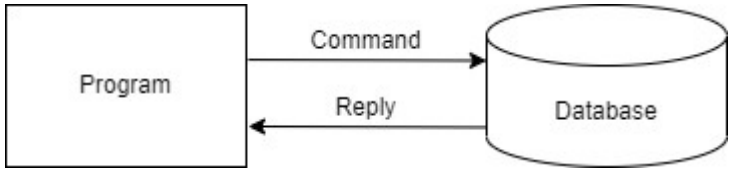
CS4650 Topic 2: Databases

Databases

- A database is a mechanism for managing data.
- New data can be *inserted* into a database.
- Existing data can be *retrieved* from the database. This is traditionally called *select* or *fetch*.
- Existing data can be *modified*. This is traditionally called *update*.
- Existing data can be *deleted* from the database.
- The database is conceptually thought of as separate from the program, a *service* that supports the program.



Relational/Non-Relational

- There are two general categories of databases:
  - Relational (SQL):
    - The data is highly structured, and various elements can be linked through *relations*.
    - Commands are passed to the database server is a well-defined industry standard language, *Structured Query Language* (SQL).
    - This is mature technology, approximately 50 years old.
  - Non-Relational (NoSQL):
    - The data is loosely structured.
    - There are actually several forms, but four main forms, of data storage.
    - Commands are passed through ad-hoc or proprietary languages.
    - This is newer technology, less than 20 years old.

Overview

- Relational Databases
- Non-Relational Databases
- Pros and Cons

Relational Databases

Database Tables

Classes			
Number	*Name*	*Location*	*Time*
CS4650	Big Data	8-04	7pm
CS1300	Discrete Structures	8-301	10am
CS5250	Advanced Computer Arch.	8-345	3pm
CS3800	Computer Networks	8-348	5pm

- A relational database contains a number of *tables*.
- Each table contains one sort of data.
- In this example, we have a table (named '*classes*') that contains information about some classes at school.
- Each table has a number of rows and a number of columns.
- A row provides information about one object, a column gives a particular attribute.

Database Tables

Classes			
Number	*Name*	*Location*	*Time*
CS4650	Big Data	8-04	7pm
CS1300	Discrete Structures	8-301	10am
CS5250	Advanced Computer Arch.	8-345	3pm
CS3800	Computer Networks	8-348	5pm

- For example, the first row gives information about the class CS4650. The second row is for a different class.
- In each row, the first column gives the class number, the second gives its name, and the remaining give the location and the time.

SQL Commands

Classes			
Number	*Name*	*Location*	*Time*
CS4650	Big Data	8-04	7pm
CS1300	Discrete Structures	8-301	10am
CS5250	Advanced Computer Arch.	8-345	3pm
CS3800	Computer Networks	8-348	5pm

- Using the SQL language, a program can ask the database to return, for example, the Name and Location for the CS4650 class.

  SELECT name, location FROM classes WHERE number='CS4650'

  Big Data, 8-04

SQL Commands

- Other very common SQL commands:
  - INSERT INTO classes (number, name, location, time) VALUES ('CS3560', 'Object Oriented', '6-03', '3pm');
    This inserts a new row into the table
  - UPDATE classes SET time = '2pm' WHERE number = 'CS3800';
    This changes one of the rows of the table
  - DELETE FROM classes WHERE number='CS5250';
    This removes one row from the table
- SQL has a lot more commands, and these commands can have many additional values.
- Quite complex operations can be performed in a single SQL command.

Primary Key

Classes			
Number	*Name*	*Location*	*Time*
CS4650	Big Data	8-04	7pm
CS1300	Discrete Structures	8-301	10am
CS5250	Advanced Computer Arch.	8-345	3pm
CS3800	Computer Networks	8-348	5pm

- The RDBMS (relational database management system) needs a way to uniquely identify each row in a table.
- One of the column values must be designated as the *primary key*.
- Each row in the table must have a value for the primary key, and all of these values must be unique, there cannot be duplicates.
- In this table, the Number column is our primary key.

