates, ordering, top-k)

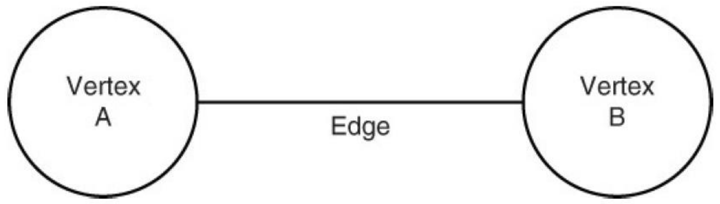
Outline

- The NoSQL Movement
- What is NoSQL?
- NoSQL Database Categories
  - Key-Value stores
  - Document-oriented Stores
  - Column-oriented Databases
  - ☞ **Graph based Databases**

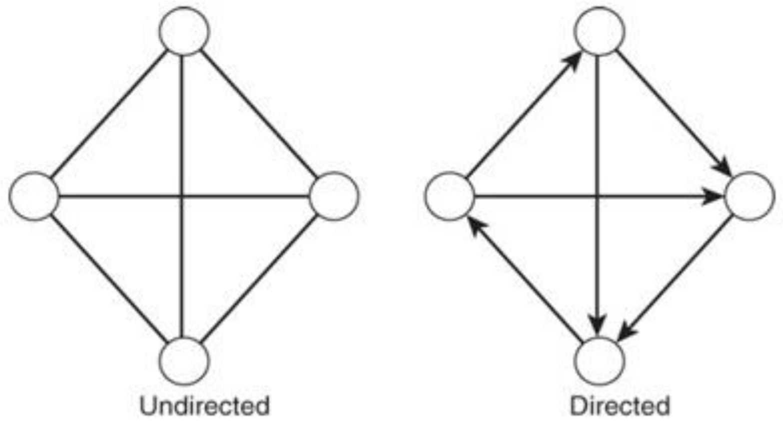
Graph-based Databases

- **Graph databases** apply graph theory to store and manage records.
- A graph is composed of **nodes** (also known as points, or vertices) and **edges** (also referred to as arcs, or lines), which connect these nodes.
- Edges can be unidirectional or bidirectional.
- Nodes represent entities within the databases, each possessing its own set of properties. These properties define the characteristics or attributes of the entity.
- Edges denote the relationships between nodes. Edges can also have associated properties, providing additional context to the relationship.

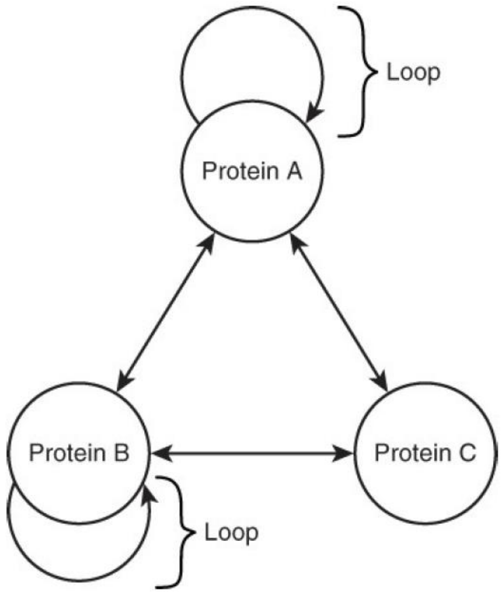
Example: Various Graphs



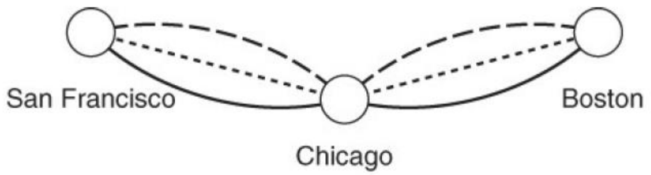
(a) A simple graph with two **nodes** (vertexes) and one **edge**



