Kevin Chen

1/30/17

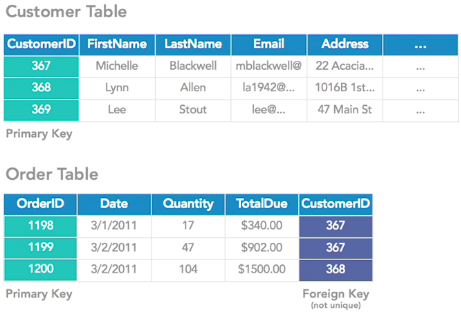
Databases

Understanding Databases

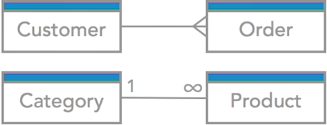
* What are databases?
  + Data storage. But just having data is not good enough of a reason to have a database; using a spreadsheet might be enough.
  + But potential problems that would require you to use a database: size (as you have too many entries, speed might be reduced), ease of updating (can multiple people edit at the same time), accuracy (things that would prevent me from inputting incorrect data), security (some of the data is sensitive, can we keep track of who edited each piece of data?), redundancy (multiple copies – good for backups, but not for within the data itself since this creates contradictions), importance (losing important data is bad)
  + Solutions: scalable, accessible, accurate, secure (who can access/edit it, tracking who did what), consistent, permanent (never lose data)
  + Databases give us structure.
* Exploring databases and database management systems
  + These aren’t databases: Oracle, SQL Server, MySQL, PostgreSQL, MongoDB. These are instead database management systems (DBMS).
  + You install the DBMS software and then you use this software to manage databases. The database is your data and your rules about that data.
  + In practice, often have multiple databases and multiple DBMS.
  + DBMS fall into broad categories. Most common by far is the relational database management system, or RDBMS (e.g. Oracle, SQL Server, DB2, MySQL). Other DBMS: hierarchical DBMS, Network DBMS, Object-Oriented DBMS, NoSQL DBMS (e.g. Cassandra, CouchDB, MongoDB)
  + Relational DBMS: are the most widely used, use the same principles across all offerings, and are foundational for understanding other systems

Database Fundamentals

* The features of a relational database
  + A database is constructed of tables. All data goes into a table.
  + Each table describes a formalized, repeating list of data. Visually shown like a spreadsheet, consisting of columns and rows.
  + Each table contains repeating information about the same kind of thing.
  + Each row represents one entry.
  + Must apply structure to each row by defining the columns within the table. Each column must not only have a name but also a type (e.g. text, date, numeric). Each row entry must have the same format defined by the column. By defining columns, we’re imposing rules on the data they store.
  + Tables and columns are defined up front. Day-to-day use is in creating and updating rules.
* Exploring unique values and primary keys
  + A key identifies one particular row in a table.
  + Typically created as one column that will contain a guaranteed unique value for each row. Most columns should not be unique, but ID’s are often unique.
  + Much of the time there isn’t a piece of data that is naturally unique, so you can tell the database to generate a new, unique column. The unique column is known as the primary key for the table. You should a primary key create one even if you have SSID because you don’t want primary keys to contain sensitive data.
  + A generated column solely for the purpose of having primary keys is also sometimes called synthetic keys or surrogate keys.
* Defining table relationships
  + Define relationships based on keys.
  + If we have a one-to-many relationship between the rows of Table A with the rows of Table B, we could copy the rows of Table A to the corresponding rows of the Table B, but that causes duplication in data and is simply unnecessary.
  + Instead, use the primary key column from Table A (which uniquely identifies each row) and add this column to Table B, matching each value within the primary key column with the row in Table B. Example:



* + As shown above, the primary key from another table is the foreign key of the compared table.
  + One-to-many relationship (very frequently occurring): one entry in Table A corresponds to multiple entries within Table B (thought it doesn’t have to be multiple; can be 1 or 0). Notation:



