Chapter 1: Conceptual Design

- 3 levels of abstraction: external (describes different parts to different users), logical (how data is perceived), physical (how data is stored)
  - Schema: logical structure
  - Instances: content at a point in time
- ER Diagrams: info about entities and relationships, integrity constraints
- Entity: Objects distinguishable from other objects
  - Attributes: Used to describe an entity. Each has a domain (ex. float, int, date, etc.)
  - Entity Set: Collection of similar entities.
- Keys: Used to distinguish an entity set, minimal set of 1+ attributes that can uniquely identify an entity in a set
  - Primary key: key chosen as principal means to identify entities in an entity set
  - Primary key attributes are underlined
- Relationship: Association between 2+ entities
  - Relationship set: collection of similar relationships
  - Degree/Arity: Number of entity sets in a relationship
- Cardinality ratio: number of relationships in the set an entity can participate in
  - 1-1: Entity in A is exclusively associated with 1 entity in B and vice-versa.
  - 1-many: Entity in A is associated with any number of entities in B, B is associated with 1 entity in A.
  - Many-1: Opposite of above.
  - Many-many: Entity in A associated with any number of entities in B and vice-versa
- Key constraints are shown with arrows in ER diagram
  - Arrow points towards the 1. Ex. A and B are 1-many, arrow goes from B to A
- Participation: Whether or not an entity participates in a relationship. Can be total or partial
  - Total participation is shown using a thick line from the entity to the relationship
- ISA relationship: If A ISA B, every A entity is a B entity
  - Used to group entities according to shared behaviour
  - Allows for more description of specific subclasses and restricts entities in a relationship
- Overlap constraints:
  - Disjoint: Superclass belongs to no more than 1 subclass
  - Overlapping: Subclasses may overlap
- Covering constraints:
  - Total: Superclass must belong to some subclass
  - Partial: Some superclass may not be in any subclass
- Weak entities can only be identified by considering the key of an owner entity
  - Owner set and weak entity must have a 1-many relationship
  - Weak must have total participation in the identifying relationship set (ie. it belongs to)
  - Weak entity sets and their identifying relationship sets are shown with thick lines around the relationship and entity
- Aggregation treats relationship set as an entity for participation in other relationships (dotted box around the aggregate)

Chapter 2: Relational Model

- Relational database: set of relations, each made of 2 parts
  - Schema: name of relation and name + domain of each attribute (ex. Student(*id*: string, *name*: string, *major*: string))
  - Instance: a table with rows (num rows = cardinality, each row is a tuple) and columns (num cols = arity, each column is an attribute)
  - Domain value: individual value in an instance
- Relational database schema: collection of schema in database
- Database instance: collection of instances of its relations

```
SELECT sid, name phone
FROM Students
WHERE major = "CPSC"
```

- Select: functions the same as R, specify what attributes are being selected. (SELECT * is all attributes)
- From: specifies the relation
- Where: acts the same as filter in R.

```
CREATE TABLE Student
    (sid INTEGER,
    name CHAR(20),
    major CHAR(4))
```

- Create table: Creates relation with whatever name we give
- Statements on the left are the attributes, statements on the right are information on what type of data is stored in each attribute

```
DROP TABLE Student
```

- Deletes Student's schema info and tuples (like drop transformation)

```
ALTER TABLE Student
    ADD COLUMN gpa REAL;
```

- Alter table: Edits the table
- Add column: adds a column/attribute with the following name

```
INSERT
INTO Student(sid, name, major)
VALUES (82518564, "N.␣Chang", "ISCI")
```

- Insert: Can manually insert values into table
- Into: Specify the relation and its attributes
- Values: Specify the tuple to be added

```
DELETE
FROM Student
WHERE name = "Nathan"
```

- Delete: Drop transformation (deletes tuples satisfying a condition)
- In general, specify the type transformation, specify the table, write out the transformation
- Integrity constraint: condition that must be true for any instance of the database (specified when schema is defined, checked when relations are modified)
  - You can make a key for a relation if no distinct tuples can have the same values for all attributes in a key and no subset of the key is a key itself
  - One of the candidate keys is chosen by the DBA to be a primary key (same line for single attribute keys, its own statement for multiple attributes)

```
CREATE TABLE Student
    (sid INTEGER PRIMARY KEY,
    name CHAR(20),
    major CHAR(4))
```