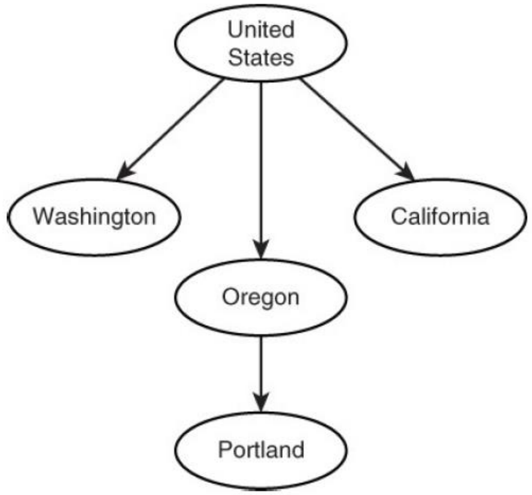
(b) Undirected and **directed** graphs



(c) A loop in an edge that links a node to itself

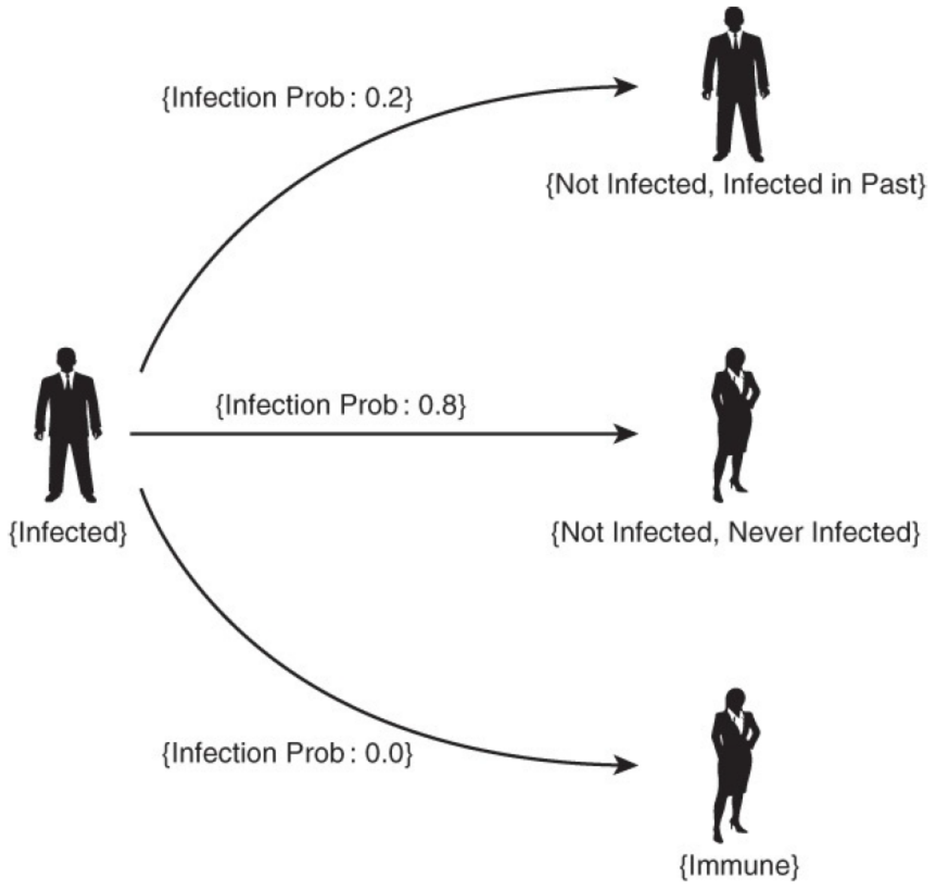


(d) A **multigraph** with multiple edges between nodes



(e) A special type of graph known as a **tree** – hierarchical relationships are modeled

Example: Graph Database Modeling of Spread of Infections



- Vertices represent individuals, each characterized by properties such as infection status, age and weight.
- Edges represent interactions between individuals, signifying potential pathways for the transmission of infections, and may have properties like infection probability.