Primary Key

- In some situations, there might not be a natural value that could qualify as the primary key, either because of duplication or because some rows wouldn't have a value.
- In these cases, the database designer would make an additional column that is usually hidden from the end user, but which would have unique integers to be used as the primary key.
- By telling SQL that this column is *primary key autoincrement*, you are instructing the RDBMS to automatically add a unique value for every row you add to the table.

Multiple Tables

Classes			
Number	*Name*	*Location*	*Time*
CS4650	Big Data	8-04	7pm
CS1300	Discrete Structures	8-301	10am
CS5250	Advanced Computer Arch.	8-345	3pm
CS3800	Computer Networks	8-348	5pm

Students	
ID	*Name*
137	Bob
642	Sue
958	Jim

Intersection		
Key	*Student*	*Class*
1	137	CS4650
2	137	CS5250
3	642	CS1300
4	642	CS4650
5	642	CS3800
6	958	CS4650

- A database usually has multiple tables. Here we see three tables:
  - Our list of classes,
  - A list of students (each of which has a student ID and a name)
  - A table which links students and classes. This table has an autoincrement primary key, the student ID number, and the class number. Each row indicates that the associated student is taking the indicated class.

Multiple Tables

Classes			
Number	*Name*	*Location*	*Time*
CS4650	Big Data	8-04	7pm
CS1300	Discrete Structures	8-301	10am
CS5250	Advanced Computer Arch.	8-345	3pm
CS3800	Computer Networks	8-348	5pm

Students	
ID	*Name*
137	Bob
642	Sue
958	Jim

Intersection		
Key	*Student*	*Class*
1	137	CS4650
2	137	CS5250
3	642	CS1300
4	642	CS4650
5	642	CS3800
6	958	CS4650

- The SQL SELECT command has options to JOIN data between tables.
- For example, you can ask for a list of all of the classes that Bob is taking:
  - Look for the ID of Bob,
  - Find all of the rows in Intersection that have this ID.
  - Print out the class information based on the class numbers from the Intersection table.

Multiple Tables

Classes			
Number	*Name*	*Location*	*Time*
CS4650	Big Data	8-04	7pm
CS1300	Discrete Structures	8-301	10am
CS5250	Advanced Computer Arch.	8-345	3pm
CS3800	Computer Networks	8-348	5pm

Students	
ID	*Name*
137	Bob
642	Sue
958	Jim

Intersection		
Key	*Student*	*Class*
1	137	CS4650
2	137	CS5250
3	642	CS1300
4	642	CS4650
5	642	CS3800
6	958	CS4650

- SQL can also group data, sort data, compute values.
- Many of the computations we will be discussing in this class can be performed in SQL.
- Skilled SQL programmers can make amazing SQL queries!

Comparing Databases to Object Oriented Programming

- A relational database is much like object oriented programming.
- In programming, you define *classes*. Each class has a number of attributes, and usually these attributes have specific datatypes.
- In databases, you define *tables*. Each table has a number of columns, and these columns may have specific datatypes.
- A database table has many similarities to a class.

Comparing Databases to Object Oriented Programming

- In object oriented programming, you create one or more instances of a class. Each of these instances is one 'object'. An instance has specific values for each of the attributes of the class.
- In databases, you have one or more rows in a table. A row has specific values for each of the columns of the table.
- The rows of a database table are quite similar to the instances of a class.

Comparing Databases to Object Oriented Programming

- As a typical program typically has multiple classes, a database typically has multiple tables.
- The classes are structured, and the tables are structured. *Meaning* is assigned to the various attributes or cells.
- Objects in OOP can *refer* to other objects, one of the attributes can be a reference to another object.
- In a relational database, one row from one table can be a reference to a row in another table. In our example above, the Intersection table had a reference to a student (the cell contained the Student ID, the primary key of the student table), and a reference to a class (the class primary key: Number).