- Primary key cannot be null and values must be unique
- Other keys are specified using the UNIQUE constant (values for a group of attributes if not null, but they can be null)
- Key constraints are checked when new values are inserted or values are modified
- Denoted by underline

```
CREATE TABLE Student
    (sid INTEGER,
    dept CHAR(4),
    courseNum CHAR(3),
    mark CHAR(2),
    PRIMARY KEY(sid, dept, courseNum),
    UNIQUE(dept, courseNum, mark))
```

- Foreign key: set of attributes in one relation used to reference a tuple in another
  - Foreign keys must correspond to the primary key of the other relation
  - Referential integrity: All foreign keys referencing existing entities
  - ex. Grade(*sid, dept, courseNum, grade*), sid is a foreign key referring to Student, dept and courseNum refer to Course
  - Bolded and underlined in relation (bolded = foreign, underline = part of the relation's primary key)

```
CREATE TABLE Student
    (sid INTEGER,
    dept CHAR(4),
    courseNum CHAR(3),
    mark CHAR(2),
    PRIMARY KEY(sid, dept, courseNum),
    FOREIGN KEY (sid) REFERENCES
        Student,
    FOREIGN KEY (dept, courseNum)
        REFERENCES Course(dept, cnum))
```

- When referencing attributes with different names between the relation and the relation being referenced from, specify which of the foreign relation's attributes are being referenced

```
CREATE TABLE Student
    (sid INTEGER,
    dept CHAR(4),
    courseNum CHAR(3),
    mark CHAR(2),
    PRIMARY KEY(sid, dept, courseNum),
    FOREIGN KEY (sid) REFERENCES
        Student
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    FOREIGN KEY (dept, courseNum)
        REFERENCES Course(dept, cnum)
        ON DELETE SET DEFAULT
        ON UPDATE CASCADE)
```

- SQL supports 4 options on delete/update:
  - Default is NO ACTION (rejects delete/update)
  - CASCADE: Updates/deletes all tuples that refer to the updated/deleted tuple
  - SET NULL/SET DEFAULT: Referencing tuple value is set to the default foreign key value/null respectively
- Weak entity sets and its identifying relationship are translated into a single table (like many-1)
  - Primary key is the owner's primary key + weak entity's partial key
  - When owner is deleted, ALL owned weak entities must also be deleted

Chapter 3: SQL Plus

- Here are some common domain types in SQL:
  - char(n): Fixed length string with length n
  - varchar(n): Variable length character strings with max length n
  - int: Integer
  - smallint: Small integer
  - numeric(p, d): Fixed point number with precision p digits with d decimal places
  - real, double precision: Floating point and double-precision floating point numbers, machine-dependent person
  - float(n): Floating point number with precision of at least n digits
- Date/Time Types:
  - date: Date with 4 digit year, month, date (2001-7-27)

- time: Time of day in hours, minutes, seconds (09:00:30.75)
- timestamp: date plus time of day (2001-7-27 09:00:30.75)
- Interval: period of time

- You can alias relations to create conditions specified in the WHERE clause of an SQL query

```
SELECT Character, Name
FROM StarsIn s, MovieStar m
WHERE s.StarID = m.StarID
```

- Joins are cross products (produce every possible combination of tuples)
- DISTINCT is used to only return unique tuples
  - Not equal is denoted by ¡¿ in SQL!
  - Can also use ¿, ¡, =, ¡=, ¿=
- You can rename relations and attributes using AS

```
SELECT Title, StarID AS ID
FROM StarsIn S, Movie M
WHERE M.MovieID = S.MovieID
```

- Below is an example of string matching:

```
SELECT DISTINCT snum
FROM enrolled
WHERE cname LIKE '%System%'
```

  - LIKE is used for string matching, _ is any 1 character and % is 0+ arbitrary characters
  - You can use other operations like ——, changing case, string length, and substrings
- ORDER BY _ is used to order the tuples in asc (default) or desc order based on a specified field
  - A OR B refers to union all
  - A AND B refers to intersect
  - A EXCEPT B keeps results found in A and not B
- Suppose a tuple occurs m times in R and n times in S, it occurs:
  - m + n times in R OR S
  - min(m, n) times in R AND S
  - max(0, m-n) times in R EXCEPT S

```
SELECT StarID
FROM Movie M, StarsIn S
WHERE M.MovieID = S.MovieID AND year
     = 1944
UNION
SELECT StarID
FROM Movie M, StarsIn S
WHERE M.MovieID = S.MovieID AND year
     = 1974

SELECT StarID
FROM Movie M, StarsIn S
WHERE M.MovieID = S.MovieID AND year
     = 1944
INTERSECT
SELECT StarID
FROM Movie M, StarsIn S
WHERE M.MovieID = S.MovieID AND year
     = 1974

SELECT snum
FROM enrolled e
WHERE cname = Operating System
    Design
EXCEPT
SELECT snum
FROM enrolled e
WHERE cname = 'Database␣Systems'
```