Figure. The spread of flu and other infectious diseases is modeled as graphs

Graph-based Databases

- Graph-based databases utilize graph structures to organize and store data efficiently.
- Additionally, they excel in executing semantic queries, **enabling users to retrieve information based on the relationships between entities** rather than just their individual attributes.
- Graph databases are sometimes referred to as **hyper-relational databases** due to their ability to model complex relationships between data points

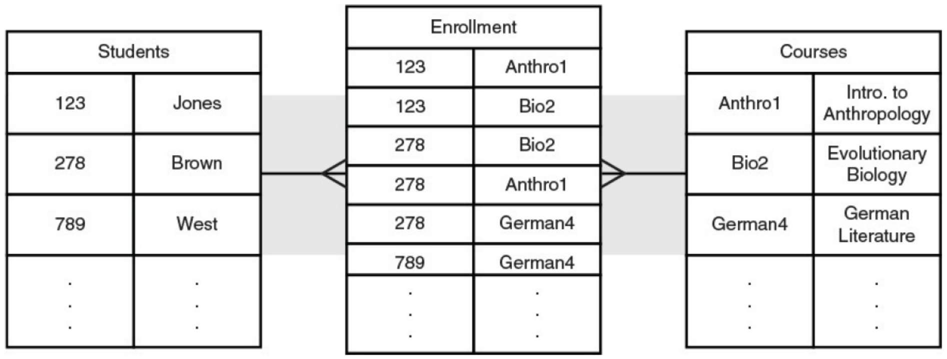
Graph Database Systems

- Popular graph database systems are:
  - Neo4j,
  - OrientDB,
  - Titan,
  - Apach Graph,
  - AllegroGraph,
  - ArangoDB,
  - InfiniteGraph

Relationship Modeling

- **Simplified modeling**
  - In a graph database, 1:1, 1:N, and N:M relationships are represented directly as edges between nodes. – No need for separate relationship tables.
    - vs. In a relational database, a relationship table is needed for N:M relations.
- **Multiple relations between entities**
  - Multiple relations between entities are easily modeled using different types of edges.
- This flexibility allows for the representation of diverse and complex relationships in the data model.

Example: Representing a Relation between Entities

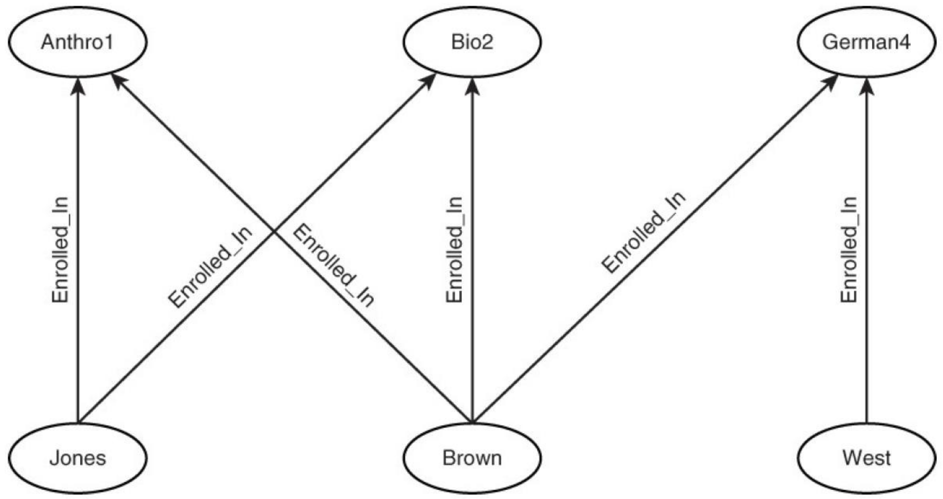


Students	
123	Jones
278	Brown
789	West
⋮	⋮

Enrollment	
123	Anthro1
123	Bio2
278	Bio2
278	Anthro1
278	German4
789	German4
⋮	⋮

Courses	
Anthro1	Intro. to Anthropology
Bio2	Evolutionary Biology
German4	German Literature
⋮	⋮

Figure. Representing a student-course relation in a relational database



The edges between students and courses allow users to quickly query all the courses a particular student is enrolled in.