Comparing Databases to Object Oriented Programming

- Many of the design principles we learned in object oriented programming apply to how we design tables in an RDBMS:
  - One class does one thing. One table holds one set of objects.
  - Don't duplicate data (dry - don't repeat yourself).
  - Avoid errors by using strongly typed variables -- and columns!

Database Normalization

- One way to optimize a relational database is through *normalization*.
- Normalization reduces the amount of information stored, while at the same time making it more searchable, usable, changeable, and reliable.
- Many of these same steps are performed when designing code!
- The main idea of normalization is that each table should be about a specific topic and only include supporting topics.
- The three main reasons to normalize a database are:
  a. To minimize duplicate data
  b. To minimize or avoid data modification issues
  c. To simplify queries
- First let's consider a database that is *not* normalized

Non-Normalized Database

Students			
Name	*Class1*	*Class2*	*Class3*
Bob	CS4650;BigData;8-04;7pm	CS5250;Advanced Computer Arch;8-345;3pm	
Sue	CS1300;Discrete Structures;8=8-301;10am	CS4650;BigData;8-04;7pm	CS3800;Computer Networks;8-348;5pm
Jim	CS4650;BigData;8-04;7pm		

- This version of the database lists our three students.
- The information for each class is given as a text string, with the various fields separated using semicolons.
- In many cases, beginner database designers write tables like this!
- This database is *not* normalized!

Normalization Process

- As we step through the normalization process, we will break complex tables into multiple simpler, more focused tables.
- There are three primary *normal forms*:
  - The *first normal form*,
  - The *second normal form*, and
  - The *third normal form*
- There are actually a half dozen additional forms, but those are more specialized and beyond the scope of our discussion!

Non-Normalized Database

Students			
Name	*Class1*	*Class2*	*Class3*
Bob	CS4650;BigData;8-04;7pm	CS5250;Advanced Computer Arch;8-345;3pm	
Sue	CS1300;Discrete Structures;8=8-301;10am	CS4650;BigData;8-04;7pm	CS3800;Computer Networks;8-348;5pm
Jim	CS4650;BigData;8-04;7pm		

- The first thing to notice is that this table contains a lot of disparate information:
  - The names of the students
  - Their list of classes
  - Information about each of the individual classes.
- Note that duplicating the information about the classes wastes a lot of space.
- It is easy to have typos in all of the replicated data.

Non-Normalized Database

Students			
Name	*Class1*	*Class2*	*Class3*
Bob	CS4650;BigData;8-04;7pm	CS5250;Advanced Computer Arch;8-345;3pm	
Sue	CS1300;Discrete Structures;8=8-301;10am	CS4650;BigData;8-04;7pm	CS3800;Computer Networks;8-348;5pm
Jim	CS4650;BigData;8-04;7pm		

- Suppose we want to change the room for the CS4650 class.
  - We have to find all of the places where CS4650 is referenced.
  - We have to extract the old location from each string, then insert the new location at that spot.
  - If we miss any of the spots, we then have a *update anomaly*.

Non-Normalized Database

Students			
Name	*Class1*	*Class2*	*Class3*
Bob	CS4650;BigData;8-04;7pm	CS5250;Advanced Computer Arch;8-345;3pm	
Sue	CS1300;Discrete Structures;8=8-301;10am	CS4650;BigData;8-04;7pm	CS3800;Computer Networks;8-348;5pm
Jim	CS4650;BigData;8-04;7pm		

- Suppose we want to add a new class.
- We can't do that until we have at least one student taking the class, as there is no place to insert the value without creating a row for a student.
- This is known as an *insert anomaly*.