- Nested queries: query with a query within it
  - SELECT, FROM, WHERE, or HAVING clauses can contain SQL queries
  - IN and NOT IN are used to query based on an input query

```
SELECT M.StarID, M.Name
FROM MovieStar M
WHERE M.Gender = female AND
M.StarID IN (SELECT S.StarID
    FROM StarsIn S
    WHERE MovieID = 28)
```

  - EXISTS/NOT EXISTS: returns true/false if the input set isn't empty
  - UNIQUE: returns true if there are no duplicates
  - You can also use set operators for nested queries
- Division is used to express queries that include "for all" or "for every"
- Method 1 (with EXCEPT):

```
SELECT sname
FROM Student S
WHERE NOT EXISTS
    ((SELECT C.name
    FROM Class C)
    EXCEPT
        (SELECT E.cname
        FROM Enrolled E
        WHERE E.snum=S.snum))
```

- Method 2 (with EXCEPT):

```
SELECT sname
FROM Student S
WHERE NOT EXISTS (
    SELECT C.name
    FROM Class C
    WHERE NOT EXISTS (SELECT E.snum
        FROM Enrolled E
        WHERE C.name= E.cname AND E.
            snum=S.snum))
```

- These aggregate operators operate on a multiset of values and return a value:
  - AVG: average value
  - MIN: minimum value
  - MAX: maximum value
  - SUM: sum of values
  - COUNT: number of values
- These versions eliminate duplicates before applying the operation to the attribute
  - COUNT(DISTINCT A)
  - SUM(DISTINCT A)
  - AVG(DISTINCT A)
- GROUP BY divides the tuple into groups, HAVING applies an aggregate
- Consider the example: Find the age of the youngest student who is at least 19, for each major with at least 2 such students.

```
SELECT major, MIN(age)
FROM Student
WHERE age >= 19
GROUP BY major
HAVING COUNT(*) > 1
```

- In general:

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
ORDER BY target-list
```

  - The target-list contains attribute names and terms with aggregate operations
  - These attributes must also be in grouping-list
    - → Each answer tuple corresponds to a group with a group being a set of tuples with the same value for all attributes in grouping-list
    - → Selected attributes must have a single value per group
  - Attributes in group-qualification are either in grouping-list or are arguments to an aggregate operator
- Views are used to hide data from some users, make queries easier, and modulate the database
  - They can be used to present necessary info, while hiding details in underlying relations
  - View updates must occur at base tables

```
Create View <view name> <attributes
    in view> AS <view definition>
```

- Consider having tables Course(<u>Course#</u>, title, dept) and Enrolled(<u>Course#, sid</u>, mark)
  - This view yields the dept, course#, and marks for those courses where someone failed

```
CREATE VIEW CourseWithFails(dept,
    course#, mark) AS
    SELECT C.dept, C.course#, mark
    FROM Course C, Enrolled E
    WHERE C.course# = E.course# AND
        mark<50
```

- Dropping a view does not affect any tuples in the underlying relations

```
DROP VIEW <view name>
```

  - DROP TABLE X RESTRICT: Drops table unless there's a view on it
  - DROP TABLE X CASCADE: Drops table and drops any view referencing it
- Tuples may have null values, you can filter these with IS NULL/IS NOT NULL
- Natural JOIN makes it so columns with the same name will only appear once (ie. returns the cross product of the two tables):
  - Associated columns must have 1 or more pairs of identically named columns
  - Columns must be the same data type
  - Don't use ON clause

```
SELECT *
FROM student s natural join enrolled
     e
```

- You need an outer join to get all students. There are several join types:
  - Inner join: default - only includes matches
  - Left outer join: Includes all tuples from the right hand relation
  - Right outer join: Includes all tuples from the left hand relation
  - Full outer join: Includes all tuples from both relations

```
SELECT *
FROM Student S NATURAL LEFT OUTER
    JOIN Enrolled E
```

- Here's the SQL from an insertion with values selected from another table:

```
INSERT INTO Enrolled
    SELECT 51135593, name
    FROM Class
    WHERE fid = 90873519
```

- SQL for deleting tuples satisfying a specific conditions:

```
DELETE FROM Student
WHERE name = Smith
```

- You can use CHECK conditional-expression to specify constraints over a single table
- You can define an assertion to apply a constraint over multiple relations:

```
CREATE ASSERTION totalEmployment
CHECK (NOT EXISTS ((SELECT StarID
    FROM MovieStar)
        EXCEPT
        (SELECT StarID FROM StarsIn)
            ))
```

- Trigger is a procedure that starts automatically if specificed changes occur to the DBMS. A trigger has 3 processes

- Event (activates trigger)
- Condition (tests whether trigger should run)
- Action (procedure executed when trigger runs)
- An active database is a database with triggers

Chapter 4: Data Warehousing

- OLAP used to perform grouping and aggregation + other complex ops. Best way is to use a multi-dimensional model
  - Set of numerical measures - quantities that are important
  - Fact table: relates dimensions to a measure via foreign keys (there can be multiple)
  - Set of dimensions - entities on which the measure depend on (ex. location, date)
  - Dimension table: table for a dimension
- Star schema: One table for the fact, and one per dimension
- Example Star Schema fact table and dimension tables:

```
AllSales(storeID, itemID, custID
    , sales) /*fact table*/
Store(storeID, city, county,
    state) /*dimension tables*/
Item(itemID, category, color)
Customer(custID, came, gender,
    age)
```

- A full star join joins the fact table and all of its dimensions (see below example code)
- A star join has less than all dimensions

```
SELECT *
    FROM AllSales F, Store S, Item I
    , Customer C
    WHERE F.storeID = S.storeID and
    F.itemID = I.itemID and
    F.custID = C.custID;
```

- In a snowflake schema, each dimension is a set of tables with 1 table per level of hierarchy per dimension
  - TIMES would be split into TIMES(timeid, date), DWEEK(date, week), DMONTH(date, month)
  - Has more foreign keys and joins but is useful if dimension table is big in size or expecting lots of updates

OLAP Operations

- Roll-up: summarizes data by changing hierarchy of a dimension of dimension reduction
- Example 1 (hierarchy): Use roll-up on total sales by store, item, and customer to find total sales by item and customer for each county.

```
SELECT storeID, itemID,
    custID, SUM(sales)
FROM AllSales F
GROUP BY storeID, itemID,
    custID

SELECT county, itemID,
    custID, SUM(sales)
FROM AllSales F, Store S
WHERE F.storeID = S.storeID
GROUP BY county, itemID,
    custID /*this is the
    roll up/
```

- Example 2 (hierarchy): Use roll-up on total sales by store, item, and customer to find total sales by item, age, and county

```
SELECT county, itemID,
    custID, SUM(sales)
FROM AllSales F, Store S
WHERE F.storeID = S.storeID
GROUP BY county, itemID,
    custID

SELECT county, itemID,
    custID, SUM(sales)
FROM AllSales F, Store S,
    Customer C
WHERE F.storeID = S.storeID
    and F.custID = C.custID
GROUP BY county, itemID, age
     /*this is the roll up/
```

- Example 3 (dimension): Use roll-up on total sales by item, age, and county to find total sales by item for each county.

```
SELECT county, itemID, age,
    SUM(sales)
FROM AllSales F, Store S,
    Customer C
WHERE F.storeID = S.storeID
    and F.custID = C.custID
GROUP BY county, itemID, age

SELECT county, itemID, SUM(
    sales)
FROM AllSales F, Store S,
WHERE F.storeID = S.storeID
GROUP BY county, itemID
```

- Drill-down (reverse of roll-up): higher level to lower level summary (more detailed). Adding new dimensions.
- Example 1 (hierarchy): Use drill-down on total sales by item and age for each county to find total sales by item and age for each city.

```
SELECT county, itemID, age,
    SUM(sales)
FROM AllSales F, Store S,
    Customer C
WHERE F.storeID = S.storeID
    AND F.custID = C.custID
GROUP BY county, itemID, age

SELECT city, itemID, age,
    SUM (sales)
FROM AllSales F, Store S,
    Customer C
WHERE F.storeID = S.storeID
    AND F.custID = C.custID
GROUP BY city, itemID, age
    /*this is the drill down
    /
```

- Example 2 (dimension): Use drill-down on total sales by item and county to find total sales by item and age for each county.