Figure. Representing a student-course relation in a graph database

Example: Multiple Relations between Entities

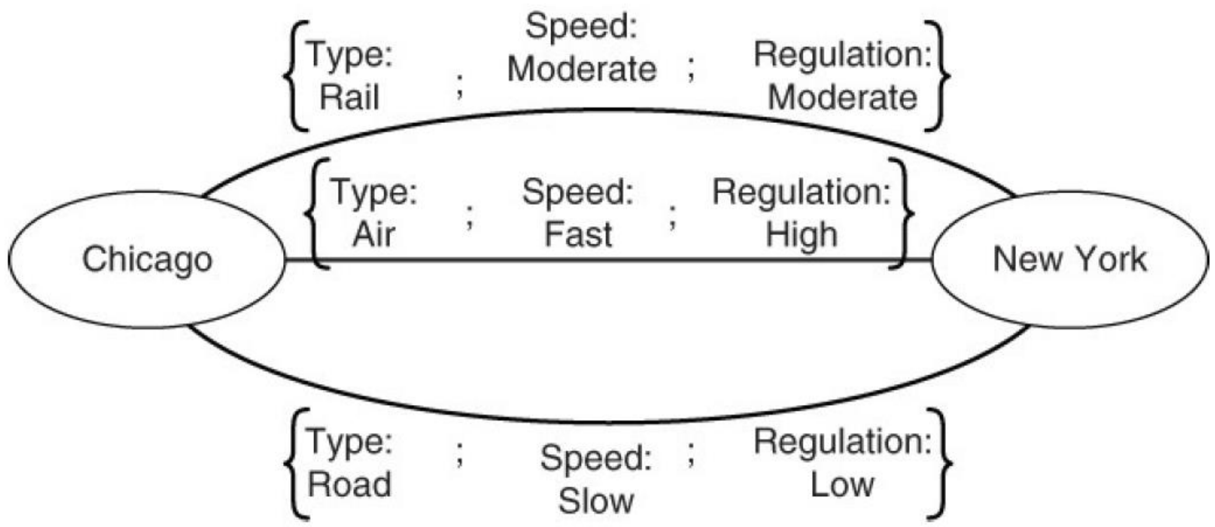

Type: Rail ; Speed: Moderate ; Regulation: Moderate
Type: Air ; Speed: Fast ; Regulation: High
Chicago
New York
Type: Road ; Speed: Slow ; Regulation: Low

Figure. Modeling multiple types of relations in a graph database. Each edge has different properties.

Types of Query with Graph Databases

- **Identifying relations between two entities:**
  - Graph databases are highly efficient in uncovering the connections between two nodes.
  - Examples include finding friends of friends or mutual acquaintances in a social network, or potential pathways in a logistic network.
- **Identifying common properties of edges from a node:**
  - By examining the edges flowing out from a particular node, graph databases can highlight common attributes or patterns.
  - For instance, in recommendation systems, you can identify common interests among user connections or detect common purchasing trends among a subset of customers.

Queries and Analysis with Graph Databases (cont.)

- **Calculating aggregate properties of edges from a node:**
  - Graph databases allow to aggregate information over the edges of a network
  - For example, summing the weights of edges to determine the overall capacity of a network's segment or to gauge the intensity of social interactions based on the frequency of communications represented by edge properties.
- **Calculating aggregate values of properties of nodes:**
  - Graph databases allow for aggregation of node properties to provide a higher-level view of the data.
  - For example, you can calculate the average age of a user’s social circle or total revenue generated from a certain group of products.