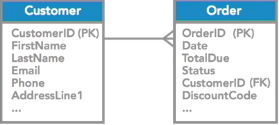
* Describing many-to-many relationships (occasionally occurring)
  + A row from Table A corresponds to multiple rows of Table B, and a row from Table B corresponds to multiple rows of Table A. These relationships are trickier and can’t be done directly.
  + One technique is to add more columns to store multiple rows that a particular row corresponds to, but this is referred to repeating groups, or repeating columns, and this is discouraged.
  + Another discouraged technique is to squeeze in more values into the foreign key column.
  + The correct solution: add another table to describe the relationship between the two tables. This table is called the junction or linking table. By convention, it is called *TableANameTableBName*. (So if you are linking Class and Student, this table would be called ClassStudent.) List each pair of corresponding rows within the table.
  + One-to-one relationship (rarely occurring): just merge the tables together
* Transactions and the ACID Test
  + Transaction: a combined unit of work; either both of the actions occur or neither of them occur.
  + Example: banking transaction: move $2000 from savings to checking. Need to remove $2000 from savings and add $2000 to checking. If you successfully removed $2000 from savings but had some error when trying to add $2000 to checking, you need to add $2000 back to savings. Only okay if both actions or neither action occurs.
  + ACID: A transition must be atomic (transaction must be completely done or not done at all), consistent (databases must remain in valid states before and after the transaction), isolated (data and transaction are locked while the transaction is occurring; can’t have another part of the system to access the data), and durable (transaction is robust).
  + Usually these capabilities are built into the system. Just tell the database that there is a transition involving *n* number of things.
* Introduction to Structured Query Language (SQL)
  + Common language that lies at the heart of RDBMS.
  + It is a declarative query language, not a procedural, imperative language. This means that you use SQL to describe what you want, and you let the database management system handle how that’s actually done. Just ask SQL what outcome you want; it handles all the steps.
  + SQL can do CRUD: create, read, update, and delete data.
  + Each database management system has a slightly different implementation of the core SQL language, but are fundamentally similar.

Database Modeling: Tables

* Introduction to database modeling
  + The database schema consists of the tables, columns, primary keys, and relationships
  + Planning is very important; agile development is not suitable. Don’t want to change rules often; changing them can be painful.
* Planning Your Database
  + Ask: what’s the point? What’s the intention? Need full description of all the functionality of the database and application.
  + Ask: what do you already have? Physical items? People and expertise? Existing “database” (could be primitive “databases” like spreadsheets)? Problems with existing “database”?
  + Ask: What entities do you have? List things that exist in the real world (e.g. customers, orders, products) and more abstract things (e.g. comments, appointments).
  + Unlike OOP, we are concerned only about our entities; the data, and their relationships. No focus on things like methods/behavior. Each entity becomes a table.
  + Entity-relationship (ER) modeling: modeling the relationships between entities
* Identifying columns and selecting data types
  + For each entity, list out the attributes you need to store. These attributes form the columns of our table.
  + When defining columns, you must be very granular, or as individual as possible. Example: instead of one column for name, have one column for first name and one for last name. It is easier to work with each individual piece of data when they are stored separately.
  + Next, specify the data type of each column (e.g. character, date). You need to be very specific here. Example: there are many types of integers. Example 2: fix the length of a character (sequence). Fixed lengths are more efficient but less flexible. By default, you probably should use Unicode.
  + Storing binary data: useful for media. BLOBS = binary large objects (large binary files). CLOBS = character large objects (large amounts of text).
  + Define whether certain columns are optional for each entry. (Allow null values?)
  + Define whether certain columns have default values. Constraints? (Maximum/minimum values, pattern matching)
* Choosing primary keys
  + Each table should have a primary key.
  + Sometimes the primary key naturally occurs within the data (e.g. ISBN). These are called natural keys. But when it doesn’t naturally occur, create an ID column as the primary key. Usually you do this by defining the column as an integer, add the option to make it automatically increment, and specify this as the primary key, or identifier.
* Using composite keys
  + One option for a primary key is a composite key. This is when two values (two columns) combine to create a unique primary key.
  + Example: table of yearbooks from multiple schools from multiple years. But if each school only has one yearbook for each year, then the composite key of school and year can be used as the primary key.
  + Even then, it might be more useful or convenient to generate a surrogate primary key column anyway. But composite keys can be useful.
  + Useful for joining tables together to create many-to-many relationships.