```
SELECT county, itemID, SUM(
    sales)
FROM AllSales F, Store S
WHERE F.storeID = S.storeID
GROUP BY county, itemID

SELECT city, itemID, age,
    SUM (sales)
FROM AllSales F, Store S,
    Customer C
WHERE F.storeID = S.storeID
    AND F.custID = C.custID
GROUP BY county, itemID, age
     /*this is the drill
    down/
```

- Slicing: Slice by picking a specific value of ONE of the dimensions
- Example 1: Use slicing on total sales by item and age for each county to find total sales by item and age in Santa Clara.

```
SELECT county, itemID, age,
    SUM(sales)
FROM AllSales F, Store S,
    Customer C
WHERE F.storeID = S.storeID
    AND F.custID = C.custID
GROUP BY county, itemID, age

SELECT itemID, age, SUM (
    sales)
FROM AllSales F, Store S,
    Customer C
WHERE F.storeID = S.storeID
    AND F.custID = C.custID
    AND S.county = "Santa␣
    Clara" /*this is the
    slicing/
GROUP BY itemID, age
```

- Example 2: Use slicing on total sales by item and age for each county to find total sales by county and age for T-shirts.

```
SELECT county, itemID, age,
    SUM(sales)
FROM AllSales F, Store S,
    Customer C
WHERE F.storeID = S.storeID
    AND F.custID = C.custID
GROUP BY county, itemID, age

SELECT itemID, age, SUM (
    sales)
FROM AllSales F, Store S,
    Customer C, Item I
WHERE F.storeID = S.storeID
    AND F.custID = C.custID
    AND F.itemID = I.itemID
    AND category = "Tshit"
    /*slicing*/
GROUP BY county, age
```

- Dicing: generate sub-cube by picking values for MULTIPLE dimensions
- Example: Use dicing on total sales by item, age, and city to find total sales by age, category, and city for red items in California (CA).

```
SELECT city, itemID, age,
    SUM(sales)
FROM AllSales F, Store S,
    Customer C
WHERE F.storeID = S.storeID
    AND F.custID = C.custID
GROUP BY city, itemID, age

SELECT category, city, age,
    SUM (sales)
FROM AllSales F, Store S,
    Customer C, Item I
WHERE F.storeID = S.storeID
    AND F.custID = C.custID
    AND F.itemID = I.itemID
    AND color = "red" AND
    state = "CA" /*dicing*/
GROUP BY category, city, age
```

- Pivoting: rotate cube to provide an alternative presentation of the data
- Example: From total sales by store and customer, pivot to find total sales by item and store.

```
SELECT storeID, custID, SUM(
    sales)
FROM AllSales
GROUP BY storeID, custID

SELECT storeID, itemID, SUM
    (sales)
FROM AllSales
GROUP BY storeID, itemID /*
    pivoting (goes from
    custID to itemID)*/
```

- Data cube: k-dimensional object containing fact data and dimensions.

- Individual blocks are called cells, each can be uniquely identified with dimensions.
- With 2 x 2 x 2 tuples, there are 3 x 3 x 3 combos (each dimension can be referred to as "All").
- Dense cubes have data for all combinations of attributes, sparse cubes are missing at least 1.
- Sparse cube size = dense cube size x sparsity factor
- In a 3-dimensional cube (ex. store, item, customer), 8 SQL queries are required to compute all cells
- A k-D cube can be represented as a series of (k-1)-D cubes

- Operation with CUBE:

```
SELECT dimension-attrs,
    aggregates
FROM tables
WHERE conditions
GROUP BY dimension-attrs
    WITH CUBE