Query Performance in Graph Databases

- In graph databases, relationships between are stored as dedicated structures rather than being inferred from key matches.
- This direct storage of relationships allows for rapid traversal across connections, making operations like finding the shortest path between two nodes or extracting subgraphs incredibly efficient.
- There's no need to perform complex joins across tables because the connections are explicit and directly accessible.
- This inherent organization of data leads to faster and more intuitive querying, particularly for operations that are inherently graph-like in nature, such as social network analysis, recommendation engines, and fraud detection.

Graph Query Language

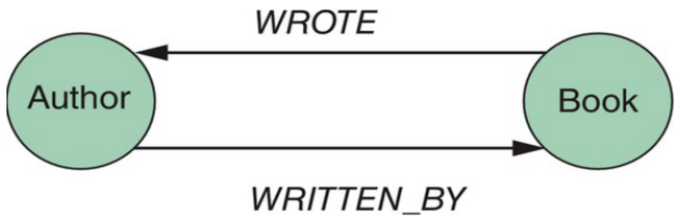
- **Cypher** (Neo4j's query language) is a declarative language similar to SQL.
  - Alternatively, **Gremlin** is a graph traversal language that works with a number of different graph database
- Queries in Cypher are composed of patterns that match the structure of the graph.

Cypher Syntax

- **Node Representation**: Nodes are represented within parentheses, and labels can be assigned using a colon followed by the label name, e.g., `(b:Book)`
- **Node Properties**: Node properties can be specified within parentheses after the node label, e.g., `(b:Book (id:'b01', title:'The Saints’))`
- **Edge Representation**: Relationships between nodes are depicted using either ``` `--` ``` for undirected edges or ``` `->` ``` for directed edges.
- **Filtering Relations**: Relations between nodes can be filtered using square brackets to specify the relationship type, e.g., `(b:Book) <- [ :WRITTEN_BY] – (a:Author)`

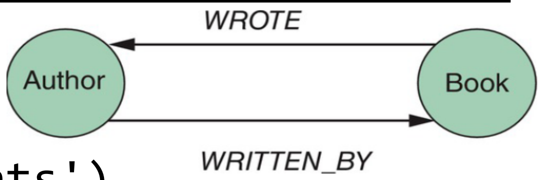
Graph Database Schema

- In a graph database, the **metadata** structure is designed to represent relationships between entities as well as their attributes.



- In the graph database schema, nodes represent entities (books and authors) with labels and properties, while edges represent relationships between nodes.
- Each node and relationship can have attributes associated with it.

Create a Graph using Cypher



- **Create nodes with labels and properties**

```
CREATE(b01:book (id:'b01', title:'The Saints')
CREATE(b02:book (id:'b02', title:'Storm Clouds')
CREATE(a01:author (id:'a01', name:'Donna Everhart')
CREATE(a02:author (id:'a02', name:'Ginny Dye')
```

- **Add edges**

```
CREATE(b01)<-[:WRITTEN_BY]-(a01), (b02)<-[:WRITTEN_BY]-(a02)
CREATE(a01)-[:WROTE]->(b01), (a02)-[:WROTE]->(b02)
```

- A comparable schemas and record insert in relational databases

```
book(id,title)
books_authors(book_id, author_id)
author(id,name)
INSERT INTO book VALUES ('b01','The Saints’);
…
```

Declarative Querying in Cypher

- **To query nodes**, use the `MATCH` command

```
MATCH (b:Book)
RETURN b;
```

  - It matches all nodes labeled as “Book” and returns them.
  - A comparable query in SQL is

```
SELECT b.*
FROM books As b;
```

- In Cypher, you specify the criteria for selecting vertices and edges, but you do not specify how to retrieve them.
- To control over how your query retrieves vertices and edges, you might consider using a graph traversal language such as Gremlin.

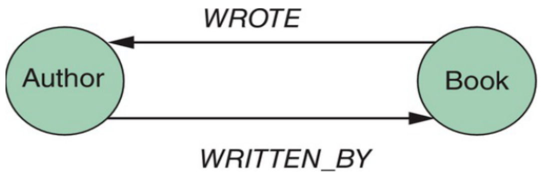
Cypher Query Clauses

- **`MATCH`**
- **`WHERE`**
- **`ORDER BY`**
- **`LIMIT`**
- **`UNION`**
- **`COUNT`**
- **`DISTINCT`**
- **`SUM`**
- **`AVG`**
- etc.