Database Modeling: Relationships

* Creating relationships
  + Three types of relationships: one-to-one-, one-to-many, and many-to-many
  + Natural order of database design: create entities (become tables), then define the attributes for those entities (becomes columns), select/generate primary keys.
  + But this process isn’t necessarily linear. In fact, the process of normalization may cause you to end up with a few more tables.
  + Benefit of specifying relationships formally is that the DBMS will keep the relationships valid and meaningful.
* Defining one-to-many relationships
  + Most common type of cardinality (relationship).
  + The rows in the table representing the “many” each needs to correspond to exactly one row in the table representing the “one”
  + You need to change the table representing the “many” in this one-to-many relationship by adding the foreign key. (Refer to above section of “Defining Table Relationships”.)
  + Often use the same column name between the foreign key of the “many” table and the primary key of the “one” table, but they don’t have to be. Or the name could try to describe the relationship. But the values need to match up.
  + Typical relationship diagram (PK denotes primary key, FK denotes foreign key):



* Exploring One-to-One Relationships
  + This is very unusual because you might as well just combine the two tables.
  + Can be done by having a foreign key in one of the table, or just use the same primary key in both tables.
  + Make sure you are thinking about your relationships both ways. You might think a relationship is one-to-one when going one direction, but when going the other direction, you may realize it is actually a one-to-many relationship.
* Exploring many-to-many relationships
  + Need to table to link the two tables together if they have a many-to-many relationship. This table is referred to as the join table, the junction table, the linking table, the bridging table, or a cross-referencing table.
  + (Refer to above section Describing many-to-many relationships)
  + No need to make a primary key for the linking table. The composite key consisting of the two rows is already a primary key.
  + Sometimes you might think a relationship is one-to-many when in reality it is a many-to-many. This may be caused by the fact that the relationship is not weighted equally. (Each row of Table A may correspond to many rows of Table B, but only a few rows of Table B may correspond to more than one row of Table A.)
* Referential Integrity and Relationship Rules
  + Relationship describes a rule. You can’t, for example, add a row in Table B that doesn’t correspond to a row that exists in Table A if there is a one-to-many relationship between Table A and B. (This is considered a referential constraint – a rule applied between 2 tables.)
  + Let’s say that Table A and B have a one-to-many relationship, and you want to add a row to Table B that corresponds to a row in Table A that you haven’t created yet. You have to create the row in Table A before Table B because the rows in Table B each needs to correspond to a valid row in Table A.
  + Referential integrity: data is valid and meaningful between your tables (as opposed to just valid within one particular table). This impacts adding/modifying rows (example provided above) and deleting rows.
  + Deleting rows: If you have Customer 367 with two Orders, and you delete Customer 367, then what happens? If you implement cascading delete, then the orders get immediately deleted as well. And if Orders corresponded to another table, then those corresponding rows get deleted too. Etc. If you implement cascading nullify (though this is rarely used), then you nullify (or set to null) all corresponding foreign key rows. But most common method is refusal: no action. Thus, you would need to delete the corresponding Orders before deleting the Customer.

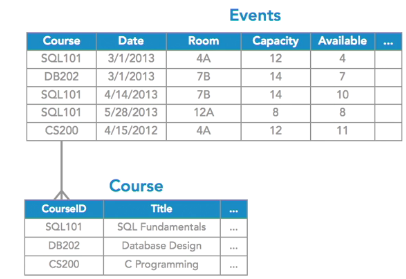
Database Modeling: Optimization

* Understanding Normalization
  + After planning out tables, we do database normalization: process of taking your database design and apply a set of formal rules called normal forms.
  + Step through them: First Normal Form (1NF) -> Second Normal Form (2NF) -> Third Normal Form (3NF).
  + Point is to make your database easier and reliable. May create a few additional tables in process, but still has minimum duplicate/redundant data. Benefits outweigh the costs by far. Everybody does this.
* First Normal Form
  + The First Normal Form says that each of your cells should contain just one value, and there should be no repeating groups.
  + Repeating groups are where you create new columns so that you can store multiple values for a single column. But this is a problem since you don’t know how many new columns you will end up needing. (It’s like creating individual variables instead of using an array in programming.)
  + Resolve by separating the repeating group into a new table and establishing a one-to-many relationship (or a many-to-many relationship with linking table if needed).
* Second Normal Form
  + You first must be in First Normal Form before you can start 2NF
  + A problem with 2NF can only occur if you are using a composite key.
  + 2NF states that any non-key field (the actual value in a particular column position for a particular row) should be dependent on the entire primary key.
  + In other words, take a look at all your non-key columns. If any non-key column is independent of any of the columns of the composite primary key, then you violated 2NF. Resolve this by creating another table containing two columns: the “partially independent column” and the column from the primary key the “partially independent column” depends on.
  + Example (course and date form the composite key, CourseTitle is only dependent on the course, not the date):

Violates 2NF:



Resolves 2NF violation:



* + This is needed because if you change an entry within the column that the “partially independent column” depends on, but you don’t change the “partially independent column”, then you got a conflict.
  + (Note, according to Wikipedia, this must be done for all candidate keys, or subsets of columns that could also form keys, instead of just the actual composite key.)
* Third Normal Form
  + No non-key field is dependent on any other non-key field.
  + 2NF asked “Can I figure out any of the values in a row from just part of the composite key”, whereas 3NF asks “can I figure out any of the values in a row from any of the values from another row”. (You shouldn’t be able to do that.)
  + Using 2NF’s example (shown above): A room always has a given capacity.
  + Fix this by pulling out a data into another table just like we did for 2NF. So for 2NF’s example, we would pull out Room and Capacity, leaving just one of them in the original table.
  + Another example is where you have the total price column be the product of the unit price column and the quantity column. Just get rid of total price.
  + Many databases offer you the option of defining a computed or calculated column. It’s not stored in a database, but is a convenient, read-only fiction. It is useful for cases like the above where you want a column calculated from performing operations on some other columns.
* Database Denormalization
  + (3NF is standard, but there are 4NF, 5NF, 6NF, and Boyce-Codd Normal Form)
  + Denormalization decision: the conscious choice of not normalizing a table into another table for the purpose of convenience and performance.
  + Example: if you have a fixed maximum number of phone numbers, you can create that number of columns of phone numbers. (But if you want to a flexible number, then you need to follow the procedure to first 1NF.)
  + Storing each value of an address (street, city, state, zip code) may seem to violate normalization rules, but it actually doesn’t since city/state aren’t completely dependent on the zip code.

Database Modeling: Querying

* Creating SQL queries
  + SELECT is the most common query. Format: SELECT *columns* FROM *table*; If you want to select multiple columns, separate them by commas. To select all columns, use the asterisk.
  + Add addition optional keyword to your query, WHERE: SELECT *columns* FROM *table* WHERE *condition*. Where clause restricts what is returned, only returning results where the condition is true.
  + Generally, SQL doesn’t care about case-sensitivity. But column/table names should match how you named them. Line breaks are okay. Semicolon at end.
  + If you want to specify which database the table comes from (e.g. when you have multiple tables with the same name), then replace *table* with *databasename.table*.
* Structuring the WHERE clause
  + Return all rows whose last name is Green: WHERE LastName = ‘Green’;
  + (String values in SQL are surrounded in single quotes. By default, string comparisons are case insensitive.)
  + Instead of equals, can use <, >, <=, >=, <> (not equal).
  + Use AND or OR instead of && or ||.
  + WHERE Department = ‘Marketing’ OR Department = ‘Sales’; can be replaced by: Department IN (‘Marketing’, ‘Sales’)
  + Instead of using equals sign, you can use LIKE to be more flexible in matching. Example of LIKE: WHERE LastName LIKE ‘Green%’. The percent sign indicates you can put any number characters after Green and make this return true. If you only want to allow one character, use underscore instead. LIKE can be less time efficient.
  + Use IS to check if something is null. Example: WHERE LastName IS NULL; Checking if not null: WHERE MiddleInitial IS NOT NULL;
* Sorting query results
  + Usually the order of the results returned is quite arbitrary.
  + Add an extra clause to the end: ORDER BY *ordering*. Can put a column name for *ordering*. By default, orders in ascending order. To make it descending, put the words DESC after *ordering*. Example: ORDER BY ListPrice DESC;
  + Picking multiple columns to sort by: separate column names by commas. Any “ties” in the ordering of a column (e.g. first column) will be broken by the next column (e.g. second column).
* Using aggregate functions
  + An aggregate (or grouping) function means a calculation performed on a set of data that returns a single value.
  + Replacing SELECT *columns* with SELECT COUNT (\*) will return the number of rows selected.
  + SELECT MAX(*column*) gets the max value based on the column specified. And there is an equivalent one for MIN. AVG for average.
  + SELECT SUM(*columns*) will sum up all the values in the column(s).
  + GROUP BY *grouping* can be used to display values based on each unique entry in *grouping*. (Put in a column for *grouping*.) Example: SELECT COUNT(\*), Color FROM Product GROUP BY Color would return a table with columns result and Color, with each row displaying the count (shown in result column) for each Color (shown in Color column).
* Joining tables
  + Use the JOIN keyword to select pieces of data from multiple tables instead of just single tables.
  + Example: SELECT FirstName, Location FROM Employee JOIN Department ON *HowToJoin*;
  + But you’ll get a name conflict if there are two columns across the two tables with the same name. Do this by prefixing the conflicting column name with the table it is from. (Example: Employee.FirstName) You can be very explicit if you want by prefixing all columns with the table they are from.
  + Need to also specify how the two tables need to join by using the ON keyword. Do it by setting two columns equal to each other (e.g. Employee.DepartmentID = Department.DepartmentID)
  + By default, you are doing inner join. As a result, any row that doesn’t have a corresponding row in the other table based on the matching columns (as specified in the ON clause) won’t be returned. For example, if Alice Bailey in the Employee table had a DepartmentID of null, she wouldn’t be returned because in the Department Table, there is no row with DepartmentID of null. Additionally, if no employee works in the Department with ID 2, then that department will never be returned. JOIN can be replaced with INNER JOIN if you want to be explicit.
  + If you want to include the rows with no matching rows in the other table, then use OUTER JOIN. Choose which table takes precedence using LEFT OUTER JOIN (table to the left of the JOIN keyword, or the table in the FROM clause, takes precedence) or RIGHT OUTER JOIN (table to the right of JOIN takes precedence). Will fill in null for the entries in the row with no corresponding value.
  + FULL OUTER JOIN includes all rows from both tables.
* Inserting, Updating, and Deleting
  + Format for inserting a single row into a table: INSERT INTO *table* (*column1*, *column2*, …) VALUES (*value1*, *value2*, …). This results in inserting the specified values into the specified columns of the specified table for a new row.’
  + Format for changing values: UPDATE *table* SET *column* = *value* WHERE *condition*. Example: UPDATE Employee SET Email = ‘hi@example.com’ WHERE EmployeeID = 734. Don’t use where clause if you want to apply the update to every row.
  + Deleting values: DELETE FROM *table* WHERE *condition*. This specifies which rows you want to delete. (Use UPDATE if you want to erase some values of a row.) If you don’t specify WHERE, then you delete the whole table.
  + Good practice for updating/deleting: first run it using a SELECT statement instead. This will return you the rows that you will be affecting with update or delete.
* The data definition language
  + Select, Insert, Update, Delete are the DML (Data Manipulation Language) part of SQL. But these words don’t alter the structure of the table. This can be done using the DDL (data definition language) part of SQL.
  + The three keywords of DDL: Create Alter and Drop.
  + Creating a table: CREATE TABLE *table* (*column\_definitions*). Example: CREATE TABLE Employee (EmployeeID INTEGER PRIMARY KEY, Name VARCHAR(35) NOT NULL, Department VARCHAR(30), Salary INTEGER). VARCHAR means Variable length Character field, or String. Number in parenthesis indicates max character length. NULL indicates it can be null, NOT NULL does the opposite.
  + Format for changing the definition: ALTER TABLE *table* *CHANGE*. Example: ALTER TABLE Employee ADD Email VARCHAR(100) adds a new column named email to the Employee table.
  + Destroying a table: DROP TABLE *table*
  + Third part of SQL: Data Control Keywords. These are Grant and Revoke. These keywords can grant or revoke permissions for people for a database.