SELECT storeID, itemID,
    custID, SUM(sales)
FROM AllSales
GROUP BY storeID, itemID
    custID WITH CUBE
```

- Materialized view: View whose tuples are store in the database. Materialize a small set of views to answer most queries effectively
- When computing costs to query, consider number of tuples to look at. Aim is to minimize cost.
- Example: Calculate benefit of materializing a view. What is the saving of S, I relative to S, I, C, S?
  S, I, C = 6M, S, I = 0.8M, S = 0.01M
  BS, I = 5.2M, BS = 0, BI = 5.2M, B = 0
  B(S, I, S, I, C S) = 5.2M * 2
- Find the best k views to materialize.
  After calculating benefit, you'll find that S, I has the largest (materialize it). Re-calculate, then you'll see C has largest - materialize that as well.

Chapter 5: Non-Relational DBs

- Here are some characteristics of Relational Databases (and SQL)
  - Great for transactional data (strict transactional requirements, ACID) - queries are flexible and independent of applications that depend on it
  - Transactional Data: Data contains information from transactions
  - Stable schema
  - Vertical scaling
  - Limited replication
  - Partitioning
  - Rapid insert/update/delete
- ACID properties help guarantee data visibility
  - Atomicity - all of nothing
  - Consistency - no incomplete transactions
  - Isolation - each transaction is independent and only uses data from complete transactions
  - Durability - record of transactions is kept for backup and logging
- Semi-structured data: Data has structure but not tabular. Data may have different attributes despite being the same "type" (ex. JSON, XML, etc.)
- Unstructured data: Data not organized in a way that is conducive to processing (ex. blog post, video, etc.)
- Big data: Contains volume, variety, velocity, and veracity
  - To deal with big data, schema flexibility is needed
  - Must be processed quickly - parallelization may be a problem with relational engines
  - Typically requires more read and aggregation queries (rather than update and delete)
- Scalability: Ability to handle increasing amounts of data.
  - Two types - vertical and horizontal
  - NoSQL does not necessarily mean distributed
  - Vertical scaling involves upgrading hardware (finite limit)
    → Everything is on 1 machine. This ensures consistency but there is only 1 point of failure
    → Setup is easier
  - Horizontal scaling involves adding more machines to the cluster. This creates a distributed database and can be scaled as needed.
    → Two ways to distribute data. Replicate same data across many servers (replication) or partition data into chunks and keep chunks on different servers (sharding)
    → May be more complicated to set up due to distributed nature of the machines
    → Data consistency across machines is an issue
- There are two types of replication: primary-replica and peer to peer
- In primary-replica, one "primary" machine manages others.
  - Only the primary machine processes writes, all machines can process reads
  - Possible data synchronization problems and the primary machine becomes a bottleneck which may impact performance
  - If the primary fails, the replicas choose a new machine
- In peer to peer, all machines can perform read-/write operations
  - Synchronizing changes can cause issues as writes can happen on different machines simultaneously. This may lead to the same query returning different results (tradeoff from horizontal scaling)
  - No bottleneck
- Sharding involves the partitioning of data and storage on different machines (shards)
- Note: Sharding is distribution across multiple servers, partitioning is across 1 server
  - Systems use different policies to determine where data should go. Different systems have different replication policies, but it's common to replicate each shard twice
  - Re-sharding may happen as time passes, during this time, the number of shards may be increased or data may be partitioned differently
  - If a shard gets full, it may split automatically
  - Acts like a single DB and usually has an index to track where data is. This allows users to move data closer to where it is needed but creates additional complexity
  - There may be issues with data consistency and joins across tables become more difficult
- Compare and contrast: transactional data vs big data
  - Transactional Data:
    → Schema: stable
    → Scaling: vertical
    → Transactions: stringent
    → Replication: limited
    → Fragmentation: partitioning
    → Processing; insert, update, delete
  - Big Data:
    → Schema: flexible
    → Scaling: horizontal
    → Transactions: relaxed
    → Replication: extensive
    → Fragmentation: sharding
    → Processing: insert, update, aggregate
- CAP theorem (consistency, availability, partition tolerance) states that in a distributed data storage system, you can only meet two out of three of these reqs
  - As system is distributed, partition tolerance is required. Trade-off is between consistency and availability
- NoSQL (not only SQL) DBs use BASE rather than acid. There are 4 types of NoSQL DBs:
  - Column store
  - Document store
  - Key value store
  - Graph database
  - Hybrid DBs use several different models within 1 database