Query in Cypher vs Query in SQL

- For returning all book titles for books written by a particular author

```
MATCH (b:Book) <- [ :WRITTEN_BY] – (a:Author)
WHERE a.name="John Smith"
RETURN b.title
```



  - A comparable query in SQL is

```
SELECT b.title
FROM book b, author a, books_authors ba
WHERE a.id=ba.author_id
AND b.id=ba.book_id
AND a.name='John Smith'
```

Queries in Cypher

- `ORDER BY` clause

  For sorting the results by price in descending order and limiting the output to 20 books:

```
MATCH (b:Book)
RETURN b
ORDER BY b.price DESC
LIMIT 20;
```

- `WHERE` clause

  For filtering the results to only include books with a title "Beginning":

```
MATCH (b:Book)
WHERE b.title = “Beginning”
RETURN b;
```

  Alternatively

```
MATCH (b:Book {title = “Beginning”})
RETURN b;
```

Queries in Cypher

- JOIN

  For listing distinct customer names who purchased a book written by John Smith, are older than 30, and paid in cash

```
MATCH (c:Customer)–[p:PURCHASED]->(b:Book)<-[ :WRITTEN_BY]-(a:Author)
WHERE a.name=“John Smith”
AND   c.age > 30 AND p.type=“cash”
RETURN DISTINCT c.name;
```

Queries in Cypher

- JOIN

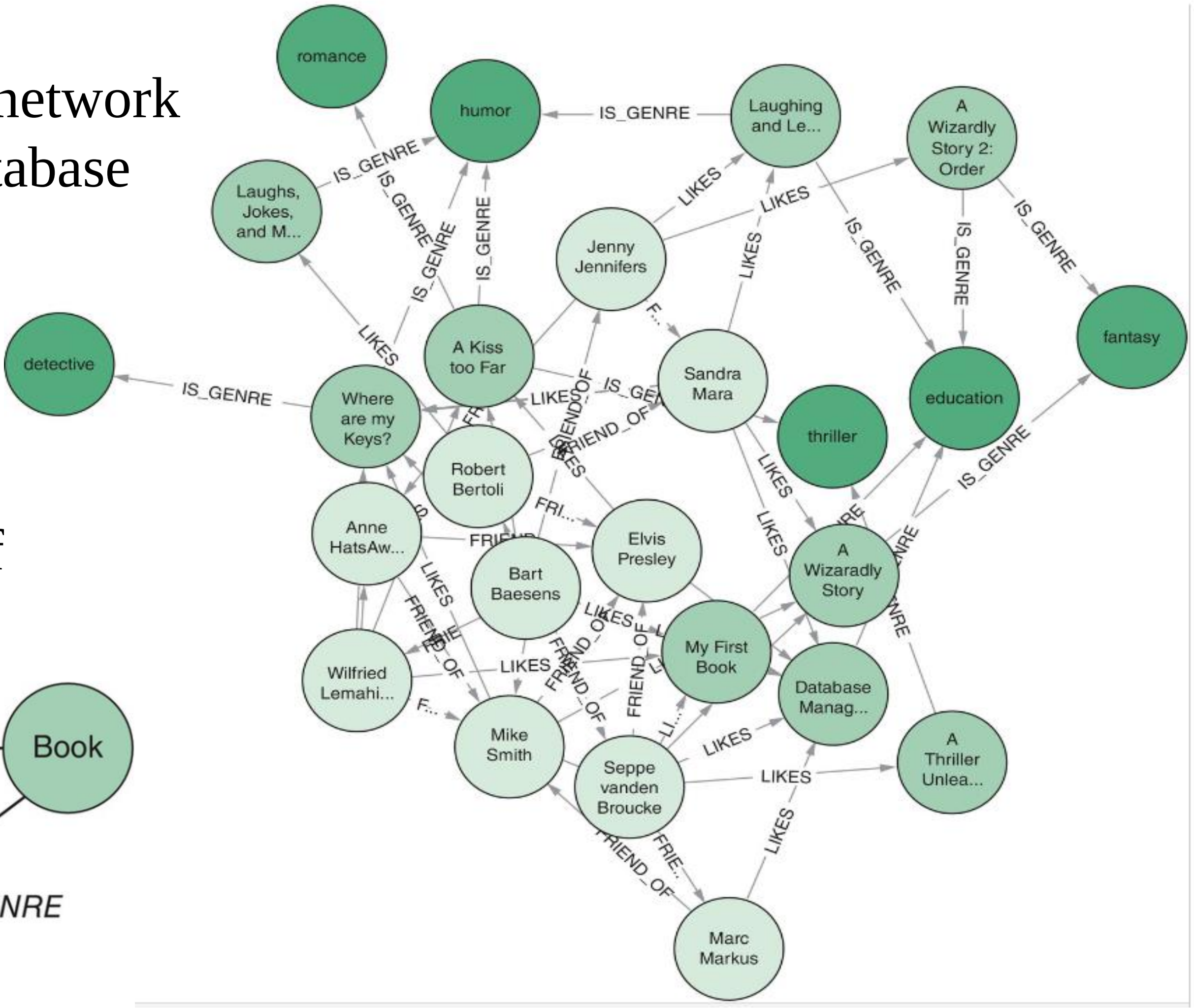
Assume a hierarchy of book genres. For listing all books in the category “Programming” and its subcategories, “SQL”, sub-subcategory, and so on.

```
MATCH (b:Book)–[:IN_GENRE]->(:Genre)
              -[:PARENT*0..]-(:Genre {name:“programming”} )
RETURN b.title;
```