Non-Normalized Database

Students			
Name	*Class1*	*Class2*	*Class3*
Bob	CS4650;BigData;8-04;7pm	CS5250;Advanced Computer Arch;8-345;3pm	
Sue	CS1300;Discrete Structures;8=8-301;10am	CS4650;BigData;8-04;7pm	CS3800;Computer Networks;8-348;5pm
Jim	CS4650;BigData;8-04;7pm		

- Suppose Bob leaves the school, so his row is eliminated from the table.
- This will remove the last reference to the CS5250 class, so it will disappear.
- This is called a *deletion anomaly*.

Non-Normalized Database

Students			
Name	*Class1*	*Class2*	*Class3*
Bob	CS4650;BigData;8-04;7pm	CS5250;Advanced Computer Arch;8-345;3pm	
Sue	CS1300;Discrete Structures;8=8-301;10am	CS4650;BigData;8-04;7pm	CS3800;Computer Networks;8-348;5pm
Jim	CS4650;BigData;8-04;7pm		

- This database would also be hard to search.
- For one thing, we would have to deconstruct the class strings to search for specific pieces of the string.
- Also consider that if we are looking for instances of CS4650, we have to check all three class columns, it could be in any of these.
- Also note that this form limits the number of classes a student can take to 3.

First Normal Form

- For a database table to satisfy the first normal form, it must have the following:
  - There must be a primary key that uniquely identifies each row.
    - In our table, there is no 'student ID number', and two students could have the same name, so there is no unique identity value.
  - There must be no repeated columns.
    - In our table, 'class1', 'class2', and 'class3' are basically a repeated column, or an array of columns.
  - Each column should have an atomic value.
    - In our table, the cell holding information about a class has the number, name, location, and time values as one text string, separated by commas.

First Normal Form

Students	
ID	*Name*
137	Bob
642	Sue
958	Jim

Classes	
ID	*Class*
137	CS4650;BigData;8-04;7pm
137	CS5250;Advanced Computer Arch;8-345;3pm
642	CS1300;Discrete Structures;8=8-301;10am
642	CS4650;BigData;8-04;7pm
642	CS3800;Computer Networks;8-348;5pm
958	CS4650;BigData;8-04;7pm

- Here are the results of our first step towards *first normal form*. We aren't there yet, but we are closer.
  - We added a primary key to the Students table. We actually added the Student ID number, which makes a good primary key: Every student has one, and there are no duplicates.
  - We've split off the class information to a separate table. This allowed us to remove the replicated columns.
  - The Student ID is the primary key in the Students table, but it is a *foreign key* in the Classes table. A foreign key does not uniquely identify a row, but it does link the two tables together.

First Normal Form

Students	
ID	*Name*
137	Bob
642	Sue
958	Jim

Classes	
ID	*Class*
137	CS4650;BigData;8-04;7pm
137	CS5250;Advanced Computer Arch;8-345;3pm
642	CS1300;Discrete Structures;8=8-301;10am
642	CS4650;BigData;8-04;7pm
642	CS3800;Computer Networks;8-348;5pm
958	CS4650;BigData;8-04;7pm

- The Students table is now in First Normal Form: There is a primary key, there are no repeated columns, and all of the values are atomic.
- The Classes table is not yet in First Normal Form: There are no repeated columns, but there is no primary key, and the values in the Class column are not atomic.
- Let's fix this table.

First Normal Form

Students	
ID	*Name*
137	Bob
642	Sue
958	Jim

Classes					
Key	*ID*	*Number*	*Name*	*Location*	*Time*
1	137	CS4650	Big Data	8-04	7pm
2	137	CS5250	Advanced Computer Arch	8-345	3pm
3	642	CS1300	Discrete Structures	8-301	10am
4	642	CS4650	Big Data	8-04	7pm
5	642	CS3800	Computer Networks	8-348	5pm
6	958	CS4650	Big Data	8-04	7pm

- Here we have added a new column, an autoincrement primary key called *key*.
- We have also split the long text strings into multiple columns, one for each value.
- Both tables are in First Normal Form.
- However, there is still a lot of waste in the Classes table. There is a lot of duplicated data.