- BASE stands for:
  - Basically Available: Guarantees the availability of the data
  - Soft state: State of the system can change over time
  - Eventual consistency: System will eventually become consistent
- Pros of NoSQL DBs:
  - Data stored in a manner similar to programming languages (easier for people with no DB experience)
  - Distributed info storage
  - Can be dynamically scaled (flexible scaling) and no set schema
  - Potentially cheaper than relational DB and typically open source
- Cons of NoSQL DBs:
  - No guarantee of consistency. Weaker consistency guarantees compared to a relational DB.
  - Administration can be difficult across multiple servers
  - No data access standard or structure. Different NoSQL DBs have different query languages and may operate differently
- Column store stores multiple versions of a value differentiated by timestamps. This allows for fast reading and writing as no searching/modifying is needed
  - Quick vocab lesson:
    → Table: The name of the table that holds the data (similar usage to relational databases)
    → Key: Used to identify individual rows (similar usage to relational databases)
    → Column Family: A group of columns referred to by a common name
    → Column: An individual column. The name of a column is unique within the column family. Each column has a single unstructured value.
    → Each row has the same number of column families but the number of columns inside each family can vary
  - Need table name, key value, column family, column name, and timestamp (optional) to identify a specific value
  - Supports a high volume of data and flexible schema
  - Columns within a family are stored together, but families are stored separately
  - Better suited for OLAP queries
- Document store is a collection of documents, eash using JSON, BSON, and/or XML to store data
  - All docs in a coll use the same format
  - Schema is adaptable as not all docs hacve to use the same schema
  - In all JSON docs, there's a unique identifier that can be searched (name values can be searched as well). More on JSON in MongoDB
- Key value store is basically a has table
  - Find and modify operations are well supported and very fast
  - Key can be any identifier and the value is most likely an object
  - Query speed is higher than a relational DB and it supports mass storage and has high concurrency
  - Great for situations where you only need has operations, such as:
    → Get: returns assocaited value based on given key
    → Put: stores a key-value pair into the DB. If

the key already exists, it replaces the associated value
  - → Delete: deletes a key-value pair
  - ○ Do not use if you need to find values based on a non-key value (ie. does not support SQL) or if the underlying data model is complex
  - ○ Supports both structured and unstructured data and no limit on the size of the data for the values
- Graph DBs consist of nodes/vertices and edges/links
  - ○ Each node represents data. Edges between nodes indicate a relationship between the nodes (directional)
  - ○ Properties are info associated with nodes and edges
  - ○ Graphs can be directed or undirected No standardized so terminology and structure can vary
  - ○ Good for situations where you are interested in relationships
  - ○ Nodes/edges/properties can be added at any time
- Property graphs are the most common type of graph DB. They are directed
  - ○ Vertex labels are a collection of vertices (like how entity sets store entities)
  - ○ Edge labels are a collection of edges
  - ○ Vertices and edges can have different properties
  - ○ Common for transactional application
- In triple store (also called RDF), each property is a vertex
  - ○ Commonly used in analytical applications
  - ○ Good for interconnected data (supports ontology)
- Data lakes are stores of data that can support structured, semi-structured, and unstructured data
  - ○ Used for analytical purposes
  - ○ No need to pre-define a schema. Data can be stored in its original format
- Compare and contrast DB vs data warehouse vs data lake
  - ○ Database:
    - → Workload - transactional
    - → Data type - structured and semi-structured
    - → Schema flexibility - rigid or flexible depending on type
    - → Data freshness - real time
    - → Data analytics approach - limited
  - ○ Data warehouse:
    - → Workload - analytical
    - → Data type - structured and semi-structured
    - → Schema flexibility - pre-defined and fixed
    - → Data freshness - depends on when files were last refreshed/when ETL was last run
    - → Data analytics approach - top down
  - ○ Data lake:
    - → Workload - analytical
    - → Data type - structured, semi-structured, and unstructured
    - → Schema flexibility - none required (schema on read)
    - → Data freshness - depends on when files were last refreshed
    - → Data analytics approach - bottom up

Chapter 6: MongoDB

- MongoDB is a document DB that stores info as JSON objects! Each JSON object is a document, a group of documents is a collection
  - ○ Each document has an _id value and a max size of 16mb
- Sample JSON Array (note the _ is from verbatim)