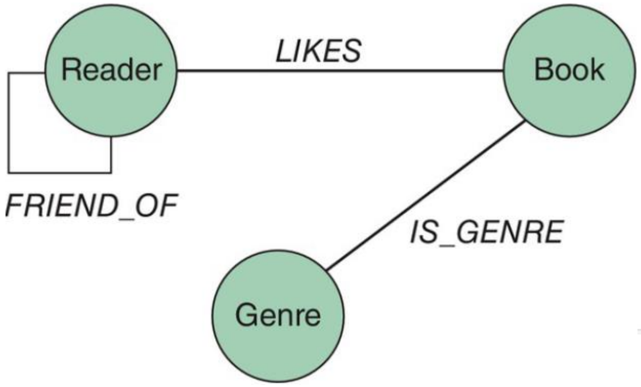
- This matches books (b) that have a :IN_GENRE relationship to a :Genre, and it traverses zero or more :PARENT relationships to reach a :Genre node with the name "programming". The *0.. denotes that it will match zero or more :PARENT relationships, meaning it will get the "Programming" genre itself and any level of subgenres beneath it.
- Cypher can express queries over hierarchies and transitive relationship of any depth simply by appending an asterisk (*) after the relationship type in the MATCH clause and providing optional min and max path lengths by placing limits within brackets following the asterisk

Example: Exploring a Social Graph

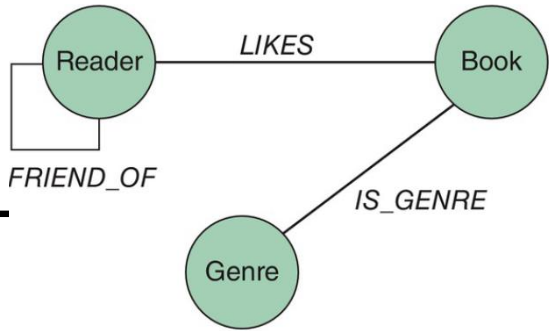
- A social network graph database



- Meta structure of the social graph



Exploring a Social Graph



[NOTE]