Second Normal Form

- For a database table to satisfy the second normal form, it must have the following:
  - The table must be in the first normal form, and
  - All of the non-key columns are dependent upon the table's primary key.
- What does that mean?
- The primary key identifies a particular row in the table, and that row represents some 'object'.
- Do each of the other values describe that same object?

Second Normal Form

- Consider the class table, specifically rows 1, 4, and 6.
- These rows are all describing the CS4650 class.
- The ID field is given the Student ID of one of the students taking this class. So that is not really describing this class, it is just presenting a relationship about this class.

Classes					
Key	*ID*	*Number*	*Name*	*Location*	*Time*
1	137	CS4650	Big Data	8-04	7pm
2	137	CS5250	Advanced Computer Arch	8-345	3pm
3	642	CS1300	Discrete Structures	8-301	10am
4	642	CS4650	Big Data	8-04	7pm
5	642	CS3800	Computer Networks	8-348	5pm
6	958	CS4650	Big Data	8-04	7pm

Second Normal Form

- By removing the ID column from the upper table, and by combining identical rows, we get the second table.
- We no longer need the Key column to be the primary key for the table, we can instead use the Class Number as the primary key.
- This table is now in second normal form.
- But what about the links showing which student took which class?

Classes					
Key	*ID*	*Number*	*Name*	*Location*	*Time*
1	137	CS4650	Big Data	8-04	7pm
2	137	CS5250	Advanced Computer Arch	8-345	3pm
3	642	CS1300	Discrete Structures	8-301	10am
4	642	CS4650	Big Data	8-04	7pm
5	642	CS3800	Computer Networks	8-348	5pm
6	958	CS4650	Big Data	8-04	7pm

Classes			
Number	*Name*	*Location*	*Time*
CS4650	Big Data	8-04	7pm
CS1300	Discrete Structures	8-301	10am
CS5250	Advanced Computer Arch.	8-345	3pm
CS3800	Computer Networks	8-348	5pm

Second Normal Form

- We add the following table, *Intersection*, to link between classes and students.
- We need a primary key, so we will use an *autoincrement* column for that.
- The two remaining columns give the key from the Student table and the key from the Class table.

Intersection		
Key	*Student*	*Class*
1	137	CS4650
2	137	CS5250
3	642	CS1300
4	642	CS4650
5	642	CS3800
6	958	CS4650

Second Normal Form

- This table provides bidirectional linkage between the tables.
- Given a student ID, we can find all of the classes that this student is taking:

  SELECT Class FROM Intersection WHERE Student=137;
- Given a Class Name, we can find all of the students that are taking this class:

  SELECT Student FROM Intersection
  WHERE Class like 'CS4650';

Intersection		
Key	*Student*	*Class*
1	137	CS4650
2	137	CS5250
3	642	CS1300
4	642	CS4650
5	642	CS3800
6	958	CS4650

Second Normal Form Benefits

- Moving to the second normal form mostly removes redundancy from the database.
- This also removed the possibilities of update, insert, or delete anomalies.
- Searching is much faster, as the data is more accessible.

Third Normal Form

- For a database table to satisfy the third normal form, it must have the following:
  - The table is in second normal form.
  - No column of the table *transitively depends* upon the primary key through any other column of the table.
- ?
- Examples:
  - A hospital database table has a patient's name, doctor's ID, and doctor's phone number. *Instead, create a new table Doctors, which has the doctor's ID and phone number. The phone number can be removed from this table.*
  - A car database table has VehicleID, Model, and Manufacturer. *A separate table can list the Manufacturer for each Model.*

Non-Relational Databases

What is NoSQL?

- NoSQL refers to "not-only-SQL", or perhaps better, "non-SQL".
- These manage databases in a non-relational form.
  - Data is not structured in a tabular manner,
  - There are no relationships between tables.
- When planning a NoSQL system, the focus is on collecting the data rather than how will the data be used.
- Consequently, NoSQL databases are more flexible.
- NoSQL can handle a greater variety of data types.
- NoSQL can support distributed databases.
- NoSQL databases can be more scalable.