```
{
    \_id": 507f1f77bcf86cd799439011,
    \firstName": \Perry",
    \lastName": \Platypus",
    \jobs": [\family pet", \secret agent"],
    \address": [
        { \buildingNo": 2366,
        \street": \Main Mall",
        \city": \Vancouver",
        "province": \BC"},
        { \buildingNo": 6245,
        \street": \Agronomy Rd",
        \city": \Vancouver",
        "province": \BC"}
    ]
}
```

- A document has 0 to n embedded documents - this keeps data tgt resulting in fewer queries and update statements
- DBrefs require a specific order - depending on driver, the behaviour may differ

```
{
\_id": 345,
// other fields
\address": {
"$ref" : "address", collection name
"$id" : ObjectId("123"), id of the document
"$db" : "users" // database name
    }
}
```

- Here's a compare and contrast between relational DBs and MongoDB:
  - ○ Database - Database
  - ○ Table/View - Collection
  - ○ Row - Document
  - ○ Column - Field
  - ○ Join - Embedded Doc
  - ○ Foreign Key - Reference
- MongoDB also follows the 4 basic operations: Create, Read, Update, Delete
- In MongoDB, inserting into a nonexistent collection creates it. All docs must have a unique identifier, if no _id is provided, one is assigned upon insertion

```
db.insert_one(
{\_id":10, \item": "canvas",
\qty": 100, \tags": ["cotton"],
    "size": { \h": 28, \w": 35.5,
    \uom": "cm" } }
)


db.insert_many(
{\item": "canvas", \qty": 100, \tags":
["cotton"], \size": { \h": 28, \w": 35.5,
\uom": "cm" } },
    {\item": "flannel", \qty": 100, \tags":
["cotton"], \size": { \h": 22, \w": 27,
\uom": "cm" } }
)
```

- In SQL, we have SELECT and FROM. In MongoDB, we have .find()

```
db.collection_name.find({<field>:<value>,
({<field>:<value>})
```

- You can use $in or $or to replicate OR
- Comparison:

```
SELECT *,
FROM Inventory,
WHERE name = notebook and qty > 20

SELECT *
FROM Inventory
WHERE name = notebook OR name =
    paper

SELECT *
FROM Inventory
WHERE name = notebook OR qty > 20

db.find({"name": \notebook", \qty": {"$gt":
30}})

db.find({"name": {"$in": \notebook",
\paper"}})

db.find({"$or": [{"name": "notebook"},
{"qty": {"$gt": 20}}]})
```

- Find functions return a cursor that can be used to iterate over the results

```
cur = db.find({"name":
    {"$in": \notebook", \paper"}})

for c in cur:
    print(c)
```

- In Mongo, there are 2 versions of update:
  - ○ update_one(): Updates the first document matching the condition
  - ○ update_many(): Updates many documents at once

```
UPDATE Inventory
SET qty = 5
WHERE name = notebook

db.update_one(
    {\name": \notebook"},
    {"$set": {"qty": 5}}
)
```

- In Mongo, there are 2 versions of delete:
  - ○ delete_one(): Deletes the first document matching the condition
  - ○ delete_many(): Deletes all docs that match the condition

```
db.collection_name.delete_many(
{condition})
```

Chapter 7: Data Mining

- Data mining: Exploration and analysis of data to discover valid, novel, possibly useful, and understandable patterns in data
- Techniques include supervised learning (classification and regression), unsupervised learning (clustering), dependency modeling (association, correlation, etc.), outliers, and trend analysis
- Rules are valid if support is above min support and confidence is above min confidence
  - ○ Support measures if items appear together a lot of time. A rule X -¿ Y holds with support sup if sup% of all transactions contain X and Y
  - ○ Confidence measures which which items suggest others will be there. A rule X -¿ Y holds confidence conf% if conf% of transactions that contain X also contain Y
- Frequent itemset: Set of items with AT LEAST minimum support
- Apriori algorithm: Each subset of a frequent itemset must also be a frequent itemset. Increase in size with successive rounds until no more can be made
- All of the itemsets found are frequent itemsets
- Example: Finding frequent itemsets given: T1 - apple, dates, rice, corn T2 - corn, dates, tuna T3 - apple, corn, dates, tuna T4 - corn, tuna *min support is 50
  Itemsets of size 1: apple = 2/4, corn = 4/4, dates = 3/4, tuna = 3/4. rice = 1/4 and is eliminated
  Find frequent itemsets of size 2: apple, corn = 2/4, apple, dates = 2/4, corn, dates = 3/4, corn, tuna

= 3/4, dates, tuna = 2/4. can eliminate everything with rice, apple, tuna has 1/4 and is also eliminated.
Find frequent itemsets of size 3: apple, corn, dates = 2/4, corn, dates, tuna = 2/4.
No valid itemsets of size 4.
Answer is union of all valid frequent itemsets!

- Clustering is a type of unsupervised learning (ie. data does not need to be labeled)
- When determining cluster quality, can consider: intra-class similarity (points in a cluster are close to each other), inter-class dissimilarity (points in different clusters are far from each other), and size similarity (clusters have similar size)
- k-means clustering algorithm:
  - Choose centroid to act as center of clusters, repeat until answer stabilizes
  - Cluster assignment - determine which of the k centroids its closest to and put it in the cluster of that centroid
  - Move centroid - average points in each cluster to get a new centroid
- if any points change clusters, repeat again!

Chapter 8: Data Privacy

- Encryption protects data by scrambling it according to an algorithm
  - End-to-end encryption makes it so only you and the person you're talking to can decrypt the message
- VPNs allow you to encrypt and disguise internet traffic by routing it through another computer first
- Cookies are small files used by a web browser to store information
- Anonymizing data takes away identifiers that would identify data as belonging to a particular person
- RTBF: ability to have private info removed from searches and directories under certain circumstances