- Because there is only one type of relationship between each node type, we can drop the squared brackets colon-selector.
- The usage of `--()--` to perform a non-directional query

- Who likes romance books?

```
MATCH (r:Reader)--(:Book)--(:Genre{name:'romance'})
RETURN r.name;
```

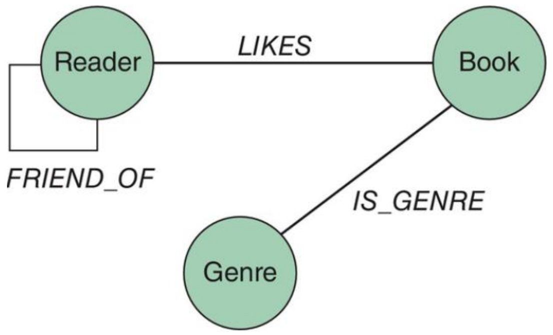
- Who are Bart's friends that liked humor books?

```
MATCH (me:Reader)--(friend:Reader)--(b:book)--(g:Genre)
WHERE g.name='humor'
AND me.name='Bart Baesens'
RETURN DISTINCT friend.name
```

Exploring a Social Graph

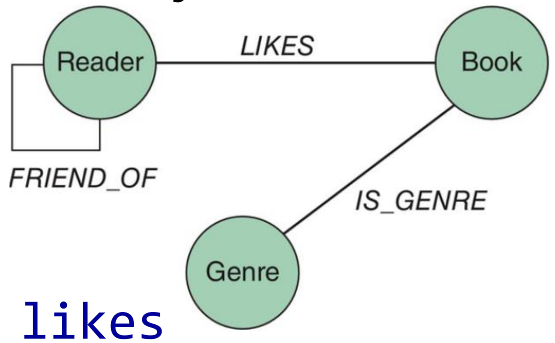
- Can you recommend humor books that Seppe's friends liked and Seppe has not liked yet?

```
MATCH (me:Reader)--(friend:Reader),
      (friend)--(b:Book),
      (b)--(genre:Genre)
WHERE NOT (me)--(b)
AND me.name='Seppe vanden Brouchke'
AND genre.name='humor'
RETURN DISTINCT b.title
```



Exploring a Social Graph

- Get a list of people with books Bart liked, sorted by most liked books in common



```
MATCH (me:Reader)--(b:Book),
      (me)--(friend:Reader)--(b)
WHERE me.name='Bart Baesens'
RETURN friend.name, count(*) AS common_likes
ORDER BY common_likes DESC
```

- List pairs of books having more than one genre in common

```
MATCH (b1:Book)--(g:Genre)--(b2:Book)
WITH b1, b2, count(*) AS common_genres
WHERE common_genres > 1
RETURN b1.title, b2.title, common_genres
```

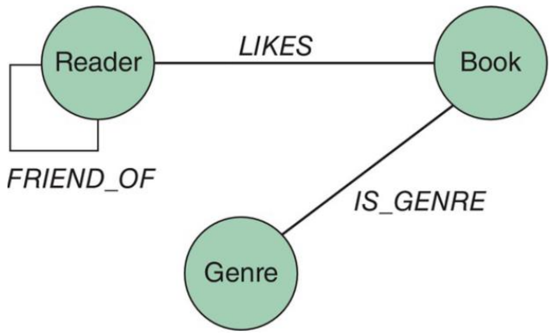
Exploring a Social Graph

- Retrieve pairs of books that have no genres in common

```
MATCH (b1:Book)--(g:Genre)--(b2:Book)
WITH b1, b2, count(*) AS common_genres
WHERE common_genres = 0
RETURN b1.title, b2.title, common_genres
```

Alternatively

```
MATCH (b1:Book), (b2:Book)
WITH b1, b2
OPTIONAL MATCH (b1)--(g:Genre)--(b2)
WHERE g IS NULL
RETURN b1.title, b2.title
```


Reader
LIKES
Book
FRIEND_OF
IS_GENRE
Genre