Types of NoSQL Databases

There are in general four types of NoSQL databases:

1. Column: This is similar to having just a single table in the database. They can have many columns, and the data can be sparse (rows might have data in only a few of the columns).
2. Document: Stores and keeps semi-structured data along with its description. In many cases, the 'document' is a JSON object.
3. Key-Value: The data is stored as an associated array (i.e. dictionary).
4. Graph: Data is organized as nodes and edges.

Types of NoSQL Databases

- In this class, we will mainly use Column oriented NoSQL databases.
- For this discussion, the examples will discuss a Document oriented NoSQL database holding JSON data.
- The following slides use examples from the MongoDB documentation.

MongoDB Basics

- MongoDB is a Document-based NoSQL database.
- The following hierarchy of elements exist:
  - The top level is a *cluster*, which can contain multiple databases
  - A *database* may contain multiple collections. A collection is roughly equivalent to a table in SQL.
  - A *collection* may contain multiple documents. A document is one 'object' or 'element' stored in the database.
  - So a collection contains a list of objects.
    - New objects (documents) can be added to a collection.
    - An object can be deleted from a collection.
    - A collection can be searched to find one or more documents that match the search.

MongoDB Document

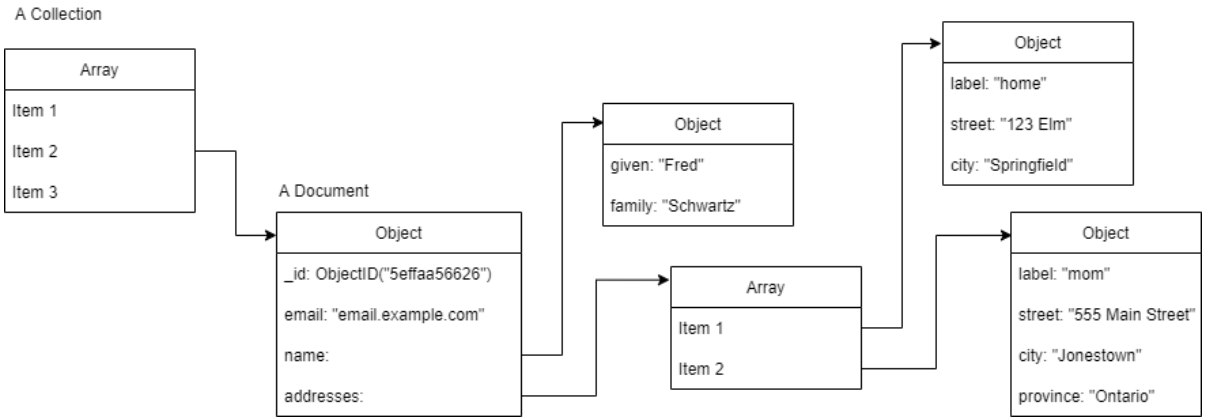
- A MongoDB Document is basically a JSON object. It is actually a BSON object, which is a Binary JSON object (so more efficient that textual JSON).
- A BSON element can be any one of the following:
  - String
  - Boolean
  - Integer (various sizes)
  - Float (various sizes)
  - Array (of BSON elements)
  - Date
  - Object (a dictionary of key-value pairs where the keys are strings and the values are any BSON element).

Sample Document

```
{ _id: ObjectID("5effaa56626"),
  email: "email@example.com",
  name: {given: "Fred",
         family: "Schwartz"},
  addresses: [
       { label: "home",
         street: "123 Elm",
         city: "Springfield"},
       { label: "mom",
         street: "555 Main Street",
         city: "Jonestown",
         province: "Ontario"}]
}
```

- The document is an object (list of key-value pairs)
- The value for name is a sub-object, with its own key-value pairs.
- The value for addresses is an array with two values, each of which is an object.
- Note that the two address objects have different sets of key-value pairs.
- New key-value pairs can be added to any object -- there is no "definition" that says what an object consists of.