Database Modeling: Indexing and Optimization

* Understanding Indexes
  + Works just like the index of an array, except now we are indexing the rows of a table.
  + Clustered index: pick a column as the Clustered Index, and the database will order the data in that table based on that column. This means the data on the physical disk is ordered this way. The Clustered Index is the primary index on a table. Most database management systems would by default make the primary key the Clustered Index. So when you use the clustered index in your where clause, you are doing an O(1) operation instead of O(n)
  + Full-Table Scan: scan each row of an unindexed column to check the column entry of a row is equal to the specified value. O(n) operation: inefficient.
  + Resolve this by creating a secondary (or Non-Clustered) index. This will create a new table with the secondary index as the sorted column and the primary key as the other column. This is very useful if you are going to use a lot of queries based on this column.
  + So why don’t we index every column? Reading is improved, but updating will be much more inefficient because everything needs to be maintained.
  + Indexing is not just “upfront” work determined only when you create the table and never touched afterwards. This is b/c you might not be able to tell the best Non-Clustered secondary indexes until later. Indexing is a trade-off: faster reads and slower writes and larger space. Indexing can be tweaked without breaking applications: doesn’t change actual data
* Understanding write conflicts
  + Write conflicts can occur if two people try to modify the table at the same time.
  + Example: we have a joint bank account with $10000 between Alice and Bob. They each see the balance, $10000, and each withdraw $1000, making it $9000. But it should be $8000.
  + This is known as a race condition: a conflict where two programming threads are trying to affect the same data.
  + Fix this by making the code atomic. Do this by adding the keywords BEGIN TRANSACTION before all the steps and then COMMIT at the end of all the steps. But this alone isn’t good enough. We also need locking to occur to prevent others from reading the table while the transaction is occurring.
  + Pessimistic Locking: we are pessimistic that transactions will conflict with each other, so as soon as the transaction starts, we lock the data it is referring to, and only unlock it once the transaction commits. If you keep using pessimistic locking, you’ll end of with a lot of people having to wait, or a lot of read errors
  + Optimistic Locking: allow the other threads to view the data and perform the operations. But if there is a conflict (i.e. the row has changed, resulting in a dirty read), then it will return an error and roll back to the beginning of the transaction.
  + The actual locking implementation (e.g. the keywords to do it, or is one type done by default) depends on the DBMS
* Understanding stored procedures and injection attacks
  + Stored procedure (SProc): just like a function/method. Syntax:

CREAT PROCEDURE *nameofprocedure*()

--Double dashes are comments

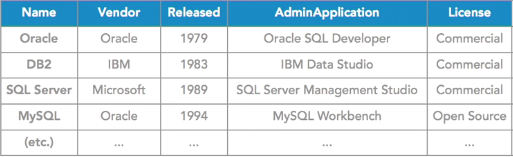
--LINES OF CODE

END

* + Then call the procedure: CALL *nameofprocedure*()
  + Stored Procedures can have parameters, as specified in the parenthesis. Syntax for parameter: IN *name* *type*. (IN indicates it is a parameter going into the function.)
  + Other than reusing code, stored procedures stay in the database and can be hopefully optimized by the database administrator. Also protects against SQL injection attack.
  + SQL injection attacks are when a malicious user tries to use the form and complete or create SQL commands to see sensitive data or modify/delete data.
  + Stored Procedures is one way of handling SQL injection attacks since they can’t be broken apart into several statements.

Database Options

* Desktop databases
  + Microsoft Access and Filemaker are the two most popular options.
  + Open source: Apache Base
  + These are aimed at power users, a step up from spreadsheets.
  + Pros: simple to install, easy to use, templates for starting, tools for setup, options for reporting the data
  + Cons: can’t accommodate many users, stored in a single file, won’t support very large data, can’t serve as a website database
* Relational database management systems



* + These systems are for enterprise level, or for many users with stringent demands
  + As databases get larger, you may need full-time database administrators.
  + Pricing of the databases vary greatly based on many factors. But even commercial databases may have free versions (look for the Express version).
* Object-based and XML-based databases
  + XML-based databases:



* + Use XQuery instead of XML to query the data.
  + If you have all XML, these are good options. But if you only have some XML, then use relational databases.
  + Oracle, DB2, SQL Server all have RDMBS XML support. (These have XQuery support.) For MySQL, you just store as text.
  + Object-oriented databases: Objective/DB, VelocityDB, Versant
  + Relational databases are still popular with object-oriented languages by using an object-relational mapping (ORM), which maps between objects in object-oriented language to regular relational database tables. Hibernate is an ORM for Java, Core Data for Objective-C, ActiveRecord for Ruby, NHibernate for C#/VB.NET.
  + Neither XML nor Object-based databases are as popular as relational databases.
* NoSQL Databases
  + “Not only SQL”. Lots of variety of databases in NoSQL. NoSQL just means that it is not a traditional relational databases.
  + Some databases: CouchDB, MongoDB, Apace Cassandra, Hypertable, HBase, Neo4j, BigTable, Riak, Project Voldemort, Redis, etc.
  + NoSQL Databases May include: not using SQL, not being table based, not relationship oriented, not ACID, no formal schema, oriented to web dev, oriented to large-scale deployment, often open source.
  + Document stores: stores documents, not rows and columns. Document means everything you store is a self-contained peace of data that describes its own schema. Examples: CouchDB and MongoDB
  + Key-value stores: everything stored as a key-value pair. Examples: MemcacheDB, Riak, and Project Voldemort. These database systems are designed to be fault-tolerant, distributed architecture, which means you can install them across multiple machines, and no one machine is a point of failure.
  + Graph Database: everything stored as a graph, or small connected nodes with relations. Examples: Neo4j, AllegroGraph, and DB2 NoSQL Graph Store
  + Reasons to Choose a NoSQL Database: flexible schema, vast amounts of data, value scaling over consistency/reliability.

Conclusion

* Final thoughts
  + Take a look at courses on the SQL language and DBMS like SQL Server, MySQL, and PostgreSQL.