Sample Document



- Our sample document. It is hierarchical, but with arbitrary, flexible values.

MongoDB Shell

- MongoDB has an API, through which programs can use the database.
- A Shell tool is provided to use the database from the command line.
- Typically, the first command when using the shell is to select which database to use:

```
> use my_database;
```

- From this point on, the variable '`db`' references this database.
- This command shows the collections in this database:

```
> show collections;
```

MongoDB Shell

- Suppose the document from above is in the collection 'users'.
- We reference this collection using 'db.users'.
- For example, to see the number of documents in the collection, type:

```
> db.users.count()
20234
>
```

- We can find the first document in the collection:

```
> db.users.findOne()
{
"_id": ObjectID("5effaa56626"),
"email": "email@example.com"

…
```

Finding a Document

- When performing a search (in SQL: SELECT), the query is given as the *pattern* of what is being sought.
- To find all documents that have a particular email address, the following query will suffice:

```
> db.users.findOne({email: "email@example.com"})
…
```

- The pattern is simply a description of what is being sought in the document, in this case a object that has an email field with a specific value.

Finding a Document

- The patterns can nest:

```
> db.users.find({"name.family": "Schwartz"}).count()
```

- Here we are looking for a document that is an object that has a 'name' element, and that element is an object which has a 'family' element, and that element is the string "Schwartz".
- This also shows that find can return multiple documents, as many as match. Here we then ask for a count of the number that matched.

Additional Options

- There are many additional commands in the MongoDB shell.
- For integer or float values, the query can express a range of values that would match.
- The results of a query can be sorted.
- If collections get very large, queries may take considerable time. To speed the process, MongoDB can create an Index.
    - This adds additional data to the database
    - This dramatically speeds up a query
    - Multiple indexes can be created, one for each key field of the data.

Examples of Graph-Based NoSQL Databases

- A Graph Database represents data as *nodes*.
- Nodes can have properties to provide further information.
- Nodes are connected to each other by *edges*.
- Edges can be labeled with properties.
- These are flexible and expandable, because new type of nodes can be added, new types of edges, new properties, and so on.

Examples of Graph-Based NoSQL Databases

- A recommendation engine might use a Graph database. You can have nodes for customers, interests (such as sports), purchase items. Edges can show friends, ratings (of products), following (of sports). Recommendations can be made by finding other customers with similar purchase histories and interests.
- Social networks use Graph databases to store who is friends with whom, how often a person likes or dislikes another person's posts. Friends of friends can be found to give friendship recommendations.
- Fraud detection can store a person's financial and purchase transactions, then examine new transactions to see if they match the person's pattern.

Pros and Cons

Which is Best?

- The type of database to use depends upon the application, data, and needs.
- As in almost all areas of computer science, it is silly to think that there is one approach that is always "best". It's like saying the best tool for a carpenter is a hammer.
- As a computer scientist, you pick the approach that works best within the constraints of the problem you are trying to solve.
- The following list summarizes some of the decisions that can indicate which approach is best for a particular application.

SQL Pros

- Standardized schema. All of the data added to the database must comply with the well-known schema of linked tables made up of rows and columns. This is especially significant when data consistency, integrity, and security are important. Another way to look at this: with SQL, the data is verified before it goes into the database; with NoSQL the data must be verified each time you take it out!
- Large User Community. The SQL language is very mature and widely used. It has a strong community, countless experts, and since there is stability, what you find on the web is usually accurate and up-to-date.

SQL Pros

- No code required. SQL is a user-friendly language, honed by over 50 years of use. Managing and querying the database is done through simple keywords.
- ACID compliance. This means the data is in-sync and the transactions are valid. If a high-level of data integrity with no room for error is required, this is the choice.
    - Atomicity: All transactions are completed as a single operation, either all or nothing. So a transaction cannot 'partially complete', leaving the database in an invalid state.
    - Consistency: The data is valid at both the start and the end of a transaction.
    - Isolation: Transactions can run concurrently.
    - Durability: When a transaction completes, the changes to the database are complete.

SQL Pros

- Complex queries: SQL is a good fit for complex queries. SQL has a standard interface for complex queries.
- Security: SQL implementation allow for strict security measures, *confidentiality, integrity, availability*. Only authorized users or systems can access the data, and some actors may have limited permissions (such as read-only). Different tables can have different access rights.

SQL Cons

- Hardware: If a SQL installation has reached its capacity, to scale-up vertically, meaning it requires a more powerful computer with more extensive memory, which is costly.
- Normalization: This is actually both a pro and a con. With normalization, the database is more efficient, easier to search, and takes less memory. Updates are also greatly simplified. However, when searching, complex joins can slow things down.
- Rigidity: The database schema is 'baked into' the database. Changing the schema *can* be done, but updates can require some effort.
- Distributed databases are difficult: There is a practical limit to the size of a database.

SQL Cons

- Hierarchical data: SQL databases are not well suited for hierarchical data. Instead, the hierarchy must be 'exploded', implemented as multiple tables and linkages. These structures have to be 'reimplemented' when the data is fetched from the database (through multiple queries).

NoSQL Pros

- Continuous availability: Since a NoSQL database can be distributed, and data duplicated on multiple nodes, the system is more resilient: If one machine goes down, the whole database is still available through the other machines in the cluster.
- Query speed: For simple queries, both SQL and NoSQL are very performant. For a query that would require complex joins in SQL but happen to be stored as a single 'document' is NoSQL, NoSQL would be faster.

NoSQL Pros

- Agility: NoSQL databases are designed for flexibility, so new data can be quickly added to the system. If new requirements arise, they can be quick to be implemented.
    - On the other hand, if 'documents' are added with data arranged in a different manner, then any code accessing the database now has to support reading *both* versions of document (and be able to distinguish between them).
- Customizable: With the four primary types of NoSQL database, the developer has the flexibility to choose the type of database that best fits the problem. (Of course, this also includes the possibility of selecting a SQL database!)
- Low Cost: NoSQL databases scale horizontally. If the dataset gets larger, it can be spread onto more machines in the cluster.

NoSQL Pros

- Hierarchical data: Depending upon the form of the NoSQL database, hierarchical data is supported. A complex, hierarchical object can be written to the database or read from the database as a unit.

NoSQL Cons

- No standardized language: The language used to communicate with the database varies depending upon the type of database, and also varies from vendor to vendor. Being much newer, the languages are not as well defined and refined. And since they change more frequently, what you find on the web may be out-of-date.
- Smaller user community: Fewer people are familiar with these systems, and there are many fewer experts. Solving a difficult problem may be a lot more challenging.
- Inefficiency with complex queries: Since the database is so flexible about the data, the system can know less about the data, so complex operations have to be coded by the programmer.

NoSQL Cons

- Data retrieval inconsistency: Because the data is distributed, it is difficult to maintain consistency of the data. The same query at the same time from different sources might receive different replies. Rather than the ACID-level of compliance of SQL, NoSQL uses BASE compliance:
  - Basic Availability: The database appears to work most of the time.
  - Soft-state: Stores don't have to be write-consistent; different replicas do not have to be mutually consistent at all times.
  - Eventual consistency: Stores exhibit consistency at some later point.
- The queries in NoSQL are less capable than SQL queries, so more work must be done by the programmer.
- Simultaneous queries may interfere with each other.

NoSQL Cons

- Security: NoSQL databases have very few security features, in order to allow for faster access. NoSQL databases are an easier target for security attacks.

That Being Said…

- SQL and NoSQL databases each have their place, both are valuable tools.
- Since this class is about big data and distributed computing, we will be using NoSQL.
- Next up: HDFS, the Hadoop Distributed File System.